The MidiDuino Library: MIDI for the Arduino

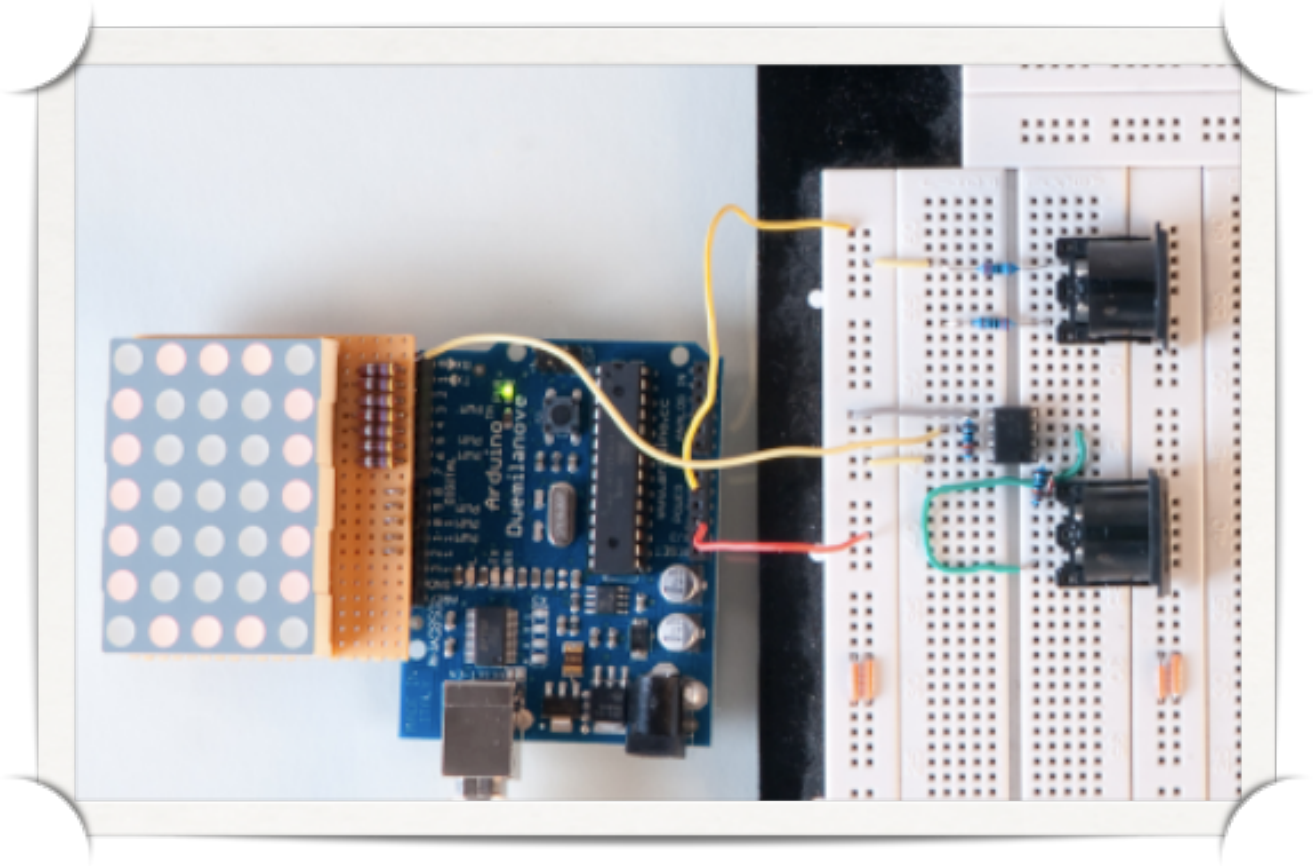


Table of Contents

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/ .

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

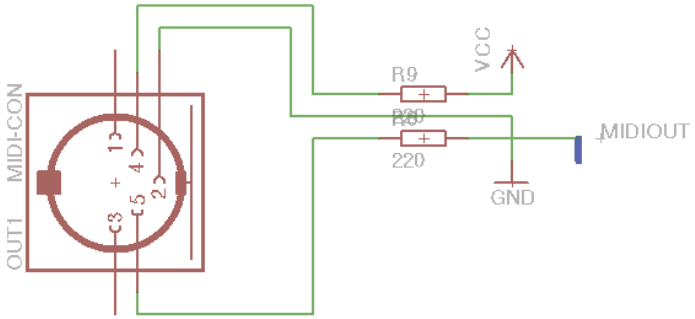
At Ruin & Wesen, we build a device called the MiniCommand, that happens to have a microcontroller (the AVR atmeg64) similar to the microcontroller on the Arduino board (the Duemilanove has an AVR atmega168 or atmega328). However, the MiniCommand at first wasn't programmed using the Arduino environment. 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 .

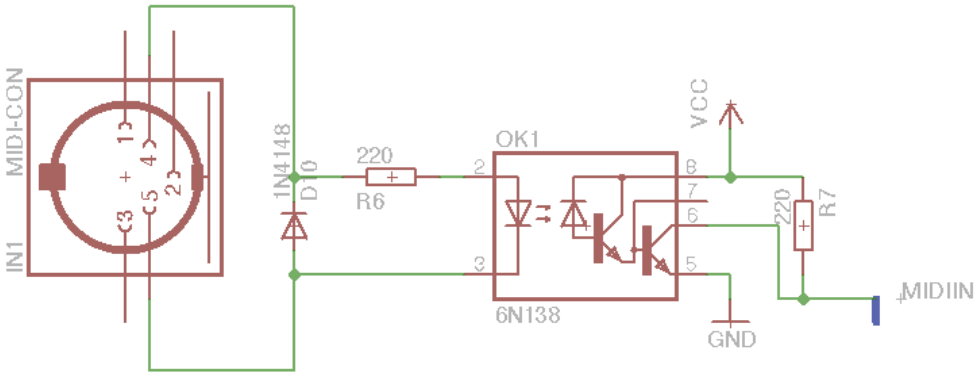
Hardware Description

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.

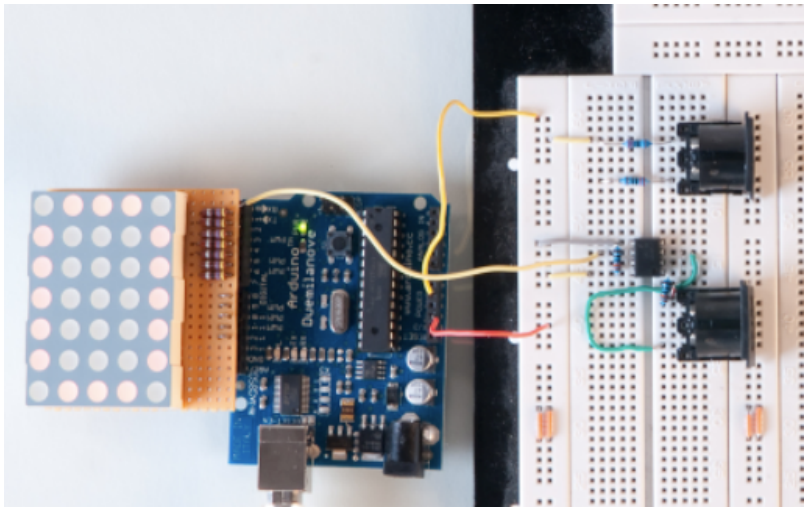
In order to send MIDI data, you just need to connect the TX pin of the Arduino to the MIDI connector (a DIN 5-pin 180 degrees connector) over a 220 Ohm resistor:



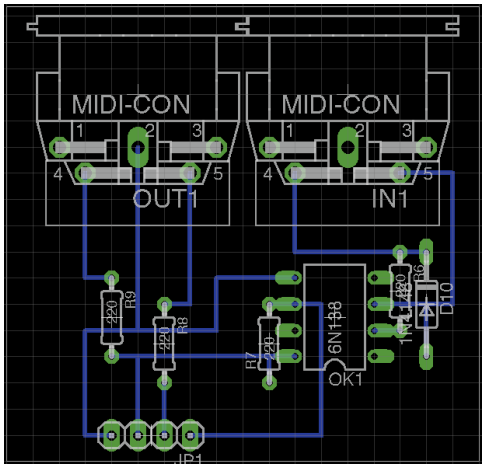
Receiving MIDI is slightly more complicated, and is usually done with a chip called an optocoupler. You may get lucky by connecting the input pin directly to RX, and hoping both devices have the same ground, but the usual circuit is the following one. You need a diode and a few 220 Ohm resistors.



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.



Software Installation

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

```
void handleGui() {
	if (BUTTON_PRESSED(Buttons.BUTTON1)) {
		MidiUart.sendNoteOn(1, 100);
	}
	if (BUTTON_RELEASED(Buttons.BUTTON1)) {
		MidiUart.sendNoteOff(1);
	}
}
```

Receiving MIDI

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 <MidiTools.h>

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() {
      Midi.setOnNoteOnCallback(onNoteOnCallback);
      Midi.setOnNoteOffCallback(onNoteOffCallback);
}

void loop() {
      GUI.updatePage();
      GUI.update();
}

void handleGui() {
}
```

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 MiniCommand is it's ability to generate a tight MIDI clock, and also to receive a MIDI clock and synchronize to it. At the moment, only the first input can be used to receive MIDI clock information, but this will soon be extended to allow the second MIDI input to be used as well. As the MiniCommand 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.

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 MiniCommand was really developed with the MachineDrum in mind, and the flexibility of the environment only came up later.

In this sketch, we are going to generate a very simple random melodic pattern. The user can set the length of the pattern using an encoder, and randomize the played notes by pressing a button. To do this we create an array **melodyNotes[]**

containing the pitches of the notes to be sent. The callback function loops over this array, constrained by the length set by the encoder **lengthEncoder**. The notes in the array are randomized by the function **randomizeNotes()**, which generates arbitrary melodies over a two-octave major scale. The **on16Callback()** function is called on every 16th note, and uses the global 16th count variable called **MidiClock.div16th_counter**, which counts the number of 16th note since the **MidiClock** was last started.

```
RangeEncoder lengthEncoder(1, 16, “LEN”);
uint8_t melodyNotes[16] = { 0 };

void randomizeNotes() {
	for (int i = 0; i < countof(melodyNotes); i++) {
		melodyNotes[i] = scalePitch(48 + random(24), 0, majorScale);
	}
}

uint8_t prevNote = 0;

void on16Callback() {
	Midi.sendNoteOff(prevNote);
	prevNote = melodyNotes[MidiClock.div16th_counter %
				lengthEncoder.getValue()];
	Midi.sendNoteOn(prevNote, 100);
}
```

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**), 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**. 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 MiniCommand 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()**.

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

Finally, here is the whole sketch, which randomizes the notes on a button press. As you can see, sequencing on the MiniCommand 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. To see more complicated examples of sequencing, take a look at the MDArpeggiator sketch or at the MDPitchEuclid sketch. The SupaTrigga reverse functionality of the MDWesenLivePatch also relies on the tempo synchronization.

```
#include <MidiTools.h>

RangeEncoder lengthEncoder(1, 16, "LEN");
uint8_t melodyNotes[16] = { 0 };

void randomizeNotes() {
	for (int i = 0; i < countof(melodyNotes); i++) {
		melodyNotes[i] = scalePitch(48 + random(24), 0, majorScale);
	}
}

uint8_t prevNote = 0;
void on16Callback() {
	MidiUart.sendNoteOff(prevNote);
	prevNote = melodyNotes[MidiClock.div16th_counter %
				lengthEncoder.getValue()];
	MidiUart.sendNoteOn(prevNote, 100);
}

EncoderPage page(&lengthEncoder);

void setup() {
	randomizeNotes();
	lengthEncoder.setValue(8);
	MidiClock.mode = MidiClock.INTERNAL;
	MidiClock.setTempo(100);
	MidiClock.setOn16Callback(on16Callback);
	MidiClock.start();
	 GUI.setPage(&page);
}

void loop() {
	GUI.updatePage();
	GUI.update();
}

void handleGui() {
	if (BUTTON_PRESSED(Buttons.BUTTON1)) {
		randomizeNotes();
	}
}
```

External synchronization

sldkf

Building a WiidiMote

sldf

Building a capacitive touch MIDI instrument

lsdkfjl

Building a polyrhythmic sequencer

dlfgkj

Mididuino API reference

MIDI Functions

Sending Messages

- **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