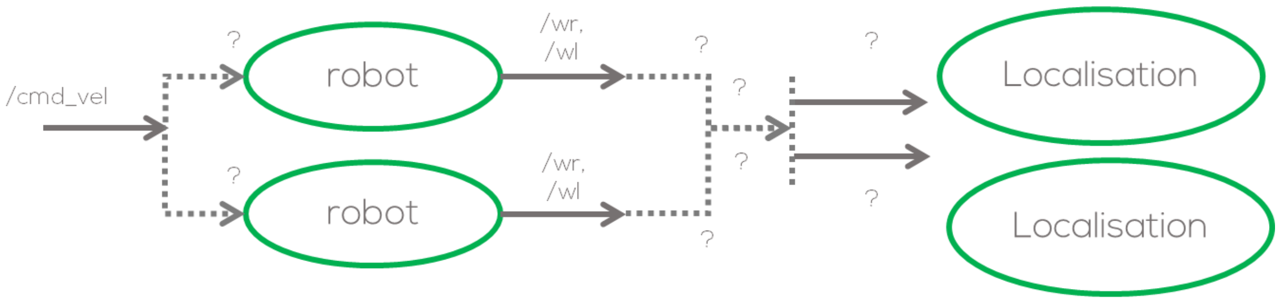
ROS2

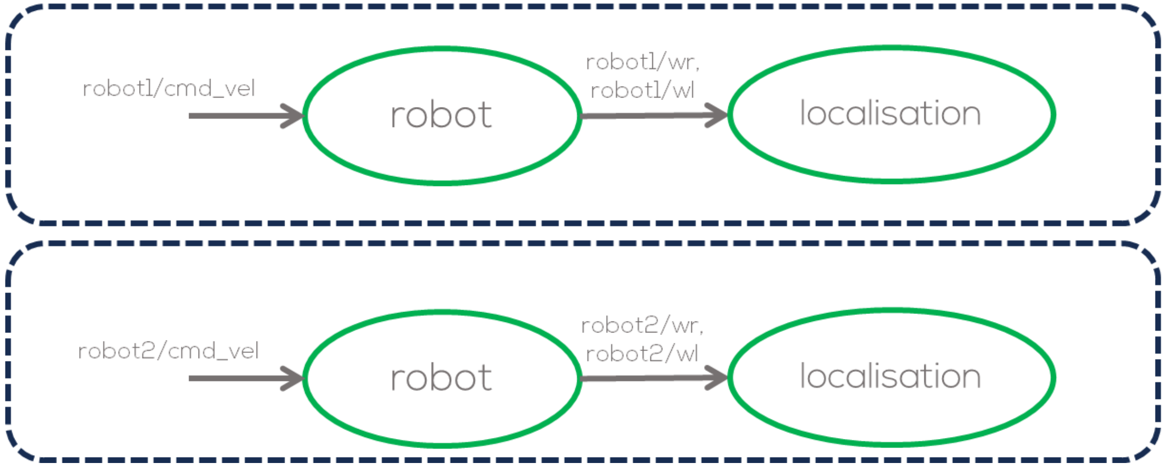
Multiple Systems

{Learn, Create, Innovate};

Introduction

- Imagine the following problem: you have a node that simulates a localisation node, and you require to simulate two (or more) robots with its respective localisation nodes using the same code.
- The problem in ROS will be the naming convention for the nodes, the topics and the names of the transforms (if used); since they will both be the same.
  - One simple solution will be to change the name of the nodes and topics manually by generating multiple .py files. For complex system this is not a good option. (What would happen if I require 10 motors?)
- Namespaces and parameters then become the best option to deal with name collisions, when systems become more complex.





Introduction

Activities 1 and 2

- Before we can simulate multiple robots, some basic concepts of namespaces and parameters must be addressed.
- To this end MCR2 has created a package called “motor_control” that will be used for Activities 1 and 2.

Activity 3

- Activity 3 will use the previous concepts to define a multiple robots using a simple simulation of the Puzzledrone and the concepts leart in Actities 1 and 2.
- Activity 3 will require the user to download the package “puzzle_drone”.

ROS2

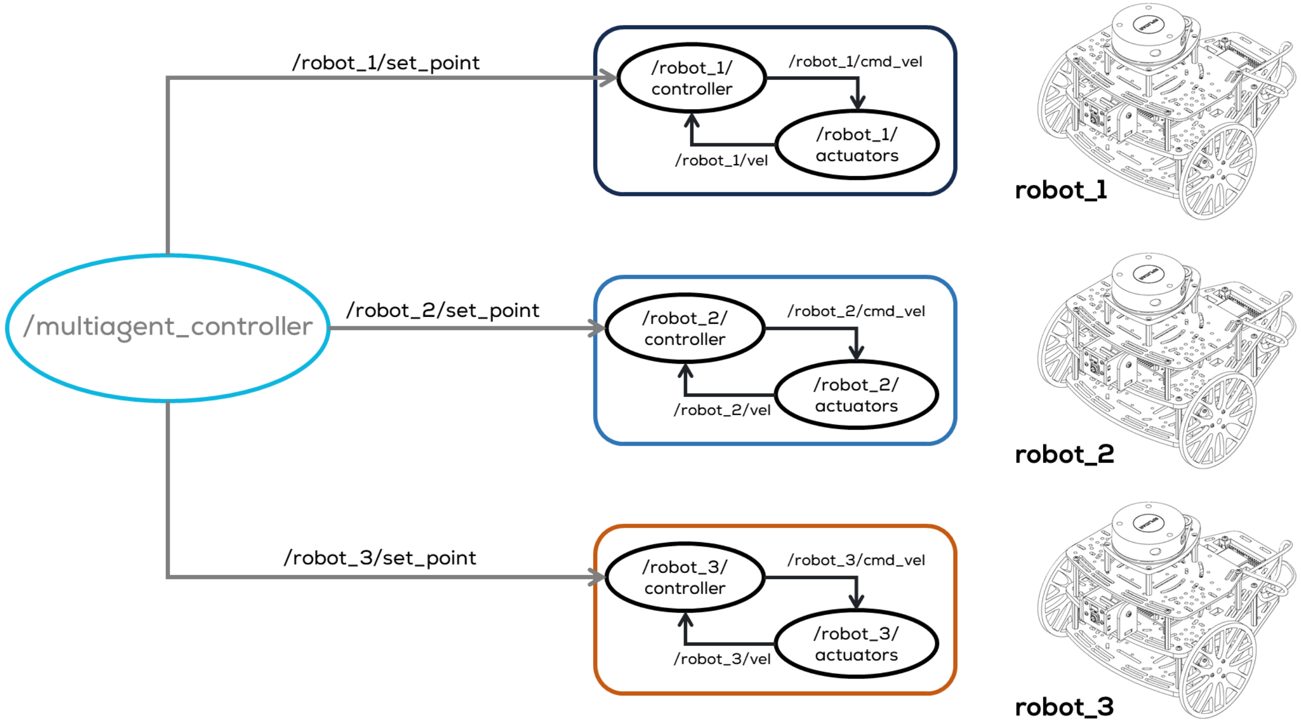
Namespaces

{Learn, Create, Innovate};

Manchester Robotics

ROS Namespaces

- A namespace in ROS can be viewed as a directory that contains items with different names.
- The items can be nodes, topics or other namespaces (hierarchy).
- There are several ways to define the namespaces. The easiest way is via the command line, which is very easy but not recommended for larger projects.
- The second way and the most used one is using, the launch file to define the namespaces.



Activity 1

ROS Namespaces

{Learn, Create, Innovate};

Manchester **Robotics**

Activity 1 – ROS Namespaces

Requirements

- You can download the motor_control template package from Github.

Objective

- The objective of this activity is to learn about namespaces.

Instructions

- Download the motor_control package from GitHub (inside Templates).
- Add it to your source directory inside your workspace

```
motor_control/
├── launch
│   └── motor_launch.py
├── LICENSE
├── motor_control
│   ├── dc_motor.py
│   ├── __init__.py
│   └── set_point.py
├── package.xml
├── resource
│   └── motor_control
├── setup.cfg
├── setup.py
└── test
    ├── test_copyright.py
    ├── test_flake8.py
    └── test_pep257.py
```

Activity 1 – ROS Namespaces

Instructions

- Compile the package using colcon

```
$ cd ~/ros2_ws
$ colcon build
$ source install/setup.bash
```

- Launch the package

```
$ ros2 launch motor_control motor_launch.py
```

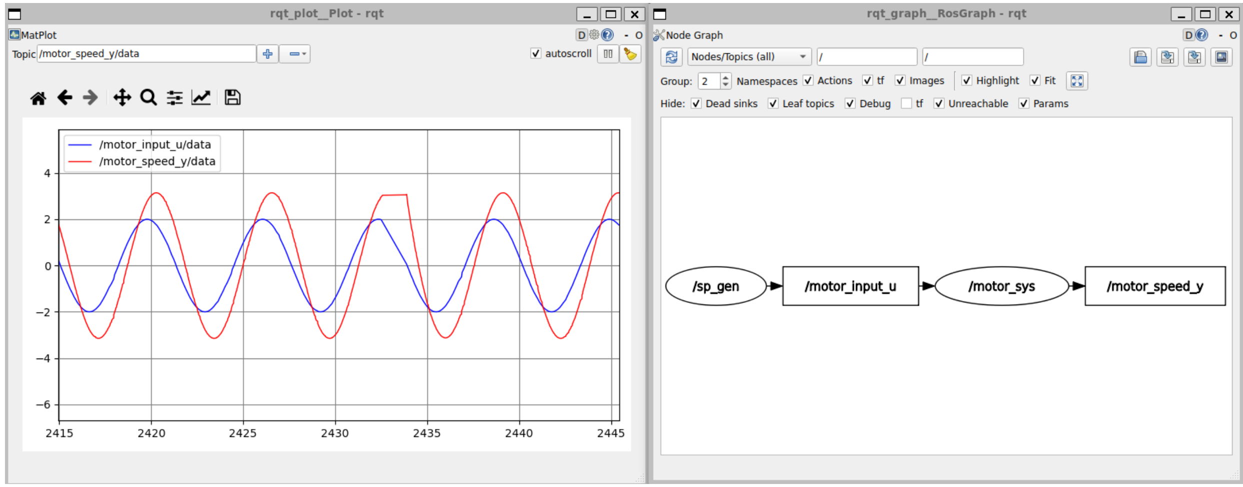
- Open two terminals run the rqt_graph and the rqt_plot

```
$ ros2 run rqt_plot rqt_plot
```

```
$ ros2 run rqt_graph rqt_graph
```

Results

- If everything goes well, you should see the following



- Check the published topics

```
mario@MarioPC:~$ ros2 topic list
/motor_input_u
/motor_speed_y
/parameter_events
/rosout
```

Activity 1 – ROS Namespaces

Motor Control package

- The package is composed of two nodes:
  - dc_motor node: Simulate a First Order System, representing a DC Motor.
  - set_point node: Providing an input for the system

```
motor_control/motor_control/dc_motor.py
motor_control/motor_control/set_point.py
```

- You can see the contents of each node by opening the file on any text editor (gedit, vscode, nano, vim, etc.)

DC Motor Node

- The DC Motor will be simulated using a First Oder system shown in here.

$$\tau \frac{dy(t)}{dt} + y(t) = Ku(t).$$

Where, τ is the time constant, K is the system gain, $y(t)$ is the system output (speed rad/s) and $u(t)$ the input signal (volts).

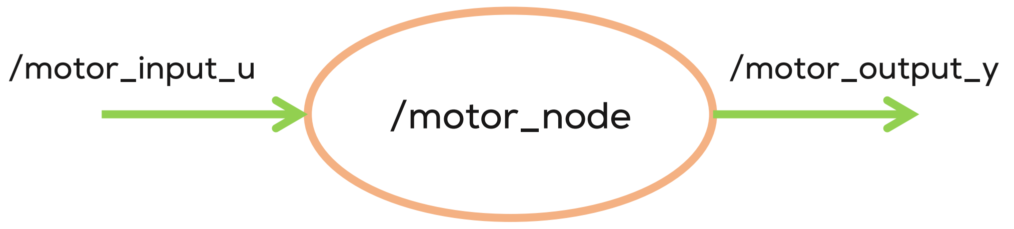
$$y[k+1] = y[k] + \left(-\frac{1}{\tau} \cdot y[k] + \frac{K}{\tau}\ u[k]\right) T_s$$

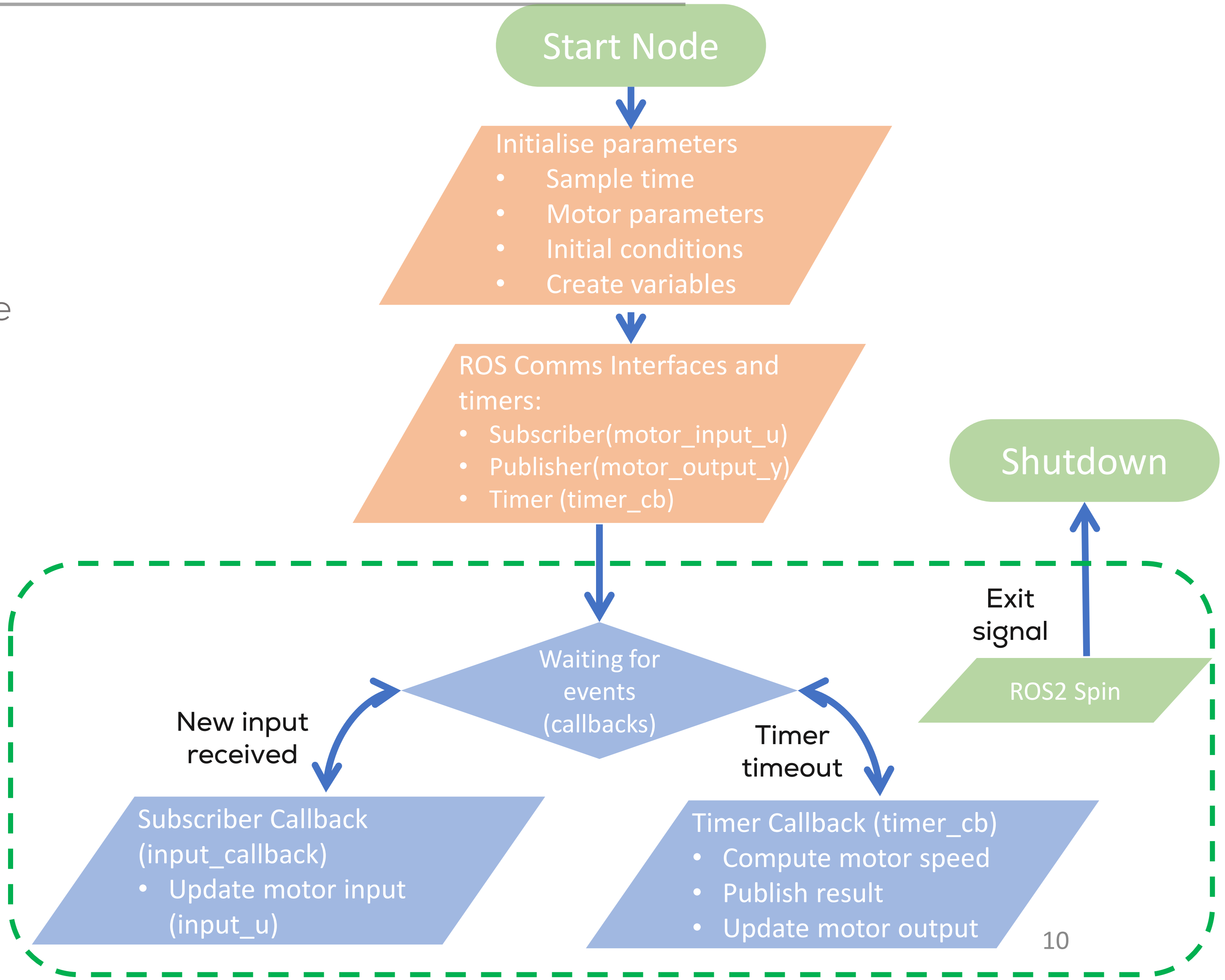
Where T_s is the sampling time.

Activity 1 – ROS Namespaces

DC Motor Node Structure

- The node subscribes to the topic "/motor_input_u" and publishes the vales of the motor speed on the topic "/motor_output_y".
- Both topics contain an interface (message) Float32





dc_motor.py

Libraries

```
# Imports
import rclpy
from rclpy.node import Node
from std_msgs.msg import Float32
```

Initialise parameters

```
#Class Definition
class DCMotor(Node):
    def __init__(self):
        super().__init__('dc_motor')

        # DC Motor Parameters
        self.sample_time = 0.02
        self.param_K = 1.75
        self.param_T = 0.5
        self.initial_conditions = 0.0

        #Set the messages
        self.motor_output_msg = Float32()

        #Set variables to be used
        self.input_u = 0.0
        self.output_y = self.initial_conditions
```

ROS publishers, subscribers Timers

```
        #Declare publishers, subscribers and timers
        self.motor_input_sub = self.create_subscription(Float32, 'motor_input_u', 
self.input_callback,10)
        self.motor_speed_pub = self.create_publisher(Float32, 'motor_speed_y', 10)
        self.timer = self.create_timer(self.sample_time, self.timer_cb)

        #Node Started
        self.get_logger().info('Dynamical System Node Started \U0001F680')
```

Timer callback: Motor simulation

```
    #Timer Callback
    def timer_cb(self):
        #DC Motor Simulation
        #DC Motor Equation y[k+1] = y[k] + ((-1/τ) y[k] + (K/τ) 
u[k]) T_s
        self.output_y += (-1.0/self.param_T * self.output_y + 
self.param_K/self.param_T * self.input_u) * self.sample_time
        #Publish the result
        self.motor_output_msg.data = self.output_y
        self.motor_speed_pub.publish(self.motor_output_msg)
```

Subscriber callback

```
    #Subscriber Callback
    def input_callback(self, input_sgn):
        self.input_u = input_sgn.data
```

Main

```
#Main
def main(args=None):
    rclpy.init(args=args)

    node = DCMotor()

    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.try_shutdown()

#Execute Node
if __name__ == '__main__':
    main()
```

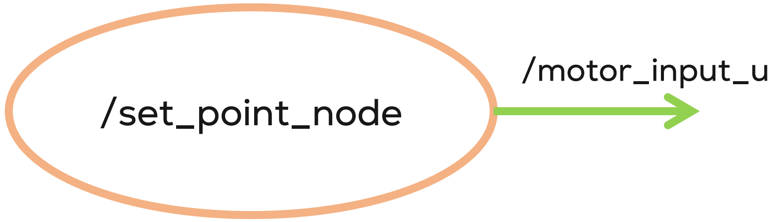
Activity 1 – ROS Namespaces

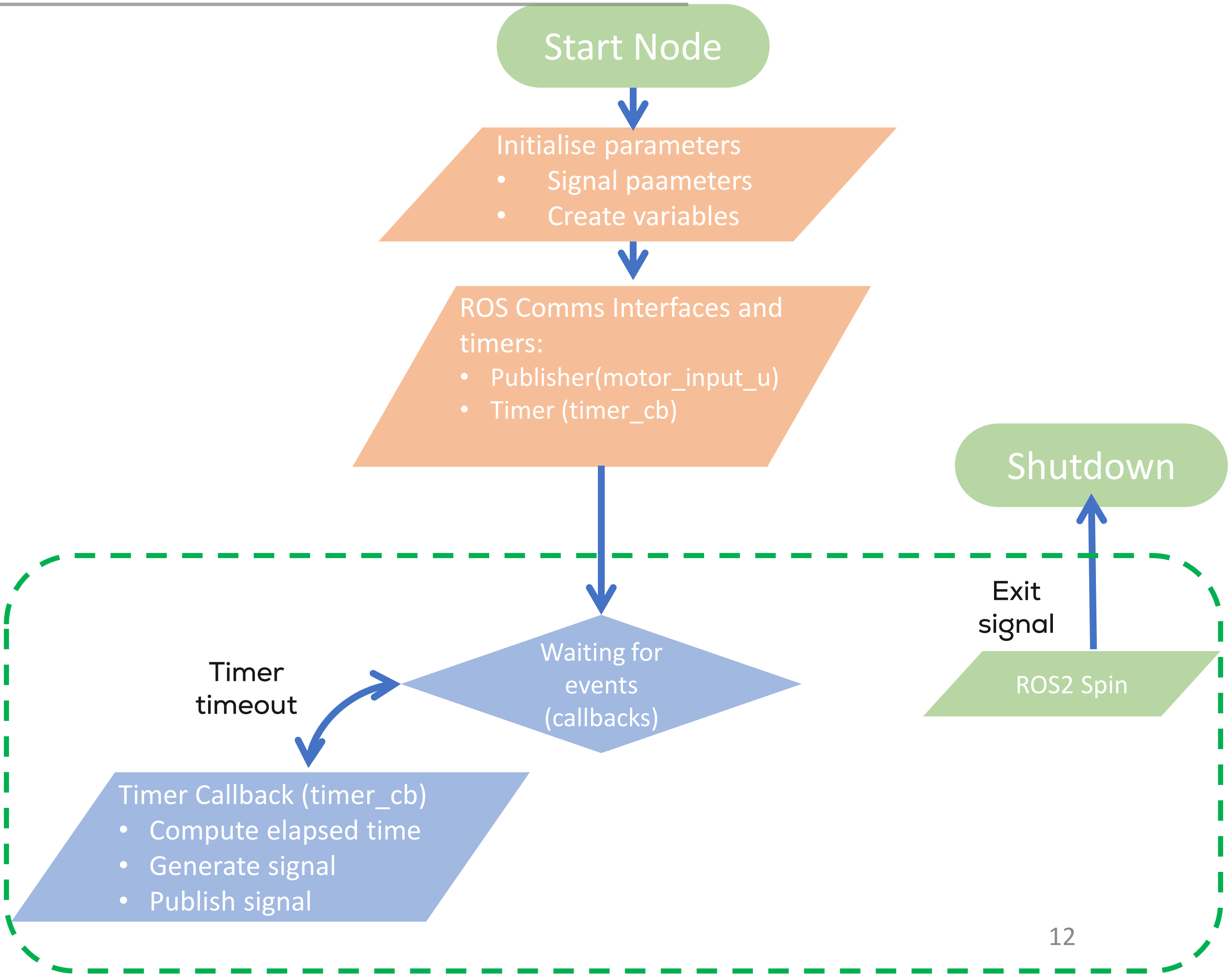
Set Point node structure

- The node publishes the vales of input signal on the topic “/motor_input_u”.

$$u(t) = A\,sin(\omega t)$$

- The topic contain an interface (message) Float32





set_point.py

```python
# Imports
import rclpy
from rclpy.node import Node
import numpy as np
from std_msgs.msg import Float32

#Class Definition
class SetPointPublisher(Node):
    def __init__(self):
        super().__init__('set_point_node')

        # Retrieve sine wave parameters
        self.amplitude = 2.0
        self.omega  = 1.0

        #Create a publisher and timer for the signal
        self.signal_publisher = self.create_publisher(Float32,
'motor_input_u', 10)
        timer_period = 0.1 #seconds
        self.timer = self.create_timer(timer_period, self.timer_cb)

        #Create a messages and variables to be used
        self.signal_msg = Float32()
        self.start_time = self.get_clock().now()

        self.get_logger().info("SetPoint Node Started \U0001F680")
```

Libraries

Initialise parameters

ROS publishers, subscribers Timers

```python
    # Timer Callback: Generate and Publish Sine Wave Signal
    def timer_cb(self):
        #Calculate elapsed time
        elapsed_time = (self.get_clock().now() -
self.start_time).nanoseconds/1e9
        # Generate sine wave signal
        self.signal_msg.data = self.amplitude *
np.sin(self.omega * elapsed_time)
        # Publish the signal
        self.signal_publisher.publish(self.signal_msg)

#Main
def main(args=None):
    rclpy.init(args=args)

    set_point = SetPointPublisher()

    try:
        rclpy.spin(set_point)
    except KeyboardInterrupt:
        pass
    finally:
        set_point.destroy_node()
        rclpy.try_shutdown()

#Execute Node
if __name__ == '__main__':
    main()
```

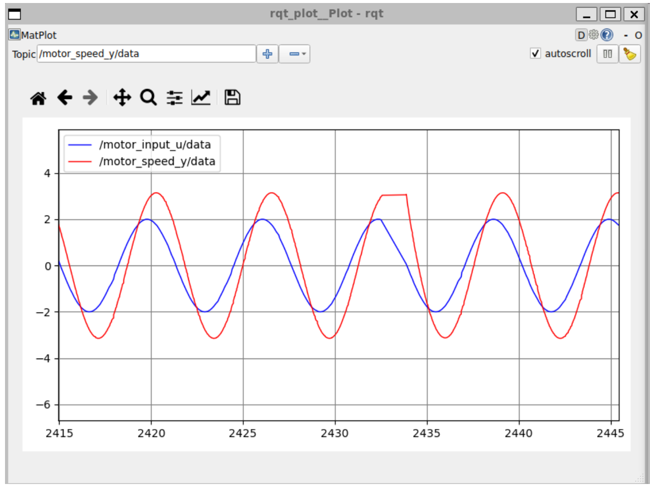
Timer callback: Signal Generator

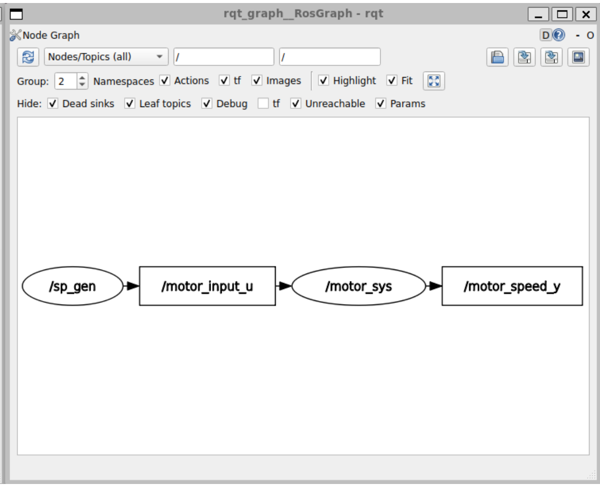
Main

motor_launch.py

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    motor_node = Node(name="motor_sys",
                      package='motor_control',
                      executable='dc_motor',
                      emulate_tty=True,
                      output='screen',
                      )

    sp_node = Node(name="sp_gen",
                      package='motor_control',
                      executable='set_point',
                      emulate_tty=True,
                      output='screen',
                      )

    l_d = LaunchDescription([motor_node, sp_node])

    return l_d
```

- The launch file starts a motor_node and a set_point node.





Activity 1 – ROS Namespaces

Adding a namespace

- Create an *motor2_launch.py* file in the launch folder of the *motor_control* package.

```
$ cd ~/ros2_ws/src/motor_control/launch
$ touch motor_2_launch.py
$ chmod +x motor_2_launch.py
```

- Open the motor_2_launch.py using a text editor.
- Copy the following code (next slide)

Folder Tree

```
motor_control/
├── launch
│   ├── motor_2_launch.py
│   └── motor_launch.py
├── LICENSE
├── motor_control
│   ├── dc_motor.py
│   ├── __init__.py
│   └── set_point.py
├── package.xml
├── resource
│   └── motor_control
├── setup.cfg
├── setup.py
└── test
    ├── test_copyright.py
    ├── test_flake8.py
    └── test_pep257.py
```

Activity 1 – ROS Namespaces

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    motor_node_1 = Node(name="motor_sys_1",
                        package='motor_control',
                        executable='dc_motor',
                        emulate_tty=True,
                        output='screen',
                        namespace="group1"
                        )

    sp_node_1 = Node(name="sp_gen_1",
                        package='motor_control',
                        executable='set_point',
                        emulate_tty=True,
                        output='screen',
                        namespace="group1"
                        )
```

Imports

Launch body

```python
    motor_node_2 = Node(name="motor_sys_2",
                        package='motor_control',
                        executable='dc_motor',
                        emulate_tty=True,
                        output='screen',
                        namespace="group2"
                        )

    sp_node_2 = Node(name="sp_gen_2",
                        package='motor_control',
                        executable='set_point',
                        emulate_tty=True,
                        output='screen',
                        namespace="group2"
                        )

    l_d = LaunchDescription([motor_node_1, sp_node_1,
motor_node_2, sp_node_2])

    return l_d
```

Launch body

Set launch content

Activity 1 – ROS Namespaces

- Build and run the newly created lunch file using colcon.

```
$ cd ~/ros2_ws
$ colcon build
$ source install/setup.bash
$ ros2 launch motor_control motor_2_launch.py
```

- Open the *rqt_graph to visualise the nodes*

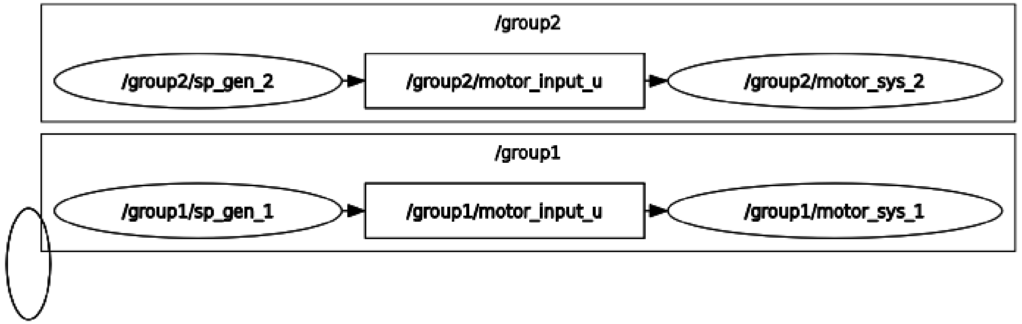
```
$ ros2 run  rqt_graph rqt_graph
```

Tips

Add the rqt_graph to the launch file:

```
rqt_graph_node = Node(name="rqt",
                      package='rqt_graph',
                      executable='rqt_graph',
                      output='screen'
                       )

l_d = LaunchDescription([motor_node_1, sp_node_1,
motor_node_2, sp_node_2, rqt_graph_node])
```

Results




```
mario@MarioPC:~$ ros2 topic list
/group1/motor_input_u
/group1/motor_speed_y
/group2/motor_input_u
/group2/motor_speed_y
/parameter_events
/rosout
```

ROS2

Parameters

{Learn, Create, Innovate};

ROS Parameters

- Any software application, especially in robotics requires parameters.
- Parameters are variables with some predefined values that are stored in a separate file or hardcoded in a program such that the user has easy access to change their value.
- At the same time parameters can be shared amongst different programs to avoid rewriting them or recompiling the nodes (C++)
- In robotics, parameters are used to store values requiring tunning, robot names, sampling times or flags.
- ROS encourage the usage of parameters to avoid making dependencies or rewriting nodes.

ROS Parameters

- ROS parameters are stored in each node.
- Nodes retrieve parameters at startup and runtime.
- The lifetime of a parameter is the same as the node.
- These parameters are used to configure nodes, e.g., robot constants, starting values, controller parameters, etc.
- ROS can only use determined types of parameters such as:

```
bool, int64, float64, string, byte[], bool[], int64[],
float64[] or string[]
```

- Parameters are composed of a key, value and descriptor.

```
key      value        descriptor
<Name> <Value>      <Description of the parameter (empty)>
```

localisation_node

params:
robot_name: Robot_1
max_speed = 1.0
Waypoints =[P1, P2]

Activity 2

Launch File Parameters

{Learn, Create, Innovate};

Activity 2 – Launch File Parameters

Requirements

- motor_control ROS2 package.

Objective

- The objective is to add parameters to the motor_control package.

Instructions

- Open the package motor_control or the file "dc_motor.py" on a text editor.

```
$ cd ~/ros2_ws/src/motor_control
$ code .    (for vscode)
```

- Normally parameters are hardcoded as shown. Sometimes is difficult to access them when they are not organised (like in the example).

```
# DC Motor Parameters
self.sample_time = 0.02
self.param_K = 1.75
self.param_T = 0.5
self.initial_conditions = 0.0
```

Activity 2 – Launch File Parameters

Instructions

- In this exercise those parameters will be set from the launch file, to allow the user change them without needing to open the code to change them.

```
# DC Motor Parameters
#Change them to ROS2 Parameters
self.sample_time = 0.02
self.param_K = 1.75
self.param_T = 0.5
self.initial_conditions = 0.0
```

Declaring a parameter

- A parameter can be declared inside a script as follows.

```
self.declare_parameter('sample_time', 0.02)
```

Name

Initial Value

- To get the value of the parameter can be done as follows.

```
self.sample_time = self.get_parameter('sample_time').value
```

Variable to store the value

Name of the parameter

Value

Activity 2 – Launch File Parameters

Instructions

- Declare the following parameters in your code inside our constructor.

```
# Declare parameters
# System sample time in seconds
self.declare_parameter('sample_time', 0.02)
# System gain K
self.declare_parameter('sys_gain_K', 1.75)
# System time constant Tau
self.declare_parameter('sys_tau_T', 0.5)
# System initial conditions
self.declare_parameter('initial_conditions', 0.0)
```

Instructions

- A Set the variables to be used with the parameter values.

```
# DC Motor Parameters
self.sample_time = self.get_parameter('sample_time').value
self.param_K = self.get_parameter('sys_gain_K').value
self.param_T = self.get_parameter('sys_tau_T').value
self.initial_conditions =
self.get_parameter('initial_conditions').value
```

dc_motor.py

```python
# Imports
import rclpy
from rclpy.node import Node
from std_msgs.msg import Float32

#Class Definition
class DCMotor(Node):
    def __init__(self):
        super().__init__('dc_motor')

        # Declare parameters
        # System sample time in seconds
        self.declare_parameter('sample_time', 0.02)
        # System gain K
        self.declare_parameter('sys_gain_K', 1.75)
        # System time constant Tau
        self.declare_parameter('sys_tau_T', 0.5)
        # System initial conditions
        self.declare_parameter('initial_conditions', 0.0)

        # DC Motor Parameters
        self.sample_time = self.get_parameter('sample_time').value
        self.param_K = self.get_parameter('sys_gain_K').value
        self.param_T = self.get_parameter('sys_tau_T').value
        self.initial_conditions = self.get_parameter('initial_conditions').value


        . . .
```

Libraries

Declare parameters

Code Continues

- The code should look like the one on the left.
- Open the launch file motor_launch.py.
- Add the parameters to the *motor_node*

```python
motor_node = Node(name="motor_sys",
                  package='motor_control',
                  executable='dc_motor',
                  emulate_tty=True,
                  output='screen',
                  parameters=[{
                       'sample_time': 0.02,
                       'sys_gain_K': 1.75,
                       'sys_tau_T': 0.5,
                       'initial_conditions': 0.0,
                       }
                   ]
                  )
```

Activity 2 – Launch File Parameters

Instructions

- Save and compile the file

```
$ cd ~/ros2_ws
$ colcon build
$ source install/setup.bash
```

- Launch the node

```
$ ros2 launch motor_control motor_launch.py
```

- Verify the new parameters on terminal

```
$ ros2 param list
```

Results

```
mario@MarioPC:~$ ros2 param list
/motor_sys:
  initial_conditions
  sample_time
  start_type_description_service
  sys_gain_K
  sys_tau_T
  use_sim_time
```

```
$ ros2 param get /motor_sys sys_gain_K
```

```
mario@MarioPC:~$ ros2 param get /motor_sys sys_gain_K
Double value is: 1.75
```

- To change a parameter, you must change it on the launch file and re-build the package using colcon build.

ROS Parameters

Parameters Command Line

- To list the parameters belonging to available nodes

```
$ ros2 param list
```

- To display the type and current value of a

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

- To change a parameter's value at runtime (current session)

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

- Dump all of a node's current parameter values into a file to save them

```
$ ros2 param dump <node_name>
```

- You can load parameters from a file to a currently running node

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

- To start the same node using your saved parameter values

```
$ ros2 run <package_name> <executable_name> --
ros-args --params-file <file_name>
```

Activity 3

Multiple Robots

{Learn, Create, Innovate};

Manchester Robotics

Activity 3 – Multiple Robots

Requirements

- “puzzle_drone” ROS2 package.

Objective

- The objective is to simulate multiple robots in ROS2.

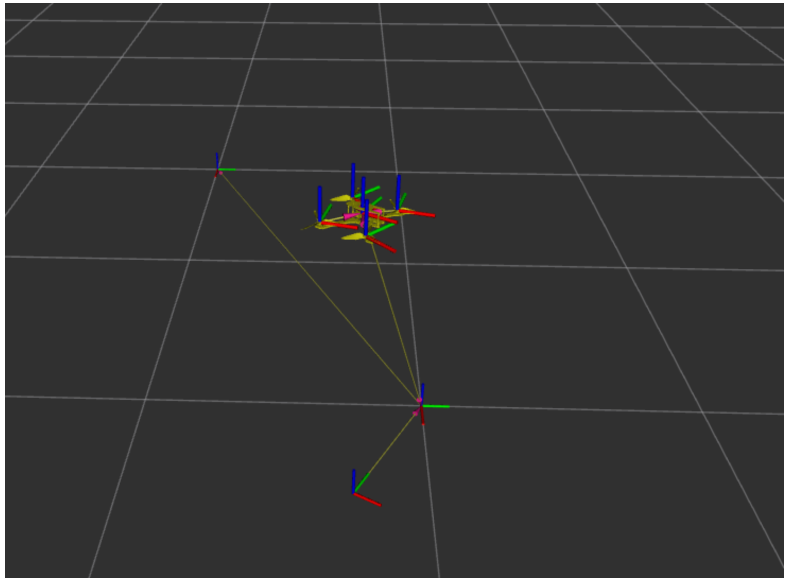
Instructions

- Download the activity template package “puzzle_drone” from GitHub.

Instructions

- Compile the package and launch the node

```
$ cd ~/ros2_ws
$ colcon build --packages-select puzzle_drone
$ source install/setup.bash
$ ros2 launch  puzzle_drone puzzledrone_launch.py
```



Activity 3 – Multiple Robots

```
import rclpy
from rclpy.node import Node
from tf2_ros import TransformBroadcaster
from geometry_msgs.msg import TransformStamped
from sensor_msgs.msg import JointState
import transforms3d
import numpy as np

class DronePublisher(Node):

    def __init__(self):
        super().__init__('puzzledorone_joint_pub')

        #Drone Initial Pose
        self.intial_pos_x = 0.0
        self.intial_pos_y = 0.0
        self.intial_pos_z = 0.0
        self.intial_pos_yaw = 0.0
        self.intial_pos_pitch = 0.0
        self.intial_pos_roll = 0.0
```

Libraries

Declare parameters

Parametrising the Joint State Pub

- In the package "puzzle_drone" open the file "puzzledrone_joint_state_pub.py" on a text editor.

```
$ cd ~/ros2_ws/src/ puzzle_drone
$ code .     (for vscode)
```

- Normally parameters are hardcoded as shown. Sometimes is difficult to access them when they are not organised (like in the example).

Activity 3 – Multiple Robots

Parametrising the Joint State Pub

- Declare the following parameters in your code inside our constructor.

```
self.declare_parameter('init_pose_x', 0.0)
self.declare_parameter('init_pose_y', 0.0)
self.declare_parameter('init_pose_z', 1.0)
self.declare_parameter('init_pose_yaw', np.pi/2)
self.declare_parameter('init_pose_pitch', 0.0)
self.declare_parameter('init_pose_roll', 0.0)
self.declare_parameter('odom_frame', 'odom')
```

Parametrising the Joint State Pub

- A Set the variables to be used with the parameter values.

```
# Retrieve the parameter value
self.odom_frame =
self.get_parameter('odom_frame').get_parameter_value().string_value.
strip('/’)

#Drone Initial Pose
self.intial_pos_x = self.get_parameter('init_pose_x').value
self.intial_pos_y = self.get_parameter('init_pose_y').value
self.intial_pos_z = self.get_parameter('init_pose_z').value
self.intial_pos_yaw = self.get_parameter('init_pose_yaw').value
self.intial_pos_pitch = self.get_parameter('init_pose_pitch').value
self.intial_pos_roll = self.get_parameter('init_pose_roll').value
```

puzzledrone_joint_state_pub.py

```python
import rclpy
from rclpy.node import Node
from tf2_ros import TransformBroadcaster
from geometry_msgs.msg import TransformStamped
from sensor_msgs.msg import JointState
import transforms3d
import numpy as np

class DronePublisher(Node):

    def __init__(self):
        super().__init__('puzzledorone_joint_pub')

        self.namespace = self.get_namespace().rstrip('/')

        # Declare the parameter with a default value
        self.declare_parameter('init_pose_x', 0.0)
        self.declare_parameter('init_pose_y', 0.0)
        self.declare_parameter('init_pose_z', 1.0)
        self.declare_parameter('init_pose_yaw', np.pi/2)
        self.declare_parameter('init_pose_pitch', 0.0)
        self.declare_parameter('init_pose_roll', 0.0)
        self.declare_parameter('odom_frame', 'odom')

        # Retrieve the parameter value
        self.odom_frame =
self.get_parameter('odom_frame').get_parameter_value().string_value.strip('/')

        #Drone Initial Pose
        self.intial_pos_x = self.get_parameter('init_pose_x').value
        self.intial_pos_y = self.get_parameter('init_pose_y').value
        self.intial_pos_z = self.get_parameter('init_pose_z').value
        self.intial_pos_yaw = self.get_parameter('init_pose_yaw').value
        self.intial_pos_pitch = self.get_parameter('init_pose_pitch').value
        self.intial_pos_roll = self.get_parameter('init_pose_roll').value

    . . .
```

Libraries

Declare parameters

Code Continues

Parametrising the Joint State Pub

- The code should look like the one on the left.
- Open the launch file puzzledrone_launch.py.
- Add the parameters to the puzzledrone_node.

```python
puzzledrone_node = Node(name="puzzledrone_joint_pub",
                        package='puzzle_drone',
                        executable='puzzledrone_joint_state_pub',
                        parameters=[{
                            'init_pose_x':0.0,
                            'init_pose_y': 0.0,
                            'init_pose_z': 1.0,
                            'init_pose_yaw': 1.57,
                            'init_pose_pitch': 0.0,
                            'init_pose_roll': 0.0,
                            'odom_frame':'odom'
                        }]
                        )
```

puzzledrone_joint_state_pub.py

- The code should look like the one on the left.
- Change the header frame for the parameter “self.odom_frame”
- This will allow the user to select the name of the parent frame of the transform. In this case “odom”.

```
def define_TF(self):

#Create Trasnform Messages
self.base_footprint_tf = TransformStamped()
self.base_footprint_tf.header.stamp = self.get_clock().now().to_msg()
self.base_footprint_tf.header.frame_id = 'odom'
self.base_footprint_tf.child_frame_id = 'base_footprint’
. . .


#Create Trasnform Messages
self.base_link_tf = TransformStamped()
self.base_link_tf.header.stamp = self.get_clock().now().to_msg()
self.base_link_tf.header.frame_id = 'odom'
self.base_link_tf.child_frame_id = 'base_link'
. . .
```

```
def define_TF(self):

#Create Trasnform Messages
self.base_footprint_tf = TransformStamped()
self.base_footprint_tf.header.stamp = self.get_clock().now().to_msg()
self.base_footprint_tf.header.frame_id = self.odom_frame
self.base_footprint_tf.child_frame_id = 'base_footprint’
. . .


#Create Trasnform Messages
self.base_link_tf = TransformStamped()
self.base_link_tf.header.stamp = self.get_clock().now().to_msg()
self.base_link_tf.header.frame_id = self.odom_frame
self.base_link_tf.child_frame_id = 'base_link'
. . .
```

Transforms and Namespaces

- Namespaces in ROS 2 are only for topics, services, and parameters.
- TF2 does not inherently use a node's namespace when broadcasting transforms.
- The "child_frame_id" and "frame_id" must be manually prefixed to avoid conflicts.
- To do this, the namespace of the node should manually be added to the Transform (this can be automatically added).

```
self.base_footprint_tf.child_frame_id = namespace/base_footprint"
. . .
```

- In the case of our program, to know the namespace of the node, at the constructor of the class the following will be defined:

```
def __init__(self):
    super().__init__('puzzledorone_joint_pub')
    self.namespace = self.get_namespace().rstrip('/')
. . .
```

- Then this is replaced in both transforms.

```
self.base_footprint_tf.child_frame_id = f"{self.namespace}/base_footprint"
. . .

self.base_link_tf.child_frame_id = f"{self.namespace}/base_link"
. . .
```

Transforms and Namespaces

```python
class DronePublisher(Node):

    def __init__(self):
        super().__init__('puzzledorone_joint_pub')
        self.namespace = self.get_namespace().rstrip('/’)

      ...

      def define_TF(self):

            #Create Trasnform Messages
            self.base_footprint_tf = TransformStamped()
            self.base_footprint_tf.header.stamp = self.get_clock().now().to_msg()
            self.base_footprint_tf.header.frame_id = self.odom_frame
            self.base_footprint_tf.child_frame_id = f"{self.namespace}/base_footprint"

            ...

            #Create Trasnform Messages
            self.base_link_tf = TransformStamped()
            self.base_link_tf.header.stamp = self.get_clock().now().to_msg()
            self.base_link_tf.header.frame_id = self.odom_frame
            self.base_link_tf.child_frame_id = f"{self.namespace}/base_link"

            ...
```

Launch Files

- Open the "multi_puzzledrone_launch.py"
- Add two groups of robots. In other words two of each node to be used and give them a namespace. In this case "group1" and "group2" will be used as namespaces.
- If Using URDF (like in this example) the robot state publisher requires the parameter "frame_prefix" to add a namespace to each transform.
- The value of the parameter must always be "{namespace}/" where {namespace} must be replaced by the user's namespace and always contain the backslash.
- Group 2 should be added the same way, just replacing the namespace to be "group2".

```
# Robot 1: group1

    robot1_state_pub = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='robot_state_publisher',
        output='screen',
        parameters=[{'frame_prefix': 'group1/',
                     'robot_description': robot_desc}],
        namespace='group1'
    )

robot1_node = Node(
        name='puzzledrone',
        package='puzzle_drone',
        executable='puzzledrone_joint_state_pub',
        namespace='group1',
        parameters=[{
                    'init_pose_x':2.0,
                    'init_pose_y': 2.0,
                    'init_pose_z': 1.0,
                    'init_pose_yaw': 1.57,
                    'init_pose_pitch': 0.0,
                    'init_pose_roll': 0.0,
                    'odom_frame':'odom'
                }]
            )
```

Activity 3

Instructions

- Save and compile the file

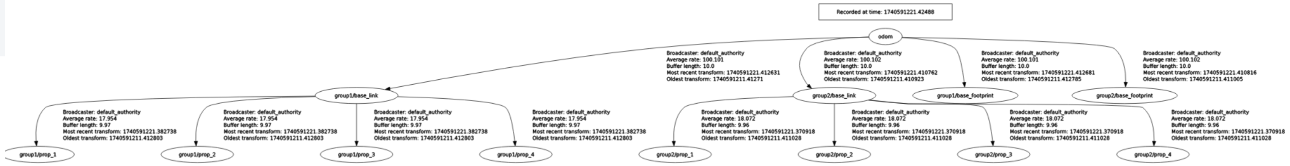
```
$ cd ~/ros2_ws
$ colcon build --packages-select puzzle_drone
$ source install/setup.bash
```

- Launch the node

```
$ ros2 launch  puzzle_drone multi_puzzledrone_launch.py
```

- Open “rqt_tf_tree” to view both robots tree’s.

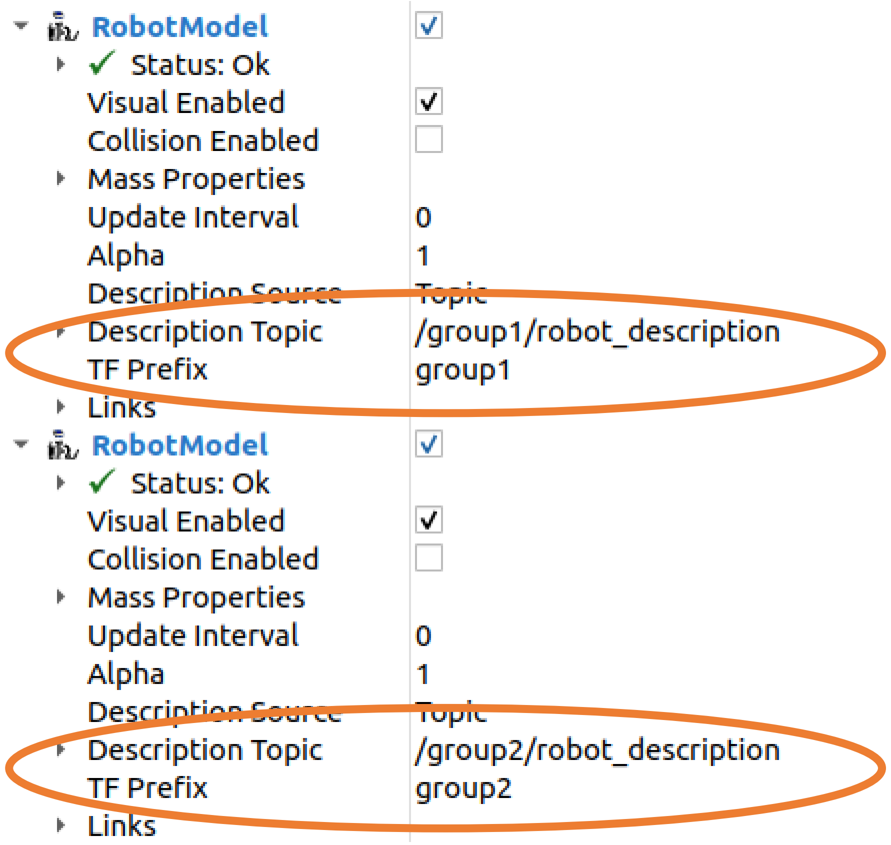
Results



Activity 3

Instructions

- Open “rviz” and add “case;” to view both transforms.
- Add two robot models.
  - In the description topic select “/group1/robot_description”
  - Since the transforms now contain a namespace, a TF Prefix must be added using the option “TF Prefix” in Rviz “Robot Model”.
  - Type in the “TF Prefix” box the namespace used in this case; the prefix must be “group1”
  - Do the same for the “group2”



Results