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

attention: 这份手册很大程度是AI生成的，笔者只负责修改其中部分，若发现有纰漏，请及时告诉我，万分感谢。

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
4. 禁止一切动态内存分配

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;
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
```

Module分支

DJI 电机使用指南

本指南将引导你完成从硬件初始化到在 RTOS 任务中控制大疆系列电机（M3508, M2006, GM6020）的完整流程。

设计理念回顾

在开始之前，请记住 RC10_LIB 的核心设计：

1. **分离原则**: `fdCANbus` 是一个纯粹的通信通道。电机的所有控制逻辑（PID计算）和协议打包都在电机类自身 (`M3508`, `DJI_Group` 等) 中完成。
2. **自动化调度**: 你 **不需要** 手动调用 PID 计算或 CAN 发送函数。`fdCANbus` 内部的 `schedulerTask` 会以 1kHz 的频率自动完成以下工作：
    - 调用所有已注册电机（或电机组）的 `update()` 方法，执行 PID 控制环路。
    - 调用 `packCommand()` 方法，将计算出的控制指令打包成 CAN 帧。
    - 将 CAN 帧发送出去。

3. **用户职责**: 你的主要工作是在一个独立的控制任务中，根据需要调用 `setTargetCurrent()`, `setTargetRPM()`, `setTargetAngle()` 来设定目标值即可。

第一步：系统初始化

所有硬件和对象的初始化都应该在启动 RTOS 调度器 (`osKernelStart()`) 之前完成。推荐在 `main.cpp` 的 `USER CODE BEGIN 2` 和 `USER CODE END 2` 之间，或者一个专门的 `user_setup.cpp` 文件中进行。

以下是一个完整的初始化示例：

```
/* main.cpp 或 user_setup.cpp */

#include "fdcan.h" // 由CubeMX生成
#include "BSP_fdCAN_Driver.h"
#include "Motor_DJI.h"

// 1. 定义全局的CAN总线和电机对象指针
//    使用指针是为了方便在不同文件中访问，你也可以选择其他方式管理对象
fdCANbus* can1_bus_p;
M3508* m3508_p;
DJI_Group* group1_4_p;

// 2. 创建一个初始化函数
void user_setup()
{
    // --- CAN 总线初始化 ---
    // 参数1: FDCAN句柄 (来自fdcan.h)
    // 参数2: 总线ID (1, 2, 或 3)，用于中断路由
    fdCANbus can1_bus_p(&hfdcan1, 1);

    // --- 电机和电机组初始化 ---
    // 创建一个 M3508 电机实例
    // 参数1: 电机ID (1-4 对应 0x201-0x204, 5-8 对应 0x1FF)
    // 参数2: 所属的CAN总线指针
    M3508 m3508_p(1, can1_bus_p);

    // 创建一个 DJI 电机组 (用于4合1打包)
    // 参数1: 基础发送ID (0x200 或 0x1FF)
    // 参数2: 所属的CAN总线指针
    DJI_Group group1_4_p(0x200, can1_bus_p);

    // --- 注册与配置 ---
    // 将电机添加到电机组
    group1_4_p->addMotor(m3508_p);
    // 你可以继续添加更多电机到这个组...
    // group1_4_p->addMotor(another_motor_p);

    // 【重要】将电机组注册到CAN总线
    // 注意：我们注册的是 group1_4_p，而不是 m3508_p。
    // fdCANbus 会自动调用 group1_4_p 的 update 和 packCommand 方法。
    can1_bus_p->registerMotor(group1_4_p);
```

```
    // --- PID 参数初始化 ---
    // 为 m3508_p 配置PID参数 (AI生成的参数，不要直接套用)
    PID_Param_Config speed_pid_params = {
        .kp = 10.0f,
        .ki = 1.0f,
        .kd = 0.1f,
        .I_Outlimit = 5000.0f,
        .isIOutlimit = true,
        .output_limit = 16384.0f, // M3508 电流范围
        .deadband = 0.0f
    };
    PID_Param_Config angle_pid_params = {
        .kp = 100.0f,
        .ki = 0.5f,
        .kd = 0.0f,
        .I_Outlimit = 300.0f, // 位置环积分输出限制，通常用于限制最大速度
        .isIOutlimit = true,
        .output_limit = 1000.0f, // 位置环输出限制，限制最大目标速度
        .deadband = 0.1f // 死区，防止在目标附近微小抖动
    };
    // 调用pid_init进行初始化
    m3508_p->pid_init(speed_pid_params, 0.1f, angle_pid_params, 5.0f);

    // --- 启动CAN总线 ---
    // 这会启动CAN硬件、配置滤波器并启动内部的 rxTask 和 schedulerTask
    can1_bus_p->init();
}

// 在 main() 函数中调用
int main(void)
{
    // ... HAL_Init(), SystemClock_Config(), MX_GPIO_Init(), MX_FDCAN1_Init() ...

    /* USER CODE BEGIN 2 */
    user_setup(); // 调用我们的初始化函数
    /* USER CODE END 2 */

    // ... osKernelInitialize(), 创建你的控制任务 ...

    osKernelStart(); // 启动调度器

    // ...
}
```

第二步：在任务中控制电机

现在，你可以创建一个或多个任务来发送控制指令。

1. **创建控制任务类**

   继承 `RtosTask` 并实现 `loop()` 方法（因为这是一个周期性任务）。

```
/* 在你的用户应用层，e.g., User/Control/control_task.h */
#include "BSP_RTOS.h"
#include "Motor_DJI.h"

// 声明在初始化代码中创建的全局对象
extern M3508* m3508_p;

class MyControlTask : public RtosTask
{
public:
    // 构造函数，设置任务名和周期 (e.g., 10ms -> 100Hz)
    MyControlTask() : RtosTask("MyCtrl", 10) {}

protected:
    // 周期性执行的循环体
    void loop() override;
};
```

2. **实现控制逻辑**

在 `loop()` 中，你可以根据系统状态切换电机的控制模式并设定目标。

```
/* 在你的用户应用层，e.g., User/Control/control_task.cpp */
#include "control_task.h"
#include <cmath> // for sin

void MyControlTask::loop()
{
    // 获取当前时间戳，用于产生变化的目标值
    uint32_t tick = osKernelGetTickCount();

    // --- 示例1：位置控制 ---
    // 让电机在 -90 到 90 度之间来回摆动
    float target_angle = 90.0f * sin(tick * 0.002f);
    m3508_p->setTargetAngle(target_angle);

    /*
    // --- 示例2：速度控制 ---
    // 设置一个恒定的目标转速 (RPM)
    // m3508_p->setTargetRPM(500.0f);
    */

    /*
    // --- 示例3：电流控制 ---
    // 设置一个恒定的目标电流 (Ampere)
    // m3508_p->setTargetCurrent(0.5f);
    */
}
```

3. **创建并启动任务**

最后，在 `main.cpp` 中 `osKernelStart()` 之前，创建任务实例并启动它。

```
/* main.cpp */

// ...
#include "control_task.h"

MyControlTask my_control_task; // 创建任务实例

int main(void)
{
    // ...
    user_setup();
    /* USER CODE BEGIN 2 */

    // 启动控制任务
    my_control_task.start(osPriorityNormal);

    /* USER CODE END 2 */

    osKernelStart();
    // ...
}
```

第三步：连接定时器中断 (移植时注意)

注意：在当前项目中，此步骤已经由开发者完成。本节内容主要用于向后继维护者或希望将此库移植到其他项目的开发者说明其工作原理和必要配置。

`fdCANbus` 的 `schedulerTask` 依赖一个 1kHz 的信号量来触发。这个信号量由一个全局中断函数 `fdcan_global_scheduler_tick_isr()` 发出。

因此，任何使用 `RC10_LIB` 的项目都 **必须** 将一个硬件定时器（如 TIM6, TIM7）配置为 1ms 触发一次中断，并在其中断回调函数中调用 `fdcan_global_scheduler_tick_isr()`。

1. 使用 CubeMX 配置一个基础定时器

```
*   选择一个定时器 (e.g., TIM7)。
*   设置 `Prescaler` 和 `Counter Period` 使其每 1ms 产生一次更新事件 (Update
Event)。
    *   例如，如果 TIM CLK 是 120MHz，可以设置 Prescaler = 120-1，Counter Period =
1000-1。
*   在 "NVIC Settings" 标签页中，**启用该定时器的更新中断**。
```

2. 在中断回调中添加调用

在 `stm32h7xx_it.c` 文件中，找到定时器的中断服务函数 `TIMx_IRQHandler`。HAL 库会在此函数中调用一个弱定义的回调函数 `HAL_TIM_PeriodElapsedCallback`。你需要重写这个回调。

```c

// 引入我们的全局调度函数
extern void fdcan_global_scheduler_tick_isr(void);

/**
  * @brief  Period elapsed callback in non-blocking mode
  * @param  htim TIM handle
  * @retval None
  */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  /* USER CODE BEGIN Callback 0 */

  /* USER CODE END Callback 0 */
  if (htim->Instance == TIM1) { // TIM1 是 HAL 库默认的 SysTick 定时器
    HAL_IncTick();
  }
  /* USER CODE BEGIN Callback 1 */

  // 如果是用于1kHz调度的定时器 (在当前项目中是TIM6)
  if (htim->Instance == TIM6)
  {
    fdcan_global_scheduler_tick_isr();
  }

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

完成以上步骤后，你的电机控制系统就完整建立起来了。编译并下载程序，电机应该会按照 `MyControlTask` 中设定的逻辑开始运动。