Inhalt

1. Einleitung

Die folgenden Kommentare beziehen sich auf die heute (02.06.2020) aktuelle Version des ROS-Doosan-Treibers. Sie beschreiben ein Problem der fehlenden Roboter-Rückmeldung.

2. Aufbau der ROS-Schnittstelle

Der grundsätzlich Aufbau eines ROS-Roboter-Treibers ist in Abbildung 2.1 zu sehen.

- Auf Applikationsebene (hier „killer_app“) wird eine TCP-Position des Roboters generiert.
- MoveIT! erstellt passend zur angeforderten Position einen kollisionsfreien Pfad, welcher aus einer Trajektorie in Achswinkeln inklusive Zeitstempel besteht. Die Punktabstände sind dabei im Zentimeter- bis Millimeterbereich.
- Der roboterunabhängige Controller (arm_controller) interpoliert die erstellte Trajektorie und leitet die einzelnen Punkte mit hoher Frequenz an das Hardwareinterface des Roboters (RobotHW) weiter. Der Controller benötigt in gleicher Frequenz eine Rückmeldung des Roboters bezüglich der aktuellen Position, Geschwindigkeit und Momente der einzelnen Achsen. Die Regelung innerhalb des Controllers geschieht entweder über Sollwerte der Position, der Geschwindigkeit oder des Moments.

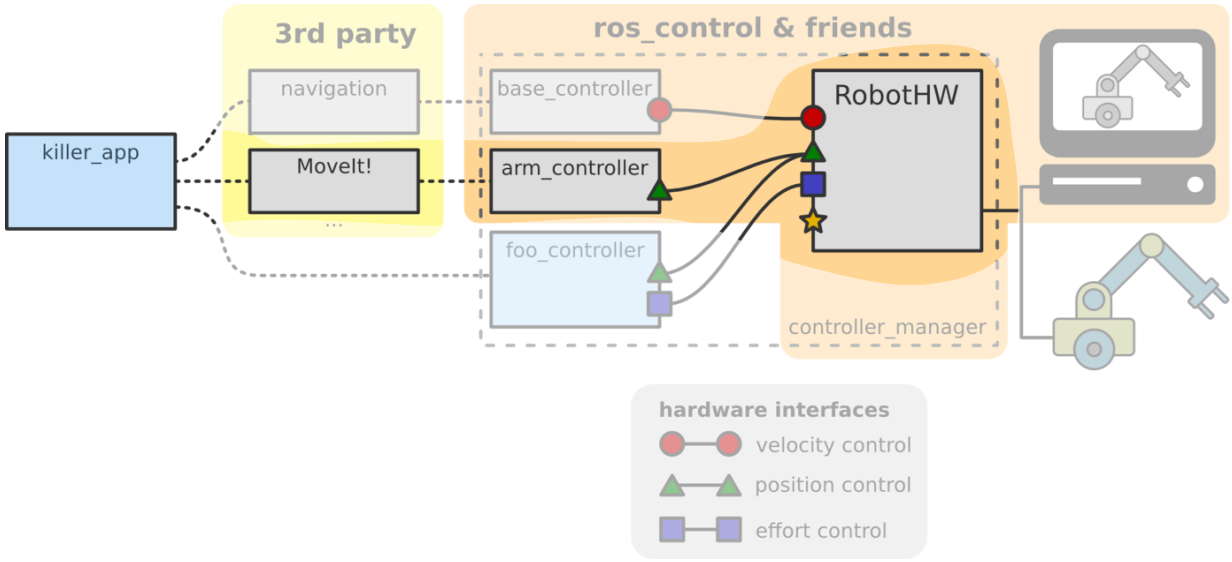

Abbildung 2.1: Systembild zur ROS-Treiber Implementation [1]

Zu implementieren ist also insbesondere ein sauberes Roboter-Hardware-Interface, wozu ROS eine entsprechende Basisklasse RobotHW bereitstellt [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];
};
```

Das Beispiel zeigt, dass insbesondere eine Funktion zum Lesen (`state_handle_a`) und eine Funktion zum Schreiben (`pos_handle_a`) der einzelnen Achswerte erstellt werden muss. Die Funktionen müssen mit hoher Frequenz aufrufbar sein.

Die main-Funktions des Interfaces muss schließlich nur noch innerhalb einer Endlos-Schleife die Werte, die es vom Arm-Controller bekommt, lesen und schreiben [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. Implementierung der Read- und Write-Funktion im Doosan-Treiber

Grundsätzlich implementiert Doosan die gewünsche ROS-Strunktur in der Klasse `dsr_control_node`. Hier wird wie vorgegeben in einer Schleife die Read- und Write-Funktion ausgeführt [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 der Read-Schnittstelle

Damit eine Steuerung durch den Controller funktionieren kann, ist das lesen der Achswerte mit mindestens 100 Hz erforderlich. Die Read-Funktion basiert jedoch letztlich auf den Daten von `DRHWInterface::OnMonitoringDataCB`, welche laut Kommentar im Quellcode nur mit 20 Hz aufgerufen werden kann [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 der Write-Schnittstelle

Schaut man sich die gesamte Write-Funktion an [5], so fällt auf, dass die Implementierung begonnen, jedoch nicht abgeschlossen wurde. Es wird lediglich eine Debug-Ausgabe generiert, während der eigentlich Schreibvorgang durch `Drfl.MoveJAsync` auskommentiert wurde.

```
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. Umgehen des Arm-Controllers durch den Doosan-Treiber

Wie zu sehen ist, wurden die Read- und Write-Funktionen des Treibers nicht vollständig Implementiert und können so nicht erfolgreich eingesetzt werden. Im Treiber wird eine Abkürzung genommen wie in Abbildung 4.1 zu sehen ist, wobei der Arm-Controller nicht verwendet wird. Um Ihn zu übergehen, wird eine eigene Callback-Funktion namens `DRHWInterface::trajectoryCallback` im NodeHandle hinterlegt, welche weder auf der Read- noch auf die Write-Funktion zurückgreift [6]. Der Doosan-Roboter fängt den Call (das goal) also ab und führt die detaillierte Interpolation selbst durch.

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

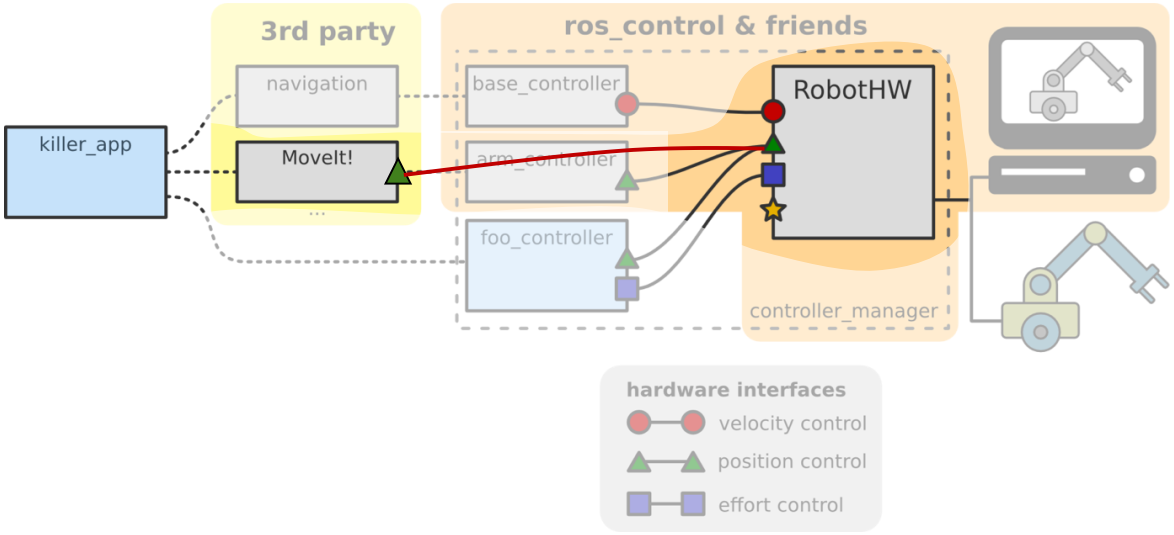

Abbildung 4.1: Umgehen des Arm-Controllers im Doosan-Treiber

Grundsätzlich ist diese Art der Implementierung in Ordnung (Universal Robots macht dies z.B. genau so), sofern sie dann vollständig Implementiert wird. Es fehlt hier jede Art der Rückmeldung sowie die Möglichkeit den Roboter über eine Stop-Action anzuhalten und die Bahn zu verändern.

5. Zusammenfassung und Lösungsmöglichkeiten

Implementierung mit Arm-Controller (bevorzugt)

Damit eine Implementierung mit Hilfe des Arm-Controllers funktioniert, muss die Read-Funktion in hoher Frequenz aufrufbar sein und die `Drfl.MoveJAsync` –Funktion für diese Aufgabe geeignet sein. Da wir keinen Einblick in die interne Steuerung der DRFL-Klasse haben, können wir selbst nicht wissen, warum die Frequenz der Read-Funktion begrenzt ist und ob die MoveJAsync-Funktion für die Aufgabe geeignet ist. Wichtig wäre hier eine Dokumentation der internen Schnittstelle DRFL.h [7].

Implementierung ohne Arm-Controler

Hierzu müssten die fehlenden Funktionen für die Rückmeldung und die Stop-Aktion ergänzt werden. Zum Vergleich kann hierzu die Implementierung von Universal Robot herangezogen werden [8]. Zu beachten ist besonders die Datei „Driver.py“. In dieser wird ebenfalls der Pfad abgegriffen, dies geschieht jedoch mit einem sog. ActionServer und unter Beachtungs des zu sendenden Feedbacks.

6. Quellenverzeichnis

[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