evdev documentation

Release 1.3.0

Georgi Valkov

Jan 11, 2020

Installation

This package provides bindings to the generic input event interface in Linux. The *evdev* interface serves the purpose of passing events generated in the kernel directly to userspace through character devices that are typically located in `/dev/input/`.

This package also comes with bindings to *uinput*, the userspace input subsystem. *Uinput* allows userspace programs to create and handle input devices that can inject events directly into the input subsystem.

In other words, *python-evdev* allows you to read and write input events on Linux. An event can be a key or button press, a mouse movement or a tap on a touchscreen.

CHAPTER 1

From a binary package

Python-evdev has been packaged for the following GNU/Linux distributions:

Consult the documentation of your OS package manager for installation instructions.

CHAPTER 2

From source

The latest stable version of *python-evdev* can be installed from pypi, provided that you have gcc/clang, pip and the Python and Linux development headers installed on your system. Installing them is distribution specific and typically falls in one of the following categories:

On a Debian compatible OS:

```
$ apt-get install python-dev python-pip gcc
$ apt-get install linux-headers-$(uname -r)
```

On a Redhat compatible OS:

```
$ yum install python-devel python-pip gcc
$ yum install kernel-headers-$(uname -r)
```

On Arch Linux and derivatives:

```
$ pacman -S core/linux-api-headers python-pip gcc
```

Once all dependencies are available, you may install *python-evdev* using pip:

```
$ sudo pip install evdev    # available globally
$ pip install --user evdev  # available to the current user
```

CHAPTER 3

Specifying header locations

By default, the setup script will look for the `input.h` and `input-event-codes.h`[1] header files `/usr/include/linux`.

You may use the `--evdev-headers` option to the `build_ext` setuptools command to specify the location of these header files. It accepts one or more colon-separated paths. For example:

```
$ python setup.py build_ext \
    --evdev-headers buildroot/input.h:buildroot/input-event-codes.h \
    --include-dirs  buildroot/ \
    install  # or any other command (e.g. develop, bdist, bdist_wheel)
```

[1] `input-event-codes.h` is found only in more recent kernel versions.

CHAPTER 4

Quick Start

4.1 Listing accessible event devices

```
>>> import evdev

>>> devices = [evdev.InputDevice(path) for path in evdev.list_devices()]
>>> for device in devices:
...     print(device.path, device.name, device.phys)
/dev/input/event1    USB Keyboard        usb-0000:00:12.1-2/input0
/dev/input/event0    USB Optical Mouse   usb-0000:00:12.0-2/input0
```

Note: If you do not see any devices, ensure that your user is in the correct group (typically `input`) to have read/write access.

4.2 Reading events from a device

```
>>> import evdev

>>> device = evdev.InputDevice('/dev/input/event1')
>>> print(device)
device /dev/input/event1, name "USB Keyboard", phys "usb-0000:00:12.1-2/input0"

>>> for event in device.read_loop():
...     if event.type == evdev.ecodes.EV_KEY:
...         print(evdev.categorize(event))
... # pressing 'a' and holding 'space'
key event at 1337016188.396030, 30 (KEY_A), down
key event at 1337016188.492033, 30 (KEY_A), up
key event at 1337016189.772129, 57 (KEY_SPACE), down
```

(continues on next page)

(continued from previous page)

```
key event at 1337016190.275396, 57 (KEY_SPACE), hold
key event at 1337016190.284160, 57 (KEY_SPACE), up
```

4.3 Accessing event codes

The `evdev.ecodes` module provides reverse and forward mappings between the names and values of the event subsystem constants.

```
>>> from evdev import ecodes

>>> ecodes.KEY_A
... 30
>>> ecodes.ecodes['KEY_A']
... 30
>>> ecodes.KEY[30]
... 'KEY_A'
>>> ecodes.bytype[ecodes.EV_KEY][30]
... 'KEY_A'

# A single value may correspond to multiple event codes.
>>> ecodes.KEY[152]
... ['KEY_COFFEE', 'KEY_SCREENLOCK']
```

4.4 Listing and monitoring input devices

The *python-evdev* package also comes with a small command-line program for listing and monitoring input devices:

```
$ python -m evdev.evtest
```

CHAPTER 5

Tutorial

5.1 Listing accessible event devices

```
>>> import evdev

>>> devices = [evdev.InputDevice(path) for path in evdev.list_devices()]
>>> for device in devices:
>>>     print(device.path, device.name, device.phys)
/dev/input/event1    Dell Dell USB Keyboard    usb-0000:00:12.1-2/input0
/dev/input/event0    Dell USB Optical Mouse    usb-0000:00:12.0-2/input0
```

5.2 Listing device capabilities

```
>>> import evdev

>>> device = evdev.InputDevice('/dev/input/event0')
>>> print(device)
device /dev/input/event0, name "Dell USB Optical Mouse", phys "usb-0000:00:12.0-2/
↪input0"

>>> device.capabilities()
... { 0: [0, 1, 2], 1: [272, 273, 274, 275], 2: [0, 1, 6, 8], 4: [4] }

>>> device.capabilities(verbose=True)
... { ('EV_SYN', 0): [('SYN_REPORT', 0), ('SYN_CONFIG', 1), ('SYN_MT_REPORT', 2)],
...   ('EV_KEY', 1): [('BTN_MOUSE', 272), ('BTN_RIGHT', 273), ('BTN_MIDDLE', 274), (
↪'BTN_SIDE', 275)], ...
```

5.3 Listing device capabilities (devices with absolute axes)

```
>>> import evdev

>>> device = evdev.InputDevice('/dev/input/event7')
>>> print(device)
device /dev/input/event7, name "Wacom Bamboo 2FG 4x5 Finger", phys ""

>>> device.capabilities()
... { 1: [272, 273, 277, 278, 325, 330, 333] ,
...   3: [(0, AbsInfo(min=0, max=15360, fuzz=128, flat=0)),
...       (1, AbsInfo(min=0, max=10240, fuzz=128, flat=0))] }

>>> device.capabilities(verbose=True)
... { ('EV_KEY', 1): [('BTN_MOUSE', 272), ('BTN_RIGHT', 273), ...],
...   ('EV_ABS', 3): [(('ABS_X', 0), AbsInfo(min=0, max=15360, fuzz=128, flat=0)),
...                   (('ABS_Y', 1), AbsInfo(min=0, max=10240, fuzz=128, flat=0)),] }

>>> device.capabilities(absinfo=False)
... { 1: [272, 273, 277, 278, 325, 330, 333],
...   3: [0, 1, 47, 53, 54, 57] }
```

5.4 Getting and setting LED states

```
>>> dev.leds(verbose=True)
... [('LED_NUML', 0), ('LED_CAPSL', 1)]

>>> dev.leds()
... [0, 1]

>>> dev.set_led(ecodes.LED_NUML, 1)  # enable numlock
>>> dev.set_led(ecodes.LED_NUML, 0)  # disable numlock
```

5.5 Getting currently active keys

```
>>> dev.active_keys(verbose=True)
... [('KEY_3', 4), ('KEY_LEFTSHIFT', 42)]

>>> dev.active_keys()
... [4, 42]
```

5.6 Reading events

Reading events from a single device in an endless loop.

```
>>> from evdev import InputDevice, categorize, ecodes
>>> dev = InputDevice('/dev/input/event1')

>>> print(dev)
```

(continues on next page)

(continued from previous page)

```
device /dev/input/event1, name "Dell Dell USB Keyboard", phys "usb-0000:00:12.1-2/
↪input0"

>>> for event in dev.read_loop():
...     if event.type == ecodes.EV_KEY:
...         print(categorize(event))
... # pressing 'a' and holding 'space'
key event at 1337016188.396030, 30 (KEY_A), down
key event at 1337016188.492033, 30 (KEY_A), up
key event at 1337016189.772129, 57 (KEY_SPACE), down
key event at 1337016190.275396, 57 (KEY_SPACE), hold
key event at 1337016190.284160, 57 (KEY_SPACE), up
```

5.7 Reading events (using `asyncio`)

Note: This requires Python 3.5+ for the async/await keywords.

```
>>> import asyncio
>>> from evdev import InputDevice, categorize, ecodes

>>> dev = InputDevice('/dev/input/event1')

>>> async def helper(dev):
...     async for ev in dev.async_read_loop():
...         print(repr(ev))

>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(helper(dev))
InputEvent(1527363738, 348740, 4, 4, 458792)
InputEvent(1527363738, 348740, 1, 28, 0)
InputEvent(1527363738, 348740, 0, 0, 0)
```

5.8 Reading events from multiple devices (using `select`)

```
>>> from evdev import InputDevice
>>> from select import select

# A mapping of file descriptors (integers) to InputDevice instances.
>>> devices = map(InputDevice, ('/dev/input/event1', '/dev/input/event2'))
>>> devices = {dev.fd: dev for dev in devices}

>>> for dev in devices.values(): print(dev)
device /dev/input/event1, name "Dell Dell USB Keyboard", phys "usb-0000:00:12.1-2/
↪input0"
device /dev/input/event2, name "Logitech USB Laser Mouse", phys "usb-0000:00:12.0-2/
↪input0"

>>> while True:
...     r, w, x = select(devices, [], [])
```

(continues on next page)

(continued from previous page)

```
...     for fd in r:
...         for event in devices[fd].read():
...             print(event)
event at 1351116708.002230, code 01, type 02, val 01
event at 1351116708.002234, code 00, type 00, val 00
event at 1351116708.782231, code 04, type 04, val 458782
event at 1351116708.782237, code 02, type 01, val 01
```

5.9 Reading events from multiple devices (using `selectors`)

This can also be achieved using the `selectors` module in Python 3.4:

```
from evdev import InputDevice
from selectors import DefaultSelector, EVENT_READ

selector = selectors.DefaultSelector()

mouse = evdev.InputDevice('/dev/input/event1')
keybd = evdev.InputDevice('/dev/input/event2')

# This works because InputDevice has a `fileno()` method.
selector.register(mouse, selectors.EVENT_READ)
selector.register(keybd, selectors.EVENT_READ)

while True:
    for key, mask in selector.select():
        device = key.fileobj
        for event in device.read():
            print(event)
```

5.10 Reading events from multiple devices (using `asyncio`)

Yet another possibility is the `asyncio` module from Python 3.4:

```
import asyncio, evdev

@asyncio.coroutine
def print_events(device):
    while True:
        events = yield from device.async_read()
        for event in events:
            print(device.path, evdev.categorize(event), sep=': ')

mouse = evdev.InputDevice('/dev/input/eventX')
keybd = evdev.InputDevice('/dev/input/eventY')

for device in mouse, keybd:
    asyncio.async(print_events(device))

loop = asyncio.get_event_loop()
loop.run_forever()
```

Since Python 3.5, the async/await syntax makes this even simpler:

```
import asyncio, evdev

mouse = evdev.InputDevice('/dev/input/event4')
keybd = evdev.InputDevice('/dev/input/event5')

async def print_events(device):
    async for event in device.async_read_loop():
        print(device.path, evdev.categorize(event), sep=': ')

for device in mouse, keybd:
    asyncio.ensure_future(print_events(device))

loop = asyncio.get_event_loop()
loop.run_forever()
```

5.11 Accessing evdev constants

```
>>> from evdev import ecodes

>>> ecodes.KEY_A, ecodes.ecodes['KEY_A']
... (30, 30)
>>> ecodes.KEY[30]
... 'KEY_A'
>>> ecodes.bytype[ecodes.EV_KEY][30]
... 'KEY_A'
>>> ecodes.KEY[152]  # a single value may correspond to multiple codes
... ['KEY_COFFEE', 'KEY_SCREENLOCK']
```

5.12 Getting exclusive access to a device

```
>>> dev.grab()  # become the sole recipient of all incoming input events
>>> dev.ungrab()
```

This functionality is also available as a context manager.

```
>>> with dev.grab_context():
...     pass
```

5.13 Associating classes with event types

```
>>> from evdev import categorize, event_factory, ecodes

>>> class SynEvent(object):
...     def __init__(self, event):
...         ...

>>> event_factory[ecodes.EV_SYN] = SynEvent
```

See *events* for more information.

5.14 Injecting input

```
>>> from evdev import UInput, ecodes as e

>>> ui = UInput()

>>> # accepts only KEY_* events by default
>>> ui.write(e.EV_KEY, e.KEY_A, 1)  # KEY_A down
>>> ui.write(e.EV_KEY, e.KEY_A, 0)  # KEY_A up
>>> ui.syn()

>>> ui.close()
```

5.15 Injecting events (using a context manager)

```
>>> ev = InputEvent(1334414993, 274296, ecodes.EV_KEY, ecodes.KEY_A, 1)
>>> with UInput() as ui:
...     ui.write_event(ev)
...     ui.syn()
```

5.16 Specifying `uinput` device options

```
>>> from evdev import UInput, AbsInfo, ecodes as e

>>> cap = {
...     e.EV_KEY : [e.KEY_A, e.KEY_B],
...     e.EV_ABS : [
...         (e.ABS_X, AbsInfo(value=0, min=0, max=255,
...                           fuzz=0, flat=0, resolution=0)),
...         (e.ABS_Y, AbsInfo(0, 0, 255, 0, 0, 0)),
...         (e.ABS_MT_POSITION_X, (0, 128, 255, 0)) ]
... }

>>> ui = UInput(cap, name='example-device', version=0x3)
>>> print(ui)
name "example-device", bus "BUS_USB", vendor "0001", product "0001", version "0003"
event types: EV_KEY EV_ABS EV_SYN

>>> print(ui.capabilities())
{0: [0, 1, 3],
 1: [30, 48],
 3: [(0,  AbsInfo(value=0, min=0, max=0,   fuzz=255, flat=0, resolution=0)),
     (1,  AbsInfo(value=0, min=0, max=0,   fuzz=255, flat=0, resolution=0)),
     (53, AbsInfo(value=0, min=0, max=255, fuzz=128, flat=0, resolution=0))]}

>>> # move mouse cursor
>>> ui.write(e.EV_ABS, e.ABS_X, 20)
>>> ui.write(e.EV_ABS, e.ABS_Y, 20)
>>> ui.syn()
```

5.17 Create `uinput` device with capabilities of another device

```
>>> from evdev import UInput, InputDevice

>>> mouse = InputDevice('/dev/input/event1')
>>> keybd = '/dev/input/event2'

>>> ui = UInput.from_device(mouse, keybd, name='keyboard-mouse-device')
>>> ui.capabilities(verbose=True).keys()
dict_keys([('EV_LED', 17), ('EV_KEY', 1), ('EV_SYN', 0), ('EV_REL', 2), ('EV_MSC',␣
↪4)])
```

5.18 Create `uinput` device capable of receiving FF-effects

```
import asyncio
from evdev import UInput, categorize, ecodes

cap = {
   ecodes.EV_FF:  [ecodes.FF_RUMBLE ],
   ecodes.EV_KEY: [ecodes.KEY_A, ecodes.KEY_B]
}

ui = UInput(cap, name='test-controller', version=0x3)

async def print_events(device):
    async for event in device.async_read_loop():
        print(categorize(event))

        # Wait for an EV_UINPUT event that will signal us that an
        # effect upload/erase operation is in progress.
        if event.type != ecodes.EV_UINPUT:
            pass

        if event.code == ecodes.UI_FF_UPLOAD:
            upload = device.begin_upload(event.value)
            upload.retval = 0

            print(f'[upload] effect_id: {upload.effect_id}, type: {upload.effect.type}
↪')
            device.end_upload(upload)

        elif event.code == ecodes.UI_FF_ERASE:
            erase = device.begin_erase(event.value)
            print(f'[erase] effect_id {erase.effect_id}')

            erase.retval = 0
            device.end_erase(erase)

asyncio.ensure_future(print_events(ui))
loop = asyncio.get_event_loop()
loop.run_forever()
```

5.19 Injecting an FF-event into first FF-capable device found

```
from evdev import ecodes, InputDevice, ff

# Find first EV_FF capable event device (that we have permissions to use).
for name in evdev.list_devices():
    dev = InputDevice(name)
    if ecodes.EV_FF in dev.capabilities():
        break

rumble = ff.Rumble(strong_magnitude=0x0000, weak_magnitude=0xffff)
effect_type = ff.EffectType(ff_rumble_effect=rumble)
duration_ms = 1000

effect = ff.Effect(
    ecodes.FF_RUMBLE, -1, 0,
    ff.Trigger(0, 0),
    ff.Replay(duration_ms, 0),
    ff.EffectType(ff_rumble_effect=rumble)
)

repeat_count = 1
effect_id = dev.upload_effect(effect)
dev.write(e.EV_FF, effect_id, repeat_count)
dev.erase_effect(effect_id)
```

CHAPTER 6

API Reference

6.1 events

This module provides the *InputEvent* class, which closely resembles the `input_event` struct defined in `linux/input.h`:

```
struct input_event {
    struct timeval time;
    __u16 type;
    __u16 code;
    __s32 value;
};
```

This module also defines several *InputEvent* sub-classes that know more about the different types of events (key, abs, rel etc). The *event_factory* dictionary maps event types to these classes.

Assuming you use the *evdev.util.categorize()* function to categorize events according to their type, adding or replacing a class for a specific event type becomes a matter of modifying *event_factory*.

All classes in this module have reasonable `str()` and `repr()` methods:

```
>>> print(event)
event at 1337197425.477827, code 04, type 04, val 458792
>>> print(repr(event))
InputEvent(1337197425L, 477827L, 4, 4, 458792L)

>>> print(key_event)
key event at 1337197425.477835, 28 (KEY_ENTER), up
>>> print(repr(key_event))
KeyEvent(InputEvent(1337197425L, 477835L, 1, 28, 0L))
```

class evdev.events.**InputEvent**(*sec*, *usec*, *type*, *code*, *value*)

A generic input event.

sec
Time in seconds since epoch at which event occurred.

usec
Microsecond portion of the timestamp.

type
Event type - one of `ecodes.EV_*`.

code
Event code related to the event type.

value
Event value related to the event type.

timestamp()
Return event timestamp as a float.

class evdev.events.**KeyEvent**(*event*, *allow_unknown=False*)
An event generated by a keyboard, button or other key-like devices.

key_up = 0

key_down = 1

key_hold = 2

scancode

keystate

keycode

event
Reference to an *InputEvent* instance.

class evdev.events.**RelEvent**(*event*)
A relative axis event (e.g moving the mouse 5 units to the left).

event
Reference to an *InputEvent* instance.

class evdev.events.**AbsEvent**(*event*)
An absolute axis event (e.g the coordinates of a tap on a touchscreen).

event
Reference to an *InputEvent* instance.

class evdev.events.**SynEvent**(*event*)
A synchronization event. Synchronization events are used as markers to separate event. Used as markers to separate events. Events may be separated in time or in space, such as with the multitouch protocol.

event
Reference to an *InputEvent* instance.

evdev.events.**event_factory = {0: <class 'evdev.events.SynEvent'>, 1: <class 'evdev.events**
A mapping of event types to *InputEvent* sub-classes. Used by *evdev.util.categorize()*

6.2 eventio

class evdev.eventio.**EventIO**
Base class for reading and writing input events.

This class is used by `InputDevice` and `UInput`.

- On, `InputDevice` it used for reading user-generated events (e.g. key presses, mouse movements) and writing feedback events (e.g. leds, beeps).
- On, `UInput` it used for writing user-generated events (e.g. key presses, mouse movements) and reading feedback events (e.g. leds, beeps).

fileno()
Return the file descriptor to the open event device. This makes it possible to pass instances directly to `select.select()` and `asyncore.file_dispatcher`.

read_loop()
Enter an endless `select.select()` loop that yields input events.

read_one()
Read and return a single input event as an instance of *InputEvent*.

Return `None` if there are no pending input events.

read()
Read multiple input events from device. Return a generator object that yields *InputEvent* instances. Raises *BlockingIOError* if there are no available events at the moment.

need_write()
Decorator that raises `EvdevError` if there is no write access to the input device.

write_event(*event*)
Inject an input event into the input subsystem. Events are queued until a synchronization event is received.

Parameters **event** (`InputEvent`) – InputEvent instance or an object with an `event` attribute (*KeyEvent*, *RelEvent* etc).

Example

```
>>> ev = InputEvent(1334414993, 274296, ecodes.EV_KEY, ecodes.KEY_A, 1)
>>> ui.write_event(ev)
```

write(*etype*, *code*, *value*)
Inject an input event into the input subsystem. Events are queued until a synchronization event is received.

Parameters

- **etype** – event type (e.g. `EV_KEY`).
- **code** – event code (e.g. `KEY_A`).
- **value** – event value (e.g. 0 1 2 - depends on event type).

Example

```
>>> ui.write(e.EV_KEY, e.KEY_A, 1) # key A - down
>>> ui.write(e.EV_KEY, e.KEY_A, 0) # key A - up
```

close()

__weakref__
list of weak references to the object (if defined)

6.3 eventio_async

class evdev.eventio_async.**EventIO**

async_read_one()
Asyncio coroutine to read and return a single input event as an instance of *InputEvent*.

async_read()
Asyncio coroutine to read multiple input events from device. Return a generator object that yields *InputEvent* instances.

async_read_loop()
Return an iterator that yields input events. This iterator is compatible with the `async for` syntax.

close()

6.4 device

class evdev.device.**AbsInfo**
Absolute axis information.

A `namedtuple` used for storing absolute axis information - corresponds to the `input_absinfo` struct:

value
Latest reported value for the axis.

min
Specifies minimum value for the axis.

max
Specifies maximum value for the axis.

fuzz
Specifies fuzz value that is used to filter noise from the event stream.

flat
Values that are within this value will be discarded by joydev interface and reported as 0 instead.

resolution
Specifies resolution for the values reported for the axis. Resolution for main axes (`ABS_X, ABS_Y, ABS_Z`) is reported in units per millimeter (units/mm), resolution for rotational axes (`ABS_RX, ABS_RY, ABS_RZ`) is reported in units per radian.

Note: The input core does not clamp reported values to the `[minimum, maximum]` limits, such task is left to userspace.

---

class evdev.device.**KbdInfo**
Keyboard repeat rate.

repeat
Keyboard repeat rate in characters per second.

delay
Amount of time that a key must be depressed before it will start to repeat (in milliseconds).

class evdev.device.**DeviceInfo**

bustype

vendor

product

version

class evdev.device.**InputDevice**(*dev*)

A linux input device from which input events can be read.

__init__(*dev*)

Parameters **dev** (*str|bytes|PathLike*) – Path to input device

path

Path to input device.

fd

A non-blocking file descriptor to the device file.

info

A *DeviceInfo* instance.

name

The name of the event device.

phys

The physical topology of the device.

uniq

The unique address of the device.

version

The evdev protocol version.

ff_effects_count

The number of force feedback effects the device can keep in its memory.

capabilities(*verbose=False*, *absinfo=True*)

Return the event types that this device supports as a mapping of supported event types to lists of handled event codes.

Example

```
>>> device.capabilities()
{ 1: [272, 273, 274],
  2: [0, 1, 6, 8] }
```

If `verbose` is `True`, event codes and types will be resolved to their names.

```
{ ('EV_KEY', 1): [('BTN_MOUSE', 272),
                  ('BTN_RIGHT', 273),
                  ('BTN_MIDDLE', 273)],
  ('EV_REL', 2): [('REL_X', 0),
                  ('REL_Y', 1),
                  ('REL_HWHEEL', 6),
                  ('REL_WHEEL', 8)] }
```

Unknown codes or types will be resolved to `'?'`.

If `absinfo` is `True`, the list of capabilities will also include absolute axis information in the form of *AbsInfo* instances:

```
{ 3: [ (0, AbsInfo(min=0, max=255, fuzz=0, flat=0)),
       (1, AbsInfo(min=0, max=255, fuzz=0, flat=0)) ]}
```

Combined with `verbose` the above becomes:

```
{ ('EV_ABS', 3): [ (('ABS_X', 0), AbsInfo(min=0, max=255, fuzz=0, flat=0)),
                   (('ABS_Y', 1), AbsInfo(min=0, max=255, fuzz=0, flat=0)) ]}
```

input_props(*verbose=False*)
Get device properties and quirks.

Example

```
>>> device.input_props()
[0, 5]
```

If `verbose` is `True`, input properties are resolved to their names. Unknown codes are resolved to `'?'`:

```
[('INPUT_PROP_POINTER', 0), ('INPUT_PROP_POINTING_STICK', 5)]
```

leds(*verbose=False*)
Return currently set LED keys.

Example

```
>>> device.leds()
[0, 1, 8, 9]
```

If `verbose` is `True`, event codes are resolved to their names. Unknown codes are resolved to `'?'`:

```
[('LED_NUML', 0), ('LED_CAPSL', 1), ('LED_MISC', 8), ('LED_MAIL', 9)]
```

set_led(*led_num*, *value*)
Set the state of the selected LED.

Example

```
>>> device.set_led(ecodes.LED_NUML, 1)
```

__eq__(*other*)
Two devices are equal if their *info* attributes are equal.

__ne__(*other*)
Return self!=value.

__fspath__()

close()

grab()
Grab input device using `EVIOCGRAB` - other applications will be unable to receive events until the device is released. Only one process can hold a `EVIOCGRAB` on a device.

> **Warning:** Grabbing an already grabbed device will raise an `IOError`.

ungrab()
Release device if it has been already grabbed (uses *EVIOCGRAB*).

> **Warning:** Releasing an already released device will raise an `IOError('Invalid argument')`.

grab_context()
A context manager for the duration of which only the current process will be able to receive events from the device.

upload_effect(*effect*)
Upload a force feedback effect to a force feedback device.

erase_effect(*ff_id*)
Erase a force effect from a force feedback device. This also stops the effect.

repeat
Get or set the keyboard repeat rate (in characters per minute) and delay (in milliseconds).

active_keys(*verbose=False*)
Return currently active keys.

Example

```
>>> device.active_keys()
[1, 42]
```

If `verbose` is `True`, key codes are resolved to their verbose names. Unknown codes are resolved to `'?'`. For example:

```
[('KEY_ESC', 1), ('KEY_LEFTSHIFT', 42)]
```

fn

absinfo(*axis_num*)
Return current *AbsInfo* for input device axis

Parameters **axis_num** (*int*) – EV_ABS keycode (example `ecodes.ABS_X`)

Example

```
>>> device.absinfo(ecodes.ABS_X)
AbsInfo(value=1501, min=-32768, max=32767, fuzz=0, flat=128, resolution=0)
```

__hash__ = None

set_absinfo(*axis_num*, *value=None*, *min=None*, *max=None*, *fuzz=None*, *flat=None*, *resolution=None*)
Update *AbsInfo* values. Only specified values will be overwritten.

Parameters **axis_num** (*int*) – EV_ABS keycode (example `ecodes.ABS_X`)

Example

```
>>> device.set_absinfo(ecodes.ABS_X, min=-2000, max=2000)
```

You can also unpack AbsInfo tuple that will overwrite all values

```
>>> device.set_absinfo(ecodes.ABS_Y, *AbsInfo(0, -2000, 2000, 0, 15, 0))
```

6.5 uinput

class evdev.uinput.**UInput**(*events=None*, *name='py-evdev-uinput'*, *vendor=1*, *product=1*, *version=1*, *bustype=3*, *devnode='/dev/uinput'*, *phys='py-evdev-uinput'*, *input_props=None*)

A userland input device and that can inject input events into the linux input subsystem.

classmethod from_device(**devices*, ***kwargs*)

Create an UInput device with the capabilities of one or more input devices.

Parameters

- **devices** (*InputDevice|str*) – Varargs of InputDevice instances or paths to input devices.
- **filtered_types** (*Tuple[event type codes]*) – Event types to exclude from the capabilities of the uinput device.
- ****kwargs** – Keyword arguments to UInput constructor (i.e. name, vendor etc.).

__init__(*events=None*, *name='py-evdev-uinput'*, *vendor=1*, *product=1*, *version=1*, *bustype=3*, *devnode='/dev/uinput'*, *phys='py-evdev-uinput'*, *input_props=None*)

Parameters

- **events** (*dict*) – Dictionary of event types mapping to lists of event codes. The event types and codes that the uinput device will be able to inject - defaults to all key codes.
- **name** – The name of the input device.
- **vendor** – Vendor identifier.
- **product** – Product identifier.
- **version** – Version identifier.
- **bustype** – Bustype identifier.
- **phys** – Physical path.
- **input_props** – Input properties and quirks.

Note: If you do not specify any events, the uinput device will be able to inject only KEY_* and BTN_* event codes.

---

name

Uinput device name.

vendor

Device vendor identifier.

product
Device product identifier.

version
Device version identifier.

bustype
Device bustype - e.g. `BUS_USB`.

phys = None
Uinput device physical path.

devnode
Uinput device node - e.g. `/dev/uinput/`.

fd
Write-only, non-blocking file descriptor to the uinput device node.

device
An *InputDevice* instance for the fake input device. `None` if the device cannot be opened for reading and writing.

syn()
Inject a `SYN_REPORT` event into the input subsystem. Events queued by `write()` will be fired. If possible, events will be merged into an 'atomic' event.

capabilities(*verbose=False*, *absinfo=True*)
See *capabilities*.

6.6 util

evdev.util.**list_devices**(*input_device_dir='/dev/input'*)
List readable character devices in `input_device_dir`.

evdev.util.**is_device**(*fn*)
Check if `fn` is a readable and writable character device.

evdev.util.**categorize**(*event*)
Categorize an event according to its type.

The *event_factory* dictionary maps event types to sub-classes of *InputEvent*. If the event cannot be categorized, it is returned unmodified.

evdev.util.**resolve_ecodes_dict**(*typecodemap*, *unknown='?'*)
Resolve event codes and types to their verbose names.

Parameters

- **typecodemap** – mapping of event types to lists of event codes.
- **unknown** – symbol to which unknown types or codes will be resolved.

Example

```
>>> resolve_ecodes_dict({ 1: [272, 273, 274] })
{ ('EV_KEY', 1): [('BTN_MOUSE',  272),
                  ('BTN_RIGHT',  273),
                  ('BTN_MIDDLE', 274)] }
```

If `typecodemap` contains absolute axis info (instances of *AbsInfo*) the result would look like:

```
>>> resolve_ecodes_dict({ 3: [(0, AbsInfo(...))] })
{ ('EV_ABS', 3L): [(('ABS_X', 0L), AbsInfo(...))] }
```

evdev.util.**resolve_ecodes**(*ecode_dict*, *ecode_list*, *unknown='?'*)

Resolve event codes and types to their verbose names.

Example

```
>>> resolve_ecodes([272, 273, 274])
[('BTN_MOUSE',  272), ('BTN_RIGHT',  273), ('BTN_MIDDLE', 274)]
```

6.7 ecodes

This modules exposes the integer constants defined in `linux/input.h` and `linux/input-event-codes.h`.

Exposed constants:

```
KEY, ABS, REL, SW, MSC, LED, BTN, REP, SND, ID, EV,
BUS, SYN, FF, FF_STATUS, INPUT_PROP
```

This module also provides reverse and forward mappings of the names and values of the above mentioned constants:

```
>>> evdev.ecodes.KEY_A
30

>>> evdev.ecodes.ecodes['KEY_A']
30

>>> evdev.ecodes.KEY[30]
'KEY_A'

>>> evdev.ecodes.REL[0]
'REL_X'

>>> evdev.ecodes.EV[evdev.ecodes.EV_KEY]
'EV_KEY'

>>> evdev.ecodes.bytype[evdev.ecodes.EV_REL][0]
'REL_X'
```

Keep in mind that values in reverse mappings may point to one or more event codes. For example:

```
>>> evdev.ecodes.FF[80]
['FF_EFFECT_MIN', 'FF_RUMBLE']

>>> evdev.ecodes.FF[81]
'FF_PERIODIC'
```

evdev.ecodes.**keys {0: 'KEY_RESERVED', 1: 'KEY_ESC', 2: 'KEY_1', ...}**

Keys are a combination of all BTN and KEY codes.

evdev.ecodes.**ecodes {'KEY_END': 107, 'FF_RUMBLE': 80, 'KEY_KPDOT': 83, 'KEY_CNT': 768, ...}**

Mapping of names to values.

evdev.ecodes.**bytype** {0: {0: 'SYN_REPORT', 1: 'SYN_CONFIG', 2: 'SYN_MT_REPORT', 3: 'SYN

Mapping of event types to other value/name mappings.

CHAPTER 7

Scope and status

Python-evdev exposes most of the more common interfaces defined in the evdev subsystem. Reading and injecting events is well supported and has been tested with nearly all event types.

The basic functionality for reading and uploading force-feedback events is there, but it has not been exercised sufficiently. A major shortcoming of the uinput wrapper is that it does not support force-feedback devices at all (see issue #23).

Some characters, such as `:` (colon), cannot be easily injected (see issue #7), Translating them into UInput events would require knowing the kernel keyboard translation table, which is beyond the scope of python-evdev. Please look into the following projects if you need more complete or convenient input injection support.

- python-uinput
- uinput-mapper
- PyUserInput (cross-platform, works on the display server level)
- pygame (cross-platform)

CHAPTER 8

Changelog

8.1 1.3.0 (Jan 12, 2020)

- Fix build on 32bit arches with 64bit time_t
- Add functionality to query device properties. See `InputDevice.input_props` and the `input_props` argument to `Uinput`.
- `KeyEvent` received an `allow_unknown` constructor argument, which determines what will happen when an event code cannot be mapped to a keycode. The default and behavior so far has been to raise `KeyError`. If set to `True`, the keycode will be set to the event code formatted as a hex number.
- Add `InputDevice.set_absinfo()` and `InputDevice.absinfo()`.
- Instruct the asyncio event loop to stop monitoring the fd of the input device when the device is closed.

8.2 1.2.0 (Apr 7, 2019)

- Add UInput support for the resolution parameter in AbsInfo. This brings support for the new method of uinput device setup, which was introduced in Linux 4.5 (thanks to @LinusCDE).
- Vendor and product identifiers can be greater or equal to *0x8000* (thanks @ivaradi).

8.3 1.1.2 (Sep 1, 2018)

- Fix installation on kernels <= 4.4.
- Fix uinput creation ignoring absinfo settings.

8.4 1.1.0 (Aug 27, 2018)

- Add support for handling force-feedback effect uploads (many thanks to @ndreys).
- Fix typo preventing ff effects that need left coefficients from working.

8.5 1.0.0 (Jun 02, 2018)

- Prevent `Uinput` device creation raising `Objects/longobject.c:415: bad argument to internal function` when a non-complete `AbsInfo` structure is passed. All missing `AbsInfo` fields are set to 0.
- Fix `Uinput` device creation raising `KeyError` when a capability filtered by default is not present.
- The `InputDevice.fn` attribute was deprecated in favor of `InputDevice.path`. Using the former will show a `DeprecationWarning`, but would otherwise continue working as before.
- Fix `InputDevice` comparison raising `AttributeError` due to a non-existant `path` attribute.
- Fix asyncio support in Python 3.5+.
- Uploading FF effect now works both on Python 2.7 and Python 3+.
- Remove the `asyncore` example from the tutorial.

8.6 0.8.1 (Mar 24, 2018)

- Fix Python 2 compatibility issue in with `Uinput.from_device`.
- Fix minor *evdev.evtest* formatting issue.

8.7 0.8.0 (Mar 22, 2018)

- Fix `InputDevice` comparison on Python 2.
- The device path is now considered when comparing two devices.
- Fix `UInput.from_device` not correctly merging the capabilities of selected devices.
- The list of excluded event types in `UInput.from_device` is now configurable. For example:

```
UInput.from_device(dev, filtered_types=(EV_SYN, EV_FF))
```

  In addition, `ecodes.EV_FF` is now excluded by default.

- Add a context manager for grabbing access to a device - `InputDevice.grab_context`. For example:

```
with dev.grab_context():
    pass
```

- Add the `InputDevice.uniq` attribute, which contains the unique identifier of the device. As with `phys`, this attribute may be empty (i.e. ‘’).

8.8 0.7.0 (Jun 16, 2017)

- `InputDevice` now accepts objects that support the path protocol. For example:

```
pth = pathlib.Path('/dev/input/event0')
dev = evdev.InputDevice(pth)
```

- Support path protocol in `InputDevice`. This means that `InputDevice` instances can be passed to callers that expect a `os.PathLike` object.
- Exceptions raised during `InputDevice.async_read()` (and similar) are now handled properly (i.e. an exception is set on the returned future instead of leaking that exception into the event loop) (Fixes #67).

8.9 0.6.4 (Oct 07, 2016)

- Exclude `ecodes.c` from source distribution (Fixes #63).

8.10 0.6.3 (Oct 06, 2016)

- Add the `UInput.from_device` class method, which allows uinput device to be created with the capabiltiies of one or more existing input devices:

```
ui = UInput.from_device('/dev/input1', '/dev/input2', **constructor_kwargs)
```

- Add the `build_ecodes` distutils command, which generates the `ecodes.c` extension module. The new way of overwriting the evdev header locations is:

```
python setup.py build \
  build_ecodes --evdev-headers path/input.h:path/input-event-codes.h \
  build_ext --include-dirs  path/ \
  install
```

  The `build*` and `install` commands no longer have to be part of the same command-line (i.e. running `install` will reuse the outputs of the last `build`).

8.11 0.6.1 (Jun 04, 2016)

- Disable tty echoing while evtest is running.
- Allow evtest to listen to more than one devices.
- The setup.py script now allows the location of the input header files to be overwritten. For example:

```
python setup.py build_ext \
  --evdev-headers path/input.h:path/input-event-codes.h \
  --include-dirs  path/ \
  install
```

8.12 0.6.0 (Feb 14, 2016)

- Asyncio and async/await support (many thanks to @paulo-raca).
- Add the ability to set the *phys* property of uinput devices (thanks @paulo-raca).
- Add a generic `InputDevice.set()` method (thanks @paulo-raca).
- Distribute the evtest script along with evdev.
- Fix issue with generating `ecodes.c` in recent kernels (`>= 4.4.0`).
- Fix absinfo item indexes in `UInput.uinput_create()` (thanks @forsenonlhaimaisentito).
- More robust comparison of `InputDevice` objects (thanks @isia).

8.13 0.5.0 (Jun 16, 2015)

- Write access to the input device is no longer mandatory. Evdev will first try to open the device for reading and writing and fallback to read-only. Methods that require write access (e.g. `set_led()`) will raise `EvdevError` if the device is open only for reading.

8.14 0.4.7 (Oct 07, 2014)

- Fallback to distutils if setuptools is not available.

8.15 0.4.6 (Oct 07, 2014)

- Rework documentation and docstrings once more.
- Fix install on Python 3.4 (works around issue21121).
- Fix `ioctl()` requested buffer size (thanks Jakub Wojciech Klama).

8.16 0.4.5 (Jul 06, 2014)

- Add method for returning a list of the currently active keys - `InputDevice.active_keys()` (thanks @spasche).
- Fix a potential buffer overflow in `ioctl_capabilities()` (thanks @spasche).

8.17 0.4.4 (Jun 04, 2014)

- Calling `InputDevice.read_one()` should always return `None`, when there is nothing to be read, even in case of a `EAGAIN` errno (thanks JPP).

8.18 0.4.3 (Dec 19, 2013)

- Silence `OSError` in destructor (thanks @polyphemus).
- Make `InputDevice.close()` work in cases in which stdin (fd 0) has been closed (thanks @polyphemus).

8.19 0.4.2 (Dec 13, 2013)

- Rework documentation and docstrings.
- Call `InputDevice.close()` from `InputDevice.__del__()`.

8.20 0.4.1 (Jul 24, 2013)

- Fix reference counting in `InputDevice.device_read()`, `InputDevice.device_read_many()` and `ioctl_capabilities()`.

8.21 0.4.0 (Jul 01, 2013)

- Add `FF_*` and `FF_STATUS` codes to `ecodes()` (thanks @bgilbert).
- Reverse event code mappings (`ecodes.{KEY,FF,REL,ABS}` and etc.) will now map to a list of codes, whenever a value corresponds to multiple codes:

```
>>> ecodes.KEY[152]
... ['KEY_COFFEE', 'KEY_SCREENLOCK']
>>> ecodes.KEY[30]
... 'KEY_A'
```

- Set the state of a LED through `InputDevice.set_led()` (thanks @accek).
- Open `InputDevice.fd` in `O_RDWR` mode from now on.
- Fix segfault in `InputDevice.device_read_many()` (thanks @bgilbert).

8.22 0.3.3 (May 29, 2013)

- Raise `IOError` from `InputDevice.device_read()` and `InputDevice.device_read_many()` when `InputDevice.read()` fails.
- Several stability and style changes (thank you debian code reviewers).

8.23 0.3.2 (Apr 05, 2013)

- Fix vendor id and product id order in `DeviceInfo()` (thanks @kived).

8.24 0.3.1 (Nov 23, 2012)

- `InputDevice.read()` will return an empty tuple if the device has nothing to offer (instead of segfaulting).
- Exclude unnecessary package data in sdist and bdist.

8.25 0.3.0 (Nov 06, 2012)

- Add ability to set/get auto-repeat settings with `EVIOC{SG}REP`.
- Add `InputDevice.version()` - the value of `EVIOCGVERSION`.
- Add `InputDevice.read_loop()`.
- Add `InputDevice.grab()` and `InputDevice.ungrab()` - exposes `EVIOCGRAB`.
- Add `InputDevice.leds()` - exposes `EVIOCGLED`.
- Replace `DeviceInfo` class with a namedtuple.
- Prevent `InputDevice.read_one()` from skipping events.
- Rename `AbsData` to `AbsInfo` (as in `struct input_absinfo`).

8.26 0.2.0 (Aug 22, 2012)

- Add the ability to set arbitrary device capabilities on uinput devices (defaults to all `EV_KEY` ecodes).
- Add `UInput.device` which is an open `InputDevice` to the input device that uinput 'spawns'.
- Add `UInput.capabilities()` which is just a shortcut to `UInput.device.capabilities()`.
- Rename `UInput.write()` to `UInput.write_event()`.
- Add a simpler `UInput.write(type, code, value)()` method.
- Make all `UInput()` constructor arguments optional (default device name is now `py-evdev-uinput`).
- Add the ability to set `absmin`, `absmax`, `absfuzz` and `absflat` when specifying the uinput device's capabilities.
- Remove the `nophys` argument - if a device fails the `EVIOCGPHYS` ioctl, phys will equal the empty string.
- Make `InputDevice.capabilities()` perform a `EVIOCGABS` ioctl for devices that support `EV_ABS` and return that info wrapped in an `AbsData` namedtuple.
- Split `ioctl_devinfo` into `ioctl_devinfo` and `ioctl_capabilities`.
- Split `UInput.uinput_open()` to `UInput.uinput_open()` and `UInput.uinput_create()`
- Add more uinput usage examples and documentation.
- Rewrite uinput tests.
- Remove `mouserel` and `mouseabs` from `UInput`.
- Tie the sphinx version and release to the distutils version.
- Set 'methods-before-attributes' sorting in the docs.
- Remove `KEY_CNT` and `KEY_MAX` from `ecodes.keys()`.

8.27 0.1.1 (May 18, 2012)

- Add `events.keys`, which is a combination of all `BTN_` and `KEY_` event codes.
- `ecodes.c` was not generated when installing through `pip`.

8.28 0.1.0 (May 17, 2012)

Initial Release

CHAPTER 9

License

This package is released under the terms of the Revised BSD License.

Python Module Index

e

Index

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

P

R

S

T

U

V

W