Raybot Robot Motion Control Software Instructions

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
  - 5.6 Gripper PP Mode and Command-Line Control
- 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 dual_arms Pose Command Rules
  - 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
- 11. Object Grasp Node (Item grasping node)
  - 11.1 Module Functions
  - 11.2 Configuration Parameters
  - 11.3 Commands and Services
  - 11.4 Complete Testing Process
    - 11.4.1 Test Environment Setup
    - 11.4.2 Test Commands
  - 11.5 Implementation Details
  - 11.6 Interaction Process
- 12. Troubleshooting
  - 12.1 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 Object Grasp Node Commands

- 13.6 Motion Stop Command

Document Overview

This document provides a detailed description of the architecture, module functions, interaction flow, parameter configuration, and usage of the Raybot robot motion control software. Based on ROS2 and a motion planning framework, this software enables robot motion control, action programming, and object grasping.

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

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

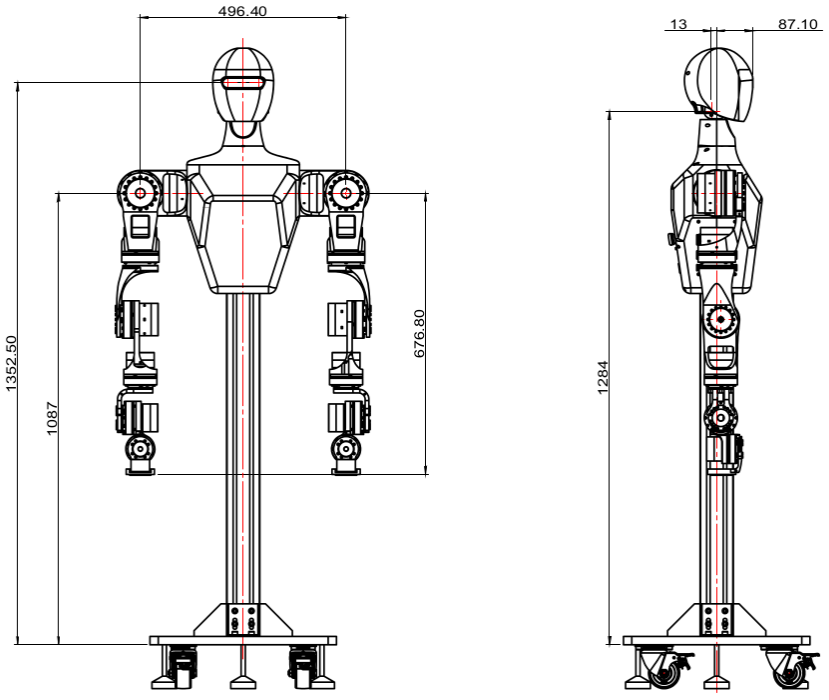
Term	Full Name	Explanation
		ROS2 is used for publishing and subscribing to messages
Service	Service	The synchronous communication mechanism in ROS2 uses a request-response pattern
Action	Action	ROS2's asynchronous communication mechanism supports long-running tasks and cancellation functionality

Robot Hardware Introduction

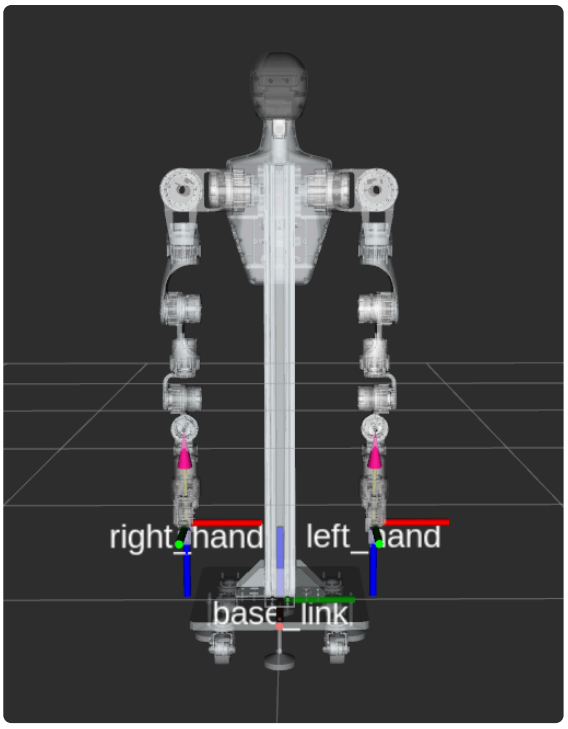
Introduction to Robot Hardware

This section uses the RB-A850-10 dual-arm robot as an example.

The harmonic dual-arm robot consists of two robotic arms, each with 7 degrees of freedom, and an RGB and depth camera mounted on a 2-degree-of-freedom gimbal. The main relative dimensions are shown in the figure below:



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

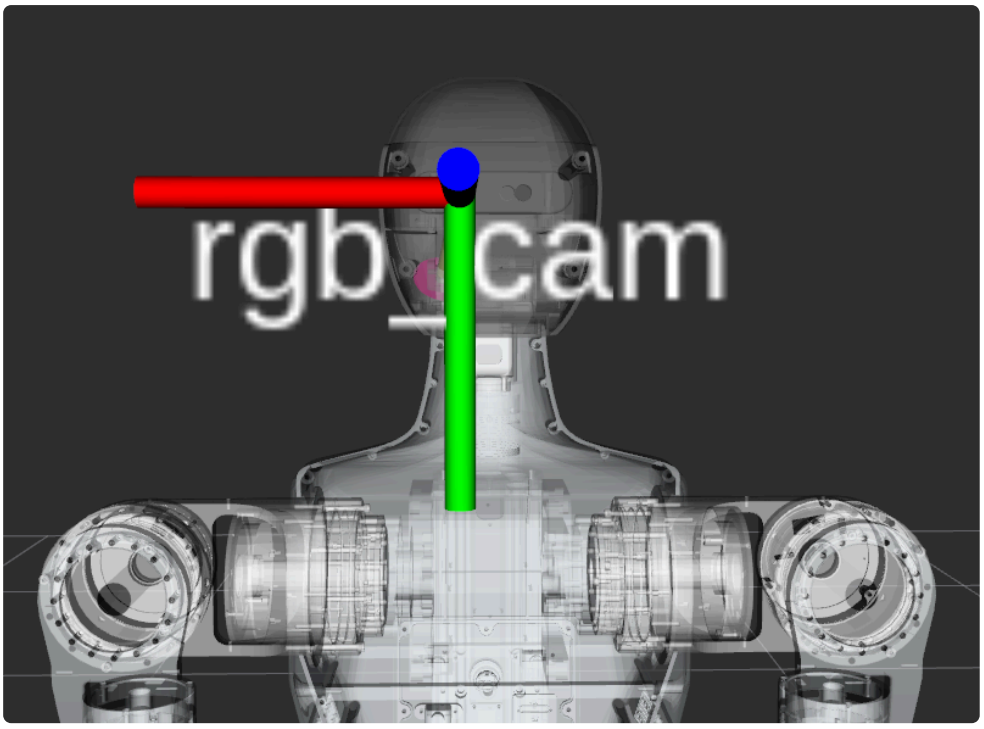
Robot end-effector pose transformation refers to the process of transforming the tool coordinate system from one position and orientation to another, which is mainly achieved through the following steps:

1. **Kinematic forward solution (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. **Kinematic inverse solution (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.

Schematic diagram of the robot vision camera coordinate system:



In the schematic diagram of the robot vision camera coordinate system, it is conventionally agreed that red, green, and blue represent the x, y, and z axes of the camera coordinate system, respectively. The z-axis points forward along the camera's optical axis (from the camera towards the observed scene), the x-axis points horizontally to the right, and the y-axis points vertically downwards. Through camera intrinsic and extrinsic parameter calibration, 3D points in space can be transformed from the robot or world coordinate system to the camera coordinate system and then projected onto the image plane, which is used for tasks such as object detection, pose estimation, and path planning.

In this system, robot vision is composed of both an **RGB camera** and a **Depth camera**, with their functions roughly divided as follows:

- **RGB camera (color camera)**
  - Acquire a 2D color image of the scene (R/G/B three channels) for use:

        - Object detection and recognition (such as identifying workpieces, obstacles, people, landmarks, etc.)
        - Texture and color feature extraction (used for segmenting different regions or objects)
        - Assisted visualization and debugging (allowing humans to visually observe the current scene)
    - The output is a standard 2D image and does not directly contain depth information.
- **Depth Camera**
    - This process provides distance information for each pixel in the image, specifically the "distance from the camera to the object surface," forming a depth map or point cloud data.
    - Mainly used for:
        - Reconstructing the 3D geometric structure of the scene (generating point clouds or grid maps)
        - Calculate the three-dimensional position and relative orientation of the target object.
        - Supports tasks such as obstacle avoidance and grasp point planning.
    - Depth information is provided in the form of a "grayscale depth map."

By registering and fusing **RGB images** with **depth data** (meaning each pixel location has both color and depth information), the system can reconstruct colored 3D environmental information in the robot coordinate system/world coordinate system, thereby achieving **precise localization** of target objects and **accurate guidance** of robotic arm movements.

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_a850_10
```

The unified startup entry is provided by raybot_bringup . Use model:=rb_a850_10 for RB-A850-10. If the on-site robot is RB-A575-3, use:

```
ros2 launch raybot_bringup robot.launch.py model:=rb_a575_3
```

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_a850_10 \
  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 harmonic dual-arm motion control software. Logically, it consists of the core execution module (`arm_motion_controller`) and three upper-layer driver modules for different usage scenarios. These modules share unified interfaces through ROS 2 topics, services, and actions, and together form 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:

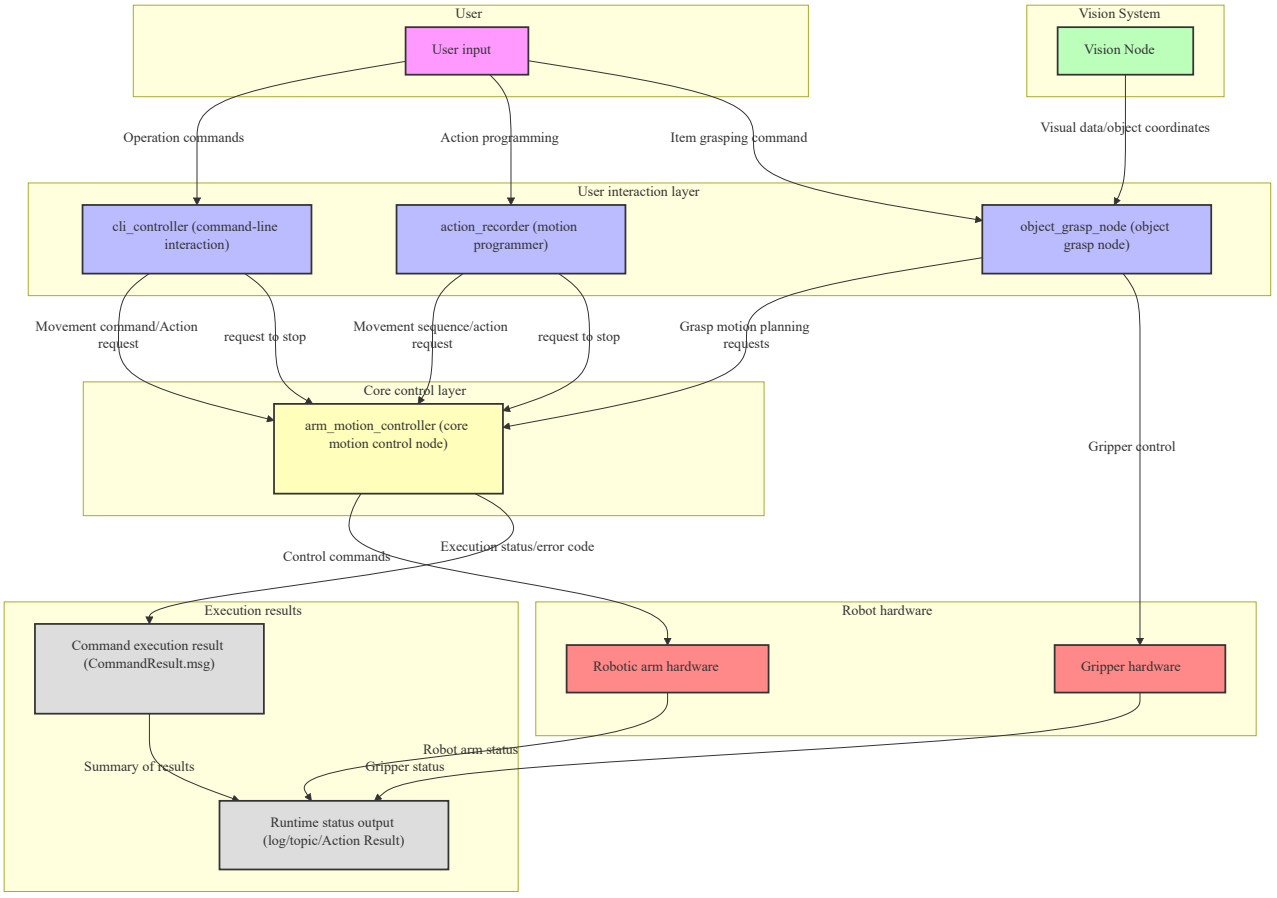
1. **cli_controller**: A command-line interaction module that receives robotic arm motion commands through topics and services, parses them, and forwards them to the core execution node. It is suitable for debugging and manual operation.
2. **action_recorder**: A motion programming and playback module 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**: An automatic grasp-and-place module that plans grasp paths from vision coordinates and controls the gripper through the full grasp-place sequence. It supports multi-stage grasp strategies and is suitable for automated grasping scenarios.

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)
object_grasp_node	Automatic grasp-and-place node that plans grasp paths from vision coordinates and	services /object_grasp_node/pick (robot_controller/srv/PerformGras

Module Name	Function	External Interface
	controls the gripper through multi-stage grasping logic	p.srv), /object_grasp_node/place (robot_controller/srv/PerformPlace.srv)
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)

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 ~/raybot_core_ws/install/setup.bash
ros2 launch raybot_bringup robot.launch.py model:=rb_a850_10
```

Notes:

- This launch file is located in the `raybot_bringup` package
- Select the robot model with `model:=rb_a850_10` or `model:=rb_a575_3`
- 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 the system starts, you can use the global mode switching services to switch the control mode of the left arm, right arm, both hands, and head together. The current system supports `moveit`, `position`, `velocity`, and `effort` as global modes. A query service is also available to read the current machine-wide mode state. Global services apply to all enabled planning groups; for PP-enabled grippers, switching globally to `moveit` temporarily activates the gripper JTC and must be followed by the gripper-specific PP service 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.

Gripper PP Mode and Command-Line Control

PP (Profile Position) is the dedicated single-axis position mode for grippers. After the target position, velocity, and acceleration are set, the motor drive plans the trajectory to the target. A gripper cannot have its JTC and PP controllers active at the same time. `rb_a850_10` and `rb_a575_3` use PP to control their grippers by default:

Planning group	Joint	PP controller	Profile Position Action
l_hand	left_hand_virtual_joint	l_hand_pp_controller	/l_hand_pp_controller/profile_position
r_hand	right_hand_virtual_joint	r_hand_pp_controller	/r_hand_pp_controller/profile_position

Robot and Gripper Mode Preparation

After starting the motion control system, use service commands to keep the arms in MoveIt/JTC and switch
the left and right grippers to PP. Run the following commands in a terminal where the workspace environment
has been sourced:

```
source /home/raybot/raybot_core_ws/install/setup.bash

# Switch all robot planning groups to MoveIt/JTC
ros2 service call /raybot_robot/switch_to_moveit std_srvs/srv/Trigger "{}"

# Switch both grippers to PP
ros2 service call /raybot_robot/switch_l_hand_to_pp std_srvs/srv/Trigger "{}"
ros2 service call /raybot_robot/switch_r_hand_to_pp std_srvs/srv/Trigger "{}"

# Query the current mode of both grippers
ros2 service call /raybot_robot/get_l_hand_control_mode std_srvs/srv/Trigger "{}"
ros2 service call /raybot_robot/get_r_hand_control_mode std_srvs/srv/Trigger "{}"
```

PP-enabled grippers have their PP controllers active by default after the standard bringup sequence completes.
The commands above are mainly used to restore the control mode explicitly or diagnose a mode error.
`switch_to_moveit` depends on `move_group`; if the service returns a failure, first confirm that MoveIt is
running, then query the left and right gripper modes individually.

Gripper Mode Services

All of the following services use `std_srvs/srv/Trigger`:

Operation	Left gripper	Right gripper
Query current mode	/raybot_robot/get_l_hand_control_mode	/raybot_robot/get_r_hand_control_mode
Switch to PP	/raybot_robot/switch_l_hand_to_pp	/raybot_robot/switch_r_hand_to_pp
Switch to MoveIt/JTC	/raybot_robot/switch_l_hand_to_moveit	/raybot_robot/switch_r_hand_to_moveit

PP-enabled grippers start with their PP controller active. Switch to MoveIt/JTC only for debugging or
legacy JTC applications, then explicitly switch back to PP.

Profile Position Action

The Action type is `raybot_pp_controller/action/ProfilePosition`. Goals use SI units, and runtime
parameters are carried by each goal rather than stored in motor YAML:

Field	Unit	Description
position	rad	Absolute target position; required
profile_velocity	rad/s	Profile velocity
profile_acceleration	rad/s²	Profile acceleration
profile_deceleration	rad/s²	Profile deceleration
quick_stop_deceleration	rad/s²	Quick-stop deceleration
timeout	s	Total timeout for the goal

Controller PP defaults are provided per model by `robots.yaml`. The current configuration
uses the same defaults for both grippers:

Model	Velocity (rad/s)	Acceleration (rad/s²)	Deceleration (rad/s²)	Quick-stop deceleration (rad/s²)	Timeout	Torque limit
rb_a850_10	15.0	60.0	60.0	60.0	10 s	30%
rb_a575_3	5.0	10.0	10.0	10.0	10 s	30%
rb_a575_3_tmp	5.0	10.0	10.0	10.0	10 s	30%

Each parameter can be specified explicitly when sending an Action goal from the terminal. A field set to
zero or left unspecified uses the current model's controller default.

`max_torque` is not a Profile Position Action field. It must be written independently to object `0x6072` through the EtherCAT command service. The default in `robots.yaml` is `30%`, which maps to the raw value `300` on the 0-1000 scale. For example, the left gripper alias is `1116`:

```
ros2 service call /raybot_robot/ethercat/command \
  ethercat_control/srv/EthercatCommand \
  "{command: '1116 max_torque 300'}"
```

Gripper Positions and Test Commands

The current URDFs for the three PP-enabled robots limit gripper joints to `0` to `0.45 rad`: `0 rad` is closed and `0.45 rad` is near the open limit. Use low speed, small displacements, and keep the hardware emergency stop ready during initial testing.

```
# Ensure the left gripper is in PP mode
ros2 service call /raybot_robot/switch_l_hand_to_pp std_srvs/srv/Trigger "{}"

# Close the left gripper to 0 rad
ros2 action send_goal /l_hand_pp_controller/profile_position \
  raybot_pp_controller/action/ProfilePosition \
  "{position: 0.0, profile_velocity: 15.0, profile_acceleration: 60.0, profile_deceleration: 60.0,
quick_stop_deceleration: 60.0, timeout: {sec: 10, nanosec: 0}}" \
  --feedback

# Open the left gripper to 0.45 rad
ros2 action send_goal /l_hand_pp_controller/profile_position \
  raybot_pp_controller/action/ProfilePosition \
  "{position: 0.45, profile_velocity: 15.0, profile_acceleration: 60.0, profile_deceleration: 60.0,
quick_stop_deceleration: 60.0, timeout: {sec: 10, nanosec: 0}}" \
  --feedback
```

An Action completes with `REACHED` after the target is reached. Use `--feedback` to monitor execution from the command line; consider the gripper at its target only after the Action Result reports success. A new PP goal submitted while another is running directly preempts the old goal; switching to another motion controller executes the new controller command. Cancel, timeout, following

error, or drive fault ends the
current goal and performs a CSV zero-speed stop.

Query the Active Model

After bringup starts, use this service to query the loaded variant name, URDF/SRDF paths, and enabled planning groups:

```
ros2 service call /raybot_robot/get_active_model ethercat_control/srv/GetActiveRobotModel "{}"
```

This service uses `ethercat_control/srv/GetActiveRobotModel`, not `std_srvs/srv/Trigger`. See the Modular Config and Auto-Generation Guide for the full response field reference.

Planning Group and Predefined Poses

Planning Group Introduction

Planning Group refers to a set of joints used to specify the moving parts of a robot. The current system supports the following planning groups:

Planning Group Name	Corresponding Part	Instruction
l_arm	left arm	Controls the seven joints of the left arm
r_arm	right arm	Controls the seven joints of the right arm
dual_arms	dual arms	Controls both left and right arms together for synchronized or coordinated motions
l_hand	left gripper	Controls the single left gripper joint for open/close motion
r_hand	right gripper	Controls the single right gripper joint for open/close motion
head	head	Controls the two head joints for camera gimbal pitch and yaw

Planning Group Joint Composition

If you only need the daily operation entry points, focus on these five planning group names first: `l_arm`, `r_arm`, `l_hand`, `r_hand`, and `head`. The full joint composition is shown below:

Planning Group	Joint Qty	Joint Names	Motor Alias
l_arm	7	left_arm_1_joint left_arm_2_joint left_arm_3_joint left_arm_4_joint left_arm_5_joint left_arm_6_joint left_arm_7_joint	1102 1103 1104 1105 1106 1107 1108
r_arm	7	right_arm_1_joint right_arm_2_joint right_arm_3_joint right_arm_4_joint right_arm_5_joint right_arm_6_joint right_arm_7_joint	1109 1110 1111 1112 1113 1114 1115
l_hand	1	left_hand_virtual_joint	1116
r_hand	1	right_hand_virtual_joint	1117
head	2	neck_joint head_joint	1100 1101

Predefined Pose

Predefined Pose, also known as a named coordinate, refers to a pre-defined robot posture that can be called using a simple name, without the need to specify complex 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.

Currently defined named coordinates in the system:

- Both the left and right arms support the `home` posture, which is the robot's zero position and its initial safe posture.

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 is responsible for generating an executable motion trajectory (joint space trajectory or Cartesian trajectory) based on the starting and target states. RAYBOT harmonic dual-arm motion control software supports multiple motion planners, allowing users to select the appropriate planner based on 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.
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	Automatically expands to the corresponding Pilz PTP / LIN 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
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 harmonic dual-arm motion control software, responsible for the motion planning and execution of the robotic arms. It encapsulates the motion planning interface and provides a unified motion control service, ensuring that multiple upper-level modules can safely share the robotic arm resources.

Supported types of sports:

- **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.
- **Action stopped**: Immediately stop the current movement.

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 workflow**:

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

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 types of sports.:
  - Predefined posture movements(type=named)
  - Joint space movement(type=joints)
  - Cartesian point motion(type=pose)

    - 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=l_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=l_arm type=named target=home exec=1
Joint space movement	type=joints	Directly control joint angles.	group=l_arm type=joints joints=left_arm_1_joint:0.5,left_arm_2_joint:0.5 exec=1
Cartesian point motion	type=pose	Move to the specified Cartesian coordinates	group=l_arm type=pose pose=0.4,0.2,0.3,0,1.57,0 frame=base_link exec=1
Dual-arm multi-end-effector pose	type=multi_pose	Specifies target poses for left and right hands together. Use with group=dual_arms	group=dual_arms type=multi_pose links=left_hand,right_hand poses=0.012,0.248,0.255,-0.589,1.571,0.983;0.012,-0.248,0.255,-0.352,1.570,-1.924 frame=base_link planner=ptp exec=1
Stop motion	type=stop	Stop the current movement.	group=l_arm type=stop

3. Common command parameters

Parameter name	Type	Instruction
group	string	Robot arm group name(l_arm/r_arm/head)
type	string	Motion type(named/joints/pose/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)
links	string	Used by `type=multi_pose`; multiple end-effector links are comma-separated, for example `left_hand,right_hand`
poses	string	Used by `type=multi_pose`; multiple 6-DOF poses are semicolon-separated and correspond one-to-one with `links`
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`, `ompl`
planner_id	string	Planner ID, such as Pilz `PTP` / `LIN` 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.

DUAL_ARMS POSE COMMAND RULES

`group=dual_arms` plans both arms together, so it does not use the single-arm `type=pose pose=...` syntax. Dual-arm end-effector targets must use `type=multi_pose`:

```
group=dual_arms type=multi_pose links=left_hand,right_hand poses=<left-hand-6DOF>;<right-hand-6DOF> frame=base_link planner=ptp exec=1
```

Each pose uses this format:

```
x,y,z,roll,pitch,yaw
```

The number of `links` and `poses` entries must match. The recommended dual-arm end-effector links are `left_hand,right_hand`. For first validation, set `exec=0` before executing real robot motion.

Dual-arm PTP example:

```
ros2 service call /cli_controller/execute_command robot_controller/srv/ExecuteCommand "{command: 'group=dual_arms type=multi_pose links=left_hand,right_hand poses=0.012,0.248,0.255,-0.589,1.571,0.983;0.012,-0.248,0.255,-0.352,1.570,-1.924 frame=base_link planner=ptp vel=0.1 acc=0.1 exec=1', wait_for_result: false, timeout: 1.0}"
```

Dual-arm LIN example:

```
ros2 service call /cli_controller/execute_command robot_controller/srv/ExecuteCommand "{command: 'group=dual_arms type=multi_pose links=left_hand,right_hand poses=0.012,0.248,0.255,0,1.5708,0;0.012,-0.248,0.255,0,1.5708,0 frame=base_link planner=lin lin_position_only=1 vel=0.1 acc=0.1 exec=0', wait_for_result: true, timeout: 30.0}"
```

LIN note: `lin_position_only=1` changes only the target positions and keeps the current orientation. LIN is more sensitive to the start state, orientation, joint limits, and singularities. Validate with `exec=0` first, then use `exec=1` only after planning succeeds.

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 .

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 gripper PP steps for `l_hand` / `r_hand` ; gripper steps call the Profile Position Action directly instead of MoveIt planning

Configuration Parameters

Parameter Name	Type	Default value	Explanation
group	string	"l_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=l_arm
```

Supported types of sports include:

- type=named : Predefined posture movements
- type=joints : Joint space motion
- type=pose : Cartesian point motion

When group=l_hand or group=r_hand is used with type=joints , the step uses gripper PP control. The step may set pp_vel , pp_acc , pp_dec , pp_qs_dec , and pp_timeout ; omitted values use the PP controller defaults. A PP step is successful only after the REACHED result is received. Gripper torque limits are not action-step parameters; configure object 0x6072 through the independent EtherCAT command.

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

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

Object Grasp Node (Item grasping node)

Module Functions

The Object Grasp Node is an advanced module used for automatic grasping and placement of objects by a robotic arm. Based on the coordinate information provided by the vision node, it automatically plans the grasping path and placement location, and controls the gripper to complete the entire grasping and placement process.

Core functions:

- Automatic grasping based on visual coordinates
- Multi-stage grasping strategy (pre-grasp, approach, grasp, lift, retreat)
- Flexible placement position configuration
- Supports multiple grasping directions and retry mechanisms
- Gripper control and point cloud collaboration
- Supports custom grasping/placement configurations

Grasping Process

1. **Pre-grasping phase:** Move to the pre-grasping position.
2. **Approach phase:** Approach the target object.
3. **Grasping phase:** Close the gripper to grasp the object.

4. **Lifting phase:** Lift the object to a safe height.

Placement Process

1. **Approach Phase:** Approaching the placement location
2. **Descent Phase:** Lowering the object to the placement surface
3. **Release Phase:** Opening the gripper to release the object
4. **Lifting Phase:** Lifting to a safe height
5. **Retreat Phase:** Returning to a safe position

Precautions for Use

1. **Reachability Requirements:** The object and its target destination must be within the robot arm's reach; otherwise, the planning will fail.
2. **LIN Movement Limitations:** Using a large number of linear (LIN) movements may increase the planning failure rate. It is recommended to slightly adjust the object's position or modify the planner parameters.
3. **Hardware Dependency:** This module depends on the gripper hardware. If the gripper is not installed or there are communication issues with the gripper, the program will not run correctly.

Configuration Parameters

Parameter Name	Type	Default Value	Explanation	Usage suggestions
group_name	string	"l_arm"	Robot arm group name	Select either the left arm (l_arm) or the right arm (r_arm) as needed.
ee_link	string	"l_arm_ee"	End effector link name	Adjust according to the actual end effector model.
target_frame	string	"object_tf"	Target object TF name	The object coordinate frame is provided by the visual system.
reference_frame	string	"base_link"	Reference coordinate system	Typically, `base_link` is used as the fixed reference frame.
planner_mode	string	"cartesian"	Planner Pattern	Choose the appropriate planner mode based on your movement requirements.
approach_speed	double	0.1	Approach speed ratio	During the debugging phase, it is recommended to set the value to 0.1-0.3; the actual value can be adjusted according to requirements during operation.
pre_grasp_offset	double	0.12	Distance from pre-grasp point to grasping point (meters)	Adjust according to object size to ensure safe approach.
approach_directions	string	"1,0,0"	Approach direction (multiple directions separated by semicolons)	Multiple directions are separated by semicolons, such as:"1,0,0;0,1,0"

Parameter Name	Type	Default Value	Explanation	Usage suggestions
direction_variation_deg	double	30.0	Direction change angle (degrees)	Increasing the angle can increase the probability of successful grasping, but it will also increase the planning time.
direction_variation_samples	int	8	Number of direction change samples	The more samples, the higher the probability of successful grasping, but the longer the planning time.
try_alternate_direction	bool	true	Should alternative approaches be attempted?	Recommended to enable this option to improve the success rate of data retrieval.
verify_before_execute	bool	true	Whether to verify the planning result before execution	It is recommended to always enable this option to ensure the safety of the movement
auto_clear_octomap	bool	true	Should the obstacle map be regenerated?	It is recommended to enable this option to ensure the environment information is up-to-date.
pause_point_cloud	bool	true	Should point cloud data acquisition be paused?	It is recommended to enable this to prevent environmental changes during movement from affecting planning.
gripper_open_command	string	null	Gripper open CAN command	Based on actual gripper hardware

Parameter Name	Type	Default Value	Explanation	Usage suggestions
				configuration
gripper_close_command	string	null	Gripper close CAN command	Based on actual gripper hardware configuration
auto_place_after_pick	int	0	Automatic placement (-1/0/1: disabled/default/enabled)	Automatic scenarios should be set to 1, manual control scenarios should be set to -1.
auto_place_retreat_home	bool	false	Whether to return to the Home position after placement	For continuous operation scenarios, it is recommended to set this to false to save time; for idle states, it can be set to true.
retreat_after_pick	bool	true	Should the robot retract after grasping?	It is recommended to always enable this option to avoid collisions with surrounding objects after grasping.
retreat_after_place	bool	true	Should it retract after placement?	It is recommended to always keep this function enabled to avoid collision with the workpiece or fixture after placement.

Commands and Services

Service Interface

Service Name	Type	Function Description
/object_grasp_node/pick	robot_controller::srv::PerformGrasp	Object grasping service

PerformGrasp Service

Function: Grasp an object identified by the specified TF frame.

Request format:

```
group_name: "r_arm"        # The robotic arm planning group determines which arm to use for grasping.
ee_link: "right_hand"      # End effector link name
target_frame: "object_1"     # The TF name of the object, published by the visual system.
retreat_after_pick: 0       # After grasping, does it return to a predefined posture? 0 means it does not return.
retreat_target: "home"       # The predefined posture returned after grasping.
auto_clear_octomap: 1        # Clear and rebuild obstacle point clouds; -1/0/1 means default/off/on
pre_grasp_offset: 0.1       # Distance from the pre-grasp position to the final grasp position (m)
lift_distance: 0.1          # Upward lift distance after grasping the object (m)
```

Response Format:

```
success: true              # Whether the data retrieval was successful; true indicates success, false indicates failure.
message: "Object grasped successfully"  # Execute the message and describe the retrieval results.
```

Complete Testing Process

Test Environment Setup

It is recommended to use cardboard boxes to build a simple test setup for validating grasp-and-place behavior. Place the test object, such as a Coca-Cola can or coaster, on a box about 80 cm high, and place that box around 30 to 40 cm in front of the robot's right arm. Once the grasp flow starts, the robot should automatically grasp the can and place it on the coaster.

Automatic grasping depends on visual detection, so you must use the camera pan-tilt to move the target object into the camera field of view. For the vision-side details, refer to the vision module software documentation.

Test Commands

Pick-and-place service call example:

```
ros2 service call /object_grasp_node/pick robot_controller/srv/PerformGrasp "{
  group_name: 'r_arm',                  # The robotic arm planning group determines which arm to use for grasping.
  ee_link: 'right_hand',                # End effector link name
  target_frame: 'cocacola_target',        # The TF name of the object to be grasped is published by the vision system.
  retreat_after_pick: 0,                # Do not return to a named pose after grasp flow.
  retreat_target: 'home',               # Named pose (unused because retreat_after_pick=0)
  gripper_group_name: 'r_hand',           # Gripper planning group (example)
```

```
  gripper_joint_name: 'right_hand_virtual_joint',  # Gripper joint name (example)
  gripper_open_angle: '0.45',               # Open angle (rad, string)
  gripper_close_angle: '0.0',               # Close angle (rad, string)
  approach_directions: '1,1,0',             # The approach direction vector list specifies the direction in which the robotic arm approaches the object.
  approach_speed: 0.1,                      # Speed scaling factor for all movement phases
  approach_acc: 0.1,                        # Acceleration scaling factor for all motion phases
  pre_grasp_offset: 0.12,                   # The distance from the pre-grasp position to the grasping position.
  lift_distance: 0.05,                      # The distance the object is lifted after being grasped is also used as the distance it is lowered during placement.
  action_pause: 1.0,                        # Pause time between each exercise phase (seconds)
  auto_clear_octomap: 1,                    # Clear Octomap before planning.
  pause_point_cloud: 1,                     # Pause incoming obstacles during motion.
  octomap_wait_time: 4.0,                   # Waiting time after clearing the octree (seconds)
  pre_grasp_pipeline_id: 'ompl',            # The planning pipeline used during the pre-fetching phase.
  pre_grasp_planner_id: 'RRTConnectkConfigDefault',  # The planner ID used during the pre-fetching phase.
  approach_pipeline_id: 'pilz_industrial_motion_planner',  # Planning pipeline used in the approach phase.
  approach_planner_id: 'LIN',               # The planner ID used in the approach phase; LIN indicates linear motion.
  lift_pipeline_id: 'pilz_industrial_motion_planner',      # Planning pipeline used during the enhancement phase.
  lift_planner_id: 'LIN',                   # Planner ID used during the enhancement phase
  auto_place: 1,                            # Whether to enable the automatic placement process (1 indicates enabled).
  place_target_frame: 'coaster_target',       # The target TF name for placing the object.
  place_translate_pipeline_id: 'pilz_industrial_motion_planner',  # Place the planning pipeline used during the translation phase.
  place_translate_planner_id: 'LIN',        # The planner ID used during the placement and translation phase.
  place_drop_pipeline_id: 'pilz_industrial_motion_planner',       # Place the planned pipelines to be used during the descent phase.
  place_drop_planner_id: 'LIN',             # Place the planner ID used during the descent phase.
  verify_before_execute: true,              # Whether to validate the planning results of all stages before execution; true indicates validation.
  auto_place_retreat: 1                     # Does it automatically retract after being placed? 1 indicates retraction.
}"
```

Implementation Details

- **Implementation file**: src/object_grasp_node.cpp
- **Core functions**: Automatic object grasping and placement based on visual coordinates.
- **Key functions**:
  - handleGraspService() : Processing data scraping service requests
  - handlePlaceService() : Processing placement service requests

- `planAndExecuteGrasp()` : Plan and execute the data scraping process.
- `planAndExecutePlace()` : Plan and execute the placement process.
- `generateGraspPoses()` : Generate multiple possible grasping poses.

- **Features:** Supports multi-stage grasping strategies, and automatically handles gripper control and environmental sensing.

Interaction Process

The object grasping node, as part of the automated pick-and-place module, operates according to the following interaction process:

1. **Input**:

- Receive pick-and-place requests through the service interface
- Receive object coordinates and point cloud data provided by the vision node

2. **Process**:

- Parsing service request parameters
- Planning multi-stage grasping/placement paths
- Calling the Arm Motion Controller to execute motion commands
- Controlling the gripper to perform grasping/releasing actions
- Processing environmental point cloud data to ensure safe operation

3. **Output**:

- Return execution results via service response
- Publish gripping/placement status information

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

- Located in the user interaction layer, it receives grasping commands from the user.
- It interacts with the vision system to obtain object coordinates.
- It calls the Arm Motion Controller in the core control layer to execute the motion.

Troubleshooting

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	Controller not active, interface mismatch	Confirm the controller status is active, check for consistent

Error Symptom	Possible Cause	Solution
trajectory	(position/velocity/torque), trajectory segments too dense	interface types, and appropriately reduce trajectory 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 the RB-A850-10 motion control system (default configuration)
# This command will start all the necessary nodes, including the robot controller, motion planner, and
```

```
visualization tools.
ros2 launch raybot_bringup robot.launch.py model:=rb_a850_10
```

```
# Start RB-A575-3
ros2 launch raybot_bringup robot.launch.py model:=rb_a575_3
```

```
# Override JTC trajectory error tolerances at startup
ros2 launch raybot_bringup robot.launch.py model:=rb_a850_10 \
  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 right arm
ros2 service call /cli_controller/execute_command robot_controller/srv/ExecuteCommand "{command:
'group=r_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 PTP command with the OMPL planner
# Move the right arm (r_arm) to the target pose using point-to-point motion with OMPL
ros2 service call /cli_controller/execute_command robot_controller/srv/ExecuteCommand "{command:
'group=r_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 LIN motion command with the Pilz industrial motion planner
# Move the right arm (r_arm) to the target pose along a straight line
ros2 service call /cli_controller/execute_command robot_controller/srv/ExecuteCommand "{command:
'group=r_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 command with the OMPL planner
# Move joints 5, 6, and 7 of the right arm (r_arm) to the specified angles
ros2 service call /cli_controller/execute_command robot_controller/srv/ExecuteCommand "{command:
```

```
'group=r_arm type=joints joints=right_arm_7_joint:0.1,right_arm_6_joint:0.1,right_arm_5_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 right-arm PTP end-effector motion
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'add type=pose
group=r_arm frame=base_link ee=right_hand 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=Right 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 to the right arm (wait 1 second after execution)
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'add type=pose
group=r_arm frame=base_link ee=right_hand 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=Right arm pre-grasp pose 2'}"

# Add a Pilz LIN straight-line motion for the right arm
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'add type=pose
group=r_arm frame=base_link ee=right_hand 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=Right arm straight-line path'}"

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

# Add an OMPL end-effector pose command for the right arm
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'add type=pose
group=r_arm frame=base_link ee=right_hand pose=0.50,-0.10,0.35,0,1.57,0 planner=ompl vel=0.25
acc=0.25 exec=1 note=Right arm OMPL pose'}"

# Add a joint-space command for the right arm
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'add type=joints
```

```
group=r_arm joints=0,0.4,-0.2,0,0.6,0,-0.1 frame=base_link planner=ompl vel=0.25 acc=0.25 exec=1
note=Right arm joint move'}"

# Insert a wait step (2.5 seconds)
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'wait
wait_duration=2.5 note=Waiting for external signal'}"

# List all current steps
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'list'}"

# Save the motion sequence to a file
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'save
file=demo_program4'}"

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

# Delete one step by index
ros2 service call /action_recorder/edit robot_controller/srv/ProgramEdit "{command: 'del index=3'}"

# Delete multiple indexed steps
ros2 service call /action_recorder/delete_steps robot_controller/srv/DeleteProgramSteps "{indices: [2,4,5],
ignore_missing: false, report_remaining: true}"

# Execute the program with replanning at each step
ros2 action send_goal /action_recorder/command robot_controller/action/ProgramCommand "{command:
'play exec=1 replan=1 steps=[]'}"

# Execute the program without replanning
ros2 action send_goal /action_recorder/command robot_controller/action/ProgramCommand "{command:
'play exec=1 replan=0'}"

# Execute the program repeatedly
ros2 action send_goal /action_recorder/command robot_controller/action/ProgramCommand "{command:
'play exec=1 replan=0 repeat_count=3'}"

# Clear all in-memory program steps
ros2 service call /action_recorder/clear_program robot_controller/srv/ClearProgram {}

# Validate planning for selected steps only
ros2 service call /action_recorder/plan_program robot_controller/srv/PlanProgram "
{use_current_state_start: true, steps: [1,3,5]}"

# Execute selected steps only
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'}"
```

Object Grasp Node Commands

Execute Grasp

It is recommended to first use a minimal request to confirm that the vision TF, right-arm planning group, and gripper control chain are all working. Only move on to the full parameter template after the minimal example succeeds.

Minimal working example:

```
ros2 service call /object_grasp_node/pick robot_controller/srv/PerformGrasp "{
  group_name: 'r_arm',
  ee_link: 'right_hand',
  target_frame: 'cocacola_target',
  retreat_after_pick: 0,
  retreat_target: 'home',
  auto_clear_octomap: 1
}"
```

Full parameter examples:

```
# Object grasp demo 1 (linear motion planning, smoother motion)
# Use the right arm to grasp the object in the cocacola_target frame and place it in the coaster_target frame
# LIN motion planning is suitable for tasks with stricter path requirements
ros2 service call /object_grasp_node/pick robot_controller/srv/PerformGrasp "{
  group_name: 'r_arm',                    # Motion planning group name (right arm)
  ee_link: 'right_hand',                  # End effector link name
  target_frame: 'cocacola_target',        # Target object frame
  retreat_after_pick: 0,                  # Do not return to a named pose after grasping
  retreat_target: 'home',                 # Return pose after grasping (unused here)
  gripper_group_name: 'r_hand',           # Gripper planning group (example)
  gripper_joint_name: 'right_hand_virtual_joint',  # Gripper joint name (example)
  gripper_open_angle: '0.45',             # Open angle (rad, string)
  gripper_close_angle: '0.0',             # Close angle (rad, string)
  approach_directions: '1,1,0',           # Candidate approach direction vectors
  approach_speed: 0.1,                    # Speed scaling factor for all stages
  approach_acc: 0.1,                      # Acceleration scaling factor for all stages
  pre_grasp_offset: 0.12,                 # Distance from pre-grasp to grasp position
  lift_distance: 0.05,                    # Lift distance after grasping, also used for place descent
  action_pause: 1.0,                      # Pause time between each stage (seconds)
  auto_clear_octomap: 1,                  # Clear Octomap before planning
  pause_point_cloud: 1,                   # Pause incoming obstacle point clouds during motion
  octomap_wait_time: 4.0,                 # Wait time after clearing Octomap
  pre_grasp_pipeline_id: 'ompl',          # Planning pipeline for the pre-grasp stage
```

```
  pre_grasp_planner_id: 'RRTConnectkConfigDefault',  # Planner for the pre-grasp stage
  approach_pipeline_id: 'pilz_industrial_motion_planner',  # Planning pipeline for the approach stage
  approach_planner_id: 'LIN',               # LIN motion planning for the approach stage
  lift_pipeline_id: 'pilz_industrial_motion_planner',  # Planning pipeline for the lift stage
  lift_planner_id: 'LIN',                   # LIN motion planning for the lift stage
  auto_place: 1,                            # Enable automatic placement function.
  place_target_frame: 'coaster_target',       # Target frame for placement
  place_translate_pipeline_id: 'pilz_industrial_motion_planner',  # Pipeline for placement translation
  place_translate_planner_id: 'LIN',         # LIN planning for placement translation
  place_drop_pipeline_id: 'pilz_industrial_motion_planner',  # Pipeline for placement descent
  place_drop_planner_id: 'LIN',              # LIN planning for placement descent
  verify_before_execute: true,              # Plan all stages before execution
  auto_place_retreat: 1                     # Automatically retreat after placement
}"

# Object grasp demo 2 (point-to-point planning, faster response)
# Use the right arm to grasp the object in the cocacola_target frame and place it in the coaster_target frame
# PTP motion planning is suitable for tasks that prioritize response time
ros2 service call /object_grasp_node/pick robot_controller/srv/PerformGrasp "{
  group_name: 'r_arm',                      # Motion planning group name (right arm)
  ee_link: 'right_hand',                    # End effector link name
  target_frame: 'cocacola_target',          # Target object frame
  retreat_after_pick: 0,                    # Do not return to a named pose after grasping
  retreat_target: 'home',                   # Return pose after grasping (unused here)
  gripper_group_name: 'r_hand',             # Gripper planning group (example)
  gripper_joint_name: 'right_hand_virtual_joint',  # Gripper joint name (example)
  gripper_open_angle: '0.45',               # Open angle (rad, string)
  gripper_close_angle: '0.0',               # Close angle (rad, string)
  approach_directions: '1,1,0',             # Candidate approach direction vectors
  approach_speed: 0.1,                      # Speed scaling factor for all stages
  approach_acc: 0.1,                        # Acceleration scaling factor for all stages
  pre_grasp_offset: 0.12,                   # Distance from pre-grasp to grasp position
  lift_distance: 0.05,                      # Lift distance after grasping, also used for place descent
  action_pause: 1.0,                        # Pause time between each stage (seconds)
  auto_clear_octomap: 1,                    # Clear Octomap before planning
  pause_point_cloud: 1,                     # Pause incoming obstacle point clouds during motion
  octomap_wait_time: 4.0,                   # Wait time after clearing Octomap
  pre_grasp_pipeline_id: 'ompl',            # Planning pipeline for the pre-grasp stage
  pre_grasp_planner_id: 'RRTConnectkConfigDefault',  # Planner for the pre-grasp stage
  approach_pipeline_id: 'pilz_industrial_motion_planner',  # Planning pipeline for the approach stage
  approach_planner_id: 'PTP',               # PTP motion planning for the approach stage
  lift_pipeline_id: 'pilz_industrial_motion_planner',  # Planning pipeline for the lift stage
  lift_planner_id: 'PTP',                   # PTP motion planning for the lift stage
  auto_place: 1,                            # Enable automatic placement function
  place_target_frame: 'coaster_target',       # Target frame for placement
  place_translate_pipeline_id: 'pilz_industrial_motion_planner',  # Pipeline for placement translation
  place_translate_planner_id: 'PTP',         # PTP planning for placement translation
  place_drop_pipeline_id: 'pilz_industrial_motion_planner',  # Pipeline for placement descent
  place_drop_planner_id: 'PTP',              # PTP planning for placement descent
```

```
  verify_before_execute: true,            # Plan all stages before execution
  auto_place_retreat: 1                   # Automatically retreat after placement
}"
```

Motion Stop Command

Stop the Specified Action Group's Movement

```
# Stop all movements of the right arm assembly (emergency stop command).
ros2 topic pub --once /arm_motion_controller/stop_request std_msgs/msg/String "{data: 'r_arm'}"

# Stop all movements of the left arm assembly (emergency stop command).
ros2 topic pub --once /arm_motion_controller/stop_request std_msgs/msg/String "{data: 'l_arm'}"
```

Shut Down the Control System

```
# Stop harmonic dual-arm control software (normal shutdown procedure)
source install/setup.bash
ros2 service call /raybot_robot/bringup_shutdown std_srvs/srv/Trigger {}
```