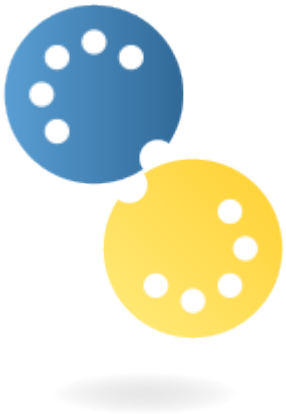

Mido Documentation

Release 1.3.2

Ole Martin Bjørndalen, Raphaël Doursenaud

Dec 15, 2023

TABLE OF CONTENTS

Part I

Introduction

CHAPTER

ONE

OVERVIEW

Mido is a *Python* library for working with *MIDI* 1.0 *ports*, *messages* and *files*:

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

```
port = mido.open_output('Port Name')
port.send(msg)
```

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

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

Mido is short for *MIDI objects*.

1.1 About this document

This document refers to Mido version 1.3.2.

Note: An up-to-date version of this document is always available at https://mido.readthedocs.io.

1.2 License

See also:

Licenses (page 88)

1.3 Community & Source Code

Come visit us at https://github.com/mido.

Everybody is welcome!

See also:

- *Code of Conduct* (page 59)
- *Contributing* (page 61)

Part II

Basics

1.3.1 Installing

1.3.1.1 Requirements

Mido requires *Python* version 3.7 or higher.

A few dependencies are also required in order to allow Mido to introspect its own version:

- *packaging*
- *importlib_metadata* for *Python* < 3.8

Note: Dependency management is handled automatically when installing using the recommended methods. No need to bother installing these manually.

1.3.1.2 Optional

Dependencies for the loaded on-demand *port backend(s)* are optional unless you want to use the *ports* feature.

See *Backends* (page 19) for help choosing a *backend*.

1.3.1.3 Installation

The recommended installation method is to use *pip* to retrieve the package from *PyPi*.

Note: Consider using a *virtual environment* to isolate your installation from your current environment.

This ensures that you always get the latest released stable version:

```
python3 -m pip install mido
```

Or, alternatively, if you want to use *ports* with the default *backend*:

```
python3 -m pip install mido[ports-rtmidi]
```

See *Backends* (page 19) for installation instructions for other *backends*.

1.3.2 Introduction (Basic Concepts)

Mido is all about messages, ports and files.

1.3.2.1 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 from 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* (page 70).

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=0, 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* (page 13).

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.

1.3.2.2 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* (page 26).

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 allows 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 ports/custom.

Virtual Ports

Virtual ports allows 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`".

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

Furthermore, RtMidi's virtual ports are not available under Microsoft Windows. See: *RtMidi (Default, Recommended)* (page 20) for details.

1.3.2.3 Raw MIDI Bytes Parser

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 messages/parsing.

New in version 1.2.

You can also create a message from `bytes` using class methods:

```
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.

Part III

Details

1.3.3 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 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* (page 70).

1.3.3.1 Control Changes

```
if msg.is_cc():
    print('Control change message received')

if msg.is_cc(7):
    print('Volume changed to', msg.value)
```

1.3.3.2 Converting To & From Bytes

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'
```

From Bytes

You can turn `bytes` back into messages with the parser.

New in version 1.2.

You can also create a message from `bytes` using class methods:

```
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.

1.3.3.3 The Time Attribute

Each message has a `time` attribute, which can be set to any value of type `int` or `float`.

Some parts of Mido use the attribute for special purposes. In `MIDI file` tracks, it is used as delta time (in *ticks*), and it must be a non-negative integer.

In other parts of Mido, this value is ignored.

Changed in version 1.1.18: In earlier versions, the `time` attribute was not included in comparisons. If you want the old behavior the easiest 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'))
```

1.3.3.4 System Exclusive Messages

System Exclusive (aka *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 between *0* and *127* 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)
```

1.3.3.5 Frozen Messages

New in version 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())

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

To check if a message is *frozen*:

```
from mido.frozen import is_frozen

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

1.3.3.6 Parsing MIDI Bytes

The MIDI protocol is a *binary protocol*. Each message is encoded as a *status* byte followed by up to three *data* bytes. (Except *SysEx* messages which can have an arbitrary number of *data* bytes immediately followed by an EOX status byte.)

New in version 1.2: `mido.Message.from_hex()`

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

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=2, 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 same parser as 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)
```

(continues on next page)

(continued from previous page)

```
>>> p.feed([0x80, 0x10, 0x20])
>>> p.pending()
2
>>> p.get_message()
Message('note_on', channel=0, note=16, velocity=32, time=0)
>>> p.get_message()
Message('note_off', 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`).

1.3.3.7 Serializing

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 characters 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

An example of messages embedded into JSON:

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

1.3.4 Backends

A backend provides the interface between Mido and the operating system level MIDI stack.

Some Mido features are only available with select backends.

Mido's backend subsystem has been designed to be extensible so you can add your own backends if required. See custom.

Providing platform specific Python-native backends is currently evaluated. See: https://github.com/mido/mido/issues/506

Todo: Insert a stack diagram to clear things up.

1.3.4.1 Choice

Mido comes with five backends:

- *RtMidi* (page 20) is the *default* and *recommended* backend. It has all the features of the other ones and more plus it is usually easier to install.
- *PortMidi* (page 22) was the default backend up until version 1.2. It uses the `portmidi` shared library and can be difficult to install on some systems.
- *Pygame* (page 23) uses the `pygame.midi` module.
- *rtmidi-python* (page 24) uses the `rtmidi_python` package, an alternative wrapper for PortMidi. It is currently very basic but easier to install on some Windows systems.
- *Amidi* (page 24) is an experimental backend for Linux/ALSA that uses the command `amidi` to send and receive messages.

You can set the backend using an environment variable: See *Environment Variables* (page 20).

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

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

Note: 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.

1.3.4.2 Environment Variables

Select Backend

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

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

Set Default ports

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' python3 program.py
```

or:

```
$ export MIDO_DEFAULT_OUTPUT='Integra-7'
$ python3 program1.py
$ python3 program2.py
```

1.3.4.3 Available Backends

RtMidi (Default, Recommended)

Name: `mido.backends.rtmidi`

Resources:

- python-rtmidi Python Library (https://pypi.org/project/python-rtmidi/)
- RtMidi C Library (https://www.music.mcgill.ca/~gary/rtmidi/)

The RtMidi backend is a thin wrapper around python-rtmidi (https://pypi.org/project/python-rtmidi/).

Features

- callbacks
- true blocking `receive()` in Python 3 (using a *callback* and a *queue*)
- virtual ports (Except on Microsoft Windows)
- ports can be opened multiple times, each will receive a copy of all messages
- a *client name* can be specified when opening a virtual port
- sends but doesn't receive active sensing (By default)
- 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 configuration files.

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

Note: Virtual Ports are **not** available under Microsoft Windows. An alternative is to use third party software such as Tobias Erichsen's loopMIDI (https://www.tobias-erichsen.de/software/loopmidi.html).

Client Name

New in version 1.2.

You can specify a client name for the port:

```
>>> 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 virtual 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 an API

The RtMidi library can be compiled with support for more than one API.

To get a list of all available APIs at runtime:

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

You can select the 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 within the program using 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.

PortMidi

Name: `mido.backends.portmidi`

Resources:

- PortMidi C Library (https://github.com/PortMidi/portmidi)

Installing

The PortMidi backend requires the `portmidi` shared library.

Ubuntu (https://www.ubuntu.com/):

```
apt install libportmidi-dev
```

Homebrew (https://mxcl.dev/homebrew/):

```
brew install portmidi
```

MacPorts (https://www.macports.org/):

```
port install portmidi
```

The backend will look for:

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

Features

- Can send but doesn't receive `active_sensing` messages.
- 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-initializing PortMidi which would break any open ports.)

Pygame

Name: `mido.backends.pygame`

Resources:

- PyGame Python Library (https://www.pygame.org)
- PortMidi C Library (https://github.com/PortMidi/portmidi)

The Pygame backend uses the pygame.midi (https://www.pygame.org/docs/ref/midi.html) module for I/O.

Features

- 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`

Resources:

- rtmidi-python Python Library (https://pypi.org/project/rtmidi-python/)

Installing

```
python3 -m 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`

Todo: 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`

Resources:

- The Advanced Linux Sound Architecture (ALSA) project (https://www.alsa-project.org/)
- ALSA Opensrc Org amidi (https://alsa.opensrc.org/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 a new `amidi` command for each message.
- known bug: is one behind when receiving messages. See below.

Operation

The `amidi` command (part of ALSA and the *alsa-utils* package) 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 message the overhead is very high.

1.3.4.4 Writing a New or Custom 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/`.

1.3.5 Ports

A Mido *port* is an *object* that can *send* and/or *receive* messages.

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` to 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.

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

1.3.5.1 Common

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 functions for input and I/O ports.

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

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.

1.3.5.2 Output

Output *ports* basically only have 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 from Mido's point of view.)

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.

1.3.5.3 Input

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 messages *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` immediately if *no message is available*.

Deprecated since version 1.2: There used to be a `pending()` method which returned the number of pending messages.

It was removed 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.

1.3.5.4 Callbacks

Instead of manually 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
```

Warning: 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.

1.3.5.5 API

Todo: Add abstract code to describe these interfaces.

Common Methods and Attributes

`close()`

Closes 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)`

Sends 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)`

Receives a message. This will block until it returns a message. If `block=False` 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.

1.3.5.6 Socket Ports - MIDI over TCP/IP

About

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

The protocol is a simple MIDI bytes stream over *TCP*.

> **Warning:** It is **not** *rtpmidi*!

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 whitelist of clients 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 an host name or IP address (as a string). It may also be '', in which case connections are accepted from any IP address on the computer.

Todo: Test and clarify "Any IP address on the computer". Does this mean only local adresses can connect or that any connection from any network is allowed?

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 Mido I/O ports. This makes it easy to write simple servers, but you don't have any control on 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()` on the other end.

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

Todo: Clarify "Connection waiting" vs "There is a 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.

1.3.5.7 Writing a New or Custom 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:
```

(continues on next page)

(continued from previous page)

```
    # 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.

1.3.6 Files

1.3.6.1 Standard MIDI Files

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

Opening

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* (page 39) 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 (though this simple implementation is subject to time drift):

```
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 (while avoiding drift). If you pass `meta_messages=True` you will also get meta messages. These **cannot** 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-like 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')
MetaMessage('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 **cannot** 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* (page 71).

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 Time Resolution

Timing in MIDI files is centered around ticks. Each message in a MIDI file has a delta time, which tells how many ticks have passed since the last message.

A tick is the smallest unit of time in MIDI and remains fixed throughout the song. Each quarter notes is divided into a certain number of ticks, often referred as the resolution of the file or pulses per quarter note (PPQN). This resolution is stored as `ticks_per_beat` in MidiFile objects.

The meaning of this `ticks_per_beat` in terms of absolute timing depends on the tempo and time signature of the file.

MIDI Tempo vs. BPM

Unlike music, tempo in MIDI is not given as beats per minute (BPM), but rather in microseconds per quarter note, with a default tempo of 500000 microseconds per quarter note. Given a default 4/4 time signature where a beat is exactly a quarter note, this corresponds to 120 beats per minute.

In case of different time signatures, the length of a beat depends on the denominator of the time signature. E.g. in 2/2 time signature a beat has a length of a half note, i.e. two quarter notes. Thus the default MIDI tempo of 500000 corresponds to a beat length of 1 second which is 60 BPM.

The meta messages 'set_tempo' and 'time_signature' can be used to change the tempo and time signature during a song, respectively.

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 tempo (either in number of beats per minute (BPM) or microseconds per quarter note, see *MIDI Tempo vs. BPM* (page 39) above) and ticks per per quarter note 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 quarter note, by setting MidiFile.ticks_per_beat to a large number. Typical values range from 96 to 480 but some use even more ticks per quarter note.

1.3.6.2 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()` determines 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
```

1.3.7 Included Programs

A few sample programs are installed with Mido and available directly from the *CLI*.

Warning: These are intended to demonstrate the capabilities of Mido and used as a template for your own programs. These are not fully fledged and may miss crucial features.

1.3.7.1 mido-ports

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

1.3.7.2 mido-play

Plays back one or more MIDI files:

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

1.3.7.3 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').

1.3.7.4 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.

Part IV

Reference

1.3.8 API Reference

1.3.8.1 Messages

class mido.**Message**(*type*, *skip_checks=False*, ***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(*skip_checks=False*, ***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.

The skip_checks arg can be used to bypass validation of message attributes and should be used cautiously.

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(*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().

classmethod from_dict(*data*)
Create a message from a dictionary.

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

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

This is the reverse of msg.hex().

classmethod from_str(*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_cc(*control=None*)
Return True if the message is of type 'control_change'.

The optional control argument can be used to test for a specific control number, for example:

if msg.is_cc(7): # Message is control change 7 (channel volume).

is_meta = False

property is_realtime
True if the message is a system realtime message.

Todo: Expose more of the internals? (Checks, decode...)

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*, *skip_checks=False*, ***args*)

class mido.frozen.**FrozenMetaMessage**(*type*, *skip_checks=False*, ***kwargs*)

class mido.frozen.**FrozenUnknownMetaMessage**(*type_byte*, *data=None*, *time=0*, *type='unknown_meta'*, ***kwargs*)

1.3.8.2 Parsing

mido.parser.**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.parser.**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.**Parser**(*data=None*)
MIDI byte stream parser

Parses a stream of MIDI 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()

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.

1.3.8.3 Tokenizing

class mido.tokenizer.**Tokenizer**(*data=None*)
Splits a MIDI byte stream into messages.

feed(*data*)
Feed MIDI bytes to the decoder.

Takes an iterable of ints in in range [0..255].

feed_byte(*byte*)
Feed MIDI byte to the decoder.

Takes an int in range [0..255].

1.3.8.4 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 list of all input port names.

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

get_output_names(***kwargs*)
Return a 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.

property loaded
Return True if the module is loaded.

property 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, it 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, it 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, it 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()*.

Todo: Expose each built-in backend internal API?

1.3.8.5 Ports

Management

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

If the environment variable MIDO_DEFAULT_INPUT is set, it 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**(*name=None*, *virtual=False*, *autoreset=False*, ***kwargs*)
Open an output port.

If the environment variable MIDO_DEFAULT_OUTPUT is set, it 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**(*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, it 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**(***kwargs*)
Return a list of all input port names.

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

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

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.

API

class mido.ports.**BaseInput**(*name=''*, ***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=''*, *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

1.3.8.6 Files

Standard MIDI Files

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

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.

property length
Playback time in seconds.

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

property merged_track

play(*meta_messages=False*, *now=<built-in function time>*)
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.

By default the system clock is used for the timing of yielded MIDI events. To use a different clock (e.g. to synchronize to an audio stream), pass now=time_fn where time_fn is a zero argument function that yields the current time in seconds.

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**(*iterable=()*, */*)

append(*object*, */*)
Append object to the end of the list.

clear()
Remove all items from list.

copy()
Return a shallow copy of the list.

count(*value*, */*)
Return number of occurrences of value.

extend(*iterable*, */*)
Extend list by appending elements from the iterable.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)
Return first index of value.

Raises ValueError if the value is not present.

insert(*index*, *object*, */*)
Insert object before index.

property 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=- 1*, */*)
Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

remove(*value*, */*)
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

sort(***, *key=None*, *reverse=False*)
Sort the list in ascending order and return None.

The sort is in-place (i.e. the list itself is modified) and stable (i.e. the order of two equal elements is maintained).

If a key function is given, apply it once to each list item and sort them, ascending or descending, according to their function values.

The reverse flag can be set to sort in descending order.

class mido.**MetaMessage**(*type*, *skip_checks=False*, ***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.

classmethod from_bytes(*msg_bytes*)

classmethod from_dict(*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_cc(*control=None*)
Return True if the message is of type 'control_change'.

The optional control argument can be used to test for a specific control number, for example:

if msg.is_cc(7): # Message is control change 7 (channel volume).

is_meta = True

property 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/pulses per quarter note, also called PPQN) and tempo (microseconds per quarter note).

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/pulses per quarter note, also called PPQN) and tempo (microseconds per quarter note). Normal rounding applies.

mido.**bpm2tempo**(*bpm*, *time_signature=(4, 4)*)
Convert BPM (beats per minute) to MIDI file tempo (microseconds per quarter note).

Depending on the chosen time signature a bar contains a different number of beats. These beats are multiples/fractions of a quarter note, thus the returned BPM depend on the time signature. Normal rounding applies.

mido.**tempo2bpm**(*tempo*, *time_signature=(4, 4)*)
Convert MIDI file tempo (microseconds per quarter note) to BPM (beats per minute).

Depending on the chosen time signature a bar contains a different number of beats. The beats are multiples/fractions of a quarter note, thus the returned tempo depends on the time signature denominator.

mido.**merge_tracks**(*tracks*, *skip_checks=False*)
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.

Pass skip_checks=True to skip validation of messages before merging. This should ONLY be used when the messages in tracks have already been validated by mido.checks.

SYX

mido.syx.**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.syx.**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).

Part V

Community

1.3.9 Code of Conduct

1.3.9.1 Our Community

Members of the Mido community are **open, considerate, and respectful**. Behaviours that reinforce these values contribute to a positive environment, and include:

- **Being open.** Members of the community are open to collaboration, whether it's on patches, problems, or otherwise.
- **Focusing on what is best for the community.** We're respectful of the processes set forth in the community, and we work within them.
- **Acknowledging time and effort.** We're respectful of the volunteer efforts that permeate the Mido community. We're thoughtful when addressing the efforts of others, keeping in mind that often times the labor was completed simply for the good of the community.
- **Being respectful of differing viewpoints and experiences.** We're receptive to constructive comments and criticism, as the experiences and skill sets of other members contribute to the whole of our efforts.
- **Showing empathy towards other community members.** We're attentive in our communications, whether in person or online, and we're tactful when approaching differing views.
- **Being considerate.** Members of the community are considerate of their peers – other Mido users.
- **Being respectful.** We're respectful of others, their positions, their skills, their commitments, and their efforts.
- **Gracefully accepting constructive criticism.** When we disagree, we are courteous in raising our issues.
- **Using welcoming and inclusive language.** We're accepting of all who wish to take part in our activities, fostering an environment where anyone can participate and everyone can make a difference.

1.3.9.2 Our Standards

Every member of our community has the right to have their identity respected. The Mido community is dedicated to providing a positive experience for everyone, regardless of age, gender identity and expression, sexual orientation, disability, physical appearance, body size, ethnicity, nationality, race, or religion (or lack thereof), education, or socio-economic status.

Examples of unacceptable behavior by participants include:

- Harassment of any participants in any form
- Deliberate intimidation, stalking, or following
- Logging or taking screenshots of online activity for harassment purposes
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Violent threats or language directed against another person
- Incitement of violence or harassment towards any individual, including encouraging a person to commit suicide or to engage in self-harm
- Creating additional online accounts in order to harass another person or circumvent a ban
- Sexual language and imagery in online communities or in any conference venue, including talks
- Insults, put downs, or jokes that are based upon stereotypes, that are exclusionary, or that hold others up for ridicule
- Excessive swearing
- Unwelcome sexual attention or advances
- Unwelcome physical contact, including simulated physical contact (eg, textual descriptions like "hug" or "backrub") without consent or after a request to stop

- Pattern of inappropriate social contact, such as requesting/assuming inappropriate levels of intimacy with others
- Sustained disruption of online community discussions, in-person presentations, or other in-person events
- Continued one-on-one communication after requests to cease
- Other conduct that is inappropriate for a professional audience including people of many different backgrounds

Community members asked to stop any inappropriate behavior are expected to comply immediately.

1.3.9.3 Consequences

If a participant engages in behavior that violates this code of conduct, the Mido project maintainers may take any action they deem appropriate, including warning the offender or expulsion from the community.

Thank you for helping make this a welcoming, friendly community for everyone.

1.3.9.4 Scope

This Code of Conduct applies to the following online spaces:

- Code repositories, issue trackers, and pull requests made against the Mido GitHub organization.
- Discussions in the Mido GitHub organization.

This Code of Conduct applies to the following people in the Mido project online spaces:

- admins of the online space
- maintainers
- reviewers
- contributors
- all community members

1.3.9.5 Contact and Procedure for Handling Incidents

Please mention "`@mido`" in the issue or discussion or open a new issue on https://github.com/mido/mido and tag the organization admins using "`@mido`".

You can also contact one or several organization admins directly:

- radovan.bast@uit.no
- ombdalen@gmail.com
- raphael@doursenaud.fr

We will then immediately discuss with the problematic user and convey why their behavior was inappropriate. We will also explain what the user can do to improve their behavior in the future. We will also explain that if the user continues to behave inappropriately, they will be banned from the community. Depending on the severity of the violation, we may also ban the user immediately.

1.3.9.6 License

This Code of Conduct is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License (https://creativecommons.org/licenses/by-sa/3.0/).

 (https://creativecommons.org/licenses/by-sa/3.0/)

1.3.9.7 Attributions

This Code of Conduct was adapted from the PSF Code of Conduct (https://www.python.org/psf/conduct/), which was forked from the example policy from the Geek Feminism wiki, created by the Ada Initiative and other volunteers (https://geekfeminism.fandom.com/wiki/Conference_anti-harassment/Policy), which is under a Creative Commons Zero license (https://creativecommons.org/publicdomain/zero/1.0/).

1.3.10 Contributing

1.3.10.1 Questions

If you have questions about using Mido, contributing code or suggestions for how to make contributing easier, please write at https://github.com/mido/mido/discussions.

1.3.10.2 Bugs & Feature Requests

Note: If you don't have a precise idea, please use the questions section outlined above instead of opening an issue.

If you encounter a bug that is reproducible or want to suggest a new feature - including its implementation details - that would fit the project nicely, feel free to open an issue at https://github.com/mido/mido/issues

Please provide as much information as possible to allow us to analyze, including but not limited to:

- Operating system name & version
- Python version
- `mido` package version & installation method (Distribution repository, PyPI, source…)
- backend used (`amidi`, `portmidi`, `rtmidi`, `PyGame`… Defaults to `python-rtmidi`.)

1.3.10.3 Forking & Pull Requests

The project welcomes all contributions!

If you wish to make a change, be it code or documentation, please fork the repository from https://github.com/mido/mido and send your pull request to https://github.com/mido/mido/pulls.

Your changes will be reviewed by a maintainer and integrated for publication in the next version of *mido* once approved.

1.3.10.4 Installation

Users

For general usage, see *Installing* (page 7).

If you wish to install from source, run the following command from the sources root directory:

```
python3 -m pip install --editable .
```

Or, alternatively if you want to use ports:

```
python3 -m pip install --editable .[ports-rtmidi]
```

Note: *No support* will be provided if you install from source.

Developers

Warning: We recommend that you first setup a *virtual environment* to avoid conflicts with already installed files.

See also:

https://packaging.python.org/en/latest/tutorials/installing-packages/

Then, to install the *development dependencies*, you can run the following command from inside your virtual environment:

```
python3 -m pip install --editable .[dev]
```

Or, alternatively, if you want to use ports:

```
python3 -m pip install --editable .[dev,ports-rtmidi]
```

This will install all needed dependencies for linting, testing, documentation generation and publishing releases.

1.3.10.5 Code Checks

Note: The following code checks are done automatically using a GitHub Actions Workflow (Defined in `.github/workflow/tests.yml`) for each push to the `main` branch and each Pull Request.

It's good practice to check your changes *locally* before submitting.

Linting

Linting is done with ruff (https://docs.astral.sh/ruff). Its configuration can be found in *pyproject.toml*.

You can lint your code using:

```
ruff check .
```

Copyright and REUSE Compliance

The project is REUSE (https://reuse.software) compliant.

If you wish to add your copyright to a file, please add an SPDX header if the form of:

```
# SPDX-FileCopyrightText: YYYY First_Name Last_Name <email_address>
#
# SPDX-License-Identifier: MIT
```

Note: Use the appropriate comment format and license for the file and only add the first line below existing copyright mentions if modifying an existing file.

The year should only be set the first time you edit a file and never touched again. There is **no** benefit in updating it constantly!

---

then run:

```
reuse lint
```

Testing

pytest (https://doc.pytest.org) is used for unit testing. The tests are found in tests/test_*.py. The default configuration is declared in the `tool.pytest.ini_options` section of `pyproject.toml`.

The test suite can be run using the command:

```
pytest
```

Checking the Release Manifest

To make sure the repository and source code manifest (`.MANIFEST.in`) are in sync:

```
check-manifest --verbose
```

Building the Documentation

The documentation is generated using Sphinx (https://www.sphinx-doc.org/).

To generate the HTML documentation:

```
sphinx-build -j auto -q -W -E --keep-going docs docs/_build
```

If you wish to build a PDF version for *local* use:

1. Install a LaTeX (https://www.latex-project.org/get) distribution

2. Install ImageMagick (https://imagemagick.org)
3. use:

```
sphinx-build -M latexpdf docs docs/_build
```

You'll find the resulting PDF file at `docs/_build/latex/Mido.pdf`.

Once generated and copied in a safe place, you may want to remove the build artifacts:

```
sphinx-build -M clean docs docs/_build
```

1.3.10.6 Testing MIDI File Support

Test Files

The Lakh MIDI Dataset (https://www.colinraffel.com/projects/lmd/) is a great resource for testing the MIDI file parser.

1.3.10.7 Releasing

The processes are now automated.

Note: The whole team has access to manual publishing to *PyPI* and *Read the Docs* in case of automation defect.

Documentation

To generate the official documentation, we use *Read the Docs* integration services for GitHub. Every time a new commit is pushed or merged onto our `main` development branch on GitHub, the `latest` version of the documentation is updated by Read the Docs. Each time a new version is tagged, the new documentation version is created, built, published and eventually promoted to `stable` following Semantic Versioning. The `stable` version of the documentation is the one served by default if no specific version is chosen.

We also build a mirror of the current `main` development branch documentation using a GitHub Workflow and hosted on GitHub pages.

All of this is defined by `.github/workflow/documentation.yml`

Package

The process uses GitHub Action Workflow defined by `.github/workflow/release.yml` and is triggered upon receiving a tag.

Preparation

Make sure all the tests pass, documentation has been updated and everything is in good order before proceeding.

Note: The version number should be **PEP 440** (https://www.python.org/dev/peps/pep-0440) & SemVer compliant.

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

---

1. update the changelog in `docs/changes.rst`. The following commands may prove useful to retrieve all Pull Requests & all commits:

```
previous_release_tag=git describe --abbrev=0
git log --oneline --merges --reverse "${previous_release_tag}.."
git log --oneline --no-merges --reverse "${previous_release_tag}.."
```

2. update version and date in `docs/changes.rst`
3. commit the changes:

```
git commit -a -c "Prepare <X.Y.Z> release."
```

4. set the version number by tagging the release:

```
git tag -a <X.Y.Z> -m "mido version <X.Y.Z>"
```

Note: We use an annotated tag here to retain all information about the tagger and create a proper object in the GIT database instead of a commit alias.

See also:

https://git-scm.com/book/en/v2/Git-Basics-Tagging

---

5. don't forget to push your changes including the tags to GitHub to trigger the auto-release process:

```
git push --tags
```

Manual steps (Recovery)

Warning: Only use if the automatic process fails for some reason.

1. Prepare a clean environment:

```
git clone --branch <X.Y.Z> --single-branch https://github.com/mido/mido mido-<X.
↪Y.Z>
cd mido-<X.Y.Z>
python3 -m venv mido-build
```

2. Build:

```
source mido-build/bin/activate
python3 -m pip install --upgrade pip setuptools wheel build twine
python3 -m build
```

3. Publish on Test PyPI:

```
python3 -m build
twine upload --repository testpypi dist/*
```

4. Check that the published package is good:

```
python3 -m pip install --index-url https://test.pypi.org/simple/ --no-deps mido
python3 -c "import mido; print(mido.version_info)"
```

Todo: Now would be a good time to run some integration tests once we have them.

---

5. Publish on PyPI:

```
twine upload dist/*
```

> **Warning:** This is the most critical step of the process. This **cannot** be undone. Make sure everything is in good order before pressing the "big red button"!

Part VI

Appendix

1.3.11 About MIDI

1.3.11.1 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 *always* 7 bits (Values: 0..127).

Each message type has a given number of data bytes, the exception being the *System Exclusive* message which has a start (`SOX`) and a stop (`EOX`) 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 (aka *nibble*) 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, includes `start`, `stop`, `continue`, `song position` (for playback of songs) and `reset`.
- System Exclusive Messages (often called *SysEx* messages) are used for sending and receiving *device specific* data such as patches and proprietary controls.

1.3.11.2 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):
```

(continues on next page)

(continued from previous page)

```
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
```

1.3.12 Message Types

1.3.12.1 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.

1.3.12.2 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` is used in MIDI files as delta time.

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))
```

1.3.13 Meta Message Types

1.3.13.1 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)

New in version 1.2.9: Time signature message have the correct default value of 4/4. In earlier versions the default value was 2/4 due to a typo in the code.

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

Changed in version 1.1.5: The mode attribute was removed. Instead, an 'm' is appended to minor keys.

sequencer_specific (0x7f)

Attribute	Values	Default
data	[..]	[]

An unprocessed sequencer specific message containing raw data.

1.3.13.2 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:

```
UnknownMetaMessage(type_byte=251, data=(1, 2, 3), time=0)
```

1.3.13.3 Implementing New or Custom 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.meta 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]
```

(continues on next page)

(continued from previous page)

```
    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)
MetaMessage('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.

1.3.14 Resources

- MIDI Association (https://midi.org/)
  - An Introduction to MIDI (https://www.midi.org/articles/an-intro-to-midi)
  - Official MIDI 1.0 detail specification (https://www.midi.org/specifications/midi1-specifications/midi-1-0-core-specifications/midi-1-0-detailed-specification-2) *Free registration required.*
  - Standard MIDI Files Specification (https://www.midi.org/specifications/file-format-specifications/standard-midi-files/rp-001-v1-0-standard-midi-files-specification-96-1-4-pdf) *Free registration required.*
  - MIDI Reference Tables (https://www.midi.org/specifications-old/category/reference-tables)
- MIDI (https://en.wikipedia.org/wiki/MIDI) (Wikipedia)
- Essentials of the MIDI Protocol (https://ccrma.stanford.edu/~craig/articles/linuxmidi/misc/essenmidi.html) (Craig Stuart Sapp, CCRMA)
- Outline of the Standard MIDI File Structure (https://www.ccarh.org/courses/253/handout/smf/) (Craig Stuart Sapp, CCRMA)
- Active Sensing (https://www.sweetwater.com/insync/active-sensing/) (Sweetwater)
- MIDI File Parsing (https://www.ccarh.org/courses/253/assignment/midifile/) (Course assignment in Music 253 (https://wiki.ccarh.org/wiki/Music_253) at Stanford University)
- Meta Message (https://www.soundonsound.com/techniques/meta-messages-logic) (Sound On Sound)

1.3.15 Freezing to EXE File

1.3.15.1 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 (https://pyinstaller.org/en/stable/when-things-go-wrong.html#listing-hidden-imports).

1.3.15.2 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.

1.3.16 Version Changes

This project uses Semantic Versioning (https://semver.org).

See also:

The Future (https://github.com/mido/mido/milestone/2) milestone on Github for future plans.

1.3.16.1 Release History

1.3.2 (2023-12-15)

- Bugfix Midifile: skip_checks were missing on MetaMessage (@almostimplemented)
- Bugfix Scripts/mido-play: UnboundLocalError (@rdoursenaud)
- Minor documentation cleanup (@rdoursenaud)

1.3.1 (2023-12-14)

- New Python 3.12 support
- Improved merged_tracks performance (@almostimplemented, pull request #474)
- Bugfix Backends/portmidi: SysEx messages were generating an error (@tweakoz, pull request #523)
- Bugfix Midifile: Defer computing merged_track (@akx, pull request #565)
- Bugfix pip release: Scripts can now be executed properly (@rdoursenaud)
- Tooling: Linting now uses ruff (@akx, pull requests #564, #566)
- Minor documentation improvements (@rdoursenaud)

1.3.0 (2023-07-21)

> **Warning:** This release drops support for Python 2 and is only compatible with 3.7 onwards.

- Bugfix Backends/rtmidi: Prevent virtual port name mangling (@rdoursenaud, thanks to @digitalsignalperson for reporting)
- Bugfix Backends/rtmidi: Remove callback before closing the port to avoid a race condition (@rdoursenaud)
- Bugfix MidiFile: Properly decode/encode SMPTE hours in the SMPTE offset Meta (Thanks to @laori93 for reporting and @heilei for investigating. Issue #156)
- Installation: support the "extras" syntax to install optional dependencies (@rdoursenaud)
- Documentation: updated, overhauled and proofread (@rdoursenaud, nomadbyte, @superbock)
- Bugfix: Backends/Portmidi (@akx, pull request #483)
- MidiFile: Move merging track out of `__iter__()` to prevent hanging on first call (@Frnot, pull request #470)

- MidiFile: `play()` can now use an optional custom clock source (@almostimplemented, pull request #153)
- The project is now REUSE compliant. See https://reuse.software/ for details (@rdoursenaud)
- Packaging is now PEP-518 compliant (@rdoursenaud)
- Backends/Socket: Disable buffering (@m-vo, pull request #342)
- Removed support for Python 2.7. * Mido now requires Python 3.7 or higher. (Ole Martin Bjørndalen, pull request #408, with additional cleanup from @rdoursenaud)
- Backends: The `rtmidi` and `python-rtmidi` 1.2.10 sometimes returned duplicate port names. (Bug introduced in 1.2.10. Fix by Maciej Sokołowski, pull request #321)
- Bugfix Backends/Socket: In Python 3, PortServer used to crashe when a socket client disconnects. (issue #290) (@kyleclaassen, pull request #291)
- MidiFile: Make `UnknownMetaMessage` robust to faulty MIDI files (@sonovice, pull request #286)
- Bugfix MIDIFile: BPM <-> MIDI tempo conversions (@superbock, pull request #114)
- MidiFile: Added `from_bytes()` to `MetaMessage` (@gulaki, pull request #149)

1.2.10 (2021-05-10)

- New `repr()` format for messages, tracks and MIDI file objects. (Implemented by John Belmonte, pull request #164.)
- added new example `midifiles/show_midifile.py` based on the new `repr()` format.
- Added `msg.is_cc()` method. Checks if message is a control change. Can also be used to check for a specific control change number, for example `msg.is_cc(7)`.
- Fixed memory leaks in RtMidi backend (issue #256, fix by The Other Days, pull request #264.)
- clip now works with sysex messages (Fix by Avatar Timo Stüber, pull request #229.)
- Improved docs and error message for time attribute in a message. (tomerv, pull request #249.)
- Improved MidiFile.play to avoid time drift. (Implemented by John Belmonte, pull request #161.)
- bugfix: MIDO_DEFAULT_INPUT was misspelled in mido-ports causing it to be show as 'not set' even though it was set. (Fix by Bernhard Wagner, pull request #192.)
- Now only copies ports once in ports.multi_receive() (Tom Ritchford, pull request #191.)
- Ports lists returned from `get_input_names()` and friends are no longer sorted. (Suggested and implemented by Ryan McCampbell, issue #298.)
- Updated linke in docs to point to the new home github.com/mido/ (Fixed by Joshua Mayers, pull request #177.)
- thanks to Christopher Arndt, Kathryn DiPippo and Timo Stüber for fixing flake8 issues.

1.2.9 (2018-10-05)

- rewrote `Parser` class around a MIDI tokenizer. Should lead to slight speedup and much cleaner code.
- bugfix: *data* attribute was missing for *UnknownMetaMessage* objects. This caused *AttributeError* when the messages were printed or saved to a file. Also, the documentation incorrectly listed the attribute as *_data* instead of *data*. (Reported by Groowy.)
- bugfix: UnknownMetaMessage encoding was broken causing crashes when saving a file with unknown meta messages. (Reported by exeex, issue #159.)
- bugfix: inputs and outputs were switched around when opening named ports with PortMidi backend. (Reproduced by Predrag Radovic, issue #108, fix by Juan Antonio Aldea, pull request #109.)

- bugfix: time signature meta messages had wrong default value of 2/4. The default value is now 4/4. (Fix by Sebastian Böck, pull request #104.)
- bugfix: `msg.copy()` didn't handle generators for sysex data. `msg.copy(data=(i for i in range(3)))` would give `data=()` instead of `data=(0,1,2)`.

  (The code should be refactored so this is handled by the same function everywhere, such as in `__init__()`, in `copy()` and in `parser.feed()`.)
- bugfix: `MultiPort._receive()` ignored the `block` parameter. (Fix by Tom Swirly, pull request #135.)
- bugfix: sequencer number meta message was incorrectly limited to range 0..255 instead of 0..65335. (Reported by muranyia, issue #144.)
- now using Tox for testing. (Implemented by Chris Apple, pull request #123.)
- Travis integration up by Carl Thomé and Chris Apple.

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. (Implemented 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 argument `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: https://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.

1.3.17 Authors

Ole Martin Bjørndalen (lead programmer), Raphaël Doursenaud (co-maintainer) and many other contributors.

Many people have contributed to Mido over the years, but this page has not been updated to include them. The *Version Changes* (page 77) page includes names of all contributors.

See also:

https://github.com/mido/mido/graphs/contributors

1.3.18 Licenses

1.3.18.1 Source Code

Mido is published under the terms of the `MIT License (MIT)`.

1.3.18.2 Project configuration

`CC0 1.0 Universal`.

(https://creativecommons.org/publicdomain/zero/1.0/)

1.3.18.3 Documentation, Illustrations & Logo

`Creative Commons Attribution 4.0 International`.

(https://creativecommons.org/licenses/by/4.0/)

1.3.18.4 Code of Conduct

Creative Commons Attribution-ShareAlike 3.0.


(https://creativecommons.org/licenses/by-sa/3.0/)

1.3.19 Acknowledgments

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.

1.3.20 Glossary

ascii American Standard Code for Information Interchange. The most popular character encoding standard.

backend

backends

backend(s) A Mido backend is the interface between the library and the operating system level MIDI stack. See *Backends* (page 19) for more informations.

callback A function called by the *backend* when message(s) are ready to process.

cli Command Line Interface.

file

files

midi file

standard midi file

SMF A standard MIDI file. As defined by the MIDI Association's specification.

message

messages A MIDI message.

midi The Musical Instrument Digital Interface. The specification is maintained by the MIDI Association (https://midi.org).

nibble Half a byte (usually 4 bits). An 8-bit byte has 2 nibbles: an upper and a lower nibble.

pip The Python Package Installer (https://pypi.org/project/pip/).

port

ports A MIDI port.

pypi The Python Package Index (https://pypi.org).

python The Python programming language (https://www.python.org).

rtd

read the docs Read the Docs (https://www.readthedocs.org) or RTD for short is a popular service to build, manage versions and host documentation generated from Sphinx (and now MkDocs) in the Python ecosystem.

rtpmidi A standard protocol to send MIDI over a TCP/IP link.

See also:

- **RFC 4695** (https://tools.ietf.org/html/rfc4695.html)

- **RFC 4696** (https://tools.ietf.org/html/rfc4696.html)

tcp Transmission Control Protocol.

See also:

RFC 9293 (https://tools.ietf.org/html/rfc9293.html)

tick

ticks The *MIDI File* unit of time.

sysex

system exclusive Special *MIDI* messages that are intended for consumption by a specific device. Details about the structure and meaning of these messages are often found in the device's manual.

Todo: Fill this glossary and add the `:term:` directive where appropriate.

PYTHON MODULE INDEX

m

INDEX

L

M

N

O

P

R

S

T

W