:github_url: https://github.com/ros-controls/ros2_control_demos/blob/{REPOS_FILE_BRANCH}/example_1/doc/userdoc.rst

.. _ros2_control_demos_example_1_userdoc:

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.

.. include:: ../../doc/run_from_docker.rst

Tutorial steps
----

1. (Optional) To check that *RRBot* descriptions are working properly use following launch commands

   .. tabs::

      .. group-tab:: Local

         .. code-block:: shell

           ros2 launch ros2_control_demo_example_1 view_robot.launch.py

      .. group-tab:: Docker

        Let's start with the docker container by running the following command:

        .. code-block:: shell

          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.

```
.. code-block:: shell

  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

```
.. code-block:: shell

  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*.

```
.. image:: rrbot.png
  :width: 400
  :alt: Revolute-Revolute Manipulator Robot
```

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

```
.. tabs::

   .. group-tab:: Local

      .. code-block:: shell

        ros2 launch ros2_control_demo_example_1 rrbot.launch.py
```

The launch file loads and starts the robot hardware, controllers and opens *RViz*.

```
      .. group-tab:: Docker

        .. code-block:: shell

          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

   .. tabs::

      .. group-tab:: Local

        .. code-block:: shell

          ros2 control list_hardware_interfaces

      .. group-tab:: Docker

        Open a bash terminal inside the already running docker container
by

        .. code-block:: shell

          docker exec -it ros2_control_demos ./entrypoint.sh bash

        and run the command

        .. code-block:: shell

          ros2 control list_hardware_interfaces

   If everything started nominally, you should see the output

   .. code-block:: shell

    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

```
   .. tabs::

      .. group-tab:: Local

        .. code-block:: shell

          ros2 control list_controllers

      .. group-tab:: Docker

        (from the docker terminal, see above)

        .. code-block:: shell

          ros2 control list_controllers
```

   You will see the two controllers in active state

```
   .. code-block:: shell

    joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] 
active
    forward_position_controller[forward_command_controller/ForwardCommandC
ontroller] active
```

5. If you get output from above you can send commands to *Forward Command Controller*, either:

   a. Manually using ROS 2 CLI interface:

```
   .. tabs::

      .. group-tab:: Local

        .. code-block:: shell

          ros2 topic pub /forward_position_controller/commands 
std_msgs/msg/Float64MultiArray "data:
          -  0.5
          -  0.5"

      .. group-tab:: Docker
```

```
        Inside the docker terminal from above, run the command

        .. code-block:: shell

          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:

   .. tabs::

      .. group-tab:: Local

        .. code-block:: shell

          ros2 launch ros2_control_demo_example_1
test_forward_position_controller.launch.py

      .. group-tab:: Docker

        Inside the docker terminal from above, run the command

        .. code-block:: shell

          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.

   .. code-block:: shell

    [RRBotSystemPositionOnlyHardware]: Got command 0.50000 for joint 0!
    [RRBotSystemPositionOnlyHardware]: Got command 0.50000 for joint 1!

   If you echo the ``/joint_states`` or ``/dynamic_joint_states`` topics
you should now get similar values, namely the simulated states of the
robot

   .. tabs::

      .. group-tab:: Local

        .. code-block:: shell

          ros2 topic echo /joint_states
          ros2 topic echo /dynamic_joint_states

      .. group-tab:: Docker
```

```
        Inside the docker terminal from above, run the command

        .. code-block:: shell

          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

   .. tabs::

      .. group-tab:: Local

        .. code-block:: shell

          ros2 control load_controller
joint_trajectory_position_controller

      .. group-tab:: Docker

        (from the docker terminal, see above)

        .. code-block:: shell

          ros2 control load_controller
joint_trajectory_position_controller

   what should return ``Successfully loaded controller
joint_trajectory_position_controller``. Check the status with

   .. tabs::

      .. group-tab:: Local

        .. code-block:: shell

          ros2 control list_controllers

      .. group-tab:: Docker

        (from the docker terminal, see above)

        .. code-block:: shell

          ros2 control list_controllers

   what shows you that the controller is loaded but unconfigured.

   .. code-block:: shell
```

```
    joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]
active
    forward_position_controller[forward_command_controller/ForwardCommandC
ontroller] active
    joint_trajectory_position_controller[joint_trajectory_controller/Joint
TrajectoryController] unconfigured

   Configure the controller by setting it ``inactive`` by

   .. tabs::

      .. group-tab:: Local

        .. code-block:: shell

          ros2 control set_controller_state
joint_trajectory_position_controller inactive

      .. group-tab:: Docker

        (from the docker terminal, see above)

        .. code-block:: shell

          ros2 control set_controller_state
joint_trajectory_position_controller inactive

   what should give ``Successfully configured
joint_trajectory_position_controller``.

   .. note::

     The parameters are already set in `rrbot_controllers.yaml
<https://github.com/ros-
controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_1/bringup/con
fig/rrbot_controllers.yaml>`__
     but the controller was not loaded from the `launch file
rrbot.launch.py <https://github.com/ros-
controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_1/bringup/lau
nch/rrbot.launch.py>`__ before.

   As an alternative, you can load the controller directly in
``inactive``-state by means of the option for ``load_controller`` with

   .. tabs::

      .. group-tab:: Local

        .. code-block:: shell

          ros2 control load_controller
joint_trajectory_position_controller --set-state inactive

      .. group-tab:: Docker
```

```
        (from the docker terminal, see above)

        .. code-block:: shell

          ros2 control load_controller
joint_trajectory_position_controller --set-state inactive

   You should get the result ``Successfully loaded controller
joint_trajectory_position_controller into state inactive``.

   See if it loaded properly with

   .. tabs::

      .. group-tab:: Local

        .. code-block:: shell

          ros2 control list_controllers

      .. group-tab:: Docker

        (from the docker terminal, see above)

        .. code-block:: shell

          ros2 control list_controllers

   what should now return

   .. code-block:: shell

     joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]
active
     forward_position_controller[forward_command_controller/ForwardCommandC
ontroller] active
     joint_trajectory_position_controller[joint_trajectory_controller/Joint
TrajectoryController] inactive

   Note that the controller is loaded but still ``inactive``. Now you can
switch the controller by

   .. tabs::

      .. group-tab:: Local

        .. code-block:: shell

          ros2 control set_controller_state forward_position_controller
inactive
          ros2 control set_controller_state
joint_trajectory_position_controller active
```

```
      .. group-tab:: Docker

        (from the docker terminal, see above)

        .. code-block:: shell

          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

   .. tabs::

      .. group-tab:: Local

        .. code-block:: shell

          ros2 control switch_controllers --activate
joint_trajectory_position_controller --deactivate
forward_position_controller

      .. group-tab:: Docker

        (from the docker terminal, see above)

        .. code-block:: shell

          ros2 control switch_controllers --activate
joint_trajectory_position_controller --deactivate
forward_position_controller

   Again, check via

   .. tabs::

      .. group-tab:: Local

        .. code-block:: shell

          ros2 control list_controllers

      .. group-tab:: Docker

        (from the docker terminal, see above)

        .. code-block:: shell

          ros2 control list_controllers

   what should now return

   .. code-block:: shell
```

```
    joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]
active
    forward_position_controller[forward_command_controller/ForwardCommandC
ontroller] inactive
    joint_trajectory_position_controller[joint_trajectory_controller/Joint
TrajectoryController] active

   Send a command to the controller using demo node, which sends four
goals every 6 seconds in a loop with

   .. tabs::

      .. group-tab:: Local

        .. code-block:: shell

          ros2 launch ros2_control_demo_example_1
test_joint_trajectory_controller.launch.py

      .. group-tab:: Docker

        (from the docker terminal, see above)

        .. code-block:: shell

          ros2 launch ros2_control_demo_example_1
test_joint_trajectory_controller.launch.py

   You can adjust the goals in `rrbot_joint_trajectory_publisher
<https://github.com/ros-
controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_1/bringup/con
fig/rrbot_joint_trajectory_publisher.yaml>`__.

Files used for this demos
-------------------------

*  Launch file: `rrbot.launch.py <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_1/bringup/
   launch/rrbot.launch.py>`__
*  Controllers yaml: `rrbot_controllers.yaml <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_1/bringup/
   config/rrbot_controllers.yaml>`__
*  URDF file: `rrbot.urdf.xacro <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_1/descript
   ion/urdf/rrbot.urdf.xacro>`__

  *  Description: `rrbot_description.urdf.xacro <https://github.com/ros-
     controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_dem
     o_description/rrbot/urdf/rrbot_description.urdf.xacro>`__
  *  ``ros2_control`` tag: `rrbot.ros2_control.xacro
     <https://github.com/ros-
     controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_1/descri
     ption/ros2_control/rrbot.ros2_control.xacro>`__
```

* RViz configuration: `rrbot.rviz <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_description/rrbot/rviz/rrbot.rviz>`__
* Test nodes goals configuration:

  + `rrbot_forward_position_publisher <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_1/bringup/config/rrbot_forward_position_publisher.yaml>`__
  + `rrbot_joint_trajectory_publisher <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_1/bringup/config/rrbot_joint_trajectory_publisher.yaml>`__

* Hardware interface plugin: `rrbot.cpp <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_1/hardware/rrbot.cpp>`__

Controllers from this demo
----

  * ``Joint State Broadcaster`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/joint_state_broadcaster>`__): :ref:`doc <joint_state_broadcaster_userdoc>`
  * ``Forward Command Controller`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/forward_command_controller>`__): :ref:`doc <forward_command_controller_userdoc>`
  * ``Joint Trajectory Controller`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/joint_trajectory_controller>`__): :ref:`doc <joint_trajectory_controller_userdoc>`

:github_url: https://github.com/ros-controls/ros2_control_demos/blob/{REPOS_FILE_BRANCH}/example_2/doc/userdoc.rst

.. _ros2_control_demos_example_2_userdoc:

****
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.

.. include:: ../../doc/run_from_docker.rst

Tutorial steps
----

1. To check that *DiffBot* description is working properly use following launch commands

    .. code-block:: shell

     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.

    .. image:: diffbot.png
     :width: 400
     :alt: Differential Mobile Robot

2. To start *DiffBot* example open a terminal, source your ROS2-workspace and execute its launch file with

    .. code-block:: shell

     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

   .. code-block:: shell

```
ros2 control list_hardware_interfaces
```

   You should get

   .. code-block:: shell

```
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

   .. code-block:: shell

```
ros2 control list_controllers
```

   You should get

   .. code-block:: shell

```
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:

.. code-block:: shell

```
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"
```

You should now see an orange box circling in *RViz*.
Also, you should see changing states in the terminal where launch file is started.

.. code-block:: shell

```
[DiffBotSystemHardware]: Got command 43.33333 for 'left_wheel_joint'!
[DiffBotSystemHardware]: Got command 50.00000 for 'right_wheel_joint'!
```

6. Let's introspect the ros2_control hardware component. Calling

.. code-block:: shell

```
ros2 control list_hardware_components
```

should give you

.. code-block:: shell

```
Hardware Component 1
        name: DiffBot
        type: system
        plugin name: ros2_control_demo_example_2/DiffBotSystemHardware
        state: id=3 label=active
        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

.. code-block:: shell

```
    ros2 launch ros2_control_demo_example_2 diffbot.launch.py
use_mock_hardware:=True

  Calling

  .. code-block:: shell

    ros2 control list_hardware_components

  now should give you

  .. code-block:: shell

    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 <https://github.com/ros-
controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_2/description
/ros2_control/diffbot.ros2_control.xacro>`__, one can find the
  instructions to load this plugin together with the parameter
``calculate_dynamics``.

  .. code-block:: xml

    <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.

  More information on mock_components can be found in the
:ref:`ros2_control documentation <mock_components_userdoc>`.

Files used for this demos
--------------------------

*  Launch file: `diffbot.launch.py <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_2/bringup/
   launch/diffbot.launch.py>`__
```

* Controllers yaml: `diffbot_controllers.yaml <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_2/bringup/config/diffbot_controllers.yaml>`__
* URDF file: `diffbot.urdf.xacro <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_2/description/urdf/diffbot.urdf.xacro>`__

  * Description: `diffbot_description.urdf.xacro <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_description/diffbot/urdf/diffbot_description.urdf.xacro>`__
  * ``ros2_control`` tag: `diffbot.ros2_control.xacro <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_2/description/ros2_control/diffbot.ros2_control.xacro>`__

* RViz configuration: `diffbot.rviz <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_description/diffbot/rviz/diffbot.rviz>`__

* Hardware interface plugin: `diffbot_system.cpp <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_2/hardware/diffbot_system.cpp>`__

Controllers from this demo
----

* ``Joint State Broadcaster`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/joint_state_broadcaster>`__): :ref:`doc <joint_state_broadcaster_userdoc>`
* ``Diff Drive Controller`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/diff_drive_controller>`__): :ref:`doc <diff_drive_controller_userdoc>`

:github_url: https://github.com/ros-controls/ros2_control_demos/blob/{REPOS_FILE_BRANCH}/example_3/doc/userdoc.rst

.. _ros2_control_demos_example_3_userdoc:

****
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.

.. include:: ../../doc/run_from_docker.rst

Tutorial steps
----

1. To check that *RRBot* descriptions are working properly use following launch commands

   .. code-block:: shell

    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

   .. code-block:: shell

    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

.. code-block:: shell

```
ros2 control list_hardware_interfaces
```

.. code-block:: shell

```
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

.. code-block:: shell

```
    ros2 control list_controllers

   gives

   .. code-block:: shell

    joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]
active
    forward_velocity_controller[velocity_controllers/JointGroupVelocityCon
troller] 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:

   #. Manually using ROS 2 CLI interface.

      *  when using ``forward_position_controller`` controller

        .. code-block:: shell

          ros2 topic pub /forward_position_controller/commands
std_msgs/msg/Float64MultiArray "data:
          -  0.5
          -  0.5"

      *  when using ``forward_velocity_controller`` controller (default)

        .. code-block:: shell

          ros2 topic pub /forward_velocity_controller/commands
std_msgs/msg/Float64MultiArray "data:
          -  5
          -  5"

      *  when using ``forward_acceleration_controller`` controller

        .. code-block:: shell

          ros2 topic pub /forward_acceleration_controller/commands
std_msgs/msg/Float64MultiArray "data:
          -  10
          -  10"


   #. Or you can start a demo node which sends two goals every 5 seconds
in a loop when using ``forward_position_controller`` controller

      .. code-block:: shell

         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.

.. code-block:: shell

```
   [RRBotSystemMultiInterfaceHardware]: Got the commands pos: 0.78500, vel: 0.00000, acc: 0.00000 for joint 0, control_lvl:1
   [RRBotSystemMultiInterfaceHardware]: Got the commands pos: 0.78500, vel: 0.00000, acc: 0.00000 for joint 1, control_lvl:1
   [RRBotSystemMultiInterfaceHardware]: Got pos: 0.78500, vel: 0.00000, acc: 0.00000 for joint 0!
   [RRBotSystemMultiInterfaceHardware]: Got pos: 0.78500, vel: 0.00000, acc: 0.00000 for joint 1!
```

6. 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

   .. code-block:: shell

```
 [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

   .. code-block:: shell

```
 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

.. code-block:: shell

```
    joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]
active
    forward_illegal1_controller[forward_command_controller/ForwardCommandC
ontroller] inactive
```

Files used for this demos
----

* Launch file: `rrbot_system_multi_interface.launch.py <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_3/bringup/launch/rrbot_system_multi_interface.launch.py>`__
* Controllers yaml: `rrbot_multi_interface_forward_controllers.yaml <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_3/bringup/config/rrbot_multi_interface_forward_controllers.yaml>`__
* URDF: `rrbot_system_multi_interface.urdf.xacro <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_3/description/urdf/rrbot_system_multi_interface.urdf.xacro>`__

  * Description: `rrbot_description.urdf.xacro <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_description/rrbot/urdf/rrbot_description.urdf.xacro>`__
  * ``ros2_control`` URDF tag: `rrbot_system_multi_interface.ros2_control.xacro <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_3/description/ros2_control/rrbot_system_multi_interface.ros2_control.xacro>`__

* RViz configuration: `rrbot.rviz <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_description/rrbot/rviz/rrbot.rviz>`__
* Hardware interface plugin: `rrbot_system_multi_interface.cpp <https://github.com/ros-controls/ros2_control_demos/blob/{REPOS_FILE_BRANCH}/example_3/hardware/rrbot_system_multi_interface.cpp>`__

Controllers from this demo

```
-------------------------
*   ``Joint State Broadcaster`` (`ros2_controllers repository
    <https://github.com/ros-
    controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/joint_state_broadcas
    ter>`__): :ref:`doc <joint_state_broadcaster_userdoc>`
*   ``Forward Command Controller`` (`ros2_controllers repository
    <https://github.com/ros-
    controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/forward_command_cont
    roller>`__): :ref:`doc <forward_command_controller_userdoc>`
```

:github_url: https://github.com/ros-controls/ros2_control_demos/blob/{REPOS_FILE_BRANCH}/example_4/doc/userdoc.rst

.. _ros2_control_demos_example_4_userdoc:

****
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``.

.. include:: ../../doc/run_from_docker.rst

Tutorial steps
----

1. To check that *RRBot* descriptions are working properly use following launch commands

   .. code-block:: shell

    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

   .. code-block:: shell

    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

   .. code-block:: shell

```
ros2 control list_hardware_interfaces
```

   .. code-block:: shell

```
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

   .. code-block:: shell

```
ros2 control list_hardware_components -v
```

   There is a single hardware component for the robot providing the command and state interfaces:

   .. code-block:: shell

```
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

   .. code-block:: shell

```
    ros2 control list_controllers
```

   .. code-block:: shell

```
    joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]
active
    fts_broadcaster
[force_torque_sensor_broadcaster/ForceTorqueSensorBroadcaster] active
    forward_position_controller[forward_command_controller/ForwardCommandC
ontroller] active
```

5. If you get output from above you can send commands to *Forward Command
    Controller*, either:

   #. Manually using ROS 2 CLI interface.

      .. code-block:: shell

```
        ros2 topic pub /forward_position_controller/commands
std_msgs/msg/Float64MultiArray "data:
        -  0.5
        -  0.5"
```

   #. Or you can start a demo node which sends two goals every 5 seconds
in a loop

      .. code-block:: shell

```
         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.

   .. code-block:: shell

```
    [RRBotSystemWithSensorHardware]: Got command 0.50000 for joint 0!
    [RRBotSystemWithSensorHardware]: Got command 0.50000 for joint 1!
```

6. Access wrench data from 2D FTS via

   .. code-block:: shell

```
    ros2 topic echo /fts_broadcaster/wrench
```

```
   shows the random generated sensor values, republished by *Force Torque
Sensor Broadcaster* as
   ``geometry_msgs/msg/WrenchStamped`` message

   .. code-block:: shell

    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
   <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_4/bringup/
   launch/rrbot_system_with_sensor.launch.py>`__
*  Controllers yaml: `rrbot_with_sensor_controllers.yaml
   <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_4/bringup/
   config/rrbot_with_sensor_controllers.yaml>`__
*  URDF: `rrbot_system_with_sensor.urdf.xacro <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_4/descript
   ion/urdf/rrbot_system_with_sensor.urdf.xacro>`__

  *  Description: `rrbot_description.urdf.xacro <https://github.com/ros-
     controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_dem
     o_description/rrbot/urdf/rrbot_description.urdf.xacro>`__
  *  ``ros2_control`` URDF tag:
     `rrbot_system_with_sensor.ros2_control.xacro <https://github.com/ros-
     controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_4/descri
     ption/ros2_control/rrbot_system_with_sensor.ros2_control.xacro>`__

*  RViz configuration: `rrbot.rviz <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_4/descript
   ion/rviz/rrbot.rviz>`__
```

* Hardware interface plugin: `rrbot_system_with_sensor.cpp <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_4/hardware/rrbot_system_with_sensor.cpp>`__

Controllers from this demo

* ``Joint State Broadcaster`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/joint_state_broadcaster>`__): :ref:`doc <joint_state_broadcaster_userdoc>`
* ``Forward Command Controller`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/forward_command_controller>`__): :ref:`doc <forward_command_controller_userdoc>`
* ``Force Torque Sensor Broadcaster`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/force_torque_sensor_broadcaster>`__): :ref:`doc <force_torque_sensor_broadcaster_userdoc>`

:github_url: https://github.com/ros-controls/ros2_control_demos/blob/{REPOS_FILE_BRANCH}/example_5/doc/userdoc.rst

.. _ros2_control_demos_example_5_userdoc:

****
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``.

.. include:: ../../doc/run_from_docker.rst

Tutorial steps
----

1. To check that *RRBot* descriptions are working properly use following launch commands

   .. code-block:: shell

    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

   .. code-block:: shell

    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*.

3. Check if the hardware interface loaded properly, by opening another terminal and executing

.. code-block:: shell

```
ros2 control list_hardware_interfaces
```

.. code-block:: shell

```
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

.. code-block:: shell

```
ros2 control list_hardware_components
```

There are two hardware components, one for the robot and one for the sensor:

.. code-block:: shell

```
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

   .. code-block:: shell

    ros2 control list_controllers

   .. code-block:: shell

    forward_position_controller[forward_command_controller/ForwardCommandC
ontroller] active
    fts_broadcaster[force_torque_sensor_broadcaster/ForceTorqueSensorBroad
caster] 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:

   #. Manually using ROS 2 CLI interface.

      .. code-block:: shell

        ros2 topic pub /forward_position_controller/commands
std_msgs/msg/Float64MultiArray "data:
        -  0.5
        -  0.5"

   #. Or you can start a demo node which sends two goals every 5 seconds
in a loop

      .. code-block:: shell

         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.

   .. code-block:: shell

    [RRBotSystemPositionOnlyHardware]: Got command 0.50000 for joint 0!
    [RRBotSystemPositionOnlyHardware]: Got command 0.50000 for joint 1!
```

6. Access wrench data from 2D FTS via

   .. code-block:: shell

    ros2 topic echo /fts_broadcaster/wrench

   shows the random generated sensor values, republished by *Force Torque Sensor Broadcaster* as
   ``geometry_msgs/msg/WrenchStamped`` message

   .. code-block:: shell

```
    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*:

   .. image:: rrbot_wrench.png
    :width: 400
    :alt: Revolute-Revolute Manipulator Robot with wrench visualization

Files used for this demos
----

* Launch file: `rrbot_system_with_external_sensor.launch.py <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_5/bringup/launch/rrbot_system_with_external_sensor.launch.py>`__
* Controllers yaml: `rrbot_with_external_sensor_controllers.yaml <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_5/bringup/config/rrbot_with_external_sensor_controllers.yaml>`__
* URDF: `rrbot_with_external_sensor_controllers.urdf.xacro <https://github.com/ros-controls/ros2_control_demos/blob/{REPOS_FILE_BRANCH}/example_5/description/urdf/rrbot_system_with_external_sensor.urdf.xacro>`__

  * Description: `rrbot_description.urdf.xacro <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_description/rrbot/urdf/rrbot_description.urdf.xacro>`__
  * ``ros2_control`` robot: `rrbot_system_position_only.ros2_control.xacro

<https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_5/description/ros2_control/rrbot_system_position_only.ros2_control.xacro>`__
  * ``ros2_control`` sensor: `external_rrbot_force_torque_sensor.ros2_control.xacro <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_5/description/ros2_control/external_rrbot_force_torque_sensor.ros2_control.xacro>`__

* RViz configuration: `rrbot.rviz <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_5/description/rviz/rrbot.rviz>`__
* Hardware interface plugin:

  * robot `rrbot.cpp <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_5/hardware/rrbot.cpp>`__
  * sensor `external_rrbot_force_torque_sensor.cpp <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_5/hardware/external_rrbot_force_torque_sensor.cpp>`__

Controllers from this demo
----
* ``Joint State Broadcaster`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/joint_state_broadcaster>`__): :ref:`doc <joint_state_broadcaster_userdoc>`
* ``Forward Command Controller`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/forward_command_controller>`__): :ref:`doc <forward_command_controller_userdoc>`
* ``Force Torque Sensor Broadcaster`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/force_torque_sensor_broadcaster>`__): :ref:`doc <force_torque_sensor_broadcaster_userdoc>`

:github_url: https://github.com/ros-controls/ros2_control_demos/blob/{REPOS_FILE_BRANCH}/example_6/doc/userdoc.rst

.. _ros2_control_demos_example_6_userdoc:

****
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``.

.. include:: ../../doc/run_from_docker.rst

Tutorial steps
----

1. To check that *RRBot* descriptions are working properly use following launch commands

   .. code-block:: shell

    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

   .. code-block:: shell

    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

.. code-block:: shell

```
ros2 control list_hardware_interfaces
```

.. code-block:: shell

```
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

.. code-block:: shell

```
ros2 control list_hardware_components
```

There are two hardware components, one for each actuator and one for each sensor:

.. code-block:: shell

```
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

   .. code-block:: shell

    ros2 control list_controllers

   .. code-block:: shell

    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:

   #. Manually using ROS 2 CLI interface.

      .. code-block:: shell

        ros2 topic pub /forward_position_controller/commands std_msgs/msg/Float64MultiArray "data:
        - 0.5
        - 0.5"

   #. Or you can start a demo node which sends two goals every 5 seconds in a loop

      .. code-block:: shell

        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.

   .. code-block:: shell

    [RRBotModularJoint]: Writing...please wait...
    [RRBotModularJoint]: Got command 0.50000 for joint 'joint1'!
    [RRBotModularJoint]: Joints successfully written!
    [RRBotModularJoint]: Writing...please wait...
    [RRBotModularJoint]: Got command 0.50000 for joint 'joint2'!
    [RRBotModularJoint]: Joints successfully written!

Files used for this demos
----

* Launch file: `rrbot_modular_actuators.launch.py
  <https://github.com/ros-

controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_6/bringup/launch/rrbot_modular_actuators.launch.py>`__
* Controllers yaml: `rrbot_modular_actuators.yaml <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_6/bringup/config/rrbot_modular_actuators.yaml>`__
* URDF: `rrbot_modular_actuators.urdf.xacro <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_6/description/urdf/rrbot_modular_actuators.urdf.xacro>`__

  * Description: `rrbot_description.urdf.xacro <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_description/rrbot/urdf/rrbot_description.urdf.xacro>`__
  * ``ros2_control`` URDF tag: `rrbot_modular_actuators.ros2_control.xacro <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_6/description/ros2_control/rrbot_modular_actuators.ros2_control.xacro>`__

* RViz configuration: `rrbot.rviz <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_description/rrbot/rviz/rrbot.rviz>`__

* Hardware interface plugin: `rrbot_actuator.cpp <https://github.com/ros-controls/ros2_control_demos/blob/{REPOS_FILE_BRANCH}/example_6/hardware/rrbot_actuator.cpp>`__

Controllers from this demo
----
* ``Joint State Broadcaster`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/joint_state_broadcaster>`__): :ref:`doc <joint_state_broadcaster_userdoc>`
* ``Forward Command Controller`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/forward_command_controller>`__): :ref:`doc <forward_command_controller_userdoc>`

:github_url: https://github.com/ros-controls/ros2_control_demos/blob/{REPOS_FILE_BRANCH}/example_7/doc/userdoc.rst

.. _ros2_control_demos_example_7_userdoc:

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

```rst
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 their offer ``command_interfaces`` and send them
to the hardware. The ``ros2_control`` node runs the main loop 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
<https://github.com/UniversalRobots/Universal_Robots_ROS2_Description>`__
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.

.. figure:: resources/robot.png
  :width: 100%
  :align: center
  :alt: r6bot

  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.

.. figure:: resources/link_1.png
```

```
   :width: 400
   :align: center
   :alt: link_1

   Link 1

.. figure:: resources/link_2_aligned.png
   :width: 400
   :align: center
   :alt: link_2_aligned

   Link 2 aligned

Each mesh should be exported to its own file after processing them.
`Blender <https://www.blender.org/>`__ 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.

.. figure:: resources/link_2_roll.png
   :width: 400
   :align: center
   :alt: link_2_roll

   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.

.. figure:: resources/link_2_roll_yaw.png
   :width: 400
   :align: center
   :alt: link_2_roll_yaw

   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.

.. figure:: resources/link_2_roll_yaw_trans.png
   :width: 400
   :align: center
   :alt: link_2_roll_yaw_trans

   Link 2 with -90 degree roll, 90 degree yaw, and translation


The described process is then repeated for all links.
```

URDF file
****

The URDF file is generally formatted according to the following template.

.. code-block:: xml

```xml
  <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" 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
<https://docs.ros.org/en/{DISTRO}/Tutorials/Intermediate/URDF/Using-Xacro-to-Clean-Up-a-URDF-File.html>`__. The complete xacro file for the robot in this tutorial is available `here <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_7/description/urdf/r6bot.urdf.xacro>`__. To verify the kinematic chain, the tool ``urdf_to_graphviz`` can be used after the URDF is generated by ``xacro``.
Running

.. code-block:: bash

```
  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 five public methods.

1. ``on_init``
2. ``export_state_interfaces``
3. ``export_command_interfaces``
4. ``read``
5. ``write``

.. code-block:: c++

```
  using CallbackReturn =
rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn;
  #include "hardware_interface/types/hardware_interface_return_values.hpp"

  class HARDWARE_INTERFACE_PUBLIC RobotSystem : public
hardware_interface::SystemInterface {
      public:
      CallbackReturn on_init(const hardware_interface::HardwareInfo &info)
override;
      std::vector<hardware_interface::StateInterface>
export_state_interfaces() override;
      std::vector<hardware_interface::CommandInterface>
export_command_interfaces() 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. Instead, vectors will be initialized that represent the state all the hardware, e.g. a vector of doubles describing joint angles, etc.

.. code-block:: c++

```
  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.

Next, ``export_state_interfaces`` and ``export_command_interfaces`` methods are called in succession. The ``export_state_interfaces`` method returns a vector of ``StateInterface``, describing the ``state_interfaces`` for each joint. The ``StateInterface`` objects are read only data handles. Their constructors require an interface name, interface type, and a pointer to a double data value. For the ``RobotSystem``, the data pointers reference class member variables. This way, the data can be access from every method.

.. code-block:: c++

```
  std::vector<hardware_interface::StateInterface>
RobotSystem::export_state_interfaces() {
      std::vector<hardware_interface::StateInterface> state_interfaces;
      // add state interfaces to ``state_interfaces`` for each joint, e.g.
`info_.joints[0].state_interfaces_`, `info_.joints[1].state_interfaces_`,
`info_.joints[2].state_interfaces_` ...
      // ...
      return state_interfaces;
    }
```

The ``export_command_interfaces`` method is nearly identical to the previous one. The difference is that a vector of ``CommandInterface`` is returned. The vector contains objects describing the ``command_interfaces`` for each joint.

.. code-block:: c++

```
  std::vector<hardware_interface::CommandInterface>
RobotSystem::export_command_interfaces() {
      std::vector<hardware_interface::CommandInterface>
command_interfaces;
      // add command interfaces to ``command_interfaces`` for each joint,
e.g. `info_.joints[0].command_interfaces_`,
`info_.joints[1].command_interfaces_`,
`info_.joints[2].command_interfaces_` ...
      // ...
      return command_interfaces;
  }
```

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.

.. code-block:: c++

```
  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.

.. code-block:: c++

```
  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.

.. code-block:: c++

```
  #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.

.. code-block:: xml

```
  <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 <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_7/ros2_control_demo_example_7.xml>`__ 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 <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_7/CMakeLists.txt>`__ for the complete ``CMakeLists.txt`` file.

.. code-block:: cmake

```
  add_library(
      robot_6_dof_hardware
      SHARED
      src/robot_hardware.cpp
  )
```

```
.. # include and link dependencies
.. # ...

.. # Causes the visibility macros to use dllexport rather than dllimport, which is appropriate when building the dll but not consuming it.
.. target_compile_definitions(robot_6_dof_hardware PRIVATE "HARDWARE_PLUGIN_DLL")
.. # export plugin
.. pluginlib_export_plugin_description_file(robot_6_dof_hardware hardware_plugin_plugin_description.xml)
.. # install libraries
.. # ...
```

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 nine public methods. The last six are `managed node <https://design.ros2.org/articles/node_lifecycle.html>`__ transitions callbacks.

1. ``command_interface_configuration``
2. ``state_interface_configuration``
3. ``update``
4. ``on_configure``
5. ``on_activate``
6. ``on_deactivate``
7. ``on_cleanup``
8. ``on_error``
9. ``on_shutdown``

```
.. code-block:: c++

  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;
      controller_interface::CallbackReturn on_cleanup(const
rclcpp_lifecycle::State &previous_state) override;
      controller_interface::CallbackReturn on_error(const
rclcpp_lifecycle::State &previous_state) override;
```

```
      controller_interface::CallbackReturn on_shutdown(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.

.. code-block:: c++

```
  using CallbackReturn =
rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn;

  controller_interface::CallbackReturn on_init(){
      // 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.

.. code-block:: c++

```
  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.

.. code-block:: c++

```
  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.

.. code-block:: c++

```
  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.

.. code-block:: c++

```
  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.

.. code-block:: c++

```
  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. It is important to release the claimed command interface in this method, so other controllers can use them if needed. This is down with the ``release_interfaces`` function.

.. code-block:: c++

```
  controller_interface::CallbackReturn on_deactivate(const
rclcpp_lifecycle::State &previous_state){
      release_interfaces();
      // The controller should be properly shutdown during this
      // ...
      return CallbackReturn::SUCCESS;
  }
```

The ``on_cleanup`` and ``on_shutdown`` are called when the controller's lifecycle node is transitioning to shutting down. Freeing any allocated memory and general cleanup should be done in these methods.

.. code-block:: c++

```
  controller_interface::CallbackReturn on_cleanup(const
rclcpp_lifecycle::State &previous_state){
    // Callback function for cleanup transition
    // ...
    return CallbackReturn::SUCCESS;
  }
```

.. code-block:: c++

```
  controller_interface::CallbackReturn on_shutdown(const
rclcpp_lifecycle::State &previous_state){
    // Callback function for shutdown transition
    // ...
    return CallbackReturn::SUCCESS;
  }
```

The ``on_error`` method is called if the managed node fails a state transition. This should generally never happen.

.. code-block:: c++

```
  controller_interface::CallbackReturn on_error(const
rclcpp_lifecycle::State &previous_state){
    // Callback function for erroneous transition
    // ...
```

```
    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 <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_7/ros2_control_demo_example_7.xml>`__ for the complete XML file.

```
.. code-block:: xml

  <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 <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_7/CMakeLists.txt>`__ for the complete ``CMakeLists.txt`` file.

```
.. code-block:: cmake

  add_library(
      r6bot_controller
      SHARED
      src/robot_controller.cpp
  )

.. # include and link dependencies
.. # ...

.. # Causes the visibility macros to use dllexport rather than dllimport, which is appropriate when building the dll but not consuming it.
.. target_compile_definitions(r6bot_controller PRIVATE "CONTROLLER_PLUGIN_DLL")
.. # export plugin
```

.. pluginlib_export_plugin_description_file(r6bot_controller robot_6_dof_controller_plugin_description.xml)
.. # install libraries
.. # ...

Launching the example

The full tutorial example can be run by first building the workspace.

.. code-block:: shell

```
  git clone -b {REPOS_FILE_BRANCH} 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.

.. code-block:: shell

```
  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.

.. code-block:: shell

```
  ros2 launch ros2_control_demo_example_7 r6bot_controller.launch.py
```

Finally, open a new terminal and run the following command.

.. code-block:: shell

```
  ros2 launch ros2_control_demo_example_7 send_trajectory.launch.py
```

You should see the tutorial robot making a circular motion in RViz.

.. figure:: resources/trajectory.gif
  :align: center
  :width: 100%
  :alt: trajectory

  Trajectory following example.

:github_url: https://github.com/ros-controls/ros2_control_demos/blob/{REPOS_FILE_BRANCH}/example_8/doc/userdoc.rst

.. _ros2_control_demos_example_8_userdoc:

****
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.

.. include:: ../../doc/run_from_docker.rst

Tutorial steps
----

1. To check that *RRBot* descriptions are working properly use following launch commands

   .. code-block:: shell

    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

   .. code-block:: shell

    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

.. code-block:: shell

```
ros2 control list_hardware_interfaces
```

.. code-block:: shell

```
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

.. code-block:: shell

```
ros2 control list_controllers
```

.. code-block:: shell

```
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:

.. code-block:: shell

```
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

.. code-block:: shell

```
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.

.. code-block:: shell

```
[RRBotTransmissionsSystemPositionOnlyHardware]: Command data:
  joint1: 0.5 --> transmission1(R=2) --> actuator1: 1
   joint2: 0.5 --> transmission2(R=4) --> actuator2: 2
[RRBotTransmissionsSystemPositionOnlyHardware]: State data:
   joint1: 0.383253 <-- transmission1(R=2) <-- actuator1: 0.766505
   joint2: 0.383253 <-- transmission2(R=4) <-- actuator2: 1.53301
```

Files used for this demos
----

* Launch file: `rrbot_transmissions_system_position_only.launch.py <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_8/bringup/launch/rrbot_transmissions_system_position_only.launch.py>`__
* Controllers yaml: `rrbot_controllers.yaml <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_8/bringup/config/rrbot_controllers.yaml>`__
* URDF file: `rrbot_transmissions_system_position_only.urdf.xacro <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_8/description/urdf/rrbot_transmissions_system_position_only.urdf.xacro>`__

  * Description: `rrbot_description.urdf.xacro <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_description/rrbot/urdf/rrbot_description.urdf.xacro>`__
  * ``ros2_control`` tag: `rrbot_transmissions_system_position_only.ros2_control.xacro <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_8/description/ros2_control/rrbot_transmissions_system_position_only.ros2_control.xacro>`__

* RViz configuration: `rrbot.rviz <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_description/rrbot/rviz/rrbot.rviz>`__

* Hardware interface plugin: `rrbot_transmissions_system_position_only.cpp <https://github.com/ros-

controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_8/hardware
/rrbot_transmissions_system_position_only.cpp>`__

Controllers from this demo
----
* ``Joint State Broadcaster`` (`ros2_controllers repository
<https://github.com/ros-
controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/joint_state_broadcas
ter>`__): :ref:`doc <joint_state_broadcaster_userdoc>`
* ``Forward Command Controller`` (`ros2_controllers repository
<https://github.com/ros-
controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/forward_command_cont
roller>`__): :ref:`doc <forward_command_controller_userdoc>`

:github_url: https://github.com/ros-controls/ros2_control_demos/blob/{REPOS_FILE_BRANCH}/example_9/doc/userdoc.rst

.. _ros2_control_demos_example_9_userdoc:

Example 9: Simulation with RRBot
=================================

With *example_9*, we demonstrate the interaction of simulators with ros2_control. More specifically,
Gazebo Classic is used for this purpose.

.. note::

  Follow the installation instructions on :ref:`ros2_control_demos_install` how to install all dependencies,
  Gazebo Classic should be automatically installed.

  * If you have installed and compiled this repository locally, you can directly use the commands below.
  * If you have installed it via the provided docker image: To run the first two steps of this example (without Gazebo Classic), use the commands as described with :ref:`ros2_control_demos_install`. To run the later steps using Gazebo Classic, execute

    .. code::

      docker run -it --rm --name ros2_control_demos --net host ros2_control_demos ros2 launch ros2_control_demo_example_9 rrbot_gazebo_classic.launch.py gui:=false

    first. Then on your local machine you can run the Gazebo Classic client with

    .. code-block:: shell

      gzclient

    and/or ``rviz2`` with

    .. code-block:: shell

      rviz2 -d src/ros2_control_demos/example_9/description/rviz/rrbot.rviz

  For details on the ``gazebo_ros2_control`` plugin, see :ref:`gazebo_ros2_control`.

Tutorial steps
----

1. To check that *RRBot* descriptions are working properly use following launch commands

   .. code-block:: shell

    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

   .. code-block:: shell

    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 and Gazebo Classic installation first, i.e., by

  .. code-block:: shell

    source /usr/share/gazebo/setup.sh

  Then, execute the launch file with

  .. code-block:: shell

    ros2 launch ros2_control_demo_example_9 rrbot_gazebo_classic.launch.py gui:=true

  The launch file loads the robot description, starts Gazebo Classic, *Joint State Broadcaster* and *Forward Command Controller*.
  If you can see two orange and one yellow "box" in Gazebo Classic everything has started properly.

  .. image:: rrbot_gazebo_classic.png
    :width: 400
    :alt: Revolute-Revolute Manipulator Robot in Gazebo Classic

4. Check if the hardware interface loaded properly, by opening another terminal and executing

   .. code-block:: shell

    ros2 control list_hardware_interfaces

   .. code-block:: shell

```
    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

   .. code-block:: shell

```
    ros2 control list_controllers
```

   .. code-block:: shell

```
    joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]
active
    forward_position_controller[forward_command_controller/ForwardCommandC
ontroller] active
```

6. If you get output from above you can send commands to *Forward Command Controller*, either:

   a. Manually using ROS 2 CLI interface:

   .. code-block:: shell

```
    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

   .. code-block:: shell

```
    ros2 launch ros2_control_demo_example_9
test_forward_position_controller.launch.py
```

   You should now see the robot moving in Gazebo Classic.

   If you echo the ``/joint_states`` or ``/dynamic_joint_states`` topics you should see the changing values,
   namely the simulated states of the robot

   .. code-block:: shell

```
    ros2 topic echo /joint_states
    ros2 topic echo /dynamic_joint_states
```

```
Files used for this demos
-------------------------

-  Launch files:

  + Hardware: `rrbot.launch.py <https://github.com/ros-
controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_9/bringup/lau
nch/rrbot.launch.py>`__
  + Gazebo Classic: `rrbot_gazebo_classic.launch.py
<https://github.com/ros-
controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_9/bringup/lau
nch/rrbot_gazebo_classic.launch.py>`__

-  Controllers yaml: `rrbot_controllers.yaml <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_9/bringup/
   config/rrbot_controllers.yaml>`__
-  URDF file: `rrbot.urdf.xacro <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_9/descript
   ion/urdf/rrbot.urdf.xacro>`__

  + Description: `rrbot_description.urdf.xacro <https://github.com/ros-
controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_des
cription/rrbot/urdf/rrbot_description.urdf.xacro>`__
  + ``ros2_control`` tag: `rrbot.ros2_control.xacro
<https://github.com/ros-
controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_9/description
/ros2_control/rrbot.ros2_control.xacro>`__

-  RViz configuration: `rrbot.rviz <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_
   description/rrbot/rviz/rrbot.rviz>`__
-  Test nodes goals configuration:

  + `rrbot_forward_position_publisher <https://github.com/ros-
controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_9/bringup/con
fig/rrbot_forward_position_publisher.yaml>`__

-  Hardware interface plugin: `rrbot.cpp <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_9/hardware
   /rrbot.cpp>`__


Controllers from this demo
--------------------------
-  ``Joint State Broadcaster`` (`ros2_controllers repository
   <https://github.com/ros-
   controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/joint_state_broadcas
   ter>`__): :ref:`doc <joint_state_broadcaster_userdoc>`
-  ``Forward Command Controller`` (`ros2_controllers repository
   <https://github.com/ros-
   controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/forward_command_cont
   roller>`__): :ref:`doc <forward_command_controller_userdoc>`
```

:github_url: https://github.com/ros-controls/ros2_control_demos/blob/{REPOS_FILE_BRANCH}/example_10/doc/userdoc.rst

.. _ros2_control_demos_example_10_userdoc:

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

   .. code-block:: shell

   ```
   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

   .. code-block:: shell

   ```
   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

   .. code-block:: console

   ```
   ros2 control list_hardware_interfaces
   ```

   .. code-block::

   ```
   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

   .. code-block:: shell

    ros2 control list_controllers

   .. code-block:: shell

    joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster] active
    gpio_controller   [ros2_control_demo_example_10/GPIOController] active
    forward_position_controller[forward_command_controller/ForwardCommandController] active

5. If you get output from above you can subscribe to the ``/gpio_controller/inputs`` topic published by the *GPIO Controller* using ROS 2 CLI interface:

   .. code-block:: shell

    ros2 topic echo /gpio_controller/inputs

   .. code-block:: shell

    interface_names:
    - flange_analog_IOs/analog_output1
    - flange_analog_IOs/analog_input1
    - flange_analog_IOs/analog_input2
    - flange_vacuum/vacuum
    values:
    - 0.0
    - 1199574016.0
    - 1676318848.0
    - 0.0

6. Now you can send commands to the *GPIO Controller* using ROS 2 CLI interface:

   .. code-block:: shell

    ros2 topic pub /gpio_controller/commands std_msgs/msg/Float64MultiArray "{data: [0.5,0.7]}"

   You should see a change in the ``/gpio_controller/inputs`` topic and a different output in the terminal where launch file is started, e.g.

   .. code-block:: shell

    [RRBotSystemWithGPIOHardware]: Got command 0.5 for GP output 0!
    [RRBotSystemWithGPIOHardware]: Got command 0.7 for GP output 1!

7. Let's introspect the ros2_control hardware component. Calling

```
  .. code-block:: shell

    ros2 control list_hardware_components

  should give you

  .. code-block:: shell

    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

  .. code-block:: shell

    ros2 launch ros2_control_demo_example_10 rrbot.launch.py
use_mock_hardware:=True

  Calling ``list_hardware_components`` with the ``-v`` option

  .. code-block:: shell

    ros2 control list_hardware_components -v

  now should give you

  .. code-block:: shell

    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]
```

```
        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

.. code-block:: shell

```
ros2 topic echo /gpio_controller/inputs
```

again and you should see that - unless commands are received - the values of the state interfaces are now ``nan`` except for the vacuum interface.

.. code-block:: shell

```
interface_names:
-  flange_analog_IOs/analog_output1
-  flange_analog_IOs/analog_input1
-  flange_analog_IOs/analog_input2
-  flange_vacuum/vacuum
values:
-  .nan
-  .nan
-  .nan
-  1.0
```

This is, because for the vacuum interface an initial value of ``1.0`` is set in the URDF file.

.. code-block:: xml

```
  <gpio name="flange_vacuum">
    <command_interface name="vacuum"/>
    <state_interface name="vacuum">
      <param name="initial_value">1.0</param>
    </state_interface>
  </gpio>
```

Call again

.. code-block:: shell

```
ros2 topic pub /gpio_controller/commands
std_msgs/msg/Float64MultiArray "{data: [0.5,0.7]}"
```

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 <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_10/bringup
   /launch/rrbot.launch.py>`__
-  Controllers yaml: `rrbot_controllers.yaml <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_10/bringup
   /config/rrbot_controllers.yaml>`__
-  URDF file: `rrbot.urdf.xacro <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_10/descrip
   tion/urdf/rrbot.urdf.xacro>`__

  + Description: `rrbot_description.urdf.xacro <https://github.com/ros-
controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_des
cription/rrbot/urdf/rrbot_description.urdf.xacro>`__
  + ``ros2_control`` tag: `rrbot.ros2_control.xacro
<https://github.com/ros-
controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_10/descriptio
n/ros2_control/rrbot.ros2_control.xacro>`__

-  RViz configuration: `rrbot.rviz <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_
   description/rrbot/rviz/rrbot.rviz>`__

-  Hardware interface plugin:

  + `rrbot.cpp <https://github.com/ros-
controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_10/hardware/r
rbot.cpp>`__
  + `generic_system.cpp <https://github.com/ros-
controls/ros2_control/tree/{REPOS_FILE_BRANCH}/hardware_interface/src/mock
_components/generic_system.cpp>`__

-  GPIO controller: `gpio_controller.cpp <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_10/control
   lers/gpio_controller.cpp>`__


Controllers from this demo
--------------------------
-  ``Joint State Broadcaster`` (`ros2_controllers repository
   <https://github.com/ros-
   controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/joint_state_broadcas
   ter>`__): :ref:`doc <joint_state_broadcaster_userdoc>`
-  ``Forward Command Controller`` (`ros2_controllers repository
   <https://github.com/ros-
   controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/forward_command_cont
   roller>`__): :ref:`doc <forward_command_controller_userdoc>`
```

:github_url: https://github.com/ros-controls/ros2_control_demos/blob/{REPOS_FILE_BRANCH}/example_11/doc/userdoc.rst

.. _ros2_control_demos_example_11_userdoc:

****
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.

.. include:: ../../doc/run_from_docker.rst

Tutorial steps
----

1. To check that *CarlikeBot* description is working properly use following launch commands

   .. code-block:: shell

    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.

   .. image:: carlikebot.png
    :width: 400

    :alt: Carlike Mobile Robot

2. To start *CarlikeBot* example open a terminal, source your ROS2-
    workspace and execute its launch file with

   .. code-block:: shell

    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

   .. code-block:: shell

    ros2 control list_hardware_interfaces

   You should get

   .. code-block:: shell

      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

   .. code-block:: shell

```
ros2 control list_controllers
```

   You should get

   .. code-block:: shell

```
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:

   .. code-block:: shell

```
ros2 topic pub --rate 30 /bicycle_steering_controller/reference geometry_msgs/msg/TwistStamped "
  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.

   .. code-block:: shell

```
[CarlikeBotSystemHardware]: Got position command: 0.03 for joint 'virtual_front_wheel_joint'.
[CarlikeBotSystemHardware]: Got velocity command: 20.01 for joint 'virtual_rear_wheel_joint'.
```

Files used for this demos

```
*  Launch file: `carlikebot.launch.py <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_11/bringup
   /launch/carlikebot.launch.py>`__
*  Controllers yaml: `carlikebot_controllers.yaml <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_11/bringup
   /config/carlikebot_controllers.yaml>`__
*  URDF file: `carlikebot.urdf.xacro <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_11/descrip
   tion/urdf/carlikebot.urdf.xacro>`__

  *  Description: `carlikebot_description.urdf.xacro
     <https://github.com/ros-
     controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_dem
     o_description/carlikebot/urdf/carlikebot_description.urdf.xacro>`__
  *  ``ros2_control`` tag: `carlikebot.ros2_control.xacro
     <https://github.com/ros-
     controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_11/descr
     iption/ros2_control/carlikebot.ros2_control.xacro>`__

*  RViz configuration: `carlikebot.rviz <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_
   description/carlikebot/rviz/carlikebot.rviz>`__

*  Hardware interface plugin: `carlikebot_system.cpp
   <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_11/hardwar
   e/carlikebot_system.cpp>`__


Controllers from this demo
--------------------------

*  ``Joint State Broadcaster`` (`ros2_controllers repository
   <https://github.com/ros-
   controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/joint_state_broadcas
   ter>`__): :ref:`doc <joint_state_broadcaster_userdoc>`
*  ``Bicycle Steering Controller`` (`ros2_controllers repository
   <https://github.com/ros-
   controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/bicycle_steering_con
   troller>`__): :ref:`doc <bicycle_steering_controller_userdoc>`
```

:github_url: https://github.com/ros-controls/ros2_control_demos/blob/{REPOS_FILE_BRANCH}/example_12/doc/userdoc.rst

.. _ros2_control_demos_example_12_userdoc:

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``.

.. include:: ../../doc/run_from_docker.rst

Tutorial steps
----

1. To check that *RRBot* descriptions are working properly use following
    launch commands

   .. code-block:: shell

    ros2 launch ros2_control_demo_example_12 view_robot.launch.py

   .. image:: rrbot.png
    :width: 400
    :alt: Revolute-Revolute Manipulator Robot

   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

   .. code-block:: shell

    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

   .. code-block:: shell

```
ros2 control list_controllers
```

   .. code-block:: shell

```
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

   .. code-block:: shell

```
ros2 control list_hardware_interfaces
```

   The output should be something like this:

   .. code-block:: shell

```
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

.. code-block:: shell

```
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

.. code-block:: shell

```
ros2 control list_controllers
```

.. code-block:: shell

```
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]
active
joint2_position_controller[passthrough_controller/PassthroughControlle
r] active
joint1_position_controller[passthrough_controller/PassthroughControlle
r] active
```

    position_controller [passthrough_controller/PassthroughController]
active
    forward_position_controller[forward_command_controller/ForwardCommandC
ontroller] active

7. Now check if the interfaces are loaded properly, by opening another terminal and executing

   .. code-block:: shell

    ros2 control list_hardware_interfaces

   The output should be something like this:

   .. code-block:: shell

```
    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*:

   .. code-block:: shell

```
    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.

   .. code-block:: shell

```
    [RRBotSystemPositionOnlyHardware]: Got command 0.50000 for joint 0!
    [RRBotSystemPositionOnlyHardware]: Got command 0.50000 for joint 1!
```

If you echo the ``/joint_states`` or ``/dynamic_joint_states`` topics you should now get similar values, namely the simulated states of the robot

.. code-block:: shell

```
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 <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_12/bringup/launch/rrbot.launch.py>`__
  + Controllers: `rrbot.launch.py <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_12/bringup/launch/launch_chained_controllers.launch.py>`__
- ROS2 Controller: `passthrough_controller.cpp <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_12/controllers/src/passthrough_controller.cpp>`__
- Controllers yaml: `rrbot_controllers.yaml <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_12/bringup/config/rrbot_chained_controllers.yaml>`__
- URDF file: `rrbot.urdf.xacro <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_12/description/urdf/rrbot.urdf.xacro>`__

  + Description: `rrbot_description.urdf.xacro <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_description/rrbot/urdf/rrbot_description.urdf.xacro>`__
  + ``ros2_control`` tag: `rrbot.ros2_control.xacro <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_12/description/ros2_control/rrbot.ros2_control.xacro>`__
- RViz configuration: `rrbot.rviz <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_description/rrbot/rviz/rrbot.rviz>`__
- Hardware interface plugin: `rrbot.cpp <https://github.com/ros-controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_12/hardware/rrbot.cpp>`__

Controllers from this demo

- ``Joint State Broadcaster`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/joint_state_broadcaster>`__): :ref:`doc <joint_state_broadcaster_userdoc>`
- ``Forward Command Controller`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/forward_command_controller>`__): :ref:`doc <forward_command_controller_userdoc>`

:github_url: https://github.com/ros-controls/ros2_control_demos/blob/{REPOS_FILE_BRANCH}/example_14/doc/userdoc.rst

.. _ros2_control_demos_example_14_userdoc:

****
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``.

.. include:: ../../doc/run_from_docker.rst

Tutorial steps
----

1. To check that *RRBot* descriptions are working properly use following launch commands

   .. code-block:: shell

    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

   .. code-block:: shell

    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

.. code-block:: shell

```
ros2 control list_hardware_interfaces
```

.. code-block:: shell

```
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

.. code-block:: shell

```
ros2 control list_hardware_components
```

There are four hardware components, one for each actuator and one for each sensor:

.. code-block:: shell

```
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
              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

   .. code-block:: shell

    ros2 control list_controllers

   .. code-block:: shell

    joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]
active
    forward_velocity_controller[forward_command_controller/ForwardCommandC
ontroller] active

5. If you get output from above you can send commands to *Forward Command Controller*:

   .. code-block:: shell

    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.

   .. code-block:: shell

    [RRBotActuatorWithoutFeedback]: Writing command: 5.000000

```
    [RRBotActuatorWithoutFeedback]: Sending data command: 5
    [RRBotActuatorWithoutFeedback]: Joints successfully written!
    [RRBotActuatorWithoutFeedback]: Writing command: 5.000000
    [RRBotActuatorWithoutFeedback]: Sending data command: 5
    [RRBotActuatorWithoutFeedback]: Joints successfully written!
    [RRBotSensorPositionFeedback]: Reading...
    [RRBotSensorPositionFeedback]: Got measured velocity 5.00000
    [RRBotSensorPositionFeedback]: Got state 0.25300 for joint 'joint1'!
    [RRBotSensorPositionFeedback]: Joints successfully read!
    [RRBotSensorPositionFeedback]: Reading...
    [RRBotSensorPositionFeedback]: Got measured velocity 5.00000
    [RRBotSensorPositionFeedback]: Got state 0.25300 for joint 'joint2'!
    [RRBotSensorPositionFeedback]: Joints successfully read!


Files used for this demos
--------------------------

*  Launch file:
   `rrbot_modular_actuators_without_feedback_sensors_for_position_feedback
   .launch.py <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_14/bringup
   /launch/rrbot_modular_actuators_without_feedback_sensors_for_position_f
   eedback.launch.py>`__
*  Controllers yaml:
   `rrbot_modular_actuators_without_feedback_sensors_for_position_feedback
   .yaml <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_14/bringup
   /config/rrbot_modular_actuators_without_feedback_sensors_for_position_f
   eedback.yaml>`__
*  URDF:
   `rrbot_modular_actuators_without_feedback_sensors_for_position_feedback
   .urdf.xacro <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_14/descrip
   tion/urdf/rrbot_modular_actuators_without_feedback_sensors_for_position
   _feedback.urdf.xacro>`__

  *  Description: `rrbot_description.urdf.xacro <https://github.com/ros-
     controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_dem
     o_description/rrbot/urdf/rrbot_description.urdf.xacro>`__
  *  ``ros2_control`` URDF tag:
     `rrbot_modular_actuators_without_feedback_sensors_for_position_feedba
     ck.ros2_control.xacro <https://github.com/ros-
     controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_14/descr
     iption/ros2_control/rrbot_modular_actuators_without_feedback_sensors_
     for_position_feedback.ros2_control.xacro>`__

*  RViz configuration: `rrbot.rviz <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_
   description/rrbot/rviz/rrbot.rviz>`__

*  Hardware interface plugins:
```

  * `rrbot_actuator_without_feedback.cpp <https://github.com/ros-controls/ros2_control_demos/blob/{REPOS_FILE_BRANCH}/example_14/hardware/rrbot_actuator_without_feedback.cpp>`__
  * `rrbot_sensor_for_position_feedback.cpp <https://github.com/ros-controls/ros2_control_demos/blob/{REPOS_FILE_BRANCH}/example_14/hardware/rrbot_sensor_for_position_feedback.cpp>`__

Controllers from this demo
----
* ``Joint State Broadcaster`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/joint_state_broadcaster>`__): :ref:`doc <joint_state_broadcaster_userdoc>`
* ``Forward Command Controller`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/forward_command_controller>`__): :ref:`doc <forward_command_controller_userdoc>`

:github_url: https://github.com/ros-
controls/ros2_control_demos/blob/{REPOS_FILE_BRANCH}/example_15/doc/userdo
c.rst

.. _ros2_control_demos_example_15_userdoc:

Example 15: Using multiple controller managers
==

This example shows how to integrate multiple robots under different
controller manager instances.

.. include:: ../../doc/run_from_docker.rst

Scenario: Using ros2_control within a local namespace
----

* Launch file: `rrbot_namespace.launch.py <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_15/bringup
   /launch/rrbot_namespace.launch.py>`__
* Controllers yaml: `rrbot_namespace_controllers.yaml
   <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_15/bringup
   /config/rrbot_namespace_controllers.yaml>`__
* URDF file: `rrbot.urdf.xacro <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_15/descrip
   tion/urdf/rrbot.urdf.xacro>`__

  * Description: `rrbot_description.urdf.xacro <https://github.com/ros-
     controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_dem
     o_description/rrbot/urdf/rrbot_description.urdf.xacro>`__
  * ``ros2_control`` tag: `rrbot.ros2_control.xacro
     <https://github.com/ros-
     controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_15/descr
     iption/ros2_control/rrbot.ros2_control.xacro>`__

* RViz configuration: `rrbot.rviz <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_
   description/rrbot/rviz/rrbot.rviz>`__
* Test nodes goals configuration:

  + `rrbot_forward_position_publisher <https://github.com/ros-
controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_15/bringup/co
nfig/rrbot_forward_position_publisher.yaml>`__
  + `rrbot_joint_trajectory_publisher <https://github.com/ros-
controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_15/bringup/co
nfig/rrbot_joint_trajectory_publisher.yaml>`__

* Hardware interface plugin: `rrbot.cpp <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_15/hardwar
   e/rrbot.cpp>`__

.. 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

.. code-block:: shell

```
  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")

.. code-block:: shell

```
  $ ros2 control list_controllers -c /rrbot/controller_manager
  joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]
active
  forward_position_controller[forward_command_controller/ForwardCommandCon
troller] active
  position_trajectory_controller[joint_trajectory_controller/JointTrajecto
ryController] inactive
```

Commanding the robot using a ``ForwardCommandController`` (name: ``/rrbot/forward_position_controller``)

.. code-block:: shell

```
  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``):

.. code-block:: shell

```
  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``)

.. code-block:: shell

  ros2 launch ros2_control_demo_example_15
test_joint_trajectory_controller.launch.py
publisher_config:=rrbot_namespace_joint_trajectory_publisher.yaml

Scenario: Using multiple controller managers on the same machine
----

* Launch file: `multi_controller_manager_example_two_rrbots.launch.py
   <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_15/bringup
   /launch/multi_controller_manager_example_two_rrbots.launch.py>`__
* Controllers yaml:
  - `multi_controller_manager_rrbot_1_controllers.yaml
     <https://github.com/ros-
     controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_15/bring
     up/config/multi_controller_manager_rrbot_1_controllers.yaml>`__
  - `multi_controller_manager_rrbot_2_controllers.yaml
     <https://github.com/ros-
     controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_15/bring
     up/config/multi_controller_manager_rrbot_2_controllers.yaml>`__
* URDF file: `rrbot.urdf.xacro <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_15/descrip
   tion/urdf/rrbot.urdf.xacro>`__

  * Description: `rrbot_description.urdf.xacro <https://github.com/ros-
     controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_dem
     o_description/rrbot/urdf/rrbot_description.urdf.xacro>`__
  * ``ros2_control`` tag: `rrbot.ros2_control.xacro
     <https://github.com/ros-
     controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_15/descr
     iption/ros2_control/rrbot.ros2_control.xacro>`__

* RViz configuration: `rrbot.rviz <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/ros2_control_demo_
   description/rrbot/rviz/rrbot.rviz>`__
* Test nodes goals configuration:

  + `rrbot_forward_position_publisher <https://github.com/ros-
controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_15/bringup/co
nfig/rrbot_forward_position_publisher.yaml>`__
  + `rrbot_joint_trajectory_publisher <https://github.com/ros-
controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_15/bringup/co
nfig/rrbot_joint_trajectory_publisher.yaml>`__

* Hardware interface plugin: `rrbot.cpp <https://github.com/ros-
   controls/ros2_control_demos/tree/{REPOS_FILE_BRANCH}/example_15/hardwar
   e/rrbot.cpp>`__

.. 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

.. code-block:: shell

```
ros2 launch ros2_control_demo_example_15
multi_controller_manager_example_two_rrbots.launch.py
```

You should see two robots in RViz:

.. image:: two_rrbot.png
   :width: 400
   :alt: Two Revolute-Revolute Manipulator Robot

``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``):

.. code-block:: shell

```
$ ros2 control list_controllers -c /rrbot_1/controller_manager
position_trajectory_controller[joint_trajectory_controller/JointTrajecto
ryController] inactive
joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]
active
forward_position_controller[forward_command_controller/ForwardCommandCon
troller] active
```

```
  $ ros2 control list_controllers -c /rrbot_2/controller_manager
  joint_state_broadcaster[joint_state_broadcaster/JointStateBroadcaster]
active
  position_trajectory_controller[joint_trajectory_controller/JointTrajecto
ryController] inactive
  forward_position_controller[forward_command_controller/ForwardCommandCon
troller] active
```

Commanding the robots using the ``forward_position_controller`` (of type ``ForwardCommandController``)

.. code-block:: shell

```
  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``:

.. code-block:: shell

```
  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``:

.. code-block:: shell

```
  ros2 launch ros2_control_demo_example_15
test_multi_controller_manager_joint_trajectory_controller.launch.py
```

Controllers from this demo
----

* ``Joint State Broadcaster`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/joint_state_broadcaster>`__): `doc <https://control.ros.org/{REPOS_FILE_BRANCH}/doc/ros2_controllers/joint_state_broadcaster/doc/userdoc.html>`__
* ``Forward Command Controller`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/forward_command_controller>`__): `doc <https://control.ros.org/{REPOS_FILE_BRANCH}/doc/ros2_controllers/forward_command_controller/doc/userdoc.html>`__
* ``Joint Trajectory Controller`` (`ros2_controllers repository <https://github.com/ros-controls/ros2_controllers/tree/{REPOS_FILE_BRANCH}/joint_trajectory_c

ontroller>`__): `doc
<https://control.ros.org/{REPOS_FILE_BRANCH}/doc/ros2_controllers/joint_trajectory_controller/doc/userdoc.html>`__

```
rviz_node = Node(
        package="rviz2",
        executable="rviz2",
        name="rviz2",
        output="log",
        arguments=["-d", rviz_config_file],
        condition=IfCondition(gui),
    )


    // open the rviz

    joint_state_broadcaster_spawner = Node(
        package="controller_manager",
        executable="spawner",
        arguments=["joint_state_broadcaster", "--controller-manager",
"/controller_manager"],

    )
// Publish the joint_states


delay is required

 # Delay rviz start after `joint_state_broadcaster`
    delay_rviz_after_joint_state_broadcaster_spawner =
RegisterEventHandler(
        event_handler=OnProcessExit(
            target_action=joint_state_broadcaster_spawner,
            on_exit=[rviz_node],
        )
    )

    # Delay start of joint_state_broadcaster after `robot_controller`
    # TODO(anyone): This is a workaround for flaky tests. Remove when
fixed.
    delay_joint_state_broadcaster_after_robot_controller_spawner =
RegisterEventHandler(
        event_handler=OnProcessExit(
            target_action=robot_controller_spawner,
            on_exit=[joint_state_broadcaster_spawner],
        )
    )
```