// ========== main.c ==========

```
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include <math.h>
#include "system_init.h"
#include "wifi_softap_module.h"
#include "rs01_motor.h"
#include "motor_web_control.h"
#include "alternating_speed.h"
#include "continuous_torque_velocity_mode.h"
#include "velocity_tracking_mode.h"
extern MotorDataCallback data_callback;
extern MI_Motor motors[2];
extern bool alternating_speed_enabled;
static const char *TAG = "exoskeleton_main";
float motor_target_speed[2] = {0.0f, 0.0f};
bool motor_speed_control_enabled = false;
#ifndef MOTOR_CONTROL_PARAMS_T_DEFINED
#define MOTOR_CONTROL_PARAMS_T_DEFINED
typedef struct {
  float torque;
  float position;
  float speed;
  float kp;
  float kd;
} motor_control_params_t;
#endif
motor_control_params_t motor_params[2] = {{0}};
motor_control_params_t sent_params[2] = {{0}};
static TaskHandle_t uart_parse_task_handle = NULL;
bool uart_parse_enabled = true;
static TaskHandle_t motion_detection_task_handle = NULL;
#define PARAM_CHANGE_THRESHOLD 0.001f
position_ring_buffer_t position_recorder_motor1;
position_ring_buffer_t position_recorder_motor2;
motion_mode_state_t motion_state;
bool motion_mode_detection_enabled = false;
bool velocity_tracking_mode_active = false;
extern motor_control_params_t motor_params[2];
void set_motor_params(int motor_id, float torque, float position, float speed, float kp, float kd);
void unified_motor_control(int motor_id, const motor_control_params_t* params);
void enable_velocity_tracking_mode(void);
void disable_velocity_tracking_mode(void);
bool is_velocity_tracking_mode_active(void);
void reset_velocity_tracking_mode(void);
void motor_data_update_callback(MI_Motor* motor) {
  ESP_LOGD(TAG, "电机%d 数据更新: 位置=%.3f, 速度=%.3f, 电流=%.3f, 温度=%.1f°C",
       motor->id, motor->position, motor->velocity, motor->current, motor->temperature);
  if (motor->error != 0) {
```

```
      ESP_LOGW(TAG, "电机%d 错误状态: 0x%02X", motor->id, motor->error);
    }
}
void UART_Parse_Task(void* pvParameters) {
    ESP_LOGI(TAG, "串口数据解析任务启动");
    while (uart_parse_enabled) {
        handle_uart_rx();
        vTaskDelay(pdMS_TO_TICKS(5));
    }
    ESP_LOGI(TAG, "串口数据解析任务退出");
    uart_parse_task_handle = NULL;
    vTaskDelete(NULL);
}
void Motion_Detection_Task(void* pvParameters) {
    ESP_LOGI(TAG, "运动模式检测任务启动");
    TickType_t last_position_check = xTaskGetTickCount();
    TickType_t last_mode_check = xTaskGetTickCount();
    TickType_t last_torque_update = xTaskGetTickCount();
    const TickType_t position_check_interval = pdMS_TO_TICKS(20);
    const TickType_t mode_check_interval = pdMS_TO_TICKS(1000);
    const TickType_t torque_update_interval = pdMS_TO_TICKS(100);
    while (1) {
        TickType_t current_time = xTaskGetTickCount();
        uint32_t timestamp = current_time * portTICK_PERIOD_MS;
        bool need_for_motion_detection = motion_mode_detection_enabled;
        bool need_for_velocity_tracking = velocity_tracking_mode_enabled &&
                                          (velocity_tracking_mode_get_state() == VELOCITY_TRACKING_ENABLED);
        bool need_position_recording = need_for_motion_detection || need_for_velocity_tracking;
        if (need_position_recording) {
            if ((current_time - last_position_check) >= position_check_interval) {
                position_ring_buffer_add_if_changed(&position_recorder_motor1, motors[0].position, timestamp);
                position_ring_buffer_add_if_changed(&position_recorder_motor2, motors[1].position, timestamp);
                last_position_check = current_time;
            }
        }
        if (motion_mode_detection_enabled) {
            if ((current_time - last_mode_check) >= mode_check_interval) {
                motion_mode_t detected_mode = detect_motion_mode(&position_recorder_motor1, timestamp, &motion_state);
                if (detected_mode != motion_state.current_mode) {
                    ESP_LOGI(TAG, "运动模式切换: %s",
                             detected_mode == MOTION_MODE_STATIC ? "静止" :
                             detected_mode == MOTION_MODE_WALKING ? "行走" : "爬楼");
                    if (detected_mode == MOTION_MODE_STATIC) {
                        position_ring_buffer_clear(&position_recorder_motor1);
                        position_ring_buffer_clear(&position_recorder_motor2);
                        ESP_LOGI(TAG, "【静止确认】清理位置缓存区，避免旧数据影响");
                    }
                }
                motor_params_t motor1_params, motor2_params;
                update_motion_mode_and_get_params(&motion_state, detected_mode, timestamp, 1, &motor1_params);
```

```
            update_motion_mode_and_get_params(&motion_state, detected_mode, timestamp, 2, &motor2_params);
            motor_params[0].speed = (float)motor1_params.velocity;
            motor_params[0].torque = motor1_params.torque;
            motor_params[0].kd = (float)motor1_params.kd;
            motor_params[0].position = 0.0f;
            motor_params[0].kp = 0.0f;
            motor_params[1].speed = (float)motor2_params.velocity;
            motor_params[1].torque = motor2_params.torque;
            motor_params[1].kd = (float)motor2_params.kd;
            motor_params[1].position = 0.0f;
            motor_params[1].kp = 0.0f;
            if (detected_mode == MOTION_MODE_STATIC) {
              ESP_LOGI(TAG, "【静止参数】电机 1: 速度=%.1f, 力矩=%.1f, kd=%.1f",
                  motor_params[0].speed, motor_params[0].torque, motor_params[0].kd);
              ESP_LOGI(TAG, "【静止参数】电机 2: 速度=%.1f, 力矩=%.1f, kd=%.1f",
                  motor_params[1].speed, motor_params[1].torque, motor_params[1].kd);
            }
            unified_motor_control(0, &motor_params[0]);
            unified_motor_control(1, &motor_params[1]);
            last_mode_check = current_time;
          }
          if ((current_time - last_torque_update) >= torque_update_interval) {
            motor_params_t motor1_params, motor2_params;
            motion_mode_t update_mode = motion_state.in_static_confirmation ? MOTION_MODE_STATIC : motion_state.current_mode;
            update_motion_mode_and_get_params(&motion_state, update_mode, timestamp, 1, &motor1_params);
            update_motion_mode_and_get_params(&motion_state, update_mode, timestamp, 2, &motor2_params);
            motor_params[0].torque = motor1_params.torque;
            motor_params[1].torque = motor2_params.torque;
            unified_motor_control(0, &motor_params[0]);
            unified_motor_control(1, &motor_params[1]);
            last_torque_update = current_time;
          }
        }
        if (velocity_tracking_mode_active && velocity_tracking_mode_is_enabled()) {
          float motor1_velocity = motors[0].velocity;
          float motor2_velocity = motors[1].velocity;
          velocity_tracking_mode_update(&position_recorder_motor1, &position_recorder_motor2, motor1_velocity, motor2_velocity,
timestamp);
        }
        vTaskDelay(pdMS_TO_TICKS(50));
      }
      ESP_LOGI(TAG, "运动模式检测任务退出");
      motion_detection_task_handle = NULL;
      vTaskDelete(NULL);}
void Start_UART_Parse_Task(void) {
      if (uart_parse_task_handle == NULL) {
        uart_parse_enabled = true;
        BaseType_t result = xTaskCreate(
          UART_Parse_Task,
          "UARTParse",
```

```
            4096,
            NULL,
            5,
            &uart_parse_task_handle
        );
        if (result == pdPASS) {
            ESP_LOGI(TAG, "串口数据解析任务创建成功");
        } else {
            ESP_LOGE(TAG, "串口数据解析任务创建失败");
            uart_parse_enabled = false;
        }
    } else {
        ESP_LOGW(TAG, "串口数据解析任务已在运行");
    }
}
void Stop_UART_Parse_Task(void) {
    if (uart_parse_task_handle != NULL) {
        uart_parse_enabled = false;
        ESP_LOGI(TAG, "请求停止串口数据解析任务");
    }
}
void Start_Motion_Detection_Task(void) {
    if (motion_detection_task_handle == NULL) {
        BaseType_t result = xTaskCreate(
            Motion_Detection_Task,
            "MotionDetect",
            4096,
            NULL,
            3,
            &motion_detection_task_handle
        );
        if (result == pdPASS) {
            ESP_LOGI(TAG, "运动模式检测任务创建成功");
        } else {
            ESP_LOGE(TAG, "运动模式检测任务创建失败");
        }
    } else {
        ESP_LOGW(TAG, "运动模式检测任务已在运行");
    }
}
void Stop_Motion_Detection_Task(void) {
    if (motion_detection_task_handle != NULL) {
        vTaskDelete(motion_detection_task_handle);
        motion_detection_task_handle = NULL;
        ESP_LOGI(TAG, "运动模式检测任务已停止");
    }
}
bool motor_params_changed(int motor_id, const motor_control_params_t* params) {
    if (motor_id < 0 || motor_id >= 2 || params == NULL) return false;
    const float threshold = 0.001f;
```

```
    return (fabsf(params->torque - sent_params[motor_id].torque) > threshold) ||
           (fabsf(params->position - sent_params[motor_id].position) > threshold) ||
           (fabsf(params->speed - sent_params[motor_id].speed) > threshold) ||
           (fabsf(params->kp - sent_params[motor_id].kp) > threshold) ||
           (fabsf(params->kd - sent_params[motor_id].kd) > threshold);
}
void update_sent_params(int motor_id, const motor_control_params_t* params) {
    if (motor_id < 0 || motor_id >= 2 || params == NULL) return;
    sent_params[motor_id] = *params;
}
void unified_motor_control(int motor_id, const motor_control_params_t* params) {
    if (motor_id < 0 || motor_id >= 2 || params == NULL) {
        ESP_LOGE(TAG, "无效的电机 ID 或参数: motor_id=%d", motor_id);
        return;
    }
    if (!motor_params_changed(motor_id, params)) {
        return;
    }
    portMUX_TYPE mux = portMUX_INITIALIZER_UNLOCKED;
    portENTER_CRITICAL(&mux);
    Motor_ControlMode(&motors[motor_id], params->torque, params->position,
                      params->speed, params->kp, params->kd);
    portEXIT_CRITICAL(&mux);
    update_sent_params(motor_id, params);
    ESP_LOGI(TAG, "电机%d 控制更新: 力矩=%.2f, 位置=%.2f, 速度=%.2f, Kp=%.2f, Kd=%.2f",
             motor_id+1, params->torque, params->position, params->speed,
             params->kp, params->kd);
}
void set_motor_params(int motor_id, float torque, float position, float speed, float kp, float kd) {
    if (motor_id < 0 || motor_id >= 2) {
        ESP_LOGE(TAG, "无效的电机 ID: %d", motor_id);
        return;
    }
    motor_params[motor_id].torque = torque;
    motor_params[motor_id].position = position;
    motor_params[motor_id].speed = speed;
    motor_params[motor_id].kp = kp;
    motor_params[motor_id].kd = kd;
    unified_motor_control(motor_id, &motor_params[motor_id]);
}
void app_main(void)
{
    ESP_LOGI(TAG, "外骨骼 WiFi 控制系统启动中...");
    ESP_ERROR_CHECK(system_init_all());
    position_ring_buffer_init(&position_recorder_motor1);
    position_ring_buffer_init(&position_recorder_motor2);
    ESP_LOGI(TAG, "位置记录缓存区初始化完成");
    motion_mode_state_init(&motion_state);
    ESP_LOGI(TAG, "运动模式状态管理初始化完成");
    data_callback = motor_data_update_callback;
```

```
    ESP_LOGI(TAG, "已更新电机数据回调函数");
    ESP_LOGI(TAG, "启动 RS01 电机数据解析任务...");
    Start_UART_Parse_Task();
    ESP_LOGI(TAG, "启动运动模式检测任务...");
    Start_Motion_Detection_Task();
    ESP_LOGI(TAG, "初始化速度跟踪模式...");
    velocity_tracking_mode_init();
    velocity_tracking_start_task();
    ESP_LOGI(TAG, "启用速度跟踪模式...");
    velocity_tracking_mode_active = true;
    velocity_tracking_mode_set_enabled(true);
    ESP_LOGI(TAG, "速度跟踪模式已默认启用");
    ESP_LOGI(TAG, "初始化电机控制网页...");
    esp_err_t web_result = motor_web_control_init();
    if (web_result == ESP_OK) {
        ESP_LOGI(TAG, "电机控制网页模块启动成功，可通过 WiFi 访问控制界面");
    } else {
        ESP_LOGE(TAG, "电机控制网页模块启动失败");
    }
    ESP_LOGI(TAG, "系统运行中，等待用户操作...");
    while (1) {
        vTaskDelay(pdMS_TO_TICKS(10000));
        ESP_LOGI(TAG, "系统运行正常，WiFi 连接数：%d", wifi_softap_get_connected_count());
    }
}
void enable_velocity_tracking_mode(void) {
    velocity_tracking_mode_active = true;
    velocity_tracking_mode_set_enabled(true);
    ESP_LOGI(TAG, "速度跟踪模式已启用");
}
void disable_velocity_tracking_mode(void) {
    velocity_tracking_mode_active = false;
    velocity_tracking_mode_set_enabled(false);
    ESP_LOGI(TAG, "速度跟踪模式已禁用");
}
bool is_velocity_tracking_mode_active(void) {
    return velocity_tracking_mode_active;
}
void reset_velocity_tracking_mode(void) {
    velocity_tracking_reset_to_enabled();
    ESP_LOGI(TAG, "速度跟踪模式已重置，等待重新激活");
}
// ========== system_init.h ==========
#ifndef SYSTEM_INIT_H
#define SYSTEM_INIT_H
#include "esp_err.h"
#include "voice_module.h"
#ifdef __cplusplus
extern "C" {
#endif
```

```
esp_err_t system_init_nvs(void);
esp_err_t system_init_wifi(void);
esp_err_t system_init_exoskeleton(void);
esp_err_t system_init_all(void);
extern VoiceModule voiceModule;
#ifdef __cplusplus
}
#endif
#endif
```

// ========== system_init.c ==========

```
#include "system_init.h"
#include "esp_log.h"
#include "nvs_flash.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "wifi_softap_module.h"
#include "rs01_motor.h"
#include "button_detector.h"
#include "voice_module.h"
static const char *TAG = "system_init";
#define DEFAULT_WIFI_SSID      "ESP32_Exoskeleton"
#define DEFAULT_WIFI_PASS      "12345678"
#define DEFAULT_WIFI_CHANNEL   1
#define DEFAULT_MAX_STA_CONN   4
extern MI_Motor motors[];
VoiceModule voiceModule;
static void wifi_event_callback(int32_t event_id, void* event_data)
{
}
static void motor_data_callback(MI_Motor* motor) {
    ESP_LOGI(TAG, "电机%d 数据更新: 位置=%.3f, 速度=%.3f, 电流=%.3f, 温度=%.1f, 模式=%d",
             motor->id, motor->position, motor->velocity, motor->current, motor->temperature, motor->mode);
}
esp_err_t system_init_nvs(void)
{
    ESP_LOGI(TAG, "初始化 NVS 存储...");
    esp_err_t ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }
    ESP_ERROR_CHECK(ret);
    ESP_LOGI(TAG, "NVS 存储初始化完成");
    return ESP_OK;
}
esp_err_t system_init_wifi(void)
{
    ESP_LOGI(TAG, "初始化 WiFi 热点...");
    wifi_softap_config_t config = {
        .ssid = DEFAULT_WIFI_SSID,
```

```
        .password = DEFAULT_WIFI_PASS,
        .channel = DEFAULT_WIFI_CHANNEL,
        .max_connection = DEFAULT_MAX_STA_CONN,
#ifdef CONFIG_ESP_WIFI_SOFTAP_SAE_SUPPORT
        .authmode = WIFI_AUTH_WPA3_PSK,
#else
        .authmode = WIFI_AUTH_WPA2_PSK,
#endif
    };
    ESP_ERROR_CHECK(wifi_softap_init(&config, wifi_event_callback));
    return ESP_OK;
}
esp_err_t system_init_exoskeleton(void)
{
    ESP_LOGI(TAG, "初始化外骨骼控制模块...");
    ESP_LOGI(TAG, "初始化语音模块...");
    voice_module_init(&voiceModule);
    UART_Rx_Init(motor_data_callback);
    vTaskDelay(pdMS_TO_TICKS(500));
    ESP_LOGI(TAG, "语音播报: 启动成功");
    voice_speak(&voiceModule, "启动成功");
    vTaskDelay(pdMS_TO_TICKS(1500));
    ESP_LOGI(TAG, "语音播报完成， 开始设置电机");
    ESP_LOGI(TAG, "设置电机为运控模式...");
    for(int i = 0; i < 2; i++) {
        MI_Motor* motor = &motors[i];
        ESP_LOGI(TAG, "设置电机 %d 上报时间参数...", motor->id);
        Set_SingleParameter(motor, REPORT_TIME, 1.0f);
        vTaskDelay(pdMS_TO_TICKS(50));
        Change_Mode(motor, CTRL_MODE);
        vTaskDelay(pdMS_TO_TICKS(50));
        Motor_Enable(motor);
        vTaskDelay(pdMS_TO_TICKS(50));
        ESP_LOGI(TAG, "开启电机 %d 上报功能...", motor->id);
        Motor_SetReporting(motor, true);
        vTaskDelay(pdMS_TO_TICKS(50));
        ESP_LOGI(TAG, "电机 %d 已设置为运控模式并使能", motor->id);
    }
    ESP_LOGI(TAG, "电机模式设置完成");
    ESP_LOGI(TAG, "初始化按键检测模块...");
    button_init();
    ESP_LOGI(TAG, "启动按键处理任务...");
    xTaskCreate(buttonProcessTask, "ButtonProcess", 4096, NULL, 5, NULL);
    ESP_LOGI(TAG, "外骨骼控制模块初始化完成");
    return ESP_OK;
}
esp_err_t system_init_all(void)
{
    ESP_LOGI(TAG, "开始系统初始化...");
    ESP_LOGI(TAG, "系统启动延时 5 秒...");
```

```
    vTaskDelay(pdMS_TO_TICKS(5000));
    ESP_LOGI(TAG, "延时完成，开始初始化各模块");
    ESP_ERROR_CHECK(system_init_nvs());
    ESP_ERROR_CHECK(system_init_wifi());
    ESP_ERROR_CHECK(system_init_exoskeleton());
    ESP_LOGI(TAG, "系统初始化完成！");
    return ESP_OK;
}
```

// ========== rs01_motor.h ==========

```
#ifndef _MI_MOTOR_H__
#define _MI_MOTOR_H__
#include <stdint.h>
#include <string.h>
#include "driver/uart.h"
#include "driver/gpio.h"
#include <math.h>
#include <string.h>
#define MOTOR_UART_NUM      UART_NUM_1
#define UART_TX_PIN         10
#define UART_RX_PIN         11
#define UART_BAUDRATE       115200
#define CAN_RAW_FRAME_LENGTH 12
#define MASTER_ID       0xFD
#define MOTER_1_ID      1
#define MOTER_2_ID      2
#define P_MIN          -12.57f
#define P_MAX          12.57f
#define V_MIN          -44.0f
#define V_MAX          44.0f
#define KP_MIN         0.0f
#define KP_MAX         500.0f
#define KD_MIN         0.0f
#define KD_MAX         5.0f
#define T_MIN          -17.0f
#define T_MAX          17.0f
#define RUN_MODE       0x7005
#define CTRL_MODE      0
#define POS_MODE_PP    1
#define SPEED_MODE     2
#define CUR_MODE       3
#define POS_MODE_CSP   5
#define IQ_REF         0x7006
#define SPD_REF        0x700A
#define LIMIT_TORQUE   0x700B
#define LOC_REF        0x7016
#define LIMIT_SPD      0x7017
#define LIMIT_CUR      0x7018
#define VELOCITY_FILTER 0x7021
#define REPORT_TIME    0x7026
typedef struct {
```

```
    uint8_t type;
    uint16_t data;
    uint8_t target_id;
    uint8_t payload[8];
} can_frame_t;
typedef struct {
    uint8_t id;
    float position;
    float velocity;
    float current;
    float temperature;
    uint8_t mode;
    uint8_t error;
    bool error_uncalibrated;
    bool error_overload;
    bool error_magnetic_encoder;
    bool error_over_temperature;
    bool error_driver_fault;
    bool error_undervoltage;
} MI_Motor;
typedef void (*MotorDataCallback)(MI_Motor*);
extern uart_config_t motor_uart_config;
extern MI_Motor motors[2];
extern MotorDataCallback data_callback;
void UART_Rx_Init(MotorDataCallback callback);
void handle_uart_rx();
void UART_Send_Frame(const can_frame_t* frame);
void Motor_Enable(MI_Motor* motor);
void Motor_Reset(MI_Motor* motor, uint8_t clear_error);
void Motor_Set_Zero(MI_Motor* motor);
void Motor_ControlMode(MI_Motor* motor, float torque, float position, float speed, float kp, float kd);
void Set_SingleParameter(MI_Motor* motor, uint16_t parameter, float value);
void Set_CurMode(MI_Motor* motor, float current);
void Set_SpeedMode(MI_Motor* motor, float speed, float current_limit);
void Change_Mode(MI_Motor* motor, uint8_t mode);
void Motor_SetReporting(MI_Motor* motor, bool enable);
int float_to_uint(float x, float x_min, float x_max, int bits);
float uint_to_float(int x_int, float x_min, float x_max, int bits);
#endif
```

// ========== rs01_motor.c ==========

```
#include "rs01_motor.h"
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
static const char *TAG = "RS01_MOTOR";
uart_config_t motor_uart_config;
MI_Motor motors[2];
MotorDataCallback data_callback = NULL;
static TickType_t lastPrintTime[2] = {0, 0};
static const TickType_t PRINT_INTERVAL = pdMS_TO_TICKS(2000);
```

```
static void parse_can_frame(const uint8_t* can_payload);
uint16_t float_to_uint16_linear(float value, float min_val, float max_val) {
   if (value < min_val) value = min_val;
   if (value > max_val) value = max_val;
   float center = (min_val + max_val) / 2.0f;
   float half_range = (max_val - min_val) / 2.0f;
   float normalized = (value - center) / half_range;
   return (uint16_t)(32768 + normalized * 32768);
}
int float_to_uint(float x, float x_min, float x_max, int bits) {
   float span = x_max - x_min;
   float offset = x_min;
   if (x > x_max) x = x_max;
   else if (x < x_min) x = x_min;
   return (int)((x - offset) * ((float)((1 << bits) - 1)) / span);
}
float uint_to_float(int x_int, float x_min, float x_max, int bits) {
   float span = x_max - x_min;
   float offset = x_min;
   return ((float)x_int * span / ((float)((1 << bits) - 1))) + offset;
}
void UART_Send_Frame(const can_frame_t* frame) {
   uint32_t extended_id = ((uint32_t)frame->type << 24) |
                  ((uint32_t)frame->data << 8) |
                  frame->target_id;
   uint8_t packet[CAN_RAW_FRAME_LENGTH];
   packet[0] = (extended_id >> 24) & 0xFF;
   packet[1] = (extended_id >> 16) & 0xFF;
   packet[2] = (extended_id >> 8) & 0xFF;
   packet[3] = extended_id & 0xFF;
   memcpy(&packet[4], frame->payload, 8);
   int written = uart_write_bytes(MOTOR_UART_NUM, packet, CAN_RAW_FRAME_LENGTH);
   if (written != CAN_RAW_FRAME_LENGTH) {
      ESP_LOGE(TAG, "UART write failed, expected %d, wrote %d", CAN_RAW_FRAME_LENGTH, written);
   }}
void UART_Rx_Init(MotorDataCallback callback) {
   data_callback = callback;
   motor_uart_config.baud_rate = UART_BAUDRATE;
   motor_uart_config.data_bits = UART_DATA_8_BITS;
   motor_uart_config.parity = UART_PARITY_DISABLE;
   motor_uart_config.stop_bits = UART_STOP_BITS_1;
   motor_uart_config.flow_ctrl = UART_HW_FLOWCTRL_DISABLE;
   motor_uart_config.source_clk = UART_SCLK_DEFAULT;
   ESP_ERROR_CHECK(uart_driver_install(MOTOR_UART_NUM, 1024, 1024, 0, NULL, 0));
   ESP_ERROR_CHECK(uart_param_config(MOTOR_UART_NUM, &motor_uart_config));
   ESP_ERROR_CHECK(uart_set_pin(MOTOR_UART_NUM, UART_TX_PIN, UART_RX_PIN, UART_PIN_NO_CHANGE,
UART_PIN_NO_CHANGE));
   for (int i = 0; i < 2; i++) {
      motors[i].id = (i == 0) ? MOTER_1_ID : MOTER_2_ID;
      motors[i].position = 0.0f;
```

```
        motors[i].velocity = 0.0f;
        motors[i].current = 0.0f;
        motors[i].temperature = 0.0f;
        motors[i].mode = 0;
        motors[i].error = 0;
        motors[i].error_uncalibrated = false;
        motors[i].error_overload = false;
        motors[i].error_magnetic_encoder = false;
        motors[i].error_over_temperature = false;
        motors[i].error_driver_fault = false;
        motors[i].error_undervoltage = false;
    }
    ESP_LOGI(TAG, "UART initialized for motor communication");
}
void handle_uart_rx() {
    static uint8_t packet_buffer[CAN_RAW_FRAME_LENGTH];
    static uint8_t packet_index = 0;
    size_t available_bytes = 0;
    uart_get_buffered_data_len(MOTOR_UART_NUM, &available_bytes);
    while (available_bytes > 0) {
        uint8_t byte;
        int length = uart_read_bytes(MOTOR_UART_NUM, &byte, 1, pdMS_TO_TICKS(1));
        if (length == 1) {
            if (packet_index < CAN_RAW_FRAME_LENGTH) {
                packet_buffer[packet_index++] = byte;
            } else {
                packet_index = 0;
                packet_buffer[packet_index++] = byte;
            }
            if (packet_index >= CAN_RAW_FRAME_LENGTH) {
                parse_can_frame(packet_buffer);
                packet_index = 0;
            }
        }
        uart_get_buffered_data_len(MOTOR_UART_NUM, &available_bytes);
    }
}
static void parse_can_frame(const uint8_t* can_payload) {
    uint32_t extended_id = ((uint32_t)can_payload[0] << 24) |
                           ((uint32_t)can_payload[1] << 16) |
                           ((uint32_t)can_payload[2] << 8) |
                           can_payload[3];
    uint8_t type = (extended_id >> 24) & 0x1F;
    if (type == 0x02 || type == 0x18) {
        uint8_t motor_id = (extended_id >> 8) & 0xFF;
        if (motor_id >= MOTER_1_ID && motor_id <= MOTER_2_ID) {
            MI_Motor* motor = &motors[motor_id - 1];
            motor->id = motor_id;
            const uint8_t* data = &can_payload[4];
            uint16_t raw_angle = (data[0] << 8) | data[1];
```

```
        uint16_t raw_speed = (data[2] << 8) | data[3];
        uint16_t raw_torque = (data[4] << 8) | data[5];
        uint16_t raw_temp = (data[6] << 8) | data[7];
        motor->position = uint_to_float(raw_angle, P_MIN, P_MAX, 16);
        motor->velocity = uint_to_float(raw_speed, V_MIN, V_MAX, 16);
        motor->current = uint_to_float(raw_torque, T_MIN, T_MAX, 16);
        motor->temperature = raw_temp / 10.0f;
        uint32_t status_part = extended_id >> 16;
        motor->error = status_part & 0x3F;
        motor->error_undervoltage     = (status_part >> 0) & 0x01;
        motor->error_driver_fault     = (status_part >> 1) & 0x01;
        motor->error_over_temperature  = (status_part >> 2) & 0x01;
        motor->error_magnetic_encoder  = (status_part >> 3) & 0x01;
        motor->error_overload         = (status_part >> 4) & 0x01;
        motor->error_uncalibrated     = (status_part >> 5) & 0x01;
        motor->mode = (status_part >> 6) & 0x03;
        TickType_t currentTime = xTaskGetTickCount();
        int motorIndex = (motor->id == 1) ? 0 : 1;
        if (currentTime - lastPrintTime[motorIndex] >= PRINT_INTERVAL) {
          ESP_LOGI(TAG, "[Parsed] ID: %d, Pos: %.2f, Vel: %.2f, Cur: %.2f, Temp: %.1f, Mode: %d, Err: 0x%X",
                motor->id, motor->position, motor->velocity, motor->current, motor->temperature, motor->mode, motor->error);
          lastPrintTime[motorIndex] = currentTime;
        }
        if (data_callback) {
          data_callback(motor);
        }
      }
    }
}
void Motor_Enable(MI_Motor* motor) {
    can_frame_t frame;
    frame.type = 0x03;
    frame.target_id = motor->id;
    frame.data = MASTER_ID;
    memset(frame.payload, 0, sizeof(frame.payload));
    UART_Send_Frame(&frame);
    ESP_LOGI(TAG, "Motor %d enabled", motor->id);
}
void Motor_Reset(MI_Motor* motor, uint8_t clear_error) {
    can_frame_t frame;
    frame.type = 0x04;
    frame.target_id = motor->id;
    frame.data = MASTER_ID;
    memset(frame.payload, 0, sizeof(frame.payload));
    if (clear_error) {
      frame.payload[0] = 1;
    }
    UART_Send_Frame(&frame);
    ESP_LOGI(TAG, "Motor %d reset", motor->id);
}
```

```
void Motor_Set_Zero(MI_Motor* motor) {
    can_frame_t frame;
    frame.type = 0x06;
    frame.target_id = motor->id;
    frame.data = MASTER_ID;
    memset(frame.payload, 0, sizeof(frame.payload));
    frame.payload[0] = 1;
    UART_Send_Frame(&frame);
    ESP_LOGI(TAG, "Motor %d set zero position", motor->id);
}
void Set_CurMode(MI_Motor* motor, float current) {
    Set_SingleParameter(motor, IQ_REF, current);
}
void Set_SpeedMode(MI_Motor* motor, float speed, float current_limit) {
    Set_SingleParameter(motor, SPD_REF, speed);
    Set_SingleParameter(motor, LIMIT_CUR, current_limit);
}
void Change_Mode(MI_Motor* motor, uint8_t mode) {
    can_frame_t frame;
    frame.type = 0x12;
    frame.target_id = motor->id;
    frame.data = MASTER_ID;
    memset(frame.payload, 0, sizeof(frame.payload));
    uint16_t index = RUN_MODE;
    frame.payload[0] = index & 0xFF;
    frame.payload[1] = (index >> 8) & 0xFF;
    frame.payload[4] = mode;
    UART_Send_Frame(&frame);
}
void Set_SingleParameter(MI_Motor* motor, uint16_t parameter, float value) {
    can_frame_t frame;
    frame.type = 0x12;
    frame.data = MASTER_ID;
    frame.target_id = motor->id;
    memset(frame.payload, 0, sizeof(frame.payload));
    frame.payload[0] = parameter & 0xFF;
    frame.payload[1] = (parameter >> 8) & 0xFF;
    memcpy(&frame.payload[4], &value, sizeof(float));
    UART_Send_Frame(&frame);
}
void Motor_SetReporting(MI_Motor* motor, bool enable) {
    can_frame_t frame;
    frame.type = 0x18;
    frame.data = MASTER_ID;
    frame.target_id = motor->id;
    frame.payload[0] = 0x01;
    frame.payload[1] = 0x02;
    frame.payload[2] = 0x03;
    frame.payload[3] = 0x04;
    frame.payload[4] = 0x05;
```

```
    frame.payload[5] = 0x06;
    frame.payload[6] = enable ? 0x01 : 0x00;
    frame.payload[7] = 0x00;
    UART_Send_Frame(&frame);
}
void Motor_ControlMode(MI_Motor* motor, float torque, float position, float speed, float kp, float kd) {
    can_frame_t frame;
    uint16_t torque_cmd = float_to_uint16_linear(torque, T_MIN, T_MAX);
    uint16_t pos_cmd = float_to_uint16_linear(position, P_MIN, P_MAX);
    uint16_t vel_cmd = float_to_uint16_linear(speed, V_MIN, V_MAX);
    uint16_t kp_cmd = float_to_uint16_linear(kp, KP_MIN, KP_MAX);
    uint16_t kd_cmd = float_to_uint16_linear(kd, KD_MIN, KD_MAX);
    frame.type = 0x01;
    frame.data = torque_cmd;
    frame.target_id = motor->id;
    frame.payload[0] = (pos_cmd >> 8) & 0xFF;
    frame.payload[1] = pos_cmd & 0xFF;
    frame.payload[2] = (vel_cmd >> 8) & 0xFF;
    frame.payload[3] = vel_cmd & 0xFF;
    frame.payload[4] = (kp_cmd >> 8) & 0xFF;
    frame.payload[5] = kp_cmd & 0xFF;
    frame.payload[6] = (kd_cmd >> 8) & 0xFF;
    frame.payload[7] = kd_cmd & 0xFF;
    UART_Send_Frame(&frame);
}
```

// ========== motor_web_control.h ==========

```
#ifndef MOTOR_WEB_CONTROL_H
#define MOTOR_WEB_CONTROL_H
#include "esp_http_server.h"
#include "esp_err.h"
#ifdef __cplusplus
extern "C" {
#endif
esp_err_t motor_web_control_init(void);
httpd_handle_t motor_web_control_start_server(void);
esp_err_t motor_web_control_stop_server(httpd_handle_t server);
esp_err_t motor_web_index_handler(httpd_req_t *req);
esp_err_t motor_web_control_api_handler(httpd_req_t *req);
esp_err_t motor_web_status_api_handler(httpd_req_t *req);
esp_err_t motor_web_params_api_handler(httpd_req_t *req);
#ifdef __cplusplus
}
#endif
#endif
```

// ========== motor_web_control.c ==========

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/param.h>
#include "motor_web_control.h"
```

```
#include "motor_web_html.h"
#include "esp_http_server.h"
#include "esp_log.h"
#include "cJSON.h"
#include "esp_wifi.h"
#include "alternating_speed.h"
#include "velocity_tracking_mode.h"
extern float motor_target_speed[2];
extern bool motor_speed_control_enabled;
extern motor_control_params_t motor_params[2];
extern bool motion_mode_detection_enabled;
extern bool alternating_speed_enabled;
extern int alternating_interval_ms;
extern float alternating_speed_x;
extern float alternating_speed_y;
extern float speed_current_limit;
extern float feedforward_torque_value;
extern float climbing_mode_torque;
extern bool velocity_tracking_mode_enabled;
extern void enable_velocity_tracking_mode(void);
extern void disable_velocity_tracking_mode(void);
extern bool is_velocity_tracking_mode_active(void);
extern void reset_velocity_tracking_mode(void);
extern float velocity_tracking_lift_leg_torque;
extern float velocity_tracking_lift_leg_speed;
extern float velocity_tracking_drop_leg_torque;
extern float velocity_tracking_drop_leg_speed;
extern uint32_t velocity_tracking_lift_leg_max_duration;
extern uint32_t velocity_tracking_lift_leg_fixed_duration_ms;
extern uint32_t velocity_tracking_drop_leg_delay_ms;
extern uint32_t velocity_tracking_drop_leg_fixed_duration_ms;
extern uint32_t velocity_tracking_default_cycle_duration_ms;
extern float velocity_tracking_enable_threshold;
extern float velocity_tracking_min_velocity;
#include "rs01_motor.h"
extern MI_Motor motors[2];
static const char *TAG = "motor_web_control";
static httpd_handle_t server = NULL;
esp_err_t motor_web_index_handler(httpd_req_t *req)
{
   httpd_resp_set_type(req, "text/html");
   httpd_resp_send(req, motor_control_html, HTTPD_RESP_USE_STRLEN);
   return ESP_OK;
}
esp_err_t motor_web_status_api_handler(httpd_req_t *req)
{
   cJSON *json = cJSON_CreateObject();
   cJSON *motor1 = cJSON_CreateObject();
   cJSON *motor2 = cJSON_CreateObject();
   cJSON_AddNumberToObject(motor1, "position", motors[0].position);
```

```
cJSON_AddNumberToObject(motor1, "velocity", motors[0].velocity);
cJSON_AddNumberToObject(motor1, "current", motors[0].current);
cJSON_AddNumberToObject(motor1, "temperature", motors[0].temperature);
cJSON_AddItemToObject(json, "motor1", motor1);
cJSON_AddNumberToObject(motor2, "position", motors[1].position);
cJSON_AddNumberToObject(motor2, "velocity", motors[1].velocity);
cJSON_AddNumberToObject(motor2, "current", motors[1].current);
cJSON_AddNumberToObject(motor2, "temperature", motors[1].temperature);
cJSON_AddItemToObject(json, "motor2", motor2);
cJSON_AddBoolToObject(json, "motor_control_enabled", motor_speed_control_enabled);
cJSON_AddBoolToObject(json, "alternating_enabled", alternating_speed_enabled);
cJSON_AddBoolToObject(json, "motion_detection_enabled", motion_mode_detection_enabled);
cJSON_AddNumberToObject(json, "walking_mode", Get_Current_Walking_Mode());
cJSON_AddNumberToObject(json, "feedforward_torque", feedforward_torque_value);
cJSON_AddNumberToObject(json, "climbing_torque", climbing_mode_torque);
cJSON_AddNumberToObject(json, "vt_lift_torque", velocity_tracking_lift_leg_torque);
cJSON_AddNumberToObject(json, "vt_lift_speed", velocity_tracking_lift_leg_speed);
cJSON_AddNumberToObject(json, "vt_drop_torque", velocity_tracking_drop_leg_torque);
cJSON_AddNumberToObject(json, "vt_drop_speed", velocity_tracking_drop_leg_speed);
cJSON_AddNumberToObject(json, "vt_lift_fixed_duration", velocity_tracking_lift_leg_fixed_duration_ms);
cJSON_AddNumberToObject(json, "vt_drop_delay", velocity_tracking_drop_leg_delay_ms);
cJSON_AddNumberToObject(json, "vt_drop_fixed_duration", velocity_tracking_drop_leg_fixed_duration_ms);
cJSON_AddNumberToObject(json, "vt_default_cycle_duration", velocity_tracking_default_cycle_duration_ms);
cJSON_AddNumberToObject(json, "vt_enable_threshold", velocity_tracking_enable_threshold);
cJSON_AddNumberToObject(json, "vt_min_velocity", velocity_tracking_min_velocity);
cJSON_AddBoolToObject(json, "velocity_tracking_enabled", velocity_tracking_mode_enabled);
if (velocity_tracking_mode_enabled) {
    velocity_tracking_state_t state = velocity_tracking_mode_get_state();
    const char* state_str = "未知";
    switch (state) {
        case VELOCITY_TRACKING_DISABLED:
            state_str = "禁用";
            break;
        case VELOCITY_TRACKING_ENABLED:
            state_str = "启用";
            break;
        case VELOCITY_TRACKING_ACTIVE:
            state_str = "激活";
            break;
    }
    cJSON_AddStringToObject(json, "velocity_tracking_state", state_str);
} else {
    cJSON_AddStringToObject(json, "velocity_tracking_state", "禁用");
}
cJSON_AddBoolToObject(json, "alternating_conflict", motion_mode_detection_enabled);
cJSON_AddBoolToObject(json, "motion_detection_conflict", alternating_speed_enabled);
char *json_string = cJSON_Print(json);
httpd_resp_set_type(req, "application/json");
httpd_resp_send(req, json_string, strlen(json_string));
free(json_string);
```

```
    cJSON_Delete(json);
    return ESP_OK;
}
esp_err_t motor_web_control_api_handler(httpd_req_t *req)
{
    char buf[1000];
    int ret, remaining = req->content_len;
    if (remaining >= sizeof(buf)) {
        httpd_resp_send_err(req, HTTPD_400_BAD_REQUEST, "Content too long");
        return ESP_FAIL;
    }
    ret = httpd_req_recv(req, buf, MIN(remaining, sizeof(buf)));
    if (ret <= 0) {
        if (ret == HTTPD_SOCK_ERR_TIMEOUT) {
            httpd_resp_send_408(req);
        }
        return ESP_FAIL;
    }
    buf[ret] = '\0';
    cJSON *json = cJSON_Parse(buf);
    if (json == NULL) {
        httpd_resp_send_err(req, HTTPD_400_BAD_REQUEST, "Invalid JSON");
        return ESP_FAIL;
    }
    cJSON *action = cJSON_GetObjectItem(json, "action");
    if (!cJSON_IsString(action)) {
        cJSON_Delete(json);
        httpd_resp_send_err(req, HTTPD_400_BAD_REQUEST, "Missing action");
        return ESP_FAIL;
    }
    const char *response_msg = "Unknown action";
    if (strcmp(action->valuestring, "start") == 0) {
        motor_speed_control_enabled = true;
        response_msg = "电机控制已启动";
        ESP_LOGI(TAG, "网页请求启动电机控制");
    }
    else if (strcmp(action->valuestring, "stop") == 0) {
        motor_speed_control_enabled = false;
        Stop_Alternating_Speed();
        motion_mode_detection_enabled = false;
        motor_target_speed[0] = 0;
        motor_target_speed[1] = 0;
        response_msg = "电机控制已停止， 所有模式已关闭";
        ESP_LOGI(TAG, "网页请求停止电机控制， 关闭所有模式");
    }
    else if (strcmp(action->valuestring, "emergency") == 0) {
        motor_speed_control_enabled = false;
        Stop_Alternating_Speed();
        motion_mode_detection_enabled = false;
        motor_target_speed[0] = 0;
```

```
    motor_target_speed[1] = 0;
    response_msg = "紧急停止已执行，所有模式已关闭";
    ESP_LOGW(TAG, "网页请求紧急停止，关闭所有模式");
  }
  else if (strcmp(action->valuestring, "mode") == 0) {
    cJSON *mode = cJSON_GetObjectItem(json, "mode");
    if (cJSON_IsNumber(mode)) {
      if (mode->valueint == 0) {
        Switch_To_Flat_Mode();
        response_msg = "已切换到平地模式";
      } else if (mode->valueint == 1) {
        Switch_To_Stairs_Mode();
        response_msg = "已切换到爬楼模式";
      }
      ESP_LOGI(TAG, "网页请求切换行走模式: %d", mode->valueint);
    }
  }
  else if (strcmp(action->valuestring, "start_alternating") == 0) {
    if (motion_mode_detection_enabled) {
      response_msg = "启动失败：智能运动控制正在运行，请先关闭智能运动控制";
      ESP_LOGW(TAG, "交替行走启动失败：智能运动控制冲突");
    } else {
      Start_Alternating_Speed();
      response_msg = "交替行走模式已启动";
      ESP_LOGI(TAG, "网页请求启动交替行走");
    }
  }
  else if (strcmp(action->valuestring, "stop_alternating") == 0) {
    Stop_Alternating_Speed();
    response_msg = "交替行走模式已停止";
    ESP_LOGI(TAG, "网页请求停止交替行走");
  }
  else if (strcmp(action->valuestring, "enable_motion_detection") == 0) {
    if (alternating_speed_enabled) {
      response_msg = "启动失败：交替行走模式正在运行，请先停止交替行走";
      ESP_LOGW(TAG, "智能运动控制启动失败：交替行走冲突");
    } else {
      motion_mode_detection_enabled = true;
      response_msg = "智能运动模式检测已启用";
      ESP_LOGI(TAG, "网页请求启用运动模式检测");
    }
  }
  else if (strcmp(action->valuestring, "disable_motion_detection") == 0) {
    motion_mode_detection_enabled = false;
    response_msg = "智能运动模式检测已关闭";
    ESP_LOGI(TAG, "网页请求关闭运动模式检测");
  }
  else if (strcmp(action->valuestring, "enable_velocity_tracking") == 0) {
    if (alternating_speed_enabled) {
      response_msg = "启动失败：交替行走模式正在运行，请先停止交替行走";
```

```
            ESP_LOGW(TAG, "速度跟踪模式启动失败：交替行走冲突");
        } else if (motion_mode_detection_enabled) {
            response_msg = "启动失败：智能运动检测正在运行，请先关闭智能运动检测";
            ESP_LOGW(TAG, "速度跟踪模式启动失败：智能运动检测冲突");
        } else {
            enable_velocity_tracking_mode();
            response_msg = "速度跟踪模式已启用，等待波峰波谷差值>=0.75 激活";
            ESP_LOGI(TAG, "网页请求启用速度跟踪模式");
        }
    }
    else if (strcmp(action->valuestring, "disable_velocity_tracking") == 0) {
        disable_velocity_tracking_mode();
        response_msg = "速度跟踪模式已关闭";
        ESP_LOGI(TAG, "网页请求关闭速度跟踪模式");
    }
    else if (strcmp(action->valuestring, "reset_velocity_tracking") == 0) {
        if (velocity_tracking_mode_enabled) {
            reset_velocity_tracking_mode();
            response_msg = "速度跟踪模式已重置到等待激活状态";
            ESP_LOGI(TAG, "网页请求重置速度跟踪模式");
        } else {
            response_msg = "重置失败：速度跟踪模式未启用";
            ESP_LOGW(TAG, "速度跟踪模式重置失败：模式未启用");
        }
    }
    httpd_resp_send(req, response_msg, strlen(response_msg));
    cJSON_Delete(json);
    return ESP_OK;
}
esp_err_t motor_web_params_api_handler(httpd_req_t *req)
{
    char buf[1000];
    int ret, remaining = req->content_len;
    if (remaining >= sizeof(buf)) {
        httpd_resp_send_err(req, HTTPD_400_BAD_REQUEST, "Content too long");
        return ESP_FAIL;
    }
    ret = httpd_req_recv(req, buf, MIN(remaining, sizeof(buf)));
    if (ret <= 0) {
        if (ret == HTTPD_SOCK_ERR_TIMEOUT) {
            httpd_resp_send_408(req);
        }
        return ESP_FAIL;
    }
    buf[ret] = '\0';
    cJSON *json = cJSON_Parse(buf);
    if (json == NULL) {
        httpd_resp_send_err(req, HTTPD_400_BAD_REQUEST, "Invalid JSON");
        return ESP_FAIL;
    }
```

```
cJSON *action = cJSON_GetObjectItem(json, "action");
if (!cJSON_IsString(action)) {
    cJSON_Delete(json);
    httpd_resp_send_err(req, HTTPD_400_BAD_REQUEST, "Missing action");
    return ESP_FAIL;
}
const char *response_msg = "未知操作";
if (strcmp(action->valuestring, "update_params") == 0) {
    cJSON *motor1 = cJSON_GetObjectItem(json, "motor1");
    cJSON *motor2 = cJSON_GetObjectItem(json, "motor2");
    if (motor1) {
        cJSON *pos = cJSON_GetObjectItem(motor1, "position");
        cJSON *speed = cJSON_GetObjectItem(motor1, "speed");
        cJSON *torque = cJSON_GetObjectItem(motor1, "torque");
        cJSON *kp = cJSON_GetObjectItem(motor1, "kp");
        cJSON *kd = cJSON_GetObjectItem(motor1, "kd");
        if (pos) motor_params[0].position = pos->valuedouble;
        if (speed) motor_params[0].speed = speed->valuedouble;
        if (torque) motor_params[0].torque = torque->valuedouble;
        if (kp) motor_params[0].kp = kp->valuedouble;
        if (kd) motor_params[0].kd = kd->valuedouble;
    }
    if (motor2) {
        cJSON *pos = cJSON_GetObjectItem(motor2, "position");
        cJSON *speed = cJSON_GetObjectItem(motor2, "speed");
        cJSON *torque = cJSON_GetObjectItem(motor2, "torque");
        cJSON *kp = cJSON_GetObjectItem(motor2, "kp");
        cJSON *kd = cJSON_GetObjectItem(motor2, "kd");
        if (pos) motor_params[1].position = pos->valuedouble;
        if (speed) motor_params[1].speed = speed->valuedouble;
        if (torque) motor_params[1].torque = torque->valuedouble;
        if (kp) motor_params[1].kp = kp->valuedouble;
        if (kd) motor_params[1].kd = kd->valuedouble;
    }
    extern void unified_motor_control(int motor_id, const motor_control_params_t* params);
    unified_motor_control(0, &motor_params[0]);
    unified_motor_control(1, &motor_params[1]);
    response_msg = "运控参数已更新";
    ESP_LOGI(TAG, "网页更新运控参数");
}
else if (strcmp(action->valuestring, "update_alternating") == 0) {
    cJSON *interval = cJSON_GetObjectItem(json, "interval");
    cJSON *speedX = cJSON_GetObjectItem(json, "speedX");
    cJSON *speedY = cJSON_GetObjectItem(json, "speedY");
    cJSON *currentLimit = cJSON_GetObjectItem(json, "currentLimit");
    cJSON *feedforwardTorque = cJSON_GetObjectItem(json, "feedforwardTorque");
    if (interval) alternating_interval_ms = interval->valueint;
    if (speedX) alternating_speed_x = speedX->valuedouble;
    if (speedY) alternating_speed_y = speedY->valuedouble;
    if (currentLimit) speed_current_limit = currentLimit->valuedouble;
```

```
    if (feedforwardTorque) feedforward_torque_value = feedforwardTorque->valuedouble;
    response_msg = "交替行走参数已更新";
    ESP_LOGI(TAG, "网页更新交替行走参数， 前馈力矩: %.2f", feedforward_torque_value);
  }
  else if (strcmp(action->valuestring, "update_motion_detection") == 0) {
    cJSON *climbingTorque = cJSON_GetObjectItem(json, "climbingTorque");
    if (climbingTorque) {
      climbing_mode_torque = climbingTorque->valuedouble;
    }
    response_msg = "智能运动检测参数已更新";
    ESP_LOGI(TAG, "网页更新智能运动检测参数， 爬楼力矩: %.2f", climbing_mode_torque);
  }
  else if (strcmp(action->valuestring, "update_velocity_tracking") == 0) {
    cJSON *liftTorque = cJSON_GetObjectItem(json, "liftTorque");
    cJSON *liftSpeed = cJSON_GetObjectItem(json, "liftSpeed");
    cJSON *dropTorque = cJSON_GetObjectItem(json, "dropTorque");
    cJSON *dropSpeed = cJSON_GetObjectItem(json, "dropSpeed");
    cJSON *liftFixedDuration = cJSON_GetObjectItem(json, "liftFixedDuration");
    cJSON *dropDelay = cJSON_GetObjectItem(json, "dropDelay");
    cJSON *dropFixedDuration = cJSON_GetObjectItem(json, "dropFixedDuration");
    cJSON *defaultCycleDuration = cJSON_GetObjectItem(json, "defaultCycleDuration");
    cJSON *enableThreshold = cJSON_GetObjectItem(json, "enableThreshold");
    cJSON *minVelocity = cJSON_GetObjectItem(json, "minVelocity");
    if (liftTorque) {
      velocity_tracking_lift_leg_torque = liftTorque->valuedouble;
    }
    if (liftSpeed) {
      velocity_tracking_lift_leg_speed = liftSpeed->valuedouble;
    }
    if (dropTorque) {
      velocity_tracking_drop_leg_torque = dropTorque->valuedouble;
    }
    if (dropSpeed) {
      velocity_tracking_drop_leg_speed = dropSpeed->valuedouble;
    }
    if (liftFixedDuration) {
      velocity_tracking_lift_leg_fixed_duration_ms = (uint32_t)liftFixedDuration->valuedouble;
    }
    if (dropDelay) {
      velocity_tracking_drop_leg_delay_ms = (uint32_t)dropDelay->valuedouble;
    }
    if (dropFixedDuration) {
      velocity_tracking_drop_leg_fixed_duration_ms = (uint32_t)dropFixedDuration->valuedouble;
    }
    if (defaultCycleDuration) {
      velocity_tracking_default_cycle_duration_ms = (uint32_t)defaultCycleDuration->valuedouble;
    }
    if (enableThreshold) {
      velocity_tracking_enable_threshold = enableThreshold->valuedouble;
    }
```

```
        if (minVelocity) {
            velocity_tracking_min_velocity = minVelocity->valuedouble;
        }
        response_msg = "Velocity Tracking 参数已更新";
        ESP_LOGI(TAG, "网页更新 Velocity Tracking 参数: 抬腿力矩=%.1f, 抬腿速度=%.2f, 放腿力矩=%.1f, 放腿速度=%.2f",
                 velocity_tracking_lift_leg_torque, velocity_tracking_lift_leg_speed,
                 velocity_tracking_drop_leg_torque, velocity_tracking_drop_leg_speed);
        ESP_LOGI(TAG, "时长参数: 固定抬腿=%lu ms, 压腿延时=%lu ms, 固定压腿=%lu ms, 默认周期=%lu ms",
                 velocity_tracking_lift_leg_fixed_duration_ms, velocity_tracking_drop_leg_delay_ms,
                 velocity_tracking_drop_leg_fixed_duration_ms, velocity_tracking_default_cycle_duration_ms);
        ESP_LOGI(TAG, "阈值参数: 启用阈值=%.2f, 最小速度=%.2f",
                 velocity_tracking_enable_threshold, velocity_tracking_min_velocity);
    }
    httpd_resp_send(req, response_msg, strlen(response_msg));
    cJSON_Delete(json);
    return ESP_OK;
}
httpd_handle_t motor_web_control_start_server(void)
{
    httpd_config_t config = HTTPD_DEFAULT_CONFIG();
    config.lru_purge_enable = true;
    ESP_LOGI(TAG, "启动 HTTP 服务器，端口: %d", config.server_port);
    if (httpd_start(&server, &config) == ESP_OK) {
        httpd_uri_t index_uri = {
            .uri      = "/",
            .method   = HTTP_GET,
            .handler  = motor_web_index_handler,
            .user_ctx  = NULL
        };
        httpd_register_uri_handler(server, &index_uri);
        httpd_uri_t status_uri = {
            .uri      = "/api/status",
            .method   = HTTP_GET,
            .handler  = motor_web_status_api_handler,
            .user_ctx  = NULL
        };
        httpd_register_uri_handler(server, &status_uri);
        httpd_uri_t control_uri = {
            .uri      = "/api/control",
            .method   = HTTP_POST,
            .handler  = motor_web_control_api_handler,
            .user_ctx  = NULL
        };
        httpd_register_uri_handler(server, &control_uri);
        httpd_uri_t params_uri = {
            .uri      = "/api/params",
            .method   = HTTP_POST,
            .handler  = motor_web_params_api_handler,
            .user_ctx  = NULL
        };
```

```
    httpd_register_uri_handler(server, &params_uri);
    ESP_LOGI(TAG, "HTTP 服务器启动成功");
    return server;
  }
  ESP_LOGE(TAG, "HTTP 服务器启动失败");
  return NULL;
}
esp_err_t motor_web_control_stop_server(httpd_handle_t server)
{
  if (server != NULL) {
    ESP_LOGI(TAG, "停止 HTTP 服务器");
    return httpd_stop(server);
  }
  return ESP_OK;
}
esp_err_t motor_web_control_init(void)
{
  ESP_LOGI(TAG, "初始化电机控制网页模块");
  server = motor_web_control_start_server();
  if (server == NULL) {
    ESP_LOGE(TAG, "电机控制网页模块初始化失败");
    return ESP_FAIL;
  }
  ESP_LOGI(TAG, "电机控制网页模块初始化成功");
  return ESP_OK;}
// ========== motor_web_html.h ==========
#ifndef MOTOR_WEB_HTML_H
#define MOTOR_WEB_HTML_H
#ifdef __cplusplus
extern "C" {
#endif
static const char motor_control_html[] = "<!DOCTYPE html>"
"<html><head>"
"<meta charset='UTF-8'>"
"<meta name='viewport' content='width=device-width, initial-scale=1.0'>"
"<title>外骨骼电机控制</title>"
"<style>"
"body{font-family:Arial,sans-serif;margin:20px;background-color:#f5f5f5}"
".container{max-width:800px;margin:0 auto;background:#fff;padding:20px;border-radius:10px;box-shadow:0 2px 10px
rgba(0,0,0,0.1)}"
"h1{color:#333;text-align:center;margin-bottom:30px}"
".section{margin-bottom:25px;padding:15px;border:1px solid #ddd;border-radius:5px}"
".section h3{color:#555;margin-top:0}"
".control-group{display:flex;gap:10px;margin-bottom:10px;align-items:center}"
".control-group label{min-width:120px;font-weight:bold}"
".control-group input{flex:1;padding:8px;border:1px solid #ccc;border-radius:4px}"
".control-group button{padding:8px 15px;background:#007bff;color:white;border:none;border-radius:4px;cursor:pointer}"
".control-group button:hover{background:#0056b3}"
".motor-status{display:grid;grid-template-columns:1fr 1fr;gap:20px}"
".status-card{padding:15px;background:#f8f9fa;border-radius:5px;border-left:4px solid #007bff}"
```

```
".status-item{margin-bottom:5px}"
".btn{padding:10px 20px;margin:5px;border:none;border-radius:5px;cursor:pointer;font-size:14px}"
".btn-primary{background:#007bff;color:white}"
".btn-success{background:#28a745;color:white}"
".btn-danger{background:#dc3545;color:white}"
".btn-warning{background:#ffc107;color:black}"
".btn:hover{opacity:0.8}"
"</style>"
"</head><body>"
"<div class='container'>"
"<h1>                    </h1>"
"<div class='section'>"
"<h3>              </h3>"
"<button onclick='updateStatus()' class='btn btn-primary'>刷新状态</button>"
"<div class='motor-status' id='motorStatus'>加载中...</div>"
"</div>"
"<div class='section'>"
"<h3>          </h3>"
"<div class='control-group'>"
"<button onclick='setWalkingMode(0)' class='btn btn-primary'>平地模式</button>"
"<button onclick='setWalkingMode(1)' class='btn btn-warning'>爬楼模式</button>"
"</div>"
"<div class='control-group'>"
"<button onclick='startAlternating()' class='btn btn-success'>启动交替行走</button>"
"<button onclick='stopAlternating()' class='btn btn-danger'>停止交替行走</button>"
"</div>"
"<div class='control-group'>"
"<button onclick='enableMotionDetection()' class='btn btn-success' id='motionDetectionBtn'>启用智能运动检测</button>"
"<span id='motionDetectionStatus' style='margin-left:10px;font-weight:bold;color:#666;'>状态： 关闭</span>"
"</div>"
"<div class='control-group'>"
"<button onclick='toggleVelocityTracking()' class='btn btn-success' id='velocityTrackingBtn'>启用速度跟踪模式</button>"
"<span id='velocityTrackingStatus' style='margin-left:10px;font-weight:bold;color:#666;'>状态： 关闭</span>"
"</div>"
"<div class='control-group'>"
"<button onclick='resetVelocityTracking()' class='btn btn-warning' id='resetVelocityBtn' disabled>重置节律控制</button>"
"<span style='margin-left:10px;font-size:12px;color:#888;'>重置到等待激活状态</span>"
"</div>"
"</div>"
"<div class='section'>"
"<h3>        运控参数设置</h3>"
"<div class='control-group'>"
"<label>电机 1 位置(rad):</label>"
"<input type='number' id='pos1' step='0.1' value='0'>"
"<label>电机 2 位置(rad):</label>"
"<input type='number' id='pos2' step='0.1' value='0'>"
"</div>"
"<div class='control-group'>"
"<label>电机 1 速度(rad/s):</label>"
"<input type='number' id='speed1' step='0.1' value='0'>"
```

```
"<label>电机 2 速度(rad/s):</label>"
"<input type='number' id='speed2' step='0.1' value='0'>"
"</div>"
"<div class='control-group'>"
"<label>电机 1 力矩(Nm):</label>"
"<input type='number' id='torque1' step='0.1' value='0'>"
"<label>电机 2 力矩(Nm):</label>"
"<input type='number' id='torque2' step='0.1' value='0'>"
"</div>"
"<div class='control-group'>"
"<label>电机 1 Kp:</label>"
"<input type='number' id='kp1' step='0.1' value='0'>"
"<label>电机 2 Kp:</label>"
"<input type='number' id='kp2' step='0.1' value='0'>"
"</div>"
"<div class='control-group'>"
"<label>电机 1 Kd:</label>"
"<input type='number' id='kd1' step='0.1' value='0'>"
"<label>电机 2 Kd:</label>"
"<input type='number' id='kd2' step='0.1' value='0'>"
"</div>"
"<button onclick='updateParams()' class='btn btn-primary'>更新参数</button>"
"</div>"
"<div class='section'>"
"<h3>              </h3>"
"<div class='control-group'>"
"<label>交替间隔(ms):</label>"
"<input type='number' id='interval' value='800'>"
"</div>"
"<div class='control-group'>"
"<label>速度 X(rad/s):</label>"
"<input type='number' id='speedX' step='0.1' value='2.25'>"
"<label>速度 Y(rad/s):</label>"
"<input type='number' id='speedY' step='0.1' value='1.5'>"
"</div>"
"<div class='control-group'>"
"<label>电流限制(A):</label>"
"<input type='number' id='currentLimit' step='0.1' value='2.0'>"
"</div>"
"<div class='control-group'>"
"<label>前馈力矩(Nm):</label>"
"<input type='number' id='feedforwardTorque' step='0.1' value='0.0'>"
"</div>"
"<button onclick='updateAlternatingParams()' class='btn btn-primary'>更新交替参数</button>"
"</div>"
"<div class='section'>"
"<h3>                </h3>"
"<div class='control-group'>"
"<label>爬楼力矩(Nm):</label>"
"<input type='number' id='climbingTorque' step='0.1' value='6.0'>"
```

```
"</div>"
"<button onclick='updateMotionDetectionParams()' class='btn btn-primary'>更新智能检测参数</button>"
"</div>"
"<div class='section'>"
"<h3>                </h3>"
"<div class='control-group'>"
"<label>抬腿力矩(Nm):</label>"
"<input type='number' id='liftTorque' step='0.1' value='6.0'>"
"<label>抬腿速度(rad/s):</label>"
"<input type='number' id='liftSpeed' step='0.1' value='2.75'>"
"</div>"
"<div class='control-group'>"
"<label>放腿力矩(Nm):</label>"
"<input type='number' id='dropTorque' step='0.1' value='-1.0'>"
"<label>放腿速度(rad/s):</label>"
"<input type='number' id='dropSpeed' step='0.1' value='-1.5'>"
"</div>"
"<div class='control-group'>"
"<label>固定抬腿时长(ms):</label>"
"<input type='number' id='liftFixedDuration' step='10' value='600' min='100' max='2000'>"
"<label>压腿延时(ms):</label>"
"<input type='number' id='dropDelay' step='10' value='100' min='0' max='500'>"
"</div>"
"<div class='control-group'>"
"<label>固定压腿时长(ms):</label>"
"<input type='number' id='dropFixedDuration' step='10' value='600' min='100' max='2000'>"
"<label>默认周期时长(ms):</label>"
"<input type='number' id='defaultCycleDuration' step='100' value='2000' min='1000' max='5000'>"
"</div>"
"<div class='control-group'>"
"<label>启用阈值:</label>"
"<input type='number' id='enableThreshold' step='0.05' value='0.65' min='0.1' max='2.0'>"
"<label>最小速度阈值:</label>"
"<input type='number' id='minVelocity' step='0.05' value='0.25' min='0.1' max='1.0'>"
"</div>"
"<button onclick='updateVelocityTrackingParams()' class='btn btn-primary'>更新速度跟踪参数</button>"
"</div>"
"</div>"
"<script>"
"function updateStatus(){"
"fetch('/api/status').then(r=>r.json()).then(data=>{"
"document.getElementById('motorStatus').innerHTML="
"'<div class=\"status-card\"><h4>电机 1</h4>'"
"+'<div class=\"status-item\">位置: '+data.motor1.position.toFixed(3)+' rad</div>'"
"+'<div class=\"status-item\">速度: '+data.motor1.velocity.toFixed(3)+' rad/s</div>'"
"+'<div class=\"status-item\">电流: '+data.motor1.current.toFixed(3)+' A</div>'"
"+'<div class=\"status-item\">温度: '+data.motor1.temperature.toFixed(1)+' °C</div>'"
"+'</div>'"
"+'<div class=\"status-card\"><h4>电机 2</h4>'"
"+'<div class=\"status-item\">位置: '+data.motor2.position.toFixed(3)+' rad</div>'"
```

```
"+'<div class=\"status-item\">速度: '+data.motor2.velocity.toFixed(3)+' rad/s</div>'"
"+'<div class=\"status-item\">电流: '+data.motor2.current.toFixed(3)+' A</div>'"
"+'<div class=\"status-item\">温度: '+data.motor2.temperature.toFixed(1)+' °C</div>'"
"+'</div>';"
"const btn=document.getElementById('motionDetectionBtn');"
"const status=document.getElementById('motionDetectionStatus');"
"if(data.motion_detection_enabled){"
"btn.textContent='关闭智能运动检测';"
"btn.className='btn btn-danger';"
"status.textContent='状态：启用';"
"status.style.color='#28a745';"
"}else{"
"btn.textContent='启用智能运动检测';"
"btn.className='btn btn-success';"
"status.textContent='状态：关闭';"
"status.style.color='#666';"
"}"
"const vBtn=document.getElementById('velocityTrackingBtn');"
"const vStatus=document.getElementById('velocityTrackingStatus');"
"const resetBtn=document.getElementById('resetVelocityBtn');"
"if(data.velocity_tracking_enabled){"
"vBtn.textContent='关闭速度跟踪模式';"
"vBtn.className='btn btn-danger';"
"resetBtn.disabled=false;"
"if(data.velocity_tracking_state==='激活'){"
"vStatus.textContent='状态：激活中';"
"vStatus.style.color='#28a745';"
"}else if(data.velocity_tracking_state==='启用'){"
"vStatus.textContent='状态：等待激活';"
"vStatus.style.color='#ffc107';"
"}else{"
"vStatus.textContent='状态：启用';"
"vStatus.style.color='#17a2b8';"
"}"
"}else{"
"vBtn.textContent='启用速度跟踪模式';"
"vBtn.className='btn btn-success';"
"vStatus.textContent='状态：关闭';"
"vStatus.style.color='#666';"
"resetBtn.disabled=true;"
"}"
"if(data.feedforward_torque!==undefined){"
"const feedforwardInput=document.getElementById('feedforwardTorque');"
"if(document.activeElement!==feedforwardInput){"
"feedforwardInput.value=data.feedforward_torque.toFixed(1);"
"}"
"}"
"if(data.climbing_torque!==undefined){"
"const climbingInput=document.getElementById('climbingTorque');"
"if(document.activeElement!==climbingInput){"
```

```
"climbingInput.value=data.climbing_torque.toFixed(1);"
"}"
"}"
"if(data.vt_lift_torque!==undefined){"
"const liftTorqueInput=document.getElementById('liftTorque');"
"if(document.activeElement!==liftTorqueInput){"
"liftTorqueInput.value=data.vt_lift_torque.toFixed(1);"
"}"
"}"
"if(data.vt_lift_speed!==undefined){"
"const liftSpeedInput=document.getElementById('liftSpeed');"
"if(document.activeElement!==liftSpeedInput){"
"liftSpeedInput.value=data.vt_lift_speed.toFixed(2);"
"}"
"}"
"if(data.vt_drop_torque!==undefined){"
"const dropTorqueInput=document.getElementById('dropTorque');"
"if(document.activeElement!==dropTorqueInput){"
"dropTorqueInput.value=data.vt_drop_torque.toFixed(1);"
"}"
"}"
"if(data.vt_drop_speed!==undefined){"
"const dropSpeedInput=document.getElementById('dropSpeed');"
"if(document.activeElement!==dropSpeedInput){"
"dropSpeedInput.value=data.vt_drop_speed.toFixed(2);"
"}"
"}"
"if(data.vt_lift_fixed_duration!==undefined){"
"const liftFixedDurationInput=document.getElementById('liftFixedDuration');"
"if(document.activeElement!==liftFixedDurationInput){"
"liftFixedDurationInput.value=data.vt_lift_fixed_duration;"
"}"
"}"
"if(data.vt_drop_delay!==undefined){"
"const dropDelayInput=document.getElementById('dropDelay');"
"if(document.activeElement!==dropDelayInput){"
"dropDelayInput.value=data.vt_drop_delay;"
"}"
"}"
"if(data.vt_drop_fixed_duration!==undefined){"
"const dropFixedDurationInput=document.getElementById('dropFixedDuration');"
"if(document.activeElement!==dropFixedDurationInput){"
"dropFixedDurationInput.value=data.vt_drop_fixed_duration;"
"}"
"}"
"if(data.vt_default_cycle_duration!==undefined){"
"const defaultCycleDurationInput=document.getElementById('defaultCycleDuration');"
"if(document.activeElement!==defaultCycleDurationInput){"
"defaultCycleDurationInput.value=data.vt_default_cycle_duration;"
"}"
```

```
"}"
"if(data.vt_enable_threshold!==undefined){"
"const enableThresholdInput=document.getElementById('enableThreshold');"
"if(document.activeElement!==enableThresholdInput){"
"enableThresholdInput.value=data.vt_enable_threshold.toFixed(2);""}""}"
"if(data.vt_min_velocity!==undefined){"
"const minVelocityInput=document.getElementById('minVelocity');"
"if(document.activeElement!==minVelocityInput){"
"minVelocityInput.value=data.vt_min_velocity.toFixed(2);""}""}"
"}).catch(e=>console.error(e))""}"
"function startControl(){"
"fetch('/api/control',{method:'POST',headers:{'Content-
Type':'application/json'},body:JSON.stringify({action:'start'})}).then(r=>r.text()).then(alert)""}"
"function stopControl(){"
"fetch('/api/control',{method:'POST',headers:{'Content-
Type':'application/json'},body:JSON.stringify({action:'stop'})}).then(r=>r.text()).then(alert)""}"
"function emergencyStop(){"
"fetch('/api/control',{method:'POST',headers:{'Content-
Type':'application/json'},body:JSON.stringify({action:'emergency'})}).then(r=>r.text()).then(alert)""}"
"function setWalkingMode(mode){"
"fetch('/api/control',{method:'POST',headers:{'Content-
Type':'application/json'},body:JSON.stringify({action:'mode',mode:mode})}).then(r=>r.text()).then(alert)"
"}"
"function startAlternating(){"
"fetch('/api/control',{method:'POST',headers:{'Content-
Type':'application/json'},body:JSON.stringify({action:'start_alternating'})}).then(r=>r.text()).then(alert)"
"}"
"function stopAlternating(){"
"fetch('/api/control',{method:'POST',headers:{'Content-
Type':'application/json'},body:JSON.stringify({action:'stop_alternating'})}).then(r=>r.text()).then(alert)"
"}"
"function enableMotionDetection(){"
"const btn=document.getElementById('motionDetectionBtn');"
"const status=document.getElementById('motionDetectionStatus');"
"if(btn.textContent==='启用智能运动检测'){"
"fetch('/api/control',{method:'POST',headers:{'Content-
Type':'application/json'},body:JSON.stringify({action:'enable_motion_detection'})}).then(r=>r.text()).then(msg=>{"
"alert(msg);"
"btn.textContent='关闭智能运动检测';"
"btn.className='btn btn-danger';"
"status.textContent='状态：启用';"
"status.style.color='#28a745';"
"})"
"}else{"
"fetch('/api/control',{method:'POST',headers:{'Content-
Type':'application/json'},body:JSON.stringify({action:'disable_motion_detection'})}).then(r=>r.text()).then(msg=>{"
"alert(msg);"
"btn.textContent='启用智能运动检测';"
"btn.className='btn btn-success';"
"status.textContent='状态：关闭';"
```

```
"status.style.color='#666';""})""}""}"
"function updateParams(){"
"const params={"
"action:'update_params',"
"motor1:{position:parseFloat(document.getElementById('pos1').value),speed:parseFloat(document.getElementById('speed1').valu
e),torque:parseFloat(document.getElementById('torque1').value),kp:parseFloat(document.getElementById('kp1').value),kd:parseFl
oat(document.getElementById('kd1').value)},"
"motor2:{position:parseFloat(document.getElementById('pos2').value),speed:parseFloat(document.getElementById('speed2').valu
e),torque:parseFloat(document.getElementById('torque2').value),kp:parseFloat(document.getElementById('kp2').value),kd:parseFl
oat(document.getElementById('kd2').value)}""};"
"fetch('/api/params',{method:'POST',headers:{'Content-
Type':'application/json'},body:JSON.stringify(params)}).then(r=>r.text()).then(alert)""}"
"function updateAlternatingParams(){"
"const params={"
"action:'update_alternating',"
"interval:parseInt(document.getElementById('interval').value),"
"speedX:parseFloat(document.getElementById('speedX').value),"
"speedY:parseFloat(document.getElementById('speedY').value),"
"currentLimit:parseFloat(document.getElementById('currentLimit').value),"
"feedforwardTorque:parseFloat(document.getElementById('feedforwardTorque').value)""};"
"fetch('/api/params',{method:'POST',headers:{'Content-
Type':'application/json'},body:JSON.stringify(params)}).then(r=>r.text()).then(alert)""}"
"function updateMotionDetectionParams(){"
"const params={"
"action:'update_motion_detection',"
"climbingTorque:parseFloat(document.getElementById('climbingTorque').value)""};"
"fetch('/api/params',{method:'POST',headers:{'Content-
Type':'application/json'},body:JSON.stringify(params)}).then(r=>r.text()).then(alert)"
"}"
"function updateVelocityTrackingParams(){"
"const params={"
"action:'update_velocity_tracking',"
"liftTorque:parseFloat(document.getElementById('liftTorque').value),"
"liftSpeed:parseFloat(document.getElementById('liftSpeed').value),"
"dropTorque:parseFloat(document.getElementById('dropTorque').value),"
"dropSpeed:parseFloat(document.getElementById('dropSpeed').value),"
"liftFixedDuration:parseFloat(document.getElementById('liftFixedDuration').value),"
"dropDelay:parseFloat(document.getElementById('dropDelay').value),"
"dropFixedDuration:parseFloat(document.getElementById('dropFixedDuration').value),"
"defaultCycleDuration:parseFloat(document.getElementById('defaultCycleDuration').value),"
"enableThreshold:parseFloat(document.getElementById('enableThreshold').value),"
"minVelocity:parseFloat(document.getElementById('minVelocity').value)"
"};"
"fetch('/api/params',{method:'POST',headers:{'Content-
Type':'application/json'},body:JSON.stringify(params)}).then(r=>r.text()).then(alert)"
"}"
"function toggleVelocityTracking(){"
"const btn=document.getElementById('velocityTrackingBtn');"
"const status=document.getElementById('velocityTrackingStatus');"
"const resetBtn=document.getElementById('resetVelocityBtn');"
```

```
"if(btn.textContent==='启用速度跟踪模式'){"
"fetch('/api/control',{method:'POST',headers:{'Content-
Type':'application/json'},body:JSON.stringify({action:'enable_velocity_tracking'})}).then(r=>r.text()).then(msg=>{"
"alert(msg);"
"btn.textContent='关闭速度跟踪模式';"
"btn.className='btn btn-danger';"
"status.textContent='状态：等待激活';"
"status.style.color='#ffc107';"
"resetBtn.disabled=false;"
"})"
"}else{"
"fetch('/api/control',{method:'POST',headers:{'Content-
Type':'application/json'},body:JSON.stringify({action:'disable_velocity_tracking'})}).then(r=>r.text()).then(msg=>{"
"alert(msg);"
"btn.textContent='启用速度跟踪模式';"
"btn.className='btn btn-success';"
"status.textContent='状态：关闭';"
"status.style.color='#666';"
"resetBtn.disabled=true;"
"})"
"}"
"}"
"function resetVelocityTracking(){"
"if(confirm('确定要重置节律控制到等待激活状态吗？')){"
"fetch('/api/control',{method:'POST',headers:{'Content-
Type':'application/json'},body:JSON.stringify({action:'reset_velocity_tracking'})}).then(r=>r.text()).then(msg=>{"
"alert(msg);"
"const status=document.getElementById('velocityTrackingStatus');"
"status.textContent='状态：等待激活';"
"status.style.color='#ffc107';""})""}""}"
"setInterval(updateStatus,2000);"
"updateStatus();"
"</script>"
"</body></html>";
#ifdef __cplusplus
}
#endif
#endif
```

// ========== wifi_softap_module.h ==========

```
#ifndef WIFI_SOFTAP_MODULE_H
#define WIFI_SOFTAP_MODULE_H
#include "esp_wifi.h"
#include "esp_event.h"
#ifdef __cplusplus
extern "C" {
#endif
typedef struct {
    char ssid[32];
    char password[64];
    uint8_t channel;
```

```
    uint8_t max_connection;
    wifi_auth_mode_t authmode;
} wifi_softap_config_t;
typedef void (*wifi_softap_event_cb_t)(int32_t event_id, void* event_data);
esp_err_t wifi_softap_init(const wifi_softap_config_t* config, wifi_softap_event_cb_t event_callback);
uint8_t wifi_softap_get_connected_count(void);
wifi_softap_config_t wifi_softap_get_default_config(void);
#ifdef __cplusplus
}
#endif
#endif
```

// ========== wifi_softap_module.c ==========

```
#include <string.h>
#include "wifi_softap_module.h"
#include "esp_mac.h"
#include "esp_wifi.h"
#include "esp_event.h"
#include "esp_log.h"
#include "esp_netif.h"
#include "lwip/err.h"
#include "lwip/sys.h"
static const char *TAG = "wifi_softap_module";
static bool is_initialized = false;
static bool is_started = false;
static uint8_t connected_count = 0;
static wifi_softap_event_cb_t user_event_callback = NULL;
static void wifi_event_handler(void* arg, esp_event_base_t event_base,
                               int32_t event_id, void* event_data)
{
    if (event_id == WIFI_EVENT_AP_STACONNECTED) {
        wifi_event_ap_staconnected_t* event = (wifi_event_ap_staconnected_t*) event_data;
        connected_count++;
        ESP_LOGI(TAG, "设备 "MACSTR" 连接， AID=%d， 当前连接数=%d",
                 MAC2STR(event->mac), event->aid, connected_count);
    } else if (event_id == WIFI_EVENT_AP_STADISCONNECTED) {
        wifi_event_ap_stadisconnected_t* event = (wifi_event_ap_stadisconnected_t*) event_data;
        connected_count--;
        ESP_LOGI(TAG, "设备 "MACSTR" 断开， AID=%d， 原因=%d， 当前连接数=%d",
                 MAC2STR(event->mac), event->aid, event->reason, connected_count);
    }
    if (user_event_callback != NULL) {
        user_event_callback(event_id, event_data);
    }
}
esp_err_t wifi_softap_init(const wifi_softap_config_t* config, wifi_softap_event_cb_t event_callback)
{
    if (is_initialized) {
        ESP_LOGW(TAG, "WiFi 热点模块已初始化");
        return ESP_OK;
    }
```

```
    if (config == NULL) {
        ESP_LOGE(TAG, "配置参数不能为空");
        return ESP_ERR_INVALID_ARG;
    }
    user_event_callback = event_callback;
    ESP_ERROR_CHECK(esp_netif_init());
    ESP_ERROR_CHECK(esp_event_loop_create_default());
    esp_netif_create_default_wifi_ap();
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));
    ESP_ERROR_CHECK(esp_event_handler_instance_register(WIFI_EVENT,
                                                        ESP_EVENT_ANY_ID,
                                                        &wifi_event_handler,
                                                        NULL,
                                                        NULL));
    wifi_config_t wifi_config = {
        .ap = {
            .ssid_len = strlen(config->ssid),
            .channel = config->channel,
            .max_connection = config->max_connection,
            .authmode = config->authmode,
            .pmf_cfg = {
                .required = false,
            },
        },
    };
    strncpy((char*)wifi_config.ap.ssid, config->ssid, sizeof(wifi_config.ap.ssid) - 1);
    strncpy((char*)wifi_config.ap.password, config->password, sizeof(wifi_config.ap.password) - 1);
    if (strlen(config->password) == 0) {
        wifi_config.ap.authmode = WIFI_AUTH_OPEN;
        wifi_config.ap.pmf_cfg.required = false;
    }
    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_AP));
    ESP_ERROR_CHECK(esp_wifi_set_config(WIFI_IF_AP, &wifi_config));
    ESP_ERROR_CHECK(esp_wifi_start());
    is_initialized = true;
    is_started = true;
    connected_count = 0;
    ESP_LOGI(TAG, "WiFi 热点启动完成 - SSID:%s, 信道:%d, 最大连接数:%d",
             config->ssid, config->channel, config->max_connection);
    return ESP_OK;
}
uint8_t wifi_softap_get_connected_count(void)
{
    return connected_count;
}
wifi_softap_config_t wifi_softap_get_default_config(void)
{
    wifi_softap_config_t default_config = {
        .ssid = "ESP32-Exoskeleton",
```

```
        .password = "",
        .channel = 1,
        .max_connection = 4,
        .authmode = WIFI_AUTH_OPEN,
    };
    return default_config;
}
// ========== velocity_tracking_mode.h ==========
#ifndef VELOCITY_TRACKING_MODE_H
#define VELOCITY_TRACKING_MODE_H
#include <stdint.h>
#include <stdbool.h>
#include "continuous_torque_velocity_mode.h"
#ifdef __cplusplus
extern "C" {
#endif
#define VELOCITY_TRACKING_ENABLE_THRESHOLD 0.65f
#define VELOCITY_TRACKING_MIN_VELOCITY 0.25f
#define VELOCITY_TRACKING_UPDATE_INTERVAL_MS 50
#define LIFT_LEG_FIXED_DURATION_MS 600
#define DROP_LEG_DELAY_MS 100
#define DROP_LEG_FIXED_DURATION_MS 600
#define LIFT_LEG_MAX_DURATION_MS 800
#define TIMEOUT_DROP_LEG_DURATION_MS 600
#define DEFAULT_CYCLE_DURATION_MS 2000
#define CYCLE_TIMEOUT_MULTIPLIER 1.0f
#define LIFT_LEG_TORQUE 6.0f
#define LIFT_LEG_POSITION 0.0f
#define LIFT_LEG_SPEED 2.75f
#define LIFT_LEG_KP 0.0f
#define LIFT_LEG_KD 1.0f
#define DROP_LEG_TORQUE -1.0f
#define DROP_LEG_POSITION 0.0f
#define DROP_LEG_SPEED -1.5f
#define DROP_LEG_KP 0.0f
#define DROP_LEG_KD 1.0f
typedef enum {
    VELOCITY_TRACKING_DISABLED = 0,
    VELOCITY_TRACKING_ENABLED = 1,
    VELOCITY_TRACKING_ACTIVE = 2
} velocity_tracking_state_t;
typedef enum {
    MOTOR_ACTION_IDLE = 0,
    MOTOR_ACTION_LIFT_LEG = 1,
    MOTOR_ACTION_DROP_LEG = 2
} motor_action_t;
typedef enum {
    VELOCITY_DIR_UNKNOWN = 0,
    VELOCITY_DIR_POSITIVE = 1,
    VELOCITY_DIR_NEGATIVE = 2
```

```
} velocity_direction_t;
typedef enum {
    WORK_CYCLE_MOTOR1 = 1,
    WORK_CYCLE_MOTOR2 = 2
} work_cycle_t;
typedef struct {
    velocity_direction_t current_dir;
    velocity_direction_t last_dir;
    uint32_t last_change_time;
    uint32_t control_duration_ms;
    uint32_t action_start_time;
    bool action_active;
    motor_action_t current_action;
    bool detection_blocked;
    bool is_resting;
    bool detection_delayed;
    uint32_t detection_delay_start_time;
    uint32_t total_cycles;
    uint32_t avg_cycle_time_ms;
    uint32_t last_cycle_times[5];
    uint8_t cycle_time_index;
} rhythm_control_t;
typedef struct {
    velocity_tracking_state_t state;
    bool enabled;
    uint32_t last_update_time;
    motion_mode_state_t motion_state;
    work_cycle_t current_work_cycle;
    bool cycle_completed;
    uint32_t cycle_start_time;
    uint32_t expected_cycle_duration_ms;
    uint32_t cycle_timeout_threshold_ms;
    uint32_t last_switch_time;
    uint32_t switch_protection_duration_ms;
    rhythm_control_t motor1_rhythm;
    rhythm_control_t motor2_rhythm;
    uint32_t activation_count;
    uint32_t total_lift_actions;
    uint32_t total_drop_actions;
    uint32_t total_work_cycles;
} velocity_tracking_context_t;
extern velocity_tracking_context_t velocity_tracking_context;
extern bool velocity_tracking_mode_enabled;
extern float velocity_tracking_lift_leg_torque;
extern float velocity_tracking_lift_leg_speed;
extern float velocity_tracking_drop_leg_torque;
extern float velocity_tracking_drop_leg_speed;
extern uint32_t velocity_tracking_lift_leg_max_duration;
extern uint32_t velocity_tracking_lift_leg_fixed_duration_ms;
extern uint32_t velocity_tracking_drop_leg_delay_ms;
```

```
extern uint32_t velocity_tracking_drop_leg_fixed_duration_ms;
extern uint32_t velocity_tracking_default_cycle_duration_ms;
extern float velocity_tracking_enable_threshold;
extern float velocity_tracking_min_velocity;
void velocity_tracking_mode_init(void);
void velocity_tracking_mode_set_enabled(bool enable);
bool velocity_tracking_mode_is_enabled(void);
velocity_tracking_state_t velocity_tracking_mode_get_state(void);
bool velocity_tracking_mode_update(position_ring_buffer_t *buffer_motor1,
                                   position_ring_buffer_t *buffer_motor2,
                                   float motor1_velocity,
                                   float motor2_velocity,
                                   uint32_t timestamp);
bool velocity_tracking_should_enable(position_ring_buffer_t *buffer_motor1,
                                     position_ring_buffer_t *buffer_motor2,
                                     uint32_t timestamp,
                                     int *triggered_motor);
motor_action_t velocity_tracking_get_motor_action(float velocity, int motor_id);
void velocity_tracking_execute_motor_action(int motor_id, motor_action_t action);
void velocity_tracking_execute_timed_motor_action(int motor_id, motor_action_t action, uint32_t duration_ms);
velocity_direction_t velocity_tracking_get_direction(float velocity);
bool velocity_tracking_update_rhythm(rhythm_control_t *rhythm, float velocity, uint32_t timestamp, int motor_id);
void velocity_tracking_init_rhythm(rhythm_control_t *rhythm);
bool velocity_tracking_process_rhythm_timing(rhythm_control_t *rhythm, int motor_id, uint32_t timestamp);
void velocity_tracking_get_statistics(velocity_tracking_context_t *context);
void velocity_tracking_reset_statistics(void);
void velocity_tracking_reset_to_enabled(void);
bool velocity_tracking_check_cycle_timeout(position_ring_buffer_t *buffer_motor1,
                                           position_ring_buffer_t *buffer_motor2,
                                           uint32_t timestamp);
void velocity_tracking_start_task(void);
void velocity_tracking_stop_task(void);
#ifndef MOTOR_CONTROL_PARAMS_T_DEFINED
#define MOTOR_CONTROL_PARAMS_T_DEFINED
typedef struct {
  float torque;
  float position;
  float speed;
  float kp;
  float kd;
} motor_control_params_t;
#endif
extern void unified_motor_control(int motor_id, const motor_control_params_t* params);
#ifdef __cplusplus
}
#endif
#endif
```

// ========== velocity_tracking_mode.c ==========

```
#include "velocity_tracking_mode.h"
#include "esp_log.h"
```

```
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include <math.h>
#include <string.h>
static const char *TAG = "velocity_tracking";
velocity_tracking_context_t velocity_tracking_context;
bool velocity_tracking_mode_enabled = false;
float velocity_tracking_lift_leg_torque = LIFT_LEG_TORQUE;
float velocity_tracking_lift_leg_speed = LIFT_LEG_SPEED;
float velocity_tracking_drop_leg_torque = DROP_LEG_TORQUE;
float velocity_tracking_drop_leg_speed = DROP_LEG_SPEED;
uint32_t velocity_tracking_lift_leg_max_duration = LIFT_LEG_MAX_DURATION_MS;
uint32_t velocity_tracking_lift_leg_fixed_duration_ms = LIFT_LEG_FIXED_DURATION_MS;
uint32_t velocity_tracking_drop_leg_delay_ms = DROP_LEG_DELAY_MS;
uint32_t velocity_tracking_drop_leg_fixed_duration_ms = DROP_LEG_FIXED_DURATION_MS;
uint32_t velocity_tracking_default_cycle_duration_ms = DEFAULT_CYCLE_DURATION_MS;
float velocity_tracking_enable_threshold = VELOCITY_TRACKING_ENABLE_THRESHOLD;
float velocity_tracking_min_velocity = VELOCITY_TRACKING_MIN_VELOCITY;
static TaskHandle_t velocity_tracking_task_handle = NULL;
void velocity_tracking_task(void* pvParameters);
void velocity_tracking_init_rhythm(rhythm_control_t *rhythm) {
   if (rhythm == NULL) return;
   memset(rhythm, 0, sizeof(rhythm_control_t));
   rhythm->current_dir = VELOCITY_DIR_UNKNOWN;
   rhythm->last_dir = VELOCITY_DIR_UNKNOWN;
   rhythm->current_action = MOTOR_ACTION_IDLE;
   rhythm->control_duration_ms = 1000;
   rhythm->detection_blocked = false;
   rhythm->is_resting = false;
   rhythm->detection_delayed = false;
   rhythm->detection_delay_start_time = 0;}
void velocity_tracking_mode_init(void) {
   memset(&velocity_tracking_context, 0, sizeof(velocity_tracking_context_t));
   velocity_tracking_context.state = VELOCITY_TRACKING_DISABLED;
   velocity_tracking_context.enabled = false;
   velocity_tracking_context.last_update_time = 0;
   motion_mode_state_init(&velocity_tracking_context.motion_state);
   velocity_tracking_context.current_work_cycle = WORK_CYCLE_MOTOR1;
   velocity_tracking_context.cycle_completed = false;
   velocity_tracking_context.cycle_start_time = 0;
   velocity_tracking_context.expected_cycle_duration_ms = velocity_tracking_default_cycle_duration_ms;
   velocity_tracking_context.cycle_timeout_threshold_ms = (uint32_t)(velocity_tracking_default_cycle_duration_ms *
CYCLE_TIMEOUT_MULTIPLIER);
   velocity_tracking_context.last_switch_time = 0;
   velocity_tracking_context.switch_protection_duration_ms = 0;
   velocity_tracking_init_rhythm(&velocity_tracking_context.motor1_rhythm);
   velocity_tracking_init_rhythm(&velocity_tracking_context.motor2_rhythm);
   velocity_tracking_context.motor1_rhythm.is_resting = false;
   velocity_tracking_context.motor2_rhythm.is_resting = true;
   velocity_tracking_mode_enabled = false;
```

```
    ESP_LOGI(TAG, "速度跟踪模式初始化完成（轮流工作周期版本）");
}
void velocity_tracking_mode_set_enabled(bool enable) {
    velocity_tracking_mode_enabled = enable;
    velocity_tracking_context.enabled = enable;
    if (enable) {
        velocity_tracking_context.state = VELOCITY_TRACKING_ENABLED;
        ESP_LOGI(TAG, "速度跟踪模式已启用");
    } else {
        velocity_tracking_context.state = VELOCITY_TRACKING_DISABLED;
        velocity_tracking_init_rhythm(&velocity_tracking_context.motor1_rhythm);
        velocity_tracking_init_rhythm(&velocity_tracking_context.motor2_rhythm);
        velocity_tracking_context.last_switch_time = 0;
        velocity_tracking_context.switch_protection_duration_ms = 0;
        ESP_LOGI(TAG, "速度跟踪模式已禁用");
    }
}
bool velocity_tracking_mode_is_enabled(void) {
    return velocity_tracking_mode_enabled;
}
velocity_tracking_state_t velocity_tracking_mode_get_state(void) {
    return velocity_tracking_context.state;
}
bool velocity_tracking_should_enable(position_ring_buffer_t *buffer_motor1,
                                     position_ring_buffer_t *buffer_motor2,
                                     uint32_t timestamp,
                                     int *triggered_motor) {
    if (buffer_motor1 == NULL || buffer_motor2 == NULL || triggered_motor == NULL) {
        if (triggered_motor) *triggered_motor = 0;
        return false;
    }
    motion_mode_t motor1_mode = detect_motion_mode(buffer_motor1, timestamp, &velocity_tracking_context.motion_state);
    bool motor1_should_enable = (motor1_mode == MOTION_MODE_CLIMBING);
    motion_mode_state_t motor2_motion_state;
    motion_mode_state_init(&motor2_motion_state);
    motion_mode_t motor2_mode = detect_motion_mode(buffer_motor2, timestamp, &motor2_motion_state);
    bool motor2_should_enable = (motor2_mode == MOTION_MODE_CLIMBING);
    bool should_enable = motor1_should_enable || motor2_should_enable;
    if (should_enable) {
        if (motor1_should_enable) {
            *triggered_motor = 1;
        } else {
            *triggered_motor = 2;
        }
    } else {
        *triggered_motor = 0;
    }
    ESP_LOGD(TAG, "双电机升档检测: 1 号电机=%s(%s), 2 号电机=%s(%s) -> velocity_tracking %s (触发电机: %d)",
             motor1_mode == MOTION_MODE_STATIC ? "静止" :
             motor1_mode == MOTION_MODE_WALKING ? "行走" : "爬楼",
```

```
        motor1_should_enable ? "满足" : "不满足",
        motor2_mode == MOTION_MODE_STATIC ? "静止" :
        motor2_mode == MOTION_MODE_WALKING ? "行走" : "爬楼",
        motor2_should_enable ? "满足" : "不满足",
        should_enable ? "启用" : "禁用",
        *triggered_motor);
  return should_enable;
}
velocity_direction_t velocity_tracking_get_direction(float velocity) {
  if (fabsf(velocity) < velocity_tracking_min_velocity) {
    return VELOCITY_DIR_UNKNOWN;
  }
  return (velocity > 0) ? VELOCITY_DIR_POSITIVE : VELOCITY_DIR_NEGATIVE;
}
bool velocity_tracking_update_rhythm(rhythm_control_t *rhythm, float velocity, uint32_t timestamp, int motor_id) {
  if (rhythm == NULL) return false;
  if (rhythm->detection_blocked) {
    ESP_LOGD(TAG, "电机%d 检测被阻止，跳过节律更新", motor_id);
    return false;
  }
  if (rhythm->detection_delayed) {
    uint32_t delay_elapsed = timestamp - rhythm->detection_delay_start_time;
    if (delay_elapsed < velocity_tracking_drop_leg_delay_ms) {
      ESP_LOGD(TAG, "电机%d 检测延迟中，已过%lu ms，还需%lu ms",
          motor_id, delay_elapsed, velocity_tracking_drop_leg_delay_ms - delay_elapsed);
      return false;
    } else {
      rhythm->detection_delayed = false;
      velocity_tracking_context.cycle_start_time = timestamp;
      ESP_LOGI(TAG, "电机%d 检测延迟结束，恢复速度检测，开始工作周期计时", motor_id);
    }
  }
  if (rhythm->action_active) {
    return false;
  }
  velocity_direction_t new_dir = velocity_tracking_get_direction(velocity);
  if (new_dir == VELOCITY_DIR_UNKNOWN) {
    return false;
  }
  bool should_lift = false;
  if (motor_id == 1 && new_dir == VELOCITY_DIR_NEGATIVE) {
    should_lift = true;
  } else if (motor_id == 2 && new_dir == VELOCITY_DIR_POSITIVE) {
    should_lift = true;
  }
  if (should_lift) {
    rhythm->current_action = MOTOR_ACTION_LIFT_LEG;
    rhythm->action_start_time = timestamp;
    rhythm->action_active = true;
    rhythm->control_duration_ms = velocity_tracking_lift_leg_fixed_duration_ms;
```

```
        rhythm->current_dir = new_dir;
        ESP_LOGI(TAG, "电机%d 检测到%s，执行抬腿 MIT（固定时长: %lu ms）",
                 motor_id,
                 new_dir == VELOCITY_DIR_POSITIVE ? "+v" : "-v",
                 rhythm->control_duration_ms);
        velocity_tracking_execute_motor_action(motor_id - 1, MOTOR_ACTION_LIFT_LEG);
        rhythm->total_cycles++;
        rhythm->last_change_time = timestamp;
        rhythm->last_dir = new_dir;
        return true;
    }
    return false;
}
motor_action_t velocity_tracking_get_motor_action(float velocity, int motor_id) {
    velocity_direction_t dir = velocity_tracking_get_direction(velocity);
    if (dir == VELOCITY_DIR_UNKNOWN) {
        return MOTOR_ACTION_IDLE;
    }
    if (motor_id == 1) {
        return (dir == VELOCITY_DIR_NEGATIVE) ? MOTOR_ACTION_LIFT_LEG : MOTOR_ACTION_DROP_LEG;
    } else if (motor_id == 2) {
        return (dir == VELOCITY_DIR_POSITIVE) ? MOTOR_ACTION_LIFT_LEG : MOTOR_ACTION_DROP_LEG;
    }
    return MOTOR_ACTION_IDLE;
}
void velocity_tracking_execute_motor_action(int motor_id, motor_action_t action) {
    motor_control_params_t params = {0};
    switch (action) {
        case MOTOR_ACTION_LIFT_LEG:
            params.torque = velocity_tracking_lift_leg_torque;
            params.position = LIFT_LEG_POSITION;
            params.speed = velocity_tracking_lift_leg_speed;
            params.kp = LIFT_LEG_KP;
            params.kd = LIFT_LEG_KD;
            if (motor_id == 0) {
                params.speed = -fabsf(params.speed);
                params.torque = -fabsf(params.torque);
            } else {
                params.speed = fabsf(params.speed);
                params.torque = fabsf(params.torque);
            }
            velocity_tracking_context.total_lift_actions++;
            ESP_LOGD(TAG, "电机%d 执行抬腿动作: 力矩=%.1f, 速度=%.1f",
                     motor_id+1, params.torque, params.speed);
            break;
        case MOTOR_ACTION_DROP_LEG:
            params.torque = velocity_tracking_drop_leg_torque;
            params.position = DROP_LEG_POSITION;
            params.speed = velocity_tracking_drop_leg_speed;
            params.kp = DROP_LEG_KP;
```

```
        params.kd = DROP_LEG_KD;
        if (motor_id == 0) {
            params.speed = fabsf(params.speed);
            params.torque = fabsf(params.torque);
        } else {
            params.speed = -fabsf(params.speed);
            params.torque = -fabsf(params.torque);
        }
        velocity_tracking_context.total_drop_actions++;
        ESP_LOGD(TAG, "电机%d 执行放腿动作: 力矩=%.1f, 速度=%.1f",
                 motor_id+1, params.torque, params.speed);
        break;
    case MOTOR_ACTION_IDLE:
    default:
        params.torque = 0.0f;
        params.position = 0.0f;
        params.speed = 0.0f;
        params.kp = 0.0f;
        params.kd = 0.0f;
        ESP_LOGD(TAG, "电机%d 设置为空闲状态", motor_id+1);
        break;
  }
  unified_motor_control(motor_id, &params);
}
void velocity_tracking_execute_timed_motor_action(int motor_id, motor_action_t action, uint32_t duration_ms) {
  velocity_tracking_execute_motor_action(motor_id, action);
  ESP_LOGI(TAG, "电机%d 执行定时动作: %s, 持续%lu ms",
           motor_id+1,
           action == MOTOR_ACTION_LIFT_LEG ? "抬腿" :
           action == MOTOR_ACTION_DROP_LEG ? "放腿" : "空闲",
           duration_ms);
}
bool velocity_tracking_process_rhythm_timing(rhythm_control_t *rhythm, int motor_id, uint32_t timestamp) {
  if (rhythm == NULL || !rhythm->action_active) {
    return false;
  }
  uint32_t elapsed_time = timestamp - rhythm->action_start_time;
  if (rhythm->current_action == MOTOR_ACTION_LIFT_LEG && elapsed_time >= rhythm->control_duration_ms) {
    if (motor_id == 0) {
        velocity_tracking_context.motor1_rhythm.is_resting = true;
        velocity_tracking_context.motor2_rhythm.is_resting = false;
        velocity_tracking_context.current_work_cycle = WORK_CYCLE_MOTOR2;
        velocity_tracking_context.total_work_cycles++;
        velocity_tracking_context.motor2_rhythm.detection_delayed = true;
        velocity_tracking_context.motor2_rhythm.detection_delay_start_time = timestamp;
        ESP_LOGI(TAG, "1 号电机抬腿完成, 切换到 2 号电机工作, 2 号检测延迟%lu ms (总周期数: %lu)",
                 velocity_tracking_drop_leg_delay_ms, velocity_tracking_context.total_work_cycles);
    } else if (motor_id == 1) {
        velocity_tracking_context.motor1_rhythm.is_resting = false;
        velocity_tracking_context.motor2_rhythm.is_resting = true;
```

```
            velocity_tracking_context.current_work_cycle = WORK_CYCLE_MOTOR1;
            velocity_tracking_context.total_work_cycles++;
            velocity_tracking_context.motor1_rhythm.detection_delayed = true;
            velocity_tracking_context.motor1_rhythm.detection_delay_start_time = timestamp;
            ESP_LOGI(TAG, "2 号电机抬腿完成，切换到 1 号电机工作，1 号检测延迟%lu ms (总周期数: %lu)",
                     velocity_tracking_drop_leg_delay_ms, velocity_tracking_context.total_work_cycles);
        }
        rhythm->current_action = MOTOR_ACTION_DROP_LEG;
        rhythm->action_start_time = timestamp;
        rhythm->control_duration_ms = velocity_tracking_drop_leg_fixed_duration_ms;
        ESP_LOGI(TAG, "电机%d 抬腿 MIT 完成，开始执行压腿 MIT（固定时长: %lu ms）",
                 motor_id+1, rhythm->control_duration_ms);
        velocity_tracking_execute_motor_action(motor_id, MOTOR_ACTION_DROP_LEG);
        return true;
    } else if (rhythm->current_action == MOTOR_ACTION_DROP_LEG && elapsed_time >= rhythm->control_duration_ms) {
        rhythm->action_active = false;
        rhythm->current_action = MOTOR_ACTION_IDLE;
        ESP_LOGI(TAG, "电机%d 压腿 MIT 完成，耗时: %lu ms，设置为空闲状态", motor_id+1, elapsed_time);
        velocity_tracking_execute_motor_action(motor_id, MOTOR_ACTION_IDLE);
        return true;
    }
    return false;
}
bool velocity_tracking_mode_update(position_ring_buffer_t *buffer_motor1,
                                   position_ring_buffer_t *buffer_motor2,
                                   float motor1_velocity,
                                   float motor2_velocity,
                                   uint32_t timestamp) {
    if (!velocity_tracking_mode_enabled || buffer_motor1 == NULL || buffer_motor2 == NULL) {
        return false;
    }
    bool state_changed = false;
    if ((timestamp - velocity_tracking_context.last_update_time) < VELOCITY_TRACKING_UPDATE_INTERVAL_MS) {
        return false;
    }
    velocity_tracking_context.last_update_time = timestamp;
    if (velocity_tracking_check_cycle_timeout(buffer_motor1, buffer_motor2, timestamp)) {
        return true;}
    if (velocity_tracking_context.state == VELOCITY_TRACKING_ENABLED) {
        int triggered_motor = 0;
        bool should_enable = velocity_tracking_should_enable(buffer_motor1, buffer_motor2, timestamp, &triggered_motor);
        if (should_enable) {
            velocity_tracking_context.state = VELOCITY_TRACKING_ACTIVE;
            velocity_tracking_context.activation_count++;
            if (triggered_motor == 1) {
                velocity_tracking_context.motor1_rhythm.is_resting = false;
                velocity_tracking_context.motor2_rhythm.is_resting = true;
                velocity_tracking_context.current_work_cycle = WORK_CYCLE_MOTOR1;
                ESP_LOGI(TAG, "速度跟踪模式激活 (激活次数: %lu) - 1 号电机触发启动，1 号工作，2 号休息",
velocity_tracking_context.activation_count);
```

```
        } else if (triggered_motor == 2) {
            velocity_tracking_context.motor1_rhythm.is_resting = true;
            velocity_tracking_context.motor2_rhythm.is_resting = false;
            velocity_tracking_context.current_work_cycle = WORK_CYCLE_MOTOR2;
            ESP_LOGI(TAG, "速度跟踪模式激活 (激活次数: %lu) - 2 号电机触发启动，2 号工作，1 号休息",
velocity_tracking_context.activation_count);
        } else {
            velocity_tracking_context.motor1_rhythm.is_resting = false;
            velocity_tracking_context.motor2_rhythm.is_resting = true;
            velocity_tracking_context.current_work_cycle = WORK_CYCLE_MOTOR1;
            ESP_LOGI(TAG, "速度跟踪模式激活 (激活次数: %lu) - 未知触发电机，默认 1 号工作",
velocity_tracking_context.activation_count);}
        state_changed = true;}
    }
    if (velocity_tracking_context.state == VELOCITY_TRACKING_ACTIVE) {
        bool motor1_rhythm_changed = false;
        bool motor2_rhythm_changed = false;
        if (!velocity_tracking_context.motor1_rhythm.is_resting) {
            velocity_tracking_context.motor1_rhythm.detection_blocked = false;
            velocity_tracking_context.motor2_rhythm.detection_blocked = true;
            velocity_direction_t current_dir = velocity_tracking_get_direction(motor1_velocity);
            if (current_dir == VELOCITY_DIR_NEGATIVE) {
                motor1_rhythm_changed = velocity_tracking_update_rhythm(
                    &velocity_tracking_context.motor1_rhythm, motor1_velocity, timestamp, 1);
            }
        } else {
            if (velocity_tracking_context.motor1_rhythm.action_active &&
                velocity_tracking_context.motor1_rhythm.current_action != MOTOR_ACTION_DROP_LEG) {
                velocity_tracking_context.motor1_rhythm.action_active = false;
                velocity_tracking_context.motor1_rhythm.current_action = MOTOR_ACTION_IDLE;
                velocity_tracking_execute_motor_action(0, MOTOR_ACTION_IDLE);
            }
        }
        if (!velocity_tracking_context.motor2_rhythm.is_resting) {
            velocity_tracking_context.motor2_rhythm.detection_blocked = false;
            velocity_tracking_context.motor1_rhythm.detection_blocked = true;
            velocity_direction_t current_dir = velocity_tracking_get_direction(motor2_velocity);
            if (current_dir == VELOCITY_DIR_POSITIVE) {
                motor2_rhythm_changed = velocity_tracking_update_rhythm(
                    &velocity_tracking_context.motor2_rhythm, motor2_velocity, timestamp, 2);
            }
        } else {
            if (velocity_tracking_context.motor2_rhythm.action_active &&
                velocity_tracking_context.motor2_rhythm.current_action != MOTOR_ACTION_DROP_LEG) {
                velocity_tracking_context.motor2_rhythm.action_active = false;
                velocity_tracking_context.motor2_rhythm.current_action = MOTOR_ACTION_IDLE;
                velocity_tracking_execute_motor_action(1, MOTOR_ACTION_IDLE);
            }
        }
        bool motor1_timing_changed = velocity_tracking_process_rhythm_timing(
```

```
        &velocity_tracking_context.motor1_rhythm, 0, timestamp);
    bool motor2_timing_changed = velocity_tracking_process_rhythm_timing(
        &velocity_tracking_context.motor2_rhythm, 1, timestamp);
    if (motor1_rhythm_changed) {
        velocity_tracking_execute_timed_motor_action(0,
            velocity_tracking_context.motor1_rhythm.current_action,
            velocity_tracking_context.motor1_rhythm.control_duration_ms);
        velocity_tracking_context.total_lift_actions +=
            (velocity_tracking_context.motor1_rhythm.current_action == MOTOR_ACTION_LIFT_LEG) ? 1 : 0;
        velocity_tracking_context.total_drop_actions +=
            (velocity_tracking_context.motor1_rhythm.current_action == MOTOR_ACTION_DROP_LEG) ? 1 : 0;
    }
    if (motor2_rhythm_changed) {
        velocity_tracking_execute_timed_motor_action(1,
            velocity_tracking_context.motor2_rhythm.current_action,
            velocity_tracking_context.motor2_rhythm.control_duration_ms);
        velocity_tracking_context.total_lift_actions +=
            (velocity_tracking_context.motor2_rhythm.current_action == MOTOR_ACTION_LIFT_LEG) ? 1 : 0;
        velocity_tracking_context.total_drop_actions +=
            (velocity_tracking_context.motor2_rhythm.current_action == MOTOR_ACTION_DROP_LEG) ? 1 : 0;
    }
    state_changed = motor1_rhythm_changed || motor2_rhythm_changed || motor1_timing_changed || motor2_timing_changed;
  }
  return state_changed;
}
void velocity_tracking_get_statistics(velocity_tracking_context_t *context) {
  if (context != NULL) {
    *context = velocity_tracking_context;
  }
}
void velocity_tracking_reset_statistics(void) {
  velocity_tracking_context.activation_count = 0;
  velocity_tracking_context.total_lift_actions = 0;
  velocity_tracking_context.total_drop_actions = 0;
  ESP_LOGI(TAG, "速度跟踪模式统计信息已重置");
}
void velocity_tracking_reset_to_enabled(void) {
  if (velocity_tracking_context.enabled) {
    velocity_tracking_context.state = VELOCITY_TRACKING_ENABLED;
    velocity_tracking_context.current_work_cycle = WORK_CYCLE_MOTOR1;
    velocity_tracking_context.cycle_completed = false;
    velocity_tracking_context.cycle_start_time = 0;
    velocity_tracking_init_rhythm(&velocity_tracking_context.motor1_rhythm);
    velocity_tracking_init_rhythm(&velocity_tracking_context.motor2_rhythm);
    velocity_tracking_context.last_switch_time = 0;
    velocity_tracking_context.switch_protection_duration_ms = 0;
    velocity_tracking_execute_motor_action(0, MOTOR_ACTION_IDLE);
    velocity_tracking_execute_motor_action(1, MOTOR_ACTION_IDLE);
    ESP_LOGI(TAG, "速度跟踪模式已重置到启用状态，轮流工作周期已初始化，等待波峰波谷差值>=0.75 重新激活");
  } else {
```

```
      ESP_LOGW(TAG, "速度跟踪模式未启用， 无法重置到启用状态");
    }
}
void velocity_tracking_task(void* pvParameters) {
    ESP_LOGI(TAG, "速度跟踪模式任务启动");
    TickType_t last_update = xTaskGetTickCount();
    const TickType_t update_interval = pdMS_TO_TICKS(VELOCITY_TRACKING_UPDATE_INTERVAL_MS);
    while (1) {
        TickType_t current_time = xTaskGetTickCount();
        if (velocity_tracking_mode_enabled) {
            if ((current_time - last_update) >= pdMS_TO_TICKS(5000)) {
                ESP_LOGI(TAG, "轮流工作周期状态报告:");
                ESP_LOGI(TAG, "  模式: %s, 激活次数: %lu, 总工作周期: %lu",
                        velocity_tracking_context.state == VELOCITY_TRACKING_DISABLED ? "禁用" :
                        velocity_tracking_context.state == VELOCITY_TRACKING_ENABLED ? "启用" : "激活",
                        velocity_tracking_context.activation_count,
                        velocity_tracking_context.total_work_cycles);
                ESP_LOGI(TAG, "  工作状态: 1 号电机=%s, 2 号电机=%s",
                        velocity_tracking_context.motor1_rhythm.is_resting ? "休息" : "工作",
                        velocity_tracking_context.motor2_rhythm.is_resting ? "休息" : "工作");
                ESP_LOGI(TAG, "  抬腿次数: %lu, 放腿次数: %lu",
                        velocity_tracking_context.total_lift_actions,
                        velocity_tracking_context.total_drop_actions);
                if (velocity_tracking_context.state == VELOCITY_TRACKING_ACTIVE) {
                    ESP_LOGI(TAG, "  电机  1: 方向=%s, 循环数=%lu, 平均周期=%lu ms, 当前动作=%s",
                            velocity_tracking_context.motor1_rhythm.current_dir == VELOCITY_DIR_POSITIVE ? "+v" :
                            velocity_tracking_context.motor1_rhythm.current_dir == VELOCITY_DIR_NEGATIVE ? "-v" : "未知",
                            velocity_tracking_context.motor1_rhythm.total_cycles,
                            velocity_tracking_context.motor1_rhythm.avg_cycle_time_ms,
                            velocity_tracking_context.motor1_rhythm.action_active ?
                            (velocity_tracking_context.motor1_rhythm.current_action == MOTOR_ACTION_LIFT_LEG ? "抬腿" : "放腿") : "空闲");
                    ESP_LOGI(TAG, "  电机  2: 方向=%s, 循环数=%lu, 平均周期=%lu ms, 当前动作=%s",
                            velocity_tracking_context.motor2_rhythm.current_dir == VELOCITY_DIR_POSITIVE ? "+v" :
                            velocity_tracking_context.motor2_rhythm.current_dir == VELOCITY_DIR_NEGATIVE ? "-v" : "未知",
                            velocity_tracking_context.motor2_rhythm.total_cycles,
                            velocity_tracking_context.motor2_rhythm.avg_cycle_time_ms,
                            velocity_tracking_context.motor2_rhythm.action_active ?
                            (velocity_tracking_context.motor2_rhythm.current_action == MOTOR_ACTION_LIFT_LEG ? "抬腿" : "放腿") : "空闲");
                }
                last_update = current_time;
            }
        }
        vTaskDelay(update_interval);
    }
    ESP_LOGI(TAG, "速度跟踪模式任务退出");
    velocity_tracking_task_handle = NULL;
    vTaskDelete(NULL);
}
bool velocity_tracking_check_cycle_timeout(position_ring_buffer_t *buffer_motor1,
                                           position_ring_buffer_t *buffer_motor2,
```

```
                                uint32_t timestamp) {
    if (velocity_tracking_context.state != VELOCITY_TRACKING_ACTIVE) {
        return false;
    }
    if (velocity_tracking_context.cycle_start_time == 0) {
        return false;
    }
    uint32_t cycle_elapsed_time = timestamp - velocity_tracking_context.cycle_start_time;
    if (cycle_elapsed_time >= velocity_tracking_context.cycle_timeout_threshold_ms) {
        int working_motor_id = (velocity_tracking_context.current_work_cycle == WORK_CYCLE_MOTOR1) ? 0 : 1;
        rhythm_control_t *working_rhythm = (working_motor_id == 0) ?
            &velocity_tracking_context.motor1_rhythm : &velocity_tracking_context.motor2_rhythm;
        ESP_LOGW(TAG, "【超时重置】%d 号电机工作周期超时！", working_motor_id + 1);
        ESP_LOGW(TAG, " 运行时间: %lu ms >= 超时阈值: %lu ms",
                 cycle_elapsed_time, velocity_tracking_context.cycle_timeout_threshold_ms);
        ESP_LOGW(TAG, " 预期周期时间: %lu ms (%.1f 倍)",
                 velocity_tracking_context.expected_cycle_duration_ms, CYCLE_TIMEOUT_MULTIPLIER);
        if (!working_rhythm->action_active && working_rhythm->current_action == MOTOR_ACTION_IDLE) {
            ESP_LOGW(TAG, " 原因: %d 号电机在预期时间%.1f 倍内未检测到抬腿动作",
                     working_motor_id + 1, CYCLE_TIMEOUT_MULTIPLIER);
        } else {
            ESP_LOGW(TAG, " 原因: %d 号电机动作执行异常或卡滞", working_motor_id + 1);
        }
        velocity_tracking_context.state = VELOCITY_TRACKING_ENABLED;
        velocity_tracking_context.cycle_start_time = 0;
        velocity_tracking_context.cycle_completed = false;
        velocity_tracking_init_rhythm(&velocity_tracking_context.motor1_rhythm);
        velocity_tracking_init_rhythm(&velocity_tracking_context.motor2_rhythm);
        velocity_tracking_context.last_switch_time = 0;
        velocity_tracking_context.switch_protection_duration_ms = 0;
        velocity_tracking_context.motor1_rhythm.is_resting = false;
        velocity_tracking_context.motor2_rhythm.is_resting = true;
        velocity_tracking_context.current_work_cycle = WORK_CYCLE_MOTOR1;
        if (buffer_motor1 != NULL && buffer_motor2 != NULL) {
            position_ring_buffer_clear(buffer_motor1);
            position_ring_buffer_clear(buffer_motor2);
            ESP_LOGI(TAG, "【超时重置】清空双电机位置缓存区，等待重新激活");
        }
        velocity_tracking_execute_motor_action(0, MOTOR_ACTION_IDLE);
        velocity_tracking_execute_motor_action(1, MOTOR_ACTION_IDLE);
        ESP_LOGI(TAG, "【超时重置】velocity_tracking 模式已重置到 ENABLED 状态，等待重新激活");
        return true;
    }
    return false;
}
void velocity_tracking_start_task(void) {
    if (velocity_tracking_task_handle == NULL) {
        BaseType_t result = xTaskCreate(
            velocity_tracking_task,
            "VelocityTrack",
```

```
        4096,
        NULL,
        3,
        &velocity_tracking_task_handle
      );
      if (result == pdPASS) {
        ESP_LOGI(TAG, "速度跟踪模式任务创建成功");
      } else {
        ESP_LOGE(TAG, "速度跟踪模式任务创建失败");
      }
    } else {
      ESP_LOGW(TAG, "速度跟踪模式任务已在运行");
    }
}
void velocity_tracking_stop_task(void) {
    if (velocity_tracking_task_handle != NULL) {
      vTaskDelete(velocity_tracking_task_handle);
      velocity_tracking_task_handle = NULL;
      ESP_LOGI(TAG, "速度跟踪模式任务已停止");
    }
}
// ========== continuous_torque_velocity_mode.h ==========
#ifndef CONTINUOUS_TORQUE_VELOCITY_MODE_H
#define CONTINUOUS_TORQUE_VELOCITY_MODE_H
#include <stdint.h>
#include <stdbool.h>
#ifdef __cplusplus
extern "C" {
#endif
#define POSITION_BUFFER_SIZE 25
#define POSITION_CHANGE_THRESHOLD 0.05f
#define MODE_DETECTION_THRESHOLD 0.7f
#define CLIMBING_DOWNGRADE_THRESHOLD 0.5f
#define STATIC_DETECTION_THRESHOLD 0.2f
#define STATIC_TIMEOUT_MS 3000
#define MODE_LOCK_DURATION_MS 2000
#define STATIC_CONFIRM_DURATION_MS 1500
#define TORQUE_INCREASE_INTERVAL_MS 500
#define TORQUE_DECREASE_INTERVAL_MS 200
#define TORQUE_INCREASE_STEP 0.5f
#define TORQUE_DECREASE_STEP 1.5f
typedef enum {
    MOTION_MODE_STATIC = 0,
    MOTION_MODE_WALKING,
    MOTION_MODE_CLIMBING
} motion_mode_t;
typedef struct {
    int velocity;
    float torque;
    int kd;
```

```
} motor_params_t;
typedef struct {
    float current_torque;
    float target_torque;
    uint32_t last_update_time;
    bool is_increasing;
} torque_gradient_t;
typedef struct {
    motion_mode_t current_mode;
    motion_mode_t previous_mode;
    uint32_t mode_start_timestamp;
    bool is_continuous_mode;
    torque_gradient_t motor1_torque;
    torque_gradient_t motor2_torque;
    uint32_t last_mode_change_time;
    uint32_t static_confirm_start_time;
    bool in_static_confirmation;
    float last_position_range;
    int climbing_downgrade_count;
} motion_mode_state_t;
typedef struct {
    float position;
    uint32_t timestamp;
    bool valid;
} position_point_t;
typedef struct {
    position_point_t buffer[POSITION_BUFFER_SIZE];
    uint16_t head;
    uint16_t tail;
    uint16_t count;
    float last_recorded_position;
    bool initialized;
} position_ring_buffer_t;
void position_ring_buffer_init(position_ring_buffer_t *buffer);
bool position_ring_buffer_add_if_changed(position_ring_buffer_t *buffer,
                                         float new_position,
                                         uint32_t timestamp);
void position_ring_buffer_add(position_ring_buffer_t *buffer,
                              float position,
                              uint32_t timestamp);
bool position_ring_buffer_get_latest(position_ring_buffer_t *buffer,
                                     position_point_t *point);
bool position_ring_buffer_get_at_index(position_ring_buffer_t *buffer,
                                       int16_t index,
                                       position_point_t *point);
uint16_t position_ring_buffer_get_count(position_ring_buffer_t *buffer);
bool position_ring_buffer_is_empty(position_ring_buffer_t *buffer);
bool position_ring_buffer_is_full(position_ring_buffer_t *buffer);
void position_ring_buffer_clear(position_ring_buffer_t *buffer);
bool position_ring_buffer_get_min_max(position_ring_buffer_t *buffer,
```

```
                        float *min_pos, float *max_pos);
motion_mode_t detect_motion_mode(position_ring_buffer_t *buffer,
                        uint32_t current_timestamp,
                        motion_mode_state_t *state);
float detect_peak_valley_difference(position_ring_buffer_t *buffer);
void get_motor_params(motion_mode_t mode, int motor_id, motor_params_t *params);
void motion_mode_state_init(motion_mode_state_t *state);
void update_motion_mode_and_get_params(motion_mode_state_t *state,
                        motion_mode_t new_mode,
                        uint32_t current_timestamp,
                        int motor_id,
                        motor_params_t *params);
void torque_gradient_init(torque_gradient_t *gradient);
float update_torque_gradient(torque_gradient_t *gradient,
                        float target_torque,
                        uint32_t current_timestamp);
#ifdef __cplusplus
}
#endif
#endif
```

// ========== continuous_torque_velocity_mode.c ==========

```
#include "continuous_torque_velocity_mode.h"
#include "esp_log.h"
#include <string.h>
#include <math.h>
static const char *TAG = "position_recorder";
static uint32_t last_debug_print_time = 0;
#define DEBUG_PRINT_INTERVAL_MS 2000
float climbing_mode_torque = 6.0f;
void position_ring_buffer_init(position_ring_buffer_t *buffer) {
  if (buffer == NULL) {
    ESP_LOGE(TAG, "缓存区指针为空");
    return;
  }
  memset(buffer->buffer, 0, sizeof(buffer->buffer));
  buffer->head = 0;
  buffer->tail = 0;
  buffer->count = 0;
  buffer->last_recorded_position = 0.0f;
  buffer->initialized = true;
}
bool position_ring_buffer_add_if_changed(position_ring_buffer_t *buffer,
                        float new_position,
                        uint32_t timestamp) {
  if (buffer == NULL || !buffer->initialized) {
    ESP_LOGE(TAG, "缓存区未初始化");
    return false;
  }
  float position_change = fabsf(new_position - buffer->last_recorded_position);
  if (position_change >= POSITION_CHANGE_THRESHOLD) {
```

```
    position_ring_buffer_add(buffer, new_position, timestamp);
    buffer->last_recorded_position = new_position;
    return true;
  }
  return false;
}
void position_ring_buffer_add(position_ring_buffer_t *buffer,
                      float position,
                      uint32_t timestamp) {
  if (buffer == NULL || !buffer->initialized) {
    ESP_LOGE(TAG, "缓存区未初始化");
    return;
  }
  buffer->buffer[buffer->head].position = position;
  buffer->buffer[buffer->head].timestamp = timestamp;
  buffer->buffer[buffer->head].valid = true;
  buffer->head = (buffer->head + 1) % POSITION_BUFFER_SIZE;
  if (buffer->count == POSITION_BUFFER_SIZE) {
    buffer->tail = (buffer->tail + 1) % POSITION_BUFFER_SIZE;
  } else {
    buffer->count++;
  }
}
bool position_ring_buffer_get_latest(position_ring_buffer_t *buffer,
                         position_point_t *point) {
  if (buffer == NULL || point == NULL || !buffer->initialized) {
    ESP_LOGE(TAG, "参数无效");
    return false;
  }
  if (buffer->count == 0) {
    return false;
  }
  uint16_t latest_index = (buffer->head - 1 + POSITION_BUFFER_SIZE) % POSITION_BUFFER_SIZE;
  *point = buffer->buffer[latest_index];
  return point->valid;
}
bool position_ring_buffer_get_at_index(position_ring_buffer_t *buffer,
                           int16_t index,
                           position_point_t *point) {
  if (buffer == NULL || point == NULL || !buffer->initialized) {
    ESP_LOGE(TAG, "参数无效");
    return false;
  }
  if (buffer->count == 0 || abs(index) >= buffer->count) {
    return false;
  }
  uint16_t actual_index;
  if (index >= 0) {
    actual_index = (buffer->head - 1 - index + POSITION_BUFFER_SIZE) % POSITION_BUFFER_SIZE;
  } else {
```

```
    actual_index = (buffer->tail - index - 1 + POSITION_BUFFER_SIZE) % POSITION_BUFFER_SIZE;
  }
  *point = buffer->buffer[actual_index];
  return point->valid;
}
uint16_t position_ring_buffer_get_count(position_ring_buffer_t *buffer) {
  if (buffer == NULL || !buffer->initialized) {
    return 0;
  }
  return buffer->count;
}
bool position_ring_buffer_is_empty(position_ring_buffer_t *buffer) {
  if (buffer == NULL || !buffer->initialized) {
    return true;
  }
  return buffer->count == 0;
}
bool position_ring_buffer_is_full(position_ring_buffer_t *buffer) {
  if (buffer == NULL || !buffer->initialized) {
    return false;
  }
  return buffer->count == POSITION_BUFFER_SIZE;
}
float detect_peak_valley_difference(position_ring_buffer_t *buffer) {
  if (buffer == NULL || !buffer->initialized || buffer->count < 3) {
    return -1.0f;
  }
  float max_peak = -999.0f;
  float min_valley = 999.0f;
  bool found_peak = false;
  bool found_valley = false;
  for (uint16_t i = 1; i < buffer->count - 1; i++) {
    uint16_t prev_index = (buffer->tail + i - 1) % POSITION_BUFFER_SIZE;
    uint16_t curr_index = (buffer->tail + i) % POSITION_BUFFER_SIZE;
    uint16_t next_index = (buffer->tail + i + 1) % POSITION_BUFFER_SIZE;
    if (!buffer->buffer[prev_index].valid ||
      !buffer->buffer[curr_index].valid ||
      !buffer->buffer[next_index].valid) {
      continue;
    }
    float prev_pos = buffer->buffer[prev_index].position;
    float curr_pos = buffer->buffer[curr_index].position;
    float next_pos = buffer->buffer[next_index].position;
    if (curr_pos > prev_pos && curr_pos > next_pos) {
      if (curr_pos > max_peak) {
        max_peak = curr_pos;
        found_peak = true;
      }
    }
    else if (curr_pos < prev_pos && curr_pos < next_pos) {
```

```
        if (curr_pos < min_valley) {
          min_valley = curr_pos;
          found_valley = true;
        }
      }
    }
    if (found_peak && found_valley) {
      float diff = max_peak - min_valley;
      ESP_LOGD(TAG, "检测到波峰: %.3f, 波谷: %.3f, 差值: %.3f",
          max_peak, min_valley, diff);
      return diff;
    }
    if (found_peak || found_valley) {
      float first_pos = buffer->buffer[buffer->tail].position;
      float last_pos = buffer->buffer[(buffer->head - 1 + POSITION_BUFFER_SIZE) % POSITION_BUFFER_SIZE].position;
      float effective_max = found_peak ? max_peak : fmaxf(first_pos, last_pos);
      float effective_min = found_valley ? min_valley : fminf(first_pos, last_pos);
      float diff = effective_max - effective_min;
      ESP_LOGD(TAG, "部分波峰波谷检测: 最大=%.3f, 最小=%.3f, 差值=%.3f",
          effective_max, effective_min, diff);
      return diff;
    }
    ESP_LOGD(TAG, "未检测到明显波峰波谷");
    return -1.0f;
}
void position_ring_buffer_clear(position_ring_buffer_t *buffer) {
    if (buffer == NULL || !buffer->initialized) {
      ESP_LOGE(TAG, "缓存区未初始化");
      return;
    }
    buffer->head = 0;
    buffer->tail = 0;
    buffer->count = 0;
    buffer->last_recorded_position = 0.0f;
}
bool position_ring_buffer_get_min_max(position_ring_buffer_t *buffer,
                          float *min_pos, float *max_pos) {
    if (buffer == NULL || min_pos == NULL || max_pos == NULL || !buffer->initialized) {
      ESP_LOGE(TAG, "参数无效或缓存区未初始化");
      return false;
    }
    if (buffer->count == 0) {
      ESP_LOGW(TAG, "缓存区为空，无法获取最大最小值");
      return false;
    }
    float min_value = buffer->buffer[buffer->tail].position;
    float max_value = buffer->buffer[buffer->tail].position;
    for (uint16_t i = 0; i < buffer->count; i++) {
      uint16_t index = (buffer->tail + i) % POSITION_BUFFER_SIZE;
      if (buffer->buffer[index].valid) {
```

```
            float pos = buffer->buffer[index].position;
            if (pos < min_value) {
                min_value = pos;
            }
            if (pos > max_value) {
                max_value = pos;
            }
        }
    }
    *min_pos = min_value;
    *max_pos = max_value;
    return true;
}
motion_mode_t detect_motion_mode(position_ring_buffer_t *buffer,
                                 uint32_t current_timestamp,
                                 motion_mode_state_t *state) {
    if (buffer == NULL || !buffer->initialized || state == NULL) {
        ESP_LOGE(TAG, "缓存区未初始化");
        return MOTION_MODE_STATIC;
    }
    if (buffer->count == 0) {
        return MOTION_MODE_STATIC;
    }
    position_point_t latest_point;
    if (!position_ring_buffer_get_latest(buffer, &latest_point)) {
        ESP_LOGW(TAG, "无法获取最新位置点");
        return MOTION_MODE_STATIC;
    }
    uint32_t time_since_last_update = current_timestamp - latest_point.timestamp;
    if (time_since_last_update > STATIC_TIMEOUT_MS) {
        return MOTION_MODE_STATIC;
    }
    float peak_valley_diff = detect_peak_valley_difference(buffer);
    if (peak_valley_diff < 0) {
        float min_pos, max_pos;
        if (!position_ring_buffer_get_min_max(buffer, &min_pos, &max_pos)) {
            ESP_LOGW(TAG, "无法获取位置范围");
            return MOTION_MODE_STATIC;
        }
        peak_valley_diff = max_pos - min_pos;
    }
    float position_range = peak_valley_diff;
    bool should_print_debug = (current_timestamp - last_debug_print_time) >= DEBUG_PRINT_INTERVAL_MS;
    if (should_print_debug) {
        ESP_LOGI(TAG, "【运动检测】波峰波谷差值: %.3f (数据点数: %d)",
                 position_range, buffer->count);
        last_debug_print_time = current_timestamp;
    }
    if (position_range < STATIC_DETECTION_THRESHOLD) {
        state->climbing_downgrade_count = 0;
```

```
        ESP_LOGI(TAG, "【模式判定】位置变化 %.3f < %.3f，判定为静止模式",
                 position_range, STATIC_DETECTION_THRESHOLD);
        return MOTION_MODE_STATIC;
    } else if (position_range >= MODE_DETECTION_THRESHOLD) {
        state->climbing_downgrade_count = 0;
        ESP_LOGI(TAG, "【模式判定】位置变化 %.3f >= %.3f，判定为爬楼模式",
                 position_range, MODE_DETECTION_THRESHOLD);
        return MOTION_MODE_CLIMBING;
    } else {
        if (state->current_mode == MOTION_MODE_CLIMBING) {
            if (position_range < CLIMBING_DOWNGRADE_THRESHOLD) {
                state->climbing_downgrade_count++;
                ESP_LOGI(TAG, "【爬楼降档】位置变化 %.3f < %.3f，降档计数: %d/6",
                         position_range, CLIMBING_DOWNGRADE_THRESHOLD, state->climbing_downgrade_count);
                if (state->climbing_downgrade_count >= 6) {
                    state->climbing_downgrade_count = 0;
                    ESP_LOGI(TAG, "【模式判定】连续 6 次低于降档阈值，爬楼模式降档到行走模式");
                    return MOTION_MODE_WALKING;
                } else {
                    ESP_LOGI(TAG, "【爬楼保持】还需 %d 次确认才能降档，保持爬楼模式",
                             6 - state->climbing_downgrade_count);
                    return MOTION_MODE_CLIMBING;
                }
            } else {
                state->climbing_downgrade_count = 0;
                ESP_LOGI(TAG, "【爬楼保持】位置变化 %.3f 在保持范围 %.3f - %.3f，保持爬楼模式",
                         position_range, CLIMBING_DOWNGRADE_THRESHOLD, MODE_DETECTION_THRESHOLD);
                return MOTION_MODE_CLIMBING;
            }
        } else {
            state->climbing_downgrade_count = 0;
            ESP_LOGI(TAG, "【模式判定】位置变化 %.3f 在 %.3f - %.3f 之间，判定为行走模式",
                     position_range, STATIC_DETECTION_THRESHOLD, MODE_DETECTION_THRESHOLD);
            return MOTION_MODE_WALKING;
        }
    }
}
void get_motor_params(motion_mode_t mode, int motor_id, motor_params_t *params) {
    if (params == NULL) {
        ESP_LOGE(TAG, "电机参数指针为空");
        return;
    }
    if (motor_id != 1 && motor_id != 2) {
        ESP_LOGE(TAG, "无效的电机 ID: %d", motor_id);
        return;
    }
    switch (mode) {
        case MOTION_MODE_STATIC:
            params->velocity = 0;
            params->torque = 0.0f;
```

```
            params->kd = 0;
            break;
        case MOTION_MODE_WALKING:
            if (motor_id == 1) {
                params->velocity = -2;
                params->torque = -1.5f;
                params->kd = 1;
            } else {
                params->velocity = 2;
                params->torque = 1.5f;
                params->kd = 1;
            }
            break;
        case MOTION_MODE_CLIMBING:
            if (motor_id == 1) {
                params->velocity = -2;
                params->torque = -climbing_mode_torque;
                params->kd = 1;
            } else {
                params->velocity = 2;
                params->torque = climbing_mode_torque;
                params->kd = 1;
            }
            break;
        default:
            ESP_LOGW(TAG, "未知运动模式: %d， 使用静止模式参数", mode);
            params->velocity = 0;
            params->torque = 0.0f;
            params->kd = 0;
            break;
    }
}
void torque_gradient_init(torque_gradient_t *gradient) {
    if (gradient == NULL) {
        ESP_LOGE(TAG, "力矩渐变指针为空");
        return;
    }
    gradient->current_torque = 0.0f;
    gradient->target_torque = 0.0f;
    gradient->last_update_time = 0;
    gradient->is_increasing = false;
}
float update_torque_gradient(torque_gradient_t *gradient,
                             float target_torque,
                             uint32_t current_timestamp) {
    if (gradient == NULL) {
        ESP_LOGE(TAG, "力矩渐变指针为空");
        return 0.0f;
    }
    if (gradient->target_torque != target_torque) {
```

```
        gradient->target_torque = target_torque;
        gradient->last_update_time = current_timestamp;
        gradient->is_increasing = (target_torque > gradient->current_torque);
        ESP_LOGI(TAG, "力矩渐变目标更新: %.1f -> %.1f", gradient->current_torque, target_torque);
    }
    if (fabs(gradient->current_torque - gradient->target_torque) < 0.1f) {
        gradient->current_torque = gradient->target_torque;
        return gradient->current_torque;
    }
    uint32_t time_interval = gradient->is_increasing ? TORQUE_INCREASE_INTERVAL_MS : TORQUE_DECREASE_INTERVAL_MS;
    if (current_timestamp - gradient->last_update_time >= time_interval) {
        float step = gradient->is_increasing ? TORQUE_INCREASE_STEP : TORQUE_DECREASE_STEP;
        if (gradient->is_increasing) {
            gradient->current_torque += step;
            if (gradient->current_torque > gradient->target_torque) {
                gradient->current_torque = gradient->target_torque;
            }
        } else {
            gradient->current_torque -= step;
            if (gradient->current_torque < gradient->target_torque) {
                gradient->current_torque = gradient->target_torque;
            }
        }
        gradient->last_update_time = current_timestamp;
    }
    return gradient->current_torque;
}
void motion_mode_state_init(motion_mode_state_t *state) {
    if (state == NULL) {
        ESP_LOGE(TAG, "状态管理指针为空");
        return;
    }
    state->current_mode = MOTION_MODE_STATIC;
    state->previous_mode = MOTION_MODE_STATIC;
    state->mode_start_timestamp = 0;
    state->is_continuous_mode = false;
    torque_gradient_init(&state->motor1_torque);
    torque_gradient_init(&state->motor2_torque);
    state->last_mode_change_time = 0;
    state->static_confirm_start_time = 0;
    state->in_static_confirmation = false;
    state->last_position_range = 0.0f;
    state->climbing_downgrade_count = 0;
}
void update_motion_mode_and_get_params(motion_mode_state_t *state,
                                       motion_mode_t new_mode,
                                       uint32_t current_timestamp,
                                       int motor_id,
                                       motor_params_t *params) {
    if (state == NULL || params == NULL) {
```

```
    ESP_LOGE(TAG, "参数指针为空");
    return;
}
motion_mode_t actual_mode = state->current_mode;
if (new_mode != state->current_mode) {
    uint32_t time_since_last_change = current_timestamp - state->last_mode_change_time;
    if (time_since_last_change < MODE_LOCK_DURATION_MS) {
        ESP_LOGI(TAG, "【模式锁定】距离上次切换仅%dms（需要%dms），忽略切换请求 %d -> %d",
                 time_since_last_change, MODE_LOCK_DURATION_MS, state->current_mode, new_mode);
        actual_mode = state->current_mode;
    }
    else if (state->current_mode != MOTION_MODE_STATIC && new_mode == MOTION_MODE_STATIC) {
        if (!state->in_static_confirmation) {
            motor_params_t current_params;
            get_motor_params(state->current_mode, motor_id, &current_params);
            torque_gradient_t *gradient = (motor_id == 1) ? &state->motor1_torque : &state->motor2_torque;
            gradient->current_torque = current_params.torque;
            ESP_LOGI(TAG, "电机%d 开始静止确认前同步力矩: %.1f", motor_id, gradient->current_torque);
            state->in_static_confirmation = true;
            state->static_confirm_start_time = current_timestamp;
            actual_mode = MOTION_MODE_STATIC;
        } else {
            uint32_t confirm_duration = current_timestamp - state->static_confirm_start_time;
            if (confirm_duration >= STATIC_CONFIRM_DURATION_MS) {
                actual_mode = MOTION_MODE_STATIC;
                state->in_static_confirmation = false;
                state->previous_mode = state->current_mode;
                state->current_mode = MOTION_MODE_STATIC;
                state->mode_start_timestamp = current_timestamp;
                state->last_mode_change_time = current_timestamp;
                state->is_continuous_mode = false;
            } else {
                actual_mode = MOTION_MODE_STATIC;
            }
        }
    }
    else {
        if (state->in_static_confirmation) {
            state->in_static_confirmation = false;
        }
        if (state->current_mode != MOTION_MODE_STATIC) {
            motor_params_t current_params;
            get_motor_params(state->current_mode, motor_id, &current_params);
            torque_gradient_t *gradient = (motor_id == 1) ? &state->motor1_torque : &state->motor2_torque;
            gradient->current_torque = current_params.torque;
            ESP_LOGI(TAG, "电机%d 模式切换前同步当前力矩: %.1f", motor_id, gradient->current_torque);
        }
        ESP_LOGI(TAG, "【模式切换】运动模式切换: %d -> %d", state->current_mode, new_mode);
        state->previous_mode = state->current_mode;
        state->current_mode = new_mode;
```

```
            state->mode_start_timestamp = current_timestamp;
            state->last_mode_change_time = current_timestamp;
            state->is_continuous_mode = false;
            actual_mode = new_mode;
        }
    } else {
        if (state->in_static_confirmation && new_mode != MOTION_MODE_STATIC) {
            state->in_static_confirmation = false;
        }
        if (!state->is_continuous_mode) {
            state->is_continuous_mode = true;
        }
        actual_mode = state->current_mode;
    }
    get_motor_params(actual_mode, motor_id, params);
    torque_gradient_t *gradient = (motor_id == 1) ? &state->motor1_torque : &state->motor2_torque;
    float target_torque = params->torque;
    float actual_torque = update_torque_gradient(gradient, target_torque, current_timestamp);
    params->torque = actual_torque;
    if (state->is_continuous_mode &&
        (actual_mode == MOTION_MODE_WALKING || actual_mode == MOTION_MODE_CLIMBING)) {
        params->kd = 0;
    }
}
```

// ========== alternating_speed.h ==========

```
#ifndef ALTERNATING_SPEED_H
#define ALTERNATING_SPEED_H
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include <stdbool.h>
#ifdef __cplusplus
extern "C" {
#endif
typedef enum {
    MODE_IDLE = 0,
    MODE_TRANSITION = 1,
    MODE_FLAT_GROUND = 2,
    MODE_STAIRS = 3
} walking_mode_t;
typedef enum {
    MOTION_STATE_IDLE = 0,
    MOTION_STATE_TRANSITION = 1,
    MOTION_STATE_WALKING = 2,
    MOTION_STATE_STAIRS = 3
} motion_state_t;
extern bool alternating_speed_enabled;
extern int alternating_interval_ms;
extern float alternating_speed_x;
extern float alternating_speed_y;
extern float speed_current_limit;
```

```
void Start_Alternating_Speed(void);
void Stop_Alternating_Speed(void);
void Switch_Walking_Mode(walking_mode_t mode);
walking_mode_t Get_Current_Walking_Mode(void);
void Switch_To_Idle_Mode(void);
void Switch_To_Transition_Mode(void);
void Switch_To_Flat_Mode(void);
void Switch_To_Stairs_Mode(void);
void Switch_To_Idle_Mode_Keep_Phase(void);
void Switch_To_Transition_Mode_Keep_Phase(void);
void Switch_To_Flat_Mode_Keep_Phase(void);
void Switch_To_Stairs_Mode_Keep_Phase(void);
motion_state_t Get_Current_Motion_State(void);
void Start_Position_Monitor_Task(void);
void Stop_Position_Monitor_Task(void);
bool Get_Position_Variation_Info(float* motor1_diff, float* motor2_diff);
void Add_Position_Sample(float positions[2]);
extern void set_motor_params(int motor_id, float torque, float position, float speed, float kp, float kd);
void Alternating_Speed_Control_Task(void* pvParameters);
void Motor_Speed_Control_Task(void* pvParameters);
void Start_Motor_Speed_Control(void);
void Stop_Motor_Speed_Control(void);
#ifdef __cplusplus
}
#endif
#endif
```

// ========== alternating_speed.c ==========

```
#include "alternating_speed.h"
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include <math.h>
#include "rs01_motor.h"
static const char *TAG = "alternating_speed";
bool alternating_speed_enabled = false;
static int speed_state = 0;
static TickType_t last_switch_time = 0;
int alternating_interval_ms = 800;
float alternating_speed_x = 2.25f;
float alternating_speed_y = 1.5f;
float speed_current_limit = 0.0f;
#define IDLE_INTERVAL_MS 1000
#define IDLE_SPEED_X 0.0f
#define IDLE_SPEED_Y 0.0f
#define IDLE_CURRENT_LIMIT 0.0f
#define TRANSITION_INTERVAL_MS 800
#define TRANSITION_SPEED_X 1.0f
#define TRANSITION_SPEED_Y 1.0f
#define TRANSITION_CURRENT_LIMIT 1.0f
#define FLAT_INTERVAL_MS 800
```