RC10_LIB FrameWork用户手册

用户手册？亦或说是RC10_LIB的说明书.

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

User

1. 机构控制类放在Control
2. 调试debug/demo类放在debug
3. Setup用于放初始化文件

RC10_LIB的核心设计原则

1. 严格分层，职责单一 框架分为硬件驱动层、设备协议层、算法层和应用层。当你添加新功能时，必须明确其归属。

   硬件驱动只负责与物理总线通信。 设备协议只负责解析和打包特定设备的报文。 算法是纯粹的数学工具。 应用只负责下达高层指令。 原则：一般算法层不涉及任何硬件设备、基层只能调用基层。

2. 信任自动化调度，分离计算与打包

    1. 例如: fdCANbus 框架提供一个高频率的中央调度器，它会自动调用所有注册设备的 update() 和 packCommand()。

    2. update(): 只用于计算。执行如PID等周期性算法，更新内部状态。

    3. packCommand(): 只用于打包。读取 update() 的计算结果，并将其组装成待发送的CAN报文。

    4. setTarget...(): 只用于接收指令。这是你的驱动提供给应用层的接口，用于设置高级目标。

    5. 原则： 永远不要在 packCommand() 中进行计算，也不要在 update() 中组装报文。相信调度器会按正确的顺序调用它们。

3. 继承统一接口，利用多态实现特异性 框架通过面向接口编程实现扩展性。所有设备驱动都必须继承自一个共同的基类（如 Motor_Base）。

   统一管理: 调度器只与基类接口交互，它不关心具体是什么设备。 虚函数实现多态: 使用 virtual 函数（如 get_GearRatio()）来让每个子类提供自己独特的信息或行为。 原则： 你的新设备驱动必须实现基类的所有纯虚函数，并利用虚函数重写（override）来实现其特定协议和功能。

4. 用户使用接口的简化 将一切的重复性工作都在类的封装中实现，使得用户在开发应用层的时候无需写太多冗杂重复的代码，更高效进行开发。

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. **RtosTask 任务封装** `RtosTask` 类提供了两种任务模式，通过构造函数的 `period` 参数区分：

   - **周期性任务 (`period > 0`)**: 任务会以 `period` 指定的Tick间隔自动循环执行 `loop()` 方法。适用于需要固定频率运行的简单逻辑。

     ```
     class MyPeriodicTask : public RtosTask {
     public:
         MyPeriodicTask() : RtosTask("MyTask", 1000) {} // 1000ms周期
     protected:
         void loop() override
         {
             // 这里的代码每1000ms执行一次
         }
     };
     ```

   - **事件驱动任务 (`period = 0`)**: 任务创建后会执行一次 `run()` 方法。`run()` 方法必须包含一个死循环 `for(;;)` 和一个阻塞调用（如 `vTaskDelay`, `xSemaphoreTake`），用于等待外部事件。适用于需要被动触发的复杂任务，例如CAN总线的调度和接收任务。

     ```
     class MyEventTask : public RtosTask {
     public:
         MyEventTask() : RtosTask("EventTask", 0) {} // 事件驱动
     protected:
         void run() override
         {
             for(;;)
             {
                 // 等待信号量或其他事件
                 xSemaphoreTake(mySemaphore, portMAX_DELAY);
                 // 处理事件...
             }
         }
     };
     ```

2. **RtosQueue 队列封装** 这是一个模板类，可以方便地创建和使用线程安全的队列。

```
// 创建一个能容纳8个int的队列
RtosQueue<int> myQueue(8);

// 在一个任务中发送数据
myQueue.send(123);

// 在另一个任务中接收数据
int received_value;
if (myQueue.recv(received_value, 100)) { // 等待100ms
    // 成功接收到数据
}
```

APP分支

APP_tool

工具类，提供如 `constrain`（限幅）等通用函数。

APP_debugTool

提供调试工具，如串口打印数据。

APP_PID

提供了位置式和增量式两种PID控制器。

1. **核心设计**
   - **位置式PID**: 采用了梯形积分、微分先行、积分分离等改进算法，适用于大部分需要精确位置控制的场景。
   - **增量式PID**: 加入了微分跟踪器(Track_D)，能有效平滑目标值的阶跃变化，减少系统震荡，适用于速度控制等场景。
   - **固定采样时间**: PID控制器内部的 `dt` 使用时间戳方式计算，但它大部分时候的值是为1ms。这是一个**核心设计**，它强依赖于调用 `pid_calc` 的 `update()` 方法被一个精确的1kHz调度器（如 `fdCANbus::schedulerTaskbody`）所调用。后续会考虑把杨哥那套用编码值计算时间的代码搬过来，可以让dt更加精确。
2. **用户该如何使用？** 在电机类（如 `M3508`）的 `pid_init` 函数中初始化PID参数，然后在 `update` 函数中调用 `pid_calc` 即可。用户无需关心 `dt` 的计算。

```
// 在 M3508::update() 中
case SPEED_CONTROL:
{
    // target_rpm_ 和 this->rpm_ 都是输出轴转速，尺度统一
    target_current_ = speed_pid_.pid_calc(target_rpm_, this->rpm_);
```

```
        break;
    }
```

如果你使用的是位置式PID 位置式PID包含了两种模式： 1. 线性模式：此模式下，适合路程式的PID 2. 循环模式：此模式下，适合云台式的PID，范围为【】

APP_CoordConvert

`APP_CoordConvert` 是一个基于 `CMSIS-DSP` 库优化的高性能坐标变换工具，用于处理2D和3D空间中的平移和旋转。

核心特性

- **高性能**: 所有矩阵运算都由 `arm_math.h` 中的函数完成，充分利用硬件加速。
- **易于使用**: 提供了 `HomogeneousTransform2D` 和 `HomogeneousTransform3D` 两个类，接口清晰直观。
- **功能完备**: 支持设置变换、应用变换、矩阵乘法（变换叠加）和求逆变换。

【重要提示】

- **角度单位**: 所有函数的角度参数（如 `theta_rad`, `roll_rad`）都必须使用 **弧度 (radians)** 作为单位。
- **命名空间**: 所有类和函数都位于 `geometry` 命名空间下。

2D变换使用示例

假设有一个传感器安装在机器人上，其坐标系相对于机器人中心坐标系有如下关系：

- 沿机器人X轴平移了 `0.2` 米。
- 沿机器人Y轴平移了 `0.1` 米。
- 逆时针旋转了 `45` 度。

现在，传感器检测到了一个在其自身坐标系下的点 `(0.5, 0.0)`，我们想知道这个点在机器人中心坐标系下的位置。

```
#include "APP_CoordConvert.h"
#include "arm_math.h" // For PI constant

// 使用命名空间
using namespace geometry;

void transform_example_2d()
{
    // 1. 定义一个 Point2D 对象来描述从传感器到机器人中心的位姿
    //    平移 (0.2, 0.1)，旋转 45 度 (PI/4 弧度)
    Point2D sensor_pose(0.2f, 0.1f, PI / 4.0f);

    // 2. 使用该位姿对象创建变换矩阵
    HomogeneousTransform2D sensor_to_robot_tf(sensor_pose);

    // 3. 定义在传感器坐标系下的点
```

```
    Point2D point_in_sensor(0.5f, 0.0f);

    // 4. 应用变换，得到在机器人坐标系下的点
    Point2D point_in_robot = sensor_to_robot_tf.apply(point_in_sensor);

    // point_in_robot.x 和 point_in_robot.y 就是最终结果
}
```

3D变换使用示例

假设相机坐标系相对于世界坐标系平移了 `(1.0, 2.0, 0.5)`，并且绕Z轴旋转了90度。

```
#include "APP_CoordConvert.h"
#include "arm_math.h"

using namespace geometry;

void transform_example_3d()
{
    // 1. 定义一个 Point3D 对象来描述从相机到世界坐标系的位姿
    //    平移 (1, 2, 0.5)，绕Z轴(yaw)旋转90度 (PI/2)
    Point3D camera_pose(1.0f, 2.0f, 0.5f, 0.0f, 0.0f, PI / 2.0f);

    // 2. 使用该位姿对象创建变换矩阵
    HomogeneousTransform3D camera_to_world_tf(camera_pose);

    // 3. 定义在相机坐标系下的一个点
    Point3D point_in_camera(0.0f, 1.0f, 0.0f);

    // 4. 应用变换，得到在世界坐标系下的点
    Point3D point_in_world = camera_to_world_tf.apply(point_in_camera);

    // 5. 计算逆变换（从世界坐标系到相机坐标系）
    HomogeneousTransform3D world_to_camera_tf = camera_to_world_tf.inverse();

    // 6. 使用逆变换将世界坐标系下的点转换回相机坐标系
    Point3D point_back_in_camera = world_to_camera_tf.apply(point_in_world);
    // 此时 point_back_in_camera 应该约等于 point_in_camera
}
```

Module分支

此分支主要包含与特定硬件模块相关的代码，例如 `Module_Encoder.cpp`，它负责将编码器的原始值（如0-8191）转换为连续的角度（-∞, +∞）和单圈角度（[0, 360)）。

---

fdCANbus如何工作的？

`fdCANbus` 是整个电机控制库的神经中枢。它负责处理底层的CAN通信，并以精确的频率自动调度所有电机控制任务，将用户从繁琐的实时控制和硬件交互中解放出来。

核心组件与工作流程

`fdCANbus` 内部主要由两个并行的RTOS任务驱动：

1. **接收任务 (`rxTask_`)**:
    - **工作**: 这是一个事件驱动的任务，它永久阻塞并等待 `rxQueue_` 队列中的新消息。
    - **数据流**:
        1. 当CAN硬件接收到一个数据帧，`HAL_FDCAN_RxFifo0Callback` 中断服务程序（ISR）被触发。
        2. ISR调用 `fdcan_global_rx_isr`，该函数从硬件缓冲区读取原始CAN报文。
        3. 原始报文被封装成 `CanFrame` 对象，并被立即推入 `rxQueue_` 队列。
        4. `rxTask_` 被唤醒，从队列中取出 `CanFrame`。
        5. `rxTask_` 遍历所有已注册的电机（`motorList_`），调用每个电机的 `matchesFrame()` 方法来寻找该报文的“主人”。
        6. 一旦找到匹配的电机，就调用其 `updateFeedback()` 方法，将报文交由电机自行解析。
2. **调度任务 (`schedulerTask_`)**:
    - **工作**: 这是一个高优先级的、由定时器精确触发的周期性任务，频率为1kHz。
    - **数据流**:
        1. 一个1kHz的硬件定时器中断触发 `fdcan_global_scheduler_tick_isr()`。
        2. 该ISR释放（Give）一个名为 `schedSem_` 的信号量，然后立即退出。
        3. `schedulerTask_` 在启动后就一直阻塞等待（Take）这个信号量。一旦获取到信号量，它就会被唤醒。
        4. **更新**: 任务首先遍历所有注册的电机（或电机组），并调用它们的 `update()` 方法。这会触发PID计算等控制逻辑。
        5. **打包**: 接着，任务再次遍历所有对象，调用 `packCommand()` 方法来收集需要发送的CAN指令帧。
        6. **发送**: 最后，任务将所有收集到的指令帧通过 `sendFrame()` 方法发送出去。`sendFrame` 内部使用互斥锁 `tx_mutex_` 来确保多任务访问CAN硬件的线程安全。
        7. 完成一轮调度后，`schedulerTask_` 返回循环的开始，再次阻塞等待下一次的信号量，从而实现精确的1ms周期。

关键设计决策

- **中断服务程序（ISR）最小化**: ISR只做最少的工作——读取数据并将其推入队列。所有耗时的操作（如遍历、匹配、解析）都转移到优先级较低的 `rxTask_` 中执行，这确保了系统的实时响应能力。
- **发送与接收分离**: 接收是完全异步和事件驱动的，而发送则是同步和周期性的。这种设计符合控制系统的典型模式：持续接收反馈，并以固定的频率输出控制指令。
- **全局中断路由**: 通过一个全局的 `g_fdcan_bus_map` 数组，可以将来自HAL库的、不区分具体总线的C风格中断回调，精确地路由到对应的 `fdCANbus` C++对象实例上。这使得代码可以轻松支持多个CAN总线。
- **线程安全**: 通过使用RTOS队列（`RtosQueue`）和互斥锁（`tx_mutex_`），`fdCANbus` 确保了在多任务环境下数据交换和硬件访问的安全性。

电机库核心设计与使用指南

本指南将引导你完成从硬件初始化到在 RTOS 任务中控制电机的完整流程。

核心设计思想

1. **数据转换前置**: 在 `DJI_Motor::updateFeedback` 函数中，从CAN总线接收到的**电机转子原始数据**（转速、编码器值）会**立即**通过虚函数 `get_GearRatio()` 获取正确的减速比，并被转换为**减速后的输出轴数据**。
2. **内部状态统一**: 转换完成后，所有存储在基类 `Motor_Base` 中的成员变量（`rpm_`, `angle_`, `totalAngle_`）的含义都统一为**输出轴的状态**。
3. **控制与反馈尺度统一**: PID控制环路（在 `update()` 方法中）的**目标值**（如 `target_rpm_`）和**反馈值**（如 `this->rpm_`）都基于**输出轴的尺度**进行计算，保证了控制的正确性。
4. **调度自动化**: 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 的频率自动完成所有已注册电机（或电机组）的 `update()` 和 `packCommand()` 调用。
5. **用户职责**: 你的工作非常简单，只需在一个独立的控制任务中，根据需要调用 `setTargetRPM()`, `setTargetAngle()` 等函数来设定**输出轴的目标值**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。

```
/* main.cpp 或 user_setup.cpp */

#include "BSP_fdCAN_Driver.h"
#include "Motor_DJI.h"

// 1. 定义全局对象
// 注意：这里直接定义对象，而不是指针，以避免动态内存分配
fdCANbus CAN1_Bus(&hfdcan1, 1); // CAN总线
M3508 m3508_1(1, &CAN1_Bus);     // M3508电机, ID为1
DJI_Group DJI_Group_1(0x200, &CAN1_Bus); // DJI电机组, 发送ID为0x200

// 2. 创建一个初始化函数
void user_setup()
{
    // --- PID参数配置 --- (AI生成的，并非通用参数)
    PID_Param_Config speed_pid_params =
    {
        .kp = 10.0f, .ki = 0.5f, .kd = 0.0f,
        .I_Outlimit = 5000.0f, .isIOutlimit = true,
        .output_limit = 16000.0f, .deadband = 0.0f
    };
    PID_Param_Config angle_pid_params =
    {
        .kp = 0.5f, .ki = 0.0f, .kd = 0.0f,
```

```
        .I_Outlimit = 100.0f, .isIOutlimit = true,
        .output_limit = 500.0f, .deadband = 0.0f
    };
    m3508_1.pid_init(speed_pid_params, 0.0f, angle_pid_params, 30.0f);

    // --- 注册与配置 ---
    // 将电机添加到电机组
    DJI_Group_1.addMotor(&m3508_1);
    // 你可以继续添加更多电机到这个组...
    // DJI_Group_1.addMotor(&another_motor);

    // 【重要】将电机本身和电机组都注册到CAN总线
    // 1. 注册电机本身，使其能接收反馈报文并更新状态
    CAN1_Bus.registerMotor(&m3508_1);
    // 2. 注册电机组，使其能被调度器调用 packCommand() 来打包发送电流指令
    CAN1_Bus.registerMotor(&DJI_Group_1);

    // --- 启动总线 ---
    // 这会启动CAN的接收中断和1kHz的调度任务
    CAN1_Bus.init();
}

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

    user_setup(); // 调用我们的初始化函数

    osKernelInitialize();
    // ... 创建其他用户任务 ...
    osKernelStart();

    // ...
}
```

如果你想拓展电机？

假设你要添加一个非DJI的、有自己独特CAN协议的电机，例如 MyMotor。

1. 创建 `Motor_MyMotor.h`

```
#include "Motor_Base.h"
#include "APP_PID.h" // 如果需要PID

class MyMotor : public Motor_Base {
public:
    // 1. 构造函数：调用基类构造函数
    MyMotor(uint32_t id, fdCANbus* bus)
        : Motor_Base(id, false, bus) // 假设使用标准帧
    {
```

```
        // 初始化该电机的私有成员
    }

    // 2. 【必须】覆盖 packCommand
    //    根据 target_current_ 等目标值，打包成该电机的CAN帧
    std::size_t packCommand(CanFrame outFrames[], std::size_t maxFrames) override;

    // 3. 【必须】覆盖 updateFeedback
    //    解析收到的CAN帧，更新 rpm_, angle_ 等成员变量
    void updateFeedback(const CanFrame& cf) override;

    // 4. 【必须】覆盖 matchesFrame
    //    判断收到的CAN帧是否属于这个电机
    bool matchesFrame(const CanFrame& cf) const override;

    // 5. 【必须】覆盖 get_GearRatio
    //    返回该电机的真实减速比
    float get_GearRatio() const override { return 27.0f; } // 假设减速比是27

    // 6. 实现 update 方法，用于执行PID计算
    void update() override;

    // 7. 实现 setTarget... 等控制接口
    void setTargetRPM(float rpm_set) override;

private:
    // 该电机的私有成员，如PID控制器
    PID_Incremental speed_pid_;
};
```

2. 在 Motor_MyMotor.cpp 中实现功能

```
#include "Motor_MyMotor.h"

std::size_t MyMotor::packCommand(CanFrame outFrames[], std::size_t maxFrames) {
    // ... 根据 this->target_current_ 打包CAN帧 ...
    // outFrames[0].ID = 0x123;
    // outFrames[0].data[0] = ...;
    return 1; // 返回打包的帧数
}

void MyMotor::updateFeedback(const CanFrame& cf) {
    // ... 解析 cf.data ...
    // float raw_rpm = ...;
    // this->rpm_ = raw_rpm / get_GearRatio(); // 转换为输出轴转速
}

bool MyMotor::matchesFrame(const CanFrame& cf) const {
    // 判断逻辑，例如：
    return (cf.ID == (0x200 + this->motor_id_));
}
```

```
void MyMotor::update() {
    // ... 调用PID计算 ...
    // target_current_ = speed_pid_.pid_calc(target_rpm_, this->rpm_);
}

void MyMotor::setTargetRPM(float rpm_set) {
    // ... 设置目标值 ...
    this->target_rpm_ = rpm_set;
}
```

3. 在应用层使用 像使用 `M3508` 一样，创建 `MyMotor` 对象，并将其注册到 `fdCANbus` 即可。调度器会自动处理后续的一切。