

1. Multi-vehicle control

1. Program function description

Connect the handle to the virtual machine and set the virtual machine to connect to the handle. After the programs on the virtual machine and the car are started, you can control two cars at the same time through the handle, and the movements of the two cars are synchronized.

2. Program reference path

After entering the docker container, the source code of this function is located at

```
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_multi/launch
```

The file started by the virtual machine, the source code is located at,

```
/home/yahboom/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_multi/yahboomcar_multi
```

3. Program startup

Note: Multi-machine communication needs to be set up in advance, that is, the virtual machine and the car (taking two cars as an example) need to be in the same LAN and the car docker, and the **ROS_DOMAIN_ID** of the virtual machine must be the same, distributed communication can be carried out. Setting method reference: [06], Linux operating system->[04], multi-machine communication configuration

3.1. Start command

After entering the docker containers of Car 1 and Car 2 respectively, enter in the terminal according to the actual car model

```
#X3 model launched
#小车1
ros2 launch yahboomcar_multi x3_bringup_multi_ctrl.launch.xml robot_name:=robot1
#小车2
ros2 launch yahboomcar_multi x3_bringup_multi_ctrl.launch.xml robot_name:=robot2
#R2 model launched
#小车1
ros2 launch yahboomcar_multi R2_bringup_multi_ctrl.launch.xml robot_name:=robot1
#小车2
ros2 launch yahboomcar_multi R2_bringup_multi_ctrl.launch.xml robot_name:=robot2
#Virtual machine side startup
ros2 run joy joy_node
#The handle program currently only supports starting cars of the same model, X3
model
ros2 run yahboomcar_multi yahboomcar_X3_ctrl
#raspi5Master execution
#ros2 run yahboomcar_ctrl yahboom_joy_X3
#The handle program currently only supports starting cars of the same model, R2
model
ros2 run yahboomcar_multi yahboomcar_R2_ctrl
```

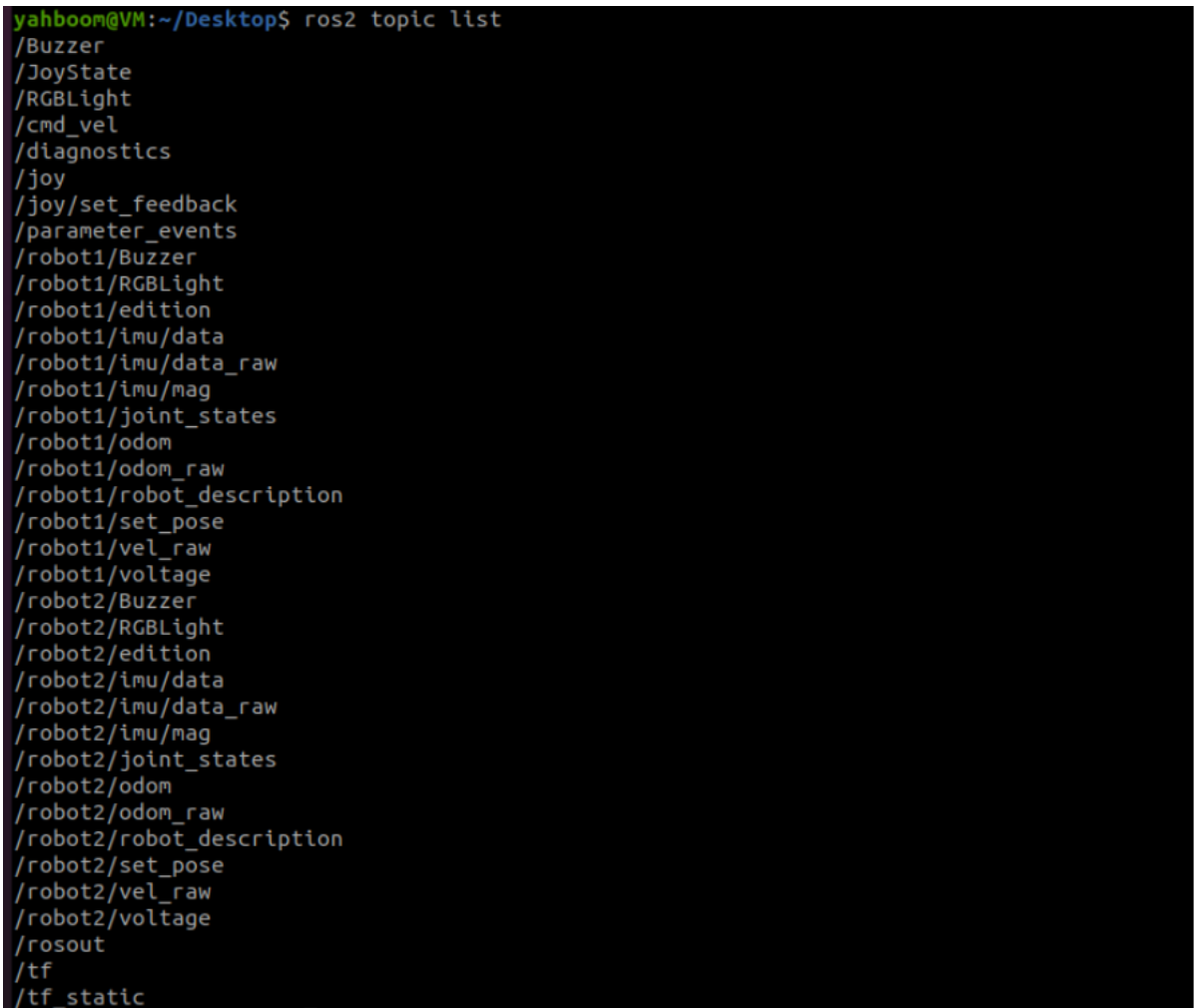
```
#raspi5Master execution
#ros2 run yahboomcar_ctrl yahboom_joy_R2
```

[robot_name] indicates the serial number of the vehicle to start. Currently, the program can choose to start robot1 and robot2.

3.2. View topic nodes

Enter the following command in the virtual machine terminal,

```
ros2 topic list
```

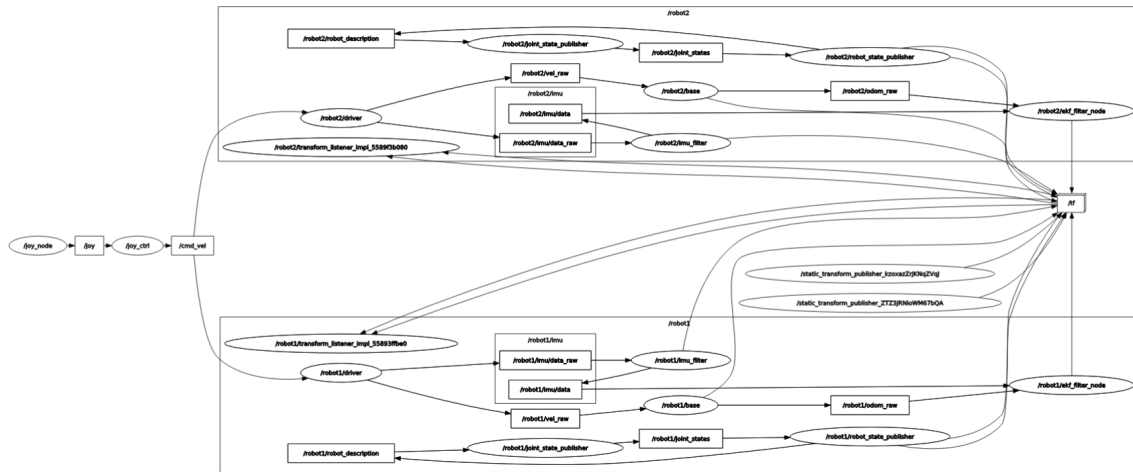
A terminal window with a black background and green text. The prompt is 'yahboom@VM:~/Desktop\$'. The command 'ros2 topic list' has been executed, resulting in a long list of topics. The topics are listed as follows:

```
yahboom@VM:~/Desktop$ ros2 topic list
/Buzzer
/JoyState
/RGBLight
/cmd_vel
/diagnostics
/joy
/joy/set_feedback
/parameter_events
/robot1/Buzzer
/robot1/RGBLight
/robot1/edition
/robot1/imu/data
/robot1/imu/data_raw
/robot1/imu/mag
/robot1/joint_states
/robot1/odom
/robot1/odom_raw
/robot1/robot_description
/robot1/set_pose
/robot1/vel_raw
/robot1/voltage
/robot2/Buzzer
/robot2/RGBLight
/robot2/edition
/robot2/imu/data
/robot2/imu/data_raw
/robot2/imu/mag
/robot2/joint_states
/robot2/odom
/robot2/odom_raw
/robot2/robot_description
/robot2/set_pose
/robot2/vel_raw
/robot2/voltage
/rosout
/tf
/tf_static
```

3.3. View the node communication diagram

Enter the following command in the virtual machine terminal,

```
ros2 run rqt_graph rqt_graph
```



It can be seen that the virtual machine published the topic speed of /cmd_vel, and the chassis of robot1 and robot2 have subscribed to the topic, so when the virtual machine publishes the topic data, both cars will receive it and then control the movement of their respective chassis.

4. Core code analysis

The code on the virtual machine side is the same as the handle control code on the car side. I won't go into details here. We mainly look at the content of the car side and the startup.

The file takes the X3 car as an example. The startup file is X3_bringup_multi_ctrl.launch.xml.

```
<launch>
<arg name="robot_name" default="robot1"/>
<group>
<push-ros-namespace namespace="$(var robot_name)"/>
<!--driver_node-->
<node name="driver" pkg="yahboomcar_bringup" exec="Mcnamu_driver_x3"
output="screen">
  <param name="imu_link" value="$(var robot_name)/imu_link"/>
  <remap from="cmd_vel" to="/cmd_vel"/>
</node>
<!--base_node-->
<node name="base" pkg="yahboomcar_base_node" exec="base_node_x3" output="screen">
<param name="odom_frame" value="$(var robot_name)/odom"/>
<param name="base_footprint_frame" value="$(var robot_name)/base_footprint"/>
</node>
<!--imu_filter_node-->
<node name="imu_filter" pkg="imu_filter_madgwick" exec="imu_filter_madgwick_node"
output="screen">
<param name="fixed_frame" value="$(var robot_name)/base_link"/>
  <param name="use_mag" value="false"/>
  <param name="publish_tf" value="false"/>
  <param name="world_frame" value="$(var robot_name)/enu"/>
  <param name="orientation_stddev" value="0.05"/>
</node>
<!--ekf_node-->
<node name="ekf_filter_node" pkg="robot_localization" exec="ekf_node">
  <param name="odom_frame" value="$(var robot_name)/odom"/>
```

```

        <param name="base_link_frame" value="$(var
robot_name)/base_footprint"/>
        <param name="world_frame" value="$(var robot_name)/odom"/>
        <param from="$(find-pkg-share yahboomcar_multi)/param/x3_ekf_$(var
robot_name).yaml"/>
        <remap from="odometry/filtered" to="odom"/>
        <remap from="/odom_raw" to="odom_raw"/>
    </node>
</group>
    <include file="$(find-pkg-share
yahboomcar_description)/launch/description_x3_multi_$(var
robot_name).launch.py"/>
</launch>

```

The xml format is used here to write the launch file, which facilitates us to add namespaces in front of multiple nodes. Adding namespace is to solve the conflict caused by the same node name. We have two cars here. Take the chassis program as an example. If both chassis nodes are named driver, then after establishing multi-machine communication, this will not work. It is allowed, so we add the namespace **namespace** before the node name. In this way, the chassis node name of car 1 is /robot1/driver, and the chassis node program of car 2 is /robot2/driver. In the launch file in XML format, a group is used to specify that within this group, both the node name and the topic name need to be added with **namespace**.

```

<group>
<push-ros-namespace namespace="$(var robot_name)"/>
</group>

```

To reference the parameters defined in the launch file in XML format, use **\$(var robot_name)**. The parameter passed in here is robot_name. When entering the command, you can enter it in the command line.

```
ros2 launch yahboomcar_multi robot_name:=robot1
```

One thing to note here is that after adding the namespace, we remapped the topic name of /robot1/cmd_vel to /cmd_vel in order to receive topic messages sent by the virtual machine.

```

<node name="driver" pkg="yahboomcar_bringup" exec="Mcnamu_driver_x3"
output="screen">
<param name="imu_link" value="$(var robot_name)/imu_link"/>
<remap from="cmd_vel" to="/cmd_vel"/>
</node>

```

Regarding the passing of parameters, adding the namespace needs to be stated in the launch file, such as the following parameters,

```

<node name="base" pkg="yahboomcar_base_node" exec="base_node_x3" output="screen">
<param name="odom_frame" value="$(var robot_name)/odom"/>
<param name="base_footprint_frame" value="$(var robot_name)/base_footprint"/>
</node>

```

It can be seen that the **odom_frame** parameter is assigned to **\$(var robot_name)/odom**, and the **base_footprint_frame** parameter is assigned to **\$(var robot_name)/base_footprint**.

If some parameters do not require a namespace, then pass them in in the form of a parameter list, for example,

```
<param from="$(find-pkg-share yahboomcar_multi)/param/x3_ekf_$(var robot_name).yaml"/>
```

What needs to be noted in this parameter table is that you need to add the namespace to find it accurately. If we start robot1, then the started node becomes robot1/ekf_filter_node, so the beginning of the parameter file should be,

```
### ekf config file ###  
robot1/ekf_filter_node:  
  ros__parameters:
```