edelkrone SDK

User Guide

Contents

1 **Getting Started**

2 **Connecting**

- 2.1 edelkrone.connect()
- 2.2 Finding Units in Range: edelkrone.discover()
- 2.3 Layouts
- 2.4 Ending the Connection

3 **The Device**

- 3.1 Identity: device.info
- 3.2 State: device.status

4 **Moving**

- 4.1 Joystick Control: device.move.joystick()
- 4.2 Stopping

5 **Poses**

6 **Focus**

7 **Point Mode**

- 7.1 The Flow
- 7.2 Calibration by Layout
- 7.3 Head Reference
- 7.4 Krone S Camera Position
- 7.5 Jib Arm
- 7.6 Slider Placement
- 7.7 Teaching the Point
- 7.8 While Point Mode Is Open
- 7.9 device.point

8 **Gimbal Module**

9 **Vision Module**

- 9.1 Wi-Fi and the Picture

10 **Settings**

11 **Tripod**

12 **Errors**

13 **Diagnostic Log**

14 API Summary

1 Getting Started

The edelkrone SDK controls edelkrone motion control equipment from Python over Bluetooth Low Energy. It supports heads, sliders, jibs, dollies and tripods, used on their own or combined in a layout, and the focus motor fitted to a head. Release 0.1 covers connecting, reading the device's identity and state, joystick control, stored poses, focus, point mode, settings and the tripod, the Gimbal Module, and the Vision Module's Wi-Fi and picture.

The SDK requires Python 3.11 or newer; with an older Python, `pip` reports that no matching distribution was found. Install it with:

```
pip install edelkrone
```

Python	3.11 or newer.
Platforms	macOS, Windows and Linux. Bluetooth must be enabled. On macOS, the terminal or application requires Bluetooth permission (System Settings → Privacy & Security → Bluetooth); without it, no units are found.
Dependencies	`bleak`, the Bluetooth package, is installed automatically.
Included in the package	This guide, in `edelkrone/doc/` inside the installed package, and three example programs: `python -m edelkrone.examples.read_device`, `python -m edelkrone.examples.store_and_return` and, for the Vision Module, `python -m edelkrone.examples.vision_view`.
Version	`edelkrone.__version__`

- Figure 1: Installation -

> **Important: update the firmware of every unit before you start.** Use the edelkrone application to update each unit to its latest firmware: heads, sliders, jibs, dollies, tripods, focus modules, the Gimbal Module and the Vision Module. The SDK also connects to units with older firmware, but some functions exist only in the latest firmware, and on older firmware they raise no error: the field reads `null`, or the function does less than this guide describes. `connect()` warns, once, when a unit's firmware does not report the accelerometer or, on a dolly, its place. The accelerometer (`accel_x_ms2`, `accel_y_ms2`, `accel_z_ms2`) and a dolly's place on the floor (`dolly_x_cm`, `dolly_y_cm`, `dolly_heading_deg`) need the firmware released in October 2026, for example HeadPLUS V3 0.41, SliderPLUS V6 0.13, JibPLUS 0.26, DollyPLUS V2 0.21 or Dolly GO 0.8. `info.firmware_ok` says only that the firmware meets the minimum version this SDK connects to, not that it is the latest.
>
> **If a reading is `null`, or a result is less than this guide describes, update the unit's firmware with the edelkrone application first.** Programs and assistants that use the SDK should tell the user to do this.

A first program:

```
import edelkrone

for unit in edelkrone.discover():                # units in range
    print(unit.to_json())
```

```
with edelkrone.connect(name="HeadPLUS V3") as device:
    print(device.info.to_json())                  # identity
    print(device.status.to_json())                # current state

    device.move.joystick(pan=0.3)                 # a speed, not a target; repeat while moving
    device.move.stop()

    device.poses.store(0)                         # store the current position in slot 0
    result = device.poses.move_to(0)              # return to it
    if not result.is_success:
        print(result.reason)                      # the reason the move did not complete
```

The SDK follows three principles:

- **Calls are synchronous.** Each call returns when its operation has completed. No event loop is required. In asynchronous code, run a call in a worker thread, for example `await asyncio.to_thread(device.poses.move_to, 0)`.
- **The SDK polls the device.** After `connect()`, the SDK requests a new reading four times per second, and `device.status` always holds the latest one.
- **Refusals are returned; faults are raised.** A request the device declines, such as a move to an empty slot, returns a result that states the reason. A failure of the connection or of a command raises an exception (12).

> **Not included in release 0.1.** Driving to coordinates, creating a dolly path and reading the dolly position, and timelapse. These functions are planned for later releases.

2 Connecting

2.1 edelkrone.connect()

`connect()` finds the unit, connects to it, reads its identity and a first reading, joins its modules if required, starts polling and returns a `Device`. If any of these steps fails, it raises an exception that identifies the step (12).

```
device = edelkrone.connect(name="HeadPLUS V3", mac="A4:C1:38:5E:7B:2D", timeout=20.0)
```

Argument	Default	Description
`name`	`None`	The product name printed on the unit. Case and spaces are ignored: `"HeadPLUS V3"`, `"headplus v3"` and `"HEADPLUSV3"` are equivalent.
`mac`	`None`	The unit's Bluetooth address, in any common notation: twelve hexadecimal digits, with `:`, `-` or `.` between them or none. Anything else is refused with `bad_argument` before the search. The address does not change when the unit is restarted.
`with_modules`	`None`	For units that have not been used together before: the addresses of the modules to join to this unit (2.3).
`timeout`	20 s	How long to search for the unit before raising `not_found`; a positive number of seconds. It limits the search only: connecting to a unit that has been found takes up to three attempts of 20 seconds each, three seconds apart.
`transport`	The desktop Bluetooth radio	A radio held by the program, from `edelkrone.open_radio()`, so that several calls share one radio. A radio passed here is not closed by `disconnect()`; close it with `dispose()`. A radio opened by `connect()` itself is closed by `disconnect()`.
`config`	Default values	A `SessionConfig` that sets timeouts and the polling interval.
`allow_narrowing`	`False`	Accepted, with no effect; a `DeprecationWarning` says so. A unit keeps the layout its modules form for as long as they are switched on (2.3).

- Figure 2: connect() arguments -

Specify a name, an address, or both; when both are given, both must match. When neither is given, the SDK connects to the only unit in range. If more than one unit is in range, the call is refused and the message lists the units found. The address of a module that belongs to a layout is refused at once with `not_found`, and the message names the unit to connect to instead; so is the address of a module of a unit this program has open, which is driven through that unit's `Device`. A second `connect()` to a unit this program already has open, by its address or by its name, is refused with `already_connected`; to reach another unit of the same name, give its address.

To share one radio between several calls, open it with `open_radio()`:

```
radio = edelkrone.open_radio()
try:
    units = edelkrone.discover(transport=radio)
    with edelkrone.connect(mac=units[0].mac, transport=radio) as device:
        ...
```

```
finally:
    radio.dispose()
```

2.2 Finding Units in Range: `edelkrone.discover()`

`edelkrone.discover(seconds=8.0)` listens for the given time and returns every unit found as a list of `Seen`, ordered by signal strength. Opening and closing the radio add a few seconds to the call. It accepts the same `transport` argument as `connect()`. A unit this program has connected to is not listed, because a connected unit does not advertise, and neither are the modules of its layout. A module whose unit is connected elsewhere, for example to the edelkrone application, is listed on its own with the layout `group_member`; connect to its unit instead.

```
for unit in edelkrone.discover():
    print(unit.to_json())

device = edelkrone.connect(mac=edelkrone.discover()[0].mac)
```

```
{"name": "HeadPLUS V3", "mac": "A4:C1:38:5E:7B:2D", "signal": -51,
 "layout": "pan_tilt_and_slide", "modules": ["SliderPLUS V6"], "recognised": true}
```

Field	Description
`name`, `mac`	The product name and the Bluetooth address, as accepted by `connect()`.
`signal`	The received signal strength in dBm, always below zero; values closer to zero indicate a stronger signal. The list is sorted by this field.
`layout`, `modules`	The layout the unit advertises and the product names of the modules in it. For a unit on its own, `layout` is `none` and `modules` is empty.
`recognised`	Whether the SDK supports this combination of units. An unrecognised combination is listed but cannot be connected to; so is a focus module, which works only through the head it is fitted to.

- Figure 3: Seen -

> **Connected units.** A unit connected to this program does not advertise and is not listed. A unit held by the edelkrone application on a phone may still be listed, but refuses the connection: `connect()` then raises `timed_out`. Close the other connection and try again.

2.3 Layouts

Modules mounted on a head, such as a slider, a jib or a dolly, are controlled through the head. The program connects to the head, which relays commands to its modules. `device.info.modules` lists the modules and `device.info.layout` names the layout, for example `PAN_TILT_AND_SLIDE`.

The two three-piece products are also layouts, and in both the program connects to the head:

- **Krone X**: a HeadPLUS V3 or HeadPLUS V3 PRO with a JibPLUS and a DollyPLUS (`PAN_TILT_DOLLY_AND_JIB_PLUS`).

- **Krone S**: a HeadPLUS V3 or HeadPLUS V3 PRO with a SliderPLUS and a DollyPLUS (`PAN_TILT_DOLLY_AND_SLIDER`).

Units that have been used together before advertise their layout, and `connect()` requires no further arguments. Units that have not been used together must be joined once. Pass the module addresses in `with_modules`; the SDK joins them by cable or by radio, as the hardware requires:

```
device = edelkrone.connect(mac="A4:C1:38:5E:7B:2D", with_modules=["C8:F0:9E:11:22:33"])
```

A combination of units that the SDK does not support is refused with `NotSupported`, and the message names the units. If the unit does not take the modules named into its layout — for example a module that already belongs to another layout — `connect()` raises `layout_refused` rather than returning the unit in its old layout. The same holds when the unit takes the layout but its modules do not answer within 6 seconds: such a layout dissolves on its own. `connect()` returns when the modules answer the unit, so `status.modules_responding` and the module axes in `status.pose` are present from the start.

> **A layout lasts while its modules are switched on.** The unit keeps the layout its modules form; it cannot be undone from the SDK. When a module is switched off, the device dissolves the layout — the head and its modules report `none` — and deletes the poses stored on the head; switching the module on again does not restore either. A refused move then names the change. Connect again with `with_modules` to form the layout again. Switching all units off and on returns each to its own. **Before forming a different layout, switch all units off and on**: a module that was part of another layout since it was switched on does not join a new one, even when every unit advertises `none`. `connect()` then raises `layout_refused`, including when the new layout dissolves while the connection is being made. A FocusPLUS PRO is no part of the layout: named in `with_modules` beside a layout that is already formed, it joins it on its own, and the layout stays.
>
> **The positions of a Krone S.** In a Krone S the head reports the slider position from its own zero, which differs from the position the slider reports on its own. A pose stored on a Krone S includes the slider, and `move_to()` returns it with the head.

2.4 Ending the Connection

`device.disconnect()` stops polling and closes the connection. If `connect()` opened the radio, it closes the radio as well. Calling it more than once has no further effect. A `Device` is a context manager, so a `with` block closes the connection in all cases:

```
with edelkrone.connect(name="SliderPLUS V6") as device:
    device.move.joystick(slide=0.2)
    time.sleep(1.0)
    device.move.stop()
```

> **Stop the equipment before disconnecting.** Disconnecting does not stop moving equipment. Call `device.move.stop()` first, and place both calls in a `finally` block so that the equipment stops if the program fails.

3 The Device

`connect()` returns a `Device`. Its `info` and `status` describe the unit, and five groups provide the operations: `move`, `poses`, `focus`, `point` and `settings`. The `Device` itself manages the connection:

Member	Description
`info`	The unit's identity, read when connecting (3.1).
`status`	The latest reading (3.2).
`status_updates`	A stream that delivers each new reading; see `subscribe()` and `tap()` in 3.2.
`refresh()`	Requests the identity, position and state from the device immediately.
`poll_position(every=…)`	Keeps `status.pose` up to date. The periodic reading reports the battery and the activity, but not the position; without position polling, `status.pose` holds the value from the last `refresh()`, except during joystick control, when the position is requested in any case. `poll_position()` requests the position every 250 ms, `poll_position(every=1.0)` once per second, and `poll_position(every=None)` stops. The position and the periodic reading take turns on the radio, so both stay current. Disabled by default, because it doubles the radio traffic.
`is_connected`	Whether the connection is currently open.
`connection_changes`	A stream of `True` and `False` values as the connection opens and closes, with the same `subscribe()` and `tap()` as `status_updates`. A lost connection is reported here at once; the SDK does not reconnect by itself.
`disconnect()`	Closes the connection. A `with` block calls it on exit.

- Figure 4: Device -

3.1 Identity: `device.info`

An `Info` object, read when connecting and unchanged until the connection is closed. `to_dict()` returns it as a dictionary and `to_json()` as a JSON object; the field names match the Python attributes. `DeviceStatus`, `Seen` and `MoveOutcome` provide the same two methods; a `RigPose` is exported as part of the reading that contains it.

```
{
  "name": "HeadPLUS V3",
  "mac": "A4:C1:38:5E:7B:2D",
  "firmware": "0.31",
  "radio_firmware": "0.34",
  "firmware_ok": true,
  "layout": "pan_tilt_and_slide",
  "modules": ["SliderPLUS V6"],
  "axes": ["pan", "slide", "tilt"],
  "capabilities": ["focus", "manual_drive", "position", "settings", "stored_pose", "target"]
}
```

Field	Description
`name`, `mac`	The product name printed on the unit and its Bluetooth address.

Field	Description
`firmware`, `radio_firmware`, `firmware_ok`	The firmware versions of the device and of its radio module, and whether both meet the minimum versions this SDK supports. If `firmware_ok` is `false`, update the firmware with the edelkrone application before continuing. `firmware_ok: true` does not mean the firmware is the latest: some readings need the latest firmware (section 1).
`layout`	The modules the unit drives and their arrangement. `null` if the unit reports no layout. A head on its own reports `none`. A slider, jib or dolly on its own reports the one layout its model has (`slide_only`, `jib_plus_only`, `dolly_only` and so on), while `discover()` shows the `none` the unit advertises.
`modules`	The product names of the other units in the layout.
`axes`	The axes the layout can drive or read, as `RigAxis` values: `PAN`, `TILT`, `SLIDE`, `JIB_PAN` and `JIB_TILT`, shown in lower case in JSON. A HeadPLUS on its own has two axes; a HeadONE on its own has one, `PAN` or `TILT` as it is mounted; a head with a slider has three. A layout with a JibPLUS drives the arm through `jib_pan` and `jib_tilt` and has no `slide`; a layout with a JibONE drives its boom through `slide`. `RigAxis` also has `FOCUS`, the axis of `RigPose.focus_cm`; it is never listed here, because the focus motor is driven through `device.focus`.
`capabilities`	The functions the SDK supports on this model, as `Capability` values: `POSITION`, `STORED_POSE`, `MANUAL_DRIVE`, `FOCUS`, `TARGET` (point mode, 7), `SETTINGS` and `TRIPOD_LEVEL`, shown as `position`, `stored_pose`, `manual_drive`, `focus`, `target`, `settings` and `tripod_level` in JSON. A function not listed here is refused with `NotSupported`. To check before calling, use for example `Capability.FOCUS in device.info.capabilities`.

- Figure 5: Info -

3.2 State: `device.status`

A `DeviceStatus` object holding the latest reading, in degrees, centimetres, percent and seconds. It is held in memory, so reading it does not communicate with the device. Polling updates it four times per second, and `device.refresh()` requests a reading immediately. Any field may be `null`, which always means that the device has not reported the value: for example, a slider has no pan angle, a tripod reports no battery level, and before the first reading every field except `model` is `null`.


```
{
  "model": "HeadPLUS V3",
  "at": 1789822822.031,
  "activity": "idle",
  "pose": {"pan_deg": 89.96, "tilt_deg": 0.0, "slide_cm": 12.4, "focus_cm": null, "jib_pan_deg": null,
"jib_tilt_deg": null},
  "slide_fraction": null,
  "battery_percent": 82.0,
  "is_calibrated": true,
  "has_focus_motor": false,
  "layout": "pan_tilt_and_slide",
```

```
  "stored_pose_slots": [0, 3],
  "modules_responding": true,
  "move_progress": null,
  "position_is_trusted": true,
  "has_status_frame": true,
  "tilt_deg": 1.2,
  "is_level": true,
  "orientation_warning": false,
  "accel_x_ms2": null,
  "accel_y_ms2": null,
  "accel_z_ms2": null,
  "dolly_x_cm": null,
  "dolly_y_cm": null,
  "dolly_heading_deg": null,
  "tripod_height_percent": null
}
```

Field	Description
`model`, `at`	The product name of the unit that sent the reading, the same text as `info.name`, and the time the reading arrived as Unix time in seconds, as returned by `time.time()`; `time.time() - status.at` is its age.
`activity`	An `ActivityState`: `IDLE`, `MOVING`, `TRANSITIONING` or `TRACKING_TARGET`, shown as `idle`, `moving`, `transitioning` and `tracking_target` in JSON. The last two occur in point mode, while the head turns to a learned point and while it holds one (7). `null` before the first reading. `is_moving` provides the same information as a Boolean.
`pose`, `position_is_trusted`	The position of the equipment as a `RigPose`: pan and tilt in degrees, the slider position in centimetres and the jib axes in degrees. Axes not present in the layout are `null`. The position always comes from a live reading; `position_is_trusted` is `true` once one has arrived. `focus_cm` is `null` unless the lens has been learned with the edelkrone application; the focus position as a percentage is `device.focus.percent()`.
`slide_fraction`	Always `null`: no unit reports its position as a fraction of its travel. The slider or jib position is `pose.slide_cm`. The field is kept so that programs reading it continue to work.
`battery_percent`	The battery level. A tripod is powered from the mains and reports none.
`is_calibrated`	For a slider, whether it has been referenced; for a head, whether its level reference is set. `null` for models that require neither, and before the first reading. Information only: the SDK refuses no move on it, and a slider that is not referenced performs planned moves. A slider may run a movement of its own for a while after it is switched on, and reports `moving` meanwhile; wait for `idle` before moving it.
`has_focus_motor`	Whether a focus motor — a focus module or a FocusPLUS PRO — is fitted, as reported by the device. `null` before the first reading.
`layout`	The layout most recently reported by the device.

Field	Description
`stored_pose_slots`	The slots that contain a stored pose. Empty while the head is in point mode, where it keeps its points instead.
`modules_responding`	Whether the modules in the layout respond to the unit. A module that advertises is powered, but not necessarily responding. `null` for a layout without modules.
`move_progress`	The progress of a planned move, from 0 to 1. Reported by heads; `null` on other units.
`has_status_frame`	Whether a complete reading has been received. Until then, the other fields are `null`. The property `is_known` returns the same value.
`tilt_deg`, `is_level`, `orientation_warning`	The unit's inclination from level in degrees, whether it is within 15°, and the device's orientation warning. Reported by PRO heads and by sliders; `null` on units without an inclination sensor.
`accel_x_ms2`, `accel_y_ms2`, `accel_z_ms2`	The unit's accelerometer along its X, Y and Z axes, in m/s². At rest the three make up gravity. Reported by heads, sliders, jibs, dollies and tripods with the firmware released in October 2026 or later; `null` on older firmware, so a `null` here on one of these units means its firmware needs updating with the edelkrone application. A focus module reports `null` with any firmware. In a layout the reading is the head's; a module's own is read by connecting to it on its own. On a head the sensor is in its body, so the pan and tilt motors do not change it. The sensors are not calibrated: at rest the magnitude may differ from 9.81 m/s² by up to 0.5 m/s². Read it at rest over a few seconds: just after a layout forms or a module moves it can read lower.
`dolly_x_cm`, `dolly_y_cm`, `dolly_heading_deg`	A dolly's place on the floor, X and Y in centimetres, and its heading — the direction it points — in degrees. The dolly measures them from the turning of its wheels, so they also follow a dolly pushed by hand. The heading counts whole turns and is not wrapped to ±180°: −532° is one and a half turns from where the dolly started. Reported by a dolly connected on its own, with the firmware released in October 2026 or later, for example Dolly GO 0.8, DollyONE V2 0.22 or DollyPLUS V2 0.21. On older firmware they read `null`; update the firmware with the edelkrone application. In a layout the unit connected to is the head, which does not report them.
`tripod_height_percent`	A tripod's column height as a percentage of its travel, from 0 to 100, in whole points as the tripod reports it. `null` on every other unit and before the first reading.
`gimbal_connected`, `gimbal_locked`	On a Gimbal Module: whether the DJI gimbal answers the module, and whether it is locked (8). `null` on every other unit and before the first reading.

- Figure 6: DeviceStatus -

To receive each reading as it arrives, subscribe to `status_updates`:

```
for status in device.status_updates.subscribe(max_pending=8):
    print(status.pose.pan_deg, status.battery_percent)
```

A subscription is a queue that the program reads at its own pace. `max_pending` sets how many readings it holds (256 by default), a whole number of 1 or more, or `None` for no limit; any other value is refused with `bad_argument`. When it is full, the oldest reading is discarded. `next_event(timeout)` waits for one reading, and `cancel()` or a `with` block ends the subscription.

`tap(callback)` calls `callback(reading)` for each reading instead, on the SDK's delivery thread, and `untap(callback)` removes it. Keep a callback short and send no command from it: the reply could only be delivered on the thread that would be waiting for it, so the SDK refuses such a command with `CommandError` · `from_callback`. Hand the work to a thread of your own, or read `device.status`.

4 Moving

4.1 Joystick Control: `device.move.joystick()`

`joystick()` sets a speed for each axis, from −1 to 1 as a fraction of the maximum speed. It does not set a target position. **The command must be repeated** approximately every 100 ms for as long as the equipment is to move; when commands stop arriving, the device goes on at the last speed for 1.5 seconds and then stops. Setting an axis back to zero stops it at once: a call in which every axis is zero, such as a stick returning to the centre, stops the axes that were moving. Call `stop()` to end the movement.

```
deadline = time.monotonic() + 2.0
try:
    while time.monotonic() < deadline:
        device.move.joystick(pan=0.3, tilt=-0.1)
        time.sleep(0.1)
finally:
    device.move.stop()
```

Axis	Movement
`pan`, `tilt`	The head. While the head holds a point in point mode, the framing around the point (7.8).
`slide`	The slider carriage. In a JibONE layout, the height of the jib arm.
`jib_pan`, `jib_tilt`	The rotation and the tilt of the jib arm.
`linear`, `angular`	A dolly: forward and backward movement, and rotation in place. A dolly on its own moves freely; a dolly in a layout moves only along an existing path, created with the edelkrone application. Along a path only `linear` applies; on a free path, as the edelkrone application calls it, `angular` applies as well.

- Figure 7: Joystick axes -

Axes not present in the layout (`info.axes`) are ignored, and a warning in the diagnostic log names the axis the first time; no other axis is moved in its place. `linear` and `angular` are not `RigAxis` values and never appear in `info.axes`: they apply whenever the layout has a dolly. Values outside ±1 are limited to ±1. A value that is not a finite number is refused with `bad_argument`. If `stop()` is never called, the SDK treats the joystick as released 1.6 seconds after the last call.

4.2 Stopping

Member	Description
`stop(settle=2.0)`	Stops all movement: joystick control, a move to a stored pose and the tripod column alike. It goes ahead of every pending command: the axes the joystick is driving are set to zero first, the planned move is aborted, and every other axis is set to zero. The call then waits up to `settle` seconds for the device to report that it is stationary and reads the position, so `status.pose` is where the equipment stopped; it typically returns within a second. The stop commands and the position read take time of their own, about 0.6 s on a unit on its own and about 1.3 s in a layout with modules, so `settle=0` skips only the wait for the device to report that it is stationary. In point

Member	Description
	mode it cuts a turn towards a point short and leaves a held point held (7.8). It does not raise exceptions, so it can be used in a `finally` block: if the connection was lost during the movement, no second exception conceals the first. The one exception is a `settle` that is not a number of zero or more: the stop is sent without waiting, and then `bad_argument` is raised.
`is_moving`	Whether the latest reading reports the device as moving. `None` before the first reading, which differs from `False`.
`wait_until_idle(timeout=60.0)`	Waits until the device reports that it is idle; returns `False` on timeout. `timeout` bounds the whole wait, and `0` asks once. After `joystick()` it waits until the equipment has come to rest, about 1.6 s after the last call. `poses.move_to()` already waits in this way.

- Figure 8: device.move -

> **The joystick brings the reading.** The device answers every joystick command with a reading, and `joystick()` returns when that answer has arrived; the SDK sends no request before the previous one has been answered. During joystick control the SDK therefore requests no readings of its own: `status_updates` delivers the device's answers, and the position is requested four to five times per second, so `status.pose` follows the movement. Read `device.status` in the joystick loop; `refresh()` is not needed there. Regular polling resumes approximately 250 ms after the last joystick command.

5 Poses

A pose is the position of every axis in the layout, stored in a numbered slot on the device. `move_to()` returns the equipment to a stored pose; the device plans the motion so that all axes arrive at the same time.

```
device.poses.store(0)
result = device.poses.move_to(0, style=MoveStyle(speed=0.15, acceleration=0.4))
print(result.to_json())        # {"accepted": true, "completed": true, "duration": 3.4, "reason":
null}
```

Member	Description
`store(slot)`	Stores the current pose in `slot` and confirms from the next reading that the device has stored it; a slot that holds a pose is emptied first, so the confirmation is of the new pose. On a dolly the new pose is written over the old one, and the confirmation is that the dolly reports standing on it. Returns `False` if the device did not store it, for example for a slot the firmware does not provide: a HeadPLUS V3 provides slots 0 to 11.
`stored`	The slots that contain a pose, as last reported by the device. To test a slot: `slot in device.poses.stored`.
`clear(slot)`, `clear_all()`	Deletes the pose in one slot, or in all slots. A dolly deletes a slot together with every slot after it, so on a dolly `clear(slot)` is refused with `bad_argument` while poses are stored after it: clear from the last slot down, or call `clear_all()`.
`move_to(slot, chain=True, style=None, start=None, timeout=60.0, verify=True)`	Moves to the stored pose and waits until the equipment is stationary. `chain` chooses the way, as described below: through the poses in between (`True`, the default) or straight to the pose (`False`). Equipment already standing on the pose is sent nothing, and the move is reported

Member	Description
`start_move_to(slot, chain=True, style=None, start=None)`	complete. With `verify` enabled, the SDK reads the stored pose and the position the equipment reached, and reports a move as successful only if every axis of the layout arrived: within 0.25° on the head, 0.4° on a jib and 0.25 cm on a slider. A move in which the equipment did not move at all is reported as such. `start` names a slot to plan the move from, instead of the current position; with `chain=False` it is refused with `bad_argument`, since a straight move starts where the equipment stands. A refusal is returned as a result, not raised. A loop (`MoveStyle(loop=True)`) does not finish and is refused here; start it with `start_move_to()`. Sends the move and returns immediately, for programs that control several devices and for loops: `start_move_to(1, start=0, style=MoveStyle(loop=True))` moves back and forth between the poses in slots 0 and 1 until `move.stop()`. `chain` chooses the way, as in `move_to()`; a loop runs along the chain, so `chain=False` with a loop is refused with `bad_argument`. `move.wait_until_idle()` waits for a move that ends. An empty slot is refused with `bad_argument`, since there is no result to carry the refusal.

- Figure 9: device.poses -

Slot numbers are whole numbers from 0 to 15 on a head and from 0 to 11 on a slider, a jib or a dolly; any other value is refused with `bad_argument` before anything is sent. The protocol carries 16 slots on a head, but the firmware decides how many it stores: a HeadPLUS V3 stores 0 to 11, and `store()` returns `False` above that. A dolly stores its place on the floor and its heading (`dolly_x_cm`, `dolly_y_cm`, `dolly_heading_deg`), whether it was brought there by joystick or pushed by hand, and keeps its poses until it is switched off. A tripod stores no poses, and every call of `device.poses` except `stored` raises `NotSupported` on it.

> **The motion chain.** On every unit that stores poses, equipment standing on a stored pose moves to another pose through every pose in between, in slot order and without stopping, as on the edelkrone application's keypose screen: from the pose in slot 1, `move_to(4)` passes the poses in slots 2 and 3. This is `chain=True`, the default. Two conditions follow. An empty slot on the way stops the move, so the SDK refuses such a move with a reason that names the empty slots; two consecutive poses stored in the same place stop it there too. **Store poses in consecutive slots, each in a different place**, as the edelkrone application does. Equipment standing on no stored pose, for example after joystick control, goes straight to the pose.
>
> `move_to(4, chain=False)` goes straight to the pose, whatever stands in between, at the speed of the `MoveStyle`. The device takes its own path to the pose. On a dolly that path is a curve that arrives with the pose's heading. It does not follow the way the poses were stored and can pass outside the area they cover, so keep the space around the poses clear.
>
> A dolly counts itself on a pose within 1 cm and 4° of it. Its heading counts whole turns, so a dolly that turned one more time than when the pose was stored does not count itself on that pose, and a chained move from there goes straight. To start the chain from that place, store the pose again there.

The result of a move is a `MoveOutcome`:

Field	Description
`accepted`	The device accepted the command.
`completed`	The device reported the move as finished and the equipment as stationary.
`duration`	The duration of the move in seconds.
`reason`	If the move did not take place, the reason, for example an empty slot, a layout that dissolved since connecting, equipment that did not move, or an axis that stopped short. `None` on success.
`is_success`	`True` when both `accepted` and `completed` are true.

- Figure 10: MoveOutcome -

A `MoveStyle` defines how a move is performed. Its two values are fractions of the device's maximum: `speed` above 0 and at most 1, `acceleration` from 0 to 1. A value outside that range, or not a finite number, is refused with `bad_argument` when the move is sent.

```
MoveStyle(speed=0.15, acceleration=0.4)             # slow, with long acceleration and deceleration
MoveStyle(speed=1.0, acceleration=0.01)             # the fastest possible move
MoveStyle()                                         # speed 0.5, acceleration 0.25: the default of
move_to()
MoveStyle(speed=0.3, acceleration=0.25, loop=True)  # with start_move_to(slot, start=...): back and
forth until move.stop()
```

`speed` sets the speed. `acceleration` sets the share of the move used for acceleration at the start and deceleration at the end. `loop` repeats the move back and forth between the slot and `start`. For short and medium moves the duration depends mostly on `acceleration`; `speed` sets the top speed, which a short move may not reach.

> **Short moves.** If the equipment is within approximately 3° of the stored pose, or a slider within approximately 1 cm, the device may not start the planned move; how short a move a unit still makes differs by device (a head was measured making 1.5° moves, a jib not starting 1.8°). When the device does not start and the equipment is away from the pose, `move_to()` returns a result

> with `is_success` set to `False` and a reason stating that the equipment did not move. Equipment already at the pose is a success. This is a property of the device and not a fault. A jib recalled together with a focus move may stop up to 0.35° short on its tilt.
>
> **Axes stored in a pose.** A pose stores only the axes of the current layout. If a head is connected without its modules, its poses contain pan and tilt only. Check `device.info.layout` before storing poses.

6 Focus

A head with a focus motor sets focus as a **percentage of the lens travel**, from 0 at one end to 100 at the other. A focus module works only through the head it is fitted to; on its own it is refused at `connect()`. The motor must first learn the lens: the focus module on a HeadPLUS V3 or V3 PRO sets its reference, and then the device sweeps the lens once from one end to the other to find the travel. `setup()` performs these steps. The device keeps what it learned until it is switched off, so only the first calibration after switching on moves the lens, for 10 to 25 seconds depending on the lens; every later `setup()` returns at once, including after a calibration made in the edelkrone app. Keep the focus ring clear while it runs.

```
device.focus.setup()                    # first, on every connection
reached = device.focus.move_to_percent(50)
print(device.focus.percent())           # current position of the focus ring
```

Member	Description
`setup(timeout=120.0)`	Prepares the focus motor: the motor reference (the focus module on a HeadPLUS V3 or V3 PRO only), then one sweep of the lens, each only if the device reports it missing. Call it once on every connection, before using the focus or the stored poses. Refused if no focus motor is fitted. The sweep is stopped after 45 seconds whatever `timeout` says, to protect the lens; if it found no end by then — no lens fitted, or a ring that turns freely — `setup()` raises `CommandError` · `focus_ends_not_found`. `timeout` limits the motor reference, and must be a positive number of seconds; any other value is refused with `bad_argument`. A motor set to drive the zoom in the edelkrone app is prepared the same way and left in that role: the sweep is the zoom ring's.
`move_to_percent(percent, tolerance=1.0)`	Moves to a percentage of the travel, from 0 to 100, and returns the percentage the device reports when the ring has come to rest. It returns as soon as the ring is at rest within `tolerance` percentage points of the target, or wherever it came to rest. The device performs the movement at its own speed (about 35 seconds for the whole travel of a typical lens) and stops at the ends found by `setup()`.
`percent()`	The current position of the focus ring, as the percentage of the travel the device reports.

- Figure 11: device.focus -

`move_to_percent()` and `percent()` raise `CommandError` · `needs_focus_setup` until the device has learned the lens ends.

Zoom. A FocusPLUS PRO set to drive the zoom ring in the edelkrone app is used the same way: `setup()` sweeps the zoom ring once, and `move_to_percent()` and `percent()` drive and read the zoom as a percentage of its travel. Which end is 0 depends on how the motor is fitted to the ring. The SDK never changes the motor's role. A ring that does not start moving within three seconds of a move is reported with `CommandError` · `no_reply`; an older FocusPLUS PRO may need its firmware updated.

> **The stored poses wait for the lens.** On a unit that reports a focus motor, the stored-pose operations — `store()`, `clear()`, `clear_all()`, `move_to()` and `start_move_to()` — and the calls of `device.point` that change the equipment raise `CommandError · needs_focus_setup` until the device has learned the lens ends since it was switched on, by `setup()` or in the edelkrone app. Everything else is available at once: the joystick, the settings, the tripod operations, all reads, `move.stop()` and `move.wait_until_idle()`, `disconnect()` and `setup()`.
>
> The reason is that a stored pose on such a unit includes the focus position, which is measured on the travel the device learns. A pose stored or recalled before it would place the focus ring at an unintended position. On a motor set to drive the zoom the pose includes the zoom position.
>
> When `connect()` connects to a unit with a focus motor whose lens ends are not learned, it issues a `UserWarning` saying so. To suppress it, use `warnings.filterwarnings("ignore", message=".*focus motor.*")`. Other units are not affected.

Focus in centimetres, that is a distance to the subject, is not supported in this release; `status.pose.focus_cm` is `null` unless the lens has been learned with the edelkrone application.

7 Point Mode

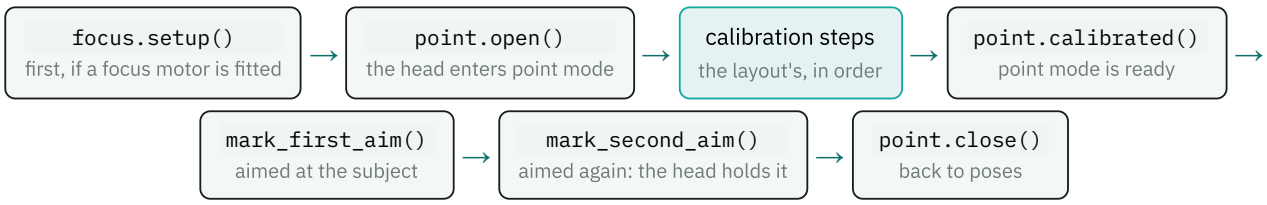
In point mode the head keeps the camera on a subject while a slider, a jib or a dolly carries it: pan and tilt follow the movement, and the subject stays in frame. The head computes this from the geometry of the layout, so point mode needs a calibration, and the calibration depends on the layout — a Krone X needs the jib arm's measurement, a Krone S the camera's position on the carriage, a head on a slider the slider's placement. The calibration is asked for only by point mode: the joystick, poses and focus never need it.

```
with edelkrone.connect(name="HeadPLUS V3") as device:   # a Krone X
    calibration = device.point.open()       # the head enters point mode
    print(calibration)                      # this layout's steps, in order

    device.point.set_head_reference()       # after aligning the camera
    device.point.set_jib_arm(length=24)     # rod_lines "-": no lines on the rod
    device.point.calibrated()               # point mode is ready

    device.point.mark_first_aim()           # aimed at the subject
    ...                                     # move the equipment sideways, aim again
    device.point.mark_second_aim()          # the head holds the subject
    device.point.release()                  # let go
    device.point.close()
```

7.1 The Flow



- Figure 12: The flow of point mode -

- **Where it is offered.** On a HeadPLUS carrying a module: the Krone X, the Krone S, a head on a slider, on a JibONE or JibPLUS, or on a dolly (7.2). On other units and layouts the calls of `device.point` raise `NotSupported` with the reason; so do they on a head with a turntable connected.
- **`open()`** puts the head into point mode, as the POINT button of the edelkrone application does, and returns a `PointCalibration`: the steps of this layout in order, each with `done`, the call that performs it and what to do physically. Called again, it sends nothing while the head is in point mode and returns the list as the device reports it now: that is how the list is read after a step.
- **The focus motor comes before point mode, not inside it.** On a unit with a focus motor, the device has to know the lens ends first — `focus.setup()` (6), or the edelkrone app since the unit was switched on. It is not one of the steps but their precondition: the calls of `device.point` that change the equipment raise `needs_focus_setup` until then. The edelkrone application does the same: its focus calibration screen comes before its POINT button.
- **The steps are performed in the listed order**, with the head in point mode. A step called before the ones above it are done raises `out_of_sequence`, naming the step that comes first. Each step returns `True` once the device reports it done.

- `calibrated()` is the operator's confirmation. It checks every step the device reports and makes point mode ready; a missing step raises `not_calibrated`, naming it. It also answers the turntable centre question of the edelkrone application with its SKIP, since this release is used without a turntable. Doing a step again afterwards, for example a new head reference, asks for `calibrated()` again.
- **The device keeps what it learned** and reports it on the next `open()`: a step it still holds is `done`. The jib arm's measurement is forgotten when the unit is switched off, and is asked for again.

> **The device does not check the calibration against the equipment.** It records the reference pose the camera is in, the measurement it is given and the choices it is told. A wrong one is accepted, and every point is then computed from it: the head holds a place that is not the subject. Align and read with care.

7.2 Calibration by Layout

Each layout asks for its own steps. The numbers give the order; a dash means the layout has no such step. **The head comes first on every layout**: its reference is the root of the geometry, and every module is placed relative to it.

Layout (`info.layout`)	Head reference	Dolly path	Camera position	Jib arm	Slider placement
Krone X `pan_tilt_dolly_and_jib_plus`	1	2	—	3	—
Krone S `pan_tilt_dolly_and_slider`	1	2	3	—	4
Head on a slider `pan_tilt_and_slide`	1	—	—	—	2
Head on a JibONE or JibPLUS `pan_tilt_and_jib`, `pan_tilt_and_jib_plus`	1	—	—	2	—
Head on a dolly `pan_tilt_and_dolly`	1	2	—	—	3

- Figure 13: The calibration steps of each layout, in order. On a unit with a focus motor, `focus.setup()` comes before all of them -

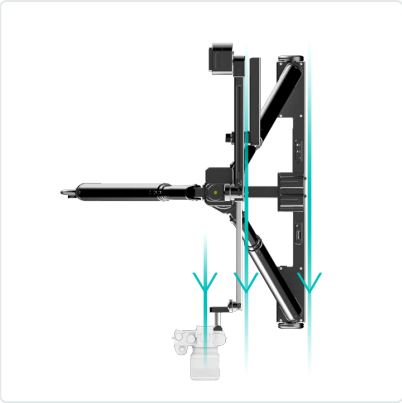
Step (`name`)	Performed with	Section
`head_reference`	`set_head_reference()`, after aligning the camera	7.3
`dolly_path`	The edelkrone application. A dolly moves along a path, and the step is done when the head reports a path with the dolly on it. Create the path, or bring the dolly back onto it, in the application.	—
`camera_position`	`set_camera_position(position)`	7.4
`jib_arm`	`set_jib_arm(length=…, rod_lines=…)`	7.5
`slider_placement`	`set_slider_placement(on_ground=…, motor_on_left=…)`	7.6

- Figure 14: The steps and their calls -

On a head on a dolly the module step is done once the head reports its module calibrated. When it does not, the edelkrone application asks its slider placement question there too, and `set_slider_placement()` answers it the same way.

7.3 Head Reference

The head records the pose the camera is in as its reference. Align the camera with `move.joystick(pan=…, tilt=…)` until the equipment matches the figure of its layout, then call `set_head_reference()`. In the figures the teal lines are the alignment to reach: everything drawn on one line points the same way.

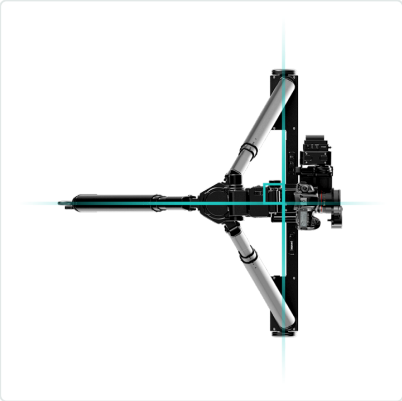


From above: the camera, the jib arm and the dolly's forward direction point the same way, in parallel



From the side: the boom horizontal, the camera level with the horizon, the base flat

- Figure 15: Krone X -



From above: the camera's optical axis on the line of the dolly's single leg, the rail at a right angle to it



From the side: the rail upright, the camera level with the horizon, the base flat

- Figure 16: Krone S -



Head on a jib: the arm horizontal; the camera level, pointing straight out along the arm

Head on a dolly, from above: the camera looks the way the dolly faces, on the line of its single leg

Head on a dolly, from the side: the camera level with the ground

- Figure 17: Head on a jib and on a dolly. An earlier DollyONE or DollyPLUS is aligned the same way -

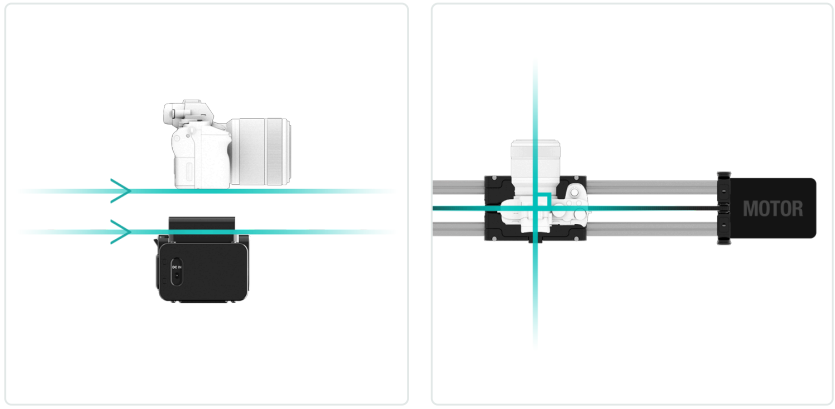
On a slider the camera stays level with the ground whatever the incline of the rail. On a SliderPLUS (and a SliderONE) it points along the rail, away from the motor end, or away from the rail when the rail is inclined or upright; on a rail with a separate motor module it looks across the rail. The SDK reads which slider the layout carries and describes the right one in the step's `instruction`.



SliderPLUS, rail level (up to 15°)

Rail inclined (15° to 75°)

Rail upright (75° and over)



A rail with a separate motor module, from the side

The same, from above: across the rail

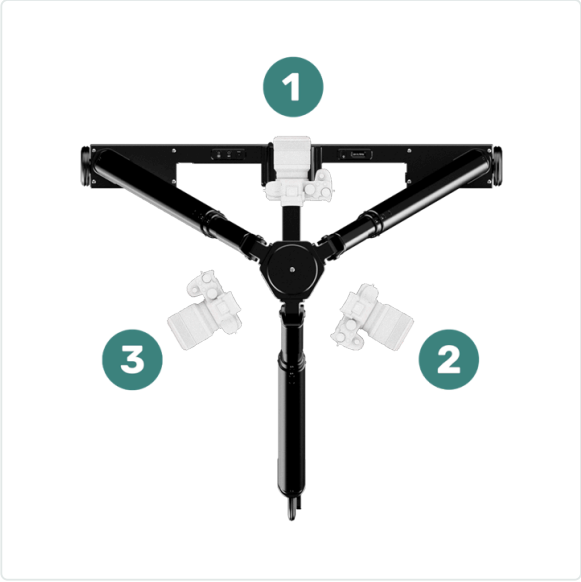
- Figure 18: Head on a slider -

Layout	From above	From the side
Krone X	The camera, the jib arm and the dolly's forward direction point the same way, in parallel.	The boom horizontal, the camera level with the horizon, the dolly's base flat.
Krone S	The camera's optical axis on the line of the dolly's single leg, the rail at a right angle to it.	The rail upright, the camera level with the horizon, the dolly's base flat.
Head on a JibONE or JibPLUS	—	The arm horizontal; the camera level, pointing straight out along the arm.
Head on a dolly	The camera looking the way the dolly faces, on the line of its single leg.	The camera level with the ground.
Head on a SliderPLUS or SliderONE	—	The camera level, pointing along the rail away from the motor end; on an inclined or upright rail, level and pointing away from the rail.
Head on a rail with a separate motor module	The camera looking across the rail, at a right angle to it.	The camera level.

- Figure 19: The head reference of each layout -

7.4 Krone S Camera Position

A Krone S is told where the camera sits on it, as numbered on the figure. The device cannot measure it, and a wrong choice turns the head the wrong way as the dolly travels.



- Figure 20: Krone S camera positions: 1 at the rail, 2 on the right-hand leg, 3 on the left-hand leg -

Position on the figure	Call
1 — at the rail, behind the dolly's leg	`set_camera_position(CameraPosition.BACK)`
2 — on the right-hand leg	`set_camera_position(CameraPosition.RIGHT)`
3 — on the left-hand leg	`set_camera_position(CameraPosition.LEFT)`

- Figure 21: CameraPosition -

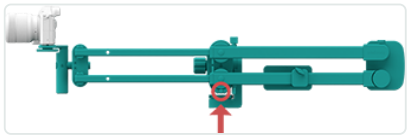
7.5 Jib Arm

The head computes where the camera is from the jib arm's length and the position of its counterweight, read off the arm. Set the jib parallel to the ground and lock its centre latch, then read: `length` is the reading on the arm's ruler, from 1 to 66, and `rod_lines` the number of lines visible on the counterweight rod.

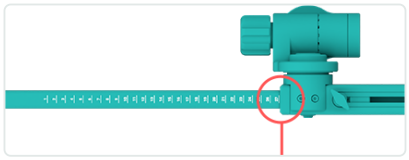
- **The newer jibs' counterweight rod carries no lines.** For them `rod_lines` is `"-"`, which is the default, so it can be left out: the edelkrone application shows the same `-` and has it as its default. The device receives 0.
- On an earlier jib whose rod carries lines, `rod_lines` is the number of lines visible, from 1 to 5, as in the figure below.

The device forgets the measurement when it is switched off: give it again after every power-up.

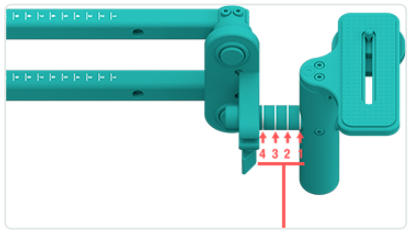
```
device.point.set_jib_arm(length=24)                # a rod without lines: "-"
device.point.set_jib_arm(length=24, rod_lines=3)   # a rod with three lines showing
```



The arm parallel to the ground, the centre latch locked



`length`: the reading on the ruler

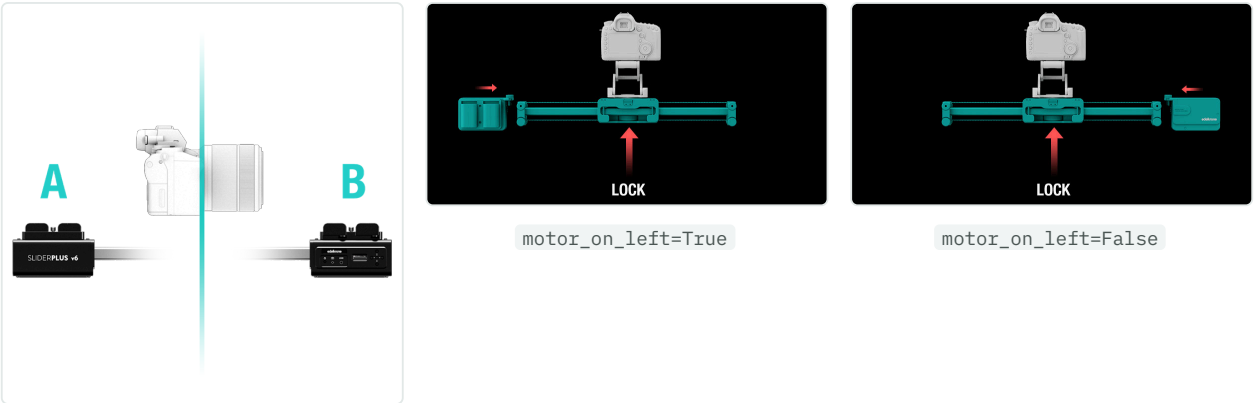


`rod_lines` on a rod with lines; `"-"` when it has none

- Figure 22: Jib arm -

7.6 Slider Placement

The head is told where the slider stands: `on_ground=True` for a slider on the ground with its leg extensions attached, `False` for one on tripods with the legs extended as recommended. On a rail with a separate motor module, `motor_on_left` says which end the motor is at, seen with the camera looking forward as drawn. A SliderPLUS carries its motor in the rail: call without `motor_on_left`. When the SDK cannot tell which slider the layout carries, it asks for `motor_on_left` rather than guess.



The motor on the left (A) or on the right (B), seen with the camera looking forward

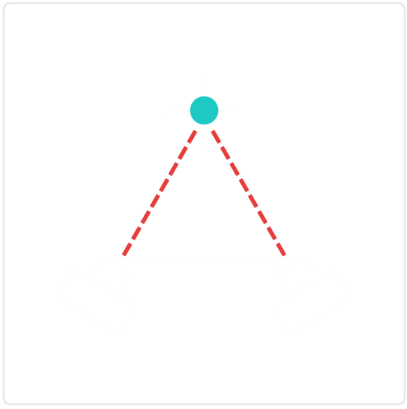
`motor_on_left=True` `motor_on_left=False`

- Figure 23: Slider placement -

7.7 Teaching the Point

The head works out where the subject is from two aims at it, taken from two positions: the two lines of sight meet at the subject. After `calibrated()`:

1. Aim the camera at the subject with `move.joystick(pan=…, tilt=…)` and call `mark_first_aim()`.
2. Move the equipment sideways with the slider, the dolly or the jib. Turning the head does not count.
3. Aim at the same subject again and call `mark_second_aim()`. The head works out the point and turns to it, which takes about ten seconds; the call returns `True` once the head holds the subject, and from then on pan and tilt follow it while the equipment moves. If the equipment has not moved far enough, it raises `baseline_too_short`: move further, aim again and call it again.



- Figure 24: A point from two aims -

```
device.point.mark_first_aim()           # aimed at the subject
end = time.monotonic() + 3.0            # move the equipment sideways
while time.monotonic() < end:
    device.move.joystick(slide=0.3)
    time.sleep(0.1)
device.move.stop()
...                                     # aim at the subject again
assert device.point.mark_second_aim()   # the head holds the subject

end = time.monotonic() + 4.0            # the head follows it
```

```
while time.monotonic() < end:
    device.move.joystick(slide=-0.4)
    time.sleep(0.1)
device.move.stop()
device.point.release()
```

One point at a time, kept nowhere. `release()` lets go of the subject, and the next point is taught again from two aims. So is a point that was lost — the head taken out of point mode, the layout dissolved, the unit switched off. `mark_first_aim()` always starts over: a point being taught is abandoned and a point held is let go. The SDK teaches into the head's first point slot (P1 in the edelkrone application) and replaces what is there.

7.8 While Point Mode Is Open

- **The joystick.** The module's axes — `slide`, `jib_pan`, `jib_tilt`, `linear`, `angular` — drive as always, and a held point is followed. While a point is held, `pan` and `tilt` move the framing around it instead of turning the head away from it, as the pan/tilt stick of the edelkrone application does.
- **`move.stop()`** stops the motors. A point that is held stays held: the head only turns to follow it while the equipment moves. `release()` lets go. A stop made from another thread while `mark_second_aim()` is still waiting for the head's turn cuts the turn short, and the point is lost.
- **Poses step aside.** In point mode the head keeps its points where it otherwise keeps its poses: `poses.stored` and `status.stored_pose_slots` read empty, and every other call of `device.poses` raises `point_mode_open`. `close()` returns the head to its poses.
- **The activity.** `status.activity` reads `transitioning` while the head turns to the point, `tracking_target` while it holds it, and `moving` while it is being taught. A held point does not end: `move.wait_until_idle()` waits for it until its timeout.
- **Another program**, such as the edelkrone application, can take the head out of point mode. The next call of `device.point` that needs point mode then raises `point_not_open`; `open()` and `calibrated()` start it again.

7.9 `device.point`

Member	Description
`open()`	Puts the head into point mode and returns the `PointCalibration` of the layout; called again, it reads the list without sending anything. Refused with `NotSupported` on a layout without point mode.
`calibrated()`	The operator's confirmation that the calibration is done. Raises `not_calibrated`, naming the missing steps, or `point_not_open` before `open()`.
`is_open`	Whether point mode is ready: `calibrated()` was given and the head is still in point mode.
`close()`	Takes the head out of point mode, back to poses. A point being taught is abandoned. Available at any time.
`set_head_reference()`	Records the pose the camera is in as the head's reference (7.3).

Member	Description
`set_camera_position(position)`	A Krone S's camera position, a `CameraPosition` (7.4).
`set_jib_arm(length=…, rod_lines="-")`	The jib arm's measurement: the ruler, 1 to 66; the rod's lines, 1 to 5, or `"-"` (the default) for a rod without lines (7.5).
`set_slider_placement(on_ground=…, motor_on_left=None)`	Where the slider stands, and where a separate motor module is (7.6).
`mark_first_aim()`	The first aim at the subject: a new point begins. Always starts over — a point being taught is abandoned, a point held is let go.
`mark_second_aim()`	The second aim, from another position: the head works out the point, turns to it and holds it; it returns once the head reports the hold, about ten seconds. Returns whether it holds it; raises `baseline_too_short` while the equipment is too close to where the first aim was made, and `out_of_sequence` without a first aim.
`release()`	Lets go of the subject, or abandons a point being taught. The next point starts from `mark_first_aim()`. Available at any time.

- Figure 25: device.point -

A `PointCalibration` has `layout`, `steps`, `missing` (the steps not done, in order) and `is_complete`; printed, it is the list with each step marked done or missing. Each `PointStep` has `name`, `title`, `done`, `action` (the call that performs it) and `instruction` (what to do physically). Both export as JSON with `to_dict()` and `to_json()`.

> **Not in this release.** Point mode on a HeadONE; a turntable connected to the head and the turntable centre's calibration; several points kept in the head and a turn back to one; learning a point with the range laser or at a typed distance. The edelkrone application provides them. A turn to a stored position of the module while the head holds a point is not in this release either: move the module with the joystick.

8 Gimbal Module

The Gimbal Module drives a DJI RS gimbal, and the SDK uses it as a head. `discover()` lists it as `Gimbal Module V1`, and `connect()` connects to it as to a HeadPLUS V3. The joystick's `pan` and `tilt`, the stored poses, point mode and focus all work on it as on a HeadPLUS V3, and it forms the same layouts. With a JibPLUS it forms `pan_tilt_and_jib_plus`: connect to the Gimbal Module and name the JibPLUS in `with_modules`, and point mode then runs on that layout (7). Its head reference sets the pan only; the tilt is measured from the gimbal's own horizon. A DJI focus motor on the gimbal is the head's focus motor (6). What the Gimbal Module adds is in `device.gimbal`.

```
device = edelkrone.connect(mac="D4:E8:5A:10:22:33", with_modules=["D2:AC:41:7B:90:12"])
if device.gimbal.is_locked:
    device.gimbal.unlock()
device.move.joystick(pan=0.3)           # repeat while the stick is held, then move.stop()
device.gimbal.recenter()
```

Member	Description
`recenter()`	Turns the gimbal back to its centre. Mounted underslung, the gimbal's own selfie recenter is used. Returns once the module has answered; the gimbal then turns on its own.
`lock()`	Locks the gimbal. The module also locks it by itself after 180 seconds at rest.
`unlock()`	Unlocks the gimbal so that it can be moved again.
`is_connected`	Whether the DJI gimbal answers the module, from the latest reading; `null` before it.
`is_locked`	Whether the gimbal is locked, from the latest reading; `null` before it.

- Figure 26: device.gimbal -

While the gimbal does not answer, the module ignores every pan and tilt move. While it is locked, the edelkrone app holds its pan/tilt stick. So the SDK refuses a joystick with `pan` or `tilt`, and a move to a stored pose, with `CommandError` · `gimbal_not_connected` or `gimbal_locked` when the latest reading says so. The JibPLUS's own axes, `jib_pan` and `jib_tilt`, are not refused, and `move.stop()` never is. On any other unit the members of `device.gimbal` raise `NotSupported`.

> **Not in this release.** The gimbal's roll axis, a zoom motor on the gimbal and the auto-lock time setting.

9 Vision Module

The Vision Module sends the picture of the camera it is plugged into to this computer, over Wi-Fi. It takes the camera's HDMI output, at 720p or 1080p. In this release the SDK sets the module's Wi-Fi up and receives its picture, both from the computer: the edelkrone application and a phone are not needed. The module's subject tracking is not in this release.

With other units. The module is not one of the units of a layout: it joins its head by the link cable, so it is not named in `with_modules`, and `connect()` does not connect to it. `connect()` says to use `connect_vision_module()` instead, and a `connect()` without a name or address does not count a Vision Module among the units in range.

9.1 Wi-Fi and the Picture

The module joins Wi-Fi through its own Bluetooth, which it advertises as `VisMod`; `edelkrone.connect_vision_module()` connects to it. Once the module is on the network, it announces itself there, and `VisionStream` receives its picture as H.264. The computer must be on the same network. The picture needs two extra packages: `pip install "edelkrone[vision]"` installs `pyzmq` for the stream and `av` and `numpy` for decoding it. Without them, `VisionStream` raises `NotSupported` saying so.

```
with edelkrone.connect_vision_module() as module:   # the module's own Bluetooth
    status = module.join_wifi("Studio", "password")  # returns once it is on the network
print(status.ip, status.hdmi)
with edelkrone.VisionStream(status.ip) as stream:
    for image in stream.images():                    # numpy arrays, height x width x 3 (BGR)
        ...
# later, or on another computer on the network:
ip = edelkrone.find_vision_modules()[0]
```

Network names and passwords. For now the module joins only a network whose name and password are made of the letters A–Z and a–z, the digits 0–9 and `-` `_` `.`: a name of 1 to 32 characters and a password of 8 to 63. A space, a letter outside A–Z such as `ü`, or a character such as `&`, `'` or `!` stops it from joining. We are sorry for the trouble; a firmware update will fix this as soon as possible. Until then, open a hotspot with a simple name and password, for example on a phone or a router, and join that; or let the module open its own network with `create_access_point()` and join that network from the computer. `join_wifi()` and `create_access_point()` refuse another name or password with `CommandError` · `bad_argument` before anything is sent, and the message gives the same advice.

A phone's hotspot. If the module does not find a phone's hotspot (`wifi_ssid_not_found`), turn on *Maximize Compatibility* in the iPhone's Personal Hotspot settings, which puts the hotspot on 2.4 GHz, and keep that screen open while the module joins; on an Android phone choose the 2.4 GHz band. The computer joins the same hotspot.

Member	Description
`edelkrone.connect_vision_module(name="VisMod", address=None, timeout=20.0)`	Connects to the module's own Bluetooth and returns a

Member	Description
	`VisionModuleLink.address` picks one module when several are in range. The module must not be connected to the edelkrone app at the same time.
`VisionModuleLink.status(timeout=3.0)`	A `VisionNetworkStatus`: `network` (`disconnected`, `connecting`, `ssid_not_found`, `wrong_password`, `wifi` or `access_point`), `ip`, `ssid`, `mac`, `hdmi`, `battery_percent`, `uptime_s` and `firmware`.
`VisionModuleLink.join_wifi(ssid, password, timeout=30.0)`	Joins the Wi-Fi network `ssid` and returns the status once the module is on it with an address. Raises `wifi_ssid_not_found`, `wifi_wrong_password` or `wifi_timed_out`.
`VisionModuleLink.create_access_point(ssid, password, timeout=30.0)`	Opens a Wi-Fi network of the module's own, for the computer to join.
`VisionModuleLink.leave_wifi()`	Takes the module off its network.
`VisionModuleLink.disconnect()`	Closes the Bluetooth connection; the module stays on its network. A `with` block calls it.
`edelkrone.find_vision_modules(seconds=3.0)`	The addresses of the modules announcing themselves on this network. When it finds none, `VisionModuleLink.status().ip` gives the address over Bluetooth.
`VisionStream(ip, quality=None)`	Receives the picture of the module at `ip`. Iterate it for `VisionFrame`s, or use `images()`. `quality`, `"low"`, `"medium"` or `"high"`, sets the picture's quality first. A context manager; `close()` ends it.
`VisionStream.set_quality(quality, timeout=3.0)`	Sets the picture's quality, `"low"`, `"medium"` or `"high"`, and returns once the module reports it. Raises `no_reply` when the module does not within `timeout` seconds.
`VisionStream.quality(timeout=2.0)`	The quality the module runs at, or `null` when its bit rate is none of the three.
`VisionStream.read(timeout=2.0)`	The next `VisionFrame`, or `null` when none arrives in time.

Member	Description
`VisionStream.images(timeout=2.0)`	`bad_frames` counts the frames dropped because they arrived damaged. Decoded pictures, as `numpy` arrays of height × width × 3 in BGR order, until no frame arrives within `timeout`.

- Figure 27: The Vision Module's network and picture -

Picture quality. The module sends its picture in one of three qualities, as in the edelkrone app: `"low"`, `"medium"` and `"high"`, bit rates of its encoder of 1, 4 and 10 Mbit/s. A lower quality sends fewer bytes for the same pictures, so on a slow or crowded network the picture keeps flowing at its full rate where a higher one would stall or drop pictures; a higher quality shows fine detail and movement with fewer blocks. The quality is the module's own setting: it applies to every program and phone receiving its picture, and stays until it is changed or the module restarts. The quality does not change how many pictures the module sends: 25 a second from a camera set to a 50 Hz output (PAL: 1080p 50 or 25) and 30 from a 60 Hz one (NTSC: 1080p 60 or 30), at every quality. For 30 pictures a second, set the camera to NTSC.

`"low"`	1 Mbit/s. Measured with a 1080p camera on a phone's hotspot: 25 pictures a second, about 1.3 KiB each.
`"medium"`	4 Mbit/s, the module's own default: 25 pictures a second, about 4.8 KiB each.
`"high"`	10 Mbit/s: 25 pictures a second, about 10.5 KiB each. The size of a picture depends on what the camera sees; a dark, still scene needs far less than the bit rate allows.

- Figure 28: The Vision Module's picture qualities -

The picture in a window. `python -m edelkrone.examples.vision_view` finds the module on the network and shows its picture live in a window; `--join NAME PASSWORD` first puts the module on that network through its Bluetooth, `--ip` gives its address, and `--quality low`, `medium` or `high` sets the picture's quality. Press S to save the picture shown as a PNG file, and Esc to stop. The window uses Tk, which is part of most Python installations (on macOS with Homebrew: `brew install python-tk`). The window shows about `--width` pixels wide, 640 by default; drawing limits it, and measured on a Mac it showed all 25 pictures a second at 640 pixels and about 22 at 960.

A `VisionFrame` has `index`, `width`, `height`, `fps`, `sps_pps` (the stream's parameter sets) and `data`, one H.264-encoded frame. A decoder of your own takes `sps_pps + data` for the first frame and `data` after it.

> **Not in this release.** The module's subject tracking — following a face or an object, which goes through the head — its easy-tracking modes and a USB camera's recording. The picture's bit rate is the module's own.

10 Settings

Member	Description
`buzzer(muted=…)`	Mutes or unmutes the buzzer, and returns once the device has taken the setting: within 0.1 s, or about 0.7 s when settings are written one after another. `muted` must be `True` or `False`.

- Figure 29: device.settings -

The device does not acknowledge settings. The call returns once the command has been sent.

11 Tripod

A motorized tripod levels itself and raises or lowers its column. The two operations are reached through `device.settings` and are refused on units other than tripods.

```
device.settings.level_tripod()          # starts the automatic levelling

end = time.monotonic() + 3.0            # raises the column for 3 seconds
while time.monotonic() < end:
    device.settings.drive_tripod(40)
    time.sleep(0.2)
device.settings.drive_tripod(0)         # stops the column
```

Member	Description
`level_tripod()`	Starts the automatic levelling of the tripod.
`drive_tripod(relative_speed)`	Moves the column at a speed from −100 to 100: positive values extend it, negative values retract it, and 0 stops it. Values outside the range are limited to −100 and 100, a fraction is rounded to the nearest whole number, and a value that is not a finite number is refused with `bad_argument`. **It must be repeated** every 100 to 200 ms for as long as the column is to move, in the same way as the joystick; a single call moves the column for about 1.5 s, after which it stops by itself. `move.stop()` stops the column as well.

- Figure 30: Tripod operations of device.settings -

The tripod does not acknowledge these commands. The calls return once the command has been sent. `status.activity` reads `moving` while the column moves and while the tripod levels, as the device reports it. After `drive_tripod(0)` it may go on reading `moving` for 1 to 1.5 s; `move.stop()` ends it at once. A tripod that is already level finishes levelling within a second. The tripod reports no inclination, so `tilt_deg` and `is_level` are `null`.

12 Errors

The SDK distinguishes two kinds of failure. A **refusal**, such as an empty slot or a move that did not start, is returned as a result with `is_success` and a `reason`. A **fault** raises one of three exceptions, all subclasses of `edelkrone.Error`. Each carries a `reason` code, a `message` and a `cause` with technical details for support requests.

```
try:
    device = edelkrone.connect(name="HeadPLUS V3")
except edelkrone.ConnectionError as e:
    print(f"{e.reason}: {e.message}")       # not_found: HeadPLUS V3 was not seen in 20.0 s. Seen: …
```

Exception and reason	Cause	Action
`ConnectionError` · `not_found`	The unit was not found within the timeout: it is switched off, out of range, or connected to another program. Or the address given is that of a module in a layout, which is reached through its head; the message names the head.	Check the power and close other connections to the unit. `discover()` lists the units in range.
`ConnectionError` · `ambiguous`	No name or address was given, and more than one unit is in range.	Specify the unit. The message lists the units found and their addresses.
`ConnectionError` · `no_radio`	The computer has no Bluetooth adapter, or the Bluetooth package is not installed.	Check the adapter. Reinstall with `pip install edelkrone`.
`ConnectionError` · `no_permission`	The operating system has not granted Bluetooth access to the program.	macOS: System Settings → Privacy & Security → Bluetooth.
`ConnectionError` · `timed_out`	The unit was found but did not accept the connection (three attempts of up to 20 seconds, three seconds apart), or did not respond to the first identity request. A unit held by the edelkrone application on a phone does this.	Close any other connection to the unit; if none, restart the unit and reduce the distance.
`ConnectionError` · `dropped`	The connection was lost while `connect()` was reading the identity or joining the modules. A connection lost later is reported by `connection_changes`, and commands then raise `not_connected`.	Reconnect.
`ConnectionError` · `already_connected`	The unit is already connected in this program.	Use the `Device` that `connect()` returned, or `disconnect()` it first.
`ConnectionError` · `wrong_device`	The unit connected but does not provide the services expected for its model, or answers with a frame this release cannot read, usually because of outdated firmware; or the unit reports a firmware error (its main firmware is not running, for example after an interrupted update), in which case `discover()` lists it with `recognised` set to `false` and no connection is attempted.	Check `info.firmware_ok` and update the firmware with the edelkrone application; after a firmware error, update it and switch it off and on.
`ConnectionError` · `layout_refused`	The unit did not take the modules named in `with_modules` into its layout — a FocusPLUS PRO counts as taken when the head reports it — or took them in but they did not answer, for example because it already forms a layout with other modules, or a module belongs to another layout.	Check that the modules are switched on. Switch all units off and on so that each advertises on its own, then name all modules in one `connect()`. For a FocusPLUS PRO, check that it is switched on and not connected to another head or to the edelkrone application.

Exception and reason	Cause	Action
`CommandError` · `not_connected`	A command was sent after the connection was lost.	Reconnect.
`CommandError` · `closed`	A command was sent after `disconnect()`.	Connect again. A `Device` cannot be reused after it is closed.
`CommandError` · `no_reply`	The device did not respond to a command in time.	Retry once. If the error persists, restart the unit.
`CommandError` · `bad_reply`	The device sent an unexpected response.	Report the error with `info.firmware` and `cause`.
`CommandError` · `bad_argument`	An argument is invalid, for example not a number, a slot out of range, a speed of `nan` or above 1, a timeout that is not a positive number of seconds, a loop given to `move_to()`, or a move started to an empty slot.	Correct the argument.
`CommandError` · `from_callback`	A command was sent from inside a `tap()` callback, where its reply could never be delivered.	Hand the work to a thread of your own, or read `device.status`.
`CommandError` · `needs_focus_setup`	The unit reports a focus motor and the device has not learned the lens ends since it was switched on, or has dropped them. Only the stored-pose operations, the calls of `device.point` that change the equipment and the focus calls raise it.	Call `device.focus.setup()` (6).
`CommandError` · `point_not_open`	A call of `device.point` that needs point mode was made before `open()` and `calibrated()`, or after the head left point mode, for example because the edelkrone application switched it.	Call `device.point.open()`, perform the missing steps and call `calibrated()` (7).
`CommandError` · `not_calibrated`	`point.calibrated()` was called with calibration steps missing; the message names them.	Perform the steps in order (7.2).
`CommandError` · `out_of_sequence`	A point mode step was called out of its order: a calibration step before the steps above it, or a second aim with no first aim before it.	Call the step the message names first.
`CommandError` · `baseline_too_short`	The second aim at the subject was made before the equipment had moved far enough from where the first was made.	Move the module further sideways, aim again and call `mark_second_aim()` again.
`CommandError` · `point_mode_open`	A pose operation was called while the head is in point mode, where it keeps points instead of poses.	Call `device.point.close()` first.
`CommandError` · `focus_ends_not_found`	`focus.setup()` swept the lens for 45 seconds without finding its ends, and stopped.	Check that a lens is fitted and the focus gear engages it.
`CommandError` · `focus_motor_not_referenced`	`focus.setup()` waited `timeout` seconds for the focus motor to set its reference, and it did not. No command cancels the reference.	Switch the head off and on before trying again, and check the cable between the head and the focus motor.
`CommandError` · `gimbal_not_connected`	A pan or tilt move was sent to a Gimbal Module whose DJI gimbal does not answer; the module would ignore it.	Check that the gimbal is switched on and its cable is in (8).
`CommandError` · `gimbal_locked`	A pan or tilt move was sent to a Gimbal Module whose gimbal is locked.	Call `device.gimbal.unlock()`.

Exception and reason	Cause	Action
`CommandError` · `wifi_ssid_not_found`, `wifi_wrong_password`, `wifi_timed_out`	`join_wifi()` or `create_access_point()`: the module did not find the network, the network refused the password, or the module was not on it in time.	Check the name and the password, and that the network is in range of the module (9.1). A name or a password with a space or a character other than A–Z, a–z, 0–9 and `-` `_` `.` is refused before it is sent, with `bad_argument`: use a hotspot with a simple name and password.
`NotSupported` · `unsupported`	The model or layout does not provide the function, for example a pose on a tripod or a dolly, focus on a slider or on a head that reports no focus motor, a focus module connected to on its own, point mode on a head without a module or with a turntable connected, or a calibration step the layout does not have; or the units named in `with_modules` do not form a supported layout.	Check `device.info.capabilities`.

- Figure 31: Errors raised by the SDK -

`edelkrone.ConnectionError` is not Python's built-in `ConnectionError`. Refer to it as `edelkrone.ConnectionError`; `from edelkrone import *` replaces the built-in name in the importing module.

Signal strength. A unit at the edge of the radio range can be found by `discover()` but may fail after connecting, with `dropped`, `not_connected` or `no_reply`. Layouts with modules are affected first. If connections are unreliable, reduce the distance to the unit.

13 Diagnostic Log

The SDK writes diagnostic messages to `EdelkroneLog`. The log is disabled by default. To enable it, assign a function to `EdelkroneLog.sink` and, optionally, set `EdelkroneLog.minimum_level` to an `EdelkroneLogLevel`: `TRACE`, `DEBUG`, `INFO` (the default), `WARNING` or `ERROR`.

```
from edelkrone import EdelkroneLog, EdelkroneLogLevel

EdelkroneLog.sink = lambda level, message: print(f"[{level.name}] {message}")
EdelkroneLog.minimum_level = EdelkroneLogLevel.DEBUG
```

A sink is an `EdelkroneLogSink`: a function that takes the level and the message. It may be called from several threads, must not raise exceptions and must return quickly. To forward messages to Python's `logging` module, call a `logging` logger from the sink.

14 API Summary

The following table lists every name exported by `import edelkrone` and the section that describes it.

Name	Description	Section
`connect()`, `discover()`, `open_radio()`	Connect to a unit; list the units in range; open a radio to share between calls.	2
`Device`	A connected unit, with its identity, state and five operation groups.	3
`Info`, `DeviceStatus`, `Seen`	The identity, a reading, and a unit found in a scan. Each can be exported as JSON.	3.1, 3.2, 2.2
`MoveControl`, `PoseController`, `FocusControl`, `PointControl`, `SettingsControl`	The types of `device.move`, `device.poses`, `device.focus`, `device.point` and `device.settings`, for use in type hints. They are not constructed directly.	4, 5, 6, 7, 10, 11
`PointCalibration`, `PointStep`	Point mode's calibration for a layout, and one of its steps.	7.1, 7.9
`CameraPosition`	A Krone S's camera position.	7.4
`GimbalControl`	The type of `device.gimbal`: the Gimbal Module's recenter and lock.	8
`connect_vision_module()`, `find_vision_modules()`, `VisionModuleLink`, `VisionNetworkStatus`, `VisionStream`, `VisionFrame`	The Vision Module's Wi-Fi, found on the network, and its picture.	9.1
`MoveOutcome`, `MoveStyle`	The result of a pose move; the speed and acceleration of a move.	5
`ActivityState`, `Capability`	The activity of a device; a function a model supports.	3.2, 3.1
`RigPose`, `RigAxis`, `EdlSetup`, `DeviceFamily`	A position, an axis, a layout, and a device family (head, slider, jib, dolly, focus module or tripod; a focus module is reached through its head).	3.2, 3.1, 2.3
`SessionConfig`, `Transport`	The types accepted by `connect(config=…)` and `connect(transport=…)`; `open_radio()` returns a `Transport`.	2.1
`Error`, `ConnectionError`, `CommandError`, `NotSupported`	The base exception and the three fault types.	12
`EdelkroneLog`, `EdelkroneLogLevel`, `EdelkroneLogSink`	The diagnostic log, its levels and the sink type.	13
`__version__`	The installed version.	1

- Figure 32: Exported names -

This content is subject to change.

If you have any questions about this document, please contact edelkrone.

Copyright © 2026 edelkrone.