Multimodal visual understand + visual following(Voice Version)

1. Course Content

1. Learn to use the robot's vision combined with the car-following function
2. Analyze newly discovered key source code

2. Preparation

2.1 Content Description

> This lesson uses the Raspberry Pi as an example. For Raspberry Pi and Jetson-Nano boards, you need to open a terminal on the host computer and enter the command to enter the Docker container. Once inside the Docker container, enter the commands mentioned in this lesson in the terminal. For instructions on entering the Docker container from the host computer, refer to **[01. Robot Configuration and Operation Guide] -- [4.Enter Docker (For JETSON Nano and RPi 5)]**. For RDKX5 and Orin boards, simply open a terminal and enter the commands mentioned in this lesson.

📝 This example uses `model:"qwen/qwen2.5-vl-72b-instruct:free","qwen-vl-latest"`

⚠ The responses from the large model may not be exactly the same for the same test command and may differ slightly from the screenshots in the tutorial. If you need to increase or decrease the diversity of the large model's responses, refer to the section on configuring the decision-making large model parameters in the **[03.AI Model Basics] -- [5.Configure AI large model]**.

⚡ It is recommended that you first try the previous visual example. This example adds voice functionality to the singleton example. The functionality is largely the same, so I will not further debug the program and describe the results in detail! ! !

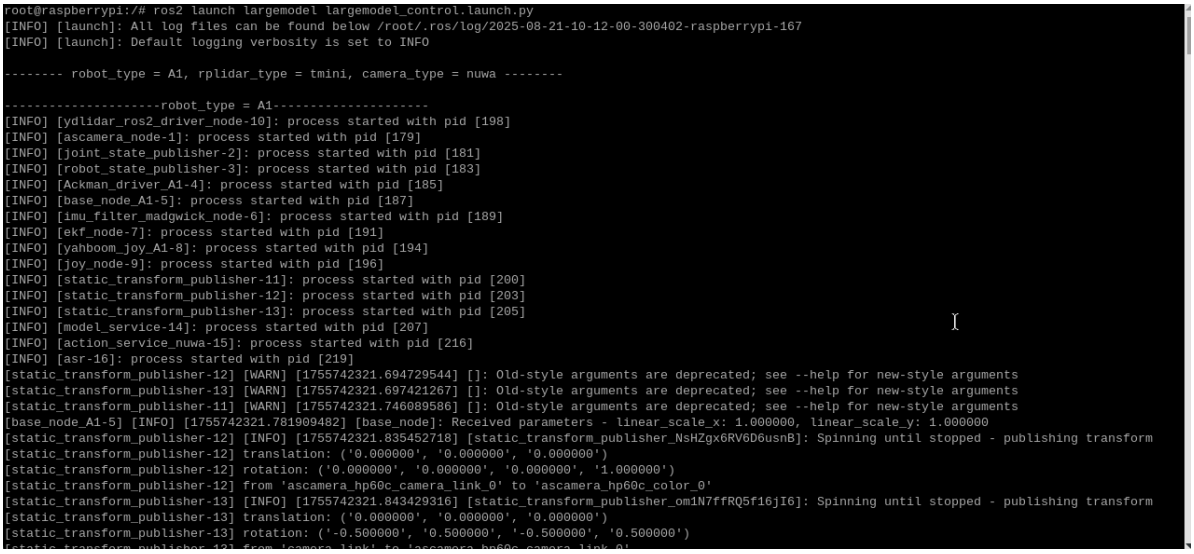
3. Running the Example

3.1 Starting the Program

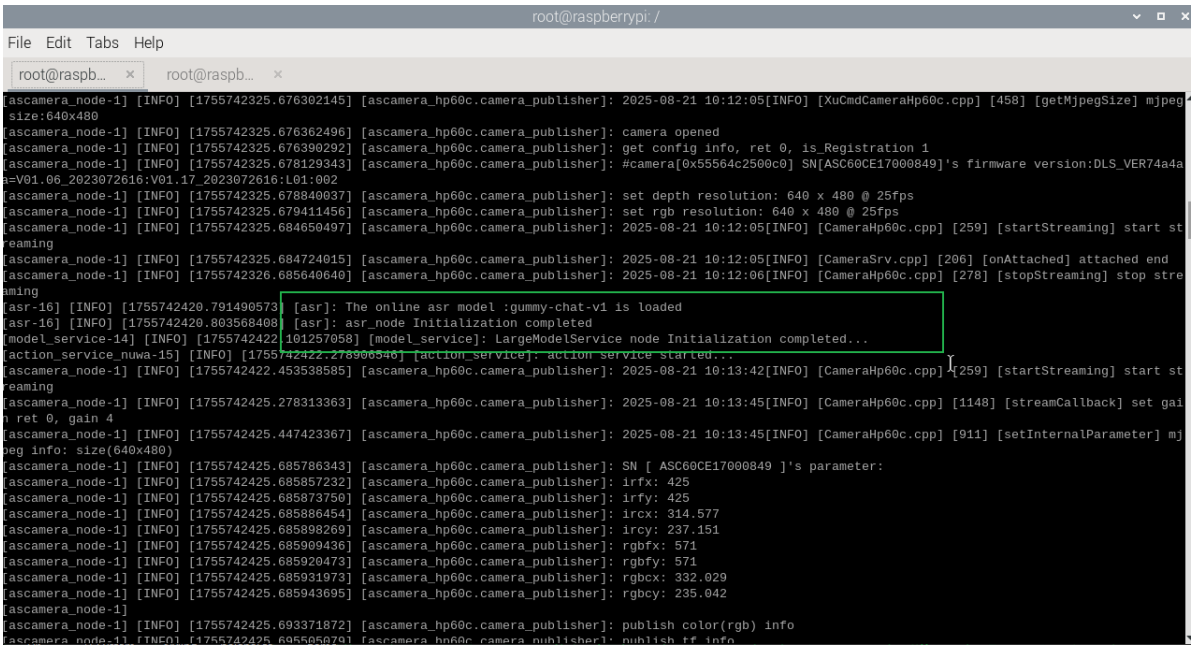
For Raspberry Pi PI5 and jetson nano, you need to enter the Docker container first. For RDKX5 and Orin main controllers, this is not necessary.

Open a terminal in Docker and enter the following command:

```
ros2 launch largemodel largemodel_control.launch.py
```



After initialization is complete, the following content will be displayed.



3.2 Test Case

Here are some reference test cases. Users can create their own test commands.

- Start Follow ××

  Color/Face/Object/Machine Code/QR Code/Gesture Recognition/Human Posture

  Color tracking, including red, green, blue, and yellow (color calibration is required according to the **AI Large Model Preparation** tutorial).

  Object tracking

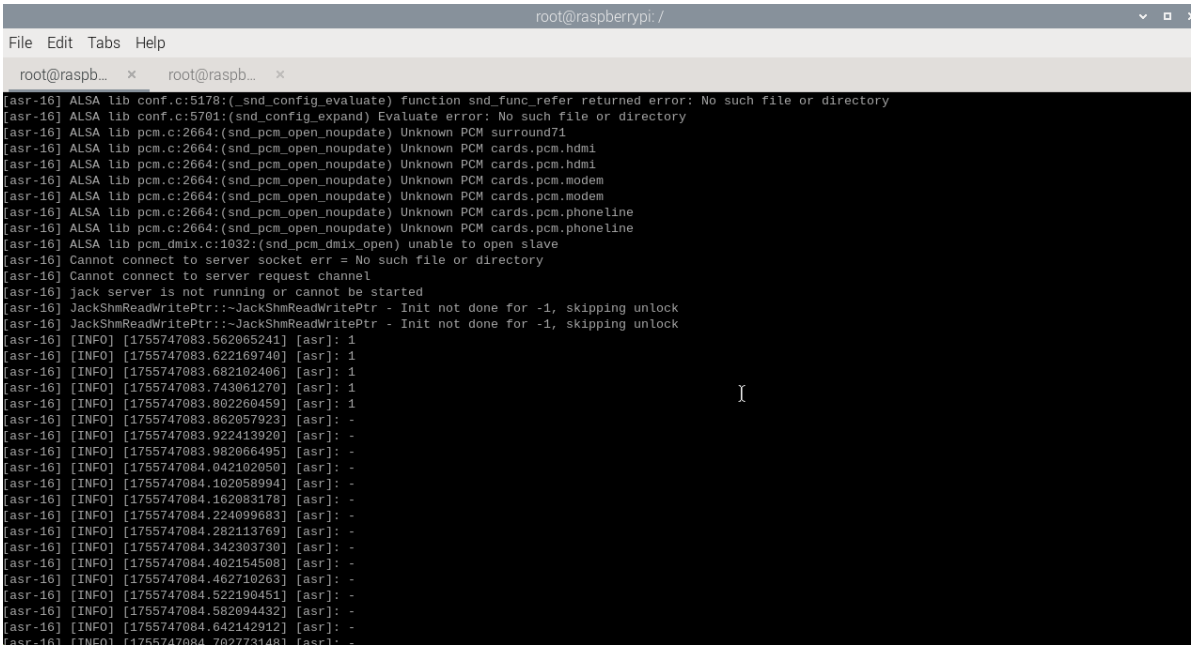
- ⚡ Note for gimbal servo USB camera users: **No object following functionality**!
  - The following example uses the principle of near-increase-increase and far-out-increase. The tracking effect is dependent on the size of the object being followed and the area detected. For program debugging, please refer to the single example.

Following Examples	Recommended Reference Object Size
Color Following	3x3cm Building Blocks
Face Following	Face from a Mobile Phone Image
AprilTag Machine Code Following	3x3cm Building Blocks
QR Code Following	QR Code Printed on A5 Paper
Mediapipe Gesture Following	Real-person gestures
Meidpipe Human Posture Following	Real-person posture

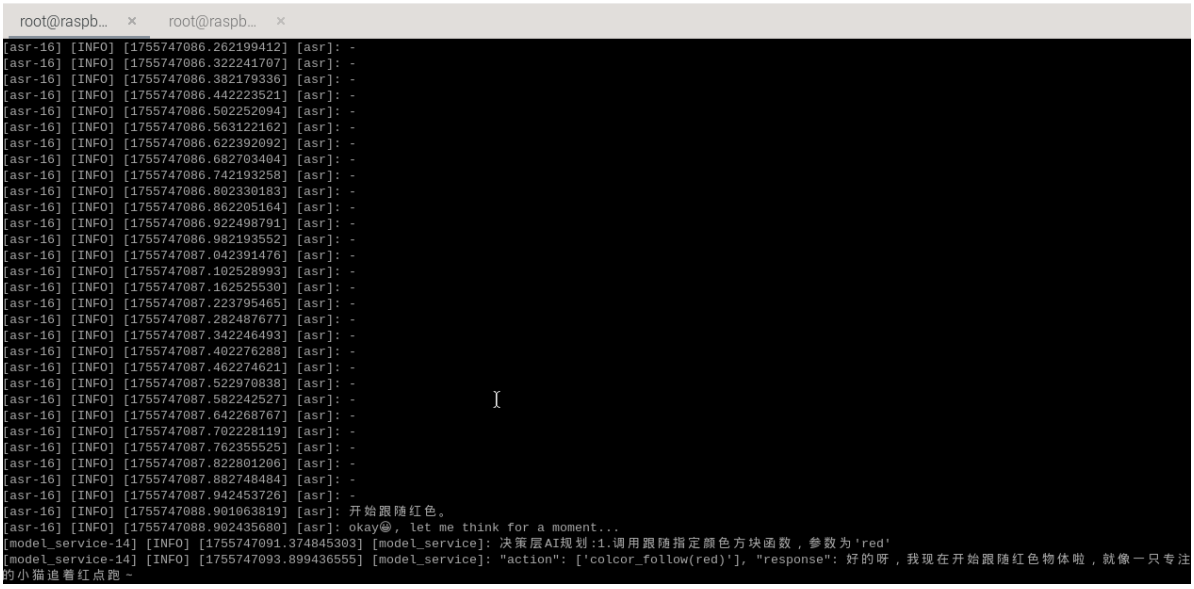
3.2.1 Example 1: "Start Following Red"

First, wake the robot with "Hi, yahboom" The robot responds, "I'm here, please." After the robot responds, the buzzer beeps briefly (beep—). The user can then speak, and the robot will detect sound activity. If there is sound activity, it will print 1; if there is no sound activity, it will print -. When the speech ends, it will detect the end of the tone. If there is silence for more than 450ms, the recording will stop.

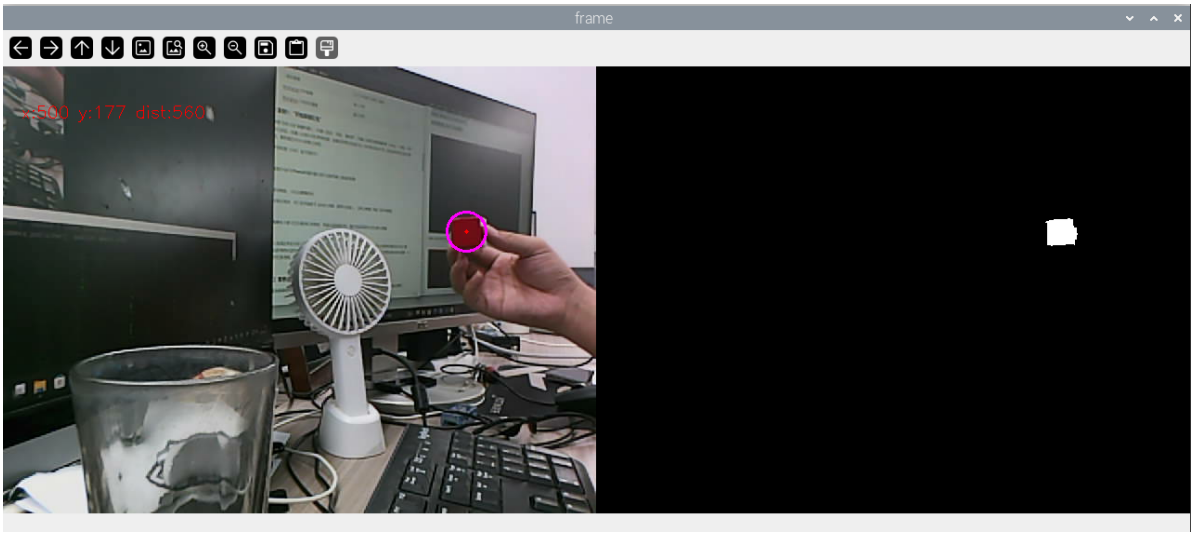
The following figure shows the dynamic voice detection (VAD):



The robot will first communicate with the user, then respond to the user's instructions. The terminal will print the following information:

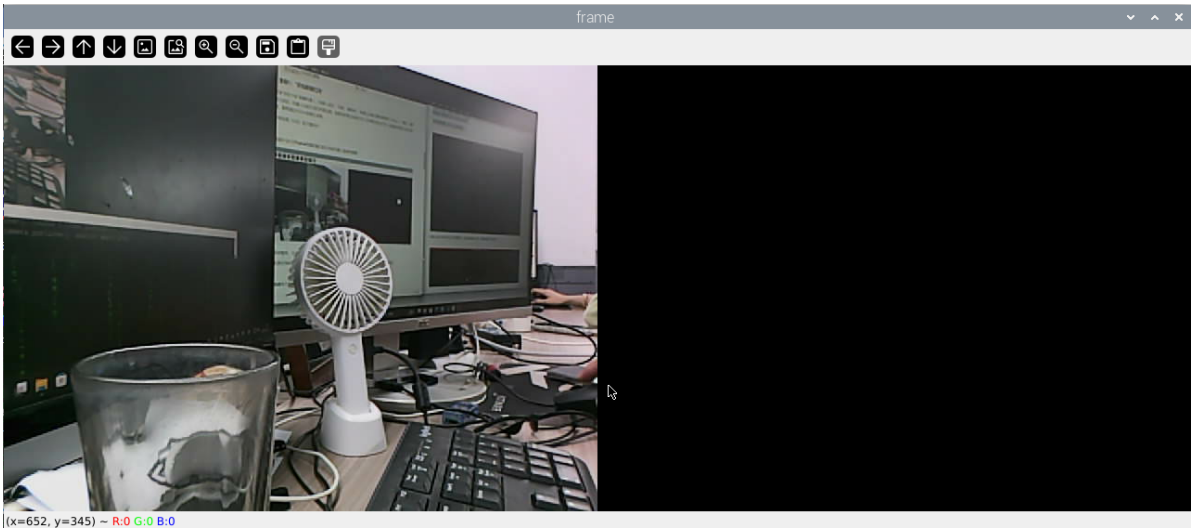


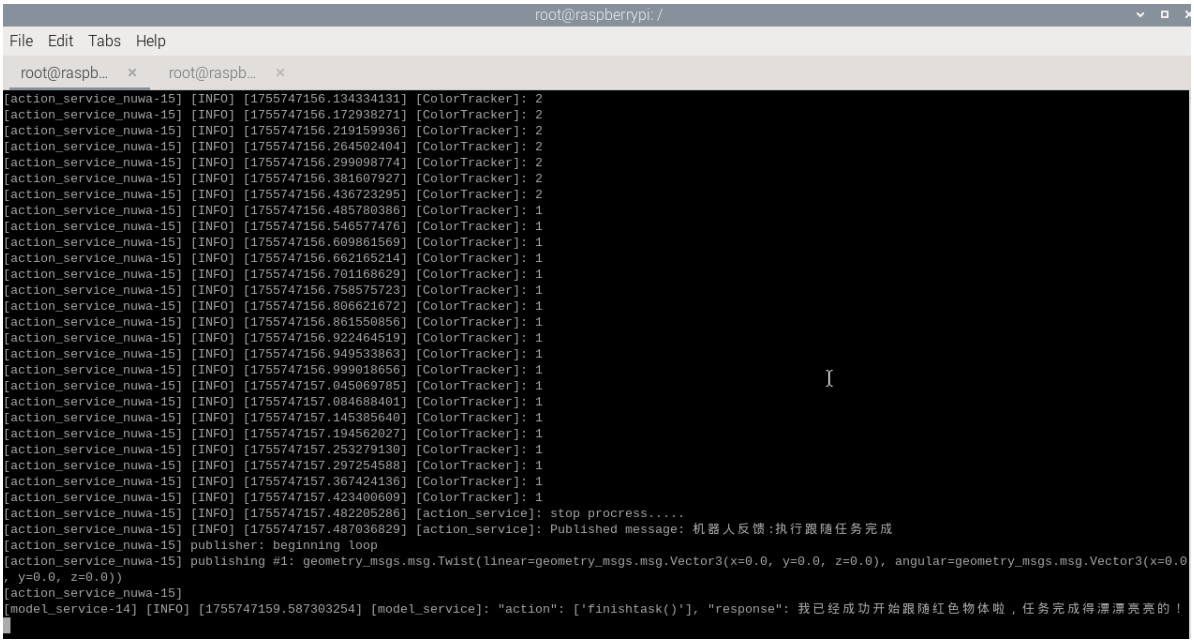
A window titled **frame** will open on the VNC screen, displaying the current robot-viewing angle.



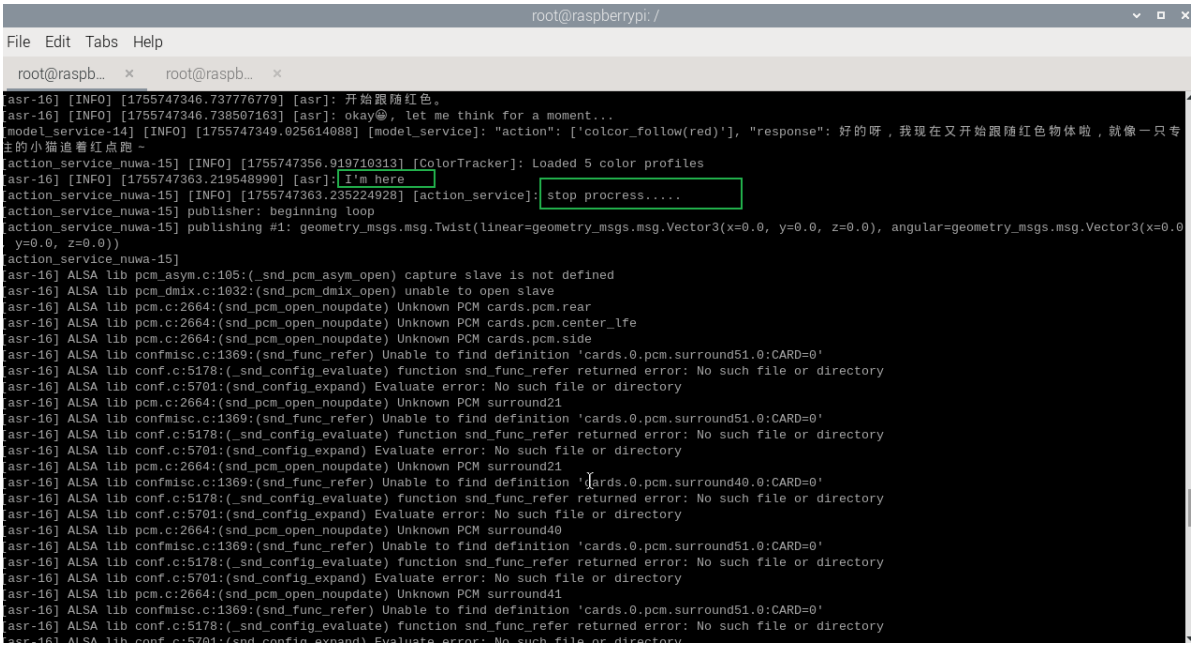
If the object is moving slowly, the robot will follow it.

If there is no target to track in the image, the program will count the time. After 10 seconds, a 5-second countdown will be printed on the terminal, and the process will automatically terminate, indicating the task is complete.

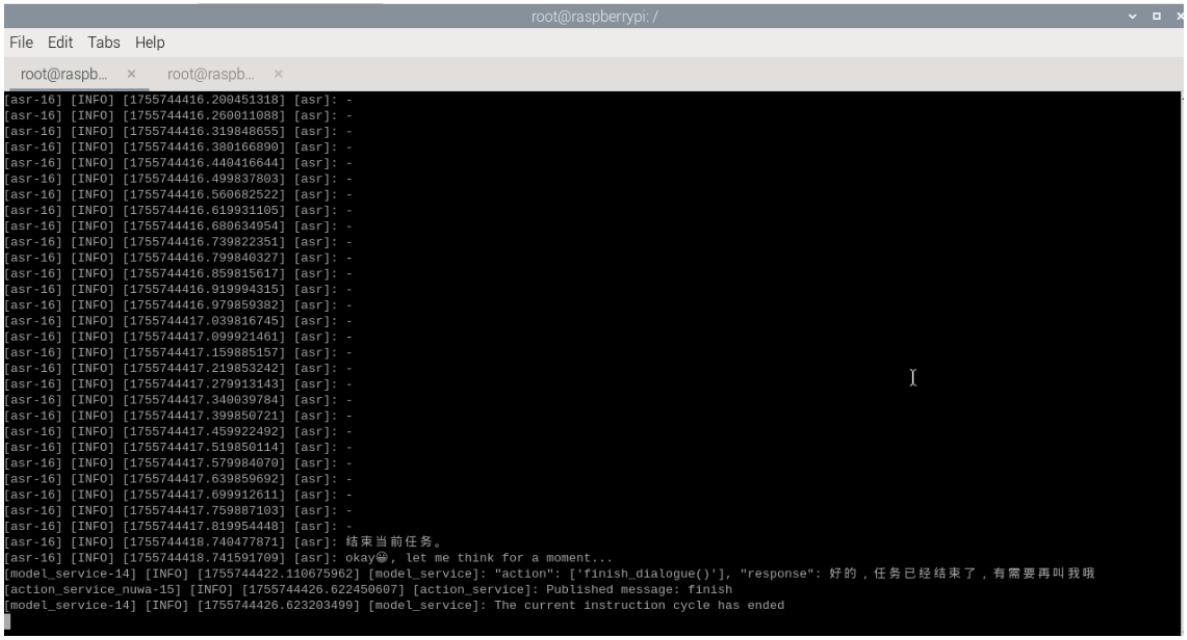




To manually terminate the task, wake the robot with the voice command "Hi, yahboom" The robot will respond, "I'm here, please." This will interrupt the program, automatically terminate the process, and allow you to proceed to the next command.



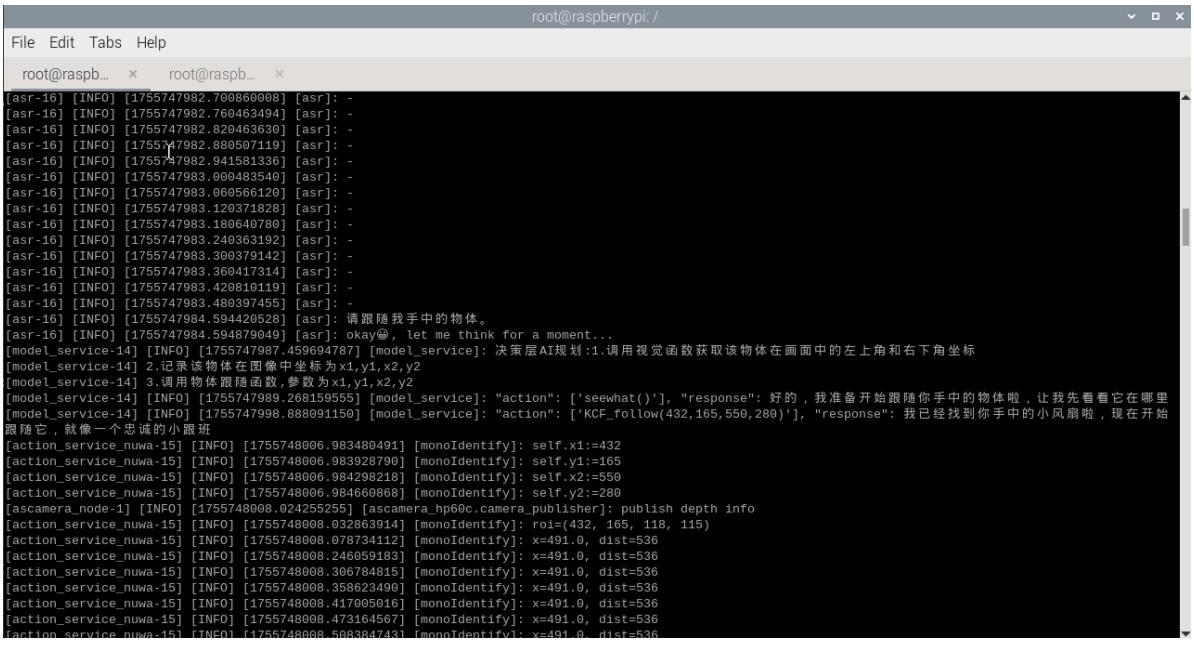
The robot will then enter a free conversation mode again, but all conversation history will be retained. At this point, you can wake yahboom up again and click "End Current Task" to end the current task cycle, clear the conversation history, and start a new one.



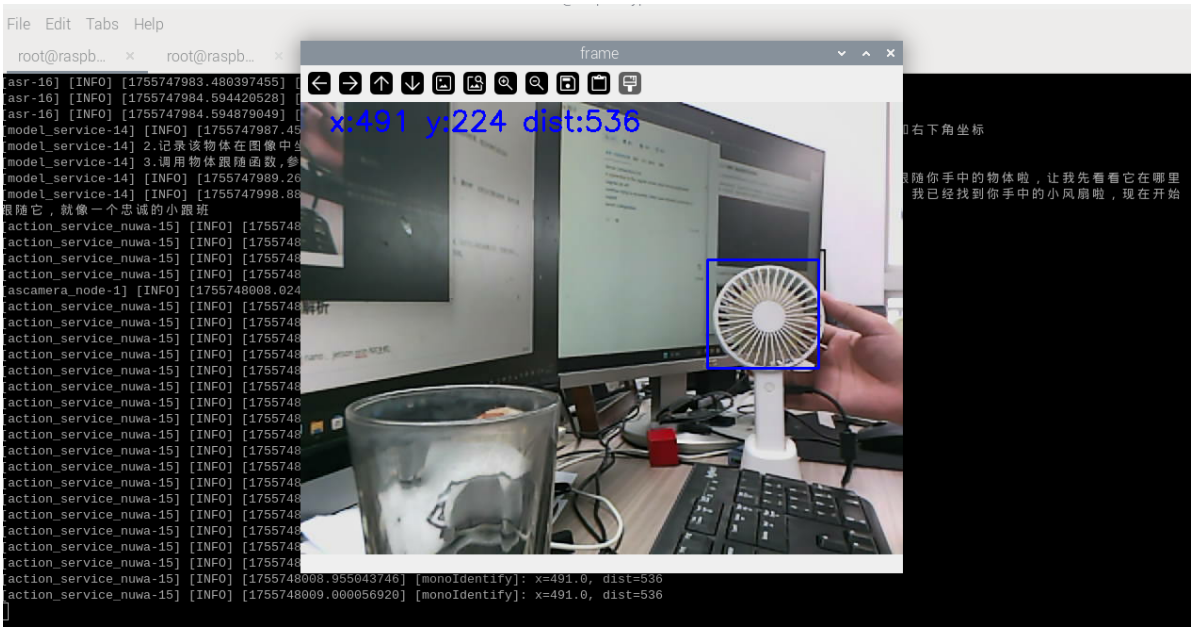
3.2.2 Example 2: "Please follow the object in my hand" (Depth Camera)

⚠ The coordinates obtained by following the object in this example are entirely derived from the inference of the large AI model. Therefore, it is recommended to use a newer model for better results!

Similar to the test in Example 1, first wake the robot with "Hello yahboom." After the robot responds, the buzzer will beep briefly (beep--). The user can then speak. After the speech is completed, the robot will respond to the user and move according to the command. Hold any object in your hand and hold it in the field of view until the tracking box appears.

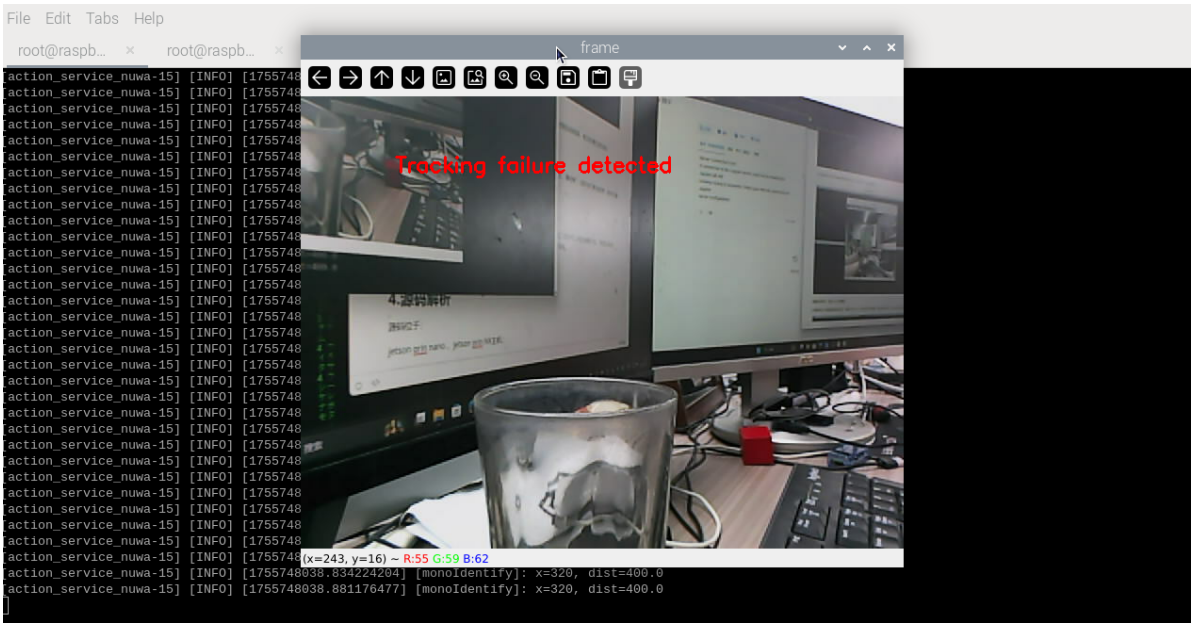


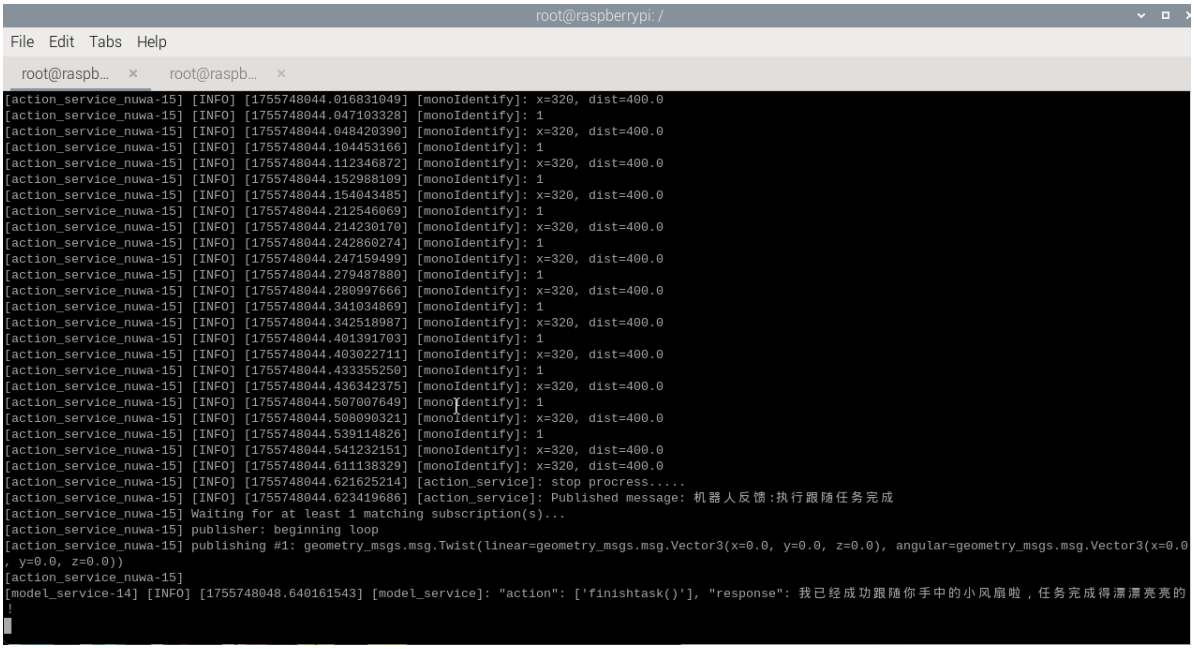
A window titled **frame** will open on the VNC screen, displaying the current robot's view.



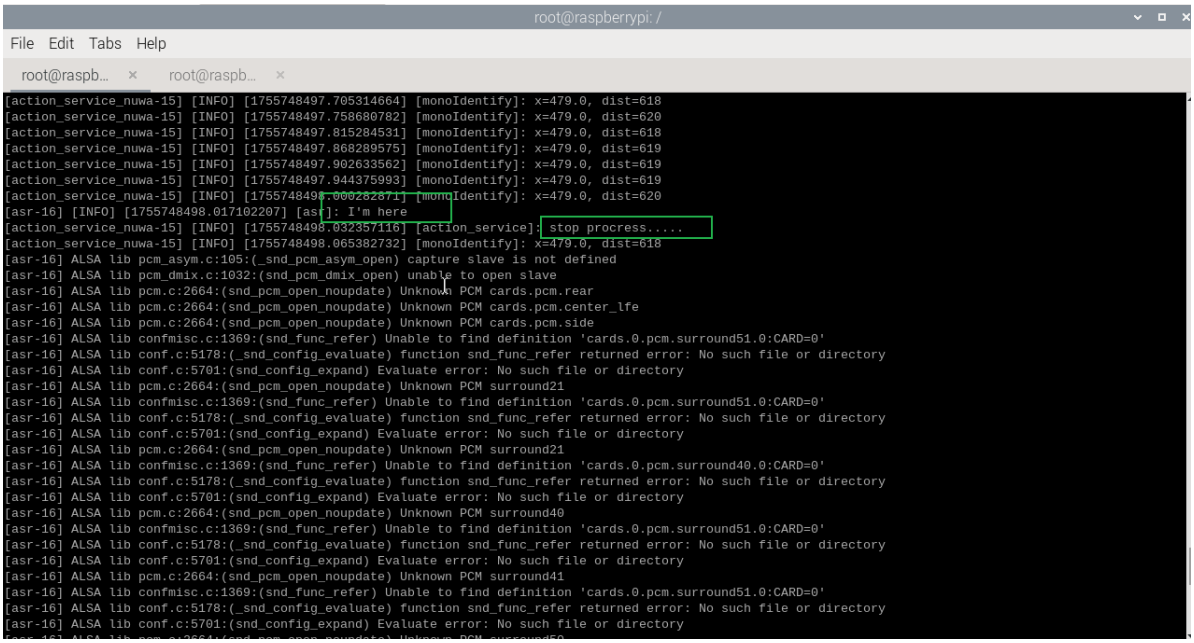
Move the object slowly, and the robot will follow.

If there is no target to follow in the image, the program will count down for 10 seconds, and the terminal will print a 5-second countdown. The process will automatically end, and the task will be considered complete.

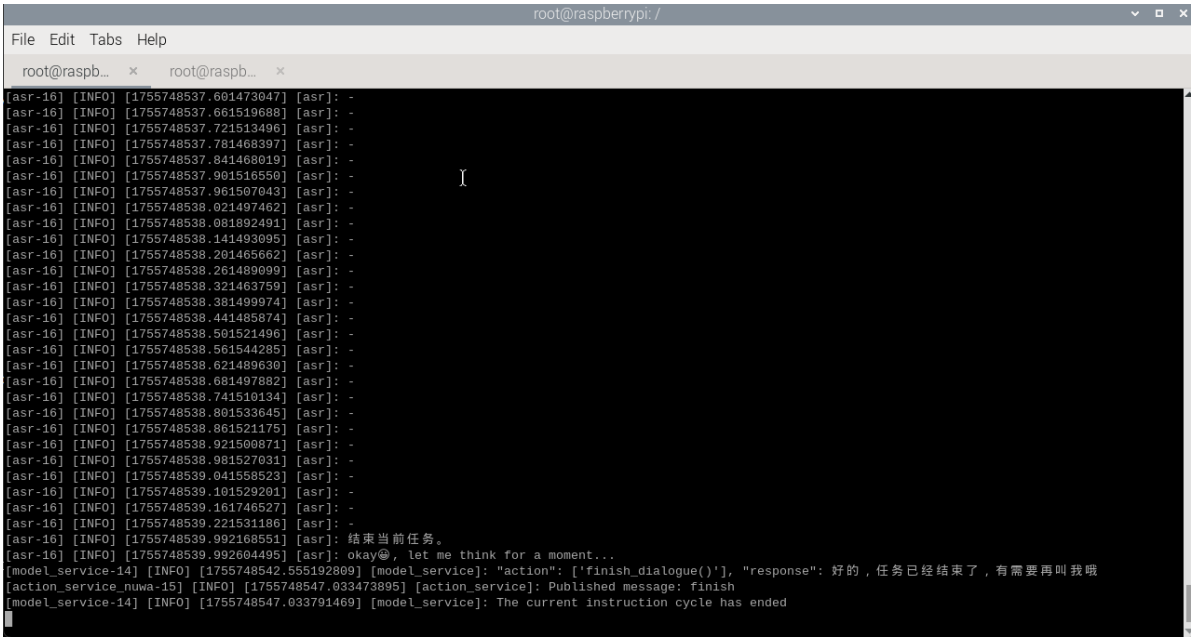




To manually end a task, wake up the robot with the voice command "Hi, yahboom" The robot will respond, "I'm here, please." This will interrupt the program and automatically end the process, allowing you to proceed to the next command.



The robot will now enter a free conversation mode, but all conversation history will be retained. You can wake yahboom up again and select "End Current Task" to end the current task cycle, clear the conversation history, and start a new one.



4. Source Code Analysis

Source code is located at:

Jetson Orin Nano:

```
#NUWA camera user
/home/jetson/yahboomcar_ros2_ws/yahboomcar_ws/src/largemodel/largemodel/action_service_nuwa.py
#USB camera user
/home/jetson/yahboomcar_ros2_ws/yahboomcar_ws/src/largemodel/largemodel/action_service_usb.py
```

```
/home/jetson/yahboomcar_ros2_ws/yahboomcar_ws/src/largemodel/largemodel/model_service.py
```

RDK X5：

```
#NUWA camera user
/home/sunrise/jetson/yahboomcar_ros2_ws/yahboomcar_ws/src/largemodel/largemodel/action_service_nuwa.py
#USB camera user
/home/sunrise/jetson/yahboomcar_ros2_ws/yahboomcar_ws/src/largemodel/largemodel/action_service_usb.py
```

```
/home/sunrise/yahboomcar_ros2_ws/yahboomcar_ws/src/largemodel/largemodel/model_service.py
```

Jetson Nano, Raspberry Pi host:

You need to first enter Docker.

```
#NUWA Camera User
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/largemodel/largemodel/action_service_nuwa.py
#USB Camera User
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/largemodel/largemodel/action_service_usb.py
```

```
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/largemodel/largemodel/model_service.py
```

4.1 Example 1

4.1.1 Nuwa Depth Camera, action_service_nuwa.py

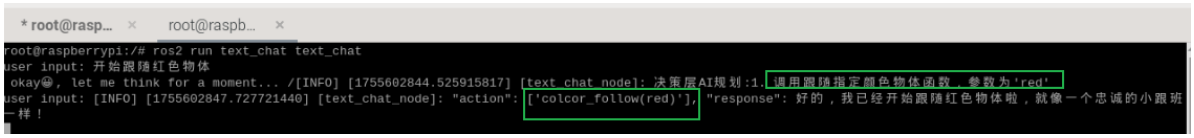
Example 1 uses the **seewhat**, **colcor_follow**, and **stop_follow()** methods in the **CustomActionServer** class.

The **seewhat** function primarily retrieves the color image from the depth camera.

The **colcor_follow(self, color)** function performs color following.

The **stop_follow()** function issues a stop command to follow.

This section focuses on the **colcor_follow(self, color)** function, which requires a color parameter, which can be 'red', 'green', 'blue', or 'yellow'.




```
#Start the color line patrol subprocess
    process_1 = subprocess.Popen(['ros2', 'run', 'yahboomcar_voice_ctrl_depth', 'voice_colorTracker','--ros-args','-p',f'colcor:={target_color}'])
```

The source code path of the subprocess in the program,

`~/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_voice_ctrl_depth/yahboomcar_voice_ctrl_depth/follow/voice_colorTracker.py`

```
def colcor_follow(self,color):
    self.colcor_follow_future = Future() #Reset Future object
    color = color.strip("'\"")  # Remove single and double quotes
    if color == 'red':
        target_color = float(1)
    elif color == 'green':
        target_color = float(2)
    elif color == 'blue':
        target_color = float(3)
    elif color == 'yellow':
        target_color = float(4)
    else:
        self.get_logger().info('color_sort:error')
        return

    #Start the color line patrol subprocess
```

```
    process_1 = subprocess.Popen(['ros2', 'run', 'yahboomcar_voice_ctrl_depth',
'voice_colorTracker','--ros-args','-p',f'colcor:={target_color}'])

    #Waiting to stop following instructions
    while not self.colcor_follow_future.done():
        if self.interrupt_flag:
            break
        time.sleep(0.1)

    self.kill_process_tree(process_1.pid)
    self.cancel()
```

When the main model receives the user input of the "Stop Following" or "End Following" command,

or if the target being followed is lost for more than 10 seconds,

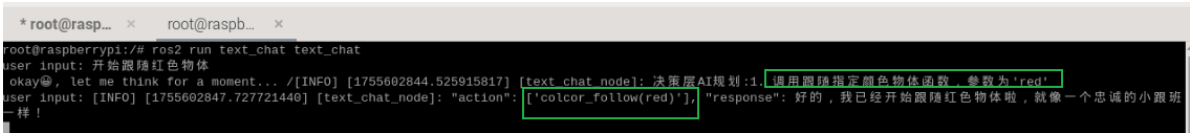
the **stop_follow** method is called, sending the future.done signal. The `while not self.colcor_follow_future.done()` block in the **colcor_follow** function then exits. The **kill_process_tree** method is then called to recursively kill the child process tree. Finally, the status of the execution action is reported to the main model at the execution layer.

4.1.2 USB Camera, action_service_usb.py

Example 1 uses the **seewhat**, **colorFollow**, and **stop_track()** methods in the **CustomActionServer** class.

- The **seewhat** function primarily retrieves the camera's color image.
- The **colorFollow(self, color)** function performs color tracking.
- The **stop_track()** function issues a stop tracking command.

This section focuses on the **colorFollow(self, color)** function, which requires a color parameter (red, green, blue, and yellow).




```
#Start the color line patrol subprocess
process_1 = subprocess.Popen(['ros2', 'run', 'yahboomcar_voice_ctrl',
'colorFollow','--ros-args','-p',f'target_color:={target_color}'])
```

The source code path of the subprocess in the program,

`~/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_voice_ctrl/yahboomcar_voice_ctrl/colorFollow.py`

```
def colorFollow(self,color):
    try:
        self.colorFollow_future = Future()
        color = color.strip("'\"")
        if color == 'red':
            target_color = int(1)
        elif color == 'green':
            target_color = int(2)
        elif color == 'blue':
            target_color = int(3)
```

```
        elif color == 'yellow':
            target_color = int(4)
        else:
            target_color = int(1)
        process_1 = subprocess.Popen(['ros2', 'run', 'yahboomcar_voice_ctrl',
'colorFollow','--ros-args','-p',f'target_color:={target_color}'])
        while not self.colorFollow_future.done():
            if self.interrupt_flag:
                break
            time.sleep(0.1)
        self.get_logger().info(f'killed process_pid')
        self.kill_process_tree(process_1.pid)
        self.cancel()
    except:
        self.get_logger().error('colorFollow Startup failure')
        return
```

When the main model receives the user input of the "Stop Following" or "End Following" command,

or if the tracking target is lost for more than 10 seconds,

the **stop_track** method is called, sending the future.done signal. The `while not self.colorFollow_future.done()` block in the **colorFollow** function then exits. The **kill_process_tree** method is then called to recursively kill the child process tree. Finally, the status of the action execution is reported to the main model at the execution layer.

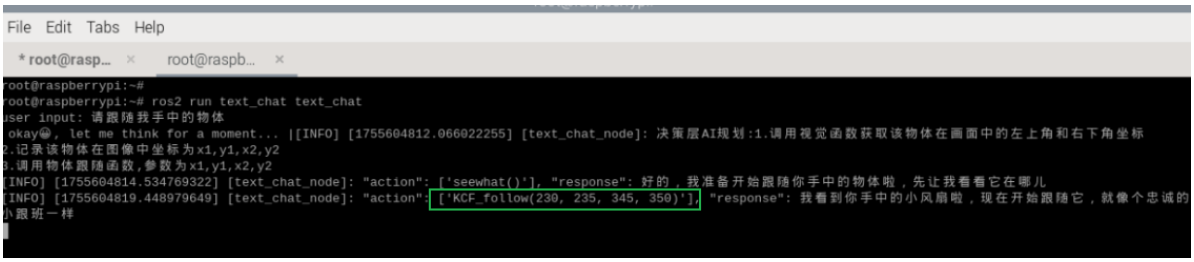
4.2 Example 2

4.2.1 nuwa Depth Camera, action_service_nuwa.py

Example 2 uses the **seewhat**, **KCF_follow**, and **stop_track** methods in the **CustomActionServer** class.

- The **seewhat** function primarily obtains the camera's color image.
- The **KCF_follow(self,x1,y1,x2,y2)** function performs object tracking. - **stop_track()** issues a stop command to follow the function.

The **seewhat** function primarily retrieves the camera's color image. The **KCF_follow(self,x1,y1,x2,y2)** function takes as parameters the coordinates of the upper-left and lower-right vertices of the object's bounding box (the upper-left corner of the image is the pixel origin). For example, the coordinates of the outer bounding box of the identified green square in Example 2 can be found from the large model's response: the upper-left corner is (230, 345) and the lower-right corner is (235, 350).




```
#Start the object tracking subprocess
process_1 = subprocess.Popen(['ros2', 'run', 'yahboomcar_voice_ctrl_depth',
'voice_KCF_Tracker','--ros-args','-p',f'x1:={x1}','-p',f'y1:={y1}','-p',f'x2:=
{x2}','-p',f'y2:={y2}'])
```

The startup program source code path is:

`~/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_voice_ctrl_depth/yahboomcar_voice_ctrl_depth/kcf/voice_KCF_Tracker.py`

```
def KCF_follow(self,x1,y1,x2,y2):
    self.KCF_follow_future = Future() #Reset Future object
    x1 = int(x1)
    y1 = int(y1)
    x2 = int(x2)
    y2 = int(y2)

    process_1 = subprocess.Popen(['ros2', 'run', 'yahboomcar_voice_ctrl_depth', 'voice_KCF_Tracker','--ros-args','-p',f'x1:={x1}','-p',f'y1:={y1}','-p',f'x2:={x2}','-p',f'y2:={y2}'])
    # time.sleep(1.0)#Sleep for 2 seconds to wait for the thread to stabilize

    while not self.KCF_follow_future.done():
        if self.interrupt_flag:
            break
        time.sleep(0.1)

    self.kill_process_tree(process_1.pid)
    self.cancel()
```

When the main model receives the user input of the "Stop Following" or "End Following" command,

or if the target is lost for more than 10 seconds,

the **stop_follow** method is called, sending the future.done signal. The `while not self.KCF_follow_future.done()` block in the **KCF_follow** function is then exited. The **kill_process_tree** method is then called to recursively kill the child process tree. Finally, the status of the execution action is reported to the main model at the execution layer.