Proceedings of the 17th Linux Audio Conference (LAC-19), CCRMA, Stanford University, USA, March 23–26, 2019

MIDIZAP: CONTROLLING MULTIMEDIA APPLICATIONS WITH MIDI

Albert Gräf

IKM, Music-Informatics
Johannes Gutenberg University (JGU) Mainz, Germany
aggraef@gmail.com

ABSTRACT

The paper introduces midizap, a new Linux utility to interface MIDI controllers with multimedia applications such as audio and video editors or computer music programs. midizap is a heavily modified version of Eric Messick's ShuttlePRO program. Its purpose is to translate MIDI controller input to commands (either MIDI or X11 keyboard and mouse events) which the application understands. Configurations are simple text files, no programming skills are required. There's also an Emacs mode to help creating and testing these configurations. Jack session and MIDI patchbay functionality is available as well, making it easy to manage separate midizap instances for different controllers and applications.

1. INTRODUCTION

These days, MIDI controllers are typically USB class devices which can be connected to a Linux computer without requiring any special hardware or drivers. Also, they're often much cheaper than specialized gear for specific uses such as photo and video editing. So wouldn't it be nice if we could just use whatever MIDI controller we have for controlling our favorite multimedia applications? The problem is, while DAW and DJ programs typically have extensive and customizable MIDI interfaces built into them, other applications may not offer any MIDI support at all, or only recognize a particular set of MIDI messages. Thus we often have to translate the MIDI input from the controller to whatever keyboard or MIDI commands the application understands, and we'd like to be able to do this without having to modify the target application.

I was surprised to find that on Linux apparently there's no simple and practical solution for this problem yet. There is the Ctlra and Mappa software from the OpenAV project [1], but it is still under development and only readily supports a handful of devices and applications right now, which means that adding a new controller or application likely requires a fair amount of C programming. A popular commercial program in this realm is the Bome MIDI translator [2], but it's only available for Mac and Windows.

Another interesting utility is Eric Messick's ShuttlePRO program [3] which targets the Contour Design "Shuttle" devices [4] designed for video editing. These devices don't speak MIDI, but Messick's program is free (GPL) software, works on Linux, and includes the necessary code to recognize applications by their window name and translate device input to X11 keyboard and mouse events. Adding Jack MIDI support to it seemed to be a piece of cake, so that's what I set out to do. The first result of this side project was a fork of the ShuttlePRO program which improves the original program in some ways and adds Jack MIDI output [5]. The next obvious step then was to replace the Shuttle input with Jack MIDI input, giving birth to the midizap program as it stands now [6].

In the following sections, we discuss midizap's most important features and some typical uses. For lack of space, this description is necessarily somewhat terse and incomplete, but should give the interested reader an idea of what capabilities the program offers and when you might want to use it. More details can be found on the Github project page or in midizap's extensive manual.

2. TRANSLATION SYNTAX

As with the ShuttlePRO program, midizap's configuration is a simple text file which is divided into sections for different applications. A sample configuration is provided in /etc/midizaprc, you can copy this to create a .midizaprc file in your home directory and edit it there as needed. You can also run midizap with any other configuration file by specifying the name of the file on the command line. A collection of configurations for various purposes (mostly Mackie emulations for different devices) can be found in the examples folder in the sources.

The configuration language is line-oriented, each line is either a *section header* or a *translation rule*. The hash sign # at the beginning of a line or after whitespace starts a comment. Each section starts with a header of the following form, specifying a section name and a regular expression pattern:

[*name*] *pattern*

The section name is only used in diagnostic messages and can essentially be chosen freely. It is the regular expression pattern which actually determines whether the translations in the section are active at any given time. To these ends, midizap matches the pattern against the `WM_CLASS` and `WM_NAME` properties of the currently selected X application window. The latter is what is actually visible in the window title, while the former is an internal property which identifies the type of application window.[1] The regular expression pattern can also be omitted, in which case the translations will always be active. Such "default" sections are to be placed near the end of the file, and their translation rules will be used as fallback translations when none of the other translation sections in the configuration match the selected application window.

The section header is followed by a list of (zero or more) *translation rules* describing the translations which should be active for the given application. These just list MIDI messages and their translations in a human-readable symbolic format. Each translation rule must be on a line by itself and consists of a single left-hand side symbol denoting the MIDI message to be translated, followed by the right-hand side which is a list of zero or more symbols specifying the MIDI messages and/or keyboard and mouse commands to be output. It thus takes the following general form:

input output$_1$ *output*$_2$ ...

[1] You can find out about the `WM_CLASS` and `WM_NAME` properties of a window with the xprop program, or invoke midizap with the `-dr` debugging option to have it print this information. midizap will try to match both by default, but you can tell it explicitly to only match class or title by prefixing the pattern with the `CLASS` or `TITLE` token, respectively.

Here is a simple example:

```
[Terminal] CLASS ^(.*-terminal.*|konsole|xterm)$
F5    XK_Up
F#5   "pwd"
G5    XK_Down
G#5   "ls"
A5    XK_Return
```

This defines a list of translations for some common types of terminal windows, as specified in the section header on the first line. The input messages are listed on the left and the corresponding keyboard output on the right. Here we map a few notes in the middle octave to the cursor up and down and return keys, as well as some frequently used shell commands. The bindings above will let you operate the shell from your MIDI keyboard when the keyboard focus is on a terminal window. To make this work, you'll first have to connect your MIDI controller to midizap's MIDI input port, e.g., using a Jack MIDI patchbay program like QjackCtl. You then click on the desired terminal window and start entering notes on the MIDI keyboard to have the corresponding commands sent to the selected window.

It is important to note here that, like the ShuttlePRO program, midizap will only ever send keyboard and mouse commands to the *currently selected window* or, more precisely, the window which has the keyboard focus. The selected window also determines which section of translation rules is currently active. Thus you have to make sure that you first click on the right application window before you can go on sending keyboard and mouse commands to it. (In contrast, MIDI commands can be sent to any application as long as it is connected to midizap's MIDI output, see below.)

Let's now have a closer look at the syntax of translation rules. The precise syntax is a bit intricate, so we have to refer the reader to the EBNF grammar in Appendix A for details. But we will try to at least sketch out the most important elements in what follows. The first token of a translation rule (the left-hand side) denotes the MIDI message to be translated, which is followed by an output sequence (the right-hand side) consisting of MIDI messages or X key and mouse events. There can be any number of these, and you can freely mix MIDI messages and X events on the output side.

The `XK` symbols indicate X key codes and must be denoted exactly as they appear in the /usr/include/X11/keysymdef.h file. A string enclosed in double quotes is simply a shorthand for a sequence of X key events.[2] Besides the key codes from the header file, there are also some special tokens to denote mouse button and scroll wheel events (`XK_Button_1`, `XK_Scroll_Up`, etc.).

MIDI note messages are denoted in a symbolic format that will be familiar to musicians: a note letter (`A` to `G`) is followed by an optional accidental (`#` or `b`) and an octave number. By default, `C5` denotes middle C, but the octave numbering can be changed with a directive in the configuration file. Other kinds of (non-system) MIDI messages are denoted using short mnemonics: `KP:`*note* (aftertouch a.k.a. key pressure for the given note); `CC`*n* (control change for the given controller number); `PC`*n* (program change for the given program number); `CP` (channel pressure); and `PB` (pitch bend). These can all be followed by a dash and the MIDI channel (the default MIDI channel being 1).

In the example above, all note messages are interpreted as key events, having an "on" and "off" status: the key goes "down" when a note-on message is received, and goes "up" again when the corresponding note-off message (or a note-on with zero velocity) arrives. We also call this a *key translation*. These work in the same way as in the ShuttlePRO program; e.g., in the above example, the `XK_Up` key is pressed when the note-on for `F5` is received, and won't be released until the corresponding note-off is detected. If there's more than one key in the output sequence, as with the double-quoted strings in the example, each key will normally be released before the next one is pressed, and only the last key in the sequence will be held until the note-off is received. There are also some special suffixes for key specifications (`/D`, `/U`, `/H`) which indicate keys to be held and released explicitly or at the end of the sequence; we refer the reader to the documentation for details.

As another, more practical example, here are some bindings for the Kdenlive and Shotcut video editors mapping some keys and the big jog wheel on a Mackie-compatible device to some common video editing functions:

```
[Kdenlive/Shotcut] CLASS ^(shotcut|kdenlive)$

# playback controls
A#7 XK_space    # Play/Pause
A7  "K"         # Stop
G7  "J"         # Rewind
G#7 "L"         # Forward

# replace/drop (sets in and out points)
D#7 "I"         # Set In
E7  "O"         # Set Out

# left/right cursor movement
D8  XK_Home     # Beginning
D#8 XK_End      # End

# the jog wheel moves left/right by single frames
CC60< XK_Left   # Frame reverse
CC60> XK_Right  # Frame forward
```

The last two rules for the jog wheel show an example of a *data translation* which translates incremental changes in the extra data byte of a message to corresponding X key presses. For ordinary (absolute) control changes these take the form `CC`*n*`-` and `CC`*n*`+`, where *n* denotes the controller number, and the `-` or `+` flag the direction of the change. However, here we employed the special `<` and `>` suffixes which indicate a *relative* change in "sign-bit" encoding [7], which is commonly used with encoders (knobs or wheels which can be turned endlessly in either direction). In either case, the up or down output sequence is emitted for each unit change in the parameter. You can also scale these responses by adding suitable step sizes on the left-hand or right-hand side of the translation rules; again we refer the reader to the documentation for details.

The rules we've seen so far all translate MIDI to X key events. midizap can also work as a MIDI mapper which translates MIDI input to MIDI output. This is useful if the target application supports MIDI, but needs the controller input to be remapped to MIDI commands it understands. The following example lets you play a little drumkit on a General MIDI (GM) synthesizer like Fluidsynth, by remapping some of the white keys in the 4th octave to a few drum notes on MIDI channel 10 (the GM drum channel). We also threw in a rule to remap the modulation wheel (`CC1`) to the volume controller on MIDI channel 10 (`CC7-10`).[3]

[2] In the current implementation, this only works with printable ASCII characters which can be mapped 1-1 to X11 key codes. Otherwise explicit key codes must be used.

[3] The notation `CC1=` being used here provides a shorthand for two data

```
[MIDI]
C4    C3-10
D4    C#3-10
E4    D3-10
F4    D#3-10
CC1=  CC7-10
```

Note that we placed the MIDI translations into a special `[MIDI]` section here. This is a default section reserved for applications accepting MIDI input. To make this work, you will have to invoke the midizap program with the `-o` option. This enables the `[MIDI]` section and equips midizap with an additional MIDI output port which can be connected to the target application (like Fluidsynth in this example). As long as your translations only output MIDI messages, you then don't have to worry about keyboard focus, as the application will receive all data from midizap through the MIDI connection (in fact the application does not need to have any X window at all in this case).

The above example does a simple 1-1 mapping of MIDI events, but in general the output sequence may consist of as many MIDI messages of as many different types as needed, and you can also mix MIDI and X keyboard and mouse output if you want. An interesting use case for MIDI translations is Mackie emulation which we'll discuss in Section 4.

3. GETTING STARTED

Before we explore some of midizap's more advanced features, let us quickly go over the mundane technicalities of using midizap.[4] midizap is a command line application, so you typically run it from the terminal. However, it is also possible to launch it from your Jack session manager (see Section 5 below) or from your desktop environment's startup files once you've set up everything to your liking. In addition, for Emacs users there's a midizap mode which makes it very easy to edit and test your midizap configurations. It does syntax highlighting, auto-completion of keywords, and also lets you launch midizap in an Emacs buffer; please check the midizap-mode.el file in the sources for details.

midizap uses Jack for its MIDI input and output, so you'll need to be familiar with Jack. We recommend using a Jack front-end like QjackCtl which makes setting up Jack and doing MIDI connections much easier. You'll also need an ALSA-Jack MIDI bridge in order to expose the ALSA sequencer ports as Jack MIDI ports, so that the MIDI inputs and outputs of your controller and other non-Jack MIDI applications can be connected to midizap. Jack's built-in bridge will work for this purpose (in the QjackCtl setup, select `seq` as the MIDI driver), or you can use Nedko Arnaudov's a2jmidid utility [8]. The latter is easier to use with Jack2, and will work with Jack1 as well.

Running just `midizap` without any arguments launches midizap with the default configuration and a single Jack MIDI input port which you'll have to connect to your MIDI controller. To utilize MIDI output, run `midizap -o`; as already mentioned, this equips midizap with an additional Jack MIDI output port to be connected to the MIDI application you wish to control. You can also run midizap with any other configuration file by simply specifying the name of the file on the command line. There are a number of other options and configuration file directives which let you set the Jack client name, number of input and output ports and the desired MIDI connections; see Section 5.

Moreover, midizap offers a fair amount of debugging options which will be very helpful when you start developing your own configurations. A good set of options to start with is `-drkm`; `r` prints the class names and titles of selected windows which is useful to determine which regular expressions to use in the section headers; `k` prints out recognized translations so that you can check that midizap is actually picking the right translation rules for some given MIDI input; and `m` activates midizap's built-in MIDI monitor which prints out recognizable MIDI input in the same syntax that's used in the configuration file, which makes it easy to figure out which MIDI messages you may want to create translations for.

The default configuration is really just an example, to help you get started. You can either edit that file or create your own configuration. To start from a clean slate, create an empty file in a text editor, say myconfig.midizaprc, and invoke midizap on it. The file will be reloaded whenever you save it, so you can just keep on adding translation sections and rules and try them out immediately, without having to restart the program. If you're an Emacs user, you will find midizap's Emacs mode most convenient to do all this.

Let's walk through a simple example to show how this works. We'll use the Shotcut video editor (`https://www.shotcut.org/`) for illustration, so let's assume that you've already launched Shotcut and loaded a video file in it. Next, make sure that Jack is running, create the myconfig.midizaprc file, run `midizap -drkm myconfig.midizaprc`, and connect your controller to midizap's MIDI input. With the Shotcut window selected, wiggle one of the controls on your MIDI gear; I'll take the modulation wheel as an example. In midizap's output you should now see something like:

```
Loading configuration: myconfig.midizaprc
[0] CC1-1 value = 40
no translation found for Untitled - Shotcut
(class shotcut)
```

This tells you the class name (`shotcut`) of the application window, as well as the name of the incoming MIDI message (`CC1-1`, which can also be abbreviated as `CC1` in the configuration, as 1 is the default MIDI channel). Having identified the application and the MIDI message we'd like to translate, we can now edit our configuration in the myconfig.midizaprc file accordingly. Let's add the following section header and translations, and save the file:

```
[Shotcut] CLASS ^shotcut$
CC1- XK_Left
CC1+ XK_Right
```

midizap should automatically reload the file. Moving the modulation wheel again (with the Shotcut window still selected) will now change the playback position in Shotcut, while the translations we just added are printed by midizap.

4. ADVANCED USES

One particularly interesting use case for MIDI translations is the emulation of Mackie controllers. The *Mackie control protocol* (MCP) has become a de facto standard for DAW programs, because it allows the various track parameters to be mapped without requiring any manual setup.[5] Also, many Mackie-compatible devices offer *feed-*

translation rules `CC1-` and `CC1+` with the same right-hand side `CC7-10`.

[4] We don't discuss installation here, which is very easy and, besides the X libraries, only needs very few dependencies which should be readily available on all Linux distributions; details can be found in the README file.

[5] Although MCP is widely used, there doesn't seem to be a publicly accessible specification of the protocol anywhere. A partial description can be found at `http://www.jjlee.com/qlab/MackieControlMIDIMap.pdf`.

back, i.e., the ability to display current parameter values and other kinds of status information using LEDs, motor faders, scribble strips and the like, which makes them very convenient to use.

Some MIDI controllers have a built-in MCP mode, but many don't. Thus it is tempting to employ midizap to emulate this mode. Even if a device already offers MCP, it may be lacking some features; this is true especially for some of the cheaper and/or smaller devices like the Behringer X-Touch Mini. In such cases midizap may be used to beef up the device's capabilities and/or modify its bindings so that they better suit your workflow.

Emulating MCP usually requires remapping some or all of the MIDI messages of the device, on both input and output (if the device offers some feedback capabilities). Especially the feedback part often poses some challenges. The purpose of this section is to dive into some of midizap's more advanced features catering to these use cases, using MCP emulation as a running example. Of course, these features may also be helpful in other situations calling for complicated translations.

4.1. Shift State

One issue we often face right away when designing a Mackie emulation is the number of available controls. For instance, your device might only provide you with 8 faders which must then be used to emulate both the volume and the panning controls of a Mackie controller. Or it may not have enough buttons for all the special MCP functions that you need. In such cases it is useful to designate a special shift key on the device which lets you switch between different functions of the available controls.

midizap provides a special `SHIFT` token for this purpose which can be used anywhere on the right-hand side of a translation. This token doesn't produce any output, it merely toggles an internal bit indicating the current shift status. This is often used in a key translation as follows:

```
D8 SHIFT
```

Now, midizap will go into shift mode whenever the device generates the note `D8` (which happens to be the shift key on an AKAI APCmini device, cf. Fig. 1(8); but any available button-like control will do). Pressing the `D8` key again disables shift mode. Thus the above rule implements a "CapsLock"-style shift button. You can also do an ordinary shift button as follows:

```
D8 SHIFT RELEASE SHIFT
```

Here, the `RELEASE` token indicates an explicit release sequence which will be invoked as soon as the `D8` key is released (i.e., the corresponding note-off is received). Hence pressing this key now toggles on the shift status, and releasing it immediately toggles it off again, just like an ordinary shift key on a computer keyboard.

Having defined the shift key, we can now use its current status in other translations. The `^` character, when used as a prefix on the left-hand side of a translation, tells midizap that the translation should only be valid in shifted state. Thus we can now have two different rules associated with each incoming MIDI message, depending on the current shift status, effectively giving us about twice as many controls as we had before.

Let's take the AKAI APCmini as an example again. We can map the first eight faders `CC48` to `CC55` on this device, cf. Fig. 1(4), to the MCP encoders `CC16` to `CC23` in shifted mode as follows:[6]

[6]Note that the MCP encoders use relative values in sign-bit encoding; the `~` suffix on the output `CC` messages indicates that these messages should be converted to that special encoding.

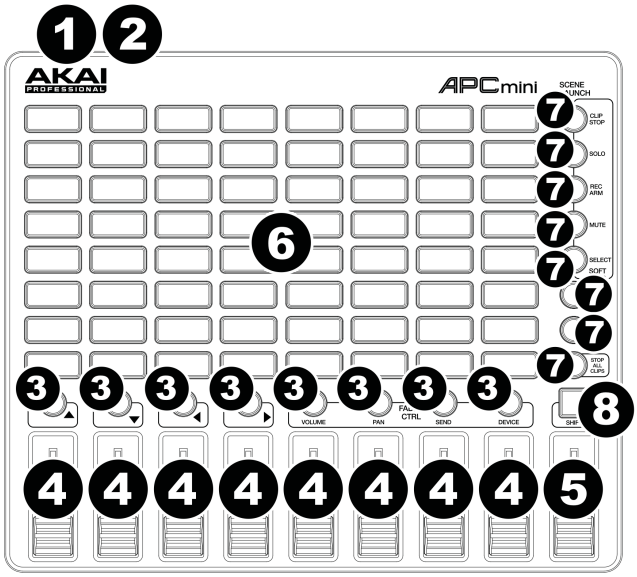

Figure 1: AKAI APCmini [9, p. 5].

```
^CC48= CC16~
^CC49= CC17~
...
^CC55= CC23~
```

The above translations will only be executed in shifted mode (i.e., by holding the designated shift key while operating the faders). In unshifted mode, the faders are still available to be mapped, e.g., to the MCP volume controls (`PB-1` to `PB-8`). For instance:[7]

```
CC48= PB[128]-1
CC49= PB[128]-2
...
CC55= PB[128]-8
```

You will find very similar rules in the APCmini.midizaprc example distributed with midizap. We've only sketched out the use of a single shift key here, but midizap actually supports up to four different shift states, which are denoted `SHIFT1` to `SHIFT4`, with the corresponding prefixes being `1^` to `4^`. The `SHIFT` token and `^` prefix we've seen above are in fact just shortcuts for `SHIFT1` and `1^`, respectively. Thus midizap lets you have up to five different "layers" of MIDI assignments (1 unshifted and 4 shifted states), which will hopefully be enough for most purposes.

4.2. Feedback

Some MIDI controllers have motor faders, LEDs, etc., requiring feedback from the application. To accommodate these, you can use the `-o2` option of midizap (or the `JACK_PORTS 2` directive in the midizaprc file, cf. Section 5), to create a second pair of MIDI input and output ports. Use of this option also activates a second MIDI default section in the midizaprc file, labeled `[MIDI2]`, which

[7]The `[128]` suffix on the `PB` output messages denotes a scale factor here, which scales up the 7 bit `CC` range to the 14 bit range of a pitch bend.

is used exclusively for translating MIDI input from the second input port and sending the resulting MIDI output to the second output port. The control output from the application is then connected to midizap's second input port, and midizap's second output port to the input of the controller, so that the feedback from the application passes through midizap on its way back to the controller.

If all this has been set up properly, MIDI feedback will eliminate most problems with controls being out of sync with the application. midizap has some built-in logic to help with this. Specifically, the current state of controls received from the host application via the second input port will be recorded, so that subsequent MIDI output for data translations on the first output port will use the proper values for determining the required relative changes. We refer to this as *automatic feedback*. Some devices may provide you with sign-bit encoders which don't need any kind of feedback for themselves. In this case the automatic feedback will be all that's needed to keep controller and application in sync, and you don't even have to write any translation rules for the feedback; just enabling the second input port and hooking it up to the application will be enough.

Other controls such as motor faders will require explicit translation rules for the feedback in the `[MIDI2]` section, however. In the simplest case these may just be the inverse of the rules in the `[MIDI]` section. For instance, if the APCmini had motor faders (it doesn't), we might use rules like the following to translate MCP feedback about the fader positions back to the device:

```
PB[128]-1= CC48
PB[128]-2= CC49
...
PB[128]-8= CC55
```

Translations can also generate their own feedback. To these ends, any MIDI message on the right-hand side of a translation can be prefixed with the `!` character (or the `^` character, which works in an analogous fashion, but has some special logic for dealing with shift keys built into it). This outputs the message as usual, but flips the output ports, so that the message will go to port 2 in a forward translation destined for port 1, and vice versa to port 1 in a feedback translation (in the `[MIDI2]` section) destined for port 2. We call this *direct feedback*. For instance, we can equip the `D8` shift key from the previous subsection with direct feedback as follows:

```
D8 SHIFT ^D8 RELEASE SHIFT ^D8
```

This might then light up the LED of the corresponding button when pressing and turn it off again when releasing the key.

Please note that any kind of controller feedback which goes beyond direct feedback requires that the target application already provides some level of MIDI feedback on its own. midizap is *not* capable of reading the internal state of a non-MIDI application by some other magical means.

4.3. Mod Translations

Most of the time, MIDI feedback uses just the standard kinds of MIDI messages readily supported by midizap, such as note messages which make buttons light up in different colors, or control change messages which set the positions of motor faders. However, there are some encodings of feedback messages which combine different bits of information in a single message, making them difficult or even impossible to translate using the simple kinds of rules we've seen so far. midizap offers a special variation of data translations to help decoding such messages. We call them *mod translations* (a.k.a. "modulus" translations), because they involve operations with integer moduli which enable you to both calculate output from input values in a direct fashion, *and* modify the output messages themselves along the way.

One important task, which we'll use as an example below, is the decoding of meter (RMS level) data in the Mackie protocol. There, each meter value is represented as a channel pressure (`CP`) message whose value consists of a mixer channel index 0..7 in the "high nibble" (bits 4..6) and the corresponding meter value in the "low nibble" (bits 0..3). We will show how to map these values to notes indicating buttons on the AKAI APCmini (Fig. 1). Mod translations aren't limited to this specific use case, however; similar rules will apply to other kinds of "scrambled" MIDI data.

In its simplest form, a mod translation looks as follows (taking channel pressure as an example):

```
CP[16] C0
```

In contrast to the simple kinds of data translations we've seen so far, there's no increment (`+` or `-`) flag here, so the translation does *not* indicate an incremental change of the input value. Instead, mod translations always work with *absolute* values, and the step size on the left-hand side is treated as a *modulus* to decompose the input value into two separate quantities, *quotient* and *remainder*. Only the latter becomes the value of the output message, while the former is used as an offset to modify the output message.

In order to describe more precisely how this works, let's assume an input value v and a modulus k. We divide v by k, yielding the quotient (offset) $q = v \operatorname{div} k$ and the remainder (value) $r = v \bmod k$. E.g., with $k = 16$ and $v = 21$, you'll get $q = 1$ and $r = 5$ (21 divided by 16 yields 1 with a remainder of 5). The calculated offset q is then applied to the note itself, and the remainder r becomes the velocity of that note. So in the example above the output would be the note `C#0` (`C0` offset by 1) with a velocity of 5. On the APCmini, this message will light up the second button in the bottom row of the 8x8 grid in yellow.

Mod translations are midizap's swiss army knife for dealing with complicated translations. There are also some special elements in the MIDI syntax which can be used in mod translations to make them even more flexible:

- The *empty modulus* bracket, denoted `[]` on the left-hand side of a mod translation, indicates a default modulus large enough (16384 for `PB`, 128 for other messages) so that the offset q always becomes zero and the translation passes on the entire input value as is.
- The *transposition* flag, denoted with the `'` (apostrophe) suffix on an output message, reverses the roles of q and r, so that the remainder becomes the offset and the quotient the value of the output message.
- The *change* flag, denoted with the `?` suffix on an output message, only outputs the message if there are any changes in offset or value.
- *Value lists*, denoted as lists of numbers separated by commas and enclosed in curly braces, provide a way to describe *discrete mappings* of input to output values. The input value is used as an index into the list to give the corresponding output value, and the last value in the list will be used for any index which runs past the end of the list. There are also some convenient shortcuts which let you construct these lists more easily: repetition $a{:}b$ (denoting b consecutive a's) and enumeration

a-b (denoting $a, a \pm 1, \ldots, b$, which ramps either up or down depending on whether $a \leq b$ or $a > b$, respectively).

We can't go into all of this here, so we have to refer the reader once again to the manual for details. But here's how we can use a single mod translation to map MCP meter feedback onto the APCmini's topmost five button rows, turning them into a colorful meter display:

```
CP[16] C2{0,1} G#2{0:3,1} E3{0:6,1} C4{0:9,5} G#4{0:12,3}
```

To understand how this works, one must know that the buttons of the 8x8 grid, cf. Fig. 1(6), can be lit up by sending them the appropriate note messages. Rows number 4 to 8 (counting from the bottom) start at notes `C2`, `G#2`, `E3`, `C4` and `G#4`, respectively. The velocities of the notes indicate the colors (0 means off, 1 green, 5 yellow, and 3 red). The rule above will thus light up buttons in different rows in different colors (depending on the low nibble of the channel pressure value), and in different columns (depending on the high nibble of the channel pressure value which ranges from 0 to 7 and indicates the mixer channel).

Mod translations are surprisingly versatile and can be used for various different purposes. In particular, they can also be called as *macros* from other translations. This adds a (rather rudimentary) programming facility to the configuration language, which isn't needed very often, but gives you some extra rope to tackle complicated translations. We won't go into this here, so please check the manual for details and many more examples.

4.4. Pass-Through

There are some situations in which it may be possible to keep most of the controller input and pass it through unchanged. In particular, this case arises in Mackie translations for devices which already support MCP, but might need some minor touches here and there to make them work exactly the way you want.

For instance, Behringer's X-Touch Mini (Fig. 2) is a fairly nice device with its eight encoders providing LED feedback, but its MCP mode is somewhat lacking. One thing that many users of the device complain about is that it doesn't have any keys for changing mixer banks. But in fact the device *has* two "layer" keys on the right which seem ideal for that purpose; alas, the Behringer engineers decided to have them assigned to some other less important MCP functions instead. With midizap it's very easy to fix this shortcoming, by just reassigning the two keys to the much wanted bank change keys:

```
C7   A#3 # BANK LEFT
C#7  B3  # BANK RIGHT
```

We still need to make sure that everything else is passed through unchanged. The most convenient way to do this is to just add the `PASSTHROUGH` directive to the configuration. You can place this anywhere, but it's most convenient to have this kind of stuff at the beginning of the configuration file, before the first translation section. The directive tells midizap to pass a message from the input to the output port if it doesn't have an explicit translation for that message. So the final configuration will look like this:

```
PASSTHROUGH

[MIDI]
C7   A#3 # BANK LEFT
C#7  B3  # BANK RIGHT

[MIDI2]
```

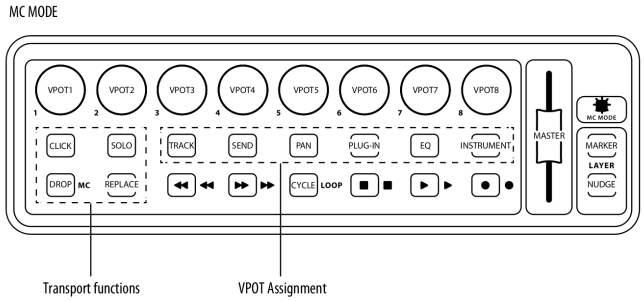

Figure 2: X-Touch Mini [10, p. 16].

```
# feedback for the BANK LEFT/RIGHT buttons
A#3  C7
B3   C#7
```

Here we also added two more translations in the `[MIDI2]` section so that the feedback for the two remapped buttons works as expected. To finish off that little example, you may want to add a few more directives, so that midizap automatically creates the feedback port and auto-connects to the right device and applications; we will discuss these in the next section. You can also find an enhanced version of this example in the sources (XTouchMini.midizaprc), which adds many other useful MCP functions.

Please note that the `PASSTHROUGH` directive only applies to normal (non-system) messages. In some cases it will be necessary to also pass on system messages, such as system exclusive, which can be done with the `SYSTEM_PASSTHROUGH` directive. System exclusive messages are used in MCP to set the contents of the scribble strips. The X-Touch Mini doesn't have these, but other devices like the X-Touch One do, and will thus need system pass-through to function properly (see the XTouchONE.midizaprc example in the sources).

5. JACK INTERFACE

There are some additional directives and corresponding command line options to configure midizap's Jack setup in various ways. If both the command line options and directives in the midizaprc file are used, the former take priority, so that it's possible to override the configuration settings from the command line. Note that all these options can only be set at program startup. If you later edit the corresponding directives in the configuration file, the changes won't take effect until you restart the program.

5.1. Client Setup

The `-j` option and the `JACK_NAME` directive change the Jack client name from the default (`midizap`) to whatever you want it to be. To use this option, simply invoke midizap with `-j` followed by the desired client name, or put a directive like the following into your midizaprc file:

```
JACK_NAME "midizap-XTouchMini"
```

This option is useful, in particular, if you're running multiple instances of midizap with different configurations for different controllers and/or target applications, and you want to have the corresponding Jack clients named differently, so that they can be identified more easily.

We've already seen the `-o` option which is used to equip the Jack client with an additional output port. This can also be achieved with the `JACK_PORTS` directive in the midizaprc file, as follows:

```
JACK_PORTS 1
```

The given number of output ports must be 0, 1 or 2. Zero means that MIDI output is disabled (which is the default). You may want to use `JACK_PORTS 1` if the configuration is primarily aimed at doing MIDI translations, so you'd like to have MIDI output enabled by default. `JACK_PORTS 2` or the `-o2` option indicates that *two* pairs of input and output ports are to be created. As already discussed in Section 4, the second port is typically used to deal with controller feedback from the application.

Not very surprisingly, at least one output port is needed if you want to output any MIDI at all; otherwise MIDI messages on the right-hand side of translations will be silently ignored.

5.2. MIDI Connections

Setting up all the required connections for the Jack MIDI ports can be a tedious and error-prone task, especially if you have to deal with complex setups involving feedback and/or multiple midizap instances. It's possible to automatize the MIDI connections, e.g., with QjackCtl's persistent MIDI patchbay facility, but this is often inconvenient if you need to accommodate multiple midizap configurations and you already have a complicated studio setup (or indeed a bunch of them) which you don't want to mess with.

Therefore midizap offers its own built-in patchbay functionality using the `JACK_IN` and `JACK_OUT` directives which let you specify the required connections in the configuration itself. The port number is tacked on to the directive, so, e.g., `JACK_IN2` connects the second input port. If the port number is omitted then it defaults to 1, so both `JACK_OUT1` and just `JACK_OUT` connect the first output port. The directive is followed by a regular expression to be matched against the Jack MIDI ports of your devices and applications. For instance, the following lines connect midizap to an X-Touch Mini device on one side and Ardour's Mackie control port on the other. (This kind of setup is rather typical for configurations involving feedback. For simple setups just specifying the `JACK_IN` and `JACK_OUT` directives is often sufficient, or even just `JACK_IN` if the target application isn't MIDI-capable.)

```
JACK_IN1  X-TOUCH MINI MIDI 1
JACK_OUT1 ardour:mackie control in
JACK_IN2  ardour:mackie control out
JACK_OUT2 X-TOUCH MINI MIDI 1
```

A connection will be established automatically by midizap whenever a MIDI port belonging to another Jack client matches the regular expression, as well as the port type and I/O direction. This also works dynamically, as new devices get added and new applications are launched at runtime. Only one directive can be specified for each port, but since midizap will connect to all ports matching the given regular expression, you can connect to more than one application or device by just listing all the alternatives. For instance, to have midizap's output connected to both Ardour and Pd, you might use a directive like:

```
JACK_OUT1 ardour:MIDI control in|Pure Data Midi-In 1
```

All matches are done against full port names including the *client-name*: prefix, so you can specify exactly which ports of which clients should be connected. However, note that in contrast to the QJackCtl patchbay, midizap does substring matches by default, so that, e.g., just "`MIDI control`" would match *any* Ardour MIDI control port, in any instance of the program (and also ports with the same name in other programs). If you want to specify an exact match, you need to use the `^` and `$` anchors as follows:

```
JACK_OUT1 ^ardour:MIDI control in$
```

5.3. Jack Sessions

midizap also supports Jack session management which provides a convenient alternative way to launch your midizap instances. Once you've finished a configuration, instead of running midizap manually each time you need it, you just invoke it once with the right command line options, and use a Jack session management program to record the session. The session manager can then be used to relaunch the program with the same options later.

Various Jack session managers are available for Linux, but if you're running QjackCtl already, you might just as well use it to record your sessions, too. QjackCtl's session manager is available in its Session dialog. To use it, launch midizap and any other Jack applications you want to have in the session, and then hit the "Save" button in the Session dialog to have the session recorded. Now, at any later time you can rerun the recorded session with the "Load" button in the same dialog, and your most recent sessions are available in the "Recent" menu from where they can be launched quickly.

6. CONCLUSIONS

I hope that you'll enjoy using midizap for your MIDI mapping needs as much as I do. I'd like to emphasize, however, that midizap is nothing more (and nothing less) than a simple and practical solution to a nagging problem that I have run into time and again (as presumably many Linux MIDI users do). midizap has its limitations, and it is definitely *not* intended as a replacement for more ambitious projects. Ctlra [1] along with its Mappa component takes a much higher-level approach based on the idea of abstracting device interfaces so that basically any Ctlra client can be used with any Ctlra-supported device. This promises to scale much more easily, but it will take its time to gather a critical mass of supported devices and applications.

In the meantime we now have midizap which is a much more modest design, but can make any MIDI controller work with pretty much any application out there, as long as the application can be controlled with keyboard and/or MIDI commands. And you don't need to be a computer expert to use it; if you know how to use Jack, a text editor, and the command line, you're good to go.

Contributions are welcome; in particular, we're looking for interesting configurations to be included in the distribution. I consider midizap itself finished at this point (ports, bugfixes and feature creep notwithstanding), but one area which could still be simplified is the configuration process. While experienced Linux users may actually prefer the textual interface that midizap provides (especially when using midizap's Emacs mode), editing configuration files and watching debugging output in a terminal can be a bit daunting. So a GUI-based configuration front-end (maybe something along the lines of existing MIDI learn facilities) might be in order here.

As Ctlra matures, another interesting possibility is to have a direct interface between Ctlra and midizap at some point. It's already possible to run midizap and Ctlra's daemon program in concert, but tighter integration could be achieved, e.g., by adding a Ctlra backend to midizap.

7. ACKNOWLEDGMENTS

This program wouldn't exist without Eric Messick's prior work, so a big thank you goes out to him. Thanks are also due to Harry van Haaren for helping me with Ctlra, which is used in the NI Maschine Mk3 Mackie emulation distributed with midizap. Last but not least, I'd also like to thank the reviewers for helpful comments.

8. REFERENCES

[1] Harry van Haaren, "openAV-Ctlra: A plain C library to program with hardware controllers," `https://github.com/openAVproductions/openAV-Ctlra`, Sept. 2018.

[2] "MIDI Translator Pro | Bome Software," `https://www.bome.com/products/miditranslator`.

[3] Eric Messick, "ShuttlePRO: User program for interpreting key, shuttle, and jog events from a Contour Design ShuttlePRO v2," `https://github.com/nanosyzygy/ShuttlePRO`, Sept. 2018.

[4] Paul White, "Contour Designs Shuttle Pro v2," `https://www.soundonsound.com/reviews/contour-designs-shuttle-pro-v2`, Oct. 2016.

[5] Albert Gräf, "ShuttlePRO fork," `https://github.com/agraef/ShuttlePRO`, Sept. 2018.

[6] Albert Gräf, "midizap: Control your multimedia applications with MIDI," `https://github.com/agraef/midizap`, Oct. 2018.

[7] "Binding Jump Prevention and Relative (Rotary) Encoder Support - Cantabile - Software for Performing Musicians," `https://www.cantabilesoftware.com/guides/controllerEncoding`.

[8] Nedko Arnaudov, "a2jmidid," `https://repo.or.cz/a2jmidid.git`.

[9] AKAI Professional, "APC mini - User Guide," `http://www.akaipro.com/products/pad-controllers/apc-mini`.

[10] Behringer, "X-TOUCH MINI Quick Start Guide," `https://media.music-group.com/media/PLM/data/docs/P0B3M/X-TOUCH%20MINI_QSG_WW.pdf`.

A. CONFIGURATION SYNTAX

```
config     ::= { directive | header | translation }
header     ::= "[" name "]" [ "CLASS" | "TITLE" ] regex
translation::= midi-token { key-token | midi-token }

directive  ::= "DEBUG_REGEX"|"DEBUG_STROKES"|"DEBUG_KEYS" |
               "DEBUG_MIDI" | "MIDI_OCTAVE" number |
               "JACK_NAME" string | "JACK_PORTS" number |
               "JACK_IN" [number] regex |
               "JACK_OUT" [number] regex |
               "PASSTHROUGH" [ number ] |
               "SYSTEM_PASSTHROUGH" [ number ]

midi-token ::= msg [ mod ] [ steps ] [ "-" number] [ flag ]
msg        ::= ( note | other | "M" ) [ number ]
note       ::= ( "A" | ... | "G" ) [ "#" | "b" ]
other      ::= "CH" | "PB" | "PC" | "CC" | "CP" | "KP:" note
mod        ::= "[" [ number ] "]"
steps      ::= "[" number "]" | "{" list "}"
list       ::= number { "," number | ":" number | "-" number }
flag       ::= "-" | "+" | "=" | "<" | ">" | "~" |
               "'" | "?" | "'?" | "?'"

key-token  ::= "RELEASE" | "SHIFT" [ number ] |
               keycode [ "/" keyflag ] | string
keycode    ::= "XK_Button_1" | "XK_Button_2" | "XK_Button_3" |
               "XK_Scroll_Up" | "XK_Scroll_Down" |
               "XK_..." (see /usr/include/X11/keysymdef.h)
keyflag    ::= "U" | "D" | "H"
string     ::= '"' { character } '"'
```