RC10_LIB FrameWork用户手册

RC10_LIB将提供大量预制菜，旨在让对底层驱动不熟悉的用户也能畅快书写应用层代码。而本用户手册也是预制菜的一环，旨在让用户可以更快上手使用RC10_LIB

程序中目前执行的命名规范

1. 在类中的变量统一带 _ 的后缀，如`rpm_`
2. 在类中的成员以小写字母开头
3. 类名不要纯小写字母和大写字母
4. RC10_LIB库中的头文件与源文件命名需带分支的前缀，如"Motor_","BSP_"

开发建议

1. 多写注释，如果懒得写，可以像我一样用vscode自带的ai补全注释
2. 当您在开发没有头绪时候，可以回顾开发手册
3. 不要将非API加入RC10_LIB

BSP分支

FreeRTOS的使用

在BSP_RTOS.h文件中，封装了基本的RTOS使用，目前有基本的任务和队列

1. 目前RtosTask的任务运行拥有两种模式
    1. 超级预制菜模式：用户在初始化时候只需给定任务名，以及书写一个初始化函数用于放置start函数即可.(**注意：必须运行start函数才能注册任务，而且start函数必须在osKernelStart();之前运行,main.cpp中的**)

```
/*举例*/
/*
    1.用户需要做的，使用RtosTask实例化任务
    2.在你的初始化函数中，如此处的init()，写入start函数，指定任务的优先级、栈大小
    3.在超级预制菜模式下，只需要重写loop，写入你想执行的任务即可
*/
class FrameDemo : public RtosTask
{
public:
    FrameDemo() : RtosTask("FrameDemo") {}
    void init();
    void loop() override;
    volatile int counter = 0;
};

void FrameDemo::loop()
{
    counter++;
}
```

```
void FrameDemo::init()
{
    start(osPriorityNormal, 256);
}
```

2. 自定义模式

```
/*举例*/
    /*
        1.用户需要做的，使用RtosTask实例化任务，同时传入构造函数的第二个参数，延迟时间(默认是1)，将其设置0，则进入自定义模式
        2.在你的初始化函数中，如此处的init()，写入start函数，指定任务的优先级、栈大小
        3.在自定义模式下，你需要自行完成任务的骨架，重写run()成员
    */
    class FrameDemo : public RtosTask
    {
    public:
        FrameDemo() : RtosTask("FrameDemo",0) {}
        void init();
        void loop() override;
        volatile int counter = 0;
    };

    void FrameDemo::run()
    {
        static int i;
        for(;;)
        {
            i++;
            if(i > 10)
            {
                counter++;
                i = 0;
            }

            osDelay(1);
        }

    }

    void FrameDemo::init()
    {
        start(osPriorityNormal, 256);
    }
```

APP分支

APP_tool

工具类

APP_PID

1. 位置式PID
    1. 采用了梯形积分、微分先行、积分分离
        1. 微分先行：传统PID在target_set突变时，微分项会产生冲激；微分先行用于不计算error的变化率，而是直接计算feedback的变化率

```
// 传统 D 项: D = kd * (error - last_error) / dt
// 微分先行 D 项: D = kd * (last_feedback - current_feedback) / dt
```

        2. 梯形积分： 比简单的矩形积分（I += ki * error * dt）更精确，尤其是在采样时间 dt 不稳定或误差变化较快时。它计算的是当前误差和上次误差构成的梯形面积。

```
integral_term += ki * (error + last_error) / 2.0f * dt;
```

        3. 积分分离： 在误差很大时，暂时禁用积分累加，防止积分项过快饱和，导致系统超调严重。只有当误差进入一个可接受的范围后，才开始累加积分。

```
if (abs(error) < I_SeparaThreshold_)
{
    // 只有在误差较小时才累加积分
    integral_term += ...;
}
```

2. 增量式PID
    1. 增量式PID加入了微分跟踪器(Track_D)，作为一个信号预处理模块
        1. 传统PID在目标值发生阶跃时候会产生突变，使系统发生震荡。
        2. 原理：
            1. 输入一个目标值target给Track_D
            2. Track_D不会立刻把target交给PID，而是在内部模拟一个二阶动态系统，生成一个平滑且连续的过渡曲线V1，使得V1平滑逼近target。
            3. Track_D将V1作为PID的实际目标值
        3. Track_D在实际场景的作用： 若你使用PID控制器让机器人从A到B点
            1. 无Track_D: 机器人会猛的启动，然后急刹车到B点，而且可能超调，不是很稳定。
            2. 有Track_D：机器人会规划一条平滑的加速曲线，然后稳稳当当加速、匀速、减速，然后精准停到B点
    2.
3. 用户该如何使用？
    1. 位置式PID 伪代码

```
//init
PID_Param_Config param_init = {···};// 设置 Kp, Ki, Kd 等参数
PID_Position  pid(param_init);
float target_pos = 100.0f;
motor.set_pos(pid.calc(target_pos, motor.get_pos()));
```

2. 增量式PID 伪代码

```
// 初始化
PID_Param_Config param_init = { ... }; // 设置 Kp, Ki, Kd 等参数
float td_ratio = 0.8f; // 设置跟踪微分器速度，0为不使用
PID_Incremental pid_speed(param_init, td_ratio);


float target_speed = 5000.0f; // RPM
float current_speed = motor.get_speed(); // 获取当前电机速度

// pid_calc返回的是“当前总输出”，可以直接使用
float motor_output = pid_speed.pid_calc(target_speed, current_speed);

// 将计算结果发送给电机
motor.set_current(motor_output); // 假设是电流环控制
```

Motor分支

Module分支