Applications of ROS

Topics being covered

- roslaunch - how to use it and how to make launch files.
- getting and using command line arguments in a ROS program.
- Serial protocol - how it works logically
- Useful ROS debugging options.

Serialization program

- Assume you have a servo controller that you really wanted to use with a robot that is using ROS. That is the basis for today's workshop.
- Your arduino will be a servo controller, capable of driving one servo.
- We are going to create a ROS node that will take as input string commands for the controller, and output whatever the controller says back.
- We will use three nodes: serializer, talker, listener.

Our servo controller

- Our servo controller is very simplistic. It takes commands over serial and uses them to do different functions. All commands are also terminated by a '\n' or newline to be recognized by the controller.
  - a <integer> : moves the servo to the integer angle
    - the integer can be from 0 to 180.
  - s : sweeps the servo from 0 to 180 and back.
  - o : turns the PWMs off, stopping the servo.
  - p <integer>: sets the pulse width of the PWM timer. Allows more accurate positioning than the use of a <integer>.
    - the integer can be from 700 to 2300.

Node descriptions

- serializer: this will be the node that does the communicating with the serial port. It will take string messages and serialize them to be sent over a serial port. It will then listen for any response on the serial port, deserialize that data and publish it.

- talker: this is the same publisher we made last time but now we are going to use it to send commands to the serializer node.

- listener: this is the same listener we made last time but now we are going to use it to listen for replies from the serializer node.

Serial

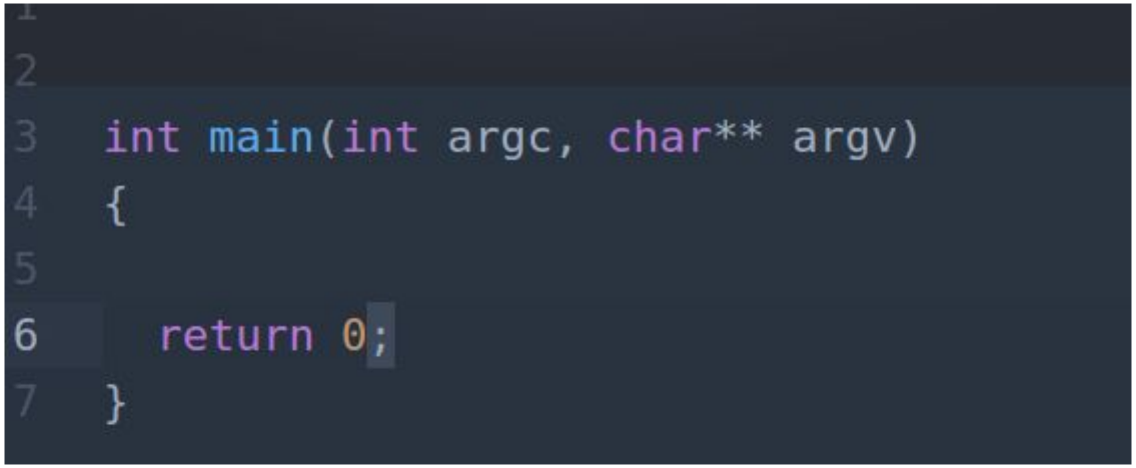
- Serial is a communications protocol where data is expressed by quantizing it into bytes and sending each byte one after another at a fixed rate (ie. serially).
- On a PC there is no serial port these days but data can be formatted as serial and sent over USB. We will be using a serial library to do this.
- Serial UARTs are implemented in many popular microcontrollers so it is a popular interface to use for inter-circuit communication.

Setting up the workspace

- git clone https://github.com/SIUE-SIGR/ROS-Workshop
- If you do not have internet or for whatever reason cant use git, a flash drive with the files on it.
- cd ~
- mkdir -p workshop2_ws/src
- cd workshop2_ws/src
- catkin_init_workspace
- cp -r ~/ROS-Workshop/ROS\ Workshop\ #2\ Material/serial/
- git clone https://github.com/wjwwood/serial

Code

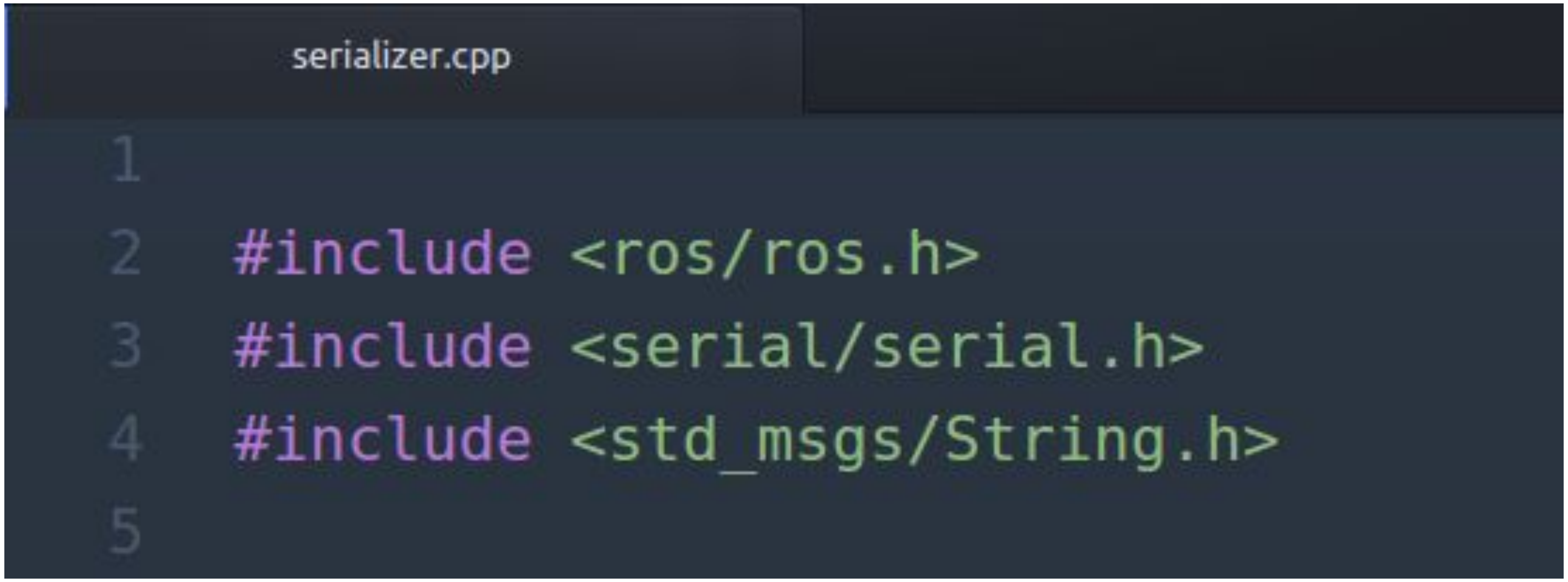
- cd ~/workshop2_ws/src/serializer/src
- Using your text editor of choice open serializer.cpp
- It should look like this:



```

int main(int argc, char** argv)
{

  return 0;
}
```

Code

- Now lets include our libraries



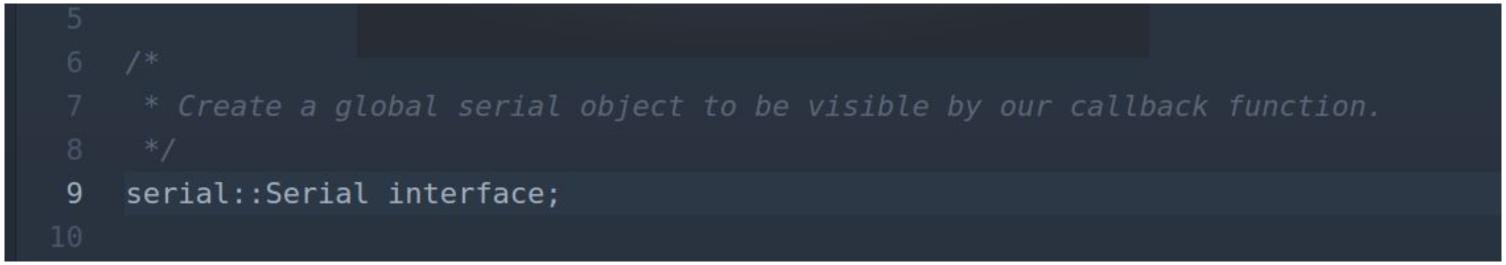
serializer.cpp

```cpp

#include <ros/ros.h>
#include <serial/serial.h>
#include <std_msgs/String.h>

```

- ros.h and std_msgs/String.h will be used for the ROS side of things just like last time.
- serial.h contains definititions for our serial library.

Code

- Now we create a global serial object.

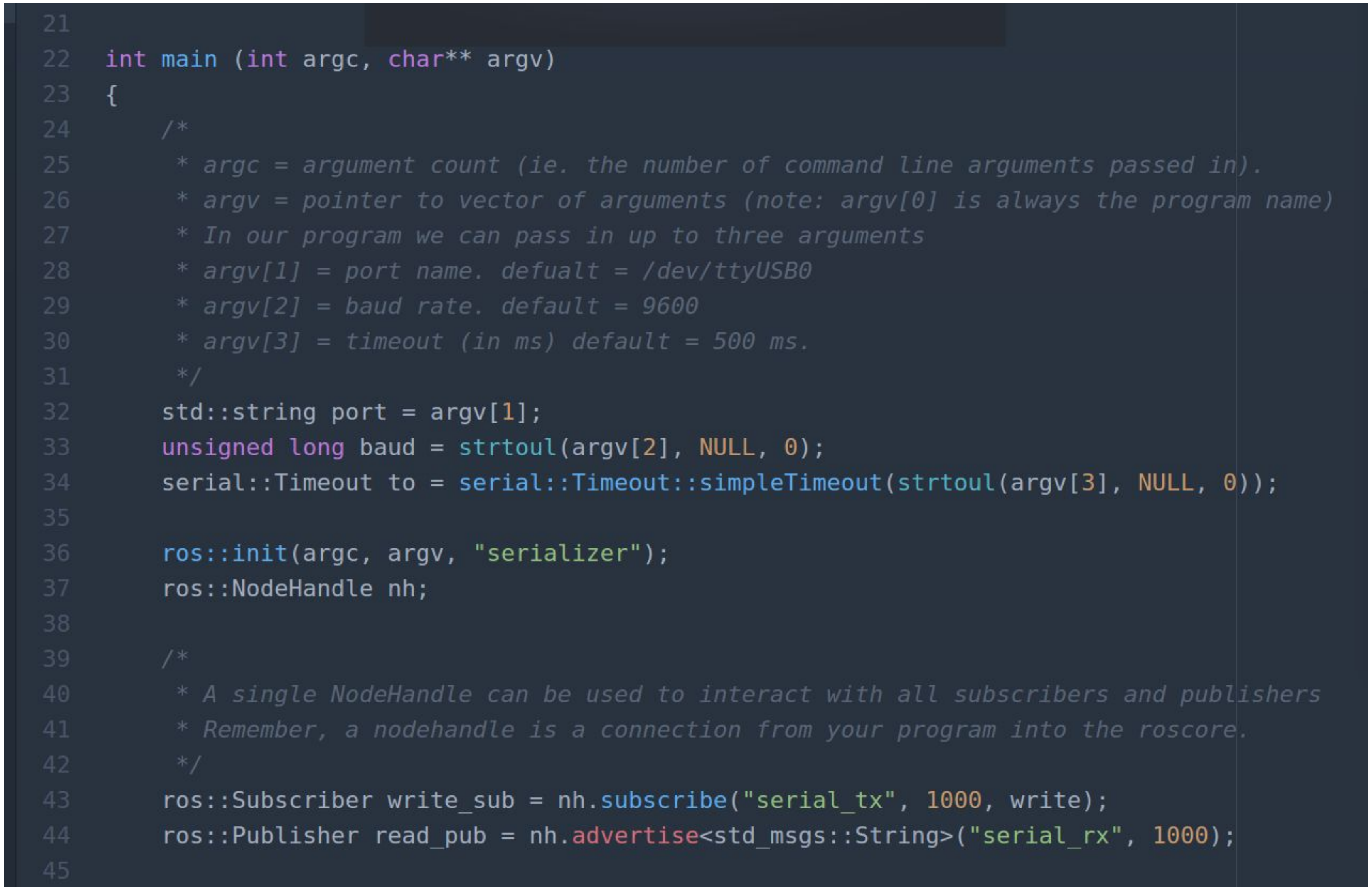


```
/*
 * Create a global serial object to be visible by our callback function.
 */
serial::Serial interface;
```

- This serial object needs to be global so that it can be seen in all functions. This is neccessary since our callback function will need to be able to write to the serial port using this object.

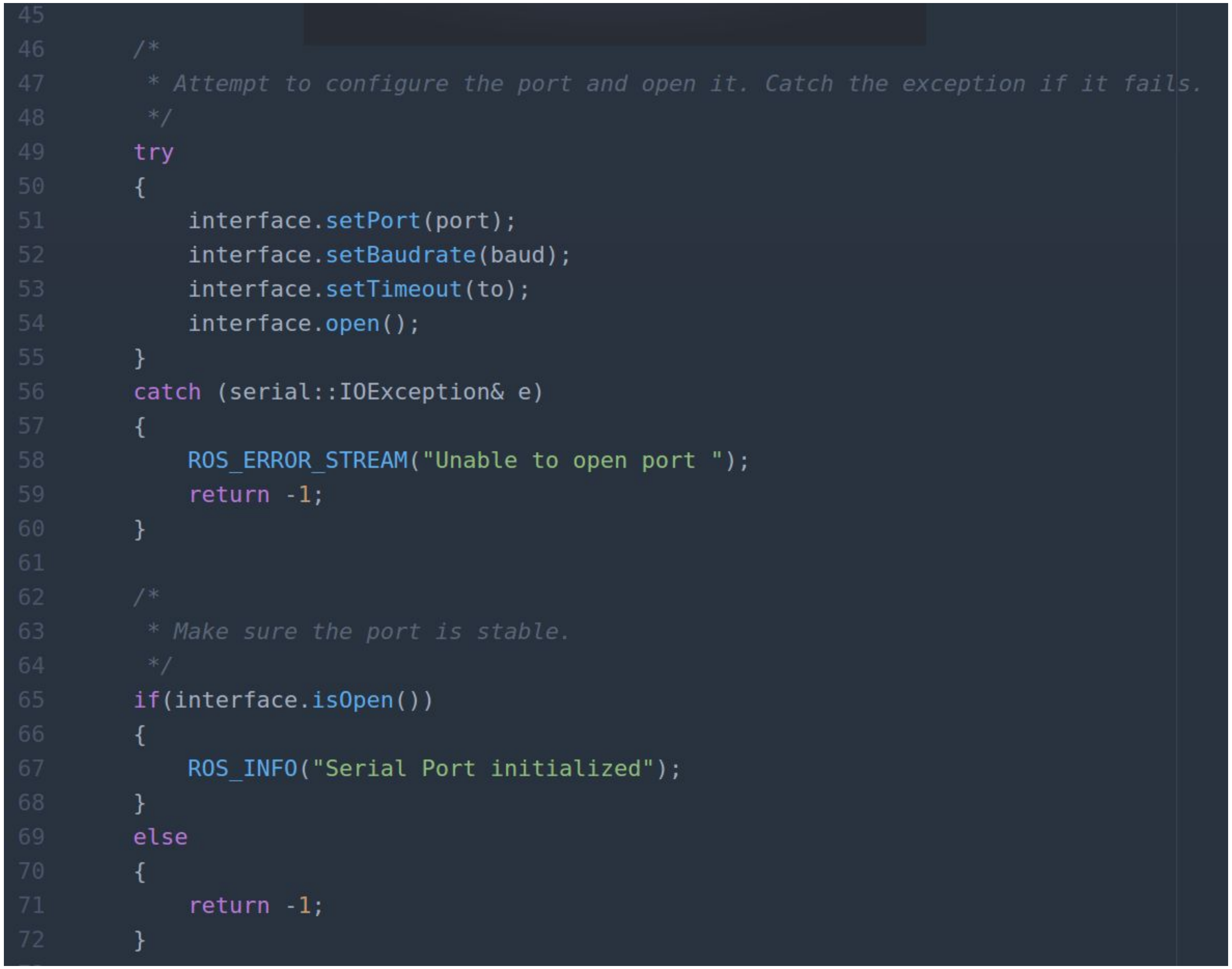
- Now we can start to fill out main() a little bit more.

Code



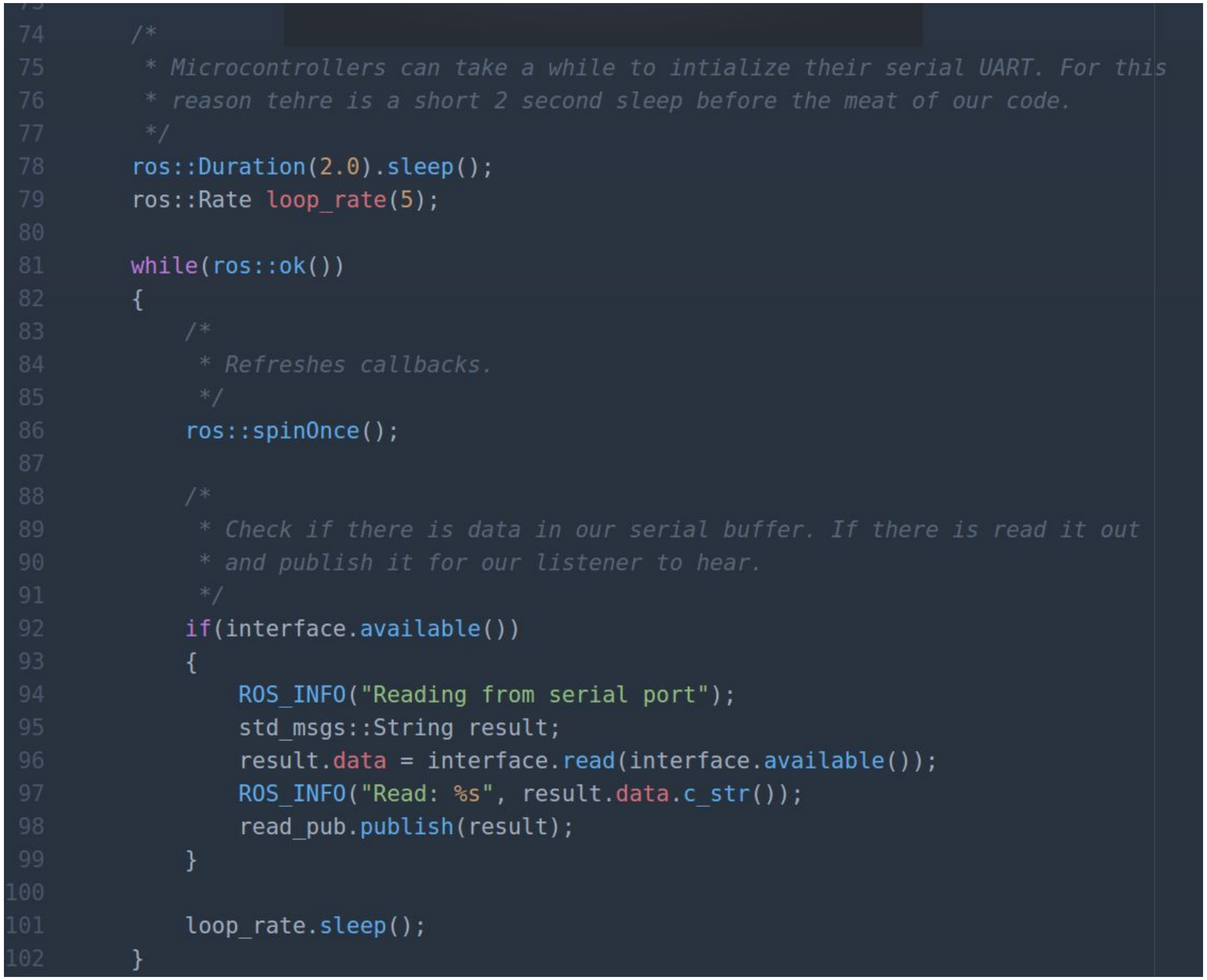
```

int main (int argc, char** argv)
{
    /*
     * argc = argument count (ie. the number of command line arguments passed in).
     * argv = pointer to vector of arguments (note: argv[0] is always the program name)
     * In our program we can pass in up to three arguments
     * argv[1] = port name. defualt = /dev/ttyUSB0
     * argv[2] = baud rate. default = 9600
     * argv[3] = timeout (in ms) default = 500 ms.
     */
    std::string port = argv[1];
    unsigned long baud = strtoul(argv[2], NULL, 0);
    serial::Timeout to = serial::Timeout::simpleTimeout(strtoul(argv[3], NULL, 0));

    ros::init(argc, argv, "serializer");
    ros::NodeHandle nh;

    /*
     * A single NodeHandle can be used to interact with all subscribers and publishers
     * Remember, a nodehandle is a connection from your program into the roscore.
     */
    ros::Subscriber write_sub = nh.subscribe("serial_tx", 1000, write);
    ros::Publisher read_pub = nh.advertise<std_msgs::String>("serial_rx", 1000);

```

Code



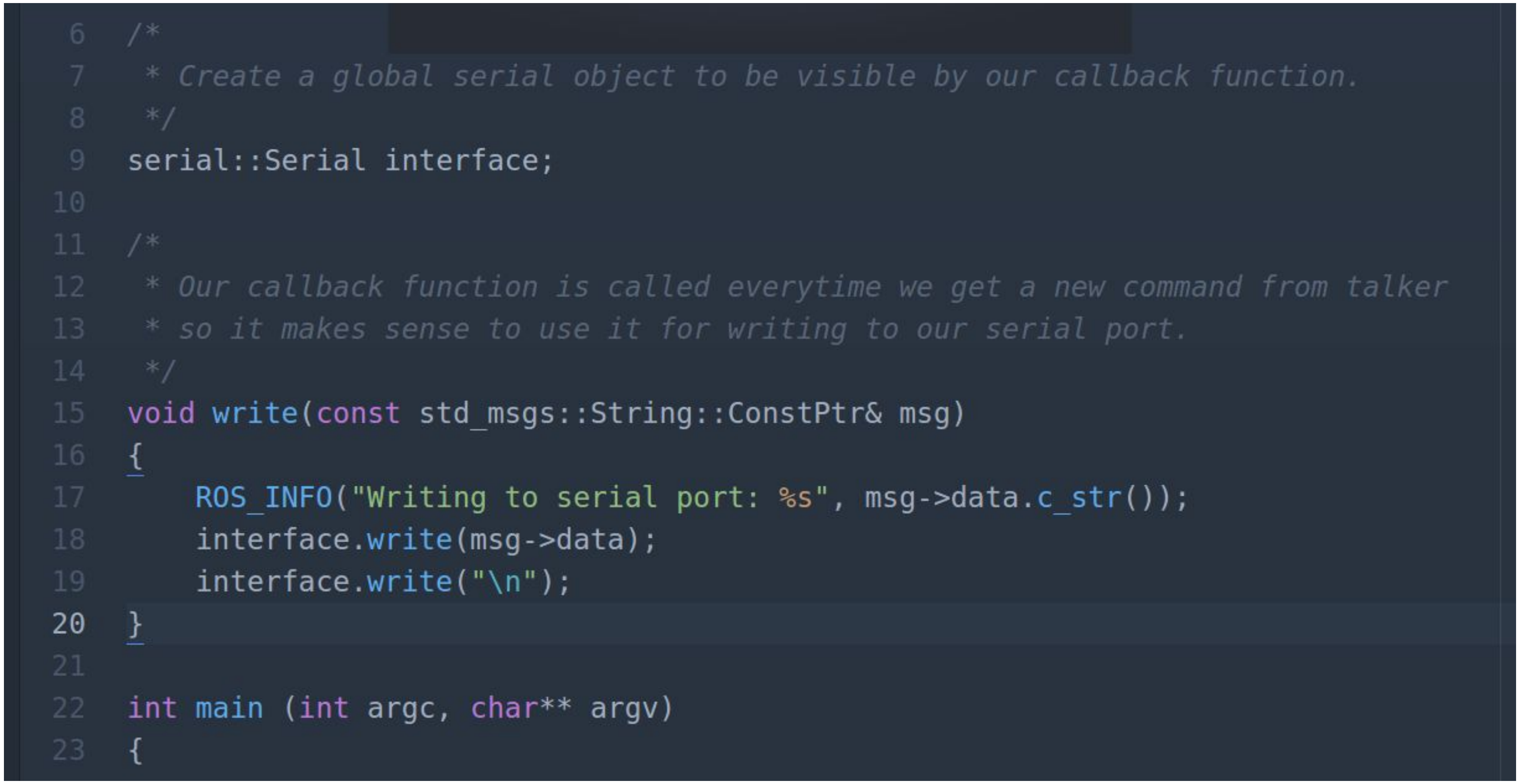
```

    /*
     * Attempt to configure the port and open it. Catch the exception if it fails.
     */
    try
    {
        interface.setPort(port);
        interface.setBaudrate(baud);
        interface.setTimeout(to);
        interface.open();
    }
    catch (serial::IOException& e)
    {
        ROS_ERROR_STREAM("Unable to open port ");
        return -1;
    }

    /*
     * Make sure the port is stable.
     */
    if(interface.isOpen())
    {
        ROS_INFO("Serial Port initialized");
    }
    else
    {
        return -1;
    }
```

Code



```
    /*
     * Microcontrollers can take a while to intialize their serial UART. For this
     * reason tehre is a short 2 second sleep before the meat of our code.
     */
    ros::Duration(2.0).sleep();
    ros::Rate loop_rate(5);

    while(ros::ok())
    {
        /*
         * Refreshes callbacks.
         */
        ros::spinOnce();

        /*
         * Check if there is data in our serial buffer. If there is read it out
         * and publish it for our listener to hear.
         */
        if(interface.available())
        {
            ROS_INFO("Reading from serial port");
            std_msgs::String result;
            result.data = interface.read(interface.available());
            ROS_INFO("Read: %s", result.data.c_str());
            read_pub.publish(result);
        }

        loop_rate.sleep();
    }
```

Code



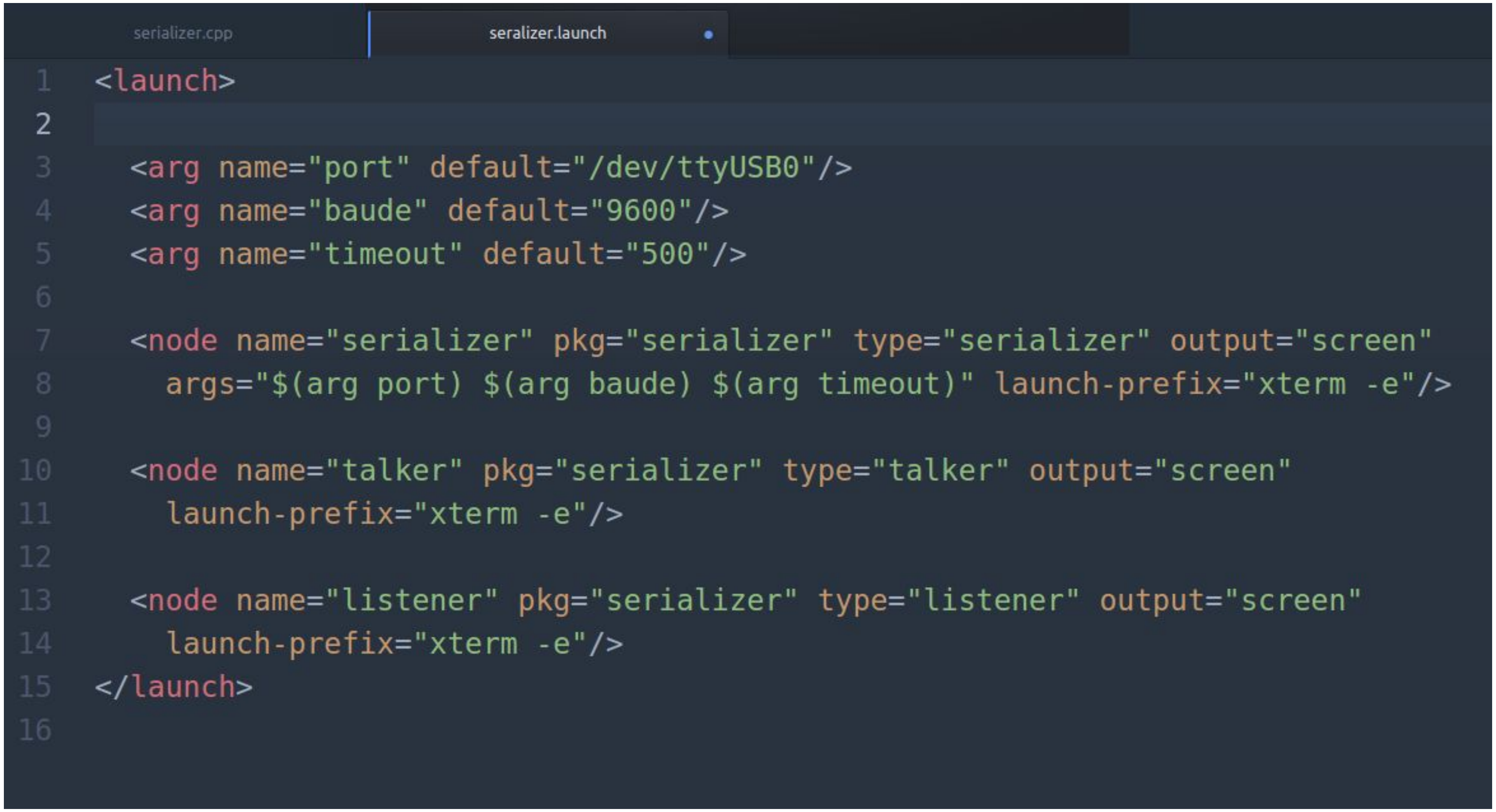
```
/*
 * Create a global serial object to be visible by our callback function.
 */
serial::Serial interface;

/*
 * Our callback function is called everytime we get a new command from talker
 * so it makes sense to use it for writing to our serial port.
 */
void write(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("Writing to serial port: %s", msg->data.c_str());
    interface.write(msg->data);
    interface.write("\n");
}

int main (int argc, char** argv)
{
```

Roslaunch

- Now we don't want to have to run three seperate rosrun commands just to make our program work. If this was a robot we would want all of the neccessary nodes to launh at once.

- For this we will make a roslaunch file that will allow us to do a few things:
    - Get command line arguments into our serializer.cpp program.
    - Launch all three nodes at the same time.
    - Get a free copy of roscore!

Launch file

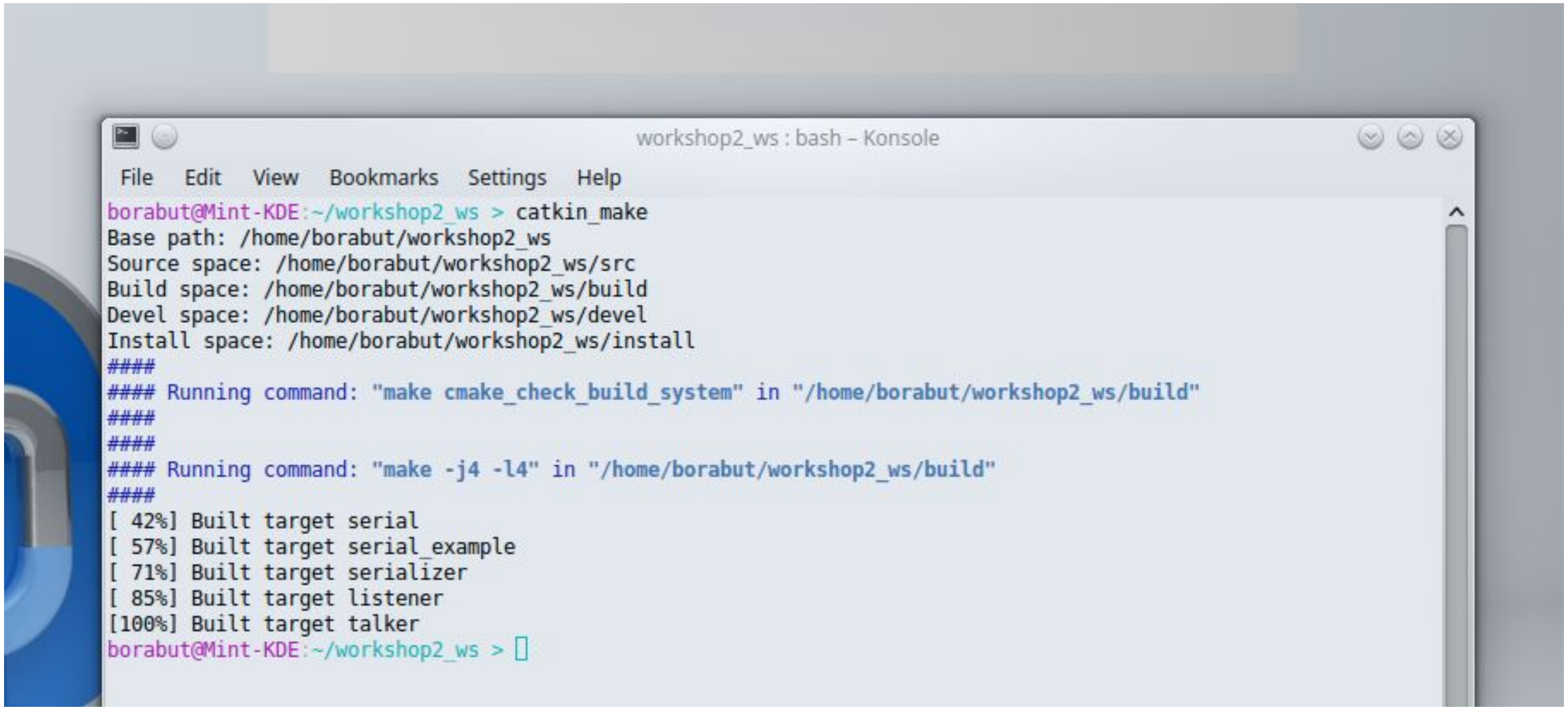


```
<launch>

  <arg name="port" default="/dev/ttyUSB0"/>
  <arg name="baude" default="9600"/>
  <arg name="timeout" default="500"/>

  <node name="serializer" pkg="serializer" type="serializer" output="screen"
    args="$(arg port) $(arg baude) $(arg timeout)" launch-prefix="xterm -e"/>

  <node name="talker" pkg="serializer" type="talker" output="screen"
    launch-prefix="xterm -e"/>

  <node name="listener" pkg="serializer" type="listener" output="screen"
    launch-prefix="xterm -e"/>
</launch>
```

Building our project

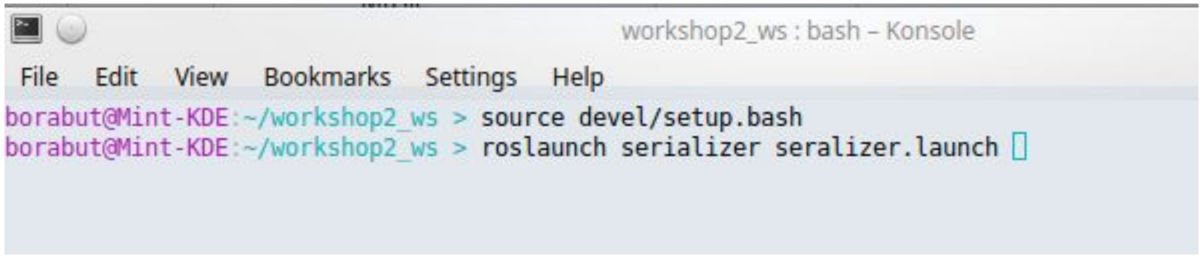
- cd ~/workshop2_ws
- catkin_make
- You should see catkin and CMake output in your terminal, this is normal. At the end it should all work out and targets will be built and linked successfully.
- If you do have problems building it will tell you what line in the code caused it if it was a problem with your programs. It will also give other useful debug output for solving other issues.

Build output


workshop2_ws : bash – Konsole
File Edit View Bookmarks Settings Help

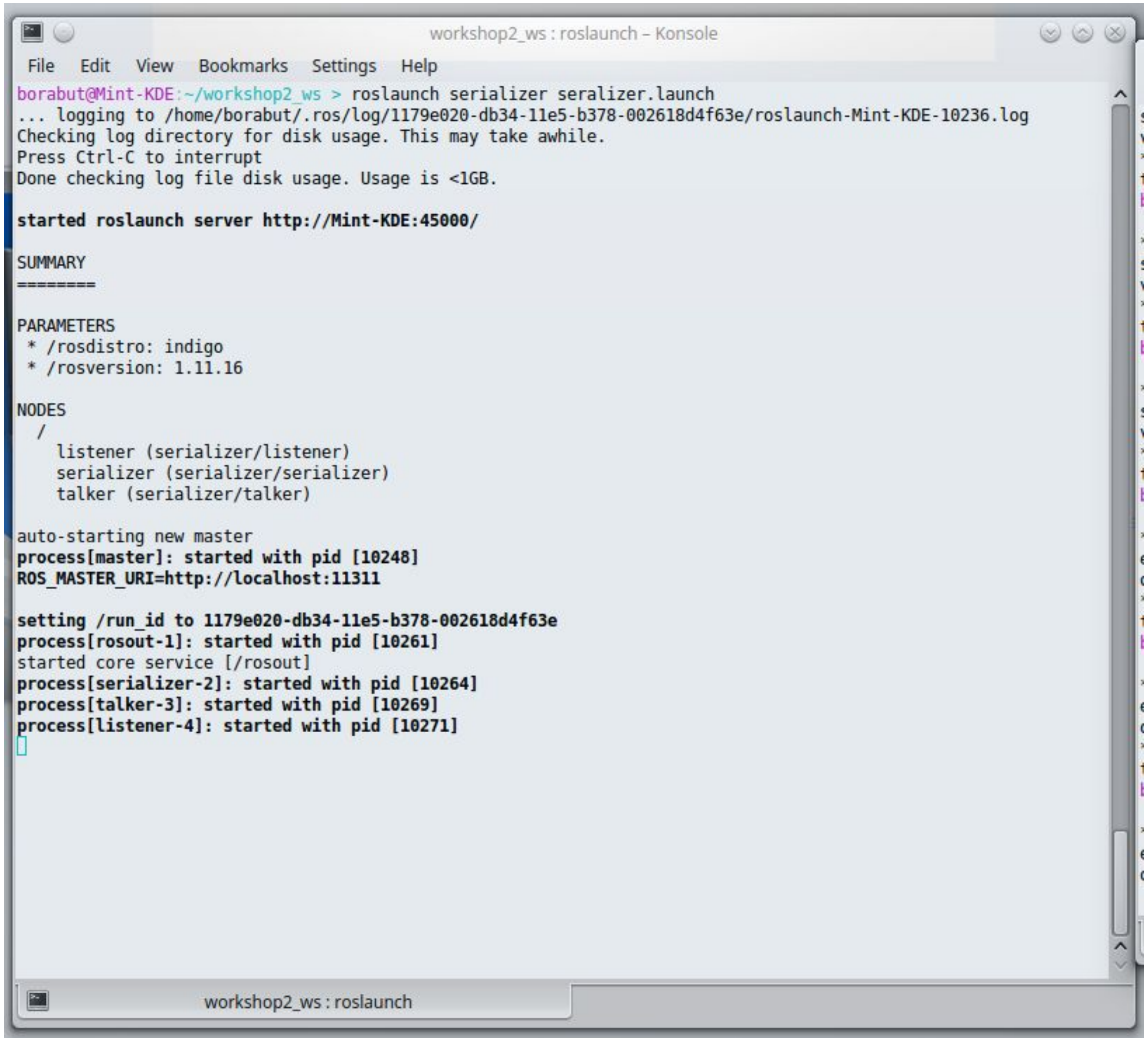
```
borabut@Mint-KDE:~/workshop2_ws > catkin_make
Base path: /home/borabut/workshop2_ws
Source space: /home/borabut/workshop2_ws/src
Build space: /home/borabut/workshop2_ws/build
Devel space: /home/borabut/workshop2_ws/devel
Install space: /home/borabut/workshop2_ws/install
####
#### Running command: "make cmake_check_build_system" in "/home/borabut/workshop2_ws/build"
####
####
#### Running command: "make -j4 -l4" in "/home/borabut/workshop2_ws/build"
####
[ 42%] Built target serial
[ 57%] Built target serial_example
[ 71%] Built target serializer
[ 85%] Built target listener
[100%] Built target talker
borabut@Mint-KDE:~/workshop2_ws > 
```

- Next we will source out setup file to make our package visible globally.
- Then we will launch our program!


workshop2_ws : bash – Konsole
File Edit View Bookmarks Settings Help

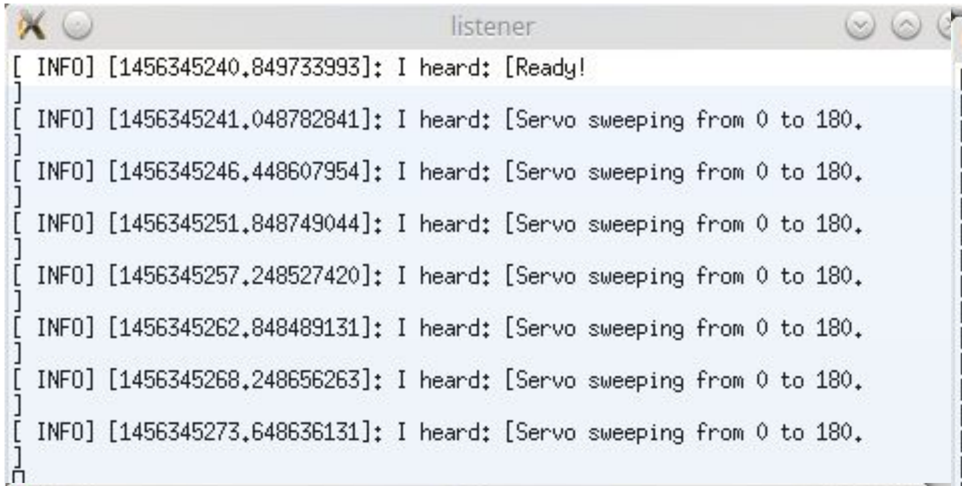
```
borabut@Mint-KDE:~/workshop2_ws > source devel/setup.bash
borabut@Mint-KDE:~/workshop2_ws > roslaunch serializer seralizer.launch
```

- You should see a ROS master in your main terminal and three xterm windows containing the outputs of your programs.

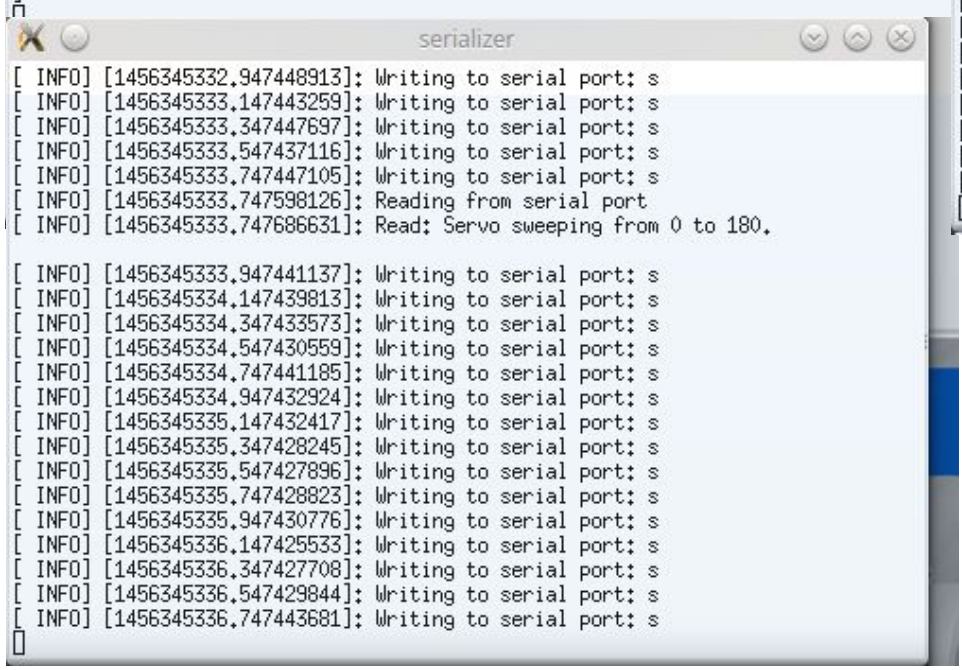

workshop2_ws : roslaunch – Konsole
File Edit View Bookmarks Settings Help
workshop2_ws : roslaunch

```
borabut@Mint-KDE:~/workshop2_ws > roslaunch serializer seralizer.launch
... logging to /home/borabut/.ros/log/1179e020-db34-11e5-b378-002618d4f63e/roslaunch-Mint-KDE-10236.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://Mint-KDE:45000/

SUMMARY
========

PARAMETERS
 * /rosdistro: indigo
 * /rosversion: 1.11.16

NODES
  /
    listener (serializer/listener)
    serializer (serializer/serializer)
    talker (serializer/talker)

auto-starting new master
process[master]: started with pid [10248]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to 1179e020-db34-11e5-b378-002618d4f63e
process[rosout-1]: started with pid [10261]
started core service [/rosout]
process[serializer-2]: started with pid [10264]
process[talker-3]: started with pid [10269]
process[listener-4]: started with pid [10271]
```

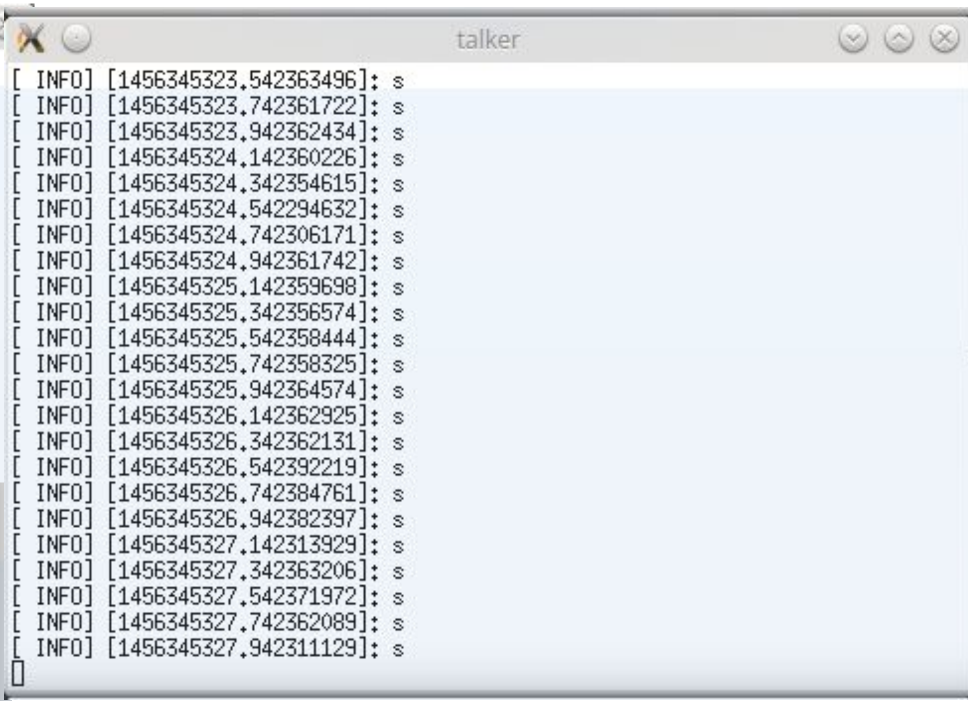
Sample output



```
[ INFO] [1456345240.849733993]: I heard: [Ready!
]
[ INFO] [1456345241.048782841]: I heard: [Servo sweeping from 0 to 180.
]
[ INFO] [1456345246.448607954]: I heard: [Servo sweeping from 0 to 180.
]
[ INFO] [1456345251.848749044]: I heard: [Servo sweeping from 0 to 180.
]
[ INFO] [1456345257.248527420]: I heard: [Servo sweeping from 0 to 180.
]
[ INFO] [1456345262.848489131]: I heard: [Servo sweeping from 0 to 180.
]
[ INFO] [1456345268.248656263]: I heard: [Servo sweeping from 0 to 180.
]
[ INFO] [1456345273.648636131]: I heard: [Servo sweeping from 0 to 180.
]
```



```
[ INFO] [1456345332.947448913]: Writing to serial port: s
[ INFO] [1456345333.147443259]: Writing to serial port: s
[ INFO] [1456345333.347447697]: Writing to serial port: s
[ INFO] [1456345333.547437116]: Writing to serial port: s
[ INFO] [1456345333.747447105]: Writing to serial port: s
[ INFO] [1456345333.747598126]: Reading from serial port
[ INFO] [1456345333.747686631]: Read: Servo sweeping from 0 to 180.

[ INFO] [1456345333.947441137]: Writing to serial port: s
[ INFO] [1456345334.147439813]: Writing to serial port: s
[ INFO] [1456345334.347433573]: Writing to serial port: s
[ INFO] [1456345334.547430559]: Writing to serial port: s
[ INFO] [1456345334.747441185]: Writing to serial port: s
[ INFO] [1456345334.947432924]: Writing to serial port: s
[ INFO] [1456345335.147432417]: Writing to serial port: s
[ INFO] [1456345335.347428245]: Writing to serial port: s
[ INFO] [1456345335.547427896]: Writing to serial port: s
[ INFO] [1456345335.747428823]: Writing to serial port: s
[ INFO] [1456345335.947430776]: Writing to serial port: s
[ INFO] [1456345336.147425533]: Writing to serial port: s
[ INFO] [1456345336.347427708]: Writing to serial port: s
[ INFO] [1456345336.547429844]: Writing to serial port: s
[ INFO] [1456345336.747443681]: Writing to serial port: s
```

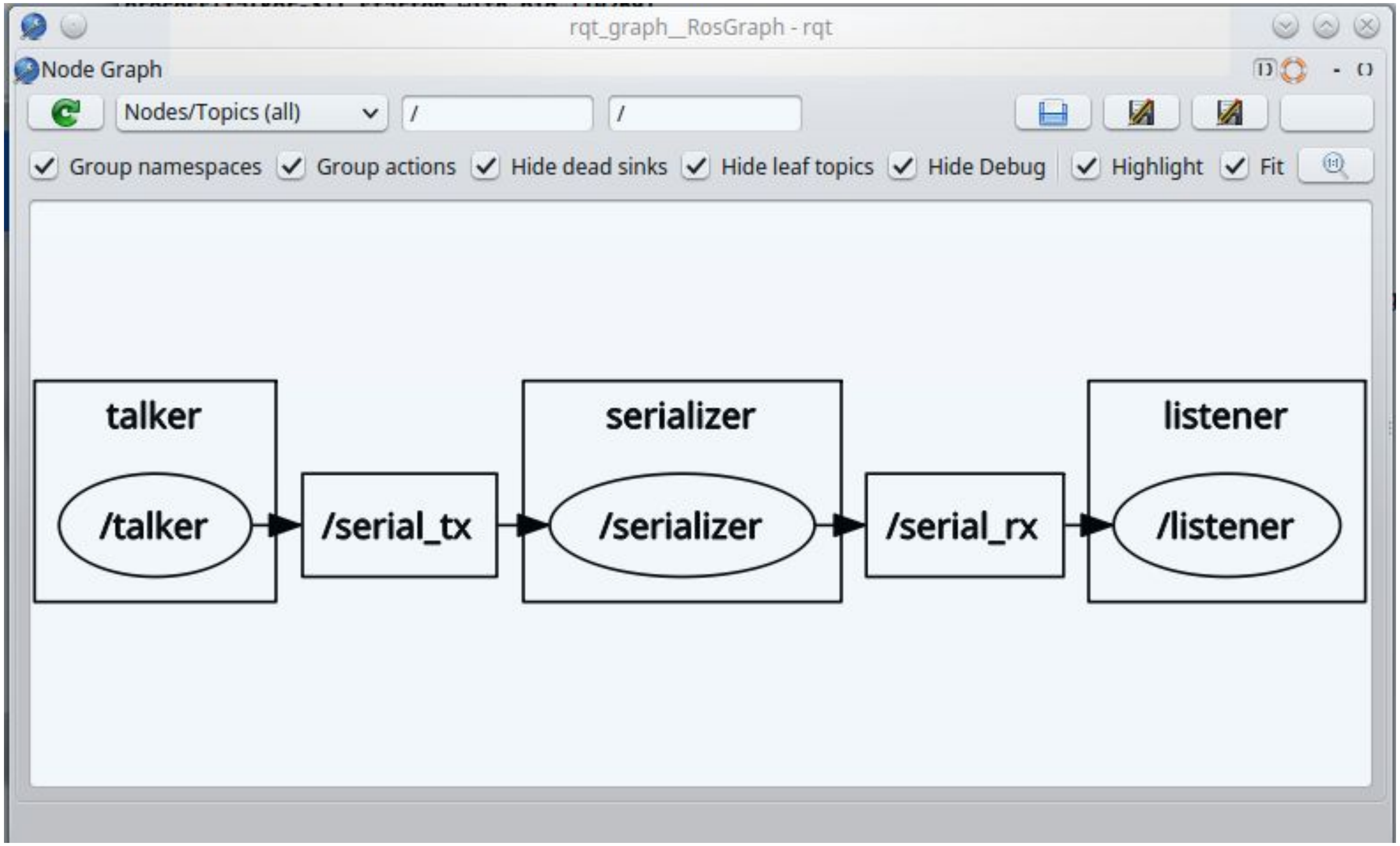


```
[ INFO] [1456345323.542363496]: s
[ INFO] [1456345323.742361722]: s
[ INFO] [1456345323.942362434]: s
[ INFO] [1456345324.142360226]: s
[ INFO] [1456345324.342354615]: s
[ INFO] [1456345324.542294632]: s
[ INFO] [1456345324.742306171]: s
[ INFO] [1456345324.942361742]: s
[ INFO] [1456345325.142359698]: s
[ INFO] [1456345325.342356574]: s
[ INFO] [1456345325.542358444]: s
[ INFO] [1456345325.742358325]: s
[ INFO] [1456345325.942364574]: s
[ INFO] [1456345326.142362925]: s
[ INFO] [1456345326.342362131]: s
[ INFO] [1456345326.542392219]: s
[ INFO] [1456345326.742384761]: s
[ INFO] [1456345326.942382397]: s
[ INFO] [1456345327.142313929]: s
[ INFO] [1456345327.342363206]: s
[ INFO] [1456345327.542371972]: s
[ INFO] [1456345327.742362089]: s
[ INFO] [1456345327.942311129]: s
```

Visual debugging

- Lets assume we had some problems with our node. It was running but wasn't doing what we expected it to.
- One issue could be that some nodes are not publishing or subscribing to the right topics. Or maybe some nodes didn't get launched properly.
- ROS has a program called rqt_graph that will let us debug these issues.
- type rosrun rqt_graph rqt_graph

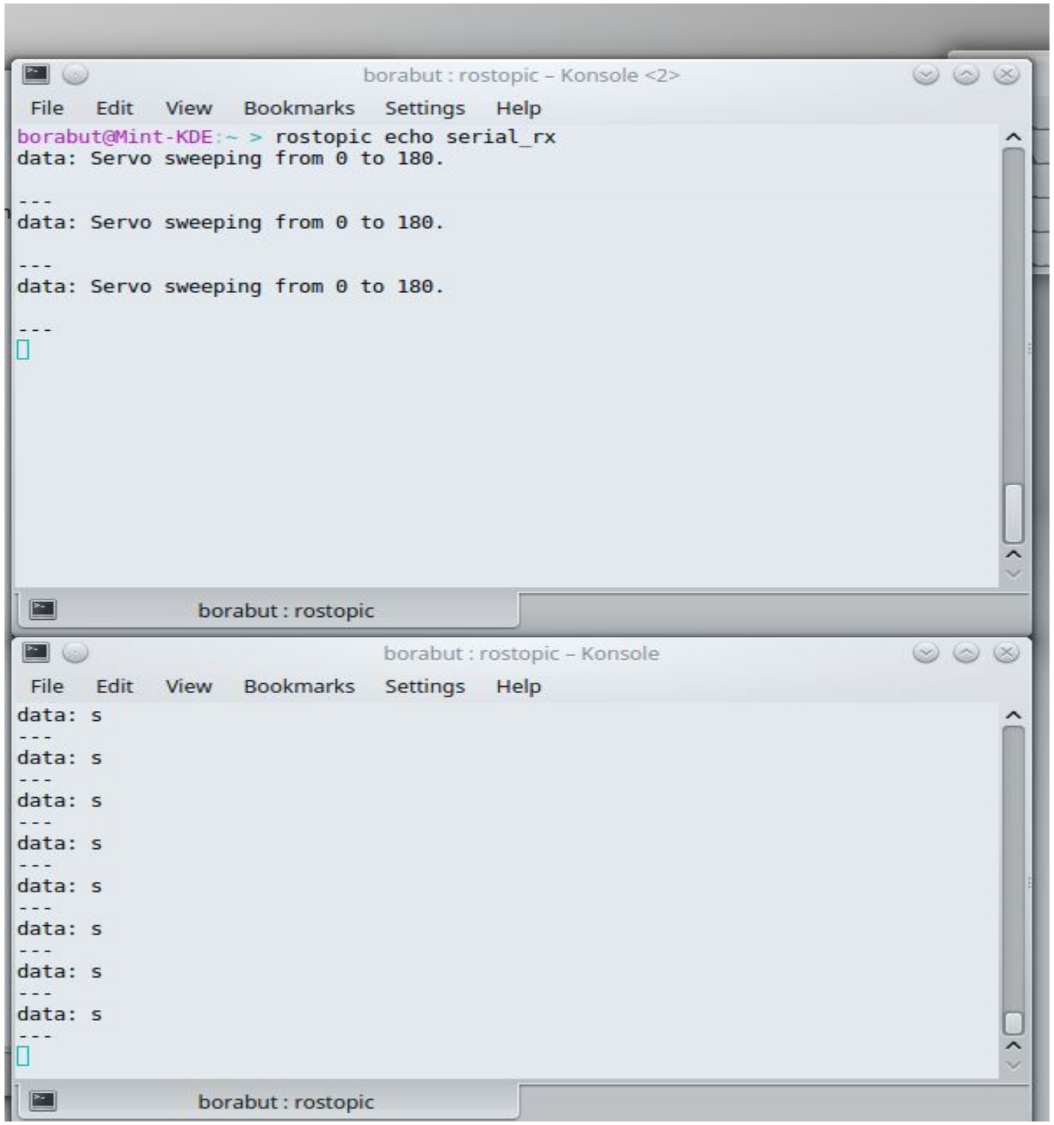
rqt_graph


rqt_graph__RosGraph - rqt
Node Graph
Nodes/Topics (all)
/
/
✓ Group namespaces
✓ Group actions
✓ Hide dead sinks
✓ Hide leaf topics
✓ Hide Debug
✓ Highlight
✓ Fit
talker
/talker
/serial_tx
serializer
/serializer
/serial_rx
listener
/listener

terminal debugging

- Another useful debugging tool rostopic echo.
- This will let us see if data is actually being published on any given topic. Let's try, open two terminals and type these commands, one in each terminal
- rostopic echo serial_tx
- rostopic echo serial_rx

rostopic echo


borabut : rostopic – Konsole <2>
File Edit View Bookmarks Settings Help

```
borabut@Mint-KDE:~ > rostopic echo serial_rx
data: Servo sweeping from 0 to 180.

---
data: Servo sweeping from 0 to 180.

---
data: Servo sweeping from 0 to 180.

---
```

borabut : rostopic
borabut : rostopic – Konsole
File Edit View Bookmarks Settings Help

```
data: s
---
data: s
---
data: s
---
data: s
---
data: s
---
data: s
---
data: s
---
data: s
---
```

The end

- We have now successfully (hopefully), made a ROS program that can actually interact with a robotic system.
- Our servo controller was very simple and basic but these same techniques can easily be expanded to more complex programs.
- Questions?