3.22 Library Reference

3.22.1 Messages

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

bin()
Encode message and return as a bytearray.

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

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

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

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

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

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

Sysex data will be returned as a list.

classmethod from_bytes(*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_meta = False

is_realtime
True if the message is a system realtime message.

3.22.2 Ports

`mido.`**`open_input`**`()`

Open an input port.

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

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

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

`mido.`**`open_output`**`()`

Open an output port.

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

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

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

`mido.`**`open_ioport`**`()`

Open a port for input and output.

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

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

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

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

`mido.`**`get_input_names`**`()`

Return a sorted list of all input port names.

`mido.`**`get_output_names`**`()`

Return a sorted list of all output port names.

`mido.`**`get_ioport_names`**`()`

Return a sorted list of all I/O port names.

3.22.3 Backends

`mido.`**`set_backend`**`(`*name=None*, *load=False*`)`

Set current backend.

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

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

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

`class` `mido.`**`Backend`**`(`*name=None*, *api=None*, *load=False*, *use_environ=True*`)`

Wrapper for backend module.

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

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

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

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

`load`()
Load the module.

Does nothing if the module is already loaded.

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

`loaded`
Return True if the module is loaded.

`module`
A reference module implementing the backend.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.22.4 Parsing

`mido.`**`parse`**`(`*data*`)`

Parse MIDI data and return the first message found.

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

`mido.`**`parse_all`**`(`*data*`)`

Parse MIDI data and return a list of all messages found.

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

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

MIDI 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() b" # Will be converted to integers in Python 2.

`feed_byte``(`*byte*`)`

Feed one MIDI byte into the parser.

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

`get_message``()`

Get the first parsed message.

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

`pending``()`

Return the number of pending messages.

3.22.5 MIDI Files

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

`add_track``(`*name=None*`)`

Add a new track to the file.

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

`length`

Playback time in seconds.

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

`play``(`*meta_messages=False*`)`

Play back all tracks.

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

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

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

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

May take argument meta_only to show only meta messages.

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

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

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

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

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

class mido.**MidiTrack**

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

copy()

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

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

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

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

name
Name of the track.

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

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

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

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

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

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

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

bin()
Encode message and return as a bytearray.

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

bytes()

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

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

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

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

Sysex data will be returned as a list.

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_meta = True

is_realtime
True if the message is a system realtime message.

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

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

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

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

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

Returns microseconds per beat as an integer:

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

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

Returns BPM as an integer or float:

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

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

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

3.22.6 SYX Files

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

Returns a list of sysex messages.

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

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

Messages other than sysex will be skipped.

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

3.22.7 Port Classes and Functions

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

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

close()
Close the port.

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

is_input = True

is_output = False

iter_pending()
Iterate through pending messages.

poll()
Receive the next pending message or None

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

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

This will block until a message arrives.

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

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

class mido.ports.**BaseOutput**(*name=u"*, *autoreset=False*, ***kwargs*)

Base class for output port.

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

close()

Close the port.

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

is_input = False

is_output = True

panic()

Send "All Sounds Off" on all channels.

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

reset()

Send "All Notes Off" and "Reset All Controllers" on all channels

send(*msg*)

Send a message on the port.

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

class mido.ports.**IOPort**(*input*, *output*)

Input / output port.

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

close()

Close the port.

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

is_input = True

is_output = True

iter_pending()

Iterate through pending messages.

panic()

Send "All Sounds Off" on all channels.

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

poll()

Receive the next pending message or None

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

receive(*block=True*)

Return the next message.

This will block until a message arrives.

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

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

reset()

Send "All Notes Off" and "Reset All Controllers" on all channels

send(*msg*)

Send a message on the port.

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

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

close()

Close the port.

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

is_input = True

is_output = True

iter_pending()

Iterate through pending messages.

panic()

Send "All Sounds Off" on all channels.

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

poll()

Receive the next pending message or None

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

receive(*block=True*)

Return the next message.

This will block until a message arrives.

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

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

reset()

Send "All Notes Off" and "Reset All Controllers" on all channels

send(*msg*)

Send a message on the port.

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

mido.ports.**multi_receive**(*ports*, *yield_ports=False*, *block=True*)

Receive messages from multiple ports.

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

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

If block=False only pending messages will be yielded.

`mido.ports.`**`multi_iter_pending`**`(`*ports*, *yield_ports=False*`)`

Iterate through all pending messages in ports.

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

`mido.ports.`**`multi_send`**`(`*ports*, *msg*`)`

Send message on all ports.

`mido.ports.`**`sleep`**`()`

Sleep for N seconds.

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

`mido.ports.`**`set_sleep_time`**`(`*seconds=0.001*`)`

Set the number of seconds sleep() will sleep.

`mido.ports.`**`get_sleep_time`**`()`

Get number of seconds sleep() will sleep.

`mido.ports.`**`panic_messages`**`()`

Yield "All Sounds Off" for all channels.

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

`mido.ports.`**`reset_messages`**`()`

Yield "All Notes Off" and "Reset All Controllers" for all channels

3.22.8 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.

3.22.9 Frozen Messages

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

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

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

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

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

Returns a mutable version of a frozen message.

Will return None if called with None.

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

class mido.frozen.**Frozen**

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

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

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

3.23 Resources

- MIDI Manufacturers Association (midi.org)
- Table of MIDI Messages (midi.org)
- Tech Specs & Info (midi.org)
- MIDI (Wikipedia)
- Essentials of the MIDI Protocol (Craig Stuart Sapp, CCRMA)
- Outline of the Standard MIDI File Structure (Craig Stuart Sapp, CCRMA)
- Active Sense (About the active sensing message.)
- Active Sensing (Sweetwater)
- MIDI Technical/Programming Docs (Jeff Glatt)
- Standard MIDI Files (cnx.org)

- MIDI File Parsing (Course assignment in Music 253 at Stanford University)
- MIDI File Format (The Sonic Spot)
- Delta time and running status (mic at recordingblogs.com)
- MIDI meta messages (recordingblog.com)
- Meta Message (Sound On Sound)

3.24 License

3.25 Authors

Ole Martin Bjørndalen (lead programmer)

Rapolas Binkys

3.26 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.