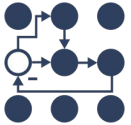


ros2_control - a robot-agnostic control framework for ROS2

WR Meetup #13 – 7 Oct. 2021

Outline

- Why do we need a control framework in ROS/ROS2?
- History & basic concepts
- How to use ros(1)_control?
- What is a bit tricky in ros(1)_control?
- Architecture of ros2_control
- “Everything is an interface 😀”
- URDF extension with <ros2_control>-tag
- Examples from users

Buckle up, this will be an adventure :)

$whoarewe

Bence Magyar

- PhD in Robotics
- Lead Software Engineer at FiveAI
- ros_control and ros2_control maintainer

Denis Štogl

- Control Engineer and Roboticist (PhD soon official)
- Robotics Consultant at Stogl Robotics Consulting
- ros2_control maintainer

Why do we need a control framework in ROS?

1. “Control” is tricky — everybody needs it (to run a physical robot)
2. “Stop reinventing the wheel when controlling hardware”

- Provide standardized interfaces for “high-level”/task control “nodes”
  - MoveIt, Navigation, <your_sexy_cool_application>, …
- Establish standard set of controllers:
  - “Implement control-method once and use it on various hardware”
- To implement access management each time is annoying
- Optimize controllers and management-functions for real-time performance
- Standardization of hardware-abstraction layer

What & where

pr2_controller_manager

(pr2_mechanism)

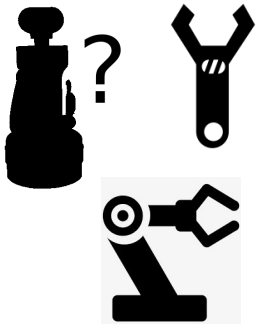
2009

ros_control

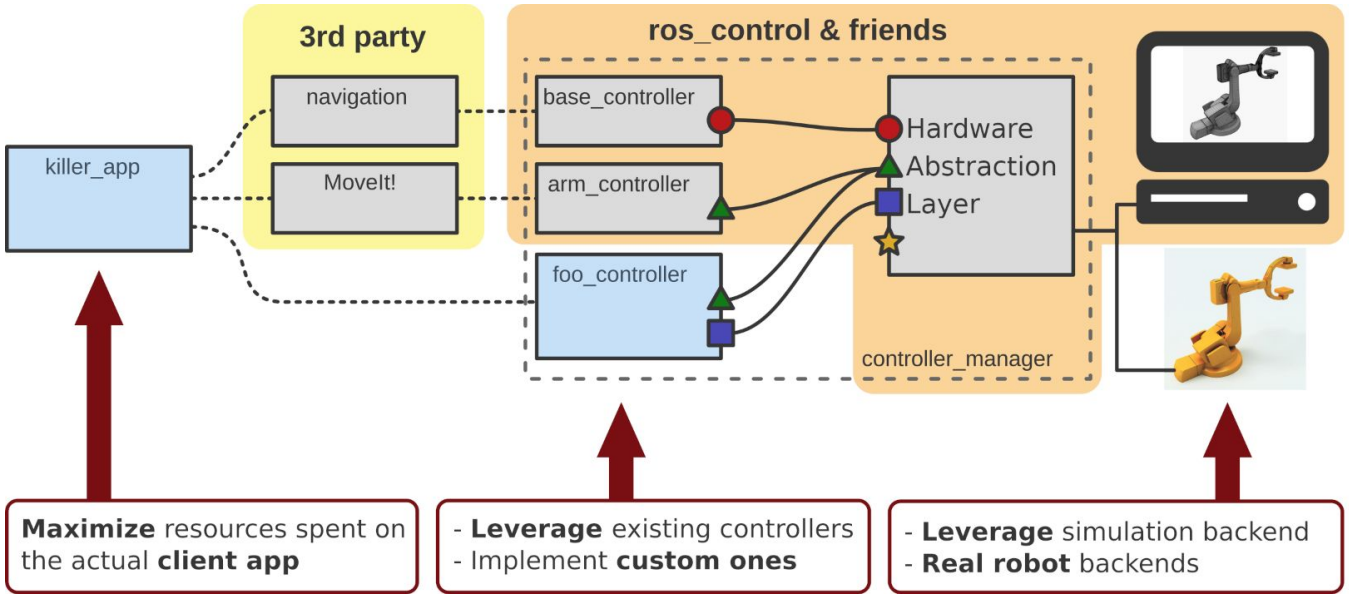
2012/2013

ros2_control

2017/2021



Basic concepts



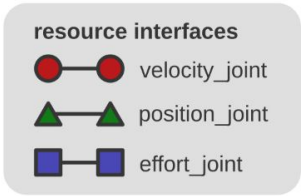
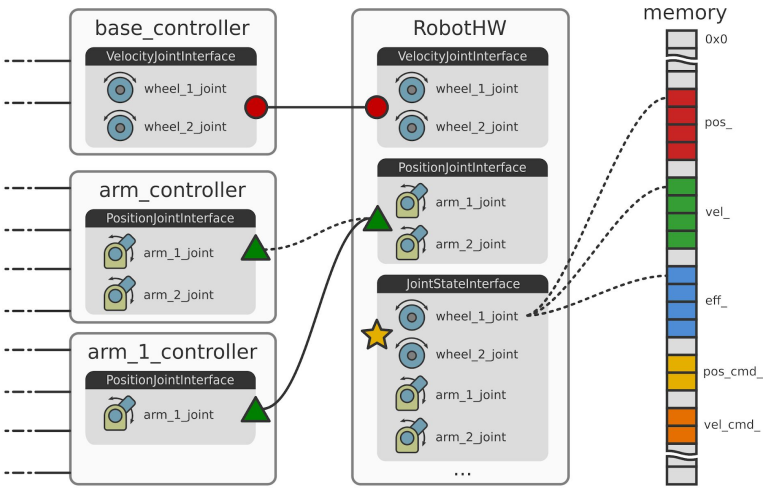
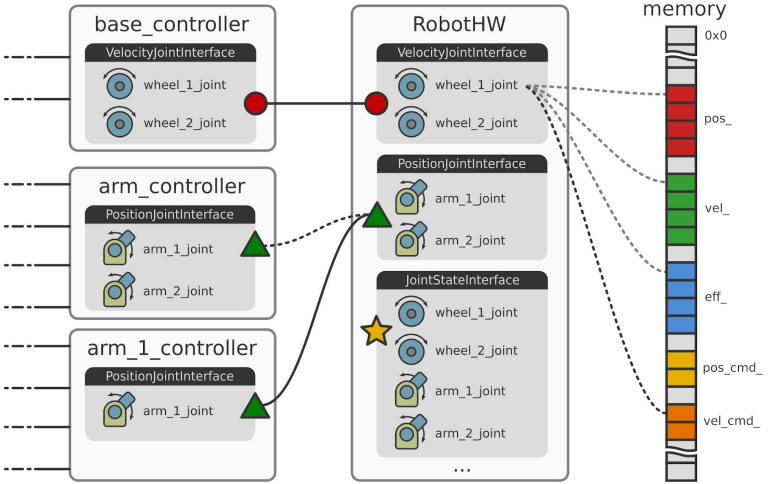
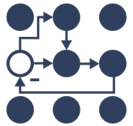


image credits:
visualpharm.com
turbosquid.com

Basic Concepts







How to use ros(1)_control?

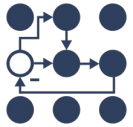
1. Implement RobotHW (hardware abstraction layer)
    a. init(), read(), write()
    b. JointStateInterface and [Position|Velocity|Effort]JointInterface

2. Implement a control node
    a. Load URDF (“/robot_description” parameter) - usually needed
    b. Initialize RobotHW
    c. Initialize ControllerManager
    d. Start main loop `while(ros::ok())`
        i. read()
        ii. update() — controllers
        iii. write()

Standard controllers

- joint_state_broadcaster
- diff_drive_controller
- joint_trajectory_controller
- gripper_controllers
- Forwarding controllers for groups of joints
  - position_controllers
  - velocity_controllers
  - effort_controllers

Image from ros_control [paper]



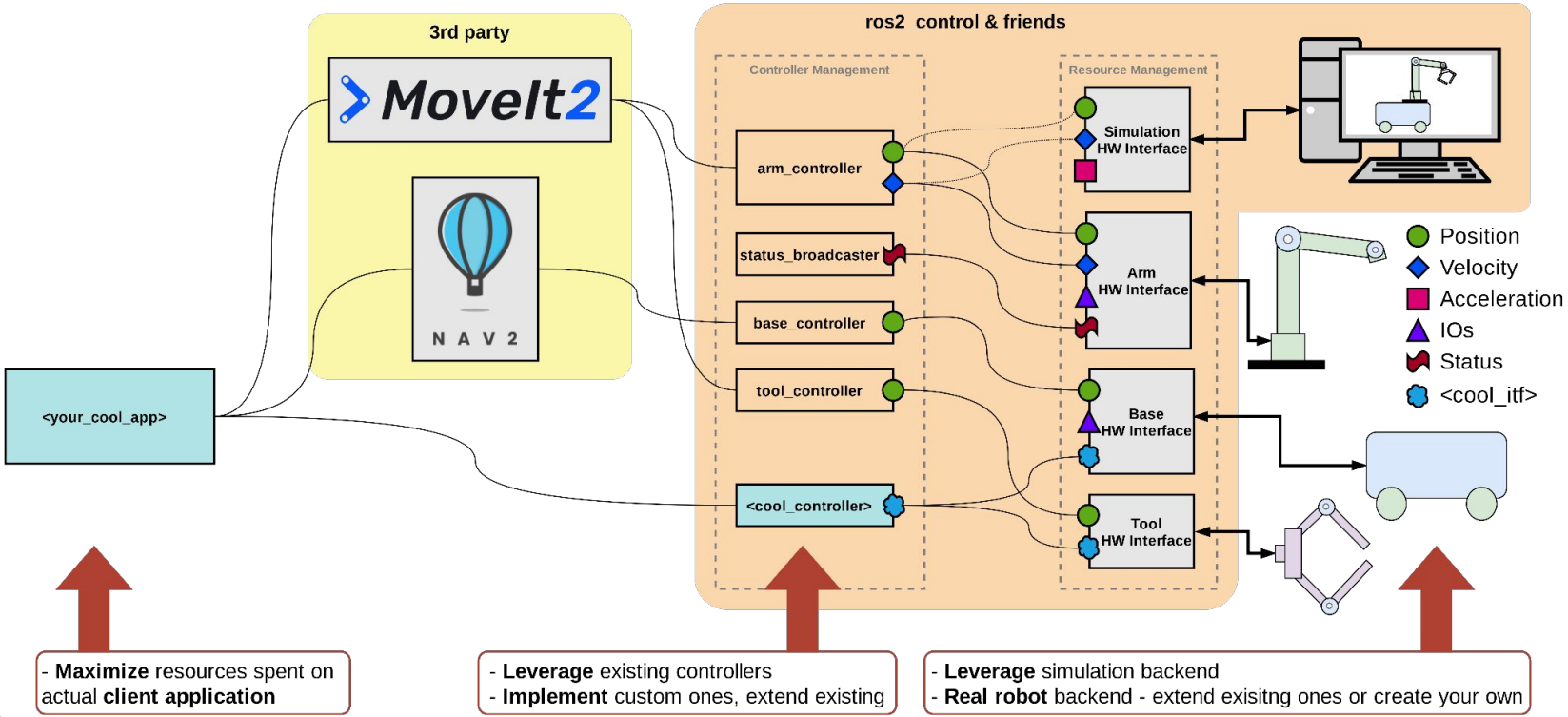
What is a bit tricky in ros(1)_control?

- Joint interfaces limited to: “position”, “velocity”, and “effort
- Complex code-base — lots of templating and inheritance
- “Control node” has to be implemented for each hardware
- Unclear semantic — everything is a *RobotHW*
- Hardware composition possible, but not straightforward

Improvements in ros2_control

- Joint interfaces limited to: “position”, “velocity”, and “effort
  - Interface types are “strings” — fully flexible
- Complex code-base — lots of templating and inheritance
  - Cleaner code-base, modern C++
- “Control node” has to be implemented for each hardware
  - Default “ros2_control_node”, no need for boilerplates
- Unclear semantic — everything is a *RobotHW*
  - *Actuator*, *Sensor*, and *System* hardware types
- Hardware composition possible, but not straightforward
  - Hardware interfaces are always plugins with lifecycle
  - A robot-cell can now be created via plug-and-play

Architecture of ros2_control



“Everything is an interface 😀”

- ∞ number of interface types → control_msgs/DynamicJointState
  - Type-names are freely choosable
  - Standard names: “position”, “velocity”, “acceleration”, and “effort”

- This enables:
  - Semantic clarity of interface-types (use as clear type-name as possible)
  - Out-of-the-box support for digital and analog inputs and outputs
  - Use of multiple sensors for the same “value”
  - “Interface-type” == string → no need for templating and complex inheritance

URDF extension with <ros2_control>-tag

- Defines hardware *type* and *name*
- <hardware>
  - *plugin* and its *parameters*
- <joint>
  - Describes 1 DoF
  - Name, interfaces, and parameters
- <sensor>
  - Sensing component that is not related to a joint
  - name, state interfaces, and parameters
- <gpio>
  - Everything else
  - *Size*, *data type*, name, interfaces, and parameters

All interfaces has internally type double!

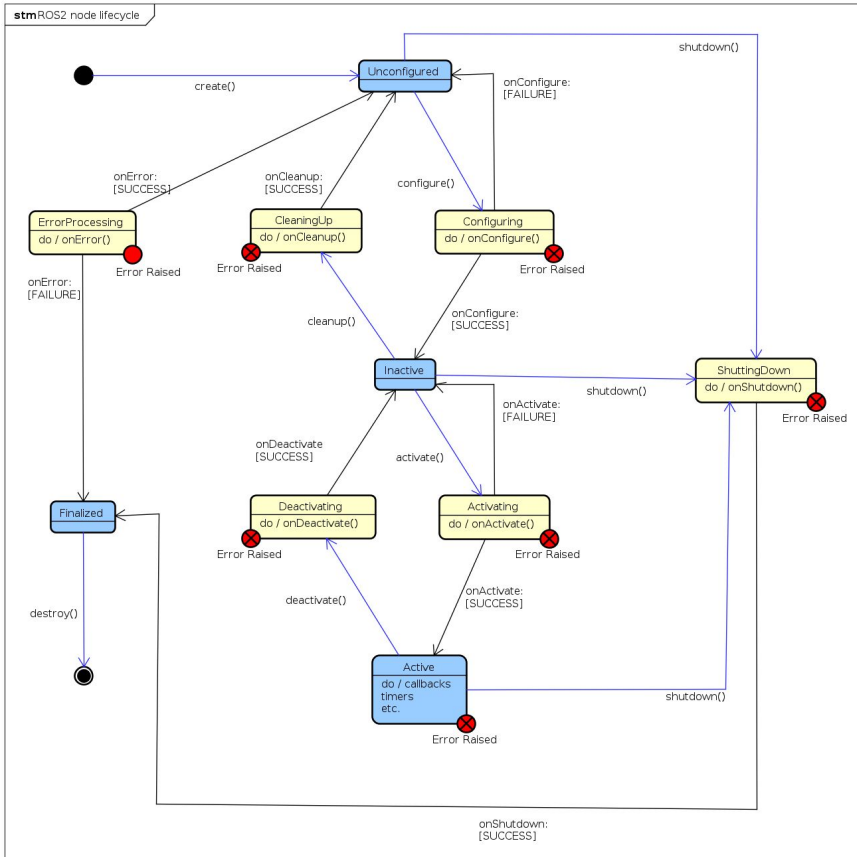
URDF extension with <ros2_control>-tag

```
<ros2_control name="robot" type="system">

  <hardware>
    <plugin>robot_package/Robot</plugin>
    <param name="hardware_parameter">some_value</param>
  </hardware>

  <joint name="joint_first">
    <command_interface name="position"/>
    <state_interface name="acceleration"/>
  </joint>

  . . .

  <gpio name="rrbot_status">
    <state_interface name="mode" data_type="int"/>
    <state_interface name="bit" data_type="bool" size="4"/>
  </gpio>

</ros2_control>


<ros2_control name="tool" type="actuator">

  <hardware>
    <plugin>tool_package/Tool</plugin>
    <param name="hardware_parameter">some_value</param>
  </hardware>

  <joint name="tool">
    <command_interface name="command"/>
  </joint>

</ros2_control>
```

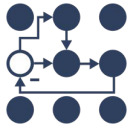
```
<ros2_control name="robot" type="system">

  <hardware>
      <plugin>robot_package/Robot</plugin>
      <param name="hardware_parameter">some_value</param>
  </hardware>

  <joint name="joint_first">
    <command_interface name="position"/>
    <state_interface name="acceleration"/>
  </joint>
  . . .
  <joint name="joint_last">
    <command_interface name="velocity">
      <param name="min">-1</param>
      <param name="max">1</param>
    </command_interface>
    <state_interface name="temperature"/>
  </joint>

  <sensor name="tcp_sensor">
    <state_interface name="sensing_inteface"/>
    <param name="sensor_parameter">another_value</param>
  </sensor>

  <gpio name="flange_IOs">
    <command_interface name="digital_output" data_type="bool" size="8" />
    <state_interface name="digital_output" data_type="bool" size="8" />
    <command_interface name="analog_output" data_type="double" size="2" />
    <state_interface name="analog_output" data_type="double" size="2" />

    <state_interface name="digital_input" data_type="bool" size="4" />
    <state_interface name="analog_input" data_type="double" size="4" />
  </gpio>

  <gpio name="rrbot_status">
    <state_interface name="mode" data_type="int"/>
    <state_interface name="bit" data_type="bool" size="4"/>
  </gpio>

  <joint name="tool">
    <command_interface name="command"/>
  </joint>

</ros2_control>
```

Lifecycle for controllers and hardware

- Managed nodes - interface



https://design.ros2.org/articles/node_lifecycle.html

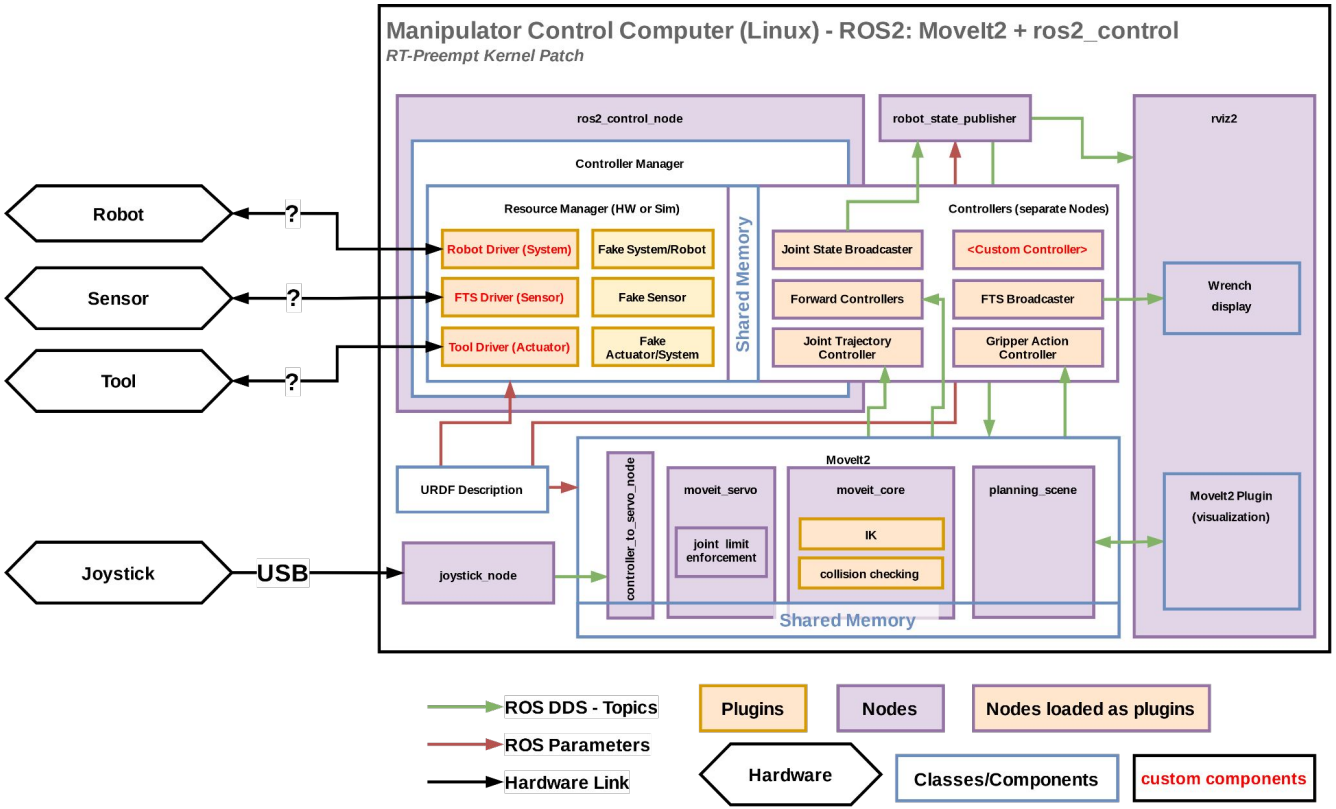


How to use ros2_control

- Write <ros2_control> tag for your robot
  - Tip: write as *xacro* macro
- Implement hardware interface
  - Actuator - for 1 DoF actuators, e.g., motors
  - Sensor - for sensors
  - System - for multi DoF actuators, e.g., robots 😜
- Configure controllers / controller manager
  - Do not forget to configure used interfaces

Have fun!

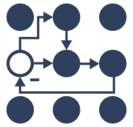
Use-cases from *wilderness* - ros2_control + MoveIt2



Use-cases from *wilderness*

- UR driver: https://github.com/UniversalRobots/Universal_Robots_ROS2_Driver
  - The first open-source driver with ros2_control integration
  - Needs special features:
    - Digital and analog inputs and outputs
    - General robot operation: unlock protective stop, restart safety, break release (TBD)
    - Loading, starting, and stopping programs (TBD)

- Dynamixel: https://github.com/youtalk/dynamixel_control
  - Uses multiple servos
  - Reference implementation for ROBOTIS OpenManipulator-X

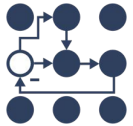


Use-cases from *wilderness*

- ros2_control_demos: https://github.com/ros-controls/ros2_control_demos
  - Tool changing — hardware-lifecycle example (PR #133)
  - “Stacking HW together” — RRBot + FTS Sensor (example 4)

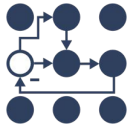
- “My robot has measurement offset”
  - Separate commanded and measured states in visualization (example coming soon)

- Hardware “architectures” and capabilities:
  - https://github.com/ros-controls/roadmap/blob/master/design_drafts/components_architecture_and_urdf_examples.md



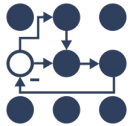
References

- ros_control paper in the Journal of Open Source Software
- ros2_control resources
  - https://control.ros.org
  - https://github.com/ros-controls/ros2_control
  - https://github.com/ros-controls/ros2_controllers
  - https://github.com/ros-controls/ros2_control_demos
  - https://github.com/ros-controls/roadmap/blob/master/documentation_resources.md
- Videos/presentations:
  - https://youtu.be/G__yFTWp_M0
  - https://www.youtube.com/watch?v=5OfOPcu8Erw&t=245s
  -



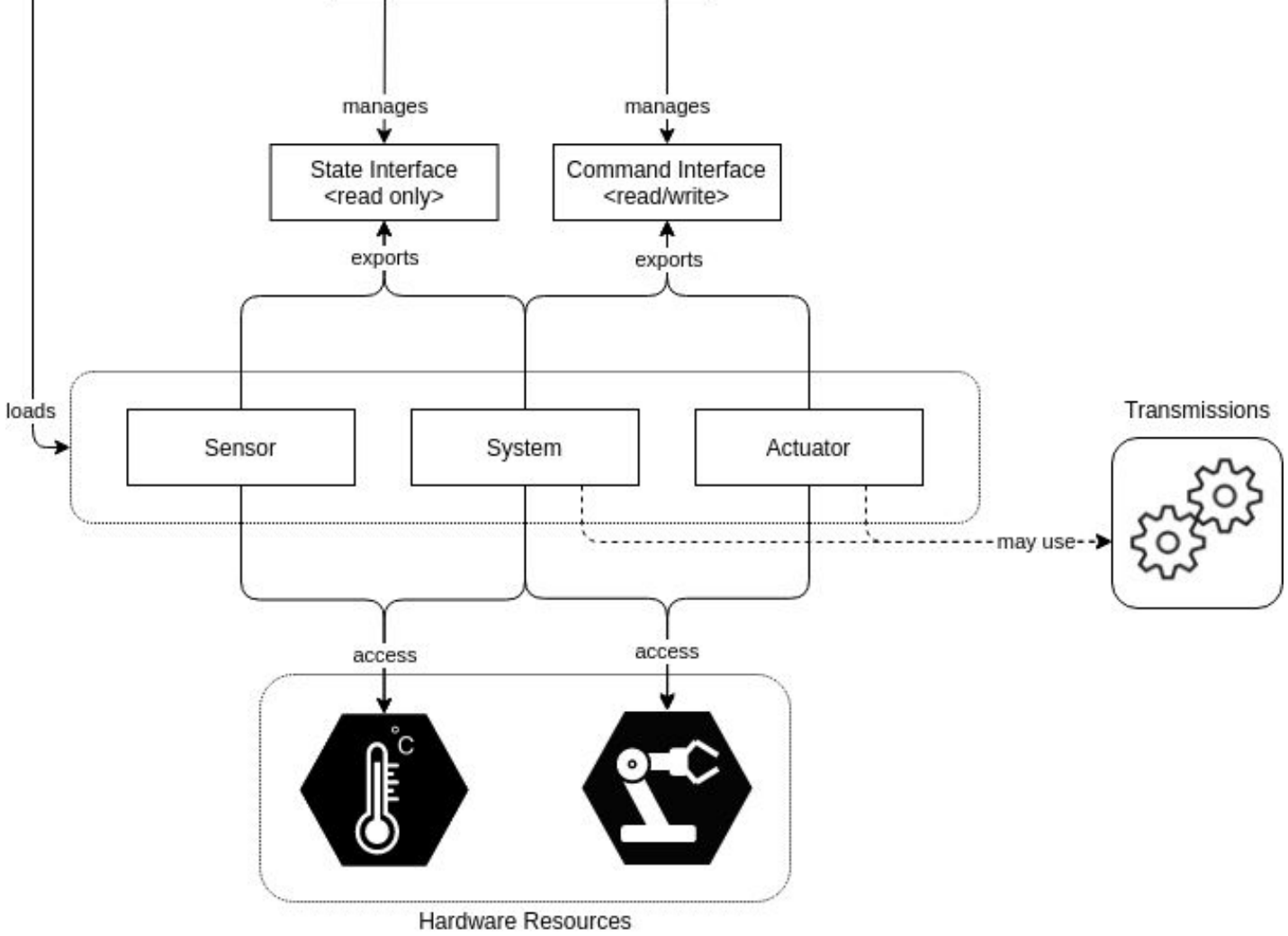
References

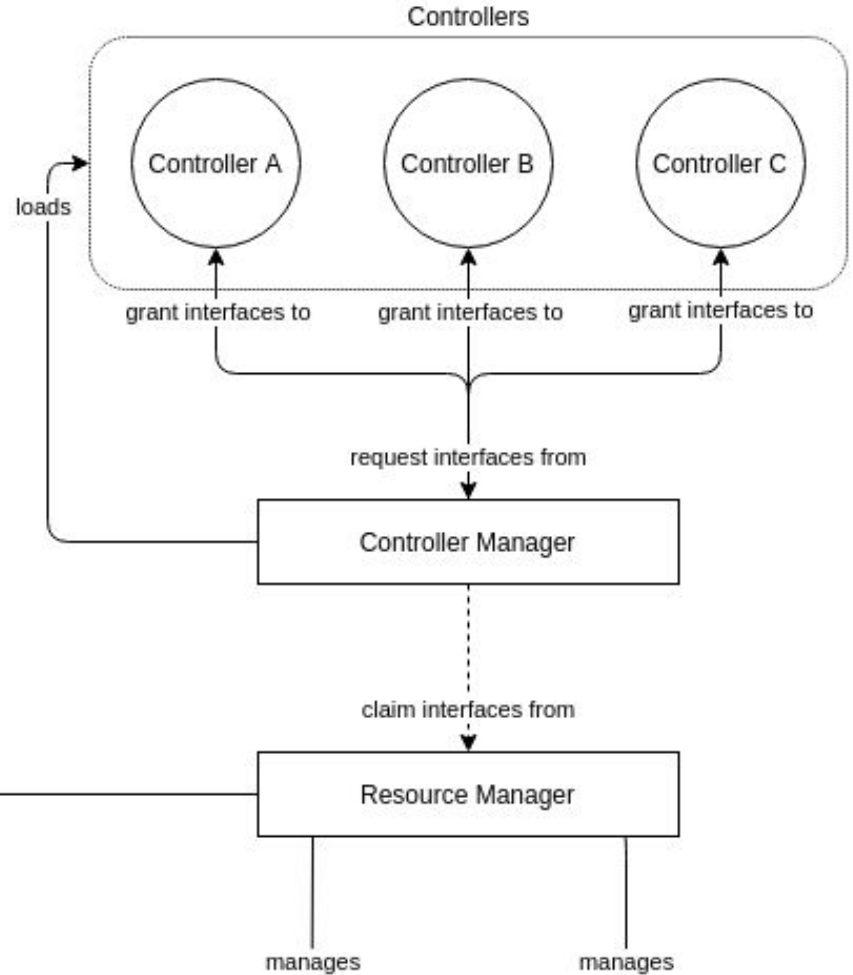
- ros_control paper in the Journal of Open Source Software
- ros2_control resources
  - https://control.ros.org
  - https://github.com/ros-controls/ros2_control
  - https://github.com/ros-controls/ros2_controllers
  - https://github.com/ros-controls/ros2_control_demos
  - https://github.com/ros-controls/roadmap/blob/master/documentation_resources.md

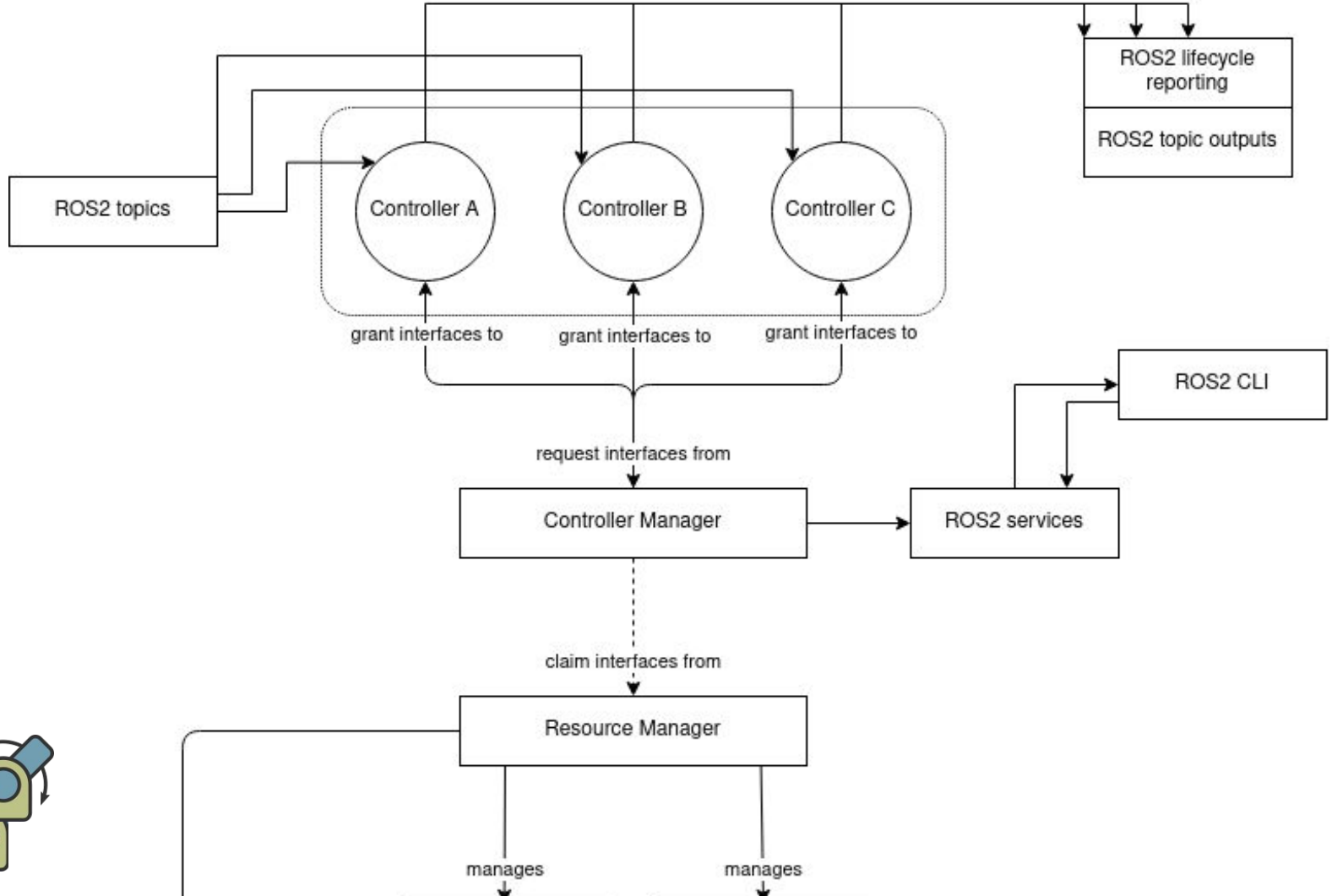


Thank you!

Karsten Knese, Victor Lopez, Jordan Palacios, Olivier Stasse, Mathias Arbo, Jaron Lundwall, Colin MacKenzie, Matthew Reynolds, Andy Zelenak, Lovro Ivanov, Jafar Abdi, Tyler Weaver, Anas Abou Allaban, Yutaka Kondo, Mateus Amarante, Auguste Bourgois and many more!

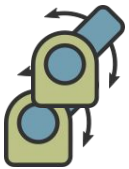

manages
manages
State Interface
<read only>
Command Interface
<read/write>
exports
exports
loads
Sensor
System
Actuator
Transmissions
may use
access
access
°C
Hardware Resources

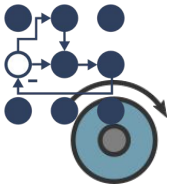

Controllers
Controller A
Controller B
Controller C
loads
grant interfaces to
grant interfaces to
grant interfaces to
request interfaces from
Controller Manager
claim interfaces from
Resource Manager
manages
manages


ROS2 lifecycle reporting
ROS2 topic outputs
ROS2 topics
Controller A
Controller B
Controller C
grant interfaces to
grant interfaces to
grant interfaces to
ROS2 CLI
request interfaces from
Controller Manager
ROS2 services
claim interfaces from
Resource Manager
manages
manages

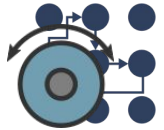
URDF and ros2_control

```
<ros2_control name="${name}" type="system">
  <hardware>
    <plugin>fake_components/GenericSystem</plugin>
  </hardware>
  <joint name="joint1">
    <command_interface name="position">
      <param name="min">-1</param>
      <param name="max">1</param>
    </command_interface>
    <state_interface name="position"/>
  </joint>
  <joint name="joint2">
    <command_interface name="position">
      <param name="min">-1</param>
      <param name="max">1</param>
    </command_interface>
    <state_interface name="position"/>
  </joint>
</ros2_control>
```

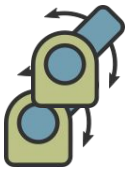


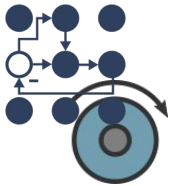


Implementing a system component

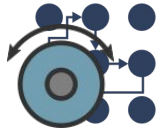


```
class RRBotHardwareInterface
  : public hardware_interface::BaseInterface<hardware_interface::SystemInterface>
{
public:
  hardware_interface::return_type configure(const hardware_interface::HardwareInfo & info) override;

  std::vector<hardware_interface::StateInterface> export_state_interfaces() override;

  std::vector<hardware_interface::CommandInterface> export_command_interfaces() override;

  hardware_interface::return_type start() override;

  hardware_interface::return_type stop() override;

  hardware_interface::return_type read() override;

  hardware_interface::return_type write() override;

private:
  std::vector<double> hw_commands_;
  std::vector<double> hw_states_;
};
```

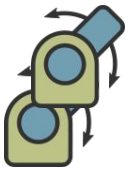


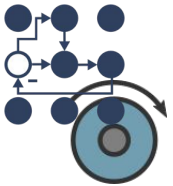


Implementing a system component



```
hardware_interface::return_type RRBotHardwareInterface::configure(
  const hardware_interface::HardwareInfo & info)
{
  if (configure_default(info) != hardware_interface::return_type::OK) {
    return hardware_interface::return_type::ERROR;
  }

  hw_states_.resize(info_.joints.size(), std::numeric_limits<double>::quiet_NaN());
  hw_commands_.resize(info_.joints.size(), std::numeric_limits<double>::quiet_NaN());

  status_ = hardware_interface::status::CONFIGURED;
  return hardware_interface::return_type::OK;
}
```

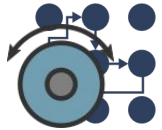




Implementing a system component

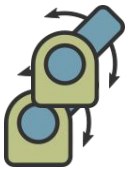
```
hardware_interface::return_type RRBotHardwareInterface::read()
{
  // read robot states from hardware, in this example print only
  RCLCPP_INFO(rclcpp::get_logger("RRBotHardwareInterface"), "Reading...");

  // write command to hardware, in this example do mirror command to states
  for (size_t i = 0; i < hw_states_.size(); ++i){
    RCLCPP_INFO(
      rclcpp::get_logger("RRBotHardwareInterface"),
      "Got state %.2f for joint %d!", hw_states_[i], i);
  }

  return hardware_interface::return_type::OK;
}

hardware_interface::return_type RRBotHardwareInterface::write()
{
  // write command to hardware, in this example do mirror command to states
  for (size_t i = 0; i < hw_commands_.size(); ++i){
    hw_states_[i] = hw_states_[i] + (hw_commands_[i] - hw_states_[i]) / 100.0;
  }

  return hardware_interface::return_type::OK;
}
```

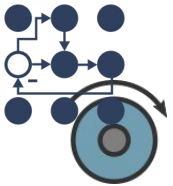
Implementing a system component



```
<ros2_control name="${name}" type="system">
  <hardware>
    <plugin>rrbot_hardware_interface/RRBotHardwareInterface</plugin>
  </hardware>
  <joint name="joint1">
    <command_interface name="position">
      <param name="min">-1</param>
      <param name="max">1</param>
    </command_interface>
    <state_interface name="position"/>
  </joint>
  <joint name="joint2">
    <command_interface name="position">
      <param name="min">-1</param>
      <param name="max">1</param>
    </command_interface>
    <state_interface name="position"/>
  </joint>
</ros2_control>
```

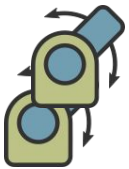
```
ros2 launch rrbot_bringup rrbot.launch.py
```

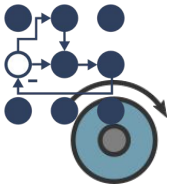




Implementing a forwarding controller

```
class RRBotControllerArray : public controller_interface::ControllerInterface
{
public:
  controller_interface::return_type init(const std::string & controller_name) override;

  controller_interface::InterfaceConfiguration command_interface_configuration() const override;

  controller_interface::InterfaceConfiguration state_interface_configuration() const override;

  CallbackReturn on_configure(const rclcpp_lifecycle::State & previous_state) override;

  CallbackReturn on_activate(const rclcpp_lifecycle::State & previous_state) override;

  CallbackReturn on_deactivate(const rclcpp_lifecycle::State & previous_state) override;

  controller_interface::return_type update() override;

...

};
```



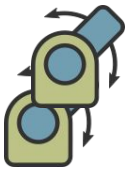


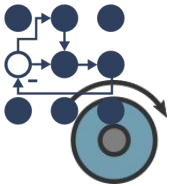
Implementing a forwarding controller

```
class RRBotControllerArray : public controller_interface::ControllerInterface
{
...

protected:
  std::vector<std::string> joint_names_;
  std::string interface_name_;

  using ControllerCommandMsg = example_interfaces::msg::Float64MultiArray;

  rclcpp::Subscription<ControllerCommandMsg>::SharedPtr command_subscriber_ = nullptr;
  realtime_tools::RealtimeBuffer<std::shared_ptr<ControllerCommandMsg>> input_command_;

  using ControllerStateMsg = control_msgs::msg::JointControllerState;
  using ControllerStatePublisher = realtime_tools::RealtimePublisher<ControllerStateMsg>;

  rclcpp::Publisher<ControllerStateMsg>::SharedPtr s_publisher_;
  std::unique_ptr<ControllerStatePublisher> state_publisher_;
};
```

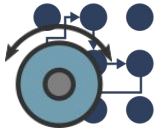
Implementing a forwarding controller

```
controller_interface::return_type RRBotControllerArray::update()
{
  auto current_command = input_command_.readFromRT();

  for (size_t i = 0; i < command_interfaces_.size(); ++i) {
    if (!std::isnan((*current_command)->data[i])) {
      command_interfaces_[i].set_value((*current_command)->data[i]);
    }
  }

  if (state_publisher_ && state_publisher_->trylock()) {
    state_publisher_->msg_.header.stamp = get_node()->now();
    state_publisher_->msg_.set_point = command_interfaces_[0].get_value();

    state_publisher_->unlockAndPublish();
  }

  return controller_interface::return_type::OK;
}
```





Implementing a forwarding controller



In rrbot_controller.xml:

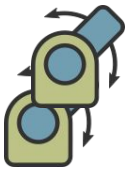
```
<library path="librrbot_controller_array">
  <class name="rrbot_controller/RRBotControllerArray"
         type="rrbot_controller::RRBotControllerArray"
 base_class_type="controller_interface::ControllerBase">
    <description>
      RRBotControllerArray ros_control controller.
    </description>
  </class>
</library>
```

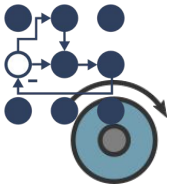
In controller.cpp

```
#include "pluginlib/class_list_macros.hpp"

PLUGINLIB_EXPORT_CLASS(rrbot_controller::RRBotControllerArray, controller_interface::ControllerBase)
```

In CMakelists.txt:

```
pluginlib_export_plugin_description_file(controller_interface rrbot_controller.xml)
```

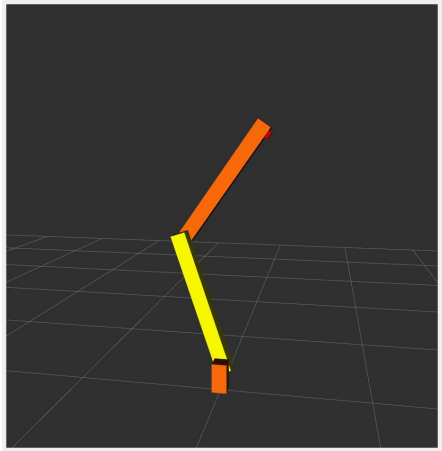




Let’s test it all!

ros2 launch rrbot_bringup rrbot_with_rrbot_controller_array.launch.py

ros2 topic pub /rrbot_controller/commands
example_interfaces/msg/Float64MultiArray "data:
- 0.5
- 0.5"



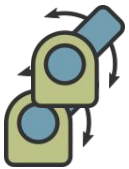
Messages and modifying a controller

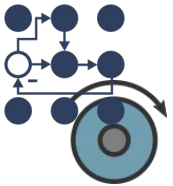
example_msgs/Float64MultiArray

```
std_msgs/MultiArrayLayout layout
  std_msgs/MultiArrayDimension[] dim
    string label
    uint32 size
    uint32 stride
  uint32 data_offset
float64[] data
```

control_msgs/JointJog

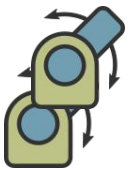
```
std_msgs/Header header
string[] joint_names
float64[] displacements
float64[] velocities
float64 duration
```

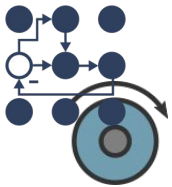


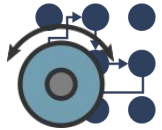


Messages and modifying a controller

```
class RRBotController : public controller_interface::ControllerInterface
{
public:
  ...
protected:
  ...

  using ControllerCommandMsg = control_msgs::msg::JointJog;

  ...
};
```

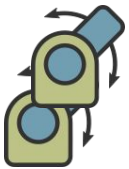


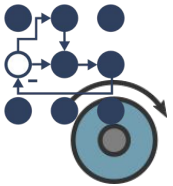


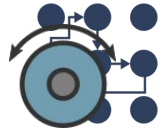


Messages and modifying a controller

```
controller_interface::return_type RRBotController::update()
{
  auto current_command = input_command_.readFromRT();

  for (size_t i = 0; i < command_interfaces_.size(); ++i) {
    if (!std::isnan((*current_command)->displacements[i])) {
      command_interfaces_[i].set_value((*current_command)->displacements[i]);
    }
  }

...

  return controller_interface::return_type::OK;
}
```







Messages and modifying a controller

ros2 launch rrbot_bringup rrbot_with_rrbot_controller.launch.py

ros2 control list_controllers
ros2 control list_hardware_interfaces

ros2 topic echo /rrbot_controller/state
ros2 topic echo /joint_states

```
ros2 topic pub /rrbot_controller/commands
control_msgs/msg/JointJog "joint_names:
- joint1
- joint2
displacements:
- 0.5
- 0.5"
```