Samwoo Seong
samwoose@gmail.com

Quiz, ROS Control 101
June 30, 2020

1.0 Objectives

-Create a package that allows to control joints of UR5 with default controllers provided by ROS Control Package. [1]
-Create a customized controller and replace one of default controllers with the customized controller. [1]

1.1 Abstract and Motivation

Often robots have mobility to perform certain tasks. For instance, robot arm moves their joints to grab a target object. Traditionally, developers are utilizing motors and controllers to create the mobility in robots. In ROS, there is a great package called ROS Control Package that takes care of complicated process behind the scene and allows us to control joints of robots with controllers by launching some nodes. Furthermore, we can customize our own controller with this package. Therefore, it is important to know how to employ this great package.

1.2 Approach and Procedures

In this practice, we have two main objectives. First, we need to create a package so that we can control UR5's joints. Second, we have to create our own controller and replace one of controllers that is already configured with it. Since movable joints and controllers are highly correlated (i.e., robots tend to have motors and controllers to move their joints), joint_state_publisher node will be also launched. For more details about this node, please refer to "Report_QUIZ_TF_ ROS 101.pdf" in "Publishing_TF_PiRobot_N_Turtlebot" directory on my github. [2] We can achieve the first objective with two major procedures.

1. Write up controller configuration file (a.k.a quiz_control.yaml) based on URDF file of UR5 robot.
2. Create a launch file to load controllers configured in quiz_control.yaml file using a node provided by ROS Control package.

From here, we can accomplish the second goal with following steps.

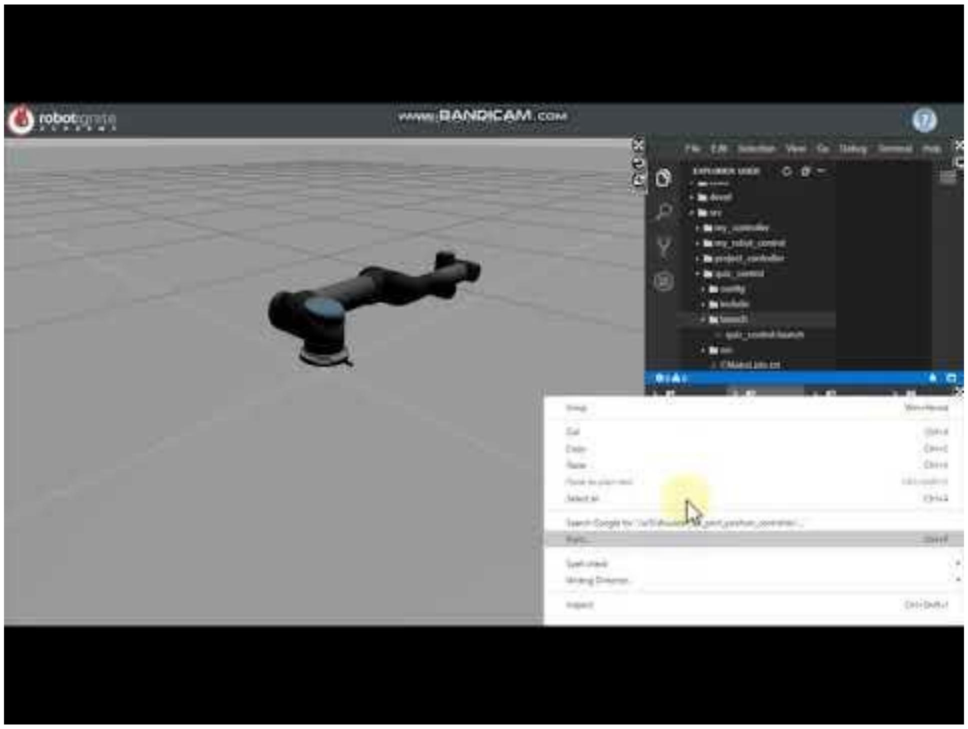
1. Create source code that defines a customized controller.

2. Create custom_controller_plugins.xml and add library path properly.
3. Modify CMakeLists.txt, package.xml, yaml file, and launch file accordingly.

More details of each step will be discussed in 1.4 Discussion.

In order to achieve two major objectives separately, we are going to create two packages called quiz_control for the first goal and project_controller for the second goal.

1.3 Experimental Results



Video 1. Control shoulder_lift_joint with a default controller provided by ROS Control package
https://www.youtube.com/watch?v=3mWaMuYgiKY&feature=youtu.be

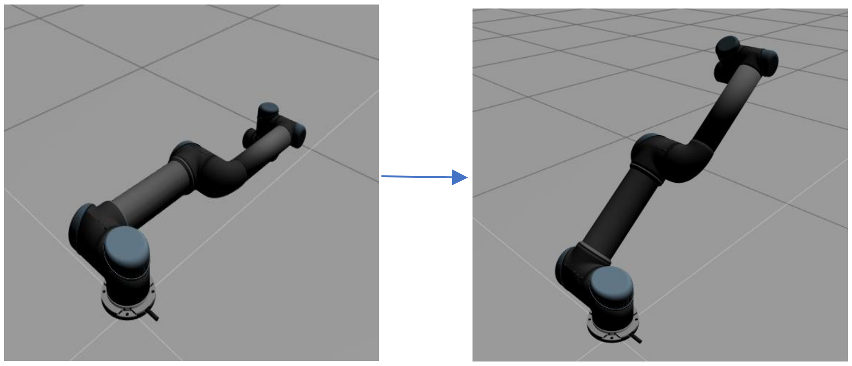
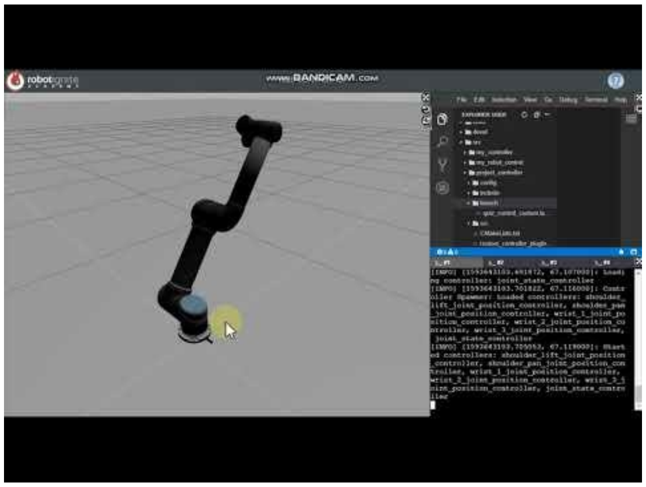


Figure 1. Before (left) and after (right) publishing data value to shoulder_lift_joint_position_controller/command topic



```
[INFO] [1593643103.691872, 67.107000]: Loadi
ng controller: joint_state_controller
[INFO] [1593643103.701822, 67.116000]: Contr
oller Spawner: Loaded controllers: shoulder_
lift_joint_position_controller, shoulder_pan
_joint_position_controller, wrist_1_joint_po
sition_controller, wrist_2_joint_position_co
ntroller, wrist_3_joint_position_controller,
 joint_state_controller
[INFO] [1593643103.705953, 67.119000]: Start
ed controllers: shoulder_lift_joint_position
_controller, shoulder_pan_joint_position_con
troller, wrist_1_joint_position_controller,
wrist_2_joint_position_controller, wrist_3_j
oint_position_controller, joint_state_contro
ller
```

Video 2. Control shoulder_lift_joint with a customized controller

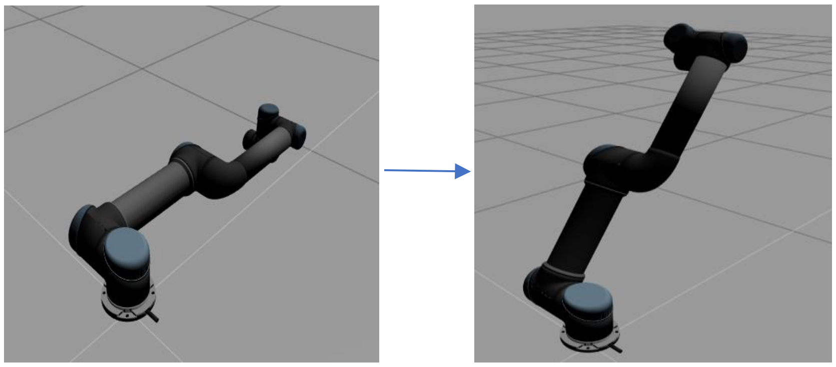
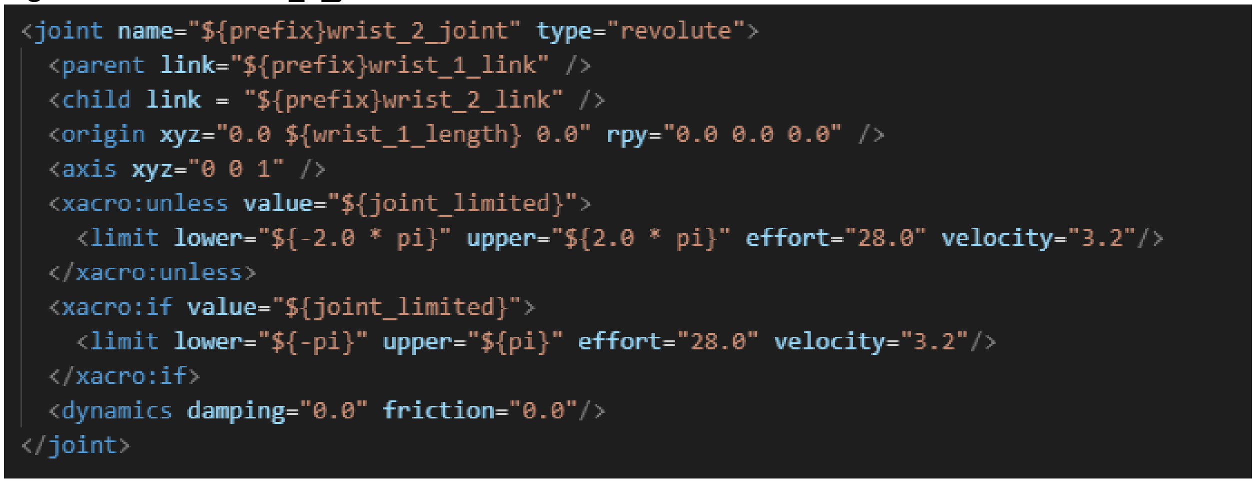


Figure 2. Before (left) and after (right) publishing data value to shoulder_lift_joint_position_controller/command topic

1.4 Discussion

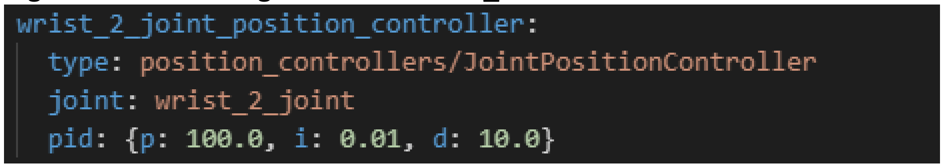
As we mentioned above, movable joints often obtain their mobility from motors and controllers. Therefore, in order to move around these joints, configuration file of these controllers is needed. We will call it quiz_control.yaml which will be located in a package, quiz_control. Since all information about our robot is stored in URDF file of UR5, we need to investigate the URDF file and find necessary information such as names of movable joints. Here are an example of a definition tag of a joint of UR5 and its controller configuration in quiz_control.yaml.

e.g. definition of wrist_2_joint



```xml
<joint name="${prefix}wrist_2_joint" type="revolute">
  <parent link="${prefix}wrist_1_link" />
  <child link = "${prefix}wrist_2_link" />
  <origin xyz="0.0 ${wrist_1_length} 0.0" rpy="0.0 0.0 0.0" />
  <axis xyz="0 0 1" />
  <xacro:unless value="${joint_limited}">
    <limit lower="${-2.0 * pi}" upper="${2.0 * pi}" effort="28.0" velocity="3.2"/>
  </xacro:unless>
  <xacro:if value="${joint_limited}">
    <limit lower="${-pi}" upper="${pi}" effort="28.0" velocity="3.2"/>
  </xacro:if>
  <dynamics damping="0.0" friction="0.0"/>
</joint>
```

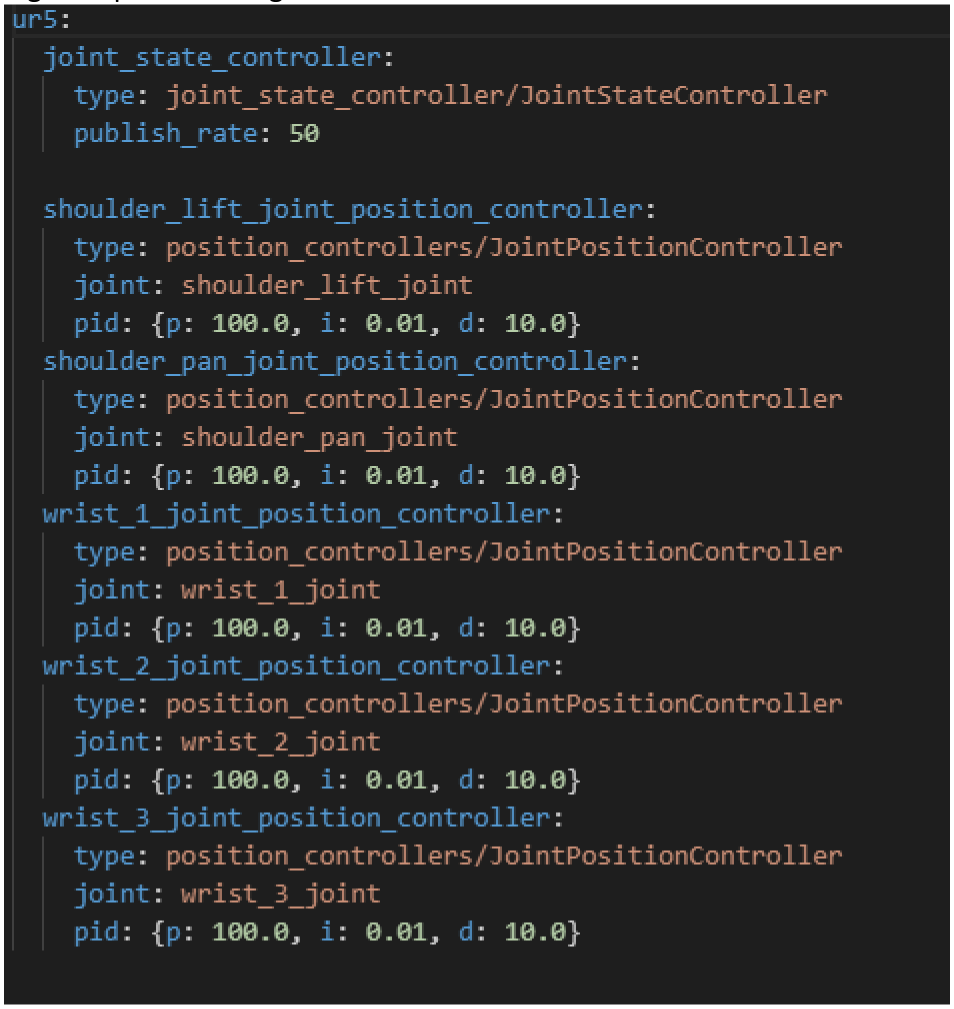
e.g. controller configuration for wrist_2



```yaml
wrist_2_joint_position_controller:
  type: position_controllers/JointPositionController
  joint: wrist_2_joint
  pid: {p: 100.0, i: 0.01, d: 10.0}
```

As we see, wrist_2_joint_position_controller is a name of the controller. It needs other definitions such as type of controller, name of joint that the controller is involved, and setting up values of PID controller. Here, we are using position_controllers/JointPositionController as controller type and values, 100, 0.01, 10.0 for PID set up as they are given in the quiz description. In order to complete this configuration step, we need to iteratively define multiple controllers for all movable joints. Here is an example of configuration file.

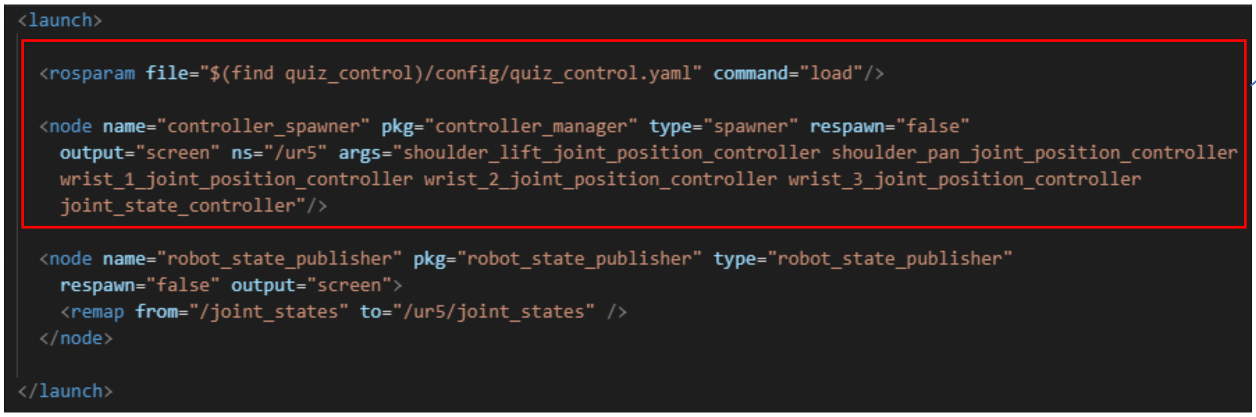
e.g. completed configuration file



```yaml
ur5:
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50

  shoulder_lift_joint_position_controller:
    type: position_controllers/JointPositionController
    joint: shoulder_lift_joint
    pid: {p: 100.0, i: 0.01, d: 10.0}
  shoulder_pan_joint_position_controller:
    type: position_controllers/JointPositionController
    joint: shoulder_pan_joint
    pid: {p: 100.0, i: 0.01, d: 10.0}
  wrist_1_joint_position_controller:
    type: position_controllers/JointPositionController
    joint: wrist_1_joint
    pid: {p: 100.0, i: 0.01, d: 10.0}
  wrist_2_joint_position_controller:
    type: position_controllers/JointPositionController
    joint: wrist_2_joint
    pid: {p: 100.0, i: 0.01, d: 10.0}
  wrist_3_joint_position_controller:
    type: position_controllers/JointPositionController
    joint: wrist_3_joint
    pid: {p: 100.0, i: 0.01, d: 10.0}
```

For more information about joint_state_controller, please read a report that can be found in the link [2].

Now that configuration file is set, it is time to write up a launch file accordingly. Major things to do with our launch file is to load controllers from our configuration file and put all controllers as arguments of node called controller_spawner which is provided by ROS Control package. Names of all controllers are the same as controller names in the configuration file. Besides, name space, ns, should be taken into account as well. Here, it will be "/ur5". An example of loading the controller configuration file and launching the controller_spawner node is shown below.
e.g. loading a yaml file and launching controller_spawner node provided by ROS Control package



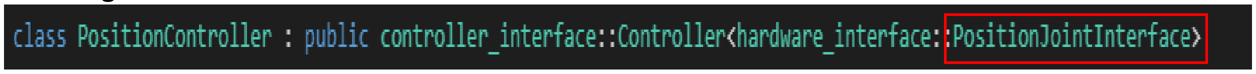
```xml
<launch>

  <rosparam file="$(find quiz_control)/config/quiz_control.yaml" command="load"/>

  <node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="false"
    output="screen" ns="/ur5" args="shoulder_lift_joint_position_controller shoulder_pan_joint_position_controller
    wrist_1_joint_position_controller wrist_2_joint_position_controller wrist_3_joint_position_controller
    joint_state_controller"/>

  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher"
    respawn="false" output="screen">
    <remap from="/joint_states" to="/ur5/joint_states" />
  </node>

</launch>
```

Part that does the job

For more details about robot_state_publisher node, please take a look at report that can be found on the link below. [2]

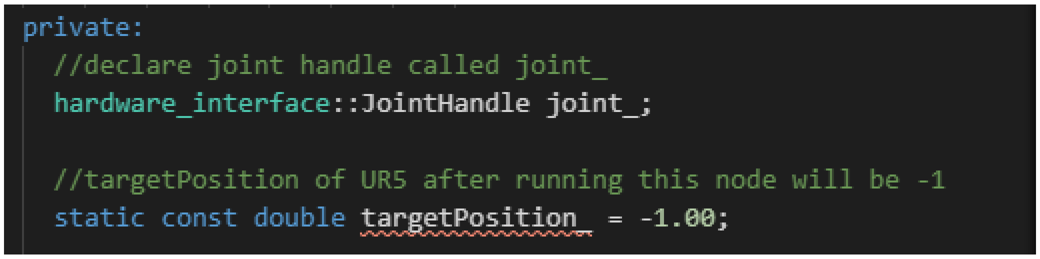
Now, let's talk about task 2. To add a customized controller, we need to complete 3 major steps as we mentioned earlier. Let's take a look at our source code that defines the controller.

<Writing a source code for a customized controller>



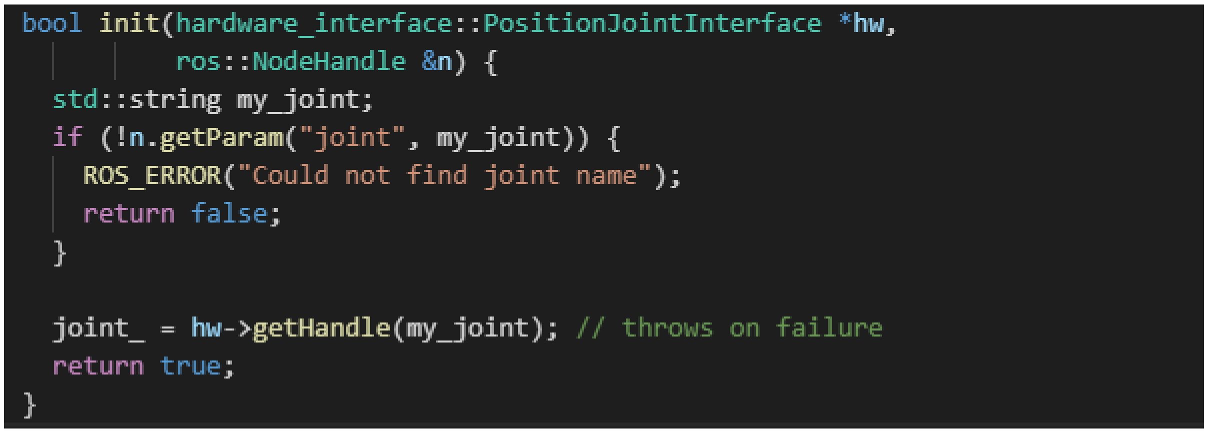
```cpp
class PositionController : public controller_interface::Controller<hardware_interface::PositionJointInterface>
```

First of all, we are going to inherit a class controller_interface::Controller<hardware_interface::PositionJointInterface> because type of all controllers is "position_controllers/JointPositionController" meaning we need position interface to accept position command. Besides, our class will be able to use its parent class's methods and variables as needed.



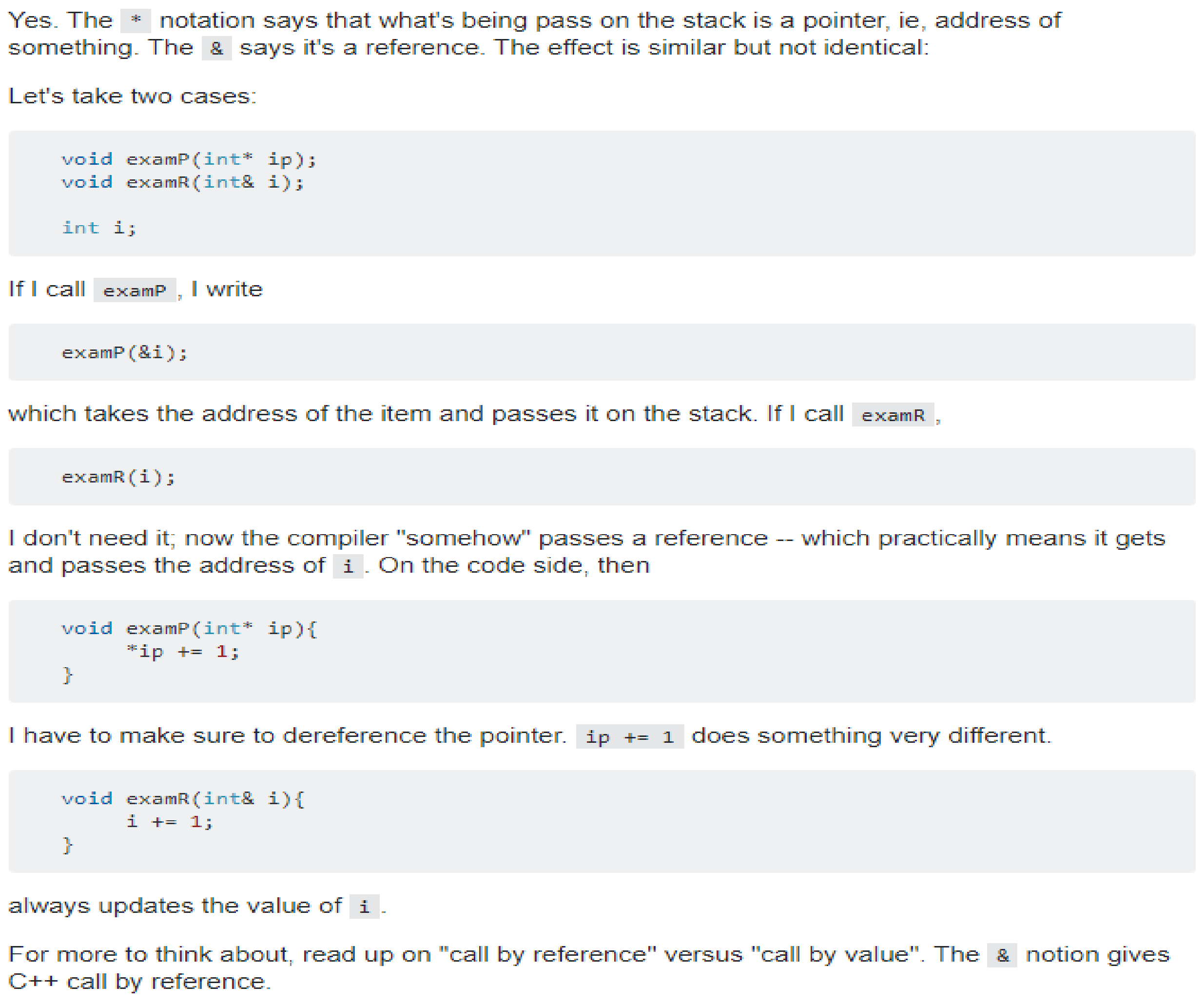
```cpp
private:
  //declare joint handle called joint_
  hardware_interface::JointHandle joint_;

  //targetPosition of UR5 after running this node will be -1
  static const double targetPosition_ = -1.00;
```

Then, we are defining JointHandle and targetPosition of the joint that will be controlled via our controller when we run this node.



```
bool init(hardware_interface::PositionJointInterface *hw,
          ros::NodeHandle &n) {
  std::string my_joint;
  if (!n.getParam("joint", my_joint)) {
    ROS_ERROR("Could not find joint name");
    return false;
  }

  joint_ = hw->getHandle(my_joint); // throws on failure
  return true;
}
```

Note: * is a pointer symbol and it means this function needs address of this argument when you call the init function. & is a reference symbol and it means you don't need to provide address of this argument when you call init function. They both do a similar job.
Here is more detailed explanation. [3]



Yes. The `*` notation says that what's being pass on the stack is a pointer, ie, address of something. The `&` says it's a reference. The effect is similar but not identical:

Let's take two cases:

```
void examP(int* ip);
void examR(int& i);

int i;
```

If I call `examP`, I write

```
examP(&i);
```

which takes the address of the item and passes it on the stack. If I call `examR`,

```
examR(i);
```

I don't need it; now the compiler "somehow" passes a reference -- which practically means it gets and passes the address of `i`. On the code side, then

```
void examP(int* ip){
     *ip += 1;
}
```

I have to make sure to dereference the pointer. `ip += 1` does something very different.

```
void examR(int& i){
     i += 1;
}
```

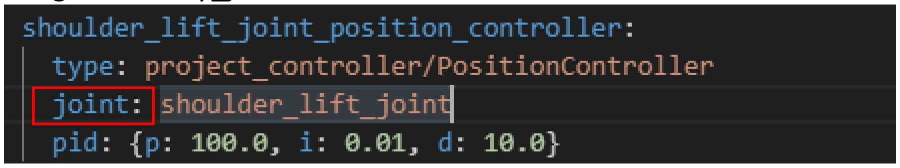
always updates the value of `i`.

For more to think about, read up on "call by reference" versus "call by value". The `&` notion gives C++ call by reference.

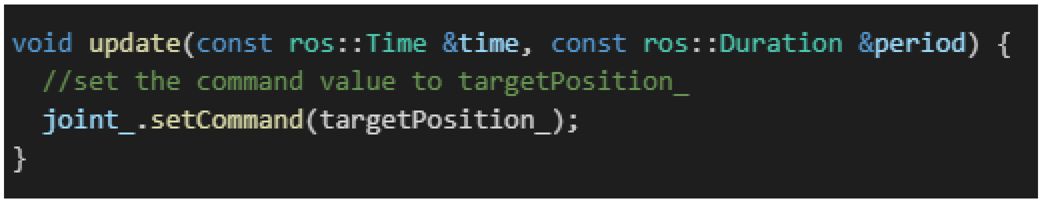
Figure 3. difference between & and * in function declaration in c++ [3]

So here, we are obtaining value of my_joint using “ros::NodeHandle” method “getParam” and key of ROS parameter server dictionary, “joint”. This method searches the key in the parameter server dictionary and returns a retrieved value. Then, we utilize “hardware_interface::PositionJointInterface” method, getHandle to get “hardware_interface::JointHandle”, joint_ (a.k.a a resource handle) by name of “my_joint”. This function will be called when our own controller is loaded by controller manager which belongs to ROS Control package.

e.g. here we can see key of parameter dictionary, “joint” and its element is “shoulder_lift_joint” in custom_control.yaml file. Therefore, when you call function, init, “shoulder_lift_joint” will be assigned to “my_joint”

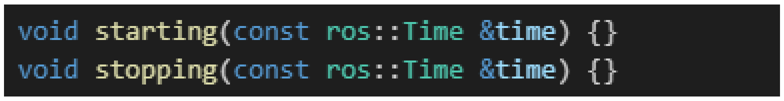


```
shoulder_lift_joint_position_controller:
  type: project_controller/PositionController
  joint: shoulder_lift_joint
  pid: {p: 100.0, i: 0.01, d: 10.0}
```



```
void update(const ros::Time &time, const ros::Duration &period) {
  //set the command value to targetPosition_
  joint_.setCommand(targetPosition_);
}
```

Then, we create “update” function that sends “targetPosition_” to the joint (in our case it is “shoulder_lift_joint”) by “hardware_interface::JointHandle”, “joint_”

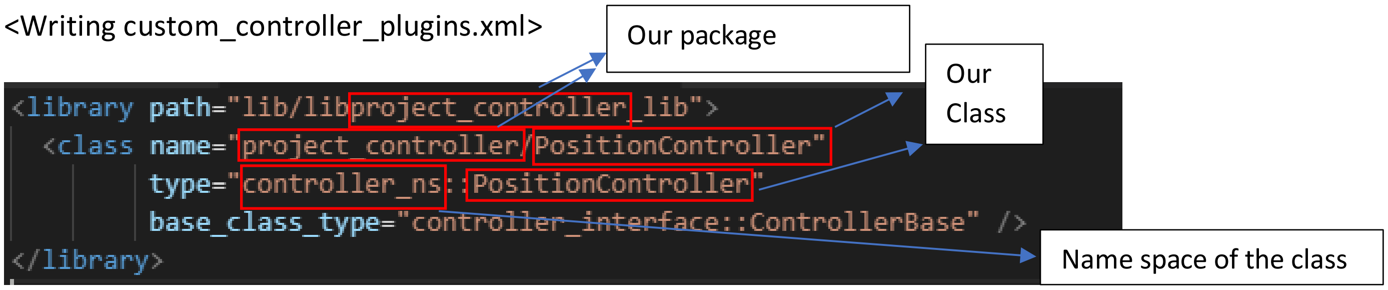


```
void starting(const ros::Time &time) {}
void stopping(const ros::Time &time) {}
```

Then, we define functions for starting and stopping our controller.

Once we are done writing source code for our own controller, the next thing to do is to create a plugin description file, custom_controller_plugins.xml in our case.

<Writing custom_controller_plugins.xml>

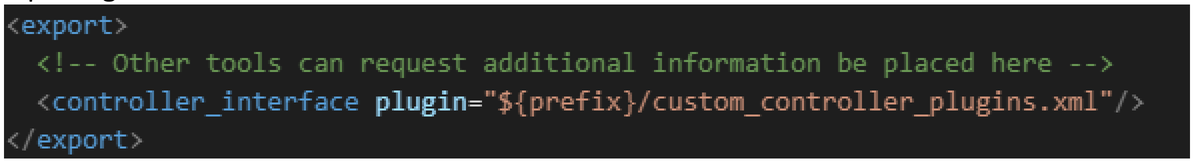


```
<library path="lib/libproject_controller_lib">
  <class name="project_controller/PositionController"
         type="controller_ns::PositionController"
         base_class_type="controller_interface::ControllerBase" />
</library>
```

This file defines path of library when it is compiled, name, type, and class of our controller that we have written in our source code.

Next, it is time to modify some existing files such as “package.xml”, “CMakeList.txt”, “custom_control.yaml”, and “quiz_control_custom.launch”.

< package.xml >

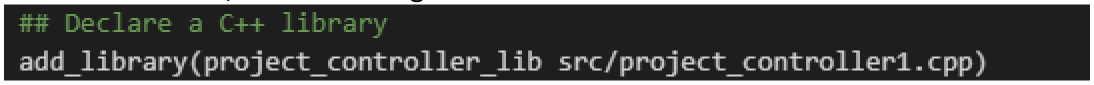


```xml
<export>
  <!-- Other tools can request additional information be placed here -->
  <controller_interface plugin="${prefix}/custom_controller_plugins.xml"/>
</export>
```

By adding this line, we let controller manager know that we are providing a plugin through custom_controller_plugins.xml so that the manager can fine and load our controller.

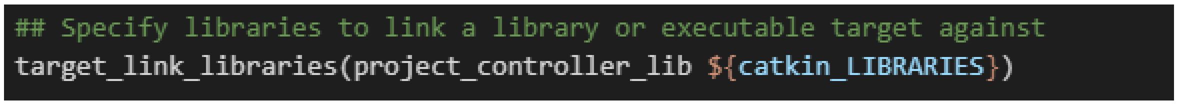
< CMakeList.txt >

In “CMakeList.txt, we are adding these two lines



```
## Declare a C++ library
add_library(project_controller_lib src/project_controller1.cpp)
```

This one is declaring a c++ library that has “project_controller1.cpp” which we wrote earlier for our own controller.

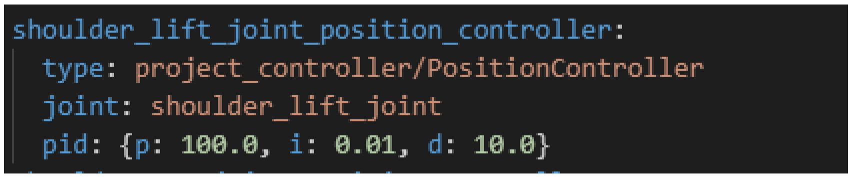


```
## Specify libraries to link a library or executable target against
target_link_libraries(project_controller_lib ${catkin_LIBRARIES})
```

Here, our target link libraries are a customized library, “project_controller_lib” and “${catkin_LIBRARIES}”.

<custom_control.yaml>

Since we want to control “shoulder_lift_joint” with our own controller, we need to modify its type from “position_controllers/JointPositionController” to “project_controller/PositionController” that we created. Here is an example below.

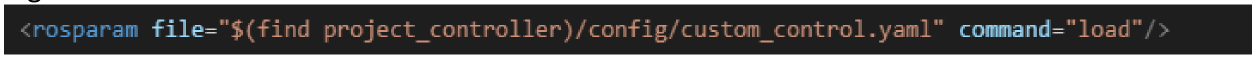


```yaml
shoulder_lift_joint_position_controller:
  type: project_controller/PositionController
  joint: shoulder_lift_joint
  pid: {p: 100.0, i: 0.01, d: 10.0}
```

<quiz_control_custom.launch>

Lastly, we need to load a correct configuration file in launch file. In our case, it will be “custom_control.yaml”

e.g.



```xml
<rosparam file="$(find project_controller)/config/custom_control.yaml" command="load"/>
```

The rest of part will remain the same as task1 (i.e., objective 1) unless you change the name of controllers in your configuration file.

As we conclude our discussion here, we have noticed that we can easily use default controller from ROS Control package by creating a few files. Furthermore, we can also create our own controller that does what we want a joint(s) of robot to do(es) by modifying contexts of some files. For future work, it would be more meaningful if we would control some robots for more advanced tasks.

References

[1] Quiz Contents Credit: Robot Ignite Academy

https://www.robotigniteacademy.com/en/course/ros-control-101/details/

[2] Report_QUIZ_TF_ ROS 101.pdf, Online Available:

https://github.com/Samwoose/Publishing_TF_PiRobot_N_Turtlebot/blob/master/Report_QUIZ_TF%20ROS%20101.pdf

[3] Difference between & and * in function declaration in c++, Online Available:

https://stackoverflow.com/questions/596636/c-difference-between-ampersand-and-asterisk-in-function-method-declar