sharpa

SHARPA DOCUMENTATION

Wave

Product documentation for setup, operation, integration, development, troubleshooting, and support.

Product documentation

CONTENTS

Table of contents

Teleoperation

MANUS Metagloves Pro

Support

Troubleshooting

Contact Support

Best Practices for Reliable Use

Legal

1. Legal Notice
2. Repair

North

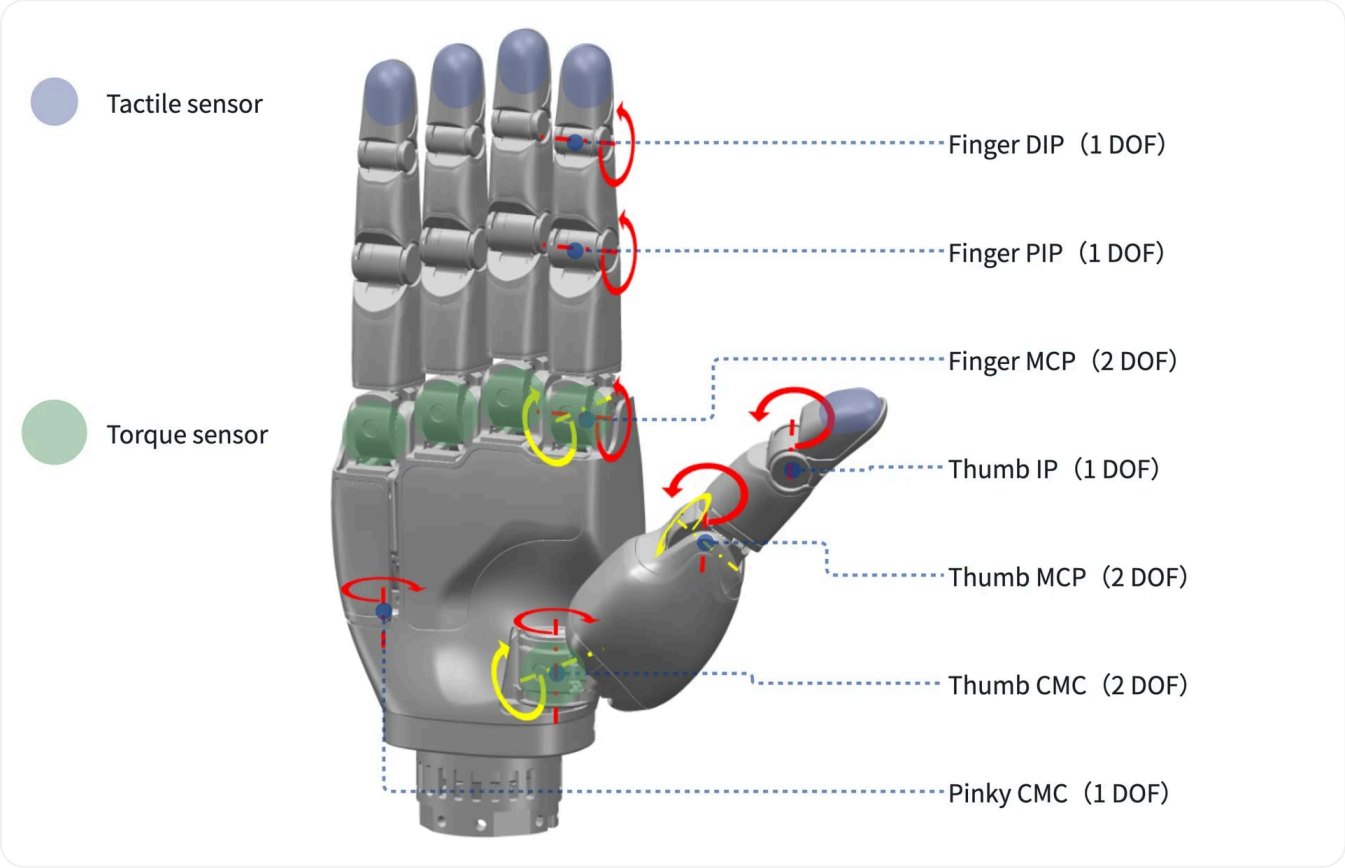
Overview

Product Overview

Sharpa Wave is a high-performance dexterous hand designed for research and advanced robotic manipulation. It enables precise interaction with tools and objects, supporting a wide range of complex tasks and experimental applications.

Key features:

- **22 active degrees of freedom** for highly dexterous manipulation
- **Human-like size and kinematics** for intuitive interaction and control
- **High-resolution tactile sensing** at each fingertip
- **Compliant joints** for safe and adaptive contact
- **High reliability** for continuous operation



Joint Naming Convention:

Joint names follow standard anatomical conventions:

- **Thumb**: CMC, MCP, IP
- **Fingers (index, middle, ring):**: MCP, PIP, DIP
- **Pinky**: CMC, MCP, PIP, DIP

The following abbreviations are used consistently in this manual and in SDK parameter names:

- **FE**: Flexion–Extension
- **AA**: Abduction–Adduction

Safety Instructions

1. General Safety Guidelines

- Read this manual before operating the product
- Use the device only as intended
- Do not disassemble the product unless explicitly permitted by Sharpa
- Always follow proper installation and operation procedures

2. Potential Hazards

2.1 Hot Surface

The product enclosure may become hot during or after operation.

- Avoid direct skin contact
- Allow at least **30 minutes** for cooling after power-off
- Keep flammable materials away

2.2 Pinch Points

Moving joints may create pinch hazards.

- Keep hands and fingers away from joint gaps during operation
- Do not place body parts near moving components

2.3 Electrical Safety

- Ensure all connectors are clean and dry before powering on
- Do not use damaged cables or connectors
- Use the provided power adapter or ensure compliance with specifications

2.4 Tactile Fingertips

- Avoid contact with sharp objects
- Maximum limits:
  - Pressure: < 2 MPa
  - Normal force: < 50 N
  - Sliding force: < 30 N

3. Operating Requirements

3.1 Personnel

- The product should be operated by personnel with an engineering background
- Follow all instructions during operation
- Contact Sharpa for technical support if needed

3.2 Medical Devices

This product may emit electromagnetic fields.

- If you use medical devices (e.g., pacemakers), consult a physician before use
- Stop using the product if interference is suspected

3.3 Environment

Do not use the product in:

- Explosive or flammable environments
- Corrosive chemical environments

- High humidity or liquid exposure conditions (IP20 rated)

Recommended storage:

- Temperature: 0–50°C
- Humidity: < 60% RH

3.4 Handling

- Do not drop, crush, or burn the product
- If the enclosure is damaged, stop using immediately
- Use proper packaging during transportation

4. Emergency Stop

Stop using the product immediately if:

- Abnormal noise or vibration is detected
- The product appears damaged
- Nearby equipment malfunctions
- Any person experiences discomfort

Contact **Sharpa Technical Support** for assistance at support@sharpa.com.

Specifications

1. General

PARAMETER	VALUE
Degrees of Freedom	22 DOF
Weight	1.3 kg
Dimensions	208 × 90 × 50 mm (Height × Width × Thickness)
Mounting Interface	Flange / Wrist / Internal

2. Performance

PARAMETER	VALUE
Maximum Active Fingertip Force	20 N
Minimum Grasp Diameter	10 mm
Fingertip Position Repeatability	±1 mm
Control Frequency	500 Hz

3. Electrical & Communication

PARAMETER	VALUE
Operating Voltage	18 – 28 V DC
Static Power Consumption	15 W
Peak Power Consumption	180 W ①
Communication Interface	Ethernet (100BASE-TX / 1000BASE-T)

① Peak power: 180 W (10 ms transient); average power: 100 W.

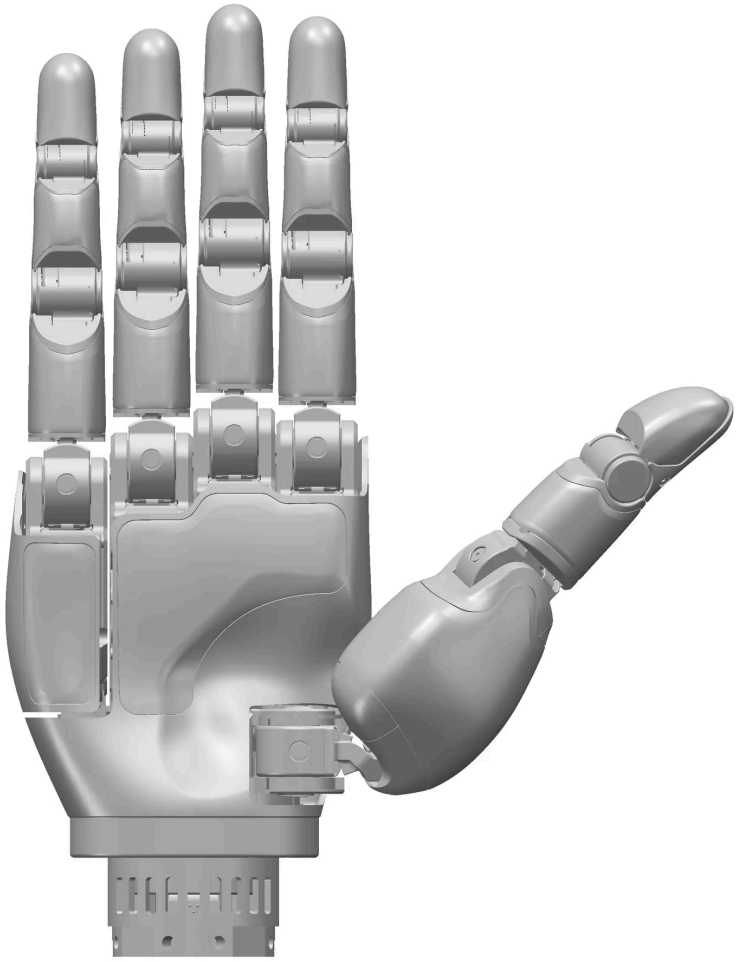
4. Range of Motion

JOINT GROUP		RANGE
Thumb	CMC FE	-5° to 105°
	CMC AA	-18° to 18°
	MCP FE	-25° to 75°
	MCP AA	-18° to 18°
	IP	0° to 95°
Index/Middle/Ring/Pinky	MCP FE	-10° to 84°
	MCP AA	-18° to 18°①
	PIP	0° to 95°
	DIP	0° to 75°
Pinky	CMC	0° to 25°

① Range restricted. Refer to "2-DOF Joint Workspace" in the Appendix for the actual workspace.

Notes

- Direction Definition
  - FE (Flexion-Extension)
    - Positive: toward the palm (flexion)
    - Negative: away from the palm (extension)
  - AA (Abduction-Adduction)
    - Positive: toward the pinky side
    - Negative: toward the thumb side
- Zero position definition
  - Defined as the neutral pose after calibration, as shown below:



- Joint Limits
  - MCP FE below -10° is reserved for collision avoidance

5. Sensing

5.1 Torque Sensors

PARAMETER	VALUE
Number of Sensors	5 (one per finger)
Sensor Location	CMC (Thumb) MCP (Index/Middle/Ring/Pinky)
Full-scale Range	5 N·m (Thumb) 3 N·m (Index/Middle/Ring/Pinky)
Noise (RMS)	≤ 0.005 N·m (Thumb) ≤ 0.003 N·m (Index/Middle/Ring/Pinky)
Total Error ①	±(1% FSR + 5% reading)
Sampling Rate	500 Hz

① Total error, specified at 25 °C after zero calibration.

5.2 Tactile Sensors

PARAMETER	VALUE
Number of Sensors	5 (one per fingertip)
Output Modalities	6-DOF force Deformation map Contact point Raw image
Force Resolution	20 mN
Spatial Resolution	1 mm
Measurement Range	30 N
Maximum Load ①	50 N
Sampling Rate	Standard Mode: 30 Hz (raw image stream & processed outputs) High-Performance Mode: up to 180 Hz (raw image stream)
Latency	50ms (Standard mode) 20ms (High-Performance mode)

① Maximum load: Maximum allowable force without causing sensor damage

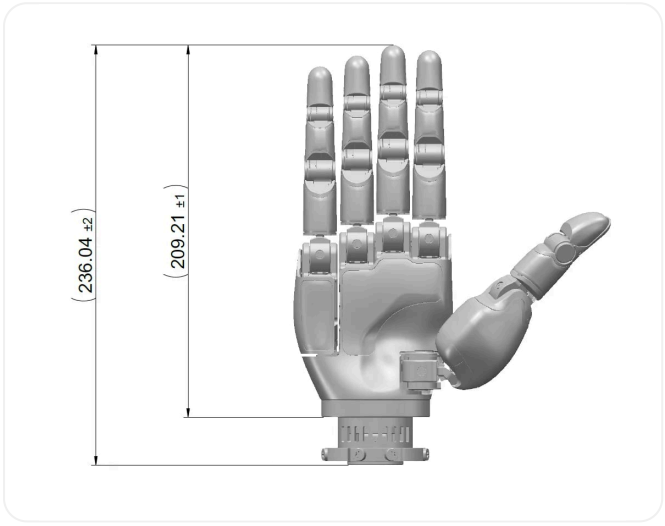
6. Environmental & Reliability

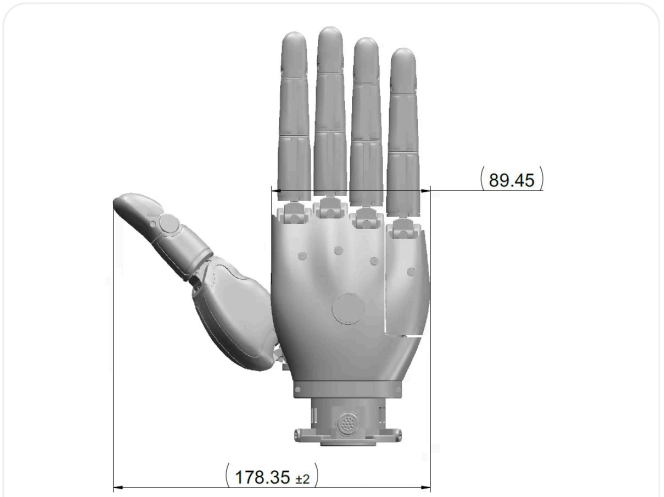
PARAMETER	VALUE
Operating Temperature	0°C to 45°C
Storage Temperature	0°C to 50°C
Humidity	< 60% RH
IP Rating	IP20
Nominal Lifetime	> 1,000,000 cycles (no load)
Tactile Sensor Lifetime	> 100,000 cycles @ 40 N

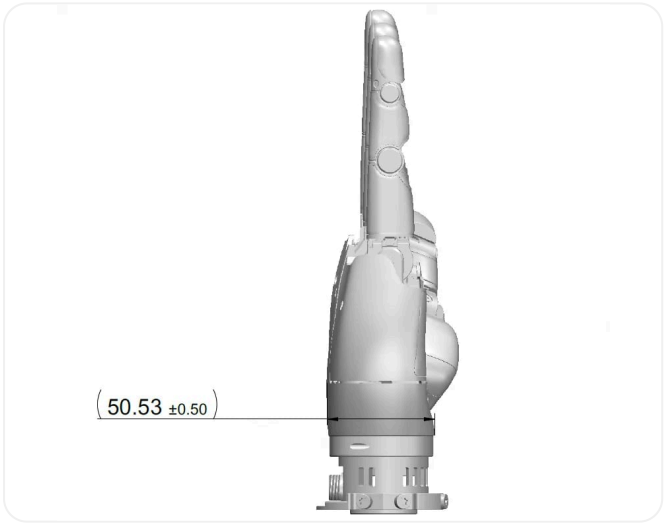
7. Appendix

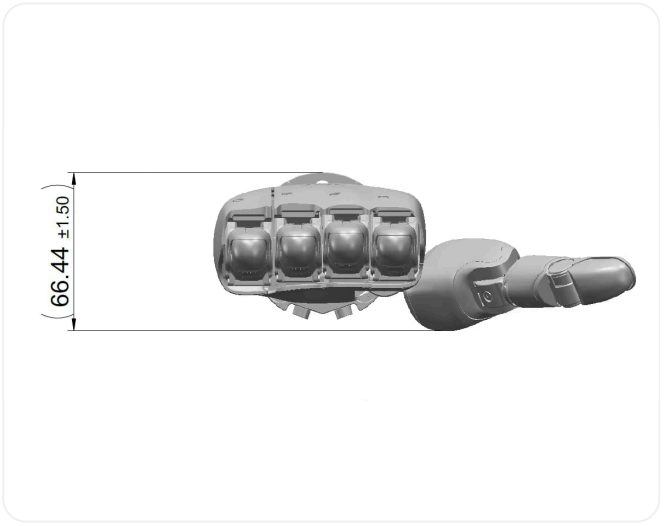
7.1 Mechanical Dimensions

The overall dimensions of the right hand are shown below (measured at the zero position, i.e., the neutral pose after calibration).

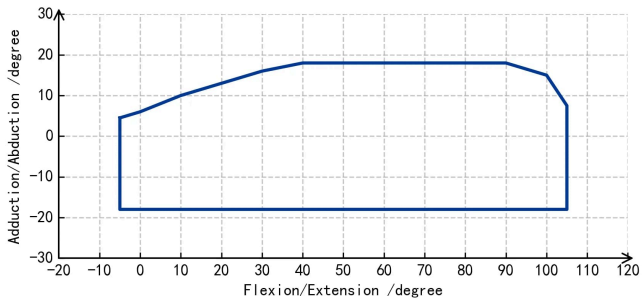




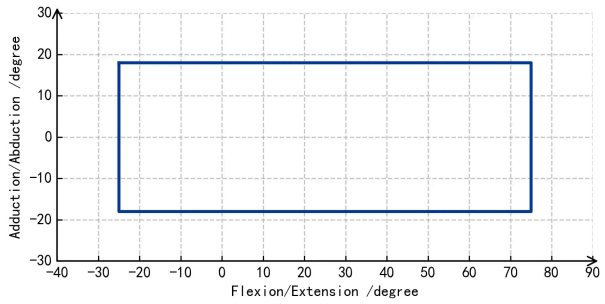




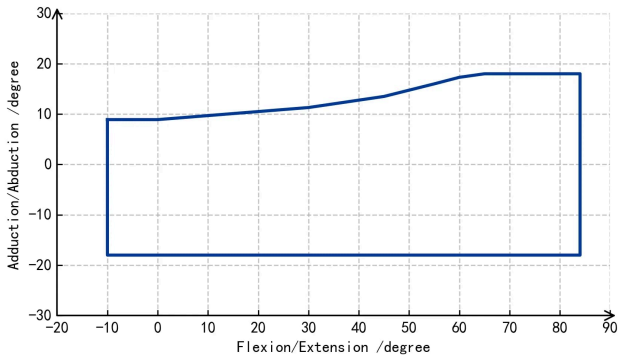
7.2 2-DOF Joint Workspace



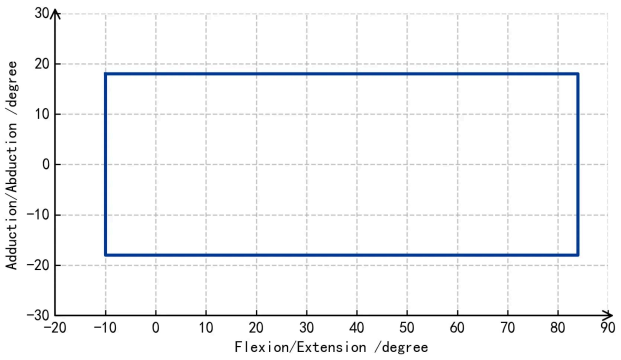
Thumb CMC



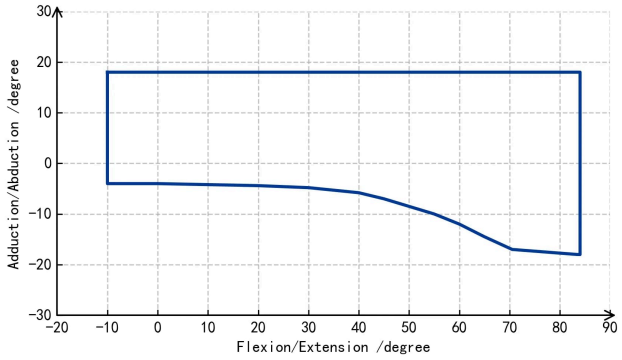
Thumb MCP



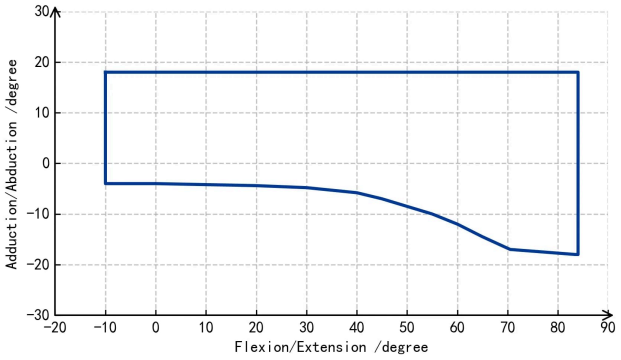
Index Finger MCP



Middle Finger MCP



Ring Finger MCP



Pinky MCP

Get Started

1. Before You Start

Before operating the device:

- Ensure the product is properly installed and placed on a stable surface
- Read the Safety Instructions carefully
- Prepare a host PC for running Sharpa Pilot app

2. Set Up Hardware

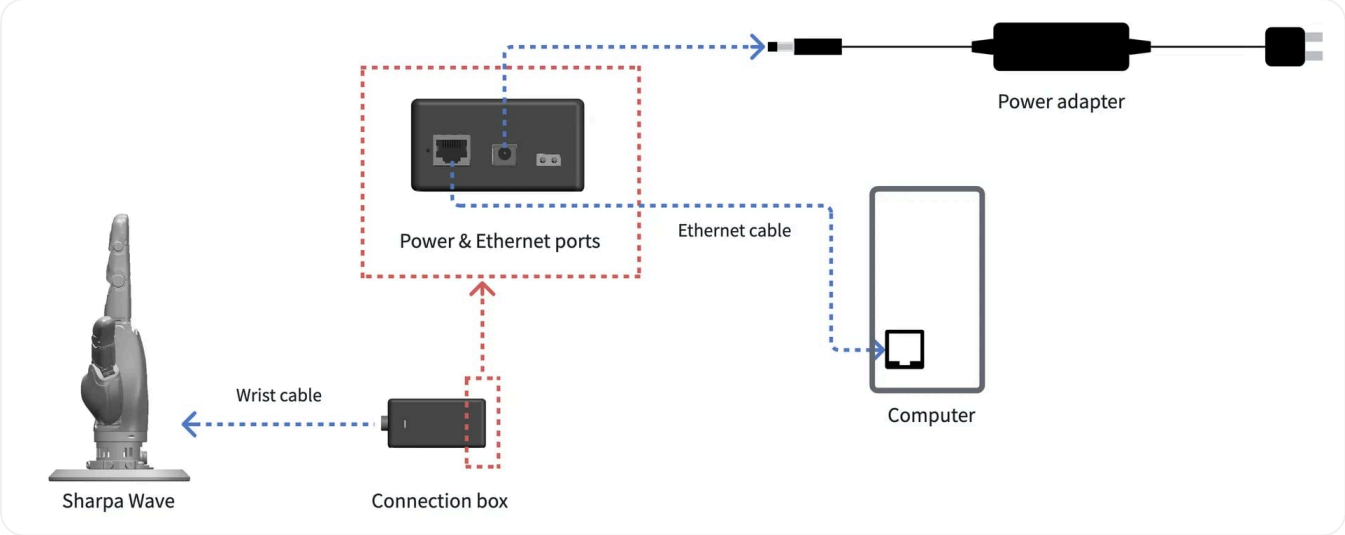
2.1 Mount the Hand

1. Attach the flange adapter to the base using M6 screws (diagonal tightening recommended)
2. Place the hand onto the adapter
3. Secure the hand using M4 screws

2.2 Place the Assembly

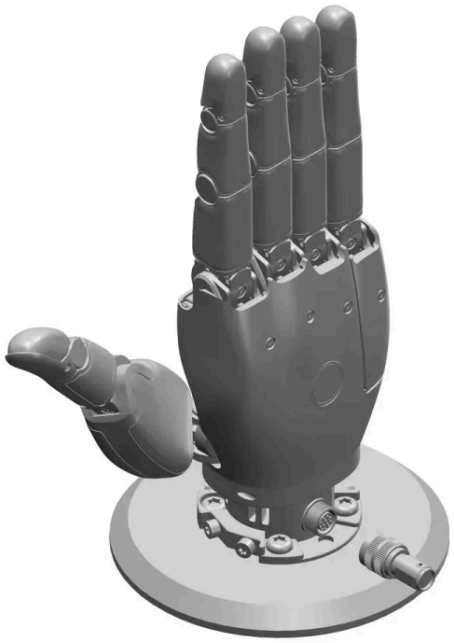
- Put the assembled unit on a flat, stable workspace

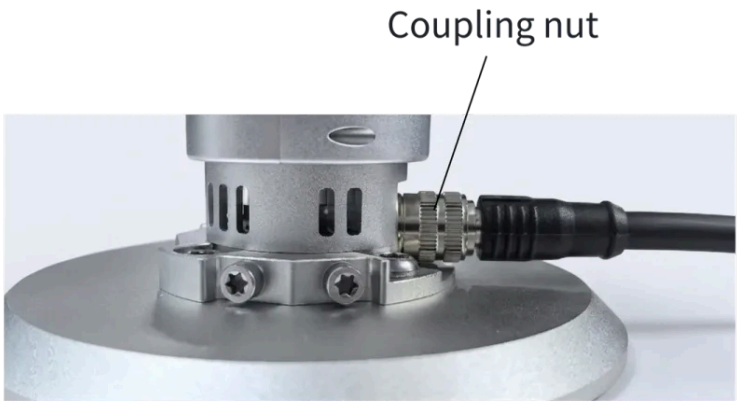
2.3 Connect the Cable

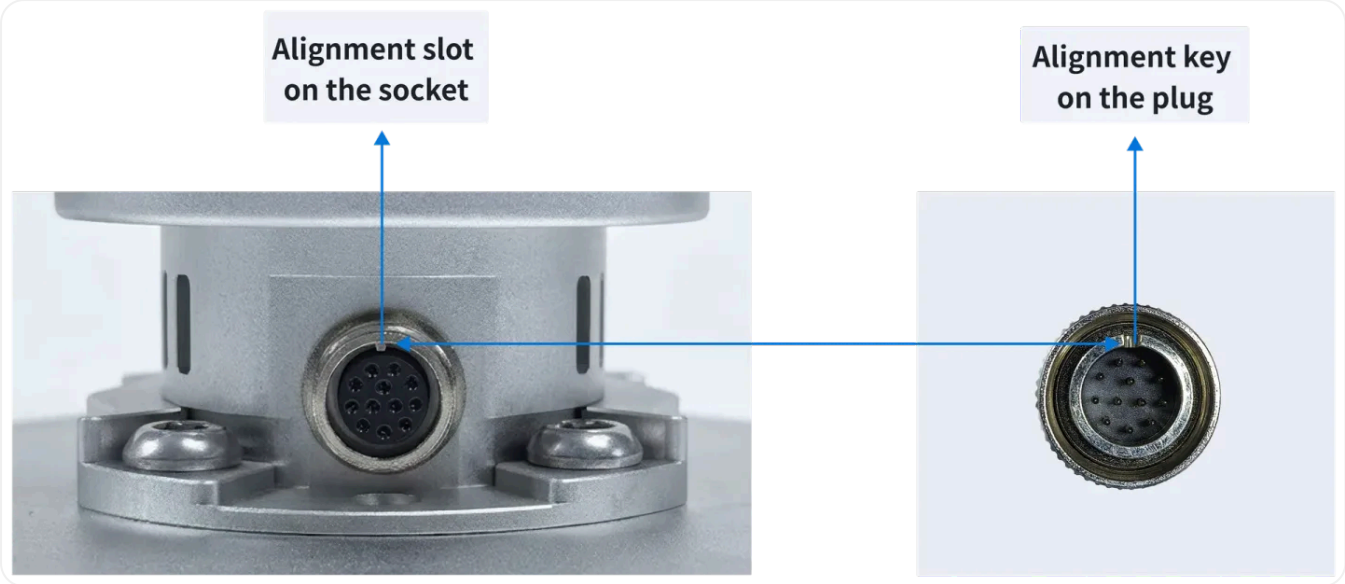


1. **Ensure power is OFF**
2. **Connect the wrist cable**

- Connect one end to the hand and the other end to the connection box
- Align the connector key with the socket
- Insert along the axis
- Tighten the coupling nut until fully locked







- Ensure proper alignment before inserting the wrist connector
- Do not force insertion
- Avoid pulling or bending the cable excessively

3. **Connect the Ethernet cable**

- Connect one end to the connection box and the other end to the computer

4. **Connect the power adapter**

- Connect the power adapter to the connection box, then plug it into a power outlet

3. Power On

3.1 Turn On the Device

After ~20 seconds:

- The rear hand indicator turns **cyan (breathing)** → device initialized
- The device becomes discoverable by the host PC

- During startup, keep fingertips free from contact

3.2 Indicator Check

LOCATION	STATUS	MEANING
Connection box	Solid green	Power OK
Hand (rear)	Breathing cyan	Connected & ready

4. Install Sharpa Pilot

4.1 Software Requirements

- OS: Ubuntu 22.04 / 24.04

4.2 Installation

Download Sharpa Pilot and install:

```
TEXT

sudo dpkg -i sharpa-pilot_xxx.deb
```

5. Connect to the Device

This section summarizes the factory default network settings of Sharpa Wave and explains how to connect the device to the host PC.

5.1 Default Network Settings

Use these values for first-time setup, device discovery, troubleshooting, and recovery after a reset. If the network settings have been intentionally changed, use the current configured values instead.

DEVICE	PARAMETER	LEFT HAND	RIGHT HAND
Sharpa Wave	IP address	192.168.10.10	192.168.10.20
	Tactile destination IP (dest_ip)	192.168.10.240	192.168.10.240
	Heartbeat port (heart_port)	54321	54321
	Tactile stream port (stream_port)	50011	50001
Host PC	IP address (same as dest_ip)	192.168.10.240	
	Subnet mask	255.255.255.0	

5.2 Configure PC Network

Configure the host PC according to the values in Section 5.1, Default Network Settings.

Notes

- Ensure that no other device on the network is using the same IP address as the host PC or Sharpa Wave.
- For instructions on changing the device IP address or tactile destination IP, see Sharpa Pilot.

6. Calibrate the Joints

Joint calibration establishes the zero reference used by the control system. **It is recommended each time the device is powered on to ensure accurate joint control.**

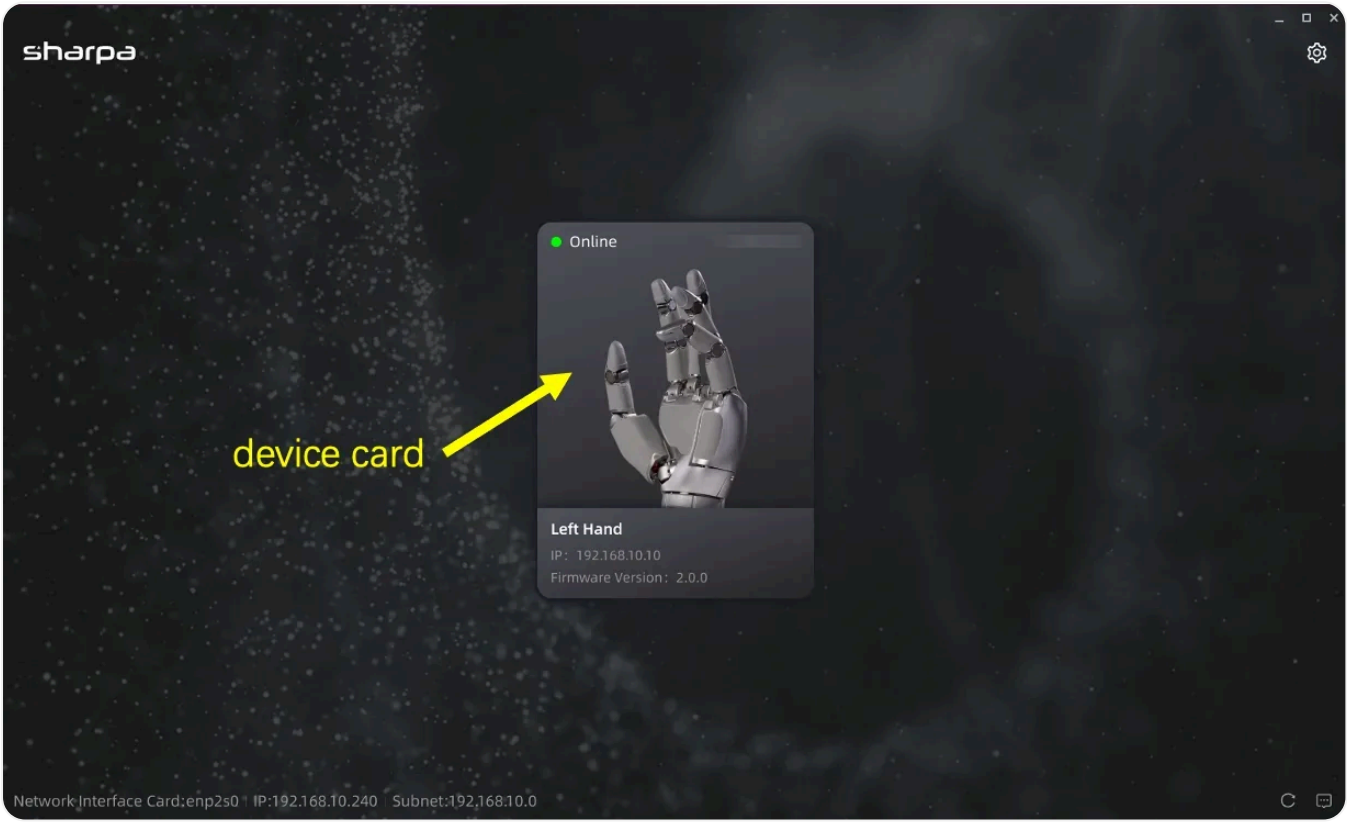
6.1 Preparation

Before calibration, ensure that:

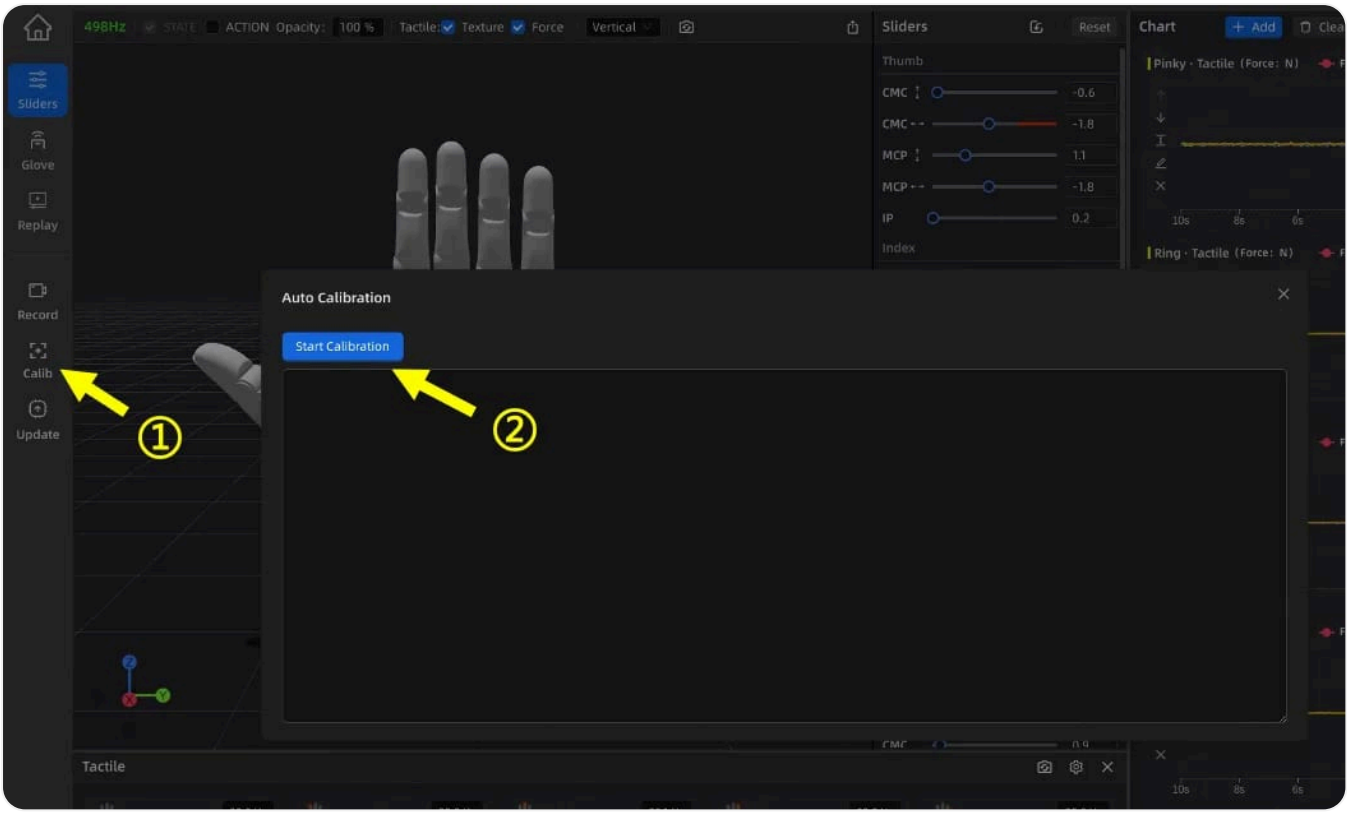
- the hand is securely mounted;
- the workspace is free of obstacles;
- no person is in contact with the hand.

6.2 Calibration Procedure

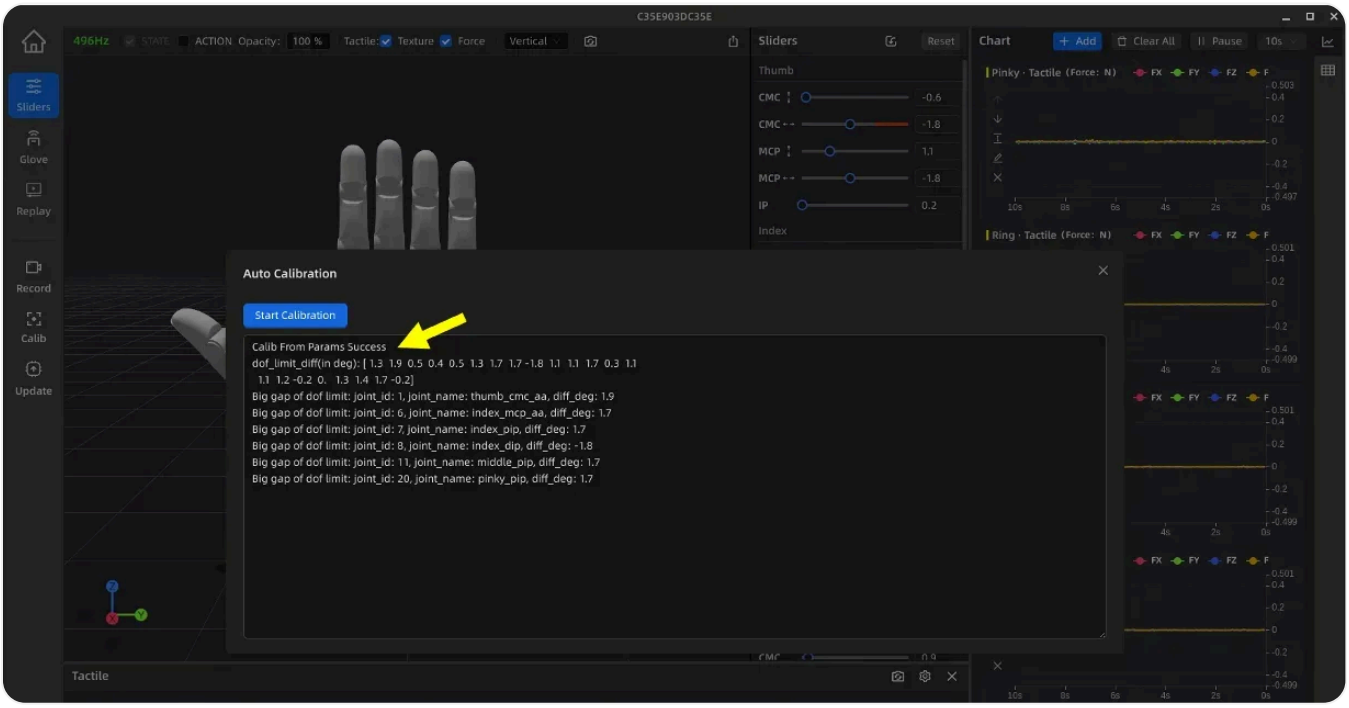
1. Launch **Sharpa Pilot**.
2. Wait until the device card appears.
3. Click the device card.



4. Open the **Calib** panel and click **Start Calibration**.



5. Wait for the automatic calibration sequence to complete. The hand will return to the zero position.
6. Confirm that the calibration completion message appears (~95 seconds).

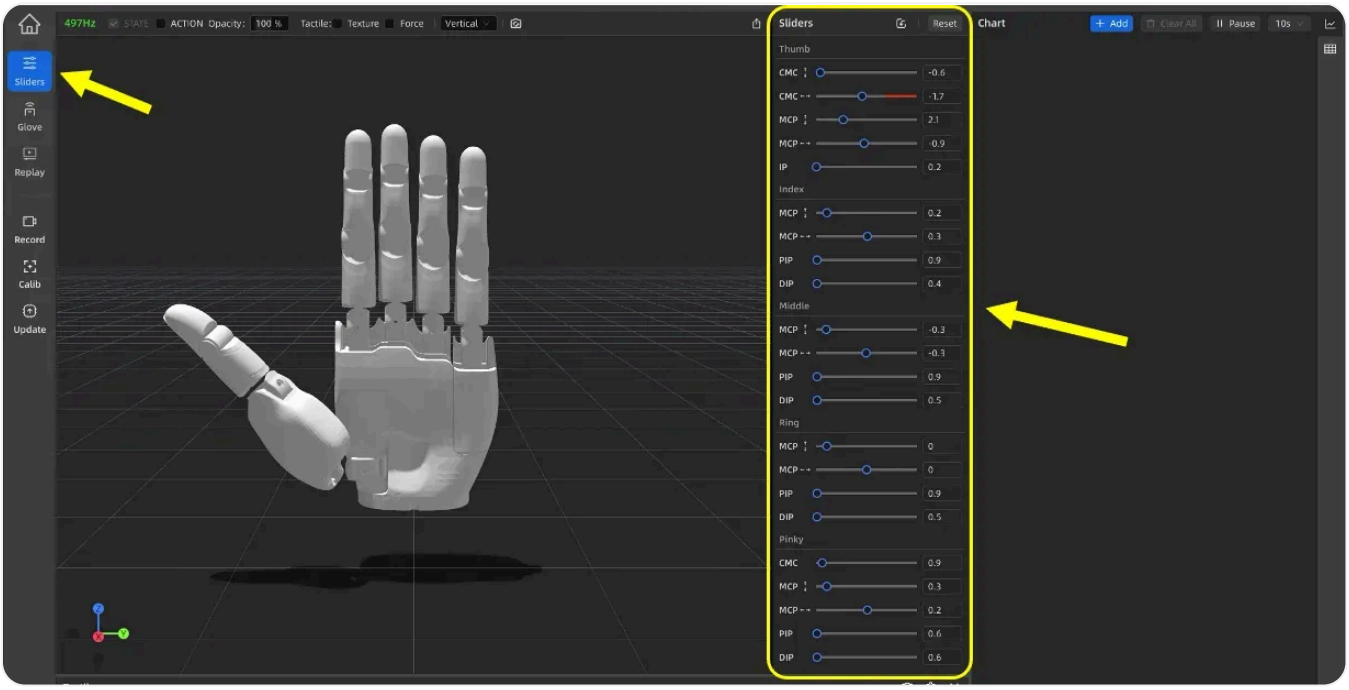


6.3 Notes

- After calibration, the system reports joint backlash variation.
- Joints with a deviation greater than 1.5° are highlighted.
- Check these joints before operation to ensure that they do not affect performance.

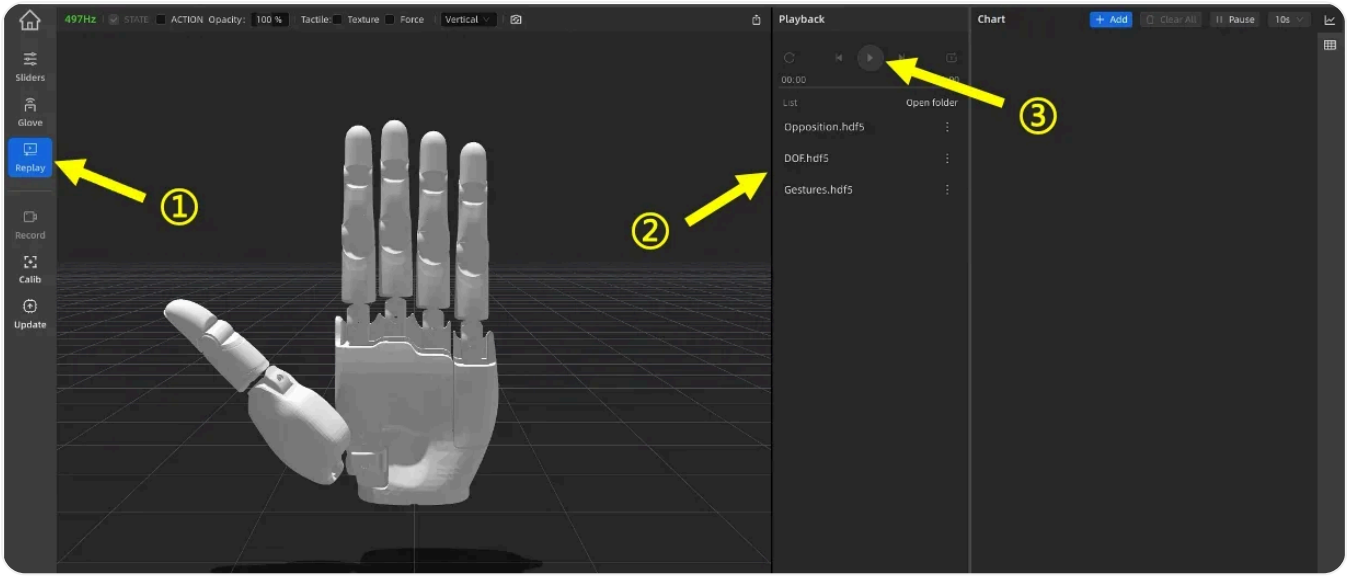
7. Test Hand Movement

1. Open **Sliders** panel in Sharpa Pilot
2. Drag sliders to move joints



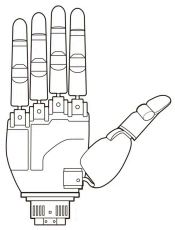
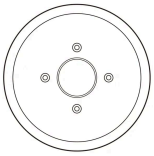
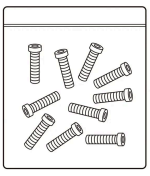
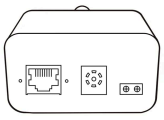
8. Run a Demo

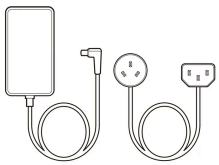
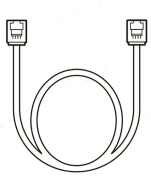
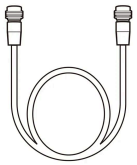
1. Go to **Replay** panel
2. Select a preset motion
3. Click **Play**


497Hz
STATE
ACTION
Opacity:
100 %
Tactile
Texture
Force
Vertical
Sliders
Glove
Replay
Record
Calib
Update
Playback
00:00
List
Open folder
Opposition.hdf5
DOF.hdf5
Gestures.hdf5
Chart
Add
Clear All
Pause
10s
①
②
③

User Guide

Package Contents

NO	ILLUSTRATION	ITEM	NOTES
1		Sharpa Wave	Main device
2		Flange adapter	Used to mount the hand onto a base or robot interface
3		Base	Provides stable support for bench-top operation
4		L-shaped wrench	Used for installation and fastening screws
5		Screws	M4 × 12 (×10), M6 × 10 (×8)
6		Connection box	Provides power input and Ethernet communication interface

NO	ILLUSTRATION	ITEM	NOTES
7		Power adapter	Converts AC to DC power for the device (cable length: 3.5 m)
8		Ethernet cable	Used for network connection between device and PC
9		Wrist cable	Connects the hand to the connection box (length: 2 m)

Hand Control

This section describes how the hand is controlled, what information it provides, and how to operate it effectively. It covers control logic, state feedback, command behavior, joint definitions, and a basic operation workflow.

1. Control Basics

This section defines how control is managed in the system, including control sources and the rules that govern command execution.

1.1 Control Sources

The system supports the following control sources. Each source represents a different way to control the hand and provides different capabilities.

CONTROL SOURCE	DESCRIPTION	CAPABILITIES	TYPICAL USE
APP (Sharpa Pilot)	Manual control via GUI	GUI-based interface for control, monitoring, and configuration • Manual control via sliders and motion replay • Real-time visualization of joint states and tactile data • Configuration of control source, control mode, and parameters • Calibration tools for joints and tactile sensors • Fault detection and system status monitoring	Testing, demonstration, manual operation
SDK (C++ / Python)	Programmatic control via API	Programmatic interface providing full access to all control and monitoring capabilities • Send joint commands and execute motion control • Read joint states and tactile data • Configure control source, control mode, and system parameters • Perform calibration procedures • Monitor device status and fault information • Integrate with external systems and control algorithms	Automation, algorithm development, teleoperation
GLOVE	Teleoperation input device	Real-time motion input	Teleoperation, e.g.: Manus Pro Glove
IDLE	No motion control	Disables motion commands	Safe standby

1.2 Control Rules

- Only **one control source** can send motion commands at a time
- Motion commands from other sources are ignored

- Configuration commands remain valid across sources
- When set to **IDLE**:
  - All motion commands are ignored
  - The hand remains non-actuated

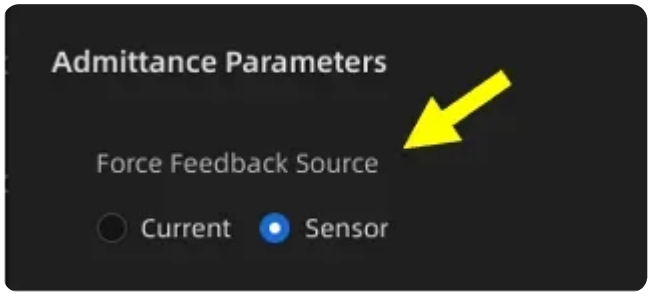
> - Ensure only one active controller is sending motion commands.
> - Conflicts may result in unexpected motion.

2. State Feedback

The hand provides real-time feedback on joint states via Sharpa Pilot or the SDK.

STATE	DESCRIPTION	UNIT IN SDK	UNIT IN SHARPA PILOT	UPDATE RATES
Angle	Current joint angle	rad, deg	deg	500 Hz
Velocity	Joint angular velocity	rad/s, deg/s	deg/s	500 Hz
Torque ①	Joint load feedback (measured or estimated)	Nm	Nm	500 Hz

① Torque Source
By default, joint torque feedback is estimated from motor current.
For the MCP joints of the four fingers (FE and AA) and the CMC joints of the thumb (FE and AA), torque feedback can alternatively be obtained from integrated force sensors.
The torque source can be switched in Sharpa Pilot (Hardware Settings → Admittance Parameters → Force Feedback Source).



When switching to sensor-based torque feedback, force sensor calibration must be performed beforehand. Calibration can be completed in Sharpa Pilot (Hardware Settings → Force Sensor Calibration).

3. Control Modes

The hand supports multiple control modes, each designed for different types of tasks and interaction patterns.

Selecting the appropriate mode is essential for achieving stable and effective behavior.

MODE	CONTROL PRINCIPLE	BEHAVIOR	TYPICAL APPLICATIONS	RECOMMENDED SCENARIO
Position	Position control	Tracks target joint angles with high accuracy; resists disturbances	Pick & place, welding, dispensing	Precise, repeatable motion
Admittance	Force → motion	Motion adapts to external force; compliant interaction	Assembly, polishing, collaboration	Interactive or compliant tasks
MIT	Torque / impedance	Controls torque via stiffness and damping; stable contact interaction	Precision insertion, contact inspection	Contact-rich or force-sensitive tasks
Zero-Force	No active control	Backdrivable; can be moved freely by external forces	Teaching, demonstration, setup	Manual guidance and teaching

Notes

- Control modes can be switched at runtime via Sharpa Pilot or Sharpa Wave SDK.
- Always ensure the correct control source is active before sending commands
- Calibration is recommended before executing precise motion
- For smooth motion, avoid large discontinuities in target positions

4. Joint Order Definitions

Joint commands and state arrays follow a fixed index order:

INDEX	JOINT NAME	FINGER	TYPE
0	Thumb CMC FE	Thumb	FE
1	Thumb CMC AA	Thumb	AA
2	Thumb MCP FE	Thumb	FE
3	Thumb MCP AA	Thumb	AA
4	Thumb IP	Thumb	FE
5	Index MCP FE	Index	FE
6	Index MCP AA	Index	AA
7	Index PIP	Index	FE
8	Index DIP	Index	FE
9	Middle MCP FE	Middle	FE
10	Middle MCP AA	Middle	AA
11	Middle PIP	Middle	FE
12	Middle DIP	Middle	FE
13	Ring MCP FE	Ring	FE
14	Ring MCP AA	Ring	AA
15	Ring PIP	Ring	FE
16	Ring DIP	Ring	FE
17	Pinky CMC	Pinky	FE
18	Pinky MCP FE	Pinky	FE
19	Pinky MCP AA	Pinky	AA
20	Pinky PIP	Pinky	FE
21	Pinky DIP	Pinky	FE

5. Control via SDK

The hand can be controlled programmatically via the SDK, which provides full access to motion control, state feedback, and system configuration.

The SDK is available in **C++ and Python**. For detailed API definitions and examples, refer to the Sharpa Wave SDK.

5.1 Getting Started

Before using the SDK:

- Ensure the hand is properly connected and powered on
- Ensure the control source is set to **SDK** (via Sharpa Pilot or API)

API Example:

TEXT

```
error = hand.set_control_source(ControlSource.SDK)
if error.code != 0:
    print(f"Failed to set control source: {error.message}")
```

5.2 System Requirements

COMPONENT	REQUIREMENT
OS	Ubuntu 22.04 / 24.04
Python SDK	Python 3.10, protobuf 3.17.2
C++ SDK	cmake ≥ 3.15 and < 4 (3.28.1 recommended)

5.3 Running the Examples

1. Download Sharpa Wave SDK from Sharpa Wave SDK
2. Follow the instructions in `README.md`

3. Run the provided examples:

- `Python: SharpaWaveSDK_x.x.x/sample/python/sharpa_wave_example.py`
- `C++: SharpaWaveSDK_x.x.x/sample/c++/sharpa_wave_example.cc`

Tactile Sensing

1. Overview

Sharpa Wave is equipped with vision-based tactile sensors at each fingertip. The sensors measure contact by capturing deformation of a compliant surface using an internal camera and processing it with onboard algorithms.

The tactile system provides real-time information about:

- contact occurrence
- contact force
- contact location and distribution on the fingertip surface

Tactile data is streamed to the host system, where it may be processed either onboard or on the host depending on the selected mode.

2. Tactile Outputs

Each fingertip provides multiple types of tactile data. These outputs describe the same contact event from different perspectives and can be used independently or jointly, depending on the application.

OUTPUT	DESCRIPTION	USE WHEN YOU NEED	TYPICAL APPLICATIONS	DATA RATE
6-DOF Force	Resultant contact force on the fingertip. Available in Sharpa Pilot and SDK	Detect contact, measure force magnitude or direction	Contact detection, force control, grasp monitoring	30 Hz
Deformation Map	2D distribution of surface deformation. Available in Sharpa Pilot and SDK	Understand where and how contact occurs	Contact localization, contact area estimation, shape analysis	30 Hz
Contact Point	Estimated contact location on the fingertip surface, derived from deformation data. Available in Sharpa Pilot and SDK	Identify a representative contact position	Contact tracking, object pose estimation, control reference point	30 Hz
Raw Image	Original camera image inside the sensor. Available via SDK	Access low-level sensor data	Debugging, custom tactile algorithms, data-driven modeling	Up to 180 Hz

2.1 Output Selection Guidelines

- Use **force** as the primary signal for most applications.
- Use **deformation map** when spatial contact distribution is required.
- Use **contact point** when a compact representation of contact location is sufficient.
- Use all outputs for data-driven or learning-based applications, and perform feature selection during model development.

2.2 Data Rate Selection Guidelines

Sharpa Wave supports two tactile processing modes:

- **Standard (Onboard) Mode**: Tactile inference is performed on the device.
- **High-Performance (Host) Mode**: Raw tactile data is streamed to the host, where inference is performed using GPU acceleration.

FEATURE	STANDARD MODE	HIGH-PERFORMANCE MODE
Data rate	30 Hz	Up to 180 Hz
Processed outputs (force, deformation, contact point)	Onboard	Host (GPU)
Raw image	Available	Available
System complexity	Low	High
Hardware requirement	• No GPU required • Ubuntu 22.04 / 24.04	• NVIDIA GPU required (RTX 3060+ recommended) • Ubuntu 22.04 / 24.04 • NVIDIA driver • NVIDIA Container Toolkit • Docker Engine
Latency	Moderate	Lower (higher temporal resolution)
Typical use	Standard applications	High-performance / research
SDK Type	Standard SDK	Docker-based SDK

Use Standard Mode when:

- a simple and stable setup is preferred
- no GPU is available
- standard tactile feedback is sufficient
- low-frequency data is sufficient (e.g., monitoring, logging, basic control)

Use High-Performance Mode when:

- higher temporal resolution is required
- real-time or high-frequency processing is needed
- advanced applications (e.g., learning or custom algorithms) require high-frequency data

3. Using Tactile Data via SDK

The SDK supports two tactile data pipelines:

- **Standard mode: Standard SDK**, no Docker required
- **High-Performance mode: Docker-based SDK** with GPU acceleration

These modes use different SDK packages and deployment workflows.

Important:

Tactile data access via the SDK cannot be used simultaneously with Sharpa Pilot. This applies to both the Standard SDK and the Docker-based SDK. Both SDK pipelines and Sharpa Pilot use the same communication ports for tactile data streaming, which will result in conflicts. **Close Sharpa Pilot before running any SDK-based tactile application.**

3.1 Standard Mode

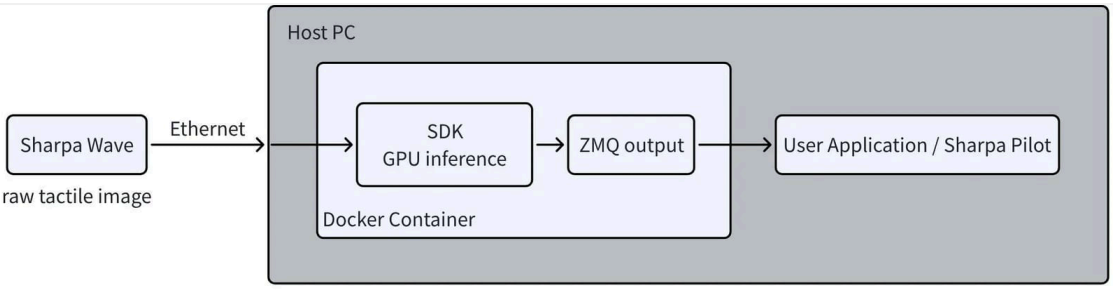
In this mode, tactile data is processed inside the device and streamed to the host.

Use the standard Sharpa Wave SDK and run the provided examples:

- Python
  - `SharpaWaveSDK_x.x.x/sample/python/sharpa_tactile_fetch.py`
  - `SharpaWaveSDK_x.x.x/sample/python/sharpa_tactile_callback.py`
- C++
  - `SharpaWaveSDK_x.x.x/sample/c++/sharpa_tactile_example.cc`

3.2 High-Performance Mode

In this mode, the device streams raw tactile images to the host, and tactile inference is performed on the host using GPU acceleration via the **Docker-based SDK.**



This section describes how to deploy and run the high-performance tactile pipeline.

1. **Install Docker and GPU Runtime**

Install Docker Engine:

```text
https://docs.docker.com/engine/install/ubuntu/
```

Add user to Docker group:

```bash
sudo usermod -aG docker $USER
reboot
```

Install NVIDIA Container Toolkit:

```text
https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html
```

Configure GPU runtime:

```bash
sudo nvidia-ctk runtime configure --runtime=docker
```

```
sudo systemctl restart docker
```

2. Prepare Docker Configuration

- Create a `docker-compose.yml` file:

BASH

```
services:
  tactile_dev:
    image: sharpadev/sharpawavesdk:<image tag>
    working_dir: /root
    container_name: sharpa_dev
    runtime: nvidia
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: all
              capabilities: [ gpu ]
    network_mode: host
    privileged: true
    volumes:
      - /tmp/.X11-unix:/tmp/.X11-unix
    environment:
      - DISPLAY=${DISPLAY}
      - QT_X11_NO_MITSHM=1
    cap_add:
      - SYS_ADMIN
    devices:
      - /dev/fuse:/dev/fuse
    security_opt:
      - apparmor:unconfined
    stdin_open: true
    tty: true
    restart: always
```

3. **Select and Pull Docker Image**

Check your GPU and driver:

```text
nvidia-smi
```

Then choose:

- here we take the docker image sharpadev/sharpawavesdk:5.0.0.8b7e for example, you can get the latest Docker image from https://hub.docker.com/r/sharpadev/sharpawavesdk
- RTX 40 series + driver ~570:

```bash
docker pull sharpadev/sharpawavesdk:5.0.0.8b7e.cu124
```

- RTX 50 series + driver ~580:

```bash
docker pull sharpadev/sharpawavesdk:5.0.0.8b7e
```

Update `<image tag>` in `docker-compose.yml` accordingly:

```yaml
services:
  tactile_dev:
    image: sharpadev/sharpawavesdk:5.0.0.8b7e
```

4. **Start Docker Container**

```bash
xhost +local:root
docker compose -f docker-compose.yml up -d
```

If Docker container already exists:

```bash
docker stop sharpa_dev && docker rm sharpa_dev
xhost +local:root
docker compose -f docker-compose.yml up -d
```

5. **Enter Docker Container**

```json
docker exec -it sharpa_dev bash
```

6. **Configure Tactile Data Rate**

The tactile data rate is configured by editing the following file inside the Docker container:

```text
$HOME/.sharpa-pilot/config/tactile.json
```

Key Parameters

PARAMETER	DESCRIPTION	HIGH-PERFORMANCE MODE	STANDARD MODE
`fps`	Tactile data rate Type: int (must be integer)	180 (recommended) Range: 0–180	30 (recommended) Range: 0–30
`infer_from_device`	Select inference location	false	true
`require_jpeg`	Enable raw image streaming	true	as needed (true: transmit, false: disable)

Ensure the following parameters are set:

JSON

```
{
  ... ...
  ... ...
  ... ...
  "left": {
    ... ...
    ... ...
    "fps": 180,
    "infer_from_device": false,
    "require_jpeg": true,
    "decode_jpeg": true
    ... ...
    ... ...
  },
  "right": {
    ... ...
    ... ...
    "fps": 180,
    "infer_from_device": false,
    "require_jpeg": true,
    "decode_jpeg": true
    ... ...
    ... ...
```

```
  }
}
```

Compatibility Note (Older Versions)

In older versions of tactile.json, the configuration is split into two sections:

- "none": used by the Standard SDK
- "cuda": used by the Docker-based SDK

For High-Performamce mode, modify only the parameters under "cuda".

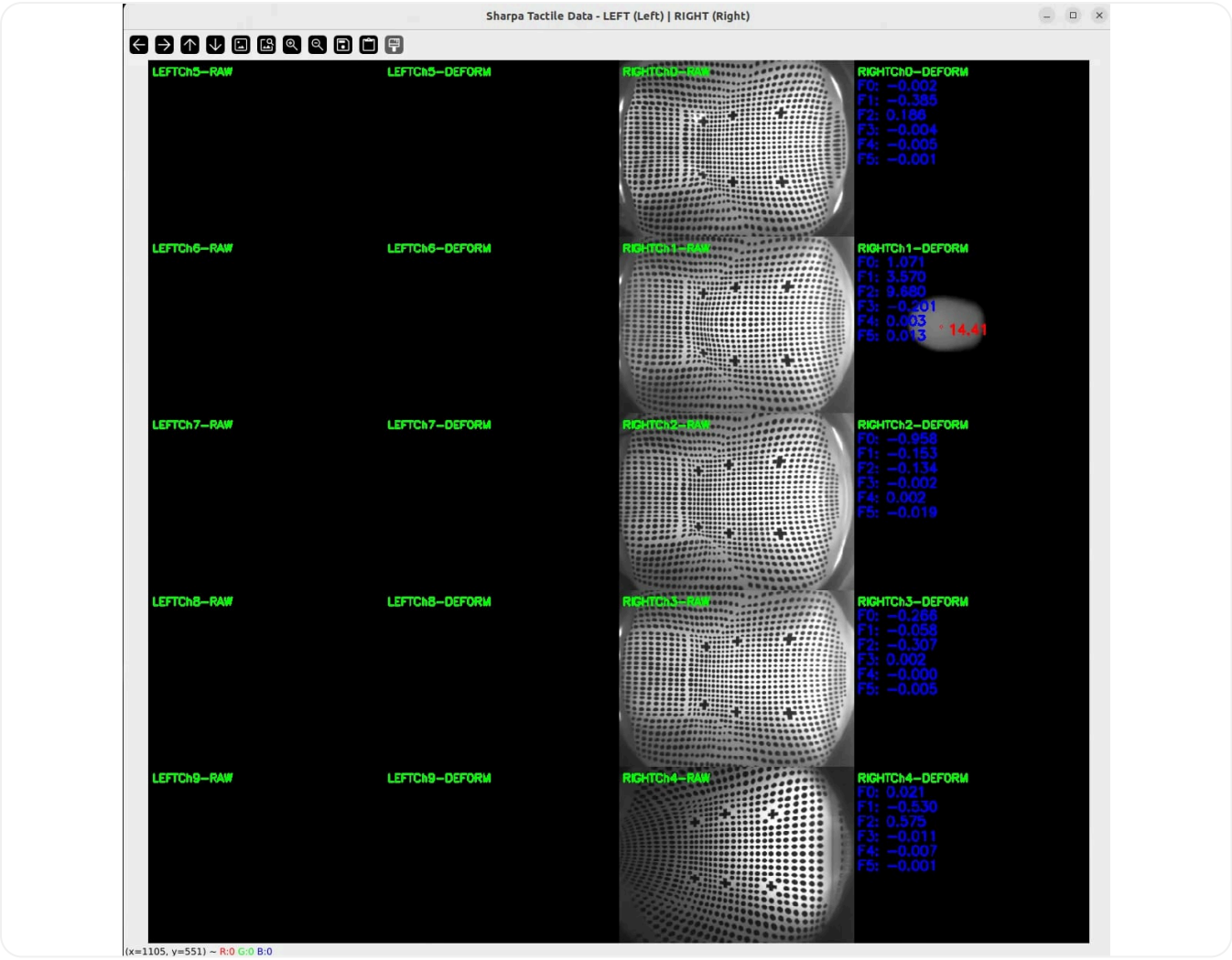
7. **Run High-Performance Tactile Pipeline**

Run:

BASH

```
python3 scripts/tactile_180fps.py
```

A window will appear. When you press the elastomer on a fingertip (e.g., the ring finger), the deformation map in the window will update accordingly.



In the figure, channels 0–4 correspond to the right hand tactile sensors. In this example, only the right hand is connected, so the left-hand channels appear black because no data is received. For channel indexing and definitions, see the Tactile Channel Mapping section.

4. Visualization in Sharpa Pilot

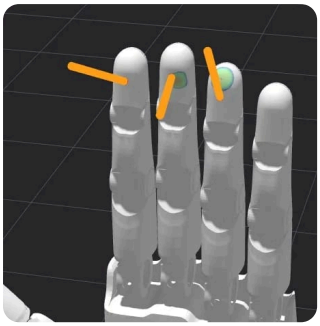
4.1 Overview

Sharpa Pilot provides real-time visualization of tactile data for both Standard and High-Performance modes.

Available views include:

- force vector
- contact distribution overlaid on the hand model
- contact point information

- real-time force components (Fx, Fy, Fz) and resultant magnitude



contact & force vector

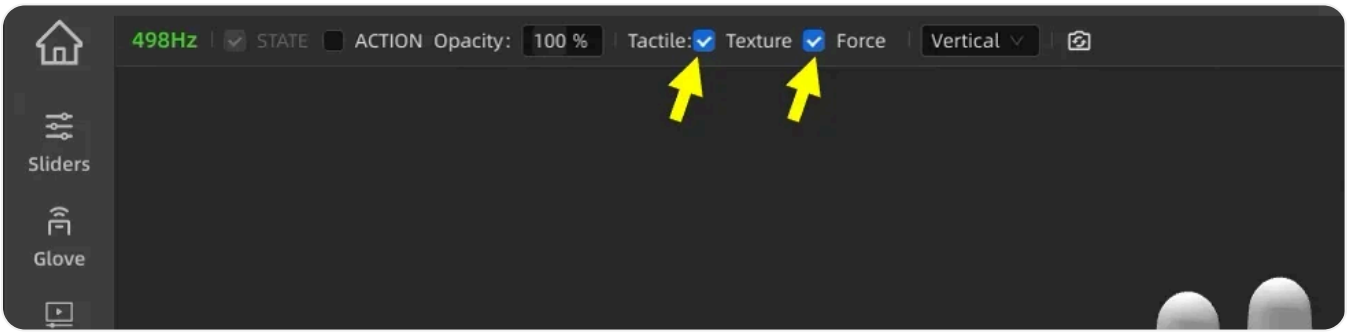
contact point

force components and magnitude plot

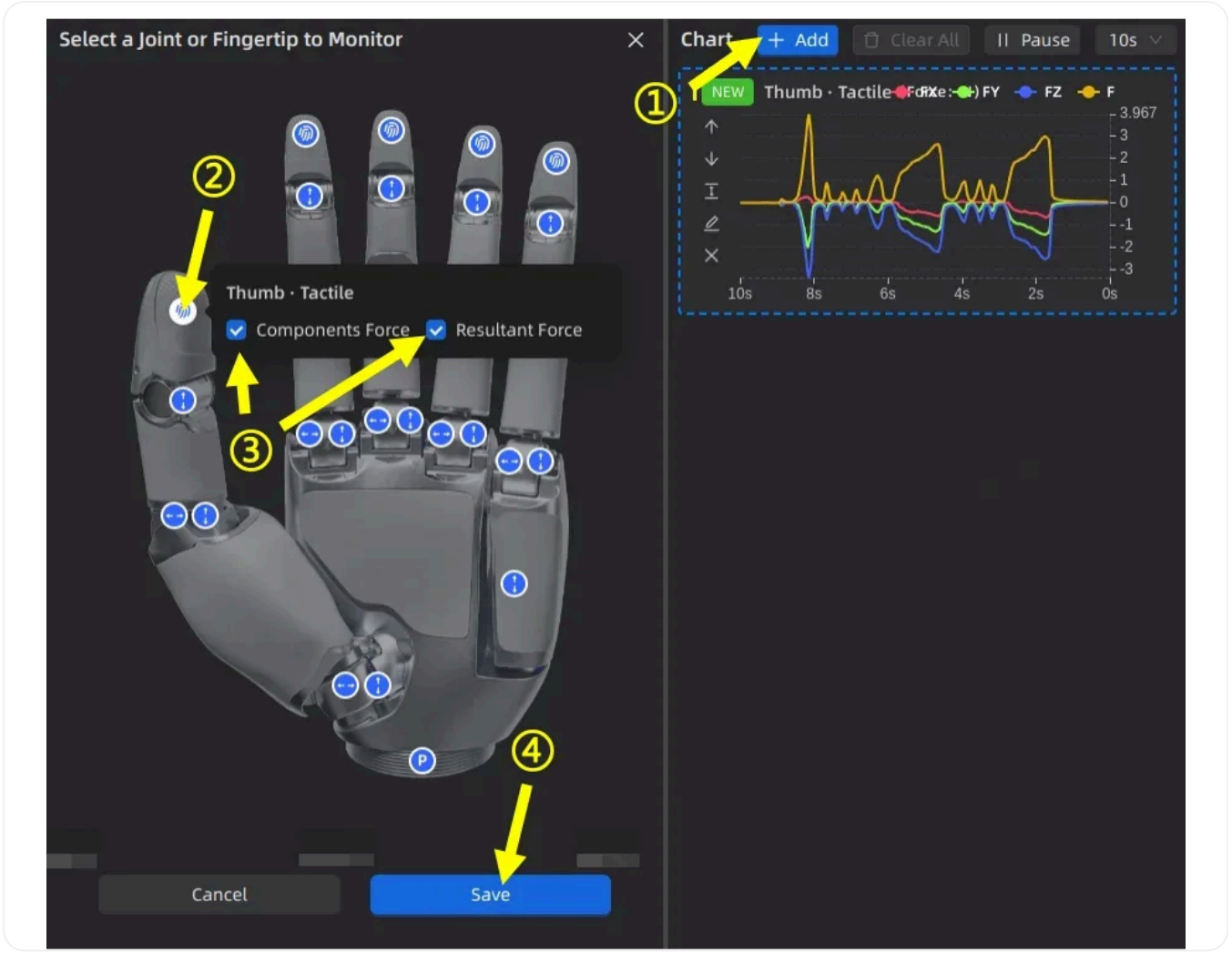
4.2 Visualization in Standard Mode

By default, Sharpa Pilot uses the standard onboard tactile pipeline.

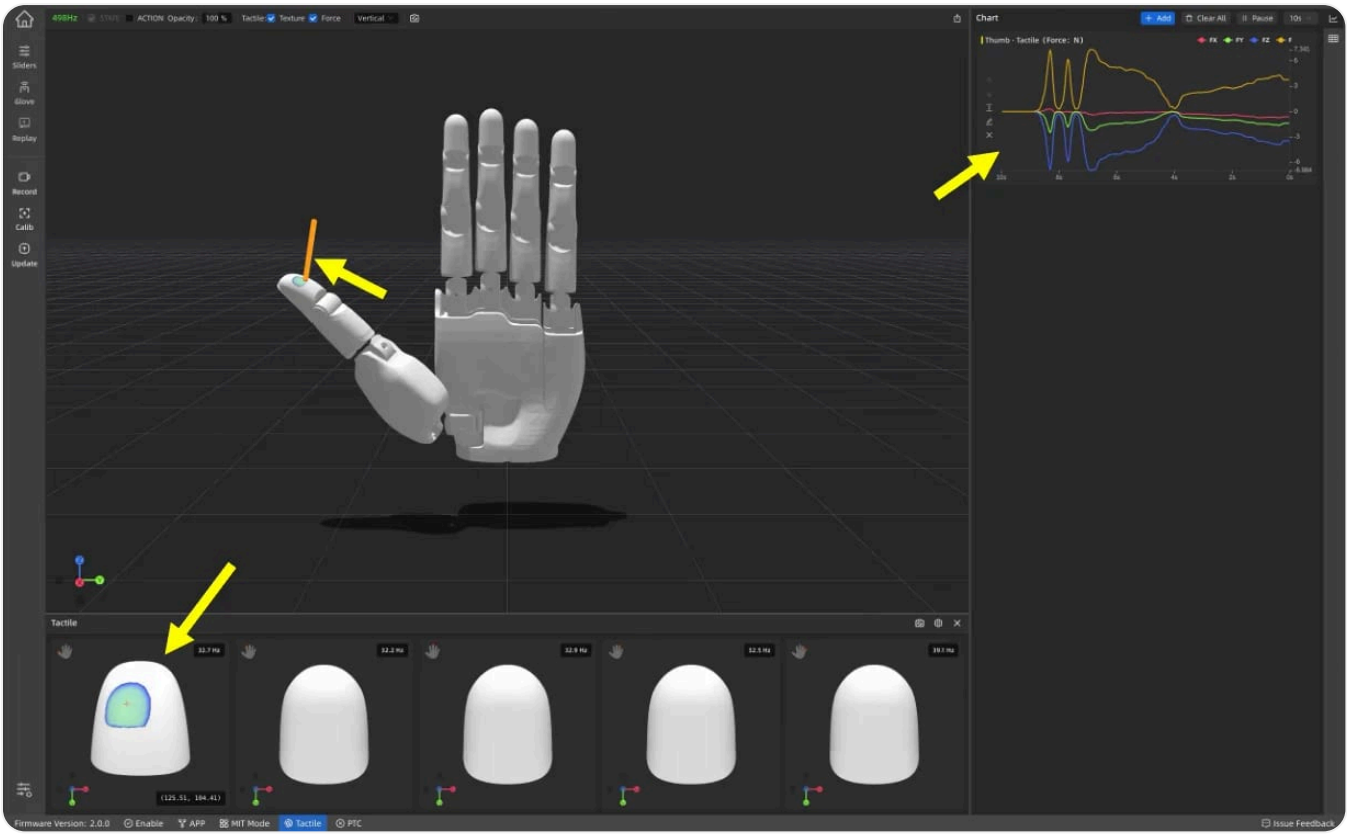
1. Open Sharpa Pilot
2. Enable “Texture” and “Force” in the top toolbar



3. Enable the real-time force components (Fx, Fy, Fz) and resultant magnitude plot



4. Apply physical contact to a fingertip
5. Observe changes in tactile display and force plot



4.3 Visualization in High-Performance Mode

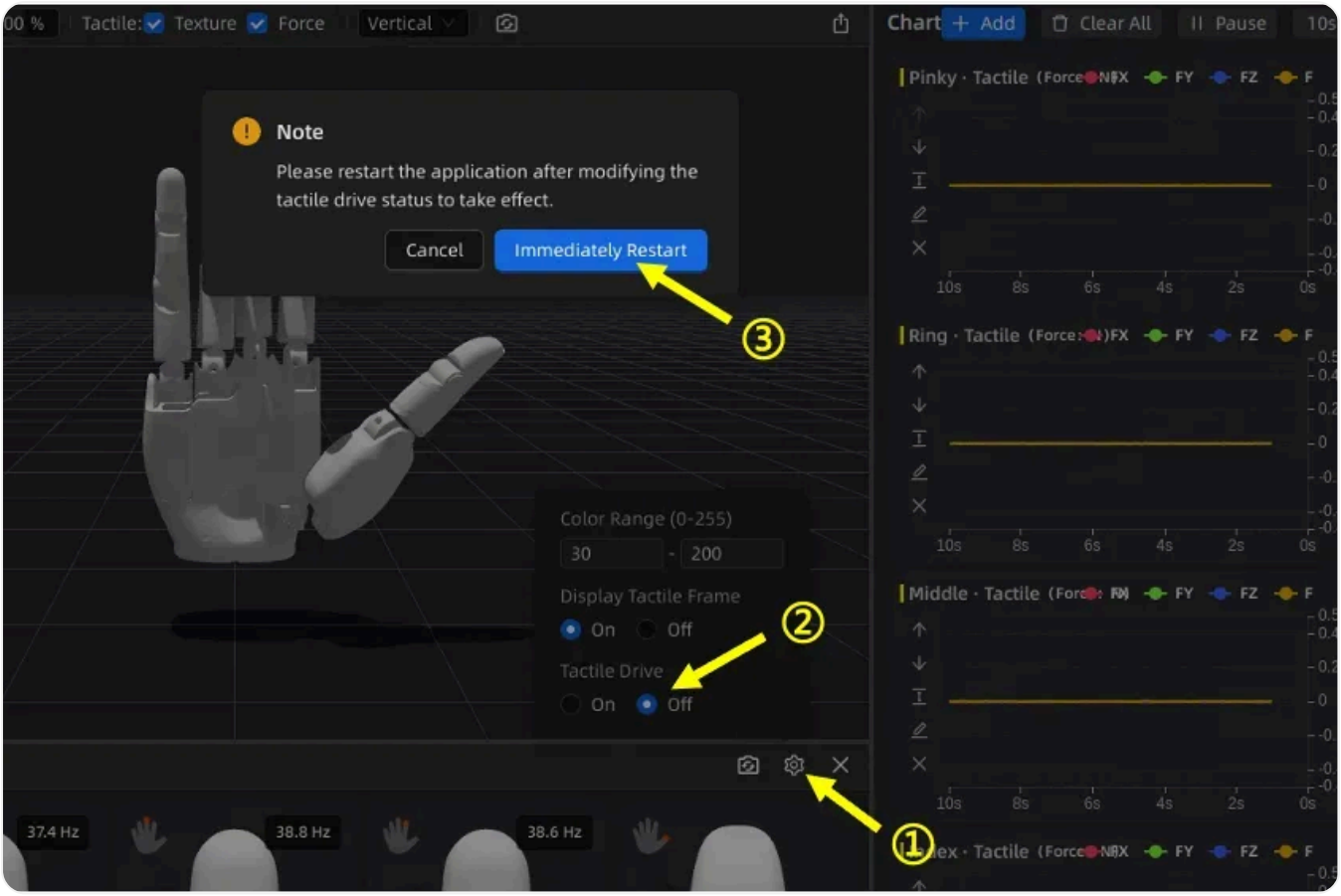
Sharpa Pilot cannot directly read data from the Docker pipeline, to visualize High-Performance host-side inference, a ZMQ publisher is used to forward tactile data to Sharpa Pilot.

1. Disable Onboard Tactile Driver

In Sharpa Pilot:

- Disable tactile driver
- Restart Pilot if needed

This ensures Pilot does not use the 30 Hz onboard pipeline.



2. Start ZMQ Publisher

Enter Docker container:

```bash
docker exec -it sharpa_dev bash
```

Run:

```bash
python3 ./docker_tactile_zmq_pub/tactile_zmq.py
```

This script publishes tactile data via ZMQ to localhost.

3. Verify Visualization

Expected result:

- Tactile visualization updates in real time (~180 Hz)
- Force and texture views respond faster than 30 Hz mode

4. If No Data Appears

Check in order:

- ZMQ script is running
- No firewall blocks localhost ports
- Restart Sharpa Pilot
- Restart Docker container

Notes

- Do not run `tactile_180fps.py` and Sharpa Pilot simultaneously, as they may use conflicting tactile pipelines.
- Ensure that only one tactile pipeline is active at a time.

5. Data Interpretation and Advanced Usage

5.1 Sensing Area

The sensing area is the region of the fingertip where tactile data is physically valid.

- It corresponds to the textured surface (ridges and grooves) of each fingertip
- All tactile outputs (force, deformation, raw image) are derived from this region

Only data within this area should be considered when interpreting contact (marked in dark color in the figure below).

 Index/Middle/Ring/Pinky fingers	 Thumb

5.2 Raw Images

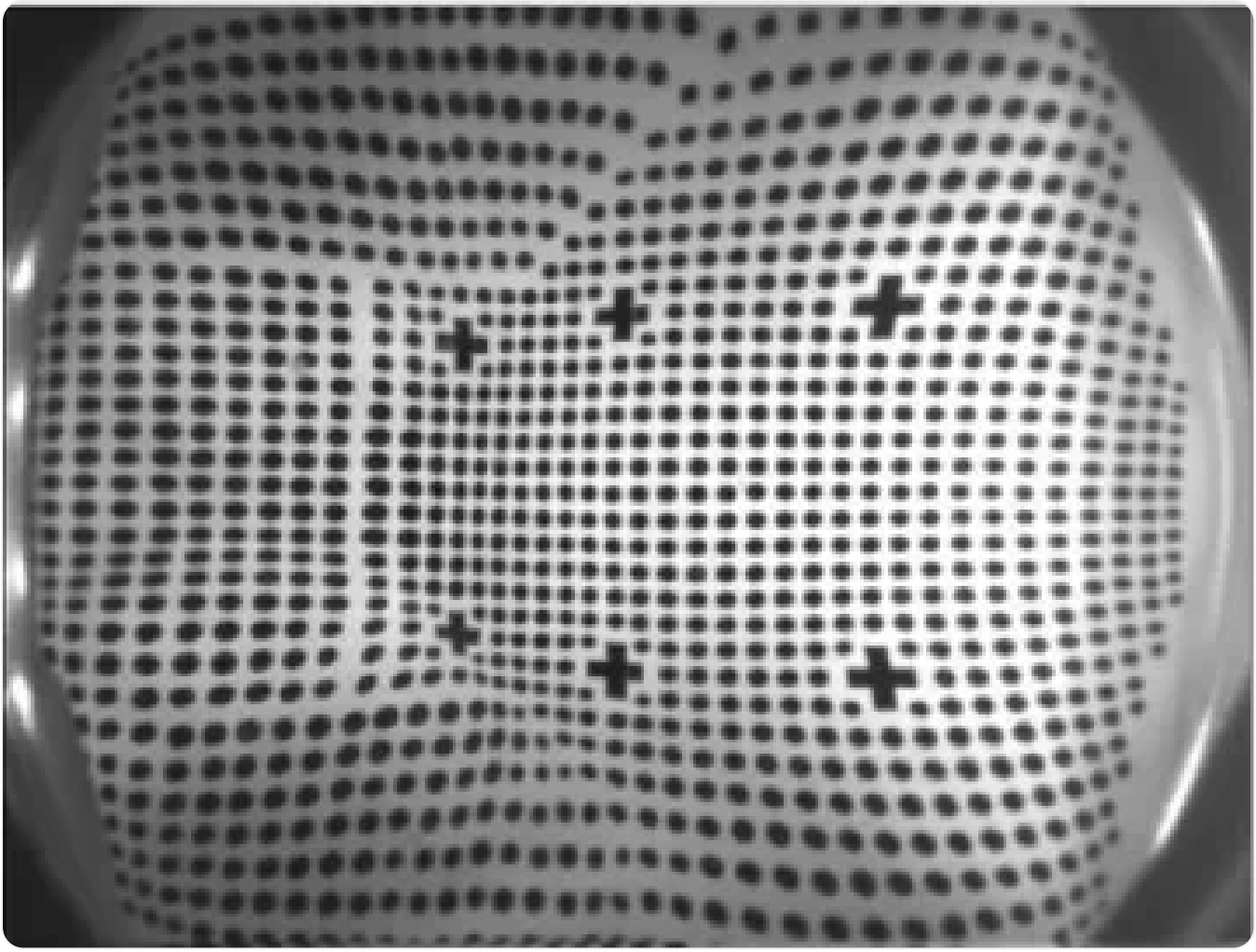
Definition

The raw image is the original camera capture inside the tactile sensor.

Interpretation

- Contains the most complete information about surface deformation
- Serves as the underlying signal from which higher-level representations (e.g., deformation map and force) are derived
- Includes visual patterns that may not be preserved in processed outputs

This representation is primarily useful when low-level sensor behavior or custom processing is required.



Raw image from the fingertip tactile sensor

5.3 Deformation Map

Definition

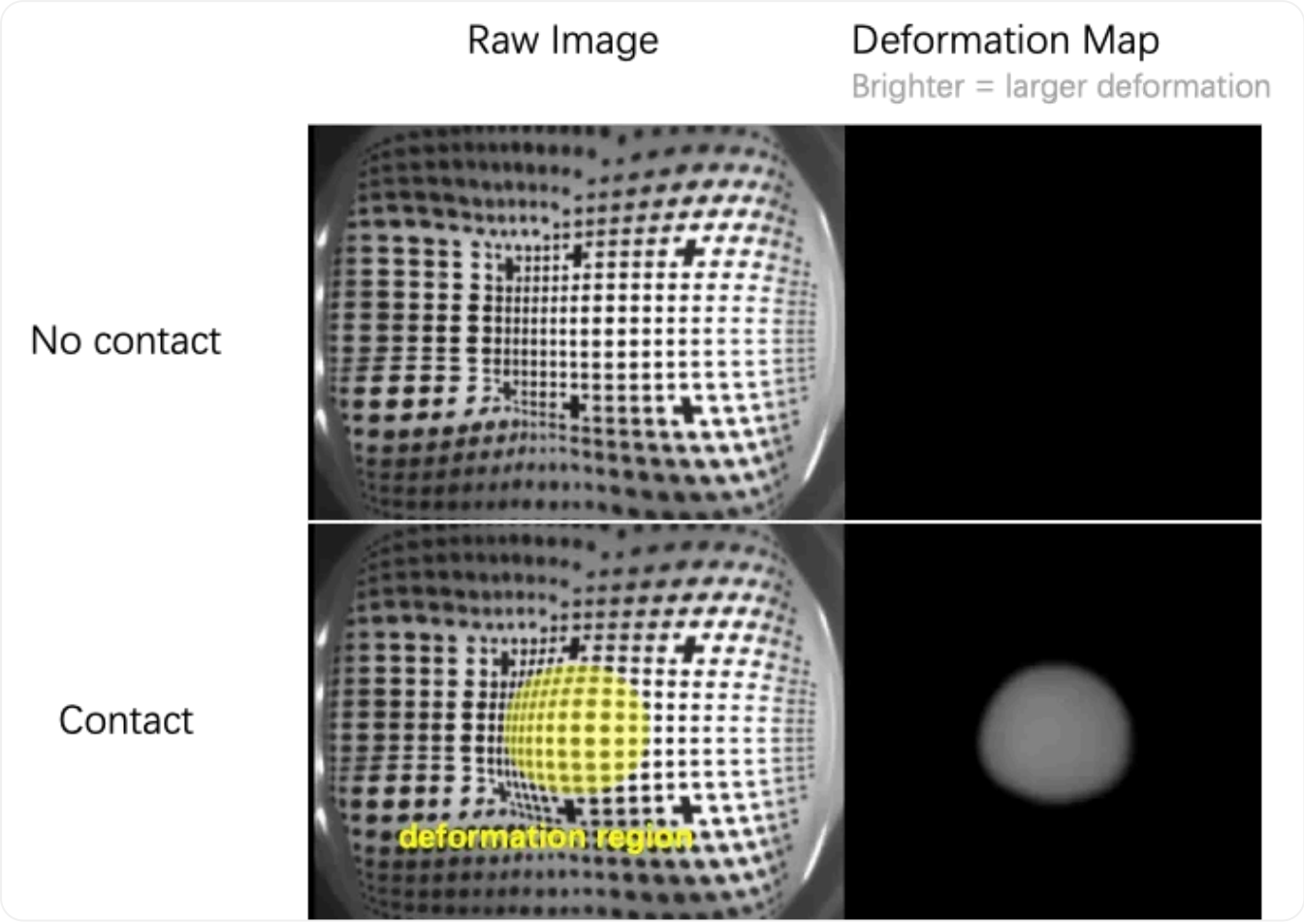
Deformation depth is defined as the displacement of each surface point along its normal direction relative to the no-contact state.

Representation

- The deformation map is a 2D grayscale image (240 × 240)
- Each pixel corresponds to a 3D point on the fingertip surface
- Pixel intensity represents deformation depth: brighter regions indicate larger deformation

The 2D representation is obtained by flattening the curved fingertip surface using an angle-preserving parameterization method (e.g., Angle-Based Flattening, ABF). This approach

preserves local geometric structure while minimizing distortion introduced during the 3D-to-2D mapping.



Interpretation

- Captures the **spatial distribution of contact**
- Reflects:
  - contact area
  - pressure distribution
  - local shape interaction

Mapping Between Grayscale Value and Deformation Depth

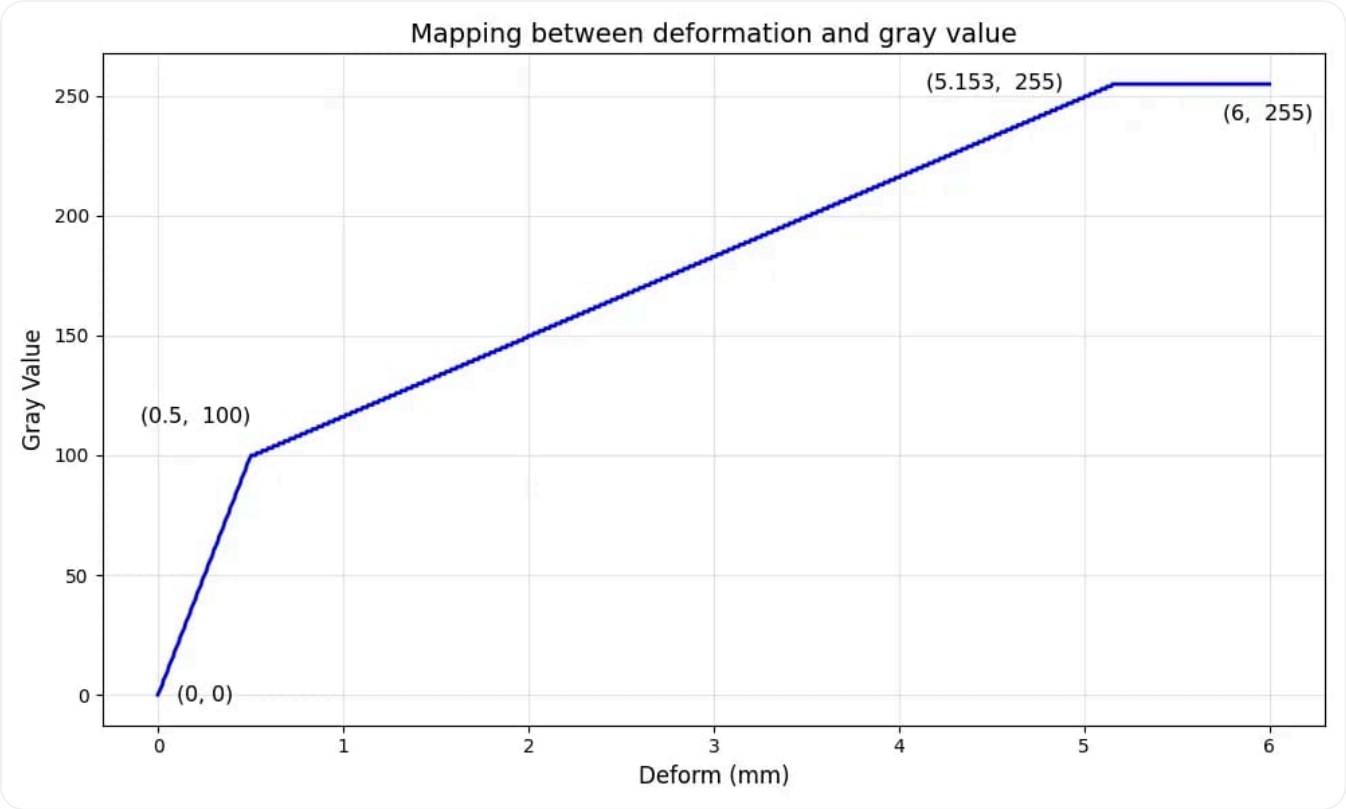
To balance high sensitivity and a wide measurement range, the deformation depth map is encoded as a grayscale image.

Each pixel value corresponds to the deformation depth of the associated surface point.

- Lower grayscale values represent small or no deformation

- Higher grayscale values represent larger deformation

The mapping between grayscale value and deformation depth is nonlinear, as illustrated in the figure below.



This design provides:

- high sensitivity in small deformation ranges
- extended dynamic range for larger deformation

For precise numerical conversion, refer to the Sharpa Wave SDK.

5.4 Contact Point

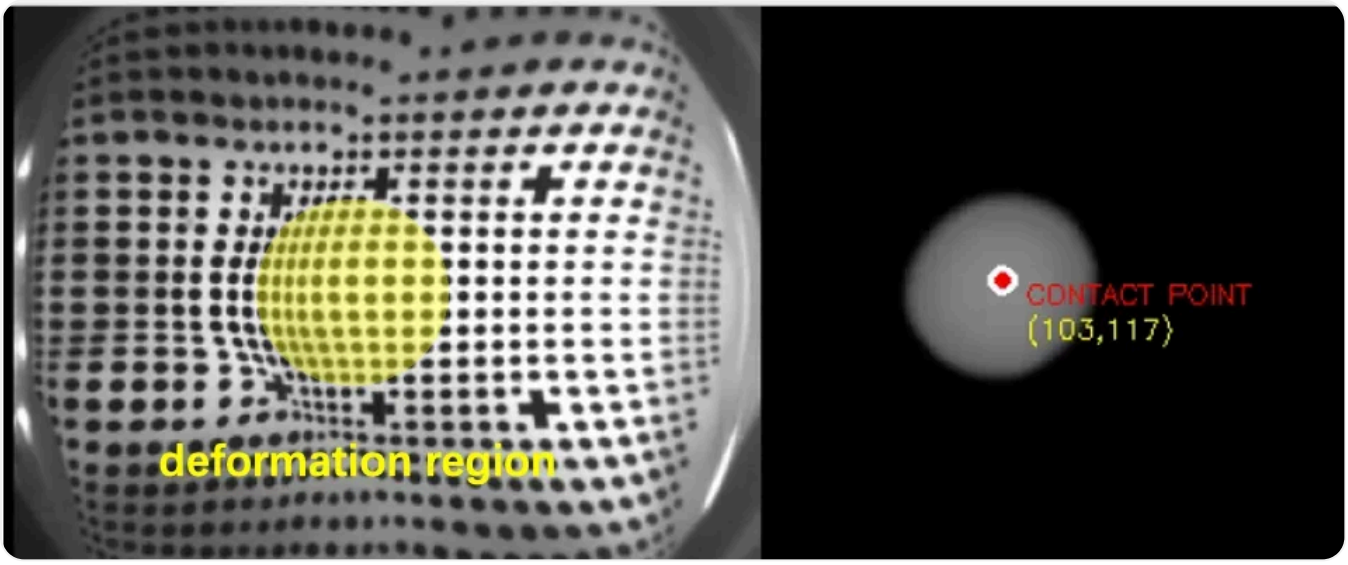
Definition

The contact point represents a single representative location of contact between the fingertip sensor and the object.

It is typically computed from the deformation map as the weighted centroid of the contact region.

Representation

- The contact point is represented as the 2D coordinates [column, row] in the deformation map
- Multiple contact points can be detected when there are multiple separated contact regions



Interpretation

- The contact point provides a compact representation of the contact location

It is typically located near the center of the deformation region, making it useful for tracking and control applications.

5.5 6-DOF Force

Definition

Each fingertip outputs a 6-DOF wrench:

(Fx, Fy, Fz, Tx, Ty, Tz)

representing the resultant external force and torque acting on the entire sensing surface.

- Force unit: Newton (N)
- Torque unit: Newton-meter (Nm)

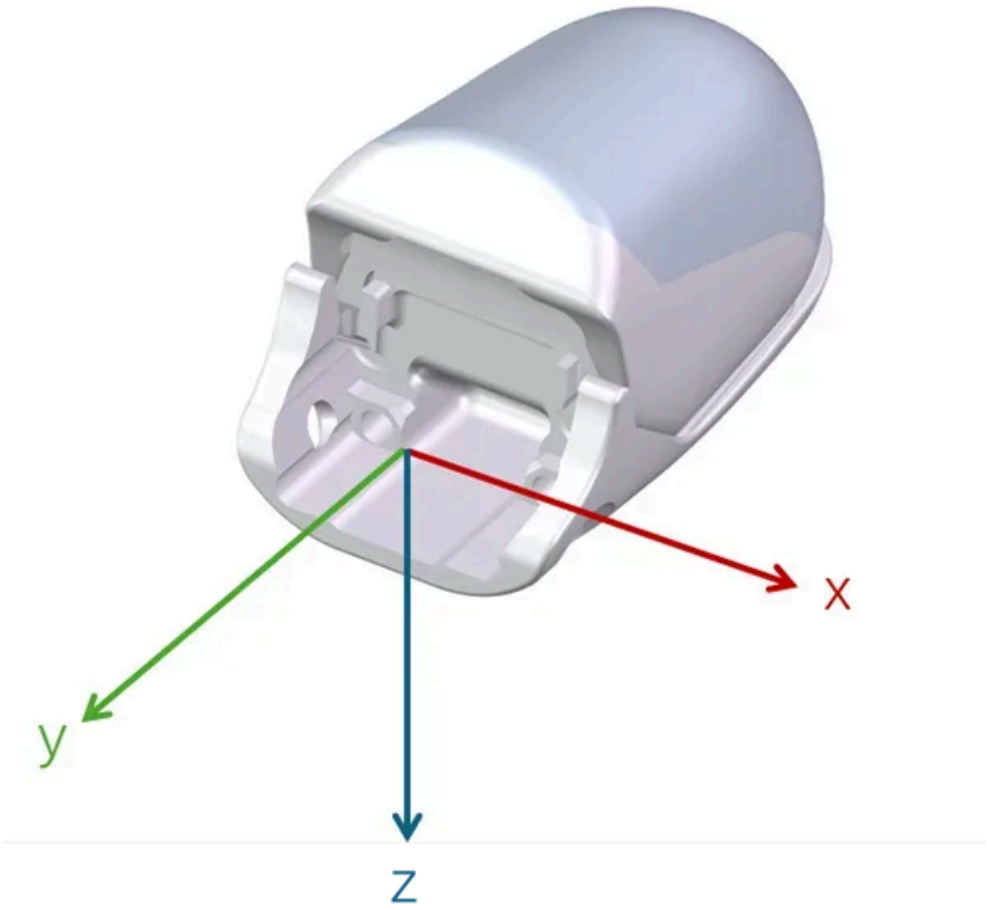
Coordinate System

The force is defined in a right-handed coordinate system.

The origin is located at:

- center of the IP joint's rotation axis (thumb)
- center of the DIP joint's rotation axis (index, middle, ring, and pinky fingers)

This definition is consistent with the SDK outputs and the URDF model.



Interpretation

- The output represents the **global effect of contact**, not local distribution
- Multiple contact points are aggregated into a single resultant wrench
- Torque components reflect the moment induced by off-center contact

This representation is suitable for understanding overall interaction but does not capture spatial details of contact.

5.6 2D–3D Surface Mapping

Definition

Each pixel in the deformation map corresponds to a 3D point on the fingertip surface.

This establishes a mapping between:

- 2D image space (pixel coordinates)
- 3D physical space (surface geometry)

Interpretation

- Enables reconstruction of contact in 3D space
- Allows deformation data to be interpreted in physically meaningful coordinates
- Supports integration with geometric models and simulation environments

Notes

Most applications do not require explicit use of this mapping.

It becomes relevant when precise geometric reasoning or reconstruction is needed.

Access

The 2D–3D mapping can be obtained in two ways:

- **Precomputed sensor model assets**: Check the `Assets/tactile_sensor_digital_model` folder in the downloading resource.
  - NPY files:

```cpp
NPY files
// For the four fingers: Normal vector (nx, ny, nz) of each 3D point,
pointing outward from the fingertip
tactileSensor_map_4F_normal.npy

// For the four fingers: Coordinates (x, y, z) of each 3D point, in units
of mm
```

```
tactileSensor_map_4F_point.npy

// For the thumb: Normal vector (nx, ny, nz) of each 3D point, pointing
outward from the fingertip
tactileSensor_map_TH_normal.npy

// For the thumb: Coordinates (x, y, z) of each 3D point, in units of mm
tactileSensor_map_TH_point.npy
```

- Since the 2D image contains $H \times W$ pixels, each of the above files contains a matrix of size $H \times W \times 3$.

- OBJ files (used for visualizing the sensor surface shape):

CPP

```
OBJ files
// for the four fingers
tactileSensor_4F.obj

// for thumb
tactileSensor_TH.obj
```

- Line starting with v: 3D point coordinates (x, y, z)
- Line starting with vt: Normalized 2D point coordinates (row, column)
- Line starting with vn: Normal vector (nx, ny, nz) of a 3D point

- **SDK interfaces**: Use the SDK to obtain the 3D point's normal vector (nx, ny, nz), which corresponds to the 2D point coordinates (row, column).
  - See the "Tactile Data" section in Sharpa Wave SDK.

Integration

1. Overview

Sharpa Wave is designed for flexible integration across a wide range of robotic platforms, from rapid prototyping setups to fully integrated systems.

It supports multiple mechanical and electrical interface combinations, enabling:

- **Fast integration** using standardized interfaces and external cabling
- **Compact integration** with internal routing and minimal wrist length

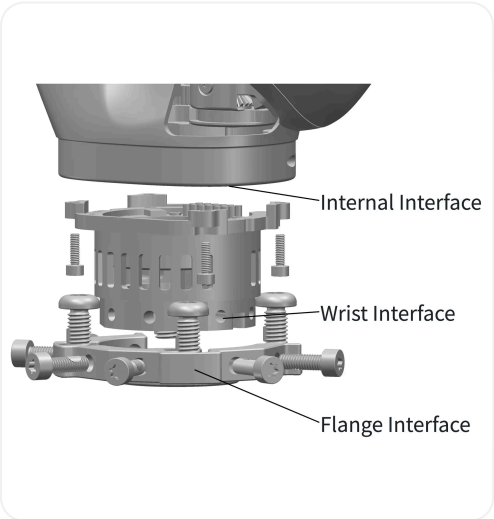
2. Interfaces

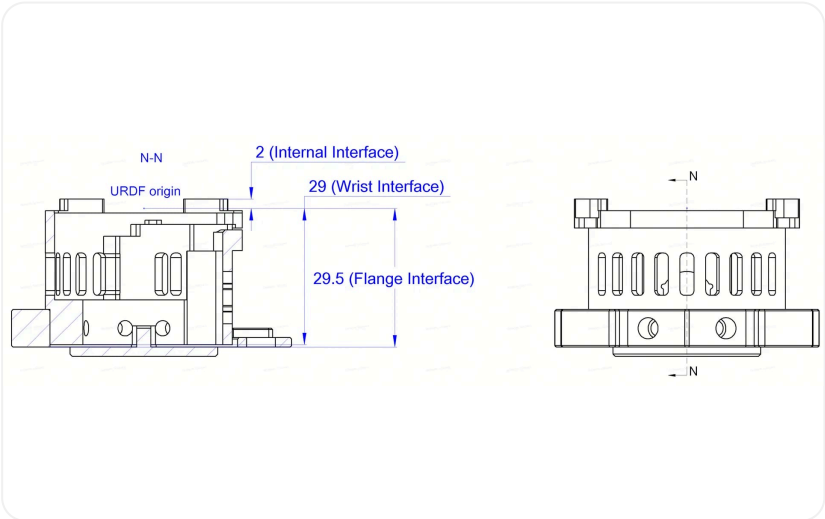
2.1 Available Interfaces

Mechanical interfaces

- **Flange**: Adapts to standard robot interfaces via a flange adapter
- **Wrist**: Connects directly to the robot wrist using screws; requires a custom adapter
- **Internal**: Integrates via an internal pogo pin interface, replacing the wrist module

The following diagram illustrates the relative positions of the three interface options:





Electrical interfaces

- **Wrist connector:** External aviation connector for power and communication
- **Internal pin pad:** Pogo pin interface for internal power and communication
  - Requires a **custom 3 × 6 pogo pin array** for mating (not a standard connector)

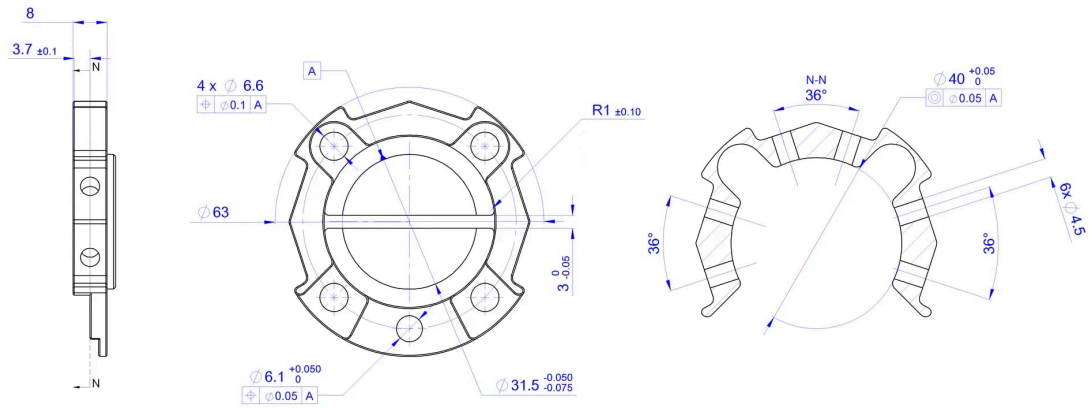
2.2 Interface Selection Guide

INTERFACE	CONNECTION TYPE	WHEN TO USE	KEY ADVANTAGES	TRADE-OFFS
Flange	Wrist connector	• Standard robot flange interfaces (e.g., UR, xArm, Franka, Flexiv) • fast and reliable integration	• Standardized compatibility • simple and robust mounting	• External cabling • longer wrist length
Wrist	Wrist connector	• Custom robot interfaces • rapid prototyping with a custom adapter	• Flexible for custom designs • easier prototyping	• Additional mechanical design effort • external cabling • longer wrist length
Internal	Internal pin pad	• Compact integrated designs • minimal wrist length; internal cable routing	• Shortest wrist length • clean integrated routing	• High alignment precision required • Requires custom pogo pin array for electrical connection • More complex mechanical and electrical integration

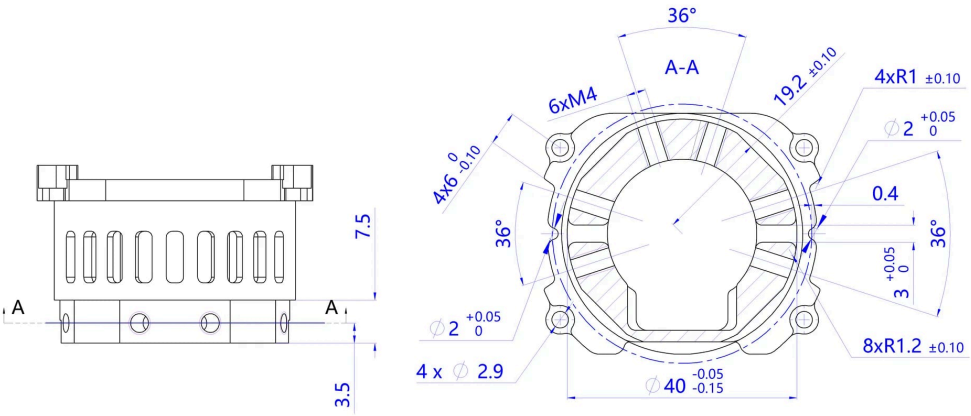
3. Mechanical Integration

3.1 Mounting Interfaces

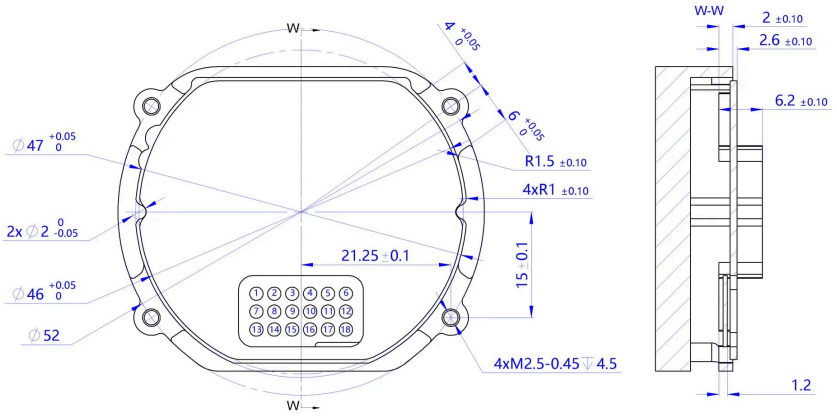
- Flange



- Wrist



- Internal



3.2 Load Requirements

- Hand weight: 1.3 kg

- Recommended robot payload ≥ 5 kg

4. Electrical Integration

4.1 Power Requirements

- Voltage: 18–28 V
- Maximum power consumption:
  - Single hand: 100 W average, 180 W peak
  - Dual hand: 200 W average, 360 W peak

4.2 Communication Interface

- Ethernet (1000BASE-T) is required for 180 Hz tactile data streaming.
- Ethernet (100BASE-TX) is sufficient for hand control and 30 Hz tactile data.
- Typical bandwidth usage (per hand):
  - 30 Hz (processed outputs): ~14 Mbps(For settings,refer to the section Tactile Raw Image Disabled)
  - 30 Hz (processed + raw image): ~80 Mbps(default)
  - 180 Hz (raw image): ~400 Mbps

4.3 Dual-Hand Configuration

Two network setups are supported for dual-hand configuration: using a network switch (recommended) or using multiple Ethernet interfaces.

1. Using a network switch (recommended):

- Connect the host and both devices via a switch.
- Use a single host IP (e.g., 192.168.10.240) as the tactile destination IP.
- Ensure all devices are on the same network segment.

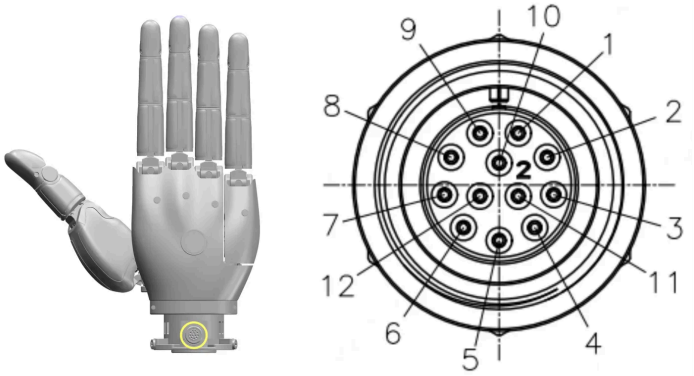
2. Using multiple Ethernet interfaces:

- Each host interface must have a unique IP address.
- Each hand and the corresponding host interface must be on the same subnet.

- Each hand must be configured with a tactile destination IP matching the corresponding host interface.

4.4 Wrist Connector

The connector is located at the wrist on the back of the hand. Pin arrangement is shown below:

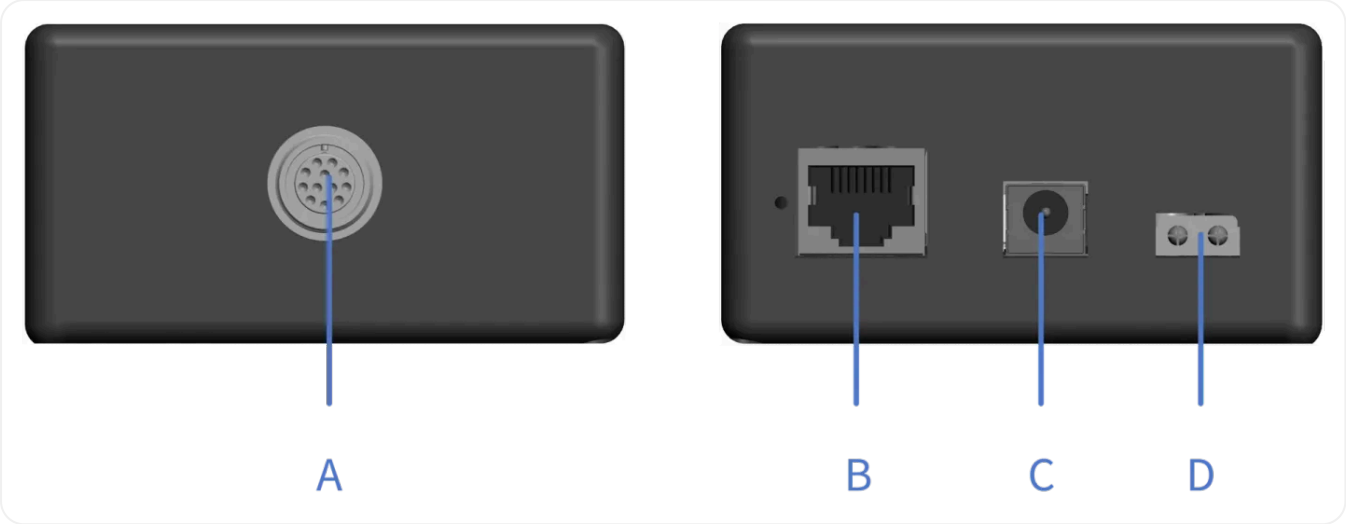


Pin No.	Signal name	Description	
1, 2	PWR	Power input (positive)	
8, 9	GND	Power input (negative)	
12	MDI0_N	Ethernet 1000BASE-T	Ethernet 100BASE-TX
10	MDI0_P		
6	MDI1_N		
7	MDI1_P		
4	MDI2_N		Not used in 100BASE-TX
5	MDI2_P		
3	MDI3_N		
11	MDI3_P		

4.5 Connection Box

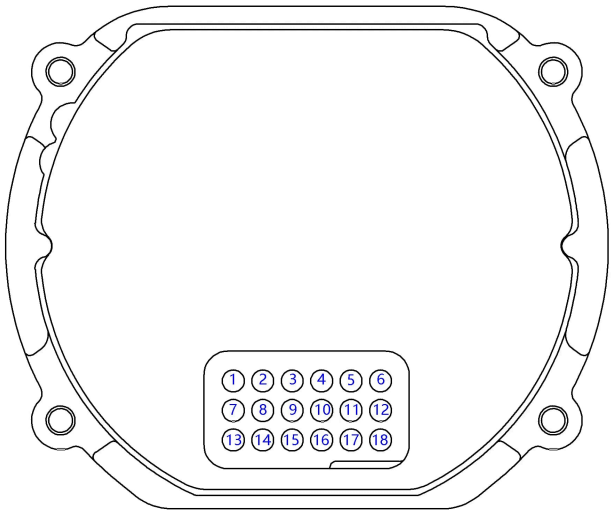
The connection box converts the custom wrist interface into standard power and Ethernet interfaces. It is typically used with the wrist connector option.

The ports of the connection box are as follows:



PORT	CONNECTOR	DESCRIPTION	NOTES
A	Customized connector	Connects to the hand via a custom wiring harness	Interface identical to the hand
B	RJ45	Ethernet interface (1000BASE-T)	Supports lockable Ethernet cables (anti-loosening design)
C	DC 5.5 × 2.5	External power input (compatible with provided power adapter)	Use either port C or D independently Do not connect both ports simultaneously
D	XT-30	External power input (compatible with XT30 power connector)	

4.6 Internal Pin Pad



Pin	Signal	Description	
1, 2, 3	PWR	Power Input (Positive)	
4, 5, 6	GND	Power Input (Negative)	
13	FAN_PWR	Fan power supply (6 V, 0.2 A max)	
7,8,14	RESERVED	Do not use	
15	MDI0_N	Ethernet 1000BASE-T	Ethernet 100BASE-TX
9	MDI0_P		
16	MDI1_N		
10	MDI1_P		
17	MDI2_N		Not used in 100BASE-TX
11	MDI2_P		
18	MDI3_N		
12	MDI3_P		

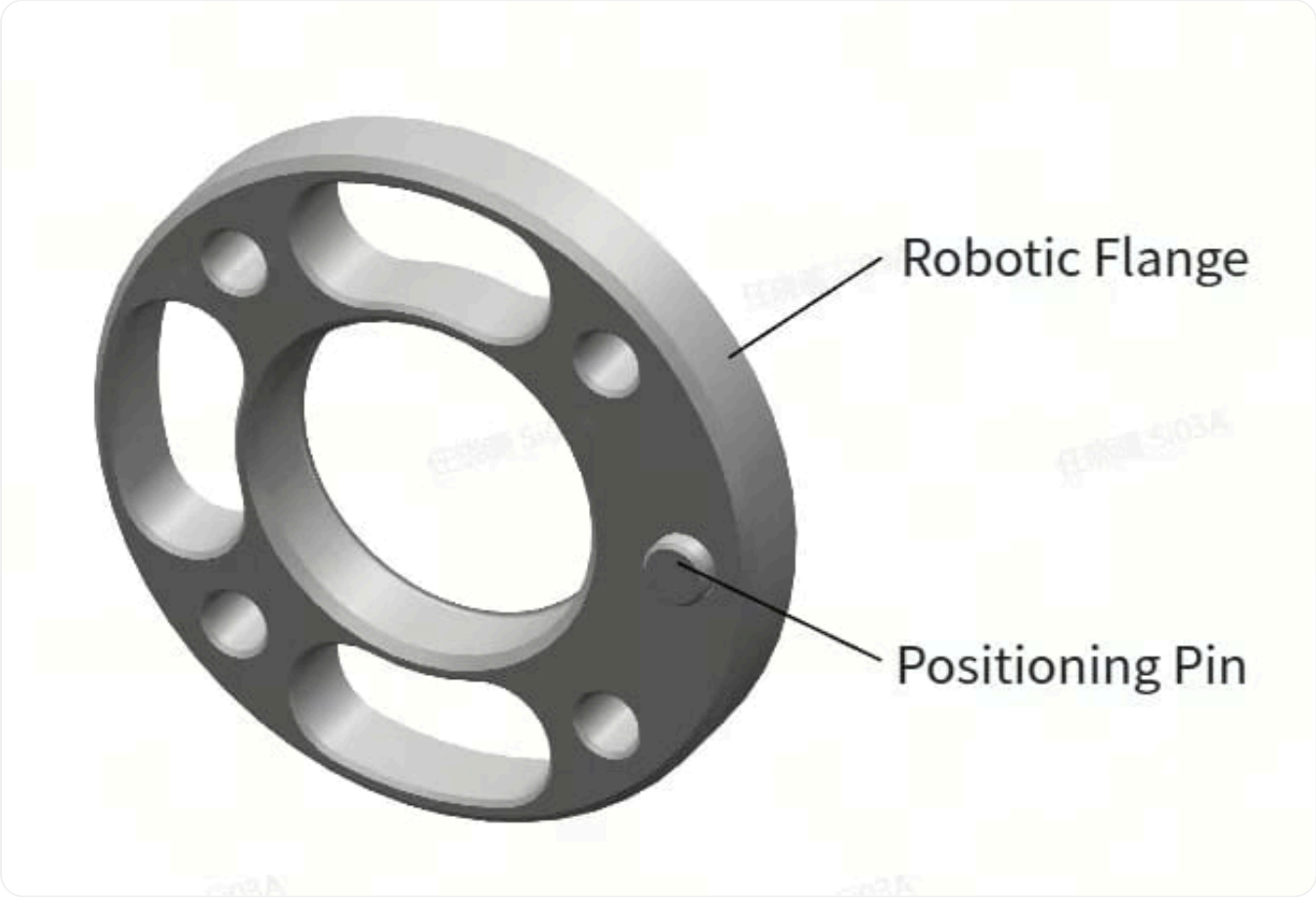
5. Integration Example

This section describes a typical integration setup using the flange interface. In this example, power and network connections are routed through the robot body, and the robot's onboard system is ARM-based and runs Ubuntu 22.04.

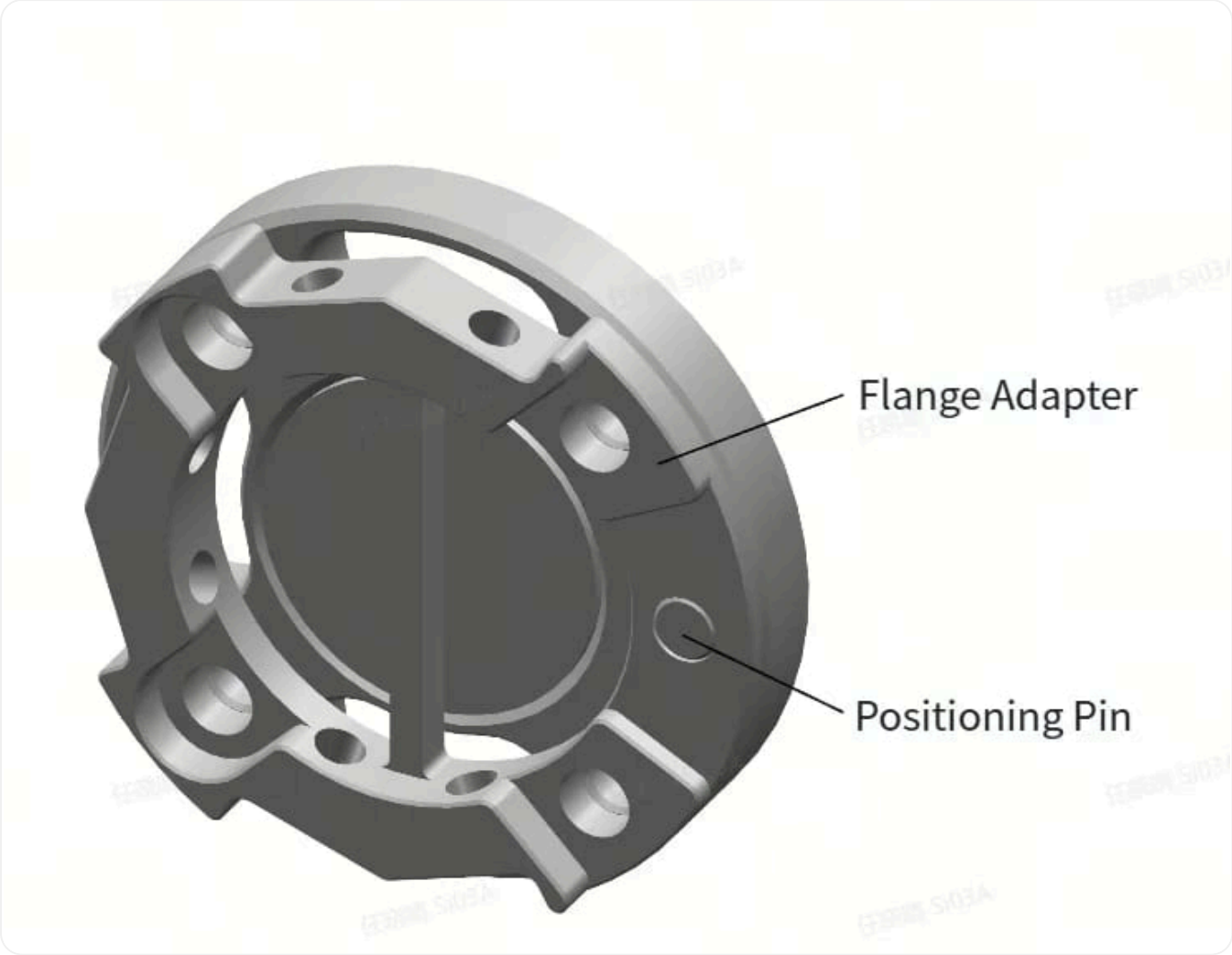
5.1 Install the Hand

This section describes how to mount the hand onto the robotic arm using the flange interface.

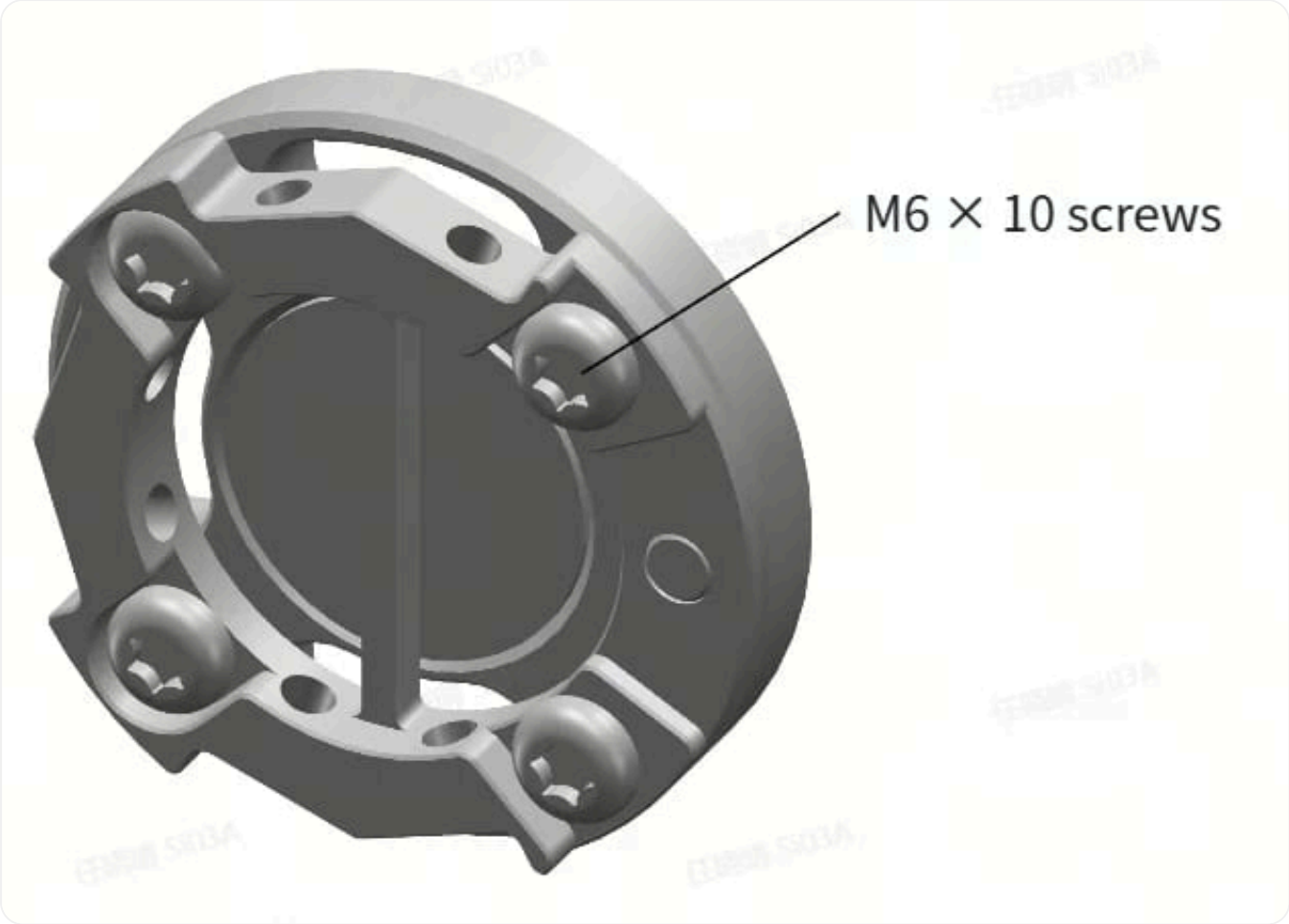
1. Insert the positioning pin (φ6 × 10) into the locating hole at the end of the robotic arm.



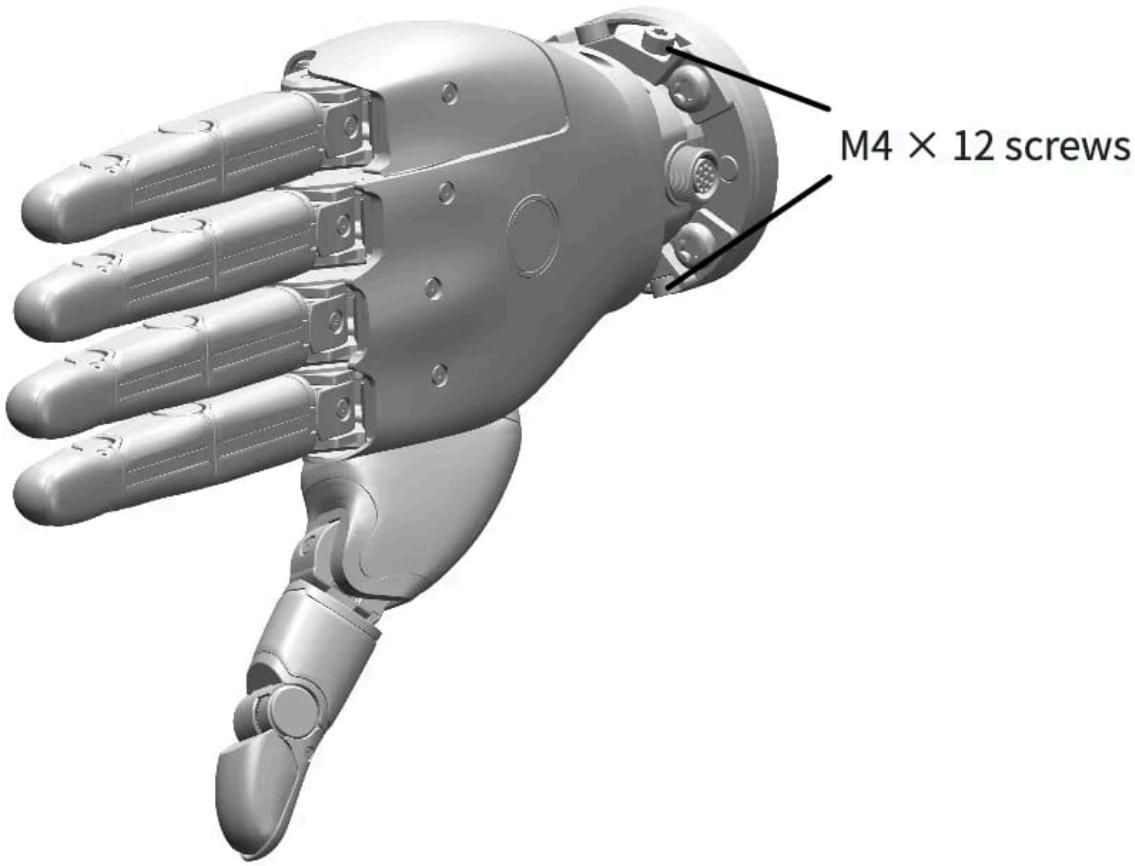
2. Align and mount the flange adapter onto the robot end, ensuring proper engagement with the positioning pin and a flush fit with the mounting surface.



3. Install the M6 × 10 screws and tighten them in a diagonal (cross) sequence.

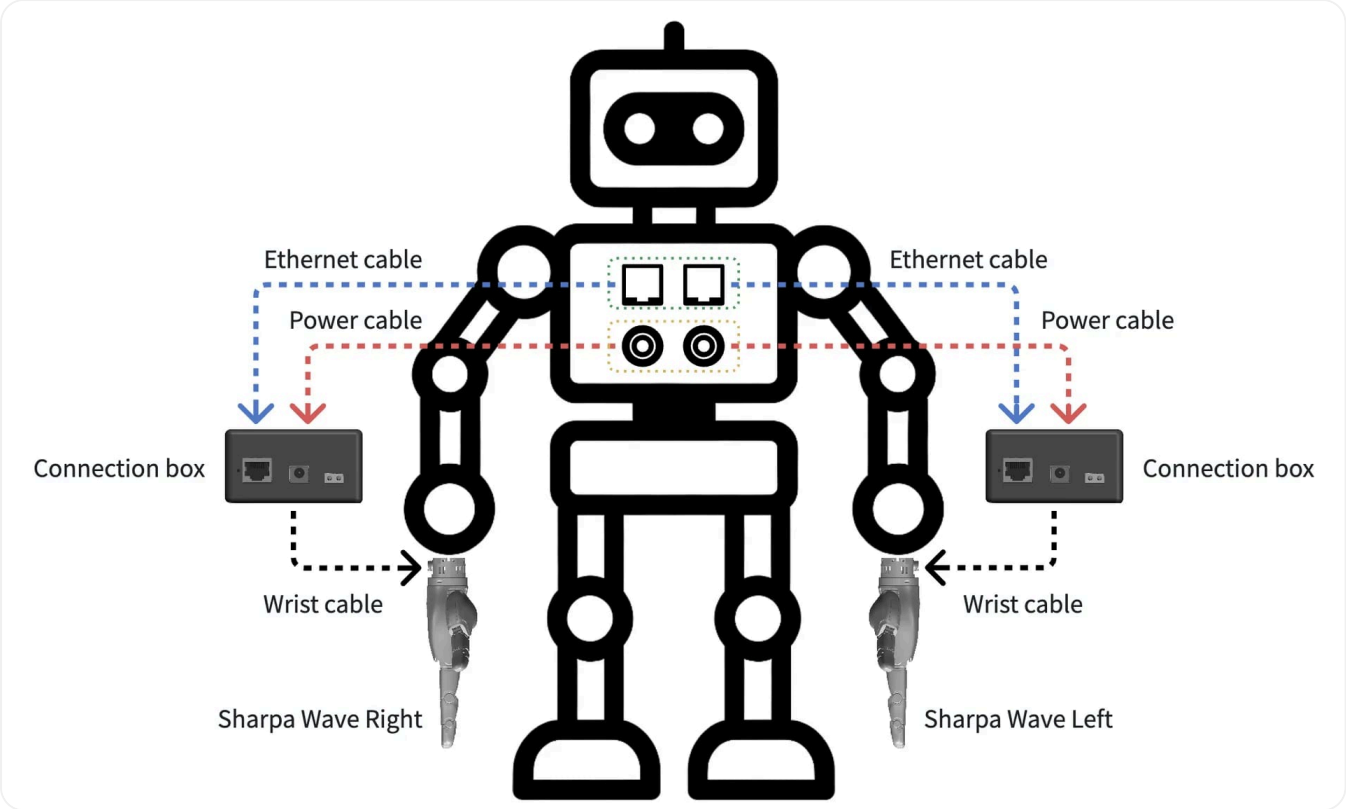


4. Mount the Sharpa Wave hand onto the flange adapter and secure it using the M4 × 12 screws



5.2 Connect Power and Network

- Connect the hand to the connection box.
- Connect the power input to the robot power interface (XT30 or DC 5.5 x 2.5), or use the provided power adapter by plugging it into an AC outlet.
- Connect the Ethernet interface:
  - If the host system has multiple Ethernet ports, connect each connection box directly to a port on the host system.
  - If the host system has only one Ethernet port, connect all devices via a network switch (host system ↔ switch ↔ connection boxes).



5.3 Check Cable Routing

- Ensure that the cables are secured along the robot structure.
- Avoid interference with robot motion.
- Avoid excessive bending or tension.

5.4 Power On and Connect

- Ensure Sharpa Wave and the host system (ARM computer) are on the same network segment
- Configure the host system IP address according to Section 5.1 (Default Network Settings), e.g., 192.168.10.240
- Download the ARM version of the Sharpa Wave SDK
- Run sharpa_wave_example following the instructions in README.md

Successful connection is indicated if Sharpa Wave moves.

6. Wrist Camera Integration

Integrating a camera at the wrist can significantly enhance perception for manipulation and teleoperation tasks.
This section provides design guidelines for mounting a camera using the **wrist interface.**

A depth camera such as the Intel RealSense D405 is used as a reference example, but the guidelines apply to similar devices.

6.1 Design Considerations

Before integration, review the camera datasheet and confirm the following:

- Mechanical mounting interface
- Cable type and routing requirements
- Field of view (FOV)

These parameters directly affect placement, visibility, and integration complexity.

6.2 Camera Placement Guidelines

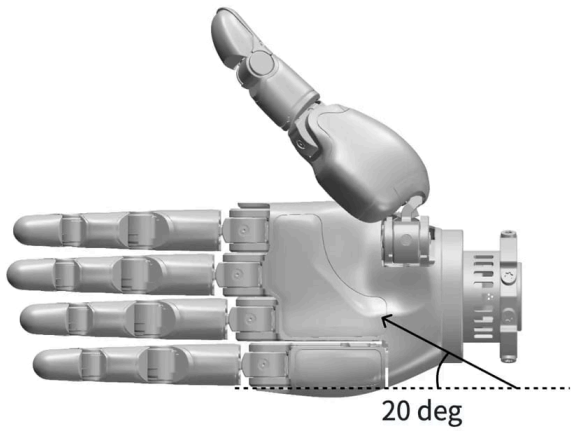
1. Viewing Direction

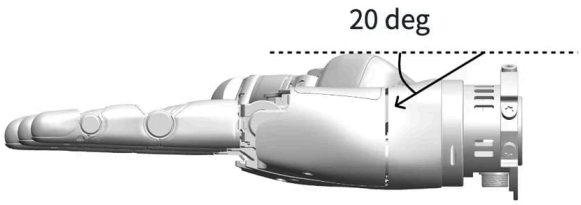
The camera should be oriented toward the workspace in front of the palm.

A slight downward tilt (approximately **20°**) is recommended to:

- improve visibility of contact areas
- reduce occlusion from fingers
- maintain a stable view during manipulation

The exact angle should be adjusted based on the application.



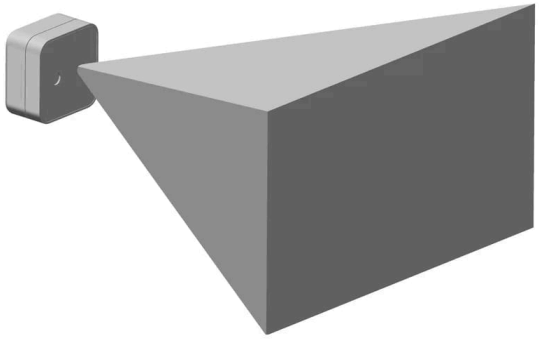


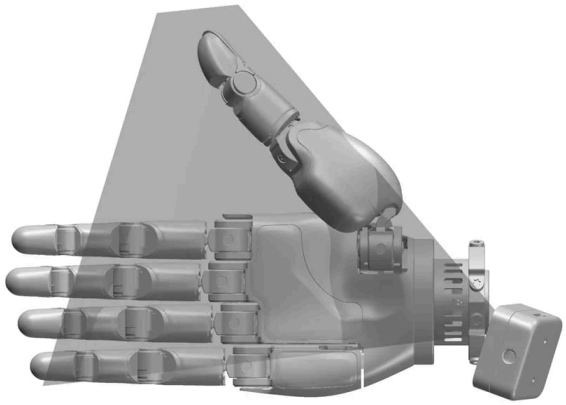
2. Field of View Coverage

The camera should be positioned such that its FOV covers the **palm and fingers.**

When designing:

- Use CAD or simple geometric models to approximate the FOV
- Verify that key interaction regions are within view
- Treat the FOV model as an approximation; validate with real hardware





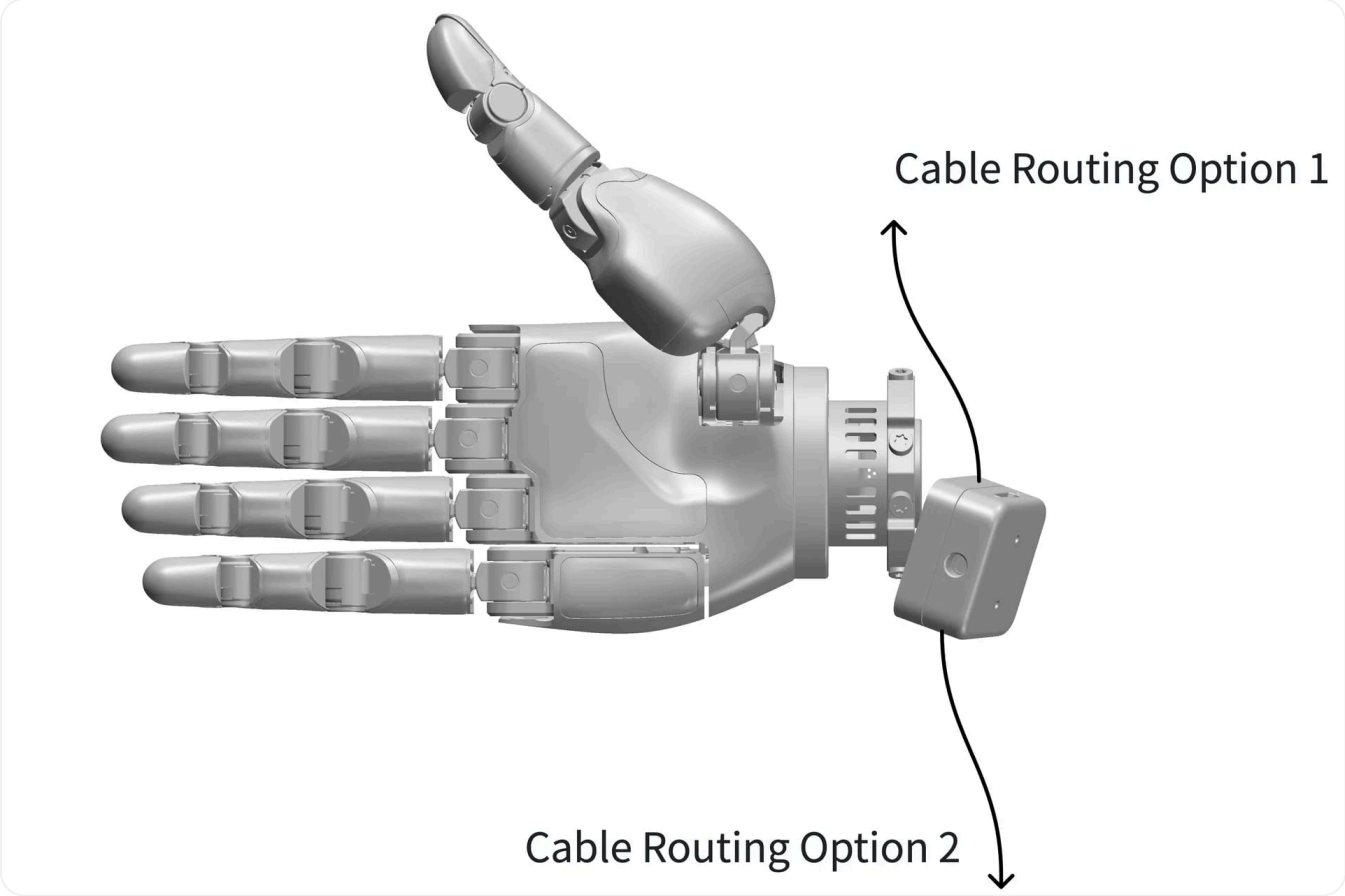
3. Cable Routing and Clearance

Cable routing must be considered early in the design:

- Avoid interference with finger motion and workspace
- Ensure sufficient clearance from surrounding structures
- Provide enough bending space for the cable

Typical routing paths include:

- along the side of the wrist
- along the underside of the wrist



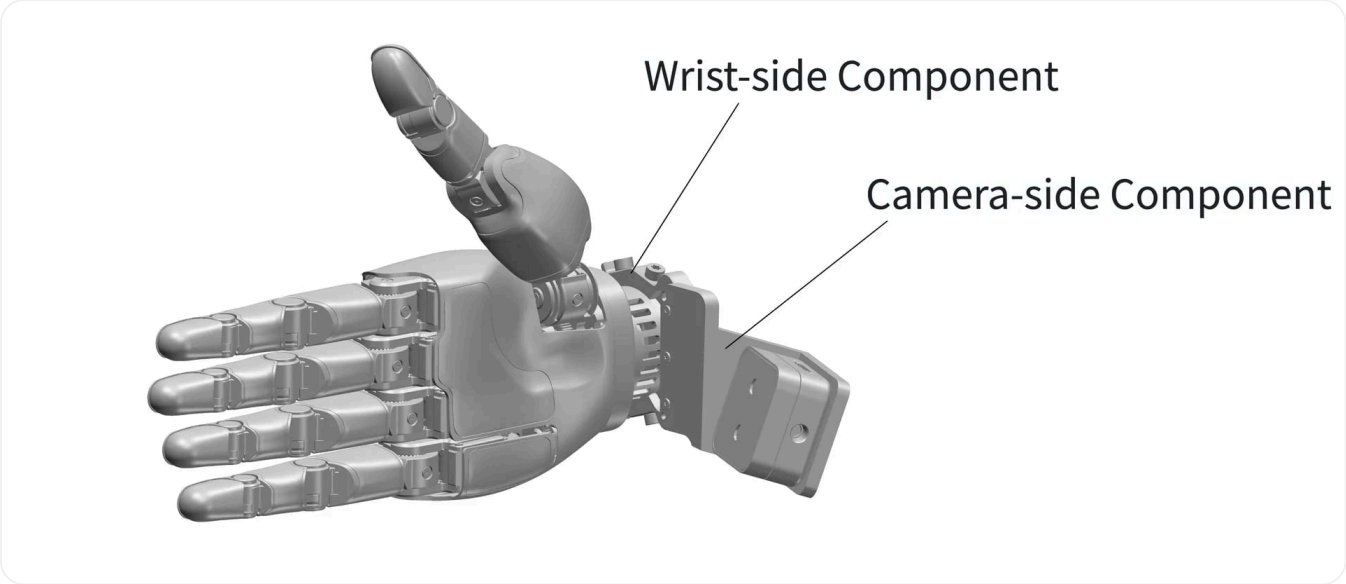
6.3 Mounting Design

To integrate the camera with the wrist interface, a custom adapter is typically required.

1. Adapter Structure

For manufacturability and flexibility, the adapter is commonly divided into two parts:

- a **wrist-side component** that attaches to the wrist interface
- a **camera-side component** that mounts the camera

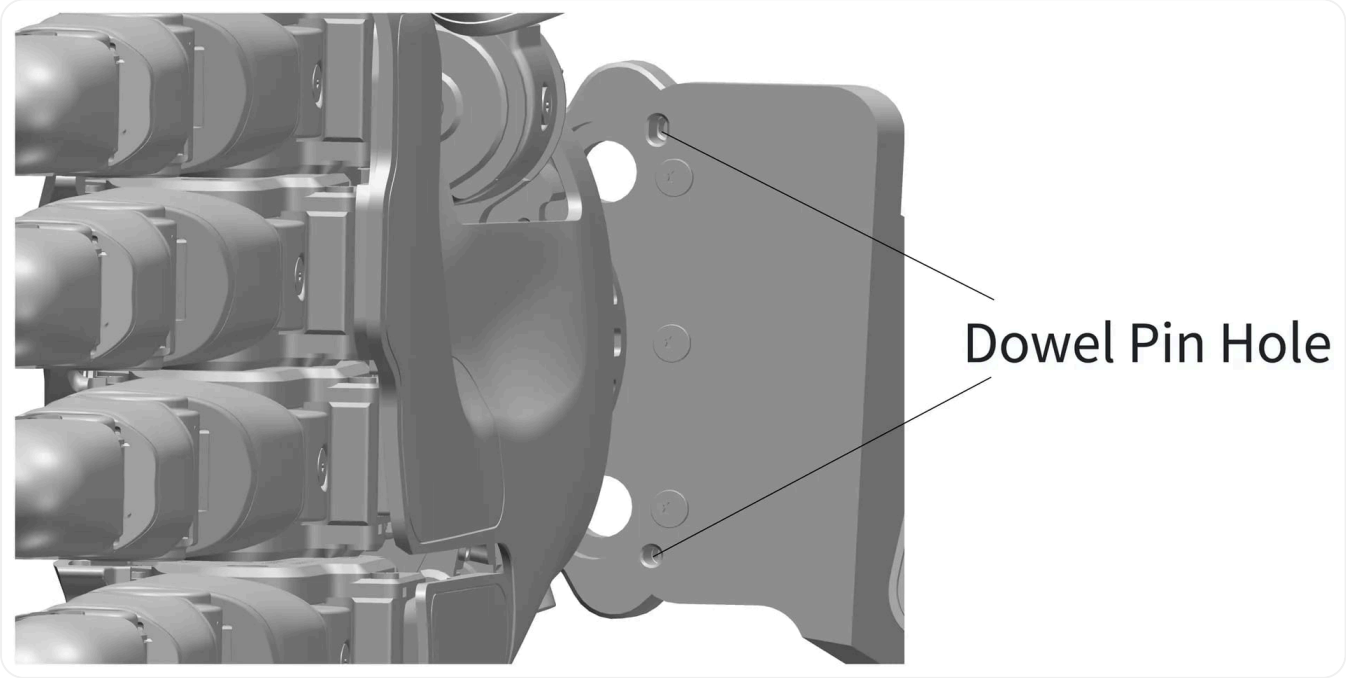


2. Alignment and Repeatability

To ensure consistent camera pose after assembly:

- include precision alignment features (e.g., dowel pins)
- avoid relying solely on screw positioning

This is especially important if the adapter is frequently removed and reinstalled.



3. Prototyping and Production

- Use **3D printing** for rapid prototyping and design validation

- After validation, use **CNC machining** for final parts to ensure strength and precision

6.4 Summary

A successful wrist camera integration should:

- provide stable and unobstructed view of the workspace
- avoid interference with hand motion
- ensure reliable cable routing
- maintain repeatable alignment after assembly

Sharpa Pilot

Coming soon

Development

SDK Overview

The Sharpa Wave SDK provides a unified programming interface for controlling the dexterous hand, accessing real-time state feedback, and integrating tactile sensing into robotic applications.

It is designed to support both rapid prototyping and production-grade system integration.

1. Supported Languages

The Sharpa Wave SDK is available in both Python and C++, with consistent core abstractions and APIs across languages.

- **Python SDK** is recommended for:
  - rapid development and prototyping
  - algorithm development and research workflows
  - data processing and machine learning integration
- **C++ SDK** is recommended for:

- performance-critical applications
- real-time control systems
- embedded or production deployment

Both SDKs expose the same core capabilities (device management, motion control, state feedback, and tactile sensing), allowing users to switch between languages with minimal changes.

A typical development workflow is to prototype in Python and migrate to C++ when higher performance or tighter system integration is required.

2. Core Capabilities

The SDK provides full access to device functionality:

- **Device Management**
  - Discover available devices on the network
  - Connect / disconnect devices
  - Query device metadata and status
- **Motion Control**
  - Joint-level position control
  - Admittance control (force → motion)
  - MIT impedance control
  - Runtime parameter tuning (stiffness, damping, etc.)
- **State Feedback**
  - Joint angles, velocities, torques (500 Hz)
  - Device status and fault information
- **Tactile Sensing**
  - Access multi-modal tactile data:
    - raw image
    - deformation map
    - 6-DOF force
    - contact point
  - Support both polling and callback-based data pipelines

- **System Configuration**
  - Control source / control mode switching
  - Network and device configuration
  - Runtime parameter adjustment

3. Architecture Overview

The SDK follows a **manager–device architecture:**

- **SharpaWaveManager**
  - Singleton entry point
  - Handles device discovery and connection lifecycle
- **SharpaWave**
  - Represents a connected hand device
  - Provides all runtime control and sensing interfaces

This abstraction is consistent across Python and C++.

4. Typical Workflow

A typical SDK usage flow is:

1. Get the global manager instance
2. Discover available devices
3. Connect to a device
4. Wait until the device is ready
5. Configure control source and control mode
6. Send commands or read data
7. Disconnect cleanly

If tactile sensing is used, the workflow may additionally include:

- waiting for tactile readiness
- performing tactile calibration
- fetching tactile frames (polling)

- processing tactile data via callbacks

5. Relationship with Sharpa Pilot

Sharpa Pilot and SDK share the same underlying communication interfaces:

- Only **one control source** can actively send motion commands
- Tactile streaming ports are shared

Important:

- Do not run SDK tactile programs and Sharpa Pilot simultaneously
- Ensure control source is switched to SDK before sending commands

Setup

1. System Requirements

COMPONENT	REQUIREMENT
CPU	x86_64 / arm64
OS	Ubuntu 22.04 / 24.04
Python SDK	Python 3.10–3.13
C++ SDK	C++17 or later

2. Installation

2.1 Download

Download the Sharpa Wave SDK from: Sharpa Wave SDK

Select the package that matches your CPU architecture (e.g., amd64 or arm64).

2.2 Install the SDK Package

Install the Debian package:

```text
sudo dpkg -i <sharpa-wave-sdk-package>.deb
```

This installs:

- runtime libraries
- C++ headers
- Python bindings
- configuration files and examples

2.3 Package Layout

A typical SDK installation provides the following directory layout:

```text
/opt/sharpa-wave-sdk/include/SharpaWaveSDK.h
/opt/sharpa-wave-sdk/lib/libsharpa-wave-sdk.so
/opt/sharpa-wave-sdk/python/sharpa/__init__.py
/opt/sharpa-wave-sdk/python/sharpa/sharpa.cpython-310-x86_64-linux-gnu.so
/opt/sharpa-wave-sdk/python/sharpa/sharpa.cpython-311-x86_64-linux-gnu.so
/opt/sharpa-wave-sdk/python/sharpa/sharpa.cpython-312-x86_64-linux-gnu.so
/opt/sharpa-wave-sdk/python/sharpa/sharpa.cpython-313-x86_64-linux-gnu.so
/opt/sharpa-wave-sdk/README.md
/opt/sharpa-wave-sdk/config/...
```

The most commonly used paths are:

- the public C++ header in `/opt/sharpa-wave-sdk/SharpaWaveSDK.h`
- the runtime library in `/opt/sharpa-wave-sdk/`
- the Python package in `/opt/sharpa-wave-sdk/python/`

- configuration files and examples in `/opt/sharpa-wave-sdk/`

2.4 Language-Specific Setup

- Python SDK
  - The Python module is installed automatically by the `sharpa-wave-sdk` Debian package.
  - The module is located at `/opt/sharpa-wave-sdk/python/`.
  - Install optional dependencies for Python examples:

```
TEXT

pip install -r python/requirements.txt
```

  - If you only run tactile visualization examples and prefer system packages, you may install:

```
TEXT

sudo apt install python3-numpy python3-opencv
```

- C++ SDK
  - The public C++ header and runtime library are installed automatically by the sharpa-wave-sdk Debian package.
  - To compile C++ examples, install a C++ build toolchain:

```
CPP

sudo apt install build-essential pkg-config
```

  - To compile tactile C++ examples, install OpenCV development headers:

```
CPP
```

```
sudo apt install libopencv-dev
```

- Link applications with:

```text
-I/opt/sharpa-wave-sdk \
-L/opt/sharpa-wave-sdk \
-lsharpa-wave-sdk -ldl \
-Wl,-rpath,/opt/sharpa-wave-sdk/
```

Python SDK Guide

1. Quick Start

This section provides the shortest recommended path to a working integration.

1.1 Before You Start

Before running the example:

- power on the Sharpa Wave
- connect the host and the device to the same subnet
- confirm that the SDK Python module is installed correctly

For physical hardware connection, safety handling, and device startup instructions, refer to the product user manual.

1.2 Minimal Python Example

```python
import sys
import time
```

```
sys.path.insert(0, "/opt/sharpa-wave-sdk/python/")

from sharpa import ControlMode, ControlSource, SharpaWaveManager

def main() -> int:
    manager = SharpaWaveManager.get_instance()
    time.sleep(1)

    sns = manager.get_all_device_sn()
    if not sns:
        print("No device found")
        return 1

    wave = manager.connect(sns[0])

    if not wave.start():
        print("Failed to start device")
        manager.disconnect_all()
        return 1

    try:
        while not wave.is_hand_ready():
            time.sleep(0.2)

        err_source = wave.set_control_source(ControlSource.SDK)
        err_mode = wave.set_control_mode(ControlMode.POSITION)
        if err_source.code != 0 or err_mode.code != 0:
            print("Failed to initialize control state")
            return 1

        state = wave.get_states()
        print(f"Current joint angles: {state.angles}")

        target_angles_deg = [0.0] * 22
        err = wave.set_joint_position_degree(target_angles_deg)
        if err.code != 0:
            message = getattr(err, "to_string", None)
            print(message() if callable(message) else err.message)
        else:
```

```
            print("Command sent: move all joints to 0 degree")

        time.sleep(1)
        return 0
    finally:
        wave.set_control_mode(ControlMode.ZERO_FORCE)
        wave.stop()
        manager.disconnect_all()

if __name__ == "__main__":
    raise SystemExit(main())
```

1.3 Run

Assuming the SDK has been installed to the default system paths, run the example with:

BASH

```
python3 sharpa_wave_quick_start.py
```

For more examples, please refer to:

TEXT

```
/opt/sharpa-wave-sdk/sample
```

2. API Usage Guide

2.1 Device Discovery and Connection

2.1.1 Getting the Manager Instance

Use the singleton manager to discover and connect devices.

PYTHON

```
manager = SharpaWaveManager.get_instance()
```

2.1.2 Discovering Devices

To obtain all discovered serial numbers:

PYTHON

```
sns = manager.get_all_device_sn()
```

To obtain complete information for all discovered devices:

PYTHON

```
devices = manager.get_all_devices()
```

`get_all_devices()` returns a list of `DeviceInfo` structures.

2.1.3 Connecting by Serial Number

PYTHON

```
wave = manager.connect(device_sn)
```

Use this overload when the target serial number is known.

2.1.4 Connecting by Hand Side

PYTHON

```
from sharpa import HandSide

wave = manager.connect(HandSide.RIGHT)
```

Use this overload when you want to connect by logical hand side instead of serial number.

2.1.5 Connecting with Configuration

Use `SharpaWaveConfig` when tactile behavior or startup behavior must be customized.

```python
from sharpa import SharpaWaveConfig

config = SharpaWaveConfig()
config.disable_tactile = False
config.disable_sync_time = False
config.tactile_config_file = ""

wave = manager.connect(device_sn, config)
```

2.1.6 Checking Connection State

```python
connected = manager.is_connected(device_sn)
```

2.1.7 Disconnecting

```python
manager.disconnect(device_sn)
manager.disconnect_all()
```

After disconnecting, references to the old `SharpaWave` object must not be reused.

2.2 Device Lifecycle

2.2.1 Waiting for Readiness

After connecting, wait until the device is ready before sending commands.

PYTHON

```python
while not wave.is_hand_ready():
    time.sleep(0.2)
```

If tactile is enabled, you may also check:

PYTHON

```python
tactile_ready = wave.is_tactile_ready()
```

2.2.2 Explicit Start and Stop

After obtaining a `SharpaWave` object from `SharpaWaveManager::connect(...)`, the application should call `start()` before using tactile-related functionality.

PYTHON

```python
started = wave.start()
```

Use `stop()` when hand operation or tactile data transmission and reception need to be stopped without fully disconnecting the device.

PYTHON

```python
stopped = wave.stop()
```

2.2.3 Destroying the Device Object

```python
wave.destroy()
destroyed = wave.is_destroyed()
```

After `destroy()`, internal resources are released. The object reference should be treated as invalid until the manager reconnects the device.

2.2.4 Rebooting the Device

```python
err = wave.reboot()
```

After reboot, wait again until the device becomes ready.

2.3 Device Information and Configuration

2.3.1 Reading Device Information

Use `get_device_info()` when you need structured metadata.

```python
info = wave.get_device_info()
```

Use `get_device_info_json()` when you need a serialized JSON representation.

```python
json_str = wave.get_device_info_json()
```

Typical fields include:

- serial number
- firmware version
- hand side
- device type
- IP address
- TCP/UDP ports
- runtime status

2.3.2 Synchronizing Time

```python
err = wave.sync_time()
```

Time synchronization is typically handled during startup unless disabled by configuration.

2.3.3 Setting the Device IP Address

```python
err = wave.set_device_ip("192.168.10.30")
```

Changing the device IP address causes a device reboot. After the reboot:

- update host network settings if needed
- make sure the host is on the same subnet
- reconnect the device

2.3.4 Current and Speed Coefficients

These interfaces can be used to scale the effective current limit and speed limit of the device during operation, without changing the application-level motion logic.

The current coefficient controls the upper scaling applied to joint current output.

PYTHON

```python
err_set = wave.set_current_coeff(0.6)
if err_set.code != 0:
    print(err_set.message)

err_get, coeff = wave.get_current_coeff()
if err_get.code == 0:
    print(f"Current coefficient: {coeff}")
```

Use a smaller coefficient when lower output force or more conservative behavior is desired.

The speed coefficient controls the upper scaling applied to joint angular speed.

PYTHON

```python
err_set = wave.set_speed_coeff(0.5)
if err_set.code != 0:
    print(err_set.message)

err_get, coeff = wave.get_speed_coeff()
if err_get.code == 0:
    print(f"Speed coefficient: {coeff}")
}
```

Use a smaller coefficient when slower motion is required for testing, demonstration, or safer interaction.

Safety Recommendation:

When adjusting current or speed coefficients:

- start from the recommended default values
- increase them only when the application has been validated sufficiently
- verify the effect in a controlled environment before task execution

For applications with uncertain control quality or incomplete tuning, it is strongly recommended to keep these coefficients conservative. Excessive values may lead to

unstable motion, excessive mechanical load, or hardware damage.

2.3.5 Hardware Live Time

These interfaces query cumulative online time (in seconds) for hardware components on the device.

- Motors: 22 entries `(MotorLivetimeEntry)`. Use `is_valid()` and `uid_hex()`; an all-zero UID means the slot is invalid or offline.
- Fingertips: 10 channels `(FingerLivetimeEntry)`. Right hand: channels 0–4; left hand: 5–9. `channel_id == -1` means the slot is invalid (not installed or unsupported).

PYTHON

```python
err, livetime = wave.get_hardware_livetime()
if err.code != 0:
    print(err.message)
else:
    for i, m in enumerate(livetime.motors):
        if m.is_valid():
            print(f"Motor[{i:02d}] uid={m.uid_hex()} live={m.live_secs}s")
        else:
            print(f"Motor[{i:02d}] (invalid / offline slot)")

    for f in livetime.right_hand_fingers():
        print(f"Right finger ch{f.channel_id}: {f.live_secs}s")

    for f in livetime.left_hand_fingers():
        print(f"Left finger ch{f.channel_id}: {f.live_secs}s")
```

2.3.6 Reading Backlash History

Backlash calibration history on the device is stored as JSON records in file `0x0004`. Use `json_store_stat` to get how many records exist, then `json_store_read` to fetch them page by page.

Step 1 — Get the number of records

PYTHON

```python
BACKLASH_JSON_STORE = 0x0004

err_stat, total = wave.json_store_stat(BACKLASH_JSON_STORE)
if err_stat.code != 0:
    print(err_stat.message)
else:
    print(f"Backlash history records: {total}")
```

Step 2 — Read all records (paged)

PYTHON

```python
import json

BACKLASH_JSON_STORE = 0x0004

err_stat, total = wave.json_store_stat(BACKLASH_JSON_STORE)
if err_stat.code != 0:
    print(err_stat.message)
elif total == 0:
    print("No backlash history on device.")
else:
    page_size = 50
    offset = 0
    while offset < total:
        err_read, _, records = wave.json_store_read(
            BACKLASH_JSON_STORE, offset, page_size
        )
        if err_read.code != 0:
            print(err_read.message)
            break

        for raw in records:
            record = json.loads(raw)
            ts = record.get("ts", 0)
```

```
            backlash = record.get("backlash_data")
            print(f"timestamp={ts}, backlash_data={backlash}")

        if not records:
            break
        offset += len(records)
```

Each record is a JSON object. Fields used for backlash history:

- `ts` — Unix timestamp (seconds) when the record was saved.
- `backlash_data` — Array of per-joint backlash values in degrees; `null` means that joint was not stored in that record.

If `total` is 0, there is no backlash history in the JSON store yet (for example, before any successful backlash commit on supported firmware).

2.3.7 Generic Parameter Access

Advanced settings can be controlled with a generic JSON interface.

To set parameters:

PYTHON

```
import json

err = wave.set_parameter(json.dumps({"key": "value"}))
```

To get parameters:

PYTHON

```
err, json_str = wave.get_parameter(["key1", "key2"])
```

Further parameter definitions will be provided where applicable.

2.4 Motion Control

2.4.1 Enable State

Before motion control, it is necessary to ensure that the state is enabled.

PYTHON

```python
err1, enabled = wave.get_enable_state()
err2 = wave.set_enable_state(True)
```

- Invoking the set_control_mode or set_control_source interface will automatically transition the device to the enabled state.
- During a call to a hardware settings modification interface, the device will automatically switch to the disabled state; upon completion, it will automatically revert to the enabled state.

2.4.2 Control Source

Applications that drive the hand through the SDK should explicitly set SDK control source.

PYTHON

```python
from sharpa import ControlSource

wave.set_control_source(ControlSource.SDK)
err, source = wave.get_control_source()
```

2.4.3 Control Mode

Set the required control mode before sending commands for that mode.

PYTHON

```python
from sharpa import ControlMode
```

```
wave.set_control_mode(ControlMode.POSITION)
err, mode = wave.get_control_mode()
```

Available public control modes include:

- `ZERO_FORCE`
- `POSITION`
- `ADMITTANCE`
- `MIT`

2.4.4 Position Control

To send joint positions in radians:

PYTHON

```python
wave.set_control_mode(ControlMode.POSITION)
angles_rad = [0.0] * 22
err = wave.set_joint_position(angles_rad)
```

Explicit radians interface:

CPP

```cpp
auto err = wave.set_joint_position_rad(angles_rad);
```

To send joint positions in degrees:

PYTHON

```python
angles_deg = [0.0] * 22
err = wave.set_joint_position_degree(angles_deg)
```

For all full-hand vector-based motion interfaces, the input vector must follow the SDK-defined joint DOF ordering.

This applies to position commands and any other APIs that use full-hand joint vectors. For the complete ordering definition, refer to Appendix 1 Joint DOF Ordering.

2.4.5 Admittance Control

In Admittance mode, the controller regulates the robot's dynamic response to external interaction forces. Admittance control accepts external force as input and generates motion commands (position) as output, emulating a virtual mass-spring-damper system.

$$M_d(\ddot{x}_{ref} - \ddot{x}_d) + B_d(\dot{x}_{ref} - \dot{x}_d) + K_d(x_{ref} - x_d) = F_{ext}$$

Where:

- F_{ext}: Measured external force/torque (N·m)
- M_d: Virtual mass/inertia (kg·m²)
- B_d: Virtual damping (N·m·s/rad)
- K_d: Virtual stiffness (N·m/rad)
- x_d: Nominal target trajectory (rad)
- x_{ref}: Modified reference trajectory (rad)

To utilize this mode, first switch the control mode to ADMITTANCE and establish the nominal target trajectory using the joint position interface:

CPP

```cpp
wave.set_control_mode(sharpa::ControlMode::ADMITTANCE);
std::vector<float> angles_rad(22, 0.0f);
auto err = wave.set_joint_position(angles_rad);
```

Once the mode is active, you can configure the virtual dynamic characteristics (mass, damping, and stiffness) for each joint via the parameter setting interface. This allows you to tune the robot's "softness" or "heaviness" during interaction:

PYTHON

```
import json

params = {
    "stiffness_coeff": 0.5,
    "stiffness": [
        3.6, 3.6, 2.4, 2.4, 2.4, 3.6, 3.6, 2.4, 0.8, 3.6, 3.6,
        2.4, 0.8, 3.6, 3.6, 2.4, 0.8, 3.6, 3.6, 3.6, 2.4, 0.8,
    ],
    "mass": [0.001] * 22,
    "damping": [
        0.25, 0.25, 0.25, 0.25, 0.15, 0.25, 0.25, 0.15, 0.05,
        0.25, 0.25, 0.15, 0.05, 0.25, 0.25, 0.15, 0.05, 0.25,
        0.25, 0.25, 0.15, 0.05,
    ],
    "torque_threshold": [
        0.8, 1.0, 0.5, 1.0, 0.08, 0.5, 1.0, 0.08, 0.08, 0.5, 1.0,
        0.08, 0.08, 0.5, 1.0, 0.08, 0.08, 0.24, 0.5, 1.0, 0.08, 0.08,
    ],
}

err_param = wave.set_parameter(json.dumps(params))
if err_param.code != 0:
    print(err_param.message)
```

Parameter Definitions:

PARAMETER	TYPE	SIZE	DESCRIPTION	UNIT
mass	float	22	Mass (adjustable per joint)	kg·m²
damping	float	22	Damping (adjustable per joint)	N·m·s/rad
stiffness	float	22	Stiffness (adjustable per joint)	N·m/rad
stiffness_coeff	float	1	Stiffness coefficient (global). The final stiffness for each joint is calculated as: stiffness[i] * stiffness_coeff	Unitless
torque_threshold	float	22	Torque threshold. Admittance logic is triggered only when joint torque exceeds this value, to avoid interference from friction, etc.	N·m

PARAMETER	TYPE	SIZE	DESCRIPTION	UNIT
torque_source	int	22	Torque source type (adjustable per joint): • 0: Current (default) • 1: Force sensor (available only at root joints of each finger, e.g., thumb CMC and finger MCP joints are equipped with this sensor)	-

2.4.6 MIT Control

In MIT mode, the position loop and velocity loop are arranged in parallel. Their outputs are summed together with the feedforward torque t_{ff} to generate the reference torque T_{ref}:

$$T_{ref} = Kp * (p_{target} - p_{actual}) + Kd * (v_{target} - v_{actual}) + t_{ff}$$

Where:

- T_{ref}: Reference torque, in N·m
- Kp: Position gain
- Kd: Velocity gain
- p_{target}: Target joint position, in rad
- p_{actual}: Actual joint position, in rad
- v_{target}: Target joint velocity, in rad/s
- v_{actual}: Actual joint velocity, in rad/s
- t_{ff}: Feedforward torque, in N.m

PYTHON

```
wave.set_control_mode(ControlMode.MIT)

pos = [0.0] * 22
vel = [0.0] * 22
tau = [0.0] * 22

err = wave.set_mit_control(pos, vel, tau)
```

In MIT control mode, a set of default kp and kdparameters is provided. If these defaults do not suit your application, you can use set_parameter interface to modify them.

PYTHON

```
import json

params = {
    "torque_source": [0] * 22,
    "mit_kp": [20, 20, 8, 25, 4, 5, 25, 4, 1, 5, 25, 4, 1, 5, 25, 4, 1, 2,
5, 25, 4, 1],
    "mit_kd": [
        0.5, 0.5, 0.2, 0.25, 0.1, 0.2, 0.25, 0.1, 0.01, 0.2, 0.25,
        0.1, 0.01, 0.2, 0.25, 0.1, 0.01, 0.01, 0.2, 0.25, 0.1, 0.01,
    ],
}

err_param = wave.set_parameter(json.dumps(params))
if err_param.code != 0:
    print(err_param.message)
```

Parameter Definitions:

PARAMETER	TYPE	SIZE	DESCRIPTION	UNIT
mit_kp	float	22	Stiffness (adjustable per joint)	N·m/rad
mit_kd	float	22	Damping (adjustable per joint)	N·m·s/rad

Note: Improper kp and kdvalues may cause system oscillation.

It is recommended to keep Kd at its default value, and tune joint stiffness by adjusting Kp. The recommended Kp range is as follows(applicable only when torque_source is set to current(default); otherwise, oscillation may occur):

JOINT INDEX	KP_MIN	KP_MAX
0	10	40
1	10	40
2	2	20
3	10	25
4	2	8
5	2	20
6	10	25
7	2	8
8	0.2	1.2
9	2	20
10	10	25
11	2	8
12	0.2	1.2
13	2	20
14	10	25
15	2	8
16	0.2	1.2
17	1	2
18	2	20
19	10	25
20	2	8
21	0.2	1.2

2.4.7 Reading Joint State

Use get_states() to read the latest joint state packet.

PYTHON

```
state = wave.get_states();
```

The returned State contains:

- timestamp
- sequence
- angles
- velocities
- torques

2.4.8 Reading Joint Position Only

Convenience APIs are available for reading positions only.

PYTHON

```
err_rad, positions_rad = wave.get_joint_position_rad()
err_deg, positions_deg = wave.get_joint_position_degree()
```

2.4.9 MCU control

The MCU enable APIs control MCU power / enable state at the device level through the PM_CTL power-management channel.

`get_mcu_enable()` reads the current MCU enable state.

PYTHON

```
err, enabled = wave.get_mcu_enable()
if err.code != 0:
```

```
    raise RuntimeError(err.message)
```

`set_mcu_enable(bool enable)` sets whether the MCU is enabled.

PYTHON

```python
from sharpa import SharpaWaveManager

manager = SharpaWaveManager.get_instance()
wave = manager.connect(device_sn)
wave.start()

err = wave.set_mcu_enable(True)   # Enable MCU
err = wave.set_mcu_enable(False)  # Disable MCU
```

2.5 Tactile Data

Sharpa Wave SDK provides tactile data access for supported devices through polling and callback-based interfaces. Tactile data can include raw image data, deformation maps, force-related features, contact-point information, and other structured outputs depending on device capability and runtime configuration.

2.5.1 Capability and Readiness

Before using tactile APIs, check both support and readiness.

PYTHON

```python
supported = wave.has_tactile_support();
ready = wave.is_tactile_ready();
```

- has_tactile_support() indicates whether the connected device supports tactile sensing.
- is_tactile_ready() indicates whether the tactile pipeline has been initialized successfully and is ready to provide data.

If tactile is not supported or not ready, tactile frame APIs may fail or return no valid data.

2.5.2 Tactile Calibration

When tactile signals show drift over time, tactile calibration can be used to reset the tactile output to zero. This helps establish a clean reference baseline before tactile sensing is used in a task.

It is recommended to perform tactile calibration before starting a new task, experiment, or data-collection procedure.

To calibrate tactile zero state:

PYTHON

```
ok = wave.calib_tactile(num_frames =20, max_retry =10);
```

Before calibration:

- keep the hand stationary
- ensure fingertips are not in contact with external objects

2.5.3 Fetching Tactile Frames

To fetch a tactile frame synchronously:

CPP

```
frame = wave.fetch_tactile_frame(0, 1.0)
if frame is not None:
    raw = frame["content"].get("RAW")
    deform = frame["content"].get("DEFORM")
    f6 = frame["content"].get("F6")
    contact_point = frame["content"].get("CONTACT_POINT")
```

Where:

- channel: tactile channel index. For the definition of tactile channel indices, see Appendix 2, Tactile Channel Mapping.
- timeout: maximum wait time in seconds

Users should not assume that every frame always contains every payload. Always check whether a key exists before reading it.

2.5.4 Tactile Callbacks

To process tactile data asynchronously:

PYTHON

```python
def tactile_callback(frame):
    if frame is None:
        return

    channel = frame["channel"]
    timestamp = frame["ts"]
    raw = frame["content"].get("RAW")
    deform = frame["content"].get("DEFORM")
    f6 = frame["content"].get("F6")
    contact_point = frame["content"].get("CONTACT_POINT")


wave.set_tactile_callback(tactile_callback)
```

Callbacks run in the data acquisition path and should remain lightweight. Heavy processing should be offloaded to worker threads or a queue.

2.5.5 Frame Content Model

In Python, tactile frames are exposed as dictionary-like objects. Common fields include:

- channel: tactile channel index
- ts: frame timestamp
- content: map from payload name to payload data
- shape: map from payload name to shape information

The content field may contain payloads such as:

PYTHON

```python
raw_jpg = frame["content"].get("RAW_JPG")
raw = frame["content"].get("RAW")
force = frame["content"].get("F6")
deform_jpg = frame["content"].get("DEFORM_JPG")
deform = frame["content"].get("DEFORM")
contact_point = frame["content"].get("CONTACT_POINT")
```

Typical payloads in `frame["content"]` may include:

TYPE	DESCRIPTION	TYPICAL SHAPE	DATA FORMAT
RAW	Raw image	[1, H, W] (H=240,W=320)	uint8_t
DEFORM	Deformation map	[1, H, W] (H=240,W=240)	uint8_t
F6	Force/Tactile features (e.g., 6-dimensional or multi-channel floating-point features)	[1, 1, 6]	float32
RAW_JPG / DEFORM_JPG	JPEG-compressed image	[1, 1, N] (N is number of Bytes of jpeg)	uint8_t
CONTACT_POINT	Contact point-related floating-point data	[N, 3] (N is number of detected contacted points)	float

The returned payloads can be converted to NumPy arrays when numerical or image processing is required.

PYTHON

```python
import numpy as np
```

```
raw_data = frame["content"].get("RAW")
if raw_data is not None:
    height = frame["shape"]["RAW"][1]
    width = frame["shape"]["RAW"][2]
    raw_image = np.asarray(raw_data).reshape(height,
width).astype(np.uint8)

deform_data = frame["content"].get("DEFORM")
if deform_data is not None:
    height = frame["shape"]["DEFORM"][1]
    width = frame["shape"]["DEFORM"][2]
    deform_image = np.asarray(deform_data).reshape(height,
width).astype(np.uint8)
```

2.5.6 Deformation Mapping Helpers

The SDK provides helper functions for mapping tactile image-space information.

PYTHON

```
point = wave.deform_map_uv(channel, row, col)
mm = wave.deform_map_value(value_ui8)
```

These helpers are useful when converting deformation-map coordinates or values into more meaningful geometric or physical interpretations.

2.5.7 Tactile Statistics

To inspect tactile reception quality:

PYTHON

```
summary = wave.tactile_summary()
```

This can be used to inspect packet reception quality, dropped data, and runtime tactile streaming behavior.

2.5.8 Tactile Port Recovery

If tactile startup fails because of host port conflicts, use the recovery helpers:

PYTHON

```
ok = wave.retry_tactile_alternate_port()
ports = wave.get_tactile_alternate_ports()
port = wave.get_next_available_tactile_port([])
bound = wave.bind_tactile_port(port)
```

These APIs help recover tactile functionality when the default host port is occupied or unavailable.

2.5.9 Tactile Raw Image Disabled

To disable the transmission of tactile raw images while retaining deformation images and six-dimensional force.

PYTHON

```
# after wave.start():
err = wave.set_parameter(json.dumps({
   "secret_function": "set_tactile_jpeg_enable",
   "enable":false,
   "channel":-1,
}))
if err.code != 0;
   print(err.message)
time.sleep(0.05)
```

2.6 Hardware Management & Control

2.6.1 Motor Upgrade overview

- Firmware package must be a `.hex` file with a valid name:
  - Motor: `MotorDriver_x.y.z.hex` (e.g. `MotorDriver_0.0.19.hex`)

- Typical workflow:

```text
Connect → Query versions → Upload .hex → Confirm package on device → Start upgrade → Poll progress → Verify versions
```

- Precautions
  - Keep the TCP/PTC session stable during upgrade. `start_motor_update` only starts the job; progress must be polled via `get_motor_update_process`. If the client disconnects, the device may still be flashing motors. Poll failures or SDK timeouts do not necessarily mean the upgrade stopped.
  - Only one client should control the hand (do not run Pilot and a custom script on the same device simultaneously).
  - Do not run two upgrade sessions in parallel. `batch_motor_update` returns `OPERATION_NOT_ALLOWED` if a session is already active.
  - Do not call `start_motor_update` / `batch_motor_update` while the device reports busy / recovery window.
  - `batch_motor_update` blocks; for 22 motors it may take a long time—do not disconnect during the call.

2.6.2 Querying Motor Versions

`get_latest_motor_version()` returns the latest motor firmware version available based on the your device's

```python
err, latest = wave.get_latest_motor_version()
if err.code != 0:
    raise RuntimeError(err.message)
# latest example: "0.0.19"
```

2.6.3 Get Motor Runtime Version

`get_all_motor_versions()` returns the currently running firmware version for all 22 motors (IDs `0-21`).

PYTHON

```python
err, versions = wave.get_all_motor_versions()
if err.code != 0:
    raise RuntimeError(err.message)
for motor_id, ver in enumerate(versions):
    print(motor_id, ver)
# versions.size() == 22
# versions[i] is motor_id i, e.g. "0.0.18"
```

`get_current_motor_version(motor_id)` returns the current running firmware version for a single motor with motor_id

PYTHON

```python
motor_id = 5
err, version = wave.get_current_motor_version(motor_id)
```

2.6.4 Uploading a Motor Firmware Package

`upload_motor_firmware(hex_path, progress_callback)` uploads the `.hex` file to the device over HTTPS into device. The basename of the file must match the naming rules above.

PYTHON

```python
def on_progress(percent: int) -> None:
    print(f"upload progress: {percent}%")

err = wave.upload_motor_firmware(
    "/path/to/MotorDriver_0.0.19.hex",
    on_progress,
```

```
)
if err.code != 0:
    raise RuntimeError(err.message)

err, latest = wave.get_latest_motor_version()
```

2.6.5 Performing Motor Upgrades

Two upgrade paths are provided:

API	USE CASE	BEHAVIOR
start_motor_update	Single motor or strain channel	Asynchronous — starts upgrade only; you must poll progress
batch_motor_update	Multiple motors (0–21)	Blocking — SDK polls internally until done and verifies versions
get_motor_update_process	Used with single-motor upgrade	Query progress and status

Motor ID conventions

- Motors: `0-21`
- Strain channels: `32-36` (supported by `start_motor_update` and `get_motor_update_process` only; `batch_motor_update` supports motors `0-21` only)

`MotorUpgradeStatus` fields

FIELD	MEANING
progress	-1 unknown / not started; 100 completed; ≥101 failed
isActive	true while upgrade is in progress
targetVersion	Target firmware version
motor_id	Motor ID currently being processed
errorMessage	Error description when failed

upgrade single motor

PYTHON

```python
import time

motor_id = 5
target_version = "0.0.19"

err = wave.start_motor_update(motor_id, target_version)
if err.code != 0:
    raise RuntimeError(err.message)

while True:
    poll_err, status = wave.get_motor_update_process(motor_id)
    if poll_err.code != 0:
        raise RuntimeError(poll_err.message)

    if not status.isActive and status.progress == 100:
        break
    if not status.isActive and status.progress >= 101:
        raise RuntimeError(status.errorMessage or "motor update failed")

    time.sleep(0.5)

err, current = wave.get_current_motor_version(motor_id)
```

Batch upgrade

PYTHON

```python
targets = [
    (0, "0.0.19"),
    (1, "0.0.19"),
    (5, "0.0.20"),
]

err, results = wave.batch_motor_update(targets)
```

```python
if err.code != 0:
    raise RuntimeError(err.message)

for motor_id, ok in results.items():
    print(motor_id, "OK" if ok else "FAILED")
```

3. Troubleshooting and Error Handling

3.1 Error Model

Most SDK operations return either an `Error` object or a tuple of (Error, value).

An Error with code == 0 indicates that the operation completed successfully. If `code != 0`, the operation failed, and the `message` field provides additional diagnostic information.

For the complete list of error codes and their meanings, refer to Appendix 3 Error Codes.

Example:

PYTHON

```python
err = wave.set_control_mode(ControlMode.POSITION)
if err.code != 0:
    print(err.message)
```

For APIs that return `(Error, value)`, always check the `Error` object before using the returned value:

PYTHON

```python
err, mode = wave.get_control_mode()
if err.code == 0:
    print(mode)
```

Recommended practice:

- always check the returned `Error` before continuing
- log both `Error.code` and `Error.message` during debugging
- do not assume that a returned value is valid unless the associated `Error` indicates success

3.2 Fault Inspection

Use `get_fault_code()` to inspect current device and software fault conditions.

```python
faults = wave.get_fault_code()
```

The returned map associates each FaultCode with a list of relevant indices, if applicable. The meaning of these indices depends on the fault context. For example:

- for motion-related faults, the indices may refer to joint indices
- for tactile-related faults, the indices may refer to tactile channels
- for some system-level faults, no specific index may apply, and the associated vector may be empty

For the complete list of fault codes and their meanings, refer to Appendix 4 Fault Codes.

Recommended practice:

- check reported fault codes when device behavior is abnormal
- interpret the associated indices according to the subsystem involved
- use the reported indices to narrow down motion, tactile, or other subsystem issues when available

3.3 Common Problems

3.3.1 No device discovered

Check the following:

- the device is powered on

- the host and device are on the same subnet
- firewall or endpoint security software is not blocking required traffic
- heartbeat packets can reach the host

3.3.2 Connection fails

Check the following:

- the serial number is correct
- the selected hand side uniquely identifies one device
- the device is reachable on the network
- the SDK runtime library is correctly installed

3.3.3 The device does not respond to motion commands

Check the following:

- `is_hand_ready()` returns `true`
- control source is set to SDK
- the correct control mode has been selected
- the device is enabled
- command vectors use the correct DOF order and length

3.3.4 Tactile data is not received

Check the following:

- the device supports tactile sensing
- tactile is enabled and ready
- the host port required for tactile reception is available
- the configured host IP and target port are correct
- another application is not occupying the tactile port

3.3.5 Port occupied or tactile bind failure

Use the tactile port recovery helpers:

- retry_tactile_alternate_port()

- get_next_available_tactile_port()
- bind_tactile_port()

3.3.6 Device becomes unreachable after IP change

Check the following:

- the device has completed reboot
- the host computer is on the same subnet as the new device IP
- the application reconnects after the reboot

3.3.7 Recommended Debugging Practices

- always log Error.code and Error.message
- wait for readiness before issuing commands
- log discovered devices and their IP addresses during startup
- validate command vector sizes before sending motion commands
- keep tactile callbacks lightweight
- disconnect cleanly before application exit

C++ SDK Guide

1. Quick Start

This section provides the shortest recommended path to a working integration.

1.1 Before You Start

Before running the example:

- power on the Sharpa Wave
- connect the host and the device to the same subnet
- confirm that the SDK runtime library is installed correctly

For physical hardware connection, safety handling, and device startup instructions, refer to the product user manual.

1.2 Minimal C++ Example

CPP

```cpp
#include "SharpaWaveSDK.h"
#include <chrono>
#include <iostream>
#include <thread>
#include <vector>

int main() {
    auto& manager = sharpa::SharpaWaveManager::get_instance();
    std::this_thread::sleep_for(std::chrono::seconds(1));
    auto sns = manager.get_all_device_sn();
    if (sns.empty()) {
        std::cout << "No device found" << std::endl;
        return 1;
    }

    auto& wave = manager.connect(sns[0]);
    if (!wave.start()) {
        std::cerr << "Failed to start device" << std::endl;
        manager.disconnect_all();
        return 1;
    }

    while (!wave.is_hand_ready()) {
        std::this_thread::sleep_for(std::chrono::milliseconds(200));
    }

    auto err_source = wave.set_control_source(sharpa::ControlSource::SDK);
    auto err_mode = wave.set_control_mode(sharpa::ControlMode::POSITION);

    if (err_source.code != 0 || err_mode.code != 0) {
        std::cerr << "Failed to initialize control state" << std::endl;
        wave.stop();
        manager.disconnect_all();
        return 1;
```

```
    }

    // Get current state
    auto state = wave.get_states();
    std::cout << "Current joint count: " << state.angles.size() <<
std::endl;

    // Move all joints to 0 degree
    std::vector<float> target_angles_deg(22, 0.0f);
    auto err = wave.set_joint_position_degree(target_angles_deg);

    if (err.code != 0) {
        std::cerr << err.to_string() << std::endl;
    } else {
        std::cout << "Command sent: move all joints to 0 degree" <<
std::endl;
    }

    std::this_thread::sleep_for(std::chrono::seconds(1));

    wave.set_control_mode(sharpa::ControlMode::ZERO_FORCE);
    wave.stop();
    manager.disconnect_all();
    return 0;
}
```

1.3 Compile and Run

Assuming the SDK has been installed to the default system paths, you can compile the example with:

CPP

```
g++ -std=c++17 sharpa_wave_quick_start.cc -I/opt/sharpa-wave-sdk -
L/opt/sharpa-wave-sdk -lsharpa-wave-sdk -ldl -Wl,-rpath,/opt/sharpa-wave-
sdk -o sharpa_wave_quick_start
```

Then run the program with:

```cpp
./sharpa_wave_quick_start
```

For more examples, please refer to `/opt/sharpa-wave-sdk/sample`.

2. API Usage Guide

2.1 Device Discovery and Connection

2.1.1 Getting the Manager Instance

Use the singleton manager to discover and connect devices.

```cpp
auto& manager = sharpa::SharpaWaveManager::get_instance();
```

2.1.2 Discovering Devices

To obtain all discovered serial numbers:

```cpp
auto sns = manager.get_all_device_sn();
```

To obtain complete information for all discovered devices:

```cpp
auto devices = manager.get_all_devices();
```

`get_all_devices()` returns a list of DeviceInfo structures.

2.1.3 Connecting by Serial Number

```cpp
auto& wave = manager.connect(device_sn);
```

Use this overload when the target serial number is known.

2.1.4 Connecting by Hand Side

```cpp
auto& wave = manager.connect(sharpa::HandSide::RIGHT);
```

Use this overload when you want to connect by logical hand side instead of serial number.

2.1.5 Connecting with Configuration

Use `SharpaWaveConfig` when tactile behavior or startup behavior must be customized.

```cpp
sharpa::SharpaWaveConfig config;
config.disable_tactile = false;
config.disable_sync_time = false;
config.tactile_config_file = "";

auto& wave = manager.connect(device_sn, config);
```

2.1.6 Checking Connection State

```cpp
bool connected = manager.is_connected(device_sn);
```

2.1.7 Disconnecting

```cpp
manager.disconnect(device_sn);
manager.disconnect_all();
```

After disconnecting, references to the old SharpaWave object must not be reused.

2.2 Device Lifecycle

2.2.1 Waiting for Readiness

After connecting, wait until the device is ready before sending commands.

```cpp
while (!wave.is_hand_ready()) {
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
}
```

If tactile is enabled, you may also check:

```cpp
bool tactile_ready = wave.is_tactile_ready();
```

2.2.2 Explicit Start and Stop

After obtaining a `SharpaWave` object from `SharpaWaveManager::connect(...)`, the application should call `start()` before using tactile-related functionality.

```cpp
bool started = wave.start();
```

Use `stop()` when hand operation or tactile data transmission and reception need to be stopped without fully disconnecting the device.

CPP

```cpp
bool stopped = wave.stop();
```

2.2.3 Destroying the Device Object

CPP

```cpp
wave.destroy();
bool destroyed = wave.is_destroyed();
```

After `destroy()`, internal resources are released. The object reference should be treated as invalid until the manager reconnects the device.

2.2.4 Rebooting the Device

CPP

```cpp
auto err = wave.reboot();
```

After reboot, wait again until the device becomes ready.

2.3 Device Information and Configuration

2.3.1 Reading Device Information

Use `get_device_info()` when you need structured metadata.

CPP

```cpp
auto info = wave.get_device_info();
```

Use `get_device_info_json()` when you need a serialized JSON representation.

```cpp
std::string json = wave.get_device_info_json();
```

Typical fields include:

- serial number
- firmware version
- hand side
- device type
- IP address
- TCP/UDP ports
- runtime status

2.3.2 Synchronizing Time

```cpp
auto err = wave.sync_time();
```

Time synchronization is typically handled during startup unless disabled by configuration.

2.3.3 Setting the Device IP Address

```cpp
auto err = wave.set_device_ip("192.168.10.30");
```

Changing the device IP address causes a device reboot. After the reboot:

- update host network settings if needed
- make sure the host is on the same subnet
- reconnect the device

2.3.4 Current and Speed Coefficients

These interfaces can be used to scale the effective current limit and speed limit of the device during operation, without changing the application-level motion logic.

The current coefficient controls the upper scaling applied to joint current output.

CPP

```cpp
auto err_set = wave.set_current_coeff(0.6f);
if (err_set.code != 0) {
    std::cerr << err_set.to_string() << std::endl;
}

auto [err_get, coeff] = wave.get_current_coeff();
if (err_get.code == 0) {
    std::cout << "Current coefficient: " << coeff << std::endl;
}
```

Use a smaller coefficient when lower output force or more conservative behavior is desired.

The speed coefficient controls the upper scaling applied to joint angular speed.

CPP

```cpp
auto err_set = wave.set_speed_coeff(0.5f);
if (err_set.code != 0) {
    std::cerr << err_set.to_string() << std::endl;
}

auto [err_get, coeff] = wave.get_speed_coeff();
if (err_get.code == 0) {
    std::cout << "Speed coefficient: " << coeff << std::endl;
}
```

Use a smaller coefficient when slower motion is required for testing, demonstration, or safer interaction.

Safety Recommendation：

When adjusting current or speed coefficients:

- start from the recommended default values
- increase them only when the application has been validated sufficiently
- verify the effect in a controlled environment before task execution

For applications with uncertain control quality or incomplete tuning, it is strongly recommended to keep these coefficients conservative. Excessive values may lead to unstable motion, excessive mechanical load, or hardware damage.

2.3.5 Hardware Live Time

These interfaces query cumulative online time (in seconds) for hardware components on the device.

- Motors: 22 entries (`MotorLivetimeEntry`). Use `is_valid()` and `uid_hex()` ; an all-zero UID means the slot is invalid or offline.
- Fingertips: 10 channels (`FingerLivetimeEntry`). Right hand: channels 0–4; left hand: 5–9. `channel_id == -1` means the slot is invalid (not installed or unsupported).

CPP

```cpp
#include "SharpaWaveSDK.h"
#include <cstdio>
#include <iostream>

auto [err, livetime] = wave->get_hardware_livetime();
if (err.code != 0) {
    std::cout << err.message << std::endl;
} else {
    for (size_t i = 0; i < livetime.motors.size(); ++i) {
        const auto& m = livetime.motors[i];
        if (m.is_valid()) {
            std::string uid_hex;
            char buf[3];
            for (uint8_t b : m.uid) {
                std::snprintf(buf, sizeof(buf), "%02x", b);
```

```
                uid_hex += buf;
            }
            std::cout << "Motor[" << i << "] uid=" << uid_hex
                      << " live=" << m.live_secs << "s" << std::endl;
        } else {
            std::cout << "Motor[" << i << "] (invalid / offline slot)" <<
std::endl;
        }
    }

    for (const auto& f : livetime.right_hand_fingers()) {
        std::cout << "Right finger ch" << static_cast<int>(f.channel_id)
                  << ": " << f.live_secs << "s" << std::endl;
    }

    for (const auto& f : livetime.left_hand_fingers()) {
        std::cout << "Left finger ch" << static_cast<int>(f.channel_id)
                  << ": " << f.live_secs << "s" << std::endl;
    }
}
```

2.3.6 Reading Backlash History

Backlash calibration history on the device is stored as JSON records in file `0x0004`. Use `json_store_stat` to get how many records exist, then `json_store_read` to fetch them page by page.

Step 1 — Get the number of records

CPP

```
constexpr uint16_t BACKLASH_JSON_STORE = 0x0004;

auto [err_stat, total] = wave->json_store_stat(BACKLASH_JSON_STORE);
if (err_stat.code != 0) {
    std::cout << err_stat.message << std::endl;
} else {
```

```
    std::cout << "Backlash history records: " << total << std::endl;
}
```

Step 2 — Read all records (paged)

CPP

```cpp
#include <nlohmann/json.hpp>

constexpr uint16_t BACKLASH_JSON_STORE = 0x0004;

auto [err_stat, total] = wave->json_store_stat(BACKLASH_JSON_STORE);
if (err_stat.code != 0) {
    std::cout << err_stat.message << std::endl;
} else if (total == 0) {
    std::cout << "No backlash history on device." << std::endl;
} else {
    const uint16_t page_size = 50;
    uint16_t offset = 0;
    while (offset < total) {
        auto [err_read, store_total, records] =
            wave->json_store_read(BACKLASH_JSON_STORE, offset, page_size);
        if (err_read.code != 0) {
            std::cout << err_read.message << std::endl;
            break;
        }

        for (const std::string& raw : records) {
            auto record = nlohmann::json::parse(raw);
            int64_t ts = record.value("ts", 0);
            auto backlash = record.find("backlash_data");
            std::cout << "timestamp=" << ts << ", backlash_data="
                      << (backlash != record.end() ? backlash->dump() :
"null")
                      << std::endl;
        }

        if (records.empty()) {
```

```
            break;
        }
        offset += static_cast<uint16_t>(records.size());
    }
}
```

Each record is a JSON object. Fields used for backlash history:

- `ts` — Unix timestamp (seconds) when the record was saved.
- `backlash_data` — Array of per-joint backlash values in degrees; `null` means that joint was not stored in that record.

If `total` is 0, there is no backlash history in the JSON store yet (for example, before any successful backlash commit on supported firmware).

2.3.7 Generic Parameter Access

Advanced settings can be controlled with a generic JSON interface.

To set parameters:

CPP

```cpp
auto err = wave.set_parameter(R"({"key":"value"})");
```

To get parameters:

CPP

```cpp
auto [err, json_str] = wave.get_parameter({"key1", "key2"});
```

Further parameter definitions will be provided where applicable.

2.4 Motion Control

2.4.1 Enable State

Before motion control, it is necessary to ensure that the state is enabled.

CPP

```cpp
auto [err1, enabled] = wave.get_enable_state();
auto err2 = wave.set_enable_state(true);
```

- Invoking the set_control_mode or set_control_source interface will automatically transition the device to the enabled state.
- During a call to a hardware settings modification interface, the device will automatically switch to the disabled state; upon completion, it will automatically revert to the enabled state.

2.4.2 Control Source

Applications that drive the hand through the SDK should explicitly set SDK control source.

CPP

```cpp
wave.set_control_source(sharpa::ControlSource::SDK);
auto [err, source] = wave.get_control_source();
```

2.4.3 Control Mode

Set the required control mode before sending commands for that mode.

CPP

```cpp
wave.set_control_mode(sharpa::ControlMode::POSITION);
auto [err, mode] = wave.get_control_mode();
```

Available public control modes include:

- ZERO_FORCE
- POSITION
- ADMITTANCE
- MIT

2.4.4 Position Control

To send joint positions in radians:

```cpp
wave.set_control_mode(sharpa::ControlMode::POSITION);
std::vector<float> angles_rad(22, 0.0f);
auto err = wave.set_joint_position(angles_rad);
```

Explicit radians interface:

```cpp
auto err = wave.set_joint_position_rad(angles_rad);
```

To send joint positions in degrees:

```cpp
std::vector<float> angles_deg(22, 0.0f);
auto err = wave.set_joint_position_degree(angles_deg);
```

For all full-hand vector-based motion interfaces, the input vector must follow the SDK-defined joint DOF ordering.
This applies to position commands and any other APIs that use full-hand joint vectors.
For the complete ordering definition, refer to Appendix 1 Joint DOF Ordering.

2.4.5 Admittance Control

In Admittance mode, the controller regulates the robot's dynamic response to external interaction forces. Admittance control accepts external force as input and generates motion commands (position) as output, emulating a virtual mass-spring-damper system.

$$M_d(\ddot{x}_{ref} - \ddot{x}_d) + B_d(\dot{x}_{ref} - \dot{x}_d) + K_d(x_{ref} - x_d) = F_{ext}$$

Where:

- F_{ext}: Measured external force/torque (N·m)
- M_d: Virtual mass/inertia (kg·m²)
- B_d: Virtual damping (N·m·s/rad)
- K_d: Virtual stiffness (N·m/rad)
- x_d: Nominal target trajectory (rad)
- x_{ref}: Modified reference trajectory (rad)

To utilize this mode, first switch the control mode to `ADMITTANCE` and establish the nominal target trajectory using the joint position interface:

CPP

```cpp
wave.set_control_mode(sharpa::ControlMode::ADMITTANCE);
std::vector<float> angles_rad(22, 0.0f);
auto err = wave.set_joint_position(angles_rad);
```

Once the mode is active, you can configure the virtual dynamic characteristics (mass, damping, and stiffness) for each joint via the parameter setting interface. This allows you to tune the robot's "softness" or "heaviness" during interaction:

JSON

```json
std::string json_str = R"({
    "stiffness_coeff": 0.5,
    "stiffness": [3.6, 3.6, 2.4, 2.4, 2.4, 3.6, 3.6, 2.4, 0.8, 3.6, 3.6,
                  2.4, 0.8, 3.6, 3.6, 2.4, 0.8, 3.6, 3.6, 3.6, 2.4, 0.8],
```

```
    "mass": [0.001, 0.001, 0.001, 0.001, 0.001, 0.001, 0.001, 0.001, 0.001,
0.001, 0.001,
             0.001, 0.001, 0.001, 0.001, 0.001, 0.001, 0.001, 0.001, 0.001,
0.001, 0.001],
    "damping": [0.25, 0.25, 0.25, 0.25, 0.15, 0.25, 0.25, 0.15, 0.05, 0.25,
0.25,
                0.15, 0.05, 0.25, 0.25, 0.15, 0.05, 0.25, 0.25, 0.25, 0.15,
0.05],
    "torque_threshold": [0.8, 1.0, 0.5, 1.0, 0.08, 0.5, 1.0, 0.08,
                         0.08, 0.5, 1.0, 0.08, 0.08, 0.5, 1.0, 0.08,
                         0.08, 0.24, 0.5, 1.0, 0.08, 0.08]
})";

auto err_param = wave.set_parameter(json_str);
if (err_param.code != 0) {
    std::cerr << err_param.to_string() << std::endl;
}
```

Parameter Definitions:

2.4.6 MIT Control

In MIT mode, the position loop and velocity loop are arranged in parallel. Their outputs are summed together with the feedforward torque t_{ff} to generate the reference torque T_{ref}:

$$T_{ref} = Kp * (p_{target} - p_{actual}) + Kd * (v_{target} - v_{actual}) + t_{ff}$$

Where:

- T_{ref}: Reference torque, in N·m
- Kp: Position gain
- Kd: Velocity gain
- p_{target}: Target joint position, in rad
- p_{actual}: Actual joint position, in rad
- v_{target}: Target joint velocity, in rad/s
- v_{actual}: Actual joint velocity, in rad/s

- t_{ff}: Feedforward torque, in N.m

```cpp
std::vector<float> pos(22, 0.0f);
std::vector<float> vel(22, 0.0f);
std::vector<float> tau(22, 0.0f);
auto err = wave.set_mit_control(pos, vel, tau);
// Alternatively, if velocity and torque are constantly zero, use:
auto err = wave.set_joint_position(pos);
```

In MIT control mode, a set of default kp and kdparameters is provided. If these defaults do not suit your application, you can use set_parameter interface to modify them.

```cpp
std::string json_str = R"({
    "torque_source": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    "mit_kp": [20, 20, 8, 25, 4, 5, 25, 4, 1, 5, 25, 4, 1, 5, 25, 4, 1, 2, 5, 25, 4, 1],
    "mit_kd": [0.5, 0.5, 0.2, 0.25, 0.1, 0.2, 0.25, 0.1, 0.01, 0.2, 0.25,
               0.1, 0.01, 0.2, 0.25, 0.1, 0.01, 0.01, 0.2, 0.25, 0.1, 0.01]
})";

auto err_param = wave.set_parameter(json_str);
if (err_param.code != 0) {
    std::cerr << err_param.to_string() << std::endl;
}
```

Parameter Definitions:

PARAMETER	TYPE	SIZE	DESCRIPTION	UNIT
mit_kp	float	22	Stiffness (adjustable per joint)	N·m/rad
mit_kd	float	22	Damping (adjustable per joint)	N·m·s/rad

Note: Improper `kp` and `kd` values may cause system oscillation.

It is recommended to keep `Kd` at its default value, and tune joint stiffness by adjusting `Kp`. The recommended Kp range is as follows(applicable only when `torque_source` is set to current(default); **otherwise, oscillation may occur**):

JOINT INDEX	KP_MIN	KP_MAX
0	10	40
1	10	40
2	2	20
3	10	25
4	2	8
5	2	20
6	10	25
7	2	8
8	0.2	1.2
9	2	20
10	10	25
11	2	8
12	0.2	1.2
13	2	20
14	10	25
15	2	8
16	0.2	1.2
17	1	2
18	2	20
19	10	25
20	2	8
21	0.2	1.2

2.4.7 Reading Joint State

`Use get_states()` to read the latest joint state packet.

```cpp
auto state = wave.get_states();
```

The returned `State` contains:

- `timestamp`
- `sequence`
- `angles`
- `velocities`
- `torques`

2.4.8 Reading Joint Position Only

Convenience APIs are available for reading positions only.

```cpp
auto [err_rad, positions_rad] = wave.get_joint_position_rad();
auto [err_deg, positions_deg] = wave.get_joint_position_degree();
```

2.4.9 MCU control

The MCU enable APIs control MCU power / enable state at the device level through the PM_CTL power-management channel.

`get_mcu_enable()` reads the current MCU enable state.

```cpp
auto [err, enabled] = wave.get_mcu_enable();
if (err.code == 0) {
    // enabled == true  -> MCU enabled
```

```
    // enabled == false -> MCU disabled
}
```

`set_mcu_enable(bool enable)` sets whether the MCU is enabled.

CPP

```cpp
#include "SharpaWaveSDK.h"

auto& manager = sharpa::SharpaWaveManager::get_instance();
auto& wave = manager.connect(device_sn);
wave.start();

sharpa::Error err = wave.set_mcu_enable(true);   // Enable MCU
if (err.code != 0) {
    // handle error
}

err = wave.set_mcu_enable(false);  // Disable MCU
```

2.5 Tactile Data

Sharpa Wave SDK provides tactile data access for supported devices through polling and callback-based interfaces. Tactile data can include raw image data, deformation maps, force-related features, contact-point information, and other structured outputs depending on device capability and runtime configuration.

2.5.1 Capability and Readiness

Before using tactile APIs, check both support and readiness.

CPP

```cpp
bool supported = wave.has_tactile_support();
bool ready = wave.is_tactile_ready();
```

- `has_tactile_support()` indicates whether the connected device supports tactile sensing.
- `is_tactile_ready()` indicates whether the tactile pipeline has been initialized successfully and is ready to provide data.

If tactile is not supported or not ready, tactile frame APIs may fail or return no valid data.

2.5.2 Tactile Calibration

When tactile signals show drift over time, tactile calibration can be used to reset the tactile output to zero. This helps establish a clean reference baseline before tactile sensing is used in a task.

It is recommended to perform tactile calibration before starting a new task, experiment, or data-collection procedure.

To calibrate tactile zero state:

CPP

```cpp
bool ok = wave.calib_tactile(/*num_frames =*/20, /*max_retry =*/ 10);
```

Before calibration:

- keep the hand stationary
- ensure fingertips are not in contact with external objects

2.5.3 Fetching Tactile Frames

The SDK provides two ways to fetch tactile data synchronously.

To fetch a full tactile frame:

CPP

```cpp
auto frame = wave.fetch_tactile_frame(/*channel=*/0, /*timeout=*/1.0);
if (frame) {
```

```
    // use frame->content
}
```

To fetch a compact tactile frame:

```cpp
auto frame = wave.fetch_tactile_frame_compact(/*channel=*/0,
/*timeout=*/1.0);
if (frame.frame_id > 0) {
    // use compact vectors
}
```

Where:

- channel: tactile channel index. For the definition of tactile channel indices, see Appendix 2, Tactile Channel Mapping.
- timeout: maximum wait time in seconds

Use `fetch_tactile_frame()` when you want the most general data representation. Use fetch_tactile_frame_compact() when you want a simplified structure with commonly used outputs unpacked into vectors.

2.5.4 Tactile Callbacks

To process tactile data asynchronously:

```cpp
wave.set_tactile_callback([](sharpa::tactile::Frame::Ptr frame) {
    if (!frame) return;
    // lightweight processing only
});
```

For compact frame callbacks:

```cpp
wave.set_tactile_callback_compact([](const sharpa::tactile::CompactFrame&
frame) {
    // lightweight processing only
});
```

Callbacks run in the data acquisition path and should remain lightweight. Heavy processing should be offloaded to worker threads or a queue.

2.5.5 Frame Content Model

The frame data returned by `fetch_tactile_frame()` or delivered through `set_tactile_callback()` contains a `content` field:

```cpp
// Type:
std::map<std::string, sharpa::tactile::DataBlock::Ptr>
```

`Frame::content` is a map from payload name to payload data block.

- The **key** is the name of the tactile data type, such as `"RAW"`, `"DEFORM"`, `"F6"`, or `"CONTACT_POINT"`.
- The **value** is an instance of `sharpa::tactile::DataBlock`, which stores the underlying tensor-like data buffer together with its shape information.

This means that a tactile frame is not limited to a single image or a single numeric vector. Instead, it is a container of one or more tactile outputs generated for the same frame.

A `DataBlock` is the generic data representation used for tactile payloads. It provides:

- `shape()` → dimensions of the payload, for example `[1, H, W]` or `[1, 1, 6]`
- `size()` → number of elements
- `nbytes()` → total byte size
- `data()` → raw pointer to the underlying buffer

Example:

```cpp
auto frame = wave.fetch_tactile_frame(0, 1.0);
if (!frame) return;

auto raw_jpg = frame->content["RAW_JPG"]; // get original image in JPG format
auto raw = frame->content["RAW"];  // get original image in matrix format
auto force  = frame->content["F6"]; // get 6-DOF force
auto deform_jpg = frame->content["DEFORM_JPG"]; // get deformation depth map in JPG format
auto deform = frame->content["DEFORM"]; // get deformation depth map in matrix format
auto contact_point = frame->content["CONTACT_POINT"]; // get contact point in matrix format
```

Because data() returns a raw pointer, the application must cast it to the correct element type, such as `uint8_t*` or `float*`, depending on the payload being read.

Typical payloads in `Frame::content` may include:

TYPE	DESCRIPTION	TYPICAL SHAPE	DATA FORMAT
RAW	Raw image	[1, H, W] (H=240,W=320)	uint8_t
DEFORM	Deformation map	[1, H, W] (H=240,W=240)	uint8_t
F6	Force/Tactile features (e.g., 6-dimensional or multi-channel floating-point features)	[1, 1, 6]	float32
RAW_JPG / DEFORM_JPG	JPEG-compressed image	[1, 1, N] (N is number of Bytes of jpeg)	uint8_t
CONTACT_POINT	Contact point-related floating-point data	[N, 3] (N is number of detected contacted points)	float

Users should not assume that every frame always contains every payload. Always check whether a key exists before reading it.

`A CompactFrame` exposes commonly used payloads through direct vectors and shape objects rather than a generic `content map`. Use `CompactFrame` when you want a simpler access path to standard tactile outputs, and use the full `Frame` representation when you need flexible payload lookup.

CPP

```cpp
auto frame = wave.fetch_tactile_frame_compact(0, 1.0);
if (frame.frame_id <= 0) {
    return;
}

// Access compact payloads directly
const auto& raw = frame.raw_data;
const auto& deform = frame.deform_data;
const auto& f6 = frame.f6_data;
const auto& contact_points = frame.contact_point_data;
```

```cpp
// Access shape metadata
auto raw_shape = frame.raw_shape;
auto deform_shape = frame.deform_shape;
auto f6_shape = frame.f6_shape;
auto contact_shape = frame.contact_point_shape;
```

2.5.6 Deformation Mapping Helpers

The SDK provides helper functions for mapping tactile image-space information.

CPP

```cpp
auto point = wave.deform_map_uv(channel, row, col);
float mm = wave.deform_map_value(value_ui8);
```

These helpers are useful when converting deformation-map coordinates or values into more meaningful geometric or physical interpretations.

2.5.7 Tactile Statistics

To inspect tactile reception quality:

CPP

```cpp
std::string summary = wave.tactile_summary();
```

This can be used to inspect packet reception quality, dropped data, and runtime tactile streaming behavior.

2.5.8 Tactile Port Recovery

If tactile startup fails because of host port conflicts, use the recovery helpers:

CPP

```cpp
bool ok = wave.retry_tactile_alternate_port();
auto ports = wave.get_tactile_alternate_ports();
```

```cpp
int port = wave.get_next_available_tactile_port({});
bool bound = wave.bind_tactile_port(port);
```

These APIs help recover tactile functionality when the default host port is occupied or unavailable.

2.5.9 Tactile Raw Image Disabled

To disable the transmission of tactile raw images while retaining deformation images and six-dimensional force.

CPP

```cpp
// after hand.start():
const std::string kDisableJpeg =
   R"
({"secret_function":"set_tactile_jpeg_enable","enable":false,"channel":-1})
";
auto err = hand.set_parameter(kDisableJpeg);
if (err.code != 0) {
    std::cerr << err.to_string() << std::endl;
}
std::this_thread::sleep_for(std::chrono::milliseconds(50));
```

2.6 Hardware Management & Control

2.6.1 Motor Upgrade overview

- Firmware package must be a `.hex` file with a valid name:
  - Motor: `MotorDriver_x.y.z.hex` (e.g. `MotorDriver_0.0.19.hex`)
- Typical workflow:

TEXT

```
Connect → Query versions → Upload .hex → Confirm package on device → Start upgrade → Poll progress → Verify versions
```

- Precautions
  - Keep the TCP/PTC session stable during upgrade. `start_motor_update` only starts the job; progress must be polled via `get_motor_update_process`. If the client disconnects, the device may still be flashing motors. Poll failures or SDK timeouts do not necessarily mean the upgrade stopped.
  - Only one client should control the hand (do not run Pilot and a custom script on the same device simultaneously).
  - Do not run two upgrade sessions in parallel. `batch_motor_update` returns `OPERATION_NOT_ALLOWED` if a session is already active.
  - Do not call `start_motor_update` / `batch_motor_update` while the device reports busy / recovery window.
  - `batch_motor_update` blocks; for 22 motors it may take a long time—do not disconnect during the call.

2.6.2 Querying Motor Versions

`get_latest_motor_version()` returns the latest motor firmware version available based on the your device's

CPP

```cpp
#include "SharpaWaveSDK.h"

auto [err, latest] = wave.get_latest_motor_version();
if (err.code != 0) {
    // handle error
}
// latest example: "0.0.19"
```

2.6.3 Get Motor Runtime Version

`get_all_motor_versions()` returns the currently running firmware version for all 22 motors (IDs `0-21`).

CPP

```cpp
auto [err, versions] = wave.get_all_motor_versions();
if (err.code != 0) {
    // handle error
}
// versions.size() == 22
// versions[i] is motor_id i, e.g. "0.0.18"
```

`get_current_motor_version(motor_id)` returns the current running firmware version for a single motor with motor_id

CPP

```cpp
uint8_t motor_id = 5;
auto [err, version] = wave.get_current_motor_version(motor_id);
```

2.6.4 Uploading a Motor Firmware Package

`upload_motor_firmware(hex_path, progress_callback)` uploads the `.hex` file to the device over HTTPS into device. The basename of the file must match the naming rules above.

CPP

```cpp
std::string hex_path = "/path/to/MotorDriver_0.0.19.hex";

sharpa::Error err = wave.upload_motor_firmware(
    hex_path,
    [](int percent) {
        std::cout << "upload progress: " << percent << "%" << std::endl;
    });
if (err.code != 0) {
    // handle error
}
```

```cpp
// Optional: confirm the device sees the uploaded package
auto [ver_err, latest] = wave.get_latest_motor_version();
```

2.6.5 Performing Motor Upgrades

Two upgrade paths are provided:

Motor ID conventions

- Motors: `0-21`
- Strain channels: `32-36` (supported by `start_motor_update` and `get_motor_update_process` only; `batch_motor_update` supports motors `0-21` only)

`MotorUpgradeStatus` fields

upgrade single motor

CPP

```cpp
uint8_t motor_id = 5;
std::string target_version = "0.0.19";

sharpa::Error start_err = wave.start_motor_update(motor_id,
target_version);
if (start_err.code != 0) {
    // handle error
}

while (true) {
    auto [poll_err, status] = wave.get_motor_update_process(motor_id);
    if (poll_err.code != 0) {
        break;
    }

    if (!status.isActive && status.progress == 100) {
        break;  // success
    }
    if (!status.isActive && status.progress >= 101) {
        // failed: status.errorMessage
```

```
        break;
    }

    std::this_thread::sleep_for(std::chrono::milliseconds(500));
}

auto [ver_err, current] = wave.get_current_motor_version(motor_id);
```

Batch upgrade

CPP

```
std::vector<std::pair<uint8_t, std::string>> targets = {
    {0, "0.0.19"},
    {1, "0.0.19"},
    {5, "0.0.20"},  // different target version is allowed
};

auto [err, results] = wave.batch_motor_update(targets);
// results[motor_id] == true  -> verified success
// results[motor_id] == false -> failed or skipped
```

3. Troubleshooting and Error Handling

3.1 Error Model

Most SDK operations return either `sharpa::Error` or `std::pair<Error, T>.`

An `Erro`r `with` code == 0 `indicates that the operation completed successfully.`
If code != 0`, the operation failed, and the` message` field provides additional diagnostic information.

For the complete list of error codes and their meanings, refer to Appendix 3 Error Codes.

Example:

CPP

```cpp
auto err = wave.set_control_mode(sharpa::ControlMode::POSITION);
if (err.code != 0) {
    std::cerr << err.to_string() << std::endl;
}
```

For APIs that return `std::pair<Error, T>`, always check the Error object before using the returned value:

CPP

```cpp
auto [err, mode] = wave.get_control_mode();
if (err.code == 0) {
    // use mode
}
```

Recommended practice:

- always check the returned Error before continuing
- log both Error.code and Error.message during debugging
- do not assume that a returned value is valid unless the associated Error indicates success

3.2 Fault Inspection

Use `get_fault_code()` to inspect current device and software fault conditions.

CPP

```cpp
std::map<FaultCode, std::vector<int>> faults = wave.get_fault_code();
```

The returned map associates each `FaultCode` with a list of relevant indices, if applicable. The meaning of these indices depends on the fault context. For example:

- for motion-related faults, the indices may refer to joint indices

- for tactile-related faults, the indices may refer to tactile channels
- for some system-level faults, no specific index may apply, and the associated vector may be empty

For the complete list of fault codes and their meanings, refer to Appendix 4 Fault Codes.

Recommended practice:

- check reported fault codes when device behavior is abnormal
- interpret the associated indices according to the subsystem involved
- use the reported indices to narrow down motion, tactile, or other subsystem issues when available

3.3 Common Problems

3.3.1 No device discovered

Check the following:

- the device is powered on
- the host and device are on the same subnet
- firewall or endpoint security software is not blocking required traffic
- heartbeat packets can reach the host

3.3.2 Connection fails

Check the following:

- the serial number is correct
- the selected hand side uniquely identifies one device
- the device is reachable on the network
- the SDK runtime library is correctly installed

3.3.3 The device does not respond to motion commands

Check the following:

- `is_hand_ready()` returns `true`

- control source is set to SDK
- the correct control mode has been selected
- the device is enabled
- command vectors use the correct DOF order and length

3.3.4 Tactile data is not received

Check the following:

- the device supports tactile sensing
- tactile is enabled and ready
- the host port required for tactile reception is available
- the configured host IP and target port are correct
- another application is not occupying the tactile port

3.3.5 Port occupied or tactile bind failure

Use the tactile port recovery helpers:

- `retry_tactile_alternate_port()`
- `get_next_available_tactile_port()`
- `bind_tactile_port()`

3.3.6 Device becomes unreachable after IP change

Check the following:

- the device has completed reboot
- the host computer is on the same subnet as the new device IP
- the application reconnects after the reboot

3.3.7 Recommended Debugging Practices

- always log `Error.code` and `Error.message`
- wait for readiness before issuing commands
- log discovered devices and their IP addresses during startup
- validate command vector sizes before sending motion commands

- keep tactile callbacks lightweight
- disconnect cleanly before application exit

Appendix

1. Joint DOF Ordering

For vectors that involve all joint DOFs, the elements are ordered as follows:

INDEX	JOINT NAME	FINGER	TYPE
0	Thumb CMC FE	Thumb	FE
1	Thumb CMC AA	Thumb	AA
2	Thumb MCP FE	Thumb	FE
3	Thumb MCP AA	Thumb	AA
4	Thumb IP	Thumb	FE
5	Index MCP FE	Index	FE
6	Index MCP AA	Index	AA
7	Index PIP	Index	FE
8	Index DIP	Index	FE
9	Middle MCP FE	Middle	FE
10	Middle MCP AA	Middle	AA
11	Middle PIP	Middle	FE
12	Middle DIP	Middle	FE
13	Ring MCP FE	Ring	FE
14	Ring MCP AA	Ring	AA
15	Ring PIP	Ring	FE
16	Ring DIP	Ring	FE
17	Pinky CMC	Pinky	FE
18	Pinky MCP FE	Pinky	FE
19	Pinky MCP AA	Pinky	AA
20	Pinky PIP	Pinky	FE
21	Pinky DIP	Pinky	FE

2. Tactile Channel Mapping

The tactile channel ranges are organized by hand side as follows:

CHANNEL	HAND SIDE	FINGER
0	Right hand	Pinky
1	Right hand	Ring
2	Right hand	Middle
3	Right hand	Index
4	Right hand	Thumb
5	Left hand	Pinky
6	Left hand	Ring
7	Left hand	Middle
8	Left hand	Index
9	Left hand	Thumb

Examples:

- wave.fetch_tactile_frame(0, 1.0) means fetching tactile data from right-hand channel 0, with a maximum wait time of 1.0 second.
- wave.fetch_tactile_frame(5, 1.0) means fetching tactile data from left-hand channel 5, with a maximum wait time of 1.0 second.

Notes:

- Applications should use the correct tactile channel for the target finger or sensing location.

3. ErrorCode (API Level)

The following table lists the result codes returned by API functions. These codes primarily focus on the success of communication, data integrity, and protocol validation for

individual instructions.

ERRORCODE	LEVEL	CATEGORY	DESCRIPTION	ACTION
0×00	Info	Success	Operation successful	—
0×01	Error	Input	Invalid parameter	Check API arguments (type, length, range) and retry
0×02	Error	Network	Connection failed	Check power, IP/port, cables, and firewall; retry
0×03	Error	Network	No data returned	Ensure device is online and active; check command expectations; retry or increase timeout
0×04	Critical	System	Memory insufficient	Reduce request load; retry; restart if needed
0×05	Warning	Version	Unsupported command	Check SDK/firmware compatibility; update or avoid API
0×06	Critical	System	Internal communication failure	Restart device; if persistent, collect logs and contact support
0×07	Error	System	Internal system error	Record logs; restart and retry; contact support if persistent
0×08	Error	System	Flash write failure	Ensure device allows parameter saving; retry or contact support
0×09	Error	Protocol	Invalid header	Check cable integrity and protocol version
0×0A	Error	Protocol	Payload too large	Reduce data size
0×0B	Error	Protocol	Incomplete payload	Check network stability
0×0C	Error	Protocol	Checksum failed	Check EMI or cable quality
0×0D	Warning	State	Operation not allowed	Check device state (e.g., busy or protected)

ERRORCODE	LEVEL	CATEGORY	DESCRIPTION	ACTION
0×10	Error	Network	TCP send failed	Check network link and host TCP/IP
0×11	Error	Network	TCP receive failed	Check network load and device response
0×12	Error	Protocol	Malformed response	Check SDK/firmware compatibility

4. FaultCode (Device Level)

The following table provides detailed descriptions of the device's internal health and safety status. These codes are reported by the firmware to indicate system warnings, hardware failures, or environmental protection triggers.

FAULTCODE	LEVEL	CATEGORY	DESCRIPTION	ACTION
0×0101	Warning	System	CPU overload	Contact support
0×0181	Critical	Calibration	Zero-position error	Contact support
0×0182	Critical	Calibration	Calibration load failed	Contact support
0×0240	Error	Power	Undervoltage	Check power supply; contact support if persistent
0×0280	Critical	Power	Overcurrent	Check power supply; power cycle; contact support if persistent
0×0281	Critical	Power	Overvoltage	Check power supply; power cycle; contact support if persistent
0×0302	Warning	Motor	Abnormal motor angle	Power cycle; contact support if persistent
0×0340	Error	Motor	Motor software error	Contact support
0×0341	Error	Motor	Motor overvoltage protection	Contact support
0×0342	Error	Motor	Motor undervoltage protection	Contact support
0×0343	Error	Motor	Motor overtemperature protection	Stop operation; allow cooling
0×0344	Error	Motor	Motor startup error	Contact support
0×0345	Error	Motor	Speed feedback error	Contact support
0×0348	Error	Motor	Encoder error (high-speed)	Contact support

FAULTCODE	LEVEL	CATEGORY	DESCRIPTION	ACTION
0x0349	Error	Motor	Encoder error (low-speed)	Contact support
0x034C	Error	Motor	Motor alignment error	Contact support
0x0351	Error	Motor	Motor communication abnormal/offline	Power cycle; contact support if persistent
0x0353	Error	Motor	Motor angle error	Contact support
0x0401	Warning	Force	Possible sensor damage	Contact support
0x0402	Warning	Force	Force/torque limit exceeded	Check for external load/collision; contact support if persistent
0x0440	Error	Calibration	Calibration parameter error	Contact support
0x0441	Error	Force	Sensor sampling error	Contact support
0x0501	Warning	Motion	Joint angle limit exceeded	Check control mode; ignore if in Zero-force manual movement
0x0503	Warning	Motion	Joint torque limit exceeded	No action if under load; otherwise contact support
0x0601	Warning	Thermal	70–80°C	Stop operation; allow cooling
0x0602	Warning	Thermal	80–90°C	Stop operation; allow cooling
0x0603	Warning	Thermal	>90°C	Stop immediately; allow cooling
0x0640	Error	Thermal	Cooling system error	Contact support
0x0701	Warning	Tactile	Config read error	Retry; power cycle if persistent

FAULTCODE	LEVEL	CATEGORY	DESCRIPTION	ACTION
0×0702	Warning	Tactile	Config write error	Retry; power cycle if persistent
0×0703	Warning	Tactile	Flash read error	Retry; power cycle if persistent
0×0704	Warning	Tactile	Flash write error	Retry; power cycle if persistent
0×0705	Warning	Tactile	Config parse error	Resend config; power cycle if persistent
0×0706	Warning	Tactile	Invalid config	Check config parameters
0×0740	Error	Tactile	Sensor not detected	Check connection; power cycle; contact support if persistent
0×0741	Error	Tactile	MCU communication error	Check connection; power cycle; contact support if persistent
0×0742	Error	Tactile	Missing NPU model	Contact support
0×0743	Error	Tactile	I2C communication error	Check cable and EMI
0×0800	Error	System	Lifecycle config error	Contact support
0×0801	Error	System	Device connection error	Check connection; retry; contact support if persistent
0×0802	Warning	System	Running error	Restart system; contact support if persistent
0×0803	Warning	System	Stop error	Restart system; contact support if persistent
0×0811	Warning	System	Hand construction error	Contact support
0×0812	Warning	System	Initialization error	Restart system; contact support if persistent

FAULTCODE	LEVEL	CATEGORY	DESCRIPTION	ACTION
0x0815	Warning	System	Touch construction error	Restart system; contact support if persistent
0x0816	Warning	System	Touch initialization error	Restart system; contact support if persistent
0x0817	Warning	Network	Invalid host IP	Check IP/subnet settings
0x0818	Warning	Network	Port occupied	Check port usage (e.g., netstat)
0x0819	Warning	Network	Host listen failure	Check network and port configuration
0x0821	Warning	Function	Control mode not supported	Check control mode setting
0x0822	Warning	Network	Heartbeat send failed	Check connections and cables; contact support if persistent
0x0830	Warning	Version	Hardware version mismatch	Use compatible SDK/firmware
0x0831	Warning	Version	Firmware mismatch	Use compatible SDK/firmware
0x0832	Warning	Version	Version config missing	Update SDK/firmware

Teleoperation

MANUS Metagloves Pro

Coming Soon

Support

Troubleshooting

This section helps you quickly diagnose and resolve common issues when using Sharpa Wave. Follow the steps below based on your observed symptom.

1. Quick Diagnosis

When an issue occurs, follow this order:

1. Check the indicator light on the back of the hand
2. Check whether the device appears in Sharpa Pilot
3. Check ErrorCode (API / communication issues) in Sharpa Pilot
4. Check FaultCode (hardware / safety issues) in Sharpa Pilot

2. Indicator Light Guide

The back-of-hand indicator provides the fastest way to assess system status.

STATUS	MEANING	ACTION
Breathing cyan	Device initialized and ready	No action required
Solid white	Device enabled and operating	No action required
Solid yellow	Warning present	Check FaultCode
Solid red	Error present	Stop operation and check FaultCode
Flashing red	Critical fault	Stop operation immediately and check FaultCode

			
Cyan	White	Yellow	Red

3. Device Issues

3.1 Device Not Discovered

Symptom
Sharpa Pilot cannot detect the device.

Steps to check

1. Confirm the device has completed startup (wait ~20 seconds)
2. Check power connection and use the official power adapter
3. Ensure that the host PC network settings are configured according to Section 5.1, Default Network Settings.
4. Temporarily disable firewall on the host PC
5. Connect the device directly to the PC (avoid switches/routers)
6. Check Ethernet cable and port
7. If IP is unknown, restore factory network settings (press rear button 3 times consecutively)

3.2 Device Detected but Cannot Be Controlled

Symptom
Device appears in Sharpa Pilot but does not respond to commands.

Steps to check

1. Check the Control Source, and Control Mode in Sharpa Pilot.
2. Look for warning banners in the UI and resolve them.
3. Check the joint data frequency (about 500 Hz is expected).

4. If the frequency is 0 Hz, the device may be offline

Check firewall settings:

- Temporarily disable firewall and test:

```
TEXT

sudo ufw disable
```

5. Restart Sharpa Pilot and try again.

4. Motion Issues

4.1 Single Joint Abnormal

Symptom

nA joint does not move or behaves abnormally.

Steps to check

1. Check FaultCode for motor-related errors

2. Power cycle the device
3. If the issue persists, contact Sharpa Support

4.2 Homing / Zero Position Incorrect

Symptom
Joint does not return to correct zero position.

Steps to check

1. Check the current angle in Sharpa Pilot.
2. If the angle is near 0° but the physical position is incorrect, recalibrate the joints.
3. If the angle is far from 0°, see Section 4.1, Single Joint Abnormal.

4.3 Large Tracking Error

Symptom
Actual joint motion does not follow commands.

Steps to check

1. Ensure that the command frequency is within the supported range.
2. Avoid large step changes greater than 20° per frame.
3. Check for hardware issues if only one joint is affected.
4. If multiple joints are affected, contact Sharpa Support.

4.4 Weak Grip Force

Symptom
Grasping force is insufficient.

Steps to check

1. Increase the joint current coefficient in Sharpa Pilot within the range of 0.5 to 1.0.
2. Check for thermal warnings (see Section 6.2).
3. Allow the device to cool down if overheating is present.

5. Tactile Sensing Issues

5.1 No Tactile Data

Symptom

No response in tactile visualization or SDK data.

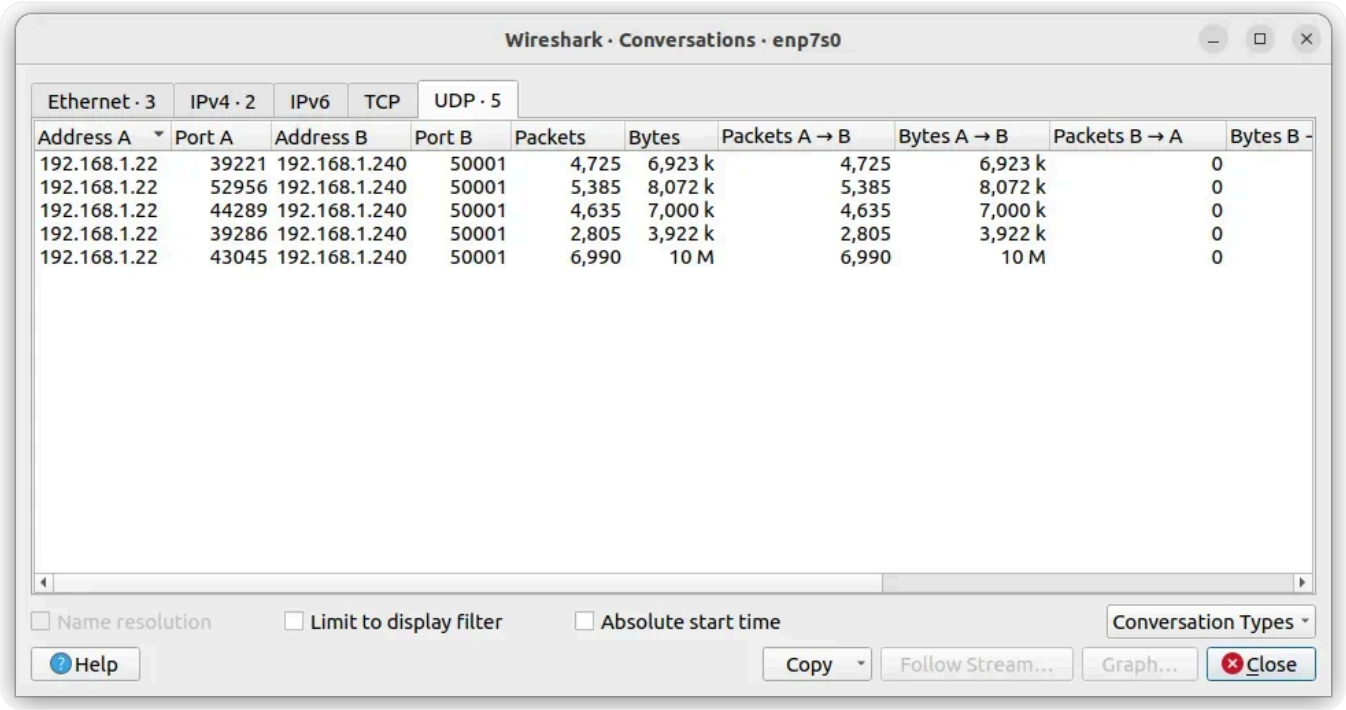
	
no response	normal response

Basic checks

1. Check the FaultCode in Sharpa Pilot.
2. Verify the network connection by pinging the device IP address.
3. Confirm that the IP address and subnet settings are correct.

Advanced checks

1. Use Wireshark to verify the UDP data stream on port 50001.



2. Confirm that all five tactile sensors are transmitting.

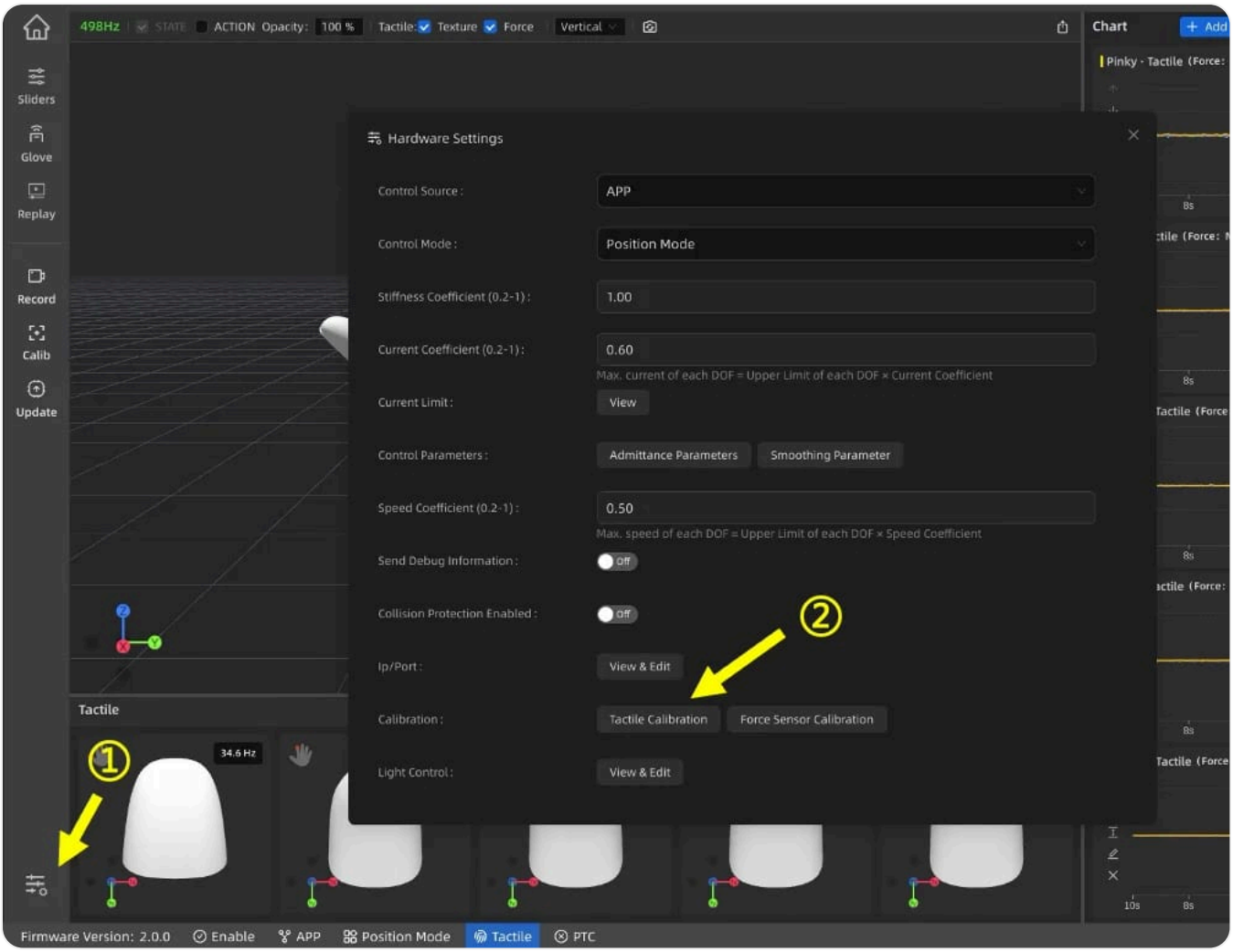
5.2 Abnormal Tactile Data

Symptom

Non-zero force or deformation without contact.

Steps to check

1. Perform tactile calibration in Sharpa Pilot



2. Power cycle the device.
3. Ensure that no contact occurs within the first 20 seconds after startup.

5.3 Low Frame Rate

Symptom

Tactile visualization is slow or <30 Hz.

Steps to check

1. Verify that the Ethernet link speed is Gigabit.
2. Check the Ethernet speed by running:

```
TEXT

ethtool <interface>
```

3. Confirm that the reported speed is 1000Mb/s.
4. Check the device temperature and cooling condition

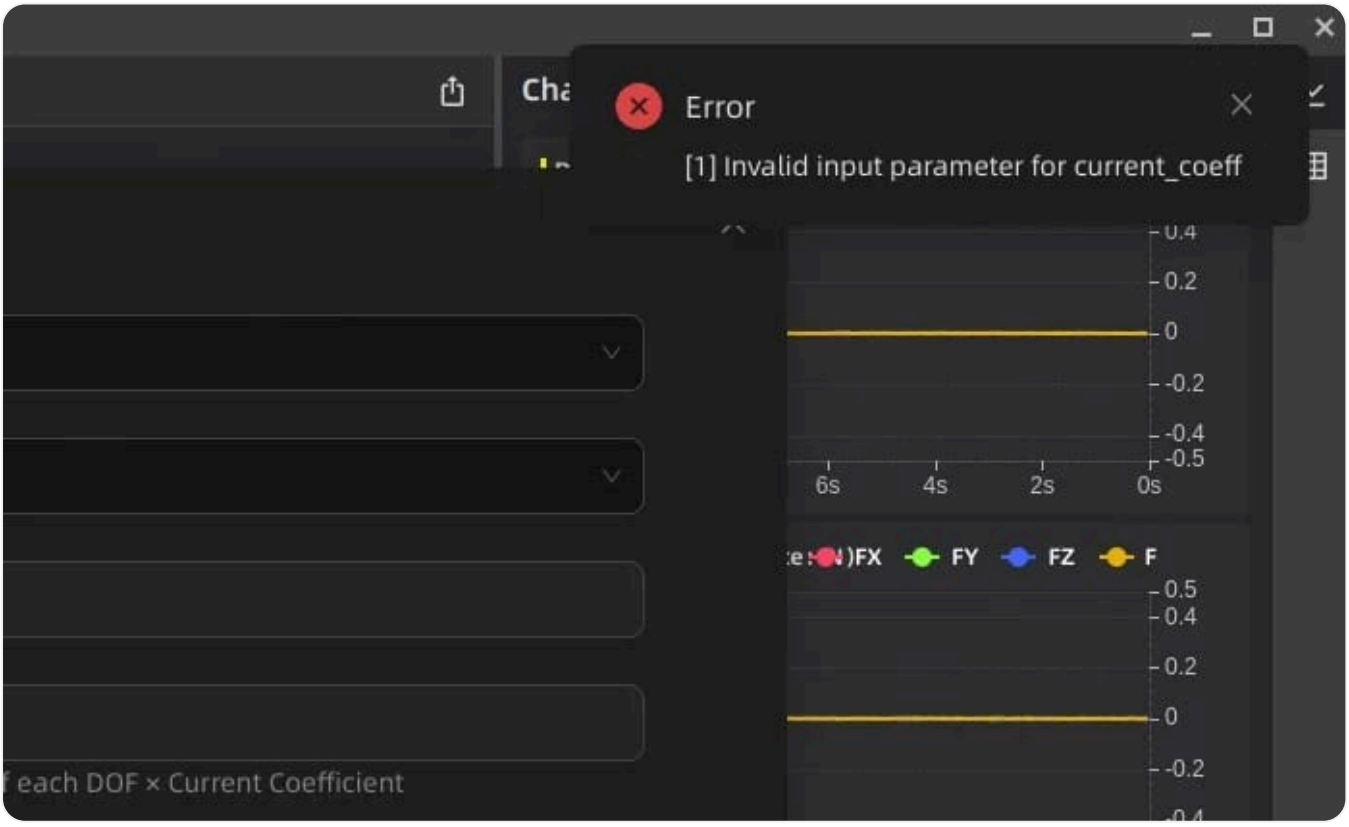
6. ErrorCode vs FaultCode

6.1 ErrorCode (API Level)

- Indicates command success or communication issues
- Example causes:
  - Invalid parameters
  - Network failure
  - Protocol mismatch

Use ErrorCode when:

- Commands fail
- SDK/API returns errors
- Errorcodes are reported in Sharpa Pilot



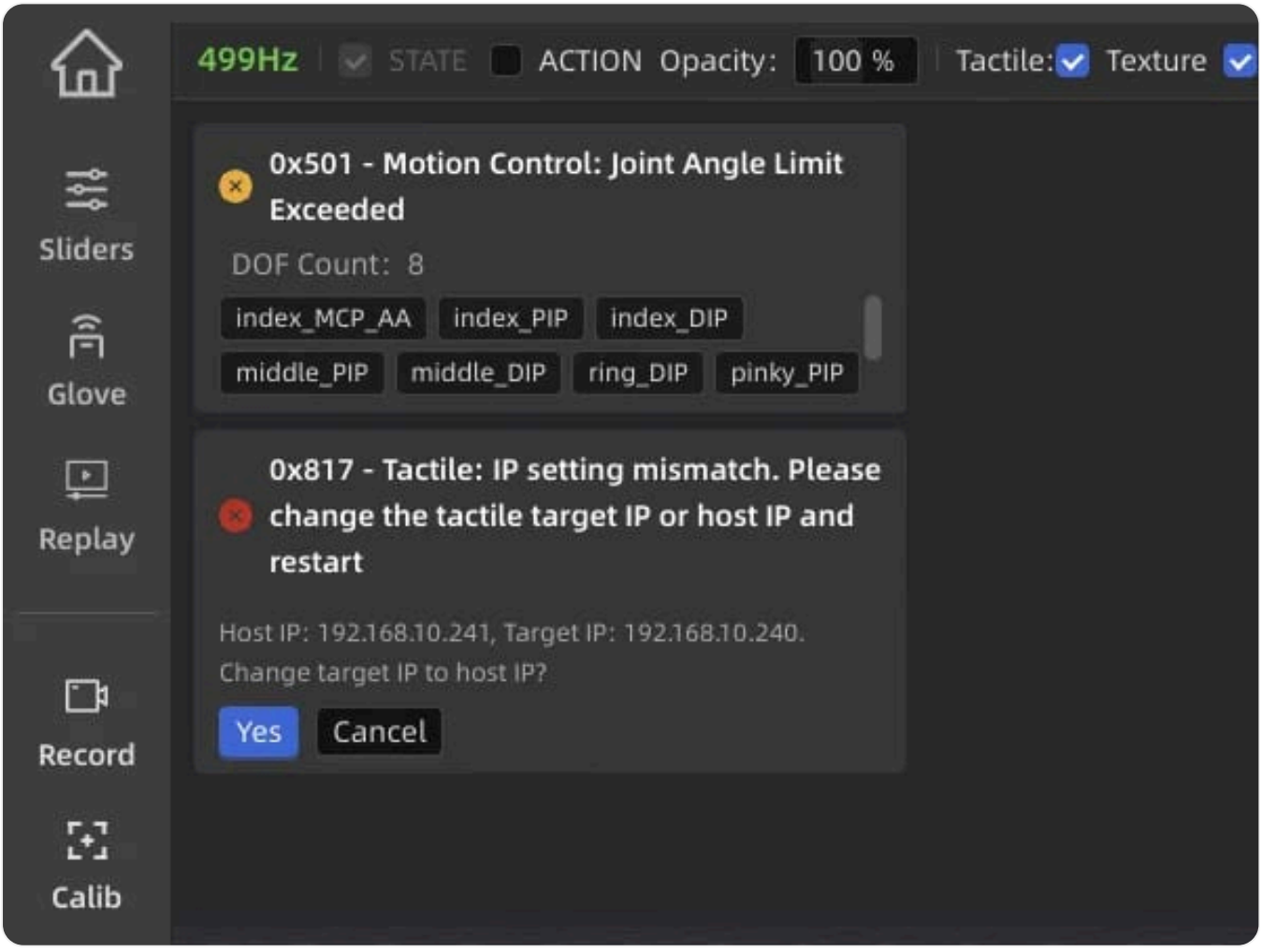
For the full list of ErrorCodes, refer to the Sharpa Wave SDK.

6.2 FaultCode (Device Level)

- Indicates hardware or safety status
- Example causes:
  - Overcurrent / overvoltage
  - Motor or sensor failure
  - Overtemperature

Use FaultCode when:

- Indicator light is yellow/red
- Device behaves abnormally
- Faultcodes are reported in Sharpa Pilot



FaultCode in Sharpa Pilot

For the full list of FaultCodes, refer to the Sharpa Wave SDK .

7. General Recovery Actions

Use the following actions to resolve most common issues.

7.1 Quick Recovery Checklist

Try the following steps in order:

- Retry the operation
- Restart Sharpa Pilot
- Power cycle the device
- Recalibrate joints or tactile sensors
- Check cables and network configuration

7.2 Button-Based Reset & Restart

The back button supports quick restart and configuration reset.

Actions

- **Press 2 times:** Restart the device
- **Press 3 times:** Reset configuration and restart

Tip:
A natural “double-click” or “triple-click” rhythm works well.

Configuration Reset Scope

A configuration reset restores the following settings:

1. Network Settings

See Section 5.1, Default Network Settings.

2. Control and System Parameters

- Control mode
- Control source
- Motor current / speed coefficients

- Admittance control parameters
- MIT control parameters

Notes

- If the presses are too slow, the action may not be triggered
- Reset will overwrite current network and control configurations
- Use reset only when necessary (e.g., device not reachable)

Contact Support

Contact support if:

- A fault persists after 2 power cycles
- Critical faults (flashing red) occur
- Motor / encoder / sensor errors are reported
- Calibration fails or parameters cannot be loaded
- Unknown or undocumented fault codes appear

Support: support@sharpa.com

Best Practices for Reliable Use

Coming Soon

Legal

1. Legal Notice

This notice applies to the use of this product, unless otherwise specified in a separate written agreement.

Users are advised to thoroughly read and understand this legal notice before using the product. Continued use of the product constitutes acceptance of and agreement to be bound by the following terms.

1. No guarantees

Unless otherwise agreed in writing, this product is provided “as is”, without any express or implied warranties, including but not limited to warranties of merchantability, fitness for a particular purpose, uninterrupted operation, or performance.

The user understands and agrees that, as this product is under continuous development, Sharpa reserves the right to modify its functionality, structure, or performance at any time, and makes no guarantees regarding the product's stability or expected results in any specific application scenario.

Confirm product development maturity with Sharpa if needed.

2. Prohibited uses

Unless otherwise explicitly authorized in writing, the user shall not use this product for any of the following purposes, including but not limited to:

- Critical mission systems (such as the main control systems in industrial automation)
- Traffic safety control systems (such as autonomous driving decision-making modules)
- Medical devices or life-support systems
- Any other scenarios where failure could result in personal injury or death, significant property damage, or risks to public safety

3. Limitation of liability

To the maximum extent permitted by applicable law, Sharpa shall not be liable for any direct, indirect, incidental, special, or consequential losses (including but not limited to

data loss, business interruption, or loss of profits) arising from the use or inability to use this product, provided that Sharpa has not engaged in gross negligence or willful misconduct.

4. Export control and sanctions compliance

This product and its related technologies may be subject to applicable export control and sanctions regulations of relevant jurisdictions. The user shall comply with all such regulations and shall not export, re-export, or transfer this product to any sanctioned or embargoed countries, regions, individuals, entities, military end-users, or law enforcement agencies. Users shall also refrain from using this product for military end use or any other prohibited end-uses.

5. Intellectual property statement

This product, including its software, firmware, design, appearance, and technical documentation, is the property of Sharpa or its licensors, and is protected by copyright law, patent law, and other applicable laws. Without prior written permission from Sharpa, users may not reproduce, modify, distribute, decompile, reverse engineer, or use it for commercial purposes in any form.

Sharpa Wave and its logo are registered trademarks of Sharpa. All other trademarks, service marks, and company names in this manual or on Sharpa's official website are the property of their respective owners.

6. Warranty terms

Sharpa warrants that its products and accessories, as manufactured and delivered, are free from defects in materials and workmanship, and conform to the specifications agreed in writing. Unless otherwise specified in writing, Sharpa provides a warranty period of twelve (12) months, starting from the date of delivery.

During the warranty period, if you discover any defects in the product, please contact Sharpa technical support in writing. Upon confirmation that the issue falls within the scope of the warranty, Sharpa will repair or replace the product free of charge.

7. Exclusions

The following cases are NOT covered by Sharpa's warranty service, and any resulting responsibilities and costs shall be borne by the user:

- Disassembly, modification, or repair of the product without Sharpa’s authorization, whether by the user or a third party
- Failure to reasonably store, install, operate, or maintain the product in accordance with industry best practices and the relevant instructions provided in this manual, resulting in malfunctions
- Malfunctions caused by exceeding the inherent operating limits of the product or its accessories

8. Revision notice

Sharpa reserves the right to update, modify, or replace the contents of this manual, and to release the latest version without prior notice.
It is recommended that users contact Sharpa to obtain the latest version of this manual before using the product. Unless otherwise agreed, the latest version shall prevail.

2. Repair

Do not disassemble, repair, modify, or alter the product by yourself or through any third party without Sharpa’s explicit written consent, except for replacing fingertip accessories.

Unauthorized disassembly, repair, modification, or alteration:

- may result in product damage, property loss, and/or personal injury;
- constitutes a breach of warranty.

For repair-related support, contact Sharpa or an authorized Sharpa service provider.

2.1 When Operating the Product

- Follow all instructions in this manual.
- Failure to comply may result in product damage, property loss, personal injury, and/or a breach of warranty.
- Seek technical support when needed at support@sharpa.com.

2.2 After Upgrading the Product

After the product is upgraded, obtain the latest version of this manual.

Do one of the following:

- Contact your Sharpa sales representative.
- Contact Sharpa technical support at support@sharpa.com.

2.3 When Integrating the Product into Your Application

- Provide this manual, or access to it, to the intended users of your product.
- This dexterous hand is intended to be used as a component of an end product.
- The end-product supplier is responsible for:
  - assessing the risks of use in accordance with applicable standards;
  - informing intended users of safety-related information.

North

Coming Soon