Mido Documentation

Release 1.0.4

Ole Martin Bjørndalen

Jul 20, 2023

Contents

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

```
>>> import mido
>>> output = mido.open_output()
>>> output.send(mido.Message('note_on', note=60, velocity=64))
```

```
>>> with input as mido.open_input('SH-201'):
...     for msg in input:
...         print(msg)
```

```
>>> msg = mido.Message('program_change', program=10)
>>> msg.type
'program_change'
>>> msg.channel = 2
>>> msg2 = msg.copy(program=9)
<message program_change channel=2, program=9, time=0>
```

Mido is short for MIDi Objects.

CHAPTER 1

Source Code

Latest version of the code: http://github.com/olemb/mido/ .

Latest development version: http://github.com/olemb/mido/tree/develop/

CHAPTER 2

Contents

2.1 Why Mido?

2.1.1 Messages as Objects

Working with MIDI messages by manipulating the bytes is painful:

```
NOTE_OFF = 0x80
NOTE_ON = 0x90
...
message = device.read()  # Returns [0x92, 0x40, 0x42]
status_byte = message[0]
if status_byte & 0xf0 in [NOTE_ON, NOTE_OFF]:
    is_note_on == status_byte & 0x10
    if is_note_on:
        message_type = 'note_on'
    else:
        message_type = 'note_off'
    channel = message[0] & 0x0f
    print('Got {} on channel {}'.format(message_type, channel))
```

This doesn't look much like Python! You could make some utility functions on top of this to make it a little bit better, but it won't get you far.

With Mido, you can instead do:

```
message = port.receive()
if message.type in ['note_on', 'note_off']:
    print('Got {} on channel {}'.format(message.type, message.channel))
```

2.1.2 Type and Value Checking

Working directly with the bytes is also error prone. While MIDI data bytes have a valid range of 0..127, the size of Python integers is only limited by how much memory is available. If you make a mistake in your computation of a data value, it could easily travel around undetected until it blows up some seemingly unrelated part of your system. These kinds of errors are tricky to track down.

Mido messages come with type and value checking built in. This happens when you assign an out of range value to an attribute:

```
>>> n = mido.Message('note_on')
>>> n.channel = 2092389483249829834
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./mido/messages.py", line 327, in __setattr__
    ret = check(value)
  File "./mido/messages.py", line 128, in check_channel
    raise ValueError('channel must be in range 0..15')
ValueError: channel must be in range 0..15
```

and when you pass some nonsense as a keyword argument to the constructor or the copy() method:

```
>>> n.copy(note=['This', 'is', 'wrong'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./mido/messages.py", line 316, in copy
    return Message(self.type, **args)
  File "./mido/messages.py", line 290, in __init__
    setattr(self, name, value)
  File "./mido/messages.py", line 327, in __setattr__
    ret = check(value)
  File "./mido/messages.py", line 181, in check_databyte
    raise TypeError('data byte must be an integer')
TypeError: data byte must be an integer
```

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

2.2 Tutorial

2.2.1 Creating Messages

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

```
>>> from mido import Message
>>>
>>> Message('note_on', note=60, velocity=100)
<message note_on channel=0, note=60, velocity=100, time=0>
```

All message parameters are optional, and if not explicitly set, will default to `0`. The exceptions are velocity which will be 64 and `sysex_data` which will be `()`:

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

>>> Message('note_off')
```

(continues on next page)

(continued from previous page)

```
<message note_off channel=0, note=0, velocity=64, time=0>

>>> Message('sysex')
<message sysex data=(), time=0>
```

64 is half way between 0 and 127, which means you can leave it out and still have a reasonable value. (This is the recommended default for devices that don't support velocity.)

The parameters for each message type are listed in *Message Types*.

2.2.2 Modifying and Copying Messages

When you have created a message, the parameters are available as attributes:

```
>>> msg = Message('note_off', channel=1, note=60, velocity=50)
>>> dir(msg)
[..., 'channel', 'note', 'time', 'type', 'velocity']
>>> msg.type
'note_on'
>>> msg.channel
1
>>> msg.note
60
>>> msg.channel = 2
>>> msg.note = 62
>>> msg
<message note_off channel=2, note=62, velocity=50, time=0>
```

You can copy a message, optionally passing keyword arguments to override attributes:

```
>>> msg.copy()  # Make an identical copy.
<message note_on channel=2, note=62, velocity=50, time=0>
>>> msg.copy(channel=4)
<message note_on channel=4, note=62, velocity=50, time=0>
```

This is useful when you pass messages around in a large system, and you want to keep a copy for yourself while allowing other parts of the system to modify the original.

Changing the type of a message is not allowed:

```
>>> msg.type = 'note_off'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mido/messages.py", line 320, in __setattr__
    raise AttributeError('{} attribute is read only'.format(name))
AttributeError: type attribute is read only
```

2.2.3 Comparing Messages

You can compare two messages to see if they are identical:

```
>>> Message('note_on', note=22) == Message('note_on', note=22)
True
>>> Message('note_on') == Message('note_off')
```

(continues on next page)

(continued from previous page)

```
False
>>> msg == msg.copy(note=100)
False
```

The `time` parameter (see below) is ignored when comparing messages:

```
>>> msg == msg.copy(time=10000)
True
```

This allows you to compare messages that come from different sources and have different time stamps. If you want to include time in the comparison, you can do:

```
>>> msg1 = note_on(time=2)
>>> msg2 = note_on(time=3)
>>> (msg1, msg1.time) == (msg2, msg2.time)
False
```

2.2.4 System Exclusive (sysex) Messages

Sysex messages have a `data` parameter, which is a sequence of bytes. The `data` parameter takes any object that generates bytes when iterated over. This is converted internally into a tuple of integers:

```
>>> Message('sysex')
<message sysex data=(), time=0>
>>> Message('sysex', data=[1, 2, 3])
<message sysex data=(1, 2, 3), time=0>
>>> Message('sysex', data=bytearray('abc'))
<message sysex data=(97, 98, 99), time=0>
```

Sysex messages inlude the `sysex_end` byte when sent and received, so while there is a `sysex_end` message type, it is never used:

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

2.2.5 Time

All messages also have an extra parameter `time`, which you can use for anything you want. Typically this is used to tag messages with time when storing them in files or sending them around in the system. `time` can have any value as long as it's a `float` or an `int`.

`copy()` will copy the `time` attribute.

2.2.6 Opening Ports

There are three types of ports in Mido: input ports, output ports and I/O ports. They are created with:

```
mido.open_input(name=None)
mido.open_output(name=None)
mido.open_ioport(name=None)
```

(`mido.open_ioport` will return a port which is a thin wrapper around an input port and an output port, and allows you to use the methods of both. This can be used for two-way communication with a device.

You can pass the name of the port, or leave it out to open the default port:

```
mido.open_input('SH-201')  # Open the port 'SH-201'.
mido.open_input()  # Open the default input port.
```

To get a list of names of available ports, you can call one of these functions:

```
>>> mido.get_input_names()
['Midi Through Port-0', 'SH-201']

>>> mido.get_output_names()
['Midi Through Port-0', 'SH-201']

>>> mido.get_ioport_names()
['Midi Through Port-0', 'SH-201']
```

Note: If a port is open, it will still be listed here.

2.2.7 Closing Ports

A port can be closed by calling the `close()` method:

```
port.close()
```

but often it is better to use the `with` statement, which will close the block automatically when the block is over:

```
with mido.open_output() as port:
    ...
```

The `closed` attribute will be `True` if the port is closed.

2.2.8 Sending Messages

Messages can be sent on output or I/O ports by calling the `send()` method:

```
port.send(Message('pitchwheel', channel=2, pitch=4000))
```

The message will be sent immediately.

2.2.9 Receiving Messages

There are several different ways to receive messages. The basic one is to call `receive()`:

```
message = port.receive()
```

This will block until a message arrives on the port. If you want to receive messages in a loop, you can do:

```
for message in port:
    ...
```

If you don't want to block, you can use `pending()` to see how many messages are available:

```
>>> port.pending()
2
>>> port.receive()
<message note_on channel=2, note=60, velocity=50, time=0>
>>> port.receive()
<message note_on channel=2, note=72, velocity=50, time=0>
>>> port.receive()
    *** blocks until the next message arrives ***
```

It is often easier to use `iter_pending()`:

```
while 1:
    for message in port.iter_pending():
        ... # Do something with message.

    ... Do other stuff.
```

Messages will be queued up inside the port object until you call `receive()` or `iter_pending()`.

If you want to receive messages from multiple ports, you can use `multi_receive()`:

```
from mido.ports import multi_receive

while 1:
    for message in multi_receive([port1, port2, port3]):
        ...
```

The ports are checked in random order to ensure fairness. There is also a non-blocking version of this function:

```
while 1:
    for message in multi_iter_pending([port1, port2, port3]):
        ...
```

2.3 Installing Mido

2.3.1 Requirements

Mido targets Python 2.7 and 3.2 and runs on Ubuntu and Mac OS X. May also run on other systems.

If you want to use message ports, you will need PortMidi installed on your system. The PortMidi library is loaded on demand, so you can use the parser and messages without it.

2.3.2 Installing

To install:

```
$ pip install mido
```

The PortMidi wrapper is written with *ctypes*, so no compilation is required.

2.3.3 Installing PortMidi

If you want to use ports, you need the PortMidi shared library. The Ubuntu package is called `libportmidi-dev`. PortMidi is also available in MacPorts and Homebrew under the name `portmidi`.

2.4 Library Reference

2.4.1 Creating Message and Opening Ports

`mido.`**`open_input`**(*name=None*)
Open an input port.

`mido.`**`open_output`**(*name=None*)
Open an output port.

`mido.`**`open_ioport`**(*name=None*)
Open a port for input and output.

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

These names can be passed to Input().

`mido.`**`get_output_names`**()
Return a sorted list of all input port names.

These names can be passed to Output().

`mido.`**`get_ioport_names`**()
Return the names of all ports that allow input and output.

2.4.2 Parsing and Parser class

`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`**
MIDI Parser

Parses a stream of bytes and produces messages.

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

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

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

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

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

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

get_message()
Get the first parsed message.

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

pending()
Return the number of pending messages.

2.4.3 Message Objects

class mido.**Message**(*type_*, ***parameters*)
MIDI message class.

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

Example:

a = Message('note_on') b = a.copy(velocity=32)

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

Each number is separated by the string sep.

2.4.4 String Serialization

There is not *format_as_string()*, but you can use *str(message)*.

mido.**parse_string**(*text*)
Parse a string of text and return a message.

The string can span multiple lines, but must contain one full message.

Raises ValueError if the string could not be parsed.

mido.**parse_string_stream**(*stream*)
Parse a stram of messages and yield (message, error_message)

stream can be any iterable that generates text strings. If a line can be parsed, (message, None) is returned. If it can't be parsed (None, error_message) is returned. The error message containes the line number where the error occured.

2.4.5 Ports

class mido.ports.**BaseInput**(*name=None*)
Base class for input port.

Override _pending() 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.

iter_pending()
Iterate through pending messages.

pending()
Return how many messages are ready to be received.

This can be used for non-blocking receive(), for example:

for _ in range(port.pending()): message = port.receive()

If this is called on a closed port, it will work as if the port was opened, but no new messages will be returned once the buffered ones run out.

receive()
Return the next message.

This will block until a message arrives. For non-blocking behavior, you can use pending() to see how many messages can safely be received without blocking.

NOTE: Blocking is currently implemented with polling and time.sleep(). This is inefficient, but the proper way doesn't work yet, so it's better than nothing.

Todo: What should happen when the port is closed? - raise exception? - return pending messages until we run out, then raise exception?

class mido.ports.**BaseOutput**(*name=None*)
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.

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

The message is sent immediately.

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.

iter_pending()
Iterate through pending messages.

pending()
Return how many messages are ready to be received.

This can be used for non-blocking receive(), for example:

for _ in range(port.pending()): message = port.receive()

If this is called on a closed port, it will work as if the port was opened, but no new messages will be returned once the buffered ones run out.

receive()
Return the next message.

This will block until a message arrives. For non-blocking behavior, you can use pending() to see how many messages can safely be received without blocking.

NOTE: Blocking is currently implemented with polling and time.sleep(). This is inefficient, but the proper way doesn't work yet, so it's better than nothing.

Todo: What should happen when the port is closed? - raise exception? - return pending messages until we run out, then raise exception?

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

The message is sent immediately.

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

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

ports is an iterable of message ports to check.

This can be used to receive messages from a set of ports in a non-blocking manner.

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

2.5 Message Types

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
undefined_f1	
songpos	pos
song_select	song
undefined_f4	
undefined_f5	
tune_request	
sysex_end	
clock	
undefined_f9	
start	
continue	
stop	
undefined_fd	
active_sensing	
reset	

2.5.1 Parameter Types

Name	Valid Range	Default Value
channel	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

`velocity` for `note_off` is release velocity, that is how quickly the note was released. Few instruments support this.

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

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

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

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

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

2.6 Parsing and Encoding Messages

MIDI is a binary protocol, which means when sending a message to a device, it is encoded as one or more consecutive bytes.

The input and output ports will decode and encode messages for you, so unless you're implementing a new MIDI backend or a file reader / writer, there is little use for this.

Message objects have a few methods that make encoding easy:

```
>>> n = Message('note_on', channel=2, note=60, velocity=100, time=3)
>>> n.bytes()
[146, 60, 100]
>>> n.hex()
'92 3C 64'
>>> n.hex(sep='-')
'92-3C-64'
>>> n.bin()
bytearray(b'\x92<d')
```

System Exclusive messages include the end byte (0xf7):

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

This means, the sysex_end() message type is needed.

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

2.6.1 Parsing Messages

If you're implementing a new port type or support for a binary file format, you may need to parse binary MIDI messages. Mido has a few functions and one class that make this easy.

To parse a single message:

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

`parse()` will only return the first message in the byte stream. To get all messages, use `parse_all()`.

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

```
>>> p = mido.Parser()
>>> p.feed([0x90, 0x10, 0x20])
>>> p.pending()
```

(continues on next page)

(continued from previous page)

```
1
>>> p.get_message()
<message note_on channel=0, note=16, velocity=32, time=0>
>>> p.feed_byte(0x90)
>>> p.feed_byte(0x10)
>>> p.feed_byte(0x20)
>>> p.get_message()
<message note_on channel=0, note=16, velocity=32, time=0>
```

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

feed() accepts any iterable that generates integers in 0..255. This includes:

```
p.feed([0x90, 0x10, 0x20])
p.feed((i for i in range(256)))
```

The messages will stay in an internal queue intil you pull them out with *get_message()* or *for message in parser:*.

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.

2.7 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:

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

2.7.1 Format

The format is simple:

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

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

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

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

or written out:

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

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

2.7.2 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 occured.

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
```

2.7.3 Examples

And example of messages embedded in JSON:

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

2.8 Adding New Port Types

Mido comes with support for PortMidi built in, and experimental support for RtMidi through the python-rtmidi package. If you want to use some other library or system, like say PyGame, you can write write custom ports.

There are two ways of adding new port types to Mido.

2.8.1 Duck Typing

The simplest way is to just create an object that has the methods that you know will be called, for example:

```
class PrintPort:
    """Port that prints out messages instead of sending them."""

    def send(self, message):
        print(message)

port = PrintPort()
port.send(mido.Message('note_on')
```

2.8.2 Subclassing

If you want the full range of behaviour, you can subclass the abstract port classes in `mido.ports`:

```
from mido.ports import BaseInput, BaseOutput

class PortCommon(object):
    ... Mixin for things that are common to your Input and Output
    ... ports (so you don't need a lot of duplicate code.

    def _open(self, **kwargs):
        ... This is where you actually # open
        ... the underlying device.
        ...
        ... self.name will be set to the name that was passed
        ... **kwargs will be passed to you by __init__()

    def _close(self):
        ... Close the underlying device.

    def _get_device_type(self):
        ... A text representation of the type of device,
        ... for example 'CoreMidi' or 'ALSA'. This is
        ... used by __repr__(). Defaults to 'Unknown'.
        return 'CoreMidi'  # For example.


class Input(PortCommon, BaseInput):
    def _pending(self):
        ... Check for new messages, feed them
        ... to the parser and return how many messages
        ... are now available.

class Output(PortCommon, BaseOutput):
    def _send(self, message):
        ... Send the message via the underlying device.
```

The base classes will take care of everything else. You may still override selected methods if you need to.

All the methods you need to override start with an underscore and is are called by the corresponding method without an underscore. This allows the base class to do some type and value checking for you before calling your implementation specific method. It also means you don't have to worry about adding doc strings.

See `mido.portmidi.py` and `extras/mido_rtmidi.py` for full examples.

2.9 About 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 Circuits with the assistance of Ikutaro Kakehashi of Roland. 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 as thus only 7 bits (0..127).

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

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).
- System common messages.
- System real time messages, the include start, stop, continue, song position (for playback of songs) and reset.
- System Exclusive messages (often called Sysex messages). These are used for sending and receiving device specific such as patch data.

2.9.1 Some Examples of Messages

```
# Turn on and off middle C
92 3C 64  note_on channel=2 note=60 velocity=100
82 3C 64  note_off channel=2 note=60 velocity=100

# Program change with program=4 on channel 2.
# (The synth will switch to another sound.)
C2 04

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

# 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
```

2.9.2 More About MIDI

- Wikipedia's page on MIDI
- MIDI Manufacturers Association
- A full table of MIDI messages

2.10 License

2.11 Author

Ole Martin Bjørndalen

The PortMidi wrapper is based on portmidizero by Grant Yoshida.

2.11.1 Contact Information

ombdalen@gmail.com

Username on Reddit: /u/halloi/

Nick on freenode: olemb

2.12 Acknowledgements

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

The PortMidi wrapper is based on portmidizero by Grant Yoshida.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index