Inhalt

1. Intruduction

The following comments refer to the current version of the ROS-Doosan driver, which is available today (June 2nd, 2020). They describe a problem of missing robot feedback.

2. Structure of the ROS interface

The basic structure of a ROS robot driver is shown in Figure 2-1.

- On application level (here "killer_app") a TCP position of the robot is generated.
- MoveIT! creates a collision free path matching the requested position, which consists of a trajectory in axis angles including a time stamp. The distances between the points are in the range of centimeters to millimeters.
- The robot independent controller (arm_controller) interpolates the created trajectory and forwards the individual points with high frequency to the hardware interface of the robot (RobotHW). The controller requires feedback from the robot at the same frequency regarding the current position, speed and moments of the individual axes. The control within the controller is done either via position, velocity or moment setpoints.

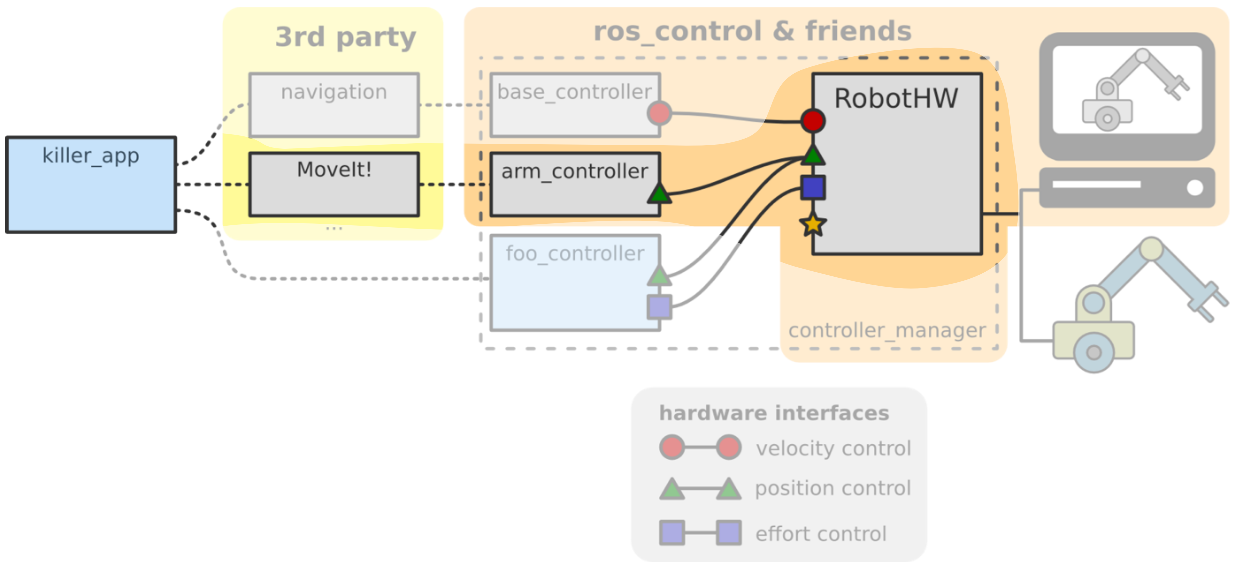

Figure 2-1 System diagram for ROS driver implementation [1]

In particular, a proper robot hardware interface must be implemented, for which ROS provides a corresponding base class RobotHW [2].

```
class MyRobot : public hardware_interface::RobotHW
{
public:
  MyRobot()
 {
   // connect and register the joint state interface
   hardware_interface::JointStateHandle state_handle_a("A", &pos[0], &vel[0], &eff[0]);
   jnt_state_interface.registerHandle(state_handle_a);

   hardware_interface::JointStateHandle state_handle_b("B", &pos[1], &vel[1], &eff[1]);
   jnt_state_interface.registerHandle(state_handle_b);

   registerInterface(&jnt_state_interface);

   // connect and register the joint position interface
   hardware_interface::JointHandle pos_handle_a(jnt_state_interface.getHandle("A"), &cmd[0]);
   jnt_pos_interface.registerHandle(pos_handle_a);

   hardware_interface::JointHandle pos_handle_b(jnt_state_interface.getHandle("B"), &cmd[1]);
   jnt_pos_interface.registerHandle(pos_handle_b);
   registerInterface(&jnt_pos_interface);
  }

private:
  hardware_interface::JointStateInterface jnt_state_interface;
  hardware_interface::PositionJointInterface jnt_pos_interface;
  double cmd[2];
  double pos[2];
  double vel[2];
  double eff[2];
};
```

The example shows that in particular a function for reading (`state_handle_a`) and a function for writing (`pos_handle_a`) the individual axis values must be implemented. The functions must be callable with high frequency.

Finally, the main function of the interface must only read and write the values it receives from the arm controller within an endless loop [2]:

```
main()
{
  MyRobot robot;
  controller_manager::ControllerManager cm(&robot);

  while (true)
  {
     robot.read();
     cm.update(robot.get_time(), robot.get_period());
     robot.write();
     sleep();
  }
}
```

3. Implementation of the read and write function in the Doosan driver

Basically Doosan implements the desired ROS structure in the class `dsr_control_node`. Here the read and write function is executed in a loop as specified [3].

```
while(ros::ok() && (false==g_nKill_dsr_control))
///while(g_nKill_dsr_control==false)
{
    try{
        ///ROS_INFO("[dsr_control] Running...(g_nKill_dsr_control=%d)",g_nKill_dsr_control);
        curr_time = ros::Time::now();
        elapsed = curr_time - last_time;
        if(pArm) pArm->read(elapsed);
        cm.update(ros::Time::now(), elapsed);
        if(pArm) pArm->write(elapsed);
        r.sleep();  //(1000/rate)[sec], default: 10ms
    }
    catch(std::runtime_error& ex)
    {
        //...
    }
}
```

Problem with the Read interface

In order for the controller to operate, it is necessary to read the axis values with at least 100 Hz. However, the read function is based on the data from `DRHWInterface::OnMonitoringDataCB`, which can only be called at 20 Hz according to the comment in the source code [4].

```
void DRHWInterface::OnMonitoringDataCB(const LPMONITORING_DATA pData)
    {
        // This function is called every 100 msec
        // Only work within 50msec
        //ROS_INFO("DRHWInterface::OnMonitoringDataCB");

        g_stDrState.nActualMode  = pData->_tCtrl._tState._iActualMode;
        g_stDrState.nActualSpace = pData->_tCtrl._tState._iActualSpace;

        for (int i = 0; i < NUM_JOINT; i++){
            if(pData){
                g_stDrState.fCurrentPosj[i] = pData->_tCtrl._tJoint._fActualPos[i];
                g_stDrState.fCurrentVelj[i] = pData->_tCtrl._tJoint._fActualVel[i];
                g_stDrState.fJointAbs[i]    = pData->_tCtrl._tJoint._fActualAbs[i];
                g_stDrState.fJointErr[i]    = pData->_tCtrl._tJoint._fActualErr[i];
                g_stDrState.fTargetPosj[i]  = pData->_tCtrl._tJoint._fTargetPos[i];
                g_stDrState.fTargetVelj[i]  = pData->_tCtrl._tJoint._fTargetVel[i];

        //..
```

Problem with the Write interface

Looking at the entire Write function [5], it is noticeable that the implementation has been started but not completed. Only a debug output is generated, while the actual write operation (`Drfl.MoveJAsync`) was commented out.

```
void DRHWInterface::write(ros::Duration& elapsed_time)
{
    //ROS_INFO("DRHWInterface::write()");
    static int count = 0;
    // joints.cmd is updated
    std::array<float, NUM_JOINT> tmp;
    for(int i = 0; i < NUM_JOINT; i++){
        ROS_DEBUG("[write]::write %d-pos: %7.3f %d-vel: %7.3f %d-cmd: %7.3f",
        i,
        joints[i].pos,
        i,
        joints[i].vel,
        i,
        joints[i].cmd);
        tmp[i] = joints[i].cmd;
    }
    if( !bCommand_ ) return;
    /*int state = Drfl.GetRobotState();
    if( state == STATE_STANDBY ){
        for(int i = 0; i < NUM_JOINT; i++){
            if( fabs(cmd_[i] - joints[i].cmd) > 0.0174532925 ){
                Drfl.MoveJAsync(tmp.data(), 50, 50);
                ROS_INFO_STREAM("[write] current state: " << GetRobotStateString(state));
                std::copy(tmp.cbegin(), tmp.cend(), cmd_.begin());
                break;
            }
        }
    }*/
}
```

4. Skipping the arm controller through the Doosan driver

As can be seen, the read and write functions of the driver have not been fully implemented and can therefore not be used successfully. The driver uses a shortcut as shown in Figure 4-1, where the arm controller is not used. To skip it, a separate callback function called `DRHWInterface::trajectoryCallback` is subscribed in the NodeHandle, which does not use either the Read or Write function [6]. So the Doosan robot intercepts the call (the goal) and performs the detailed interpolation itself.

```
m_sub_joint_trajectory =
private_nh_.subscribe("dsr_joint_trajectory_controller/follow_joint_trajectory/goal", 10,
&DRHWInterface::trajectoryCallback, this);
```

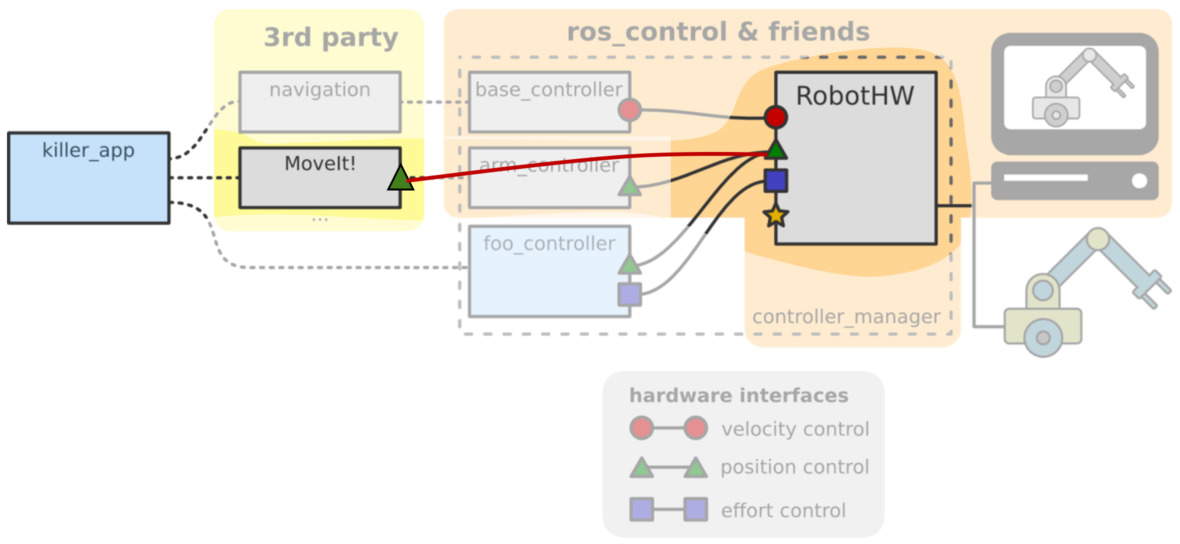

Figure 4-1: Skipping the arm controller in the Doosan driver

Basically, this type of implementation is okay (Universal Robots does the same thing, for example), as long as it is then fully implemented. Here, any kind of feedback is missing as well as the possibility to stop the robot via a stop action to change the path while the robot is running.

Summary and possible solutions

Implementation with arm controller (preferred)

For a correctly working implementation using the arm controller, the Read function must be callable at high frequency and the `Drfl.MoveJAsync` function must be suitable for this task. Since we have no insight into the internal control of the DRFL class, we ourselves cannot know why the frequency of the Read function is limited and whether the MoveJAsync function is suitable for the task. It would be important to have documentation the internal interface DRFL.h [7].

Implementation without arm controller

This would require the missing functions for feedback and stop action to be added. The implementation of Universal Robot can be used for comparison [8]. The file "Driver.py" is to be observed in particular. In this file the path is also tapped, but this is done with a so-called ActionServer and under consideration of the feedback to be sent.

5. List of sources

[1]
https://roscon.ros.org/2014/wp-content/uploads/2014/07/ros_control_an_overview.pdf

[2]
https://github.com/ros-controls/ros_control/wiki/hardware_interface

[3]
https://github.com/doosan-robotics/doosan-robot/blob/94cda64855ac75cec56205061b494ca92ffceec4/dsr_control/src/dsr_control_node.cpp#L73

[4]
https://github.com/doosan-robotics/doosan-robot/blob/94cda64855ac75cec56205061b494ca92ffceec4/dsr_control/src/dsr_hw_interface.cpp#L144

[5]
https://github.com/doosan-robotics/doosan-robot/blob/94cda64855ac75cec56205061b494ca92ffceec4/dsr_control/src/dsr_hw_interface.cpp#L987

[6]
https://github.com/doosan-robotics/doosan-robot/blob/94cda64855ac75cec56205061b494ca92ffceec4/dsr_control/src/dsr_hw_interface.cpp#L708

[7]
https://github.com/doosan-robotics/doosan-robot/blob/94cda64855ac75cec56205061b494ca92ffceec4/common/include/DRFL.h

[8]
https://github.com/ros-industrial/universal_robot/tree/kinetic-devel/ur_driver/src/ur_driver