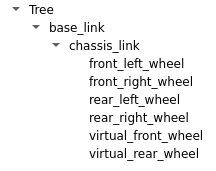
Great! Now that you're ready to experiment with **Example 11** of the ros2\_control\_demo\_example\_11 package, here’s a structured list of tasks designed to help you understand and modify the system incrementally. Each task builds on the previous one, enabling you to explore different aspects of the example.

ros2 launch ros2\_control\_demo\_example\_11 carlikebot.launch.py remap\_odometry\_tf:=true

**Task 1: Understand the Current Setup**

**Steps:**

1. **Launch Example 11:**
2. ros2 launch ros2\_control\_demo\_example\_11 demo\_robot.launch.py
3. **Verify Components:**
   * Open RViz and visualize the robot model (RobotModel display).
4. Add the **TF** display to inspect the TF tree. 
   * Use:
   * ros2 topic echo /joint\_states

---

header:

stamp:

sec: 1733074770

nanosec: 790501909

frame\_id: ''

name:

- virtual\_rear\_wheel\_joint

- rear\_right\_wheel\_joint

- rear\_left\_wheel\_joint

- virtual\_front\_wheel\_joint

- front\_right\_wheel\_joint

- front\_left\_wheel\_joint

position:

- 0.0

- 0.0

- 0.0

- 0.0

- 0.0

- 0.0

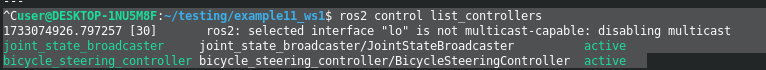
velocity: []

effort: []

---

Verify the joints publish position, velocity, or both.

1. **Inspect Available Controllers:**
2. ros2 control list\_controllers
   * Note which controllers are active and their types.



**Expected Outcome:**

* Understand the initial robot model, its joints, and active controllers.

**Task 2: Experiment with Joint Commands**

**Steps:**

1. **Send Commands to the Joints:** **Position Command (e.g., steering):**

ros2 topic pub /bicycle\_steering\_controller/reference geometry\_msgs/TwistStamped "twist:

linear:

x: 1.0

y: 0.0

z: 0.0

angular:

x: 0.0

y: 0.0

z: 0.5"

This moves the virtual\_front\_wheel\_joint linear.x: Forward velocity (1.0 m/s).

• angular.z: Turning rate (0.5 rad/s)

**Velocity Command (e.g., rear wheel):**

1. **Visualize Changes:**
   * Use RViz to see the robot model reflect changes.
   * Confirm state updates with:

ros2 topic echo /joint\_states

header:

stamp:

sec: 1733075711

nanosec: 922421853

frame\_id: ''

name:

- virtual\_front\_wheel\_joint

- virtual\_rear\_wheel\_joint

position:

- 0.16109190453758052

- 6948.004279820011

velocity:

- .nan

- 20.0

effort:

- .nan

- .nan

**Expected Outcome:**

* Observe joint movements in RViz or joint state updates on the /joint\_states topic.

**Experiment Tasks**

**Task 1: Analyze the Current State**

1. **Inspect Controllers:**

ros2 control list\_controllers – done

* + **Expected Output:** You already have:
    - joint\_state\_broadcaster (publishes joint states).
    - bicycle\_steering\_controller (handles the Ackermann-style steering).

1. **Inspect Parameters:**

bash

Copy code

ros2 param list /bicycle\_steering\_controller

user@DESKTOP-1NU5M8F:~/testing/example11\_ws1$ ros2 param list /bicycle\_steering\_controller

1733081328.056233 [30] ros2: selected interface "lo" is not multicast-capable: disabling multicast

base\_frame\_id

enable\_odom\_tf

front\_steering

front\_wheel\_radius

front\_wheels\_names

front\_wheels\_state\_names

odom\_frame\_id

open\_loop

pose\_covariance\_diagonal

position\_feedback

rear\_wheel\_radius

rear\_wheels\_names

rear\_wheels\_state\_names

reference\_timeout

twist\_covariance\_diagonal

update\_rate

use\_sim\_time

use\_stamped\_vel

velocity\_rolling\_window\_size

wheelbase

* + **Focus Areas:**
    - Parameters like wheelbase, front\_steering, and rear\_wheel\_radius determine your robot's behavior. Test their impact in later tasks.

1. **Verify Joint States:**

ros2 topic echo /joint\_states

* + **Expected Output:** Joint positions (virtual\_front\_wheel\_joint) and velocities (virtual\_rear\_wheel\_joint) should be updating based on commands.

---

header:

stamp:

sec: 1733081531

nanosec: 565535758

frame\_id: ''

name:

- virtual\_front\_wheel\_joint

- virtual\_rear\_wheel\_joint

position:

- 0.0

- 0.0

velocity:

- .nan

- 0.0

effort:

- .nan

- .nan

---

snapshot of a ROS2 **/joint\_states** message, which contains information about the position, velocity, and effort for each joint. However, the presence of .nan (Not a Number) indicates potential issues with the hardware interface or simulation

Why virtual wheel?

The **virtual wheels** in your system are placeholders that simplify control and simulation. Instead of directly controlling all individual wheels (like the left and right rear wheels), you control a single virtual wheel that represents their combined behavior. This setup allows you to calculate motion dynamics (like velocity or position) centrally for the virtual wheel and then mimic this motion to the actual physical wheels. It reduces complexity and ensures consistency between how the robot moves and how its wheels are modeled, especially in systems like Ackermann steering where front and rear wheels have distinct roles.

**Task 2: Explore Hardware Plugin Integration**

**Steps:**

1. **Locate the Hardware Plugin Code:**
   * Open ros2\_control\_demo\_example\_11/src/CarlikeBotSystemHardware.cpp.
   * Focus on the read() and write() methods.
2. **Understand Plugin Methods:**
   * **Initialization (on\_init):** Sets up the plugin, defining joint names and hardware parameters.
   * **Read Method:**
     + Reads hardware state (e.g., joint positions, velocities).
     + Example:

cpp

Copy code

state\_interfaces\_.emplace\_back("virtual\_rear\_wheel\_joint/position", &rear\_wheel\_position\_);

* + **Write Method:**
    - Sends commands (e.g., velocities) to hardware.
    - Example:

cpp

Copy code

velocity\_commands\_["virtual\_rear\_wheel\_joint"] = \*command\_velocity\_;

1. **Modify the Plugin:**
   * Add print statements in the read() and write() methods to observe interactions:

cpp

Copy code

RCLCPP\_INFO(rclcpp::get\_logger("Hardware"), "Writing velocity command: %f", command\_velocity\_);

1. **Rebuild and Test:**

bash

Copy code

colcon build --packages-select ros2\_control\_demo\_example\_11

ros2 launch ros2\_control\_demo\_example\_11 demo\_robot.launch.py

it worksd,

To print the values from the read() and write() methods in your hardware interface, you need to ensure the logging statements within these methods are executed. Since both methods already contain std::stringstream logging, you simply need to run the ROS2 system and trigger these methods by performing some control or state-related operations.

Here's how you can do it step by step:

**1. Enable Logging**

Make sure the RCLCPP\_INFO\_THROTTLE statements in your methods are set up correctly. In the given code:

* **read()** logs the current state of the joints (position and velocity).
* **write()** logs the commands sent to the joints.

For debugging purposes, you can temporarily replace RCLCPP\_INFO\_THROTTLE with RCLCPP\_INFO to log values on every call.

**Updated Code:**

**In read():**

RCLCPP\_INFO(get\_logger(), "Reading states:\n"

"\tposition: %.2f for joint '%s'\n"

"\tposition: %.2f for joint '%s'\n"

"\tvelocity: %.2f for joint '%s'",

hw\_interfaces\_["steering"].state.position, hw\_interfaces\_["steering"].joint\_name.c\_str(),

hw\_interfaces\_["traction"].state.position, hw\_interfaces\_["traction"].joint\_name.c\_str(),

hw\_interfaces\_["traction"].state.velocity, hw\_interfaces\_["traction"].joint\_name.c\_str());

**In write():**

RCLCPP\_INFO(get\_logger(), "Writing commands:\n"

"\tposition: %.2f for joint '%s'\n"

"\tvelocity: %.2f for joint '%s'",

hw\_interfaces\_["steering"].command.position, hw\_interfaces\_["steering"].joint\_name.c\_str(),

hw\_interfaces\_["traction"].command.velocity, hw\_interfaces\_["traction"].joint\_name.c\_str());

**2. Rebuild the Package**

Rebuild your package to apply these changes:

colcon build --packages-select ros2\_control\_demo\_example\_11

**3. Launch Your System**

Start the robot system with the relevant launch file:

ros2 launch ros2\_control\_demo\_example\_11 demo\_robot.launch.py

**4. Trigger Read and Write Cycles**

Perform operations that invoke the read() and write() methods:

* Send commands to the robot using controllers or directly publish commands to the relevant topics.

**Example Command:**

Publish a steering position and rear-wheel velocity:

ros2 topic pub /bicycle\_steering\_controller/reference geometry\_msgs/TwistStamped "{twist: {linear: {x: 1.0}, angular: {z: 0.5}}}"

**5. Monitor the Logs**

View the output in the terminal where your launch file was run. You should see logs similar to:

**From read():**

[INFO] [<timestamp>] [CarlikeBotSystemHardware]: Reading states:

position: 0.16 for joint 'steering\_joint'

position: 6948.00 for joint 'traction\_joint'

velocity: 20.00 for joint 'traction\_joint'

**From write():**

[INFO] [<timestamp>] [CarlikeBotSystemHardware]: Writing commands:

position: 0.50 for joint 'steering\_joint'

velocity: 1.00 for joint 'traction\_joint'

**6. Verify and Analyze**

* **Check Correctness:**
  + Ensure values for positions and velocities are reasonable.
  + Confirm that read() updates states based on write() commands and simulation period.
* **Troubleshoot Discrepancies:**
  + If values don’t update as expected, debug the read() and write() methods to ensure they correctly compute and handle commands and states.

**Optional: Log to File**

If you want to save the output for analysis:

1. Start your system with logs redirected to a file:
2. ros2 launch ros2\_control\_demo\_example\_11 demo\_robot.launch.py > hardware\_log.txt
3. Analyze the hardware\_log.txt file for logged values.

This approach ensures that you can monitor and debug the values being processed by your hardware interface in a straightforward manner. Let me know if you encounter specific issues!

**Task 3: Simulate Hardware Latency**

**Steps:**

1. **Modify Plugin Timing:**
   * Add artificial latency to the read() method in the plugin:

cpp

Copy code

std::this\_thread::sleep\_for(std::chrono::milliseconds(50));

* + This simulates real-world delays in reading hardware states.

1. **Observe System Impact:**
   * Send commands to the joints:

bash

Copy code

ros2 topic pub /bicycle\_steering\_controller/reference geometry\_msgs/TwistStamped "{twist: {linear: {x: 1.0}, angular: {z: 0.5}}}"

* + Monitor delays in /joint\_states updates.

1. **Experiment with Different Delays:**
   * Gradually increase the delay and observe controller behavior and system performance.

**Task 4: Add a New Joint with Hardware Interface**

**Steps:**

1. **Add a Joint in URDF:**

xml

Copy code

<joint name="extra\_joint" type="continuous">

<parent link="chassis\_link"/>

<child link="extra\_link"/>

<origin xyz="0 0 0.1" rpy="0 0 0"/>

<axis xyz="1 0 0"/>

</joint>

1. **Register the Joint in the Plugin:**
   * Add the joint to the on\_init method:

joint\_names\_.emplace\_back("extra\_joint");

command\_interfaces\_.emplace\_back("extra\_joint/position", &extra\_joint\_command\_);

state\_interfaces\_.emplace\_back("extra\_joint/position", &extra\_joint\_state\_);

1. **Rebuild and Test:**
   * Send commands to the new joint using a custom controller or via direct topic publishing.

Joint added in urdf

