重新设计编写机器人控制模式

2022年1月28日 电控组 陈卓勋

1. 背景

最终比赛我们需要模式之间的切换。以前的第一版程序是为了让大家了解机器人控制和调试架构最基础和简单的部分。但是，现场比赛需要加入遥控器设置，需要考虑把机器人在场上移动，需要考虑自动瞄准、NUC挂掉、鼠标、键盘遥控器挂掉等特殊情况。因此，重新设计了机器人控制模式，使用枚举、结构体等方式让机器人的控制模式更容易拓展。在代码中增加了更多注释，方便大家学习和扩充。

1. 主要思路

需要把各个功能模块分开。由于机器人自己有不同的总模式，底盘和云台都有多种模式，但是对于我们目前这个轴承摩擦力过大、没有电滑环和限位装置的机器人，底盘工作在某些模式下时，云台就不能工作在某些模式。例如，当底盘跟随云台时，若机器人的云台是无力（由于摩擦力太大）状态，就会导致机器人疯狂旋转——底盘想跟上云台，但是云台和底盘的相对角度是不变的。因此，需要一个新任务，根据遥控器和键盘的值改变机器人的总模式，云台和底盘任务再根据机器人总模式分别进入不同的模式，由此，可以确保机器人模式的一致性和协调性。

1. 代码实现和操作过程

一些模式设置和总模式提供给云台、底盘任务的接口

#ifndef ROBOT\_TOTAL\_MODE\_H

#define ROBOT\_TOTAL\_MODE\_H

enum RobotState\_e{

    RobotState\_e\_Powerless,      // 无力状态，当遥控器掉线时使用，也便于在场上推动机器人运动。这时，所有机构发送零电流。

    RobotState\_e\_StickyYaw,      // 固定云台方位。对于我们第一版的、云台轴承阻力过大的机器人使用。这时，云台yaw发送零电压

    RobotState\_e\_ChassisFollowGimbal,    // 底盘跟随云台，只是跟随角度。这时，云台必须是能完全控制且为绝对角度控制的。

                            // 底盘通过旋转自己减小和云台的角度差。云台受手动或者自动控制。

    RobotState\_e\_QuasiSpinner,   // 准小陀螺运动模式——用于没有电滑环但还是可以靠软件控制旋转一定圈数范围。

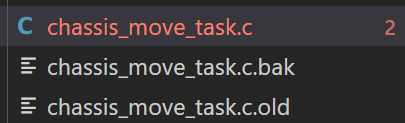
                    // 此时云台是绝对角度控制，底盘是绝对角速度控制（闭环控制轮子电机转速，但是不管实际整体转速）

    RobotState\_e\_Spinner,        // 小陀螺模式，此时云台绝对角度控制，底盘绝对角速度（同上）

};

int robotIsAuto(void);  // 对外提供的接口，返回目前机器人是否自动

enum RobotState\_e getRobotPresentMode(void);   //返回机器人当前状态



重新编写chassis\_move\_task.c，原来的代码以.old结尾作为备份。

新编写的文档里对宏定义进行了分类，使用Solidworks装配图距离测量功能对一些参数重新确定。编写大量注释。

// 底盘模式

enum ChassisMode\_e{

    ChassisMode\_e\_Powerless,       //遥控器掉线时，底盘无力

    ChassisMode\_e\_Independent,     //对于第一版本yaw与底盘卡住的，直接用独立的底盘。第一版本默认的模式。

    ChassisMode\_e\_ChassisFollowYaw,//按键C开启，跟随云台，此时用底盘的旋转减小与云台的角度差。云台必须用绝对角度控制。

    ChassisMode\_e\_QuasiSpinner,     //由键盘和遥控器控制开启。V开启（长得根陀螺一样）准小陀螺模式

    ChassisMode\_e\_Spinner,          //由键盘和遥控器开启。  V开启   由宏定义决定开启哪一种。若有电滑环，则开启真小陀螺模式。

};

// 坐标系

enum MovingAxis\_e{

    MovingAxis\_e\_GimbalAxis,

    MovingAxis\_e\_ChassisAxis,

};

//机器人控制量

struct RobotControl\_s{

    fp32 vx,vy,w;

    enum MovingAxis\_e axis;

};

//云台yaw电机相对于底盘角度结构体

struct Angle\_s{

    int16\_t initGimbalYawECD;       //开机时的ECD

    int16\_t nowGimbalYawECD;        //目前的ECD

    int32\_t rotateRounds;           //目前旋转了多少圈，用于准小陀螺模式时，防止线断掉。

    fp32    gimbalAngleFromChassis; //云台相对于最初的角度，逆时针为正方向。单位弧度。

};

//底盘电机控制

struct MotorControl\_s{

    pid\_type\_def    vpid;

    fp32            presentMotorSpeed;

    fp32            wantedMotorSpeed;

    int16\_t         giveCurrent;

};

用这些结构、枚举可以方便实现机器人模式和功能拓展。具体方法为：在枚举类型里增加新的定义。在每一个根据模式计算控制量的代码后，增加if else，实现新的模式的控制量计算。

全局变量一般设置为静态，防止与其他文件的变量冲突。通过函数接口返回指针实现变量访问。使用结构体，对变量分类。例如，与云台-底盘夹角相关的值放在了someAngle结构体变量中，以减少直接出现在全局变量区的变量数量，更好管理变量。

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*用于控制的全局变量\*\*\*\*\*\*\*\*\*\*//

static int zeroCurrentMark;    //当离线时，直接发送零电流的时候

static struct Angle\_s someAngle;   //用于记录所有的云台、底盘角度相关值

static struct MotorControl\_s driveMotor[4];

static const RC\_ctrl\_t \*rc\_p;   //遥控器位置指针，需要初始化

static const toSTM32\_t \*nuc\_p;  //NUC数据位置。未来制造全自动机器人时需要

const static fp32 motor\_speed\_pid[3] = {M3505\_MOTOR\_SPEED\_PID\_KP, M3505\_MOTOR\_SPEED\_PID\_KI, M3505\_MOTOR\_SPEED\_PID\_KD}; //PID 初始化参数

static enum ChassisMode\_e chassisState=ChassisMode\_e\_Powerless;

static struct RobotControl\_s robotTotalSpeedControl;

static int16\_t vx\_channel,vy\_channel,w\_channel; //遥控器deadband limit后输出的值

根据机器人总模式进入云台和底盘各自的模式的控制方法

static void setChassisMode(void)

{

    enum RobotState\_e robot=getRobotPresentMode();

    zeroCurrentMark=0;

    if(robot==RobotState\_e\_Powerless)

    {

        zeroCurrentMark=1;

        chassisState=ChassisMode\_e\_Powerless;

    }

    else if(robot==RobotState\_e\_StickyYaw)

        chassisState=ChassisMode\_e\_Independent;

    else if(robot==RobotState\_e\_ChassisFollowGimbal)

        chassisState=ChassisMode\_e\_ChassisFollowYaw;

    else if(robot==RobotState\_e\_Spinner)

        chassisState=ChassisMode\_e\_Spinner;

    else if(robot==RobotState\_e\_QuasiSpinner)

        chassisState=ChassisMode\_e\_QuasiSpinner;

    else

        zeroCurrentMark=1;

}

为了应对突发情况，例如遥控器掉线和NUC掉线，增加一些控制逻辑

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*掉线检测\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

        if(!NUCIsValid())

            robotAuto=0;

        if(!RCIsValid())

            robotState=RobotState\_e\_Powerless;

        //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

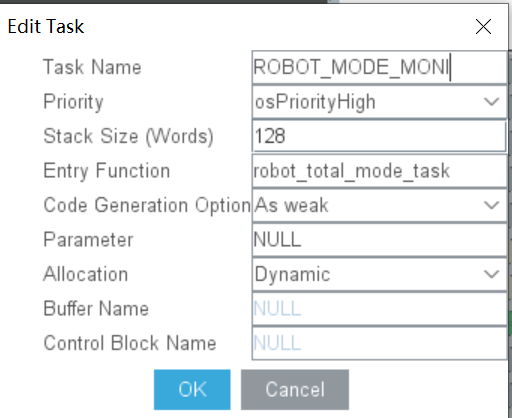
当然，在赛场上，可以通过关闭遥控器，让机器人底盘无力，从而在不关闭电源的情况下推动机器人。（当设置闭环控制电机速度为0时，由于PID控制的作用，机器人很难推动）

另外，为了方便机械结构相关改进导致的代码修改，新建RobotStructure.h,记录电滑环、轴承等重要部件的存在性，并分情况编写代码。

#define HAVE\_SLIP\_RING  // 有电滑环。默认没有。

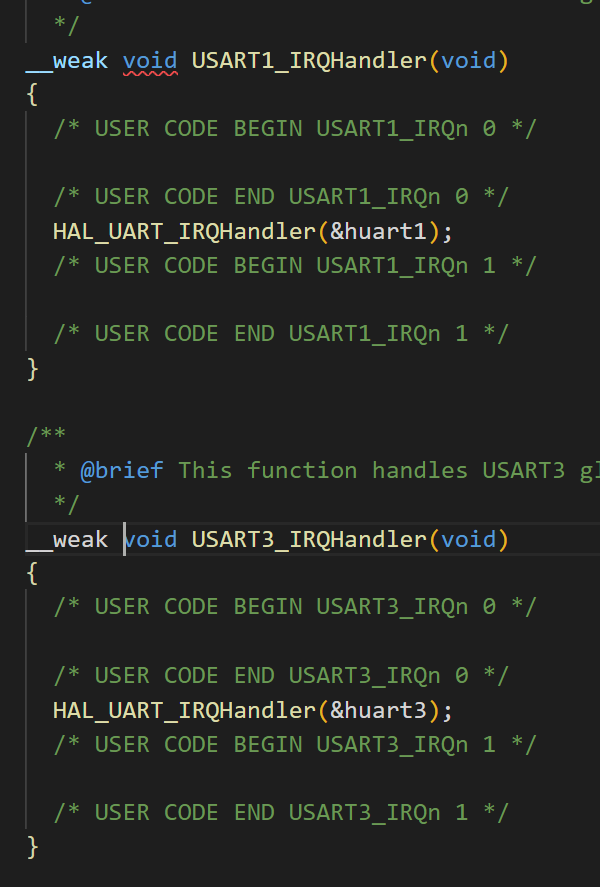
#define GOOD\_YAW\_BEARING// 有好的yaw轴轴承。默认没有。

Cubemx里设置新任务

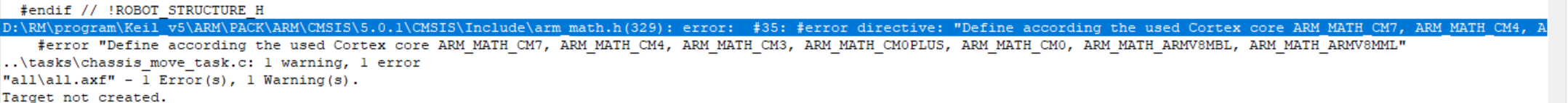


在μvision 中添加新任务的文件。

另外，还有在每次生成文件后，在这src文件夹的XX\_it那个文件两个函数前加上\_\_weak.因为我们的遥控器和NUC那里已经定义了这两个中断处理函数了

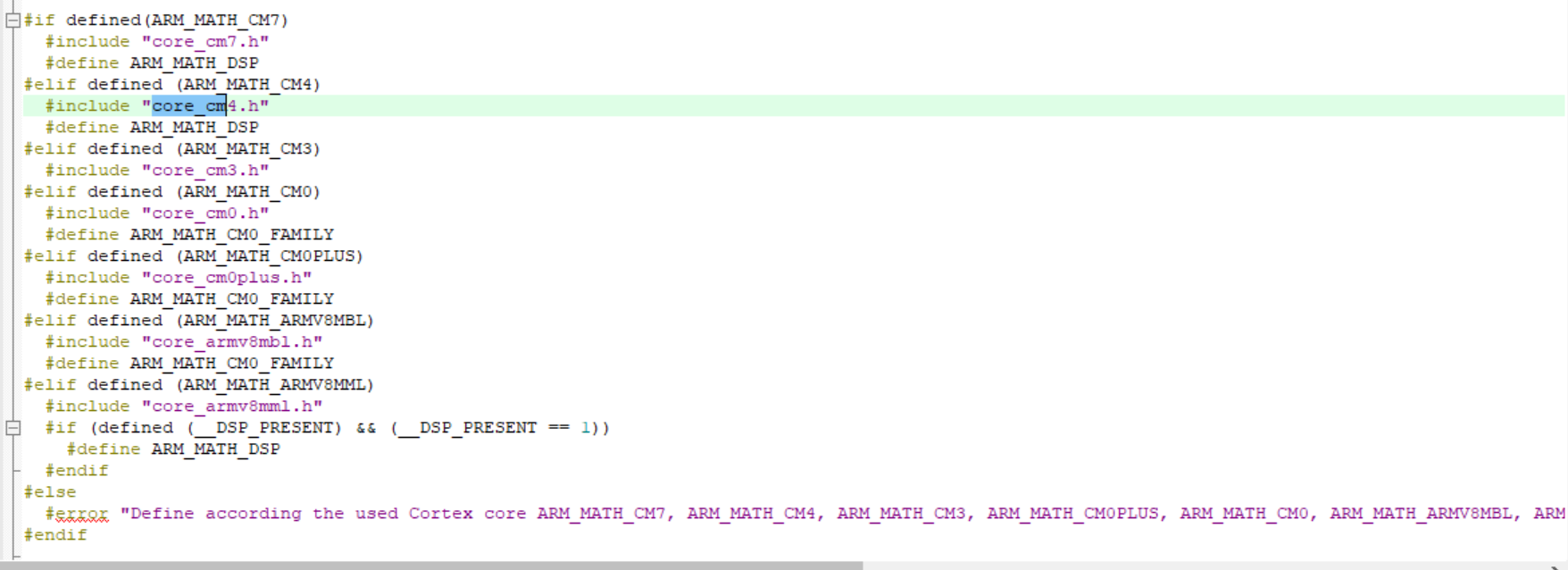


由于添加了arm\_math.h，出现了报错

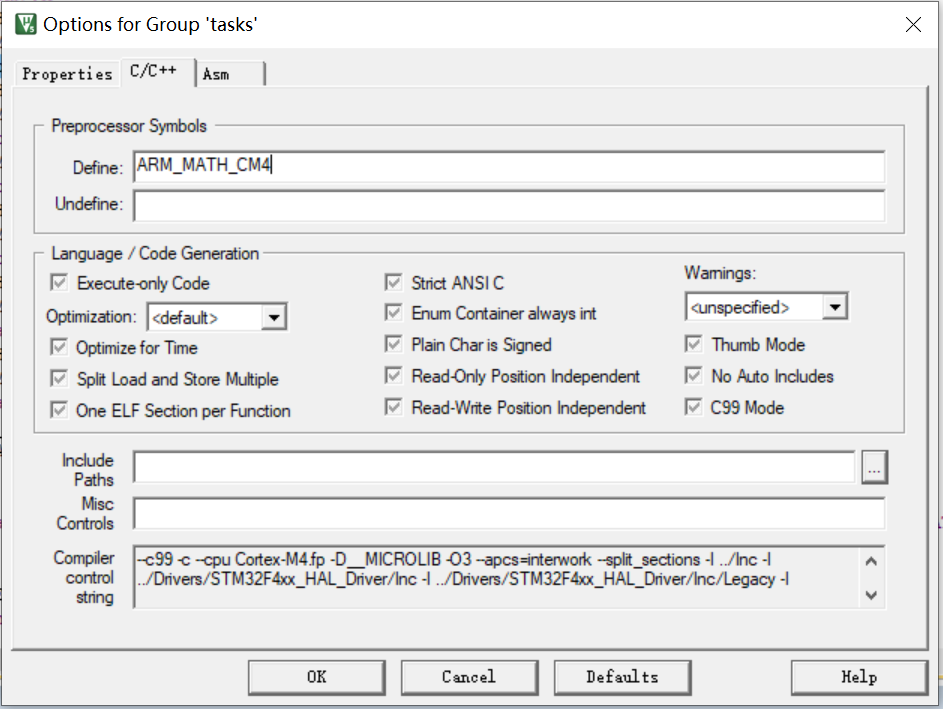


打开相应头文件

需要定义使用的芯片

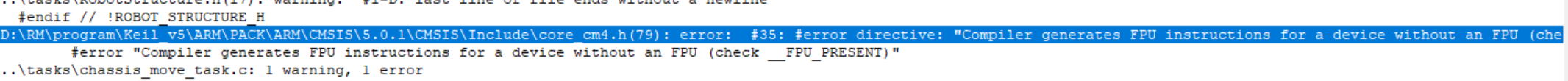


我们用的是cortex 4 CPU的stm32MCU，因此打开option，在定义里面定义

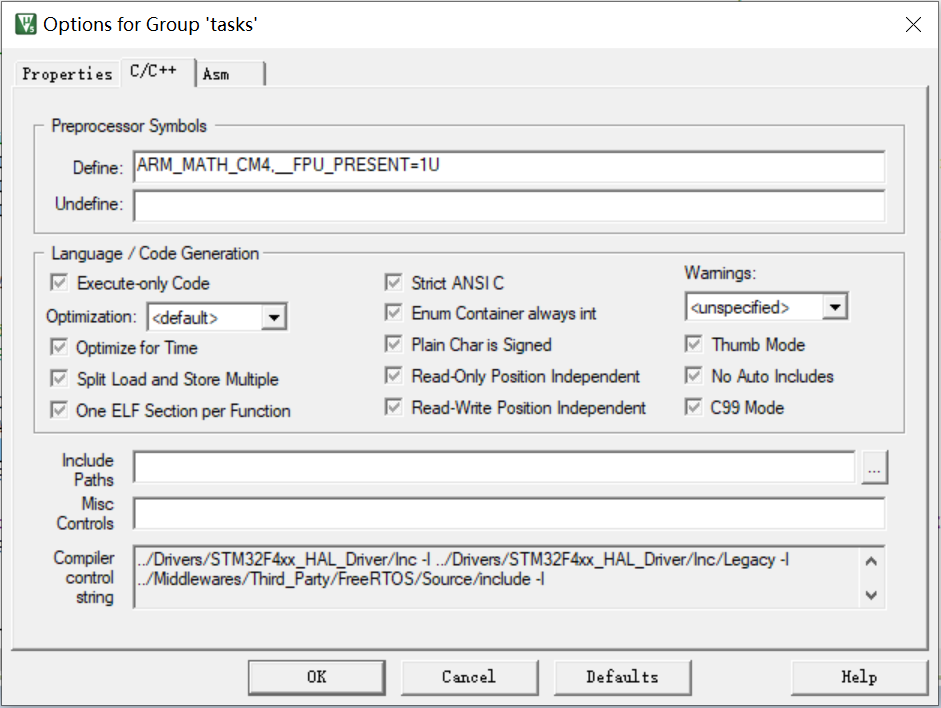


再次尝试，

新错误

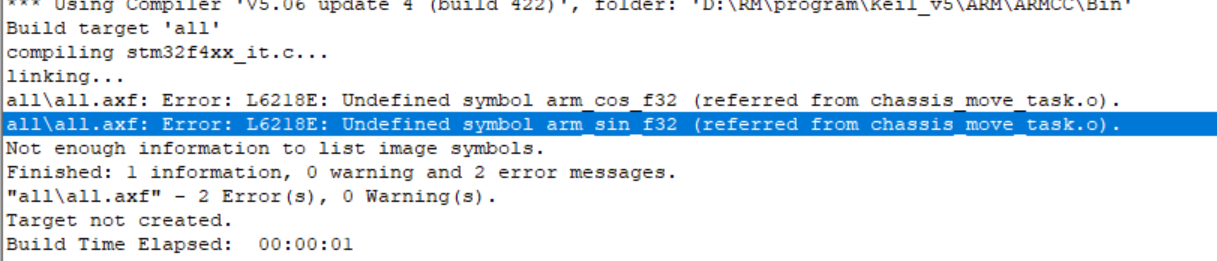


没有FPU，于是，再定义FPU



好了，再试

报错

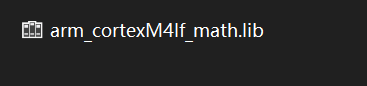


查找一番，是需要库。

工程文件里，新建文件夹

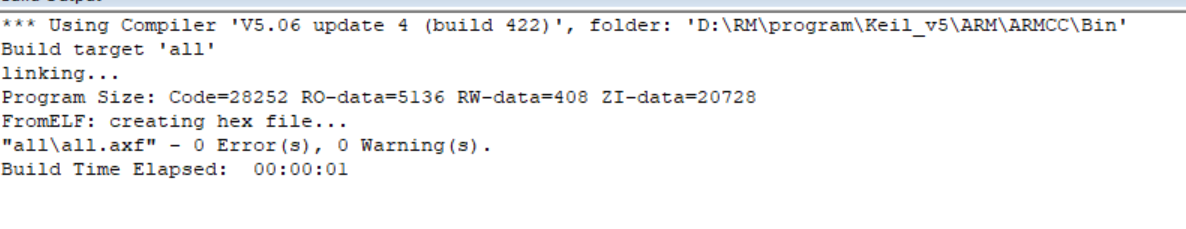


放入库



在IDE中添加库，再看。

好耶



1. 上机测试

主要是看能不能实现云台坐标系移动。

Xs，电机上出现奇怪脉冲，抖个不停，开始debug

需要一些debug的方法。重新编写新的OLED显示控制

总结一下用OLED调试所需的代码

总开关（并且在一定时间频率内刷新）

//OLED调试相关

#define OLED\_DEBUG\_TOTAL\_MODE\_ON

#ifdef OLED\_DEBUG\_TOTAL\_MODE\_ON

#define OLED\_REFRESH\_TIME 300

#endif

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*调试所用变量\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

#ifdef OLED\_DEBUG\_TOTAL\_MODE\_ON

static int time\_passed\_debug=0;

#endif

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*OLED 显示调试\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

        #ifdef OLED\_DEBUG\_TOTAL\_MODE\_ON

        time\_passed\_debug+=ROBOT\_MODE\_TASK\_TIME;

        if(time\_passed\_debug>OLED\_REFRESH\_TIME)

        {

            time\_passed\_debug=0;

            //\*\*\*\*\*\*\*\*\*输出和更新\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

            OLED\_printf(0,0,"                    ");

            OLED\_printf(1,0,"                    ");

            OLED\_printf(2,0,"                    ");

            OLED\_printf(3,0,"                    ");

            OLED\_printf(4,0,"                    ");

            OLED\_printf(0,0,"robot mode:%d",getRobotPresentMode());

            OLED\_printf(1,0,"key:%d",keyBoardAndMouseHasChanged);

            OLED\_refresh\_gram();

            //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

        }

        #endif

        //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

调试：关闭电流输出。

OLED\_printf(0,0,"ECD:%d",someAngle.nowGimbalYawECD);

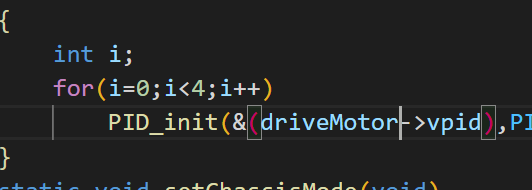
            OLED\_printf(1,0,"vx:%f",robotTotalSpeedControl.vx);

            OLED\_printf(2,0,"vy:%f",robotTotalSpeedControl.vy);

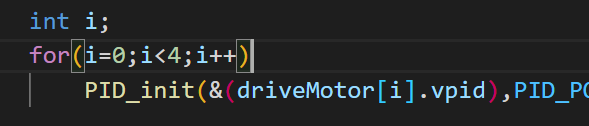
            OLED\_printf(3,0,"w:%f",robotTotalSpeedControl.w);

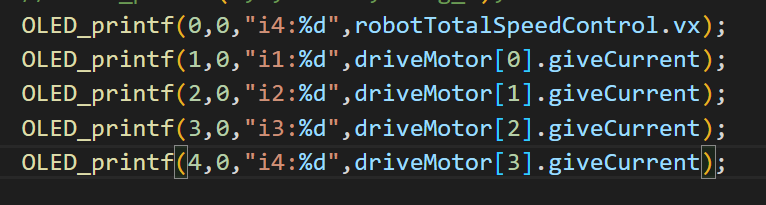
这些是正确的

吐了，查错n小时，发现是这个



应该是



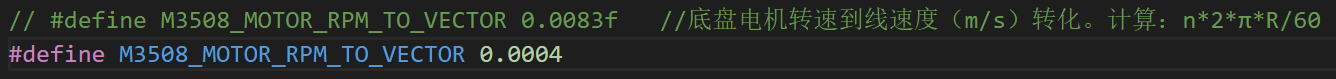


而且这是整形数据……

然后还是不对，

Xs,最后终于发现了，之前拿手册看轮子长度，而后把一个参数“改正”了。然后系统出现了巨大振荡。

Xs，这是减速电机呀！改的前后，这里两个值差了20倍，改之后系统发生高频振荡。



那问题就解决了。

总结一下：一个是循环时参数写错了，把数组内的值写成了第一个元素，导致剩下的3个PID没有初始化。另一个是当参数过大会导致高频振荡，不知道数量级的时候，还是从小的值开始测试吧。

五、测试结果：键盘和遥控器按键能正常控制机器人。且云台坐标系变换工作正常。

迷糊了，电机转动正方向是逆时针方向。

好了，现在改了一堆参数，正常遥控。得到如下效果。

遥控器右边通道控制前后左右平移，左边左右通道控制向左右转。电脑键盘wasd控制前后左右平移。移动方向是云台坐标系。机器人启动的时候需要让头指向正前方。

上机测试时，我们会遇到许多问题。有的是特别小而低级的错误，查错很久才找出来。需要一些调试方法：例如OLED的三步调试法：宏定义开关和刷新时间、需要的变量、输出和刷新。

六、展望：还有云台控制和发射机构控制需要重写。另外，需要增加超声模块、超级电容模块。这是必须的，不仅是因为“创新”，也是因为如果不加就无法满足“充分再设计”条件，就会导致机器人无法通过检录，就上不了场。