DOBOT

Hexplorer User Guide

Original Instructions

Issue: V1.0

Date: 2025-10-21

SHENZHEN DOBOT CORP LTD | China

Disclaimer

To the maximum extent permitted by applicable law, the products described (including its hardware, software and firmware, etc.) in this document are provided **AS IS**, which may have flaws, errors or faults. Dobot makes no warranties of any kind, express or implied, including but not limited to, merchantability, satisfaction of quality, fitness for a particular purpose and non-infringement of third party rights. In no event will Dobot be liable for any special, incidental, consequential or indirect damages resulting from the use of our products and documents.

Before using our product, please thoroughly read and understand the contents of this document and related technical documents that are published online, to ensure that the robot is used on the premise of fully understanding the robot and related knowledge. Please use this document with technical guidance from professionals. Even if follow this document or any other related instructions, damages or losses may happen in the using process. Dobot shall not be considered as a guarantee regarding all security information contained in this document.

The user has the responsibility to make sure following the relevant practical laws and regulations of the country, in order that there is no significant danger in the use of the robot.

SHENZHEN DOBOT CORP LTD

Address: Room 1003, Building 2, Chongwen Garden, Nanshan iPark, Liuxian Blvd, Nanshan District, Shenzhen, Guangdong Province, China

Website: www.dobot-robots.com

DOBOT Europe GmbH

Address: Waldeckerstraße 11, 64546 Mörfelden-Walldorf

Contact: Lang Xulin, lang@dobot-robots.com

Preface

This manual is intended to guide users in the operation and secondary development of the Hexplorer hexapod bionic robot platform.

Intended audience

This document is intended for:

- Customer
- Sales Engineer
- Installation and Commissioning Engineer
- Technical Support Engineer

Revision history

Date	Version	Revised content
2025-10-21	V1.0	The first release

Symbol conventions

The symbols that may be found in this document are defined as follows.

Symbol	Description
DANGER	Indicates a hazard with a high level of risk which, if not avoided, could result in death or serious injury.
WARNING	Indicates a hazard with a medium level or low level of risk which, if not avoided, could result in minor or moderate injury, robot arm damage.
NOTICE	Indicates a potentially hazardous situation which, if not avoided, could result in robot arm damage, data loss, or unanticipated result.
NOTE	Provides additional information to emphasize or supplement important points in the main text.

Contents

1. Product Overview

1.1 Overall View of the Robot

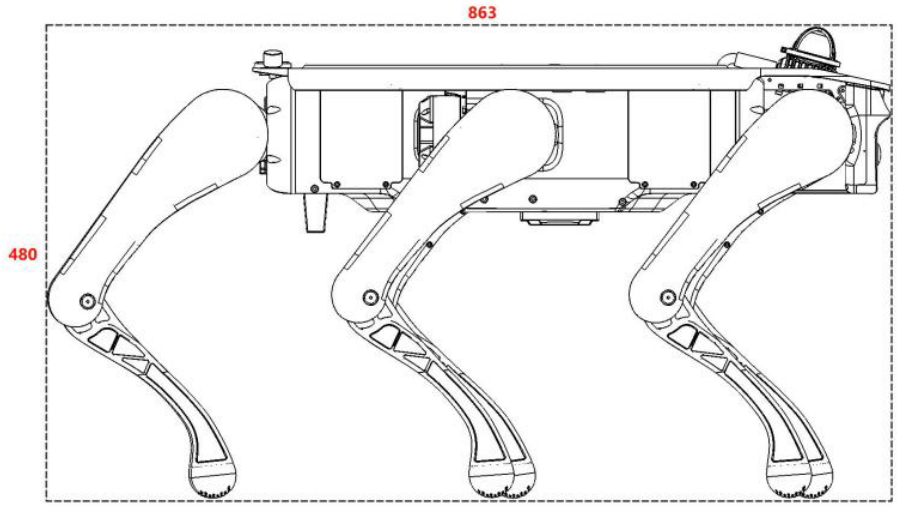


Figure 1.1 Side View

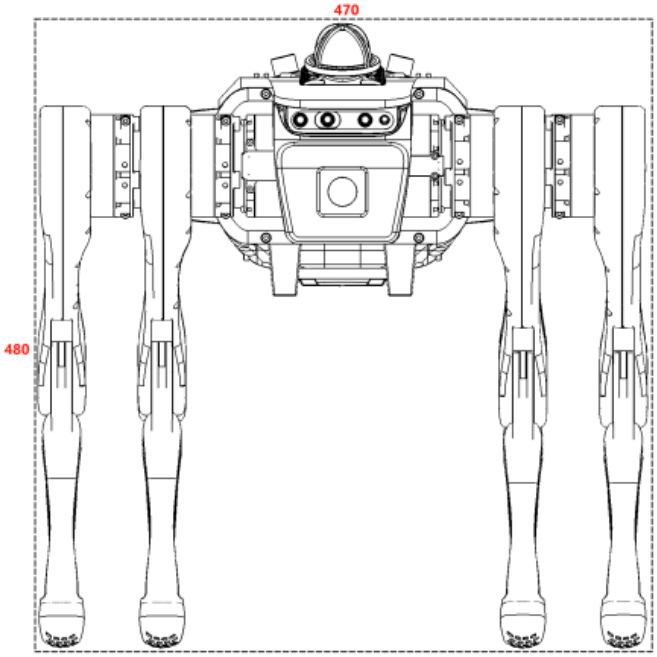
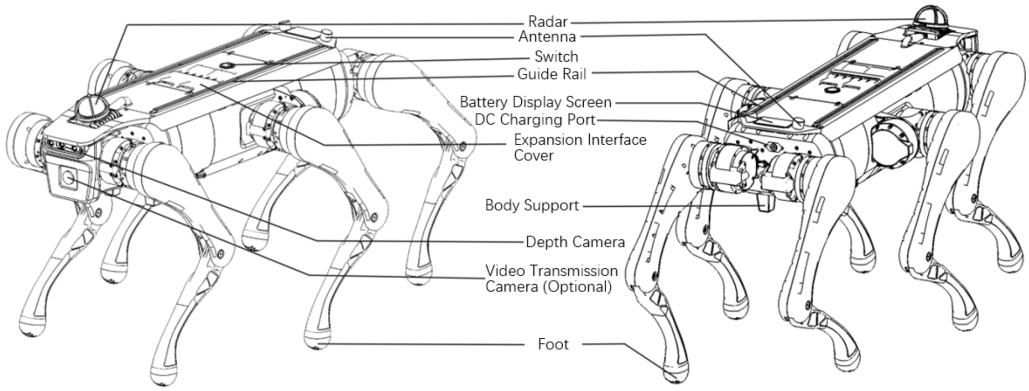


Figure 1.2 Front View

1.2 Parts Description



1.3 Specifications

Parameter Name	Numbers
Model	DT-HEX-6R010-L
Dimension (Standing)	86.3cm*47cm*48cm
Dimension (Folded)	73.4cm*47cm*24.4cm
Dimension (Package)	81.5cm*54.3cm*35cm
Degree of Freedom	18
Weight	Robot body: ~20 kg, With packaging: ~30 kg
Maximum walking speed	1.8m/s
Maximum Obstacle Height	20cm
Maximum Slope Angle Adaption	40°
Load Capacity	Rated: 10 kg, Maximum: 15 kg

Parameter Name	Numbers
Joint Motion Range	Front & Middle Hip: -60° ~ 180°
	Rear Hip: -60° ~ 239°
	Lower Leg: -30° ~ -160°
	Body: ±38°
Battery Capacity	504wh
Endurance Time	2.5 hours
Charging Time	1.5 hours
Operating Environment	Temperature: 0°C – 40°C Humidity: 25%–85%, non-condensing
Input	100 - 240VAC, 50 - 60HZ
Output	48V
Rated Power	240W
Full Load Current	5A
Motor Peak Torque	33 Nm (Knee Joint)
Noise Level	Motion noise ≤ 55 dB
	Test environment: indoor thin carpet, sound meter approx. 1 meter distance.

NOTICE

Users must verify the quantity and condition of all items inside the package upon unboxing. The images shown in this manual are for reference only; the actual product shall prevail. Configurations may vary slightly between different models, and included accessories should be based on the actual items of the purchased model.

2. Functional Diagram

2.1 System Architecture Information

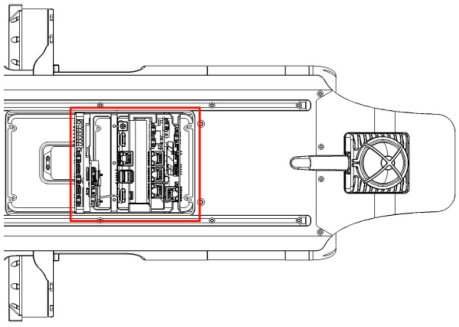
Robot Control Platform	Intel Mini PC, x86_64 architecture
	CPU: Core i7/i5, 6 cores, 16 threads, nominal frequency 1.2 GHz
	RAM: 16GB; ROM：95GB(Available);
Computing Platform	Jetson Orin Nano, ARM architecture
	CPU: 6x A78, RAM: 16 GB; ROM: 128 GB; Computing power: 40 TOPS
Wireless Controller	Thor G30s, Bluetooth 5.0, communication range 10 m, frequency 2.4 GHz
Speaker	Supported
LiDAR	Livox Mid360
Depth Camera (RGB-D)	Realsense D435

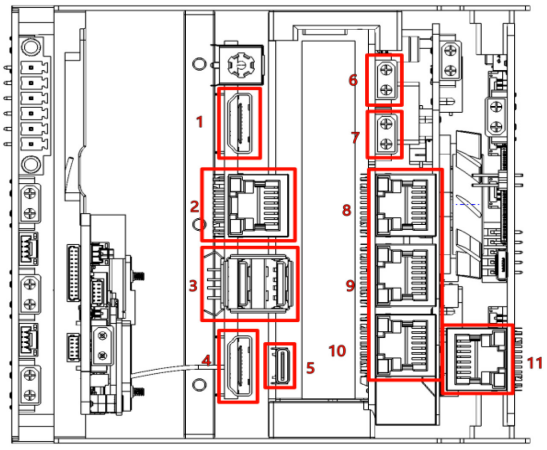
2.2 Function introduction

The Hexplorer hexapod bionic robot is designed around powerful performance and exceptional terrain adaptability, offering dual value for both research/education and industrial applications. It maintains a stable and agile posture across diverse environments and possesses locomotion capabilities suited for complex terrains. Equipped with a complete SDK and development documentation, it supports secondary development and functional expansion.At the hardware level, the Hexplorer robot integrates sensors such as the Livox Mid360 LiDAR, Realsense D435 depth camera, and high-precision IMU. Its hardware design also provides ample expansion interfaces, allowing users to mount additional sensors, extend functionalities, or perform customized development according to their needs. Based on this platform, users can implement advanced features, including but not limited to:

- Walking on complex terrain
- Autonomous navigation
- Obstacle avoidance
- Human detection
- Target following

The Hexplorer platform provides developers with comprehensive support for secondary development and offers a wide range of onboard hardware interfaces. The locations of these interfaces and their corresponding descriptions are shown below:





Number	Interface Name
1	Nano-DP
2	Nano-TypeA
3	Nano-ETH
4	Nano-TypeC
1	MiniPC-HDMI
2	MiniPC-ETH
3	MiniPC-TypeA
4	MiniPC-HDMI
5	MiniPC-TypeC
6	Power-19V2A(output)
7	Power-19V2A(output)
8-10	Switch-RJ45(1000M)
11	CommunicationBoard-RJ45

The Hexplorer body provides interfaces for the Jetson Nano, Intel Mini PC, power output, and high-speed communication ports, supporting a variety of peripheral expansions.

External Expansion	Supports collaborative robotic arm expansion (*future support)
	Supports long-distance remote control and video transmission (*future support)
	Supports wireless automatic charging (*future support)
Power & Communication Expansion	Power Supply Interface ×1: 19V/2A (for connecting external PC) Communication Expansion: USB Type-C ×1, Ethernet ×1

3. Transportation, Handling, and Storage

3.1 Transportation

The dedicated transport case for the robot weighs approximately 30 kg in total, with the robot bodyaccounting for about 20 kg and the remainder comprising packaging materials and accessories. To ensure transport safety, please make sure that all latches on the transport case are fully secured before handling or loading. During transportation, avoid severe vibrations, overturning, or impacts to prevent damage to the robot's internal precision components. For long-distance transport, it is recommended to use cushioning pads or securing devices to enhance shock resistance and protection.

When packing the robot, place it in the transport case according to the prescribed storage posture: the six legs should be folded inward, and the robot body should fit closely against the cushioning lining to prevent movement or collisions during transport. The front and side packing illustrations shown below can serve as a reference; however, the actual packing should follow the design of the transport case lining.

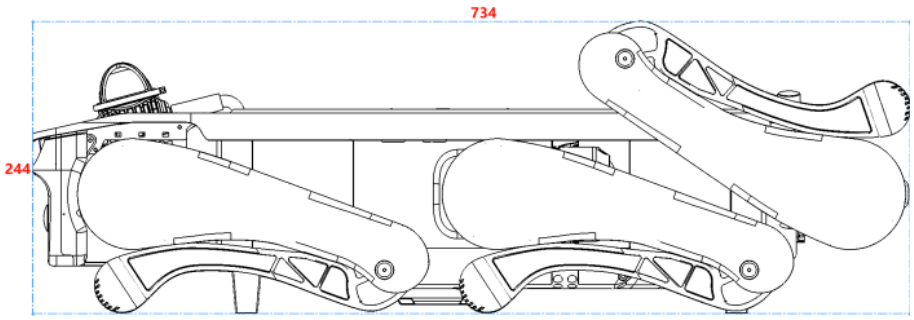


Figure 3.1 Storage Posture – Side View

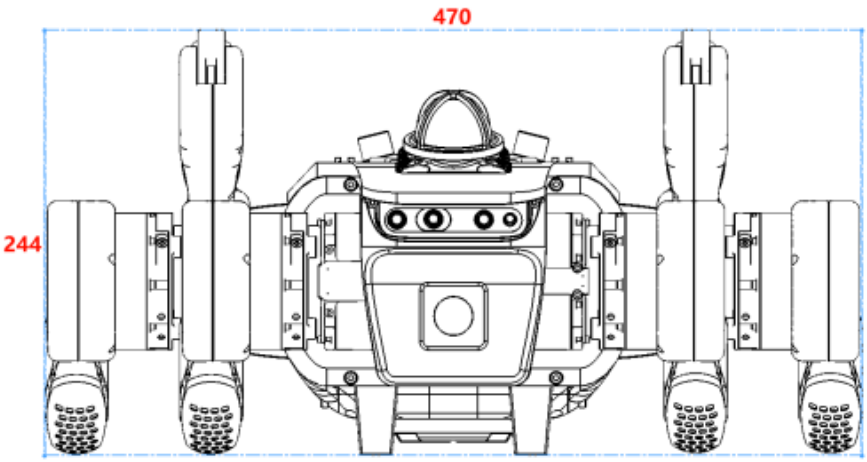


Figure 3.2 Storage Posture – Front View

3.2 Handling

The hexapod robot should only be handled when the power is off, or when the robot is powered on and in damping mode (all joints act like dampers). Handling requires two people working together. Use the handle positions indicated in the diagram below (one

person holds the red-colored lower legs, the other holds the blue-colored lower legs) for lifting and moving the robot. During handling, take care to avoid pinching or entangling clothing.

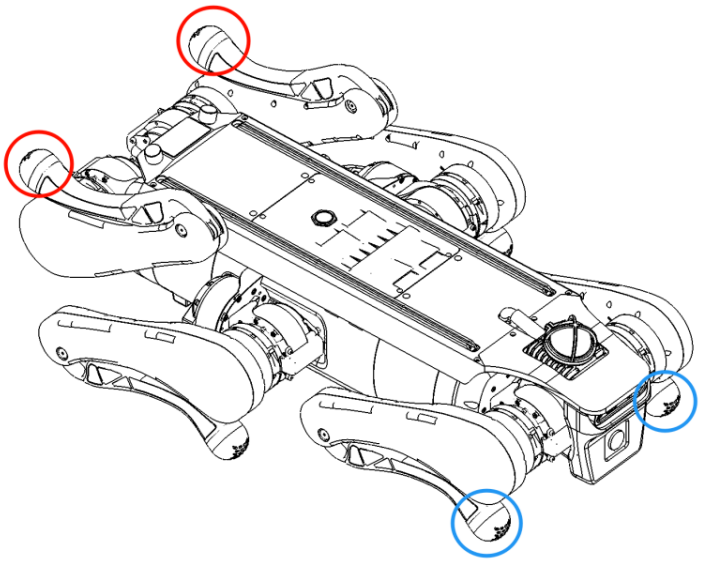


Figure 3.3 Grip and Pinch-Avoidance Locations

3.3 Storage

WARNING

The robot power must be turned off. Storage temperature: -20°C to 40°C; relative humidity: 30%–70%. Do not pour water or any other liquids onto the robot. Do not place any objects within the joint rotation range. It is recommended to designate a safe, tidy area specifically for robot storage. Use the transport case designed for the hexapod robot for storage. For long-term storage, it is recommended to charge the robot once every three months to maintain battery activity.

4. How to charge

4.1 Robot Body

Charge the robot using the 58.8V lithium battery by plugging the charger into the blue charging port (as shown in the figure below). The charger's red light indicates charging, and the green light indicates a full charge. The robot's LCD screen will also display the remaining battery level. For the location of the DC charging port, please refer to "Product Overview – Component Description."

Figure 4.1 DC Charging Port

Figure 4.2 Robot Charging

4.2 Controller

Charge the robot using the 5V-2A Type-C charging port. A flashing red light indicates charging, and the light off indicates a full charge.

4.3 External Power Supply

Plug the charger into the blue charging port (as shown above), then press the power button to turn on the robot. Note that if motion control of the robot is required, the charging port should be secured to prevent the charging cable from disconnecting.

5. How to Use Your Robot

5.1 Pre-Use Preparation

NOTICE

Please pay attention to the following points to ensure safe and effective operation.

- Avoid entering or staying in the robot's operational area without a necessary purpose.
- Keep the robot within your line of sight while operating.
- Avoid approaching a moving robot, especially its active joints or other components.
- Do not attempt to interfere with the robot's motion, particularly when it is unstable.
- Climbing on the robot is strictly prohibited.
- Other environment-related precautions:
- Maintain a safety distance of at least 2 meters from people, water, open flames, etc., while the robot is operating.
- Avoid operating the robot in environments with severe WiFi interference, and turn off other WiFi devices if necessary.
- Do not come into contact with hazardous substances.
- Do not directly or indirectly touch live components.
- Do not use the robot in potentially explosive environments.
- Before operating the robot in environments with strong electromagnetic interference, consult after-sales support.

Pre-power-on checks

1) **Robot Integrity:** Check whether the robot's exterior is intact and whether the mechanical limits are undamaged.
2) **Battery Level:** Press the circular power button on the top of the robot and check the battery percentage on the side LCD screen.
3) **Controller Battery:** Especially for outdoor use, check the controller battery by pressing its power **button**. A flashing green light indicates sufficient battery, while a flashing red light indicates low battery and requires charging.

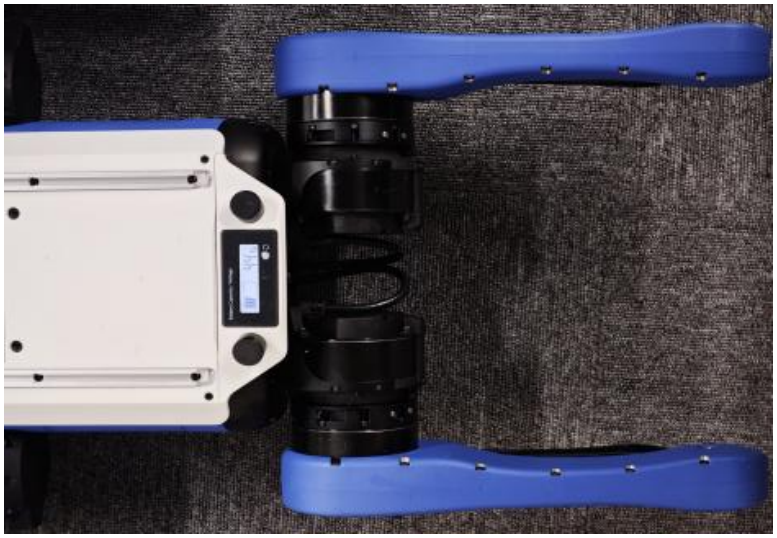


Figure 5.1 Sufficient Robot Battery

Figure 5.2 Sufficient Controller Battery

After confirming everything is correct, place the robot on a level surface as shown in the figure below, with the upper and lower legs folded as illustrated.

Figure 5.3 Folded Six Legs

Power On

1) After completing the on-site inspection and preparation work, press the round button on the top cover of the robot. The indicator light will turn on, indicating that the robot is powered on.
2) After holding the power switch for 2 seconds, you will hear a slight humming sound from the motor, which means that both the motor and the main controller have been powered on.
3) Approximately 40 seconds later, turn on the handle switch. When the handle vibrates and the power indicator remains steadily on, it indicates that the handle has been successfully connected.

5.2 State machine transition

After the robot is powered on and successfully connected to the handle, it will automatically enter the damping mode. The robot includes the following states.

State	Despcription
Damping Mode	Each motor applies a small damping torque, allowing the user to feel a slight resistance when swinging the joints. This mode is mainly used for safety control when an abnormal condition occurs in the robot.
Position Folding Mode	All six limbs move automatically to their predefined starting positions, and each motor applies high damping torque. This mode is used for folding, storage, or static positioning of the robot.
Force-Control Standing Mode	The robot stands upright, and the joystick controls its posture: Left joystick horizontal axis: Yaw angle Left joystick vertical axis: Base height Right joystick horizontal axis: Roll angle Right joystick vertical axis: Pitch angle This mode is used for checking the working status of the robot's joints.
Force-Control Walking Mode	The left joystick controls the yaw direction (horizontal axis) and forward/backward motion (vertical axis); the right joystick controls left/right motion (horizontal axis), while its vertical axis is inactive.
Complex Terrain Walking Mode	The robot slightly lowers its center of gravity, enhancing its adaptability to various terrains such as stairs, flat ground, grass, wetlands, and gravel surfaces.
Boxing Mode	The robot supports itself with its middle and rear limbs while the front limbs perform rapid punching motions. This is a demonstration mode, and upon completion, the robot automatically returns to the Force-

State	Despcription
	Control Standing Mode.
Navigation Mode	---

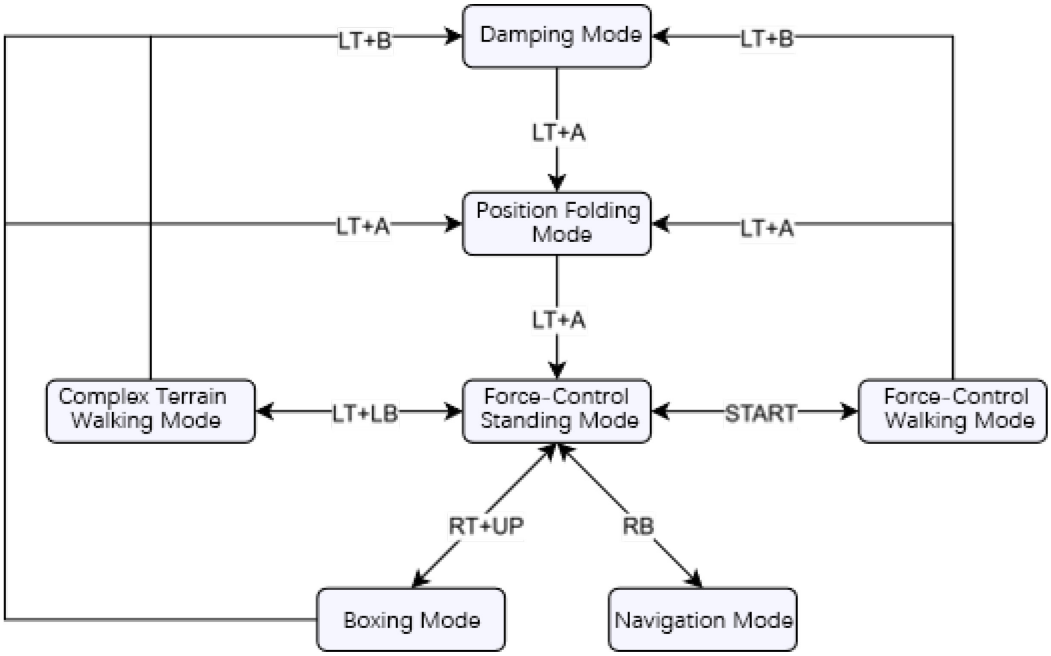
Figure 5.4 Position Folding Mode

Figure 5.5 Forcing control standing mode

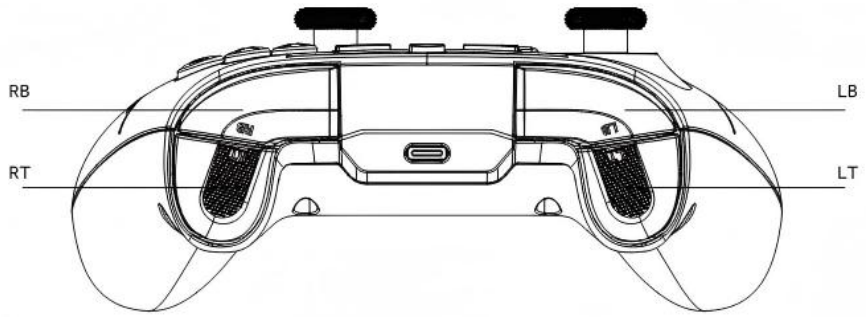
Figure 5.6 Forcing control walking mode

Figure 5.7 Complex terrain walking mode

The transition relationships between different states are shown in the figure below.



The button layout of the handle controller is shown in the figure below.



Controller Key Feature Descriptions:

Key	Status	Description
Home Button	Blue Solid	Normal Connection
	Red Flashing	Low Battery
	Blue Flashing	Connecting...
	Press Once (Short Press)	Triggers controller pairing; successful connection causes vibration.
	Press and Hold (Long Press)	Controller Power Off
Left Joystick	Force Control Standby Mode	Left/Right controls Yaw (Heading/Rotation) posture.
		Up/Down controls Robot height for standing/crouching.
	Force Control Walk Mode	Left/Right controls Yaw Speed (Rotational movement speed).
		Up/Down controls Forward/Backward walking speed.
Right Joystick	Force Control Standby Mode	Left/Right controls Roll (Side-to-side tilt) posture.
		Up/Down controls Pitch (Front/Back tilt) posture.
	Force Control Walk Mode	Left/Right controls Lateral (Strafe) Speed.
		Up/Down has No control value (Zero control).

5.3 Power Off

Fold the robot to its designated position or activate Damping Mode, then power off the robot's mainswitch. The controller will automatically power off, requiring no further action.

5.4 Emergency Operation

WARNING

If the robot becomes unstable, immediately activate Damping Mode (LT+B) or directly shut down the system power. If the joint positions appear abnormal after switching from Damping Mode to Position Mode, power off the unit, manually place it in the correct configuration, and restart.

6. Secondary Development and SDK

6.1 Operating System and Development Environment

The Hexplorer Motion Control Platform uses an Intel Mini PC with i7/i5 Core CPUs, a nominal frequency of 1.2 GHz, and supports both energy-saving and performance modes. It features 6 cores and 16 threads, 16 GB of RAM, and 95 GB of available ROM. The computing platform is based on the Jetson Orin Nano, equipped with an Arm 6XA789 CPU, 8 GB of RAM, 128 GB of ROM, and a GPU with 40 TOPS of computing power.

The operating system is Ubuntu 22.04, with ROS2 Humble pre-installed. The system comes with pre-installed tools including C/C++, Python (3.10), Miniconda, Docker, program compilation tools, and virtual development/debugging environments.

If a conda environment is needed, users must manually activate it. Two activation methods are provided.

1) Effective permanently for all terminals

Uncomment lines 125–135 (inclusive) in the *~/.bashrc* file, then execute the command.

```
source ~/.bashrc
```

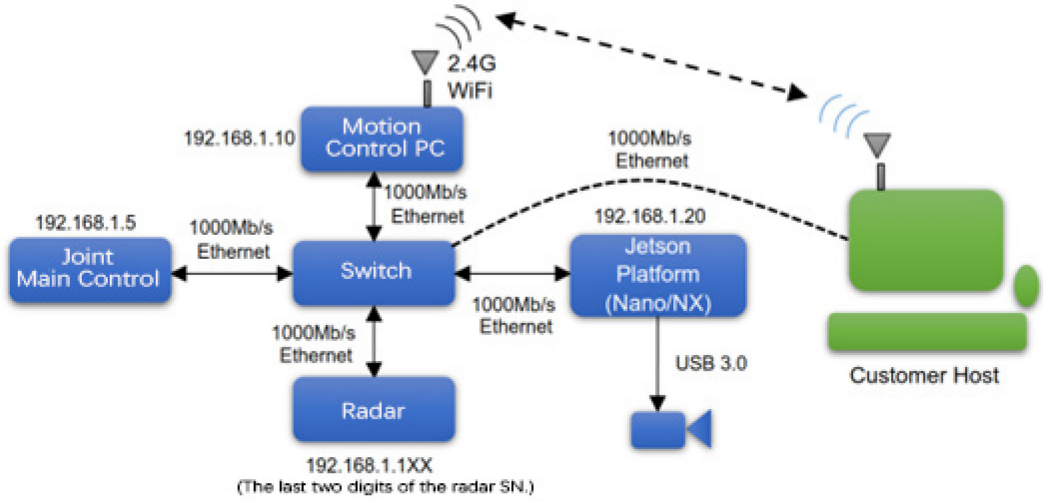
2) Effective temporarily for a single terminal

If you only need to use it in the current terminal, simply run the command directly.

```
source /etc/conda/setup.bash
```

6.2 Network and Access

6.2.1 Internal Network Topology



6.2.2 Network Connection

By default, the robot's Wi-Fi function is disabled. To enable Wi-Fi, you can either connect an external keyboard, mouse, and monitor to modify the settings directly on the onboard desktop, or use the script *start_wifi.sh* by executing the following command in the terminal:

```
./start_wifi.sh <WiFiName> <Password>
```

6.2.3 onboard access

According to the network topology, the robot's wireless and wired network interfaces have the following IP addresses: **192.168.12.1**, **192.168.1.10** (MiniPC), and **192.168.1.20** (Jetson Orin Nano). In addition, the robot comes with **SSH** and **Telnet** pre-installed and both services start automatically upon boot.

For users, there are three ways to connect to the onboard system:

1) Direct Ethernet connection

Either **the MiniPC** or **the Jetson Orin Nano** can be selected — only one of them is required.

```
ssh robot@192.168.1.10 # The username is robot, and the password is 123
ssh robot@192.168.1.20 # The username is robot, and the password is 123
```

2) Wi-Fi connection (recommended)

The Wi-Fi name is **YJ-MiniHexV2-xxx**, and the password is **1234abcd**.

```
ssh robot@192.168.12.1 # The username is robot, and the password is 123
```

3) Direct connection

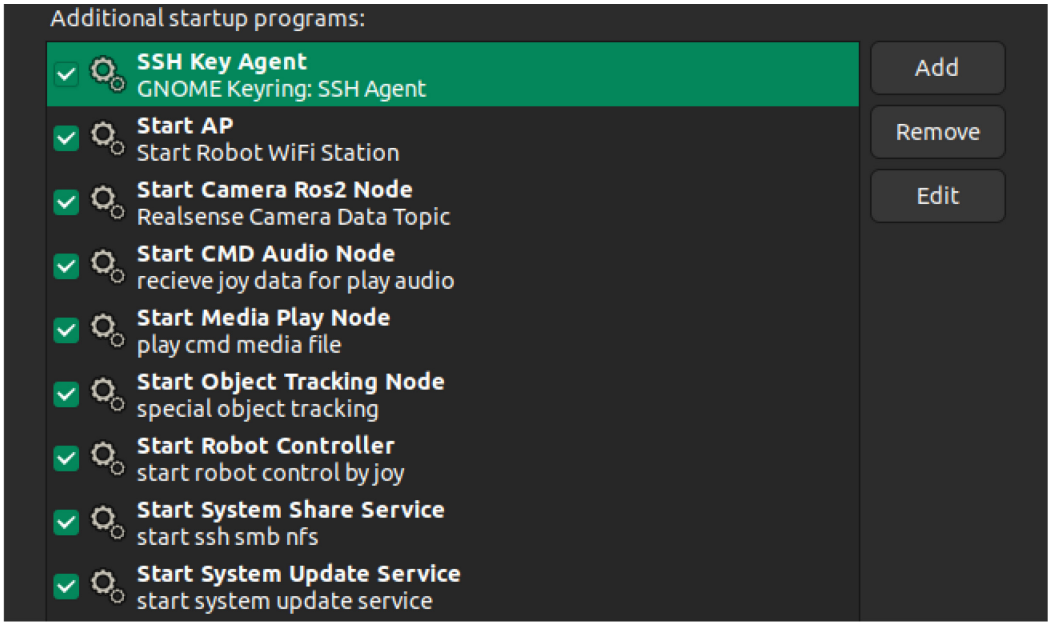
By directly connecting an external monitor, keyboard, and mouse to the onboard hardware interfaces, you can access the Ubuntu desktop directly. This method is not suitable for remote control. For the hardware interface diagram, please refer to "Functional Diagram – External Expansion Support".

6.2.4 File access

The robot comes with NFS and Samba pre-installed, as well as SSH and Telnet services. Users can utilize these file-sharing services as needed.

6.2.5 Autostart Service Management

The system's autostart services are managed using **Startup Applications**. The interface for accessing it is shown in the figure below:



Each item in the interface represents a service that starts automatically when the system boots. By default, the system enables SSH, Samba, NFS, system update services, automatic Wi-Fi hotspot, the Realsense D435 camera node (automatically enabled if the standard camera is installed; ignore if not), the radar node, the robot's basic motion control node, and any custom function-related nodes.

If certain nodes are not needed, they can be disabled. System-related services must not be disabled, as doing so may cause device upgrade failures.

6.3 ROS2 and Node Topology

6.3.1 ROS2 Environment Configuration

The ROS2 environment on the robot's onboard system is pre-configured by default. The installation of a specific ROS2 version is highly dependent on the Ubuntu system version, so make sure to install the ROS2 version that corresponds to the correct Ubuntu version. If users need to reconfigure it, please follow the steps in the provided script:

```
1 ./ros2_env.bash
```

6.3.2 Node Topology Structure

The robot's software system uses ROS2 as the middleware. The robot's operational nodes are developed based on ROS2, as shown in the figure below:

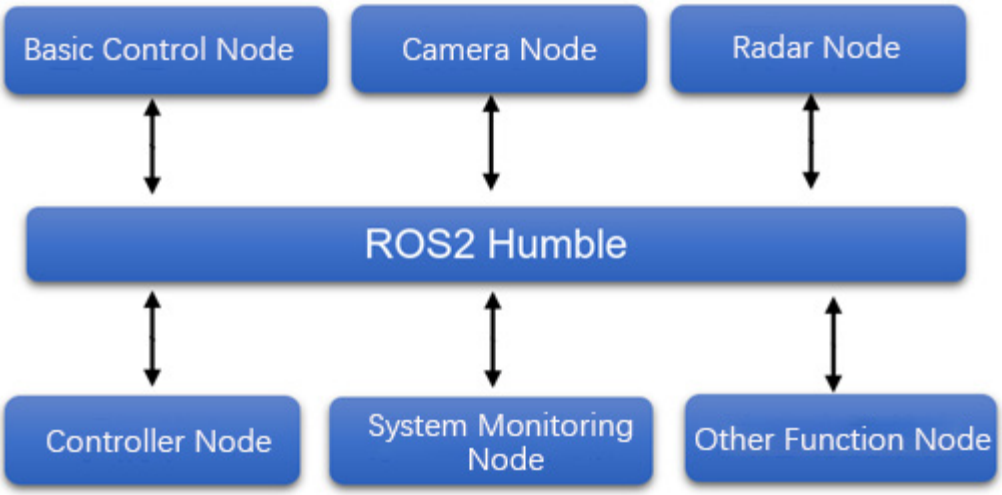


Figure 6.1 Software Architecture Diagram

Any PC on the robot system's network can view the active nodes published by the robot system using the following command:

```
1 ros2 topic list # Check all topics
```

a. On the user's host, if connected via a direct Ethernet link, all nodes published and subscribed within the robot system can be accessed.

b. On the user's host, if connected via Wi-Fi, the nodes published and subscribed by the robot control board can be accessed.

> **NOTE**
>
> The nodes running in the robot system may be pre-installed or not, depending on the configuration. For detailed information, please contact technical support.

6.4 Control System and SDK

All motion-related programs of the robot system are located in the *robot_controller_release* folder under the user directory of the motion control PC system. This folder contains the **config**, **third_party**, and **executable** subfolders:

- **config**: Contains the robot's description files, defining the robot type, including calibration positions for all joints, rotation directions, IDs, etc. Manual modification by the user is generally not required.
- **executable**: Contains executable files such as the main motion control program (main), the joint testing program (motor_test_example), and the system calibration assistant program (motor_test_calibration).
- **third_party**: Contains third-party software resources.

Currently, updates to the control system must be performed locally by the user. Updates are applied by overwriting the entire robot_controller_release folder. After each update, third-party software resources must be reinstalled and updated, and executable permissions must be set for the programs. The detailed procedure is as follows:


```
cd ~/robot_controller_release
./setup.sh
```

Restart the robot. If it starts normally following the standard startup procedure, this indicates that the control software update has been successfully completed.

6.4.1 High-Level Control SDK

Message Types

The robot's internal data communication software architecture uses **ROS2** as the middleware, with communication carried out via publishing or subscribing to ROS2 topic messages.

All high-level message types of the robot are defined in the package named *custom_msg*. The robot system comes with the *custom_msg* package pre-installed, located at:

```
robot_controller_release/ros2_packages/custom_msg/share/custom_msg
```

The directory structure is as follows:

```
drwxrwxr-x 2 robot robot 4096 Sep 18 03:10 ./
drwxrwxr-x 6 robot robot 4096 Sep 18 03:10 ../
-rw-rw-r-- 1 robot robot   88 Sep 18 03:10 CustomPoint.msg
-rw-rw-r-- 1 robot robot   49 Sep 18 03:10 DetectCommand.msg
-rw-rw-r-- 1 robot robot  108 Sep 18 03:10 LivoxPointcloud.msg
-rw-rw-r-- 1 robot robot   55 Sep 18 03:10 MediaPlayer.msg
-rw-rw-r-- 1 robot robot  154 Sep 18 03:10 MotorState.msg
-rw-rw-r-- 1 robot robot   52 Sep 18 03:10 ObjectPosition.msg
-rw-rw-r-- 1 robot robot  109 Sep 18 03:10 ObstacleDetection.msg
-rw-rw-r-- 1 robot robot  118 Sep 18 03:10 RobotCommand.msg
-rw-rw-r-- 1 robot robot  317 Sep 18 03:10 RobotState.msg
-rw-rw-r-- 1 robot robot  171 Sep 18 03:10 TagsDetection.msg
-rw-rw-r-- 1 robot robot  213 Sep 18 03:10 Template.msg
```

NOTE

This development dependency package may vary across different robot models and versions. For detailed information, please contact technical support.

Below is an explanation of some key message data structures.

1) RobotState.msg

Data Name	Description	Data Name	Description
header	Timestamp, coordinate frame, etc.	jpower_total	Total joint power
control_cmd	Control commands	pos_body	Body real-time position, order: (x, y, z), unit: meters (m)

Data Name	Description	Data Name	Description
jtau_leg	Leg joint torque	vel_body	Body real-time velocity, order: (x, y, z), unit: meters per second (m/s)
jvel_leg	Leg joint velocity	acc_body	Body real-time acceleration, order: (x, y, z), unit: meters per second squared (m/s²)
jpos_leg	Leg joint position	ori_body	Body real-time attitude, unit: quaternion, format: (w, x, y, z)
jtau_leg_des	Target leg joint torque	omega_body	Body angular velocity in the world coordinate system, order: (x, y, z), unit: rad/s
jvel_leg_des	Target leg joint velocity	pos_foot_left	Bipedal: Left foot end position
jpos_leg_des	Target leg joint position	pos_foot_right	Bipedal: Right foot end position
jpower_leg	Leg joint power	vel_foot_left	Bipedal: Left foot end velocity
jpos_cmd_leg	Leg joint position command	vel_foot_right	Bipedal: Right foot end velocity
jtau_arm	Arm joint torque	grf_left	Bipedal: Left foot end feedback
jvel_arm	Arm joint velocity	grf_right	Bipedal: Right foot end feedback force
jpos_arm	Arm joint position	rf_vertical_filtered	Bipedal: FZ
jtau_total	Total joint torque	temp	Reserved: temp[10] used to report robot operating status 0: passive 1: standDown 2: standUp 3: balanceStand 4: walk

2) RobotCommand.msg

Data Name	Description
header	Timestamp, coordinate frame, etc.
target_state	Robot operation mode switch: 0: passive 1: standDown 2: standUp 3: balanceStand 4: walk
temp	Reserved

3) geometry_msgs/msg/Twist

Data Name	Description / Explanation
linear.x	Robot linear velocity in the X direction, unit: m/s
linear.y	Robot linear velocity in the Y direction, unit: m/s
angular.z	Robot yaw angular velocity, unit: rad/s

Terminal Command Control

1) Constant-Speed Walking

When the robot is in the walking state, the walking speed command is a combination of the joystick input and the user's high-level command. High-level user commands are sent via the **geometry_msgs/Twist** message, where:

- **linear.x**: linear velocity command in the robot's forward direction (m/s)
- **linear.y**: linear velocity command in the robot's lateral direction (m/s)
- **angular.z**: rotational velocity command of the robot (rad/s)

For example, by entering the following command in the terminal, the robot will walk forward at 0.1 m/s:

```
source ~/robot_controller_release/ros2_packages/setup.bash
ros2 topic pub -r 10 /vel_cmd geometry_msgs/Twist '{linear: {x: 0.1, y: 0, z: 0}, angular: {x: 0, y: 0, z: 0}}'
```

2) State Switching

The following command can be used to control the robot's state switching:


```
ros2 topic pub -1 /robot_cmd custom_msg/msg/RobotCommand "{target_state: 0}"
```

C++ High-Level Control

Users can download the example code **dobot_hex_sdk**, with the example programs located in: `dobot_hex_sdk/high_level/src/`

This repository contains three examples. Some of these examples will be explained below. Before that, to support user-defined programs, the control framework setup will be introduced.

1) Package Creation, Compilation, and Execution

Users can directly use **CMake** to build the package without relying on ROS2. This means that tools like **ament** and **colcon** are not required. This method is the simplest and most direct approach and is also the one used by the example programs (recommended).

If users need to use **colcon** and **ament_cmake** as the build tools, they must first create a ROS2 package by executing the following command in the **src** directory of the workspace:

```
ros2 pkg create --build-type ament_cmake <package_name>
```

This will create a package directory named `<package_name>`. The overall workspace directory structure may look like this:

Users can develop control programs within their created package. Once the program is complete, including source files, `CMakeLists.txt`, and other necessary files, execute the following command in the root directory of the workspace (`workspace_folder/`):

```
rosdep install -i --from-path src --rosdistro humble -y
```

This command will check whether all dependencies of the package are satisfied.

> **NOTE**
>
> Before using *rosdep* and the build commands, make sure that the ROS2 environment variables are properly sourced.

For the example programs, the following commands need to be executed:

```
source /opt/ros/humble/setup.bash
source ~/robot_controller_release/ros2_packages/setup.bash
```

After completing the dependency check, still in the workspace root directory, execute the build command:

```
colcon build --packages-select <package_name>
```

Here, `<package_name>` refers to the name of the package created by the user. This command will create **build**, **install**, and **log** folders under the **src** directory. The package and its executables will be installed in the **install** directory.

To be able to call the package and its executables within the workspace, execute the following command in the **src** directory from the terminal:

```
source install/setup.bash
```

After that, the node can be executed/run.

```
ros2 run <package_name> <executable file>
```


The name of the executable here is defined by the user in the **CMakeLists.txt** file.

2) Example 1：body_twist

The file is located at: *dobot_hex_sdk/high_level/src/body_twist.cpp*

This example is intended to check the status of each joint motor. Place the robot flat on a level surface with its six legs folded as shown in the figure below:

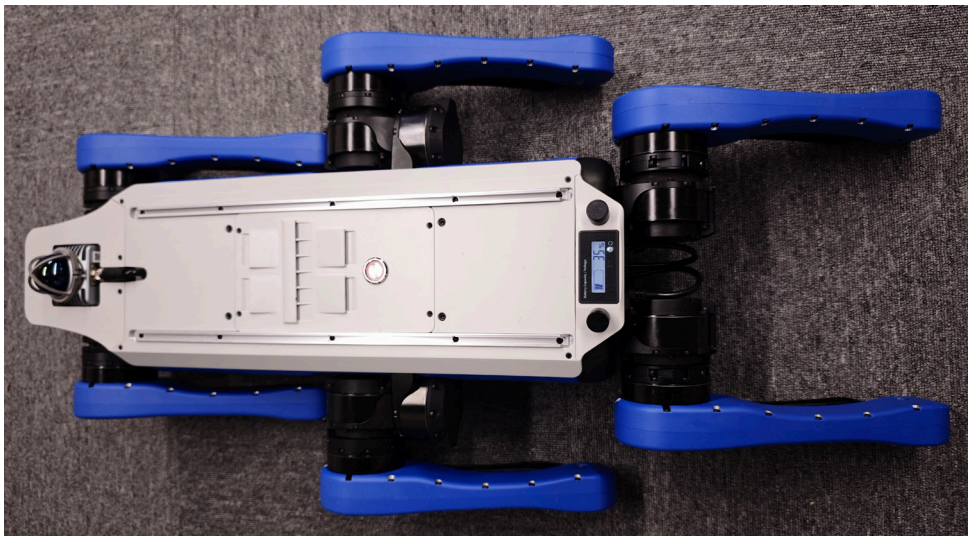


Figure 6.2 Place the six legs folded on a level surface

When running this example under normal conditions, the user should observe that the hexapod robot first assumes a standing posture, after which each joint motor moves periodically according to a certain pattern, including posture movements in roll, pitch, and yaw directions, as well as base height adjustments.

> **NOTE**
>
> When stopping the program, first stop the example program, then use the joystick to switch the robot to another mode.

This example involves publishing and subscribing to the **robot_cmd**, **robot_state**, and **joy** topics. Users can refer to this code for developing custom high-level control.

Next is a brief explanation of the core code:

The main program is executed as a ROS2 node. Any publishing or subscribing to existing topics must be implemented within the node. Therefore, functional classes should inherit from *rclcpp::Node* and create timers, publishers, and subscribers, etc. The code below is a simplified version; please refer to the actual program for the complete implementation.

```
01  class controlFuncNode : public rclcpp::Node
02  {
03  public:
04  controlFuncNode(const std::string &strNodeName) : Node(strNodeName)
05  {
06  using namespace std::literals::chrono_literals;
07  using std::placeholders::_1;
08
09
```

```
m_pControlTopicTimer = this->create_wall_timer(20ms,
std::bind(&controlFuncNode::controlMsgFunc, this));
m_pRobotcommandPublisher =
this->create_publisher<custom_msg::msg::Robotcommand>(m_strRobotcommandTopicName, 1);
m_pRobotstatesubscriber =
this->create_subscription<custom_msg::msg::Robotstate>(m_strRobotstateTopicName, 1,
std::bind(&controlFuncNode::parseRobotStateMsgFunc, this, _1));
}
      void controlMsgFunc(void);
      void parseRobotStateMsgFunc(const custom_msg::msg::RobotState::SharedPtr msg);

private:
rclcpp::TimerBase::SharedPtr m_pControlTopicTimer = nullptr;
rclcpp::Publisher<custom_msg::msg::RobotCommand>::SharedPtr m_pRobotcommandPublisher =
nullptr;
rclcpp::Subscription<custom_msg::msg::RobotState>::SharedPtr m_pRobotStateSubscriber =
nullptr
};
```

ROS2 calls the function **controlMsgFunc** at a 20 ms interval. After constructing the message to be published, the publisher sends it out. The subscriber, upon receiving the message, passes it to the bound function **parseRobotStateMsgFunc** for further processing.

A reference example for publishing and subscribing is as follows:

```
void controlFuncNode::controlMsgFunc(void)
{
custom_msg::msg::RobotCommand cmdMsg;
cmdMsg.target_state = m_msgRobotState.temp[10] + 1;
m_pRobotCommandPublisher->publish(cmdMsg);
}

void controlFuncNode::parseRobotStateMsgFunc(const custom_msg::msg::RobotState::SharedPtr
msg)
{
m_msgRobotState.temp[10] = msg->temp[10]; // Robot State Get Byte
RCLCPP_INFo(this->get_logger(), "subscribe Robot State Msg, m_msgRobotstate.temp[10] = %f",
m_msgRobotstate.temp[10]);
}
```

This example uses information from the **joy** topic. ROS2 has this message type built in, under the package sensor_msgs. If users need to simulate joystick input, they can refer to the following code:

```
const float f_roll = 0.012;
const float f_pitch = 0.012;
const float f_yaw = 0.012; // Hz：yaw frequency (LX)
const float f_z = 0.012;

sensor_msgs::msg::Joy joyMsg;
joyMsg.header.stamp = this->now();
joyMsg.axes.assign(8, 0.0f);
joyMsg.buttons.assign(11, 0);
joyMsg.axes[2] = 1.0f;
joyMsg.axes[5] = 1.0f;
```



```
12
13  joyMsg.axes[3] = std::clamp(static_cast<float>(0.7 * cos(2 * M_PI * f_roll * count)), -1.0f, 1.0f);    // roll
14  joyMsg.axes[4] = std::clamp(static_cast<float>(0.7 * cos(2 * M_PI * f_z * count)), -1.0f, 1.0f);         // pitch
15  joyMsg.axes[0] = std::clamp(static  cast<float>(0.7 * sin(2 * M  PI * f  yaw * count)), -1.0f, 1.0f);     // yaw
16  joyMsg.axes[1] = std::clamp(static_cast<float>(0.7 * sin(2 * M_PI * f_pitch * count)), -1.0f, 1.0f); // z
17  m_pJoyTopicPublisher->publish(joyMsg);
18
```

The joystick contains 8 axes and 11 buttons. The mapping of the axes is as follows:

ID	Axis/Button	Description	ID	Axis/Button	Description
0	Left joystick horizontal	Left to right: [1, -1]	4	Right joystick horizontal	Up to down: [1, -1]
1	Left joystick vertical	Up to down: [1, -1]	5	RT	Press to release: [-1, 1]
2	LT	Press to release: [-1, 1]	6	D-pad horizontal	Left 1, Right -1, None 0
3	Right joystick horizontal	Left to right: [1, -1]	7	D-pad vertical	Up 1, Down -1, None 0

Buttons: All buttons are 1 when pressed and 0 when released.

ID	Button	ID	Button
0	A	6	SELECT
1	B	7	START
2	X	8	HOME
3	Y	9	LEFT STICK PRESS
4	LB	10	RIGHT STICK PRESS
5	RB		

To correctly locate **custom** and **built-in message types**, users need to specify them in **CMakeLists.txt**. For example, in the above example, we use the **custom_msg** package. In **CMakeLists.txt**, this would be:

```
1  find_package(custom_msg REQUIRED)
2  include_directories(
```



```
3 ${custom_msg_INCLUDE_DIRS}
4 )
5 tartt_link_libraries(<your_executable_name>
6 ${custom_msg_LIBRARIES}
7 )
```

3) Example 2：locomotion

The file is located at: *dobot_hex_sdk/high_level/src/locomotion.cpp*

This example is intended to check the walking behavior of the hexapod robot. As before, place the robot flat on a level surface with its six legs folded.

1. Under normal operation, running this example should make the robot:
2. Move from a folded position to a standing posture.
3. Walk forward, backward, left, and right.
4. Turn left and right.

Finally, automatically lie down.

> **NOTE**
>
> When stopping the program, first stop the example, then use the joystick to switch the robot to another mode.

Compared to the first example, this one involves an additional topic: **geometry_msgs**, which contains 3D velocity information. Users need to create a **publisher** for this command, for example:

```
1 m_pRobotControllVelCmdPublisher =
  this->create_publisher<geometry_msgs::msg::Twist>(m_strVelCmdTopicName, 1);
```

Users can define the robot's velocity in three directions: **x**, **y**, and **yaw (angular z axis)**. The reference code is as follows:

```
01 auto controlMsg = geometry_msgs::msg::Twist();
02 controlMsg.linear.x = fXSpeed;
03 controlMsg.linear.y = fYSpeed;
04 controlMsg.angular.z = fYawSpeed;
05
06 controlMsg.linear.x = m_msgRobotControl.linear.x * 0.5 + fXSpeed * 0.5;
07 controlMsg.linear.y = m_msgRobotControl.linear.y * 0.6 + fYSpeed * 0.4;
08 controlMsg.angular.z = m_msgRobotControl.angular.z * 0.65 + fYawSpeed * 0.35;
09
10 m_pRobotControllVelCmdPublisher->publish(controlMsg);
```

4) Example 3：robot_state

This example is intended to read relevant robot data to facilitate subsequent debugging and secondary development. The robot_state message contains multiple variables; here we use pos_body, vel_body, acc_body, ori_body, and omega_body as examples.

Under normal operation, running this example will make the hexapod robot remain in

force-controlled standing mode, and these values can be observed in the terminal, as shown below:

```
robot@minipc-2204:~/robot_controller_examples/high_level_cpp/robot_state/build$ ./robotStateSubscriber
[INFO] [1758532817.351805819] [robot_state_subscriber]: Main robot states as below
control with gamepad and observe the states
+--------------+-------------------------------------+
| Name         | Value                               |
+--------------+-------------------------------------+
| pos_body     | [-0.014, -0.018, 0.079]             |
| vel_body     | [0.000, -0.000, 0.001]              |
| acc_body     | [-0.015, 0.039, 0.038]              |
| ori_body     | [1.000, -0.019, 0.013, -0.000]      |
| omega_body   | [0.000, 0.000, 0.000]               |
+--------------+-------------------------------------+
```

After running this example, users can use the joystick to test the movement of each joint while the robot is in force-controlled standing mode. The changes in the data mentioned above can be observed in the terminal.

For example, by moving the left joystick vertical axis, the robot's base will descend, and a noticeable change in pos_body can be seen in the terminal.

> **NOTE**
>
> When stopping the program, first stop the example program, then use the joystick to switch the robot to another mode.

Python High-Level Control

Users can download the example code dobot_hex_sdk, with the Python example programs located at: *dobot_hex_sdk/high_level/py/*

This repository contains three examples. If using the ROS package mode to build the program, please refer to the previous section "C++ High-Level Control – Package Creation, Compilation, and Execution" before writing your program.

To execute Python files, you need to access the Intel MiniPC and use the ROS2-provided Python 3.10 interpreter, not the Conda Python environment. Therefore, ensure that the ROS2 environment is properly activated.

```
1  source /opt/ros/humble/setup.bash
```

1) Example 1：body_twist

The functionality implemented in Python is identical to the C++ version, with only slight differences in the way ROS2 functions are called.

The file is located at: *dobot_hex_sdk/high_level/py/body_twist.py*

This example is intended to check the status of each joint motor. Place the robot flat on a level surface with its six legs folded, as shown in the figure below:

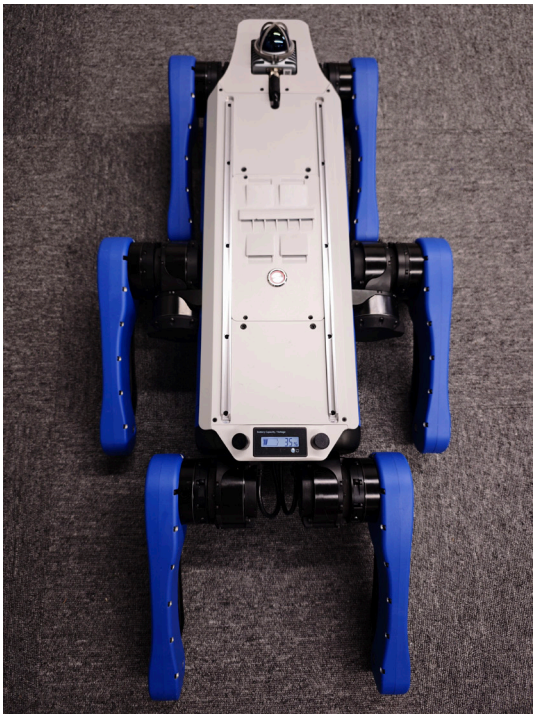


Figure 6.3 Place the six legs folded on a level surface

This example involves publishing and subscribing to the robot_cmd, robot_state, and joy topics. Users can refer to this code to develop their own custom high-level control. In the Python environment, the creation of timers, publishers, and subscribers can be referenced in the following code:

```
self.robot_command_topic_name = "/robot_cmd"
self.robot_state_topic_name = "/robot_state"
self.joy_handle_topic_name = "/joy"

self.publisher_timer = self.create_timer(0.02, self.body_twist_control_msg)

self.robot_command_publisher = self.create_publisher(
RobotCommand, self.robot_command_topic_name, 1
)
self.joyTopicPublisher = self.create_publisher(
Joy, self.joy_handle_topic_name, 1
)
self.robot_state_subscriber = self.create_subscription(
RobotState, self.robot_state_topic_name, self.robot_state_parse, 1
)
```

This example utilizes information from the joy topic. ROS2 has this message type built in, under the package sensor_msgs. If users need to simulate joystick input, they can refer to the following code:

```
f_roll = 0.012
f_pitch = 0.012
f_yaw = 0.012    # Hz：yaw frequency（Lx）
f_z = 0.012

```



```
joyMsg = Joy()
joyMsg.header.stamp = self.get_clock().now().to_msg()

joyMsg.axes = [0.0] * 8
joyMsg.buttons = [0] * 11

# 固定触发器
joyMsg.axes[2] = 1.0
joyMsg.axes[5] = 1.0

roll_val = 0.7 * math.cos(2 * math.pi * f_roll * self.count)    # roll
pitch_val = 0.7 * math.cos(2 * math.pi * f_z * self.count)    # pitch
yaw_val = 0.7 * math.sin(2 * math.pi * f_yaw * self.count)    # yaw
z_val = 0.7 * math.sin(2 * math.pi * f_pitch * self.count)    # z

clamp = lambda x: max(-1.0, min(1.0, float(x)))
joyMsg.axes[3] = clamp(roll_val)
joyMsg.axes[4] = clamp(pitch_val)
joyMsg.axes[0] = clamp(yaw_val)
joyMsg.axes[1] = clamp(z_val)

self.joyTopicPublisher.publish(joyMsg)
```

This joystick contains 8 axes and 11 buttons. The mapping of the buttons is as follows:

Axes:

ID	Button/Axis	Description	ID	Button/Axis	Description
0	Left Stick Horizontal	Left to Right: [1, -1]	4	Right Stick Horizontal	Up to Down: [1, -1]
1	Left Stick Vertical	Up to Down: [1, -1]	5	RT	Pressed to Released: [-1, 1]
2	LT	Pressed to Released: [-1, 1]	6	D-Pad Horizontal	Left 1, Right -1, Not Pressed 0
3	Right Stick Horizontal	Left to Right: [1, -1]	7	D-Pad Vertical	Up 1, Down -1, Not Pressed 0

Buttons: All keys are ‘1’ when pressed and ‘0’ when released

ID	Button/Axis	ID	Button/Axis
0	A	6	SELECT
1	B	7	START
2	X	8	HOME

ID	Button/Axis	ID	Button/Axis
3	Y	9	LEFT STICK PRESS
4	LB	10	RIGHT STICK PRESS
5	RB		

The main program is executed as a ROS2 node, publishing to and subscribing from existing nodes via topics. After writing the program, run the following command in the console:

```
python3 <executable file>
```

Under normal operation of this example, the user should see the hexapod robot first assume a standing posture, after which the joint motors continuously move in a periodic pattern. This includes movements in roll, pitch, and yaw directions, as well as motion along the base height.

> **NOTE**
>
> When stopping the program, please first stop the example program, and then use the joystick to switch the robot to other modes.

2) Example 2：locomotion

The file is located at *dobot_hex_sdk/high_level/py/locomotion.py*. This example is intended to test the walking behavior of the hexapod robot. The robot should still be placed flat on a level surface with its six legs folded, as described above.

Under normal operation of this example, the user should see the robot transition from a folded position to a standing posture, and then move sequentially: forward, backward, left strafe, right strafe, left turn, and right turn.

> **NOTE**
>
> When stopping the program, please first stop the example program, and then use the joystick to switch the robot to other modes.

Compared with Example 1, this example involves an additional topic, **geometry_msgs**, which includes velocities in 3D space. As in Example 1, the user needs to create a publisher for this command, for instance:

```
self.vel_cmd_publisher = self.create_publisher(Twist, self.vel_cmd_topic_name, 1)
```

Users can define the robot's velocity in three directions: **x**, **y**, and **yaw** (angular z-axis). The reference code is as follows:

```
controlMsg = Twist()
controlMsg.linear.x = fXSpeed * 0.5
controlMsg.linear.y = fYSpeed * 0.4
controlMsg.angular.z = fYawSpeed * 0.35

```

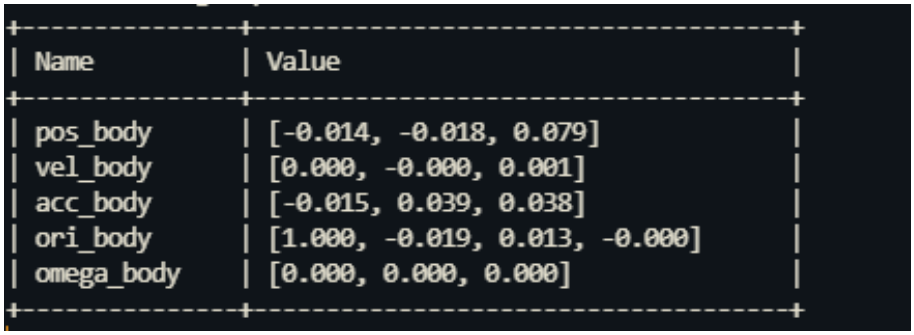


```
6    self.vel_cmd_publisher.publish(controlMsg)
```

3) Example 3：robot_state

This example is designed to read relevant robot data for subsequent debugging and secondary development. The **robot_state** contains multiple variables; here, **pos_body**, **vel_body**, **acc_body**, **ori_body**, and **omega_body** are used as examples.

Under normal operation, running this example will keep the hexapod robot in force-controlled standing mode, and the values of these variables can be observed in the terminal, as shown below:



After running this example, users can use the joystick to test the movement of each joint while the robot is in force-controlled standing mode. The changes in the data mentioned above can also be observed in the console. For example, pushing the left stick along the vertical axis will lower the robot's base, and a corresponding change in **pos_body** can be clearly seen in the terminal.

> **NOTE**
>
> When stopping the program, please first stop the example program, and then use the joystick to switch the robot to other modes.

6.4.2 Joint Control SDK

C++ Joint Control

motor_sdk is Dobot Technology's upper-level software for their self-developed multi-joint robot communication board. It is mainly used for robot control, debugging, data acquisition, and secondary development. The folder is located under the motion control PC system at */home/robot/robot_controller_release* (user's home directory).

Header File Overview:

Header File	Description
absolute_timer.h	System Time Function & Absolute Wait Class – Used to control communication frequency; the unit is seconds.
comm.h	Communication Data Structures – Defines structures for motors, IMU, and battery data.
error_class.h	IO Exception Class – Used to print exception or error statuses related to IO ports.
IOPort.h	General IO Communication Class – Base class for other communication types.
serial_port.h	Serial Communication Class – Handles USB serial communication.
UDPPort.h	UDP Communication Class – Handles Ethernet (UDP) communication.
motor_sdk.h	Motor SDK Class – Provides upper-level API for the communication board to control motors.
robot_sdk.h	Robot SDK Class – Encapsulates multi-threaded robot control, communication, and monitoring; ready for direct user use.
periodic_function.h	Periodic Task Function – Provides functions for tasks that run periodically.

The example files are located in */motor_sdk/example*. Before running the joint control SDK examples, you need to first terminate the robot's onboard control program.

```
1 ps -ef | grep start_controller.sh
2 sudo kill <progress ID>
3 ps -ef | grep main
4 sudo kill <progress ID>
```

Then, compile the examples.

```
1 cd ~/motor_sdk
2 mkdir build && cd build
```



```
cmake ..
make
```

NOTICE

Before running these two examples, please ensure that the hexapod robot's six legs are folded and placed on a flat surface.

```
./motor_read miniHex_v2
./motor_wave miniHex_v2
```

Two example programs are provided for reference.

- **motor_read**: Under normal operation, this program prints the angles and output torques of all 18 motors in the console. While it is running, users can manually move the joints and feel slight resistance from the motors. Corresponding changes in joint angles can also be observed in the terminal output.
- **motor_wave**: This program is designed to test motor command writing. It sends a set of predefined motor commands to the motors, causing each joint to move slightly in a periodic pattern.

Below, we take **motor_wave** as an example to briefly explain the core code:

```
class Custom {
public:
void damp(RobotCommand &command) {
command.qDes.setZero();
command.qdDes.setZero();
command.tauDes.setZero();
command.kpDes.setZero();
command.kdDes.setConstant(0.5);
}
void controlFunction(const RobotState &state, RobotCommand &command) {
if (control_iter < 10) {
q_init = state.q;
std::cout << "get q init " << q_init.transpose() << std::endl;
}
else if (control_iter < 5000){
double s = double(control_iter - 1000) / 500.0;
        for (int i = 0; i < q_init.size(); i++) {
command.qDes(i) = q_init(i) + sin(2 * M_PI * s) * 0.2;
command.qdDes.setZero();
command.tauDes.setZero();
command.kpDes.setConstant(5);
command.kdDes.setConstant(0.25);
}
}
else {
damp(command);
flag = true;
}
control_iter++;
}
private:
```



```
32    int control_iter = 0;
33    DVecd q_init;
34    };
```

Users create a **RobotSDK** object by providing a configuration file, the control period, and a user-defined control function, **controlFunction**, as inputs.

- **controlFunction Parameters:**
  - RobotState: the current robot state read from the system.
  - RobotCommand: the robot control commands to be sent by the user.
- **Custom Controller Class:**

  In the example shown, the user's custom controller performs the following:

  1. During the first 10 control_iter cycles, it reads the initial joint positions q_init.
  2. For the next 5000 cycles, all joints perform periodic positional oscillations based on q_init as the initial state, with joint stiffness set to 1 and zero-velocity damping coefficient set to 0.2.
  3. After 5000 cycles, the robot enters a fully damped state with a damping coefficient of 0.5.
- **RobotSDK Backend:**

  The SDK handles virtual joint and motor-level interactions, ensuring safety such as system position limits.
- **Recommendation:**

  To make the robot's movement more noticeable, it is advised to position the joints away from the mechanical limits before starting the program.

Python Joint Control

Pybind11 Integration for Python Joint Control

Pybind11 is a library used to bind C++ code with Python, exposing C++ classes and functions for Python use.

- **Library Path:**

  motor_sdk/python_wrapper/third-party/pybind11/
- **C++ Interface API:**

  motor_sdk/python_wrapper/robot_interface.cpp
  - This file is based on the C++ SDK.
  - It defines the **Robot Interface class** used to send commands to motors and receive motor states.
  - The class is exposed to Python using `PYBIND11_MODULE`, allowing Python scripts to directly use the C++ API.
- **Example Files:**

  Located in */motor_sdk/python_wrapper/*

- **Important:**

  Before running the Python joint control SDK examples, you must first terminate the robot's onboard control program.

```
ps -ef | grep start_controller.sh
sudo kill <progress ID>
ps -ef | grep main
sudo kill <progress ID>
```

Before running the example, please ensure that the hexapod robot's six legs are folded and placed on a flat surface.

Then, execute the example:

```
python3 motor_read.py
python3 motor_wave.py
```

Under normal operation, **motor_read** prints the angles and output torques of all 18 motors in the console. While it is running, users can manually move the joints and feel slight resistance from the motors. Corresponding changes in joint angles can also be observed in the terminal output.

The **motor_wave** program is designed to test motor command writing. It sends a set of predefined motor commands to the motors, causing each joint to move slightly in a periodic pattern.

Below, we take **motor_wave** as an example to briefly explain the core code:

```
def step(self):
self._command = self._q_init
self._count = 0
for t in np.arange(0.0, 10.0, self._dt):
# print("---------- step ", self._count, "---------- ")
s = self._count / 500.0
self._count += 1
for idx in np.arange(0, self._motor_num, 1):
self._kpDes[idx] = 5.0
self._kdDes[idx] = 0.25
self._qDes[idx] = self._q_init[idx] + math.sin(2*math.pi*s) * 0.2
self._qdDes[idx] = 0.0
self._tauDes[idx] = 0.0

self._interface.send_command(
self._kpDes, self._kdDes, self._qDes, self._qdDes, self._tauDes
)
self._observation = self._interface.receive_observation()
self.wait()
```

The key point is that the user needs to create an instance of **RobotInterface** and use this instance to communicate with the underlying motors.

To make the robot's movements more noticeable, it is recommended to position the joints away from their mechanical limits before starting the program.

6.4.3 Depth Camera

The robot system comes pre-installed with an Intel RealSense D435 camera. A node capable of acquiring camera data is integrated into the robot system and installed in the system directory:

~/robot_controller_release/ros2_packages/

Depth camera structural installation parameters:

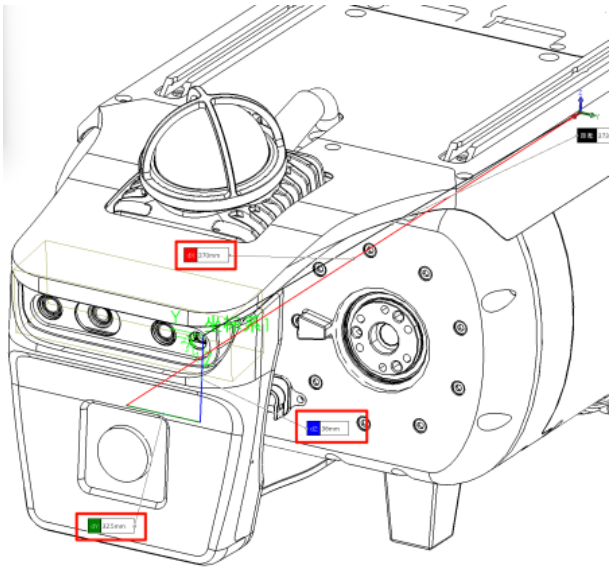


Figure 6.4 Camera data relative to the geometric center coordinate frame

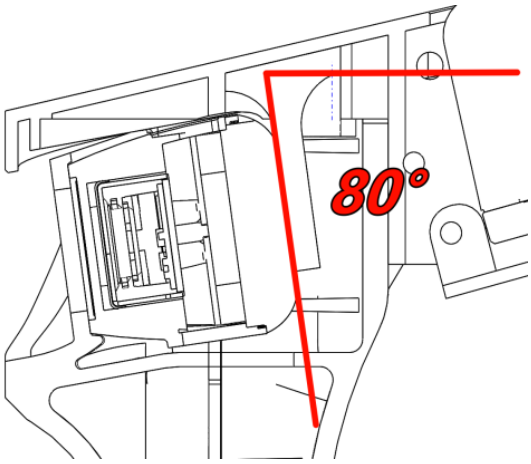


Figure 6.5 The camera plane forms an 80° angle with the horizontal

Launch the node

The depth camera node needs to be started on the Jetson platform (Nano/NX). Its IP address, as referenced in the “Network and Access” section, is **192.167.1.20**.

```
1  ssh robot@192.168.1.20   # Passcode 123
```

Execute the command under the ros2_packages directory.

```
1  source ./setup.bash
2  ros2 launch realsense_camera_node start_node.launch.py
```

You should see the following output:


```
robot@orin-nx-super-2204:~/robot_controller_release/ros2_packages$ ros2 launch realsense_camera_node start_node.launch.py
[INFO] [launch]: All log files can be found below /home/robot/.ros/log/1970-02-12-08-14-51-502848-orin-nx-super-2204-3707
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [realsense_camera_node-1]: process started with pid [3708]
[realsense_camera_node-1] [INFO] [0003629693.149966688] [realsense_camera_node]: realsense_camera_node started
[realsense_camera_node-1] [INFO] [0003629693.177001568] [realsense_camera_node]: 1 Realsense Camera Detected.
[realsense_camera_node-1] [INFO] [0003629693.983069824] [realsense_camera_node]: Start 408122070053 Camera Pipeline Success!
[realsense_camera_node-1] [INFO] [0003629696.170387328] [realsense_camera_node]: Create Camera Stream Process Thread OK!
[realsense_camera_node-1] [INFO] [0003629696.171110336] [realsense_camera_node]: Start 0 Camera Stream Processing ...
```

Keep the current terminal open, then open a new terminal and run the command to check whether the node is running properly. If everything is normal, you should see the following new topics (the sequence number after **sn** is not fixed):

```
/realsense_camera_node/description
/realsense_camera_node/sn408122070053/camera_info
/realsense_camera_node/sn408122070053/color/bgr/image_raw
/realsense_camera_node/sn408122070053/depth/u16/image_raw
```

Node Configuration

The directory for storing the camera node configuration file:

/home/robot/robot_controller_release/ros2_packages/realsense_camera_node/share/realsense_camera_node/config

The camera node configuration file is:

realsense_camera_node_parameters_setting.yaml

Main contents are as follows:

```
realsense_camera_node:
    ros__parameters:
        system_monitor_topic: "system_monitor_defined"
camera_image_frame_width: 640
camera_image_frame_height: 480
camera_image_frame_rate: 30
camera_imu_frame_rate: 200
robot_base_frame_id: "robot_base_frame_defined"
point_cloud_sparse_coffes: 4.0
is_compressed_color_image: false
is_publish_pointcloud: false
is_depth2color: true
```

The meanings of the main parameters in the configuration file are as follows:

- system_monitor_topic：Robot Monitoring Node Name
- camera_image_frame_width：Camera image width
- camera_image_frame_height：Camera image height
- camera_image_frame_rate：Camera image frame rate
- camera_imu_frame_rate： Camera IMU output frame rate: The current node does not output IMU data externally. Please contact technical support if needed.

- robot_base_frame_id：Robot base coordinate system name
- point_cloud_sparse_coffes：Point cloud sparsity coefficient: The total number of points in the point cloud is pc_num = camera_image_frame_width * camera_image_frame_height. After sparsification, pc_num = camera_image_frame_width * camera_image_frame_height / point_cloud_sparse_coefficient.
- Is_publish_pointcloud：Whether to publish the depth camera's point cloud image
- is_depth2color：Whether to perform synchronization and registration of depth and color images

Topic publishing and subscribing

After the node starts, it will register several related topics. Once the topics are published, users can visualize the data using tools like Rviz. The topics and their meanings are as follows:

1. **/realsense_camera_node/description**

Camera node description and topic publishing: This mainly introduces the main content of the camera node. Its message type is **std::msg::String** and is organized in JSON format. The specific content can be viewed by converting the topic message into JSON format.

2. **/realsense_camera_node/sn408122070053/camera_info**

Camera information, topic publishing: Mainly publishes the camera image width and height, intrinsic parameters, etc.

3. **realsense_camera_node/sn408122070053/color/bgr/image_raw**

Camera color image information, topic publishing: Publishes image messages in BGR format.

4. **/realsense_camera_node/sn408122070053/depth/u16/image_raw**

Camera depth image information, topic publishing: The depth image is a single-channel grayscale image.

5. **/realsense_camera_node/sn052622070190/xyz/pointcloud**

Camera point cloud information, topic publishing: The camera point cloud is organized as XYZ, without color information.

6. **/tf_static**

Camera TF information, topic publishing: Publishes the camera TF information, mainly the transformation from the robot to the camera link, and from the camera link to the camera coordinate system.

7. /system_monitor_defined

Monitoring messages, topic subscribing: Subscribes to messages from the monitoring node, used to execute commands from the control node.

Figure 6.6 Depth image

Figure 6.7 Camera image

Program subscribes to topics

The files are located in *dobot_hex_sdk/realsense/src/*, with the Python version in *dobot_hex_sdk/realsense/py/*. This example demonstrates how to subscribe to topic messages using ROS2 custom message types. After starting the depth camera node according to the above documentation, create a build and compile it in the **realsense** directory, then execute the two executable files. Note that all operations related to the depth camera must be performed on the Jetson platform; otherwise, the performance of user-developed programs will be significantly reduced.

1) **camera_info** uses the message type **sensor_msgs::msg::CameraInfo**, which includes internal parameters of the camera. The example program prints its image width, image height, and key parameters in the camera's intrinsic matrix, including the focal lengths in the x and y directions and the principal point coordinates. Normally, users should see these parameters being printed to the console at low frequency.

```
1  auto qos = rclcpp::SensorDataQoS();
2
```

```
m_pCameraInfoSubscriber =
this->create_subscription<sensor_msgs::msg::CameraInfo>(m_strDepthCameraInfoTopicName,
qos, std::bind(&DepthLiteCameraInfoSubscriber::infoCallback, this, _1));
```

2) **`camera_fps`**, message type involved is **sensor_msgs::msg::Image**, which is the data type for images acquired by the camera, including BGR color images and depth images. Under normal conditions, users should see the frame rate, encoding format, and image size of the color and depth images being continuously printed at high frequency in the console.

```
1 auto qos = rclcpp::SensorDataQoS();
2 m_pDepthImageSubscriber =
this->create_subscription<sensor_msgs::msg::Image>(m_strDepthImageTopicName, qos,
std::bind(&DepthLiteCameraFPSSubscriber::depthCallback, this, _1));
3 m_pBGRImageSubscriber =
this->create_subscription<sensor_msgs::msg::Image>(m_strBGRImageTopicName, qos,
std::bind(&DepthLiteCameraFPSSubscriber::bgrCallback, this, _1));
```

6.4.4 Radar

The robot system is pre-installed with the Mid-360 LiDAR. A node capable of acquiring LiDAR data information is integrated into the robot system and installed in the system directory: *~/robot_controller_release/ros2_packages/*

The LiDAR installation parameters are as follows:

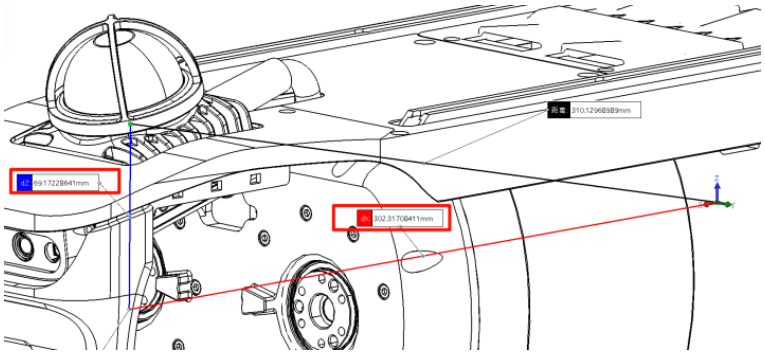


Figure 6.8 LiDAR relative data with respect to the geometric center coordinate system

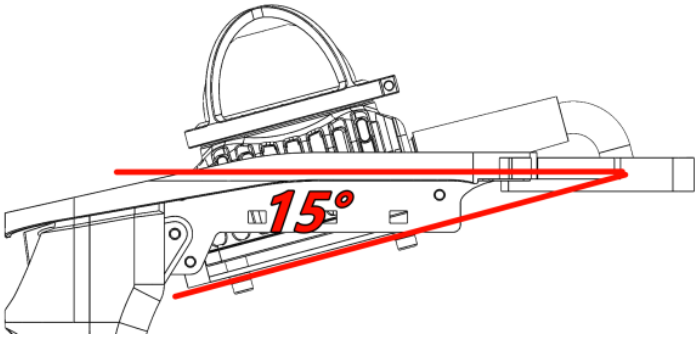


Figure 6.9 LiDAR plane forms an 80° angle with the horizontal plane

Start the node

To start the camera, you need to access the Intel Mini PC, whose IP address can be found in the “Network and Access” section and is 192.167.12.1.

```
ssh robot@192.168.12.1    # Passcode 123
```

Execute the command under the **ros2_packages** directory.

```
source ./setup.bash
ros2 launch livox_lidar_node start_node.launch.py
```

You should see the following output:

Keep the current terminal open, then open a new terminal to run the command to check whether the node has started correctly. If it is running normally, you should see the following new topics:

```
/livox_Lidar_node/description
/livox_Lidar_node/sn105/imu/raw_data
/livox_Lidar_node/sn105/xyz/pointcloud
```

> **NOTE**
>
> The radar node needs to be accessed on the onboard **minipc** to open and subscribe to topics there. Subscribing to topics on the **jetson orin** may lead to significant bandwidth degradation issues.

Node configuration

LiDAR node configuration file directory: */home/robot/robot_controller_release/ros2_packages/livox_lidar_node/share/livox_lidar_node/config*. The LiDAR node configuration file is livox_lidar_node_parameters_setting.yaml, and its main contents are as follows:

```
livox_lidar_node:
ros__parameters:
system_monitor_topic: "system_monitor_defined"
robot_base_frame_id: "robot_base_frame_defined"
is_custom_pointcloud: true
```

The contents of the LiDAR parameter configuration file **lidar_parameters.json** are as follows:

```
{
"MID360": {
"lidar_net_info" : {
"cmd_data_port"    : 56100,
"push_msg_port"    : 56200,
"point_data_port": 56300,
"imu_data_port"    : 56400,
```



```
08    "log_data_port"    : 56500
09    },
10    "host_net_info" : [
11    {
12    "host_ip"              : "192.168.1.20",
13    "lidar_ip"             : ["192.168.1.190"],
14    "cmd_data_port"    : 56101,
15    "push_msg_port"    : 56201,
16    "point_data_port": 56301,
17    "imu_data_port"    : 56401,
18    "log_data_port"    : 56501
19    }
20    ]
21    }
22    }
```

The **<"host_ip": "192.168.1.10">** field is used to configure the host IP for starting the LiDAR, and the **<"lidar_ip": ["192.168.1.183"]>** field specifies the LiDAR IP (default is 192.168.1.1 plus the last two digits of the LiDAR serial number). The LiDAR serial number can be found above the QR code at the back of the LiDAR installation location.

In addition, the main meanings of the parameters in the configuration file are as follows:

- system_monitor_topic：Robot monitoring node name
- robot_base_frame_id： Robot base coordinate frame name
- is_custom_pointcloud: LiDAR custom/standard point cloud format switch flag

Topic publishing and subscribing

After the node starts, it will register several related topics. Once the topics are published, users can visualize the data using tools like Rviz. The topics and their meanings are as follows:

1. /livox_Lidar_node/description

LiDAR node description and topic publishing: This mainly introduces the main content of the LiDAR node. Its message type is std::msg::String and is organized in JSON format. The specific content can be viewed by converting the topic message into JSON format.

2. /livox_Lidar_node/snXXX/xyz/pointcloud

LiDAR point cloud information, topic publishing: The LiDAR point cloud is organized as XYZ, without color information.

3. /livox_Lidar_node/snXXX/imu/raw_data

LiDAR IMU information, topic publishing.

4. /system_monitor_defined

Monitoring messages, topic subscribing: Subscribes to messages from the monitoring node, used to execute commands from the control node.

5. /tf_static

LiDAR static TF information, topic publishing: Publishes LiDAR TF information, mainly the transformation from the robot coordinate frame to the LiDAR link.

6. /tf

LiDAR dynamic TF information, topic publishing: Publishes LiDAR TF information, mainly the transformation from the LiDAR IMU to the LiDAR link.

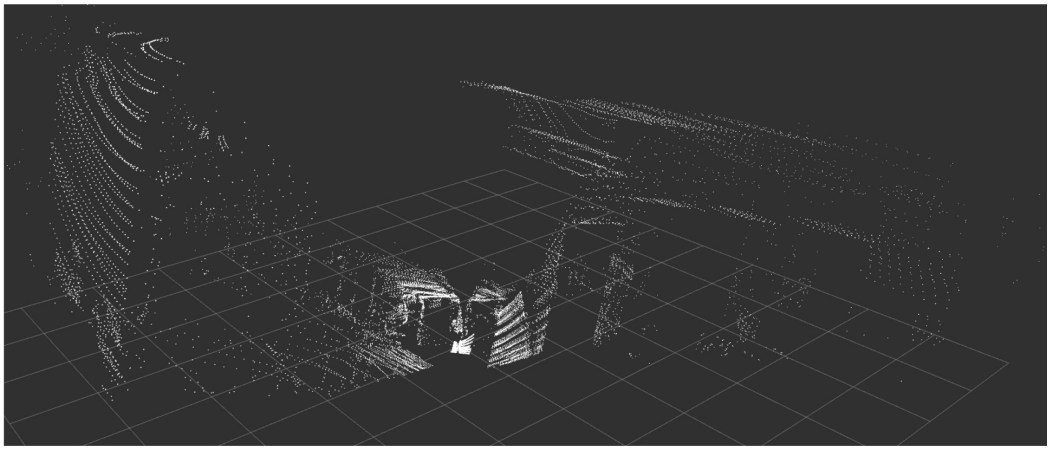


Figure 6.10 Rviz

Program subscribes to topics

The files are located in *dobot_hex_sdk/livox/src/*, with the Python version in *dobot_hex_sdk/livox/py/*. This example demonstrates how to subscribe to topic messages using ROS2 custom message types. After starting the depth camera node according to the above documentation, create a build andcompile it in the livox directory, then execute the two executable files.

1) **`point_cloud2`**，The message type used is **sensor_msgs::msg::PointCloud2**, which is the data type for point clouds obtained from LiDAR scans. Under normal conditions, users should see the point cloud publishing frequency (10 Hz) and the number of points.

```
1  auto qos = rclcpp::SensorDataQoS();
2  sub_ = this->create_subscription<sensor_msgs::msg::PointCloud2>(m_strPointCloudTopicName,
   qos, std::bind(&PCLiteFPSCount::cloudCallback, this, _1));
```

2) **`Imu`**，The message type involved is **`sensor_msgs::msg::Imu`**, which is the data type for the IMU on the LiDAR. When this example program runs normally, users should be able to see it.

```
1  sub_ = this->create_subscription<sensor_msgs::msg::Imu>(imu_topic_, qos,
   std::bind(&LivoxImuLite::imuCallback, this, _1));
```

6.5 Mapping example

The user downloads **dobot_hex_mapping** on the local machine. This example uses the Fast-LIO2 method as a demonstration. When performing mapping with Fast-LIO2, IMU data is used for state propagation, allowing the system to estimate continuous changes in position and orientation between two consecutive point cloud frames, while point cloud data serves as observations for posterior correction, comparing predicted results with actual measurements. Similar to an extended Kalman filter, during the prediction phase, the IMU provides high-frequency acceleration and angular velocity data, which the algorithm integrates to estimate the robot's instantaneous motion. During the update phase, point cloud data is used to correct accumulated errors from prediction via matching and map constraints, producing a stable and accurate trajectory map.

In practical operation, Fast-LIO2 relies on several external components, including Livox-SDK2, livox_ros_driver2, and GTSAM.

6.5.1 Environment dependencies

Livox-SDK2

Livox-SDK2 can be used to communicate with Livox series LiDARs, parsing the LiDAR's output binary data into standard formats such as point coordinates, timestamps, etc. Execute the following commands in any directory:

```
git clone https://github.com/Livox-SDK/Livox-SDK2.git
cd ./Livox-SDK2/
mkdir build
cd build
cmake .. && make -j
sudo make install
```

Livox_ros_driver2

livox_ros_driver2 is a ROS2-based driver for Livox LiDARs. It relies on Livox-SDK2 to acquire the underlying data and then converts this data into standard ROS2 topics, such as converting point cloud data decoded by SDK2 into **sensor_msgs/PointCloud2** messages. Execute the following commands in any directory:

```
mkdir -p ws_livox/src
git clone https://github.com/Livox-SDK/livox_ros_driver2.git ws_livox/src/livox_ros_driver2
cd ws_livox/src/livox_ros_driver2
source /opt/ros/humble/setup.sh
./build.sh humble

```

GTSAM

GTSAM is an open-source factor graph optimization library responsible for fusing and optimizing observations such as IMU and point cloud data. Through factor graph modeling, nonlinear optimization, and smoothing filters, it ultimately outputs high-precision poses and maps. Execute the following commands in any directory:


```
git clone https://github.com/borglab/gtsam
mkdir build && cd build
cmake ..
make -j
sudo make install
```

6.5.2 Compile the program

Prepare environment dependencies before compilation

```
source /opt/ros/humble/setup.bash
source ws_livox/install/setup.bash
```

Execute the compilation command

```
cd dobot_hex_mapping
mkdir build && cd build
cmake ..
make -j
```

6.5.3 Data acquisition

After connecting to the Hexplorer, the data acquisition script is located at */home/robot/Documents/record.sh.* Start the Hexplorer and enter force-control walking mode, then run the data acquisition script and control the Hexplorer to walk. The collected data will be saved in the same directory as the script, with timestamp identifiers.

After data collection, download the metadata configuration file and the data files to the **record** directory under **dobot_hex_mapping** on the local machine. Next, prepare to run the mapping program.

6.5.4 Run mapping

Before running, ensure that the runtime library paths of the program are known.

```
echo "/usr/local/lib" | sudo tee /etc/ld.so.conf.d/usr-local-lib.conf >/dev/null
sudo ldconfig

```

Then, run the mapping program.

```
./fastlio2/lio_mapping <bag_path> <config_file_path>
```

For example

```
./fastlio2/lio_mapping ../record/ ~/dobot_hex_mapping/fastlio2/config/lio_YJ.yaml
```

Open Rviz2 and import the path and point_cloud topics to visualize the trajectory and point cloud. After the program finishes, the mapping results can be found in the record directory and can be visualized using pcl-viewer, for example:

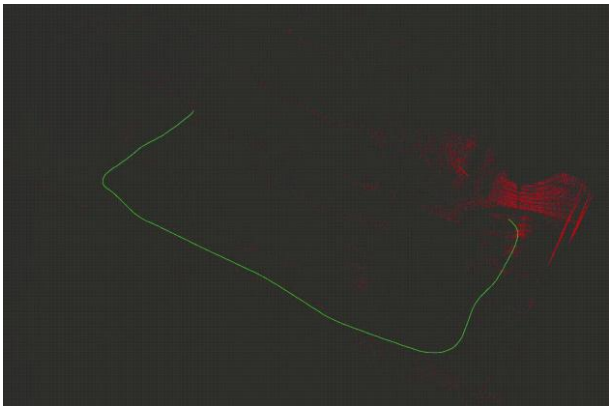


Figure 6.11 Rviz visualization

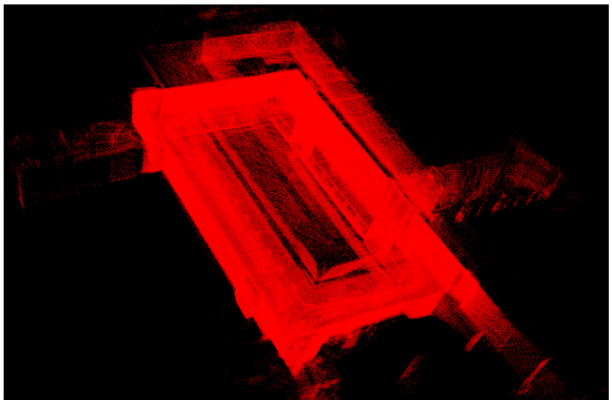


Figure 6.12 PCL-Viewer visualization

7. Software and hardware upgrades

Robot software integration update package, generally named as *inffni_robotics_update_package_${platform}_${robot_id}_${version}.bin.* For example, *inffni_robotics_update_package_mini_pc_mini_hex_v1.6.bin* is the update package for the hexapod robot with the main control platform **mini_pc** and version 1.6. When upgrading the robot software, ensure that the robot is in damping mode! The robot system will reboot during the software upgrade process.

7.1 Windows System Program Update Method

1) Switch the robot state to damping mode
2) In the file explorer, enter **\\192.168.12.1.** Upon successful connection, a user login interface will appear with username: **robot** and password: **123**. You will then enter the robot update program shared folder.

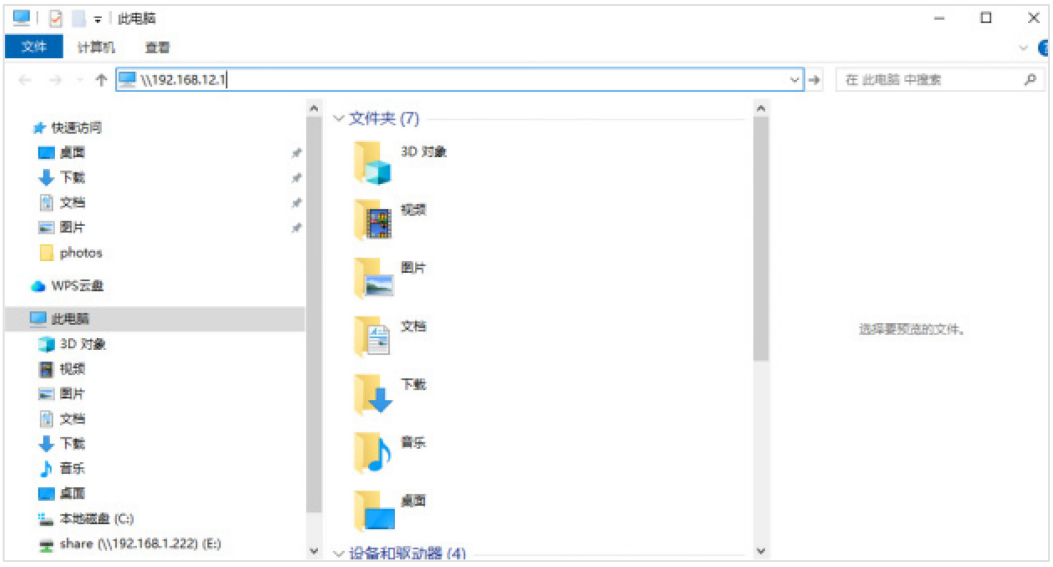
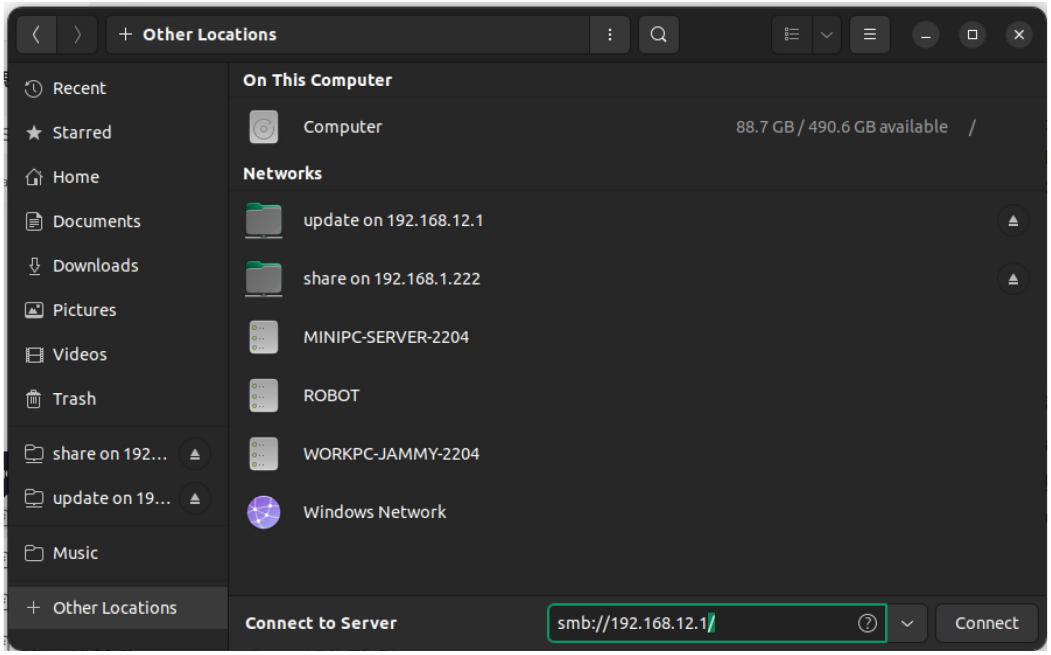


Figure 7.1 Samba shared folder login interface

3) Place the update program *inffni_robotics_update_package_${platform}_${robot_id}_${version}.bin* into the folder.
4) Shut down and restart the robot, or manually enter the robot control system to reboot the robot.
5) Wait for the robot upgrade to complete. After the upgrade, log in again to the robot update shared folder. Upon successful upgrade, there will be a version information file version.txt. This file shows the software version corresponding to the files in the update package. Please do not delete this file!

7.2 Ubuntu System Program Update Method

1) Switch the robot state to damping mode
2) In the file explorer, enter **smb://192.168.12.1/**. Upon successful connection, a user login interface will appear with username: **robot** and password: **123**. You will then enter the robot update program shared folder.



3) Place the update program *inffni_robotics_update_package_${platform}_${robot_id}_${version}.bin* into the folder.

4) Shut down and restart the robot, or manually enter the robot control system to reboot the robot.

5) Wait for the robot upgrade to complete. After the upgrade, log in again to the robot update shared folder. Upon successful upgrade, there will be a version information file version.txt. This file shows the software version corresponding to the files in the update package. Please do not delete this file!

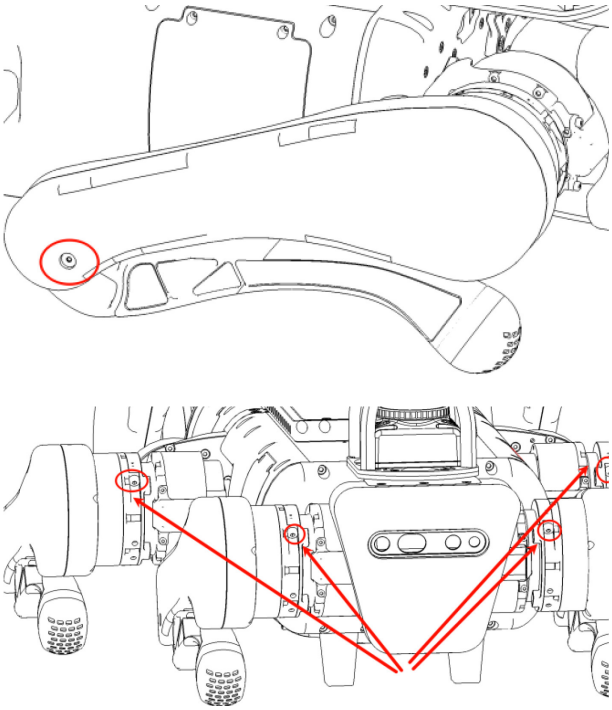
8. Routine Maintenance and Care

8.1 Mechanical inspection

1) Mainly check for any loose screws on the legs, as shown in the figure (inspect once a month).

   a. **Knee joint bolts on all six legs:** If loose, use a T10 Torx bit and tighten with a torque wrench to 2 Nm.

   b. **M4 cup-head hex screws connecting the knee motor and thigh motor (two per leg):** If loose, use an H3.0 hex bit and tighten with a torque wrench to 3 Nm.



2) If abnormal noise increases inside the thigh or at the knee joint pivot, it may indicate grease depletion and require disassembly for maintenance.

3) Regularly inspect and replace excessively worn foot ends.

4) Periodically contact after-sales support to check the condition of the joint motors; technical support staff will assist with inspection and recalibration. If any damage is found during regular checks, please contact after-sales support.

8.2 Controller battery check

Press the controller power button to check if the red light flashes. A red light indicates low battery and requires charging (check once a week).

8.3 Regular cleaning

1) Robot shell: Use clean water or a mild detergent to remove any dust and dirt observed on the robot shell.

2) Camera lens: Use a non-abrasive cloth moistened with a mild cleaning solution to remove possible dirt.
3) LiDAR cover: Use a non-abrasive cloth moistened with a mild cleaning solution to remove possible dirt.
4) LiDAR protective cover: Regularly check whether the LiDAR protective cover is intact and free of cracks.
5) Electrical interfaces: Damaged ports or connectors, or foreign objects, may pose electrical hazards.
6) Ventilation openings: Check for dust blockages to prevent overheating failures.

9. Frequently Asked Questions (FAQ)

9.1 Hardware issues

How to determine if the device is online?

1. After turning on the switch for 2 seconds, listen for a motor humming sound, indicating that both the motors and the main controller are powered on.
2. After about 40 seconds, turn on the controller and use it to control the robot, checking whether the robot enters the folded position mode.

Battery power issues

- **Below 40 V:** Battery total voltage is too low, triggering protection; the battery pack will restart.
- **Below 42 V:** Control program protection is activated; the robot enters damping mode.
- **Below 47 V:** LCD screen shows 0%.
- **57 V:** LCD screen shows 100%.

9.2 Software and algorithm issues

Joint zeroing

1. Power on the robot from the starting position (robot lying on the ground).
2. First, kill the auto-start script start_controller, then kill the motion program.

```
ps -ef | grep start_controller.sh
sudo kill <progress ID>
ps -ef | grep main
sudo kill <progress ID>
```

3. Run the calibration program.

```
cd ~/robot_controller_release/executable
./motor_set_zero miniHex2
```

4. Under normal conditions, the position (pos) of each joint should be below 1e-2 as displayed on the console.

Joint calibration

1. Power on the robot from the starting position (robot lying on the ground).
2. First, kill the auto-start script **start_controller**, then kill the motion program.

```
ps -ef | grep start_controller.sh
sudo kill <progress ID>
ps -ef | grep main
sudo kill <progress ID>
```

3. Run the calibration program.

```
cd ~/robot_controller_release/executable
```



```
2 ./motor_test_calibration miniHex_v2
```

4. Move the joint to be calibrated from the maximum limit to the minimum limit, repeating twice.
5. Copy the offset data for the corresponding joint and replace the corresponding offset data in the YAML file.

