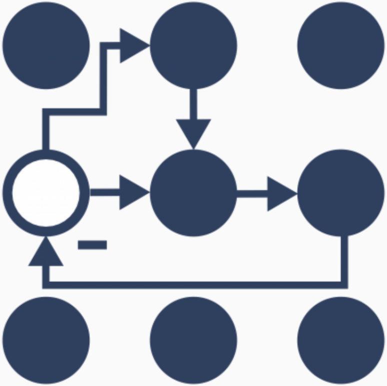
Using Ros2 Control

Driving robot motion with ROS2_Control

Jan van Hulzen

October 9, 2025

Overview



Assignment 1: Installation

- Check if Rviz2 is installed:

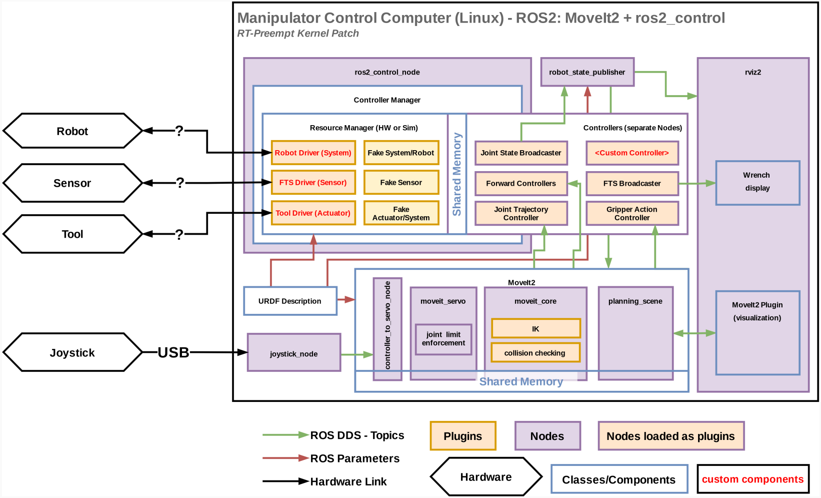
```
ros2 run rviz2 rviz2
```

- Install moveit2:
  https://moveit.ai/install-moveit2/binary/
- Check installation:

```
ros2 launch moveit2_tutorials demo.launch.py
```

- Bore-out preventer: https://moveit.picknik.ai/main/doc/tutorials/quickstart_in_rviz/quickstart_in_rviz_tutorial.html#getting-started

Overview software architecture ROS2 Control



Overview software architecture ROS2 Control

- Robot control is based on Moveit2+ros2_control
- Ros2_control nodes require configuration from .YAML files
    - Controller Manager
    - Resource Manager
    - Controllers (Joint State Breadcaster/Joint Trajectory Controller)
- Moveit planners require configuration
    - URDF
    - Moveit setup assistant to generate SRDF and .YAML files

Assignment 2: Installation

- install ros2_control if it is not already installed:

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

- Ros2_control information: https://control.ros.org/jazzy/doc/getting_started/getting_started.html
- Bore-out preventer: https://control.ros.org/jazzy/doc/ros2_control_demos/doc/index.html

Assignment 3: Create package

- For this lesson we will use the already existing ramps16_ws downloadable from moodle.

```
cd ~
mkdir -p ramps16_ws/src
unzip ramps16_ws.zip -d ~/ramps16_ws/src
cd ramps16_ws
code .
```

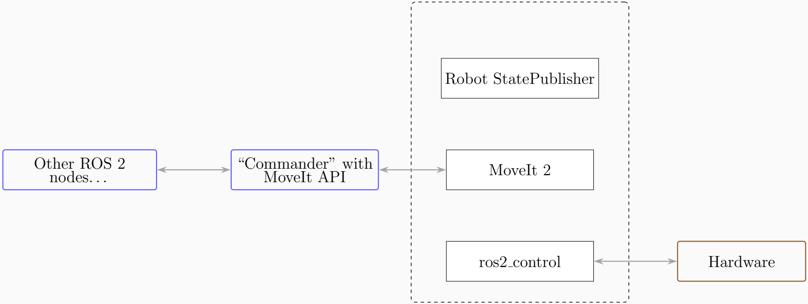
- Examine package

Interfacing with the hardware

Custom ROS2 Controller and Robot_driver

- After planning is completed the plan is executed by a ros2_control node by a custom controller plugin.
- The robot driver is a part of the resource manager and is an interface between the ros2_control plugin and hardware outside the OS and the microprocessor ROS2 runs on (e.g. linux/intel I5/I7).
- Tipically, the robot driver is a part ROS2 and part pure C++ and communicates with real time hardware, running on a microcontroller (e.g. Arduino/Mbed Nucleo)
- The C++ code for the microcontroller driving the steppers and reading encoders or limit switches, (i.e. Arduino/ Raspberry PI/Nucleo-Mbed).

The hardware interface



- The hardware interface that will be a class that is converted into a plugin.
- Develop a reusable C++ Moveit commander template.

The ROS2_control controller manager

- The Controller Manager is the main component of ROS2_control and manages the lifecycle of controllers and access to the hardware interfaces.
- The ros_parameters: are
  - update_rate (int)
  - <controller_name> .type, is the name of a plugin exported using pluginlib for a controller and is the class from which controller's instance with name "controller_name" is created.
- Use a .yaml file like my_robot_controllers.yaml.
- See also `https://control.ros.org/jazzy/doc/ros2_control/controller_manager/doc/userdoc.html`

The file my_robot_controllers.yaml

```yaml
controller_manager:
 ros__parameters:
  update_rate: 10

 joint_state_broadcaster:
   type: joint_state_broadcaster/JointStateBroadcaster

 arm_joints_controller:
 type:
     forward_command_controller/ForwardCommandController

arm_joints_controller:
 ros__parameters:
  joints:["axis_1","axis_2","axis_3","axis_4","axis_5"]
  interface_name: "velocity"
```

The test

- Source workspace and start bringup:

```
ros2 launch my_robot_bringup my_robot.launch.xml
```

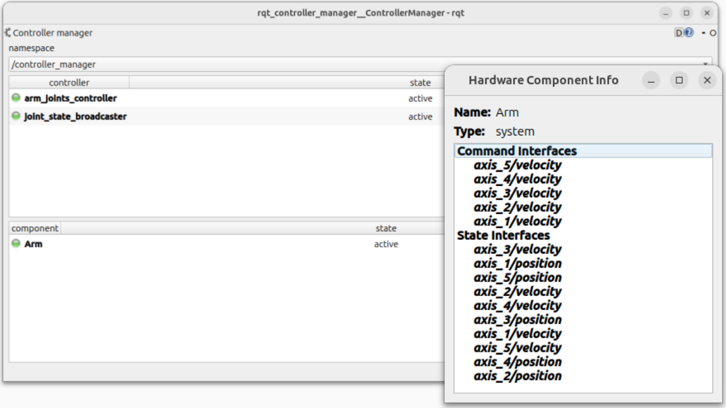
- In a second terminal run:

```
ros2 topic pub -1 /arm_joints_controller/commands
    std_msgs/msg/Float64MultiArray "{data:[10.0,
    20.0,30.0,40.0,50.0]}"
```

- in a third terminal run

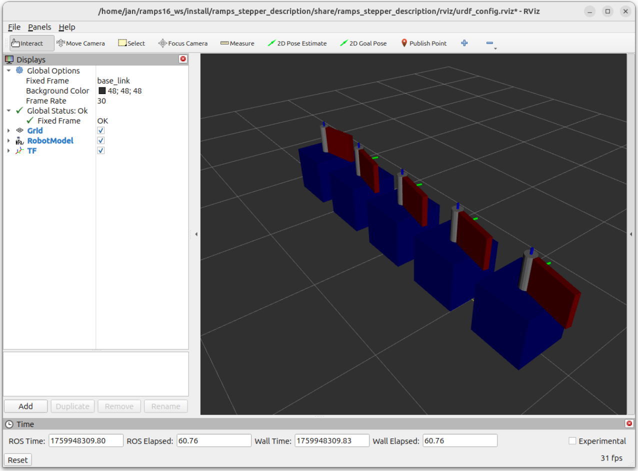
```
sudo apt install ros-jazzy-controller-manager
ros2 run rqt_controller_manager
    rqt_controller_manager
```

Assignment 4: run rqt_controller_manager



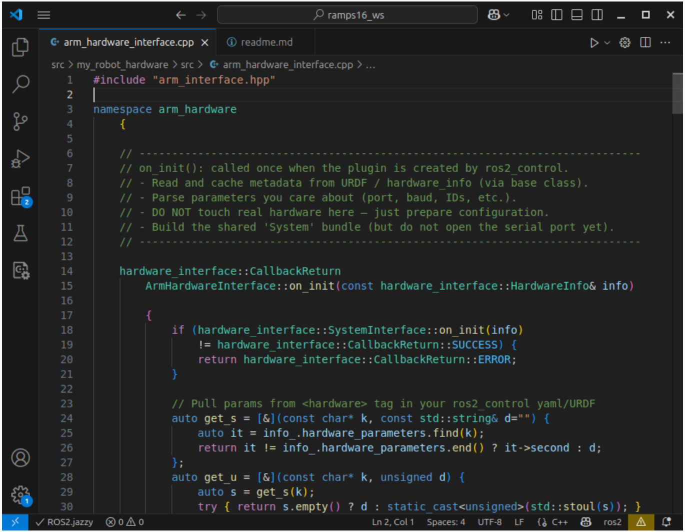
- use rightclick and dubbleclick to activate/deactivate or obtain additional info.

Assignment 4: run rqt_controller_manager



- In this case five steppers are controlled

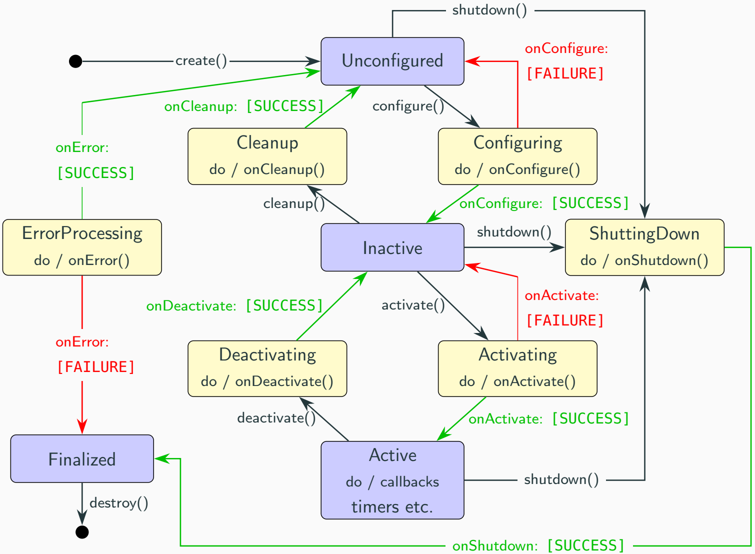
Assignment 4: explore arm_hardware_interface.cpp



```cpp
#include "arm_interface.hpp"

namespace arm_hardware
    {

    // ----------------------------------------------------------------------------
    // on_init(): called once when the plugin is created by ros2_control.
    // - Read and cache metadata from URDF / hardware_info (via base class).
    // - Parse parameters you care about (port, baud, IDs, etc.).
    // - DO NOT touch real hardware here – just prepare configuration.
    // - Build the shared 'System' bundle (but do not open the serial port yet).
    // ----------------------------------------------------------------------------

    hardware_interface::CallbackReturn
        ArmHardwareInterface::on_init(const hardware_interface::HardwareInfo& info)

        {
            if (hardware_interface::SystemInterface::on_init(info)
                != hardware_interface::CallbackReturn::SUCCESS) {
                return hardware_interface::CallbackReturn::ERROR;
            }

            // Pull params from <hardware> tag in your ros2_control yaml/URDF
            auto get_s = [&](const char* k, const std::string& d="") {
                auto it = info_.hardware_parameters.find(k);
                return it != info_.hardware_parameters.end() ? it->second : d;
            };
            auto get_u = [&](const char* k, unsigned d) {
                auto s = get_s(k);
                try { return s.empty() ? d : static_cast<unsigned>(std::stoul(s)); }
```

- examine arm_hardware_interface.cpp

Lifecycle node



Data structure

- The `arm_hardware` namespace encapsulates the custom hardware interface implementation keeping all related classes and functions logically grouped and preventing name conflicts.
- The `ArmHardwareInterface` class defines the hardware abstraction layer used by ROS 2 control. It provides standardized methods for configuration, reading sensor data, and writing actuator commands.

```
#include "arm_interface.hpp"

namespace arm_hardware
   {
   } // namespace arm_hardware
```

Overrides

- the arm_interface.hpp file contains C++ overrides.
- **Why *override*?** The C++ `override` specifier guarantees that each declared function exactly matches a virtual function in the base class (`hardware_interface::SystemInterface` or lifecycle hooks). This enables correct dynamic dispatch and produces a compile-time error if the signature drifts, improving safety during ROS 2 and API upgrades.
- Explore arm_interface.hpp

Assignment 5: arm_interface.hpp

```
#ifndef ARM_HARDWARE_INTERFACE_HPP
#define ARM_HARDWARE_INTERFACE_HPP

#include "hardware_interface/system_interface.hpp"

#include <rclcpp/rclcpp.hpp>
#include "SerialStreamHelper.hpp"
#include "StepperClient.hpp"
#include "StepperDriver.hpp"

#include <chrono>
#include <cmath>
#include <memory>
#include <stdexcept>
#include <string>


...
#endif // arm_HARDWARE_INTERFACE_HPP
```

Assignment 5: arm_interface.hpp

```
namespace arm_hardware {
 class ArmHardwareInterface : public hardware_interface
     ::SystemInterface{
 public:
 // livecycle overrides
 hardware_interface::CallbackReturn
         on_configure(const rclcpp_lifecycle::State &
             previous_state) override;
 hardware_interface::CallbackReturn
         on_activate(const rclcpp_lifecycle::State &
             previous_state) override;
 hardware_interface::CallbackReturn
         on_deactivate(const rclcpp_lifecycle::State &
             previous_state) override;
...
} // namespace arm_hardware
#endif // arm_HARDWARE_INTERFACE_HPP
```

Assignment 5: arm_interface.hpp

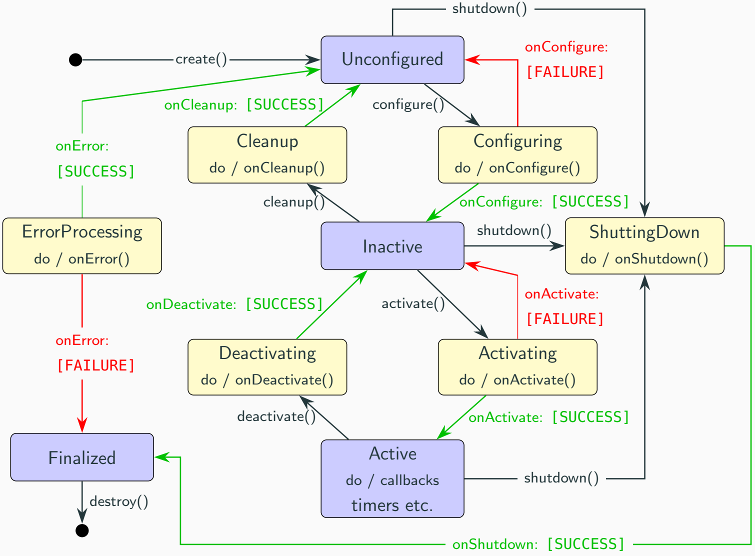
```
namespace arm_hardware {
 class ArmHardwareInterface : public hardware_interface
     ::SystemInterface{
 public:
...
  // system interface overrides
  hardware_interface::CallbackReturn
         on_init(const hardware_interface::HardwareInfo
             & info) override;
  hardware_interface::return_type
         read(const rclcpp::Time & time, const rclcpp::
             Duration & period) override;
  hardware_interface::return_type
         write(const rclcpp::Time & time, const rclcpp::
             Duration & period) override;
...
} // namespace arm_hardware
#endif // arm_HARDWARE_INTERFACE_HPP
```

Assignment 5: arm_interface.hpp

```cpp
namespace arm_hardware {
 class ArmHardwareInterface : public hardware_interface
     ::SystemInterface{
  ...
  private:
   std::shared_ptr<SerialStreamHelper> serial_;
   int motor_id_1_;
     ...
   std::string port_;
   int baudrate_;
   float steps_per_rev_;
   bool hex_dump_;
   bool probe_;
   std::string protocol_;
   // Our stepper system bundle (driver, client, etc.)
    std::shared_ptr<stepper::System> sys_;
    }; // class ArmInterface
} // namespace arm_hardware
#endif // arm_HARDWARE_INTERFACE_HPP
```

ROS2 lifecycle node, the interface with the hardware driver

Lifecycle node: on_init() & on_configure()



Assignment 6: explore the function on_init() 1/6

- The `on_init()` function is called automatically when the lifecycle node object is constructed, before the node enters any managed state.
- It allows setup operations that must occur before the lifecycle state machine starts — such as allocating memory, declaring parameters, or setting up internal variables.
- Unlike callbacks such as `on_configure()` or `on_activate()`, the node is not yet in any lifecycle state when `on_init()` runs.
- The function returns a `CallbackReturn` (typically `SUCCESS` or `ERROR`).
- Use `on_init()` for lightweight, non-hardware setup (logging setup, etc) — heavier resource initialization should be done by `on_configure()`.

Assignment 6: explore the function on_init() 2/6

```
hardware_interface::CallbackReturn
   ArmHardwareInterface::on_init(const
       hardware_interface::HardwareInfo& info){

 if (hardware_interface::SystemInterface::on_init(info)
   != hardware_interface::CallbackReturn::SUCCESS) {
 return hardware_interface::CallbackReturn::ERROR;
}

// Pull params from <hardware> in ros2_control yaml/URDF
auto get_s = [&](const char* k,const std::string& d=""){
auto it = info_.hardware_parameters.find(k);
return it != info_.hardware_parameters.end() ? it->
    second : d;
};
```

Assignment 6: explore the function on_init() 3/6

```
hardware_interface::CallbackReturn
   ArmHardwareInterface::on_init(const
       hardware_interface::HardwareInfo& info){

 if (hardware_interface::SystemInterface::on_init(info)
   != hardware_interface::CallbackReturn::SUCCESS) {
 return hardware_interface::CallbackReturn::ERROR;
}

// Pull params from <hardware> in ros2_control yaml/URDF
auto get_s = [&](const char* k,const std::string& d=""){
auto it = info_.hardware_parameters.find(k);
return it != info_.hardware_parameters.end() ? it->
    second : d;
};
```

Assignment 6: explore the function on_init() 4/6

```
auto get_u = [&](const char* k, unsigned d) {
  auto s = get_s(k);
  try { return s.empty() ? d : static_cast <unsigned> (
      std::stoul(s)); }
  catch (...) { return d; }
};

auto get_d = [&](const char* k, double dflt) {
  auto s = get_s(k);
  try { return s.empty() ? dflt : std::stod(s); }
  catch (...) { return dflt; }
};

auto get_b = [&](const char* k, bool dflt) {
  auto s = get_s(k);
  if (s == "1"||s == "true" ||s == "True") return true;
  if (s == "0"||s == "false"||s == "False")return false;
  return dflt;
};
```

Assignment 6: explore the function on_init() 5/6

```
const std::string port = get_s("port", "/dev/ttyACM0");
const unsigned baud = get_u("baudrate", 57600);
const uint8_t
      id1 = static_cast<uint8_t>(get_u("motor_id_1",1));
const uint8_t
      id2 = static_cast<uint8_t>(get_u("motor_id_2",2));
const uint8_t
      id3 = static_cast<uint8_t>(get_u("motor_id_3",3));
const uint8_t
      id4 = static_cast<uint8_t>(get_u("motor_id_4",4));
const uint8_t
      id5 = static_cast<uint8_t>(get_u("motor_id_5",5));
const double spr = get_d("steps_per_rev", 200.0);
const bool hex_dump = get_b("hex_dump", false);
const bool probe = get_b("probe", false);
const std::string proto = get_s("protocol", "register");
```

Assignment 6: explore the function on_init() 6/6

```
// Build the config-only bundle
  sys_ = std::make_shared<stepper::System>(
  stepper::StepperDriver::Prepare(
  port, baud, id1, id2, id3, id4, id5, spr, hex_dump,
      probe, proto));

RCLCPP_INFO(get_logger(),
"Prepared Stepper System: device='%s' baud=%u id1=%u id2
    =%u id3=%u id4=%u id5=%u spr=%.1f hex=%d probe=%d
    proto=%s",
  sys_->device.c_str(), sys_->baud,
  sys_->motor_id_1, sys_->motor_id_2,
  sys_->motor_id_3, sys_->motor_id_4,
  sys_->motor_id_5, sys_->steps_per_rev,
  sys_->hex_dump, sys_->probe, sys_->protocol.c_str());
  return hardware_interface::CallbackReturn::SUCCESS;
}
...
```

Assignment 7: explore the function on_configure() 1/5

- The `on_configure()` function is called when the node transitions from the `unconfigured` to the `inactive` state.
- It is used to initialize or allocate required resources such as publishers, subscribers, services, or action servers.
- At this stage, the node exists and has parameters available, but it is not yet performing its main activities (e.g., publishing or processing data).
- The function returns a `CallbackReturn` (either `SUCCESS` or `FAILURE`), which determines whether the transition to the `inactive` state succeeds.
- Use `on_configure()` for hardware setup, topic creation, and parameter-dependent initialization — essentially preparing the node to be safely activated.

Assignment 7: explore the function on_configure() 2/5

```
...
hardware_interface::CallbackReturn
  ArmHardwareInterface::on_configure(const
                            rclcpp_lifecycle::State&){
if (!sys_) {
   RCLCPP_ERROR(get_logger(), "System bundle (sys_) is
       null in on_configure()");
   return hardware_interface::CallbackReturn::ERROR;
}
if (sys_->device.empty()) {
   RCLCPP_ERROR(get_logger(), "Serial device parameter
       is empty");
   return hardware_interface::CallbackReturn::ERROR;
}
...
```

Assignment 7: explore the function on_configure() 3/5

```
...
try { stepper::StepperDriver::Init(*sys_); }
  catch (const std::exception& e) {
  RCLCPP_ERROR(get_logger(), "StepperDriver::Init failed:
       %s", e.what());
    return hardware_interface::CallbackReturn::ERROR;
}

RCLCPP_INFO(get_logger(),"Stepper system initialized.");
return hardware_interface::CallbackReturn::SUCCESS;
}
...
```

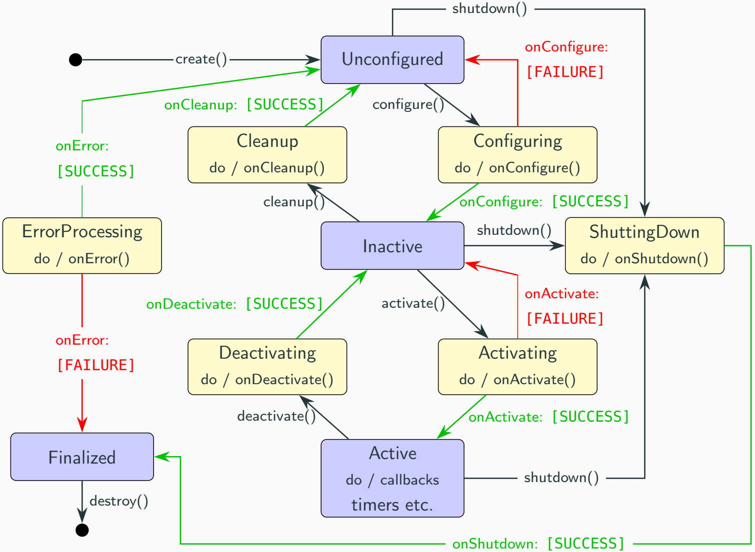
Assignment 7: explore the function Init() 4/5

- `on_configure()` callback calls the `Init()` function to establish low-level communication with external hardware (e.g., an Arduino over a serial link). `on_configure()` is responsible for hardware bring-up and validation.
- The `Init()` function first verifies that the serial device path is valid, then opens a serial connection using a `SerialStreamHelper` object configured with the specified baud rate.
- After opening the port, it waits briefly (≈ 1.8 s) to allow the microcontroller to reset and become ready. Optionally, it scans for a "READY" banner to confirm the device is responsive, which aids in diagnostics during bring-up.

Assignment 7: explore the function Init() 4/5

- Once the serial link is verified, `Init()` constructs a `StepperClient` and a `Stepper Driver`, configuring retry policies, timeouts, and protocol mode (`compact` or `register`). This effectively initializes the hardware communication stack.
- Placing `Init()` inside `on_configure()` aligns with ROS 2 lifecycle design: resource-heavy hardware setup is performed only when entering the `inactive` state, allowing safe teardown in `on_cleanup()` or reconfiguration without restarting the node.
- Explore StepperDriver.hpp, StepperClient.hpp and SerialStreamHelper.hpp which are included and compiled when arm_hardware_interface.cpp is build.

Lifecycle node: on_activate()



Assignment 8: explore the function on_activate 1/3

- The `on_activate()` function is called when the node transitions from the `inactive` state to the `active` state, marking the start of its operational phase.
- `on_activate()` is used to enable runtime behavior such as publishing messages, processing data, or interacting with hardware components that were set up during configure.
- At this point, all necessary resources already exist and are properly configured; `on_activate()` simply "turns them on" for active use.
- The function returns a `CallbackReturn` (typically `SUCCESS` or `FAILURE`) to indicate whether the activation succeeded and the node can begin normal operation.
- Use `on_activate()` to activate publishers, start timers, or enable control loops—ensuring the node begins its main tasks in a well-defined, recoverable state.

Assignment 8: explore the function on_activate 2/3

```
hardware_interface::CallbackReturn
   ArmHardwareInterface::on_activate(const
                        rclcpp_lifecycle::State&){
if (!sys_ || !sys_->ready() || !sys_->driver) {
  RCLCPP_ERROR(get_logger(), "Driver not ready in
                                   on_activate()");
  return hardware_interface::CallbackReturn::ERROR;
}
auto& drv = *sys_->driver;
if (drv.activateWithVelocityMode(sys_->motor_id_1) != 0)
  return hardware_interface::CallbackReturn::ERROR;
          ...
if (drv.activateWithVelocityMode(sys_->motor_id_5) != 0)
  return hardware_interface::CallbackReturn::ERROR;
  ...
```

Assignment 8:explore the function on_activate 3/3

```
 ...
 set_state("axis_1/velocity",0.0);
          ...
 set_state("axis_5/velocity",0.0);
 set_state("axis_1/position",0.0);
          ...
 set_state("axis_5/position",0.0);
return hardware_interface::CallbackReturn::SUCCESS;
 }
}
```

- Do not forget to set all states to 0.0 otherwise the TF subssytem will throw a fit.

Assignment 9: explore the function read() 1/4

- The `read()` function retrieves the most recent sensor data or hardware state from external devices and updates the node's internal state variables accordingly.
- It forms the input stage of the control cycle, typically called before the `update()` and `write()` functions in real-time controllers or hardware interfaces.
- During `read()`, raw data such as joint positions, velocities, or sensor values are collected, filtered if needed, and stored in shared state buffers accessible to the controller.
- The `read()` function is only executed while the node is in the `active` state.
- Implement `read()` to communicate with hardware drivers (e.g., serial, CAN, or Ethernet), synchronize state feedback, and maintain consistency between the physical system and the ROS 2 control framework.

Assignment 9: explore the function read() 2/4

```
hardware_interface::return_type ArmHardwareInterface::
    read
      (const rclcpp::Time & time, const rclcpp::Duration
          & period) {
 (void)time;
 auto& drv = *sys_->driver;
 const auto& cal = sys_->cal;
 // read the current velocity of both motors
 // and update the state variables accordingly
 // we assume that the velocity is constant
 // over the period so we can integrate the
 // position by pos = pos + vel * dt
 double vel_motor_1_ = drv.getVelocityRadianPerSec(
          sys_->motor_id_1, cal);
 int32_t p1 = drv.getPositionSteps(sys_->motor_id_1);
 }
```

Assignment 9: explore the function read() 3/4

```
hardware_interface::return_type ArmHardwareInterface::
    read
      (const rclcpp::Time & time, const rclcpp::Duration
          & period){
...
 double vel_motor_1_ = drv.getVelocityRadianPerSec(sys_
     ->motor_id_1, cal);
 int32_t p1 = drv.getPositionSteps(sys_->motor_id_1);
 ...
 double vel_motor_5_ = drv.getVelocityRadianPerSec(sys_
     ->motor_id_5, cal);
 int32_t p1 = drv.getPositionSteps(sys_->motor_id_1);

 if (abs(vel_motor_1_) < 0.03) { vel_motor_1_ = 0.0; }
 ...
 if (abs(vel_motor_5_) < 0.03) { vel_motor_5_ = 0.0; }
}
```

Assignment 9: explore the function read() 3/4

```
hardware_interface::return_type ArmHardwareInterface::
    read
      (const rclcpp::Time & time, const rclcpp::Duration
           & period){
...
 set_state("axis_1/velocity",vel_motor_1_);
 ...
 set_state("axis_1/position",get_state("axis_1/position"
     ) + vel_motor_1_* period.seconds());
 ...
 RCLCPP_INFO(get_logger(), "pos1=%.3f pos2=%.3f pos3=%.3
     f pos4=%.3f pos5=%.3f vel1=%.3f vel2=%.3f vel3=%.3f
      vel4=%.3f vel5=%.3f",rad1_, rad2_, rad3_, rad4_,
     rad5_, vel_motor_1_, vel_motor_2_,vel_motor_3_,
     vel_motor_4_,vel_motor_5_);
return hardware_interface::return_type::OK;
}
```

Assignment 10: explore the function write() 1/3

- The `write()` function sends command outputs from the node (or controller) to the external hardware, such as actuators, motors, or other devices.
- It forms the output stage of the control cycle, typically called after `read()` and `update()`, to apply computed control signals to the physical system.
- During `write()`, the latest command values—such as desired positions, velocities, or torques—are transmitted to the hardware interface or communication bus.
- The `write()` function is only executed when the node is in the `active` state, ensuring that actuator commands are sent only when the system is properly configured and safe to operate.
- Implement `write()` to push control data to drivers (e.g., via serial or CAN), ensuring real-time synchronization between the software controller and the physical hardware.

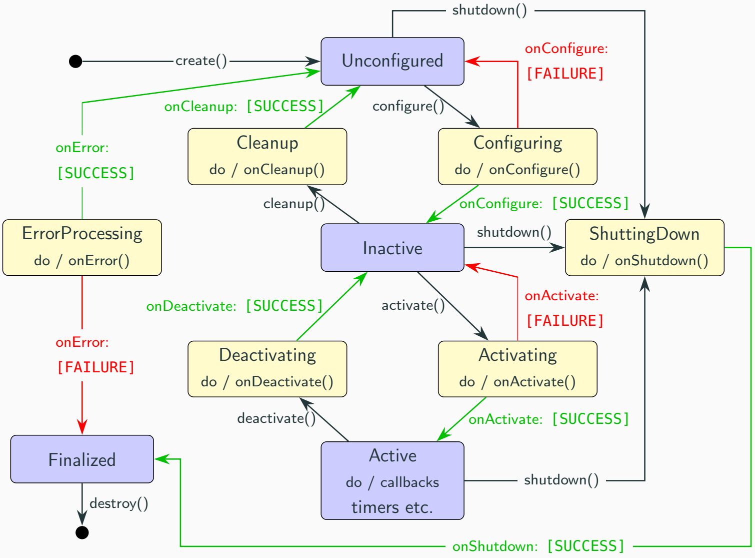
Assignment 10: explore the function write() 2/3

```
hardware_interface::return_type ArmHardwareInterface::
    write
     (const rclcpp::Time & time, const rclcpp::Duration
         & period){
 (void)time;
 (void)period;
 if (sys_->driver->setTargetVelocityRadianPerSec(sys_->
    motor_id_1, get_command("axis_1/velocity"), sys_->
    cal) != 0)
 return hardware_interface::return_type::ERROR;
          ...
 if (sys_->driver->setTargetVelocityRadianPerSec(sys_->
    motor_id_5, get_command("axis_5/velocity"), sys_->
    cal) != 0)
 return hardware_interface::return_type::ERROR;
...
}
```

Assignment 11: explore the function write() 3/3

```
hardware_interface::return_type ArmHardwareInterface::
    write
     (const rclcpp::Time & time, const rclcpp::Duration
         & period){
...
RCLCPP_INFO(get_logger(), "axis 1 vel: %lf, axis 2 vel:
     %lf, axis 3 vel: %lf, axis 4 vel: %lf axis 5 vel:
    %lf", get_command("axis_1/velocity"), get_command("
    axis_2/velocity"), get_command("axis_3/velocity"),
    get_command("axis_4/velocity"), get_command("axis_5
    /velocity"));
return hardware_interface::return_type::OK;
}
```

Lifecycle node: on_deactivate()



Assignment 12: explore the function on_deactivate 1/3

- The `on_deactivate()` function is called when the node transitions from the `active` state to the `inactive` state, signaling that runtime operations should stop safely.
- It is used to disable or pause ongoing activities such as publishing, control loops, or hardwareo cmmands without destroying the underlying resources created during `on_configure()`.
- During `on_deactivate()`, publishers and timers are typically deactivated and active control signals are set to a safe default state.
- The function returns a `CallbackReturn` (usually `SUCCESS` or `FAILURE`)
- Implement `on_deactivate()` to gracefully stop hardware actions, pause data streams, and ensure the system remains in a consistent, safe condition before reactivation or cleanup.

Assignment 12: explore the function on_deactivate 2/3

```
hardware_interface::CallbackReturn ArmHardwareInterface
    ::on_deactivate
(const rclcpp_lifecycle::State & previous_state) {
(void)previous_state;

// Stop
if (sys_->driver->setTargetVelocityRadianPerSec(sys_->
    motor_id_1, 0.0, sys_->cal) != 0)
   return hardware_interface::CallbackReturn::ERROR;
...
if (sys_->driver->setTargetVelocityRadianPerSec(sys_->
    motor_id_5, 0.0, sys_->cal) != 0)
   return hardware_interface::CallbackReturn::ERROR;

// wait a bit
std::this_thread::sleep_for(std::chrono::milliseconds
    (300));
...
}
```

Assignment 12: explore the function on_deactivate 3/3

```
hardware_interface::CallbackReturn ArmHardwareInterface
    ::on_deactivate
(const rclcpp_lifecycle::State & previous_state) {
(void)previous_state;
...
// Disable

if (sys_->driver->deactivate(motor_id_1_) != 0)
   return hardware_interface::CallbackReturn::ERROR;
...
if (sys_->driver->deactivate(motor_id_5_) != 0)
   return hardware_interface::CallbackReturn::ERROR;

return hardware_interface::CallbackReturn::SUCCESS;
}
```

Summary & concluding remarks

- Setting up the lifecycle node can be challanging, take small steps and develop your hardware drivers in a pur C++ fashion outside ROS2 first.
- The ROS2 control framework is not a true real time system and on slower PC systems timing issues will eventually arise, keep this in mind when desiging control loops an leave the true real time tasks to a seperate microcontroller.
- In my experience, implementing your hardware driver in .hpp form and incorporating it with ROS2 node during compilation rather than linking leads to better results.

Assignment 14: Parol_6 lifecycle node

- The challange will be to construct a lifecycle node for the parol6 robot.
- Download the code on github.
- Construct an interface between the control board and ROS2.
- `https://github.com/PCrnjak/PAROL6-Desktop-robot-arm/tree/main/PAROL6controlboardmainsoftware`