

# Using MIDI in Flickernoise Patches

Werner Almesberger  
werner@almesberger.net

February 27, 2012

## 1 Introduction

This document describes the mechanisms Flickernoise provides for interacting with MIDI controls and explains how patches can make use of this functionality.

Note that this work is still in progress. For example,

- the MIDI device specifications should not have to be part of patches,
- Flickernoise should only consider devices that are really present, and
- we should support other events than just MIDI control message, such as other MIDI message types, DMX, keyboard, IR remote, etc.

## 2 Quick start

While the finer details of MIDI controls can get complicated, the items in the following example are often all that is needed to use many MIDI devices:

```
1 midi "Gizmo"  main = fader(102); aux = pot(103); but1 = button(16);  
but2 = button(17); select = switch(24);
```

```
sensitivity = range(main);
```

```
per_frame : wave_scale = sensitivity * 10; sensitivity = sensitivity * 0.99;
```

In lines 1 through 7 we describe the controls the MIDI device called “Gizmo” provides. In this case, we have one fader, one potentiometer, two buttons, and one switch. We assign them names that represent their role: **main** for the principal fader, **aux** for the potentiometer, and so on.

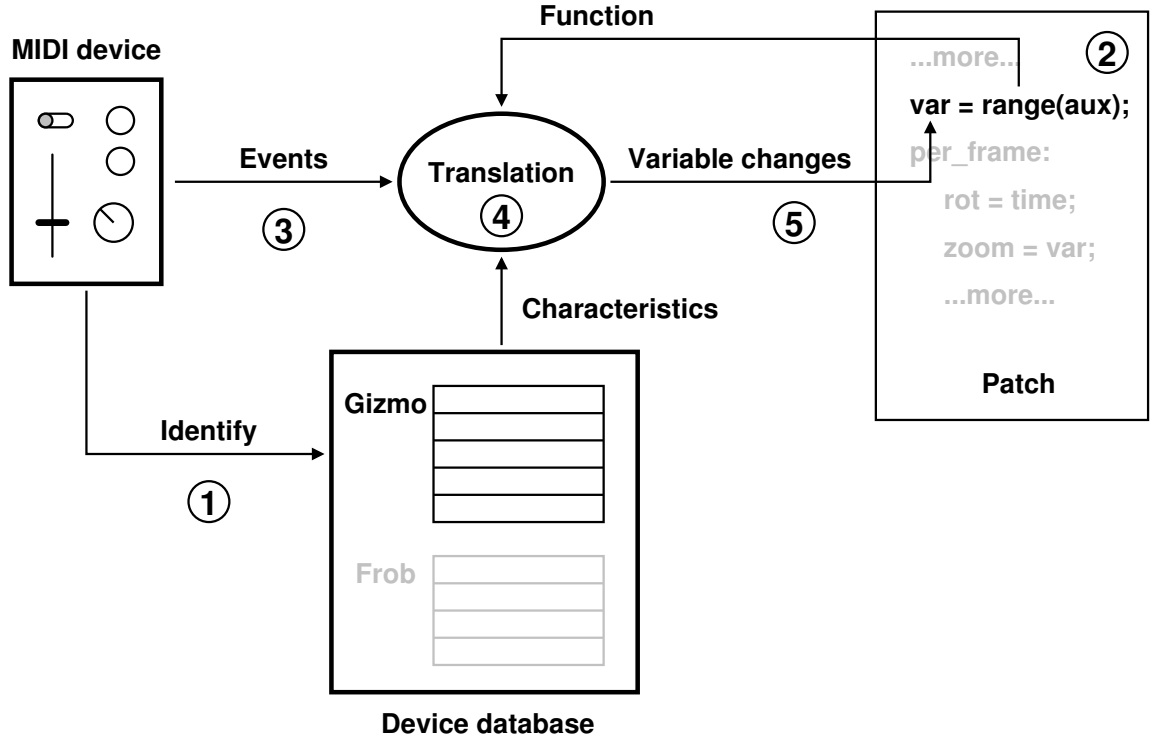


Figure 1: Input event processing in Flickernoise.

The arguments 102, 103, ..., are the numbers of the MIDI controllers. They depend on the device and can be found with the MIDI monitor function of the “MIDI settings” dialog in Flickernoise.

In line 9, we bind the `main` control to a variable. This variable receives the value 0 if the fader is at its minimum and 1 at its maximum. It is then used in per-frame and per-vertex equations. As line 13 shows, one can also change this variable, e.g., to make the sensitivity slowly decay if there is no input from the MIDI device.

### 3 Architecture

Figure 1 shows how MIDI messages are processed in Flickernoise: For each MIDI device, an entry in the device database is selected (1). This entry describes the characteristics of the elements of the MIDI device, e.g., what kind of control elements they are and what addresses they use.

A patch using MIDI devices binds control elements to variables (2). When binding, the patch specifies how it expects the element to behave, e.g.,

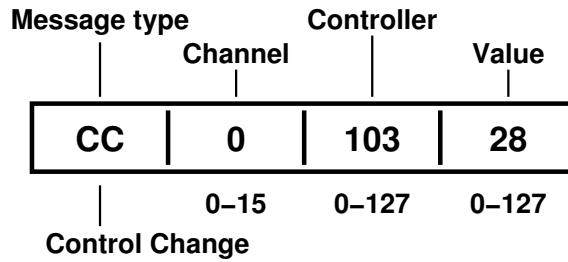


Figure 2: Anatomy of a MIDI Control Change message.

whether it should produce continuous values between 0 and 1 or just 0 when "off" and 1 when "on".

Event messages from the MIDI device (3) identify the element they belong to and carry a numeric value indicating the element's state. This value is translated (4) according to the element's characteristics from the device database combined with the expected function from the patch.

Finally, the result is stored in the control variable (5) where it can then be used by the equations in the patch.

### 3.1 MIDI control messages

A MIDI device sends a message each time one of its elements is actuated. There are various types of elements, keys, pitch wheels, controls, etc. Controls can be all sorts of things, including faders, potentiometers, and push buttons.

For now, we only consider controls. When a control is actuated, the MIDI device generates a MIDI Control Change message of the structure shown in figure 2.

Channel numbers are encoded as values from 0 to 15 in the actual MIDI message but are commonly presented as 1 to 16 to the user. Also Flickernoise follows this convention.

Most MIDI devices use a single channel. The number of this channel can sometimes be set by the user. The controller numbers are typically fixed.

When Flickernoise receives a MIDI message, it uses the message type, the channel number, and the controller number to determine where to send the value.

### 3.2 Control variables

Patches communicate with the outside world through variables. MIDI control input is no exception. Instead of using pre-defined names like it is the case for `time`, `bass`, etc., the names of MIDI control variables can be chosen freely.

Values from MIDI controls are usually translated to the range 0 to 1. This can be as simple a division by 127. Section 4.2 describes more ways to translate MIDI messages.

Updates of control variables are synchronized with patch execution such that updates never happen while the patch is running.

## 4 Using controls in patches

The following sections describe the syntax and semantics of the language constructs that give access to MIDI controls.

### 4.1 Device database

The device database tells Flickernoise how to identify MIDI devices and their elements and how the elements behave. It also assigns names to the elements that are then used to refer to them in patches.

Device specifications are added to the device database with a MIDI device entry that looks as follows:

```
midi identification { element ... }
```

Each element in the device entry has the following syntax:

```
name = device_type([channel,] control_number);
```

*device\_type* provides a broad characterization of the control element and can be `fader`, `pot`, `differential`, `button`, or `switch`. Control elements are discussed in detail in section 5.

*channel* is the MIDI channel number, from 1 to 16. If the channel is omitted, the element will match any channel.

*control\_number* is a number from 0 to 127 the MIDI device uses to identify the control element.

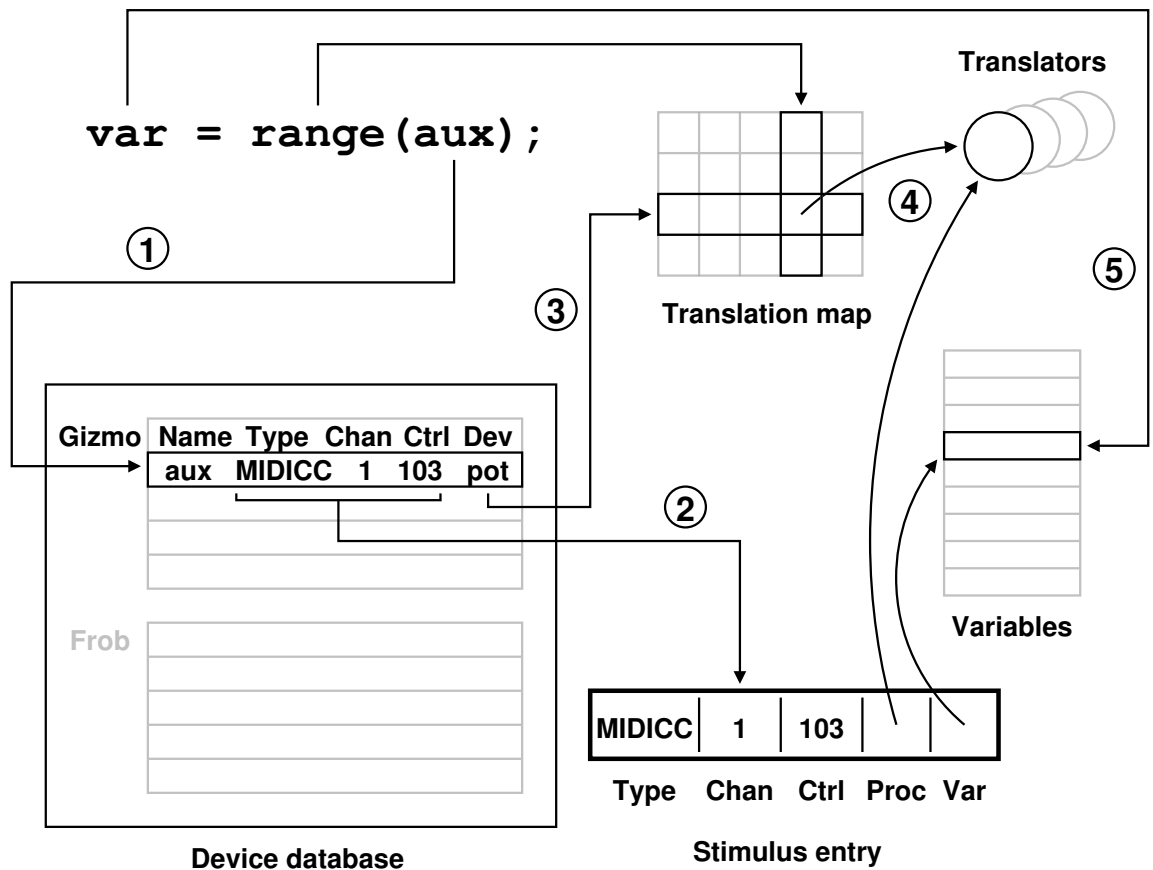


Figure 3: When binding a control variable, information from the device database, a translation map, and the table of patch variables is combined into a stimulus entry.

Below is the example from the quick start section with channel numbers added. We will use the **aux** element as the basis for further examples.

```
1 midi "Gizmo" main = fader(1, 102); aux = pot(1, 103); but1 = button(1, 16); but2 = button(1, 17); select = switch(1, 24);
```

## 4.2 Binding

In order to use a control in a patch, we have to establish a connection between the control element (in the device database) and a patch variable. We call such a variable a *control variable*.

Control variables are bound with a construct that looks like a variable assignment:

`control_variable = function(element_name);`

*control\_variable* can be a pre-defined per-frame or per-vertex variable or it can be a user-defined variable. Variables that are updated by Flickernoise itself cannot be used as control variables. These variables are **bass**, **bass\_att**, **frame**, **idmxn**, **mid**, **mid\_att**, **midin**, **oscn**, **time**, **treb**, and **treb\_att**.

*function* describes how the patch expects the control to behave. Flickernoise then tries to adapt the behaviour of the actual device to what the patch expects. The following functions are available:

**range** The control variable has a value between 0 and 1, depending on the setting of the device. This is commonly used for faders and potentiometers.

**unbounded**, **cyclic** These are special functions used with some rotary encoders. They are described in detail in section 5.7. With other control elements, they behave just like **range**.

**button** The control variable receives the value 1 when the button is pressed and returns to zero when it is released. When applied to elements that send values between 0 and 1, the value is rounded.

**switch** The control variable can be set to 0 or 1 and retains the value until the element is actuated again.

*element\_name* is the name of the control element, as in the device database.

The result of binding is a stimulus entry that tells Flickernoise how to interpret incoming messages. Figure 3 illustrates the steps in creating the stimulus entry: Flickernoise selects a control element with the desired name from the available devices (1). The information needed to identify messages from this control element is copied to the stimulus entry (2). Using the device type from the element record (3) and the function from the binding instruction, a suitable translator is selected from the translation map. Also this is recorded in the stimulus entry (4). Finally, the variable is looked up or added to the patch (5) and a reference to it is placed in the stimulus entry.

### 4.3 Event processing

Figure 4 illustrates how events are translated into changes of control variables.

Each time a MIDI Control Change message arrives, Flickernoise looks for a stimulus that matches the message type, i.e., MIDI control, the channel

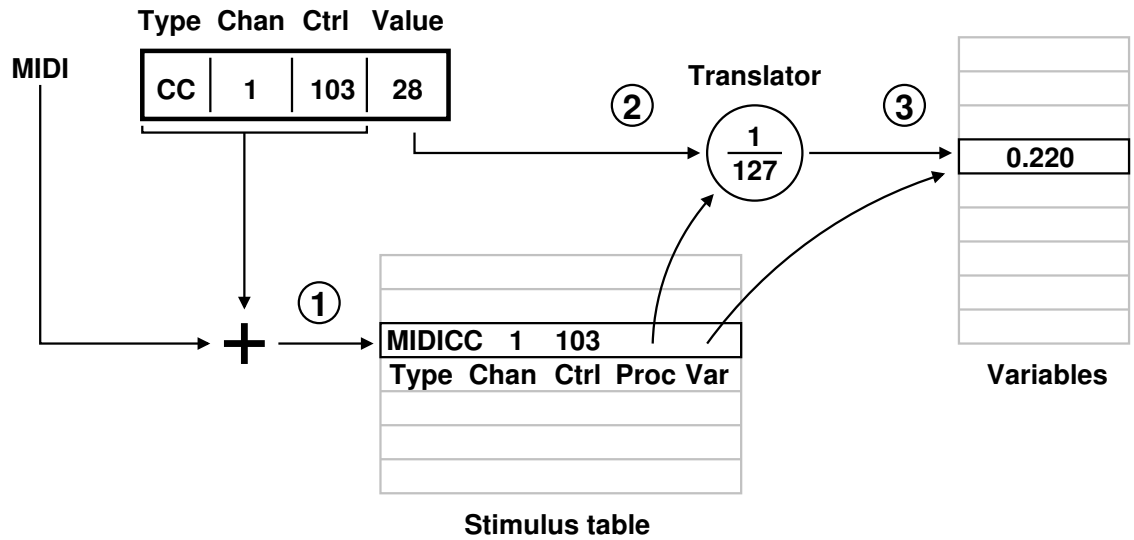


Figure 4: When a MIDI control message is received, Flickernoise looks for a matching stimulus and processes the value accordingly.

number and the controller number (1). If no matching entry exists, the message is ignored.

The stimulus tells Flickernoise which translator to use and where to store the result. In our example, we have a **pot to range** translation (2), which is simply a division by 127. The result,  $28127 = 0.220$ , is stored in the location of the variable **var** (3).

#### 4.4 Writing to control variables

A patch can overwrite the values of control variables. In the case of user-defined per-frame variables, they retain their modified content across frames until a new event arrives. The new event will always overwrite the previous value of the variable without regard for any changes that may have been made.

Changes to per-vertex variables are lost at the end of the program and so are changes to per-frame variables that are not user-defined.

It is sometimes desirable to make a control change a variable relative to a value set by the patch. The following example illustrates how this can be accomplished:

```
1 last = 0; cvar = range(foo);
```

```
perframe : var = var + cvar - last; last = cvar; / * ...use "var" ... * / var =
condition?new_value : var;
```

Instead of writing to the control variable **cvar** directly, we propagate any relative changes of **cvar** to **var** in lines 5 and 6, and only modify **var** in line 8.

Note that **var** may assume values outside the range  $0 \dots 1$ . If this is not desirable, it can be clipped after line 5 with

```
1 var = max(min(var, 1), 0);
```

## 4.5 Multiple bindings

So far, we have only seen one element being bound to one control variable at a time. A control variable can also be bound to multiple elements and the same element can be bound to multiple control variables.

Binding the same element to different variables can be useful in cases where the translation differs as well. For example, the following code snippet would control two parameters with the same element:

```
1 growth = range(main); tilt = cyclic(main);

per_vertex : zoom = growth * 0.2; angle = 2 * 3.14159 * tilt; wave_x = cx +
0.1 * cos(angle); wave_y = cy - 0.1 * sin(angle);
```

A situation that may be even more common is to have multiple elements that change the same variable, e.g., when using multiple input devices. Example:

```
1 midi "foo"  foo_pot = pot(12);

midi "bar"  bar_pot = pot(34);

sensitivity = range(foo_pot); sensitivity = range(bar_pot);

perframe : wave_scale = sensitivity * 20;
```

## 5 Control elements

In the sections below, we describe the various control elements, how they are described in the device database, and their behaviour. We give examples that show the physical state of the element, the value a MIDI device may typically send for the element in that state, and then the resulting values for the various translations.

Since **range**, **unbounded**, and **cycle** only differ from each other in one case, they are usually abbreviated to just “**range**, ...”.



## 5.1 Faders






Faders are slide potentiometers that normally cover the whole range of values a MIDI controller can send: 0–127. They retain their position when released.

Fader elements of a device are declared with `fader()` and are typically bound with `range()`:

```
1 midi ... name = fader(...);
```

```
var = range(name);
```

The following example shows how Flickernoise maps faders to control variables. We start with the fader in the 0% position, move it to the 40%, then 60%, and the 100% position. At the end, we return it to the 0% position.

User input					
MIDI value		51	76	127	0
Translation	0	0.4	0.6	1	0
	0	0	1	1	0
					<code>range, ...</code>
					<code>button, switch</code>

The mapping is quite straightforward: `range`, `unbounded`, and `cycle` produce a value from 0 to 1 corresponding to the position of the knob. `button` and `switch` produce 0 if the knob is in the lower half of the range, 1 if it is in the upper half.

## 5.2 Rotary potentiometers

Rotary potentiometers work exactly like faders except that they are declared with `pot()`:<sup>1</sup>

```
1 midi ... name = pot(...);
```





```
var = range(name);
```

They have mechanical stops at the beginning and at the end of their range, which distinguishes them from the rotary encoders described in the next section.

The example below shows a potentiometer that travels over an angle of 270°:

---

<sup>1</sup>The current implementation does not distinguish at all between `fader` and `pot`, but they may be represented with different symbols in a future GUI.

User input					
MIDI value		42	85	127	
Translation	0	0.33	0.67	1	range, ...
	0	0	1	1	button, switch


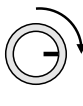
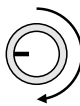


### 5.3 Rotary encoders acting as potentiometers

Rotary encoders look similar to potentiometers but differ from them by not having a mechanical stop. This means that they can be turned indefinitely in the same direction.

MIDI devices usually emulate the behaviour of potentiometers by ignoring any turns at the end of the value range. When the direction is reversed, the values change immediately.

Rotary encoders acting as potentiometers are also declared with **pot** and bound with **range**.

The example below shows how a rotary encoder covering the full value range in one  $360^\circ$  turn behaves when it is first turned  $450^\circ$  clockwise and then  $45^\circ$  counterclockwise:

User input						
MIDI value		32	96	127	121	
Translation	0	0.25	0.75	1	0.875	<b>range, ...</b>
	0	0	1	1	1	<b>button, switch</b>

The encoder stops at MIDI value 127 at the third turn and the remaining  $90^\circ$  are ignored.

Note that the the above is a bit simplified. Rotary encoders commonly found in MIDI devices only have 20–30 positions per full turn. When turning them slowly, it therefore takes several full turns to cross the entire range. To permit quick changes, MIDI controllers usually change the value in steps larger than one when the encoder is turned rapidly.

## 5.4 Push buttons

Push buttons are activated by pressing them and they return to the inactive state when released. Buttons can only be either fully on or fully off, without intermediate values.






Push buttons are declared with `button()` and bound with `button()` or `switch()`:

```
1 midi ... name = button(...);
```

```
var = button(name);
```

`switch()` turns the control on when the button is pressed the first time and then off again when pressed a second time.

Example:

User input						
MIDI value		127	0	127	0	
Translation	0	1	0	1	0	range, ..., button
	0	1	1	0	0	switch




## 5.5 Switches

A switch is set to either on or off and retains its state until actuated again. Switches are declared with `switch()` and typically bound with `switch()` as well:

```
1 midi ... name = switch(...);
```

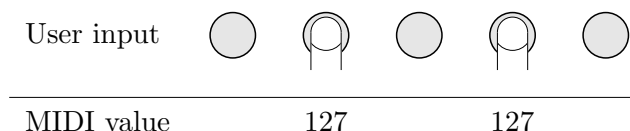
```
var = switch(name);
```

Switches behave the same for all functions:

User input				
	ON OFF	ON OFF	ON OFF	
MIDI value		127	0	
Translation	0	1	0	all

## 5.6 One-way buttons

Some devices have buttons that send a MIDI event only when pressed but not when released. E.g.,



Since the control variables in Flickernoise reflect the state of a control and not events, we cannot directly use such devices. However, by exploiting the ability to overwrite a control variable, we can simulate a release as follows:

```
1 midi ... name = button(...);
var = button(name);
perframe : / * use "var" * /var = 0;
```

When the button is pressed, it will set **var** to 1. At the next frame, this value can be used. At the end of the frame (and before running per-vertex equations), it is reset to zero. **var** thus acts like a **button** control variable where the button is always pressed for exactly one frame duration.

## 5.7 Differential encoders

The rotary encoders of some controllers send differential values instead of the usual absolute values. Differential values allow Flickernoise not only to translate them to absolute values, but also enable more sophisticated translations.

Differential values are encoded as signed 7 bit numbers. This means that values in the range 0–63 represent an increase by that amount while values in the range 64–127 represent a decrease by 128 minus the value. E.g., a value of 127 would mean a decrease by one.






Differential encoders are declared with **differential()**:

```
1 midi ... name = differential(...);
```



There are three distinct options for binding them to continuous values: with **range()**, the emulate potentiometers with a range from 0 to 1. With **unbounded**, the control variable can also assume values below 0 or above 1. Finally, **cyclic** limits the control variable to a range from 0 to 1 but makes it wrap from 1 to 0 when increasing, and from 0 to 1 when decreasing.

1 var1 = range(name1); var2 = unbounded(name2); var3 = cyclic(name3);






This example illustrates the values that result for the different translations when turning an encoder that needs one  $360^\circ$  turn for the full value range by a total of  $405^\circ$ :

User input						
MIDI value		32	32	63	16	
Translation	0	0.25	0.5	1	1	range
	0	0.25	0.5	1	1.125	unbounded
	0	0.25	0.5	1	0.125	cyclic





Continuing the example above, we turn the controller now by  $90^\circ$  counter-clockwise:

User input			
MIDI value	96		
Translation	1	0.75	range
	1.125	0.875	unbounded
	0.125	0.875	cyclic

Starting from zero again, we now turn the controller first  $45^\circ$  clockwise and then  $450^\circ$  counterclockwise:

User input						
MIDI value		16	96	64	64	
Translation	0	0.125	0	0	0	range
	0	0.125	-0.125	-0.625	-1.125	unbounded
	0	0.125	0.875	0.375	0.875	cyclic

When emulation a button or a switch, any clockwise turn will set the control variable to 1 while any counterclockwise turn will set it to zero:

User input				
MIDI value		32	112	96
Translation	0	1	0	0

**button, switch**

Like the rotary encoders emulating potentiometers described in section 5.3, the values a device sends for differential encoders may be increased if turning rapidly. Furthermore, instead of the very large values used in these examples, one would see a sequence of smaller values as the encoder is turned.