Table of Contents

What is evsieve?

Evsieve (from "event sieve") is a low-level utility that can read events from Linux event devices (evdev) and write them to virtual event devices (uinput), performing simple manipulations on the events along the way. Examples of things evsieve can do are:

- Remap certain keyboard keys to others;
- Send some keyboard keys to one virtual device and other keyboard keys to another virtual device;
- Turn a joystick into a virtual keyboard.

Evsieve is particularly intended to be used in conjunction with the evdev-passthrough functionality of Qemu. For other purposes, you may be better off using a higher-level utility than evsieve.

Examples of things that evsieve can achieve that are useful for evdev-passthrough are:

- Keeping certain keys on the host while other keys are passed through to the guest;
- Run scripts based on hotkeys even if your keyboard is passed through to the guest;
- Remap another key to lctrl+rctrl, so you can use another key to attach/detach your keyboard to your VM;
- Split the events over multiple virtual devices, possibly passing some of those devices to different VMs or keeping one of them on the host;
- Remap keys on your keyboard on such a low level that even the guest OS can't tell the difference between real and mapped keys.

Evsieve is intended to make simple manipulations easy. It is not a tool for keyboard macro's or complex event manipulation. In case you want to do manipulations that are beyond the scope of evsieve, we recommend using the Python-evdev library instead.

Compilation

Compiling this program requires the Rust toolchain and the libevdev library. To install these dependencies:

- Debian / Ubuntu: sudo apt install cargo libevdev2 libevdev-dev
- Fedora: sudo dnf install cargo libevdev libevdev-devel
- Arch Linux: sudo pacman -S rust libevdev

After you've installed the dependencies, you can obtain a copy of the source code and compile the program using:

```
wget https://github.com/KarsMulder/evsieve/archive/v1.3.1.tar.gz -O evsieve-1.3.1.tar.gz
tar -xzf evsieve-1.3.1.tar.gz && cd evsieve-1.3.1
cargo build --release
```

An executable binary can then be found in target/release/evsieve. You can either execute it as-is, or copy it to somewhere in your PATH for your convenience, e.g.:

```
sudo install -m 755 -t /usr/local/bin target/release/evsieve
```

The source code repository contains some files which were automatically generated. You can optionally regenerate these files before compilation. To regenerate these files, run the generate_bindings.sh script before running cargo build. This script requires the rust-bindgen tool to be installed on your system.

Usage: Basic concepts

Linux event devices are a tool the kernel uses to inform userspace applications (such as Xorg, Wayland) about input events such as keyboard presses, mouse movements. Event devices reside in /dev/input and usually have named symlinks pointed to them in /dev/input/by-id.

Event devices emit a stream of events. For example, when you press the A key on your key board, it will emit an (EV_KEY, KEY_A, 1) event, and when you release it it will emit an (EV_KEY, KEY_A, 0) event.

The quickest way to get an idea of how it works is to just look at it. You can either install the utility evtest (shipped by most major Linux distributions), run it as sudo evtest, select your keyboard/mouse/whatever and look at the events it emits, or run evsieve with the following arguments to see all events that are emitted on your computer:

```
sudo evsieve --input /dev/input/event* --print
```

Reading and writing

Evsieve is a command-line program designed to read and transform input events emitted by (real) event devices and write them to virtual event devices. It most likely needs to run as root for most features.

Assume you have a keyboard, and a symlink to its event device can be found at /dev/input/by-id/keyboard. A very basic usage of evsieve would be the following:

```
evsieve --input /dev/input/by-id/keyboard grab --output
```

In this example, evsieve will open the event device /dev/input/by-id/keyboard for exclusive read access (specified by the grab flag), read events from it, and write those same events to an unnamed virtual event device. This changes the flow of events from:

- Keyboard â Kernel â Event Device â Xorg/Wayland â Applications

To:

- Keyboard â Kernel â Event Device â Evsieve â Virtual Event Device â Xorg/Wayland â Applications

This has effectively accomplished nothing besides adding about 0.15ms of latency. However, if we add additional arguments to evsieve, we can use this construction to add, change, or suppress events. For example, the following script will turn the capslock into a second backspace:

```
evsieve --input /dev/input/by-id/keyboard grab \
        --map key:capslock key:backspace \
        --output
```

Since evsieve has opened your keyboard /dev/input/by-id/keyboard with exclusive access due to the grab flag, the userspace programs won't notice when you press capslock on your real keyboard. Evsieve will then read this capslock event, turn it into a backspace key event and write that to the virtual device, tricking userspace into thinking you just pressed the backspace button instead of the capslock button.

Evsieve actively maps events from the input devices to the output devices as long as evsieve runs. When evsieve exits, the virtual devices will be removed and any grabbed input devices will be released. You can exit evsieve by sending it a SIGINT or SIGTERM signal, for example by pressing Ctrl+C in the terminal where evsieve runs.

Take note that evsieve deals with events, not keys. This doesn't turn the capslock key into a backspace key, but rather turns every event associated with capslock into an event associated with backspace, e.g. a key_down for capslock is turned into a key_down for backspace, and a key_up event for capslock is turned into a key_up event for backspace. This distinction is important for some of the more advanced uses of evsieve.

Sequential event processing

The order of the arguments provided to evsieve matters. This is similar to how ImageMagick works and unlike the POSIX convention.

The --input argument reads events from the disk and generates a stream of events that will be processed by the arguments that come after. Each argument modifies the stream of events generated by previous argument. The --map arguments modify the stream of events by turning some events into other events. The --output argument consumes the events and writes them to a virtual event device.

In bash terms, you could think of all evsieve's arguments as if they were mini programs which have their outputs piped to each other. The previous script can be thought of as if it were something like the following:

```
# Note: this is not an actually valid evsieve script.
evsieve-input --grab /dev/input/by-id/keyboard | evsieve-map key:capslock key:backspace |
```

Due to performance concerns and some technical reasons, evsieve is not structured as a set of mini-programs but rather as a single monolithic process, but the idea is the same.

Usage: Examples

In this section we'll introduce the capabilities of evsieve through example scripts. All examples will assume that your keyboard can be found at /dev/input/by-id/keyboard and your mouse can be found at /dev/input/by-id/mouse. If you want to use these scripts, you need to replace these placeholder paths with the real paths to your keyboard and mouse.

All these scripts need to run with a certain amount of privileges. That usually means you'll need to execute them as root using sudo. If you want to put effort in making it run with more minimal permissions using ACLs, it is sufficient for most purposes to have evsieve run as a user with read/write access to /dev/uinput and all devices you try to open.

See which events are being emitted by your keyboard

```
evsieve --input /dev/input/by-id/keyboard --print
```

You can insert a --print argument at any point in your script to see which events are being processed by evsieve at that point. The events are printed the way they are seen by evsieve at that point; if you applied some maps or other transformations before the --print statement, then the events printed are the mapped events, not the original ones.

This is useful for debugging your scripts, and is also useful if you want to know what a certain key or button is called by evsieve. You can alternatively use the standard utility evtest (shipped by most major Linux distributions), which will provide a bit more detailed information using the Linux kernel's standard terminology.

Execute a script when some hotkey is pressed

```
evsieve --input /dev/input/by-id/keyboard \
        --hook key:leftctrl key:h exec-shell="echo Hello, world!"
```

You can execute a certain script when a certain key or combination of keys are pressed simultaneously using the --hook argument.

Each time the left control and the H key are pressed simultaneously, evsieve will execute the command echo Hello, world! using your OS' default POSIX shell, i.e. /bin/sh -c "echo Hello, world!". Note that the script will be executed as the same user as evsieve runs as, which may be root.

Since we did not need to modify the events and merely read them, there is no grab flag on the input device, causing evsieve to open it in nonexclusive mode. Since we didn't claim exclusive access to the device, there is no need to generate a virtual device using --output either.

In case some other program (such as Qemu) claims exclusive access to your keyboard and you still want to be able to listen to hotkeys, you can generate a virtual device and let Qemu claim access to the virtual device instead:

```
evsieve --input /dev/input/by-id/keyboard grab \
        --hook key:leftctrl key:h exec-shell="echo Hello, world!" \
        --output create-link=/dev/input/by-id/virtual-keyboard
```

This will generate a virtual device and create a link to it at /dev/input/by-id/virtual-keyboard, which you can pass to Qemu or whatever program demands exclusive access.

Remap your favourite key to lctrl+rctrl

```
evsieve --input /dev/input/by-id/keyboard grab \
        --map key:scrolllock key:leftctrl key:rightctrl \
        --output create-link=/dev/input/by-id/virtual-keyboard
```

In case you want to use Qemu evdev passthrough and use some other key than lctrl+rctrl to switch between guest and host, this is the second-easiest solution. (The easiest solution, of course, is using Qemu's built-in functionality for this purpose.)

Swap the control and the shift keys

```
evsieve --input /dev/input/by-id/keyboard grab    \
        --map yield key:leftctrl   key:leftshift  \
        --map yield key:leftshift  key:leftctrl   \
        --map yield key:rightctrl  key:rightshift \
        --map yield key:rightshift key:rightctrl  \
        --output
```

In previous examples, we mapped a single key using --map. In this example we do the same, except this time we also pass the yield flag to the --map arguments.

Remember that the output events of one argument are input events for the next argument. The --input argument emits a constant stream of events that are processed by all the maps until they encounter an --output event.

A first map like --map key:leftctrl key:leftshift could turn a control key event into a shift key event without problem. However, if the next argument were to naively be --map key:leftshift key:leftctrl, then the second map would turn the shift key event back into a control keyâ   undoing the effect of the first map.

There are use cases where this behaviour is useful. For cases where it isn't, the --map argument accepts a yield flag. All events that are generated by a --map yield are considered "yielded" and will not trigger any further maps, nor any other arguments except for the --output argument. This prevents the shift event generated by the first map from getting turned back to a ctrl event by later maps.

Process two devices simultaneously

```
evsieve --input /dev/input/by-id/keyboard grab \
        --output                               \
        --input /dev/input/by-id/mouse grab    \
        --output
```

Evsieve is not restricted to a single input or output, and is also not restricted to only keyboards.

This results into two virtual devices, one virtual device for the keyboard event (created by the first --output) and a virtual device for all mouse events (created by the second --output). The keyboard arguments are not sent to the virtual mouse because no event in the event processing stream gets past the first --output argument.

It is also possible to map both the keyboard and the mouse to a single virtual hybrid key-mouse device in the following way.:

```
evsieve --input /dev/input/by-id/keyboard grab \
        --input /dev/input/by-id/mouse grab    \
        --output
```

Keeping certain keyboard keys on the host after passing a keyboard to the guest

```
evsieve --input /dev/input/by-id/keyboard grab \
        --output key:f9 key:f10 key:f11 key:f12 \
        --output create-link=/dev/input/by-id/virtual-keyboard
```

This example is useful for those who wish to do evdev passthrough using Qemu.

It is possible to specify event filters on --output arguments. By default an --output argument will take all events and write them to an output device, but if one or more filters are specified, then only events that match at least one of those filters will be written to that output device.

In this example, the first --output argument creates an unnamed virtual device, and writes all events related to the F9â   F12 keys to it. Those events are then removed from the processing stream.

All events that were not removed by the first --output argument encounter the second --output argument, which applies to all events and sends them to a second virtual device which is linked to at /dev/input/by-id/virtual-keyboard.

You can then pass the latter device /dev/input/by-id/virtual-keyboard through to your virtual machine using evdev passthrough and keep the former unnamed device on the host. Now your host will still be able to receive the F9â   F12 keys from the real keyboard while the majority of your keys are send to your VM through the latter keyboard.

Toggle events between two virtual devices

```
evsieve --input  /dev/input/by-id/keyboard domain=kb grab \
        --input  /dev/input/by-id/mouse    domain=ms grab \
        --hook   key:scrolllock toggle   \
        --map    key:scrolllock @host-kb \
        --toggle @kb @guest-kb @host-kb  \
        --toggle @ms @guest-ms @host-ms  \
        --output @host-kb create-link=/dev/input/by-id/host-keyboard   \
        --output @host-ms create-link=/dev/input/by-id/host-mouse      \
        --output @guest-kb create-link=/dev/input/by-id/guest-keyboard \
        --output @guest-ms create-link=/dev/input/by-id/guest-mouse
```

A somewhat more involved solution that's primarily useful for switching your input between the host and guest when doing evdev passthrough. This script reads your real keyboard and mouse, and creates two virtual devices for each of them.

By default, all events will be written to the guest-* devices, but by pressing the scrolllock key, you can change that to send write all of them to the host-* devices instead. Pressing scrolllock again with switch them back to the guest-* devices.

If you then pass the guest-* devices through to your virtual machine and leave the host-* devices on your host, then you can switch your input between your host and guest by pressing the scrolllock key. The scrolllock key is always sent to the host using --map so the LED indicator works properly.

You could also do something like passing both of the devices through to two different virtual machines, so you can toggle between both of the machines instead of toggling between host and guest. Of course, it is also possible to create more than two virtual devices per real device: just add more @something domains to the --toggles along with corresponding --outputs.

Explanation

Two new features are introduced in this script: the --toggle argument, and the concept of domains. A toggle is like a map, except that only one of the output keys is active at a time, and the active output key is switched every time a --hook toggle is triggered. An argument like --toggle key:a key:b key:c will by default map all A key events to B key events, but when a --hook toggle is triggered, it will start mapping them to C key events instead. Triggering a --hook toggle again will make the toggle go back to turning them into B key events.

In this script, we do not match events by type or code such as key:a, but instead by domains. In evsieve logic, all events have a domain attached to them. A domain is a string of text without any intrinsic meaning; they are not something that exists in the evdev protocol, but they are useful for writing evsieve scripts. Because the domain=kb clause was set on an input device, all events read from that device will have the domain kb attached to it.

A filter like @kb matches all events which have domain kb, just like a filter key:a matches all events with type EV_KEY and code KEY_A. The argument --toggle @kb @guest-kb @host-kb will take all events with domain kb and change their domain to guest-kb. When the --hook toggle is triggered, it will start changing their domain to host-kb instead.

Next, the output devices also filter by domain. There is one virtual output which will write all events with domain guest-kb to it, and another output which will write all events with domain host-kb to it. Thus, by changing the domains of events, this --toggle effectively decides to which virtual device the events will be written.

Advanced variations

In case you want a set/reset behaviour instead of toggle behaviour, i.e. one key that always switches to host and one key that always switches to guest, you can specify an index for the toggle, for example:

```
evsieve --input  /dev/input/by-id/keyboard domain=kb grab \
        --input  /dev/input/by-id/mouse    domain=ms grab \
        --hook   key:kpenter toggle=:1  \
        --hook   key:kpdot   toggle=:2  \
        --block  key:kpenter key:kpdot  \
        --toggle @kb @guest-kb @host-kb \
        --toggle @ms @guest-ms @host-ms \
        --output @host-kb create-link=/dev/input/by-id/host-keyboard   \
        --output @host-ms create-link=/dev/input/by-id/host-mouse      \
        --output @guest-kb create-link=/dev/input/by-id/guest-keyboard \
        --output @guest-ms create-link=/dev/input/by-id/guest-mouse
```

This uses the numpad's enter key to switch to the guest and the numpad's dot key to switch to the host. The --block argument drops all events matching one of the keys specified, so the numpad enter/dot are sent to neither host or guest.

You can go further and specify some exec=shell= clause on the hooks to execute a certain script whenever the input target changes. For example, if your monitor supports the DDC2 protocol, you may be able to use the ddcutil utility to change its input source every time you switch between host and guest:

```
evsieve --input  /dev/input/by-id/keyboard domain=kb grab \
        --input  /dev/input/by-id/mouse    domain=ms grab \
        --hook   key:kpenter toggle=:1 exec-shell="ddcutil setvcp 60 0x11" \
        --hook   key:kpdot   toggle=:2 exec-shell="ddcutil setvcp 60 0x0f" \
        --block  key:kpenter key:kpdot  \
        --toggle @kb @guest-kb @host-kb \
        --toggle @ms @guest-ms @host-ms \
        --output @host-kb create-link=/dev/input/by-id/host-keyboard   \
        --output @host-ms create-link=/dev/input/by-id/host-mouse      \
        --output @guest-kb create-link=/dev/input/by-id/guest-keyboard \
        --output @guest-ms create-link=/dev/input/by-id/guest-mouse
```

You may need to adjust the numbers involved depending on your monitor. Not all monitors have proper support for the DDC2 protocol and/or may refuse to listen to commands that do not come from its active input. Read the ddcutil manual for more information.

Control your mouse using your keyboard

```
evsieve --input /dev/input/by-id/keyboard \
        --input /dev/input/by-id/mouse grab domain=mouse \
        --map key:left:1~2      rel:x:-20@mouse \
        --map key:right:1~2     rel:x:20@mouse  \
        --map key:up:1~2        rel:y:-20@mouse \
        --map key:down:1~2      rel:y:20@mouse  \
        --map key:enter:0~1     btn:left@mouse  \
        --map key:backslash:0~1 btn:right@mouse \
        --output @mouse
```

This script will create a virtual mouse that moves with every key_down or key_repeat event on the arrow keys, and can left/right click with the enter/backslash keys. Keyboard to mouse mapping is not what evsieve was intended to be used for, but it works anyway to some extent.

A new feature in this script is value mapping. In previous scripts, when we saw an argument like --map key:a key:b it meant "turn any event of type EV_KEY, code KEY_A, any value, into an event of type EV_KEY, code KEY_B, with the same value". In this script, we're not only mapping the event type and code, but also the value.

The key rel:x:-20 means "an event with type EV_REL, code REL_X, value -20". The key key:left:1~2 means "an event with type EV_KEY, code KEY_LEFT, value between 1 and 2", where the ~ denotes a range. Thus the map --map key:left:1~2 rel:x:-20 maps any event of type EV_KEY, code KEY_LEFT, value 1 or 2 to an event of type EV_REL, code REL_X, value -20. This effectively makes any key down (value 1) or key repeat (value 2) event on the left arrow key change the mouse's X axis with -20.

In the later map --map key:enter:0~1 btn:left you may notice that no value is specified for the output event. This means that it's value is taken to be the same as the input event, which conveniently maps a key down event (value 1) to a click event, and a key up event (value 0) to a release event.

Finally, all events with the domain mouse are written to a virtual device. This are all events generated by the real mouse and all events that were created by a map. Events generated by your real keyboard that didn't match any map will not be written to this virtual device. Note that there is no grab flag provided to the keyboard input, so evsieve will open the keyboard in nonexclusive mode, allowing other programs to read from it as well.

Map a controller to a virtual keyboard

If an application does not support controllers or joysticks but you want to control them with a controller or joystick anyway, you can use evsieve to turn your controller/joystick into a virtual keyboard.

Controllers are seen as joysticks by the Linux kernel. The good news is that they have event devices just like keyboards and can be processed by evsieve. The bad news is that the buttons on them do not tend to have sensible names or event codes, so the commands you need to map them tend to be cryptic and dependent on what controller you use.

Below are example scripts that create a virtual keyboard based on a DualSense or Xbox One controller. You can change which keyboard keys the controller buttons get mapped to as you please.

If you want to write a mapping script for some other controller or joystick, we recommend using the --print argument or the Linux utility evtest to discover which event codes are associated with the buttons on your controller.

DualSense controller:

```
evsieve --input /dev/input/by-id/usb-Sony_Interactive_Entertainment_Wireless_Controller-if
                                                     \
        `# Directional pad`                          \
        --copy abs:hat0x:-1      key:left:1@kb  \
        --copy abs:hat0x:-1..0~  key:left:0@kb  \
        --copy abs:hat0x:1       key:right:1@kb \
        --copy abs:hat0x:1..~0   key:right:0@kb \
        --copy abs:hat0y:-1      key:up:1@kb    \
        --copy abs:hat0y:-1..0~  key:up:0@kb    \
        --copy abs:hat0y:1       key:down:1@kb  \
        --copy abs:hat0y:1..~0   key:down:0@kb  \
                                                     \
        `# Shape buttons: circle, cross, square, triangle` \
        --copy btn:c             key:z@kb       \
        --copy btn:east          key:x@kb       \
        --copy btn:south         key:c@kb       \
        --copy btn:north         key:v@kb       \
                                                     \
        `# Left pad`                                 \
        --copy abs:z:38~..~37    key:j:1@kb     \
        --copy abs:z:~37..38~    key:j:0@kb     \
        --copy abs:z:~216..217~  key:l:1@kb     \
        --copy abs:z:217~..~216  key:l:0@kb     \
        --copy abs:rz:38~..~37   key:i:1@kb     \
        --copy abs:rz:~37..38~   key:i:0@kb     \
        --copy abs:rz:~216..217~ key:k:1@kb     \
        --copy abs:rz:217~..~216 key:k:0@kb     \
                                                     \
        `# Right pad`                                \
        --copy abs:x:38~..~37    key:a:1@kb     \
        --copy abs:x:~37..38~    key:a:0@kb     \
        --copy abs:x:~216..217~  key:d:1@kb     \
        --copy abs:x:217~..~216  key:d:0@kb     \
        --copy abs:y:38~..~37    key:w:1@kb     \
        --copy abs:y:~37..38~    key:w:0@kb     \
        --copy abs:y:~216..217~  key:s:1@kb     \
        --copy abs:y:217~..~216  key:s:0@kb     \
                                                \
        `# Press left pad, press right pad`     \
        --copy btn:select        key:t@kb       \
        --copy btn:start         key:g@kb       \
                                                \
        `# Back buttons: LB, RB, LT, RT`        \
        --copy btn:west         key:q@kb        \
        --copy btn:z            key:e@kb        \
        --copy abs:rx:~50..51~  key:u:1@kb      \
        --copy abs:rx:51~..~50  key:u:0@kb      \
        --copy abs:ry:~50..51~  key:o:1@kb      \
        --copy abs:ry:51~..~50  key:o:0@kb      \
                                                \
        `# Special buttons: create, start, PlayStation, touch pad, microphone` \
        --copy btn:tl2           key:y@kb       \
        --copy btn:tr2           key:enter@kb   \
        --copy btn:mode          key:esc@kb     \
        --copy btn:thumbl        key:n@kb       \
        --copy btn:thumbr        key:m@kb       \
                                                \
        --output @kb repeat
```

Xbox One controller:

```
evsieve --input /dev/input/by-id/usb-Microsoft_Controller_*-event-joystick \
                                                      \
        `# Directional pad`                           \
        --copy abs:hat0y:-1           key:up:1@kb     \
        --copy abs:hat0y:-1..0~       key:up:0@kb     \
        --copy abs:hat0y:1            key:down:1@kb   \
        --copy abs:hat0y:1..~0        key:down:0@kb   \
        --copy abs:hat0x:-1           key:left:1@kb   \
        --copy abs:hat0x:-1..0~       key:left:0@kb   \
        --copy abs:hat0x:1            key:right:1@kb  \
        --copy abs:hat0x:1..~0        key:right:0@kb  \
                                                      \
        `# Letter buttons: A, B, X, Y`                \
        --copy btn:south              key:z@kb        \
        --copy btn:east               key:x@kb        \
        --copy btn:north              key:c@kb        \
        --copy btn:west               key:v@kb        \
                                                      \
        `# Left pad`                                  \
        --copy abs:x:~23169..23170~   key:d:1@kb      \
        --copy abs:x:23170~..~23169   key:d:0@kb      \
        --copy abs:x:-23169~..~-23170 key:a:1@kb      \
        --copy abs:x:~-23170..-23169~ key:a:0@kb      \
        --copy abs:y:~23169..23170~   key:s:1@kb      \
        --copy abs:y:23170~..~23169   key:s:0@kb      \
        --copy abs:y:-23169~..~-23170 key:w:1@kb      \
        --copy abs:y:~-23170..-23169~ key:w:0@kb      \
                                                      \
        `# Right pad`                                 \
        --copy abs:rx:~23169..23170~   key:l:1@kb     \
        --copy abs:rx:23170~..~23169   key:l:0@kb     \
        --copy abs:rx:-23169~..~-23170 key:j:1@kb     \
        --copy abs:rx:~-23170..-23169~ key:j:0@kb     \
        --copy abs:ry:~23169..23170~   key:k:1@kb     \
        --copy abs:ry:23170~..~23169   key:k:0@kb     \
        --copy abs:ry:-23169~..~-23170 key:i:1@kb     \
        --copy abs:ry:~-23170..-23169~ key:i:0@kb     \
                                                      \
        `# Press left pad, press right pad`           \
        --copy btn:thumbl              key:t@kb       \
        --copy btn:thumbr              key:g@kb       \
                                                      \
        `# Back buttons: LB, RB, LT, RT`              \
        --copy btn:tl                  key:q@kb       \
        --copy btn:tr                  key:e@kb       \
        --copy abs:z:~203..204~        key:u:1@kb     \
        --copy abs:z:204~..~203        key:u:0@kb     \
        --copy abs:rz:~203..204~       key:o:1@kb     \
        --copy abs:rz:204~..~203       key:o:0@kb     \
                                                      \
        `# Special buttons: start, select, Xbox button` \
        --copy btn:start               key:enter@kb   \
        --copy btn:select              key:space@kb   \
        --copy btn:mode                key:esc@kb     \
                                                      \
        --output @kb repeat
```

The major new feature used in these scripts are transitions: a transition like ~216..217~ matches an event with a value of 217 or greater, provided that the last event with the same type and code had a value of 216 or less. These transitions are useful for detecting when an absolute axis event goes over or under a certain threshold.

The chosen thresholds determine how far you need to move a stick or trigger to make them cause a keyboard event. You can adjust them to make the maps more or less sensitive to small movements.

You may further notice that these scripts use the --copy argument instead of --map. The --copy and --map arguments are identical with one difference: source events matching a --map are removed from the processing stream, but those matching --copy are not. In these scripts --copy is used because it might be theoretically possible that e.g. a abs:hat0x:-1 event gets followed up by a abs:hat0x:1 event, which should simultaneously release the left key and press the right key.

The repeat flag makes the virtual keyboard emit key repeat events if a button is held down, like a keyboard does, even though the controllers do not emit repeat events themselves. Since most applications that read event devices ignore repeat events anyway, this clause isn't very important.

Change your keyboard layout on an evdev level

It is probably better to change your keyboard layout by changing your OS settings, but if that isn't possible for some reason, it is possible to use evsieve to map every key to its corresponding key on your favourite keyboard layout. For example, the following script will map the Qwerty keys to Colemak:

```
evsieve --input /dev/input/by-id/keyboard grab \
        --map yield key:e key:f \
        --map yield key:r key:p \
        --map yield key:t key:g \
        --map yield key:y key:j \
        --map yield key:u key:l \
        --map yield key:i key:u \
        --map yield key:o key:y \
        --map yield key:s key:r \
        --map yield key:d key:s \
        --map yield key:f key:t \
        --map yield key:g key:d \
        --map yield key:j key:n \
        --map yield key:k key:e \
        --map yield key:l key:i \
        --map yield key:n key:k \
        --map yield key:p key:semicolon \
        --map yield key:semicolon key:o \
        --map yield key:capslock key:backspace \
        --output
```

Change how keys are mapped at runtime

Similar to how domains and toggles can be used to decide to which output device events are written, they can also be used to determine whether maps apply to events. For example, the following script will toggle between using the default keyboard layout (assumed to be Qwerty) and the Colemak layout by pressing lctrl+rctrl:

```
evsieve --input /dev/input/by-id/keyboard grab \
        --hook key:leftctrl key:rightctrl toggle \
        --toggle "" @qwerty @colemak \
        --map yield key:e@colemak key:f \
        --map yield key:r@colemak key:p \
        --map yield key:t@colemak key:g \
        --map yield key:y@colemak key:j \
        --map yield key:u@colemak key:l \
        --map yield key:i@colemak key:u \
        --map yield key:o@colemak key:y \
        --map yield key:s@colemak key:r \
        --map yield key:d@colemak key:s \
        --map yield key:f@colemak key:t \
        --map yield key:g@colemak key:d \
        --map yield key:j@colemak key:n \
        --map yield key:k@colemak key:e \
        --map yield key:l@colemak key:i \
        --map yield key:n@colemak key:k \
        --map yield key:p@colemak key:semicolon \
        --map yield key:semicolon@colemak key:o \
        --map yield key:capslock@colemak key:backspace \
        --output
```

Based on the state of the toggle, all events either get tagged with the domain qwerty or colemak. The further maps then only apply to events that are tagged with the domain colemak. No matter which domain the events belong to, they get written to the same output device.

Running evsieve as a systemd service

Since version 1.3, evsieve has optional systemd integration, specifically if evsieve is ran as a systemd service with service type "notify", evsieve will notify systemd when it has created all virtual output devices and is ready to start listening for input events.

The systemd-run command makes evsieve easy to integrate in shell scripts where you want to do something with the virtual output devices, for example:

```
systemd-run --service-type=notify --unit=virtual-keyboard.service evsieve \
        --input /dev/input/by-id/keyboard \
        --output create-link=/dev/input/by-id/virtual-keyboard
```

```
# After systemd-run returns successfully, /dev/input/by-id/virtual-keyboard exists. You ca
# immediately run any scripts that depend on it existing. There is no need for any kind of
# `sleep` call here.

# Example: pass the virtual device to Qemu and wait until Qemu exits.
qemu-system-x86_64 \
    `# More arguments here` \
    -object input-linux,id=virtual-keyboard,evdev=/dev/input/by-id/virtual-keyboard

# Finally, after the virtual devices are no longer needed, you can stop evsieve like this.
systemctl stop virtual-keyboard.service
```

Micro-tutorial: error handling with systemd-run:**

If evsieve exits with an error status (e.g. invalid argument or input device not available) then the unit name (in the above example, virtual-keyboard.service) remains in use. When you try to run the same script again, you will get the following error message:

```
Failed to start transient service unit: Unit virtual-keyboard.service already exists.
```

This is a double edged sword: on one hand, this makes it easy to find which part of your shell script broke and allows you to check the status of virtual-keyboard.service to see what went wrong. To check the status and error messages, use systemctl status virtual-keyboard.service or journalctl -u virtual-keyboard.service.

On the other hand, it prevents you from just rerunning the same script without manual intervention. If you get the above error message, you need to run systemctl reset-failed virtual-keyboard.service to make that unit name available again. Alternatively, you can pass the --collect flag to systemd-run to make sure the unit name stays available even if evsieve exits with error status.

Caveats and known issues:

When using a SELinux-enabled operating system such as Fedora, systemd may fail to execute the evsieve binary unless it is located in one of the standard executable directories such as /usr/local/bin.

When evsieve is not ran as root, evsieve may be unable to notify the systemd daemon that it is ready unless the property NotifyAccess=all is set. When running evsieve using systemd-run, this property can be set like systemd-run --service-type=notify --property=NotifyAccess=all evsieve.

Usage: In detail

In this section, we'll provide comprehensive documentation about all the arguments evsieve accepts.

Maps

The --map and --copy arguments have the following basic syntax:

```
--map  SOURCE_EVENT [TARGET_EVENT...] [yield]
--copy SOURCE_EVENT [TARGET_EVENT...] [yield]
```

A map triggers every time an event matching the SOURCE_EVENT is generated by a physical device, and then generates event(s) matching all TARGET_EVENT(s) specified.

When a source event triggers a --map, it is removed from the processing stream and will not trigger any further arguments. The --copy argument has the same syntax and semantics as --map with the single difference being that source events that trigger --copy are not removed from the processing stream.

If a target event is not fully specified (e.g. it specifies a keycode but not a value), all missing details of the target event shall be taken to be the same as those of the source event. If zero target events are specified on a --map, then the source event is removed from the processing stream and no new events are added.

Key format

Keys can be provided to the --map arguments. They use the same form for both the input event and output events. They have the form

```
event_type:event_code:event_value@domain
```

For more information about the "@domain" part, see the "Domains" section below. We will first talk about the type, code, and value.

Most parts are optional. In formal format, all parts between [] are optional:

```
[event_type[:event_code[:event_value]]][@domain]
```

Any part that is not specified will be interpreted to mean "any" for source events. For example:

```
""               # The empty string matches any event
key:a            # Matches any event with type EV_KEY, code KEY_A
key:a:1          # Matches any event with type EV_KEY, code KEY_A, value 1 (down)
key:a@keyboard   # Matches any event with type EV_KEY, code KEY_A with domain "keyboard
@keyboard        # Matches any event with domain "keyboard"
```

For output events, any part that is not specified will be interpreted as "same as the source event". For example:

```
--map key:a   key:b    # Will map any EV_KEY, KEY_A event to a EV_KEY, KEY_B event wit
--map key:a:1 key:b:0  # Will map any EV_KEY, KEY_A, value=1 event to a EV_KEY, KEY_B,
--map key:a   ""       # Will map any EV_KEY, KEY_A event to an identical event.
```

The value for input events may also be expressed as an (optionally bounded) range. For example:

```
key:a:1~2       # Matches any event with type EV_KEY, code KEY_A, a value of 1 (down)
abs:x:10~50     # Matches any event with type EV_ABS, code ABS_X, a value between 10 a
abs:x:51~       # Matches any event with type EV_ABS, code ABS_X, a value of 51 or hi
```

Furthermore, when matching an input event, it is possible to specify a transition of values. If a transition is specified, then a certain event only matches if its value lies in the specified range and the value of the previous event with the same type and code from the same input device also lies within a certain range. For example:

```
key:a:1..2       # Matches an A key event with value 2 (repeat),
                 # if the last A key event had value 1 (down).
abs:x:~50..51~   # Matches an X axis event with a value of 51 or greater,
                 # provided that the previous X axis value was 50 or less.
```

Ranges and transitions are particularly useful for mapping absolute events to key events. For example, the following maps will press the A key when the X axis goes over 200, and releases the key when it goes under it:

```
--map abs:x:~199..200~ key:a:1
--map abs:x:200~..~199 key:a:0
```

Finally, output events can take d or [factor]d as value where d (short for "delta" or "difference") represents the difference between the event's value and the value of the previous event with the same type and code generated by the same input device. This is mainly useful for mapping abs events to rel events. For example:

```
# If an abs:x:10 event gets followed up by a abs:x:50 event, then the latter one gets r
--map abs:x rel:x:d

# If an abs:y:10 event gets followed up by a abs:y:50 event, then the latter one gets r
--map abs:y rel:y:0.2d
```

Key names

All names that evsieve uses for events are derived from the names used by the Linux kernel for such events, using a fairly systematic way that's understood most quickly by looking at the following examples:

```
EV_KEY, KEY_A -> key:a
EV_REL, REL_X -> rel:x
EV_ABS, ABS_X -> abs:x
EV_KEY, KEY_LEFTCTRL -> key:leftctrl
EV_REL, REL_WHEEL_HI_RES -> rel:wheel_hi_res
```

In case you're wondering what the kernel calls a certain button on your device, we recommend using the third-party program evtest, which is probably shipped by your favourite distribution.

There is one notable exception to the above scheme. There are events with type EV_KEY and a code of form BTN_*. These are called btn:something by evsieve even though though there is no EV_BTN event type. For example:

```
EV_KEY, BTN_RIGHT -> btn:right
EV_KEY, BTN_NORTH -> btn:north
```

Domains

Domains are not something that exists according to the evdev protocol, they are merely a tool invented by evsieve to help you write advanced maps. Domains are strings of text. Any event being processed by evsieve has a domain attached to it. This domain can be specified using the domain= clause on an --input argument, otherwise the domain of an event is set to the path to the input device that emitted said event.

You can specify a domain on a map source or target by adding @domain after the rest of the key. For example, a source of key:a@my-domain means "any event with type EV_KEY, code KEY_A and domain my-domain". If no domain is specified on a source, it will be interpreted as "any domain". If a domain is specified on a target, then the generated event will have that domain. If no domain is specified on a target, the generated event will have the same domain as the source event.

If the part before the @ is empty, then it will be interpreted as "any event with this domain", for example --map @foo @bar will turn any event with domain foo into the same event with domain bar. This is particularly handy for output devices: --output @foo will write all events with domain foo to a virtual device and leave the other events untouched.

The yield flag

It is possible to add the yield flag to an --map or --copy argument, for example:

```
--map yield key:a key:b
```

The position of the yield flag does not matter. All events generated by a --map yield or --copy yield will not be processed by any further maps, copies, hooks, toggles, blocks, or any other kind of argument except for --output. For example, the following arguments will transpose the A and B keys:

```
--map yield key:a key:b \
--map yield key:b key:a \
```

If yield was not provided here, then the second map would turn any B-key events generated by the first map back into A-key events.

If a yield flag is provided to a --copy argument, then source events matching the --copy argument are not yielded, only the generated events are. For example, the following arguments will turn the A key into a B+C key.

```
--copy yield key:a key:b \
--map key:a key:c \
--map key:b key:d \
```

The --block argument**

The --block arguments have the form:

```
--block [SOURCE_EVENT...]
```

The --block argument takes a list of event filters with the same format as the source events for --maps. All events that match one of the filters provided will be removed from the processing stream. For example, the following argument will drop all events related to the A or B keys:

```
--block key:a key:b
```

Using --block is equivalent to using maps that generate zero output events; the above argument would have the same effect as --map key:a --map key:b.

If no source events are specified, the --block argument will drop all events from the processing stream.

The --merge argument**

The --merge arguments have the form:

```
        --merge [SOURCE_EVENT...]
```

--merge is useful if you want to map multiple keys to a single one. For example, consider the following script:

```
evsieve --input /dev/input/by-id/keyboard \
        --map key key:enter \
        --output
```

This will map every key on the keyboard to the Enter key. If the user were to press two keys simultaneously, e.g. the A and B keys, then the output device would receive two KEY_DOWN events on the Enter key, followed up by two KEY_UP events on the Enter key. This by itself is not a big problem: if a key is already down, all subsequent KEY_DOWN events are ignored. However, it can have a counterintuitive result: as soon as one of the two (A, B) keys is released, the Enter key is released, even if the other key is still held down.

The following table shows an example how a serie of keystrokes on the keyboard would affect the visible state of the output keyboard if the above script were to be used:

Input event	State of input keyboard	Output event	State of output keyboard
key:a:1	The A key is down	key:enter:1	The Enter key is down
key:b:1	The A,B keys are down	key:enter:1	The Enter key is down
key:b:0	The A key is down	key:enter:0	The Enter key is up
key:c:1	The A,C keys are down	key:enter:1	The Enter key is down
key:c:0	The A key is down	key:enter:0	The Enter key is up
key:a:0	All keys are up	key:enter:0	The Enter key is up

This is where the --merge argument jumps in. --merge keeps track of how many events of type KEY_DOWN and KEY_UP it has seen for all keys. It drops KEY_DOWN events for keys that are already down, and also drops KEY_UP events until the amount of KEY_UPs it has seen are equal to the amount of KEY_DOWNs it has seen. If we amend the previous script to include a --merge after the --map, the table turns into the following:

```
evsieve --input /dev/input/by-id/keyboard \
        --map key key:enter \
        --merge \
        --output
```

Input event	State of input keyboard	Output event	State of output keyboard
key:a:1	The A key is down	key:enter:1	The Enter key is down
key:b:1	The A,B keys are down	(none)	The Enter key is down
key:b:0	The A key is down	(none)	The Enter key is down
key:c:1	The A,C keys are down	(none)	The Enter key is down
key:c:0	The A key is down	(none)	The Enter key is down
key:a:0	All keys are up	key:enter:0	The Enter key is up

It is possible to specify a filter after the --merge argument to make it apply to only a specific set of events, e.g. --merge key:a will only merge (EV_KEY, KEY_A) events and leave other events untouched. If no filter is specified, --merge will apply to all events of type EV_KEY.

Only events that have the same event code and domain are merged with each other.

Toggles

The --toggle argument has the following basic syntax:

```
--toggle SOURCE_EVENT TARGET_EVENT... [id=ID] [mode=consistent|passive]
```

Toggles work the same way as --maps do, with one difference: a --map will map each source event to all of its target events, whereas a --toggle will map each source events to only one of its target events. The target event it gets mapped to is called the "active target".

The first target listed becomes the active target when the program starts. The active target can be changed using a --hook with a toggle clause. For more information, see the "Toggles" subsection under the "Hooks" section.

An optional id= clause can be specified to give this --toggle a name. This ID can be used to modify specific toggles in the --hook toggle= clause. No two toggles may have the same ID.

Modes

There are two modes of operation for toggles: consistent and passive. The mode of operation can be chosen by supplying a mode= clause to a --toggle argument. If no mode is specified, then "consistent" will be chosen by default.

The easiest mode to understand is "passive". If mode=passive is specified on a toggle, then all source events will be mapped to the currently active target event, no exceptions. However, this mode of operation can introduce some unexpected issues. For example, consider the following argument:

```
--toggle key:a key:b key:c mode=passive \
--hook key:z toggle
```

Suppose the user presses the A key. Then a B key down event will be generated. Now suppose the user presses the Z key before releasing the A key. When the user releases the A key, the A key up event will be mapped to a C key up event, which results in the B key being stuck in a key_down state.

To address this problem, mode=consistent exists. If a toggle operates in consistent mode, then for every key it will remember which target was active when it received a key_down event of that key, and and will then map all events related to that key to that target until a key_up event of that key is received, even if the active target changed in the meanwhile. In the above example, this ensures that an A key up event is mapped to a B key up event event if the active target was changed.

Hooks

The --hook argument has the following basic syntax:

```
--hook KEY... [exec-shell=COMMAND]... [toggle[=[ID][:INDEX]]]...
```

Hooks take actions upon events, but do not modify events. An example of such an action is executing a certain script. The following hook will print "Hello, world!" every time lctrl+A is pressed:

```
--hook key:a key:leftcontrol exec-shell="echo Hello, world!"
```

One or more KEYs can be specified. The syntax for specifying the keys that trigger the hook is the same as the one used to match events for maps, but the semantics are different. The simple explanation of KEYs is that the hook will trigger whenever all those keys are held down simultaneously, and that is probably all you need to remember about them.

In detail: key format

As said, the syntax is the same as the syntax used for maps, but the semantics are different. These semantics are intended to facilitate writing hooks that trigger when a certain combination of keys such as lctrl+A is pressed. First, if no event value is specified for a key, then the value is interpreted to be "1~". Thus, the above example is equivalent to:

```
--hook key:a:1~ key:leftcontrol:1~ exec-shell="echo Hello, world!"
```

Formally said, the hook triggers when (1) an event is received that matches one of the KEYs specified, (2) the last event whose type, code and domain matched that same KEY did not have a value matching that KEY, and (3) for all other KEYs, the last event whose type, code and domain matched those KEYs had a value matching those KEYs.

It is not possible to specify transitions in these KEYs, e.g. --hook key:a:1..0 will throw an error.

Exec-shell

If an exec-shell clause is specified, then a certain command will be executed using the system's default POSIX shell (/bin/sh). Thus, if exec-shell="echo Hello, world!" is specified, the following will be executed:

```
/bin/sh -c "echo Hello, world!"
```

This will be executed as the same user that evsieve is running as, which may be root. If you want to run it as another user, you can make sudo -u $USER part of the command.

The command is executed asynchronously, which means that the processing of events will not halt until the command's execution has completed. In case the command takes sufficiently long to complete and the user presses keys sufficiently fast, it may be possible to trigger the hook again before the command's execution has finished, which will lead to multiple copies of the command being executed simultaneously.

Any processes spawned by exec-shell that are still running when evsieve exits will be sent a SIGTERM signal. This is may change in future versions of evsieve.

Toggles

Hooks are capable of modifying the active target of --toggle arguments specified elsewhere in the script. Any hook can modify any toggle, it doesn't matter whether the --hook or the --toggle argument was specified first, e.g. the following two orders are functionally identical:

```
# Order 1
--hook key:leftctrl toggle \
--toggle @source @target-1 @target-2

# Order 2, functionally identical to Order 1
--toggle @source @target-1 @target-2 \
--hook key:leftctrl toggle
```

If a toggle flag is specified on a hook, then all toggles specified anywhere in the script will have their active target moved to the next one, wrapping around. The above example maps all events with domain source to domain target-1 by default. Once the lctrl key is pressed, they will start mapping the events to target-2 instead. By pressing lctrl again, they will start mapping the events to target-1 again.

It is also possible to specify a toggle clause, using the syntax:

```
--hook toggle=[ID][:TARGET]
```

Both the ID and the TARGET are optional. If an ID is specified, then only the --toggle with the matching ID will be affected. For example, the following hook will affect the first toggle but not the second:

```
--toggle @source-1 @target-1 @target-2 id=first-toggle \
--toggle @source-2 @target-1 @target-2 id=second-toggle \
--hook key:leftctrl toggle=first-toggle
```

By default, the active target is always moved to the next one in the list. It is also possible to move the active target to a specific one by providing a numerical index for TARGET, for example:

```
--toggle @source-1 @target-1 @target-2 id=first-toggle \
--toggle @source-2 @target-1 @target-2 id=second-toggle \
--hook key:leftctrl toggle=first-toggle:1 \
--hook key:rightctrl toggle=:2 \
```

This will move the active target to the first one (@target-1) for first-toggle when lctrl is pressed, and move the active target to the second one (@target-2) for all toggles when rctrl is pressed.

Inputs

The --input argument has the following basic syntax:

```
--input PATH [PATH...] [domain=DOMAIN] [grab[=auto|force]] [persist=reopen|none]
```

At least one path to a device to open is mandatory, everything else is optional. All paths must be represented in absolute form, i.e. starting with a "/" character. It is possible to provide more than one path, in which case multiple devices will be opened with a single argument.

If the domain= clause is provided, then all events read from this input device will have the specified domain attached to them, otherwise the domain of those events shall be equal to the path of said input device. Domains have no intrinsic meaning, but are useful for writing maps. See the "Key format" section under "In detail: Maps" for more information.

Grab modes

Using the grab clause, it is possible to "grab" an input device, by which evsieve will claim exclusive reading access to said device and prevent other programs from reading from that device. This will prevent the X server and similar programs from acting upon the events generated by said device.

Evsieve has two different modes of grabbing a device, auto and force. If merely the grab flag is specified, it will be interpreted as grab=auto. If no grab flag or clause is specified, the input device will not be grabbed.

The two modes differ only in when the input device is grabbed. Under the force mode, evsieve will immediately grab the event device as soon as evsieve starts. Under the auto mode, evsieve will not grab the device until no keys are in the "down" state.

Depending on your use case, there may be an annoying issue with the force mode: if you start evsieve from a terminal by typing in a command or scriptname and then press enter, evsieve will probably grab your device before you release the enter key, causing the terminal/X server to not see you release the enter key and treat it as stuck in the key_down state. The auto mode solves this by not actually grabbing your device until the enter key (and all other keys) are released.

The auto mode is considered the default mode because it has the least potential for surprises. However, if evsieve does refuses to grab your device because some key is permanently stuck in the down state for some reason, you may want to specify grab=force.

There are some questions left surrounding the design of the auto mode, so it is possible that its behaviour will change in future versions of evsieve.

Persistence

The persist= clause tells evsieve what to do in case it somehow fails to read events from input devices, most likely because the USB cable has been disconnected.

For the sake of backwards compatibility, the default mode is persist=none, which tells evsieve to close any device that it fails to read events from and try to continue without said device. If all input devices have been closed, evsieve will exit.

If persist=reopen has been specified, evsieve will instead wait until the input device becomes available again and then try to reopen it. Even if persist=reopen is used, all input devices must be available when evsieve starts.

If the closed and reopened input devices are somehow not identical, evsieve may destroy and recreate some virtual output devices if necessary to ensure all virtual output devices have the correct capabilities.

Outputs

The basic syntax for the --output argument is:

```
    --output [EVENTS...] [create-link=PATH] [name=NAME] [repeat[=enable|disable|passive]]
```

The --output argument creates a virtual event device and sends events to it. If the --output argument is specified multiple times, a different virtual device will be created for each argument.

By default, all events that reach the --output argument will be sent to its virtual device, but it is possible to specify a filter to send only some of the events events that reach it. Any event that get sends to an output device is removed from the evsieve processing stream, e.g. in the following example:

```
evsieve --input /dev/input/by-id/keyboard \
        --output \
        --output
```

All events generated by the --input argument will be written to a virtual device created by the first --output argument. No events will reach the second output argument because all events have already been consumed by the first.

It is possible to create a filter upon which events to send to an output device by specifying the EVENTS part. The syntax used for matching these EVENTS is the same as the one used by the --map argument, see the "Maps" section above. If zero EVENTS are specified, all events will be written to the output device. If one or more EVENTS are specified, only events matching at least one of the EVENTS are written to the output device. For example:

```
evsieve --input /dev/input/by-id/keyboard \
        --output key:a key:b \
        --output
```

This script will write all events relating to the A or B keys to the first output device, and all other events to the second output device.

Symlinks

It is possible to ask evsieve to create a symlink to the device created by an --output by specifying the --create-link= part. For example, --output create-link=/dev/input/by-id/my-virtual-device will create a symlink at /dev/input/by-id/my-virtual-device that points to the actual event device node. The actual event device node will probably have an unpredictable name of /dev/input/event__ where __ is an arbitrary number.

It is customary for links to event devices to reside in /dev/input/by-id/, but this is by no means a requirement. You can create a link anywhere you want (as long as evsieve has write permission to the directory where you put it). Note that putting links in /dev/input/by-id/ requires root privileges on most distributions, so if you want to run evsieve with less privileges, you may want to put your links elsewhere.

Evsieve will create the link when it starts, and try to remove the link when it exits. Note that there are circumstances under which evsieve may be unable to clean up the link it created, such as when evsieve is SIGKILL'd or in case of unexpected power loss. In such cases you may end up with a dangling symlink on your filesystem.

In case a symlink already exists at the path you provided to create-link=, evsieve will overwrite that link. This behaviour has been chosen to not make any scripts involving evsieve mysteriously break after an unexpected power loss.

Names

A name for the device can be specified using the name= clause, e.g.:

```
    --output name="My keyboard"
```

If no name is specified, then Evsieve Virtual Device is chosen by default. The device name is usually of little consequence, but some third-party tools may care about it. For example, the evtest utility is able to display the device name.

Repeats

Some devices, like most keyboards, will send repeat events when a key or button is held down. These are events with type EV_KEY and value 2. Most applications ignore these repeat events and use their own internal logic to detect keys that are held down, but for correctness' sake, evsieve is capable of handling them.

Repeat events are read from input devices if input devices emit them and processed just like any other event. There are three things an output device can do when it receives repeat events:

- If repeat=passive is set on an --output device, then all repeat events that reach this --output argument will be written to this device;
- If repeat=disable is set on an --output device, then all repeat events that reach this --output argument will be dropped;
- If repeat=enable is set on an --output device, then all repeat events that reach this --output argument will be dropped, but the kernel will be asked to automatically generate repeat events for this device.

If no repeat= clause is specified, then repeat=passive will be chosen by default. If a repeat flag is specified without a mode, then repeat=enable is chosen.

Prints

The basic syntax for the --print argument is:

```
    --print [EVENTS...] [format=default|direct]
```

The --print arguments prints all events in the event processing stream to stdout. It does not modify the event processing stream. This is mostly useful for debugging your scripts and for discovering what evsieve calls certain events.

It prints the events the way they are seen by evsieve at the point where the --print argument is placed. For example, the following script will print key:b events when the A key is pressed:

```
evsieve --input /dev/input/by-id/keyboard \
        --map key:a key:b \
        --print
```

The optional EVENTS can be a filter for events that you want to print, similar to the EVENTS in the --output argument. For example, --print @foo will only print events that have domain foo. If no filter is specified, all events are printed.

Formats

The format= clause can specify how the events should be printed, and can choose between format=default and format=direct. If no format= clause is specified, format=default shall be assumed.

The default formatter prints events in a human-readable way, for example:

```
    Event:  type:code = key:a          value = 1 (down)    domain = /dev/input/by-id/keyboard
```

If you specify format=direct, then it will print the events using evsieve's standard key format that's also used for maps and other arguments, for example:

```
    key:a:1@/dev/input/by-id/keyboard
```

Note: --print is intended for human readers, not for scripts. Even if format=<something> is specified, evsieve makes absolutely no guarantees about how the events are printed. Future versions of evsieve may change the format of the printed events without warning. It is not recommended to attempt to programmatically parse the output of evsieve.

License