{Learn, Create, Innovate};

Robot Operating System – ROS

Practicalities

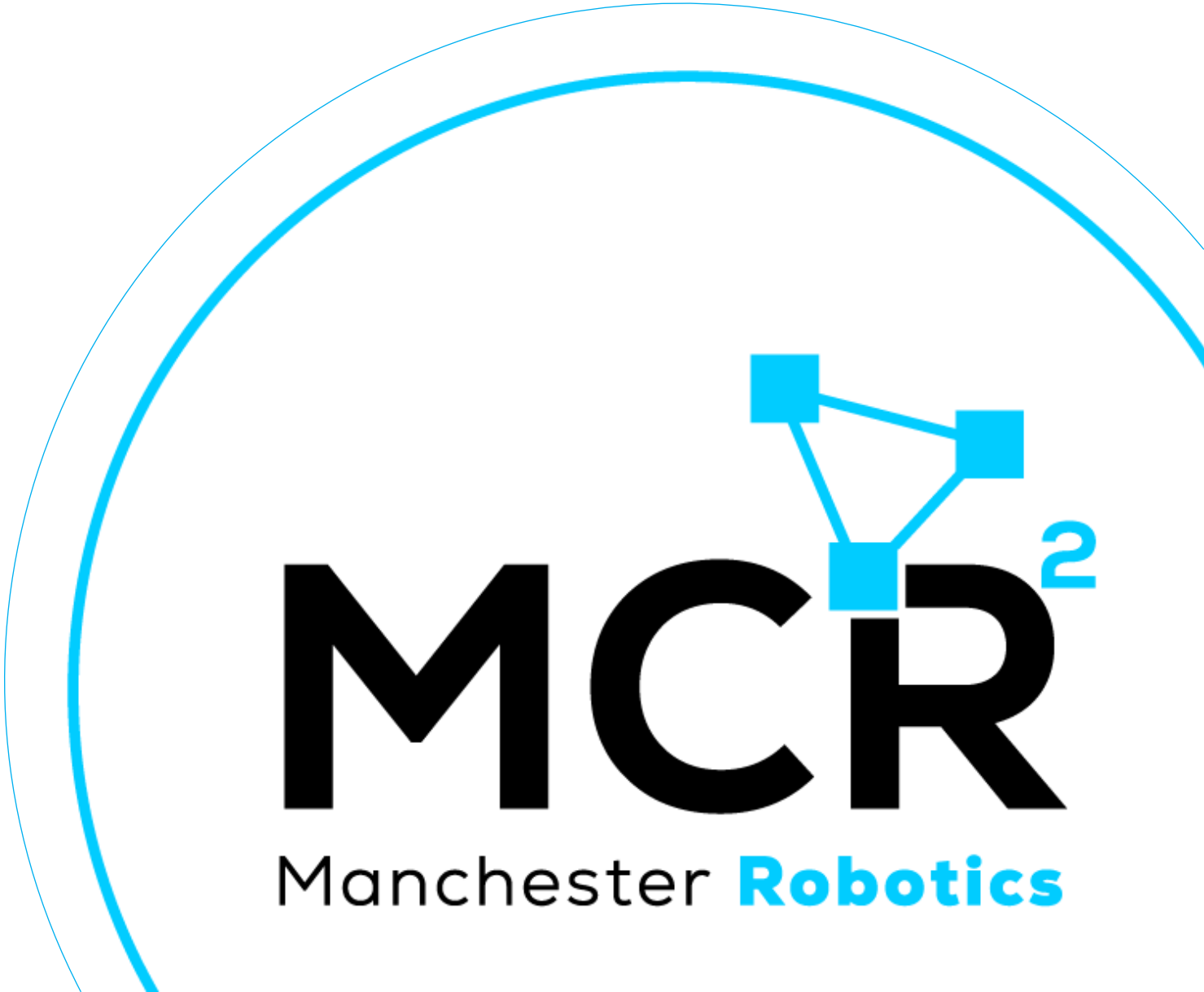
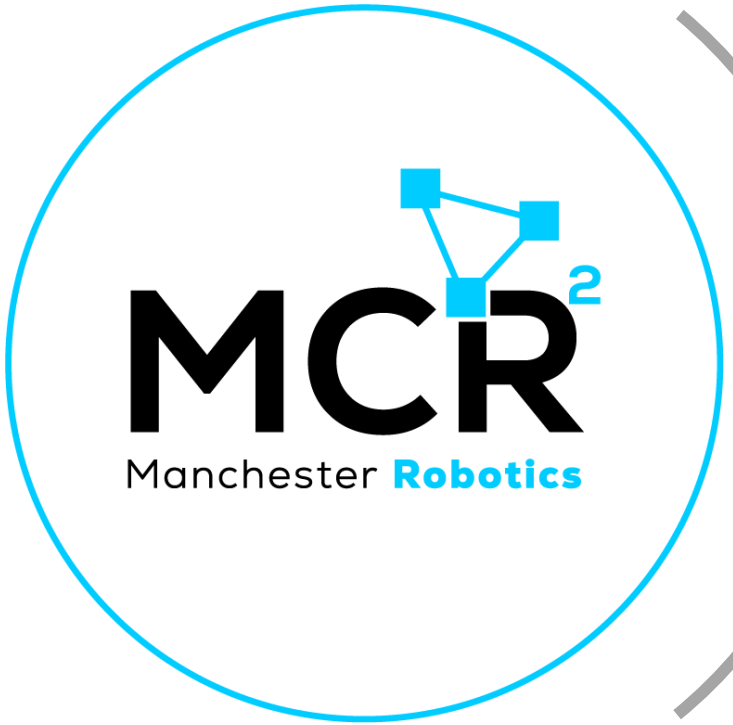


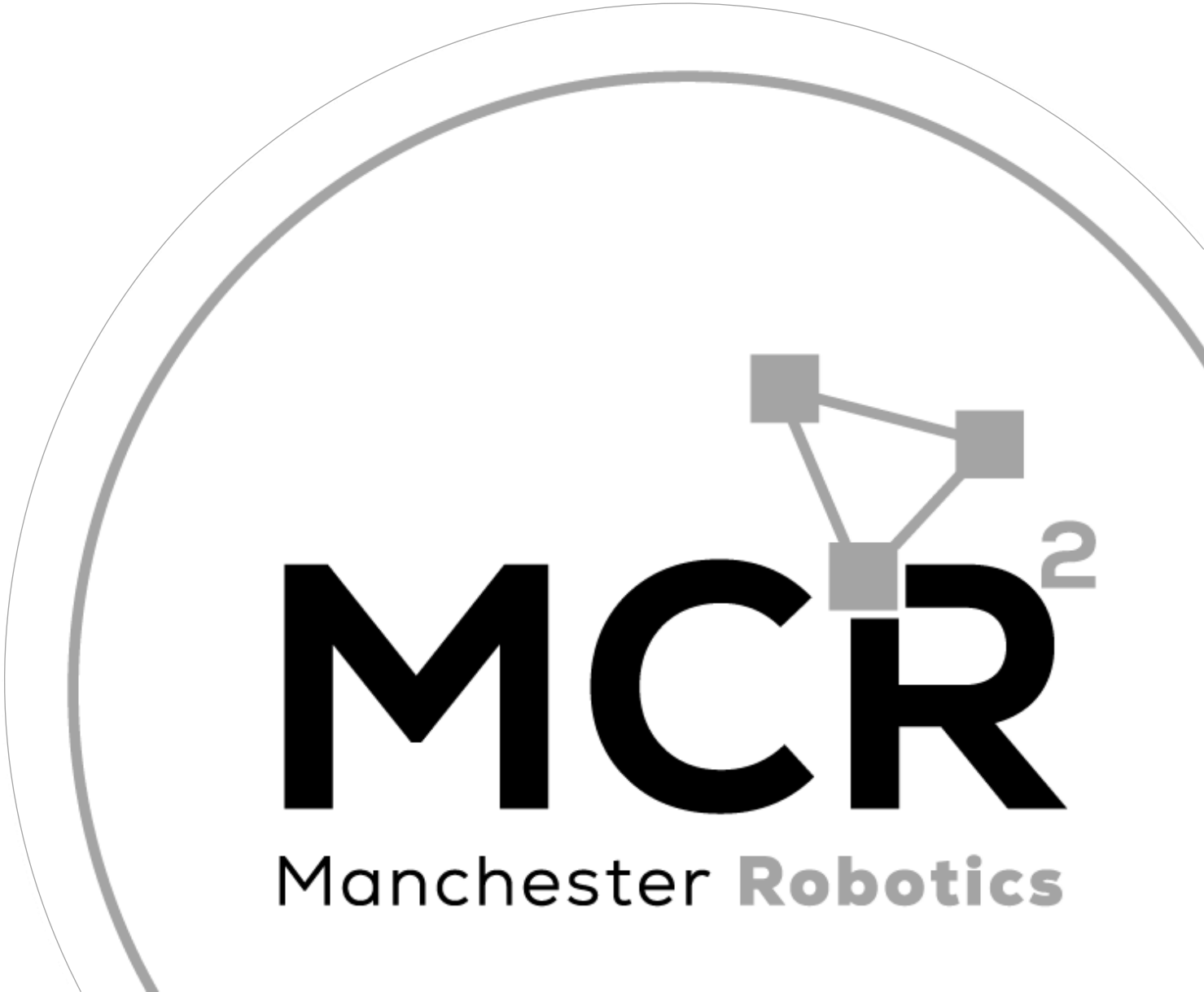
Table of contents



Robot Operating System – ROS

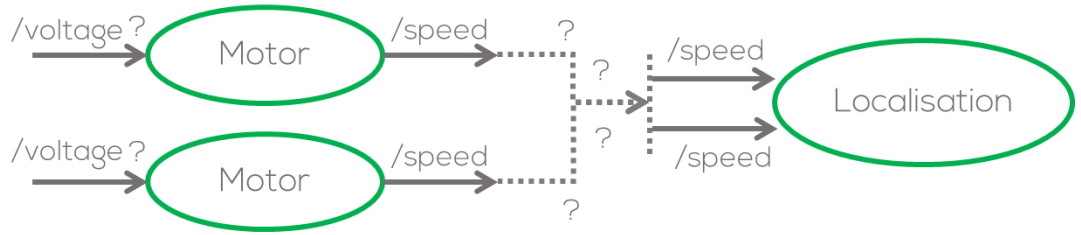
Namespaces

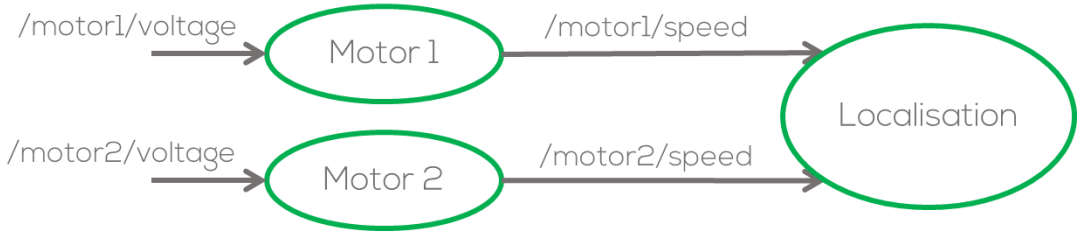
{Learn, Create, Innovate};



ROS Namespaces

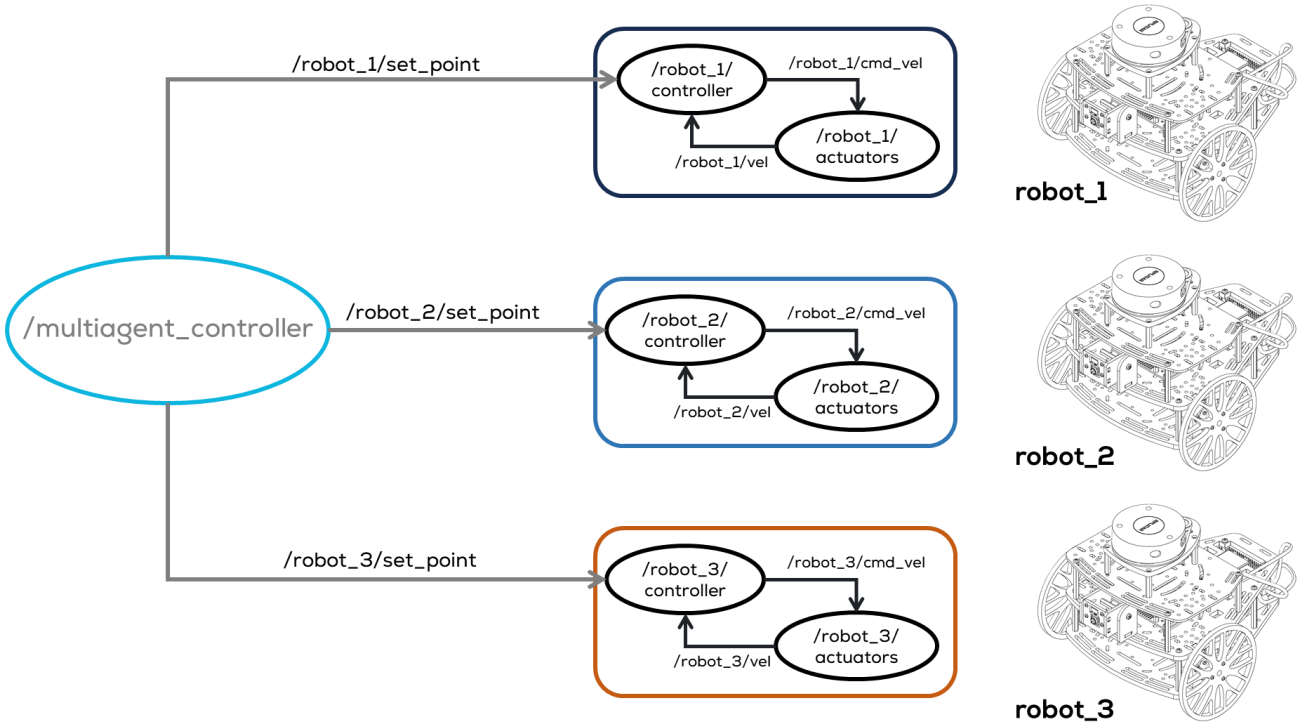
- Imagine the following problem: you have a node that simulates a motor, and you require to simulate two (or more) motors using the same code.
- The problem in ROS will be the naming convention for the nodes and the topics to which the motor node subscribes, and where it publishes; since they will both 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 then become the best option to deal with name collisions, when systems become more complex.





ROS Namespaces

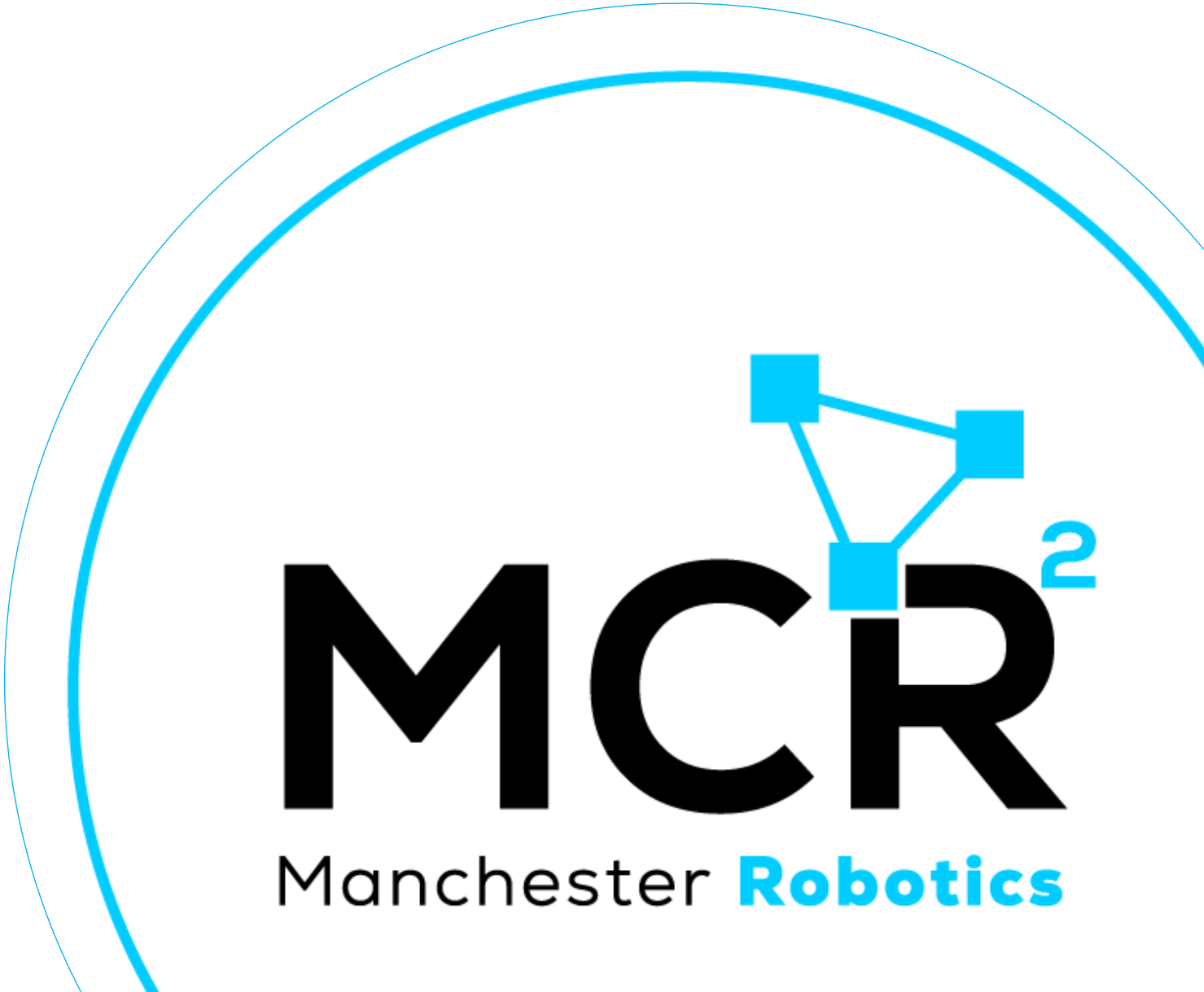
- A namespace in ROS can be viewed as a directory that contains items with different names.
- The items can be nodes, topics or other namespaces (hierarchy).
- There are several ways to define the namespaces. The easiest way is via the command line, which is very easy but not recommended for larger projects.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



Activity 1

ROS Namespaces

{Learn, Create, Innovate};



Activity 1 – ROS Namespaces

Requirements

- You can download the motor_control template package from Github (Week 2/Activities/Activity 2/Templates).
- Activity starts in slide 14

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
```

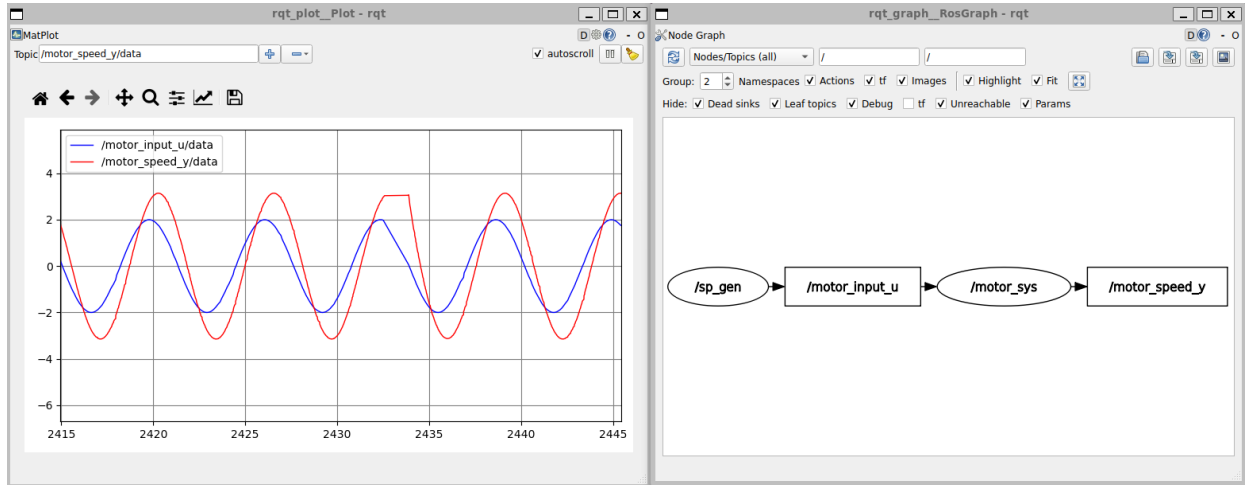
- Open two terminals run the rqt_graph and the rqt_plot

```
$ 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).

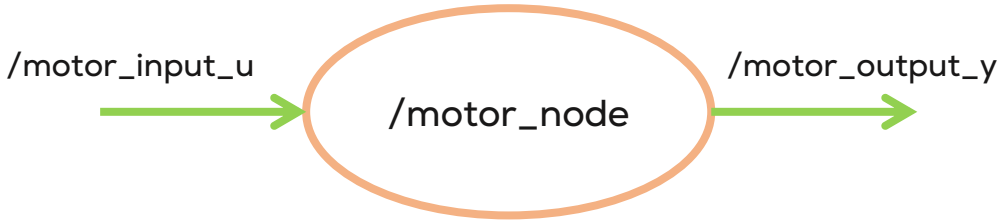
$$y[k+1] = y[k] + \left(-\frac{1}{\tau} \cdot y[k] + \frac{K}{\tau}\ u[k] \right) T_s$$

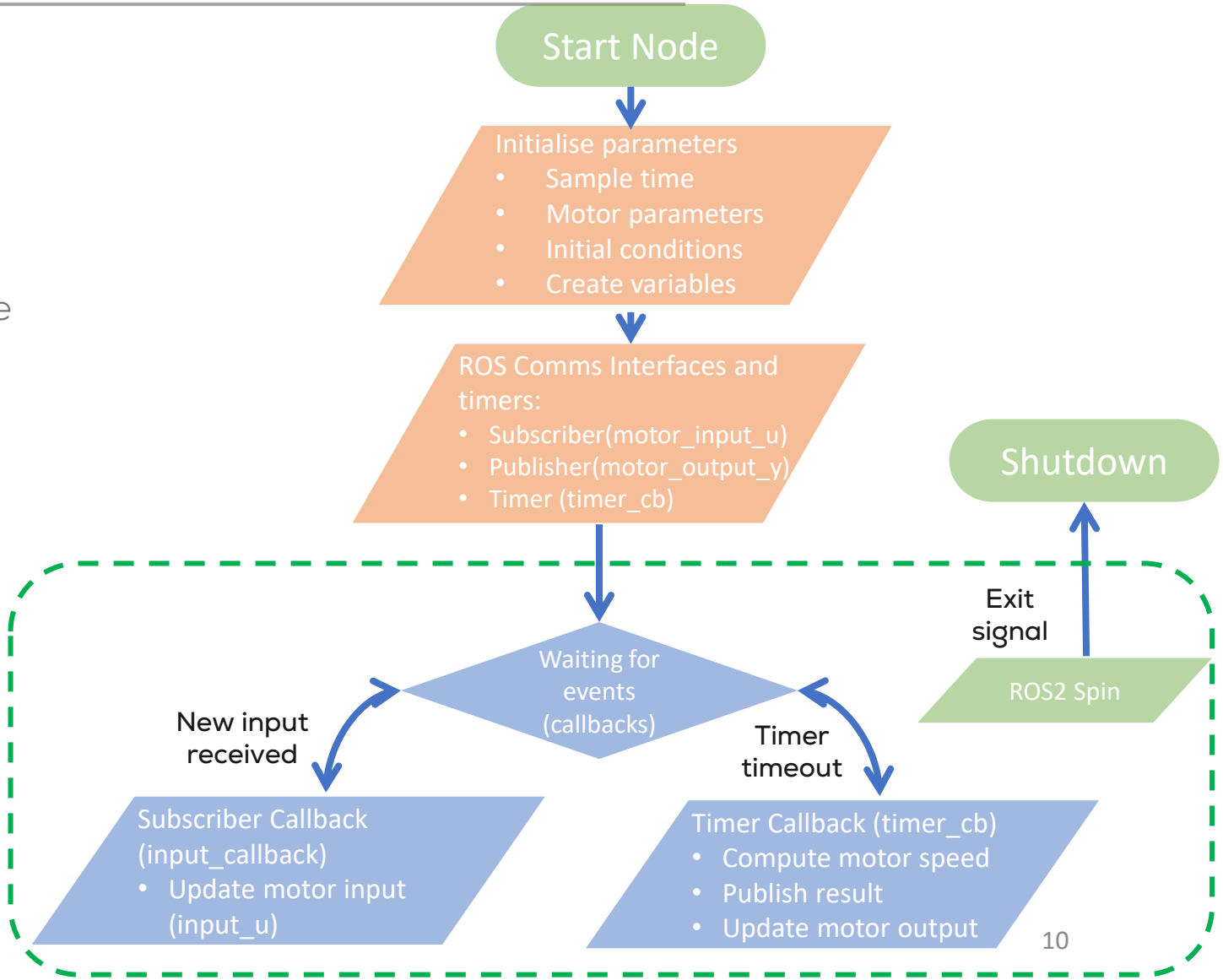
Where T_s is the sampling time.

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

```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')

        # 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

        #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')
```

Libraries

Initialise parameters

ROS publishers, subscribers Timers

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

#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()
```

Timer callback: Motor simulation

Subscriber callback

Main

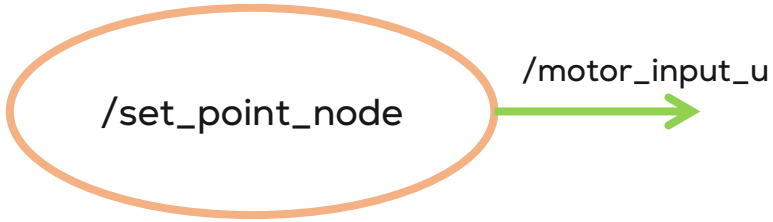
Activity 1 – ROS Namespaces

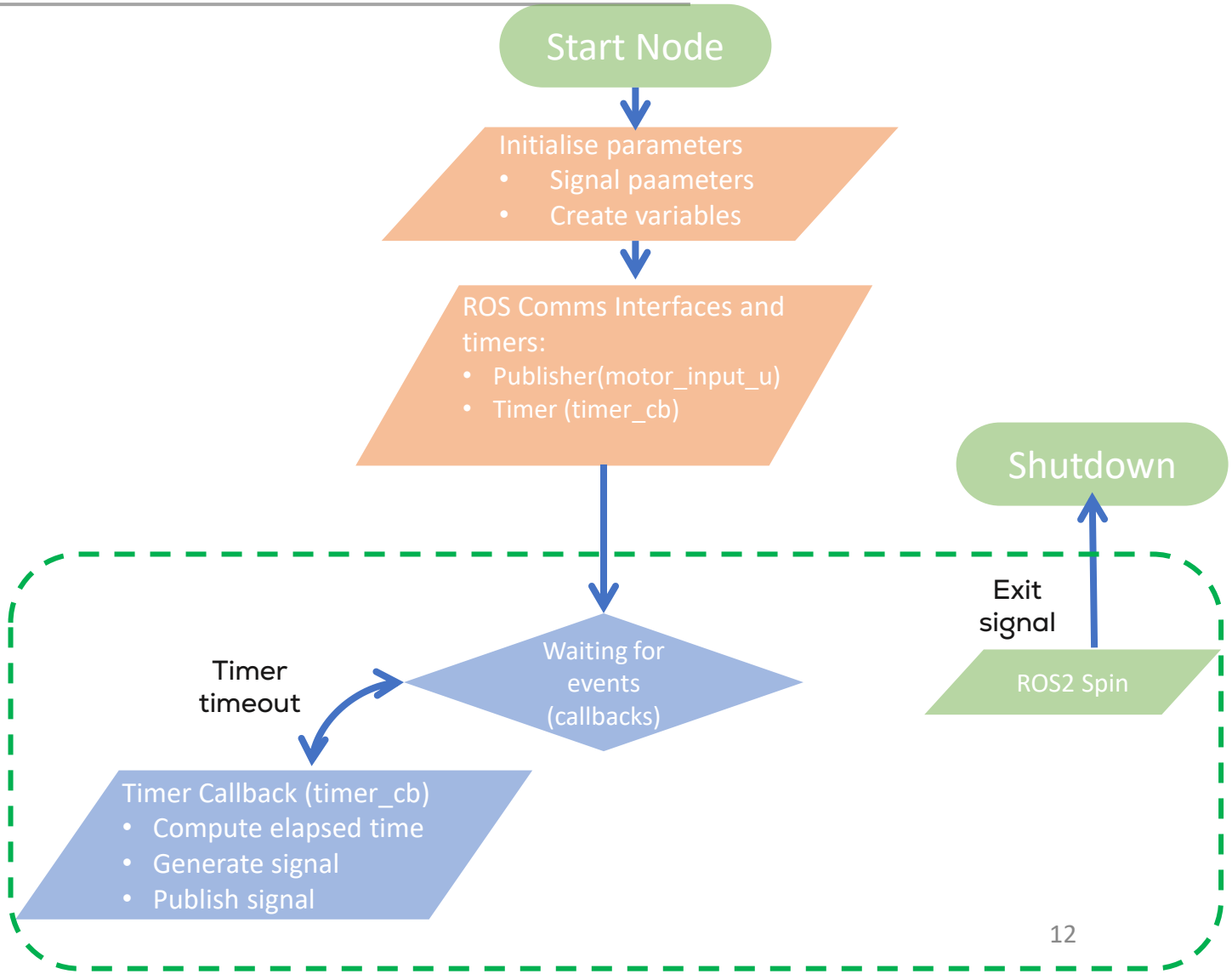
Set Point node structure

- 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

```
# 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

```
    # 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()
```

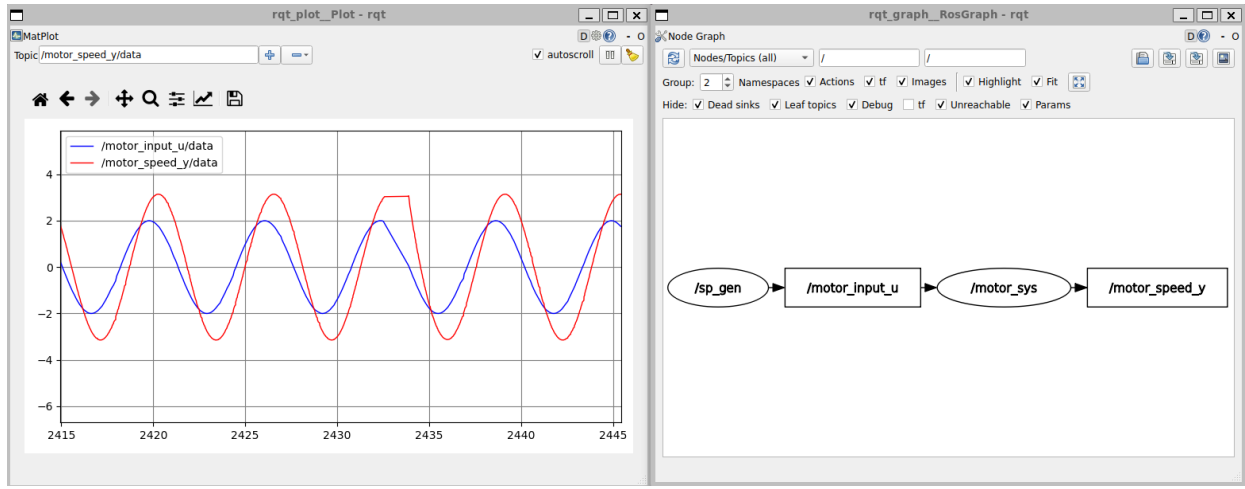
Timer callback: Signal Generator

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

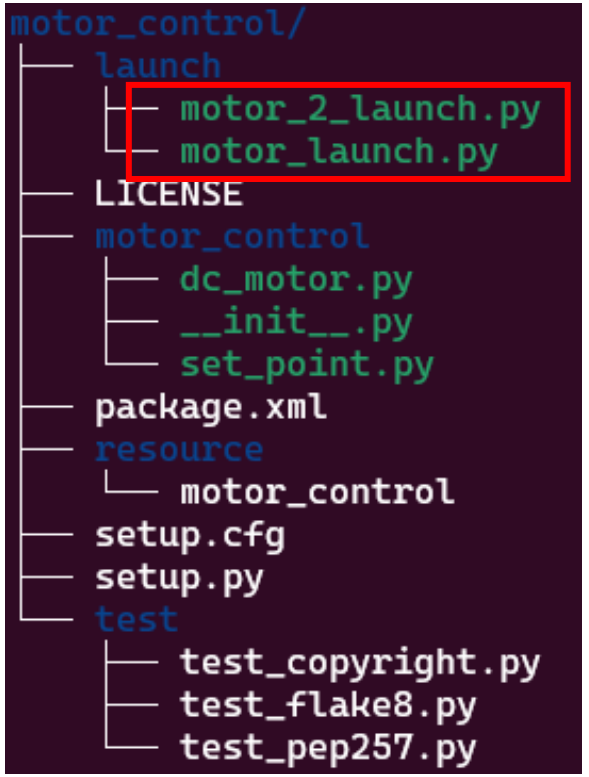
Adding a namespace

- 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



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*

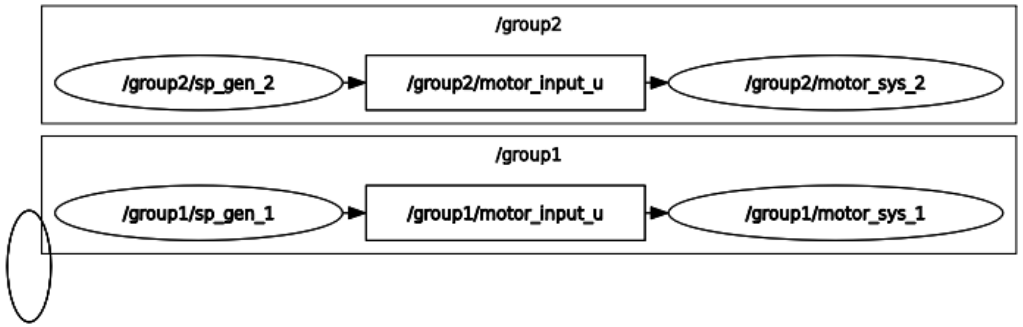
```
$ ros2 run  rqt_graph rqt_graph
```

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

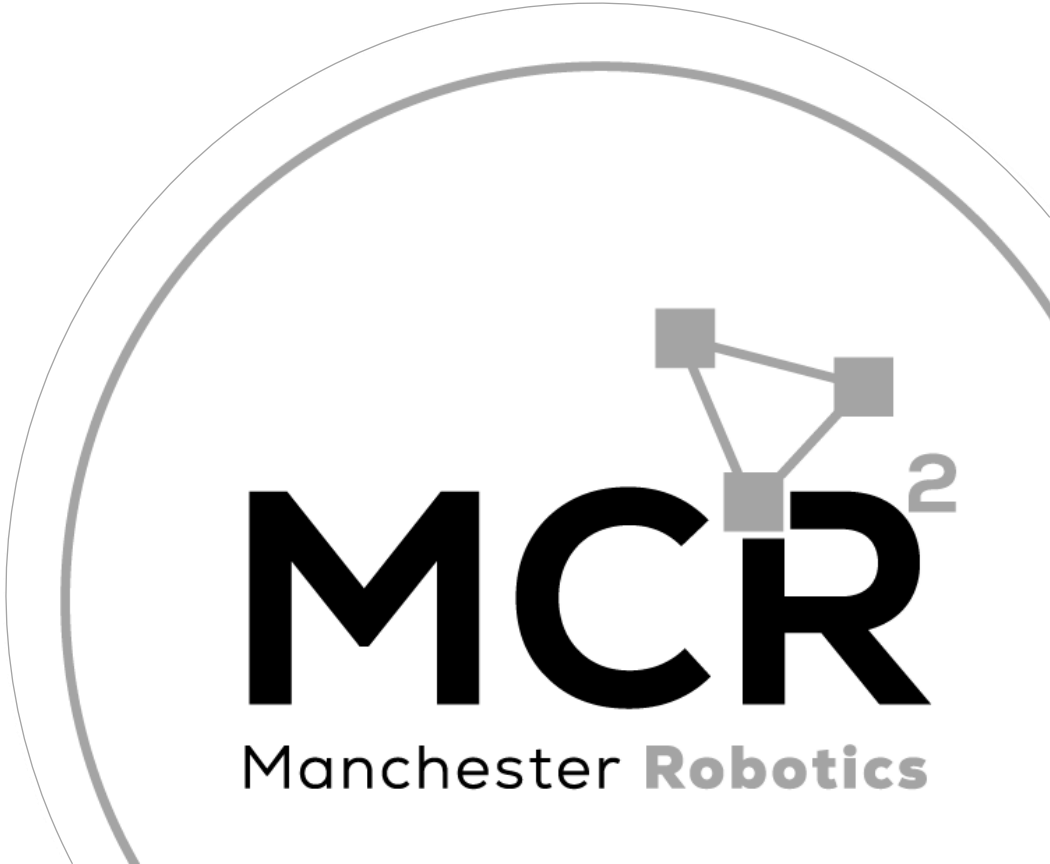



```
mario@MarioPC:~$ ros2 topic list
/group1/motor_input_u
/group1/motor_speed_y
/group2/motor_input_u
/group2/motor_speed_y
/parameter_events
/rosout
```

Activity 1.1

ROS Namespaces 2

{Learn, Create, Innovate};



Activity 1.1 – ROS Namespaces 2

- In the previous exercise, a namespace for each node was defined using the parameter

```
namespace= 'group1',
```

- If the **launch file contains many nodes**, defining namespaces can become very difficult.
- One solution is to **call the entire launch file directly from another launch file** (nested launch file) and **assign a namespace so that every nested node will inherit that namespace**.

Requirements

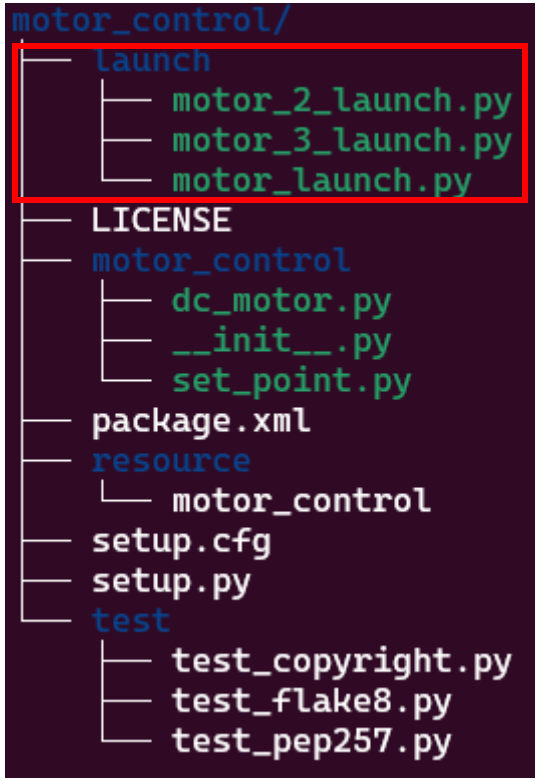
- This activity requires the package motor_control.
- For this example, three groups of nodes will be generated using namespaces.
- A launch file will be to nest another launch file.

Activity 1.1 – ROS Namespaces 2

- Create and open "*motor_3_launch.py*" file in the launch folder of the *motor_control* package.

```
$ cd ~/ros2_ws/src/motor_control/launch
$ touch motor_3_launch.py
$ chmod +x motor_3_launch.py
```

- Open the motor_3_launch.py on a text editor.



Activity 1.1 – ROS Namespaces 2

```python
#IMPORTS REQUIRED TO SET THE PACKAGE ADDRESS (DIRECTORIES)
import os
from ament_index_python.packages import get_package_share_directory

#iMPORTS REQUIRED FOR CALLING OTHER LAUNCH FILES (NESTING)
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource

#IMPORTS REQUIRED TO PUSH A NAMESPACE (APPEND) A NAMESPACE TO A NESTED LAUNCH FILE
from launch.actions import GroupAction
from launch_ros.actions import PushRosNamespace

#INITIATE LAUNCH FILE
def generate_launch_description():

    #USER VARIABLES
    package = 'motor_control'             #Package to be launched
    launch_file = 'motor_launch.py' #Launch file to get a namespace

    group1_ns = 'group1'    #namespace to be used for group 1
    group2_ns = 'group2'    #namespace to be used for group 2
    group3_ns = 'group3'    #namespace to be used for group 3

    #Get the address of the package
    package_directory = get_package_share_directory(package)
    #Get the address of the launch file
    launch_file_path = os.path.join(package_directory, 'launch', launch_file)
    #Set the launch file source for the group1 and group2
    launch_source1 = PythonLaunchDescriptionSource(launch_file_path)
    launch_source2 = PythonLaunchDescriptionSource(launch_file_path)
```

```python
    #Include the launch description for group1
    talker_listener_launch_1 = IncludeLaunchDescription(launch_source1)
    #Include the launch description for group2
    talker_listener_launch_2 = IncludeLaunchDescription(launch_source2)

    #Include the launch description for group3
    #THIS IS ANOTHER WAY OF DOING THE PREVIOUS STEPS (MORE COMPACT) THE RESULTS ARE THE SAME
    talker_listener_launch_3= IncludeLaunchDescription(
        PythonLaunchDescriptionSource([os.path.join(
            get_package_share_directory('motor_control'), 'launch'),
            '/motor_launch.py'])
    )

    #SET NAMESPACE FOR ALL THE NODES INSIDE THE LAUNCH FILE
    motor_control_group1 = GroupAction(
        actions=[PushRosNamespace(group1_ns),
                 talker_listener_launch_1,
                 ]
    )

    motor_control_group2 = GroupAction(
        actions=[PushRosNamespace(group2_ns),
                 talker_listener_launch_2,
                 ]
    )

    motor_control_group3 = GroupAction(
        actions=[PushRosNamespace(group3_ns),
                 talker_listener_launch_3,
                 ]
    )

    #LAUNCH THE DESCRIPTION
    l_d = LaunchDescription([motor_control_group1, motor_control_group2,
motor_control_group3])
    return l_d
```

Activity 1.1 – ROS Namespaces 2

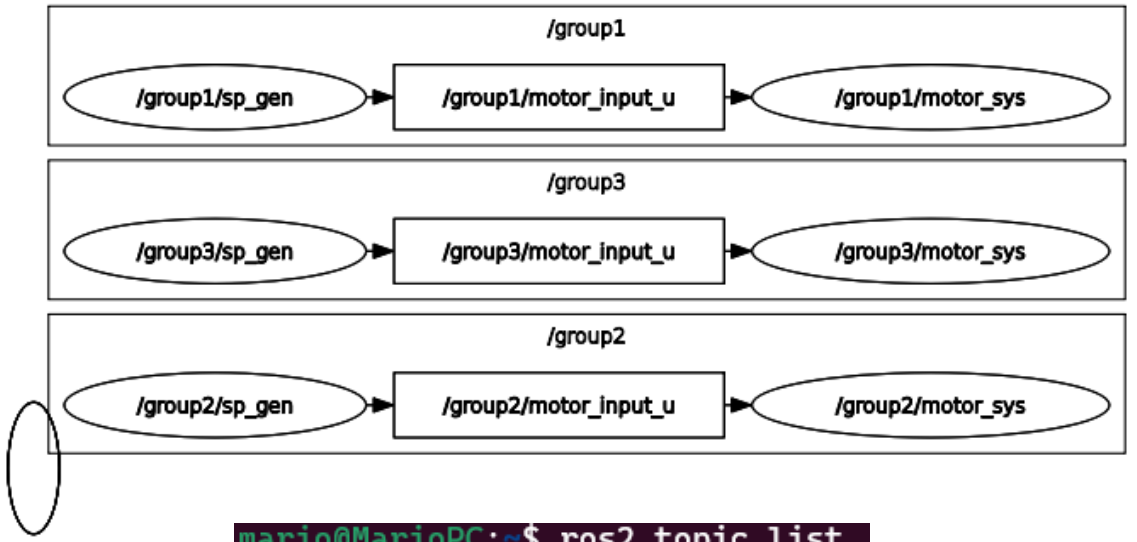
- Build and run the newly created lunch file using colcon.

```
$ cd ~/ros2_ws
$ colcon build
$ source install/setup.bash
$ ros2 launch motor_control motor_3_launch.py
```

- Open in another terminal the *rqt_graph to visualise the nodes*

```
$ ros2 run  rqt_graph rqt_graph
```

Results

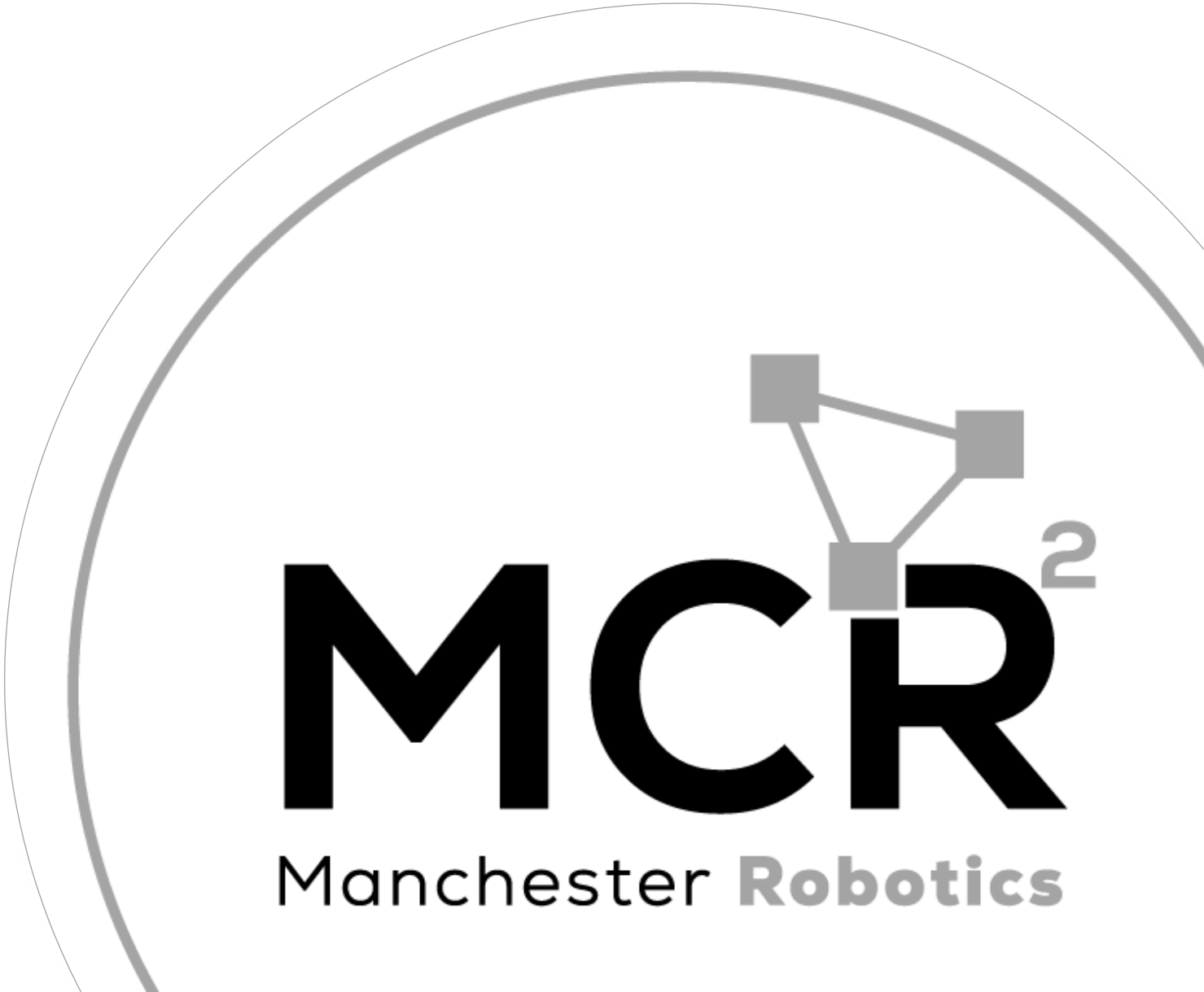



```
mario@MarioPC:~$ ros2 topic list
/group1/motor_input_u
/group1/motor_speed_y
/group2/motor_input_u
/group2/motor_speed_y
/group3/motor_input_u
/group3/motor_speed_y
/parameter_events
/rosout
```

Robot Operating System – ROS

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.

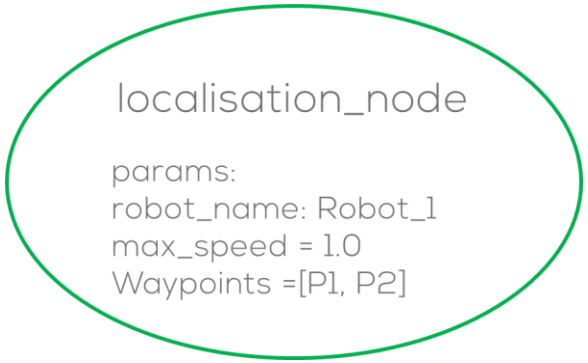
ROS Parameters

- 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.

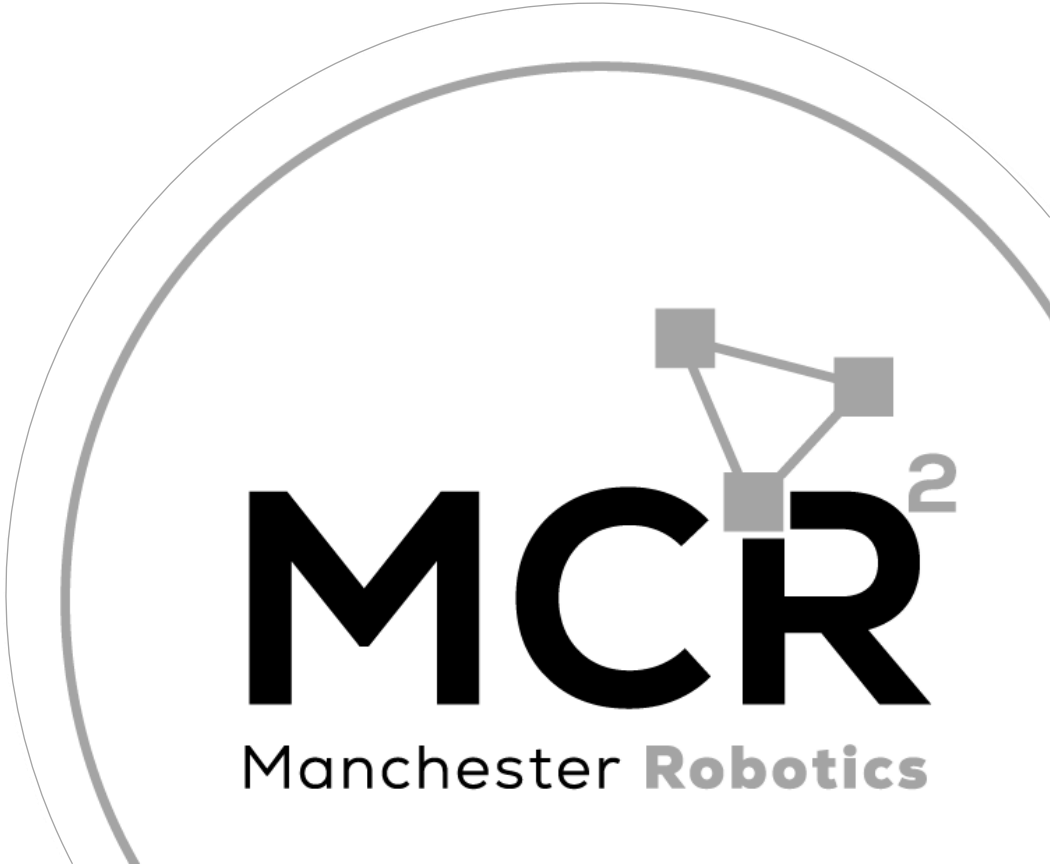
```
key      value      descriptor
<Name> <Value>     <Description of the parameter (empty)>
```



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

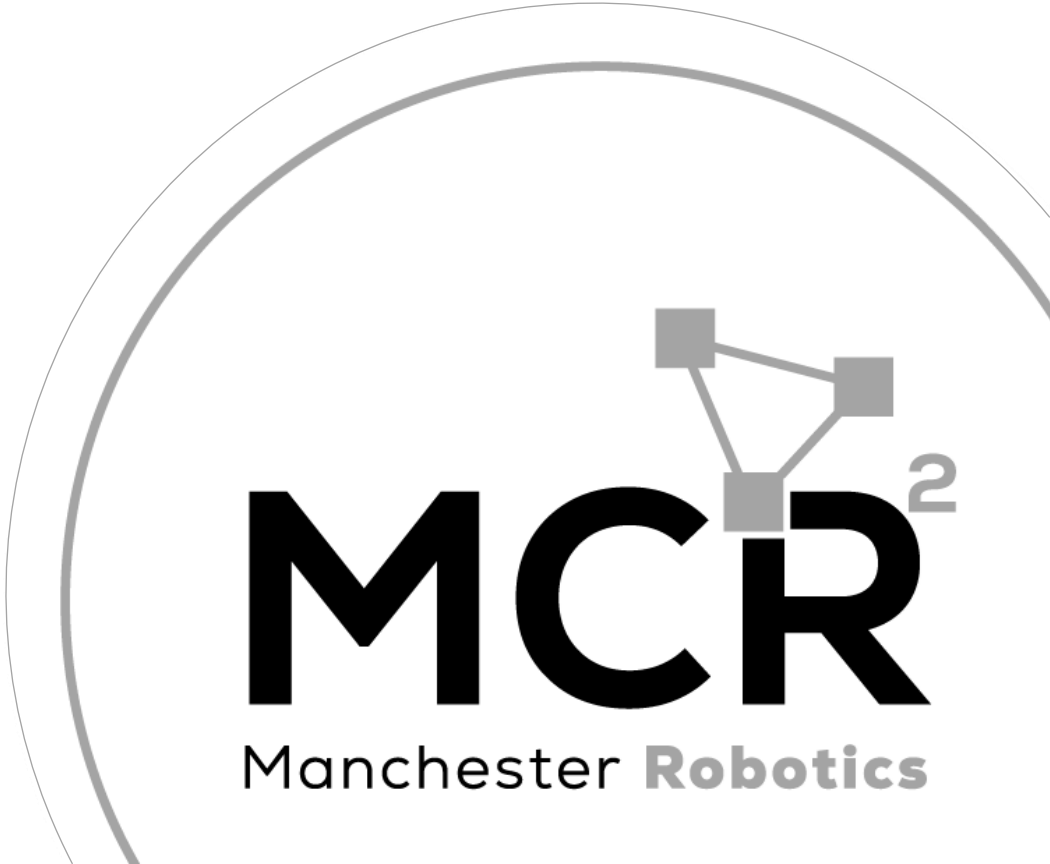
```
$ ros2 param load <node_name> <parameter_file>
```

- To start the same node using your saved parameter values

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

Robot Operating System – ROS

Parameter Callbacks



{Learn, Create, Innovate};

Parameter Callbacks

- Setting a parameter as in the previous manner, allows us to set different values from the launch file.
- The previous method does not allow to change parameters at runtime, nor verify if the parameters are the correct data type or value is within bounds.

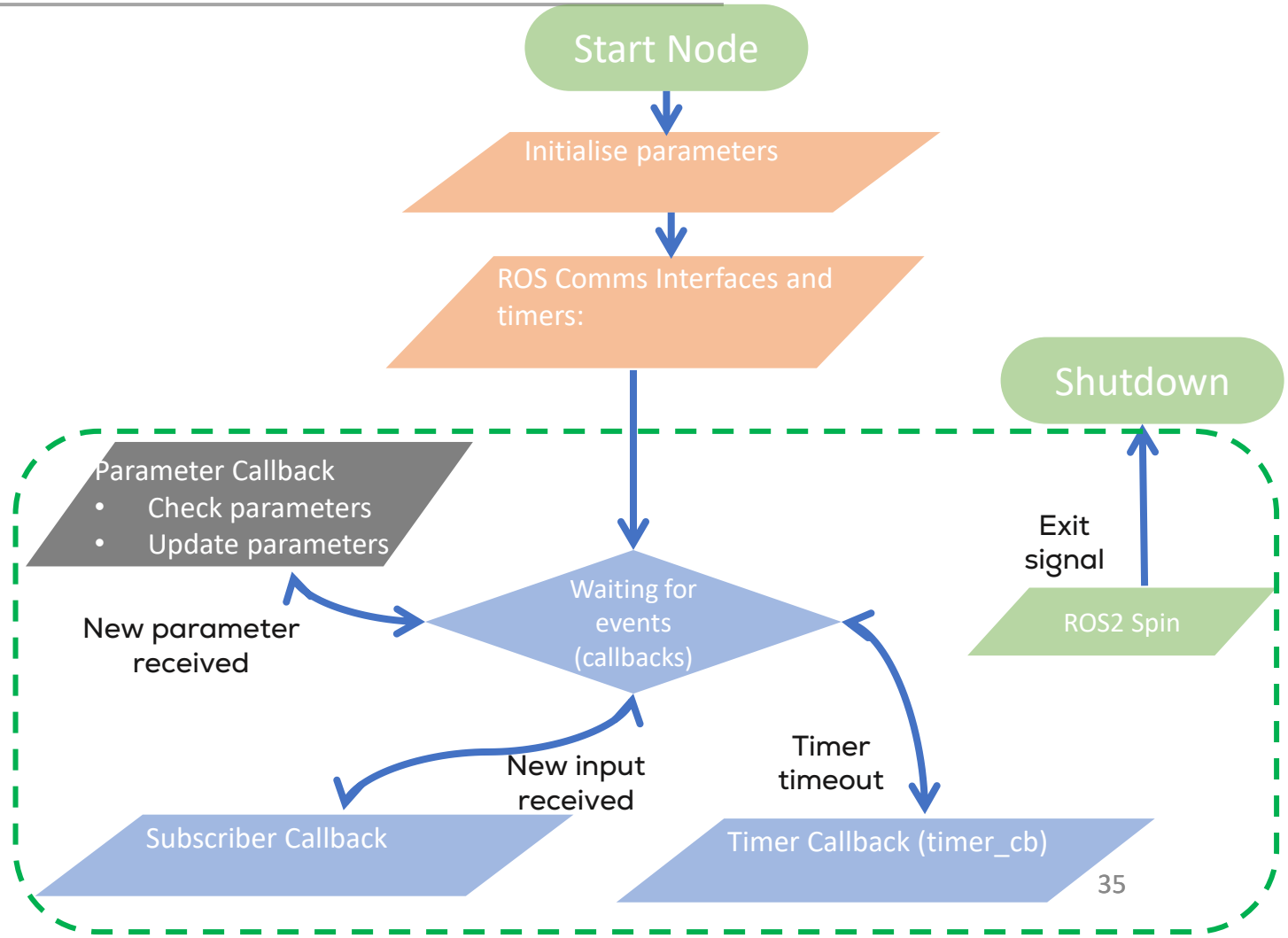
Parameter callbacks

- One simple solution would be to move the parameter get value inside the timer function…
  - This can be computationally expensive and unnecessary, especially if the parameter rarely changes.
  - If the loop is executing at a high frequency (e.g., every 10 ms), querying the parameter every iteration would be redundant and inefficient.
- ROS2 implements parameter callbacks to ensure the parameters are updated only when a parameter is modified.

Parameter Callbacks

Parameter callbacks

- A parameter callback triggers only when a parameter is modified, ensuring that updates are applied in real-time without needing to check in every loop iteration.
- This is particularly useful for applications where parameters might change at runtime (e.g., tuning control gains, adjusting robot behaviour dynamically).
  - The callback handles parameter updates.
  - The main loop handles the actual execution.



Parameter Callbacks (Humble)

Parameter Callbacks

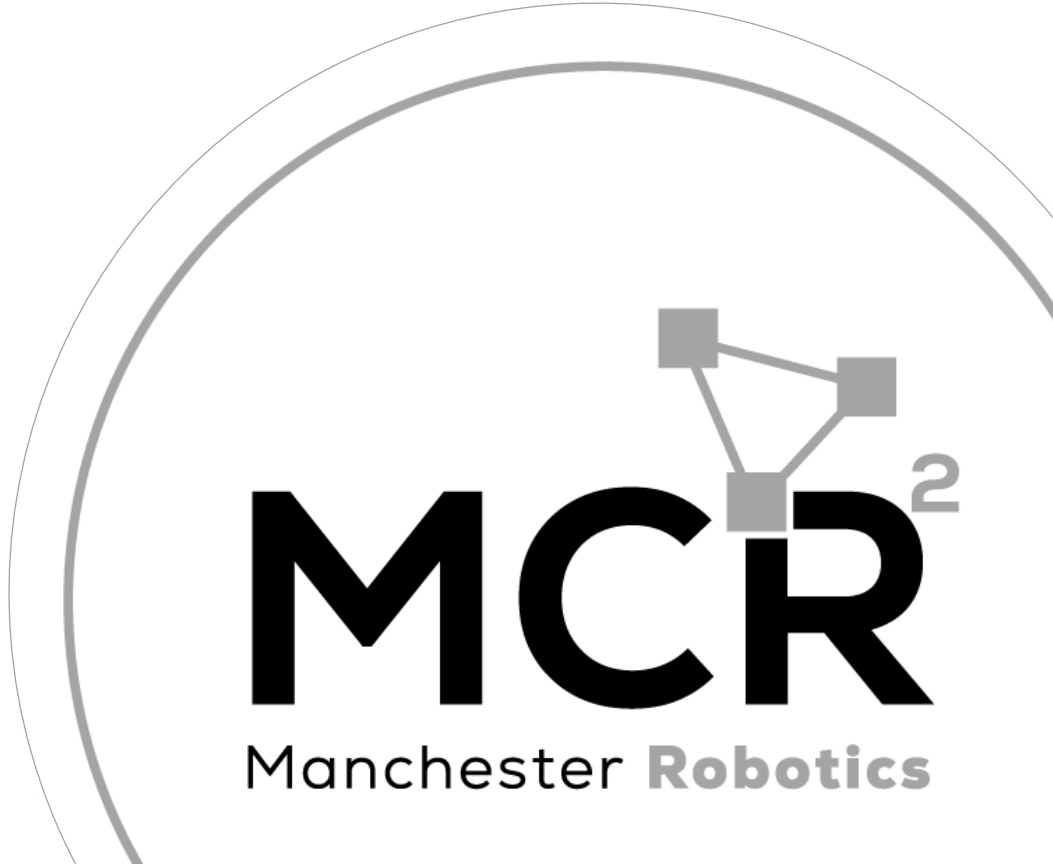
- OnSet Callback (add_on_set_parameters_callback()):
  - Allows pre-validation of parameters before they are set.
  - The callback is passed a list of immutable Parameter objects.
  - Returns an rcl_interfaces/msg/SetParametersResult.
  - The main purpose of this callback is to give the user the ability to inspect the upcoming change to the parameter and explicitly reject the change.

```python
import rclpy
from rclpy.node import Node
from rcl_interfaces.msg import SetParametersResult

class Control(Node):

    def parameters_callback(self, params):
        # validate parameters, update class attributes, etc.
        return SetParametersResult(successful=True)


    def __init__(self):
        super().__init__('params_callback')
        self.declare_parameter('P_gain', 0.5)
        self.p_gain = self.get_parameter('P_gain').value

        self.add_on_set_parameters_callback(self.parameters_callback)
     ...
```

Activity 2.1

Changing parameters at runtime

{Learn, Create, Innovate};



Activity 2.1: Parameter Callbacks

Setting the Parameter Callbacks

- Open the dc_motor.py file with a text editor.
- Add the message (interface) SetParameterResult at the top.
- Set parameter the callback in the constructor.
- Define a callback function inside the class DCMotor

```
# Imports
...
from rcl_interfaces.msg import SetParametersResult


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

        # Declare parameters
        # System sample time in seconds
        self.declare_parameter('sample_time', 0.02)
        ...

        #Parameter Callback
        self.add_on_set_parameters_callback(self.parameters_callback)

        ...


    def parameters_callback(self, params):
            # validate parameters, update class attributes, etc.
        return SetParametersResult(successful=True)
```

Activity 2.1: Parameter Callbacks

Callbacks Function

- Fill the callback function as shown.
- The callback function will receive the list of the parameters to be modified (params).
- For each parameter inside the list:
  - Verify the parameters are within bounds
  - Set the variables that drive the system.

```python
def parameters_callback(self, params):
    for param in params:
        #system gain parameter check
        if param.name == "sys_gain_K":
            #check if it is negative
            if (param.value < 0.0):
                self.get_logger().warn("Invalid sys_gain_K! It cannot be negative.")
                return SetParametersResult(successful=False, reason="sys_gain_K cannot be negative")
            else:
                self.param_K = param.value  # Update internal variable
                self.get_logger().info(f"sys_gain_K updated to {self.param_K}")

            #system gain parameter check
        if param.name == "sys_tau_T":
            #check if it is negative
            if (param.value < 0.0):
                self.get_logger().warn("Invalid sys_tau_T! It cannot be negative.")
                return SetParametersResult(successful=False, reason="sys_tau_T cannot be negative")
            else:
                self.param_T = param.value  # Update internal variable
                self.get_logger().info(f"sys_tau_T updated to {self.param_T}")

    return SetParametersResult(successful=True)
```

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
```

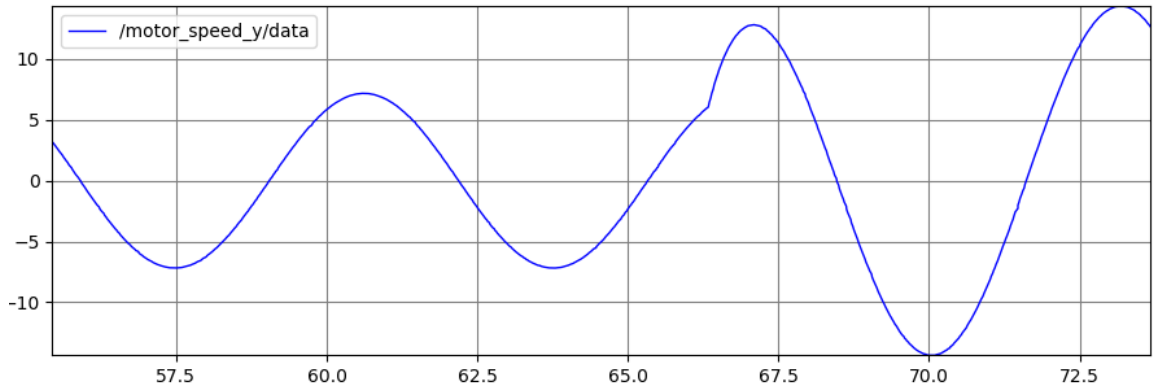
Results

- Open the rqt_plot

```
$ ros2 run rqt_plot rqt_plot
```

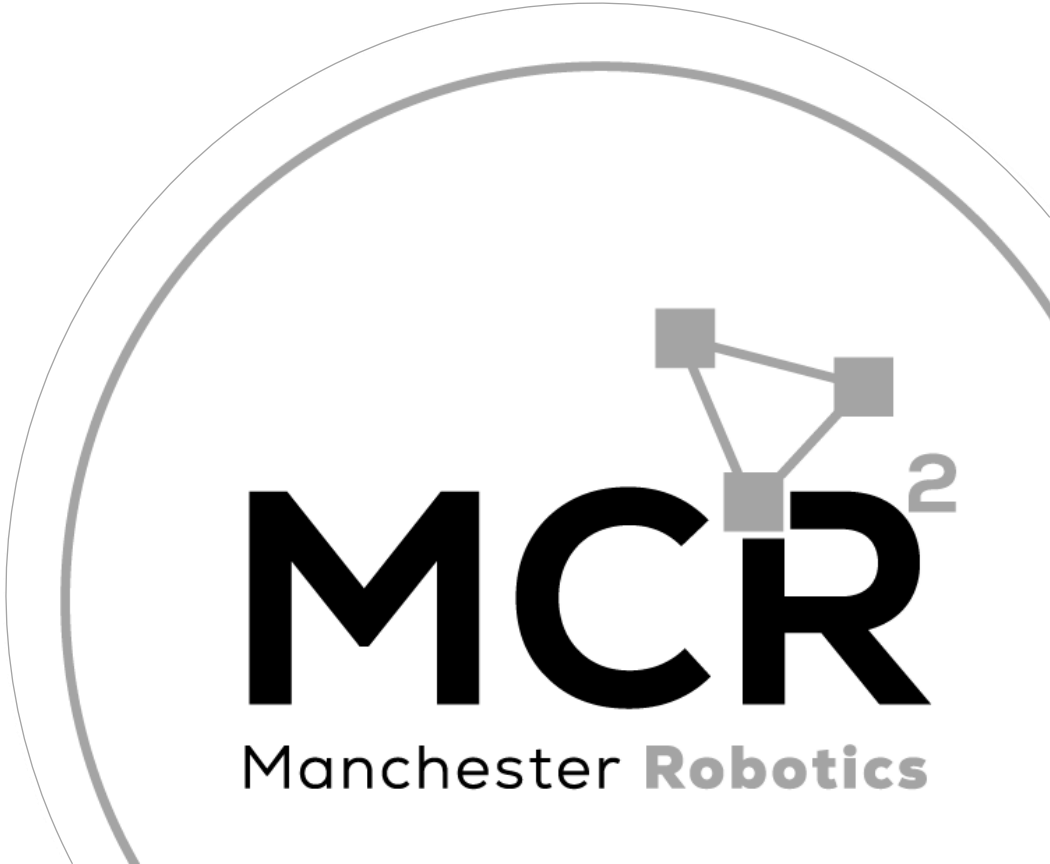
- Change a parameter gain K

```
$ ros2 param set /motor_sys sys_gain_K 8.0
```



Robot Operating System – ROS

Parameter Files

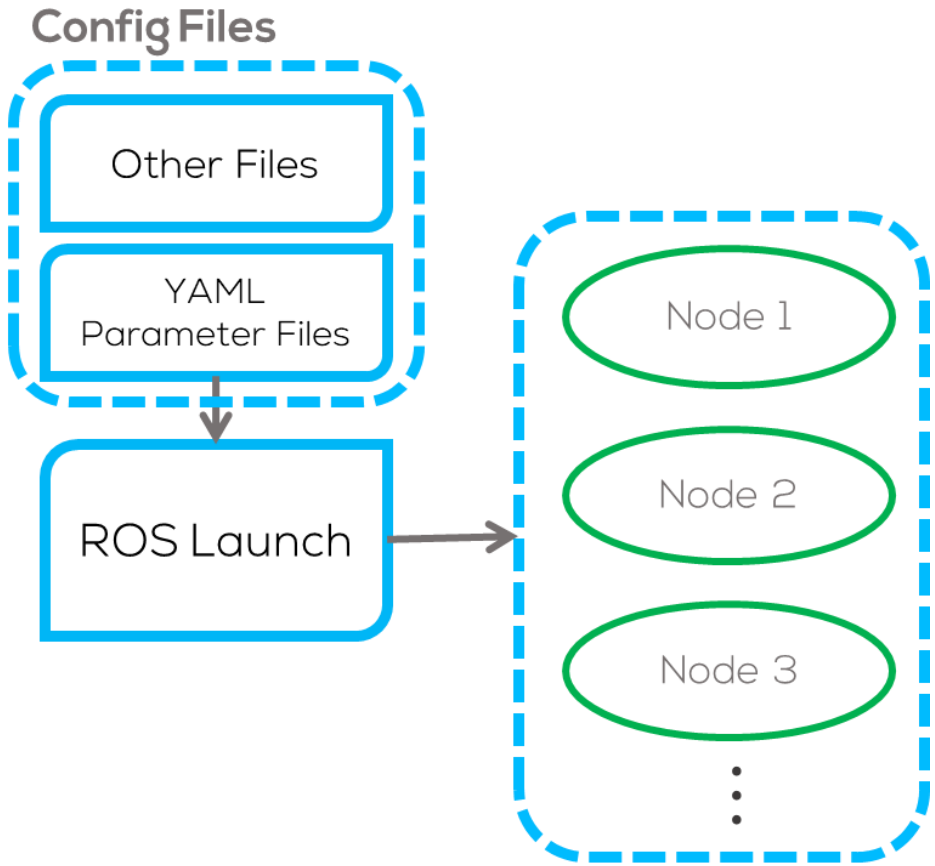


{Learn, Create, Innovate};

Parameter files

Parameter Files

- The previous way of defining parameters is very useful, but for the case when having to define many different parameters this can become very inefficient.
- ROS offers the capability to define parameters using a parameter file.
- This parameter file is called YAML file because of the language is written (YAML: yet another markup language)



Parameter files

YAML Files

- These files are commonly used in other languages to set up parameters or variables.
- The way to define the hierarchy of a parameter, like in python depends on spacing!!!.
- The parameters set up in the config files (YAML Files) can be declared, private to a node or using “wildcards” will assign all the parameters in every node, despite differences in node names and namespaces.

```yaml
/**:
  ros__parameters:
    sample_time: 0.5
node_1:
  ros__parameters:
    some_text: "abc"
node_2:
  ros__parameters:
    int_number: 27
    float_param: 45.2
node_3:
  ros__parameters:
    int_number: 45
```

Parameter files

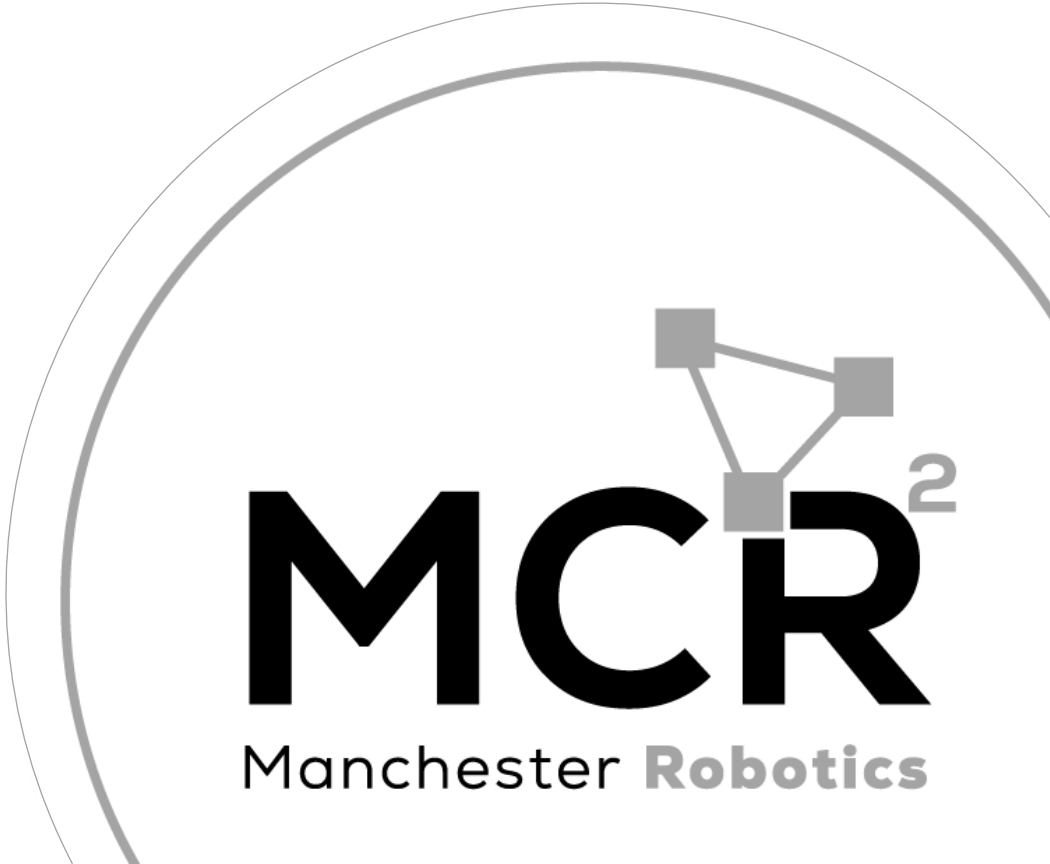
YAML file

- The same as you would put your launch files into a “launch” folder, you can put all your YAML config files into a “config” folder, directly at the root of your package.
- Inside the config folder create a file named “params.yaml” (the file can have any name just make sure follow a convention properly).
- The YAML file will be called by the launch file and set all the parameters according to the node and namespace (if applicable).

Activity 2.2

Parameter Files

{Learn, Create, Innovate};



Activity 2.2 – Parameter file

Requirements

- For this activity the motor_control package will be used.
- The previous namespace activity must be completed since motor_2_launch.py file will be used.

Instructions

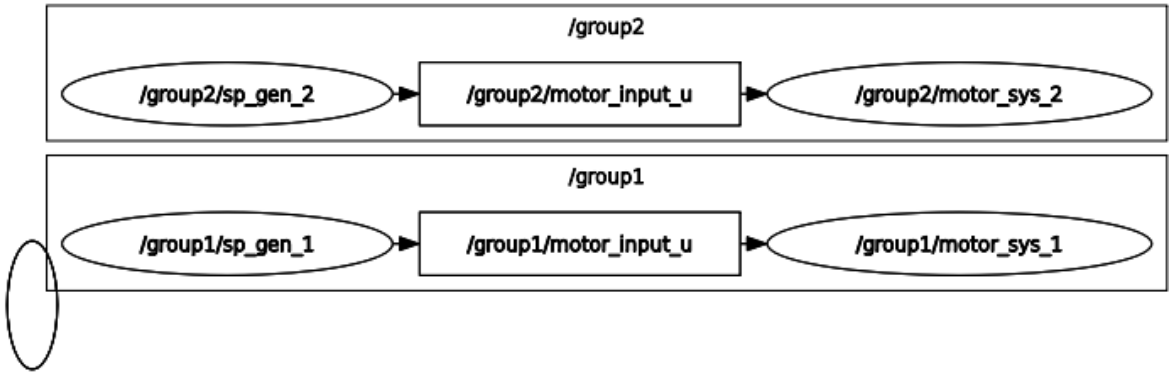
- Create a “config” folder in “motor_control” package.
- Create a “params.yaml” file inside the “config” folder.

```
$ cd ~/ros2_ws
$ mkdir src/motor_control/config
$ touch src/motor_control/config/params.yaml
```

Activity 2.2 – Parameter file

Instructions

- The parameters will be set for the “motor_2_launch.py” file.
- The “motor_2_launch.py” invokes two groups of nodes containing a motor node and a set point node respectively with a namespace, respectively.



Activity 2.2 – Parameter file

Instructions

- Open the “parameter.yaml” file using a text editor.
- According to the namespace and the robot's name assign the parameters to be set.

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

motor_2_launch.py

```
/**:
   ros__parameters:
    #WILDCARD: Parameters to be assigned
    #in every node, despite differences
    #in node names and namespaces

/group1/motor_sys_1:
  ros__parameters: #Careful is double underscore __
    sys_gain_K: 2.0 #Parameter to be changed
    sys_tau_T: 0.9


/group2/motor_sys_1:
  ros__parameters:
    sys_gain_K: 1.75
    sys_tau_T: 0.5
```

Activity 2.2 – Parameter file

Instructions

- Copy the following parameters inside the "params.yaml" file (careful with the spaces).

```
/group1/motor_sys_1:
  ros__parameters:
    sys_gain_K: 4.0
    sys_tau_T: 0.9


/group2/motor_sys_1:
  ros__parameters:
    sys_gain_K: 1.75
    sys_tau_T: 0.5
```

- Open the file setup.py of the motor_control package, and add the following to the data_files part:

```
(os.path.join('share', package_name, 'config'),
glob(os.path.join('config', '*.[yma]*'))),
```

- As follows (don't forget the comma before and after):

```
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', '*.[yma]*'))),
  ],
```

- Open the launch file "motor_2_launch.py" on a text editor and modify it as shown on the next slide.

Activity 2.2 – Parameter file

```python
#Packages to get the address of the YAML file
import os
from ament_index_python.packages import
get_package_share_directory

#Launch Pacckages
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():

    #Get the address of the YAML File
    config = os.path.join(
    get_package_share_directory('motor_control'),
                          'config',
                          'params.yaml'
                          )

    motor_node_1 = Node(name="motor_sys_1",
                        package='motor_control',
                        executable='dc_motor',
                        emulate_tty=True,
                        output='screen',
                        namespace="group1",
                        parameters=[config]
                        )
```

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

    motor_node_2 = Node(name="motor_sys_2",
                        package='motor_control',
                        executable='dc_motor',
                        emulate_tty=True,
                        output='screen',
                        namespace="group2",
                        parameters=[config]
                        )

    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
```

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_2_launch.py
```

- Verify the new parameters on terminal

```
$ ros2 param list
```

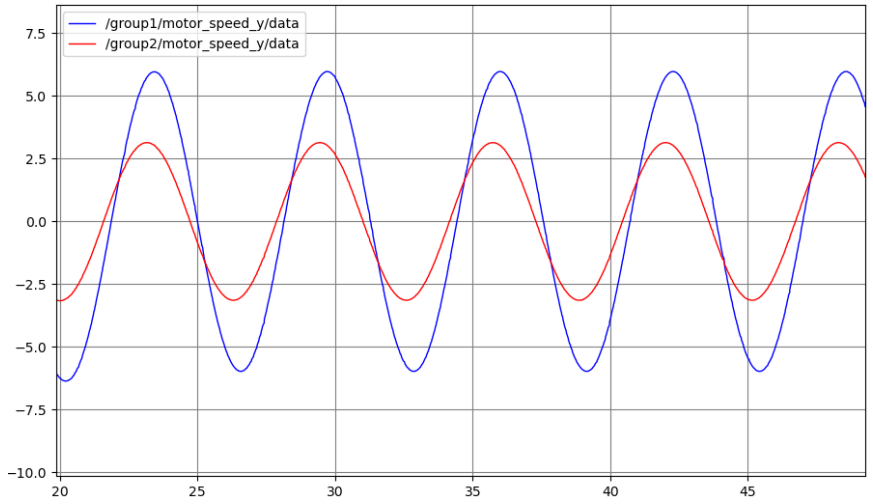
Results

- Open the rqt_plot

```
$ ros2 run rqt_plot rqt_plot
```

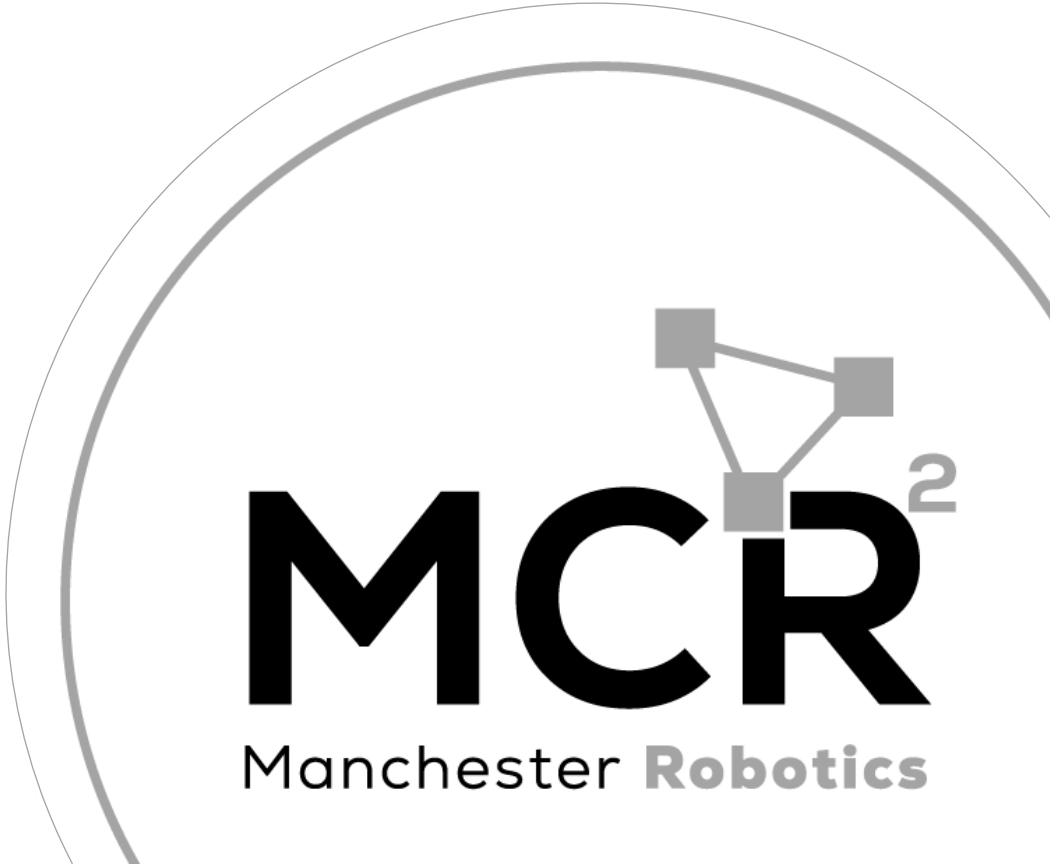
- Check the parameter gain K

```
$ ros2 param get /group1/motor_sys_1 sys_gain_K
Double value is: 4.0
```



Robot Operating System – ROS

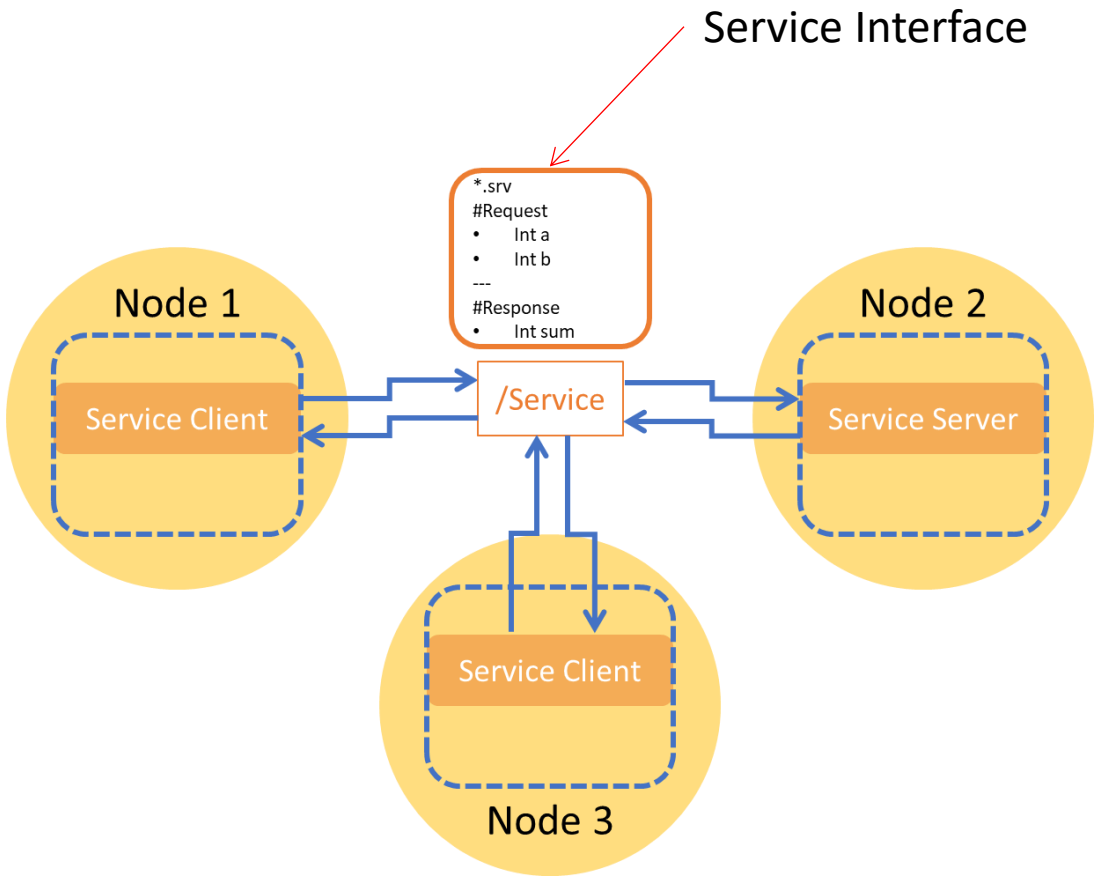
Services



{Learn, Create, Innovate};

Services

- Services are a vital method of communication for nodes within the Robot Operating System (ROS).
- A service operates on a request/response pattern, where a client sends a request to the service-providing node. The service processes this request and delivers a response.
- In contrast to topics that deliver continuous updates, services are highly efficient, providing data exclusively upon client request. This makes services an essential choice for on-demand data retrieval.



Topics vs Services

Topics	Services
Continuous data exchange (e.g., sensor readings, odometry)	On-demand queries (e.g., turning a robot on/off, getting robot state)
One-way (publish)	Two-way (request & response)
Asynchronous (Publisher/Subscriber)	Synchronous (Not recommended), Asynchronous (Recommended)
Multiple nodes can receive the same topic data	Communication only between the client and the server.
Topics continuously publish messages, even when they are not needed.	Services only communicate when needed, saving bandwidth.
Best-effort or reliable (QoS settings)	Always expects a response
Can be used for every process.	They should never be used for longer running processes, in particular processes that might be required to preempt if exceptional situations occur.
Topics have no built-in response verification	allow **error handling** since they can return a response indicating success, failure, or an error message.

Topics vs Services

Use Case	Best Option
Continuous sensor data (e.g., LiDAR, camera)	Topic
Sending control commands (e.g., velocity commands)	Topic
Requesting the robot’s current position	Service
Sending a start/stop command for a task	Service
Broadcasting environmental conditions (e.g., battery level, wheel pressure)	Service

Service interfaces

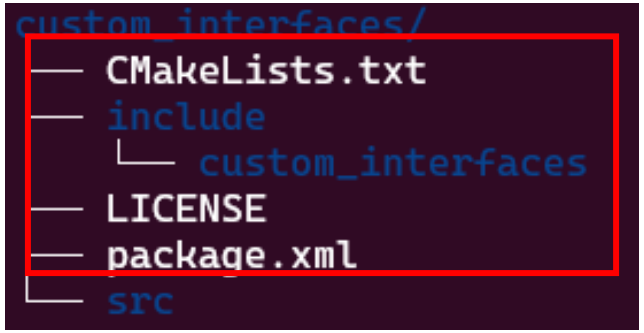
- ROS2 services use a **request-response** communication model.
- A service consists of **two parts**:
  - **Request**: The message (interface) sent by the client.
  - **Response**: The message returned by the server.
- Service interfaces as messages consists of a data structure, where the user represents the request and the response separated by "---"

```
# This is aservice to set a boolean value.
# This can be used for starting a process

bool enable    # Request e.g. for hardware enabling / disabling
---            #Response
bool success   # Response indicate successful run service
string message # informational, e.g. for error messages
```

- Some Service Interfaces are predefined by ROS (here) but normally, they are customised for each need.
- To develop custom interfaces, a package must be made using CMake.

CMake Packages

- As with Python, ROS can develop packages using CMake (C++ based) when building.
  - CMake (ament_cmake): Used when compiling C++-based ROS2 packages.
  - Python (ament_python): Used for Python-based ROS2 packages.
- When building a C++ node using CMake in Colcon, the compiler requires some configurations, dependencies and build rules.
- The CMakeLists and package.xml are the files containing the configuration, dependencies paths and build rules.



- CMake generates configuration used by the compiler files using CMakeLists.txt file.
- More information about CMakeLists when using C++ can be found here and here.
- Examples of CMakeLists files can be found here.

Package.xml File

- Package manifest file.
- Contains the metadata of the package.
- XML File that must be included with any catkin-compliant package.
- Defines the properties about the package such as the package name, version numbers, authors, maintainers, and dependencies on other catkin packages.
- More information about Package.xml files can be found here, here and here.

```xml
<?xml version="1.0"?>
  <package format="2">
  <name>package_name</name>
  <version>1.0.0</version>
  <description>The example package</description>
  <maintainer email="me@todo.todo">linux</maintainer>
  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>

  <depend>rospy</depend>

</package>
```

package.xml File Example

Package.xml File Dependencies

- **exec_depend:** Packages required at runtime. The most common dependency for a python-only package. If your package or launchfile imports/runs code from another package.
- **build_depend**: Package required at build time. Python packages usually do not require this. Some exceptions is when you depend upon messages, services or other packages.
- **build_export_depend:** Specify which packages are needed to build libraries against this package. This is the case when you transitively include their headers in public headers in this package

```
<?xml version="1.0"?>
<package format="2">
  <name>courseworks</name>
  <version>1.0.0</version>
  <description>The courseworks package</description>
  <maintainer email="mario.mtz@manchester-robotics.com">Mario Martinez</maintainer>
  <license>BSD</license>
  <url type="website">http://www.manchester-robotics.com</url>
  <author email="mario.mtz@manchester-robotics.com">Mario Martinez</author>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>numpy</build_depend>

  <build_export_depend>rospy</build_export_depend>
  <build_export_depend>std_msgs</build_export_depend>
  <build_export_depend>numpy</build_export_depend>

  <exec_depend>rospy</exec_depend>
  <exec_depend>std_msgs</exec_depend>
  <exec_depend>numpy</exec_depend>
</package>
```

Package Metadata

Specify build system tools required by the package.

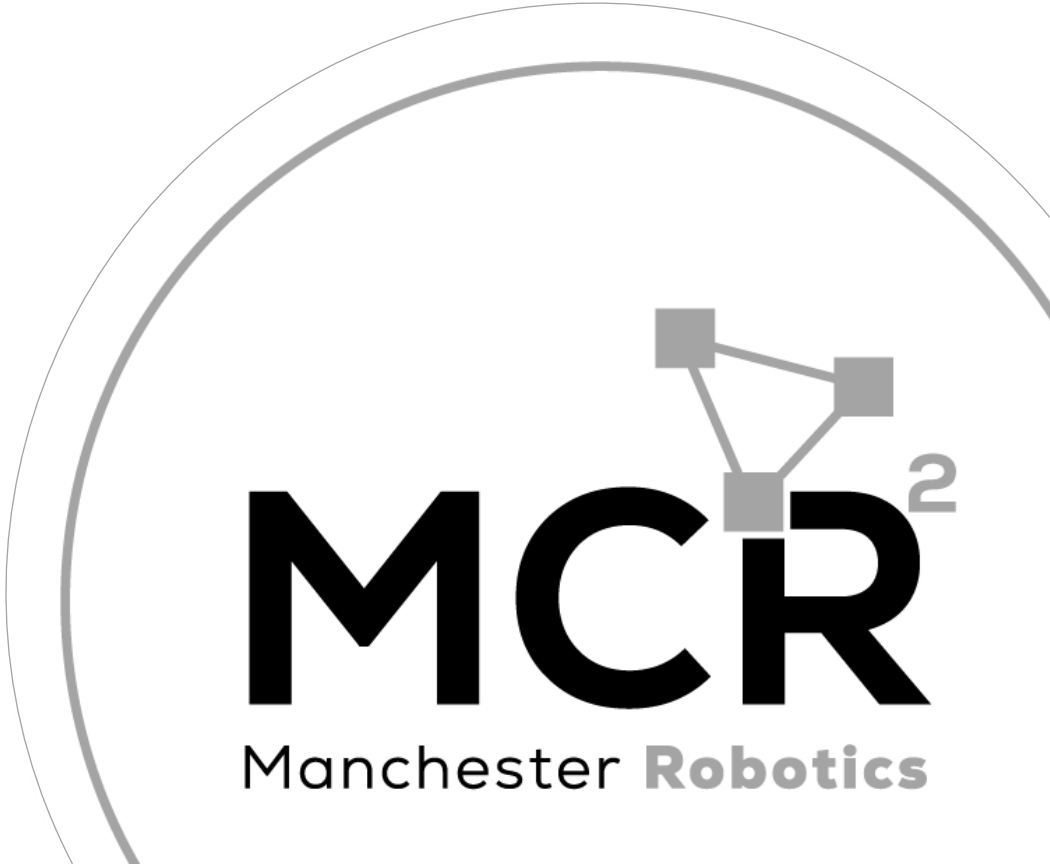
Dependency tags:

- **build_depend:** Dependencies must be present when the code is built.
- **build_export_depend:** Specify which packages are needed to build libraries against this package.
- **exec_depend:** Packages needed to be installed along our package in order to run. Packages required at runtime.

Activity 3

Custom Interface Services

{Learn, Create, Innovate};



Activity 3 – Custom Interface Services (Custom Interface)

Requirements

- For this activity, the previously modified motor_control package will be used.

Objective

- The objective in tis activity is to set a simple service between the set_point generator and the dc_motor node to start and stop the dc_motor node.
- For this activity, a server will be set in the dc_motor node and a client on the set_point node.
- A custom interface will be used for this activity.

Custom Interface Package

- Create a CMake “custom_interfaces” package

```
$ cd ~/ros2_ws/src
$ ros2 pkg create --build-type ament_cmake
custom_interfaces --dependencies std_msgs geometry_msgs
--license Apache-2.0
```

- Make a folder called srv inside the “custom_interfaces” package and a Service interface called SetProcessBool.srv.

```
$ mkdir custom_interfaces/srv
$ touch custom_interfaces/srv/SetProcessBool.srv
```

Activity 3 – Custom Interface Services (Custom Interface)

Custom Interface Package

- Open the SetProcessBool.srv file and write the following.

```
# Custom Interface
# This is a service to set a boolean value.
# This can be used for starting a process

bool enable   #for hardware enabling / disabling
---
bool success   # indicate successful run service
string message # informational
```

- Open CMakeLists.txt and add the following lines and save the file.

```
# find dependencies
find_package(ament_cmake REQUIRED)
find_package(std_msgs REQUIRED)
find_package(geometry_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "srv/SetProcessBool.srv"
  DEPENDENCIES std_msgs
)
```

Activity 3 – Custom Interface Services (Custom Interface)

Custom Interface Package

- Open package.xml, add the following lines and save the file.

```
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

- Build the workspace and source it

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

Results

- Validate that the sphere message has been properly created

```
$ ros2 interface show custom_interfaces/srv/SetProcessBool
```

```
mario@MarioPC:~/ros2_ws$ ros2 interface show custom_interfaces/srv
/SetProcessBool
# This is aservice to set a boolean value.
# This can be used for starting a process

bool enable # e.g. for hardware enabling / disabling
---
bool success   # indicate successful run of triggered service
string message # informational, e.g. for error messages
```

Activity 3 – Custom Interface Services (Server)

Creating a server

- Open the dc_motor.py from motor_control package.
- Import the newly created custom interface at the top.

```
# Imports
import rclpy
from rclpy.node import Node
from std_msgs.msg import Float32
from rcl_interfaces.msg import SetParametersResult
from custom_interfaces.srv import SetProcessBool
```

- Define a Boolean variable “simulation_running” in the constructor.
- Define a server callback function inside the constructor.

```
def __init__(self):
    super().__init__('dc_motor’)
        ...

        self.simulation_running = False

        #Set Server callback
        self.srv = self.create_service(SetProcessBool, 'EnableProcess’,
            self.simulation_service_callback)
```

Activity 3 – Custom Interface Services (Server)

Define the Server Callback

- Create the function "simulation_service_callback" within the class.
- This function will update the variable "simulation_running".

```
    # Service Callback to Start/Stop Simulation
    def simulation_service_callback(self, request, response):
        if request.enable:
            self.simulation_running = True
            self.get_logger().info("🚀 Simulation Started")
            response.success = True
            response.message = "Simulation Started Successfully"
        else:
            self.simulation_running = False
            self.get_logger().info("🛑 Simulation Stopped")
            response.success = True
            response.message = "Simulation Stoped Successfully"

        return response
```

- Inside the timer callback use the "simulation_running" variable to stop the DC_Motor Simulation.

```
    #Timer Callback
    def timer_cb(self):

        if not self.simulation_running:
            return  # Stop processing if simulation is not running

        #DC Motor Simulation
        #DC Motor Equation y[k+1] = y[k] + ((-1/τ) y[k] + (K/τ) u[k]) T_s
        ...
```

Activity 3 – Custom Interface Services (Server)

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
```

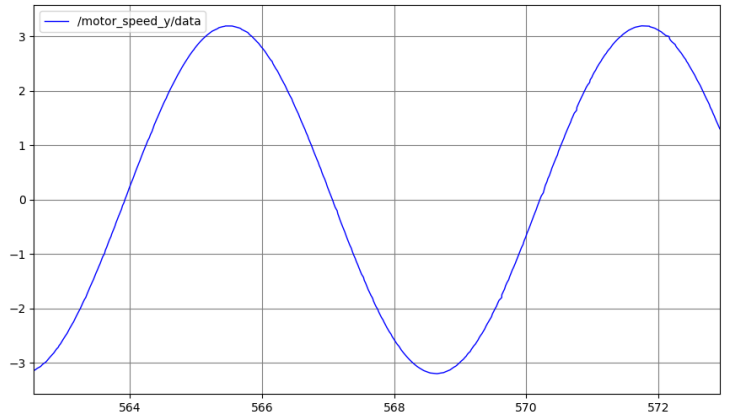
- Open the rqt_plot and verify the output signal

```
$ ros2 run rqt_plot rqt_plot
```

Server Results

- Open a new terminal (when calling a service always source the terminals)

```
$ source install/setup.bash
$ ros2 service call /EnableProcess
custom_interfaces/srv/SetProcessBool '{enable: true}'

$ ros2 service call /EnableProcess
custom_interfaces/srv/SetProcessBool '{enable: false}'
```



ROS Services - Command Line

Services Command Line

- To verify type of a service

```
$ ros2 service type <service_name>
```

- To see the types of all the active services

```
$ ros2 service list -t
```

- To find all the services of a specific type

```
$ ros2 service find <type_name>
```

- To know the structure of the input arguments of a service

```
$ ros2 interface show <type_name>
```

- To call a service

```
$ ros2 service call <service_name>
<service_type> <arguments>
```

Activity 3 – Custom Interface Services (Client)

Define a Client

- Open the set_point.py inside motor_control package.
- Import the newly created custom interface at the top.

```
# Imports
import rclpy
from rclpy.node import Node
import numpy as np
from std_msgs.msg import Float32
from custom_interfaces.srv import SetProcessBool
```

Define a Client

- Create a client “cli” and wit for the service to be available.
- Define a send request function (to be created) in the constructor.
- Define a variable “system_running in the constructor as well.”

```
def __init__(self):
    super().__init__('dc_motor’)
        ...

        self.system_running = False

        #Create a service client for /EnableProcess
        self.cli = self.create_client(SetProcessBool, 'EnableProcess')
        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('service not available, waiting again...’)


        ...
        self.send_request(True)
```

Activity 3 – Custom Interface Services (Client)

Define a Client

- This function sends a request to the server, as True in the enable variable from the custom interface.
- *future* is an asynchronous task that will eventually contain the result of the service call.
- It allows non-blocking execution, meaning the program can continue running while waiting for the service response.
- When the service call is completed, the result will
- Adds a callback to the response to make the server communication asynchronous.

```
def send_request(self, enable: bool):
    request = SetProcessBool.Request()
    request.enable = enable

    #Send a request to start or stop the simulation
    future = self.cli.call_async(request)
    future.add_done_callback(self.response_callback)
```

custom interface

```
bool enable    #Request
---
bool success    # Response
string message
```

Activity 3 – Custom Interface Services (Client)

Define a Client Callback function

- This function process the callback from the server, once the server has finished its task.
- The callback sets the variable “system_running” to True and logs the message received by the server in the Response “string message” of the interface.

```python
def response_callback(self, future):
    """Process the service response."""
    try:
        response = future.result()
        if response.success:
            self.system_running = True
            self.get_logger().info(f'Success: {response.message}')
        else:
            self.simulation_running = False
            self.get_logger().warn(f'Failure: {response.message}')
    except Exception as e:
        self.simulation_running = False
        self.get_logger().error(f'Service call failed: {e}')
```

Activity 3 – Custom Interface Services (Client)

- Using the “system_running” variable, it is possible now to activate the “set_point” node the same way as the “dc_motor” node.

```
# Timer Callback: Generate and Publish Sine Wave Signal
def timer_cb(self):
    if not self.system_running:
        return  # Stop processing if simulation is not running
...
```

- The node will activate after the process node is active, giving us more control on the order/sequence on how the nodes activate.

1. set_point and dc_motor nodes are launched (without order)
2. set_point node, does not start publishing (idle) (system_running = False).
3. set_point node sends a request to the server (dc_motor node).
4. dc_motor node receives request and starts publishing.
5. dc_motor node sends response to the set_point node.
6. set_point node receives response, process it and starts publishing.

Activity 3 – Custom Interface Services (Client)

Instructions

- Save and compile the file

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

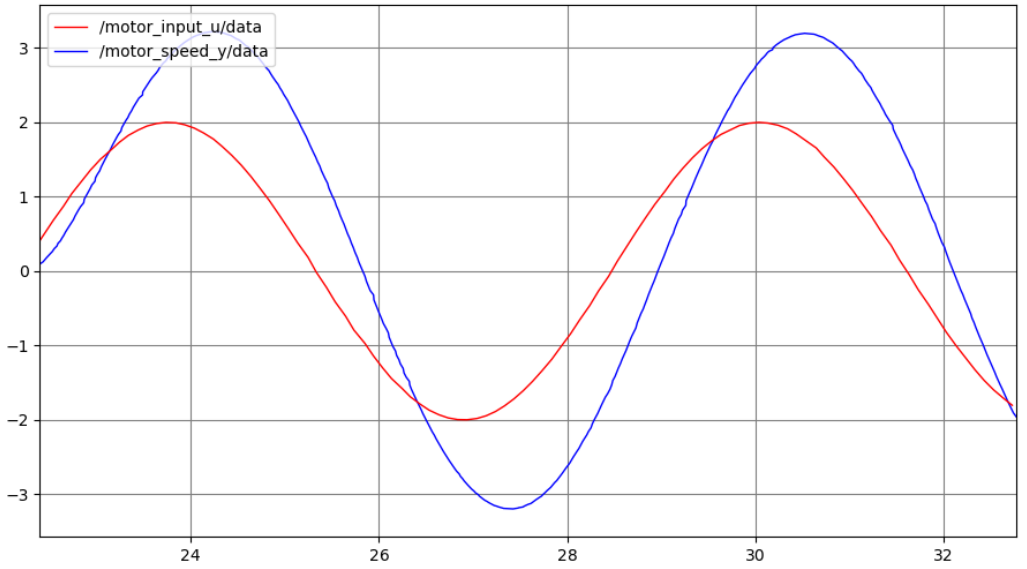
- Launch the node

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

- Open the rqt_plot and verify the output signal

```
$ ros2 run rqt_plot rqt_plot
```

Results



Symbol	Code	Meaning
✅	`\u2705`	Check mark (Success)
❌	`\u274C`	Cross mark (Error)
🚀	`\U0001F680`	Rocket (Launch)
⚙	`\U00002699`	Gear (Processing)
⏳	`\U000023F3`	Hourglass (Time)
🔴	`\U0001F534`	Red circle (Stop)
🟢	`\U0001F7E2`	Green circle (Start)
⚡	`\u26A1`	Lightning (Power)
🔥	`\U0001F525`	Fire (Performance)
📡	`\U0001F4E1`	Signal (Communication)

Q&A

Questions?

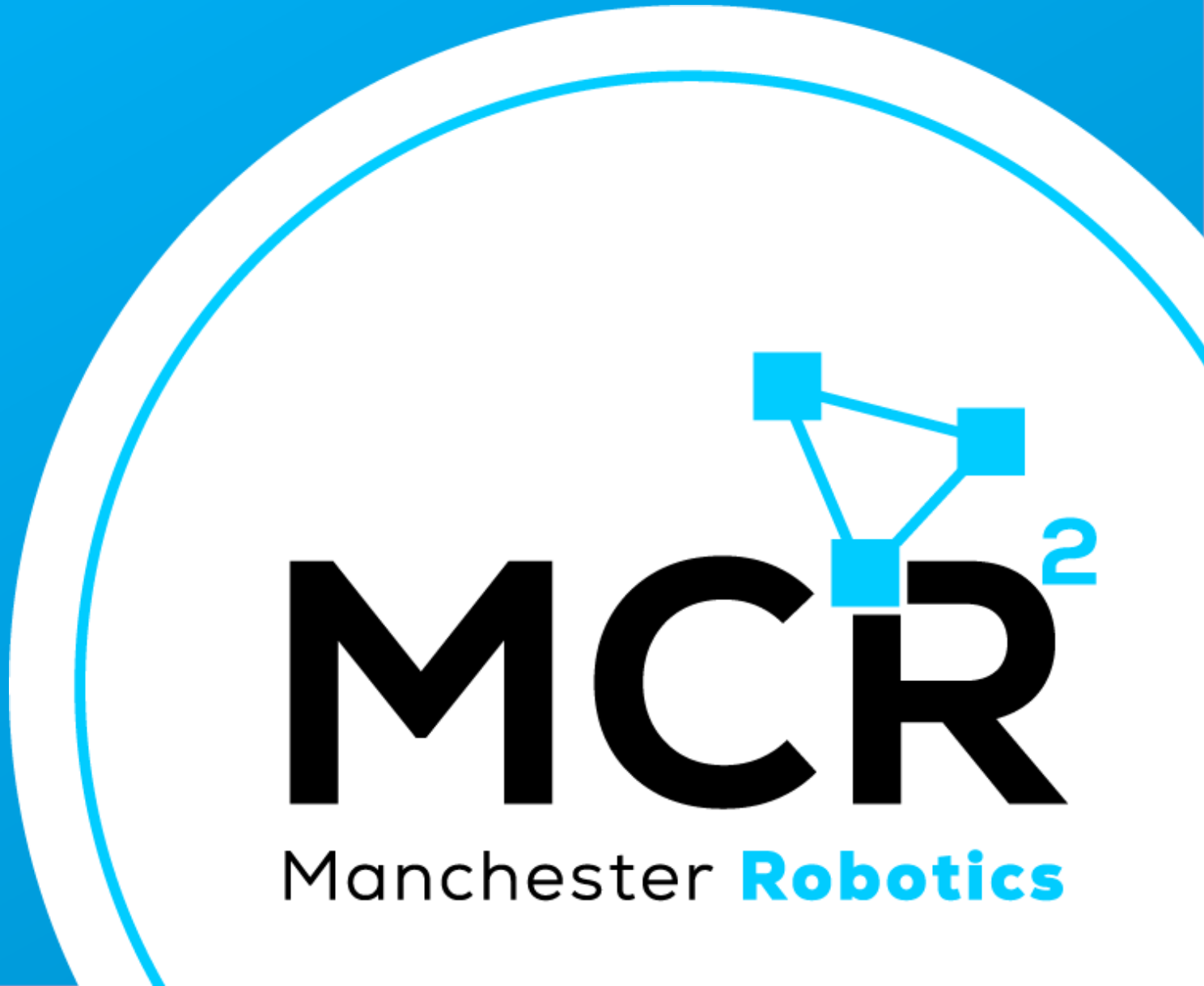
{Learn, Create, Innovate};

MCR² Manchester Robotics

Thank You

Robotics For Everyone

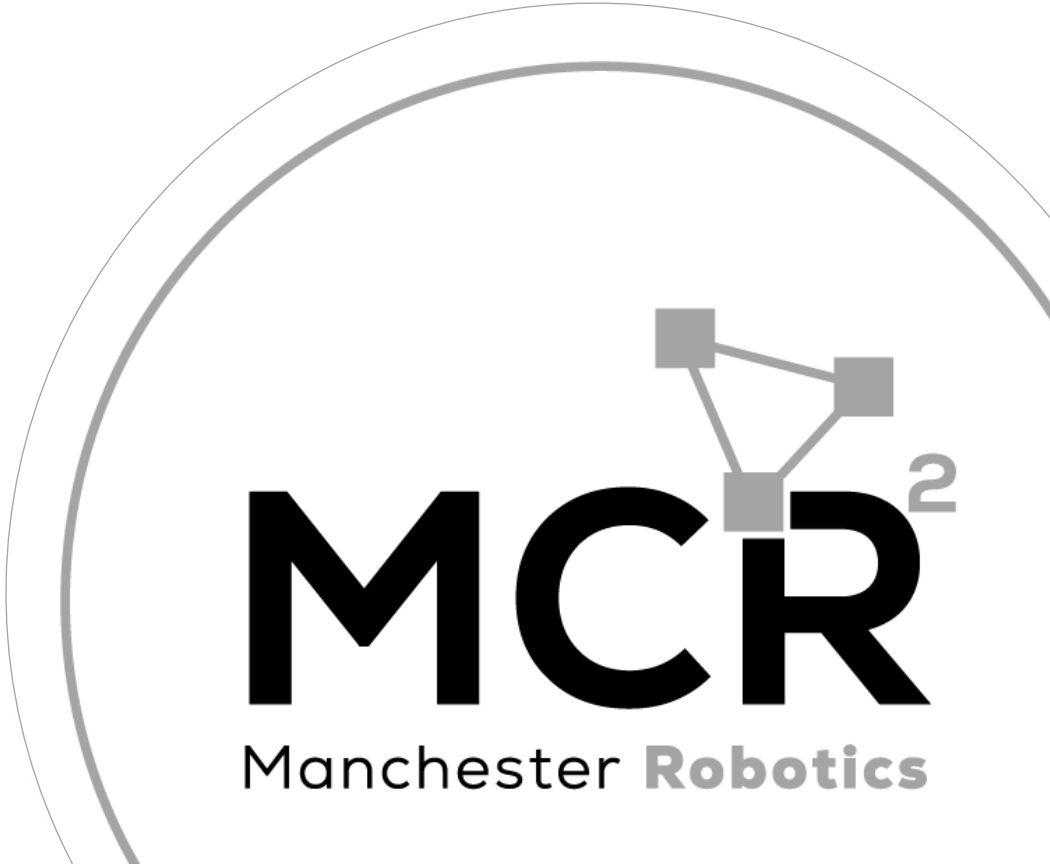
{Learn, Create, Innovate};



T&C

Terms and conditions

{Learn, Create, Innovate};



Terms and conditions

- *THE PIECES, IMAGES, VIDEOS, DOCUMENTATION, ETC. SHOWN HERE ARE FOR INFORMATIVE PURPOSES ONLY. THE DESIGN IS PROPRIETARY AND CONFIDENTIAL TO MANCHESTER ROBOTICS LTD. (MCR2). THE INFORMATION, CODE, SIMULATORS, DRAWINGS, VIDEOS PRESENTATIONS ETC. CONTAINED IN THIS PRESENTATION IS THE SOLE PROPERTY OF MANCHESTER ROBOTICS LTD. ANY REPRODUCTION, RESELL, REDISTRIBUTION OR USAGE IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF MANCHESTER ROBOTICS LTD. IS STRICTLY PROHIBITED.*
- *THIS PRESENTATION MAY CONTAIN LINKS TO OTHER WEBSITES OR CONTENT BELONGING TO OR ORIGINATING FROM THIRD PARTIES OR LINKS TO WEBSITES AND FEATURES IN BANNERS OR OTHER ADVERTISING. SUCH EXTERNAL LINKS ARE NOT INVESTIGATED, MONITORED, OR CHECKED FOR ACCURACY, ADEQUACY, VALIDITY, RELIABILITY, AVAILABILITY OR COMPLETENESS BY US.*
- *WE DO NOT WARRANT, ENDORSE, GUARANTEE, OR ASSUME RESPONSIBILITY FOR THE ACCURACY OR RELIABILITY OF ANY INFORMATION OFFERED BY THIRD-PARTY WEBSITES LINKED THROUGH THE SITE OR ANY WEBSITE OR FEATURE LINKED IN ANY BANNER OR OTHER ADVERTISING.*