1. Introduction to ROS

The ROS is an open source operating system consisting of tools, libraries, device drivers and utility functions. Like other operating systems, ROS contains many files that describe its functionality: Packages, manifests, messages, services, and etc.

The ROS master can be interpreted as the namespace of the nodes and services. A real robot or the ROS connection with a real robot starts the ROS Master. On the other hand, it must be started with the code **`roscore`** if no robot is present.

2. ROS Computational Graph Level

The ROS creates a network with all its processes. This communication takes place via the ROS master.

A ROS node is the process that performs the calculations. Often many nodes are used together to control different and complex functions. Each node can reach this network, subscribe to it or publish something there.

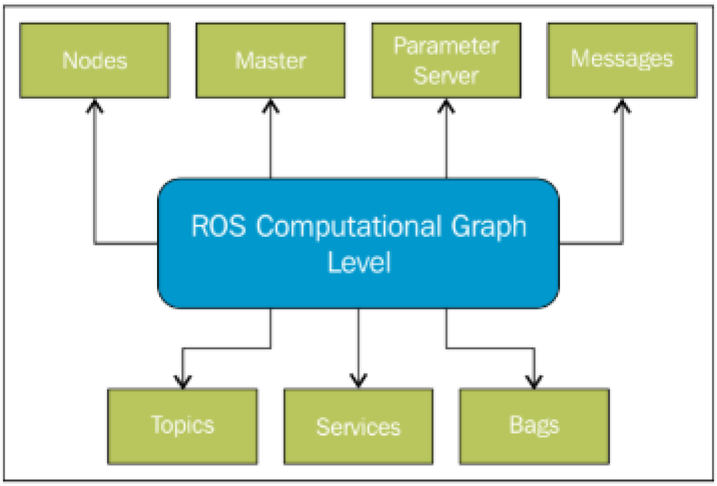


Figure 2.1. – ROS Computational Graph Level

3. Communication of nodes of e.DO robot

Each element shown in Figure 2.1. has different functions in this network. The ROS also provides various tools for debugging this network.

`rqt_graph` is a tool that graphically displays the active topics and nodes. There the nodes are displayed as squares and the topics as ellipses.

`rqt_topic` is another ROS tool that creates a list of active topics. The following images show these tools.

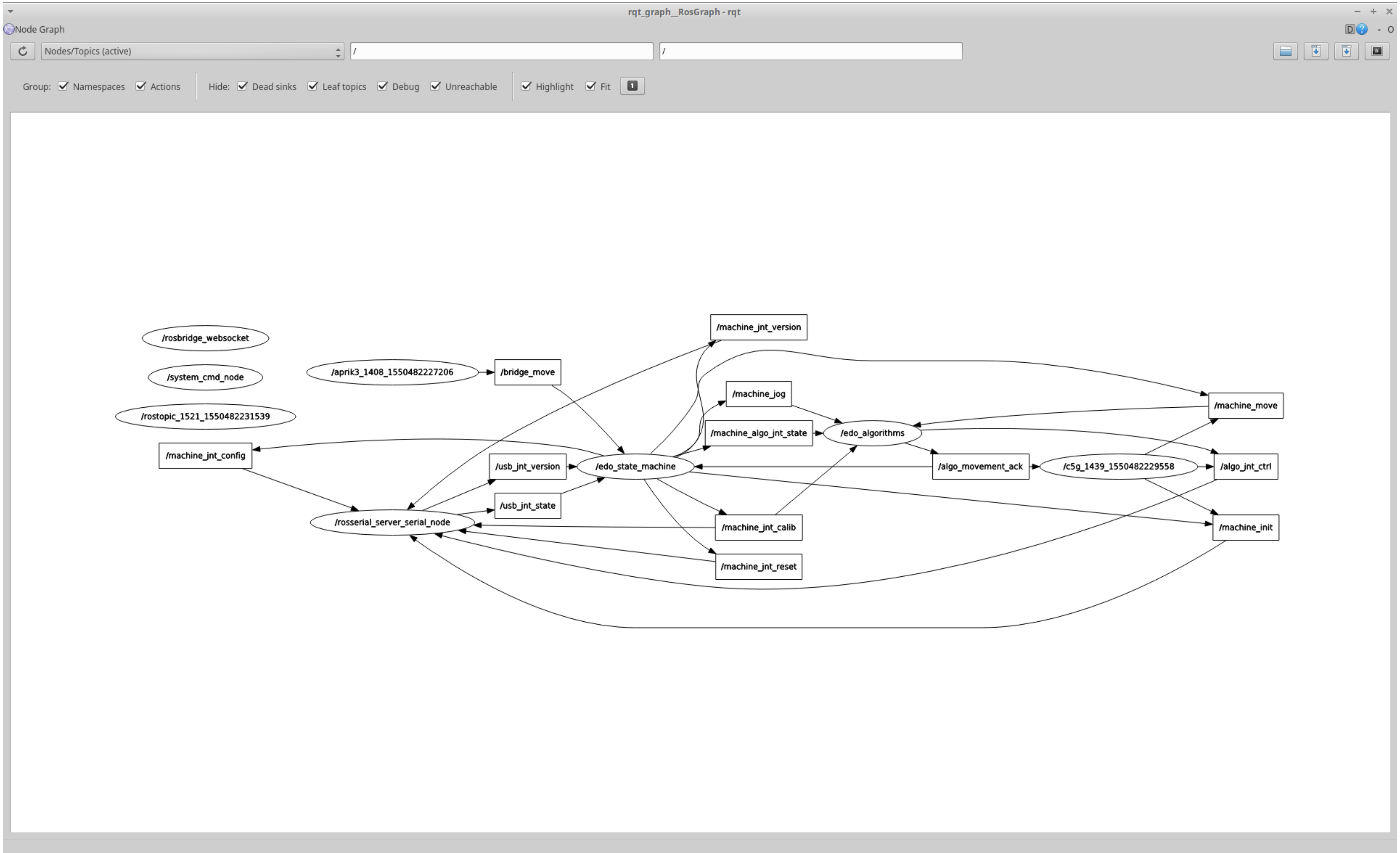


Figure 2.2. – Standard Situation of nodes of the e.DO Robot

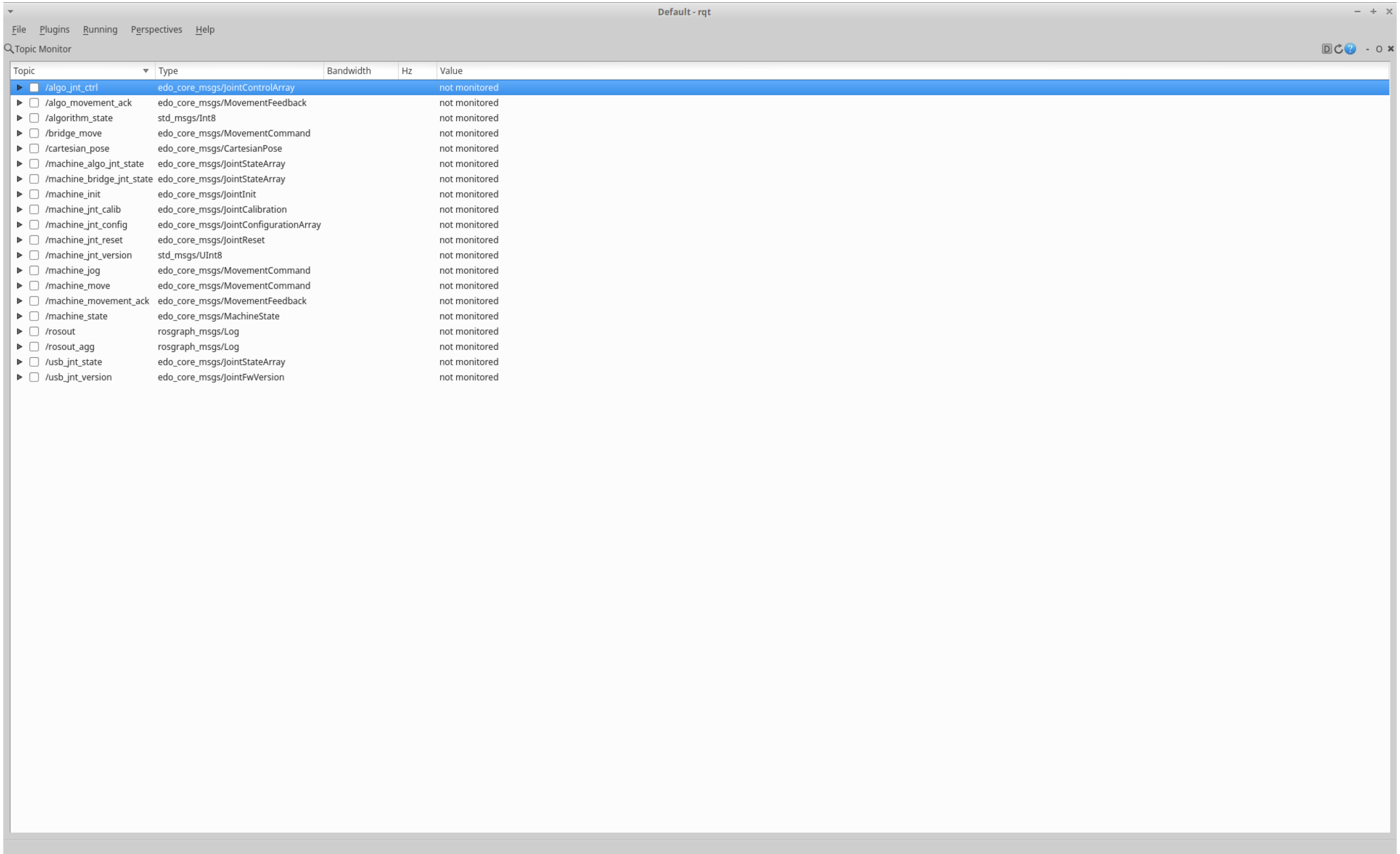


Figure 2.3. – Standard Situation of the Nodes of the e.DO Robot

Two of the topics presented here are particularly important. They forward the motion commands to the robot. **/bridge_jog** and **/bridge_move.**

/bridge_jog is currently not visible because it has not yet been started by a node. The first command it will activate will be sent to this topic.

/joint_states and **/machine_state** are other topics that inform us about the positions of the joints and the situation of the robot.

3. How to use sensor information with e.DO?

As mentioned in the previous chapters, the ROS supports many external devices and also provides their drivers. A joystick is also one of these devices that can be used with a robot controlled by ROS.

In the following sections of this report, a teleoperation node for Microsoft's x-Box controller is written. That this device is a joystick supported by Linux, its driver can be installed with the following code:

$ sudo apt-get install ros-kinetic-joy

With the following code the Joy-Node can be started:

$ rosrun joy joy_node

And with the following code you can listen to the Joy-Topic:

$ 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$
```

Figure 3.1. – **rosnode info joy**, before the node is started

```
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$
```

Figure 3.2 - **rostopic info joy** after starting the node. **sensor_msgs/Joy** is the message type we need.

```
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]
---
```

Figure 3.3. and 3.4. -**`rostopic echo joy`** and different reactions of keys and axes

`rostopic info` informs us not only about current publishers and subscribers, but also about the message types of the topic. Figure 3.2 shows how we determined the message type. Another code **`rosmsg show`** shows what information is published with this message. It is noticeable that we have exactly the same information when listening to the topics.

Figure 3.3 and 3.4 show different reactions of the axes and keys of the joystick. In both cases the reactions are different, but the content of the message is always the same.

Figure 3.6 shows that the joystick node has been started but is not connected to anything. In this way we can retrieve the information from the joystick node.

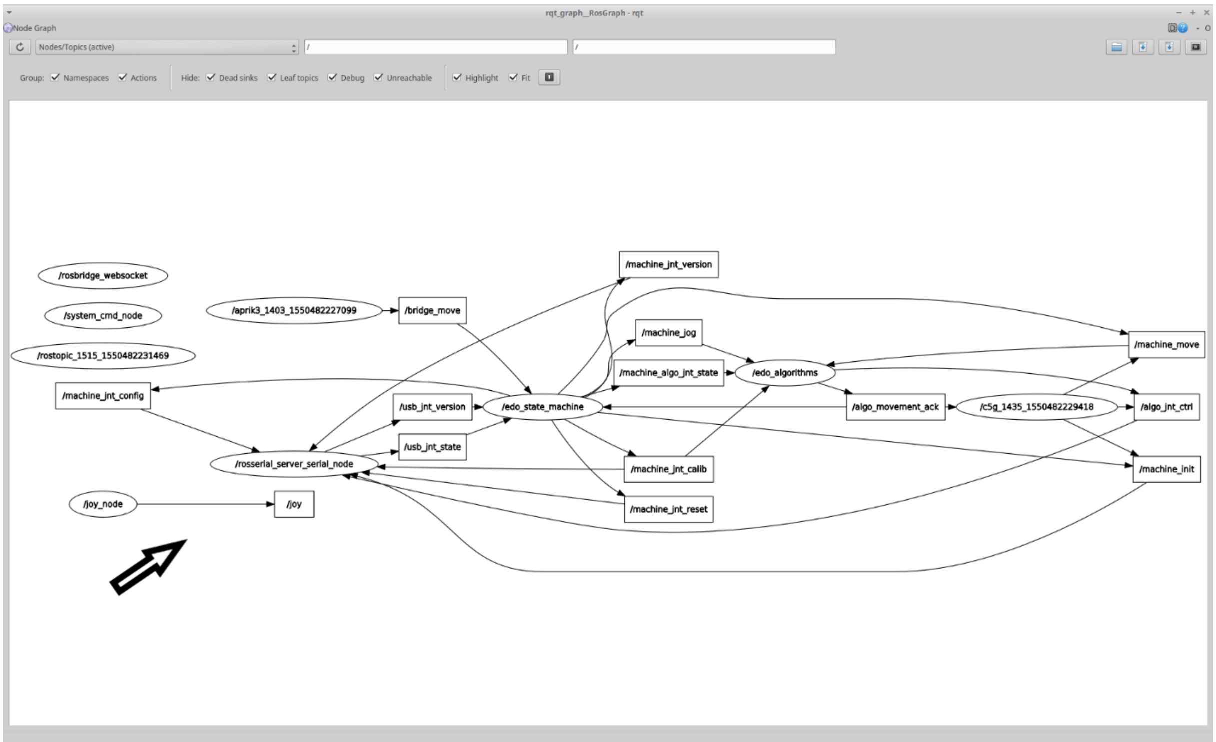


Figure 3.6 - The Network of Nodes after Starting the Joystick Node

4. Implementation of the Data from Joystick

Comau's tablet application is a very good opportunity to find out how to communicate properly with the robot. It can be listened to the corresponding topics determined in the previous sections with the corresponding codes or tools.

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

Figure 4.1 - The type of message used to send motion commands to the robot

The same procedure can be applied to any topic. In the end, you always have the name of the message.

The information shown in Figure 4.2 above must therefore be published on the corresponding topic so that the robot can move.

Our goal is to find out how the tablet sends this information and with what content. **/rostopic echo bridge_jog** again allows us to listen to the topic.

After we have 'subscribed' to the topic with the code, we send a motion command for the Joint 1 from the tablet. Figure 4.3 shows the content we have published.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

ros_admin@ros-pc:~/edoj_ws$ rosmsg 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
```

Figure 4.2 - **`rosmsg show edo_core_msgs/MovementCommand`** specifies what information is passed to the robot to move it.

```
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
---
```

Figure 4.3 - Information published by Tablet on the topic

It has now become clear to us what information should be sent for a movement of a joint. In pure theory, you can also send a similar content from a computer to move it.

`/bridge_init` and **`/bridge_jnt_reset`** should be the other important topics we will use. We inquire from **`rqt_topic`**.

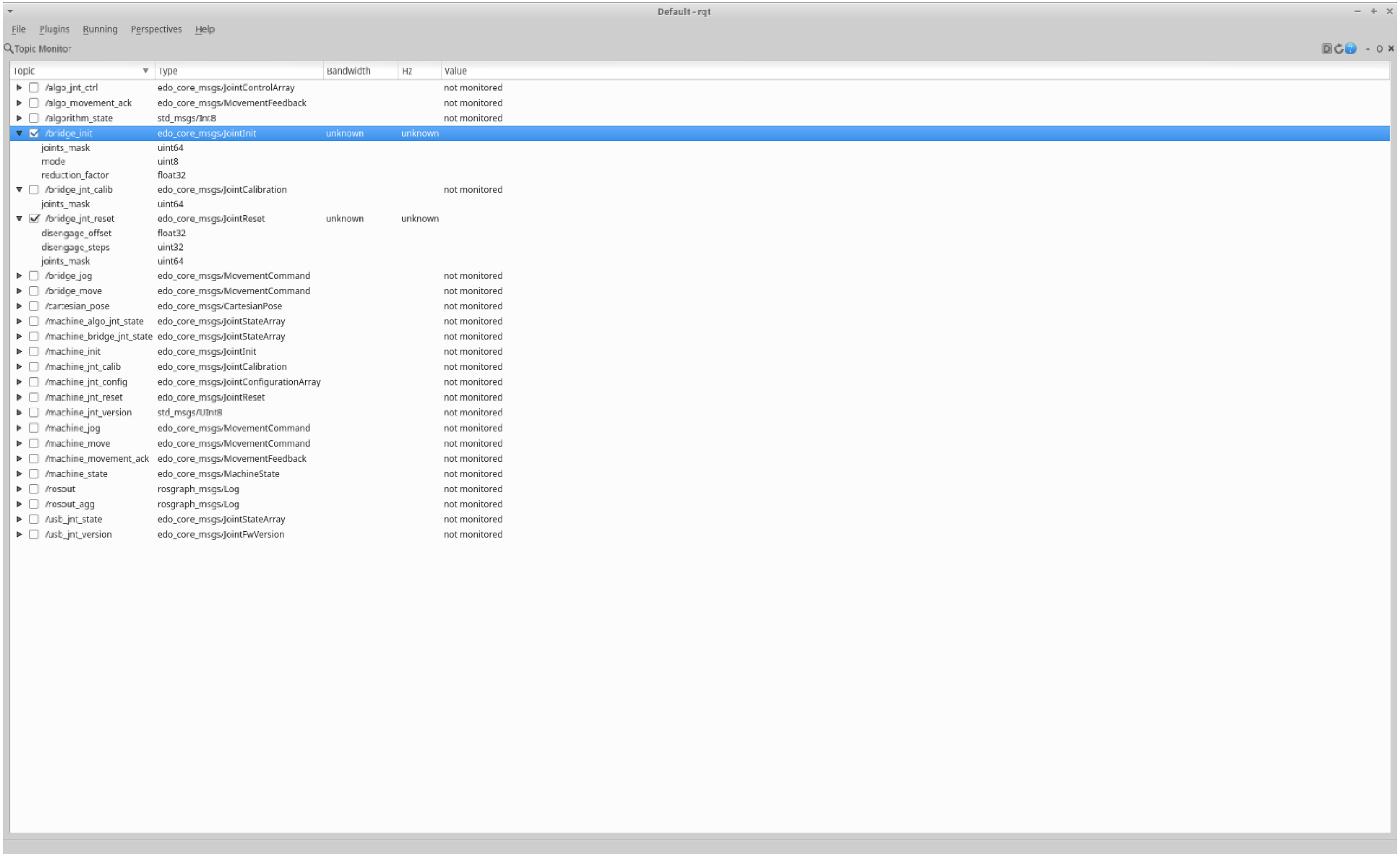
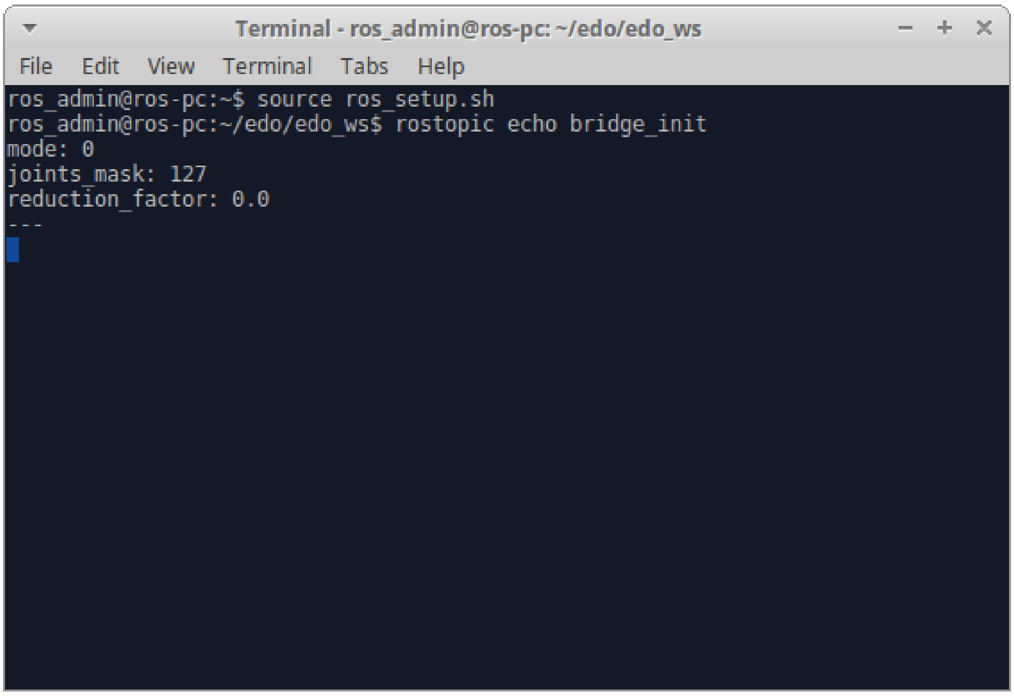


Figure 4.4 - The feedback from **rqt_topic**

Figure 4.4 gives us information about the types and contents of messages published on this topic.

With the following lines, **`$ rostopic echo /bridge_init`** and **`$ rostopic echo /bridge_jnt_reset`** we listen to topics. We assumed that these two topics initialize the robot and trigger its brakes, the robot should first be restarted.

The following figures inform us about the output.



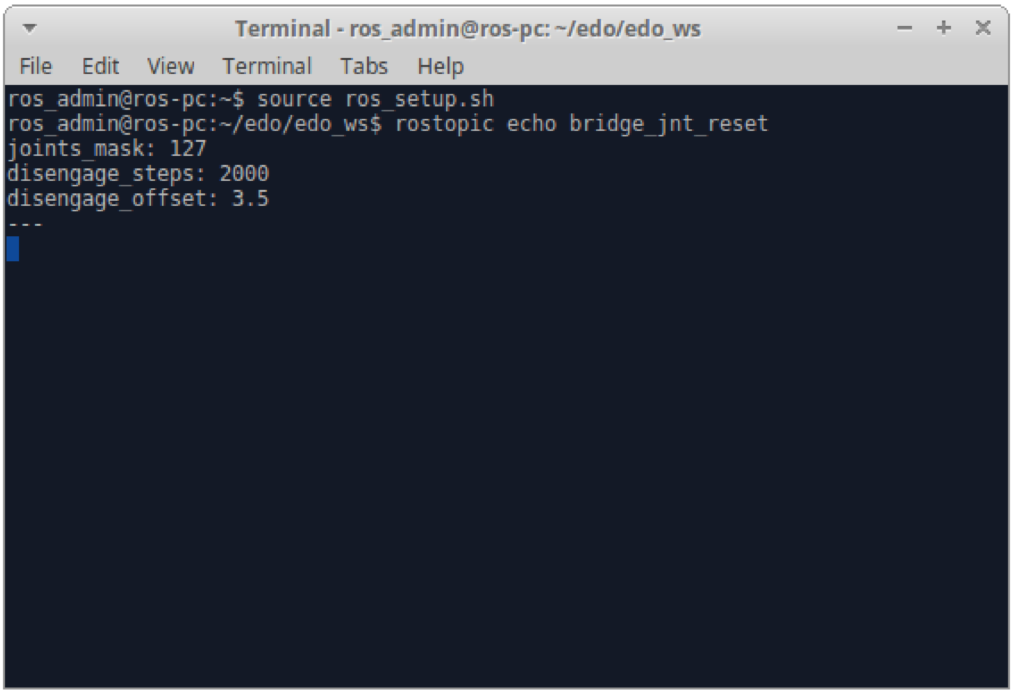


Figure 4.5. and 4.6. - Information published by the tablet on different topics

After restarting the robot, it will be initialized again by the tablet and receive the information shown above.

In this section we now know how to start the robot, initialize it, release its brakes and send motion commands.

5. How to write a teleoperation node for e.DO and x-Box Controller?

In the following section we will look at the implementation of a separate node whose task it is to control the robot by sending similar commands via the ROS.

```
#include <ros/ros.h>
#include "edo_core_msgs/MovementCommand.h"  // for move commands
#include "edo_core_msgs/JointReset.h" // for joint reset commands
#include "edo_core_msgs/JointInit.h" // for the initialization of robot
#include <sensor_msgs/Joy.h> // includes the joystick messages, so that we can listen the joy topic
#include <iostream>
#include <queue>
#include <ncurses.h>
#include <stdlib.h>
#include <stdio.h>
#include <thread>
#include <chrono>
#include <fstream>
#include <queue>
#include <string>
#include <iomanip>
```

In this part the **`/sensor_msgs`** for the joystick and the **`/edo_core_msgs`** for the communication with the robot are added. The rest is for various activities of the C++ code.

```
//Global variables for axes and buttons
double intturnA = 0;
double intturnB = 0;
double intturnC = 0;
double intturnD = 0;
double intturnE = 0;
double intturnF = 0;
double intturnG = 0;

//Global variables for velocity;
double vel = 1;
double velocitymax = 1.1;
double velocityup = 0;
double velocitydown = 0;
double velocitymin = 0;
```

Various global variables have been defined here so that they can later be used anywhere in the code.

```
//Shortcut for the robot - Fixed Values for a "jog" move
edo_core_msgs::MovementCommand createJog(){
    edo_core_msgs::MovementCommand msg;
    msg.move_command = 74;
    msg.move_type = 74;
    msg.ovr = 100;
    msg.target.data_type = 74;
    msg.target.joints_mask = 127; // Without gripper it should be 63
    msg.target.joints_data.resize(10, 0.0);
    return msg;
}

```

The previously determined content of the motion command is defined here as a function. Experimenting with the robot has shown that the joint-mask for a model without gripper should be 63.

```
eDOTeleop::eDOTeleop()
{
  //Joystick subscriber
  joy_sub_ = nh_.subscribe<sensor_msgs::Joy>("joy", 10, &eDOTeleop::joyCallback, this);
  //ROS Publisher to "/bridge_jog" topic
  jog_ctrl_pub = nh_.advertise<edo_core_msgs::MovementCommand>("bridge_jog", 10);
}

void eDOTeleop::joyCallback(const sensor_msgs::Joy::ConstPtr& joy)
//Callback function for the joystick
{
    int i;

    for (unsigned i = 0; i < joy->axes.size(); ++i) {
    //ROS_INFO("Axis %d is now at position %f", i, joy->axes[i]);
    }

```

The definition of the class and the **`joyCallback`** function are here to get the sensor messages from the joystick. Subscribers and publishers are also defined here for later use.

The for loop in **`void`** allows you to redefine the previously mentioned variables with the joystick information. The axes and keys of the joystick are used to move the joints, while only the keys are used to enter the speed.

The figure below shows the **`velocity`** function which allows you to determine the input range of the velocity. This function reacts to buttons A and B on the joystick.

```cpp
double velocity(double veloc)
{
  if (velocityup==1 && veloc<velocitymax)
  {
    //ROS_INFO("Vel %f", vel);
    //ROS_INFO("velocity up %f", velocityup);
    vel=veloc+0.05;
  }

  if (velocitydown==1 && veloc>velocitymin)
  {
    //ROS_INFO("Vel %f", vel);
    //ROS_INFO("velocity down %f", velocitydown);
    vel=veloc-0.05;
  }

  return veloc;
}
```

```cpp
//Definition of the correctvalues function - so that the robot does not react
//to all movements of the joystick.
double correctvalues(double value){

  double calibrator=0.3;
  double retval=0;

  if (value > 0 && value < 0.3)
  {
    retval = 0;
  }

  if (value > -0.3 && value < 0)
  {
    retval=0;
  }

  if (value == 0)
  {
    retval=0;
  }

  if (value == -0.0)
  {
    retval=0;
  }

 if (value>calibrator && value<1.0)
    {
      retval=-1;
    }

  if (value>-1.0 && value<-0.5)
  {
      retval=1;
  }

   if (value ==-1)
  {
    retval=1;
  }

  if (value ==1)
  {
     retval=-1;
  }
    return retval;
}
```

The figure above shows the **correctvalues** function. This prevents the sensitivity of the joystick. That the commands are sent to the robot from a value of 0.3 is defined here.

```cpp
int main(int argc, char** argv)
{
//Initialize "eDOTeleop" ROS Node
//ROS_INFO("Start ROS");

  ros::init(argc, argv, "eDOTeleop");
  eDOTeleop eDOTeleop;

  ros::NodeHandle nh_;
  char proceed = '\n';
  edo_core_msgs::MovementCommand msg = createJog();

  ros::Rate loop_rate(100); // Loop frequency
  ros::Publisher jog_ctrl_pub = nh_.advertise<edo_core_msgs::MovementCommand>("bridge_jog", 10);

  //ROS_INFO("Start while");

  //Robot startup messages
  ros::Publisher reset_pub = nh_.advertise<edo_core_msgs::JointReset>("/bridge_jnt_reset",10); //ROS Publisher to publish joint reset command
  ros::Publisher init_pub = nh_.advertise<edo_core_msgs::JointInit>("/bridge_init",10);  //ROS Publisher to publish robot init command
  std::chrono::milliseconds timespan(10000); //Time for initialization
  edo_core_msgs::JointReset reset_msg; //Definition of the messages
  edo_core_msgs::JointInit init_msg;
```

Then, we initialize our ROS node and create the eDOTeleop node. Afterwards the publishers for the initialization of the robot are defined.

```cpp
  std::cout << "\033[2J\033[1;1H"; // Clean the screen;
  std::this_thread::sleep_for(timespan/4); //For the joy_node;
  while (proceed!='y'){
    std::cout << "Enter 'y' to initialize 6-Axis e.DO-Robot with gripper.\n"
              << "The process takes 10 secs. == ";
    std::cin >> proceed;
  }

  proceed = '\n';
  init_msg.mode = 0; //Fixed values to start up the robot
  init_msg.joints_mask = 127;
  init_msg.reduction_factor = 0.0;

  while(init_pub.getNumSubscribers() == 0){
    loop_rate.sleep();
  }

  init_pub.publish(init_msg);
  ros::spinOnce();
  loop_rate.sleep();

  std::cout << "\033[2J\033[1;1H"; // Clean the screen;
  std::this_thread::sleep_for(timespan); // While e.DO starts up

  while(proceed != 'y'){
    std::cout << "Automatic motion! \n"
              << "The robot will move, type 'y' to disengage breaks == ";
    std::cin >> proceed;
  }

  proceed = 'n';
  reset_msg.joints_mask = 127; //Fixed values to disengage the breaks
  reset_msg.disengage_steps = 2000;
  reset_msg.disengage_offset = 3.5;

  while(reset_pub.getNumSubscribers() == 0){
    loop_rate.sleep();
  }
  reset_pub.publish(reset_msg);
  ros::spinOnce();
  loop_rate.sleep();
  //End of robot startup
```

The previously defined publishers are used here to initialize the robot and release its brakes.

After the robot is started correctly, we slowly reach our goal.

```
//Visualisation of joystick interface
std::cout
  <<"____________Welcome on board!____________\n"
  <<"|---------------------------------------|*\n"
  <<"|    ___LB___               ___RB___    |*\n"
  <<"|   /   __   \\_____________/        \\   |*\n"
  <<"|  |  //JL\\    < >    < >      (Y)   |  |*\n"
  <<"|  |  \\__//                 (X)   (B)|  |*\n"
  <<"|  |        _           __     (A)   |  |*\n"
  <<"|  |      _|^|_       //JR\\          |  |*\n"
  <<"|  |     |<   >|       \\__//         |  |*\n"
  <<"|  |       |v|                       |  |*\n"
  <<"|  |        _______/--\\_______       |  |*\n"
  <<"|  |       |                 |       |  |*\n"
  <<"|   \\     /                   \\      /  |*\n"
  <<"|    \\___/                     \\____/   |*\n"
  <<"|---------------------------------------|*\n"
  <<"You are in the joystick mode of the e.DO robot.\n"
  <<"|---------------------------------------|*\n"
  <<"For Joint 1 -> Left joystick up and down  \n"
  <<"For Joint 2 -> Left joystick left and right\n"
  <<"For Joint 3 -> Right joystick up and down\n"
  <<"For Joint 4 -> Right joystick left and right\n"
  <<"For Joint 5 -> ^ for up and v for down\n"
  <<"For Joint 6 -> < for left and > for right\n"
  <<"For gripper -> RB to open and **RB and RT** to close\n"
  <<"----------------------------------------\n"
  <<"Button A to speed up and Button B to speed down\n"
  << std::flush;
```

The user interface of the node is described here in more detail and a user manual is created.

Then comes the most important part of the code: The part that sends this information to the robot.

First, the values of each individual message are defined. A function consisting of predefined functions is then created for each joint. The code **`msg.target.joints_data`** informs the joints how much to move.

It is important that the joints are saved as a list in the system. Joint 1 is therefore on the 'zero' place. **`usleep()`** determines the frequency of the messages.

```
    while(true){
    //End of the visualisation of joystick interface
    //Fixed values for a "jog" move
          msg.move_command=74;
          msg.move_type=74;
          msg.ovr=100;
          msg.target.data_type = 74;
          msg.target.joints_mask = 127;


    //With correctvalues() - Publish the messages to the robot
          msg.target.joints_data[0] = (0,(velocity(vel))*correctvalues(intturnA));
          jog_ctrl_pub.publish(msg);

          msg.target.joints_data[1] = (0,(velocity(vel))*correctvalues(intturnB));
          jog_ctrl_pub.publish(msg);

          msg.target.joints_data[2] = (0,(velocity(vel))*correctvalues(intturnC));
          jog_ctrl_pub.publish(msg);

          msg.target.joints_data[3] = (0,(velocity(vel))*correctvalues(intturnD));
          jog_ctrl_pub.publish(msg);

          msg.target.joints_data[4] = (0,(velocity(vel))*correctvalues(intturnE));
          jog_ctrl_pub.publish(msg);

          msg.target.joints_data[5] = (0,(velocity(vel))*correctvalues(intturnF));
          jog_ctrl_pub.publish(msg);

    //For gripper control
          msg.target.joints_data[6] = (0,(velocity(vel))*intturnG);
          jog_ctrl_pub.publish(msg);

          usleep(100000);
          ros::spinOnce();
    }
  }
```

6. Creating an environment for e.DO

This code creates and compiles a new folder in the Catkin workspace.

```
$ cd ~/catkin_ws/src

$ catkin_create_pkg joyteleop roscpp joy sensor_msgs
message_generation edo_core_msgs std_msgs

$ cd ~/catkin_ws/

$ catkin_make
```

The code written and explained in the last section must now be copied to the **`joyteleop.cpp`** file. This file should be created in the directory **`joyteleop/src`**.

Add the following lines to the Package.xml file in the Joyteleop folder:

```
<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>
```

Add the following lines to the CMakeLists.txt file in the same folder:

```
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 Other Dependencies of the Package

- NCurses is used for asynchronous control of the joints and can be downloaded and installed online.

- eDO_core_msgs folder contains the messages for the communication of the robot. You can find it on the GitHub page of the Comau.

- • C++ 11

These dependencies of the package must be installed before the package is compiled. Otherwise it won't compile anyway. ☺

6.2. Creating a Start File

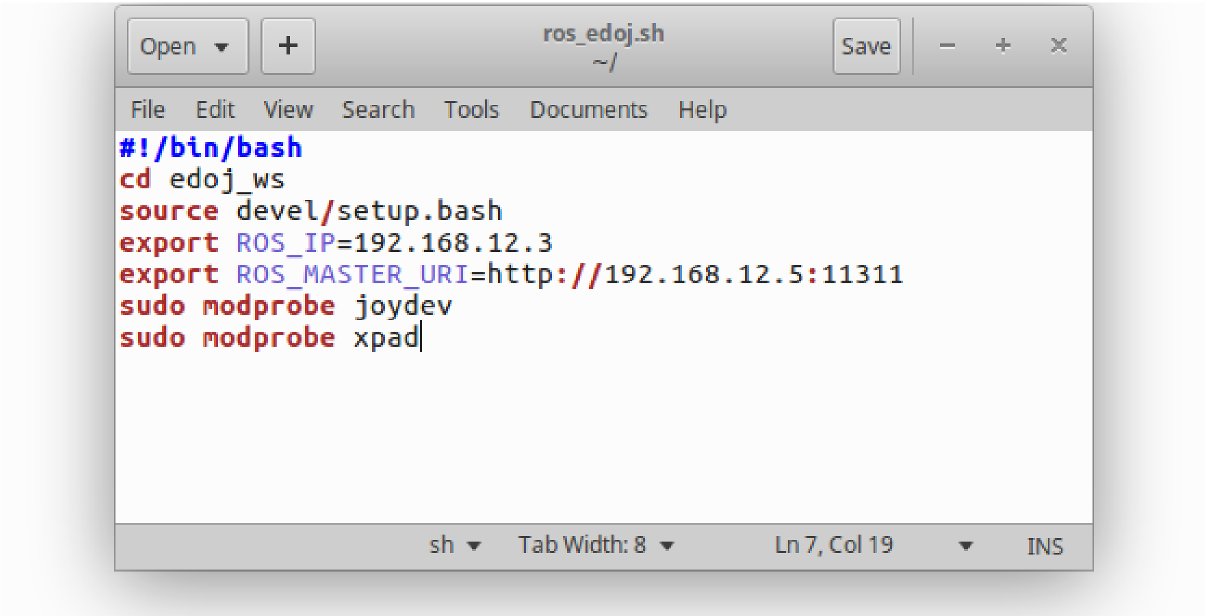


Figure 6.2.1 - The start file **ros_edoj.sh**

The creation of a start file serves to ensure that the environment is started correctly and all at once. So it saves the effort later on. This file is best located in the home directory. It also connects us to the robot. The IP addresses below must therefore be replaced with the correct ones depending on the situation.

- **export ROS_IP** = the own IP address of the computer
- **export ROS_MASTER_URI** = the IP address of the robot or the ROS master.
- **sudo modprobe joydev**
- **sudo modprobe xpad**

The last two lines activate the joystick when starting the environment.

6.3. Connecting with the robot

Two options are available: WLAN connection and LAN connection.

6.3.1. Wi-Fi Connection

You must first connect to the robot's own WLAN network and execute the start file. Make sure that the IP addresses have been replaced with the correct ones.

6.3.2. LAN Connection

In order to reach the robot via LAN, it must first be established an SSH connection with the following code.

$ ssh edo@10.42.0.49

This is the robot's default IP address. So that we can create our own environment, it is best to reserve an IP address on our router.

The password for the robot is: raspberry. With the following code the **ministarter** of the robot is opened:

$ sudo nano ministarter

The corresponding lines containing the IP address must be replaced with the correct ones. After saving the new settings, the robot must be restarted.

$ sudo reboot

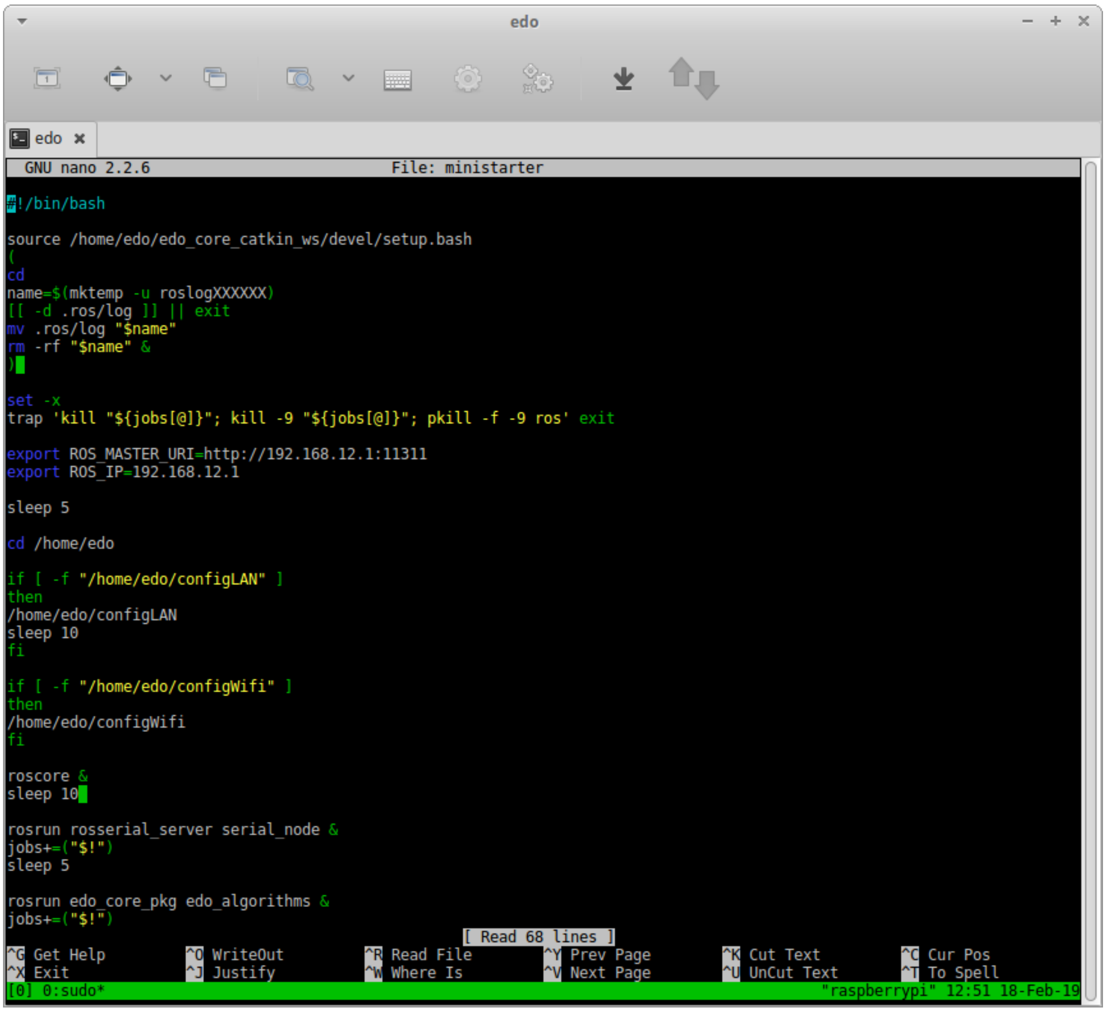


Figure 6.3.1 - The ministarter file

6.4. Creating a Launch File

A simple launch file saves us the time and effort to start the nodes individually.

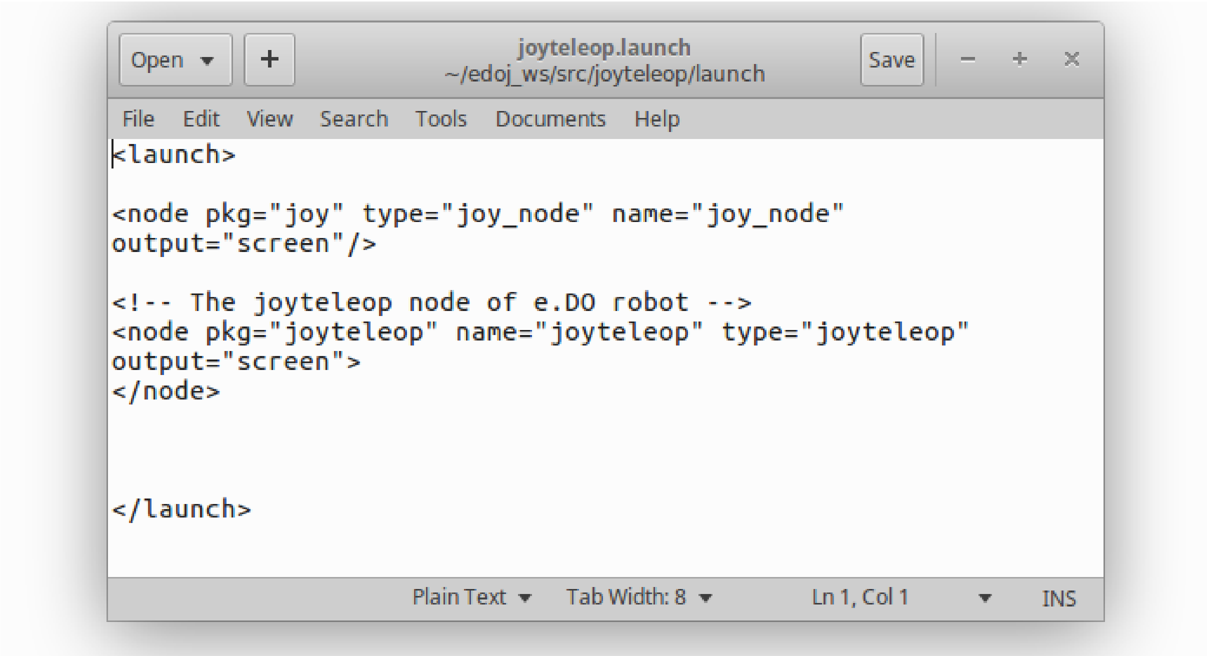


Figure 6.4 - The launch file joyteleop.launch

6.5. Starting and Checking the node

The following section starts our teleoperation node and checks its environment.

The following code is used to execute the start and launch files and therefore the two nodes should be started with a proper connection to the e.DO robot.

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

The following slides show the new state of **`rqt_topic`** and **`rqt_graph`.**

In the right case the marked elements should be recognizable. It is important that the Joy-Node is started and connected to its topic, because this information will be published later on our Teleoperation-Node.

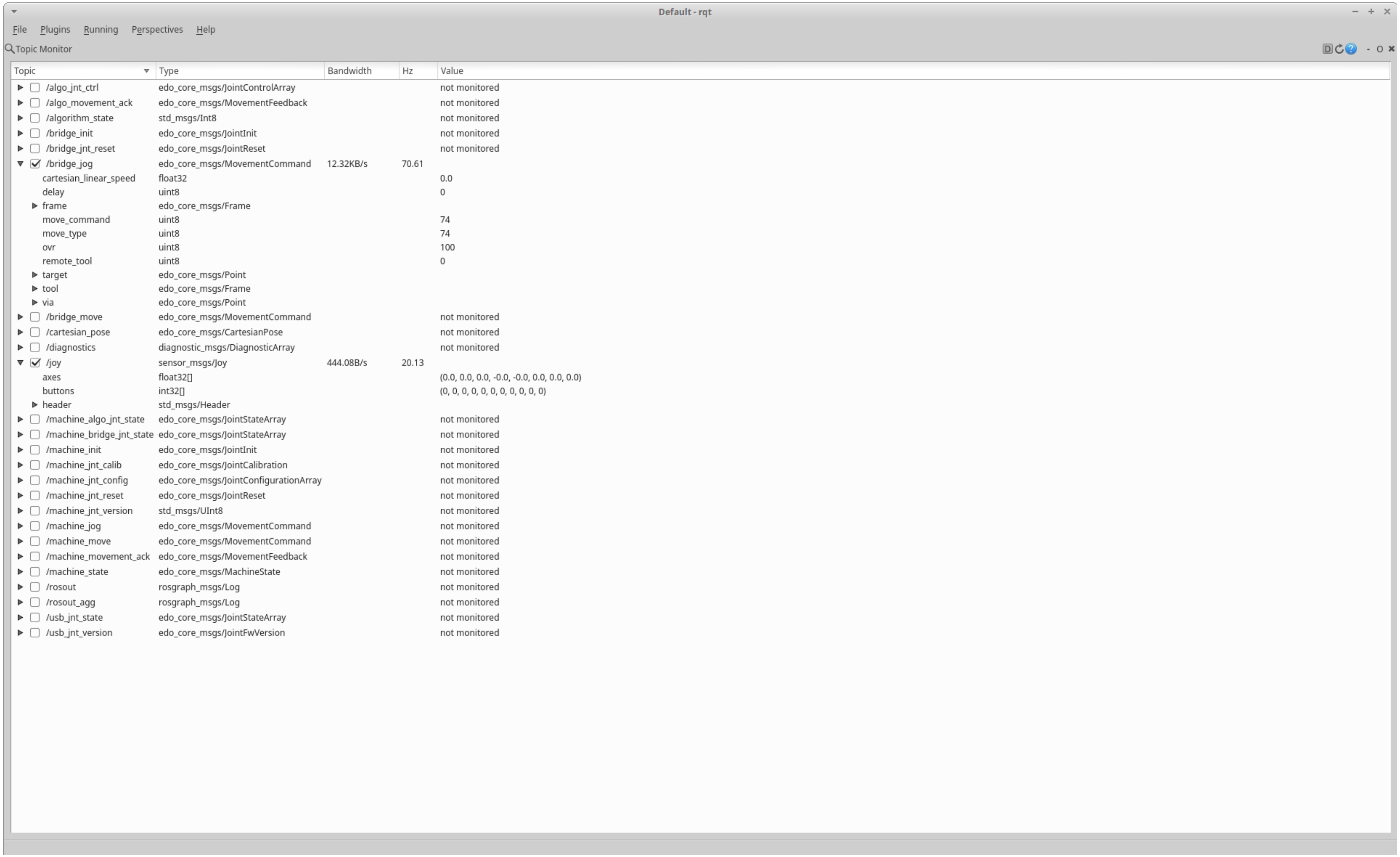


Figure 6.5.1 - The feedback of **`rqt_topics`** after the files have been executed correctly

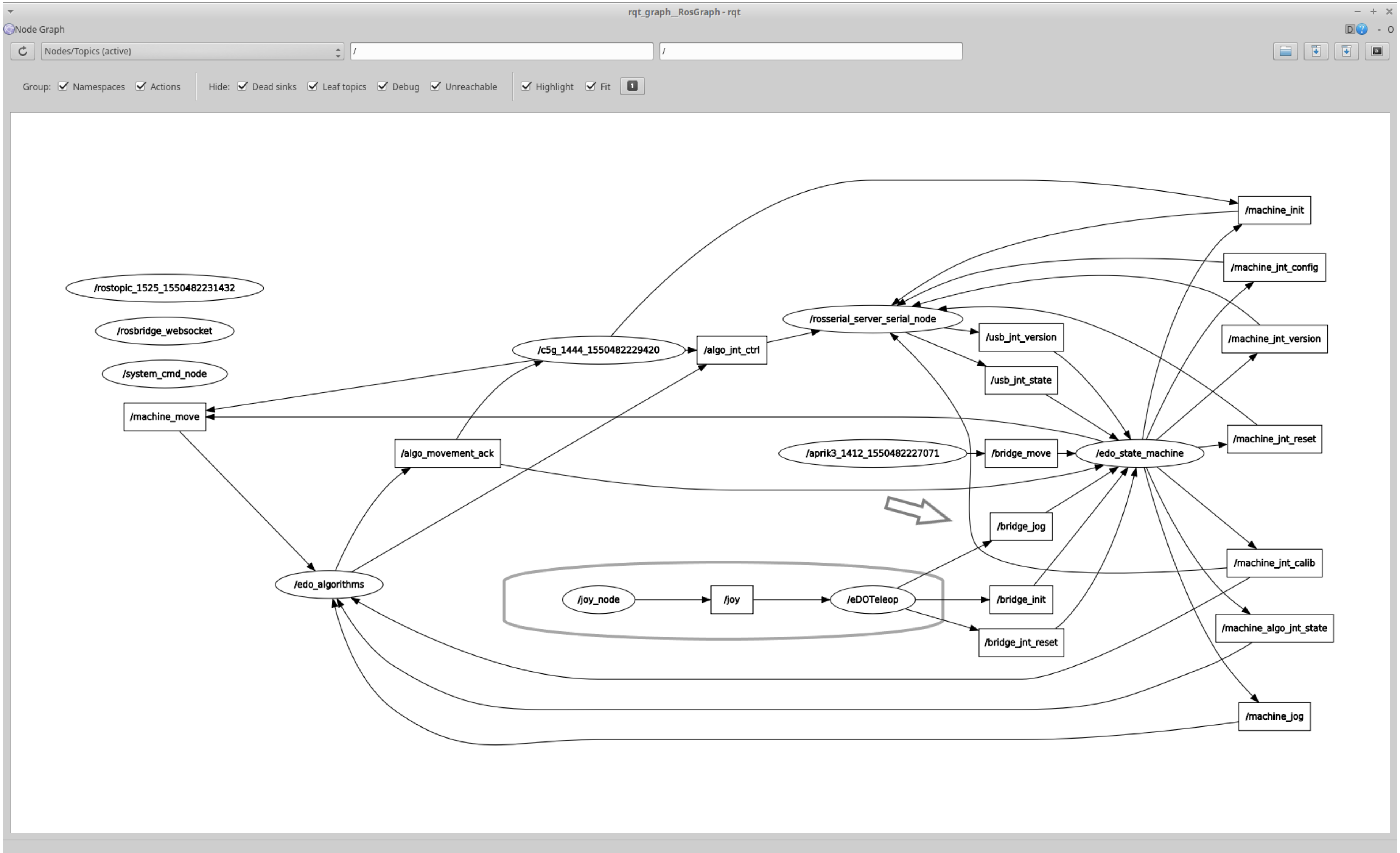


Figure 6.5.2 - The **rqt_graph** feedback after the files have been executed correctly