RBQ Software Guide

Table of Contents

1. Developers Guide

1.1 System Diagram

1.2 Developer's Guide

1.3 RBQ API Overview

1.4 DOCKING_STATE

1.5 RBQ GYM

1.6 RBQ-Lab

2. SDK Examples (c/c++)

2.1 🚀 Simple-motion

2.2 Simple-command

2.3 Simple-RL

3. Release Note

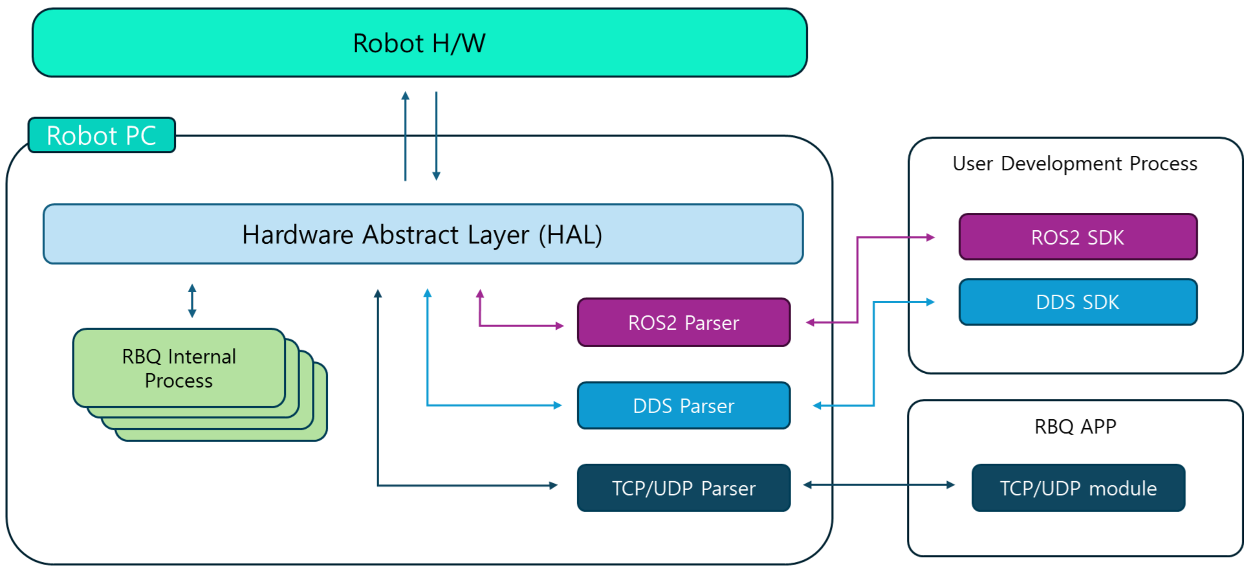
3.1 How to update software

3.2 🖥️ How to Connect to the RBQ Development PC

4. Troubleshooting

4.1 Error Reporting

1.1 System Diagram



SDK

Overview

The **RBQ SDK** is built to provide developers with a flexible and scalable interface for working with the **RBQ quadruped robot**.

It supports both **low-level control** and **high-level commands**, so whether you're building your own controller or integrating the robot into an existing system, there's a suitable entry point.

Low-Level Access

The **LV0 SDK** offers direct access to the robot's sensors and actuators.

It is intended for developers who need **precise control** and **real-time feedback** from the hardware.

✅ Key Features:

- Read raw sensor data: IMU, joint encoders, etc.
- Send joint-level commands: position, velocity, torque
- Ideal for:
  - Custom controller development
  - Low-latency real-time experiments
  - Research in locomotion control

🎯 High-Level Command Interface

The **LV1 SDK** provides a **simplified interface** to control the robot using predefined actions and commands, without needing to manage low-level details.

✅ Key Features:

- Execute common motions and gait behaviors with a single command
- Seamless integration with the **RBQ APP** command set
- APIs available in:
  - C/C++
  - Python
  - ROS2
- Ideal for:
  - Application developers
  - System integration
  - Rapid prototyping with minimal setup

Supported Communication Interfaces

The SDK supports multiple communication protocols depending on your use case:

Interface	Description
ROS2 SDK	For robotics developers in the ROS2 ecosystem
DDS SDK	For real-time, distributed communication using DDS
TCP / UDP	Lightweight direct packet communication for custom tools

1.2 Developer's Guide

📚 Introduction

Welcome to the RBQ simulation, command, and deploy/update manual

This manual provides a comprehensive guide on how to **simulate** the RBQ robot, command it, and **deploy** your own algorithms to the real robot.

✅ 🖥️ Ready-to-use Simulator and Graphical User Interface for developers

- The RBQ robot can be simulated accurately using **MuJoCo**: "A fast and accurate physics engine suitable for advanced robotics simulation."

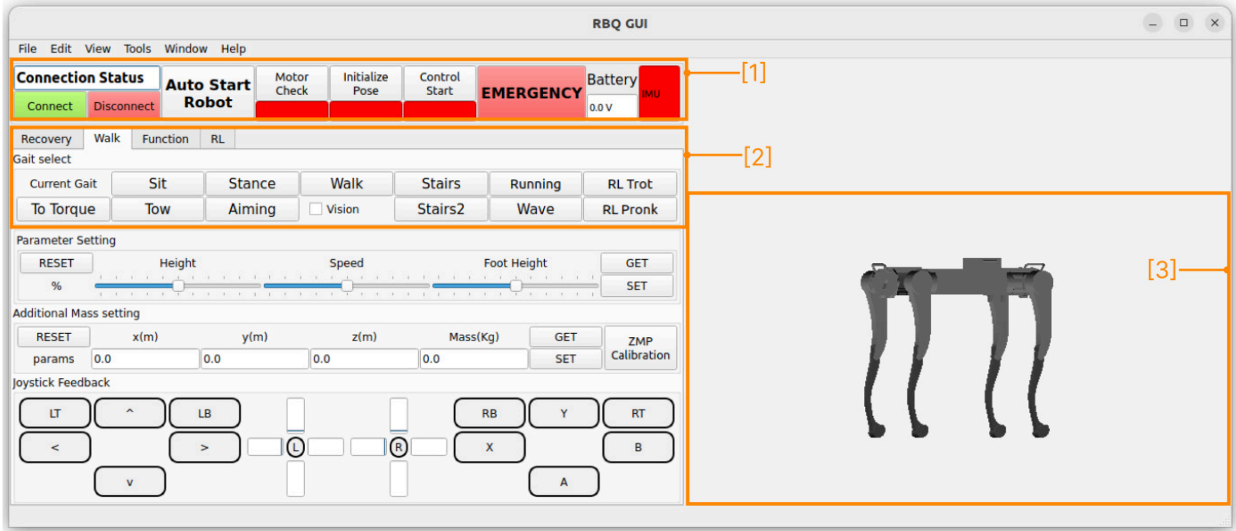
  📝 *Developers can modify the simulation environment "playground" by editing the file:* `resources/model/environment.xml`

- The **RBQ GUI** is designed to provide users with an intuitive interface for commanding and monitoring the robot.
  - It provides visual feedback of the robot status, sensor states, and allows both keyboard and joystick inputs for commands.

  ▶ Open on YouTube (click)

GUI Overview

- This section explains how to use the **RBQ GUI**, including its key features and interface layout.



- [1] : TOP Panel
- [2] : Central Panel
- [3] : 3D Panel

📌 Top Panel

The top panel provides the core functionality for connecting to the robot, initializing its pose, and checking motor status. It also includes a button for emergency stops and indicators for battery voltage and IMU sensor status.

- **Connection Status and Buttons:** Shows whether the robot is connected. You can connect or disconnect using these buttons.
- **Auto Start Robot** button: Automatically initiates the robot [Motor Check → Initialize Pose → Control Start] sequentially and prepares the system for control.
- **Motor Check / Initialize Pose / Control Start** buttons: Perform key startup actions separately. Although it's recommended to use the **Auto Start Robot** button instead.
  📝 *In simulation: only Control Start shows green. Meanwhile, on the real robot: all three button indicators turn green when ready.*
- EMERGENCY button: Immediately halts all robot actions so the robot will softly fall down.

- To resume operation after an emergency stop, press the **Stance** button from the **Gait select** group box.

  📝 *The system enters automatic RECOVERY mode (yellow light) before transitioning to stance motion (green light).*

▶ Open on YouTube (click)

📌 Central Panel

- The central panel includes tabs with buttons for sending various commands to the robot.

  *On a **real robot**, additional **Joint, Firmware, Power** tabs are available, although it is not recommended to use them.*

- **[Walk tab]** commands the robot's walking behavior and transitions between gaits.

✅ Button Action	Sit	Stance	Walk	Stairs	Running	RL-Trot	RL-Pronk
⌨️ Keyboard key	1	2	3	4	5	R	F

3D Panel

- This panel displays a 3D rendered digital twin of the current posture of the connected robot.

📦 Installation

- This section explains how to install and set up the RBQ robot simulation and development environment from scratch.

💻 Requirements

To run the RBQ simulation and deployment environment smoothly, the following requirements are recommended:

- OS: Ubuntu 22.04 (x86)

- Minimum PC specs:
  - CPU: Intel Core i7 - 12th Gen
  - RAM: 16 GB
  - Storage: 25 GB
- Control Devices
  - Keyboard: W, A, S, D, Q, E, R, F, [1-5] keys
  - Joystick: Logitech Wireless Gamepad F710 (Recommended)

📥 1. Download the RBQ Repository

- Clone the repository using the following command:

```bash
git clone https://github.com/RainbowRobotics/RBQ.git
```

🛠️ 2. Set up Environment

- Install necessary libraries using the following command:

```bash
bash scripts/ex/install/setup.bash
```

▶️ Simulation

- Start the **RBQ Control Modules, MuJoCo simulator, and GUI** all together with the help of a script.

```bash
bash scripts/sim.bash --vision
```

NOTE: `ARGUMENTS:`

- `--help` : Display help message and exit.
- `--vision` : Run vision modules (e.g., vision sensors, climb stairs enabled).
- If prompted, type your `user password` in the newly opened CLI `terminals` to start the processes.

🔗 Initialize - Connect and Start the Simulated Robot

- In the `RBQ GUI`, click the **Connect** button to establish the connection.
  - If successful, the `connection indicator` will change to **Simulator Connected** text and color **blue**.
- In the `RBQ GUI`, click the **Auto Start Robot** button to initiate control.
  - If successful, the `control indicator` will change to color **green**.

📌 Stance - Command the Robot to Stand Up

- Path: `"Walk" tab --> "Gait select" group-box --> "Stance" button` or simply press keyboard button `2`
- Once commanded, the robot will start moving into a standing posture. After it is done, it will respond to further commands.
  - 📝 *It might take a second to transition into a standing posture, depending on the current posture.*

🕹️ Joystick	Command	Keyboard key	Robot Action
L Stick	Forward / Backward	W / S	Nose up / down
L Stick	Left / Right	Q / E	Roll Left / Right
R Stick	Forward / Backward		Height up / down
R Stick	Left / Right	A / D	Turn Left / Right
		Shift	Full range of motion

- *If a gamepad is detected, only the number keys of the keyboard are bound.*

📌 Walk - Command the Robot to Walk

- Path: `"Walk" tab --> "Gait select" group-box --> "Walk" button` or simply press keyboard button `3`
- Once commanded, the robot will transition to a walking gait. After it is done, it will respond to further commands.

🕹️ Joystick	Command	Keyboard key	Robot Action
L Stick	Forward / Backward	W / S	Go Forward / Backward
L Stick	Left / Right	Q / E	Go Left / Right
R Stick	Forward / Backward		Pitch Forward / Backward
R Stick	Left / Right	A / D	Turn Left / Right
		Shift	Full speed of motion

- *If a gamepad is detected, only the number keys of the keyboard are bound.*

📌 Stairs - Command the Robot to Climb Stairs

- Path: `"Walk" tab --> "Gait select" group-box --> "Stairs" button` or simply press keyboard button `4`
- The robot utilizes its vision capabilities to climb stairs.

🕹️ Joystick	Command	Keyboard key	Robot Action
L Stick	Forward / Backward	W / S	Go Forward / Backward
L Stick	Left / Right	Q / E	Go Left / Right
R Stick	Forward / Backward		
R Stick	Left / Right	A / D	Turn Left / Right
		Shift	Full speed of motion

- 📝 *If a gamepad is detected, only the number keys of the keyboard are bound.*

📌 Running - Command the Robot to Run

- Path: `"Walk" tab --> "Gait select" group-box --> "Run" button` or simply press keyboard button `5`
- The robot performs rapid and agile movements.
  - *Before transitioning into RUN, transitioning to the STANCE posture might be necessary, depending on the current posture and gait.*

🕹️ Joystick	Command	Keyboard key	Robot Action
L Stick	Forward / Backward	W / S	Go Forward / Backward
L Stick	Left / Right	Q / E	Go Left / Right
R Stick	Forward / Backward		
R Stick	Left / Right	A / D	Turn Left / Right
		Shift	Full speed of motion

- *If a gamepad is detected, only the number keys of the keyboard are bound.*

📌 Wave Walking - Command the Robot to Walk Smoothly

- Path: `"Walk" tab --> "Gait select" group-box --> "Wave" button`
- The robot walks smoothly with a wave gait pattern.

🕹️ Joystick	Command	Keyboard key	Robot Action
L Stick	Forward / Backward	W / S	Go Forward / Backward
L Stick	Left / Right	Q / E	Go Left / Right
R Stick	Forward / Backward		Pitch Forward / Backward
R Stick	Left / Right	A / D	Turn Left / Right
		Shift	Full speed of motion

- 📝 *If a gamepad is detected, only the number keys of the keyboard are bound.*

📌 Aim - Command the Robot to Aim at a Target

- Path: `"Walk" tab --> "Gait select" group-box --> "Aiming" button`
- Use this mode to aim the robot precisely toward a specific target or direction.

Joystick	Command	Keyboard key	Robot Action
L Stick	Forward / Backward	W / S	Nose up / down
L Stick	Left / Right	Q / E	
R Stick	Forward / Backward		
R Stick	Left / Right	A / D	Turn Left / Right
		Shift	Full speed of motion

- *If a gamepad is detected, only the number keys of the keyboard are bound.*

📌 RL-Trot - Command the Robot to RL-Trot

- Path: `"Walk" tab --> "Gait select" group-box --> "RL-Trot" button` or simply press keyboard button `R`
- This mode activates a trot gait generated by a reinforcement learning policy. The robot transitions into the RL-based walking gait and becomes responsive to user commands afterward.

🕹️ Joystick	Command	Keyboard key	Robot Action
L Stick	Forward / Backward	W / S	Go Forward / Backward
L Stick	Left / Right	Q / E	Go Left / Right

R Stick	Forward / Backward		Pitch Forward / Backward
R Stick	Left / Right	A / D	Turn Left / Right
		Shift	Full speed of motion

- 📝 *If a gamepad is detected, only the number keys of the keyboard are bound.*

📌 RL-Pronk - Command the Robot to RL-Pronk

- Path: `"Walk" tab --> "Gait select" group-box --> "RL-Pronk" button` or simply press keyboard button `F`
- This mode activates a pronk gait generated by a reinforcement learning policy. The robot transitions into the RL-based pronking gait and becomes responsive to user commands afterward.

🕹️ Joystick	Command	Keyboard key	Robot Action
L Stick	Forward / Backward	W / S	Go Forward / Backward
L Stick	Left / Right	Q / E	Go Left / Right
R Stick	Forward / Backward		Pitch Forward / Backward
R Stick	Left / Right	A / D	Turn Left / Right
		Shift	Full speed of motion

- *If a gamepad is detected, only the number keys of the keyboard are bound.*

📌 Sit - Command the Robot to Sit Down

- Path: `"Walk" tab --> "Gait select" group-box --> "Sit" button` or simply press keyboard button `1`
- Command the robot to smoothly transition into a sitting posture.
  - *It might take a second to transition into a sitting posture, depending on the current posture.*

🤖 Real Robot

This section explains how to operate the **real RBQ robot**.

📖 *For detailed operation steps, refer to the **Operation Guide***

1. Connect to the Robot

- Connect your PC to the robot's Wi-Fi.

  📝 *Wi-Fi SSID:* `RBQ_xxxx`.

2. Launch the GUI and Set Robot Control

Start the `GUI` in the `bin` directory:

```bash
cd <your workspace>/RBQ
```

```bash
./bin/GUI
```

- Click the **Connect** button.
  - If successful, the `connection indicator` will change to **Robot Connected** text and color **green**.
- Click the **Auto Start** button.
  - If successful, the `control indicator` will change to color **green**, indicating that the robot has been successfully activated and is now ready for operation.

3. Commands are exactly the same as in simulation

- 📝 *It is strongly recommended to practice commanding the robot in the simulator first.*

🚀 Deploy Applications (Binaries) to the Robot

- Run the following command on your development PC to deploy binaries to the Robot:

NOTE: `ARGUMENTS:`

- `--help` : Display help message and exit.
- `--scripts` : deploy new scripts

```bash
bash scripts/deploy.bash
```

You will be prompted for an SSH password.

- **Password:** Please ask the person in charge for the SSH password.
- ⚠️ Important: Once `deploy.bash` has completed, **restart** the robot by turning it off and on to run with the newly deployed binaries.

1.3 RBQ API Overview

The **RBQ API** provides a comprehensive C++ interface for developers to control and interact with the RBQ quadruped robot. It is designed to support both low-level real-time control and high-level behavioral commands in a multi-process environment.

C++ Documentation

View C++ Documentation

Process ID (Motion Ownership)

When creating an `RBQ_API` instance, a **process ID** must be provided:

```cpp
RBQ_API api(20);  // Process ID between 20 and 39 for user applications
```

This ID ensures safe multi-process control using a **motion ownership mechanism**, where each joint can only be controlled by its owner process.

> Always call setMotionOwner() before sending joint commands.

API Structure Summary

Category	Component	Description
Control	`Joint`	Low-level joint control: position, torque, gains (Kp/Kd)
	`startMoveJoint()` / `stopMoveJoint()`	Start/stop jog motion per joint

Category	Component	Description
	`lockUnlockJoint()`	Locks/unlocks joints for manual manipulation
Sensor	`IMU`	Access to orientation (quaternion/RPY), gyro, and acceleration
	`getBatteryVoltage()`	Reads battery voltage (internal only)
Gamepad	`Gamepad`	Access Logitech F710 gamepad inputs: jogs, triggers, buttons
High-level Motion	`motionStaticReady()` / `motionDynamicWalk()` etc.	Predefined robot behaviors

Joint Control API (`RBQ_API::Joint`)

Reading Sensor Values

Function	Description
`getPos()`	Read joint position (rad)
`getVel()`	Read joint velocity (rad/s)
`getTorque()`	Read joint torque (Nm)
`getGainKp()` / `getGainKd()`	Read current Kp/Kd gains

Sending Commands

Function	Description
`setMotionOwner()`	Take control of a joint
`setPosRef()`	Set joint position target
`setTorqueRef()`	Set torque target (Nm)
`setGainKp()` / `setGainKd()`	Set control gains with quantization

> ✅ Each joint must have an owner process to receive control commands.

IMU Sensor API (`RBQ_API::IMU`)

Provides access to robot orientation and motion state.

Function	Output
`getQuaternion()`	Orientation as Eigen quaternion (w, x, y, z)
`getRPY()`	Roll, Pitch, Yaw (ZYX Euler angles, radians)
`getGyro()`	Angular velocity in rad/s (X, Y, Z)
`getAcc()`	Linear acceleration in m/s^2 (X, Y, Z)

Gamepad API (`RBQ_API::Gamepad`)

Supports **Logitech F710 (X mode)** for teleoperation.

Type	Function	Description
Joystick	`getLeftJogX/Y()` , `getRightJogX/Y()`	Range: [-1.0, 1.0]
Trigger	`getLeftTrigger()` , `getRightTrigger()`	Range: [0.0, 1.0]
Button	`getButtonState(Button::X)`	Returns `true` if pressed

Built-in Motion Programs

Easily trigger predefined robot behaviors:

Function	Description
`motionStaticReady()`	Move to ready pose
`motionStaticGround()`	Lie down to ground pose
`motionDynamicWalk()`	Start dynamic walking
`motionDynamicRun()`	Start dynamic running
`motionDynamicAim()`	Aim pose for manipulation/vision

Function	Description
`motionDynamicStairs()`	Stair climbing motion
`motionDynamicHealthCheck()`	Diagnostic motion

📘 RBQ Introduction

Welcome to the **RBQ ROS 2 SDK Manual**.

- This guide will help you quickly get started with controlling and visualizing the RBQ robot through ROS 2, both in simulation and on real hardware.

What This SDK Provides

> This SDK enables the following key features:

- **Robot Control**: Execute posture commands, gait switching, and position-based navigation.
- **Sensor Feedback Streaming**: Continuously stream IMU, foot contact, battery state, and more.

- **Vision Data Streaming**: Provides visual information from front, rear, and downward-facing cameras for perception and navigation.
- **RViz2 Visualization**: Visualize robot state and sensors like LiDAR in real-time.

💻 System requirements

Component	Require Setting
Operating System	Ubuntu 22.04
ROS 2 Distro	ROS2 Humble
Python Version	python 3.8|3.9|3.10

🛠️ Installation

📝 *Instructions assume that the CLI terminal commands are all input in the **main directory**.*

- Clone the RBQ repository to your workspace:

```bash
git clone https://github.com/RainbowRobotics/RBQ.git
```

System Setup

> This "System Setup" process should be executed on the **'developer PC'**. This package is tested for `Ubuntu 22.04 and ROS 2 Humble`.

- Install `ROS Humble`

```bash
cd <your workspace>/RBQ
sudo bash scripts/ex/install/ros.bash
```

- Source `ROS Humble`

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

- Initialize and `update ros dependencies`.

```bash
sudo rosdep init && rosdep update
```

Package Installation

- Go-to the `ros2` directory and `install ros dependencies`

```bash
cd ros2
rosdep install --from-paths src -y --ignore-src
```

- `Build rbq_ros2` packages

```bash
colcon build --symlink-install
```

▶️ Run nodes

Go to your workspace directory first:

```bash
cd <your workspace>/RBQ
```

- To Run `rbq_driver` on **'real robot'**, execute the following command:

This section explains how to operate the **real RBQ robot**.

📖 *For detailed operation steps, refer to the* ***Operation Guide***

---

1. Connect your PC to the robot's Wi-Fi

📝 *Wi-Fi SSID:* `RBQ_xxxx`.

---

2. Connect to robot pc via SSH

```bash
ssh rbq@192.168.0.10
```

⚠️ **Make Sure Auto Start the robot before RUN Simple-RL**

---

3. Run the process on robot pc

To run the process on the robot PC, follow these steps in order.

1. Connect the network on the **robot PC** using a **C to LAN hub**.
2. **Connect to the robot PC via SSH** (see step 2 above: `ssh rbq@192.168.0.10`).
3. On the connected robot PC, clone the repository and run the same setup as on the developer PC.
   - In your working directory (e.g. `cd ~`), run `git clone https://github.com/RainbowRobotics/RBQ.git`, then go into the cloned **RBQ** directory and perform system setup (ROS Humble install, sourcing, rosdep init, etc.) and package installation (`cd ros2`, then `rosdep install`, `colcon build`) as in the 🛠️ **Installation** section.
4. If you get permission errors, run the command again after entering `sudo`.

> **Note**: The commands below must be run on the **real robot** (the robot PC you connected to via SSH), not on the developer PC.

- To use ROS with the **real robot** from your **'developer PC'**, run:

  ```bash
  cd <your workspace>/RBQ
  source ros2/install/local_setup.bash
  ./scripts/start_ros_driver.bash
  ```

  > **Note**: The commands below must be run on the **developer PC**, not on the real robot (robot PC).

- To use ROS in **simulation** on your **'developer PC'**, run:

  ```bash
  cd <your workspace>/RBQ
  source ros2/install/local_setup.bash
  ```

```
./scripts/sim.bash --ros
```

---

4. DDS setup on developer PC (when sending topics to robot PC)

On the **developer PC**, in the terminal where you send DDS traffic to the robot PC, set the following environment variables. Use this terminal for `ros2 topic list`, `ros2 topic pub`, etc.

- **Domain ID** is fixed at `0`.
- **Network interface** must match your PC. `wlp4s0` is an example; replace it with the interface name you get from `ifconfig`.

```bash
export CYCLONEDDS_URI='<CycloneDDS><Domain><Id>0</Id><General>
<Interfaces><NetworkInterface name="wlp4s0"/></Interfaces></General>
</Domain></CycloneDDS>'
export RMW_IMPLEMENTATION=rmw_cyclonedds_cpp
```

Identify the interface to use from the `ifconfig` output below (e.g. Wi-Fi `wlp4s0`, wired `eth0`), then change the `name="wlp4s0"` part in `CYCLONEDDS_URI` to that name.

To check your interfaces:

```bash
ifconfig
```

5. Test list of active topics

Robot PC: If the driver was started in a terminal with the step 3 environment variables set, you can run `ros2 topic list` in that same terminal.

Developer PC: To send or inspect topics to/from the robot PC, run the following in a terminal where **step 4 DDS setup** (domain 0, Cyclone DDS) is applied:

```bash
source ros2/install/local_setup.bash
ros2 topic list
```

- The RBQ driver includes all topics needed to control the RBQ over ROS 2.

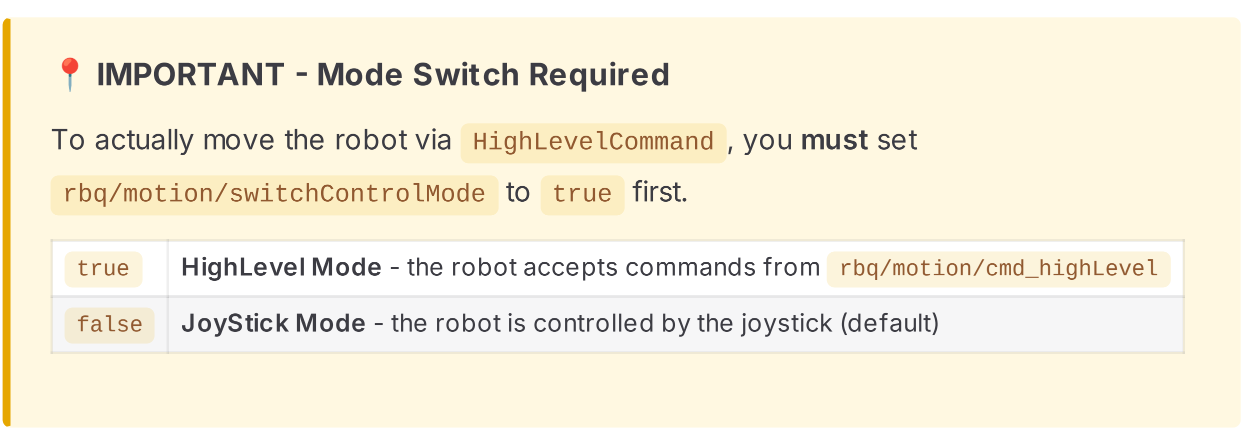
The rbq_driver process starts the following nodes:

- Robot state publisher
- Sensor feedback publishers (IMU, battery, foot contact)
- Command subscriber
- Vision data publishers (front, rear, and bottom camera streams)

⚡ Quick Start

This section walks you through the essential commands to get the robot moving via ROS 2 topics.

1. Switch to HighLevel Command Mode



📍 **IMPORTANT - Mode Switch Required**

To actually move the robot via `HighLevelCommand`, you **must** set `rbq/motion/switchControlMode` to `true` first.

`true`	**HighLevel Mode** - the robot accepts commands from `rbq/motion/cmd_highLevel`
`false`	**JoyStick Mode** - the robot is controlled by the joystick (default)

```bash
ros2 topic pub --once /rbq/motion/switchControlMode std_msgs/msg/Bool "{data: true}"
```

2. Auto Start (initialize joints and start control)

> ⚠️ Before running `autoStart`, ensure the robot is in the correct **initial posture** on a flat surface. For testing, sit the robot first (`switchGait gait_id: 0`).

```bash
ros2 topic pub --once /rbq/motion/autoStart std_msgs/msg/Bool "{data: true}"
```

3. Gait Transition (Sit → Stand → Walk)

Use `rbq/motion/switchGait` with the target `gait_id` to change the robot's gait state.

Sit (gait_id: 0)

```bash
ros2 topic pub --once /rbq/motion/switchGait std_msgs/msg/Int8 "{data: 0}"
```

Stand (gait_id: 1)

```bash
ros2 topic pub --once /rbq/motion/switchGait std_msgs/msg/Int8 "{data: 1}"
```

Walk (gait_id: 3)

```bash
ros2 topic pub --once /rbq/motion/switchGait std_msgs/msg/Int8 "{data: 3}"
```

> Available Gait States :

State ID	Name	Description
-2	Fall Mode	Triggered upon unexpected loss of balance
-1	Control Off	All actuators disabled
0	Sitting	Low posture, resting on the ground
1	Standing	Neutral posture, ready to walk
2	Aim Mode	Aiming posture for targeting
3	Walk Mode	walking Trot gait
4	Stairs Mode	Stair-adaptive gait using camera sensor
5	Wave Mode	walking Walk gait
6	Run Mode	High-speed gait (if supported)

State ID	Name	Description
30	RL Trot	Reinforcement Learning Trot gait
31	RL Front Walk	Reinforcement Learning Front Walk gait
33	RL Left Walk	Reinforcement Learning Left Walk gait
34	RL Right Walk	Reinforcement Learning Right Walk gait
35	RL Bound	Reinforcement Learning Bound gait
36	RL Pace	Reinforcement Learning Pace gait
37	RL Pronk	Reinforcement Learning Pronk gait
38-41	RL 3Leg	Reinforcement Learning 3-Leg gaits (HR, HL, FR, FL)
42	RL Trot Vision	Reinforcement Learning Trot with Vision
45	RL Trot Run	Reinforcement Learning Trot Run gait
46	RL Silent	Reinforcement Learning Silent gait

4. Send Velocity Commands

Once the robot is in Walk Mode (gait_id: 3), use `rbq/motion/cmd_highLevel` to send velocity commands.

Move forward at 0.3 m/s

```bash
ros2 topic pub --once /rbq/motion/cmd_highLevel
rbq_msgs/msg/HighLevelCommand \
'{header: {stamp: {sec: 0, nanosec: 0}, frame_id: "base"},
  identifier: "walk_fwd",
  roll: 0.0,
  pitch: 0.0,
  yaw: 0.0,
  vel_x: 0.3,
  vel_y: 0.0,
  omega_z: 0.0,
  delta_body_h: 0.0,
  delta_foot_h: 0.0,
  gait_state: 3,
  gait_transition: false}'
```

Move diagonally at 45° (forward + lateral)

```bash
ros2 topic pub --once /rbq/motion/cmd_highLevel
rbq_msgs/msg/HighLevelCommand \
```

```
'{header: {stamp: {sec: 0, nanosec: 0}, frame_id: "base"},
  identifier: "walk_diag",
  roll: 0.0,
  pitch: 0.0,
  yaw: 0.0,
  vel_x: 0.3,
  vel_y: 0.3,
  omega_z: 0.0,
  delta_body_h: 0.0,
  delta_foot_h: 0.0,
  gait_state: 3,
  gait_transition: false}'
```

> 💡 For 45° diagonal movement, set the same `vel_x` and `vel_y` (0.3 × cos45° ≈ 0.212 m/s).

> HighLevelCommand message definition :

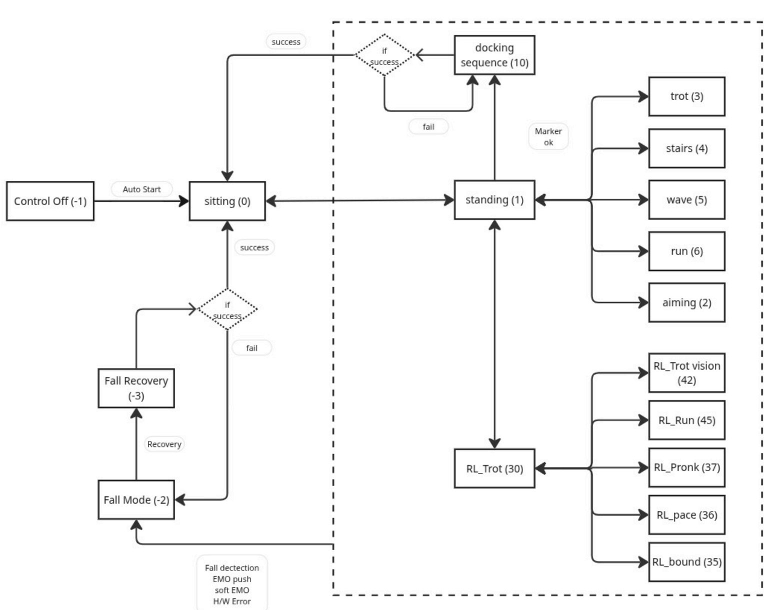
```bash
std_msgs/Header header          # ROS message header with timestamp and frame_id
string identifier               # Command identifier for tracking/logging purposes
float64 roll                    # Body roll angle in degrees (ZYX Euler angles)
float64 pitch                   # Body pitch angle in degrees (ZYX Euler angles)
float64 yaw                     # Body yaw angle in degrees (ZYX Euler angles)
float64 vel_x                   # Linear velocity in X direction (m/s)
float64 vel_y                   # Linear velocity in Y direction (m/s)
float64 omega_z                 # Angular velocity around Z axis (deg/s)
float64 delta_body_h            # Body height adjustment from default (m)
float64 delta_foot_h            # Foot height adjustment from default (m)
int8    gait_state              # Target gait state (use constants defined above)
bool    gait_transition         # Enable(true)/disable(false) gait transition during command execution
```

📋 Topics

1. Gait State Topics

Gait state refers to the robot's current motion or posture mode.

Transitions between these states are triggered by incoming high-level commands, sensor conditions, or internal logic.



> Available Gait States :

State ID	Name	Description
-2	Fall Mode	Triggered upon unexpected loss of balance
-1	Control Off	All actuators disabled
0	Sitting	Low posture, resting on the ground
1	Standing	Neutral posture, ready to walk
2	Aim Mode	Aiming posture for targeting
3	Walk Mode	walking Trot gait
4	Stairs Mode	Stair-adaptive gait using camera sensor
5	Wave Mode	walking Walk gait
6	Run Mode	High-speed gait (if supported)

State ID	Name	Description
30	RL Trot	Reinforcement Learning Trot gait
31	RL Front Walk	Reinforcement Learning Front Walk gait
33	RL Left Walk	Reinforcement Learning Left Walk gait
34	RL Right Walk	Reinforcement Learning Right Walk gait
35	RL Bound	Reinforcement Learning Bound gait
36	RL Pace	Reinforcement Learning Pace gait
37	RL Pronk	Reinforcement Learning Pronk gait
38-41	RL 3Leg	Reinforcement Learning 3-Leg gaits (HR, HL, FR, FL)
42	RL Trot Vision	Reinforcement Learning Trot with Vision
45	RL Trot Run	Reinforcement Learning Trot Run gait
46	RL Silent	Reinforcement Learning Silent gait

> The **Available Commands** in the second column show which commands can be used in each gait state. In contrast, **Common Commands** and **Power Control** are always available regardless of the gait state.
> **Note**: All gait transitions use the unified `rbq/motion/switchGait` topic with specific `gait_id` values.
> Check Gait State table above for available gait_id values at "Available Gait States"

<table>
<tr><th>Gait State</th><th>Available Commands</th><th>Common Commands</th><th>Power Control</th></tr>
<tr><td>Control Off</td><td colspan="2">rbq/motion/autoStart</td><td rowspan="4">rbq/powerControl/setPortState

PDU Port ID
int8 PDU_PORT_48V_LEG = 0x00
int8 PDU_PORT_48V_ADD = 0x01
int8 PDU_PORT_48V_EXT = 0x02
int8 PDU_PORT_12V_VisionPC = 0x10
int8 PDU_PORT_12V_COMM = 0x11
int8 PDU_PORT_12V_Lidar =</td></tr>
<tr><td>Sitting Mode</td><td>rbq/motion/canCheck</td><td rowspan="3">rbq/motion/switchGait
(gait_id: 0-46)

rbq/motion/cmd_highLevel

rbq/motion/emergency

rbq/motion/static</td></tr>
<tr><td>Standing Mode</td><td>rbq/motion/cmd_navigateTo
rbq/stateEstimation/comEstimationCompensation</td></tr>
<tr><td>Walk Mode</td><td>-</td></tr>
</table>

Gait State	Available Commands	Common Commands	Power Control
Stairs Mode	-	Lock rbq/motion/static Ready rbq/motion/static Ground rbq/motion/switchControlMode (true:HighLevel Mode, false:JoyStick Mode)	0x12 int8 PDU_PORT_12V_CCTV = 0x13 int8 PDU_PORT_12V_THER = 0x14 int8 PDU_PORT_12V_IRLed = 0x15 int8 PDU_PORT_12V_Speaker = 0x16 int8 PDU_PORT_5V_CAMERAS = 0x20 int8 PDU_PORT_5V_AUDIO_SIDE_CAM_USBHUB = 0x21
Run Mode	-		
Wave Mode	rbq/motion/cmd_navigateTo		
RL Mode	-		
Fall Mode	rbq/motion/recoveryFlex rbq/motion/switchGait (gait_id: 1)		

📡 2. Low-level State Topics

These topics provide real-time feedback about the robot's internal state, such as posture, sensor readings, and system status.

Topics	Data Structure	Description
rbq/status/robot_status	std_msgs/Header header bool con_start bool ready_pos bool ground_pos bool force_con bool ext_joy bool is_standing bool can_check bool find_home int8 gait_id	This message contains robot status information and is published at 50Hz. Contains motor control status, position states, communication status, gait information, and docking state. Check Gait State table above for available gait_id values at "Available Gait States" # Docking States int8 DOCKING_MAX_FAIL_CNT_REACHED = -6 int8 DOCKING_MARKER_POS_INVALID_ROTATION = -5 int8 DOCKING_MARKER_POS_INVALID_TOO_FAR = -4 int8 DOCKING_MARKER_POS_INVALID_WRONG_DIR = -3 int8 DOCKING_MARKER_NOT_FOUND = -2 int8 DOCKING_FAILED = -1 int8 DOCKING_OPERATION_MODE = 0 int8 DOCKING_APPROACH_OFFSET = 1 int8 DOCKING_APPROACH = 2 int8 DOCKING_APPROACH_WIDE = 3

Topics	Data Structure	Description
	bool is_fall int8 docking_state bool imu_success	int8 DOCKING_SIT_DOWN = 4 int8 DOCKING_SUCCESS = 5 int8 DOCKING_SUCCESS_CHARGING = 6 int8 DOCKING_SUCCESS_NO_CHARGING = 7
rbq/powerControl/battery_status	std_msgs/Header header string identifier float64 charge_percentage float64 current float64 voltage float64[] temperatures uint8 status	# Status uint8 STATUS_UNKNOWN = 0 uint8 STATUS_MISSING = 1 uint8 STATUS_CHARGING = 2 uint8 STATUS_DISCHARGING = 3 uint8 STATUS_BOOTING = 4
rbq/stateEstimation/footStates	std_msgs/Header header geometry_msgs/Point[] foot_position_rt_body geometry_msgs/Point[] foot_velocity_rt_body uint8[] contact	Foot position/velocity with respect to the local body frame. The origin of the body frame is at the center of the robot body, with the front direction as +x, the left direction as +y, and the upward direction as +z. All arrays contain 4 elements (one for each leg). # Contact uint8 CONTACT_UNKNOWN = 0 uint8 CONTACT_MADE = 1 uint8 CONTACT_LOST = 2 # Leg number Leg num 0 : Right-Hind leg (RH) Leg num 1 : Left-Hind Leg (LH) Leg num 2 : Right-Front Leg (RF) Leg num 3 : Left-Front Leg (LF)
rbq/joint/joint_status	std_msgs/Header header bool[] connected int8[] temperature int8[] motor_temp	This message contains detailed joint information including connection status, temperature, motor status flags, and control data. Published at 50Hz for each joint.

RAINBOW ROBOTICS

Topics	Data Structure	Description
	bool[] status_fet bool[] status_run bool[] status_init bool[] status_mod bool[] status_non_ctr bool[] status_bat bool[] status_calib bool[] status_mt_err bool[] status_jam bool[] status_cur bool[] status_big bool[] status_inp bool[] status_flt bool[] status_tmp bool[] status_ps1 bool[] status_ps2 bool[] status_rsvd float32[] position_ref float32[] position_enc float32[] velocity float32[] torque_ref float32[] current	

Topics	Data Structure	Description
	float32[] kp float32[] kd int32[] owner	

> 💡 **Examples** : Subscribe to each state topic

Robot Status

```bash
ros2 topic echo --once rbq/status/robot_status
```

Joint Status

```bash
ros2 topic echo --once rbq/joint/joint_status
```

Battery Status

```bash
ros2 topic echo --once rbq/powerControl/battery_status
```

Foot States

```bash
ros2 topic echo --once rbq/stateEstimation/footStates
```

> The msg about state is defined as :

- RobotStatus

```bash
# DOCKING_STATE
int8 DOCKING_MAX_FAIL_CNT_REACHED = -6          # (ABORT SEQENCE) MAX
Docking retry count rechead (10 times)
int8 DOCKING_MARKER_POS_INVALID_ROTATION  = -5   # (ABORT SEQENCE)
Marker rotation angle is more than +-40 degree
int8 DOCKING_MARKER_POS_INVALID_TOO_FAR   = -4   # (ABORT SEQENCE)
Marker is too far more than 5m
int8 DOCKING_MARKER_POS_INVALID_WRONG_DIR = -3   # (ABORT SEQENCE)
```

```
Marker is detected on front side of robot
int8 DOCKING_MARKER_NOT_FOUND = -2              # (ABORT SEQENCE)
Marker not found
int8 DOCKING_FAILED           = -1              # docking failed -->
Auto retry
int8 DOCKING_OPERATION_MODE   = 0               # robot is on normal
operation
int8 DOCKING_APPROACH_OFFSET  = 1               # 1st reach to charger
by offset
int8 DOCKING_APPROACH         = 2               # 2nd reach to charger
int8 DOCKING_APPROACH_WIDE    = 3               # 3rd reach to charger
with wide stance
int8 DOCKING_SIT_DOWN         = 4               # sitting down to
charger
int8 DOCKING_SUCCESS          = 5               # docking success :
charger connected
int8 DOCKING_SUCCESS_CHARGING = 6               # docking success :
charging
int8 DOCKING_SUCCESS_NO_CHARGING = 7            # docking success :
no_charging

# gait_state constants
int8 STATE_FALL                = -2    # Fall Mode - Triggered upon
unexpected loss of balance
int8 STATE_OFF                 = -1    # Control Off - All actuators
disabled
int8 STATE_SIT                 = 0     # Sitting - Low posture, resting
on the ground
int8 STATE_STAND               = 1     # Standing - Neutral posture,
ready to walk
int8 STATE_AIM                 = 2     # Aim Mode - Aiming posture for
targeting
int8 STATE_WALK                = 3     # Walk Mode - walking Trot gait
int8 STATE_STAIRS              = 4     # Stairs Mode - Stair-adaptive
gait using camera sensor
int8 STATE_WAVE                = 5     # Wave Mode - walking Walk gait
int8 STATE_RUN                 = 6     # Run Mode - High-speed gait (if
supported)
int8 STATE_RL_TROT             = 30    # RL Trot - Reinforcement
Learning Trot gait
int8 STATE_RL_FRONT_WALK       = 31    # RL Front Walk - Reinforcement
Learning Front Walk gait
int8 STATE_RL_LEFT_WALK        = 33    # RL Left Walk - Reinforcement
Learning Left Walk gait
int8 STATE_RL_RIGHT_WALK       = 34    # RL Right Walk - Reinforcement
Learning Right Walk gait
int8 STATE_RL_BOUND            = 35    # RL Bound - Reinforcement
Learning Bound gait
int8 STATE_RL_PACE             = 36    # RL Pace - Reinforcement
Learning Pace gait
int8 STATE_RL_PRONK            = 37    # RL Pronk - Reinforcement
```

```
    Learning Pronk gait
    int8 STATE_RL_3LEG_HR          = 38    # RL 3Leg HR - Reinforcement
    Learning 3-Leg gait (Hind Right)
    int8 STATE_RL_3LEG_HL          = 39    # RL 3Leg HL - Reinforcement
    Learning 3-Leg gait (Hind Left)
    int8 STATE_RL_3LEG_FR          = 40    # RL 3Leg FR - Reinforcement
    Learning 3-Leg gait (Front Right)
    int8 STATE_RL_3LEG_FL          = 41    # RL 3Leg FL - Reinforcement
    Learning 3-Leg gait (Front Left)
    int8 STATE_RL_TROT_VISION      = 42    # RL Trot Vision - Reinforcement
    Learning Trot with Vision
    int8 STATE_RL_TROT_RUN         = 45    # RL Trot Run - Reinforcement
    Learning Trot Run gait
    int8 STATE_RL_SILENT           = 46    # RL Silent - Reinforcement
    Learning Silent gait

    std_msgs/Header header
    bool con_start          # Motor control enabled (1 = enabled, 0 = not
    enabled)
    bool ready_pos          # Robot is in ready position (1 = ready, 0 =
    not ready)
    bool ground_pos         # Robot is in ground/sitting position (1 =
    sitting, 0 = not sitting)
    bool force_con          # Force control mode enabled (1 = enabled, 0 =
    disabled)
    bool ext_joy            # External joystick connected (1 = connected, 0
    = not connected)
    bool is_standing        # Robot is standing (1 = standing, 0 = not
    standing)
    bool can_check          # CAN communication check (1 = success, 0 =
    failure)
    bool find_home          # Encoder homing status (1 = success, 0 =
    failure)
    int8 gait_id            # Current gait mode identifier (returns value
    defined GAIT_STATE enum)
    bool is_fall            # Fall detection status (1 = robot has fallen,
    0 = normal)
    int8 docking_state      # Docking process status (returns value defined
    in DOCKING_STATE enum)
    bool imu_success        # IMU connection status (1 = success, 0 =
    failure)
```

- JointStatus

```bash
std_msgs/Header header

# Joint Detail Information
# Connection and Temperature Status
bool[] connected          # Joint connection status (true = connected,
false = disconnected)
```

```
    int8[] temperature          # Board temperature in Celsius
    int8[] motor_temp           # Motor temperature in Celsius

    # Motor Status Flags
    bool[] status_fet           # FET (Field Effect Transistor) status (true
    = ON, false = OFF)
    bool[] status_run           # Motor running status (true = running, false
    = stopped)
    bool[] status_init          # Initialization status (true = initialized,
    false = not initialized)
    bool[] status_mod           # Control mode status (true = position
    control, false = torque control)
    bool[] status_non_ctr       # Nonius count error (true = error, false =
    normal)
    bool[] status_bat           # Battery status (true = low battery, false =
    normal)
    bool[] status_calib         # Calibration status (true = in calibration
    mode, false = normal)
    bool[] status_mt_err        # Motor error status (true = error, false =
    normal)
    bool[] status_jam           # JAM error status (true = jammed, false =
    normal)
    bool[] status_cur           # Over current error (true = over current,
    false = normal)
    bool[] status_big           # Big position error (true = large error,
    false = normal)
    bool[] status_inp           # Input error (true = input error, false =
    normal)
    bool[] status_flt           # FET driver fault error (true = fault, false
    = normal)
    bool[] status_tmp           # Temperature error (true = over temperature,
    false = normal)
    bool[] status_ps1           # Position limit error - lower bound (true =
    limit reached, false = normal)
    bool[] status_ps2           # Position limit error - upper bound (true =
    limit reached, false = normal)
    bool[] status_rsvd          # Reserved status bit

    # Control Data
    float32[] position_ref      # Reference position in radians
    float32[] position_enc      # Encoder position in radians
    float32[] velocity          # Joint velocity in deg/s
    float32[] torque_ref        # Reference torque in Nm
    float32[] current           # Motor current in Amperes
    float32[] kp                # Proportional gain (P gain)
    float32[] kd                # Derivative gain (D gain)
    int32[] owner               # Joint owner ID (which controller owns this
    joint)
```

- BatteryState

```bash
# Status
uint8 STATUS_UNKNOWN = 0
uint8 STATUS_MISSING = 1
uint8 STATUS_CHARGING = 2
uint8 STATUS_DISCHARGING = 3
uint8 STATUS_BOOTING = 4

std_msgs/Header header
string identifier
float64 charge_percentage
float64 current
float64 voltage
float64[] temperatures
uint8 status
```

- FootStates

```bash
# Contact constants
uint8 CONTACT_UNKNOWN = 0
uint8 CONTACT_MADE = 1
uint8 CONTACT_LOST = 2

std_msgs/Header header

# Arrays for 4 legs (FL, FR, RL, RR)
geometry_msgs/Point[] foot_position_rt_body    # Foot position relative
to body frame (4 elements)
geometry_msgs/Point[] foot_velocity_rt_body    # Foot velocity relative
to body frame (4 elements)
uint8[] contact                                 # Contact state for
each foot (4 elements)
```

🎮 3. Command Topics

The following topics are used to send high-level and navigation commands to the RBQ robot.

All commands follow standard ROS 2 message types or custom `rbq_msgs` definitions.

> Command topics are defined as :

Topics	Description
rbq/motion/autoStart	Initialize Joint position and start joint control AutoStart process can be checked by 'RobotStatus' topic CAN check → Find Home → Control Start flags will be on
rbq/motion/emergency	Emergency stop - sets all leg joints to high damping mode. This action may cause a shock to the robot.
rbq/motion/switchGait	Switch robot gait mode using gait_id parameter. Check Gait State table above for available gait_id values at "Available Gait States".
rbq/motion/recoveryFlex rbq/motion/switchGait (gait_id: 1)	The 'recoveryFlex' topic is accepted only when the robot is in Fall mode. It commands the robot to return to sitting mode through specific joint movements. Alternatively, you can use 'switchGait' topic with gait_id: 1 (Standing) to recover from Fall mode.
rbq/motion/switchControlMode	Enable or disable HighLevel Command mode. When set to `true`, HighLevel Command mode is enabled and the robot accepts commands from `rbq/motion/cmd_highLevel` topic. When set to `false`, Joystick control mode is enabled instead.

> 💡 **Examples** : Publish to each command topic

Auto Start (initialize joints and start control)

> ⚠️ Before running `autoStart`, make sure the robot is in the correct **initial pose** on a flat surface. If you are testing, sit the robot down first (`switchGait gait_id: 0`).

```bash
ros2 topic pub --once /rbq/motion/autoStart std_msgs/msg/Bool "{data: true}"
```

Emergency Stop

> ⚠️ This command sets all leg joints to **high damping** mode - the robot will collapse immediately. If you are testing, sit the robot down first (`switchGait gait_id: 0`) before using this command.

```bash
ros2 topic pub --once /rbq/motion/emergency  std_msgs/msg/Bool "{data:
true}"
```

Switch Gait (e.g. to Walk Mode, gait_id: 3)

```bash
ros2 topic pub --once /rbq/motion/switchGait std_msgs/msg/Int8 "{data:
3}"
```

Recovery from Fall Mode

```bash
ros2 topic pub --once /rbq/motion/recoveryFlex std_msgs/msg/Bool "
{data: true}"
```

Switch to HighLevel Command Mode

```bash
ros2 topic pub --once rbq/motion/switchControlMode std_msgs/msg/Bool
'{data: true}'
```

Switch to JoyStick Mode

```bash
ros2 topic pub --once rbq/motion/switchControlMode std_msgs/msg/Bool
'{data: false}'
```

Static Lock (lock joints in current position)

```bash
ros2 topic pub --once /rbq/motion/staticLock std_msgs/msg/Bool "{data:
true}"
```

Static Ready

```bash
ros2 topic pub --once /rbq/motion/staticReady std_msgs/msg/Bool "{data:
true}"
```

Static Ground

```bash
ros2 topic pub --once /rbq/motion/staticGround std_msgs/msg/Bool "
```

```
    {data: true}"
```

> High-level command topics are defined as :

Topics	Data Structure	Description
rbq/motion/cmd_highLevel	std_msgs/Header header string identifier float64 roll float64 pitch float64 yaw float64 vel_x float64 vel_y float64 omega_z float64 delta_body_h float64 delta_foot_h int8 gait_state	High-level command topic for robot control based on current gait state. (Check Gait State table and gait_id values at "Available Gait States") • **Standing Mode:** Adjusts robot body posture based on roll, pitch, yaw (ZYX Euler angles) , The parameter body_H specifies the height of the robot's body Range: Roll: -25~25°, Pitch: -20~20°, Yaw: -25~25°, Delta_body_h: -0.15~0.05m • **Walk Mode:** Set movement speed of the robot in the local coordinate system while in the trotting gait state. Vx, Vy represents the forward speed, lateral speed and Wz represents rotational speed. During trotting, delta_body_H is used to set the height of the body from its default body height, foot_H is used to set the foot lifting height, and pitch is used to set the pitch angle of the body. Range: Vx: -1.0~1.2m/s, Vy: -0.4~0.4m/s, Wz: -75~75°/s Delta_body_h: -0.15~+0.05m, Delta_foot_h: -0.06~+0.04m • **Stairs Mode:** Forward, lateral, rotational speeds for stairs navigation Range: Vx: -0.5~0.5m/s, Vy: -0.2~0.2m/s, Wz: -15~15°/s • **Run Mode:** Forward, lateral, rotational speeds for High-speed movement control Range: Vx: -1.0~1.8m/s, Vy: -0.6~0.6m/s, Wz: -75~75°/s • **Wave Mode:** Forward, lateral, rotational speeds for Slow movement control Range: Vx: -0.3~0.3m/s, Vy: -0.2~0.2m/s, Wz: -20~20°/s • **RL TROT Mode:** Reinforcement Learning Trot gait with enhanced speed control Range: Vx: -1.5~2.0m/s, Vy: -1.0~1.0m/s, Wz: -75~75°/s • **RL TROT VISION Mode:** Reinforcement Learning Trot with Vision integration Range: Vx: -1.5~2.0m/s, Vy: -1.0~1.0m/s, Wz: -75~75°/s

Topics	Data Structure	Description
	bool gait_transition	
rbq/motion/cmd_navigateTo	geometry_msgs/Pose pose uint8 mode	Navigation command to move robot to specified position: **# Approach Modes:** 0: Rotate to target → straight walk → rotate to target yaw 1: Rotate to target yaw → diagonal walk to target 2: Diagonal walk to target → rotate to target yaw 3: Rotate to target yaw and diagonal walk simultaneously 4: Approach mode 3 with wide leg walking

> 💡 **Examples** : High-level and navigation commands

Walk forward at 0.5 m/s (Walk Mode)

```bash
ros2 topic pub --once /rbq/motion/cmd_highLevel
rbq_msgs/msg/HighLevelCommand '{header: {stamp: {sec: 0, nanosec: 0},
frame_id: "base"},
  identifier: "walk_fwd",
  roll: 0.0,
  pitch: 0.0,
  yaw: 0.0,
  vel_x: 0.5,
  vel_y: 0.0,
  omega_z: 0.0,
  delta_body_h: 0.0,
  delta_foot_h: 0.0,
  gait_state: 3,
  gait_transition: false}'
```

Turn 30°/s (Walk Mode)

```bash
ros2 topic pub --once /rbq/motion/cmd_highLevel
rbq_msgs/msg/HighLevelCommand '{header: {stamp: {sec: 0, nanosec: 0},
frame_id: "base"},
  identifier: "turn",
  roll: 0.0,
  pitch: 0.0,
```

```
  yaw: 0.0,
  vel_x: 0.0,
  vel_y: 0.0,
  omega_z: 30.0,
  delta_body_h: 0.0,
  delta_foot_h: 0.0,
  gait_state: 3,
  gait_transition: false}'
```

Stop in place (zero velocity)

```bash
ros2 topic pub --once /rbq/motion/cmd_highLevel
rbq_msgs/msg/HighLevelCommand '{header: {stamp: {sec: 0, nanosec: 0},
frame_id: "base"},
  identifier: "stop",
  roll: 0.0,
  pitch: 0.0,
  yaw: 0.0,
  vel_x: 0.0,
  vel_y: 0.0,
  omega_z: 0.0,
  delta_body_h: 0.0,
  delta_foot_h: 0.0,
  gait_state: 3,
  gait_transition: false}'
```

Navigate to World Coordinate position (x: 20cm, y: 10cm)

```bash
ros2 topic pub --once /rbq/motion/cmd_navigateTo
geometry_msgs/msg/PoseStamped "
header:
  frame_id: map
pose:
  position:
    x: 0.2
    y: 0.1
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 1.0
"
```

PDU port control mapping (port_id for setPortState)

The port IDs used by the `rbq/powerControl/setPortState` topic match the enum below.

```cpp
enum PDU_PORT_IDs_e : unsigned char {
    PDU_PORT_48V_LEG                   = 0x00,
    PDU_PORT_48V_ADD                   = 0x01,
    PDU_PORT_48V_EXT                   = 0x02,

    PDU_PORT_12V_VisionPC              = 0x10,
    PDU_PORT_12V_COMM                  = 0x11,
    PDU_PORT_12V_Lidar                 = 0x12,
    PDU_PORT_12V_CCTV                  = 0x13,
    PDU_PORT_12V_THER                  = 0x14,
    PDU_PORT_12V_IRLed                 = 0x15,
    PDU_PORT_12V_Speaker               = 0x16,

    PDU_PORT_5V_CAMERAS                = 0x20,
    PDU_PORT_5V_AUDIO_SIDE_CAM_USBHUB  = 0x21,
};
```

PDU port control (e.g. turn on 12V LiDAR port)

12V LiDAR is `PDU_PORT_12V_Lidar = 0x12` (18).

```bash
ros2 topic pub --once /rbq/powerControl/setPortState
std_msgs/msg/Int8MultiArray "{data: [18, 1]}"
```

> The msg of High-Level command is defined as :

- HighLevelCommand

```bash
std_msgs/Header header          # ROS message header with timestamp and
frame_id
string identifier               # Command identifier for
tracking/logging purposes
float64 roll                    # Body roll angle in degrees (ZYX Euler
angles)
float64 pitch                   # Body pitch angle in degrees (ZYX Euler
angles)
float64 yaw                     # Body yaw angle in degrees (ZYX Euler
angles)
float64 vel_x                   # Linear velocity in X direction (m/s)
```

```
    float64 vel_y                    # Linear velocity in Y direction (m/s)
    float64 omega_z                  # Angular velocity around Z axis (deg/s)
    float64 delta_body_h             # Body height adjustment from default
(m)
    float64 delta_foot_h             # Foot height adjustment from default
(m)
    int8    gait_state               # Target gait state (use constants
defined above)
    bool    gait_transition          # Enable(true)/disable(false) gait
transition during command execution
```

> 💡 **Example**: Standing mode with custom pose (roll: 15°, pitch: 30°, yaw: 20°)

```bash
ros2 topic pub --once rbq/motion/cmd_highLevel
rbq_msgs/msg/HighLevelCommand \
'{header: {stamp: {sec: 0, nanosec: 0}, frame_id: "base"}, identifier:
"standing_pose", roll: 15.0, pitch: 30.0, yaw: 20.0, vel_x: 0.0, vel_y:
0.0, omega_z: 0.0, delta_body_h: 0.0, delta_foot_h: 0.0, gait_state: 1,
gait_transition: true}'
```

📷 4. Vision Topics

- PTZ Camera Control Topics

Topic	Description
rbq/ptzCamera/setPanTiltZoom	PTZ camera control command using Float32MultiArray [pan, tilt, zoom]

> example:

```bash
ros2 topic pub --once rbq/ptzCamera/setPanTiltZoom
std_msgs/msg/Float32MultiArray "{data: [0.0, 0.0, 1.0]}"
```

- Camera Sensor Topics

> Camera sensors provide RGB, IR, and depth images from multiple positions around the robot. Images are published in both raw and compressed formats with corresponding camera calibration information.

> 📍 **Hardware Information:** For camera specifications, FOV, and transformation matrices, see **Front & Rear Cameras**, **Ground-view Camera**, **LiDAR (OS1-32)**, **LiDAR (MID-360)**, and **PTZ Camera** under Manual → Hardware.

Camera	Image Type	Format	Topic Name
Bottom (0-3)	IR	Raw	rbq/vision/sensor_bottom_0/ir rbq/vision/sensor_bottom_1/ir rbq/vision/sensor_bottom_2/ir rbq/vision/sensor_bottom_3/ir
		Compressed	rbq/vision/sensor_bottom_0/ir/compressed rbq/vision/sensor_bottom_1/ir/compressed rbq/vision/sensor_bottom_2/ir/compressed rbq/vision/sensor_bottom_3/ir/compressed
		Camera Info	rbq/vision/sensor_bottom_0/ir/camera_info rbq/vision/sensor_bottom_1/ir/camera_info rbq/vision/sensor_bottom_2/ir/camera_info rbq/vision/sensor_bottom_3/ir/camera_info
	Depth	Raw	rbq/vision/sensor_bottom_0/depth rbq/vision/sensor_bottom_1/depth rbq/vision/sensor_bottom_2/depth rbq/vision/sensor_bottom_3/depth
		Compressed	rbq/vision/sensor_bottom_0/depth/compressed rbq/vision/sensor_bottom_1/depth/compressed rbq/vision/sensor_bottom_2/depth/compressed rbq/vision/sensor_bottom_3/depth/compressed
		Camera Info	rbq/vision/sensor_bottom_0/depth/camera_info rbq/vision/sensor_bottom_1/depth/camera_info rbq/vision/sensor_bottom_2/depth/camera_info rbq/vision/sensor_bottom_3/depth/camera_info
Front/Rear	RGB	Raw	rbq/vision/sensor_front/rgb rbq/vision/sensor_rear/rgb
		Compressed	rbq/vision/sensor_front/rgb/compressed rbq/vision/sensor_rear/rgb/compressed
		Camera Info	rbq/vision/sensor_front/rgb/camera_info rbq/vision/sensor_rear/rgb/camera_info
	IR	Raw	rbq/vision/sensor_front/ir rbq/vision/sensor_rear/ir

		Compressed	rbq/vision/sensor_front/ir/compressed rbq/vision/sensor_rear/ir/compressed
		Camera Info	rbq/vision/sensor_front/ir/camera_info rbq/vision/sensor_rear/ir/camera_info
	Depth	Raw	rbq/vision/sensor_front/depth rbq/vision/sensor_rear/depth
		Compressed	rbq/vision/sensor_front/depth/compressed rbq/vision/sensor_rear/depth/compressed
		Camera Info	rbq/vision/sensor_front/depth/camera_info rbq/vision/sensor_rear/depth/camera_info

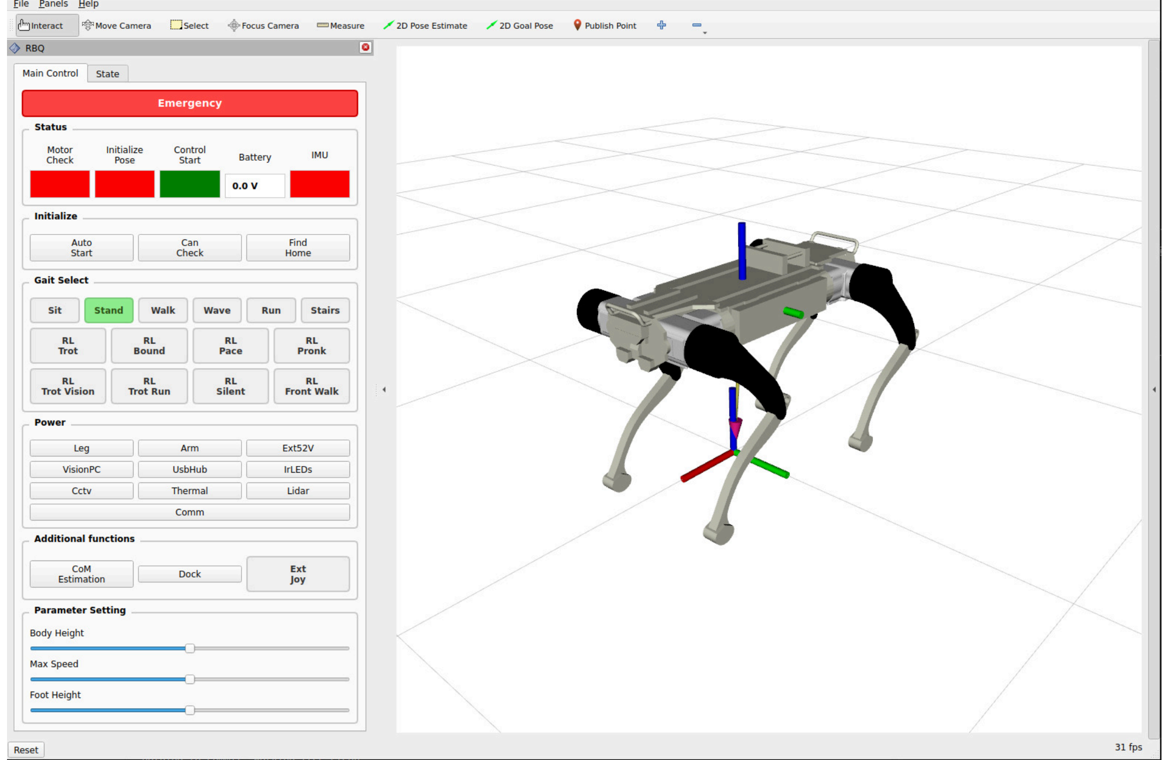
---

🖼️ RViz Visualization

Launch RViz2 with GUI

> You can simultaneously visualize and send commands to the robot through the GUI on the left side of RViz.
> Go to your workspace directory first:
>
> ```bash
> cd <your workspace>/RBQ
> ```
>
> Then launch RViz:

```bash
source ros2/install/local_setup.bash
./scripts/start_rviz.bash
```



📘 Help

If you encounter problems when using this repository, feel free to open an issue.

Register Git Issue

👥 Contributors

This project is provided by the **Rainbow-Robotics**.

Rainbow-Robotics contributors:

- Gurbann
- Jinwon Seo

1.5 RBQ GYM

📚 Introduction

Welcome to the Open-source RBQ gym simulation environment

This manual provides a comprehensive guide on how to **simulate** the RBQ robot in IsaacGym to **train** your own locomotion policy using Reinforcement-Learning, then **play** to evaluate, and then **deploy** it to the real robot.

💻 Requirements

To run the RBQ gym simulation environment smoothly, the following requirements are recommended:

- OS: Ubuntu 22.04 (x86)
- Minimum PC specs:
  - CPU: Intel Core i7 - 12th Gen
  - RAM: 16 GB
  - Storage: 25 GB
  - GPU: Nvidia RTX 4080
- Control Devices
  - Keyboard

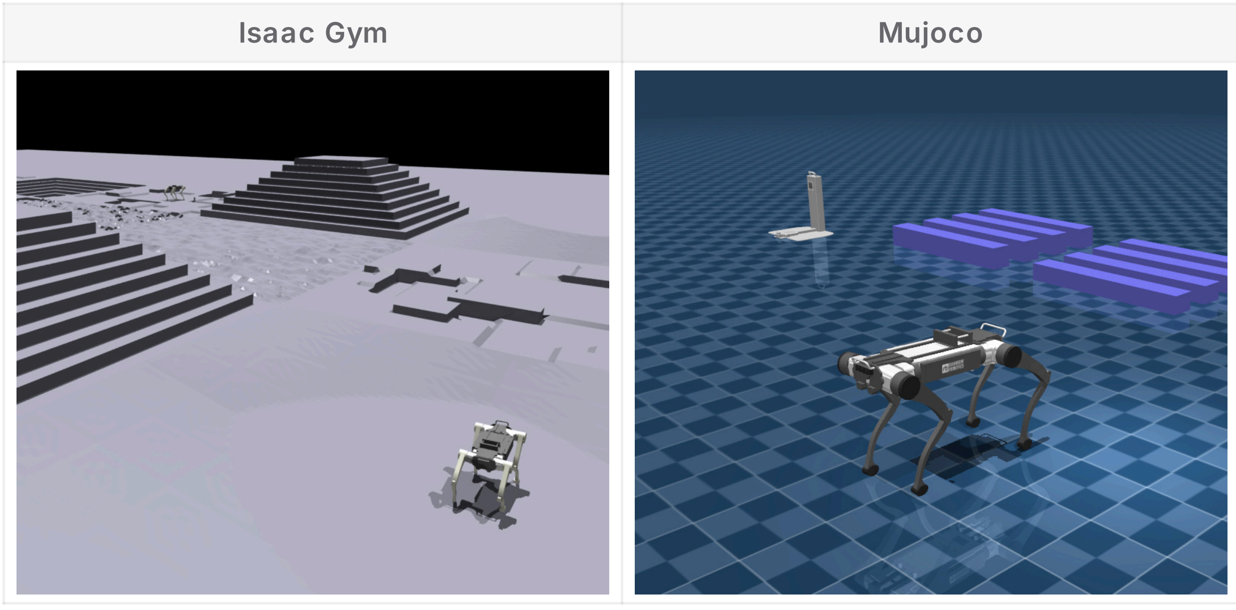
> **GPU Compatibility**
>
> Isaac Gym does not support **NVIDIA RTX 50 series (Blackwell)** GPUs. Please use RTX 40 series or earlier GPUs for training.

🔁 Process Overview

The basic workflow for using reinforcement learning to achieve motion control is:

Train → Play → Sim2Sim

- **Train**: Use the IsaacGym simulation environment to let the robot interact with the environment and find a policy that maximizes the designed rewards through Reinforcement-Learning with PPO algorithm.
- **Play**: Use the Play command to verify the trained policy and ensure it meets expectations within the IsaacGym environment.
- **Sim2Sim**: Evaluate the trained policy in **Mujoco** simulator to ensure the performance and reliability.
- **Deploy** : Deploy the evaluated policy to the real robot.



Setup an environment

Go to the **rbq_gym directory**:

```bash
cd <your workspace>/RBQ/rbq_gym
```

Within the **rbq_gym directory** run the following command to setup the environment:

```bash
bash scripts/setup.bash
```

Train

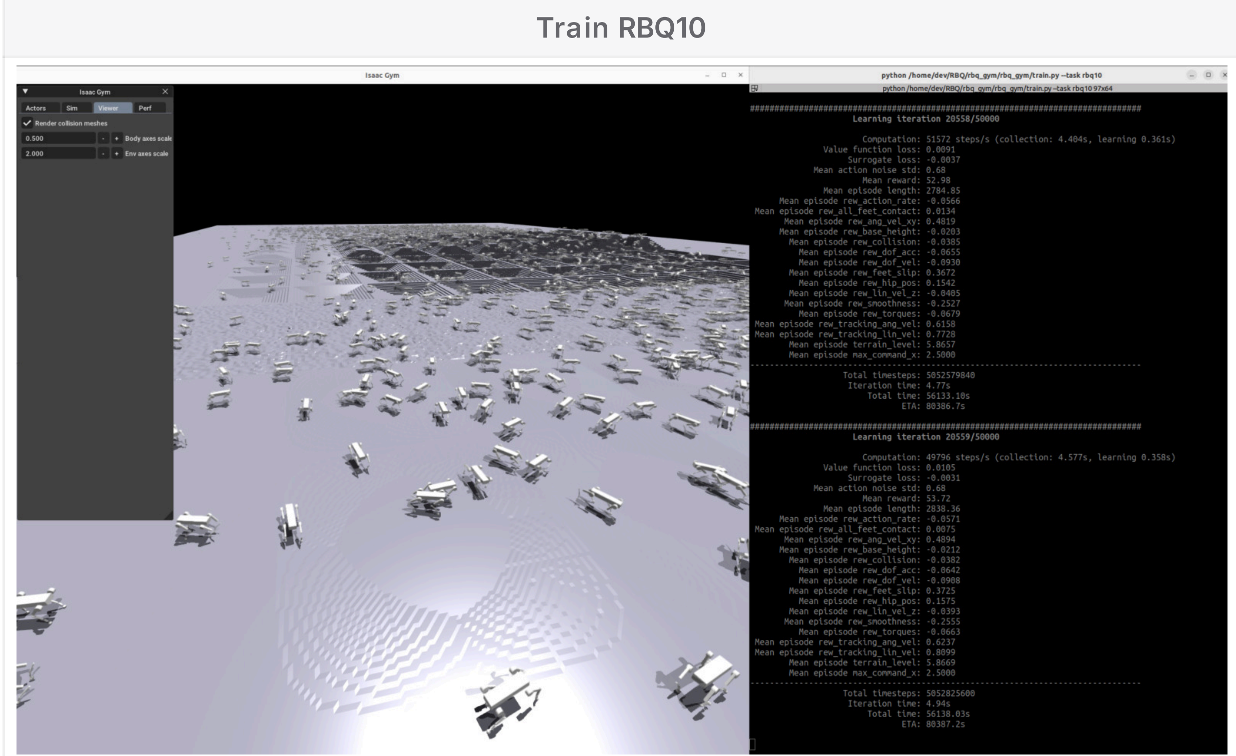
Within the **rbq_gym directory** run the following command to start training:

```bash
bash scripts/train.bash
```

- To run headless (no rendering), add `--headless`

> **TIP**
>
> To improve performance, once the training starts press `v` to stop the rendering. You can enable it later to check the progress.

Train RBQ10



Play

Within the **rbq_gym directory** run the following command to evaluate the training result:

```bash
bash scripts/play.bash
```

- To run on CPU, add the following arguments: `--sim_device=cpu`
- By default, the loaded policy is the last model of the last run of the experiment folder.
- Other runs/model iteration can be selected by setting `load_run` and `checkpoint`.

Play RBQ10
▶ Open on YouTube (click)

🎯 Sim2Sim & Deploy

To evaluate and deploy the trained policy, refer to the **Simple-RL** for detailed instructions.

📂 Directory Structure

```
rbq_gym/
├── 3rdparty
├── policy
│   └── rbq10
│       ├── info.json
│       └── policy.onnx
├── rbq_gym
│   ├── envs
│   │   ├── base
│   │   │   ├── base_config.py
│   │   │   ├── base_task.py
│   │   │   ├── rbquad_config.py
│   │   │   └── rbquad_env.py
│   │   ├── __init__.py
│   │   └── rbq10
│   │       ├── rbq10_config.py
│   │       ├── rbq10_env.py
│   │       └── rewards.py
│   │
```

```
│   ├── __init__.py
│   ├── model_test.py
│   ├── play.py
│   ├── train.py
│   └── utils
│       ├── helpers.py
│       ├── __init__.py
│       ├── keyboard.py
│       ├── logger.py
│       ├── math.py
│       ├── task_registry.py
│       └── terrain.py
├── scripts
│   ├── activate.bash
│   ├── clear.bash
│   ├── configure.bash
│   ├── play.bash
│   ├── python.bash
│   ├── setup.bash
│   └── train.bash
├── dependencies.yaml
└── setup.py
```

- `setup.bash`, `train.bash`, `play.bash` : scripts for setup, train and play.
- `envs/base/` : base environment classes and configurations for the rbq robot.
- `envs/rbq10/` : RBQ10 environment, configuration, and reward functions.
- `utils/` : utility modules including helpers, keyboard input, logging, math, task registry, and terrain generation.
- `3rdparty/` : third-party dependencies and libraries.
- `policy/` : trained policies in ONNX format along with their info files.

🔧 Adding a New Environment

The base environment `rbquad_env` implements a rough terrain locomotion task. To add a new environment:

1. Add a new folder to `envs/` with `<your_env>_config.py`, inheriting from existing environment configs.
2. If adding a new robot:

  - Add the corresponding assets to the `resources/` directory.
  - In the config, set the asset path, define body names, `default_joint_positions`, and PD gains.
  - Specify the desired `train_cfg` and environment class name.
  - In `train_cfg`, set `experiment_name` and `run_name`.
3. (If needed) Implement your environment in `<your_env>_env.py`, inheriting from an existing environment, and overwrite desired functions or add reward functions.
4. Register your environment in `rbq_gym/envs/__init__.py`.
5. Modify/tune other parameters as needed. To remove a reward, set its scale to zero. Do not modify parameters of other environments.

1.6 RBQ-Lab

📚 Introduction

RBQ-Lab user guide

This document explains the basic procedure to train **RBQ10** quadruped locomotion policy in **Isaac Sim / Isaac Lab**, then **replay (Play)** and use the outputs for deployment.

💻 System Requirements

For RBQ-Lab, the following environment is recommended:

- OS: Ubuntu 22.04 (x86)
- GPU: CUDA-capable NVIDIA GPU
- Disk: minimum **80 GB**, recommended **120 GB or more** (includes Isaac Sim / Isaac Lab / Conda / cache)
- Network: initial setup needs large downloads (`wget`, `git clone`, `pip`)

> **Version pins**
>
> For the actual operating versions, check the pins in **`dependencies.yaml`** and **`scripts/setup.bash`** first.

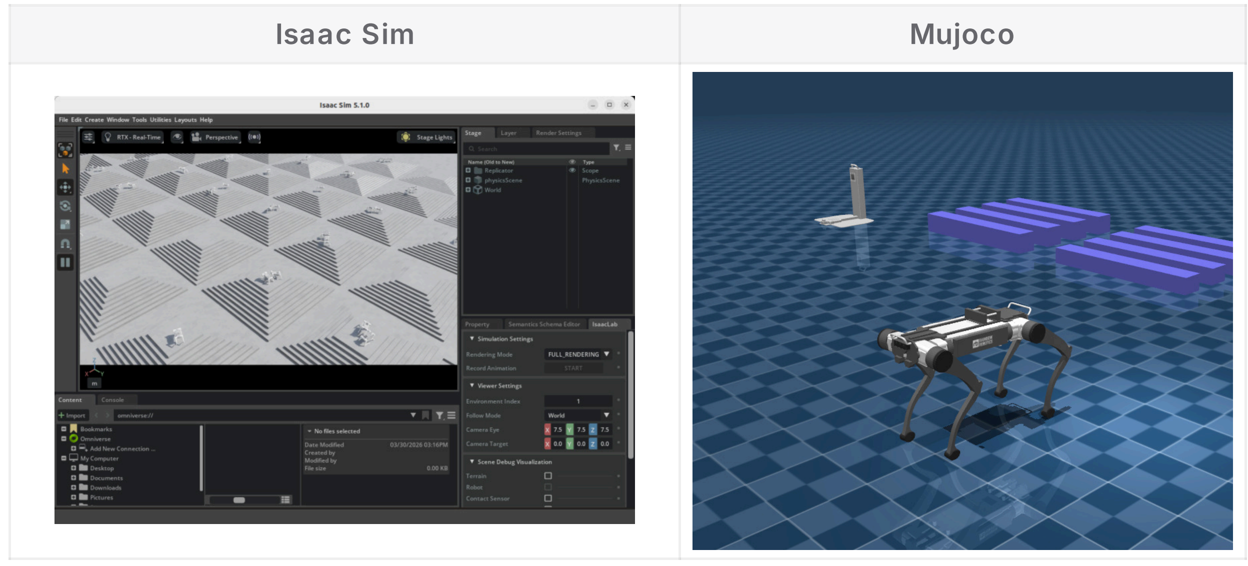
🔁 Process overview

The default workflow is:

Setup → Train → Play → Deploy

- **Setup**: install dependencies/environment with `setup.bash`

- **Train**: train the `rbq10` task
- **Play**: replay/verify the `rbq10_play` task
- **Deploy**: use the exported policy artifacts (`policy.onnx` , `info.json` , etc.)



📦 Setup

Go to the `rbq_lab` **directory**:

```bash
cd <your workspace>/RBQ/rbq_lab
```

Inside the `rbq_lab` **directory**, run:

```bash
bash scripts/setup.bash
```

Train

Default training:

```bash
bash scripts/train.bash
```

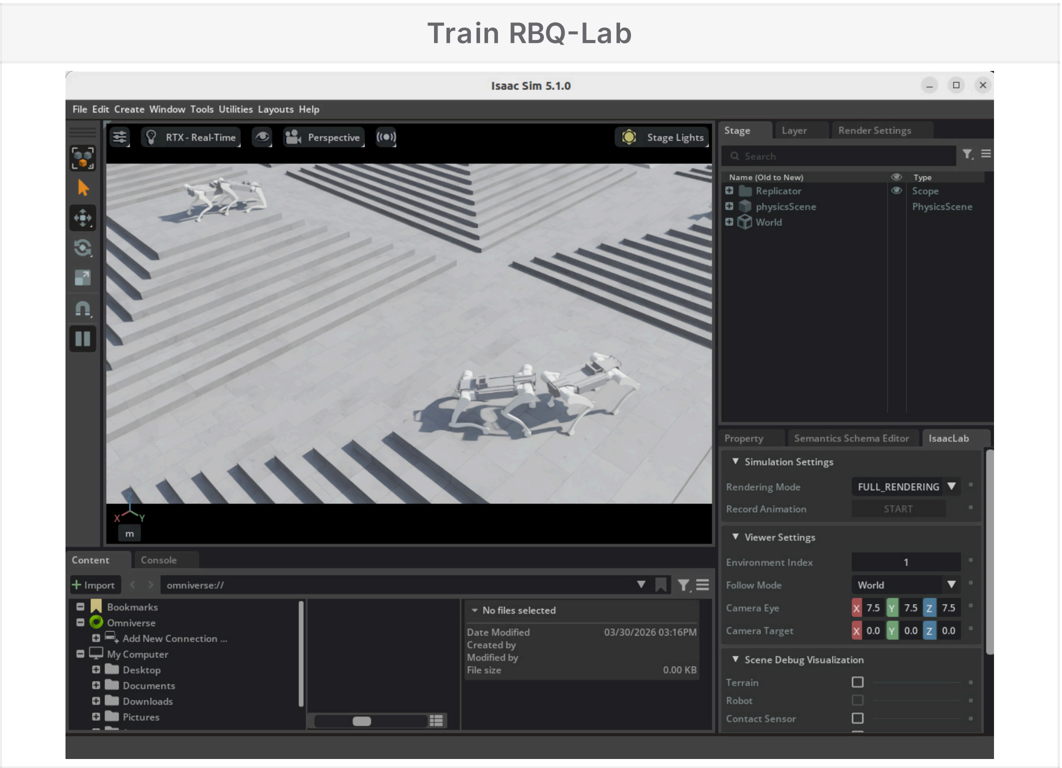
Default task is `rbq10`. Training logs are usually stored under `logs/rsl_rl/<experiment_name>/...`.

Options available in `train.py` (reference)

`--task`, `--num_envs`, `--device`, `--headless`, `--resume`, `--checkpoint`, `--max_iterations`, `--seed`, `--video`, etc. Full list: `bash scripts/python.bash rbq_lab/train.py --help`.

If you want to hardcode options, edit the `train.py` **invocation inside** **`scripts/train.bash`**. Since execution is mostly headless, the training window may not appear (which can be normal).

Train RBQ-Lab



Training logs: `logs/rsl_rl/<experiment_name>/...`

Play

Default replay:

```bash
bash scripts/play.bash
```

Default task is `rbq10_play`. Training checkpoints (.pt) are typically under `logs/rsl_rl/<experiment_name>/.../model_*.pt`. When running Play, specifying the exact `model_*.pt` path via `--checkpoint` is the most reliable.

Options available in `play.py` (reference)

`--use_pretrained_checkpoint`, `--no_keyboard`, `--real-time`, `--video`, `--checkpoint`, etc. Full list: `bash scripts/python.bash rbq_lab/play.py --help`.

If you want to hardcode options, edit the **`play.py` invocation inside `scripts/play.bash`**.

Play RBQ-Lab
▶ Open on YouTube (click)

🎯 Deploy

To evaluate the trained policy and deploy it to the real robot, refer to **Simple-RL**.

After Play, `policy.jit`, `policy.onnx`, and `info.json` may be generated under `logs/<experiment_name>/exported/`.

📂 Directory structure

```
rbq_lab/
├── 3rdparty
├── policy
│   └── rbq10
│       ├── info.json
│       └── policy.onnx
├── rbq_lab
│   ├── envs
```

```
│   │   ├── base
│   │   │   ├── base_task.py
│   │   │   └── base_task_cfg.py
│   │   ├── __init__.py
│   │   └── rbq10
│   │       ├── __init__.py
│   │       ├── env.py
│   │       ├── env_cfg.py
│   │       ├── env_mdp.py
│   │       ├── rbq10.py
│   │       └── rsl_rl_ppo_cfg.py
│   ├── __init__.py
│   ├── play.py
│   ├── train.py
│   └── utils
│       ├── camera.py
│       ├── cli_args.py
│       ├── keyboard.py
│       ├── marker.py
│       ├── math.py
│       └── rough.py
├── scripts
│   ├── activate.bash
│   ├── clear.bash
│   ├── configure.bash
│   ├── isaacsim.bash
│   ├── play.bash
│   ├── python.bash
│   ├── setup.bash
│   └── train.bash
├── dependencies.yaml
└── setup.py
```

- USD and other assets can be placed under a separate `resources/` directory; code uses `LAB_ASSET_DIR`.
- Training logs: `logs/rsl_rl/<experiment_name>/...`
- Replay/checkpoint lookup: `logs/<experiment_name>/...`
- Cleanup: `bash scripts/clear.bash` (`*.egg-info`, `__pycache__`, and optionally `logs/`)

🔧 Add a new Lab environment

The typical approach is to copy and modify the existing `rbq10` environment to register a new task.

1. Add a new environment folder under `rbq_lab/envs/`.
2. Create `env.py`, `env_cfg.py`, `env_mdp.py`, and `rsl_rl_ppo_cfg.py` for the new task.
3. If needed, add robot/environment assets (USD, etc.) under `resources/` and connect the paths in the environment config.
4. Register the new environment/task id in `rbq_lab/envs/__init__.py`.
5. Verify that `train.py` selects the new task correctly, then validate with training and replay.

After adding a task, it's recommended to run with a small `num_envs` first to catch early issues (asset paths, observation/action dimensions, reward definition).

2.1 🚀 Simple-motion

🛠️ Build the examples

- Run the following command to build the project:

```bash
bash scripts/ex/docker/run.bash --no-cache
```

- *arguments:*
    - `--help` : Display help message and exit.
    - `--no-cache` : rebuild.

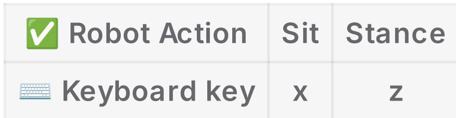
🧪 Test

- Run `simulator` script

```bash
bash scripts/sim.bash
```

- Run `Simple-motion` process

```bash
sudo ./examples/bin/Simple-motion
```



✅ Robot Action	Sit	Stance
⌨️ Keyboard key	x	z

[▶ Open on YouTube (click)]

🚀 Deploy and Run in the real Robot

- Connect your PC to the robot's Wi-Fi.

  📝 *Wi-Fi SSID: `RBQ_xxxx` (replace `xxxx` with the specific identifier of your robot's Wi-Fi).*

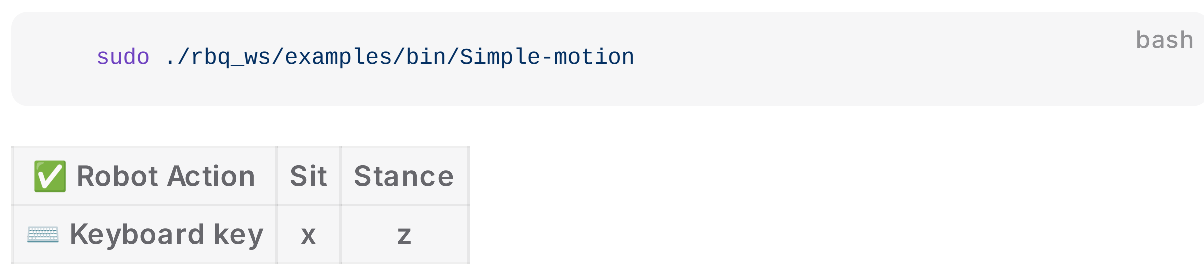
- Copy the binary to the robot pc

```bash
scp examples/bin/Simple-motion rbq@192.168.0.10:~/rbq_ws/examples/bin/.
```

- Secure shell to the robot pc

```bash
ssh rbq@192.168.0.10
```

- Run the process on robot pc



```bash
sudo ./rbq_ws/examples/bin/Simple-motion
```

✅ Robot Action	Sit	Stance
⌨️ Keyboard key	x	z

Software Version Compatibility

Make sure that the **robot's onboard software version** matches the **software version installed on your development PC**.

- We recommend updating both environments to the latest official release before deployment. You can download the latest release from the **official RBQ software repository**.

2.2 🚀 Simple-command

🛠️ Build the examples

- Run the following command to build the project:

```bash
bash scripts/ex/docker/run.bash --no-cache
```

- *arguments:*
  - `--help` : Display help message and exit.
  - `--no-cache` : rebuild.

🧪 Test

- Run `simulator` script

```bash
bash scripts/sim.bash
```

- Run `Simple-command` process

```bash
./examples/bin/Simple-command
```

✅ Button Action	Sit	Stance	Walk	Stairs	Running
⌨️ Keyboard key	1	2	3	4	5

▶ Open on YouTube (click)

🚀 Deploy and Run in the real Robot

- Connect your PC to the robot's Wi-Fi.

  📝 *Wi-Fi SSID:* `RBQ_xxxx` *(replace* `xxxx` *with the specific identifier of your robot's Wi-Fi).*

- Copy the binary to the robot pc

```bash
scp examples/bin/Simple-command rbq@192.168.0.10:~/rbq_ws/examples/bin/.
```

- Secure shell to the robot pc

```bash
ssh rbq@192.168.0.10
```

- Run the process on robot pc

```bash
./rbq_ws/examples/bin/Simple-command
```

✅ Button Action	Sit	Stance	Walk	Stairs	Running
⌨️ Keyboard key	1	2	3	4	5

⚠️ Software Version Compatibility

Make sure that the **robot's onboard software version** matches the **software version installed on your development PC**.

- We recommend updating both environments to the latest official release before deployment. You can download the latest release from the **official RBQ software repository**.

2.3 🚀 Simple-RL

🛠️ Build the examples

- Run the following command to build the project:

```bash
bash scripts/ex/docker/run.bash --no-cache
```

  - *arguments:*
    - `--help` : Display help message and exit.
    - `--no-cache` : rebuild.

🧪 Test

- Run `simulator` script

```bash
bash scripts/sim.bash
```

⚠️ Make Sure Auto Start the robot before RUN Simple-RL

- Run `Simple-RL` process

```bash
sudo ./examples/bin/Simple-RL
```

✅ Robot Action	Sit	Stance	Control Mode	Command
Keyboard Key	Z	X	C	W - 🔼 Forward S - 🔽 Backward A - ◀️ Left Turn D - ▶️ Right Turn Q - 🔄 Yaw Left E - 🔃 Yaw Right

▶ Open on YouTube (click)

🚀 Deploy and Run in the real Robot

- Connect your PC to the robot's Wi-Fi.

  📝 *Wi-Fi SSID: `RBQ_xxxx` (replace `xxxx` with the specific identifier of your robot's Wi-Fi).*

- Connect to Robot PC

```bash
ssh rbq@192.168.0.10
```

- Make the file directory

```bash
mkdir -p /home/rbq/rbq_ws/rbq_gym/policy/rbq10
mkdir -p /home/rbq/rbq_ws/examples/bin
```

- Return to your PC and Copy the binary,policy file to the robot pc

```bash
scp -r examples/bin/ rbq@192.168.0.10:~/rbq_ws/examples/
scp -r rbq_gym/policy/rbq10/ rbq@192.168.0.10:~/rbq_ws/rbq_gym/policy/
```

- Secure shell to the robot pc

```bash
ssh rbq@192.168.0.10
```

⚠️ Make Sure Auto Start the robot before RUN Simple-RL

- Run the process on robot pc

```bash
sudo ~/rbq_ws/examples/bin/Simple-RL
```

✅ Robot Action	Sit	Stance	Control Mode	Command

Keyboard Key	Z	X	C	W - 🔼 Forward S - 🔽 Backward A - ◀️ Move Left D - ▶️ Move Right Q - 🔄 Yaw Left E - 🔃 Yaw Right

---

Software Version Compatibility

Make sure that the **robot's onboard software version** matches the **software version installed on your development PC**.

- ✅ We recommend updating both environments to the latest official release before deployment. You can download the latest release from the **official RBQ software repository**.

3.1 How to update software

Step 1. Download the Latest Release

Download the latest source code from the official RBQ GitHub Releases page:

> 📝 Make sure to choose the correct version for your robot platform.

📦 Step 2. Deploy Applications (Binaries) to the Robot

After downloading the `source code` file:

1. **Connect your development PC to the robot via network (SSH enabled)**
2. Run the deployment script from your development PC:

```bash
bash scripts/deploy.bash
```

You will be prompted for an SSH password.

- **Password:** Please ask the person in charge for the SSH password.
- ⚠️ Important: Once `deploy.bash` has completed, **restart** the robot by turning it off and on to run with the newly deployed binaries.

📋 Additional Update Information

Prerequisites

- Ensure you have SSH access to the robot
- Make sure your development environment is properly set up
- Verify network connectivity between your PC and the robot

Update Process Steps

1. **Prepare your binaries**: Make sure all your compiled applications are ready for deployment
2. **Run deployment script**: Execute the `deploy.bash` script from your development PC
3. **Provide credentials**: Enter the SSH password when prompted
4. **Restart the robot**: Power cycle the robot to load the new binaries

Troubleshooting

- If deployment fails, check network connectivity
- Ensure you have the correct SSH credentials
- Verify that the robot is accessible and powered on

📜 history

- 2025.07.18 : "RCL : Robot Control Library" frisrt release
- 2025.07.29 : "RCL" State Estimator added

3.2 How to Connect to the RBQ Development PC

This guide explains how to connect to the **RBQ Dev PC** using **NoMachine** for remote desktop access.

🚀 Step 1. Download NoMachine

1. Download the NoMachine application from the official website: 🔗 **NoMachine Download Page**
2. Be sure to select the correct version for your platform (Linux).

📦 Step 2. Install NoMachine on the RBQ Dev PC

Open a terminal and run the following commands to install NoMachine:

```bash
cd /usr
sudo tar zxvf ~/Downloads/nomachine_latest_version.tar.gz
sudo /usr/NX/nxserver --install
sudo /usr/NX/bin/nxserver --status
```

> ✅ Use `nxserver --status` to verify that the server is running properly.

🔌 Step 3. Connect from Your Development PC

1. Make sure your **development PC is connected to the same network** as the RBQ (via Ethernet or SSH-enabled connection).

2. Launch the **NoMachine app** on your PC.

3. Click **"Add"** to create a new connection.

4. Fill in the following details:

```
Name : rbq
Host : 192.168.0.10
```

> 💡 Recommended Nomachine Display Resolution Settings:
>
> - ✅ Enable: **"Scale the remote desktop to fit the window"**
> - 🚫 Disable: **"Resize the remote display"**

✅ You're Now Connected!

You can now control and develop on the **RBQ Dev PC** remotely via a user-friendly desktop interface.

Enjoy your development workflow!

4.1 Error Reporting

We are continuously working to gather and address various runtime errors and issues experienced by users. If you encounter any problems, we highly encourage you to report them through the following channels:

- **GitHub Issues**: If you're facing any technical issues or bugs, please submit them to our GitHub Issues page. This helps us track and resolve problems efficiently.
  Submit an Issue

- **GitHub Discussions**: For general questions, feedback, or to discuss common runtime errors with other users and developers, you can join our GitHub Discussions forum.
  Join the Discussion

- **Email for Sensitive Inquiries**: If you have more sensitive or private questions, feel free to reach out directly via email at
  rbq.support@rainbow-robotics.com