# Interface com o *Hardware* no ROS 2

## *Quanser 2DSFJ*

Walter Fetter Lages

fetter@ece.ufrgs.br

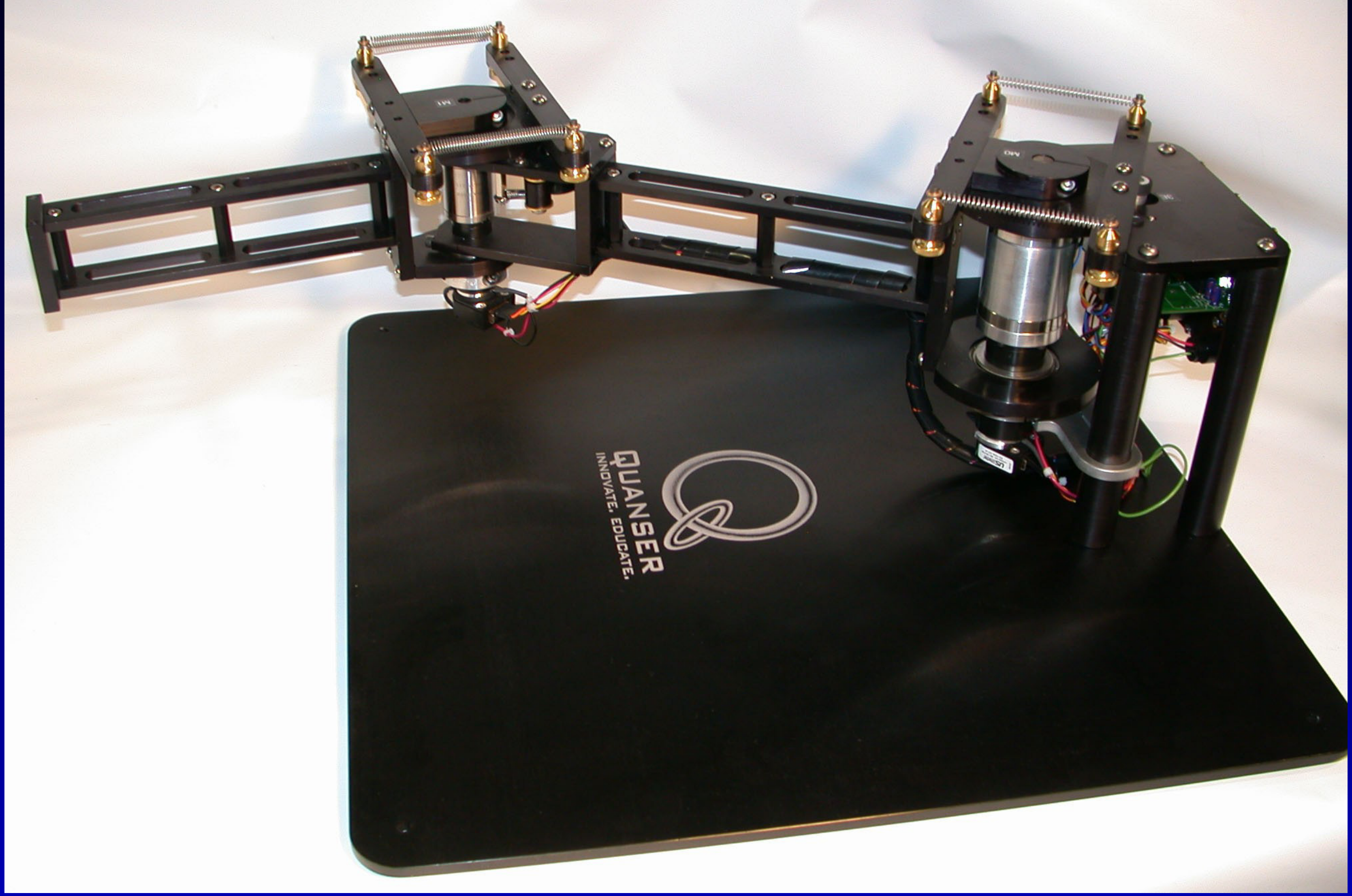
Universidade Federal do Rio Grande do Sul
Escola de Engenharia
Departamento de Sistemas Elétricos de Automação e Energia

ENG10052 Laboratório de Robótica

# Introdução

- Existem várias formas de interface com o *hardware* no ROS 2
  - A maioria delas não suporta operação em tempo-real
  - O *framework* `ros2_control` oferece ferramentas *real-time safe* para implementação da interface com o *hardware*
- Aqui será criado um pacote para implementar uma interface com o *hardware* de forma compatível com o *framework* `ros2_control`
  - Robô Quanser 2DSFJ

# Robô Quanser 2DSFJ

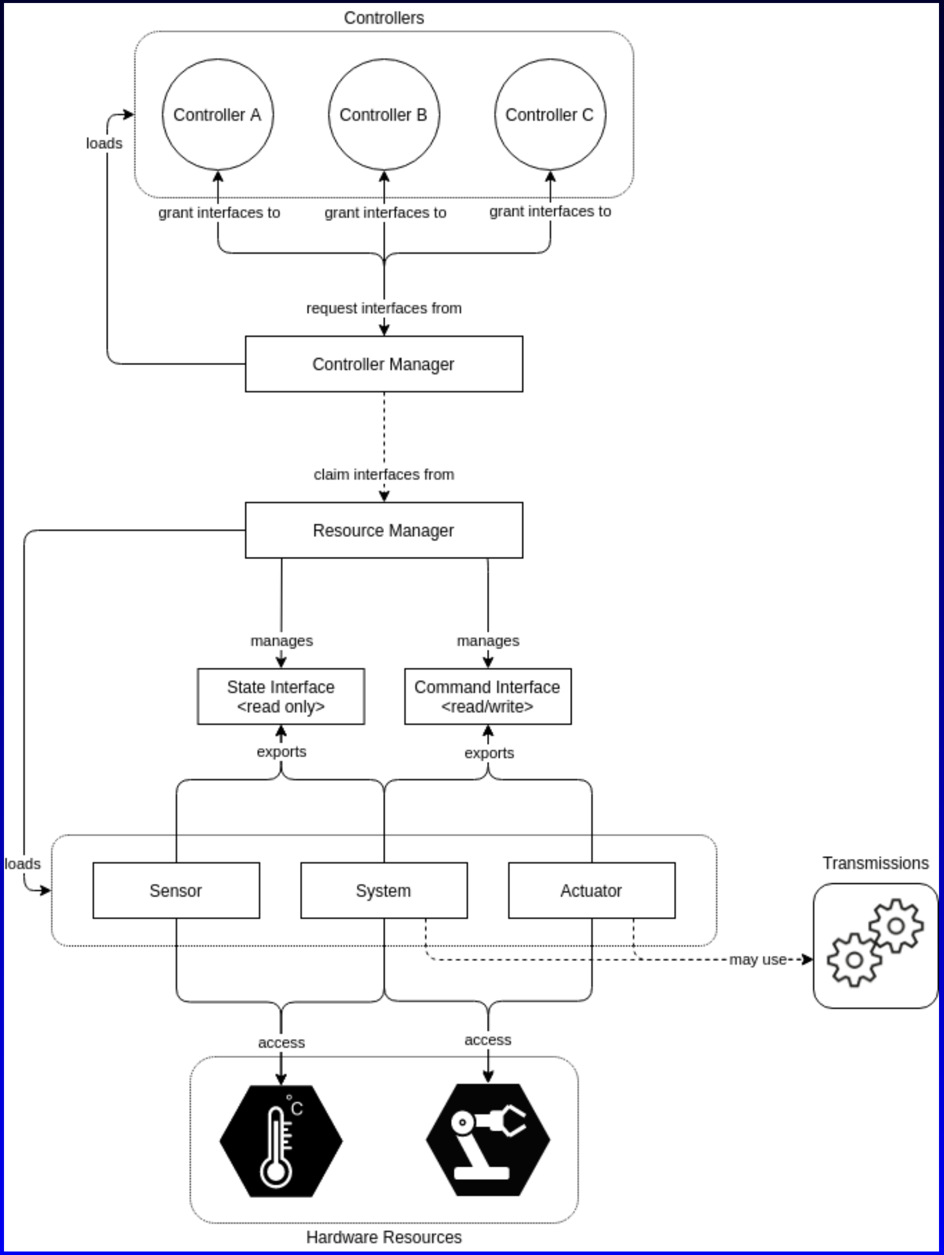


- 2 juntas atuadas em tensão
- 2 encoders em cada junta
  - Juntas flexíveis “travadas”
- 2 sensores fim de curso em cada junta

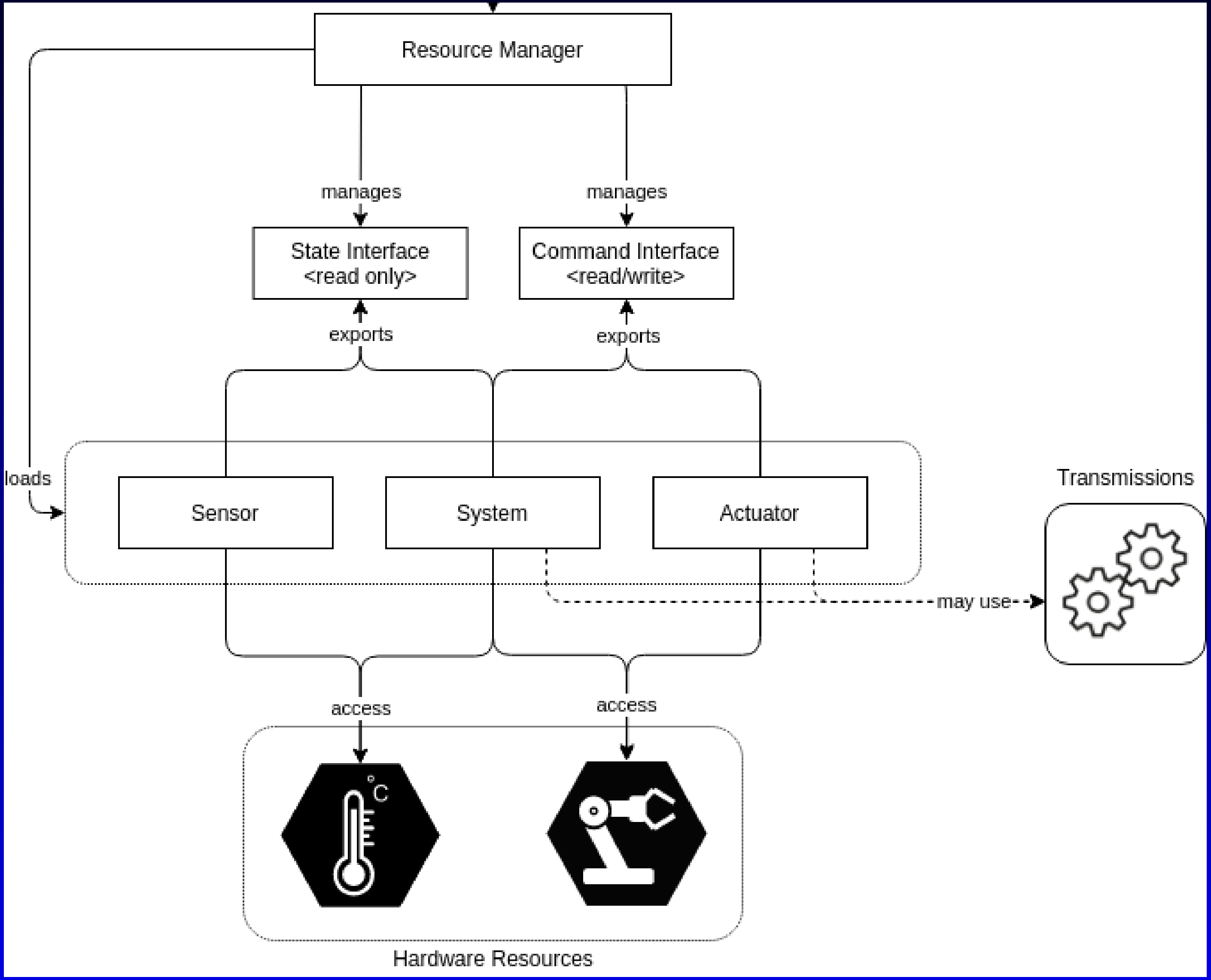
# *Framework* `ros2_control`

- A definição de controlador do `ros_control` não é a clássica de Sistemas de Controle
    - Módulo que é carregado pelo gerenciador de controladores
    - Não necessariamente implementa um controlador
        - O exemplo clássico é o `joint_state_broadcaster`
    - Alguns controladores já implementados

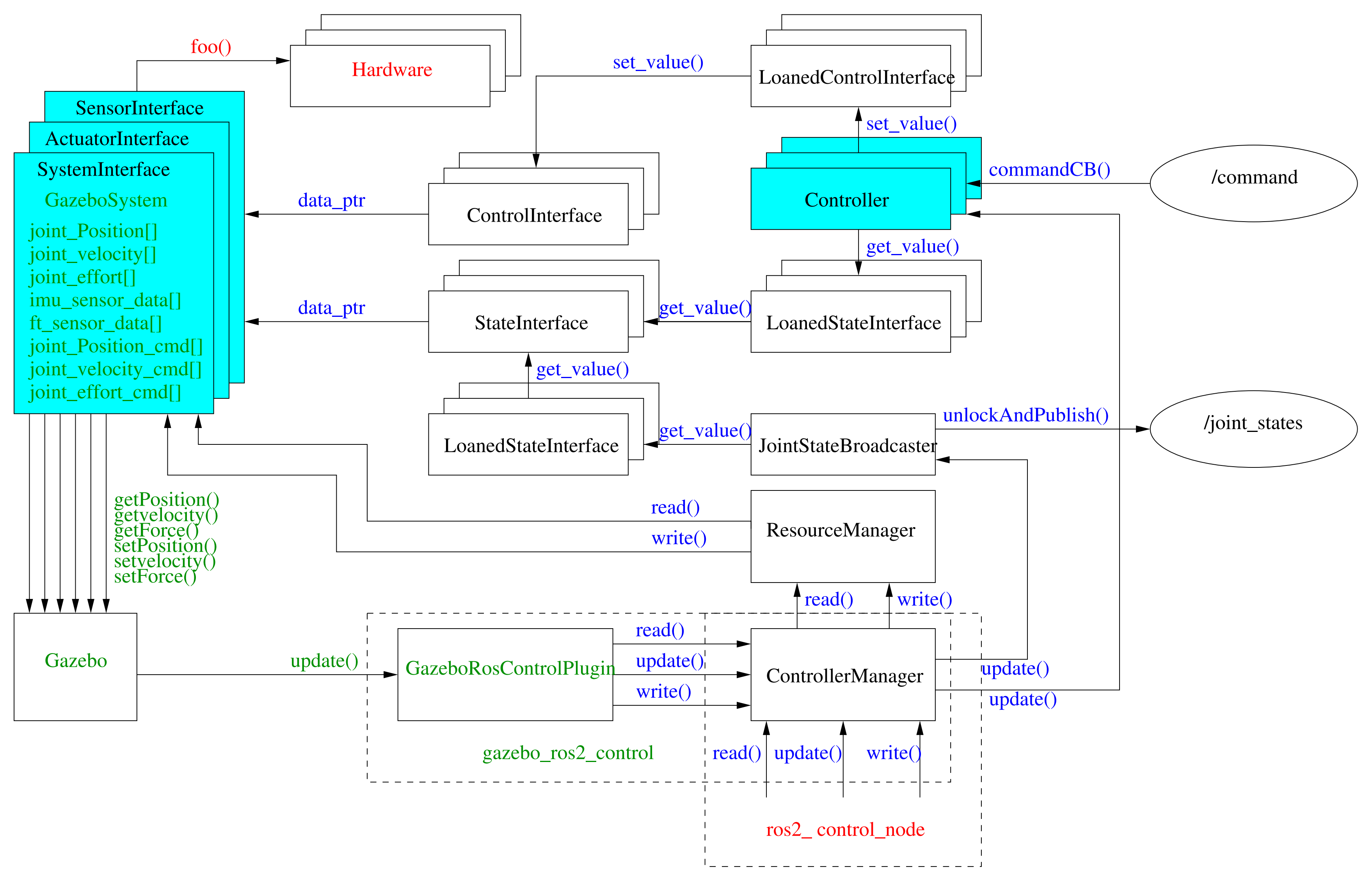
# *Framework* `ros2_control`



# *Framework* `ros2_control`



# Laço de Tempo Real



foo()
Hardware
set_value()
LoanedControlInterface
SensorInterface
ActuatorInterface
SystemInterface
GazeboSystem
joint_Position[]
joint_velocity[]
joint_effort[]
imu_sensor_data[]
ft_sensor_data[]
joint_Position_cmd[]
joint_velocity_cmd[]
joint_effort_cmd[]
set_value()
Controller
commandCB()
/command
data_ptr
ControlInterface
get_value()
data_ptr
StateInterface
get_value()
LoanedStateInterface
get_value()
LoanedStateInterface
get_value()
JointStateBroadcaster
unlockAndPublish()
/joint_states
getPosition()
getvelocity()
getForce()
setPosition()
setvelocity()
setForce()
read()
write()
ResourceManager
read()
write()
Gazebo
update()
GazeboRosControlPlugin
read()
update()
write()
ControllerManager
update()
update()
gazebo_ros2_control
read()
update()
write()
ros2_ control_node

# *Plugin* para o ros2_control

- Incluir na descrição URDF

```
<ros2_control  name="Q2dSystem"  type="system">
   <xacro:if  value="${hardware  ==  ’gazebo’}">
      <!-- Gazebo Classic -->
      <hardware>
         <plugin>gazebo_ros2_control/GazeboSystem<
/plugin>
      </hardware>
   </xacro:if>
   <xacro:if  value="${hardware  ==  ’ignition’}">
      <!-- Gazebo Ignition -->
      <hardware>
         <plugin>ign_ros2_control/IgnitionSystem</
plugin>
      </hardware>
   </xacro:if>
```

# *Plugin* para o ros2_control

```
<xacro:if value="${hardware == 'real_robot'}">
    <!-- Actual Hardware -->
    <hardware>
        <plugin>q2d_hardware/Q2dSystemHardware</plugin>
    </hardware>
</xacro:if>

<joint name="shoulder_active_joint">
    <command_interface name="effort">
        <param name="min">-27.94</param>
        <param name="max">27.94</param>
    </command_interface>
    <state_interface name="position"/>
    <state_interface name="velocity"/>
    <state_interface name="effort"/>
</joint>
```

# *Plugin* para o ros2_control

```
    <joint name="elbow_active_joint">
      <command_interface name="effort">
        <param name="min">-13.62</param>
        <param name="max">13.62</param>
      </command_interface>
      <state_interface name="position"/>
      <state_interface name="velocity"/>
      <state_interface name="effort"/>
    </joint>
  </ros2_control>
```

# Dependências

- ros2_control
  - hardware_interface

```
sudo apt install ros–$ROS_DISTRO–ros2–control
```

# Dependências

- `xacro`
- `test_interface_files`
- `test_msgs`
- Não incluídas na variante desktop do Humble

```
sudo apt install ros–$ROS_DISTRO–xacro
sudo apt install ros–$ROS_DISTRO–test–interface–files
sudo apt install ros–$ROS_DISTRO–test–msgs
```

# Criação do Pacote

```
cd ~/colcon_ws/src
ros2 pkg create ––build–type ament_cmake ––dependencies rclcpp
    hardware_interface pluginlib ––library–name q2d_hardware
    q2d_hardware
```

```
q2d_hardware/
 ├─CMakeLists.txt
 ├─include/
 │  └─q2d_hardware/
 │     ├─q2d_system.hpp
 │     └─visibility_control.h
 ├─package.xml
 ├─q2d_hardware.xml
 └─src/
    └─q2d_system.cpp
```

# package.xml

- Editar o arquivo `q2d_hardware/package.xml`
  - Descrição
  - Mantenedor
  - Licença
  - Dependências

# `CMakeLists.txt`

- Editar `CMakeLists.txt` para descomentar e ajustar as *tags*:

```
add_library(q2d_hardware SHARED src/q2d_system.cpp)
target_compile_features(q2d_hardware PUBLIC c_std_99
      cxx_std_17)
target_include_directories(q2d_hardware PRIVATE include)

ament_target_dependencies(
  q2d_hardware
  "rclcpp"
  "hardware_interface"
  "pluginlib"
)
```

# CMakeLists.txt

```
pluginlib_export_plugin_description_file(hardware_interface
      q2d_hardware.xml)

install(
  TARGETS q2d_hardware
  EXPORT export_${PROJECT_NAME}
  ARCHIVE DESTINATION lib
  LIBRARY DESTINATION lib
  RUNTIME DESTINATION bin
)

install(DIRECTORY launch
      DESTINATION share/${PROJECT_NAME}
)
```

# CMakeLists.txt

```
ament_export_libraries(
  q2d_hardware
)
```

# q2d_hardware.xml

```
<library path="q2d_hardware">
    <class name="q2d_hardware/Q2dSystemHardware"
        type="q2d_hardware::Q2dSystemHardware"
        base_class_type="
    hardware_interface::SystemInterface">
        <description>
            Quanser 2DSFJ  system hardware interface.
    It uses effort
            command interfaces and position and
    velocity state interfaces.
        </description>
    </class>
</library>
```

# Funções a Implementar

- `on_init()`
- `on_configure()`
- `export_state_interfaces()`
- `export_command_interfaces()`
- `on_activate()`
- `on_deactivate()`
- `read()`
- `write()`
- Opcionais
  - `prepare_command_mode_switch()`
  - `perform_command_mode_switch()`

# q2d_system.hpp

```
#ifndef Q2D_HARDWARE__Q2D_SYSTEM_HPP_
#define Q2D_HARDWARE__Q2D_SYSTEM_HPP_

#include "hardware_interface/system_interface.hpp"
#include "rclcpp/macros.hpp"

#include "q2d_hardware/visibility_control.h"

#include "aic.h"
```

# q2d_system.hpp

```
namespace q2d_hardware
{
    class Q2dSystemHardware: public hardware_interface::
    SystemInterface
    {
        public:
        RCLCPP_SHARED_PTR_DEFINITIONS(
    Q2dSystemHardware)

        Q2D_HARDWARE_PUBLIC
        hardware_interface::CallbackReturn on_init(
    const hardware_interface::HardwareInfo &info)
    override;
```

# q2d_system.hpp

```
   hardware_interface::CallbackReturn
on_configure(const rclcpp_lifecycle::State &
previous_stte) override;

   Q2D_HARDWARE_PUBLIC
   std::vector<hardware_interface::StateInterface
> export_state_interfaces(void) override;

   Q2D_HARDWARE_PUBLIC
   std::vector<hardware_interface::
CommandInterface> export_command_interfaces(void)
 override;
```

# q2d_system.hpp

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

    Q2D_HARDWARE_PUBLIC
    hardware_interface::CallbackReturn
on_deactivate(const rclcpp_lifecycle::State &
previous_state) override;

    Q2D_HARDWARE_PUBLIC
    hardware_interface::return_type read(const
rclcpp::Time &time,const rclcpp::Duration &period)
 override;
```

# q2d_system.hpp

```
      hardware_interface::return_type write(const
   rclcpp::Time &time,const rclcpp::Duration &period)
    override;

      private:
      std::vector<double> command_;
      std::vector<double> position_;
      std::vector<double> velocity_;
      std::vector<double> effort_;

      std::vector<std::unique_ptr<aic>> board_;
      std::vector<double> Kt_;
      std::vector<double> Ka_;
      std::vector<double> Ra_;
      std::vector<double> N_;
   };

}  // namespace q2d_hardware
```

# q2d_system.cpp

```
#include "hardware_interface/types/
   hardware_interface_type_values.hpp"
#include "rclcpp/rclcpp.hpp"

#include "q2d_hardware/q2d_system.hpp"

namespace q2d_hardware
{
   hardware_interface::CallbackReturn
   Q2dSystemHardware::on_init(const
   hardware_interface::HardwareInfo & info)
   {
      if(hardware_interface::SystemInterface::on_init
    (info) != CallbackReturn::SUCCESS)
         return CallbackReturn::ERROR;

      /* Initialize Hardware */
      board_.resize(info_.joints.size());
```

# q2d_system.cpp

```
position_.resize(info_.joints.size());
velocity_.resize(info_.joints.size());
effort_.resize(info_.joints.size());

command_.resize(info_.joints.size());

Kt_.resize(info_.joints.size());
Kt_[0]=0.119;
Kt_[1]=0.0234;
Ka_.resize(info_.joints.size());
Ka_[0]=0.119;
Ka_[1]=0.0234;
Ra_.resize(info_.joints.size());
Ra_[0]=11.5;
Ra_[1]=2.32;
N_.resize(info_.joints.size());
```

# q2d_system.cpp

```
  for (const hardware_interface::ComponentInfo &
joint : info_.joints)
  {
     if(joint.command_interfaces.size() != 1)
     {
        RCLCPP_FATAL(rclcpp::get_logger("
Q2dSystemHardware"),
           "Joint ’%s’ has %ld command
interfaces found. 1 expected.",
           joint.name.c_str(),joint.
command_interfaces.size());
        return CallbackReturn::ERROR;
     }
```

# q2d_system.cpp

```
      if(joint.command_interfaces[0].name !=
hardware_interface::HW_IF_EFFORT)
      {
         RCLCPP_FATAL(rclcpp::get_logger("
Q2dSystemHardware"),
            "Joint '%s' have %s command
interfaces found. '%s' expected.",
            joint.name.c_str(),
            joint.command_interfaces[0].name.
c_str(),
            hardware_interface::HW_IF_EFFORT);
         return CallbackReturn::ERROR;
      }
```

# q2d_system.cpp

```
      if(joint.state_interfaces.size() != 3 )
      {
         RCLCPP_FATAL(rclcpp::get_logger("
Q2dSystemHardware"),
            "Joint ’%s’ has %ld state interface.
2 expected.",
            joint.name.c_str(),
            joint.state_interfaces.size());
         return CallbackReturn::ERROR;
      }
```

# q2d_system.cpp

```
     if(joint.state_interfaces[0].name !=
hardware_interface::HW_IF_POSITION)
     {
        RCLCPP_FATAL(rclcpp::get_logger("
Q2sSystemHardware"),
           "Joint ’%s’ have ’%s’ as first state
interface. ’%s’ expected.",
           joint.name.c_str(), joint.
state_interfaces[0].name.c_str(),
           hardware_interface::HW_IF_POSITION);
        return CallbackReturn::ERROR;
     }
```

# q2d_system.cpp

```
     if(joint.state_interfaces[1].name !=
hardware_interface::HW_IF_VELOCITY)
     {
        RCLCPP_FATAL(rclcpp::get_logger("
Q2sSystemHardware"),
           "Joint ’%s’ have ’%s’ as second state
 interface. ’%s’ expected.",
           joint.name.c_str(), joint.
state_interfaces[1].name.c_str(),
           hardware_interface::HW_IF_VELOCITY);
        return CallbackReturn::ERROR;
     }
```

# q2d_system.cpp

```
      if(joint.state_interfaces[2].name !=
hardware_interface::HW_IF_EFFORT)
      {
         RCLCPP_FATAL(rclcpp::get_logger("
Q2sSystemHardware"),
            "Joint '%s' have '%s' as second state
 interface. '%s' expected.",
            joint.name.c_str(), joint.
state_interfaces[2].name.c_str(),
            hardware_interface::HW_IF_EFFORT);
         return CallbackReturn::ERROR;
      }
   }
   return CallbackReturn::SUCCESS;
}
```

# q2d_system.cpp

```
hardware_interface::CallbackReturn
Q2dSystemHardware::on_configure(const
rclcpp_lifecycle::State & /*previous_stte*/)
{
    /* configure hardware */
    aic_comm_config_t param;
    param.aic_comm_device=rs232;
```

# q2d_system.cpp

```
   for (auto i=0u;i < board_.size();i++)
   {
      param.aic_serial_port="/dev/tty/USB"+ (’0’+
i);
      if((board_[i]=std::make_unique<aic>(param))
== NULL)
         RCLCPP_ERROR_STREAM(rclcpp::get_logger("
Q2dSystemHardware"),
            "Can not configure " << param.
aic_serial_port << " for interfacing with joint
" << i);
   }
   return CallbackReturn::SUCCESS;
}
```

# q2d_system.cpp

```
std::vector<hardware_interface::StateInterface>
Q2dSystemHardware::export_state_interfaces(void)
{
   std::vector<hardware_interface::StateInterface
> state_interfaces;
   for(auto i=0u;i < info_.joints.size();i++)
   {
      state_interfaces.emplace_back(
hardware_interface::StateInterface(info_.joints[
i].name,
         hardware_interface::HW_IF_POSITION,&
position_[i]));
      state_interfaces.emplace_back(
hardware_interface::StateInterface(info_.joints[
i].name,
         hardware_interface::HW_IF_VELOCITY,&
velocity_[i]));
```

# q2d_system.cpp

```
     state_interfaces.emplace_back(
hardware_interface::StateInterface(info_.joints[
i].name,
        hardware_interface::HW_IF_EFFORT,&
command_[i]));
  }
  return state_interfaces;
}
```

# q2d_system.cpp

```
std::vector<hardware_interface::CommandInterface>
 Q2dSystemHardware::export_command_interfaces(
void)
{
   std::vector<hardware_interface::
CommandInterface> command_interfaces;
   for(auto i=0u;i < info_.joints.size();i++)
   {
      command_interfaces.emplace_back(
hardware_interface::CommandInterface(info_.
joints[i].name,
         hardware_interface::HW_IF_EFFORT,&
command_[i]));
   }
   return command_interfaces;
}
```

# q2d_system.cpp

```
hardware_interface::CallbackReturn
 Q2dSystemHardware::on_activate(const
 rclcpp_lifecycle::State &/*previous_state*/)
 {
     for(auto i=0u;i < position_.size();i++)
     {
         position_[i]=0;
         velocity_[i]=0;
         effort_[i]=0;
         command_[i]=0;
     }

     /* activate hardware */
     for(auto i=0u;i < board_.size();i++) board_[i
 ]->set_motor_voltage(0);

     return CallbackReturn::SUCCESS;
 }
```

# q2d_system.cpp

```
hardware_interface::CallbackReturn
 Q2dSystemHardware::on_deactivate(const
 rclcpp_lifecycle::State &/*previous_state*/)
 {
     /* deactivate hardware */
     for(auto i=0u;i < board_.size();i++) board_[i
 ]->~aic();

     return CallbackReturn::SUCCESS;;
 }
```

# q2d_system.cpp

```
hardware_interface::return_type Q2dSystemHardware
::read(const rclcpp::Time &/*time*/,const rclcpp::
Duration &period)
{
    /* read hardware */
    for(auto i=0u;i < board_.size();i++)
    {
        auto dp=board_[i]->read_displacement_sensors
();
        velocity_[i]=(dp.joint_displacement-
position_[i])/period.seconds();
        position_[i]=dp.joint_displacement;
        effort_[i]=command_[i];
    }

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

# q2d_system.cpp

```
    hardware_interface::return_type Q2dSystemHardware
    ::write(const rclcpp::Time &/*time*/,const rclcpp::
    Duration &/*period*/)
    {
        /* write hardware */
        for(auto i=0u;i < board_.size();i++)
            board_[i]->set_motor_voltage(N_[i]*Ra_[i]/
    Kt_[i]*command_[i]+Ka_[i]/N_[i]*velocity_[i]);

        return hardware_interface::return_type::OK;
    }

}  // namespace q2d_hardware

#include "pluginlib/class_list_macros.hpp"
PLUGINLIB_EXPORT_CLASS(q2d_hardware::
    Q2dSystemHardware,hardware_interface::
    SystemInterface)
```

# Baixar o Pacote

- Clonagem inicial:

```
cd ~/colcon_ws/src
git clone –b humble_hardware http://git.ece.ufrgs.br/eng10052/
      q2d
```

- Se já clonado:

```
cd ~/colcon_ws/src/q2d
git pull
git checkout humble_hardware
```

# Compilação do Pacote

```
cd ~/colcon_ws
colcon build ––symlink–install
source ~/colcon_ws/install/setup.bash
```

# Carregar a Interface

```
ros2 launch q2d_hardware controller_manager.launch.xml
```

```
<launch>
   <node name="controller_manager" pkg="
   controller_manager" exec="ros2_control_node">
      <param name="robot_description" value="$(
   command ’xacro $(find-pkg-share q2d_description)
   /urdf/q2d.urdf hardware:=real_robot’)" type="str"
   />
      <param name="use_sim_time" value="false"/>
      <param name="update_rate" value="1000"/>
   </node>
</launch>
```

# ros2 control

- Listar os componentes *hardware*

```
ros2 control list_hardware_components
```

```
Hardware Component 0
        name: Q2dSystem
        type: system
        plugin name: plugin name missing!
        state: id=3 label=active
        command interfaces
                shoulder_active_joint/effort [available] [unclaimed]
                elbow_active_joint/effort [available] [unclaimed]
```

# ros2 control

- Listar as interfaces de *hardware*

```
ros2 control list_hardware_interfaces
```

```
command interfaces
        elbow_active_joint/effort [available] [unclaimed]
        shoulder_active_joint/effort [available] [unclaimed]
state interfaces
        elbow_active_joint/effort
        elbow_active_joint/position
        elbow_active_joint/velocity
        shoulder_active_joint/effort
        shoulder_active_joint/position
        shoulder_active_joint/velocity
```

# Conversão de Torque para Tensão

- É possível modelar um motor D.C. (incluindo a redução) como um atuador de torque ideal sujeito à um atrito viscoso

$$\tau_j = \frac{K_\tau}{nR_a} v_a - \frac{K_\tau K_a}{n^2 R_a} \dot{\theta}_j$$

$$v_a = \frac{nR_a}{K_\tau} \tau_j + \frac{K_a}{n} \dot{\theta}_j$$

- Para o Quanser 2DSFJE:

| | Motor1 | Motor 2 | Unidade |
|---|---|---|---|
| Constante de torque | 0.119 | 0.0234 | N m/A |
| Constante de armadura | 0.119 | 0.0234 | V s/rad |
| Resistência de armadura | 11.5 | 2.32 | Ω |
| Redução | 0.01 | 0.02 | |

# Controlador

- Para movimentar o robô é necessário carregar o(s) controladore(s)
- Será usado um controlador de *by-pass*
- `effort_controllers/JointGroupEffortController`
  - Copia a referência para a saída
- Para ober a leitura dos sensores é necessário carregar o(s) *broadcasters*
  - `joint_state_broadcaster/JointStateBroadcaster`

# config/group_bypass.yaml

```
group_controller:
    ros__parameters:
        joints:
            – shoulder_active_joint
            – elbow_active_joint
```

# Carregar o Controlador

```
ros2 launch q2d_bringup group_bypass.launch.xml
```

```
<launch>
    <arg name="config" default="$(find-pkg-share
    q2d_bringup)/config/group_bypass.yaml"/>

    <node name="group_controller_spawner" pkg="
    controller_manager" exec="spawner"
        args="-t effort_controllers/
    JointGroupEffortController -p $(var config)
    group_controller"/>

    <node name="joint_state_broadcaster_spawner" pkg="
    controller_manager" exec="spawner"
        args="-t joint_state_broadcaster/
    JointStateBroadcaster joint_state_broadcaster"/>
</launch>
```

# Carga do Gerenciador e Controlador

```
ros2 launch q2d_bringup hardware.launch.xml
```

```
<launch>
    <arg name="gui" default="true"/>

    <arg name="controller" default="group_bypass"/>
    <arg name="config" default="$(find-pkg-share
     q2d_bringup)/config/$(var controller).yaml"/>

    <include file="$(find-pkg-share q2d_hardware)/launch
     /controller_manager.launch.xml"/>

    <include file="$(find-pkg-share q2d_bringup)/launch
     /$(var controller).launch.xml">
        <arg name="config" value="$(var config)"/>
        <arg name="use_sim_time" value="false"/>
    </include>
```

# Carga do Gerenciador e Controlador

```
    <include if="$(var gui)" file="$(find-pkg-share
    q2d_description)/launch/display.launch.xml">
        <arg name="gui" value="false"/>
    </include>
</launch>
```

# Gráfico de Computação

```
rqt_graph &
```

# Movimentação do Robô

- Com o controlador *bypass* se aplica diretamente o torque nas juntas (malha aberta)

```
ros2 topic pub /group_controller/commands std_msgs/msg/
    Float64MultiArray '{data: [1.0, 1.0] }' –1
```

- ou

```
ros2 run q2d_bringup group_torque_step.sh 1 1
```

- O controlador fica aplicando o torque até receber outra referência
- Não é muito útil para movimentar o robô "na mão"
- Usado em testes em malha aberta

# scripts/group_torque_step.sh

```
#!/bin/bash

if [ "$#" –ne 2 ]; then
        echo "Usage: $0 <shoulder torque> <elbow toque>"
        exit –1;
fi;

ros2 topic pub /group_controller/commands std_msgs/msg/
       Float64MultiArray "{data: [$1, $2] }" –1
```