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

Raspberry Pi PI5 master control, you need to enter the docker container first, Orin motherboard does not need to enter,

The location of the function source code is

```
~/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_multi/launch
```

The file for virtual machine startup, the source code location is,

```
~/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_multi/yahboomcar_multi
```

3. Program startup

Note: Here you need to set up multi-machine communication 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 ros environment, the virtual machine's **ROS_DOMAIN_ID** must be the same, in order to be able to perform distributed communication. Setting method reference: [06], Linux operating system -> [04], multi-machine communication configuration

3.1, startup command

** For Raspberry Pi PI5 master control, you need to enter the docker container of car 1 and car 2 respectively, but you do not need to enter the docker container of Orin motherboard.**

According to the actual model, enter in the terminal

```
#X3 model launch
#Car 1
ros2 launch yahboomcar_multi X3_bringup_multi_ctrl.launch.xml robot_name:=robot1
#Car 2
ros2 launch yahboomcar_multi X3_bringup_multi_ctrl.launch.xml robot_name:=robot2
#R2 model launch
#Car 1
ros2 launch yahboomcar_multi R2_bringup_multi_ctrl.launch.xml robot_name:=robot1
#Car 2
ros2 launch yahboomcar_multi R2_bringup_multi_ctrl.launch.xml robot_name:=robot2

#Virtual machine launch
#The handle program currently only supports the launch of the same model of car, X3 model
ros2 launch yahboomcar_multi X3_joy_ctrl.launch.py
#raspi5 master control execution
#ros2 run yahboomcar_ctrl yahboom_joy_X3
```

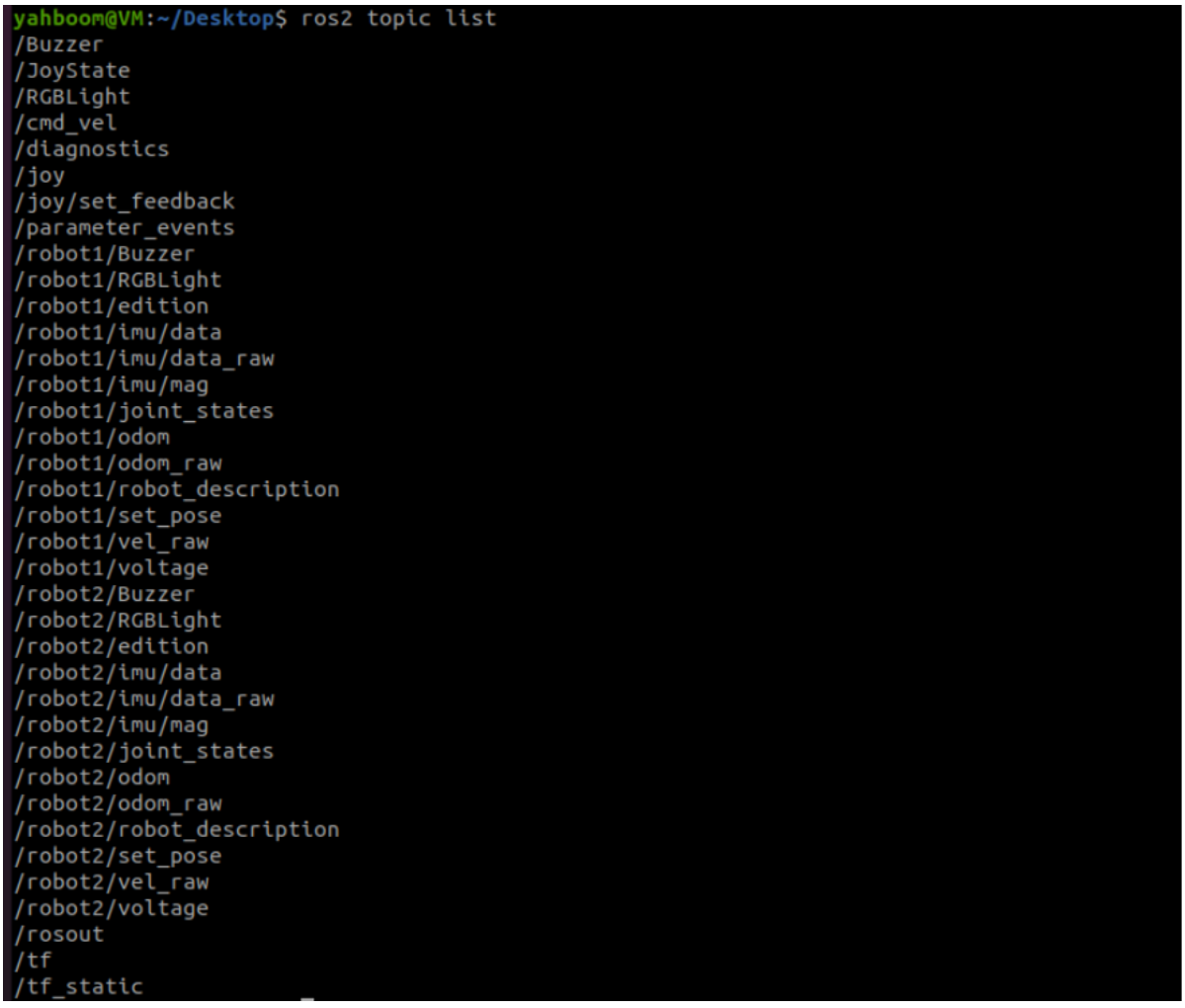
```
#The handle program currently only supports starting the car of the same model,
R2 model
ros2 launch yahboomcar_multi R2_joy_ctrl.launch.py
#raspi5 main control execution
#ros2 run yahboomcar_ctrl yahboom_joy_R2
```

[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,

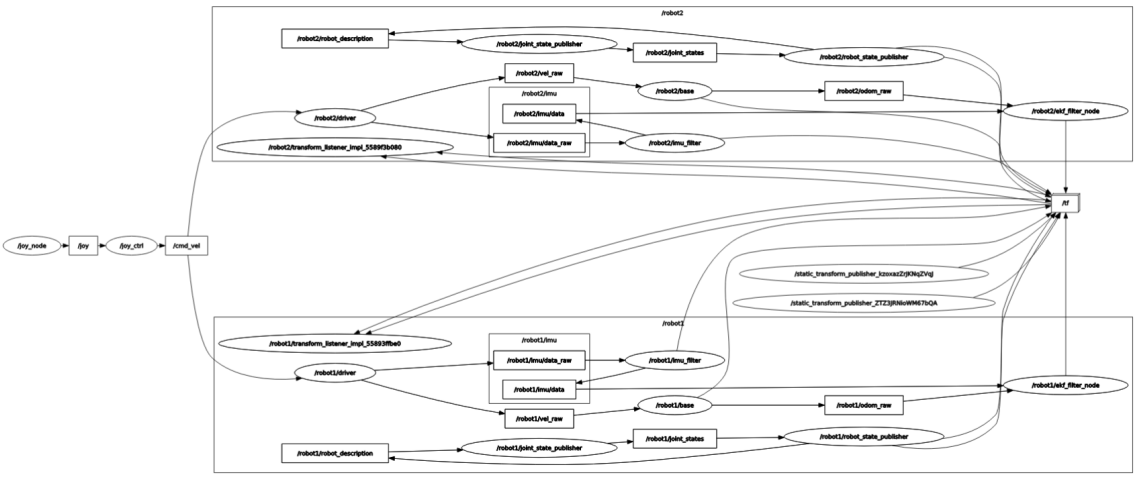
```
ros2 topic list
```



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:
```