Introduction to ROS2 Basics 2 (parameters, launch files, LED circuit exercise)

Introduction to Robotics Lab

Konstantinos Asimakopoulos *k_asimakopoulos@ac.upatras.gr*
Lampros Printzios *printzios_lampros@ac.upatras.gr*

Department of Electrical and Computer Engineering
University of Patras

October 21, 2025

Table of Contents

Parameters - What are parameters?

1. **Parameters** are key - value pairs owned by a node, making it more ***reusable*** and ***configurable***.
2. We use parameters to **change the behavior of nodes** without having to modify their source code. They can be used from ***tuning*** algorithm settings, to providing ***environment-specific configuration***.
3. Parameters can be:
   - **Static:** read once (one-off) at node startup
   - **Dynamic:** read periodically and updated during runtime via callbacks
4. Parameters are set:
   - **programmatically** using the ROS2 client libraries (e.g. rclpy, rclcpp)
   - through **configuration files (YAML)**
   - from the **Command Line Interface (CLI)**

Parameters - Declare parameters

In the `luna_commander` node we declare the *static* parameters `period` & `topic_name` and the *dynamic* ones (because of the `timer_callback`) `linear_speed` & `angular_speed`. Note that the static parameter `period` could be dynamic if we had created a proper callback function and used the method `add_on_set_parameters_callback` (more on that later).

```python
class LunaCommanderNode(Node):
    def __init__(self) -> None:
        super().__init__('luna_commander')

        # Static parameters
        self.declare_parameter('topic_name', 'cmd_vel')
        topic_name = self.get_parameter('topic_name').value
        [redacted]

        # Dynamic parameters
        self.declare_parameter('linear_speed', 0.2)
        [redacted]

    def timer_callback(self) -> None:
        v = self.get_parameter('linear_speed').value
        [redacted]
```

parameters declared for luna_commander node

Declare parameter

`declare_parameter(name, value)`

Get parameter value

`get_parameter(name).value`

Parameters - Manage parameters via CLI

1. While running a node called `<node_name>` with *ros2 run*, we can access its parameters and read/change them. `ros2 param list` outputs a list of the available/visible parameters.
2. We can **get/read static and dynamic parameters** like this:

Get Parameter Value

```
ros2 param get /<node_name> <parameter_name>
```

3. We can **set/change only dynamic parameters** like this:

Set Parameter Value

```
ros2 param set /<node_name> <parameter_name> <value>
```

ROS2 Parameters Functionalities

```
ros2 param -h
```

Parameters - Manage parameters changes

1. It's often useful for a ***node to react*** whenever (dynamic) parameters are ***changed at runtime*** (e.g. via CLI). Some example cases are:
   - **preventing invalid values**
   - dynamically **adjusting timers**
   - **logging** or **monitoring** parameter updates
2. How it works:
   - using the method `add_on_set_parameters_callback(callback)`, register a **callback function** to be called when a parameter is set
   - the callback function must return a `SetParameterResult` message to accept or reject the change

```python
from rclpy.parameter import Parameter
from rcl_interfaces.msg import SetParametersResult
def parameters_callback(self, params: List[Parameter]):
    for param in params:
        ...  # what to do for every changed parameter
    return SetParametersResult(successful = True, reason = "Correct!")
```

parameters properties

`name`

`type_`

`value`

Parameters - YAML Files

1. **YAML** (YAML Ain't Markup Language) is a human-readable data serialization format, widely used for **configuration files** and for exchanging **structured data**.
2. In ROS2, YAML files are often used to **store** and **reload** the parameters of a node.
3. The general structure of a ROS2 parameter file looks like this:

```yaml
/luna_commander:
  ros__parameters:
    angular_speed: 0.7
    linear_speed: 1.4
    period: 2.8
    topic_name: "luna_cmd_vel"
```

Example YAML file
luna_commander.yaml

YAML file structure

- The top-level key corresponds to the `<node_name>`.
- All parameters must be placed under the special key `ros__parameters`.
- Parameters can be *integers*, *floats*, *strings*, *booleans*, *lists*, ...

Parameters - Saving & Loading Parameters

1. When a node is running with a working configuration, it can be useful to save its parameters for later use. Via the CLI, we can export the node's parameters to a YAML file called `<node_name>.yaml`, by typing the following command (a common good practice is to save the parameters into a *config* folder in the package directory, so `--output-dir` would be `./config`):

Export node parameters to YAML

```
ros2 param dump /<node_name> --output-dir <abs_path>
```

2. The saved parameters can then be reloaded into the node `<node_name>` at any time using:

Load parameters from file

```
ros2 param load /<node_name> <parameter_file>
```

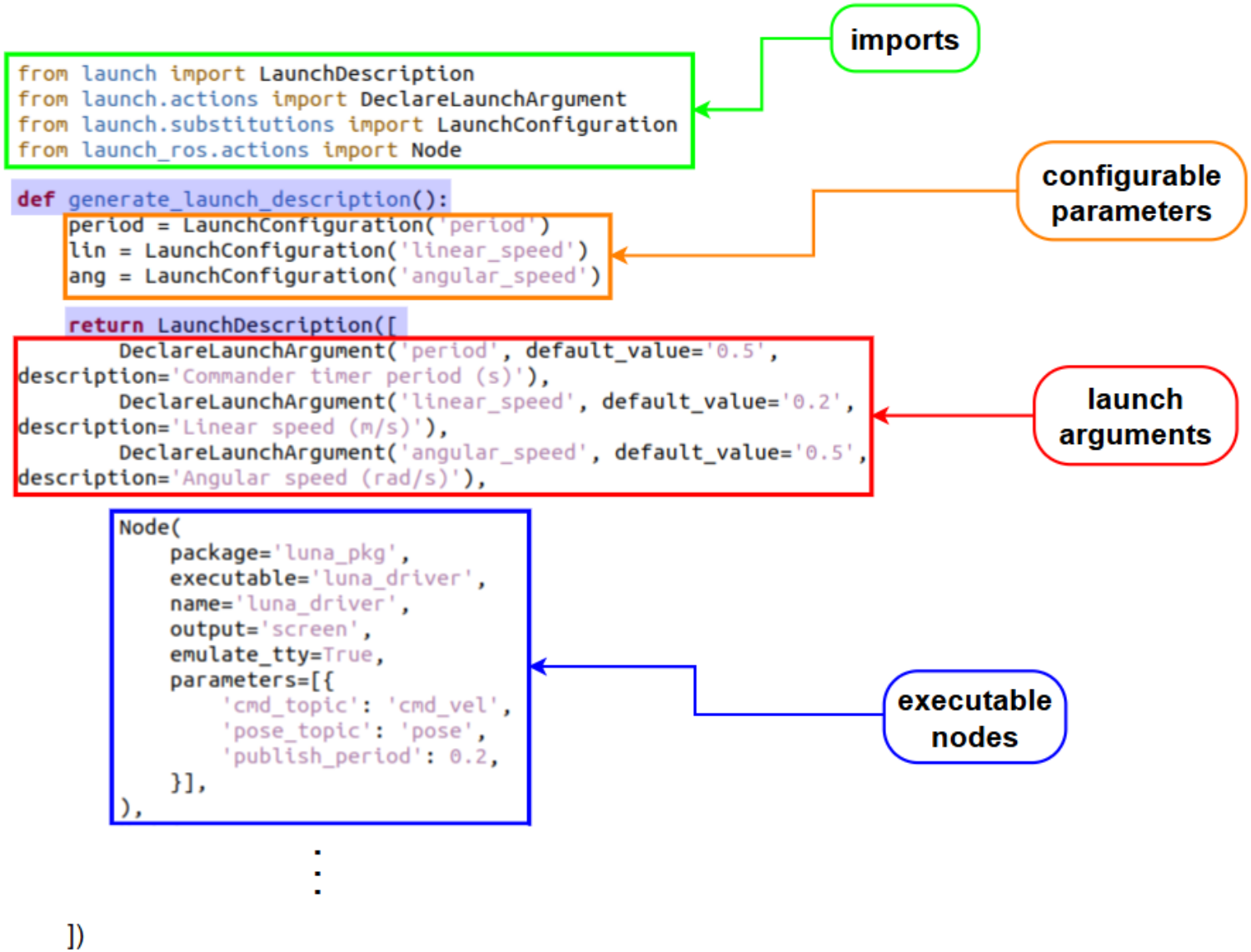
Launch files - What are launch files?

1. A **launch file** is a ***Python script*** that describes how to ***start up*** and ***configure*** multiple **ROS2 nodes** together (the nodes may be written in any programming language: Python, C++, etc.).
2. Launch files provide:
   - **Automation:** run several nodes with a single command
   - **Configuration:** set parameters, remap topics, declare arguments
   - **Reusability:** reuse the same launch description for different experiments/environments
3. A common good practice is to save the launch files into a *launch* folder in the package directory.

ROS2 Launch Functionalities

```
ros2 launch -h
```

Launch files - Creating a launch file



launch file simple example

Launch files - Creating a launch file - Adding executable nodes

Each `Node()` **action** launches one ROS2 node.

```
Node(
    package='luna_pkg',
    executable='luna_driver',
    name='luna_driver',
    output='screen',
    emulate_tty=True,
    parameters=[{
        'cmd_topic': 'cmd_vel',
        'pose_topic': 'pose',
        'publish_period': 0.2,
    }],
),
```

Executable Node arguments

- `package`: ROS2 package where the executable lives
- `executable`: Node's binary or Python entry point
- `name`: Logical node name (overrides internal name)
- `output`: Logs go to `screen` (terminal) or `log` (file)
- `emulate_tty`: Enables formatted log output
- `parameters`: YAML list of ROS2 parameters

Remappings

- Allow changing node/topic/service/action names *without modifying code*
- Syntax: `remappings=[('old_name', 'new_name')]`
- CLI equivalent: `ros2 run <pkg> <exec_node> --ros-args -r old_name:=new_name`

Launch files - Creating a launch file - Declaring Parameters

1. Enable **flexible** and **reusable** launch files, allowing **runtime tuning**.
2. **Parameters** are node values; **Launch Arguments** are external inputs (from launch file or CLI).

```
period = LaunchConfiguration('period')
lin = LaunchConfiguration('linear_speed')
ang = LaunchConfiguration('angular_speed')

return LaunchDescription([
    DeclareLaunchArgument('period', default_value='0.5', description='Commander timer period (s)'),
    DeclareLaunchArgument('linear_speed', default_value='0.2', description='Linear speed (m/s)'),
    DeclareLaunchArgument('angular_speed', default_value='0.5', description='Angular speed (rad/s)'),
```

LaunchConfiguration & DeclareLaunchArgument

- `DeclareLaunchArgument`: defines a configurable variable for the launch file
- `LaunchConfiguration`: retrieves the runtime value of that variable (from default or CLI)

Parameter Priority (Highest → Lowest)

1. Launch file parameters at startup
2. YAML file parameters (`parameters=[<YAML_file_path> or <YAML_list>]`)
3. Node's internal defaults (in source code)

Launch files - Creating a launch file - Modifying setup.py

```
import os
from glob import glob
from setuptools import find_packages, setup

package_name = 'luna_pkg'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name, 'launch'), glob(os.path.join('launch',
'*launch.[pxy][yma]*'))),
        (os.path.join('share', package_name, 'config'), glob(os.path.join('config',
'*.yaml'))),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='lampros',
    maintainer_email='[redacted]',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'luna_commander=luna_pkg.luna_commander_node:main',
            '[redacted]'
        ],
    },
)
```

setup.py file structure

Important points:

- `packages=find_packages()`: includes the python modules
- `data_files`: installs non-python assets (like `'launch/'` and `'config/'`) into the package's share directory, so that parameter and launch files are visible and executable with `ros2 launch`

Important points:

- `entry_points['console_scripts']`: maps each command ¡name¿ 'module:function', so that `ros2 run` knows what to execute

Launch files - Running a launch file

1. To execute a launch file we use:

Run a launch file

```
ros2 launch <package_name> <launch_file_name>
<launch_arguments>
```

2. This will start all the nodes defined in the file, with the specified parameters and remappings.

```
[INFO] [launch]: All log files can be found below /home/lampros/.ros/log/2025-10-16-17-50-19-532405-Ubuntu2004Lampros-9748
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [luna_driver-1]: process started with pid [9750]
[INFO] [luna_commander-2]: process started with pid [9752]
[luna_driver-1] [INFO] [1760626220.305305951] [luna_driver]: Pose -> x: 0.00, y: 0.00,  theta: 0.00
[luna_driver-1] [INFO] [1760626220.475235388] [luna_driver]: Pose -> x: 0.00, y: 0.00,  theta: 0.00
[luna_commander-2] [INFO] [1760626220.597868508] [luna_commander]: Command: turn left at 0.50 rad/s
[luna_driver-1] [INFO] [1760626220.680939669] [luna_driver]: Pose -> x: 0.00, y: 0.00,  theta: 0.10
[luna_driver-1] [INFO] [1760626220.874652606] [luna_driver]: Pose -> x: 0.00, y: 0.00,  theta: 0.10
[luna_commander-2] [INFO] [1760626221.073236390] [luna_commander]: Command: forward at 0.20 m/s
```

Here we launch the nodes /luna_driver and /luna_commander with:
`ros2 launch luna_pkg luna_package_launch.py`

We could also launch like this to change some parameters:
`ros2 launch luna_pkg luna_package_launch.py period:=2.0 linear_speed:=10.0 angular_speed:=1.0`

Inspecting ROS2 Graph activity - Parameters

examples from Luna Rover Exercise

Output a list of available parameters

```
ros2 param list
```

```
/luna_commander:
  angular_speed
  linear_speed
  period
  topic_name
  use_sim_time
```

```
/luna_driver:
  cmd_topic
  pose_topic
  publish_period
  use_sim_time
```

Get parameter

```
ros2 param get /<node_name>
    <parameter_name>
```

```
ros2 param get /luna_driver
    publish_period
Double value is: 2.0
```

Set parameter

```
ros2 param set /<node_name>
  <parameter_name> <value>
```

```
ros2 param set /luna_commander
    linear_speed 0.7
Set parameter successful
```

Inspecting ROS2 Graph activity - Parameters

examples from Luna Rover Exercise

Dump the parameters of a node to a yaml file

`ros2 param dump /<node_name>`

```
ros2 param dump /luna_commander
    --output-dir config/
```

```
Saving to:  config/luna_commander.yaml
```

```
cat config/luna_commander.yaml
/luna_commander:
  ros__parameters:
    angular_speed: 0.5
    linear_speed: 0.2
    period: 2.0
    topic_name: cmd_vel
    use_sim_time: false
```

Load parameter file for a node

`ros2 param load /<node_name> <parameter_file>`

```
ros2 param load /luna_commander
    config/luna_commander.yaml
```

```
Set parameter angular_speed successful
Set parameter linear_speed successful
Set parameter period successful
Set parameter topic_name successful
Set parameter use_sim_time successful
```

Inspecting ROS2 Graph activity - Nodes

examples from Luna Rover Exercise

Output a list of available nodes

```
ros2 node list
```

```
/luna_commander
/luna_driver
```

Output information about a node

```
ros2 node info
  /<node_name>
```

`ros2 node info /luna_driver`

```
/luna_driver
  Subscribers:
    /cmd_vel: geometry_msgs/msg/Twist
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /pose: geometry_msgs/msg/Pose2D
    /rosout: rcl_interfaces/msg/Log
  Service Servers:
    /luna_driver/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /luna_driver/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /luna_driver/get_parameters: rcl_interfaces/srv/GetParameters
    /luna_driver/list_parameters: rcl_interfaces/srv/ListParameters
    /luna_driver/set_parameters: rcl_interfaces/srv/SetParameters
    /luna_driver/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
  Service Clients:

  Action Servers:

  Action Clients:
```

Inspecting ROS2 Graph activity - Topics

examples from Luna Rover Exercise

Output a list of available topics

```
ros2 topic list
```

```
/cmd_vel
/parameter_events
/pose
/rosout
```

Output information about a topic

```
ros2 topic info
 /<topic_name>
```

```
ros2 topic info /pose
Type: geometry_msgs/msg/Pose2D
Publisher count: 1
Subscription count: 0
```

Output messages from a topic

```
ros2 topic echo
 /<topic_name>
```

```
ros2 topic echo /pose
x: -10.983736057091209
y: -2.591432153770138
theta: 65.0
---
```

Inspecting ROS2 Graph activity - Topics

examples from Luna Rover Exercise

Publish a message to a topic

ros2 topic pub -r <Hz> /<topic_name> <message_type> <values>

```
publisher: beginning loop
publishing #1: geometry_msgs.msg.Twist(linear=geometry_msgs.msg.
Vector3(x=0.5, y=0.0, z=0.0), angular=geometry_msgs.msg.Vector3(
x=0.0, y=0.0, z=1.0))
```

```
ros2 topic pub -r 2 /cmd_vel geometry_msgs/msg/Twist "linear:  x:
  0.5, y:  0.0, z:  0.0, angular:  x:  0.0, y:  0.0, z:  1.0"
```

Print a topic's type

ros2 topic type /<topic_name>

```
ros2 topic type /cmd_vel
geometry_msgs/msg/Twist
```

Output a list of available topics of a given type

ros2 topic find <topic_type>

```
ros2 topic find
geometry_msgs/msg/Twist
/cmd_vel
```

Inspecting ROS2 Graph activity - Services

examples from Jerry Robot Exercise

Output a list of available services

```
ros2 service list
```

```
/distance
/jerry_robot_node/describe_parameters
/jerry_robot_node/get_parameter_types
/jerry_robot_node/get_parameters
/jerry_robot_node/list_parameters
/jerry_robot_node/set_parameters
/jerry_robot_node/set_parameters_atomically
```

Output a service's type

```
ros2 service type
  /<service_name>
```

```
ros2 service type /distance
jerry_pkg_interfaces/srv/DistanceFromObstacle
```

Output a list of available services of a given type

```
ros2 service find
  <service_type>
```

```
ros2 service find
jerry_pkg_interfaces/srv/DistanceFromObstacle
/distance
```

Inspecting ROS2 Graph activity - Services

examples from Jerry Robot Exercise

Call a service

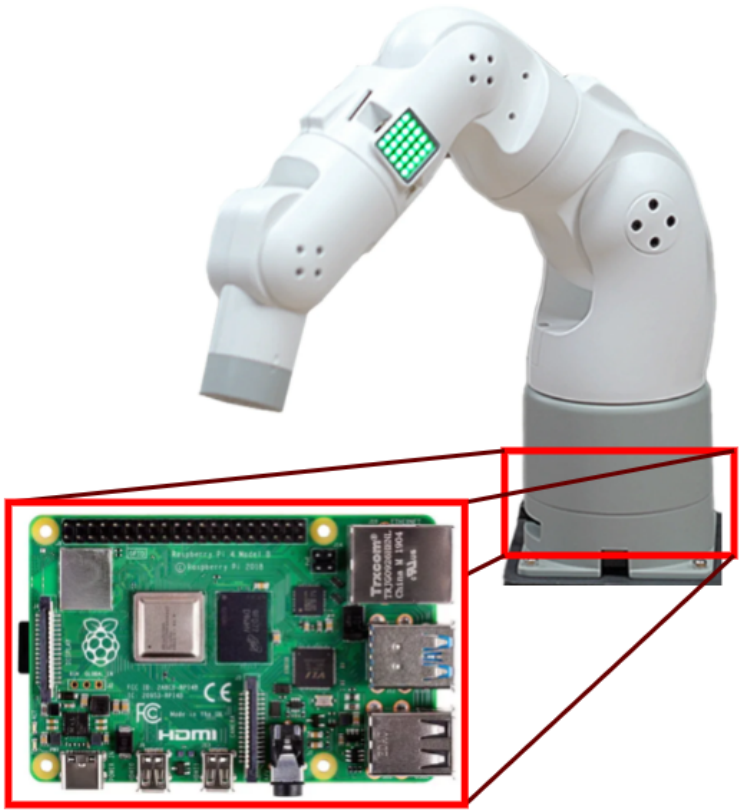
`ros2 service call -r <Hz> /<service_name> <service_type> <values>`

```
requester: making request: jerry_pkg_interfaces.srv.DistanceFromObstacle_Request(x=3.0, y=4.0)

response:
jerry_pkg_interfaces.srv.DistanceFromObstacle_Response(dist=14.422204971313477)

requester: making request: jerry_pkg_interfaces.srv.DistanceFromObstacle_Request(x=3.0, y=4.0)

response:
jerry_pkg_interfaces.srv.DistanceFromObstacle_Response(dist=12.041594505310059)

requester: making request: jerry_pkg_interfaces.srv.DistanceFromObstacle_Request(x=3.0, y=4.0)

response:
jerry_pkg_interfaces.srv.DistanceFromObstacle_Response(dist=9.848857879638672)
```

`ros2 service call -r 0.5 /distance jerry_pkg_interfaces/srv/DistanceFromObstacle "{x: 3.0, y: 4.0}"`

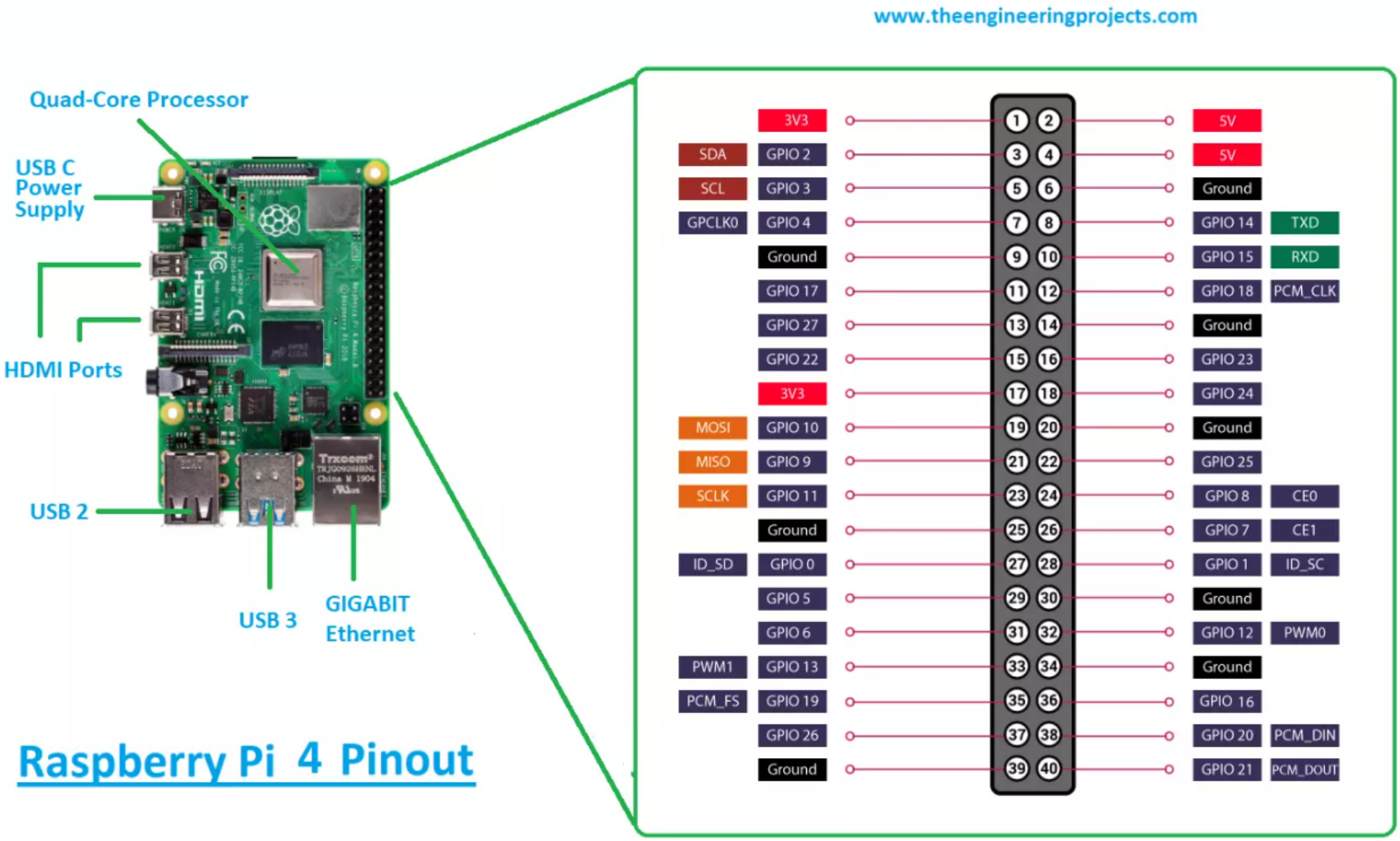
Raspberry Pi - RPi4 basics

1. **RPi4** (**Raspberry Pi 4**) is a powerful **dual-display single-board computer**, incorporated with a **quad-core processor**.
2. It can be used for **prototyping**, **programming**, and **IoT** projects. It is also widely used for **automation** and **robotics**.
3. To access the Raspberry Pi's **GPIO** (General-Purpose Input/Output) **pins** in Python, we can use the library **RPi.GPIO**.
4. We are going to use the RPi4 board that is integrated in **Elephant myArm 300 Pi 7-DOF robotic arm**.



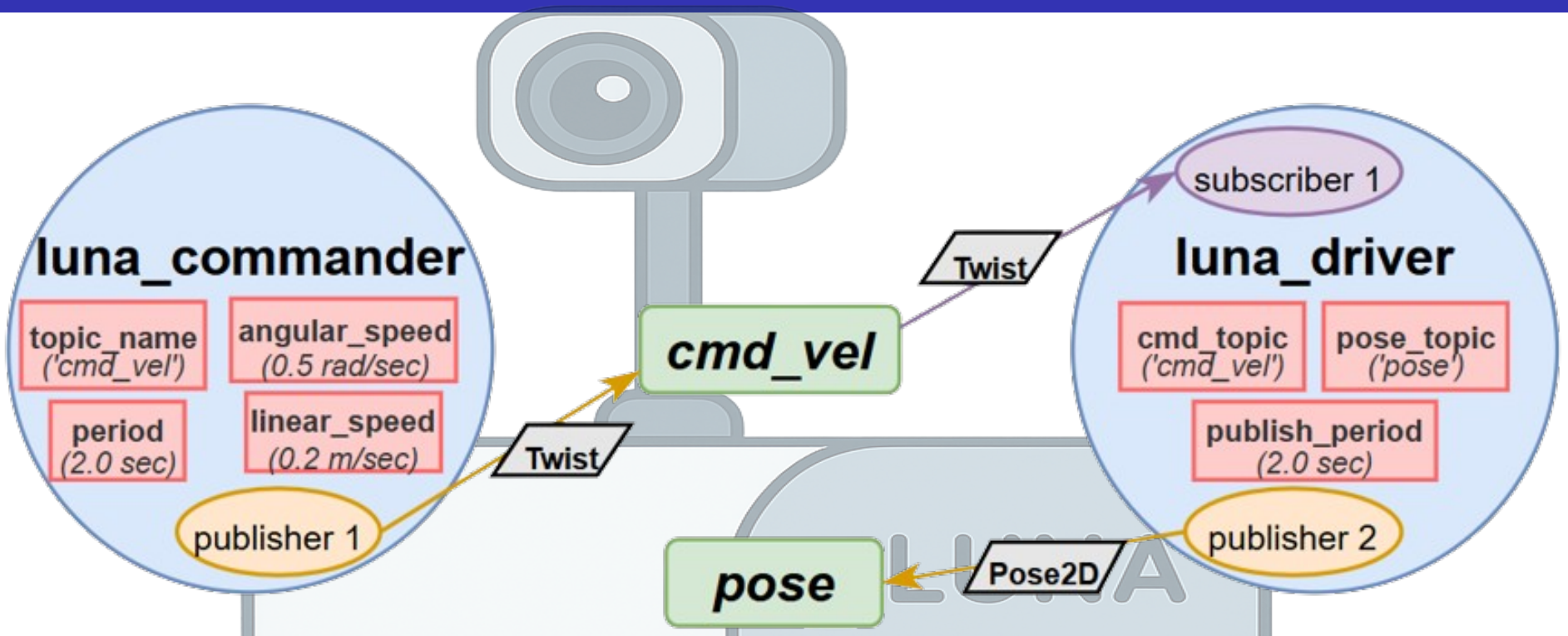
RPi4 inside Elephant myArm 300 Pi 7-DOF robotic arm

Raspberry Pi - RPi4 pinout diagram



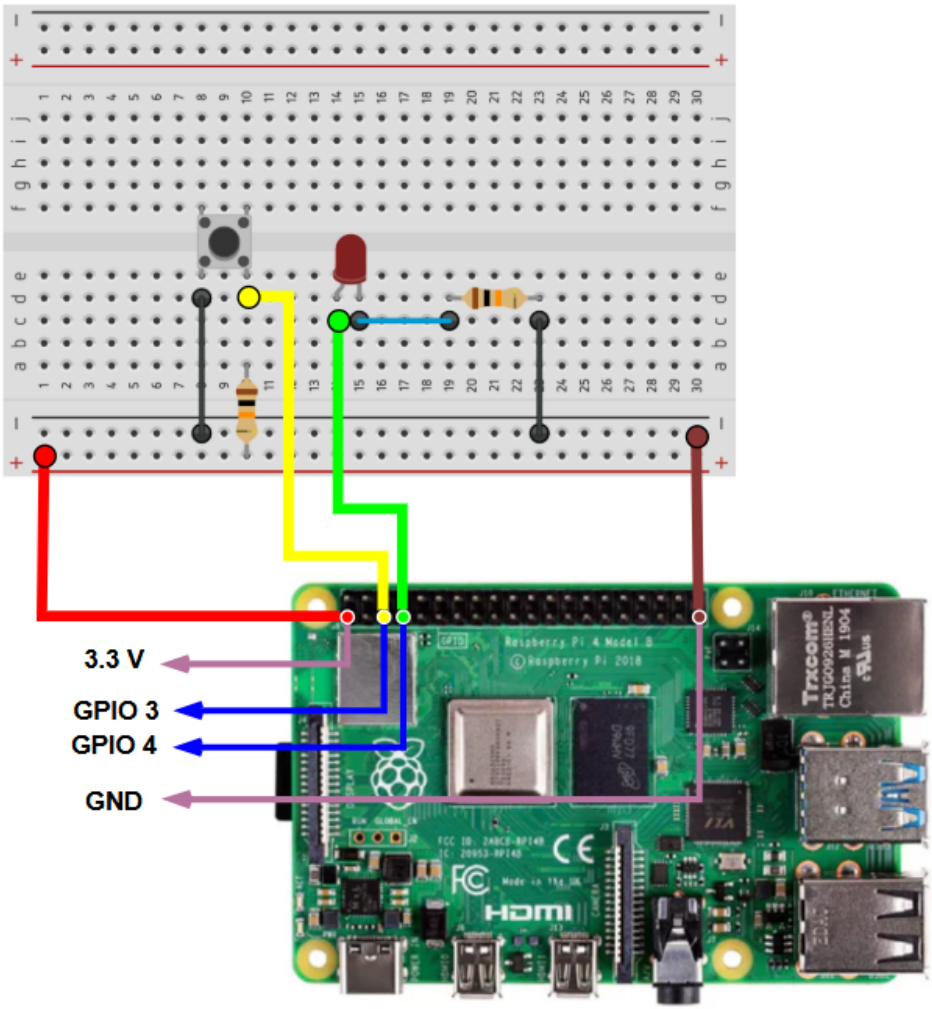
RPi4 pinout diagram

Luna the Rover Exercise



We are going to simulate a robot called ***Luna***, moving in 2D space. The `luna_commander` node randomly generates velocity commands and publishes them as *Twist* messages. The `luna_driver` node subscribes to these commands, integrates them over time, and updates the robot's 2D pose, publishing it continuously as *Pose2D* messages. We can see that each node owns some key-value pairs, which we call parameters.

LED Exercise



RPi4 - LED circuit scheme

Let's test our ROS2 skills! Our goal: control a simple LED. We need:

- 1 RPi4
- 1 breadboard
- 1 LED
- 2 resistors (e.g. 10 kΩ)
- 1 Button
- connecting wires

Important!

Be very careful **not to short the 3.3V and GND pins!** At best, this might cause the RPi4 to reboot, but in the worst case, it could damage the board!

LED Exercise - Part 1: Blink LED

1. First, we need to create a *publisher node* that sends **alternating 0 and 1 values** (our message) at a **fixed rate** of 1 second over a topic.
2. Then, we implement a *subscriber node* that listens to the same topic and triggers a **callback function to blink the LED** accordingly.

For the message type, you may use the `std_msgs` interface, or define your own!

Useful commands from RPi.GPIO Python library:

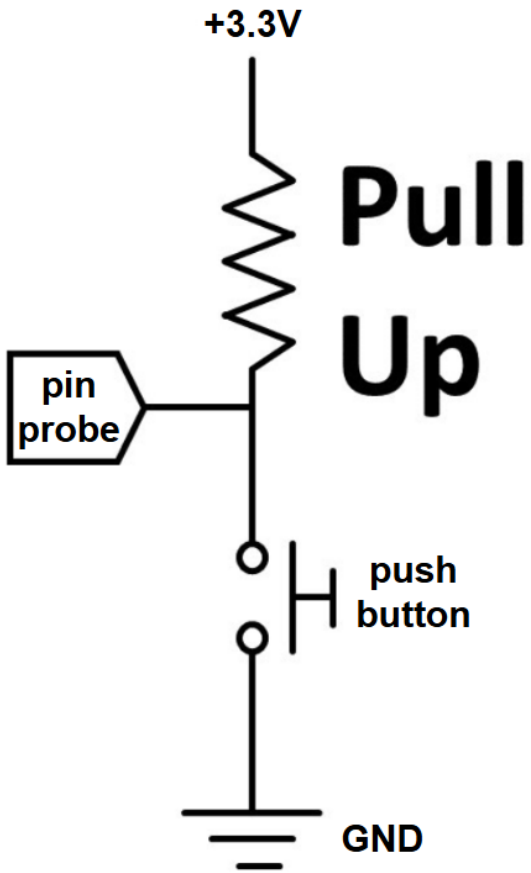
```
# use Broadcom (BCM) or Board (BOARD) pin numbering scheme
RPi.GPIO.setmode(RPi.GPIO.BCM)
# configure a GPIO pin as an output (OUT) or input (IN)
RPi.GPIO.setup(pin, RPi.GPIO.OUT)
# read a GPIO pin's value; it returns False (0) or True (1)
RPi.GPIO.input(pin)
# set a GPIO pin to value 0 (RPi.GPIO.LOW) or 1 (RPi.GPIO.HIGH)
RPi.GPIO.output(pin, value)
```

LED Exercise - Part 2: Change Blink Rate

1. For the second part, we have to modify the publisher node from part 1 to make the **blink period configurable at runtime**. This allows us to **change the LED blink frequency without restarting the node**.
2. So, we need to **declare a parameter for our timer's period**. When this parameter is updated, the timer should be updated accordingly. *Remember that, currently, our timer is only defined on startup!*
3. Once you're done with the modifications, **experiment with different timer periods** and **observe the results on your physical circuit**. To **update the parameter from the CLI** we can use the ROS2 functionalities we have learned. For example:
`ros2 param set /blink_led_publisher timer_period 0.5`.

LED Exercise - Part 3: Add Button

1. For the last part, we create a *second publisher node* that **reads a physical push button** connected to a GPIO pin.
2. The push button is connected using a **pull-up resistor configuration**. We want the LED:
   - **ON** when the button is **pressed**.
   - **OFF** when the button is **released**.
3. The new publisher node should **periodically publish the proper message** (according to the button state) over the same *topic* (and to the same *subscriber node*) as Part 1.

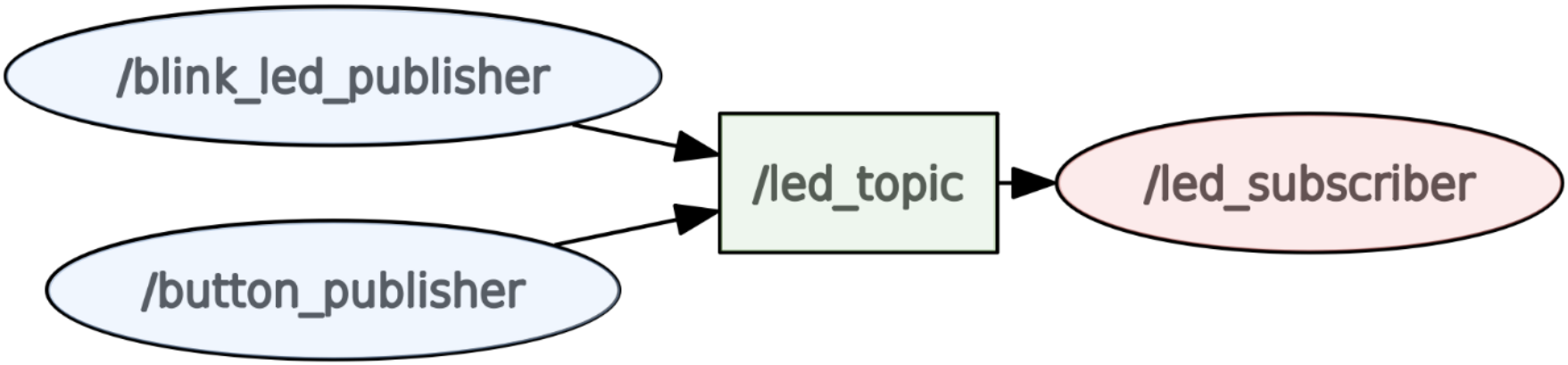


Push button wiring with pull-up resistor configuration

When the button is released, the pin probe is HIGH via the resistor.
When the button is pressed, the pin probe connects directly to GND and reads LOW.

LED Exercise - Part 4: Create Launch files

1. The **ROS2 graph** for the LED Exercise looks now like this:



2. We can create **2 launch files**, in order to not do `ros2 run` every time we want to run our nodes:
   - 1 for the blinking led (upper route of the ROS2 graph: `/blink_led_publisher` → `/led_topic` → `/led_subscriber`)
   - 1 for the button addition (lower route of the ROS2 graph: `/button_publisher` → `/led_topic` → `/led_subscriber`)

References

- **ROS Home**
- **ROS2 Documentation for all distros, here for Foxy**
- **Murilo's ROS2 Tutorial**
- **Kevin Wood ROS2 Youtube Tutorials**
- **Understanding Parameters in ROS2**
- **ROS2 YAML files for parameters**
- **Raspberry Pi 4 introduction and basics**
- **Control LED with RPi4 & ROS2: here_1 & here_2**
- **Elephant Robotics Documentation**