The MidiDuino Library: MIDI for the Arduino

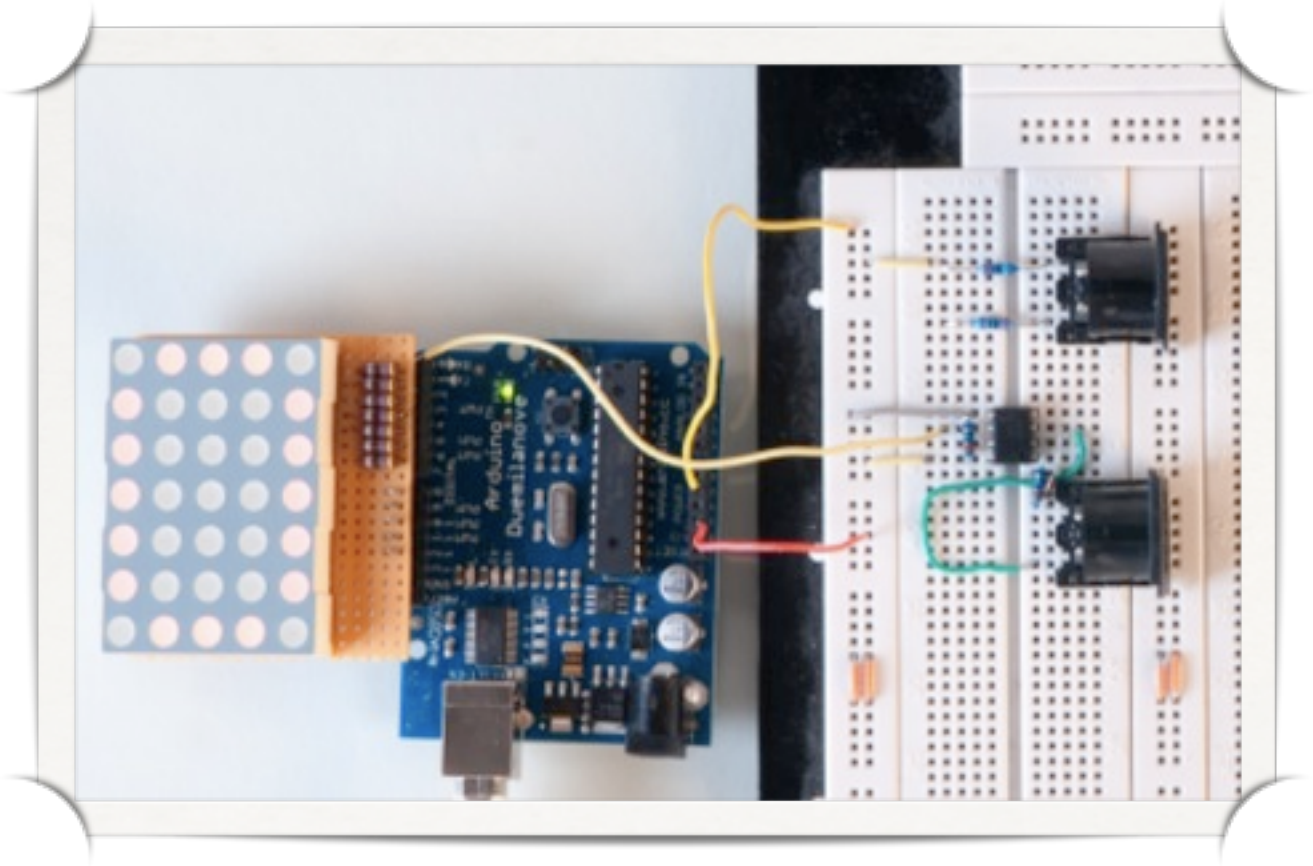


Table of Contents

Introduction

Welcome to the Ruin & Wesen MidiDuino library, which is a nice set of opensource libraries to interface your Arduino with MIDI. The MIDI libraries take care of configuring the Arduino serial interface for MIDI use, offers functions to send most common MIDI messages, and takes care of parsing and handling incoming MIDI messages. No need to worry about understanding all the nooks and crannies of the complex MIDI specification, you can now write a few lines of code and have a robust and ready to go MIDI project. Furthermore, the MidiDuino libraries offer very tight timing synchronization facilities, allowing you to either sequence external MIDI gear, or synchronize to an external MIDI clock. The MidiDuino libraries also come with an extensive library for the Elektron MachineDrum, if you happen to own such a drum machine. The MidiDuino libraries are completely opensource and the sourcecode repository and bug tracker can be found at http://mididuino.googlecode.com/ . The library release versions, as well as the documentation and example sketches, can be downloaded from http://ruinwesen.com/mididuino/ .

For now, the MidiDuino libraries have been tested on the Arduino Duemilanove with atmega168 and atmega328. They should work fine on every atmega168 or atmega328 based Arduino, support for the Arduino Mega is in progress.

MidiDuino and the MiniCommand

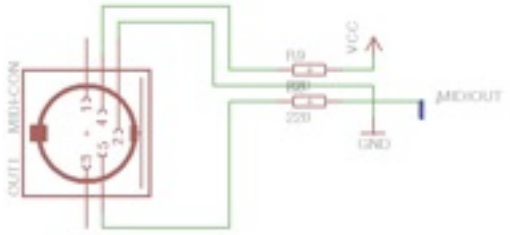
At Ruin & Wesen, we build a device called the MiniCommand, that happens to have a microcontroller (the AVR atmeg64) similar to the microcontroller on the Arduino board (the Duemilanove has an AVR atmega168 or atmega328). However, the MiniCommand at first wasn't programmed using the Arduino environment. Later on, we decided to modify the Arduino environment to be able to program the MiniCommand using sketches, and modified the Arduino editor to support direct uploading of firmwares over MIDI. However, the whole codebase of the MiniCommand is completely separate. We later on decided to separate all the MIDI functionality out of the MiniCommand codebase and make it compatible with the normal Arduino, which has resulted in the MidiDuino libraries. So MidiDuino is both a name for the development environment for the MiniCommand, which cannot be used with the normal Arduino, and the name for the set of libraries that are used to handle MIDI on the MiniCommand, and which work on the Arduino as well.

If you want to build MIDI controllers with the MidiDuino libraries, we encourage you to take a look at the MiniCommand, which is a small, handy and very robust device that can be very easily programmed in a similar fashion to programming the Arduino. It comes with four encoders, four buttons, a small LCD screen, additional memory (128 kilobytes of memory), hardwired MIDI ports, a microSD-Card for storage, and can be programmed directly over MIDI. Furthermore, we have some very nice libraries to program the MIDI controller user interface and take care of all the buttons and encoders. You can check it out at http://ruinwesen.com/digital .

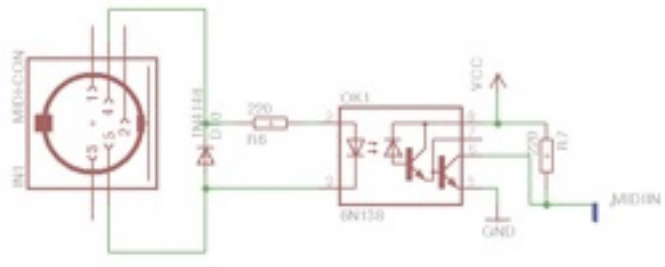
Hardware Description

In order to interface your Arduino with a MIDI device, you need to build a small circuit. MIDI is a serial interface (with separate ports for input and output), and data is communicated over a current loop. That means that not voltage is used to signal either 0 or 1, but current, which has to be converted back into voltage in order for the Arduino to be able to read that information. The reason for this setup is to allow ground-loops in a big MIDI setup.

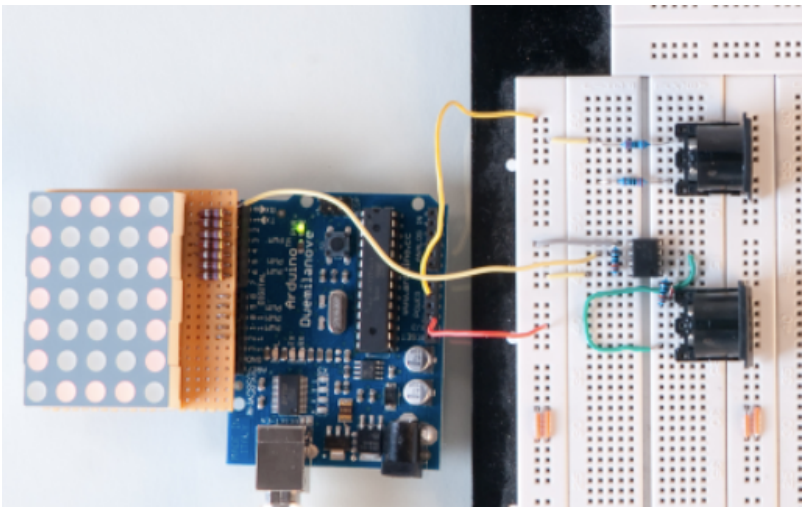
In order to send MIDI data, you just need to connect the TX pin of the Arduino to the MIDI connector (a DIN 5-pin 180 degrees connector) over a 220 Ohm resistor:



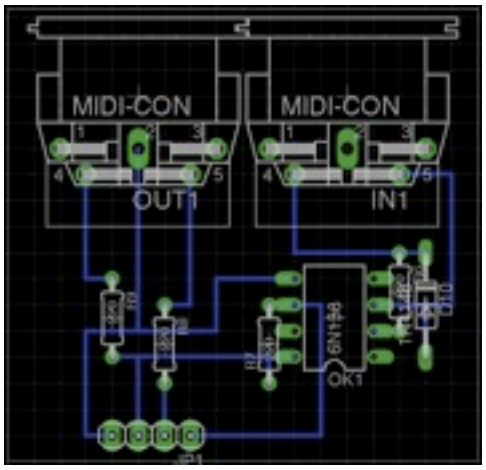
Receiving MIDI is slightly more complicated, and is usually done with a chip called an optocoupler. You may get lucky by connecting the input pin directly to RX, and hoping both devices have the same ground, but the usual circuit is the following one. You need a diode and a few 220 Ohm resistors.



These circuits can easily be built on a breadboard, as shown in the following picture.



However, this is not very solid, and will not last super long. A better way is to build a small stripboard circuit that you can plug into the Arduino. In order to make it easier for you to build these circuits, here is a layout of a small stripboard version of this circuit that you can use to build your MIDI circuits.



It is also possible to send MIDI data directly over the USB serial link to the computer. To do this, you can pass a "standard" serial speed to the initialization call **MidiUart.init()**, for example by setting it to 115200 bps by calling **MidiUart.init(115200)**. For more information about how to use the library, see the tutorial and the API reference below.

However, the operating system doesn't know that the received data is in fact MIDI data. To inject the received data into the MIDI stack of the operating system, you need a program to convert the received serial data into MIDI. Under Linux, you can use either the serial midi driver of the operating system, or a program such as MidiBang (http://code.google.com/p/midibang/). Under MacOSX, you can use the ardrumo software (http://code.google.com/p/ardrumo/) or Serial MIDI Converter (http://www.spikenzielabs.com/SpikenzieLabs/Serial_MIDI.html). Under Windows, you can use s2midi (http://www.memeteam.net/2007/s2midi/). Be sure to disconnect these programs before uploading a new sketch to the Arduino, as they will interfere.

Installation

There are two different versions of the MidiDuino library. The first version is a plain version that you can use in addition to the normal Arduino core libraries, and that will allow you to receive and send MIDI. However, due to some internal reasons of how the serial interface is used, there may be slight jitters on the MIDI Clock synchronization. To take care of a really tight MIDI Clock synchronization, you have to install an additional Core library, and change the board description in Arduino.

Installing the standard libraries is very simple. Download the latest library ZIP-Archive from http://ruinwesen.com/mididuino, and copy the included folders (MidiUart into the libraries folder of your Arduino installation. This libraries folder can be found inside the hardware/libraries folder in the Arduino folder.

Installing the additional arduino core is a bit more complicated. You have to copy the arduinomidi folder into the hardware/cores folder of your Arduino installation. You then need to add the content of the included boards.txt file to the end of the hardware/boards.txt file in your Arduino installation. After restarting the Arduino environment, you will be able to select the “Arduino Duemilanove w/ Atmega328 and MIDI” or “Arduino Diecimila or Duemilanove w/ Atmega168 and MIDI” from the Tools -> Boards menu in the Arduino environment. All the older functions of the Arduino core are still available, the only difference is that incoming MIDI clock bytes are handled directly from the interrupt routine servicing the serial port.

Programming with the MidiDuino library

This chapter will (hopefully) give you a smooth introduction into programming MIDI programs on the Arduino. You first need to install the MidiDuino environment as described in the first chapter of this manual. Once you are able to upload new firmwares, you are also ready to develop your own. Open the MidiDuino editor, and create a new empty patch. The programming language used to write firmwares for the MiniCommand is basically the Arduino language, which basically is C++. If you have never written a program in Arduino or C++, we encourage you to take a look at the Arduino reference pages under http://www.arduino.cc/en/Reference/HomePage , and sheepdog's programming tutorials at http://sheepdogsoftware.co.uk/pltut.htm . We are planning to provide an extensive programming tutorial aimed especially at musicians.

Sending MIDI

In this first sketch, we are going to see how to initialise the MIDI library and use it to send a very simple note message at regular intervals. We first need to include the MIDI libraries by adding these lines at the beginning of the sketch.

```
#include <MidiUart.h>
#include <Midi.h>
```

These include files are used to integrate the two big parts of the MidiDuino library. The MidiUart functionality is used to interface with the sending side of the serial interface. It configures the speed of the serial interface, and is used to send out MIDI messages. It also takes care of something called the "Running Status" on MIDI, which allows to send more data by removing duplicate status bytes. The Midi part, which is not going to be used in this first sketch, takes care of handling incoming MIDI bytes, parsing them, recognizing messages, and doing clock synchronization.

In the next step, we need to initialize the MidiUart part of the library. This is done with a call to the function **MidiUart.init()**. This will set up the correct speed of the serial interface (31250 bps for standard MIDI), set up the transmission buffer (so that sending out bytes will run in the background), and set up a clock to enable clock synchronization. An optional speed argument can be passed to **MidiUart.init()** to set a specific bitrate for the serial port. This can be useful if the USB serial link is used to transmit MIDI data to the computer.

```
void setup() {
  MidiUart.init();
}
```

Once the MidiUart has been configured, we are ready to go to send MIDI data. This can be done with a big number of functions used to send different types of MIDI messages. Take a look at the API reference at the end of this manual to see which functions are available. In this first sketch, we just send a NOTE ON message at regular intervals.

```
void loop() {
  MidiUart.sendNoteOn(100, 100);
  delay(800);
}
```

The complete code of this first sketch is:

```
#include <MidiUart.h>
#include <Midi.h>

void setup() {
  MidiUart.init();
}

void loop() {
  MidiUart.sendNoteOn(100, 100);
  delay(800);
}
```

Sending a potentiometer out as MIDI

In this sketch, we are actually going to build something useful. A potentiometer is connected to the analog input pin 0, and its value is going to be polled regularly in the main loop, and sent out over MIDI as a Controller Change message. The source code is pretty easy to understand, and doesn't deserve much explanations. The value returned by **analogRead()** goes from 0 to 1023, and needs to be scaled down to the parameter range of MIDI, which goes from 0 to 127. The next line is used to end out MIDI values only when the controller value changes by more than 1, to avoid sending out long strings of values induced by noise.

```
#include <MidiUart.h>
#include <Midi.h>

void setup() {
  MidiUart.init();
}

int oldPotWert = -10;
void loop() {
  int potWert = map(analogRead(0), 0, 1023, 0, 127);
  if (abs(potWert - oldPotWert) >= 2) {
    MidiUart.sendCC(1, potWert);
    oldPotWert = potWert;
  }
}
```

Receiving MIDI

Receiving MIDI data is slightly more complicated. We first have to receive bytes from the serial interface, and pass these to the actual MIDI stack so that they are interpreted and parsed. When the MIDI stack recognizes messages (it takes care of all the usually nasty stuff such as recognizing running status, handling realtime messages, synchronizing the MIDI clock, and handling sysex messages), it checks to see if a callback function has been registered. If a callback function for the current message is present, the stack calls it and passes it the recognized message as an argument.

The first step to receive MIDI data is to include the MidiDuino libraries, and instantiate a MIDI stack. This is done with the following lines.

```
#include <MidiUart.h>
#include <Midi.h>
MidiClass Midi;
```

The next step is to setup the MidiUart like in the previous sketch, and register callback functions with the MIDI stack. We want to be able to receive MIDI Note On and Note Off messages, and toggle the on-board LED of the Arduino board. To do this, we configure the **ledPin** 13 as an output, and register two functions **onNoteOnCallback()** and **onNoteOffCallback()** as callback functions.

```
void setup() {
  MidiUart.init();
  Midi.setOnNoteOnCallback(onNoteOnCallback);
  Midi.setOnNoteOffCallback(onNoteOffCallback);
  pinMode(ledPin, OUTPUT);
}
```

These two functions are very simple, and just switch on the LED when a NOTE ON message is received, without checking to see if the received message is on the correct channel or of the correct pitch, and switch off the LED when a NOTE OFF message is received.

```
void onNoteOnCallback(byte *msg) {
  digitalWrite(ledPin, HIGH);
}

void onNoteOffCallback(byte *msg) {
  digitalWrite(ledPin, LOW);
}
```

Finally, the last step is to get the data received on the serial interface, and pass those bytes to the MIDI stack. We don't use the **Serial** functionality provided by Arduino, but instead use the MidiUart functionality of the MidiDuino library.

```
void loop() {
  while (MidiUart.avail()) {
    Midi.handleByte(MidiUart.getc());
  }
}
```

MidiUart.avail() returns the number of bytes in the receive buffer of the serial interface, while **MidiUart.getc()** returns then next available byte. **Midi.handleByte()** is then called to parse the received byte.

The final code for this sketch is:

```
#include <MidiUart.h>
#include <Midi.h>
MidiClass Midi;

int ledPin = 13;

void onNoteOnCallback(byte *msg) {
  digitalWrite(ledPin, HIGH);
}

void onNoteOffCallback(byte *msg) {
  digitalWrite(ledPin, LOW);
}

void setup() {
  MidiUart.init();
  Midi.setOnNoteOnCallback(onNoteOnCallback);
  Midi.setOnNoteOffCallback(onNoteOffCallback);
  pinMode(ledPin, OUTPUT);
}

void loop() {
  while (MidiUart.avail()) {
    Midi.handleByte(MidiUart.getc());
  }
}
```

A very simple MIDI filter

We can also use these functions for input and output to create nice MIDI filters. For example, assume we want to convert incoming notes to make them match the C-major scale. We can do the necessary calculations by using the function **scalePitch()** out of the MidiTools functionality.

```
#include <MidiUart.h>
#include <Midi.h>
#include <MidiTools.h>
MidiClass Midi;

void onNoteOnCallback(uint8_t *msg) {
  MidiUart.sendNoteOn(MIDI_VOICE_CHANNEL(msg[0]),
                      scalePitch(msg[1], 0, majorScale),
                      msg[2]);
}

void onNoteOffCallback(uint8_t *msg) {
  MidiUart.sendNoteOff(MIDI_VOICE_CHANNEL(msg[0]),
                       scalePitch(msg[1], 0, majorScale),
                       msg[2]);
}

void setup() {
  MidiUart.init();
  Midi.setOnNoteOnCallback(onNoteOnCallback);
  Midi.setOnNoteOffCallback(onNoteOffCallback);
}

void loop() {
  while (MidiUart.avail()) {
    Midi.handleByte(MidiUart.getc());
  }
}
```

The possibilities are of course endless, as you can transpose tracks, filter out messages, add new messages, etc...

Sequencing with the MidiDuino library

A very important feature of the MidiDuino library is it's ability to generate a tight MIDI clock, and also to receive a MIDI clock and synchronize to it. As the MidiDuino uses a clever internal construction called a phase-locked-loop to synchronize to a MIDI clock, it can also be used to kind of "smooth" out a shaky MIDI clock signal. However, because in the standard version of the library, the **loop()** is used to handle received bytes, tight synchronization can be a bit off. The separate arduinomidi core (see Installation above) is used to handle clock synchronization bytes as soon as they are received in the interrupt routine, so that if you want to have tight external synchronization, you may want to install the separate core.

The MIDI sequencing environment is handled by an object called **MidiClock**. It can be used to generate an internal clock, or used to synchronize to an external clock on **MidiUart**. It can then be used to call a callback on every 16th note, allowing you to run a function at regular intervals, and for example produce rhythms or melodic patterns. At the moment the sequencing environment is very simple, supporting only 16th notes and no swing, and not allowing for a very easy way to trigger notes of different lengths, and geared toward techno, but we are working on a very cool and flexible

sequencing environment. The reason for this is that the MidiDuino library was first developed to interface with the Elektron MachineDrum, which is a pretty interesting 16-step sequencer that is often used to produce techno.

This sketch is a very simple sequencer sending out a MIDI note on every quarter note. It also blinks the on-board LED on every quarter note.

The internal clock is very easy to setup. The **MidiClock.mode** variable sets if the clock is generated internally (by setting it to **MidiClock.INTERNAL_MIDI**), or if it syncs to an external clock source, which needs to send START and STOP commands as well, by setting it to **MidiClock.EXTERNAL_MIDI**. In the case of an internally generated clock, we need to set the tempo by hand by calling **MidiClock.setTempo()**. At the moment these values are not yet really exact, so setting it to 100 bpm is more akin to setting it to 105 bpm, this will be fixed very soon. We can enable the sending of the MIDI clock signals on the output of the Arduino by setting the variable **MidiClock.transmit** to true. This can be used to chain out the MIDI clock as well (as the MiniCommand has no THRU). Finally, we enable the 16th note callback by calling **MidiClock.setOn16Callback()**. The following code is called in the **setup()** function.

```
MidiClock.mode = MidiClock.INTERNAL_MIDI;
MidiClock.setTempo(100);
MidiClock.setOn16Callback(on16Callback);
MidiClock.start();
```

Finally, here is the whole sketch. As you can see, sequencing on the MidiDuino library is very easy, and a future API will allow for a much easier scheduling of events like notes of a certain duration, swing, and events that are not on the tempo grid. Using external synchronization is just a matter of changing **MidiClock.INTERNAL_MIDI** to **MidiClock.EXTERNAL_MIDI**.

```
#include <MidiUart.h>
#include <Midi.h>
#include <MidiClock.h>
MidiClass Midi;

int ledPin = 13;

void on16Callback() {
  if ((MidiClock.div16th_counter % 4) == 0) {
    digitalWrite(ledPin, HIGH);
    MidiUart.sendNoteOn(100, 100);
  } else if ((MidiClock.div16th_counter % 4) == 2) {
    digitalWrite(ledPin, LOW);
    MidiUart.sendNoteOff(100);
  }

}

void setup() {
  pinMode(ledPin, OUTPUT);
  MidiUart.init();
  MidiClock.mode = MidiClock.INTERNAL_MIDI;
  MidiClock.setTempo(100);
  MidiClock.transmit = true;
  MidiClock.setOn16Callback(on16Callback);
  MidiClock.start();
}

void loop() {
  while (MidiUart.avail()) {
    Midi.handleByte(MidiUart.getc());
  }
}
```

Mididuino API reference

MIDI Functions

Sending Messages

- **MidiUart.init**(), **MidiUart**(long speed)
    - Initialize the serial interface to activate buffered sending, setting MIDI speeding, and setting up the clock for MIDI synchronization
- **MidiUart.sendNoteOn**(uint8_t channel, uint8_t note, uint8_t velocity), **MidiUart.sendNoteOn**(uint8_t note, uint8_t velocity)
    - send a note on message
- **MidiUart.sendNoteOff**(uint8_t channel, uint8_t note), **MidiUart.sendNoteOff**(uint8_t note)
    - send a note off message, you may need to use **sendNoteOn()** with a velocity of 0 on most modern synthesizers
- **MidiUart.sendCC**(uint8_t channel, uint8_t cc, uint8_t value), **MidiUart.sendCC**(uint8_t cc, uint8_t value)
    - send a controller change message
- **MidiUart.sendProgramChange**(uint8_t channel, uint8_t program), **MidiUart.sendProgramChange**(uint8_t program)
    - send a program change message
- **MidiUart.sendPolyKeyPressure**(uint8_t channel, uint8_t note, uint8_t pressure), **MidiUart.sendPolyKeyPressure**(uint8_t note, uint8_t pressure)
    - send a polyphonic key pressure message
- **MidiUart.sendChannelPressure**(uint8_t channel, uint8_t pressure), **MidiUart.sendChannelPressure**(uint8_t pressure)
    - send a channel pressure message
- **MidiUart.sendPitchBend**(uint8_t channel, int16_t pitchbend), **MidiUart.sendPitchBend**(int16_t pitchbend)
    - send a pitch bend message
- **MidiUart.sendNRPN**(uint8_t channel, uint16_t parameter, uint8_t value), **MidiUart.sendCC**(uint16_t parameter, uint8_t value)
    - send a NRPN message
- **MidiUart.sendRPN**(uint8_t channel, uint16_t parameter, uint8_t value), **MidiUart.sendCC**(uint16_t parameter, uint8_t value)

    - send a RPN message
- **MidiUart.sendRaw**(uint8_t *msg, uint8_t cnt), **MidiUart.sendRaw**(uint8_t byte)
    - send raw bytes
- **MidiUart.currentChannel**
    - Stores the default MIDI channel. Assign a new value to this variable to change the default channel.
- **MidiUart.useRunningStatus**
    - Set to true to enable running status on the output of the Arduino. This will save up a lot of transmission data if you are using a lot of notes.
- **MidiUart.resetRunningStatus()**
    - Use this to reset the running status and allow the next status byte to be sent in full

Receiving Messages

- void **midiCallback**(uint8_t *msg)
    - prototype for a MIDI callback function
- **Midi.setOnNoteOnCallback**(midi_callback_t callback)
    - set a callback for receiving note on messages (check for velocity == 0 -> note off message)
- **Midi.setOnNoteOffCallback**(midi_callback_t callback)
    - set a callback for receiving note off messages
- **Midi.setOnControlChangeCallback**(midi_callback_t callback)
    - set a callback for receiving controller change messages
- **Midi.setOnAfterTouchCallback**(midi_callback_t callback)
    - set a callback for receiving aftertouch messages
- **Midi.setOnChannelPressureCallback**(midi_callback_t callback)
    - set a callback for receiving channel pressure messages
- **Midi.setOnProgramChangeCallback**(midi_callback_t callback)
    - set a callback for receiving program change messages
- **Midi.setOnPitchWheelCallback**(midi_callback_t callback)
    - set a callback for receiving pitchwheel messages

Midi Clock and Midi Synchronization

- **MidiClock.mode** = **MidiClock.EXTERNAL_MIDI**

    - set synchronization to external sync
- **MidiClock.mode** = **MidiClock.INTERNAL_MIDI**
    - set synchronization to internal sync
- **MidiClock.transmit** = true / false
    - activate sending synchronizaton on MidiUart
- **MidiClock.start**() / **MidiClock.stop**() / **MidiClock.pause**()
    - control the clock engine
- **MidiClock.setTempo**(uint16_t tempo)
    - set the tempo when using internal synchronization (not correctly mapped yet)
- uint16_t **MidiClock.getTempo**()
    - get the tempo of the midi synchronization (a bit flaky)
- **MidiClock.setOn16Callback**(void (*callback)())
    - set a function to be called on each 16th note (in interrupt, so keep callback short)

Sequencing

- **DrumTrack**(uint32_t pattern, uint8_t len = 16, uint8_t offset = 0)
    - create a drum pattern
- bool drumTrack.**isHit**(uint8_t pos)
    - check if there is a hit at a certain position in the pattern
- **PitchTrack**(DrumTrack *track, uint8_t len)
    - create a pitch track linked to a drum pattern
- pitchTrack.**pitches**[]
    - array storing the note pitches of the pitch track
- pitchTrack.**playHit**(uint8_t pos)
    - play the pitch at a certain position (if there is a hit)
- **EuclidDrumTrack**(uint8_t pulses, uint8_t len, uint8_t offset = 0)
    - euclidean drum track sequencer
- **euclidDrumTrack.setEuclid**(uint8_t pulses, uint8_t len, uint8_t offset = 0)
    - reset the euclidean drum track parameters