QEADynamicPlanning

QEA小车最终测试（动态路径规划）

项目简介

本项目旨在利用激光雷达实现在固定矩形区域内扫描和检测方形垃圾桶作为障碍物，以及圆形垃圾桶作为目标物。通过使用梯度上升法自动规划最优路径，从机器的当前位置到达目标圆形垃圾桶，并且能够自动避开方形垃圾桶。在测试过程中，还会进行人为随机改动障碍物的位置。

PID算法编写

由于我们项目中会用到很多次PID算法，所以我们将PID算法写成库的形式供其他模块调用。

定义PID句柄结构体:

```
enum PIDType
{
    PIDType_Pos = 1, //位置式
    PIDType_Inc = 2, //增量式
};

volatile struct PIDHanldeDef
{
    double kp; //p比例系数
    double ki; //i比例系数
    double kd; //d比例系数
    double value; //实际测量值
    double target; //目标值
    double error; //误差
    double totalError; //总误差
    double lastError; //上一次误差
    double lastError2; //上上一次误差
    double minPWM; //最小输出PWM
    double maxPWM; //最大输出PWM
    double pwm; //最终输出PWM
    enum PIDType pidType; //PID算法类型
};
```

创建PID句柄

```
void CreatePID(volatile struct PIDHanldeDef* def, enum PIDType pidType,
double kp, double ki, double kd, double minPWM, double maxPWM)
{
    def->kp = kp;
    def->ki = ki;
    def->kd = kd;
    def->value = 0;
    def->target = 0;
    def->error = 0;
    def->totalError = 0;
    def->lastError = 0;
    def->lastError2 = 0;
    def->minPWM = minPWM;
    def->maxPWM = maxPWM;
    def->pwm = 0;
    def->pidType = pidType;
}
```

外部程序通过调用PIDTick完成一次PID计算

```
double PIDTick(volatile struct PIDHanldeDef* def, double newValue)
{
    def->value = newValue;
    switch (def->pidType) {
        case PIDType_Pos: //位置式PID
            PosTick(def);
            break;
        case PIDType_Inc: //增量式PID
            IncTick(def);
            break;
    }
    return def->pwm;
}
```

位置式PID实现原理

```
void PosTick(volatile struct PIDHanldeDef* def)
{
    def->lastError = def->error;
    def->error = def->target - def->value;
    def->totalError += def->error;

    double pwm = def->kp * def->error +
                 def->ki * def->totalError +
                 def->kd * (def->error - def->lastError);
    def->pwm += pwm;
    if (def->pwm > def->maxPWM) def->pwm = def->maxPWM;
    if (def->pwm < def->minPWM) def->pwm = def->minPWM;
}
```

增量式PID实现原理

```
void IncTick(volatile struct PIDHanldeDef* def)
{
    def->lastError2 = def->lastError;
    def->lastError = def->error;
    def->error = def->target - def->value;

    double pwm = def->kp * (def->error - def->lastError) +
                 def->ki * def->error +
                 def->kd * (def->error - 2 * def->lastError + def-
>lastError2);
    def->pwm += pwm;
    if (def->pwm > def->maxPWM) def->pwm = def->maxPWM;
    if (def->pwm < def->minPWM) def->pwm = def->minPWM;
}
```

电机速度PID控制

我们通过增量式PID算法实现小车电机速度的闭环控制。

电机PID控制结构体：

```
volatile struct WheelPWM
{
    volatile struct PIDHanldeDef pidHanldeDef; //pid算法句柄
    double speed; //电机速度
    double dis; //路程
    double pwm; //最终输出pwm
};
```

初始化电机PID控制

```
void Wheel_PID_Init()
{
    CreatePID(&leftPWM.pidHanldeDef, PIDType_Inc, 960, 240, 0, -1200,
1200);
    CreatePID(&rightPWM.pidHanldeDef, PIDType_Inc, 960, 240, 0, -1200,
1200);
}
```

完成一次电机PID计算并更新PWM

```
void Wheel_PID_Tick()
{
    //获取左右轮速度
    leftPWM.speed = (short) __HAL_TIM_GET_COUNTER(&htim8);
    __HAL_TIM_SET_COUNTER(&htim8, 0);
    rightPWM.speed = (short) __HAL_TIM_GET_COUNTER(&htim4);
    __HAL_TIM_SET_COUNTER(&htim4, 0);
```

```
        //通过速度积分计算行驶过的路程
        leftPWM.dis += leftPWM.speed;
        rightPWM.dis += rightPWM.speed;

        //计算一次pid
        leftPWM.pwm = PIDTick(&leftPWM.pidHanldeDef, leftPWM.speed);
        rightPWM.pwm = PIDTick(&rightPWM.pidHanldeDef, rightPWM.speed);

        //更新左轮pwm
        if (leftPWM.pwm > 0) {
            __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_3, leftPWM.pwm);
            __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_4, 0);
        } else if (leftPWM.pwm < 0) {
            __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_3, 0);
            __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_4, -leftPWM.pwm);
        } else {
            __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_3, 0);
            __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_4, 0);
        }

        //更新右轮pwm
        if (rightPWM.pwm > 0) {
            __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_1, rightPWM.pwm);
            __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_2, 0);
        } else if (rightPWM.pwm < 0) {
            __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_1, 0);
            __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_2, -rightPWM.pwm);
        } else {
            __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_1, 0);
            __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_2, 0);
        }
    }
```

启动电机PID速度控制

```

        //启动编码器
        HAL_TIM_Encoder_Start(&htim4, TIM_CHANNEL_1 | TIM_CHANNEL_2);
        HAL_TIM_Encoder_Start(&htim8, TIM_CHANNEL_1 | TIM_CHANNEL_2);

        //开启pwm输出 驱动频率30kHz 0-1200
        HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_1);
        HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_2);
        HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_3);
        HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_4);

        //开启编码器速度采集 1kHz
        HAL_TIM_Base_Start_IT(&htim7);
```

在定时器回调函数中调用PID控制算法：

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  /* USER CODE BEGIN Callback 0 */

  /* USER CODE END Callback 0 */
  if (htim->Instance == TIM6) {
    HAL_IncTick();
  }
  /* USER CODE BEGIN Callback 1 */

  //进行一次pid计算
  if (htim->Instance == TIM7) {
      Wheel_PID_Tick();
  }

  /* USER CODE END Callback 1 */
}
```

读取解析IMU数据

我们通过STM32CubeMX配置IMU串口和FreeRTOS，在DMA中断回调函数中使用双缓冲接收串口数据，并通过任务通知将接收到的数据传递给IMUDecoder任务来进行解析。 IMUDecoder任务函数：

```
void IMUDecoder(void *argument)
{
  /* USER CODE BEGIN IMUDecoder */
    imuTaskHandle = xTaskGetCurrentTaskHandle(); //将当前任务句柄送给imu
    /* Infinite loop */
    for(uint32_t cnt = 0; ; ++cnt)
    {
        if (HAL_UART_STATE_READY == HAL_UART_GetState(&huart2)) {
            HAL_UART_Receive_DMA(&huart2, imuRecvBuff, 128); //开启imu串口通
信
        }

        uint32_t value = 0;
        if (pdTRUE == xTaskNotifyWait(0, 0xffffffff, &value,
pdMS_TO_TICKS(100)) && value == 0x12345678) {
            while (!DecodeIMUPackage()) {} //解析imu数据包
        }
        osDelay(1);
    }

  /* USER CODE END IMUDecoder */
}
```

通过串口传输完一半和传输完回调函数实现双缓冲。
DMA中断回调函数：

```
void IMU_RxHalfCpltCallback()
{
    imuPackage = imuRecvBuff + 0;
    //通知解析imu任务
    BaseType_t flag = 0;
    xTaskNotifyFromISR(imuTaskHandle, 0x12345678, eSetValueWithOverwrite,
&flag);
    portYIELD_FROM_ISR(flag);
}

void IMU_RxCpltCallback()
{
    imuPackage = imuRecvBuff + 64;
    //通知解析imu任务
    BaseType_t flag = 0;
    xTaskNotifyFromISR(imuTaskHandle, 0x12345678, eSetValueWithOverwrite,
&flag);
    portYIELD_FROM_ISR(flag);
}
```

IMU解析函数：

```
int DecodeIMUPackage()
{
    if (imuPackage) {
        if (imuBuffOffset + 64 >= 1024) imuBuffOffset = 0;
        memcpy(imuBuff + imuBuffOffset, imuPackage, 64);
        imuBuffOffset += 64;
        imuPackage = 0;
    }
    //定位帧头
    int packageStart = 0;
    int flag = 0;
    for (; imuBuffOffset > 5 && packageStart < imuBuffOffset - 4;
++packageStart) {
        uint8_t crc8 = CRC8_Table(imuBuff + packageStart, 4);

        if (imuBuff[packageStart] == 0xFC && crc8 == imuBuff[packageStart
+ 4]) {
            flag = 1;
            break;
        }
    }
    //将帧头移动到起始位置
    if (flag) {
        memcpy(imuBuff, imuBuff + packageStart, imuBuffOffset -
packageStart);
        imuBuffOffset -= packageStart;
    }

    int flag2 = 0;
    //判断帧头
    uint8_t crc8 = CRC8_Table(imuBuff, 4);
    if (imuBuff[0] == 0xFC && crc8 == imuBuff[4]) { //是帧头
        int cmd = imuBuff[1]; //指令类别
        int dataLen = imuBuff[2]; //数据长度

        //判断是否接收完整个数据
        if (imuBuffOffset >= 5 + 2 + dataLen + 1) { //5->帧头 2->crc16 1->帧尾标记
            uint16_t crc16FromBuff = (imuBuff[5] << 8) | imuBuff[6];
            uint16_t crc16 = CRC16_Table(imuBuff + 7, dataLen);

            if (crc16 == crc16FromBuff) {
                HandleIMUPackage(cmd, imuBuff + 7, dataLen);
                flag2 = 1;
            }

            int frameLen = 5 + 2 + dataLen + 1;
            memcpy(imuBuff, imuBuff + frameLen, imuBuffOffset - frameLen);
            imuBuffOffset -= frameLen;
        }
    }
    return flag2;
}
```

将IMU传感器数据发送给上位机

```
char message[128] = {0};
snprintf(message, 128, "[imu]heading:%.4f pitch:%.4f roll:%.4f\r\n",
robotIMU.heading, robotIMU.pitch, robotIMU.roll);
HAL_UART_Transmit(&huart1, message, strlen(message), 100);
```

上位机接收到的数据示例：

```
[imu]heading:5.6123 pitch:0.0045 roll:0.0247
[imu]heading:5.6419 pitch:0.0047 roll:0.0246
[imu]heading:5.6761 pitch:0.0048 roll:0.0245
[imu]heading:5.7051 pitch:0.0049 roll:0.0242
[imu]heading:5.7284 pitch:0.0051 roll:0.0239
```

小车运动位置PID控制

光有速度控制还不行，我们还需要在速度PID控制的基础上实现小车运动的位置闭环控制，通过位置式PID算法实现小车能够从当前位置以指定速度和距离移动，并且通过IMU返回的角度数据，实现小车转弯闭环控制。

定义机器人运动

```
volatile struct RobotMotion
{
    double VL; //左轮速度
    double VR; //右轮速度
};

extern volatile struct RobotMotion robotMotion;

void ClearSpeed();
void CommitSpeed();
void MoveForward(double speed);
void MoveForwardWithDis(double speed, double dis);
void MoveBackward(double speed);
void MoveBackwardWithDis(double speed, double dis);
void SpinLeft(double speed);
void SpinRight(double speed);
void SpinTo(double speed, double radian);
```

清除左右轮速度

```
void ClearSpeed()
{
    robotMotion.VL = 0;
    robotMotion.VR = 0;
}
```

提交左右轮速度，会实际更新到小车上

```
void CommitSpeed()
{
    leftPWM.pidHanldeDef.target = robotMotion.VL;
    rightPWM.pidHanldeDef.target = -robotMotion.VR;
}
```

向前移动，不会实际更新到小车上

```
void MoveForward(double speed)
{
    robotMotion.VL += speed;
    robotMotion.VR += speed;
}
```

向左转，不会实际更新到小车上

```
void SpinLeft(double speed)
{
    robotMotion.VL -= speed;
    robotMotion.VR += speed;
}
```

向前移动一定距离, 调用此函数会一直阻塞到移动结束

```
void MoveForwardWithDis(double speed, double dis)
{
    volatile struct PIDHanldeDef defL; //定义pid控制句柄
    volatile struct PIDHanldeDef defR; //定义pid控制句柄
    CreatePID(&defL, PIDType_Pos, 0.05, 0, 2, -speed, speed); //创建句柄
    CreatePID(&defR, PIDType_Pos, 0.05, 0, 2, -speed, speed); //创建句柄
    leftPWM.dis = 0; //清除距离
    rightPWM.dis = 0; //清除距离
    defL.target = dis; //目标距离
    defR.target = dis; //目标距离
    while (1) {
        if (fabs(leftPWM.dis - dis) < 10 && fabs(rightPWM.dis - dis) < 10)
break; //左右轮同时到达目标位置后才break
        double retVL = PIDTick(&defL, leftPWM.dis);
        double retVR = PIDTick(&defR, rightPWM.dis);
        ClearSpeed();
        robotMotion.VL += retVL;
        robotMotion.VR += retVR;
        CommitSpeed();
        osDelay(1);
    }
    //停止运动
    ClearSpeed();
    CommitSpeed();
}
```

实现小车转弯到指定角度, 调用此函数会一直阻塞到移动结束

```
void SpinTo(double speed, double radian)
{
    while (1) {
        double dr = fabs(radian - robotIMU.heading);
        if (dr < 0.001) break;
        double maxSpeed = speed;
        double speed = dr * 140;
        if (speed > maxSpeed) speed = maxSpeed;
        if (radian - robotIMU.heading > 0) {
            if (radian - robotIMU.heading > PI) {
                ClearSpeed();
                SpinRight(speed);
                CommitSpeed();
            } else {
                ClearSpeed();
                SpinLeft(speed);
                CommitSpeed();
            }
        } else {
            if (radian - robotIMU.heading > -PI) {
                ClearSpeed();
                SpinRight(speed);
                CommitSpeed();
            } else {
                ClearSpeed();
                SpinLeft(speed);
                CommitSpeed();
            }
        }
    }
    ClearSpeed();
    CommitSpeed();
}
```

实现上位机控制机器人移动

下位机定义接收数据缓冲区、消息缓冲区句柄、串口回调函数

```
extern uint8_t robotRecvBuff[1024];
extern int robotRecvOffset;
extern MessageBufferHandle_t messageBufferHandle;
void Robot_RxCpltCallback();
```

下位机接收上位机的串口接收完成回调函数

```
void Robot_RxCpltCallback()
{
    if (robotRecvBuff[robotRecvOffset] == '\n') {
        robotRecvBuff[robotRecvOffset + 1] = 0;
        BaseType_t flag;
        xMessageBufferSendFromISR(messageBufferHandle, robotRecvBuff,
robotRecvOffset + 1, &flag);
        portYIELD_FROM_ISR(flag);
        robotRecvOffset = 0;
    } else {
        robotRecvOffset++;
        if (robotRecvOffset >= 1024) robotRecvOffset = 0;
    }
    HAL_UART_Receive_IT(&huart1, robotRecvBuff + robotRecvOffset, 1);
}
```

下位机新建RobotController任务函数接收并处理上位机的指令

```
void RobotController(void *argument)
{
  /* USER CODE BEGIN RobotController */
  /* Infinite loop */
    osDelay(1000);

    HAL_UART_Receive_IT(&huart1, robotRecvBuff + robotRecvOffset, 1);
    /* Infinite loop */
    for(;;)
    {
        char message[128] = {0};
        xMessageBufferReceive(messageBufferHandle, message, 128,
portMAX_DELAY);

        if (startsWith(message, "[forward]")) {
            double speed, dis;
            sscanf(message, "[forward]speed=%lf dis=%lf", &speed, &dis);
            MoveForwardWithDis(speed, dis);

            //移动结束发送OK
            portENTER_CRITICAL();
            char msgOK[] = "[motion]OK\r\n";
            xSemaphoreTake(debugUartMutexHandle, portMAX_DELAY); //获取串口
调试资源
            HAL_UART_Transmit(&huart1, msgOK, strlen(msgOK), 100);
            xSemaphoreGive(debugUartMutexHandle); //释放串口调试资源
            portEXIT_CRITICAL();
        } else if (startsWith(message, "[backward]")) {
            double speed, dis;
            sscanf(message, "[backward]speed=%lf dis=%lf", &speed, &dis);
            MoveBackwardWithDis(speed, dis);

            //移动结束发送OK
            portENTER_CRITICAL();
            char msgOK[] = "[motion]OK\r\n";
            xSemaphoreTake(debugUartMutexHandle, portMAX_DELAY); //获取串口
调试资源
            HAL_UART_Transmit(&huart1, msgOK, strlen(msgOK), 100);
            xSemaphoreGive(debugUartMutexHandle); //释放串口调试资源
            portEXIT_CRITICAL();
        } else if (startsWith(message, "[spin]")) {
            double speed, radian;
            sscanf(message, "[spin]speed=%lf radian=%lf", &speed,
&radian);
            SpinTo(speed, radian);

            //移动结束发送OK
            portENTER_CRITICAL();
            char msgOK[] = "[motion]OK\r\n";
            xSemaphoreTake(debugUartMutexHandle, portMAX_DELAY); //获取串口
调试资源
```

```
                HAL_UART_Transmit(&huart1, msgOK, strlen(msgOK), 100);
                xSemaphoreGive(debugUartMutexHandle); //释放串口调试资源
                portEXIT_CRITICAL();
            }
            osDelay(1);
        }
      /* USER CODE END RobotController */
    }
```

编写上位机CMakeLists.txt文件, 由于后期会使用opencv绘制图像，所以导入了opencv库。

```
cmake_minimum_required(VERSION 3.14)
project(qea)
set(CMAKE_CXX_STANDARD 20)
find_package(OpenCV 4.5 REQUIRED)
include_directories(${OpenCV_INCLUDE_DIRS})
aux_source_directory(. SRC_LISTS)
add_executable(qea ${SRC_LISTS})
target_link_libraries(qea ${OpenCV_LIBS} -lwiringPi -pthread)
```

定义Robot

```
#define ROBOT_DEV_PATH "/dev/ttyACM0" //下位机串口路径
#define ROBOT_BAUD 115200 //下位机串口波特率

struct RobotInfo
{
    double xPos; //机器人x坐标
    double yPos; //机器人y坐标
    double roll; //翻滚角
    double pitch; //俯仰角
    double heading; //偏航角
};

extern struct RobotInfo robotInfo;
extern int robotSerial;
extern pthread_t robotThreadHandle;
extern char robotRecvBuff[1024];
extern int robotRecvOffset;

void Robot_Init();
void Robot_Start();

void Robot_MoveForward(double speed, double dis);
void Robot_MoveBackward(double speed, double dis);
void Robot_SpinTo(double speed, double radian);
```

定义Robot线程，负责接收下位机的数据并完成解析。

```
void* RobotThread(void*)
{
    robotSerial = serialOpen(ROBOT_DEV_PATH, ROBOT_BAUD);
    if ( robotSerial < 0) {
        std::cout << "cannot open robot dev!" << std::endl;
        exit(0);
        return nullptr;
    }
    while (true) {
        if (robotRecvOffset >= 1024) robotRecvOffset = 0;
        robotRecvBuff[robotRecvOffset] = serialGetchar(robotSerial);
        if (robotRecvBuff[robotRecvOffset] == '\n') {
            Handle_Robot_Message(robotRecvBuff, robotRecvOffset);
            robotRecvOffset = 0;
        } else {
            ++robotRecvOffset;
        }
    }
    return nullptr;
}
```

启动Robot线程

```
void Robot_Start()
{
    pthread_create(&robotThreadHandle, nullptr, RobotThread, nullptr);
}
```

控制下位机向前移动

```
void Robot_MoveForward(double speed, double dis)
{
    motionOK = false;
    char message[128] = {0};
    snprintf(message, 128, "[forward]speed=%.4lf dis=%.4lf\n", speed,
dis);
    std::cout << message << std::endl;
    serialPuts(robotSerial, message);
    while (!motionOK) {
        usleep(1000);
    }
}
```

控制下位机转弯

```
void Robot_SpinTo(double speed, double radian)
{
    motionOK = false;
    char message[128] = {0};
    snprintf(message, 128, "[spin]speed=%.4lf radian=%.4lf\n", speed,
radian);
    std::cout << message << std::endl;
    serialPuts(robotSerial, message);
    while (!motionOK) {
        usleep(1000);
    }
}
```

读取解析激光雷达数据

定义激光雷达数据结构

```
#define POINT_BUFF_SZ 360
#define POINT_PER_PACK 12
#define HEADER 0x54

#define LIDAR_DEV_PATH "/dev/ttyUSB0" //激光雷达设备路径
#define LIDAR_BAUD 230400 //雷达波特率

typedef struct __attribute__((packed)) {
    uint16_t distance;
    uint8_t intensity;
}LidarPointDataDef;

typedef struct __attribute__((packed)) {
    uint8_t header;
    uint8_t ver_len;
    uint16_t speed;
    uint16_t start_angle;
    LidarPointDataDef point[POINT_PER_PACK];
    uint16_t end_angle;
    uint16_t time_stamp;
    uint8_t crc8;
}LiDARFrameTypeDef;

typedef struct __attribute__((packed)) {
    uint16_t distance;
    uint8_t intensity;
    double angle;
}LidarPointData;
```

激光雷达线程，读取串口数据并解析

```
void* LidarThread(void*)
{
    int fd = serialOpen(LIDAR_DEV_PATH, LIDAR_BAUD);
    if (fd < 0) {
        std::cout << "cannot open lidar dev!" << std::endl;
        exit(0);
        return nullptr;
    }
    while (true) {
        if (lidarBuffOffset > 900) lidarBuffOffset = 0;
        //一次性读取64字节
        for (int i = 0; i < 64; ++i) {
            lidarBuff[lidarBuffOffset++] = serialGetchar(fd);
        }
        while (DecodeLIDARPackage() != 0) {}
    }
    return nullptr;
}
```

CRC8算法

```
uint8_t CalCRC8(uint8_t *p, uint8_t len)
{
    uint8_t crc = 0; uint16_t i;
    for (i = 0; i < len; i++){
        crc = CrcTable[(crc ^ *p++) & 0xff];
    }
    return crc;
}
```

解析雷达数据

```
int DecodeLIDARPackage()
{
    //定位帧头
    int packageStart = 0;
    int flag = 0;
    for (; lidarBuffOffset > 2 && packageStart < lidarBuffOffset - 1;
++packageStart) {
        if (lidarBuff[packageStart] == 0x54 && lidarBuff[packageStart + 1]
== 0x2C) { //帧头固定字节0x54 0x2C
            flag = 1;
            break;
        }
    }
    //将帧头移动到起始位置
    if (flag) {
        memcpy(lidarBuff, lidarBuff + packageStart, lidarBuffOffset -
packageStart);
```

```
        lidarBuffOffset -= packageStart;
    }
    if (lidarBuffOffset >= 47) { //有一个完整的数据包
        LiDARFrameTypeDef* liDarFrameTypeDef =
(LiDARFrameTypeDef*)lidarBuff;
        uint8_t crc8 = CalCRC8(lidarBuff, 46);
        int flag2 = 0;
        if (crc8 == liDarFrameTypeDef->crc8) { //判断crc
            flag2 = 1;
            //解析角度 (end_angle - start_angle)/(len - 1)
            double step = 0;
            if (liDarFrameTypeDef->end_angle - liDarFrameTypeDef-
>start_angle > 0) {
                step = (liDarFrameTypeDef->end_angle - liDarFrameTypeDef-
>start_angle) / 11.0;
            } else {
                step = (36000 + liDarFrameTypeDef->end_angle -
liDarFrameTypeDef->start_angle) / 11.0;
            }
            for (int i = 0; i < 12; ++i) {
                double angle = (liDarFrameTypeDef->start_angle + step * i)
/ 100.0;
                angle = fmod(angle + 360, 360);
                int distance = liDarFrameTypeDef->point[i].distance;
                uint8_t intensity = liDarFrameTypeDef->point[i].intensity;
                lidarPointData[(int)angle].distance = distance;
                lidarPointData[(int)angle].intensity = intensity;
                lidarPointData[(int)angle].angle = angle;
            }
        }
        memcpy(lidarBuff, lidarBuff + 47, lidarBuffOffset - 47);
        lidarBuffOffset -= 47;
        return flag2;
    }
    return 0;
}
```

获取激光雷达扫描结果（360个点）

```
std::vector<LidarPointData> GetPointData()
{
    std::vector<LidarPointData> vec;
    for (int i = 0; i < POINT_BUFF_SZ; ++i) {
        vec.push_back(lidarPointData[i]);
    }
    std::sort(std::begin(vec), std::end(vec), [](const auto& p1, const
auto& p2) {
        return p1.angle < p2.angle;
    });
    return vec;
}
```