轮 趣 科 技

ROS2-WHEELTEC 机器人的

ROS 和 STM32 通信过程

推荐关注我们的公众号获取更新资料

版本说明：

版本	日期	内容说明
V1.0	2025/05/20	第一次发布

网址:www.wheeltec.net

目录

1. WHEELTEC 机器人的 ROS2 和 STM32 通信过程

Wheeltec 机器人中包含了两个核心控制器和一些常用的传感器，如雷达、相机、IMU 等。两个核心控制器分别是 ROS 主控和 STM32 控制器。常见的 ROS 主控设备有 Jetson Orin Nano、Jetson Orin NX 等英伟达 Jetson 系列开发板、树莓派、RDK X3、RDK X5 鲁班猫等，主要用于 ROS2 开发。Wheeltec 机器人通常使用 STM32F103 和 STM32F407 作为主要的 STM32 控制器，用于采集里程计信息、陀螺仪信息和点击控制。

1.1 ROS 和 STM32 的硬件连接

① 硬件连接示意图

ROS 主控通过 USB Type-C 数据线与 STM32 开发板通信，STM32 开发板内置的 USB 转 TTL 电平转换芯片负责将信号转换为 TTL 电平，以便 STM32 芯片处理。以下是 ROS 主控与 STM32 控制器的连接示意图。

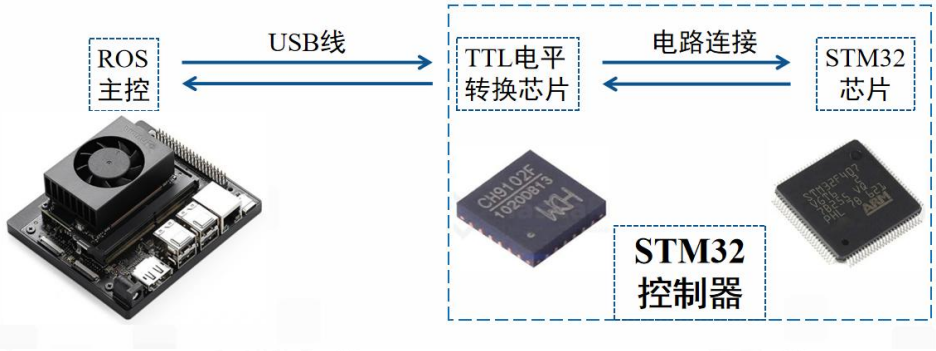

图 1-1-1ROS 主控和 STM32 硬件连接示意图

② 电平转换芯片修改串口号

ROS 主控连接上 STM32 控制器后，使用”lsusb”可以查看到 usb 设备信息，图示为电平转换芯片的设备信息，该电平转换芯片型号为 CH9102。使用”ll /dev”可以看到该设备被识别为”/dev/ttyCH343USB0”。

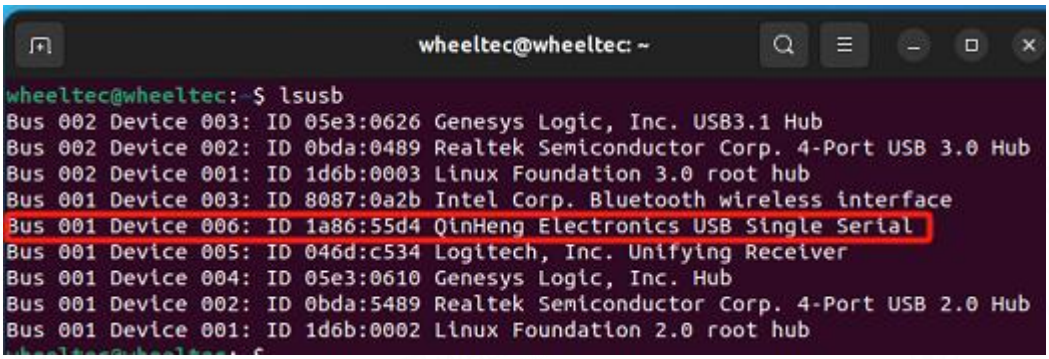

图 1-1-2 ROS 主控中的设备信息

在 Linux 系统中，USB 设备接入后通常会被识别为如 /dev/ttyCH343USB0 或 /dev/ttyUSB0 等串口设备。需要注意的是，这些设备名在每次连接时可能会发生变化，且默认可能没有读写权限，需手动赋予。为简化操作并确保稳定识别，可以通过为 USB 设备创建固定别名的方式来解决。

为了匹配 WHEELTEC ROS2 源码使用，首先需要在 windows 系统中使用 CH34xSerCfg.exe 软件修改 CH9102 的串口号为 0002。操作如下图：

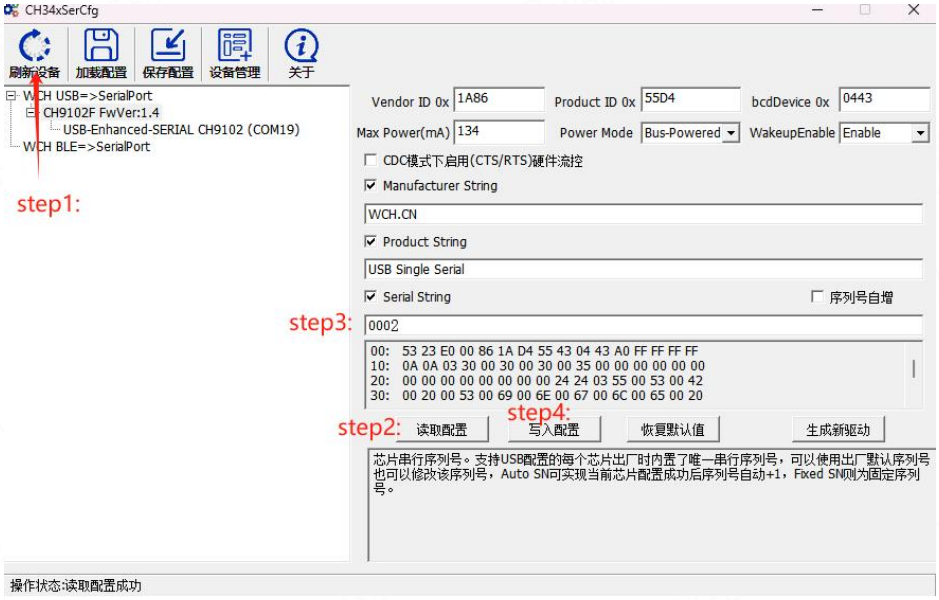

图 1-1-3 修改串口号

1.2 ROS 和 STM32 的软件连接

① 创建设备别名

ROS 主控不单只连接一个 TTL 电平转换芯片，有些传感器如雷达、惯导上的转接板也会集成有 TTL 电平转换芯片。为了避免在代码中手动指定设备号，Linux 系统可以通过为 USB 设备创建固定别名来解决该问题。别名脚本存放在

turn_on_wheeltec_robot 功能包源码文件夹里的 wheeltec_udev.sh 文件中。为了匹配 WHEELTEC ROS2 源码使用，以下为 WHEELTEC ROS 主控中外接的传感器设备号与别名说明。

传感器	设备号	设置别名
串口版雷达	0001	/dev/wheeltec_lidar
STM32 控制器	0002	/dev/wheeltec_controller
惯导 IMU	0003	/dev/wheeltec_IMU
语音传感器	0004	/dev/wheeltec_mic
GNSS 接收机	0005	/dev/wheeltec_gnss

wheeltec_udev.sh 文件中关于固定 STM32 串口号的脚本内容如图所示：

图 1-2-1 wheeltec_udev.sh 文件脚本内容

修改设备号后，将 STM32 接入 ROS 系统中，运行 wheeltec_udev.sh 文件：

```
文件赋权：sudo chmod 777 wheeltec_udev.sh
执行脚本：sudo sh wheeltec_udev.sh
```

此时重新插拔设备后，再次在终端中使用“ll /dev”查看输出时，可以看到设备已成功使用“wheeltec_controller”作为别名表示 STM32 的串口模块。此后无论连接到哪个 USB 接口，使用时都无需再考虑设备号变化的问题。

图 1-2-2 设备别名效果

1.3 turn_on_wheeltec_robot 功能包组成

turn_on_wheeltec_robot 功能包用于建立 ROS 主控与 STM32 控制板之间的通信链路，负责数据交互与控制指令的传输，实现控制指令下发与传感器数据回传的功能。turn_on_wheeltec_robot 功能包路径为：

```
/home/wheeltec/wheeltec_ros2/src/turn_on_wheeltec_robot
```

turn_on_wheeltec_robot 的文件组成如图所示。

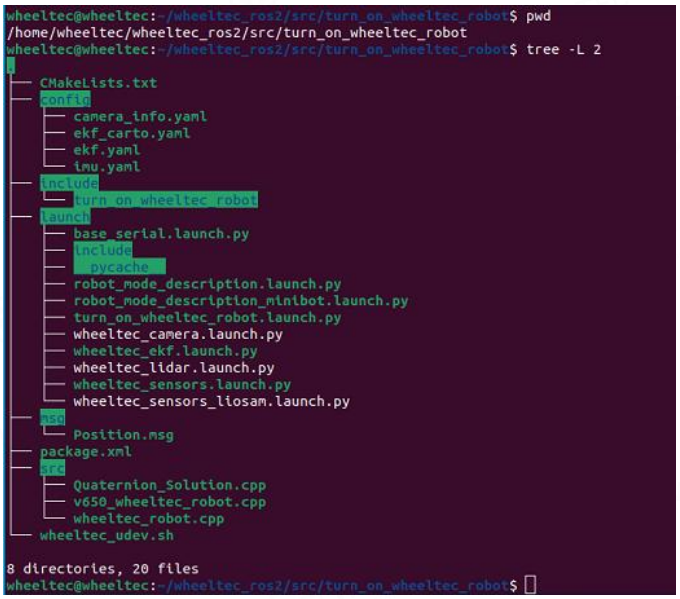

图 1-3-1　turn_on_wheeltec_robot 的文件结构

CMakeLists.txt 是 CMake 构建系统的脚本文件，用户可以在该文件中设置功能包名称、添加编译选项、指定依赖包编译 cpp 文件为可执行节点。

package.xml 用于定义功能包的元信息，包括：功能包名称、版本、描述，作者和维护者信息构建依赖和运行依赖（如 rclcpp, std_msgs 等）。

include/用于存放头文件以供 src 中的源文件引用；Msg/用于存放自定义消息类型（.msg 文件）；src/是源代码目录，包含实际 ROS 与 STM32 的通信 C++节点实现文件 wheeltec_robot.cpp。

config/用于存放配置文件，camera_info.yaml 是 astra 相机的内参文件，ekf.yaml 和 ekf_carto.yaml 是 ekf 滤波节点用到的参数文件，imu.yaml 是 wheeltec_h30 惯导用到的参数文件。

launch/用于存放启动文件，其内容已经功能如下表所示。

文件名	功能

base_serial.launch.py	启动 STM32 底层控制节点
robot_mode_description.launch.py robot_mode_description_minibot.launch.py	读取和发布机器人描述文件(URDF)中定义的关节状态和发布 TF 转换
wheeltec_camera.launch.py	启动 wheeltec 机器人中的相机传感器
wheeltec_ekf.launch.py	启动 ekf 滤波和传感器融合节点
wheeltec_lidar.launch.py	启动 wheeltec 机器人中的雷达传感器
wheeltec_sensors.launch.py	启动 wheeltec 机器人的外接传感器，包括雷达、相机和惯导
wheeltec_sensors_liosam.launch.py	启动 wheeltec 机器人在使用 lio-sam 建图算法时要用到的传感器，包括雷达、GNSS 接收机和惯导
turn_on_wheeltec_robot.launch.py	配置并启动机器人的各个核心功能节点，如底盘通信、IMU 处理、EKF 状态估计、TF 变换以及机器人模型

1.4 turn_on_wheeltec_robot.launch.py 文件说明

turn_on_wheeltec_robot.launch.py 启动文件配置并启动机器人的各个核心功能节点，如底盘通信、IMU 处理、EKF 状态估计、TF 变换以及机器人模型。文件内容如图所示。

```python
def generate_launch_description():
    bringup_dir = get_package_share_directory('turn_on_wheeltec_robot')
    launch_dir = os.path.join(bringup_dir, 'launch')
    ekf_config = Path(get_package_share_directory('turn_on_wheeltec_robot'), 'config', 'ekf.yaml')
    ekf_carto_config = Path(get_package_share_directory('turn_on_wheeltec_robot'), 'config', 'ekf_carto.yaml')
    imu_config = Path(get_package_share_directory('turn_on_wheeltec_robot'), 'config', 'imu.yaml')
    carto_slam = LaunchConfiguration('carto_slam', default='false')
    carto_slam_dec = DeclareLaunchArgument('carto_slam',default_value='false')
    wheeltec_robot = IncludeLaunchDescription(
            PythonLaunchDescriptionSource(os.path.join(launch_dir, 'base_serial.launch.py')),
            launch_arguments={'use_wheeltec_imu':'true'}.items(),)
    robot_ekf = IncludeLaunchDescription(
            PythonLaunchDescriptionSource(os.path.join(launch_dir, 'wheeltec_ekf.launch.py')),
            launch_arguments={'carto_slam':carto_slam}.items(),          )
    base_to_link = launch_ros.actions.Node(
            package='tf2_ros',
            executable='static_transform_publisher',
            name='base_to_link',
            arguments=['0', '0', '0','0', '0','0','base_footprint','base_link'],)
    base_to_gyro = launch_ros.actions.Node(
            package='tf2_ros',
            executable='static_transform_publisher',
            name='base_to_gyro',
            arguments=['0', '0', '0','0', '0','0','base_footprint','gyro_link'],)
    imu_filter_node =  launch_ros.actions.Node(
        package='imu_filter_madgwick',
        executable='imu_filter_madgwick_node',
        parameters=[imu_config])
    joint_state_publisher_node = launch_ros.actions.Node(
            package='joint_state_publisher',
            executable='joint_state_publisher',
            name='joint_state_publisher',)
    minibot_type = IncludeLaunchDescription(
            PythonLaunchDescriptionSource(os.path.join(launch_dir, 'robot_mode_description_minibot.launch.py')),
            launch_arguments={'mini_mec': 'true'}.items(),)
    flagship_type = IncludeLaunchDescription(
            PythonLaunchDescriptionSource(os.path.join(launch_dir, 'robot_mode_description.launch.py')),
            launch_arguments={'senior_akm': 'true'}.items(),)
    ld = LaunchDescription()
    #ld.add_action(minibot_type)
    ld.add_action(flagship_type)
    ld.add_action(carto_slam_dec)
    ld.add_action(wheeltec_robot)
    ld.add_action(base_to_link)
    ld.add_action(base_to_gyro)
    ld.add_action(joint_state_publisher_node)
    ld.add_action(imu_filter_node)
    ld.add_action(robot_ekf)
    return ld
```

图 1-4-1 turn_on_wheeltec_robot.launch.py 的文件内容

turn_on_wheeltec_robot.launch.py 启动文件包含 8 个节点，在文件末尾处 ld.add_action 将所有节点和配置添加到启动描述中，供 ROS2 启动器执行。从上到下解析为：

ld.add_action(flagship_type)——加载不同型号的机器人如（MiniBot 或旗舰版）的 URDF 模型配置。flagship_type 表示加载旗舰版 senior_akm 型号的机器人模型配置，minibot_type 表示加载 mini_mec 型号的机器人模型配置。用户可以通过修改[launch_arguments]的参数来修改机器人型号。注意：flagship_type 和 minibot_type 不能同时运行。

ld.add_action(carto_slam_dec)——声明了一个名为 carto_slam 的启动参数，默认值为 “false”。在 wheeltec_cartographer 功能中会修改这个参数值为“true”，用来区分使用不同的 robot_ekf 配置参数。

ld.add_action(wheeltec_robot)——启动子文件 launch/base_serial.launch.py，实现机器人底盘串口通信相关的功能。通过 launch_arguments 把 use_wheeltec_imu=true 传给 base_serial.launch.py，表示启用 Wheeltec 外接的 H30 惯导。

ld.add_action(base_to_link)、ld.add_action(base_to_gyro)用于广播坐标变换（base_footprint→base_link、gyro_link）的静态 TF 数据。

ld.add_action(joint_state_publisher_node)——发布关节状态信息的节点，常用于可视化机器人模型。

ld.add_action(imu_filter_node)——启动 imu_filter_madgwick 滤波器节点，用于姿态估计。

ld.add_action(robot_ekf)——启动 wheeltec_ekf.launch.py 文件，使用扩展卡尔曼滤波器用于机器人状态估计，如果在启动 wheeltec_cartographer 功能时，则使用 turn_on_wheeltec_robot/config/ekf_carto.yaml 参数文件，否则使用 turn_on_wheeltec_robot/config/ekf.yaml 参数文件

1.5 wheeltec_robot_node 节点说明

用户在启动机器人底层时都需要运行 base_serial.launch.py，这个文件启动的节点是 wheeltec_robot_node，对应的源码文件为 wheeltec_robot.cpp 和 Quaternion_Solution.cpp。

```
set(wheeltec_robot_node_SRCS
  src/wheeltec_robot.cpp
  src/Quaternion_Solution.cpp
)

add_executable(wheeltec_robot_node src/wheeltec_robot.cpp src/Quaternion_Solution.cpp)
ament_target_dependencies(wheeltec_robot_node tf2_ros tf2 tf2_geometry_msgs rclcpp std_msg

install(TARGETS
wheeltec_robot_node
```

图 1-5-1 wheeltec_robot_node 源码内容

Quaternion_Solution.cpp 是基于 STM32 板载 IMU 数据的轻量级四元数姿态解算模块，通过融合加速度计和陀螺仪数据实时计算出机器人在三维空间中的旋转状态。

wheeltec_robot.cpp 主要用于处理 STM32 的陀螺仪数据和发布电机控制数据，实现 WHEELTEC 机器人 ROS 与 STM32 控制器的通信。

1.6 wheeltec_robot.cpp 数据接收与处理

① wheeltec_robot.cpp 文件结构

wheeltec_robot.cpp 的头文件存放在 turn_on_wheeltec_robot/include/目录下。wheeltec_robot.cpp 代码中的函数结构如图所示。turn_on_robot 构造函数实现初始化功能，在启动 wheeltec_robot_node 节点时，代码先完成初始化功能，然后在主函数中实例化一个对象 Robot_Control 循环执行数据采集和发布话题等操作。

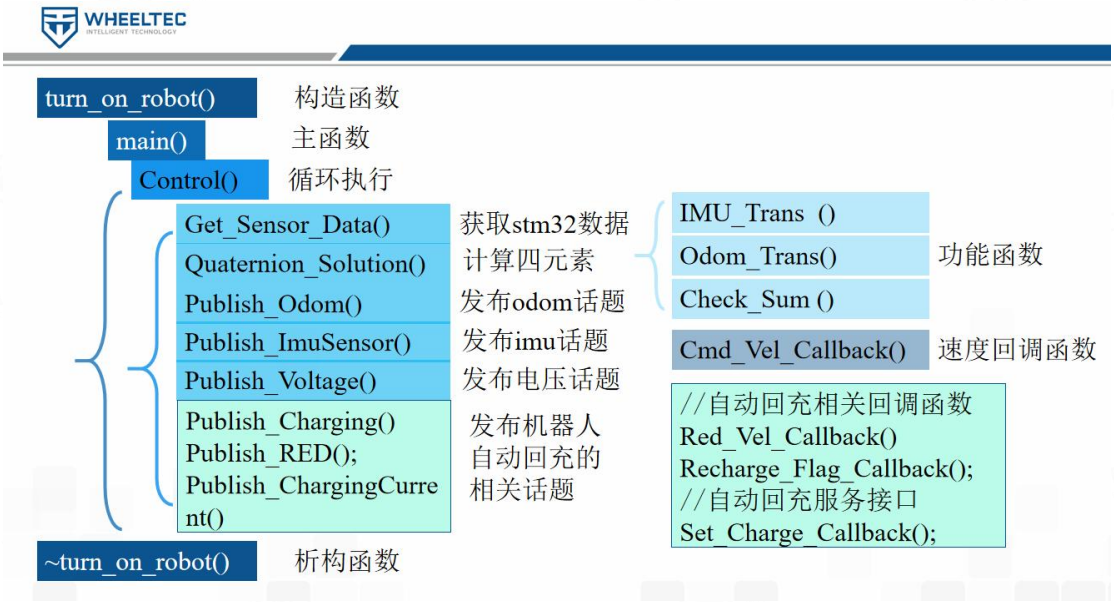

图 1-6-1 wheeltec_robot.cpp 代码结构

Control()函数里循环执行的函数内容和流程为：

使用 Get_Sensor_Data()获取 stm32 发送过来的数据，获取到数据后，将陀螺仪的原始参数输入 Quaternion_Solution()函数得到四元数，将解算出来的数据发布到里程计、IMU、和电池电压这三个话题。

根据判断机器人是否使用自动回充来确认是否发布机器人自动回充的相关话题。Publish_Charging()发布机器人是否在充电的话题，Publish_RED()发布机器人是否寻找到红外信号(充电桩)的话题，Publish_ChargingCurrent()发布机器人充电电流的话题。

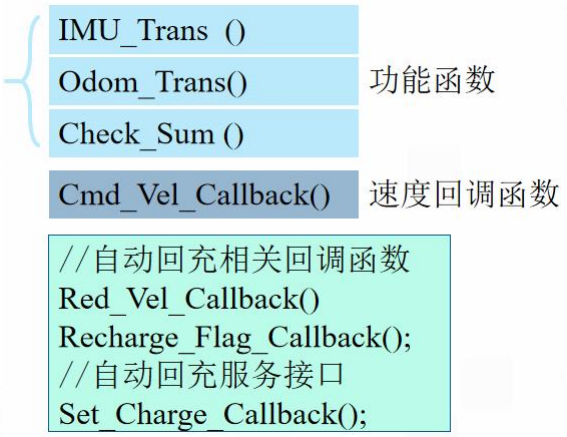

图 1-6-2 wheeltec_robot.cpp 代码结构

图示右上方这里有三个功能函数 IMU_Trans()、Odom_Trans()和 Check_Sum()，这三个功能函数用于数据的整合处理和校验，可以减少重复性代码。回调函数 Cmd_vel_callback()的作用是订阅机器人的目标速度话题，在这个回调函数中将机器人的目标速度通过串口发送给 stm32。

图示右下角绿色部分为自动回充相关的函数使用，其中 Red_Vel_Callback()函数订阅红外对接速度话题/red_vel，根据订阅的指令通过串口发指令设置红外对接速度；Recharge_Flag_Callback()函数订阅机器人是否回充标志位话题/robot_recharge_flag，用于告诉下位机速度命令是正常命令还是回充命令；Set_Charge_Callback()是自动回充服务接口，根据请求指令通过串口向 STM32 控制板发送充电控制指令，并返回是否设置成功。

在退出 wheeltec_robot_node 节点时，通过执行~turn_on_robot() 析构函数，向串口发送一帧目标速度为 0 的控制指令，以停止机器人运动，指令发送完成

后再关闭串口，确保机器人在节点退出前停止运动。

在 ROS 端启动机器人底层：

```
ros2 launch turn_on_wheeltec_robot turn_on_wheeltec_robot.launch.py
```

使用“ros2 node info /wheeltec_robot”指令查看节点运行状态，可以看到节点发布和订阅话题的情况，包括话题的消息类型等信息：

```
wheeltec@wheeltec:~$ ros2 node info /wheeltec_robot
/wheeltec_robot
  Subscribers:
    /cmd_vel: geometry_msgs/msg/Twist
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /red_vel: geometry_msgs/msg/Twist
    /robot_recharge_flag: std_msgs/msg/Int8
  Publishers:
    /PowerVoltage: std_msgs/msg/Float32
    /imu/data_board: sensor_msgs/msg/Imu
    /odom: nav_msgs/msg/Odometry
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /robot_charging_current: std_msgs/msg/Float32
    /robot_charging_flag: std_msgs/msg/Bool
    /robot_red_flag: std_msgs/msg/UInt8
    /rosout: rcl_interfaces/msg/Log
  Service Servers:
    /set_charge: turtlesim/srv/Spawn
    /wheeltec_robot/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /wheeltec_robot/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /wheeltec_robot/get_parameters: rcl_interfaces/srv/GetParameters
    /wheeltec_robot/list_parameters: rcl_interfaces/srv/ListParameters
    /wheeltec_robot/set_parameters: rcl_interfaces/srv/SetParameters
    /wheeltec_robot/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
  Service Clients:

  Action Servers:

  Action Clients:
```

图 1-6-3 wheeltec_robot 节点运行情况

② Control()函数内容

进入 Control()函数，首先记录进入这个函数的当前时间，rclcpp::Node::now()记录当前时间，为后续计算速度积分提供参考。记录时间后进入主循环，只要用户没有使用 Ctrl+C 停止程序，就进入这个持续运行的循环。在主循环中再次获取当前时间_Now，并计算自上次循环以来的时间间隔 Sampling_Time，用于做速度积分（速度 × 时间 = 位移）。

调用串口接口函数 Get_Sensor_Data_New()，从下位机（STM32）读取 IMU、编码器等数据，并验证校验位。如果数据成功解析，就进入数据处理逻辑。

计算 Robot_Vel：使用 odom_*_scale 参数对速度做比例修正，考虑编码器和陀螺仪可能存在的误差。X, Y, Z 分别是机器人在平面上的线速度和角速度（Z 表示绕垂直方向的旋转）。

计算 Robot_Pos：使用速度积分法更新机器人当前位姿（Position），X/Y 方向通过速度×采样时间算出，再由当前角度修正方向，Z 方向是角度累加。Robot_Vel 和 Robot_Pos 的数据将被发布为里程计的话题数据。

Quaternion_Solution()通过 MPU6050 提供的 IMU 原始数据（角速度和加

速度），计算三轴方向姿态角，处理后的数据将被发布为板载 IMU 的话题数据。在发布完话题数据后，更新_Last_Time，用于下一次循环中重新计算时间间隔。

rclcpp::spin_some(this->get_node_base_interface())是 ROS2 非阻塞行回调函数处理，检查当前节点是否有可以立即处理的回调，如果有，就立刻执行。

```
void turn_on_robot::Control()
{
  //_Last_Time = ros::Time::now();
  _Last_Time = rclcpp::Node::now();
  while(rclcpp::ok())
  {
    try
    {
    //_Now = ros::Time::now();
    _Now = rclcpp::Node::now();
    Sampling_Time = (_Now - _Last_Time).seconds();  //Retrieves time interval, which is used to integrate velocity to obtain displacement (mileage)
                                                   //获取时间间隔，用于积分速度获得位移(里程)
    if (true == Get_Sensor_Data_New()) //The serial port reads and verifies the data sent by the lower computer, and then the data is converted to internati
                                       //通过串口读取并校验下位机发送过来的数据，然后数据转换为国际单位
    {
      //Odometer error correction //里程计误差修正
      Robot_Vel.X = Robot_Vel.X*odom_x_scale;
      Robot_Vel.Y = Robot_Vel.Y*odom_y_scale;
      if( Robot_Vel.Z>=0 )  Robot_Vel.Z = Robot_Vel.Z*odom_z_scale_positive;
      else  Robot_Vel.Z = Robot_Vel.Z*odom_z_scale_negative;
      Robot_Pos.X+=(Robot_Vel.X * cos(Robot_Pos.Z) - Robot_Vel.Y * sin(Robot_Pos.Z)) * Sampling_Time; //Calculate the displacement in the X direction, unit:
      Robot_Pos.Y+=(Robot_Vel.X * sin(Robot_Pos.Z) + Robot_Vel.Y * cos(Robot_Pos.Z)) * Sampling_Time; //Calculate the displacement in the Y direction, unit:
      Robot_Pos.Z+=Robot_Vel.Z * Sampling_Time; //The angular displacement about the Z axis, in rad //绕Z轴的角位移，单位：rad
      //Calculate the three-axis attitude from the IMU with the angular velocity around the three-axis and the three-axis acceleration
      //通过IMU绕三轴角速度与三轴加速度计算三轴姿态
      Quaternion_Solution(Mpu6050.angular_velocity.x, Mpu6050.angular_velocity.y, Mpu6050.angular_velocity.z,\
                          Mpu6050.linear_acceleration.x, Mpu6050.linear_acceleration.y, Mpu6050.linear_acceleration.z);

      Publish_Odom();      //Pub the speedometer topic //发布里程计话题
      Publish_ImuSensor(); //Pub the IMU topic //发布IMU话题
      Publish_Voltage();   //Pub the topic of power supply voltage //发布电源电压话题

      _Last_Time = _Now; //Record the time and use it to calculate the time interval //记录时间，用于计算时间间隔

    }
    //自动回充数据话题
    if(check_AutoCharge_data)
    {
      Publish_Charging();  //Pub a topic about whether the robot is charging //发布机器人是否在充电的话题
      Publish_RED();       //Pub the topic whether the robot finds the infrared signal (charging station) //发布机器人是否寻找到红外信号(充电桩)的话题
      Publish_ChargingCurrent(); //Pub the charging current topic //发布充电电流话题
      check_AutoCharge_data = false;
    }
    rclcpp::spin_some(this->get_node_base_interface());   //The loop waits for the callback function //循环等待回调函数
    }
    catch (const rclcpp::exceptions::RCLError & e )
  {
  RCLCPP_ERROR(this->get_logger(),"unexpectedly failed whith %s",e.what());
  }
}
```

图 1-6-4 Control()函数内容

③ ROS 端读取 STM32 发送的数据

Get_Sensor_Data_New()函数的作用是通过串口读取并逐帧校验下位机(STM32)发送过来的数据，校验后解析数据转换为国际单位，数据转化为机器人运动里程计信息、IMU 信息和电池电压数据。

```
bool turn_on_robot::Get_Sensor_Data_New()
{
  short transition_16=0; //Intermediate variable //中间变量
  //uint8_t i=0;
  uint8_t check=0,check2=0, error=1,error2=1,Receive_Data_Pr[1]; //Temporary variable to save the data of the lower machine //临时变量，保存下位机数据
  static int count,count2; //Static variable for counting //静态变量，用于计数
  Stm32_Serial.read(Receive_Data_Pr,sizeof(Receive_Data_Pr)); //Read the data sent by the lower computer through the serial port //通过串口读取下位机发送过来的数据

  Receive_Data.rx[count] = Receive_Data_Pr[0]; //Fill the array with serial data //串口数据填入数组
  Receive_AutoCharge_Data.rx[count2] = Receive_Data_Pr[0];

  Receive_Data.Frame_Header = Receive_Data.rx[0]; //The first part of the data is the frame header 0X7B //数据的第一位是帧头0X7B
  Receive_Data.Frame_Tail = Receive_Data.rx[23];  //The last bit of data is frame tail 0X7D //数据的最后一位是帧尾0X7D

  //接收到自动回充数据的帧头、上一个数据是24字节的帧尾，表明自动回充数据开始到来
  if((Receive_Data_Pr[0] == AutoCharge_HEADER )||count2>0)
    count2++;
  else
    count2=0;

  if(Receive_Data_Pr[0] == FRAME_HEADER || count>0) //Ensure that the first data in the array is FRAME_HEADER //确保数组第一个数据为FRAME_HEADER
    count++;
  else
    count=0;

  //自动回充数据处理
  if(count2 == AutoCharge_DATA_SIZE)
  { }

  if(count == 24) //Verify the length of the packet //验证数据包的长度
  {
    count=0; //Prepare for the serial port data to be refill into the array //为串口数据重新填入数组做准备
    if(Receive_Data.Frame_Tail == FRAME_TAIL) //Verify the frame tail of the packet //验证数据包的帧尾
    {
      check=Check_Sum(22,READ_DATA_CHECK);  //BCC check passes or two packets are interlaced //BCC校验通过或者两组数据包交错

      if(check == Receive_Data.rx[22])
      {
        error=0;  //XOR bit check successful //异或位校验成功
      }
      if(error == 0)
      { }
    }
  }
  return false;
}
```

图 1-6-5 Get_Sensor_Data_New()函数内容

进入函数内部，首先是初始化一些接收数据时用到的中间变量，以及一个临时存放数据的数组 Receive_Data_Pr[]。

接着用 Stm32_Serial.read（）函数，将串口数据读取到 Receive_Data_Pr[]这个数组里面，再存放至 Receive_Data.rx[]和 Receive_AutoCharge_Data.rx[]数组中，并提取帧头帧尾，如果是普通数据帧（如运动数据）帧头，或处于接收中，则累加 count 和 count2。完整的一帧数据为 24 字节，当完成一个完整数据帧的接收，准备处理数据。

处理数据过程：先确认数据包长度是否为 24，并验证数据包的帧尾、校验和。若校验通过，开始解析各项数据：

```
if(error == 0)
{
  Receive_Data.Flag_Stop=Receive_Data.rx[1]; //set aside //预留位
  Robot_Vel.X = Odom_Trans(Receive_Data.rx[2],Receive_Data.rx[3]); //Get the speed of the moving chassis in the X direction //获取运动底盘X方向速度

  Robot_Vel.Y = Odom_Trans(Receive_Data.rx[4],Receive_Data.rx[5]); //Get the speed of the moving chassis in the Y direction, The Y speed is only va
                                                                   //获取运动底盘Y方向速度，Y速度仅在全向移动机器人底盘有效
  Robot_Vel.Z = Odom_Trans(Receive_Data.rx[6],Receive_Data.rx[7]); //Get the speed of the moving chassis in the Z direction //获取运动底盘Z方向速度

  //MPU6050 stands for IMU only and does not refer to a specific model. It can be either MPU6050 or MPU9250
  //Mpu6050仅代表IMU，不指代特定型号，既可以是MPU6050也可以是MPU9250
  Mpu6050_Data.accele_x_data = IMU_Trans(Receive_Data.rx[8],Receive_Data.rx[9]);   //Get the X-axis acceleration of the IMU     //获取IMU的X轴加速度
  Mpu6050_Data.accele_y_data = IMU_Trans(Receive_Data.rx[10],Receive_Data.rx[11]); //Get the Y-axis acceleration of the IMU     //获取IMU的Y轴加速度
  Mpu6050_Data.accele_z_data = IMU_Trans(Receive_Data.rx[12],Receive_Data.rx[13]); //Get the Z-axis acceleration of the IMU     //获取IMU的Z轴加速度
  Mpu6050_Data.gyros_x_data = IMU_Trans(Receive_Data.rx[14],Receive_Data.rx[15]);  //Get the X-axis angular velocity of the IMU //获取IMU的X轴角速度
  Mpu6050_Data.gyros_y_data = IMU_Trans(Receive_Data.rx[16],Receive_Data.rx[17]);  //Get the Y-axis angular velocity of the IMU //获取IMU的Y轴角速度
  Mpu6050_Data.gyros_z_data = IMU_Trans(Receive_Data.rx[18],Receive_Data.rx[19]);  //Get the Z-axis angular velocity of the IMU //获取IMU的Z轴角速度
  //Linear acceleration unit conversion is related to the range of IMU initialization of STM32, where the range is ±2g=19.6m/s^2
  //线性加速度单位转化，和STM32的IMU初始化的时候的量程有关,这里量程±2g=19.6m/s^2
  Mpu6050.linear_acceleration.x = Mpu6050_Data.accele_x_data / ACCEl_RATIO;
  Mpu6050.linear_acceleration.y = Mpu6050_Data.accele_y_data / ACCEl_RATIO;
  Mpu6050.linear_acceleration.z = Mpu6050_Data.accele_z_data / ACCEl_RATIO;
  //The gyroscope unit conversion is related to the range of STM32's IMU when initialized. Here, the range of IMU's gyroscope is ±500°/s
  //Because the robot generally has a slow Z-axis speed, reducing the range can improve the accuracy
  //陀螺仪单位转化，和STM32的IMU初始化的时候的量程有关，这里IMU的陀螺仪的量程是±500°/s
  //因为机器人一般Z轴速度不快，降低量程可以提高精度
  Mpu6050.angular_velocity.x =  Mpu6050_Data.gyros_x_data * GYROSCOPE_RATIO;
  Mpu6050.angular_velocity.y =  Mpu6050_Data.gyros_y_data * GYROSCOPE_RATIO;
  Mpu6050.angular_velocity.z =  Mpu6050_Data.gyros_z_data * GYROSCOPE_RATIO;

  //Get the battery voltage
  //获取电池电压
  transition_16 = 0;
  transition_16 |=  Receive_Data.rx[20]<<8;
  transition_16 |=  Receive_Data.rx[21];
  Power_voltage = transition_16/1000+(transition_16 % 1000)*0.001; //Unit conversion millivolt(mv)->volt(v) //单位转换毫伏(mv)->伏(v)

  return true;
}
```

图 1-6-6 Get_Sensor_Data_New()函数内容

在处理数据中，先解析 Robot_Vel 速度信息，这里将读取到的 STM32 的串口数据传入 odom_trans()函数，将数据合并并进行单位转换转为机器人运动底盘的 xyz 方向速度。

接着读取陀螺仪的原始数据存放到 Mpu6050_Data 结构体中，imu_trans()函数作用是完成数据合并，将两个 8 位的数据合成一个 16 位的数据。Mpu6050_Data 的数据还是未进行单位处理过的原始数据，需要将 IMU 数据转为实际物理单位。

```
//ACCEl_RATIO: 加速度计分辨率因子，数值为 1671.84
//GYROSCOPE_RATIO: 陀螺仪分辨率，值为 0.00026644
```

```
/**************************************
Date: January 28, 2021
Function: Data conversion function
功能: 数据转换函数
**************************************/
short turn_on_robot::IMU_Trans(uint8_t Data_High,uint8_t Data_Low)
{
  short transition_16;
  transition_16 = 0;
  transition_16 |=  Data_High<<8;
  transition_16 |=  Data_Low;
  return transition_16;
}
float turn_on_robot::Odom_Trans(uint8_t Data_High,uint8_t Data_Low)
{
  float data_return;
  short transition_16;
  transition_16 = 0;
  transition_16 |=  Data_High<<8;  //Get the high 8 bits of data   //获取数据的高8位
  transition_16 |=  Data_Low;      //Get the lowest 8 bits of data //获取数据的低8位
  data_return   =  (transition_16 / 1000)+(transition_16 % 1000)*0.001; // The speed unit is changed from mm/s to m/s //速度单位从mm/s转换为m/s
  return data_return;
}
```

图 1-6-7 _trans()转换函数内容

最后处理的数据是电池电压，同样的在发送前做了一个数据拆分和单位转换，这里接收后需要做一个数据还原。直到这一步，Get_Sensor_Data_New()函数才会返回 true 值。

④ ROS 端向 STM32 发送数据

Cmd_vel_callback()回调函数的作用是订阅机器人的目标速度话题/cmd_vel，将其转换为串口协议格式，通过串口发送给下位机控制器 STM32。函数内容如下所示。

```
void turn_on_robot::Cmd_Vel_Callback(const geometry_msgs::msg::Twist::SharedPtr twist_aux)
{
  short  transition;  //intermediate variable //中间变量

  Send_Data.tx[0]=FRAME_HEADER; //frame head 0x7B //帧头0X7B
  Send_Data.tx[1] = AutoRecharge; //set aside //预留位
  Send_Data.tx[2] = 0; //set aside //预留位

  //The target velocity of the X-axis of the robot
  //机器人x轴的目标线速度
  transition=0;
  transition = twist_aux->linear.x*1000; //将浮点数放大一千倍，简化传输
  Send_Data.tx[4] = transition;     //取数据的低8位
  Send_Data.tx[3] = transition>>8;  //取数据的高8位

  //The target velocity of the Y-axis of the robot
  //机器人y轴的目标线速度
  transition=0;
  transition = twist_aux->linear.y*1000;
  Send_Data.tx[6] = transition;
  Send_Data.tx[5] = transition>>8;

  //The target angular velocity of the robot's Z axis
  //机器人z轴的目标角速度
  transition=0;
  transition = twist_aux->angular.z*1000;
  Send_Data.tx[8] = transition;
  Send_Data.tx[7] = transition>>8;

  Send_Data.tx[9]=Check_Sum(9,SEND_DATA_CHECK); //For the BCC check bits, see the Check_Sum function //BCC校验位，规则参见Check_Sum函数
  Send_Data.tx[10]=FRAME_TAIL; //frame tail 0x7D //帧尾0X7D
  try
  {
    Stm32_Serial.write(Send_Data.tx,sizeof (Send_Data.tx)); //Sends data to the downloader via serial port //通过串口向下位机发送数据
  }
  catch (serial::IOException& e)
  {
    RCLCPP_ERROR(this->get_logger(),("Unable to send data through serial port")); //If sending data fails, an error message is printed
  }
}
```

图 1-6-8 Cmd_vel_callback()回调函数内容

twist_aux 是一个智能指针，包含机器人目标的线速度和角速度数据。函数中定义一个中间变量 transition，用于临时保存速度值的整数形式。在发送数据前，需要构造发送数据帧的帧头信息。Send_Data.tx[] 是用于 ROS 向下位机（STM32）

发送的字节缓存数组，长度为 11 字节。

以 X 轴线速度为例，twist_aux->linear.x 是目标线速度（单位：m/s），为了便于在 STM32 中处理，该浮点数会先放大 1000 倍（如 0.123m/s→123m/s），并转换为 short 类型整数。随后将该整数拆分为高 8 位和低 8 位，分别存入 tx[3]和 tx[4]中。以相同方式处理 Y 轴线速度与 Z 轴角速度后，使用 Check_Sum()函数计算前 9 个字节的校验值，生成一个校验值以确保数据完整性，并存入 tx[9]。最后添加帧尾 tx[10](0x7D)，用于标识一帧数据的结束。构造完整数据帧后，调用 Stm32_Serial.write() 函数，通过串口将该帧数据发送给下位机（STM32）。