Mido Documentation

Release 1.2.8

Ole Martin Bjørndalen

Jun 30, 2017

Contents

Version 1.2.8

Mido is a library for working with MIDI messages and ports. It's designed to be as straight forward and Pythonic as possible:

```
>>> import mido
>>> msg = mido.Message('note_on', note=60)
>>> msg.note
60
>>> msg.bytes()
[144, 60, 64]
>>> msg.copy(channel=2)
<message note_on channel=2 note=60 velocity=64 time=0>
```

```
with mido.open_input('LinnStrument') as inport:
    for msg in inport:
        print(msg)
```

```
port = mido.open_output()
port.send(msg)
```

```
mid = mido.MidiFile('song.mid')
for msg in mid.play():
    port.send(msg)
```

Mido is short for MIDI objects.

CHAPTER 1

Source code

https://github.com/olemb/mido/

CHAPTER 2

About This Document

This document is available at https://mido.readthedocs.io/

To build locally:

```
python setup.py docs
```

This requires Sphinx. The resulting files can be found in `docs/_build/`.

CHAPTER 3

Contents

Changes

(See *Roadmap* for future plans.)

Release History

1.2.8 (2017-06-30)

- bugfix: nonblocking receive was broken for RtMidi IO ports. (Reported by Chris Apple, issue #99.)
- bugfix: `IOPort.poll()` would block if another thread was waiting for `receive()`. Fixed the problem by removing the lock, which was never needed in the first place as the embedded input port does its own locking.

1.2.7 (2017-05-31)

- added max length when reading message from a MIDI file. This prevents Python from running out of memory when reading a corrupt file. Instead it will now raise an `IOError` with a descriptive error message. (Implented by Curtis Hawthorne, pull request #95.)
- removed dependency on `python-rtmidi` from tests. (Reported by Josue Ortega, issue #96.)

1.2.6 (2017-05-04)

- bugfix: Sending sysex with Pygame in Python 3 failed with `"TypeError: array() argument 1 must be a unicode character, not byte"`. (Reported by Harry Williamson.)
- now handles `sequence_number` and `midi_port` messages with 0 data bytes. These are incorrect but can occur in rare cases. See `mido/midifiles/test_midifiles.py` for more. (Reported by Gilthans (issue #42) and hyst329 (issue #93)).

1.2.5 (2017-04-28)

- bugfix: RtMidi backend ignored `api` argument. (Fix by Tom Feist, pull request #91.)

1.2.4 (2017-03-19)

- fixed outdated python-rtmidi install instructions. (Reported by Christopher Arndt, issue #87.)

1.2.3 (2017-03-14)

- typo and incorrect links in docs fixed by Michael (miketwo) (pull requests #84 and #85).

1.2.2 (2017-03-14)

- bugfix: sysex data was broken in string format encoding and decoding. The data was encoded with spaces ('data=(1, 2, 3)') instead of as one word ('data=(1,2,3)').
- added some tests for string format.
- bugfix: `BaseOutput.send()` raised string instead of `ValueError`.

1.2.1 (2017-03-10)

- bugfix: IO port never received anything when used with RtMidi backend. (Reported by dagargo, issue #83.)

  This was caused by a very old bug introduced in 1.0.3. IOPort mistakenly called the inner method `self.input._receive()` instead of `self.input.receive()`. This happens to work for ports that override `_receive()` but not for the new RtMidi backend which overrides `receive()`. (The default implementation of `_receive()` just drops the message on the floor.)
- bugfix: PortMidi backend was broken due to missing import (`ctypes.byref`). (Introduced in 1.2.0.)

1.2.0 (2017-03-07)

New implementation of messages and parser:

- completely reimplemented messages. The code is now much simpler, clearer and easier to work with.
- new contructors `Message.from_bytes()`, `Message.from_hex()`, `Message.from_str()`.
- new message attributes `is_meta` and `is_realtime`.

Frozen (immutable) messages:

- added `FrozenMessage` and `FrozenMetaMessage`. These are immutable versions of `Message` and `MetaMessage` that are hashable and thus can be used as dictionary keys. These are available in `mido.frozen`. (Requested by Jasper Lyons, issue #36.)

RtMidi is now the default backend:

- switched default backend from PortMidi to RtMidi. RtMidi is easier to install on most systems and better in every way.

  If you want to stick to PortMidi you can either set the environment variable `$MIDO_BACKEND=mido.backends.portmidi` or call `mido.set_backend('mido.backends.portmidi')` in your program.

- refactored the RtMidi backend to have a single `Port` class instead of inheriting from base ports. It was getting hard to keep track of it all. The code is now a lot easier to reason about.
- you can now pass `client_name` when opening RtMidi ports: `open_output('Test', client_name='My Client')`. When `client_name` is passed the port will automatically be a virtual port.
- with `LINUX_ALSA` you can now omit client name and ALSA client/port number when opening ports, allowing you to do `mido.open_output('TiMidity port 0')` instead of `mido.open_output('TiMidity:TiMidity port 0 128:0')`. (See RtMidi backend docs for more.)

Changes to the port API:

- ports now have `is_input` and `is_output` attributes.
- new functions `tick2second()` and `second2tick()`. (By Carl Thomé, pull request #71.)
- added `_locking` attribute to `BasePort`. You can set this to `False` in a subclass to do your own locking.
- `_receive()` is now allowed to return a messages. This makes the API more consistent and makes it easier to implement thread safe ports.
- `pending()` is gone. This had to be done to allow for the new `_receive()` behavior.
- improved MIDI file documentation. (Written by Carl Thomé.)

Other changes:

- bugfix: if a port inherited from both `BaseInput` and `BaseOutput` this would cause `BasePort.__init__()` to be called twice, which means `self._open()` was also called twice. As a workaround `BasePort.__init__()` will check if `self.closed` exists.
- added `mido.version_info`.
- `mido.set_backend()` can now be called with `load=True`.
- added `multi_send()`.
- `MIN_PITCHWHEEL`, `MAX_PITCHWHEEL`, `MIN_SONGPOS` and `MAX_SONGPOS` are now available in the top level module (for example `mido.MIN_PITCHWHEEL`).
- added experimental new backend `mido.backends.amidi`. This uses the ALSA `amidi` command to send and receive messages, which makes it very inefficient but possibly useful for sysex transfer.
- added new backend `mido.backends.rtmidi_python` (previously available in the examples folder.) This uses the `rtmidi-python` package instead of `python-rtmidi`. For now it lacks some of features of the `rtmidi` backend, but can still be useful on systems where `python-rtmidi` is not available. (Requested by netchose, issue #55.)

1.1.24 (2017-02-16)

- bugfix: PortMidi backend was broken on macOS due to a typo. (Fix by Sylvain Le Groux, pull request #81.)

1.1.23 (2017-01-31)

- bugfix: `read_syx_file()` didn't handle 'n' in text format file causing it to crash. (Reported by Paul Forgey, issue #80.)

1.1.22 (2017-01-27)

- the bugfix in 1.1.20 broke blocking receive() for RtMidi. Reverting the changes. This will need some more investigation.

1.1.21 (2017-01-26)

- bugfix: MidiFile save was broken in 1.1.20 due to a missing import.

1.1.20 (2017-01-26)

- bugfix: close() would sometimes hang for RtMidi input ports. (The bug was introduced in 1.1.18 when the backend was rewritten to support true blocking.)
- Numpy numbers can now be used for all message attributes. (Based on implementation by Henry Mao, pull request #78.)

  The code checks against numbers.Integral and numbers.Real (for the time attribute) so values can be any subclass of these.

1.1.19 (2017-01-25)

- Pygame backend can now receive sysex messages. (Fix by Box of Stops.)
- bugfix: `libportmidi.dylib` was not found when using MacPorts. (Fix by yam655, issue #77.)
- bugfix: `SocketPort.__init()` was not calling `IOPort.__init__()` which means it didn't get a `self._lock`. (Fixed by K Lars Lohn, pull request #72. Also reported by John J. Foerch, issue #79.)
- fixed typo in intro example (README and index.rst). Fix by Antonio Ospite (pull request #70), James McDermott (pull request #73) and Zdravko Bozakov (pull request #74).
- fixed typo in virtual ports example (Zdravko Bozakov, pull request #75.)

1.1.18 (2016-10-22)

- `time` is included in message comparison. `msg1 == msg2` will now give the same result as `str(msg1) == str(msg2)` and `repr(msg1) == repr(msg2)`.

  This means you can now compare tracks wihout any trickery, for example: `mid1.tracks == mid2.tracks`.

  If you need to leave out time the easiest was is `msg1.bytes() == msg2.bytes()`.

  This may in rare cases break code.
- bugfix: `end_of_track` messages in MIDI files were not handled correctly. (Reported by Colin Raffel, issue #62).
- bugfix: `merge_tracks()` dropped messages after the first `end_of_track` message. The new implementation removes all `end_of_track` messages and adds one at the end, making sure to adjust the delta times of the remaining messages.
- refactored MIDI file code.
- `mido-play` now has a new option `-m / --print-messages` which prints messages as they are played back.

- renamed `parser._parsed_messages` to `parser.messages`. `BaseInput` and `SocketPort` use it so it should be public.
- `Parser()` now takes an option arugment `data` which is passed to `feed()`.

1.1.17 (2016-10-06)

- RtMidi now supports true blocking `receive()` in Python 3. This should result in better performance and lower latency. (Thanks to Adam Roberts for helping research queue behavior. See issue #49 for more.)
- bugfix: `MidiTrack.copy()` (Python 3 only) returned `list`.
- fixed example `queue_port.py` which broke when locks where added.

1.1.16 (2016-09-27)

- bugfix: `MidiTrack` crashed instead of returning a message on `track[index]`. Fix by Colin Raffel (pull request #61).
- added `__add__()` and `__mul__()` to `MidiTrack` so + and * will return tracks instead of lists.
- added `poll()` method to input ports as a shortcut for `receive(block=False)`.
- added example `rtmidi_python_backend.py`, a backend for the rtmidi-python package (which is different from the python-rtmidi backend that Mido currently uses.) This may at some point be added to the package but for now it's in the examples folder. (Requested by netchose, issue #55.)
- removed custom `_import_module()`. Its only function was to make import errors more informative by showing the full module path, such as `ImportError: mido.backends.rtmidi` instead of just `ImportError: rtmidi`. Unfortunately it ended up masking import errors in the backend module, causing confusion.

  It turns `importlib.import_module()` can be called with the full path, and on Python 3 it will also display the full path in the `ImportError` message.

1.1.15 (2016-08-24)

- Sending and receiving messages is now thread safe. (Initial implementation by Adam Roberts.)
- Bugfix: `PortServer` called `__init__` from the wrong class. (Fix by Nathan Hurst.)
- Changes to `MidiTrack`:
  - `MidiTrack()` now takes a as a parameter an iterable of messages. Examples:

    ```
    MidiTrack(messages)
    MidiTrack(port.iter_pending())
    MidiTrack(msg for msg in some_generator)
    ```

  - Slicing a `MidiTrack` returns a `MidiTrack`. (It used to return a `list`.) Example:

    ```
    track[1:10]
    ```

- Added the ability to use file objects as well as filenames when reading, writing and saving MIDI files. This allows you to create a MIDI file dynamically, possibly *not* using mido, save it to an io.BytesIO, and then play that in-memory file, without having to create an intermediate external file. Of course the memory file (and/or the MidiFile) can still be saved to an external file. (Implemented by Brian O'Neill.)

- PortMidi backend now uses pm.lib.Pm_GetHostErrorText() to get host error messages instead of just the generic "PortMidi: 'Host error"'. (Implemented by Tom Manderson.)

Thanks to Richard Vogl and Tim Cook for reporting errors in the docs.

1.1.14 (2015-06-09)

- bugfix: merge_tracks() concatenated the tracks instead of merging them. This caused tracks to be played back one by one. (Issue #28, reported by Charles Gillingham.)
- added support for running status when writing MIDI files. (Implemented by John Benediktsson.)
- rewrote the callback system in response to issues #23 and #25.
- there was no way to set a callback function if the port was opened without one. (Issue#25, reported by Nils Werner.)

  Callbacks can now be set and cleared at any time by either passing one to `open_input()` or updating the `callback` attribute.

  This causes some slight changes to the behavior of the port when using callbacks. Previously if you opened the port with a callback and then set `port.callback = None` the callback thread would keep running but drop any incoming messages. If you do the same now the callback thread will stop and the port will return normal non-callback behavior. If you want the callback thread to drop messages you can set `port.callback = lambda message: None`.

  Also, `receive()` no longer checks `self.callback`. This was inconsistent as it was the only method to do so. It also allows ports that don't support callbacks to omit the `callback` attribute.
- bugfix: closing a port would sometimes cause a segfault when using callbacks. (Issue #24, reported by Francesco Ceruti.)
- bugfix: Pygame ports were broken due to a faulty check for `virtual=True`.
- now raises `ValueError` instead of `IOError` if you pass `virtual` or `callback` while opening a port and the backend doesn't support them. (An unsupported argument is not an IO error.)
- fixed some errors in backend documentation. (Pull request #23 by velolala.)
- `MultiPort` now has a `yield_port` argument just like `multi_receive()`.

1.1.13 (2015-02-07)

- the PortMidi backend will now return refresh the port list when you ask for port names are open a new port, which means you will see devices that you plug in after loading the backend. (Due to limitations in PortMidi the list will only be refreshed if there are no open ports.)
- bugfix: `tempo2bpm()` was broken and returned the wrong value for anything but 500000 microseconds per beat (120 BPM). (Reported and fixed by Jorge Herrera, issue #21)
- bugfix: `merge_tracks()` didn't work with empty list of tracks.
- added proper keyword arguments and doc strings to open functions.

1.1.12 (2014-12-02)

- raises IOError if you try to open a virtual port with PortMidi or Pygame. (They are not supported by these backends.)
- added `merge_tracks()`.

- removed undocumented method `MidiFile.get_messages()`. (Replaced by `merge_tracks(mid.tracks)`.)
- bugfix: `receive()` checked `self.callback` which didn't exist for all ports, causing an `AttributeError`.

1.1.11 (2014-10-15)

- added `bpm2tempo()` and `tempo2bpm()`.
- fixed error in documentation (patch by Michael Silver).
- added notes about channel numbers to documentation (reported by ludwig404 / leonh, issue #18).

1.1.10 (2014-10-09)

- bugfix: MidiFile.length was computer incorrectly.
- bugfix: tempo changes caused timing problems in MIDI file playback. (Reported by Michelle Thompson.)
- mido-ports now prints port names in single ticks.
- MidiFile.__iter__() now yields end_of_track. This means playback will end there instead of at the preceding message.

1.1.9 (2014-10-06)

- bugfix: _compute_tick_time() was not renamed to _compute_seconds_per_tick() everywhere.
- bugfix: sleep time in play() was sometimes negative.

1.1.8 (2014-09-29)

- bugfix: timing in MIDI playback was broken from 1.1.7 on. Current time was subtracted before time stamps were converted from ticks to seconds, leading to absurdly large delta times. (Reported by Michelle Thompson.)
- bugfix: `read_syx_file()` didn't handle empty file.

1.1.7 (2014-08-12)

- some classes and functions have been moved to more accessible locations:

```
from mido import MidiFile, MidiTrack, MetaMessage
from mido.midifiles import MetaSpec, add_meta_spec
```

- you can now iterate over a MIDI file. This will generate all MIDI messages in playback order. The `time` attribute of each message is the number of seconds since the last message or the start of the file. (Based on suggestion by trushkin in issue #16.)
- added get_sleep_time() to complement set_sleep_time().
- the Backend object no longer looks for the backend module exists on startup, but will instead just import the module when you call one of the `open_*()` or `get_*()` functions. This test didn't work when the library was packaged in a zip file or executable.

This means that Mido can now be installed as Python egg and frozen with tools like PyInstaller and py2exe. See "Freezing Mido Programs" for more on this.

(Issue #17 reported by edauenhauer and issue #14 reported by netchose.)

- switched to pytest for unit tests.

1.1.6 (2014-06-21)

- bugfix: package didn't work with easy_install. (Issue #14, reported by netchose.)
- bugfix: 100% memory consumption when calling blocking receive() on a PortMidi input. (Issue #15, reported by Francesco Ceruti.)
- added wheel support: http://pythonwheels.com/

1.1.5 (2014-04-18)

- removed the 'mode' attribute from key_signature messages. Minor keys now have an 'm' appended, for example 'Cm'.
- bugfix: sysex was broken in MIDI files.
- bugfix: didn't handle MIDI files without track headers.
- bugfix: MIDI files didn't handle channel prefix > 15
- bugfix: MIDI files didn't handle SMPTE offset with frames > 29

1.1.4 (2014-10-04)

- bugfix: files with key signatures Cb, Db and Gb failed due to faulty error handling.
- bugfix: when reading some MIDI files Mido crashed with the message "ValueError: attribute must be in range 0..255". The reason was that Meta messages set running status, which caused the next statusless message to be falsely interpreted as a meta message. (Reported by Domino Marama).
- fixed a typo in MidiFile._read_track(). Sysex continuation should work now.
- rewrote tests to make them more readable.

1.1.3 (2013-10-14)

- messages are now copied on send. This allows the sender to modify the message and send it to another port while the two ports receive their own personal copies that they can modify without any side effects.

1.1.2 (2013-10-05)

- bugfix: non-ASCII character caused trouble with installation when LC_ALL=C. (Reported by Gene De Lisa)
- bugfix: used old exception handling syntax in rtmidi backend which broke in 3.3
- fixed broken link in

1.1.1 (2013-10-04)

- bugfix: mido.backends package was not included in distribution.

1.1.0 (2013-10-01)

- added support for selectable backends (with MIDO_BACKEND) and included python-rtmidi and pygame backends in the official library (as mido.backend.rtmidi and mido.backend.pygame).
- added full support for MIDI files (read, write playback)
- added MIDI over TCP/IP (socket ports)
- added utility programs mido-play, mido-ports, mido-serve and mido-forward.
- added support for SMPTE time code quarter frames.
- port constructors and `open_*()` functions can now take keyword arguments.
- output ports now have reset() and panic() methods.
- new environment variables MIDO_DEFAULT_INPUT, MIDO_DEFAULT_OUTPUT and MIDO_DEFAULT_IOPORT. If these are set, the open_*() functions will use them instead of the backend's default ports.
- added new meta ports MultiPort and EchoPort.
- added new examples and updated the old ones.
- format_as_string() now takes an include_time argument (defaults to True) so you can leave out the time attribute.
- sleep time inside sockets can now be changed.
- Message() no longer accepts a status byte as its first argument. (This was only meant to be used internally.)
- added callbacks for input ports (PortMidi and python-rtmidi)
- PortMidi and pygame input ports now actually block on the device instead of polling and waiting.
- removed commas from repr() format of Message and MetaMessage to make them more consistent with other classes.

1.0.4 (2013-08-15)

- rewrote parser

1.0.3 (2013-07-12)

- bugfix: __exit__() didn't close port.
- changed repr format of message to start with "message".
- removed support for undefined messages. (0xf4, 0xf5, 0xf7, 0xf9 and 0xfd.)
- default value of velocity is now 64 (0x40). (This is the recommended default for devices that don't support velocity.)

1.0.2 (2013-07-31)

- fixed some errors in the documentation.

1.0.1 (2013-07-31)

- multi_receive() and multi_iter_pending() had wrong implementation. They were supposed to yield only messages by default.

1.0.0 (2013-07-20)

Initial release.

Basic functionality: messages, ports and parser.

Roadmap

This will be developed into a proper roadmap but for now it's more of a list of ideas.

Near Future

- create a place for general discussion, for example a google group (mido-discuss and perhaps a separate mido-dev).
- a PEP like process for new features and major changes.

Various Improvements to MIDI File Parsing

- add `mido.exceptions.MidiParseError`.
- add better error handling to MIDI file parser as discussed in issue #63.
- support RIFF MIDI files (issue #43)
- support MIDI files that end in empty meta message (issue #42)

Better Support for Concurrency and Multithreading

Mido was not originally designed for multithreading. Locks were added to the port base classes as an attempt to get around this but it is a crude solution that has created numerous problems and headaches.

The RtMido backend has abandoned locking in favor of using RtMidi's user callback to feed a queue. If you write your port so that `send()`, `receive()` and `poll()` are thread safe the rest of the rest of the API will be as well.

For ports that do actual I/O (MIDI devices, sockets, files, pipes etc.) it is always best for the port itself to ensure thread safety. It's less clear what to do for utility ports like `MultiPort`.

Mido is currently not very good at multiplexing input. You can use `MultiPort` and `multi_receive()`, but since it can't actually block on more than one port it uses poll and wait. This uses more resources and adds latency.

The alternative is to use callbacks, but only a few backends support (and some like PortMidi fake them with a thread that polls and waits, taking you back to square one). Programming with callbacks also forces you to deal with multithreading (since the callback runs in a different thread) and to break your program flow up into callback handlers. This is not always desirable.

In Go the solution would be to use channels instead of ports. Each port would then have its own channel with a goroutine reading from the device and feeding the channel, and the language would take care of the multiplexing. I am not sure how one would achieve something like this in Python.

Making Messages Immutable

See: https://github.com/olemb/mido/issues/36

The current workaround is frozen messages (`mido.freeze`).

In any case, the documentation should be updated to encourage copying over mutation.

Native Backends (ALSA, JACK, CoreMIDI, Windows MIDI)

See https://github.com/olemb/mido-native-backends

No Default Backend?

Currently one backend is chosen as the default. Perhaps it would be better to require the user to specify the backend with `$MIDO_BACKEND` or `mido.set_backend()`.

New API for Creating New Port Types

The current system uses multiple inheritance making the code very hard to follow and reason about:

- too much magic and too much you need to keep in your head
- attributes like `self.name` and `self.closed` appear in your name space and you have to dig around in the base classes to see where they come from
- there is a `self._parser` in every port even if you don't need it
- `self._parser` is not thread safe
- `BaseInput.receive()` needs to account for all the numerous current and historical behaviors of `self._receive()`.
- blocking and nonblocking receive can not be protected by the same lock (since a call to `receive()` would then block a call to `poll()`), which means ports that due true blocking need to do their own locking.
- if you want to do our own locking you need to remember to set `_locking=False`. This will replace the lock with a dummy lock, which while doing nothing still adds a bit of overhead.

A good principle is for any part of the code to know as little as possible about the rest of the code. For example a backend port should only need to worry about:

- opening and closing the device
- reading and writing data (blocking and nonblocking)

It should not have to worry about things like `autoreset`, `closed=True/False` and iteration. Also, as long as its `send()` and `receive()/poll()` methods are thread safe the rest of the API will be as well.

Some alternatives to subclassing:

- embedding: write a basic port and wrap it in a `FancyPort` which provides the rest of the API
- mixins: write a basic port and use mixins (`PortMethods`, `InputMethods`, `OutputMethods`) to import the rest of the API.

Maybe

- add a way to convert between MIDI file types as suggested in issue #92.
- RtMidi backend: allow user to list ports without client name and ALSA port numbers.
- Add native backends? See https://github.com/olemb/mido-native-backends
- Currently all backends ignore `active_sensing` messages because they create a lot of noise and are usually not very useful. Should this be changed (perhaps as an option)?

  Filtering can be turned off with:

  - rtmidi: `self._rt.ignore_types(False, False, False)`
  - portmidi: `pm.lib.Pa_SetFilter(self._stream, 0)`
  - rtmidi_python: `self._rt.ignore_types(False, False, False)`
  - pygame: (is there a way to configure this?)
  - amidi: (not sure if this receives `active_sensing` already)
- Refactor `rtmidi` and `rtmidi_python` backends to avoid code duplication. This would give `rtmidi_python` all of the features of `rtmidi` (as long as they are supported in the package).
- Add more fine grained error types, for example `PortNotFound` instead of just `IOError`. (This should be a subclass so old code still works.) One problem here is that new code that uses `PortNotFound` would not work with backends that raise `IOError`.

Installing Mido

Requirements

Mido targets Python 2.7 and 3.2. It is developed and tested in Ubuntu and Mac OS X but should also work in Windows.

There are no external dependencies unless you want to use the port backends, which are loaded on demand.

Mido comes with backends for RtMidi (python-rtmidi) , PortMidi and Pygame. See *Backends* for help choosing a backend.

Installing

To install:

```
pip install mido
```

If you want to use ports:

```
pip install python-rtmidi
```

See *Backends* for installation instructions for other backends.

Backends

RtMidi (Default, Recommended)

Name: `mido.backends.rtmidi`

The RtMidi backend is a thin wrapper around python-rtmidi

Features:

- callbacks
- true blocking `receive()` in Python 3 (using a callback and a queue)
- virtual ports
- ports can be opened multiple times, each will receive a copy of each message
- client name can be specified when opening a virtual port
- sends but doesn't receive active sensing
- port list is always up to date
- all methods but `close()` are thread safe

Port Names (Linux/ALSA

When you're using Linux/ALSA the port names include client name and ALSA client and port numbers, for example:

```
>>> mido.get_output_names()
['TiMidity:TiMidity port 0 128:0']
```

The ALSA client and port numbers ("128:0" in this case) can change from session to session, making it hard to hard code port names or use them in config files.

To get around this the RtMidi backend allows you to leave out the the port number of port number and client names. These lines will all open the port above:

```
mido.open_output('TiMidity port 0')
```

```
mido.open_output('TiMidity:TiMidity port 0')
```

```
mido.open_output('TiMidity:TiMidity port 0 128:0')
```

There is currently no way to list ports without port number or client name. This can be added in a future version of there is demand for it and a suitable API is found.

Virtual Ports

RtMidi is the only backend that can create virtual ports:

```
>>> port = mido.open_input('New Port', virtual=True)
>>> port
<open input 'New Port' (RtMidi/LINUX_ALSA)>
```

Other applications can now connect to this port. (One oddity is that, at least in Linux, RtMidi can't see its own virtual ports, while PortMidi can see them.)

Client Name

You can specify a client name for the port: (New in 1.2.0.)

```
>>> port = mido.open_input('New Port', client_name='My Client')
```

This requires python-rtmidi >= 1.0rc1. If `client_name` is passed the port will be a virtal port.

Note: Unfortunately, at least with ALSA, opening two ports with the same `client_name` creates two clients with the same name instead of one client with two ports.

There are a couple of problems with port names in Linux. First, RtMidi can't see some software ports such as `amSynth MIDI IN`. PortMidi uses the same ALSA sequencer API, so this is problem in RtMidi.

Second, in some versions of RtMidi ports are named inconsistently. For example, the input port 'Midi Through 14:0' has a corresponding output named 'Midi Through:0'. Unless this was intended, it is a bug in RtMidi's ALSA implementation.

Choosing API

The RtMidi library can be compiled with support for more than one API. You can select API by adding it after the module name, either in the environment variable:

```
$ export MIDO_BACKEND=mido.backends.rtmidi/LINUX_ALSA
$ export MIDO_BACKEND=mido.backends.rtmidi/UNIX_JACK
```

or in one of these:

```
>>> mido.set_backend('mido.backends.rtmidi/LINUX_ALSA')
>>> mido.backend
<backend mido.backends.rtmidi/LINUX_ALSA (not loaded)>

>>> mido.Backend('mido.backends.rtmidi/UNIX_JACK')
<backend mido.backends.rtmidi/UNIX_JACK (not loaded)>
```

This allows you to, for example, use both ALSA and JACK ports in the same program.

To get a list of available APIs:

```
>>> mido.backend.module.get_api_names()
['LINUX_ALSA', 'UNIX_JACK']
```

PortMidi

Name: `mido.backends.portmidi`

Installing

The PortMidi backend requires the `portmidi` shared library.

Ubuntu:

```
apt install libportmidi-dev
```

Homebrew:

```
brew install portmidi
```

MacPorts:

```
port install portmidi
```

The backend will look for:

```
portmidi.so       (Linux)
portmidi.dll      (Windows)
portmidi.dynlib   (macOS)
```

Features

Can send but doesn't receive `active_sensing` messages.

PortMidi has no callback mechanism, so callbacks are implemented in Python with threads. Each port with a callback has a dedicated thread doing blocking reads from the device.

Due to limitations in PortMidi the port list will not be up-to-date if there are any ports open. (The refresh is implemented by re-initalizing PortMidi which would break any open ports.)

Pygame

Name: `mido.backends.pygame`

The Pygame backend uses `pygame.midi` for I/O.

Doesn't receive `active_sensing`.

Callbacks are currently not implemented.

Pygame.midi is implemented on top of PortMidi.

rtmidi_python

Name: `mido.backends.rtmidi_python`

Installing

```
pip install rtmidi-python
```

Features

- uses the `rtmidi_python` package rather than `python_rtmidi`
- supports callbacks
- limited support for virtual ports (no client name)

- no true blocking
- sends but doesn't receive active sensing

Since the API of `rtmidi_python` and `python_rtmidi` are almost identical it would make sense to refactor so they share most of the code.

amidi (Experimental)

Name: `mido.backends.amidi`

Features:

- Linux only.
- very basic implementation.
- no callbacks
- can only access physical ports. (Devices that are plugged in.)
- high overhead when sending since it runs the `amidi` command for each messages.
- known bug: is one behind when receiving messages. See below.

The `amidi` command (a part of ALSA) is used for I/O:

```
* ``amidi -l`` to list messages (in ``get_input_names()`` etc.)
```

- `amidi -d -p DEVICE` to receive messages. `amidi` prints these out one on each line as hex bytes. Unfortunately it puts the newline at the beginning of the line which flushes the buffer before the message instead of after. This causes problems with non-blocking receiption using `select.poll()` which means messages are received one behind. This needs to be looked into.
- `amidi --send-hex MESSAGE_IN_HEX -p DEVICE` to send messages. Since this is called for every messages the overhead is very high.

Choosing a Backend

Mido comes with five backends:

- *RtMidi* is the recommended backends. It has all the features of the other ones and more and is usually easier to install.
- *PortMidi* was the default backend up until 1.2. It uses the `portmidi` shared library and can be difficult to install on some systems.
- *Pygame* uses the `pygame.midi`.
- *rtmidi-python* uses the `rtmidi_python` package, an alternative wrapper for PortMidi. It is currently very basic but easier to install on some Windows systems.
- *Amidi* is an experimental backend for Linux/ALSA that uses the command `amidi` to send and receive messages.

If you want to use another than the RtMidi you can override this with the `MIDO_BACKEND` environment variable, for example:

```
$ MIDO_BACKEND=mido.backends.portmidi ./program.py
```

Alternatively, you can set the backend from within your program:

```
>>> mido.set_backend('mido.backends.portmidi')
>>> mido.backend
<backend mido.backends.portmidi (not loaded)>
```

This will override the environment variable.

If you want to use more than one backend at a time, you can do:

```
rtmidi = mido.Backend('mido.backends.rtmidi')
portmidi = mido.Backend('mido.backends.portmidi')

input = rtmidi.open_input()
output = portmidi.open_output()
for message in input:
    output.send(message)
```

The backend will not be loaded until you call one of the `open_` or `get_` methods. You can pass `load=True` to have it loaded right away.

If you pass `use_environ=True` the module will use the environment variables `MIDO_DEFAULT_INPUT` etc. for default ports.

Environment Variables

You can override the backend's choice of default ports with these three environment variables:

```
MIDO_DEFAULT_INPUT
MIDO_DEFAULT_OUTPUT
MIDO_DEFAULT_IOPORT
```

For example:

```
$ MIDO_DEFAULT_INPUT='SH-201' python program.py
```

or:

```
$ export MIDO_DEFAULT_OUTPUT='Integra-7'
$ python program1.py
$ python program2.py
```

Contributing

Testing

pytest is used for unit testing. The tests are found in *mido/test_*.py*.

If possible please run tests in both Python 2 and Python 3 before you commit code:

```
python2 -m pytest && python3 -m pytest
```

You can also set up a commit hook:

```
echo "python2 -m pytest && python3 -m pytest" >.git/hooks/pre-commit
chmod +x .git/hooks/pre-commit
```

This will run tests when you commit and cancel the commit if any tests fail.

Testing MIDI file support

Test Files

The Lakh MIDI Dataset is a great resouce for testing the MIDI file parser.

Publishing (Release Checklist)

I am currently the only one with access to publishing on PyPI and readthedocs. This will hopefully change in the future.

First Time: Register With PyPI

```
./setup.py register
```

Test

```
rm -rf docs/_build && ./setup.py docs
pytest2 && pytest3
check-manifest -v
```

(*pip3 install check-manifest*)

You can also test that the package installs by installing it in a virtualenv with *pip* and *easy_install* (Python 2 and 3) and importing it. This is a bit tedious. Perhaps there is a good way to automate it.

Bump Version

X.Y.Z is the version, for example 1.1.18 or 1.2.0.

- update version and date in *docs/changes.rst*
- update version in *mido/version.py*
- *git commit -a -c "Bumped version to X.Y.Z."*

Then:

```
git tag X.Y.Z
git push
git push --tags
```

Update the stable branch (if this is a stable release):

```
git checkout stable
git pull . master
git push
git checkout master
```

Publish

Publish in PyPI:

```
python setup.py publish
python setup.py bdist_wheel upload
```

Last thing:

Update readthedocs

Introduction (Basic Concepts)

Mido is all about messages and ports.

Messages

Mido allows you to work with MIDI messages as Python objects. To create a new message:

```
>>> from mido import Message
>>> msg = Message('note_on', note=60)
>>> msg
<message note_on channel=0 note=60 velocity=64 time=0>
```

Note: Mido numbers channels 0 to 15 instead of 1 to 16. This makes them easier to work with in Python but you may want to add and subtract 1 when communicating with the user.

A list of all supported message types and their parameters can be found in *Message Types*.

The values can now be accessed as attributes:

```
>>> msg.type
'note_on'
>>> msg.note
60
>>> msg.velocity
64
```

Attributes are also settable but this should be avoided. It's better to use `msg.copy()`:

```
>>> msg.copy(note=100, velocity=127)
<message note_on channel=2 note=100 velocity=127 time=0)
```

Type and value checks are done when you pass parameters or assign to attributes, and the appropriate exceptions are raised. This ensures that the message is always valid.

For more about messages, see *Messages*.

Type and Value Checking

Mido messages come with type and value checking built in:

```
>>> import mido
>>> mido.Message('note_on', channel=2092389483249829834)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/olemb/src/mido/mido/messages/messages.py", line 89, in __init__
    check_msgdict(msgdict)
  File "/home/olemb/src/mido/mido/messages/checks.py", line 100, in check_msgdict
    check_value(name, value)
  File "/home/olemb/src/mido/mido/messages/checks.py", line 87, in check_value
    _CHECKS[name](value)
  File "/home/olemb/src/mido/mido/messages/checks.py", line 17, in check_channel
    raise ValueError('channel must be in range 0..15')
ValueError: channel must be in range 0..15
```

This means that the message object is always a valid MIDI message.

Ports

To create an output port and send a message:

```
>>> outport = mido.open_output()
>>> outport.send(msg)
```

To create an input port and receive a message:

```
>>> inport = mido.open_input()
>>> msg = inport.receive()
```

Note: Multiple threads can safely send and receive notes on the same port.

This will give you the default output and input ports. If you want to open a specific port, you will need its name. To get a list of all available input ports:

```
>>> mido.get_input_names()
['Midi Through Port-0', 'SH-201', 'Integra-7']
>>> inport = mido.open_input('SH-201')
```

All Mido ports can be used with the `with` statement, which will close the port for you:

```
with mido.open_input('SH-201') as inport:
    ...
```

To iterate through all incoming messages:

```
for msg in inport:
    ...
```

You can also receive and iterate over messages in a non-blocking way.

For more about ports, see *Ports*.

All Ports are Ports

The input and output ports used above are device ports, which communicate with a (physical or virtual) MIDI device.

Other port types include:

- `MultiPort`, which wraps around a set of ports and allow you to send to all of them or receive from all of them as if they were one.
- `SocketPort`, which communicates with another port over a TCP/IP (network) connection.
- `IOPort`, which wraps around an input and an output port and allows you to send and receive messages as if the two were the same port.

Ports of all types look and behave the same way, so they can be used interchangeably.

It's easy to write new port types. See *Writing a New Port*.

Virtual Ports

Virtual ports allow you to create new ports that other applications can connect to:

```
with mido.open_input('New Port', virtual=True) as inport:
    for message in inport:
        print(message)
```

The port should now appear to other applications as "New Port".

Unfortunately virtual ports are not supported by PortMidi and Pygame so this only works with RtMidi.

Parsing MIDI Bytes

Mido comes with a parser that allows you to turn bytes into messages. You can create a new parser:

```
>>> p = mido.Parser()
>>> p.feed([0x90, 0x40])
>>> p.feed_byte(0x60)
```

You can then fetch messages out of the parser:

```
>>> p.pending()
1
>>> for message in p:
...     print(message)
...
note_on channel=0 note=64 velocity=96 time=0
```

For more on parsers and parsing see *Parsing MIDI Bytes*.

You can also create a message from bytes using class methods (new in 1.2):

```
msg1 = mido.Message.from_bytes([0x90, 0x40, 0x60])
msg2 = mido.Message.from_hex('90, 40 60')
```

The bytes must contain exactly one complete message. If not `ValueError` is raised.

Backends

Mido comes with backends for RtMidi and PortMidi and Pygame. The default is RtMidi. You can select another backend or even use multiple backends at the same time. For more on this, see *Backends*.

Messages

A Mido message is a Python object with methods and attributes. The attributes will vary depending on message type.

To create a new message:

```
>>> mido.Message('note_on')
<message note_on channel=0 note=0 velocity=64 time=0>
```

You can pass attributes as keyword arguments:

```
>>> mido.Message('note_on', note=100, velocity=3, time=6.2)
<message note_on channel=0 note=100 velocity=3 time=6.2>
```

All attributes will default to 0. The exceptions are `velocity`, which defaults to 64 (middle velocity) and `data` which defaults to `()`.

You can set and get attributes as you would expect:

```
>>> msg = mido.Message('note_on')
>>> msg.note
0
```

The `type` attribute can be used to determine message type:

```
>>> msg.type
'note_on'
```

Attributes are also settable but it's always better to use `msg.copy()`:

```
>>> msg.copy(note=99, time=100.0)
<message note_on channel=0 note=99 velocity=64 time=100.0>
```

Note: Mido always makes a copy of messages instead of modifying them so if you do the same you have immutable messages in practice. (Third party libraries may not follow the same rule.)

Note: *Frozen Messages* are a variant of messages that are hashable and can be used as dictionary keys. They are also safe from tampering by third party libraries. You can freely convert between the two and use frozen messages wherever normal messages are allowed.

Mido supports all message types defined by the MIDI standard. For a full list of messages and their attributes, see *Message Types*.

Converting To Bytes

You can convert a message to MIDI bytes with one of these methods:

```
>>> msg = mido.Message('note_on')
>>> msg
<message note_on channel=0 note=0 velocity=64 time=0>
>>> msg.bytes()
[144, 0, 64]
>>> msg.bin()
```

```
bytearray(b'\x90\x00@')
>>> msg.hex()
'90 00 40'
```

Converting From Bytes

You can turn bytes back into messages with the */parser*.

You can also create a message from bytes using class methods (new in 1.2):

```
msg1 = mido.Message.from_bytes([0x90, 0x40, 0x60])
msg2 = mido.Message.from_hex('90, 40 60')
```

The bytes must contain exactly one complete message. If not `ValueError` is raised.

The Time Attribute

Each message has a `time` attribute, which can be set to any value of type `int` or `float` (and in Python 2 also `long`). What you do with this value is entirely up to you.

Some parts of Mido use the attribute for special purposes. In MIDI file tracks, it is used as delta time (in ticks).

Note: Before 1.1.18 the `time` attribute was not included in comparisons. If you want the old behavior the easies way is `msg1.bytes() == msg2.bytes()`.

To sort messages on time you can do:

```
messages.sort(key=lambda message: message.time)
```

or:

```
import operator

messages.sort(key=operator.attrgetter('time'))
```

System Exclusive Messages

System Exclusive (SysEx) messages are used to send device specific data. The `data` attribute is a tuple of data bytes which serves as the payload of the message:

```
>>> msg = Message('sysex', data=[1, 2, 3])
>>> msg
<message sysex data=(1, 2, 3) time=0>
>>> msg.hex()
'F0 01 02 03 F7'
```

You can also extend the existing data:

```
>>> msg = Message('sysex', data=[1, 2, 3])
>>> msg.data += [4, 5]
>>> msg.data += [6, 7, 8]
```

```
>>> msg
<message sysex data=(1, 2, 3, 4, 5, 6, 7, 8) time=0>
```

Any sequence of integers is allowed, and type and range checking is applied to each data byte. These are all valid:

```
(65, 66, 67)
[65, 66, 67]
(i + 65 for i in range(3))
(ord(c) for c in 'ABC')
bytearray(b'ABC')
b'ABC'  # Python 3 only.
```

For example:

```
>>> msg = Message('sysex', data=bytearray(b'ABC'))
>>> msg.data += bytearray(b'DEF')
>>> msg
<message sysex data=(65, 66, 67, 68, 69, 70) time=0>
```

Frozen Messages

(New in 1.2.)

Since Mido messages are mutable (can change) they can not be hashed or put in dictionaries. This makes it hard to use them for things like Markov chains.

In these situations you can use frozen messages:

```
from mido.frozen import FrozenMessage

msg = FrozenMessage('note_on')
d = {msg: 'interesting'}
```

Frozen messages are used and behave in exactly the same way as normal messages with one exception: attributes are not settable.

There are also variants for meta messages (`FrozenMetaMessage` and `FrozenUnknownMetaMessage`).

You can freeze and thaw messages with:

```
from mido.frozen import freeze_message, thaw_message

frozen = freeze_message(msg)
thawed = thaw_message(frozen)
```

`thaw_message()` will always return a copy. Passing a frozen message to `freeze_message()` will return the original message.

Both functions return `None` if you pass `None` which is handy for things like:

```
msg = freeze_message(port.receive())

# Python 3 only:
for msg in map(freeze_message, port):
    ...

# Python 2 and 3:
```

```
for msg in (freeze_message(msg) for msg in port):
    ...
```

To check if a message is frozen:

```
from mido.frozen import is_frozen

if is_frozen(msg):
    ...
```

Ports

A Mido port is an object that can send or receive messages (or both).

You can open a port by calling one of the open methods, for example:

```
>>> inport = mido.open_input('SH-201')
>>> outport = mido.open_output('Integra-7')
```

Now you can receive messages on the input port and send messages on the output port:

```
>>> msg = inport.receive()
>>> outport.send(msg)
```

The message is copied by `send()`, so you can safely modify your original message without causing breakage in other parts of the system.

In this case, the ports are device ports, and are connected to some sort of (physical or virtual) MIDI device, but a port can be anything. For example, you can use a `MultiPort` receive messages from multiple ports as if they were one:

```
from mido.ports import MultiPort

...
multi = MultiPort([inport1, inport2, inport3])
for msg in multi:
    print(msg)
```

This will receive messages from all ports and print them out. Another example is a socket port, which is a wrapper around a TCP/IP socket.

No matter how the port is implemented internally or what it does, it will look and behave like any other Mido port, so all kinds of ports can be used interchangeably.

Note: Sending and receiving messages is thread safe. Opening and closing ports and listing port names are not.

Common Things

How to open a port depends on the port type. Device ports (PortMidi, RtMidi and others defined in backends) are opened with the open functions, for example:

```
port = mido.open_output()
```

Input and I/O ports (which support both input and output) are opened with `open_input()` and `open_ioport()` respectively. If you call these without a port name like above, you will get the (system specific) default port. You can override this by setting the `MIDO_DEFAULT_OUTPUT` etc. environment variables.

To get a list of available ports, you can do:

```
>>> mido.get_output_names()
['SH-201', 'Integra-7']
```

and then:

```
>>> port = mido.open_output('Integra-7')
```

There are corresponding function for input and I/O ports.

To learn how to open other kinds of ports, see the documentation for the port type in question.

The port name is available in `port.name`.

To close a port, call:

```
port.close()
```

or use the `with` statement to have the port closed automatically:

```
with mido.open_input() as port:
    for message in port:
        do_something_with(message)
```

You can check if the port is closed with:

```
if port.closed:
    print("Yup, it's closed.")
```

If the port is already closed, calling `close()` will simply do nothing.

Output Ports

Output ports basically have only one method:

```
outport.send(message)
```

This will send the message immediately. (Well, the port can choose to do whatever it wants with the message, but at least it's sent.)

There are also a couple of utility methods:

```
outport.reset()
```

This will send "all notes off" and "reset all controllers" on every channel. This is used to reset everything to the default state, for example after playing back a song or messing around with controllers.

If you pass `autoreset=True` to the constructor, `reset()` will be called when the port closes:

```
with mido.open_output('Integra-7') as outport:
    for msg in inport:
        outport.send(msg)
# reset() is called here
```

```
outport.close()  # or here
```

Sometimes notes hang because a `note_off` has not been sent. To (abruptly) stop all sounding notes, you can call:

```
outport.panic()
```

This will not reset controllers. Unlike `reset()`, the notes will not be turned off gracefully, but will stop immediately with no regard to decay time.

Input Ports

To iterate over incoming messages::

```
for msg in port:
    print(msg)
```

This will iterate over messages as they arrive on the port until the port closes. (So far only socket ports actually close by themselves. This happens if the other end disconnects.)

You can also do non-blocking iteration:

```
for msg in port.iter_pending():
    print(msg)
```

This will iterate over all messages that have already arrived. It is typically used in main loops where you want to do something else while you wait for messages:

```
while True:
    for msg in port.iter_pending():
        print(msg)

    do_other_stuff()
```

In an event based system like a GUI where you don't write the main loop you can install a handler that's called periodically. Here's an example for GTK:

```
def callback(self):
    for msg in self.inport:
        print(msg)

gobject.timeout_add_seconds(timeout, callback)
```

To get a bit more control you can receive messagas one at a time:

```
msg = port.receive()
```

This will block until a message arrives. To get a message only if one is available, you can use *poll()*:

```
msg = port.poll()
```

This will return `None` if no message is available.

Note: There used to be a `pending()` method which returned the number of pending messages. It was removed in 1.2.0 for three reasons:

- with `poll()` and `iter_pending()` it is no longer necessary
- it was unreliable when multithreading and for some ports it doesn't even make sense
- it made the internal method API confusing. *_send()* sends a message so *_receive()* should receive a message.

Callbacks

Instead of reading from the port you can install a callback function which will be called for every message that arrives.

Here's a simple callback function:

```
def print_message(message):
    print(message)
```

To install the callback you can either pass it when you create the port or later by setting the `callback` attribute:

```
port = mido.open_input(callback=print_message)
port.callback = print_message
...
port.callback = another_function
```

Note: Since the callback runs in a different thread you may need to use locks or other synchronization mechanisms to keep your main program and the callback from stepping on each other's toes.

Calling `receive()`, `__iter__()`, or `iter_pending()` on a port with a callback will raise an exception:

```
ValueError: a callback is set for this port
```

To clear the callback:

```
port.callback = None
```

This will return the port to normal.

Port API

Common Methods and Attributes

`close()`

Close the port. If the port is already closed this will simply do nothing.

`name`

Name of the port or None.

`closed`

True if the port is closed.

Output Port Methods

`send(message)`

Send a message.

`reset()`

Sends "all notes off" and "reset all controllers on all channels.

`panic()`

Sends "all sounds off" on all channels. This will abruptly end all sounding notes.

Input Port Methods

`receive(block=True)`

Receive a message. This will block until it returns a message. If `block=True` is passed it will instead return `None` if there is no message.

`poll()`

Returns a message, or `None` if there are no pending messages.

`iter_pending()`

Iterates through pending messages.

`__iter__()`

Iterates through messages as they arrive on the port until the port closes.

MIDI Files

MidiFile objects can be used to read, write and play back MIDI files.

Opening a File

You can open a file with:

```
from mido import MidiFile

mid = MidiFile('song.mid')
```

Note: Sysex dumps such as patch data are often stored in SYX files rather than MIDI files. If you get "MThd not found. Probably not a MIDI file" try `mido.read_syx_file()`. (See *SYX Files* for more.)

The `tracks` attribute is a list of tracks. Each track is a list of messages and meta messages, with the `time` attribute of each messages set to its delta time (in ticks). (See Tempo and Beat Resolution below for more on delta times.)

To print out all messages in the file, you can do:

```
for i, track in enumerate(mid.tracks):
    print('Track {}: {}'.format(i, track.name))
    for msg in track:
        print(msg)
```

The entire file is read into memory. Thus you can freely modify tracks and messages, and save the file back by calling the `save()` method. (More on this below.)

Iterating Over Messages

Iterating over a `MidiFile` object will generate all MIDI messages in the file in playback order. The `time` attribute of each message is the number of seconds since the last message or the start of the file.

Meta messages will also be included. If you want to filter them out, you can do:

```
if msg.is_meta:
    ...
```

This makes it easy to play back a MIDI file on a port:

```
for msg in MidiFile('song.mid'):
    time.sleep(msg.time)
    if not msg.is_meta:
        port.send(msg)
```

This is so useful that there's a method for it:

```
for msg in MidiFile('song.mid').play():
    port.send(msg)
```

This does the sleeping and filtering for you. If you pass `meta_messages=True` you will also get meta messages. These can not be sent on ports, which is why they are off by default.

Creating a New File

You can create a new file by calling MidiFile without the `filename` argument. The file can then be saved by calling the `save()` method:

```
from mido import Message, MidiFile, MidiTrack

mid = MidiFile()
track = MidiTrack()
mid.tracks.append(track)

track.append(Message('program_change', program=12, time=0))
track.append(Message('note_on', note=64, velocity=64, time=32))
track.append(Message('note_off', note=64, velocity=127, time=32))

mid.save('new_song.mid')
```

The `MidiTrack` class is a subclass of list, so you can use all the usual methods.

All messages must be tagged with delta time (in ticks). (A delta time is how long to wait before the next message.)

If there is no 'end_of_track' message at the end of a track, one will be written anyway.

A complete example can be found in `examples/midifiles/`.

The `save` method takes either a filename (`str`) or, using the `file` keyword parameter, a file object such as an in-memory binary file (an `io.BytesIO`). If you pass a file object, `save` does not close it. Similarly, the `MidiFile`

constructor can take either a filename, or a file object by using the `file` keyword parameter. if you pass a file object to `MidiFile` as a context manager, the file is not closed when the context manager exits. Examples can be found in `test_midifiles2.py`.

File Types

There are three types of MIDI files:

- type 0 (single track): all messages are saved in one track
- type 1 (synchronous): all tracks start at the same time
- type 2 (asynchronous): each track is independent of the others

When creating a new file, you can select type by passing the `type` keyword argument, or by setting the `type` attribute:

```
mid = MidiFile(type=2)
mid.type = 1
```

Type 0 files must have exactly one track. A `ValueError` is raised if you attempt to save a file with no tracks or with more than one track.

Playback Length

You can get the total playback time in seconds by accessing the `length` property:

```
mid.length
```

This is only supported for type 0 and 1 files. Accessing `length` on a type 2 file will raise `ValueError`, since it is impossible to compute the playback time of an asynchronous file.

Meta Messages

Meta messages behave like normal messages and can be created in the usual way, for example:

```
>>> from mido import MetaMessage
>>> MetaMessage('key_signature', key='C#', mode='major')
<meta message key_signature key='C#' mode='major' time=0>
```

You can tell meta messages apart from normal messages with:

```
if msg.is_meta:
    ...
```

or if you know the message type you can use the `type` attribute:

```
if msg.type == 'key_signature':
    ...
elif msg.type == 'note_on':
    ...
```

Meta messages can not be sent on ports.

For a list of supported meta messages and their attributes, and also how to implement new meta messages, see *Meta Message Types*.

About the Time Attribute

The `time` attribute is used in several different ways:

- inside a track, it is delta time in ticks. This must be an integer.
- in messages yielded from `play()`, it is delta time in seconds (time elapsed since the last yielded message)
- (only important to implementers) inside certain methods it is used for absolute time in ticks or seconds

Tempo and Beat Resolution

Timing in MIDI files is centered around ticks and beats. A beat is the same as a quarter note. Beats are divided into ticks, the smallest unit of time in MIDI.

Each message in a MIDI file has a delta time, which tells how many ticks have passed since the last message. The length of a tick is defined in ticks per beat. This value is stored as `ticks_per_beat` in MidiFile objects and remains fixed throughout the song.

MIDI Tempo vs. BPM

Unlike music, tempo in MIDI is not given as beats per minute, but rather in microseconds per beat.

The default tempo is 500000 microseconds per beat, which is 120 beats per minute. The meta message 'set_tempo' can be used to change tempo during a song.

You can use `bpm2tempo()` and `tempo2bpm()` to convert to and from beats per minute. Note that `tempo2bpm()` may return a floating point number.

Converting Between Ticks and Seconds

To convert from MIDI time to absolute time in seconds, the number of beats per minute (BPM) and ticks per beat (often called pulses per quarter note or PPQ, for short) have to be decided upon.

You can use `tick2second()` and `second2tick()` to convert to and from seconds and ticks. Note that integer rounding of the result might be necessary because MIDI files require ticks to be integers.

If you have a lot of rounding errors you should increase the time resolution with more ticks per beat, by setting MidiFile.ticks_per_beat to a large number. Typical values range from 96 to 480 but some use even more ticks per beat.

SYX Files

SYX files are used to store SysEx messages, usually for patch data.

Reading and Writing

To read a SYX file:

```
messages = mido.read_syx_file('patch.syx')
```

To write a SYX file:

```
mido.write_syx_file('patch.syx', messages)
```

Non-sysex messages will be ignored.

Plain Text Format

Mido also supports plain text SYX files. These are read in exactly the same way:

```
messages = mido.read_syx_file('patch.txt')
```

`read_syx_file()` determins which format the file is by looking at the first byte. It Raises ValueError if file is plain text and byte is not a 2-digit hex number.

To write plain text:

```
mido.write_syx_file('patch.txt', messages, plaintext=True)
```

This will write the messages as hex encoded bytes with one message per line:

```
F0 00 01 5D 02 00 F7
F0 00 01 5D 03 00 F7
```

Parsing MIDI Bytes

MIDI is a binary protocol. Each each message is encoded as a status byte followed by up to three data bytes. (Sysex messages can have any number of data bytes and use a stop byte instead.)

Note: To parse a single message you can use the class methods `mido.Message.from_bytes()` and `mido.Message.from_hex()` (new in 1.2).

Mido comes with a parser that turns MIDI bytes into messages. You can create a parser object, or call one of the utility functions:

```
>>> mido.parse([0x92, 0x10, 0x20])
<message note_on channel=0 note=16 velocity=32 time=0>

>>> mido.parse_all([0x92, 0x10, 0x20, 0x82, 0x10, 0x20])
[<message note_on channel=2 note=16 velocity=32 time=0>,
 <message note_off channel=2 note=16 velocity=32 time=0>]
```

These functions are just shortcuts for the full `Parser` class. This is the parser used inside input ports to parse incoming messages. Here are a few examples of how it can be used:

```
>>> p = mido.Parser()
>>> p.feed([0x90, 0x10, 0x20])
>>> p.pending()
1
>>> p.get_message()
<message note_on channel=0 note=16 velocity=32 time=0>

>>> p.feed_byte(0x90)
>>> p.feed_byte(0x10)
```

```
>>> p.feed_byte(0x20)
>>> p.feed([0x80, 0x10, 0x20])
<message note_on channel=0 note=16 velocity=32 time=0>
```

`feed()` accepts any iterable that generates integers in 0..255. The parser will skip and stray status bytes or data bytes, so you can safely feed it random data and see what comes out the other end.

`get_message()` will return `None` if there are no messages ready to be gotten.

You can also fetch parsed messages out of the parser by iterating over it:

```
>>> p.feed([0x92, 0x10, 0x20, 0x82, 0x10, 0x20])
>>> for message in p:
...     print(message)
note_on channel=2 note=16 velocity=32 time=0
note_off channel=2 note=16 velocity=32 time=0
```

The messages are available in *p.messages* (a *collections.deque*).

For the full table of MIDI binary encoding, see: http://www.midi.org/techspecs/midimessages.php

String Encoding

Mido messages can be serialized to a text format, which can be used to safely store messages in text files, send them across sockets or embed them in JSON, among other things.

To encode a message, simply call `str()` on it:

```
>>> cc = control_change(channel=9, control=1, value=122, time=60)
>>> str(cc)
'control_change channel=9 control=1 value=122 time=60'
```

To convert the other way (new method in 1.2):

```
>>> mido.Message.from_str('control_change control=1 value=122')
<message control_change channel=0 control=1 value=122 time=0>
```

Alternatively, you can call the `format_as_string` function directly:

```
>>> mido.format_as_string(cc)
'control_change channel=9 control=1 value=122 time=60'
```

If you don't need the time attribute or you want to store it elsewhere, you can pass `include_time=False`:

```
>>> mido.format_as_string(cc)
'control_change channel=9 control=1 value=122'
```

(This option is also available in `mido.Message.from_str()`.)

Format

The format is simple:

```
MESSAGE_TYPE [PARAMETER=VALUE ...]
```

These are the same as the arguments to `mido.Message()`. The order of parameters doesn't matter, but each one can only appear once.

Only these character will ever occur in a string encoded Mido message:

```
[a-z][0-9][ =_.+()]
```

or written out:

```
'abcdefghijklmnopqrstuvwxyz0123456789 =_.+()'
```

This means the message can be embedded in most text formats without any form of escaping.

Parsing

To parse a message, you can use `mido.parse_string()`:

```
>>> parse_string('control_change control=1 value=122 time=0.5')
<message control_change channel=0 control=1 value=122 time=0.5>
```

Parameters that are left out are set to their default values. `ValueError` is raised if the message could not be parsed. Extra whitespace is ignored:

```
>>> parse_string('  control_change   control=1  value=122')
<message control_change channel=0 control=1 value=122 time=0>
```

To parse messages from a stream, you can use `mido.messages.parse_string_stream()`:

```
for (message, error) in parse_string_stream(open('some_music.text')):
    if error:
        print(error)
    else:
        do_something_with(message)
```

This will return every valid message in the stream. If a message could not be parsed, `message` will be `None` and `error` will be an error message describing what went wrong, as well as the line number where the error occurred.

The argument to `parse_string_stream()` can be any object that generates strings when iterated over, such as a file or a list.

`parse_string_stream()` will ignore blank lines and comments (which start with a # and go to the end of the line). An example of valid input:

```
# A very short song with an embedded sysex message.
note_on channel=9 note=60 velocity=120 time=0
# Send some data

sysex data=(1,2,3) time=0.5

pitchwheel pitch=4000  # bend the not a little time=0.7
note_off channel=9 note=60 velocity=60 time=1.0
```

Examples

And example of messages embedded in JSON:

```
{'messages': [
   '0.0 note_on channel=9 note=60 velocity=120',
   '0.5 sysex data=(1,2,3)',
   ...
])
```

Socket Ports - MIDI over TCP/IP

About Socket Ports

Socket ports allow you to send MIDI messages over a computer network.

The protocol is standard MIDI bytes over a TCP stream.

Caveats

The data is sent over an unencrypted channel. Also, the default server allows connections from any host and also accepts arbitrary sysex messages, which could allow anyone to for example overwrite patches on your synths (or worse). Use only on trusted networks.

If you need more security, you can build a custom server with a white list of clients that are allowed to connect.

If timing is critical, latency and jitter (especially on wireless networks) may make socket ports unusable.

Sending Messages to a Server

First, let's import some things:

```
from mido.sockets import PortServer, connect
```

After that, a simple server is only two lines:

```
for message in PortServer('localhost', 8080):
    print(message)
```

You can then connect to the server and send it messages:

```
output = connect('localhost', 8080):
output.send(message)
```

Each end of the connection behaves like a normal Mido I/O port, with all the usual methods.

The host may be a DNS host name or IP address (as a string). It may also be '', in which case connections are accepted on any ip address on the computer.

Turning Things on their Head

If you want the server to send messages the client, you can instead do:

```
server = PortServer('localhost', 8080):
while True:
    server.send(message)
    ...
```

and then on the client side:

```
for message in connect('localhost', 8080):
    print(message)
```

The client will now print any message that the server sends. Each message that the server sends will be received by all connected clients.

Under the Hood

The examples above use the server and client ports as normal I/O ports. This makes it easy to write simple servers, but you don't have any control connections and the way messages are sent and received.

To get more control, you can ignore all the other methods of the `PortServer` object and use only `accept()`. Here's a simple server implemented this way:

```
with PortServer('localhost', 8080) as server:
    while True:
        client = server.accept()
        for message in client:
            print(message)
```

`accept()` waits for a client to connect, and returns a SocketPort object which is connected to the SocketPort object returned by `connect()` at the other end.

The server above has one weakness: it allows only one connection at a time. You can get around this by using `accept(block=False)`. This will return a SocketPort if there is a connection waiting and None if there is connection yet.

Using this, you can write the server any way you like, for example:

```
with PortServer('localhost', 8080) as server:
    clients = []
    while True:
        # Handle connections.
        client = server.accept(block=False)
        if client:
            print('Connection from {}'.format(client.name))
            clients.append(client)

        for i, client in reversed(enumerate(clients)):
            if client.closed:
                print('{} disconnected'.format(client.name))
                del clients[i]

        # Receive messages.
        for client in clients:
            for message in client.iter_pending()
                print('Received {} from {}'.format(message, client))

        # Do other things
        ...
```

Possible Future Additions

Optional HTTP-style headers could be added. As long as these are 7-bit ASCII, they will be counted as data bytes and ignored by clients or servers who don't expect them.

Included Programs

These are installed with Mido.

mido-play

Plays back one or more MIDI files:

```
$ mido-play song1.mid song2.mid
```

mido-ports

Lists available input and output ports and shows environment variables and the current backend module.

mido-serve

Serves one or more ports over the network, for example:

```
$ mido-serve :9080 'Integra-7'
```

You can now connect to this port with `mido-forward` (or use `mido.sockets.connect()` and send messages to it. The messages will be forwarded to every port you listed (in this case 'Integra-7').

mido-connect

Forwards all messages that arrive on one or more ports to a server.

For example, to use the SH-201 keyboard connected to this computer to play sounds on the Integra-7 on a computer named `mac.local` (which runs the server as above), you can do:

```
$ mido-connect mac.local:9080 'SH-201'
```

Note that you may experience latency and jitter, so this may not be very useful for live playing or for playing back songs.

There is also no security built in, so you should only use this on a trusted network. (Anyone can connect and send anything, including harmful sysex messages.)

`mido-serve` and `mido-connect` are only included as fun programs to play with, but may in the future be expanded into something more usable.

Writing a New Port

The Mido port API allows you to write new ports to do practically anything.

A new port type can be defined by subclassing one of the base classes and overriding one or more methods. Here's an example:

```
from mido.ports import BaseOutput

class PrintPort(BaseOutput):
    def _send(message):
        print(message)

>>> port = PrintPort()
>>> port.send(msg)
note_on channel=0 note=0 velocity=64 time=0
```

`_send()` will be called by `send()`, and is responsible for actually sending the message somewhere (or in this case print it out).

Overridable Methods

There are four overridable methods (all of them default to doing nothing):

```
``_open(self, **kwargs)``
```

> Should do whatever is necessary to initialize the port (for example opening a MIDI device.)
>
> Called by `__init__()`. The `name` attribute is already set when `_open()` is called, but you will get the rest of the keyword arguments.
>
> If your port takes a different set of arguments or has other special needs, you can override `__init__()` instead.

`_close(self)`

> Should clean up whatever resources the port has allocated (such as closing a MIDI device).
>
> Called by `close()` if the port is not already closed.

`_send(self, message)`

> (Output ports only.)
>
> Should send the message (or do whatever else that makes sense).
>
> Called by `send()` if the port is open and the message is a Mido message. (You don't need any type checking here.)
>
> Raise IOError if something goes wrong.

`_receive(self, block=True)`

> (Input ports only.)
>
> Should return a message if there is one available.
>
> If `block=True` it should block until a message is available and then return it.
>
> If `block=False` it should return a message or `None` if there is no message yet. If you return `None` the enclosing `pending()` method will check `self._messages` and return one from there.

Note: Prior to 1.2.0 ``_receive() would put messages in self._messages (usually via the parser) and rely on receive() to return them to the user.

Since this was not thread safe the API was changed in 1.2.0 to allow the _receive() to return a message. The old behavior is still supported, so old code will work as before.

Raise IOError if something goes wrong.

Each method corresponds to the public method of the same name, and will be called by that method. The outer method will take care of many things, so the inner method only needs to do the very minimum. The outer method also provides the doc string, so you don't have to worry about that.

The base classes are BaseInput, BaseOutput and BaseIOPort (which is a subclass of the other two.)

Locking

The calls to _receive() and _send() will are protected by a lock, left.lock. As a result all send and receive will be thread safe.

Note: If your _receive() function actually blocks instead of letting the parent class handle it poll() will not work. The two functions are protected by the same lock, so when receive() blocks it will also block other threads calling poll(). In this case you need to implement your own locking.

If you want to implement your own thread safety you can set the _locking attribute in your class:

```
class MyInput(ports.BaseInput):
    _locking = False

    ...
```

An example of this is mido.backends.rtmidi where the callback is used to feed an internal queue that receive() reads from.

Examples

An full example of a device port for the imaginary MIDI library fjopp:

```
import fjopp
from mido.ports import BaseIOPort

# This defines an I/O port.
class FjoppPort(BaseIOPort):
    def _open(self, **kwargs):
        self._device = fjopp.open_device(self.name)

    def _close(self):
        self._device.close()

    def _send(self, message):
        self.device.write(message.bytes())

    def _receive(self, block=True):
        while True:
```

```
            data = self.device.read()
            if data:
                self._parser.feed(data)
            else:
                return
```

If `fjopp` supports blocking read, you can do this to actually block on the device instead of letting `receive()` and friends poll and wait for you:

```
def _receive(self, block=True):
    if block:
        # Actually block on the device.
        # (``read_blocking()`` will always return some data.)
        while not ``self._messages``:
            data = self._device.read_blocking()
            self._parser.feed(data)
    else:
        # Non-blocking read like above.
        while True:
            data = self.device.read()
            if data:
                 self._parser.feed(data)
```

This can be used for any kind of port that wants to block on a pipe, an socket or another input source. Note that Mido will still use polling and waiting when receiving from multiple ports (for example in a `MultiPort`).

If you want separate input and output classes, but the `_open()` and `_close()` methods have a lot in common, you can implement this using a mix-in.

Sometimes it's useful to know inside the methods whether the port supports input or output. The way to do this is to check for the methods `` `send() `` and `receive()`, for example:

```
def _open(self, **kwargs):
    if hasattr(self, 'send'):
        # This is an output port.

    if hasattr(self, 'receive'):
        # This is an input port.

    if hasattr(self, 'send') and hasattr(self, 'receive'):
        # This is an I/O port.
```

Attributes

A port has some attributes that can be useful inside your methods.

`name`

The name of the port. The value is device specific and does not have to be unique. It can have any value, but must be a string or `None`.

This is set by `__init__()`.

`closed`

True if the port is closed. You don't have to worry about this inside your methods.

`_messages`

This is a `collections.deque` of messages that have been read and are ready to be received. This is a shortcut to `_parser.messages`.

`_device_type` (Optional.)

If this attribute exists, it's a string which will be used in `__repr__()`. If it doesn't exist, the class name will be used instead.

Writing a New Backend

A backend is a Python module with one or more of these:

```
Input -- an input port class
Output -- an output port class
IOPort -- an I/O port class
get_devices() -- returns a list of devices
```

Once written, the backend can be used by setting the environment variable `MIDO_BACKEND` or by calling `mido.set_backend()`. In both cases, the path of the module is used.

`Input`

And input class for `open_input()`. This is only required if the backend supports input.

`Output`

And output class for `open_output()`. This is only required if the backend supports output.

`IOPort`

An I/O port class for `open_ioport()`. If this is not found, `open_ioport()` will return `mido.ports.IOPort(Input(), Output())`.

`get_devices(**kwargs)`

Returns a list of devices, where each device is dictionary with at least these three values:

```
{
  'name': 'Some MIDI Input Port',
  'is_input': True,
  'is_output': False,
}
```

These are used to build return values for `get_input_names()` etc.. This function will also be available to the user directly.

For examples, see `mido/backends/`.

Freezing to EXE File

PyInstaller

When you build an executable with PyInstaller and run it you may get import errors like this one:

```
ImportError: No module named mido.backends.portmidi
```

The reason is that Mido uses `import_module()` to import the backend modules, while PyInstaller looks for `import` statements.

The easiest fix is to import the module at the top of the program:

```
import mido
import mido.backends.portmidi  # The backend you want to use.
print(mido.get_input_names())
```

and then run `pyinstaller` like usual:

```
$ pyinstaller --onefile midotest.py
$ ./dist/midotest
[u'Midi Through Port-0']
```

If you don't want to change the program, you can instead declare the backend module as a hidden import.

bbFreeze, py2exe, cx_Freeze, py2app, etc.

I suspect the same is true for these, but I have not had a chance to try it out yet.

Adding the explicit `import` statement should always work, though, since Mido backends are just normal Python modules.

About MIDI

A Short Introduction To MIDI

MIDI is a simple binary protocol for communicating with synthesizers and other electronic music equipment.

It was developed in 1981 by Dave Smith and Chet Wood of Sequential Systems. MIDI was quickly embraced by all the major synth manufacturers and led to developments such as microcomputer sequencers, and with them the electronic home studio. Although many attempts have been made to replace it, it is still the industry standard.

MIDI was designed for the 8-bit micro controllers found in synthesizers at the beginning of the 80's. As such, it is a very minimal byte-oriented protocol. The message for turning a note on is only three bytes long (here shown in hexadecimal):

```
92 3C 64
```

This message consists of:

```
92 -- 9 == message type note on
      2 == channel 2

3C -- note 60 (middle C)

64 -- velocity (how hard the note is hit)
```

The first byte is called a status byte. It has the upper bit set, which is how you can tell it apart from the following data bytes. Data bytes are thus only 7 bits (0..127).

Each message type has a given number of data bytes, the exception being the System Exclusive message which has a start and a stop byte and any number of data bytes in-between these two:

```
F0 ... F7
```

Messages can be divided into four groups:

- Channel messages. These are used to turn notes on and off, to change patches, and change controllers (pitch bend, modulation wheel, pedal and many others). There are 16 channels, and the channel number is encoded in the lower 4 bits of the status byte. Each synth can choose which channel (or channels) it responds to. This can typically be configured.
- System common messages.
- System real time messages, the include start, stop, continue, song position (for playback of songs) and reset.
- System Exclusive messages (often called Sysex messages). These are used for sending and receiving device specific such as patch data.

Some Examples of Messages

```
# Turn on middle C on channel 2:
92 3C 64

# Turn it back off:
82 3C 64

# Change to program (sound) number 4 on channel 2:
C2 04

# Continue (Starts a song that has been paused):
FB

# Sysex data request for the Roland SH-201 synthesizer:
F0 41 10 00 00 16 11 20 00 00 00 00 00 00 21 3F F7
```

Further Reading

- An Introduction to MIDI
- MIDI Basics (by Yamaha)
- Wikipedia's page on MIDI
- MIDI Manufacturers Association
- A full table of MIDI messages
- Essentials of the MIDI protocol

Message Types

Supported Messages

Name	Keyword Arguments / Attributes
note_off	channel note velocity
note_on	channel note velocity
polytouch	channel note value
control_change	channel control value
program_change	channel program
aftertouch	channel value
pitchwheel	channel pitch
sysex	data
quarter_frame	frame_type frame_value
songpos	pos
song_select	song
tune_request	
clock	
start	
continue	
stop	
active_sensing	
reset	

`quarter_frame` is used for SMPTE time codes. See: http://www.electronics.dit.ie/staff/tscarff/Music_technology/midi/MTC.htm

Parameter Types

Name	Valid Range	Default Value
channel	0..15	0
frame_type	0..7	0
frame_value	0..15	0
control	0..127	0
note	0..127	0
program	0..127	0
song	0..127	0
value	0..127	0
velocity	0..127	64
data	(0..127, 0..127, ...)	() (empty tuple)
pitch	-8192..8191	0
pos	0..16383	0
time	any integer or float	0

Note: Mido numbers channels 0 to 15 instead of 1 to 16. This makes them easier to work with in Python but you may want to add and subtract 1 when communicating with the user.

`velocity` is how fast the note was struck or released. It defaults to 64 so that if you don't set it, you will still get a reasonable value. (64 is the recommended default for devices that don't support it attack or release velocity.)

The `time` parameter is not included in the encoded message, and is (currently) not used by Mido in any way. You can use it for whatever purpose you wish.

The `data` parameter accepts any iterable that generates numbers in 0..127. This includes:

```
mido.Message('sysex', data=[1, 2, 3])
mido.Message('sysex', data=range(10))
mido.Message('sysex', data=(i for i in range(10) if i % 2 == 0))
```

For details about the binary encoding of a MIDI message, see:

http://www.midi.org/techspecs/midimessages.php

Meta Message Types

Supported Messages

sequence_number (0x00)

Attribute	Values	Default
number	0..65535	0

Sequence number in type 0 and 1 MIDI files; pattern number in type 2 MIDI files.

text (0x01)

Attribute	Values	Default
text	string	''

General "Text" Meta Message. Can be used for any text based data.

copyright (0x02)

Attribute	Values	Default
text	string	''

Provides information about a MIDI file's copyright.

track_name (0x03)

Attribute	Values	Default
name	string	''

Stores a MIDI track's name.

instrument_name (0x04)

Attribute	Values	Default
name	string	''

Stores an instrument's name.

lyrics (0x05)

Attribute	Values	Default
text	string	''

Stores the lyrics of a song. Typically one syllable per Meta Message.

marker (0x06)

Attribute	Values	Default
text	string	''

Marks a point of interest in a MIDI file. Can be used as the marker for the beginning of a verse, solo, etc.

cue_marker (0x07)

Attribute	Values	Default
text	string	''

Marks a cue. IE: 'Cue performer 1', etc

device_name (0x09)

Attribute	Values	Default
name	string	''

Gives the name of the device.

channel_prefix (0x20)

Attribute	Values	Default
channel	0..255	0

Gives the prefix for the channel on which events are played.

midi_port (0x21)

Attribute	Values	Default
port	0..255	0

Gives the MIDI Port on which events are played.

end_of_track (0x2f)

Attribute	Values	Default
n/a	n/a	n/a

An empty Meta Message that marks the end of a track.

set_tempo (0x51)

Attribute	Values	Default
tempo	0..16777215	500000

Tempo is in microseconds per beat (quarter note). You can use `bpm2tempo()` and `tempo2bpm()` to convert to and from beats per minute. Note that `tempo2bpm()` may return a floating point number.

smpte_offset (0x54)

Attribute	Values	Default
frame_rate	24, 25, 29.97, 30	24
hours	0..255	0
minutes	0..59	0
seconds	0..59	0
frames	0..255	0
sub_frames	0..99	0

time_signature (0x58)

Attribute	Values	Default
numerator	0..255	4
denominator	1..2**255	4
clocks_per_click	0..255	24
notated_32nd_notes_per_beat	0..255	8

Time signature of:

4/4 : MetaMessage('time_signature', numerator=4, denominator=4)

3/8 : MetaMessage('time_signature', numerator=3, denominator=8)

key_signature (0x59)

Attribute	Values	Default
key	'C', 'F#m', ...	'C'

Valid values: A A#m Ab Abm Am B Bb Bbm Bm C C# C#m Cb Cm D D#m Db Dm E Eb Ebm Em F F# F#m Fm G G#m Gb Gm

Note: the mode attribute was removed in 1.1.5. Instead, an 'm' is appended to minor keys.

sequencer_specific (0x7f)

Attribute	Values	Default
data	[..]	[]

An unprocessed sequencer specific message containing raw data.

Unknown Meta Messages

Unknown meta messages will be returned as `UnknownMetaMessage` objects, with `type` set to `unknown_meta`. The messages are saved back to the file exactly as they came out.

Code that depends on `UnknownMetaMessage` may break if the message in question is ever implemented, so it's best to only use these to learn about the format of the new message and then implement it as described below.

`UnknownMetaMessage` have two attributes:

```
``type_byte`` - a byte which uniquely identifies this message type
``_data`` - the message data as a list of bytes
```

These are also visible in the `repr()` string:

```
<unknown meta message _type_byte=0x## _data=[...], time=0>
```

Implementing New Meta Messages

If you come across a meta message which is not implemented, or you want to use a custom meta message, you can add it by writing a new meta message spec:

```
from mido.midifiles import MetaSpec, add_meta_spec

class MetaSpec_light_color(MetaSpec):
    type_byte = 0xf0
    attributes = ['r', 'g', 'b']
    defaults = [0, 0, 0]

def decode(self, message, data):
    # Interpret the data bytes and assign them to attributes.
    (message.r, message.g, message.b) = data

def encode(self, message):
    # Encode attributes to data bytes and
    # return them as a list of ints.
    return [message.r, message.g, message.b]

def check(self, name, value):
    # (Optional)
    # This is called when the user assigns
    # to an attribute. You can use this for
    # type and value checking. (Name checking
    # is already done.
    #
    # If this method is left out, no type and
    # value checking will be done.

    if not isinstance(value, int):
        raise TypeError('{} must be an integer'.format(name))

    if not 0 <= value <= 255:
        raise TypeError('{} must be in range 0..255'.format(name))
```

Then you can add your new message type with:

```
add_meta_spec(MetaSpec_light_color)
```

and create messages in the usual way:

```
>>> from mido import MetaMessage
>>> MetaMessage('light_color', r=120, g=60, b=10)
<meta message light_color r=120 g=60 b=10 time=0>
```

and the new message type will now work when reading and writing MIDI files.

Some additional functions are available:

```
encode_string(unicode_string)
decode_string(byte_list)
```

These convert between a Unicode string and a list of bytes using the current character set in the file.

If your message contains only one string with the attribute name `text` or `name`, you can subclass from one of the existing messages with these attributes, for example:

```
class MetaSpec_copyright(MetaSpec_text):
    type_byte = 0x02

class MetaSpec_instrument_name(MetaSpec_track_name):
    type_byte = 0x04
```

This allows you to skip everything but `type_byte`, since the rest is inherited.

See the existing MetaSpec classes for further examples.

Library Reference

Messages

class `mido.`**`Message`**(*type*, ***args*)

`bin`()
Encode message and return as a bytearray.

This can be used to write the message to a file.

`bytes`()
Encode message and return as a list of integers.

`copy`(***overrides*)
Return a copy of the message.

Attributes will be overridden by the passed keyword arguments. Only message specific attributes can be overridden. The message type can not be changed.

`dict`()
Returns a dictionary containing the attributes of the message.

Example: {'type': 'sysex', 'data': [1, 2], 'time': 0}

Sysex data will be returned as a list.

classmethod **from_bytes**(*cl*, *data*, *time=0*)
Parse a byte encoded message.

Accepts a byte string or any iterable of integers.

This is the reverse of msg.bytes() or msg.bin().

from_dict(*cl*, *data*)
Create a message from a dictionary.

Only "type" is required. The other will be set to default values.

classmethod **from_hex**(*cl*, *text*, *time=0*, *sep=None*)
Parse a hex encoded message.

This is the reverse of msg.hex().

classmethod **from_str**(*cl*, *text*)
Parse a string encoded message.

This is the reverse of str(msg).

hex(*sep=' '*)
Encode message and return as a string of hex numbers,

Each number is separated by the string sep.

is_meta = False

is_realtime
True if the message is a system realtime message.

Ports

mido.**open_input**(*self*, *name=None*, *virtual=False*, *callback=None*, ***kwargs*)
Open an input port.

If the environment variable MIDO_DEFAULT_INPUT is set, if will override the default port.

virtual=False Passing True opens a new port that other applications can connect to. Raises IOError if not supported by the backend.

callback=None A callback function to be called when a new message arrives. The function should take one argument (the message). Raises IOError if not supported by the backend.

mido.**open_output**(*self*, *name=None*, *virtual=False*, *autoreset=False*, ***kwargs*)
Open an output port.

If the environment variable MIDO_DEFAULT_OUTPUT is set, if will override the default port.

virtual=False Passing True opens a new port that other applications can connect to. Raises IOError if not supported by the backend.

autoreset=False Automatically send all_notes_off and reset_all_controllers on all channels. This is the same as calling *port.reset()*.

mido.**open_ioport**(*self*, *name=None*, *virtual=False*, *callback=None*, *autoreset=False*, ***kwargs*)
Open a port for input and output.

If the environment variable MIDO_DEFAULT_IOPORT is set, if will override the default port.

virtual=False Passing True opens a new port that other applications can connect to. Raises IOError if not supported by the backend.

callback=None A callback function to be called when a new message arrives. The function should take one argument (the message). Raises IOError if not supported by the backend.

autoreset=False Automatically send all_notes_off and reset_all_controllers on all channels. This is the same as calling *port.reset()*.

mido.**get_input_names**(*self*, ***kwargs*)
Return a sorted list of all input port names.

mido.**get_output_names**(*self*, ***kwargs*)
Return a sorted list of all output port names.

mido.**get_ioport_names**(*self*, ***kwargs*)
Return a sorted list of all I/O port names.

Backends

mido.**set_backend**(*name=None*, *load=False*)
Set current backend.

name can be a module name like 'mido.backends.rtmidi' or a Backend object.

If no name is passed, the default backend will be used.

This will replace all the open_*() and get_*_name() functions in top level mido module. The module will be loaded the first time one of those functions is called.

class mido.**Backend**(*name=None*, *api=None*, *load=False*, *use_environ=True*)
Wrapper for backend module.

A backend module implements classes for input and output ports for a specific MIDI library. The Backend object wraps around the object and provides convenient 'open_*()' and 'get_*_names()' functions.

get_input_names(***kwargs*)
Return a sorted list of all input port names.

get_ioport_names(***kwargs*)
Return a sorted list of all I/O port names.

get_output_names(***kwargs*)
Return a sorted list of all output port names.

load()
Load the module.

Does nothing if the module is already loaded.

This function will be called if you access the 'module' property.

loaded
Return True if the module is loaded.

module
A reference module implementing the backend.

This will always be a valid reference to a module. Accessing this property will load the module. Use .loaded to check if the module is loaded.

open_input(*name=None*, *virtual=False*, *callback=None*, ***kwargs*)
Open an input port.

If the environment variable MIDO_DEFAULT_INPUT is set, if will override the default port.

virtual=False Passing True opens a new port that other applications can connect to. Raises IOError if not supported by the backend.

callback=None A callback function to be called when a new message arrives. The function should take one argument (the message). Raises IOError if not supported by the backend.

open_ioport(*name=None*, *virtual=False*, *callback=None*, *autoreset=False*, ***kwargs*)
Open a port for input and output.

If the environment variable MIDO_DEFAULT_IOPORT is set, if will override the default port.

virtual=False Passing True opens a new port that other applications can connect to. Raises IOError if not supported by the backend.

callback=None A callback function to be called when a new message arrives. The function should take one argument (the message). Raises IOError if not supported by the backend.

autoreset=False Automatically send all_notes_off and reset_all_controllers on all channels. This is the same as calling *port.reset()*.

open_output(*name=None*, *virtual=False*, *autoreset=False*, ***kwargs*)
Open an output port.

If the environment variable MIDO_DEFAULT_OUTPUT is set, if will override the default port.

virtual=False Passing True opens a new port that other applications can connect to. Raises IOError if not supported by the backend.

autoreset=False Automatically send all_notes_off and reset_all_controllers on all channels. This is the same as calling *port.reset()*.

Parsing

mido.**parse**(*data*)
Parse MIDI data and return the first message found.

Data after the first message is ignored. Use parse_all() to parse more than one message.

mido.**parse_all**(*data*)
Parse MIDI data and return a list of all messages found.

This is typically used to parse a little bit of data with a few messages in it. It's best to use a Parser object for larger amounts of data. Also, tt's often easier to use parse() if you know there is only one message in the data.

class mido.**Parser**(*data=None*)
MIDI Parser

Parses a stream of bytes and produces messages.

Data can be put into the parser in the form of integers, byte arrays or byte strings.

feed(*data*)
Feed MIDI data to the parser.

Accepts any object that produces a sequence of integers in range 0..255, such as:

[0, 1, 2] (0, 1, 2) [for i in range(256)] (for i in range(256)] bytearray() b" # Will be converted to integers in Python 2.

feed_byte(*byte*)
Feed one MIDI byte into the parser.

The byte must be an integer in range 0..255.

get_message()
Get the first parsed message.

Returns None if there is no message yet. If you don't want to deal with None, you can use pending() to see how many messages you can get before you get None, or just iterate over the parser.

pending()
Return the number of pending messages.

MIDI Files

class mido.**MidiFile**(*filename=None*, *file=None*, *type=1*, *ticks_per_beat=480*, *charset='latin1'*, *debug=False*)

add_track(*name=None*)
Add a new track to the file.

This will create a new MidiTrack object and append it to the track list.

length
Playback time in seconds.

This will be computed by going through every message in every track and adding up delta times.

play(*meta_messages=False*)
Play back all tracks.

The generator will sleep between each message by default. Messages are yielded with correct timing. The time attribute is set to the number of seconds slept since the previous message.

By default you will only get normal MIDI messages. Pass meta_messages=True if you also want meta messages.

You will receive copies of the original messages, so you can safely modify them without ruining the tracks.

print_tracks(*meta_only=False*)
Prints out all messages in a .midi file.

May take argument meta_only to show only meta messages.

Use: print_tracks() -> will print all messages print_tracks(meta_only=True) -> will print only MetaMessages

save(*filename=None*, *file=None*)
Save to a file.

If file is passed the data will be saved to that file. This is typically an in-memory file or and already open file like sys.stdout.

If filename is passed the data will be saved to that file.

Raises ValueError if both file and filename are None, or if a type 0 file has != one track.

class mido.**MidiTrack**

append()
L.append(object) – append object to end

copy()

count(*value*) → integer – return number of occurrences of value

extend()
L.extend(iterable) – extend list by appending elements from the iterable

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

insert()
L.insert(index, object) – insert object before index

name
Name of the track.

This will return the name from the first track_name meta message in the track, or '' if there is no such message.

Setting this property will update the name field of the first track_name message in the track. If no such message is found, one will be added to the beginning of the track with a delta time of 0.

pop([*index*]) → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

remove()
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

reverse()
L.reverse() – reverse *IN PLACE*

sort()
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

class mido.**MetaMessage**(*type*, ***kwargs*)

bin()
Encode message and return as a bytearray.

This can be used to write the message to a file.

bytes()

copy(***overrides*)
Return a copy of the message

Attributes will be overridden by the passed keyword arguments. Only message specific attributes can be overridden. The message type can not be changed.

dict()
Returns a dictionary containing the attributes of the message.

Example: {'type': 'sysex', 'data': [1, 2], 'time': 0}

Sysex data will be returned as a list.

from_dict(*cl*, *data*)
Create a message from a dictionary.

Only "type" is required. The other will be set to default values.

hex(*sep=' '*)
Encode message and return as a string of hex numbers,

Each number is separated by the string sep.

is_meta = True

is_realtime
True if the message is a system realtime message.

mido.**tick2second**(*tick*, *ticks_per_beat*, *tempo*)
Convert absolute time in ticks to seconds.

Returns absolute time in seconds for a chosen MIDI file time resolution (ticks per beat, also called PPQN or pulses per quarter note) and tempo (microseconds per beat).

mido.**second2tick**(*second*, *ticks_per_beat*, *tempo*)
Convert absolute time in seconds to ticks.

Returns absolute time in ticks for a chosen MIDI file time resolution (ticks per beat, also called PPQN or pulses per quarter note) and tempo (microseconds per beat).

mido.**bpm2tempo**(*bpm*)
Convert beats per minute to MIDI file tempo.

Returns microseconds per beat as an integer:

```
240 => 250000
120 => 500000
60 => 1000000
```

mido.**tempo2bpm**(*tempo*)
Convert MIDI file tempo to BPM.

Returns BPM as an integer or float:

```
250000 => 240
500000 => 120
1000000 => 60
```

mido.**merge_tracks**(*tracks*)
Returns a MidiTrack object with all messages from all tracks.

The messages are returned in playback order with delta times as if they were all in one track.

SYX Files

mido.**read_syx_file**(*filename*)
Read sysex messages from SYX file.

Returns a list of sysex messages.

This handles both the text (hexadecimal) and binary formats. Messages other than sysex will be ignored. Raises ValueError if file is plain text and byte is not a 2-digit hex number.

mido.**write_syx_file**(*filename*, *messages*, *plaintext=False*)
Write sysex messages to a SYX file.

Messages other than sysex will be skipped.

By default this will write the binary format. Pass `plaintext=True` to write the plain text format (hex encoded ASCII text).

Port Classes and Functions

class mido.ports.**BaseInput**(*name=u''*, ***kwargs*)
Base class for input port.

Subclass and override _receive() to create a new input port type. (See portmidi.py for an example of how to do this.)

close()
Close the port.

If the port is already closed, nothing will happen. The port is automatically closed when the object goes out of scope or is garbage collected.

is_input = True

is_output = False

iter_pending()
Iterate through pending messages.

poll()
Receive the next pending message or None

This is the same as calling *receive(block=False)*.

receive(*block=True*)
Return the next message.

This will block until a message arrives.

If you pass block=False it will not block and instead return None if there is no available message.

If the port is closed and there are no pending messages IOError will be raised. If the port closes while waiting inside receive(), IOError will be raised. Todo: this seems a bit inconsistent. Should different errors be raised? What's most useful here?

class mido.ports.**BaseOutput**(*name=u''*, *autoreset=False*, ***kwargs*)
Base class for output port.

Subclass and override _send() to create a new port type. (See portmidi.py for how to do this.)

close()
Close the port.

If the port is already closed, nothing will happen. The port is automatically closed when the object goes out of scope or is garbage collected.

is_input = False

is_output = True

panic()
Send "All Sounds Off" on all channels.

This will mute all sounding notes regardless of envelopes. Useful when notes are hanging and nothing else helps.

reset()
Send "All Notes Off" and "Reset All Controllers" on all channels

send(*msg*)
Send a message on the port.

A copy of the message will be sent, so you can safely modify the original message without any unexpected consequences.

class mido.ports.**IOPort**(*input*, *output*)
Input / output port.

This is a convenient wrapper around an input port and an output port which provides the functionality of both. Every method call is forwarded to the appropriate port.

close()
Close the port.

If the port is already closed, nothing will happen. The port is automatically closed when the object goes out of scope or is garbage collected.

is_input = True

is_output = True

iter_pending()
Iterate through pending messages.

panic()
Send "All Sounds Off" on all channels.

This will mute all sounding notes regardless of envelopes. Useful when notes are hanging and nothing else helps.

poll()
Receive the next pending message or None

This is the same as calling *receive(block=False)*.

receive(*block=True*)
Return the next message.

This will block until a message arrives.

If you pass block=False it will not block and instead return None if there is no available message.

If the port is closed and there are no pending messages IOError will be raised. If the port closes while waiting inside receive(), IOError will be raised. Todo: this seems a bit inconsistent. Should different errors be raised? What's most useful here?

reset()
Send "All Notes Off" and "Reset All Controllers" on all channels

send(*msg*)
Send a message on the port.

A copy of the message will be sent, so you can safely modify the original message without any unexpected consequences.

class mido.ports.**MultiPort**(*ports*, *yield_ports=False*)

close()
Close the port.

If the port is already closed, nothing will happen. The port is automatically closed when the object goes out of scope or is garbage collected.

is_input = True

is_output = True

iter_pending()
Iterate through pending messages.

panic()
Send "All Sounds Off" on all channels.

This will mute all sounding notes regardless of envelopes. Useful when notes are hanging and nothing else helps.

poll()
Receive the next pending message or None

This is the same as calling *receive(block=False)*.

receive(*block=True*)
Return the next message.

This will block until a message arrives.

If you pass block=False it will not block and instead return None if there is no available message.

If the port is closed and there are no pending messages IOError will be raised. If the port closes while waiting inside receive(), IOError will be raised. Todo: this seems a bit inconsistent. Should different errors be raised? What's most useful here?

reset()
Send "All Notes Off" and "Reset All Controllers" on all channels

send(*msg*)
Send a message on the port.

A copy of the message will be sent, so you can safely modify the original message without any unexpected consequences.

mido.ports.**multi_receive**(*ports*, *yield_ports=False*, *block=True*)
Receive messages from multiple ports.

Generates messages from ever input port. The ports are polled in random order for fairness, and all messages from each port are yielded before moving on to the next port.

If yield_ports=True, (port, message) is yielded instead of just the message.

If block=False only pending messages will be yielded.

mido.ports.**multi_iter_pending**(*ports*, *yield_ports=False*)
Iterate through all pending messages in ports.

This is the same as calling multi_receive(ports, block=False). The function is kept around for backwards compatability.

mido.ports.**multi_send**(*ports*, *msg*)
Send message on all ports.

mido.ports.**sleep**()
Sleep for N seconds.

This is used in ports when polling and waiting for messages. N can be set with set_sleep_time().

mido.ports.**set_sleep_time**(*seconds=0.001*)
Set the number of seconds sleep() will sleep.

mido.ports.**get_sleep_time**()
Get number of seconds sleep() will sleep.

mido.ports.**panic_messages**()
Yield "All Sounds Off" for all channels.

This will mute all sounding notes regardless of envelopes. Useful when notes are hanging and nothing else helps.

mido.ports.**reset_messages**()
Yield "All Notes Off" and "Reset All Controllers" for all channels

Socket Ports

class mido.sockets.**PortServer**(*host*, *portno*, *backlog=1*)

accept(*block=True*)
Accept a connection from a client.

Will block until there is a new connection, and then return a SocketPort object.

If block=False, None will be returned if there is no new connection waiting.

close()
Close the port.

If the port is already closed, nothing will happen. The port is automatically closed when the object goes out of scope or is garbage collected.

is_input = True

is_output = True

iter_pending()
Iterate through pending messages.

panic()
Send "All Sounds Off" on all channels.

This will mute all sounding notes regardless of envelopes. Useful when notes are hanging and nothing else helps.

poll()
Receive the next pending message or None

This is the same as calling *receive(block=False)*.

receive(*block=True*)
Return the next message.

This will block until a message arrives.

If you pass block=False it will not block and instead return None if there is no available message.

If the port is closed and there are no pending messages IOError will be raised. If the port closes while waiting inside receive(), IOError will be raised. Todo: this seems a bit inconsistent. Should different errors be raised? What's most useful here?

reset()
Send "All Notes Off" and "Reset All Controllers" on all channels

send(*msg*)
Send a message on the port.

A copy of the message will be sent, so you can safely modify the original message without any unexpected consequences.

class mido.sockets.**SocketPort**(*host*, *portno*, *conn=None*)

close()
Close the port.

If the port is already closed, nothing will happen. The port is automatically closed when the object goes out of scope or is garbage collected.

is_input = True

is_output = True

iter_pending()
Iterate through pending messages.

panic()
Send "All Sounds Off" on all channels.

This will mute all sounding notes regardless of envelopes. Useful when notes are hanging and nothing else helps.

poll()
Receive the next pending message or None

This is the same as calling *receive(block=False)*.

receive(*block=True*)
Return the next message.

This will block until a message arrives.

If you pass block=False it will not block and instead return None if there is no available message.

If the port is closed and there are no pending messages IOError will be raised. If the port closes while waiting inside receive(), IOError will be raised. Todo: this seems a bit inconsistent. Should different errors be raised? What's most useful here?

reset()
Send "All Notes Off" and "Reset All Controllers" on all channels

send(*msg*)
Send a message on the port.

A copy of the message will be sent, so you can safely modify the original message without any unexpected consequences.

mido.sockets.**parse_address**(*address*)
Parse and address on the format host:port.

Returns a tuple (host, port). Raises ValueError if format is invalid or port is not an integer or out of range.

Frozen Messages

mido.frozen.**freeze_message**(*msg*)
Freeze message.

Returns a frozen version of the message. Frozen messages are immutable, hashable and can be used as dictionary keys.

Will return None if called with None. This allows you to do things like:

```
msg = freeze_message(port.poll())
```

mido.frozen.**thaw_message**(*msg*)
Thaw message.

Returns a mutable version of a frozen message.

Will return None if called with None.

mido.frozen.**is_frozen**(*msg*)
Return True if message is frozen, otherwise False.

class mido.frozen.**Frozen**

class mido.frozen.**FrozenMessage**(*type*, ***args*)

class mido.frozen.**FrozenMetaMessage**(*type*, ***kwargs*)

class mido.frozen.**FrozenUnknownMetaMessage**(*type_byte*, *data=None*, *time=0*)

Resources

- MIDI Manufacturers Association (midi.org)
- Table of MIDI Messages (midi.org)
- Tech Specs & Info (midi.org)
- MIDI (Wikipedia)
- Essentials of the MIDI Protocol (Craig Stuart Sapp, CCRMA)
- Outline of the Standard MIDI File Structure (Craig Stuart Sapp, CCRMA)
- Active Sense (About the active sensing message.)
- Active Sensing (Sweetwater)
- MIDI Technical/Programming Docs (Jeff Glatt)
- Standard MIDI Files (cnx.org)
- MIDI File Parsing (Course assignment in Music 253 at Stanford University)
- MIDI File Format (The Sonic Spot)
- Delta time and running status (mic at recordingblogs.com)
- MIDI meta messages (recordingblog.com)
- Meta Message (Sound On Sound)

License

Authors

Ole Martin Bjørndalen (lead programmer)

Rapolas Binkys

Acknowledgements

Thanks to /u/tialpoy/ on Reddit for extensive code review and helpful suggestions.

Thanks to everyone who has sent bug reports and patches.

The PortMidi wrapper is based on portmidizero by Grant Yoshida.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

L

M

N

O

P

R

S

T

W