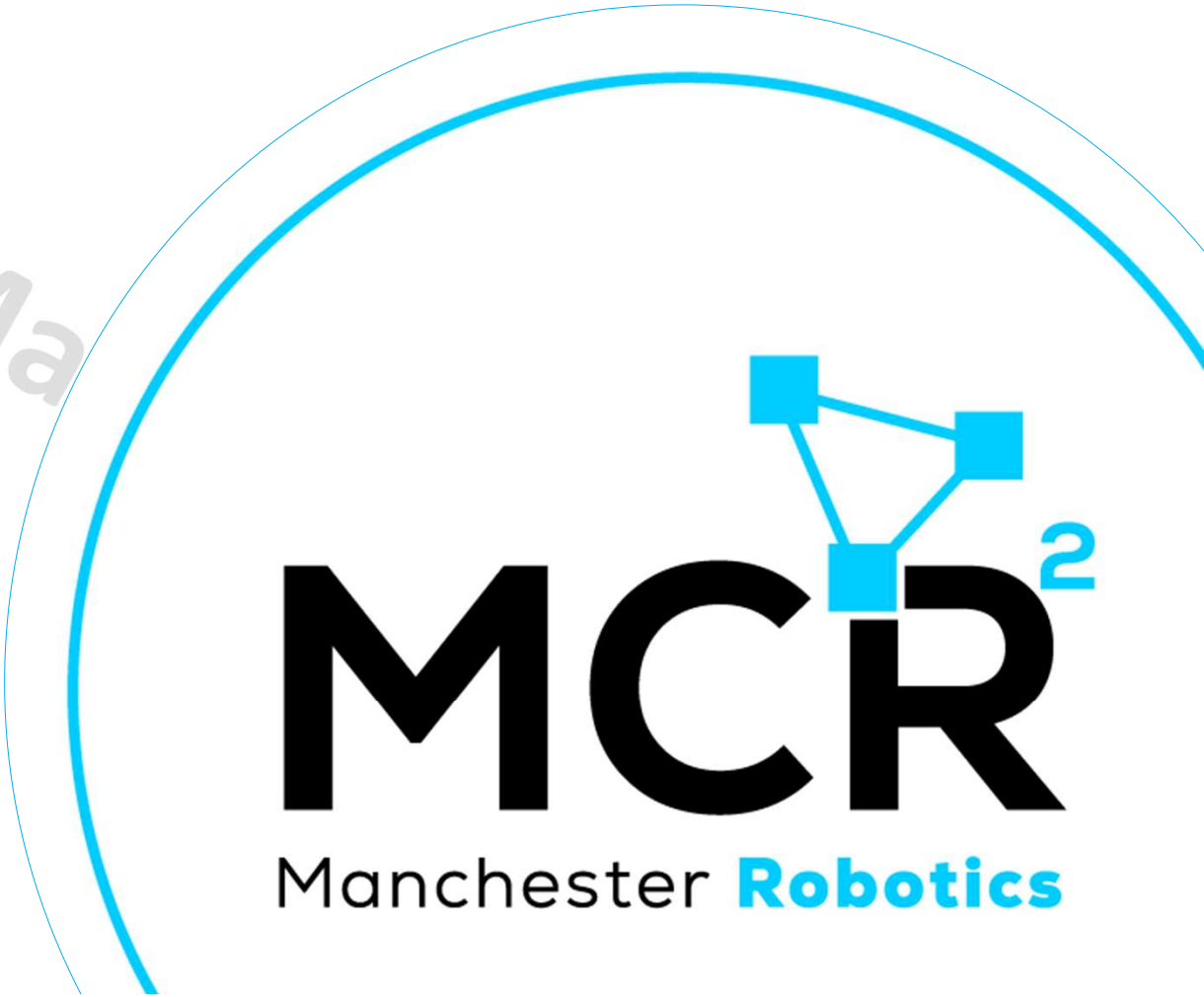
{Learn, Create, Innovate};

Robot Operating System – ROS

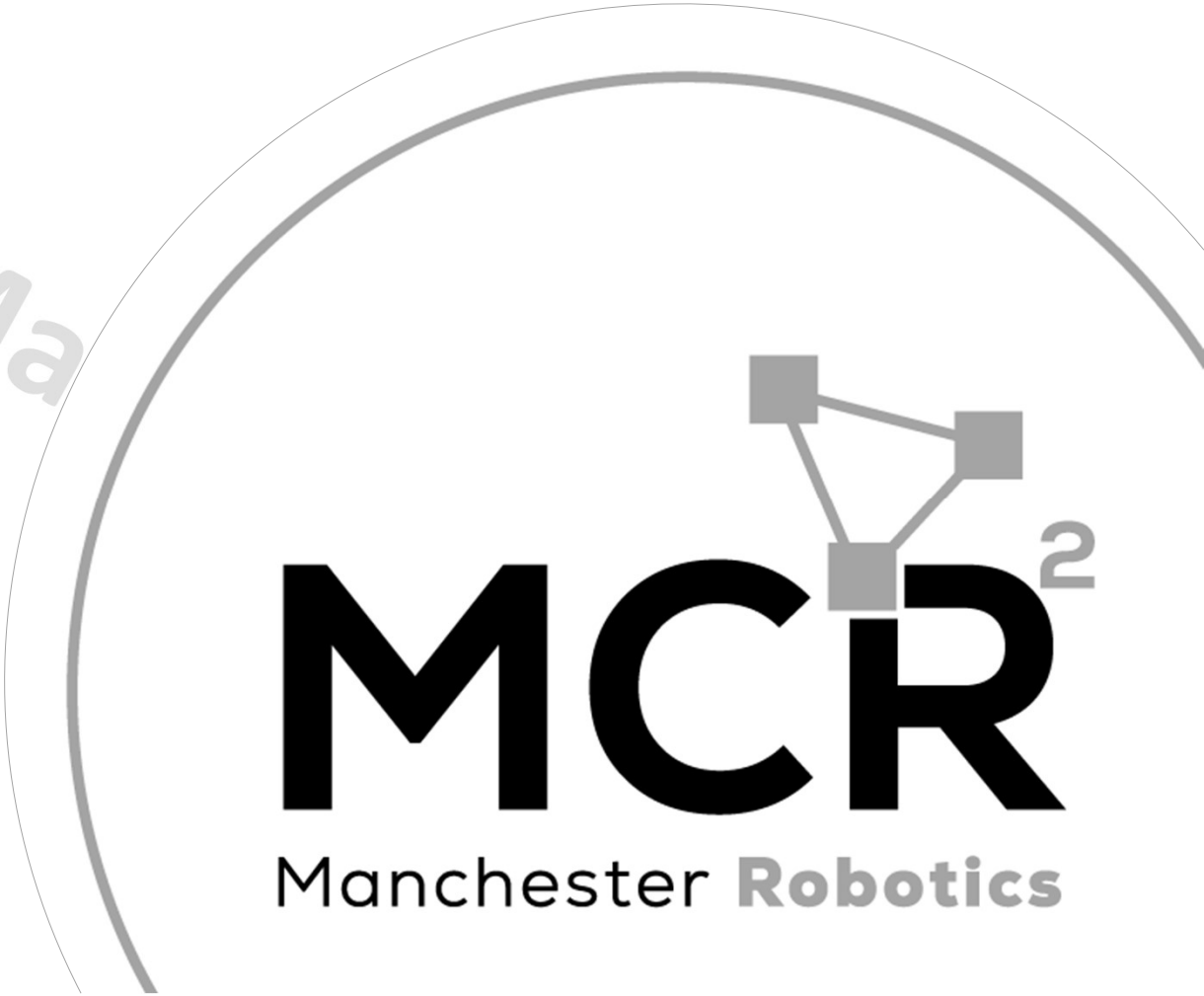
Practicalities



Robot Operating System – ROS

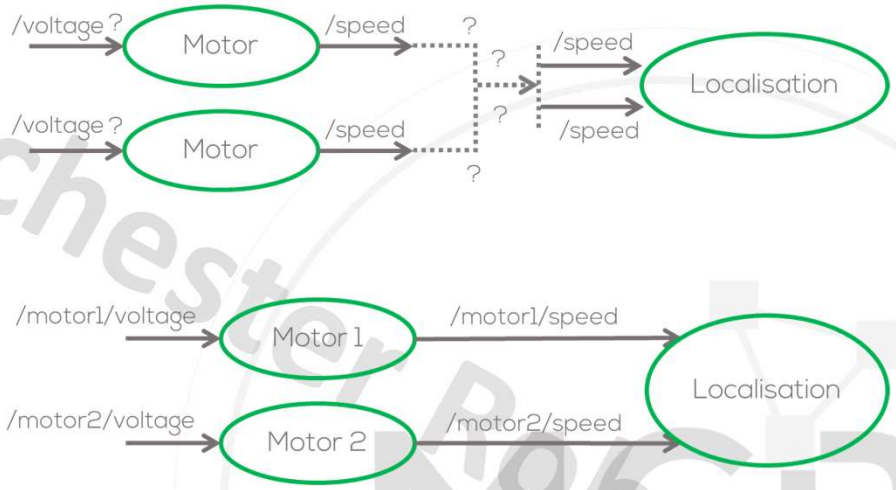
Namespaces

{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 way to define the namespaces. The easiest way is via command line, which is very easy but for larger projects is not recommended.
- In this presentation, the launch file will be used to define the namespaces.

Activity 1 – ROS Namespaces

- Namespaces in ROS Example
- For this example, two talker nodes and two listener nodes will be generated using namespaces.
- Create a *chatter_group_launch.py* file in the launch folder of the *basic_comms* package.

```
$ cd ~/ros2_ws/src/basic_comms/launch
$ touch chatter_group_launch.py
$ chmod +x chatter_group_launch.py
```

ROS2 Launch file – includes and groups

```
import os
from ament_index_python import get_package_share_directory
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription, GroupAction, DeclareLaunchArgument
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch_ros.actions import Node, PushRosNamespace
from launch.substitutions import LaunchConfiguration, TextSubstitution
```

Imports

```
def generate_launch_description():

    chatter_ns_launch_arg = DeclareLaunchArgument(
        "chatter_group", default_value=TextSubstitution(text="my/chatter/ns")
    )

    launch_include = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(
                get_package_share_directory('basic_comms'),
                'launch/chatter.launch.py'))
    )

    launch_include_with_namespace = GroupAction(
        actions=[
            # push-ros-namespace to set namespace of included nodes
            PushRosNamespace(LaunchConfiguration('chatter_group')),
            IncludeLaunchDescription(
                PythonLaunchDescriptionSource(
                    os.path.join(
                        get_package_share_directory('basic_comms'),
                        'launch/chatter.launch.py'))
            ),
        ]
    )
```

Launch body

```
    l_d = LaunchDescription([chatter_ns_launch_arg, launch_include, launch_include_with_namespace])
    return l_d
```

Set launch content

ROS Namespaces

- In another terminal source the workspace and execute the launch file

```
$ source install/setup.bash
$ ros2 launch basic_comms chatter_group_launch.py
```

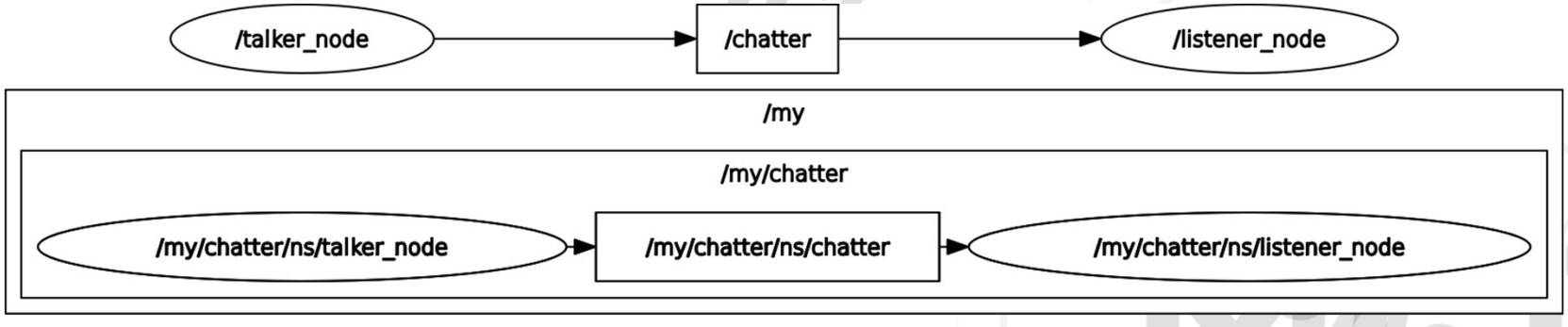
- In another terminal list the topics

```
$ ros2 topic list
```

```
/chatter
/my/chatter/ns/chatter
/parameter_events
/rosout
```

- Finally, in a new terminal, execute the rqt_graph to visualise the nodes

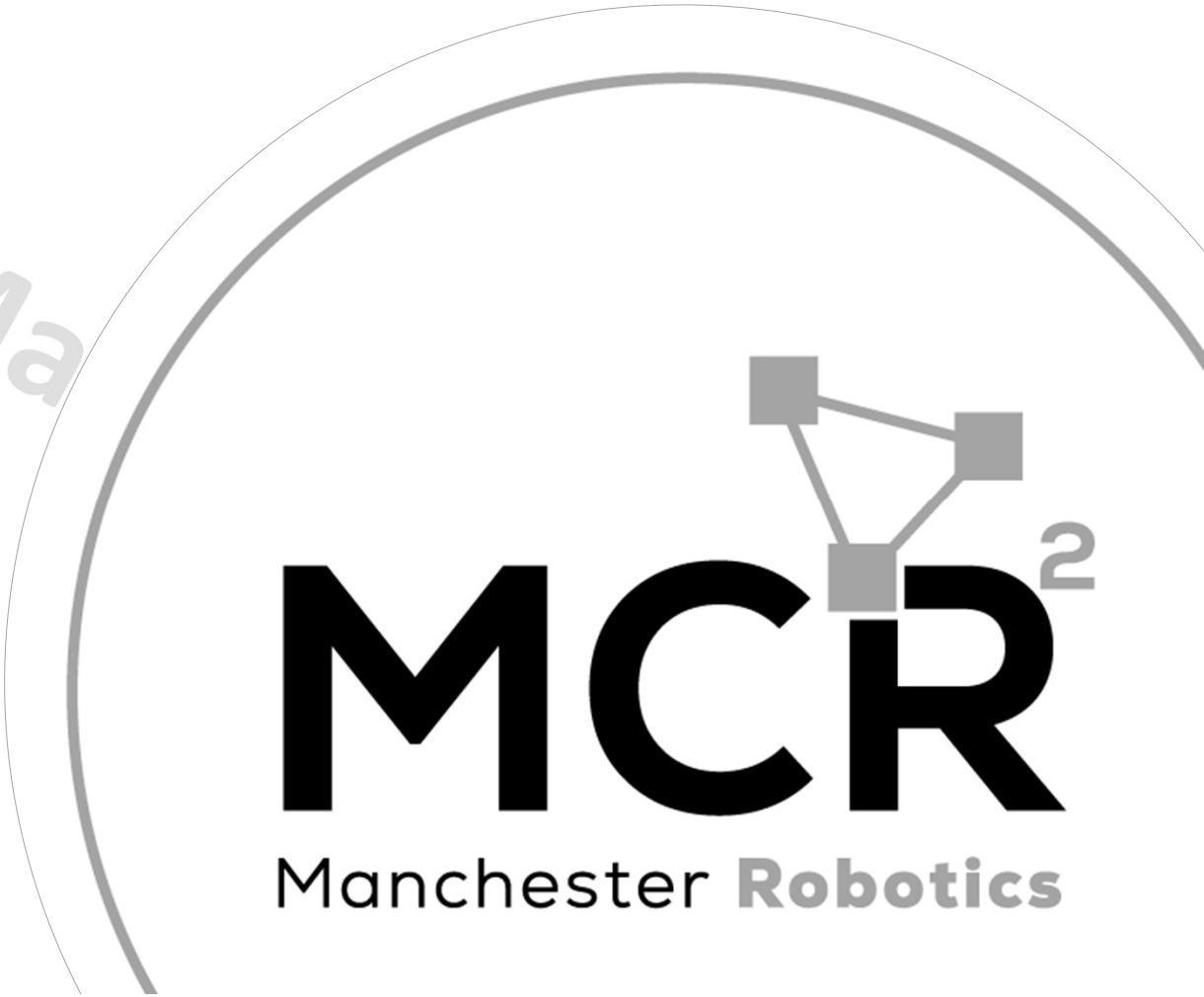
```
$ rosrun rqt_graph rqt_graph
```



Robot Operating System – ROS

Parameter Files

{Learn, Create, Innovate};

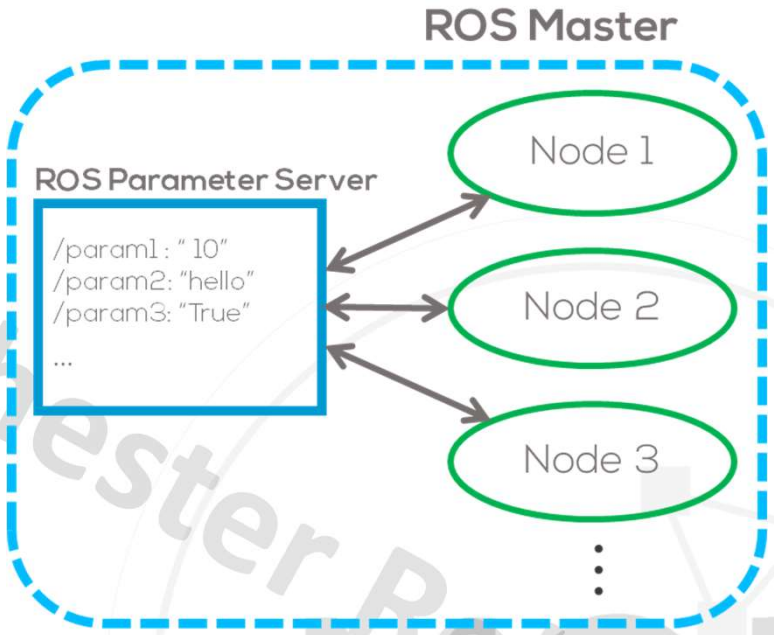


ROS Parameters

- 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.
- To this end, ROS uses a “dictionary” to store and share the parameters to be used by its nodes. This dictionary is called Parameter Server.

ROS Parameters

- As stated before, ROS allows to load variables into a server, run by the master which is accessible by all the nodes of the system.
- Nodes use this server to store and retrieve parameters at runtime.
- These parameters, are used to load settings, robot constants, or other data that may be used in different scenarios where the same code is applied.
- Parameters are composed of a name and a datatype. ROS can only use determined types of parameters.



- Intengers
- Booleans
- Byte arrays
- Strings
- Floats
- Arrays

Parameters

- Configuration values of a node (node settings).
- Parameters are used to configure nodes at startup (and during runtime), without changing the code.
- To list the parameters belonging to available nodes
  - ros2 param list
- To display the type and current value of a parameter
  - 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>

Parameters

- 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 – Parameters

- Create a package called *basic_comms_params*, and create a *talker_params.py* file

```
  $ cd ~/ros2_ws/src/
  $ ros2 pkg create --build-type ament_python --license Apache-2.0 basic_comms_params --
dependencies rclpy
  $ touch basic_comms_params/basic_comms_params/talker.py
  $ chmod +x basic_comms_params/basic_comms_params/talker_params.py
```

Activity – Parameters


```
import rclpy
from rclpy.node import Node

class MinimalParam(Node):
    def __init__(self):
        super().__init__('minimal_param_node')
        self.declare_parameter('my_parameter', 'world')
        self.timer = self.create_timer(1, self.timer_callback)

    def timer_callback(self):
        my_param = self.get_parameter('my_parameter').get_parameter_value().string_value
        self.get_logger().info('Hello %s!' % my_param)

        my_new_param = rclpy.parameter.Parameter(
            'my_parameter',
            rclpy.Parameter.Type.STRING,
            'world'
        )
        all_new_parameters = [my_new_param]
        self.set_parameters(all_new_parameters)

def main():
    rclpy.init()
    node = MinimalParam()
    rclpy.spin(node)

if __name__ == '__main__':
    main()
```

Activity – Parameters

- Add an entry point in the setup.py file

```
entry_points={
        'console_scripts': [
            'talker_params = basic_comms_params.talker_params:main'
        ],
```

- Perform a colcon build on the workspace
- Open two terminals and source the workspace

Parameters via console

Terminal 1

- Run the node with
  - ros2 run basic_comms_params talker_params

Terminal 2

- Check the available params
  - ros2 param list
- Change the parameter with the console
  - ros2 param set /minimal_param_node my_parameter friends

Parameters via a launch file

- Create a *chatter_params_launch.py* file in the launch folder of the *basic_comms_params* package.

```
$ cd ~/ros2_ws/src/basic_comms_params/launch
$ touch chatter_params_launch.py
$ chmod +x chatter_params_launch.py
```

Parameters via a launch file

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

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='basic_comms_params',
            executable='talker_params',
            output='screen',
            emulate_tty=True,
            parameters=[
                {'my_parameter': 'buddies'}
            ]
        )
    ])
```

Parameters via a launch file

- Modify the setup.py file to include the launch files

```
data_files=[
    # ...
    (os.path.join('share', package_name), glob('launch/*launch.[pxy][yma]*')),
  ]
```

- Build the workspace, source it, and launch the node we just created.

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

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.
- These types of files are configuration files, written in YAML. 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.

```
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

- 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).

Activity – Parameter file

```
$ cd ~/ros2_ws/
$ touch src/basic_comms_params/launch/node_params_launch.py
$ chmod +x src/basic_comms_params/launch/ node_params_launch.py

$ mkdir src/basic_comms_params/config
$ touch src/basic_comms_params/config/params.yaml

$ touch src/basic_comms_params/basic_node_params/node_params.py
$ chmod +x src/basic_comms_params/basic_node_params/node_params.py
```

- Add the following line to the console scripts in the setup.py file
  - `'node_params = basic_comms_params.node_params:main'`

Parameter file

```yaml
example_node_params:
  ros__parameters:
    bool_value: True
    int_number: 5
    float_number: 3.14
    str_text: "Hello Universe"
    bool_array: [True, False, True]
    int_array: [10, 11, 12, 13]
    float_array: [7.5, 400.4]
    str_array: ['Nice', 'more', 'params']
    bytes_array: [0x01, 0xF1, 0xA2]
    nested_param:
      another_int: 7
      another_float: 7.0
```

Load a YAML config file for a node

```
import rclpy
from rclpy.node import Node

class TestYAMLParams(Node):

    def __init__(self):
        super().__init__('example_node_params')
        self.declare_parameters(
            namespace='',
            parameters=[
                ('bool_value', rclpy.Parameter.Type.BOOL), ('int_number', rclpy.Parameter.Type.INTEGER),
                ('float_number', rclpy.Parameter.Type.DOUBLE), ('str_text', rclpy.Parameter.Type.STRING),
                ('bool_array', rclpy.Parameter.Type.BOOL_ARRAY), ('int_array', rclpy.Parameter.Type.INTEGER_ARRAY),
                ('float_array', rclpy.Parameter.Type.DOUBLE_ARRAY), ('str_array', rclpy.Parameter.Type.STRING_ARRAY),
                ('bytes_array', [0x00]), ('nested_param.another_int', rclpy.Parameter.Type.INTEGER )
            ])

def main(args=None):
    rclpy.init(args=args)
    node = TestYAMLParams()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

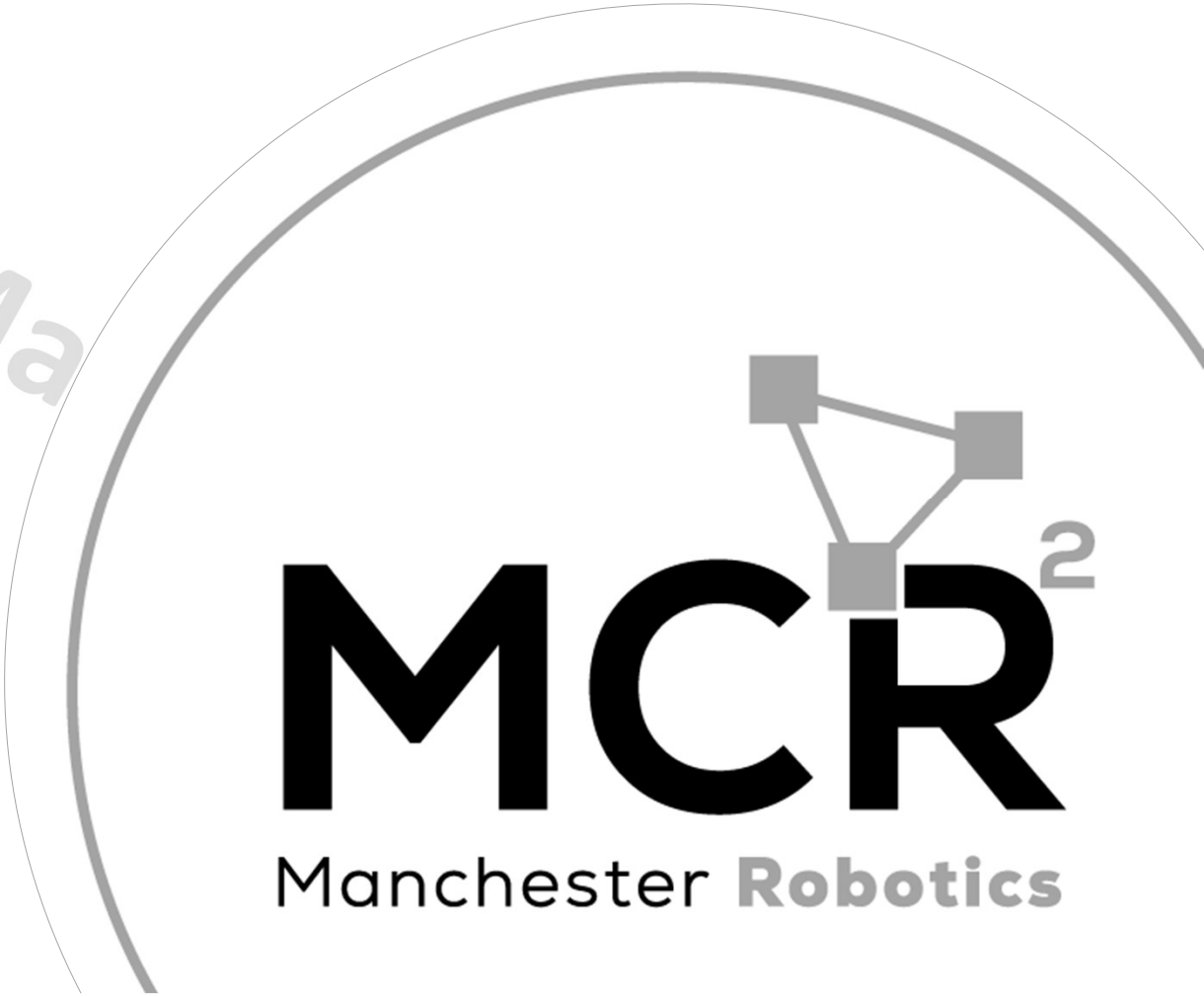
Launch file

```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    ld = LaunchDescription()

    config = os.path.join(
        get_package_share_directory('basic_comms_params'),
        'config',
        'params.yaml'
        )

    node=Node(
        package = 'basic_comms_params',
        name = 'example_node_params',
        executable = 'node_params',
        parameters = [config]
    )

    ld.add_action(node)
    return ld
```

Robot Operating System – ROS

Custom Messages

{Learn, Create, Innovate};



Configuration Files

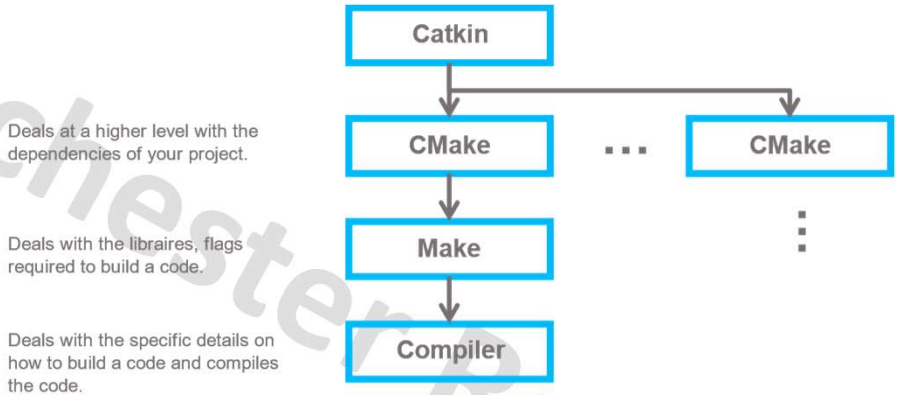
- Packages are the “buildable and redistributable units” of ROS code. Therefore, to build a package it is necessary to build the entire package as one piece.
- In most projects, the user would require to build multiple packages to group nodes, libraries and other resources for different purposes/functionalities.
- Colcon requires different configuration files to properly define the dependencies and properties (meta information) for the different packages.
- Such packages are called CMakeLists.txt and package.xml

Catkin and CMakeLists Files

- Compilers and executables usually require libraries, external files or flags to be able to compile and run a program.
- Build tools, describe the setup of the project (set of rules and commands) at a user level to generate the specific flags and paths required by the program to run.
- In Linux these build tools are commonly known as Make files.
- Make files sometimes require information from the system, like paths to libraries, dependencies, targets to be built, locations, etc.
- When sharing a project, the paths to those libraries may change.
- CMake is a tool (build system generator) that allows the user to deal with the issue by searching for the library paths and generating the flags in other machines to generate the make file.
  - CMake tool only works with one project at a time.
- CMake uses CMakeLists files to work.

Catkin and CMake Files

- ROS is a very large collection of packages. That means lots of independent packages which depend on each other, utilize various programming languages, tools, and code organization conventions.
- The build process for a target in some package may be completely different from the way another target is built.
- To coordinate all the CMake files across projects the Colcon tool was created. Colcon is the official build system of ROS
- Colcon specifically tries to improve development on large sets of related packages in a consistent and conventional way.



CMakeLists Structure


```
# Declare the version of the CMake API for forward-compatibility
cmake_minimum_required(VERSION 2.8)

# Declare the name of the CMake Project
project(hello_world_tutorial)

# Find Catkin
find_package(catkin REQUIRED)
# Declare this project as a catkin package
catkin_package()

# Find and get all the information about the roscpp package
find_package(roscpp REQUIRED)

# Add the headers from roscpp
include_directories(${roscpp_INCLUDE_DIRS})
# Define an executable target called hello_world_node
add_executable(hello_world_node hello_world_node.cpp)
# Link the hello_world_node target against the libraries used by roscpp
target_link_libraries(hello_world_node ${roscpp_LIBRARIES})
```

CMakeLists.txt Example

- **Required CMake Version** (cmake_minimum_required)
- **Package Name** (project())
- **Find other CMake/Catkin packages needed for build** (find_package())
- **Enable Python module support** (catkin_python_setup())
- **Message/Service/Action Generators** (add_message_files(), add_service_files(), add_action_files())
- **Invoke message/service/action generation** (generate_messages())
- **Specify package build info export** (catkin_package())
- **Libraries/Executables to build** (add_library()/add_executable()/target_link_libraries())
- **Tests to build** (catkin_add_gtest())
- **Install rules** (install())

CMake Files

- When using Python, compared with C++, CMakeLists files are used to define the list of dependencies or modules, create custom messages, installation rules or when using a client/server model.
- When using C++, CmakeLists can become more “complex” because you will need to declare more parameters, flags and libraries into the file.
- More information about CMakeLists when using C++ can be found here and here.
- Examples of CMakeLists files can be found here.

CMake Files

```
cmake_minimum_required(VERSION 3.0.2)
project(courseworks)

## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  rospy
  std_msgs
)

## Uncomment this if the package has a setup.py. This macro ensures
## modules and global scripts declared therein get installed
# catkin_python_setup()

catkin_package(
#  INCLUDE_DIRS
#  LIBRARIES
CATKIN_DEPENDS rospy std_msgs
#  DEPENDS
)

## Specify additional locations of header files
## Your package locations should be listed before other locations
include_directories(
# include
  ${catkin_INCLUDE_DIRS}
)

## Mark executable scripts (Python etc.) for installation
## in contrast to setup.py, you can choose the destination
 catkin_install_python(PROGRAMS
   scripts/signal_generator.py scripts/process.py
   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
 )
```

CMake Version and project name (same name as in package.xml)

List of packages required by your package to be built (must be declared in the package.xml)

Install python modules/scripts the current package or from other packages (requires a setup.py file)

Specify Build Export Information:
INCLUDE_DIRS: Dirs. With header files.
LIBRARIES: Libraries created for this project.
CATKIN_DEPENDS: Packages dependent projects also require.
DEPENDS: System dependencies, dependent projects require (listed in package.xml)

Specify locations of header files (if required) not for python.

Installation of the scripts

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

Package.xml File Dependencies

```
<?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:

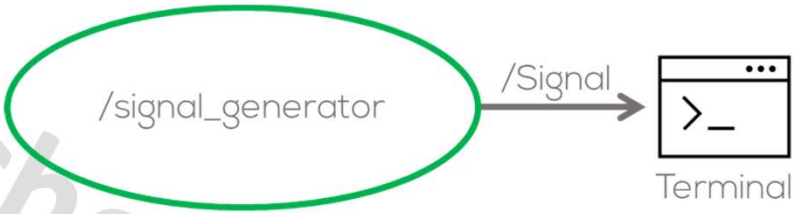
- **build_depend:** Dependencies must be present when the code is built.
- **build_export_depend:** Specify which packages are needed to build libraries against this package.
- **exec_depend:** Packages needed to be installed along our package in order to run. Packages required at runtime.

ROS Custom messages

- ROS has some predefined messages like the std_messages, geometric_messages, etc.
- Sometimes, the message structures are required to be altered for a custom application.
- ROS allows the user to customise the messages and create new messages.
- Custom messages are a way to personalise your own messages for a specific purpose or application.
- Custom messages are created by the user, and must be linked to the package where they will be used.
- More information here, here and here .

ROS Custom messages

- For this example, a new message will be generated to contain two different values of data.
- Two types of message will be used for the message.
- For this example, only one topic will be necessary, and the message will include the information that describes the position and radius of a sphere




```
Custom msg signal_msg
Float32 time_x
Float32 signal_y
```

Activity – Message interfaces

- Navigate to your workspace and go to the src folder
- Create a new package

```
$ cd ~/ros2_ws/src/
$ ros2 pkg create --build-type ament_cmake --license Apache-2.0 srv_int
$ cd srv_int
$ mkdir msg
$ touch msg/Sphere.msg
```

- Open the Sphere.msg file and write the following

```
geometry_msgs/Point center
float64 radius
```

Activity – Message interfaces

- Open CMakeLists.txt and add the following lines

```
find_package(geometry_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
"msg/Sphere.msg"
DEPENDENCIES geometry_msgs
)
```

- Do the same inside package.xml

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

Activity – Message interfaces

- Build the workspace and source it

```
$ colcon build
$ source install/setup.bash
```

- Validate that the sphere message has been properly created

```
$ ros2 interface show srv_int/msg/Sphere
```

Activity – Message interfaces

- Navigate to your workspace and go to the src folder
- Create a new package

```
  $ cd ~/ros2_ws/src/
  $ ros2 pkg create --build-type ament_python --license Apache-2.0 sphere_msgs --
node-name sphere_node --dependencies geometry_msgs srv_int
```

- Open the sphere_node.py file and write the following

```
import rclpy
from rclpy.node import Node
from srv_int.msg import Sphere
import numpy as np

class CustomMsgPublisher(Node):
    def __init__(self):
        super().__init__('custom_msg_node')
        self.pub = self.create_publisher(Sphere, 'sphere_position', 10)
        self.timer_period = 0.1
        self.timer = self.create_timer(self.timer_period, self.timer_callback)
        self.x = 0
        self.y = 0
        self.z = 0
        self.r = 1.5
        self.time = 0
        self.get_logger().info('Custom Sphere msg initialized')

    def timer_callback(self):
        msg = Sphere()
        self.r = 1.5 + 0.5*np.sin(self.time)
        self.time += self.timer_period
        msg.radius = self.r
        self.pub.publish(msg)


def main(args=None):
    rclpy.init(args=args)
    c_m_p = CustomMsgPublisher()
    rclpy.spin(c_m_p)
    c_m_p.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```


Q&A

Questions?

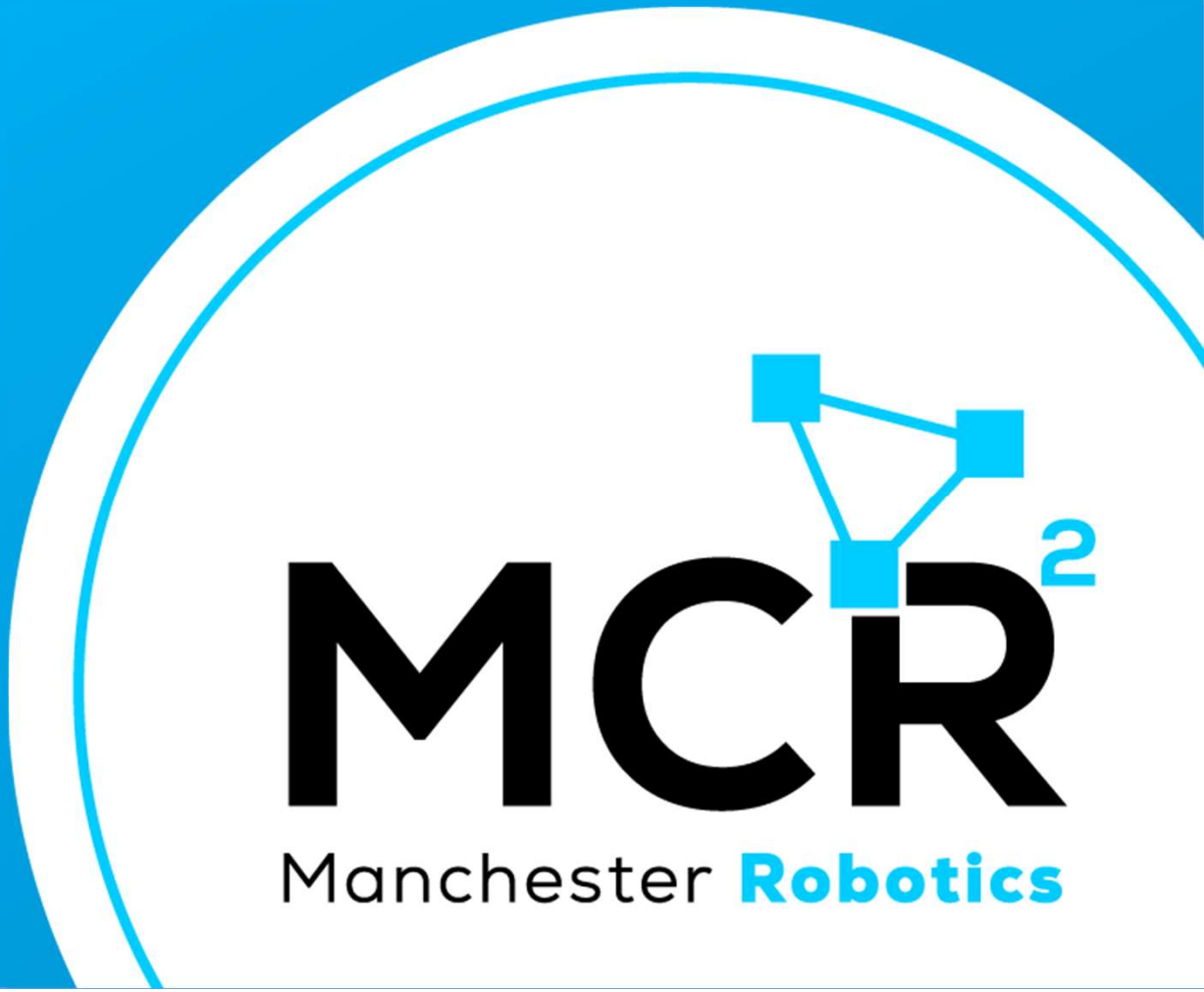
{Learn, Create, Innovate};

MCR² Manchester Robotics

Thank You

Robotics For Everyone

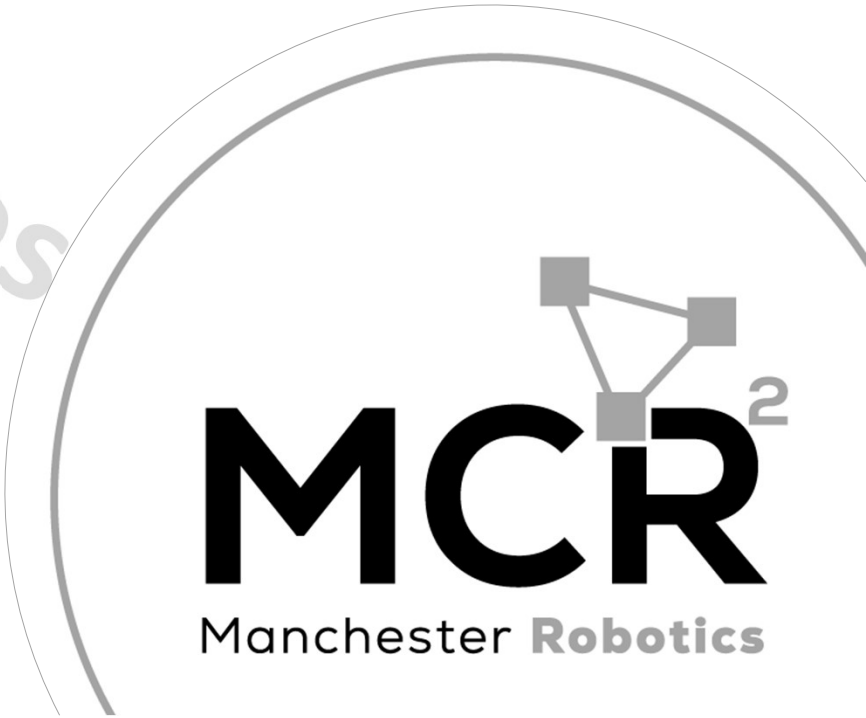
{Learn, Create, Innovate};



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.*