

1. Einführung in das ROS

Das ROS ist ein ‚opensource‘ Betriebssystem bestehend aus Werkzeugen, Bibliotheken, Gerätetreibern und Utilityfunktionen. Wie andere Betriebssysteme beinhaltet das ROS viele Dateien, die seine Funktionalität beschreiben: Packages, Manifests, Messages, Services, etc.

Der ROS-Master kann als der Namespace der Nodes und Services interpretiert werden. Ein echter Roboter bzw. die ROS-Verbindung mit einem echten Roboter startet den ROS-Master. Demgegenüber ist er mit dem Code **roscore** zu starten, wenn kein Roboter vorhanden ist.

2. Kommunikation innerhalb des ROS

Das ROS erstellt ein Netzwerk mit allen seinen Prozessen. Diese Kommunikation erfolgt über dem ROS-Master.

Ein ROS-Node ist der Prozess, der die Berechnungen durchführt. Häufig werden viele Nodes zusammen benutzt, um verschiedene und komplexe Funktionen anzusteuern. Jeder Node kann dieses Netzwerk erreichen, abonnieren oder dort etwas publizieren.

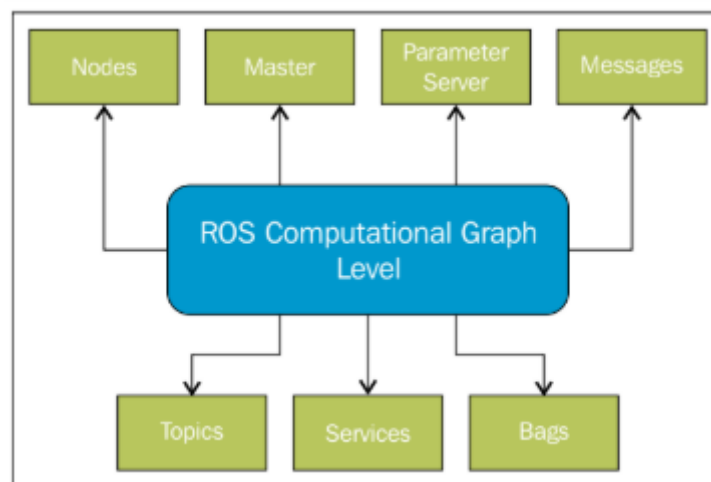


Abbildung 2.1. – ROS Computational Graph Level

3. Kommunikation der Nodes des e.DO-Roboters

Jedes in der Abbildung 2.1. dargestellte Element hat verschiedene Funktionen in diesem Netzwerk. Das ROS stellt außerdem verschiedene Tools zur Verfügung, damit dieses Netzwerk debuggt werden kann.

rqt_graph ist ein Tool, welches uns die aktiven Topics und Nodes grafisch darstellt. Dort werden die Nodes als Viereck und die Topics als Ellipse dargestellt.

rqt_topic ist ein anderes Tool des ROS, welches eine Liste der aktiven Topics erstellt. Folgende Abbildungen zeigen diese Tools.

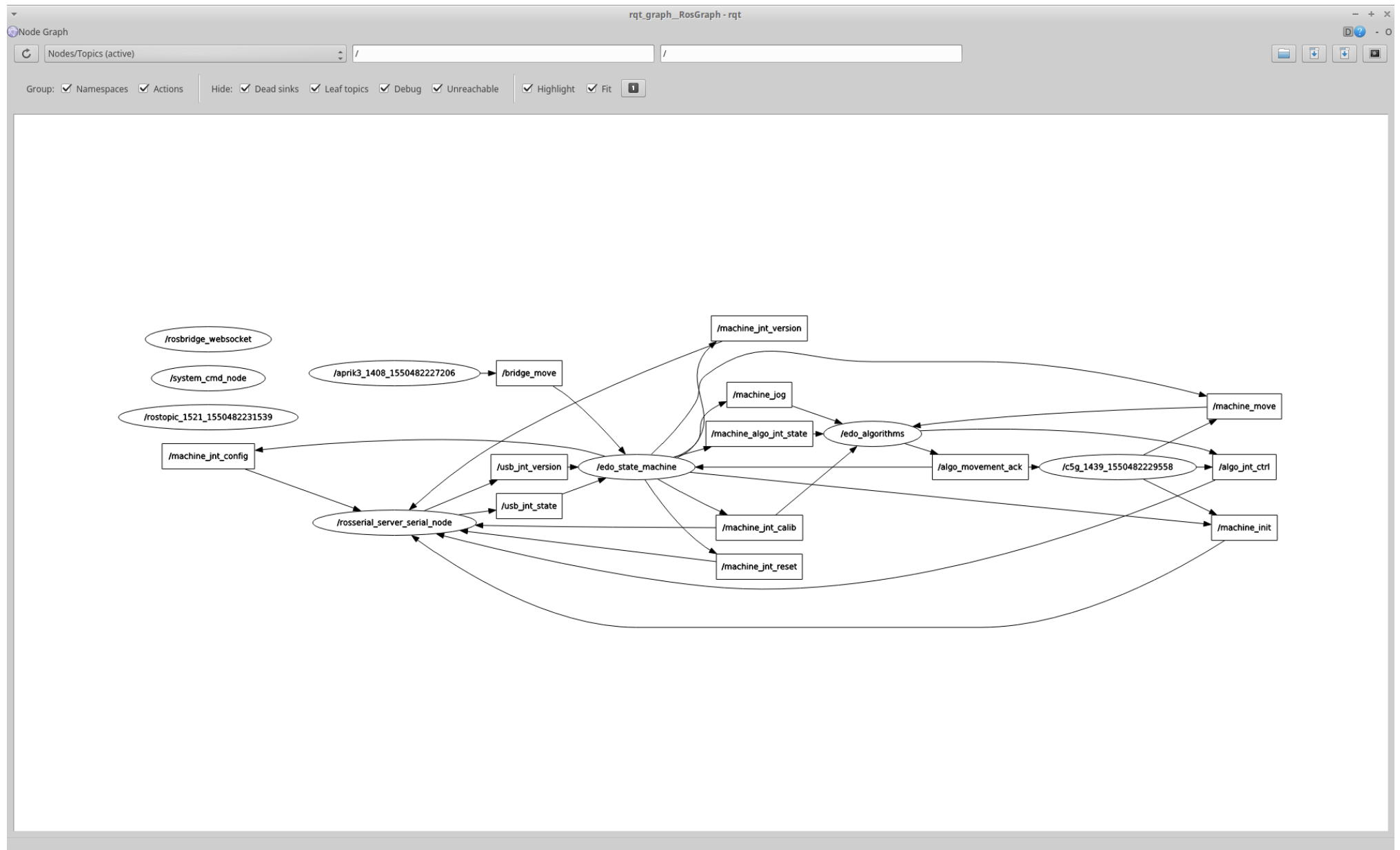


Abbildung 2.2. – Standardisierung der Nodes des e.DO-Roboters

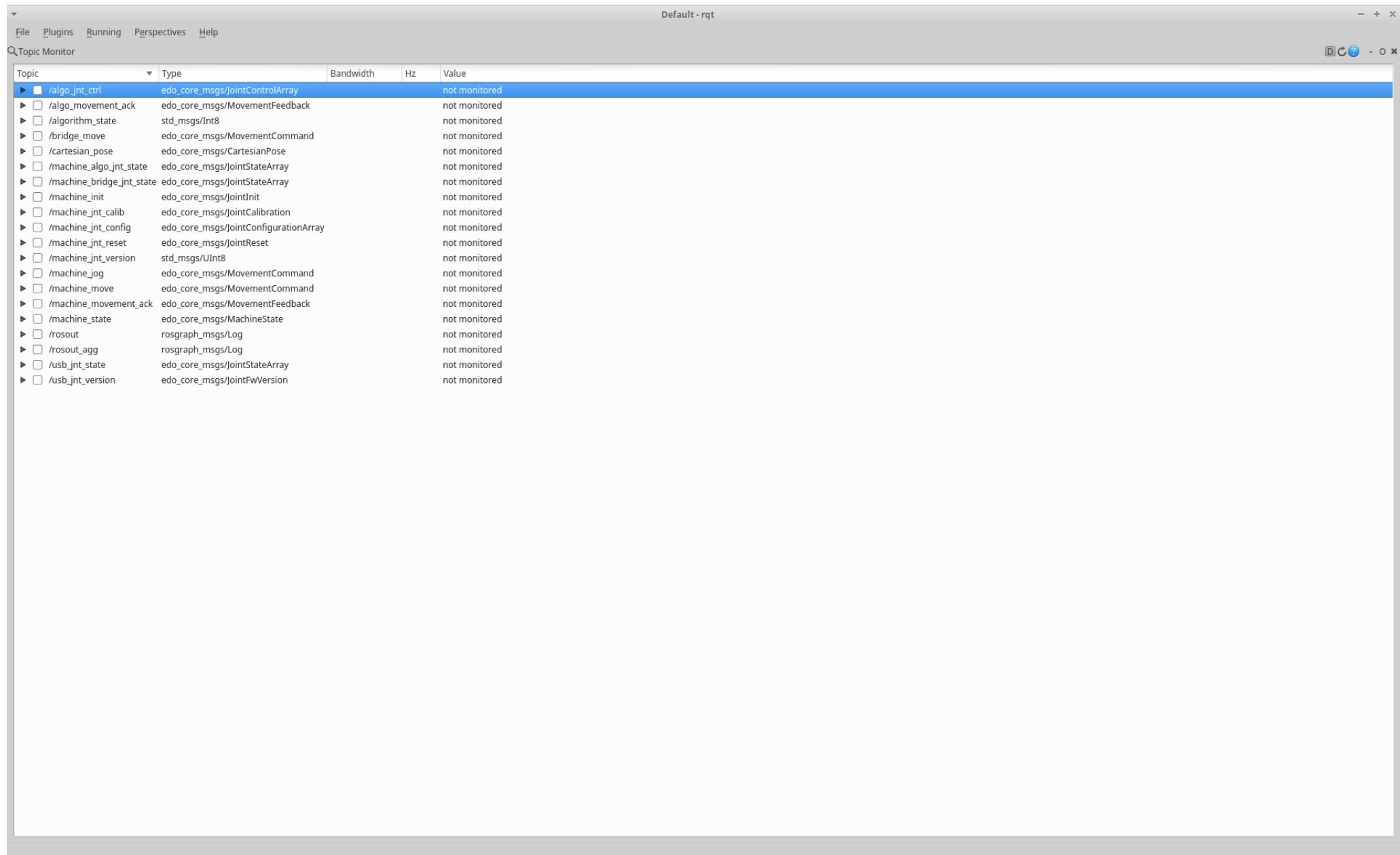


Abbildung 2.3. – Standardssituation des Nodes

Zwei der hier dargestellten Topics sind besonders wichtig. Sie leiten dem Roboter die Bewegungsbefehle weiter. **/bridge_jog** und **/bridge_move**.

/bridge_jog ist zurzeit nicht erkennbar, weil es noch von keinem Node gestartet wurde. Der erste Befehl wird es aktivieren, den auf dieses Topic gesendet wird.

/joint_states und **/machine_state** sind andere Topics, die uns über die Positionen der Joints und die Situation des Roboters informieren.

3. Verwendung von Sensoren mit dem e.DO-Roboter

Wie es davon in den vorherigen Kapiteln erwähnt wurde, unterstützt das ROS viele externe Geräte und stellt deren Treiber auch zur Verfügung. Ein Joystick ist auch eines dieser Geräte, das mit einem über ROS gesteuerten Roboter benutzt werden kann.

In den folgenden Abschnitten dieses Berichts wird ein Teleoperation-Node für x-Box Controller des Microsofts geschrieben. Da dieses Gerät ein von Linux unterstützter Joystick ist, lässt sich sein Treiber mit folgendem Code installieren:

```
$ sudo apt-get install ros-kinetic-joy
```

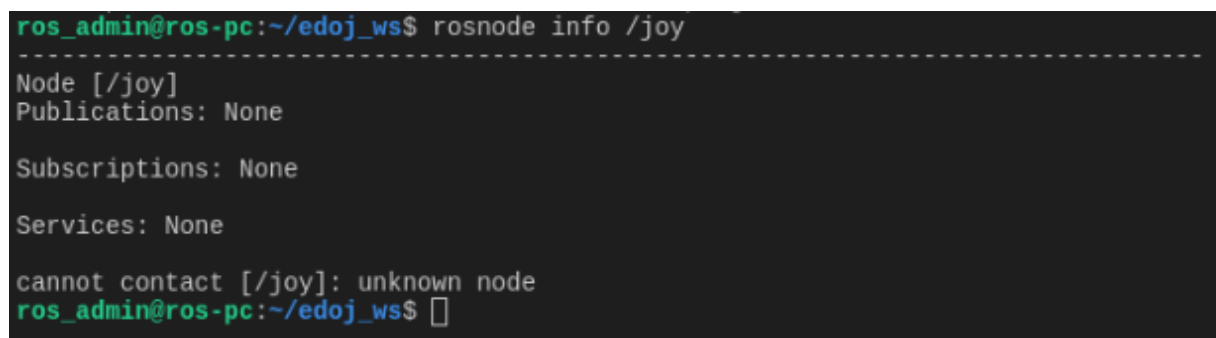
Hierbei wird die ROS-Kinetic-Version installiert. Für eine andere Version muss der Name des OS ersetzt werden.

Mit folgendem Code lässt sich der Joy-Node starten:

```
$ rosrun joy joy_node
```

Und mit folgendem Code kann man das Joy-Topic zuhören:

```
$ rostopic echo joy oder $ rostopic info joy
```



```
ros_admin@ros-pc:~/edoj_ws$ rosnode info /joy
-----
Node [/joy]
Publications: None

Subscriptions: None

Services: None

cannot contact [/joy]: unknown node
ros_admin@ros-pc:~/edoj_ws$
```

Abbildung 3.1. – **rosnode info joy**, bevor der Node gestartet wird

```

ros_admin@ros-pc:~/edoj_ws$ rostopic info /joy
Type: sensor_msgs/Joy

Publishers:
 * /joy_node (http://192.168.12.3:38639/)

Subscribers: None

ros_admin@ros-pc:~/edoj_ws$ █

```

Abbildung 3.2 - `rostopic info joy`, nachdem der Node gestartet wird. `sensor_msgs/Joy` ist der Message typ, den wir brauchen.

```

ros_admin@ros-pc:~/edoj_ws$ rostopic echo joy
header:
  seq: 1
  stamp:
    secs: 1554315762
    nsecs: 520366788
  frame_id: ''
axes: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
buttons: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
---
header:
  seq: 2
  stamp:
    secs: 1554315762
    nsecs: 592367362
  frame_id: ''
axes: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
buttons: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
---
█

```

```

---
header:
  seq: 36
  stamp:
    secs: 1554316537
    nsecs: 543679433
  frame_id: ''
axes: [-0.0, -0.007762945257127285, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
buttons: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
---
header:
  seq: 37
  stamp:
    secs: 1554316537
    nsecs: 551653979
  frame_id: ''
axes: [-0.0, -0.011136043816804886, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
buttons: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
---

```

Abbildung 3.3. und 3.4. - `rostopic echo joy` und verschiedene Reaktionen von Tasten und Achsen

```

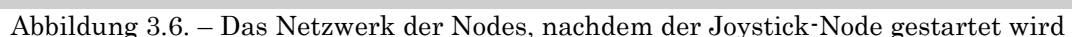
ros_admin@ros-pc:~/edoj_ws$ rosmmsg show sensor_msgs/Joy
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
float32[] axes
int32[] buttons

ros_admin@ros-pc:~/edoj_ws$ █

```

Abbildung 3.5. - `rosmmsg show sensor_msgs/Joy`

Abbildung 3.3 und 3.4 zeigen daher verschiedene Reaktionen der Achsen und der Tasten des Joysticks. In den beiden Fällen sind die Reaktionen anders, aber der Inhalt der Message ist immer gleich.



4. Implementierung der von Joystick erhaltenen Informationen

```
ros_admin@ros-pc:~/edoj_ws$ rostopic type /bridge_jog
edo_core_msgs/MovementCommand
```

Der gleiche Vorgang kann für jedes Topic angewendet werden. Im Endeffekt hat man dann immer den Namen der Message.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

ros_admin@ros-pc:~/edoj_ws$ rosmmsg show edo_core_msgs/MovementCommand
uint8 move_command
uint8 move_type
uint8 ovr
uint8 delay
uint8 remote_tool
float32 cartesian_linear_speed
edo_core_msgs/Point target
  uint8 data_type
  edo_core_msgs/CartesianPose cartesian_data
    float32 x
    float32 y
    float32 z
    float32 a
    float32 e
    float32 r
    string config_flags
  uint64 joints_mask
  float32[] joints_data
edo_core_msgs/Point via
  uint8 data_type
  edo_core_msgs/CartesianPose cartesian_data
    float32 x
    float32 y
    float32 z
    float32 a
    float32 e
    float32 r
    string config_flags
  uint64 joints_mask
  float32[] joints_data
edo_core_msgs/Frame tool
  float32 x
  float32 y
  float32 z
  float32 a
  float32 e
  float32 r
edo_core_msgs/Frame frame
  float32 x
  float32 y
  float32 z
  float32 a
  float32 e
  float32 r
```

- Abbildung 4.2. - **rosmmsg show edo_core_msgs/MovementCommand** gibt an, welche Informationen an den Roboter weitergeleitet werden, um ihn zu bewegen.

Die Informationen, die oben in der Abbildung 4.2. dargestellt werden, müssen also auf das entsprechende Topic publiziert werden, damit sich der Roboter bewegen kann.

Unser Ziel ist es, zu erfahren, wie das Tablet mit welchem Inhalt diese Informationen schickt. **/rostopic echo bridge_jog** ermöglicht uns wieder, dass wir dem Topic zuhören.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

[100%] Built target edo_manual_ctrl
ros_admin@ros-pc:~/edoj_ws$ rostopic echo bridge_jog
move_command: 74
move_type: 74
ovr: 100
delay: 0
remote_tool: 0
cartesian_linear_speed: 0.0
target:
  data_type: 74
  cartesian_data:
    x: 0.0
    y: 0.0
    z: 0.0
    a: 0.0
    e: 0.0
    r: 0.0
    config_flags: ''
  joints_mask: 127
  joints_data: [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
via:
  data_type: 0
  cartesian_data:
    x: 0.0
    y: 0.0
    z: 0.0
    a: 0.0
    e: 0.0
    r: 0.0
    config_flags: ''
  joints_mask: 0
  joints_data: []
tool:
  x: 0.0
  y: 0.0
  z: 0.0
  a: 0.0
  e: 0.0
  r: 0.0
frame:
  x: 0.0
  y: 0.0
  z: 0.0
  a: 0.0
  e: 0.0
  r: 0.0
---
```

Abbildung 4.3. – Die von Tablet auf das Topic publizierten Informationen

Nachdem wir das Topic mit dem Code ‚abonniert‘ haben, schicken wir vom Tablet einen Bewegungsbefehl für den Joint 1. Die Abbildung 4.3. zeigt, welchen Inhalt wir dabei publiziert haben.

Uns ist es jetzt klar geworden, welche Informationen für eine Bewegung eines Joints geschickt werden sollen. Reintheoretisch kann man einen ähnlichen Inhalt auch vom Computer schicken, um ihn zu bewegen.

/bridge_init und **/bridge_jnt_reset** sollen die anderen wichtigen Topics sein, die wir uns benutzen werden. Wir erkundigen uns dabei von **rqt_topic**.

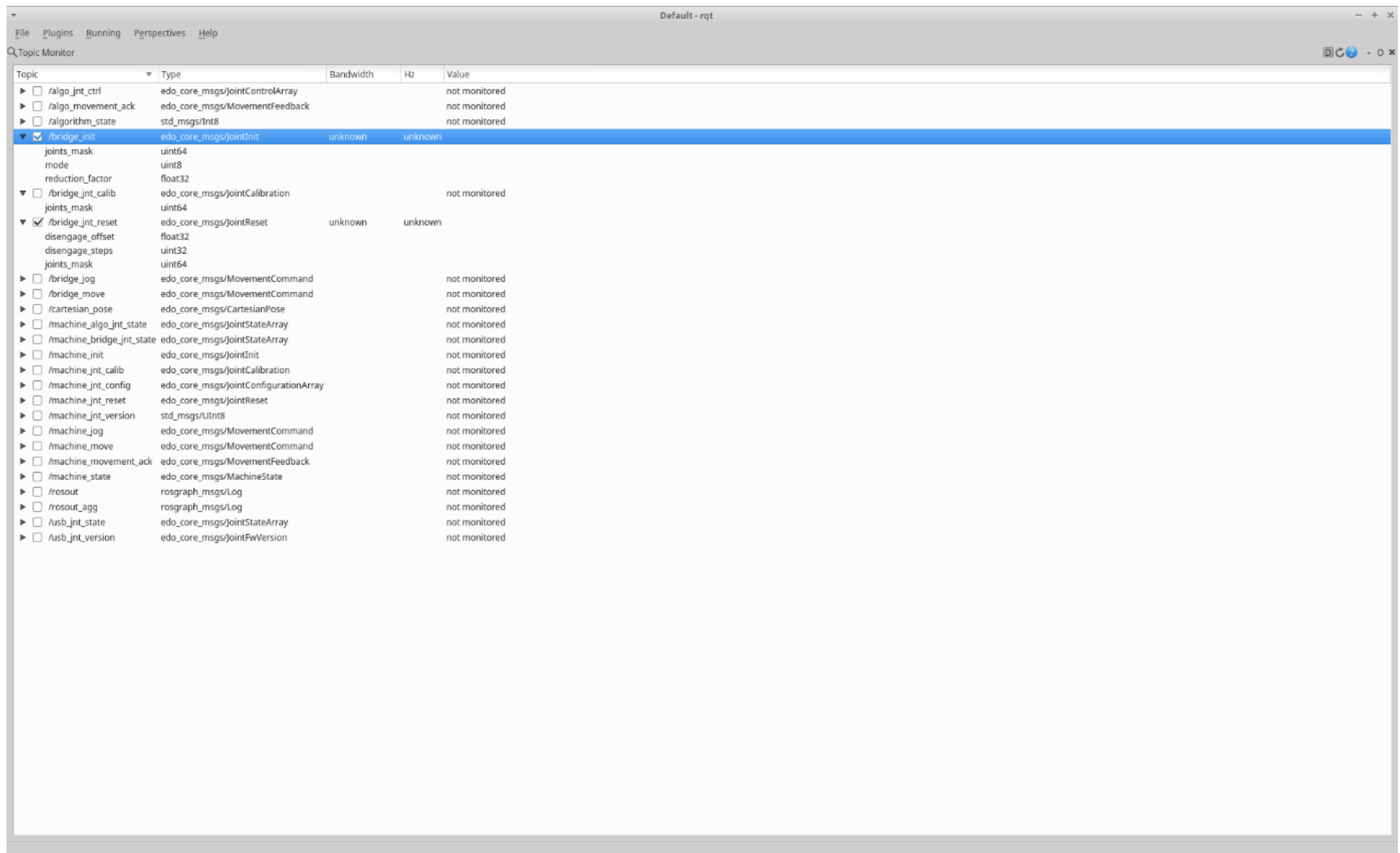
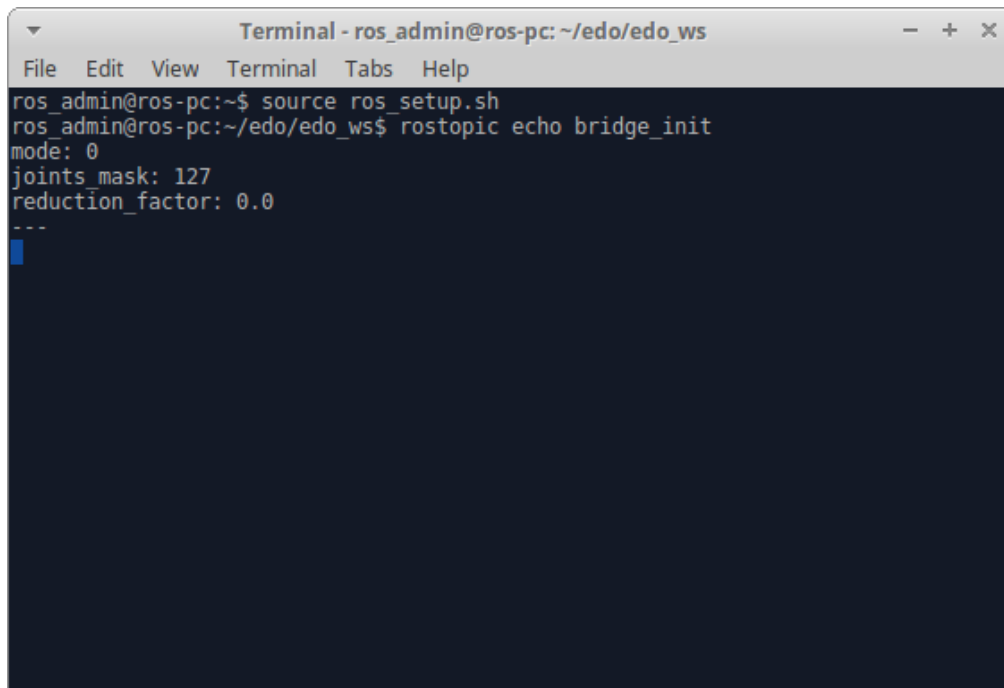


Abbildung 4.4. – Das Feedback vom `rqt_topic`

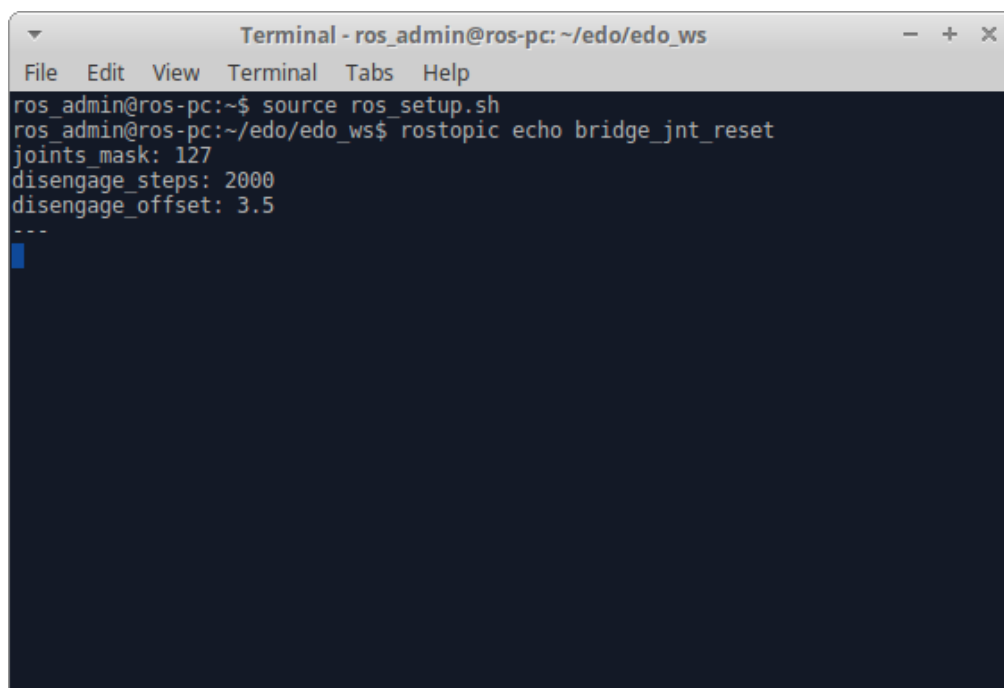
Die Abbildung 4.4. gibt uns die Auskunft über die Typen und den Inhalt der Messages, die auf dieses Topic publiziert werden.

Mit folgenden Zeilen, `$ rostopic echo /bridge_init` und `$ rostopic echo /bridge_jnt_reset` hören wir den sogenannten Topic zu. Da wir davon ausgegangen sind, dass diese beiden Topics den Roboter initialisieren und seine Bremsen auslösen, soll der Roboter zunächst neugestartet werden.

Folgende Abbildungen informieren uns über den Output.



```
Terminal - ros_admin@ros-pc: ~/edo/edo_ws
File Edit View Terminal Tabs Help
ros_admin@ros-pc:~$ source ros_setup.sh
ros_admin@ros-pc:~/edo/edo_ws$ rostopic echo bridge_init
mode: 0
joints_mask: 127
reduction_factor: 0.0
---
```



```
Terminal - ros_admin@ros-pc: ~/edo/edo_ws
File Edit View Terminal Tabs Help
ros_admin@ros-pc:~$ source ros_setup.sh
ros_admin@ros-pc:~/edo/edo_ws$ rostopic echo bridge_jnt_reset
joints_mask: 127
disengage_steps: 2000
disengage_offset: 3.5
---
```

Abbildung 4.5. und 4.6. – Die vom Tablet auf verschiedene Topics publizierten Informationen

Nach dem Neustarten des Roboters wird er wieder vom Tablet initialisiert und oben dargestellte Informationen bekommen.

In diesem Abschnitt ist es uns jetzt bekannt geworden, wie wir den Roboter starten, initialisieren, seine Bremsen auslösen und Bewegungsbefehle schicken.

5. Schreiben eines Teleoperation Nodes für den e.DO-Roboter und x-Box-Controller

In dem folgenden Abschnitt beschäftigen wir uns mit der Implementierung eines eigenen Nodes, dessen Aufgabe es ist, den Roboter anzusteuern, indem er ähnliche Befehle über das ROS schickt.

```
1  #include <ros/ros.h>
2  #include "edo_core_msgs/MovementCommand.h" // for move commands
3  #include "edo_core_msgs/JointReset.h" // for joint reset commands
4  #include "edo_core_msgs/JointInit.h" // for the initialization of robot
5  #include <sensor_msgs/Joy.h> // includes the joystick messages, so that we can listen the joy topic
6  #include <iostream>
7  #include <queue>
8  #include <ncurses.h>
9  #include <stdlib.h>
10 #include <stdio.h>
11 #include <thread>
12 #include <chrono>
13 #include <fstream>
14 #include <queue>
15 #include <string>
16 #include <iomanip>
```

In diesem Teil werden für den Joystick die `/sensor_msgs` und für die Kommunikation mit dem Roboter die `/edo_core_msgs` hinzugefügt. Der Rest ist für verschiedene Aktivitäten des C++-Codes.

```
18 //Global variables for axes and buttons
19 double intturnA = 0;
20 double intturnB = 0;
21 double intturnC = 0;
22 double intturnD = 0;
23 double intturnE = 0;
24 double intturnF = 0;
25 double intturnG = 0;
26
27 //Global variables for velocity;
28 double vel = 1;
29 double velocitymax = 1.1;
30 double velocityup = 0;
31 double velocitydown = 0;
32 double velocitymin = 0;
```

Hier wurden verschiedene globale Variablen definiert, um diese später überall im Code verwenden zu können.

```

34 //Shortcut for the robot - Fixed Values for a "jog" move
35 edo_core_msgs::MovementCommand createJog(){
36     edo_core_msgs::MovementCommand msg;
37     msg.move_command = 74;
38     msg.move_type = 74;
39     msg.ovr = 100;
40     msg.target.data_type = 74;
41     msg.target.joints_mask = 127; // Without gripper it should be 63
42     msg.target.joints_data.resize(10, 0.0);
43     return msg;
44 }
45

```

Der vorher bestimmte Inhalt des Bewegungsbefehls wird hier als eine Funktion definiert. Das Experimentieren mit dem Roboter hat es gezeigt, dass die Joints-Maske für ein Modell ohne Greifer 63 sein sollte.

```

62 eD0Teleop::eD0Teleop()
63 {
64     //Joystick subscriber
65     joy_sub_ = nh_.subscribe<sensor_msgs::Joy>("joy", 10, &eD0Teleop::joyCallback, this);
66     //ROS Publisher to "/bridge_jog" topic
67     jog_ctrl_pub = nh_.advertise<edo_core_msgs::MovementCommand>("bridge_jog", 10);
68 }
69
70 void eD0Teleop::joyCallback(const sensor_msgs::Joy::ConstPtr& joy)
71 //Callback function for the joystick
72 {
73     int i;
74
75     for (unsigned i = 0; i < joy->axes.size(); ++i) {
76         //ROS_INFO("Axis %d is now at position %f", i, joy->axes[i]);
77     }
78

```

Als nächstes kommen die Definition der Klasse und darin die **joyCallback** Funktion, um die Sensor-Messages vom Joystick zu bekommen. Hier werden außerdem Subscriber und Publisher für eine spätere Benutzung definiert.

Die for-Schleife im void ermöglicht es, die vorher genannten Variablen mit den Joystick-Informationen noch einmal zu definieren. Für die Bewegungen der Joints werden die Achsen und Tasten des Joysticks benutzt, während für die Eingabe der Geschwindigkeit nur die Tasten benutzt werden.

Die unten dargestellte Abbildung zeigt die **velocity** Funktion ermöglicht es, den Eingabebereich der Geschwindigkeit zu bestimmen. Diese Funktion reagiert auf die Taste A und B auf dem Joystick.

```

97  double velocity(double veloc)
98  {
99      if (velocityup==1 && veloc<velocitymax)
100      {
101          //ROS_INFO("Vel %f", vel);
102          //ROS_INFO("velocity up %f", velocityup);
103          vel=veloc+0.05;
104      }
105
106      if (velocitydown==1 && veloc>velocitymin)
107      {
108          //ROS_INFO("Vel %f", vel);
109          //ROS_INFO("velocity down %f", velocitydown);
110          vel=veloc-0.05;
111      }
112
113      return veloc;
114  }

```

```

116  //Definition of the correctvalues function - so that the robot does not react
117  //to all movements of the joystick.
118  double correctvalues(double value){
119
120      double calibrator=0.3;
121      double retval=0;
122
123      if (value > 0 && value < 0.3)
124      {
125          retval = 0;
126      }
127
128      if (value > -0.3 && value < 0)
129      {
130          retval=0;
131      }
132
133      if (value == 0)
134      {
135          retval=0;
136      }
137
138      if (value == -0.0)
139      {
140          retval=0;
141      }
142
143      if (value>calibrator && value<1.0)
144      {
145          retval=-1;
146      }
147
148      if (value>-1.0 && value<-0.5)
149      {
150          retval=1;
151      }
152
153      if (value ==-1)
154      {
155          retval=1;
156      }
157
158      if (value ==1)
159      {
160          retval=-1;
161      }
162      return retval;
163  }
164

```

Die oben dargestellte Abbildung zeigt die **correctvalues** Funktion. Die verhindert die Sensibilität des Joysticks. Dass die Befehle ab einem Wert von 0.3 an den Roboter geschickt werden, wird hier definiert.

```
165 int main(int argc, char** argv)
166 {
167     //Initialize "eDOTEleop" ROS Node
168     //ROS_INFO("Start ROS");
169
170     ros::init(argc, argv, "eDOTEleop");
171     eDOTEleop eDOTEleop;
172
173     ros::NodeHandle nh_;
174     char proceed = '\n';
175     edo_core_msgs::MovementCommand msg = createJog();
176
177     ros::Rate loop_rate(100); // Loop frequency
178     ros::Publisher jog_ctrl_pub = nh_.advertise<edo_core_msgs::MovementCommand>("bridge_jog", 10);
179
180     //ROS_INFO("Start while");
181
182     //Robot startup messages
183     ros::Publisher reset_pub = nh_.advertise<edo_core_msgs::JointReset>("/bridge_jnt_reset",10); //ROS Publisher to publish joint reset command
184     ros::Publisher init_pub = nh_.advertise<edo_core_msgs::JointInit>("/bridge_init",10); //ROS Publisher to publish robot init command
185     std::chrono::milliseconds timespan(10000); //Time for initialization
186     edo_core_msgs::JointReset reset_msg; //Definition of the messages
187     edo_core_msgs::JointInit init_msg;
```

Als nächstes initialisieren wir unsere ROS-Node und erstellen den eDOTEleop-Node. Nachher werden die Publisher für die Initialisierung des Roboters definiert.

```
189     std::cout << "\033[2J\033[1;1H"; // Clean the screen;
190     std::this_thread::sleep_for(timespan/4); //For the joy_node;
191     while (proceed!='y'){
192         std::cout << "Enter 'y' to initialize 6-Axis e.DO-Robot with gripper.\n"
193             << "The process takes 10 secs. == ";
194         std::cin >> proceed;
195     }
196
197     proceed = '\n';
198     init_msg.mode = 0; //Fixed values to start up the robot
199     init_msg.joints_mask = 127;
200     init_msg.reduction_factor = 0.0;
201
202     while(init_pub.getNumSubscribers() == 0){
203         loop_rate.sleep();
204     }
205
206     init_pub.publish(init_msg);
207     ros::spinOnce();
208     loop_rate.sleep();
209
210     std::cout << "\033[2J\033[1;1H"; // Clean the screen;
211     std::this_thread::sleep_for(timespan); // While e.DO starts up
212
213     while(proceed != 'y'){
214         std::cout << "Automatic motion! \n"
215             << "The robot will move, type 'y' to disengage breaks == ";
216         std::cin >> proceed;
217     }
218
219     proceed = '\n';
220     reset_msg.joints_mask = 127; //Fixed values to disengage the breaks
221     reset_msg.disengage_steps = 2000;
222     reset_msg.disengage_offset = 3.5;
223
224     while(reset_pub.getNumSubscribers() == 0){
225         loop_rate.sleep();
226     }
227     reset_pub.publish(reset_msg);
228     ros::spinOnce();
229     loop_rate.sleep();
230     //End of robot startup
```


Die vorher definierten Publishers werden hier benutzt, um den Roboter zu initialisieren und seine Bremsen auszulösen.

Nachdem der Roboter richtig gestartet wird, kommen wir langsam zu unserem Ziel.

```
234 //Visualisation of joystick interface
235 std::cout
236 <<"_____Welcome on board!_____\\n"
237 <<"|-----|*\\n"
238 <<"|      LB      \\      RB      \\      |*\\n"
239 <<"|  /      \\      < >      < >      (Y)  |*\\n"
240 <<"|  //JL\\      < >      < >      (X)  (B)|*\\n"
241 <<"|  \\_//      < >      < >      (A)  |*\\n"
242 <<"|  |      |      //JR\\      |*\\n"
243 <<"|  |< >|      \\_//      |*\\n"
244 <<"|  |v|      |*\\n"
245 <<"|  |      |*\\n"
246 <<"|  |      |*\\n"
247 <<"|  |      |*\\n"
248 <<"|  \\      /      \\      /      |*\\n"
249 <<"|  \\      /      \\      /      |*\\n"
250 <<"|-----|*\\n"
251 <<"You are in the joystick mode of the e.DO robot.\\n"
252 <<"|-----|*\\n"
253 <<"For Joint 1 -> Left joystick up and down \\n"
254 <<"For Joint 2 -> Left joystick left and right\\n"
255 <<"For Joint 3 -> Right joystick up and down\\n"
256 <<"For Joint 4 -> Right joystick left and right\\n"
257 <<"For Joint 5 -> ^ for up and v for down\\n"
258 <<"For Joint 6 -> < for left and > for right\\n"
259 <<"For gripper -> RB to open and **RB and RT** to close\\n"
260 <<"-----\\n"
261 <<"Button A to speed up and Button B to speed down\\n"
262 << std::flush;
```

Die Benutzeroberfläche des Nodes wird hier genauer beschrieben und eine Bedienungsanleitung erstellt.

Als nächstes kommt der wichtigste Teil des Codes: Der Teil, der dem Roboter diese Informationen weiterleitet.

Zunächst werden die Werte jeder einzelnen Message definiert. Dann wird für jeden Joint eine Funktion erstellt, die aus vorher definierten Funktionen besteht. Mit dem Code `msg.target.joints_data` wird die Joints informiert, wie ‚viel‘ sie bewegen sollen.

Wichtig ist es, dass die Joints als *eine Liste* im System gespeichert werden. Joint 1 befindet sich daher auf dem ‚nullten‘ Platz. `usleep()` bestimmt die Frequenz der Messages.

```

263     while(true){
264         //End of the visualisation of joystick interface
265         //Fixed values for a "jog" move
266         msg.move_command=74;
267         msg.move_type=74;
268         msg.ovr=100;
269         msg.target.data_type = 74;
270         msg.target.joints_mask = 127;
271
272
273         //With correctvalues() - Publish the messages to the robot
274         msg.target.joints_data[0] = (0,(velocity(vel))*correctvalues(intturnA));
275         jog_ctrl_pub.publish(msg);
276
277         msg.target.joints_data[1] = (0,(velocity(vel))*correctvalues(intturnB));
278         jog_ctrl_pub.publish(msg);
279
280         msg.target.joints_data[2] = (0,(velocity(vel))*correctvalues(intturnC));
281         jog_ctrl_pub.publish(msg);
282
283         msg.target.joints_data[3] = (0,(velocity(vel))*correctvalues(intturnD));
284         jog_ctrl_pub.publish(msg);
285
286         msg.target.joints_data[4] = (0,(velocity(vel))*correctvalues(intturnE));
287         jog_ctrl_pub.publish(msg);
288
289         msg.target.joints_data[5] = (0,(velocity(vel))*correctvalues(intturnF));
290         jog_ctrl_pub.publish(msg);
291
292         //For gripper control
293         msg.target.joints_data[6] = (0,(velocity(vel))*intturnG);
294         jog_ctrl_pub.publish(msg);
295
296         usleep(100000);
297         ros::spinOnce();
298     }
299 }

```

6. Erstellen einer richtigen Umgebung

```

$ cd ~/catkin_ws/src

$ catkin_create_pkg joyteleop roscpp joy sensor_msgs
message_generation edo_core_msgs std_msgs

$ cd ~/catkin_ws/

$ catkin_make

```

Mit diesem Code wird ein neuer Ordner im Catkin-Workspace erstellt und kompiliert.

Der im letzten Abschnitt geschriebene und erklärte Code muss jetzt in die Datei joyteleop.cpp kopiert werden. Diese Datei soll im Verzeichnis **joyteleop/src** erstellt werden.

In die Package.xml-Datei im Joyteleop-Ordner sind folgende Zeilen hinzuzufügen:

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>joy</build_depend>
<build_depend>roscpp</build_depend>
<build_export_depend>joy</build_export_depend>
<build_export_depend>roscpp</build_export_depend>
<exec_depend>joy</exec_depend>
<exec_depend>roscpp</exec_depend>
<exec_depend>joy</exec_depend>
<depend>edo_core_msgs</depend>
<depend>std_msgs</depend>
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
<depend>sensor_msgs</depend>
```

In die CMakeLists.txt-Datei in demselben Ordner sind folgende Zeilen hinzuzufügen:

```
add_compile_options(-std=c++11)
find_package(Curses REQUIRED)
generate_messages(
  DEPENDENCIES
  edo_core_msgs
  std_msgs
  sensor_msgs)
catkin_package(
  #INCLUDE_DIRS include
  CATKIN_DEPENDS joy roscpp message_runtime edo_core_msgs
  std_msgs sensor_msgs)
include_directories(
  include
  ${catkin_INCLUDE_DIRS}
  {CURSES_INCLUDE_DIR})
add_executable(joyteleop src/joyteleop.cpp)
target_link_libraries(joyteleop ${catkin_LIBRARIES})
```

6.1. Andere Abhängigkeiten des Packages

- NCurses dient zu einer asynchronen Kontrolle der Joints und ist online herunterzuladen und zu installieren.
- eDO_core_msgs-Ordner beinhaltet die Messages für die Kommunikation des Roboters. Auf der GitHub-Seite vom Comau lässt sich finden.
- C++ 11

Diese Abhängigkeiten des Packages müssen installiert werden, bevor das Package kompiliert werden. Ansonsten lässt es sich sowieso nicht kompilieren. ☺

6.2. Erstellen einer Start-Datei

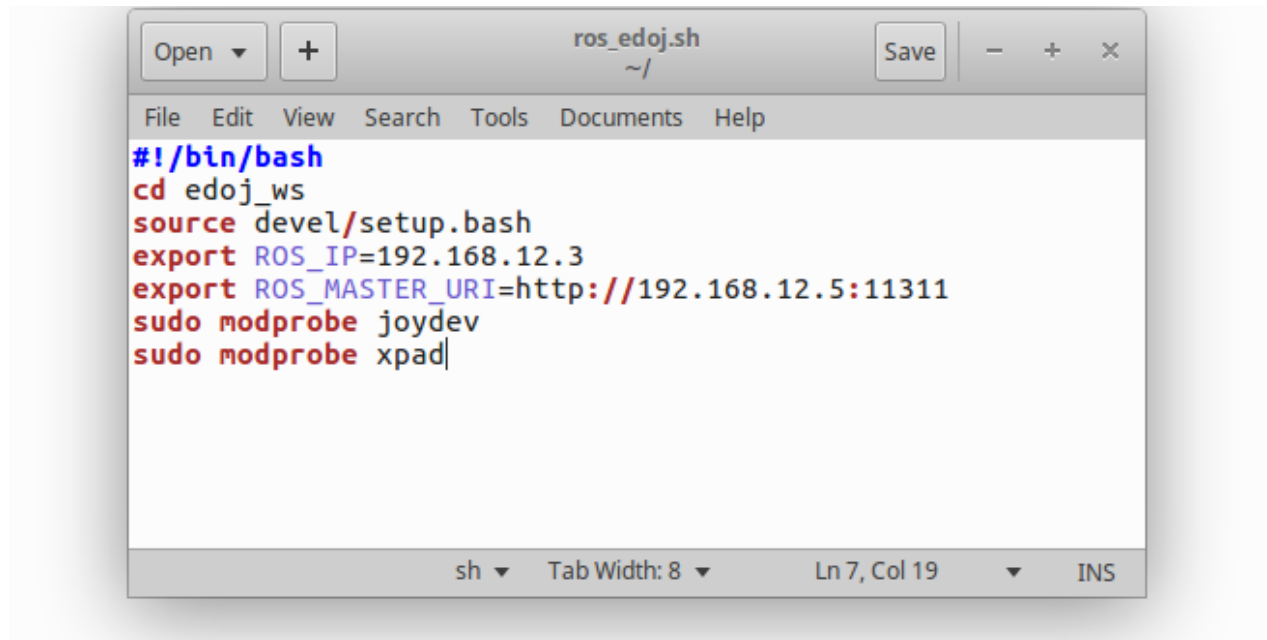


Abbildung 6.2.1. – Die Start-Datei ros_edoj.sh

Das Erstellen einer Start-Datei dient dazu, dass die Umgebung richtig und auf einmal gestartet wird. Es spart also den späteren Aufwand. Diese Datei soll sich am Besten im Home-Verzeichnis befinden. Sie verbindet uns außerdem mit dem Roboter. Die untenstehenden IP-Adressen müssen deswegen je nach Situation mit den richtigen ersetzt werden.

- **export ROS_IP** = die eigene IP-Adresse des Computers
- **export ROS_MASTER_URI** = die IP-Adresse des Roboters bzw. ROS-Masters
- **sudo modprobe joydev**
- **sudo modprobe xpad**

Die letzten beiden Zeilen aktivieren den Joystick beim Starten der Umgebung.

6.3. Verbinden mit dem e.DO-Roboter

Es stehen zwei Möglichkeiten zur Verfügung: WLAN-Verbindung und LAN-Verbindung.

6.3.1. WLAN-Verbindung

Man muss zunächst mit dem eigenen WLAN-Netz des Roboters verbinden und die Start-Datei ausführen. Stellen Sie sich sicher, dass die IP-Adressen mit den richtigen ersetzt worden sind.

6.3.2. LAN-Verbindung

Um den Roboter über LAN zu erreichen, muss es erstens mit dem folgenden Code eine SSH-Verbindung erstellt werden.

```
$ ssh edo@10.42.0.49
```

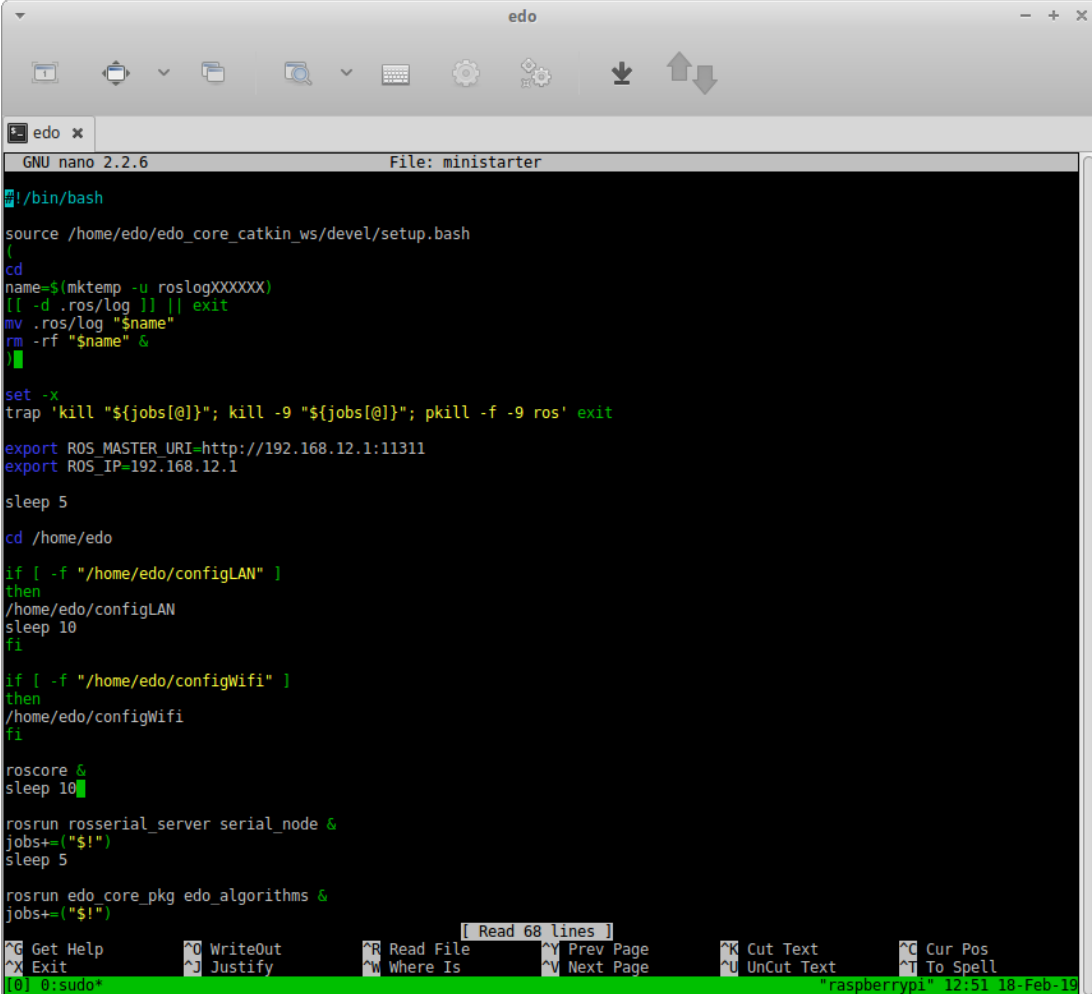
Das hier ist die Standard-IP-Adresse des Roboters. Damit wir unsere eigene Umgebung erstellen können, sollen wir am besten eine IP-Adresse auf unserem Router reservieren.

Das Passwort für den Roboter ist: **raspberrypi**. Mit folgendem Code wird der ministarter des Roboters geöffnet:

```
$ sudo nano ministarter
```

Die entsprechenden Zeilen, die die IP-Adresse beinhalten, müssen mit den richtigen ersetzt werden. Nach dem Speichern der neuen Einstellungen muss der Roboter neugestartet werden.

```
$ sudo reboot
```



The screenshot shows a terminal window titled 'edo' with a standard Linux desktop toolbar at the top. The terminal is running the GNU nano 2.2.6 text editor, editing a file named 'ministarter'. The content of the file is as follows:

```
#!/bin/bash
source /home/edo/edo_core_catkin_ws/devel/setup.bash
(
cd
name=$(mktemp -u roslogXXXXXX)
[[ -d .ros/log ]] || exit
mv .ros/log "$name"
rm -rf "$name" &
)

set -x
trap 'kill "${jobs[@]}"; kill -9 "${jobs[@]}"; pkill -f -9 ros' exit

export ROS_MASTER_URI=http://192.168.12.1:11311
export ROS_IP=192.168.12.1

sleep 5

cd /home/edo

if [ -f "/home/edo/configLAN" ]
then
/home/edo/configLAN
sleep 10
fi

if [ -f "/home/edo/configWifi" ]
then
/home/edo/configWifi
fi

roscore &
sleep 10

roslaunch roscpp_serial_server serial_node &
jobs+=("${!}")
sleep 5

roslaunch edo_core_pkg edo_algorithms &
jobs+=("${!}")
```

At the bottom of the terminal, a status bar shows various keyboard shortcuts (e.g., ^G Get Help, ^X Exit, ^O WriteOut, ^J Justify, ^R Read File, ^W Where Is, ^P Prev Page, ^N Next Page, ^K Cut Text, ^U UnCut Text, ^C Cur Pos, ^T To Spell) and the prompt '(0) 0:sudo*' on the left. On the right, it displays the system information: 'raspberrypi 12:51 18-Feb-19'.

Abbildung 6.3.1 – Die ministarter-Datei

6.4. Erstellen einer Launch-Datei

Eine einfache Launch-Datei spart uns die Zeit und den Aufwand, um die Nodes einzeln zu starten.

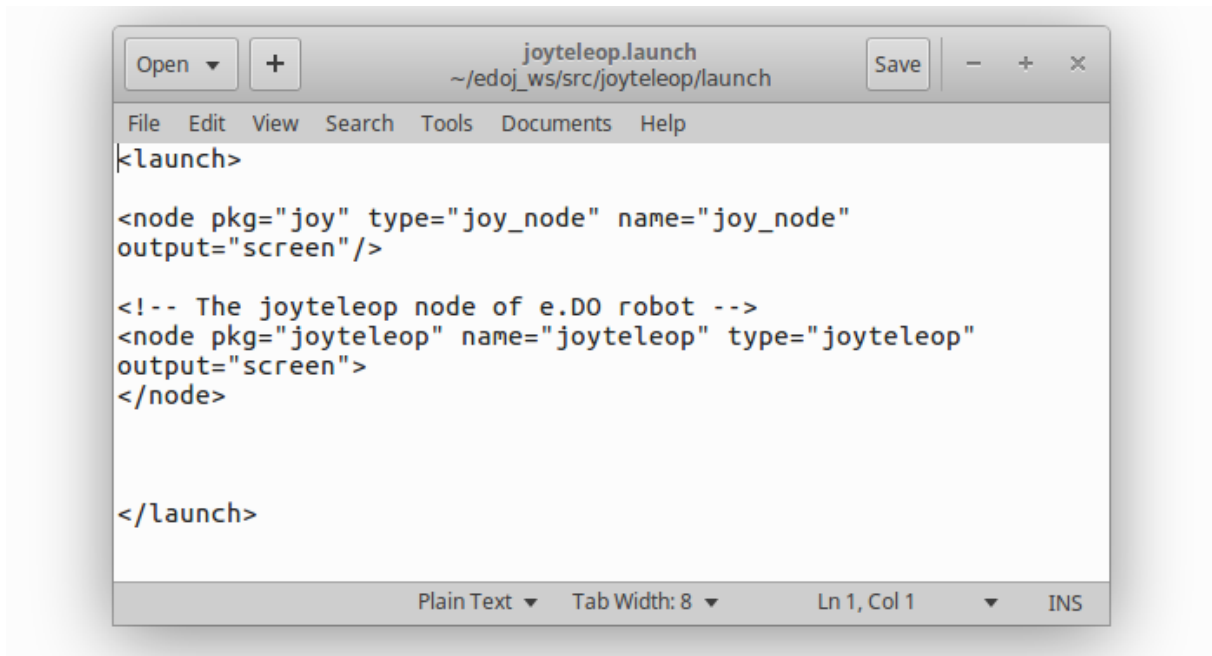


Abbildung 6.4. – Die Launch-Datei joyteleop.launch

6.5. Starten und Überprüfen des Nodes

Im folgenden Abschnitt wird unser Teleoperation-Node gestartet und seine Umgebung überprüft.

Mit folgendem Code werden die Start- und Launch-Dateien ausgeführt und deswegen sollen die beiden Nodes mit einer richtigen Verbindung mit dem e.DO-Roboter gestartet werden.

- **\$ source ros_edoj.sh**
- **\$ roslaunch joyteleop joyteleop.launch**

Die folgenden Folien zeigen der neue Stand von **rqt_topic** und **rqt_graph**.

Im richtigen Fall sollen die markierten Elemente erkennbar sein. Es ist wichtig, dass der Joy-Node gestartet und mit seinem Topic verbunden ist, da diese Informationen nachher auf unseren Teleoperation-Node publiziert werden.

Default - rqt					
File Plugins Running Perspectives Help					
Topic Monitor					
Topic	Type	Bandwidth	Hz	Value	
<input type="checkbox"/> /algo_jnt_ctrl	edo_core_msgs/JointControlArray			not monitored	
<input type="checkbox"/> /algo_movement_ack	edo_core_msgs/MovementFeedback			not monitored	
<input type="checkbox"/> /algorithm_state	std_msgs/Int8			not monitored	
<input type="checkbox"/> /bridge_init	edo_core_msgs/JointInit			not monitored	
<input type="checkbox"/> /bridge_jnt_reset	edo_core_msgs/JointReset			not monitored	
<input checked="" type="checkbox"/> /bridge_jog	edo_core_msgs/MovementCommand	12.32KB/s	70.61		
cartesian_linear_speed	float32			0.0	
delay	uint8			0	
▶ frame	edo_core_msgs/Frame				
move_command	uint8			74	
move_type	uint8			74	
ovr	uint8			100	
remote_tool	uint8			0	
▶ target	edo_core_msgs/Point				
▶ tool	edo_core_msgs/Frame				
▶ via	edo_core_msgs/Point				
<input type="checkbox"/> /bridge_move	edo_core_msgs/MovementCommand			not monitored	
<input type="checkbox"/> /cartesian_pose	edo_core_msgs/CartesianPose			not monitored	
<input type="checkbox"/> /diagnostics	diagnostic_msgs/DiagnosticArray			not monitored	
<input checked="" type="checkbox"/> /joy	sensor_msgs/Joy	444.08B/s	20.13		
axes	float32[]			(0.0, 0.0, 0.0, -0.0, -0.0, 0.0, 0.0, 0.0)	
buttons	int32[]			(0, 0, 0, 0, 0, 0, 0, 0)	
▶ header	std_msgs/Header				
<input type="checkbox"/> /machine_algo_jnt_state	edo_core_msgs/JointStateArray			not monitored	
<input type="checkbox"/> /machine_bridge_jnt_state	edo_core_msgs/JointStateArray			not monitored	
<input type="checkbox"/> /machine_init	edo_core_msgs/JointInit			not monitored	
<input type="checkbox"/> /machine_jnt_calib	edo_core_msgs/JointCalibration			not monitored	
<input type="checkbox"/> /machine_jnt_config	edo_core_msgs/JointConfigurationArray			not monitored	
<input type="checkbox"/> /machine_jnt_reset	edo_core_msgs/JointReset			not monitored	
<input type="checkbox"/> /machine_jnt_version	std_msgs/UInt8			not monitored	
<input type="checkbox"/> /machine_jog	edo_core_msgs/MovementCommand			not monitored	
<input type="checkbox"/> /machine_move	edo_core_msgs/MovementCommand			not monitored	
<input type="checkbox"/> /machine_movement_ack	edo_core_msgs/MovementFeedback			not monitored	
<input type="checkbox"/> /machine_state	edo_core_msgs/MachineState			not monitored	
<input type="checkbox"/> /rosout	rosgraph_msgs/Log			not monitored	
<input type="checkbox"/> /rosout_agg	rosgraph_msgs/Log			not monitored	
<input type="checkbox"/> /usb_jnt_state	edo_core_msgs/JointStateArray			not monitored	
<input type="checkbox"/> /usb_jnt_version	edo_core_msgs/JointFwVersion			not monitored	

Abbildung 6.5.1. – Das Feedback des **rqt_topics**, nachdem die Dateien richtig ausgeführt wurden

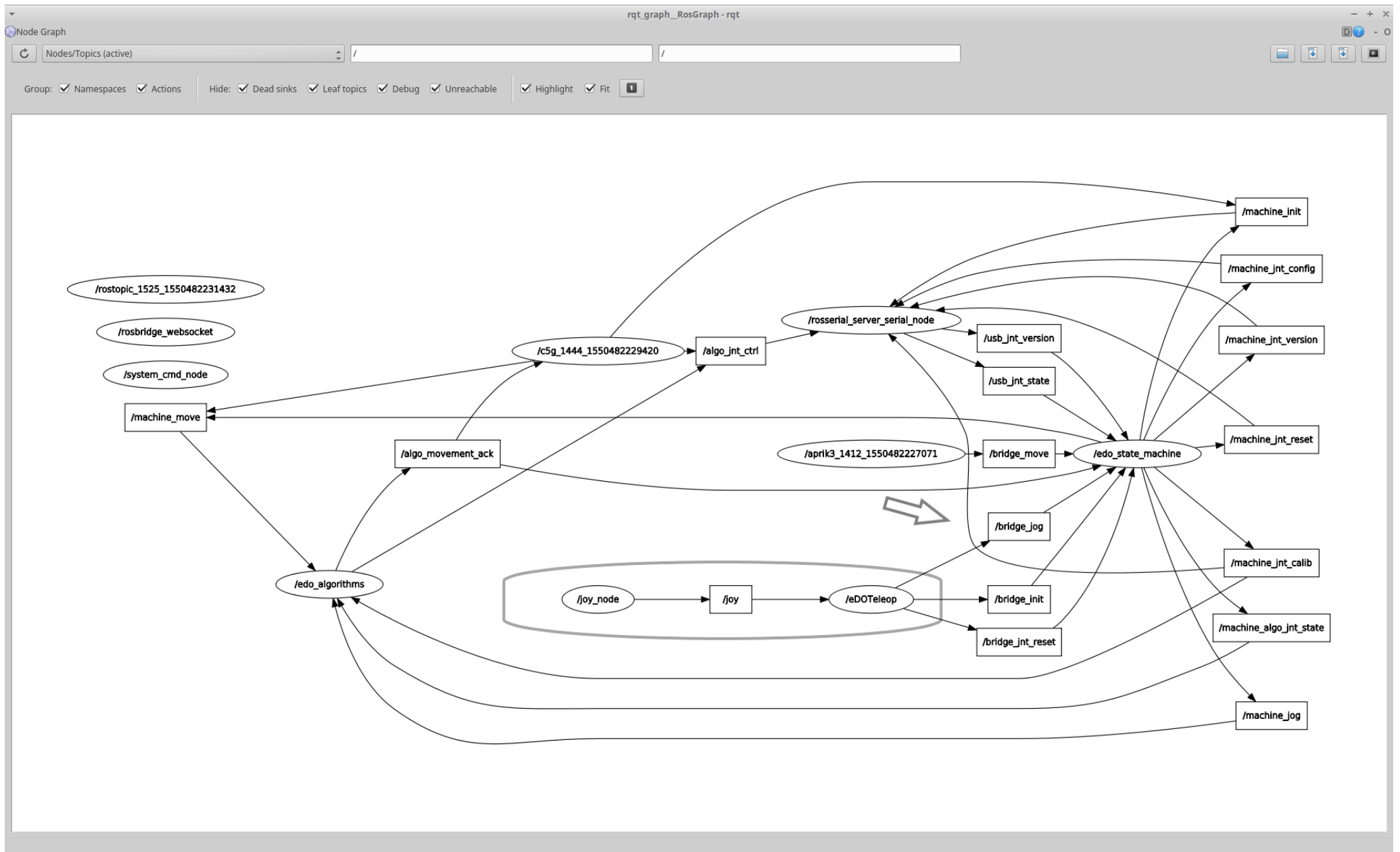


Abbildung 6.5.2. – Das Feedback des **rqt_graphs**, nachdem die Dateien richtig ausgeführt wurden