

第八届

全国大学生集成电路创新创业大赛

报告类型*: 设计报告

参赛杯赛*: 飞腾杯

作品名称*: 基于飞腾派的智能建图小车

队伍编号*: CICC7101

团队名称*: 不求名次只求经历

目录

目录	2
1.系统需求分析.....	3
1.1 功能需求:	3
1.2 性能需求:	3
2.系统架构设计.....	3
2.1 硬件架构:	3
2.2 软件架构:	3
2.2.1 总架构.....	4
2.2.2 从核架构.....	5
2.2.3 主核架构.....	6
2.2.4 详细主从架构图.....	7
3.系统详细设计:	8
3.1 硬件设计:	8
3.2 软件设计:	12
3.2.1 rpmsg.h 模块	12
3.2.2 Mainwindow.cpp 模块.....	14
3.2.3 car_gui.cpp 模块	23
3.2.4 rpmsg-echo_os.c 模块	27
3.2.5 car.c 模块	31
3.2.6 car_lidar_gui.py 模块.....	35
4.系统测试与分析.....	37
4.1 需求功能实现验证.....	37
4.1.1 项目启动.....	37
4.1.2 小车控制功能.....	38
4.1.3 前紧急制动功能.....	41
4.1.4 建图功能.....	42
4.2 性能达标验证.....	44

1.系统需求分析

1.1 功能需求:

建图小车需要具备多项基本功能，包括环境感知、地图构建、数据传输与存储等。环境感知功能使其能够感知周围环境的情况，例如检测障碍物。地图构建功能允许小车在运动过程中利用传感器数据建立地图。数据传输与存储功能负责将感知到的环境信息和建立的地图数据传输至主控制系统，并在需要时进行存储。

除了基本功能外，建图小车还需要支持一系列高级功能以提升其应用价值和安全性。其中包括远程控制功能，使用户可以通过远程设备对小车进行操作和监控。障碍物检测功能能够及时发现并识别路径上的障碍物，并采取相应措施进行处理。紧急制动功能则在遇到紧急情况时，如发现突发障碍或检测到碰撞威胁时，能够迅速采取制动措施以确保安全。这些高级功能使建图小车能够更加智能化、安全化地执行任务，满足不同应用场景的需求。

1.2 性能需求:

建图小车应具备较高的定位精度和稳定性，其数据处理能力和实时响应速度也应满足实际需求。定位精度对于建立准确的地图和实现精准导航至关重要，因此小车需要准确计算运动一定时间后相距起始点的距离和朝向。同时，小车还需要能够快速处理大量的感知数据(如激光雷达扫描数据等)，并实时地进行环境分析和地图构建。同时，实时响应速度也是保障小车安全性和操作效率的重要因素，它需要在各种复杂环境和应急情况下，能够迅速做出相应的控制和决策。

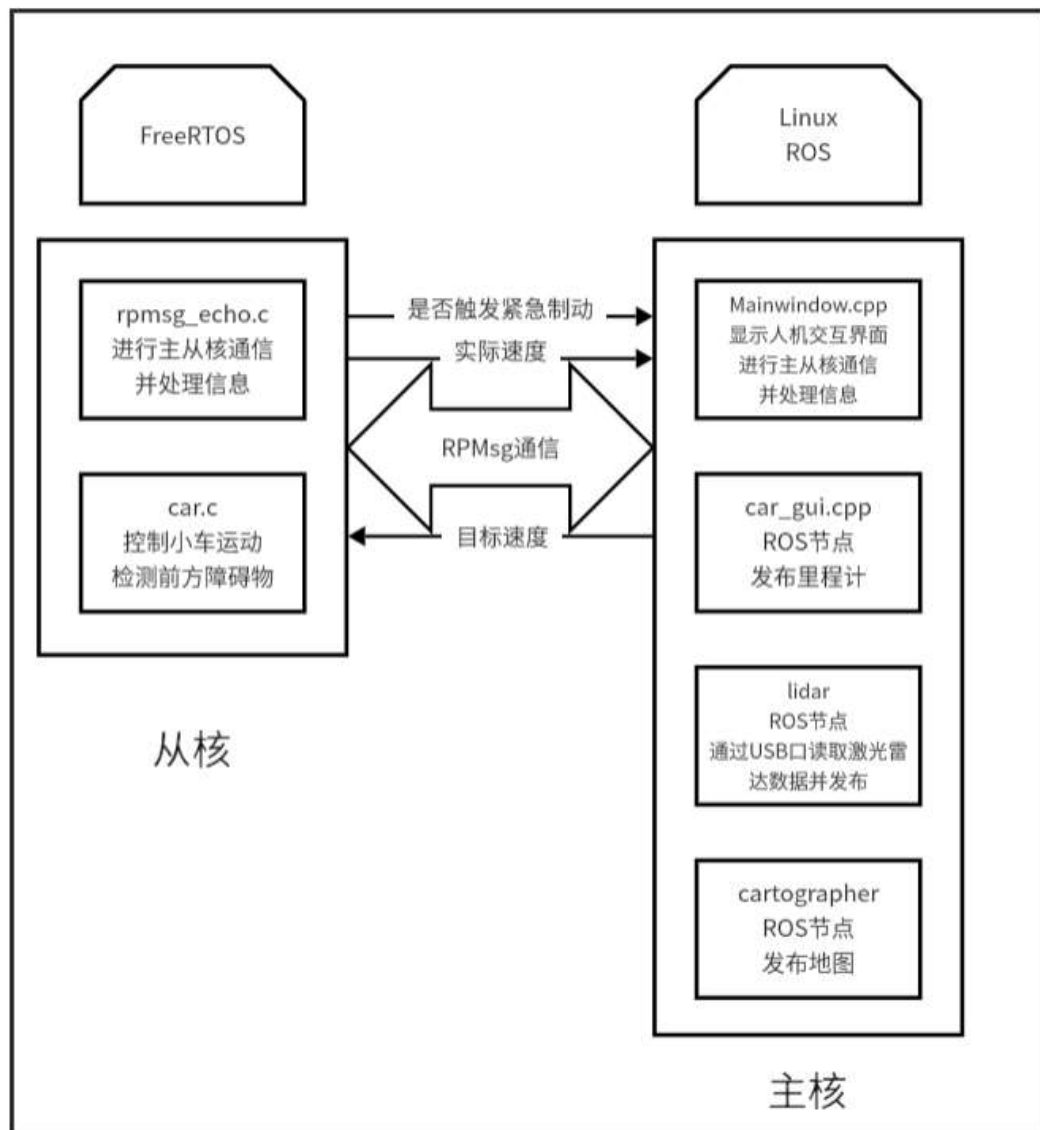
2.系统架构设计

2.1 硬件架构:

建图小车的硬件主要包括飞腾派主板、激光雷达、前避障传感器（红外传感器）、阿克曼后驱底盘（电机搭载霍尔传感器）、电机驱动模块（稳定输出电压和电流）、电源等组件。这些硬件组成了小车的核心系统，配合软件，实现了对小车的控制，环境的感知、数据的采集和处理。

2.2 软件架构:

2.2.1 总架构



系统采用了主从核设计，实现了高效的任务并行运行与协作。

从核运行在基于 FreeRTOS 的裸机高实时性系统上，专门负责控制小车的运动速度和方向，并读取前避障传感器的状态来判断是否需要触发紧急制动。同时利用 RPMsg 通信，将实时的实际速度信息反馈给主核并接收主核发送的目标速度进行响应并控制。

主核运行在基于 Linux 的 Ubuntu 22.04 系统上，通过 ROS 框架进行对多个并行节点管理和调度。

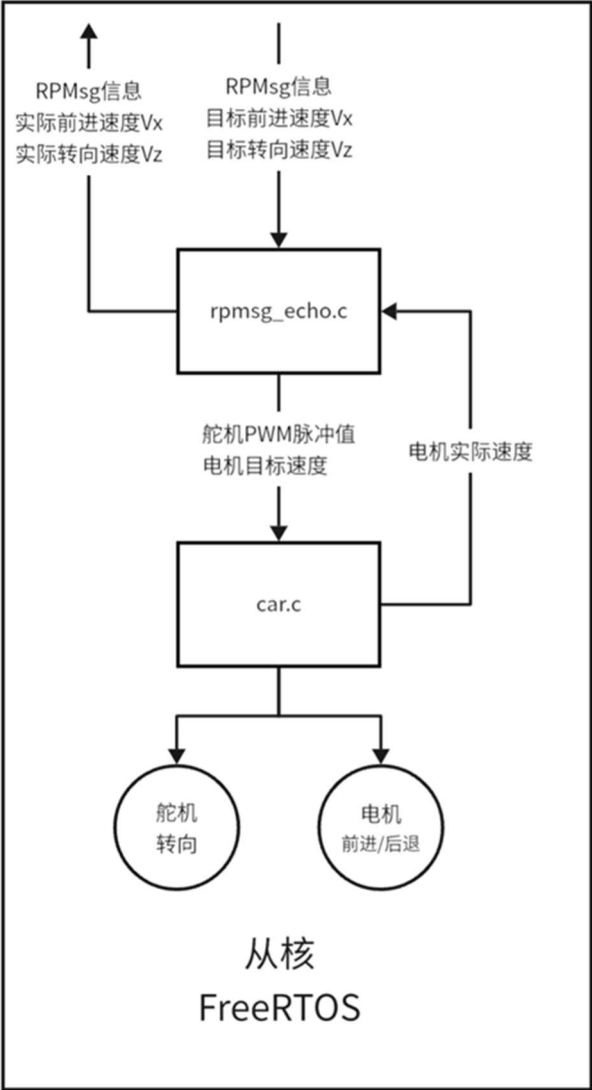
首先，主核承担了生成和处理人机交互界面的任务，为用户提供友好的操作界面，使其能够方便地监控和控制小车的运动。会它接收从核发送的当前实际速度信息，并在人机交互界面上进行显示，同时接收从核发送的是否触发紧急制动

来判断是否需要提示用户当前的小车状态。此外，主核还负责根据用户在人机交互界面的控制，下发期望的目标速度。

同时，主核利用激光雷达进行环境感知，结合收到的实际速度计算得到的里程计，实时地构建地图。

这种分工合作的设计架构使得系统能够高效地完成各项任务，具备了较强的实用性和扩展性。

2.2.2 从核架构



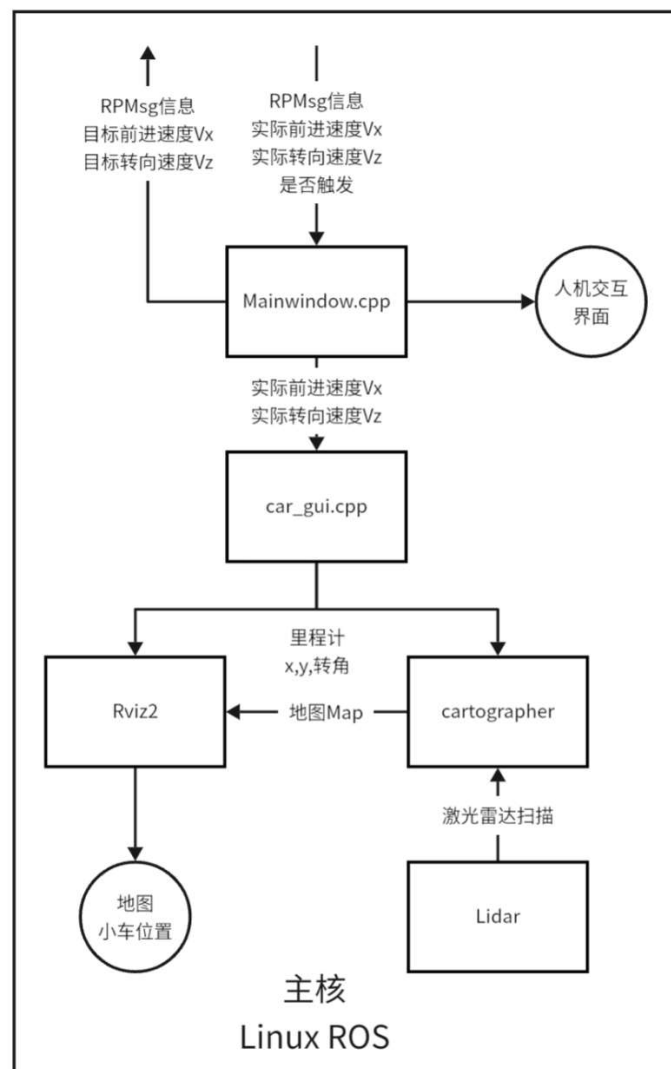
从核运行在基于 FreeRTOS 的高实时性系统上，核心程序为 rpmmsg_echo.c。它的主要任务根据来自主核下发的目标前进速度 V_x 和目标转向速度 V_z ，通过计算得到舵机的 PWM 脉冲数和电机目标速度。

car.c 根据舵机的 PWM 脉冲数和电机的目标速度，控制舵机的转向和调节电机的速度，从而实现小车的运动控制。

同时，car.c 还承担着监测电机实际速度的任务。它通过测量电机所配备的霍尔传感器一定时间内输出的脉冲数，得到电机的转速后计算得到电机的实际速度。之后 rpmsg_echo.c 便根据电机的实时速度计算得到实际的前进速度 V_x 和转向速度 V_z ，以便主核系统及时了解小车的运动状态。

这种双向通信和实时反馈机制确保了小车在运行过程中能够及时调整速度和方向，以应对各种情况，提高了系统的稳定性和可靠性。

2.2.3 主核架构



主核运行在基于 Linux 的 Ubuntu 22.04 系统上，并通过 ROS 框架进行管理和控制，核心程序为 `mainwindow.cpp`。这个程序负责构建人机交互界面，提供给用户友好的交互体验。通过主界面，用户可以遥控、查看小车运动状态等。

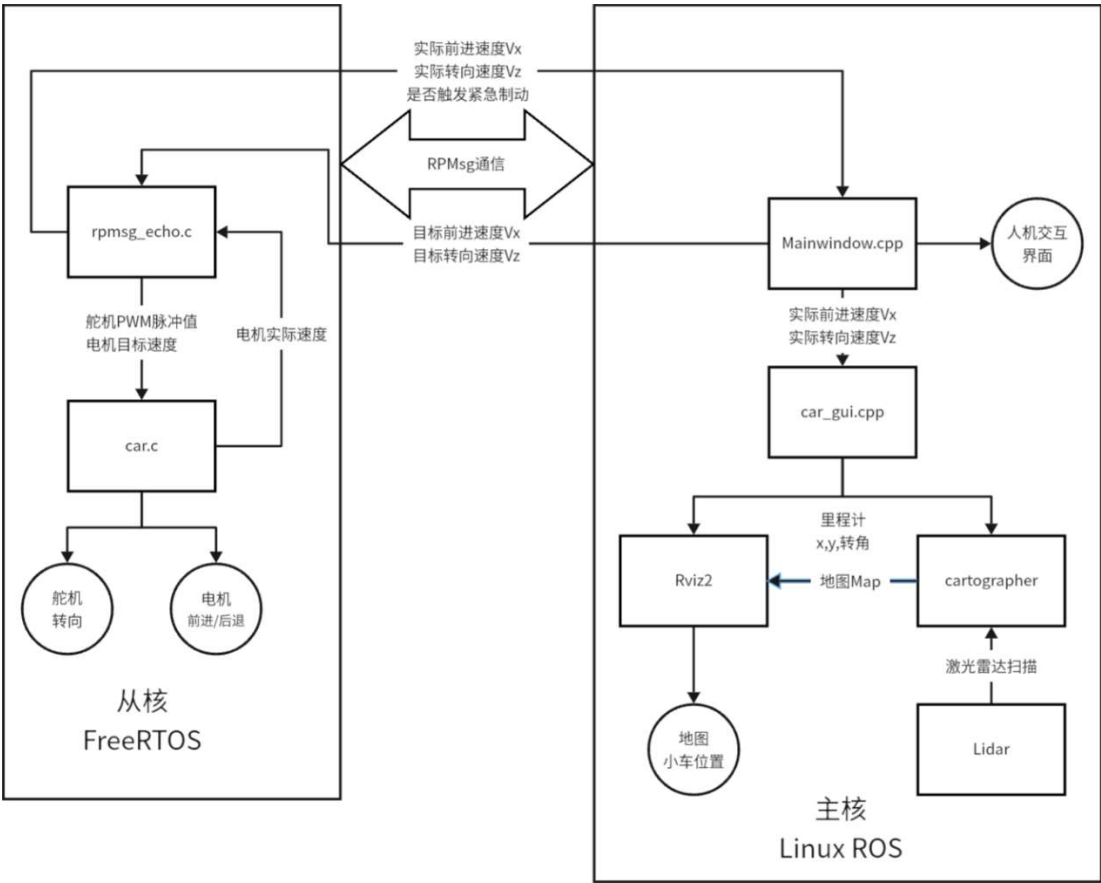
`mainwindow.cpp` 将来自从核的当前前进速度 V_x 和转向速度 V_z 传递给 `car_gui.cpp`。`car_gui.cpp` 接收到这些速度信息后，根据车辆的运动模型计算出里程计 x, y 和转角。这些里程计数据随后分别传递给 `Cartographer` 和 `Rviz`。

在 `Cartographer` 中，接收到里程计以及来自激光雷达 `lidar` 扫描的数据后，开始生成地图。这个地图随后传递给 `Rviz` 进行进一步处理和显示。

`Rviz` 结合了里程计的信息，将实时地图构建出来，并将小车的位置与地图进行实时匹配，使用户可以清晰地看到小车在地图上的实时位置。

这样的系统架构和数据流设计，使得用户可以通过用户界面直观地控制小车，并实时了解小车的位置和周围环境。

2.2.4 详细主从架构图

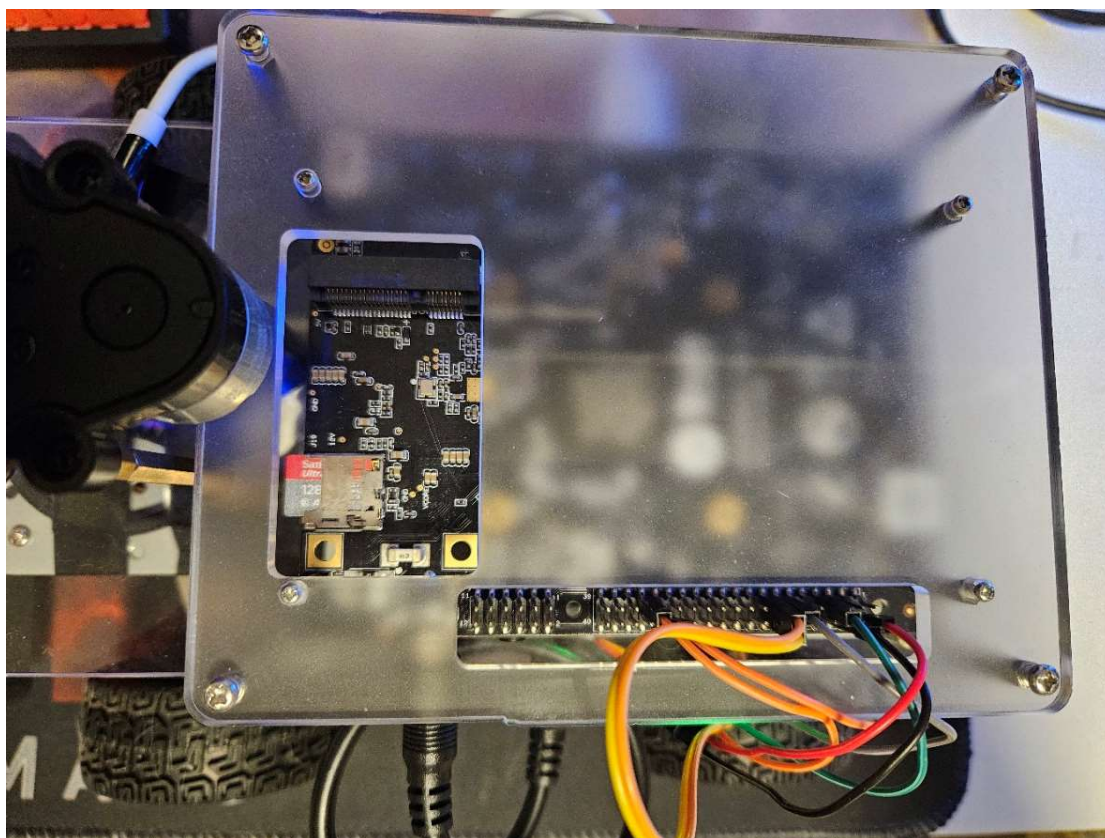


3.系统详细设计：

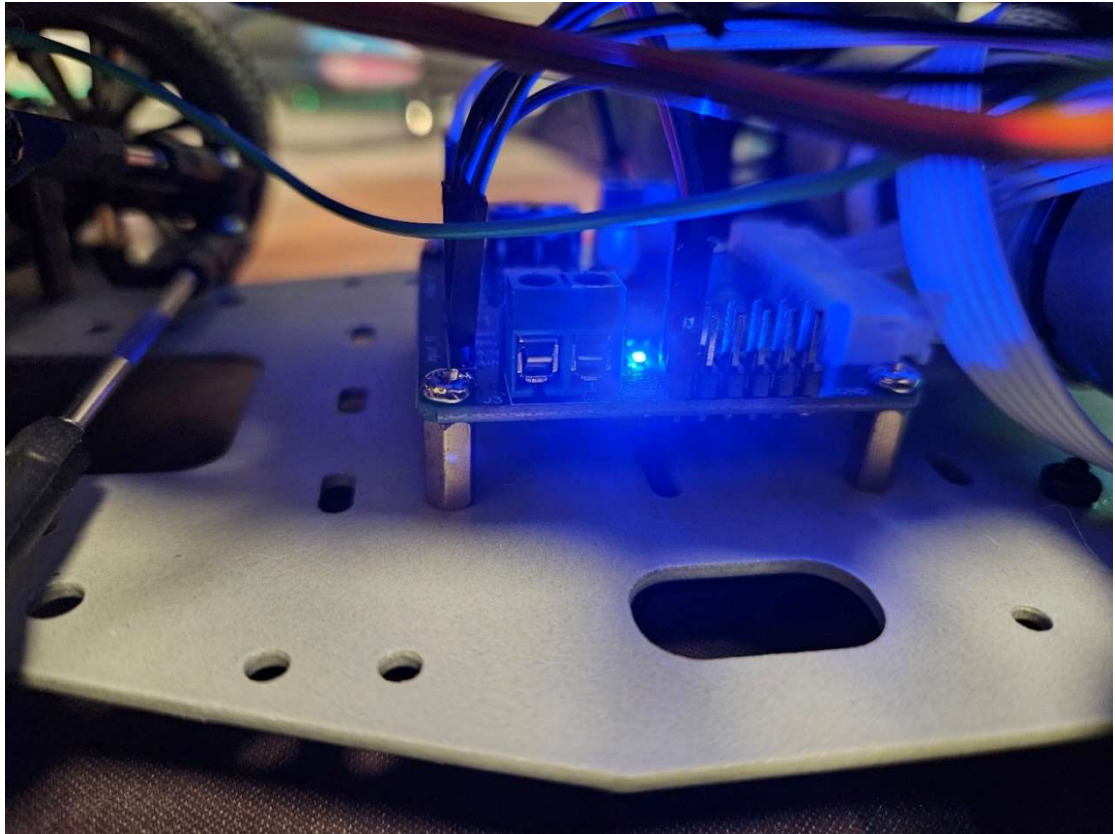
3.1 硬件设计：

建图小车的硬件设计包括多个关键组件。

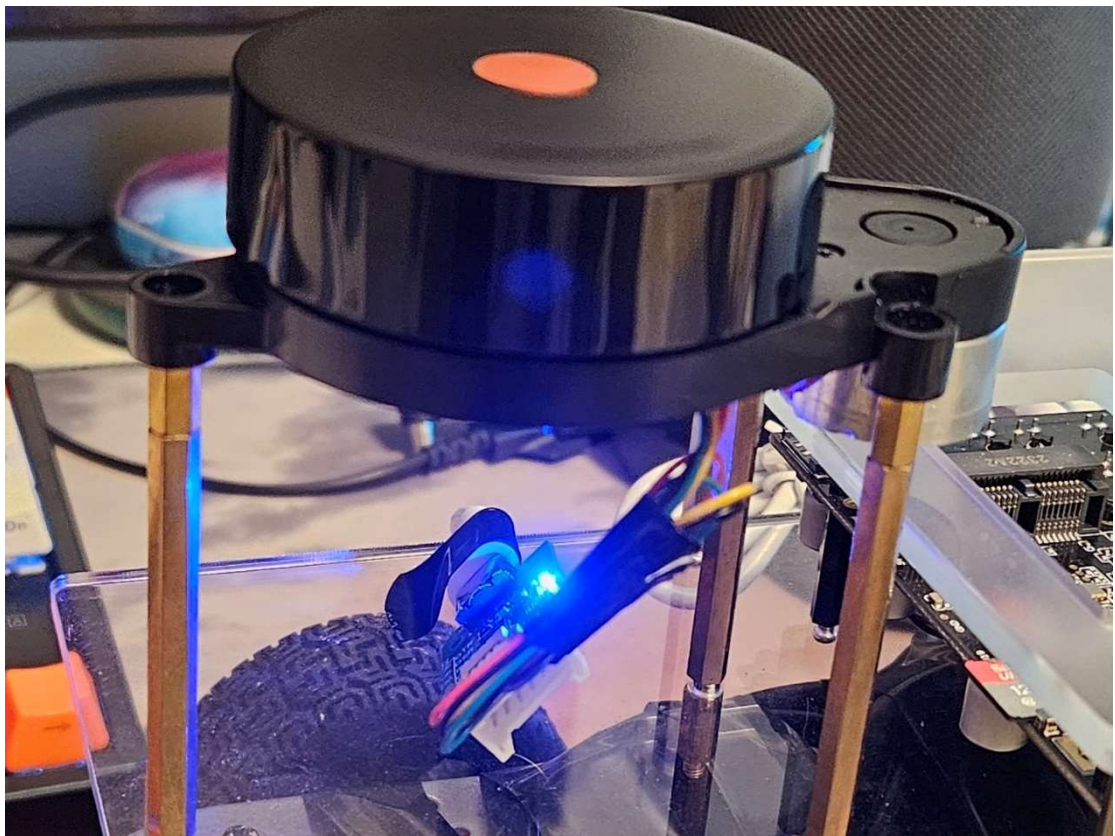
飞腾派主板作为核心控制器，承担着处理传感器数据，对舵机及电机进行控制以及显示人机交互界面的任务。它与激光雷达、前避障传感器、阿克曼后驱底盘的舵机和电机以及电源连接。



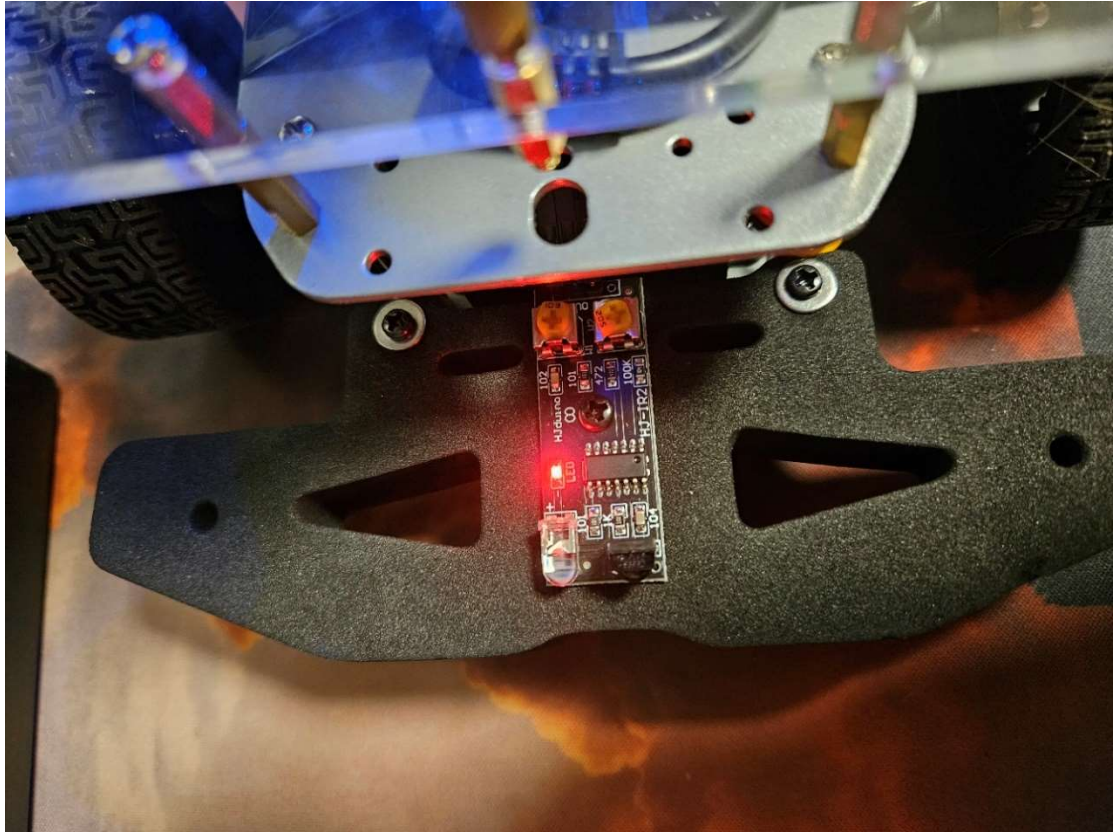
由于飞腾派接口的输出电流较小且不稳定，为了稳定输出电流和电压，添加了电机驱动模块进行稳压稳流。



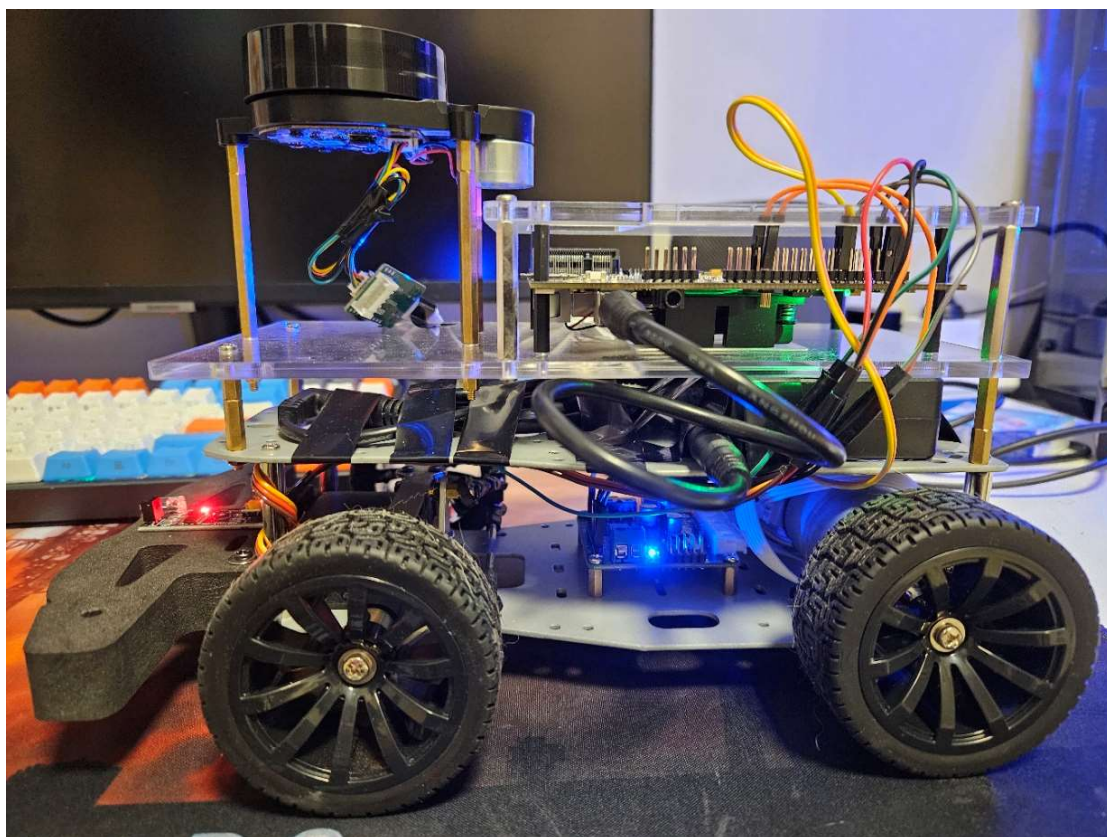
激光雷达用于扫描周围环境以供建图。



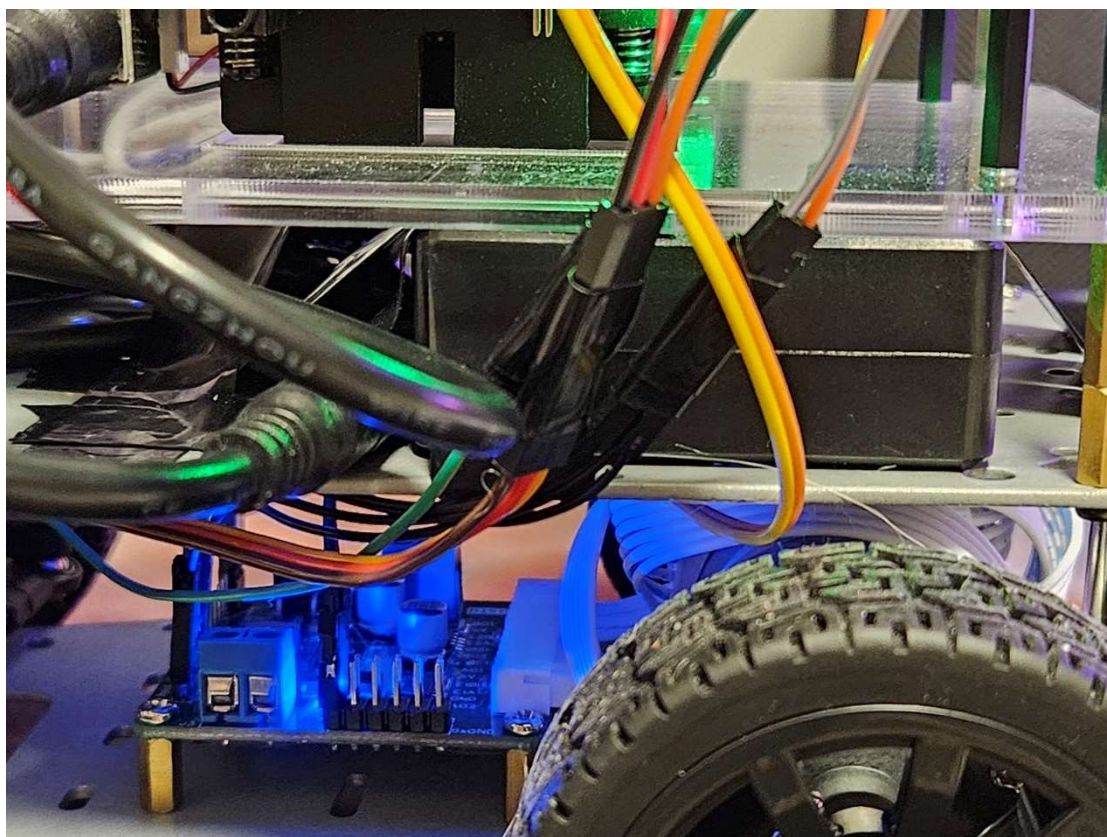
前避障传感器则位于车辆前方，使用红外传感器检测前方障碍物，以避免碰撞。



阿克曼后驱底盘包括电机，舵机以及支架等设备，是小车移动与支撑的核心部件。其电机搭载了霍尔编码器，实现对电机状态的监控。



最后，电源模块则为整个系统提供稳定的电源供应，确保各组件正常运行。



这样的硬件设计旨在实现建图小车的高效运行和精准感知周围环境，为地图构建提供可靠的支持。

3.2 软件设计:

主核从核都采用了模块化设计，方便后续维护和升级。

3.2.1 rpmsg.h 模块

rpmsg.h 代码分析:

主要功能介绍:

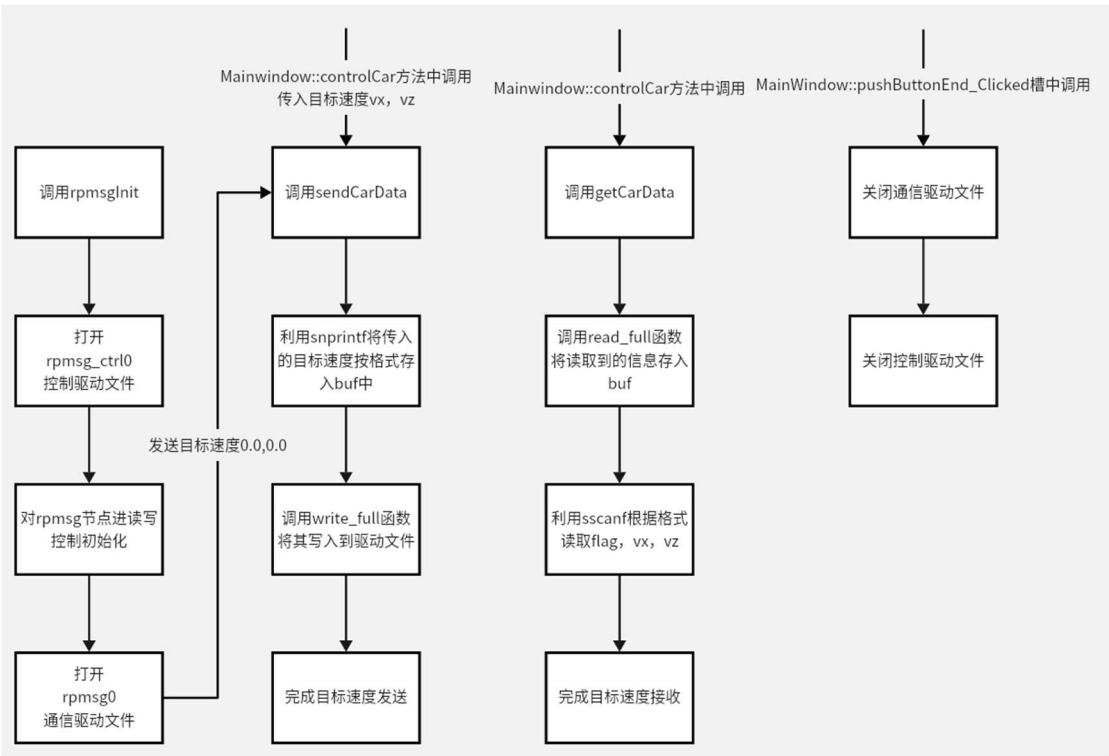
rpmsg.h 在 Mainwindow.h 中被包括，在 Mainwindow.cpp 中调用了其提供的函数实现了主从核通信。

主要流程:

要实现主从核通信，首先需要调用 rpmsgInit，打开 rpmsg_ctrl0 驱动文件，之后对 rpmsg 节点进读写控制初始化，然后打开 rpmsg0 驱动，最后调用 rpmsgSendCarData 向从核发送目标速度 0 和 0 激活从核接收回调函数。

之后即可调用 rpmsgGetCarData 获取从核发来的小车实际速度数据，调用 rpmsgSendCarData 向从核发送小车目标速度。

需要退出时，调用 rpmsgQuit 即可关闭驱动文件，以便下次打开。



关键代码分析：

驱动及控制节点启动：

基于飞腾提供的 `rpmsg_demo.c`，实现了对控制驱动，控制节点，通信驱动的开启及初始化。之后调用调用 `rpmsgSendCarData` 向从核发送目标速度 0 和 0 激活从核接收回调函数。

```
int rpmsgInit()
{
    int ret;
    ctrl_fd = open("/dev/rpmsg_ctrl0", O_RDWR);
    if (ctrl_fd < 0) {
        perror("open rpmsg_ctrl0 failed.\n");
        return -1;
    }
    memcpy(eptinfo.name, "xxxx", 32);
    eptinfo.src = 0;
    eptinfo.dst = 0;
    ret = ioctl(ctrl_fd, RPMSG_CREATE_EPT_IOCTL, eptinfo);
    if (ret != 0) {
        perror("ioctl RPMSG_CREATE_EPT_IOCTL failed.\n");
        close(ctrl_fd);
        return 0;
    }
    rpmsg_fd = open("/dev/rpmsg0", O_RDWR);
    if (rpmsg_fd < 0) {
        perror("open rpmsg0 failed.\n");
        close(rpmsg_fd);
        close(ctrl_fd);
        return 1;
    }
    memset(&fds, 0, sizeof(struct pollfd));
    fds.fd = rpmsg_fd;
    fds.events |= POLLIN;
    //Because in the slave core program, the send action is in the receive
    callback function,
    //so we need to send a message to the slave core to trigger the receive
    callback function,
    //then the slave core can receive the message from the master core and
    send back the data.
    rpmsgSendCarData(0.0,0.0);
    return 2;
}
```

目标速度发送函数：

根据从核 `rpmsg_echo_os.c` 中回调函数定义好的接收目标速度信息格式 "`%.2f %.2f`", 利用 `snprintf` 将传入的目标速度按其格式存入 `buf` 中后, 调用飞腾提供的 `rpmsg_demo.c` 中提供的 `write_full` 函数将其写入到驱动文件中实现目标速度发送。

```
int rpmsgSendCarData(float Vx, float Vz)
{
    char buf[32];
    int ret;
    snprintf(buf, sizeof(buf), "%.2f %.2f", Vx, Vz); // 将用户输入放入 buf
    perror(buf);
    ret = write_full(buf, sizeof(buf)); // 发送数据
    return ret;
}
```

当前状态接收：

首先调用 `poll` 查询当前是否收到数据, 若收到数据则调用飞腾提供的 `rpmsg_demo.c` 中提供的 `read_full` 函数, 将读取到的信息存入 `buf`, 然后根据从核 `rpmsg_echo_os.c` 中回调函数定义好的发送当前状态信息格式 "`%d %f %f`", 利用 `sscanf` 读取是否触发紧急制动 `break_flag`, 当前实际速度 `Vx` 和当前实际速度 `Vz`。

```
int rpmsgGetCarData(int *break_flag, float *Vx, float *Vz)
{
    char buf[32];
    int ret;
    ret = poll(&fds, 1, 0);
    if (ret < 0) {
        return ret;
    }
    memset(buf, 0, 32);
    ret = read_full(buf, 32);
    if (ret < 0) {
        return ret;
    }
    sscanf(buf, "%d %f %f", break_flag, Vx, Vz);
    return ret;
}
```

3.2.2 Mainwindow.cpp 模块

Mainwindow.cpp 代码分析：

主要功能介绍：

Mainwindow 类对象被创建在 **car_gui.cpp** 节点中，主要用于显示人机交互界面并利用 **RPMsg** 通道进行主从核通信。

Mainwindow 类的功能包括显示当前状态条，显示当前小车朝向，显示当前小车实际速度，接收鼠标移动摇杆动作并得到当前摇杆位置，启动按钮启动从核 FreeRTOS 系统并调用 **rpmsg.h** 初始化函数创建 **RPMsg** 通道，停止按钮关闭交互界面并调用 **rpmsg.h** 退出函数关闭 **RPMsg** 通道，调用 **rpmsg.h** 中的 **rpmsgGetCarData** 和 **rpmsgSendCarData** 函数来利用 **RPMsg** 通道与从核进行通信。

主要流程：

Mainwindow 类对象创建后，其构造函数会对基于 **QLabel** 控件实现的状态提示条和速度显示屏标题，基于 **QPushButton** 控件实现的启动/停止按钮，基于 **QComboBox** 实现的最高速度选择栏，基于 **QLCDNumber** 实现的小车实际速度显示屏，以及调用 **QPoint**、**QPainter** 和 **QPixmap** 实现的摇杆和小车朝向进行初始化设置。其中开始按钮 **btnStart** 的按下信号与 **pushButtonStart_Clicked** 槽进行连接。

之后状态提示条会显示等待启动。启动按钮按下后，在 **pushButtonStart_Clicked** 槽内，首先会启动从核 **openamp.elf** 可执行文件并加载 **RPMsg** 驱动，然后会调用 **rpmsg.h** 中的初始化函数加载 **RPMsg** 驱动文件。之后将 **btnStart** 文字改为停止变成停止按钮，断开 **btnStart** 与 **pushButtonStart_Clicked** 槽的连接并改为与 **pushButtonEnd_Clicked** 槽相连。

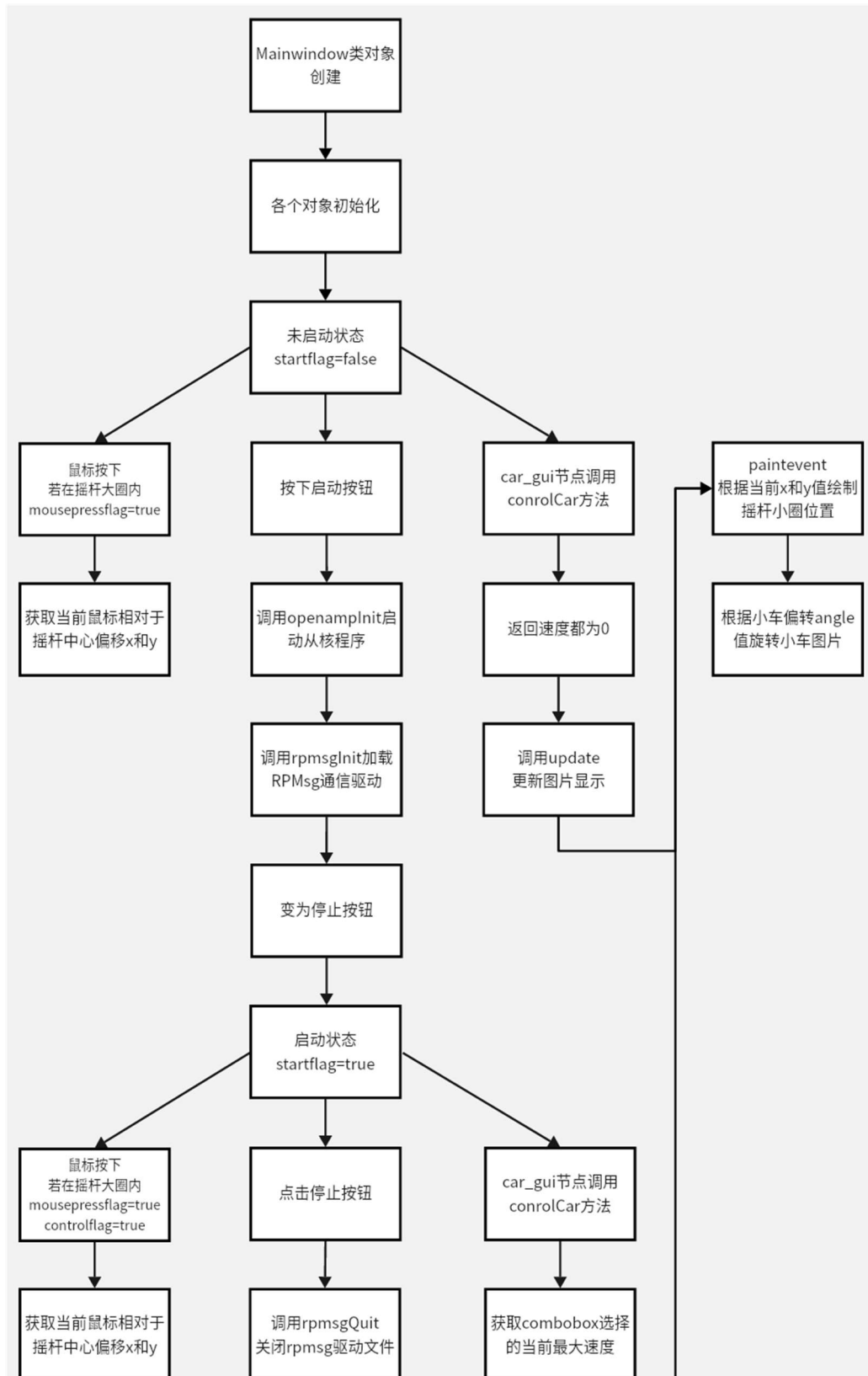
启动完成后，当鼠标在摇杆大圈内按下，即移动了摇杆，就会将 **MousePressFlag** 标志位置 **true**。**mouseMoveEvent** 就会得到鼠标点击的位置与摇杆中心点的位置差得到摇杆位置 **x** 和 **y**。

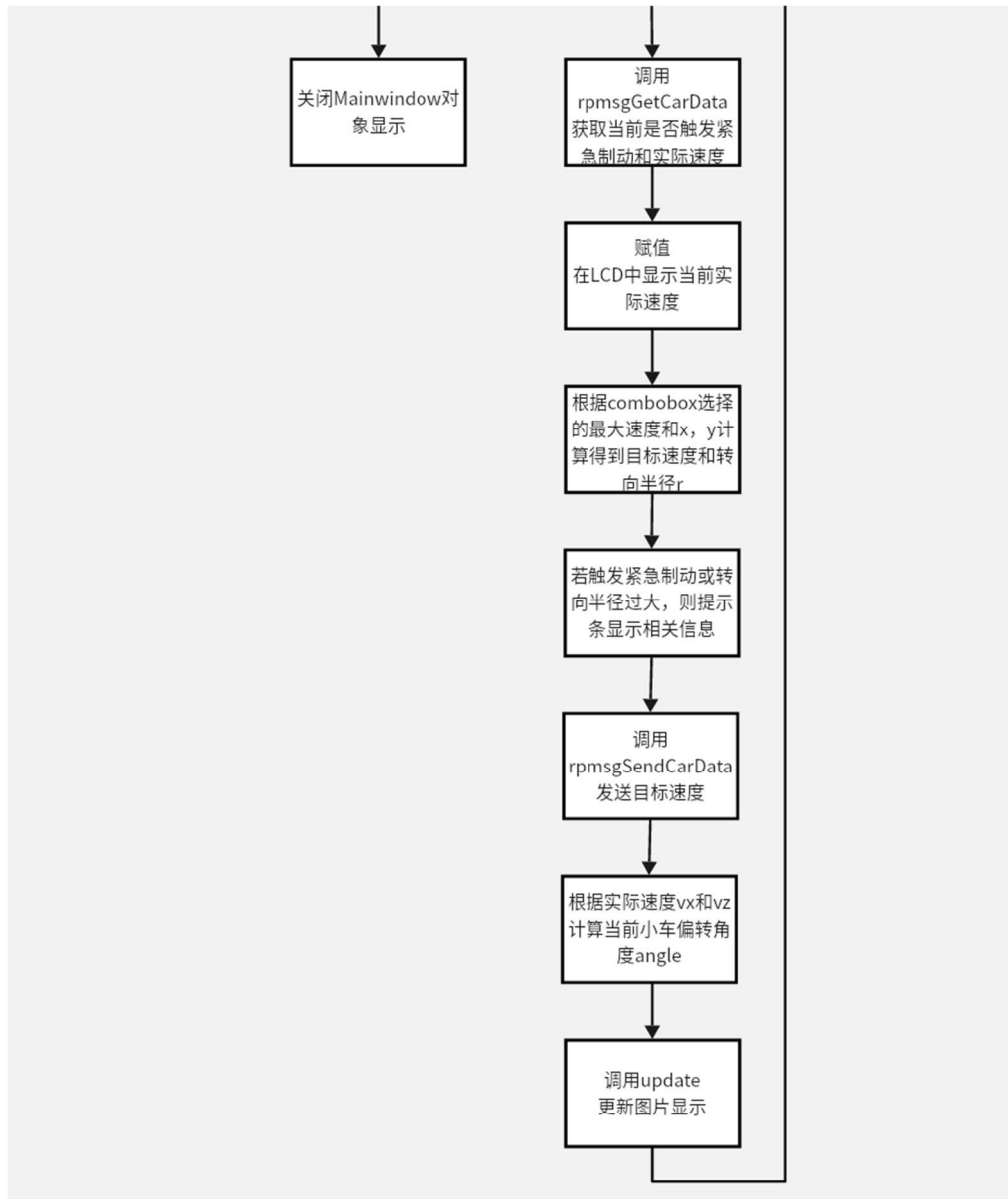
另外还有保存地图功能，点击保存按钮之后，就会将 **rviz** 中生成的地图保存到预设的位置。

当 **car_gui** 节点调用 **Mainwindow** 类中的 **controlCar** 方法后，首先会判断是否启动，如果没有启动就返回速度为 0。如果启动完成了，就会调用 **rpmsgGetCarData** 得到从核传来的实际速度，并将其赋值给 **car_gui.cpp** 传来的指针变量来实现速度的传回。之后会根据摇杆的 **x** 和 **y** 得到控制的目标速度，并调用 **rpmsgSendCarData** 向从核发送目标速度。最后会对显示界面进行更新，调用

paintEvent, 将小摇杆图片的位置移动到当前鼠标的位置, 并对小车图片进行旋转。

最后当按下停止按钮时, 就会调用 rpmMsgQuit 关闭驱动文件并关闭此窗口





关键代码分析:

从核启动程序:

```

void MainWindow::openampInit()
{
    system("echo start > /sys/class/remoteproc/remoteproc0/state");
    delay_ms(2000);
    labelError->setText("启动完成! 正在启动从核通信...");
    system("echo rpmsg_chrdev > /sys/bus/rpmsg/devices/virtio0.rpmsg-openamp-demo-channel.-1.0/driver_override");
    delay_ms(1000);
    system("modprobe rpmsg_char");
}

```

利用 Qt 中的 `system` 函数，模拟终端发送指令，启动 `remoteproc` 从核。之后加载 `rpmsg_char` 驱动。

小车控制程序：

首先检查是否完成启动，若未完成则直接传回 `vx` 和 `vy` 速度均为 0。

```
void MainWindow::controlCar(int* flag,float* vx, float* vz)
{
    //Initialize the speed and the break flag
    int break_flag = 0;
    float current_vx = 0.0, current_vz = 0.0, target_vx = 0.0, target_vz = 0.0, r = 0.0;
    //If the car is started and the control is enabled, will get the current speed and the break flag of the car
    if(startFlag && controlFlag){...
    }else{
        //If the car is not started or the control is not enabled
        //The break flag will be set to 1
        break_flag = 1;
        //The current speed will be set to 0
        current_vx = 0;
        current_vz = 0;
        //Assign the current speed and the break flag to the flag and the vx and vz
        *flag = break_flag;
        *vx = current_vx;
        *vz = current_vz;
    }
}
```

若已完成启动，则先获取当前选择的最高速度，然后调用 `rpmsgGetCarData` 得到是否触发紧急制动，当前前进速度和当前转向速度，并将其赋值给传入的指针变量，然后在速度显示 LCD 控件中显示当前前进速度和转向速度

```
//Get the max speed according to the speed control combo
int index = speedBox->currentIndex();
switch(index){
    case 0: maxSpeed = 0.5; break;
    case 1: maxSpeed = 0.75; break;
    case 2: maxSpeed = 1.0; break;
    case 3: maxSpeed = 1.5; break;
}
//Get the current speed and the break flag of the car using rpmsgGetCarData
rpmsgGetCarData(&break_flag,&current_vx,&current_vz);
```

```

        //Assign the current speed and the break flag to the flag and the
vx and vz
        //Transfer these data back to car_gui node
        *flag = break_flag;
        *vx = current_vx;
        *vz = current_vz;
        //Display the current speed on the lcd display
        lcdDisplayVx->display(current_vx);
        lcdDisplayVz->display(current_vz);

```

之后根据摇杆位置 x 和 y ，结合选择的最高速度得到目标速度和转向半径。之后根据是否触发紧急制动或转向角度是否超过限幅来在提示条上显示提示信息。最后调用 `rpmsgSendCarData` 发送目标速度

```

//Calculate the target speed according to the position of the rocker
target_vx = -1.0 * maxSpeed * y / 90.0;
target_vz = -1.0 * maxSpeed * x / 90.0;
//if the target_vx is negative, the target_vz will be negative
if (target_vx < 0) {
    target_vz = target_vz * -1.0;
}
r=target_vx / target_vz;
if(break_flag == 1){
    //If the break flag is 1, which means there is an obstacle in front
of the car, the error message will be displayed
    labelError->setText("前方有障碍物，紧急制动触发!");
}else if((r < 0.35 && r > 0) || (r < 0 && r > -0.35)){
    //If the angle is too large, the error message will be displayed
    labelError->setText("转向角度超过限幅!");
}else{
    //If the control is normal, the error message will be cleared
    labelError->setText("");
}
//Send the target speed to the slave core using rpmsgSendCarData
rpmsgSendCarData(target_vx,target_vz);

```

最后根据小车实际速度，更新小车的图像转角。

```

//Update the car display angle according to the current speed
angle = 0.0;
if(current_vx > 0.01){
    int Vx = 100 * current_vx;
    int Vz = 100 * current_vz;
    angle = atan(-1 * Vz / Vx) * (180.0 / 3.1415926) * 1.0;
}else if(current_vx < -0.01){
    int Vx = 100 * current_vx;

```

```

        int Vz = 100 * current_vz;
        angle = atan(Vz / Vx) * (180.0 / 3.1415926) * 1.0;
    }
    //Update the window display
    update();
}

```

摇杆移动:

首先，若要控制摇杆移动肯定要先按下鼠标，按下鼠标会触发 `mousePressEvent`。首先判断鼠标是否在摇杆大圈的范围内按下，如果不是则代表用户并不想移动摇杆；如果是则对鼠标按下控制摇杆标志位置 1，同时若是启动完成后移动摇杆则代表用户想要控制小车，还需将控制标志位置 1。

```

void MainWindow::mousePressEvent(QMouseEvent* e)
{
    QPoint rocker_xy;

    //Get the current mouse position
    rocker_xy = e->pos();
    //QDebug() << "rocker position:" << rocker_xy;

    //If the mouse is pressed inside the big circle, will set the
    MousePressFlag to true
    if (pow((rocker_xy.x() - SmallCir_xy.x()), 2) + pow((rocker_xy.y() -
    SmallCir_xy.y()), 2) <= 8100)
    {
        MousePressFlag = true;
        //If the rpsmsg communication is started, and mouse is pressed to
        control the rock
        //will set the control flag to true
        if(startFlag){
            controlFlag = true;
        }
    }
    else
    {
        MapRemov_Old = rocker_xy;
    }
}

```

鼠标按下后，当鼠标移动时，则代表用户想要移动摇杆，就需要得到当前鼠标的实时位置，即摇杆位置。首先需要判定鼠标是否在摇杆大圈内按下，若否则无需移动摇杆；若是，则需先判断鼠标是否移动到摇杆大圈外，若是，则需要移动摇杆小圈到大圈的边缘处；若否则直接将摇杆移动到鼠标处即可

```

void MainWindow::mouseMoveEvent(QMouseEvent* e)

```

```

{
    QPoint rocker_xy;
    QByteArray xy;
    xy.resize(2);
    //Get the current mouse position
    rocker_xy = e->pos();
    //If the mouse is pressed inside the big circle, will calculate the
    position of the rocker
    if (MousePressFlag)
    {
        //If the small circle is out of the big circle, the rocker will be
        at the edge of the big circle
        if (pow((rocker_xy.x() - SmallCir_xy.x()), 2) + pow((rocker_xy.y()
        - SmallCir_xy.y()), 2) > 8100)
        {
            //Calculate the x and y of the rocker
            x = int(90 * cos(atan2(abs(rocker_xy.y() - SmallCir_xy.y()),
            abs(rocker_xy.x() - SmallCir_xy.x()))));
            y = int(90 * sin(atan2(abs(rocker_xy.y() - SmallCir_xy.y()),
            abs(rocker_xy.x() - SmallCir_xy.x()))));

            //Calculate the position of the rocker according to the position
            of the mouse
            if (rocker_xy.x() > SmallCir_xy.x() && rocker_xy.y() >
            SmallCir_xy.y())
            {
                //The first quadrant
                BigCir_xy.setX(x + SmallCir_xy.x());
                BigCir_xy.setY(y + SmallCir_xy.y());
            }
            else if (rocker_xy.x() < SmallCir_xy.x() && rocker_xy.y() >
            SmallCir_xy.y())
            {
                //The second quadrant
                BigCir_xy.setX(-x + SmallCir_xy.x());
                BigCir_xy.setY(y + SmallCir_xy.y());
                x = -x;
            }
            else if (rocker_xy.x() < SmallCir_xy.x() && rocker_xy.y() <
            SmallCir_xy.y())
            {
                //The third quadrant
                BigCir_xy.setX(-x + SmallCir_xy.x());
                BigCir_xy.setY(-y + SmallCir_xy.y());
            }
        }
    }
}

```

```

        x = -x;
        y = -y;
    }
    else if (rocker_xy.x() > SmallCir_xy.x() && rocker_xy.y() <
SmallCir_xy.y())
    {
        //The fourth quadrant
        BigCir_xy.setX(x + SmallCir_xy.x());
        BigCir_xy.setY(-y + SmallCir_xy.y());
        y = -y;
    }
}
else//The rocker is inside the big circle and the position of the
rocker is the same as the mouse
{
    BigCir_xy = rocker_xy;
    x = rocker_xy.x() - SmallCir_xy.x();
    y = rocker_xy.y() - SmallCir_xy.y();
}
xy[0] = char(x);
xy[1] = char(y);
//QDebug() << x << y;
//update();
}
MapRemov_Old = rocker_xy;
}

```

之后当 Update 时，会调用 paintEvent，根据 mouseMoveEvent 中得到的 x 和 y，绘制小圈

```

//draw the small circle of the rocker
painter.drawPixmap(BigCir_xy.x() - SMALL_CIRCLE_RADIUS, BigCir_xy.y()
- SMALL_CIRCLE_RADIUS, \
    SMALL_CIRCLE_RADIUS * 2, SMALL_CIRCLE_RADIUS * 2,
smallCircle_Pixmap);

```

最后，当鼠标释放后，会调用 mouseReleaseEvent，在其中将鼠标按下标志位置 0 并将 x 和 y 都置 0.

```

void MainWindow::mouseReleaseEvent(QMouseEvent* e)
{
    Q_UNUSED(e);
    //The mouse is released, reset the MousePressFlag to false
    MousePressFlag = false;
    //The rocker will return to the center of the big circle
    BigCir_xy.setX(SmallCir_xy.x());
    BigCir_xy.setY(SmallCir_xy.y());
    x=0;

```

```

        y=0;
    }
    另外，还增加了保存地图功能，当点击保存按钮时，就会将当前地图保存到预设位置。
void MainWindow::pushButtonSave_Clicked()
{
    system("cd /home/user/car_ws");
    system("source install/setup.bash");
    system("ros2 run nav2_map_server map_saver_cli -t map -f car_map");
    labelError->setText("保存成功！已保存到 car_ws 目录下！");
}

```

3.2.3 car_gui.cpp 模块

car_gui.cpp 代码分析：

主要功能介绍：

这个文件的主要作用是在 ROS 框架下创建 Publisher 节点并挂载 Mainwindow，并开启动态调度，同时创建 Publisher 对象。

在 Publisher 的构造函数中，首先初始化了 ROS 框架下的定时器，并设置了定时周期以及注册了中断服务函数。然后，初始化了 Odometry 里程计消息发布对象，并创建并显示了 Mainwindow 类对象。

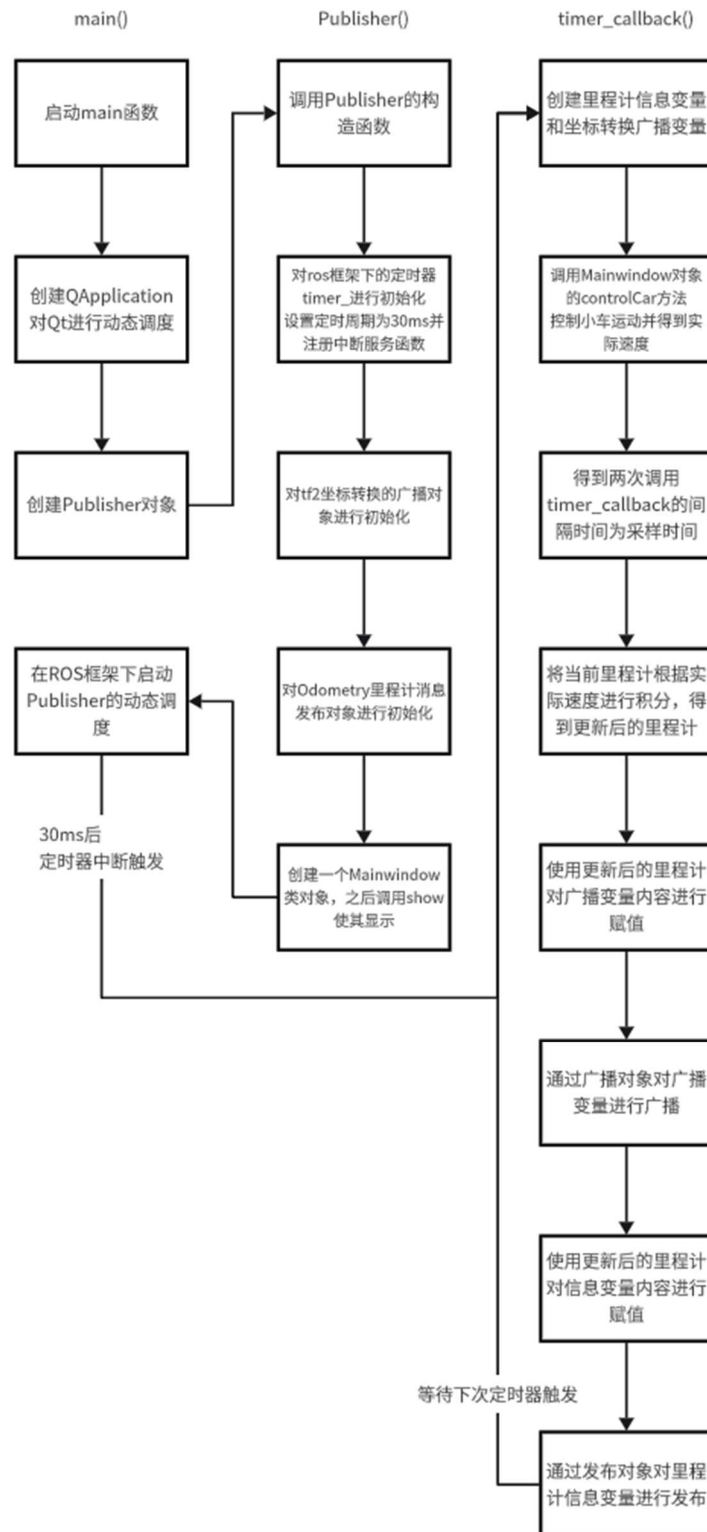
在 ROS 框架下启动 Publisher 的动态调度后，它会等待定时器中断触发，一旦中断发生，就会进入中断服务函数，该函数会调用 Mainwindow 中的 controlCar 方法，通过 rmsg 通道对底盘下发目标速度，并得到实际速度，然后根据实际速度更新里程计，并将新的里程计广播和发布。最后，Publisher 会等待下次定时器中断触发。

主要流程：

启动 main 函数后，开始创建 QApplication 对 Qt 进行动态调度，创建 Publisher 对象并调用其构造函数，在这个函数中，对 ros 框架下的定时器 timer_进行初始化，设置定时周期为 30ms 并注册中断服务函数，对 tf2 坐标转换的广播对象和 Odometry 里程计消息发布对象进行初始化，创建并挂载一个 Mainwindow 类对象，之后调用 show 使其显示。

接着在 ROS 框架下启动 Publisher 的动态调度，等待 30ms 后定时器中断触发进入 timer_callback 中断服务函数，先创建里程计信息变量和坐标转换广播变量，接着调用 Mainwindow 中的 controlCar 方法，通过 rmsg 通道对底盘下发目标速

度，并得到实际速度，这里我们以两次调用 `timer_callback` 的间隔时间为采样时间（较为准确），将当前里程计根据实际速度进行积分，得到更新后的里程计，然后将新的里程计赋值给信息变量和广播变量赋值并进行发布和广播。最后重新等待以 `30ms` 为周期的定时器中断触发。



关键代码分析：

Publisher 构造函数：

Publisher 构造函数以 "car_gui" 作为节点名称初始化，然后初始化了一个定时器 timer_，设置定时周期为 30ms，每次触发定时器时会调用 timer_callback 函数。接着进行了 tf2 坐标转换广播器 odom_broadcaster_ 和 Odometry 里程计消息发布者 odom_publisher_ 的初始化，其中里程计发布者将发布 odometry 消息。最后创建了一个 MainWindow 对象 mainwindow 并调用 show 显示。

```
Publisher::Publisher() : Node("car_gui")
{
    RCLCPP_INFO(this->get_logger(), "node %s started", "car_gui");
    // timer initialization, the timer will call the timer_callback function
    every 30ms
    timer_ =
this->create_wall_timer(std::chrono::milliseconds(30), std::bind(&Publisher::timer_callback, this));
    // tf2 broadcaster and odom publisher initialization, the odom publisher
    will publish the odometry message
    odom_broadcaster_ =
std::make_shared<tf2_ros::TransformBroadcaster>(this);
    odom_publisher_ =
this->create_publisher<nav_msgs::msg::Odometry>("odom", 50);
    // mainwindow initialization, the mainwindow is used to control the car
    and display the GUI
    mainwindow = new MainWindow();
    mainwindow->show();
}
```

中断服务函数：

首先进行了里程计消息和 tf2 坐标转换的初始化，并初始化了紧急制动标志和速度变量。然后调用 mainwindow 对象的 controlCar 方法，通过 rpmsg 通道对底盘下发目标速度，并得到实际速度。接着进行了实际采样时间的计算和里程计计算的修正。在里程计计算部分，根据当前速度和采样时间更新了车辆在 odom 坐标系下的位置和偏航角，并发布了里程计消息和 tf2 坐标转换。

```
void Publisher::timer_callback()
{
    geometry_msgs::msg::TransformStamped odom_trans;
    nav_msgs::msg::Odometry odom_msg;
    int flag = 0;
    float vx = 0.0, vz = 0.0;
```

```

// call the mainwindow's controlCar function to get the current car speed
mainwindow->controlCar(&flag, &vx, &vz);
//RCLCPP_INFO(this->get_logger(),"publishing: %f %f", vx, vz);
// odometry calculation correction
vx = vx * 2.0;
vz = vz * 2.0;
// sampling time calculation
now_time_ = this->now();
if(last_time_.seconds() == 0) last_time_ = now_time_;
sampling_time = (now_time_ - last_time_).seconds();
last_time_ = now_time_;
/*****odometry calculation*****/
// x and y are the position of the car in the odom frame
// z is the yaw of the car in the odom frame
x += (vx*cos(z)) * sampling_time;
y += (vx*sin(z)) * sampling_time;
z += vz * sampling_time;
// print the odometry message for testing
RCLCPP_INFO(this->get_logger(),"x: %lf y: %lf
z: %lf %lf",x,y,z,sampling_time);
/*****odometry message publishing*****/
// orientation calculation
tf2::Quaternion orientation;
// call the setRPY function to set the orientation of the car according
to the yaw
orientation.setRPY(0.0,0.0,z);
odom_trans.header.stamp = this->now();
odom_trans.header.frame_id = "odom";
odom_trans.child_frame_id = "base_link";
odom_trans.transform.translation.x = 0 - x;
odom_trans.transform.translation.y = 0 - y;
odom_trans.transform.rotation = tf2::toMsg(orientation);
odom_broadcaster_->sendTransform(odom_trans);
// set the odometry message x, y, vx, vz
// x, y are the position of the car in the odom frame
// vx is the linear velocity of the car
// vz is the angular velocity of the car
odom_msg.header.stamp = this->now();
odom_msg.header.frame_id = "odom";
odom_msg.pose.pose.position.x = 0 - x;
odom_msg.pose.pose.position.y = 0 - y;
odom_msg.pose.pose.orientation = tf2::toMsg(orientation);
odom_msg.child_frame_id = "base_link";
odom_msg.twist.twist.linear.x = -vx;

```

```
    odom_msg.twist.twist.angular.z = -vz;  
    odom_publisher_->publish(odom_msg);  
}
```

3.2.4 rpmsg-echo_os.c 模块

rpmsg-echo_os.c 代码分析:

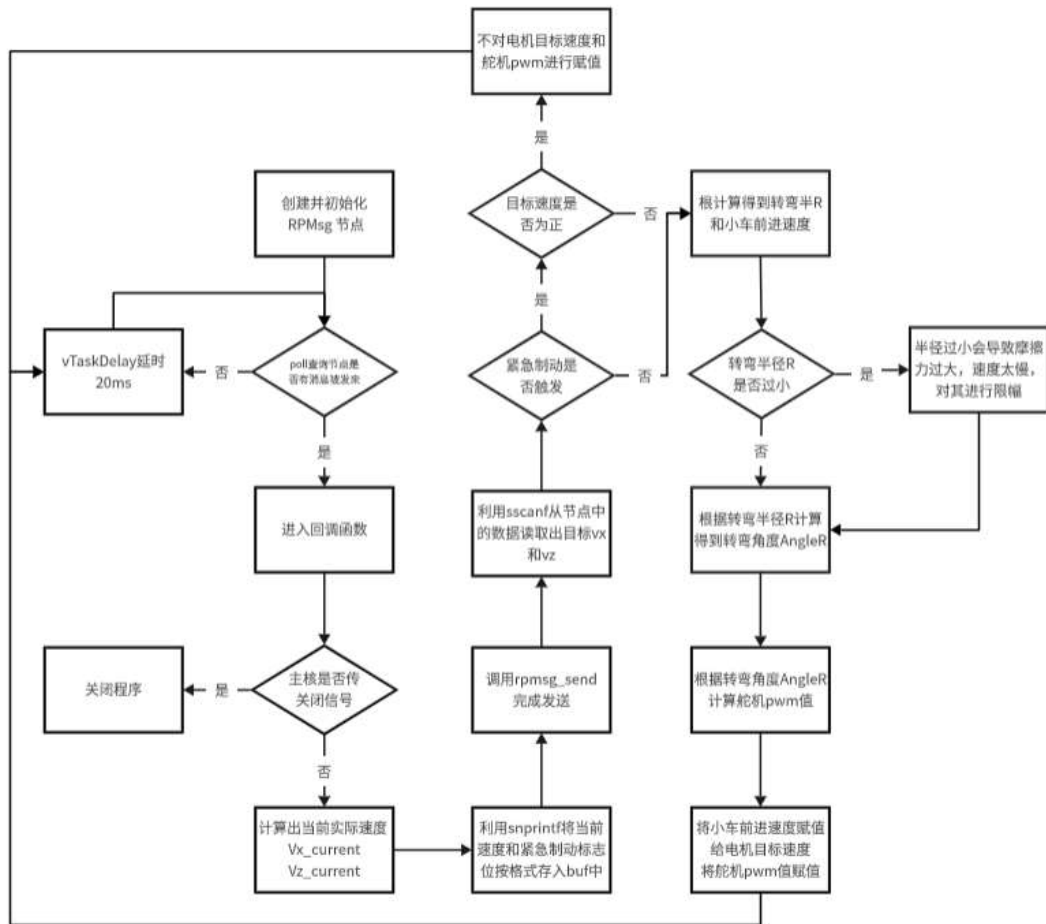
主要功能介绍:

这段代码在 FreeRTOS 系统上运行, 创建且初始化 rpmsg 节点, 通过 RPMsg 通信实现了从核与主核之间的双向通信。它接收来自主核发送的目标速度数据和控制指令, 并根据需求判断是否需要触发紧急制动, 然后处理这些数据以控制小车的运动, 并将小车当前的状态数据发送回主核。

主要流程:

首先通过 FRpmsgEchoApp 函数创建并初始化了 RPMsg 节点, 每 20 毫秒查询一次是否有消息发来, 若接收到主核发来的消息, 则进入 rpmsg_endpoint_cb 回调函数。在此回调函数中, 首先检查是否收到了关闭程序的信号, 若没有, 则计算并发送当前实际速度 $Vx_current$ 和 $Vz_current$, 以及紧急制动标志位给主核。

接着, 解析主核发送的目标速度数据 Vx 和 Vz , 若未触发紧急制动, 则根据阿克曼小车的转弯半径公式 $R = Vx / Vz$ 计算转弯半径, 并检查其是否符合要求。如果转弯半径不符合要求, 则调整目标角速度以保证转弯半径达到要求。然后, 根据转弯半径计算舵机转动所需的 PWM 值, 并将电机目标速度和舵机 PWM 值数据下发给控制小车运动的 car.c 模块。



关键代码分析：

节点创建并初始化函数：

```

int FRpmMsgEchoApp(struct rpmMsg_device* rdev, void* priv)
{
    int ret;
    struct rpmMsg_endpoint lept;
    shutdown_req = 0;
    OPENAMP_SLAVE_INFO("Try to create rpmMsg endpoint.");
    /* Initialize RPMMSG framework */
    创建并初始化 rpmMsg 节点
    ret = rpmMsg_create_ept(&lept, rdev, RPMMSG_SERVICE_NAME, 0,
    RPMMSG_ADDR_ANY, rpmMsg_endpoint_cb, rpmMsg_service_unbind);
    if (ret)
    {
        OPENAMP_SLAVE_ERROR("Failed to create endpoint,ret = %d.\r\n",
    ret);
        return -1;
    }
}
  
```

进入一个循环，poll 查询节点是否有消息被发来，若有则进入回调函数，若没有则 vTaskDelay 延时 20ms，回到循环。

```
while (1)
{
    platform_poll_loop(priv);
    /* we got a shutdown request, exit */
    if (shutdown_req)
    {
        break;
    }
    vTaskDelay(RPMSG_POLL_PERIOD);
}
rpmsg_destroy_ept(&lept);
return 0;
},
```

回调函数：

接收到主核发来的消息后，进入回调函数。

```
static int rpmsg_endpoint_cb(struct rpmsg_endpoint* ept, void* data, size_t
len, uint32_t src, void* priv)
{
    (void)priv;
    (void)src;
    /* On reception of a shutdown we signal the application to terminate */
    if ((*((unsigned int*)data) == SHUTDOWN_MSG) {
        OPENAMP_SLAVE_INFO("shutdown message is received.");
        shutdown_req = 1;
        return RPMSG_SUCCESS;
    }
}
```

若从主核接收到关闭信号则关闭程序。

```
memset(cmd, 0, len);
memcpy(cmd, data, len);
printf("success 从核收到消息: %s\r\n", cmd);
const char* str = cmd;
char* token;
char* remaining = (char*)str;
float numbers[2];
sscanf(cmd, " %f %f", &numbers[0], &numbers[1]);
```

利用 sscanf 从节点中的数据读取出主核发来的消息 numbers[0]，numbers[1]，即目标速度 vx，vz。

```
float Vx_current, Vz_current;
Vx_current = MotorASpeed;
```

```

    if(R == 0) Vz_current = 0;
    else Vz_current = MotorASpeed / R;
    uint8_t buf[32];
    snprintf(buf, sizeof(buf), " %d %.2f %.2f", front_sensor_flag,
Vx_current,Vz_current);
    // Send data back to master
    if (rpmsg_send(ept, buf, sizeof(buf)) < 0)
        OPENAMP_SLAVE_ERROR("rpmsg_send failed!!!\r\n");

```

计算出当前实际速度 $Vx_current$ 、 $Vz_current$ 之后，利用 `snprintf` 将当前速度和紧急制动标志位按格式存入 `buf` 中，接着调用 `rpmsg_send` 完成发送。

```

float Vx, Vz;
Vx = numbers[0];
Vz = numbers[1];
if (front_sensor_flag == 1 && Vx > 0) {
    //printf("前紧急避障触发后，小车无法前进，仅能后退!!! \r\n 请后退直到前方检测不到障碍物\r\n");
}

```

若触发紧急制动，且目标速度 Vx 为正时，不对电机目标速度和舵机 `pwm` 进行赋值。

否则，根据计算得到转弯半径 R 和小车前进速度 Vx 。由于转弯半径过小会导致摩擦力过大，速度太慢，要对其进行限幅，如果转弯半径过小则降低降低目标角速度，配合前进速度，提高转弯半径到最小转弯半径。接着根据符合要求的转弯半径 R 计算得到转弯角度 $AngleR$ ，再由此计算舵机 `pwm` 值。最后将小车前进速度赋值给电机目标速度，将舵机 `pwm` 值赋值。

```

else {
    float AngleR, Angle_Servo;
    if(Vz!=0 && Vx!=0)
    {
        if(float_abs(Vx/Vz)<=MINRADIUS)
        {
            if(Vz>0)
                Vz= float_abs(Vx)/(MINRADIUS);
            else
                Vz=-float_abs(Vx)/(MINRADIUS);
        }
        R=Vx/Vz;
        AngleR=atan(SPACING/R);
    }
    else
    {
        AngleR=0;
    }
}

```

```

        // Front wheel steering Angle limit (front wheel steering Angle
        controlled by steering engine), unit: rad
        //前轮转向角度限幅(舵机控制前轮转向角度), 单位: rad
        AngleR = target_limit_float(AngleR, -0.4f, 0.4f);
        R = SPACING / tan(AngleR);
        //Inverse kinematics //运动学逆解
        motor_target_speed = Vx;
        //舵机 PWM 值, 舵机控制前轮转向角度
        Angle_Servo = -0.628f * pow(AngleR, 3) + 1.269f * pow(AngleR, 2) -
        1.772f * AngleR + 1.573f;
        pwm_steer_pulse = 150 + (Angle_Servo - 1.572f) * RATIO;
        //电机目标速度限幅
        motor_target_speed = target_limit_float(motor_target_speed, -1.5,
        1.5);
        pwm_steer_pulse = target_limit_int(pwm_steer_pulse, 80,
        220); //Servo PWM value limit //舵机 PWM 值限幅
    }
    return RMSG_SUCCESS;
}

```

3.2.5 car.c 模块

car.c 代码分析:

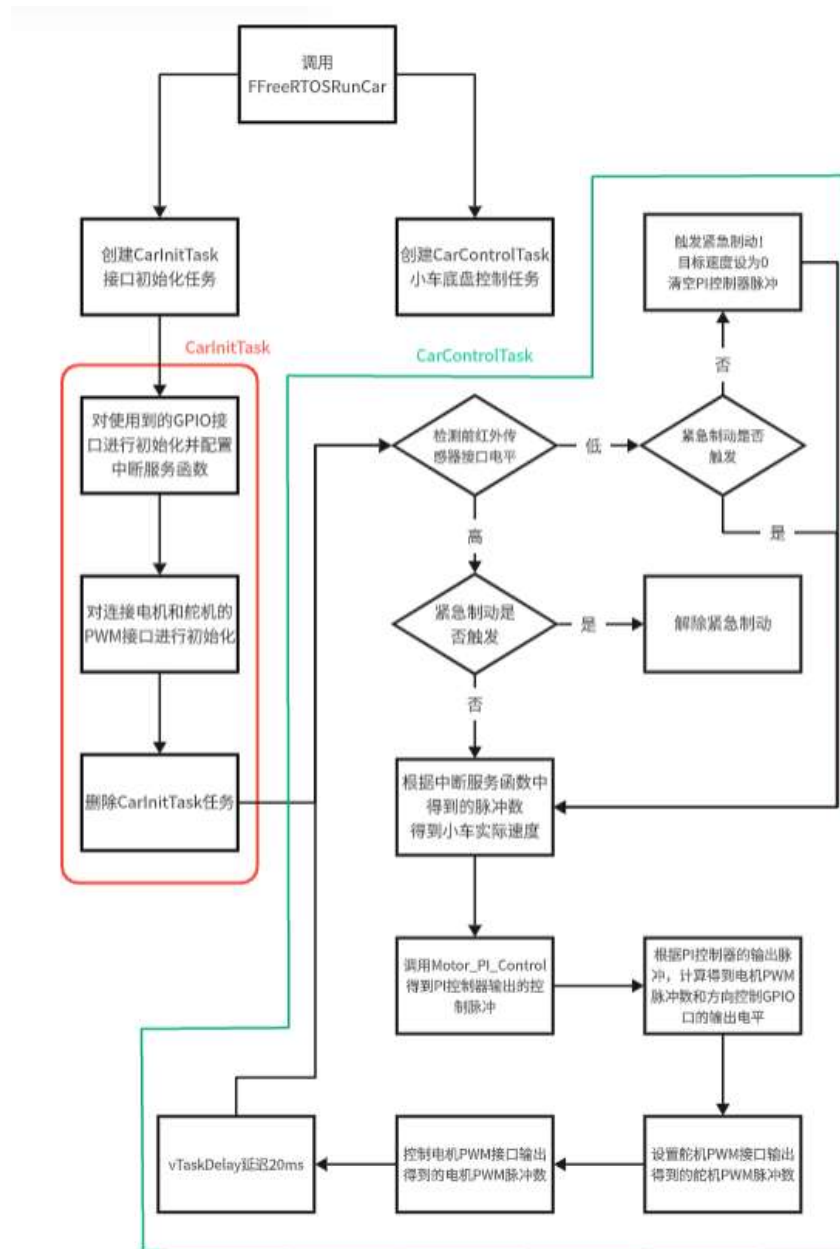
主要功能介绍:

这段代码通过调用 FFreeRTOSRunCar 函数来创建了接口初始化任务和小车底盘控制任务。接口初始化任务负责初始化小车的各个引脚以及 PWM 参数, 而小车底盘控制任务则实现了一系列功能: 检测前方是否有障碍物, 并根据情况触发紧急制动; 利用 PI 控制器计算电机脉冲以实现速度控制, 通过控制舵机角度来实现转向; 最后, 通过周期性执行控制任务, 实现对小车运动的实时控制。

主要流程:

调用 FFreeRTOSRunCar 函数来创建了接口初始化任务 CarInitTask, 完成了小车的各个引脚以及 PWM 参数的初始化之后删除 CarInitTask; 继续在 FFreeRTOSRunCar 函数下创建了小车底盘控制任务 CarControlTask, 调用 CarDetectFront 读取检测红外传感器信号的 gpio 口的电平信号以检测前方障碍物, 并根据情况触发紧急制动; 随后调用 CarGetCurrentSpeed 函数获取了当前速度后, 结合目标速度调用 Motor_PI_Control 计算电机脉冲 pwm, 再调用 CarControlSteer

给舵机 pwm 接口赋值舵机控制脉冲 pwm，以控制小车转弯方向，然后调用 CarControlMotor 给电机电机机 pwm、gpio 接口赋值，以控制小车运动的前后方向和速度。最后 vTaskDelay 20ms 后再次从 CarDetectFront 开始，即以 20ms 周期循环执行底盘控制任务 CarControlTask 来实现对小车运动的实时控制。



关键代码分析：

检测前方障碍物函数：

读取检测红外传感器信号的 gpio 口的电平信号，若为低电平且标志位为 0（表示之前没有触发紧急制动），则表示检测到前方有障碍物了，将触发紧急避障，此时小车无法前进只能后退，将紧急制动标志位置 1，将电机的目标速度设为 0，

并清除 PI 控制器的脉冲；如果检测红外传感器信号的 gpio 口的电平信号为高电平且标志位为 1（表示之前触发了紧急制动），则表示前方已经没有障碍物了，此时取消紧急制动：将标志位清零。

```
static void CarDetectFront()
{
    //Detect front obstacle, if front sensor is low, trigger emergency break
    if (FGpioGetInputValue(frontsensor_pin) == FGPIOPIN_LOW &&
front_sensor_flag == 0) {
        front_sensor_flag = 1;
        //If emergency break triggered, set target speed as 0 and clear PI
controller pulse, restart integral
        motor_target_speed = 0;
        pulse = 0;
    }
    //If front sensor is high, clear emergency break flag
    if (FGpioGetInputValue(frontsensor_pin) == FGPIOPIN_HIGH &&
front_sensor_flag == 1) {
        front_sensor_flag = 0;
    }
}
```

电机当前速度计算函数：

根据以下公式计算，编码器脉冲（MotorACount）：记录了一段时间内电机编码器的脉冲数；控制周期（Control_Period）：控制任务执行的周期，以毫秒为单位；轮子周长（Wheel_Perimeter）：轮子的周长，单位为米；编码器精度（Encorder_Precision）：编码器的精度，表示每转多少个脉冲。

```
static void CarGetCurrentSpeed()
{
    MotorASpeed = (MotorACount * (1000 / Control_Period) * Wheel_Perimeter)
/ (2.0 * Encorder_Precision);
    MotorACount = 0;
}
```

电机速度控制脉冲 pwm 计算函数：

根据当前速度和目标速度的偏差，用 pi 控制器调整电机速度控制脉冲 pwm 的值

```
int Motor_PI_Control(float current_speed, float target_speed) {
    bias = target_speed - current_speed;
    bias_change = bias - last_bias;
    pulse += Kp * (bias - last_bias) + Ki * bias;
```

```

    pulse = Pulse_limit_int(pulse, 1 - PWM_MOTOR_PERIOD, PWM_MOTOR_PERIOD
- 1);
    last_bias = bias;
    return pulse;
}

```

舵机转弯方向控制函数：

将 rpsmsg-echo_os.c 下发的舵机 pwm 脉冲赋值给舵机 pwm 接口以控制小车转弯方向。

```

static void CarControlSteer()
{
    FFreeRTOSPwmPulseSet(os_pwm_ctrl_steer_pin, PWM_STEER_CHANNEL_USE,
pwm_steer_pulse);
}

```

电机控制函数：

电机有一个控制转动速度的 pwm 接口 os_pwm_ctrl_motor_pin 和一个控制转动方向的 gpio 接口 out_gpio_direction，当 pwm 接口的电压高于 gpio 接口的电压时电机正转，反之反转，根据不同电压差来决定电机转动速度，由于硬件不能直接输出一系列连续电压，因此用 pwm 的形式模拟不同电压差。

根据 Motor_PI_Control 函数计算的电机控制脉冲 PI_Pulse 的正负进行判断：当电机控制脉冲 PI_Pulse 为正时，gpio 接口置低电平，将 PI_Pulse 赋值给 pwm 接口，电机正转；电机控制脉冲 PI_Pulse 为负时，gpio 接口置高电平，将在一个周期内取反后的 PI_Pulse 赋值给 pwm 接口，电机反转。

```

static void CarCalculateDirectionAndPulse(int PI_Pulse, int*
motor_direction, u32* pwm_motor_pulse)
{
    if (PI_Pulse < 0) {
        *motor_direction = 0;
        *pwm_motor_pulse = (0 - PI_Pulse);
    }
    else {
        *motor_direction = 1;
        *pwm_motor_pulse = PI_Pulse;
    }
}

```

```

static void CarControlMotor(int direction, int pwm_pulse)
{
    if (direction) {
        //前进
        FFreeRTOSPinWrite(out_gpio_direction, out_pin_direction,
FGPIO_PIN_LOW);
        //pwm 控制电机
        FFreeRTOSPwmPulseSet(os_pwm_ctrl_motor_pin,
PWM_MOTOR_CHANNEL_USE, pwm_pulse);
    }
}

```

```

    }
    else {
        //后退
        FFreeRTOSPinWrite(out_gpio_direction, out_pin_direction,
FGPIO_PIN_HIGH);
        //pwm 控制电机
        FFreeRTOSPwmPulseSet(os_pwm_ctrl_motor_pin,
PWM_MOTOR_CHANNEL_USE, PWM_MOTOR_PERIOD - pwm_pulse);
    }
    if (Debug_Info_Print) printf("设置电机方向为: %s, 电机脉冲为: %d\r\n",
(direction == 1) ? "前进" : "后退", pwm_pulse);
}

```

3.2.6 car_lidar_gui.py 模块

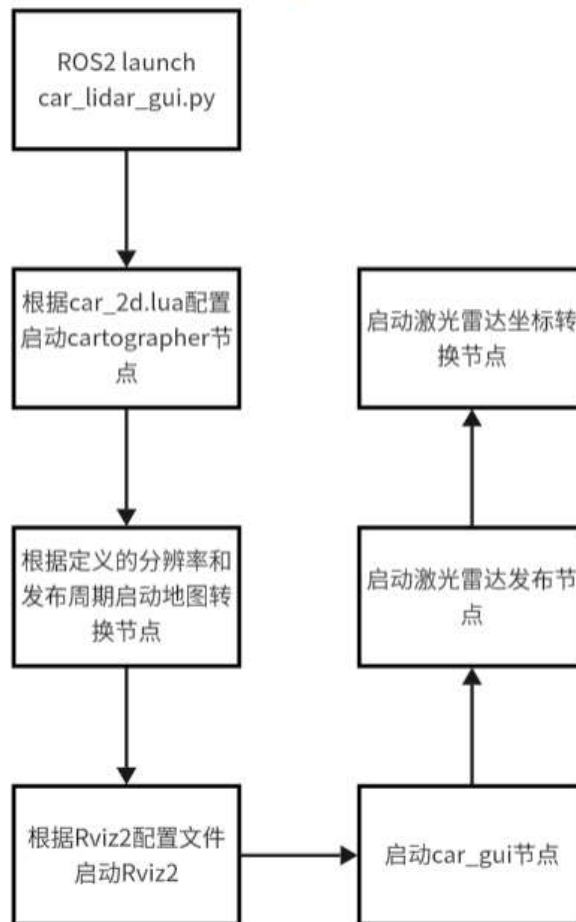
car_lidar_gui.py 代码分析:

主要功能介绍:

这段代码是一个 ROS2 launch 文件，其主要功能是启动了一系列节点以实现多项任务：首先，根据配置文件 car_2d.lua 启动了 Cartographer 节点，用于地图构建；然后，启动了 Cartographer 的栅格地图转换节点，将地图转换为占用栅格地图以供显示；同时，还启动了 RViz2 节点，用于地图和机器人当前位置的可视化；接着，启动了一个 car_gui 节点，用于显示人机交互界面并实现车辆控制；然后，启动了激光雷达节点来获取雷达数据；最后，启动了一个激光雷达坐标转换节点，用于定义 base_link 和 base_laser 之间的静态变换。

主要流程:

根据 car_2d.lua 配置启动 cartographer 节点；根据定义的分辨率和发布周期启动地图转换节点；根据 Rviz2 配置文件启动 Rviz2；启动 car_gui 节点；启动激光雷达发布节点；启动激光雷达坐标转换节点。



car2d.lua 的主要配置：

`map_frame = "map"`: 设置地图框架为 "map", 表示地图在 ROS 中的框架名称为 "map"。

`tracking_frame = "base_laser"`: 设置跟踪框架为 "base_laser", 表示在 SLAM 中跟踪的参考框架。

`published_frame = "odom"`: 设置发布框架为 "odom", 表示发布的坐标帧为 "odom"。

`odom_frame = "odom"`: 设置里程计框架为 "odom", 表示里程计的坐标帧为 "odom"。

`provide_odom_frame = false`: 设置不提供推测的里程计数据, 即不使用 SLAM 的推测里程计。

`publish_frame_projected_to_2d = true`: 设置仅发布 2D 位姿地图。

`use_odometry = true`: 使用里程计数据辅助建图。

`num_laser_scans = 1`: 使用一个激光雷达。

`num_multi_echo_laser_scans = 0`: 不使用多波雷达。

`lookup_transform_timeout_sec = 0.2`: 查找变换超时时间为 0.2 秒。

`submap_publish_period_sec = 0.3`: 发布子地图的周期为 0.3 秒。

`pose_publish_period_sec = 5e-3`: 发布位姿的周期为 5 毫秒。

`trajectory_publish_period_sec = 30e-3`: 发布轨迹的周期为 30 毫秒。

`TRAJECTORY_BUILDER_2D.min_range = 0.10`: 设置激光雷达的最小扫描距离为 0.10 米。

`TRAJECTORY_BUILDER_2D.max_range = 8`: 设置激光雷达的最大扫描距离为 8 米。

`TRAJECTORY_BUILDER_2D.missing_data_ray_length = 1.`: 设置传感器数据超出有效范围的最大值为 1。

`TRAJECTORY_BUILDER_2D.use_imu_data = false`: 不使用 IMU 数据。

`TRAJECTORY_BUILDER_2D.use_online_correlative_scan_matching = true`: 使用实时回环检测来进行前端的扫描匹配。

`TRAJECTORY_BUILDER_2D.motion_filter.max_angle_radians = math.rad(0.1)`: 设置对运动的敏感度。

`TRAJECTORY_BUILDER_2D.ceres_scan_matcher.translation_weight = 100`: 设置对移动的敏感度。

`TRAJECTORY_BUILDER_2D.ceres_scan_matcher.rotation_weight = 0.001`: 设置对转向的敏感度。

`POSE_GRAPH.constraint_builder.min_score = 0.65`: 设置 Fast csm 的最低分数。

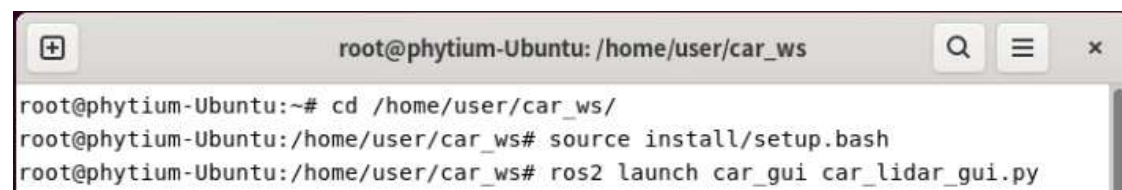
`POSE_GRAPH.constraint_builder.global_localization_min_score = 0.7`: 设置全局定位的最小分数。

4. 系统测试与分析

4.1 需求功能实现验证

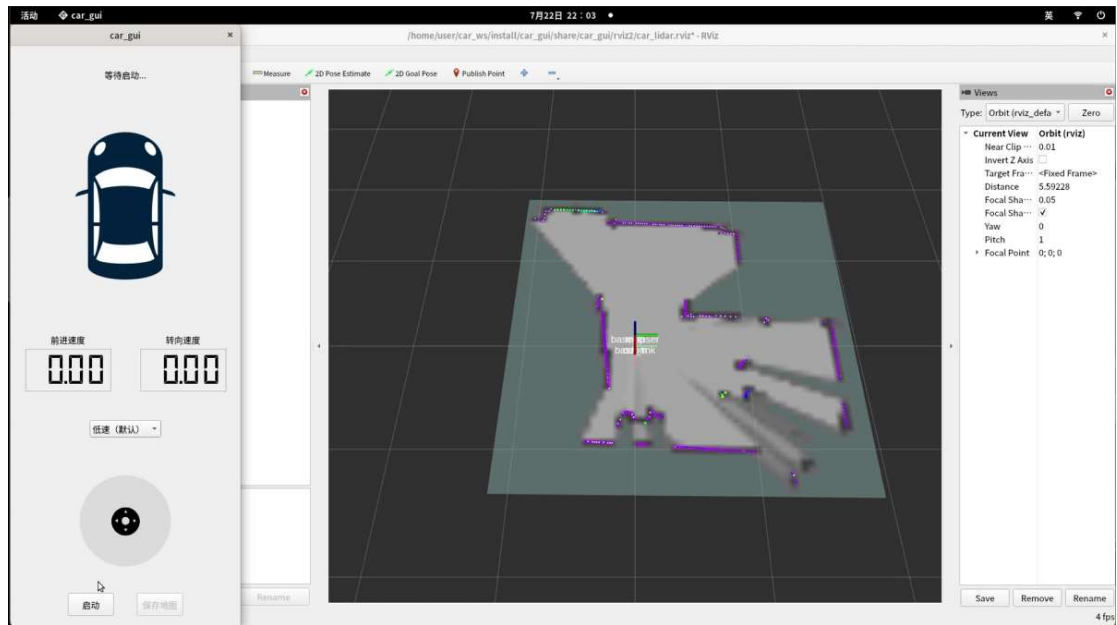
4.1.1 项目启动

将利用 `rpmsg_echo_os.c` 和 `car.c` 编译得到从核 `openamp_core0.elf` 可执行文件放入飞腾派中的 `/usr/lib/firmware` 中后，打开 `/home/user/car_ws`，输入 `source install/setup.bash` 加载 `ros2` 环境，之后通过 `ros2 launch car_gui car_lidar_gui.py` 即可启动全部程序。



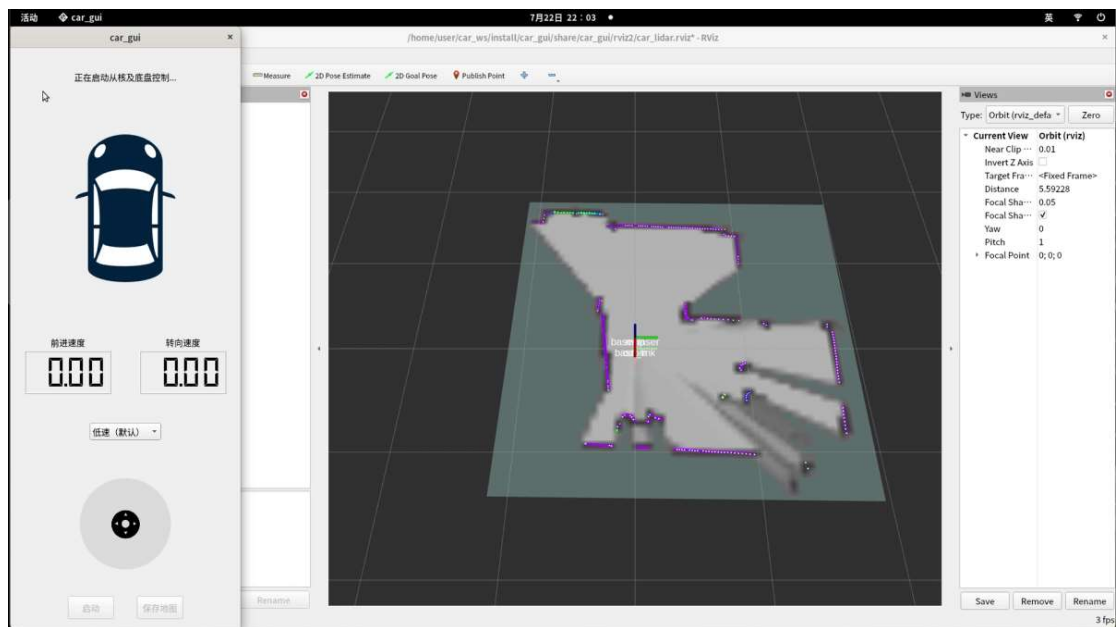
```
root@phytium-Ubuntu: /home/user/car_ws
root@phytium-Ubuntu:~# cd /home/user/car_ws/
root@phytium-Ubuntu:/home/user/car_ws# source install/setup.bash
root@phytium-Ubuntu:/home/user/car_ws# ros2 launch car_gui car_lidar_gui.py
```

启动完成后，即可看到 `Mainwindow` 提供的人机交互界面以及 `Rviz2` 的地图及坐标显示界面

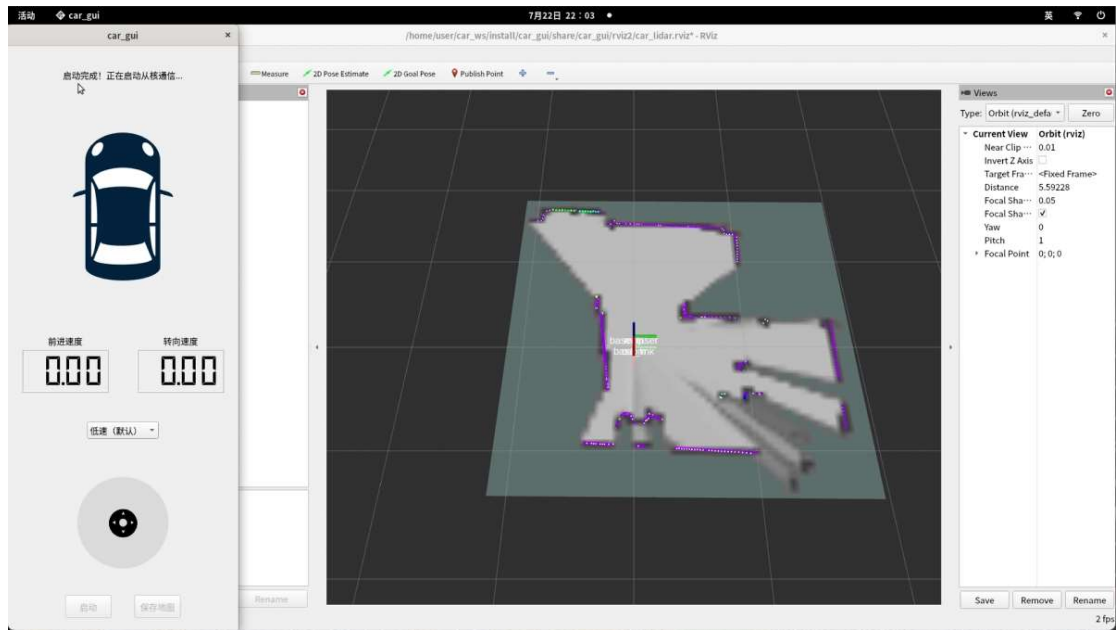


4.1.2 小车控制功能

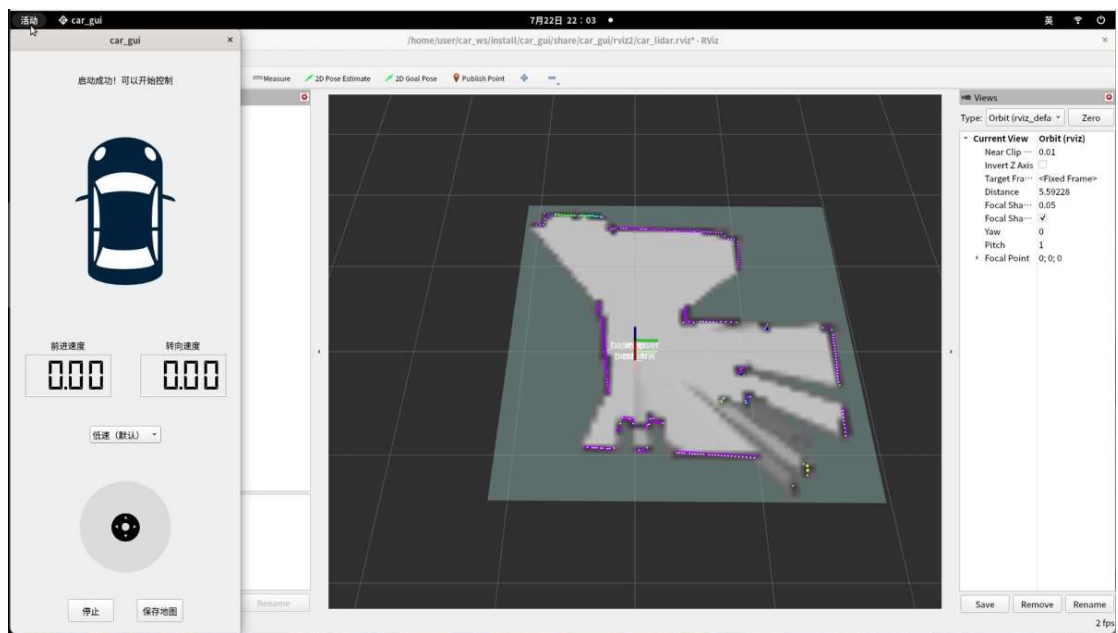
点击启动按钮，首先启动从核程序。



之后开启 RPMsg 通道并加载 RPMsg 驱动。

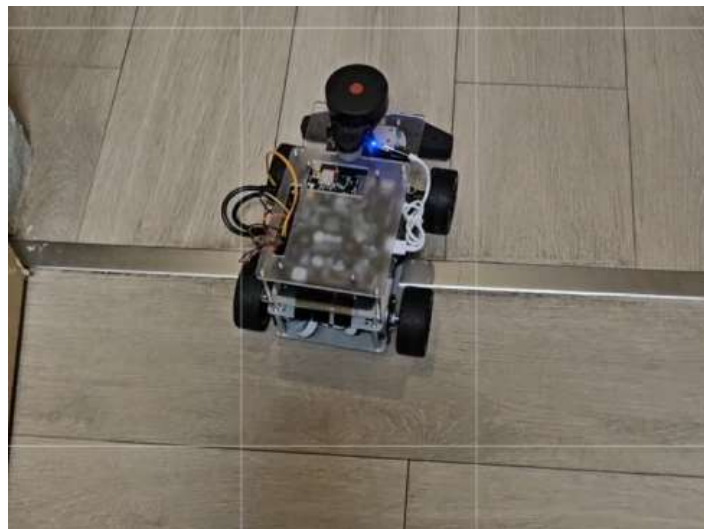
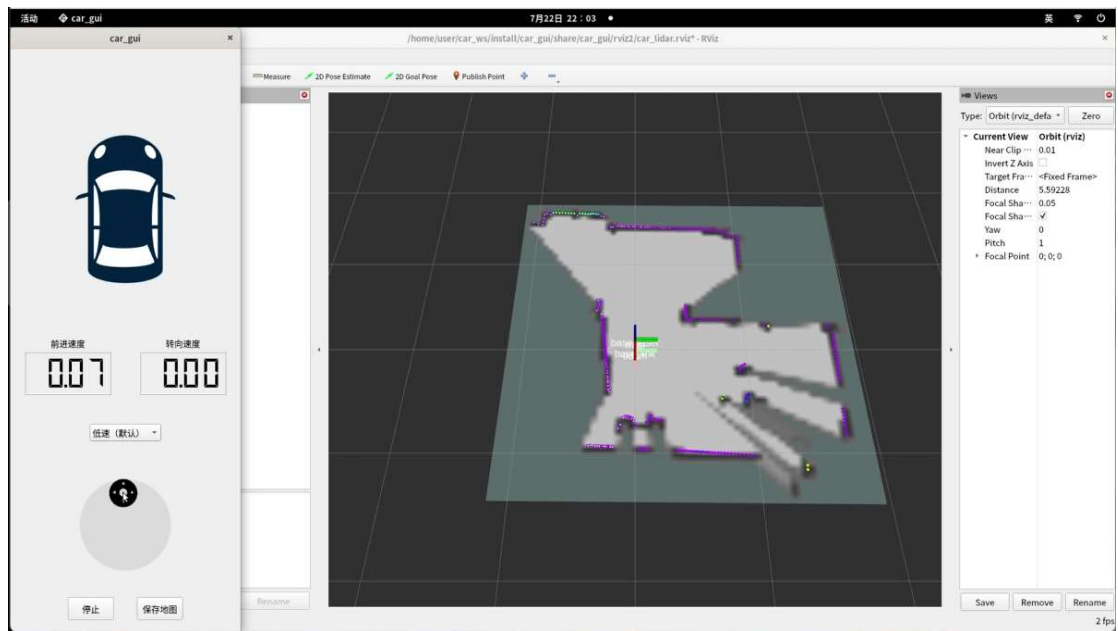


启动完成，提示可以开始控制小车。

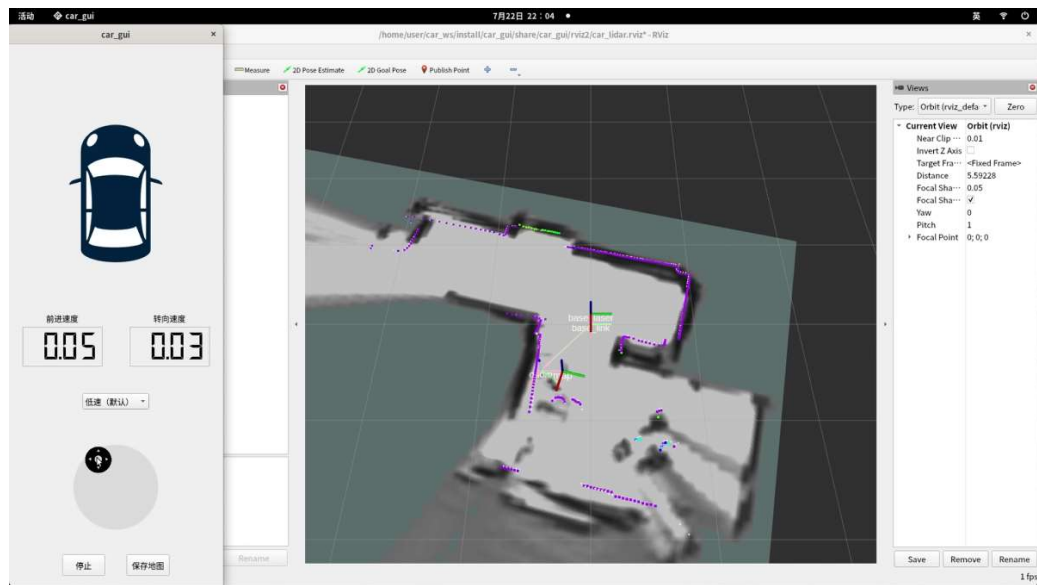




拖动摇杆向前，小车即可向前移动。

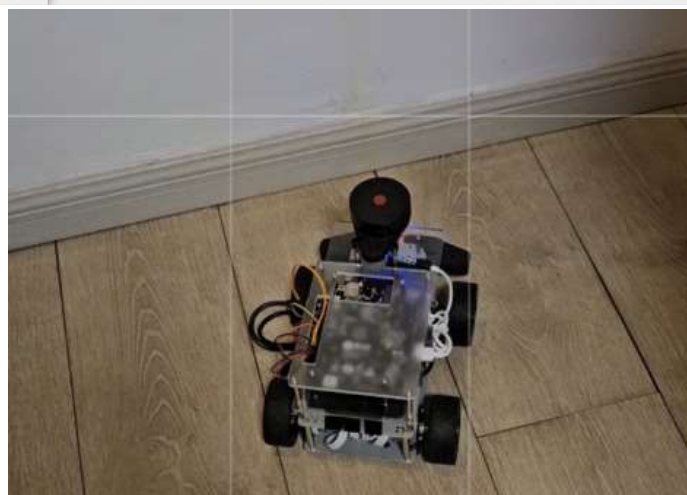
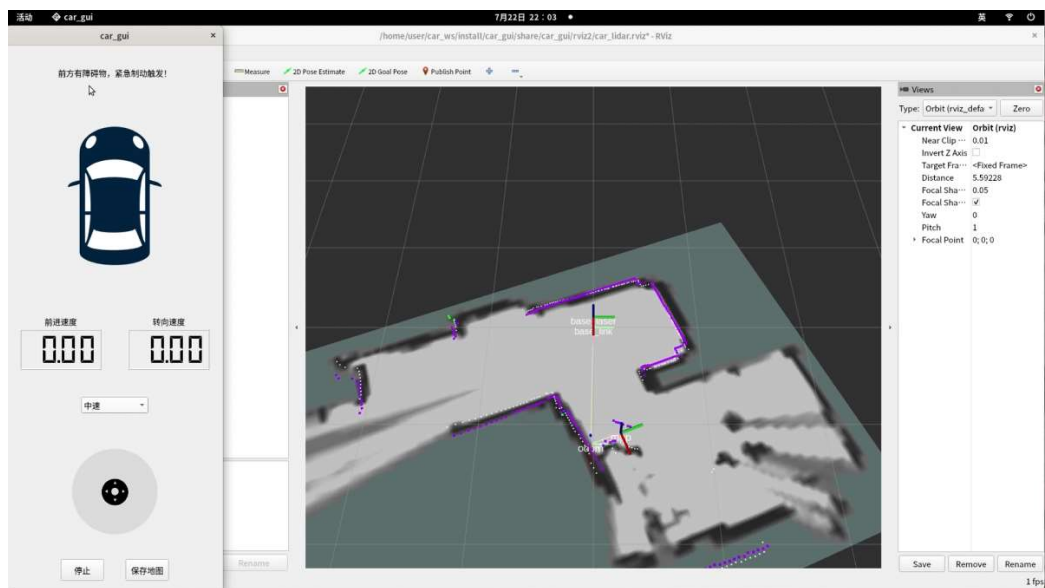


将摇杆向左前方拖动后，小车向左前方行驶同时在人机交互界面的小车向左旋转



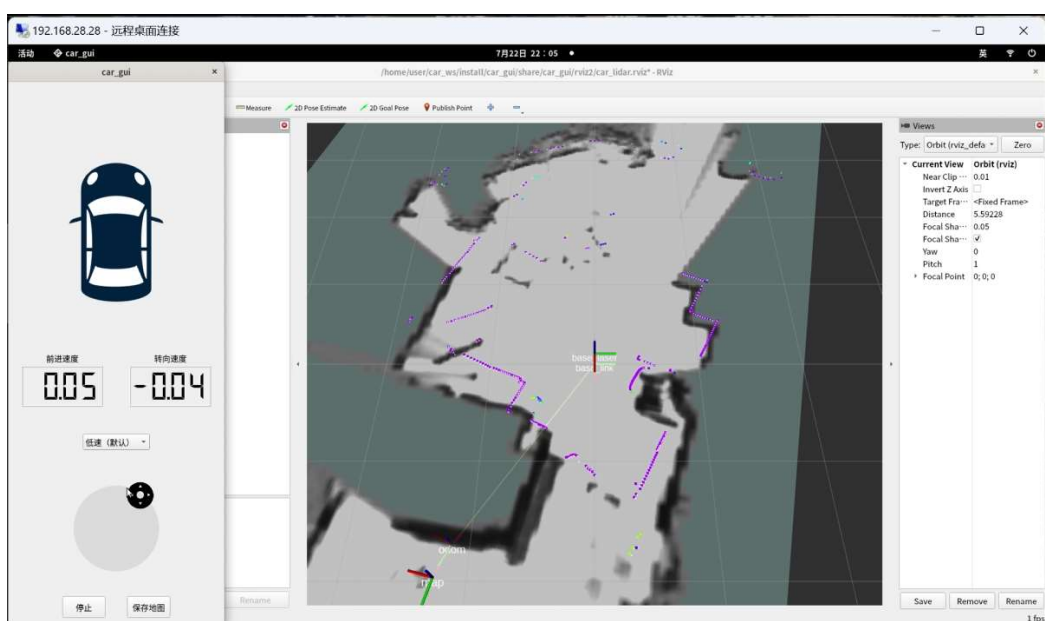
4.1.3 前紧急制动功能

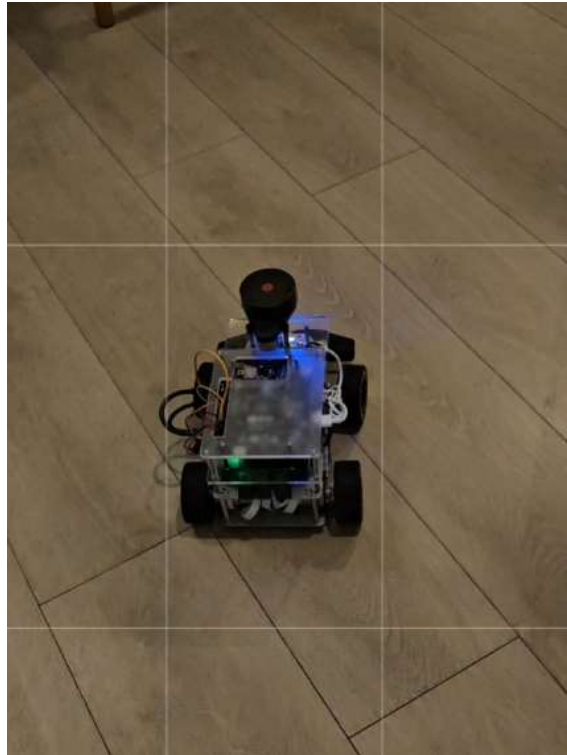
移动小车到靠近墙的位置后，小车自动停止，并在人机交互界面的提示条上显示触发紧急制动



4.1.4 建图功能

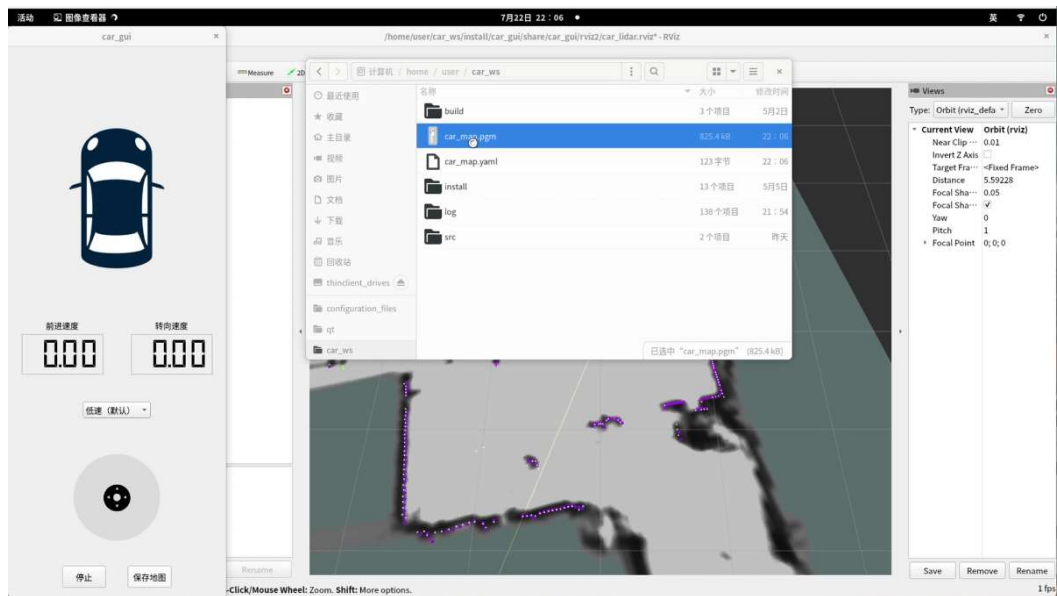
持续移动小车，地图可以连续建立。

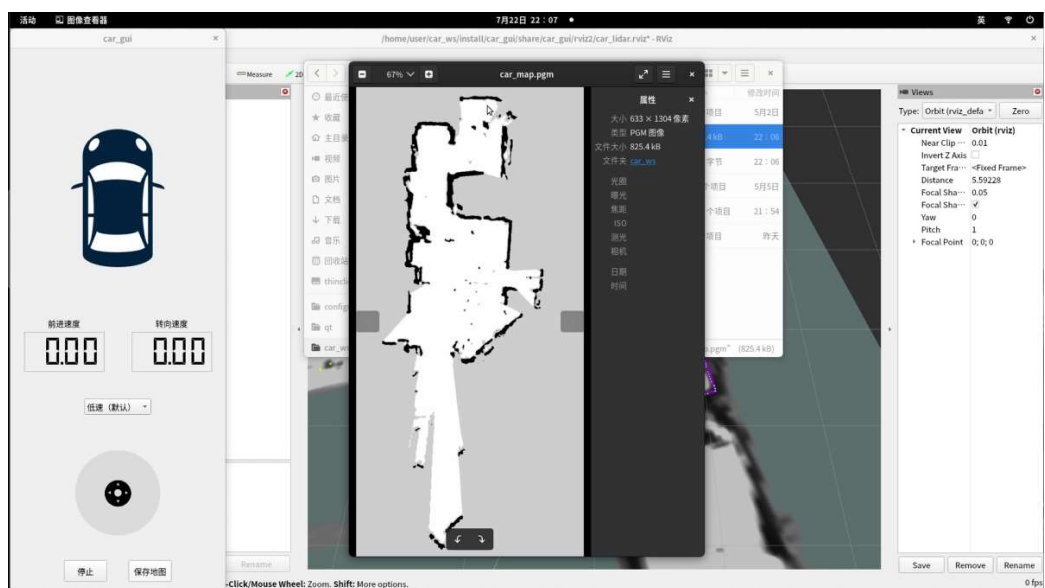




4.1.5 保存地图功能

点击保存地图按钮，就能将小车所建立的地图保存到指定位置。





4.2 性能达标验证

融合里程计与激光雷达信息后，小车可以在复杂环境下连续建图，同时利用回环检测，精确度有一定的保障，性能达到预期要求。