# Desktop Robot Arm Control Guide

> *This document is intended for independent use by desktop robot arm customers. It follows the Raybot robot motion control guide, with the model, planning group, joint names, and startup procedures adapted for the* `rb_a630_2` *six-axis desktop arm.*

## Table of Contents


- 1. Document Overview
  - 1.1 Scope
  - 1.2 Intended Readers
- 2. Glossary
- 3. Robot Hardware Introduction
  - 3.1 Introduction to Robot Hardware
  - 3.2 Knowledge Related to Coordinate Systems
- 4. System Startup and Environment Configuration
  - 4.1 Special Notice Regarding ROS2 Communication
  - 4.2 Detailed Explanation of Startup Commands
  - 4.3 Launch Parameter Configuration
- 5. Overall Architecture
  - 5.1 Core Components and Relationships
  - 5.2 Module Function Comparison Table
  - 5.3 System Architecture Diagram
  - 5.4 Complete System Startup
  - 5.5 Global Control Mode Switching
- 6. Planning Group and Predefined Poses
  - 6.1 Planning Group Introduction
    - 6.1.1 Planning Group Joint Composition
    - 6.1.2 Predefined Poses
- 7. Planner Configuration
  - 7.1 Supported Planner Types
  - 7.2 pipeline_id / planner_id Configuration

    - 7.3 Analysis of the Reasons for Planning Failure
- 8. Arm Motion Controller (Core execution node)
    - 8.1 Module Functions
- 9. CLI Controller (Command-line robotic arm motion control)
    - 9.1 Module Functions
    - 9.2 Command Format and Syntax
        - 9.2.1 Circular Motion (Pilz CIRC)
    - 9.3 External Interface Description
    - 9.4 Complete Test Command
    - 9.5 Implementation Details
- 10. Action Recorder (Motion programmer)
    - 10.1 Module Functions
    - 10.2 Configuration Parameters
    - 10.3 External Interface Description
    - 10.4 Complete Test Command
    - 10.5 Implementation Details
    - 10.6 Interaction Process
- 12. Troubleshooting
    - 12.1 Clearing Motor Errors
    - 12.2 Common Errors and Solutions
- 13. Example Command Set
    - 13.1 ROS2 Environment Configuration Commands
    - 13.2 System Startup Command
    - 13.3 CLI Controller Commands
    - 13.4 Action Recorder Commands
    - 13.5 Motion Stop Commands


## Document Overview

This document introduces ROS2 environment configuration, system startup, MoveIt motion planning, CLI control, motion programming, stopping procedures, and troubleshooting for RB-A630-2 desktop robot arm customers.

### Scope

- Raybot robot motion control
- Motion program recording and execution
- Automatic object grasping and placement
- ROS2 robot application development

### Intended Readers

- Robotics System Integration Engineer
- ROS2 Application Developer
- Robot Operator and Maintenance Technician
- Researchers in related fields

## Glossary

> ***Configuration boundary**: The standard `rb_a630_2` configuration contains only the `arm` planning group and six arm joints. Gripper and pan-tilt control are disabled by default.*

| Term | Full Name | Explanation |
| --- | --- | --- |
| ROS2 | Robot Operating System 2 | The second-generation robot operating system provides the libraries and tools necessary for robot development |
| PTP Motion | Point-to-Point Motion | Point-to-point motion: The robotic arm's end effector reaches the target position via the fastest path |
| LIN Motion | Linear Motion | It moves along a straight line trajectory, suitable for scenarios requiring precise path control |
| MoveGroupInterface | MoveGroupInterface | High-level interfaces in ROS2 for controlling robotic arm movement. |
| TF | Transform | Coordinate transformations are used to represent the positional relationships between the various components of a robot |
| OMPL | Open Motion Planning Library | An open-source motion planning library, providing various sampling-based planning algorithms |
| PILZ | PILZ Industrial Motion | Industrial-grade motion planner, providing industrial motion types such as PTP, LIN, and CIRC |
| Planner | Planner | An algorithmic module used for calculating the motion path of a robotic arm |
| Planning Pipeline | Planning Pipeline | The motion planning workflow includes steps such as planner selection and collision detection |
| Joint Space | Joint Space | The motion space defined by joint angles |
| Cartesian Space | Cartesian Space | The motion space defined by the position and orientation of the end effector |
| Obstacle Avoidance Planning | Obstacle Avoidance Planning | Motion planning that automatically avoids obstacles |
| Topic | Topic | The asynchronous communication mechanism in |

| Term | Full Name | Explanation |
| --- | --- | --- |
| | | ROS2 is used for publishing and subscribing to messages |
| Service | Service | The synchronous communication mechanism in ROS2 uses a request-response pattern |
| Action | Action | ROS2's asynchronous communication mechanism supports long-running tasks and cancellation functionality |

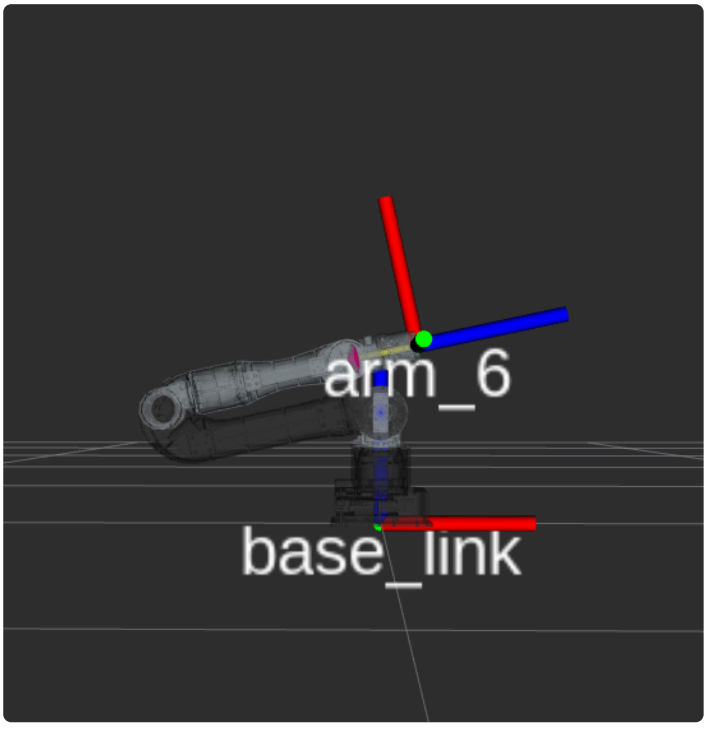
## Robot Hardware Introduction

### Introduction to Robot Hardware

This section uses the RB-A630-2 desktop robot arm as an example to explain the hardware structure.

The standard desktop robot arm consists of one 6-DOF arm. Refer to the actual delivery list to determine whether a gripper, camera, or other end effector is installed.

The diagram below shows the robot base coordinate system and the tool coordinate system:




In the diagram, red, green, and blue represent the x, y, and z axes, respectively. In the base coordinate system, the positive x-axis points forward

from the robotic arm, the positive y-axis points to the left of the robotic arm, and the positive z-axis points upwards. By specifying the pose of the robot's end-effector coordinate system relative to the base coordinate system, the position and orientation of the robotic arm's end-effector in space can be controlled.

## Knowledge Related to Coordinate Systems

### The meaning of the robot tool coordinate system

The robot tool coordinate system (TCP) is a coordinate system defined on the robot's end effector, used to precisely describe the position and orientation of the end effector. Its main significance includes:

- **Simplify programming**: Using the tool coordinate system, users can directly specify the target position of the tool tip without having to consider the size and shape of the tool itself.
- **Improve accuracy**: Accurate calibration of the tool coordinate system can improve the positioning accuracy of the robot.
- **Easy to change tools**: When changing to a different tool, only the tool coordinate system needs to be recalibrated; the robot program does not need to be modified.
- **Performing complex actions**: The tool coordinate system is fundamental to achieving Cartesian motion and complex path planning.

### Background information on RPY rotation

RPY(Roll-Pitch-Yaw) rotation is a way to describe an object's orientation, representing its direction through rotations around three coordinate axes:

- **Roll**: Rotate around the X-axis
- **Pitch**: Rotate around Y-axis
- **Yaw**: Rotate around Z-axis

RPY rotation uses a fixed-axis rotation sequence, specifically Z-Y-X order. This means rotating around the Z-axis first (Yaw angle), then around the Y-axis (Pitch angle), and finally around the X-axis (Roll angle).

### Implementation of robot end-effector pose transformation

Robot end-effector pose transformation refers to the process of transforming the tool coordinate system from one position and orientation to another, which is mainly achieved through the following steps:

1. **Forward Kinematics (FK)**
   Based on the current joint angles, the pose of the end-effector in the specified reference coordinate system is calculated, which is used to obtain the initial state or for state feedback.

2. **Target Pose Construction and Coordinate Transformation (Pose Transform)**
   The target pose is transformed between different coordinate systems (for example, converting the target point from the camera coordinate system to the base coordinate system) to obtain the target end-effector pose usable by the control system.

3. **Inverse Kinematics (IK)**
   The target end-effector pose is converted into a set (or sets) of feasible joint angle solutions, and suitable solutions are selected based on constraints such as joint limits, continuity, and collision avoidance.

4. **Trajectory planning (Trajectory Planning)**
   Generate a smooth trajectory from the current state to the target state in joint space or Cartesian space, satisfying constraints on velocity, acceleration, and jerk.

5. **Joint control and servo actuation (Control / Servo)**
   The controller outputs position, velocity, and torque commands according to the planned trajectory, driving each joint to move and perform closed-loop tracking, ultimately enabling the end effector to reach the target pose.

## System Startup and Environment Configuration

### Special Notice Regarding ROS2 Communication

The system defaults to using a global ROS2 communication mechanism. In a local area network, all ROS2 processes on different hosts will automatically communicate with each other (i.e., a ROS2 process on one host can by default

discover the topics of ROS2 processes on other hosts), which may lead to node conflicts or unexpected execution of control programs. To avoid such problems, it is recommended that users configure the following restrictions before starting ROS2 programs.

1. Set the ROS2 communication range.

```
# Set the ROS2 communication scope to the local host (isolating communication to the local network).
echo "export ROS_LOCALHOST_ONLY=1" >> ~/.bashrc

# Set the ROS2 communication range to the local area network (allowing communication with other devices on the network).
echo "export ROS_LOCALHOST_ONLY=0" >> ~/.bashrc
```

2. Configure ROS_DOMAIN_ID to isolate communication domains.

ROS_DOMAIN_ID is used to divide independent communication domains. When the ROS_DOMAIN_ID of all ROS2 processes is the same, they can communicate with each other; conversely, even on the same host machine, different ROS_DOMAIN_ID 's ROS2 processes cannot communicate. This configuration is similar to port number settings in a network, achieving communication isolation by specifying a non-zero number.

```
# Check the current ID
printenv ROS_DOMAIN_ID

# Set ROS_DOMAIN_ID to 72
export ROS_DOMAIN_ID=72
```

## Detailed Explanation of Startup Commands

### Setting ROS2 environment variables:

```
source /opt/ros/humble/setup.bash
source ~/raybot_core_ws/install/setup.bash
```

### Complete startup command:

```
ros2 launch raybot_bringup robot.launch.py \
  model:=rb_a630_2 \
  use_rviz:=true
```

The unified startup entry point is provided by `raybot_bringup`. Start the RB-A630-2 with the following command:

```
ros2 launch raybot_bringup robot.launch.py \
  model:=rb_a630_2 \
  use_rviz:=true
```

The startup time is approximately 2 minutes. Operate the device only after startup is complete.

## Launch Parameter Configuration

### JTC Trajectory Error Tolerances

`raybot_bringup robot.launch.py` supports overriding JointTrajectoryController (JTC) trajectory error tolerances at startup. These parameters control trajectory execution error checking: if the error exceeds the configured tolerance, JTC may abort the trajectory and report an error. If these parameters are not provided, the system uses the default values in `ros2_controllers.yaml`.

| Parameter | Example Value | Description |
| --- | --- | --- |
| jtc_trajectory_tolerance | 0.8 | Joint path tracking error tolerance during trajectory execution. Unit is rad for revolute joints. |
| jtc_goal_tolerance | 0.25 | Final joint position error tolerance at the trajectory goal. Unit is rad for revolute joints. |
| jtc_goal_time | 3.0 | Extra time allowed after the trajectory end to satisfy the final goal tolerance, in seconds. |
| jtc_stopped_velocity_tolerance | 0.2 | Stopped velocity tolerance at the goal. The unit follows the joint velocity interface; for revolute joints it is usually rad/s. |

Example:

```
ros2 launch raybot_bringup robot.launch.py model:=rb_a630_2 \
  jtc_trajectory_tolerance:=0.8 \
  jtc_goal_tolerance:=0.25 \
  jtc_goal_time:=3.0 \
  jtc_stopped_velocity_tolerance:=0.2
```

## Overall Architecture

```
robot_controller\
├── action
│   ├── ArmMotion.action
│   ├── ExecuteTrajectory.action
│   └── ProgramCommand.action
├── config
├── include
│   └── robot_controller
│   └── arm_motion_controller.hpp
├── msg
│   ├── CommandResult.msg
│   ├── JointStateSeed.msg
│   └── PlannedTrajectory.msg
├── src
│   ├── action_recorder.cpp
│   ├── arm_motion_controller.cpp
│   ├── arm_motion_server.cpp
│   ├── cli_controller.cpp
│   ├── object_grasp_node.cpp
│   └── robot_controller.cpp
└── srv
    ├── ClearProgram.srv
    ├── DeleteProgramSteps.srv
    ├── ExecuteCommand.srv
    ├── GetLastResult.srv
    ├── PerformGrasp.srv
    ├── PerformPlace.srv
    ├── ProgramEdit.srv
    └── StopMotion.srv
```

### Core Components and Relationships

The `robot_controller` package is the core of the desktop motion control software. It consists logically of the core execution module ( `arm_motion_controller` ) and three upper-layer driver modules for different use cases. These modules share unified interfaces through ROS 2 topics, services, and actions, forming the complete motion control system. Controller modes are arbitrated by `raybot_control_mode_manager` ; PP-enabled grippers use the dedicated `raybot_pp_controller/ProfilePositionController` Profile Position Action.

The current version uses the `arm_motion_supervisor` single-process hosting model. At startup, the supervisor creates and manages component nodes such as `arm_motion_controller`, `cli_controller`, `action_recorder`, and `object_grasp_node` in one process. The core module `arm_motion_controller` handles motion planning and execution, while the upper-layer modules generate commands based on their own responsibilities and delegate final execution to the same backend, which keeps overall motion behavior consistent and reusable.

**Upper-layer driver module introduction:**

1. **cli_controller**: A command-line interaction module that receives robotic arm motion commands through topics and services, parses them, and forwards them to the core execution node. It is suitable for debugging and manual operation.

2. **action_recorder**: A motion programming and playback module used to create, edit, and execute complex motion sequences. It supports saving and loading programs and provides service and action interfaces, making it suitable for repeated multi-step tasks.

3. **object_grasp_node**: Disabled for the standard A630-2. Projects with vision and gripper extensions can integrate this module separately.

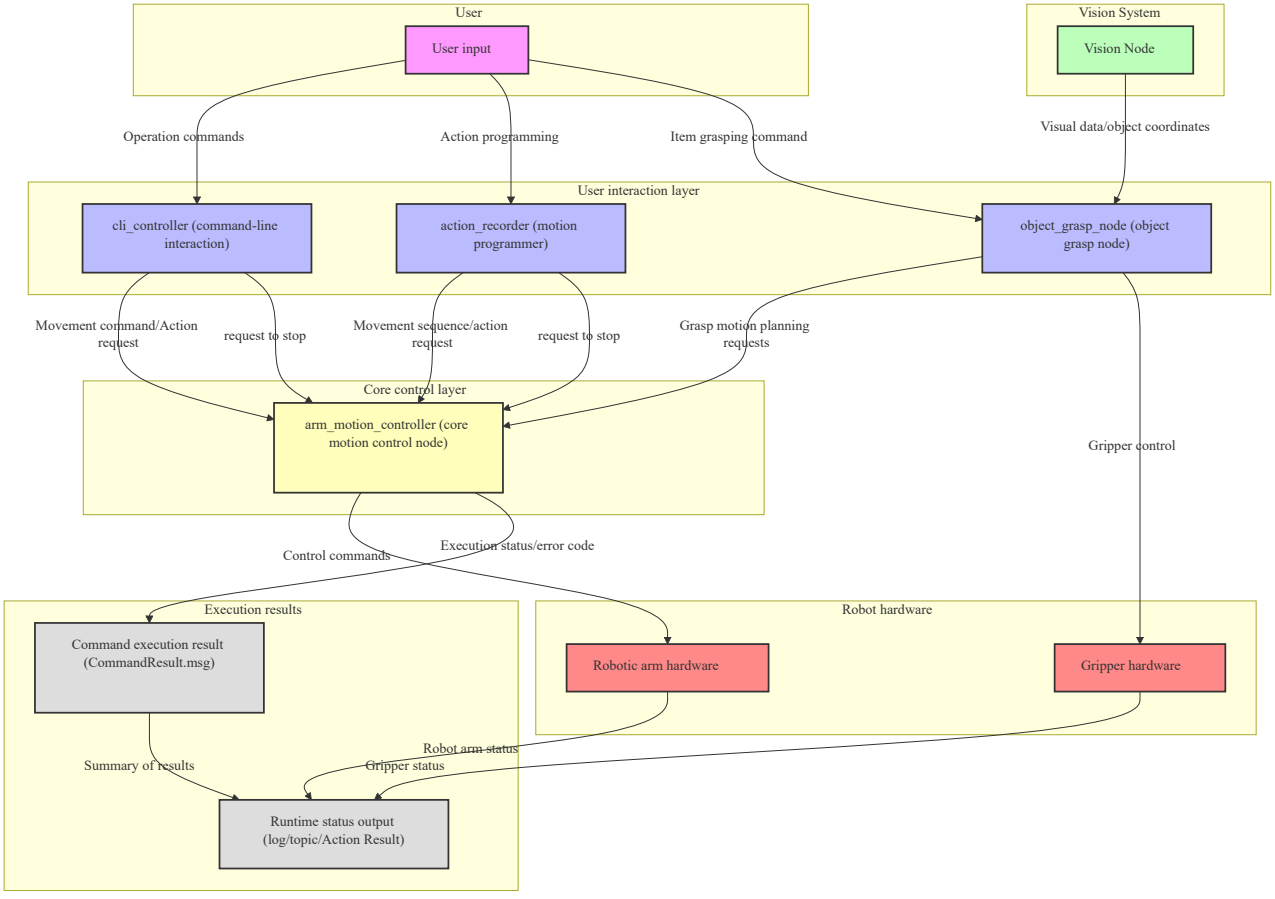
## Module Function Comparison Table

| Module Name | Function | External Interface |
| --- | --- | --- |
| arm_motion_controller | Core motion control node responsible for motion planning and execution, providing unified motion control and action service interfaces | /arm_motion_controller/result ( robot_controller/msg/CommandResult.msg ), /arm_motion_controller/get_last_result ( robot_controller/srv/GetLastResult.srv ), /arm_motion_controller/stop ( robot_controller/srv/StopMotion.srv ), /arm_motion_controller/arm_motion ( robot_controller/action/ArmMotion.action ), /arm_motion_controller/execute_trajectory ( robot_controller/action/ExecuteTrajectory.action ) |
| cli_controller | Command-line interaction node for debugging and manual operation scenarios | Topic Subscriber /cli_controller/cmd ( std_msgs::msg::String ), Service /cli_controller/execute_command ( robot_controller/srv/ExecuteCommand ) |
| action_recorder | Motion programming and playback node used to create, edit, and execute complex motion sequences, with save/load support | Topic Subscriber /action_recorder/cmd ( std_msgs::msg::String ), Service /action_recorder/edit ( robot_controller/srv/ProgramEdit.srv ), Service /action_recorder/clear_program ( robot_controller/srv/ClearPrograma.srv ), Service /action_recorder/delete_steps ( robot_controller/srv/DeleteProgramSteps.srv ), Service /action_recorder/plan_program ( robot_controller/srv/PlanProgram.srv ), Action /action_recorder/command ( robot_controller/action/ProgramCommand.action ) |

| Module Name | Function | External Interface |
| --- | --- | --- |
| object_grasp_node | Optional grasping extension; disabled for the standard A630-2 | Available only in project-specific extension configurations |
| raybot_control_mode_manager | Arbitrates global and per-group controller switching so that one joint is claimed by only one controller at a time | /raybot_robot/switch_to_* , /raybot_robot/switch_<group>_to_* , /raybot_robot/get_<group>_control_mode （ std_srvs/srv/Trigger ） |
| raybot_pp_controller | Provides single-axis gripper position goals, profile parameters, feedback, and reached results for PP-enabled joints | /l_hand_pp_controller/profile_position , /r_hand_pp_controller/profile_position （ raybot_pp_controller/action/ProfilePosition ） |

The standard rb_a630_2 does not load the gripper PP controllers listed above; these interfaces are provided only as a reference for configurations extended with grippers.

## System Architecture Diagram




## Complete System Startup

**Startup method**: Use raybot_bringup robot.launch.py to start the full system. This launch file starts the bringup management flow, launches arm_motion_supervisor

under the manager, and then lets the supervisor host component nodes such as `arm_motion_controller`, `cli_controller`, `action_recorder`, and `object_grasp_node` in-process.

```
source /home/raybot/raybot_core_ws/install/setup.bash
ros2 launch raybot_bringup robot.launch.py \
  model:=rb_a630_2 \
  use_rviz:=true
```

**Notes**:

- This launch file is located in the `raybot_bringup` package
- Select the robot model with `model:=rb_a630_2`
- All required configuration parameters are loaded automatically
- Control components are hosted uniformly by `arm_motion_supervisor`, reducing cross-process communication complexity

### Global Control Mode Switching

After startup, the global mode-switching services switch the arm and head control modes together. The supported global modes are `moveit`, `position`, `velocity`, and `effort`; a query service reads the current whole-robot mode. Global services affect all enabled planning groups. For PP-enabled grippers, globally switching to `moveit` temporarily activates the gripper JTC; call the gripper-specific PP service afterward to restore PP.

| Target Mode | Service Name | Command | Description |
| --- | --- | --- | --- |
| moveit | /raybot_robot/switch_to_moveit | ros2 service call /raybot_robot/switch_to_moveit std_srvs/srv/Trigger "{}" | Switch the full robot to MoveIt trajectory control mode. |
| position | /raybot_robot/switch_to_position | ros2 service call /raybot_robot/switch_to_position std_srvs/srv/Trigger "{}" | Switch the full robot to low-level position control mode. |
| velocity | /raybot_robot/switch_to_velocity | ros2 service call /raybot_robot/switch_to_velocity std_srvs/srv/Trigger "{}" | Switch the full robot to low-level velocity control mode. |
| effort | /raybot_robot/switch_to_effort | ros2 service call /raybot_robot/switch_to_effort std_srvs/srv/Trigger "{}" | Switch the full robot to low-level torque/effort control mode. |
| Current mode query | /raybot_robot/get_control_mode | ros2 service call /raybot_robot/get_control_mode std_srvs/srv/Trigger "{}" | Query the current control mode of the full robot. |

**Usage notes**:

- `get_control_mode` returns the current full-robot mode state.
- If the result is `mixed`, the planning groups are not currently using the same mode.
- Switching to `moveit` requires `move_group` to already be running; otherwise the request will fail.

**Service response fields**:

All five services use `std_srvs/srv/Trigger` and return the following fields:

| Field | Type | Description |
| --- | --- | --- |
| success | bool | Whether the call succeeded: mode-switching services report whether the mode changed, and the query service reports whether the current mode was read successfully. |
| message | string | Result description. Successful switches usually include the target mode; a successful query returns the current whole-robot mode directly. |

Common `message` values returned by `get_control_mode`:

| Example `message` | Meaning |
| --- | --- |
| moveit | The whole robot is in MoveIt trajectory control mode |
| position | The whole robot is in low-level position control mode |
| velocity | The whole robot is in low-level velocity control mode |
| effort | The whole robot is in low-level torque control mode |
| mixed | The planning groups are using different control modes |

Example query response:

```
success: true
message: "moveit"
```

### Grippers, Pan-Tilt Units, and Other End Effectors

The standard `rb_a630_2` configuration contains no gripper planning groups, pan-tilt planning groups, or PP controllers. Only the six-axis `arm` group is enabled by default. If a project adds an end effector, follow its project-specific configuration and acceptance instructions. This customer guide does not provide control commands for hardware that was not delivered.

# Planning Group and Predefined Poses

## Planning Group Introduction

The default planning group for `rb_a630_2` is `arm`, the end-effector link is `arm_6`, and the named poses include `home` and `test`. Use `group=arm` when sending commands; this model uses only `group=arm`.

A planning group is a collection of joints that identifies the part of the robot to move. The standard `rb_a630_2` currently supports only the following group:

| Planning Group Name | Corresponding Part | Description |
| --- | --- | --- |
| arm | Arm | Controls the six joints responsible for arm motion |

### Planning Group Joint Composition

For `rb_a630_2`, normal control uses the `arm` planning group. This model has no left-arm, gripper, or pan-tilt planning group by default. The joint composition is listed below:

| Planning Group | Joint Qty | Joint Names | Motor Alias |
| --- | --- | --- | --- |
| arm | 6 | arm_1_joint<br>arm_2_joint<br>arm_3_joint<br>arm_4_joint<br>arm_5_joint<br>arm_6_joint | 1100<br>1101<br>1102<br>1103<br>1104<br>1105 |

### Predefined Pose

A predefined pose, also called a named pose, is a preset robot configuration that can be selected by name without specifying individual joint angles or Cartesian coordinates.

**Features and Advantages**:

- Simplified programming: Using names instead of complex coordinate values, improves programming efficiency.
- Consistency: Ensures that each call reaches the same precise position.
- Reliability: Verified safe postures reduce movement risks.
- Ease of use: Suitable for quick testing and manual operation scenarios.

**Named poses currently defined in the system:**

- The arm supports the `home` and `test` poses. `home` is the robot's zero pose and its initial safe pose.

Using predefined poses can simplify robot programming, improve motion consistency and reliability, and is particularly suitable for debugging and routine operation scenarios.

## Planner Configuration

The motion planner generates an executable joint-space or Cartesian trajectory from the start state to the target state. The Raybot desktop motion control software supports multiple planners for different motion requirements.

### Supported Planner Types

| Planner Type | Planner ID | Feature | Applicable Scenarios |
|---|---|---|---|
| **PILZ Industrial motion planner** | pilz_ptp | Industrial-grade point-to-point motion planning, fast and smooth path generation | Suitable for most industrial applications, especially tasks requiring rapid positioning. |
| **PILZ Industrial motion planner** | pilz_lin | Industrial-grade linear motion planning, high precision, strict adherence to a straight path, and smooth and reliable motion | Suitable for applications requiring precise linear motion, such as welding, spraying, and assembly. |
| **PILZ Industrial motion planner** | pilz_circ | Circular TCP motion defined by a center or interim point | Single-arm short arcs, semicircles, and major arcs |
| **OMPL Sampling Planner** | RRTConnectkConfigDefault | Based on sampling-based random tree algorithms, it excels at obstacle avoidance in complex environments and can handle high-dimensional planning problems | Obstacle avoidance planning in complex environments, especially in scenarios with multiple obstacles, such as confined space operations and dynamic obstacle avoidance. |

#### PIPELINE_ID / PLANNER_ID CONFIGURATION

Both CLI Controller and Action Recorder support two planner configuration styles:

| Style | Example | Description |
| --- | --- | --- |
| Explicit configuration | pipeline_id=pilz_industrial_motion_planner planner_id=PTP | Most explicit and recommended for troubleshooting planner selection |
| Planner alias | planner=ptp / planner=lin / planner=circ | Automatically expands to the corresponding Pilz PTP / LIN / CIRC configuration |

Common mappings:

| Motion Type | Recommended Syntax | Equivalent Explicit Syntax |
| --- | --- | --- |
| Pilz PTP | planner=ptp | pipeline_id=pilz_industrial_motion_planner planner_id=PTP |
| Pilz LIN | planner=lin | pipeline_id=pilz_industrial_motion_planner planner_id=LIN |
| Pilz CIRC | type=arc planner=circ | pipeline_id=pilz_industrial_motion_planner planner_id=CIRC |
| OMPL | planner=ompl | pipeline_id=ompl |

Notes:

- Use the full pipeline name pipeline_id=pilz_industrial_motion_planner ; non-standard abbreviations such as pipeline=pilz are not recommended.
- If only planner_id=PTP or planner_id=LIN is provided while the current default pipeline is not Pilz, the planner configuration may be inconsistent. Use planner=ptp/lin , or specify both pipeline_id and planner_id .
- exec=0 plans only and does not execute the trajectory. Use it for first validation. exec=1 executes the trajectory after planning succeeds.

### Analysis of the Reasons for Planning Failure

When using different types of motion planning algorithms, planning failures may occur. Understanding common causes of failure in advance can help in adjusting planning parameters, optimizing environmental modeling, and improving the overall planning success rate.

#### Reasons for the failure of the LIN (line) command

- **Kinematics and workspace limitations**: Some points along the linear trajectory may be outside the workspace or lead to joint limits being exceeded/unreachable.

- **Limited obstacle avoidance capabilities**: LIN forces the end effector to move in a straight line; if this straight line passes through an obstacle, it is usually impossible to avoid a collision by "going around" it.
- **Dynamic constraints are not satisfied.**: Setting the velocity and acceleration too high, or having insufficient planning time/sampling precision, may result in the trajectory failing to be generated or failing to be parameterized by time.
- **Singularity effects**: When the trajectory passes through or approaches a singularity point, the required joint velocities may increase abnormally, leading to failure during the planning or execution phase.

#### Reasons for point-to-point command failure

- **Target pose/joint target is unreachable.**: The target point is outside the workspace, or there is no feasible solution under the given constraints (limitations, posture requirements, etc.).
- **Collision detection failed.**: The planned path collides with the environment or the robot's own structure.
- **The planner and parameter configuration do not match.**: Choosing an inappropriate planner type, or setting parameters such as planning time, sampling density, and constraint weights incorrectly, can make it difficult to find a feasible path.

#### Suggestions for improving the success rate of planning

Based on the reasons for the planning failures mentioned above, here are some general suggestions for improving the success rate of planning:

1. **Optimize path design**

- Reduce the number of waypoints and keep only the key points.
- Keep adjacent waypoints at a reasonable distance.
- Avoid placing waypoints on the workspace boundary.
- Adjust straight-line trajectories to avoid singularities and obstacles.

2. **Adjust planning parameters**

- Reduce speed and acceleration when joint motion becomes infeasible.
- Increase planning time, especially in complex scenes.

3. **Choose the planner appropriately**

- In open environments and for routine point-to-point moves, try `pilz_ptp` first.
- For process motions that require the end effector to follow a strict straight line, try `pilz_lin` first.
- If there are obstacles, narrow spaces, or detours are needed, prioritize `ompl`.
- If one planner repeatedly fails, do not keep retrying the same parameter combination. Change the planner or simplify the motion first.

#### Comparison of Planner Features

| Comparison Dimensions | PILZ Planner | OMPL Planner |
|---|---|---|
| **Planned speed** | fast | medium |
| **Path quality** | smooth and predictable | Smooth, with random generation for each plan. |
| **Obstacle avoidance capability** | Basic obstacle avoidance | Powerful obstacle avoidance capabilities, suitable for complex environments |
| **Applicable occasions** | Industrial applications | Scientific research and complex environments |
| **Real-time** | high | medium |

## Arm Motion Controller (Core execution node)

### Module Functions

`Arm Motion Controller` is the core execution node of the desktop motion control software. It handles arm motion planning and execution, encapsulates the motion planning interface, and provides a unified service so that multiple upper-layer modules can safely share arm resources.

**Supported motion types**:

- **PTP Motion**: Point-to-point motion: The robotic arm's end effector reaches the target position via the fastest path.
- **LIN Motion**: It moves along a straight line trajectory, suitable for scenarios requiring precise path control.
- **Joint Motion**: Direct control of joint angles is suitable for debugging and precise position adjustment.
- **Return to Zero**: Return the robotic arm to zero position.
- **Motion Stop**: Immediately stop the current motion.

#### Interaction Process

Arm Motion Controller is the core execution node of the system, its interaction process is as follows:

1. **Input**:

   - Receiving motion commands from the upper-level module.
   - Supports multiple motion types: PTP motion, LIN linear motion, joint motion, return-to-zero operation, and motion stop.

2. **Process**:

   - Parse command-line parameters and configure MoveGroupInterface
   - Select the appropriate planner and planning pipeline
   - Perform motion planning
   - Execute the planned trajectory (or only plan without execution)
   - Monitor the execution process in real time

3. **Output**:

   - The execution status is returned in real time through action feedback.
   - The execution result is published via the `/arm_motion_controller/result` topic.
   - The latest result can also be queried through the `/arm_motion_controller/get_last_result` service.

4. **Position in the overall system process**:

   - Located in the core control layer, it receives commands from the user interaction layer.
   - Controlling the movement of the robot's hardware layer.

### External Interface Description

Arm Motion Controller exposes the following core result interfaces:

| Interface type | Name | Message/Service Type | Function Description |
| --- | --- | --- | --- |
| Topic Publisher | /arm_motion_controller/result | robot_controller/msg/CommandResult | Publishes the latest planning/execution result |
| Service | /arm_motion_controller/get_last_result | robot_controller/srv/GetLastResult | Queries the latest result, optionally filtered by `group` or `command_id` |
| Service | /arm_motion_controller/stop | robot_controller/srv/StopMotion | Requests a stop for motion/planning associated with the specified group or command |

**/arm_motion_controller/get_last_result** **RESPONSE**

Response fields:

| Field | Type | Description |
| --- | --- | --- |
| available | bool | Whether a matching result was found. `false` means no matching result is currently cached. |
| result | robot_controller/msg/CommandResult | The latest matching motion result; valid only when `available=true` . |

Fields in `CommandResult` :

| Field | Type | Description |
| --- | --- | --- |
| command_id | string | Unique command ID |
| group | string | Planning group name, such as arm |
| goal_type | string | Goal type, such as named , joints , pose , or stop |
| success | bool | Whether planning/execution succeeded |
| moveit_code | int32 | MoveIt return code for diagnosing planning failures |
| cartesian_fraction | float64 | Cartesian path completion fraction; usually 0.0 for non-Cartesian operations |
| executed | bool | Whether the trajectory was actually executed; false for planning without execution |
| message | string | Result description or failure reason |
| start_time | builtin_interfaces/Time | Start time |
| end_time | builtin_interfaces/Time | End time |

Example response:

```yaml
available: true
result:
  command_id: "cmd_20260521_001"
  group: "arm"
  goal_type: "pose"
  success: true
  moveit_code: 1
  cartesian_fraction: 0.0
  executed: true
  message: "Execution completed"
```

**/arm_motion_controller/stop** RESPONSE

Response fields:

| Field | Type | Description |
| --- | --- | --- |
| accepted | bool | Whether the stop request was accepted; may be `false` if the target group or command does not exist |
| success | bool | Whether the stop/cancellation logic was successfully triggered |
| message | string | Result description, such as execution interrupted, planning cancelled, or no stoppable target |

## CLI Controller (Command-line robotic arm motion control)

### Module Functions

CLI controller is a command-line interface module that allows users to directly control the robotic arm's movement via topic or service commands. It is suitable for debugging and manual control scenarios, supporting various motion types and parameter configurations.

**Core functions**:

- Receive and parse motion commands in command-line format.
- Supports multiple motion types:
  - Predefined posture movements(type=named)
  - Joint space movement(type=joints)
  - Cartesian point motion(type=pose)
  - Circular motion(type=arc)
  - Return to zero operation
- Motion stop function

### Command Format and Syntax

1. **Command format**

The commands use a `key=value` key-value pair format, with multiple key-value pairs separated by spaces. The motion type is specified using the `type` field:

```
group=arm type=named target=home exec=1
```

## 2. Supported command types

| Command types | Key words | Description | Example |
|---|---|---|---|
| Predefined posture movements | type=named | Move to a predefined posture. | group=arm type=named target=home exec=1 |
| Joint space movement | type=joints | Directly control joint angles. | group=arm type=joints joints=arm_1_joint:0.5,arm_2_joint:0.5 exec=1 |
| Cartesian point motion | type=pose | Move to the specified Cartesian coordinates | group=arm type=pose pose=0.4,0.2,0.3,0,1.57,0 frame=base_link exec=1 |
| Circular motion | type=arc | Single-arm center or interim-point arc | Tested complete commands |
| Stop motion | type=stop | Stop the current movement. | group=arm type=stop |

## 3. Common command parameters

| Parameter Name | Type | Description |
| --- | --- | --- |
| group | string | The standard A630-2 uses `arm` |
| type | string | Motion type(named/joints/pose/arc/cart/stop) |
| target | string | Predefined pose names(only used when type=named) |
| joints | string | Joint angle values, supporting sequential or name:value format. |
| pose | string | Cartesian coordinates(x,y,z,roll,pitch,yaw) |
| arc_mode | string | Required for `type=arc` : `center` or `interim` |
| arc_point | string | Required for `type=arc` : auxiliary point `x,y,z` in m |
| ee | string | Arc TCP link, explicitly set in the examples |
| frame | string | Reference coordinate system (default: base_link) |
| vel | double | Speed scaling factor(0-1) |
| acc | double | Acceleration scaling factor(0-1) |
| exec | bool | Should the planned result be executed? (true/false) |
| planner | string | Planner alias. Common values: `ptp` , `lin` , `circ` , `ompl` |
| planner_id | string | Planner ID, such as Pilz `PTP` / `LIN` / `CIRC` or OMPL `RRTConnectkConfigDefault` |
| pipeline / pipeline_id | string | Planning pipeline. Supported values include `pilz_industrial_motion_planner` and `ompl` |
| lin_position_only | bool | Used only for Pilz LIN. Set to `1` to use target positions while keeping the current orientation |

Note: CLI commands do not support the `replan` key. If you pass it in, it will be treated as an invalid parameter.

### Applicability of Dual-Arm Commands

The standard A630-2 supports single-arm commands only. Use `group=arm`, validate planning with `exec=0`, and then execute with `exec=1`. This customer guide does not provide dual-arm `multi_pose` examples.

### Circular Motion (Pilz CIRC)

`type=arc` moves the selected TCP along a circular arc. The start is FK from the joint feedback captured for this request; `pose` is the complete endpoint pose, and `arc_point` is the center or an interim point. Arcs default to `pilz_industrial_motion_planner/CIRC`; the examples explicitly set `planner=circ`.

| Field | Meaning and Units |
| --- | --- |
| arc_mode=center | `arc_point=x,y,z` is the center; selects a short arc strictly below 180° |
| arc_mode=interim | `arc_point=x,y,z` lies on the arc; supports short arcs, semicircles, and major arcs; it is not a stop point |
| pose=x,y,z,roll,pitch,yaw | Endpoint position in m, RPY in rad; orientation interpolates from start to endpoint |
| frame / ee | This section uses `base_link` and an explicit TCP link; CLI uses the same `frame` for auxiliary point and endpoint |
| vel / acc | Velocity and acceleration scaling in `(0,1]`; equal scaling does not imply equal TCP speed across models |
| exec=0 / exec=1 | Plan only / execute after successful planning; each CLI execution replans from current feedback |
| planner=circ | Equivalent to `pipeline_id=pilz_industrial_motion_planner` `planner_id=CIRC`; do not substitute OMPL or PTP |

- Only single-arm chain groups are supported. On dual-arm robots select `l_arm` or `r_arm`; `dual_arms` does not support synchronized arcs. The runtime must load the Pilz pipeline and Cartesian limits.
- Center mode cannot define semicircles or major arcs; use interim mode for those. Full circles are unsupported. Center-mode start and end radii must differ by no more than `0.00001 m`.

- Do not use `type=pose planner=circ` or set `lin_position_only=1` for an arc. Incomplete or incompatible requests fail without changing the motion to PTP.
- Pilz may reject very small, shallow three-point arcs because their plane is not properly defined. These examples use a 50 mm radius; preserve their scale and coordinate precision.

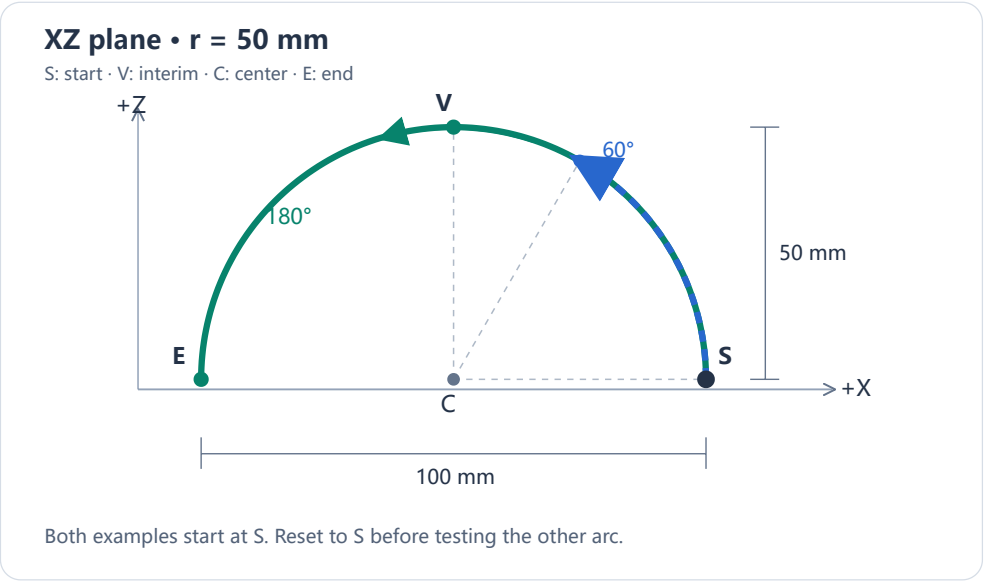
### Web Circular Motion Control

On the Robot Control page, select a single-arm planning group and open "Circular CIRC". The current model or variant supplies the group and TCP. Composite groups and grippers do not support arcs.

1. Confirm the reference frame and inspect "Start S (actual TCP feedback)". The start comes from actual feedback, rather than the target ghost. Use "Refresh start" to read it again.
2. Select center or interim mode, then enter the auxiliary XYZ point and complete endpoint XYZ/RPY, in m and rad. "Read current TCP as endpoint" copies once and leaves the target editable; the endpoint must still differ from the start.
3. Inspect the dashed arc with S, C/V, and E markers, radius, and sweep. This is geometry only and does not confirm IK, collision, or dynamic feasibility.
4. In simulation mode, choose "Plan only"; in real mode the button reads "Plan and execute". The result distinguishes planned, executed, failed, and unconfirmed outcomes. Acceptance does not mean completion.
5. Choose "Add arc action" to add a step. Refreshing or loading the list preserves CIRC. TCP, frame, and scaling omitted from the summary are shown as stored in the program. Step playback uses the original backend parameters; in simulation mode, selections containing an arc only plan.
6. Set up the arc again after changing model, planning group, or TCP. Applying a new reference frame clears the endpoint and auxiliary point; unavailable TF blocks submission. Arc targets retain their internal precision and are not rebuilt from the rounded start display.

Use the examples below as coordinate and start-state references for initial integration. Complete the matching preparation move before entering the arc on the web page. In an action program, the arc starts at the preceding step's end state.

## Viewing and Reproduction Steps




1. Select the matching model and TCP, then view the XZ plane along the Y axis. The green semicircle goes from S through V to E; the blue short arc starts from the same S.
2. Plan the preparation move, check its path, and execute it to reach the start state. Planning alone does not change joint feedback; moving the blue goal ghost does not change the arc start either.
3. At the prepared state, send example A with `exec=0` and inspect the planned trajectory in RViz MotionPlanning. `success=true`, `result_received=true`, and `result.outcome=1` (`PLANNED`) confirm planning only.
4. The Web 3D model shows the actual joint feedback; `exec=0` does not move it. To observe motion, change the arc command to `exec=1`. `result.outcome=3` (`EXECUTED`) and `result.executed=true` confirm execution and feedback acceptance; `accepted=true` only confirms request acceptance.
5. Return to the same preparation state before each demonstration. Examples A and B both start at S; do not execute them consecutively without resetting the start.

These values come from the 2026-10-28 three-model tests and passed real MoveIt/Pilz CIRC planning and TCP FK checks. CLI commands and action programs were retested at the precision printed here. Tests used model feedback without physical hardware; passing means planning and interface acceptance. All examples default to `exec=0`; the arc TCP speed limit is approximately 20 mm/s.

## RB-A630-2: Desktop Arm Example

The model is `rb_a630_2`, the arc group is `robot_1_arm`, and the TCP is `arm_6`. This is the actual group in the tested model SRDF, differing from `arm` in other examples in this guide. Use `robot_1_arm` for this section; check the actual planning group and TCP before using another configuration. After preparation, the TCP starts at approximately `S=(0.225774561,0.056496106,0.470095219) m`. The passing source cases are start 0, XZ plane, radius 50 mm: case 12 (center, 60°) and case 14 (interim, semicircle).

**Prepare the start state:** These joint targets are in rad. Plan first; after checking the path, change `exec=0` to `exec=1` in this preparation command and wait for joint feedback to reach the target.

```
ros2 service call /cli_controller/execute_command \
  robot_controller/srv/ExecuteCommand \
  "{command: 'group=robot_1_arm type=joints
joints=0.30000000,1.20000000,1.00000000,0.30000000,0.50000000,0.20000000 planner=ptp vel=0.02
acc=0.017777777777777778 exec=0', wait_for_result: true, timeout: 60.0}"
```

**Example A: interim-point semicircle; observe this one first.** From S, pass through V 50 mm higher and finish at E with X decreased by 100 mm and Z back at its initial height. Y and the TCP orientation stay unchanged.

```
ros2 service call /cli_controller/execute_command \
  robot_controller/srv/ExecuteCommand \
  "{command: 'group=robot_1_arm type=arc arc_mode=interim
arc_point=0.175774561,0.056496106,0.520095219
pose=0.125774561,0.056496106,0.470095219,-0.773447403,-0.860998183,-2.092098530 frame=base_link
ee=arm_6 planner=circ vel=0.02 acc=0.017777777777777778 planning_time=5 exec=0', wait_for_result:
true, timeout: 60.0}"
```

**Example B: center-defined short arc.** Run the preparation move again to return to S before requesting this 60° arc. Relative to S, the endpoint decreases X by 25 mm and increases Z by approximately 43.30 mm.

```
ros2 service call /cli_controller/execute_command \
  robot_controller/srv/ExecuteCommand \
  "{command: 'group=robot_1_arm type=arc arc_mode=center
arc_point=0.175774561,0.056496106,0.470095219
pose=0.200774561,0.056496106,0.513396489,-0.773447403,-0.860998183,-2.092098530 frame=base_link
ee=arm_6 planner=circ vel=0.02 acc=0.017777777777777778 planning_time=5 exec=0', wait_for_result:
true, timeout: 60.0}"
```

**Add to an action program:** With the start state reached, add example A as an arc step. If the program must also move to the start, add the joint preparation step above before it.

```
ros2 service call /action_recorder/edit \
  robot_controller/srv/ProgramEdit \
  "{command: 'add group=robot_1_arm type=arc arc_mode=interim
arc_point=0.175774561,0.056496106,0.520095219
pose=0.125774561,0.056496106,0.470095219,-0.773447403,-0.860998183,-2.092098530 frame=base_link
ee=arm_6 planner=circ vel=0.02 acc=0.017777777777777778 planning_time=5'}"
```

After adding the step, plan every step in the current program. This call only plans; it does not execute. Any other steps already in the program are also included.

```
ros2 service call /action_recorder/plan_program \
  robot_controller/srv/PlanProgram \
  "{use_current_state_start: true, steps: []}"
```

Replan after changing the auxiliary point, endpoint, reference frame, or preceding start state.

#### Interpreting Results and Failures

A reachable endpoint does not guarantee a reachable arc. For `NO_IK_SOLUTION` or code `-31`, check both the endpoint and intermediate poses. Reducing speed can help with velocity/acceleration limits, but IK branch changes near joint limits may still fail at lower speed. `Plane for motion is not properly defined` indicates circle construction failure; check the geometry or use center mode for a short arc. Arcs do not use Cartesian completion fractions for acceptance, so `cartesian_fraction=-1` does not mean failure.

Download the start states, commands, and test results. Coordinates depend on the stated model, TCP, and full start state; they are not generic targets for other variants or tools.

### External Interface Description

CLI controller provides the following external interfaces for direct invocation by external systems or users:

| Interface type | Name | Message/Service Type | Function Description |
|---|---|---|---|
| Topic Subscriber | /cli_controller/cmd | std_msgs::msg::String | Receive external command strings in `key=value ...` format |
| Service | /cli_controller/execute_command | robot_controller/srv/ExecuteCommand | Execute a single motion command and return the result, including `type=stop` stop requests |

Note: CLI does not expose standalone `/cli_controller/result`, `/cli_controller/get_last_result`, `/cli_controller/stop_broadcast`, or `/cli_controller/stop` interfaces. Results are returned directly from `execute_command`, and runtime status can be observed through `/arm_motion_controller/result`.

#### /cli_controller/execute_command RESPONSE

Response fields:

| Field | Type | Description |
|---|---|---|
| accepted | bool | Whether the command was accepted. Usually `false` for parsing errors, busy resources, or invalid stop targets. |
| command_id | string | Unique command ID; may be empty if the command fails before parsing. |
| success | bool | Whether the call ultimately succeeded. With `wait_for_result=false`, this may only indicate successful command dispatch. |
| result_received | bool | Whether the final execution-layer result has been received. |
| timed_out | bool | Whether waiting for the result timed out; meaningful only with `wait_for_result=true`. |
| result | robot_controller/msg/CommandResult | Final result details; contains only basic fields or defaults if no result was received. |
| message | string | Call result, failure reason, or timeout information. |

The `result` structure is the same `CommandResult` returned by `/arm_motion_controller/get_last_result`, including `command_id`, `group`, `goal_type`, `success`,

`moveit_code` , `cartesian_fraction` , `executed` , `message` , `start_time` , and `end_time` .

Example response when waiting synchronously for the result:

```
accepted: true
command_id: "cmd_20260521_002"
success: true
result_received: true
timed_out: false
result:
  command_id: "cmd_20260521_002"
  group: "arm"
  goal_type: "pose"
  success: true
  moveit_code: 1
  cartesian_fraction: 0.0
  executed: true
  message: "Execution completed"
message: "Execution completed"
```

### Complete Test Command

**See below for examples of test commands.** Example command set

### Implementation Details

- **Implementation file**: `src/cli_controller.cpp`
- **Core functions**: Commands are received via the `/cli_controller/cmd` topic and the `/cli_controller/execute_command` service, parsed, and then used to execute motion control.
- **Features**: It provides a concise command-line interface, suitable for debugging and manual operation scenarios.
- **Command Format**: Using a `key=value` key-value pair format, it supports various exercise types and parameter configurations.

## Action Recorder (Motion programmer)

### Module Functions

Action Recorder is a module for programming, recording, editing, and replaying robotic arm movements. It allows users to create complex motion sequences and supports management and execution through command or service interfaces.

**Core functions**:

- Recording and editing of motion sequences
- Supports combinations of various motion types
- Provides functions for saving and loading motion programs
- Supports pre-planned and real-time planning modes
- Supports waiting and delays between motion steps
- Supports planning, execution, and playback of motion steps; the standard A630-2 has no gripper PP steps

### Configuration Parameters

| Parameter Name | Type | Default Value | Description |
|---|---|---|---|
| group | string | "arm" | Default robotic arm group |
| planner_mode | string | "ompl" | Default planner mode |
| vel_scaling | double | 0.2 | Default speed scaling factor |
| acc_scaling | double | 0.2 | Default acceleration scaling factor |
| planning_time | double | 5.0 | Default planning time (seconds) |
| execute | bool | true | Should the planning results be executed by default? |

### External Interface Description

Action Recorder provides various external interfaces, supporting interaction via command line, Topic, Service, and Action:

#### Topic Interface

| Interface type | Name | Message type | Function Description |
|---|---|---|---|
| Topic Subscriber | /action_recorder/cmd | std_msgs::msg::String | Receive external command strings in `key=value ...` format |

The current implementation does not provide a separate program status topic. Execution progress is returned through action feedback on `/action_recorder/command`, and edit operations return results through service responses.

#### Command Format

Action Recorder supports the following command types:

| Command keywords | Function Description | Example |
| --- | --- | --- |
| add | Add action steps | add type=pose pose=0.4,0.2,0.3,0,1.57,0 wait=1.5 |
| del | Delete action steps | del index=1 |
| wait | Add a waiting step | wait duration=2.0 |
| play | Execute the action sequence, including repeated playback | play exec=1 replan=0 repeat_count=3 |
| save | Save action program | save file=/tmp/program.txt |
| load | Loading action program | load file=/tmp/program.txt |
| clear | Clear the action program. | clear |

**Action steps and command format**:

```
add type=pose pose=0.4,0.2,0.3,0,1.57,0 planner=pilz_ptp wait=1.5 group=arm
```

Supported motion types include:

- type=named : Predefined posture movements
- type=joints : Joint space motion
- type=pose : Cartesian point motion
- type=arc : Arc steps use the same fields as CLI arc commands

The standard A630-2 does not support gripper PP motion steps.

### Service Interface

| Service Name | Service Type | Function Description |
| --- | --- | --- |
| /action_recorder/edit | robot_controller::srv::ProgramEdit | Edit the action program; saving and loading are also handled through save and load commands |
| /action_recorder/clear_program | robot_controller::srv::ClearProgram | Clear the current action program |
| /action_recorder/delete_steps | robot_controller::srv::DeleteProgramSteps | Delete selected program steps |
| /action_recorder/plan_program | robot_controller::srv::PlanProgram | Validate planning for selected steps only, without execution |

### Service Response Fields

Response fields for /action_recorder/edit :

| Field | Type | Description |
| --- | --- | --- |
| success | bool | Whether the editing command completed successfully |
| code | int32 | Internal result code distinguishing invalid parameters, busy state, execution failure, and other outcomes |
| message | string | Result description, such as parsing, save/load results, or failure reasons |
| steps | string[] | Readable descriptions of the current program steps; especially useful after `list`, `add`, `del`, or `load` |

Response fields for `/action_recorder/clear_program`:

| Field | Type | Description |
| --- | --- | --- |
| success | bool | Whether the motion program was cleared successfully |
| message | string | Result description, such as `Program cleared` or `Program already empty` |
| remaining_steps | uint32 | Number of remaining steps after clearing; normally `0` on success |

Response fields for `/action_recorder/delete_steps`:

| Field | Type | Description |
| --- | --- | --- |
| success | bool | Whether the deletion succeeded |
| message | string | Result description, including skipped out-of-range indices or failure reasons |
| removed_count | int32 | Number of steps actually removed |
| failed_indices | int32[] | Step indices that could not be deleted, commonly because they were out of range |
| remaining_steps | string[] | Descriptions of remaining steps; the full list is returned only when `report_remaining=true` |

Response fields for `/action_recorder/plan_program`:

| Field | Type | Description |
|---|---|---|
| success | bool | Whether all selected steps were planned successfully |
| failed_step | int32 | First failed step number (1-based); `0` on success |
| message | string | Planning result description or failure reason |

Example response from `/action_recorder/edit`:

```
success: true
code: 0
message: "Command completed"
steps:
  - "1. pose group=arm frame=base_link ee=arm_6 ..."
```

#### ACTION INTERFACE

| Action name | Action type | Function Description |
|---|---|---|
| /action_recorder/command | robot_controller::action::ProgramCommand | Execute the action program and provide real-time execution feedback. |

### Complete Test Command

**See below for examples of test commands.** Example command set

### Implementation Details

- **Implementation file**: src/action_recorder.cpp
- **Core functions**: Maintains action procedures and states, and supports editing and executing action sequences.
- **Key functions**:
  - handleEditService(): Processing action program editing service requests.
  - handlePlayCommand(): Processing action program execution commands.
  - executeProgram(): Execute the action sequence.
- **Feature**: Provides services and action servers, supporting saving/loading action programs.

### Interaction Process

The motion programmer acts as the motion sequence management module and follows the workflow below:

1. **Input**:
   - Receives command strings from the `/action_recorder/cmd` topic
   - Supports `add`, `del`, `wait`, `play`, `save`, `load`, and `clear`
   - Receives editing requests through services
   - Receives execution requests through the action interface
2. **Process**:
   - Add, delete, and clear action steps
   - Parse and validate program command parameters
   - Execute the action sequence step by step
   - Save and load action programs from files
3. **Output**:
   - Provides real-time execution progress through `/action_recorder/command` action feedback
   - Returns operation results through service responses
   - Records detailed execution logs

## Optional Extensions

The standard `rb_a630_2` delivery configuration disables grippers, pan-tilt units, vision grasping, and the object grasp node by default. This customer guide provides no executable commands for those features. If the project includes the corresponding hardware and software extensions, follow its project-specific configuration and acceptance instructions.

## Troubleshooting

### Clearing Motor Errors

With robot control started and the hardware plugin still running, call `clear_motor_errors` to clear errors manually. The **Clear errors** button in the web robot control toolbar sits between **Shutdown** and **Emergency stop** and targets all mapped motors in the current runtime instance. Wait for per-axis confirmation,

then expand the result panel below the buttons to inspect each result, error codes before and after clearing, and the final status word. The execution log also records the complete service request and response.

The service type is `ethercat_control/srv/ClearMotorErrors`; its default path is `/raybot_robot/clear_motor_errors`. For a custom `interface_ns` or multiple instances, replace the path with that of the target instance. The web interface uses the current runtime namespace automatically. Use the same `ROS_DOMAIN_ID` as the control process and source the workspace that provides this service type:

```
source /opt/ros/humble/setup.bash
source /home/raybot/raybot_core_ws/install/setup.bash

# 单轴清错：1107 仅为示例，请替换为当前电机配置的 EtherCAT alias。
ros2 service call /raybot_robot/clear_motor_errors ethercat_control/srv/ClearMotorErrors \
  "{clear_all: false, axis_aliases: [1107]}"

# 多轴清错：列表只接受 alias，不接受关节序号或内部轴索引。
ros2 service call /raybot_robot/clear_motor_errors ethercat_control/srv/ClearMotorErrors \
  "{clear_all: false, axis_aliases: [1107, 1108]}"

# 清除当前硬件插件映射的全部电机错误。
ros2 service call /raybot_robot/clear_motor_errors ethercat_control/srv/ClearMotorErrors \
  "{clear_all: true, axis_aliases: []}"
```

With `clear_all: true`, `axis_aliases` is ignored. An explicit list is deduplicated; an empty list with `clear_all: false` is rejected. An axis without a fault returns "Already fault-free" without changing its motion or enable state. The aliases 1107 and 1108 above are examples and must be replaced with those in the actual motor configuration.

The service waits for completion before responding, with one 30-second deadline for the whole batch. Set a custom client timeout above 31 seconds; the web interface waits 35 seconds. `success: true` means every target is "Cleared" or "Already fault-free." Partial success returns `false`; inspect `results` and avoid repeatedly resetting a persistent fault. Each entry includes `alias`, `pdo_profile`, `status`, `error_before`, `error_after`, `status_word`, and `message`:

| status | State | Meaning |
|---|---|---|
| 1 / CLEARED | Cleared | Three fresh feedback sets confirm zero error and exit from Fault / Fault reaction, with recovery hold confirmed |
| 2 / ALREADY_CLEAR | Already fault-free | Fresh feedback confirms no fault; axis outputs were unchanged |
| 3 / INVALID_TARGET | Invalid target | Alias is absent, out of range, or not mapped by the current hardware plugin |
| 4 / UNSUPPORTED | Unsupported | PDO profile does not declare MT_Device or CrabNova error clearing support |
| 5 / BUSY | Busy | Another clear batch is running, or the target is undergoing zeroing, parameter saving, or other maintenance |
| 6 / COMMUNICATION_FAILED | Communication failed | Target is offline, fresh feedback is unavailable, or the cyclic thread is not running |
| 7 / FAULT_REMAINS | Fault remains | A fault remains after one reset; inspect `message` for the device-specific reason |
| 8 / TIMEOUT | Timed out | Error confirmation or CSV takeover did not finish before the deadline |
| 9 / CANCELED | Canceled | Hardware deactivation, stop, or shutdown lock preempted the operation |

After a communication failure, zero-valued error fields do not confirm that the fault is cleared; check `status`. A web disconnect or timeout also means the result is unconfirmed. No automatic retry is sent; inspect actual motor state before deciding what to do next.

The PDO profile declares the device protocol used for error clearing:

- **MT_Device**: `standard`, `pvt`, `pdo260224`, `standard260330`, and `standard2026090301` interpret `0x603F` as a bitmask. Both older firmware that briefly reboots/disconnects after reset and newer firmware with continuous communication are supported.

- **CrabNova**: only `crabnova`, interpreting the vendor Fxxx single alarm code. Reset is sent only after the alarm has remained stable for at least 20 ms.
- Both protocols send at most one $0x0000 \rightarrow 0x0080 \rightarrow 0x0000$ control-word sequence per axis per call and require three distinct fresh feedback confirmations. They do not loop resets; other profiles return "Unsupported."

If another coupler branch remains powered off and Domain WKC is incomplete, fresh asynchronous SDO responses from an online target can provide confirmation. The target itself must recover to OP and exchange PDOs. A target that remains offline, cannot enter OP, or has a persistent fault will fail or time out; inspect power, cabling, and the reported alarm.

For faulted axes that need resetting, clearing discards old motion targets, PP requests, and feedforward sessions. New motion commands received during clearing are also discarded rather than replayed afterward. Previously disabled axes remain disabled; previously enabled axes enter zero-speed hold after successful clearing and CSV confirmation. Failure or timeout also retains hold without resuming old trajectories. After checking results, send a new valid motion command; feedforward control requires a new session. Non-target axes are unaffected. Shutdown and emergency stop remain available while waiting.

## Common Errors and Solutions

| Error Symptom | Possible Cause | Solution |
|---|---|---|
| Planning failed | Target point unreachable | Adjust the target position to ensure it is within the workspace. |
| Planning failed | Collision detection failed | Check for obstacles in the environment and adjust planning parameters |
| Planning was successful, but execution failed | Speed/acceleration too high | Reduce speed and acceleration parameters and add delays between movements |
| Service call failed | Node not started | Check if the relevant nodes are running correctly |
| Movement is not smooth | PID parameters are unsuitable | Adjust PID parameters |
| Motor disconnected / Slave device disconnected | High CPU load causing real-time thread timeout | First ensure safe shutdown → Restart the control software → If it still cannot be restored, power off and restart the motor drive power supply (power off for ≥5~10 seconds before powering on again) and restart the software, while also reducing the system load. |
| Motor connection lost / Communication interrupted | Poor contact of network cable/connector, inadequate shielding/grounding, excessively long cable, or proximity to strong interference sources | Replace network cable and connectors, use industrial shielded cable, optimize wiring to keep it away from motor power lines, and check grounding and shielding connections. |
| The motor(s) cannot enter OP mode, either completely or partially | Slave sequence/alias mismatch, inconsistent with the alias configuration file | Verify the slave topology and alias configuration, ensure consistency with the actual equipment, and re-export/generate the configuration and synchronize it to the device if necessary. |
| Planning is normal but trajectory tracking error is large | Joint zero position/direction configuration error, inconsistent units/gear ratios, unsuitable controller parameters | Verify joint directions, gear ratios, and encoder units, re-check URDF/ros2_control configuration, and retune control parameters. |
| MoveIt planning succeeded, but the controller rejected the trajectory | Controller not active, interface mismatch (position/velocity/torque), trajectory segments too dense | Confirm the controller status is active, check for consistent interface types, and appropriately reduce trajectory |

| Error Symptom | Possible Cause | Solution |
| --- | --- | --- |
| | | sampling density/increase time parameters |
| Increased latency and slow command response | Temporary CLI process publishing (topic pub/service call) incurs startup overhead, and DDS discovery is time-consuming | Use a "resident node" as the command entry point to reduce frequent startup/shutdown of command-line publishing, and optimize DDS configuration when necessary |
| Insufficient storage space | Insufficient system memory or disk space | Clean up unnecessary cache data to ensure sufficient space for storing logs, configuration files, etc |

# Example Command Set

## ROS2 Environment Configuration Commands

### Setting the ROS2 Communication Range

```
# Set the ROS2 communication range to the local host.
echo "export ROS_LOCALHOST_ONLY=1" >> ~/.bashrc

# Set the ROS2 communication range to the local area network.
echo "export ROS_LOCALHOST_ONLY=0" >> ~/.bashrc
```

### Configure ROS_DOMAIN_ID

```
# Check the current ROS_DOMAIN_ID (to confirm the communication domain isolation status).
printenv ROS_DOMAIN_ID

# Temporarily set ROS_DOMAIN_ID to 5 (effective only in the current terminal, used for communication domain isolation).
export ROS_DOMAIN_ID=5

# Permanently set ROS_DOMAIN_ID (write to the bash configuration file; the change will take effect after restarting the terminal).
echo "export ROS_DOMAIN_ID=5" >> ~/.bashrc
```

## System Startup Command

### Start the Motion Control System

```
# Start RB-A630-2 motion control with the default configuration
# This starts all required nodes, including robot controllers, motion planners, and visualization tools
ros2 launch raybot_bringup robot.launch.py model:=rb_a630_2
```

```
# Start RB-A630-2
ros2 launch raybot_bringup robot.launch.py model:=rb_a630_2
```

```
# Override JTC trajectory error tolerances at startup
ros2 launch raybot_bringup robot.launch.py model:=rb_a630_2 \
  jtc_trajectory_tolerance:=0.8 \
  jtc_goal_tolerance:=0.25 \
  jtc_goal_time:=3.0 \
  jtc_stopped_velocity_tolerance:=0.2
```

## CLI Controller Commands

### Send Movement Commands

Run one minimal command first to confirm that `group`, `pose`, `planner_id`, and the low-level control chain are all working before trying more combinations.

**Minimal working example:**

```
# Execute one PTP end-effector pose motion with the arm
ros2 service call /cli_controller/execute_command robot_controller/srv/ExecuteCommand "{command:
'group=arm type=pose pose=0.45,-0.20,0.4,0,1.57,0 frame=base_link
pipeline=pilz_industrial_motion_planner planner_id=PTP vel=0.2 acc=0.2 exec=1', wait_for_result: true,
timeout: 0}"
```

**More examples:**

```
# Send a point-to-point motion command using OMPL
# Move the arm to the specified pose with point-to-point motion using OMPL
ros2 service call /cli_controller/execute_command robot_controller/srv/ExecuteCommand "{command:
'group=arm type=pose pose=0.6,-0.18,0.80,0,-1.57,3.14 frame=base_link vel=0.2 acc=0.2 pipeline=ompl
planner_id=RRTConnectkConfigDefault exec=1', wait_for_result: true, timeout: 0}"

# Send a linear motion command using the Pilz industrial motion planner
# Move the arm along a straight line to the specified pose in LIN mode
ros2 service call /cli_controller/execute_command robot_controller/srv/ExecuteCommand "{command:
'group=arm type=pose pose=0.45,-0.30,0.4,0,1.57,0 frame=base_link
pipeline=pilz_industrial_motion_planner planner_id=LIN vel=0.2 acc=0.2 exec=1', wait_for_result: true,
timeout: 0}"

# Send a joint-space motion command using OMPL
# Move arm joints 4, 5, and 6 to the specified angles
ros2 service call /cli_controller/execute_command robot_controller/srv/ExecuteCommand "{command:
'group=arm type=joints joints=arm_6_joint:0.1,arm_5_joint:0.1,arm_4_joint:0.1 pipeline=ompl
planner_id=RRTConnectkConfigDefault exec=1', wait_for_result: true, timeout: 0}"
```

## Action Recorder Commands

### Edit Action Program

For first use, verify in this order: add one step -> check with `list` -> execute with `play`. Do not stack many steps at the beginning.

#### Minimal working example:

```
# Add one PTP end-effector motion for the arm
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'add type=pose group=arm frame=base_link ee=arm_6 pose=0.45,-0.20,0.4,0,1.57,0 planner=pilz_ptp vel=0.2 acc=0.2 exec=1 wait=1.0 note=Arm pre-grasp pose'}"

# View the current motion program
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'list'}"

# Execute the motion program
ros2 action send_goal /action_recorder/command robot_controller/action/ProgramCommand "{command: 'play exec=1 replan=1'}"
```

#### Common edit and execution commands:

```
# Add a Pilz PTP end-effector motion for the arm, followed by a 1-second wait
# Use the Pilz planner in PTP mode to move the arm to the specified end-effector pose
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'add type=pose group=arm frame=base_link ee=arm_6 pose=0.45,-0.20,0.4,0,1.57,0 planner=pilz_ptp vel=0.2 acc=0.2 exec=1 wait=1.0 note=Arm pre-grasp pose2'}"

# Add a Pilz LIN motion command for the arm
# Use the Pilz planner in LIN mode to move the arm along a straight line to the target pose
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'add type=pose group=arm frame=base_link ee=arm_6 pose=0.55,-0.20,0.4,0,1.57,0 planner=pilz_lin vel=0.2 acc=0.2 exec=1 wait=1.0 note=Arm linear path'}"

# Add an OMPL return-to-home command for the arm
# Use OMPL to return the arm to home, the predefined safe pose
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'add type=named group=arm frame=base_link target=home planner=ompl vel=0.3 acc=0.3 exec=1 note=Arm return home'}"

# Add an OMPL point-to-point end-effector motion for the arm
# Use OMPL to move the arm to the specified end-effector pose
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'add type=pose group=arm frame=base_link ee=arm_6 pose=0.50,-0.10,0.35,0,1.57,0 planner=ompl vel=0.25 acc=0.25 exec=1 note=Arm OMPL pose'}"

# Add a joint-space motion command for the arm
# Control individual joint angles directly for precise joint-space motion
```

```
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'add type=joints group=arm joints=0,0.4,-0.2,0,0.6,0 frame=base_link planner=ompl vel=0.25 acc=0.25 exec=1 note=Arm joint motion'}"

# Insert a 2.5-second wait step
# Add a wait to the sequence for external signals or other systems to become ready
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'wait wait_duration=2.5 note=Waiting for external signal'}"

# List all motion steps
# View all steps in the current motion program and their order
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'list'}"

# Save the motion sequence to a file
# Save the current motion program to the specified file for later loading and execution
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'save file=demo_program4'}"

# Load a program
# Load a saved motion program from the specified file
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'load file=demo_program'}"

# Delete a motion step
# Delete the program step at index 3 (indices start at 0)
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'del index=3'}"

# Delete multiple steps by index
# Delete steps at indices 2, 4, and 5 while retaining the other steps
ros2 service call /action_recorder/delete_steps robot_controller/srv/DeleteProgramSteps "{indices: [2,4,5], ignore_missing: false, report_remaining: true}"

# Execute the sequence with replanning at each step
# Execute all program steps, replanning each path to account for environmental changes
ros2 action send_goal /action_recorder/command robot_controller/action/ProgramCommand "{command: 'play exec=1 replan=1 steps=[]'}"

# Execute the sequence without replanning
# Execute all program steps using preplanned paths; this is faster but does not adapt to environmental changes
ros2 action send_goal /action_recorder/command robot_controller/action/ProgramCommand "{command: 'play exec=1 replan=0'}"

# Execute the sequence repeatedly
# repeat_count is a positive integer specifying how many times to run the complete selection
ros2 action send_goal /action_recorder/command robot_controller/action/ProgramCommand "{command: 'play exec=1 replan=0 repeat_count=3'}"

# Clear all motion steps stored in memory
```

```
# Remove all steps from the current motion program
ros2 service call /action_recorder/clear_program robot_controller/srv/ClearProgram {}

# Plan selected steps without executing them
# Plan paths for steps at indices 1, 3, and 5 without executing them
# Use this to validate planning feasibility and safety
ros2 service call /action_recorder/plan_program robot_controller/srv/PlanProgram "
{use_current_state_start: true, steps: [1,3,5]}"

# Execute selected steps only
# Execute only steps at indices 1, 3, and 5, skipping the others
ros2 action send_goal /action_recorder/command robot_controller/action/ProgramCommand "{command:
'play exec=1 replan=1 steps=[1,3,5]'}"

# Execute selected steps and repeat them
ros2 action send_goal /action_recorder/command robot_controller/action/ProgramCommand "{command:
'play exec=1 replan=1 steps=[1,3,5] repeat_count=2'}"
```

## Optional Extension Commands

The standard configuration provides no object grasping commands. Do not directly reuse gripper or vision grasping examples for other models.

## Motion Stop Command

### Stop the Specified Action Group's Movement

```
# Stop all motion in the arm group (emergency stop command)
ros2 topic pub --once /arm_motion_controller/stop_request std_msgs/msg/String "{data: 'arm'}"

# Stop all motion in the arm group (emergency stop command)
ros2 topic pub --once /arm_motion_controller/stop_request std_msgs/msg/String "{data: 'arm'}"
```

### Shut Down the Control System

```
# Stop the desktop control software using the normal shutdown procedure
source /home/raybot/raybot_core_ws/install/setup.bash
ros2 service call /raybot_robot/bringup_shutdown std_srvs/srv/Trigger {}
```