python-libinput Documentation

Release 0.1.0

Tomas Ravinskas

Mar 04, 2018

Contents

This package is a pure python wrapper around *libinput* using ctypes. It provides high-level object oriented api, taking care of reference counting, memory management and the like automatically.

libinput is a library that handles input devices for display servers and other applications that need to directly deal with input devices. It provides device detection, device handling, input device event processing and abstraction so minimize the amount of custom input code the user of libinput need to provide the common set of functionality that users expect. Input event processing includes scaling touch coordinates, generating pointer events from touchpads, pointer acceleration, etc.

libinput does this by reading character files in `/dev/input/`, so to use this package you need to run your code as root or to belong to `input` group.

CHAPTER 1

Installation

1.1 Dependencies

This package depends on `libinput` version >= 1.8.2

On python versions < 3.6, `aenum` package is also needed.

On python versions < 3.4, `selectors34` package is also needed.

On python versions < 3.3, `monotonic` package is also needed.

1.2 pip

`python-libinput` is distributed as a source package. To install, simply run:

```
pip install python-libinput
```

1.3 Source

Just run

```
python setup.py install
```

from the source directory.

CHAPTER 2

Usage

2.1 Creating manual context

```
>>>from libinput import LibInput, constant

>>> li = LibInput(context_type=constant.ContextType.PATH)
>>> device = li.path_add_device('/dev/input/event7')
>>> li.path_remove_device(device)
```

2.2 Creating udev context

udev context adds/removes devices from a given seat as they're physically added/removed. `LibInputUdev.assign_seat()` should only be called once per context.

```
>>> li = LibInput(context_type=constant.ContextType.UDEV)
>>> li.assign_seat('seat0')
```

2.3 Viewing device information

```
>>> device.name
'SIGMACHIP Usb Mouse'
>>> device.has_capability(constant.DeviceCapability.POINTER)
True
>>> device.pointer_has_button(0x110) # BTN_LEFT
True
```

2.4 Getting/filtering events

```
>>> for event in li.get_event():
>>>     if event.type == constant.EventType.POINTER_MOTION:
>>>         print(event.delta)
(15, 76)
>>> ...
```

CHAPTER 3

Contexts

3.1 LibInput

class libinput.**LibInput**(*context_type=<ContextType.PATH: 1>*, *grab=False*, *debug=False*)
A base/factory class for libinput context.

Context is used to manage devices and get events.

__init__(*context_type=<ContextType.PATH: 1>*, *grab=False*, *debug=False*)
Initialize context.

Parameters

- **context_type** (ContextType) – If *UDEV* devices are added/removed from udev seat. If *PATH* devices have to be added/removed manually.
- **grab** (*bool*) – If true get exclusive access to device(s).

Note: Grabbing an already grabbed device raises OSError

- **debug** (*bool*) – If false, only errors are printed.

log_handler
Callable that handles error/info/debug messages.

Parameters

- **priority** (LogPriority) – Message priority.
- **message** (*str*) – The message.

Default handler prints messages to stdout.

suspend()
Suspend monitoring for new devices and close existing devices.

This all but terminates libinput but does keep the context valid to be resumed with *resume()*.

resume()
Resume a suspended libinput context.

This re-enables device monitoring and adds existing devices.

> **Warning:** Resuming udev context before assigning seat causes segfault.

get_event(*timeout=None*)
Yield events from the internal libinput's queue.

Yields device events that are subclasses of *Event*.

If *timeout* is positive number, the generator will only block for *timeout* seconds when there are no events. If *timeout* is None (default) the generator will block indefinitely.

Parameters **timeout** (*float*) – Seconds to block when there are no events.

Yields *Event* – A generic event.

next_event_type()
Return the type of the next event in the internal queue.

This method does not pop the event off the queue and the next call to *get_event()* returns that event.

Returns The event type of the next available event or *NONE* if no event is available.

Return type *EventType*

3.2 LibInputPath

class libinput.**LibInputPath**(**args*, ***kwargs*)
libinput path context.

For a context of this type, devices have to be added/removed manually with *add_device()* and *remove_device()* respectively.

Note: Do not instanciate this class directly, instead call *LibInput* with context_type *PATH*.

---

add_device(*path*)
Add a device to a libinput context.

If successful, the device will be added to the internal list and re-opened on *resume()*. The device can be removed with *remove_device()*. If the device was successfully initialized, it is returned.

Parameters **path** (*str*) – Path to an input device.

Returns A device object or None.

Return type Device

remove_device(*device*)
Remove a device from a libinput context.

Events already processed from this input device are kept in the queue, the *DEVICE_REMOVED* event marks the end of events for this device.

If no matching device exists, this method does nothing.

Parameters **Device** (*Device*) – A previously added device.

3.3 LibInputUdev

class libinput.**LibInputUdev**(**args*, ***kwargs*)

libinput udev context.

For a context of this type, devices are added/removed automatically from the assigned seat.

Note: Do not instanciate this class directly, instead call *LibInput* with `context_type` *UDEV*.

---

assign_seat(*seat*)

Assign a seat to this libinput context.

New devices or the removal of existing devices will appear as events when iterating over *get_event()*.

assign_seat() succeeds even if no input devices are currently available on this seat, or if devices are available but fail to open. Devices that do not have the minimum capabilities to be recognized as pointer, keyboard or touch device are ignored. Such devices and those that failed to open are ignored until the next call to *resume()*.

Warning: This method may only be called once per context.

Parameters **seat** (*str*) – A seat identifier.

CHAPTER 4

Events

4.1 Event

class libinput.event.**Event**(*hevent*, *libinput*)
Base class for device events.

type
An enum describing event type.

Returns Event type.

Return type *EventType*

device
The device associated with this event.

For device added/removed events this is the device added or removed. For all other device events, this is the device that generated the event.

Returns Device object.

Return type Device

4.2 PointerEvent

class libinput.event.**PointerEvent**(**args*)
A pointer event.

An event representing relative or absolute pointer movement, a button press/release or scroll axis events.

time

Note: Timestamps may not always increase. See Event timestamps for details.

Returns The event time for this event in microseconds.

Return type int

delta
The delta between the last event and the current event.

For pointer events that are not of type *POINTER_MOTION*, this property raises AssertionError.

If a device employs pointer acceleration, the delta returned by this method is the accelerated delta.

Relative motion deltas are to be interpreted as pixel movement of a standardized mouse. See Normalization of relative motion for more details.

Returns The relative (x, y) movement since the last event.

Return type (float, float)

Raises AssertionError

delta_unaccelerated
The relative delta of the unaccelerated motion vector of the current event.

For pointer events that are not of type *POINTER_MOTION*, this property raises AssertionError.

Relative unaccelerated motion deltas are raw device coordinates. Note that these coordinates are subject to the device's native resolution. Touchpad coordinates represent raw device coordinates in the (X, Y) resolution of the touchpad. See Normalization of relative motion for more details.

Any rotation applied to the device also applies to unaccelerated motion (see `config_rotation_set_angle()`).

Returns The unaccelerated relative (x, y) movement since the last event.

Return type (float, float)

Raises AssertionError

absolute_coords
The current absolute coordinates of the pointer event, in mm from the top left corner of the device.

To get the corresponding output screen coordinate, use *transform_absolute_coords()*.

For pointer events that are not of type *POINTER_MOTION_ABSOLUTE*, this property raises AssertionError.

Returns The current absolute coordinates.

Return type (float, float)

Raises AssertionError

transform_absolute_coords(*width*, *height*)
Return the current absolute coordinates of the pointer event, transformed to screen coordinates.

For pointer events that are not of type *POINTER_MOTION_ABSOLUTE*, this method raises AssertionError.

Parameters

- **width** (*int*) – The current output screen width.
- **height** (*int*) – The current output screen height.

Returns The current absolute (x, y) coordinates transformed to a screen coordinates.

Return type (float, float)

button
The button that triggered this event.

For pointer events that are not of type *POINTER_BUTTON*, this property raises AssertionError.

Returns The button triggering this event.

Return type int

Raises AssertionError

button_state
The button state that triggered this event.

For pointer events that are not of type *POINTER_BUTTON*, this property raises AssertionError.

Returns The button state triggering this event.

Return type *ButtonState*

Raises AssertionError

seat_button_count
The total number of buttons pressed on all devices on the associated seat after the event was triggered.

For pointer events that are not of type *POINTER_BUTTON*, this property raises AssertionError.

Returns The seat wide pressed button count for the key of this event.

Return type int

Raises AssertionError

has_axis(*axis*)
Check if the event has a valid value for the given axis.

If this method returns True for an axis and *get_axis_value()* returns a value of 0, the event is a scroll stop event.

For pointer events that are not of type *POINTER_AXIS*, this method raises AssertionError.

Parameters **axis** (PointerAxis) – The axis to check.

Returns True if this event contains a value for this axis.

Return type bool

Raises AssertionError

get_axis_value(*axis*)
Return the axis value of the given axis.

The interpretation of the value depends on the axis. For the two scrolling axes *SCROLL_VERTICAL* and *SCROLL_HORIZONTAL*, the value of the event is in relative scroll units, with the positive direction being down or right, respectively. For the interpretation of the value, see *axis_source*.

If *has_axis()* returns False for an axis, this method returns 0 for that axis.

For pointer events that are not of type POINTER_AXIS, this method raises AssertionError.

Parameters **axis** (PointerAxis) – The axis who's value to get.

Returns The axis value of this event.

Return type float

Raises AssertionError

axis_source
The source for a given axis event.

Axis events (scroll events) can be caused by a hardware item such as a scroll wheel or emulated from other input sources, such as two-finger or edge scrolling on a touchpad.

If the source is *FINGER*, libinput guarantees that a scroll sequence is terminated with a scroll value of 0. A caller may use this information to decide on whether kinetic scrolling should be triggered on this scroll sequence. The coordinate system is identical to the cursor movement, i.e. a scroll value of 1 represents the equivalent relative motion of 1. If the source is *WHEEL*, no terminating event is guaranteed (though it may happen). Scrolling is in discrete steps, the value is the angle the wheel moved in degrees. The default is 15 degrees per wheel click, but some mice may have differently grained wheels. It is up to the caller how to interpret such different step sizes.

If the source is *CONTINUOUS*, no terminating event is guaranteed (though it may happen). The coordinate system is identical to the cursor movement, i.e. a scroll value of 1 represents the equivalent relative motion of 1. If the source is *WHEEL_TILT*, no terminating event is guaranteed (though it may happen). Scrolling is in discrete steps and there is no physical equivalent for the value returned here. For backwards compatibility, the value of this property is identical to a single mouse wheel rotation by this device (see the documentation for *WHEEL* above). Callers should not use this value but instead exclusively refer to the value returned by *get_axis_value_discrete()*.

For pointer events that are not of type POINTER_AXIS, this property raises AssertionError.

Returns The source for this axis event.

Return type *PointerAxisSource*

Raises AssertionError

get_axis_value_discrete(*axis*)
Return the axis value in discrete steps for a given axis event.

How a value translates into a discrete step depends on the source. If the source is *WHEEL*, the discrete value correspond to the number of physical mouse wheel clicks.

If the source is *CONTINUOUS* or *FINGER*, the discrete value is always 0.

Parameters **axis** (PointerAxis) – The axis who's value to get.

Returns The discrete value for the given event.

Return type float

Raises AssertionError

4.3 KeyboardEvent

class libinput.event.**KeyboardEvent**(**args*)
A keyboard event representing a key press/release.

time

Note: Timestamps may not always increase. See Event timestamps for details.

Returns The event time for this event in microseconds.

Return type int

key
The keycode that triggered this event.

Returns The keycode that triggered this key event.

Return type int

key_state
The logical state of the key.

Returns The state change of the key.

Return type *KeyState*

seat_key_count
The total number of keys pressed on all devices on the associated seat after the event was triggered.

Returns The seat wide pressed key count for the key of this event.

Return type int

4.4 TouchEvent

class libinput.event.**TouchEvent**(**args*)
Touch event representing a touch down, move or up, as well as a touch cancel and touch frame events.

time

Note: Timestamps may not always increase. See Event timestamps for details.

Returns The event time for this event in microseconds.

Return type int

slot
The slot of this touch event.

See the kernel's multitouch protocol B documentation for more information.

If the touch event has no assigned slot, for example if it is from a single touch device, this property returns -1.

For events not of type *TOUCH_DOWN*, *TOUCH_UP*, *TOUCH_MOTION* or *TOUCH_CANCEL*, this property raises AssertionError.

Returns The slot of this touch event.

Return type int

Raises AssertionError

seat_slot
The seat slot of the touch event.

A seat slot is a non-negative seat wide unique identifier of an active touch point.

Events from single touch devices will be represented as one individual touch point per device.

For events not of type *TOUCH_DOWN*, *TOUCH_UP*, *TOUCH_MOTION* or *TOUCH_CANCEL*, this property raises AssertionError.

Returns The seat slot of the touch event.

Return type int

Raises AssertionError

coords
The current absolute coordinates of the touch event, in mm from the top left corner of the device.

To get the corresponding output screen coordinates, use *transform_coords()*.

For events not of type *TOUCH_DOWN*, *TOUCH_MOTION*, this property raises AssertionError.

Returns The current absolute (x, y) coordinates.

Return type (float, float)

Raises AssertionError

transform_coords(*width*, *height*)
Return the current absolute coordinates of the touch event, transformed to screen coordinates.

For events not of type *TOUCH_DOWN*, *TOUCH_MOTION*, this method raises AssertionError.

Parameters

- **width** (*int*) – The current output screen width.
- **height** (*int*) – The current output screen height.

Returns The current absolute (x, y) coordinates transformed to screen coordinates.

Return type (float, float)

4.5 GestureEvent

class libinput.event.**GestureEvent**(**args*)
A gesture event representing gesture on a touchpad.

Gesture sequences always start with a GESTURE_FOO_START event. All following gesture events will be of the GESTURE_FOO_UPDATE type until a GESTURE_FOO_END is generated which signals the end of the gesture.

See Gestures for more information on gesture handling.

time

Note: Timestamps may not always increase. See Event timestamps for details.

Returns The event time for this event in microseconds.

Return type int

finger_count
The number of fingers used for a gesture.

This can be used e.g. to differentiate between 3 or 4 finger swipes. This property is valid for all gesture events and the returned finger count value will not change during a sequence.

Returns The number of fingers used for a gesture.

Return type int

cancelled

Return if the gesture ended normally, or if it was cancelled.

For gesture events that are not of type *GESTURE_SWIPE_END* or *GESTURE_PINCH_END*, this property raises AssertionError.

Returns True indicating that the gesture was cancelled.

Return type bool

Raises AssertionError

delta

The delta between the last event and the current event.

For gesture events that are not of type *GESTURE_SWIPE_UPDATE* or *GESTURE_PINCH_UPDATE*, this property raises AssertionError.

If a device employs pointer acceleration, the delta returned by this property is the accelerated delta.

Relative motion deltas are normalized to represent those of a device with 1000dpi resolution. See Normalization of relative motion for more details.

Returns The relative (x, y) movement since the last event.

Return type (float, float)

delta_unaccelerated

The relative delta of the unaccelerated motion vector of the current event.

For gesture events that are not of type *GESTURE_SWIPE_UPDATE* or *GESTURE_PINCH_UPDATE*, this property raises AssertionError.

Relative unaccelerated motion deltas are normalized to represent those of a device with 1000dpi resolution. See Normalization of relative motion for more details. Note that unaccelerated events are not equivalent to 'raw' events as read from the device.

Any rotation applied to the device also applies to gesture motion (see config_rotation_set_angle()).

Returns The unaccelerated relative (x, y) movement since the last event.

Return type (float, float)

scale

The absolute scale of a pinch gesture, the scale is the division of the current distance between the fingers and the distance at the start of the gesture.

The scale begins at 1.0, and if e.g. the fingers moved together by 50% then the scale will become 0.5, if they move twice as far apart as initially the scale becomes 2.0, etc.

For gesture events that are of type *GESTURE_PINCH_BEGIN*, this property returns 1.0.

For gesture events that are of type *GESTURE_PINCH_END*, this property returns the scale value of the most recent *GESTURE_PINCH_UPDATE* event (if any) or 1.0 otherwise.

For all other events this property raises AssertionError.

Returns The absolute scale of a pinch gesture.

Return type float

Raises AssertionError

angle_delta
The angle delta in degrees between the last and the current *GESTURE_PINCH_UPDATE* event.

For gesture events that are not of type *GESTURE_PINCH_UPDATE*, this property raises `AssertionError`.

The angle delta is defined as the change in angle of the line formed by the 2 fingers of a pinch gesture. Clockwise rotation is represented by a positive delta, counter-clockwise by a negative delta. If e.g. the fingers are on the 12 and 6 location of a clock face plate and they move to the 1 resp. 7 location in a single event then the angle delta is 30 degrees.

If more than two fingers are present, the angle represents the rotation around the center of gravity. The calculation of the center of gravity is implementation-dependent.

Returns The angle delta since the last event.

Return type float

Raises `AssertionError`

4.6 TabletToolEvent

class libinput.event.**TabletToolEvent**(**args*)
Tablet tool event representing an axis update, button press, or tool update.

Valid event types for this event are *TABLET_TOOL_AXIS*, *TABLET_TOOL_PROXIMITY*, *TABLET_TOOL_TIP* and *TABLET_TOOL_BUTTON*.

coords_have_changed
Check if the (x, y) axes were updated in this event.

For events that are not of type *TABLET_TOOL_AXIS*, *TABLET_TOOL_TIP*, or *TABLET_TOOL_PROXIMITY*, this property is `False`.

Returns `True` if the axes were updated or `False` otherwise.

Return type bool

pressure_has_changed
Check if the pressure axis was updated in this event.

For events that are not of type *TABLET_TOOL_AXIS*, *TABLET_TOOL_TIP*, or *TABLET_TOOL_PROXIMITY*, this property is `False`.

Returns `True` if the axis was updated or `False` otherwise.

Return type bool

distance_has_changed
Check if the distance axis was updated in this event.

For events that are not of type *TABLET_TOOL_AXIS*, *TABLET_TOOL_TIP*, or *TABLET_TOOL_PROXIMITY*, this property is `False`. For tablet tool events of type *TABLET_TOOL_PROXIMITY*, this property is always `True`.

Returns `True` if the axis was updated or `False` otherwise.

Return type bool

tilt_has_changed
Check if the tilt axes were updated in this event.

For events that are not of type *TABLET_TOOL_AXIS*, *TABLET_TOOL_TIP*, or *TABLET_TOOL_PROXIMITY*, this property is `False`.

Returns `True` if the axes were updated or `False` otherwise.

Return type bool

rotation_has_changed
Check if the z-rotation axis was updated in this event.

For events that are not of type *TABLET_TOOL_AXIS*, *TABLET_TOOL_TIP*, or *TABLET_TOOL_PROXIMITY*, this property is `False`.

Returns `True` if the axis was updated or `False` otherwise.

Return type bool

slider_has_changed
Check if the slider axis was updated in this event.

For events that are not of type *TABLET_TOOL_AXIS*, *TABLET_TOOL_TIP*, or *TABLET_TOOL_PROXIMITY*, this property is `False`.

Returns `True` if the axis was updated or `False` otherwise.

Return type bool

wheel_has_changed
Check if the wheel axis was updated in this event.

For events that are not of type *TABLET_TOOL_AXIS*, *TABLET_TOOL_TIP*, or *TABLET_TOOL_PROXIMITY*, this property is `False`.

Returns `True` if the axis was updated or `False` otherwise.

Return type bool

coords
The (X, Y) coordinates of the tablet tool, in mm from the top left corner of the tablet in its current logical orientation.

Use *transform_coords()* for transforming the axes values into a different coordinate space.

Note: On some devices, returned value may be negative or larger than the width of the device. See Out-of-bounds motion events for more details.

Returns The current values of the the axes.

Return type (float, float)

delta
The delta between the last event and the current event.

If the tool employs pointer acceleration, the delta contained in this property is the accelerated delta.

This value is in screen coordinate space, the delta is to be interpreted like the value of *PointerEvent.delta*. See Relative motion for tablet tools for more details.

Returns The relative (x, y) movement since the last event.

Return type (float, float)

pressure
The current pressure being applied on the tool in use, normalized to the range [0, 1].

If this axis does not exist on the current tool, this property is 0.

Returns The current value of the the axis.

Return type float

distance
The current distance from the tablet's sensor, normalized to the range [0, 1].

If this axis does not exist on the current tool, this property is 0.

Returns The current value of the the axis.

Return type float

tilt_axes
The current tilt along the (X, Y) axes of the tablet's current logical orientation, in degrees off the tablet's Z axis.

That is, if the tool is perfectly orthogonal to the tablet, the tilt angle is 0. When the top tilts towards the logical top/left of the tablet, the x/y tilt angles are negative, if the top tilts towards the logical bottom/right of the tablet, the x/y tilt angles are positive.

If these axes do not exist on the current tool, this property returns (0, 0).

Returns The current value of the axes in degrees.

Return type (float, float)

rotation
The current Z rotation of the tool in degrees, clockwise from the tool's logical neutral position.

For tools of type *MOUSE* and *LENS* the logical neutral position is pointing to the current logical north of the tablet. For tools of type *BRUSH*, the logical neutral position is with the buttons pointing up.

If this axis does not exist on the current tool, this property is 0.

Returns The current value of the the axis.

Return type float

slider_position
The current position of the slider on the tool, normalized to the range [-1, 1].

The logical zero is the neutral position of the slider, or the logical center of the axis. This axis is available on e.g. the Wacom Airbrush.

If this axis does not exist on the current tool, this property is 0.

Returns The current value of the the axis.

Return type float

wheel_delta
The delta for the wheel in degrees.

Returns The delta of the wheel, in degrees, compared to the last event.

Return type float

wheel_delta_discrete
The delta for the wheel in discrete steps (e.g. wheel clicks).

Returns The delta of the wheel, in discrete steps, compared to the last event.

Return type int

transform_coords(*width*, *height*)
Return the current absolute (x, y) coordinates of the tablet tool event, transformed to screen coordinates.

Note: On some devices, returned value may be negative or larger than the width of the device. See Out-of-bounds motion events for more details.

Parameters

- **width** (*int*) – The current output screen width.
- **height** (*int*) – The current output screen height.

Returns The current absolute (x, y) coordinates transformed to screen coordinates.

Return type (float, float)

tool
The tool that was in use during this event.

If the caller keeps a reference to a tool, the tool object will compare equal to the previously obtained tool object.

Note: Physical tool tracking requires hardware support. If unavailable, libinput creates one tool per type per tablet. See Tracking unique tools for more details.

Returns The new tool triggering this event.

Return type *TabletTool*

proximity_state
The new proximity state of a tool from a proximity event.

Used to check whether or not a tool came in or out of proximity during an event of type *TABLET_TOOL_PROXIMITY*.

See Handling of proximity events for recommendations on proximity handling.

Returns The new proximity state of the tool from the event.

Return type *TabletToolProximityState*

tip_state
The new tip state of a tool from a tip event.

Used to check whether or not a tool came in contact with the tablet surface or left contact with the tablet surface during an event of type *TABLET_TOOL_TIP*.

Returns The new tip state of the tool from the event.

Return type *TabletToolTipState*

button
The button that triggered this event.

For events that are not of type *TABLET_TOOL_BUTTON*, this property raises `AssertionError`.

Returns The button triggering this event.

Return type int

button_state
The button state of the event.

For events that are not of type *TABLET_TOOL_BUTTON*, this property raises AssertionError.

Returns The button state triggering this event.

Return type *ButtonState*

seat_button_count
The total number of buttons pressed on all devices on the associated seat after the the event was triggered.

For events that are not of type *TABLET_TOOL_BUTTON*, this property raises AssertionError.

Returns The seat wide pressed button count for the key of this event.

Return type int

time

Note: Timestamps may not always increase. See Event timestamps for details.

Returns The event time for this event in microseconds.

Return type int

4.7 TabletPadEvent

class libinput.event.**TabletPadEvent**(**args*)
Tablet pad event representing a button press or ring/strip update on the tablet pad itself.

Valid event types for this event are *TABLET_PAD_BUTTON*, *TABLET_PAD_RING* and *TABLET_PAD_STRIP*.

ring_position
The current position of the ring, in degrees counterclockwise from the northern-most point of the ring in the tablet's current logical orientation.

If the source is *FINGER*, libinput sends a terminating event with a ring value of -1 when the finger is lifted from the ring. A caller may use this information to e.g. determine if kinetic scrolling should be triggered.

For events not of type *TABLET_PAD_RING*, this property raises AssertionError.

Returns The current value of the the axis. -1 if the finger was lifted.

Return type float

Raises AssertionError

ring_number
The number of the ring that has changed state, with 0 being the first ring.

On tablets with only one ring, this method always returns 0.

For events not of type *TABLET_PAD_RING*, this property raises AssertionError.

Returns The index of the ring that changed state.

Return type int

Raises AssertionError

ring_source
The source of the interaction with the ring.

If the source is *FINGER*, libinput sends a ring position value of -1 to terminate the current interaction.

For events not of type *TABLET_PAD_RING*, this property raises AssertionError.

Returns The source of the ring interaction.

Return type *TabletPadRingAxisSource*

Raises AssertionError

strip_position
The current position of the strip, normalized to the range [0, 1], with 0 being the top/left-most point in the tablet's current logical orientation.

If the source is *FINGER*, libinput sends a terminating event with a value of -1 when the finger is lifted from the strip. A caller may use this information to e.g. determine if kinetic scrolling should be triggered.

For events not of type *TABLET_PAD_STRIP*, this property raises AssertionError.

Returns The current value of the the axis. -1 if the finger was lifted.

Return type float

Raises AssertionError

strip_number
The number of the strip that has changed state, with 0 being the first strip.

On tablets with only one strip, this method always returns 0.

For events not of type *TABLET_PAD_STRIP*, this property raises AssertionError.

Returns The index of the strip that changed state.

Return type int

Raises AssertionError

strip_source
The source of the interaction with the strip.

If the source is *FINGER*, libinput sends a strip position value of -1 to terminate the current interaction.

For events not of type *TABLET_PAD_STRIP*, this property raises AssertionError.

Returns The source of the strip interaction.

Return type *TabletPadStripAxisSource*

Raises AssertionError

button_number
The button number that triggered this event, starting at 0.

For events that are not of type TABLET_PAD_BUTTON, this property raises AssertionError.

Note that the number returned is a generic sequential button number and not a semantic button code as defined in linux/input.h. See Tablet pad button numbers for more details.

Returns The button triggering this event.

Return type int

Raises AssertionError

button_state
The button state of the event.

For events not of type *TABLET_PAD_BUTTON*, this property raises AssertionError.

Returns The button state triggering this event.

Return type *ButtonState*

Raises AssertionError

mode
The mode the button, ring, or strip that triggered this event is in, at the time of the event.

The mode is a virtual grouping of functionality, usually based on some visual feedback like LEDs on the pad. See Tablet pad modes for details. Mode indices start at 0, a device that does not support modes always returns 0.

Mode switching is controlled by libinput and more than one mode may exist on the tablet. This method returns the mode that this event's button, ring or strip is logically in. If the button is a mode toggle button and the button event caused a new mode to be toggled, the mode returned is the new mode the button is in.

Note that the returned mode is the mode valid as of the time of the event. The returned mode may thus be different to the mode returned by *mode*. See *mode* for details.

Returns The 0-indexed mode of this button, ring or strip at the time of the event.

Return type int

mode_group
The mode group that the button, ring, or strip that triggered this event is considered in.

The mode is a virtual grouping of functionality, usually based on some visual feedback like LEDs on the pad. See Tablet pad modes for details.

Returns The mode group of the button, ring or strip that caused this event.

Return type *TabletPadModeGroup*

time

Note: Timestamps may not always increase. See Event timestamps for details.

Returns The event time for this event in microseconds.

Return type int

4.8 SwitchEvent

class libinput.event.**SwitchEvent**(**args*)
A switch event representing a changed state in a switch.

switch
The switch that triggered this event.

Returns The switch triggering this event.

Return type *Switch*

switch_state
The switch state that triggered this event.

Returns The switch state triggering this event.

Return type *SwitchState*

time

Note: Timestamps may not always increase. See Event timestamps for details.

Returns The event time for this event in microseconds.

Return type int

4.9 DeviceNotifyEvent

class libinput.event.**DeviceNotifyEvent**(**args*)
An event notifying the caller of a device being added or removed.

CHAPTER 5

Devices and Seats

5.1 Device

class libinput.device.**Device**(**args*)

An input device.

sysname

The system name of the device.

To get the descriptive device name, use *name*.

Returns System name of the device.

Return type str

name

The descriptive device name as advertised by the kernel and/or the hardware itself.

To get the sysname for this device, use *sysname*.

Returns The device name.

Return type str

id_product

The product ID for this device.

Returns The product ID of this device.

Return type int

id_vendor

The vendor ID for this device.

Returns The vendor ID of this device.

Return type int

seat
The seat associated with this input device, see Seats for details.

A seat can be uniquely identified by the physical and logical seat name. As long as a reference to a seat is kept, it will compare equal to another seat object with the same physical/logical name pair.

Returns The seat this input device belongs to.

Return type *Seat*

set_seat_logical_name(*seat*)
Change the logical seat associated with this device by removing the device and adding it to the new seat.

This command is identical to physically unplugging the device, then re-plugging it as a member of the new seat. libinput will generate a *DEVICE_REMOVED* event and this *Device* is considered removed from the context; it will not generate further events. A *DEVICE_ADDED* event is generated with a new *Device*. It is the caller's responsibility to update references to the new device accordingly.

If the logical seat name already exists in the device's physical seat, the device is added to this seat. Otherwise, a new seat is created.

Note: This change applies to this device until removal or *suspend()*, whichever happens earlier.

Parameters **seat** (*str*) – The new logical seat name.

Raises AssertionError

udev_device
A udev handle to the device that is this libinput device, if any.

The returned handle has a refcount of at least 1, the caller must call `udev_device_unref()` once to release the associated resources. See the libudev documentation for details.

Some devices may not have a udev device, or the udev device may be unobtainable. This function returns None if no udev device was available.

Calling this function multiple times for the same device may not return the same udev handle each time.

Returns A udev handle to the device with a refcount of >= 1 or None if this device is not represented by a udev device.

Return type int

led_update(*leds*)
Update the LEDs on the device, if any.

If the device does not have LEDs, or does not have one or more of the LEDs given in the mask, this method does nothing.

Parameters **leds** (Led) – A mask of the LEDs to set, or unset.

has_capability(*capability*)
Check if the given device has the specified capability.

Parameters

- **capability** (DeviceCapability) – A capability
- **check for.** (*to*) –

Returns True if the given device has the capability or False otherwise.

Return type bool

size
The physical size of a device in mm, where meaningful.

This property is only valid on devices with the required data, i.e. tablets, touchpads and touchscreens. For other devices this property raises `AssertionError`.

Returns (Width, Height) in mm.

Return type (float, float)

Raises `AssertionError`

pointer_has_button(*button*)
Check if a *POINTER* device has a given button.

Parameters **button** (*int*) – Button to check for, see `input.h` for button definitions.

Returns `True` if the device has this button, `False` if it does not.

Return type bool

Raises `AssertionError`

keyboard_has_key(*key*)
Check if a *KEYBOARD* device has a given key.

Parameters **key** (*int*) – Key to check for, see `input.h` for key definitions.

Returns `True` if the device has this key, `False` if it does not.

Return type bool

Raises `AssertionError`

tablet_pad_get_num_buttons()
Return the number of buttons on a device with the *TABLET_PAD* capability.

Buttons on a pad device are numbered sequentially, see Tablet pad button numbers for details.

Returns The number of buttons supported by the device.

Return type int

Raises `AssertionError`

tablet_pad_get_num_rings()
Return the number of rings a device with the *TABLET_PAD* capability provides.

Returns The number of rings or 0 if the device has no rings.

Return type int

Raises `AssertionError`

tablet_pad_get_num_strips()
Return the number of strips a device with the *TABLET_PAD* capability provides.

Returns The number of strips or 0 if the device has no strips.

Return type int

Raises `AssertionError`

tablet_pad_get_num_mode_groups()
Most devices only provide a single mode group, however devices such as the Wacom Cintiq 22HD provide two mode groups.

If multiple mode groups are available, a caller should use *has_button()*, *has_ring()* and *has_strip()* to associate each button, ring and strip with the correct mode group.

Returns The number of mode groups available on this device.

Return type int

Raises AssertionError

tablet_pad_get_mode_group(*group*)

While a reference is kept by the caller, the returned mode group will compare equal with mode group returned by each subsequent call of this method with the same index and mode group returned from *mode_group*, provided the event was generated by this mode group.

Parameters **group** (*int*) – A mode group index.

Returns The mode group with the given index or None if an invalid index is given.

Return type *TabletPadModeGroup*

config

Device configuration.

Returns An object providing device configuration methods.

Return type *DeviceConfig*

5.2 Seat

class libinput.device.**Seat**(*hseat*, *libinput*)

A seat has two identifiers, the physical name and the logical name.

A device is always assigned to exactly one seat. It may change to a different logical seat but it cannot change physical seats. See Seats for details.

Two instances of *Seat* compare equal if they refer to the same physical/logical seat.

physical_name

The physical name of the seat.

For libinput contexts created from udev, this is always the same value as passed into *assign_seat()* and all seats from that context will have the same physical name.

The physical name of the seat is one that is usually set by the system or lower levels of the stack. In most cases, this is the base filter for devices - devices assigned to seats outside the current seat will not be available to the caller.

Returns The physical name of this seat.

Return type str

logical_name

The logical name of the seat.

This is an identifier to group sets of devices within the compositor.

Returns The logical name of this seat.

Return type str

5.3 DeviceConfig

class libinput.device.**DeviceConfig**(**args*)

A configuration object.

tap

Tapping-related configuration methods.

Returns

Return type *DeviceConfigTap*

calibration

Calibration matrix configuration methods.

Returns

Return type *DeviceConfigCalibration*

send_events

Event sending configuration methods.

Returns

Return type *DeviceConfigSendEvents*

accel

Pointer acceleration configuration methods.

Returns

Return type *DeviceConfigAccel*

scroll

Scrolling configuration methods.

Returns

Return type *DeviceConfigScroll*

left_handed

Left-handed usage configuration methods.

Returns

Return type *DeviceConfigLeftHanded*

click

Click method configuration methods.

Returns

Return type *DeviceConfigClick*

middle_emulation

Middle mouse button emulation configuration methods.

Returns

Return type *DeviceConfigMiddleEmulation*

dwt

Disable-while-typing configuration methods.

Returns

Return type *DeviceConfigDwt*

rotation
Rotation configuration methods.

Returns

Return type *DeviceConfigRotation*

class libinput.device.**DeviceConfigTap**(**args*)
Tapping-related configuration methods.

finger_count
Check if the device supports tap-to-click and how many fingers can be used for tapping.

See *set_enabled()* for more information.

Returns The number of fingers that can generate a tap event, or 0 if the device does not support tapping.

Return type int

set_enabled(*state*)
Enable or disable tap-to-click on this device, with a default mapping of 1, 2, 3 finger tap mapping to left, right, middle click, respectively.

Tapping is limited by the number of simultaneous touches supported by the device, see *finger_count*.

Parameters **state** (TapState) – *ENABLED* to enable tapping or *DISABLED* to disable tapping.

Returns A config status code. Disabling tapping on a device that does not support tapping always succeeds.

Return type *ConfigStatus*

enabled
Check if tap-to-click is enabled on this device.

If the device does not support tapping, this property is always *DISABLED*.

Returns Whether tapping is enabled or disabled.

Return type *TapState*

default_enabled
The default setting for whether tap-to-click is enabled on this device.

Returns Whether tapping is enabled or disabled.

Return type *TapState*

set_button_map(*button_map*)
Set the finger number to button number mapping for tap-to-click.

The default mapping on most devices is to have a 1, 2 and 3 finger tap to map to the left, right and middle button, respectively. A device may permit changing the button mapping but disallow specific maps. In this case *UNSUPPORTED* is returned, the caller is expected to handle this case correctly.

Changing the button mapping may not take effect immediately, the device may wait until it is in a neutral state before applying any changes.

The mapping may be changed when tap-to-click is disabled. The new mapping takes effect when tap-to-click is enabled in the future.

If *finger_count* is 0, this method raises AssertionError.

Parameters **button_map** (TapButtonMap) – The new finger-to-button number mapping.

Returns A config status code.

Return type *ConfigStatus*

Raises AssertionError

button_map
The finger number to button number mapping for tap-to-click.

For devices that do not support tapping (i.e. *finger_count* is 0), this property raises AssertionError.

Returns The current finger-to-button number mapping.

Return type *TapButtonMap*

Raises AssertionError

default_button_map
The default finger number to button number mapping for tap-to-click.

For devices that do not support tapping (i.e. *finger_count* is 0), this property raises AssertionError.

Returns The default finger-to-button number mapping.

Return type *TapButtonMap*

Raises AssertionError

set_drag_enabled(*state*)
Enable or disable tap-and-drag on this device.

When enabled, a single-finger tap immediately followed by a finger down results in a button down event, subsequent finger motion thus triggers a drag. The button is released on finger up. See Tap-and-drag for more details.

Parameters **state** (DragState) – *ENABLED* to enable, *DISABLED* to disable tap-and-drag.

Returns Whether this method succeeds.

Return type *ConfigStatus*

drag_enabled
Whether tap-and-drag is enabled or disabled on this device.

Returns Whether tap-and-drag is enabled.

Return type *DragState*

default_drag_enabled
Whether tap-and-drag is enabled or disabled by default on this device.

Returns Whether tap-and-drag is enabled by default.

Return type *DragState*

set_drag_lock_enabled(*state*)
Enable or disable drag-lock during tapping on this device.

When enabled, a finger may be lifted and put back on the touchpad within a timeout and the drag process continues. When disabled, lifting the finger during a tap-and-drag will immediately stop the drag. See Tap-and-drag for details.

Enabling drag lock on a device that has tapping disabled is permitted, but has no effect until tapping is enabled.

Parameters **state** (DragLockState) – *ENABLED* to enable drag lock or *DISABLED* to disable drag lock.

Returns A config status code. Disabling drag lock on a device that does not support tapping always succeeds.

Return type *ConfigStatus*

drag_lock_enabled
Check if drag-lock during tapping is enabled on this device.

If the device does not support tapping, this function always returns *DISABLED*.

Drag lock may be enabled even when tapping is disabled.

Returns Whether drag lock is enabled.

Return type *DragLockState*

default_drag_lock_enabled
Check if drag-lock during tapping is enabled by default on this device.

If the device does not support tapping, this function always returns *DISABLED*.

Drag lock may be enabled by default even when tapping is disabled by default.

Returns Whether drag lock is enabled by default.

Return type *DragLockState*

class libinput.device.**DeviceConfigCalibration**(**args*)
Calibration matrix configuration methods.

has_matrix()
Check if the device can be calibrated via a calibration matrix.

Returns True if the device can be calibrated, False otherwise.

Return type bool

set_matrix(*matrix*)
Apply the 3x3 transformation matrix to absolute device coordinates.

This matrix has no effect on relative events.

Given a 6-element array [a, b, c, d, e, f], the matrix is applied as

```
[ a  b  c ]   [ x ]
[ d  e  f ] * [ y ]
[ 0  0  1 ]   [ 1 ]
```

The translation component (c, f) is expected to be normalized to the device coordinate range. For example, the matrix

```
[ 1 0  1 ]
[ 0 1 -1 ]
[ 0 0  1 ]
```

moves all coordinates by 1 device-width to the right and 1 device-height up.

The rotation matrix for rotation around the origin is defined as

```
[ cos(a) -sin(a) 0 ]
[ sin(a)  cos(a) 0 ]
[   0       0    1 ]
```

Note that any rotation requires an additional translation component to translate the rotated coordinates back into the original device space. The rotation matrixes for 90, 180 and 270 degrees clockwise are:

```
90 deg cw:              180 deg cw:             270 deg cw:
[ 0 -1 1]               [ -1  0 1]              [  0 1 0 ]
[ 1  0 0]               [  0 -1 1]              [ -1 0 1 ]
[ 0  0 1]               [  0  0 1]              [  0 0 1 ]
```

Parameters **matrix** (*iterable*) – An array representing the first two rows of a 3x3 matrix as described above.

Returns A config status code.

Return type *ConfigStatus*

matrix
The current calibration matrix for this device.

Returns `False` if no calibration is set and the returned matrix is the identity matrix, `True` otherwise. `tuple` representing the first two rows of a 3x3 matrix as described in *set_matrix()*.

Return type (bool, (float, float, float, float, float, float))

default_matrix
The default calibration matrix for this device.

On most devices, this is the identity matrix. If the udev property `LIBINPUT_CALIBRATION_MATRIX` is set on the respective udev device, that property's value becomes the default matrix, see Static device configuration via udev.

Returns `False` if no calibration is set and the returned matrix is the identity matrix, `True` otherwise. `tuple` representing the first two rows of a 3x3 matrix as described in `config_calibration_set_matrix()`.

Return type (bool, (float, float, float, float, float, float))

class libinput.device.**DeviceConfigSendEvents**(**args*)
Event sending configuration methods.

modes
The possible send-event modes for this device.

These modes define when a device may process and send events.

Returns A bitmask of possible modes.

Return type *SendEventsMode*

set_mode(*mode*)
Set the send-event mode for this device.

The mode defines when the device processes and sends events to the caller.

The selected mode may not take effect immediately. Events already received and processed from this device are unaffected and will be passed to the caller on the next call to *get_event()*.

If the mode is a bitmask of *SendEventsMode*, the device may wait for or generate events until it is in a neutral state. For example, this may include waiting for or generating button release events.

If the device is already suspended, this function does nothing and returns success. Changing the send-event mode on a device that has been removed is permitted.

Parameters **mode** (SendEventsMode) – A bitmask of send-events modes.

Returns A config status code.

Return type *ConfigStatus*

mode
The send-event mode for this device.

The mode defines when the device processes and sends events to the caller.

If a caller enables the bits for multiple modes, some of which are subsets of another mode libinput may drop the bits that are subsets. In other words, don't expect *mode* to always be exactly the same bitmask as passed into *set_mode()*.

Returns The current bitmask of the send-event mode for this device.

Return type *SendEventsMode*

default_mode
The default send-event mode for this device.

The mode defines when the device processes and sends events to the caller.

Returns The bitmask of the send-event mode for this device.

Return type *SendEventsMode*

class libinput.device.**DeviceConfigAccel**(**args*)
Pointer acceleration configuration methods.

is_available()
Check if a device uses libinput-internal pointer-acceleration.

Returns False if the device is not accelerated, True if it is accelerated

Return type bool

set_speed(*speed*)
Set the pointer acceleration speed of this pointer device within a range of [-1, 1], where 0 is the default acceleration for this device, -1 is the slowest acceleration and 1 is the maximum acceleration available on this device.

The actual pointer acceleration mechanism is implementation-dependent, as is the number of steps available within the range. libinput picks the semantically closest acceleration step if the requested value does not match a discrete setting.

Parameters **speed** (*float*) – The normalized speed, in a range of [-1, 1].

Returns A config status code.

Return type *ConfigStatus*

speed
The current pointer acceleration setting for this pointer device.

The returned value is normalized to a range of [-1, 1]. See *set_speed()* for details.

Returns The current speed, range -1 to 1.

Return type float

default_speed
The default speed setting for this device, normalized to a range of [-1, 1].

See *set_speed()* for details.

Returns The default speed setting for this device.

Return type float

profiles
A bitmask of the configurable acceleration modes available on this device.

Returns A bitmask of all configurable modes available on this device.

Return type *AccelProfile*

set_profile(*profile*)
Set the pointer acceleration profile of this pointer device to the given mode.

Parameters

- **profile** (AccelProfile) – The mode to set
- **device to.** (*the*) –

Returns A config status code.

Return type *ConfigStatus*

profile
The current pointer acceleration profile for this pointer device.

Returns The currently configured pointer acceleration profile.

Return type *AccelProfile*

default_profile
The default pointer acceleration profile for this pointer device.

Returns The default acceleration profile for this device.

Return type *AccelProfile*

class libinput.device.**DeviceConfigScroll**(**args*)
Scrolling configuration methods.

has_natural_scroll()
True if the device supports "natural scrolling".

In traditional scroll mode, the movement of fingers on a touchpad when scrolling matches the movement of the scroll bars. When the fingers move down, the scroll bar moves down, a line of text on the screen moves towards the upper end of the screen. This also matches scroll wheels on mice (wheel down, content moves up).

Natural scrolling is the term coined by Apple for inverted scrolling. In this mode, the effect of scrolling movement of fingers on a touchpad resemble physical manipulation of paper. When the fingers move down, a line of text on the screen moves down (scrollbars move up). This is the opposite of scroll wheels on mice.

A device supporting natural scrolling can be switched between traditional scroll mode and natural scroll mode.

Returns False if natural scrolling is not supported, True if natural scrolling is supported by this device.

Return type bool

set_natural_scroll_enabled(*enable*)
Enable or disable natural scrolling on the device.

Parameters **enable** (*bool*) – True to enable, False to disable natural scrolling.

Returns A config status code.

Return type *ConfigStatus*

natural_scroll_enabled
The current mode for scrolling on this device.

Returns `False` if natural scrolling is disabled, `True` if enabled.

Return type bool

default_natural_scroll_enabled
The default mode for scrolling on this device.

Returns `False` if natural scrolling is disabled by default, `True` if enabled.

Return type bool

methods
Check which scroll methods a device supports.

The method defines when to generate scroll axis events instead of pointer motion events.

Returns A bitmask of possible methods.

Return type *ScrollMethod*

set_method(*method*)
Set the scroll method for this device.

The method defines when to generate scroll axis events instead of pointer motion events.

Note: Setting *ON_BUTTON_DOWN* enables the scroll method, but scrolling is only activated when the configured button is held down. If no button is set, i.e. *button* is 0, scrolling cannot activate.

Parameters **method** (`ScrollMethod`) – The scroll method for this device.

Returns A config status code.

Return type *ConfigStatus*

method
The scroll method for this device.

The method defines when to generate scroll axis events instead of pointer motion events.

Returns The current scroll method for this device.

Return type *ScrollMethod*

default_method
The default scroll method for this device.

The method defines when to generate scroll axis events instead of pointer motion events.

Returns The default scroll method for this device.

Return type *ScrollMethod*

set_button(*button*)
Set the button for the *ON_BUTTON_DOWN* method for this device.

When the current scroll method is set to *ON_BUTTON_DOWN*, no button press/release events will be send for the configured button.

When the configured button is pressed, any motion events along a scroll-capable axis are turned into scroll axis events.

Note: Setting the button does not change the scroll method. To change the scroll method call *set_method()*. If the button is 0, button scrolling is effectively disabled.

Parameters **button** (*int*) – The button which when pressed switches to sending scroll events.

Returns A config status code.

Return type *ConfigStatus*

button

The button for the *ON_BUTTON_DOWN* method for this device.

If *ON_BUTTON_DOWN* scroll method is not supported, or no button is set, this property is 0.

Note: The return value is independent of the currently selected scroll-method. For button scrolling to activate, a device must have the *ON_BUTTON_DOWN* method enabled, and a non-zero button set as scroll button.

Returns The button which when pressed switches to sending scroll events.

Return type int

default_button

The default button for the *ON_BUTTON_DOWN* method for this device.

If *ON_BUTTON_DOWN* scroll method is not supported, or no default button is set, this property is 0.

Returns The default button for the *ON_BUTTON_DOWN* method.

Return type int

class libinput.device.**DeviceConfigLeftHanded**(**args*)

Left-handed usage configuration methods.

is_available()

Check if a device has a configuration that supports left-handed usage.

Returns `True` if the device can be set to left-handed, or `False` otherwise

Return type bool

set(*enable*)

Set the left-handed configuration of the device.

The exact behavior is device-dependent. On a mouse and most pointing devices, left and right buttons are swapped but the middle button is unmodified. On a touchpad, physical buttons (if present) are swapped. On a clickpad, the top and bottom software-emulated buttons are swapped where present, the main area of the touchpad remains a left button. Tapping and clickfinger behavior is not affected by this setting.

Changing the left-handed configuration of a device may not take effect until all buttons have been logically released.

Parameters

- **enable** (*bool*) – `False` to disable, `True` to enable

- **mode.** (*left-handed*) –

Returns A configuration status code.

Return type *ConfigStatus*

enabled
The current left-handed configuration of the device.

Returns `False` if the device is in right-handed mode, `True` if the device is in left-handed mode.

Return type bool

default_enabled
The default left-handed configuration of the device.

Returns `False` if the device is in right-handed mode by default, or `True` if the device is in left-handed mode by default.

Return type bool

class libinput.device.**DeviceConfigClick**(**args*)
Click method configuration methods.

methods
Check which button click methods a device supports.

The button click method defines when to generate software-emulated buttons, usually on a device that does not have a specific physical button available.

Returns A bitmask of possible methods.

Return type *ClickMethod*

set_method(*method*)
Set the button click method for this device.

The button click method defines when to generate software-emulated buttons, usually on a device that does not have a specific physical button available.

Note: The selected click method may not take effect immediately. The device may require changing to a neutral state first before activating the new method.

Parameters **method** (`ClickMethod`) – The button click method.

Returns A config status code.

Return type *ConfigStatus*

method
The button click method for this device.

The button click method defines when to generate software-emulated buttons, usually on a device that does not have a specific physical button available.

Returns The current button click method for this device.

Return type *ClickMethod*

default_method
The default button click method for this device.

The button click method defines when to generate software-emulated buttons, usually on a device that does not have a specific physical button available.

Returns The default button click method for this device.

Return type *ClickMethod*

class libinput.device.**DeviceConfigMiddleEmulation**(**args*)

Middle mouse button emulation configuration methods.

is_available()

Check if middle mouse button emulation configuration is available on this device.

See Middle button emulation for details.

Note: Some devices provide middle mouse button emulation but do not allow enabling/disabling that emulation. These devices return `False` in *`is_available`*.

Returns `True` if middle mouse button emulation is available and can be configured, `False` otherwise.

Return type bool

set_enabled(*state*)

Enable or disable middle button emulation on this device.

When enabled, a simultaneous press of the left and right button generates a middle mouse button event. Releasing the buttons generates a middle mouse button release, the left and right button events are discarded otherwise.

See Middle button emulation for details.

Parameters **state** (`MiddleEmulationState`) – *`DISABLED`* to disable, *`ENABLED`* to enable middle button emulation.

Returns A config status code. Disabling middle button emulation on a device that does not support middle button emulation always succeeds.

Return type *ConfigStatus*

enabled

Check if configurable middle button emulation is enabled on this device.

See Middle button emulation for details.

If the device does not have configurable middle button emulation, this method returns *`DISABLED`*.

Note: Some devices provide middle mouse button emulation but do not allow enabling/disabling that emulation. These devices always return *`DISABLED`*.

Returns *`DISABLED`* if disabled or not available/configurable, *`ENABLED`* if enabled.

Return type *MiddleEmulationState*

default_enabled

Check if configurable middle button emulation is enabled by default on this device.

See Middle button emulation for details.

If the device does not have configurable middle button emulation, this method returns *DISABLED*.

Note: Some devices provide middle mouse button emulation but do not allow enabling/disabling that emulation. These devices always return *DISABLED*.

Returns *DISABLED* if disabled or not available, *ENABLED* if enabled.

Return type *MiddleEmulationState*

class libinput.device.**DeviceConfigDwt**(**args*)

Disable-while-typing configuration methods.

is_available()

Check if this device supports configurable disable-while-typing feature.

This feature is usually available on built-in touchpads and disables the touchpad while typing. See Disable-while-typing for details.

Returns `False` if this device does not support disable-while-typing, or `True` otherwise.

Return type bool

set_enabled(*state*)

Enable or disable the disable-while-typing feature.

When enabled, the device will be disabled while typing and for a short period after. See Disable-while-typing for details.

Note: Enabling or disabling disable-while-typing may not take effect immediately.

Parameters **state** (`DwtState`) – *DISABLED* to disable disable-while-typing, *ENABLED* to enable.

Returns A config status code. Disabling disable-while-typing on a device that does not support the feature always succeeds.

Return type *ConfigStatus*

enabled

Check if the disable-while typing feature is currently enabled on this device.

If the device does not support disable-while-typing, this property is *DISABLED*.

Returns *DISABLED* if disabled, *ENABLED* if enabled.

Return type *DwtState*

default_enabled

Check if the disable-while typing feature is enabled on this device by default.

If the device does not support disable-while-typing, this property is *DISABLED*.

Returns *DISABLED* if disabled, *ENABLED* if enabled.

Return type *DwtState*

class libinput.device.**DeviceConfigRotation**(**args*)

Rotation configuration methods.

is_available()
Check whether a device can have a custom rotation applied.

Returns `True` if a device can be rotated, `False` otherwise.

Return type bool

set_angle(*degrees_cw*)
Set the rotation of a device in degrees clockwise off the logical neutral position.

Any subsequent motion events are adjusted according to the given angle.

The angle has to be in the range of [0, 360] degrees, otherwise this method returns *INVALID*. If the angle is a multiple of 360 or negative, the caller must ensure the correct ranging before calling this method.

libinput guarantees that this method accepts multiples of 90 degrees. If a value is within the [0, 360] range but not a multiple of 90 degrees, this method may return *INVALID* if the underlying device or implementation does not support finer-grained rotation angles.

The rotation angle is applied to all motion events emitted by the device. Thus, rotating the device also changes the angle required or presented by scrolling, gestures, etc.

Parameters **degrees_cw** (*int*) – The angle in degrees clockwise.

Returns A config status code. Setting a rotation of 0 degrees on a device that does not support rotation always succeeds.

Return type *ConfigStatus*

angle
The current rotation of a device in degrees clockwise off the logical neutral position.

If this device does not support rotation, the return value is always 0.

Returns The angle in degrees clockwise.

Return type int

default_angle
The default rotation of a device in degrees clockwise off the logical neutral position.

If this device does not support rotation, the return value is always 0.

Returns The default angle in degrees clockwise.

Return type int

CHAPTER 6

Misc. objects

6.1 TabletTool

class libinput.define.**TabletTool**(*htablettool*, *libinput*)

An object representing a tool being used by a device with the *TABLET_TOOL* capability.

Tablet events generated by such a device are bound to a specific tool rather than coming from the device directly. Depending on the hardware it is possible to track the same physical tool across multiple *Device* instances, see Tracking unique tools.

As long as a reference to a *TabletTool* is kept, multiple instances will compare equal if they refer to the same physical tool and the hardware supports it.

type

The tool type of a tool object.

See Vendor-specific tablet tool types for details.

Returns The tool type for this tool object.

Return type *TabletToolType*

tool_id

The tool ID of a tool object.

If nonzero, this number identifies the specific type of the tool with more precision than the type returned in *type*, see Vendor-specific tablet tool types. Not all tablets support a tool ID.

Tablets known to support tool IDs include the Wacom Intuos 3, 4, 5, Wacom Cintiq and Wacom Intuos Pro series.

Returns The tool ID for this tool object or 0 if none is provided.

Return type int

has_pressure()

Return whether the tablet tool supports pressure.

Returns `True` if the axis is available, `False` otherwise.

Return type bool

has_distance()
Return whether the tablet tool supports distance.

Returns `True` if the axis is available, `False` otherwise.

Return type bool

has_tilt()
Return whether the tablet tool supports tilt.

Returns `True` if the axis is available, `False` otherwise.

Return type bool

has_rotation()
Return whether the tablet tool supports z-rotation.

Returns `True` if the axis is available, `False` otherwise.

Return type bool

has_slider()
Return whether the tablet tool has a slider axis.

Returns `True` if the axis is available, `False` otherwise.

Return type bool

has_wheel()
Return whether the tablet tool has a relative wheel.

Returns `True` if the axis is available, `False` otherwise.

Return type bool

has_button(*button*)
Check if a tablet tool has a specified button.

Parameters **button** (*int*) – Button to check for. See `input.h`.

Returns `True` if the tool supports this button, `False` if it does not.

Return type bool

is_unique()
Return `True` if the physical tool can be uniquely identified by libinput, or `False` otherwise.

If a tool can be uniquely identified, keeping a reference to the tool allows tracking the tool across proximity out sequences and across compatible tablets. See Tracking unique tools for more details.

Returns `True` if the tool can be uniquely identified, `False` otherwise.

Return type bool

serial
The serial number of a tool.

If the tool does not report a serial number, this method returns zero. See Tracking unique tools for details.

Returns The tool serial number.

Return type int

6.2 TabletPadModeGroup

class libinput.define.**TabletPadModeGroup**(*hmodegroup*, *libinput*)

A mode on a tablet pad is a virtual grouping of functionality, usually based on some visual feedback like LEDs on the pad.

The set of buttons, rings and strips that share the same mode are a "mode group". Whenever the mode changes, all buttons, rings and strips within this mode group are affected. See Tablet pad modes for detail.

Most tablets only have a single mode group, some tablets provide multiple mode groups through independent banks of LEDs (e.g. the Wacom Cintiq 24HD). libinput guarantees that at least one mode group is always available.

index

The returned number is the same index as passed to *tablet_pad_get_mode_group()*.

For tablets with only one mode this number is always 0.

Returns The numeric index this mode group represents, starting at 0.

Return type int

num_modes

Query the mode group for the number of available modes.

The number of modes is usually decided by the number of physical LEDs available on the device. Different mode groups may have a different number of modes. Use *mode* to get the currently active mode.

libinput guarantees that at least one mode is available. A device without mode switching capability has a single mode group and a single mode.

Returns The number of modes available in this mode group.

Return type int

mode

The current mode this mode group is in.

Returns The numeric index of the current mode in this group, starting at 0.

Return type int

has_button(*button*)

Devices without mode switching capabilities return `True` for every button.

Parameters **button** (*int*) – A button index, starting at 0.

Returns `True` if the given button index is part of this mode group or `False` otherwise.

Return type bool

has_ring(*ring*)

Devices without mode switching capabilities return `True` for every ring.

Parameters **ring** (*int*) – A ring index, starting at 0.

Returns `True` if the given ring index is part of this mode group or `False` otherwise.

Return type bool

has_strip(*strip*)

Devices without mode switching capabilities return `True` for every strip.

Parameters **strip** (*int*) – A strip index, starting at 0.

Returns `True` if the given strip index is part of this mode group or `False` otherwise.

Return type bool

button_is_toggle(*button*)

The toggle button in a mode group is the button assigned to cycle to or directly assign a new mode when pressed.

Not all devices have a toggle button and some devices may have more than one toggle button. For example, the Wacom Cintiq 24HD has six toggle buttons in two groups, each directly selecting one of the three modes per group.

Devices without mode switching capabilities return `False` for every button.

Parameters **button** (*int*) – A button index, starting at 0.

Returns `True` if the button is a mode toggle button for this group, or `False` otherwise.

Return type bool

CHAPTER 7

Constants and Enumerations

class libinput.constant.**LogPriority**(**args*, ***kwds*)

DEBUG = 10

INFO = 20

ERROR = 30

class libinput.constant.**ContextType**(**args*, ***kwds*)

PATH = 1

UDEV = 2

class libinput.constant.**EventType**(**args*, ***kwds*)

NONE = 0

DEVICE_ADDED = 1

DEVICE_REMOVED = 2

KEYBOARD_KEY = 300

POINTER_MOTION = 400

POINTER_MOTION_ABSOLUTE = 401

POINTER_BUTTON = 402

POINTER_AXIS = 403

TOUCH_DOWN = 500

TOUCH_UP = 501

TOUCH_MOTION = 502

TOUCH_CANCEL = 503

TOUCH_FRAME = 504

TABLET_TOOL_AXIS = 600

TABLET_TOOL_PROXIMITY = 601

TABLET_TOOL_TIP = 602

TABLET_TOOL_BUTTON = 603

TABLET_PAD_BUTTON = 700

TABLET_PAD_RING = 701

TABLET_PAD_STRIP = 702

GESTURE_SWIPE_BEGIN = 800

GESTURE_SWIPE_UPDATE = 801

GESTURE_SWIPE_END = 802

GESTURE_PINCH_BEGIN = 803

GESTURE_PINCH_UPDATE = 804

GESTURE_PINCH_END = 805

SWITCH_TOGGLE = 900

is_device()
Macro to check if this event is a *DeviceNotifyEvent*.

is_keyboard()
Macro to check if this event is a *KeyboardEvent*.

is_pointer()
Macro to check if this event is a *PointerEvent*.

is_touch()
Macro to check if this event is a *TouchEvent*.

is_tablet_tool()
Macro to check if this event is a *TabletToolEvent*.

is_tablet_pad()
Macro to check if this event is a *TabletPadEvent*.

is_gesture()
Macro to check if this event is a *GestureEvent*.

is_switch()
Macro to check if this event is a *SwitchEvent*.

class libinput.constant.**DeviceCapability**(**args*, ***kwds*)

KEYBOARD = 0

POINTER = 1

TOUCH = 2

TABLET_TOOL = 3

TABLET_PAD = 4

GESTURE = 5

SWITCH = 6

class libinput.constant.**KeyState**(**args*, ***kwds*)

RELEASED = 0

PRESSED = 1

class libinput.constant.**Led**(**args*, ***kwds*)

NUM_LOCK = 1

CAPS_LOCK = 2

SCROLL_LOCK = 4

class libinput.constant.**ButtonState**(**args*, ***kwds*)

RELEASED = 0

PRESSED = 1

class libinput.constant.**PointerAxis**(**args*, ***kwds*)

SCROLL_VERTICAL = 0

SCROLL_HORIZONTAL = 1

class libinput.constant.**PointerAxisSource**(**args*, ***kwds*)

NONE = 0

WHEEL = 1

FINGER = 2

CONTINUOUS = 3

WHEEL_TILT = 4

class libinput.constant.**TabletPadRingAxisSource**(**args*, ***kwds*)

UNKNOWN = 1

FINGER = 2

class libinput.constant.**TabletPadStripAxisSource**(**args*, ***kwds*)

UNKNOWN = 1

FINGER = 2

class libinput.constant.**TabletToolType**(**args*, ***kwds*)

PEN = 1

ERASER = 2

BRUSH = 3

PENCIL = 4

AIRBRUSH = 5

MOUSE = 6

LENS = 7

class libinput.constant.**TabletToolProximityState**(**args*, ***kwds*)

OUT = 0

IN = 1

class libinput.constant.**TabletToolTipState**(**args*, ***kwds*)

UP = 0

DOWN = 1

class libinput.constant.**SwitchState**(**args*, ***kwds*)

OFF = 0

ON = 1

class libinput.constant.**Switch**(**args*, ***kwds*)

LID = 1

class libinput.constant.**ConfigStatus**(**args*, ***kwds*)

SUCCESS = 0

UNSUPPORTED = 1

INVALID = 2

class libinput.constant.**TapState**(**args*, ***kwds*)

DISABLED = 0

ENABLED = 1

class libinput.constant.**TapButtonMap**(**args*, ***kwds*)

LRM = 0

LMR = 1

class libinput.constant.**DragState**(**args*, ***kwds*)

DISABLED = 0

ENABLED = 1

class libinput.constant.**DragLockState**(**args*, ***kwds*)

DISABLED = 0

ENABLED = 1

class libinput.constant.**SendEventsMode**(**args*, ***kwds*)

ENABLED = 0

DISABLED = 1

DISABLED_ON_EXTERNAL_MOUSE = 2

class libinput.constant.**AccelProfile**(**args*, ***kwds*)

NONE = 0

FLAT = 1

ADAPTIVE = 2

class libinput.constant.**ClickMethod**(**args*, ***kwds*)

NONE = 0

BUTTON_AREAS = 1

CLICKFINGER = 2

class libinput.constant.**MiddleEmulationState**(**args*, ***kwds*)

DISABLED = 0

ENABLED = 1

class libinput.constant.**ScrollMethod**(**args*, ***kwds*)

NO_SCROLL = 0

SCROLL_2FG = 1

EDGE = 2

ON_BUTTON_DOWN = 4

class libinput.constant.**DwtState**(**args*, ***kwds*)

DISABLED = 0

ENABLED = 1

CHAPTER 8

Contributors

- Thanks to Peter Hutterer for his advice on designing a pythonic API.

Python Module Index

l

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

W