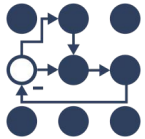
ros2_control:
From Async Hardware Drivers to RL Inference Engines



Grab a USB drive o
follow...

Open https://control.ros.org/rolling/doc/resources/roscon2026_workshop.html

Or https://control.ros.org

Github Repo

wget https://tinyurl.com/ros2control2026 -O docker-compose.yaml

docker compose pull

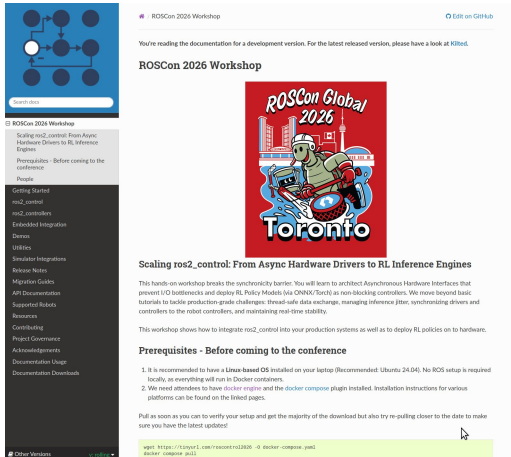
docker compose up -d

Attach from another terminal:

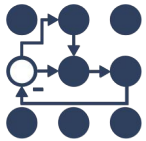
docker exec -it ros2_control_roscon26 bash

alias rc="docker exec -it ros2_control_roscon26 bash"

If you have USB run: *source setup.txt*



$whoarewe *(no command found)*



Bence Magyar [Bent'seh]
- PhD in Robotics
- Principal Software Engineer at Locus Robotics
- ros2_control PMC Leader

Christoph Fröhlich
- Research Engineer at AIT Austrian Institute of Technology
- Motion planning and system integration
- ros2_control PMC Member

Sai Kishor Kothakota
- Experienced Roboticist
- Robotics Engineer at PAL Robotics
- ros2_control PMC Member

Marq Rasmussen
- Software Engineer at Kuka Robotics
- Mobile manipulation and Planning
- ros2_control PMC Member

Denis Štogl [Dr. Denis]
- PhD, Control Engineer and Roboticist
- Robotics Consultant at Stogl Robotics Consulting
- ros2_control PMC Co-Leader

ros-controls Organization

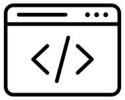
Charter

Project charter

Meetings

CEST

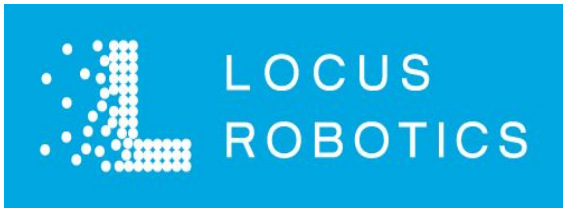
Repositories



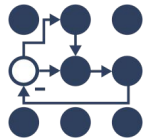
GitHub ros-controls

Docs

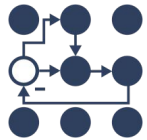
control.ros.org



Thank you for being here!!

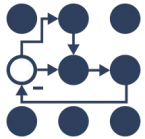


Christoph Fröhlich, Sai Kishor Kothakota, Marq Rasmussen, Bence Magyar, Denis Štogl, Erik Holum, Alejandro Hernández Cordero, Eneji Peacemaker Ohieku, Shahazad Abdullah, Kamalkant Thangaraju, Vedhas Talnikar, Ishan Pathak, Nikola Banović, Sahil Lakhmani, Christian Rauch, M. Ege Kural, Junius Santoso, Nathan Dunkelberger, Julia Jia, Soham Patil, Dennis Lanov, Sebastian Castro, Suryansh Singh, Tobias Fischer, Shlok Mehndiratta, José Luis Pérez Martín, Sergi de las Muelas, Krzysztof Pochwała, Shivam Maurya, Naitik Pahwa, Michele Cereda, Abdullah Theonewhomadethings, Felix Exner, Souri Rishik, Noel Jiménez García, Dhruv Patel ..... and many more!



Container Setup

ros2_control:
From Async Hardware Drivers to RL Inference Engines



Grab a USB drive o
follow...

Open https://control.ros.org/rolling/doc/resources/roscon2026_workshop.html

Or https://control.ros.org

Github Repo

wget https://tinyurl.com/ros2control2026 -O docker-compose.yaml

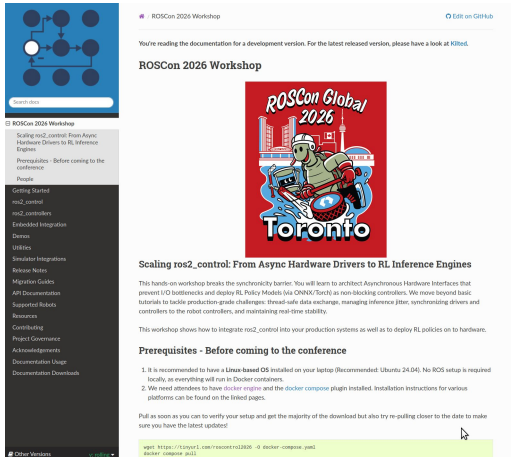
docker compose pull

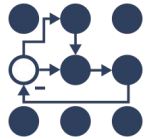
docker compose up -d

Attach from another terminal:

docker exec -it ros2_control_roscon26 bash

alias rc="docker exec -it ros2_control_roscon26 bash"





ros2_control Overview

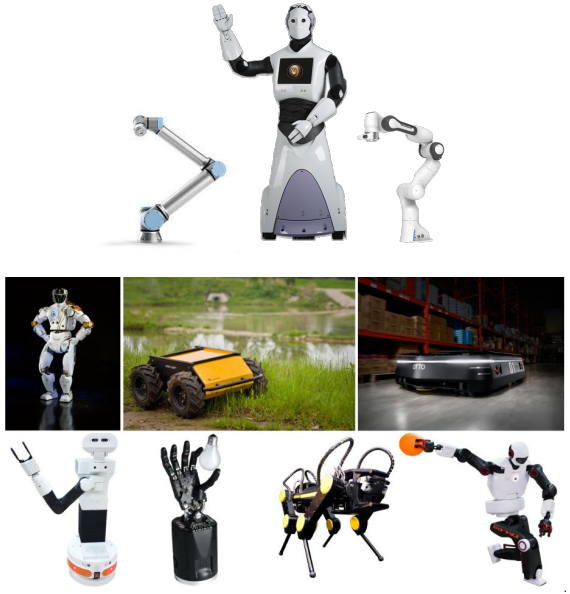
History

pr2_controller_manager

(pr2_mechanism)

ros_control

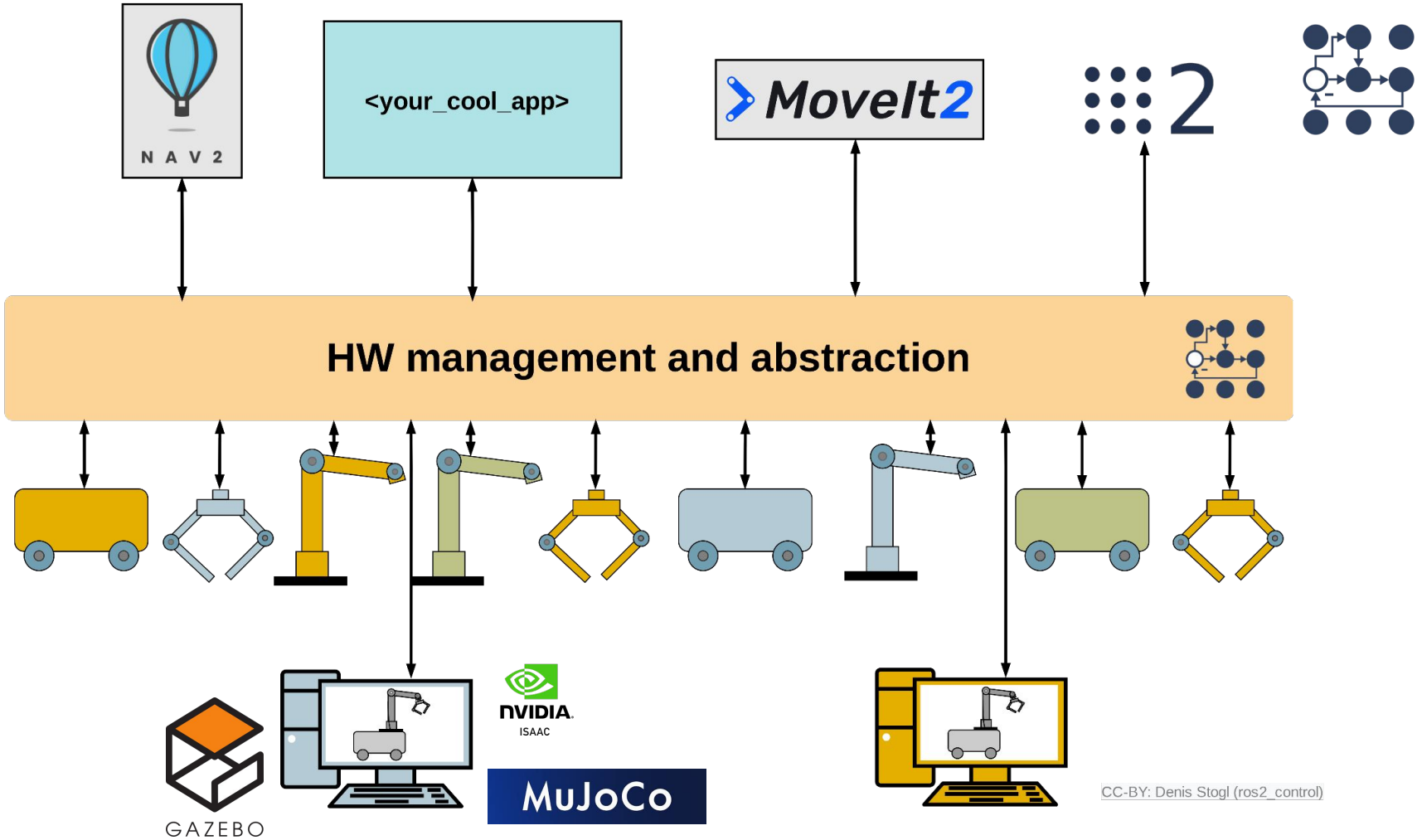
2012/2017

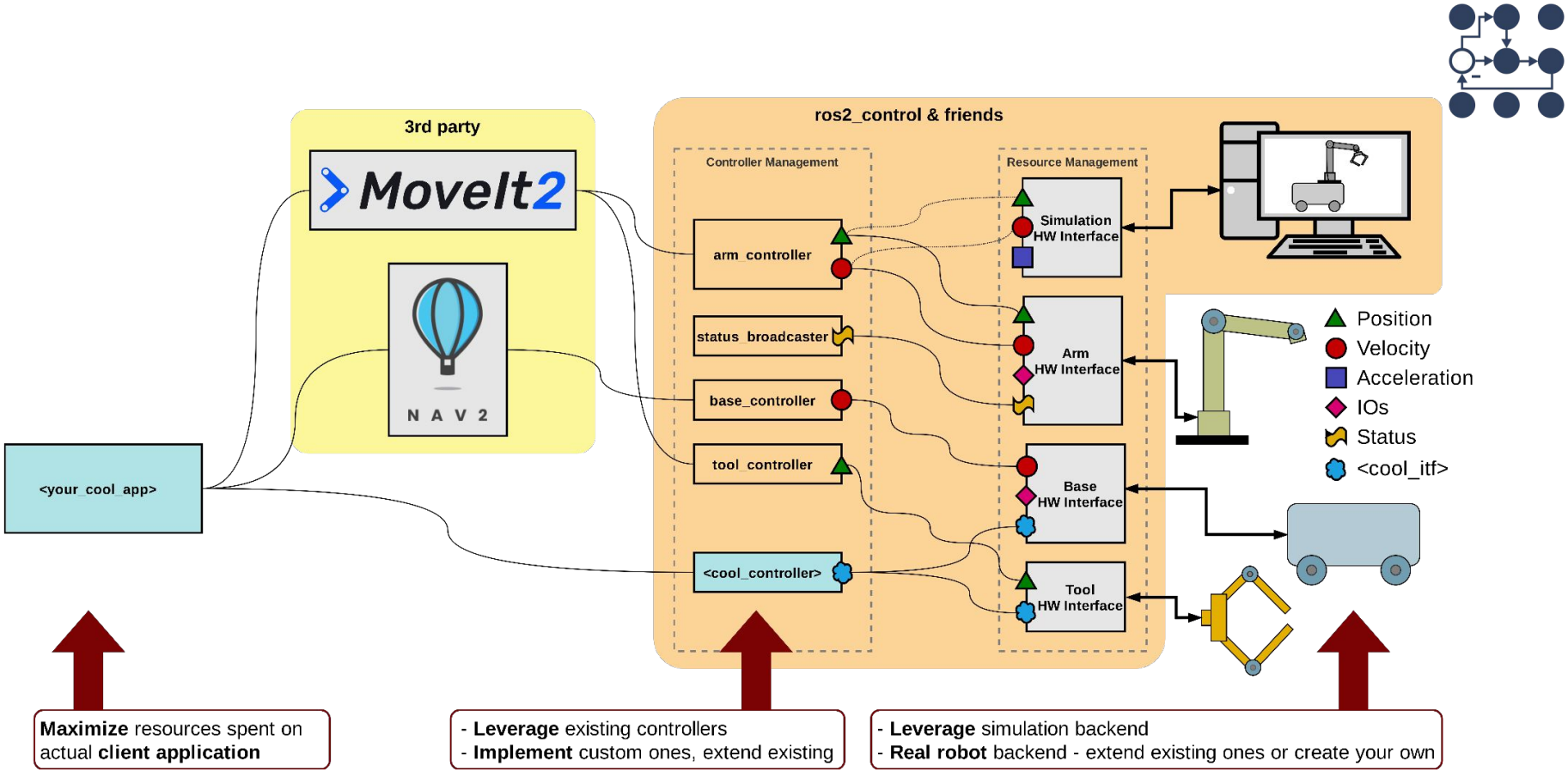


ros2_control

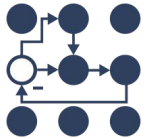
2017/today

https://control.ros.org/master/doc/supported_robots/supported_robots.html




3rd party
MoveIt2
NAV 2
ros2_control & friends
Controller Management
Resource Management
arm_controller
status_broadcaster
base_controller
tool_controller
<cool_controller>
Simulation HW Interface
Arm HW Interface
Base HW Interface
Tool HW Interface
Position
Velocity
Acceleration
IOs
Status
<cool_itf>
<your_cool_app>
Maximize resources spent on actual **client application**
- **Leverage** existing controllers
- **Implement** custom ones, extend existing
- **Leverage** simulation backend
- **Real robot** backend - extend existing ones or create your own

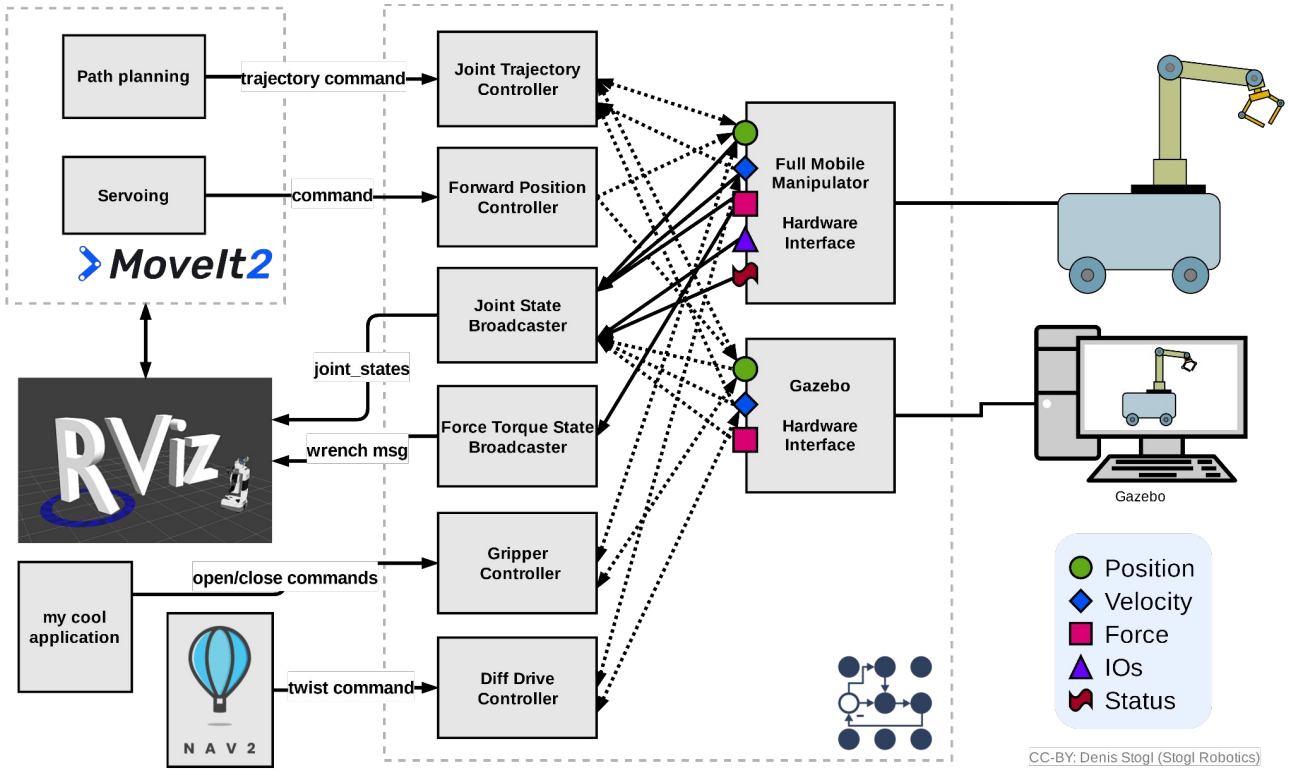
Overview of the Main “Components”



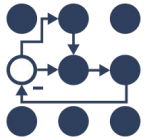
- ros2_control_node—executable that runs Controller Manager » *when using real or mock hardware*
    - Gazebo - the node is embedded in the simulator
    - Isaac Sim - uses ROS topic based interface
    - MuJoCo - uses a separate node, connects via MuJoCo API
- Hardware Components—hardware drivers
    - Defined & configured in URDF (xml - “ros2_control”-tag)
- Controllers
    - Configured through parameters (*yaml*) on Controller Manager
- ROS interfaces
    - ControllerManager services & diagnostics topics
    - Controllers define their own ROS interfaces, typically actions and/or topics
- Standard ROS CLI verb

Configuring standard controllers

```yaml
controller_manager:
  update_rate: 500  # Hz

  joint_trajectory_controller:
    type: joint_trajectory_controller/JointTrajectoryController

  forward_position_controller:
    type: position_controllers/JointGroupPositionController

  joint_state_broadcaster:
    type: joint_state_broadcaster/JointStateBroadcaster

  force_torque_sensor_broadcaster:
    type: force_torque_sensor_broadcaster/ForceTorqueStateBroadcaster

  gripper_controller:
    type: position_controllers/GripperActionController

  diff_drive_controller:
    type: diff_drive_controller/DiffDriveController

joint_trajectory_controller:
  joints:
    - joint1
    - ...
  command_interfaces:
    - position
  state_interfaces:
    - position
    - velocity

forward_position_controller:
  joints:
    - joint1
    - ...

force_torque_sensor_broadcaster:
  sensor_name: tcp_fts_sensor
  frame_id: tool0
  topic_name: ft_data


gripper_controller:
  joints:
    - gripper_joint
  command_interface: position

diff_drive_controller:
  left_weel_names:
    - left_wheel_1
    ...
```



Overview of Hardware Components



Base Classes:

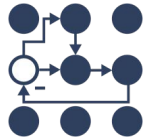
- ~~Sensor Component - **only** state interfaces~~
- ~~Actuator Component - interfaces only relevant to **one joint**~~
- System Component - state and command interfaces **all you want**

Technology stack

- Hardware Components are subclassed from the above
- Written in C++
- Exported as pluginlib plugins
- Configured via <ros2_control> XML section in URDF
- Can use internally anything they want

https://control.ros.org/rolling/doc/supported_robots/supported_robots.html

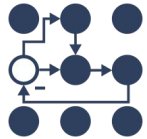
Example Hardware Definition *“the <ros2_control> - Tag”*



```
<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>
```

CLI - *ros2controlcli package*



- ros2 control TAB TAB (check autocompletion)

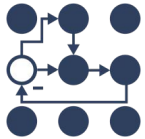
- Management of controllers and hardware components
- Introspection of the system
  - What is running, what is loaded, what is available?
  - Which interfaces are offered by HW component?
  - Which controller claims which interfaces?

Hot tip:

ros2 has a modular CLI system.
You can easily write your own CLI verbs too!

https://control.ros.org/rolling/doc/ros2_control/ros2controlcli/doc/userdoc.html

ros2_control:
From Async Hardware Drivers to RL Inference Engines



Grab a USB drive o
follow...

Open https://control.ros.org/rolling/doc/resources/roscon2026_workshop.html

Or https://control.ros.org

Github Repo

wget https://tinyurl.com/ros2control2026 -O docker-compose.yaml

docker compose pull

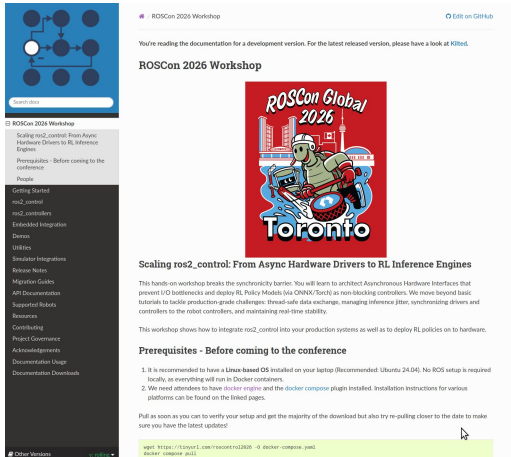
docker compose up -d

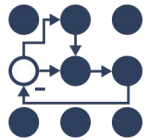
Attach from another terminal:

docker exec -it ros2_control_roscon26 bash

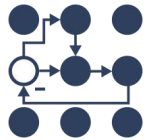
alias rc="docker exec -it ros2_control_roscon26 bash"

If you have USB run: *source setup.txt*



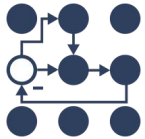


The Synchronous Problem



Introduction to the standard loop. Demonstrating jitter/overruns and explain their significance in realtime.

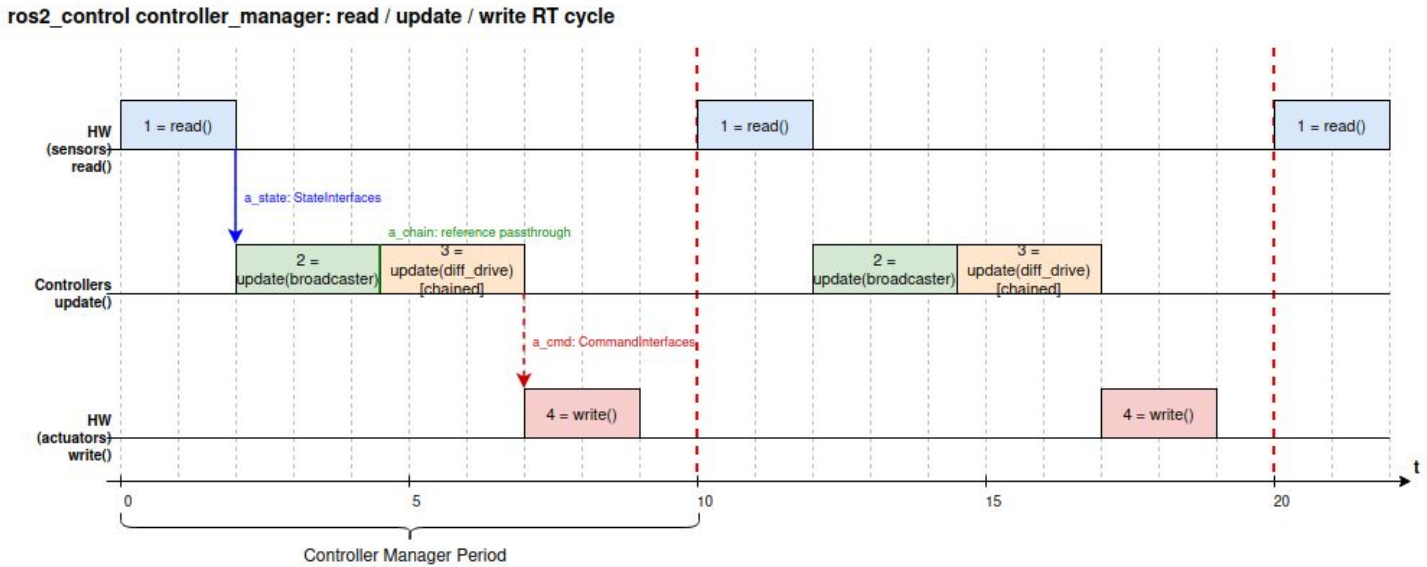
controller_manager RT cycle



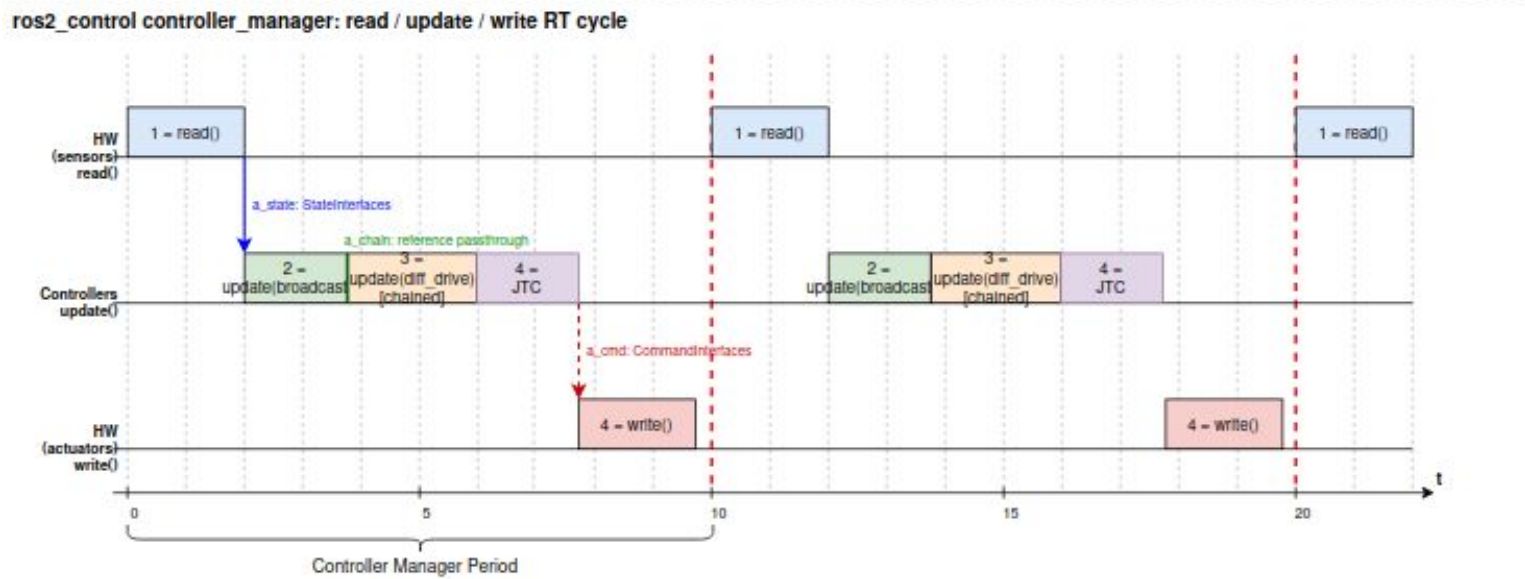
- read->update->write
- Controller execution order is sorted with an algorithm similar to Directed Acyclic Graph
- Hardware execution order acc. to URDF

```
while (rclcpp::ok())
{
        // calculate measured period
        auto const current_time = get_cm_now(cm);
        auto const measured_period = current_time - previous_time;
        previous_time = current_time;

        // execute update loop
        cm->read(current_time, measured_period);
        cm->update(current_time, measured_period);
        cm->write(current_time, measured_period);

        // sleep until next cycle
        next_iteration_time += period;
        std::this_thread::sleep_until(next_iteration_time);
}
```

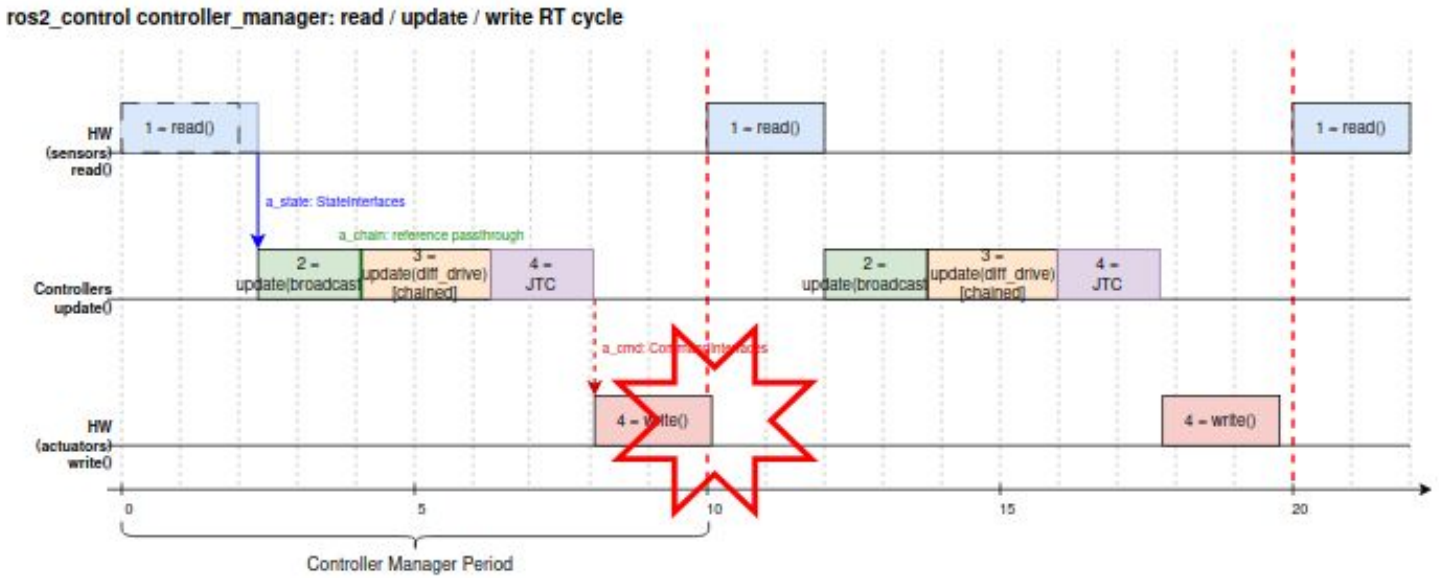
controller_manager RT cycle: ideal case

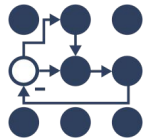


controller_manager RT cycle: bad case



controller_manager RT cycle: bad case



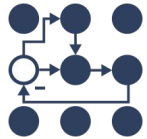


Benchmarking & Visualization

Two numbers, two failures

Periodicity and execution time break in different ways

The controller_manager publishes both, for every controller and hardware component



Periodicity (Hz)

THE QUESTION

Did I run when I said I would?

BAD LOOKS LIKE

Drift, jitter, missed deadlines

Execution time (µs)

THE QUESTION

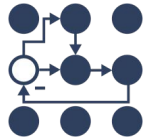
How long did the work take?

BAD LOOKS LIKE

Your code is too slow

A component can be perfectly periodic and far too slow.
It can be very fast and still miss half its cycles.

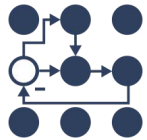
Where the numbers already are



- **`~/statistics`** — execution time and periodicity
- **`/diagnostics`** — the rolled-up OK / WARN / ERROR summary
- **`~/introspection_data`** — every state and command interface at each update cycle
- Each publishes `/full`, `/names` and `/values`. Echo /full by hand, plot /names + /values, which stay cheap at 1 kHz.

```
# is anything already unhappy?
ros2 topic echo /diagnostics --once


# the raw per-component numbers
ros2 topic echo /controller_manager/statistics/full
ros2 control list_hardware_components -v
```

The default thresholds

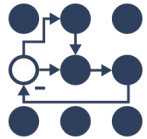


All of these are parameters under diagnostics.threshold.*
Tune them per deployment rather than learning to ignore the warnings.

	WARN	ERROR	
Periodicity — mean error	5 Hz	10 Hz	off-rate by 5 Hz at 100 Hz
Periodicity — std dev	5 Hz	10 Hz	the jitter number
Execution time — mean error	1000 µs	2000 µs	1 ms of a 10 ms budget
Execution time — std dev	100 µs	200 µs	tightest of the four

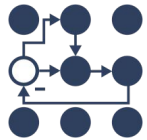
Watch it live

PlotJuggler, and adding your own variables to the stream



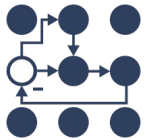
- **`ros2 run plotjuggler plotjuggler`** → Streaming → ROS2 Topic Subscriber → pick `/names` and `/values`, then drag
- Anything you can cast to a double can join the same stream
- Registration is not real-time safe: do it in `on_configure()` only
  - make sure the variable outlives the component or you will segfault reading freed memory

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


CallbackReturn MySpecialHW::on_configure(const rclcpp_lifecycle::State &)
{
  REGISTER_ROS2_CONTROL_INTROSPECTION("bus_retry_count", &bus_retry_count_);
  ...
}
```



Hands-on: Analysis of RT cycle

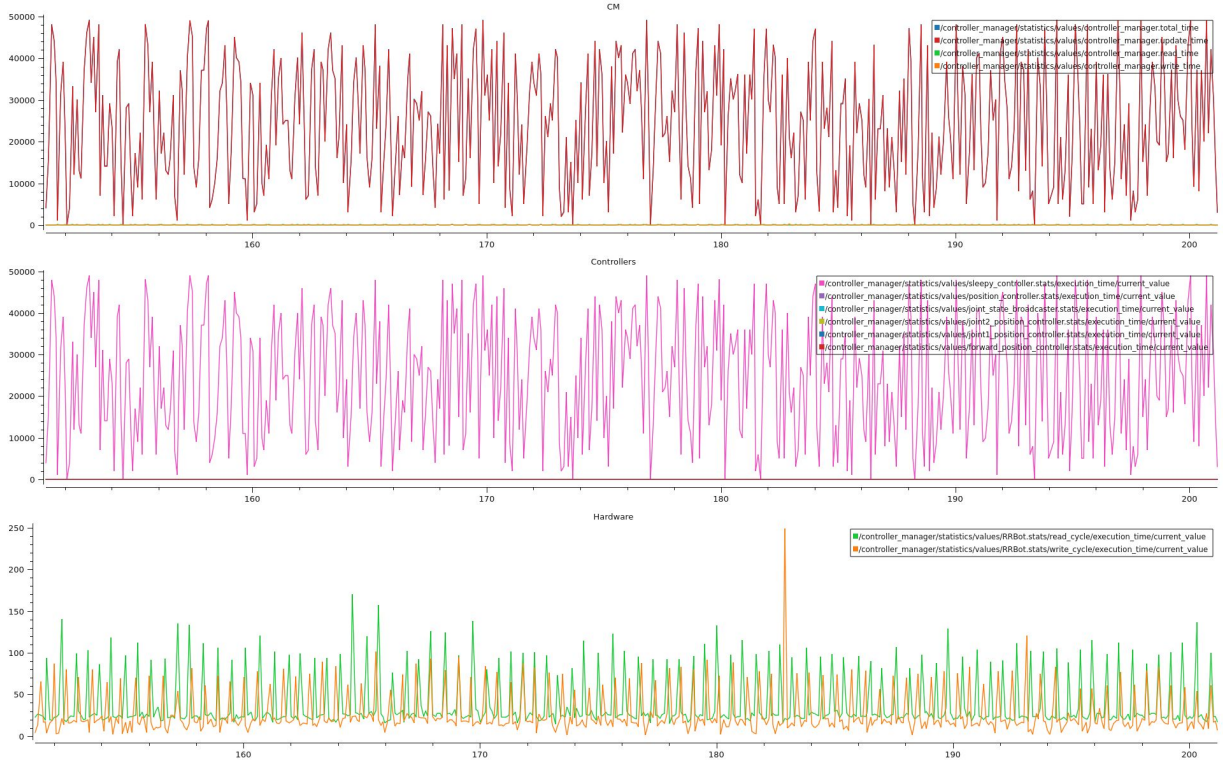
Hands-on: Analysis of RT cycle



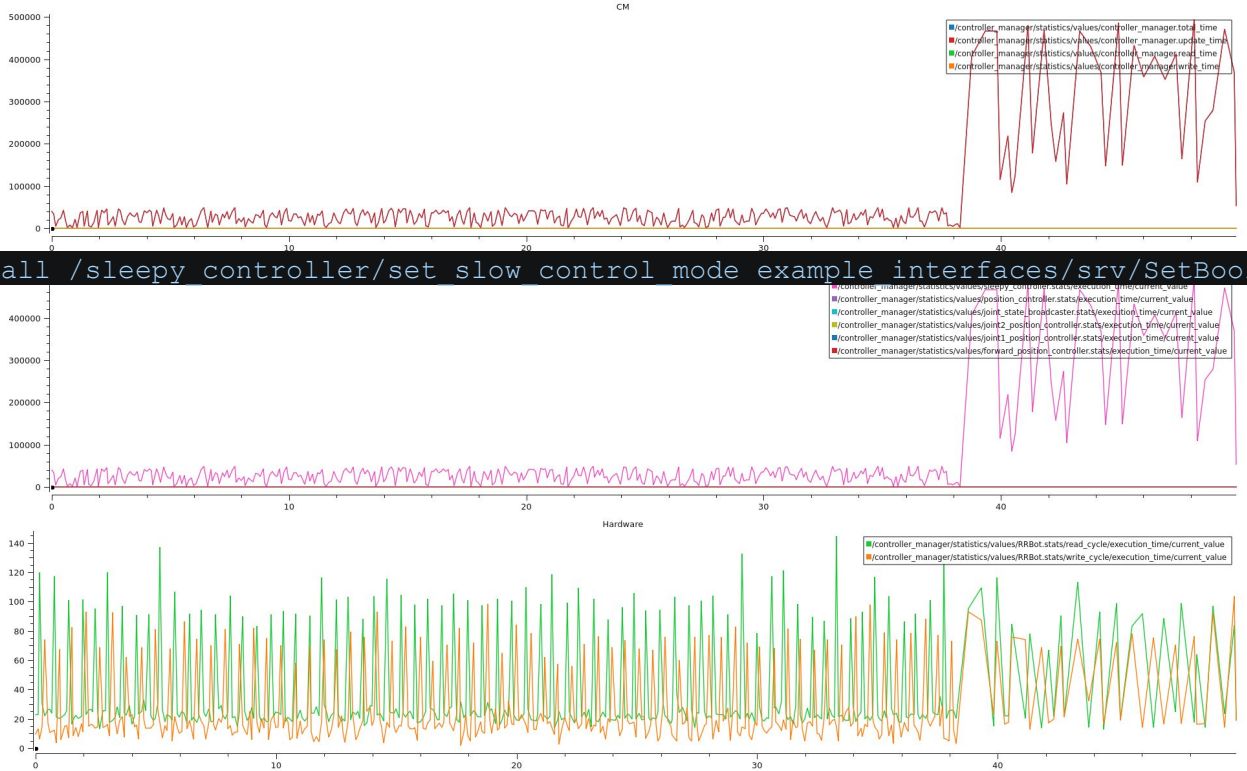
Starting a simple example and analyze the setup

```
$ ros2 launch workshop_bringup rrbot.launch.py
$ ros2 control list_controllers
$ ros2 run rqt_controller_manager rqt_controller_manager
$ ros2 control view_controller_chains --save && qpdfview controller_diagram.pdf
$ ros2 run swri_console swri_console
$ ros2 run plotjuggler plotjuggler --layout src/roscon2026_control_workshop/workshop_bringup/pj.xml
```

Hands-on: Analysis of RT cycle

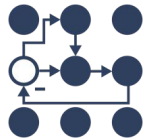


Hands-on: Analysis of RT cycle




```
ros2 service call /sleepy_controller/set_slow_control_mode example_interfaces/srv/SetBool "data: true"
```

Async Saves You

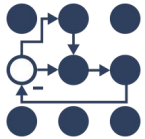


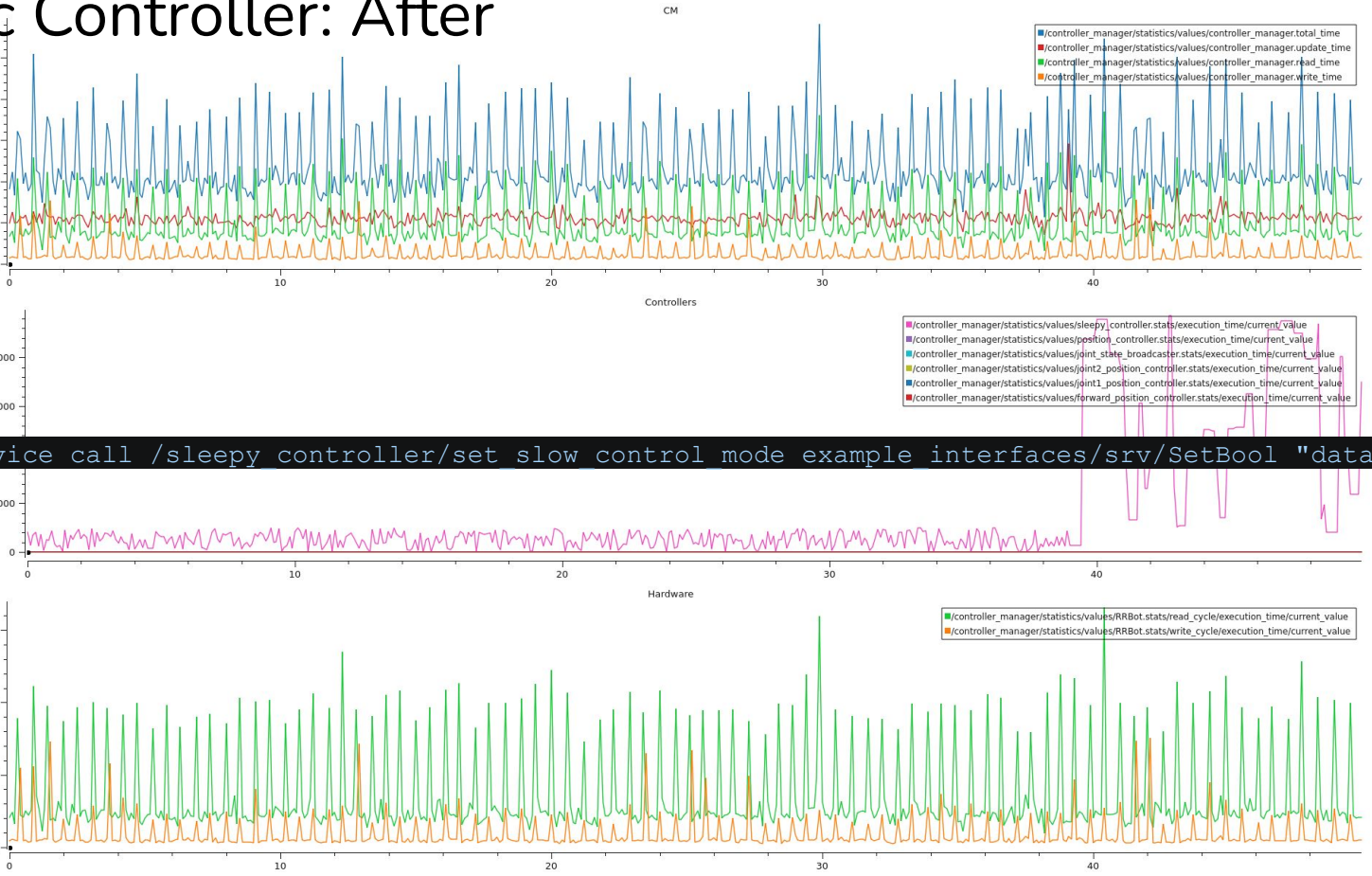
Make it async, restart

```
sleepy_controller:
 ros__parameters:
   type: workshop_controllers/SleepyController
   max_sleep_time: 0.5
   is_async: true
```

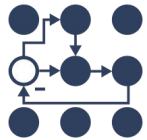
Async Controller: Before

Async Controller: After





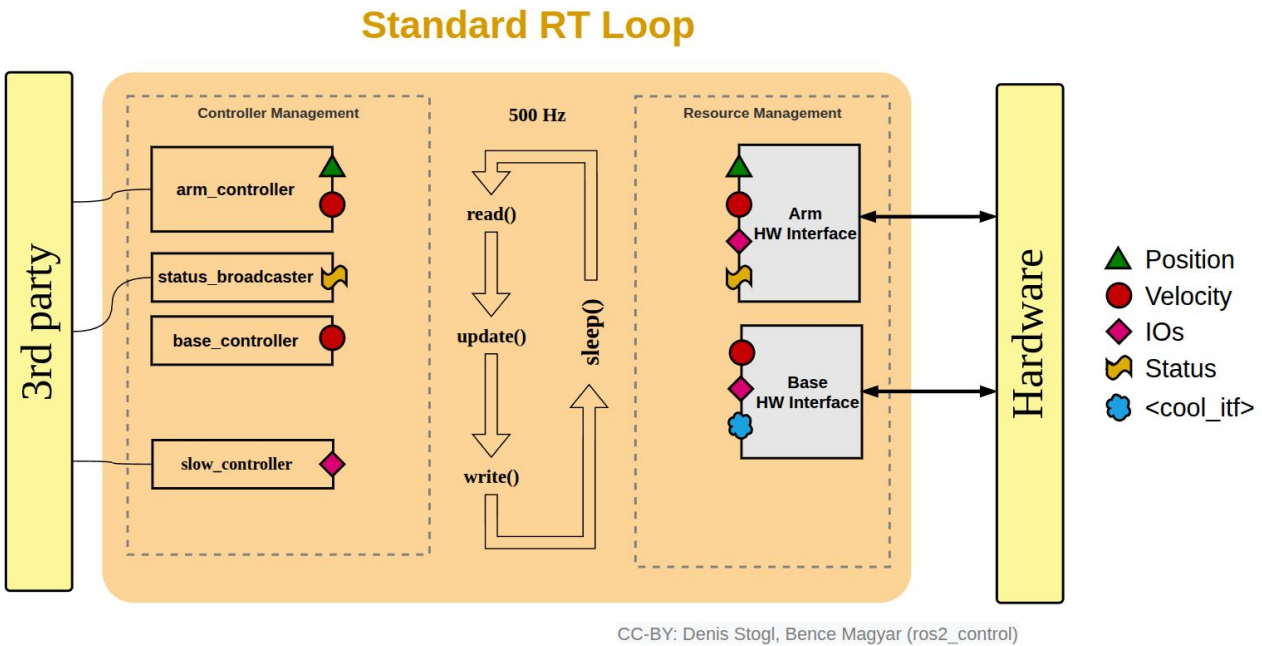

```
ros2 service call /sleepy_controller/set_slow_control_mode example_interfaces/srv/SetBool "data: true"
```



The main loop

One thread.

1. read() every hardware component
2. update() every controller
3. write() every hardware component
4. sleep() until the next deadline.



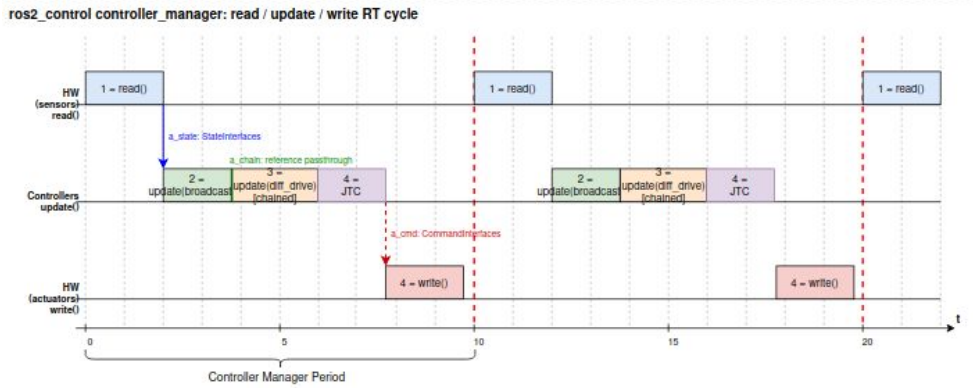
Synchronous cycle

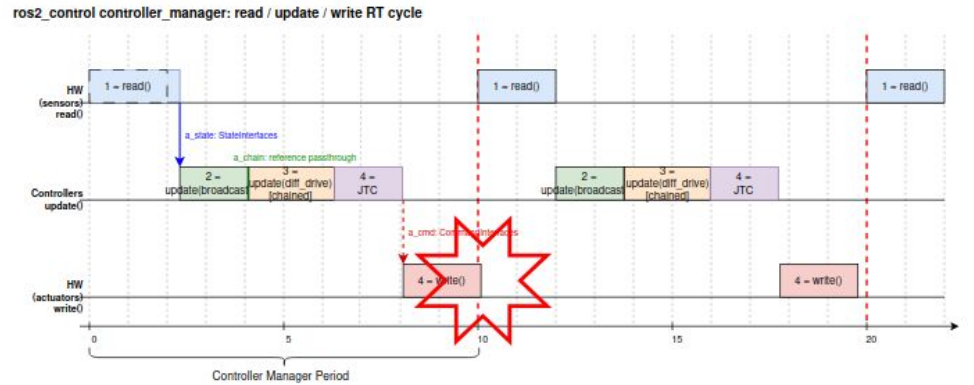
All active ***hardware_interfaces*** & ***controllers*** share the "***read /update/write***" loop

If one component is slow then everything misses rate

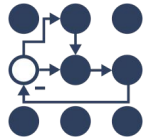
Two options:

- Slow everything down
- Take the slow component off the loop





is_async="true"



- For **hardware_interface** add the attribute to the **`<ros2_control>`** tag of your **ros2_control.xacro**
- For **controllers** add the attribute to the **ros2_control.yaml**
- The ResourceManager wraps the component in an **`AsyncFunctionHandler`** from realtime_tools

ros2_control.xacro

```
<ros2_control name="${name}"
              type="system"
              is_async="true">
  ...
</ros2_control>
```

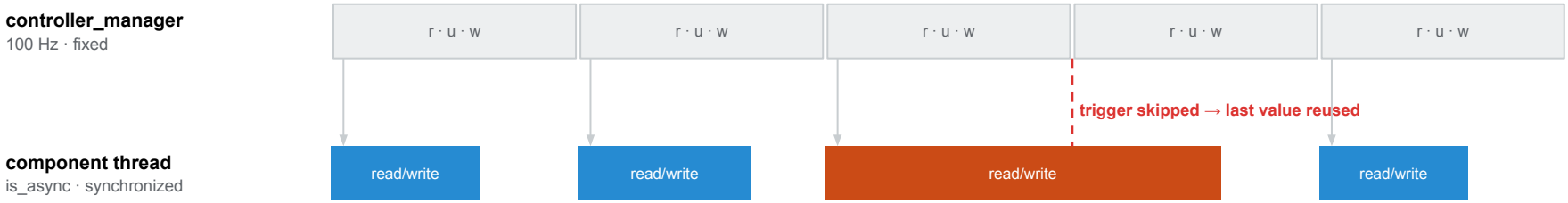
ros2_control.yaml

```
sleepy_controller:
 ros__parameters:
   type: workshop_controllers/SleepyController
   max_sleep_time: 0.5
   is_async: true
```

Mode 1: synchronized

- The controller_manager triggers **read/write**
- The component runs on its own thread and the CM carries on without waiting

  When the previous cycle has not finished, the trigger is skipped and the last value is reused
- Timing stays phase-locked to the control cycle

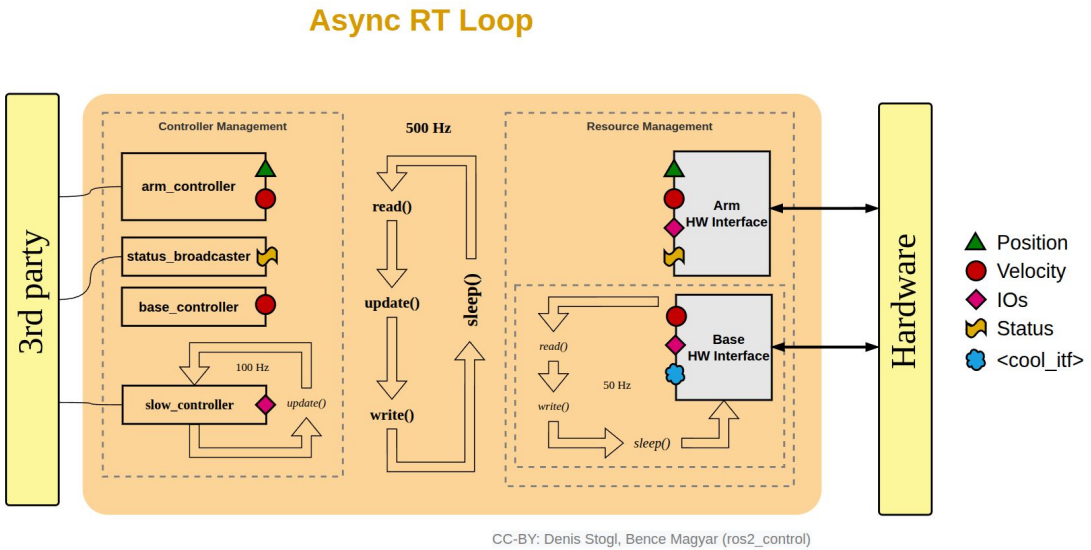



```
<ros2_control name="${name}" type="system"
              is_async="true"  rw_rate="500">
  <properties>
    <async scheduling_policy="synchronized"/>
  </properties>
  ...
</ros2_control>
```

Mode 2: detached

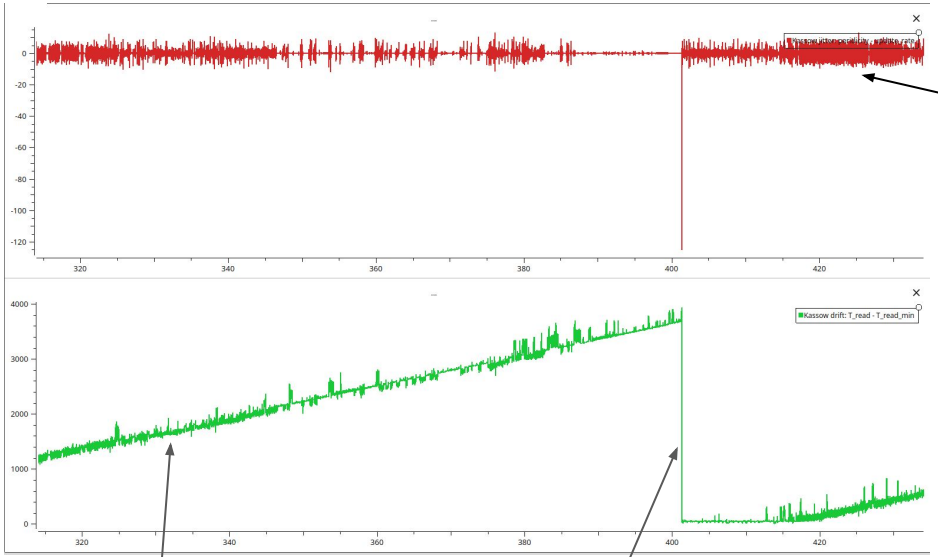
- The component owns its clock and read/write loop
- Best for hardware that dictate their own timing
  Example: blocking socket reads that are not held to a fixed rate
- State and command are no longer aligned to the control cycle. Data age becomes a design parameter you have to reason about
- Two clocks with no common divisor: the newest sample is anywhere from 0 to 20 ms old, and the value that is used is a race.

```
<ros2_control name="${name}" type="system"
              is_async="true"  rw_rate="500">
  <properties>
    <async scheduling_policy="detached"/>
  </properties>
  ...
</ros2_control>
```

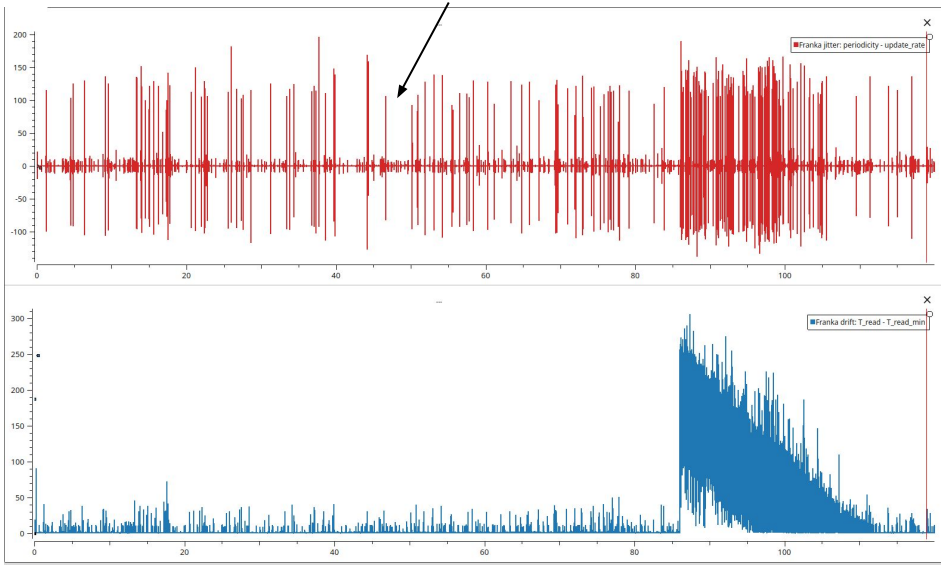


Mode 2: detached - problem loop drifting

Kassow (500 Hz) and Franka (1 kHz), both free-running against the CM loop



Flat update rate, drift is invisible to the jitter metric



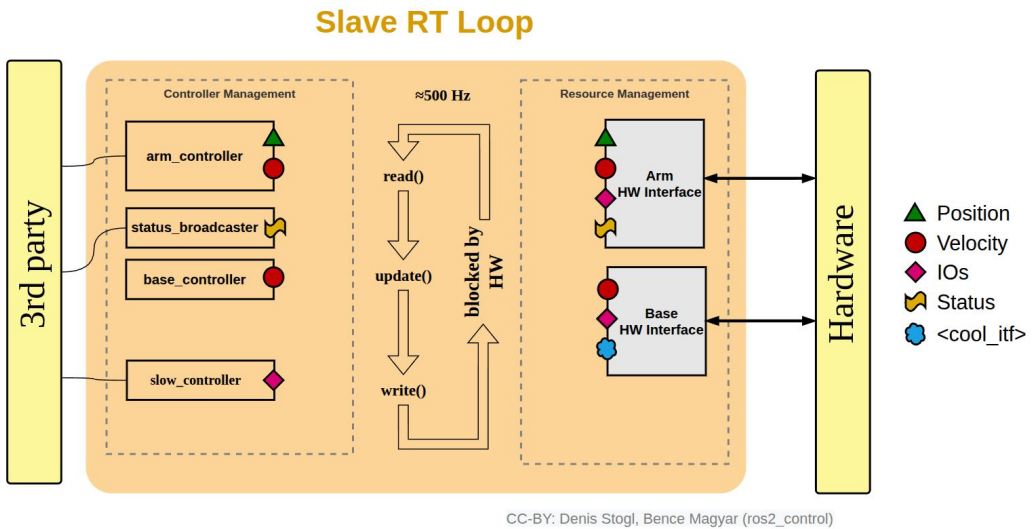
Age of the newest sample

Phase of the 2 clocks slips a full cycle, then starts over

Mode 3: hardware_driven (a.k.a slave)

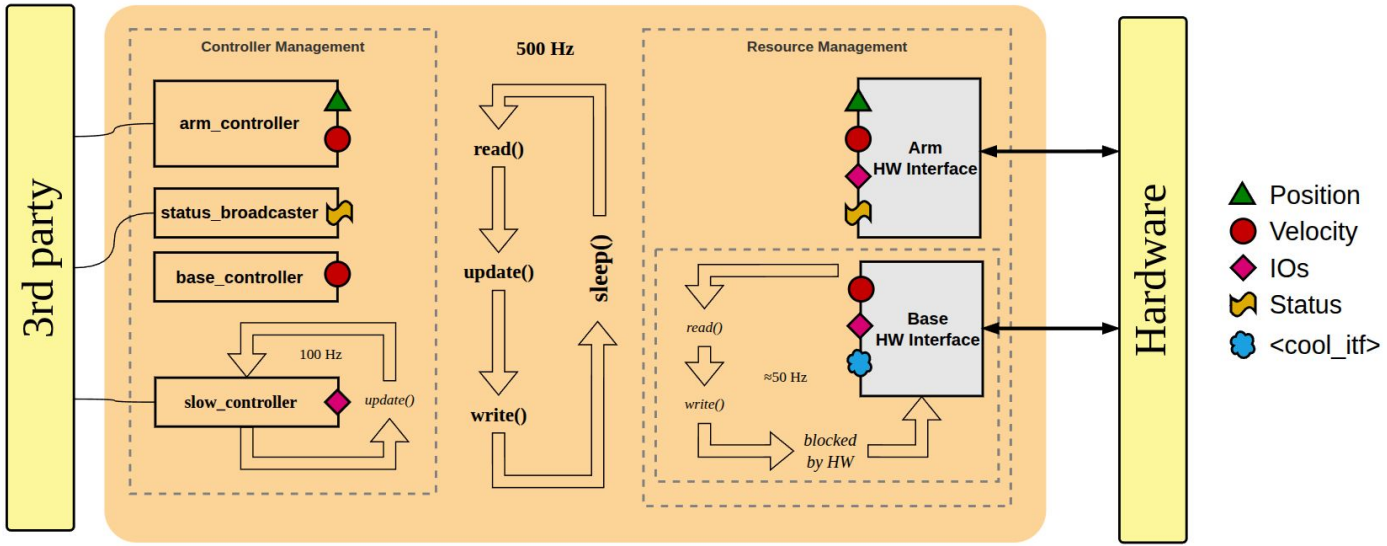
- hardware's blocking read() sets the rate
- Use when fieldbus frame already is the clock EtherCAT, CAN, anything where the master's cycle is authoritative.
- The loop inherits the bus's jitter characteristics instead of the kernel's timer jitter (but your compute still needs to keep up)

```yaml
controller_manager:
  ros__parameters:
    hardware_synchronization:
      expect_blocking_read_write: true
      minimum_cycle_time: 0.0001
```



Combining Async & Slave

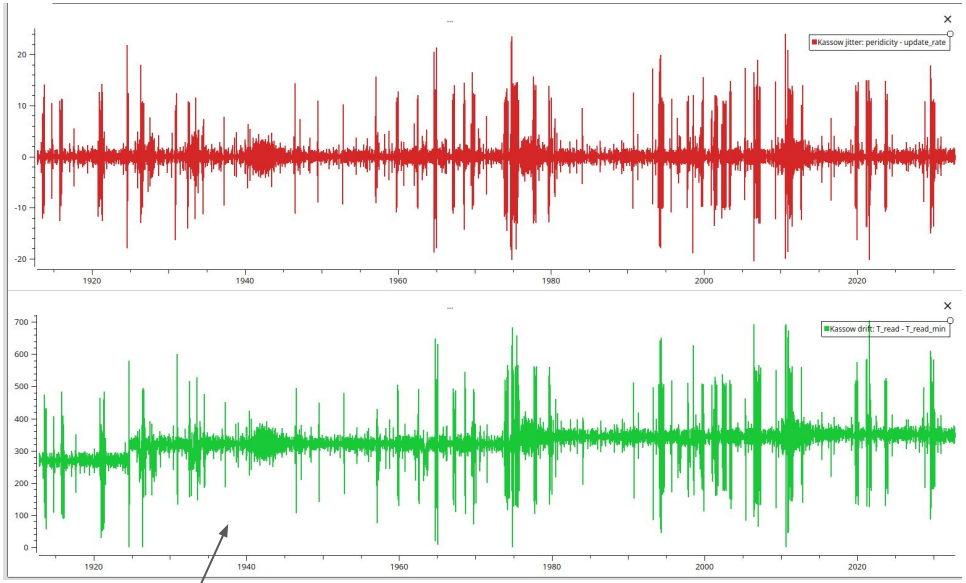
Async Slave RT Loop



Base HW Interface runs on its own thread at its own rate

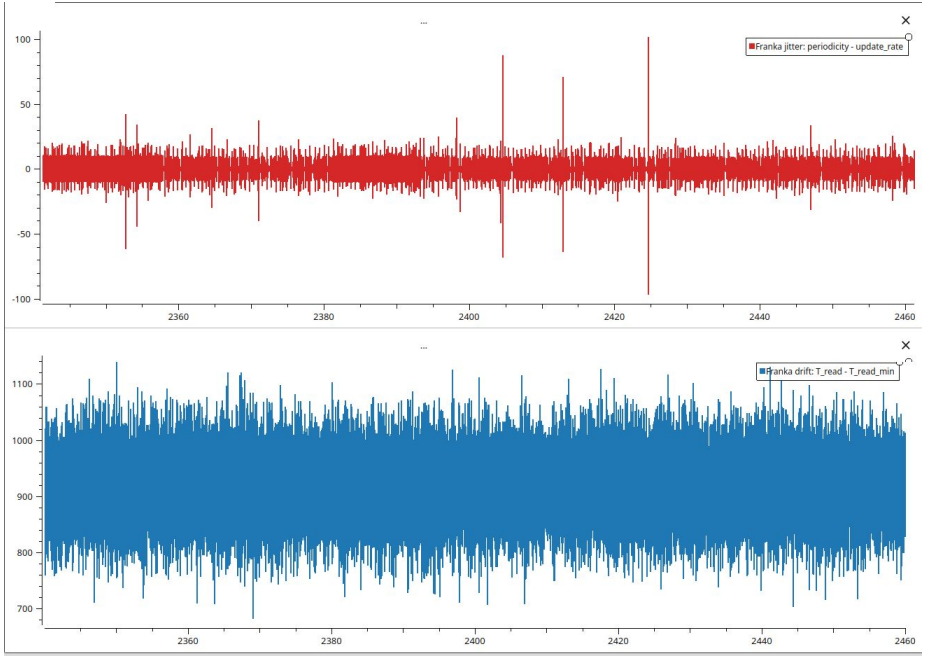
1. inside that thread its read() blocks on the hardware instead of sleeping
2. The 500 Hz main loop never waits for either

Solving the sync problem (on hardware side)

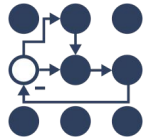


Age is now bounded by one bus frame

We didn't measure the drift away.
We removed the second clock!



Tuning Your Async hardware_interface

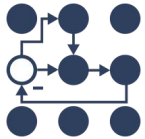


```
<ros2_control name="${name}" type="system"
              is_async="true"  rw_rate="500">
  <properties>
    <async thread_priority="60"
           affinity="[2,3]"
           scheduling_policy="detached"/>
  </properties>
  ...
</ros2_control>
```

- **`thread_priority`**
  0–99, default 50, SCHED_FIFO. Leave it at 50 and you are competing with the controller_manager itself.
- **`affinity`**
  Pin it off the CM core otherwise you have not removed contention, you have renamed it.
- **rw_rate**
  Read/write rate. Capped at the CM rate.

Which mode?

Who owns the clock?



Synchronized

CLOCK OWNER
controller_manager

DOES THE CM WAIT?
No it triggers and moves on

USE WHEN
The driver is usually fast but spikes

WHAT BITES YOU
Skipped triggers silently reuse stale values

Detached

CLOCK OWNER
The component itself

DOES THE CM WAIT?
No It doesn’t trigger it

USE WHEN
The transport dictates the rate

WHAT BITES YOU
Data age drifts and never repeats

Hardware-Driven Mode

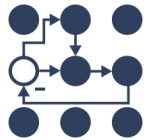
CLOCK OWNER
The bus / fieldbus master

DOES THE CM WAIT?
Yes (blocking read)

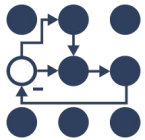
USE WHEN
The frame is already authoritative

WHAT BITES YOU
A driver that doesn't actually block the loop free-runs at the `minimum_cycle_time`

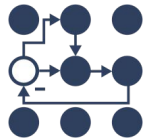
What Async does not fix



- **It does not make your driver faster.** It makes the loop survive a slow driver
- **Stale data is now yours to manage.** A command written three cycles late is still a wrong command
- **Two threads means two threads.** Priority inversion, cache contention and core assignment are now your problem
- **The kernel still decides.** rtprio limits, locked memory and an RT or lowlatency kernel are prerequisites
  Async on a stock kernel relocates jitter rather than removing it



Hands-on: Decoupling I/O

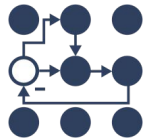


We increase processing time

```
<hardware>
   <plugin>workshop_hardware/RRBotSystemPositionOnlyHardware</plugin>
   ...
   <param name="write_processing_time_ms">150</param>
</hardware>
```

and see the results

10 Hz → 100 ms budget
150 ms write causes overrun

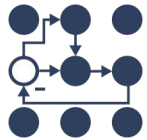


Configure async and scheduling_policy

```
<ros2_control name="${name}" type="system"
              is_async="true" rw_rate="5">
  <properties>
    <async thread_priority="60"
           scheduling_policy="detached"
           print_warnings="true"/>
  </properties>
</ros2_control>
```

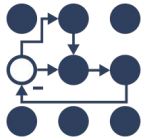
restart and see the results

Refactoring a blocking hardware component into a Detached Async setup with AsyncFunctionHandler



- Example_12
  - Add parameter called processing time to both controller and HW
  - Adding write_processing_time, we show the overruns
  - Adding processing time to passthrough controller we show the overrun
- Is_async is your friend
- Thread properties to be able to choose the type of synchronization
  - Synchronized
  - Detached
-

ros2_control:
From Async Hardware Drivers to RL Inference Engines



Grab a USB drive o
follow...

Open https://control.ros.org/rolling/doc/resources/roscon2026_workshop.html

Or https://control.ros.org

Github Repo

wget https://tinyurl.com/ros2control2026 -O docker-compose.yaml

docker compose pull

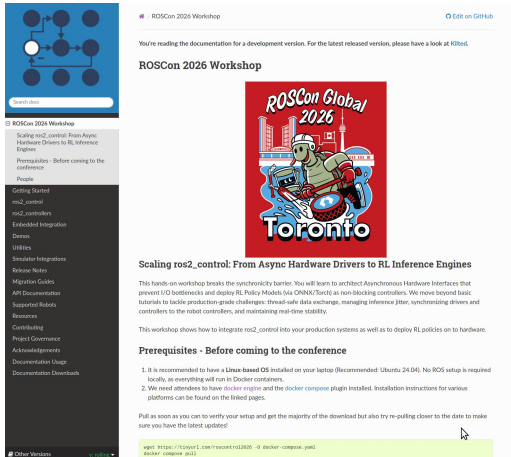
docker compose up -d

Attach from another terminal:

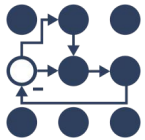
docker exec -it ros2_control_roscon26 bash

alias rc="docker exec -it ros2_control_roscon26 bash"

If you have USB run: *source setup.txt*



Cheatsheet



Run this please:

- echo 'alias rc="docker exec -it ros2_control_roscon26 bash"' >> ~/.bashrc

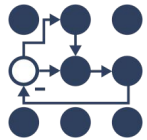
Cheat codes:

- rc = open shell in the container
- z = start up the zenoh daemon
- sim = mujoco with Kangaroo
- demo = deploy Kangaroo walking policy
- teleop = keyboard teleop

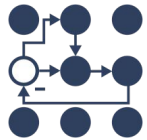
Everything also available in https://tinyurl.com/ros2control-workshop

cd roscon2026_control_workshop

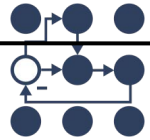
RUN “docker compose up -d”



Coffee break until 14:45



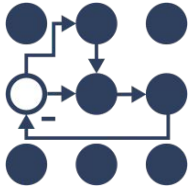
RL Deployment



Physical AI Inference and Trajectory Upscaling for ros2_control

Contributor: Vedhas Talnikar

Mentors: Dr. Bence Magyar, Sai Kishor Kothakota



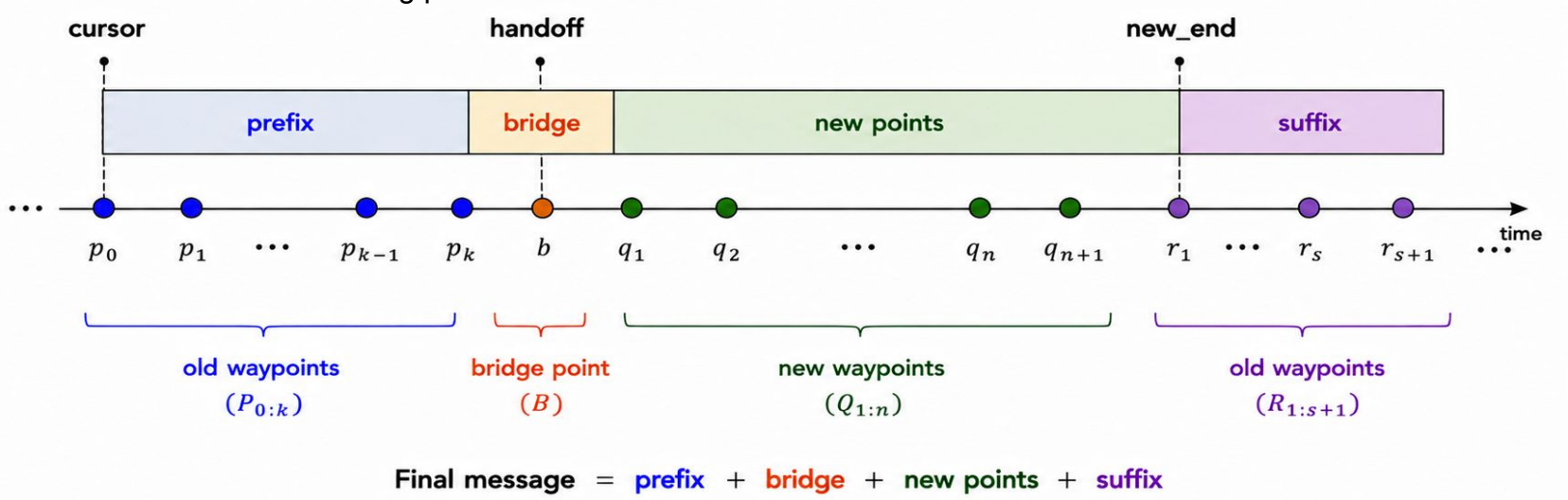
Trajectory replacement with full blending

Why do we need it?

1. AI policies emit fast chunks; replan every 50 to 500 ms.
2. Hard swapping causes jarring jumps / discontinuity in position / velocity.
3. Use case: Preserves prior obstacle avoidance plans for a specific joint.
4. Resolves old ROS 1 remaining port.

What we implemented:

1. Merges new trajectories in place with the old one.
2. Analytic bridge point retains current velocity.
3. Omitted joints complete original motion.
4. Re-anchored for accurate speed scaling.



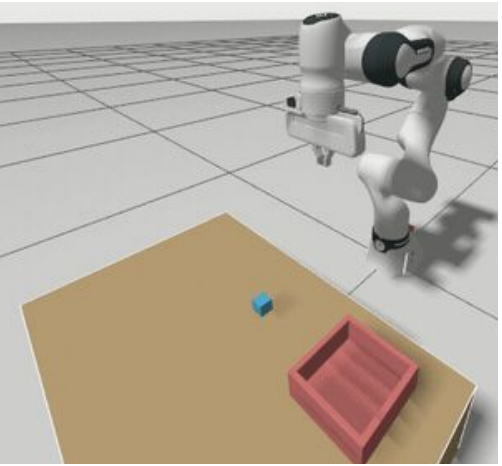
Positions-only action chunks upsampling in JTC

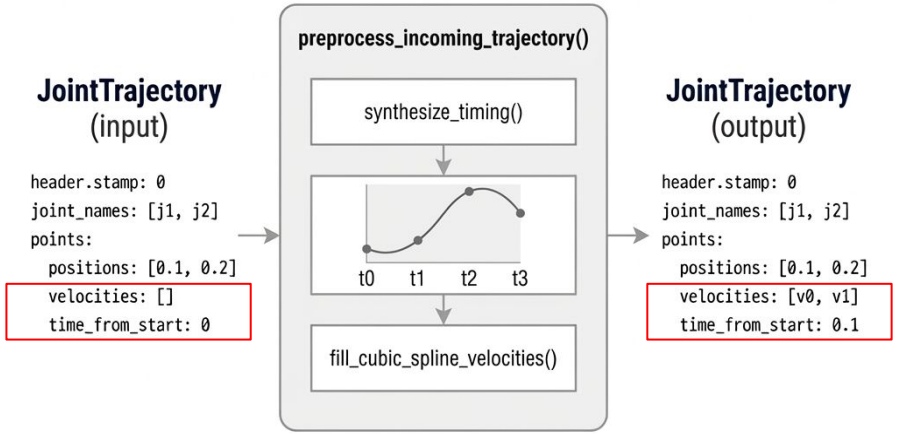
The Challenge

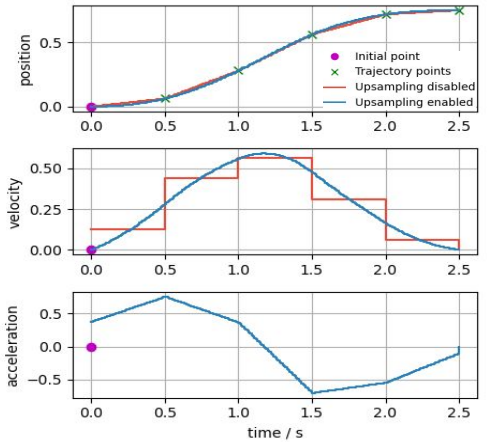
- AI policies yield low-frequency, position-only waypoints (e.g., 50Hz).
- The hardware expects a fresh command every control cycle at 1000 Hz
- Raw execution causes severe velocity steps.
- Infinite acceleration spikes damage real hardware.

The Solution: Upsampling

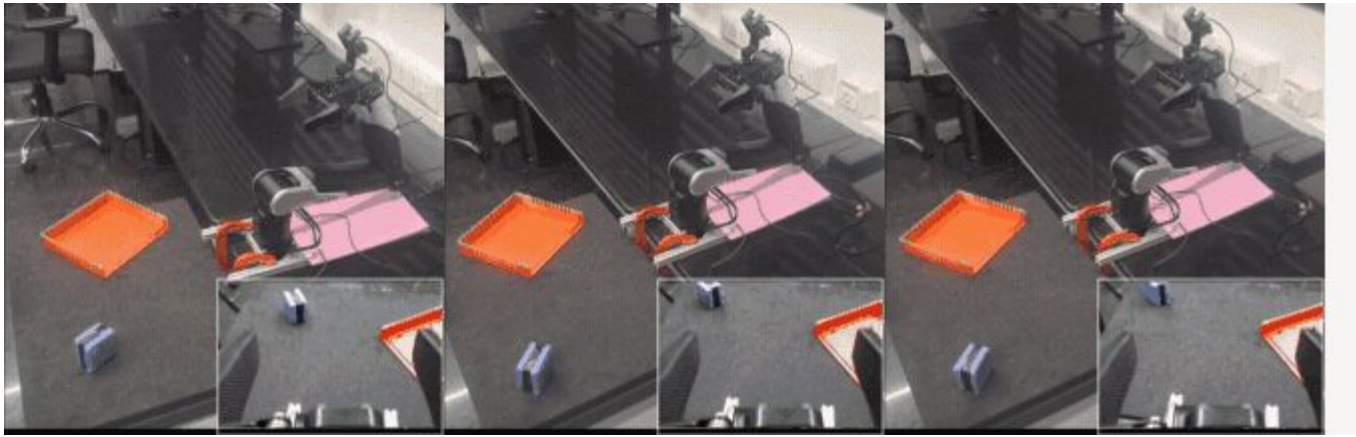
- Implements an O(n) tridiagonal velocity solver.
- Generates continuous cubic-spline trajectories.
- Eliminates discontinuities for smooth acceleration.
- Outputs safe, high-frequency commands (up to 2kHz).

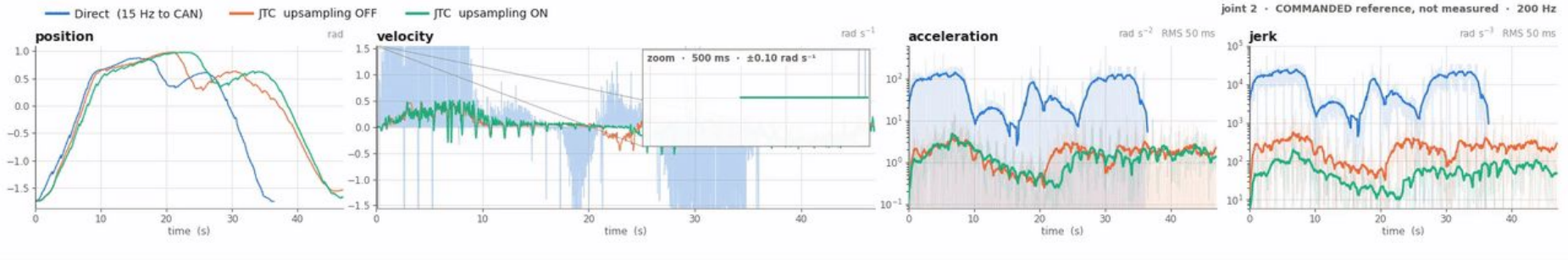






direct upsampling off upsampling on

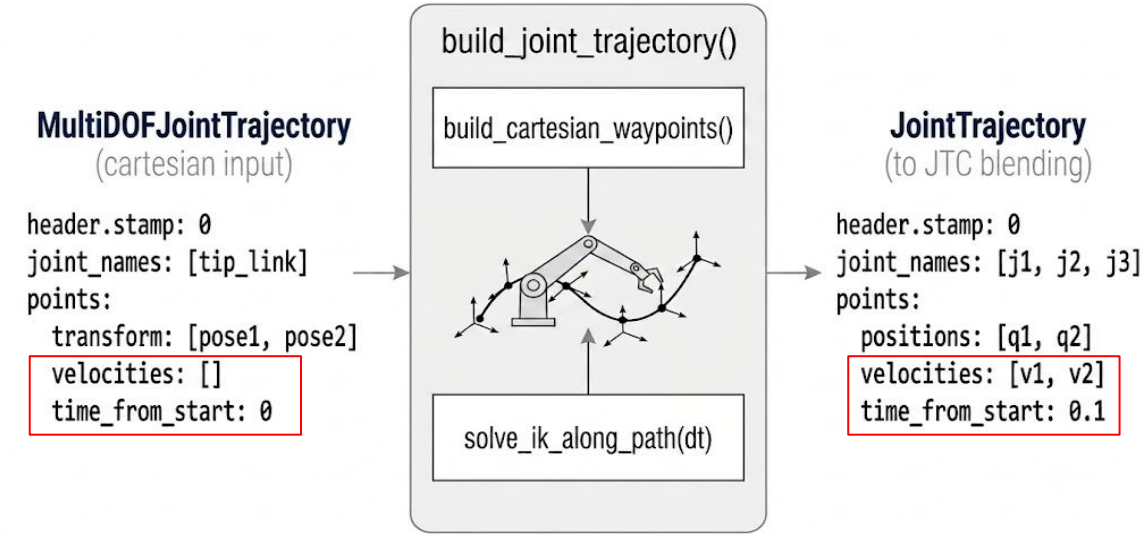


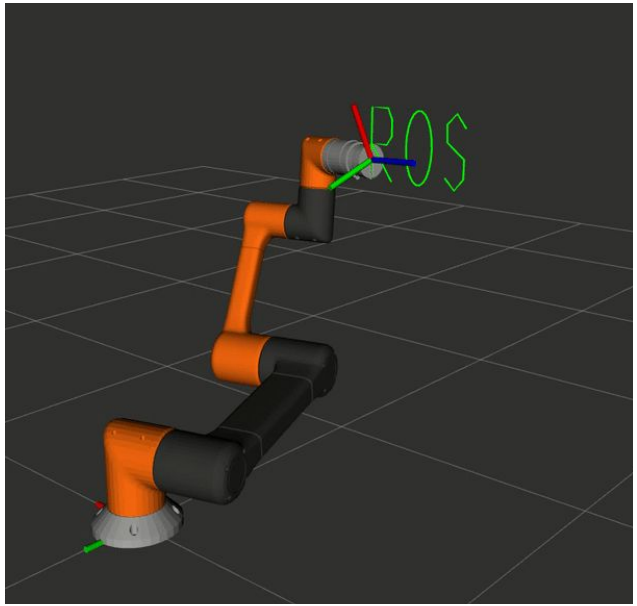


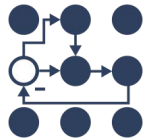
Cartesian trajectory controller

Joint-space interpolation does not move the tool along a straight line, and several policy families (diffusion policies, OpenVLA, RT-2) emit end-effector poses rather than joint angles.

- **Input:** Multi-DOF trajectories on `~/cartesian_reference`
- **Path interpolation:** Cubic Hermite (translation) + SLERP (rotation)
- **IK:** Differential Jacobian-based, default 100 Hz sampling
- **Output:** Joint trajectory passed on to base controller JTC



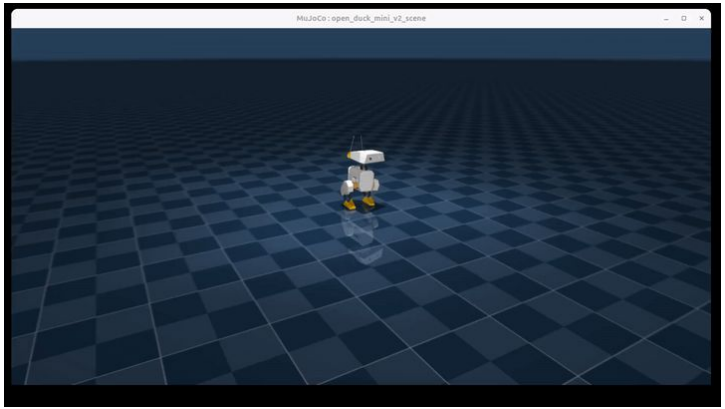




RL Deployment

The two ways to integrate

Policy in-controller, as async controller



Policy outside + ros2_controller

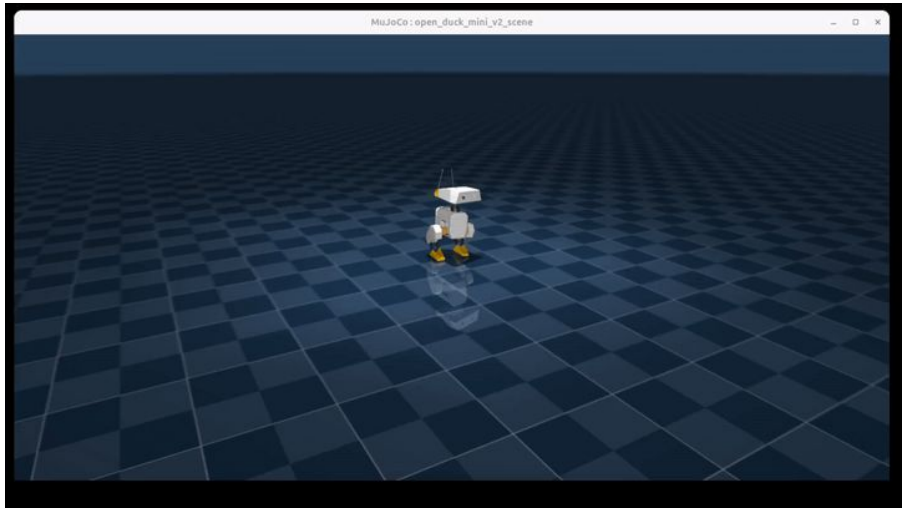
Example 18 - Policy in-controller

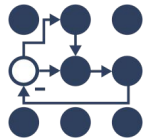
What: first ros2_control demo running a learned (ONNX) locomotion policy.

A Open Duck Mini walking in MuJoCo.

How: ONNX policy controller at 50 Hz taking velocity + head commands, custom DuckMiniMujocoSystemInterface exposing foot-contact sensing, state_interfaces_broadcaster publishing IMU, joint states, contacts

Why: shows RL policies as ordinary ros2_control controllers with the same lifecycle, interfaces, and launch tooling as classic controllers.





Hands-on: Inference Integration

non-RT layer

RT layer

ROS 2 Node

ros2_control layer

/controller_manager/introspection_data/names
/controller_manager/introspection_data/values

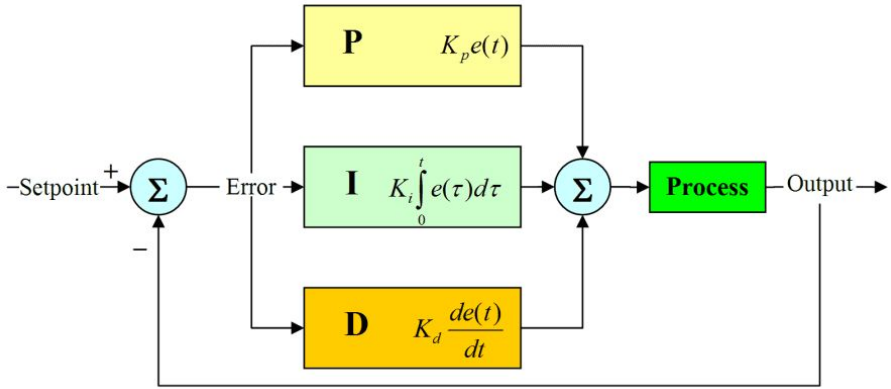
ONNX policy executor (pal_policy_deployer)

PID controller

/cmd_vel

Why pid_controller?

- Compliance is very important when dealing with contacts
- Easy to tune and sync between sim and hardware
- What we need is a PD controller (I gain to zero)
- Works with effort/current control mode of the actuator
- Chosen due to the transparency of the actuation



mjlab Documentation

Search... Ctrl + K

Version main

User Guide

- Installation Guide
- Tutorials
- Contributing

Concepts

- Architecture Overview
- Entity
- Actuators
- Sensors
- Scene
- Terrain

The Manager Layer

- Environment Configuration
- Observations
- Actions

Welcome to mjlab!

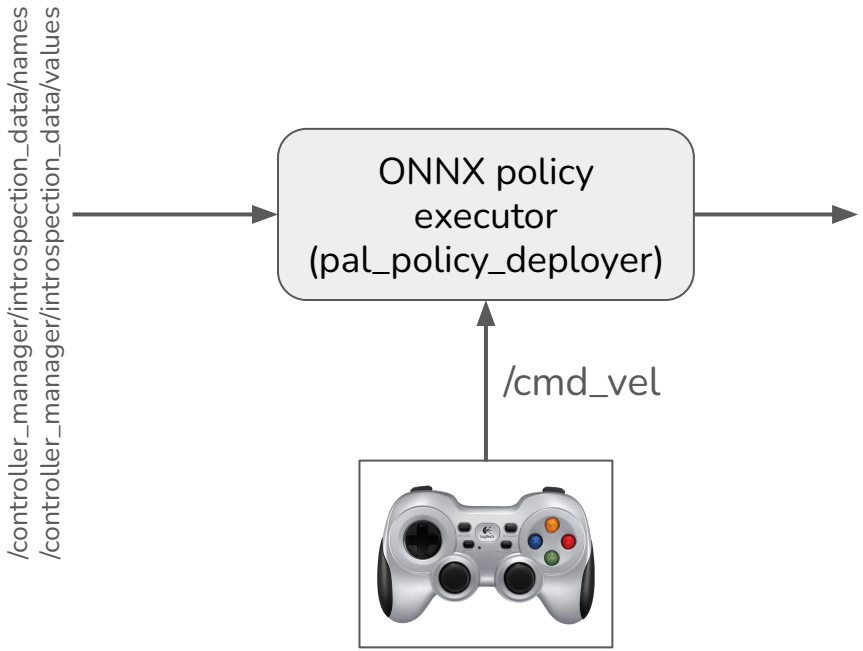
mjlab is a lightweight, open-source framework for robot learning that combines GPU-accelerated simulation with composable environments and minimal setup friction. It adopts the manager-based API introduced by Isaac Lab, where users compose modular building blocks for observations, rewards, and events, and pairs it with MuJoCo Warp for GPU-accelerated physics. The result is a framework installable with a single command, requiring minimal dependencies, and providing direct access to native MuJoCo data structures.

https://github.com/mujocolab/mjlab


ROUND 200
shuffling forward: 30 cm in 5 seconds

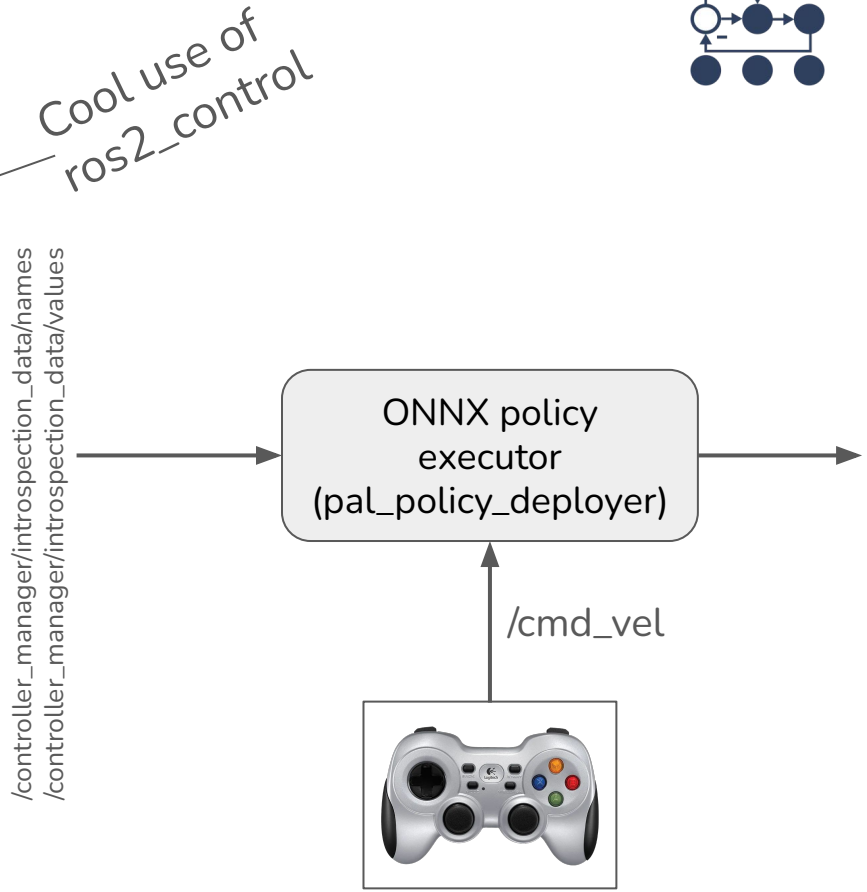
Observations

- Joint Positions
- Joint Velocities
- Projected Gravity Vector (Based on the IMU Orientation)
- IMU Angular Velocity
- IMU Linear Acceleration
- Command Twist (linear.x, linear.y and angular.z)
- Last action
- Observation history, if any



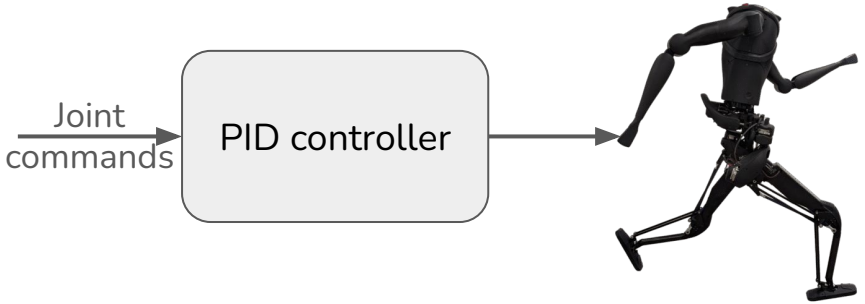
Observations

- Latency between different data should be minimal
- Better data coherence leads to better sim2real gap
- Multiple data sources vs Single source
- **Single source of information**
  - Controller manager introspection_data
  - state_interface_broadcaster

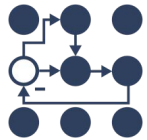


Actions

- Actions are output of the inference which are needed to be multiplied by the action scales, if any
- Every action is the amount of deviation from the default configuration
- Compliance is the key for robots dealing with the contact
- SySID to reduce the sim2real actuation gap



Actuation on the MJLAB end



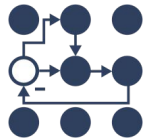
- The `BuiltinPositionActuator` of the mjlab is based on the position actuator of mjlab with stiffness and damping gain
- The physics engine computes the control law and integrates velocity-dependent damping forces implicitly.
- `BuiltinPositionActuator` puts kd on the `<position>` element and implicitly assumes a zero velocity reference

$$K(q_d - q) + D(\dot{q}_d - \dot{q}) = \tau_{ext}$$

q_d = Reference from policy

$\dot{q}_d$ = 0

RL details



Setup description:

- PPO Based
- Soft Actor-Critic Approach (Teacher-Student)
- Actor = Observation
- Critic = Privileged Observation
  - Base linear velocity
  - Noise-less observation
  - Contact Information
    - Feet air time
    - Contact forces

Final results:

After **30k** iterations

With **4096** environments

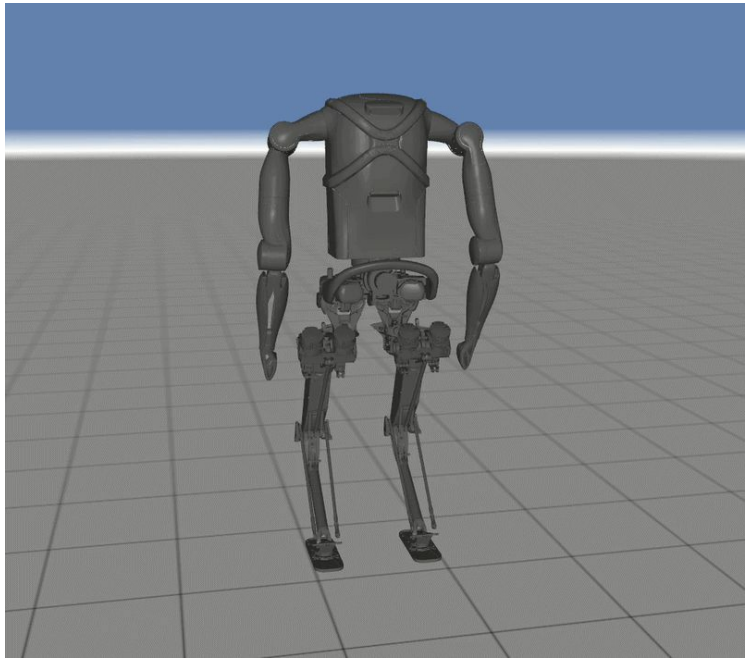
Sim timestep: **0.005**

Takes around **4-6 hours** on Nvidia RTX 5090

Live demo of  starting up training

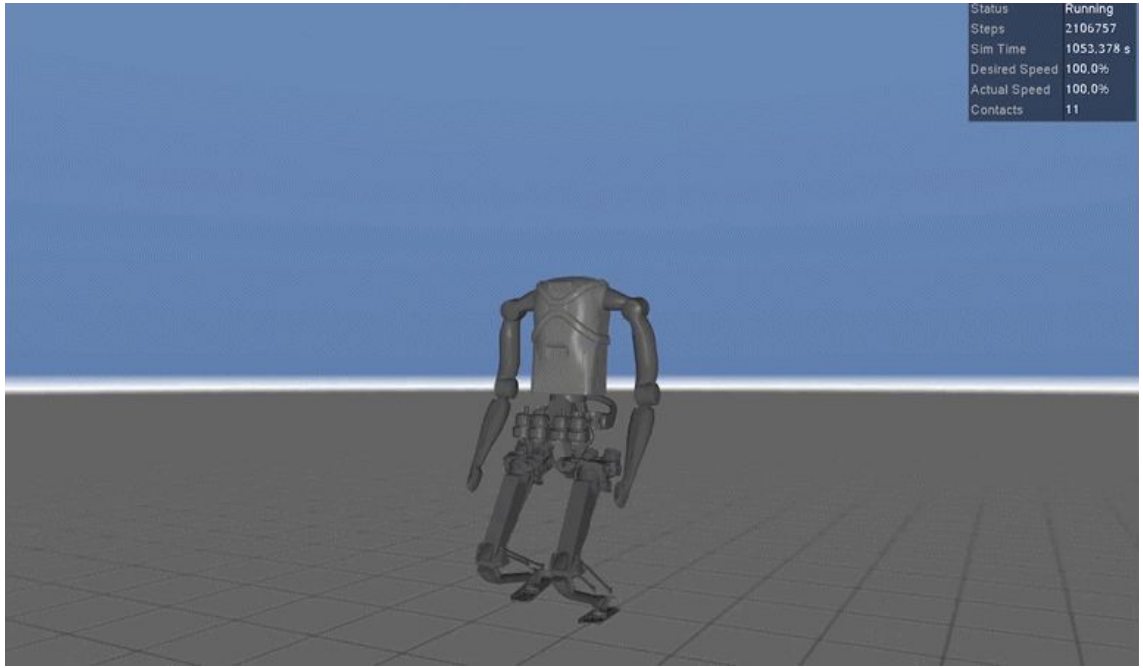
Hands-on: mujoco_ros2_control

```
ros2 launch kangaroo_mujoco kangaroo_mujoco.launch.py
```

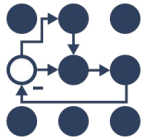


Hands-on: pal_policy_deployer

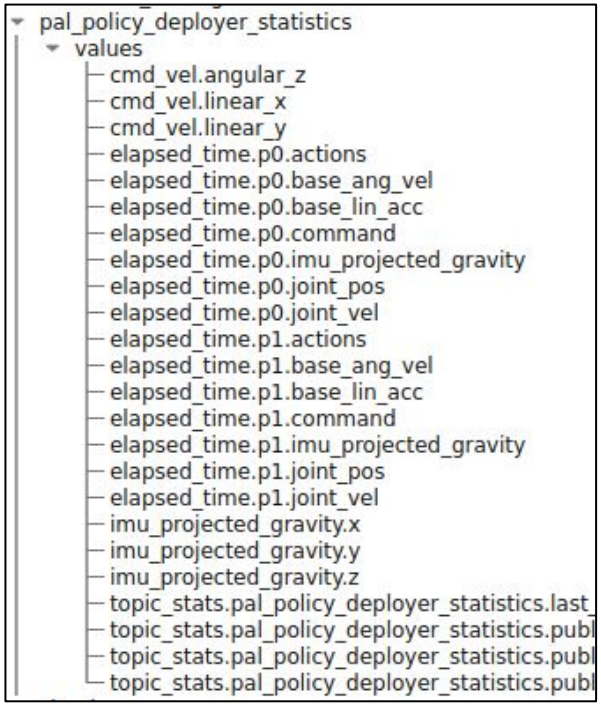
```
ros2 launch pal_policy_deployer kang_mjlab_policy_deployer.launch.py
```

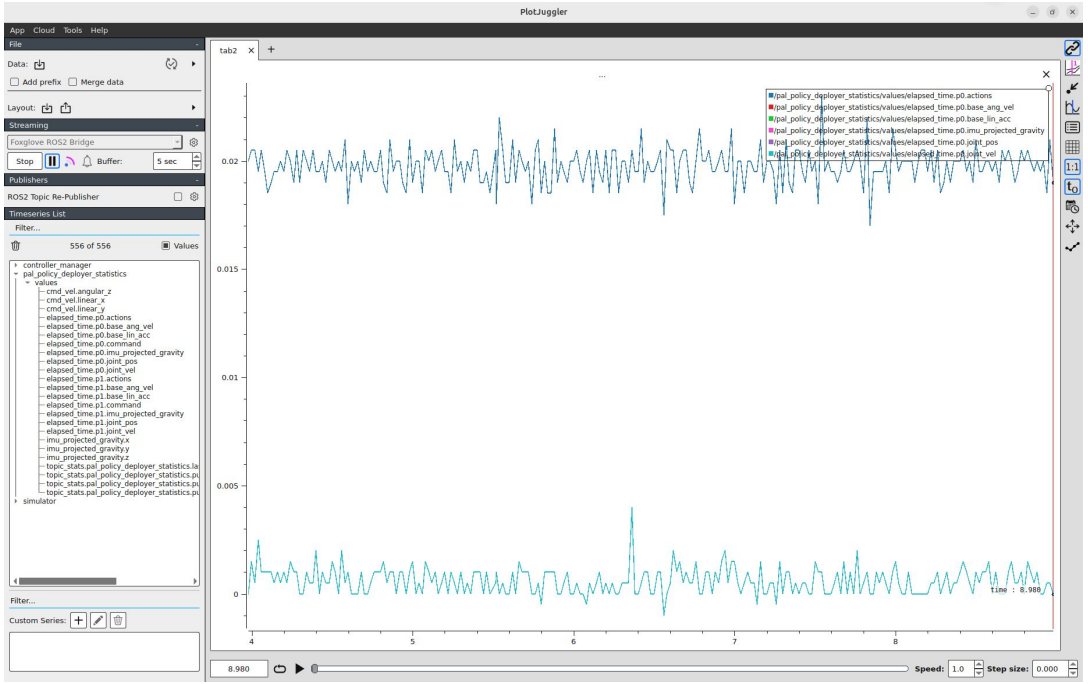


Hands-on: pal_policy_deployer stats



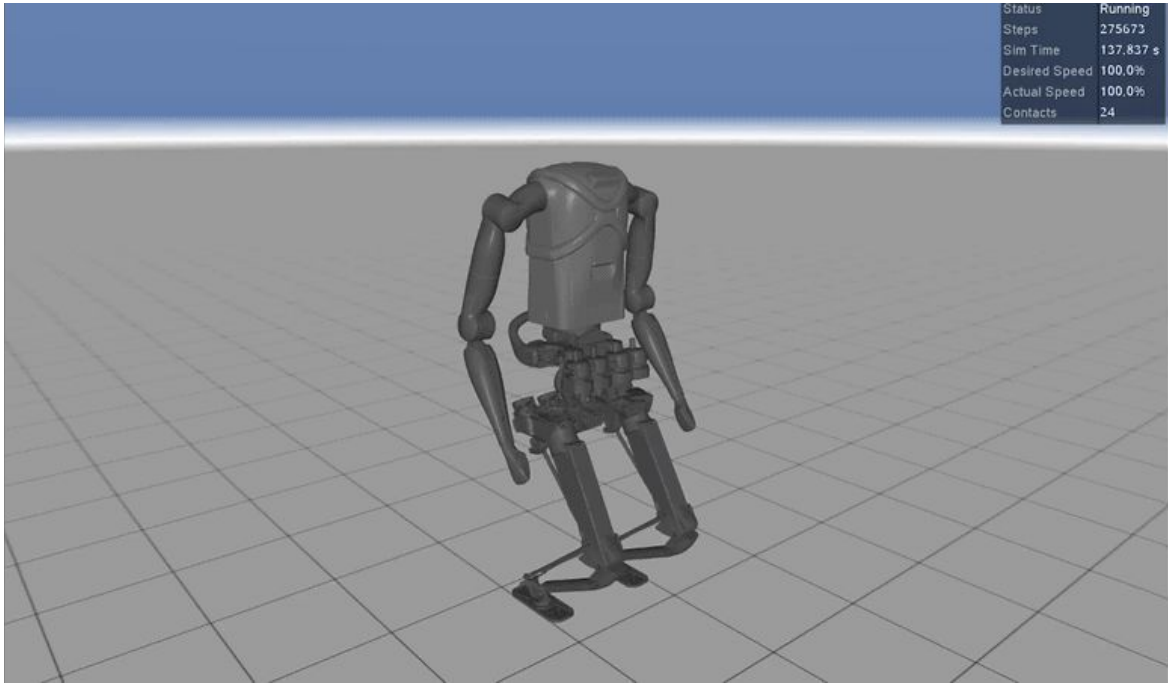
ros2 run plotjuggler plotjuggler

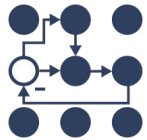




Hands-on: kangaroo mujoco

Push the robot and see if it can sustain

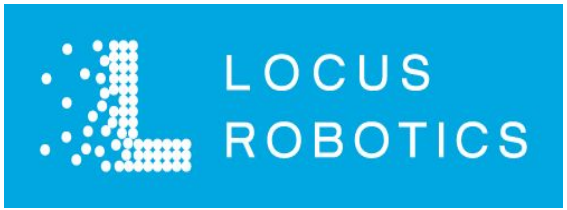




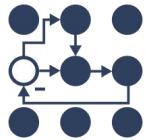
Closing Remarks

Please provide us feedback!

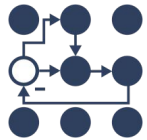
https://tinyurl.com/ros-controls-feedback-2026



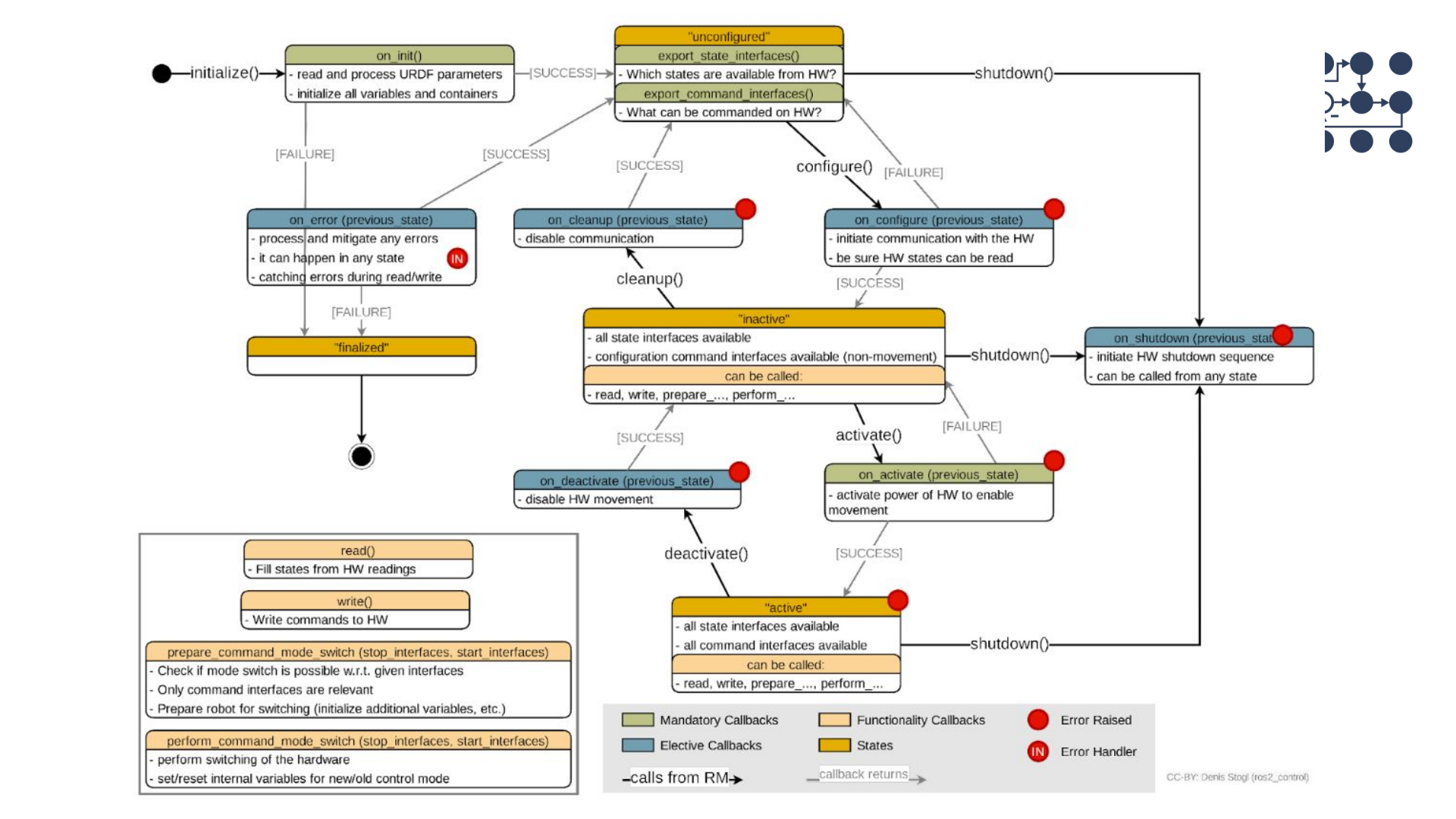
Thank you for being here!!



Christoph Fröhlich, Sai Kishor Kothakota, Marq Rasmussen, Bence Magyar, Denis Štogl, Erik Holum, Alejandro Hernández Cordero, Eneji Peacemaker Ohieku, Shahazad Abdullah, Kamalkant Thangaraju, Vedhas Talnikar, Ishan Pathak, Nikola Banović, Sahil Lakhmani, Christian Rauch, M. Ege Kural, Junius Santoso, Nathan Dunkelberger, Julia Jia, Soham Patil, Dennis Lanov, Sebastian Castro, Suryansh Singh, Tobias Fischer, Shlok Mehndiratta, José Luis Pérez Martín, Sergi de las Muelas, Krzysztof Pochwała, Shivam Maurya, Naitik Pahwa, Michele Cereda, Abdullah Theonewhomadethings, Felix Exner, Souri Rishik, Noel Jiménez García, Dhruv Patel ..... and many more!



Backup Slides


initialize()
on_init()
- read and process URDF parameters
- initialize all variables and containers
[SUCCESS]
"unconfigured"
export_state_interfaces()
- Which states are available from HW?
export_command_interfaces()
- What can be commanded on HW?
shutdown()
[FAILURE]
[SUCCESS]
[SUCCESS]
configure()
[FAILURE]
on_error (previous_state)
- process and mitigate any errors
- it can happen in any state
- catching errors during read/write
IN
on_cleanup (previous_state)
- disable communication
on_configure (previous_state)
- initiate communication with the HW
- be sure HW states can be read
cleanup()
[SUCCESS]
[FAILURE]
"finalized"
"inactive"
- all state interfaces available
- configuration command interfaces available (non-movement)
can be called:
- read, write, prepare_..., perform_...
shutdown()
on_shutdown (previous_state)
- initiate HW shutdown sequence
- can be called from any state
[SUCCESS]
activate()
[FAILURE]
on_deactivate (previous_state)
- disable HW movement
on_activate (previous_state)
- activate power of HW to enable movement
deactivate()
[SUCCESS]
"active"
- all state interfaces available
- all command interfaces available
can be called:
- read, write, prepare_..., perform_...
shutdown()
read()
- Fill states from HW readings
write()
- Write commands to HW
prepare_command_mode_switch (stop_interfaces, start_interfaces)
- Check if mode switch is possible w.r.t. given interfaces
- Only command interfaces are relevant
- Prepare robot for switching (initialize additional variables, etc.)
perform_command_mode_switch (stop_interfaces, start_interfaces)
- perform switching of the hardware
- set/reset internal variables for new/old control mode
Mandatory Callbacks
Elective Callbacks
Functionality Callbacks
States
Error Raised
IN
Error Handler
calls from RM
callback returns
CC-BY: Denis Stogl (ros2_control)