SDK Reference

Read with AI

> The Python semantic API for Wuji Hand 2. Unified pattern: `hand.{resource}().{action}()`. All operations target the whole hand (20 joints).

This page focuses on the Wuji Hand 2–specific semantic API. The SDK builds on the shared **Wuji SDK** (`wuji_sdk`, `pip install wuji-sdk`)—for general installation, device connection, and data subscription, see the Wuji SDK docs; for source, see the wuji-sdk repository.

1. Connection

> **Network prerequisite**: Wuji Hand 2 uses a static IP, not DHCP. Each device ships with a fixed address by handedness — left hand `192.168.1.110`, right hand `192.168.1.111`, gateway `192.168.1.1`, subnet mask `255.255.255.0`. Before the first connection, set your host NIC to the same subnet (for example `192.168.1.100`), then connect by address. The factory IP is changeable later — see ip (SET/GET).

1.1 Connect via SdkManager (Recommended)

```python
from wuji_sdk import SdkManager, Handedness

manager = SdkManager.instance()

# Auto-discover and connect to the first Wuji Hand 2 on the LAN
hand = manager.auto_connect(device_name="wuji_hand_2")

# Connect explicitly by serial number or address
hand = manager.connect(sn=<device_sn>, device_name="wuji_hand_2")
hand = manager.connect(address="192.168.3.110:50001", device_name="wuji_hand_2")

# Or connect by handedness directly, without a serial number
hand = manager.connect(handedness=Handedness.Right, device_name="wuji_hand_2")
```

The `device_name="wuji_hand_2"` overloads of `auto_connect` and `connect` return a `WujiHand2` instance directly (with type hints). `handedness` is mutually exclusive with `sn` and `address`. If multiple Hand 2 units of the same handedness are on the network, `AmbiguousHandedness` is raised—specify `sn` instead.

1.2 ConnectOptions

```
from wuji_sdk import SdkManager, ConnectOptions

opts = ConnectOptions(
    timeout_ms=1000,
    retry_count=3,
    enable_bridge=True,    # Default True: lets multiple clients (Wuji Studio + recording scripts + your app)
)
hand = manager.connect(sn=<device_sn>, device_name="wuji_hand_2", options=opts)
```

Set `enable_bridge=False` for exclusive single-client mode.

1.3 Instance Attributes

Attribute	Type	Description
`serial_number`	`str`	Device serial number
`device_name`	`str`	Name given at connect (default `"wuji_hand_2"`)
`info`	`Optional[DeviceInfo]`	Device info: `serial_number`, `firmware_version`
`is_connected`	`bool`	Connection state

`hand.hw_version().get()` returns the factory hardware version `HwVersion(major, minor, patch)`. `0.0.0` means it wasn't written at the factory.

2. Hand-Level Resources

This section covers the whole-hand API (20 joints). Single-joint reads and actions go through the `JointHandle` returned by `hand.joint(k)` / `hand.joints()` (see Section 3, Joint Traversal). The device runs in MIT control mode by default — the control mode isn't set from the Python API.

2.1 handedness (GET)

Get the **handedness** result (left / right).

```
side = hand.handedness().get()   # → "left" or "right"
```

2.2 online_joints_count (GET)

Get the **online joint count** (0–20).

```
n = hand.online_joints_count().get()   # → int, 0-20
```

2.3 joint_diagnostics (SUB)

Joint diagnostics stream: subscribe to per-joint status word, current, bus voltage, temperature, and error code. Frames are variable-length and contain only online joints — identify each entry by its `nid`.

```
sub = hand.joint_diagnostics().subscribe()   # → Subscription[JointDiagnosticsFrame]
frame = await sub.recv_async()
for j in frame.joints:
    print(j.nid, j.vbus_v_fb, j.mcu_temp_c_fb, j.error_code_current)
sub.close()
```

Decode `error_code_current` into an error name and description with the static method `WujiHand2.describe_error(code)`.

> For joint angles, use `position` from the `joint_states` stream — the diagnostics stream doesn't carry joint angles.

2.4 comm_diag (GET, 1 Hz)

Communication diagnostics: returns the throughput / error rate of the 5 fingers, once per second.

```
diag = hand.comm_diag().get()           # → HandCommunicationDiagnostics
for finger in diag.fingers:             # throughput / error rate for each of the 5 fingers
    print(finger.tx_kbps, finger.rx_kbps, finger.error_per_sec)
```

Use it to troubleshoot communication between the joints and the control board.

2.5 effort_limit (SET/GET)

Effort limit: read / write the per-joint torque cap (A).

```
hand.effort_limit().set(1.5)            # set all joints to 1.5 A
limits = hand.effort_limit().get()      # → list[Optional[float]]
```

2.6 mit_params (SET/GET)

MIT impedance parameters: read / write per-joint kp / kd.

```
hand.mit_params().set((1.0, 0.05))          # same kp / kd for all joints
hand.mit_params().set([(1.0, 0.05)] * 20)   # list of 20, per joint
mp = hand.mit_params().get()                # → list[Optional[MitParam]]; offline joints are None
```

Writes reject NaN, Inf, and negative values.

2.7 clear_fault (EXEC)

Clear faults: a direct action. Without arguments it acts on the whole hand.

```
hand.clear_fault()                  # clear faults on all joints
hand.clear_fault(joints=mask)       # optional: 0/1 mask of length 20 — only joints with a 1 are cleared
```

2.8 User Origin

Calibrate the current physical position as the joint-side zero. If a joint is moving, it takes effect at the next IDLE.

```
hand.set_origin()                   # whole hand: refresh the user origin of all joints
hand.clear_origin()                 # whole hand: clear the user origin of all joints
hand.set_origin(joints=mask)        # optional: 0/1 mask of length 20 — only joints with a 1 are affected
```

2.9 Enable / Disable / Emergency Stop

```
hand.enable()               # enable all joints
hand.disable()              # disable all joints
hand.enable(joints=mask)    # optional: 0/1 mask of length 20 — acts on joints with a 1
hand.emergency_stop()       # emergency stop: whole-hand action, no mask
```

2.10 joint_states (SUB)

Joint state stream: subscribe to the real-time state frames of the 20 joints.

```
sub = hand.joint_states().subscribe()   # → Subscription[JointStateFrame]
frame = sub.recv()                      # synchronous, non-blocking; None = no data yet
frame = await sub.recv_async()          # asynchronous wait
print(frame.header.seq, frame.header.timestamp_us)
for j in frame.joints:
    print(j.nid, j.position, j.velocity, j.effort)
sub.close()
```

Callback mode:

```
def on_state(frame):
    print(frame.header.seq, [j.position for j in frame.joints[:4]])

cb = hand.joint_states().subscribe_with_callback(on_state)
# ...
cb.close()
```

> `position` is the joint-side angle (rad) and `velocity` the joint-side angular velocity (rad/s)—this is the only recommended path for joint angle and speed. Frames are variable-length and contain only online joints — identify each entry by its `nid`. The `joint_diagnostics` stream doesn't carry joint angles.

2.11 joint_command (PUB)

Joint command: publish positions / velocities / torque feedforward for the 20 joints. Each send takes exactly 20 `JointCommand` entries, one `position` / `velocity` / `effort` per joint.

```
from wuji_sdk import JointCommand

pub = hand.joint_command().publish()    # → JointCommandPublisher
pub.send([JointCommand(position=p, velocity=0.0, effort=0.0) for p in positions])
pub.close()
```

To turn human hand keypoints into these 20-joint commands, use Wuji SDK Hand Retargeting.

2.12 Fingertip Tactile Streams (SUB)

> Fingertip tactile requires Beta 2 hardware equipped with the fingertip tactile sensors, plus firmware v2.1.0 and SDK v2026.7.21 or later, upgraded together.

Self-describing per-finger sensor streams: first fetch the finger's `FingertipSensorInfo` metadata (its `format` JSON describes the data-frame layout), then subscribe to that finger's `FingertipSensorData` stream and decode per the format. The thumb has 40 sensing points, the other fingers 34, streamed at 100 Hz.

```
import json

info = hand.get_fingertip_info(0)               # 0=thumb … 4=pinky → FingertipSensorInfo
fmt = json.loads(info.format)                   # the format fully describes the frame layout — never hardcod

sub = hand.fingertip_thumb_data().subscribe()   # one resource per finger: fingertip_{thumb,index,middle,ring
frame = await sub.recv_async()                  # → FingertipSensorData
if frame.info_digest != info.digest:            # a digest mismatch means the info changed
    info = hand.get_fingertip_info(0)           # re-fetch and rebuild the decoder
# decode frame.data per fmt's point_fields / aggregate_fields
sub.close()
```

For a complete consumer-side reference (including decoder construction), see the `3.fingertip_typed.py` example under `examples/python/wuji_hand_2/`.

2.13 Fingertip Tactile Calibration (EXEC) and Status (GET)

Zero-baseline recalibration: one call recalibrates all 5 fingertips to a fresh zero baseline.

```
hand.tactile_calibrate()      # whole-hand action; fails fast on the first offline finger
```

> Keep all sensor surfaces unloaded (no contact force) during the call — otherwise the calibrated zero baseline is wrong.

A successful call means the calibration command reached every finger, not that calibration finished. Confirm with the status query:

```
from wuji_sdk import TactileState

status = hand.tactile_status("thumb")   # "thumb" / "index" / "middle" / "ring" / "pinky"
print(status.model)                     # TactileType.Thumb (40 points) or TactileType.Standard (34 points)
print(status.state)                     # TactileState.Ready / TactileState.Calibrating
```

`tactile_status` is a blocking query (up to about 1 s). Calibration is complete once polling reads `state == TactileState.Ready`.

2.14 Flash Log Export (Diagnostics)

```
# Export the running logs of the current firmware (use this for most troubleshooting)
result = await hand.dump_hand_logs(bank="current", out_dir="./logs")
```

`bank`	**Meaning**	**When to use**
`"current"`	Logs written by the firmware currently running	Default — troubleshoot live issues
`"other"`	Logs left by the firmware version before an upgrade or rollback	Only when you need to trace behavior before an upgrade / rollback

Each call creates a separate session directory `<sn>-<unix_ts>/` under `out_dir`, containing `joint{0..19}.log` for each joint plus one `sboard.log` (JSONL: one `{"timestamp_ms", "level", "target", "message"}` per line). `out_dir` is optional and defaults to `~/.wuji/hand_logs`. Each call returns a dict:

```
result = await hand.dump_hand_logs(bank="current")
print(result["session_dir"])    # str: absolute path of this session directory
print(result["files"])          # list[str]: paths of the written .log files
```

2.15 ip (SET/GET)

Read and write the device's static IP address. `set` writes the new IP to the device's flash. The firmware doesn't hot-swap the running Ethernet stack, so the new IP takes effect only after the next reboot. Between `set` and reboot, `get` still returns the current IP.

```
hand.ip().get()                  # read the current IP, e.g. "192.168.3.110"
hand.ip().set("192.168.2.111")   # write to flash, takes effect after reboot
```

Full round-trip to change the IP: `set` → `reboot` → reconnect at the new IP → `get` to confirm.

```
hand.ip().set("192.168.2.111")
hand.reboot()
manager.disconnect_all()
# wait for the device to reboot and Ethernet to come up, about 8 seconds
hand = manager.connect(address="192.168.2.111:50001", device_name="wuji_hand_2")
hand.ip().get()                  # → "192.168.2.111"
```

> ⚠ The new IP takes effect only after a reboot. `set` writes to flash without changing the live connection. Before the reboot, `get` still returns the current IP.

2.16 reboot (EXEC)

Reboot the device. The device disconnects and needs a fresh connection. Pair it with `ip().set()` to apply a new IP written to flash.

```
hand.reboot()
```

3. Joint Traversal

`JointHandle` provides `label` / `index` (for indexing into 20-element return arrays such as `effort_limit()` / `mit_params()`), plus single-joint resources and actions. `FingerHandle` traverses joints by finger.

Method	Return type	Description
`hand.joints()`	`list[JointHandle]`	All 20 joints
`hand.fingers()`	`list[FingerHandle]`	All 5 fingers

```
for joint in hand.joints():
    print(joint.label, joint.index)        # e.g. "thumb_S1" 0

for finger in hand.fingers():
    for joint in finger.joints():          # 4 joints per finger, in S1..S4 order
        print(joint.label)                 # "thumb_S1", "thumb_S2", ...
```

3.1 JointHandle

Joint handle: provides `label` / `index`, single-joint resources, and single-joint actions.

Member	Type	Description
`label`	`str`	Property. Joint label, format `{finger}_S{1..4}`, e.g. `"thumb_S1"`, `"pinky_S4"`
`index`	`int`	Property. Global index (0–19)
`effort_limit()`	Resource (SET/GET)	Single-joint torque cap (A)
`error_code()`	Resource (GET)	The joint's current error code — decode with `describe_error()`
`status_word()`	Resource (GET)	The joint's status word
`enable()` / `disable()`	Action	Enable / disable this joint
`clear_fault()`	Action	Clear this joint's faults
`set_origin()` / `clear_origin()`	Action	Set / clear this joint's user origin

3.2 FingerHandle

Finger handle: traverse the 4 joints of this finger.

Method	Return type	Description
`joints()`	`list[JointHandle]`	The 4 joints of this finger, in S1..S4 order

4. Data Types

4.1 JointDiagnosticsFrame / JointDiagnosticsEntry

Joint diagnostics frame: frame type of the `hand.joint_diagnostics().subscribe()` stream. Variable-length, online joints only.

```
class JointDiagnosticsFrame:
    header: FrameHeader                  # seq / timestamp_us / frame_id
    num_joints: int
    joints: list[JointDiagnosticsEntry]

class JointDiagnosticsEntry:
    nid: int                             # node ID — identifies the joint across frames
    status_word: StatusWord              # status word
    current: float                       # current (A)
    vbus_v_fb: float                     # bus voltage (V)
    mcu_temp_c_fb: float                 # MCU temperature (°C)
    error_code_current: int              # current error code — decode with describe_error()
```

4.2 MitParam

MIT parameters: element of `mit_params().get()`.

```
class MitParam:
    kp: float
    kd: float
```

`set` accepts a single `(kp, kd)` (applied to all joints) or a list of 20 (per joint). `get` returns 20 entries — offline joints are `None`.

4.3 JointStateFrame

Joint state frame: return value of `hand.joint_states().subscribe().recv()`. Variable-length, online joints only.

Attribute	Type	Description
`header`	`FrameHeader`	`seq` / `timestamp_us` / `frame_id`
`num_joints`	`int`	Joint count in this frame
`joints`	`list[JointStateEntry]`	Joint state array

4.4 JointStateEntry

Single-joint state.

Attribute	Type	Description
`nid`	`int`	Node ID — identifies the joint across frames
`position`	`float`	Position (rad, joint-side)
`velocity`	`float`	Velocity (rad/s)
`effort`	`float`	Torque (A)

4.5 JointCommand

Joint command: element of the `joint_command().publish().send()` argument.

```
class JointCommand:
    position: float    # position (rad)
    velocity: float    # velocity (rad/s)
    effort: float      # torque feedforward (A)
```

4.6 HandCommunicationDiagnostics

Communication diagnostics data.

```
class HandCommunicationDiagnostics:
    fingers: list[FingerCommunicationDiagnostics]   # length 5

class FingerCommunicationDiagnostics:
    tx_frame_total: int
    rx_frame_total: int
    tx_kbps: int
    rx_kbps: int
    error_per_sec: int
    crc_error_total: int
    frame_format_error_total: int
    uart_hw_error_total: int
    transfer_stats: list[TransferStats]
    nodes: list[NodeDiagnostics]    # per-node: online / ms_since_last_response / response_rate_pct
```

4.7 FingertipSensorInfo / FingertipSensorData

Fingertip sensor metadata and data frames: return value of `get_fingertip_info(finger)` and frame type of the `fingertip_{finger}_data` subscriptions.

```
class FingertipSensorInfo:
    header: FrameHeader    # seq / timestamp_us / frame_id
    digest: int            # CRC32 of the info content — binds data frames to this info
    model: str             # sensor model string (may be empty)
    device_type: int       # sensor type enum
    rate_hz: float         # data stream rate (Hz, currently 100)
    format: str            # JSON: data-frame layout (point_count / point_stride / point_fields / aggregate_f

class FingertipSensorData:
    header: FrameHeader
    info_digest: int       # digest of the info in effect — on mismatch, re-fetch the info
    data: list[int]        # pure value payload, interpreted per info.format
```

4.8 TactileStatus / TactileType / TactileState

Tactile status: return value of `hand.tactile_status(finger)`.

```
class TactileStatus:
    model: TactileType     # Standard (34-point fingertip) or Thumb (40-point thumb)
    state: TactileState    # Ready or Calibrating
```

Enum members compare equal to the firmware integer, for example `TactileState.Calibrating == 1`.

5. Joint Numbering

Finger	S1	S2	S3	S4
thumb	0	1	2	3
index	4	5	6	7
middle	8	9	10	11
ring	12	13	14	15
pinky	16	17	18	19

Label format: `{finger}_S{1..4}` (e.g. `thumb_S1`, `pinky_S4`).

6. Exception Handling

All SDK operation errors raise a unified `WujiException` whose message carries an error-type prefix (`Disconnected`, `Timeout`, `PathNotFound`, `SchemaMismatch`, `SerializeError`, ...). Use `try` / `except` as needed.

```
from wuji_sdk import WujiException

try:
    hand.joint_states().subscribe()
except WujiException as e:
    print(f"SDK error: {e}")
```