Erik Bartmann

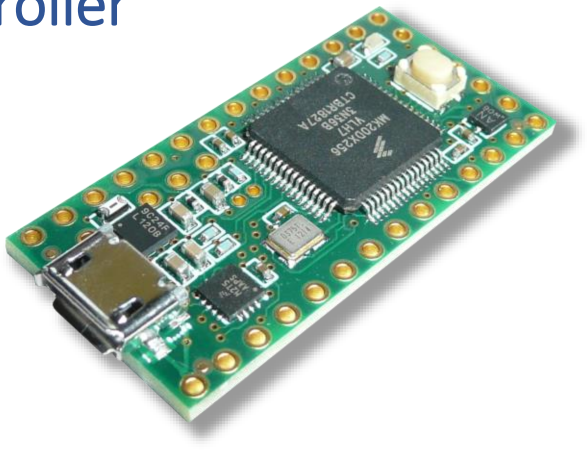
Discover modulare Synthesizer with VCV-Rack 2

The Teensy MIDI Controller

Author	Erik Bartmann
Internet	https://erik-bartmann.de/
Topic	The Teensy MIDI controller
Version	1.01
Date	January 26, 2022

Table of contents

A Teensy MIDI Controller



What is it all about?

The following topics are discussed in this paper.

- What is a MIDI controller?
- What is a teensy board?
- What is MIDI?
- The Teensy MIDI Controller
- The MIDIView program
- A test with the VCV Rack

When it comes to getting control of music software on the computer via external hardware, so-called MIDI controllers come into play. This paper is about developing a rudimentary MIDI controller with a microcontroller and some other components like pushbuttons, resistors and potentiometers.

The Teensy-Board

A large number of microcontrollers can be found on the market and especially in the hobby area the Arduino board with its countless variants has become a quasi-standard. The included Arduino development environment (Arduino IDE) enables even newcomers to find a suitable start without a steep learning curve. But the board we are talking about now is not an Arduino board, but the so-called *Teensy* board. It fits perfectly on a breadboard and is in many ways more powerful than a standard Arduino board like the Arduino-Uno. The Teeny-Board is developed by *Paul Stoffregen* and is optimal for building a USB-MIDI controller, because it is very easy to use and configure as a USB-MIDI device.

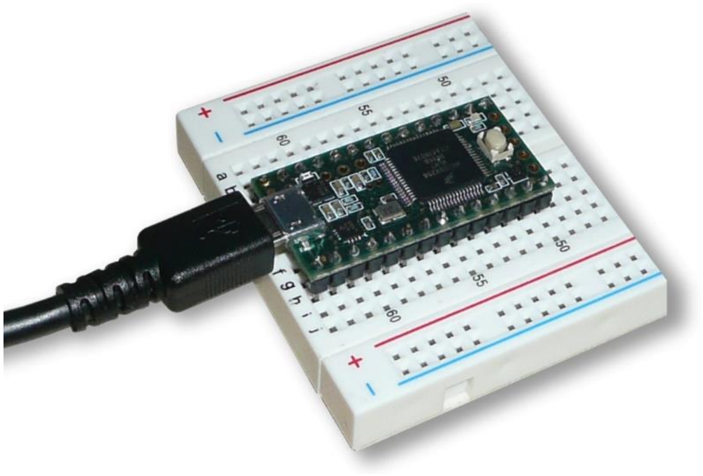


Figure 1 The Teensy 3.2 on a breadboard

But let's approach the whole thing step by step.

The installation of the development software

I had just briefly mentioned the Arduino IDE, which provides a very easy introduction to programming Arduino microcontrollers. However, there is another and crucial advantage. This IDE can be extended via external software so that other microcontrollers can also be programmed via it. This is also the case with the Teensy board. But of course the Arduino IDE has to be installed first.

Step 1: The installation of the Arduino IDE

The Arduino development environment for different platforms such as Windows, Linux and MacOS X can be downloaded via the following link.

Hyperlink!

https://www.arduino.cc/en/software

After the simple installation, the second step is now to install the Teensy extension.

Step 2: The installation of the Teensy extension

The following link can be used to download the Teensy installation software.

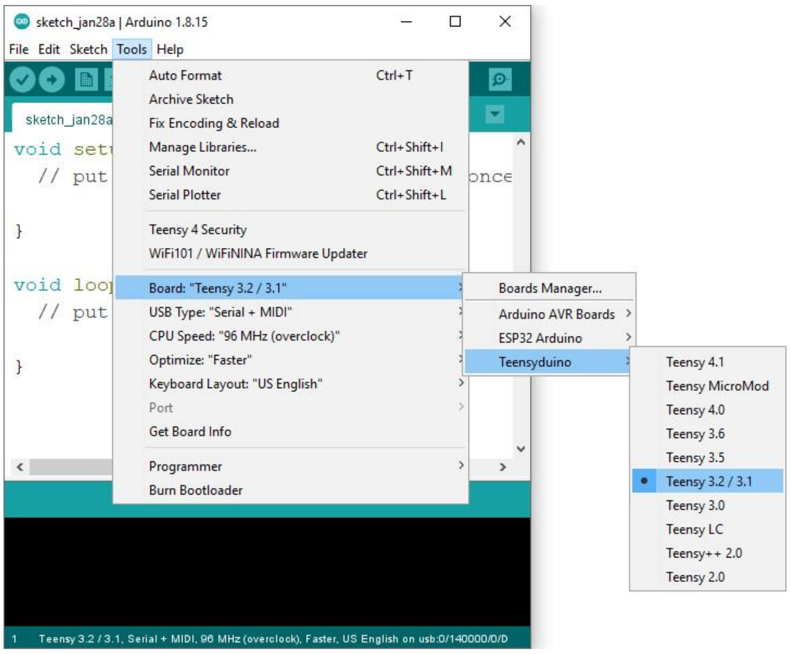
Hyperlink!

https://www.pjrc.com/teensy/td_download.html

After the also very simple installation, a first test of the Teensy board can be performed.

Step 3: The first test of the Teensy board

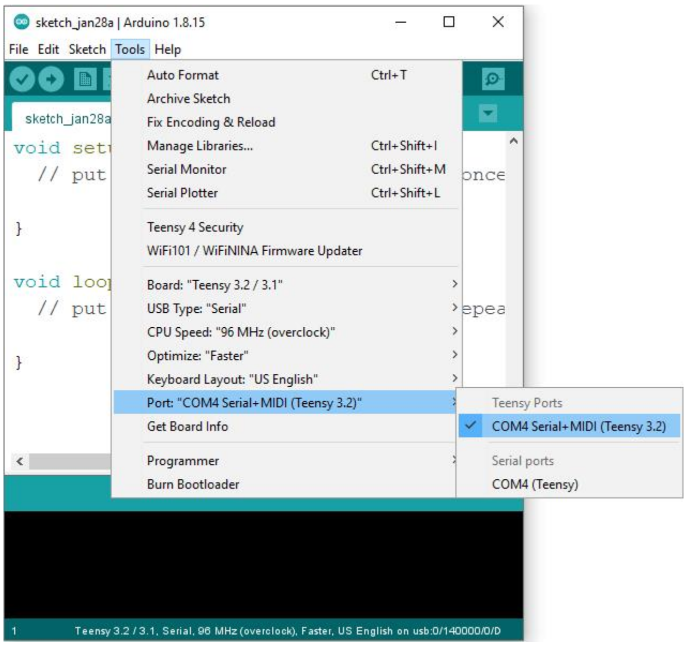
Of course, the Teensy board must be connected to the computer via a suitable USB cable. After the connection is made, the board is automatically recognized, so no additional driver is required. The installation software of the Teensy extension for the Arduino IDE has all the necessary components to start programming immediately. To begin, of course, you need to make sure that the correct board is selected in the Arduino IDE. Since I am using the Teensy board 3.2, this looks like this. Under the menu item *Tools|Board* this board has to be selected.



```
1    Teensy 3.2 / 3.1, Serial + MIDI, 96 MHz (overclock), Faster, US English on usb:0/140000/0/D
```

Figure 2 The Teensy 3.2 board is selected

The next step is to select the correct communication port via the menu item *Tools|Port*. Usually this is the only port that is offered for selection there and a somewhat cryptic one and may also be different from what I have shown here.


sketch_jan28a | Arduino 1.8.15
File Edit Sketch Tools Help
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
Teensy 4 Security
WiFi101 / WiFiNINA Firmware Updater
Board: "Teensy 3.2 / 3.1"
USB Type: "Serial"
CPU Speed: "96 MHz (overclock)"
Optimize: "Faster"
Keyboard Layout: "US English"
Port: "COM4 Serial+MIDI (Teensy 3.2)"
Get Board Info
Programmer
Burn Bootloader
Teensy Ports
✓ COM4 Serial+MIDI (Teensy 3.2)
Serial ports
COM4 (Teensy)

```
1    Teensy 3.2 / 3.1, Serial, 96 MHz (overclock), Faster, US English on usb:0/140000/0/D
```

Figure 3 Der Teensy-Port ist ausgewählt

In the last step, the obligatory *Blink* sketch can be loaded for a first test. This is done via the menu item *File|Examples|Teensy|Tutorial1|Blink*.

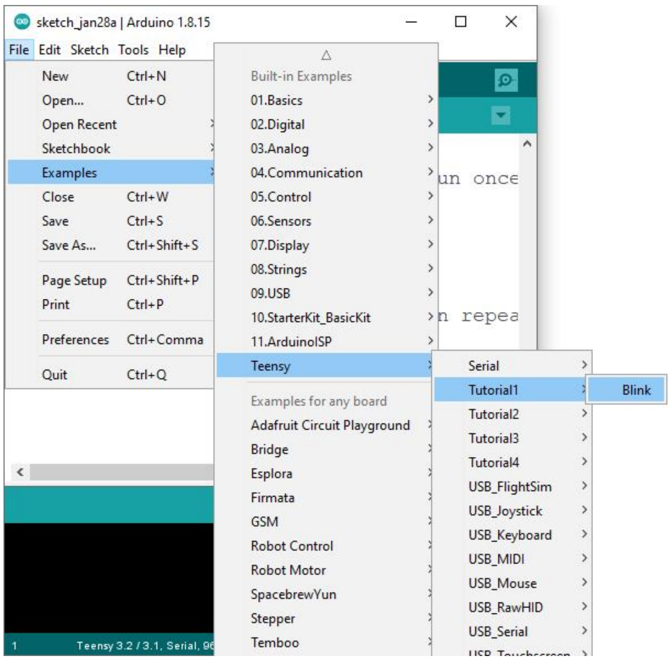


Figure 4 Loading the Blink Sketch

Der Sketch schaut wie folgt aus.

```
      const int ledPin = 13; // LED-Pin

      void setup() {
        pinMode(ledPin, OUTPUT); // Output-Pin
      }

      void loop() {
        digitalWrite(ledPin, HIGH);    // LED on
        delay(1000);                   // 1 Sec. Pause (1000ms)
        digitalWrite(ledPin, LOW);     // LED off
        delay(1000);                   // 1 Sec. Pause (1000ms)
}
```

After uploading the Blink sketch, a small Teensy utility will pop up to help you upload and launch the sketch.

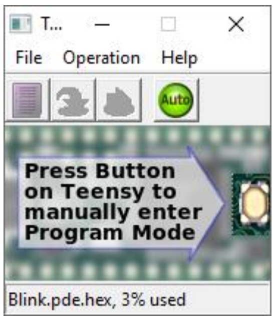


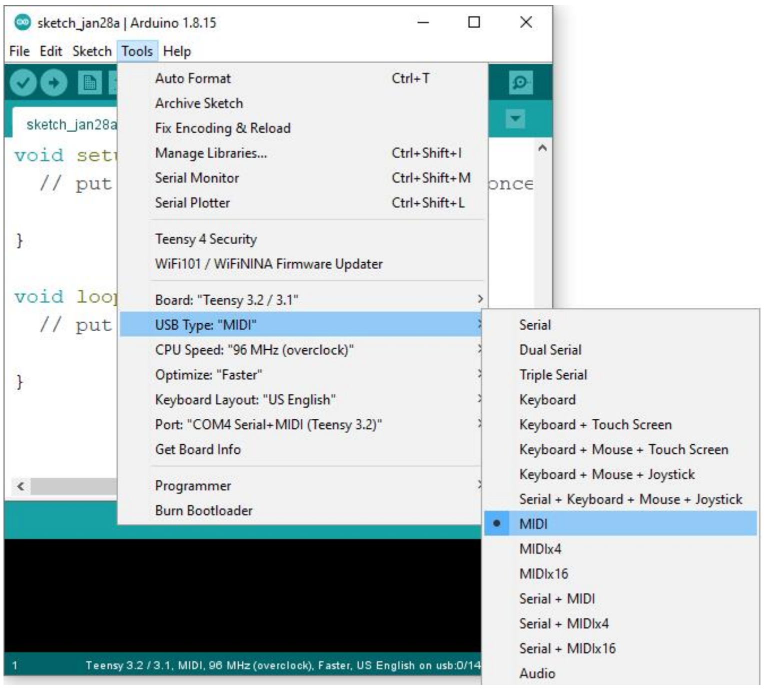
Figure 5 The Teensy Help Program

It may be necessary to press the small white push button on the board to effect a reset, but this is not usually required. After the upload has been completed, a small light emitting diode (LED) on the

Teensy board should start flashing every second. If this is the case, everything has obviously been done correctly and is a sign that the communication between the computer and the microcontroller board is working fine.

The configuration as a MIDI device

The Teensy board should appear as a MIDI device and for this a special configuration is necessary. This is selected under the menu item *Tools|USB-Type|MIDI*.


sketch_jan28a | Arduino 1.8.15
File Edit Sketch Tools Help
sketch_jan28a
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
Teensy 4 Security
WiFi101 / WiFiNINA Firmware Updater
Board: "Teensy 3.2 / 3.1"
USB Type: "MIDI"
CPU Speed: "96 MHz (overclock)"
Optimize: "Faster"
Keyboard Layout: "US English"
Port: "COM4 Serial+MIDI (Teensy 3.2)"
Get Board Info
Programmer
Burn Bootloader
Serial
Dual Serial
Triple Serial
Keyboard
Keyboard + Touch Screen
Keyboard + Mouse + Touch Screen
Keyboard + Mouse + Joystick
Serial + Keyboard + Mouse + Joystick
MIDI
MIDIx4
MIDIx16
Serial + MIDI
Serial + MIDIx4
Serial + MIDIx16
Audio
Teensy 3.2 / 3.1, MIDI, 96 MHz (overclock), Faster, US English on usb:0/14

```
void set
  // put
}

void loo
  // put
}
```

Figure 6 The correct USB type

The Teensy as MIDI controller

Now it's time to move on from the simple Blink sketch and do something about MIDI. MIDI is a digital interface for musical instruments and stands for ***M**usical **I**nstrument **D**igital **I**nterface*. More details can be found in a special tutorial "What is MIDI?" on my website.

Hyperlink!

https://erik-bartmann.de/userfiles/downloads/Musik/VCV-Rack/EN_MIDI.pdf

Also here a first test can take place, because it should be ensured that also everything runs in correct ways. This MIDI test consists of two parts, where the first one should show how the sent MIDI information looks like on the lowest level and in the second part something can be heard.

First MIDI test

For the first MIDI test, a suitable software must be installed to display the MIDI information. This software is freely available, is called *MIDIView* and can be obtained from the following link.

Hyperlink!

https://hautetechnique.com/midi/midiview/

It is a so-called monitor program that displays all sent MIDI data. But more about that later. First of course a suitable Teensy sketch must be programmed, so that something can be seen and later heard. The Teensy board should now send standard MIDI messages. Sounds a bit strange, doesn't it?! Quite simple! Such a message can for example look like that a note should be played and then fade away. This is called in detail

- Note on
- Note off

Such events are of course supported in Teensy MIDI programming. They are

- `usbMIDI.sendNoteOn(note, velocity, channel);`
- `usbMIDI.sendNoteOff(note, velocity, channel);`

The three arguments when calling such a method (methods are actually functions and are so called in ***O**bject-**O**riented **P**rogramming*) have the following task.

- *note*: Pitch
- *velocity*:
- *channel*: the used MIDI channel

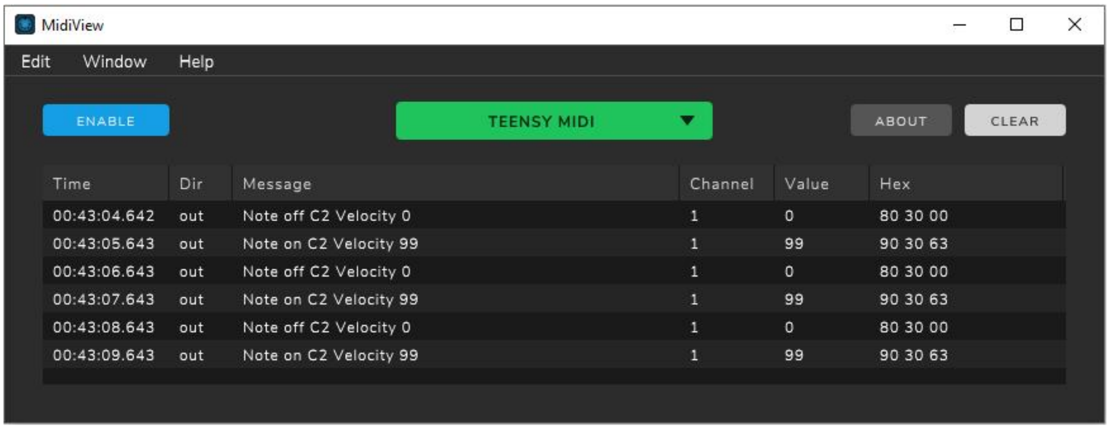
Now you can very easily extend the used Blink Sketch with these two methods to play a note and then not again.

```
const int ledPin = 13; // LED-Pin

void setup() {
  pinMode(ledPin, OUTPUT); // Output-Pin
}

void loop() {
  digitalWrite(ledPin, HIGH);      // LED on
  usbMIDI.sendNoteOn(48, 99, 1);   // Note on (C2 / C3)
  delay(1000);                     // 1 Sec. Pause
  digitalWrite(ledPin, LOW);       // LED off
  usbMIDI.sendNoteOff(48, 0, 1);   // Note off (C2 / C3)
  delay(1000);                     // 1 Sec. Pause
}
```

It can be seen that when a note is muted, the velocity value 0 is simply used, meaning that the key is not struck at all on a keyboard. So the following line could also have been used.

```
      ...
        usbMIDI.sendNoteOn(48, 0, 1);
      ...
```

The Sketch reacts as before that further the small LED flashes every second, but now also in this interval the two messages Note on and Note off are sent over the methods mentioned. A look into the program *MIDIView* shows the following continuous messages.



Time	Dir	Message	Channel	Value	Hex
00:43:04.642	out	Note off C2 Velocity 0	1	0	80 30 00
00:43:05.643	out	Note on C2 Velocity 99	1	99	90 30 63
00:43:06.643	out	Note off C2 Velocity 0	1	0	80 30 00
00:43:07.643	out	Note on C2 Velocity 99	1	99	90 30 63
00:43:08.643	out	Note off C2 Velocity 0	1	0	80 30 00
00:43:09.643	out	Note on C2 Velocity 99	1	99	90 30 63

Figure 7 The MIDI messages in MIDIView

In the *Message* column you can see that the note with pitch C2 was recognized. In the corresponding Teensy sketch, however, I wrote C2 and C3 respectively as comments. Why this difference of one octave? The reason is that there are different standards, which actually contradicts the standard to be a standard. Confusing, isn't it? There are two standards, with the so-called *Middle-C* being C3 at *Yamaha* and C4 at *Roland*. So the MIDI note value used in the sketch, which is given as 48, represents the pitch C2 with a frequency of 130.81Hz. I prepared a VCV rack patch for this, using *NYSTHI's HOT TUNA* plugin to display both the frequency and the detected note in it. But I'll talk about this patch in more detail.

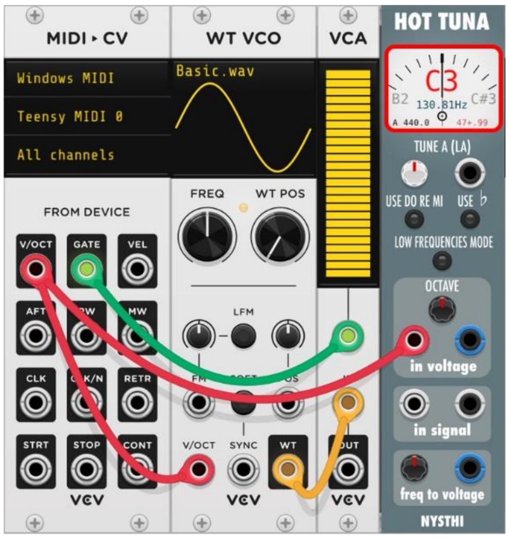


Figure 8 The VCV rack patch for displaying note and frequency

A small MIDI controller

Now we come to something that can really be brought into real operation. It is about a very simple MIDI controller, which is equipped with four push buttons and two knobs. This can of course be expanded as desired. I think this little project makes you want to do more. Let's get started. For this a bit of theory, which can be read in detail in the tutorial "*What is MIDI*?" on my website. There are eight different categories or command types in MIDI, as you can see in the following table.

Command type	Status-Byte (binary)	Status (hex)
Note off	1000 *nnnn*	8*n*
Note on	1001 *nnnn*	9*n*
Poly Pressure	1010 *nnnn*	A*n*
Control Change	1011 *nnnn*	B*n*
Program Change	1100 *nnnn*	C*n*
Channel Pressure	1101 *nnnn*	D*n*
Pitch-Bend	1110 *nnnn*	E*n*

Table 1 MIDI command types

The first two were already used in the first MIDI test, which is used to send notes. But now there is the command type *Control Change* (CC), which is there to modify any parameters of a synthesizer (modulation). This can be the case, for example, when the pitch or the cut-off frequency is changed. This is then usually done via knobs, which now come into play. The planned MIDI controller should therefore have buttons for controlling the pitch and pots for adjusting various parameters, as can be seen schematically in the following illustration.

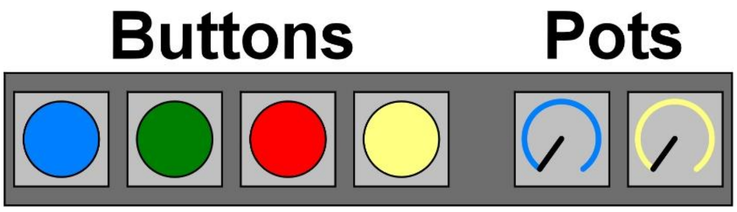


Figure 9 The controls of the MIDI controller

If one of the four keys is pressed and released again, a corresponding *note on* or *note off* command with a certain pitch and velocity value is to be sent. The situation is similar for the knobs. There the respective position is to be determined, in order to use this then for the manipulation of a desired parameter. So that MIDI information is not sent continuously, in the end it only happens when a value really changes. Especially with the potentiometers this is a problem I will come to. Regarding a control change it has to be mentioned that it has to be identified in a MIDI controller of course. After all, it should be clear which knob was moved at all. For this reason, certain *CC-ID*s are assigned to these elements. For our project these are the following values.

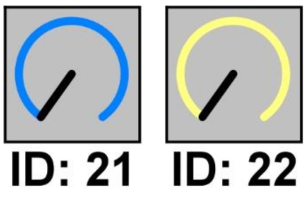


Figure 10 The CC-IDs of the two potentiometers

The mentioned controller elements, i.e. pushbuttons and knobs, must now be connected to the Teensy board in some way. For this purpose, it should be clear beforehand which individual connections on the board have which functions.

The Pin Layout of the Teensy 3.2

Let's now take a closer look at the so-called pinout of the Teensy 3.2, where I more or less limit myself to the pin groups required for the project.

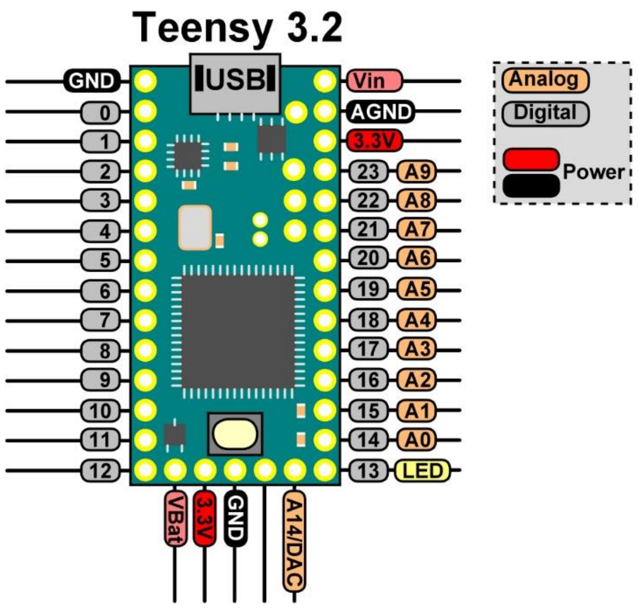


Figure 11 The simplified pin-out of the Teensy 3.2

It can be seen that basically there are two different pin groups. On the one hand the analog pins for measuring continuous voltage values, which are supplied via the rotary controls, and the digital pins for querying whether a button is pressed or not pressed.

The circuit diagram

For the two rotary controls I will use the analog pins *A0* and *A1* and connect the push buttons to the digital pins *0*, *1*, *2* and *3*. Let's have a closer look in a circuit diagram that shows all electrical connections.

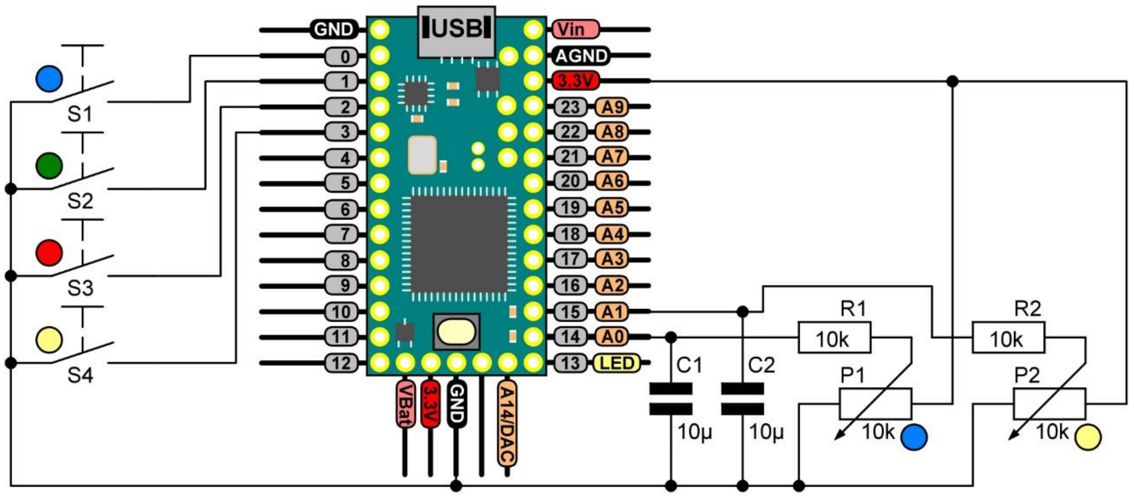


Figure 12 The circuit diagram of the MIDI controller

I would now like to address a problem that I had already briefly mentioned with regard to the knobs. MIDI information should only be sent when a value changes. Let's assume that there is a resistance value of 500Ω at a potentiometer. This is then converted by the program into a corresponding voltage value and processed further. A corresponding MIDI information is sent. If this value does not change, because the position of the knob is no longer changed, no further MIDI transmission should take place. Now, however, a potentiometer has the property that it always swings back and forth between two resistance values due to mechanical properties or small contaminations, perhaps at the current position, which is called *jitter*. So the continuous measurement results are, for example, 500Ω, 501Ω, 500Ω, 499Ω, 500Ω, etc. Of course, this leads to the program thinking that the knob is constantly being moved and as a result continuously sends unwanted MIDI information. Now you can try to get this under control programmatically. In addition, however, I advise a small extension in the form of a so-called low-pass filter, which can also be seen in the schematic and was realized by the additional resistors and capacitors. Here is the schematic of a potentiometer with downstream low-pass filter.

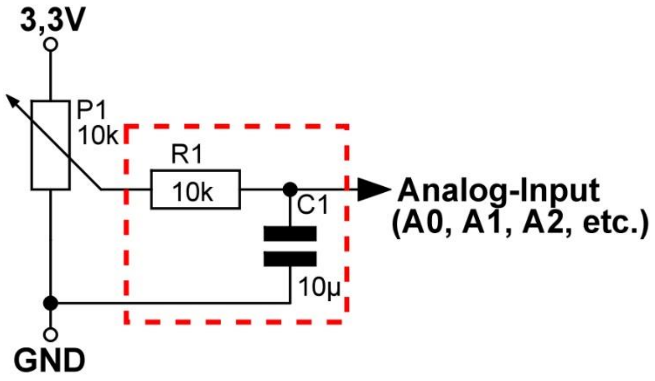


Figure 13 The low pass filter - outlined in red

This filter causes a derivation of the signal to ground (GND) in case of very fast changes, so that they are not passed on to the analog input and only low frequencies can pass. The mechanical pushbuttons also have an unpleasant side effect. They can emit several logical levels in succession for a short time instead of just one when closing and opening the contact. This behavior is called *bouncing* and is compensated by a corresponding library. Now let's have a look at the construction on a breadboard.

The assembly on the breadboard

With the mentioned components and the clear wiring, the assembly on a breadboard is very quickly implemented.

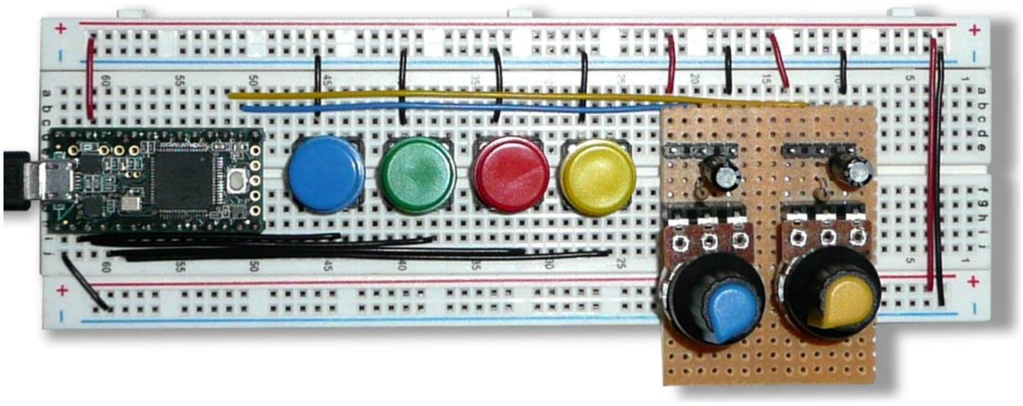


Figure 14 The MIDI controller on a breadboard

The required components are.

- Teensy 3.2
- Breadboard 1x
- Micro pushbutton 4x
- Potentiometer 10K linear 2x
- Resistors 10K 2x
- Capacitors 10µ 2x
- board rest + pin headers (only if desired)
- Cable

The Teensy Sketch

The starting point for the realization is an example sketch, which is located at the following menu item of the Arduino IDE.

File|Examples|Teensy|USB_MIDI|Many_Button_Knobs

This sketch was modified by me to work with respect to 4 pushbuttons and 2 knobs. I do not show the sketch here in its full length, but only the modifications.

Determination of the number of analog and digital pins:

```
    ...
    const int A_PINS = 2; // number of Analog PINS
    const int D_PINS = 4; // number of Digital PINS
    ...
```

Definition of the analog input pins and the control IDs of the potentiometers:

```
...
const int ANALOG_PINS[A_PINS] = {A0, A1}; // analog Input-Pins
const int CCID[A_PINS] = {21, 22};         // Control-ID
...
```

Defining the digital input pins and the MIDI note values:

```
...
const int DIGITAL_PINS[D_PINS] = {0, 1, 2, 3}; // digital Input-Pins
const int note[D_PINS] = {60, 61, 62, 63};     // MIDI-Note-Values
...
```

Determination of the analog input pins with regard to the minimization of the jitter effect:

```
...
ResponsiveAnalogRead analog[]{
  {ANALOG_PINS[0],true},
  {ANALOG_PINS[1],true},
  {ANALOG_PINS[2],true},
  {ANALOG_PINS[3],true}
};
...
```

Definition of the digital input pins for debouncing the keys:

```
...
Bounce digital[] =    {
   Bounce(DIGITAL_PINS[0],BOUNCE_TIME),
   Bounce(DIGITAL_PINS[1], BOUNCE_TIME),
   Bounce(DIGITAL_PINS[2], BOUNCE_TIME),
   Bounce(DIGITAL_PINS[3], BOUNCE_TIME)
};
...
```

These changes mean that the sketch is now matched to the hardware and cabling used. After uploading the sketch, a first MIDI controller test can be performed using the *MIDIView* program.

Testing the MIDI controller

After starting MIDIView and selecting the Teensy MIDI controller with the name *TEENSY MIDI* the test can be started. At the beginning the message window should be empty and there should be no MIDI information running up there.

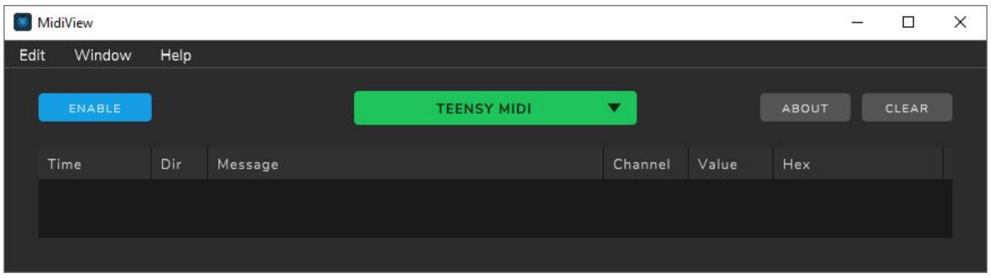
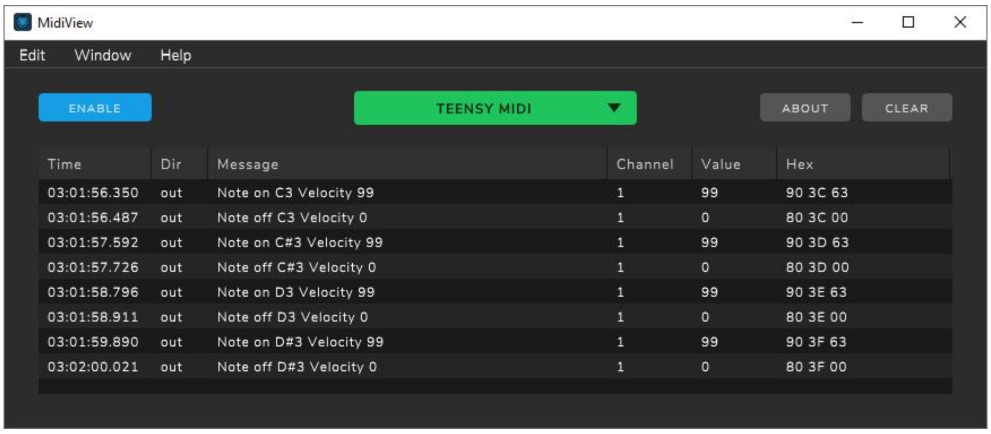


Figure 15 An empty MIDIView window

Now I will press the keys one after the other from left to right and see what is happening in MIDIView.



Time	Dir	Message	Channel	Value	Hex
03:01:56.350	out	Note on C3 Velocity 99	1	99	90 3C 63
03:01:56.487	out	Note off C3 Velocity 0	1	0	80 3C 00
03:01:57.592	out	Note on C#3 Velocity 99	1	99	90 3D 63
03:01:57.726	out	Note off C#3 Velocity 0	1	0	80 3D 00
03:01:58.796	out	Note on D3 Velocity 99	1	99	90 3E 63
03:01:58.911	out	Note off D3 Velocity 0	1	0	80 3E 00
03:01:59.890	out	Note on D#3 Velocity 99	1	99	90 3F 63
03:02:00.021	out	Note off D#3 Velocity 0	1	0	80 3F 00

Figure 16 The MIDI messages of the 4 buttons

It is wonderful to see that there is both a note on and note off event per button and that the MIDI note values defined in the Sketch are interpreted here. These were the values defined via the following line.

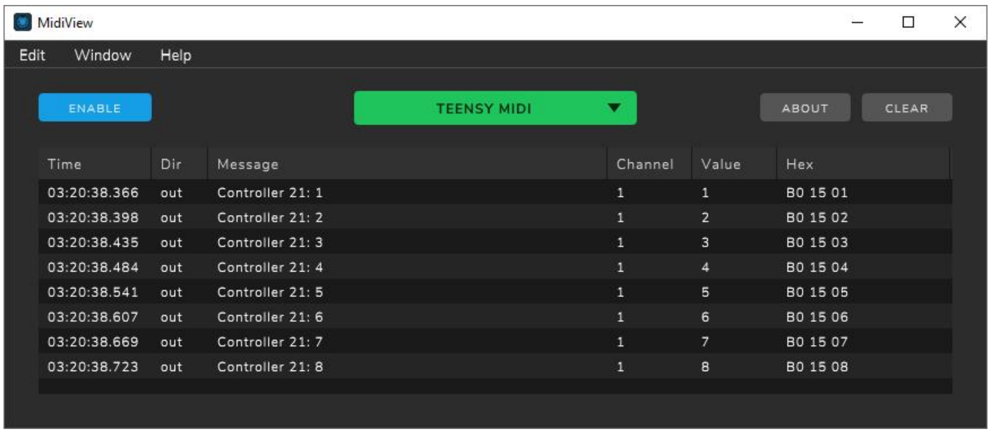
```
...
const int note[D_PINS] = {60, 61, 62, 63};      // MIDI-Note-Values
...
```

The following table shows the MIDI values and the corresponding notes, where I marked the used MIDI values in red.

Octave	Notes											
	C	C#	D	D#	E	F	F#	G	G#	A	A#	H (B)
-2	0	1	2	3	4	5	6	7	8	9	10	11
-1	12	13	14	15	16	17	18	19	20	21	22	23
0	24	25	26	27	28	29	30	31	32	33	34	35
1	26	37	38	39	40	41	42	43	44	45	46	47
2	48	49	50	51	52	53	54	55	56	57	58	59
3	60	61	62	63	64	65	66	67	68	69	70	71
4	72	73	74	75	76	77	78	79	80	81	82	83
5	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107
7	108	109	110	111	112	113	114	115	116	117	118	119
8	120	121	122	123	124	125	126	127				

Table 2 MIDI note values

You can see that these are the notes *C*, *C#*, *D* and *D#* in the 3rd octave, which can also be seen in the MIDI view. Then let's see how the two knobs show up. First I move the left one a little bit to the right, which was in the left stop at the beginning.



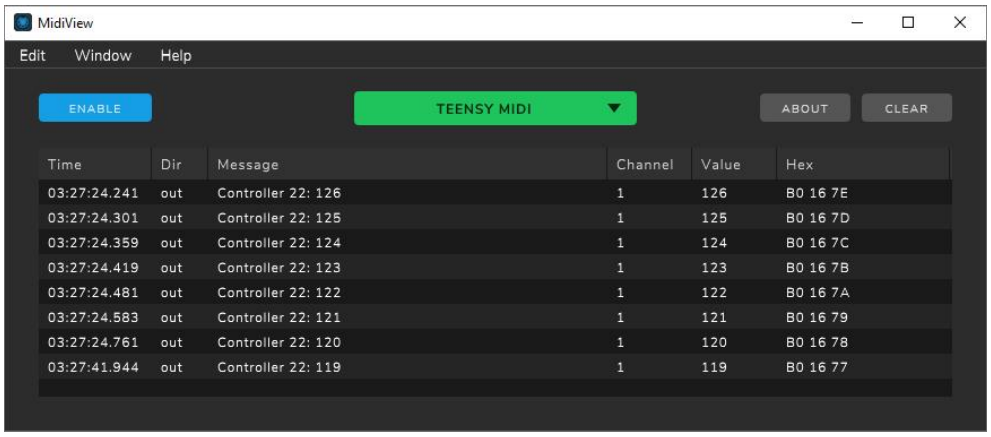
```
Time          Dir  Message           Channel  Value  Hex
03:20:38.366  out  Controller 21: 1  1        1      B0 15 01
03:20:38.398  out  Controller 21: 2  1        2      B0 15 02
03:20:38.435  out  Controller 21: 3  1        3      B0 15 03
03:20:38.484  out  Controller 21: 4  1        4      B0 15 04
03:20:38.541  out  Controller 21: 5  1        5      B0 15 05
03:20:38.607  out  Controller 21: 6  1        6      B0 15 06
03:20:38.669  out  Controller 21: 7  1        7      B0 15 07
03:20:38.723  out  Controller 21: 8  1        8      B0 15 08
```

Figure 17 The MIDI messages of the left knob

The two knobs have been defined with respect to their CC-IDs as follows.

```
...
const int CCID[A_PINS] = {21, 22};          // Control-ID
...
```

And exactly this *ID 21* is to be recognized also in the MIDIView in the column Messages. In addition, of course, the detected value is displayed, which can range from 0 to 127. I will now do this with the right knob, which was in the right stop at the beginning and is now slowly turned to the left.



```
Time          Dir  Message                  Channel  Value  Hex
03:27:24.241  out  Controller 22: 126       1        126    B0 16 7E
03:27:24.301  out  Controller 22: 125       1        125    B0 16 7D
03:27:24.359  out  Controller 22: 124       1        124    B0 16 7C
03:27:24.419  out  Controller 22: 123       1        123    B0 16 7B
03:27:24.481  out  Controller 22: 122       1        122    B0 16 7A
03:27:24.583  out  Controller 22: 121       1        121    B0 16 79
03:27:24.761  out  Controller 22: 120       1        120    B0 16 78
03:27:41.944  out  Controller 22: 119       1        119    B0 16 77
```

Figure 18 The MIDI messages of the right knob

The *ID 22* is displayed in the MIDIView with the corresponding value. So everything runs exactly as it was defined before in the Sketch.

Assign another MIDI identifier

In the program MIDIView you could see that our MIDI controller identifies itself with the name *TEENSY MIDI*. This is a predefined identifier by Sketch, but it can be customized very easily if desired. A corresponding example can be found under the following menu item.

File|Examples|Teensy|USB_MIDI|MIDI_name

The procedure is the following.

Step 1: Add a new tab named name.c to the Sketch.

Step 2: Insert the following code.

```
#include "usb_names.h"

#define MIDI_NAME   {'E','r','i','k','\'','s',' ','M','I','D','I'}
#define MIDI_NAME_LEN  11

// Do not change this part.  This exact format is required by USB.
struct usb_string_descriptor_struct usb_string_product_name = {
        2 + MIDI_NAME_LEN * 2,
        3,
        MIDI_NAME
};
```

The name I want is to be *Erik's MIDI* and was defined via the MIDI_NAME array in the form of single letters. The length of the array has to be adjusted in the next line, of course. So just count the number of letters and store them there as value. After the upload this new name will appear in MIDIView for selection.

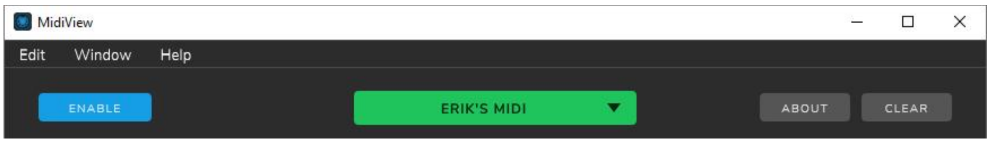


Figure 19 The new MIDI name

A real test in VCV Rack

With so much theory, it is certainly appropriate to test the MIDI controller in a real environment. Of course, the VCV Rack is a good choice again. I have prepared the following patch for this.

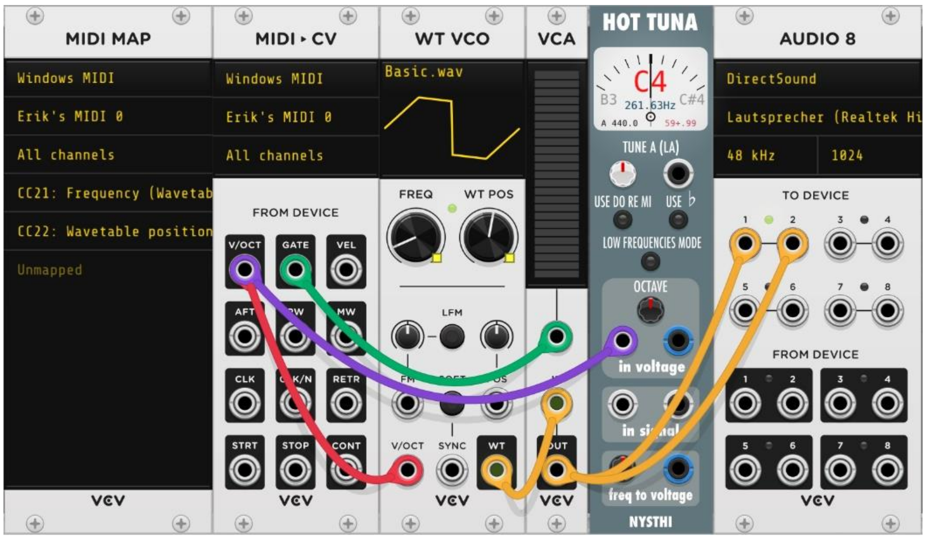


Figure 20 The VCV rack patch for the Teensy MIDI controller

Which VCV Rack modules are necessary for this? Well, this is in any case the left module with the name *MIDI-MAP*. There it comes to an assignment of a MIDI controller control to a VCV-Rack control, which is done by the so-called mapping. I have created a special video for this. Via the two knobs on the MIDI controller it is now possible to influence the two knobs *FREQ* and *WT POS* on the *WT VCO* module.

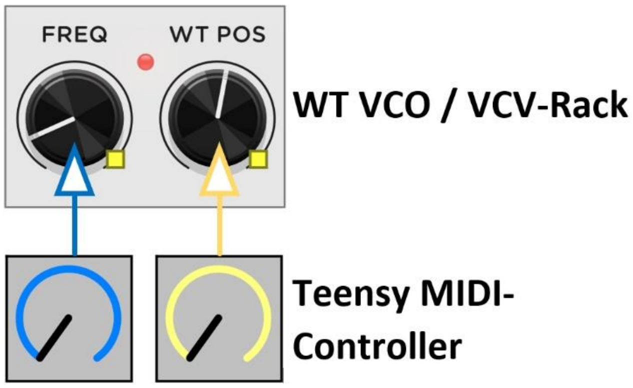


Figure 21 The knobs influence the WT-VCO module

Of course, the four keys can also be used to change the pitch accordingly within very narrow limits. This example serves only as a small introduction to the topic of MIDI controllers and I hope that it helps to achieve an inspiring effect.

More information can be found on my website.

Hyperlinks!

https://erik-bartmann.de/

https://erik-bartmann.de/?Musik___VCV-Rack

Happy Frickeling!

Erik Bartmann