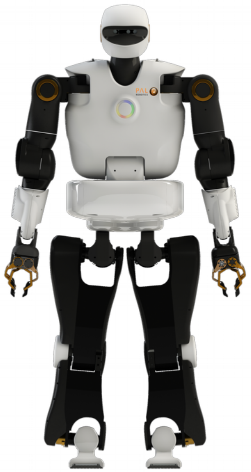
TALOS Control System Architecture

and Whole Body Controller



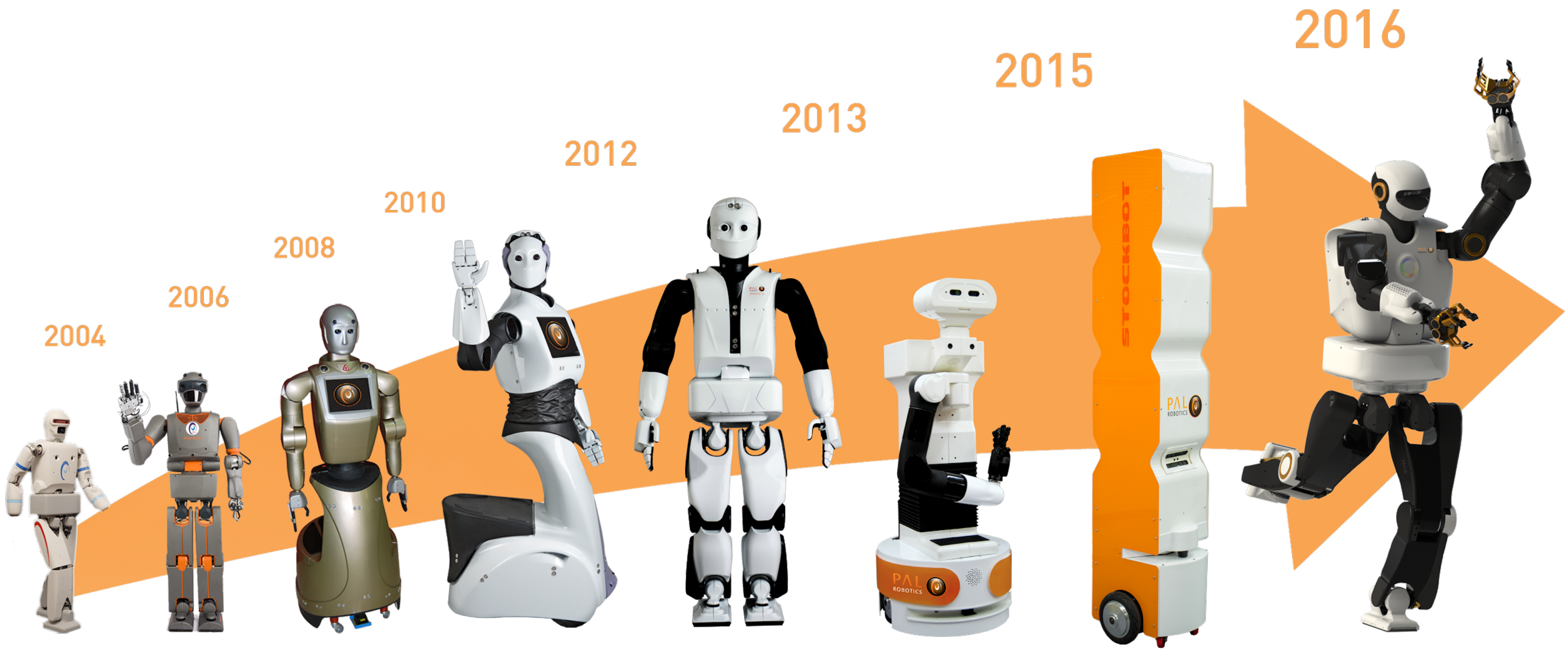
Luca Marchionni, CTO at PAL Robotics

January 30, Martigny, Memmo Winter School

Outline

- PAL Robotics
- TALOS presentation
- Control system architecture
- ROS Control
- Whole Body Control in PAL
- Some videos

PAL Robotics in a nutshell



Partners and customers

Public repositories

TIAGO

wiki.ros.org/Robots/TIAGo

wiki.ros.org/Robots/TIAGo/Tutorials

REEM-C

wiki.ros.org/Robots/REEM-C

wiki.ros.org/Robots/REEM-C/Tutorials

TALOS

https://github.com/pal-robotics/talos_robot

https://github.com/pal-robotics/talos_simulation

https://github.com/pal-robotics/talos_tutorials

TALOS high performance robot

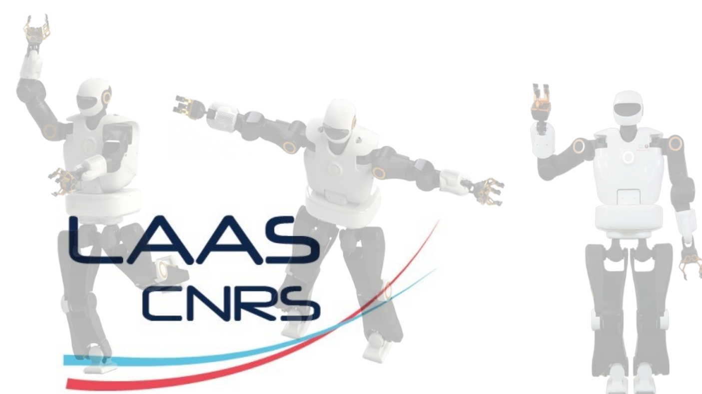
Height: 1,75 m
Weight: 95 Kg

Torque control at joint level

6 Kg Payload per arm (fully extended)

EtherCAT control loop up to 5 KHz

100% Electrical actuators



TALOS high performance robot

High performance BLDC motors

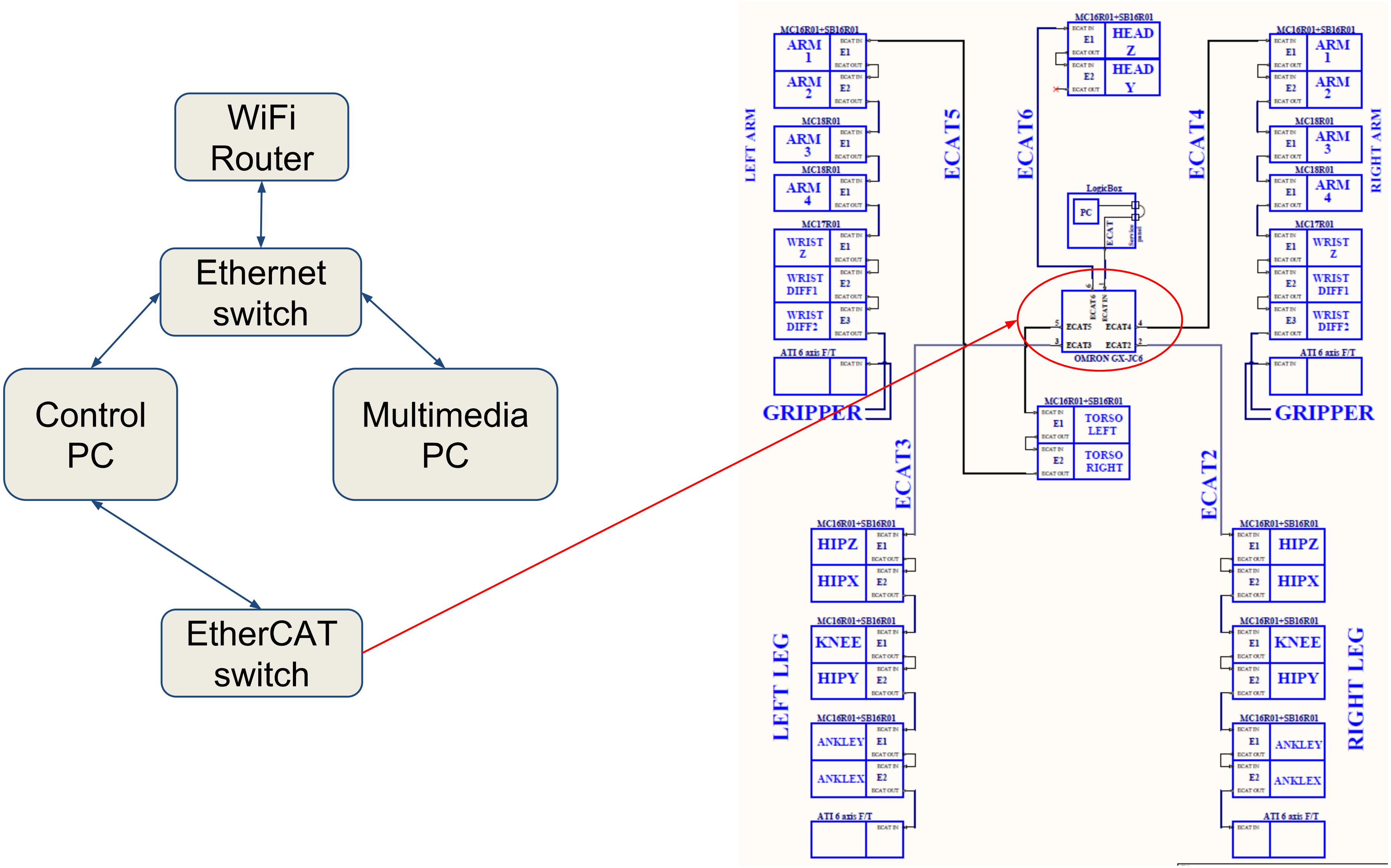
Industrial grade wire hardness

Modular design for easy maintenance

X2 Intel i7 (upgradeable COM Express 6)

TALOS introduction video

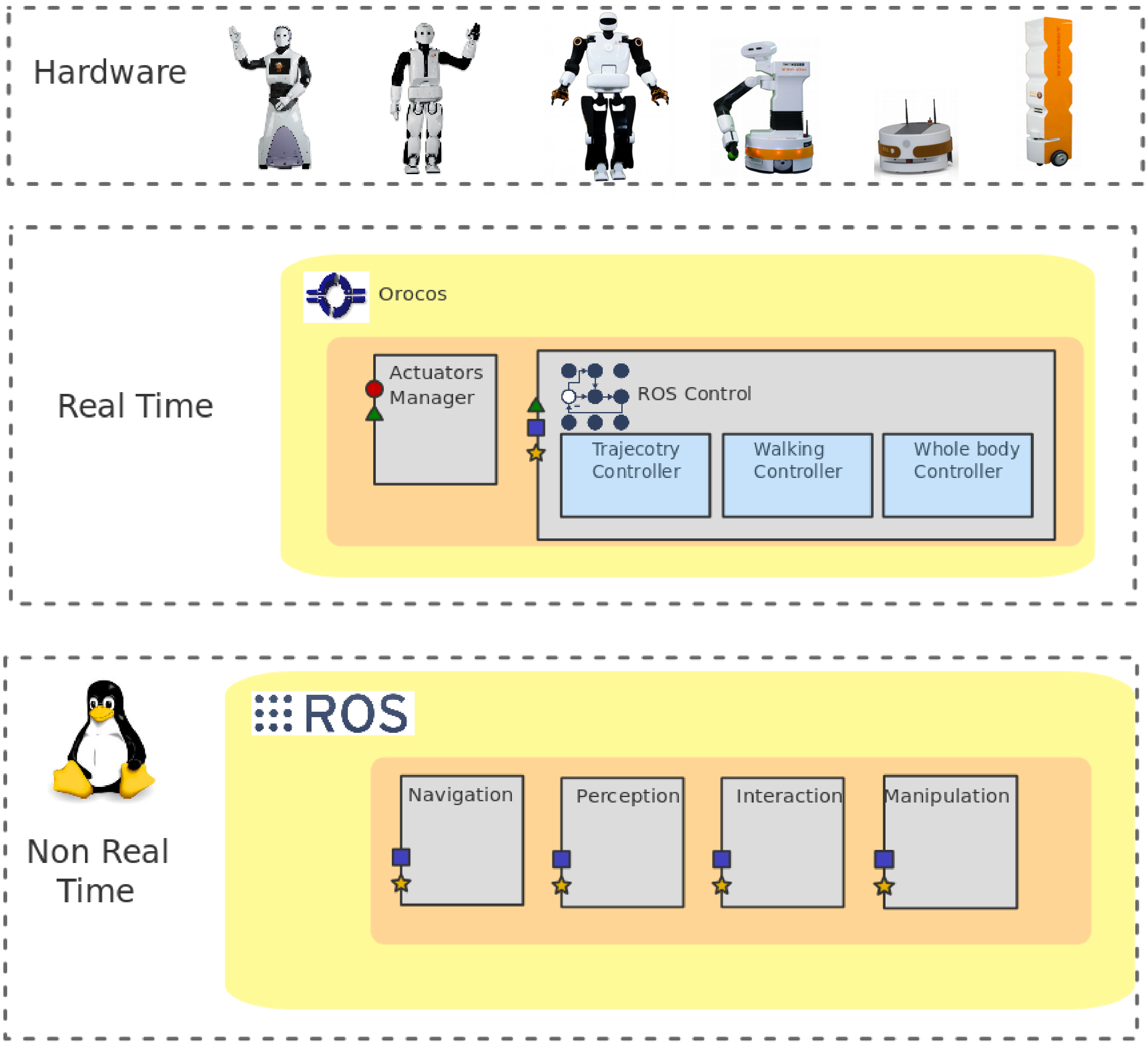
TALOS network architecture



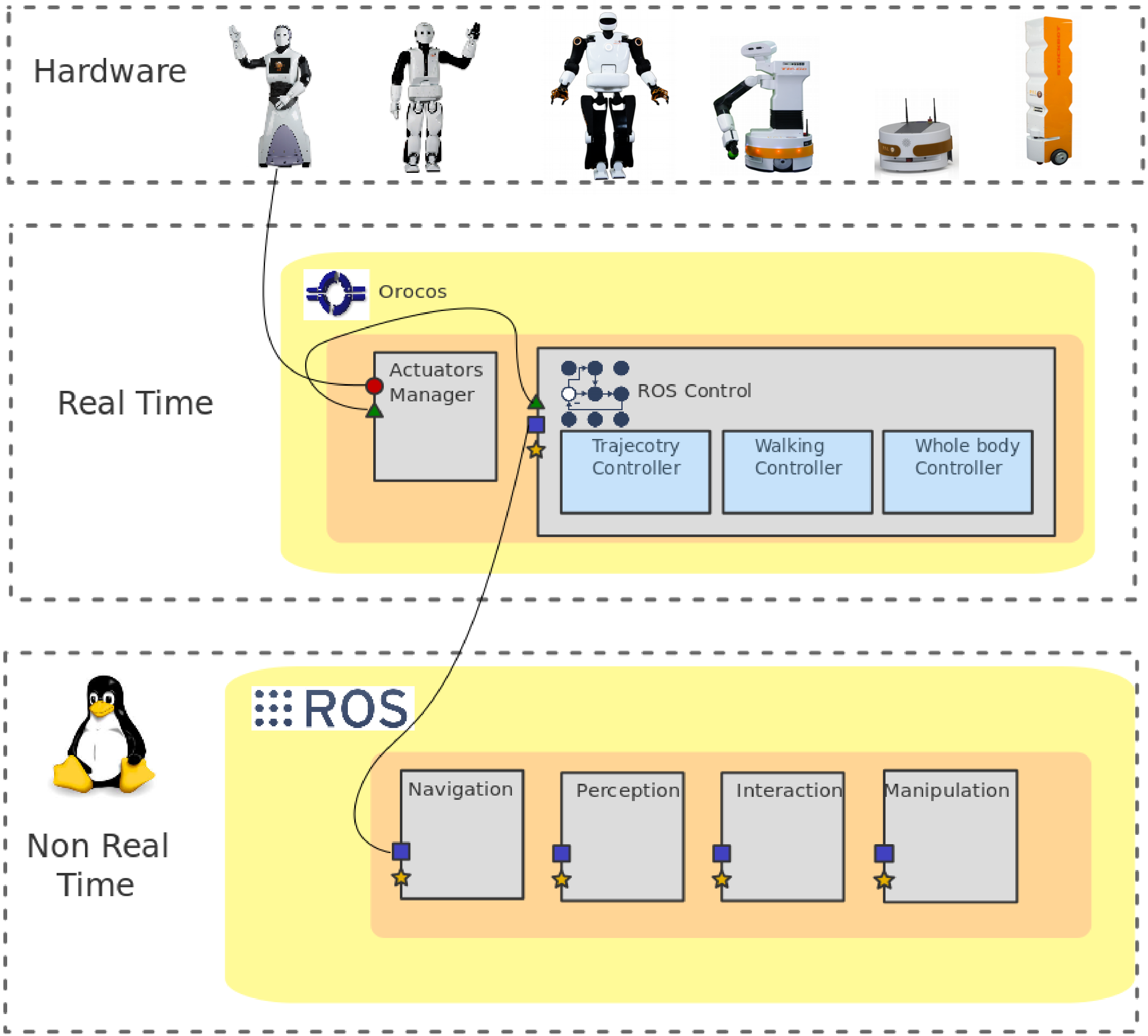
Software overview

	Stable	Work in progress	Future?
Operating System	• Ubuntu 16.04 LTS • Linux Preemp-rt	• Ubuntu 18.04 LTS • Linux Preemp-rt	• **Linux Real Time**
Robotics middleware	• Orocos 2.8 • ROS Kinetic • PAL Erbium	• Orocos 2.8 • ROS Melodic • PAL Fermium	• **ROS 2.0** • **PAL Gallium?**

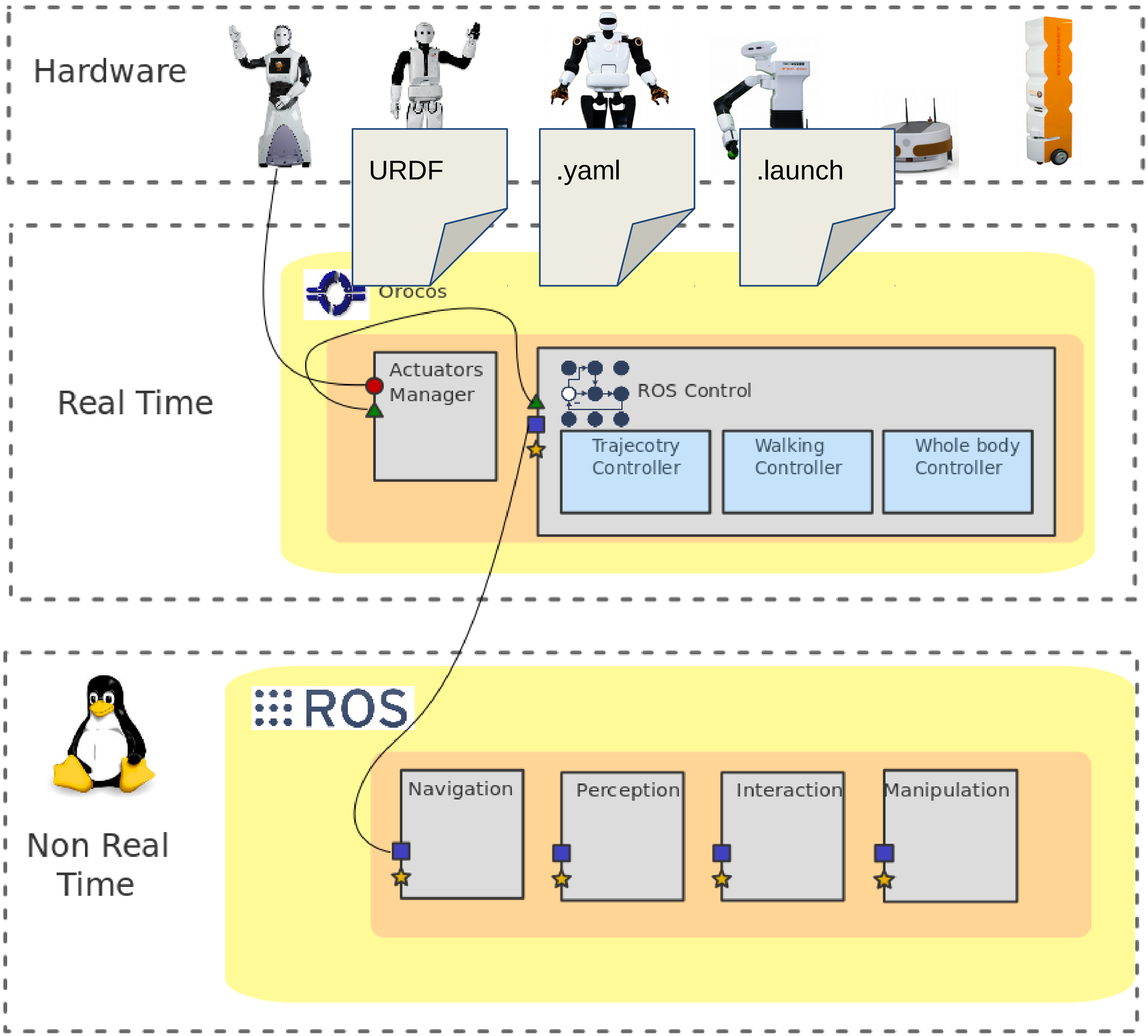
Control architecture



Control architecture



Control architecture



ROS control motivation

Have you ever?

- **used** a controller / robot driver **not** written by you?
- **implemented** a controller / robot driver **yourself**?
  - subject to **real-time** constraints?

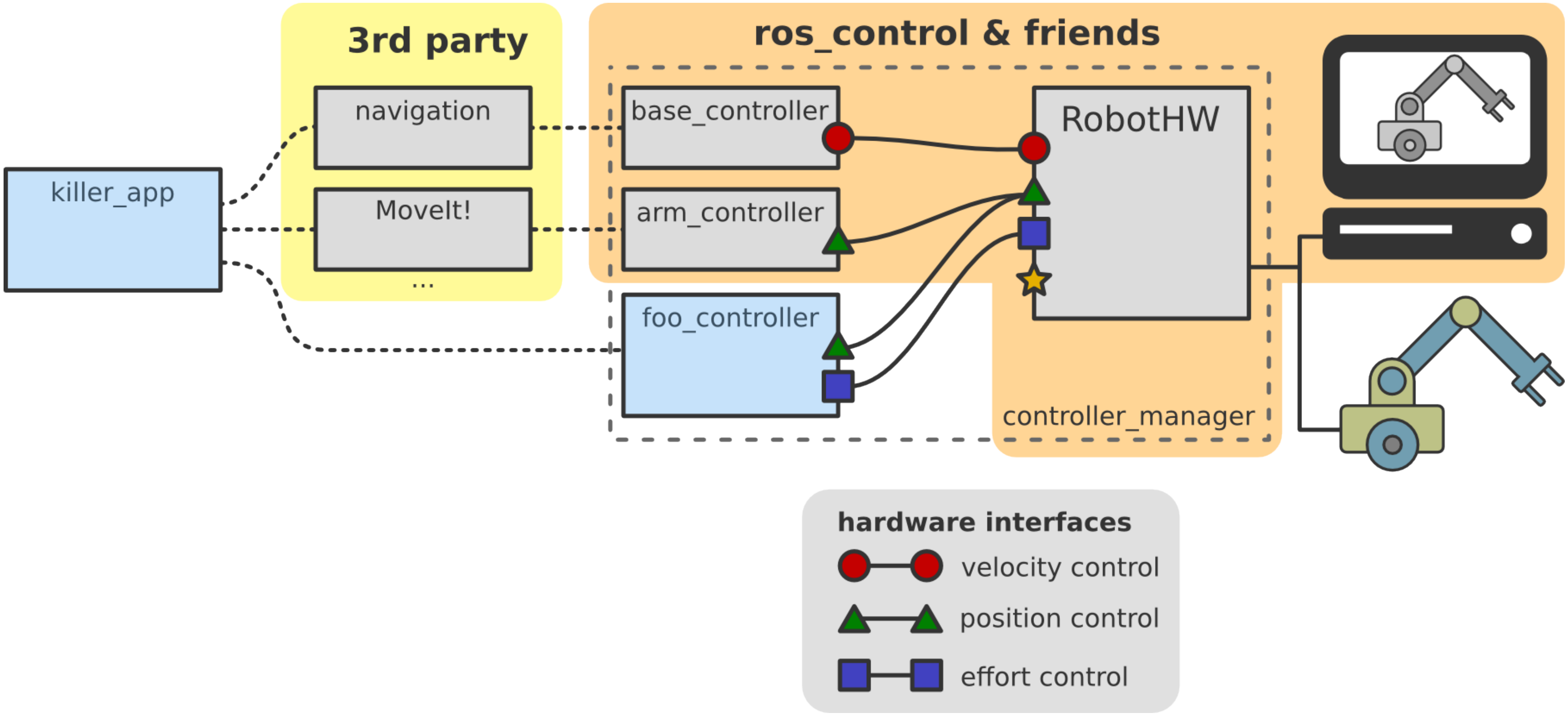
ROS control history

- **pr2_controller_manager** (2009)
  - developed mainly by **Willow Garage** (WG)
  - **PR2**-specific
- **ros_control** (late 2012)
  - started by **hiDOF**, in collaboration with **WG**
  - continued by **PAL Robotics** and **community**
  - **robot-agnostic** version of the pr2_controller_manager
- **ROSIN** ros_control project (late 2018)
  - Merge pal-robotics forks with ros_control master

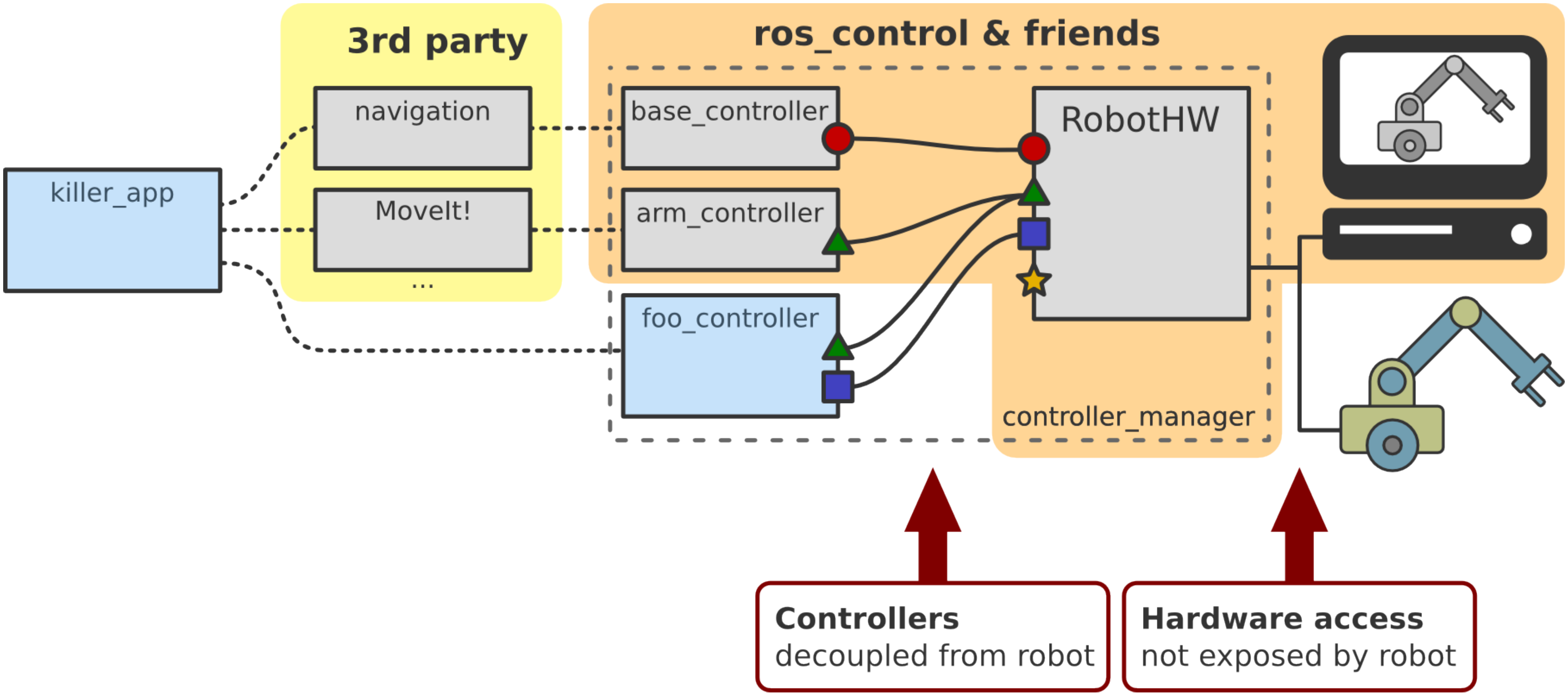
ROS control resources

- http://wiki.ros.org/ros_control
- https://github.com/ros-controls/
- ROScon 2014 talk “ros_control: An overview”, Adolfo Rodríguez Tsouroukdissian
- S. Chitta, E. Marder-Eppstein, W. Meeussen, V. Pradeep, A. Rodríguez Tsouroukdissian, J. Bohren, D. Coleman, B. Magyar, G. Raiola, M. Lüdtke and E. Fernandez Perdomo **"ros_control: A generic and simple control framework for ROS"**, The Journal of Open Source Software, 2017.

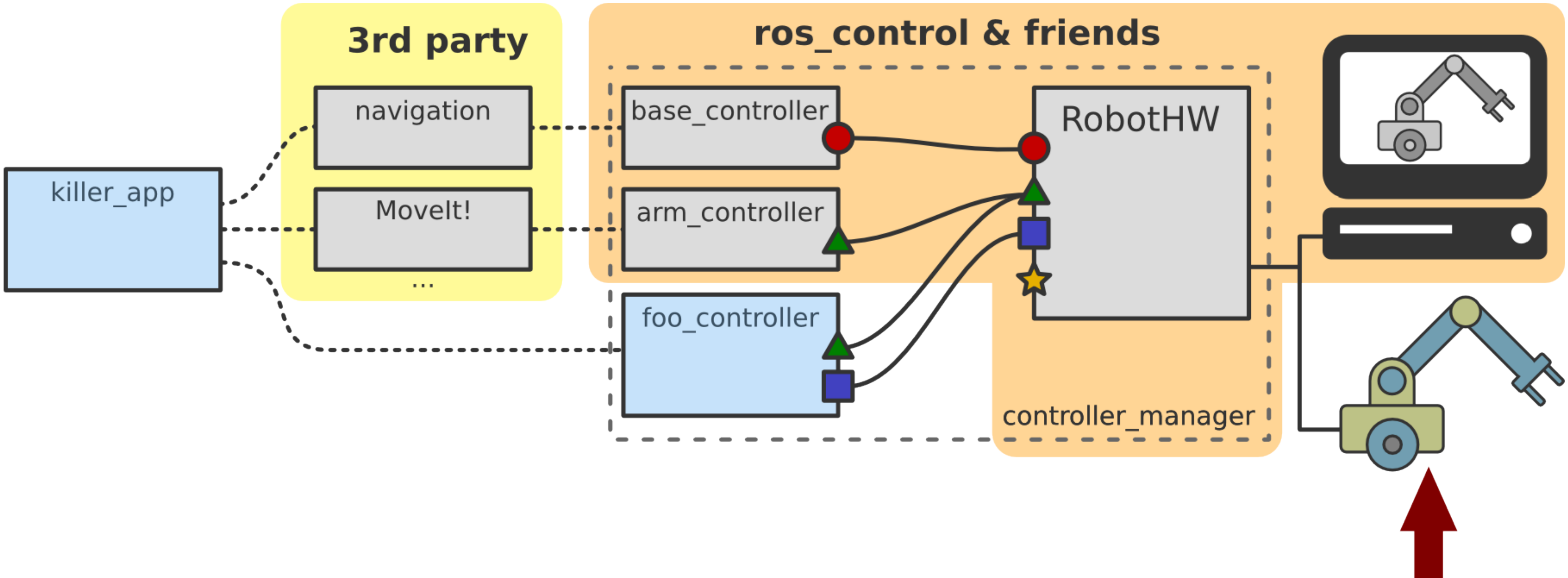
ROS control – the big picture



ROS control – the big picture

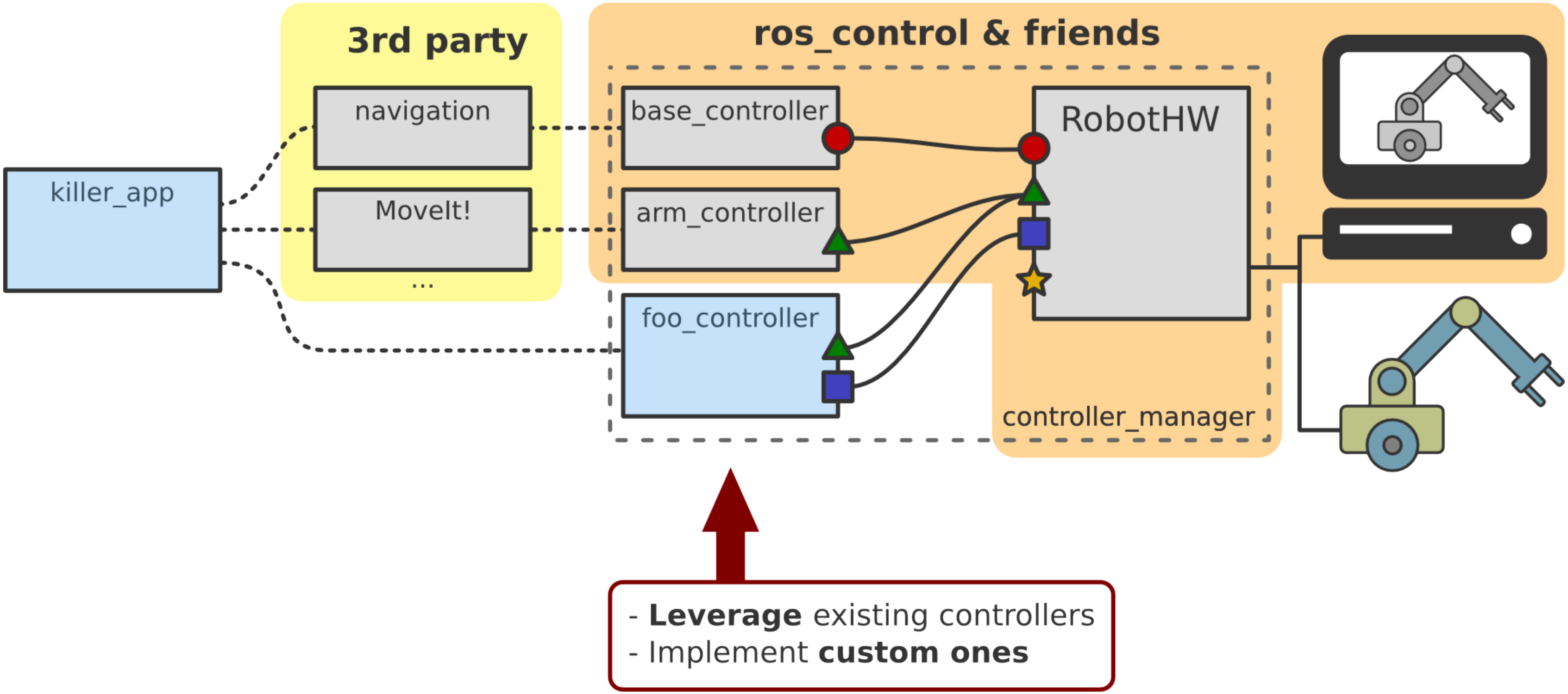


ROS control – the big picture



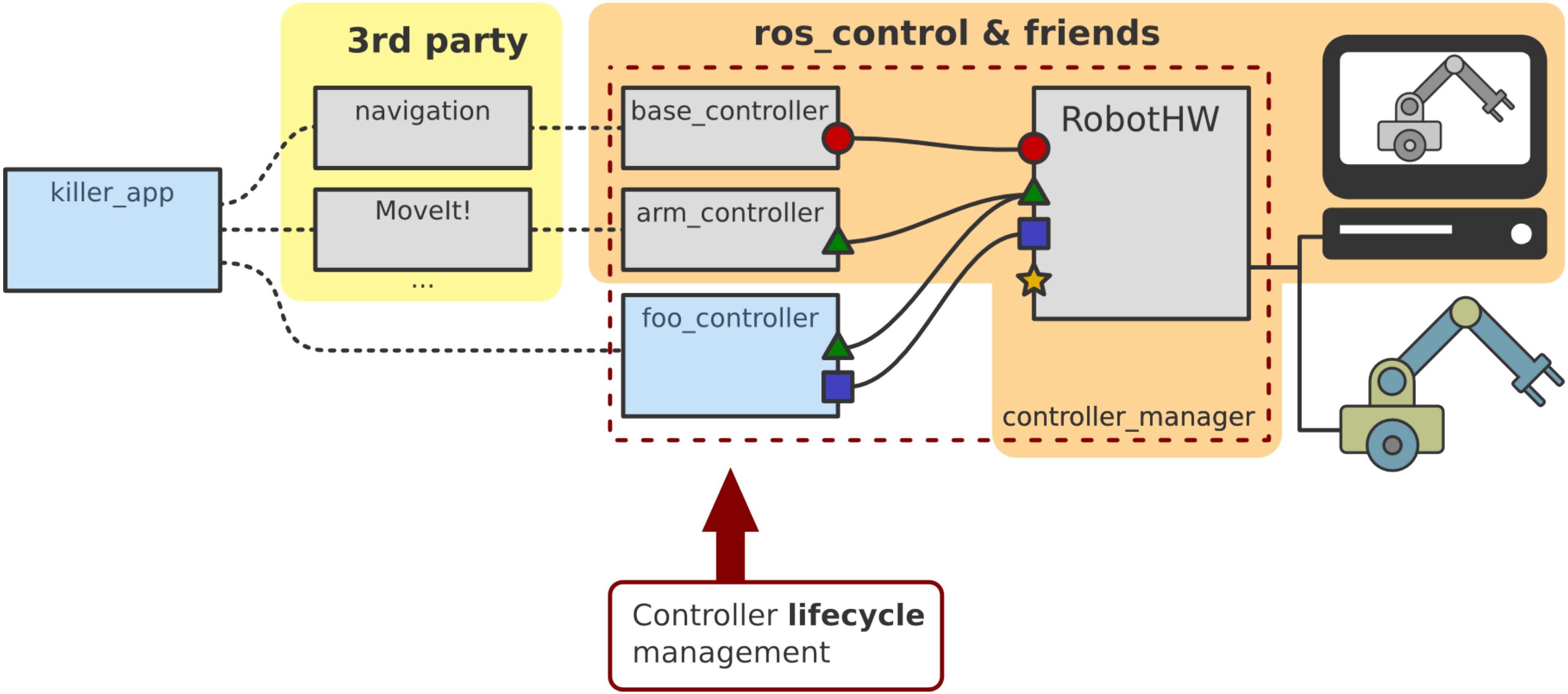
- **Leverage** simulation backend
- **Custom** hardware backend using **ready-to-use** blocks

ROS control – the big picture

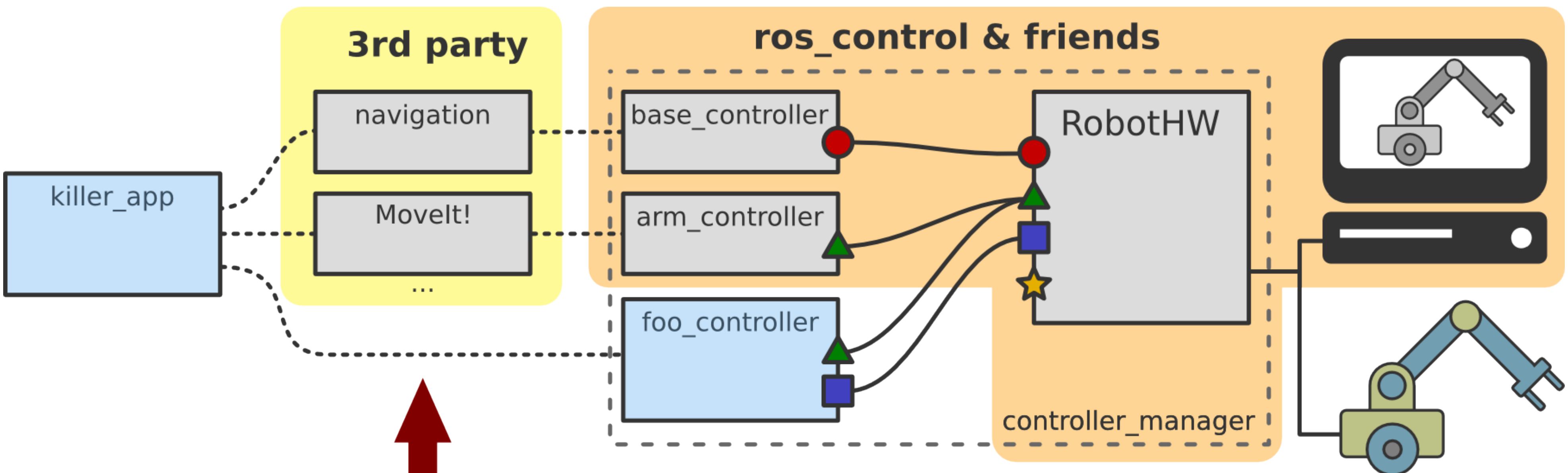


- **Leverage** existing controllers
- Implement **custom ones**

ROS control – the big picture

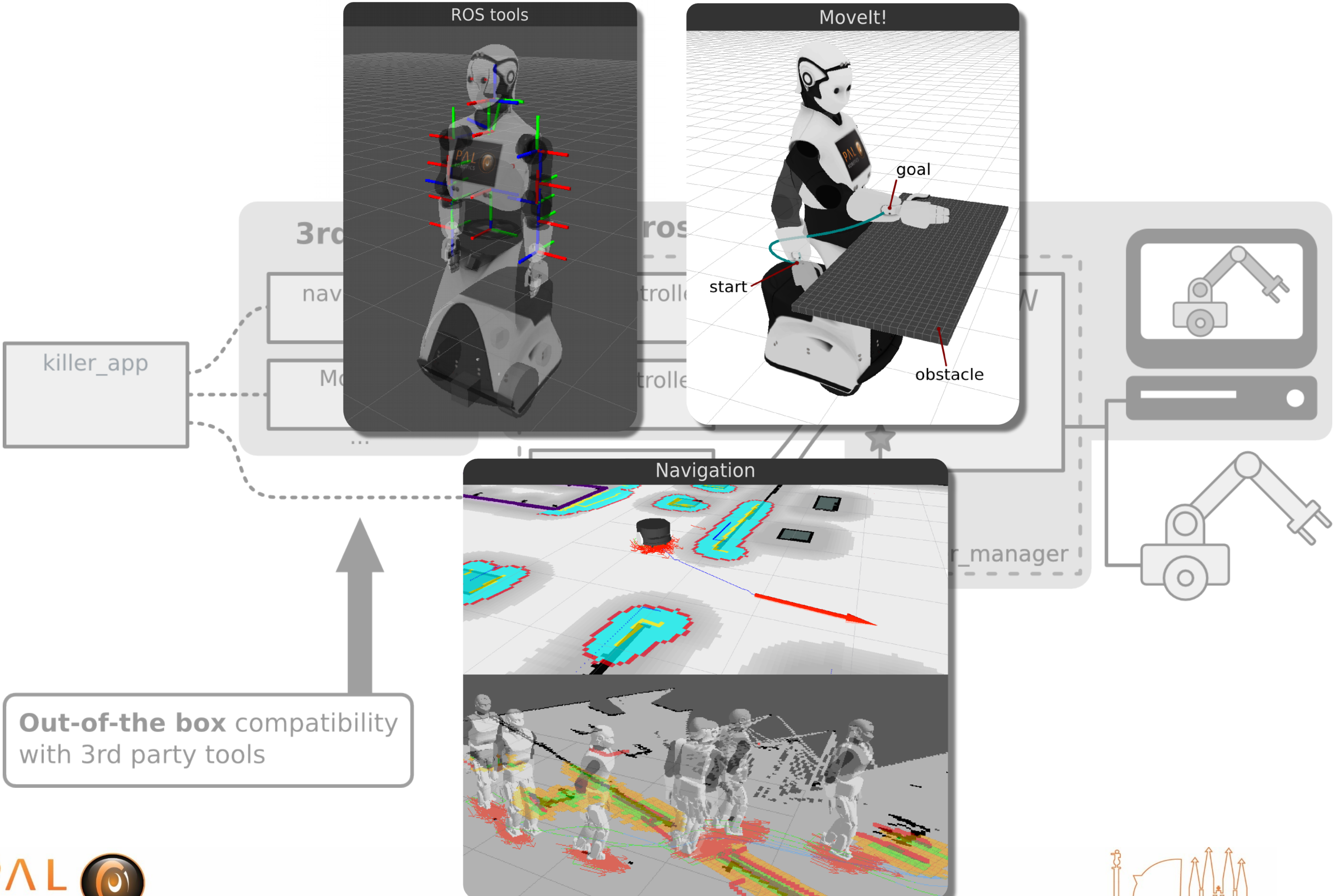


ROS control – the big picture

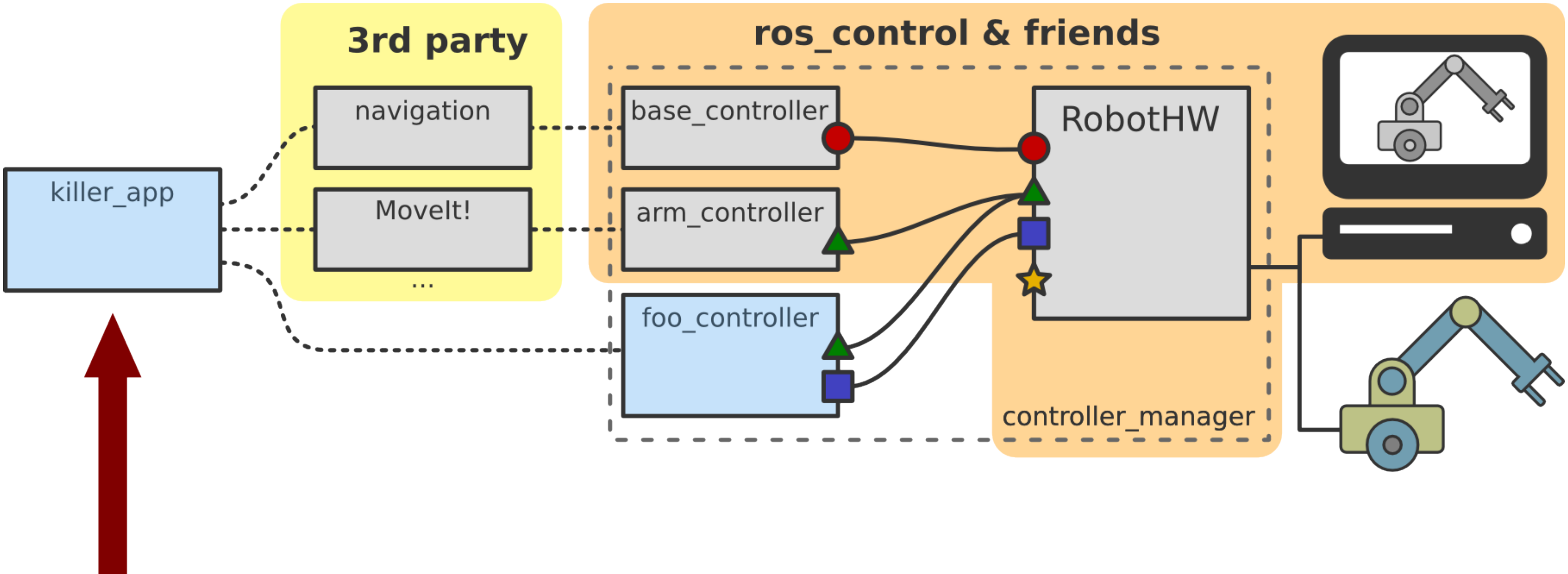


Out-of-the box compatibility with 3rd party tools

ROS control – the big picture

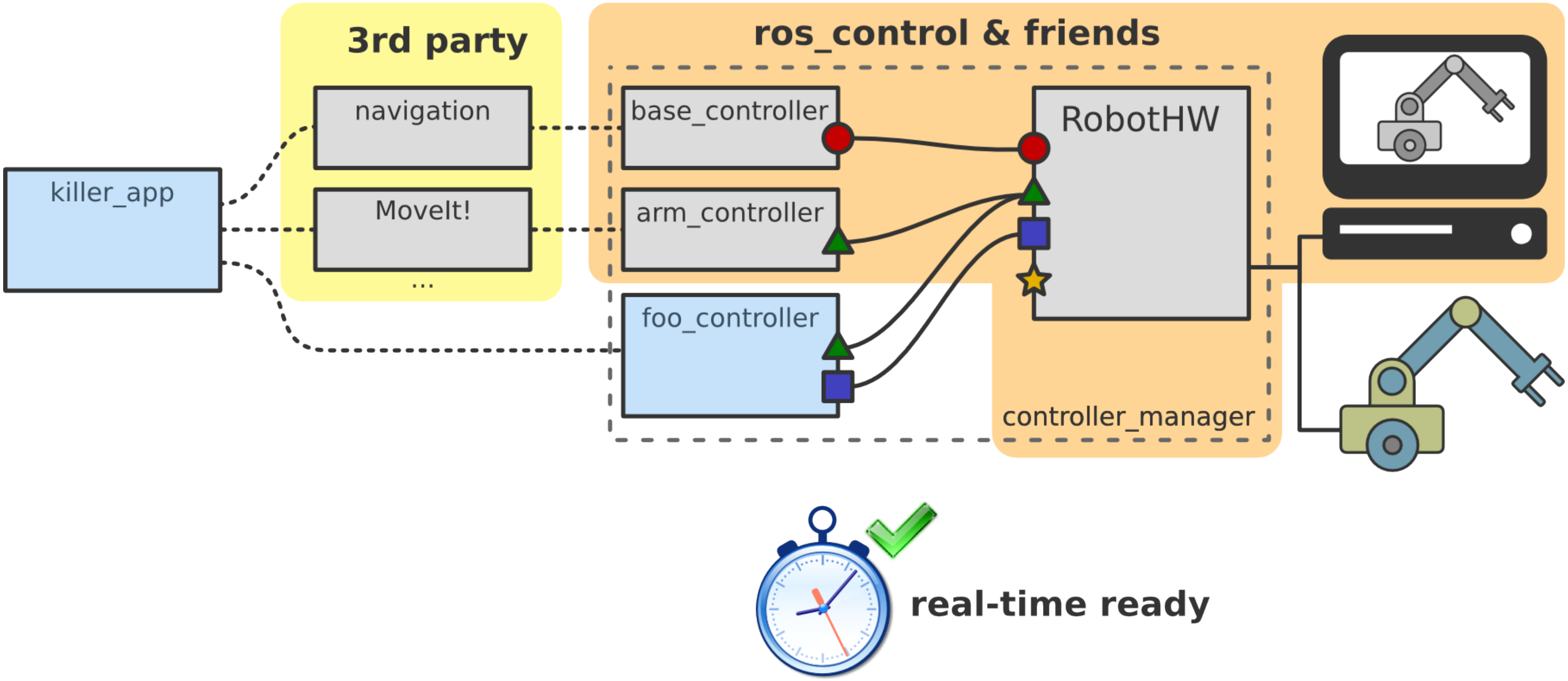


ROS control – the big picture



Maximize resources spent on the actual **application**

ROS control – the big picture



Real-time ready

- **Compatible** with real-time deployments
- **Not imposed**, use if needed
- RTOS choice is **up to you**
  - **PREEMPT-RT** extension
  - **Xenomai** co-kernel
  - **Proprietary:** VxWorks, QNX, etc.
  - ...

PAL ROBOTICS

ROS control – goals

- Lower **entry barrier** for exposing HW to ROS
- Promote **reuse** of control code
- Provide **ready-to-use** tools
  - **Simulation backend** for Gazebo
  - **Controller lifecycle** management
  - **Controllers** with standard ROS interfaces
  - **Building blocks** for creating new robots & controllers
  - **Tools** for user interaction
- **Real-time ready** implementation

code repositories

ros-controls

github.com/ros-controls

control_msgs

messages and actions useful for controlling robots

realtime_tools

tools that can be used from a hard realtime thread

control_toolbox

tools useful for writing controllers and robot abstractions

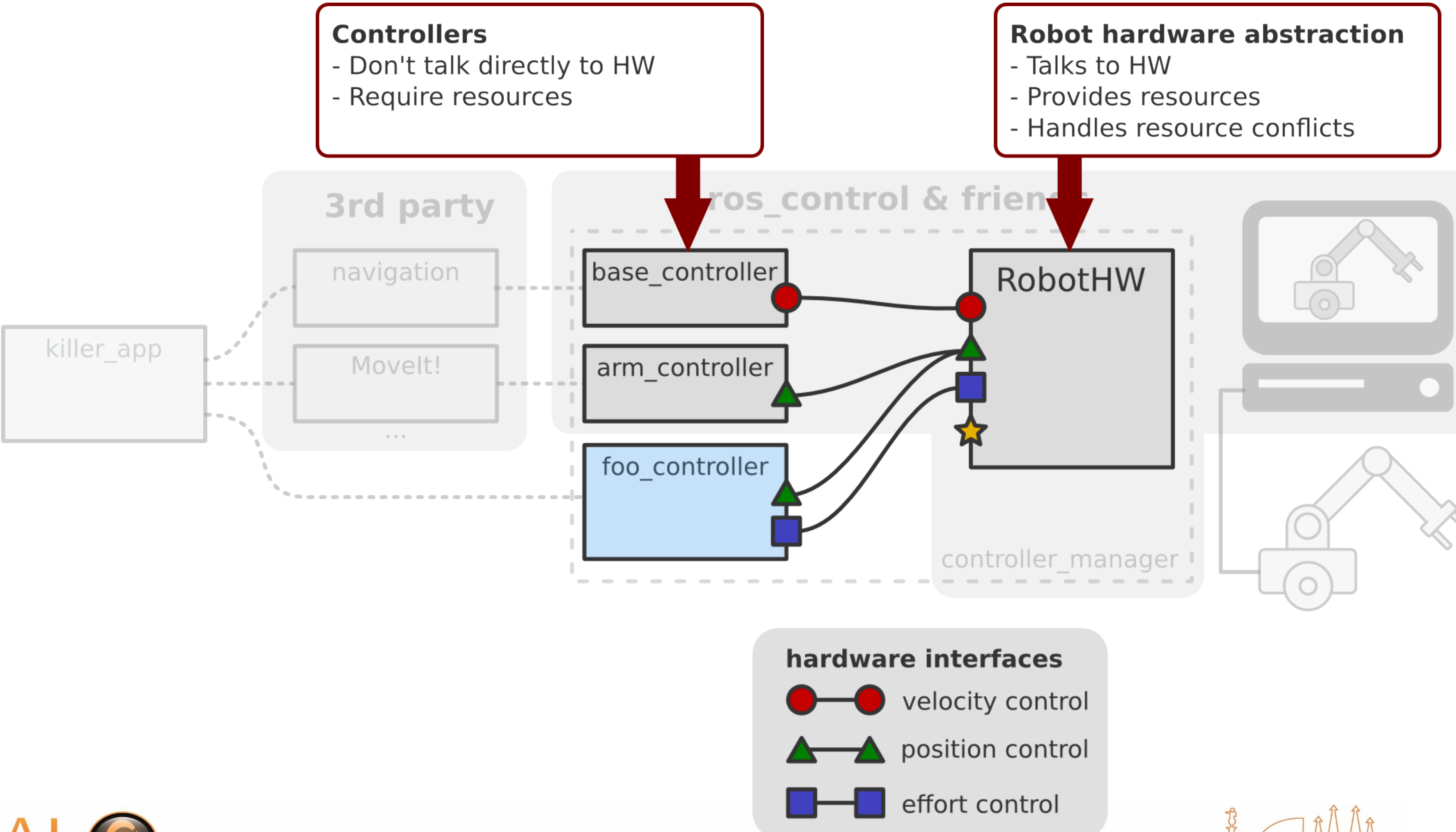
ros_control

generic and basic controller framework for ROS

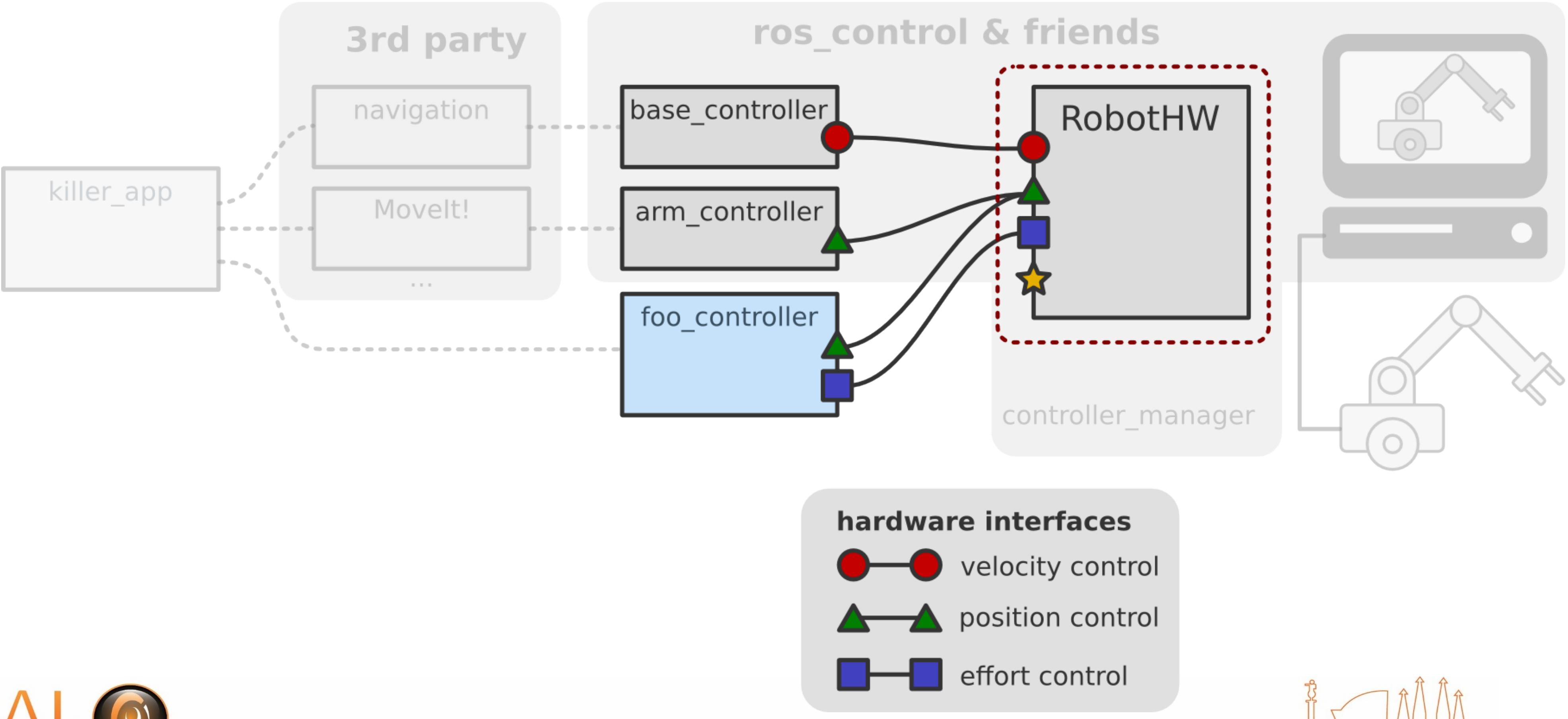
ros_controllers

generic robot controllers for ros_control

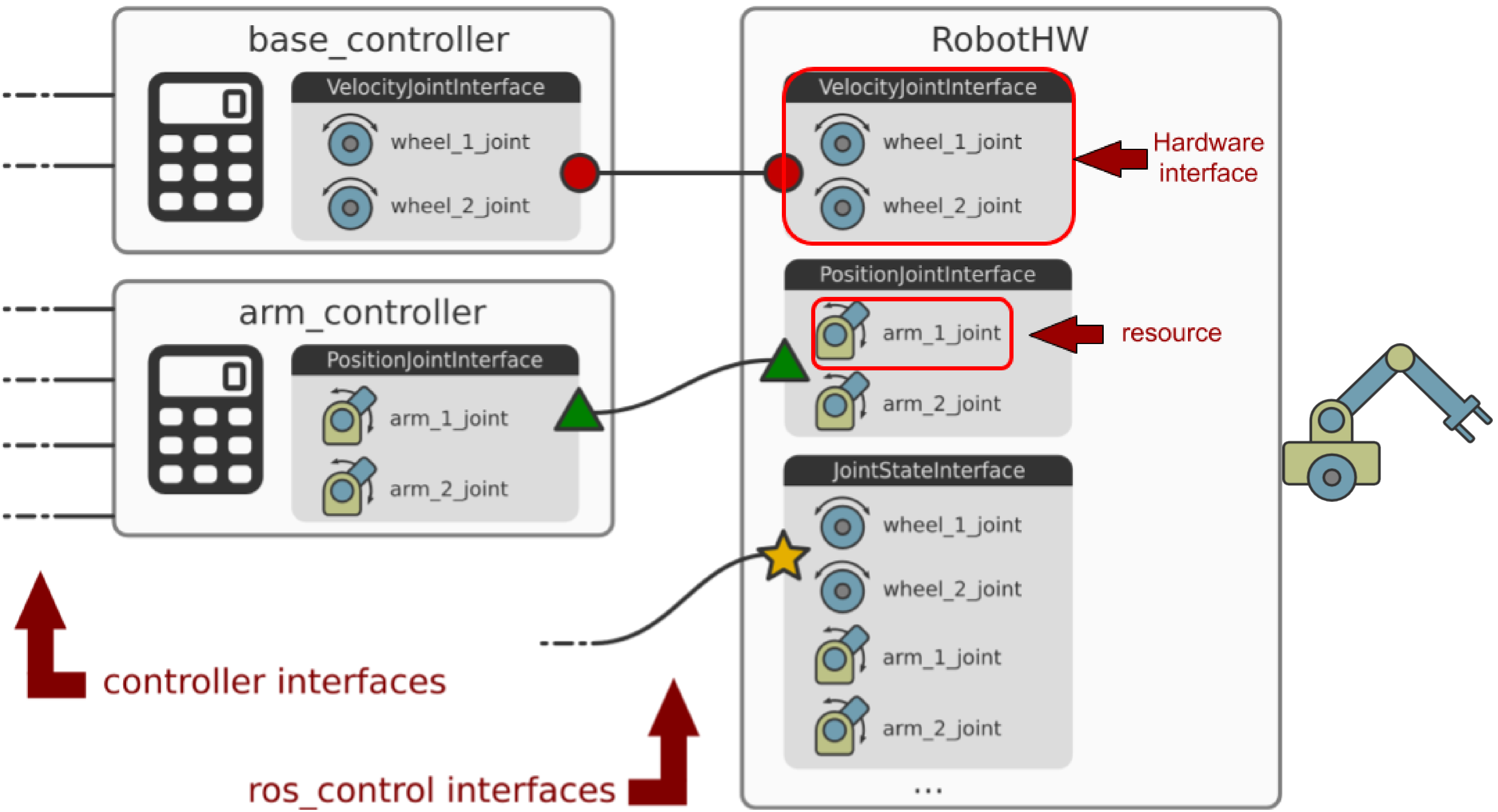
Setting up a robot



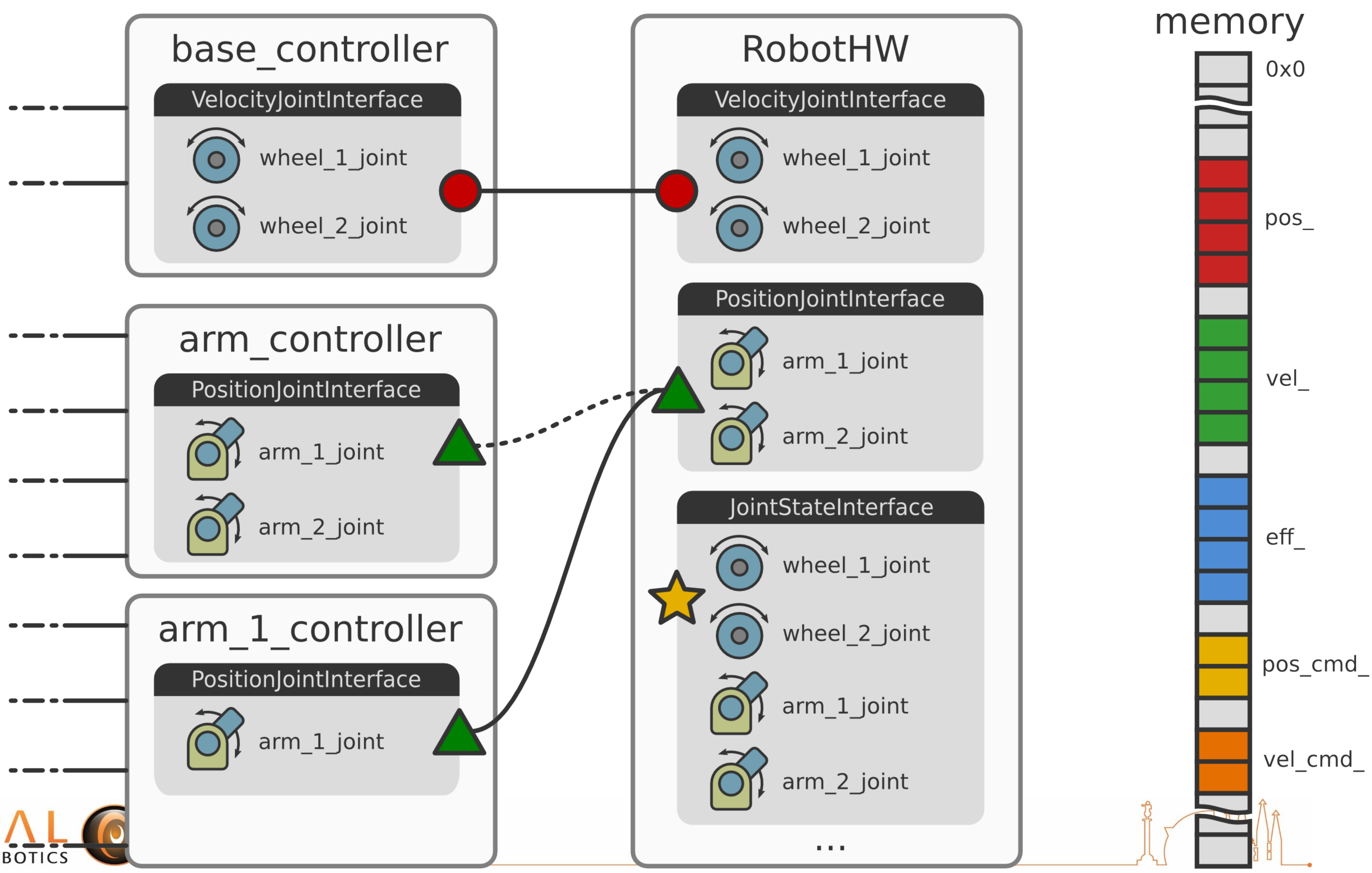
Setting up a robot



Setting up a robot

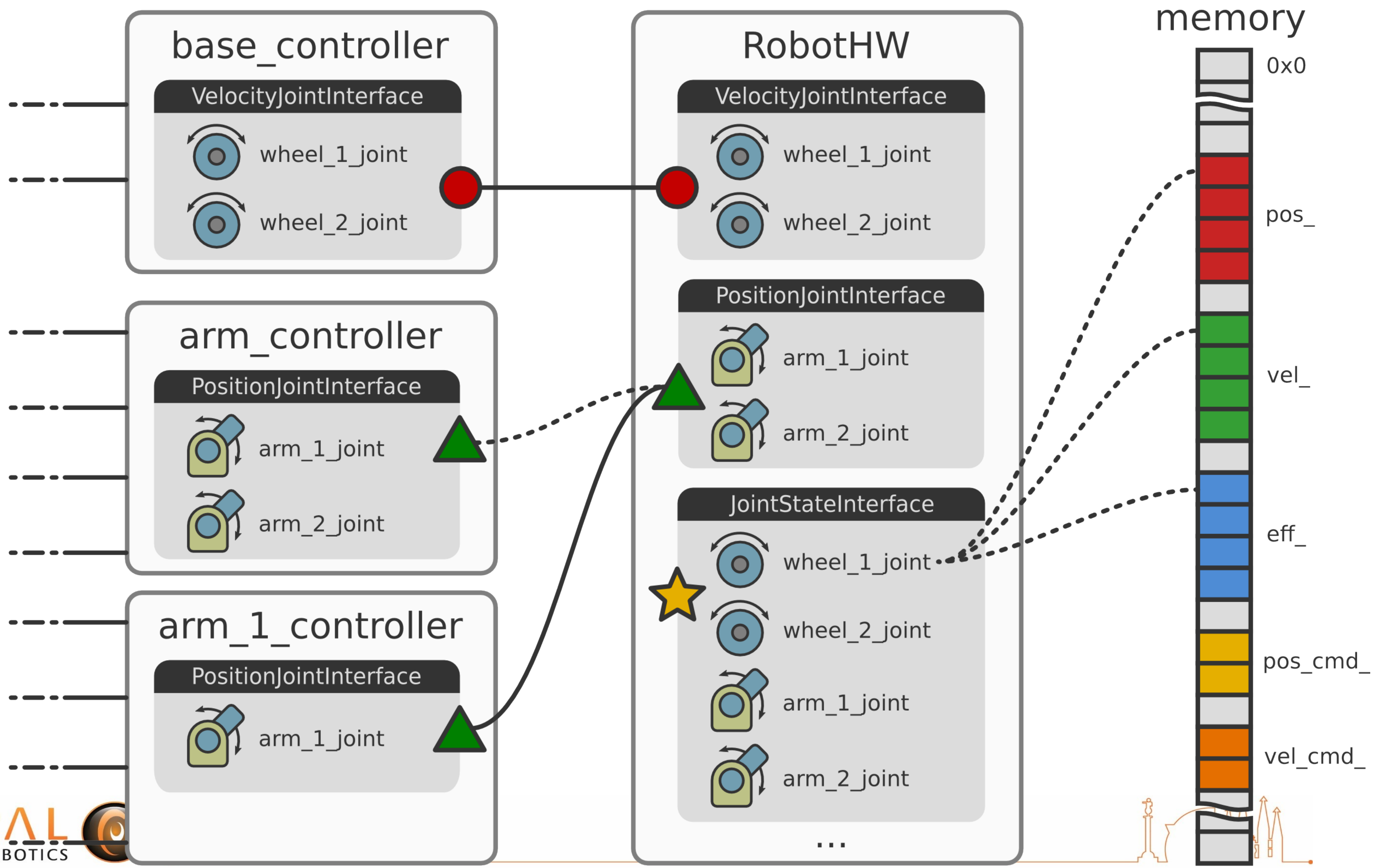


Setting up a robot

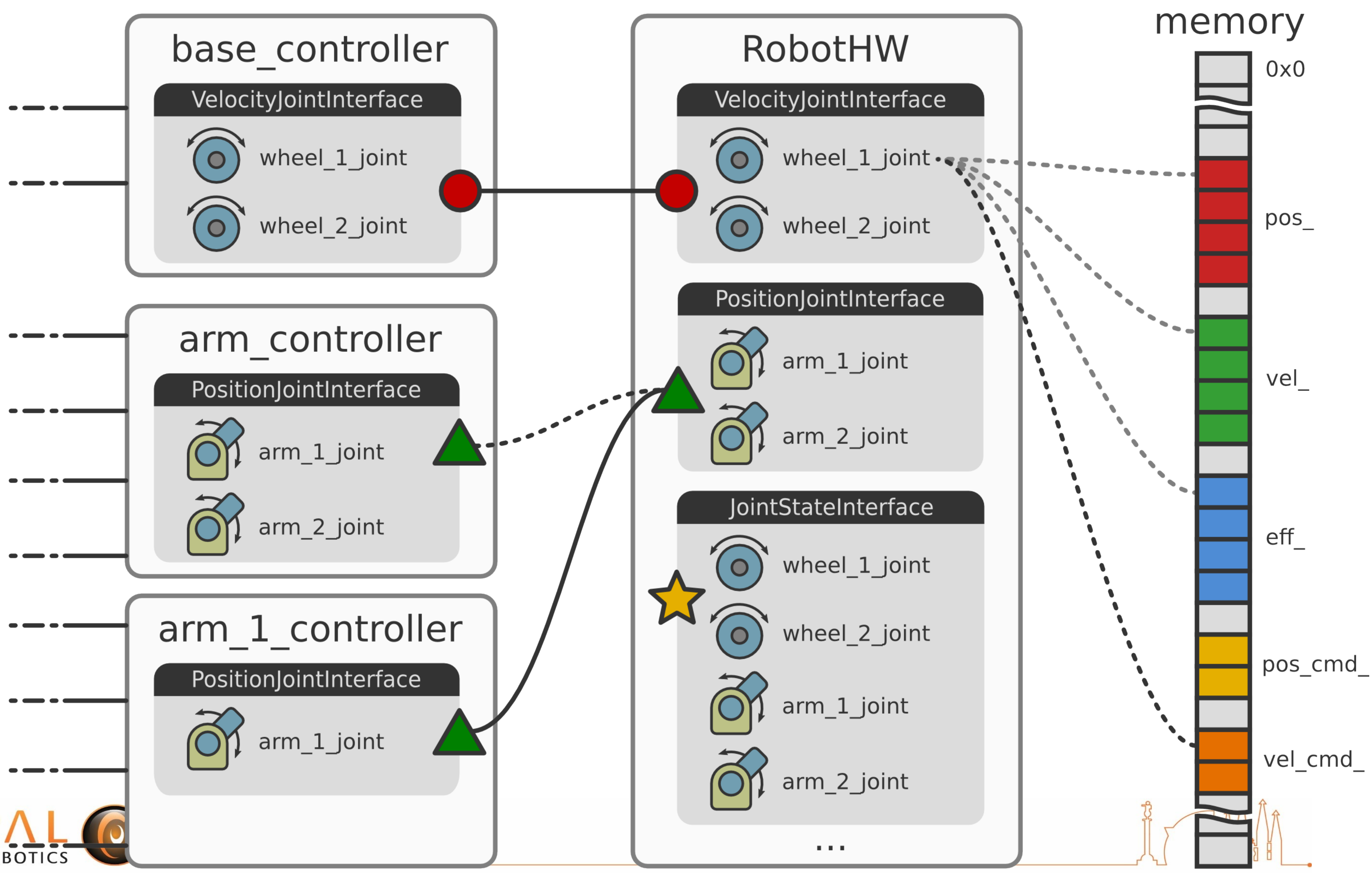


PAL ROBOTICS

Setting up a robot



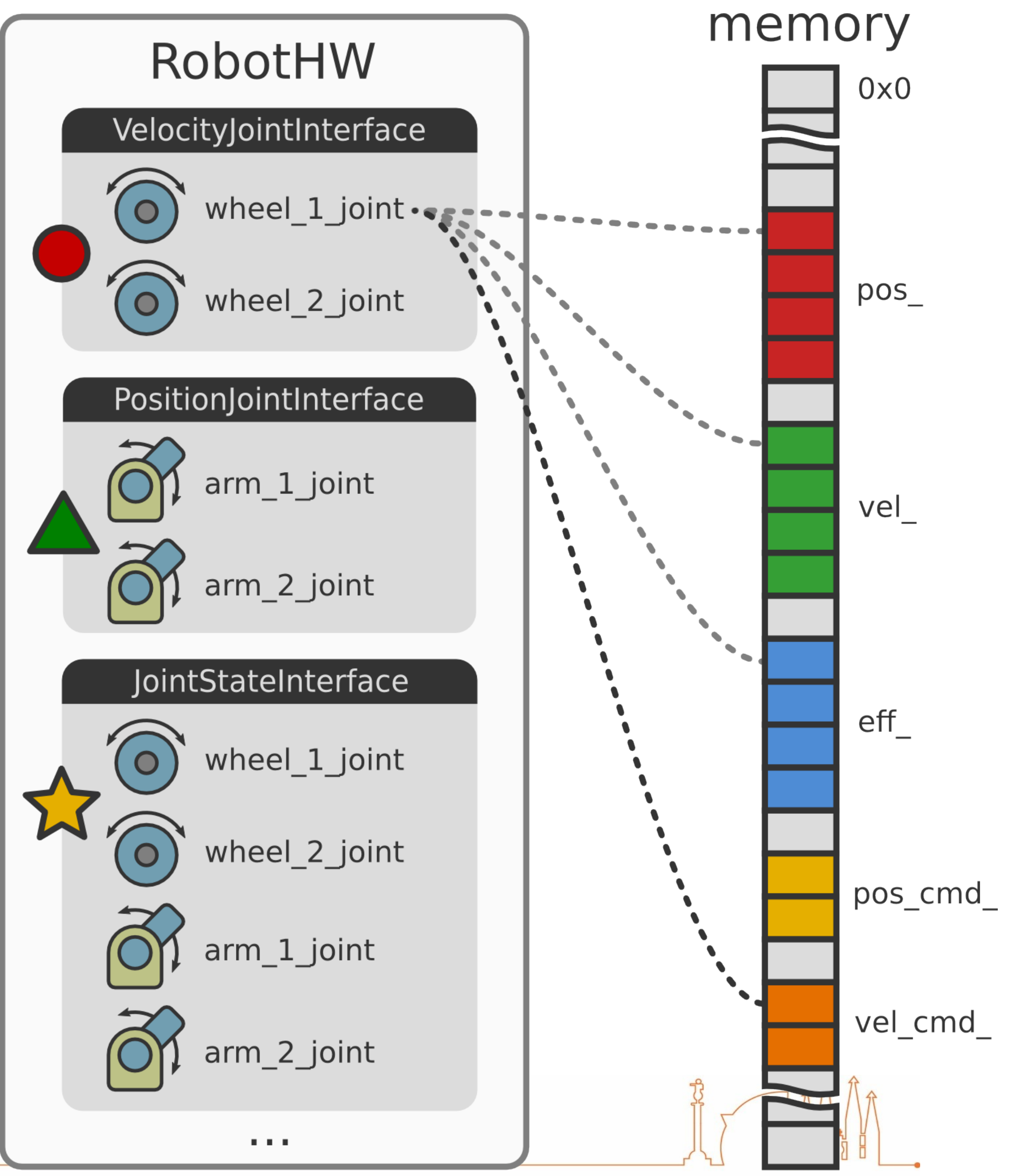
Setting up a robot



PAL ROBOTICS

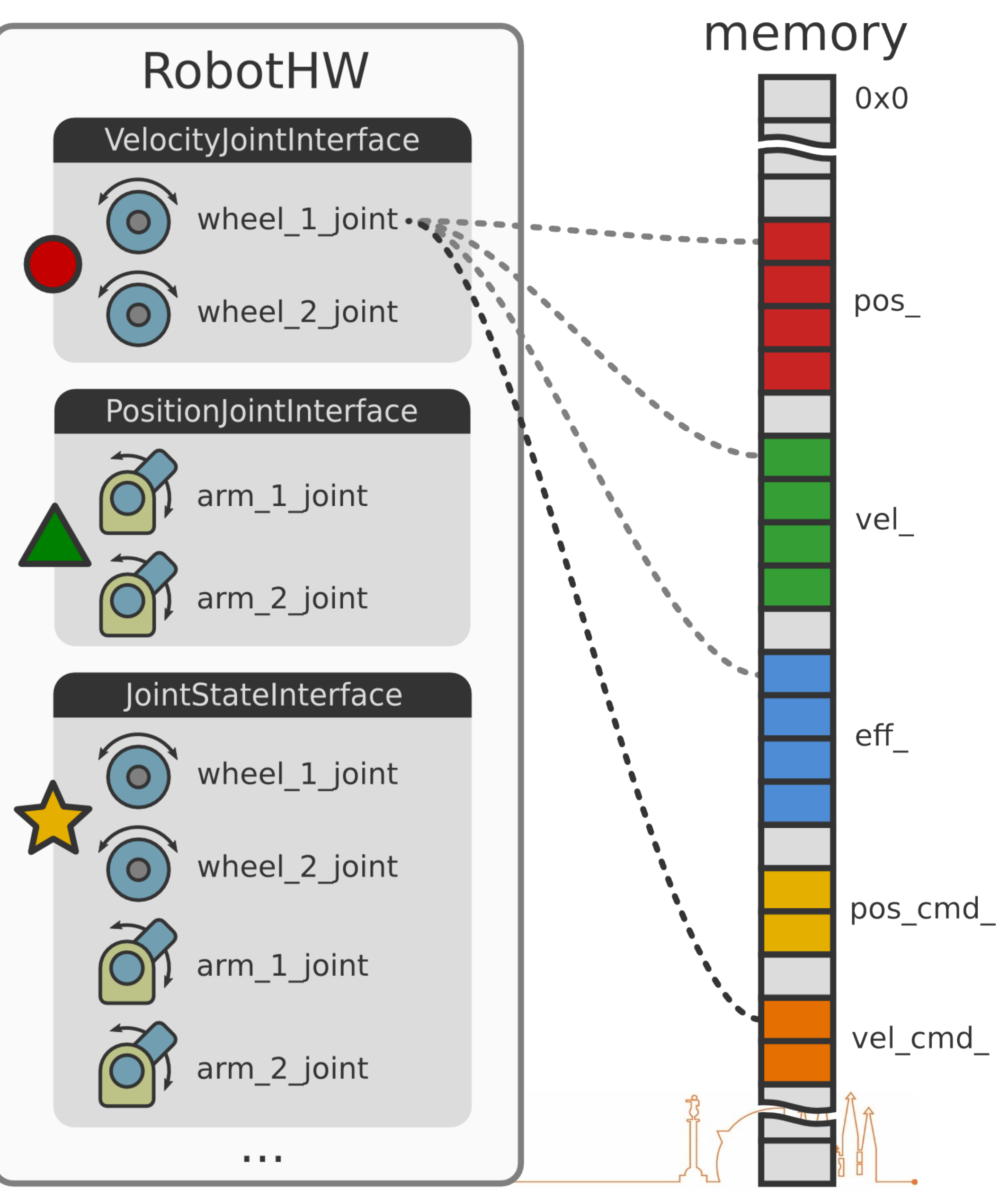
Setting up a robot

```
class MyRobot :
  public hardware_interface::RobotHW
{
public:
  MyRobot(); // Setup robot

  // Talk to HW
  void read();
  void write();

  // Reimplement only if needed
  virtual bool checkForConflict(...) const;
};
```



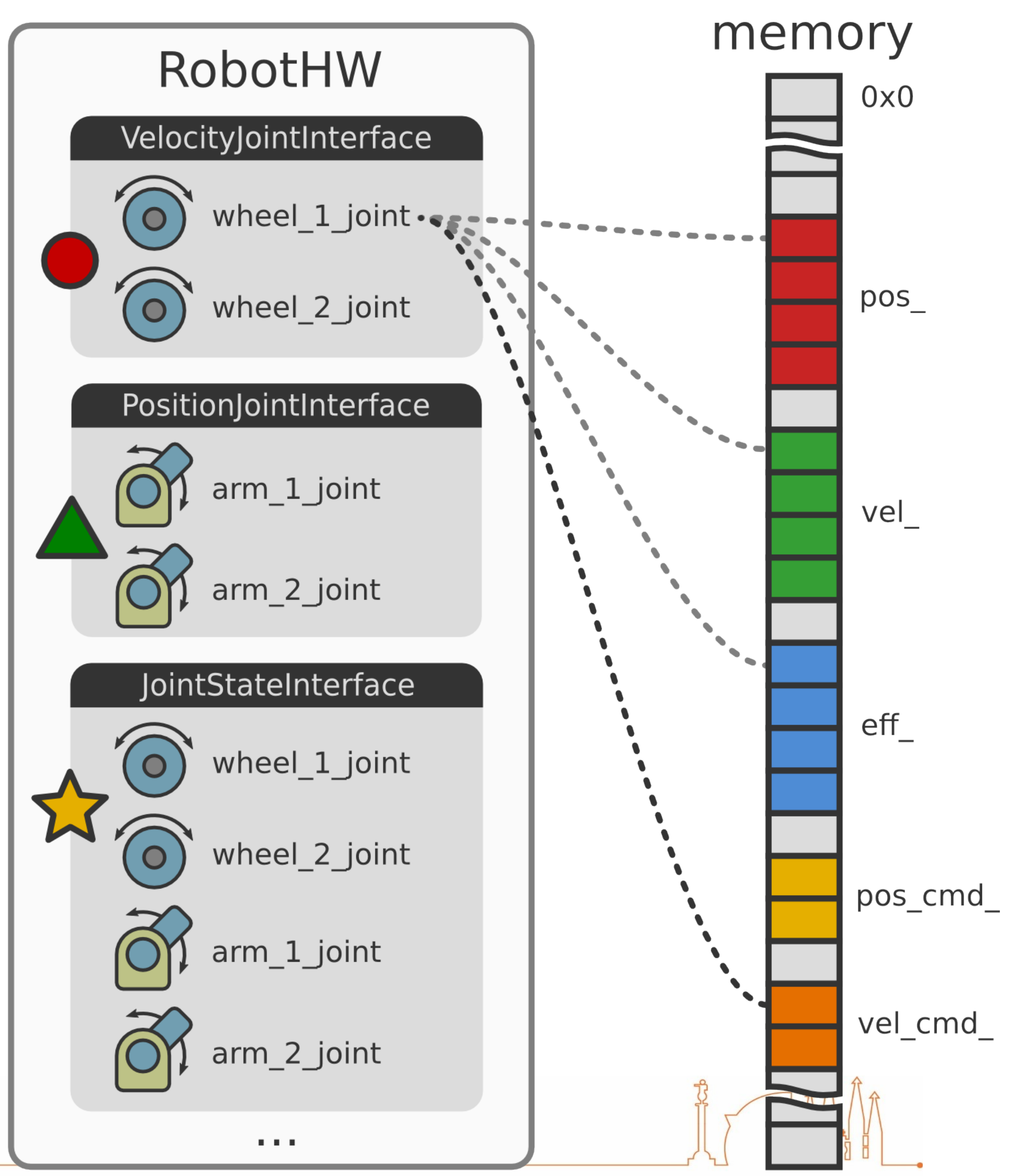
Setting up a robot

```
class MyRobot :
  public hardware_interface::RobotHW
{
public:
  MyRobot(); // Setup robot

  // Talk to HW
  void read();
  void write();

  // Reimplement only if needed
  virtual bool checkForConflict(...) const;
};
```



Setting up a robot

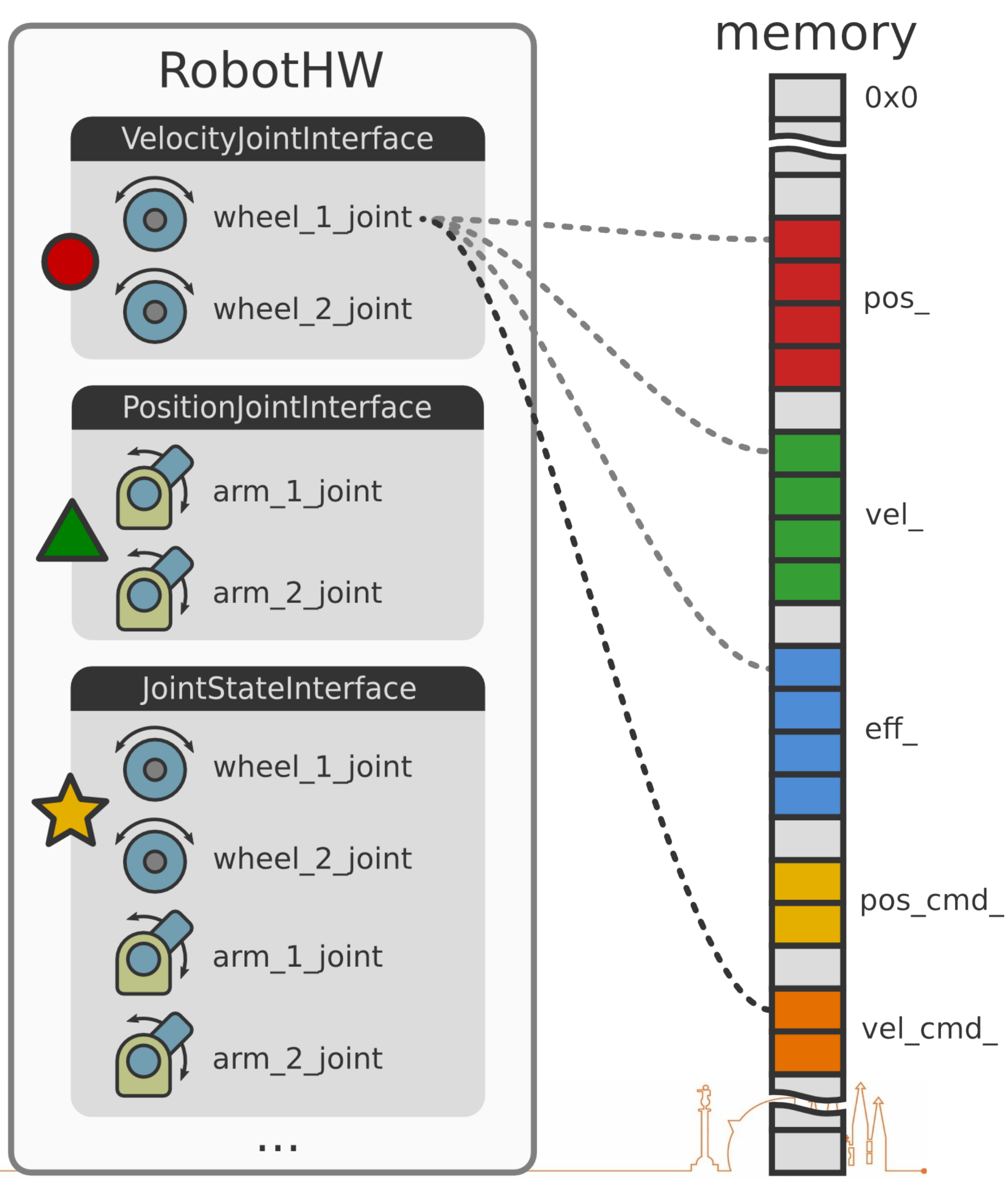
```
class MyRobot :
  public hardware_interface::RobotHW
{
public:
  MyRobot(); // Setup robot

  // Talk to HW
  void read();
  void write();

  // Reimplement only if needed
  virtual bool checkForConflict(...) const;
};
```



Setting up a robot

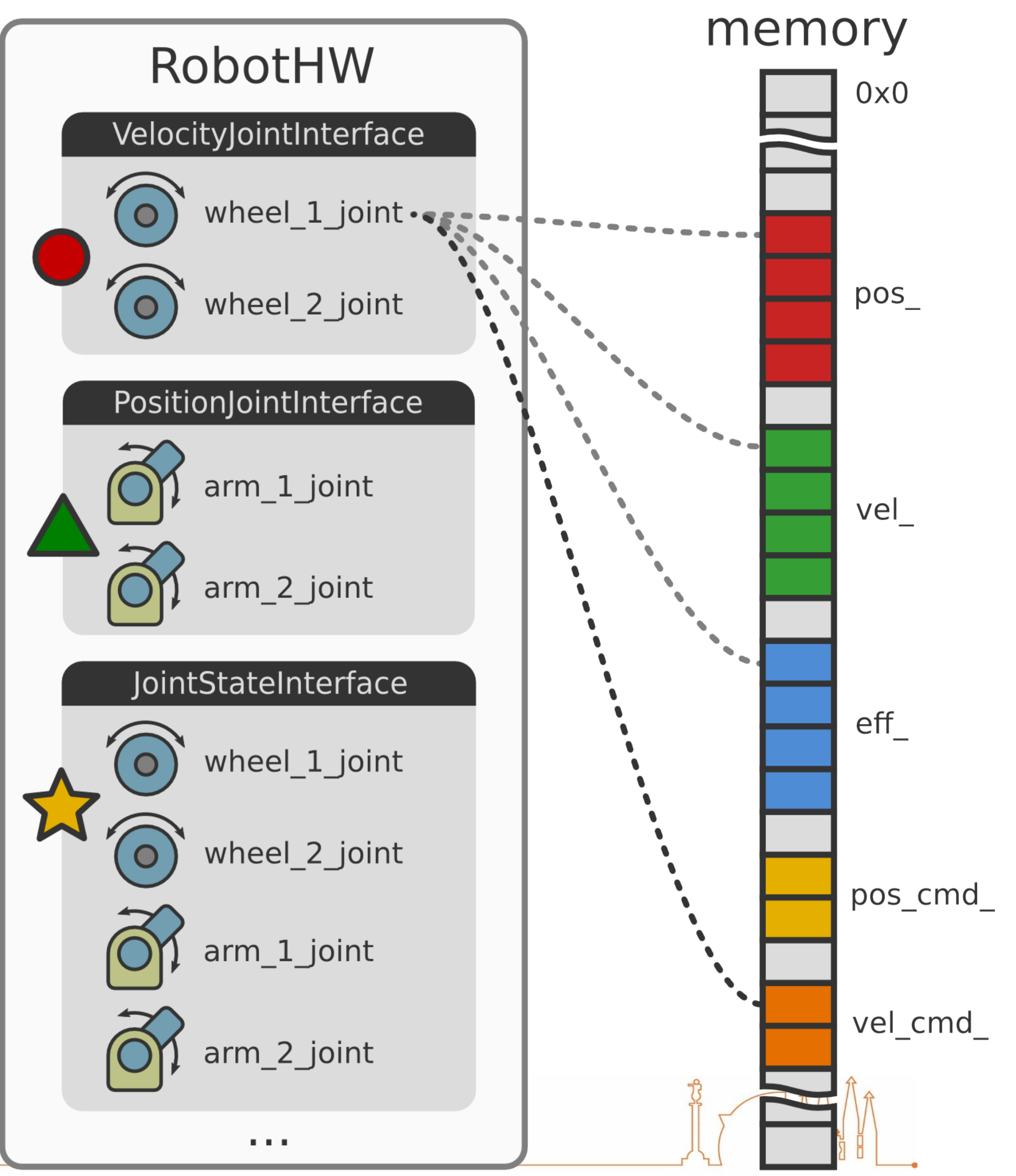
```
class MyRobot :
  public hardware_interface::RobotHW
{
public:
  MyRobot(); // Setup robot

  // Talk to HW
  void read();
  void write();

  // Reimplement onl
  virtual bool checkForConflict(...) const;
};
```

- ROS industrial
- rosserial
- SR RONEX
- custom
- ...



Setting up a robot

```
class MyRobot :
  public hardware_interface::RobotHW
{
public:
  MyRobot(); // Setup robot

  // Talk to HW
  void read();
  void write();

  // Reimplement only if needed
  virtual bool checkForConflict(...) const;
};
```



PAL ROBOTICS

hardware_interface

Robot hardware abstraction

- **Software** representation of robot
- Abstracts **hardware** away
  - **Resource:** actuators, joints, sensors
  - **Interface:** Set of similar resources
  - **Robot:** Set of interfaces
- Handles **resource conflicts**
  - **Exclusive** ownership by default

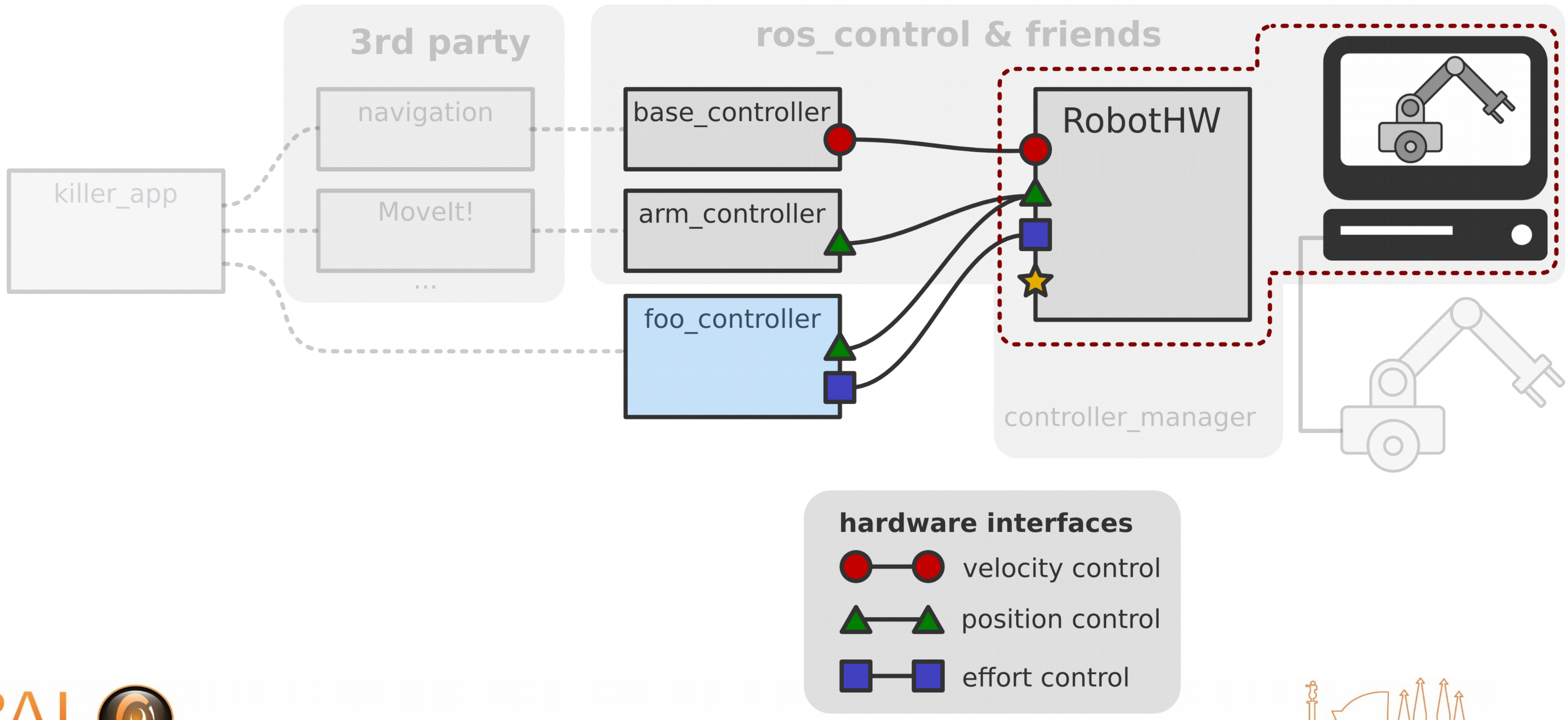
hardware_interface

Resources and interfaces

- Read-only
  - Joint state*
  - IMU
  - Force-torque sensor
- Read-write
  - Position joint*
  - Velocity joint*
  - Effort joint*

* Equivalent interfaces exist for actuators

Setting up a robot in simulation



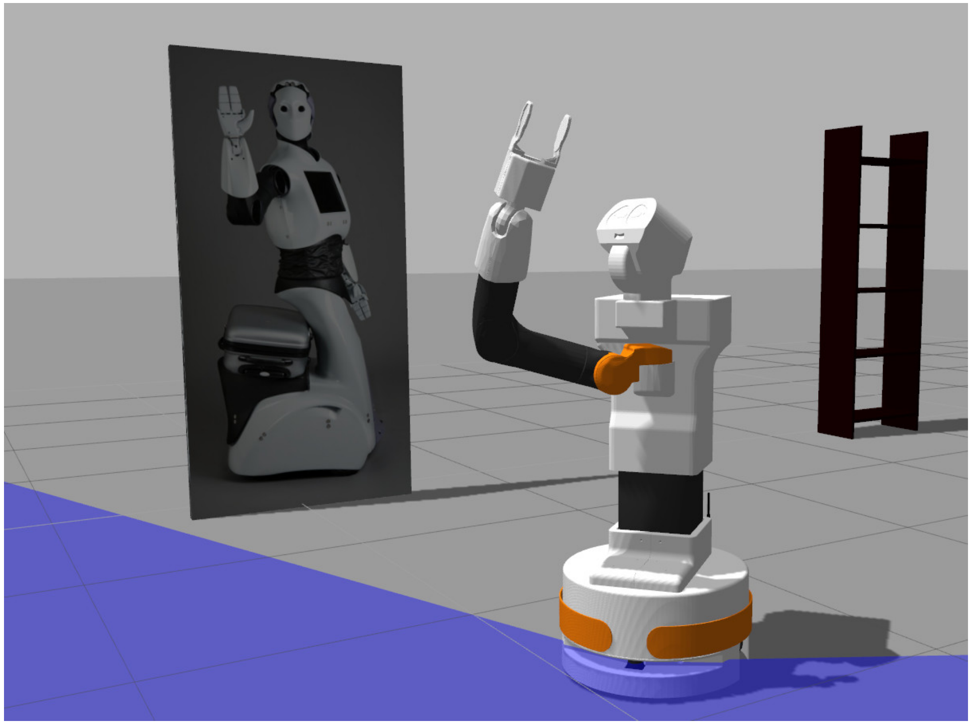
gazebo_ros_control

- Lives **outside** ros-controls repos
  - ros-simulation/**gazebo_ros_pkgs**
- **Gazebo** plugin for **ros_control**
  - **Default plugin:**
    - Populates **joint interfaces** from **URDF**
    - Reads **transmission** and **joint limits** specs
  - **Custom plugin:** Up to you

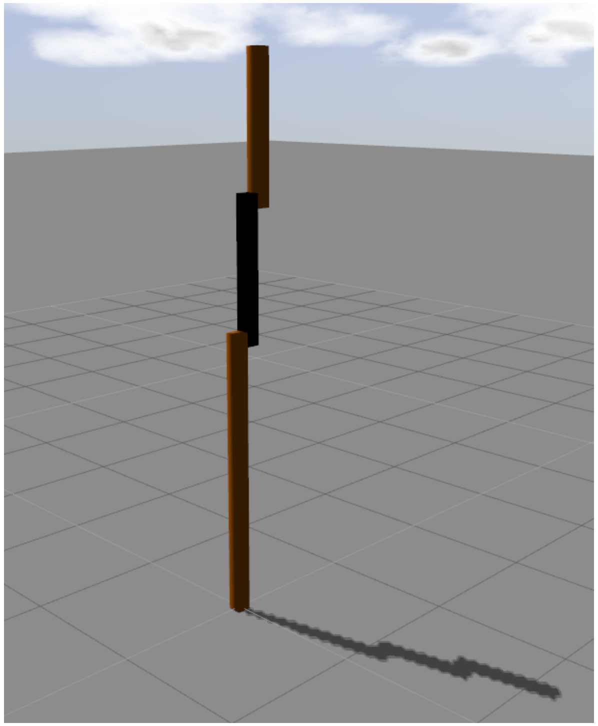
```
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/my_robot</robotNamespace>
  </plugin>
</gazebo>
```

gazebo_ros_control

- Test ros_control without coding a RobotHW!

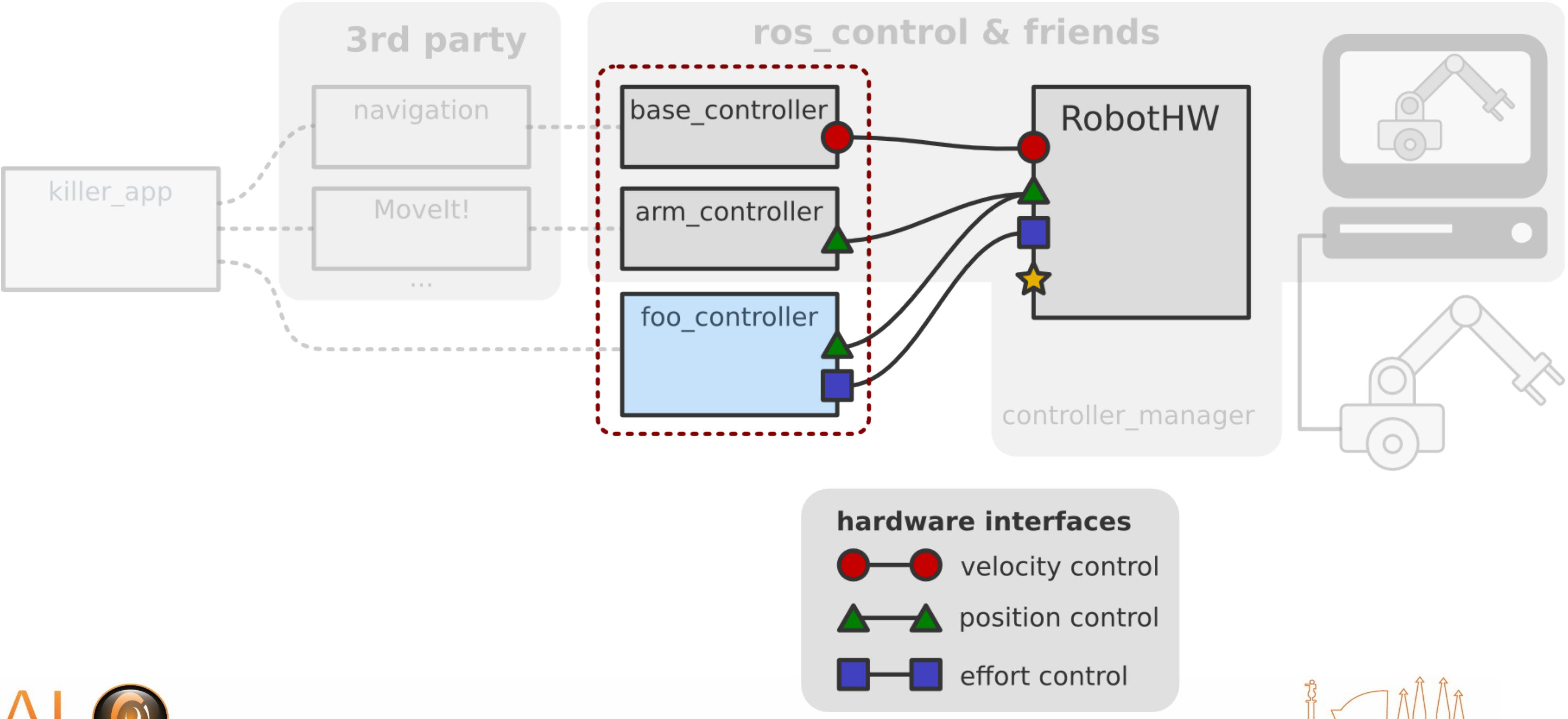


http://wiki.ros.org/Robots/TIAGo/Tutorials

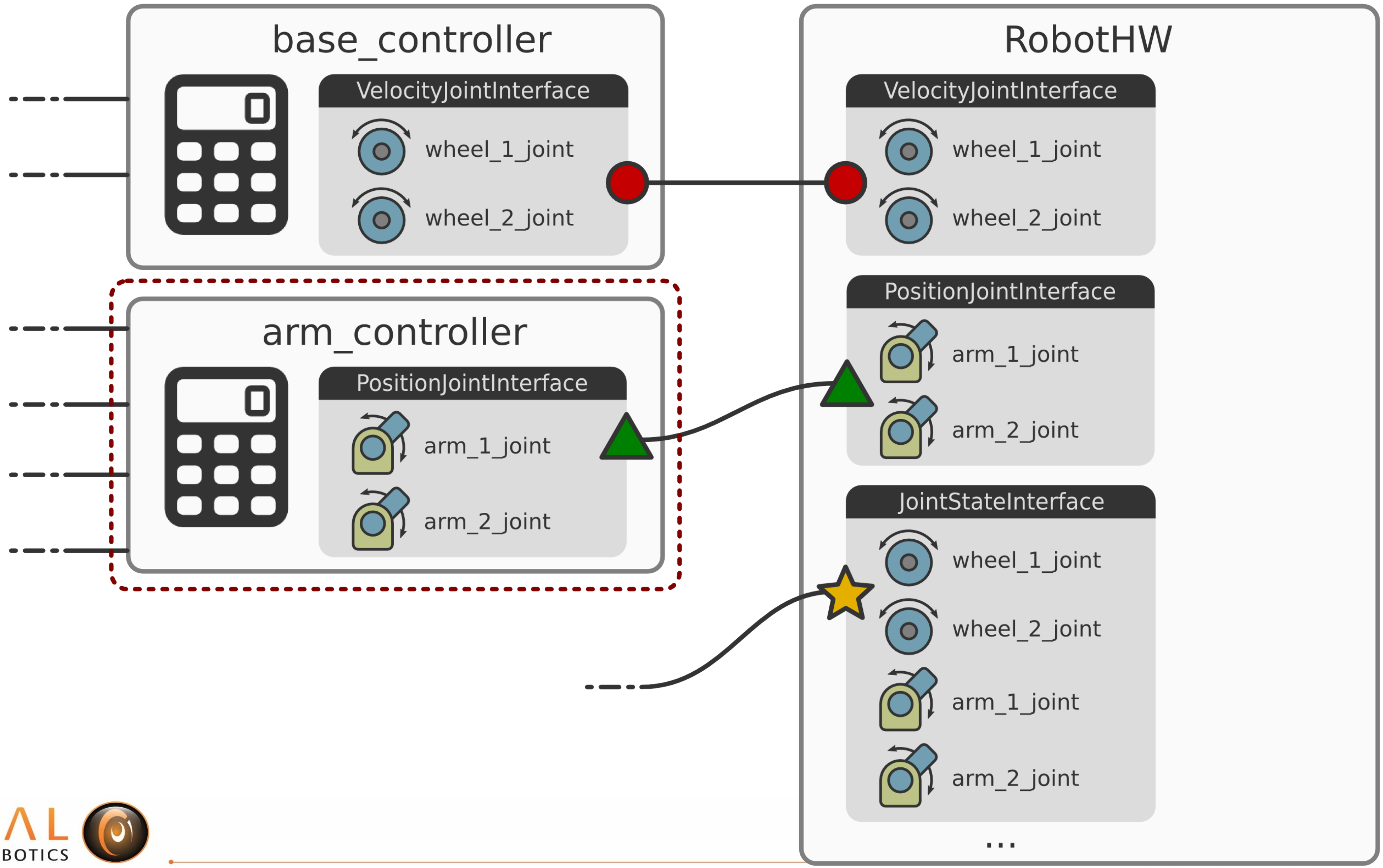


https://github.com/PickNikRobotics/ros_control_boilerplate.git

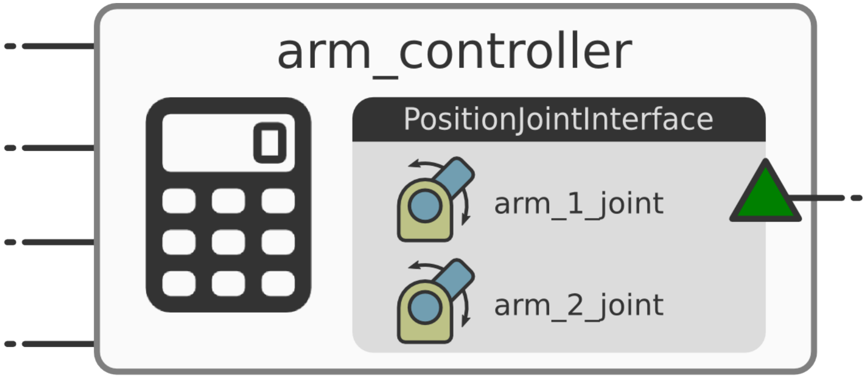
ROS Controllers



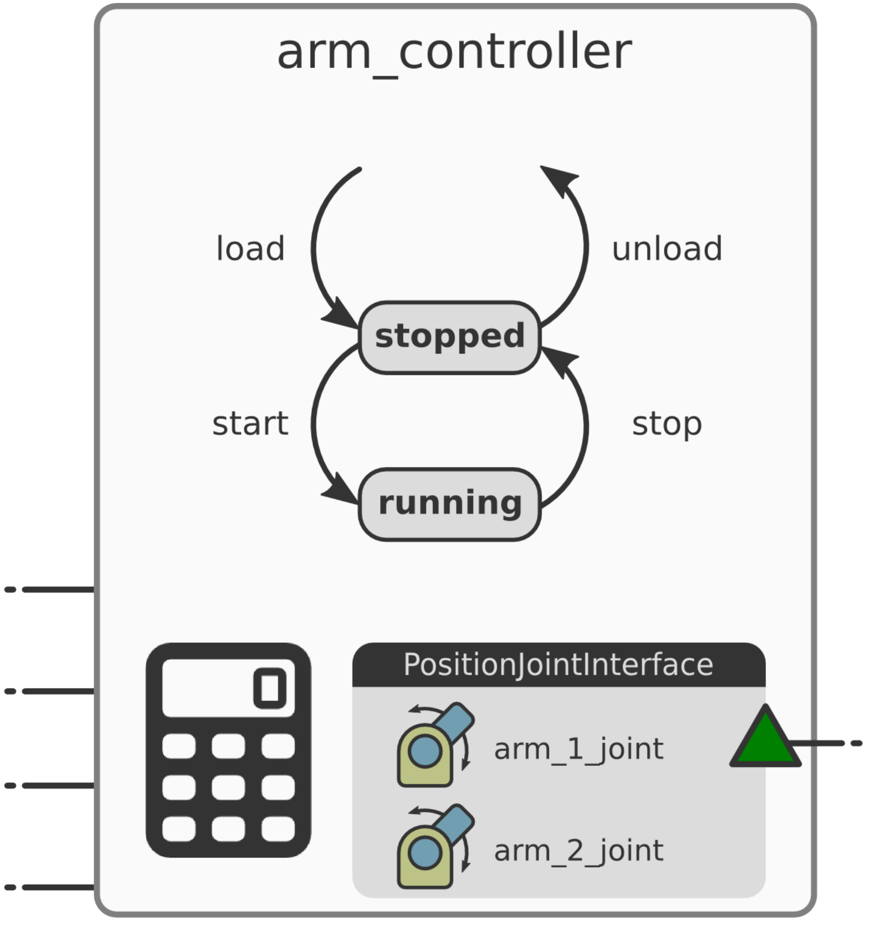
ROS Controllers



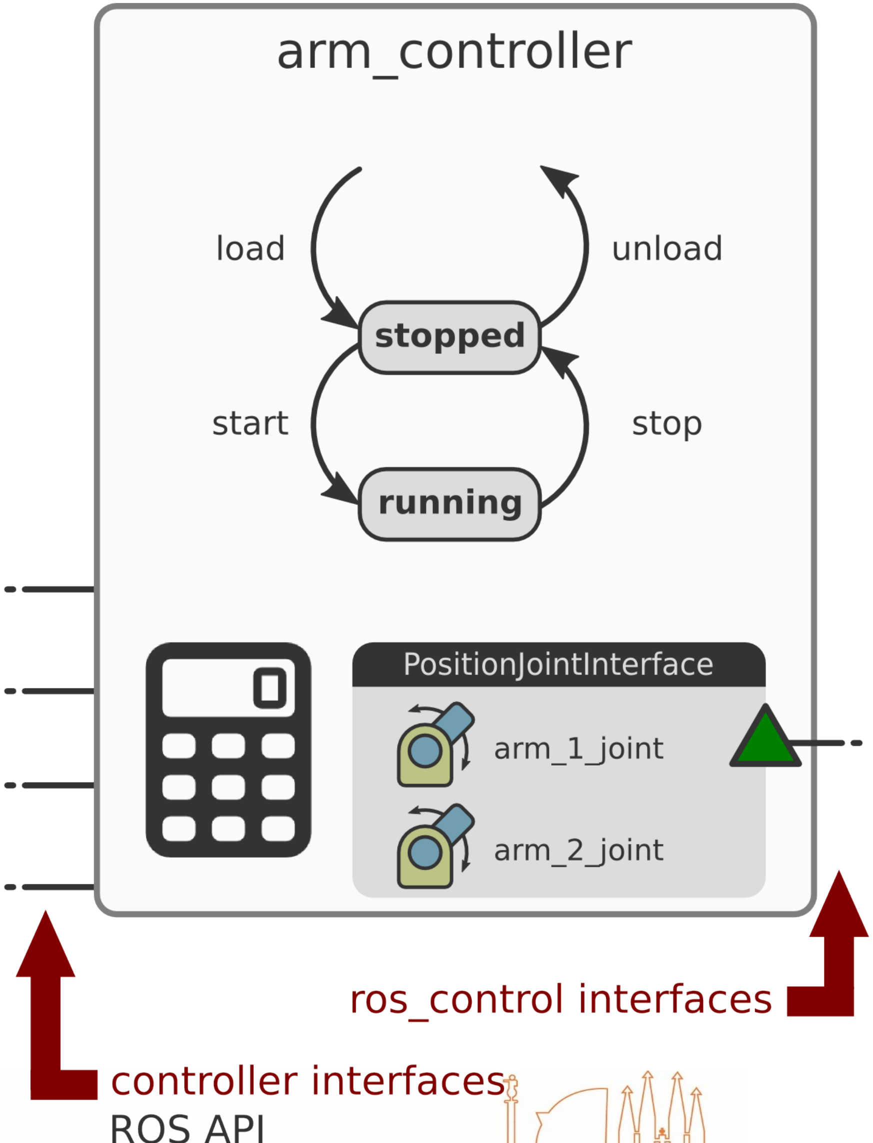
ROS Controllers



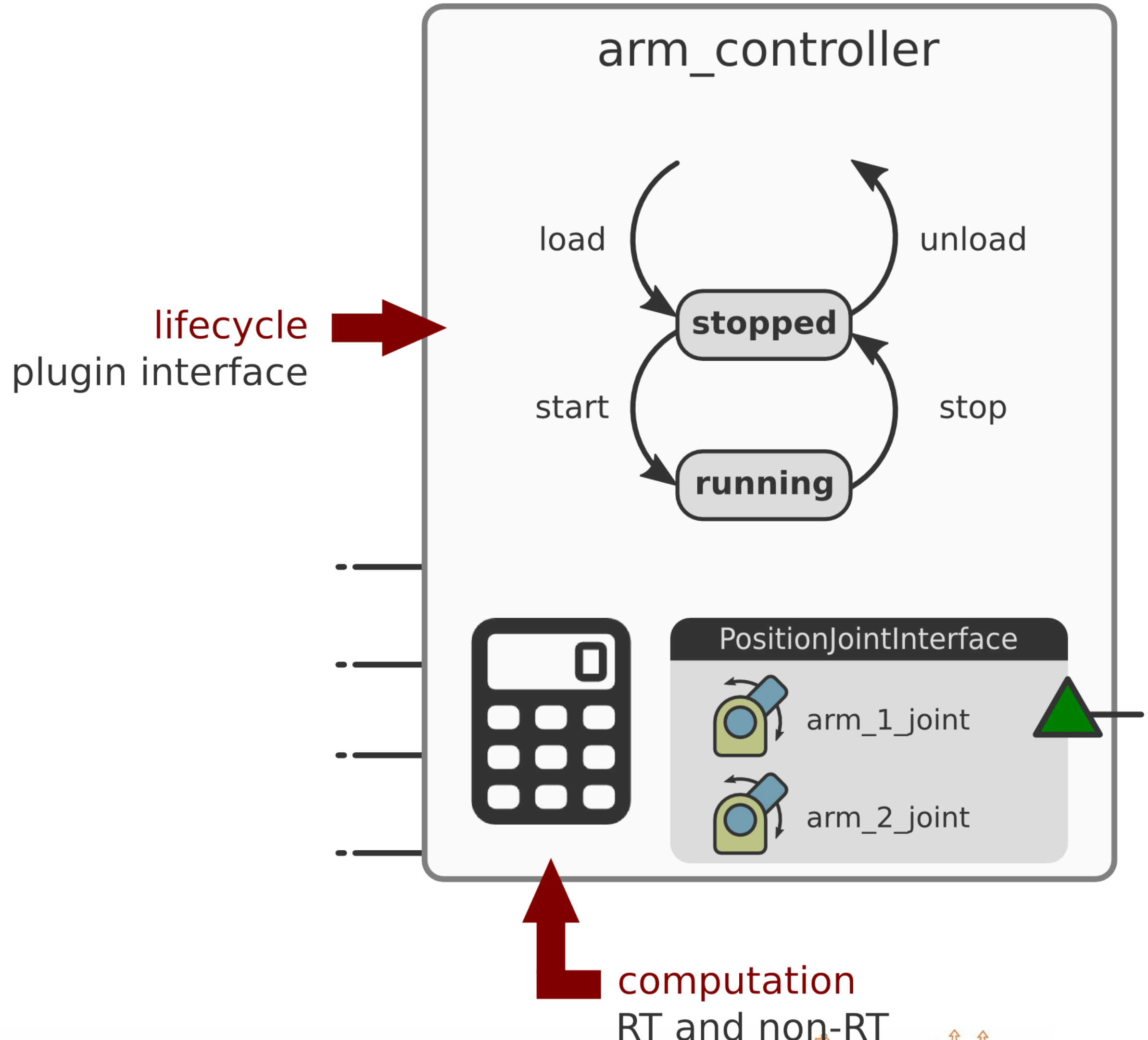
ROS Controllers



ROS Controllers



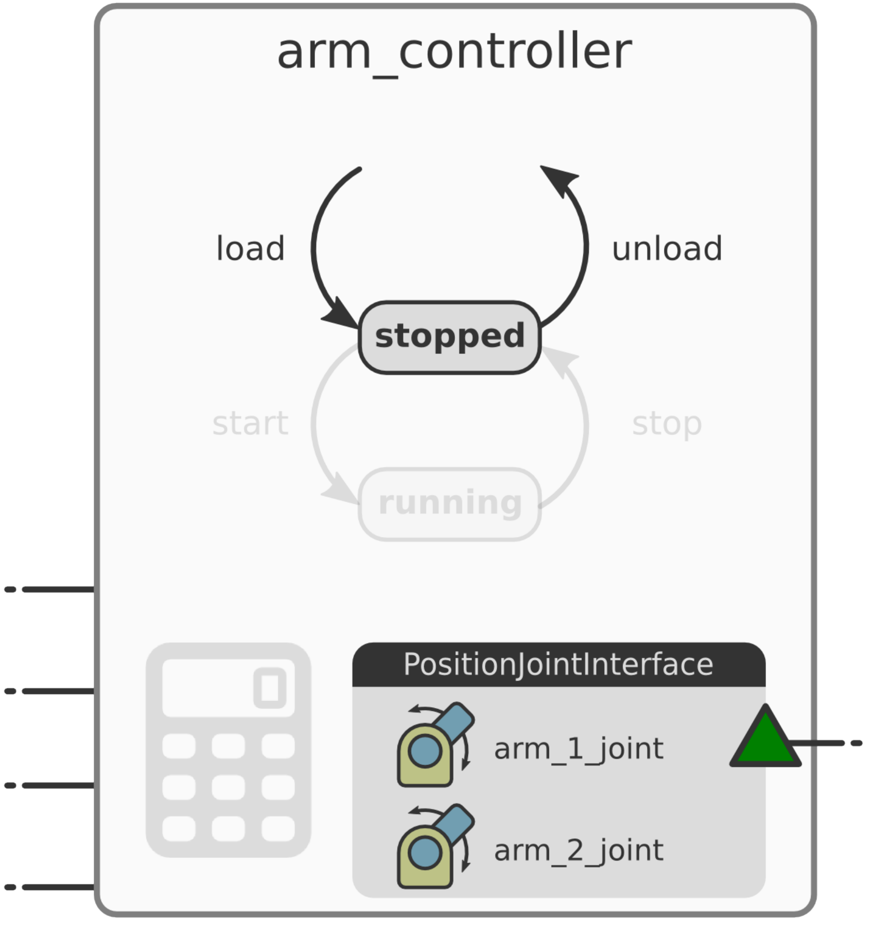
ROS Controllers



ROS Controllers

Non real-time operations

- **load**
  - load + initialize plugin
  - check requisites (can fail)
    - hardware resource **existence***
    - configuration
  - setup ROS interfaces
- **unload**
  - destroy + unload plugin

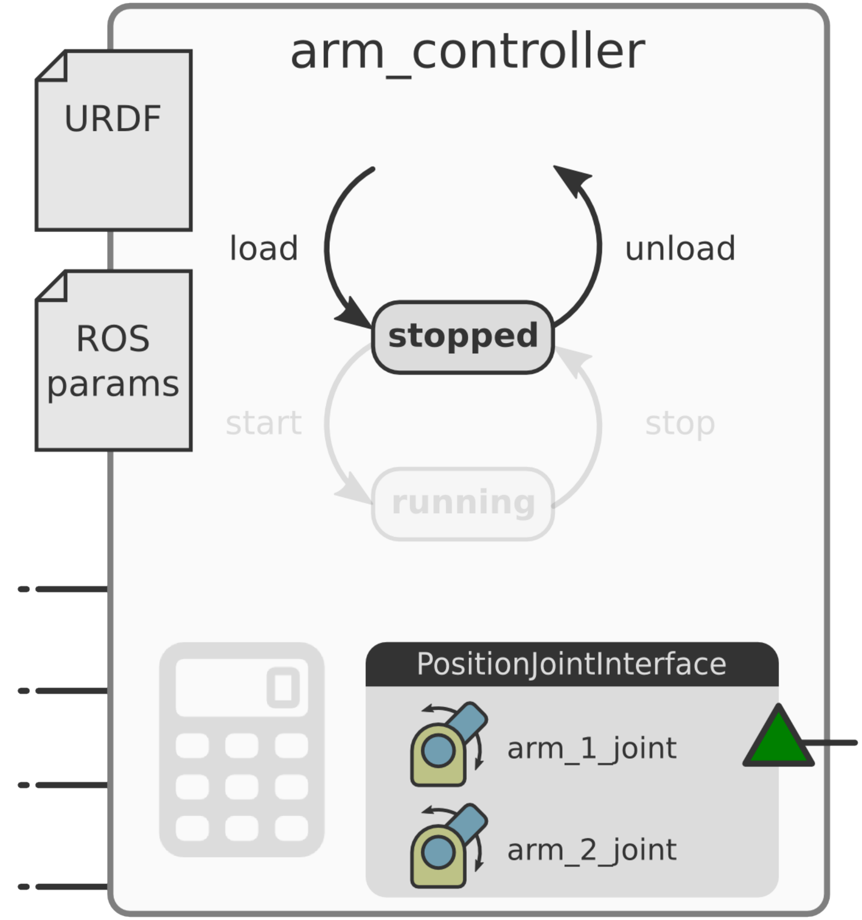


*not the same as resource **conflict** handling

ROS Controllers

Non real-time operations

- **load**
  - load + initialize plugin
  - check requisites (can fail)
    - hardware resource **existence***
    - configuration
  - setup ROS interfaces
- **unload**
  - destroy + unload plugin



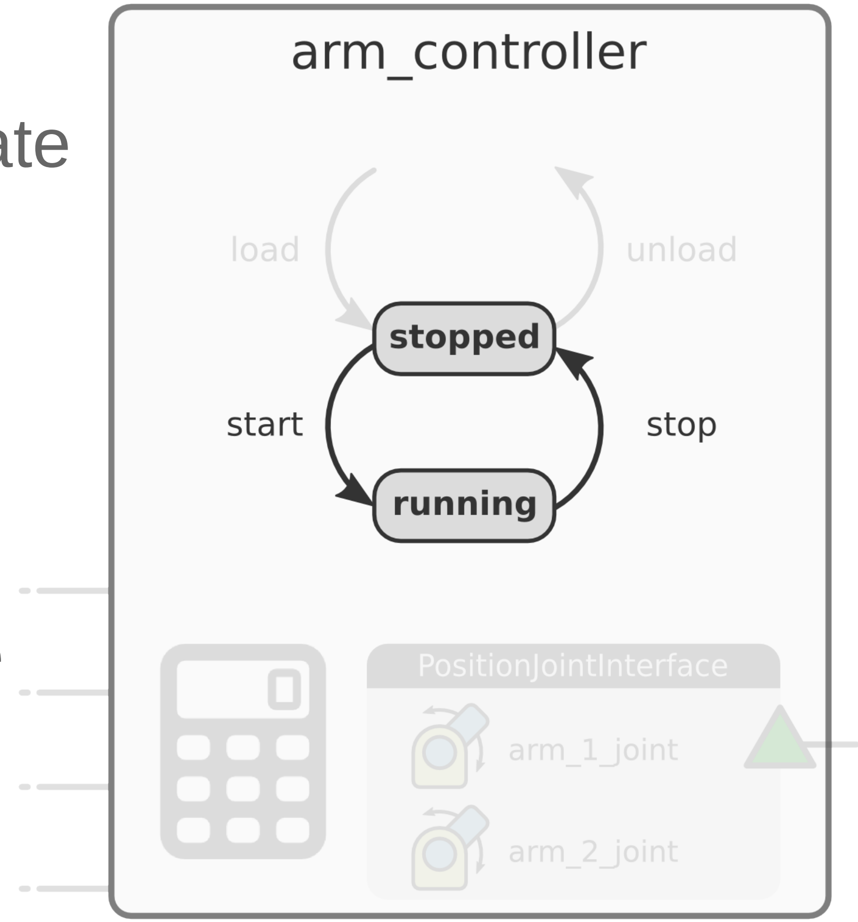
*not the same as resource **conflict** handling

ROS Controllers

Real-time safe operations

- **start** executed before first update
  - resource **conflict** handling
  - typical policy: semantic zero

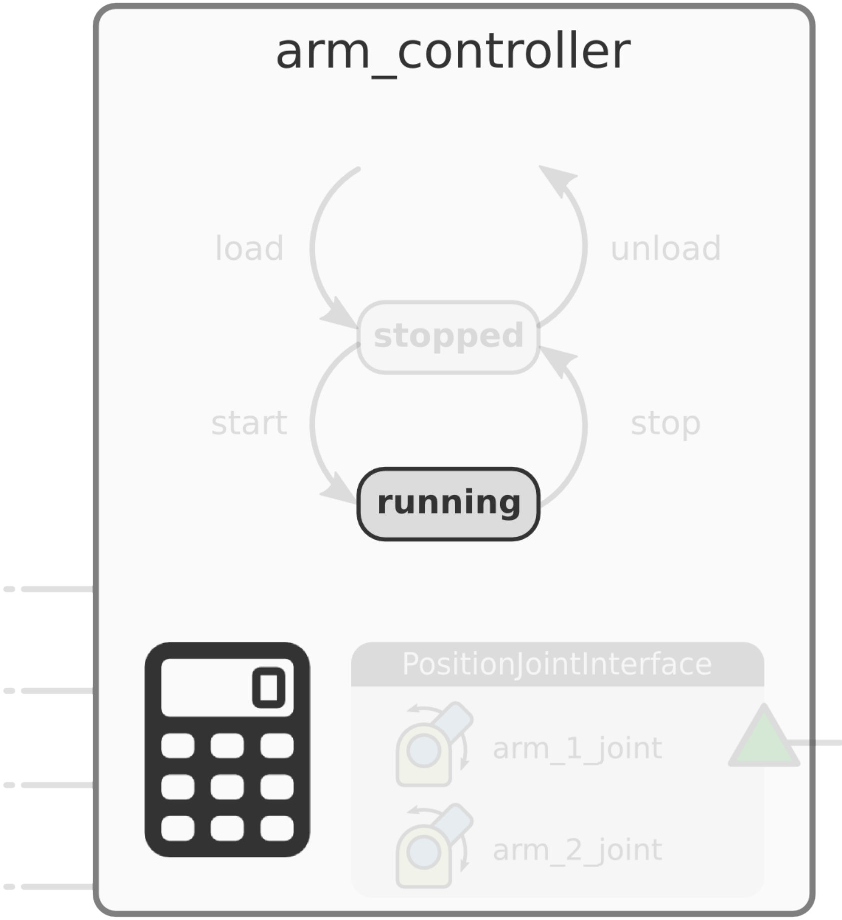
- **stop** executed after last update
  - typical policy: cancel goals



ROS Controllers

Real-time safe* operations

- **update**
  - real-time safe computation
  - executed periodically

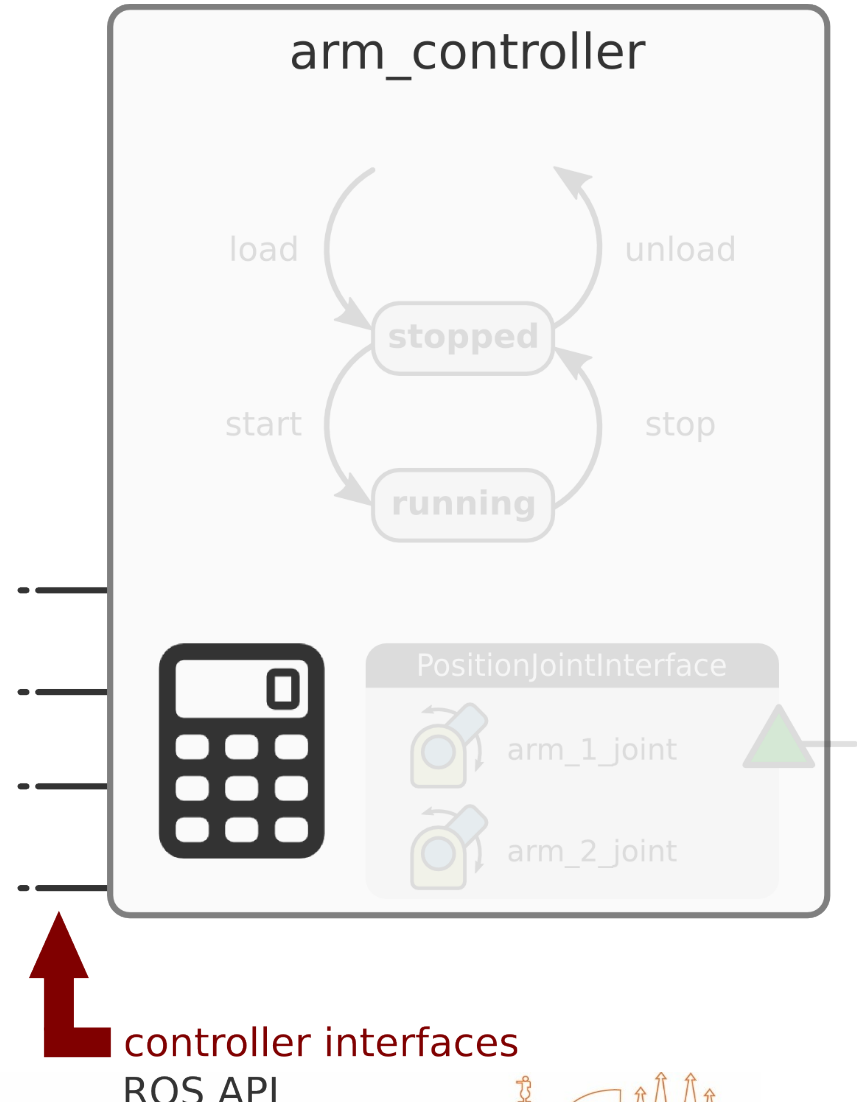


*requirement on implementation

ROS Controllers

Non real-time operations

- **callbacks**
  - non real-time computation
  - executed asynchronously



ROS Controllers

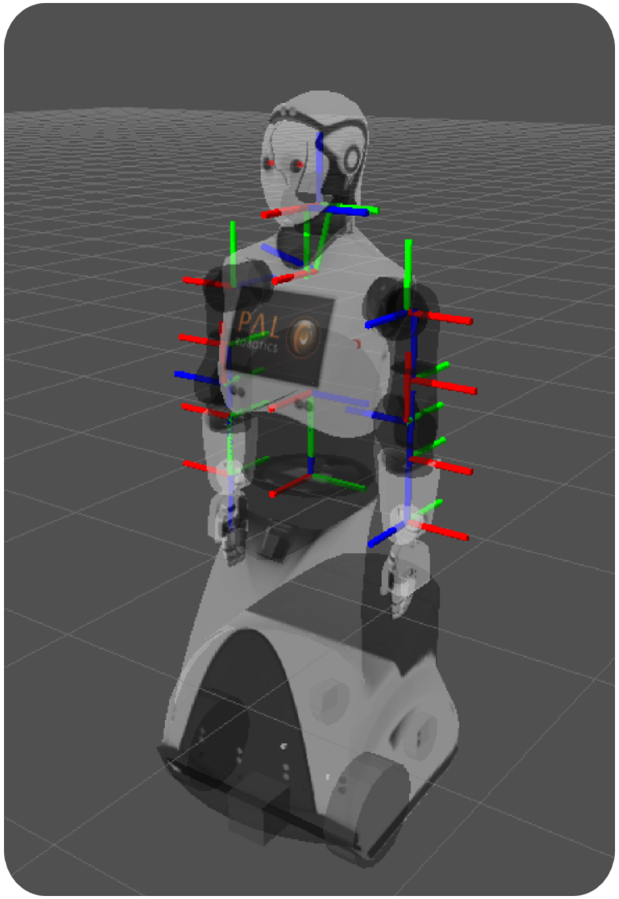
Summary

- Dynamically loadable **plugins**
- Interface defines a simple **state machine**
- Interface clearly **separates**
  - operations that are **non real-time**
  - operations required to be **real-time safe**
- Computation
  - **controller update:** periodic, real-time safe
  - **ROS API callbacks:** asynchronous, non real-time

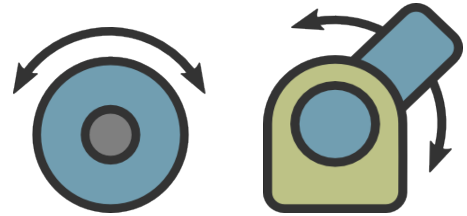
Off-the-shelf ros controllers

Sensor state reporting

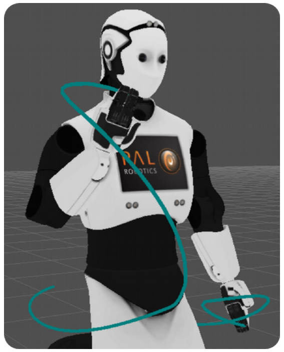
- **joint_state_controller**
  - publishes: **sensor_msgs/JointState** topic
- **imu_sensor_controller**
  - publishes: **sensor_msgs/Imu** topics
- **force_torque_sensor_controller**
  - publishes: **geometry_msgs/Wrench** topics



Off-the-shelf ros controllers

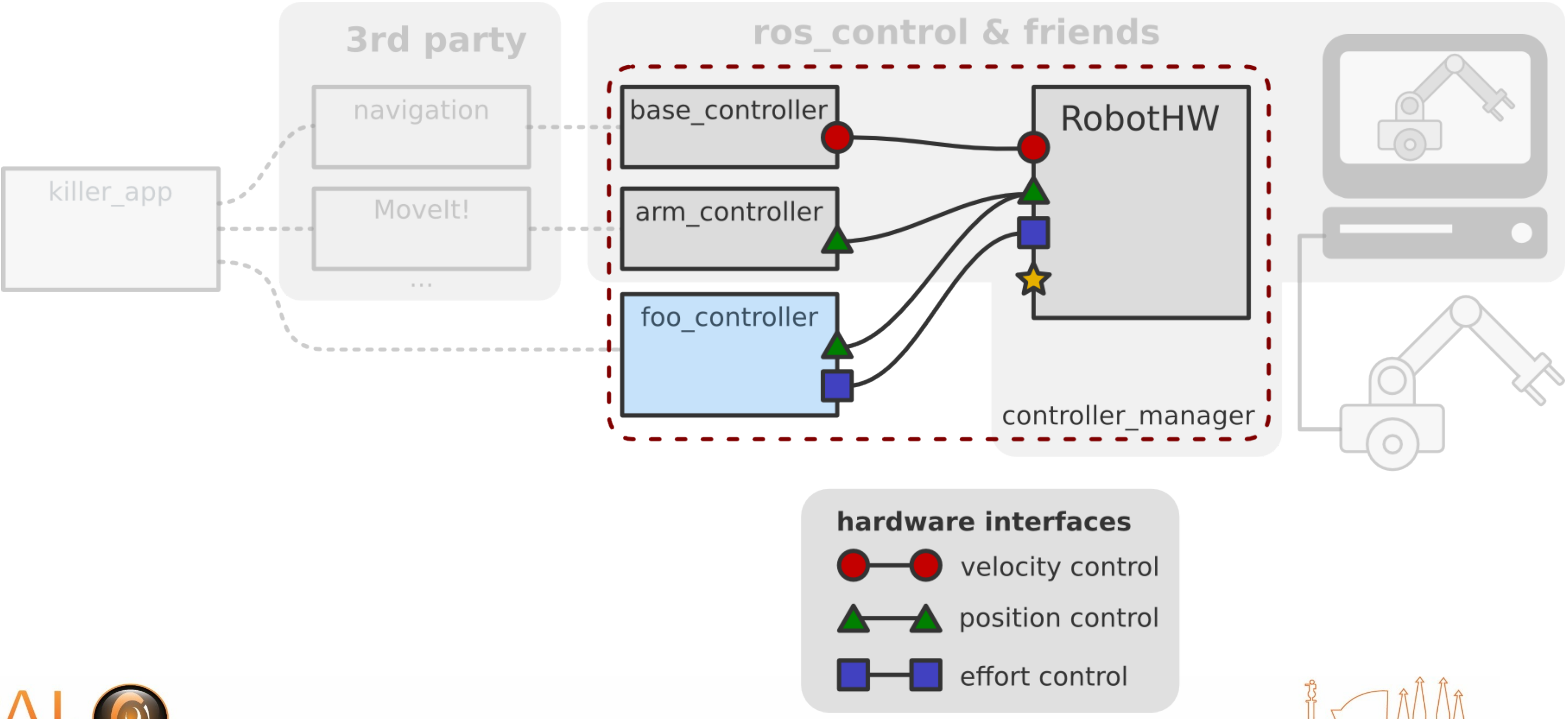


- **[position,velocity,effort]_controllers**
  - single-joint controllers in different control spaces
- **joint_trajectory_controller** (compatible with: **MoveIt!)**
  - multi-joint trajectory interpolator
  - commands:
  - **control_msgs/FollowJointTrajectory** action
  - **trajectory_msgs/JointTrajectory** topic



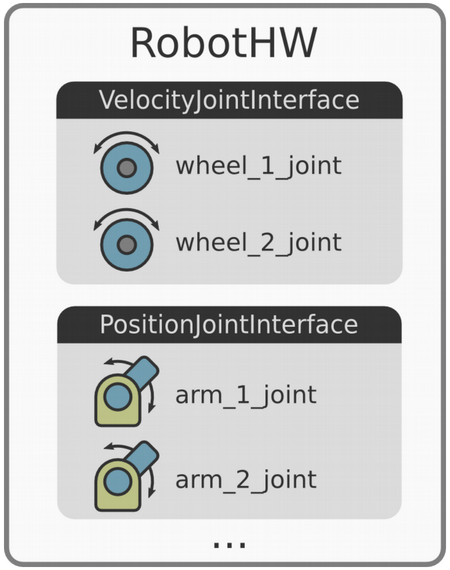
- **diff_drive_controller** or **four_wheel_steering_controller**
  - commands: **geometry_msgs/Twist** topic
  - publishes: odometry to **tf** and **nav_msgs/Odometry** topic
  - compatible with: the **ROS navigation stack**
- **gripper_action_controller** (compatible with: **MoveIt!)**
  - **single-dof gripper controller**
  - **commands:**
  - **control_msgs::GripperCommandAction** action

The control loop

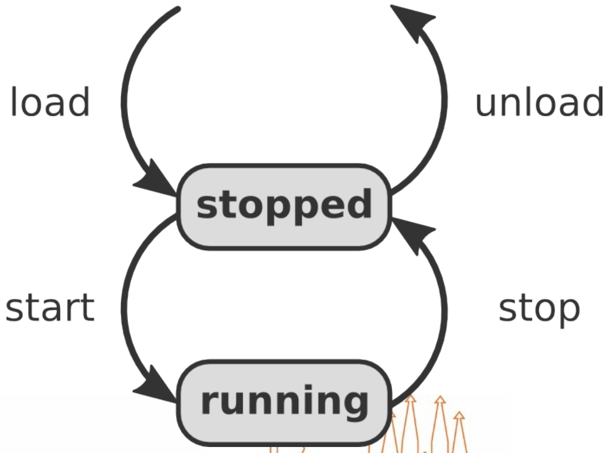


controller_manager

- Robot resource management
  - knows available resources
  - enforces resource conflict policy
  - HW interface switch (effort->position)



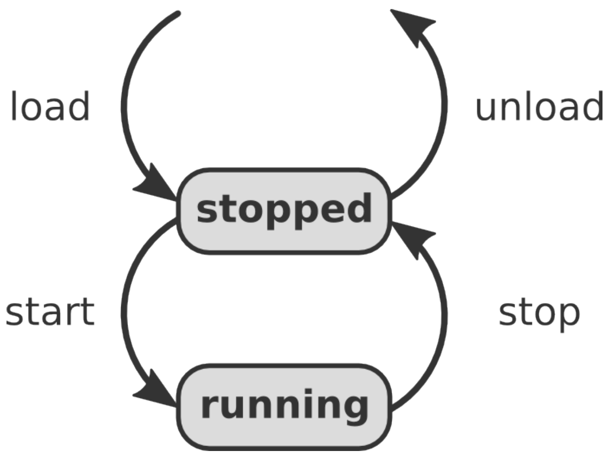
- Controller life-cycle management
  - **transitions** controller state machine
  - **updates** running controllers
  - **periodic**, **serialized** updates



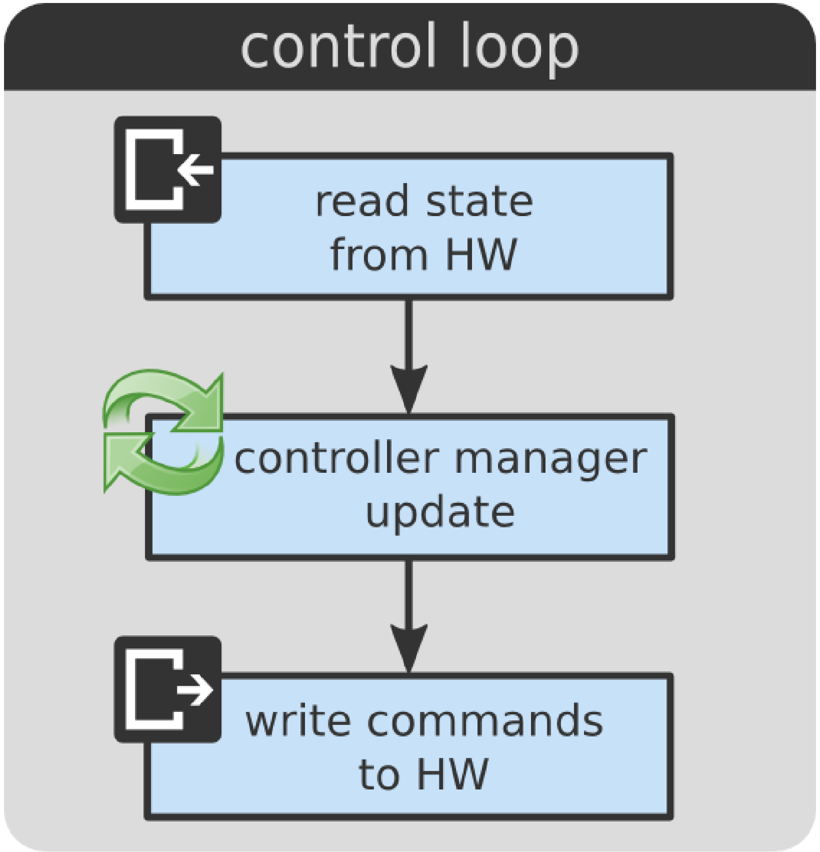
controller_manager

ROS service API

- Controller lifecycle management
  - load_controller
  - unload_controller
  - switch_controller
- Queries
  - list_controllers
  - list_controller_types
- Other
  - reload_controller_libraries



The control loop



The control loop

```
#include <ros/ros.h>
#include <my_robot/my_robot.h>
#include <controller_manager/controller_manager.h>

int main(int argc, char **argv)
{
  // Setup
  ros::init(argc, argv, "my_robot");

  MyRobot::MyRobot robot;
  controller_manager::ControllerManager cm(&robot);

  ros::AsyncSpinner spinner(1);
  spinner.start();

  // Control loop
  ros::Time prev_time = ros::Time::now();
  ros::Rate rate(10.0);

  while (ros::ok())
  {
     const ros::Time     time   = ros::Time::now();
     const ros::Duration period = time - prev_time;

     robot.read();
     cm.update(time, period);
     robot.write();

     rate.sleep();
  }
  return 0;
}
```

The control loop

```
#include <ros/ros.h>
#include <my_robot/my_robot.h>
#include <controller_manager/controller_manager.h>

int main(int argc, char **argv)
{
  // Setup
  ros::init(argc, argv, "my_robot");

  MyRobot::MyRobot robot;
  controller_manager::ControllerManager cm(&robot);

  ros::AsyncSpinner spinner(1);
  spinner.start();

  // Control loop
  ros::Time prev_time = ros::Time::now();
  ros::Rate rate(10.0);

  while (ros::ok())
  {
     const ros::Time     time   = ros::Time::now();
     const ros::Duration period = time - prev_time;

     robot.read();
     cm.update(time, period);
     robot.write();

     rate.sleep();
  }
  return 0;
}
```

The control loop

```
#include <ros/ros.h>
#include <my_robot/my_robot.h>
#include <controller_manager/controller_manager.h>

int main(int argc, char **argv)
{
  // Setup
  ros::init(argc, argv, "my_robot");

  MyRobot::MyRobot robot;
  controller_manager::ControllerManager cm(&robot);

  ros::AsyncSpinner spinner(1);
  spinner.start();

  // Control loop
  ros::Time prev_time = ros::Time::now();
  ros::Rate rate(10.0);

  while (ros::ok())
  {
     const ros::Time     time   = ros::Time::now();
     const ros::Duration period = time - prev_time;

     robot.read();
     cm.update(time, period);
     robot.write();

     rate.sleep();
  }
  return 0;
}
```

The control loop

```
#include <ros/ros.h>
#include <my_robot/my_robot.h>
#include <controller_manager/controller_manager.h>

int main(int argc, char **argv)
{
  // Setup
  ros::init(argc, argv, "my_robot");

  MyRobot::MyRobot robot;
  controller_manager::ControllerManager cm(&robot);

  ros::AsyncSpinner spinner(1);
  spinner.start();

  // Control loop
  ros::Time prev_time = ros::Time::now();
  ros::Rate rate(10.0);

  while (ros::ok())
  {
     const ros::Time     time   = ros::Time::now();
     const ros::Duration period = time - prev_time;

     robot.read();
     cm.update(time, period);
     robot.write();

     rate.sleep();
  }
  return 0;
}
```

The control loop

```
#include <ros/ros.h>
#include <my_robot/my_robot.h>
#include <controller_manager/controller_manager.h>

int main(int argc, char **argv)
{
  // Setup
  ros::init(argc, argv, "my_robot");

  MyRobot::MyRobot robot;
  controller_manager::ControllerManager cm(&robot);

  ros::AsyncSpinner spinner(1);
  spinner.start();

  // Control loop
  ros::Time prev_time = ros::Time::now();
  ros::Rate rate(10.0);

  while (ros::ok())
  {
    const ros::Time     time   = ros::Time::now();
    const ros::Duration period = time - prev_time;

    robot.read();
    cm.update(time, period);
    robot.write();

    rate.sleep();
  }
  return 0;
}
```

The control loop

```
#include <ros/ros.h>
#include <my_robot/my_robot.h>
#include <controller_manager/controller_manager.h>

int main(int argc, char **argv)
{
  // Setup
  ros::init(argc, argv, "my_robot");

  MyRobot::MyRobot robot;
  controller_manager::ControllerManager cm(&robot);

  ros::AsyncSpinner spinner(1);
  spinner.start();

  // Control loop
  ros::Time prev_time = ros::Time::now();
  ros::Rate rate(10.0);

  while (ros::ok())
  {
    const ros::Time     time   = ros::Time::now();
    const ros::Duration period = time - prev_time;

    robot.read();
    cm.update(time, period);
    robot.write();

    rate.sleep();
  }
  return 0;
}
```

The control loop

```
#include <ros/ros.h>
#include <my_robot/my_robot.h>
#include <controller_manager/controller_manager.h>

int main(int argc, char **argv)
{
  // Setup
  ros::init(argc, argv, "my_robot");

  MyRobot::MyRobot robot;
  controller_manager::ControllerManager cm(&robot);

  ros::AsyncSpinner spinner(1);
  spinner.start();

  // Control loop
  ros::Time prev_time = ros::Time::now();
  ros::Rate rate(10.0);

  while (ros::ok())
  {
    const ros::Time     time   = ros::Time::now();
    const ros::Duration period = time - prev_time;

    robot.read();
    cm.update(time, period);
    robot.write();

    rate.sleep();
  }
  return 0;
}
```

PAL ROBOTICS

The control loop

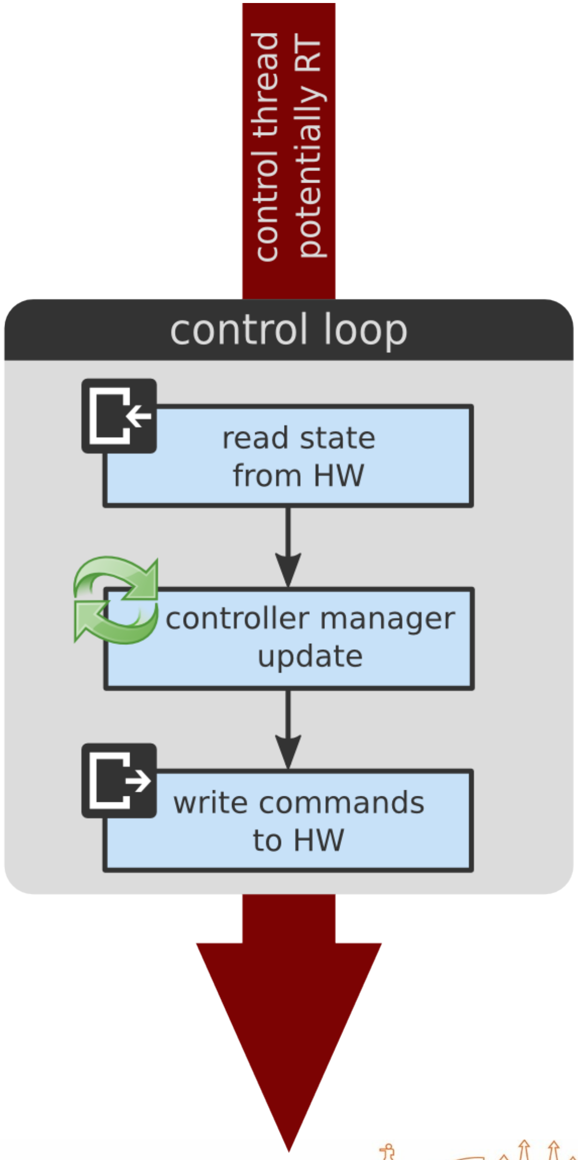
```
#include <ros/ros.h>
#include <my_robot/my_robot.h>
#include <controller_manager/controller_manager.h>

int main(int argc, char **argv)
{
  // Setup
  ros::init(argc, argv, "my_robot");

  MyRobot::MyRobot robot;
  controller_manager::ControllerManager cm(&robot);

  ros::AsyncSpinner spinner(1);
  spinner.start();

  // Control loop
  ros::Time prev_time = ros::Time::now();
  ros::Rate rate(10.0);

  while (ros::ok())
  {
     const ros::Time     time   = ros::Time::now();
     const ros::Duration period = time - prev_time;

     robot.read();
     cm.update(time, period);
     robot.write();

     rate.sleep();
  }
  return 0;
}
```

The control loop

```
#include <ros/ros.h>
#include <my_robot/my_robot.h>
#include <controller_manager/controller_manager.h>

int main(int argc, char **argv)
{
  // Setup
  ros::init(argc, argv, "my_robot");

  MyRobot::MyRobot robot;
  controller_manager::ControllerManager cm(&robot);

  ros::AsyncSpinner spinner(1);
  spinner.start();

  // Control loop
  ros::Time prev_time = ros::Time::now();
  ros::Rate rate(10.0);

  while (ros::ok())
  {
     const ros::Time     time   = ros::Time::now();
     const ros::Duration period = time - prev_time;

     robot.read();
     cm.update(time, period);
     robot.write();

     rate.sleep();
  }
  return 0;
}
```

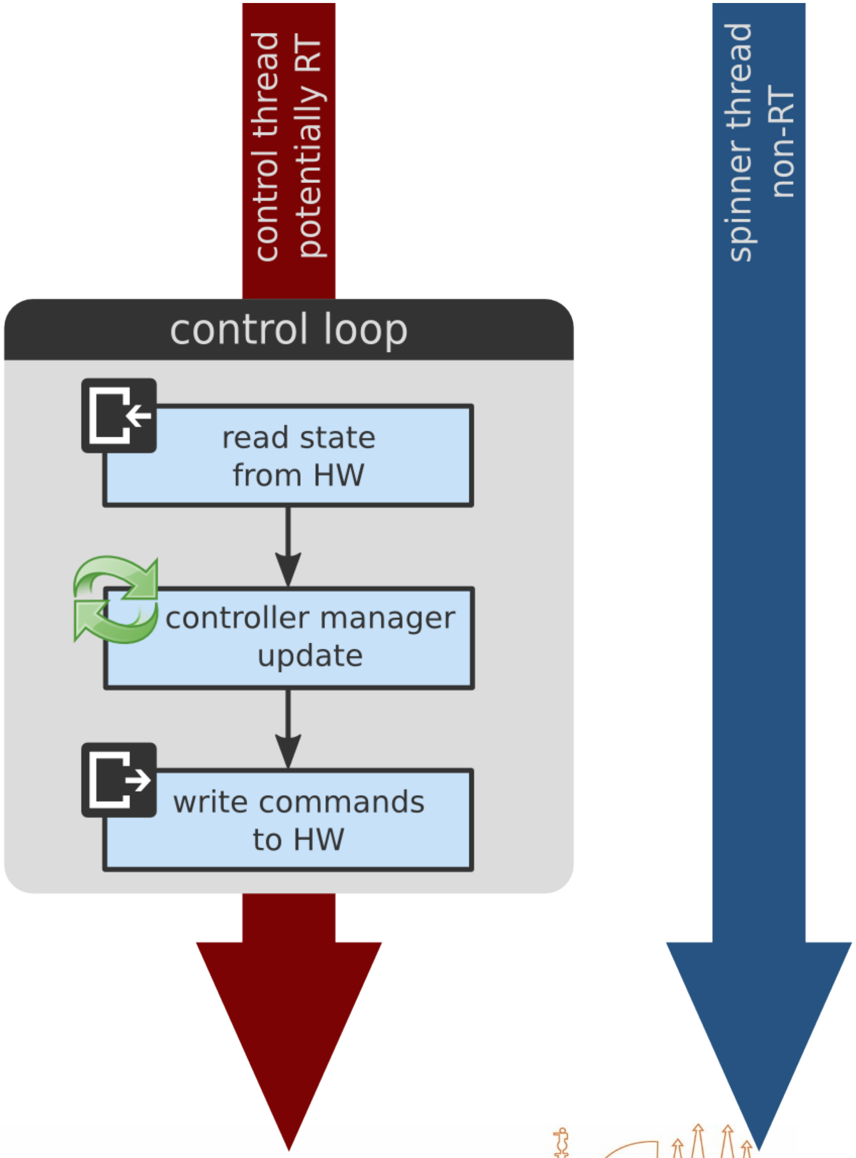
The control loop

```
#include <ros/ros.h>
#include <my_robot/my_robot.h>
#include <controller_manager/controller_manager.h>

int main(int argc, char **argv)
{
  // Setup
  ros::init(argc, argv, "my_robot");

  MyRobot::MyRobot robot;
  controller_manager::ControllerManager cm(&robot);

  ros::AsyncSpinner spinner(1);
  spinner.start();

  // Control loop
  ros::Time prev_time = ros::Time::now();
  ros::Rate rate(10.0);

  while (ros::ok())
  {
    const ros::Time     time   = ros::Time::now();
    const ros::Duration period = time - prev_time;

    robot.read();
    cm.update(time, period);
    robot.write();

    rate.sleep();
  }
  return 0;
}
```



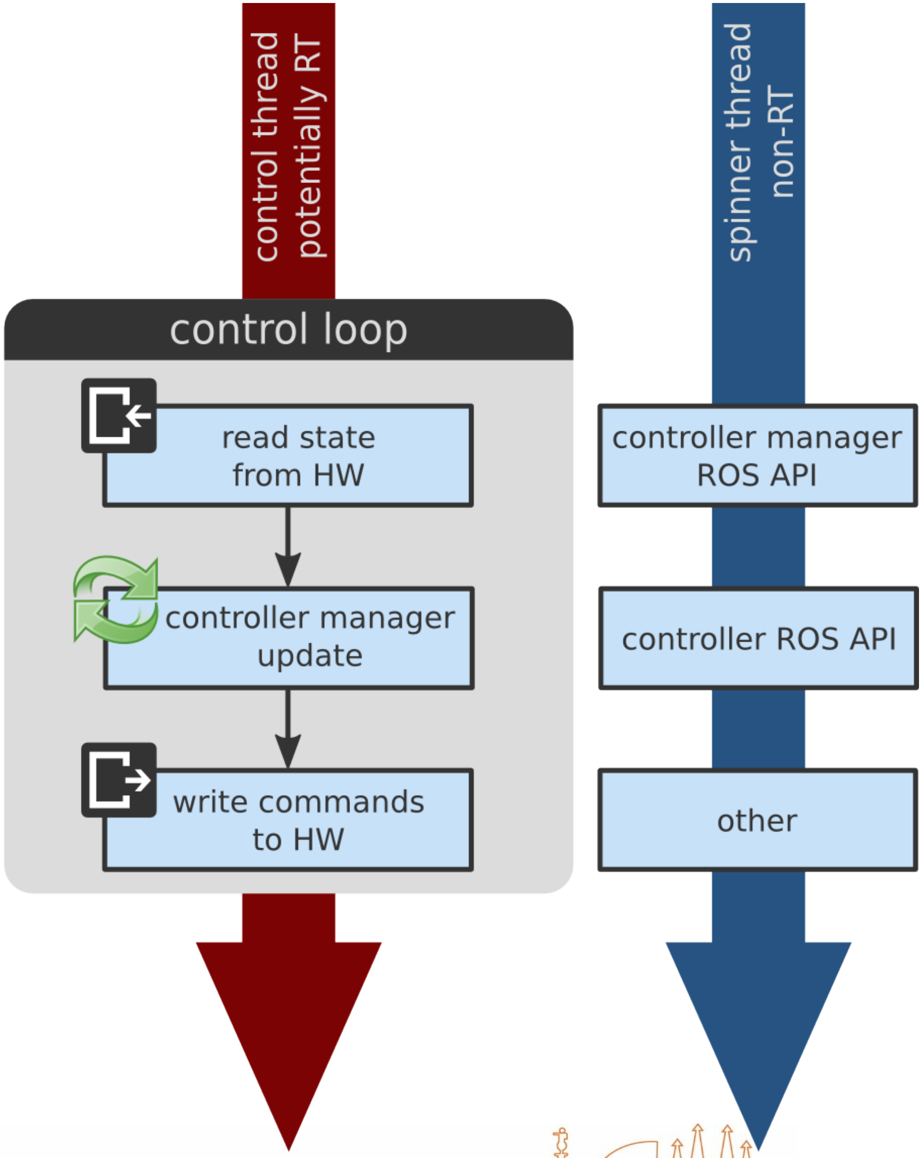
The control loop

```
#include <ros/ros.h>
#include <my_robot/my_robot.h>
#include <controller_manager/controller_manager.h>

int main(int argc, char **argv)
{
  // Setup
  ros::init(argc, argv, "my_robot");

  MyRobot::MyRobot robot;
  controller_manager::ControllerManager cm(&robot);

  ros::AsyncSpinner spinner(1);
  spinner.start();

  // Control loop
  ros::Time prev_time = ros::Time::now();
  ros::Rate rate(10.0);

  while (ros::ok())
  {
    const ros::Time     time   = ros::Time::now();
    const ros::Duration period = time - prev_time;

    robot.read();
    cm.update(time, period);
    robot.write();

    rate.sleep();
  }
  return 0;
}
```



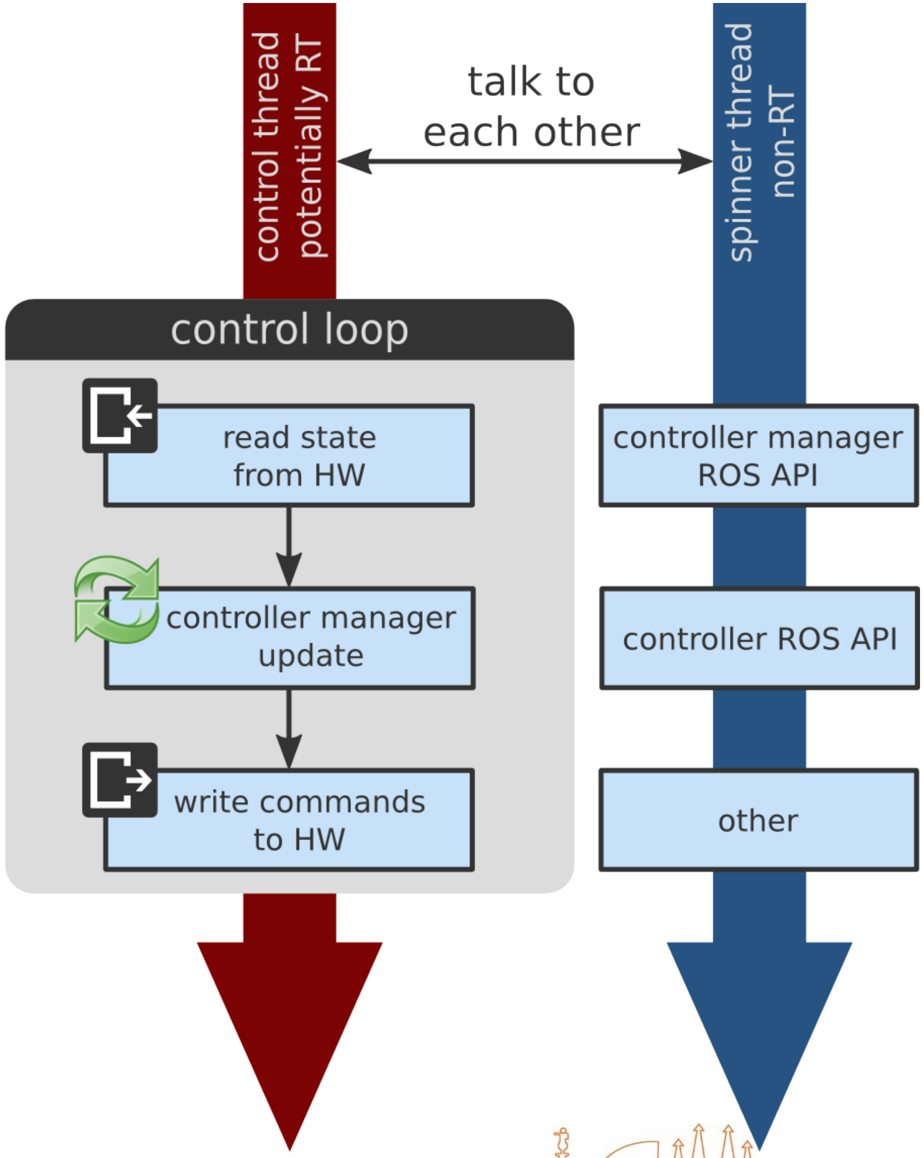
The control loop

```
#include <ros/ros.h>
#include <my_robot/my_robot.h>
#include <controller_manager/controller_manager.h>

int main(int argc, char **argv)
{
  // Setup
  ros::init(argc, argv, "my_robot");

  MyRobot::MyRobot robot;
  controller_manager::ControllerManager cm(&robot);

  ros::AsyncSpinner spinner(1);
  spinner.start();

  // Control loop
  ros::Time prev_time = ros::Time::now();
  ros::Rate rate(10.0);

  while (ros::ok())
  {
     const ros::Time     time   = ros::Time::now();
     const ros::Duration period = time - prev_time;

     robot.read();
     cm.update(time, period);
     robot.write();

     rate.sleep();
  }
  return 0;
}
```



The control loop

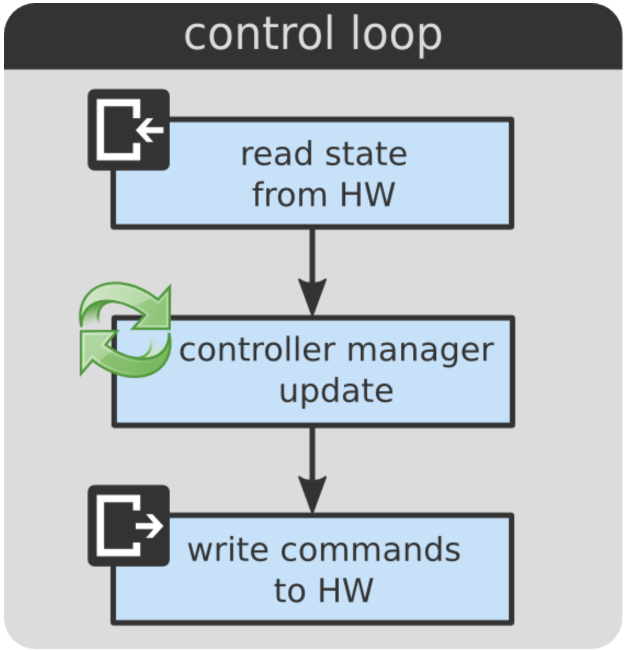
```
#include <ros/ros.h>
#include <my_robot/my_robot.h>
#include <controller_manager/controller_manager.h>

int main(int argc, char **argv)
{
  // Setup
  ros::init(argc, argv, "my_robot");

  MyRobot::MyRobot robot;
  controller_manager::ControllerManager cm(&robot);

  ros::AsyncSpinner spinner(1);
  spinner.start();

  // Control loop
  ros::Time prev_time = ros::Time::now();
  ros::Rate rate(10.0);

  while (ros::ok())
  {
    const ros::Time     time   = ros::Time::now();
    const ros::Duration period = time - prev_time;

    robot.read();
    cm.update(time, period);
    robot.write();

    rate.sleep();
  }
  return 0;
}
```



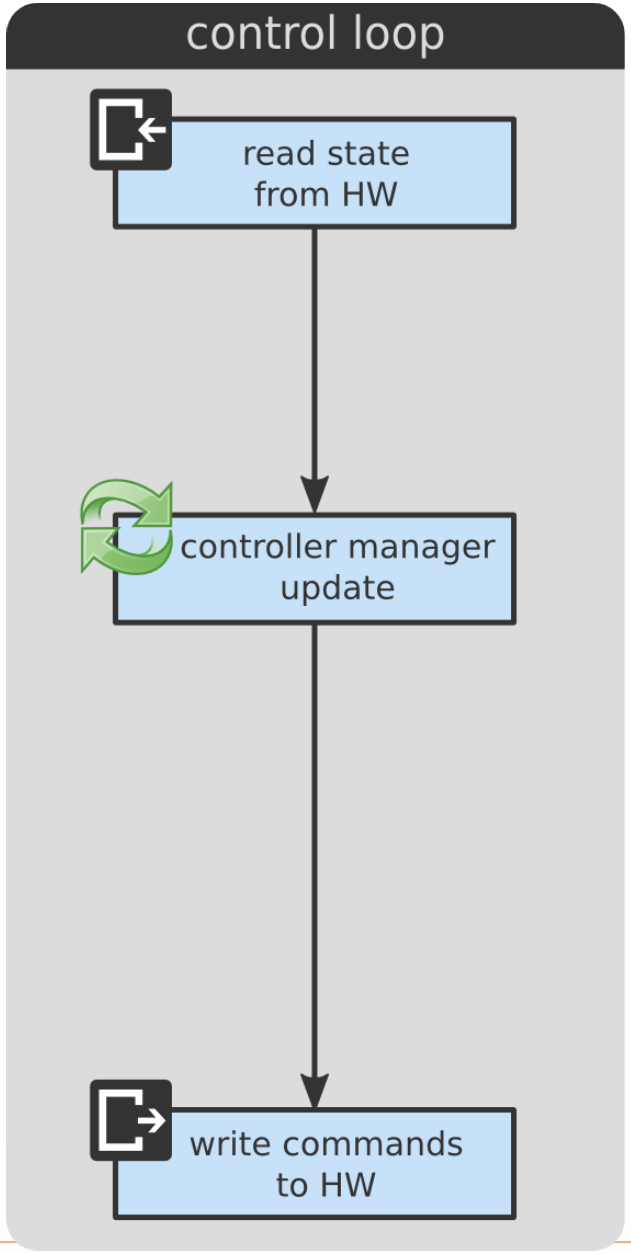
The control loop

- Tools usable from a **real-time** thread
  - **RealtimePublisher** Publish to a ROS topic
  - **RealtimeBuffer** Share resource with non-RT thread
  - **RealtimeClock** Query system clock
  - ...

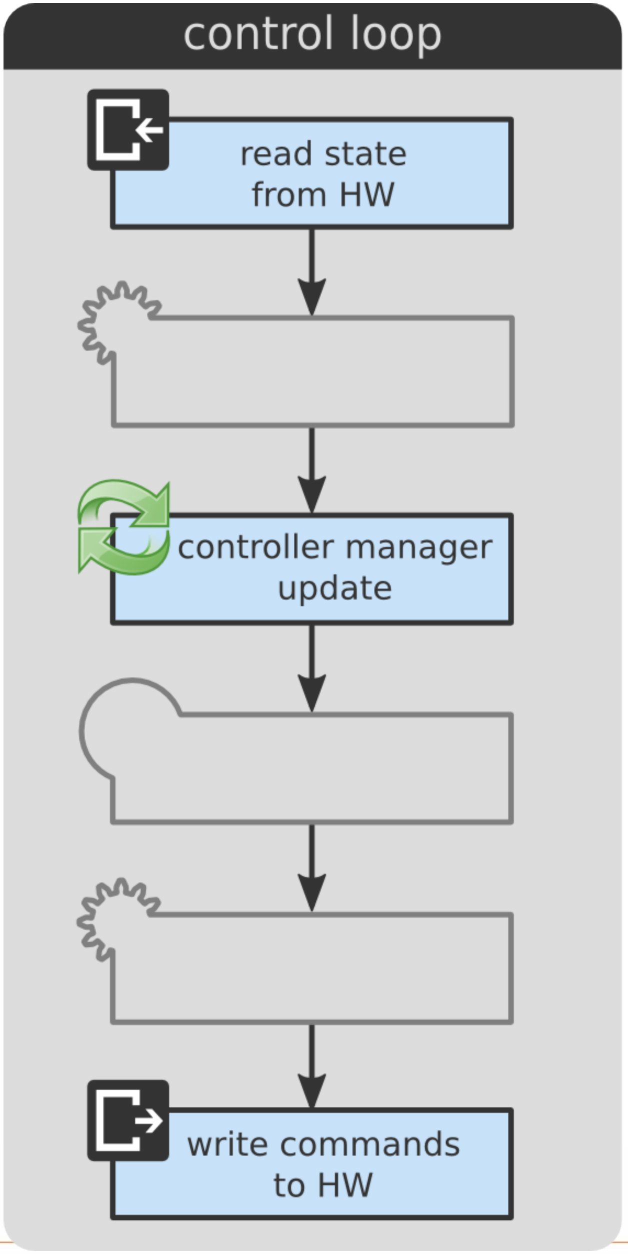
The control loop (more)



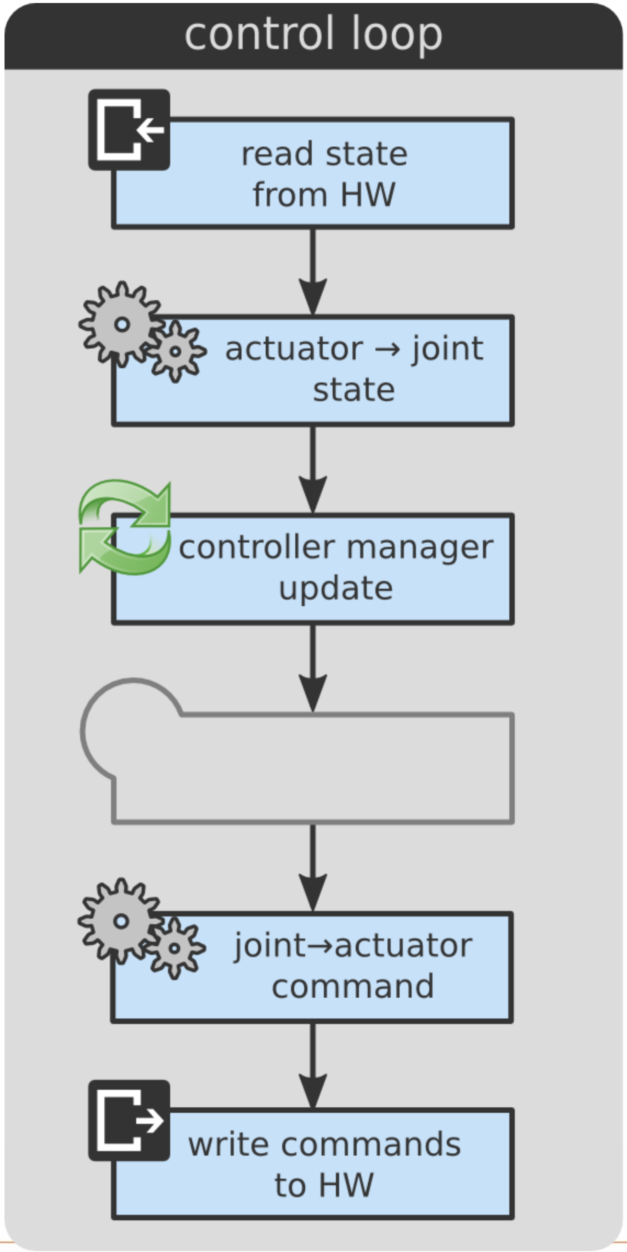
The control loop (more)



The control loop (more)

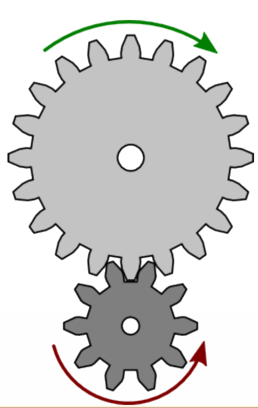


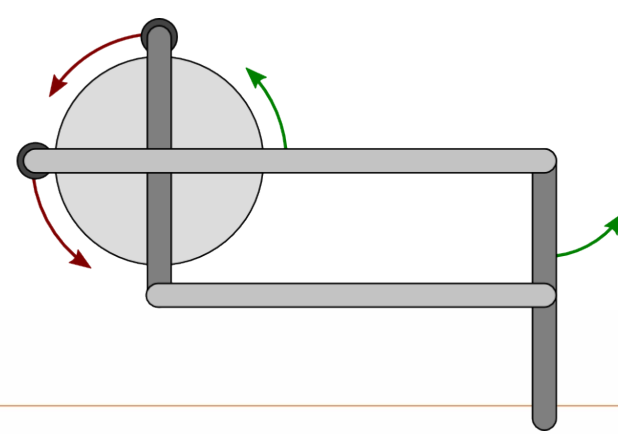
The control loop (more)



transmission_interface

- Mechanical transmission representation
  - propagate between **actuator** ↔ **joint** spaces...
  - **position**, **velocity** and **effort** variables
- Available transmissions
  - Simple reducer
  - Four-bar linkage
  - Differential

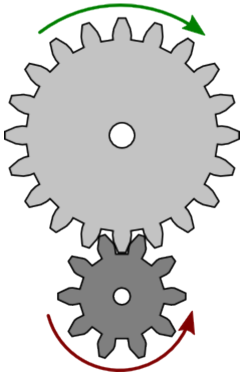




source: cgimotion.com

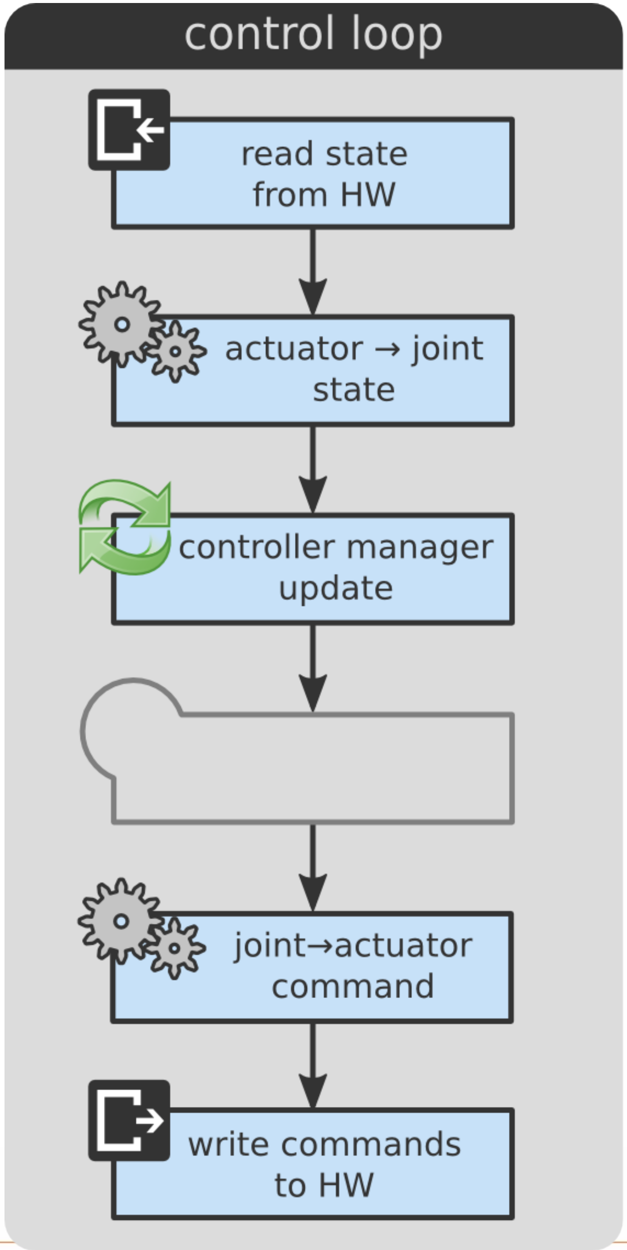
transmission_interface

- **Plugins** for loading from URDF
  - **Simplifies** populating **RobotHW** interfaces

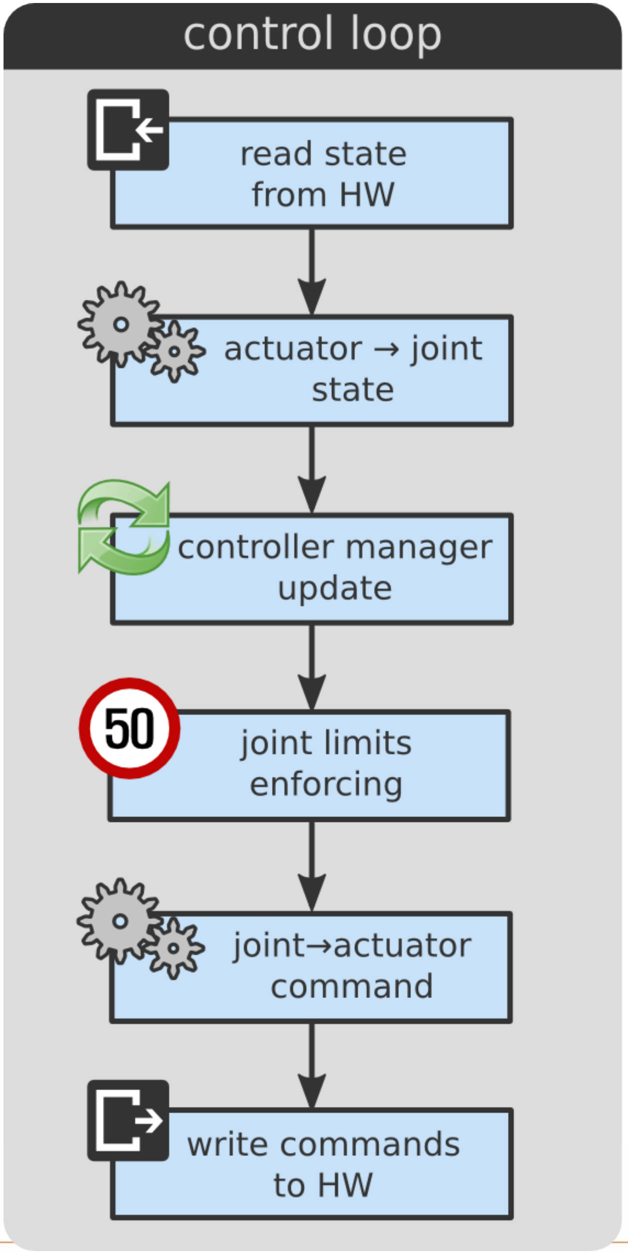


```
<transmission name="arm_1_trans">
  <type>transmission_interface/SimpleTransmission</type>
  <actuator name="arm_1_motor" >
    <mechanicalReduction>42</mechanicalReduction>
  </actuator>
  <joint name="arm_1_joint">
    <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
  </joint>
</transmission>
```

The control loop (more)



The control loop (more)



pal-robotics-forks

- **ros_control resources and interfaces**
  - motor/joint absolute encoders
  - joint torque sensors
- **ros_controllers**
  - mode_state_controller
  - joint_torque_sensor_state_controller
  - temperature_sensor_controller

Whole Body Control

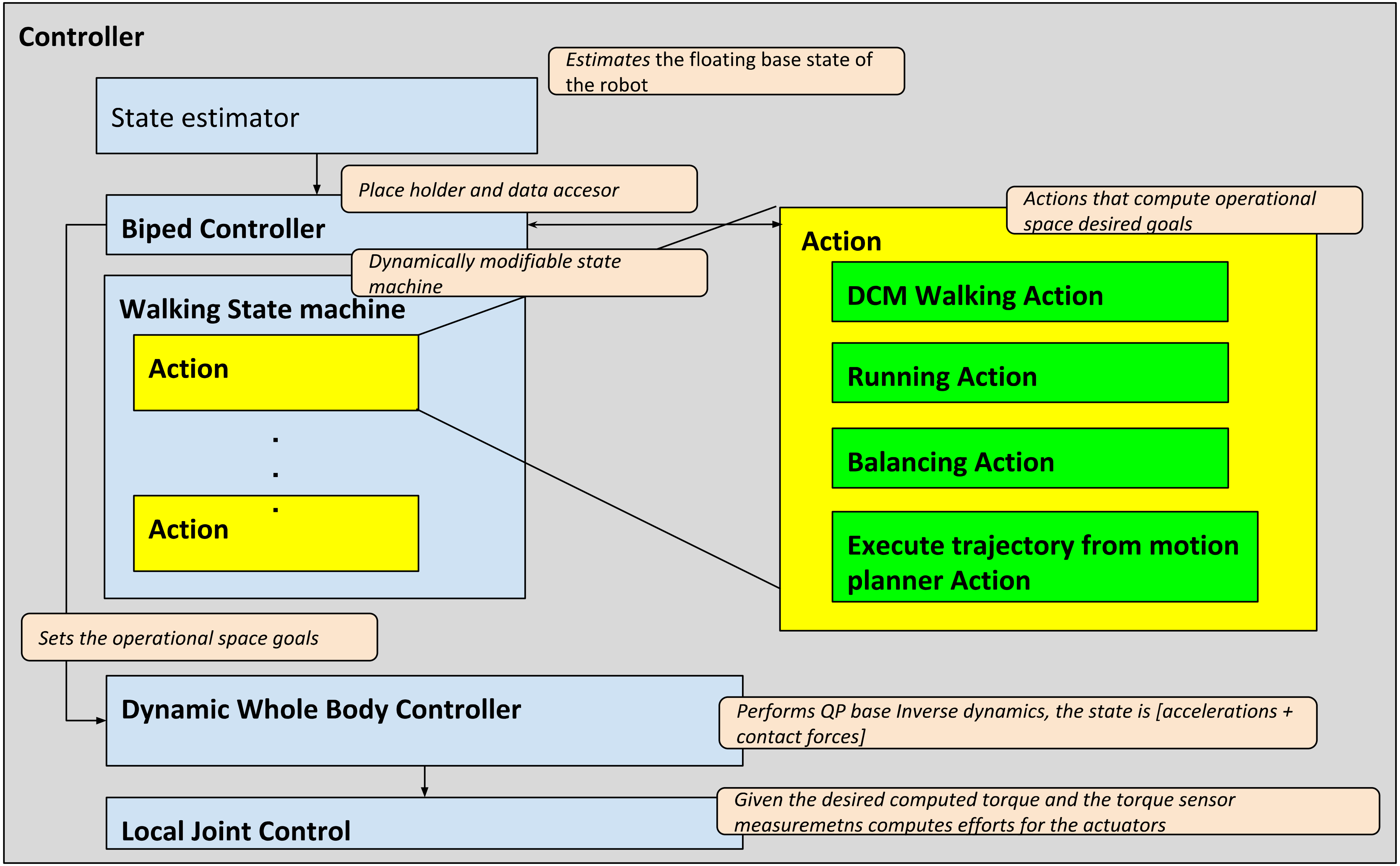
1) set of simple, low-dimensional rules

2) the rules are sufficient to guarantee the correct execution of any single task or of simultaneous multiple tasks

3) exploiting the full capabilities of the entire body of redundant, floating-based robots in compliant multi-contact interaction with the environment

source: http://www.ieee-ras.org/whole-body-control

WBC architecture



State representation

The state of our WBC is defined as $S = \begin{bmatrix} \dot{v}_q & f \end{bmatrix}$

We work in our QP controller with the **unactuated** part of the dynamics as constraints

$$M_u(q)\dot{v}_q + h_u(q, v_q) = J_u(q)^\top f \quad (1)$$

And then use the **actuated** part of the dynamics to recover the the desired torques

$$M_a(q)\dot{v}_q + h_a(q, v_q) = \tau + J_a(q)^\top f \quad (2)$$

Even if we don't have explicitly the torques in the state, we impose limits and objectives on them by **reformulating** the objectives or constrains also using the actuated part of the dynamics.

Contact force definition

There are different ways to describe the **contact forces** of the feet with ground (Bipeds)

- A single wrench at each foot
- Four contact forces in the edges of each foot

Friction constraints

- Formulate the contact forces as **friction pyramid**
- Impose **constraints on the normal forces** with respect to the tangential forces

When using regularization on the contact forces, the different formulations will result in different contact forces when there is redundancy in the solution "Optimal distribution of contact forces with inverse-dynamics control", Righetti et al., 2013

State estimation

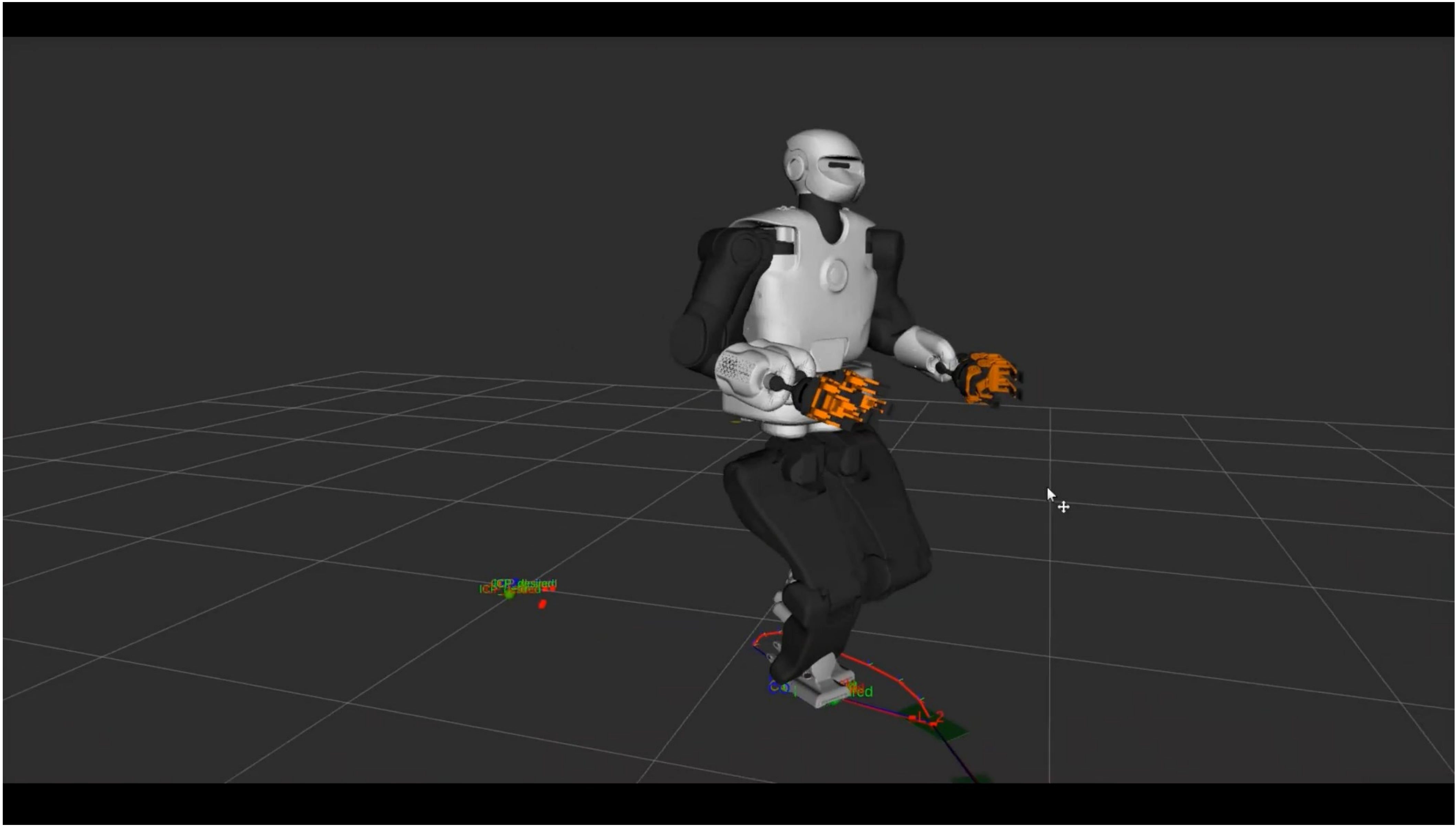
We cannot directly measure some parts of the unactuated part of the state $S = \begin{bmatrix} q_u & \dot{v}_u \end{bmatrix}$. To circumvent this we use a floating base estimator

- Stochastic "State estimation for a humanoid robot", Rotella et al., 2014
- Least Squares "Torque-Based Dynamic Walking - A Long Way from Simulation to Experiment", Englsberger et al., 2018

Performing estimation on SE(3) and SO(3) is not trivial:

- "A micro Lie theory for state estimation in robotics", Solà, Deray, and Atchuthan, 2018

State estimation

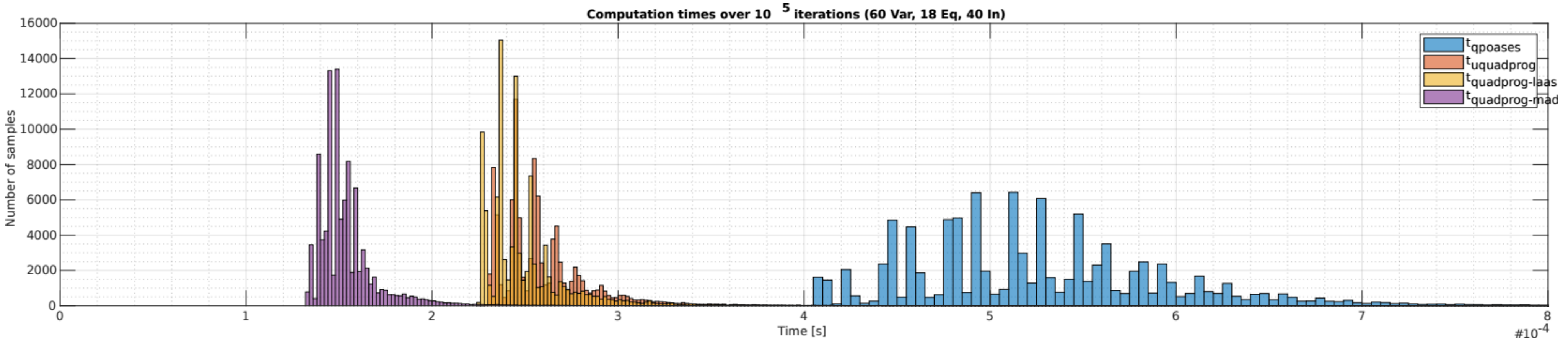


PAL ROBOTICS

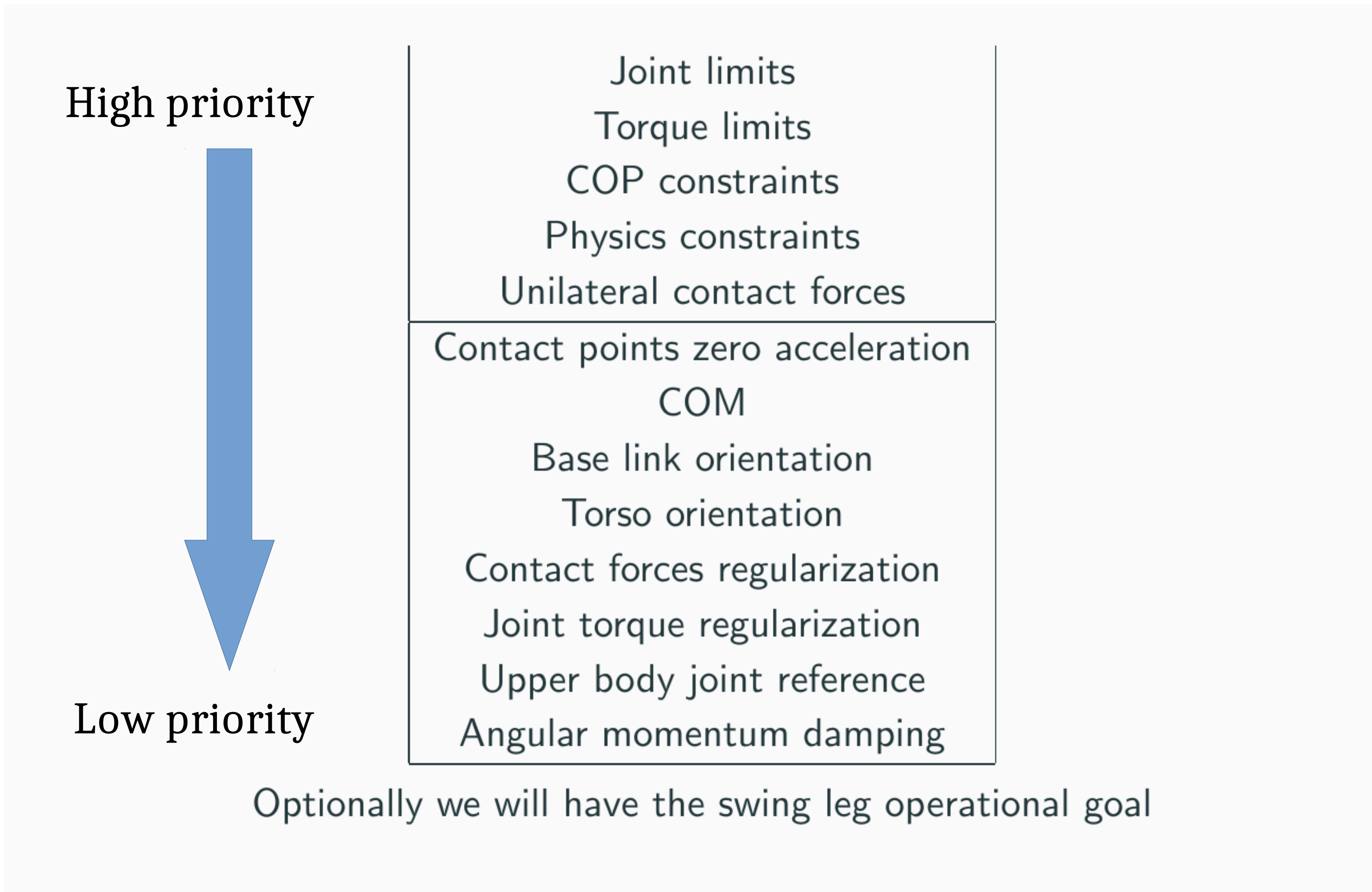
Optimization

qpmad `https://github.com/asherikov/qpmad`

- Inspired by the Goldfarb-Idnani quadratic programming solver uquadprog
- Support for double sided inequality constraints $lb <= Ax <= ub$
- Simple bounds $lb <= x <= ub$



WBC Stack


High priority
Low priority
Joint limits
Torque limits
COP constraints
Physics constraints
Unilateral contact forces
Contact points zero acceleration
COM
Base link orientation
Torso orientation
Contact forces regularization
Joint torque regularization
Upper body joint reference
Angular momentum damping
Optionally we will have the swing leg operational goal

WBC Balance

COM Regulation

Our balancing policy is to regulate the divergent component of motion.
Reference paper: “Three-dimensional bipedal walking control using Divergent Component of Motion”, Englsberger, Ott, and Albu-Schffer,

More balancing experiments

Joint limits
Torque limits
COP constraints
Physics constraints
Unilateral contact forces
Contact points zero acceleration
COM
Base link orientation
Torso orientation
Contact forces regularization
Joint torque regularization
Upper body joint reference
Angular momentum damping

+ swing leg operational space target

Preliminary walking experiment

Fast upper body motion

Conclusions

Thanks for your attention!

Hiring

We have 2 open positions to work with our humanoid robots in control, optimization, wbc and motion planning

recruit@pal-robotics.com

Kinematic whole body control

Stack of Tasks	
High priority	Joint limits
	Self collision avoidance
	Fixed feet + CoM centered
	Gaze
	Hands position
	Torso orientation upright
Low priority	Joint reference posture

Kinematic whole body control