ros2_control Documentation (Rolling Ridley)

Release Apr 2026

ros2_control Development Team

Apr 14, 2026

CONTENTS

1 Getting Started **1**
- 1.1 Installation . . . 1
- 1.2 Architecture . . . 2
- 1.3 Hardware Components . . . 5

2 ros2_control **9**
- 2.1 API Documentation . . . 9
- 2.2 Concepts . . . 9
- 2.3 Guidelines and Best Practices . . . 55

3 ros2_controllers **61**
- 3.1 Guidelines and Best Practices . . . 61
- 3.2 Controllers for Wheeled Mobile Robots . . . 75
- 3.3 Controllers for Manipulators and Other Robots . . . 99
- 3.4 Broadcasters . . . 148
- 3.5 Filters . . . 161
- 3.6 Common Controller Parameters . . . 163

4 Demos **165**
- 4.1 What you can find in this repository . . . 165
- 4.2 Goals . . . 165
- 4.3 Examples Overview . . . 166
- 4.4 Installation . . . 167
- 4.5 Quick Hints . . . 169
- 4.6 Examples . . . 170

5 Utilities **251**
- 5.1 Command Line Interface . . . 251
- 5.2 control_toolbox . . . 259
- 5.3 realtime_tools . . . 264

6 Simulator Integrations **267**
- 6.1 Hosted by ros-controls . . . 267
- 6.2 Community . . . 290

7 Release Notes **291**
- 7.1 Release Notes: Kilted Kaiju to Lyrical Luth . . . 291
- 7.2 Release Notes: Kilted Kaiju to Lyrical Luth . . . 292
- 7.3 Release Notes: Jazzy to Kilted . . . 294
- 7.4 Release Notes: Kilted Kaiju to Lyrical Luth . . . 294

8 Migration Guides **295**
8.1 Coming from ros_control (ROS 1) . . . 295
8.2 Between different ROS 2 distributions . . . 297

9 API Documentation **303**
9.1 ros2_control stack . . . 303
9.2 Per-Package API Documentation . . . 303

10 Supported Robots **307**
10.1 Communication protocols . . . 307
10.2 End-effectors . . . 307
10.3 Non robot-devices . . . 307
10.4 Official (supported by robot manufacturer) . . . 308
10.5 Unofficial (from the community) . . . 308

11 Resources **309**
11.1 Presentations . . . 309
11.2 Diagrams . . . 320
11.3 Images . . . 321

12 Contributing **339**
12.1 Contributing Guidelines . . . 339

13 Project Governance **343**
13.1 Current ros-controls PMC Constituents . . . 343
13.2 Current ros-controls Committers . . . 344
13.3 Past ros-controls PMC Constituents . . . 344
13.4 Repositories managed by the ros-controls PMC . . . 344
13.5 Releases, Versioning, and Public API . . . 345

14 Acknowledgements **349**
14.1 Maintainers . . . 349
14.2 Contributors . . . 349
14.3 Companies and Institutions . . . 350

15 Documentation Usage **351**

16 Documentation Downloads **353**

17 `ros2_control` Repositories **355**

18 Development Organisation and Communication **357**

CHAPTER

ONE

GETTING STARTED

1.1 Installation

1.1.1 Binary packages

The ros2_control framework is released for ROS 2 rolling on Ubuntu and RHEL according to REP-2000. To use it, you have to install `ros-rolling-ros2-control` and `ros-rolling-ros2-controllers` packages, e.g., by running the following commands:

For Ubuntu deb packages

```
sudo apt install ros-rolling-ros2-control ros-rolling-ros2-controllers
```

For RHEL (RPM) packages

```
sudo dnf install ros-rolling-ros2-control ros-rolling-ros2-controllers
```

1.1.2 Building from Source

The rolling branch is compatible with both Humble and Jazzy ROS distributions. You can find more information about this compatibility on the respective Humble and Jazzy versions of this page.

If you want to install the framework from source, e.g., for contributing to the framework, use the following commands:

- Download all repositories

```
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/
wget https://raw.githubusercontent.com/ros-controls/ros2_control_ci/master/ros_
↪controls.$ROS_DISTRO.repos
vcs import src < ros_controls.$ROS_DISTRO.repos
```

- Install dependencies:

```
rosdep update --rosdistro=$ROS_DISTRO
sudo apt-get update
rosdep install --from-paths src --ignore-src -r -y
```

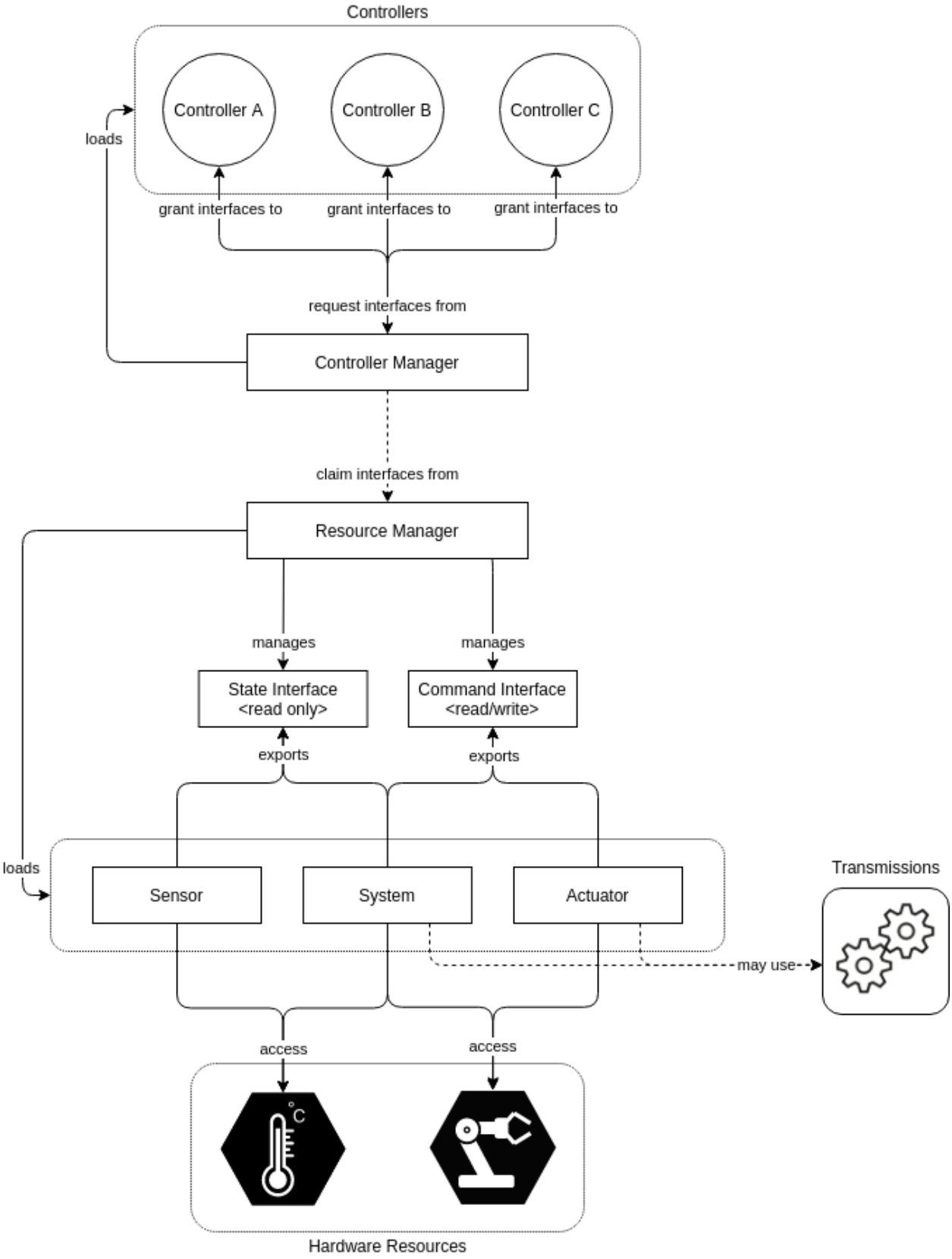
- Build everything, e.g. with:

```
. /opt/ros/${ROS_DISTRO}/setup.sh
colcon build --symlink-install
```

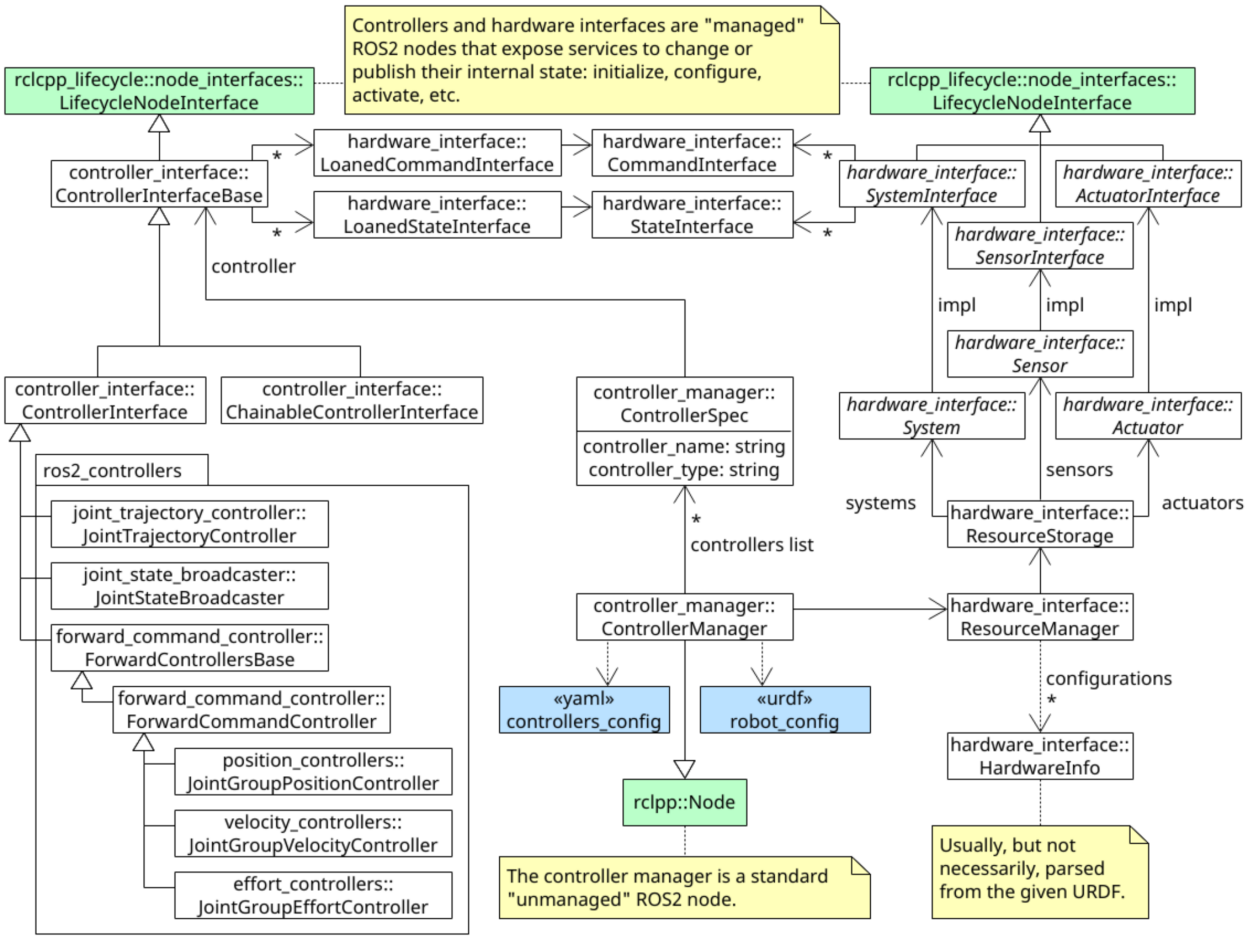
- Do not forget to source `setup.bash` from the `install` folder!

1.2 Architecture

The source code for the ros2_control framework can be found in the ros2_control and ros2_controllers GitHub repositories. The following figure shows the architecture of the ros2_control framework.



The following UML Class Diagram describes the internal implementation of the ros2_control framework.



1.2.1 Controller Manager

The *Controller Manager* (CM) connects the controllers and hardware-abstraction sides of the ros2_control framework. It also serves as the entry-point for users via ROS services. The CM implements a node without an executor so that it can be integrated into a custom setup. However, it's usually recommended to use the default node-setup implemented in ros2_control_node file from the `controller_manager` package. This manual assumes that you use this default node-setup.

On the one hand, CM manages (e.g. loads, activates, deactivates, unloads) controllers and the interfaces they require. On the other hand, it has access (via the Resource Manager) to the hardware components, i.e. their interfaces. The Controller Manager matches *required* and *provided* interfaces, granting controllers access to hardware when enabled, or reporting an error if there is an access conflict.

The execution of the control-loop is managed by the CM's `update()` method. It reads data from the hardware components, updates outputs of all active controllers, and writes the result to the components.

1.2.2 Resource Manager

The *Resource Manager* (RM) abstracts physical hardware and its drivers (called *hardware components*) for the ros2_control framework. The RM loads the components using the `pluginlib`-library, manages their lifecycle and components' state and command interfaces. The abstraction provided by RM allows reuse of implemented hardware components, e.g., robot and gripper, without any implementation, and flexible hardware application for state and command interfaces, e.g., separate hardware/communication libraries for motor control and encoder reading.

In the control loop execution, the RM's `read()` and `write()` methods handle the communication with the hardware components.

1.2.3 Controllers

The controllers in the ros2_control framework are based on control theory. They compare the reference value with the measured output and, based on this error, calculate a system's input. The controllers are objects derived from ControllerInterface (`controller_interface` package in ros2_control) and exported as plugins using `pluginlib`-library. For an example of a controller check the ForwardCommandController implementation in the ros2_controllers repository. The controller lifecycle is based on the LifecycleNode class, which implements the state machine described in the Node Lifecycle Design document.

When the control-loop is executed, the `update()` method is called. This method can access the latest hardware state and enable the controller to write to the hardware command interfaces.

1.2.4 User Interfaces

Users interact with the ros2_control framework using *Controller Manager*'s services. For a list of services and their definitions, check the `srv` folder in the controller_manager_msgs package.

While service calls can be used directly from the command line or via nodes, there exists a user-friendly `Command Line Interface` (CLI) which integrates with the `ros2 cli`. This supports auto-complete and has a range of common commands available. The base command is `ros2 control`. For the description of our CLI capabilities, see the *Command Line Interface (CLI) documentation*.

1.3 Hardware Components

The *hardware components* realize communication to physical hardware and represent its abstraction in the ros2_control framework. The components have to be exported as plugins using `pluginlib`-library. The *Resource Manager* dynamically loads those plugins and manages their lifecycle.

There are three basic types of components:

System
: Complex (multi-DOF) robotic hardware like industrial robots. The main difference between the *Actuator* component is the possibility to use complex transmissions like needed for humanoid robot's hands. This component has reading and writing capabilities. It is used when there is only one logical communication channel to the hardware (e.g., KUKA-RSI).

Sensor
: Robotic hardware is used for sensing its environment. A sensor component is related to a joint (e.g., encoder) or a link (e.g., force-torque sensor). This component type has only reading capabilities.

Actuator
: Simple (1 DOF) robotic hardware like motors, valves, and similar. An actuator implementation is related to only one joint. This component type has reading and writing capabilities. Reading is not mandatory if not possible (e.g.,

DC motor control with Arduino board). The actuator type can also be used with a multi-DOF robot if its hardware enables modular design, e.g., CAN-communication with each motor independently.

A detailed explanation of hardware components is given in the Hardware Access through Controllers design document.

1.3.1 Hardware Description in URDF

The ros2_control framework uses the `<ros2_control>`-tag in the robot's URDF file to describe its components, i.e., the hardware setup. The chosen structure enables tracking together multiple `xacro`-macros into one without any changes. The example hereunder shows a position-controlled robot with 2-DOF (RRBot), an external 1-DOF force-torque sensor, and an externally controlled 1-DOF parallel gripper as its end-effector. For more examples and detailed explanations, check the *ros2_control_demos site* and ROS 2 Control Components URDF Examples design document.

```
<ros2_control name="RRBotSystemPositionOnly" type="system">
 <hardware>
   <plugin>ros2_control_demo_hardware/RRBotSystemPositionOnlyHardware</plugin>
   <param name="example_param_write_for_sec">2</param>
   <param name="example_param_read_for_sec">2</param>
 </hardware>
 <joint name="joint1">
   <command_interface name="position">
     <param name="min">-1</param>
     <param name="max">1</param>
   </command_interface>
   <state_interface name="position"/>
 </joint>
 <joint name="joint2">
   <command_interface name="position">
     <param name="min">-1</param>
     <param name="max">1</param>
   </command_interface>
   <state_interface name="position"/>
 </joint>
</ros2_control>
<ros2_control name="RRBotForceTorqueSensor1D" type="sensor">
 <hardware>
   <plugin>ros2_control_demo_hardware/ForceTorqueSensor1DHardware</plugin>
   <param name="example_param_read_for_sec">0.43</param>
 </hardware>
 <sensor name="tcp_fts_sensor">
   <state_interface name="force"/>
   <param name="frame_id">rrbot_tcp</param>
   <param name="min_force">-100</param>
   <param name="max_force">100</param>
 </sensor>
</ros2_control>
<ros2_control name="RRBotGripper" type="actuator">
 <hardware>
   <plugin>ros2_control_demo_hardware/PositionActuatorHardware</plugin>
   <param name="example_param_write_for_sec">1.23</param>
   <param name="example_param_read_for_sec">3</param>
 </hardware>
 <joint name="gripper_joint ">
   <command_interface name="position">
     <param name="min">0</param>
     <param name="max">50</param>
   </command_interface>
```

(continues on next page)

(continued from previous page)

```
    <state_interface name="position"/>
    <state_interface name="velocity"/>
  </joint>
</ros2_control>
```

1.3.2 Running the Framework for Your Robot

To run the ros2_control framework, do the following. The example files can be found in the ros2_control_demos repository.

1. Create a YAML file with the configuration of the controller manager and two controllers. (Example configuration for RRBot)
2. Extend the robot's URDF description with needed `<ros2_control>` tags. It is recommended to use macro files (xacro) instead of pure URDF. (Example URDF for RRBot)
3. Create a launch file to start the node with *Controller Manager*. You can use a default ros2_control node (recommended) or integrate the controller manager in your software stack. (Example launch file for RRBot)

NOTE: You could alternatively use a script to create setup a skeleton of the "robot bringup" package by using the scripts provided by one of our maintainers. Extension of xacro for ros2_control file you can find in the templates for "robot description" package.

CHAPTER

TWO

ROS2_CONTROL

This is the documentation of the ros2_control framework core.

GitHub Repository

2.1 API Documentation

API documentation is parsed by doxygen and can be found here

2.2 Concepts

2.2.1 Controller Manager

Controller Manager is the main component in the ros2_control framework. It manages lifecycle of controllers, access to the hardware interfaces and offers services to the ROS-world.

Determinism

For best performance when controlling hardware you want the controller manager to have as little jitter as possible in the main control loop.

Independent of the kernel installed, the main thread of Controller Manager attempts to configure `SCHED_FIFO` with a priority of `50`. Read more about the scheduling policies for example here.

For real-time tasks, a priority range of 0 to 99 is expected, with higher numbers indicating higher priority. By default, users do not have permission to set such high priorities. To give the user such permissions, add a group named realtime and add the user controlling your robot to this group:

```
$ sudo addgroup realtime
$ sudo usermod -a -G realtime $(whoami)
```

Afterwards, add the following limits to the realtime group in `/etc/security/limits.conf`:

```
@realtime soft rtprio 99
@realtime soft priority 99
@realtime soft memlock unlimited
@realtime hard rtprio 99
@realtime hard priority 99
@realtime hard memlock unlimited
```

The limits will be applied after you log out and in again.

You can run ros2_control with real-time requirements also from a docker container. Pass the following capability options to allow the container to set the thread priority and lock memory, e.g.,

```
$ docker run -it \
    --cap-add=sys_nice \
    --ulimit rtprio=99 \
    --ulimit memlock=-1 \
    --rm --net host <IMAGE>
```

For more information, see the Docker engine documentation about resource_constraints and linux capabilities.

The normal linux kernel is optimized for computational throughput and therefore is not well suited for hardware control. Alternatives to the standard kernel include

- Real-time Ubuntu on Ubuntu (also for RaspberryPi)
- linux-image-rt-amd64 or linux-image-rt-arm64 on Debian for 64-bit PCs
- lowlatency kernel (`sudo apt install linux-lowlatency`) on any Ubuntu

Though installing a realtime-kernel will definitely get the best results when it comes to low jitter, using a lowlatency kernel can improve things a lot with being really easy to install.

Note: Avoid using the get_lifecycle_state() method in the real-time control loop of the controllers and the hardware components as it is not real-time safe.

Publishers

~/activity [controller_manager_msgs::msg::ControllerManagerActivity]
: A topic that is published every time there is a change of state of the controllers or hardware components managed by the controller manager. The message contains the list of the controllers and the hardware components that are managed by the controller manager along with their lifecycle states. The topic is published using the "transient local" quality of service, so subscribers should also be "transient local".

Subscribers

robot_description [std_msgs::msg::String]
: String with the URDF xml, e.g., from `robot_state_publisher`. Reloading of the URDF is not supported yet. All joints defined in the `<ros2_control>`-tag have to be present in the URDF.

Parameters

<controller_name>.type
: Name of a plugin exported using `pluginlib` for a controller. This is a class from which controller's instance with name "`controller_name`" is created.

<controller_name>.params_file
: The absolute path to the YAML file with parameters for the controller. The file should contain the parameters for the controller in the standard ROS 2 YAML format.

<controller_name>.fallback_controllers
: List of controllers that are activated as a fallback strategy, when the spawned controllers fail by returning `return_type::ERROR` during the `update` cycle. It is recommended to add all the controllers needed for the

fallback strategy to the list, including the chainable controllers whose interfaces are used by the main fallback controllers.

Warning: The fallback controllers activation is subject to the availability of the state and command interfaces at the time of activation. It is recommended to test the fallback strategy in simulation before deploying it on the real robot.

update_rate (int)
The frequency of controller manager's real-time update loop. This loop reads states from hardware, updates controllers and writes commands to hardware.

Read only: True

Default: 100

enforce_command_limits (bool)
If true, the controller manager will enforce command limits defined in the robot description. If false, no limits will be enforced. If true, when the command is outside the limits, the command is clamped to be within the limits depending on the type of configured joint limits defined in the robot description. If the command is within the limits, the command is passed through without any changes.

Read only: True

Default: true

handle_exceptions (bool)
If true, the controller manager will catch exceptions thrown during the different operations of controllers and hardware components. If false, exceptions will propagate up and will cause the controller manager to crash.

Read only: True

Default: true

hardware_components_initial_state
Map of parameters for controlled lifecycle management of hardware components. The names of the components are defined as attribute of `<ros2_control>`-tag in `robot_description`. Hardware components found in `robot_description`, but without explicit state definition will be immediately activated. Detailed explanation of each parameter is given below. The full structure of the map is given in the following example:

```
hardware_components_initial_state:
  unconfigured:
    - "arm1"
    - "arm2"
  inactive:
    - "base3"
```

hardware_components_initial_state.unconfigured (string_array)
Defines which hardware components will be only loaded when controller manager is started. These hardware components will need to be configured and activated manually or via a hardware spawner.

Default: {}

Constraints:

- contains no duplicates

hardware_components_initial_state.inactive (string_array)
Defines which hardware components will be configured when controller manager is started. These hardware components will need to be activated manually or via a hardware spawner.

Default: {}

Constraints:

- contains no duplicates

hardware_components_initial_state.shutdown_on_initial_state_failure (bool)
Specifies whether the controller manager should shut down if setting the desired initial state fails during startup.

Read only: True

Default: false

defaults.switch_controller.strictness (string)
The default switch controller strategy. This strategy is used when no strategy is specified in the switch_controller service call.

Default: "strict"

Constraints:

- parameter is not empty
- one of the specified values: ['strict', 'best_effort']

defaults.allow_controller_activation_with_inactive_hardware (bool)
If true, controllers are allowed to claim resources from inactive hardware components. If false, controllers can only claim resources from active hardware components. However, it is not recommended to set this parameter to true for the safety reasons with the hardware and unexpected movement, this is purely added for backward compatibility.

Read only: True

Default: false

defaults.deactivate_controllers_on_hardware_self_deactivate (bool)
If set to true, when a hardware component returns DEACTIVATE on the write cycle, controllers using those interfaces will be deactivated. When set to false, controllers using those interfaces will continue to run. It is not recommended to set this parameter to false for safety reasons with hardware. This will be the default behaviour of the controller manager and this parameter will be removed in future releases. Please use with caution.

Read only: True

Default: true

diagnostics.threshold.controller_manager.periodicity.mean_error.warn (double)
The warning threshold for the mean error of the controller manager's periodicity in Hz. If the mean error exceeds this threshold, a warning diagnostic will be published.

Default: 5.0

Constraints:

- greater than 0.0

diagnostics.threshold.controller_manager.periodicity.mean_error.error (double)
The error threshold for the mean error of the controller manager's periodicity in Hz. If the mean error exceeds this threshold, an error diagnostic will be published.

Default: 10.0

Constraints:

- greater than 0.0

diagnostics.threshold.controller_manager.periodicity.standard_deviation.warn (double)
The warning threshold for the standard deviation of the controller manager's periodicity in Hz. If the standard deviation exceeds this threshold, a warning diagnostic will be published.

Default: 5.0

Constraints:

- greater than 0.0

diagnostics.threshold.controller_manager.periodicity.standard_deviation.error (double)
The error threshold for the standard deviation of the controller manager's periodicity in Hz. If the standard deviation exceeds this threshold, an error diagnostic will be published.

Default: 10.0

Constraints:

- greater than 0.0

diagnostics.threshold.controllers.periodicity
The `periodicity` diagnostics will be published for the asynchronous controllers, because any affect to the synchronous controllers will be reflected directly in the controller manager's periodicity. It is also published for the synchronous controllers that have a different update rate than the controller manager update rate.

diagnostics.threshold.controllers.periodicity.mean_error.warn (double)
The warning threshold for the mean error of the controller update loop in Hz. If the mean error exceeds this threshold, a warning diagnostic will be published.

Default: 5.0

Constraints:

- greater than 0.0

diagnostics.threshold.controllers.periodicity.mean_error.error (double)
The error threshold for the mean error of the controller update loop in Hz. If the mean error exceeds this threshold, an error diagnostic will be published.

Default: 10.0

Constraints:

- greater than 0.0

diagnostics.threshold.controllers.periodicity.standard_deviation.warn (double)
The warning threshold for the standard deviation of the controller update loop in Hz. If the standard deviation exceeds this threshold, a warning diagnostic will be published.

Default: 5.0

Constraints:

- greater than 0.0

diagnostics.threshold.controllers.periodicity.standard_deviation.error (double)
The error threshold for the standard deviation of the controller update loop in Hz. If the standard deviation exceeds this threshold, an error diagnostic will be published.

Default: 10.0

Constraints:

- greater than 0.0

diagnostics.threshold.controllers.execution_time.mean_error
The `execution_time` diagnostics will be published for all controllers. The `mean_error` for a synchronous controller will be computed against zero, as it should be as low as possible. However, the `mean_error` for an asynchronous controller will be computed against the controller's desired update period, as the controller can take a maximum of the desired period to execute its update cycle.

diagnostics.threshold.controllers.execution_time.mean_error.warn (double)
The warning threshold for the mean error of the controller's update cycle execution time in microseconds. If the mean error exceeds this threshold, a warning diagnostic will be published.

Default: 1000.0

Constraints:

- greater than 0.0

diagnostics.threshold.controllers.execution_time.mean_error.error (double)
The error threshold for the mean error of the controller's update cycle execution time in microseconds. If the mean error exceeds this threshold, an error diagnostic will be published.

Default: 2000.0

Constraints:

- greater than 0.0

diagnostics.threshold.controllers.execution_time.standard_deviation.warn (double)
The warning threshold for the standard deviation of the controller's update cycle execution time in microseconds. If the standard deviation exceeds this threshold, a warning diagnostic will be published.

Default: 100.0

Constraints:

- greater than 0.0

diagnostics.threshold.controllers.execution_time.standard_deviation.error (double)
The error threshold for the standard deviation of the controller's update cycle execution time in microseconds. If the standard deviation exceeds this threshold, an error diagnostic will be published.

Default: 200.0

Constraints:

- greater than 0.0

diagnostics.threshold.hardware_components.periodicity
The `periodicity` diagnostics will be published for the asynchronous hardware components, because any affect to the synchronous hardware components will be reflected directly in the controller manager's periodicity. It is also published for the synchronous hardware components that have a different read/write rate than the controller manager update rate.

diagnostics.threshold.hardware_components.periodicity.mean_error.warn (double)
The warning threshold for the mean error of the hardware component's read/write loop in Hz. If the mean error exceeds this threshold, a warning diagnostic will be published.

Default: 5.0

Constraints:

- greater than 0.0

diagnostics.threshold.hardware_components.periodicity.mean_error.error (double)
The error threshold for the mean error of the hardware component's read/write loop in Hz. If the mean error exceeds this threshold, an error diagnostic will be published.

Default: 10.0

Constraints:

- greater than 0.0

diagnostics.threshold.hardware_components.periodicity.standard_deviation.warn (double)
The warning threshold for the standard deviation of the hardware component's read/write loop in Hz. If the standard deviation exceeds this threshold, a warning diagnostic will be published.

Default: 5.0

Constraints:

- greater than 0.0

diagnostics.threshold.hardware_components.periodicity.standard_deviation.error (double)
The error threshold for the standard deviation of the hardware component's read/write loop in Hz. If the standard deviation exceeds this threshold, an error diagnostic will be published.

Default: 10.0

Constraints:

- greater than 0.0

diagnostics.threshold.hardware_components.execution_time.mean_error
The `execution_time` diagnostics will be published for all hardware components. The `mean_error` for a synchronous hardware component will be computed against zero, as it should be as low as possible. However, the `mean_error` for an asynchronous hardware component will be computed against its desired read/write period, as the hardware component can take a maximum of the desired period to execute the read/write cycle.

diagnostics.threshold.hardware_components.execution_time.mean_error.warn (double)
The warning threshold for the mean error of the hardware component's read/write cycle execution time in microseconds. If the mean error exceeds this threshold, a warning diagnostic will be published.

Default: 1000.0

Constraints:

- greater than 0.0

diagnostics.threshold.hardware_components.execution_time.mean_error.error (double)
The error threshold for the mean error of the hardware component's read/write cycle execution time in microseconds. If the mean error exceeds this threshold, a error diagnostic will be published.

Default: 2000.0

Constraints:

- greater than 0.0

diagnostics.threshold.hardware_components.execution_time.standard_deviation.warn (double)
The warning threshold for the standard deviation of the hardware component's read/write cycle execution time in microseconds. If the standard deviation exceeds this threshold, a warning diagnostic will be published.

Default: 100.0

Constraints:

- greater than 0.0

diagnostics.threshold.hardware_components.execution_time.standard_deviation.error (double)
The error threshold for the standard deviation of the hardware component's update cycle execution time in microseconds. If the standard deviation exceeds this threshold, an error diagnostic will be published.

Default: 200.0

Constraints:

- greater than 0.0

overruns.print_warnings (bool)
If true, the controller manager will print a warning message to the console if an overrun is detected in its real-time loop (`read`, `update` and `write`). By default, it is set to true, except when used with `use_sim_time` parameter set to true.

An example parameter file:

```yaml
controller_manager:
  ros__parameters:
    defaults:
      allow_controller_activation_with_inactive_hardware: false
      deactivate_controllers_on_hardware_self_deactivate: true
      switch_controller:
        strictness: strict
    diagnostics:
      threshold:
        controller_manager:
          periodicity:
            mean_error:
              error: 10.0
              warn: 5.0
            standard_deviation:
              error: 10.0
              warn: 5.0
        controllers:
          execution_time:
            mean_error:
              error: 2000.0
              warn: 1000.0
            standard_deviation:
              error: 200.0
              warn: 100.0
          periodicity:
            mean_error:
              error: 10.0
              warn: 5.0
            standard_deviation:
              error: 10.0
              warn: 5.0
        hardware_components:
          execution_time:
            mean_error:
              error: 2000.0
              warn: 1000.0
            standard_deviation:
              error: 200.0
              warn: 100.0
          periodicity:
            mean_error:
              error: 10.0
              warn: 5.0
            standard_deviation:
              error: 10.0
              warn: 5.0
    enforce_command_limits: true
    handle_exceptions: true
    hardware_components_initial_state:
      inactive: '{}'
```

(continues on next page)

(continued from previous page)

```
      shutdown_on_initial_state_failure: false
      unconfigured: '{}'
    overruns:
      print_warnings: ''
    update_rate: 100.0
```

Handling Multiple Controller Managers

When dealing with multiple controller managers, you have two options for managing different robot descriptions:

1. **Using Namespaces:** You can place both the `robot_state_publisher` and the `controller_manager` nodes into the same namespace.

```
control_node = Node(
    package="controller_manager",
    executable="ros2_control_node",
    parameters=[robot_controllers],
    output="both",
    namespace="rrbot",
)
robot_state_pub_node = Node(
    package="robot_state_publisher",
    executable="robot_state_publisher",
    output="both",
    parameters=[robot_description],
    namespace="rrbot",
)
```

2. **Using Remappings:** You can use remappings to handle different robot descriptions. This involves relaying topics using the `remappings` tag, allowing you to specify custom topics for each controller manager.

```
control_node = Node(
    package="controller_manager",
    executable="ros2_control_node",
    parameters=[robot_controllers],
    output="both",
    remappings=[('robot_description', '/rrbot/robot_description')]
)
robot_state_pub_node = Node(
    package="robot_state_publisher",
    executable="robot_state_publisher",
    output="both",
    parameters=[robot_description],
    namespace="rrbot",
)
```

Helper scripts

There are two scripts to interact with controller manager from launch files:

1. `spawner` - loads, configures and start a controller on startup.
2. `unspawner` - stops and unloads a controller.
3. `hardware_spawner` - activates and configures a hardware component.

spawner

```
$ ros2 run controller_manager spawner -h
usage: spawner [-h] [-c CONTROLLER_MANAGER] [-p PARAM_FILE] [--load-only] [--
↪inactive] [-u] [--controller-manager-timeout CONTROLLER_MANAGER_TIMEOUT] [--switch-
↪timeout SWITCH_TIMEOUT]
            [--service-call-timeout SERVICE_CALL_TIMEOUT] [--activate-as-group] [--
↪switch-asap | --no-switch-asap] [--controller-ros-args CONTROLLER_ROS_ARGS]
            controller_names [controller_names ...]

positional arguments:
  controller_names      List of controllers

options:
  -h, --help            show this help message and exit
  -c CONTROLLER_MANAGER, --controller-manager CONTROLLER_MANAGER
                        Name of the controller manager ROS node
  -p PARAM_FILE, --param-file PARAM_FILE
                        Controller param file to be loaded into controller node␣
↪before configure. Pass multiple times to load different files for different␣
↪controllers or to override the parameters of the same controller.
  --load-only           Only load the controller and leave unconfigured.
  --inactive            Load and configure the controller, however do not activate␣
↪them
  -u, --unload-on-kill  Wait until this application is interrupted and unload␣
↪controller
  --controller-manager-timeout CONTROLLER_MANAGER_TIMEOUT
                        Time to wait for the controller manager service to be␣
↪available
  --switch-timeout SWITCH_TIMEOUT
                        Time to wait for a successful state switch of controllers.␣
↪Useful when switching cannot be performed immediately, e.g., paused simulations at␣
↪startup
  --service-call-timeout SERVICE_CALL_TIMEOUT
                        Time to wait for the service response from the controller␣
↪manager
  --activate-as-group   Activates all the parsed controllers list together instead of␣
↪one by one. Useful for activating all chainable controllers altogether
  --switch-asap, --no-switch-asap
                        Option to switch the controllers in the realtime loop at the␣
↪earliest possible time or in the non-realtime loop.
  --controller-ros-args CONTROLLER_ROS_ARGS
                        The --ros-args to be passed to the controller node, e.g., for␣
↪remapping topics. Pass multiple times for every argument.

When launching ``spawner`` with ROS parameter files that use substitutions (for␣
↪example, launch ``allow_substs=True``),
```

(continues on next page)

(continued from previous page)

```
the resolved ``--params-file`` path(s) used by the spawner node are automatically␣
↪forwarded to each controller along
with any explicit ``--param-file`` arguments passed to the spawner command.
```

Note: If a single parameter file is used for multiple controllers, the spawner will automatically forward the resolved path(s) to each controller. The following methods are recommended:

```
Node(
    package="controller_manager",
    executable="spawner",
    arguments=[
        "my_controller",
        "--param-file",
        PathSubstitution(FindPackageShare("my_config_pkg"))
        / "config"
        / "controllers.yaml",
    ],
),
```

(or)

```
Node(
    package="controller_manager",
    executable="spawner",
    parameters=[
        ParameterFile(
            PathSubstitution(FindPackageShare("my_config_pkg"))
            / "config"
            / "controllers.yaml",
        ),
    ],
    arguments=[
        "my_controller"
    ],
),
```

The `spawner` now supports per controller arguments, while parsing the arguments for multiple controllers using `--controller` option. For example, to spawn two controllers with different parameter files and remapping topics, the following command can be used:

```
$ ros2 run controller_manager spawner --controller position_trajectory_controller \
    --param-file /path/to/position_trajectory_controller_params.yaml \
    --controller-ros-args "--ros-args --remap /joint_states:=/rrbot/joint_states" \
    --controller velocity_trajectory_controller \
    --param-file /path/to/velocity_trajectory_controller_params.yaml \
    --controller-ros-args "--ros-args --remap /joint_states:=/rrbot/joint_states"
```

```
$ ros2 run  controller_manager spawner --controller -h
Usage: spawner [global_options] --controller <name> [controller_options] --controller
↪<name> ...

Global Options:
usage: spawner [-c CONTROLLER_MANAGER] [--controller-manager-timeout CONTROLLER_
↪MANAGER_TIMEOUT] [--switch-timeout SWITCH_TIMEOUT] [--service-call-timeout SERVICE_
```

(continues on next page)

(continued from previous page)

```
↪CALL_TIMEOUT] [--activate-as-group]
              [--switch-asap | --no-switch-asap] [-u] [-h]

options:
  -c CONTROLLER_MANAGER, --controller-manager CONTROLLER_MANAGER
                        Name of the controller manager
  --controller-manager-timeout CONTROLLER_MANAGER_TIMEOUT
                        Timeout for controller manager services
  --switch-timeout SWITCH_TIMEOUT
                        Timeout for switch controller service
  --service-call-timeout SERVICE_CALL_TIMEOUT
                        Timeout for service calls
  --activate-as-group   Activate controllers as a group
  --switch-asap, --no-switch-asap
                        Switch controllers as soon as possible
  -u, --unload-on-kill  Deactivate the active controllers and unload them on kill
  -h, --help            Show help

Controller Options:
usage: spawner [-p PARAM_FILE] [--load-only] [--inactive] [--controller-ros-args␣
↪CONTROLLER_ROS_ARGS] controller_name

positional arguments:
  controller_name       Name of the controller

options:
  -p PARAM_FILE, --param-file PARAM_FILE
                        Parameter files to load for the controller
  --load-only           Load the controller but do not configure/activate it
  --inactive            Configure the controller but do not switch it
  --controller-ros-args CONTROLLER_ROS_ARGS
                        ROS arguments to pass to the controller
```

The parsed controller config file can follow the same conventions as the typical ROS 2 parameter file format. Now, the spawner can handle config files with wildcard entries and also the controller name in the absolute namespace. See the following examples on the config files:

```yaml
/**:
  ros__parameters:
    type: joint_trajectory_controller/JointTrajectoryController

    command_interfaces:
      - position
      .....

position_trajectory_controller_joint1:
  ros__parameters:
    joints:
      - joint1

position_trajectory_controller_joint2:
  ros__parameters:
    joints:
      - joint2
```

```yaml
/**/position_trajectory_controller:
```

(continues on next page)

(continued from previous page)

```
  ros__parameters:
    type: joint_trajectory_controller/JointTrajectoryController
    joints:
      - joint1
      - joint2

    command_interfaces:
      - position
      .....
```

```
/position_trajectory_controller:
  ros__parameters:
    type: joint_trajectory_controller/JointTrajectoryController
    joints:
      - joint1
      - joint2

    command_interfaces:
      - position
      .....
```

```
position_trajectory_controller:
  ros__parameters:
    type: joint_trajectory_controller/JointTrajectoryController
    joints:
      - joint1
      - joint2

    command_interfaces:
      - position
      .....
```

```
/rrbot_1/position_trajectory_controller:
  ros__parameters:
    type: joint_trajectory_controller/JointTrajectoryController
    joints:
      - joint1
      - joint2

    command_interfaces:
      - position
      .....
```

unspawner

```
$ ros2 run controller_manager unspawner -h
usage: unspawner [-h] [-c CONTROLLER_MANAGER] [--switch-timeout SWITCH_TIMEOUT]␣
↪controller_names [controller_names ...]

positional arguments:
  controller_names      Name of the controller

options:
```

(continues on next page)

(continued from previous page)

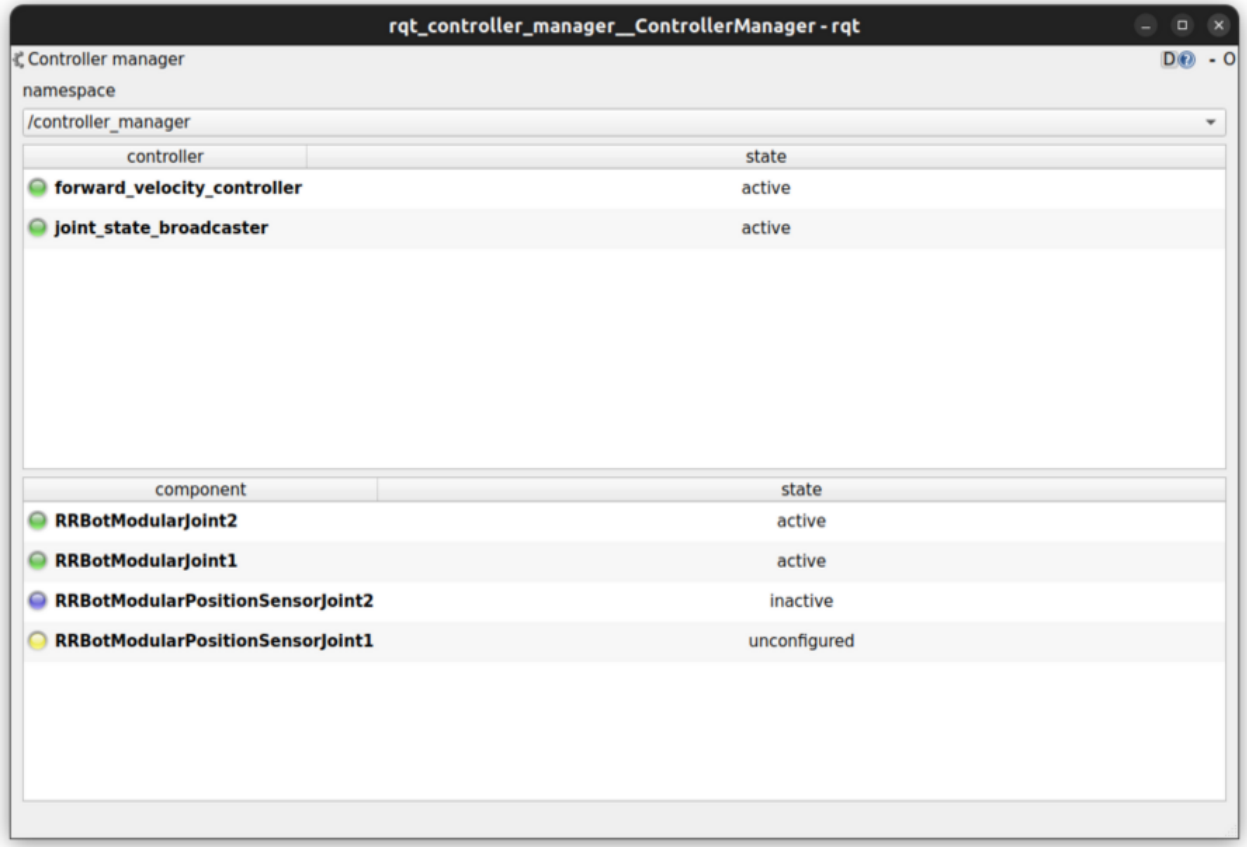
```
  -h, --help            show this help message and exit
  -c CONTROLLER_MANAGER, --controller-manager CONTROLLER_MANAGER
                        Name of the controller manager ROS node
  --switch-timeout SWITCH_TIMEOUT
                        Time to wait for a successful state switch of controllers.␣
↪Useful if controllers cannot be switched immediately, e.g., paused
                        simulations at startup
```

hardware_spawner

```
$ ros2 run controller_manager hardware_spawner -h
usage: hardware_spawner [-h] [-c CONTROLLER_MANAGER] [--controller-manager-timeout␣
↪CONTROLLER_MANAGER_TIMEOUT]
                        (--activate | --configure)
                        hardware_component_names [hardware_component_names ...]

positional arguments:
  hardware_component_names
                        The name of the hardware components which should be activated.

options:
  -h, --help            show this help message and exit
  -c CONTROLLER_MANAGER, --controller-manager CONTROLLER_MANAGER
                        Name of the controller manager ROS node
  --controller-manager-timeout CONTROLLER_MANAGER_TIMEOUT
                        Time to wait for the controller manager
  --activate            Activates the given components. Note: Components are by␣
↪default configured before activated.
  --configure           Configures the given components.
```

rqt_controller_manager

A GUI tool to interact with the controller manager services to be able to switch the lifecycle states of the controllers as well as the hardware components.



It can be launched independently using the following command or as rqt plugin:

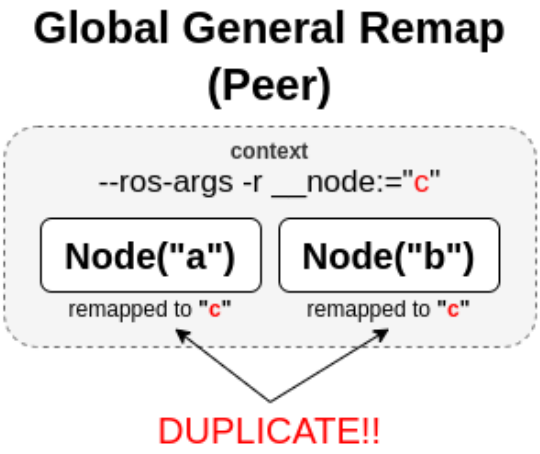
```
ros2 run rqt_controller_manager rqt_controller_manager
```

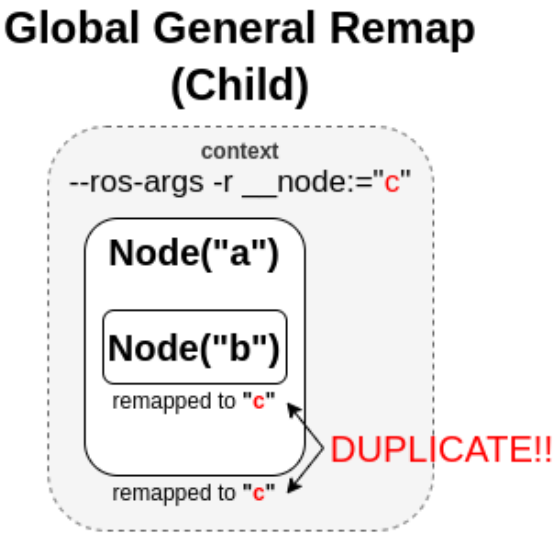
- Double-click on a controller or hardware component to show the additional info.
- Right-click on a controller or hardware component to show a context menu with options for lifecycle management.

Using the Controller Manager in a Process

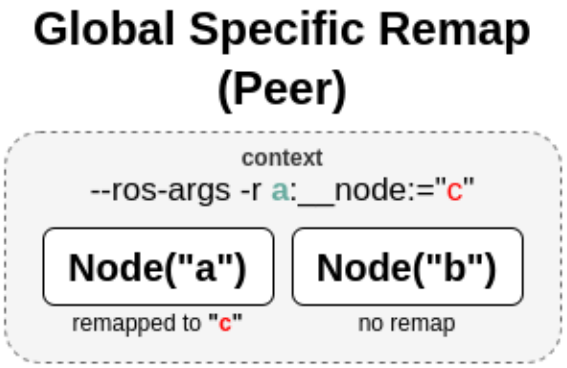
The `ControllerManager` may also be instantiated in a process as a class, but proper care must be taken when doing so. The reason for this is because the `ControllerManager` class inherits from `rclcpp::Node`.

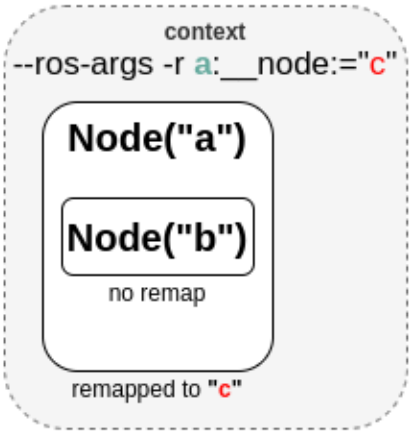
If there is more than one Node in the process, global node name remap rules can forcibly change the `ControllerManager's` node name as well, leading to duplicate node names. This occurs whether the Nodes are siblings or exist in a hierarchy.





The workaround for this is to specify another node name remap rule in the `NodeOptions` passed to the `Controller-Manager` node (causing it to ignore the global rule), or ensure that any remap rules are targeted to specific nodes.






```
auto options = controller_manager::get_cm_node_options();
  options.arguments({
    "--ros-args",
    "--remap", "_target_node_name:__node:=dst_node_name",
    "--log-level", "info"});

  auto cm = std::make_shared<controller_manager::ControllerManager>(
    executor, "_target_node_name", "some_optional_namespace", options);
```

Launching controller_manager with ros2_control_node

The controller_manager can be launched with the ros2_control_node executable. The following example shows how to launch the controller_manager with the ros2_control_node executable:

```
control_node = Node(
    package="controller_manager",
    executable="ros2_control_node",
    parameters=[robot_controllers],
    output="both",
)
```

The ros2_control_node executable uses the following parameters from the `controller_manager` node:

lock_memory (optional; bool; default: false for a non-realtime kernel, true for a realtime kernel)
: Locks the memory of the `controller_manager` node at startup to physical RAM in order to avoid page faults and to prevent the node from being swapped out to disk. Find more information about the setup for memory locking in the following link : How to set ulimit values The following command can be used to set the memory locking limit temporarily : `ulimit -l unlimited`.

cpu_affinity (optional; int (or) int_array;)
: Sets the CPU affinity of the `controller_manager` node to the specified CPU core. If it is an integer, the node's affinity will be set to the specified CPU core. If it is an array of integers, the node's affinity will be set to the specified set of CPU cores.

thread_priority (optional; int; default: 50)
: Sets the thread priority of the `controller_manager` node to the specified value. The value must be between 0 and 99.

use_sim_time (optional; bool; default: false)
: Enables the use of simulation time in the `controller_manager` node.

overruns.manage (optional; bool; default: true)
: Enables or disables the handling of overruns in the real-time loop of the `controller_manager` node. If set to true, the controller manager will detect overruns caused by system time changes or longer execution times of the controllers and hardware components. If an overrun is detected, the controller manager will print a warning message to the console. When used with `use_sim_time` set to true, this parameter is ignored and the overrun handling is disabled.

Concepts

Restarting all controllers

The simplest way to restart all controllers is by using `switch_controllers` services or CLI and adding all controllers to `start` and `stop` lists. Note that not all controllers have to be restarted, e.g., broadcasters.

Restarting hardware

If hardware gets restarted then you should go through its lifecycle again in order to reconfigure and export the interfaces

Hardware and Controller Errors

If the hardware during it's `read` or `write` method returns `return_type::ERROR`, the controller manager will stop all controllers that are using the hardware's command and state interfaces. Likewise, if a controller returns `return_type::ERROR` from its `update` method, the controller manager will deactivate the respective controller (or) the entire controller chain it is part of, then the controller manager will try to start any available fallback controllers.

Factors that affect Determinism

When run under the conditions determined in the above section, the determinism is assured up to the limitations of the hardware and the real-time kernel. However, there are some situations that can affect determinism:

- When a controller fails to activate in the realtime loop, the controller_manager will call the methods `prepare_command_mode_switch` and `perform_command_mode_switch` to stop the started interfaces. These calls can cause jitter in the main control loop.
- If a controller does not complete a successful update cycle in the realtime loop (for example, returns `return_type::ERROR`), the controller manager will deactivate that controller (or) the entire controller chain it is part of. It will then invoke `prepare_command_mode_switch` and `perform_command_mode_switch` to stop the interfaces used by the affected controller(s). These actions can introduce jitter into the main control loop.

Support for Asynchronous Updates

For some applications, it is desirable to run a controller at a lower frequency than the controller manager's update rate. For instance, if the `update_rate` for the controller manager is 100Hz, the sum of the execution times of all controllers' `update` calls and hardware components `read` and `write` calls must be below 10ms. If one controller requires 15ms of execution time, it cannot be executed synchronously without affecting the overall system update rate. Running a controller asynchronously can be beneficial in this scenario.

The async update support is transparent to each controller implementation. A controller can be enabled for asynchronous updates by setting the `is_async` parameter to `true`. The controller manager will load the controller accordingly. For example:

```
controller_manager:
  ros__parameters:
    update_rate: 100  # Hz
    ...

example_async_controller:
```

(continues on next page)

(continued from previous page)

```
  ros__parameters:
    type: example_controller/ExampleAsyncController
    is_async: true
    update_rate: 20  # Hz
    ...
```

You can set more parameters of the `AsyncFunctionHandler`, which handles the thread of the controller, using the `async_parameters` namespace. For example:

```
example_async_controller:
  ros__parameters:
    type: example_controller/ExampleAsyncController
    update_rate: 20  # Hz
    is_async: true
    async_parameters:
      cpu_affinity: [2, 4]
      thread_priority: 50
      wait_until_initial_trigger: false
      ...
```

will result in the controller being loaded and configured to run at 20Hz, while the controller manager runs at 100Hz. The description of the parameters can be found in the Common Controller Parameters section of the ros2_controllers documentation.

Scheduling Behavior

From a design perspective, the controller manager functions as a scheduler that triggers updates for asynchronous controllers during the control loop.

In this case, the `ControllerInterfaceBase` calls `AsyncFunctionHandler` to handle the actual `update` callback of the controller, which is the same mechanism used by the resource manager to support read/write operations for asynchronous hardware. When a controller is configured to run asynchronously, the controller interface creates an async handler during the controller's configuration and binds it to the controller's update method. The async handler thread created by the controller interface has either the same thread priority as the controller manager or the priority specified by the `thread_priority` parameter. When triggered by the controller manager, the async handler evaluates if the previous trigger is successfully finished and then calls the update method.

If the update takes significant time and another update is triggered while the previous update is still running, the result of the previous update will be used. When this situation occurs, the controller manager will print a missing update cycle message, informing the user that they need to lower their controller's frequency as the computation is taking longer than initially estimated, as shown in the following example:

```
[ros2_control_node-1] [WARN] [1741626670.311533972] [example_async_controller]: The␣
↪controller missed xx update cycles out of yy total triggers.
```

If the async controller's update method throws an unhandled exception, the controller manager will handle it the same way as the synchronous controllers, deactivating the controller. It will also print an error message, similar to the following:

```
[ros2_control_node-1] [ERROR] [1741629098.352771957] [AsyncFunctionHandler]:␣
↪AsyncFunctionHandler: Exception caught in the async callback thread!
...
[ros2_control_node-1] [ERROR] [1741629098.352874151] [controller_manager]: Caught␣
↪exception of type : St13runtime_error while updating controller
[ros2_control_node-1] [ERROR] [1741629098.352940701] [controller_manager]:␣
```

(continues on next page)

(continued from previous page)

```
↪Deactivating controllers : [example_async_controller] as their update resulted in␣
↪an error!
```

Monitoring and Tuning

ros2_control `controller_interface` has a `ControllerUpdateStats` structure which can be used to monitor the controller update rate and the missed update cycles. The data is published to the `/diagnostics` topic. This can be used to fine tune the controller update rate.

Different Clocks used by Controller Manager

The controller manager internally uses the following two different clocks for a non-simulation setup:

- `RCL_ROS_TIME`: This clock is used mostly in the non-realtime loops.
- `RCL_STEADY_TIME`: This clock is used mostly in the realtime loops for the `read`, `update`, and `write` loops. However, when the controller manager is used in a simulation environment, the `RCL_ROS_TIME` clock is used for triggering the `read`, `update`, and `write` loops.

The `time` argument in the `read` and `write` methods of the hardware components is of type `RCL_STEADY_TIME`, as most of the hardware expects the time to be monotonic and not affected by the system time changes. However, the `time` argument in the `update` method of the controller is of type `RCL_ROS_TIME` as the controller is the one that interacts with other nodes or topics to receive the commands or publish the state. This `time` argument can be used by the controllers to validate the received commands or to publish the state at the correct timestamp. The `period` argument in the `read`, `update` and `write` methods is calculated using the trigger clock of type `RCL_STEADY_TIME` so it is always monotonic.

The reason behind using different clocks is to avoid the issues related to the affect of system time changes in the real-time loops. The `ros2_control_node` now also detects the overruns caused by the system time changes and longer execution times of the controllers and hardware components. The controller manager will print a warning message if the controller or hardware component misses the update cycle due to the system time changes or longer execution times.

Color Output Handling

The helper scripts (`spawner` and `hardware_spawner`) now use an environment-aware `bcolors` class. The color output automatically adapts to the environment:

- `RCUTILS_COLORIZED_OUTPUT=0` -> disables color output
- `RCUTILS_COLORIZED_OUTPUT=1` -> forces color output
- Unset -> automatically detects TTY and enables color only in interactive terminals

2.2.2 Controller Chaining / Cascade Control

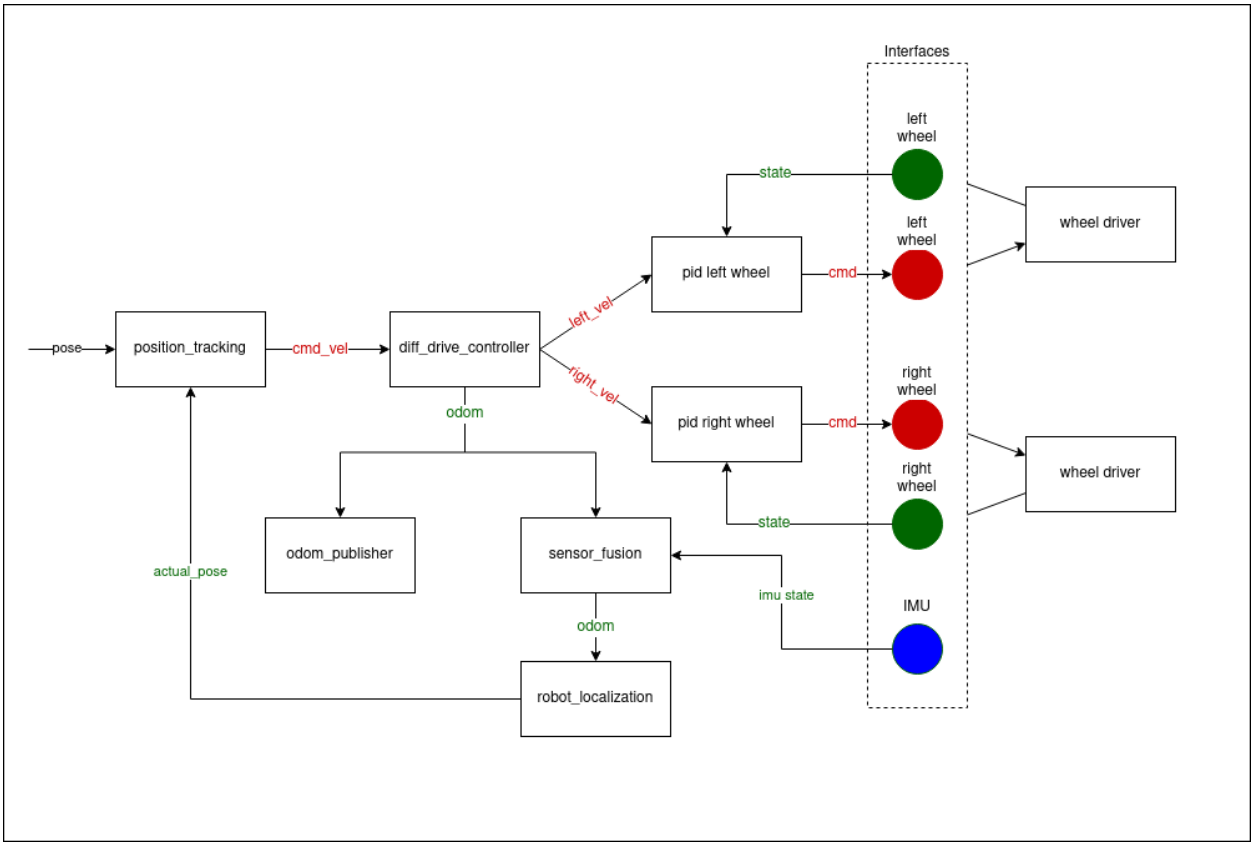
This document proposes a minimal-viable-implementation of serial controller chaining as described in Chaining Controllers design document. Cascade control is a specific type of controller chaining.

Scope of the Document and Background Knowledge

This approach focuses only on serial chaining of controllers and tries to reuse existing mechanisms for it. It focuses on inputs and outputs of a controller and their management in the controller manager. The concept of controller groups will be introduced only for clarity reasons, and its only meaning is that controllers in that group can be updated in arbitrary order. This doesn't mean that the controller groups as described in the controller chaining document will not be introduced and used in the future. Nevertheless, the author is convinced that this would add only unnecessary complexity at this stage, although in the long term they *could* provide clearer structure and interfaces.

Motivation, Purpose and Use

To describe the intent of this document, lets focus on the simple yet sufficient example Example 2 from 'controllers_chaining' design docs:



In this example, we want to chain 'position_tracking' controller with 'diff_drive_controller' and two PID controllers as well as the 'robot_localization' controller. Let's now imagine a use-case where we don't only want to run all those controllers as a group, but also flexibly add preceding steps. This means the following:

1. When a robot is started, we want to check if motor velocity control is working properly and therefore only PID controllers are activated. At this stage we can control the input of PID controller also externally using topics. However, these controllers also provide virtual interfaces, so we can chain them.

2. Then "diff_drive_controller" is activated and attaches itself to the virtual input interfaces of PID controllers. PID controllers also get informed that they are working in chained mode and therefore disable their external interface through subscriber. Now we check if kinematics of differential robot is running properly.
3. Once the 'diff_drive_controller' is activated, it exposes the 'odom' state interfaces that is used by 'odom_publisher' as well as 'sensor_fusion' controllers. The 'odom_publisher' controller is activated and attaches itself to the exported 'odom' state interfaces of 'diff_drive_controller'. The 'sensor_fusion' controller is activated and attaches itself to the exported 'odom' state interfaces of 'diff_drive_controller' along with the 'imu' state interfaces.
4. Once the 'sensor_fusion' controller is activated, it exposes the 'odom' state interfaces that is used by 'robot_localization' controller. The 'robot_localization' controller is activated and attaches itself to the 'odom' state interfaces of 'sensor_fusion' controller. Once activated, the 'robot_localization' controller exposes the 'actual_pose' state interfaces that is used by 'position_tracking' controller.
5. After that, "position_tracking" can be activated and attached to "diff_drive_controller" that disables its external interfaces and to the 'robot_localization' controller which provides the 'actual_pose' state interface.
6. If any of the controllers is deactivated, also all preceding controllers needs to be deactivated.

Note: Controllers that expose the reference interfaces are switched to chained mode only when their reference interfaces are used by other controllers. When their reference interfaces are not used by other controllers, they are switched back to get references from the subscriber. However, the controllers that expose the state interfaces are not triggered to chained mode when their state interfaces are used by other controllers.

Note: This document uses terms *preceding* and *following* controller. These terms refer to such ordering of controllers that controller A *precedes* controller B if A claims (*connects its output to*) B's reference interfaces (*inputs*). In the example diagram at the beginning of this section, 'diff_drive_controller' *precedes* 'pid left wheel' and 'pid right wheel'. Consequently, 'pid left wheel' and 'pid right wheel' are controllers *following* after 'diff_drive_controller'.

Implementation

A Controller Base-Class: ChainableController

A `ChainableController` extends `ControllerInterface` class with `virtual std::vector<hardware_interface::CommandInterface> export_reference_interfaces() = 0` method as well as `virtual std::vector<hardware_interface::StateInterface> export_state_interfaces() = 0` method. This method should be implemented for each controller that **can be preceded** by another controller exporting all the reference/state interfaces. For simplicity reasons, it is assumed for now that controller's all reference interfaces are used by other controller. However, the state interfaces exported by the controller, can be used by multiple controllers at the same time and with the combination they want. Therefore, do not try to implement any exclusive combinations of reference interfaces, but rather write multiple controllers if you need exclusivity.

The `ChainableController` base class implements `void set_chained_mode(bool activate)` that sets an internal flag that a controller is used by another controller (in chained mode) and calls `virtual void on_set_chained_mode(bool activate) = 0` that implements controller's specific actions when chained mode is activated or deactivated, e.g., deactivating subscribers.

As an example, PID controllers export one virtual interface `pid_reference` and stop their subscriber `<controller_name>/pid_reference` when used in chained mode. 'diff_drive_controller' controller exports list of virtual interfaces `<controller_name>/v_x`, `<controller_name>/v_y`, and `<controller_name>/w_z`, and stops subscribers from topics `<controller_name>/cmd_vel` and `<controller_name>/cmd_vel_unstamped`. Its publishers can continue running.

Nomenclature

There are two types of interfaces within the context of `ros2_control`: `CommandInterface` and `StateInterface`.

- The `CommandInterface` is a Read-Write type of interface that can be used to get and set values. Its typical use-case is to set command values to the hardware.
- The `StateInterface` is a Read-Only type of interface that can be used to get values. Its typical use-case is to get actual state values from the hardware.

These interfaces are utilized in different places within the controller in order to have a functional controller or controller chain that commands the hardware.

- The `virtual InterfaceConfiguration command_interface_configuration() const` method defined in the `ControllerInterface` class is used to define the command interfaces used by the controller. These interfaces are used to command the hardware or the exposed reference interfaces from another controller. The `controller_manager` uses this configuration to claim the command interfaces from the `ResourceManager`.
- The `virtual InterfaceConfiguration state_interface_configuration() const` method defined in the `ControllerInterface` class is used to define the state interfaces used by the controller. These interfaces are used to get the actual state values from the hardware or the exposed state interfaces from another controller. The `controller_manager` uses this configuration to claim the state interfaces from the `ResourceManager`.
- The `std::vector<hardware_interface::CommandInterface> export_reference_interfaces()` method defined in the `ChainableController` class is used to define the reference interfaces exposed by the controller. These interfaces are used to command the controller from other controllers.
- The `std::vector<hardware_interface::StateInterface> export_state_interfaces()` method defined in the `ChainableController` class is used to define the state interfaces exposed by the controller. These interfaces are used to get the actual state values from the controller by other controllers.

Inner Resource Management

After configuring a chainable controller, controller manager calls `export_reference_interfaces` and `export_state_interfaces` method and takes ownership over controller's exported reference/state interfaces. This is the same process as done by `ResourceManager` and hardware interfaces. Controller manager maintains "claimed" status of interface in a vector (the same as done in `ResourceManager`).

Activation and Deactivation Chained Controllers

Chained controllers must be activated and deactivated together or in the proper order. This means you must first activate all following controllers to have the preceding one activated. For the deactivation there is the inverse rule - all preceding controllers have to be deactivated before the following controller is deactivated. One can also think of it as an actual chain, you can not add a chain link or break the chain in the middle. The chained controllers can also be activated when parsed as in a single list through the fields `activate_controllers` or `deactivate_controllers` in the `switch_controllers` service provided by the controller_manager. The controller_manager `spawner` can also be used to activate all the controllers of the chain in a single call, by parsing the argument `--activate-as-group`.

Debugging outputs

- The reference interfaces are `unavailable` and `unclaimed`, when the controller exporting them is in inactive state
- The reference interfaces are `available` and `unclaimed`, when the controller exporting them is in an active state but is not in chained mode with any other controller (The controllers gets its references from the subscriber)
- The reference interfaces are `available` and `claimed`, when the controller exporting them is in active state and also in chained mode with other controllers (The controller gets its references from the controllers it is chained with)

Closing remarks

- Maybe addition of the new controller's type `ChainableController` is not necessary. It would also be feasible to add implementation of `export_reference_interfaces()` and `export_state_interfaces()` method into `ControllerInterface` class with default result `interface_configuration_type::NONE`.

2.2.3 Joint Kinematics for ros2_control

This page should give an overview of the joint kinematics in the context of ros2_control. It is intended to give a brief introduction to the topic and to explain the current implementation in ros2_control.

Nomenclature

Degrees of Freedom (DoF)
From wikipedia:

> In physics, the degrees of freedom (DoF) of a mechanical system is the number of independent parameters that define its configuration or state.

Joint
A joint is a connection between two links. In the ROS ecosystem, three types are more typical: Revolute (A hinge joint with position limits), Continuous (A continuous hinge without any position limits) or Prismatic (A sliding joint that moves along the axis).

In general, a joint can be actuated or non-actuated, also called passive. Passive joints are joints that do not have their own actuation mechanism but instead allow movement by external forces or by being passively moved by other joints. A passive joint can have a DoF of one, such as a pendulum, or it can be part of a parallel kinematic mechanism with zero DoF.

Serial Kinematics
Serial kinematics refers to the arrangement of joints in a robotic manipulator where each joint is independent of the others, and the number of joints is equal to the DoF of the kinematic chain.

A typical example is an industrial robot with six revolute joints, having 6-DoF. Each joint can be actuated independently, and the end-effector can be moved to any position and orientation in the workspace.

Kinematic Loops
On the other hand, kinematic loops, also known as closed-loop mechanisms, involve several joints that are connected in a kinematic chain and being actuated together. This means that the joints are coupled and cannot be moved independently: In general, the number of DoFs is smaller than the number of joints. This structure is typical for parallel kinematic mechanisms, where the end-effector is connected to the base by several kinematic chains.

An example is the four-bar linkage, which consists of four links and four joints. It can have one or two actuators and consequently one or two DoFs, despite having four joints. Furthermore, we can say that we have one (two) actuated joint and three (two) passive joints, which must satisfy the kinematic constraints of the mechanism.

URDF

URDF is the default format to describe robot kinematics in ROS. However, only serial kinematic chains are supported, except for the so-called mimic joints. See the URDF specification for more details.

Mimic joints can be defined in the following way in the URDF

```
<joint name="right_finger_joint" type="prismatic">
  <axis xyz="0 1 0"/>
  <origin xyz="0.0 -0.48 1" rpy="0.0 0.0 0.0"/>
  <parent link="base"/>
  <child link="finger_right"/>
  <limit effort="1000.0" lower="0" upper="0.38" velocity="10"/>
</joint>
<joint name="left_finger_joint" type="prismatic">
  <mimic joint="right_finger_joint" multiplier="1" offset="0"/>
  <axis xyz="0 1 0"/>
  <origin xyz="0.0 0.48 1" rpy="0.0 0.0 3.1415926535"/>
  <parent link="base"/>
  <child link="finger_left"/>
  <limit effort="1000.0" lower="0" upper="0.38" velocity="10"/>
</joint>
```

Mimic joints are an abstraction of the real world. For example, they can be used to describe

- simple closed-loop kinematics with linear dependencies of the joint positions and velocities
- links connected with belts, like belt and pulley systems or telescope arms
- a simplified model of passive joints, e.g. a pendulum at the end-effector always pointing downwards
- abstract complex groups of actuated joints, where several joints are directly controlled by low-level control loops and move synchronously. Without giving a real-world example, this could be several motors with their individual power electronics but commanded with the same setpoint.

Mimic joints defined in the URDF are parsed from the resource manager and stored in a class variable of type `HardwareInfo`, which can be accessed by the hardware components. The mimic joints must not have command interfaces but can have state interfaces.

```
<ros2_control>
  <joint name="right_finger_joint">
    <command_interface name="effort"/>
    <state_interface name="position"/>
    <state_interface name="velocity"/>
    <state_interface name="effort"/>
  </joint>
  <joint name="left_finger_joint">
    <state_interface name="position"/>
    <state_interface name="velocity"/>
    <state_interface name="effort"/>
  </joint>
</ros2_control>
```

From the officially released packages, the following packages are already using this information:

- *mock_components (generic system)*
- *gz_ros2_control*

As the URDF specifies only the kinematics, the mimic tag has to be independent of the hardware interface type used in ros2_control. This means that we interpret this info in the following way:

- **position = multiplier * other_joint_position + offset**
- **velocity = multiplier * other_joint_velocity**

If someone wants to deactivate the mimic joint behavior for whatever reason without changing the URDF, it can be done by setting the attribute `mimic=false` of the joint tag in the `<ros2_control>` section of the XML.

```
<joint name="left_finger_joint" mimic="false">
  <state_interface name="position"/>
  <state_interface name="velocity"/>
  <state_interface name="effort"/>
</joint>
```

Transmission Interface

Mechanical transmissions transform effort/flow variables such that their product (power) remains constant. Effort variables for linear and rotational domains are force and torque; while the flow variables are respectively linear velocity and angular velocity.

In robotics it is customary to place transmissions between actuators and joints. This interface adheres to this naming to identify the input and output spaces of the transformation. The provided interfaces allow bidirectional mappings between actuator and joint spaces for effort, velocity and position. Position is not a power variable, but the mappings can be implemented using the velocity map plus an integration constant representing the offset between actuator and joint zeros.

The `transmission_interface` provides a base class and some implementations for plugins, which can be integrated and loaded by custom hardware components. They are not automatically loaded by any hardware component or the gazebo plugins, each hardware component is responsible for loading the appropriate transmission interface to map the actuator readings to joint readings.

Currently the following implementations are available:

- `SimpleTransmission`: A simple transmission with a constant reduction ratio and no additional dynamics.
- `DifferentialTransmission`: A differential transmission with two actuators and two joints.
- `FourBarLinkageTransmission`: A four-bar-linkage transmission with two actuators and two joints.

For more information, see *example_8* or the transmission_interface documentation.

Simulating Closed-Loop Kinematic Chains

Depending on the simulation plugin, different approaches can be used to simulate closed-loop kinematic chains. The following list gives an overview of the available simulation plugins and their capabilities:

gazebo_ros2_control:

- mimic joints
- closed-loop kinematics are supported with `<gazebo>` tags in the URDF, see, e.g., here.

gz_ros2_control:

- mimic joints

- closed-loop kinematics are not directly supported yet, but can be implemented by using a `DetachableJoint` via custom plugins. Follow this issue for updates on this topic.

2.2.4 Joint Limiting for ros2_control

ros2_control provides several mechanisms to handle joint limits for different hardware interfaces.

Enable Joint Limits

There exists several ways in controlling the joint limits handling in ros2_control:

- globally for all interfaces of all hardware components via the `ros2_control_node` parameter `enforce_command_limits`, for details see *here*.
- for all interfaces of a joint

```
<joint name="joint1">
  ...
  <limits enable="false"/>
  ...
</joint>
```

- for a single interface of a joint

```
<joint name="joint1">
  ...
  <command_interface name="position">
    <limits enable="false"/>
  </command_interface>
  ...
</joint>
```

If joint limits are active for a specific interface, the controller_manger will print a similar message on startup:

```
[controller_manager]: Using JointLimiter for joint 'joint1' in hardware
↪'RRBot' : '  has position limits: true [-1, 1]
[ros2_control_node-1]   has velocity limits: true [1]
[ros2_control_node-1]   has acceleration limits: false [nan]
[ros2_control_node-1]   has deceleration limits: false [nan]
[ros2_control_node-1]   has jerk limits: false [nan]
[ros2_control_node-1]   has effort limits: true [100]
[ros2_control_node-1]   angle wraparound: true'
```

Configuration of Limits

tba

Description of the Limiter Algorithms

tba

2.2.5 Hardware Components

Hardware components represent abstraction of physical hardware in ros2_control framework. There are three types of hardware Actuator, Sensor and System. For details on each type check *Hardware Components* description.

Guidelines and Best Practices

`ros2_control` hardware interface types

The `ros2_control` framework provides a set of hardware interface types that can be used to implement a hardware component for a specific robot or device. The following sections describe the different hardware interface types and their usage.

Overview

Hardware in ros2_control is described as URDF and internally parsed and encapsulated as `HardwareInfo`. The definition can be found in the ros2_control repository. You can check the structs defined there to see what attributes are available for each of the xml tags. A generic example which shows the structure is provided below. More specific examples can be found in the Example part below.

```
<ros2_control name="Name_of_the_hardware" type="system">
  <hardware>
    <plugin>library_name/ClassName</plugin>
    <!-- added to hardware_parameters -->
    <param name="example_param">value</param>
  </hardware>
  <joint name="name_of_the_component">
    <!-- `data_type` argument is optional (defaults to double). -->
    <command_interface name="interface_name" data_type="double">
      <!-- All of them are optional. -->
      <param name="min">-1</param>
      <param name="max">1</param>
      <param name="initial_value">0.0</param>
       <!-- Optional. Added to the key/value storage parameters -->
      <param name="own_param_1">some_value</param>
      <param name="own_param_2">other_value</param>
    </command_interface>
    <!-- Short form to define StateInterface. Can be extended like CommandInterface. -
↪->
    <state_interface name="position"/>
  </joint>
</ros2_control>
```

Joints

<joint>-tag groups the interfaces associated with the joints of physical robots and actuators. They have command and state interfaces to set the goal values for hardware and read its current state.

All joints defined in the <ros2_control>-tag have to be present in the URDF received *by the controller manager*.

State interfaces of joints can be published as a ROS topic by means of the *joint_state_broadcaster*

Sensors

<sensor>-tag groups multiple state interfaces describing, e.g., internal states of hardware.

Depending on the type of sensor, there exist a couple of specific semantic components with broadcasters shipped with ros2_controllers, see details in the *semantic_components*.

GPIOs

The <gpio>-tag is used for describing input and output ports of a robotic device that cannot be associated with any joint or sensor. Parsing of <gpio>-tag is similar to this of a <joint>-tag having command and state interfaces. The tag must have at least one <command>- or <state>-tag as a child.

The keyword "gpio" is chosen for its generality. Although strictly used for digital signals, it describes any electrical analog, digital signal, or physical value.

The <gpio> tag can be used as a child of all three types of hardware components, i.e., system, sensor, or actuator.

Because ports implemented as <gpio>-tag are typically very application-specific, there exists no generic publisher within the ros2_control framework. A custom gpio-controller has to be implemented for each application. As an example, see *the GPIO controller example* as part of the demo repository.

Hardware Groups

Hardware Component Groups serve as a critical organizational mechanism within complex systems, facilitating error handling and fault tolerance. By grouping related hardware components together, such as actuators within a manipulator, users can establish a unified framework for error detection and response.

Hardware Component Groups play a vital role in propagating errors across interconnected hardware components. For instance, in a manipulator system, grouping actuators together allows for error propagation. If one actuator fails within the group, the error can propagate to the other actuators, signaling a potential issue across the system. By default, the actuator errors are isolated to their own hardware component, allowing the rest to continue operation unaffected. In the provided ros2_control configuration, the <group> tag within each <ros2_control> block signifies the grouping of hardware components, enabling error propagation mechanisms within the system.

Data Types

By default, command and state interfaces use the `double` data type. However, other data types can be specified using the optional `data_type` argument in the `<command_interface>` and `<state_interface>` tags. The following data types are supported, with their default initial value if not specified:

- double (default): NaN
- float32: NaN
- bool: false
- uint8: 255
- int8: 127
- uint16: 65535
- int16: 32767
- uint32: 4294967295
- int32: 2147483647

Examples

The following examples show how to use the different hardware interface types in a `ros2_control` URDF. They can be combined together within the different hardware component types (system, actuator, sensor) (*see detailed documentation*) as follows

1. Robot with multiple GPIO interfaces
   - RRBot System
   - Digital: 4 inputs and 2 outputs (bool)
   - Analog: 2 inputs and 1 output (uint16)
   - Vacuum valve at the flange (bool)

```
<ros2_control name="RRBotSystemMutipleGPIOs" type="system">
  <hardware>
    <plugin>ros2_control_demo_hardware/RRBotSystemPositionOnlyHardware</
↪plugin>
    <param name="example_param_hw_start_duration_sec">2.0</param>
    <param name="example_param_hw_stop_duration_sec">3.0</param>
    <param name="example_param_hw_slowdown">2.0</param>
  </hardware>
  <joint name="joint1">
    <command_interface name="position">
      <param name="min">-1</param>
      <param name="max">1</param>
    </command_interface>
    <state_interface name="position"/>
  </joint>
  <joint name="joint2">
    <command_interface name="position">
      <param name="min">-1</param>
      <param name="max">1</param>
    </command_interface>
    <state_interface name="position"/>
```

(continues on next page)

(continued from previous page)

```
  </joint>
  <gpio name="flange_digital_IOs">
    <command_interface name="digital_output1" data_type="bool"/>
    <state_interface name="digital_output1" data_type="bool"/>    <!--␣
↪Needed to know current state of the output -->
    <command_interface name="digital_output2" data_type="bool"/>
    <state_interface name="digital_output2" data_type="bool"/>
    <state_interface name="digital_input1" data_type="bool"/>
    <state_interface name="digital_input2" data_type="bool"/>
  </gpio>
  <gpio name="flange_analog_IOs">
    <command_interface name="analog_output1" data_type="uint16"/>
    <state_interface name="analog_output1" data_type="uint16">    <!--␣
↪Needed to know current state of the output -->
      <param name="initial_value">3.1</param>  <!-- Optional initial value␣
↪for mock_hardware -->
    </state_interface>
    <state_interface name="analog_input1" data_type="uint16"/>
    <state_interface name="analog_input2" data_type="uint16"/>
  </gpio>
  <gpio name="flange_vacuum">
    <command_interface name="vacuum" data_type="bool"/>
    <state_interface name="vacuum" data_type="bool"/>    <!-- Needed to know␣
↪current state of the output -->
  </gpio>
</ros2_control>
```

2. Gripper with electrical and suction grasping possibilities
   - Multimodal gripper
   - 1-DoF parallel gripper
   - suction on/off

```
<ros2_control name="MultimodalGripper" type="actuator">
  <hardware>
    <plugin>ros2_control_demo_hardware/MultimodalGripper</plugin>
  </hardware>
  <joint name="parallel_fingers">
    <command_interface name="position">
      <param name="min">0</param>
      <param name="max">100</param>
    </command_interface>
    <state_interface name="position"/>
  </joint>
  <gpio name="suction">
    <command_interface name="suction"/>
    <state_interface name="suction"/>    <!-- Needed to know current state␣
↪of the output -->
  </gpio>
</ros2_control>
```

3. Force-Torque-Sensor with temperature feedback and adjustable calibration
   - 2D FTS
   - Temperature feedback in °C
   - Choice between 3 calibration matrices, i.e., calibration ranges

```
<ros2_control name="RRBotForceTorqueSensor2D" type="sensor">
  <hardware>
    <plugin>ros2_control_demo_hardware/ForceTorqueSensor2DHardware</plugin>
    <param name="example_param_read_for_sec">0.43</param>
  </hardware>
  <sensor name="tcp_fts_sensor">
    <state_interface name="fx"/>
    <state_interface name="tz"/>
    <param name="frame_id">kuka_tcp</param>
    <param name="fx_range">100</param>
    <param name="tz_range">100</param>
  </sensor>
  <sensor name="temp_feedback">
    <state_interface name="temperature"/>
  </sensor>
  <gpio name="calibration">
    <command_interface name="calibration_matrix_nr"/>
    <state_interface name="calibration_matrix_nr"/>
  </gpio>
</ros2_control>
```

4. Robot with multiple hardware components belonging to same group : `Group1`

   - RRBot System 1 and 2
   - Digital: Total 4 inputs and 2 outputs
   - Analog: Total 2 inputs and 1 output
   - Vacuum valve at the flange (on/off)
   - Group: Group1

```
<ros2_control name="RRBotSystem1" type="system">
  <hardware>
    <plugin>ros2_control_demo_hardware/RRBotSystemPositionOnlyHardware</
↪plugin>
    <group>Group1</group>
    <param name="example_param_hw_start_duration_sec">2.0</param>
    <param name="example_param_hw_stop_duration_sec">3.0</param>
    <param name="example_param_hw_slowdown">2.0</param>
  </hardware>
  <joint name="joint1">
    <command_interface name="position">
      <param name="min">-1</param>
      <param name="max">1</param>
    </command_interface>
    <state_interface name="position"/>
  </joint>
  <gpio name="flange_analog_IOs">
    <command_interface name="analog_output1"/>
    <state_interface name="analog_output1">    <!-- Needed to know current␣
↪state of the output -->
      <param name="initial_value">3.1</param>  <!-- Optional initial value␣
↪for mock_hardware -->
    </state_interface>
    <state_interface name="analog_input1"/>
    <state_interface name="analog_input2"/>
  </gpio>
```

(continues on next page)

(continued from previous page)

```
  <gpio name="flange_vacuum">
    <command_interface name="vacuum"/>
    <state_interface name="vacuum"/>    <!-- Needed to know current state of␣
↪the output -->
  </gpio>
</ros2_control>
<ros2_control name="RRBotSystem2" type="system">
  <hardware>
    <plugin>ros2_control_demo_hardware/RRBotSystemPositionOnlyHardware</
↪plugin>
    <group>Group1</group>
    <param name="example_param_hw_start_duration_sec">2.0</param>
    <param name="example_param_hw_stop_duration_sec">3.0</param>
    <param name="example_param_hw_slowdown">2.0</param>
  </hardware>
  <joint name="joint2">
    <command_interface name="position">
      <param name="min">-1</param>
      <param name="max">1</param>
    </command_interface>
    <state_interface name="position"/>
  </joint>
  <gpio name="flange_digital_IOs">
    <command_interface name="digital_output1"/>
    <state_interface name="digital_output1"/>    <!-- Needed to know current␣
↪state of the output -->
    <command_interface name="digital_output2"/>
    <state_interface name="digital_output2"/>
    <state_interface name="digital_input1"/>
    <state_interface name="digital_input2"/>
  </gpio>
</ros2_control>
```

Writing a Hardware Component

In ros2_control hardware system components are libraries, dynamically loaded by the controller manager using the pluginlib interface. The following is a step-by-step guide to create source files, basic tests, and compile rules for a new hardware interface.

1. **Preparing package**

   If the package for the hardware interface does not exist, then create it first. The package should have `ament_cmake` as a build type. The easiest way is to search online for the most recent manual. A helpful command to support this process is `ros2 pkg create`. Use the `--help` flag for more information on how to use it. There is also an option to create library source files and compile rules to help you in the following steps.

2. **Preparing source files**

   After creating the package, you should have at least `CMakeLists.txt` and `package.xml` files in it. Create also `include/<PACKAGE_NAME>/` and `src` folders if they do not already exist. In `include/<PACKAGE_NAME>/` folder add `<robot_hardware_interface_name>.hpp` and `<robot_hardware_interface_name>.cpp` in the `src` folder.

3. **Adding declarations into header file (.hpp)**

   1. Take care that you use header guards. ROS2-style is using `#ifndef` and `#define` preprocessor directives. (For more information on this, a search engine is your friend :)).

2. Include `"hardware_interface/$interface_type$_interface.hpp"`. `$interface_type$` can be `Actuator`, `Sensor` or `System` depending on the type of hardware you are using. for more details about each type check *Hardware Components description*.
3. Define a unique namespace for your hardware_interface. This is usually the package name written in `snake_case`.
4. Define the class of the hardware_interface, extending `$InterfaceType$Interface`, e.g., .. code:: c++ class HardwareInterfaceName : public hardware_interface::$InterfaceType$Interface
5. Add a constructor without parameters and the following public methods implementing `LifecycleNodeInterface`: `on_configure`, `on_cleanup`, `on_shutdown`, `on_activate`, `on_deactivate`, `on_error`; and overriding `$InterfaceType$Interface` definition: `on_init`, `export_state_interfaces`, `export_command_interfaces`, `prepare_command_mode_switch` (optional), `perform_command_mode_switch` (optional), `read`, `write`.

   For further explanation of hardware-lifecycle check the pull request and for exact definitions of methods check the `"hardware_interface/$interface_type$_interface.hpp"` header or doxygen documentation for *Actuator*, *Sensor* or *System*.

4. **Adding definitions into source file (.cpp)**
   1. Include the header file of your hardware interface and add a namespace definition to simplify further development.
   2. Implement `on_init` method. Here, you should initialize all member variables and process the parameters from the `info` argument. In the first line usually the parents `on_init` is called to process standard values, like name. This is done using: `hardware_interface::(Actuator|Sensor|System)Interface::on_init(info)`. If all required parameters are set and valid and everything works fine return `CallbackReturn::SUCCESS` or `return CallbackReturn::ERROR` otherwise.
      1. **(Optional) Adding Publisher, Services, etc.**

         A common requirement for a hardware component is to publish status or diagnostic information without interfering with the real-time control loop.

         This allows you to add any standard ROS 2 component (publishers, subscribers, services, timers) to your hardware interface without compromising real-time performance. There are three primary ways to achieve this.

         Method 1: Using the Framework-Managed Publisher (Recommended & Simplest for HardwareStatus Messages)

         Refer *Framework Managed Publisher*

         Method 2: Using the Framework-Managed Node (Recommended & Simplest for Custom Messages)

         The framework internally creates a dedicated ROS 2 node for each hardware component. Your hardware plugin can then get a handle to this node and use it.

         1. **Access and using the Default Node**: You can get a `shared_ptr` to the node by calling the `get_node()` method and use it just like any other `rclcpp::Node::SharedPtr` to create publishers, timers, etc.

```
// Continuing inside on_configure()
if (get_node())
{
   my_publisher_ = get_node()->create_publisher<std_
```

(continues on next page)

(continued from previous page)

```
↪msgs::msg::String>("~/status", 10);

  using namespace std::chrono_literals;
  my_timer_ = get_node()->create_wall_timer(1s, [this]() {
     std_msgs::msg::String msg;
     msg.data = "Hardware status update!";
     my_publisher_->publish(msg);
  });
}
```

Method 3: Using the Executor from `HardwareComponentInterfaceParams`

For more advanced use cases where you need direct control over node creation, the `on_init` method can be configured to receive a `HardwareComponentInterfaceParams` struct. This struct contains a `weak_ptr` to the `ControllerManager`'s executor.

1. **Update ``on_init`` Signature**: First, your hardware interface must override the `on_init` version that takes `HardwareComponentInterfaceParams`.

```
// In your <robot_hardware_interface_name>.hpp
hardware_interface::CallbackReturn on_init(
const hardware_interface::HardwareComponentInterfaceParams &␣
↪params) override;
```

2. **Lock and Use the Executor**: Inside `on_init`, you must safely "lock" the `weak_ptr` to get a usable `shared_ptr`. You can then create your own node and add it to the executor.

```
// In your <robot_hardware_interface_name>.cpp, inside on_
↪init(params)
if (auto locked_executor = params.executor.lock())
{
   my_custom_node_ = std::make_shared<rclcpp::Node>("my_custom_
↪node");
   locked_executor->add_node(my_custom_node_->get_node_base_
↪interface());
   // ... create publishers/timers on my_custom_node_ ...
}
```

For a complete, working implementation that uses the framework-managed node to publish diagnostic messages, see the demo in *Example 17*.

3. Write the `on_configure` method where you usually setup the communication to the hardware and set everything up so that the hardware can be activated.
4. Implement `on_cleanup` method, which does the opposite of `on_configure`.
5. `Command-/StateInterfaces` are now created and exported automatically by the framework via the `on_export_command_interfaces()` or `on_export_state_interfaces()` methods based on the interfaces defined in the `ros2_control` XML-tag, which gets parsed and the `InterfaceDescription` is created accordingly (check the hardware_info.hpp).
   - To access the automatically created `Command-/StateInterfaces` we provide the `std::unordered_map<std::string, InterfaceDescription>`, where the string is the fully qualified name of the interface and the `InterfaceDescription` is the configuration of the interface. The `std::unordered_map<>` are divided into `type_state_interfaces_` and `type_command_interfaces_` where type can be: `joint`, `sensor`, `gpio` and `unlisted`. E.g. the `CommandInterfaces` for all

joints can be found in the `joint_command_interfaces_` map. The `unlisted` includes all interfaces not listed in the `ros2_control` XML-tag but were created by overriding the `export_unlisted_command_interface_descriptions()` or `export_unlisted_state_interface_descriptions()` function by creating some custom `Command-/StateInterfaces`.

- For the `Sensor`-type hardware interface there is no `export_command_interfaces` method.
- As a reminder, the full interface names have structure `<joint_name>/<interface_type>`.

6. (optional) If you want some unlisted `Command-/StateInterfaces` not included in the `ros2_control` XML-tag you can follow those steps:

   1. Override the `virtual std::vector<hardware_interface::InterfaceDescription> export_unlisted_command_interface_descriptions()` or `virtual std::vector<hardware_interface::InterfaceDescription> export_unlisted_state_interface_descriptions()`
   2. Create the InterfaceDescription for each of the interfaces you want to create in the override `export_unlisted_command_interface_descriptions()` or `export_unlisted_state_interface_descriptions()` function, add it to a vector and return the vector:

      ```
      std::vector<hardware_interface::InterfaceDescription> my_unlisted_
      ↪interfaces;

      InterfaceInfo unlisted_interface;
      unlisted_interface.name = "some_unlisted_interface";
      unlisted_interface.min = "-5.0";
      unlisted_interface.data_type = "double";
      my_unlisted_interfaces.push_back(InterfaceDescription(info_.name,␣
      ↪unlisted_interface));

      return my_unlisted_interfaces;
      ```

   3. The unlisted interface will then be stored in either the `unlisted_command_interfaces_` or `unlisted_state_interfaces_` map depending in which function they are created.
   4. You can access it like any other interface with the `get_state(name)`, `set_state(name, value)`, `get_command(name)` or `set_command(name, value)`. E.g. `get_state("some_unlisted_interface")`.

7. (optional) In case the default implementation (`on_export_command_interfaces()` or `on_export_state_interfaces()`) for exporting the `Command-/StateInterfaces` is not enough you can override them. You should however consider the following things:

   - If you want to have unlisted interfaces available you need to call the `export_unlisted_command_interface_descriptions()` or `export_unlisted_state_interface_descriptions()` and add them to the `unlisted_command_interfaces_` or `unlisted_state_interfaces_`.
   - Don't forget to store the created `Command-/StateInterfaces` internally as you only return shared_ptrs and the resource manager will not provide access to the created `Command-/StateInterfaces` for the hardware. So you must take care of storing them yourself.
   - Names must be unique!

8. (optional) For *Actuator* and *System* types of hardware interface implement `prepare_command_mode_switch` and `perform_command_mode_switch` if your hardware accepts multiple control modes.

9. Implement the `on_activate` method where hardware "power" is enabled.
10. Implement the `on_deactivate` method, which does the opposite of `on_activate`.
11. Implement `on_shutdown` method where hardware is shutdown gracefully.
12. Implement `on_error` method where different errors from all states are handled.
13. Implement the `read` method getting the states from the hardware and storing them to internal variables defined in `export_state_interfaces`.
14. Implement `write` method that commands the hardware based on the values stored in internal variables defined in `export_command_interfaces`.
15. (optional) **Framework Managed Publisher**

    Implement `init_hardware_status_message` and `update_hardware_status_message` methods to publish the framework-supported hardware status reporting through `control_msgs/msg/HardwareStatus` messages:

    - **`init_hardware_status_message`**: This non-realtime method is called once during initialization. You must override it to define the **static structure** of your status message. This includes setting the `hardware_id`, resizing the `hardware_device_states` vector, and for each device, resizing its specific status vectors (e.g., `generic_hardware_status`, `canopen_states`) and populating static fields like `device_id` and interface `name`. Pre-allocating the message structure here is crucial for real-time safety.

```
// In your <robot_hardware_interface_name>.hpp
hardware_interface::CallbackReturn init_hardware_status_message(
control_msgs::msg::HardwareStatus & msg_template) override;

// In your <robot_hardware_interface_name>.cpp
hardware_interface::CallbackReturn MyHardware::init_hardware_status_
↪message(
control_msgs::msg::HardwareStatus & msg)
{
   msg.hardware_id = get_hardware_info().name;
   msg.hardware_device_states.resize(get_hardware_info().joints.
↪size());

   for (size_t i = 0; i < get_hardware_info().joints.size(); ++i)
   {
      msg.hardware_device_states[i].device_id = get_hardware_info().
↪joints[i].name;
      // This example uses one generic status per joint
      msg.hardware_device_states[i].generic_hardware_status.resize(1);
   }
   return hardware_interface::CallbackReturn::SUCCESS;
}
```

    - **`update_hardware_status_message`**: This real-time safe method is called from the framework's timer callback. You must override it to **fill in the dynamic values** of the pre-structured message. This typically involves copying your internal state variables (updated in your *read()* method) into the fields of the message. This method must be fast and non-allocating.

```
// In your <robot_hardware_interface_name>.hpp
hardware_interface::return_type update_hardware_status_message(
control_msgs::msg::HardwareStatus & msg) override;

// In your <robot_hardware_interface_name>.cpp
```

(continues on next page)

(continued from previous page)

```
hardware_interface::return_type MyHardware::update_hardware_status_
↪message(
control_msgs::msg::HardwareStatus & msg)
{
   for (size_t i = 0; i < get_hardware_info().joints.size(); ++i)
   {
      auto & generic_status = msg.hardware_device_states[i].generic_
↪hardware_status;
      // Example: Map internal state to a standard status field
      if (std::abs(hw_positions_[i]) > joint_limits_[i].max_position)
      {
         generic_status.health_status = control_
↪msgs::msg::GenericState::HEALTH_ERROR;
      }
      else
      {
         generic_status.health_status = control_
↪msgs::msg::GenericState::HEALTH_OK;
      }
   }
   return hardware_interface::return_type::OK;
}
```

- **Enable in URDF**: Finally, to activate the publisher, add the `status_publish_rate` parameter to your `<hardware>` tag in the URDF. Setting it to 0.0 disables the feature.

```
<ros2_control name="MyHardware" type="system">
<hardware>
   <plugin>my_package/MyHardware</plugin>
   <param name="status_publish_rate">20.0</param>
</hardware>
...
</ros2_control>
```

For a complete, working implementation that uses the framework-managed node to publish diagnostic messages, see the demo in *Example 17*.

16. IMPORTANT: At the end of your file after the namespace is closed, add the `PLUGINLIB_EXPORT_CLASS` macro.

    For this you will need to include the `"pluginlib/class_list_macros.hpp"` header. As first parameters you should provide exact hardware interface class, e.g., `<my_hardware_interface_package>::<RobotHardwareInterfaceName>`, and as second the base class, i.e., `hardware_interface::(Actuator|Sensor|System)Interface`.

5. **Writing export definition for pluginlib**

   1. Create the `<my_hardware_interface_package>.xml` file in the package and add a definition of the library and hardware interface's class which has to be visible for the pluginlib. The easiest way to do that is to check definition for mock components in the *hardware_interface mock_components* section.

   2. Usually, the plugin name is defined by the package (namespace) and the class name, e.g., `<my_hardware_interface_package>/<RobotHardwareInterfaceName>`. This name defines the hardware interface's type when the resource manager searches for it. The other two parameters have to correspond to the definition done in the macro at the bottom of the `<robot_hardware_interface_name>.cpp` file.

6. **Writing a simple test to check if the controller can be found and loaded**

1. Create the folder `test` in your package, if it does not exist already, and add a file named `test_load_<robot_hardware_interface_name>.cpp`.
2. You can copy the `load_generic_system_2dof` content defined in the test_generic_system.cpp package.
3. Change the name of the copied test and in the last line, where hardware interface type is specified put the name defined in `<my_hardware_interface_package>.xml` file, e.g., `<my_hardware_interface_package>/<RobotHardwareInterfaceName>`.

7. **Add compile directives into ``CMakeLists.txt`` file**
    1. Under the line `find_package(ament_cmake REQUIRED)` add further dependencies. Those are at least: `hardware_interface`, `pluginlib`, `rclcpp` and `rclcpp_lifecycle`.
    2. Add a compile directive for a shared library providing the `<robot_hardware_interface_name>.cpp` file as the source.
    3. Add targeted include directories for the library. This is usually only `include`.
    4. Add ament dependencies needed by the library. You should add at least those listed under 1.
    5. Export for pluginlib description file using the following command: .. code:: cmake

        pluginlib_export_plugin_description_file(hardware_interface <my_hardware_interface_package>.xml)
    6. Add install directives for targets and include directory.
    7. In the test section add the following dependencies: `ament_cmake_gmock`, `hardware_interface`.
    8. Add compile definitions for the tests using the `ament_add_gmock` directive. For details, see how it is done for mock hardware in the ros2_control package.
    9. (optional) Add your hardware interface`s library into `ament_export_libraries` before `ament_package()`.
8. **Add dependencies into ``package.xml`` file**
    1. Add at least the following packages into `<depend>` tag: `hardware_interface`, `pluginlib`, `rclcpp`, and `rclcpp_lifecycle`.
    2. Add at least the following package into `<test_depend>` tag: `ament_add_gmock` and `hardware_interface`.
9. **Compiling and testing the hardware component**

1. Now everything is ready to compile the hardware component using the `colcon build <my_hardware_interface_package>` command. Remember to go into the root of your workspace before executing this command.
2. If compilation was successful, source the `setup.bash` file from the install folder and execute `colcon test <my_hardware_interface_package>` to check if the new controller can be found through `pluginlib` library and be loaded by the controller manager.

That's it! Enjoy writing great controllers!

Useful External References

- Templates and scripts for generating controllers shell

> **Note:** The script is currently only recommended to use with Foxy and Humble, not compatible with the API from Jazzy and onwards.

Different update rates for Hardware Components

The `ros2_control` framework allows to run different hardware components at different update rates. This is useful when some of the hardware components needs to run at a different frequency than the traditional control loop frequency which is same as the one of the `controller_manager`. It is very typical to have different components with different frequencies in a robotic system with different sensors or different components using different communication protocols. This is useful when you have a hardware component that needs to run at a higher rate than the rest of the components. For example, you might have a sensor that needs to be read at 1000Hz, while the rest of the components can run at 500Hz, given that the control loop frequency of the `controller_manager` is higher than 1000Hz. The read/write rate can be defined easily by adding the parameter `rw_rate` to the `ros2_control` tag of the hardware component.

Examples

The following examples show how to use the different hardware interface types with different update frequencies in a `ros2_control` URDF. They can be combined together within the different hardware component types (system, actuator, sensor) (*see detailed documentation*) as follows

For a RRBot with Multimodal gripper and external sensor running at different rates:

```
<ros2_control name="RRBotSystemMutipleGPIOs" type="system" rw_rate="500">
  <hardware>
    <plugin>ros2_control_demo_hardware/RRBotSystemPositionOnlyHardware</plugin>
    <param name="example_param_hw_start_duration_sec">2.0</param>
    <param name="example_param_hw_stop_duration_sec">3.0</param>
    <param name="example_param_hw_slowdown">2.0</param>
  </hardware>
  <joint name="joint1">
    <command_interface name="position">
      <param name="min">-1</param>
      <param name="max">1</param>
    </command_interface>
    <state_interface name="position"/>
  </joint>
  <joint name="joint2">
    <command_interface name="position">
      <param name="min">-1</param>
      <param name="max">1</param>
    </command_interface>
    <state_interface name="position"/>
  </joint>
  <gpio name="flange_digital_IOs">
    <command_interface name="digital_output1"/>
    <state_interface name="digital_output1"/>    <!-- Needed to know current state of␣
↪the output -->
    <command_interface name="digital_output2"/>
```

(continues on next page)

(continued from previous page)

```
    <state_interface name="digital_output2"/>
    <state_interface name="digital_input1"/>
    <state_interface name="digital_input2"/>
  </gpio>
</ros2_control>
<ros2_control name="MultimodalGripper" type="actuator" rw_rate="200">
  <hardware>
    <plugin>ros2_control_demo_hardware/MultimodalGripper</plugin>
  </hardware>
  <joint name="parallel_fingers">
    <command_interface name="position">
      <param name="min">0</param>
      <param name="max">100</param>
    </command_interface>
    <state_interface name="position"/>
  </joint>
  <gpio name="suction">
    <command_interface name="suction"/>
    <state_interface name="suction"/>    <!-- Needed to know current state of the␣
↪output -->
  </gpio>
</ros2_control>
<ros2_control name="RRBotForceTorqueSensor2D" type="sensor" rw_rate="250">
  <hardware>
    <plugin>ros2_control_demo_hardware/ForceTorqueSensor2DHardware</plugin>
    <param name="example_param_read_for_sec">0.43</param>
  </hardware>
  <sensor name="tcp_fts_sensor">
    <state_interface name="fx"/>
    <state_interface name="tz"/>
    <param name="frame_id">kuka_tcp</param>
    <param name="fx_range">100</param>
    <param name="tz_range">100</param>
  </sensor>
  <sensor name="temp_feedback">
    <state_interface name="temperature"/>
  </sensor>
  <gpio name="calibration">
    <command_interface name="calibration_matrix_nr"/>
    <state_interface name="calibration_matrix_nr"/>
  </gpio>
</ros2_control>
```

In the above example, the system hardware component that controls the joints of the RRBot is running at 500 Hz, the multimodal gripper is running at 200 Hz and the force torque sensor is running at 250 Hz.

Note: In the above example, the `rw_rate` parameter is set to 500 Hz, 200 Hz and 250 Hz for the system, actuator and sensor hardware components respectively. This parameter is optional and if not set, the default value of 0 will be used which means that the hardware component will run at the same rate as the `controller_manager`. However, if the specified rate is higher than the `controller_manager` rate, the hardware component will then run at the rate of the `controller_manager`.

Running Hardware Components Asynchronously

The `ros2_control` framework allows to run hardware components asynchronously. This is useful when some of the hardware components need to run in a separate thread or executor. For example, a sensor takes longer to read data that affects the periodicity of the `controller_manager` control loop. In this case, the sensor can be run in a separate thread or executor to avoid blocking the control loop.

Parameters

The following parameters can be set in the `ros2_control` hardware configuration to run the hardware component asynchronously:

- `is_async`: (optional) If set to `true`, the hardware component will run asynchronously. Default is `false`.

Under the `ros2_control` tag, a `properties` tag can be added to specify the following parameters of the asynchronous hardware component:

- `thread_priority`: (optional) The priority of the thread that runs the hardware component. The priority is an integer value between 0 and 99. The default value is 50.
- `affinity`: (optional) The CPU affinity of the thread that runs the hardware component. The affinity is a list of CPU core IDs. The default value is an empty list, which means that the thread can run on any CPU core.
- `scheduling_policy`: (optional) The scheduling policy of the thread that runs the hardware component. The scheduling policy can be one of the following values:
  - `synchronized` (default): The thread will run with the synchronized with the main controller_manager thread. The controller_manager is responsible for triggering the read and write calls of the hardware component.
  - `detached`: The thread will run independently of the main controller_manager thread. The hardware component will manage its own timing for triggering the read and write calls.
- `print_warnings`: (optional) If set to `true`, a warning will be printed if the thread is not able to meet its timing requirements. Default is `true`.

Note: The thread priority is only used when the hardware component is run asynchronously. When the hardware component is run asynchronously, it uses the FIFO scheduling policy.

Examples

The following examples show how to use the different hardware interface types synchronously and asynchronously with `ros2_control` URDF. They can be combined together within the different hardware component types (system, actuator, sensor) (*see detailed documentation*) as follows

For a RRBot with multimodal gripper and external sensor:

```
<ros2_control name="RRBotSystemMutipleGPIOs" type="system">
  <hardware>
    <plugin>ros2_control_demo_hardware/RRBotSystemPositionOnlyHardware</plugin>
    <param name="example_param_hw_start_duration_sec">2.0</param>
    <param name="example_param_hw_stop_duration_sec">3.0</param>
    <param name="example_param_hw_slowdown">2.0</param>
  </hardware>
  <joint name="joint1">
```

(continues on next page)

(continued from previous page)

```
    <command_interface name="position">
      <param name="min">-1</param>
      <param name="max">1</param>
    </command_interface>
    <state_interface name="position"/>
  </joint>
  <joint name="joint2">
    <command_interface name="position">
      <param name="min">-1</param>
      <param name="max">1</param>
    </command_interface>
    <state_interface name="position"/>
  </joint>
  <gpio name="flange_digital_IOs">
    <command_interface name="digital_output1"/>
    <state_interface name="digital_output1"/>    <!-- Needed to know current state of␣
↪the output -->
    <command_interface name="digital_output2"/>
    <state_interface name="digital_output2"/>
    <state_interface name="digital_input1"/>
    <state_interface name="digital_input2"/>
  </gpio>
</ros2_control>
<ros2_control name="MultimodalGripper" type="actuator" is_async="true">
  <properties>
    <async affinity="[2,4]" scheduling_policy="synchronized" print_warnings="true"␣
↪thread_priority="30"/>
  </properties>
  <hardware>
    <plugin>ros2_control_demo_hardware/MultimodalGripper</plugin>
  </hardware>
  <joint name="parallel_fingers">
    <command_interface name="position">
      <param name="min">0</param>
      <param name="max">100</param>
    </command_interface>
    <state_interface name="position"/>
  </joint>
</ros2_control>
<ros2_control name="RRBotForceTorqueSensor2D" type="sensor" is_async="true">
  <hardware>
    <plugin>ros2_control_demo_hardware/ForceTorqueSensor2DHardware</plugin>
    <param name="example_param_read_for_sec">0.43</param>
  </hardware>
  <sensor name="tcp_fts_sensor">
    <state_interface name="fx"/>
    <state_interface name="tz"/>
    <param name="frame_id">kuka_tcp</param>
    <param name="fx_range">100</param>
    <param name="tz_range">100</param>
  </sensor>
  <sensor name="temp_feedback">
    <state_interface name="temperature"/>
  </sensor>
  <gpio name="calibration">
    <command_interface name="calibration_matrix_nr"/>
    <state_interface name="calibration_matrix_nr"/>
```

(continues on next page)

(continued from previous page)

```
  </gpio>
</ros2_control>
```

In the above example, the following components are defined:

- A system hardware component named `RRBotSystemMutipleGPIOs` with two joints and a GPIO component that runs synchronously.
- An actuator hardware component named `MultimodalGripper` with a joint that runs asynchronously with a thread priority of 30.
- A sensor hardware component named `RRBotForceTorqueSensor2D` with two sensors and a GPIO component that runs asynchronously with the default thread priority of 50.

Semantic Components

In order to streamline the configuration of commonly used hardware interface, so-called semantic components can be used to wrap mechanisms to claim / release the interfaces. The base components `semantic_components::SemanticComponentInterface` and `semantic_components::SemanticComponentCommandInterface` are used to define semantic components for read-only and write-only devices, respectively.

List of existing `SemanticComponentInterface` (link to header file) and associated broadcaster, if any:

- IMUSensor, used by *IMU Sensor Broadcaster*
- ForceTorqueSensor, used by *Force Torque Sensor Broadcaster*
- GPSSensor
- MagneticFieldSensor
- PoseSensor, used by *Pose Broadcaster*
- RangeSensor, used by *Range Sensor Broadcaster*

List of existing `SemanticComponentCommandInterface` (link to header file) and associated controller, if any:

- LedRgbDevice

Lifecycle of a Hardware Component

Methods return values have type `rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackRet` with the following meaning:

- `CallbackReturn::SUCCESS` method execution was successful.
- `CallbackReturn::FAILURE` method execution has failed and the lifecycle transition is unsuccessful.
- `CallbackReturn::ERROR` critical error has happened that should be managed in `on_error` method.

The hardware transitions to the following state after each method:

- **UNCONFIGURED** (`on_init`, `on_cleanup`):

  Hardware is only initialized, but communication is not started and no interfaces are imported into `ResourceManager`.

- **INACTIVE** (`on_configure`, `on_deactivate`):

  Communication with the hardware is established and hardware component is configured. States can be read, but command interfaces (System and Actuator only) are not available.

  As of now, it is left to the hardware component implementation to continue using the command received from the `CommandInterfaces` or to skip them completely.

  Note: We plan to implement safety-critical interfaces, see this PR in the roadmap. But currently, all command interfaces are available and will be written, see this issue describing the situation.

- **FINALIZED** (`on_shutdown`):

  Hardware interface is ready for unloading/destruction. Allocated memory is cleaned up.

- **ACTIVE** (`on_activate`):

  States can be read.

  System and Actuator only:

  > Power circuits of hardware are active and hardware can be moved, e.g., brakes are disengaged. Command interfaces are available and the commands should be sent to the hardware

Handling of errors that happen during read() and write() calls

If `hardware_interface::return_type::ERROR` is returned from `read()` or `write()` methods of a hardware_interface class, `on_error(previous_state)` method will be called to handle any error that happened.

Error handling follows the node lifecycle. If successful `CallbackReturn::SUCCESS` is returned and hardware is again in `UNCONFIGURED` state, if any `ERROR` or `FAILURE` happens the hardware ends in `FINALIZED` state and can not be recovered. The only option is to reload the complete plugin, but there is currently no service for this in the Controller Manager.

2.2.6 Mock Components

Mock components are trivial "simulations" of the hardware components, i.e., System, Sensor, and Actuator. They provide ideal behavior by mirroring commands to their states. The corresponding hardware interface can be added instead of real hardware for offline testing of ros2_control framework. The main advantage is that you can test all the "piping" inside the framework without having access to the hardware. This means that you can test your controllers, broadcaster, launch files, and even integrations with, e.g., MoveIt. The main intention is to reduce debugging time on the physical hardware and boost your development.

Generic System

The component implements `hardware_interface::SystemInterface` supporting command and state interfaces. For more information about hardware components check *detailed documentation*.

Features:

- support for mimic joints, which is parsed from the URDF (see the URDF wiki)
- mirroring commands to states with and without offset
- fake command interfaces for setting sensor data from an external node (combined with a *forward controller*)
- fake gpio interfaces for setting sensor data from an external node (combined with a *forward controller*)

Parameters

A full example including all optional parameters (with default values):

```
<ros2_control name="MockHardwareSystem" type="system">
  <hardware>
    <plugin>mock_components/GenericSystem</plugin>
    <param name="calculate_dynamics">false</param>
    <param name="custom_interface_with_following_offset"></param>
    <param name="disable_commands">false</param>
    <param name="mock_gpio_commands">false</param>
    <param name="mock_sensor_commands">false</param>
    <param name="position_state_following_offset">0.0</param>
  </hardware>
  <joint name="joint1">
    <command_interface name="position"/>
    <command_interface name="velocity"/>
    <state_interface name="position">
      <param name="initial_value">3.45</param>
    </state_interface>
    <state_interface name="velocity"/>
    <state_interface name="acceleration"/>
  </joint>
  <joint name="joint2">
    <command_interface name="velocity"/>
    <command_interface name="acceleration"/>
    <state_interface name="position">
      <param name="initial_value">2.78</param>
    </state_interface>
    <state_interface name="position"/>
    <state_interface name="velocity"/>
    <state_interface name="acceleration"/>
  </joint>
  <gpio name="flange_vacuum">
    <command_interface name="vacuum"/>
    <state_interface name="vacuum" data_type="double"/>
  </gpio>
</ros2_control>
```

See *example_2* for an example using `calculate_dynamics` or *example_10* for using in combination with GPIO interfaces.

Component Parameters

calculate_dynamics (optional; boolean; default: false)
Calculation of states from commands by using Euler-forward integration or finite differences.

custom_interface_with_following_offset (optional; string; default: “”)
Mapping of offsetted commands to a custom interface (see `position_state_following_offset`).

disable_commands (optional; boolean; default: false)
Disables mirroring commands to states. This option is helpful to simulate an erroneous connection to the hardware when nothing breaks, but suddenly there is no feedback from a hardware interface. Or it can help you to test your setup when the hardware is running without feedback, i.e., in open loop configuration.

mock_gpio_commands (optional; boolean; default: false)
Creates fake command interfaces for faking GPIO states with an external command. Those interfaces are usually

used by a *forward controller* to provide access from ROS-world.

mock_sensor_commands (optional; boolean; default: false)
Creates fake command interfaces for faking sensor measurements with an external command. Those interfaces are usually used by a *forward controller* to provide access from ROS-world.

position_state_following_offset (optional; double; default: 0.0)
Following offset added to the state values when commands are mirrored to states. If `custom_interface_with_following_offset` is empty, the offset is applied to the `position` state interface. If a custom interface is set, the `position` state value + offset is applied to that interface.

Per-Interface Parameters

initial_value (optional; double)
Initial value of certain state interface directly after startup. Example:

```
<state_interface name="position">
  <param name="initial_value">3.45</param>
</state_interface>
```

If unset, the initial value is set to 0.0 for state interfaces of joints in `configure` lifecycle transition.

Note: This parameter is shared with the gz_ros2_control plugins for joint interfaces. For Mock components it is also possible to set initial values for gpio or sensor state interfaces.

2.3 Guidelines and Best Practices

2.3.1 Debugging

All controllers and hardware components are plugins loaded into the `controller_manager`. Therefore, the debugger must be attached to the `controller_manager`. If multiple `controller_manager` instances are running on your robot or machine, you need to attach the debugger to the `controller_manager` associated with the hardware component or controller you want to debug.

How-To

- Install `xterm`, `gdb` and `gdbserver` on your system

```
sudo apt install xterm gdb gdbserver
```

- Make sure you run a "debug" or "release with debug information" build: This is done by passing `--cmake-args -DCMAKE_BUILD_TYPE=RelWithDebInfo` to `colcon build`. Remember that in release builds some breakpoints might not behave as you expect as the the corresponding line might have been optimized by the compiler. For such cases, a full Debug build (`--cmake-args -DCMAKE_BUILD_TYPE=Debug`) is recommended.
- Adapt the launch file to run the controller manager with the debugger attached:
  - Version A: Run it directly with the gdb CLI:

    Add `prefix=['xterm -e gdb -ex run --args']` to the `controller_manager` node entry in your launch file. Due to how `ros2launch` works we need to run the specific node in a separate terminal instance.

- Version B: Run it with gdbserver:

  Add `prefix=['gdbserver localhost:3000']` to the `controller_manager` node entry in your launch file. Afterwards, you can either attach a gdb CLI instance or any IDE of your choice to that `gdbserver` instance. Ensure you start your debugger from a terminal where you have sourced your workspace to properly resolve all paths.

Example launch file entry:

```
# Obtain the controller config file for the ros2 control node
controller_config_file = get_package_file("<package name>", "config/controllers.
↪yaml")

controller_manager = Node(
    package="controller_manager",
    executable="ros2_control_node",
    parameters=[controller_config_file],
    output="both",
    emulate_tty=True,
    remappings=[
        ("~/robot_description", "/robot_description")
    ],
    prefix=['xterm -e gdb -ex run --args']  # or prefix=['gdbserver localhost:3000
↪']
)

ld.add_action(controller_manager)
```

Catching Exceptions

- The controller manager by default catches most of the exceptions thrown by controllers and hardware components to avoid crashing the whole system. However, it does print the exception type and message to the console. This can make debugging difficult as the debugger might not catch the exception and also when the exception message is not clear enough to identify the root cause. This behaviour can be disabled by setting the parameter `handle_exceptions` to `false` in the controller manager node, this way the exceptions will propagate up to the controller manager and can be caught by the debugger (or) crash by printing the stacktrace during normal execution.

  Example controller manager config file:

```
controller_manager:
  ros__parameters:
    update_rate: 1000
    handle_exceptions: false
```

Additional notes

- Debugging plugins

  You can only set breakpoints in plugins after the plugin has been loaded. In the ros2_control context this means after the controller / hardware component has been loaded:

- Debug builds

  It's often practical to include debug information only for the specific package you want to debug. `colcon build --packages-select [package_name] --cmake-args`

`-DCMAKE_BUILD_TYPE=RelWithDebInfo` or `colcon build --packages-select [package_name] --cmake-args -DCMAKE_BUILD_TYPE=Debug`

- Realtime

> **Warning:** The `update/on_activate/on_deactivate` method of a controller and the `read/write/on_activate/perform_command_mode_switch` methods of a hardware component all run in the context of the realtime update loop. Setting breakpoints there can and will cause issues that might even break your hardware in the worst case.

From experience, it might be better to use meaningful logs for the real-time context (with caution) or to add additional debug state interfaces (or publishers in the case of a controller).

However, running the controller_manager and your plugin with gdb can still be very useful for debugging errors such as segfaults, as you can gather a full backtrace.

References

- ROS 2 and GDB
- Using GDB to debug a plugin
- GDB CLI Tutorial

2.3.2 Introspection of the ros2_control setup

With the integration of the `pal_statistics` package, the `controller_manager` node publishes the registered variables within the same process to the `~/introspection_data` topics. By default, all `State` and `Command` interfaces in the `controller_manager` are registered when they are added, and are unregistered when they are removed from the `ResourceManager`. The state of the all the registered entities are published at the end of every `update` cycle of the `controller_manager`. For instance, In a complete synchronous ros2_control setup (with synchronous controllers and hardware components), this data in the `Command` interface is the command used by the hardware components to command the hardware.

All the registered variables are published over 3 topics: `~/introspection_data/full`, `~/introspection_data/names`, and `~/introspection_data/values`. - The `~/introspection_data/full` topic publishes the full introspection data along with names and values in a single message. This can be useful to track or view variables and information from command line. - The `~/introspection_data/names` topic publishes the names of the registered variables. This topic contains the names of the variables registered. This is only published every time a a variables is registered and unregistered. - The `~/introspection_data/values` topic publishes the values of the registered variables. This topic contains the values of the variables registered.

The topics `~/introspection_data/full` and `~/introspection_data/values` are always published on every update cycle asynchronously, provided that there is at least one subscriber to these topics.

The topic `~/introspection_data/full` can be used to integrate with your custom visualization tools or to track the variables from the command line. The topic `~/introspection_data/names` and `~/introspection_data/values` are to be used for visualization tools like PlotJuggler or RQT plot to visualize the data.

Note: If you have a high frequency of data, it is recommended to use the `~/introspection_data/names` and `~/introspection_data/values` topic. So, that the data transferred and stored is minimized.

Along with the above introspection data, the `controller_manager` also publishes the statistics of the execution time and periodicity of the read and write cycles of the hardware components and the update cycle of the controllers. This is done by registering the statistics of these variables and publishing them on the `~/statistics` topic.

All the registered variables are published over 3 topics: `~/statistics/full`, `~/statistics/names`, and `~/statistics/values`. - The `~/statistics/full` topic publishes the full introspection data along with names and values in a single message. This can be useful to track or view variables and information from command line. - The `~/statistics/names` topic publishes the names of the registered variables. This topic contains the names of the variables registered. This is only published every time a a variables is registered and unregistered. - The `~/statistics/values` topic publishes the values of the registered variables. This topic contains the values of the variables registered.

This topic is mainly used to introspect the behaviour of the realtime loops, this is very crucial for hardware that need to meet strict deadlines and also to understand the which component of the ecosystem is consuming more time in the realtime loop.

How to introspect internal variables of controllers and hardware components

Any member variable of a controller or hardware component can be registered for the introspection. It is very important that the lifetime of this variable exists as long as the controller or hardware component is available.

Note: If a variable's lifetime is not properly managed, it may be attempted to read, which in the worst case scenario will cause a segmentation fault.

How to register a variable for introspection

1. Include the necessary headers in the controller or hardware component header file.

```
#include <hardware_interface/introspection.hpp>
```

2. Register the variable in the configure method of the controller or hardware component.

```
void MyController::on_configure()
{
  ...
  // Register the variable for introspection (disabled by default)
  // The variable is introspected only when the controller is active and
  // then deactivated when the controller is deactivated.
  REGISTER_ROS2_CONTROL_INTROSPECTION("my_variable_name", &my_variable_);
  ...
}
```

3. By default, the introspection of all the registered variables of the controllers and the hardware components is only activated, when they are active and it is deactivated when the controller or hardware component is deactivated.

   Note: If you want to keep the introspection active even when the controller or hardware component is not active, you can do that by calling `this->enable_introspection(true)` in the `on_configure` and `on_deactivate` method of the controller or hardware component after registering the variables.

Types of entities that can be introspected

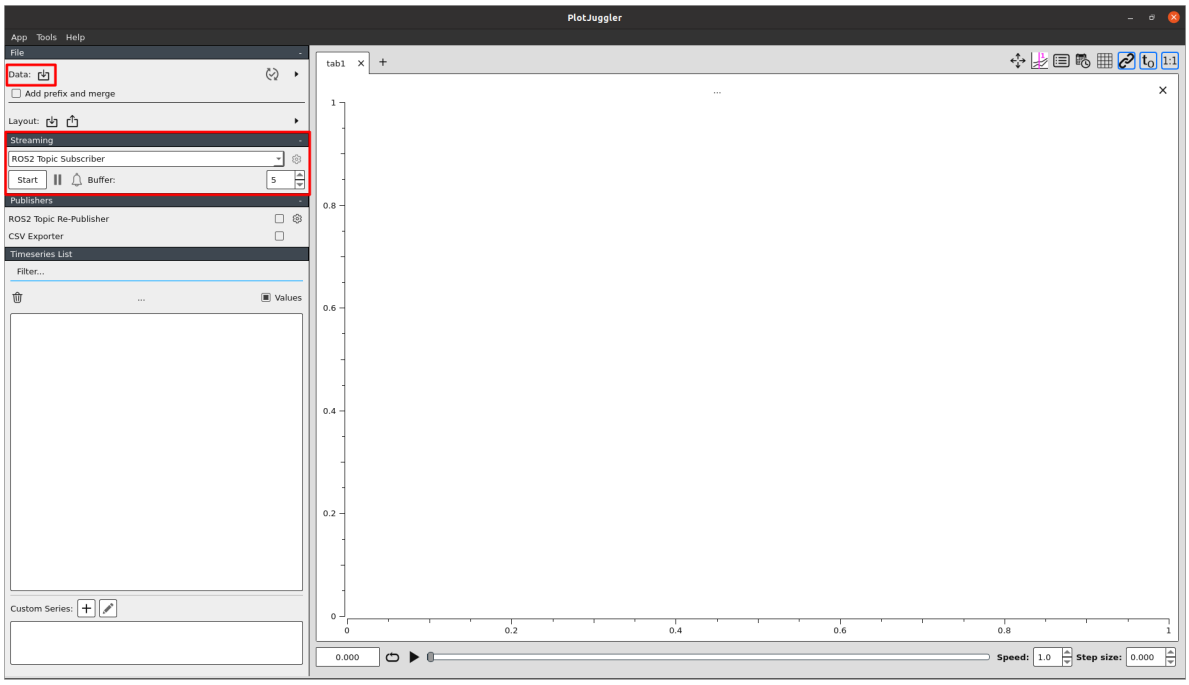
- Any variable that can be cast to a double is suitable for registration.
- A function that returns a value that can be cast to a double is also suitable for registration.
- Variables of complex structures can be registered by having defined introspection for their every internal variable.
- Introspection of custom types can be done by defining a custom introspection function.

Note: Registering the variables for introspection is not real-time safe. It is recommended to register the variables in the `on_configure` method only.

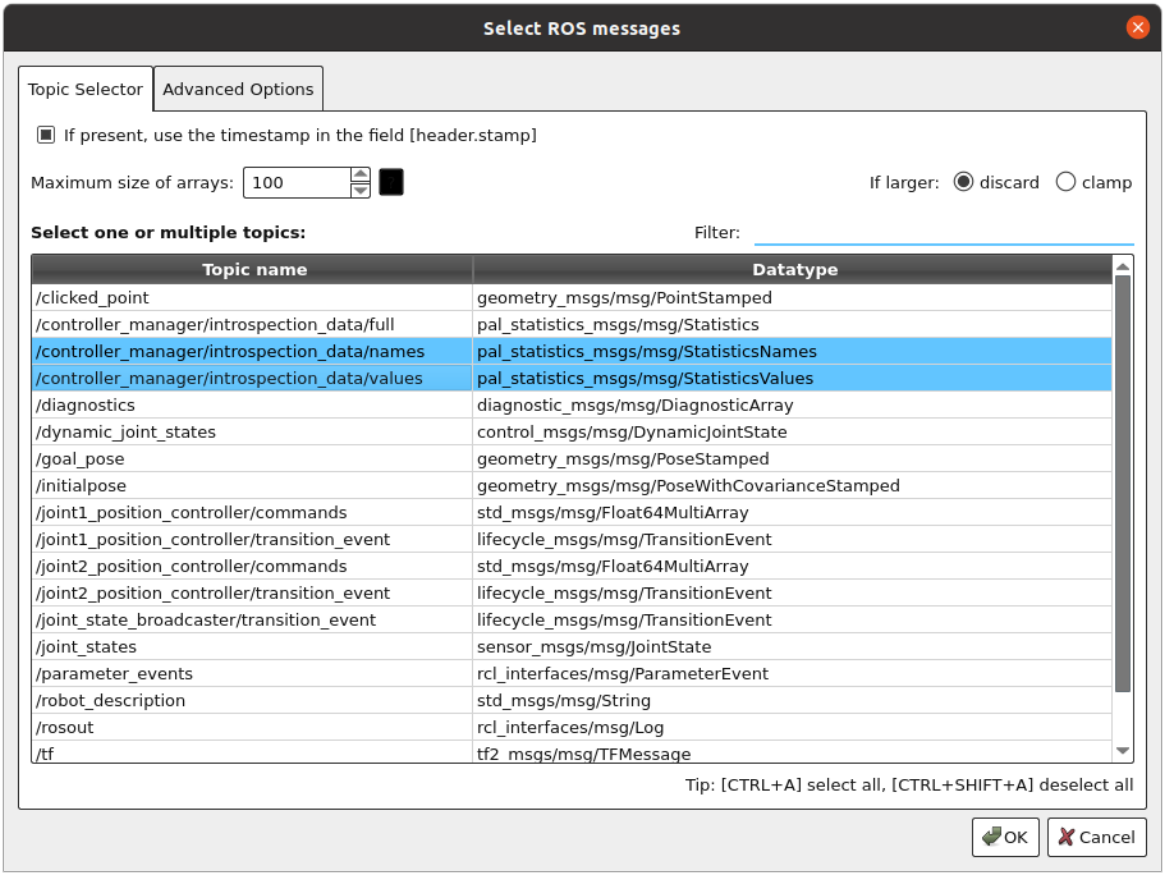
2.3.3 Data Visualization

Data can be visualized with any tools that display ROS topics, but we recommend PlotJuggler for viewing high resolution live data, or data in bags.

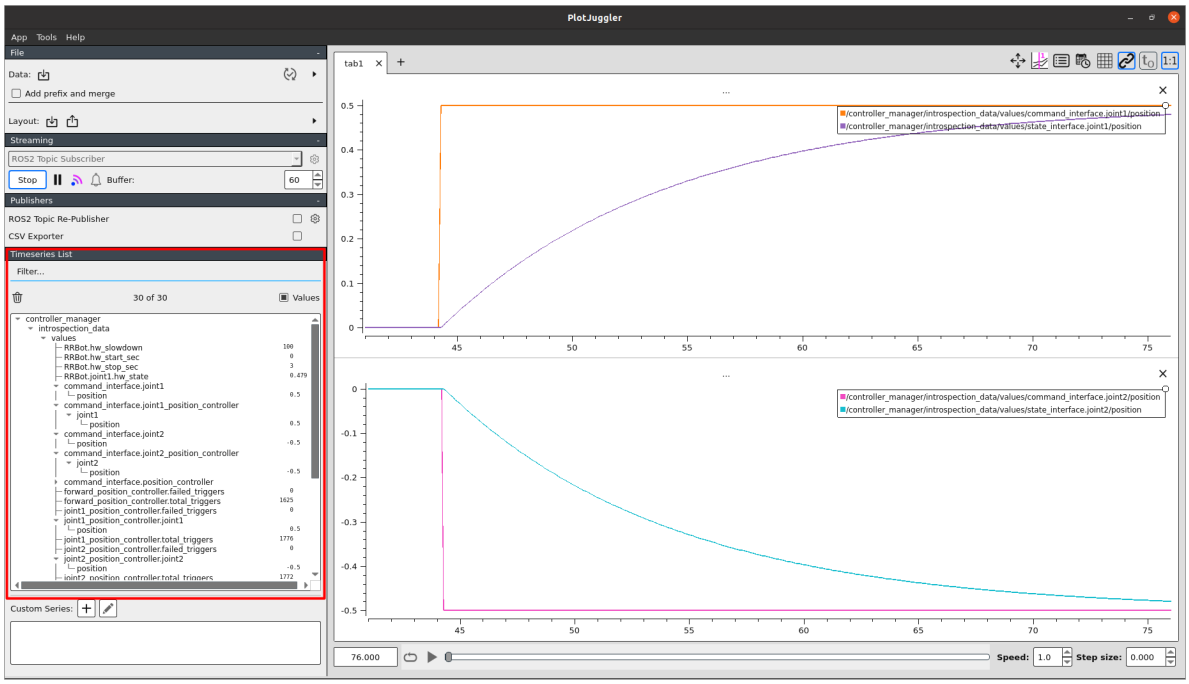
1. Open `PlotJuggler` by running `ros2 run plotjuggler plotjuggler` from the command line.



2. Visualize the data by importing from the ros2bag file or subscribing to the ROS2 topics live with the `ROS2 Topic Subscriber` option under `Streaming` header.
3. Choose the topics `~/introspection_data/names` and `~/introspection_data/values` from the popup window.



4. Then, select the variables that are of your interest and drag them to the plot.



CHAPTER

THREE

ROS2_CONTROLLERS

Commonly used and generalized controllers for ros2_control framework that are ready to use with many robots, MoveIt2 and Nav2.

Link to GitHub Repository

3.1 Guidelines and Best Practices

3.1.1 Wheeled Mobile Robot Kinematics

This page introduces the kinematics of different wheeled mobile robots. For further reference see Siciliano et.al - Robotics: Modelling, Planning and Control and Kevin M. Lynch and Frank C. Park - Modern Robotics: Mechanics, Planning, And Control.

Wheeled mobile robots can be classified in two categories:

Omnidirectional robots
: which can move instantaneously in any direction in the plane, and

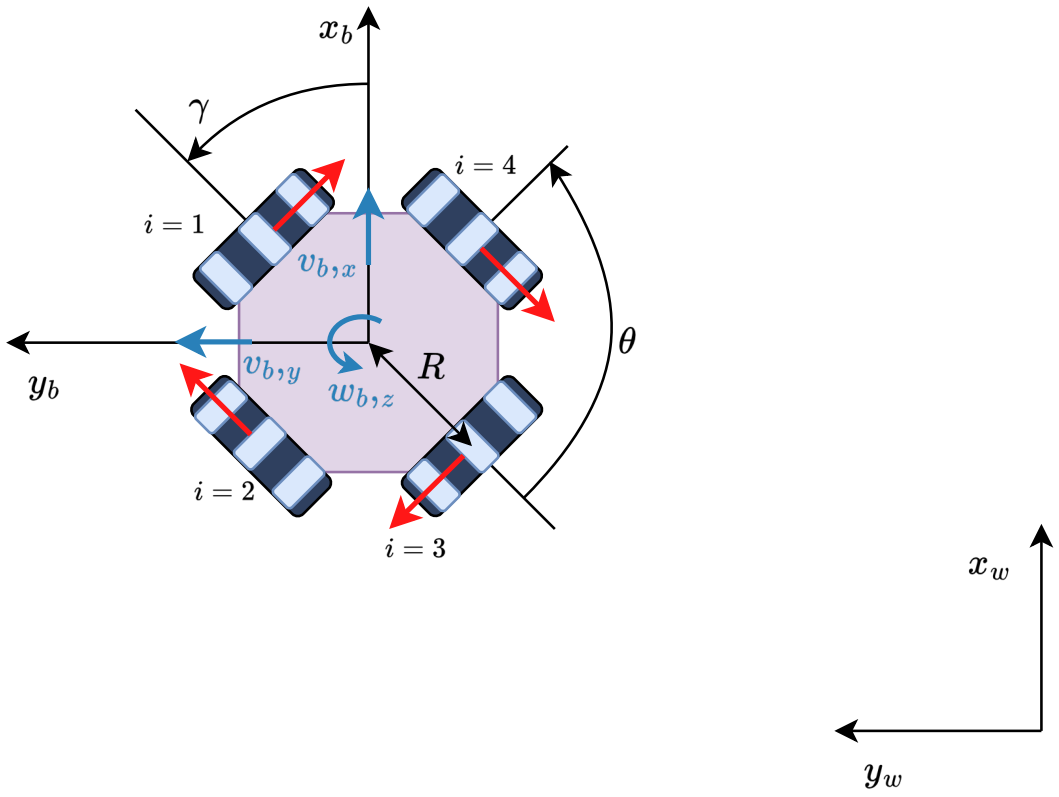
Nonholonomic robots
: which cannot move instantaneously in any direction in the plane.

The forward integration of the kinematic model using the encoders of the wheel actuators — is referred to as **odometric localization** or **passive localization** or **dead reckoning**. We will call it just **odometry**.

Omnidirectional Wheeled Mobile Robots

Omnidirectional Drive Robots using Omni Wheels

The below explains the kinematics of omnidirectional drive robots using 3 or more omni wheels. It follows the coordinate conventions defined in ROS REP 103.



- x_b, y_b is the robot's body-frame coordinate system, located at the contact point of the wheel on the ground.
- x_w, y_w is the world coordinate system.
- $v_{b,x}$, is the robot's linear velocity on the x-axis.
- $v_{b,y}$ is the robot's linear velocity on the y-axis.
- $\omega_{b,z}$ is the robot's angular velocity on the z-axis.
- R is the robot's radius / the distance between the robot's center and the wheels.
- Red arrows on the wheel i signify the positive direction of their rotation ω_i
- γ is the angular offset of the first wheel from x_b.
- θ is the angle between each wheel which can be calculated using the below equation where n is the number of wheels.

$$\theta = \frac{2\pi}{n}$$

Inverse Kinematics

The necessary angular velocity of the wheels to achieve a desired body twist can be calculated using the below matrix:

$$A = \begin{bmatrix} \sin(\gamma) & -\cos(\gamma) & -R \\ \sin(\theta+\gamma) & -\cos(\theta+\gamma) & -R \\ \sin(2\theta+\gamma) & -\cos(2\theta+\gamma) & -R \\ \sin(3\theta+\gamma) & -\cos(3\theta+\gamma) & -R \\ \vdots & \vdots & \vdots \\ \sin((n-1)\theta+\gamma) & -\cos((n-1)\theta+\gamma) & -R \end{bmatrix}$$

$$\begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \\ \vdots \\ \omega_n \end{bmatrix} = \frac{1}{r} A \begin{bmatrix} v_{b,x} \\ v_{b,y} \\ \omega_{b,z} \end{bmatrix}$$

Here $\omega_1, \ldots, \omega_n$ are the angular velocities of the wheels and r is the radius of the wheels. These equations can be written in algebraic form for any wheel i like this:

$$\omega_i = \frac{\sin((i-1)\theta + \gamma) v_{b,x} - \cos((i-1)\theta + \gamma) v_{b,y} - R\omega_{b,z}}{r}$$

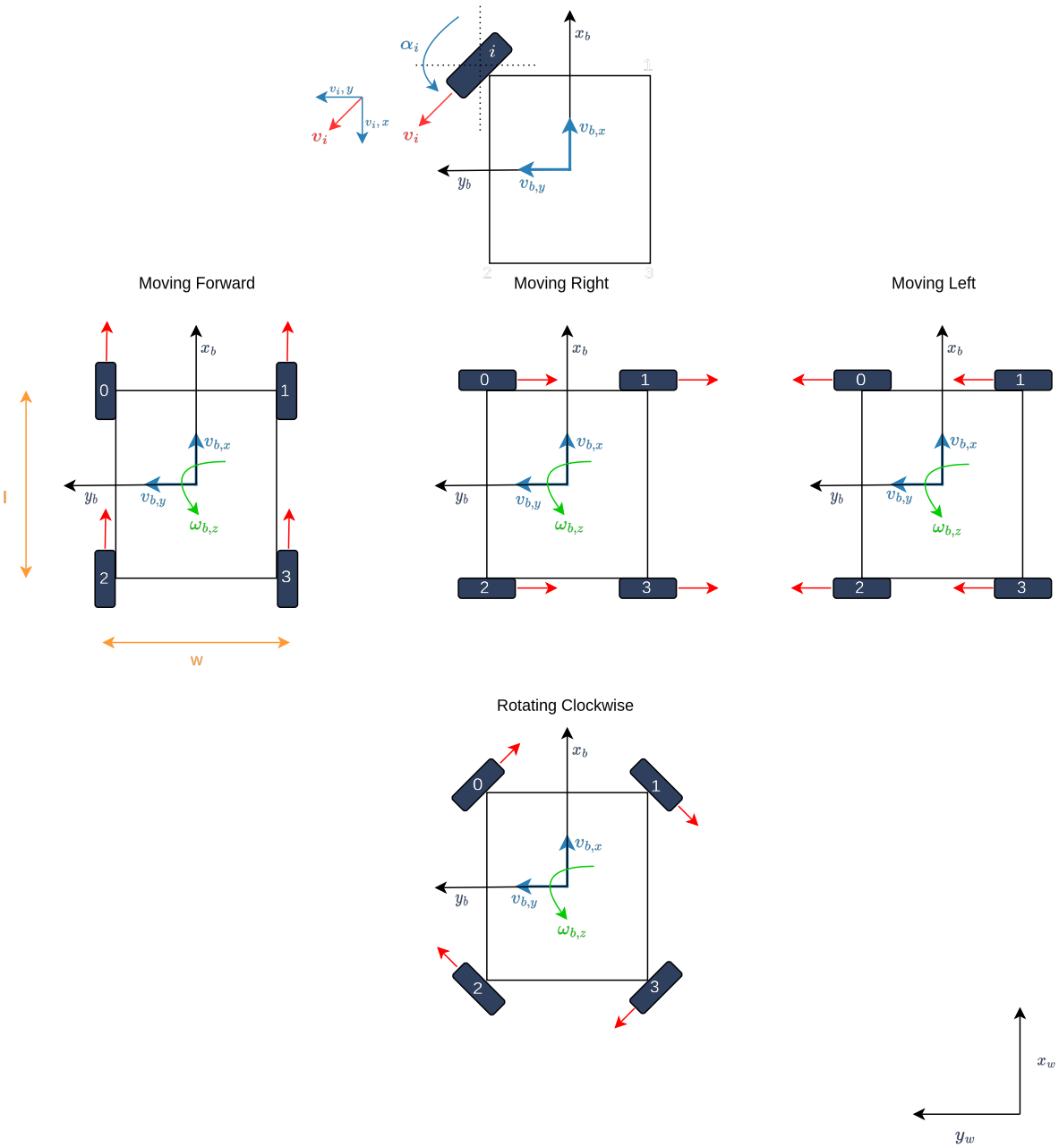
Forward Kinematics

The body twist of the robot can be obtained from the wheel velocities by using the pseudoinverse of matrix A.

$$\begin{bmatrix} v_{b,x} \\ v_{b,y} \\ \omega_{b,z} \end{bmatrix} = r A^{\dagger} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \\ \vdots \\ \omega_n \end{bmatrix}$$

Swerve Drive Robots

The below explains the kinematics of omnidirectional drive robots using four swerve modules, each with independently controlled steering and driving motors. It follows the coordinate conventions defined in REP-103.



- x_b, y_b is the robot's body-frame coordinate system, located at the geometric center of the robot.
- x_w, y_w is the world coordinate system.
- $v_{b,x}$ is the robot's linear velocity on the x-axis.
- $v_{b,y}$ is the robot's linear velocity on the y-axis.
- $\omega_{b,z}$ is the robot's angular velocity on the z-axis.
- l is the wheelbase (distance between front and rear wheels).
- w is the track width (distance between left and right wheels).
- Red arrows on wheel i signify the direction of the wheel's velocity v_i.

Each swerve module i, for $i = 0, 1, 2, 3$ (typically front-left, front-right, back-left, back-right) is located at $(l_{i,x}, l_{i,y})$ relative to the center, typically:

- Front-left: $(l/2, w/2)$
- Front-right: $(l/2, -w/2)$
- Back-left: $(-l/2, w/2)$
- Back-right: $(-l/2, -w/2)$

Inverse Kinematics

For each module i at position $(l_{i,x}, l_{i,y})$, the velocity vector is:

$$\begin{bmatrix} v_{i,x} \\ v_{i,y} \end{bmatrix} = \begin{bmatrix} v_{b,x} - \omega_{b,z} l_{i,y} \\ v_{b,y} + \omega_{b,z} l_{i,x} \end{bmatrix}$$

The wheel velocity v_i and steering angle ϕ_i are:

$$v_i = \sqrt{v_{i,x}^2 + v_{i,y}^2}$$

$$\phi_i = \arctan 2(v_{i,y}, v_{i,x})$$

Odometry

The body twist of the robot is computed from the wheel velocities v_i and steering angles ϕ_i. Each module's velocity components in the body frame are:

$$v_{i,x} = v_i \cos(\phi_i), \quad v_{i,y} = v_i \sin(\phi_i)$$

The chassis velocities are calculated as:

$$v_{b,x} = \frac{1}{4} \sum_{i=0}^{3} v_{i,x}, \quad v_{b,y} = \frac{1}{4} \sum_{i=0}^{3} v_{i,y}$$

$$\omega_{b,z} = \frac{\sum_{i=0}^{3} (v_{i,y} l_{i,x} - v_{i,x} l_{i,y})}{\sum_{i=0}^{3} (l_{i,x}^2 + l_{i,y}^2)}$$

Odometry updates the robot's pose (x, y, θ) in the global frame using the computed chassis velocities. The global velocities are:

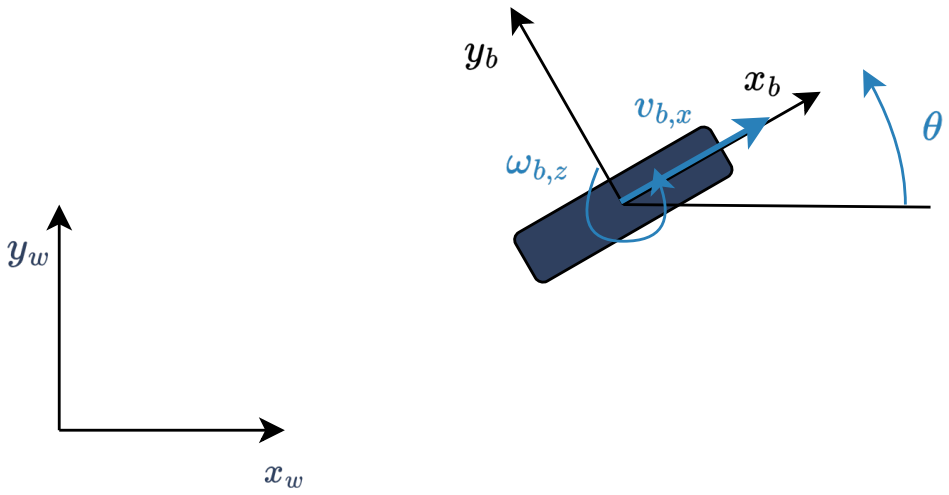
$$v_{x,\text{global}} = v_{b,x} \cos(\theta) - v_{b,y} \sin(\theta)$$

$$v_{y,\text{global}} = v_{b,x} \sin(\theta) + v_{b,y} \cos(\theta)$$

Nonholonomic Wheeled Mobile Robots

Unicycle model

To define the coordinate systems (ROS coordinate frame conventions, the coordinate systems follow the right-hand rule), consider the following simple unicycle model



- x_b, y_b is the robot's body-frame coordinate system, located at the contact point of the wheel on the ground.
- x_w, y_w is the world coordinate system.
- x, y are the robot's Cartesian coordinates in the world coordinate system.
- θ is the robot's heading angle, i.e. the orientation of the robot's x_b-axis w.r.t. the world's x_w-axis.

In the following, we want to command the robot with a desired body twist

$$\vec{\nu}_b = \begin{bmatrix} \vec{\omega}_b \\ \vec{v}_b \end{bmatrix},$$

where $\vec{v}_b$ is the linear velocity of the robot in its body-frame, and $\vec{\omega}_b$ is the angular velocity of the robot in its body-frame. As we consider steering robots on a flat surface, it is sufficient to give

- $v_{b,x}$, i.e. the linear velocity of the robot in direction of the x_b axis.
- $\omega_{b,z}$, i.e. the angular velocity of the robot about the x_z axis.

as desired system inputs. The forward kinematics of the unicycle can be calculated with

$$\begin{aligned} \dot{x} &= v_{b,x} \cos(\theta) \\ \dot{y} &= v_{b,x} \sin(\theta) \\ \dot{\theta} &= \omega_{b,z} \end{aligned}$$

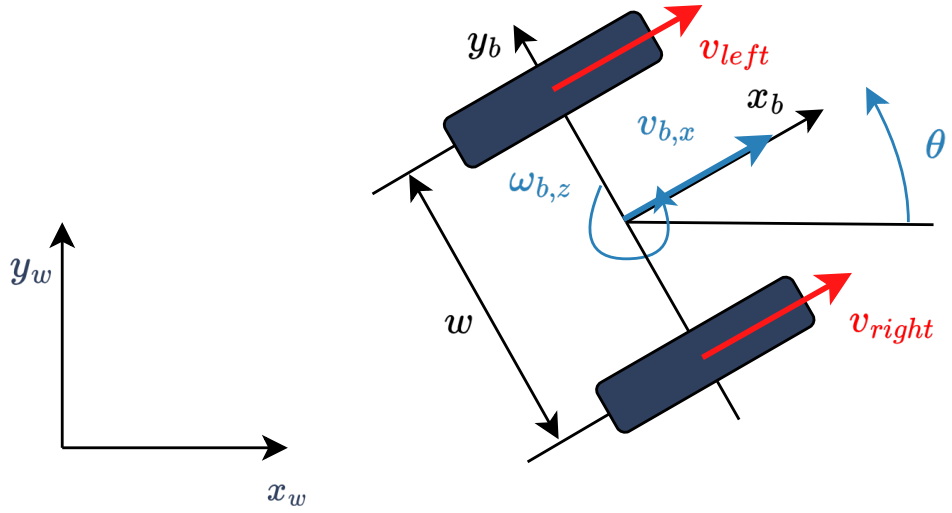
We will formulate the inverse kinematics to calculate the desired commands for the robot (wheel speed or steering) from the given body twist.

Differential Drive Robot

Citing Siciliano et.al - Robotics: Modelling, Planning and Control:

```
A unicycle in the strict sense (i.e., a vehicle equipped with a single wheel)
is a robot with a serious problem of balance in static conditions. However,
there exist vehicles that are kinematically equivalent to a unicycle but more
stable from a mechanical viewpoint.
```

One of these vehicles is the differential drive robot, which has two wheels, each of which is driven independently.



- w is the wheel track (the distance between the wheels).

Forward Kinematics

The forward kinematics of the differential drive model can be calculated from the unicycle model above using

$$v_{b,x} = \frac{v_{right} + v_{left}}{2}$$
$$\omega_{b,z} = \frac{v_{right} - v_{left}}{w}$$

Inverse Kinematics

The necessary wheel speeds to achieve a desired body twist can be calculated with:

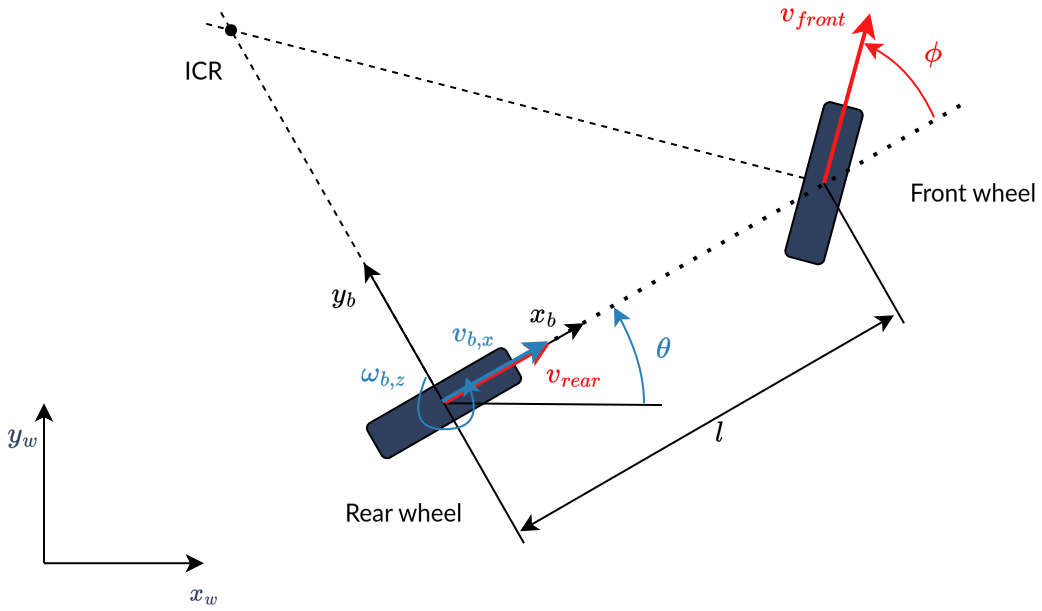
$$v_{left} = v_{b,x} - \omega_{b,z} w/2$$
$$v_{right} = v_{b,x} + \omega_{b,z} w/2$$

Odometry

We can use the forward kinematics equations above to calculate the robot's odometry directly from the encoder readings.

Car-Like (Bicycle) Model

The following picture shows a car-like robot with two wheels, where the front wheel is steerable. This model is also known as the bicycle model.



- ϕ is the steering angle of the front wheel, counted positive in direction of rotation around x_z-axis.
- v_{rear}, v_{front} is the velocity of the rear and front wheel.
- l is the wheelbase.

We assume that the wheels are rolling without slipping. This means that the velocity of the contact point of the wheel with the ground is zero and the wheel's velocity points in the direction perpendicular to the wheel's axis. The **Instantaneous Center of Rotation** (ICR), i.e. the center of the circle around which the robot rotates, is located at the intersection of the lines that are perpendicular to the wheels' axes and pass through the contact points of the wheels with the ground.

As a consequence of the no-slip condition, the velocity of the two wheels must satisfy the following constraint:

$$v_{rear} = v_{front} \cos(\phi)$$

Forward Kinematics

The forward kinematics of the car-like model can be calculated with

$$\begin{aligned}\dot{x} &= v_{b,x} \cos(\theta) \\ \dot{y} &= v_{b,x} \sin(\theta) \\ \dot{\theta} &= \frac{v_{b,x}}{l} \tan(\phi)\end{aligned}$$

Inverse Kinematics

The steering angle is one command input of the robot:

$$\phi = \arctan\left(\frac{l w_{b,z}}{v_{b,x}}\right)$$

For the rear-wheel drive, the velocity of the rear wheel is the second input of the robot:

$$v_{rear} = v_{b,x}$$

For the front-wheel drive, the velocity of the front wheel is the second input of the robot:

$$v_{front} = \frac{v_{b,x}}{\cos(\phi)}$$

Odometry

We have to distinguish between two cases: Encoders on the rear wheel or on the front wheel.

For the rear wheel case:

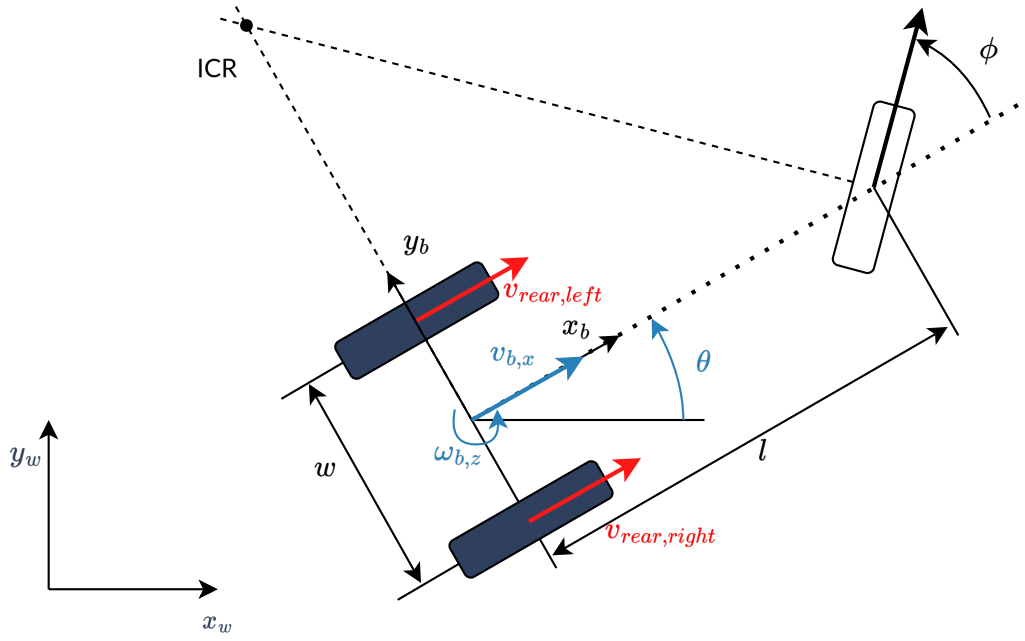
$$\dot{x} = v_{rear} \cos(\theta)$$
$$\dot{y} = v_{rear} \sin(\theta)$$
$$\dot{\theta} = \frac{v_{rear}}{l} \tan(\phi)$$

For the front wheel case:

$$\dot{x} = v_{front} \cos(\theta) \cos(\phi)$$
$$\dot{y} = v_{front} \sin(\theta) \cos(\phi)$$
$$\dot{\theta} = \frac{v_{front}}{l} \sin(\phi)$$

Double-Traction Axle

The following image shows a car-like robot with three wheels, with two independent traction wheels at the rear.



- w_r is the wheel track of the rear axle.

Forward Kinematics

The forward kinematics is the same as the car-like model above.

Inverse Kinematics

The turning radius of the robot is

$$R_b = \frac{l}{\tan(\phi)}$$

Then the velocity of the rear wheels must satisfy these conditions to avoid skidding

$$v_{rear,left} = v_{b,x} \frac{R_b - w_r/2}{R_b}$$
$$v_{rear,right} = v_{b,x} \frac{R_b + w_r/2}{R_b}$$

Odometry

The calculation of $v_{b,x}$ from two encoder measurements of the traction axle is overdetermined. If there is no slip and the encoders are ideal,

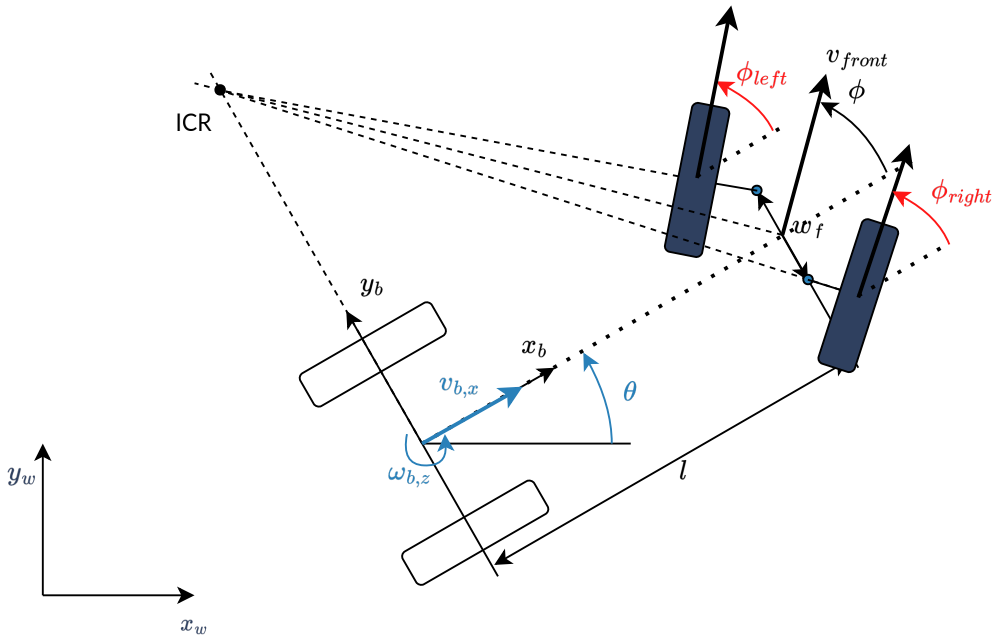
$$v_{b,x} = v_{rear,left} \frac{R_b}{R_b - w_r/2} = v_{rear,right} \frac{R_b}{R_b + w_r/2}$$

holds. But to get a more robust solution, we take the average of both , i.e.,

$$v_{b,x} = 0.5 \left(v_{rear,left} \frac{R_b}{R_b - w_r/2} + v_{rear,right} \frac{R_b}{R_b + w_r/2} \right).$$

Ackermann Steering

The following image shows a four-wheeled robot with two independent steering wheels in the front.



- w_f is the wheel track of the front axle, measured between the two kingpins.

To prevent the front wheels from slipping, the steering angle of the front wheels cannot be equal. This is the so-called **Ackermann steering**.

Note: Ackermann steering can also be achieved by a mechanical linkage between the two front wheels. In this case the robot has only one steering input, and the steering angle of the two front wheels is mechanically coupled. The inverse kinematics of the robot will then be the same as in the car-like model above.

Forward Kinematics

The forward kinematics is the same as for the car-like model above.

Inverse Kinematics

The turning radius of the robot is

$$R_b = \frac{l}{\tan(\phi)}$$

Then the steering angles of the front wheels must satisfy these conditions to avoid skidding

$$\phi_{left} = \arctan\left(\frac{l}{R_b - w_f/2}\right) = \arctan\left(\frac{2l\sin(\phi)}{2l\cos(\phi) - w_f\sin(\phi)}\right)$$
$$\phi_{right} = \arctan\left(\frac{l}{R_b + w_f/2}\right) = \arctan\left(\frac{2l\sin(\phi)}{2l\cos(\phi) + w_f\sin(\phi)}\right)$$

Odometry

The calculation of ϕ from two angle measurements of the steering axle is overdetermined. If there is no slip and the measurements are ideal,

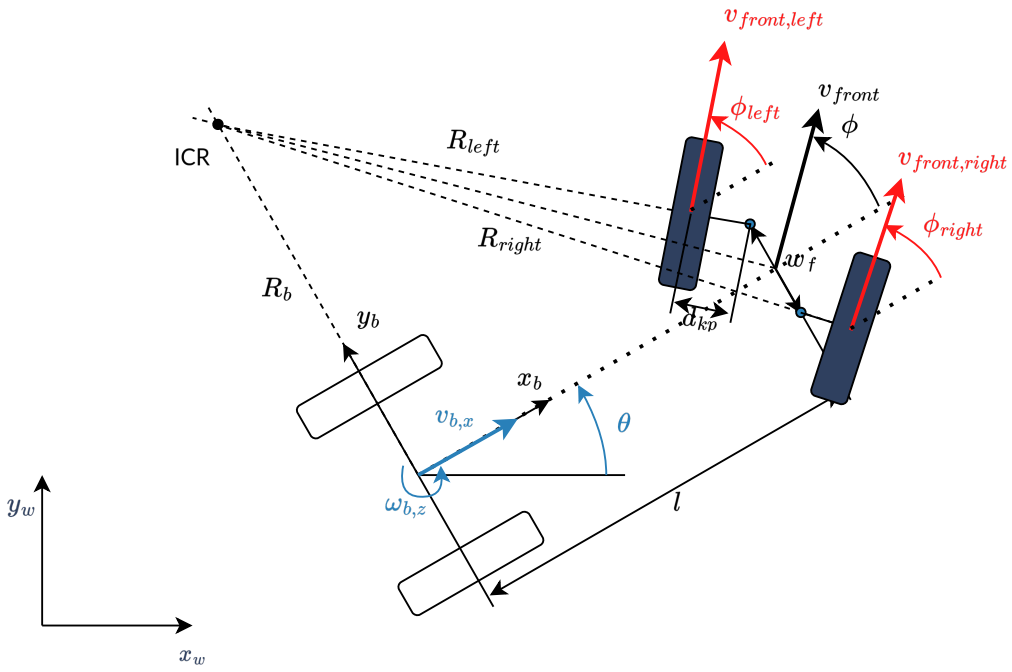
$$\phi = \arctan\left(\frac{l\tan(\phi_{left})}{l + w_f/2\tan(\phi_{left})}\right) = \arctan\left(\frac{l\tan(\phi_{right})}{l - w_f/2\tan(\phi_{right})}\right)$$

holds. But to get a more robust solution, we take the average of both , i.e.,

$$\phi = 0.5\left(\arctan\left(\frac{l\tan(\phi_{left})}{l + w_f/2\tan(\phi_{left})}\right) + \arctan\left(\frac{l\tan(\phi_{right})}{l - w_f/2\tan(\phi_{right})}\right)\right).$$

Ackermann Steering with Traction

The following image shows a four-wheeled car-like robot with two independent steering wheels at the front, which are also driven independently.



- d_{kp} is the distance from the kingpin to the contact point of the front wheel with the ground.

Forward Kinematics

The forward kinematics is the same as the car-like model above.

Inverse Kinematics

To avoid slipping of the front wheels, the velocity of the front wheels cannot be equal and

$$\frac{v_{front,left}}{R_{left}} = \frac{v_{front,right}}{R_{right}} = \frac{v_{b,x}}{R_b}$$

with turning radius of the robot and the left/right front wheel

$$\begin{aligned} R_b &= \frac{l}{\tan(\phi)} \\ R_{left} &= \frac{l - d_{kp}\sin(\phi_{left})}{\sin(\phi_{left})} \\ R_{right} &= \frac{l + d_{kp}\sin(\phi_{right})}{\sin(\phi_{right})}. \end{aligned}$$

This results in the following inverse kinematics equations

$$\begin{aligned} v_{front,left} &= \frac{v_{b,x}(l - d_{kp}\sin(\phi_{left}))}{R_b\sin(\phi_{left})} \\ v_{front,right} &= \frac{v_{b,x}(l + d_{kp}\sin(\phi_{right}))}{R_b\sin(\phi_{right})} \end{aligned}$$

with the steering angles of the front wheels from the Ackermann steering equations above.

Odometry

The calculation of $v_{b,x}$ from two encoder measurements of the traction axle is again overdetermined. If there is no slip and the encoders are ideal,

$$v_{b,x} = v_{front,left}\frac{R_b\sin(\phi_{left})}{l - d_{kp}\sin(\phi_{left})} = v_{front,right}\frac{R_b\sin(\phi_{right})}{l + d_{kp}\sin(\phi_{right})}$$

holds. But to get a more robust solution, we take the average of both , i.e.,

$$v_{b,x} = 0.5\left(v_{front,left}\frac{R_b\sin(\phi_{left})}{l - d_{kp}\sin(\phi_{left})} + v_{front,right}\frac{R_b\sin(\phi_{right})}{l + d_{kp}\sin(\phi_{right})}\right).$$

3.1.2 Writing a new controller

In this framework controllers are libraries, dynamically loaded by the controller manager using the pluginlib interface. The following is a step-by-step guide to create source files, basic tests, and compile rules for a new controller.

1. **Preparing package**

   If the package for the controller does not exist, then create it first. The package should have `ament_cmake` as a build type. Generally, you can use `ros2 pkg create <controller_name_package> --build-type ament_cmake`. Use the `--help` flag for more information on how to use this command. There is also an option to create library source files and compile rules to help you in the following steps.

2. **Preparing source files**

   After creating the package, you should have at least `CMakeLists.txt` and `package.xml` files in it. Create also `include/<PACKAGE_NAME>/` and `src` folders if they do not already exist. In `include/<PACKAGE_NAME>/` folder add `<controller_name>.hpp` and `<controller_name>.cpp` in the `src` folder.

3. **Adding declarations into header file (.hpp)**
    1. Take care that you use header guards. ROS 2-style is using `#ifndef` and `#define` preprocessor directives.
    2. include `"controller_interface/controller_interface.hpp"`.
    3. Define a unique namespace for your controller. This is usually a package name written in `snake_case`.
    4. Define the class of the controller, extending `ControllerInterface`. Now, your code should look similar to this:

```
#ifndef <CONTROLLER_NAME>_HPP_
#define <CONTROLLER_NAME>_HPP_

#include "controller_interface/controller_interface.hpp"

namespace <controller_name>
{

class ControllerName : public controller_interface::ControllerInterface
{
  // ...
};

}  // namespace <controller_name>

#endif  // <CONTROLLER_NAME>_HPP_
```

    5. Add a constructor without parameters and the following public methods overriding the `ControllerInterface` definition: `on_init`, `command_interface_configuration`, `state_interface_configuration`, `on_configure`, `on_activate`, `on_deactivate`, `update`. For exact definitions check the `controller_interface/controller_interface.hpp` header or one of the controllers from ros2_controllers.
    6. (Optional) The NodeOptions of the LifecycleNode can be personalized by overriding the default method `define_custom_node_options`.
    7. (Optional) Often, controllers accept lists of joint names and interface names as parameters. If so, you can add two protected string vectors to store those values.
4. **Adding definitions into source file (.cpp)**
    1. Include the header file of your controller and add a namespace definition to simplify further development.
    2. (optional) Implement a constructor if needed. There, you could initialize member variables. This could also be done in the `on_init` method.
    3. Implement the `on_init` method. The first line usually calls the parent `on_init` method. Here is the best place to initialize the variables, reserve memory, and most importantly, declare node parameters used by the controller. If everything works fine return `controller_interface::CallbackReturn::SUCCESS` or `controller_interface::CallbackReturn::ERROR` otherwise.
    4. Write the `on_configure` method. Parameters are usually read here, and everything is prepared so that the controller can be started.
    5. Implement `command_interface_configuration` and `state_interface_configuration` where required interfaces are defined. There are three options of the interface configuration `ALL`, `INDIVIDUAL`, and `NONE` defined in `controller_interface/controller_interface.hpp"`. `ALL` and `NONE` option will ask for access to all available interfaces or none of them. The `INDIVIDUAL` configuration needs a detailed list of required interface names. Those are usually provided as parameters. The full

interface names have structure `<joint_name>/<interface_type>`.

6. Implement the `on_activate` method with checking, and potentially sorting, the interfaces and assigning members' initial values. This method is part of the real-time loop, therefore avoid any reservation of memory and, in general, keep it as short as possible.
7. Implement the `on_deactivate` method, which does the opposite of `on_activate`. In many cases, this method is empty. This method should also be real-time safe as much as possible.
8. Implement the `update` method as the main entry point. The method should be implemented with real-time constraints in mind. When this method is called, the state interfaces have the most recent values from the hardware, and new commands for the hardware should be written into command interfaces.
9. IMPORTANT: At the end of your file after the namespace is closed, add the `PLUGINLIB_EXPORT_CLASS` macro. For this you will need to include the `"pluginlib/class_list_macros.hpp"` header. As first parameters you should provide exact controller class, e.g., `<controller_name_namespace>::<ControllerName>`, and as second the base class, i.e., `controller_interface::ControllerInterface`.

5. **Writing export definition for pluginlib**
   1. Create the `<controller_name>.xml` file in the package and add a definition of the library and controller's class which has to be visible for the pluginlib. The easiest way to do that is to check other controllers in the ros2_controllers package.
   2. Usually, the plugin name is defined by the package (namespace) and the class name, e.g., `<controller_name_package>/<ControllerName>`. This name defines the controller's type when the controller manager searches for it. The other two parameters have to correspond to the definition done in macro at the bottom of the `<controller_name>.cpp` file.
6. **Writing simple test to check if the controller can be found and loaded**
   1. Create the folder `test` in your package, if it does not exist already, and add a file named `test_load_<controller_name>.cpp`.
   2. You can safely copy the file's content for any controller defined in the ros2_controllers package.
   3. Change the name of the copied test and in the last line, where controller type is specified put the name defined in `<controller_name>.xml` file, e.g., `<controller_name_package>/<ControllerName>`.
7. **Add compile directives into ``CMakeLists.txt`` file**
   1. Under the line `find_package(ament_cmake REQUIRED)` add further dependencies. Those are at least: `controller_interface`, `hardware_interface`, `pluginlib`, `rclcpp` and `rclcpp_lifecycle`.
   2. Add a compile directive for a shared library providing the `<controller_name>.cpp` file as the source.
   3. Add targeted include directories for the library. This is usually only `include`.
   4. Add ament dependencies needed by the library. You should add at least those listed under 1.
   5. Export for pluginlib description file using the following command:

```
pluginlib_export_plugin_description_file(controller_interface <controller_
↪name>.xml)
```

   6. Add install directives for targets and include directory.
   7. In the test section add the following dependencies: `ament_cmake_gmock`, `controller_manager`, `hardware_interface`, `ros2_control_test_assets`.

8. Add compile definitions for the tests using the `ament_add_gmock` directive. For details, see how it is done for controllers in the ros2_controllers package.
9. (optional) Add your controller`s library into `ament_export_libraries` before `ament_package()`.

8. **Add dependencies into ``package.xml`` file**
    1. Add at least the following packages into `<depend>` tag: `controller_interface`, `hardware_interface`, `pluginlib`, `rclcpp` and `rclcpp_lifecycle`.
    2. Add at least the following package into `<test_depend>` tag: `ament_add_gmock`, `controller_manager`, `hardware_interface`, and `ros2_control_test_assets`.
9. **Compiling and testing the controller**
    1. Now everything is ready to compile the controller using the `colcon build --packages-select <controller_name_package>` command. Remember to go into the root of your workspace before executing this command.
    2. If compilation was successful, source the `setup.bash` file from the install folder and execute `colcon test --packages-select <controller_name_package>` to check if the new controller can be found through `pluginlib` library and be loaded by the controller manager.

That's it! Enjoy writing great controllers!

Useful External References

- Templates and scripts for generating controllers shell

Note: The script is currently only recommended to use with Humble, not compatible with the API from Jazzy and onwards.

3.2 Controllers for Wheeled Mobile Robots

3.2.1 diff_drive_controller

Controller for mobile robots with differential drive.

As input it takes velocity commands for the robot body, which are translated to wheel commands for the differential drive base.

Odometry is computed from hardware feedback and published.

For an introduction to mobile robot kinematics and the nomenclature used here, see *Wheeled Mobile Robot Kinematics*.

Other features

- Realtime-safe implementation.
- Odometry publishing
- Task-space velocity, acceleration and jerk limits
- Automatic stop after command time-out
- Chainable Controller

Description of controller's interfaces

References

When controller is in chained mode, it exposes the following references which can be commanded by the preceding controller:

- `<controller_name>/linear/velocity` double, in m/s
- `<controller_name>/angular/velocity` double, in rad/s

Together, these represent the body twist (which in unchained-mode would be obtained from ~/cmd_vel). The `<controller_name>` is commonly set to `diff_drive_controller`.

Feedback

As feedback interface type the joints' position (`hardware_interface::HW_IF_POSITION`) or velocity (`hardware_interface::HW_IF_VELOCITY`,if parameter `position_feedback=false`) are used. Unless the parameter `open_loop=true` is set, then no external state interfaces are used (the commanded velocity is used for odometry instead).

Output

Joints' velocity (`hardware_interface::HW_IF_VELOCITY`) are used.

ROS 2 Interfaces

Subscribers

~/cmd_vel [geometry_msgs/msg/TwistStamped]
: Velocity command for the controller. The controller extracts the x component of the linear velocity and the z component of the angular velocity. Velocities on other components are ignored.

Publishers

~/odom [nav_msgs::msg::Odometry]
This represents an estimate of the robot's position and velocity in free space.

/tf [tf2_msgs::msg::TFMessage]
tf tree. Published only if `enable_odom_tf=true`

~/cmd_vel_out [geometry_msgs/msg/TwistStamped]
Velocity command for the controller, where limits were applied. Published only if `publish_limited_velocity=true`

Services

~/set_odometry [control_msgs::srv::SetOdometry]
This service can be used to set the current odometry of the robot to desired values.

Parameters

This controller uses the generate_parameter_library to handle its parameters. The parameter definition file located in the src folder contains descriptions for all the parameters used by the controller.

left_wheel_names (string_array)
Names of the left side wheels' joints

Default: {}

Constraints:

- parameter is not empty

right_wheel_names (string_array)
Names of the right side wheels' joints

Default: {}

Constraints:

- parameter is not empty

wheel_separation (double)
Shortest distance between the left and right wheels. If this parameter is wrong, the robot will not behave correctly in curves.

Default: 0.0

Constraints:

- greater than 0.0

wheel_radius (double)
Radius of a wheel, i.e., wheels size, used for transformation of linear velocity into wheel rotations. If this parameter is wrong the robot will move faster or slower than expected.

Default: 0.0

Constraints:

- greater than 0.0

wheel_separation_multiplier (double)
Correction factor when the actual wheel separation differs from the nominal value in the `wheel_separation` parameter.

Default: 1.0

left_wheel_radius_multiplier (double)
Correction factor when radius of left wheels differs from the nominal value in `wheel_radius` parameter.

Default: 1.0

right_wheel_radius_multiplier (double)
Correction factor when radius of right wheels differs from the nominal value in `wheel_radius` parameter.

Default: 1.0

tf_frame_prefix_enable (bool)
Deprecated: this parameter will be removed in a future release. Use 'tf_frame_prefix' instead.

Default: true

tf_frame_prefix (string)
(optional) Prefix to be appended to the tf frames, will be added to odom_id and base_frame_id before publishing. If the parameter is empty, controller's namespace will be used.

Default: ""

odom_frame_id (string)
Name of the frame for odometry. This frame is parent of `base_frame_id` when controller publishes odometry.

Default: "odom"

base_frame_id (string)
Name of the robot's base frame that is child of the odometry frame.

Default: "base_link"

pose_covariance_diagonal (double_array)
Odometry covariance for the encoder output of the robot for the pose. These values should be tuned to your robot's sample odometry data, but these values are a good place to start: `[0.001, 0.001, 0.001, 0.001, 0.001, 0.01]`.

Default: {0.0, 0.0, 0.0, 0.0, 0.0, 0.0}

twist_covariance_diagonal (double_array)
Odometry covariance for the encoder output of the robot for the speed. These values should be tuned to your robot's sample odometry data, but these values are a good place to start: `[0.001, 0.001, 0.001, 0.001, 0.001, 0.01]`.

Default: {0.0, 0.0, 0.0, 0.0, 0.0, 0.0}

open_loop (bool)
If set to true the odometry of the robot will be calculated from the commanded values and not from feedback.

Default: false

position_feedback (bool)
Is there position feedback from hardware.

Default: true

enable_odom_tf (bool)
Publish transformation between `odom_frame_id` and `base_frame_id`.

Default: true

cmd_vel_timeout (double)
Timeout in seconds, after which input command on `cmd_vel` topic is considered staled.

Default: 0.5

publish_limited_velocity (bool)
Publish limited velocity value.

Default: false

velocity_rolling_window_size (int)
Size of the rolling window for calculation of mean velocity use in odometry.

Default: 10

publish_rate (double)
Publishing rate (Hz) of the odometry and TF messages. This parameter is deprecated and will be removed in a future release.

Default: 50.0

linear.x
Joint limits structure for the linear `x`-axis.

linear.x.max_velocity (double)
Default: std::numeric_limits<double>::quiet_NaN()

Constraints:

- Custom validator: control_filters::gt_eq_or_nan: 0.0

linear.x.min_velocity (double)
Default: std::numeric_limits<double>::quiet_NaN()

Constraints:

- Custom validator: control_filters::lt_eq_or_nan: 0.0

linear.x.max_acceleration (double)
Maximum acceleration in forward direction.

Default: std::numeric_limits<double>::quiet_NaN()

Constraints:

- Custom validator: control_filters::gt_eq_or_nan: 0.0

linear.x.max_deceleration (double)
Maximum deceleration in forward direction.

Default: std::numeric_limits<double>::quiet_NaN()

Constraints:

- Custom validator: control_filters::lt_eq_or_nan: 0.0

linear.x.max_acceleration_reverse (double)
Maximum acceleration in reverse direction. If not set, -max_acceleration will be used.

Default: std::numeric_limits<double>::quiet_NaN()

Constraints:

- Custom validator: control_filters::lt_eq_or_nan: 0.0

linear.x.max_deceleration_reverse (double)
Maximum deceleration in reverse direction. If not set, -max_deceleration will be used.

Default: std::numeric_limits<double>::quiet_NaN()

Constraints:

- Custom validator: control_filters::gt_eq_or_nan: 0.0

linear.x.max_jerk (double)
Default: std::numeric_limits<double>::quiet_NaN()

Constraints:

- Custom validator: control_filters::gt_eq_or_nan: 0.0

linear.x.min_jerk (double)
Default: std::numeric_limits<double>::quiet_NaN()

Constraints:

- Custom validator: control_filters::lt_eq_or_nan: 0.0

angular.z
Joint limits structure for the rotation about `z`-axis.

angular.z.max_velocity (double)
Default: std::numeric_limits<double>::quiet_NaN()

Constraints:

- Custom validator: control_filters::gt_eq_or_nan: 0.0

angular.z.min_velocity (double)
Default: std::numeric_limits<double>::quiet_NaN()

Constraints:

- Custom validator: control_filters::lt_eq_or_nan: 0.0

angular.z.max_acceleration (double)
Maximum acceleration in positive direction.

Default: std::numeric_limits<double>::quiet_NaN()

Constraints:

- Custom validator: control_filters::gt_eq_or_nan: 0.0

angular.z.max_deceleration (double)
Maximum deceleration in positive direction.

Default: std::numeric_limits<double>::quiet_NaN()

Constraints:

- Custom validator: control_filters::lt_eq_or_nan: 0.0

angular.z.max_acceleration_reverse (double)
Maximum acceleration in negative direction. If not set, -max_acceleration will be used.

Default: std::numeric_limits<double>::quiet_NaN()

Constraints:

- Custom validator: control_filters::lt_eq_or_nan: 0.0

angular.z.max_deceleration_reverse (double)
Maximum deceleration in negative direction. If not set, -max_deceleration will be used.

Default: std::numeric_limits<double>::quiet_NaN()

Constraints:

- Custom validator: control_filters::gt_eq_or_nan: 0.0

angular.z.max_jerk (double)
Default: std::numeric_limits<double>::quiet_NaN()

Constraints:

- Custom validator: control_filters::gt_eq_or_nan: 0.0

angular.z.min_jerk (double)
Default: std::numeric_limits<double>::quiet_NaN()

Constraints:

- Custom validator: control_filters::lt_eq_or_nan: 0.0

An example parameter file for this controller can be found in the test directory:

```yaml
test_diff_drive_controller:
  ros__parameters:
    left_wheel_names: ["left_wheels"]
    right_wheel_names: ["right_wheels"]

    wheel_separation: 0.40
    wheel_radius: 0.02

    wheel_separation_multiplier: 1.0
    left_wheel_radius_multiplier: 1.0
    right_wheel_radius_multiplier: 1.0

    odom_frame_id: odom
    base_frame_id: base_link
    pose_covariance_diagonal: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
    twist_covariance_diagonal: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

    position_feedback: false
    open_loop: true
    enable_odom_tf: true

    cmd_vel_timeout: 0.5 # seconds
    publish_limited_velocity: true
    velocity_rolling_window_size: 10

    linear.x.max_velocity: .NAN
    linear.x.min_velocity: .NAN
    linear.x.max_acceleration: .NAN
    linear.x.max_deceleration: .NAN
    linear.x.max_acceleration_reverse: .NAN
    linear.x.max_deceleration_reverse: .NAN
    linear.x.max_jerk: .NAN
    linear.x.min_jerk: .NAN

    angular.z.max_velocity: .NAN
    angular.z.min_velocity: .NAN
```

(continues on next page)

(continued from previous page)

```
    angular.z.max_acceleration: .NAN
    angular.z.max_deceleration: .NAN
    angular.z.max_acceleration_reverse: .NAN
    angular.z.max_deceleration_reverse: .NAN
    angular.z.max_jerk: .NAN
    angular.z.min_jerk: .NAN
```

3.2.2 mecanum_drive_controller

Library with shared functionalities for mobile robot controllers with mecanum drive (four mecanum wheels). The library implements generic odometry and update methods and defines the main interfaces.

Execution logic of the controller

The controller uses velocity input, i.e., stamped Twist messages where linear `x`, `y`, and angular `z` components are used. Values in other components are ignored. In the chain mode, the controller provides three reference interfaces, one for linear velocity and one for steering angle position. Other relevant features are:

- odometry publishing as Odometry and TF message;
- input command timeout based on a parameter.

Note about odometry calculation: In the DiffDRiveController, the velocity is filtered out, but we prefer to return it raw and let the user perform post-processing at will. We prefer this way of doing so as filtering introduces delay (which makes it difficult to interpret and compare behavior curves).

Description of controller's interfaces

References (from a preceding controller)

When controller is in chained mode, it exposes the following references which can be commanded by the preceding controller:

- `<controller_name>/linear/x/velocity`, in m/s
- `<controller_name>/linear/y/velocity`, in m/s
- `<controller_name>/angular/z/velocity`, in rad/s

Commands

- `<*_wheel_command_joint_name>/velocity`, in rad/s

States

- `<joint_name>/velocity`, in rad/s

Note: `joint_name` can be of `*_wheel_state_joint_name` parameter (if used), `*_wheel_command_joint_name` otherwise.

Subscribers

Used when the controller is not in chained mode (`in_chained_mode == false`).

- `<controller_name>/reference` [`geometry_msgs/msg/TwistStamped`]

Publishers

- `<controller_name>/odometry` [`nav_msgs/msg/Odometry`]
- `<controller_name>/tf_odometry` [`tf2_msgs/msg/TFMessage`]
- `<controller_name>/controller_state` [`control_msgs/msg/MecanumDriveControllerState`]

Services

~/set_odometry [control_msgs::srv::SetOdometry]
This service can be used to set the current odometry of the robot to desired values.

Parameters

This controller uses the generate_parameter_library to handle its parameters. The parameter definition file located in the src folder contains descriptions for all the parameters used by the controller.

reference_timeout (double)
Timeout for controller references after which they will be reset. This is especially useful for controllers that can cause unwanted and dangerous behavior if reference is not reset, e.g., velocity controllers. If value is 0 the reference is reset after each run.

Default: 0.0

front_left_wheel_command_joint_name (string)
Name of the joints for commanding front left wheel.

Read only: True

Default: “”

Constraints:

- parameter is not empty

front_right_wheel_command_joint_name (string)
Name of the joints for commanding front right wheel.

Read only: True

Default: “”

Constraints:

- parameter is not empty

rear_right_wheel_command_joint_name (string)
Name of the joints for commanding rear right wheel.

Read only: True

Default: “”

Constraints:

- parameter is not empty

rear_left_wheel_command_joint_name (string)
Name of the joints for commanding rear left wheel.

Read only: True

Default: “”

Constraints:

- parameter is not empty

front_left_wheel_state_joint_name (string)
(optional) Specifies a joint name for reading states of front left wheel. This parameter is only relevant when state joints are different then command joint, i.e., when a following controller is used.

Read only: True

Default: “”

front_right_wheel_state_joint_name (string)
(optional) Specifies a joint name for reading states of front right wheel. This parameter is only relevant when state joints are different then command joint, i.e., when a following controller is used.

Read only: True

Default: “”

rear_right_wheel_state_joint_name (string)
(optional) Specifies a joint name for reading states of rear right wheel. This parameter is only relevant when state joints are different then command joint, i.e., when a following controller is used.

Read only: True

Default: “”

rear_left_wheel_state_joint_name (string)
(optional) Specifies a joint name for reading states of rear left wheel. This parameter is only relevant when state joints are different then command joint, i.e., when a following controller is used.

Read only: True

Default: “”

kinematics.base_frame_offset.x (double)
Base frame offset along X axis of base_frame (base_link frame).

Read only: True

Default: 0.0

kinematics.base_frame_offset.y (double)
Base frame offset along Y axis of base_frame (base_link frame).

Read only: True

Default: 0.0

kinematics.base_frame_offset.theta (double)
Base frame offset along Theta axis of base_frame (base_link frame).

Read only: True

Default: 0.0

kinematics.wheels_radius (double)
Wheel's radius.

Default: 0.0

Constraints:

- greater than 0.0

kinematics.sum_of_robot_center_projection_on_X_Y_axis (double)
Wheels geometric param used in mecanum wheels' IK. lx and ly represent the distance from the robot's center to the wheels projected on the x and y axis with origin at robots center respectively, sum_of_robot_center_projection_on_X_Y_axis = lx+ly

Default: 0.0

tf_frame_prefix_enable (bool)
Deprecated: this parameter will be removed in a future release. Use 'tf_frame_prefix' instead.

Default: true

tf_frame_prefix (string)
(optional) Prefix to be appended to the tf frames, will be added to odom_frame_id and base_frame_id before publishing. If the parameter is empty, controller's namespace will be used.

Default: ""

base_frame_id (string)
Base frame_id set to value of base_frame_id.

Default: "base_link"

odom_frame_id (string)
Odometry frame_id set to value of odom_frame_id.

Default: "odom"

enable_odom_tf (bool)
Publishing to tf is enabled or disabled?

Default: true

twist_covariance_diagonal (double_array)
Diagonal values of twist covariance matrix.

Default: {0.1, 0.1, 0.1, 0.1, 0.1, 0.1}

pose_covariance_diagonal (double_array)
Diagonal values of pose covariance matrix.

Default: {0.1, 0.1, 0.1, 0.1, 0.1, 0.1}

An example parameter file for this controller can be found in the test directory:

```yaml
/**:
  test_mecanum_drive_controller:
    ros__parameters:
      reference_timeout: 0.9

      front_left_wheel_command_joint_name: "front_left_wheel_joint"
      front_right_wheel_command_joint_name: "front_right_wheel_joint"
      rear_right_wheel_command_joint_name: "back_right_wheel_joint"
      rear_left_wheel_command_joint_name: "back_left_wheel_joint"

      kinematics:
        base_frame_offset: { x: 0.0, y: 0.0, theta: 0.0 }
        wheels_radius: 0.5
        sum_of_robot_center_projection_on_X_Y_axis: 1.0

      base_frame_id: "base_link"
      odom_frame_id: "odom"
      enable_odom_tf: true
      twist_covariance_diagonal: [0.0, 7.0, 14.0, 21.0, 28.0, 35.0]
      pose_covariance_diagonal: [0.0, 6.0, 12.0, 18.0, 24.0, 30.0]


  test_mecanum_drive_controller_with_rotation:
    ros__parameters:
      reference_timeout: 5.0

      front_left_wheel_command_joint_name: "front_left_wheel_joint"
      front_right_wheel_command_joint_name: "front_right_wheel_joint"
      rear_right_wheel_command_joint_name: "rear_right_wheel_joint"
      rear_left_wheel_command_joint_name: "rear_left_wheel_joint"

      kinematics:
        base_frame_offset: { x: 1.0, y: 2.0, theta: 3.0 }
        wheels_radius: 0.05
        sum_of_robot_center_projection_on_X_Y_axis: 0.2925

      base_frame_id: "base_link"
      odom_frame_id: "odom"
      enable_odom_tf: true
      pose_covariance_diagonal: [0.001, 0.001, 0.001, 0.001, 0.001, 0.001]
      twist_covariance_diagonal: [0.001, 0.001, 0.001, 0.001, 0.001, 0.001]
```

3.2.3 omni_wheel_drive_controller

Controller for mobile robots with omnidirectional drive.

Supports using three or more omni wheels spaced at an equal angle from each other in a circular formation. To better understand this, have a look at *Wheeled Mobile Robot Kinematics*.

The controller uses velocity input, i.e., stamped Twist messages where linear `x`, `y`, and angular `z` components are used. Values in other components are ignored.

Odometry is computed from hardware feedback and published.

Other features

- Realtime-safe implementation.
- Odometry publishing
- Automatic stop after command time-out

Description of controller's interfaces

References (from a preceding controller)

When controller is in chained mode, it exposes the following references which can be commanded by the preceding controller:

- `<controller_name>/linear/x/velocity` double, in m/s
- `<controller_name>/linear/y/velocity` double, in m/s
- `<controller_name>/angular/z/velocity` double, in rad/s

Together, these represent the body twist (which in unchained-mode would be obtained from ~/cmd_vel).

State interfaces

As feedback interface type the joints' position (`hardware_interface::HW_IF_POSITION`) or velocity (`hardware_interface::HW_IF_VELOCITY`, if parameter `position_feedback=false`) are used.

Command interfaces

Joints' velocity (`hardware_interface::HW_IF_VELOCITY`) are used.

ROS 2 Interfaces

Subscribers

~/cmd_vel [geometry_msgs/msg/TwistStamped]
: Velocity command for the controller. The controller extracts the x and y component of the linear velocity and the z component of the angular velocity. Velocities on other components are ignored.

Publishers

~/odom [nav_msgs::msg::Odometry]
: This represents an estimate of the robot's position and velocity in free space.

/tf [tf2_msgs::msg::TFMessage]
: tf tree. Published only if `enable_odom_tf=true`

Services

~/set_odometry [control_msgs::srv::SetOdometry]
This service can be used to set the current odometry of the robot to desired values.

Parameters

This controller uses the generate_parameter_library to handle its parameters. The parameter definition file located in the src folder contains descriptions for all the parameters used by the controller.

wheel_offset (double)
Angular offset (radians) of the first wheel from the positive direction of the x-axis of the robot.

Read only: True

wheel_names (string_array)
Names of the wheels' joints, given in an anti-clockwise order starting from the wheel aligned with the positive direction of the x-axis of the robot / offset from it by the value specified in `wheel_offset`.

Read only: True

Constraints:

- parameter is not empty

robot_radius (double)
Radius of the robot, distance between the center of the robot and the wheels. If this parameter is wrong, the robot will not behave correctly in curves.

Read only: True

Constraints:

- greater than 0.0

wheel_radius (double)
Radius of a wheel, i.e., wheels size, used for transformation of linear velocity into wheel rotations. If this parameter is wrong the robot will move faster or slower than expected.

Read only: True

Constraints:

- greater than 0.0

tf_frame_prefix_enable (bool)
Deprecated: this parameter will be removed in a future release. Use 'tf_frame_prefix' instead.

Read only: True

Default: true

tf_frame_prefix (string)
(optional) Prefix to be appended to the tf frames, will be added to odom_id and base_frame_id before publishing. If the parameter is empty, controller's namespace will be used.

Read only: True

Default: ""

odom_frame_id (string)
Name of the frame for odometry. This frame is parent of `base_frame_id` when controller publishes odometry.

Read only: True

Default: “odom”

base_frame_id (string)
Name of the robot’s base frame that is child of the odometry frame.

Read only: True

Default: “base_link”

diagonal_covariance.pose (double_array)
Odometry covariance for the encoder output of the robot for the pose. These values should be tuned to your robot’s sample odometry data, but these values are a good place to start: `[0.001, 0.001, 0.001, 0.001, 0.001, 0.01]`.

Read only: True

Default: {0.0, 0.0, 0.0, 0.0, 0.0, 0.0}

diagonal_covariance.twist (double_array)
Odometry covariance for the encoder output of the robot for the speed. These values should be tuned to your robot’s sample odometry data, but these values are a good place to start: `[0.001, 0.001, 0.001, 0.001, 0.001, 0.01]`.

Read only: True

Default: {0.0, 0.0, 0.0, 0.0, 0.0, 0.0}

open_loop (bool)
If set to true the odometry of the robot will be calculated from the commanded values and not from feedback.

Read only: True

Default: false

position_feedback (bool)
Only valid if `open_loop` is set to false. If there is position feedback from the hardware, set the parameter to `true`, else set it to `false`.

Read only: True

Default: true

enable_odom_tf (bool)
Publish transformation between `odom_frame_id` and `base_frame_id`.

Read only: True

Default: true

cmd_vel_timeout (double)
Timeout in seconds, after which input command on `~/cmd_vel` topic is considered stale.

Read only: True

Default: 0.5

An example parameter file for this controller can be found in the test directory:

```
test_omni_wheel_drive_controller:
  ros__parameters:
    wheel_offset: 0.0
    wheel_names:
      [
        "front_wheel_joint",
        "left_wheel_joint",
```

(continues on next page)

(continued from previous page)

```
        "back_wheel_joint",
        "right_wheel_joint",
      ]

    robot_radius: 0.20
    wheel_radius: 0.02

    odom_frame_id: odom
    base_frame_id: base_link
    pose_covariance_diagonal: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
    twist_covariance_diagonal: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

    open_loop: true
    position_feedback: false
    enable_odom_tf: true

    cmd_vel_timeout: 0.5
    publish_limited_velocity: true
```

3.2.4 steering_controllers_library

Library with shared functionalities for mobile robot controllers with steering drives (2 degrees of freedom), with so-called non-holonomic constraints.

The library implements generic odometry and update methods and defines the main interfaces.

The update methods only use inverse kinematics, it does not implement any feedback control loops like path-tracking controllers etc.

For an introduction to mobile robot kinematics and the nomenclature used here, see *Wheeled Mobile Robot Kinematics*.

Execution logic of the controller

The controller uses velocity input, i.e., stamped twist messages where linear `x` and angular `z` components are used. Values in other components are ignored.

In the chain mode the controller provides two reference interfaces, one for linear velocity and one for steering angle position. Other relevant features are:

- support for front and rear steering configurations;
- odometry publishing as Odometry and TF message;
- input command timeout based on a parameter.

The command for the wheels are calculated using `odometry` library where based on concrete kinematics traction and steering commands are calculated.

Currently implemented kinematics

- Bicycle - with one steering and one drive joints;
- Tricycle - with one steering and two drive joints;
- Ackermann - with two steering and two drive joints.

bicycle_steering_controller

This controller implements the kinematics with two axes and two wheels, where the wheel on one axis is fixed (traction/drive), and the wheel on the other axis is steerable.

The controller expects to have one commanding joint for traction, and one commanding joint for steering. If your Ackermann steering vehicle uses differentials on axes, then you should probably use this controller since you can command only one traction velocity and steering angle for virtual wheels in the middle of the axes.

For more details on controller's execution and interfaces check the *Steering Controller Library*.

Parameters

This controller uses the generate_parameter_library to handle its parameters.

For an exemplary parameterization see the `test` folder of the controller's package.

Additionally to the parameters of the *Steering Controller Library*, this controller adds the following parameters:

wheelbase (double)
Distance between front and rear wheel. For details see: https://en.wikipedia.org/wiki/Wheelbase

Default: 0.0

Constraints:

- greater than 0.0

traction_wheel_radius (double)
Traction wheel radius.

Default: 0.0

Constraints:

- greater than 0.0

tricycle_steering_controller

This controller implements the kinematics with two axes and three wheels, where two wheels on an axis are fixed (traction/drive), and the wheel on the other axis is steerable.

The controller expects to have two commanding joints for traction, one for each fixed wheel and one commanding joint for steering.

For more details on controller's execution and interfaces check the *Steering Controller Library*.

Parameters

This controller uses the generate_parameter_library to handle its parameters.

For an exemplary parameterization see the `test` folder of the controller's package.

Additionally to the parameters of the *Steering Controller Library*, this controller adds the following parameters:

traction_track_width (double)
Axle track

Default: 0.0

Constraints:

- greater than 0.0

wheelbase (double)
Distance between front and rear wheels. For details see: https://en.wikipedia.org/wiki/Wheelbase

Default: 0.0

Constraints:

- greater than 0.0

traction_wheels_radius (double)
Traction wheels radius.

Default: 0.0

Constraints:

- greater than 0.0

ackermann_steering_controller

This controller implements the kinematics with two axes and four wheels, where the wheels on one axis are fixed (traction/drive) wheels, and the wheels on the other axis are steerable.

The controller expects to have two commanding joints for traction, one for each fixed wheel and two commanding joints for steering, one for each wheel.

For more details on controller's execution and interfaces check the *Steering Controller Library*.

Parameters

This controller uses the generate_parameter_library to handle its parameters.

For an exemplary parameterization see the `test` folder of the controller's package.

Additionally to the parameters of the *Steering Controller Library*, this controller adds the following parameters:

steering_track_width (double)
(Optional) Steering wheel track length. If not set, 'traction_track_width' will be used.

Default: 0.0

traction_track_width (double)
Traction wheel track length. For details see: https://en.wikipedia.org/wiki/Wheelbase

Default: 0.0

Constraints:

- greater than 0.0

wheelbase (double)
Distance between front and rear wheels. For details see: https://en.wikipedia.org/wiki/Wheelbase

Default: 0.0

Constraints:

- greater than 0.0

traction_wheels_radius (double)
Traction wheels radius.

Default: 0.0

Constraints:

- greater than 0.0

Description of controller's interfaces

References (from a preceding controller)

Used when controller is in chained mode (`in_chained_mode == true`).

- `<controller_name>/linear/velocity` double, in m/s
- `<controller_name>/angular/velocity` double, in rad/s

representing the body twist.

Command interfaces

- `<steering_joints_names[i]>/position` double, in rad
- `<traction_joints_names[i]>/velocity` double, in rad/s

State interfaces

Depending on the `position_feedback`, different feedback types are expected

- `position_feedback == true` -> `TRACTION_FEEDBACK_TYPE = position`
- `position_feedback == false` -> `TRACTION_FEEDBACK_TYPE = velocity`

With the following state interfaces:

- `<steering_joints_names[i]>/position` double, in rad
- `<traction_joints_names[i]>/<TRACTION_FEEDBACK_TYPE>` double, in rad or rad/s

Subscribers

Used when controller is not in chained mode (`in_chained_mode == false`).

- `<controller_name>/reference` [geometry_msgs/msg/TwistStamped]

Publishers

- `<controller_name>/odometry` [nav_msgs/msg/Odometry]
- `<controller_name>/tf_odometry` [tf2_msgs/msg/TFMessage]
- `<controller_name>/controller_state` [control_msgs/msg/SteeringControllerStatus]

Services

- `~/set_odometry` [control_msgs/srv/SetOdometry]

Parameters

This controller uses the generate_parameter_library to handle its parameters.

For an exemplary parameterization see the `test` folder of the controller's package.

reference_timeout (double)
Timeout for controller references after which they will be reset. This is especially useful for controllers that can cause unwanted and dangerous behavior if reference is not reset, e.g., velocity controllers. If value is 0 the reference is reset after each run.

Default: 1.0

traction_joints_names (string_array)
Names of traction wheel joints. For kinematic configurations with two traction joints, the expected order is: right joint, left joint.

Read only: True

Default: {}

Constraints:

- length is less than 5
- contains no duplicates
- parameter is not empty

steering_joints_names (string_array)
Names of steering joints. For kinematic configurations with two steering joints, the expected order is: right joint, left joint. The orientation of the steering axes is expected as such: When positive steering position value is commanded, then the robot should turn in positive direction of the z-axis of the vehicle (see REP-103).

Read only: True

Default: {}

Constraints:

- length is less than 5

- contains no duplicates
- parameter is not empty

traction_joints_state_names (string_array)
(Optional) Names of tractions joints to read states from. If not set joint names from 'traction_joints_names' will be used.

Read only: True

Default: {}

Constraints:

- length is less than 5
- contains no duplicates

steering_joints_state_names (string_array)
(Optional) Names of steering joints to read states from. If not set joint names from 'steering_joints_names' will be used.

Read only: True

Default: {}

Constraints:

- length is less than 5
- contains no duplicates

open_loop (bool)
Choose if open-loop or not (feedback) is used for odometry calculation.

Default: false

reduce_wheel_speed_until_steering_reached (bool)
Reduce wheel speed until the steering angle has been reached.

Default: false

velocity_rolling_window_size (int)
The number of velocity samples to average together to compute the odometry twist.linear.x and twist.angular.z velocities.

Default: 10

base_frame_id (string)
Base frame_id set to value of base_frame_id.

Default: "base_link"

odom_frame_id (string)
Odometry frame_id set to value of odom_frame_id.

Default: "odom"

enable_odom_tf (bool)
Publishing to tf is enabled or disabled?

Default: true

tf_frame_prefix (string)
(optional) Prefix to be prepended to odom_id and base_frame_id tf frames before publishing. Tilde('~') character in this prefix will be replaced with the node namespace.

Read only: True

Default: “”

twist_covariance_diagonal (double_array)
diagonal values of twist covariance matrix.

Default: {0.0, 7.0, 14.0, 21.0, 28.0, 35.0}

pose_covariance_diagonal (double_array)
diagonal values of pose covariance matrix.

Default: {0.0, 7.0, 14.0, 21.0, 28.0, 35.0}

position_feedback (bool)
Choice of feedback type, if position_feedback is false then `HW_IF_VELOCITY` is taken as interface type, if position_feedback is true then `HW_IF_POSITION` is taken as interface type

Default: false

3.2.5 tricycle_controller

Controller for mobile robots with a single double-actuated wheel, including traction and steering. An example is a tricycle robot with the actuated wheel in the front and two trailing wheels on the rear axle.

Input for control are robot base_link twist commands which are translated to traction and steering commands for the tricycle drive base. Odometry is computed from hardware feedback and published.

For an introduction to mobile robot kinematics and the nomenclature used here, see *Wheeled Mobile Robot Kinematics*.

Other features

Realtime-safe implementation. Odometry publishing Velocity, acceleration and jerk limits Automatic stop after command timeout

ROS 2 Interfaces

Subscribers

~/cmd_vel [geometry_msgs/msg/TwistStamped]
Velocity command for the controller. The controller extracts the x component of the linear velocity and the z component of the angular velocity. Velocities on other components are ignored.

Parameters

This controller uses the generate_parameter_library to handle its parameters.

traction_joint_name (string)
Name of the traction joint

Default: “”

Constraints:

- parameter is not empty

steering_joint_name (string)
Name of the steering joint

Default: “”

Constraints:

- parameter is not empty

wheelbase (double)
Shortest distance between the front wheel and the rear axle. If this parameter is wrong, the robot will not behave correctly in curves.

Default: 0.0

Constraints:

- greater than 0.0

wheel_radius (double)
Radius of a wheel, i.e., wheels size, used for transformation of linear velocity into wheel rotations. If this parameter is wrong the robot will move faster or slower then expected.

Default: 0.0

Constraints:

- greater than 0.0

odom_frame_id (string)
Name of the frame for odometry. This frame is parent of `base_frame_id` when controller publishes odometry.

Default: “odom”

base_frame_id (string)
Name of the robot’s base frame that is child of the odometry frame.

Default: “base_link”

pose_covariance_diagonal (double_array)
Odometry covariance for the encoder output of the robot for the pose. These values should be tuned to your robot’s sample odometry data, but these values are a good place to start: `[0.001, 0.001, 0.001, 0.001, 0.001, 0.01]`.

Default: {0.0, 0.0, 0.0, 0.0, 0.0, 0.0}

twist_covariance_diagonal (double_array)
Odometry covariance for the encoder output of the robot for the speed. These values should be tuned to your robot’s sample odometry data, but these values are a good place to start: `[0.001, 0.001, 0.001, 0.001, 0.001, 0.01]`.

Default: {0.0, 0.0, 0.0, 0.0, 0.0, 0.0}

open_loop (bool)
If set to true the odometry of the robot will be calculated from the commanded values and not from feedback.

Default: false

enable_odom_tf (bool)
Publish transformation between `odom_frame_id` and `base_frame_id`.

Default: false

odom_only_twist (bool)
for doing the pose integration in separate node.

Default: false

cmd_vel_timeout (int)
Timeout in milliseconds, after which input command on `cmd_vel` topic is considered staled.

Default: 500

publish_ackermann_command (bool)
Publish limited commands.

Default: false

velocity_rolling_window_size (int)
Size of the rolling window for calculation of mean velocity use in odometry.

Default: 10

Constraints:

- greater than 0

traction.max_velocity (double)
Default: std::numeric_limits<double>::quiet_NaN()

traction.min_velocity (double)
Default: std::numeric_limits<double>::quiet_NaN()

traction.max_acceleration (double)
Default: std::numeric_limits<double>::quiet_NaN()

traction.min_acceleration (double)
Default: std::numeric_limits<double>::quiet_NaN()

traction.max_deceleration (double)
Default: std::numeric_limits<double>::quiet_NaN()

traction.min_deceleration (double)
Default: std::numeric_limits<double>::quiet_NaN()

traction.max_jerk (double)
Default: std::numeric_limits<double>::quiet_NaN()

traction.min_jerk (double)
Default: std::numeric_limits<double>::quiet_NaN()

steering.max_position (double)
Default: std::numeric_limits<double>::quiet_NaN()

steering.min_position (double)
Default: std::numeric_limits<double>::quiet_NaN()

steering.max_velocity (double)
Default: std::numeric_limits<double>::quiet_NaN()

steering.min_velocity (double)
Default: std::numeric_limits<double>::quiet_NaN()

steering.max_acceleration (double)
Default: std::numeric_limits<double>::quiet_NaN()

steering.min_acceleration (double)
Default: std::numeric_limits<double>::quiet_NaN()

3.3 Controllers for Manipulators and Other Robots

3.3.1 Admittance Controller

Admittance controller enables you do zero-force control from a force measured on your TCP. The controller implements `ChainedControllerInterface`, so it is possible to add another controllers in front of it, e.g., `JointTrajectoryController`.

The controller requires an external kinematics plugin to function. The kinematics_interface repository provides an interface and an implementation that the admittance controller can use.

ROS 2 interface of the controller

Parameters

The admittance controller uses the generate_parameter_library to handle its parameters. The parameter definition file located in the src folder contains descriptions for all the parameters used by the controller.

joints (string_array)
Specifies which joints will be used by the controller.

Read only: True

command_joints (string_array)
(optional) Specifies the joints for writing into another controllers reference. This parameter is only relevant when chaining the output of this controller to the input of another controller.

Read only: True

Default: {}

command_interfaces (string_array)
Specifies which command interfaces the controller will claim.

Read only: True

state_interfaces (string_array)
Specifies which state interfaces the controller will claim.

Read only: True

chainable_command_interfaces (string_array)
Specifies which reference interfaces the controller will export. Normally, the position and velocity are used.

Read only: True

Default: {“position”, “velocity”}

kinematics.plugin_name (string)
Specifies the name of the kinematics plugin to load.

kinematics.plugin_package (string)
Specifies the package name that contains the kinematics plugin.

kinematics.base (string)
Specifies the base link of the robot description used by the kinematics plugin.

kinematics.tip (string)
Specifies the end effector link of the robot description used by the kinematics plugin.

kinematics.alpha (double)
Specifies the damping coefficient for the Jacobian pseudo inverse.

Default: 0.01

ft_sensor.name (string)
Specifies the name of the force torque sensor in the robot description which will be used in the admittance calculation.

ft_sensor.frame.id (string)
Specifies the frame/link name of the force torque sensor.

ft_sensor.filter_coefficient (double)
Specifies the filter coefficient for the sensor's exponential filter.

Default: 0.05

control.frame.id (string)
Specifies the control frame used for admittance calculation.

fixed_world_frame.frame.id (string)
Specifies the world frame use for admittance calculation. Gravity must point down in this frame.

gravity_compensation.frame.id (string)
Specifies the frame which center of gravity (CoG) is defined in. Normally, the force torque sensor frame should be used.

gravity_compensation.CoG.pos (double_array)
Specifies the position of the center of gravity (CoG) of the end effector in the gravity compensation frame.

Constraints:

- length must be equal to 3

gravity_compensation.CoG.force (double)
Specifies the weight of the end effector, e.g mass * 9.81.

Default: 0.0

admittance.selected_axes (bool_array)
Specifies whether the axes x, y, z, rx, ry, and rz should be included in the admittance calculation.

Constraints:

- length must be equal to 6

admittance.mass (double_array)
Specifies the mass values for x, y, z, rx, ry, and rz used in the admittance calculation.

Constraints:

- length must be equal to 6
- each element of array must be within bounds 0.0001

admittance.damping_ratio (double_array)
Specifies damping ratio values for x, y, z, rx, ry, and rz used in the admittance calculation. The damping ratio is defined as: zeta = D / (2 * sqrt(M * S)).

Constraints:

- length must be equal to 6

admittance.stiffness (double_array)
Specifies the stiffness values for x, y, z, rx, ry, and rz used in the admittance calculation.

Constraints:

- length must be equal to 6
- each element of array must be within bounds 0.0

admittance.joint_damping (double)
Specifies the joint damping applied used in the admittance calculation.

Default: 5.0

Constraints:

- greater than or equal to 0.0

enable_parameter_update_without_reactivation (bool)
If enabled, the parameters will be dynamically updated while the controller is running.

Default: true

An example parameter file for this controller can be found in the test folder:

```yaml
load_admittance_controller:
  # contains minimal parameters that need to be set to load controller
  ros__parameters:
    joints:
      - joint1
      - joint2

    command_interfaces:
      - velocity

    state_interfaces:
      - position
      - velocity

    chainable_command_interfaces:
      - position
      - velocity


test_admittance_controller:
  # contains minimal needed parameters for kuka_kr6
  ros__parameters:
    joints:
      - joint1
      - joint2
      - joint3
      - joint4
      - joint5
      - joint6

    command_interfaces:
      - position

    state_interfaces:
      - position

    chainable_command_interfaces:
      - position
      - velocity
```

(continues on next page)

(continued from previous page)

```
    kinematics:
      plugin_name: kinematics_interface_kdl/KinematicsInterfaceKDL
      plugin_package: kinematics_interface
      base: base_link # Assumed to be stationary
      tip: tool0
      group_name: arm
      alpha: 0.0005

    ft_sensor:
      name: ft_sensor_name
      frame:
        id: link_6  # tool0 Wrench measurements are in this frame
        external: false # force torque frame exists within URDF kinematic chain
      filter_coefficient: 0.005

    control:
      frame:
        id: tool0 # Admittance calcs (displacement etc) are done in this frame.␣
↪Usually the tool or end-effector
        external: false # control frame exists within URDF kinematic chain

    fixed_world_frame:   # Gravity points down (neg. Z) in this frame (Usually: world␣
↪or base_link)
      frame:
        id: base_link # Admittance calcs (displacement etc) are done in this frame.␣
↪Usually the tool or end-effector
        external: false # control frame exists within URDF kinematic chain

    gravity_compensation:
      frame:
        id: tool0
        external: false

      CoG: # specifies the center of gravity of the end effector
        pos:
          - 0.1 # x
          - 0.0 # y
          - 0.0 # z
        force: 23.0 # mass * 9.81

    admittance:
      selected_axes:
        - true # x
        - true # y
        - true # z
        - true # rx
        - true # ry
        - true # rz

      # Having ".0" at the end is MUST, otherwise there is a loading error
      # F = M*a + D*v + S*(x - x_d)
      mass:
        - 5.5
        - 6.6
        - 7.7
        - 8.8
```

(continues on next page)

(continued from previous page)

```
      - 9.9
      - 10.10

    damping_ratio: # damping can be used instead: zeta = D / (2 * sqrt( M * S ))
      - 2.828427 # x
      - 2.828427 # y
      - 2.828427 # z
      - 2.828427 # rx
      - 2.828427 # ry
      - 2.828427 # rz

    stiffness:
      - 214.1
      - 214.2
      - 214.3
      - 214.4
      - 214.5
      - 214.6
```

Topics

~/joint_references (input topic) [trajectory_msgs::msg::JointTrajectoryPoint]
Target joint commands when controller is not in chained mode.

~/wrench_reference (input topic) [geometry_msgs::msg::WrenchStamped]
Target wrench offset (WrenchStamped has to be in the frame of the FT-sensor).

~/state (output topic) [control_msgs::msg::AdmittanceControllerState]
Topic publishing internal states.

ros2_control interfaces

References

The controller has `position` and `velocity` reference interfaces exported in the format: `<controller_name>/<joint_name>/[position|velocity]`

States

The state interfaces are defined with `joints` and `state_interfaces` parameters as follows: `<joint>/<state_interface>`. Supported state interfaces are `position`, `velocity`, and `acceleration` as defined in the hardware_interface/hardware_interface_type_values.hpp. If some interface is not provided, the last commanded interface will be used for calculation.

For handling TCP wrenches *Force Torque Sensor* semantic component (from package *controller_interface*) is used. The interfaces have prefix `ft_sensor.name`, building the interfaces: `<sensor_name>/[force.x|force.y|force.z|torque.x|torque.y|torque.z]`.

Commands

The command interfaces are defined with `joints` and `command_interfaces` parameters as follows: `<joint>/<command_interface>`. Supported state interfaces are `position`, `velocity`, and `acceleration` as defined in the hardware_interface/hardware_interface_type_values.hpp.

3.3.2 effort_controllers

This is a collection of controllers that work using the "effort" joint command interface but may accept different joint-level commands at the controller level, e.g. controlling the effort on a certain joint to achieve a set position.

The package contains the following controllers:

effort_controllers/JointGroupEffortController

> **Warning:** `effort_controllers/JointGroupEffortController` is deprecated. Use *forward_command_controller* instead by adding the `interface_name` parameter and set it to `effort`.

This is specialization of the *forward_command_controller* that works using the "effort" joint interface.

ROS 2 interface of the controller

Topics

~/commands (input topic) [std_msgs::msg::Float64MultiArray]
Joints' effort commands

Parameters

This controller overrides the interface parameter from *forward_command_controller*, and the `joints` parameter is the only one that is required.

An example parameter file is given here

```
controller_manager:
  ros__parameters:
    update_rate: 100  # Hz

    effort_controller:
      type: effort_controllers/JointGroupEffortController

effort_controller:
  ros__parameters:
    joints:
      - slider_to_cart
```

3.3.3 forward_command_controller

A selection of controllers that forward commands of different types.

forward_command_controller and multi_interface_forward_command_controller

Both controllers forward `std_msgs::msg::Float64MultiArray` to a set of interfaces, which can be parameterized as follows: While `forward_command_controller/ForwardCommandController` only claims a single interface type per joint (`joint[i] + "/" + interface_name`), the `forward_command_controller/MultiInterfaceForwardCommandController` claims the combination of all interfaces specified in the `interface_names` parameter (`joint[i] + "/" + interface_names[j]`).

Hardware interface type

This controller can be used for every type of command interface, not only limited to joints.

ROS 2 interface of the controller

Topics

~/commands (input topic) [std_msgs::msg::Float64MultiArray]
Target joint commands

Parameters

This controller uses the generate_parameter_library to handle its parameters.

forward_command_controller

joints (string_array)
Name of the joints to control

Read only: True

Constraints:

- parameter is not empty

interface_name (string)
Name of the interface of the joint

Read only: True

Constraints:

- parameter is not empty

multi_interface_forward_command_controller

joint (string)
Name of the joint to control

Read only: True

Constraints:

- parameter is not empty

interface_names (string_array)
Names of the interfaces per joint to claim

Read only: True

Constraints:

- parameter is not empty

3.3.4 joint_trajectory_controller

Controller for executing joint-space trajectories on a group of joints. The controller interpolates in time between the points so that their distance can be arbitrary. Even trajectories with only one point are accepted. Trajectories are specified as a set of waypoints to be reached at specific time instants, which the controller attempts to execute as well as the mechanism allows. Waypoints consist of positions, and optionally velocities and accelerations.

Parts of this documentation were originally published in the ROS 1 wiki under the CC BY 3.0 license. *Citations are given in the respective section, but were adapted for the ROS 2 implementation.*[1]

Hardware interface types

Currently, joints with hardware interface types `position`, `velocity`, `acceleration`, and `effort` (defined here) are supported in the following combinations as command interfaces:

- `position`
- `position`, `velocity`
- `position`, `velocity`, `acceleration`
- `velocity`
- `effort`
- `position`, `effort`

This means that the joints can have one or more command interfaces, where the following control laws are applied at the same time:

- For command interfaces `position`, the desired positions are simply forwarded to the joints,
- For command interfaces `acceleration`, desired accelerations are simply forwarded to the joints.
- For `velocity` command interfaces, the position+velocity trajectory following error is mapped to `velocity` commands through a PID loop if it is configured (*Details about parameters*).
- For `effort` command interface (without `position` command interface), the position+velocity trajectory following error is mapped to `effort` commands through a PID loop if it is configured (*Details about parameters*). In addition, it adds trajectory's effort as feedforward effort to the PID output.
- For `position, effort` command interface, PID loop is not used. If the trajectory contains effort, its value will be passed directly to the `effort` interface while the desired positions will be forwarded to the `position` interface. This could be useful for manipulation tasks where you need to add that extra force to maintain contact.

This leads to the following allowed combinations of command and state interfaces:

- With command interface `position`, there are no restrictions for state interfaces.
- With command interface `velocity`:
  - if command interface `velocity` is the only one, state interfaces must include `position, velocity`.

[1] Adolfo Rodriguez: joint_trajectory_controller

- With command interface `effort` or `position, effort`, state interfaces must include `position, velocity`.
- With command interface `acceleration`, state interfaces must include `position, velocity`.

Further restrictions of state interfaces exist:

- `velocity` state interface cannot be used if `position` interface is missing.
- `acceleration` state interface cannot be used if `position` and `velocity` interfaces are not present."

Example controller configurations can be found *below*.

Other features

- Realtime-safe implementation.
- Proper handling of wrapping (continuous) joints.
- Robust to system clock changes: Discontinuous system clock changes do not cause discontinuities in the execution of already queued trajectory segments.
- Optional smooth deceleration on cancel: Instead of abruptly holding position, the controller can decelerate joints to a stop using configurable per-joint deceleration limits. See *Decelerate on cancel*.

Using Joint Trajectory Controller(s)

The controller expects at least position feedback from the hardware. Joint velocities and accelerations are optional. Currently the controller does not internally integrate velocity from acceleration and position from velocity. Therefore if the hardware provides only acceleration or velocity states they have to be integrated in the hardware-interface implementation of velocity and position to use these controllers.

The generic version of Joint Trajectory controller is implemented in this package. A yaml file for using it could be:

```
controller_manager:
  ros__parameters:
    joint_trajectory_controller:
    type: "joint_trajectory_controller/JointTrajectoryController"

joint_trajectory_controller:
  ros__parameters:
    joints:
      - joint1
      - joint2
      - joint3
      - joint4
      - joint5
      - joint6

    command_interfaces:
      - position

    state_interfaces:
      - position
      - velocity

    action_monitor_rate: 20.0
```

(continues on next page)

(continued from previous page)

```
    allow_partial_joints_goal: false
    interpolate_from_desired_state: true
    constraints:
      stopped_velocity_tolerance: 0.01
      goal_time: 0.0
      joint1:
        trajectory: 0.05
        goal: 0.03
```

Preemption policy[Page 106, 1]

Only one action goal can be active at any moment, or none if the topic interface is used. Path and goal tolerances are checked only for the trajectory segments of the active goal.

When an active action goal is preempted by another command coming from the action interface, the goal is canceled and the client is notified. The trajectory is replaced in a defined way, see *trajectory replacement*.

Sending an empty trajectory message from the topic interface (not the action interface) will override the current action goal and not abort the action.

Description of controller's interfaces

References

(the controller is not yet implemented as chainable controller)

States

The state interfaces are defined with `joints` and `state_interfaces` parameters as follows: `<joint>/<state_interface>`.

Legal combinations of state interfaces are given in section *Hardware Interface Types*.

Commands

There are two mechanisms for sending trajectories to the controller:

- via action, see *actions*
- via topic, see *subscriber*

Both use the `trajectory_msgs/msg/JointTrajectory` message to specify trajectories, and require specifying values for all the controller joints (as opposed to only a subset) if `allow_partial_joints_goal` is not set to `True`. For further information on the message format, see *trajectory representation*.

Actions[Page 106, 1]

<controller_name>/follow_joint_trajectory [control_msgs::action::FollowJointTrajectory]
Action server for commanding the controller

The primary way to send trajectories is through the action interface, and should be favored when execution monitoring is desired.

Action goals allow to specify not only the trajectory to execute, but also (optionally) path and goal tolerances. For details, see the JointTolerance message:

```
The tolerances specify the amount the position, velocity, and
accelerations can vary from the setpoints.  For example, in the case
of trajectory control, when the actual position varies beyond
(desired position + position tolerance), the trajectory goal may
abort.

There are two special values for tolerances:
  * 0 - The tolerance is unspecified and will remain at whatever the default is
  * -1 - The tolerance is "erased".  If there was a default, the joint will be
         allowed to move without restriction.
```

When no tolerances are specified, the defaults given in the parameter interface are used (see *Details about parameters*). If tolerances are violated during trajectory execution, the action goal is aborted, the client is notified, and the current position is held.

The action server returns success to the client and continues with the last commanded point after the target is reached within the specified tolerances.

Subscriber[Page 106, 1]

<controller_name>/joint_trajectory [trajectory_msgs::msg::JointTrajectory]
Topic for commanding the controller

The topic interface is a fire-and-forget alternative. Use this interface if you don't care about execution monitoring. The goal tolerance specification is not used in this case, as there is no mechanism to notify the sender about tolerance violations. If state tolerances are violated, the trajectory is aborted and the current position is held. Note that although some degree of monitoring is available through the `~/query_state` service and `~/controller_state` topic it is much more cumbersome to realize than with the action interface.

Publishers

<controller_name>/controller_state [control_msgs::msg::JointTrajectoryControllerState]
Topic publishing internal states with the update-rate of the controller manager

Services

<controller_name>/query_state [control_msgs::srv::QueryTrajectoryState]
Query controller state at any future time

Further information

Trajectory Representation

Trajectories are represented internally with `trajectory_msgs/msg/JointTrajectory` data structure.

Currently, two interpolation methods are implemented: `none` and `spline`. By default, a spline interpolator is provided, but it's possible to support other representations.

Warning: The user has to ensure that the correct inputs are provided for the trajectory, which are needed by the controller's setup of command interfaces and PID configuration. There is no sanity check and missing fields in the sampled trajectory might cause segmentation faults.

Interpolation Method `none`

It returns the initial point until the time for the first trajectory data point is reached. Then, it simply takes the next given datapoint.

Warning: It does not deduce (integrate) trajectory from derivatives, nor does it calculate derivatives. I.e., one has to provide position and its derivatives as needed.

Interpolation Method `spline`

The spline interpolator uses the following interpolation strategies depending on the waypoint specification:

- Linear:
  - Used, if only position is specified.
  - Returns position and velocity
  - Guarantees continuity at the position level.
  - Discouraged because it yields trajectories with discontinuous velocities at the waypoints.
- Cubic:
  - Used, if position and velocity are specified.
  - Returns position, velocity, and acceleration.
  - Guarantees continuity at the velocity level.
- Quintic:
  - Used, if position, velocity and acceleration are specified
  - Returns position, velocity, and acceleration.

  - Guarantees continuity at the acceleration level.

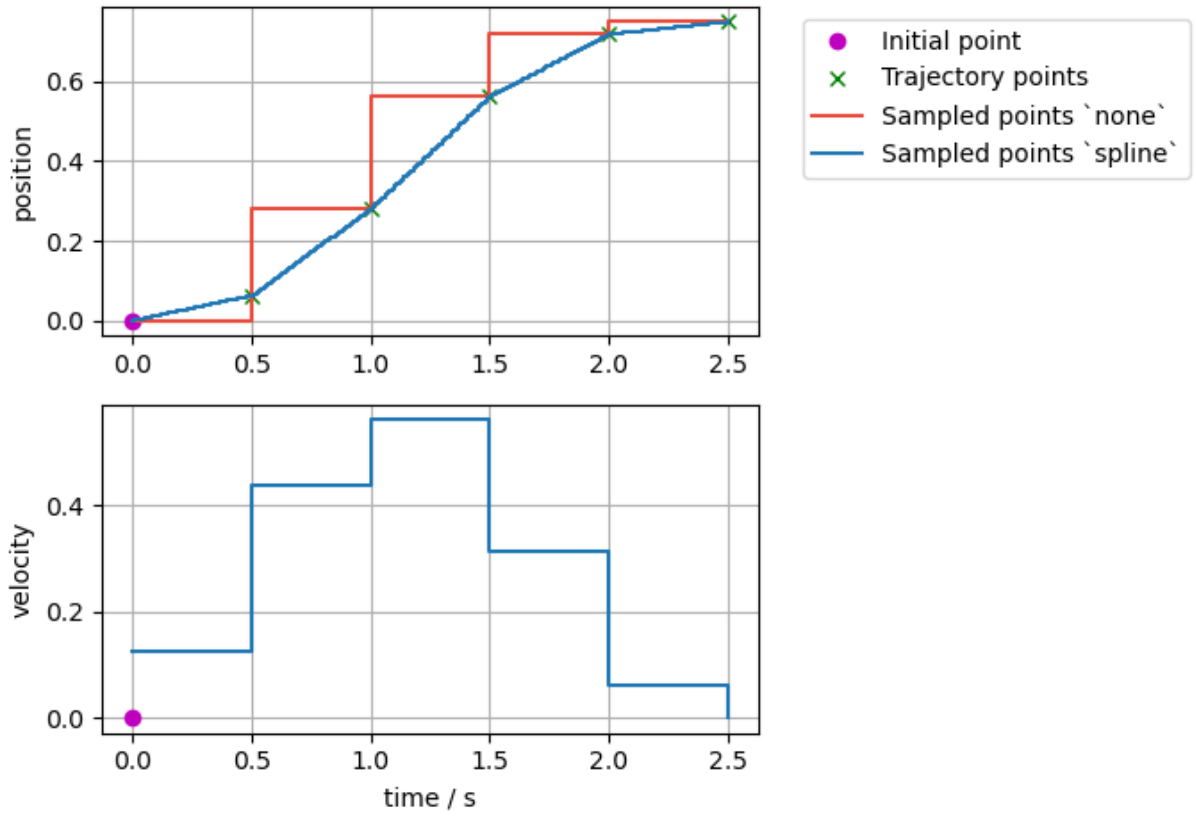
Trajectories with velocity fields only, velocity and acceleration only, or acceleration fields only can be processed and are accepted, if `allow_integration_in_goal_trajectories` is true. Position (and velocity) is then integrated from velocity (or acceleration, respectively) by Heun's method.

Effort trajectories are allowed for controllers that claim the `effort` command interface and they are treated as feed-forward effort that is added to the position feedback. Effort is handled separately from position, velocity and acceleration. We use linear interpolation for effort when the `spline` interpolation method is selected.

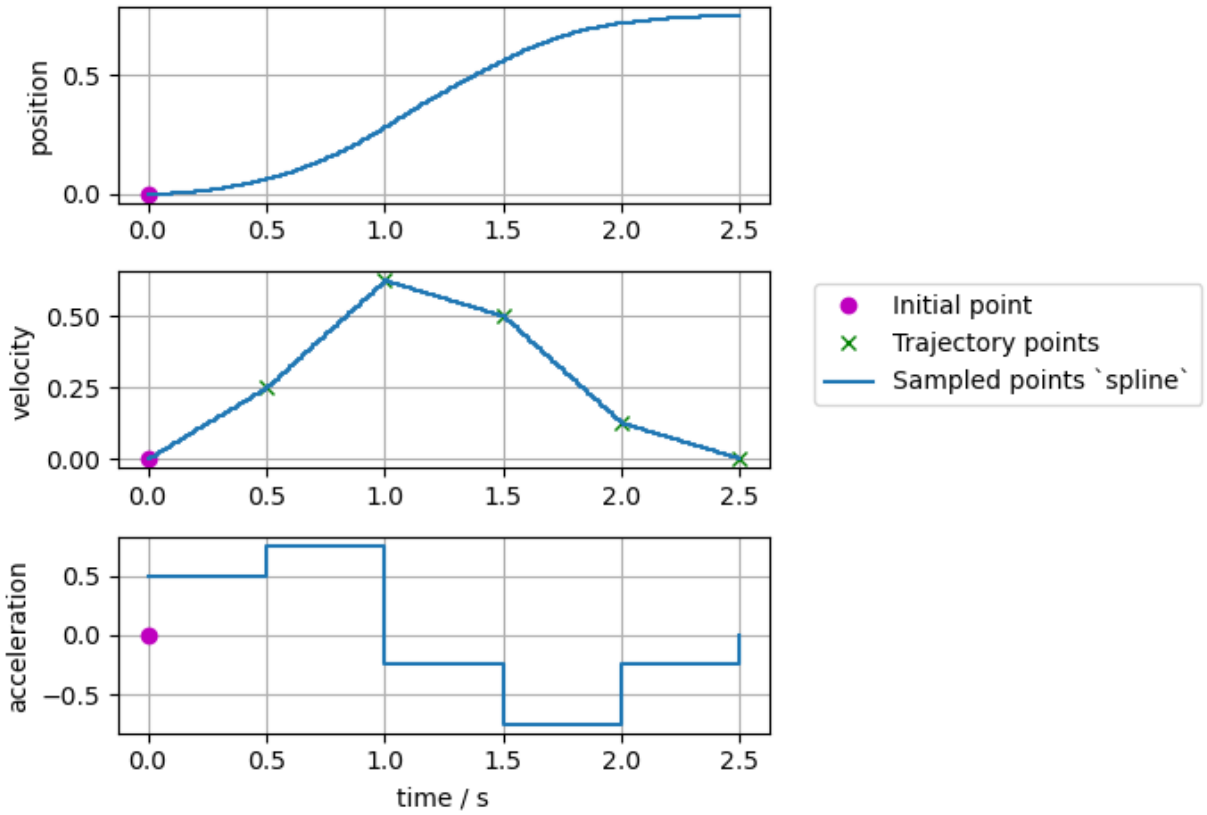
Visualized Examples

To visualize the difference of the different interpolation methods and their inputs, different trajectories defined at a 0.5s grid and are sampled at a rate of 10ms.

- Sampled trajectory with linear spline if position is given only:

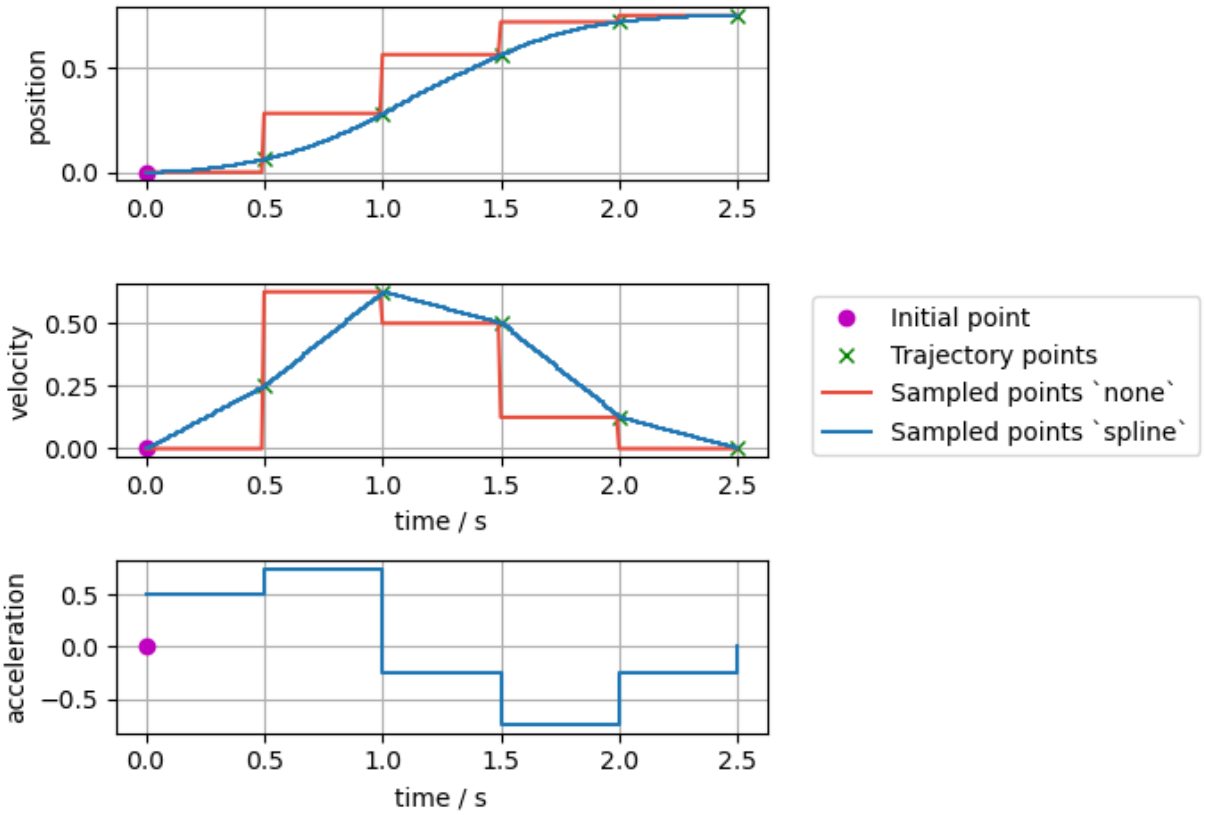


- Sampled trajectory with cubic splines if velocity is given only (no deduction for interpolation method `none`):

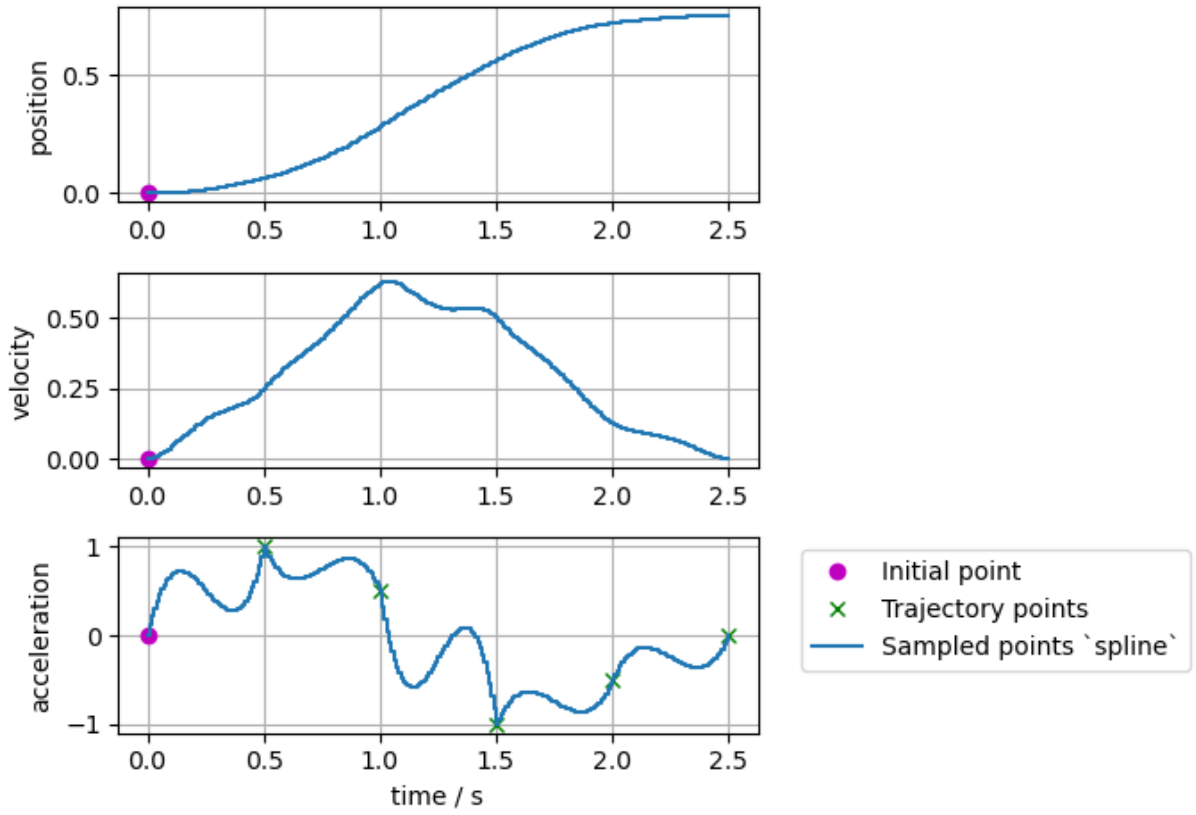


- Sampled trajectory if position and velocity is given:

Note: If the same integration method was used (`Trajectory` class uses Heun's method), then the `spline` method this gives identical results as above where velocity only was given as input.

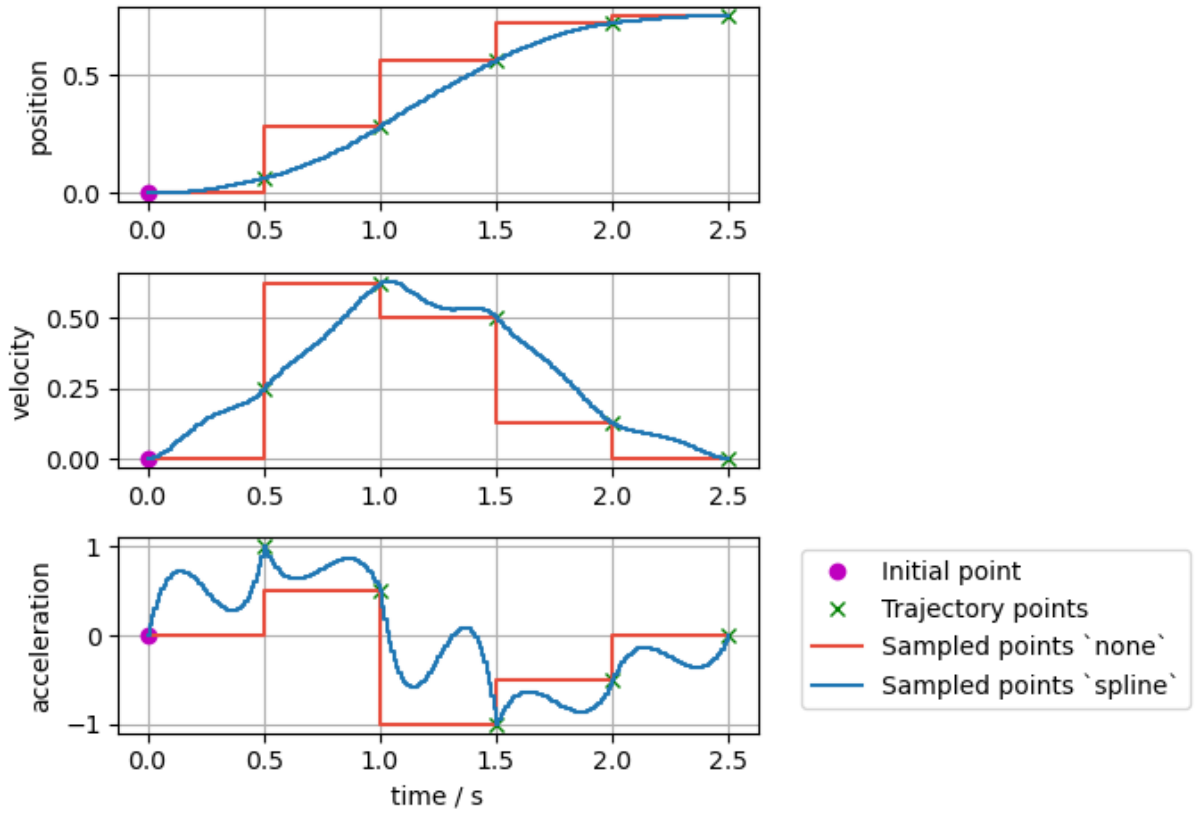


- Sampled trajectory with quintic splines if acceleration is given only (no deduction for interpolation method `none`):

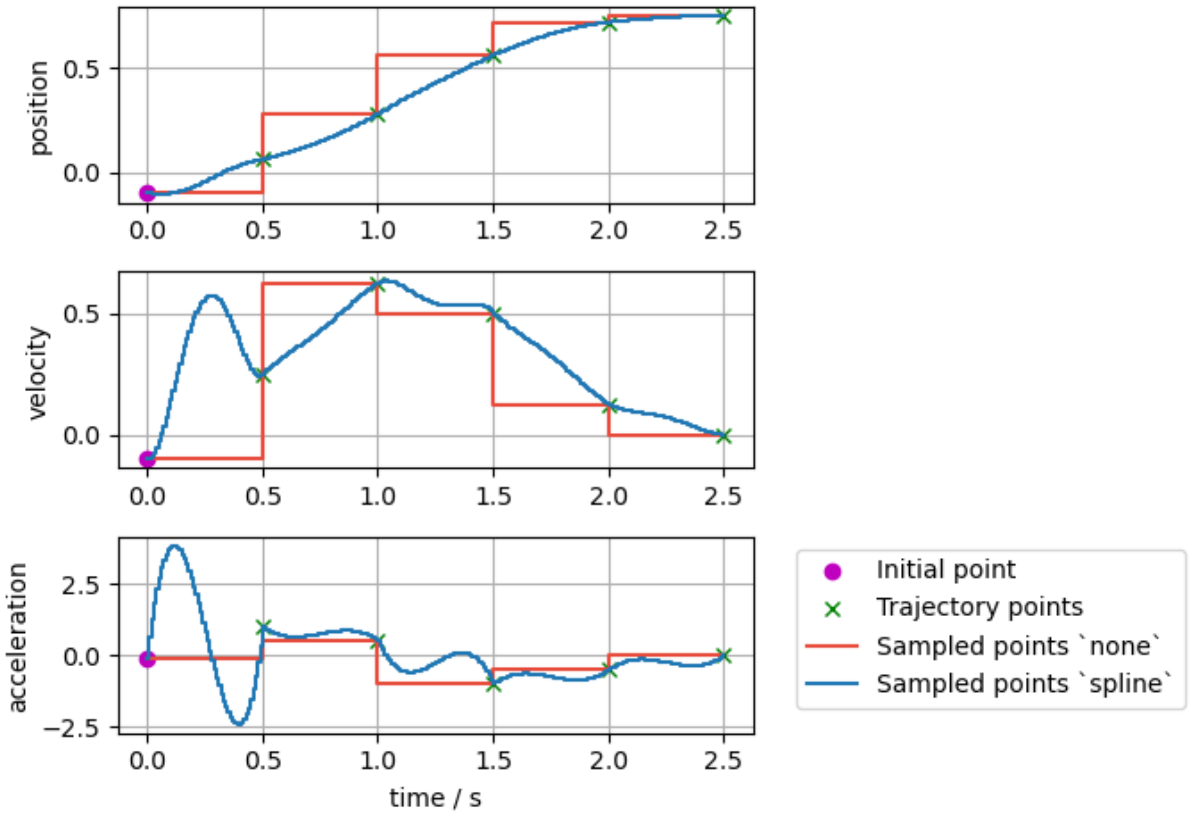


- Sampled trajectory if position, velocity, and acceleration points are given:

Note: If the same integration method was used (`Trajectory` class uses Heun's method), then the `spline` method this gives identical results as above where acceleration only was given as input.

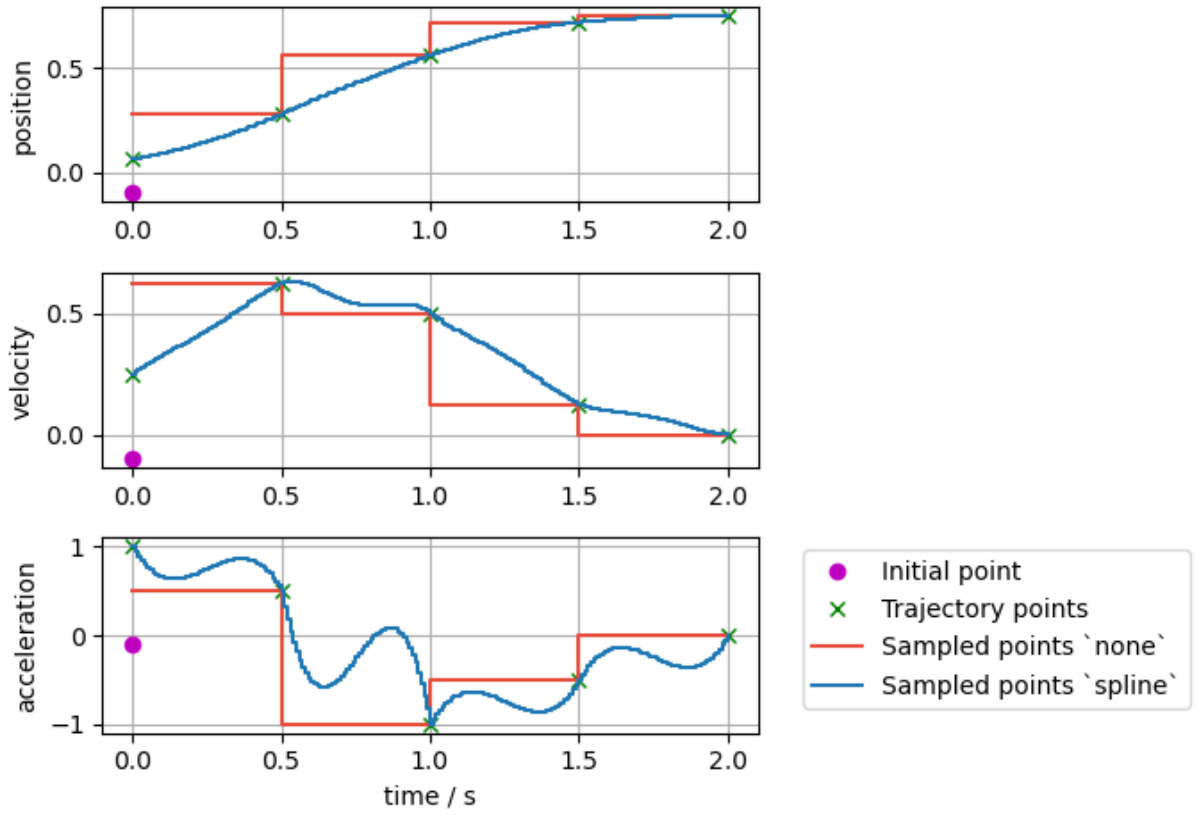


- Sampled trajectory if the same position, velocity, and acceleration points as above are given, but with a nonzero initial point:



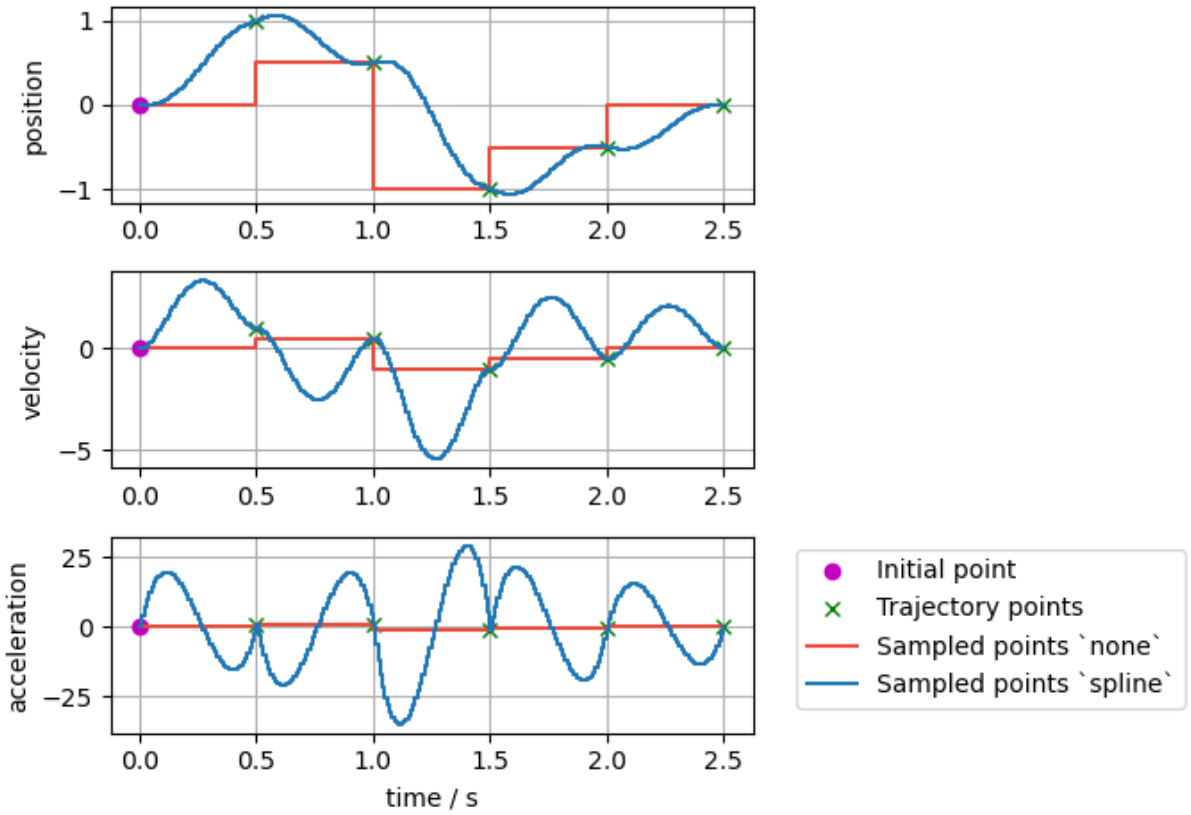
- Sampled trajectory if the same position, velocity, and acceleration points as above are given but with the first point starting at `t=0`:

Note: If the first point is starting at `t=0`, there is no interpolation from the initial point to the trajectory.



- Sampled trajectory with splines if inconsistent position, velocity, and acceleration points are given:

Note: Interpolation method `none` only gives the next input points, while the `spline` interpolation method shows high overshoot to match the given trajectory points.



Trajectory Replacement

Parts of this documentation were originally published in the ROS 1 wiki under the CC BY 3.0 license.[1]

Joint trajectory messages allow to specify the time at which a new trajectory should start executing by means of the header timestamp, where zero time (the default) means "start now".

Warning: As of now, this functionality is not ported to ROS 2, see this issue for more information. The current implementation just forgets the old trajectory.

The arrival of a new trajectory command does not necessarily mean that the controller will completely discard the currently running trajectory and substitute it with the new one. Rather, the controller will take the useful parts of both and combine them appropriately, yielding a smarter trajectory replacement strategy.

The steps followed by the controller for trajectory replacement are as follows:

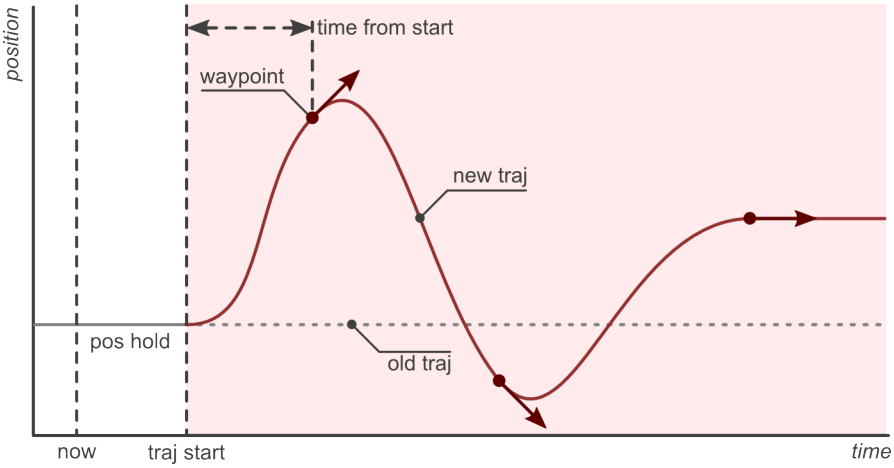
- Get useful parts of the new trajectory: Preserve all waypoints whose time to be reached is in the future, and discard those with times in the past. If there are no useful parts (ie. all waypoints are in the past) the new trajectory is rejected and the current one continues execution without changes.
- Get useful parts of the current trajectory: Preserve the current trajectory up to the start time of the new trajectory, discard the later parts.
- Combine the useful parts of the current and new trajectories.

[1] Adolfo Rodriguez: Understanding trajectory replacement

The following examples describe this behavior in detail.

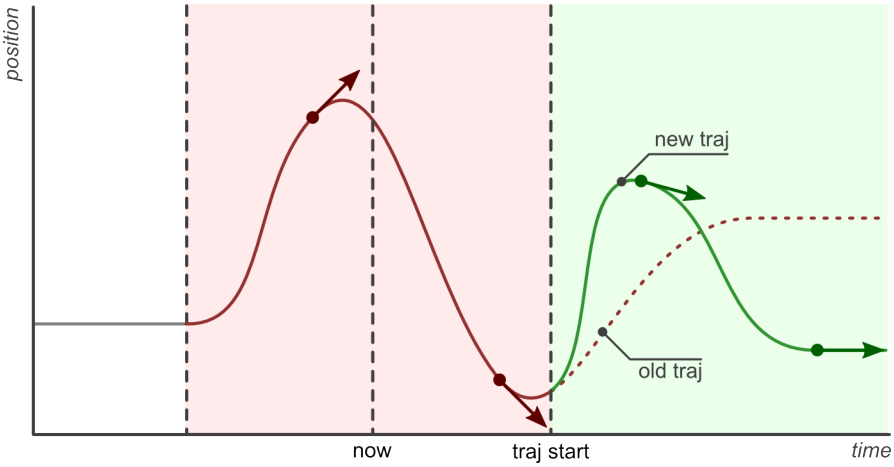
The first example shows a joint which is in hold position mode (flat grey line labeled *pos hold* in the figure below). A new trajectory (shown in red) arrives at the current time (now), which contains three waypoints and a start time in the future (*traj start*). The time at which waypoints should be reached (`time_from_start` member of `trajectory_msgs/JointTrajectoryPoint`) is relative to the trajectory start time.

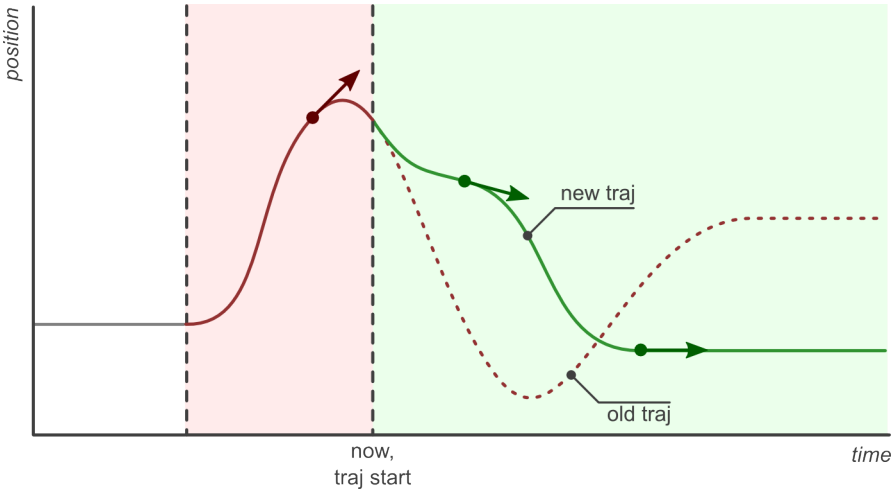
The controller splices the current hold trajectory at time *traj start* and appends the three waypoints. Notice that between now and *traj start* the previous position hold is still maintained, as the new trajectory is not supposed to start yet. After the last waypoint is reached, its position is held until new commands arrive.



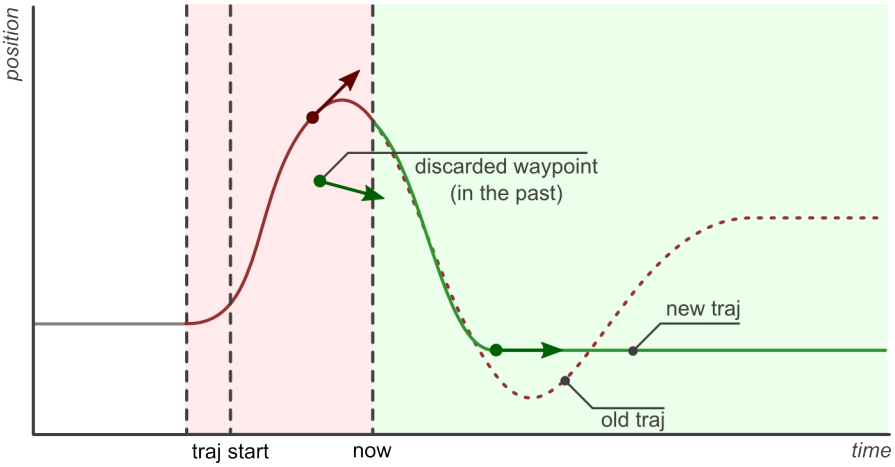
The controller guarantees that the transition between the current and new trajectories will be smooth. Longer times to reach the first waypoint mean slower transitions.

The next examples discuss the effect of sending the same trajectory to the controller with different start times. The scenario is that of a controller executing the trajectory from the previous example (shown in red), and receiving a new command (shown in green) with a trajectory start time set to either zero (start now), a future time, or a time in the past.





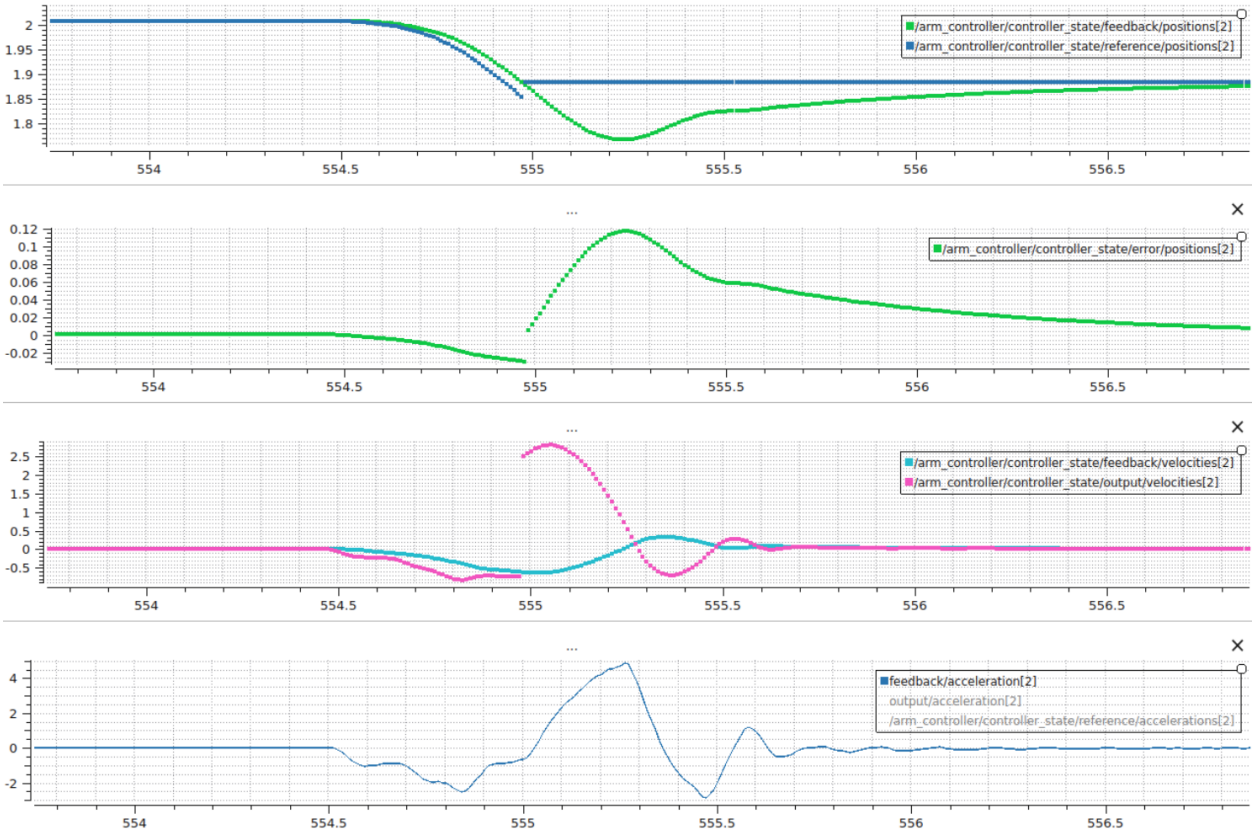
Of special interest is the last example, where the new trajectory start time and first waypoint are in the past (before now). In this case, the first waypoint is discarded and only the second one is realized.



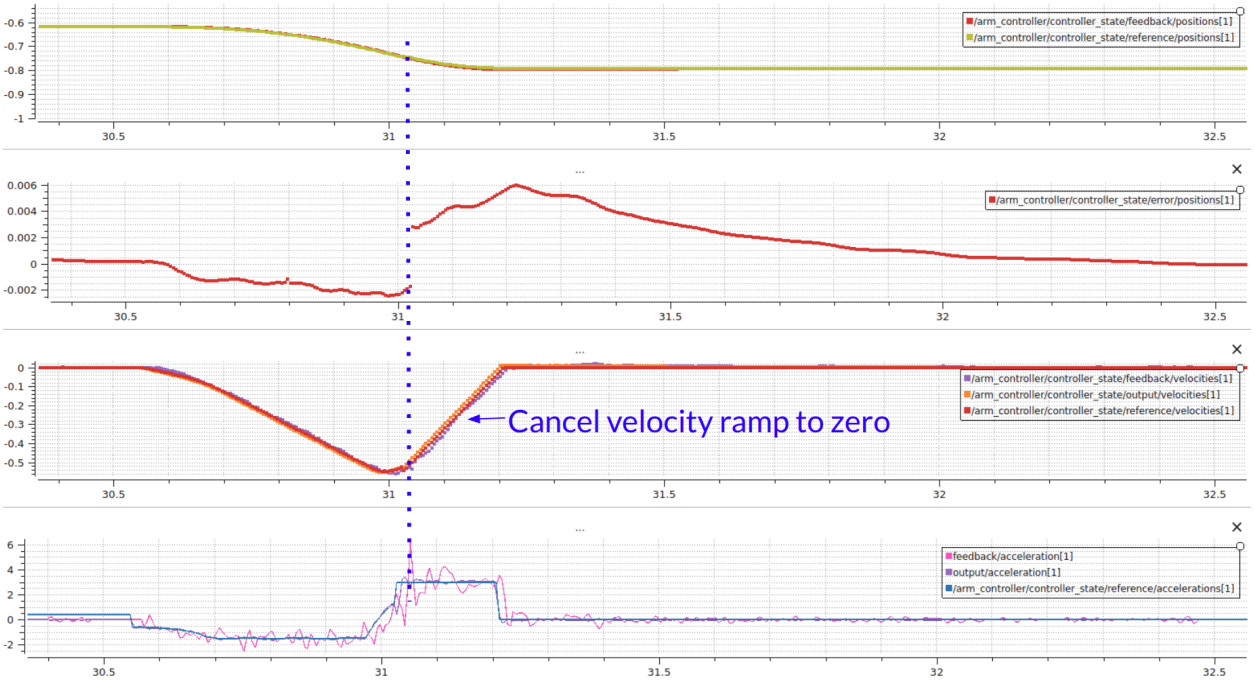
Decelerate on cancel

By default, when a trajectory is canceled or preempted, the controller immediately holds the current position. This can cause problems for hardware that is moving at high velocity, as the abrupt stop may trigger faults or cause excessive wear.

When the `decelerate_on_cancel` feature is enabled, the controller instead generates a smooth stop trajectory that decelerates each joint to zero velocity using a constant deceleration profile before holding position.



Default behavior: the controller instantly sets the command position to the current position when a trajectory is canceled, causing an abrupt stop.



With decelerate on cancel enabled: the controller generates a ramped stop trajectory that smoothly decelerates each joint before holding position.

How it works

When a trajectory is canceled or preempted and `decelerate_on_cancel` is enabled, the controller:

1. Reads the current velocity of each joint from the velocity state interface.
2. Computes the stopping distance and time for each joint using its configured `max_deceleration_on_cancel` value:

$$t_{stop} = \frac{|v|}{a_{max}}$$

$$d_{stop} = \frac{v \cdot t_{stop}}{2}$$

   where v is the current velocity and a_{max} is the configured maximum deceleration.
3. Generates a trajectory with intermediate waypoints that linearly ramp velocity to zero over the computed stopping time.
4. Appends a final hold-position point at the computed stop position.

Each joint decelerates independently based on its own `max_deceleration_on_cancel` value, but the trajectory is synchronized so all joints finish at the same time (the slowest joint's stop time).

Requirements

- The hardware must provide a `velocity` state interface for all joints in the controller. If velocity state feedback is not available, the controller falls back to the default hold-position behavior.
- Each joint must have a valid (greater than zero) `max_deceleration_on_cancel` value. Joints with a value of `0.0` cause the controller to fall back to hold-position behavior.

Configuration

Enable the feature by setting `constraints.decelerate_on_cancel` to `true` and providing a `max_deceleration_on_cancel` value (in rad/s^2 or m/s^2) for each joint under `constraints.<joint_name>`:

```yaml
controller_name:
  ros__parameters:
    joints:
      - joint_1
      - joint_2
      - joint_3

    constraints:
      decelerate_on_cancel: true
      stopped_velocity_tolerance: 0.01
      goal_time: 0.0
      joint_1:
        max_deceleration_on_cancel: 10.0
      joint_2:
        max_deceleration_on_cancel: 3.0
      joint_3:
        max_deceleration_on_cancel: 6.0
```

Note: Both `decelerate_on_cancel` and `max_deceleration_on_cancel` are read-only parameters. They can only be set at controller configuration time and cannot be changed at runtime.

Note: Choose `max_deceleration_on_cancel` values that are within the physical limits of your hardware. Values that are too high may still cause faults, while values that are too low will result in longer stopping distances.

Speed scaling

The `joint_trajectory_controller` (JTC) supports dynamically scaling its trajectory execution speed. That means, when specifying a scaling factor f of less than 1, execution will proceed only $f \cdot \Delta_t$ per control step where Δ_t is the controller's cycle time.

Methods of speed scaling

Generally, the speed scaling feature has two separate scaling approaches in mind: On-Robot scaling and On-Controller scaling. They are both conceptually different and to correctly configure speed scaling it is important to understand the differences.

On-Robot speed scaling

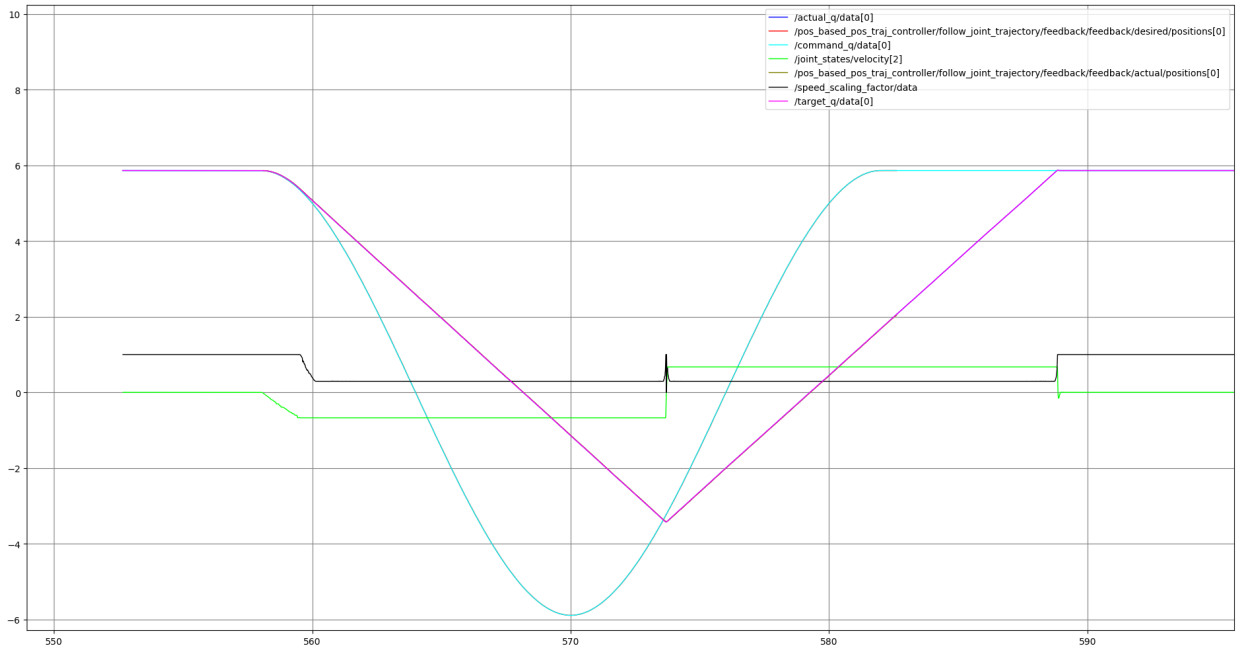
This scaling method is intended for robots that provide a scaling feature directly on the robot's teach pendant and / or through a safety feature. One example of such robots are the Universal Robots manipulators.

The hardware interface needs to report the speed scaling through a state interface so it can be read by the controller. Optionally, a command interface for setting the speed scaling value on the hardware can be provided (if applicable) in order to set speed scaling through a ROS topic.

For the scope of this documentation a user-defined scaling and a safety-limited scaling will be treated the same resulting in a "hardware scaling factor".

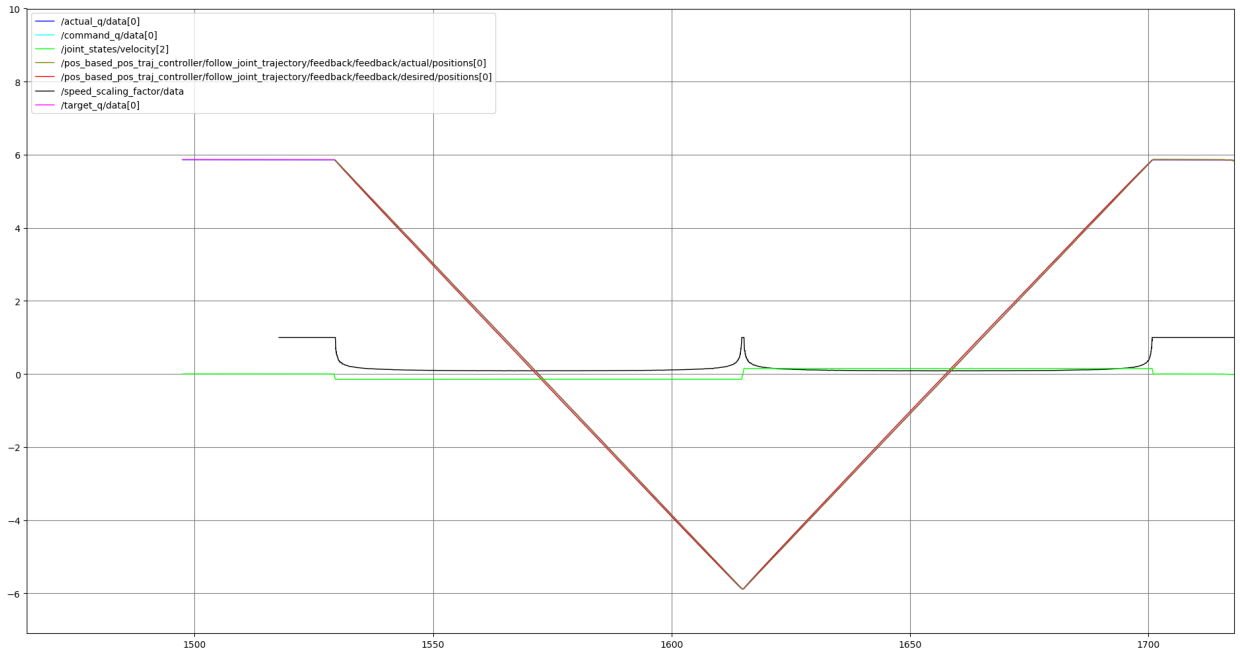
In this setup, the hardware will treat the command sent from the ROS controller (e.g. Reach joint configuration θ within Δ_t seconds.). This effectively means that the robot will only make half of the way towards the target configuration when a scaling factor of 0.5 is given (neglecting acceleration and deceleration influences during this time period).

The following plot shows trajectory execution (for one joint) with a hardware-scaled execution and a controller that is **not** aware of speed scaling:



The graph shows a trajectory with one joint being moved to a target point and back to its starting point. As the joint's speed is limited to a very low setting on the teach pendant, speed scaling (black line) activates and limits the joint speed (green line). As a result, the target trajectory (light blue) doesn't get executed by the robot, but instead the pink trajectory is executed. The vertical distance between the light blue line and the pink line is the path error in each control cycle. We can see that the path deviation gets above 300 degrees at some point and the target point at -6 radians never gets reached.

With the scaled version of the trajectory controller the example motion shown in the previous diagram becomes:



The deviation between trajectory interpolation on the ROS side and actual robot execution stays minimal and the robot reaches the intermediate setpoint instead of returning "too early" as in the example above.

Scaling is done in such a way, that the time in the trajectory is virtually scaled. For example, if a controller runs with a

cycle time of 100 Hz, each control cycle is 10 ms long. A speed scaling of 0.5 means that in each time step the trajectory is moved forward by 5 ms instead. So, the beginning of the 3rd timestep is 15 ms instead of 30 ms in the trajectory.

Command sampling is performed as in the unscaled case, with the timestep's start plus the **full** cycle time of 10 ms. The robot will scale down the motion command by 50% resulting in only half of the distance being executed, which is why the next control cycle will be started at the current start plus half of the step time.

On-Controller speed scaling

Conceptually, with this scaling the robot hardware isn't aware of any scaling happening. The JTC generates commands to be sent to the robot that are already scaled down accordingly, so they can be directly executed by the robot.

Since the hardware isn't aware of speed scaling, the speed-scaling related command and state interfaces should not be specified and the scaling factor will be set through the `~/speed_scaling_input` topic directly:

```
$ ros2 topic pub --qos-durability transient_local --once \
  /joint_trajectory_controller/speed_scaling_input control_msgs/msg/
↪SpeedScalingFactor "{factor: 0.5}"
```

Note: The `~/speed_scaling_input` topic uses the QoS durability profile `transient_local`. This means you can restart the controller while still having a publisher on that topic active.

Note: The current implementation only works for position-based interfaces.

Effect on tolerances

When speed scaling is used while executing a trajectory, the tolerances configured for execution will be scaled, as well.

Since commands are generated from the scaled trajectory time, **path errors** will also be compared to the scaled trajectory.

The **goal time tolerance** also uses the virtual trajectory time. This means that a trajectory being executed with a constant scaling factor of 0.5 will take twice as long for execution than the `time_from_start` value of the last trajectory point specifies. As long as the robot doesn't take longer than that the goal time tolerance is considered to be met.

If an application relies on the actual execution time as set in the `time_from_start` fields, an external monitoring has to be wrapped around the trajectory execution action.

Details about parameters

This controller uses the generate_parameter_library to handle its parameters. The parameter definition file located in the src folder contains descriptions for all the parameters used by the controller.

List of parameters

joints (string_array)
Joint names of the system

Read only: True

Default: {}

Constraints:

- contains no duplicates
- length is greater than 0

command_joints (string_array)
Joint names to control. If left empty, `joints` will be used. This parameters can be used if:

- JTC is used in a controller chain where command and state interfaces don't have same names.
- If the number of command joints is smaller than the degrees-of-freedom. For example to track the state and error of passive joints. `command_joints` must then be a subset of `joints`.

Read only: True

Default: {}

Constraints:

- contains no duplicates

command_interfaces (string_array)
Command interfaces provided by the hardware interface for all joints

Read only: True

Default: {}

Constraints:

- contains no duplicates
- length is greater than 0
- every element is one of the list ['position', 'velocity', 'acceleration', 'effort']
- Custom validator: joint_trajectory_controller::command_interface_type_combinations

state_interfaces (string_array)
State interfaces provided by the hardware for all joints.

Read only: True

Default: {}

Constraints:

- contains no duplicates
- length is greater than 0
- every element is one of the list ['position', 'velocity', 'acceleration']
- Custom validator: joint_trajectory_controller::state_interface_type_combinations

speed_scaling.initial_scaling_factor (double)
The initial value of the scaling factor if no exchange with hardware takes place.

Read only: True

Default: 1.0

Constraints:

- greater than or equal to 0.0

speed_scaling.state_interface (string)
Fully qualified name of the speed scaling state interface name

Read only: True

Default: “”

speed_scaling.command_interface (string)
Command interface name used for setting the speed scaling factor on the hardware.

Read only: True

Default: “”

allow_partial_joints_goal (bool)
Allow joint goals defining trajectory for only some joints.

Default: false

interpolate_from_desired_state (bool)
Interpolate from the current desired state when receiving a new trajectory.

The controller ignores the states provided by hardware interface but using last commands as states for starting the trajectory interpolation.

This is useful if hardware states are not following commands, i.e., an offset between those (typical for hydraulic manipulators). Furthermore, it is necessary if you have a reference trajectory that you send over multiple messages (e.g. for MPC-style trajectory planning).

If this flag is set, the controller tries to read the values from the command interfaces on activation. If they have real numeric values, those will be used instead of state interfaces. Therefore it is important set command interfaces to NaN (i.e., `std::numeric_limits<double>::quiet_NaN()`) or state values when the hardware is started.

Read only: True

Default: false

allow_integration_in_goal_trajectories (bool)
Allow integration in goal trajectories to accept goals without position or velocity specified

Default: false

set_last_command_interface_value_as_state_on_activation (bool)
When set to true, the last command interface value is used as both the current state and the last commanded state upon activation. When set to false, the current state is used for both.

Default: true

action_monitor_rate (double)
Rate to monitor status changes when the controller is executing action (`control_msgs::action::FollowJointTrajectory`)

Read only: True

Default: 20.0

Constraints:

- greater than or equal to 0.1

interpolation_method (string)
The type of interpolation to use, if any

Read only: True

Default: "splines"

Constraints:

- one of the specified values: ['splines', 'none']

allow_nonzero_velocity_at_trajectory_end (bool)
If false, the last velocity point has to be zero or the goal will be rejected

Default: false

cmd_timeout (double)
Timeout after which the input command is considered stale. Timeout is counted from the end of the trajectory (the last point). `cmd_timeout` must be greater than `constraints.goal_time`, otherwise ignored. If zero, timeout is deactivated

Default: 0.0

constraints
Default values for tolerances if no explicit values are set in the `JointTrajectory` message. Also contains the `decelerate_on_cancel` option and per-joint `max_deceleration_on_cancel` values for smooth stopping behavior (see *Decelerate on cancel*).

constraints.stopped_velocity_tolerance (double)
Velocity tolerance for at the end of the trajectory that indicates that controlled system is stopped.

Default: 0.01

constraints.goal_time (double)
Time tolerance for achieving trajectory goal before or after commanded time. If set to zero, the controller will wait a potentially infinite amount of time.

Default: 0.0

Constraints:

- greater than or equal to 0.0

constraints.decelerate_on_cancel (bool)
If true and each joints max_deceleration_on_cancel is greater than 0, decelerate to a stop when the request is canceled or the goal constraints are violated. Requires velocity state interface from all joints in the controller.

Read only: True

Default: false

constraints.<joints>.trajectory (double)
Per-joint trajectory offset tolerance during movement.

Default: 0.0

constraints.<joints>.goal (double)
Per-joint trajectory offset tolerance at the goal position.

Default: 0.0

constraints.<joints>.max_deceleration_on_cancel (double)
Per-joint max acceleration used to calculate the stopping position when a request is canceled or preempted.

Read only: True

Default: 0.0

Constraints:

- greater than or equal to 0.0

gains
The parameters are used to configure PID loops for the `velocity` or `effort`-only command interfaces. This structure contains the controller gains for every joint with following the control laws

- If `velocity` is the only command interface:

$$u = k_{ff} v_d + k_p e + k_i \sum e dt + k_d (v_d - v)$$

with the desired velocity v_d, the measured velocity v, the position error e (definition see below), the controller period dt, and the `velocity` manipulated variable (control variable) u, respectively.

- If `effort` is the only command interface:

$$u = k_{ff} v_d + \delta_d + k_p e + k_i \sum e dt + k_d (v_d - v)$$

with the desired velocity v_d, the desired effort δ_d if provided in the trajectory (or 0 otherwise), the measured velocity v, the position error e (definition see below), the controller period dt, and the `effort` manipulated variable (control variable) u, respectively.

If the joint is of continuous type, the position error $e = normalize(s_d - s)$ is normalized between $-\pi, \pi$, i.e., the shortest rotation to the target position is the desired motion. Otherwise $e = s_d - s$ is used, with the desired position s_d and the measured position s from the state interface.

If you want to turn off the PIDs (open loop control), set the feedback gains to zero and an appropriate value for feed-forward gain k_{ff}.

gains.<joints>.p (double)
Proportional gain k_p for PID

Default: 0.0

gains.<joints>.i (double)
Integral gain k_i for PID

Default: 0.0

gains.<joints>.d (double)
Derivative gain k_d for PID

Default: 0.0

gains.<joints>.ff_velocity_scale (double)
Feed-forward scaling k_{ff} of velocity

Default: 0.0

gains.<joints>.u_clamp_max (double)
Upper output clamp.

Default: std::numeric_limits<double>::infinity()

gains.<joints>.u_clamp_min (double)
Lower output clamp.

Default: -std::numeric_limits<double>::infinity()

gains.<joints>.i_clamp_max (double)
Upper integral clamp.

Default: std::numeric_limits<double>::infinity()

gains.<joints>.i_clamp_min (double)
Lower integral clamp.

Default: -std::numeric_limits<double>::infinity()

gains.<joints>.antiwindup_strategy (string)
Specifies the anti-windup strategy. Options: 'back_calculation', 'conditional_integration' or 'none'. Note that the 'back_calculation' strategy use the tracking_time_constant parameter to tune the anti-windup behavior.

Default: "none"

Constraints:

- one of the specified values: ['back_calculation', 'conditional_integration', 'none']

gains.<joints>.tracking_time_constant (double)
Specifies the tracking time constant for the 'back_calculation' strategy. If set to 0.0 when this strategy is selected, a recommended default value will be applied.

Default: 0.0

gains.<joints>.error_deadband (double)
Is used to stop integration when the error is within the given range.

Default: 0.0

An example parameter file

```yaml
joint_trajectory_controller:
  ros__parameters:
    action_monitor_rate: 20.0
    allow_integration_in_goal_trajectories: false
    allow_nonzero_velocity_at_trajectory_end: false
    allow_partial_joints_goal: false
    cmd_timeout: 0.0
    command_interfaces: '{}'
    command_joints: '{}'
    constraints:
      <joints>:
        goal: 0.0
        max_deceleration_on_cancel: 0.0
        trajectory: 0.0
      decelerate_on_cancel: false
      goal_time: 0.0
      stopped_velocity_tolerance: 0.01
    gains:
      <joints>:
        antiwindup_strategy: none
        d: 0.0
        error_deadband: 0.0
        ff_velocity_scale: 0.0
        i: 0.0
        i_clamp_max: std::numeric_limits<double>::infinity()
        i_clamp_min: -std::numeric_limits<double>::infinity()
        p: 0.0
```

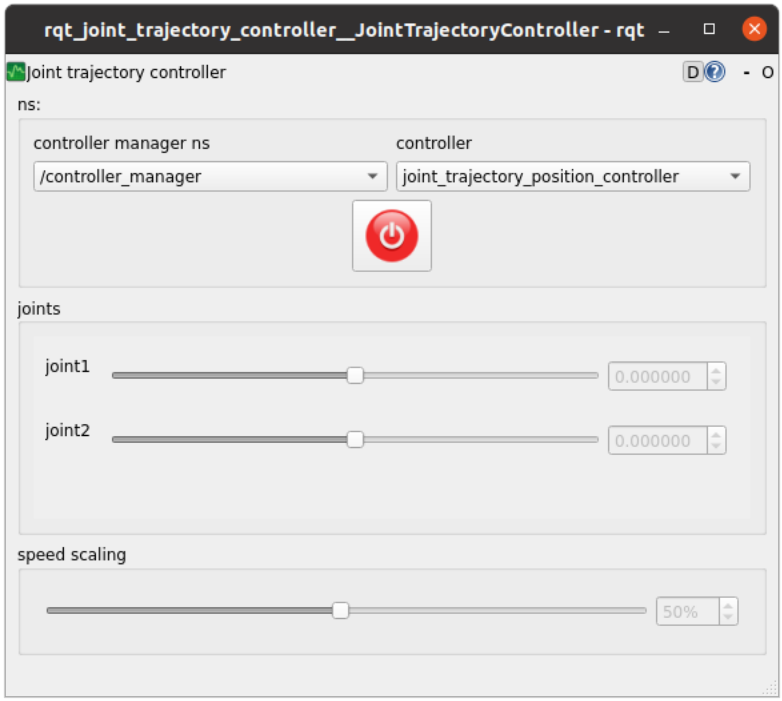
(continues on next page)

(continued from previous page)

```
      tracking_time_constant: 0.0
      u_clamp_max: std::numeric_limits<double>::infinity()
      u_clamp_min: -std::numeric_limits<double>::infinity()
  interpolate_from_desired_state: false
  interpolation_method: splines
  joints: '{}'
  set_last_command_interface_value_as_state_on_activation: true
  speed_scaling:
    command_interface: ''
    initial_scaling_factor: 1.0
    state_interface: ''
  state_interfaces: '{}'
```

rqt_joint_trajectory_controller

The rqt_joint_trajectory_controller provides an intuitive graphical way to test different joint positions and trajectories without having to manually construct complex trajectory messages or use command line interfaces.



The interface allows you to:

- Select the controller manager namespace and controller from dropdown menus.
- Adjust target positions for joints (joint1 and joint2) using interactive sliders.
- Fine-tune joint positions with precise numerical inputs.
- Control the motion speed using the speed scaling slider.
- Activate the trajectory execution with the central power button.
- Visualize current joint configurations in real-time.

Footnote

3.3.5 Parallel Gripper Action Controller

Controller for executing a `ParallelGripperCommand` action for simple parallel grippers. This controller supports grippers that offer position only control as well as grippers that allow configuring the velocity and effort. By default, the controller will only claim the `{joint}/position` interface for control. The velocity and effort interfaces can be optionally claimed by setting the `max_velocity_interface` and `max_effort_interface` parameter, respectively. By default, the controller will try to claim position and velocity state interfaces. The claimed state interfaces can be configured by setting the `state_interfaces` parameter.

Parameters

This controller uses the generate_parameter_library to handle its parameters.

action_monitor_rate (double)
Hz

Read only: True

Default: 20.0

Constraints:

- greater than or equal to 0.1

joint (string)
Read only: True

Constraints:

- parameter is not empty

state_interfaces (string_array)
Default: {“position”, “velocity”}

goal_tolerance (double)
Default: 0.01

Constraints:

- greater than or equal to 0.0

allow_stalling (bool)
Allow stalling will make the action server return success if the gripper stalls when moving to the goal

Default: false

stall_velocity_threshold (double)
stall velocity threshold

Default: 0.001

stall_timeout (double)
stall timeout

Default: 1.0

Constraints:

- greater than or equal to 0.0

max_effort_interface (string)
Controller will claim the specified effort interface, e.g. robotiq_85_left_knuckle_joint/max_effort_interface.

Default: “”

max_effort (double)
Default effort used for grasping (Newtons)

Default: 10.0

Constraints:

- greater than or equal to 0.0

max_velocity_interface (string)
Controller will claim the speed specified interface, e.g. robotiq_85_left_knuckle_joint/max_effort_interface.

Default: “”

max_velocity (double)
Default target velocity used for grasping (meters/second)

Default: 0.1

Constraints:

- greater than or equal to 0.0

3.3.6 PID Controller

PID Controller implementation that uses PidROS implementation from control_toolbox package. The controller can be used directly by sending references through a topic or in a chain having preceding or following controllers. It also enables to use the first derivative of the reference and its feedback to have second-order PID control.

Depending on the reference/state and command interface of the hardware a different parameter setup of PidROS should be used as for example:

- reference/state POSITION; command VELOCITY –> PI CONTROLLER
- reference/state VELOCITY; command ACCELERATION –> PI CONTROLLER
- reference/state VELOCITY; command POSITION –> PD CONTROLLER
- reference/state ACCELERATION; command VELOCITY –> PD CONTROLLER
- reference/state POSITION; command POSITION –> PID CONTROLLER
- reference/state VELOCITY; command VELOCITY –> PID CONTROLLER
- reference/state ACCELERATION; command ACCELERATION –> PID CONTROLLER
- reference/state EFFORT; command EFFORT –> PID CONTROLLER

Note: Theoretically one can misuse *Joint Trajectory Controller (JTC)* for the same purpose by sending only one reference point into it. Nevertheless, this is not recommended. JTC should be used if you need to interpolate between trajectory points using linear, cubic or quintic interpolation. PID Controller doesn't do that. PID term of JTC has different purpose - it enables commanding only `velocity` or `effort` interfaces to hardware.

Execution logic of the controller

The controller can be also used in "feed-forward" mode where feed-forward gain is used to increase controllers dynamics. If one type of the reference and state interfaces is used, only immediate error is used. If there are two, then the second interface type is considered to be the first derivative of the first type. For example a valid combination would be `position` and `velocity` interface types.

Using the controller

Pluginlib-Library: pid_controller Plugin name: pid_controller/PidController

Description of controller's interfaces

References (from a preceding controller)

- <reference_and_state_dof_names[i]>/<reference_and_state_interfaces[j]> [double] **NOTE**: `reference_and_state_dof_names[i]` can be from `reference_and_state_dof_names` parameter, or if it is empty then `dof_names`.

Commands

- <dof_names[i]>/<command_interface> [double]

States

- <reference_and_state_dof_names[i]>/<reference_and_state_interfaces[j]> [double] **NOTE**: `reference_and_state_dof_names[i]` can be from `reference_and_state_dof_names` parameter, or if it is empty then `dof_names`.

Subscribers

If controller is not in chained mode (`in_chained_mode == false`):

- <controller_name>/reference [control_msgs/msg/MultiDOFCommand]

If controller parameter `use_external_measured_states` is true:

- <controller_name>/measured_state [control_msgs/msg/MultiDOFCommand]

Publishers

- <controller_name>/controller_state [control_msgs/msg/MultiDOFStateStamped]
- <controller_name>/<dof>/pid_state [control_msgs/msg/PidState]

Initially the PidState publisher is turned off. It can be turned on by using `gains.<dof>.activate_state_publisher` parameter.

Parameters

The PID controller uses the generate_parameter_library to handle its parameters.

List of parameters

dof_names (string_array)
Specifies dof_names or axes used by the controller. If 'reference_and_state_dof_names' parameter is defined, then only command dof names are defined with this parameter.

Read only: True

Default: {}

Constraints:

- contains no duplicates
- parameter is not empty

reference_and_state_dof_names (string_array)
(optional) Specifies dof_names or axes for getting reference and reading states. This parameter is only relevant when state dof names are different then command dof names, i.e., when a following controller is used.

Read only: True

Default: {}

Constraints:

- contains no duplicates

command_interface (string)
Name of the interface used by the controller for writing commands to the hardware.

Read only: True

Default: ""

Constraints:

- parameter is not empty

reference_and_state_interfaces (string_array)
Name of the interfaces used by the controller getting hardware states and reference commands. The second interface should be the derivative of the first.

Read only: True

Default: {}

Constraints:

- parameter is not empty
- length is greater than 0
- length is less than 3

set_current_state_as_first_setpoint (bool)
When true, the controller will set the current state as the first setpoint when the controller is activated. This can help to avoid large initial errors and sudden jumps in the control output when the controller starts.

Read only: True

Default: true

use_external_measured_states (bool)
Use external states from a topic instead from state interfaces.

Default: false

gains.<dof_names>.p (double)
Proportional gain for PID

Default: 0.0

gains.<dof_names>.i (double)
Integral gain for PID

Default: 0.0

gains.<dof_names>.d (double)
Derivative gain for PID

Default: 0.0

gains.<dof_names>.u_clamp_max (double)
Upper output clamp.

Default: std::numeric_limits<double>::infinity()

gains.<dof_names>.u_clamp_min (double)
Lower output clamp.

Default: -std::numeric_limits<double>::infinity()

gains.<dof_names>.i_clamp_max (double)
Upper integral clamp.

Default: std::numeric_limits<double>::infinity()

gains.<dof_names>.i_clamp_min (double)
Lower integral clamp.

Default: -std::numeric_limits<double>::infinity()

gains.<dof_names>.antiwindup_strategy (string)
Specifies the anti-windup strategy. Options: 'back_calculation', 'conditional_integration' or 'none'. Note that the 'back_calculation' strategy use the tracking_time_constant parameter to tune the anti-windup behavior.

Default: "none"

Constraints:

- one of the specified values: ['back_calculation', 'conditional_integration', 'none']

gains.<dof_names>.tracking_time_constant (double)
Specifies the tracking time constant for the 'back_calculation' strategy. If set to 0.0 when this strategy is selected, a recommended default value will be applied.

Default: 0.0

gains.<dof_names>.error_deadband (double)
Is used to stop integration when the error is within the given range.

Default: 1e-16

gains.<dof_names>.feedforward_gain (double)
Gain for the feed-forward part.

Default: 0.0

gains.<dof_names>.angle_wraparound (bool)
For joints that wrap around (i.e., are continuous). Normalizes position-error to -pi to pi.

Default: false

gains.<dof_names>.save_i_term (bool)
Indicating if integral term is retained after re-activation

Default: true

gains.<dof_names>.activate_state_publisher (bool)
Individual state publisher activation for each DOF. If true, the controller will publish the state of each DOF to the topic */<controller_name>/<dof_name>/pid_state*.

Default: false

An example parameter file

An example parameter file for this controller can be found in the test folder (standalone):

```yaml
/**:
  ros__parameters:
# TODO(christohfroehlich): Remove this global parameters once the deprecated␣
↪antiwindup parameters are removed.
    gains:
      joint1: {antiwindup_strategy: "none", i_clamp_max: .inf, i_clamp_min: -.inf}
      joint2: {antiwindup_strategy: "none", i_clamp_max: .inf, i_clamp_min: -.inf}

test_pid_controller:
  ros__parameters:
    dof_names:
      - joint1

    command_interface: position

    reference_and_state_interfaces: ["position"]

    set_current_state_as_first_setpoint: true

    gains:
      joint1: {p: 1.0, i: 2.0, d: 3.0, u_clamp_max: 5.0, u_clamp_min: -5.0}

test_pid_controller_unlimited:
  ros__parameters:
    dof_names:
      - joint1

    command_interface: position

    reference_and_state_interfaces: ["position"]

    set_current_state_as_first_setpoint: true

    gains:
      joint1: {p: 1.0, i: 2.0, d: 3.0}

test_pid_controller_angle_wraparound_on:
  ros__parameters:
```

(continues on next page)

(continued from previous page)

```
    dof_names:
      - joint1

    command_interface: position

    reference_and_state_interfaces: ["position"]

    set_current_state_as_first_setpoint: true

    gains:
      joint1: {p: 1.0, i: 2.0, d: 3.0, angle_wraparound: true}

test_pid_controller_with_feedforward_gain:
  ros__parameters:
    dof_names:
      - joint1

    command_interface: position

    reference_and_state_interfaces: ["position"]

    set_current_state_as_first_setpoint: true

    gains:
      joint1: {p: 0.5, i: 0.0, d: 0.0, feedforward_gain: 1.0}

test_pid_controller_with_feedforward_gain_dual_interface:
  ros__parameters:
    dof_names:
      - joint1
      - joint2

    command_interface: velocity

    reference_and_state_interfaces: ["position", "velocity"]

    set_current_state_as_first_setpoint: true

    gains:
      joint1: {p: 0.5, i: 0.3, d: 0.4, feedforward_gain: 1.0}
      joint2: {p: 0.5, i: 0.3, d: 0.4, feedforward_gain: 1.0}

test_save_i_term_off:
  ros__parameters:
    dof_names:
      - joint1

    command_interface: position

    reference_and_state_interfaces: ["position"]

    set_current_state_as_first_setpoint: true

    gains:
      joint1: {p: 1.0, i: 2.0, d: 3.0, save_i_term: false}

test_save_i_term_on:
```

(continues on next page)

(continued from previous page)

```
  ros__parameters:
    dof_names:
      - joint1

    command_interface: position

    reference_and_state_interfaces: ["position"]

    set_current_state_as_first_setpoint: true

    gains:
      joint1: {p: 1.0, i: 2.0, d: 3.0, save_i_term: true}

test_pid_controller_no_first_setpoint:
  ros__parameters:
    dof_names:
      - joint1

    command_interface: position

    reference_and_state_interfaces: ["position"]

    set_current_state_as_first_setpoint: false

    gains:
      joint1: {p: 1.0, i: 2.0, d: 3.0, u_clamp_max: 5.0, u_clamp_min: -5.0}
```

or as preceding controller:

```
test_pid_controller:
  ros__parameters:
    dof_names:
      - joint1

    command_interface: position

    reference_and_state_interfaces: ["position"]

    reference_and_state_dof_names:
      - joint1state
```

3.3.7 position_controllers

This is a collection of controllers that work using the "position" joint command interface but may accept different joint-level commands at the controller level, e.g. controlling the position on a certain joint to achieve a set velocity.

The package contains the following controllers:

position_controllers/JointGroupPositionController

Warning: `position_controllers/JointGroupPositionController` is deprecated. Use *forward_command_controller* instead by adding the `interface_name` parameter and set it to `position`.

This is specialization of the *forward_command_controller* that works using the "position" joint interface.

ROS 2 interface of the controller

Topics

~/commands (input topic) [std_msgs::msg::Float64MultiArray]
Joints' position commands

Parameters

This controller overrides the interface parameter from *forward_command_controller*, and the `joints` parameter is the only one that is required.

An example parameter file is given here

```
controller_manager:
  ros__parameters:
    update_rate: 100  # Hz

    position_controller:
      type: position_controllers/JointGroupPositionController

position_controller:
  ros__parameters:
    joints:
      - slider_to_cart
```

3.3.8 velocity_controllers

This is a collection of controllers that work using the "velocity" joint command interface but may accept different joint-level commands at the controller level, e.g. controlling the velocity on a certain joint to achieve a set position.

The package contains the following controllers:

velocity_controllers/JointGroupVelocityController

Warning: `velocity_controllers/JointGroupVelocityController` is deprecated. Use *forward_command_controller* instead by adding the `interface_name` parameter and set it to `velocity`.

This is specialization of the *forward_command_controller* that works using the "velocity" joint interface.

ROS 2 interface of the controller

Topics

~/commands (input topic) [std_msgs::msg::Float64MultiArray]
Joints' velocity commands

Parameters

This controller overrides the interface parameter from *forward_command_controller*, and the `joints` parameter is the only one that is required.

An example parameter file is given here

```
controller_manager:
  ros__parameters:
    update_rate: 100  # Hz

    velocity_controller:
      type: velocity_controllers/JointGroupVelocityController

velocity_controller:
  ros__parameters:
    joints:
      - slider_to_cart
```

3.3.9 gpio_controllers

This is a collection of controllers for hardware interfaces of type GPIO (`<gpio>` tag in the URDF).

gpio_command_controller

gpio_command_controller let the user expose command interfaces of given GPIO interfaces and publishes state interfaces of the configured command interfaces.

Description of controller's interfaces

- `/<controller_name>/gpio_states` [`control_msgs/msg/DynamicJointState`]: Publishes all state interfaces of the given GPIO interfaces.
- `/<controller_name>/commands` [`control_msgs/msg/DynamicJointState`]: A subscriber for configured command interfaces.

Parameters

This controller uses the generate_parameter_library to handle its parameters. The parameter definition file located in the src folder contains descriptions for all the parameters used by the controller.

gpios (string_array)
List of gpios

Read only: True

Constraints:

- length is greater than 0
- contains no duplicates

command_interfaces.<gpios>.interfaces (string_array)
List of command interfaces for each gpio.

Read only: True

Default: {}

Constraints:

- contains no duplicates

state_interfaces.<gpios>.interfaces (string_array)
List of state interfaces for each gpio. If empty all available gpios' states are used.

Read only: True

Default: {}

Constraints:

- contains no duplicates

The controller expects at least one GPIO interface and the corresponding command interface names or state interface. However, these Command and State interfaces are optional. The controller behaves as a broadcaster when no Command Interface is present, thereby publishing the configured GPIO state interfaces if set, else the one present in the URDF.

Note: When no state interface is provided in the param file, the controller will try to use state_interfaces from ros2_control's config placed in the URDF for configured gpio interfaces. However, command interfaces will not be configured based on the available URDF setup.

```
gpio_command_controller:
  ros__parameters:
    type: gpio_controllers/GpioCommandController
    gpios:
      - Gpio1
```

(continues on next page)

(continued from previous page)

```
    - Gpio2
  command_interfaces:
    Gpio1:
      - interfaces:
        - dig.1
        - dig.2
        - dig.3
    Gpio2:
      - interfaces:
        - ana.1
        - ana.2
  state_interfaces:
    Gpio2:
      - interfaces:
        - ana.1
        - ana.2
```

With the above-defined controller configuration, the controller will accept commands for all gpios' interfaces and will only publish the state of Gpio2.

3.3.10 motion_primitive_controllers

Package to control robots using motion primitives like LINEAR_JOINT (PTP/ MOVEJ), LINEAR_CARTESIAN (LIN/ MOVEL) and CIRCULAR_CARTESIAN (CIRC/ MOVEC)

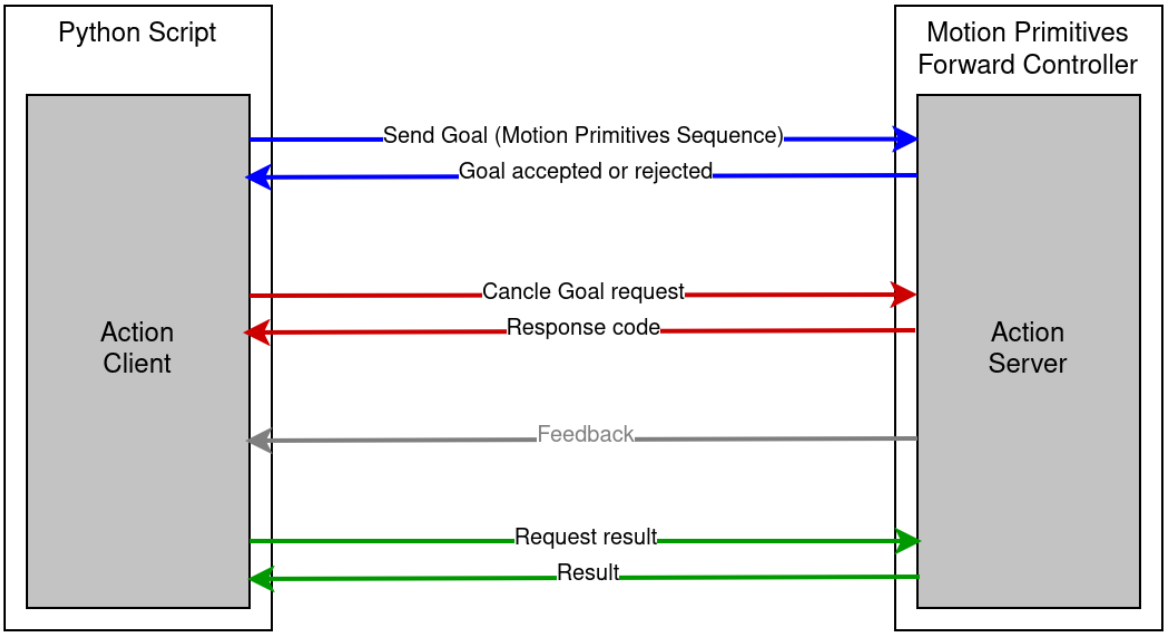
3.3.11 Description

This project provides an interface for sending motion primitives to an industrial robot controller using the `ExecuteMotionPrimitiveSequence.action` action from control_msgs. The controller receives the primitives via the action interface and forwards them through command interfaces to the robot-specific hardware interface. Currently, hardware interfaces for Universal Robots and KUKA Robots are implemented.

- Supported motion primitives:
  - `LINEAR_JOINT`
  - `LINEAR_CARTESIAN`
  - `CIRCULAR_CARTESIAN`

If multiple motion primitives are passed to the controller via the action, the controller forwards them to the hardware interface as a sequence. To do this, it first sends `MOTION_SEQUENCE_START`, followed by each individual primitive, and finally `MOTION_SEQUENCE_END`. All primitives between these two markers will be executed as a single, continuous sequence. This allows seamless transitions (blending) between primitives.

The action interface also allows stopping the current execution of motion primitives. When a stop request is received, the controller sends `STOP_MOTION` to the hardware interface, which then halts the robot's movement. Once the controller receives confirmation that the robot has stopped, it sends `RESET_STOP` to the hardware interface. After that, new commands can be sent.



This can be done, for example, via a Python script as demonstrated in the `example python script` in the `Universal_Robots_ROS2_Driver` package.

Command and State Interfaces

To transmit the motion primitives, the following command and state interfaces are required. All interfaces use the naming scheme `tf_prefix_ + "motion_primitive/<interface name>"` where the `tf_prefix` is provided to the controller as a parameter.

Command Interfaces

These interfaces are used to send motion primitive data to the hardware interface:

- `motion_type`: Type of motion primitive (LINEAR_JOINT, LINEAR_CARTESIAN, CIRCULAR_CARTESIAN)
- `q1` – `q6`: Target joint positions for joint-based motion
- `pos_x`, `pos_y`, `pos_z`: Target Cartesian position
- `pos_qx`, `pos_qy`, `pos_qz`, `pos_qw`: Orientation quaternion of the target pose
- `pos_via_x`, `pos_via_y`, `pos_via_z`: Intermediate via-point position for circular motion
- `pos_via_qx`, `pos_via_qy`, `pos_via_qz`, `pos_via_qw`: Orientation quaternion of via-point
- `blend_radius`: Blending radius for smooth transitions
- `velocity`: Desired motion velocity
- `acceleration`: Desired motion acceleration
- `move_time`: Optional duration for time-based execution (For LINEAR_JOINT and LINEAR_CARTESIAN. If move_time > 0, velocity and acceleration are ignored)

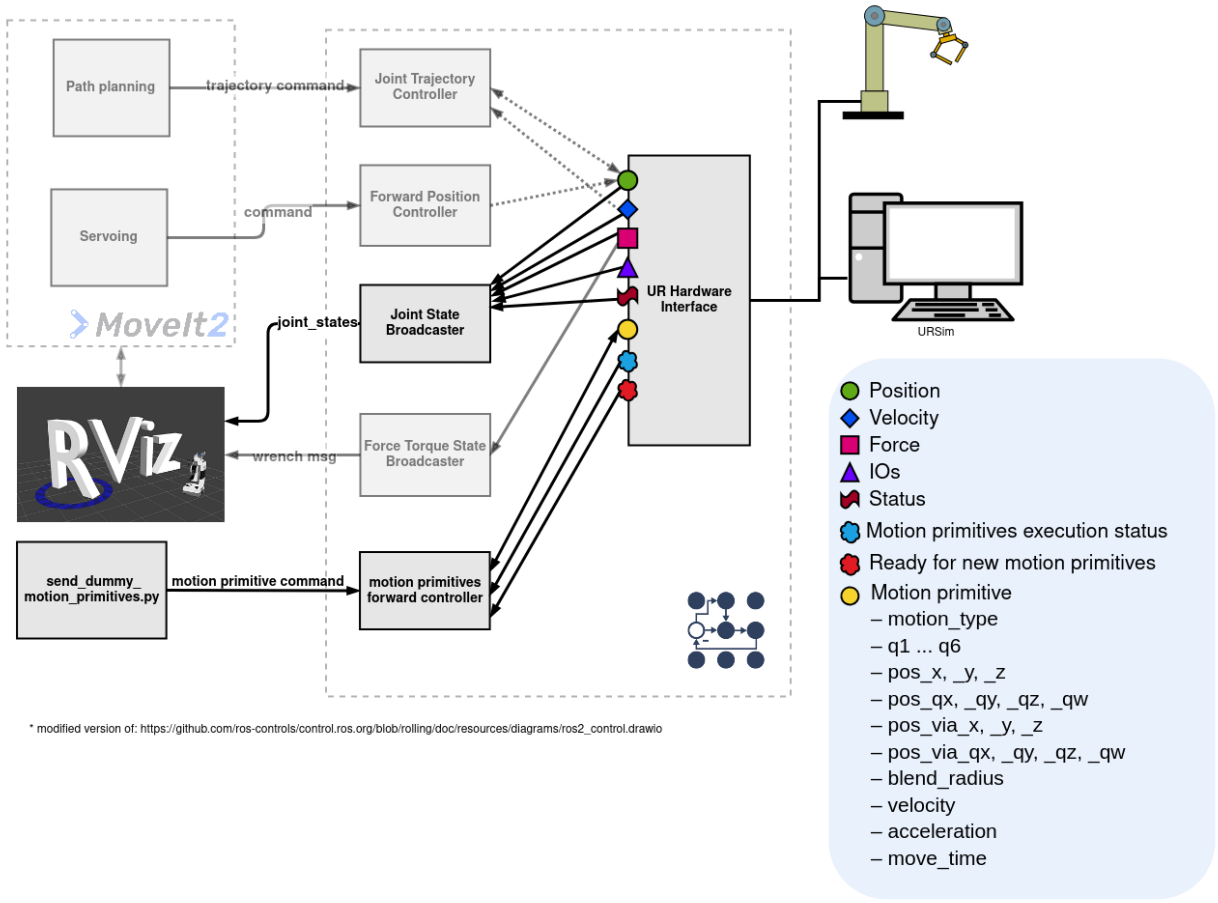
State Interfaces

These interfaces are used to communicate the internal status of the hardware interface back to the `motion_primitives_forward_controller`.

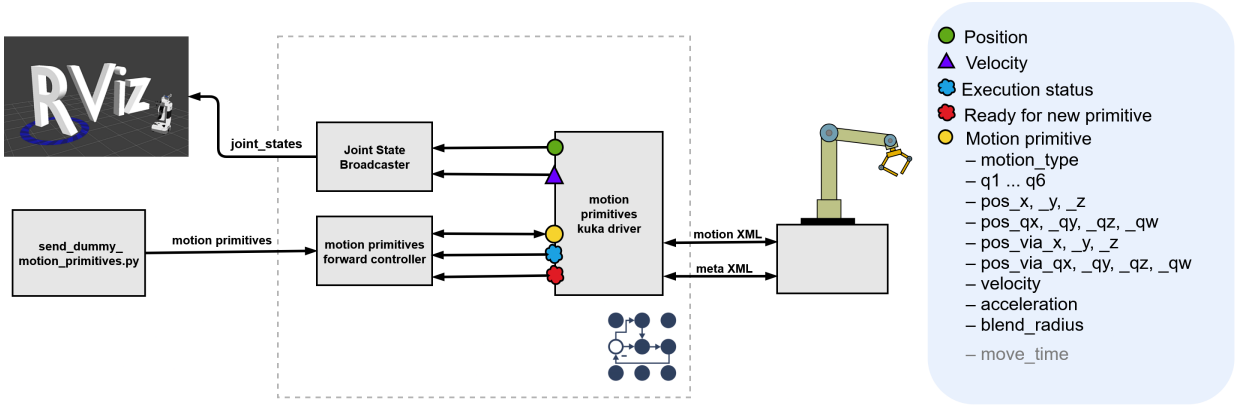
- `execution_status`: Indicates the current execution state of the primitive. Possible values are:
  - `IDLE`: No motion in progress
  - `EXECUTING`: Currently executing a primitive
  - `SUCCESS`: Last command finished successfully
  - `ERROR`: An error occurred during execution
  - `STOPPING`: The hardware interface has received the `STOP_MOTION` command, but the robot has not yet come to a stop.
  - `STOPPED`: The robot was stopped using the `STOP_MOTION` command and must be reset with the `RESET_STOP` command before executing new commands.
- `ready_for_new_primitive`: Boolean flag indicating whether the interface is ready to receive a new motion primitive

3.3.12 Architecture Overview

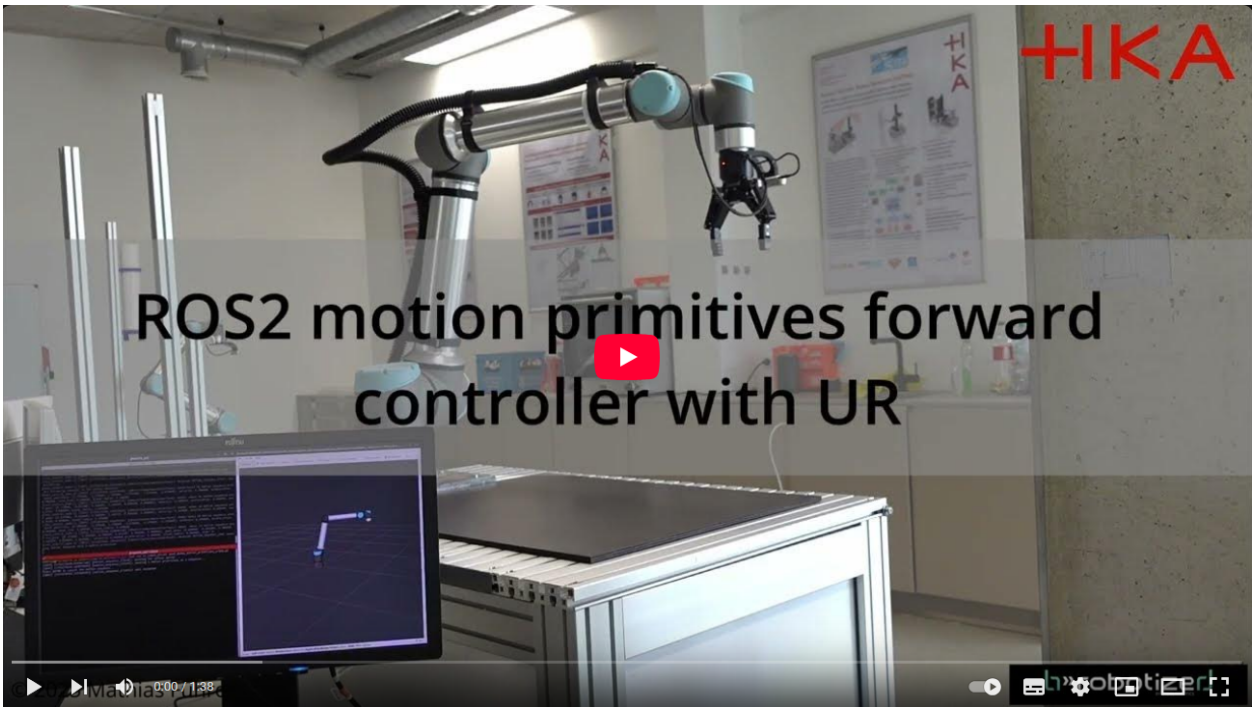
Architecture for a UR robot with `Universal_Robots_ROS2_Driver` in motion primitives mode.



Architecture for a KUKA robot with `kuka_eki_motion_primitives_hw_interface`.



3.3.13 Demo-Video with UR10e



3.3.14 Demo-Video with KR3

3.3.15 Parameters

This controller uses the generate_parameter_library to handle its parameters. The parameter definition file located in the src folder contains descriptions for all the parameters used by the controller.

tf_prefix (string)
All interfaces use the naming scheme *tf_prefix_ + motion_primitive/<interface name>*

Read only: True

Default: “”

An example parameter file for this controller can be found in the test directory:

```
test_motion_primitives_forward_controller:
  ros__parameters:
    tf_prefix: ""
```

3.4 Broadcasters

Broadcasters are used to publish sensor data from hardware components to ROS topics. In the sense of ros2_control, broadcasters are still controllers using the same controller interface as the other controllers above.

3.4.1 Force Torque Sensor Broadcaster

Broadcaster of messages from force/torque state interfaces of a robot or sensor. The published message type is `geometry_msgs/msg/WrenchStamped`.

The broadcaster also supports filtering the force/torque readings using a filter chain, enabling the sequential application of multiple filters. In this case, an additional topic with the `_filtered` suffix will be published containing the final result. For more details on filters, refer to the filters package repository. See the parameter section for instructions on configuring a filter chain with an arbitrary number of filters.

The controller is a wrapper around `ForceTorqueSensor` semantic component (see `controller_interface` package).

Parameters

This controller uses the generate_parameter_library to handle its parameters. The parameter definition file located in the src folder contains descriptions for all the parameters used by the controller.

The interfaces can be defined in two ways, using the `sensor_name` or the `interface_names` parameter: Those two parameters cannot be defined at the same time.

The filter chain is configured as a sequential list of filters under the `sensor_filter_chain` parameter, where each filter is identified by the key `filterN`, with `N` representing its position in the chain (e.g., `filter1`, `filter2`, etc.). Each filter entry must specify the `name` and `type` of the plugin, along with any additional parameters required by that specific filter plugin. The chain will use the pluginlib library to load each filter at runtime, passing the specified parameters.

The chain processes the data sequentially, passing the output of one filter as the input to the next.

Full list of parameters:

frame_id (string)
Sensor’s frame_id in which values are published.

Default: “”

Constraints:

- parameter is not empty

sensor_name (string)
Name of the sensor used as prefix for interfaces if there are no individual interface names defined. If used, standard interface names for a 6D FTS will be used: `<sensor_name>/force.x, ..., <sensor_name>/torque.z`

Default: “”

interface_names
(optional) Defines custom, per axis interface names. This is used if different prefixes, i.e., sensor name, or non-standard interface names are used. It is sufficient that only one `interface_name` is defined. This enables the broadcaster to use force sensing cells with less than six measuring axes. An example definition is:

```
interface_names:
  force:
    x: example_name/example_interface
```

interface_names.force.x (string)
Name of the state interface with force values on ‘x’ axis.

Default: “”

interface_names.force.y (string)
Name of the state interface with force values on ‘y’ axis.

Default: “”

interface_names.force.z (string)
Name of the state interface with force values on ‘z’ axis.

Default: “”

interface_names.torque.x (string)
Name of the state interface with torque values around ‘x’ axis.

Default: “”

interface_names.torque.y (string)
Name of the state interface with torque values around ‘y’ axis.

Default: “”

interface_names.torque.z (string)
Name of the state interface with torque values around ‘z’ axis.

Default: “”

offset.force.x (double)
The offset of force values on ‘x’ axis.

Default: 0.0

offset.force.y (double)
The offset of force values on ‘y’ axis.

Default: 0.0

offset.force.z (double)
The offset of force values on ‘z’ axis.

Default: 0.0

offset.torque.x (double)
The offset of torque values around 'x' axis.

Default: 0.0

offset.torque.y (double)
The offset of torque values around 'y' axis.

Default: 0.0

offset.torque.z (double)
The offset of torque values around 'z' axis.

Default: 0.0

multiplier.force.x (double)
The multiplier of force value on 'x' axis.

Default: 1.0

multiplier.force.y (double)
The multiplier of force value on 'y' axis.

Default: 1.0

multiplier.force.z (double)
The multiplier of force value on 'z' axis.

Default: 1.0

multiplier.torque.x (double)
The multiplier of torque value around 'x' axis.

Default: 1.0

multiplier.torque.y (double)
The multiplier of torque value around 'y' axis.

Default: 1.0

multiplier.torque.z (double)
The multiplier of torque value around 'z' axis.

Default: 1.0

An example parameter file for this controller can be found in the test directory:

```yaml
test_force_torque_sensor_broadcaster:
  ros__parameters:
    frame_id:  "fts_sensor_frame"
test_force_torque_sensor_broadcaster_with_chain:
  ros__parameters:
    frame_id:  "fts_sensor_frame"
    sensor_name: "fts_sensor"
    sensor_filter_chain:
      filter1:
        name: dummy
        type: filters/IncrementFilterWrench
```

3.4.2 Wrench Transformer Node

The package provides a standalone ROS 2 node `wrench_transformer_node` that transforms wrench messages published by the `ForceTorqueSensorBroadcaster` controller to different target frames using TF2. This is useful when applications need force/torque data in coordinate frames other than the sensor frame.

The node subscribes to wrench messages from the broadcaster (either raw or filtered) and publishes transformed versions to separate topics for each target frame.

Usage

The wrench transformer can be launched with target frames passed directly as positional arguments:

```
ros2 run force_torque_sensor_broadcaster wrench_transformer_node frame1 frame2
```

Target frames may also be set via the `target_frames` parameter:

```
ros2 run force_torque_sensor_broadcaster wrench_transformer_node \
  --ros-args -p target_frames:="['frame1','frame2']"
```

Positional arguments override the parameter value when both are provided.

Wrench Transformer Parameters

The wrench transformer uses the generate_parameter_library to handle its parameters. The parameter definition file for the wrench transformer contains descriptions for all the parameters.

Full list of parameters:

target_frames (string_array)
Array of target frame names to transform the input wrench to. For each frame, a separate output topic will be published.

Default: {}

Constraints:

- parameter is not empty

tf_timeout (double)
Timeout in seconds for TF lookups when transforming wrenches between frames.

Default: 0.1

Topics

The node subscribes to:

- `~/wrench` (raw wrench messages). To subscribe to filtered wrench messages, use topic remapping: `ros2 run ... --ros-args -r ~/wrench:=<namespace>/wrench_filtered`

The node publishes:

- `<namespace>/<target_frame>/wrench` for each target frame specified in `target_frames`
  - If the node is in the root namespace (`/`), the namespace defaults to the node name (e.g., `/fts_wrench_transformer/<target_frame>/wrench`)

- If the input topic is remapped to a filtered topic (contains "filtered" in the name), the output topics automatically append `_filtered` suffix (e.g., `<namespace>/<target_frame>/wrench_filtered`)
- This allows users to distinguish between transformed raw wrench data and transformed filtered wrench data

3.4.3 IMU Sensor Broadcaster

Broadcaster of messages from IMU sensor. The published message type is `sensor_msgs/msg/Imu`.

The controller is a wrapper around `IMUSensor` semantic component (see `controller_interface` package).

Parameters

This controller uses the generate_parameter_library to handle its parameters. The parameter definition file located in the src folder contains descriptions for all the parameters used by the controller.

List of parameters

sensor_name (string)
Defines sensor name used as prefix for its interfaces. Interface names are: `<sensor_name>/orientation.x, ..., <sensor_name>/angular_velocity.x, ..., <sensor_name>/linear_acceleration.x`.

Read only: True

Default: ""

Constraints:

- parameter is not empty

frame_id (string)
Sensor's frame_id in which values are published.

Read only: True

Default: ""

Constraints:

- parameter is not empty

static_covariance_orientation (double_array)
Static orientation covariance. Row major about x, y, z axes

Read only: True

Default: {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0}

Constraints:

- length must be equal to 9

static_covariance_angular_velocity (double_array)
Static angular velocity covariance. Row major about x, y, z axes

Read only: True

Default: {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0}

Constraints:

- length must be equal to 9

static_covariance_linear_acceleration (double_array)
Static linear acceleration covariance. Row major about x, y, z axes

Read only: True

Default: {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0}

Constraints:

- length must be equal to 9

rotation_offset.roll (double)
Roll offset value in radians. (Z-Y'-X" convention)

Read only: True

Default: 0.0

rotation_offset.pitch (double)
Pitch offset value in radians. (Z-Y'-X" convention)

Read only: True

Default: 0.0

rotation_offset.yaw (double)
Yaw offset value in radians. (Z-Y'-X" convention)

Read only: True

Default: 0.0

An example parameter file

An example parameter file for this controller can be found in the test directory:

```
test_imu_sensor_broadcaster:
  ros__parameters:

    sensor_name: "imu_sensor"
    frame_id:  "imu_sensor_frame"
```

3.4.4 joint_state_broadcaster

The broadcaster reads all state interfaces and reports them on `/joint_states` and `/dynamic_joint_states`.

Commands

Broadcasters are not real controllers, and therefore take no commands.

Hardware interface type

By default, all available *joint state interfaces* are used, unless configured otherwise. In the latter case, resulting interfaces is defined by a "matrix" of interfaces defined by the cross-product of the `joints` and `interfaces` parameters. If some requested interfaces are missing, the controller will print a warning about that, but work for other interfaces. If none of the requested interface are not defined, the controller returns error on activation.

Published topics

- `/joint_states` (`sensor_msgs/msg/JointState`):

  Publishes *movement-related* interfaces only — `position`, `velocity`, and `effort` — for joints that provide them. If a joint does not expose a given movement interface, that field is omitted/left empty for that joint in the message.

- `/dynamic_joint_states` (`control_msgs/msg/DynamicJointState`):

  Publishes **all available state interfaces** for each joint. This includes the movement interfaces (position/velocity/effort) *and* any additional or custom interfaces provided by the hardware (e.g., temperature, voltage, torque sensor readings, calibration flags).

  The message maps `joint_names` to per-joint interface name lists and values. Example payload:

  ```
  joint_names: [joint1, joint2]
  interface_values:
    - interface_names: [position, velocity, effort]
      values: [1.5708, 0.0, 0.20]
    - interface_names: [position, temperature]
      values: [0.7854, 42.1]
  ```

  Use this topic if you need *every* reported interface, not just movement.

Note: If `use_local_topics` is set to `true`, both topics are published in the controller's namespace (e.g., `/my_state_broadcaster/joint_states` and `/my_state_broadcaster/dynamic_joint_states`). If `false` (default), they are published at the root (e.g., `/joint_states`).

Parameters

This controller uses the generate_parameter_library to handle its parameters. The parameter definition file located in the src folder contains descriptions for all the parameters used by the controller.

List of parameters

use_local_topics (bool)
Defining if `joint_states` and `dynamic_joint_states` messages should be published into local namespace, e.g., `/my_state_broadcaster/joint_states`.

Read only: True

Default: false

joints (string_array)
Parameter to support broadcasting of only specific joints and interfaces. It has to be used in combination with the

`interfaces` parameter. If either `joints` or `interfaces` is left empty, all available state interfaces will be published. Joint state broadcaster asks for access to all defined interfaces on all defined joints.

Read only: True

Default: {}

extra_joints (string_array)
Names of extra joints to be added to `joint_states` and `dynamic_joint_states` with state set to 0.

Read only: True

Default: {}

interfaces (string_array)
Parameter to support broadcasting of only specific joints and interfaces. It has to be used in combination with the `joints` parameter. If either `joints` or `interfaces` is left empty, all available state interfaces will be published.

Read only: True

Default: {}

map_interface_to_joint_state
Optional parameter (map) providing mapping between custom interface names to standard fields in `joint_states` message. Usecases:

1. Hydraulics robots where feedback and commanded values often have an offset and reliance on open-loop control is common. Typically one would map both values in separate interfaces in the framework. To visualize those data multiple joint_state_broadcaster instances and robot_state_publishers would be used to visualize both values in RViz.
2. A robot provides multiple measuring techniques for its joint values which results in slightly different values. Typically one would use separate interface for providing those values in the framework. Using multiple joint_state_broadcaster instances we could publish and show both in RViz.

Format (each line is optional):

```
map_interface_to_joint_state:
    position: <custom_interface>
    velocity: <custom_interface>
    effort: <custom_interface>
```

Examples:

```
map_interface_to_joint_state:
    position: kf_estimated_position
```

```
map_interface_to_joint_state:
    velocity: derived_velocity
    effort: derived_effort
```

```
map_interface_to_joint_state:
    effort: torque_sensor
```

```
map_interface_to_joint_state:
    effort: current_sensor
```

map_interface_to_joint_state.position (string)
Read only: True

Default: “position”

map_interface_to_joint_state.velocity (string)
Read only: True

Default: “velocity”

map_interface_to_joint_state.effort (string)
Read only: True

Default: “effort”

use_urdf_to_filter (bool)
Uses the robot_description to filter the `joint_states` topic. If true, the broadcaster will publish the data of the joints present in the URDF alone. If false, the broadcaster will publish the data of any interface that has type `position`, `velocity`, or `effort`.

Read only: True

Default: true

frame_id (string)
The frame_id to be used in the published joint states. This parameter allows rviz2 to visualize the effort of the joints.

Read only: True

Default: “base_link”

publish_dynamic_joint_states (bool)
[Deprecated] Whether to publish dynamic joint states. This parameter is deprecated and will be removed in future releases.

Read only: True

Default: false

An example parameter file

```yaml
joint_state_broadcaster:
  ros__parameters:
    extra_joints: '{}'
    frame_id: base_link
    interfaces: '{}'
    joints: '{}'
    map_interface_to_joint_state:
      effort: effort
      position: position
      velocity: velocity
    publish_dynamic_joint_states: false
    use_local_topics: false
    use_urdf_to_filter: true
```

Order of the joints in the message

The order of the joints in the message can determined by 3 different parameter settings:

1. **No defined `joints` parameter and `use_urdf_to_filter` set to `false`:**
   The order of the joints in the message is the same as the order of the joints' state interfaces registered in the resource manager. This is typically the order in which the hardware components are loaded and configured.
2. **No defined `joints` parameter and `use_urdf_to_filter` set to `true`:**
   The order of the joints in the message is the same as the order of the joints in the URDF file, which is inherited from the loaded URDF model and independent of the order in the *ros2_control* tag.
3. **Defined `joints` parameter along with `interfaces` parameter:**
   The order of the joints in the message is the same as the order of the joints in the `joints` parameter.

   If the `joints` parameter is a subset of the total available joints in the URDF (or) the total available state interfaces, then only the joints in the `joints` parameter are published in the message.

   If any of the combinations of the defined `joints` parameter and `interfaces` parameter are not in the available state interfaces, the controller will fail to activate.

Note: If the `extra_joints` parameter is set, the joints in the `extra_joints` parameter are appended to the end of the joint names in the message.

3.4.5 Range Sensor Broadcaster

Broadcaster of messages from Range sensor. The published message type is `sensor_msgs/msg/Range`.

The controller is a wrapper around `RangeSensor` semantic component (see `controller_interface` package).

Parameters

The Range Sensor Broadcaster uses the generate_parameter_library to handle its parameters. The parameter definition file located in the src folder contains descriptions for all the parameters used by the controller.

List of parameters

sensor_name (string)
Name of the sensor used as prefix for interfaces if there are no individual interface names defined.

Default: ""

frame_id (string)
Sensor's frame_id in which values are published.

Default: ""

radiation_type (int)
The type of radiation used by the sensor / 0 = Ultrason / 1 = Infrared

Default: 0

field_of_view (double)
The size of the arc that the distance reading is valid for [rad]

Default: 0.52

min_range (double)
Minimum range value [m]

Default: 0.52

max_range (double)
Maximum range value [m]

Default: 4.0

variance (double)
Variance of the range value

Default: 0.0

An example parameter file

```yaml
range_sensor_broadcaster:
  ros__parameters:
    field_of_view: 0.52
    frame_id: ''
    max_range: 4.0
    min_range: 0.52
    radiation_type: 0.0
    sensor_name: ''
    variance: 0.0
```

An example parameter file for this controller can be found in the test directory:

```yaml
test_range_sensor_broadcaster:
  ros__parameters:
    # Setting mendatory parameters
    sensor_name: "range_sensor"
    frame_id: "range_sensor_frame"

    # Setting parameters with changed default value to check those are used
    radiation_type: 1
    field_of_view: 0.1
    min_range: 0.10
    max_range: 7.0
    variance: 1.0
```

3.4.6 Pose Broadcaster

Broadcaster for poses measured by a robot or a sensor. Poses are published as `geometry_msgs/msg/PoseStamped` messages and optionally as tf transforms.

The controller is a wrapper around the `PoseSensor` semantic component (see `controller_interface` package).

Parameters

This controller uses the generate_parameter_library to handle its parameters. The parameter definition file located in the src folder contains descriptions for all the parameters used by the controller.

List of parameters

frame_id (string)
frame_id in which values are published

Default: “”

Constraints:

- parameter is not empty

pose_name (string)
Base name used as prefix for controller interfaces. The state interface names are: `<pose_name>/position.x`, ..., `<pose_name>/position.z`, `<pose_name>/orientation.x`, ..., `<pose_name>/orientation.w`

Default: “”

Constraints:

- parameter is not empty

tf.enable (bool)
Whether to publish the pose as a tf transform

Default: true

tf.child_frame_id (string)
Child frame id of published tf transforms. Defaults to `pose_name` if left empty.

Default: “”

An example parameter file

An example parameter file for this controller can be found in the test directory:

```
test_pose_broadcaster:
  ros__parameters:
    pose_name: "test_pose"
    frame_id: "pose_frame"
```

3.4.7 GPS Sensor Broadcaster

Broadcaster of messages from GPS sensor. The published message type is `sensor_msgs/msg/NavSatFix`.

The controller is a wrapper around `GPSSensor` semantic component (see `controller_interface` package).

Parameters

This controller uses the generate_parameter_library to handle its parameters. The parameter definition file located in the src folder contains descriptions for all the parameters used by the controller.

List of parameters

sensor_name (string)
Defines sensor name used as prefix for its interfaces. Interface names are: `<sensor_name>/orientation.x, ..., <sensor_name>/angular_velocity.x, ..., <sensor_name>/linear_acceleration.x.`

Default: “”

Constraints:

- parameter is not empty

frame_id (string)
Sensor’s frame_id in which values are published.

Default: “”

Constraints:

- parameter is not empty

static_position_covariance (double_array)
Static position covariance.

Default: {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0}

Constraints:

- length must be equal to 9

read_covariance_from_interface (bool)
Read covariance from state interface

Default: false

An example parameter file

An example parameter file for this controller can be found in the test directory:

```
test_gps_sensor_broadcaster:
  ros__parameters:
    sensor_name: gps_sensor
    frame_id: gps_sensor_frame
    service: service_gps
```

3.4.8 State Interfaces Broadcaster

The State Interfaces Broadcaster publishes the state of specified hardware interfaces of double data type. The broadcaster publishes two topics:

- `/state_interfaces_broadcaster/names`: Publishes the names of the hardware interfaces being monitored with a message type of `control_msgs/msg/Keys`.
- `/state_interfaces_broadcaster/values`: Publishes the current state values of the specified hardware interfaces with a message type of `control_msgs/msg/Float64Values`.

Parameters

This controller uses the generate_parameter_library to handle its parameters. The parameter definition file located in the src folder contains descriptions for all the parameters used by the controller.

List of parameters

interfaces (string_array)
The list of hardware interfaces information to be published by the interfaces state broadcaster.

Read only: True

Default: {}

Constraints:

- parameter is not empty

An example parameter file

An example parameter file for this controller can be found in the test directory:

```
test_state_interfaces_broadcaster:
  ros__parameters:
    interfaces: ["joint1/position", "joint2/velocity"]
```

3.5 Filters

Chainable controllers for filtering of state interfaces. They export the filtered values as state interfaces, which can be used by other controllers or broadcasters, and don't publish to ROS topics.

3.5.1 Chained Filter Controller

This controller wraps around `filter_chain` library from the filters package. It allows to chain multiple filters together, where the output of one filter is the input of the next one. The controller can be used to apply a sequence of filters to a single interface, the same filter chain to multiple interfaces, or different filter chains to different interfaces.

Parameters

This controller uses the generate_parameter_library to handle its parameters. The parameter definition file located in the src folder contains descriptions for all the parameters used by the controller.

Full list of parameters:

input_interfaces (string_array)
Name of the input state interface

Read only: True

Default: {}

Constraints:

- parameter is not empty

output_interfaces (string_array)
Name of the output state interface

Read only: True

Default: {}

Constraints:

- parameter is not empty

An example parameter file for this controller can be found in the test directory for a single interface:

```yaml
test_chained_filter:
  ros__parameters:
    filter_chain:
      filter1:
        name: filter1
        type: "filters/MeanFilterDouble"
        params:
          number_of_observations: 2
    input_interfaces: ["wheel_left/position"]
    output_interfaces: ["wheel_left/filtered_position"]
```

or for multiple interfaces:

```yaml
test_chained_filter_multiple_interfaces:
  ros__parameters:
    filter_chain:
      filter1:
        name: position_filter
        type: "filters/MeanFilterDouble"
        params:
          number_of_observations: 2
    input_interfaces:
      - "wheel_left/position"
```

(continues on next page)

(continued from previous page)

```
      - "wheel_right/position"
    output_interfaces:
      - "wheel_left/filtered_position"
      - "wheel_right/filtered_position"

test_chained_filter_multiple_interfaces_config_per_input:
  ros__parameters:
    input_interfaces:
      - "wheel_left/position"
      - "wheel_right/position"
    output_interfaces:
      - "wheel_left/filtered_position"
      - "wheel_right/filtered_position"
    wheel_left/position:
      filter_chain:
        filter1:
          name: wheel_left_filter
          type: "filters/MeanFilterDouble"
          params:
            number_of_observations: 2
    wheel_right/position:
      filter_chain:
        filter1:
          name: wheel_right_filter
          type: "filters/MeanFilterDouble"
          params:
            number_of_observations: 3
```

3.6 Common Controller Parameters

Every controller and broadcaster has a few common parameters. They are optional, but if needed they have to be set before `onConfigure` transition to `inactive` state, see lifecycle documents. Once the controllers are already loaded, this transition is done using the service `configure_controller` of the controller_manager.

- `update_rate`: An unsigned integer parameter representing the rate at which every controller/broadcaster runs its update cycle. When unspecified, they run at the same frequency as the controller_manager.
- `is_async`: A boolean parameter that is needed to specify if the controller update needs to run asynchronously.

CHAPTER

FOUR

DEMOS

This GitHub Repository provides templates for the development of `ros2_control`-enabled robots and simple simulations to demonstrate and prove `ros2_control` concepts.

If you want to have a rather step by step manual how to do things with `ros2_control` checkout ros-control/roscon2022_workshop repository.

4.1 What you can find in this repository

This repository demonstrates the following `ros2_control` concepts:

- Creating a `HardwareInterface` for a System, Sensor, and Actuator.
- Creating a robot description in the form of URDF files.
- Loading the configuration and starting a robot using launch files.
- Control of a differential mobile base *DiffBot*.
- Control of two joints of *RRBot*.
- Control of a 6-degrees-of-freedom robot.
- Implementing a controller switching strategy for a robot.
- Using joint limits and transmission concepts in `ros2_control`.

4.2 Goals

The repository has two other goals:

1. Implements the example configuration described in the `ros-controls/roadmap` repository file components_architecture_and_urdf_examples.
2. The repository is a validation environment for `ros2_control` concepts, which can only be tested during run-time (e.g., execution of controllers by the controller manager, communication between robot hardware and controllers).

4.3 Examples Overview

Example 1: RRBot
RRBot - or "Revolute-Revolute Manipulator Robot" - a simple position controlled robot with one hardware interface. This example also demonstrates the switching between different controllers.

Example 2: DiffBot
DiffBot, or "Differential Mobile Robot", is a simple mobile base with differential drive. The robot is basically a box moving according to differential drive kinematics.

Example 3: "RRBot with multiple interfaces"
RRBot with multiple interfaces.

Example 4: "Industrial robot with integrated sensor"
RRBot with an integrated sensor.

Example 5: "Industrial robot with externally connected sensor"
RRBot with an externally connected sensor.

Example 6: "Modular robot with separate communication to each actuator"
The example shows how to implement robot hardware with separate communication to each actuator.

Example 7: "6-DOF robot"
A full tutorial for a 6 DOF robot for intermediate ROS 2 users.

Example 8: "Using transmissions"
RRBot with an exposed transmission interface.

Example 9: "Gazebo Classic"
Demonstrates how to switch between simulation and hardware.

Example 10: "GPIO interfaces"
Industrial robot with GPIO interfaces

Example 11: "CarlikeBot"
CarlikeBot with a bicycle steering controller

Example 12: "Controller chaining"
The example shows a simple chainable controller and its integration to form a controller chain to control the joints of *RRBot*.

Example 13: "Multi-robot system with hardware lifecycle management"
This example shows how to handle multiple robots in a single controller manager instance.

Example 14: "Modular robots with actuators not providing states and with additional sensors"
The example shows how to implement robot hardware with actuators not providing states and with additional sensors.

Example 15: "Using multiple controller managers"
This example shows how to integrate multiple robots under different controller manager instances.

Example 16: "DiffBot with chained controllers"
This example shows how to create chained controllers using diff_drive_controller and pid_controllers to control a differential drive robot.

Example 17: "RRBot with Hardware Component that publishes diagnostics"
This example shows how to publish diagnostics from a hardware component using the Executor passed from Controller Manager.

4.4 Installation

You can install the demos locally or use the provided docker file.

4.4.1 Local installation

If you have ROS 2 installed already, choose the right version of this documentation and branch of the `ros2_control_demos` repository matching you ROS 2 distribution, see this table.

Otherwise, install ROS 2 rolling on your computer.

Note: `ros2_control` and `ros2_controllers` packages are released and can be installed using a package manager. We provide officially released and maintained debian packages, which can easily be installed via aptitude. However, there might be cases in which not-yet released demos or features are only available through a source build in your own workspace.

Build from debian packages

Download the `ros2_control_demos` repository and install its dependencies with

```
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src
git clone https://github.com/ros-controls/ros2_control_demos -b master
cd ~/ros2_ws/
sudo apt-get update
rosdep update --rosdistro=$ROS_DISTRO
rosdep install --from-paths ./ -i -y --rosdistro ${ROS_DISTRO}
```

Now you can build the repository (source your ROS 2 installation first)

```
cd ~/ros2_ws/
. /opt/ros/${ROS_DISTRO}/setup.sh
colcon build --merge-install
```

Build from source

- Download all repositories

  ```
  mkdir -p ~/ros2_ws/src
  cd ~/ros2_ws/src
  git clone https://github.com/ros-controls/ros2_control_demos
  cd ~/ros2_ws/
  vcs import src < src/ros2_control_demos/ros2_control_demos.$ROS_DISTRO.repos
  rosdep update --rosdistro=$ROS_DISTRO
  sudo apt-get update
  ```

  If you want to install the development version of ros2_control having the latest feature, use this repos file instead

  ```
  vcs import src --input https://raw.githubusercontent.com/ros-controls/ros2_
  ↪control_ci/master/ros_controls.rolling-on-$ROS_DISTRO.repos
  ```

- Install dependencies:

```
rosdep install --from-paths src --ignore-src -r -y
```

- Build everything, e.g. with:

```
. /opt/ros/${ROS_DISTRO}/setup.sh
colcon build --symlink-install
```

- Do not forget to source `setup.bash` from the `install` folder!

4.4.2 Using Docker

First, build the dockerfile with

```
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src
git clone https://github.com/ros-controls/ros2_control_demos
cd ros2_control_demos
docker build . -t ros2_control_demos -f Dockerfile/Dockerfile
```

To view the robot

Docker now allows us to run the demo without the GUI if configured properly. Now we can view the robot by the following procedure:

After having ROS 2 installed on your local system (not inside the docker), we can use `rviz2` to visualize the robot state and `joint_state_publisher_gui` package to give manual joint values to the robot. To install the package you can run:

```
sudo apt-get install -y ros-${ROS_DISTRO}-joint-state-publisher-gui ros-${ROS_DISTRO}-
↪rviz2
```

Then we are ready to bring up all the components to view the robot. Let's start with the docker container by running the following command:

```
docker run -it --rm --name ros2_control_demos --net host ros2_control_demos ros2␣
↪launch ros2_control_demo_example_1 view_robot.launch.py gui:=false
```

Note: Depending on your machine settings, it might be possible that you have to omit `--net host`.

Now, we need to start `rviz2` to view the robot as well as `joint_state_publisher_gui`, each in their own terminals after sourcing our ROS 2 installation.

Terminal 1:

```
source /opt/ros/${ROS_DISTRO}/setup.bash
ros2 run joint_state_publisher_gui joint_state_publisher_gui
```

Terminal 2:

```
source /opt/ros/${ROS_DISTRO}/setup.bash
cd ~/ros2_ws
rviz2 -d src/ros2_control_demos/ros2_control_demo_description/rrbot/rviz/rrbot.rviz
```

Now, you can see the robot moving by changing the values of the joints by moving the sliders around in the `joint_state_publisher_gui`.

To run the ros2_control demos

The following command runs the demo without the GUI:

```
docker run -it --rm --name ros2_control_demos --net host ros2_control_demos
```

Note: Depending on your machine settings, it might be possible that you have to omit `--net host`.

Then on your local machine, you can run rviz2 with the config file specified:

```
cd ~/ros2_ws
source /opt/ros/${ROS_DISTRO}/setup.sh
rviz2 -d src/ros2_control_demos/ros2_control_demo_description/rrbot/rviz/rrbot.rviz
```

You can also run other commands or launch files from the docker, e.g.

```
docker run -it --rm --name ros2_control_demos --net host ros2_control_demos ros2␣
↪launch ros2_control_demo_example_2 diffbot.launch.py
```

or launch a second terminal inside the docker container by

```
docker exec -it ros2_control_demos bash
```

4.5 Quick Hints

These are some quick hints, especially for those coming from a ROS1 control background:

- There are now three categories of hardware components: *Sensor*, *Actuator*, and *System*. *Sensor* is for individual sensors; *Actuator* is for individual actuators; *System* is for any combination of multiple sensors/actuators. You could think of a Sensor as read-only. All components are used as plugins and therefore exported using `PLUGINLIB_EXPORT_CLASS` macro.
- *ros(1)_control* only allowed three hardware interface types: position, velocity, and effort. *ros2_control* allows you to create any interface type by defining a custom string. For example, you might define a `position_in_degrees` or a `temperature` interface. The most common (position, velocity, acceleration, effort) are already defined as constants in hardware_interface/types/hardware_interface_type_values.hpp.
- In ros2_control, all parameters for the driver are specified in the URDF. The ros2_control framework uses the **<ros2_control>** tag in the URDF.
- Joint names in <ros2_control> tags in the URDF must be compatible with the controller's configuration.

4.6 Examples

4.6.1 Example 1: RRBot

RRBot, or "Revolute-Revolute Manipulator Robot", is a simple 3-linkage, 2-joint arm that we will use to demonstrate various features.

It is essentially a double inverted pendulum and demonstrates some fun control concepts within a simulator and was originally introduced for Gazebo tutorials.

For *example_1*, the hardware interface plugin is implemented having only one interface.

- The communication is done using proprietary API to communicate with the robot control box.
- Data for all joints is exchanged at once.
- Examples: KUKA RSI

The *RRBot* URDF files can be found in the `description/urdf` folder.

Note: The commands below are given for a local installation of this repository and its dependencies as well as for running them from a docker container. For more information on the docker usage see *Using Docker*.

Tutorial steps

1. (Optional) To check that *RRBot* descriptions are working properly use following launch commands

   Local

   ```
   ros2 launch ros2_control_demo_example_1 view_robot.launch.py
   ```

   Docker

   Let's start with the docker container by running the following command:

   ```
   docker run -it --rm --name ros2_control_demos --net host ros2_control_demos ros2␣
   ↪launch ros2_control_demo_example_1 view_robot.launch.py gui:=false
   ```

   Now, we need to start `joint_state_publisher_gui` as well as `rviz2` to view the robot, each in their own terminals after sourcing our ROS 2 installation.

   ```
   source /opt/ros/${ROS_DISTRO}/setup.bash
   ros2 run joint_state_publisher_gui joint_state_publisher_gui
   ```

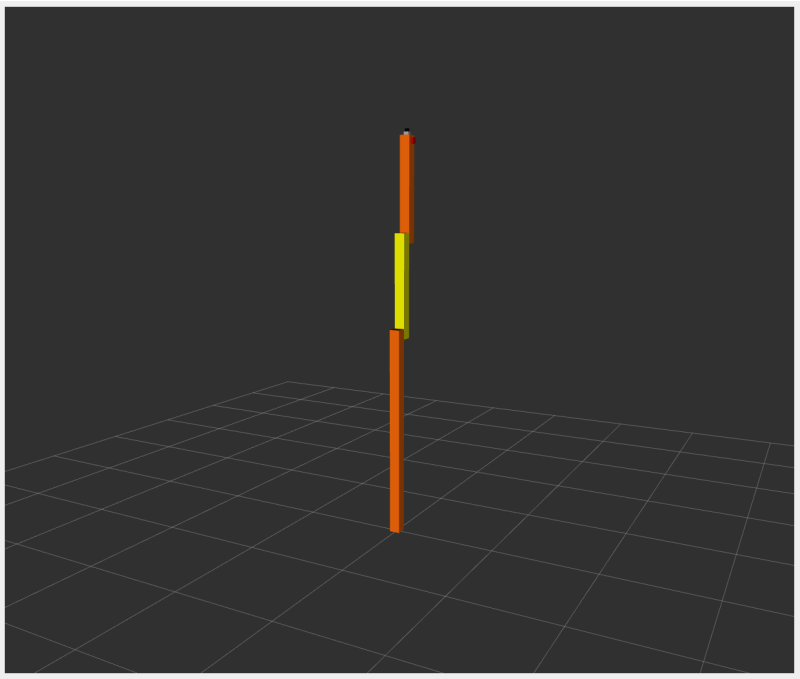
   The *RViz* setup can be recreated following these steps:

   - The robot models can be visualized using `RobotModel` display using `/robot_description` topic.
   - Or you can simply open the configuration from `ros2_control_demo_description/rrbot/rviz` folder manually or directly by executing from another terminal

   ```
   source /opt/ros/${ROS_DISTRO}/setup.bash
   rviz2 -d src/ros2_control_demos/ros2_control_demo_description/rrbot/rviz/rrbot.
   ↪rviz
   ```

Note: Getting the following output in terminal is OK: `Warning: Invalid frame ID "odom" passed to canTransform argument target_frame - frame does not exist`. This happens because `joint_state_publisher_gui` node need some time to start.

The `joint_state_publisher_gui` provides a GUI to change the configuration for *RRbot*. It is immediately displayed in *RViz*.



Once it is working you can stop rviz using CTRL+C as the next launch file is starting RViz.

2. To start *RRBot* example open a terminal, source your ROS2-workspace and execute its launch file with

   Local

   ```
   ros2 launch ros2_control_demo_example_1 rrbot.launch.py
   ```

   The launch file loads and starts the robot hardware, controllers and opens *RViz*.

   Docker

   ```
   docker run -it --rm --name ros2_control_demos --net host ros2_control_demos ros2␣
   ↪launch ros2_control_demo_example_1 rrbot.launch.py gui:=false
   ```

   The launch file loads and starts the robot hardware and controllers. Open *RViz* in a new terminal as described above.

   In starting terminal you will see a lot of output from the hardware implementation showing its internal states. This is only of exemplary purposes and should be avoided as much as possible in a hardware interface implementation.

   If you can see two orange and one yellow rectangle in *RViz* everything has started properly. Still, to be sure, let's introspect the control system before moving *RRBot*.

3. Check if the hardware interface loaded properly, by opening another terminal and executing

   Local

```
ros2 control list_hardware_interfaces
```

Docker

Open a bash terminal inside the already running docker container by

```
docker exec -it ros2_control_demos ./entrypoint.sh bash
```

and run the command

```
ros2 control list_hardware_interfaces
```

If everything started nominally, you should see the output

```
command interfaces
      joint1/position [available] [claimed]
      joint2/position [available] [claimed]
state interfaces
      joint1/position
      joint2/position
```

Marker `[claimed]` by command interfaces means that a controller has access to command *RRBot*.

4. Check if controllers are running by

   Local

   ```
   ros2 control list_controllers
   ```

   Docker

   (from the docker terminal, see above)

   ```
   ros2 control list_controllers
   ```

   You will see the two controllers in active state

   ```
   joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
   forward_position_controller[forward_command_controller/ForwardCommandController]␣
   ↪active
   ```

5. If you get output from above you can send commands to *Forward Command Controller*, either:

   a. Manually using ROS 2 CLI interface:

   Local

   ```
   ros2 topic pub /forward_position_controller/commands std_msgs/msg/
   ↪Float64MultiArray "data:
   - 0.5
   - 0.5"
   ```

   Docker

   Inside the docker terminal from above, run the command

   ```
   ros2 topic pub /forward_position_controller/commands std_msgs/msg/
   ↪Float64MultiArray "data:
   - 0.5
   - 0.5"
   ```

B. Or you can start a demo node which sends two goals every 5 seconds in a loop:

Local

```
ros2 launch ros2_control_demo_example_1 test_forward_position_controller.launch.py
```

Docker

Inside the docker terminal from above, run the command

```
ros2 launch ros2_control_demo_example_1 test_forward_position_controller.launch.py
```

You should now see orange and yellow blocks moving in *RViz*. Also, you should see changing states in the terminal where launch file is started, e.g.

```
[ros2_control_node-1] [INFO] [1721763082.437870177] [controller_manager.resource_
↪manager.hardware_component.system.RRBot]: Writing commands:
[ros2_control_node-1]    0.50 for joint 'joint2/position'
[ros2_control_node-1]    0.50 for joint 'joint1/position'
```

If you echo the `/joint_states` or `/dynamic_joint_states` topics you should now get similar values, namely the simulated states of the robot

Local

```
ros2 topic echo /joint_states
ros2 topic echo /dynamic_joint_states
```

Docker

Inside the docker terminal from above, run the command

```
ros2 topic echo /joint_states
ros2 topic echo /dynamic_joint_states
```

6. Let's switch to a different controller, the `Joint Trajectory Controller`. Load the controller manually by

Local

```
ros2 control load_controller joint_trajectory_position_controller $(ros2 pkg␣
↪prefix ros2_control_demo_example_1 --share)/config/rrbot_jtc.yaml
```

Docker

(from the docker terminal, see above)

```
ros2 control load_controller joint_trajectory_position_controller $(ros2 pkg␣
↪prefix ros2_control_demo_example_1 --share)/config/rrbot_jtc.yaml
```

what should return `Successfully loaded controller joint_trajectory_position_controller`. Check the status with

Local

```
ros2 control list_controllers
```

Docker

(from the docker terminal, see above)

```
ros2 control list_controllers
```

what shows you that the controller is loaded but unconfigured.

```
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
forward_position_controller[forward_command_controller/ForwardCommandController]␣
↪active
joint_trajectory_position_controller[joint_trajectory_controller/
↪JointTrajectoryController] unconfigured
```

Configure the controller by setting it `inactive` by

Local

```
ros2 control set_controller_state joint_trajectory_position_controller inactive
```

Docker

(from the docker terminal, see above)

```
ros2 control set_controller_state joint_trajectory_position_controller inactive
```

what should give `Successfully configured joint_trajectory_position_controller`.

As an alternative, you can load the controller directly in `inactive`-state by means of the option for `load_controller` with

Local

```
ros2 control load_controller --set-state inactive joint_trajectory_position_
↪controller $(ros2 pkg prefix ros2_control_demo_example_1 --share)/config/rrbot_
↪jtc.yaml
```

Docker

(from the docker terminal, see above)

```
ros2 control load_controller --set-state inactive joint_trajectory_position_
↪controller $(ros2 pkg prefix ros2_control_demo_example_1 --share)/config/rrbot_
↪jtc.yaml
```

You should get the result `Successfully loaded controller joint_trajectory_position_controller into state inactive`.

See if it loaded properly with

Local

```
ros2 control list_controllers
```

Docker

(from the docker terminal, see above)

```
ros2 control list_controllers
```

what should now return

```
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
forward_position_controller[forward_command_controller/ForwardCommandController]␣
↪active
joint_trajectory_position_controller[joint_trajectory_controller/
↪JointTrajectoryController] inactive
```

Note that the controller is loaded but still `inactive`. Now you can switch the controller by

Local

```
ros2 control set_controller_state forward_position_controller inactive
ros2 control set_controller_state joint_trajectory_position_controller active
```

Docker

(from the docker terminal, see above)

```
ros2 control set_controller_state forward_position_controller inactive
ros2 control set_controller_state joint_trajectory_position_controller active
```

or simply via this one-line command

Local

```
ros2 control switch_controllers --activate joint_trajectory_position_controller --
↪deactivate forward_position_controller
```

Docker

(from the docker terminal, see above)

```
ros2 control switch_controllers --activate joint_trajectory_position_controller --
↪deactivate forward_position_controller
```

Again, check via

Local

```
ros2 control list_controllers
```

Docker

(from the docker terminal, see above)

```
ros2 control list_controllers
```

what should now return

```
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
forward_position_controller[forward_command_controller/ForwardCommandController]␣
↪inactive
joint_trajectory_position_controller[joint_trajectory_controller/
↪JointTrajectoryController] active
```

Send a command to the controller using demo node, which sends four goals every 6 seconds in a loop with

Local

```
ros2 launch ros2_control_demo_example_1 test_joint_trajectory_controller.launch.py
```

Docker

(from the docker terminal, see above)

```
ros2 launch ros2_control_demo_example_1 test_joint_trajectory_controller.launch.py
```

You can adjust the goals in rrbot_joint_trajectory_publisher.

7. Alternatively, you can use the *rqt_joint_trajectory_controller* GUI to control the robot. First, load and switch to the Joint Trajectory Controller with a single command:

   Local

   ```
   ros2 control load_controller joint_trajectory_position_controller $(ros2 pkg␣
   ↪prefix ros2_control_demo_example_1 --share)/config/rrbot_jtc.yaml --set-state␣
   ↪inactive && ros2 control switch_controllers --activate joint_trajectory_
   ↪position_controller --deactivate forward_position_controller
   ```

   Docker

   (from the docker terminal, see above)

   ```
   ros2 control load_controller joint_trajectory_position_controller --set-state␣
   ↪inactive && ros2 control switch_controllers --activate joint_trajectory_
   ↪position_controller --deactivate forward_position_controller
   ```

   Then, launch the rqt_joint_trajectory_controller GUI:

   Local

   ```
   ros2 run rqt_joint_trajectory_controller rqt_joint_trajectory_controller
   ```

   Docker

   (from the docker terminal, see above)

   ```
   ros2 run rqt_joint_trajectory_controller rqt_joint_trajectory_controller
   ```

   This will open a graphical interface that allows you to:

   - Select the controller from a dropdown menu
   - Set target positions for each joint using sliders
   - Control the execution time of trajectories
   - Send the trajectory commands to the robot

   The rqt_joint_trajectory_controller provides an intuitive way to test different joint positions without having to manually construct trajectory messages.

Files used for this demos

- Launch file: rrbot.launch.py
- Controllers yaml:
  - rrbot_controllers.yaml
  - rrbot_jtc.yaml
- URDF file: rrbot.urdf.xacro
  - Description: rrbot_description.urdf.xacro
  - `ros2_control` tag: rrbot.ros2_control.xacro
- RViz configuration: rrbot.rviz
- Test nodes goals configuration:
  - rrbot_forward_position_publisher
  - rrbot_joint_trajectory_publisher
- Hardware interface plugin: rrbot.cpp

Controllers from this demo

- `Joint State Broadcaster` (ros2_controllers repository): *doc*
- `Forward Command Controller` (ros2_controllers repository): *doc*
- `Joint Trajectory Controller` (ros2_controllers repository): *doc*

4.6.2 DiffBot

DiffBot, or "Differential Mobile Robot", is a simple mobile base with differential drive. The robot is basically a box moving according to differential drive kinematics.

For *example_2*, the hardware interface plugin is implemented having only one interface.

- The communication is done using proprietary API to communicate with the robot control box.
- Data for all joints is exchanged at once.

The *DiffBot* URDF files can be found in `description/urdf` folder.

Note: The commands below are given for a local installation of this repository and its dependencies as well as for running them from a docker container. For more information on the docker usage see *Using Docker*.

This demo shows also how you can use custom controller manager name and simplify your configuration. Besided that provides also `.launch.xml` version of the launch file to directly compare it with the `.launch.py` version.

You can rename the `controller_manager` node, e.g., when want to separate the hardware management on the same machine. Check the `launch.py` to see the name set to default name and `launch.xml` to see how to set a custom name.

```
ros2 node list | grep diffbot
```

You should see the manager `/my_diffbot_manager` and the hardware node `/diffbot` (or similar, based on your URDF name).

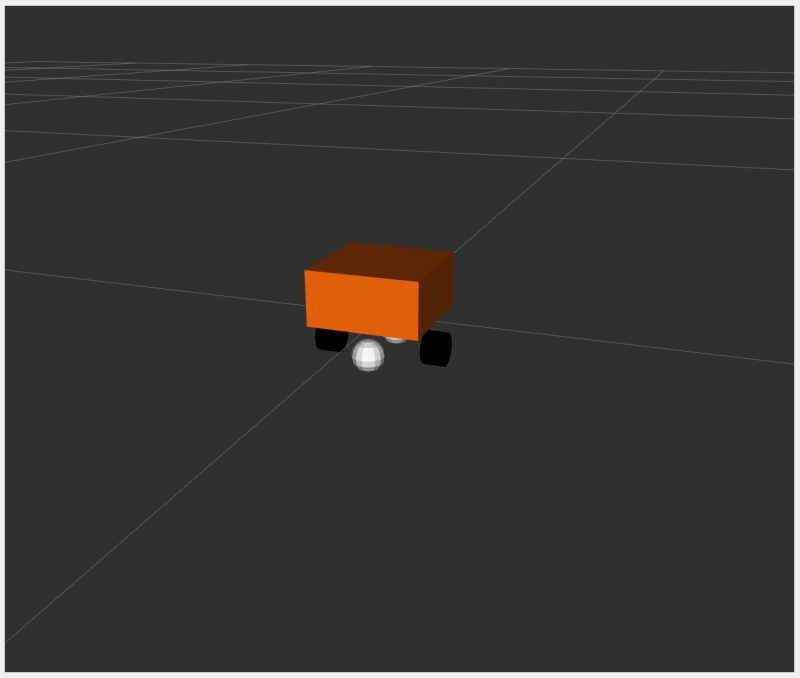
Configuration Note: The `diffbot_controllers.yaml` file demonstrates that you do not need to nest controller parameters under a `controller_manager` namespace. Since the spawner loads parameters directly for each controller, the YAML file can simply list the controllers at the root level. This makes the configuration more flexible and easier to reuse.

Tutorial steps

1. To check that *DiffBot* description is working properly use following launch commands

```
ros2 launch ros2_control_demo_example_2 view_robot.launch.py
```

> **Warning:** Getting the following output in terminal is OK: `Warning: Invalid frame ID "odom" passed to canTransform argument target_frame - frame does not exist.` This happens because `joint_state_publisher_gui` node need some time to start.



2. To start *DiffBot* example open a terminal, source your ROS2-workspace and execute its launch file with

```
ros2 launch ros2_control_demo_example_2 diffbot.launch.py
```

The launch file loads and starts the robot hardware, controllers and opens *RViz*. In the starting terminal you will see a lot of output from the hardware implementation showing its internal states. This excessive printing is only added for demonstration. In general, printing to the terminal should be avoided as much as possible in a hardware interface implementation.

If you can see an orange box in *RViz* everything has started properly. Still, to be sure, let's introspect the control system before moving *DiffBot*.

3. Check if the hardware interface loaded properly, by opening another terminal and executing

```
ros2 control list_hardware_interfaces
```

You should get

```
command interfaces
      left_wheel_joint/velocity [available] [claimed]
      right_wheel_joint/velocity [available] [claimed]
state interfaces
      left_wheel_joint/position
      left_wheel_joint/velocity
      right_wheel_joint/position
      right_wheel_joint/velocity
```

The `[claimed]` marker on command interfaces means that a controller has access to command *DiffBot*.

Furthermore, we can see that the command interface is of type `velocity`, which is typical for a differential drive robot.

4. Check if controllers are running

```
ros2 control list_controllers
```

You should get

```
diffbot_base_controller[diff_drive_controller/DiffDriveController] active
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
```

5. If everything is fine, now you can send a command to *Diff Drive Controller* using ROS 2 CLI interface:

```
ros2 topic pub --rate 10 /cmd_vel geometry_msgs/msg/TwistStamped "
  header: auto
  twist:
    linear:
      x: 0.7
      y: 0.0
      z: 0.0
    angular:
      x: 0.0
      y: 0.0
      z: 1.0"
```

You should now see an orange box circling in *RViz*. Also, you should see changing states in the terminal where launch file is started.

```
[ros2_control_node-1] [INFO] [1721762311.808415917] [controller_manager.resource_
↪manager.hardware_component.system.DiffBot]: Writing commands:
[ros2_control_node-1]    command 43.33 for 'left_wheel_joint'!
[ros2_control_node-1]    command 50.00 for 'right_wheel_joint'!
```

6. Let's introspect the ros2_control hardware component. Calling

```
ros2 control list_hardware_components
```

should give you

```
Hardware Component 1
        name: DiffBot
        type: system
        plugin name: ros2_control_demo_example_2/DiffBotSystemHardware
        state: id=3 label=active
```

(continues on next page)

(continued from previous page)

```
        command interfaces
                left_wheel_joint/velocity [available] [claimed]
                right_wheel_joint/velocity [available] [claimed]
```

This shows that the custom hardware interface plugin is loaded and running. If you work on a real robot and don't have a simulator running, it is often faster to use the `mock_components/GenericSystem` hardware component instead of writing a custom one. Stop the launch file and start it again with an additional parameter

```
ros2 launch ros2_control_demo_example_2 diffbot.launch.py use_mock_
↪hardware:=True
```

Calling

```
ros2 control list_hardware_components
```

now should give you

```
Hardware Component 1
    name: DiffBot
    type: system
    plugin name: mock_components/GenericSystem
    state: id=3 label=active
    command interfaces
            left_wheel_joint/velocity [available] [claimed]
            right_wheel_joint/velocity [available] [claimed]
```

You see that a different plugin was loaded. Having a look into the diffbot.ros2_control.xacro, one can find the instructions to load this plugin together with the parameter `calculate_dynamics`.

```
<hardware>
  <plugin>mock_components/GenericSystem</plugin>
  <param name="calculate_dynamics">true</param>
</hardware>
```

This enables the integration of the velocity commands to the position state interface, which can be checked by means of `ros2 topic echo /joint_states`: The position values are increasing over time if the robot is moving. You now can test the setup with the commands from above, it should work identically as the custom hardware component plugin.

Another parameter of `mock_components/GenericSystem` is `disable_commands`. When set to `true`, the hardware plugin stops mirroring commanded values to the state interfaces. This simulates a disconnected driver scenario — commands are sent to the hardware successfully, but no feedback signal is received back, as would happen with a broken encoder cable or a dropped network connection.

Stop the launch file and restart with an additional parameter:

```
ros2 launch ros2_control_demo_example_2 diffbot.launch.py use_mock_
↪hardware:=True disable_commands:=True
```

Send velocity commands as before. Then check joint states:

```
ros2 topic echo /joint_states
```

You will notice that the position and velocity values **do not change** over time, even though commands are actively being published. This is confirmed by the warning in the controller manager terminal:

```
[ros2_control_node-1] [WARN] [...] [controller_manager.hardware_component.
↪system.DiffBot]: Command propagation is disabled - no values will be␣
↪returned!
```

The relevant configuration in diffbot.ros2_control.xacro is:

```
<hardware>
  <plugin>mock_components/GenericSystem</plugin>
  <param name="calculate_dynamics">true</param>
  <param name="disable_commands">true</param>
</hardware>
```

More information on mock_components can be found in the *ros2_control documentation*.

Files used for this demos

- Launch file: diffbot.launch.py
- Controllers yaml: diffbot_controllers.yaml
- URDF file: diffbot.urdf.xacro
  - Description: diffbot_description.urdf.xacro
  - `ros2_control` tag: diffbot.ros2_control.xacro
- RViz configuration: diffbot.rviz
- Hardware interface plugin: diffbot_system.cpp

Controllers from this demo

- `Joint State Broadcaster` (ros2_controllers repository): *doc*
- `Diff Drive Controller` (ros2_controllers repository): *doc*

4.6.3 Example 3: Robots with multiple interfaces

The example shows how to implement multi-interface robot hardware taking care about interfaces used.

For *example_3*, the hardware interface plugin is implemented having multiple interfaces.

- The communication is done using proprietary API to communicate with the robot control box.
- Data for all joints is exchanged at once.
- Examples: KUKA FRI, ABB Yumi, Schunk LWA4p, etc.

Two illegal controllers demonstrate how hardware interface declines faulty claims to access joint command interfaces.

Note: The commands below are given for a local installation of this repository and its dependencies as well as for running them from a docker container. For more information on the docker usage see *Using Docker*.

Tutorial steps

1. To check that *RRBot* descriptions are working properly use following launch commands

```
ros2 launch ros2_control_demo_example_3 view_robot.launch.py
```

Note: Getting the following output in terminal is OK: `Warning: Invalid frame ID "odom" passed to canTransform argument target_frame - frame does not exist`. This happens because `joint_state_publisher_gui` node need some time to start. The `joint_state_publisher_gui` provides a GUI to generate a random configuration for rrbot. It is immediately displayed in *RViz*.

---

2. To start *RRBot* example open a terminal, source your ROS2-workspace and execute its launch file with

```
ros2 launch ros2_control_demo_example_3 rrbot_system_multi_interface.launch.py
```

Useful launch-file options:

`robot_controller:=forward_position_controller`
starts demo and spawns position controller. Robot can be then controlled using `forward_position_controller` as described below.

`robot_controller:=forward_acceleration_controller`
starts demo and spawns acceleration controller. Robot can be then controlled using `forward_acceleration_controller` as described below.

The launch file loads and starts the robot hardware, controllers and opens *RViz*. In starting terminal you will see a lot of output from the hardware implementation showing its internal states. This is only of exemplary purposes and should be avoided as much as possible in a hardware interface implementation.

If you can see two orange and one yellow rectangle in in *RViz* everything has started properly. Still, to be sure, let's introspect the control system before moving *RRBot*.

3. Check if the hardware interface loaded properly, by opening another terminal and executing

```
ros2 control list_hardware_interfaces
```

```
command interfaces
    joint1/acceleration [available] [unclaimed]
    joint1/position [available] [unclaimed]
    joint1/velocity [available] [claimed]
    joint2/acceleration [available] [unclaimed]
    joint2/position [available] [unclaimed]
    joint2/velocity [available] [claimed]
state interfaces
    joint1/acceleration
    joint1/position
    joint1/velocity
    joint2/acceleration
    joint2/position
    joint2/velocity
```

Marker `[claimed]` by command interfaces means that a controller has access to command *RRBot*.

4. Check which controllers are running

```
ros2 control list_controllers
```

gives

```
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
forward_velocity_controller[forward_command_controller/ForwardCommandController]␣
↪active
```

Check how this output changes if you use the different launch file arguments described above.

5. If you get output from above you can send commands to *Forward Command Controller*, either:
   1. Manually using ROS 2 CLI interface.
      - when using `forward_position_controller` controller

      ```
      ros2 topic pub /forward_position_controller/commands std_msgs/msg/
      ↪Float64MultiArray "data:
      - 0.5
      - 0.5"
      ```

      - when using `forward_velocity_controller` controller (default)

      ```
      ros2 topic pub /forward_velocity_controller/commands std_msgs/msg/
      ↪Float64MultiArray "data:
      - 5
      - 5"
      ```

      - when using `forward_acceleration_controller` controller

      ```
      ros2 topic pub /forward_acceleration_controller/commands std_msgs/msg/
      ↪Float64MultiArray "data:
      - 10
      - 10"
      ```

   2. Or you can start a demo node which sends two goals every 5 seconds in a loop when using `forward_position_controller` controller

      ```
      ros2 launch ros2_control_demo_example_3 test_forward_position_controller.
      ↪launch.py
      ```

   You should now see orange and yellow blocks moving in *RViz*. Also, you should see changing states in the terminal where launch file is started, e.g.

   ```
   [ros2_control_node-1] [INFO] [1728857332.160329225] [controller_manager.resource_
   ↪manager.hardware_component.system.RRBotSystemMultiInterface]: Writing commands:
   [ros2_control_node-1]    command pos: 0.00, vel: 5.00, acc: 0.00 for joint 0,␣
   ↪control lvl: 2
   [ros2_control_node-1]    command pos: 0.00, vel: 5.00, acc: 0.00 for joint 1,␣
   ↪control lvl: 2
   [ros2_control_node-1] [INFO] [1728857332.320242591] [controller_manager.resource_
   ↪manager.hardware_component.system.RRBotSystemMultiInterface]: Reading states:
   [ros2_control_node-1]    pos: 0.67, vel: 5.00, acc: 0.00 for joint 0
   [ros2_control_node-1]    pos: 0.67, vel: 5.00, acc: 0.00 for joint 1
   ```

6. Now you can also switch controllers during runtime, which also changes the command mode automatically. First, you have to load the new controller, for example the `forward_position_controller` if you haven't changed the launch file argument.

```
ros2 control load_controller forward_position_controller $(ros2 pkg prefix␣
↪ros2_control_demo_example_3 --share)/config/rrbot_multi_interface_forward_
↪controllers.yaml
ros2 control set_controller_state forward_position_controller inactive
```

Then you can switch controllers using the following command:

```
ros2 control switch_controllers --deactivate forward_velocity_controller --
↪activate forward_position_controller
```

Observe the output of the following CLI commands, and see how the command interfaces are claimed by the new controller.

```
ros2 control list_controllers
ros2 control list_hardware_interfaces
```

Try now to send commands to the new controller, as described in the previous step.

7. To demonstrate illegal controller configuration, use one of the following launch file arguments:
   - `robot_controller:=forward_illegal1_controller` or
   - `robot_controller:=forward_illegal2_controller`

   You will see the following error messages, because the hardware interface enforces all joints having the same command interface

```
[ros2_control_node-1] [ERROR] [1676209982.531163501] [resource_manager]:␣
↪Component 'RRBotSystemMultiInterface' did not accept new command resource␣
↪combination:
[ros2_control_node-1]  Start interfaces:
[ros2_control_node-1] [
[ros2_control_node-1]   joint1/position
[ros2_control_node-1] ]
[ros2_control_node-1] Stop interfaces:
[ros2_control_node-1] [
[ros2_control_node-1] ]
[ros2_control_node-1]
[ros2_control_node-1] [ERROR] [1676209982.531223835] [controller_manager]: Could␣
↪not switch controllers since prepare command mode switch was rejected.
[spawner-4] [ERROR] [1676209982.531717376] [spawner_forward_illegal1_controller]:␣
↪Failed to activate controller
```

Running `ros2 control list_hardware_interfaces` shows that no interface is claimed

```
command interfaces
      joint1/acceleration [available] [unclaimed]
      joint1/position [available] [unclaimed]
      joint1/velocity [available] [unclaimed]
      joint2/acceleration [available] [unclaimed]
      joint2/position [available] [unclaimed]
      joint2/velocity [available] [unclaimed]
state interfaces
      joint1/acceleration
      joint1/position
      joint1/velocity
      joint2/acceleration
      joint2/position
      joint2/velocity
```

and `ros2 control list_controllers` indicates that the illegal controller was not loaded

```
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
forward_illegal1_controller[forward_command_controller/ForwardCommandController]␣
↪inactive
```

Files used for this demos

- Launch file: rrbot_system_multi_interface.launch.py
- Controllers yaml: rrbot_multi_interface_forward_controllers.yaml
- URDF: rrbot_system_multi_interface.urdf.xacro
  - Description: rrbot_description.urdf.xacro
  - `ros2_control` URDF tag: rrbot_system_multi_interface.ros2_control.xacro
- RViz configuration: rrbot.rviz
- Hardware interface plugin: rrbot_system_multi_interface.cpp

Controllers from this demo

- `Joint State Broadcaster` (ros2_controllers repository): *doc*
- `Forward Command Controller` (ros2_controllers repository): *doc*

4.6.4 Example 4: Industrial robot with integrated sensor

This example shows how a sensor can be integrated in a hardware interface:

- The communication is done using proprietary API to communicate with the robot control box.
- Data for all joints is exchanged at once.
- Sensor data are exchanged together with joint data
- Examples: KUKA RSI with sensor connected to KRC (KUKA control box) or a prototype robot (ODRI interface).

A 2D Force-Torque Sensor (FTS) is simulated by generating random sensor readings via a hardware interface of type `hardware_interface::SystemInterface`.

Note: The commands below are given for a local installation of this repository and its dependencies as well as for running them from a docker container. For more information on the docker usage see *Using Docker*.

Tutorial steps

1. To check that *RRBot* descriptions are working properly use following launch commands

```
ros2 launch ros2_control_demo_example_4 view_robot.launch.py
```

Note: Getting the following output in terminal is OK: `Warning: Invalid frame ID "odom" passed to canTransform argument target_frame - frame does not exist`. This happens because `joint_state_publisher_gui` node need some time to start. The `joint_state_publisher_gui` provides a GUI to generate a random configuration for rrbot. It is immediately displayed in *RViz*.

2. To start *RRBot* example open a terminal, source your ROS2-workspace and execute its launch file with

```
ros2 launch ros2_control_demo_example_4 rrbot_system_with_sensor.launch.py
```

The launch file loads and starts the robot hardware, controllers and opens *RViz*. In starting terminal you will see a lot of output from the hardware implementation showing its internal states. This is only of exemplary purposes and should be avoided as much as possible in a hardware interface implementation.

If you can see two orange and one yellow rectangle in in *RViz* everything has started properly. Still, to be sure, let's introspect the control system before moving *RRBot*.

3. Check if the hardware interface loaded properly, by opening another terminal and executing

```
ros2 control list_hardware_interfaces
```

```
command interfaces
        joint1/position [available] [claimed]
        joint2/position [available] [claimed]
state interfaces
        joint1/position
        joint2/position
        tcp_fts_sensor/force.x
        tcp_fts_sensor/torque.z
```

Marker `[claimed]` by command interfaces means that a controller has access to command *RRBot*.

Now, let's introspect the hardware components with

```
ros2 control list_hardware_components -v
```

There is a single hardware component for the robot providing the command and state interfaces:

```
Hardware Component 1
    name: RRBotSystemWithSensor
    type: system
    plugin name: ros2_control_demo_example_4/RRBotSystemWithSensorHardware
    state: id=3 label=active
    command interfaces
            joint1/position [available] [claimed]
            joint2/position [available] [claimed]
    state interfaces
            joint1/position [available]
            joint2/position [available]
            tcp_fts_sensor/force.x [available]
            tcp_fts_sensor/torque.z [available]
```

4. Check if controllers are running

```
ros2 control list_controllers
```

```
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
fts_broadcaster     [force_torque_sensor_broadcaster/
↪ForceTorqueSensorBroadcaster] active
forward_position_controller[forward_command_controller/ForwardCommandController]␣
↪active
```

5. If you get output from above you can send commands to *Forward Command Controller*, either:

   1. Manually using ROS 2 CLI interface.

```
ros2 topic pub /forward_position_controller/commands std_msgs/msg/
↪Float64MultiArray "data:
- 0.5
- 0.5"
```

   2. Or you can start a demo node which sends two goals every 5 seconds in a loop

```
ros2 launch ros2_control_demo_example_4 test_forward_position_controller.
↪launch.py
```

   You should now see orange and yellow blocks moving in *RViz*. Also, you should see changing states in the terminal where launch file is started, e.g.

```
[ros2_control_node-1] [INFO] [1721763738.761847562] [controller_manager.resource_
↪manager.hardware_component.system.RRBotSystemWithSensor]: Writing commands:
[ros2_control_node-1]    0.50 for joint 'joint1'
[ros2_control_node-1]    0.50 for joint 'joint2'
```

6. Access wrench data from 2D FTS via

```
ros2 topic echo /fts_broadcaster/wrench
```

   shows the random generated sensor values, republished by *Force Torque Sensor Broadcaster* as `geometry_msgs/msg/WrenchStamped` message

```
header:
  stamp:
    sec: 1676444704
    nanosec: 332221422
  frame_id: tool_link
wrench:
  force:
    x: 2.946532964706421
    y: .nan
    z: .nan
  torque:
    x: .nan
    y: .nan
    z: 4.0540995597839355
```

Warning: Wrench messages are not displayed properly in *RViz* as NaN values are not handled in *RViz* and FTS Broadcaster may send NaN values.

Files used for this demo

- Launch file: rrbot_system_with_sensor.launch.py
- Controllers yaml: rrbot_with_sensor_controllers.yaml
- URDF: rrbot_system_with_sensor.urdf.xacro
  - Description: rrbot_description.urdf.xacro
  - `ros2_control` URDF tag: rrbot_system_with_sensor.ros2_control.xacro
- RViz configuration: rrbot.rviz
- Hardware interface plugin: rrbot_system_with_sensor.cpp

Controllers from this demo

- `Joint State Broadcaster` (ros2_controllers repository): *doc*
- `Forward Command Controller` (ros2_controllers repository): *doc*
- `Force Torque Sensor Broadcaster` (ros2_controllers repository): *doc*

4.6.5 Example 5: Industrial robot with externally connected sensor

This example shows how an externally connected sensor can be accessed:

- The communication is done using proprietary API to communicate with the robot control box.
- Data for all joints is exchanged at once.
- Sensor data are exchanged independently of joint data.
- Examples: KUKA RSI and FTS connected to independent PC with ROS 2.

A 3D Force-Torque Sensor (FTS) is simulated by generating random sensor readings via a hardware interface of type `hardware_interface::SensorInterface`.

Note: The commands below are given for a local installation of this repository and its dependencies as well as for running them from a docker container. For more information on the docker usage see *Using Docker*.

Tutorial steps

1. To check that *RRBot* descriptions are working properly use following launch commands

   ```
   ros2 launch ros2_control_demo_example_5 view_robot.launch.py
   ```

   Note: Getting the following output in terminal is OK: `Warning: Invalid frame ID "odom" passed to canTransform argument target_frame - frame does not exist`. This happens because `joint_state_publisher_gui` node need some time to start. The `joint_state_publisher_gui` provides a GUI to generate a random configuration for rrbot. It is immediately displayed in *RViz*.

2. To start *RRBot* example open a terminal, source your ROS2-workspace and execute its launch file with

```
ros2 launch ros2_control_demo_example_5 rrbot_system_with_external_sensor.launch.
↪py
```

The launch file loads and starts the robot hardware, controllers and opens *RViz*. In starting terminal you will see a lot of output from the hardware implementation showing its internal states. This is only of exemplary purposes and should be avoided as much as possible in a hardware interface implementation.

If you can see two orange and one yellow rectangle in in *RViz* everything has started properly. Still, to be sure, let's introspect the control system before moving *RRBot*.

Note: The launch file supports an optional `use_wrench_transformer` argument to enable the wrench transformer node. See step 7 for details on using the wrench transformer feature.

---

3. Check if the hardware interface loaded properly, by opening another terminal and executing

```
ros2 control list_hardware_interfaces
```

```
command interfaces
  joint1/position [available] [claimed]
  joint2/position [available] [claimed]
state interfaces
  joint1/position
  joint2/position
  tcp_fts_sensor/force.x
  tcp_fts_sensor/force.y
  tcp_fts_sensor/force.z
  tcp_fts_sensor/torque.x
  tcp_fts_sensor/torque.y
```

Marker `[claimed]` by command interfaces means that a controller has access to command *RRBot*.

Now, let's introspect the hardware components with

```
ros2 control list_hardware_components
```

There are two hardware components, one for the robot and one for the sensor:

```
Hardware Component 1
        name: ExternalRRBotFTSensor
        type: sensor
        plugin name: ros2_control_demo_example_5/
↪ExternalRRBotForceTorqueSensorHardware
        state: id=3 label=active
        command interfaces
Hardware Component 2
        name: RRBotSystemPositionOnly
        type: system
        plugin name: ros2_control_demo_example_5/RRBotSystemPositionOnlyHardware
        state: id=3 label=active
        command interfaces
                joint1/position [available] [claimed]
                joint2/position [available] [claimed]
```

4. Check if controllers are running

```
ros2 control list_controllers
```

```
forward_position_controller[forward_command_controller/ForwardCommandController]␣
↪active
fts_broadcaster[force_torque_sensor_broadcaster/ForceTorqueSensorBroadcaster]␣
↪active
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
```

5. If you get output from above you can send commands to *Forward Command Controller*, either:

   1. Manually using ROS 2 CLI interface.

      ```
      ros2 topic pub /forward_position_controller/commands std_msgs/msg/
      ↪Float64MultiArray "data:
      - 0.5
      - 0.5"
      ```

   2. Or you can start a demo node which sends two goals every 5 seconds in a loop

      ```
      ros2 launch ros2_control_demo_example_5 test_forward_position_controller.
      ↪launch.py
      ```

   You should now see orange and yellow blocks moving in *RViz*. Also, you should see changing states in the terminal where launch file is started, e.g.

```
[ros2_control_node-1] [INFO] [1721764191.201301188] [controller_manager.resource_
↪manager.hardware_component.system.RRBotSystemPositionOnly]: Writing commands:
[ros2_control_node-1]    0.50 for joint 'joint1'
[ros2_control_node-1]    0.50 for joint 'joint2'
```

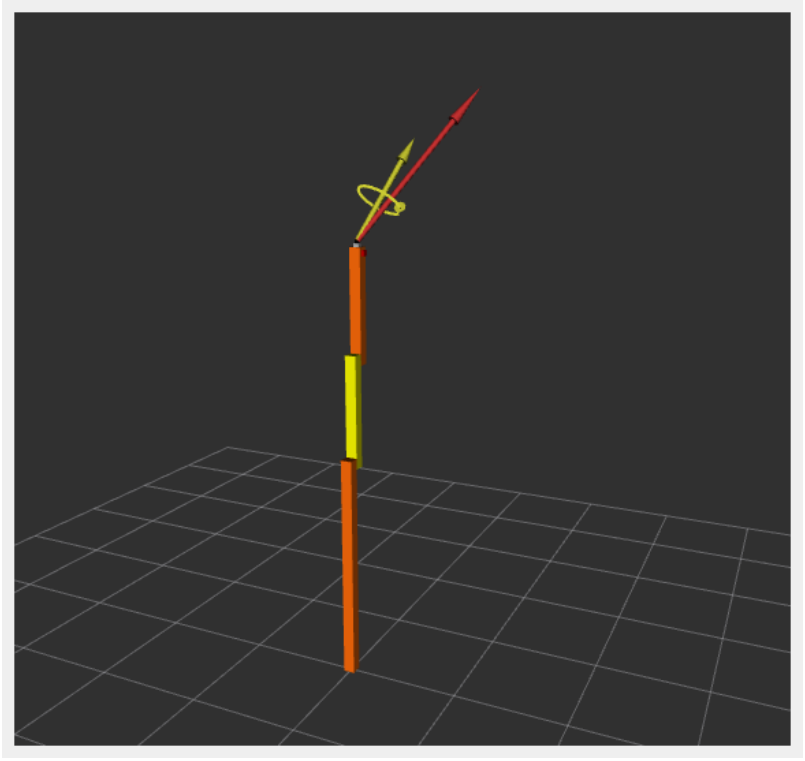
6. Access wrench data from 2D FTS via

```
ros2 topic echo /fts_broadcaster/wrench
```

   shows the random generated sensor values, republished by *Force Torque Sensor Broadcaster* as `geometry_msgs/msg/WrenchStamped` message

```
header:
  stamp:
    sec: 1676444704
    nanosec: 332221422
  frame_id: tool_link
wrench:
  force:
    x: 1.2126582860946655
    y: 2.3202226161956787
    z: 3.4302282333374023
  torque:
    x: 4.540233612060547
    y: 0.647800624370575
    z: 1.7602499723434448
```

   Wrench data are also visualized in *RViz*:



7. Access transformed wrench data in different frames using the *Wrench Transformer Node* (optional):

   The launch file can optionally start a `wrench_transformer_node` that transforms wrench messages from the sensor frame (`tool_link`) to other target frames using TF2. This feature is disabled by default.

   To enable the wrench transformer, launch with the `use_wrench_transformer` argument:

```
ros2 launch ros2_control_demo_example_5 rrbot_system_with_external_sensor.launch.
↪py use_wrench_transformer:=true
```

   Once enabled, check available transformed wrench topics:

```
ros2 topic list | grep wrench
```

   You should see topics like:

```
/fts_wrench_transformer/base_link/wrench
/fts_wrench_transformer/link1/wrench
```

   View transformed wrench data in the `base_link` frame:

```
ros2 topic echo /fts_wrench_transformer/base_link/wrench
```

   The transformed wrench messages will have the same structure as the original wrench, but with `frame_id` set to the target frame (e.g., `base_link`) and the force/torque values transformed to that coordinate frame.

```
header:
  stamp:
    sec: 1676444704
    nanosec: 332221422
  frame_id: base_link
wrench:
```

(continues on next page)

(continued from previous page)

```
  force:
    x: <transformed_value>
    y: <transformed_value>
    z: <transformed_value>
  torque:
    x: <transformed_value>
    y: <transformed_value>
    z: <transformed_value>
```

The wrench transformer configuration can be customized in the parameter file: wrench_transformer_params.yaml

Files used for this demos

- Launch file: rrbot_system_with_external_sensor.launch.py
- Controllers yaml: rrbot_with_external_sensor_controllers.yaml
- Wrench transformer params: wrench_transformer_params.yaml
- URDF: rrbot_with_external_sensor_controllers.urdf.xacro
  - Description: rrbot_description.urdf.xacro
  - `ros2_control` robot: rrbot_system_position_only.ros2_control.xacro
  - `ros2_control` sensor: external_rrbot_force_torque_sensor.ros2_control.xacro
- RViz configuration: rrbot.rviz
- Hardware interface plugin:
  - robot rrbot.cpp
  - sensor external_rrbot_force_torque_sensor.cpp

Controllers from this demo

- `Joint State Broadcaster` (ros2_controllers repository): *doc*
- `Forward Command Controller` (ros2_controllers repository): *doc*
- `Force Torque Sensor Broadcaster` (ros2_controllers repository): *doc*

4.6.6 Example 6: Modular Robots with separate communication to each actuator

The example shows how to implement robot hardware with separate communication to each actuator:

- The communication is done on actuator level using proprietary or standardized API (e.g., canopen_402, Modbus, RS232, RS485).
- Data for all actuators is exchanged separately from each other.
- Examples: Mara, Arduino-based-robots

This is implemented with a hardware interface of type `hardware_interface::ActuatorInterface`.

Note: The commands below are given for a local installation of this repository and its dependencies as well as for running them from a docker container. For more information on the docker usage see *Using Docker*.

Tutorial steps

1. To check that *RRBot* descriptions are working properly use following launch commands

```
ros2 launch ros2_control_demo_example_6 view_robot.launch.py
```

Note: Getting the following output in terminal is OK: `Warning: Invalid frame ID "odom" passed to canTransform argument target_frame - frame does not exist`. This happens because `joint_state_publisher_gui` node need some time to start. The `joint_state_publisher_gui` provides a GUI to generate a random configuration for rrbot. It is immediately displayed in *RViz*.

---

2. To start *RRBot* example open a terminal, source your ROS2-workspace and execute its launch file with

```
ros2 launch ros2_control_demo_example_6 rrbot_modular_actuators.launch.py
```

The launch file loads and starts the robot hardware, controllers and opens *RViz*. In starting terminal you will see a lot of output from the hardware implementation showing its internal states. This is only of exemplary purposes and should be avoided as much as possible in a hardware interface implementation.

If you can see two orange and one yellow rectangle in in *RViz* everything has started properly. Still, to be sure, let's introspect the control system before moving *RRBot*.

3. Check if the hardware interface loaded properly, by opening another terminal and executing

```
ros2 control list_hardware_interfaces
```

```
command interfaces
  joint1/position [available] [claimed]
  joint2/position [available] [claimed]
state interfaces
  joint1/position
  joint2/position
```

Marker `[claimed]` by command interfaces means that a controller has access to command *RRBot*.

Now, let's introspect the hardware components with

```
ros2 control list_hardware_components
```

There are two hardware components, one for each actuator and one for each sensor:

```
Hardware Component 1
        name: RRBotModularJoint2
        type: actuator
        plugin name: ros2_control_demo_example_6/RRBotModularJoint
        state: id=3 label=active
        command interfaces
                joint2/position [available] [claimed]
Hardware Component 2
        name: RRBotModularJoint1
        type: actuator
        plugin name: ros2_control_demo_example_6/RRBotModularJoint
        state: id=3 label=active
        command interfaces
                joint1/position [available] [claimed]
```

4. Check if controllers are running

```
ros2 control list_controllers
```

```
forward_position_controller[forward_command_controller/ForwardCommandController]␣
↪active
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
```

5. If you get output from above you can send commands to *Forward Command Controller*, either:
   1. Manually using ROS 2 CLI interface.

      ```
      ros2 topic pub /forward_position_controller/commands std_msgs/msg/
      ↪Float64MultiArray "data:
      - 0.5
      - 0.5"
      ```

   2. Or you can start a demo node which sends two goals every 5 seconds in a loop

      ```
      ros2 launch ros2_control_demo_example_6 test_forward_position_controller.
      ↪launch.py
      ```

   You should now see orange and yellow blocks moving in *RViz*. Also, you should see changing states in the terminal where launch file is started, e.g.

```
[ros2_control_node-1] [INFO] [1721764663.304187517] [controller_manager.resource_
↪manager.hardware_component.actuator.RRBotModularJoint1]: Writing commands:
[ros2_control_node-1]    0.50 for joint 'joint1'
[ros2_control_node-1] [INFO] [1721764663.304196897] [controller_manager.resource_
↪manager.hardware_component.actuator.RRBotModularJoint2]: Writing commands:
[ros2_control_node-1]    0.50 for joint 'joint2'
```

Files used for this demos

- Launch file: rrbot_modular_actuators.launch.py
- Controllers yaml: rrbot_modular_actuators.yaml
- URDF: rrbot_modular_actuators.urdf.xacro
  - Description: rrbot_description.urdf.xacro
  - `ros2_control` URDF tag: rrbot_modular_actuators.ros2_control.xacro
- RViz configuration: rrbot.rviz
- Hardware interface plugin: rrbot_actuator.cpp

Controllers from this demo

- `Joint State Broadcaster` (ros2_controllers repository): *doc*
- `Forward Command Controller` (ros2_controllers repository): *doc*

4.6.7 Example 7: Full tutorial with a 6DOF robot

ros2_control is a realtime control framework designed for general robotics applications. Standard c++ interfaces exist for interacting with hardware and querying user defined controller commands. These interfaces enhance code modularity and robot agnostic design. Application specific details, e.g. what controller to use, how many joints a robot has and their kinematic structure, are specified via YAML parameter configuration files and a Universal Robot Description File (URDF). Finally, the ros2_control framework is deployed via ROS 2 launch a file.

This tutorial will address each component of ros2_control in detail, namely:

1. ros2_control overview
2. Writing a URDF
3. Writing a hardware interface
4. Writing a controller

ros2_control overview

ros2_control introduces `state_interfaces` and `command_interfaces` to abstract hardware interfacing. The `state_interfaces` are read only data handles that generally represent sensors readings, e.g. joint encoder. The `command_interfaces` are read and write data handles that hardware commands, like setting a joint velocity reference. The `command_interfaces` are exclusively accessed, meaning if a controller has "claimed" an interface, it cannot be used by any other controller until it is released. Both interface types are uniquely designated with a name and type. The names and types for all available state and command interfaces are specified in a YAML configuration file and a URDF file.

ros2_control provides the `ControllerInterface` and `HardwareInterface` classes for robot agnostic control. During initialization, controllers request `state_interfaces` and `command_interfaces` required for operation through the `ControllerInterface`. On the other hand, hardware drivers offer `state_interfaces` and `command_interfaces` via the `HardwareInterface`. ros2_control ensure all requested interfaces are available before starting the controllers. The interface pattern allows vendors to write hardware specific drivers that are loaded at runtime.

The main program is a realtime read, update, write loop. During the read call, hardware drivers that conform to `HardwareInterface` update their offered `state_interfaces` with the newest values received from the hardware. During the update call, controllers calculate commands from the updated `state_interfaces` and writes them into its `command_interfaces`. Finally, during to write call, the hardware drivers read values from the `command_interfaces` and sends them to the hardware. The `ros2_control_node` runs the main loop in a realtime thread. The `ros2_control_node` runs a second non-realtime thread to interact with ROS publishers, subscribers, and services.

Writing a URDF

The URDF file is a standard XML based file used to describe characteristic of a robot. It can represent any robot with a tree structure, except those with cycles. Each link must have only one parent. For ros2_control, there are three primary tags: `link`, `joint`, and `ros2_control`. The `joint` tag define the robot's kinematic structure, while the `link` tag defines the dynamic properties and 3D geometry. The `ros2_control` defines the hardware and controller configuration.

Geometry

Most commercial robots already have `robot_description` packages defined, see the Universal Robots for an example. However, this tutorial will go through the details of creating one from scratch.

First, we need a 3D model of our robot. For illustration, a generic 6 DOF robot manipulator will be used.

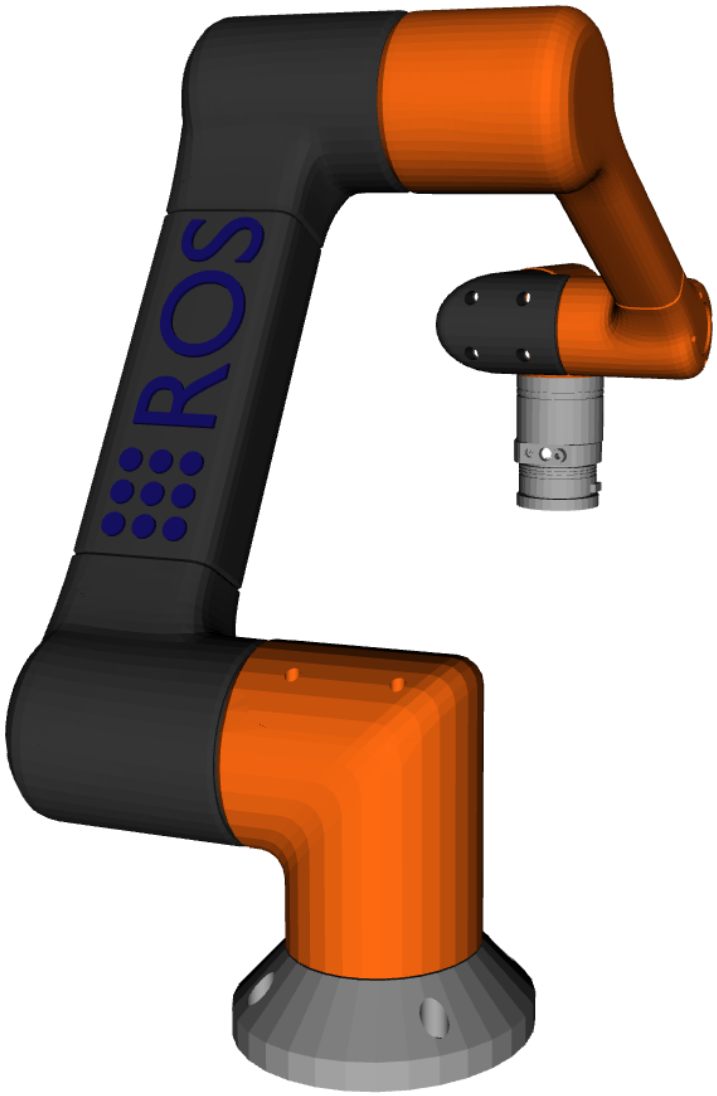


Fig. 1: a generic 6 DOF robot manipulator

The robot's 6 links each need to be processed and exported to their own `.stl` and `.dae` files. Generally, the `.stl` 3D model files are coarse meshes used for fast collision checking, while the `.dae` files are used for visualization purposed only. We will use the same mesh in our case for simplicity.

By convention, each `.stl` file expresses the position its vertices in its own reference frame. Hence, we need to specify the linear transformation (rotation and translation) between each link to define the robot's full geometry. The 3D model for each link should be adjusted such that the proximal joint axis (the axis that connects the link to its parent) is in the z-axis direction. The 3D model's origin should also be adjusted such that the bottom face of the mesh is co-planer with the xy-plane. The following mesh illustrates this configuration.

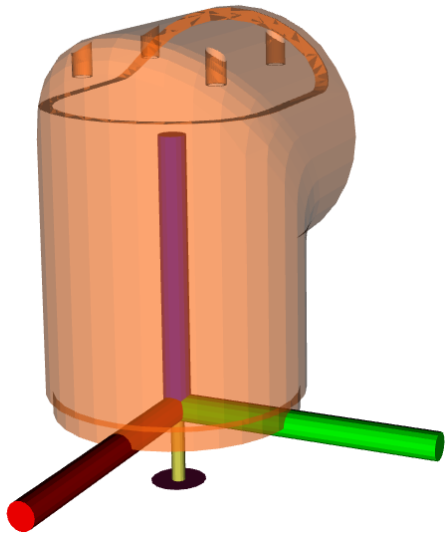


Fig. 2: Link 1

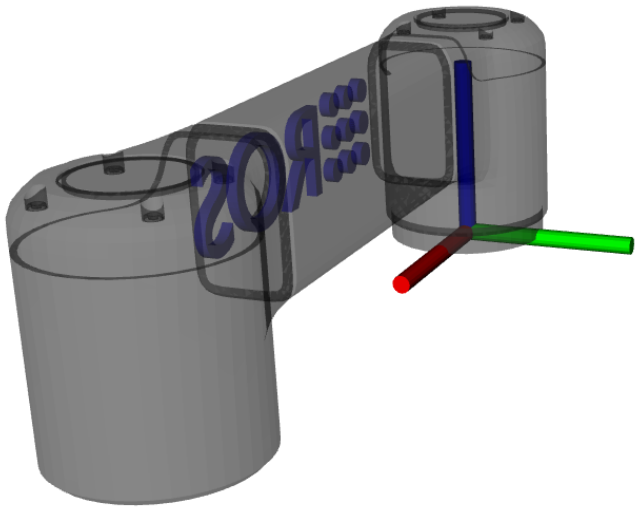


Fig. 3: Link 2 aligned

Each mesh should be exported to its own file after processing them. Blender is an open source 3D modeling software, which can import/export `.stl` and `.dae` files and manipulate their vertices. Blender was used to process the robot model in this tutorial.

We can finally calculate the transforms between the robot's joints and begin writing the URDF. First, apply a negative 90 degree roll to link 2 in its frame.

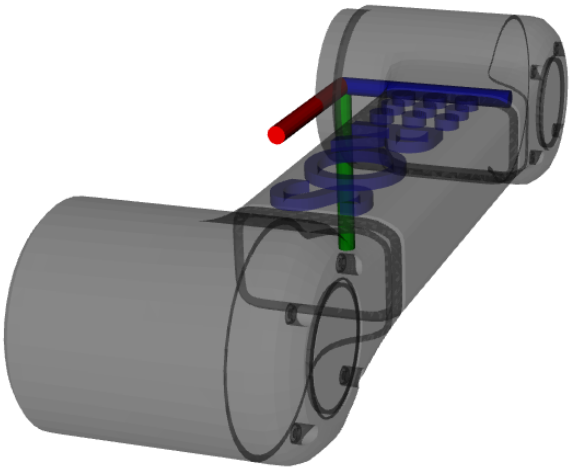


Fig. 4: Link 2 with -90 degree roll

To keep the example simple, we will not apply a pitch now. Then, we apply a positive 90 degree yaw.

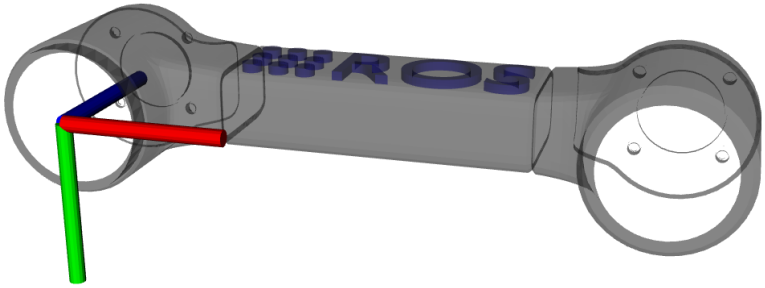


Fig. 5: Link 2 with -90 degree roll and 90 degree yaw

Finally, we apply a translation of -0.1 meters in the x-axis and 0.18 meters in the z-axis between the link 2 and link 1 frame. The final result is shown below.

The described process is then repeated for all links.

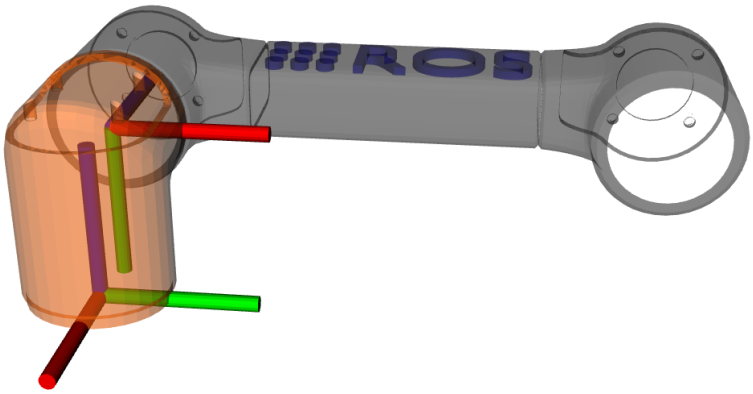


Fig. 6: Link 2 with -90 degree roll, 90 degree yaw, and translation

URDF file

The URDF file is generally formatted according to the following template.

```
<robot name="robot_6_dof">
  <!-- create link fixed to the "world" -->
  <link name="base_link">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://robot_6_dof/meshes/visual/link_0.dae"/>
      </geometry>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://robot_6_dof/meshes/collision/link_0.stl"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="1"/>
      <inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
    </inertial>
  </link>
  <!-- additional links ... -->
  <link name="world"/>
  <link name="tool0"/>
  <joint name="base_joint" type="fixed">
    <parent link="world"/>
    <child link="base_link"/>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <axis xyz="0 0 1"/>
  </joint>
  <!-- joints - main serial chain -->
  <joint name="joint_1" type="revolute">
    <parent link="base_link"/>
    <child link="link_1"/>
    <origin rpy="0 0 0" xyz="0 0 0.061584"/>
    <axis xyz="0 0 1"/>
    <limit effort="1000.0" lower="-3.141592653589793" upper="3.141592653589793"␣
```

(continues on next page)

(continued from previous page)

```
↪velocity="2.5"/>
  </joint>
  <!-- additional joints ... -->
  <!-- ros2 control tag -->
  <ros2_control name="robot_6_dof" type="system">
    <hardware>
      <plugin>
        <!-- {Name_Space}/{Class_Name}-->
      </plugin>
    </hardware>
    <joint name="joint_1">
      <command_interface name="position">
        <param name="min">{-2*pi}</param>
        <param name="max">{2*pi}</param>
      </command_interface>
      <!-- additional command interfaces ... -->
      <state_interface name="position">
        <param name="initial_value">0.0</param>
      </state_interface>
      <!-- additional state interfaces ... -->
    </joint>
    <!-- additional joints ...-->
    <!-- additional hardware/sensors ...-->
  </ros2_control>
</robot>
```

- The `robot` tag encloses all contents of the URDF file. It has a name attribute which must be specified.
- The `link` tag defines the robot's geometry and inertia properties. It has a name attribute which will be referred to by the `joint` tags.
- The `visual` tag specifies the rotation and translation of the visual mesh. If the meshes were process as described previously, then the `origin` tag can be left at all zeros.
- The `geometry` and `mesh` tags specify the location of the 3D mesh file relative to a specified ROS 2 package.
- The `collision` tag is equivalent to the `visual` tag, except the specified mesh is used for collision checking in some applications.
- The `inertial` tag specifies mass and inertia for the link. The origin tag specifies the link's center of mass. These values are used to calculate forward and inverse dynamics. Since our application does not use dynamics, uniform arbitrary values are used.
- The `<!-- additional links ... -->` comments indicates that many consecutive `link` tags will be defined, one for each link.
- The `<link name="world"/>` and `<link name="tool0"/>` elements are not required. However, it is convention to set the link at the tip of the robot to tool0 and to define the robot's base link relative to a world frame.
- The `joint` tag specifies the kinematic structure of the robot. It has two required attributes: name and type. The type specifies the viable motion between the two connected links. The subsequent `parent` and `child` links specify which two links are joined by the joint.
- The `axis` tag species the joint's degree of freedom. If the meshes were process as described previously, then the axis value is always `"0 0 1"`.
- The `limits` tag specifies kinematic and dynamics limits for the joint.
- The `ros2_control` tag specifies hardware configuration of the robot. More specifically, the available state and command interfaces. The tag has two required attributes: name and type. Additional elements, such as sensors,

are also included in this tag.

- The `hardware` and `plugin` tags instruct the ros2_control framework to dynamically load a hardware driver conforming to `HardwareInterface` as a plugin. The plugin is specified as `<{Name_Space}/{Class_Name}`.
- Finally, the `joint` tag specifies the state and command interfaces that the loaded plugins will offer. The joint is specified with the name attribute. The `command_interface` and `state_interface` tags specify the interface type, usually position, velocity, acceleration, or effort.

To simplify the URDF file, `xacro` is used to define macros, see this tutorial. The complete xacro file for the robot in this tutorial is available here. To verify the kinematic chain, the tool `urdf_to_graphviz` can be used after the URDF is generated by `xacro`. Running

```
xacro description/urdf/r6bot.urdf.xacro > r6bot.urdf
urdf_to_graphviz r6bot.urdf r6bot
```

generates `r6bot.pdf`, showing the kinematic chain of the robot.

Writing a hardware interface

In ros2_control, hardware system components are integrated via user defined driver plugins that conform to the `HardwareInterface` public interface. Hardware plugins specified in the URDF are dynamically loaded during initialization using the pluginlib interface. In order to run the `ros2_control_node`, a parameter named `robot_description` must be set. This normally done in the ros2_control launch file.

The following code blocks will explain the requirements for writing a new hardware interface.

The hardware plugin for the tutorial robot is a class called `RobotSystem` that inherits from `hardware_interface::SystemInterface`. The `SystemInterface` is one of the offered hardware interfaces designed for a complete robot system. For example, The UR5 uses this interface. The `RobotSystem` must implement the following public methods.

1. `on_init`
2. `on_configure`
3. `read`
4. `write`

There are more methods that can be implemented for lifecycle changes, but they are not required for the tutorial robot.

```
using CallbackReturn = rclcpp_lifecycle::node_
↪interfaces::LifecycleNodeInterface::CallbackReturn;
#include "hardware_interface/types/hardware_interface_return_values.hpp"

class RobotSystem : public hardware_interface::SystemInterface {
    public:
    CallbackReturn on_init(const hardware_interface::HardwareInfo &info) override;
    CallbackReturn on_configure(const rclcpp_lifecycle::State & previous_state)␣
↪override;
    return_type read(const rclcpp::Time &time, const rclcpp::Duration &period)␣
↪override;
    return_type write(const rclcpp::Time & /*time*/, const rclcpp::Duration & /
↪*period*/) override;
    // private members
    // ...
}
```

The `on_init` method is called once during ros2_control initialization if the `RobotSystem` was specified in the URDF. In this method, communication between the robot hardware needs to be setup and memory dynamic should be allocated. Since the tutorial robot is simulated, explicit communication will not be established.

```
CallbackReturn RobotSystem::on_init(const hardware_interface::HardwareInfo &info) {
    if (hardware_interface::SystemInterface::on_init(info) !=␣
↪CallbackReturn::SUCCESS) {
        return CallbackReturn::ERROR;
    }
    // setup communication with robot hardware
    // ...
    return CallbackReturn::SUCCESS;
}
```

Notably, the behavior of `on_init` is expected to vary depending on the URDF file. The `SystemInterface::on_init(info)` call fills out the `info` object with specifics from the URDF. For example, the `info` object has fields for joints, sensors, gpios, and more. Suppose the sensor field has a name value of `tcp_force_torque_sensor`. Then the `on_init` must try to establish communication with that sensor. If it fails, then an error value is returned.

The `on_configure` method is called once during ros2_control lifecycle of the `RobotSystem`. Initial values are set for state and command interfaces that represent the state all the hardware, e.g. doubles describing joint angles, etc.

```
CallbackReturn RobotSystem::on_configure(
  const rclcpp_lifecycle::State & /*previous_state*/)
    // setup communication with robot hardware
    // ...
    return CallbackReturn::SUCCESS;
}
```

The `read` method is core method in the ros2_control loop. During the main loop, ros2_control loops over all hardware components and calls the `read` method. It is executed on the realtime thread, hence the method must obey by realtime constraints. The `read` method is responsible for updating the data values of the `state_interfaces`. Since the data value point to class member variables, those values can be filled with their corresponding sensor values, which will in turn update the values of each exported `StateInterface` object.

```
return_type RobotSystem::read(const rclcpp::Time & time, const rclcpp::Duration &
↪period) {
    // read hardware values for state interfaces, e.g joint encoders and sensor␣
↪readings
    // ...
    return return_type::OK;
}
```

The `write` method is another core method in the ros2_control loop. It is called after `update` in the realtime loop. For this reason, it must also obey by realtime constraints. The `write` method is responsible for updating the data values of the `command_interfaces`. As opposed to `read`, `write` accesses data values pointer to by the exported `CommandInterface` objects sends them to the corresponding hardware. For example, if the hardware supports setting a joint velocity via TCP, then this method accesses data of the corresponding `command_interface` and sends a packet with the value.

```
return_type write(const rclcpp::Time & time, const rclcpp::Duration & period) {
    // send command interface values to hardware, e.g joint set joint velocity
    // ...
    return return_type::OK;
}
```

Finally, all ros2_control plugins should have the following two lines of code at the end of the file.

```
#include "pluginlib/class_list_macros.hpp"

PLUGINLIB_EXPORT_CLASS(robot_6_dof_hardware::RobotSystem, hardware_
↪interface::SystemInterface)
```

`PLUGINLIB_EXPORT_CLASS` is a c++ macro creates a plugin library using `pluginlib`.

Plugin description file (hardware)

The plugin description file is a required XML file that describes a plugin's library name, class type, namespace, description, and interface type. This file allows the ROS 2 to automatically discover and load plugins. It is formatted as follows.

```
<library path="{Library_Name}">
  <class
    name="{Namespace}/{Class_Name}"
    type="{Namespace}::{Class_Name}"
    base_class_type="hardware_interface::SystemInterface">
  <description>
    {Human readable description}
  </description>
  </class>
</library>
```

The `path` attribute of the `library` tags refers to the cmake library name of the user defined hardware plugin. See here for the complete XML file.

CMake library (hardware)

The general CMake template to make a hardware plugin available in ros2_control is shown below. Notice that a library is created using the plugin source code just like any other cmake library. In addition, an extra compile definition and cmake export macro (`pluginlib_export_plugin_description_file`) need to be added. See here for the complete `CMakeLists.txt` file.

```
add_library(
    robot_6_dof_hardware
    SHARED
    src/robot_hardware.cpp
)
```

Writing a controller

In ros2_control, controllers are implemented as plugins that conform to the `ControllerInterface` public interface. Similar to the hardware interfaces, the controller plugins to load are specified using ROS parameters. This is normally achieved by passing a YAML parameter file to the `ros2_control_node`. Unlike hardware interfaces, controllers exists in a finite set of states:

1. Unconfigured
2. Inactive
3. Active
4. Finalized

Certain interface methods are called during transitions between these states. During the main control loop, the controller is in the active state.

The following code blocks will explain the requirements for writing a new controller.

The controller plugin for the tutorial robot is a class called `RobotController` that inherits from `controller_interface::ControllerInterface`. The `RobotController` must implement the following public methods:

1. `command_interface_configuration`
2. `state_interface_configuration`
3. `update`

The following methods are managed node transitions callbacks. These overrides are optional and only the `on_configure`, `on_activate` and `on_deactivate` have been used in this example. The `on_cleanup` and `on_shutdown` methods are called when the controller's lifecycle node transitions to the shutdown state. They should handle memory deallocation and general cleanup. The `on_error` method is invoked if the managed node encounters a failure during a state transition.

1. `on_configure`
2. `on_activate`
3. `on_deactivate`
4. `on_cleanup`
5. `on_error`
6. `on_shutdown`

```
class RobotController : public controller_interface::ControllerInterface {
    public:
    controller_interface::InterfaceConfiguration command_interface_configuration()␣
↪const override;
    controller_interface::InterfaceConfiguration state_interface_configuration()␣
↪const override;
    controller_interface::return_type update(const rclcpp::Time &time, const␣
↪rclcpp::Duration &period) override;
    controller_interface::CallbackReturn on_init() override;
    controller_interface::CallbackReturn on_configure(const rclcpp_lifecycle::State &
↪previous_state) override;
    controller_interface::CallbackReturn on_activate(const rclcpp_lifecycle::State &
↪previous_state) override;
    controller_interface::CallbackReturn on_deactivate(const rclcpp_lifecycle::State &
↪previous_state) override;
// private members
// ...
}
```

The `on_init` method is called immediately after the controller plugin is dynamically loaded. The method is called only once during the lifetime for the controller, hence memory that exists for the lifetime of the controller should be allocated. Additionally, the parameter values for `joints`, `command_interfaces` and `state_interfaces` should be declared and accessed. Those parameter values are required for the next two methods.

```
using CallbackReturn = rclcpp_lifecycle::node_
↪interfaces::LifecycleNodeInterface::CallbackReturn;

controller_interface::CallbackReturn on_init(){
```

(continues on next page)

(continued from previous page)

```
    // declare and get parameters needed for controller initialization
    // allocate memory that will exist for the life of the controller
    // ...
    return CallbackReturn::SUCCESS;
}
```

The `on_configure` method is called immediately after the controller is set to the inactive state. This state occurs when the controller is started for the first time, but also when it is restarted. Reconfigurable parameters should be read in this method. Additionally, publishers and subscribers should be created.

```
controller_interface::CallbackReturn on_configure(const rclcpp_lifecycle::State &
↪previous_state){
    // declare and get parameters needed for controller operations
    // setup realtime buffers, ROS publishers, and ROS subscribers
    // ...
  return CallbackReturn::SUCCESS;
}
```

The `command_interface_configuration` method is called after `on_configure`. The method returns a list of `InterfaceConfiguration` objects to indicate which command interfaces the controller needs to operate. The command interfaces are uniquely identified by their name and interface type. If a requested interface is not offered by a loaded hardware interface, then the controller will fail.

```
controller_interface::InterfaceConfiguration command_interface_configuration(){
    controller_interface::InterfaceConfiguration conf;
    // add required command interface to `conf` by specifying their names and␣
↪interface types.
    // ..
    return conf
}
```

The `state_interface_configuration` method is then called, which is similar to the last method. The difference is that a list of `InterfaceConfiguration` objects representing the required state interfaces to operate is returned.

```
controller_interface::InterfaceConfiguration state_interface_configuration() {
    controller_interface::InterfaceConfiguration conf;
    // add required state interface to `conf` by specifying their names and interface␣
↪types.
    // ..
    return conf
}
```

The `on_activate` is called once when the controller is activated. This method should handle controller restarts, such as setting the resetting reference to safe values. It should also perform controller specific safety checks. The `command_interface_configuration` and `state_interface_configuration` methods are also called again when the controller is activated.

```
controller_interface::CallbackReturn on_activate(const rclcpp_lifecycle::State &
↪previous_state){
  // Handle controller restarts and dynamic parameter updating
  // ...
  return CallbackReturn::SUCCESS;
}
```

The `update` method is part of the main control loop. Since the method is part of the realtime control loop, the realtime

constraint must be enforced. The controller should read from its state interfaces, read its reference and calculate a control output. Normally, the reference is accessed via a ROS 2 subscriber. Since the subscriber runs on the non-realtime thread, a realtime buffer is used to a transfer the message to the realtime thread. The realtime buffer is eventually a pointer to a ROS message with a mutex that guarantees thread safety and that the realtime thread is never blocked. The calculated control output should then be written to the command interface, which will in turn control the hardware.

```
controller_interface::return_type update(const rclcpp::Time &time, const␣
↪rclcpp::Duration &period){
  // Read controller inputs values from state interfaces
  // Calculate controller output values and write them to command interfaces
  // ...
  return controller_interface::return_type::OK;
}
```

The `on_deactivate` is called when a controller stops running. In our case it is empty:

```
controller_interface::CallbackReturn on_deactivate(const rclcpp_lifecycle::State &
↪previous_state){
    // The controller should be properly shutdown during this
    // ...
    return CallbackReturn::SUCCESS;
}
```

Plugin description file (controller)

The plugin description file is again required for the controller, since it is exported as a library. The controller plugin description file is formatted as follows. See here for the complete XML file.

```
<library path="{Library_Name}">
  <class
    name="{Namespace}/{Class_Name}"
    type="{Namespace}::{Class_Name}"
    base_class_type="controller_interface::ControllerInterface">
  <description>
    {Human readable description}
  </description>
  </class>
</library>
```

CMake library (controller)

The plugin must be specified in the CMake file that builds the controller plugin. See here for the complete `CMakeLists.txt` file.

```
add_library(
    r6bot_controller
    SHARED
    src/robot_controller.cpp
)
```

Launching the example

The full tutorial example can be run by first building the workspace.

```
git clone -b master https://github.com/ros-controls/ros2_control_demos.git
cd ros2_control_demos
colcon build --symlink-install
source install/setup.bash
```

To view the robot, open a terminal and launch the `view_r6bot.launch.py` file from the `ros2_control_demo_example_7` package.

```
ros2 launch ros2_control_demo_example_7 view_r6bot.launch.py
```

With the `joint_state_publisher_gui` you can now change the position of every joint.

Next, kill the process in the launch file and start the simulation of the controlled robot. Open a terminal and launch the `r6bot_controller.launch.py` file from the `ros2_control_demo_example_7` package.

```
ros2 launch ros2_control_demo_example_7 r6bot_controller.launch.py
```

Finally, open a new terminal and run the following command.

```
ros2 launch ros2_control_demo_example_7 send_trajectory.launch.py
```

You should see the tutorial robot making a circular motion in RViz.

4.6.8 Example 8: Industrial Robots with an exposed transmission interface

RRBot, or "Revolute-Revolute Manipulator Robot", is a simple 3-linkage, 2-joint arm that we will use to demonstrate various features.

In this example, both joints use an exposed transmission interface:

- The communication is done using proprietary API to communicate with the robot control box.
- Data for all joints is exchanged at once.

Note: The commands below are given for a local installation of this repository and its dependencies as well as for running them from a docker container. For more information on the docker usage see *Using Docker*.

Tutorial steps

1. To check that *RRBot* descriptions are working properly use following launch commands

   ```
   ros2 launch ros2_control_demo_example_8 view_robot.launch.py
   ```

   Note: Getting the following output in terminal is OK: `Warning: Invalid frame ID "odom" passed to canTransform argument target_frame - frame does not exist`. This happens because `joint_state_publisher_gui` node need some time to start. The `joint_state_publisher_gui` provides a GUI to change the configuration for rrbot. It is immediately displayed in *RViz*.

   ---

2. To start *RRBot* example open a terminal, source your ROS2-workspace and execute its launch file with

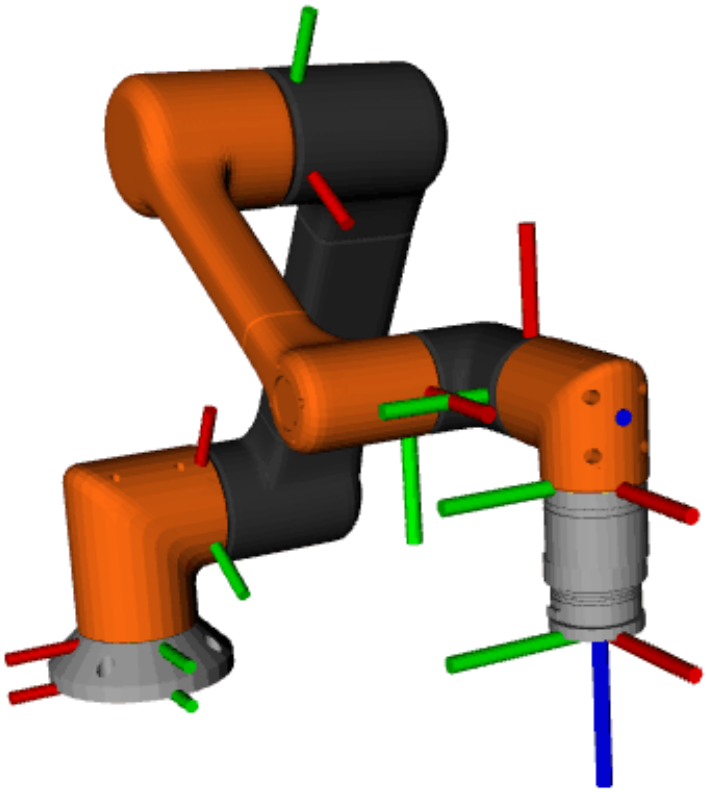


Fig. 7: Trajectory following example.

```
ros2 launch ros2_control_demo_example_8 rrbot_transmissions_system_position_only.
↪launch.py
```

The launch file loads and starts the robot hardware, controllers and opens *RViz*. In starting terminal you will see a lot of output from the hardware implementation showing its internal states. This is only of exemplary purposes and should be avoided as much as possible in a hardware interface implementation.

If you can see two orange and one yellow rectangle in in *RViz* everything has started properly. Still, to be sure, let's introspect the control system before moving *RRBot*.

3. Check if the hardware interface loaded properly, by opening another terminal and executing

```
ros2 control list_hardware_interfaces
```

```
command interfaces
      joint1/position [available] [claimed]
      joint2/position [available] [claimed]
state interfaces
      joint1/position
      joint2/position
```

Marker `[claimed]` by command interfaces means that a controller has access to command *RRBot*.

4. Check if controllers are running by

```
ros2 control list_controllers
```

```
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
forward_position_controller[forward_command_controller/ForwardCommandController]␣
↪active
```

5. If you get output from above you can send commands to *Forward Command Controller*, either:

   a. Manually using ROS 2 CLI interface:

```
ros2 topic pub /forward_position_controller/commands std_msgs/msg/
↪Float64MultiArray "data:
- 0.5
- 0.5"
```

   B. Or you can start a demo node which sends two goals every 5 seconds in a loop

```
ros2 launch ros2_control_demo_example_8 test_forward_position_controller.launch.py
```

You should now see orange and yellow blocks moving in *RViz*. Also, you should see changing states in the terminal where launch file is started, e.g.

```
[ros2_control_node-1] [INFO] [1728857106.562714002] [controller_manager.resource_
↪manager.hardware_component.system.RRBotTransmissionsSystemPositionOnly]:␣
↪Command data:
[ros2_control_node-1]   joint1: 0.5 --> transmission1(R=2) --> actuator1: 1
[ros2_control_node-1]   joint2: 0.5 --> transmission2(R=4) --> actuator2: 2
[ros2_control_node-1] [INFO] [1728857106.762624114] [controller_manager.resource_
↪manager.hardware_component.system.RRBotTransmissionsSystemPositionOnly]: State␣
↪data:
[ros2_control_node-1]   joint1: 0.166196 <-- transmission1(R=2) <-- actuator1: 0.
↪332392
```

(continues on next page)

(continued from previous page)

```
[ros2_control_node-1]   joint2: 0.166196 <-- transmission2(R=4) <-- actuator2: 0.
↪664784
```

Files used for this demos

- Launch file: rrbot_transmissions_system_position_only.launch.py
- Controllers yaml: rrbot_controllers.yaml
- URDF file: rrbot_transmissions_system_position_only.urdf.xacro
  - Description: rrbot_description.urdf.xacro
  - `ros2_control` tag: rrbot_transmissions_system_position_only.ros2_control.xacro
- RViz configuration: rrbot.rviz
- Hardware interface plugin: rrbot_transmissions_system_position_only.cpp

Controllers from this demo

- `Joint State Broadcaster` (ros2_controllers repository): *doc*
- `Forward Command Controller` (ros2_controllers repository): *doc*

4.6.9 Example 9: Simulation with RRBot

With *example_9*, we demonstrate the interaction of simulators with ros2_control. More specifically, gazebo is used for this purpose.

Note: Follow the installation instructions on *Installation* how to install all dependencies, gazebo should be automatically installed. For details on the `gz_ros2_control` plugin, see *gz_ros2_control*.

Tutorial steps

1. To check that *RRBot* descriptions are working properly use following launch commands

   ```
   ros2 launch ros2_control_demo_example_9 view_robot.launch.py
   ```

   The `joint_state_publisher_gui` provides a GUI to change the configuration for *RRbot*. It is immediately displayed in *RViz*.

2. To start *RRBot* with the hardware interface instead of the simulators, open a terminal, source your ROS2-workspace and execute its launch file with

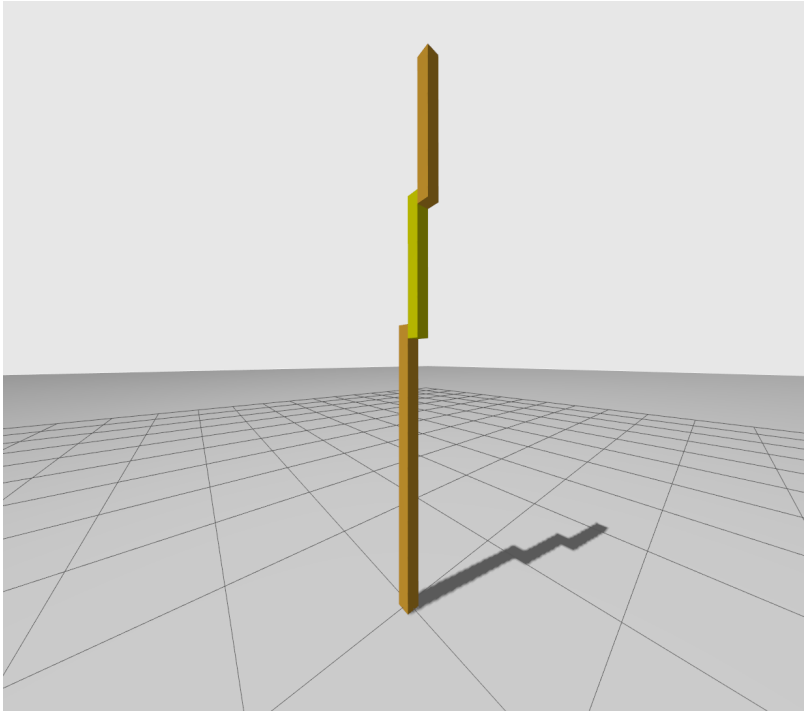
   ```
   ros2 launch ros2_control_demo_example_9 rrbot.launch.py
   ```

   It uses an identical hardware interface as already discussed with *example_1*, see its docs on details on the hardware interface.

3. To start *RRBot* in the simulators, open a terminal, source your ROS2-workspace first. Then, execute the launch file with

```
ros2 launch ros2_control_demo_example_9 rrbot_gazebo.launch.py
```

The launch file loads the robot description, starts gazebo, *Joint State Broadcaster* and *Forward Command Controller*. If you can see two orange and one yellow "box" in gazebo everything has started properly.



4. Check if the hardware interface loaded properly, by opening another terminal and executing

```
ros2 control list_hardware_interfaces
```

```
command interfaces
      joint1/position [available] [claimed]
      joint2/position [available] [claimed]
state interfaces
      joint1/position
      joint2/position
```

Marker `[claimed]` by command interfaces means that a controller has access to command *RRBot*.

5. Check if controllers are running by

```
ros2 control list_controllers
```

```
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
forward_position_controller[forward_command_controller/ForwardCommandController]␣
↪active
```

6. If you get output from above you can send commands to *Forward Command Controller*, either:

   a. Manually using ROS 2 CLI interface:

```
ros2 topic pub /forward_position_controller/commands std_msgs/msg/
↪Float64MultiArray "data:
```

(continues on next page)

(continued from previous page)

```
- 0.5
- 0.5"
```

B. Or you can start a demo node which sends two goals every 5 seconds in a loop

```
ros2 launch ros2_control_demo_example_9 test_forward_position_controller.launch.py
```

You should now see the robot moving in gazebo.

If you echo the `/joint_states` or `/dynamic_joint_states` topics you should see the changing values, namely the simulated states of the robot

```
ros2 topic echo /joint_states
ros2 topic echo /dynamic_joint_states
```

Files used for this demos

- Launch files:
  - Hardware: rrbot.launch.py
  - gazebo: rrbot_gazebo.launch.py
- Controllers yaml: rrbot_controllers.yaml
- URDF file: rrbot.urdf.xacro
  - Description: rrbot_description.urdf.xacro
  - `ros2_control` tag: rrbot.ros2_control.xacro
- RViz configuration: rrbot.rviz
- Test nodes goals configuration:
  - rrbot_forward_position_publisher
- Hardware interface plugin: rrbot.cpp

Controllers from this demo

- `Joint State Broadcaster` (ros2_controllers repository): *doc*
- `Forward Command Controller` (ros2_controllers repository): *doc*

4.6.10 Example 10: Industrial robot with GPIO interfaces

This demo shows how to interact with GPIO interfaces.

The *RRBot* URDF files can be found in the `description/urdf` folder.

1. To check that *RRBot* descriptions are working properly use following launch commands

   ```
   ros2 launch ros2_control_demo_example_10 view_robot.launch.py
   ```

2. To start *RRBot* example open a terminal, source your ROS2-workspace and execute its launch file with

```
ros2 launch ros2_control_demo_example_10 rrbot.launch.py
```

The launch file loads and starts the robot hardware and controllers.

3. Check if the hardware interface loaded properly, by opening another terminal and executing

```
ros2 control list_hardware_interfaces
```

```
command interfaces
    flange_analog_IOs/analog_output1 [available] [claimed]
    flange_vacuum/vacuum [available] [claimed]
    joint1/position [available] [claimed]
    joint2/position [available] [claimed]
state interfaces
    flange_analog_IOs/analog_input1
    flange_analog_IOs/analog_input2
    flange_analog_IOs/analog_output1
    flange_vacuum/vacuum
    joint1/position
    joint2/position
```

In contrast to the *RRBot* of example_1, you see in addition to the joints now also GPIO interfaces.

4. Check if controllers are running by

```
ros2 control list_controllers
```

```
joint_state_broadcaster     joint_state_broadcaster/JointStateBroadcaster       ␣
↪active
gpio_controller             gpio_controllers/GpioCommandController              ␣
↪active
forward_position_controller forward_command_controller/ForwardCommandController ␣
↪active
```

5. If you get output from above you can subscribe to the `/dynamic_joint_states` topic published by the *joint_state_broadcaster* using ROS 2 CLI interface:

```
ros2 topic echo /dynamic_joint_states --once
```

This includes not only the state interfaces of the joints but also the GPIO interfaces.

```
header:
  stamp:
    sec: 1730670203
    nanosec: 875008879
  frame_id: ''
joint_names:
- joint1
- joint2
- flange_vacuum
- flange_analog_IOs
interface_values:
- interface_names:
  - position
  values:
  - 0.0
- interface_names:
```

(continues on next page)

(continued from previous page)

```
  - position
  values:
  - 0.0
- interface_names:
  - vacuum
  values:
  - 0.0
- interface_names:
  - analog_input2
  - analog_input1
  - analog_output1
  values:
  - 92747888.0
  - 1764536320.0
  - 0.0
---
```

You can also subscribe to the `/gpio_controller/gpio_states` topic published by the *gpio_controller* using ROS 2 CLI interface:

```
ros2 topic echo /gpio_controller/gpio_states
```

which shows the values of the state_interfaces of the configured GPIOs

```
header:
  stamp:
    sec: 1731875120
    nanosec: 2015630
  frame_id: ''
interface_groups:
- flange_analog_IOs
- flange_vacuum
interface_values:
- interface_names:
  - analog_output1
  - analog_input1
  - analog_input2
  values:
  - 0.0
  - 991951680.0
  - 1467646976.0
- interface_names:
  - vacuum
  values:
  - 0.0
---
```

6. Now you can send commands to the *gpio_controller* using ROS 2 CLI interface. You can set a single interface or all at once in one message:

```
ros2 topic pub /gpio_controller/commands control_msgs/msg/
↪DynamicInterfaceGroupValues "{interface_groups: [flange_analog_IOs], interface_
↪values: [{interface_names: [analog_output1], values: [0.5]}]}"

ros2 topic pub /gpio_controller/commands control_msgs/msg/
↪DynamicInterfaceGroupValues "{interface_groups: [flange_vacuum], interface_
```

(continues on next page)

(continued from previous page)

```
↪values: [{interface_names: [vacuum], values: [0.27]}]}"

ros2 topic pub /gpio_controller/commands control_msgs/msg/
↪DynamicInterfaceGroupValues "{interface_groups: [flange_vacuum, flange_analog_
↪IOs], interface_values: [{interface_names: [vacuum], values: [0.27]},
↪{interface_names: [analog_output1], values: [0.5]} ]}"
```

You should see a change in the `/gpio_controller/gpio_states` topic and a different output in the terminal where launch file is started, e.g.

```
[ros2_control_node-1] [INFO] [1721765648.271058850] [controller_manager.resource_
↪manager.hardware_component.system.RRBot]: Writing commands:
[ros2_control_node-1]    0.50 for GPIO output '0'
[ros2_control_node-1]    0.70 for GPIO output '1'
```

7. Let's introspect the ros2_control hardware component. Calling

```
ros2 control list_hardware_components
```

should give you

```
Hardware Component 1
    name: RRBot
    type: system
    plugin name: ros2_control_demo_example_10/RRBotSystemWithGPIOHardware
    state: id=3 label=active
    command interfaces
            joint1/position [available] [claimed]
            joint2/position [available] [claimed]
            flange_analog_IOs/analog_output1 [available] [claimed]
            flange_vacuum/vacuum [available] [claimed]
```

This shows that the custom hardware interface plugin is loaded and running. If you work on a real robot and don't have a simulator running, it is often faster to use the `mock_components/GenericSystem` hardware component instead of writing a custom one. Stop the launch file and start it again with an additional parameter

```
ros2 launch ros2_control_demo_example_10 rrbot.launch.py use_mock_
↪hardware:=True
```

Calling `list_hardware_components` with the `-v` option

```
ros2 control list_hardware_components -v
```

now should give you

```
Hardware Component 1
    name: RRBot
    type: system
    plugin name: mock_components/GenericSystem
    state: id=3 label=active
    command interfaces
            joint1/position [available] [claimed]
            joint2/position [available] [claimed]
            flange_analog_IOs/analog_output1 [available] [claimed]
            flange_vacuum/vacuum [available] [claimed]
```

(continues on next page)

(continued from previous page)

```
    state interfaces
            joint1/position [available]
            joint2/position [available]
            flange_analog_IOs/analog_output1 [available]
            flange_analog_IOs/analog_input1 [available]
            flange_analog_IOs/analog_input2 [available]
            flange_vacuum/vacuum [available]
```

One can see that the plugin `mock_components/GenericSystem` was now loaded instead: It will mirror the command interfaces to state interfaces with identical name. Call

```
ros2 topic echo /gpio_controller/gpio_states
```

again and you should see that - unless commands are received - the values of the state interfaces are now `nan` except for the vacuum interface.

```
header:
  stamp:
    sec: 1731875298
    nanosec: 783713170
  frame_id: ''
interface_groups:
- flange_analog_IOs
- flange_vacuum
interface_values:
- interface_names:
  - analog_output1
  - analog_input1
  - analog_input2
  values:
  - .nan
  - .nan
  - .nan
- interface_names:
  - vacuum
  values:
  - 1.0
---
```

This is, because for the vacuum interface an initial value of `1.0` is set in the URDF file.

```
<gpio name="flange_vacuum">
  <command_interface name="vacuum"/>
  <state_interface name="vacuum">
    <param name="initial_value">1.0</param>
  </state_interface>
</gpio>
```

Send again topics to `/gpio_controller/commands` and you will see that the GPIO command interfaces will be mirrored to their respective state interfaces.

Files used for this demos

- Launch file: rrbot.launch.py
- Controllers yaml: rrbot_controllers.yaml
- URDF file: rrbot.urdf.xacro
  - Description: rrbot_description.urdf.xacro
  - `ros2_control` tag: rrbot.ros2_control.xacro
- RViz configuration: rrbot.rviz
- Hardware interface plugin:
  - rrbot.cpp
  - generic_system.cpp

Controllers from this demo

- `Joint State Broadcaster` (ros2_controllers repository): *doc*
- `Forward Command Controller` (ros2_controllers repository): *doc*
- `GPIO Command Controller` (ros2_controllers repository): *doc*

4.6.11 CarlikeBot

CarlikeBot is a simple mobile base using bicycle model with 4 wheels.

This example shows how to use the bicycle steering controller, which is a sub-design of the steering controller library.

Even though the robot has 4 wheels with front steering, the vehicle dynamics of this robot is similar to a bicycle. There is a virtual front wheel joint that is used to control the steering angle of the front wheels and the front wheels on the robot mimic the steering angle of the virtual front wheel joint. Similarly the rear wheels are controlled by a virtual rear wheel joint.

This example shows how to use the bicycle steering controller to control a carlike robot with 4 wheels but only 2 joints that can be controlled, one for steering and one for driving.

- The communication is done using proprietary API to communicate with the robot control box.
- Data for all joints is exchanged at once.

The *CarlikeBot* URDF files can be found in `ros2_control_demo_description/carlikebot/urdf` folder.

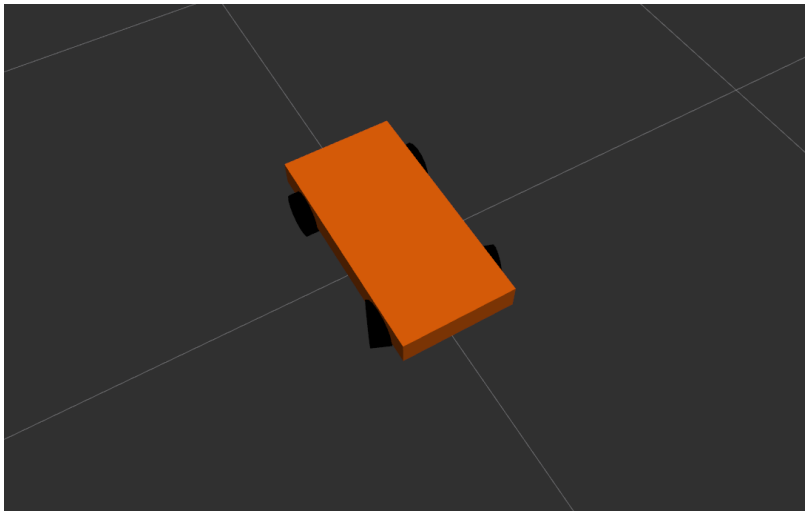
Note: The commands below are given for a local installation of this repository and its dependencies as well as for running them from a docker container. For more information on the docker usage see *Using Docker*.

Tutorial steps

1. To check that *CarlikeBot* description is working properly use following launch commands

```
ros2 launch ros2_control_demo_example_11 view_robot.launch.py
```

Warning: Getting the following output in terminal is OK: `Warning: Invalid frame ID "odom" passed to canTransform argument target_frame - frame does not exist.` This happens because `joint_state_publisher_gui` node needs some time to start.



2. To start *CarlikeBot* example open a terminal, source your ROS2-workspace and execute its launch file with

```
ros2 launch ros2_control_demo_example_11 carlikebot.launch.py remap_odometry_
↪tf:=true
```

The launch file loads and starts the robot hardware, controllers and opens *RViz*. In the starting terminal you will see a lot of output from the hardware implementation showing its internal states. This excessive printing is only added for demonstration. In general, printing to the terminal should be avoided as much as possible in a hardware interface implementation.

If you can see an orange box with four wheels in *RViz* everything has started properly.

Note: For robots being fixed to the world frame, like the RRbot examples of this repository, the `robot_state_publisher` subscribes to the `/joint_states` topic and creates the TF tree. For mobile robots, we need a node publishing the TF tree including the pose of the robot in the world coordinate systems. The most simple one is the odometry calculated by the `bicycle_steering_controller`.

By default, the controller publishes the odometry of the robot to the `~/tf_odometry` topic. The `remap_odometry_tf` argument is used to remap the odometry TF to the `/tf` topic. If you set this argument to `false` (or not set it at all) the TF tree will not be updated with the odometry data.

3. Now, let's introspect the control system before moving *CarlikeBot*. Check if the hardware interface loaded properly, by opening another terminal and executing

```
ros2 control list_hardware_interfaces
```

You should get

```
command interfaces
   bicycle_steering_controller/angular/position [unavailable] [unclaimed]
   bicycle_steering_controller/linear/velocity [unavailable] [unclaimed]
   virtual_front_wheel_joint/position [available] [claimed]
   virtual_rear_wheel_joint/velocity [available] [claimed]
state interfaces
   virtual_front_wheel_joint/position
   virtual_rear_wheel_joint/position
   virtual_rear_wheel_joint/velocity
```

The `[claimed]` marker on command interfaces means that a controller has access to command *CarlikeBot*.

4. Check if controllers are running

```
ros2 control list_controllers
```

You should get

```
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
bicycle_steering_controller[bicycle_steering_controller/
↪BicycleSteeringController] active
```

5. If everything is fine, now you can send a command to *bicycle_steering_controller* using ROS 2 CLI:

```
ros2 topic pub --rate 30 /bicycle_steering_controller/reference geometry_msgs/msg/
↪TwistStamped "
  header: auto
  twist:
    linear:
      x: 1.0
      y: 0.0
      z: 0.0
    angular:
      x: 0.0
      y: 0.0
      z: 0.1"
```

You should now see an orange box circling in *RViz*. Also, you should see changing states in the terminal where launch file is started.

```
[ros2_control_node-1] [INFO] [1721766165.108212153] [controller_manager.resource_
↪manager.hardware_component.system.CarlikeBot]: Writing commands:
[ros2_control_node-1]    position: 0.03 for joint 'virtual_front_wheel_joint'
[ros2_control_node-1]    velocity: 20.00 for joint 'virtual_rear_wheel_joint'
```

Files used for this demos

- Launch file: carlikebot.launch.py
- Controllers yaml: carlikebot_controllers.yaml
- URDF file: carlikebot.urdf.xacro
  - Description: carlikebot_description.urdf.xacro
  - `ros2_control` tag: carlikebot.ros2_control.xacro
- RViz configuration: carlikebot.rviz

- Hardware interface plugin: carlikebot_system.cpp

Controllers from this demo

- `Joint State Broadcaster` (ros2_controllers repository): *doc*
- `Bicycle Steering Controller` (ros2_controllers repository): *doc*

4.6.12 Example 12: Controller chaining with RRBot

The example shows how to write a simple chainable controller, and how to integrate it properly to have a functional controller chaining.

For *example_12*, we will use RRBot, or "Revolute-Revolute Manipulator Robot", is a simple 3-linkage, 2-joint arm to demonstrate the controller chaining functionality in ROS2 control.

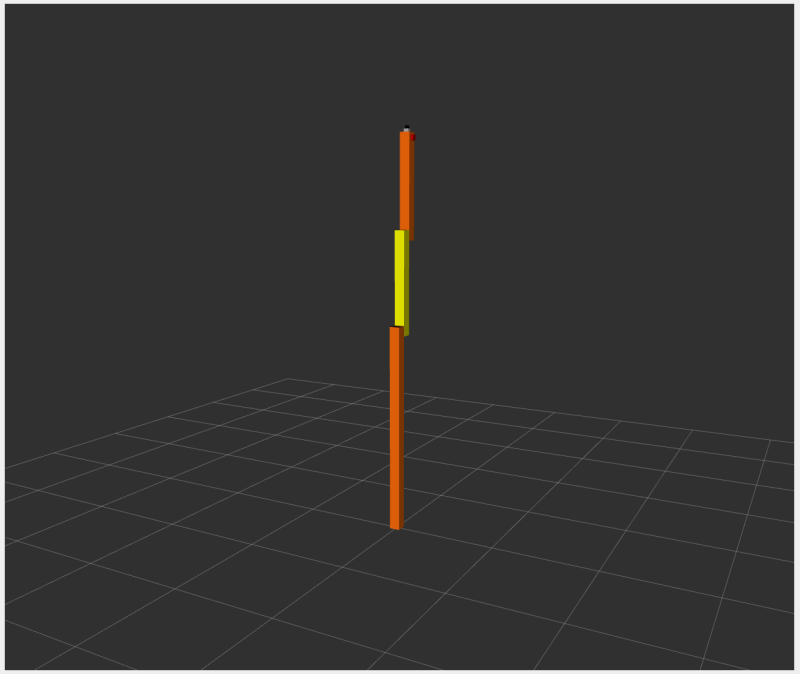
For *example_12*, a simple chainable ros2_controller has been implemented that takes a vector of interfaces as an input and simple forwards them without any changes. Such a controller is simple known as a `passthrough_controller`.

Note: The commands below are given for a local installation of this repository and its dependencies as well as for running them from a docker container. For more information on the docker usage see *Using Docker*.

Tutorial steps

1. To check that *RRBot* descriptions are working properly use following launch commands

```
ros2 launch ros2_control_demo_example_12 view_robot.launch.py
```



The `joint_state_publisher_gui` provides a GUI to change the configuration for *RRbot*. It is immediately displayed in *RViz*.

2. To start *RRBot* with the hardware interface, open a terminal, source your ROS2-workspace and execute its launch file with

```
ros2 launch ros2_control_demo_example_12 rrbot.launch.py
```

The launch file loads and starts the robot hardware, controllers and opens RViz. In starting terminal you will see a lot of output from the hardware implementation showing its internal states. It uses an identical hardware interface as already discussed with *example_1*, see its docs on details on the hardware interface.

If you can see two orange and one yellow rectangle in in RViz everything has started properly. Still, to be sure, let's introspect the control system before moving RRBot.

3. Check if controllers are running by

```
ros2 control list_controllers
```

```
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
joint2_position_controller[passthrough_controller/PassthroughController] active
joint1_position_controller[passthrough_controller/PassthroughController] active
```

4. Check if the hardware interface loaded properly, by opening another terminal and executing

```
ros2 control list_hardware_interfaces
```

The output should be something like this:

```
command interfaces
      joint1/position [available] [claimed]
      joint1_position_controller/joint1/position [unavailable] [unclaimed]
      joint2/position [available] [claimed]
      joint2_position_controller/joint2/position [unavailable] [unclaimed]
state interfaces
      joint1/position
      joint2/position
```

At this stage the reference interfaces of controllers are listed under `command_interfaces` when `ros2 control list_hardware_interfaces` command is executed.

- Marker `[available]` by command interfaces means that the hardware interfaces are available and are ready to command.
- Marker `[claimed]` by command interfaces means that a controller has access to command *RRBot*.
- Marker `[unavailable]` by command interfaces means that the hardware interfaces are unavailable and cannot be commanded. For instance, when there is an error in reading or writing an actuator module, it's interfaces are automatically become unavailable.
- Marker `[unclaimed]` by command interfaces means that the reference interfaces of `joint1_position_controller` and `joint2_position_controller` are not yet in chained mode. However, their reference interfaces are available to be chained, as the controllers are active.

Note: In case of chained controllers, the command interfaces appear to be `unavailable` and `unclaimed`, even though the controllers whose exposed reference interfaces are active, because these command interfaces become `available` only in chained mode i.e., when an another controller makes use of these command interface. In non-chained mode, it is expected for the chained controller to use references from subscribers, hence they are marked as `unavailable`.

---

5. To start the complete controller chain, open a terminal, source your ROS2-workspace and execute its launch file with

```
ros2 launch ros2_control_demo_example_12 launch_chained_controllers.launch.py
```

This launch file starts the `position_controller` that uses the reference interfaces of both `joint1_position_controller` and `joint2_position_controller` and streamlines into one, and then the `forward_position_controller` uses the reference interfaces of the `position_controller` to command the *RRBot* joints.

Note: The second level `position_controller` is only added for demonstration purposes, however, a new chainable controller can be configured to directly command the reference interfaces of both `joint1_position_controller` and `joint2_position_controller`.

6. Check if the new controllers are running by

```
ros2 control list_controllers
```

```
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
joint2_position_controller[passthrough_controller/PassthroughController] active
joint1_position_controller[passthrough_controller/PassthroughController] active
position_controller [passthrough_controller/PassthroughController] active
forward_position_controller[forward_command_controller/ForwardCommandController]␣
↪active
```

7. Now check if the interfaces are loaded properly, by opening another terminal and executing

```
ros2 control list_hardware_interfaces
```

The output should be something like this:

```
command interfaces
      joint1/position [available] [claimed]
      joint1_position_controller/joint1/position [available] [claimed]
      joint2/position [available] [claimed]
      joint2_position_controller/joint2/position [available] [claimed]
      position_controller/joint1_position_controller/joint1/position [available]␣
↪[claimed]
      position_controller/joint2_position_controller/joint2/position [available]␣
↪[claimed]
state interfaces
      joint1/position
      joint2/position
```

At this stage the reference interfaces of all the controllers are listed under `command_interfaces` should be `available` and `claimed` when `ros2 control list_hardware_interfaces` command is executed. Marker `[claimed]` by command interfaces means that a controller has access to command *RRBot*.

8. If you get output from above you can send commands to *Forward Command Controller*:

```
ros2 topic pub /forward_position_controller/commands std_msgs/msg/
↪Float64MultiArray "data:
- 0.5
- 0.5"
```

You should now see orange and yellow blocks moving in *RViz*. Also, you should see changing states in the terminal where launch file is started, e.g.

```
[ros2_control_node-1] [INFO] [1721766407.439574931] [controller_manager.resource_
↪manager.hardware_component.system.RRBot]: Writing commands:
[ros2_control_node-1]    0.50 for joint 'joint1'
[ros2_control_node-1]    0.50 for joint 'joint2'
```

If you echo the `/joint_states` or `/dynamic_joint_states` topics you should now get similar values, namely the simulated states of the robot

```
ros2 topic echo /joint_states
ros2 topic echo /dynamic_joint_states
```

This clearly shows that the controller chaining is functional, as the commands sent to the `forward_position_controller` are passed through properly and then it is reflected in the hardware interfaces of the *RRBot*.

Files used for this demos

- Launch files:
  - Hardware: rrbot.launch.py
  - Controllers: rrbot.launch.py
- ROS2 Controller: passthrough_controller.cpp
- Controllers yaml: rrbot_controllers.yaml
- URDF file: rrbot.urdf.xacro
  - Description: rrbot_description.urdf.xacro
  - `ros2_control` tag: rrbot.ros2_control.xacro
- RViz configuration: rrbot.rviz
- Hardware interface plugin: rrbot.cpp

Controllers from this demo

- `Joint State Broadcaster` (ros2_controllers repository): *doc*
- `Forward Command Controller` (ros2_controllers repository): *doc*

4.6.13 Example 13: Multi-robot system with lifecycle management

This example shows how to include multiple robots in a single controller manager instance. Additionally, hardware lifecycle management is demonstrated.

Hardware and interfaces

- RRBotSystemPositionOnly (auto-start)
  - Command interfaces:
    - rrbot_joint1/position
    - rrbot_joint2/position
  - State interfaces:
    - rrbot_joint1/position
    - rrbot_joint2/position
- ExternalRRBotFTSensor (auto-start)
  - State interfaces:
    - rrbot_tcp_fts_sensor/force.x
    - rrbot_tcp_fts_sensor/force.y
    - rrbot_tcp_fts_sensor/force.z
    - rrbot_tcp_fts_sensor/torque.x
    - rrbot_tcp_fts_sensor/torque.y
    - rrbot_tcp_fts_sensor/torque.z
- RRBotSystemWithSensor (auto-configure)
  - Command interfaces:
    - rrbot_with_sensor_joint1/position
    - rrbot_with_sensor_joint2/position
  - State interfaces:
    - rrbot_with_sensor_joint1/position
    - rrbot_with_sensor_joint2/position
    - rrbot_with_sensor_tcp_fts_sensor/force.x
    - rrbot_with_sensor_tcp_fts_sensor/torque.z
- ThreeDofBot
  - Command interfaces
    - threedofbot_joint1/position
    - threedofbot_joint1/pid_gain
    - threedofbot_joint2/position
    - threedofbot_joint2/pid_gain
    - threedofbot_joint3/position
    - threedofbot_joint3/pid_gain
  - State interfaces:
    - threedofbot_joint1/position
    - threedofbot_joint1/pid_gain

    * threedofbot_joint2/position
    * threedofbot_joint2/pid_gain
    * threedofbot_joint3/position
    * threedofbot_joint3/pid_gain

Available controllers

- Global
  - `joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]`
- RRBotSystemPositionOnly
  - `rrbot_joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]`
  - `rrbot_position_controller[forward_command_controller/ForwardCommandController]`
- ExternalRRBotFTSensor
  - `rrbot_external_fts_broadcaster[force_torque_sensor_broadcaster/ForceTorqueSensorBroadcaster]`
- RRBotSystemPositionOnly
  - `rrbot_with_sensor_joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]`
  - `rrbot_with_sensor_position_controller[forward_command_controller/ForwardCommandController]`
  - `rrbot_with_sensor_fts_broadcaster[force_torque_sensor_broadcaster/ForceTorqueSensorBroadcaster]`
- FakeThreeDofBot
  - `threedofbot_joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]`
  - `threedofbot_position_controller[forward_command_controller/ForwardCommandController]`
  - `threedofbot_pid_gain_controller[forward_command_controller/ForwardCommandController]`

Caveats on hardware lifecycling

- There is currently no synchronization between available interface and controllers using them. This means that you should stop controller before making interfaces they are using unavailable. If you don't do this and deactivate/cleanup your interface first your computer will catch fire!
- Global Joint State Broadcaster will not broadcast interfaces that become available after it is started. To solve this restart it manually, for now. During restart TF-transforms are not available.
- There is a possibility that hardware lifecycling (state changes) interfere with the `update`-loop if you are trying to start/stop a controller at the same time.

Tutorial steps

Note: The commands below are given for a local installation of this repository and its dependencies as well as for running them from a docker container. For more information on the docker usage see *Using Docker*.

1. After starting the example with

```
ros2 launch ros2_control_demo_example_13 three_robots.launch.py
```

there should be the following scene:

- right robot is moving (RRBotSystemPositionOnly - using auto-start)
  - All interfaces are available and position controller is started and receives commands
  - all controllers running
- left robot is standing upright (RRBotWithSensor - using auto-configure)
  - only state interfaces are available therefore it can visualized, but not moved
  - only position command controller is not running
- middle robot is "broken" (FakeThreeDofBot - it is only initialized)
  - no interfaces are available
  - all controllers inactive

Hardware status:

```
$ ros2 control list_hardware_components
Hardware Component 1
  name: FakeThreeDofBot
  type: system
  plugin name: mock_components/GenericSystem
  state: id=1 label=unconfigured
  command interfaces
          threedofbot_joint1/position [unavailable] [unclaimed]
          threedofbot_joint1/pid_gain [unavailable] [unclaimed]
          threedofbot_joint2/position [unavailable] [unclaimed]
          threedofbot_joint2/pid_gain [unavailable] [unclaimed]
          threedofbot_joint3/position [unavailable] [unclaimed]
          threedofbot_joint3/pid_gain [unavailable] [unclaimed]
Hardware Component 2
        name: RRBotSystemWithSensor
        type: system
        plugin name: ros2_control_demo_hardware/RRBotSystemWithSensorHardware
        state: id=2 label=inactive
        command interfaces
                rrbot_with_sensor_joint1/position [available] [unclaimed]
                rrbot_with_sensor_joint2/position [available] [unclaimed]
Hardware Component 3
        name: ExternalRRBotFTSensor
        type: sensor
        plugin name: ros2_control_demo_hardware/
↪ExternalRRBotForceTorqueSensorHardware
        state: id=3 label=active
        command interfaces
```

(continues on next page)

(continued from previous page)

```
Hardware Component 4
        name: RRBotSystemPositionOnly
        type: system
        plugin name: ros2_control_demo_hardware/
↪RRBotSystemPositionOnlyHardware
        state: id=3 label=active
        command interfaces
                rrbot_joint1/position [available] [claimed]
                rrbot_joint2/position [available] [claimed]
```

Controllers status:

```
$ ros2 control list_controllers
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
rrbot_external_fts_broadcaster[force_torque_sensor_broadcaster/
↪ForceTorqueSensorBroadcaster] active
rrbot_joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]␣
↪active
rrbot_position_controller[forward_command_controller/
↪ForwardCommandController] active
rrbot_with_sensor_fts_broadcaster[force_torque_sensor_broadcaster/
↪ForceTorqueSensorBroadcaster] active
rrbot_with_sensor_joint_state_broadcaster[joint_state_broadcaster/
↪JointStateBroadcaster] active
rrbot_with_sensor_position_controller[forward_command_controller/
↪ForwardCommandController] inactive
threedofbot_joint_state_broadcaster[joint_state_broadcaster/
↪JointStateBroadcaster] inactive
threedofbot_pid_gain_controller[forward_command_controller/
↪ForwardCommandController] inactive
threedofbot_position_controller[forward_command_controller/
↪ForwardCommandController] inactive
```

2. Activate `RRBotWithSensor` hardware component. Use either the ros2controlcli

```
ros2 control set_hardware_component_state RRBotSystemWithSensor active
```

or call the service manually

```
ros2 service call /controller_manager/set_hardware_component_state␣
↪controller_manager_msgs/srv/SetHardwareComponentState "
name: RRBotSystemWithSensor
target_state:
  id: 0
  label: active"
```

Then activate its position controller via

```
ros2 control switch_controllers --activate rrbot_with_sensor_position_
↪controller
```

Scenario state:

- right robot is moving
- left robot is moving
- middle robot is “broken”

Hardware status: `RRBotSystemWithSensor` is now in active state

```
$ ros2 control list_hardware_components
...
Hardware Component 2
    name: RRBotSystemWithSensor
    type: system
    plugin name: ros2_control_demo_example_4/RRBotSystemWithSensorHardware
    state: id=3 label=active
    command interfaces
            rrbot_with_sensor_joint1/position [available] [claimed]
            rrbot_with_sensor_joint2/position [available] [claimed]
...
```

Controllers status: `rrbot_with_sensor_position_controller` is now in active state:

```
$ ros2 control list_controllers
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
rrbot_external_fts_broadcaster[force_torque_sensor_broadcaster/
↪ForceTorqueSensorBroadcaster] active
rrbot_joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]␣
↪active
rrbot_position_controller[forward_command_controller/
↪ForwardCommandController] active
rrbot_with_sensor_fts_broadcaster[force_torque_sensor_broadcaster/
↪ForceTorqueSensorBroadcaster] active
rrbot_with_sensor_joint_state_broadcaster[joint_state_broadcaster/
↪JointStateBroadcaster] active
rrbot_with_sensor_position_controller[forward_command_controller/
↪ForwardCommandController] active
threedofbot_joint_state_broadcaster[joint_state_broadcaster/
↪JointStateBroadcaster] inactive
threedofbot_pid_gain_controller[forward_command_controller/
↪ForwardCommandController] inactive
threedofbot_position_controller[forward_command_controller/
↪ForwardCommandController] inactive
```

3. Configure `FakeThreeDofBot` and its joint state broadcaster. Call

```
ros2 control set_hardware_component_state FakeThreeDofBot inactive
ros2 control switch_controllers --activate threedofbot_joint_state_
↪broadcaster
```

Scenario state:

- right robot is moving
- left robot is moving
- middle robot is still "broken"

Hardware status: `FakeThreeDofBot` is in inactive state.

```
$ ros2 control list_hardware_components
Hardware Component 1
      name: FakeThreeDofBot
      type: system
      plugin name: mock_components/GenericSystem
      state: id=2 label=inactive
```

(continues on next page)

(continued from previous page)

```
      command interfaces
              threedofbot_joint1/position [available] [unclaimed]
              threedofbot_joint1/pid_gain [available] [unclaimed]
              threedofbot_joint2/position [available] [unclaimed]
              threedofbot_joint2/pid_gain [available] [unclaimed]
              threedofbot_joint3/position [available] [unclaimed]
              threedofbot_joint3/pid_gain [available] [unclaimed]
...
```

Controllers status, `threedofbot_joint_state_broadcaster` is in active state now:

```
$ ros2 control list_controllers
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
rrbot_external_fts_broadcaster[force_torque_sensor_broadcaster/
↪ForceTorqueSensorBroadcaster] active
rrbot_joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]␣
↪active
rrbot_position_controller[forward_command_controller/
↪ForwardCommandController] active
rrbot_with_sensor_fts_broadcaster[force_torque_sensor_broadcaster/
↪ForceTorqueSensorBroadcaster] active
rrbot_with_sensor_joint_state_broadcaster[joint_state_broadcaster/
↪JointStateBroadcaster] active
rrbot_with_sensor_position_controller[forward_command_controller/
↪ForwardCommandController] active
threedofbot_joint_state_broadcaster[joint_state_broadcaster/
↪JointStateBroadcaster] active
threedofbot_pid_gain_controller[forward_command_controller/
↪ForwardCommandController] inactive
threedofbot_position_controller[forward_command_controller/
↪ForwardCommandController] inactive
```

4. Restart global joint state broadcaster to broadcast all available states from the framework. First check output to have comparison:

```
ros2 topic echo /joint_states --once
```

Restart:

```
ros2 control switch_controllers --deactivate joint_state_broadcaster
ros2 control switch_controllers --activate joint_state_broadcaster
```

Check output for comparison, now the joint_states of `threedofbot` and `rrbot_with_sensor` are broadcasted, too.

```
ros2 topic echo /joint_states --once
```

Scenario state (everything is broken during `joint_state_broadcaster` restart):

- right robot is moving
- left robot is moving
- middle robot is now still “standing”

5. Activate `FakeThreeDofBot` and its controllers. Call

```
ros2 control set_hardware_component_state FakeThreeDofBot active
ros2 control switch_controllers --activate threedofbot_pid_gain_controller␣
↪threedofbot_position_controller
```

Scenario state:

- right robot is moving
- left robot is moving
- middle robot is moving

Hardware status: `FakeThreeDofBot` is in active state.

```
$ ros2 control list_hardware_components
Hardware Component 1
    name: FakeThreeDofBot
    type: system
    plugin name: mock_components/GenericSystem
    state: id=3 label=active
    command interfaces
            threedofbot_joint1/position [available] [claimed]
            threedofbot_joint1/pid_gain [available] [claimed]
            threedofbot_joint2/position [available] [claimed]
            threedofbot_joint2/pid_gain [available] [claimed]
            threedofbot_joint3/position [available] [claimed]
            threedofbot_joint3/pid_gain [available] [claimed]
...
```

Controllers status (all active now):

```
$ ros2 control list_controllers
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
rrbot_external_fts_broadcaster[force_torque_sensor_broadcaster/
↪ForceTorqueSensorBroadcaster] active
rrbot_joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]␣
↪active
rrbot_position_controller[forward_command_controller/
↪ForwardCommandController] active
rrbot_with_sensor_fts_broadcaster[force_torque_sensor_broadcaster/
↪ForceTorqueSensorBroadcaster] active
rrbot_with_sensor_joint_state_broadcaster[joint_state_broadcaster/
↪JointStateBroadcaster] active
rrbot_with_sensor_position_controller[forward_command_controller/
↪ForwardCommandController] active
threedofbot_joint_state_broadcaster[joint_state_broadcaster/
↪JointStateBroadcaster] active
threedofbot_pid_gain_controller[forward_command_controller/
↪ForwardCommandController] active
threedofbot_position_controller[forward_command_controller/
↪ForwardCommandController] active
```

6. Deactivate `RRBotSystemPositionOnly` and its position controller (first). Call

```
ros2 control switch_controllers --deactivate rrbot_position_controller
ros2 control set_hardware_component_state RRBotSystemPositionOnly inactive
```

Scenario state:

- right robot is now “standing” at the last position

- left robot is moving
- middle robot is moving

Hardware status: `RRBotSystemPositionOnly` is in inactive state.

```
$ ros2 control list_hardware_components
...
Hardware Component 4
    name: RRBotSystemPositionOnly
    type: system
    plugin name: ros2_control_demo_example_5/RRBotSystemPositionOnlyHardware
    state: id=2 label=inactive
    command interfaces
            rrbot_joint1/position [available] [unclaimed]
            rrbot_joint2/position [available] [unclaimed]
...
```

Controllers status: `rrbot_position_controller` is now in inactive state

```
$ ros2 control list_controllers
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
rrbot_external_fts_broadcaster[force_torque_sensor_broadcaster/
↪ForceTorqueSensorBroadcaster] active
rrbot_joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]␣
↪active
rrbot_position_controller[forward_command_controller/
↪ForwardCommandController] inactive
rrbot_with_sensor_fts_broadcaster[force_torque_sensor_broadcaster/
↪ForceTorqueSensorBroadcaster] active
rrbot_with_sensor_joint_state_broadcaster[joint_state_broadcaster/
↪JointStateBroadcaster] active
rrbot_with_sensor_position_controller[forward_command_controller/
↪ForwardCommandController] active
threedofbot_joint_state_broadcaster[joint_state_broadcaster/
↪JointStateBroadcaster] active
threedofbot_pid_gain_controller[forward_command_controller/
↪ForwardCommandController] active
threedofbot_position_controller[forward_command_controller/
↪ForwardCommandController] active
```

7. Set `RRBotSystemPositionOnly` in unconfigured state, and deactivate its joint state broadcaster. Also restart global joint state broadcaster. Call

```
ros2 control switch_controllers --deactivate rrbot_joint_state_broadcaster␣
↪joint_state_broadcaster
ros2 control set_hardware_component_state RRBotSystemPositionOnly␣
↪unconfigured
ros2 control switch_controllers --activate joint_state_broadcaster
```

Scenario state (everything is broken during `joint_state_broadcaster` restart):

- right robot is standing still.
- left robot is moving
- middle robot is moving

Controllers status:

```
$ ros2 control list_controllers
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
rrbot_external_fts_broadcaster[force_torque_sensor_broadcaster/
↪ForceTorqueSensorBroadcaster] active
rrbot_joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]␣
↪inactive
rrbot_position_controller[forward_command_controller/
↪ForwardCommandController] inactive
rrbot_with_sensor_fts_broadcaster[force_torque_sensor_broadcaster/
↪ForceTorqueSensorBroadcaster] active
rrbot_with_sensor_joint_state_broadcaster[joint_state_broadcaster/
↪JointStateBroadcaster] active
rrbot_with_sensor_position_controller[forward_command_controller/
↪ForwardCommandController] active
threedofbot_joint_state_broadcaster[joint_state_broadcaster/
↪JointStateBroadcaster] active
threedofbot_pid_gain_controller[forward_command_controller/
↪ForwardCommandController] active
threedofbot_position_controller[forward_command_controller/
↪ForwardCommandController] active
```

Files used for this demos

- Launch file: three_robots.launch.py
- Controllers yaml: three_robots_controllers.yaml
- URDF file: three_robots.urdf.xacro
  - Description: threedofbot_description.urdf.xacro
  - `ros2_control` tag: + threedofbot.ros2_control.xacro + rrbot_system_position_only.ros2_control.xacro + rrbot_system_with_sensor.ros2_control.xacro
- RViz configuration: three_robots.rviz

Controllers from this demo

- `Joint State Broadcaster` (ros2_controllers repository): doc
- `Forward Command Controller` (ros2_controllers repository): doc

4.6.14 Example 14: Modular robot with actuators not providing states

The example shows how to implement robot hardware with separate communication to each actuator as well as separate sensors for position feedback:

- The communication is done on actuator level using proprietary or standardized API (e.g., canopen_402, Modbus, RS232, RS485).
- Data for all actuators and sensors is exchanged separately from each other
- Examples: Arduino-based-robots, custom robots

This is implemented with hardware interfaces of type `hardware_interface::ActuatorInterface` and `hardware_interface::SensorInterface`.

Note: The commands below are given for a local installation of this repository and its dependencies as well as for running them from a docker container. For more information on the docker usage see *Using Docker*.

Tutorial steps

1. To check that *RRBot* descriptions are working properly use following launch commands

```
ros2 launch ros2_control_demo_example_14 view_robot.launch.py
```

Note: Getting the following output in terminal is OK: `Warning: Invalid frame ID "odom" passed to canTransform argument target_frame - frame does not exist`. This happens because `joint_state_publisher_gui` node need some time to start. The `joint_state_publisher_gui` provides a GUI to generate a random configuration for rrbot. It is immediately displayed in *RViz*.

2. To start *RRBot* example open a terminal, source your ROS2-workspace and execute its launch file with

```
ros2 launch ros2_control_demo_example_14 rrbot_modular_actuators_without_feedback_
↪sensors_for_position_feedback.launch.py
```

The launch file loads and starts the robot hardware, controllers and opens *RViz*. In starting terminal you will see a lot of output from the hardware implementation showing its internal states. This is only of exemplary purposes and should be avoided as much as possible in a hardware interface implementation.

If you can see two orange and one yellow rectangle in in *RViz* everything has started properly. Still, to be sure, let's introspect the control system before moving *RRBot*.

3. Check if the hardware interface loaded properly, by opening another terminal and executing

```
ros2 control list_hardware_interfaces
```

```
command interfaces
  joint1/position [available] [claimed]
  joint2/position [available] [claimed]
state interfaces
  joint1/position
  joint2/position
```

Marker `[claimed]` by command interfaces means that a controller has access to command *RRBot*.

Now, let's introspect the hardware components with

```
ros2 control list_hardware_components
```

There are four hardware components, one for each actuator and one for each sensor:

```
Hardware Component 1
        name: RRBotModularJoint2
        type: actuator
        plugin name: ros2_control_demo_example_14/RRBotActuatorWithoutFeedback
        state: id=3 label=active
        command interfaces
                joint2/velocity [available] [claimed]
Hardware Component 2
```

(continues on next page)

(continued from previous page)

```
        name: RRBotModularJoint1
        type: actuator
        plugin name: ros2_control_demo_example_14/RRBotActuatorWithoutFeedback
        state: id=3 label=active
        command interfaces
                joint1/velocity [available] [claimed]
Hardware Component 3
        name: RRBotModularPositionSensorJoint2
        type: sensor
        plugin name: ros2_control_demo_example_14/RRBotSensorPositionFeedback
        state: id=3 label=active
        command interfaces
Hardware Component 4
        name: RRBotModularPositionSensorJoint1
        type: sensor
        plugin name: ros2_control_demo_example_14/RRBotSensorPositionFeedback
        state: id=3 label=active
        command interfaces
```

4. Check if controllers are running

```
ros2 control list_controllers
```

```
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
forward_velocity_controller[forward_command_controller/ForwardCommandController]␣
↪active
```

5. If you get output from above you can send commands to *Forward Command Controller*:

```
ros2 topic pub /forward_velocity_controller/commands std_msgs/msg/
↪Float64MultiArray "data:
- 5
- 5"
```

You should now see orange and yellow blocks moving in *RViz*. Also, you should see changing states in the terminal where launch file is started, e.g.

```
[ros2_control_node-1] [INFO] [1728858168.276013464] [controller_manager.resource_
↪manager.hardware_component.actuator.RRBotModularJoint1]: Writing...
[ros2_control_node-1] Writing command: 5.00
[ros2_control_node-1] Sending data command: 5
[ros2_control_node-1]
[ros2_control_node-1] [INFO] [1728858168.776052116] [controller_manager.resource_
↪manager.hardware_component.actuator.RRBotModularJoint2]: Writing...
[ros2_control_node-1] Writing command: 5.00
[ros2_control_node-1] Sending data command: 5
[ros2_control_node-1]
[ros2_control_node-1] [INFO] [1728858169.275878132] [controller_manager.resource_
↪manager.hardware_component.sensor.RRBotModularPositionSensorJoint1]: Reading...
[ros2_control_node-1] Got measured velocity 5.00
[ros2_control_node-1] Got state 0.34 for joint 'joint1'
[ros2_control_node-1]
[ros2_control_node-1] [INFO] [1728858169.775863217] [controller_manager.resource_
↪manager.hardware_component.sensor.RRBotModularPositionSensorJoint2]: Reading...
[ros2_control_node-1] Got measured velocity 5.00
[ros2_control_node-1] Got state 0.29 for joint 'joint2'
```

(continues on next page)

(continued from previous page)

```
[ros2_control_node-1]
```

Files used for this demos

- Launch file: rrbot_modular_actuators_without_feedback_sensors_for_position_feedback.launch.py
- Controllers yaml: rrbot_modular_actuators_without_feedback_sensors_for_position_feedback.yaml
- URDF: rrbot_modular_actuators_without_feedback_sensors_for_position_feedback.urdf.xacro
  - Description: rrbot_description.urdf.xacro
  - `ros2_control` URDF tag: rrbot_modular_actuators_without_feedback_sensors_for_position_feedback.ros2_control.xacro
- RViz configuration: rrbot.rviz
- Hardware interface plugins:
  - rrbot_actuator_without_feedback.cpp
  - rrbot_sensor_for_position_feedback.cpp

Controllers from this demo

- `Joint State Broadcaster` (ros2_controllers repository): *doc*
- `Forward Command Controller` (ros2_controllers repository): *doc*

4.6.15 Example 15: Using multiple controller managers

This example shows how to integrate multiple robots under different controller manager instances.

Note: The commands below are given for a local installation of this repository and its dependencies as well as for running them from a docker container. For more information on the docker usage see *Using Docker*.

Scenario: Using ros2_control within a local namespace

Note: When running `ros2 control` CLI commands you have to use additional parameter with exact controller manager node name, i.e., `-c /rrbot/controller_manager`.

Launch the example with

```
ros2 launch ros2_control_demo_example_15 rrbot_namespace.launch.py
```

- Command interfaces:
  - joint1/position
  - joint2/position
- State interfaces:
  - joint1/position

- joint2/position

Available controllers: (nodes under namespace "/rrbot")

```
$ ros2 control list_controllers -c /rrbot/controller_manager
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
forward_position_controller[forward_command_controller/ForwardCommandController]␣
↪active
position_trajectory_controller[joint_trajectory_controller/JointTrajectoryController]␣
↪inactive
```

Commanding the robot using a `ForwardCommandController` (name: `/rrbot/forward_position_controller`)

```
ros2 launch ros2_control_demo_example_15 test_forward_position_controller.launch.py␣
↪publisher_config:=rrbot_namespace_forward_position_publisher.yaml
```

Abort the command and switch controller to use `JointTrajectoryController` (name: `/rrbot/position_trajectory_controller`):

```
ros2 control switch_controllers -c /rrbot/controller_manager --deactivate forward_
↪position_controller --activate position_trajectory_controller
```

Commanding the robot using `JointTrajectoryController` (name: `/rrbot/position_trajectory_controller`)

```
ros2 launch ros2_control_demo_example_15 test_joint_trajectory_controller.launch.py␣
↪publisher_config:=rrbot_namespace_joint_trajectory_publisher.yaml
```

Files used for this demo:

- Launch file: rrbot_namespace.launch.py
- Controllers yaml: rrbot_namespace_controllers.yaml
- URDF file: rrbot.urdf.xacro
  - Description: rrbot_description.urdf.xacro
  - `ros2_control` tag: rrbot.ros2_control.xacro
- RViz configuration: rrbot.rviz
- Test nodes goals configuration:
  - rrbot_forward_position_publisher
  - rrbot_joint_trajectory_publisher
- Hardware interface plugin: rrbot.cpp

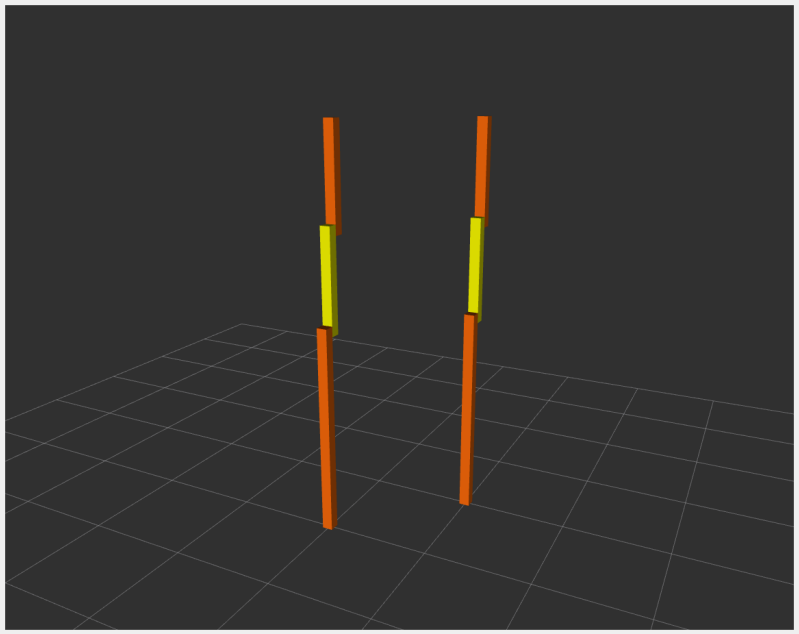
Scenario: Using multiple controller managers on the same machine

Note: When running `ros2 control` CLI commands you have to use additional parameter with exact controller manager node name, e.g., `-c /rrbot_1/controller_manager` or `-c /rrbot_2/controller_manager`.

Launch the example with

```
ros2 launch ros2_control_demo_example_15 multi_controller_manager_example_two_rrbots.
↪launch.py
```

You should see two robots in RViz:



`rrbot_1` namespace:

- Command interfaces:
  - rrbot_1_joint1/position
  - rrbot_1_joint2/position
- State interfaces:
  - rrbot_1_joint1/position
  - rrbot_1_joint2/position

`rrbot_2` namespace:

- Command interfaces:
  - rrbot_2_joint1/position
  - rrbot_2_joint2/position
- State interfaces:
  - rrbot_2_joint1/position
  - rrbot_2_joint2/position

Available controllers (nodes under namespace `/rrbot_1` and `/rrbot_2`):

```
$ ros2 control list_controllers -c /rrbot_1/controller_manager
position_trajectory_controller[joint_trajectory_controller/JointTrajectoryController]␣
↪inactive
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
forward_position_controller[forward_command_controller/ForwardCommandController]␣
↪active

$ ros2 control list_controllers -c /rrbot_2/controller_manager
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
position_trajectory_controller[joint_trajectory_controller/JointTrajectoryController]␣
↪inactive
forward_position_controller[forward_command_controller/ForwardCommandController]␣
↪active
```

Commanding the robots using the `forward_position_controller` (of type `ForwardCommandController`)

```
ros2 launch ros2_control_demo_example_15 test_multi_controller_manager_forward_
↪position_controller.launch.py
```

Switch controller to use the `position_trajectory_controller` (of type `JointTrajectoryController`) - alternatively start main launch file with argument `robot_controller:=position_trajectory_controller`:

```
ros2 control switch_controllers -c /rrbot_1/controller_manager --deactivate forward_
↪position_controller --activate position_trajectory_controller
ros2 control switch_controllers -c /rrbot_2/controller_manager --deactivate forward_
↪position_controller --activate position_trajectory_controller
```

Commanding the robots using the now activated `position_trajectory_controller`:

```
ros2 launch ros2_control_demo_example_15 test_multi_controller_manager_joint_
↪trajectory_controller.launch.py
```

Files used for this demo:

- Launch file: multi_controller_manager_example_two_rrbots.launch.py
- Controllers yaml: - multi_controller_manager_rrbot_generic_controllers.yaml
- URDF file: rrbot.urdf.xacro
  - Description: rrbot_description.urdf.xacro
  - `ros2_control` tag: rrbot.ros2_control.xacro
- RViz configuration: rrbot.rviz
- Test nodes goals configuration:
  - rrbot_forward_position_publisher
  - rrbot_joint_trajectory_publisher
- Hardware interface plugin: rrbot.cpp

Controllers from this demo

- `Joint State Broadcaster` (ros2_controllers repository): doc
- `Forward Command Controller` (ros2_controllers repository): doc
- `Joint Trajectory Controller` (ros2_controllers repository): doc

4.6.16 DiffBot with Chained Controllers

This example shows how to create chained controllers using diff_drive_controller and pid_controllers to control a differential drive robot. It extends *example_2*. If you haven't already, you can find the instructions for *example_2* in *DiffBot*. It is recommended to follow the steps given in that tutorial first before proceeding with this one.

This example demonstrates controller chaining as described in *Controller Chaining / Cascade Control*. The control chain flows from the diff_drive_controller through two PID controllers to the DiffBot hardware. The diff_drive_controller converts desired robot twist into wheel velocity commands, which are then processed by the PID controllers to directly control the wheel velocities. Additionally, this example shows how to enable the feedforward mode for the PID controllers.

Furthermore, this example shows how to use plotjuggler to visualize the controller states.

The *DiffBot* URDF files can be found in `description/urdf` folder.

Note: The commands below are given for a local installation of this repository and its dependencies as well as for running them from a docker container. For more information on the docker usage see *Using Docker*.

Tutorial steps

1. To start *DiffBot* example open a terminal, source your ROS2-workspace and execute its launch file with

   ```
   ros2 launch ros2_control_demo_example_16 diffbot.launch.py
   ```

   The launch file loads and starts the robot hardware, controllers and opens *RViz*. In the starting terminal you will see a lot of output from the hardware implementation showing its internal states. This excessive printing is only added for demonstration. In general, printing to the terminal should be avoided as much as possible in a hardware interface implementation.

   If you can see an orange box in *RViz* everything has started properly. Let's introspect the control system before moving *DiffBot*.

2. Check controllers

   ```
   ros2 control list_controllers
   ```

   You should get

   ```
   joint_state_broadcaster          joint_state_broadcaster/
   ↪JointStateBroadcaster  active
   diffbot_base_controller          diff_drive_controller/DiffDriveController  ␣
   ↪    active
   pid_controller_right_wheel_joint pid_controller/PidController              ␣
   ↪    active
   pid_controller_left_wheel_joint  pid_controller/PidController              ␣
   ↪    active
   ```

3. Check the hardware interface loaded by opening another terminal and executing

```
ros2 control list_hardware_interfaces
```

You should get

```
command interfaces
  diffbot_base_controller/angular/velocity [unavailable] [unclaimed]
  diffbot_base_controller/linear/velocity [unavailable] [unclaimed]
  left_wheel_joint/velocity [available] [claimed]
  pid_controller_left_wheel_joint/left_wheel_joint/velocity [available]␣
↪[unclaimed]
  pid_controller_right_wheel_joint/right_wheel_joint/velocity [available]␣
↪[unclaimed]
  right_wheel_joint/velocity [available] [claimed]
state interfaces
  left_wheel_joint/position
  left_wheel_joint/velocity
  pid_controller_left_wheel_joint/left_wheel_joint/velocity
  pid_controller_right_wheel_joint/right_wheel_joint/velocity
  right_wheel_joint/position
  right_wheel_joint/velocity
```

The `[claimed]` marker on command interfaces means that a controller has access to command *DiffBot*. There are two `[claimed]` interfaces from pid_controller, one for left wheel and one for right wheel. These interfaces are referenced by diff_drive_controller. By referencing them, diff_drive_controller can send commands to these interfaces. If you see these, we've successfully chained the controllers.

There are also two `[unclaimed]` interfaces from diff_drive_controller, one for angular velocity and one for linear velocity. These are provided by the diff_drive_controller because it is chainable. You can ignore them since we don't use them in this example.

4. To see the pid_controller in action, let's subscribe to the controller_state topic, e.g. pid_controller_left_wheel_joint/controller_state topic.

```
ros2 topic echo /pid_controller_left_wheel_joint/controller_state
```

5. Now we are ready to send a command to move the robot. Send a command to *Diff Drive Controller* by opening another terminal and executing

```
ros2 topic pub --rate 10 /cmd_vel geometry_msgs/msg/TwistStamped "
  header: auto
  twist:
    linear:
      x: 0.7
      y: 0.0
      z: 0.0
    angular:
      x: 0.0
      y: 0.0
      z: 1.0"
```

You should now see robot is moving in circles in *RViz*.

6. In the terminal where launch file is started, you should see the commands being sent to the wheels and how they are gradually stabilizing to the target velocity similar to following output.

```
[ros2_control_node-1] [INFO] [1739860498.634431841] [controller_manager.
↪resource_manager.hardware_component.system.DiffBot]: Reading states:
[ros2_control_node-1]        position 5.78 and velocity 27.52 for 'right_
```

(continues on next page)

(continued from previous page)

```
↪wheel_joint/position'!
[ros2_control_node-1]         position 5.23 and velocity 24.90 for 'left_
↪wheel_joint/position'!
[ros2_control_node-1] [INFO] [1739860498.634800954] [controller_manager.
↪resource_manager.hardware_component.system.DiffBot]: Writing commands:
[ros2_control_node-1]        command 35.55 for 'right_wheel_joint/velocity'!
[ros2_control_node-1]        command 32.16 for 'left_wheel_joint/velocity'!
[ros2_control_node-1] [INFO] [1739860499.234393780] [controller_manager.
↪resource_manager.hardware_component.system.DiffBot]: Reading states:
[ros2_control_node-1]         position 36.99 and velocity 60.61 for 'right_
↪wheel_joint/position'!
[ros2_control_node-1]         position 33.19 and velocity 53.17 for 'left_
↪wheel_joint/position'!
[ros2_control_node-1] [INFO] [1739860499.234812092] [controller_manager.
↪resource_manager.hardware_component.system.DiffBot]: Writing commands:
[ros2_control_node-1]        command 60.53 for 'right_wheel_joint/velocity'!
[ros2_control_node-1]        command 52.27 for 'left_wheel_joint/velocity'!
```

7. Let's go back to the terminal where we subscribed to the controller_state topic and see the changing states.

```
---
header:
  stamp:
    sec: 1739860821
    nanosec: 314185285
  frame_id: ''
dof_states:
- name: left_wheel_joint
  reference: 18.967273133333336
  feedback: 13.294059091678035
  feedback_dot: 0.0
  error: 5.673214041655301
  error_dot: 0.0
  time_step: 0.100077098
  output: 24.857442270936623
---
header:
  stamp:
    sec: 1739860821
    nanosec: 414126639
  frame_id: ''
dof_states:
- name: left_wheel_joint
  reference: 25.296198783333335
  feedback: 19.8859538167493
  feedback_dot: 0.0
  error: 5.410244966584035
  error_dot: 0.0
  time_step: 0.099941354
  output: 32.090205119481226
```

Visualize the convergence of DiffBot's wheel velocities and commands

In the section below, we will use *plotjuggler* to observe the convergence of DiffBot's wheel velocities and commands from PID controllers.

plotjuggler is an open-source data visualization tool and widely embraced by ROS2 community. If you don't have it installed, you can find the instructions from plotjuggler website.

Before we proceed, we stop all previous steps from terminal and start from the beginning.

1. To start *DiffBot* example open a terminal, source your ROS2-workspace and execute its launch file with

   ```
   ros2 launch ros2_control_demo_example_16 diffbot.launch.py
   ```

   Like before, if you can see an orange box in *RViz*, everything has started properly.

2. To start the plotjuggler with a provided layout file(plotjuggler.xml), open another terminal and run following command.

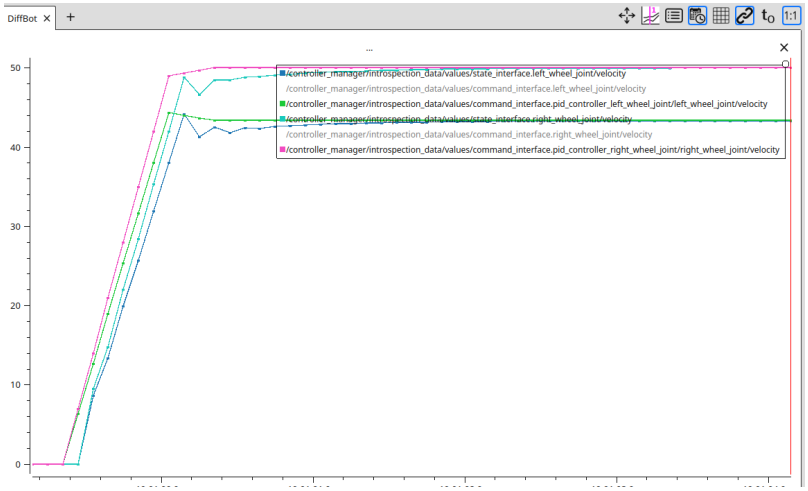
   ```
   ros2 run plotjuggler plotjuggler --layout $(ros2 pkg prefix ros2_control_
   ↪demo_example_16 --share)/config/plotjuggler.xml
   ```

   After this, you will see a few dialogs popping up. For example:

   ```
   Start the previously used streaming plugin?

   ROS2 Topic Subscriber
   ```

   Click 'Yes' for the first dialog and 'OK" to the following two dialogs, then you will see the plotjuggler window.

3. From the plotjuggler, you can see the controllers' states and commands being plotted, similar to following figure. From the figure, the DiffBot's wheel velocities and commands from PID controllers are converged to the target velocity fairly quickly.



4. Change the `gains` in the `diffbot_chained_controllers.yaml` file with some different values, repeat above steps and observe its effect to the pid_controller commands. For example, to change the `feedforward_gain` of the right wheel to 0.50, you can use the following command:

   ```
   ros2 param set /pid_controller_right_wheel_joint gains.right_wheel_joint.
   ↪feedforward_gain 0.50
   ```

Files used for this demo

- Launch file: diffbot.launch.py
- Controllers yaml: diffbot_chained_controllers.yaml
- URDF file: diffbot.urdf.xacro
  - Description: diffbot_description.urdf.xacro
  - `ros2_control` tag: diffbot.ros2_control.xacro
- RViz configuration: diffbot.rviz
- Hardware interface plugin: diffbot_system.cpp

Controllers from this demo

- `Joint State Broadcaster` (ros2_controllers repository): *doc*
- `Diff Drive Controller` (ros2_controllers repository): *doc*
- `pid_controller` (ros2_controllers repository): *doc*

4.6.17 Example 17: RRBot with Hardware Component that publishes diagnostics and status messages

This example shows how to publish diagnostics and status messages from a hardware component using ROS 2 features available within the `ros2_control` framework.

It is essentially the same as Example 1, but with a modified hardware interface plugin that demonstrates three methods for publishing status information:

1. Using the standard `diagnostic_updater` on the default node to publish to the `/diagnostics` topic.
2. Using the Controller Manager's Executor to add a custom ROS 2 node for publishing to a separate, non-standard topic.
3. Using the framework managed default publisher, which publishes with a `HardwareStatus` message, this is the recommended way when you want to publish structured messages.

Note: Structured messages mentioned above are in reference to hardware_status roadmap

See *Implementation Details of the Diagnostic Publisher* and *Implementation Details of the Hardware Status Publisher* for more information.

For *example_17*, the hardware interface plugin is implemented having only one interface.

- The communication is done using proprietary API to communicate with the robot control box.
- Data for all joints is exchanged at once.
- Examples: KUKA RSI

The *RRBot* URDF files can be found in the `description/urdf` folder.

Note: The commands below are given for a local installation of this repository and its dependencies as well as for running them from a docker container. For more information on the docker usage see *Using Docker*.

Tutorial steps

Follow the same basic steps as in Example 1. You can find the details here: *Example 1: RRBot System Position Only*.

This tutorial differs by including a hardware interface that publishes diagnostics using two different mechanisms:

1. A **default node** is automatically provided to the hardware component. We use the standard `diagnostic_updater` library on this node to publish structured diagnostic data.
2. A **custom ROS 2 node** is created inside the hardware interface and added to the executor provided by the controller manager. This demonstrates a more manual approach useful for non-diagnostic topics.

as well as

3. A **default publisher** is automatically created through steps detailed in *Implementation Details of the Hardware Status Publisher*.

The nodes and topics:

- Default node (via `diagnostic_updater`):
  - Is named after the hardware component (e.g., `/RRBot`).
  - Publishes periodically to the standard `/diagnostics` topic.
  - Uses message type `diagnostic_msgs/msg/DiagnosticArray`.
- Custom node:
  - Is named `<hardware_name>_custom_node` (e.g., `/rrbot_custom_node`).
  - Publishes on the topic `/rrbot_custom_status`.
  - Uses message type `std_msgs/msg/String`.
  - Sends a message every 2 seconds.
- Default Publisher:
  - Publishes on the topic `/rrbot/hardware_status`.
  - Uses message type `control_msgs/msg/HardwareStatus`.
  - Sends a message at rate specified by `status_publish_rate` parameter in ros2_control tag.

To check that the nodes are running and diagnostics are published correctly:

Local

```
# List available nodes
ros2 node list

# You should see something like:
# /rrbot           (the default node)
# /rrbot_custom_node (the custom node)
# /controller_manager
# /robot_state_publisher

# List topics and confirm diagnostics topics are available
ros2 topic list

# Confirm message type of the diagnostics topic
ros2 topic info /diagnostics
# Should show: diagnostic_msgs/msg/DiagnosticArray
```

(continues on next page)

(continued from previous page)

```
# Confirm message type of the custom status topic
ros2 topic info /rrbot_custom_status
# Should show: std_msgs/msg/String

# Confirm message type of the default publisher topic
ros2 topic info /rrbot/hardware_status
# Should show: control_msgs/msg/HardwareStatus

# Echo the raw messages published by the default node
ros2 topic echo /diagnostics

# Echo the messages published by the custom node
ros2 topic echo /rrbot_custom_status

# Echo the messages published by default publisher
ros2 topic echo /rrbot/hardware_status
```

Docker

Enter the running container shell first:

```
docker exec -it ros2_control_demos ./entrypoint.sh bash
```

Then run the same commands inside the container:

```
ros2 node list
ros2 topic info /diagnostics
ros2 topic info /rrbot_custom_status
ros2 topic info /rrbot/hardware_status
ros2 topic echo /diagnostics
ros2 topic echo /rrbot_custom_status
ros2 topic echo /rrbot/hardware_status
```

The echoed messages should look similar to:

```
# From /diagnostics (showing the default node's output)
header:
  stamp:
    sec: 1678886401
    nanosec: 123456789
status:
- level: 0
  name: "RRBot"
  message: "Hardware is OK"
  hardware_id: ""
  values: []
```

```
# From /rrbot_custom_status (showing the custom node's output)
data: RRBot 'RRBot' custom node is alive at 1751087597.146549
```

```
# From /rrbot/hardware_status (showing the default publisher's output)
 header:
   stamp:
     sec: 1007
     nanosec: 560488417
   frame_id: ''
```

(continues on next page)

(continued from previous page)

```
hardware_id: RRBot
hardware_device_states:
- header:
    stamp:
      sec: 1761387923
      nanosec: 368924291
    frame_id: joint1
  device_id: joint1
  hardware_status:
  - health_status: 1
    error_domain: []
    operational_mode: 2
    power_state: 3
    connectivity_status: 0
    manufacturer: ''
    model: ''
    firmware_version: ''
    state_details:
    - key: position_state
      value: '0.000000'
  canopen_states: []
  ethercat_states: []
  vda5050_states: []
- header:
    stamp:
      sec: 1761387923
      nanosec: 368970350
    frame_id: joint2
  device_id: joint2
  hardware_status:
  - health_status: 1
    error_domain: []
    operational_mode: 2
    power_state: 3
    connectivity_status: 0
    manufacturer: ''
    model: ''
    firmware_version: ''
    state_details:
    - key: position_state
      value: '0.000000'
  canopen_states: []
  ethercat_states: []
  vda5050_states: []
```

Note: The custom diagnostics node and its timer are created only if the executor is successfully passed to the hardware component. If you don't see the topic or node, ensure the hardware plugin is correctly implemented and that the controller manager is providing an executor.

Implementation Details of the Hardware Status Publisher

1. Using the Framework-Managed Status Publisher (Recommended for HardwareStatus Messages)

The `ros2_control` framework provides a built-in, real-time safe mechanism for publishing standardized hardware status via the `control_msgs/msg/HardwareStatus` message. This is the simplest and most robust way to provide detailed status messages. It is enabled by implementing two virtual methods in your hardware component.

a. **Override init_hardware_status_message**: This non-realtime method is called once during initialization. You must override it to define the **static structure** of your status message. This includes setting the `hardware_id`, resizing the `hardware_device_states` vector, and for each device, resizing its specific status vectors (e.g., `generic_hardware_status`) and populating static fields like `device_id`.

```
// In rrbot.hpp, add the override declaration:
hardware_interface::CallbackReturn init_hardware_status_message(
  control_msgs::msg::HardwareStatus & msg_template) override;

// In rrbot.cpp
hardware_interface::CallbackReturn RRBotSystemPositionOnlyHardware::init_
↪hardware_status_message(
  control_msgs::msg::HardwareStatus & msg)
{
  msg.hardware_id = get_hardware_info().name;
  msg.hardware_device_states.resize(get_hardware_info().joints.size());

  for (size_t i = 0; i < get_hardware_info().joints.size(); ++i)
  {
    msg.hardware_device_states[i].device_id = get_hardware_info().
↪joints[i].name;
    // This example uses one generic status per joint
    msg.hardware_device_states[i].generic_hardware_status.resize(1);
  }
  return hardware_interface::CallbackReturn::SUCCESS;
}
```

b. **Override update_hardware_status_message**: This real-time safe method is called from the framework's internal timer. You override it to **fill in the dynamic values** of the pre-structured message, typically by copying your hardware's internal state variables (updated in your `read()` method) into the fields of the message.

```
// In rrbot.hpp, add the override declaration:
hardware_interface::return_type update_hardware_status_message(
  control_msgs::msg::HardwareStatus & msg) override;

// In rrbot.cpp
hardware_interface::return_type RRBotSystemPositionOnlyHardware::update_
↪hardware_status_message(
  control_msgs::msg::HardwareStatus & msg)
{
  for (size_t i = 0; i < get_hardware_info().joints.size(); ++i)
  {
    auto & generic_status = msg.hardware_device_states[i].generic_
↪hardware_status;
    // Example: Map internal state to a standard status field
    double position = get_state(get_hardware_info().joints[i].name + "/
↪position");
```

(continues on next page)

(continued from previous page)

```
    if (std::abs(position) > 2.5) // Arbitrary warning threshold
    {
      generic_status.health_status = control_
↪msgs::msg::GenericState::HEALTH_WARNING;
    }
    else
    {
      generic_status.health_status = control_
↪msgs::msg::GenericState::HEALTH_OK;
    }
    generic_status.operational_mode = control_
↪msgs::msg::GenericState::MODE_AUTO;
    generic_status.power_state = control_msgs::msg::GenericState::POWER_
↪ON;
  }
  return hardware_interface::return_type::OK;
}
```

c. **Enable in URDF**: To activate the publisher, add the `status_publish_rate` parameter to your `<hardware>` tag in the URDF. Setting it to 0.0 disabled the feature.

```
<ros2_control name="RRBotSystemPositionOnly" type="system">
  <hardware>
    <plugin>ros2_control_demo_example_17/RRBotSystemPositionOnlyHardware
↪</plugin>
    <param name="status_publish_rate">20.0</param> <!-- Defaults to 0.0␣
↪Hz -->
  </hardware>
  ...
</ros2_control>
```

This will create a publisher on the topic `/rrbot/hardware_status`.

Implementation Details of the Diagnostic Publisher

This example demonstrates the two recommended ways for a hardware component to perform ROS 2 communications: using the standard `diagnostic_updater` on the default node, and creating a separate custom node added to the Controller Manager's executor.

1. Using the ``diagnostic_updater`` with the Default Node (Standard Method)

The `diagnostic_updater` library is the standard ROS 2 tool for publishing diagnostics. It automatically handles timer creation and publishing to the `/diagnostics` topic, making it the preferred method. The `HardwareComponentInterface` provides a `get_node()` method to access the default node, which is passed to the updater.

The key steps are:

1. **Create an Updater**: Instantiate `diagnostic_updater::Updater`, passing it the default node. The updater internally creates a publisher to `/diagnostics` and a periodic timer (default 1 Hz).
2. **Set Hardware ID**: Set a unique identifier for the hardware component.
3. **Add a Diagnostic Task**: Add a function that will be called periodically by the updater's internal timer. This function populates a `DiagnosticStatusWrapper` with the current status of the hardware.

```
#include "diagnostic_updater/diagnostic_updater.hpp"
```

(continues on next page)

(continued from previous page)

```
if (get_node())
{
  updater_ = std::make_shared<diagnostic_updater::Updater>(get_node());
  updater_->setHardwareID(get_hardware_info().name);

  updater_->add(
    get_hardware_info().name + " Status", this, &
↪RRBotSystemPositionOnlyHardware::produce_diagnostics);
}

void RRBotSystemPositionOnlyHardware::produce_diagnostics(
  diagnostic_updater::DiagnosticStatusWrapper & stat)
{
  // Add status summary
  stat.summary(diagnostic_msgs::msg::DiagnosticStatus::OK, "Hardware is OK");
  // Optionally add key-value pairs
  // stat.add("voltage", "24.1V");
}
```

2. Creating a Custom Node with the Executor (Advanced Method)

For non-diagnostic topics or when a separate node identity is required, a hardware component can create its own node and add it to the Controller Manager's executor.

1. **Receiving the Executor Reference**: The `on_init` method of the hardware interface is implemented with an updated signature that accepts `HardwareComponentInterfaceParams`. This struct contains a weak pointer to the `ControllerManager`'s executor.

```
// Get Weak Pointer to Executor from HardwareComponentInterfaceParams
executor_ = params.executor;
```

2. **Safely Accessing the Executor**: Before using the executor, its `weak_ptr` must be "locked" into a `shared_ptr`. This is a crucial safety check to ensure the executor is still valid.

```
if (auto locked_executor = executor_.lock())
{
  // ... executor is valid and can be used here ...
}
else
{
  return hardware_interface::CallbackReturn::ERROR;
}
```

3. **Creating and Adding a Node**: Inside the locked scope, a standard `rclcpp::Node` is created. This new node is then added to the `ControllerManager`'s executor. This allows the node's callbacks (e.g., for timers or subscriptions) to be processed by the same multi-threaded executor.

```
// Inside the `if (auto locked_executor = ...)` block
custom_status_node_ = std::make_shared<rclcpp::Node>(get_hardware_info().name + "_
↪custom_node");
locked_executor->add_node(custom_status_node_->get_node_base_interface());
```

4. **Publishing Status Messages**: A publisher and a periodic wall timer are created on the new custom node. The timer's callback periodically publishes a status message, demonstrating that the node is alive and running alongside the main control loop.

```
// Inside the `if (auto locked_executor = ...)` block
custom_status_publisher_ =
  custom_status_node_->create_publisher<std_msgs::msg::String>("rrbot_custom_status",␣
↪10);

custom_status_timer_ = custom_status_node_->create_wall_timer(
  2s, [this](){ /* ... lambda to publish message ... */ });
```

3. (Extra)Using the Default Node, but with a Custom Publisher

This is not implemented in the example, but is also a viable option.

```
// Get Default Node added to executor
auto default_node = get_node();
if (default_node)
{
    default_status_publisher_ = default_node->create_publisher<std_msgs::msg::String>(
↪"rrbot_default_status", 10);
    using namespace std::chrono_literals;
    default_status_timer_ = default_node->create_wall_timer(2.5s, [this](){ /* ... */␣
↪});
}
```

This pattern is the recommended approach for hardware components that need to perform their own ROS communications without interfering with the real-time control loop.

Files used for this demos

- Launch file: rrbot.launch.py
- Controllers yaml:
  - rrbot_controllers.yaml
  - rrbot_jtc.yaml
- URDF file: rrbot.urdf.xacro
  - Description: rrbot_description.urdf.xacro
  - `ros2_control` tag: rrbot.ros2_control.xacro
- RViz configuration: rrbot.rviz
- Hardware interface plugin: rrbot.cpp

Controllers from this demo

- `Joint State Broadcaster` (ros2_controllers repository): *doc*
- `Forward Command Controller` (ros2_controllers repository): *doc*

CHAPTER

FIVE

UTILITIES

There exist several utility packages, like command line tools or reusable classes for writing controllers and real-time critical code.

5.1 Command Line Interface

The following commands support interacting with the `controller_manager` from the command line. They are available through the standard ROS2 CLI framework.

Currently supported commands are

- ros2 control list_controllers
- ros2 control list_controller_types
- ros2 control list_hardware_components
- ros2 control list_hardware_interfaces
- ros2 control load_controller
- ros2 control reload_controller_libraries
- ros2 control set_controller_state
- ros2 control set_hardware_component_state
- ros2 control switch_controllers
- ros2 control unload_controller
- ros2 control cleanup_controller
- ros2 control view_controller_chains
- ros2 control view_hardware_status

5.1.1 list_controllers

```
$ ros2 control list_controllers -h
usage: ros2 control list_controllers [-h] [--spin-time SPIN_TIME] [-s] [--claimed-
↪interfaces] [--required-state-interfaces] [--required-command-interfaces] [--
↪chained-interfaces] [--exported-interfaces] [--verbose] [-c CONTROLLER_MANAGER] [--
↪include-hidden-nodes]
                                [--ros-args ...]

Output the list of loaded controllers, their type and status

options:
  -h, --help            show this help message and exit
  --spin-time SPIN_TIME
                        Spin time in seconds to wait for discovery (only applies when␣
↪not using an already running daemon)
  -s, --use-sim-time    Enable ROS simulation time
  --claimed-interfaces  List controller's claimed interfaces
  --required-state-interfaces
                        List controller's required state interfaces
  --required-command-interfaces
                        List controller's required command interfaces
  --chained-interfaces  List interfaces that the controllers are chained to
  --exported-interfaces
                        List controller's exported state and reference interfaces
  --verbose, -v         List controller's claimed interfaces, required state␣
↪interfaces and required command interfaces
  -c CONTROLLER_MANAGER, --controller-manager CONTROLLER_MANAGER
                        Name of the controller manager ROS node (default: controller_
↪manager)
  --include-hidden-nodes
                        Consider hidden nodes as well
  --ros-args ...        Pass arbitrary arguments to the executable
```

Example output:

```
$ ros2 control list_controllers
test_controller_name[test_controller]    active
```

5.1.2 list_controller_types

```
$ ros2 control list_controller_types -h
usage: ros2 control list_controller_types [-h] [--spin-time SPIN_TIME] [-s] [-c␣
↪CONTROLLER_MANAGER] [--include-hidden-nodes] [--ros-args ...]

Output the available controller types and their base classes

options:
  -h, --help            show this help message and exit
  --spin-time SPIN_TIME
                        Spin time in seconds to wait for discovery (only applies when␣
↪not using an already running daemon)
  -s, --use-sim-time    Enable ROS simulation time
  -c CONTROLLER_MANAGER, --controller-manager CONTROLLER_MANAGER
                        Name of the controller manager ROS node (default: controller_
```

(continues on next page)

(continued from previous page)

```
↪manager)
  --include-hidden-nodes
                        Consider hidden nodes as well
  --ros-args ...        Pass arbitrary arguments to the executable
```

Example output:

```
$ ros2 control list_controller_types
diff_drive_controller/DiffDriveController                                controller_
↪interface::ControllerInterface
joint_state_broadcaster/JointStateBroadcaster                            controller_
↪interface::ControllerInterface
joint_trajectory_controller/JointTrajectoryController                    controller_
↪interface::ControllerInterface
```

5.1.3 list_hardware_components

```
$ ros2 control list_hardware_components -h
usage: ros2 control list_hardware_components [-h] [--spin-time SPIN_TIME] [-s] [--
↪verbose] [-c CONTROLLER_MANAGER] [--include-hidden-nodes] [--ros-args ...]

Output the list of available hardware components

options:
  -h, --help            show this help message and exit
  --spin-time SPIN_TIME
                        Spin time in seconds to wait for discovery (only applies when␣
↪not using an already running daemon)
  -s, --use-sim-time    Enable ROS simulation time
  --verbose, -v         List hardware components with command and state interfaces␣
↪along with their data types
  -c CONTROLLER_MANAGER, --controller-manager CONTROLLER_MANAGER
                        Name of the controller manager ROS node (default: controller_
↪manager)
  --include-hidden-nodes
                        Consider hidden nodes as well
  --ros-args ...        Pass arbitrary arguments to the executable
```

Example output:

```
$ ros2 control list_hardware_components
Hardware Component 0
    name: RRBot
    type: system
    plugin name: ros2_control_demo_hardware/RRBotSystemPositionOnlyHardware
    state: id=3 label=active
    command interfaces
      joint2/position [available] [claimed]
      joint1/position [available] [claimed]
```

```
$ ros2 control list_hardware_components -v
Hardware Component 0
    name: RRBot
    type: system
```

(continues on next page)

(continued from previous page)

```
    plugin name: ros2_control_demo_hardware/RRBotSystemPositionOnlyHardware
    state: id=3 label=active
    command interfaces
      joint2/position [double] [available] [claimed]
      joint1/position [double] [available] [claimed]
    state interfaces
      joint2/position [double] [available]
      joint1/position [double] [available]
```

5.1.4 list_hardware_interfaces

```
$ ros2 control list_hardware_interfaces -h
usage: ros2 control list_hardware_interfaces [-h] [--spin-time SPIN_TIME] [-s] [--
↪verbose] [-c CONTROLLER_MANAGER] [--include-hidden-nodes] [--ros-args ...]

Output the list of available command and state interfaces

options:
  -h, --help            show this help message and exit
  --spin-time SPIN_TIME
                        Spin time in seconds to wait for discovery (only applies when␣
↪not using an already running daemon)
  -s, --use-sim-time    Enable ROS simulation time
  --verbose, -v         List hardware interfaces and their data types
  -c CONTROLLER_MANAGER, --controller-manager CONTROLLER_MANAGER
                        Name of the controller manager ROS node (default: controller_
↪manager)
  --include-hidden-nodes
                        Consider hidden nodes as well
  --ros-args ...        Pass arbitrary arguments to the executable
```

```
$ ros2 control list_hardware_interfaces
command interfaces
  joint1/position [unclaimed]
  joint2/position [unclaimed]
state interfaces
  joint1/position
  joint2/position
```

```
$ ros2 control list_hardware_interfaces -v
command interfaces
  joint1/position [double] [unclaimed]
  joint2/position [double] [unclaimed]
state interfaces
  joint1/position [double]
  joint2/position [double]
```

5.1.5 load_controller

```
$ ros2 control load_controller -h
usage: ros2 control load_controller [-h] [--spin-time SPIN_TIME] [-s] [--set-state
↪{inactive,active}] [-c CONTROLLER_MANAGER] [--include-hidden-nodes] [--ros-args ...
↪] controller_name [param_file]

Load a controller in a controller manager

positional arguments:
  controller_name       Name of the controller
  param_file            The YAML file with the controller parameters

options:
  -h, --help            show this help message and exit
  --spin-time SPIN_TIME
                        Spin time in seconds to wait for discovery (only applies when␣
↪not using an already running daemon)
  -s, --use-sim-time    Enable ROS simulation time
  --set-state {inactive,active}
                        Set the state of the loaded controller
  -c CONTROLLER_MANAGER, --controller-manager CONTROLLER_MANAGER
                        Name of the controller manager ROS node (default: controller_
↪manager)
  --include-hidden-nodes
                        Consider hidden nodes as well
  --ros-args ...        Pass arbitrary arguments to the executable
```

5.1.6 reload_controller_libraries

```
$ ros2 control reload_controller_libraries -h
usage: ros2 control reload_controller_libraries [-h] [--spin-time SPIN_TIME] [-s] [--
↪force-kill] [-c CONTROLLER_MANAGER] [--include-hidden-nodes] [--ros-args ...]

Reload controller libraries

options:
  -h, --help            show this help message and exit
  --spin-time SPIN_TIME
                        Spin time in seconds to wait for discovery (only applies when␣
↪not using an already running daemon)
  -s, --use-sim-time    Enable ROS simulation time
  --force-kill          Force stop of loaded controllers
  -c CONTROLLER_MANAGER, --controller-manager CONTROLLER_MANAGER
                        Name of the controller manager ROS node (default: controller_
↪manager)
  --include-hidden-nodes
                        Consider hidden nodes as well
  --ros-args ...        Pass arbitrary arguments to the executable
```

5.1.7 set_controller_state

```
$ ros2 control set_controller_state -h
usage: ros2 control set_controller_state [-h] [--spin-time SPIN_TIME] [-s] [-c␣
↪CONTROLLER_MANAGER] [--include-hidden-nodes] [--ros-args ...] controller_name
↪{unconfigured,inactive,active}

Adjust the state of the controller

positional arguments:
  controller_name       Name of the controller to be changed
  {unconfigured,inactive,active}     State in which the controller should be changed␣
↪to

options:
  -h, --help            show this help message and exit
  --spin-time SPIN_TIME
                        Spin time in seconds to wait for discovery (only applies when␣
↪not using an already running daemon)
  -s, --use-sim-time    Enable ROS simulation time
  -c CONTROLLER_MANAGER, --controller-manager CONTROLLER_MANAGER
                        Name of the controller manager ROS node (default: controller_
↪manager)
  --include-hidden-nodes
                        Consider hidden nodes as well
  --ros-args ...        Pass arbitrary arguments to the executable
```

5.1.8 set_hardware_component_state

```
$ ros2 control set_hardware_component_state -h
usage: ros2 control set_hardware_component_state [-h] [--spin-time SPIN_TIME] [-s] [-
↪c CONTROLLER_MANAGER] [--include-hidden-nodes] [--ros-args ...] hardware_component_
↪name {unconfigured,inactive,active}

Adjust the state of the hardware component

positional arguments:
  hardware_component_name
                        Name of the hardware_component to be changed
  {unconfigured,inactive,active}
                        State in which the hardware component should be changed to

options:
  -h, --help            show this help message and exit
  --spin-time SPIN_TIME
                        Spin time in seconds to wait for discovery (only applies when␣
↪not using an already running daemon)
  -s, --use-sim-time    Enable ROS simulation time
  -c CONTROLLER_MANAGER, --controller-manager CONTROLLER_MANAGER
                        Name of the controller manager ROS node (default: controller_
↪manager)
  --include-hidden-nodes
                        Consider hidden nodes as well
  --ros-args ...        Pass arbitrary arguments to the executable
```

5.1.9 switch_controllers

```
$ ros2 control switch_controllers -h
usage: ros2 control switch_controllers [-h] [--spin-time SPIN_TIME] [-s] [--
↪deactivate [DEACTIVATE ...]] [--activate [ACTIVATE ...]] [--strict] [--activate-
↪asap] [--switch-timeout SWITCH_TIMEOUT]
                                       [-c CONTROLLER_MANAGER] [--include-hidden-
↪nodes] [--ros-args ...]

Switch controllers in a controller manager

options:
  -h, --help            show this help message and exit
  --spin-time SPIN_TIME
                        Spin time in seconds to wait for discovery (only applies when␣
↪not using an already running daemon)
  -s, --use-sim-time    Enable ROS simulation time
  --deactivate [DEACTIVATE ...]
                        Name of the controllers to be deactivated
  --activate [ACTIVATE ...]
                        Name of the controllers to be activated
  --strict              Strict switch
  --activate-asap       Start asap controllers
  --switch-timeout SWITCH_TIMEOUT
                        Timeout for switching controllers
  -c CONTROLLER_MANAGER, --controller-manager CONTROLLER_MANAGER
                        Name of the controller manager ROS node (default: controller_
↪manager)
  --include-hidden-nodes
                        Consider hidden nodes as well
  --ros-args ...        Pass arbitrary arguments to the executable
```

5.1.10 unload_controller

```
$ ros2 control unload_controller -h
usage: ros2 control unload_controller [-h] [--spin-time SPIN_TIME] [-s] [-c␣
↪CONTROLLER_MANAGER] [--include-hidden-nodes] [--ros-args ...] controller_name

Unload a controller in a controller manager

positional arguments:
  controller_name       Name of the controller

options:
  -h, --help            show this help message and exit
  --spin-time SPIN_TIME
                        Spin time in seconds to wait for discovery (only applies when␣
↪not using an already running daemon)
  -s, --use-sim-time    Enable ROS simulation time
  -c CONTROLLER_MANAGER, --controller-manager CONTROLLER_MANAGER
                        Name of the controller manager ROS node (default: controller_
↪manager)
  --include-hidden-nodes
                        Consider hidden nodes as well
  --ros-args ...        Pass arbitrary arguments to the executable
```

5.1.11 cleanup_controller

```
$ ros2 control cleanup_controller -h
usage: ros2 control cleanup_controller [-h] [--spin-time SPIN_TIME] [-s] [-c␣
↪CONTROLLER_MANAGER] [--include-hidden-nodes] [--ros-args ...] controller_name

Cleanup a controller in a controller manager

positional arguments:
  controller_name       Name of the controller

options:
  -h, --help            show this help message and exit
  --spin-time SPIN_TIME
                        Spin time in seconds to wait for discovery (only applies when␣
↪not using an already running daemon)
  -s, --use-sim-time    Enable ROS simulation time
  -c CONTROLLER_MANAGER, --controller-manager CONTROLLER_MANAGER
                        Name of the controller manager ROS node (default: controller_
↪manager)
  --include-hidden-nodes
                        Consider hidden nodes as well
  --ros-args ...        Pass arbitrary arguments to the executable
```

5.1.12 view_controller_chains

```
$ ros2 control view_controller_chains -h
usage: ros2 control view_controller_chains [-h] [--spin-time SPIN_TIME] [-s] [-c␣
↪CONTROLLER_MANAGER] [--include-hidden-nodes] [--ros-args ...]

Generates a diagram of the loaded chained controllers into /tmp/controller_diagram.gv.
↪pdf

options:
  -h, --help            show this help message and exit
  --spin-time SPIN_TIME
                        Spin time in seconds to wait for discovery (only applies when␣
↪not using an already running daemon)
  -s, --use-sim-time    Enable ROS simulation time
  -c CONTROLLER_MANAGER, --controller-manager CONTROLLER_MANAGER
                        Name of the controller manager ROS node (default: controller_
↪manager)
  --include-hidden-nodes
                        Consider hidden nodes as well
  --ros-args ...        Pass arbitrary arguments to the executable
```

5.1.13 view_hardware_status

```
$ ros2 control view_hardware_status -h
usage: ros2 control view_hardware_status [-h] [--spin-time SPIN_TIME] [-s] [-i␣
↪HARDWARE_ID] [-d DEVICE_ID]

Echo hardware status messages with filtering capabilities

options:
  -h, --help            show this help message and exit
  --spin-time SPIN_TIME
                        Spin time in seconds to wait for discovery (only applies when␣
↪not using an already running daemon)
  -s, --use-sim-time    Enable ROS simulation time
  -i HARDWARE_ID, --hardware-id HARDWARE_ID
                        Filter by a specific hardware component ID.
  -d DEVICE_ID, --device-id DEVICE_ID
                        Filter by a specific device ID within a hardware component.
```

5.2 control_toolbox

This package contains several C++ classes and filter plugins useful in writing controllers.

5.2.1 Base classes

PID

PID Controller

The PID (Proportional-Integral-Derivative) controller is a widely used feedback controller. This class implements a generic structure that can be used to create a wide range of PID controllers. It can function independently or be subclassed to provide more specific controls based on a particular control loop. Integral retention on reset is supported, which prevents re-winding the integrator after temporary disabling in presence of constant disturbances.

PID Equation

The standard PID equation is given by:

command = pterm + iterm + dterm

where:

- pterm = pgain * error
- iterm = iterm + igain * error * dt
- dterm = dgain * derror

and:

- error = desired_state - measured_state
- derror = (error - errorlast) / dt

Parameters

- `p` (Proportional gain): This gain determines the reaction to the current error. A larger proportional gain results in a larger change in the controller output for a given change in the error.
- `i` (Integral gain): This gain determines the reaction based on the sum of recent errors. The integral term accounts for past values of the error and integrates them over time to produce the `i_term`. This helps in eliminating steady-state errors.
- `d` (Derivative gain): This gain determines the reaction based on the rate at which the error has been changing. The derivative term predicts future errors based on the rate of change of the current error. This helps in reducing overshoot, settling time, and other transient performance variables.
- `u_clamp` (Minimum and maximum bounds for the controller output): These bounds are applied to the final command output of the controller, ensuring the output stays within acceptable physical limits.
- `tracking_time_constant` (Tracking time constant): This parameter is specific to the 'back_calculation' anti-windup strategy. If set to 0.0 when this strategy is selected, a recommended default value will be applied.
- `antiwindup_strat` (Anti-windup strategy): This parameter selects how the integrator is prevented from winding up when the controller output saturates. Available options are:
  - `NONE`: no anti-windup technique; the integral term accumulates without correction.
  - `BACK_CALCULATION`: adjusts the integral term based on the difference between the unsaturated and saturated outputs using the tracking time constant `tracking_time_constant`. Faster correction for smaller `tracking_time_constant`.
  - `CONDITIONAL_INTEGRATION`: only updates the integral term when the controller is not in saturation or when the error drives the output away from saturation, freezing integration otherwise.

Anti-Windup Strategies

Anti-windup functionality is crucial for PID controllers, especially when the control output is subject to saturation (clamping). Without anti-windup, the integral term can accumulate excessively when the controller output is saturated, leading to large overshoots and sluggish response once the error changes direction. The `control_toolbox::Pid` class offers two anti-windup strategies:

- **`BACK_CALCULATION`**: This strategy adjusts the integral term based on the difference between the saturated and unsaturated controller output. When the controller output `command` exceeds the output limits (`u_max` or `u_min`), the integral term `i_term` is adjusted by subtracting a value proportional to the difference between the saturated output `command_sat` and the unsaturated output `command`. This prevents the integral term from accumulating beyond what is necessary to maintain the output at its saturation limit. The `tracking_time_constant` parameter is used to tune the speed of this adjustment. A smaller value results in faster anti-windup action.

  The update rule for the integral term with back-calculation is:

  iterm += dt * (igain * error + (1 / trktc) * (commandsat - command))

  If `trk_tc`, i.e., `tracking_time_constant` parameter, is set to 0.0, a default value is calculated based on the proportional and derivative gains:

  - If `d_gain` is not zero: trktc = $\sqrt{}$(dgain / igain)
  - If `d_gain` is zero: trktc = pgain / igain

- **`CONDITIONAL_INTEGRATION`**: In this strategy, the integral term is only updated when the controller is not in saturation or when the error has a sign that would lead the controller out of saturation. Specifically, the integral term is frozen (not updated) if the controller output is saturated and the error has the same sign as the saturated output. This prevents further accumulation of the integral term in the direction of saturation.

The integral term is updated only if the following condition is met:

(command - commandsat = 0) ∨ (error * command ≤ 0)

This means the integral term `i_term` is updated as `i_term += dt * i_gain * error` only when the controller is not saturated, or when it is saturated but the error is driving the output away from the saturation limit.

Usage Example

To use the `Pid` class, you should first call some version of `initialize()` and then call `compute_command()` at every update step. For example:

```
control_toolbox::Pid pid;
pid.initialize(6.0, 1.0, 2.0, 5, -5,2,control_toolbox::AntiwindupStrategy::BACK_
↪CALCULATION);
double position_desired = 0.5;
...
rclcpp::Time last_time = get_clock()->now();
while (true) {
  rclcpp::Time time = get_clock()->now();
  double effort = pid.compute_command(position_desired - currentPosition(), time -␣
↪last_time);
  last_time = time;
}
```

References

1. Visioli, A. *Practical PID Control*. London: Springer-Verlag London Limited, 2006. 476 p.
2. Vrancic, D., Horowitz, R., & Hagiwara, T. "Antiwindup, Bumpless, and Conditioned Transfer Techniques for PID Controllers." *IEEE Control Systems Magazine*, vol. 16, no. 4, 1996, pp. 48–57.
3. Bohn, C.; Atherton, D. "An analysis package comparing PID anti-windup strategies." *IEEE Control Systems Magazine*, 1995, pp. 34–40.
4. Åström, K.; Hägglund, T. *PID Controllers: Theory, Design and Tuning*. Research Triangle Park, USA: ISA Press / Springer-Verlag London Limited, 1995. 343 p.

5.2.2 Control filters

Implement filter plugins for control purposes as ros/filters

Available filters

- Gravity Compensation: implements a gravity compensation algorithm, removing the gravity component from the incoming data (Wrench).
- Low Pass: implements a low-pass filter based on a time-invariant Infinite Impulse Response (IIR) filter, for different data types (doubles or wrench).
- Exponential Filter: Exponential filter for double data type.

Gravity compensation filter

This filter implements an algorithm compensating for the gravity forces acting at the center of gravity (CoG) of a known mass, computed at a `sensor_frame` and applied to a `data_in` wrench.

The filter relies on tf2, and might fail if transforms are missing.

Note that, for convenience, the filter can perform additional frame changes if data_out frame id is given.

GC: Required parameters

- `world_frame` ($\mathscr{R}$w): frame in which the `CoG.force` is represented.
- `sensor_frame` ($\mathscr{R}$s): frame in which the `CoG.pos` is defined
- `CoG.pos` (ps): position of the CoG of the mass the filter should compensate for
- `CoG.force` (gw): constant (but updatable) force of gravity at the Cog (typically m.G), defined along axes of the `world_frame`

GC: Algorithm

Given

- above-required parameters, $\mathscr{R}$w, $\mathscr{R}$s, ps, gw
- `data_in`, a wrench $\mathscr{F}$i = {fi, τi} represented in the `data_in` frame $\mathscr{R}$i
- access to tf2 homogeneous transforms:
  - Tsi from $\mathscr{R}$i to $\mathscr{R}$s
  - Tsw from $\mathscr{R}$w to $\mathscr{R}$s
  - Tos from $\mathscr{R}$s to $\mathscr{R}$o

Compute `data_out` compensated wrench $\mathscr{F}$co = {fco, τco} represented in the `data_out` frame $\mathscr{R}$o if given, or the `data_in` frame $\mathscr{R}$i otherwise, with equations:

$\mathscr{F}$co = Tos.$\mathscr{F}$cs,

with $\mathscr{F}$cs = {fcs, τcs} the compensated wrench in `sensor_frame` (common frame for computation)

and,

fcs = fs - Tswgw

its force and,

τcs = τs - ps x (Tswgw)

its torque, and

$\mathscr{F}$s = Tsi.$\mathscr{F}$i = {fs, τs}

the full transform of the input wrench $\mathscr{F}$i to sensor frame $\mathscr{R}$s

Remarks :

- a full vector is used for gravity force, to not impose gravity to be only along z of `world_frame`.

- `data_in` frame is usually equal to `sensor_frame`, but could be different since measurement of wrench might occur in another frame. E.g.: measurements are at the **FT sensor flange** = `data_in` frame, but CoG is given in **FT sensor base** = `sensor_frame` (=frame to which it is mounted on the robot), introducing an offset (thickness of the sensor) to be accounted for.
- `data_out` frame is usually `data_in` frame, but for convenience, can be set to any other useful frame. E.g.: wrench expressed in a `control_frame` like the center of a gripper.
- Tsw will only rotate the gw vector, because gravity is a field applied everywhere, and not a wrench (no torque should be induced by transforming from $\mathscr{R}$w to $\mathscr{R}$s).

Low Pass filter

This filter implements a low-pass filter in the form of an IIR filter, applied to a `data_in` (double or wrench). The feedforward and feedback coefficients of the IIR filter are computed from the low-pass filter parameters.

LPF: Required parameters

- sampling frequency as `sf`
- damping frequency as `df`
- damping intensity as `di`

LPF: Algorithm

Given

- above-required parameters, `sf`, `df`, `di`
- `data_in`, a double or wrench `x`

Compute `data_out`, the filtered output `y(n)` with equation:

y(n) = b x(n-1) + a y(n-1)

with

- a the feedbackward coefficient such that a = exp(-1/sf (2 pi df) / (10^(di/-10)))
- b the feedforward coefficient such that b = 1 - a

Exponential filter

EF: Required parameters

- `alpha`: the exponential decay factor

EF: Algorithm

smoothed_value = alpha * current_value + (1 - alpha) * last_smoothed_value;

5.3 realtime_tools

Contains a set of tools that can be used from a hard realtime thread, without breaking the realtime behavior.

5.3.1 Exchange data between different threads

This package contains different concepts for exchanging data between different threads. In the following, a guideline for the usage for ros2_controllers is given.

Provided concepts

RealtimeThreadSafeBox

A Box that ensures thread-safe access to the boxed contents. Access is best effort. If it can not lock it will return.

LockFreeQueue

This class provides a base implementation for lock-free queues on top of Boost.Lockfree for lock-free queues with various functionalities, such as pushing, popping, and checking the state of the queue. It supports both single-producer single-consumer (SPSC) and multiple-producer multiple-consumer (MPMC) queues.

RealtimePublisher

The `realtime_tools::RealtimePublisher` allows users that write C++ ros2_controllers to publish messages on a ROS topic from a hard realtime loop. The normal ROS publisher is not realtime safe, and should not be used from within the update loop of a realtime controller. The realtime publisher is a wrapper around the ROS publisher; the wrapper creates an extra non-realtime thread that publishes messages on a ROS topic. The example below shows a typical usage of the realtime publisher in the `on_configure()` (non-realtime method) and `update()` (realtime method) methods of a realtime controller:

```
#include <realtime_tools/realtime_publisher.hpp>

class MyController : public controller_interface::ControllerInterface
{
...
private:
  std::unique_ptr<realtime_tools::RealtimePublisher<my_msgs::msg::MyMsg>> state_
↪publisher_;
  std::shared_ptr<rclcpp::Publisher<my_msgs::msg::MyMsg>> s_publisher_;
  my_msgs::msg::MyMsg some_msg_;
}

controller_interface::CallbackReturn MyController::on_configure(
  const rclcpp_lifecycle::State & /*previous_state*/)
```

(continues on next page)

(continued from previous page)

```
{
  ...
  s_publisher_ = get_node()->create_publisher<my_msgs::msg::MyMsg>(
    "~/status", rclcpp::SystemDefaultsQoS());
  state_publisher_ =
    std::make_unique<realtime_tools::RealtimePublisher<my_msgs::msg::MyMsg>>(s_
↪publisher_);
  some_msg_.header.frame_id = params_.frame_id;
  ...
}

controller_interface::return_type MyController::update(
  const rclcpp::Time & /*time*/, const rclcpp::Duration & /*period*/)
{
  ...
  // Publish controller state
  some_msg_.header.stamp = get_node()->now();
  // Fill in the rest of the message
  ....
  state_publisher_->try_publish(some_msg_);
}
```

Guidelines

There exist the following typical use-cases for ros2_controllers:

- Passing command messages from topic subscribers in a non-realtime thread to the realtime-thread (one-way).
  - If you care only about the latest received message, use the `realtime_tools::RealtimeThreadSafeBox`.
  - If you care about the intermediate messages, i.e., receiving more than one message before the realtime-thread can process it: Use the `realtime_tools::LockFreeQueue` and set a suitable queue size for your application, i.e., consider the maximum expected topic rate vs controller update rate.
- Send data from the realtime thread to the non-realtime thread for publishing data (one-way): Use the `realtime_tools::RealtimePublisher`.
- Exchange data (two-way) between a non-realtime thread and a realtime-thread like state information, current goals etc.:
  - For primitive types like `bool`, you can simply use `std::atomic<bool>`, see cppreference.
  - For all other types, when missing some data samples from RT loop is not of major importance, then use the `realtime_tools::RealtimeThreadSafeBox` with `try_set` method. In the contrary situation, it is recommended to use `realtime_tools::LockFreeQueue` to avoid missing any samples.

5.3.2 Other classes and helper methods

Tba.

CHAPTER

SIX

SIMULATOR INTEGRATIONS

6.1 Hosted by ros-controls

6.1.1 gz_ros2_control

This is a ROS 2 package for integrating the *ros2_control* controller architecture with the Gazebo simulator.

This package provides a Gazebo-Sim system plugin which instantiates a *ros2_control* controller manager and connects it to a Gazebo model.

Installation

Binary packages

`gz_ros2_control` is released for ROS 2 rolling on Ubuntu. To use it, you have to install `ros-rolling-gz-ros2-control` package, e.g., by running the following command:

```
sudo apt install ros-rolling-gz-ros2-control ros-rolling-gz-ros2-control-demos
```

Building from source

To use latest yet-to-be-released features or use a non-default Gazebo combination (see the compatibility matrix in the README for currently supported combinations), you have to build the package from source.

Note that `gz_ros2_control` depends on the version of Gazebo that is provided by the Gazebo Vendor packages gz_plugin_vendor and gz_sim_vendor. Currently, for ROS 2 Jazzy and Rolling, the Gazebo version is Harmonic.

To compile `gz_ros2_control` from source, create a workspace, clone the correct branch of this repo and compile it:

```
mkdir -p ~/gz_ros2_control_ws/src
cd ~/gz_ros2_control_ws/src
git clone https://github.com/ros-controls/gz_ros2_control -b $ROS_DISTRO
rosdep install -r --from-paths . --ignore-src --rosdistro $ROS_DISTRO -y
cd ~/gz_ros2_control_ws
colcon build
```

Using docker

Build the docker image

```
cd Dockerfile
docker build -t gz_ros2_control .
```

and run the demo

1. Using Docker

   Docker allows us to run the demo without the GUI if configured properly. The following command runs the demo without the GUI:

   ```
   docker run -it --rm --name gz_ros2_control_demo --net host gz_ros2_control␣
   ↪ros2 launch gz_ros2_control_demos cart_example_position.launch.py␣
   ↪gui:=false
   ```

   Then on your local machine, you can run the Gazebo client:

   ```
   gz sim -g
   ```

2. Using Rocker

   To run the demo with GUI we are going to use rocker which is a tool to run docker images with customized local support injected for things like nvidia support. Rocker also supports user id specific files for cleaner mounting file permissions. You can install this tool with the following instructions. (make sure you meet all of the prerequisites.

   The following command will launch Gazebo:

   ```
   rocker --x11 --nvidia --name gz_ros2_control_demo gz_ros2_control:latest
   ```

   The following commands allow the cart to be moved along the rail:

   ```
   docker exec -it gz_ros2_control_demo bash
   source /home/ros2_ws/install/setup.bash
   ros2 run gz_ros2_control_demos example_position
   ```

Add ros2_control tag to a URDF or SDF

Simple setup

To use *ros2_control* with your robot, you need to add some additional elements to your URDF or SDF. You should include the tag `<ros2_control>` to access and control the robot interfaces. We should include:

- a specific `<plugin>` for our robot
- `<joint>` tag including the robot controllers: commands and states.

```
<ros2_control name="GazeboSimSystem" type="system">
  <hardware>
    <plugin>gz_ros2_control/GazeboSimSystem</plugin>
  </hardware>
  <joint name="slider_to_cart">
    <command_interface name="effort">
      <param name="min">-1000</param>
```

(continues on next page)

(continued from previous page)

```
      <param name="max">1000</param>
    </command_interface>
    <state_interface name="position">
      <param name="initial_value">1.0</param>
    </state_interface>
    <state_interface name="velocity"/>
    <state_interface name="effort"/>
  </joint>
</ros2_control>
```

Using mimic joints in simulation

To use `mimic` joints in *gz_ros2_control* you should define its parameters in your URDF or SDF, i.e, set the `<mimic>` tag to the mimicked joint (see the URDF specification or the SDF specification)

```
<joint name="right_finger_joint" type="prismatic">
  <axis xyz="0 1 0"/>
  <origin xyz="0.0 -0.48 1" rpy="0.0 0.0 0.0"/>
  <parent link="base"/>
  <child link="finger_right"/>
  <limit effort="1000.0" lower="0" upper="0.38" velocity="10"/>
</joint>
<joint name="left_finger_joint" type="prismatic">
  <mimic joint="right_finger_joint" multiplier="1" offset="0"/>
  <axis xyz="0 1 0"/>
  <origin xyz="0.0 0.48 1" rpy="0.0 0.0 3.1415926535"/>
  <parent link="base"/>
  <child link="finger_left"/>
  <limit effort="1000.0" lower="0" upper="0.38" velocity="10"/>
</joint>
```

The mimic joint must not have command interfaces configured in the `<ros2_control>` tag, but state interfaces can be configured.

Using force-torque sensors in simulation

To use `force-torque` sensors in *gz_ros2_control* you should define its parameters in your URDF or SDF (see the SDF specification)

An example in SDF is shown here:

```
<sensor name="force_torque_sensor" type="force_torque">
  <update_rate>10.0</update_rate>
  <always_on>true</always_on>
  <visualize>true</visualize>
  <topic>force_torque_sensor</topic>
</sensor>
```

It is important to add this as `reference` sensor in the `<gazebo>` tag in your URDF file where the reference is the joint you will be attaching the force torque sensor to:

```
<gazebo reference="attached_joint">
  <!-- If 'attached_joint' is of 'fixed' type,
```

(continues on next page)

(continued from previous page)

```
  setting 'preserveFixedJoint' to true will prevent the
  links from being lumped together during the URDF to
  SDF conversion. Otherwise, it can be omitted. -->
  <preserveFixedJoint>true</preserveFixedJoint>
  <sensor name="force_torque_sensor" type="force_torque">
    <update_rate>10.0</update_rate>
    <always_on>true</always_on>
    <visualize>true</visualize>
    <topic>force_torque_sensor</topic>
  </sensor>
</gazebo>
```

Add the gz_ros2_control plugin

In addition to the *ros2_control* tags, a Gazebo plugin needs to be added to your URDF or SDF that actually parses the *ros2_control* tags and loads the appropriate hardware interfaces and controller manager. By default the *gz_ros2_control* plugin is very simple, though it is also extensible via an additional plugin architecture to allow power users to create their own custom robot hardware interfaces between *ros2_control* and Gazebo.

URDF

```
<gazebo>
  <plugin filename="libgz_ros2_control-system.so" name="gz_ros2_
↪control::GazeboSimROS2ControlPlugin">
    <parameters>$(find gz_ros2_control_demos)/config/cart_controller.yaml</parameters>
  </plugin>
</gazebo>
```

SDF

```
<plugin filename="libgz_ros2_control-system.so" name="gz_ros2_
↪control::GazeboSimROS2ControlPlugin">
  <parameters>$(find gz_ros2_control_demos)/config/cart_controller.yaml</parameters>
</plugin>
```

The *gz_ros2_control* `<plugin>` tag also has the following optional child elements:

- `<parameters>`: A YAML file with the configuration of the controllers. This element can be given multiple times to load multiple files.
- `<controller_manager_name>`: Set controller manager name (default: `controller_manager`)

The following additional parameters can be set via child elements in the URDF/SDF or via ROS parameters in the YAML file above:

- `<hold_joints>`: if set to true (default), it will hold the joints' position if their interface was not claimed, e.g., the controller hasn't been activated yet.
- `<position_proportional_gain>`: Set the proportional gain. (default: 0.1) This determines the set-point for a position-controlled joint `joint_velocity = joint_position_error * position_proportional_gain`.

or via ROS parameters:

```
gz_ros_control:
  ros__parameters:
```

(continues on next page)

(continued from previous page)

```
    hold_joints: false
    position_proportional_gain: 0.5
```

Additionally, one can specify a namespace and remapping rules, which will be forwarded to the controller_manager and loaded controllers. Add the following `<ros>` section:

URDF

```
<gazebo>
  <plugin filename="libgz_ros2_control-system.so" name="gz_ros2_
↪control::GazeboSimROS2ControlPlugin">
    ...
    <ros>
      <namespace>my_namespace</namespace>
      <remapping>/robot_description:=/robot_description_full</remapping>
    </ros>
  </plugin>
</gazebo>
```

SDF

```
<plugin filename="libgz_ros2_control-system.so" name="gz_ros2_
↪control::GazeboSimROS2ControlPlugin">
  ...
  <ros>
    <namespace>my_namespace</namespace>
    <remapping>/robot_description:=/robot_description_full</remapping>
  </ros>
</plugin>
```

Advanced: custom gz_ros2_control Simulation Plugins

The *gz_ros2_control* Gazebo plugin also provides a pluginlib-based interface to implement custom interfaces between Gazebo and *ros2_control* for simulating more complex mechanisms (nonlinear springs, linkages, etc) or actuator dynamics.

These plugins must inherit the `gz_ros2_control::GazeboSimSystemInterface`, which implements a simulated *ros2_control* `hardware_interface::SystemInterface`.

The respective GazeboSimSystemInterface is specified in a URDF or SDF model and is loaded when the robot model is loaded. For example, the following XML will load a custom plugin instead:

URDF

```
<ros2_control name="GazeboSimSystem" type="system">
  <hardware>
    <plugin>gz_ros2_control_demos/GazeboCustomSimSystem</plugin>
  </hardware>
  ...
<ros2_control>
<gazebo>
  <plugin name="gz_ros2_control::GazeboSimROS2ControlPlugin" filename="libgz_ros2_
↪control-system">
    ...
  </plugin>
</gazebo>
```

SDF

```
<ros2_control name="GazeboSimSystem" type="system">
  <hardware>
    <plugin>gz_ros2_control_demos/GazeboCustomSimSystem</plugin>
  </hardware>
  ...
<ros2_control>
<plugin name="gz_ros2_control::GazeboSimROS2ControlPlugin" filename="libgz_ros2_
↪control-system">
  ...
</plugin>
```

The `gz_ros2_control_demos/GazeboCustomSimSystem` demonstrates how to implement actuator dynamics for a joint with velocity command interface by using a configurable low pass filter. Run

```
ros2 launch gz_ros2_control_demos cart_example_velocity_custom_plugin.launch.py
```

and compare it with the behavior of `cart_example_velocity.launch.py` using any plotting tool like plotjuggler.

Set up controllers

Use the tag `<parameters>` inside `<plugin>` to set the YAML file with the controller configuration and use the tag `<controller_manager_prefix_node_name>` to set the controller manager node name.

URDF

```
<gazebo>
  <plugin name="gz_ros2_control::GazeboSimROS2ControlPlugin" filename="libgz_ros2_
↪control-system">
    <parameters>$(find gz_ros2_control_demos)/config/cart_controller.yaml</parameters>
    <controller_manager_prefix_node_name>controller_manager</controller_manager_
↪prefix_node_name>
  </plugin>
<gazebo>
```

SDF

```
<plugin name="gz_ros2_control::GazeboSimROS2ControlPlugin" filename="libgz_ros2_
↪control-system">
  <parameters>$(find gz_ros2_control_demos)/config/cart_controller.yaml</parameters>
  <controller_manager_prefix_node_name>controller_manager</controller_manager_prefix_
↪node_name>
</plugin>
```

The following is a basic configuration of the controllers:

- `joint_state_broadcaster`: This controller publishes the state of all resources registered to a `hardware_interface::StateInterface` to a topic of type `sensor_msgs/msg/JointState`.
- `joint_trajectory_controller`: This controller creates an action called `/joint_trajectory_controller/follow_joint_trajectory` of type `control_msgs::action::FollowJointTrajectory`.

```
controller_manager:
  ros__parameters:
    update_rate: 1000  # Hz
```

(continues on next page)

(continued from previous page)

```
    joint_state_broadcaster:
      type: joint_state_broadcaster/JointStateBroadcaster

joint_trajectory_controller:
  ros__parameters:
    type: joint_trajectory_controller/JointTrajectoryController
    joints:
      - slider_to_cart
    command_interfaces:
      - position
    state_interfaces:
      - position
      - velocity
```

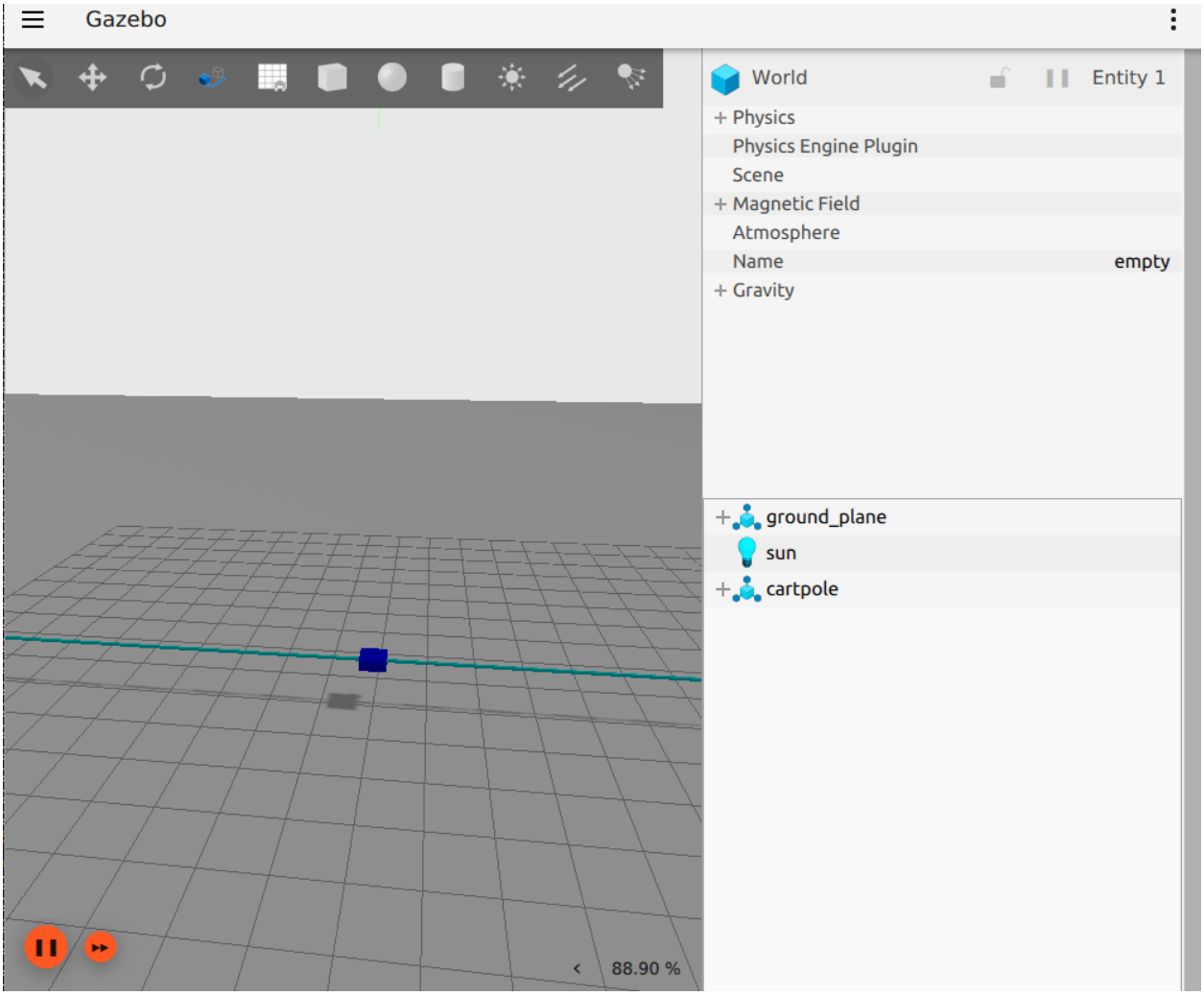
gz_ros2_control_demos

There are some examples in the *gz_ros2_control_demos* package. To specify whether to use URDF or SDF, you can launch the demo in the following way (the default is URDF):

```
ros2 launch gz_ros2_control_demos <launch file> description_format:=sdf
```

Cart on rail

These examples allow to launch a cart in a 30 meter rail.



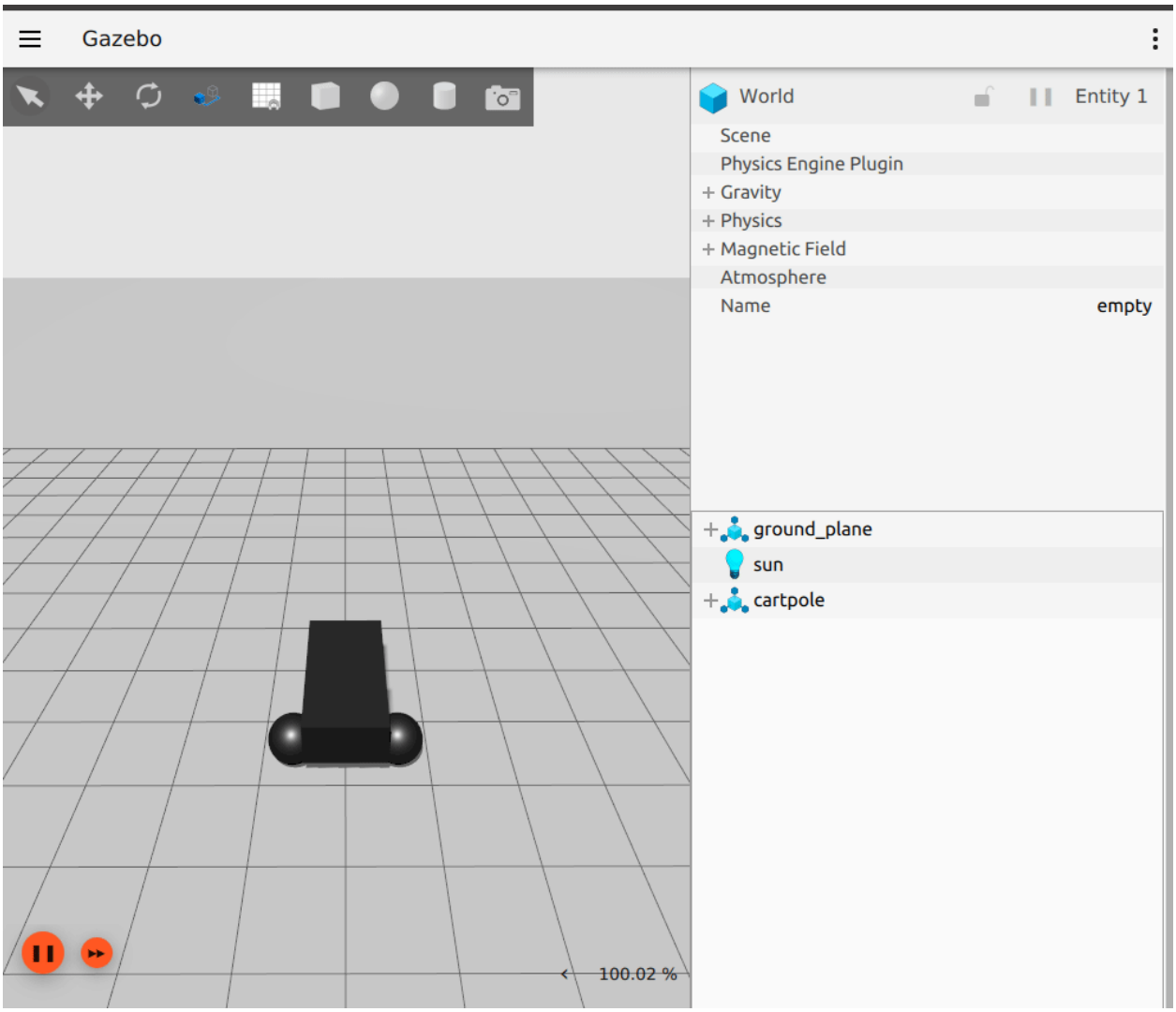
You can run some of the example configurations by running the following commands:

```
ros2 launch gz_ros2_control_demos cart_example_position.launch.py
ros2 launch gz_ros2_control_demos cart_example_velocity.launch.py
ros2 launch gz_ros2_control_demos cart_example_effort.launch.py
```

When the Gazebo world is launched, you can run some of the following commands to move the cart.

```
ros2 run gz_ros2_control_demos example_position
ros2 run gz_ros2_control_demos example_velocity
ros2 run gz_ros2_control_demos example_effort
```

Mobile robots



You can run some of the mobile robots running the following commands:

```
ros2 launch gz_ros2_control_demos diff_drive_example.launch.py
ros2 launch gz_ros2_control_demos tricycle_drive_example.launch.py
ros2 launch gz_ros2_control_demos ackermann_drive_example.launch.py
ros2 launch gz_ros2_control_demos mecanum_drive_example.launch.py
```

When the Gazebo world is launched you can run the following command to move the robots.

```
ros2 run gz_ros2_control_demos example_mobile_robots
```

You can also drive the robots from the keyboard using the following command:

```
ros2 run teleop_twist_keyboard teleop_twist_keyboard --ros-args -p stamped:=true
```

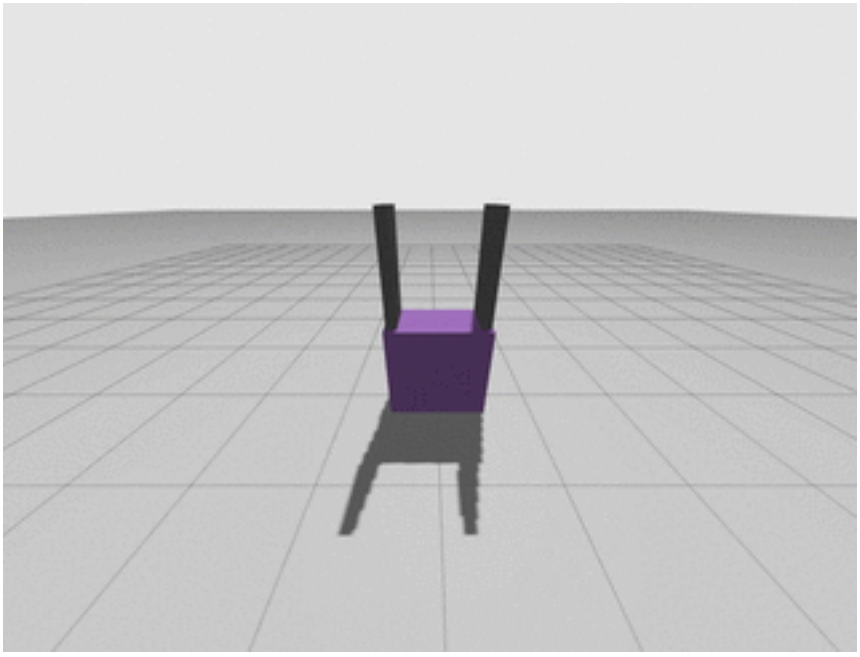
You can also manually publish on the `/cmd_vel` topic to drive the robots:

```
ros2 topic pub --rate 10 /cmd_vel geometry_msgs/msg/TwistStamped "
twist:
  linear:
    x: 0.7
    y: 0.0
    z: 0.0
  angular:
    x: 0.0
    y: 0.0
    z: 1.0"
```

Gripper

The following example shows a parallel gripper with a mimic joint:

```
ros2 launch gz_ros2_control_demos gripper_mimic_joint_example_position.launch.py
```



To demonstrate the setup of the initial position and a position-mimicked joint in case of an effort command interface of the joint to be mimicked, run

```
ros2 launch gz_ros2_control_demos gripper_mimic_joint_example_effort.launch.py
```

instead.

Send example commands:

```
ros2 run gz_ros2_control_demos example_gripper
```

Pendulum with passive joints (cart-pole)

The following example shows a cart with a pendulum arm:

```
ros2 launch gz_ros2_control_demos pendulum_example_effort.launch.py
ros2 run gz_ros2_control_demos example_effort
```

This uses the effort command interface for the cart's degree of freedom on the rail. To demonstrate that the physics of the passive joint of the pendulum is solved correctly even with the position command interface, run

```
ros2 launch gz_ros2_control_demos pendulum_example_position.launch.py
ros2 run gz_ros2_control_demos example_position
```

Notes on the command interface

The *gz_ros2_control* plugin receives commands from the ROS2 Control controllers through various command interfaces and applies them to Gazebo simulated joints. Specifically, there are three types of command interfaces. Their current behavior is described below:

- **Effort Command Interface**: The force or torque requested by the controller is applied directly to the joint as is.
- **Velocity Command Interface**: The velocity requested by the controller is applied directly to the joint as is. Note that on some vehicle models, using the velocity command interface to drive the wheels may cause slippage and odometer errors. This is because the wheels are accelerated to the required speed instantaneously, but the chassis cannot reach the same speed immediately.
- **Position Command Interface**: The *gz_ros2_control* plugin controls the velocity of the joints to make them reach the position required by the controller. The velocity is calculated as `joint_velocity = position_proportional_gain * joint_position_error * controller_manager_update_rate`, where `position_proportional_gain` is configurable as described above. For those who are designing control systems: This means that the response of the joint is equivalent to a discrete-time first-order system. The system's time constant is `T = 1 / (position_proportional_gain * controller_manager_update_rate)`. In theory, `position_proportional_gain` cannot be greater than 2 to maintain system stability, and cannot be greater than 1 to avoid oscillations.

6.1.2 MuJoCo

MuJoCo ros2_control Simulation

This package contains a ros2_control system interface for the MuJoCo Simulator. It was originally written for simulating robot hardware in NASA Johnson's iMETRO facility.

The system interface wraps MuJoCo's Simulate App to provide included functionality. Because the app is not bundled as a library, we compile it directly from a local install of MuJoCo.

Parts of this library are also based on the MoveIt mujoco_ros2_control package.

URDF Model Conversion

MuJoCo does not support the full feature set of xacro/URDFs in the ROS 2 ecosystem. As such, users are required to convert any existing robot description files to an MJCF format. This includes adding actuators, sensors, and cameras as needed to the MJCF XML.

This conversion can be done either offline or at run time. We have built a *highly experimental* tool to automate URDF conversion. For more information refer to the documentation.

Hardware Interface Setup

Plugin

This application is shipped as a ros2_control hardware interface, and can be configured as such. Just specify the plugin and point to a valid MJCF on launch:

```
  <ros2_control name="MujocoSystem" type="system">
    <hardware>
      <plugin>mujoco_ros2_control/MujocoSystemInterface</plugin>
      <param name="mujoco_model">$(find my_description)/description/scene.xml</param>

      <!--
       Optional parameter to load the PIDs that can be used with the actuators loaded␣
↪with the MuJoCo model.
       The velocity actuator supports position mode with the PID gains, and the rest␣
↪of the actuation models
       support both position and velocity mode provided the corresponding PID gains.␣
↪The gains should be in ROS
       parameters format to be loaded by the control_toolbox::PidROS class.
        -->
      <param name="pids_config_file">$(find my_description)/config/pids.yaml</param>

      <!--
       Optional parameter to override the speed scaling parameters from the Simulate␣
↪App window
       and just attempt to run at whatever the desired rate is here. This allows␣
↪users to run the simulation
       faster than real time. For example, at 500% speed as set here. If this param␣
↪is omitted or set to
       a value <0, then the simulation will run using the slowdown requested from the␣
↪App.
      -->
      <param name="sim_speed_factor">5.0</param>

      <!--
        Optional parameter to use a particular keyframe already present in the parsed␣
↪MJCF configuration.
        If the key frame is present, it applies it to the MuJoCo state. If not, it␣
↪continues with the default one.
        This parameter will have no affect, if the `override_start_position_file`␣
↪parameter is also set.
        The simulation will instead use the frame from the parsed file.
      -->
      <param name="initial_keyframe">optional_frame</param>
```

(continues on next page)

(continued from previous page)

```
      <!--
        Optional parameter to use the keyframe from a provided file as the starting␣
↪configuration. This is mutually exclusive with
        the initial_value that can be used for state interfaces. This is intended to␣
↪provide an alternative method to load an entire
        MuJoCo model state from a configuration that was saved by clicking 'Copy state
↪' in the simulate window, and pasted into a
        config file. Expected use cases are to work on a specific part of an␣
↪application that involves the environment being in a
        very specific starting configuration. If this parameter is an empty string,␣
↪it will be ignored.
      -->
      <param name="override_start_position_file">$(find my_description)/config/start_
↪positions.xml</param>

      <!--
        Optional parameter to choose a topic name from which it can subscribe and get␣
↪the MJCF contents
        to load the model. This is only used, when the mujoco_model parameter is not␣
↪set.
        By default, this parameter is initialized to /mujoco_robot_description
      -->
      <param name="mujoco_model_topic">/mujoco_robot_description</param>

      <!--
        Optional parameter to update the simulated camera's color and depth image␣
↪publish rates. If no
        parameter is set then all cameras will publish at 5 hz. Note that all cameras␣
↪in the sim currently
        publish at the same intervals.
      -->
      <param name="camera_publish_rate">6.0</param>

      <!--
        Optional parameter to update the simulated lidar sensor's scan message␣
↪publish rates.
        All lidar sensors in the simulation will be configured to publish these scan␣
↪messages at the same rate.
      -->
      <param name="lidar_publish_rate">10.0</param>

      <!--
        The parameter headless can be used to choose whether to launch the MuJoCo␣
↪simulator in headless mode or not.
        By default, it is set to false
      -->
      <param name="headless">false</param>

      <!--
        The optional parameters to choose the name of the floating base joint and the␣
↪topic name to publish the
        odometry of the robot. The provided joint needs to be a valid MuJoCo free␣
↪joint.
      -->
      <param name="odom_free_joint_name">floating_base_joint</param>
      <param name="odom_topic">/simulator/floating_base_state</param>
    </hardware>
```

(continues on next page)

(continued from previous page)

```
  ...
```

Due to compatibility issues, we use a slightly modified ROS 2 control node. It is the same executable and parameters as the upstream, but requires updating the launchfile:

```
    control_node = Node(
        # Specify the control node from this package!
        package="mujoco_ros2_control",
        executable="ros2_control_node",
        output="both",
        parameters=[
            {"use_sim_time": True},
            controller_parameters,
        ],
    )
```

[!NOTE] We can remove the the ROS 2 control node after the next ros2_control upstream release, as the simulation requires this PR to run.

Joints

Joints in the ros2_control interface are mapped to actuators defined in the MJCF, either directly or as transmission interfaces. The system supports different joint control modes based on the actuator type and available command interfaces.

We rely on MuJoCo's PD-level ctrl input for direct position, velocity, or effort control. For velocity, motor, or custom actuators, a position or velocity PID is created if specified using ROS parameters to enable accurate control. Incompatible actuator-interface combinations trigger an error.

Refer to MuJoCo's actuation model for more information.

Of note, only one type of MuJoCo actuator per-joint can be controllable at a time, and the type CANNOT be switched during runtime (i.e., switching from position to motor actuator is not supported). However, the active command interface can be switched dynamically, allowing control to shift between position, velocity, or effort as supported by the actuator type. Users are required to manually adjust actuator types and command interfaces to ensure that they are compatible.

For example a position controlled joint on the MuJoCo

```
  <actuator>
    <position joint="joint1" name="joint1" kp="25000" dampratio="1.0" ctrlrange="0.0␣
↪2.0"/>
  </actuator>
```

Could map to the following hardware interface:

```
  <joint name="joint1">
    <command_interface name="position"/>
    <!-- Initial values for state interfaces can be specified, but default to 0 if␣
↪they are not. -->
    <state_interface name="position">
      <param name="initial_value">0.0</param>
    </state_interface>
    <state_interface name="velocity"/>
    <state_interface name="effort"/>
  </joint>
```

Supported modes between MuJoCo actuators and ros2_control command interfaces:

[!NOTE] The `torque` and `force` command/state interfaces are semantically equivalent to `effort`, and map to the same underlying data in the sim.

Grippers and Mimic Joints

Many robot grippers include mimic joints, where control of a single actuator affects the state of many joints. There are many possible ways to handle this on the MuJoCo side of things, we recommend research and experimentation, or referring to existing models in the wild.

In the current implementation drivers *require* a motor type actuator for joint control and state information. In particular, tendons and other "non-standard" joint types in an MJCF are not directly controllable through the drivers. If mimic joints are required (for instance, parallel jaw mechanisms), we recommend combining tendon actutors with an equality constraint.

For example, from the test robot:

```
  <actuator>
    <position tendon="split" name="gripper_left_finger_joint" kp="1000" dampratio="3.0
↪" ctrlrange="-0.09 0.005"/>
  </actuator>
  <tendon>
    <fixed name="split">
      <joint joint="gripper_left_finger_joint" coef="0.5"/>
      <joint joint="gripper_right_finger_joint" coef="-0.5"/>
    </fixed>
  </tendon>
  <equality>
    <joint joint1="gripper_left_finger_joint" joint2="gripper_right_finger_joint"␣
↪polycoef="0 -1 0 0 0" solimp="0.95 0.99 0.001" solref="0.005 1"/>
  </equality>
```

Note that the tendon name *matches* the controllable joint in the ros2_control configuration. This way, the drivers will only provide control and state information from that single joint, but the underlying simulation will ensure that the right finger's position matches the left's.

Sensors

The hardware interfaces supports force-torque sensors (FTS) and inertial measurement units (IMUs). MuJoCo does not support modeling complete FTS and IMUs out of the box, so we combine supported MJCF constructs to map to a ros2_control sensor. The types and other parameters can be specified in the ros2_control xacro, as noted below.

For FTS, we model the `force` and `torque` sensors individually in the MJFC. For a sensor named `fts_sensor`, we suffix each entry accordingly as `fts_sensor_force` and `fts_sensor_torque`. For example, in the MJCF,

```
  <sensor>
    <force name="fts_sensor_force" site="ft_frame"/>
    <torque name="fts_sensor_torque" site="ft_frame"/>
  </sensor>
```

In the corresponding ros2_control xacro, this becomes a single sensor:

```
  <sensor name="fts_sensor">
    <param name="mujoco_type">fts</param>
    <!-- There is no requirement for the mujoco_sensor_name to match the ros2_control␣
↪sensor name -->
```

(continues on next page)

(continued from previous page)

```
    <param name="mujoco_sensor_name">fts_sensor</param>
    <!-- Default value of force_mjcf_suffix is '_force' -->
    <param name="force_mjcf_suffix">_force</param>
    <!-- Default value of torque_mjcf_suffix is '_torque' -->
    <param name="torque_mjcf_suffix">_torque</param>
    <state_interface name="force.x"/>
    <state_interface name="force.y"/>
    <state_interface name="force.z"/>
    <state_interface name="torque.x"/>
    <state_interface name="torque.y"/>
    <state_interface name="torque.z"/>
  </sensor>
```

Similarly, for an IMU, we simulate a `framequat`, `gyro`, and `accelerometer` as a single IMU.

```
  <sensor>
      <framequat name="imu_sensor_quat" objtype="site" objname="imu_sensor" />
      <gyro name="imu_sensor_gyro" site="imu_sensor" />
      <accelerometer name="imu_sensor_accel" site="imu_sensor" />
  </sensor>
```

Which then map to the corresponding ros2_control sensor:

```
  <sensor name="imu_sensor">
    <param name="mujoco_type">imu</param>
    <!-- There is no requirement for the mujoco_sensor_name to match the ros2_control␣
↪sensor name -->
    <param name="mujoco_sensor_name">imu_sensor</param>
    <!-- Default value of orientation_mjcf_suffix is '_quat' -->
    <param name="orientation_mjcf_suffix">_quat</param>
    <!-- Default value of angular_velocity_mjcf_suffix is '_gyro' -->
    <param name="angular_velocity_mjcf_suffix">_gyro</param>
    <!-- Default value of linear_acceleration_mjcf_suffix is '_accel' -->
    <param name="linear_acceleration_mjcf_suffix">_accel</param>
    <state_interface name="orientation.x"/>
    <state_interface name="orientation.y"/>
    <state_interface name="orientation.z"/>
    <state_interface name="orientation.w"/>
    <state_interface name="angular_velocity.x"/>
    <state_interface name="angular_velocity.y"/>
    <state_interface name="angular_velocity.z"/>
    <state_interface name="linear_acceleration.x"/>
    <state_interface name="linear_acceleration.y"/>
    <state_interface name="linear_acceleration.z"/>
  </sensor>
```

These sensor state interfaces can then be used out of the box with the standard broadcasters.

Cameras

Any `camera` included in the MJCF will automatically have its RGB-D images and info published to ROS topics. Currently all images are published at a fixed 5hz rate.

Cameras must include a string `<name>`, which sets defaults for the frame and topic names. By default, the ROS 2 wrapper assumes the camera is attached to a frame named `<name>_frame`. Additionally camera_info, color, and depth images will be published to topics called `<name>/camera_info`, `<name>/color`, and `<name>/depth`, respectively. Also note that MuJuCo's conventions for cameras are different than ROS's, and which must be accounted for. Refer to the documentation for more information.

For example,

```
<camera name="wrist_mounted_camera" fovy="58" mode="fixed" resolution="640 480" pos=
↪"0 0 0" quat="0 0 0 1"/>
```

Will publish the following topics:

```
$ ros2 topic info /wrist_mounted_camera/camera_info
Type: sensor_msgs/msg/CameraInfo
$ ros2 topic info /wrist_mounted_camera/color
Type: sensor_msgs/msg/Image
$ ros2 topic info /wrist_mounted_camera/depth
Type: sensor_msgs/msg/Image
```

The frame and topic names are also configurable from the ros2_control xacro. Default parameters can be overridden with:

```
  <!-- For cameras, the sensor name _must_ match the camera name in the MJCF -->
  <sensor name="wrist_mounted_camera">
    <param name="frame_name">wrist_mounted_camera_mujoco_frame</param>
    <param name="info_topic">/wrist_mounted_camera/color/camera_info</param>
    <param name="image_topic">/wrist_mounted_camera/color/image_raw</param>
    <param name="depth_topic">/wrist_mounted_camera/aligned_depth_to_color/image_raw</
↪param>
  </sensor>
```

Lidar

MuJoCo does not include native support for lidar sensors. However, this package offers a ROS 2-like lidar implementation by wrapping sets of rangefinders together.

MuJoCo rangefinders measure the distance to the nearest surface along the positive `Z` axis of the sensor site. The ROS 2 lidar wrapper uses the standard defined in LaserScan messages. In particular, the first rangefinder's `Z` axis (e.g. `rf-00`) must align with the ROS 2 lidar sensor's positive `X` axis.

In the MJCF, use the `replicate` tag along with a `-` separator to add N sites to attach sensors to. For example, the following will add 12 sites named `rf-00` to `rf-11` each at a 0.025 radian offset from each other:

```
  <replicate count="12" sep="-" offset="0 0 0" euler="0 0.025 0">
    <site name="rf" size="0.01" pos="0.0 0.0 0.0" quat="0.0 0.0 0.0 1.0"/>
  </replicate>
```

Then a set of rangefinders can be attached to each site with:

```
  <sensor>
    <!-- We require a sensor name be provided -->
    <rangefinder name="lidar" site="rf" />
  </sensor>
```

The lidar sensor is then configurable through ROS 2 control xacro with:

```
    <!-- Lidar sensors are matched to a set of rangefinder sensors in the MJCF, which␣
↪should be -->
    <!-- generated with "replicate" and will generally be of the form "<sensor_name>-
↪01". -->
    <!-- We assume the lidar sensor starts at angle 0, increments by the specified␣
↪`angle_increment`, and -->
    <!-- that there are exactly `num_rangefinders` all named from <sensor_name>-000␣
↪to the max -->
    <!-- <sensor_name>-<num_rangefinders> -->
    <sensor name="lidar">
      <param name="frame_name">lidar_sensor_frame</param>
      <param name="angle_increment">0.025</param>
      <param name="num_rangefinders">12</param>
      <param name="range_min">0.05</param>
      <param name="range_max">10</param>
      <param name="laserscan_topic">/scan</param>
    </sensor>
  </ros2_control>
```

Simulation - Topics and Services

Topics

- `/mujoco_actuators_states` (`sensor_msgs/msg/JointState`): Provides information on all internal MuJoCo joints, regardless of whether their interfaces are exposed via `ros2_control`.
- `/clock` (`rosgraph_msgs/msg/Clock`): Contains the internal physics clock tracked by each MuJoCo simulation step.

Services

- `~/set_pause` (`mujoco_ros2_control_msgs/srv/SetPause`): Pauses or resumes the simulation.
  - Set `paused` to `true` to pause, or `false` to resume.
  - Returns immediately — no blocking. Returns `success = true` even if the simulation is already in the requested state.
  - When resuming, the physics loop automatically re-syncs its wall-clock reference so no catch-up steps are executed.

```
# Pause the simulation
ros2 service call /ros2_control_node/set_pause mujoco_ros2_control_msgs/srv/
↪SetPause "{paused: true}"

# Resume the simulation
ros2 service call /ros2_control_node/set_pause mujoco_ros2_control_msgs/srv/
↪SetPause "{paused: false}"
```

- `~/reset_world` (`mujoco_ros2_control_msgs/srv/ResetWorld`): Resets the simulation state.
  - If the optional `keyframe` string field is empty, the simulation is restored to the state captured at startup (initial joint positions, velocities, and control values).
  - If a `keyframe` name is provided, that named keyframe from the MJCF is applied instead.
  - Returns `success` and a human-readable `message`.

```
# Reset to startup state
ros2 service call /ros2_control_node/reset_world mujoco_ros2_control_msgs/srv/
↪ResetWorld "{}"

# Reset to a named MJCF keyframe
ros2 service call /ros2_control_node/reset_world mujoco_ros2_control_msgs/srv/
↪ResetWorld "{keyframe: 'home'}"
```

    [!IMPORTANT] If controllers are active during the service call, the robot may reset to the initial state and then immediately snap back to its previous commanded position. To avoid this, it is recommended to deactivate any active joint controllers before calling this service.

- `~/step_simulation` (`mujoco_ros2_control_msgs/srv/StepSimulation`): Advances the paused simulation by an exact number of physics steps and blocks until all steps have completed.
  - The `steps` field (`uint32`) specifies how many physics steps to execute. Must be ≥ 1.
  - The call returns only after all requested steps are finished (or a timeout/divergence is detected).
  - Returns `success` and a human-readable `message`.
  - **The simulation must be paused** before calling this service. If `sim_->run` is true the call returns immediately with `success = false`.
  - Timeout: whichever is larger — 30 s, or 10 ms × `steps`.

```
# Step the simulation forward by 100 physics steps
ros2 service call /ros2_control_node/step_simulation mujoco_ros2_control_msgs/srv/
↪StepSimulation "{steps: 100}"
```

Debugging

The simulator provides several mechanisms for pausing execution and advancing it in a controlled, step-by-step fashion. This is useful for inspecting robot state, verifying controller output, or reproducing intermittent issues.

Pausing the Simulation

Click the **Pause** button in the MuJoCo Simulate window (or press **Space**) to pause the physics loop. When paused, the simulation clock stops advancing and no physics steps are executed until explicitly requested.

Single-Stepping via the Keyboard

While the simulation window is focused and the simulation is **paused**, press the **right arrow key** (→) to advance the simulation by exactly one physics step. Holding the key down will advance the simulation continuously one step at a time, allowing slow, frame-by-frame inspection of the robot's motion.

The status overlay in the top-right corner of the simulation window shows the current state (`Running` / `Paused`) and the total number of physics steps executed.

Single-Stepping via ROS 2 Service

The `~/step_simulation` service allows programmatic step-by-step control from the command line or from test/debug scripts. This is particularly useful for automated testing scenarios where a reproducible sequence of physics steps is needed.

```
# Pause the simulation first (from the UI or via another mechanism), then:

# Advance by a single physics step
ros2 service call /ros2_control_node/step_simulation mujoco_ros2_control_msgs/srv/
↪StepSimulation "{steps: 1}"

# Advance by 500 steps (blocks until complete)
ros2 service call /ros2_control_node/step_simulation mujoco_ros2_control_msgs/srv/
↪StepSimulation "{steps: 500}"
```

The service call **blocks** until all requested steps have been executed, the simulation diverges, or a timeout is reached. This makes it safe to pipeline service calls sequentially without additional synchronisation (e.g. send a command → step N times → read state → repeat).

> [!NOTE] `~/step_simulation` requires the simulation to be **paused**. Calling it while the simulation is running returns `success: false` immediately without executing any steps.

Test Robot System

While examples are limited, we maintain a functional example 2-dof robot system in the demos space (see `mujoco_ros2_control_demos` package). We generally recommend looking there for examples and recommended workflows.

Development

More information is provided in the developers guide document (see `docs/DEVELOPMENT.md` in the mujoco_ros2_control repository).

6.1.3 topic_based_hardware_interfaces

Joint State Topic Based System

The Joint State Topic Based System implements a ros2_control `hardware_interface::SystemInterface` supporting command and state interfaces through the ROS topic communication layer.

ros2_control urdf tag

The `joint_state_topic_hardware_interface` has a few `ros2_control` urdf tags to customize its behavior.

Parameters

- joint_commands_topic: (default: "/robot_joint_command"). Example: `<param name="joint_commands_topic">/my_topic_joint_commands</param>`.
- joint_states_topic: (default: "/robot_joint_states"). Example: `<param name="joint_states_topic">/my_topic_joint_states</param>`.
- trigger_joint_command_threshold: (default: 1e-5). Used to avoid spamming the joint command topic when the difference between the current joint state and the joint command is smaller than this value, set to -1 to always send the joint command. Example: `<param name="trigger_joint_command_threshold">0.001</param>`.
- sum_wrapped_joint_states: (default: "false"). Used to track the total rotation for joint states the values reported on the `joint_commands_topic` wrap from 2*pi to -2*pi when rotating in the positive direction. (Isaac Sim only reports joint states from 2*pi to -2*pi) Example: `<param name="sum_wrapped_joint_states">true</param>`.

Per-joint Parameters

- mimic: Defined name of the joint to mimic. This is often used concept with parallel grippers. Example: `<param name="mimic">joint1</param>`.
- multiplier: Multiplier of values for mimicking joint defined in mimic parameter. Example: `<param name="multiplier">-2</param>`.

Modifying the urdf `ros2_control` tag for new robots

If your robot description support mock_components you simply add an if-else statement to switch between it and `<plugin>joint_state_topic_hardware_interface/JointStateTopicSystem</plugin>`, make sure to add the `joint_commands_topic` and `joint_states_topic` to point to the correct topics.

```
        <ros2_control name="name" type="system">
            <hardware>
              <xacro:if value="${mock_hardware}">
                <plugin>mock_components/GenericSystem</plugin>
                <param name="fake_sensor_commands">false</param>
                <param name="state_following_offset">0.0</param>
                <param name="calculate_dynamics">true</param>
              </xacro:if>
```

(continues on next page)

(continued from previous page)

```
            <xacro:unless value="${mock_hardware}">
              <plugin>joint_state_topic_hardware_interface/JointStateTopicSystem</
↪plugin>
              <param name="joint_commands_topic">/topic_based_joint_commands</param>
              <param name="joint_states_topic">/topic_based_joint_states</param>
              <param name="sum_wrapped_joint_states">true</param>
            </xacro:if>
          </hardware>
          <joint name="joint_1">
              <command_interface name="position"/>
              <state_interface name="position">
                <param name="initial_value">0.0</param>
              </state_interface>
              <state_interface name="velocity"/>
          </joint>
          ...
          <joint name="joint_n">
              <command_interface name="position"/>
              <state_interface name="position">
                <param name="initial_value">0.0</param>
              </state_interface>
              <state_interface name="velocity"/>
          </joint>
          ...
          <joint name="mimic_joint_1">
              <param name="mimic">joint_k</param>
              <param name="multiplier">1</param>
              <command_interface name="position" />
              <state_interface name="position">
                <param name="initial_value">0.0</param>
              </state_interface>
              <state_interface name="velocity"/>
          </joint>
          ...
          <joint name="mimic_joint_n">
              <param name="mimic">joint_kn</param>
              <param name="multiplier">1</param>
              <command_interface name="position" />
              <state_interface name="position">
                <param name="initial_value">0.0</param>
              </state_interface>
              <state_interface name="velocity"/>
          </joint>
      </ros2_control>
```

cm_topic_hardware_component

The controller_manager topic System implements a ros2_control `hardware_interface::SystemInterface` that subscribes to topics of type `pal_statistics_msgs::msg::StatisticsNames` and `pal_statistics_msgs::msg::StatisticsValues`, and sets its state interface to the received values (if present).

Per default, the ros2_control controller manager publishes these topics to `/controller_manager/introspection_data/names` and `/controller_manager/introspection_data/values`.

This component serves as a possibility to replay ROS bags and inject the states from a hardware component into the

ros2_control stack. For example, use `ros2bag` CLI to extract these two topics via

```
ros2 bag play <bag_file> \
  --topics /controller_manager/introspection_data/names /controller_manager/
↪introspection_data/values \
  --remap /controller_manager/introspection_data/names:=/<hardware_component_name>/
↪names\
          /controller_manager/introspection_data/values:=/<hardware_component_name>/
↪values
```

Note that with this setup the current time of your OS is used and not published from the ROS bag. If you want to control the speed of playback, run

- your ros2_control_node with `--ros-args -p --use_sim_time:=true`
- and the `--rate` and `--clock` options, for example

```
ros2 bag play <bag_file> --rate 2.0 --clock 100 \
  --topics /controller_manager/introspection_data/names /controller_manager/
↪introspection_data/values \
  --remap /controller_manager/introspection_data/names:=/<hardware_component_name>
↪/names\
          /controller_manager/introspection_data/values:=/<hardware_component_
↪name>/values
```

ROS subscribers

- /<hardware_component_name>/names: `pal_statistics_msgs::msg::StatisticsNames`
- /<hardware_component_name>/values: `pal_statistics_msgs::msg::StatisticsValues`

ros2_control section in URDF

```
<ros2_control name="hardware_component_name" type="system">
  <hardware>
    <plugin>cm_topic_hardware_component/CMTopicSystem</plugin>
  </hardware>
  <joint name="joint_1">
    <command_interface name="position"/>
    <state_interface name="position">
      <param name="initial_value">0.2</param>
    </state_interface>
    <state_interface name="velocity"/>
  </joint>
  <joint name="joint_2">
    <command_interface name="position"/>
    <state_interface name="position">
      <param name="initial_value">0.3</param>
    </state_interface>
    <state_interface name="velocity"/>
  </joint>
  <joint name="joint_3">
    <command_interface name="position"/>
    <state_interface name="position">
      <param name="initial_value">0.1</param>
```

(continues on next page)

(continued from previous page)

```
    </state_interface>
    <state_interface name="velocity"/>
  </joint>
</ros2_control>
```

6.2 Community

This page hosts a list of supported simulators with *ros2_control* integration. To add your *ros2_control* integration, submit a PR to this page on Github!

- Isaac Sim
- Webots
- MuJoCo (FZI)
- Algoryx AGX Dynamic

CHAPTER

SEVEN

RELEASE NOTES

This page should highlight the most important changes of the ros2_control framework in the rolling release. If you are skipping a distribution update, make sure to read the release notes of all intermediate distributions.

These lists might not be complete, see the `CHANGELOG.rst` files in the respective repositories for a full list of changes.

For necessary changes to your code for a version update, see the *Migration Guides*.

7.1 Release Notes: Kilted Kaiju to Lyrical Luth

This list summarizes important changes between Kilted Kaiju (previous) and Lyrical Luth (current) releases.

7.1.1 controller_interface

- The new `MagneticFieldSensor` semantic component provides an interface for reading data from magnetometers. (#2627)
- The controller_manager will now deactivate the entire controller chain if a controller in the chain fails during the update cycle. (#2681)
- The update rate of the controller will now be approximated to a closer achievable frequency, when its frequency is not achievable with the current controller manager update rate. (#2828)
- The lifecycle ID is cached internally in the controller to avoid calls to get_lifecycle_state() in the real-time control loop. (#2884)
- Added 2 new interface_configuration_types: `INDIVIDUAL_BEST_EFFORT` and `REGEX`. These allow for more flexible controller interface configurations. (#2902)
- Added new methods `on_export_state_interfaces_list` and `on_export_reference_interfaces_list` are added exporting the interface pointers for chainable controller. (#2988)

7.1.2 controller_manager

- The `bcolors` class now respects the `RCUTILS_COLORIZED_OUTPUT` environment variable to automatically disable colors in non-TTY and CI environments.
- The default strictness for `switch_controller` is changed to `strict`. (#2742)
- A new parameter `handle_exceptions` is added to the controller manager to control whether exceptions thrown by controllers during update are caught and handled internally or propagated. (#2807)

- The `spawner` now supports per controller arguments, while parsing the arguments for multiple controllers using `--controller` option. (#2895)
- Added new `cleanup_controller` service to the controller manager to allow cleaning up controllers from external clients. (#2414)
- Removed forwarding of the controller manager's ros arguments to the controllers via NodeOptions. (#3016)
- The `spawner` now forwards all the parameter files parsed to the spawner node to the spawned controllers. This would support `allow_substs` approach. (#3136)

7.1.3 hardware_interface

- The lifecycle ID is cached internally in the controller to avoid calls to get_lifecycle_state() in the real-time control loop. (#2884)
- Handles now also support `float32`, `uint8`, `int8`, `uint16`, `int16`, `uint32`, `int32` data types in addition to double and bool. (#2879)

7.1.4 ros2controlcli

- Added CLI support for invoking controller cleanup. (#2414)

7.1.5 transmission_interface

- The `simple_transmission` and `differential_transmission` now also support the `force` interface (#2588).

7.2 Release Notes: Kilted Kaiju to Lyrical Luth

This list summarizes important changes between Kilted Kaiju (previous) and Lyrical Luth (current) releases.

7.2.1 state_interfaces_broadcaster

- The state_interfaces_broadcaster was added (#2006).

7.2.2 force_torque_sensor_broadcaster

- Added support for transforming Wrench messages to a given list of target frames. This is useful when applications need force/torque data in their preferred coordinate frames. (#2021).

7.2.3 diff_drive_controller

- Parameter `tf_frame_prefix_enable` got deprecated and will be removed in a future release (#1997).
- Now any tilde ("~") character in `tf_frame_prefix` is substituted with node namespace. (#1997).
- Set odometry service added to be used at runtime. (#2096).

7.2.4 mecanum_drive_controller

- Parameter `tf_frame_prefix_enable` got deprecated and will be removed in a future release (#2063).
- Now any tilde ("~") character in `tf_frame_prefix` is substituted with node namespace. (#2063).
- Set odometry service added to be used at runtime. (#2110).

7.2.5 joint_state_broadcaster

- Make all parameters read-only (the never got re-evaluated after initialization anyways). (#2064)
- Added parameter `publish_dynamic_joint_states` to enable/disable publishing of dynamic joint states. (#2064)
- Removed interfaces with other data types than double for publishing to `dynamic_joint_states`. (#2115)
- Parameter `publish_dynamic_joint_states` is now deprecated (default changed to `false`).

7.2.6 omni_wheel_drive_controller

- Parameter `tf_frame_prefix_enable` got deprecated and will be removed in a future release (#2073).
- Now any tilde ("~") character in `tf_frame_prefix` is substituted with node namespace. (#2073).
- Set odometry service added to be used at runtime. (#2148).

7.2.7 joint_trajectory_controller

- Fill in 0 velocities and accelerations into point before trajectories if the state interfaces don't contain velocity / acceleration information, but the trajectory does. This way, the segment up to the first waypoint will use the same interpolation as the rest of the trajectory. (#2043)
- Added decelerate-to-stop functionality when a trajectory is canceled or preempted. Instead of immediately holding position, the controller can now smoothly decelerate each joint to a stop using the per-joint `max_deceleration_on_cancel` parameter. (#2163)

7.2.8 pid_controller

- Added parameter `set_current_state_as_first_setpoint` (default: true) to set the current state as the first setpoint when the controller is activated, helping to avoid large initial errors and sudden jumps in control output.

7.2.9 steering_controllers_library

- Parameter `tf_frame_prefix` added with the similar functionality to other controllers. (#2080).
- Set odometry service added to be used at runtime. (#2244).

7.3 Release Notes: Jazzy to Kilted

This list summarizes the changes between Jazzy (previous) and Kilted (current) releases.

7.3.1 Pid/PidROS

- Added a saturation feature to PID output and two anti-windup techniques (back calculation and conditional integration) (#298).
- Added a constructor argument to `PidROS` to control if the PID state publisher is initially active or not. Can be changed during runtime by using `activate_state_publisher` parameter. (#431).

7.4 Release Notes: Kilted Kaiju to Lyrical Luth

This list summarizes important changes between Kilted Kaiju (previous) and Lyrical Luth (current) releases.

CHAPTER

EIGHT

MIGRATION GUIDES

8.1 Coming from ros_control (ROS 1)

8.1.1 Differences to ros_control (ROS 1)

Hardware Structures - classes

The ros_control framework uses the `RobotHW` class as a rigid structure to handle any hardware. This makes it impossible to extend the existing robot with additional hardware, like sensors, actuators, and tools, without coding. The `CombinedRobotHardware` corrects this drawback. Still, this solution is not optimal, especially when combining robots with external sensors.

The ros2_control framework defines three types of hardware `Actuator`, `Sensor` and `System`. Using a combination (composition) of those basic components, any physical robotic cell (robot and its surrounding) can be described. This also means that multi-robot, robot-sensor, robot-gripper combinations are supported out of the box. Section *Hardware Components* describes this in detail.

Hardware Interfaces

The ros_control framework allows only three types of interfaces (joints), i.e., `position`, `velocity`, and `effort`. The `RobotHW` class makes it very hard to use any other data to control the robot.

The ros2_control approach does not enforce a fixed set of interface types, but they are defined as strings in *hardware's description*. To ensure compatibility of standard controllers, standard interfaces are defined as constants in hardware_interface package.

Controller's Access to Hardware

In ros_control, the controllers had direct access to the `RobotHW` class requesting access to its interfaces (joints). The hardware itself then took care of registered interfaces and resource conflicts.

In ros2_control, `ResourceManager` takes care of the state of available interfaces in the framework and enables controllers to access the hardware. Also, the controllers do not have direct access to hardware anymore, but they register their interfaces to the `ControllerManager`.

8.1.2 Migration Guide to ros2_control

RobotHardware to Components

1. The implementation of `RobotHW` is not used anymore. This should be migrated to SystemInterface class or, for more granularity, SensorInterface and ActuatorInterface. See the above description of "Hardware Components" to choose a suitable strategy.
2. Decide which component type is suitable for your case. Maybe it makes sense to separate `RobotHW` into multiple components.
3. Implement ActuatorInterface, SensorInterface or SystemInterface classes as follows:
    1. In the constructor, initialize all variables needed for communication with your hardware or define the default one.
    2. In the configure function, read all the parameters your hardware needs from the parsed URDF snippet (i.e., from the HardwareInfo structure). Here you can cross-check if all joints and interfaces in URDF have allowed values or a combination of values.
    3. Define interfaces to and from your hardware using `export_*_interfaces` functions. The names are `<joint>/<interface>` (e.g., `joint_a2/position`). This can be extracted from the HardwareInfo structure or be hard-coded if sensible.
    4. Implement `start()` and `stop()` methods for your hardware. This usually includes changing the hardware state to receive commands or set it into a safe state before interrupting the command stream. It can also include starting and stopping communication.
    5. Implement `read()` and `write()` methods to exchange commands with the hardware. This method is equivalent to those from `RobotHW`-class in ROS 1.
    6. Do not forget the `PLUGINLIB_EXPORT_CLASS` macro at the end of the .cpp file.
4. Create .xml library description for the pluginlib, for example see RRBotSystemPositionOnlyHardware.

Controller Migration

An excellent example of a migrated controller is the JointTrajectoryController. The real-time critical methods are marked as such.

1. Implement ControllerInterface class as follows:
    1. If there are any member variables, initialized those in the constructor.
    2. In the `init()` method, first call `ControllerInterface::init()` to initialize the lifecycle of the controller. Following this, declare all parameters defining their default values.
    3. Implement the `state_interface_configuration()` and `command_interface_configuration()` methods.
    4. Design the `update()` function for the controller. (**real-time**)
    5. **Add the required lifecycle management methods (others are optional):**
        - `on_configure()` - reads parameters and configures controller.
        - `on_activate()` - called when controller is activated (started) (**real-time**)
        - `on_deactivate()` - called when controller is deactivated (stopped) (**real-time**)
    6. Finally, do not forget to add the `PLUGINLIB_EXPORT_CLASS` macro at the end of the .cpp file.
2. Create .xml library description for the pluginlib, for example see joint_trajectory_plugin.xml.

8.2 Between different ROS 2 distributions

This list summarizes necessary changes to your code for a version update to rolling. If you are skipping a distribution update, make sure to read the migration guides of all intermediate distributions.

For non-breaking updates, see the *Release Notes*.

8.2.1 Migration Guides: Kilted Kaiju to Lyrical Luth

This list summarizes important changes between Kilted Kaiju (previous) and Lyrical Luth (current) releases, where changes to user code might be necessary.

controller_interface

ChainableControllerInterface

- The `on_export_state_interfaces()` method is deprecated and replaced by `on_export_state_interfaces_list()` (#2988). The new method returns shared pointers instead of objects by value:

```
// Old (deprecated)
std::vector<hardware_interface::StateInterface> on_export_state_interfaces()

// New
std::vector<hardware_interface::StateInterface::SharedPtr> on_export_state_
↪interfaces_list()
```

Example migration:

```
// Old implementation
std::vector<hardware_interface::StateInterface>
MyController::on_export_state_interfaces()
{
  std::vector<hardware_interface::StateInterface> state_interfaces;
  state_interfaces.emplace_back(
    std::string(get_node()->get_name()) + "/my_state", "position", &my_state_
↪value_);
  return state_interfaces;
}

// New implementation
std::vector<hardware_interface::StateInterface::SharedPtr>
MyController::on_export_state_interfaces_list()
{
  std::vector<hardware_interface::StateInterface::SharedPtr> state_interfaces;
  auto state_interface = std::make_shared<hardware_interface::StateInterface>(
    std::string(get_node()->get_name()) + "/my_state", "position");
  state_interface->set_value(std::numeric_limits<double>::quiet_NaN());
  state_interfaces.push_back(state_interface);
  return state_interfaces;
}
```

- The `on_export_reference_interfaces()` method is deprecated and replaced by `on_export_reference_interfaces_list()` (#2988). The new method returns shared pointers instead of objects by value:

```
// Old (deprecated)
std::vector<hardware_interface::CommandInterface> on_export_reference_interfaces()

// New
std::vector<hardware_interface::CommandInterface::SharedPtr> on_export_reference_
↪interfaces_list()
```

Example migration:

```
// Old implementation
std::vector<hardware_interface::CommandInterface>
MyController::on_export_reference_interfaces()
{
  reference_interfaces_.resize(1, std::numeric_limits<double>::quiet_NaN());
  std::vector<hardware_interface::CommandInterface> reference_interfaces;
  reference_interfaces.emplace_back(
    std::string(get_node()->get_name()) + "/my_ref", "velocity", &reference_
↪interfaces_[0]);
  return reference_interfaces;
}

// New implementation
std::vector<hardware_interface::CommandInterface::SharedPtr>
MyController::on_export_reference_interfaces_list()
{
  std::vector<hardware_interface::CommandInterface::SharedPtr> reference_
↪interfaces;
  auto cmd_interface = std::make_shared<hardware_interface::CommandInterface>(
    std::string(get_node()->get_name()) + "/my_ref", "velocity");
  cmd_interface->set_value(std::numeric_limits<double>::quiet_NaN());
  reference_interfaces.push_back(cmd_interface);
  return reference_interfaces;
}
```

- The exported state interfaces are now returned as `ConstSharedPtr` from `export_state_interfaces()` to ensure they are read-only for consumers (#1767).
- The internal storage variables will be removed in upcoming releases. Controllers should now use the ordered exported interface containers (`ordered_exported_state_interfaces_` and `ordered_exported_reference_interfaces_`) which store shared pointers instead of raw values (#2988).
- The controller manager's ros arguments are no longer forwarded to the controllers via NodeOptions. (#3016) So, any remapping done at the controller manager level will not be visible to the controllers anymore. It is recommended to use the `--controller-ros-args` option of the spawner to pass ros arguments to controllers.

```
control_node = Node(
    package="controller_manager",
    executable="ros2_control_node",
    parameters=[robot_controllers],
    remappings=[("/diffbot_base_controller/cmd_vel", "/cmd_vel")],
    output="both",
)
```

to

```
control_node = Node(
    package="controller_manager",
    executable="ros2_control_node",
```

(continues on next page)

(continued from previous page)

```
    parameters=[robot_controllers],
    output="both",
)
spawner_node = Node(
    package="controller_manager",
    executable="spawner",
    arguments=[
        "diffbot_base_controller",
        "--controller-ros-args",
        "--remap",
        "/diffbot_base_controller/cmd_vel:=/cmd_vel",
    ],
)
```

controller_manager

hardware_interface

- The signature for the `on_init` method in all `hardware_interface::*Interface` classes has changed (#2323, #2589) from

```
CallbackReturn on_init(const hardware_interface::HardwareInfo& info)
```

to

```
CallbackReturn on_init(const HardwareComponentInterfaceParams& params)
```

The `HardwareInfo` object can be accessed from the `HardwareComponentInterfaceParams` object using `params.hardware_info`. See *Writing a Hardware Component* for advanced usage of the `HardwareComponentInterfaceParams` object.

- The signature for the `init()` method in all `hardware_interface::*Interface` classes has changed (#2344, #2589) from

```
CallbackReturn init(const HardwareInfo & hardware_info, rclcpp::Logger logger,␣
↪rclcpp::Clock::SharedPtr clock)
```

to

```
CallbackReturn init(const hardware_interface::HardwareComponentParams & params)
```

- The `initialize` methods of all hardware components (such as `Actuator`, `Sensor`, etc.) have been changed from passing a `const HardwareInfo &` to passing a `const HardwareComponentParams &` (#2323, #2589).
- The `get_value` of LoanedStateInterface and LoanedCommandInterface is now accessed using `get_optional` method. The value will be returned as an `std::optional<T>`. (#2061).

  This change was made to better handle cases where the interface value may not be accessible due to a concurrent access from other threads in the system.
- The `double get_value()` of standard StateInterface and CommandInterface is now accessed using `get_optional` or `bool get_value(T & value, bool wait_for_lock)` method. The value will be returned as an `std::optional<T>` when using `get_optional` (#2831).

Likewise, the `set_value` method has been updated to `bool set_value(const T & value, bool wait_for_lock)` and return value is to indicate success or failure of the operation (#2831).

You can use the return values of these methods to handle cases where the interface value may not be accessible due to a concurrent access from other threads in the system. You can set the `wait_for_lock` parameter to `true` to block until the lock is acquired, however, this is not real-time safe and should be used with caution in real-time contexts.

8.2.2 Migration Guides: Kilted Kaiju to Lyrical Luth

This list summarizes important changes between Kilted Kaiju (previous) and Lyrical Luth (current) releases, where changes to user code might be necessary.

joint_state_broadcaster

- Removed interfaces with other data types than double for publishing to `dynamic_joint_states`. (#2115). Use a custom controller for publishing non-double interfaces.
- Parameter `publish_dynamic_joint_states` is now deprecated (default changed to `false`). (#2107) For publishing non-standard interfaces, consider using alternatives:
  - *state_interfaces_broadcaster* for broadcasting arbitrary state interfaces
  - *gpio_controllers* for GPIO and custom hardware interfaces
  - pal_statistics for flexible runtime statistics publishing

effort_controllers

- `effort_controllers/JointGroupEffortController` is deprecated. Use *forward_command_controller* instead by adding the `interface_name` parameter and set it to `effort`. (#1913).

position_controllers

- `position_controllers/JointGroupPositionController` is deprecated. Use *forward_command_controller* instead by adding the `interface_name` parameter and set it to `position`. (#1913).

velocity_controllers

- `velocity_controllers/JointGroupVelocityController` is deprecated. Use *forward_command_controller* instead by adding the `interface_name` parameter and set it to `velocity`. (#1913).

diff_drive_controller

- Instead of using `tf_frame_prefix_enable:=false`, set an empty `tf_frame_prefix:=""` parameter instead. (#1997).
- For using node namespace as tf prefix: Set `tf_frame_prefix:="~"`, where the ("~") character is substituted with node namespace. (#1997).

mecanum_drive_controller

- Instead of using `tf_frame_prefix_enable:=false`, set an empty `tf_frame_prefix:=""` parameter instead. (#1997).
- For using node namespace as tf prefix: Set `tf_frame_prefix:="~"`, where the ("~") character is substituted with node namespace. (#1997).

omni_wheel_drive_controller

- Instead of using `tf_frame_prefix_enable:=false`, set an empty `tf_frame_prefix:=""` parameter instead. (#2073).
- For using node namespace as tf prefix: Set `tf_frame_prefix:="~"`, where the ("~") character is substituted with node namespace. (#2073).

8.2.3 Migration Guides: Jazzy to Kilted

This list summarizes important changes between Jazzy (previous) and Kilted (current) releases, where changes to user code might be necessary.

Pid/PidROS

- The parameters `antiwindup`, `i_clamp_max`, and `i_clamp_min` have been removed. The anti-windup behavior is now configured via the `AntiWindupStrategy` enum. (#298).
- The constructors of `PidROS` have changed and `prefix_is_for_params` argument has been deprecated. Use explicit parameter and topic prefix instead (#431).

8.2.4 Migration Guides: Kilted Kaiju to Lyrical Luth

This list summarizes important changes between Kilted Kaiju (previous) and Lyrical Luth (current) releases, where changes to user code might be necessary.

RealtimeBox

- `RealtimePublisher` is updated with a new `try_publish` API.
  - Update your code with a local message variable and call `try_publish` with that variable. (#323).
  - `msg_` variable is inaccessible now (#421).

CHAPTER

NINE

API DOCUMENTATION

9.1 ros2_control stack

API documentation for the whole framework is parsed by doxygen and can be found here.

9.2 Per-Package API Documentation

In the following, you can find links to the per-package API documentation published on docs.ros.org.

9.2.1 ros2_control

Package Name	API	ROS Index
controller_interface	API	ROS Index
controller_manager	API	ROS Index
controller_manager_msgs	API	ROS Index
hardware_interface	API	ROS Index
ros2_control_test_assets	API	ROS Index
transmission_interface	API	ROS Index

9.2.2 ros2_controllers

Package Name	API	ROS Index
ackermann_steering_controller	API	ROS Index
admittance_controller	API	ROS Index
bicycle_steering_controller	API	ROS Index
diff_drive_controller	API	ROS Index
effort_controllers	API	ROS Index
force_torque_sensor_broadcaster	API	ROS Index
forward_command_controller	API	ROS Index
imu_sensor_broadcaster	API	ROS Index
joint_state_broadcaster	API	ROS Index
joint_trajectory_controller	API	ROS Index
pid_controller	API	ROS Index
position_controllers	API	ROS Index
range_sensor_broadcaster	API	ROS Index
steering_controllers_library	API	ROS Index
tricycle_controller	API	ROS Index
tricycle_steering_controller	API	ROS Index
velocity_controllers	API	ROS Index

9.2.3 control_msgs

Package Name	API	ROS Index
control_msgs	API	ROS Index

9.2.4 control_toolbox

Package Name	API	ROS Index
control_toolbox	API	ROS Index

9.2.5 kinematics_interface

Package Name	API	ROS Index
kinematics_interface	API	ROS Index
kinematics_interface_kdl	API	ROS Index

9.2.6 realtime_tools

Package Name	API	ROS Index
control_msgs	API	ROS Index

CHAPTER

TEN

SUPPORTED ROBOTS

This page hosts a list of supported robots and references to them. To add your robot, submit a PR to this page on Github!

10.1 Communication protocols

- Beckhoff ADS
- CanOpen
- ctrlX DataLayer non-realtime
- Ethercat
- Modbus (client)

10.2 End-effectors

- Schunk SVH 5-finger Hand
- TESOLLO Delto Gripper-M

10.3 Non robot-devices

- ctrlX AUTOMATION
- Feetech Servos
- Force Dimension haptic devices
- Hoverboard motors
- NDI measurement systems
- ODrive Motor Controller
- ODRI Motor Controller
- PCA9685 16-Channel 12-bit PWM/Servo Driver
- ROBOTIS Dynamixel

10.4 Official (supported by robot manufacturer)

- Clearpath Robotics Dingo Dx1X0
- Clearpath Robotics Husky A200
- Clearpath Robotics Husky A300
- Clearpath Robotics Jackal J100
- Clearpath Robotics Ridgeback R100
- Clearpath Robotics Warthog W200
- Denso Robots - 4-Axis (SCARA) robots, 5-and 6-Axis robots, and Collaborative robots (not OSS type)
- Fanuc
- Flexiv Robotics Rizon robots
- Franka Robotics - Franka Research 3 (FR3)
- Husarion ROSbot 2R / 2 PRO
- Husarion ROSbot XL
- Husarion ROSbot XL with OpenMANIPULATOR-X (including MoveIt2)
- igus/Commonplace Robotics
- Kinova® Kortex™ Gen3
- Mitsubishi MELFA
- PMB2 - Differential Drive Mobile Base
- ROBOTIS AI Worker
- ROBOTIS OpenMANIPULATOR
- TIAGo - Mobile Manipulator
- Universal Robots
- xArm

10.5 Unofficial (from the community)

- ABB - EGM interface
- KUKA All Robots (iiQKA-ECI, Sunrise-FRI, KSS-RSI)
- KUKA IIWA (KUKA Fast Robot Interface (FRI))
- KUKA industrial robots (KUKA Robot Sensor Interface (RSI))
- KUKA LBRs (IIWA 7/14 and Med 7/14 via KUKA Fast Robot Interface (FRI))
- Mitsubishi RV1A

CHAPTER

ELEVEN

RESOURCES

Make sure you also check the ROSCon presentations and videos for the ros2_control updates - those are not listed here.

The resources provided in the `resources` folder are available for use under CC-BY license. The original authors are named either in the documents or in the list down below.

Any files submitted to the documentation should be "licensed" by stating your name and `ros2_control` organization if no company name applicable, e.g., `CC-BY My Name (ros2_control/company_name)`.

11.1 Presentations

11.1.1 2025-10 ROSCon 2025

`Presentation: ros-controls project update`

Summary:

From last year's ROSCon there were many changes and updates to ros2_control. The number of maintainers has doubled, ros-controls is now an OSRA project and some of the long awaited features have been merged! From 2025, after many months of intensive development and a few bugs later, you can enjoy fully fledged async components, support for variants, access to URDF from every component, integrated joint limiters on the hardware layer that controllers can also use. Of course, many details more, like a dedicated repository with shared CMake definitions and pre-defined CI actions. Join the talk and save some time in the future!

Recording

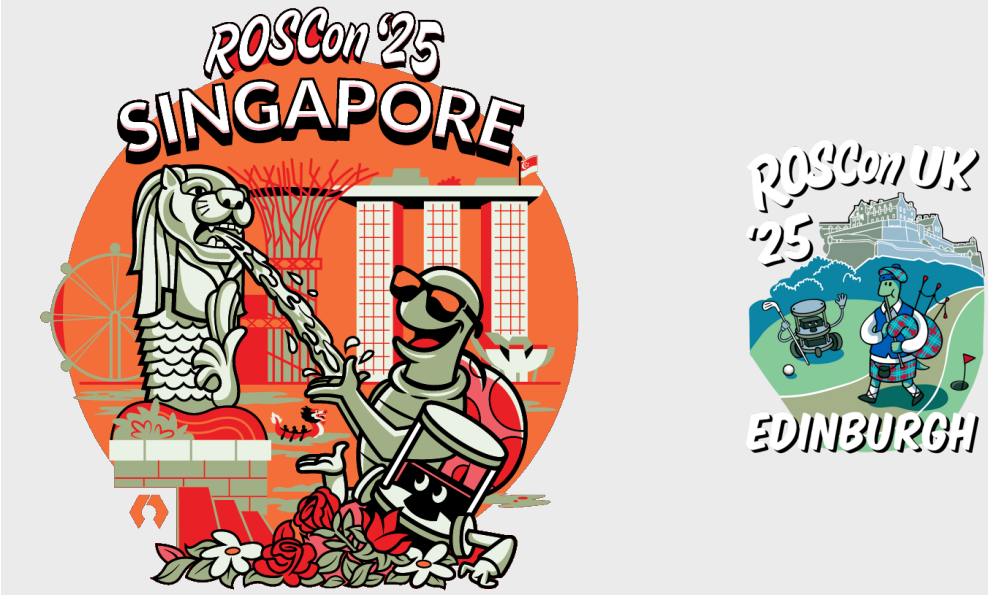
Presenter: Dr. Denis Stogl

Authors:

- Dr. Bence Magyar (Locus Robotics)
- Dr. Denis Stogl (b»robotized)

11.1.2 2025-09-26 ROSCon UK Workshop: Writing Custom Robot Drivers and Control Workshop at ROSCon 2025 Singapore

ROSCon 2025 Workshop



ros2_control: Writing Custom Robot Drivers

ros2_control is a hardware-agnostic control framework for abstracting hardware and low-level control for 3rd party solutions like `MoveIt2` and `Nav2` systems.

This workshop provides a practical deep dive into writing robot drivers with ros2_control. You will be introduced to hands-on integration of an embedded board that implements a differential drive robot.

Additionally, we'll demonstrate examples from different domains and best practices for using ros2_control for ease of use, increased flexibility and robustness.

Prerequisites - Before coming to the conference

1. Please bring a **USB-C cable you can plug to your laptop**! It should be power- and data-capable.
2. It is recommended to have a **Linux-based OS** installed on your laptop (Recommended: Ubuntu 22.04 or 24.04). No ROS setup is required locally, as everything will run in Docker containers.
3. We need attendees to have docker engine and the docker compose plugin installed. Installation instructions for various platforms can be found on the linked pages.

Pull as soon as you can to verify your setup and get the majority of the download but also try re-pulling closer to the date to make sure you have the latest updates!

```
wget https://tinyurl.com/roscontrol2025 -O docker-compose.yaml
docker compose pull
```

For optimal copy&paste experience, you can pull the github repository. Some things are not yet finalized but pulling early and often is a good idea.

```
git clone https://github.com/ros-controls/roscon2025_control_workshop
```

Slides

`Slides for: ros2_control: Fun with Robot Drivers`

People

This workshop was brought to you by

- Dr.-Ing. Denis Stogl, b>>robotized
- Dr. Bence Magyar, Locus Robotics
- Marq Rasmussen, Locus Robotics
- Sai Kishor Kothakota, PAL Robotics
- Christoph Fröhlich, Austrian Institute Of Technology

Workshop page

Authors:

- Christoph Fröhlich (Austrian Institute Of Technology)
- Sai Kishor Kothakota (PAL Robotics)
- Dr. Bence Magyar (Locus Robotics)
- Marq Rasmussen (Locus Robotics)
- Dr. Denis Stogl (b»robotized)

11.1.3 2024-10 ROSCon 2024

`Presentation: Something big is coming in ros2_control with ROS 2 Jazzy!`

Summary:

We know you love ros2_control, but… Have you ever wanted to control your robot with data that is not a C++ double value? Annoyed with maintaining data storage in robot drivers? This is solved in ROS 2 Jazzy! Now the framework manages storage for you and even allows you to pass strings to your robot! Plus…there is more! Do you have a fancy AI-based controller or something with inverse dynamics that runs slower than your hardware expects? No problem! If you are keen to learn more about these and other features, you have to be at this presentation!

Recording

Presenter: Dr. Bence Magyar

Authors:

- Dr. Bence Magyar (Locus Robotics)
- Dr. Denis Stogl (Stogl Robotics Consulting)

11.1.4 2024-10-21 ROSCon Workshop: Fun with Controllers

ROSCon 2024 Workshop

Location: Room 200

Time: 13:00-17:00, 21. October 2024

ros2_control: Fun with Controllers

Summary

If you already know that the ros2_control framework acts as a Kernel for ROS 2 robotics systems you are using but need help with application complexity, then this workshop is for you. The workshop covers the use of ros2_control controllers in products from various industries and shows solutions for all the little issues when running 24/7.

You will get a practical overview of concepts like controller chaining - used for cascade control and real-time state estimators; and asynchronous and "side-load" controllers that enable you to run complex calculations without jitter in your control loops. We expect your active involvement!

Before coming to the conference

Recommended system setup:

- Ubuntu 22.04 or Ubuntu 24.04
- docker engine & docker compose installed

If you don't have docker, follow the docker instructions to install it.

If you don't have docker compose, run `sudo apt-get install docker-compose-plugin` or follow the docker compose instructions to install it.

Slides are available in pdf and pptx .

Try pulling the container we use prior to coming to the workshop:

```
docker pull bmagyar/roscon2024_workshop
```

People

This workshop was brought to you by

- Dr. Denis Stogl, Stogl Robotics Consulting
- Dr. Bence Magyar, Locus Robotics

Summary:

If you already know that the ros2_control framework acts as a Kernel for ROS 2 robotics systems you are using but need help with application complexity, then this workshop is for you. The workshop covers the use of ros2_control controllers in products from various industries and shows solutions for all the little issues when running 24/7.

You will get a practical overview of concepts like controller chaining - used for cascade control and real-time state estimators; and asynchronous and "side-load" controllers that enable you to run complex calculations without jitter in your control loops. We expect your active involvement!

Workshop page

Authors:

- Dr. Denis Stogl (Stogl Robotics Consulting)
- Dr. Bence Magyar (Locus Robotics)

11.1.5 2023-10-18 ROSCon Workshop: ros2_control on Steroids

ROSCon 2023 Workshop

Location: Imperial 9

Time: 13:00-17:00, 18. October 2023

ros2_control on Steroids

Summary

If you already know that the ros2_control framework acts as a Kernel for ROS 2 robotics systems, you are using it but struggling with application complexity, then this workshop is for you. The workshop covers the use of ros2_control in products from various industries and shows solutions for all the little issues when running 24/7.

You will get a practical overview of concepts like controller chaining, hardware modularization, multi-robot architectures, parameter injection, and debugging of complex systems. On top of showcasing these functionalities, we expect your involvement in the discussion by bringing your complex application and discussing existing and potentially missing tooling in ros2_control.

Slides

`Presentation: ros2_control - ros2_control on Steroids`

Before coming to the conference

Recommended system setup:

- Ubuntu 20.04 or Ubuntu 22.04
- docker engine & docker compose installed

If you don't have docker, follow the docker instructions to install it.

If you don't have docker compose, run `sudo apt-get install docker-compose-plugin` or follow the docker compose instructions to install it.

Once done, grab the latest version of the workshop container by running:

`docker pull bmagyar/roscon2023_workshop:latest`

Now, to set up a workspace, run the following commands where you want this to be placed:

```
mkdir -p ws/src
cd ws/src
git clone https://github.com/ros-controls/roscon2023_control_workshop
vcs import --input roscon2023_control_workshop/roscon2023_control_workshop.
↪ci.repos .
```

You can run things locally if you have all dependencies set up. The alternative is using the container which includes all dependencies & comes ready to compile the workspace. Using the same terminal as before (or a new one parked at `ws/src`) run:

```
docker compose -f roscon2023_control_workshop/docker/docker-compose.yaml run␣
↪dev
tmux
source /opt/ros/rolling/setup.bash
colcon build --symlink-install
source install/setup.bash
```

Open 2 more terminals in `tmux` by using `CTRL+B` and `"` and `CTRL+B` and `%`.

You can navigate in tmux using `CTRL+B` and `ARROW` keys.

People

This workshop was brought to you by

- Denis Stogl
- Bence Magyar

Summary:

If you already know that the ros2_control framework acts as a Kernel for ROS 2 robotics systems, you are using it but struggling with application complexity, then this workshop is for you. The workshop covers the use of ros2_control in products from various industries and shows solutions for all the little issues when running 24/7.

You will get a practical overview of concepts like controller chaining, hardware modularization, multi-robot architectures and debugging of complex systems. On top of showcasing these functionalities, we expect your involvement in the discussion by bringing your complex application and discussing existing and potentially missing tooling in ros2_control.

Workshop page

Authors:

- Dr. Bence Magyar (Locus Robotics)
- Dr. Denis Stogl (Stogl Robotics Consulting)

11.1.6 2023-09-19 ROSCon Spain Talk: Introduction to ros2_control

`Presentation: Introduction to ros2_control`

Summary:

This presentation aims to introduce the audience to the ros2_control framework, a hardware-agnostic control framework focusing on the modular composition of control systems for robots, sharing of controllers as well as real-time performance. The framework provides controller-lifecycle and hardware management on top of abstractions of real or virtual hardware interfaces. The talk explains different modules within the ros2_control ecosystem, such as hardware interfaces, controllers, controller managers, and how they interact with each other.

Recording

Presenter: Sai Kishor Kothakota

Authors:

- Sai Kishor Kothakota (PAL Robotics)

11.1.7 2023-07-07 ROS Developers Day 2023: Configure a Mobile Manipulator with ros2_control

Recording

Github repo with code

TBD add Construct rosject link

Summary:

In this hands-on presentation, we demonstrate how to set up a mobile manipulator with ros2_control in steps. First, we take a robot mobile base and demonstrate setting up ros2_control simulation for it in the URDF. Once done, a robot arm will be added to the mobile base, turning it into a mobile manipulator robot. The existing ros2_control configuration will be adjusted to accommodate for the new robot parts. Finally, the rosject concludes with a demonstration in Gazebo moving the robot using some off-the-shelf ros2_control controllers.

Attendees will learn

- how to prototype a mobile manipulator with URDF and ros2_control
- how to configure a gazebo simulation with a given URDF and ros2_control

Presenter: Bence Magyar

Authors:

- Dr. Bence Magyar (FiveAI / Bosch)

11.1.8 2023-02 ROS Meetup Munich #5

`Presentation: Tricycle Controller with ros2_control`

Meetup event link

Summary:
In this presentation Pixel Robotics presents the contributed the Tricycle controller to ros2_controllers, prefaced by an introduction to ros2_control.

Presenters: Johannes Plapp & Tony Najjar

Authors:

- Johannes Plapp (Pixel Robotics)
- Tony Najjar (Pixel Robotics)

11.1.9 2022-12 ROS-Industrial Conference 2022

`Presentation: ros2_control - Kernel for ROS 2 controlled robots`

Summary:
ros2_control is a hardware-agnostic control framework focusing on the modular composition of control systems for robots, sharing of controllers as well as real-time performance. The framework provides "kernel" functionality for robots by abstracting the hardware and doing heavy low-level management, for example, hardware lifecycle, communication and access control.

Presenter: Dr. Denis Stogl

Authors:

- Dr. Denis Stogl (Stogl Robotics Consulting)

11.1.10 2022-10 ROSCon 2022

`Presentation: A practitioner's guide to ros2_control`

Summary:
ros2_control is a hardware-agnostic control framework focusing on the modular composition of control systems for robots, sharing of controllers as well as real-time performance. The framework provides controller-lifecycle and hardware management on top of abstractions of real or virtual hardware interfaces.

This talk delves deeper into ros2_control, showcasing new features and what they could be used for, such as explicit lifecycle management, chaining controllers, emergency-stop handlers and mock components. Finally, we showcase different usages of ros2_control on openly accessible examples.

Recording

Presenter: Dr. Bence Magyar

Authors:

- Dr. Bence Magyar (FiveAI Ltd)
- Dr. Denis Stogl (Stogl Robotics Consulting)

11.1.11 2022-06 ROSCon Fr 2022

`Presentation: What is new in the best (and only) control framework for ROS2 - ros2_control`

Summary:

ros2_control is a hardware-agnostic control framework with a focus on both real-time performance and sharing of controllers. The framework has become one of the main utilities for abstracting hardware and low-level control for 3rd party solutions like *MoveIt2* and *Nav2* systems.

The presentation provides practical tips to use ros2_control, from creating a robot description, writing hardware drivers to configuring standard controllers. Some hot-new features, like controller chaining, will be shown. Furthermore, you will get introduced to concepts like modular reuse of hardware drivers, multi-robot architectures and parameters injection for controllers.

Recording

Presenter: Dr. Denis Stogl

Authors:

- Dr. Denis Stogl (Stogl Robotics Consulting)

11.1.12 2021-10 ROS World 2021

`Presentation: ros2_control - The future of ros_control`

Summary:

ros2_control is a robot-agnostic control framework with a focus on both real-time performance and sharing of controllers. The framework offers controller lifecycle and hardware resource management, as well as abstractions on hardware interfaces.

Controllers expose ROS interfaces for 3rd party solutions to robotics problems like manipulation path planning (*moveit2*) and autonomous navigation (*nav2*). The modular design makes it ideal for both research and industrial use. A robot made up of a mobile base and an arm that supports ros2_control needs no extra code, only a few controller configuration files and it is ready to go.

Recording

Presenter: Dr. Bence Magyar

Authors:

- Dr. Bence Magyar (FiveAI Ltd)
- Denis Stogl (Stogl Robotics Consulting)

`Presentation: Making a robot ROS 2 powered - a case study using the UR manipulators`

Summary:

With the release of ros2_control and MoveIt 2, ROS 2 Foxy finally has all the "ingredients" needed to power a robot with similar features as in ROS 1. We present the driver for Universal Robot's manipulators as a real-world example of how robots can be run using ROS 2. We show how to realize multi-interface support for position and velocity commands in the driver and how to support scaling controllers while respecting factors set on the teach pendant. Finally, we show how this real-world example influences development of ros2_control to support non-joint related inputs and outputs in its real-time control loop.

Recording

Presenter: Denis Štogl

Authors:

- Denis Štogl (PickNik Inc.)
- Dr. Nathan Brooks (PickNik Inc.)
- Lovro Ivanov (PickNik Inc.)
- Dr. Andy Zelenak (PickNik Inc.)
- Rune Søe-Knudsen (Universal Robots)

`Presentation: Online Trajectory Generation and Admittance Control in ROS2`

Summary:

One of the top reasons to upgrade from ROS1 to ROS2 is better suitability for realtime tasks. We discuss the development of a new ROS2 controller to handle realtime contact tasks such as tool insertion with industrial robots. The admittance controller handles trajectories and single-waypoint streaming commands, making it compatible with MoveIt and many teleoperation frameworks. Part of the work involved ensuring kinematic limits (position/velocity/acceleration/jerk) are obeyed while limiting interaction forces with the environment. Finally, we give practical recommendations and examples of the admittance controller. A live demo will be shown at our booth.

Recording

Presenter: Dr. Andy Zelenak

Authors:

- Dr. Andy Zeleank (PickNik Inc.)
- Denis Štogl (PickNik Inc.)

11.1.13 2021-10-07 Weekly Robotics Meetup #13

`Meetup presentation: Getting started with ros2_control`

Summary:

ros2_control is a robot-agnostic control framework with a focus on both real-time performance and sharing of controllers. The framework offers controller lifecycle and hardware-resource management, as well as abstractions on hardware interfaces.

Controllers expose ROS interfaces for 3rd party solutions to robotics problems like manipulation path planning (*moveit2*) and autonomous navigation (the ROS2 navigation stack). Hardware components on the other side provide a unified interface for robotic hardware, enabling standardized life-cycle and access management. The modular design makes ros2_control ideal for both research and industrial use. For example, a robot made up of a mobile base and an arm that supports ros2_control needs no extra code, only a few controller configuration files, and it is ready to go.

In this talk, we will discuss concepts of ros2_control framework compared to ros(1)_control framework and show examples of their use in the wild.

Recording

Presenters: Dr. Bence Magyar and Denis Štogl

Authors:

- Dr. Bence Magyar (FiveAI Ltd)
- Denis Stogl (Stogl Robotics Consulting)

11.1.14 2021-06 ROSDevDay 2021

Presentation materials

Recording

Presenters: Dr. Bence Magyar and Denis Štogl

Authors:

- Dr. Bence Magyar (FiveAI Ltd)
- Denis Stogl (Stogl Robotics Consulting)

11.1.15 2021-05 ROSCon Fr 2021

`Presentation: Getting started with ros2_control`

Summary:
The presentation gives a quick overview on the basic concepts and some simple implementation examples. We show implementing a simple Hardware Abstraction Layer (aka SystemComponent) and a forwarding controller. Once done, we also look into modifying the controller with the example goal of changing the type of the command topic.

Recording

Presenter: Dr. Bence Magyar

11.2 Diagrams

Folder with diagrams and sources for the images. Simply use diagrams.net for editing.

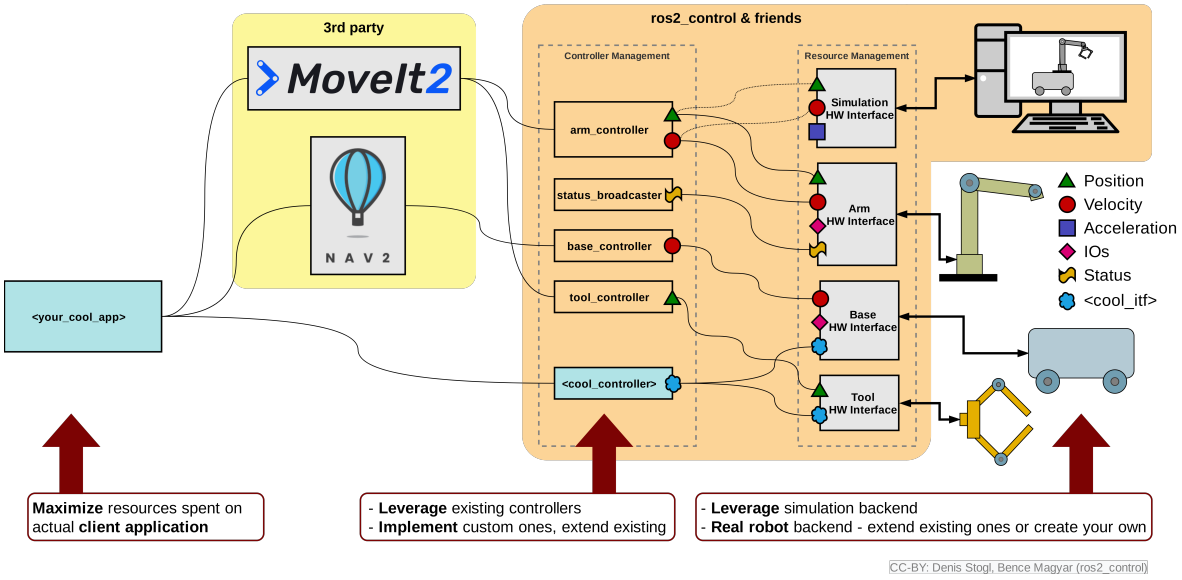
ros2_control - a collection of `ros2_control`-related diagrams.

- overview diagrams
- integration with MoveIt2
- class diagrams
- lifecycle diagrams
- command and state interfaces examples
- mobile manipulator architectures
- Force-Control architectures

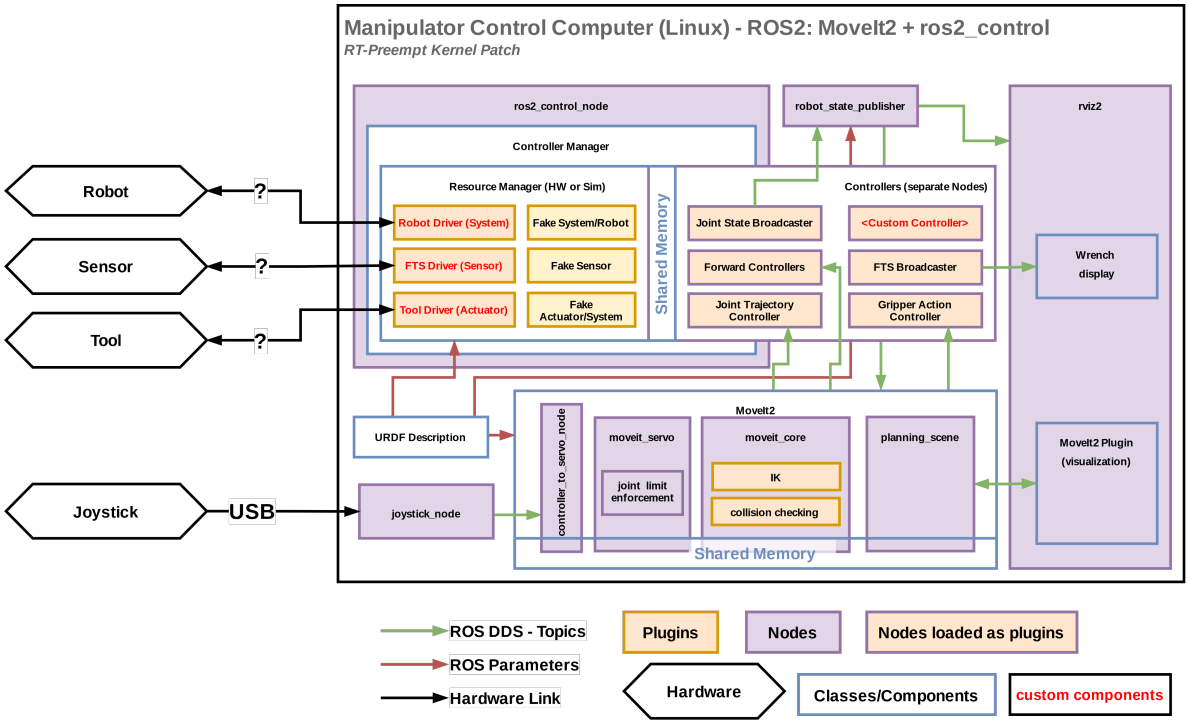
11.3 Images

Generated images for the presentation which can be useful also for the documentation.

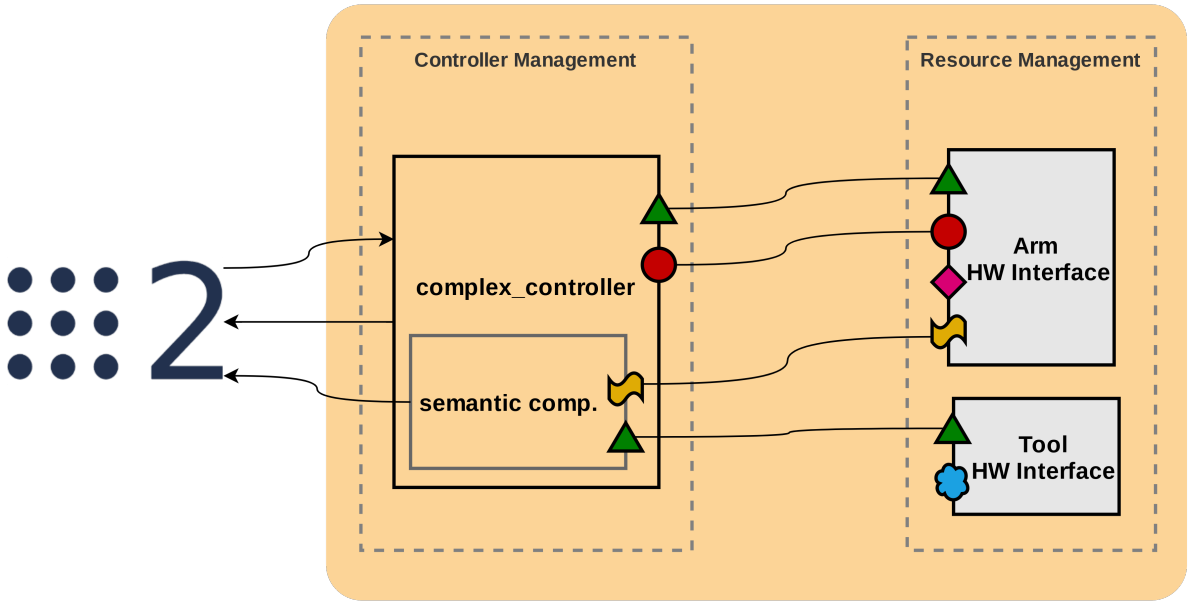
Overview of ros2_control



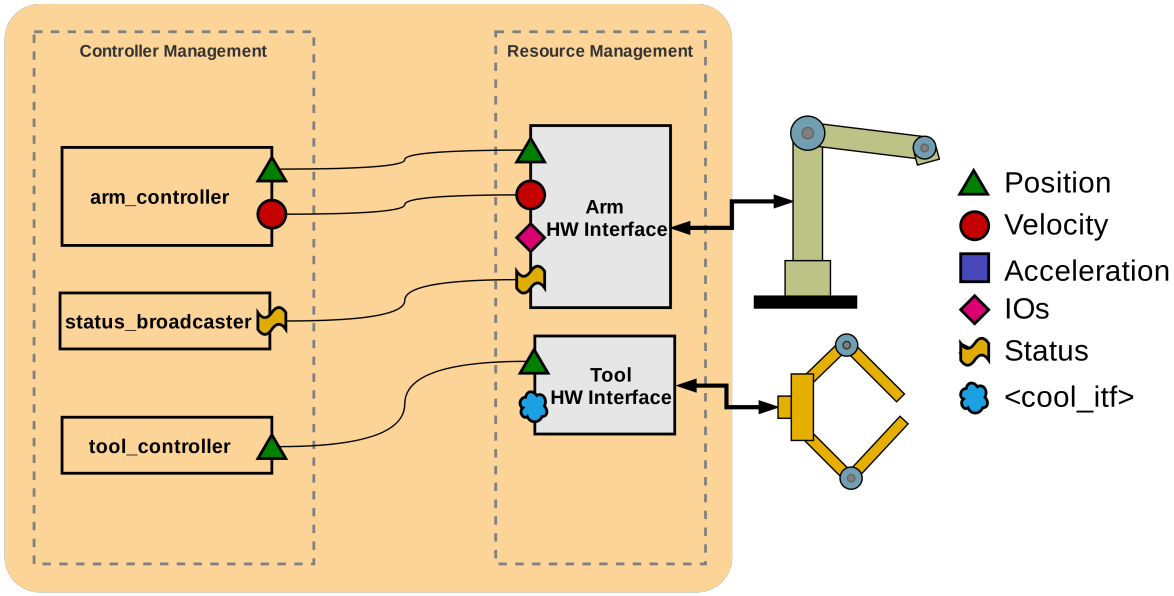
ros2_control robot integration with MoveIt2



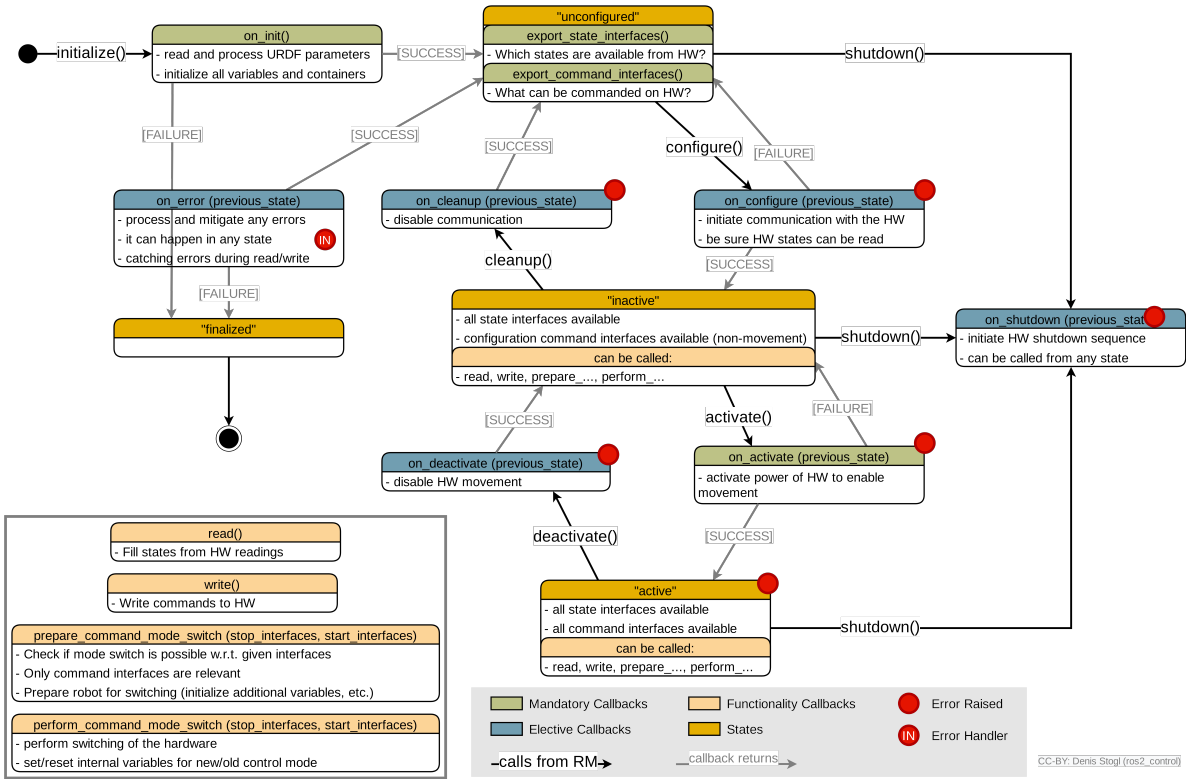
Architecture of complex controller and semantic components:



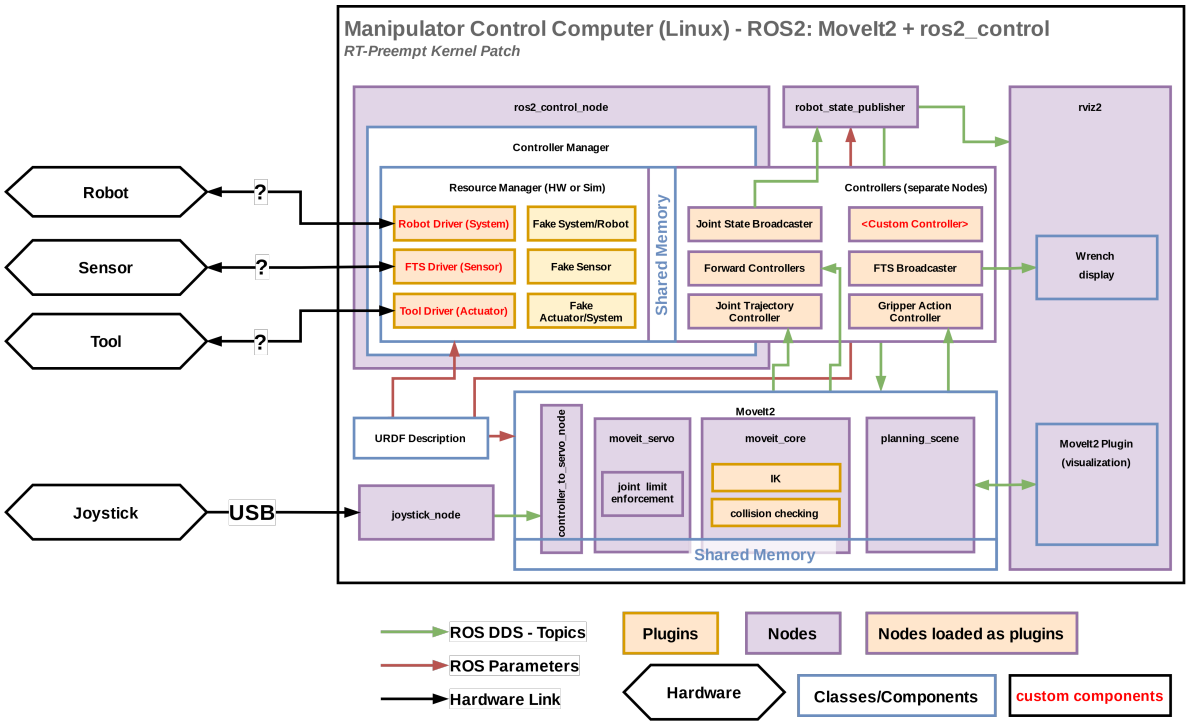
Architecture of command and state interfaces:



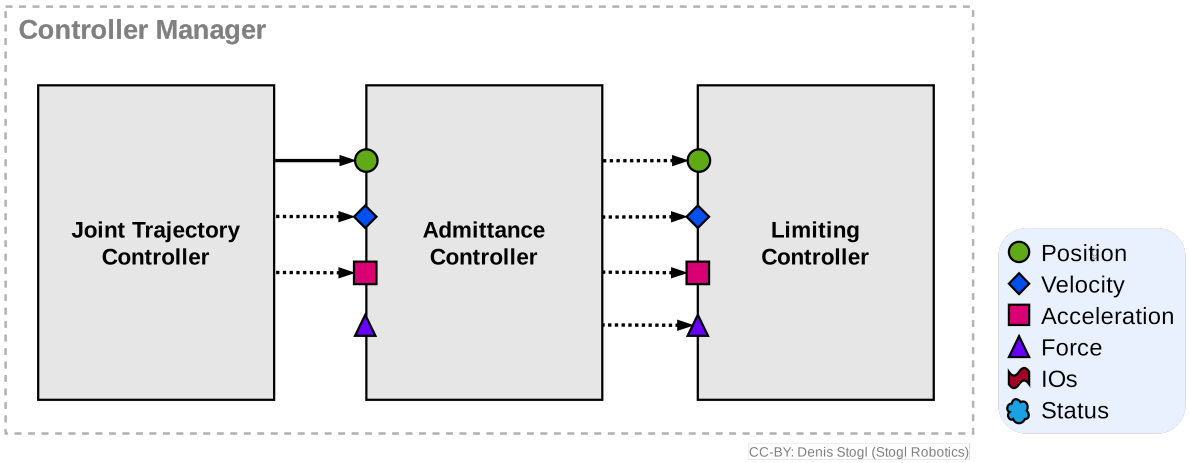
Lifecycle of hardware interfaces:



ros2_control integration with MoveIt2



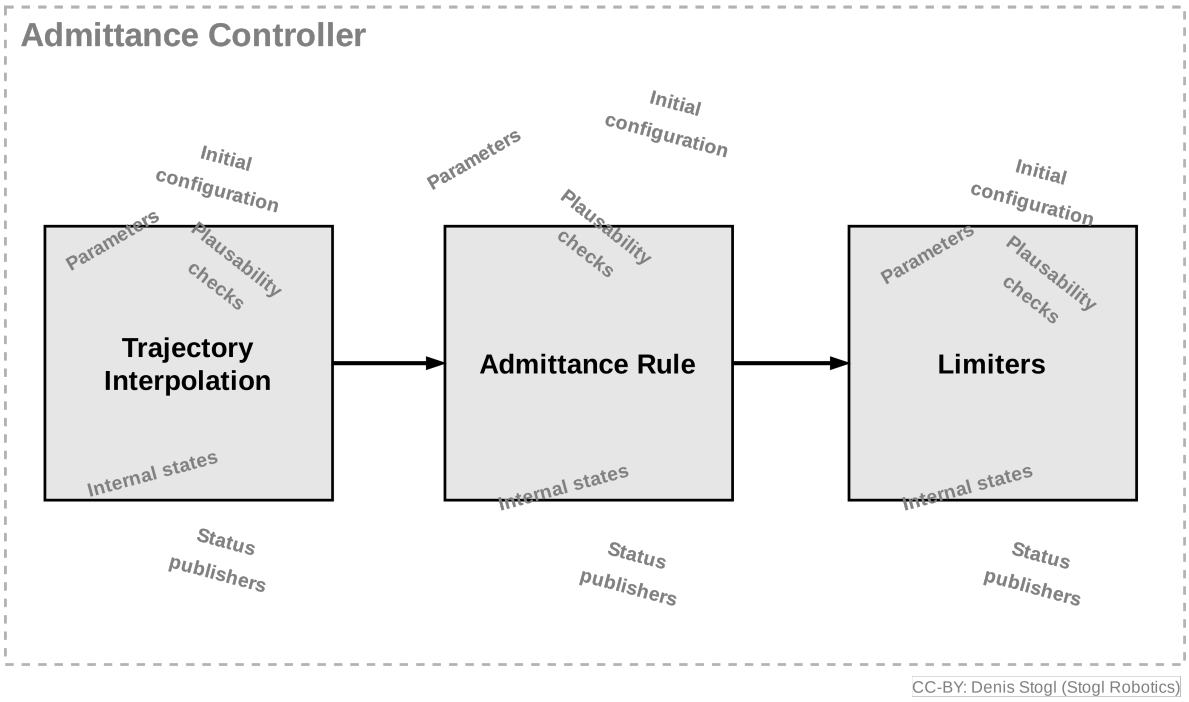
Controllers architecture with chained controllers - admittance controller example



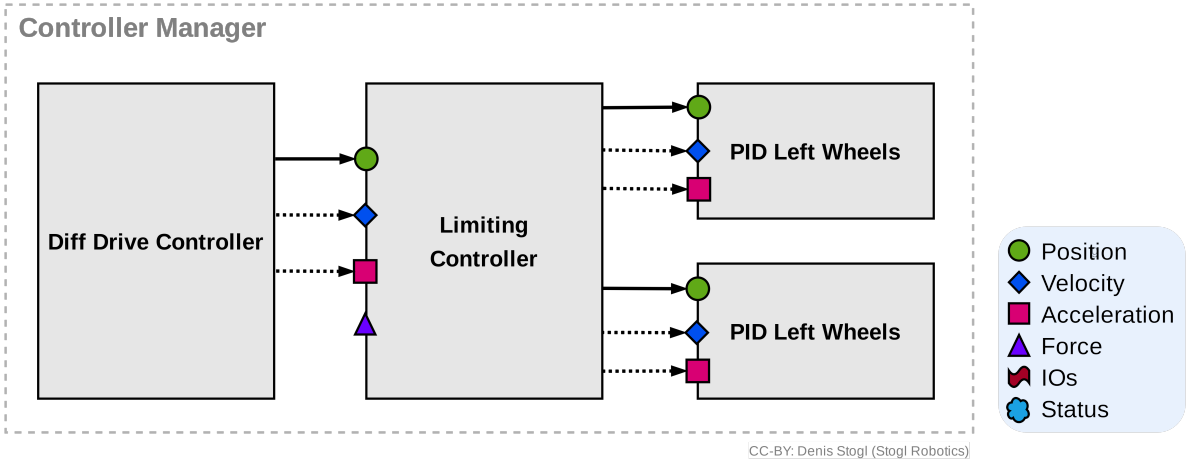
Controllers architecture with chained controllers - admittance controller example (URDF)

```yaml
controller_manager:
  update_rate: 500  # Hz

  joint_trajectory_controller:
    type: joint_trajectory_controller/JointTrajectoryController

  admittance_controller:
    type: admittance_controller/AdmittanceController

  limiting_controller:
    type: limiting_controllers/JointLimitingController


joint_trajectory_controller:
  joints:
    - joint1
    - ...
  command_joints:
    - admittance_controller/joint1
    - ...
  command_interfaces:
    - position
  state_interfaces:
    - position
    - velocity

# export reference interfaces: "<controller_name>/<joint_name>/<interface_name>"
admittance_controller:
  joints:
    - joint1
    - ...
  command_joints:
    - limiting_controller/joint1
    - ...
  command_interfaces:
    - position
  state_interfaces:
    - position
    - velocity


# export reference interfaces: "<controller_name>/<joint_name>/<interface_name>"
limiting_controller:
  joints:
    - joint1
    - ...
  interfaces:
    - position
limiting_controller:
  joints:
    - joint1
    - ...
  interfaces:
    - position
```

Controllers architecture without chained controllers - admittance controller example



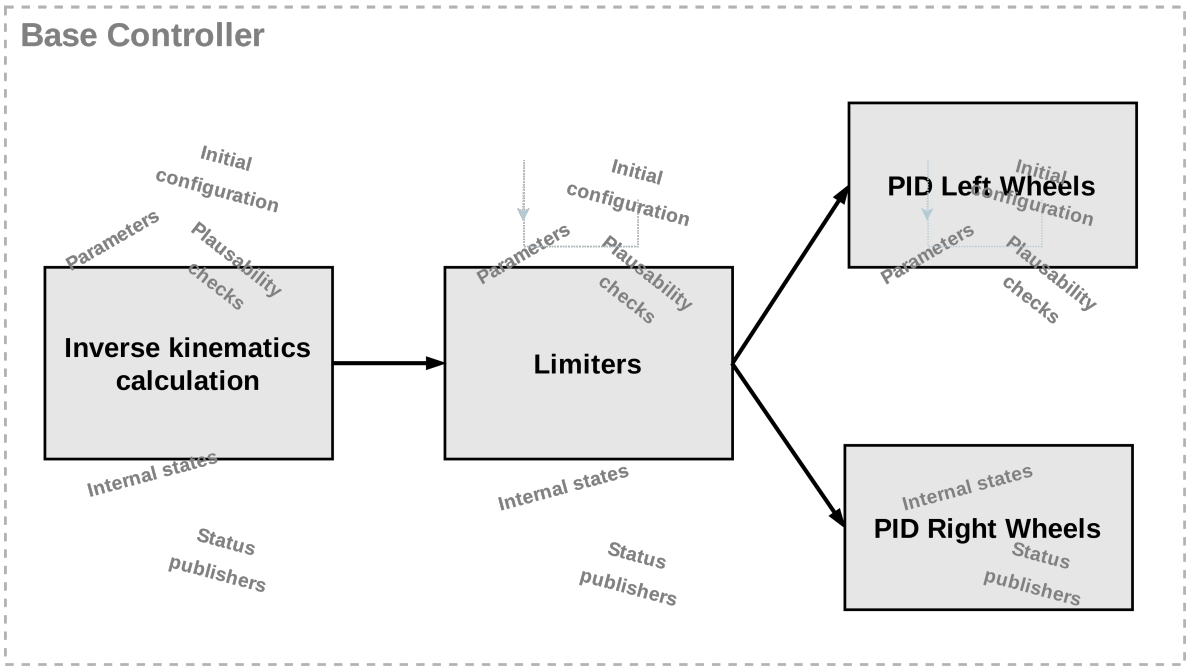
Controllers architecture with chained controllers - mobile base controller example



Controllers architecture with chained controllers - mobile base controller example (URDF)

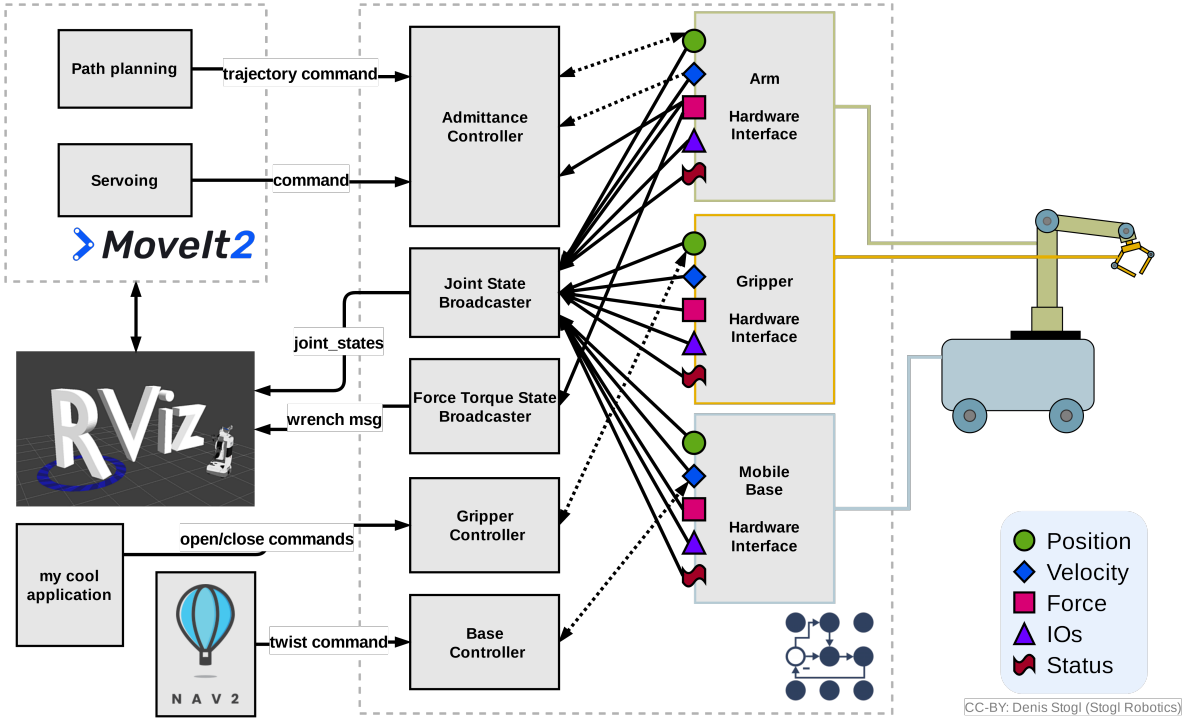
```yaml
controller_manager:
  update_rate: 500  # Hz

  diff_drive_controller:
    type: diff_drive_controller/DiffDriveController

  limiting_controller:
    type: limiting_controllers/JointLimitingController

  pid_left_wheels:
    type: pid_controllers/PIDController

  pid_right_wheels:
    type: pid_controllers/PIDController


diff_drive_controller:
  left_weel_names:
    - left_wheel_1
    ...

# export reference interfaces: "<controller_name>/<joint_name>/<interface_name>"
limiting_controller:
  joints:
    - left_wheel_1
    - ...
  command_joints:
    - pid_left_wheels/joint1/velocity
    - ...
    - pid_right_wheels/joint1/velocity
    - ...
  interfaces:
    - velocity


# export reference interfaces: "<controller_name>/<joint_name>/<interface_name>"
pid_left_wheels:
  joints:
    - left_wheel_1
    ...


# export reference interfaces: "<controller_name>/<joint_name>/<interface_name>"
pid_right_wheels:
  joints:
    - right_wheel_1
    ...
```

Controllers architecture without chained controllers - admittance controller example



CC-BY: Denis Stogl (Stogl Robotics)

Controllers architecture - overview

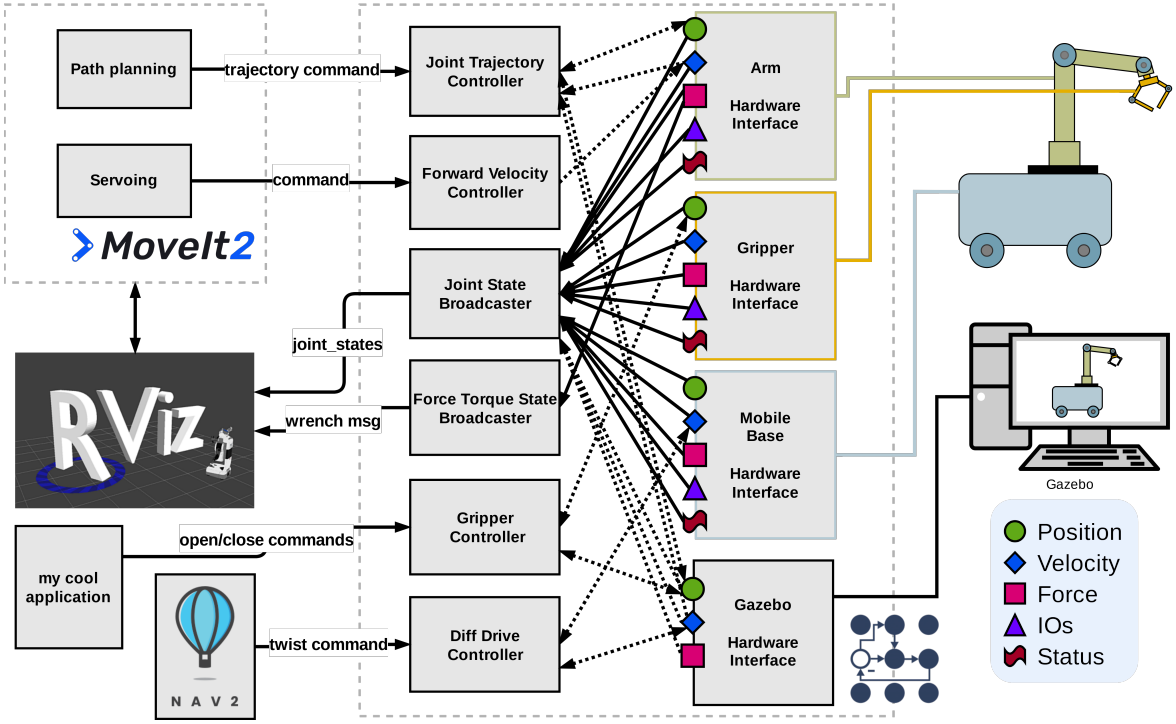


CC-BY: Denis Stogl (Stogl Robotics)

Controllers architecture - URDF

```yaml
controller_manager:
  update_rate: 500  # Hz

  admittance_controller:
    type: ros2_control_mm/AdmittanceController

  forward_position_controller:
    type: position_controllers/JointGroupPositionController

  joint_state_broadcaster:
    type: joint_state_broadcaster/JointStateBroadcaster

  force_torque_sensor_broadcaster:
    type: force_torque_sensor_broadcaster/ForceTorqueStateBroadcaster

  gripper_controller:
    type: position_controllers/GripperActionController

  base_controller:
    type: ros2_control_mm/BaseControllers


admittance_controller:
  joints:
    - joint1
    - ...
  command_interfaces:
    - position
  state_interfaces:
    - position
    - velocity


forward_position_controller:
  joints:
    - joint1
    - ...


force_torque_sensor_broadcaster:
  sensor_name: tcp_fts_sensor
  frame_id: tool0
  topic_name: ft_data


gripper_controller:
  joints:
    - gripper_joint
  command_interface: position


base_controller:
  left_weel_names:
    - left_wheel_1
    ...
```

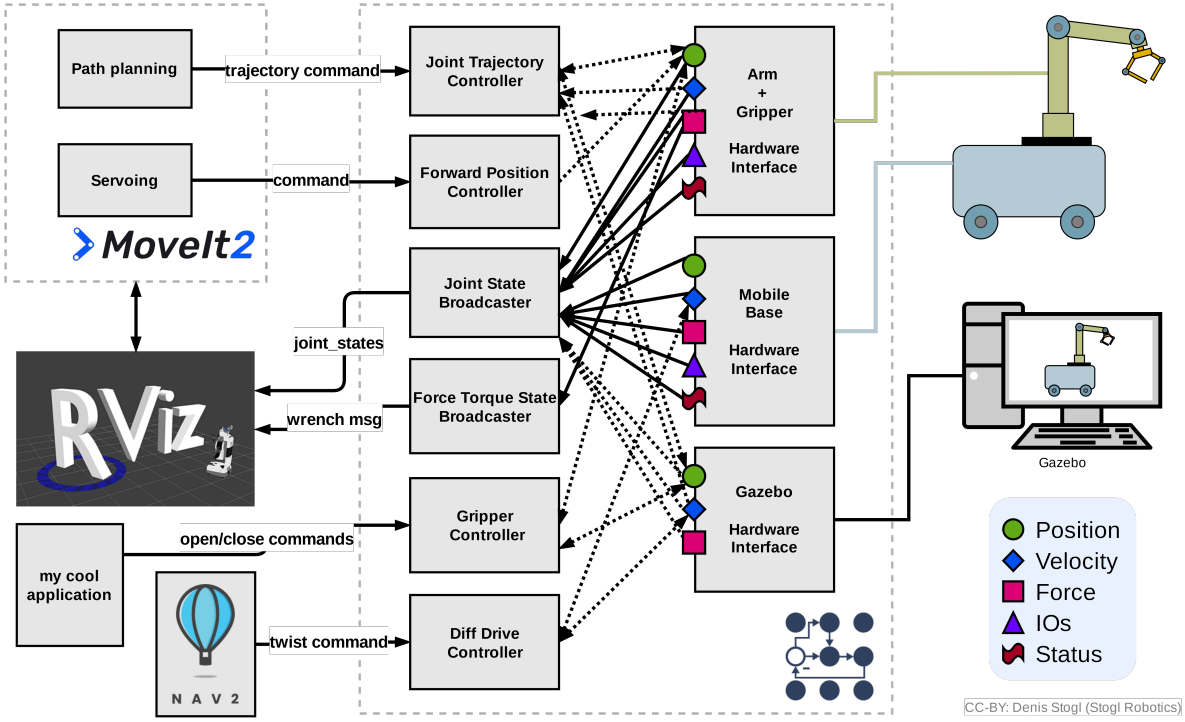
Hardware architecture - independent communication to the hardware (modular hardware)



Hardware architecture - independent communication to the hardware (modular hardware) (URDF)

```
<ros2_control name="MobileBase" type="system">
  <hardware>
    <plugin>ros2_contro_robot/Mobile_Base</plugin>
    ...
  </hardware>
  <joint name="wheel1_joint">
    <command_interface name="velocity"/>
    <state_interface name="position"/>
    <state_interface name="velocity"/>
    <state_interface name="status"/>
  </joint>
  ...
</ros2_control>

<ros2_control name="Arm" type="system">
  <hardware>
    <plugin>ros2_contro_robot/Arm</plugin>
    ...
  </hardware>
  <joint name="joint1">
    <command_interface name="position"/>
    <state_interface name="position"/>
    <state_interface name="velocity"/>
    <state_interface name="status"/>
  </joint>
  ...
</ros2_control>

<ros2_control name="Gripper" type="actuator">
  <hardware>
    <plugin>ros2_contro_robot/Gripper</plugin>
    ...
  </hardware>
  <joint name="gripper_joint">
    <command_interface name="position"/>
    <state_interface name="position"/>
    <state_interface name="velocity"/>
    <state_interface name="status"/>
  </joint>
  ...
</ros2_control>
```

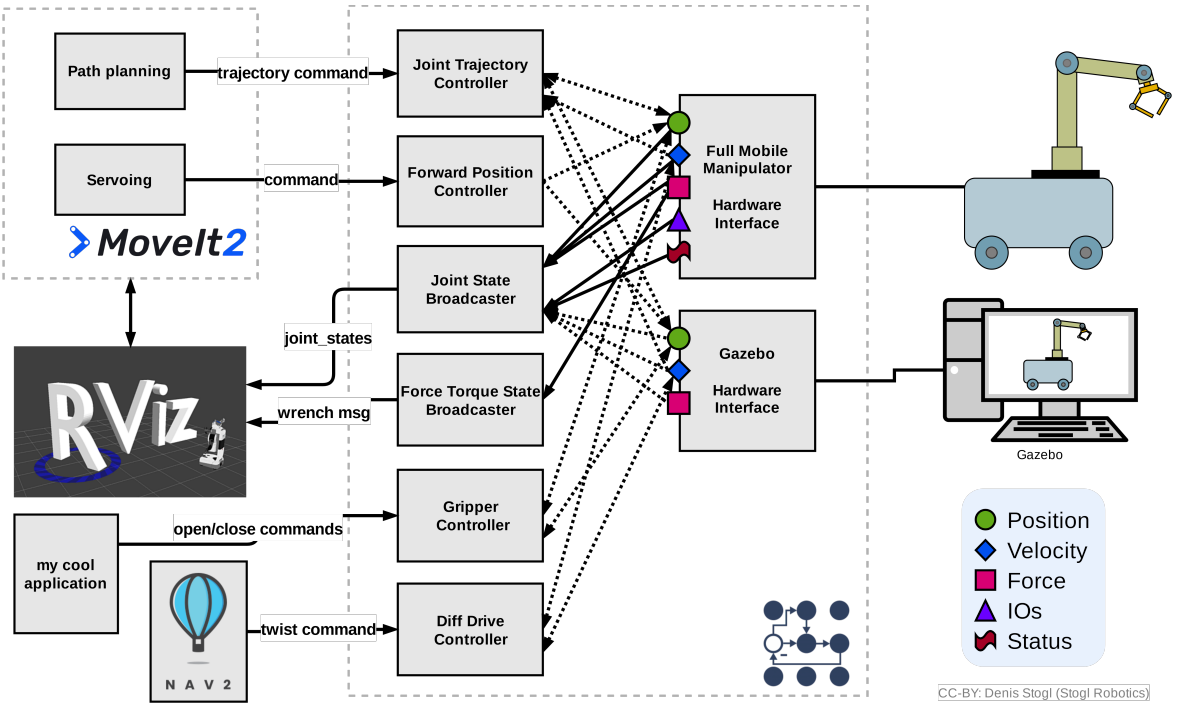
Hardware architecture - gripper communication through Arm



Hardware architecture - gripper communication through Arm (URDF)

```
<ros2_control name="MobileBase" type="system">
  <hardware>
    <plugin>ros2_contro_robot/Mobile_Base</plugin>
    ...
  </hardware>
  <joint name="wheel1_joint">
    <command_interface name="velocity"/>
    <state_interface name="position"/>
    <state_interface name="velocity"/>
    <state_interface name="status"/>
  </joint>
  ...
</ros2_control>

<ros2_control name="ArmWithGripper" type="system">
  <hardware>
    <plugin>ros2_contro_robot/Arm_With_Gripper</plugin>
    ...
  </hardware>
  <joint name="joint1">
    <command_interface name="position"/>
    <state_interface name="position"/>
    <state_interface name="velocity"/>
    <state_interface name="status"/>
  </joint>
  ...
  <joint name="gripper_joint">
    <command_interface name="position"/>
    <state_interface name="position"/>
    <state_interface name="velocity"/>
    <state_interface name="status"/>
  </joint>
  ...
</ros2_control>
```

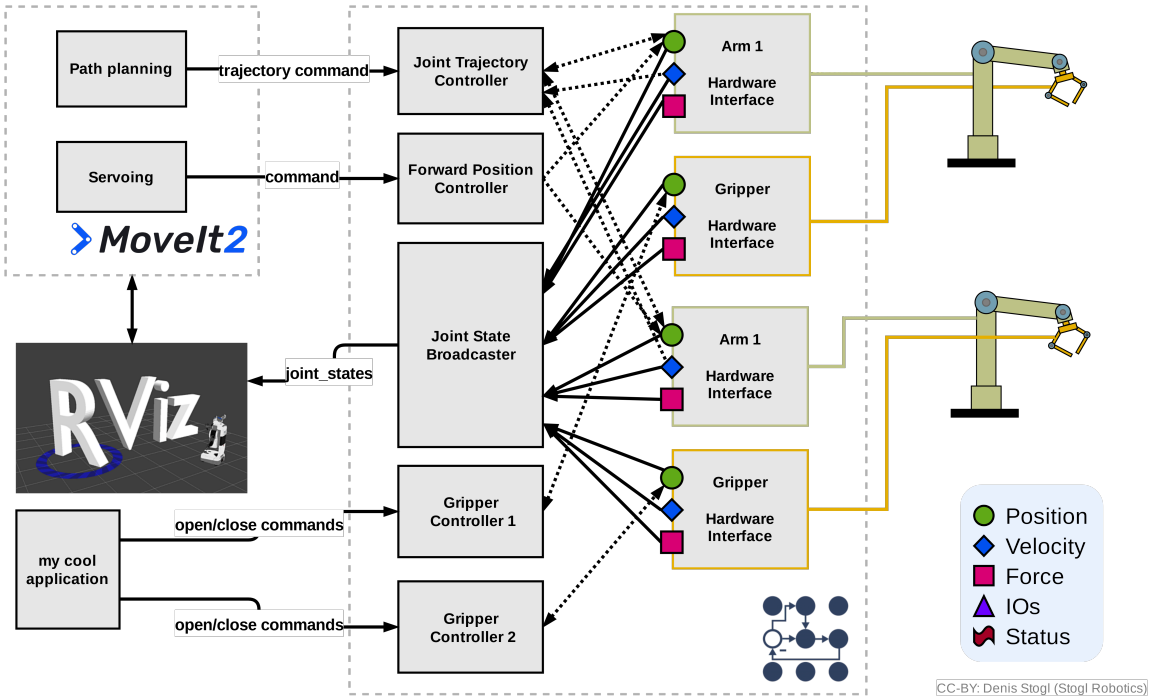
Hardware architecture - monolitic communication to hardware



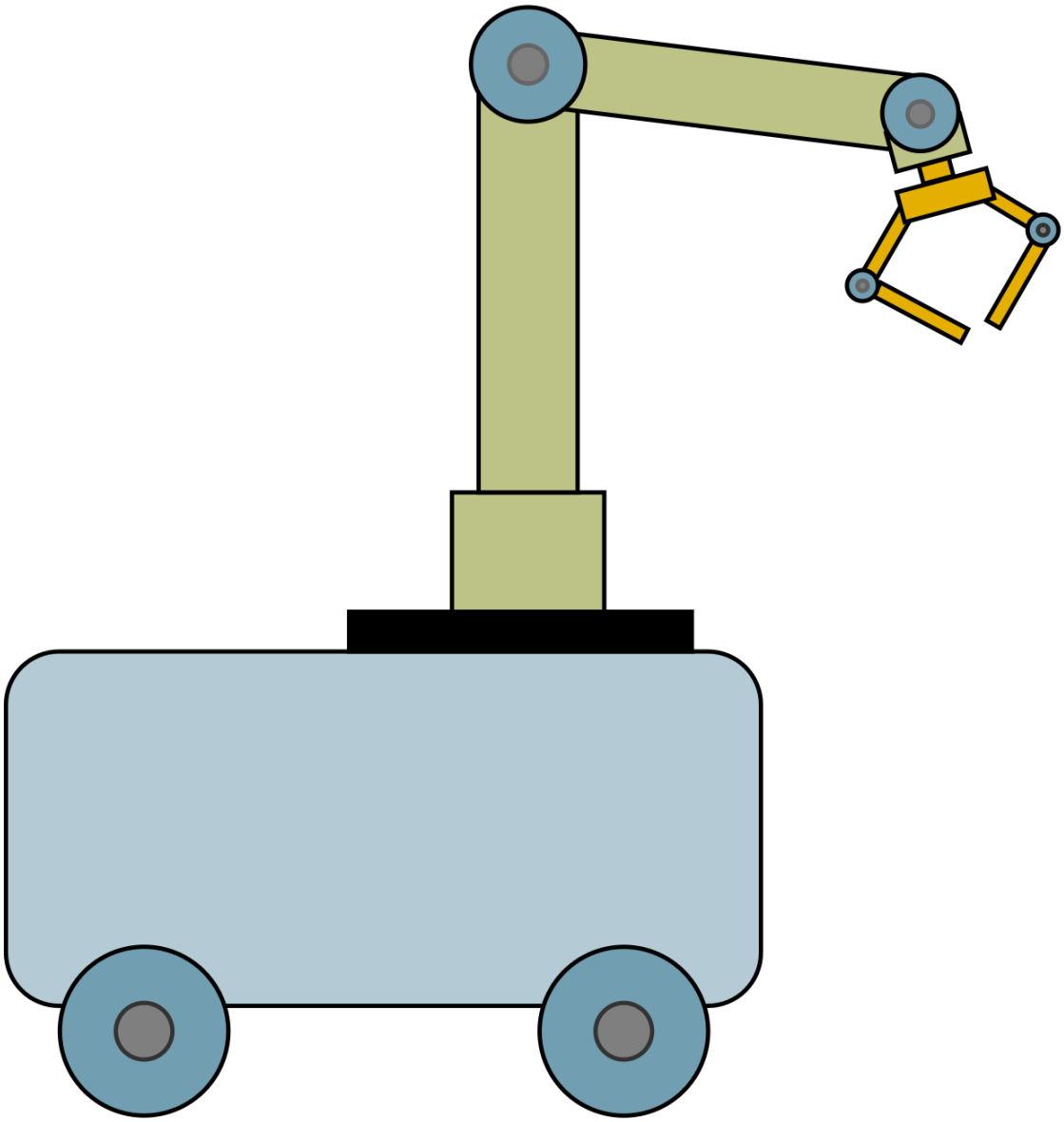
Hardware architecture - monolitic communication to hardware (URDF)

```
<ros2_control name="MobileManipulator" type="system">
  <hardware>
    <plugin>ros2_contro_robot/Full_Mobile_Manipulator</plugin>
    ...
  </hardware>
  <joint name="wheel1_joint">
    <command_interface name="velocity"/>
    <state_interface name="position"/>
    <state_interface name="velocity"/>
    <state_interface name="status"/>
  </joint>
  ...
  <joint name="joint1">
    <command_interface name="position"/>
    <state_interface name="position"/>
    <state_interface name="velocity"/>
    <state_interface name="status"/>
  </joint>
  ...
  <joint name="gripper_joint">
    <command_interface name="position"/>
    <state_interface name="position"/>
    <state_interface name="velocity"/>
    <state_interface name="status"/>
  </joint>
  ...
</ros2_control>
```

Hardware architecture - multiple hardware in one controller manager



Example files - ros2_control - "Controlko" mobile manipulator



Example files - ros2_control - “Controlko” mobile manipulator (URDF)

```yaml
controller_manager:
  update_rate: 500  # Hz

  joint_trajectory_controller:
    type: joint_trajectory_controller/JointTrajectoryController

  forward_position_controller:
    type: position_controllers/JointGroupPositionController

  joint_state_broadcaster:
    type: joint_state_broadcaster/JointStateBroadcaster

  force_torque_sensor_broadcaster:
    type: force_torque_sensor_broadcaster/ForceTorqueStateBroadcaster

  gripper_controller:
    type: position_controllers/GripperActionController

  diff_drive_controller:
    type: diff_drive_controller/DiffDriveController


joint_trajectory_controller:
  joints:
    - joint1
    - ...
  command_interfaces:
    - position
  state_interfaces:
    - position
    - velocity


forward_position_controller:
  joints:
    - joint1
    - ...


force_torque_sensor_broadcaster:
  sensor_name: tcp_fts_sensor
  frame_id: tool0
  topic_name: ft_data


gripper_controller:
  joints:
    - gripper_joint
  command_interface: position


diff_drive_controller:
  left_weel_names:
    - left_wheel_1
    ...
```

CHAPTER

TWELVE

CONTRIBUTING

12.1 Contributing Guidelines

First, thank you for considering contributing to the ros2_control project. As an open-source project, we welcome each contributor, regardless of their background and experience. To reduce the entropy of the universe and our vivid, open, and collaborative environment, we have set up some standards and methods for contributions.

12.1.1 Reporting Bugs/Feature Requests

We welcome you to use the GitHub issue tracker to report bugs or suggest features.

When filing an issue, please check existing open issues or recently closed issues to make sure somebody else hasn't already reported the issue. Please try to include as much information as you can. Details like these are incredibly useful:

- A reproducible test case or series of steps
- The version of our code being used
- Any modifications you've made relevant to the bug
- Anything unusual about your environment or deployment

12.1.2 Finding Contributions to Work on

Looking at the existing issues is a great way to find something to contribute on. We created a project board to help you find issues that are good for newcomers, see the Contributing Board.

12.1.3 Contributing via Pull Requests

Contributions via pull requests are much appreciated. Before sending us a pull request, please ensure that:

1. Limited scope. Your PR should do one thing or one set of things. Avoid adding "random fixes" to PRs. Put those on separate PRs.
2. Give your PR a descriptive title. Add a short summary, if required.
3. Make sure the pipeline is green.
4. New code = new tests. If you are adding new functionality, always make sure to add some tests exercising the code and serving as live documentation of your original intention.
5. If a PR is merged all commits will get squashed, a linear commit history is not required.

6. Once a PR got reviewed, please don't do force pushes as this will break github UI and subsequent reviews will take more time.

To send us a pull request, please:

1. Fork the repository.
2. Modify the source; please focus on the specific change you are contributing. If you also reformat all the code, it will be hard for us to focus on your change.
3. Ensure local tests pass. (`colcon test` and `pre-commit run` (requires you to install pre-commit by `pip3 install pre-commit`)
4. Commit to your fork using clear commit messages.
5. Send a pull request, answering any default questions in the pull request interface.
6. Pay attention to any automated CI failures reported in the pull request, and stay involved in the conversation.

GitHub provides additional documentation on forking a repository and creating a pull request.

12.1.4 Rules for the Repositories and Process of Merging Pull Requests

This section targets maintainers, but you are also welcome to read it to understand the process of how we handle PRs in our organization. This guideline is especially applicable for the following repositories:

- ros2_control,
- ros2_controllers,
- ros2_control_demos.

Please keep the following in mind:

1. Please work from your fork when submitting PR. That way, we are keeping the main repo clean from feature branches.
2. Each PR should have all checks satisfied before they can be considered for merging.
3. Each PR must be approved by two maintainers (explicitly, please!). Only exceptions are PR's from other active maintainers in the repository, where one approval backed up with traceable discussion is sufficient.
4. There is no need to do "squash and merge" of commits to your PR. We will squash the commits when merging the PR into the head branch.
5. Please do not do "cowboy-style" PR merges over the weekend. It doesn't matter how trivial PR is. Give people a chance to do a proper review and comment on it.
6. Be aware of the impact a PR has and give other maintainers and contributors sufficient time for the review proportional to its impact. Ping them if necessary, repeatedly if necessary.

```
Even if you are not a maintainer, you are still encouraged to review pull requests.
This helps us increase the review pace and increase code quality.
Also, you are very likely to find some issues/limitations nobody else is seeing.
```

12.1.5 Writing Documentation

We use Sphinx with Read The Docs theme for documentation.

General information is located in the control.ros.org repository, while the documentation for the packages is written in the respective repositories.

12.1.6 Licensing

Any contribution that you make to this project will be under the Apache 2 License, as dictated by that license:

```
5. Submission of Contributions. Unless You explicitly state otherwise,
any Contribution intentionally submitted for inclusion in the Work
by You to the Licensor shall be under the terms and conditions of
this License, without any additional terms or conditions.
Notwithstanding the above, nothing herein shall supersede or modify
the terms of any separate license agreement you may have executed
with Licensor regarding such Contributions.
```

12.1.7 Repository Structure and CI configuration

Three build stages are checking the current and future compatibility of the framework.

1. `binary` - against released packages (main and testing) in ROS distributions. This shows that direct local build is possible.
2. `semi-binary` - against released core ROS packages (main and testing), but the immediate dependencies are pulled from the source. This shows that local build with dependencies is possible, and if it fails there, we can expect that after the next package sync, we will not be able to build.
3. `source` - also core ROS packages are build from source. It shows potential issues in the mid future.

Each repository has two types of branches: development, and stable. PR's should always be submitted against the development branch. When the PR is accepted, and there are no API and ABI changes to a stable branch, the maintainers will consider a backport to the stable branches.

We use the following naming conventions for branches:

Development branch:

- Name: `master`
- CI rule for merge:
  - must: `semi-binary` (working against development branch of ros2_control)
  - good: `binary` (working against the same stable branch of other ros2_control repositories)
- `source` build each day check against master branches of ROS 2

Stable branches:

- Name: `<ros_distro>` (e.g., humble, jazzy)
- CI rule for merge:
  - must: `semi-binary` (working against the same stable branch of other ros2_control repositories)
  - must: `binary` (working against released versions of ros2_control) - except for adding new non-braking features

- `source` build each day against distribution branches

CHAPTER

THIRTEEN

PROJECT GOVERNANCE

Table of Contents

- *Current ros-controls PMC Constituents*
- *Current ros-controls Committers*
- *Past ros-controls PMC Constituents*
- *Repositories managed by the ros-controls PMC*
- *Releases, Versioning, and Public API*
  - *Versioning*
  - *Public API declaration*
  - *Deprecation strategy*

Since 2025, the ros-controls project has been governed by the Open Source Robotics Alliance (OSRA). The information below is meant to give a quick overview of the project governance, but for full information please see the OSRA's website.

The ros-controls Project Management Committee is responsible for the day-to-day operations of the ros-controls project. The ros-controls PMC consists of the Project Leader, the ros-controls PMC Members (who have full voting rights), a Supporting Individual Representative, and the Chair of the TGC. The project also has Committers, who help manage one or more repositories but are not a part of the PMC. The Project Leader, all PMC Members, and all Committers are chosen on a meritocratic basis.

The day-to-day operations of the ros-controls PMC include managing the members and committers, managing the repositories that make up ros-controls, reviewing and merging code from the ros-controls community, maintaining the repositories, and making technical decisions that decide the direction of the project.

For more details about the ros-controls PMC, please see the Charter for the ros-controls Project.

13.1 Current ros-controls PMC Constituents

The ros-controls PMC currently consists of the following constituents:

Name	Affiliation	GitHub handle	PMC role	Time Zone (optional)
Bence Maygar	Locus Robotics	bmagyar	Project Leader	GMT (UTC+0)
Denis Stogl	b»robotized	destogl	Project Co-Leader	CET (UTC+1)/CEST (UTC+2)
Christoph Fröhlich	AIT - Austrian Institute of Technology GmbH	christoph-froehlich	Member	CET (UTC+1)/CEST (UTC+2)
Sai Kishor Kothakota	PAL Robotics S.L	saikishor	Member	CET (UTC+1)/CEST (UTC+2)
Marq Rasmussen	Locus Robotics	MarqRazz	Member	MST (UTC-7)/MDT (UTC-6)

13.2 Current ros-controls Committers

The ros-controls committers (who are not also part of the ros-controls PMC) consists of the following constituents:

Name	Affiliation	GitHub handle	Time Zone (optional)
Alejandro Hernandez Cordero	Honu Robotics	ahcorde	CET (UTC+1)/CEST (UTC+2)
Julia Jia	Independent	Juliaj	PST (UTC-8)/PDT (UTC-7)
Nathan Dunkelberger	NASA JSC Robotics	ndunkelb-nasa	CST (UTC-6)/CDT (UTC-5)
Emma Zemler	NASA JSC Robotics	ezemler-nasa	CST (UTC-6)/CDT (UTC-5)
Michael Tobia	NASA JSC Robotics	mtobia-nasa	CST (UTC-6)/CDT (UTC-5)
Erik Holum	NASA JSC Robotics	eholum-nasa	EST (UTC-5)/EDT (UTC-4)

13.3 Past ros-controls PMC Constituents

The ros-controls PMC thanks the following past constituents for their service:

Name	PMC role	GitHub handle (optional)
None yet	None yet	None yet

13.4 Repositories managed by the ros-controls PMC

The following repositories are managed by the ros-controls PMC:

Repository URL	Committers
https://github.com/ros-controls/ros2_control	None yet
https://github.com/ros-controls/ros2_controllers	None yet
https://github.com/ros-controls/ros2_control_cmake	None yet
https://github.com/ros-controls/ros2_control_ci	None yet
https://github.com/ros-controls/ros2_control_demos	None yet
https://github.com/ros-controls/control_msgs	None yet
https://github.com/ros-controls/control_toolbox	None yet
https://github.com/ros-controls/control.ros.org	None yet
https://github.com/ros-controls/gz_ros2_control	Alejandro Hernandez Cordero
https://github.com/ros-controls/kinematics_interface	None yet
https://github.com/ros-controls/realtime_tools	None yet
https://github.com/ros-controls/topic_based_hardware_interfaces	Marq Rasmussen
https://github.com/ros-controls/onnxruntime_vendor	Julia Jia
https://github.com/ros-controls/.github	None yet
https://github.com/ros-controls/mujoco_ros2_control	Nathan Dunkelberger, Emma Zemler, Michael Tobia, Erik Holum

13.5 Releases, Versioning, and Public API

As ros-controls PMC is independent of the ROS PMC we thrive to follow its strategy but have to adapt it according to our needs. This includes an asynchronous release cycle, where ros-controls repositories have reached a stable state for a ROS distro on **1st of October after an official ROS distribution release**. It is very likely that ros2_control packages will be available via the ROS build farm earlier than this date.

This stable release for a ROS distro, called **stable ros2_control release** from now, will be announced on ROS discourse on time.

13.5.1 Versioning

We will use the ROS-specific rules on top of `semver's` for versioning, but also adhere to some ros-controls-specific rules:

- Major version increments (i.e. API breaking changes) should not be made within a **stable ros2_control release**.
- ros2_control heavily relies on the usage of pluginlib. Therefore, we distinguish two types of compiled code: non-plugin code together with plugin base classes, and plugins itself.
  - ABI of plugins may change at every release, i.e., also within a **stable ros2_control release**. Plugins built by the buildfarm will still be loaded by pluginlib's class loader, but code linking against the exported libraries will break.
  - For non-plugin code, ABI breaks within a **stable ros2_control release** are less likely but still unavoidable to fix code, which is critical regarding safety aspects of robot control.

Important:

- Always update your full ROS installation, not only a single package. For example, on Ubuntu run `sudo apt update && sudo apt upgrade` after you install a new package.
- Recompile your custom code after updating any upstream ROS packages.

- The same applies for run-time behavior changes: Changes within a **stable ros2_control release** are less likely but still unavoidable to fix safety-critical behavior. Where possible, we try to keep the legacy behavior configurable together with a deprecation warning, see section below.

The ros2_control maintainer team considers safety a top priority and bugs or issues found in the framework may be back ported to distros regardless of API stability. These issues will be discussed at the biweekly PMC meeting where the community can decide the best route forward.

13.5.2 Public API declaration

According to `semver`, every package must clearly declare a public API.

- For most C++ packages the declaration is any header that it installs. Private class members and methods are not part of the public API.
- For other languages like Python, a public API must be explicitly defined, so that it is clear what symbols can be relied on with respect to the versioning guidelines.

If something you are using is not explicitly listed as part of the public API in the package's documentation, then you cannot depend on it not changing between minor or patch versions.

13.5.3 Deprecation strategy

Where possible, we will use the tick-tock deprecation and migration strategy for breaking changes (API or behavior-breaking changes).

- New deprecations can be run-time messages or compiler warnings expressing that the functionality is being deprecated. The functionality will be completely removed in any future release, or at latest in the next **stable ros2_control release** (there may be details in the deprecation note).
- New deprecations can also come in every release of **stable ros2_control release** by performing backports of changes from the rolling version. These are meant to help users migrate early, however the functionality will remain available in that specific ROS distribution.

Have a look at *Release Notes* and *Migration Guides*, where we will highlight necessary changes within every ros2_control version of a ROS distro.

Important: Don't use compiler flags like `-Wall -Werror` in your development environment, as they may cause unnecessary build failures if deprecation notes are added.

Example of function `foo` deprecated and replaced by function `bar`:

Package version	Description	API
x.y.z, <x-1>.y.z	Original version	`void foo();`
x.<y+1>.z	New feature including deprecation	`[[deprecated("use bar()")]] void foo();` `void bar();`
<x-1>.<y+1>.z	Backport to **stable ros2_control release**	`[[deprecated("use bar()")]] void foo();` `void bar();`
x.<y+2>.z	New release of development version	`void bar();`
<x-1>.<y+2>.z	New release of **stable ros2_control release**	`[[deprecated("use bar()")]] void foo();` `void bar();`

CHAPTER

FOURTEEN

ACKNOWLEDGEMENTS

14.1 Maintainers

The following people were maintaining the `ros2_control` framework, showing their review activity and contributions:

Recent Contributions

Contributions during the past 12 months

All-Time Contrib

All-time contributions

Recent Reviews

Reviews during the past 12 months

All-Time Reviews

All-time reviews

14.2 Contributors

The following people have contributed to the development of this project by providing valuable reviews or by submitting pull requests, see *Contributing* for more information.

Recent Contributions

Contributions during the past 12 months

All-Time Contrib

All-time contributions

Recent Reviews

Reviews during the past 12 months

All-Time Reviews

All-time reviews

14.3 Companies and Institutions

The project has received major contributions from the following companies and institutions.

Supported by ROSIN - ROS-Industrial Quality-Assured Robot Software Components. More information: https://cordis.europa.eu/project/id/732287

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 732287.

CHAPTER

FIFTEEN

DOCUMENTATION USAGE

CHAPTER

SIXTEEN

DOCUMENTATION DOWNLOADS

The documentation can be downloaded to read offline or on another device.

PDF downloads are published on the generated Rolling HTML site.

The ros2_control is a framework for (real-time) control of robots using (ROS 2). Its packages are a rewrite of ros_control packages used in ROS (Robot Operating System). ros2_control's goal is to simplify integrating new hardware and overcome some drawbacks.

If you are not familiar with the control theory, please get some idea about it (e.g., at Wikipedia) to get familiar with the terms used in this manual.

CHAPTER

SEVENTEEN

ROS2_CONTROL REPOSITORIES

The framework consists of the following Github repositories hosted under the ros-controls Github organization:

- ros2_control - the main interfaces and components of the framework;
- ros2_controllers - widely used controllers, such as forward command controller, joint trajectory controller, differential drive controller;
- control_toolbox - some widely-used control theory implementations (e.g. PID) used by controllers;
- realtime_tools - general toolkit for realtime support, e.g., realtime buffers and publishers;
- control_msgs - common messages;
- kinematics_interface - for using C++ kinematics frameworks;
- gz_ros2_control - Plugin for Gazebo;
- topic_based_hardware_interfaces - hardware_interfaces for simulators and other hardware that only support ROS topic-based communication;
- mujoco_ros2_control - Plugin for MuJoCo;
- onnxruntime_vendor - Vendor package for ONNX Runtime.

Additionally, the following (unreleased) packages are relevant for documentation and project management:

- ros2_control_demos - example implementations of common use-cases for a smooth start;
- roadmap - planning and design docs for the project;
- ros2_control_ci - reusable Github actions and Docker images for Ubuntu, RHEL, and Debian CI jobs;
- .github - Github organization-wide files, such as issue templates;
- ros2_control_cmake - CMake macros for the project;
- control.ros.org - this documentation page.

CHAPTER

EIGHTEEN

DEVELOPMENT ORGANISATION AND COMMUNICATION

OSRA
The ros-controls project is governed by the Open Source Robotics Alliance (OSRA). See *Project Governance* for more details.

Questions
Please use Robotics Stack Exchange and tag your questions with `ros2_control`.

PMC Meeting
Every second Wednesday there is a PMC meeting. To join the meeting check the announcement on ROS Discourse. You can join the meeting through google groups or directly on Zoom (check the announcement). To propose new discussion points, or review notes from previous meetings, check this document.

Projects
GitHub projects under ros-control organization are used to track the work.

Bug reports and feature requests
Use the issue tracker in the corresponding repository for this. Give a short summary of the problem Make sure to provide a minimal list of steps one can follow to reproduce the issue you found Provide relevant information regarding the operating system, ROS distribution, etc.

General discussions
Please use ROS Discourse.

Built on 2026-04-14 at 10:36 GMT