

{Learn, Create, Innovate};

Robot Operating System – ROS

Practicalities

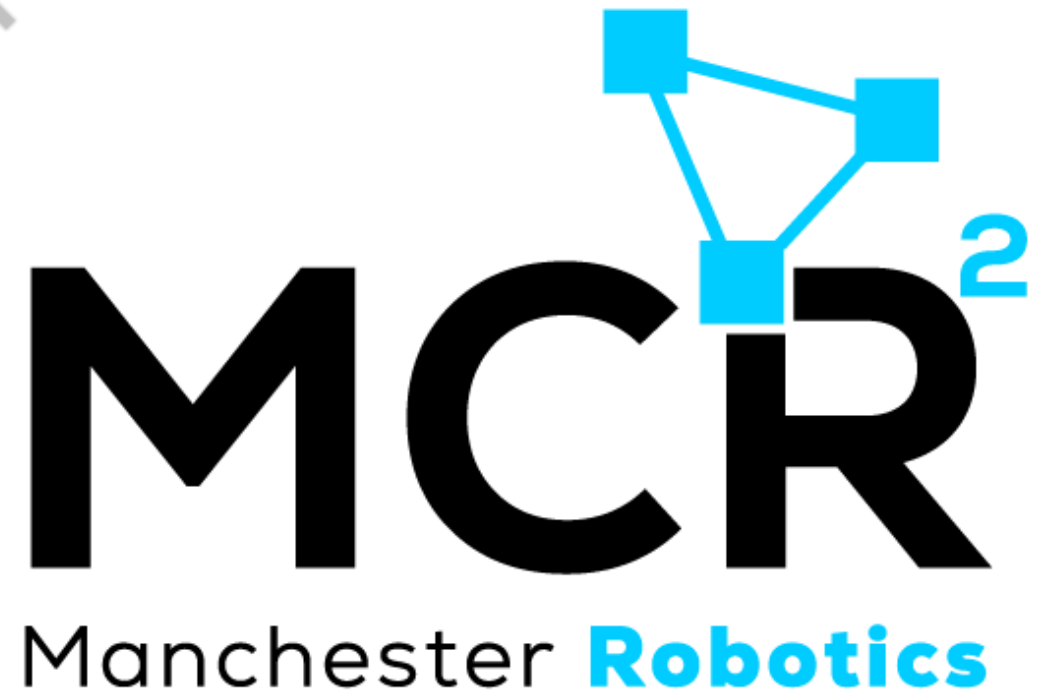
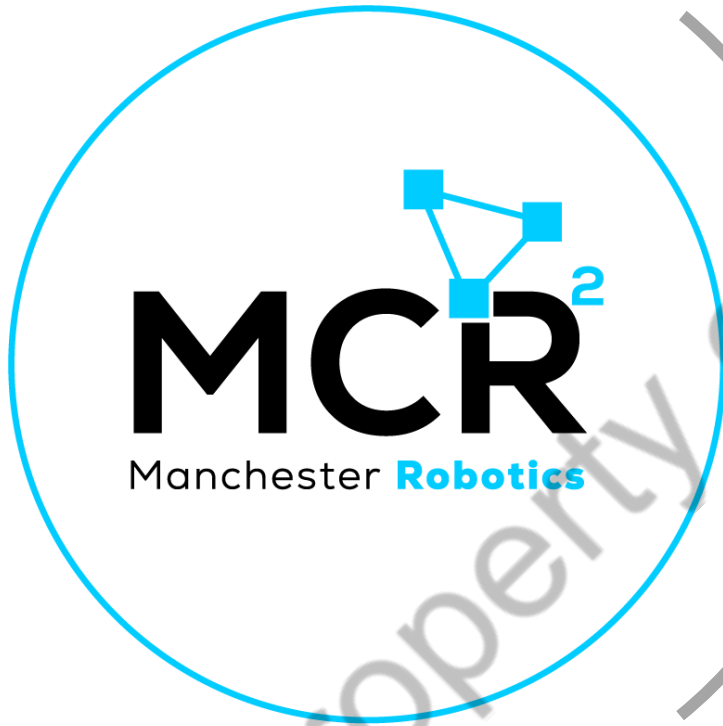


Table of contents



- 1 ROS Namespaces
- 2 ROS Namespaces Example
- 3 ROS Parameter Files
- 4 ROS Parameter Files Examples
- 5 ROS Custom Messages
- 6 ROS Custom Messages Examples
- 7 ROS Activity
- 8 Questions

Robot Operating System – ROS

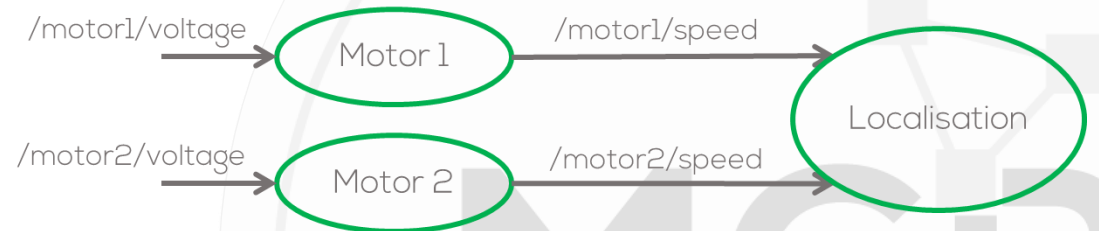
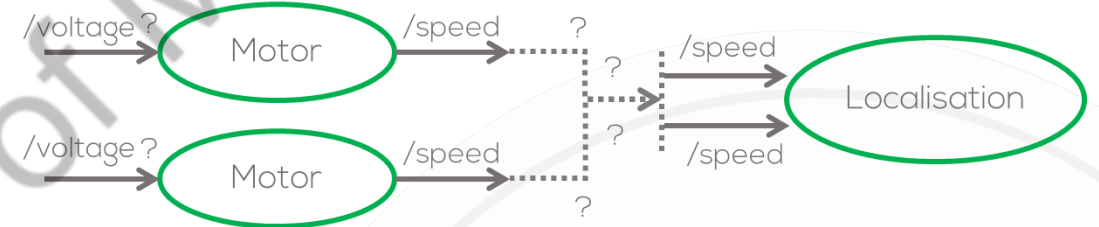
Namespaces

{Learn, Create, Innovate};



Manchester **Robotics**

- 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 file roslaunch will be used to define the namespaces.

Namespaces in ROS Example

For this example two talker nodes and two listener nodes will be generated using namespaces.

1. Open the launch file of the previous Talker and Listener Example (activity1.launch).
2. Modify it as follows

```
<?xml version="1.0" ?>
<launch>
  <group ns = "Group1">
    <node name="talker" pkg="basic_comms" type="talker.py" output="screen" launch-
      prefix="gnome-terminal --command" />

    <node name="listener" pkg="basic_comms" type="listener.py" output="screen" launch-
      prefix="gnome-terminal --command" />
    </group>

    <group ns = "Group2">
    <node name="talker" pkg="basic_comms" type="talker.py" output="screen" launch-
      prefix="gnome-terminal --command" />

    <node name="listener" pkg="basic_comms" type="listener.py" output="screen" launch-
      prefix="gnome-terminal --command" />
    </group>

  </launch>
```

ROS Launch

```
<?xml version="1.0" ?>
<launch>
  <group ns = "Group1">
    <node name="talker" pkg="basic_comms" type="talker.py"
      output="screen" launch-prefix="gnome-terminal --command" />

    <node name="listener" pkg="basic_comms" type="listener.py"
      output="screen" launch-prefix="gnome-terminal --command" />
  </group>
```

Group
Namespace

```
<group ns = "Group2">
  node name="talker" pkg="basic_comms" type="talker.py"
  output="screen" launch-prefix="gnome-terminal --command" />

  <node name="listener" pkg="basic_comms" type="listener.py"
    output="screen" launch-prefix="gnome-terminal --command" />
</group>
```

Group
Namespace

```
</launch>
```

1. Execute the Launch file

```
$ roslaunch basic_comms activity1.launch
```

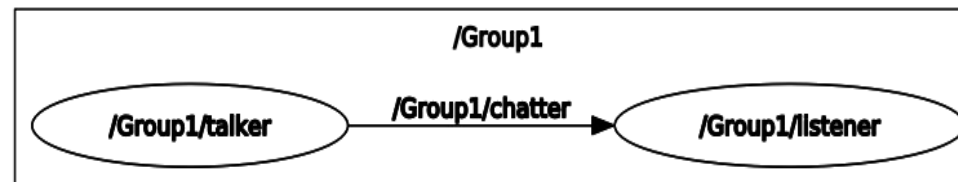
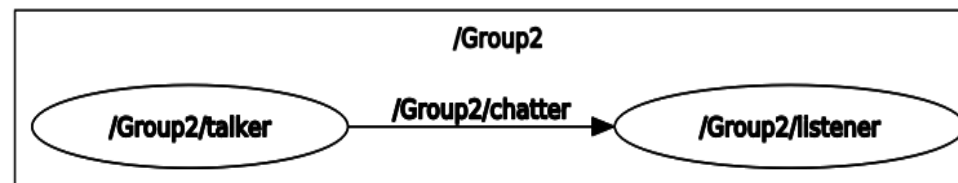
2. Execute the following command in a new terminal

```
$ rostopic list
```

3. In a new terminal, execute the rqt_graph to visualise the nodes

```
$ rosrun rqt_graph rqt_graph
```

```
student@ubuntu:~$ rostopic list
/Group1/chatter
/Group2/chatter
/rosout
/rosout_agg
student@ubuntu:~$
```



Robot Operating System – ROS

Parameter Files

{Learn, Create, Innovate};



Manchester **Robotics**

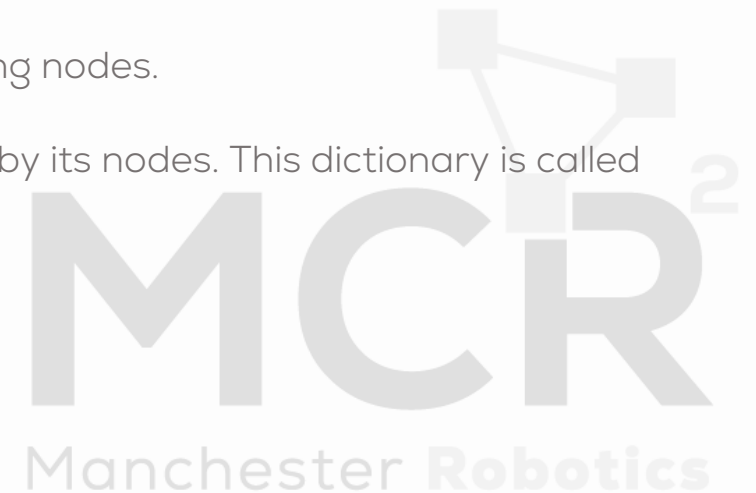


ROS Parameters



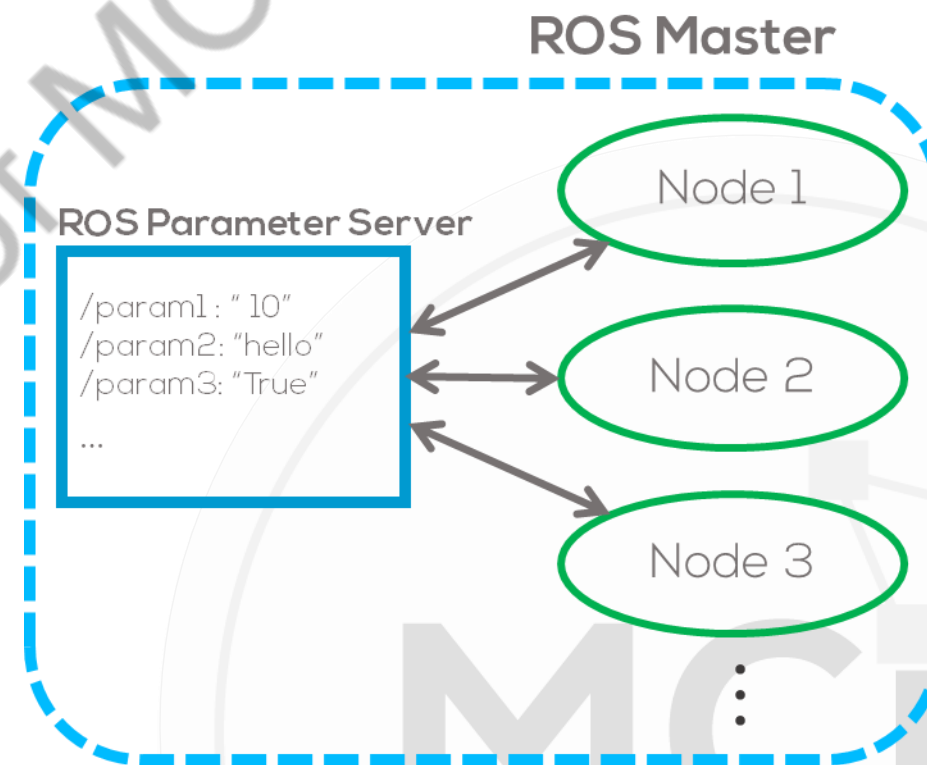
- Any software application, specially in robotics requires parameters.
- Parameters are variables with some predefined values that are stored in a separate file or hardcoded in a program such that the user has easy access to change their value.
- At the same time parameters can be shared amongst different programs to avoid rewriting them or recompiling the nodes (C++)
- In robotics, parameters are used to store values requiring tuning, 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.

Property of MCR2



ROS Parameter Server

- 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.
 - This is a globally viewed static library.
 - Parameters are composed of a name and a datatype. ROS can use the following types of parameters. More information [here](#).
- | | | |
|-------------------|-----------|------------------------------|
| • 32-bit integers | • strings | • base64-encoded binary data |
| • Booleans | • doubles | |
| • iso8601 dates | • lists | |



ROS Parameter Server

- In ROS parameters can be set in different ways.
- By the user in the launch files to be used by the nodes.
- They can also be taken from the command line and passed directly into the nodes.
- Set by nodes themselves.
- Parameters can be global (global namespace), within a local namespace (to a group of nodes only) or private i.e., specific to a node.
- It useful to point out that the parameter server depends on the execution of the ROS master not to the node execution time. Therefore, if you kill a node but not the master, the parameter will keep the value if it was modified.

ROS Parameter command line tools

- `rosparam`
 - `rosparam set [param]`: set a parameter
 - `rosparam get [param]`: get a parameter
 - `rosparam delete [param]`: delete parameter
 - `rosparam list`: list parameter names

** This is only a list with the most basic and used command line parameter tools. Other parameter command line tools can be seen [here](#).



Parameters example



ROS Parameter Example (Setting the parameters)

For this example, a global, a local, and a private parameters will be declared in the launch file.

1. Open the previous example Launch file "activity1.launch", and overwrite it as follows

```
<?xml version="1.0" ?>
<launch>

<param name = "Message" value = "Manchester Robotics Global!" />

  <group ns = "Group1">
    <param name = "Message" value = "Manchester Robotics Local!" />

    <node name="talker" pkg="basic_comms" type="talker.py" output="screen" launch-prefix="gnome-terminal --command" >
      <param name = "Message" value = "Manchester Robotics Private!" />
    </node>
    <node name="listener" pkg="basic_comms" type="listener.py" output="screen" launch-prefix="gnome-terminal --command" />
  </group>

  <group ns = "Group2">
    <param name = "Message" value = "Manchester Robotics Local!" />
    <node name="talker" pkg="basic_comms" type="talker.py" output="screen" launch-prefix="gnome-terminal --command" >
      <param name = "Message" value = "Manchester Robotics Private!" />
    </node>
    <node name="listener" pkg="basic_comms" type="listener.py" output="screen" launch-prefix="gnome-terminal --command" />
  </group>

</launch>
```



Parameters example



- Save the launch file and execute it as defined previously.
- You will see the parameters being displayed when ROS initializes

```
student@ubuntu:~$ roslaunch basic_comms activity1.launch
... logging to /home/student/.ros/log/769930ac-72f6-11ed-a959-375ca6caa471/roslaunch-ubuntu-193805.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://ubuntu:46737/

SUMMARY
=====
PARAMETERS
 * /Group1/Message: Manchester Robotl...
 * /Group1/talker/Message: Manchester Robotl...
 * /Group2/Message: Manchester Robotl...
 * /Group2/talker/Message: Manchester Robotl...
 * /Message: Manchester Robotl...
 * /roscdistro: noetic
 * /rosversion: 1.15.14

NODES
 /Group1/
 listener (basic_comms/listener.py)
 talker (basic_comms/talker.py)
 /Group2/
 listener (basic_comms/listener.py)
 talker (basic_comms/talker.py)
```

Parameters {

Created Local Parameter

Created Private Parameter

Created Global Parameter

- Open a terminal and type the following command to view the parameters in ROS

```
$ rosparam list
```

```
student@ubuntu:~$ rosparam list
/Group1/Message
/Group1/talker/Message
/Group2/Message
/Group2/talker/Message
/Message
/roscdistro
/roslaunch/uris/host_ubuntu__46737
/rosversion
/run_id
```

Launch file parameters

- The parameters declared outside the group's namespace are classified as global parameters. They are in the "root" of the directory therefore they only start with the forward slash "/". For this case, the parameter is depicted as `"/Message"`
- If the parameter is in the namespace of the group, the address decreases one level in this case `"/Group1/Message"`.
- Finally, if the parameter is located inside the node definition it goes down another level `"/Group1/Talker/Message"`

```
<param name = "Message" value = "Manchester Robotics Global!" /> ..... GlobalParameter
```

```
<group ns = "Group1">
```

```
  <param name = "Message" value = "Manchester Robotics Local!" /> ..... Local Namespace Parameter
```

```
  <node name="talker" pkg="basic_comms" type="talker.py" output="screen"
    launch-prefix="gnome-terminal --command" >
```

```
    <param name = "Message" value = "Manchester Robotics Private!" /> ..... Private Parameter
```

```
  </node>
```

```
  <node name="listener" pkg="basic_comms" type="listener.py"
    output="screen" launch-prefix="gnome-terminal --command" />
```

```
</group>
```

Group Namespace

** This is just part of the launch file



Parameters example



ROS Parameter Example (Getting the parameters)

This example will show how to use the parameters inside the nodes.

The command to retrieve parameters in python is the following

```
rospy.get_param(<parameter_name>, <default_value>)
```

- Open the file "talker.py"
- **Modify** the line

```
hello_str = "hello world %s" % rospy.get_time()
```

- For the following

```
hello_str = rospy.get_param("/Message", "No Parameter Found") + " " +  
    + str(rospy.get_time())
```

- Save the file and execute it using the roslaunch file.

- The following results must be shown.

```
[INFO] [1670064275.717228]: Manchester Robotics Global! 1670064275.717126 [INFO] [1670064275.718528]: I heard Manchester Robotics Global! 1670064275.717126  
[INFO] [1670064275.817284]: Manchester Robotics Global! 1670064275.8171816 [INFO] [1670064275.818409]: I heard Manchester Robotics Global! 1670064275.8171816  
[INFO] [1670064275.917292]: Manchester Robotics Global! 1670064275.9171872 [INFO] [1670064275.918779]: I heard Manchester Robotics Global! 1670064275.9171872  
[INFO] [1670064276.017400]: Manchester Robotics Global! 1670064276.0172698 [INFO] [1670064276.018655]: I heard Manchester Robotics Global! 1670064276.0172698  
[INFO] [1670064276.117449]: Manchester Robotics Global! 1670064276.1173494 [INFO] [1670064276.118640]: I heard Manchester Robotics Global! 1670064276.1173494  
[INFO] [1670064276.217287]: Manchester Robotics Global! 1670064276.2171874 [INFO] [1670064276.218414]: I heard Manchester Robotics Global! 1670064276.2171874  
[INFO] [1670064276.317145]: Manchester Robotics Global! 1670064276.3170837 [INFO] [1670064276.318421]: I heard Manchester Robotics Global! 1670064276.3170837  
[INFO] [1670064276.417691]: Manchester Robotics Global! 1670064276.4175897 [INFO] [1670064276.418944]: I heard Manchester Robotics Global! 1670064276.4175897  
[INFO] [1670064276.517579]: Manchester Robotics Global! 1670064276.5174305 [INFO] [1670064276.518891]: I heard Manchester Robotics Global! 1670064276.5174305  
[INFO] [1670064276.617259]: Manchester Robotics Global! 1670064276.6170766 [INFO] [1670064276.618487]: I heard Manchester Robotics Global! 1670064276.6170766  
[INFO] [1670064276.717471]: Manchester Robotics Global! 1670064276.7173698 [INFO] [1670064276.718555]: I heard Manchester Robotics Global! 1670064276.7173698  
[INFO] [1670064276.817382]: Manchester Robotics Global! 1670064276.8172798 [INFO] [1670064276.818524]: I heard Manchester Robotics Global! 1670064276.8172798  
[INFO] [1670064276.917815]: Manchester Robotics Global! 1670064276.9175203 [INFO] [1670064276.919064]: I heard Manchester Robotics Global! 1670064276.9175203  
[INFO] [1670064277.018272]: Manchester Robotics Global! 1670064277.0181584 [INFO] [1670064277.019555]: I heard Manchester Robotics Global! 1670064277.0181584  
[INFO] [1670064277.117606]: Manchester Robotics Global! 1670064277.117495 [INFO] [1670064277.118975]: I heard Manchester Robotics Global! 1670064277.117495  
[INFO] [1670064277.217251]: Manchester Robotics Global! 1670064277.2171304 [INFO] [1670064277.218598]: I heard Manchester Robotics Global! 1670064277.2171304  
[INFO] [1670064277.317409]: Manchester Robotics Global! 1670064277.3172138 [INFO] [1670064277.318597]: I heard Manchester Robotics Global! 1670064277.3172138  
[INFO] [1670064277.417441]: Manchester Robotics Global! 1670064277.4173372 [INFO] [1670064277.418653]: I heard Manchester Robotics Global! 1670064277.4173372  
[INFO] [1670064277.517434]: Manchester Robotics Global! 1670064277.517327 [INFO] [1670064277.518763]: I heard Manchester Robotics Global! 1670064277.517327  
[INFO] [1670064277.617280]: Manchester Robotics Global! 1670064277.6171782 [INFO] [1670064277.618609]: I heard Manchester Robotics Global! 1670064277.6171782  
[INFO] [1670064277.717501]: Manchester Robotics Global! 1670064277.717397 [INFO] [1670064277.718939]: I heard Manchester Robotics Global! 1670064277.717397  
[INFO] [1670064277.818274]: Manchester Robotics Global! 1670064277.8181229 [INFO] [1670064277.819718]: I heard Manchester Robotics Global! 1670064277.8181229  
[INFO] [1670064277.917745]: Manchester Robotics Global! 1670064277.9175513 [INFO] [1670064277.919090]: I heard Manchester Robotics Global! 1670064277.9175513
```

- For accessing the local namespace parameters and the private parameters modify the previous code accordingly.
Run the code as before, and you should see the other parameters being printed. More information [here](#).

Local Parameter

```
hello_str = rospy.get_param("Message", "No Parameter Found") + " " +  
    + str(rospy.get_time())
```

Private Parameter

```
hello_str = rospy.get_param("~Message", "No Parameter Found") + " " +  
    + str(rospy.get_time())
```

****As in ubuntu use the tilde ~ for private addresses.**



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.
 - 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 following example will be used to show how to create config files and interact with them.
- The parameters set up in the config files (YAML Files) can be declared as before global, local to a parent namespace or private.
 - The way to define the hierarchy of a parameter, like in python depends on spacing.

```
Message: "Parameter YAML File Global"
```

```
Group1:
```

```
  Message: "Parameter YAML File local G1"
```

```
  talker:
```

```
    Message: "Parameter YAML File private G1"
```

- For this case, "Message" is a global parameter which will be named `"/Message"` since its located in the root directory.
- The same applies for the following "Message" Parameters, since they will be located a namespace level and a private level for the node "talker".



Parameter files

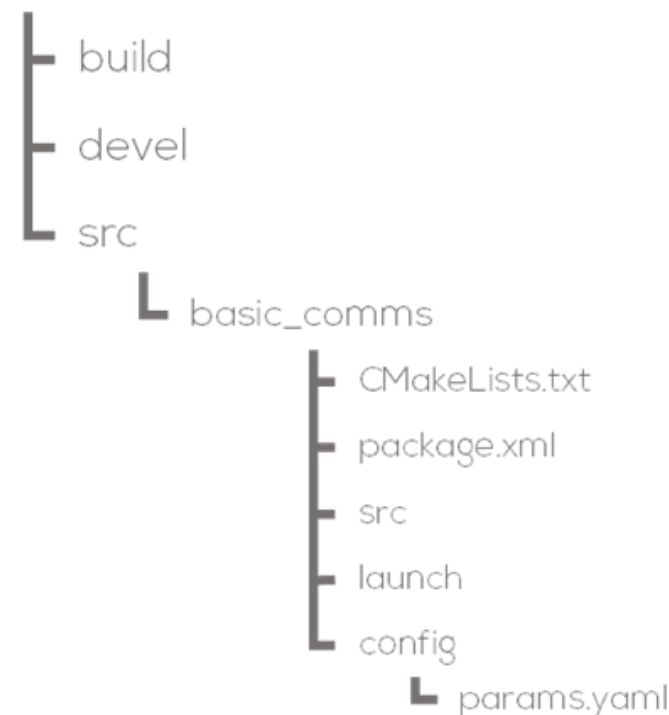


Parameter files example

For this example, we will use the previous example.

1. Inside the "basic_comms" package, create a folder named "config".
2. Inside the config folder create a file named "params.yaml" (the file can have any name just make sure follow a convention properly).

catkin_workspace





Parameter files



- Open the newly created parameter file (params.yaml) and edit it as follows

```
Message: "Parameter YAML File Global"
```

```
Group1:
```

```
  Message: "Parameter YAML File local G1"
```

```
  talker:
```

```
    Message: "Parameter YAML File private G1"
```

```
Group2:
```

```
  Message: "Parameter YAML File local G2"
```

```
  talker:
```

```
    Message: "Parameter YAML File private G2"
```

- Save it and close the file.
- Open the roslaunch file (activity1.launch)
- Modify the launch file as follows, save it and close it.

```
<?xml version="1.0" ?>
<launch>

  <rosparam file = "${find basic_comms)/config/params.yaml" command = "load" />

  <group ns = "Group1">
    <node name="talker" pkg="basic_comms" type="talker.py" output="screen" launch-
      prefix="gnome-terminal --command" />
    <node name="listener" pkg="basic_comms" type="listener.py" output="screen"
      launch-prefix="gnome-terminal --command" />
  </group>

  <group ns = "Group2">
    <node name="talker" pkg="basic_comms" type="talker.py" output="screen" launch-
      prefix="gnome-terminal --command" />
    <node name="listener" pkg="basic_comms" type="listener.py" output="screen"
      launch-prefix="gnome-terminal --command" />
  </group>

</launch>
```

- The first line is where the address of the params.yaml file is located. The label "\${find basic_comms)" allows to the launch file to determine that the path to the config file is inside the package "basic_comms".



Parameter files



- Run the modified launch file.
- Open a new terminal and type **rosparam list**

```
student@ubuntu:~/catkin_ws$ rosparam list
/Group1/Message
/Group1/talker/Message
/Group2/Message
/Group2/talker/Message
/Message
/rosdistro
/roslaunch/uris/host_ubuntu__34625
/rosversion
/run_id
```

- To access each parameter as before, the **get_parameter** function must be modified as follows.

Global Parameter

```
hello_str = rospy.get_param("/Message", "No Parameter Found") + " "
+ str(rospy.get_time())
```

Local Namespace Parameter

```
hello_str = rospy.get_param("Message", "No Parameter Found") + " "
+ str(rospy.get_time())
```

Private Parameter

```
hello_str = rospy.get_param("~Message", "No Parameter Found") + " "
+ str(rospy.get_time())
```

- The Following image, shows the result when using the global parameter.

The image displays four terminal windows showing the output of `rosparam list` and `rospy.get_param` for global, local, and private parameters. The first two terminals show the output of `rosparam list`, listing parameters such as `/Group1/Message`, `/Group1/talker/Message`, `/Group2/Message`, `/Group2/talker/Message`, `/Message`, `/rosdistro`, `/roslaunch/uris/host_ubuntu__34625`, `/rosversion`, and `/run_id`. The next two terminals show the output of `rospy.get_param` for global, local, and private parameters, demonstrating that the parameters are found and returned as expected.

Robot Operating System – ROS

Custom Messages

{Learn, Create, Innovate};

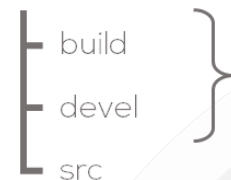


Manchester **Robotics**

Configuration Files

- As seen before, 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.
- ROS has the package tool called Catkin to help build multiple packages at once.
 - Catkin, lets us group the packages into a workspace, and allows us to build all the packages inside the workspace.
- To do this, Catkin require 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_workspace



“ Folders autogenerated by catkin_make when building/compiling the workspace

Package

Package 1



Configuration Files



ROS Packages



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.



ROS Packages



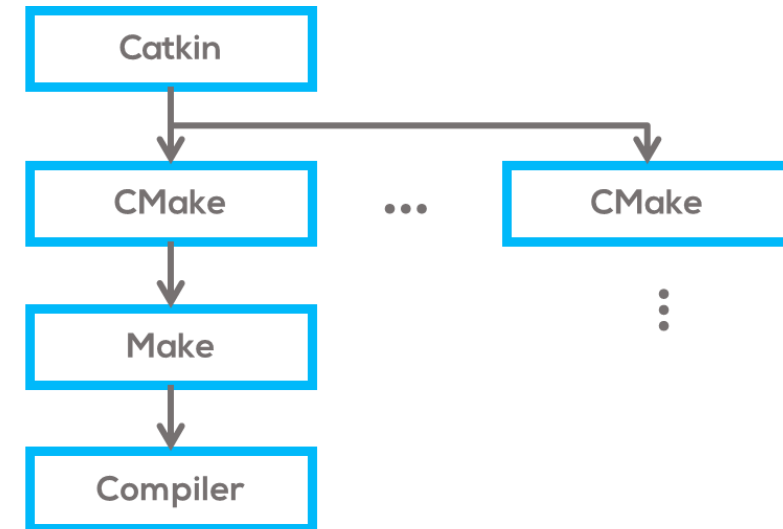
Catkin and CMake Files

- 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 Catkin tool was created. Catkin is the official build system of ROS
- Catkin specifically tries to improve development on large sets of related packages in a consistent and conventional way.
- More information about catkin can be found [here](#).
- More information about these files can be found [here](#) and [here](#).

Deals at a higher level with the dependencies of your project.

Deals with the libraires, flags required to build a code.

Deals with the specific details on how to build a code and compiles the code.





ROS Packages



```
# 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

CMakeLists Structure

- 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())



ROS Packages



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_minimum_required(VERSION 3.0.2)
project(courseworks)
```

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

```
## 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
)
```

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

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

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

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

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

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

```
## 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}
)
```

} Installation of the scripts



ROS Packages



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 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



ROS Packages



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.



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 Message Example

This example will help the student to understand the concepts of custom messages, how to implement them and how to add them to the package.

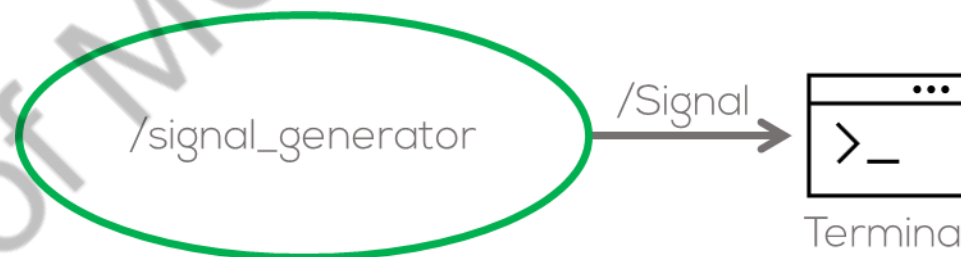
- For this example, a simpler version of the previous activity (Signal Generator) code will be used.
 - It is encouraged for the students to use their coursework code and modify it to use a custom message.
- Since this is not a challenge, we will use the previous challenge 1 package for this example. **The student can follow the same steps in any package.**



ROS Custom messages



- For this example, a new message will be generated to contain two different values of data and two values of metadata (just as an example).
- In the previous exercise, the user was required to publish two messages, each on a different topic.
- For this activity, only one topic will be necessary, and the message will include all the information required in the previous exercise.



```
Custom msg signal_msg
float32 amplitude
float32 frequency
float64 time_x
float32 signal_y
```

ROS Custom Message Creation

To start let's create the custom message

1. In the previous challenge package (or the student's own package), create a folder named “msg”
2. Inside the folder “msg” create a file called “signal_msg.msg”
3. Open the file using a text editor and write the following.

```
float32 amplitude  
float32 frequency  
float64 time_x  
float32 signal_y
```

4. Save the file. The custom message has been created!

ROS Custom Message Configuration

The following step is to tell ROS that a custom message will be used, where the message is located and its dependencies.

To do this, the CmakeList.txt and the Package.xml will be modified (Follow each step carefully).

1. Open the CMakeLists.txt, of the package (or the students' package) and find the following line and modify it as follows

```
find_package(catkin REQUIRED COMPONENTS rospy std_msgs  
message_generation)
```

- This line tells ROS to add the message_generation component (to create new messages) when compiling the program.

- Find the following lines, uncomment it and modify them as follows.

```
add_message_files(  
  FILES  
    signal_msg.msg  
)  
  
generate_messages(  
  DEPENDENCIES  
    std_msgs  
)
```

- These lines tell ROS the name of the message files contained in the msg folder, and the dependencies of such messages (for more complex message you must add the dependencies here).

- Modify the *catkin_package* line as follows. This line tells the dependencies used by catkin.

```
catkin_package(  
  # INCLUDE_DIRS include  
  # LIBRARIES basic_comms  
    CATKIN_DEPENDS message_runtime  
  # DEPENDS system_lib  
)
```

- Save and Close the *CmakeLists.txt*.
- Open the file *package.xml*
- Uncomment or add the following lines

```
<build_depend>message_generation</build_depend>  
<exec_depend>message_runtime</exec_depend>
```

- Save and close.
- Done! The configuration for using a custom message has been finished.



ROS Custom messages



ROS Custom Message Usage

The following code, as stated before, is just a simple example on how to use the newly created message. This code is based on the previous challenge.

- Firstly, the new message must be imported from the library
Change the <PACKAGE> for the name of the current package

```
from <PACKAGE>.msg import signal_msg
```

- Define the new publisher and its attributes, to publish the new message as follows.

```
signal_pub=rospy.Publisher("signal",signal_msg, queue_size=10)
```

- Finally, fill out the message with the data to be sent and publish it as follows

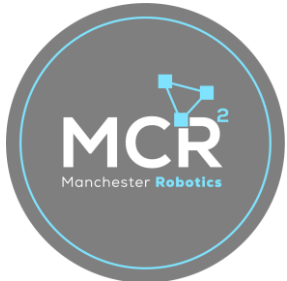
```
msg = signal_msg()
msg.amplitude = Amplitude
msg.frequency = Omega
msg.time_x = start_time
msg.signal_y = signal
signal_pub.publish(msg)
```

```
import rospy
import numpy as np
from courseworks.msg import signal_msg
msg = signal_msg()
first = True
start_time = 0.0
signal = 0.0
time = 0.0
if __name__ == '__main__':
    #Initialise Node
    rospy.init_node("signal_generator")
    #Setup the node
    rate = rospy.Rate(10)
    #Setup Publishers and subscribers
    signal_pub=rospy.Publisher("signal",signal_msg, queue_size=10)

    init_time = rospy.get_time()
    while not rospy.is_shutdown():
        #Setup Function
        if first == True:
            start_time = rospy.get_time()
            first = False
        else:
            time = rospy.get_time()-init_time
            signal = np.sin(Omega*time)
            msg.amplitude = Amplitude
            msg.frequency = Omega
            msg.time_x = time
            msg.signal_y = signal

            signal_pub.publish(msg)

        rate.sleep()
```



ROS Custom messages



ROS Custom Message Usage

- Save the previous file and close it. (do not forget to make it executable `sudo chmod +x file.py`)
- Recompile the workspace using "`catkin_make`".
- Open a new terminal and start a `roscore`.
- Launch the node
- Open another terminal and print the topics using `rostopic list`
- Echo the topic (`rostopic echo /signal`) to verify if the information is being sent.
- The following results are expected.

```
time_x: 42.62652254104614
signal_y: -1.4285545349121094
---
amplitude: 2.0
frequency: 6.283199787139893
time_x: 42.67646789550781
signal_y: -1.790870189666748
---
amplitude: 2.0
frequency: 6.283199787139893
time_x: 42.7266628742218
signal_y: -1.978720784187317
---
amplitude: 2.0
frequency: 6.283199787139893
time_x: 42.776490926742554
signal_y: -1.9721505641937256
---
amplitude: 2.0
frequency: 6.283199787139893
time_x: 42.82658338546753
signal_y: -1.7723089456558228
```

Activity

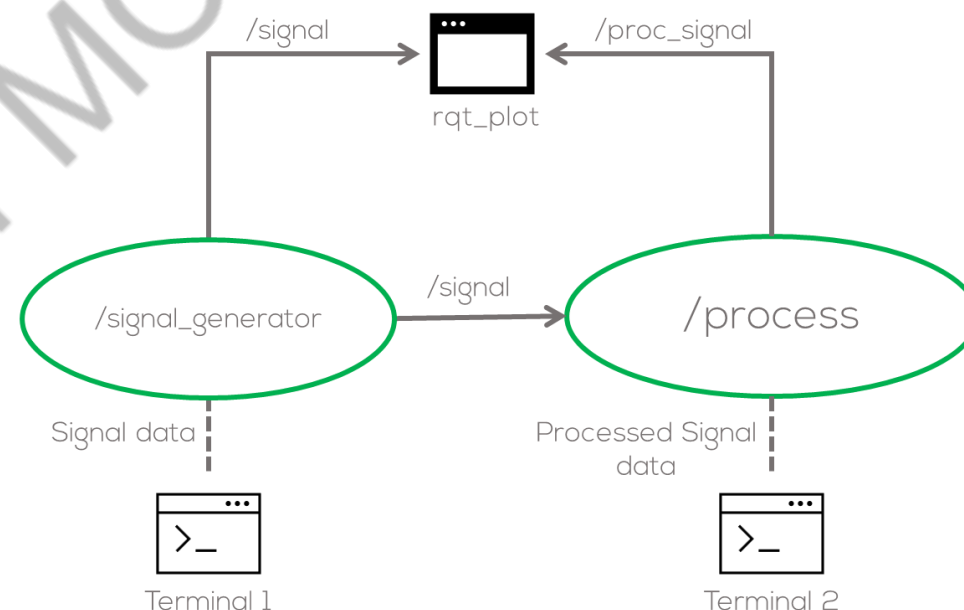
- Parametrise the “process” and “signal_generator” Nodes (created in the previous challenge), subscribe to the new topic and use the custom message.
- Make a roslaunch file for both nodes.

Hints: import the signal message and don't forget to declare the variable (to be global) appropriately inside the call-back function

```
from basic_comms.msg import signal_msg

signal_data = 0
time_data = 0
```

```
def callback(msg):
    global signal_data, time_data
    signal_data = msg.signal_y
    time_data = msg.time_x
```



```
Custom msg signal_msg
float32 amplitude
float32 frequency
float64 time_x
float32 signal_y
```




Using custom messages from other ROS packages



Commercial ROS packages: custom messages.

Commercial ROS packages typically include proprietary messages that are specific to their software or hardware.

As users of these packages, we must utilize their messages to communicate with their nodes.

To avoid re-declaring new messages, we must follow certain steps to use the messages defined in commercial packages in our own packages.

CMakeLists file:

1. Localise the CmakeLists file of your package.
2. Add the “external” package name where the custom messages to be used are located (“package_name”) and the “message_generation” packages to the “find_package”

```
find_package(catkin REQUIRED COMPONENTS message_generation
package_name)
```

3. Add the “message_runtime” and the “external” package name, as a dependencies for your package to the CATKIN_DEPENDS

```
catkin_package(
  CATKIN_DEPENDS message_runtime package_name
)
```



Using custom messages from other ROS packages



Package XML file:

1. Localise the package.xml file of your package.
2. Add the “external” package name as an execution dependency and build dependency.

```
<exec_depend>package_name</exec_depend>  
<build_depend>package_name </build_depend>
```

3. Add the “message_runtime” and the “message_generation” packages as execution and build dependencies respectively to the file.

```
<build_depend>message_generation</build_depend>  
<exec_depend>message_runtime</exec_depend>
```

Script

1. Now you can import and use the message as a regular custom message as follows

```
from package_name.msg import custom_message
```

Where “package_name” is the name of the external package and “custom_message” is the name of the message of the “external” package.



ROS Launch XML.

Roslaunch uses XML files that describe the nodes that should be run, parameters that should be set, and other attributes of launching a collection of ROS nodes.

- Roslaunch evaluates the XML file in a single pass.
- Includes are processed in depth-first traversal order.
- **Tags** are evaluated serially and the last setting wins. Thus, if there are multiple settings of a parameter, the last value specified for the parameter will be used.
- Roslaunch tag attributes can make use of **substitution args**, which roslaunch will resolve prior to launching nodes.

```
<launch>
<!-- start a group of nodes in the 'wg2' namespace -->
<group ns="wg2">

  <!-- remap applies to all future statements in this scope. -->
  <remap from="chatter" to="hello"/>

  <node pkg="rospy_tutorials" type="listener" name="listener"
    args="--test" respawn="true" />

  <node pkg="rospy_tutorials" type="talker" name="talker">
    <!-- set a private parameter for the node -->
    <param name="talker_1_param" value="a value" />
    <!-- nodes can have their own remap args -->
    <remap from="chatter" to="hello-1"/>
    <!-- you can set environment variables for a node -->
    <env name="ENV_EXAMPLE" value="some value" />
  </node>

</group>
</launch>
```



Roslaunch: Substitution args



Substitution args `$(env)` and `$(optenv)`

- `$(env ENVIRONMENT_VARIABLE)`: Substitute the value of a variable from the current environment. The launch will fail if the environment variable is not set. In other words, get the values of the environment variables from the environment roslaunch was invoked from.
- `$(optenv ENVIRONMENT_VARIABLE default_value)`: Substitute the value of an environment variable if it is set. If `default_value` is provided, it will be used if the environment variable is not set. If `default_value` is not provided, an empty string will be used.

Example:

Launch file `$(env)`

```
<param name="foo" value="$(env NUM_CPUS)" />  
<param name="foo" value="$(optenv NUM_CPUS 1)" />
```



Roslaunch: Substitution args



Substitution args \$(find) and \$(dirname)

- **\$(find pkg)**: Specifies a package-relative path. The filesystem path to the package directory will be substituted inline.

Example

```
$(find rospy)/manifest.xml.
```

- **\$(dirname)**: Returns the absolute path to the directory of the launch file in which it appears.

Example

```
<include file="$(dirname)/other.launch" />
```

Substitution args \$(arg) and \$(eval)

- **\$(arg foo)** evaluates to the value specified by an <arg> tag. There must be a corresponding <arg> tag in the same launch file that declares the arg.

Example

Launch File

```
<param name="foo" value= " $(arg my_foo)" />
```

Terminal

```
roslaunch pkg_name launch_file.launch foo:=1
```

- **\$(eval <expression>)**: allows to evaluate arbitrary complex python expressions..

Example

```
<param name="foo" value= " $(eval 2.* 3.1415 * arg('radius'))" />
```



Roslaunch: Substitution args



If and unless attributes

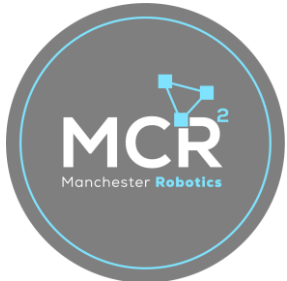
All tags support if and unless attributes, which include or exclude a tag based on the evaluation of a value. "1" and "true" are considered true values. "0" and "false" are considered false values.

More information about substitution arguments can be found [here](#).

Example

```
<group if="$(arg foo)">
  <!-- stuff that will only be evaluated if foo is true -->
</group>

<param name="foo" value="bar" unless="$(arg foo)" /> <!-- This param
won't be set when "unless" condition is met -->
```



Roslaunch: Tags



Tags

Allow the user to define variables, structure the launch files and specify the various components that need to be launched and configured to run a ROS system effectively

- <launch>
- <node>
- <machine>
- <include>
- <remap>
- <env>
- <param>
- <rosparam>
- <group>
- <test>
- <arg>

Tags <launch>

The <launch> tag is the root element of any roslaunch file. Its sole purpose is to act as a container for the other elements.

Example

```
<launch>
  Things to be launched
</launch>
```



Roslaunch: Tags



Tags <node>

This tag is used to specify the details of a ROS node that needs to be launched, such as its package, type, name, namespace, and any arguments required. More information and attributes [here](#).

Example

```
node name="listener1" pkg="rospy_tutorials" type="listener.py" args="-  
-test" respawn="true" />
```

Tags <param>

It is used to define ROS parameters that can be set when launching nodes. Parameters are key-value pairs used to configure node behavior. More information and attributes [here](#).

Example

```
<param name="publish_frequency" type="double" value="10.0" />
```




Roslaunch: Tags



Tags <node>

This tag is used to specify the details of a ROS node that needs to be launched, such as its package, type, name, namespace, and any arguments required. More information and attributes [here](#).

Example

```
node name="listener1" pkg="rospy_tutorials" type="listener.py" args="-  
-test" respawn="true" />
```

Tags <param>

It is used to define ROS parameters that can be set when launching nodes. Parameters are key-value pairs used to configure node behavior. More information and attributes [here](#).

Example

```
<param name="publish_frequency" type="double" value="10.0" />
```



Roslaunch



Tags <env>

- The `<env>` tag allows you to set environment variables on nodes that are launched. This tag may only be used within the scope of a `<launch>`, `<include>`, `<node>` or `<machine>` tag. When it is used inside of a `<launch>` tag, the `<env>` tag only applies to nodes declared after.
- NOTE: Values set using the `<env>` tag will not be seen by `$(env ...)`, so the `<env>` tag cannot be used to parametrize launch files.

Example:

Launch file `<env>`

```
<env name="TESTVAR" value="$(optenv INTVAR  
/home/user/default_directory1:/home/user/default_directory2)" />
```

Script file

```
#!/usr/bin/env python  
import os  
print "\r\n"  
print "TESTVAR = ", os.environ['TESTVAR']  
print "\r\n"
```

Result (terminal)

```
TESTVAR = /home/user/default_directory1:/home/user/default_directory2
```

Substitute (terminal)

```
export INTVAR=/different/directory
```

Result (terminal)

```
TESTVAR = /different/directory
```



Roslaunch



Tags <group>

- The `<group>` tag makes it easier to apply settings to a group of nodes. It has an `ns` attribute that lets you push the group of nodes into a separate namespace.

Example:

```
<?xml version="1.0" ?>
<launch>
  <group ns = "Group1">
    <node name="talker" pkg="basic_comms" type="talker.py" output="screen"
      launch-prefix="gnome-terminal --command" />

    <node name="listener" pkg="basic_comms" type="listener.py"
      output="screen" launch-prefix="gnome-terminal --command" />
  </group>

  <group ns = "Group2">
    <node name="talker" pkg="basic_comms" type="talker.py" output="screen"
      launch-prefix="gnome-terminal --command" />

    <node name="listener" pkg="basic_comms" type="listener.py"
      output="screen" launch-prefix="gnome-terminal --command" />
  </group>
</launch>
```

Tags <remap>

The `<remap>` tag allows you to pass in name remapping arguments to the ROS node that you are launching in a more structured manner than setting the `args` attribute of a `<node>` directly.

Example:

```
<remap from="chatter" to="hello"/>
```



Roslaunch: Tags



Tags <include>

The <include> tag enables you to import another roslaunch XML file into the current file.

Example

```
<include file="$(find pkg-name)/path/filename.launch"/>
```

Tags <arg>

The <arg> tag specifies values that are passed via the command-line, passing in via an <include>, or declared for higher-level files.

- args are not global.
- An arg declaration is specific to a single. Args must be passed to an include file.

Example

Launch file:

```
<include file="included.launch">  
  <!-- all vars that included.launch requires must be set -->  
  <arg name="hoge" value="fuga" />  
</include>
```

Included Launch file:

```
<launch>  
  <!-- declare arg to be passed in -->  
  <arg name="hoge" />  
  
  <!-- read value of arg -->  
  <param name="param" value="$(arg hoge)"/>  
</launch>
```



Roslaunch: Tags



Tags `<arg>`

As stated previously, args can be used to configure nodes by setting the parameters, furthermore, they can be used to pass values from other launch files.

```
<arg name="foo" default="1" /> #Declares foo with a default value. foo  
can be overridden by command-line argument (if top-level) or via  
<include> passing.
```

```
<arg name="foo" value="bar" /> #Declares foo with constant value. The  
value for foo cannot be overridden
```

Example 2: System Launch file

```
<?xml version="1.0"?>  
<launch>  
  <!--System Parameters-->  
  <arg name="system_gain" default="13.2"/>  
  <arg name="system_tau" default="0.05"/>  
  
  <!--Simulation Parameters-->  
  <arg name="sample_time" default="0.02"/>  
  <arg name="node_rate" default="150"/>  
  
  <!--Initial conditions-->  
  <arg name="initial_conditions" default="0.0"/>  
  <!--System Limitations (Saturation)-->  
  <arg name="max_output" default="13.0"/>  
  <arg name="min_input" default="0.2"/>  
  
  <!--Initialise kinematic_model -->  
  <node pkg="first_order_sys_sim" type="first_order_sys.py" name="first_order_sys"  
    output="screen">  
    <param name="system_node_rate" value="$(arg node_rate)"/>  
    <param name="system_sample_time" value="$(arg sample_time)"/>  
    <param name="system_param_K" value="$(arg system_gain)"/>  
    <param name="system_param_T" value="$(arg system_tau)"/>  
    <param name="system_initial_conditions" value="$(arg initial_conditions)"/>  
    <param name="system_max_output" value="$(arg max_output)"/>  
    <param name="system_min_input" value="$(arg min_input)"/>  
  </node>  
</launch>
```



Roslaunch: Tags



Example 3: System Launch File (previous)

```
<?xml version="1.0"?>
<launch>
  <!--System Parameters-->
  <arg name="system_gain" default="13.2"/>
  <arg name="system_tau" default="0.05"/>

  <!--Simulation Parameters-->
  <arg name="sample_time" default="0.02"/>
  <arg name="node_rate" default="150"/>

  <!--Initial conditions-->
  <arg name="initial_conditions" default="0.0"/>
  <!--System Limitations (Saturation)-->
  <arg name="max_output" default="13.0"/>
  <arg name="min_input" default="0.2"/>

  <!--Initialise kinematic_model -->
  <node pkg="first_order_sys_sim" type="first_order_sys.py"
name="first_order_sys" output="screen">
    <param name="system_node_rate" value="$(arg node_rate)"/>
    <param name="system_sample_time" value="$(arg sample_time)"/>
    <param name="system_param_K" value="$(arg system_gain)"/>
    <param name="system_param_T" value="$(arg system_tau)"/>
    <param name="system_initial_conditions" value="$(arg initial_conditions)"/>
    <param name="system_max_output" value="$(arg max_output)"/>
    <param name="system_min_input" value="$(arg min_input)"/>
  </node>
</launch>
```

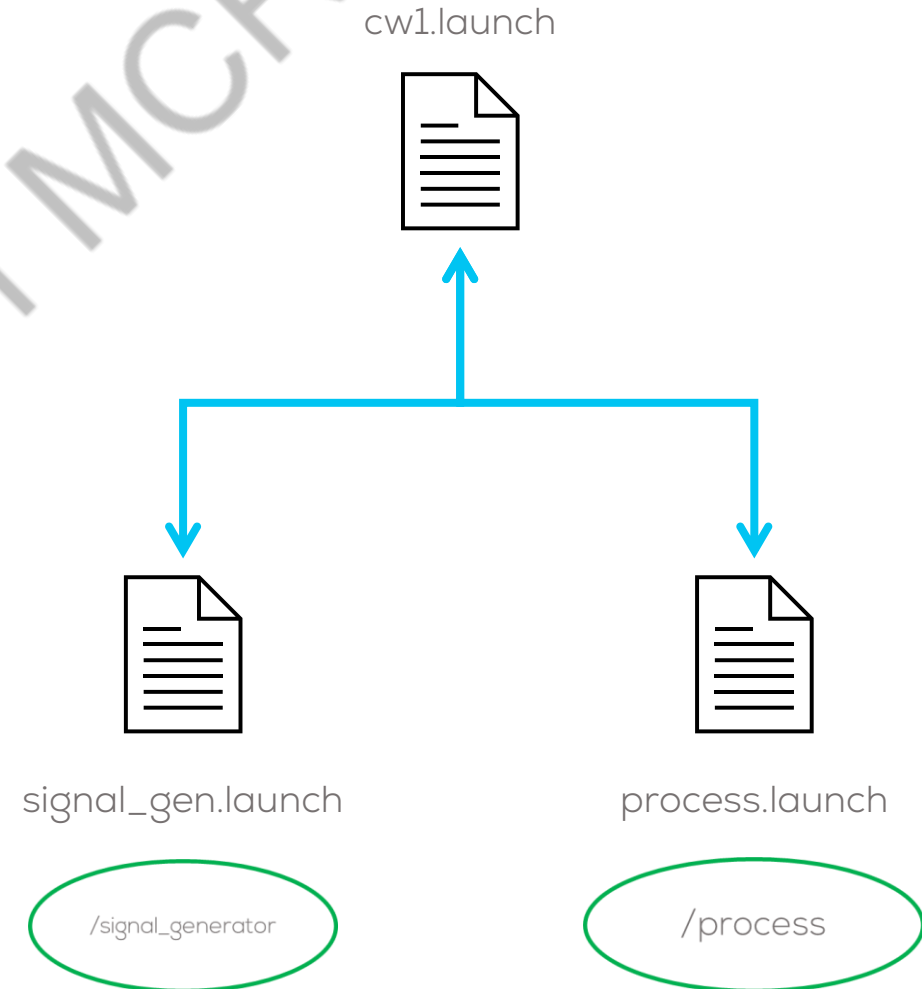
Example 3: Including (calling) the System Launch file into another launch file

```
<?xml version="1.0"?>
<launch>
  <!--System Parameters-->
  <arg name="system_gain" default="13.2"/>
  <arg name="system_tau" default="0.05"/>
  <arg name="initial_conditions" default="0.0"/>
  <arg name="max_output" default="13.0"/>
  <arg name="min_input" default="0.05"/>

  <!--Initialise system model -->
  <include file="$(find first_order_sys_sim)/launch/system.launch">
    <arg name="system_gain" value="$(arg system_gain)"/>
    <arg name="system_tau" value="$(arg system_tau)"/>
    <arg name="sample_time" value="$(arg sample_time)"/>
    <arg name="node_rate" value="$(arg node_rate)"/>
    <arg name="initial_conditions" value="$(arg initial_conditions)"/>
    <arg name="max_output" value="$(arg max_output)"/>
    <arg name="min_input" value="$(arg min_input)"/>
  </include>
</launch>
```

Activity

- Develop a roslaunch file structure for both nodes.
 - Use arg tags and substitution args, to set the parameters in a launch file for each node.
 - process.launch
 - signal_gen.launch
- Develop a roslaunch file that invokes both nodes and set their parameters using arg tags.
 - cw1.launch



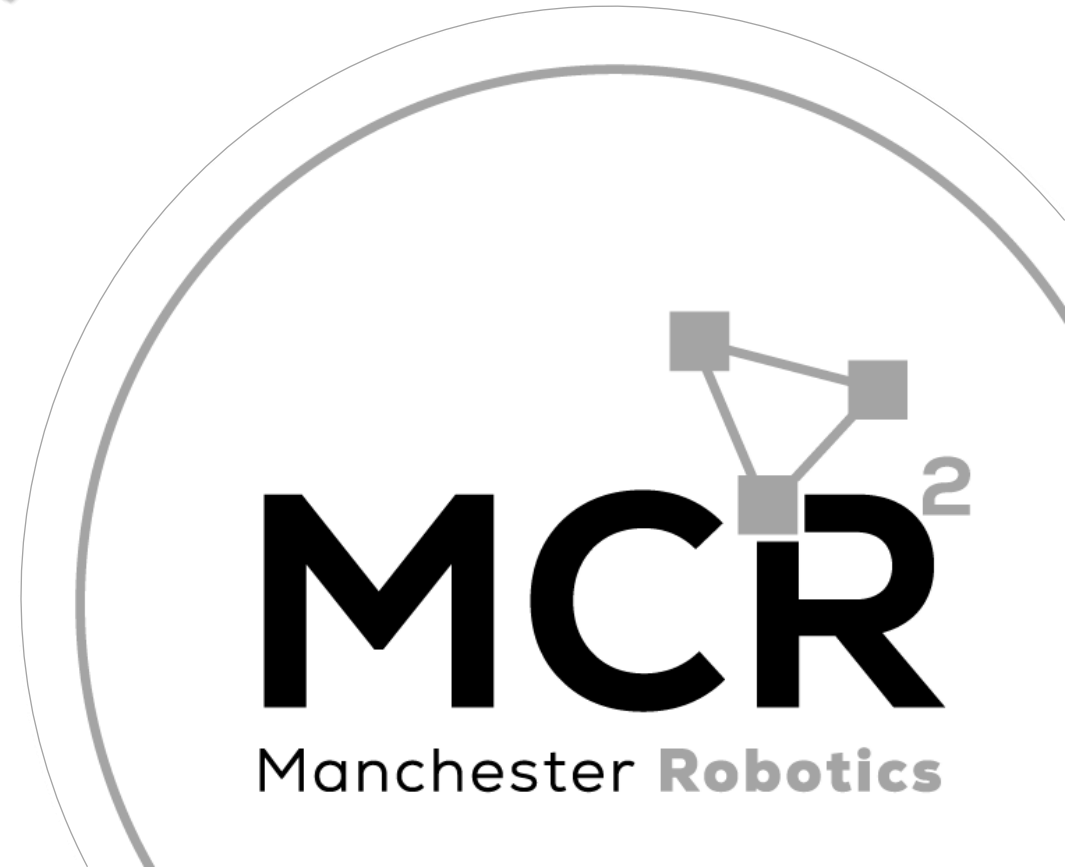


Q&A

Questions?

Property of MCR2

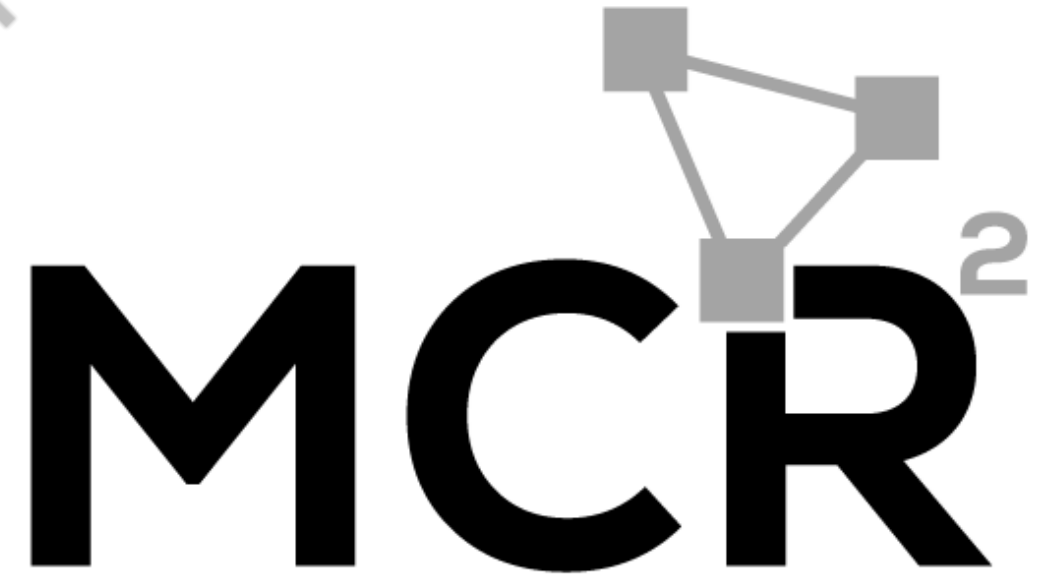
{Learn, Create, Innovate};



{Learn, Create, Innovate};

Thank you

Property of MCR²



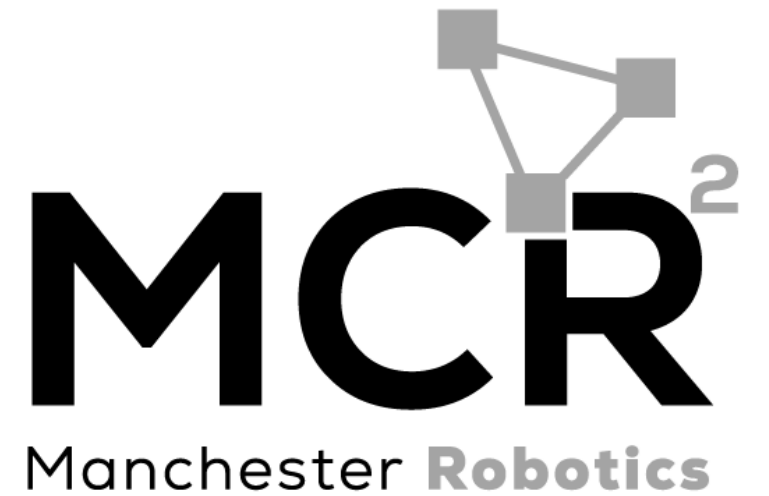
Manchester **Robotics**

T&C

Terms and conditions

{Learn, Create, Innovate};

Property of MCR2





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.*