

# 《智能植保微环境控制器》接口文档

参考，并非最终版本，不需要全部都看，项目中已经搭建好了相关结构与接口

## 1. 项目目录结构

```
SmartPlantGuard/
├── README.md                      # 项目说明文档
├── .gitignore                      # Git 忽略文件
├── CMakeLists.txt                  # CLion 项目配置文件
├── STM32CubeMX/                    # CubeMX 工程文件
|   ├── .mxproject
|   └── SmartPlantGuard.ioc
└── Core/
    ├── Inc/
    |   ├── main.h
    |   ├── stm32xxxx_hal_conf.h
    |   └── freertos.h               # FreeRTOS 配置（如果使用）
    └── Src/
        ├── main.c
        ├── stm32xxxx_hal_msp.c
        ├── stm32xxxx_it.c
        └── freertos.c              # FreeRTOS 任务
├── Drivers/
|   ├── STM32xxxx_HAL_Driver/
|   └── CMSIS/
└── Modules/
    ├── Sensor/
    |   ├── Inc/
    |   |   ├── sensor_manager.h    # 传感器管理器
    |   |   ├── soil_moisture.h    # 土壤湿度传感器
    |   |   ├── dht11.h            # DHT11 温湿度
    |   |   ├── light_sensor.h     # 光敏传感器
    |   |   └── sensor_types.h    # 传感器数据结构和枚举
    |   └── Src/
    |       ├── sensor_manager.c
    |       ├── soil_moisture.c
    |       ├── dht11.c
    |       └── light_sensor.c
    └── Actuator/
        ├── Inc/
        |   ├── actuator_manager.h   # 执行器管理器
        |   ├── relay_driver.h      # 继电器驱动
        |   ├── pwm_driver.h        # PWM 驱动（风扇）
        |   └── actuator_types.h    # 执行器数据结构和枚举
        └── Src/
            ├── actuator_manager.c
            ├── relay_driver.c
            └── pwm_driver.c
    └── Controller/
        ├── Inc/
        |   ├── controller_core.h   # 核心控制逻辑
```

```

|   |   |   └── hysteresis_logic.h      # 滞回控制算法
|   |   |   └── task_scheduler.h        # 任务调度器
|   |   |   └── storage_flash.h        # Flash 存储
|   |   |   └── controller_types.h    # 控制器数据结构
|   |   └── Src/
|   |       ├── controller_core.c
|   |       ├── hysteresis_logic.c
|   |       ├── task_scheduler.c
|   |       └── storage_flash.c
|   └── Communication/
|       ├── Inc/
|       |   ├── communication_manager.h # 通信管理器
|       |   ├── bluetooth_hc05.h        # 蓝牙模块
|       |   ├── wifi_esp8266.h         # WiFi模块（预留）
|       |   └── protocol.h            # 通信协议定义
|       └── Src/
|           ├── communication_manager.c
|           ├── bluetooth_hc05.c
|           └── wifi_esp8266.c
└── System/
    ├── Inc/
    |   ├── system_config.h          # 系统配置（阈值、引脚等）
    |   ├── system_state.h          # 系统状态管理
    |   └── error_handler.h         # 错误处理
    └── Src/
        ├── system_config.c
        ├── system_state.c
        └── error_handler.c
└── Tools/
└── Docs/

```

## 2. API 接口

### 2.1 系统配置与状态管理 (System/)

#### 2.1.1 系统配置 (system\_config.h)

```

#ifndef SYSTEM_CONFIG_H
#define SYSTEM_CONFIG_H

// 引脚定义
// 传感器引脚 (Port A)
// 注意：土壤和光敏必须接 3.3V，绝对不能接 5V！
#define SOIL_MOISTURE_ADC_CHANNEL    ADC_CHANNEL_0    // PA0
#define LIGHT_SENSOR_ADC_CHANNEL     ADC_CHANNEL_1    // PA1
#define DHT11_PORT                  GPIOB
#define DHT11_PIN                   GPIO_PIN_12      // PB12
// 通信引脚 (Port A)
// 蓝牙 (USART1)
#define BLUETOOTH_UART              USART1
#define BLUETOOTH_TX_PORT           GPIOA
#define BLUETOOTH_TX_PIN            GPIO_PIN_9       // PA9
#define BLUETOOTH_RX_PORT           GPIOA

```

```

#define BLUETOOTH_RX_PIN           GPIO_PIN_10 // PA10
// 执行器引脚 (Port B)
// 继电器全部移至 Port B, 减少对 ADC 的干扰
#define RELAY_PUMP_PORT           GPIOB
#define RELAY_PUMP_PIN             GPIO_PIN_13 // PB13
#define RELAY_FAN_PORT             GPIOB
#define RELAY_FAN_PIN              GPIO_PIN_14 // PB14
// 风扇 PWM
#define FAN_PWM_PORT               GPIOB
#define FAN_PWM_PIN                GPIO_PIN_0 // PB0
#define FAN_PWM_TIM                TIM3
#define FAN_PWM_CHANNEL             TIM_CHANNEL_3
// 系统引脚
// 板载 LED (PC13), 低电平点亮
#define SYSTEM_LED_PORT            GPIOC
#define SYSTEM_LED_PIN              GPIO_PIN_13
// WiFi 预留 (USART2)
// #define WIFI_UART                USART2
// #define WIFI_TX_PIN               GPIO_PIN_2 // PA2
// #define WIFI_RX_PIN               GPIO_PIN_3 // PA3

// 默认阈值配置
#define DEFAULT_SOIL_MOISTURE_LOW   30.0f // 土壤湿度低于 30% 开启水泵
#define DEFAULT_SOIL_MOISTURE_HIGH    40.0f // 土壤湿度高于 40% 关闭水泵
#define DEFAULT_TEMP_HIGH            30.0f // 温度高于 30°C 开启风扇
#define DEFAULT_TEMP_LOW             25.0f // 温度低于 25°C 关闭风扇

// 系统参数
#define SYSTEM_TICK_MS              1000 // 主循环周期
#define HYSTERESIS_BAND              2.0f // 滞回带宽
#define MAX_RETRY_COUNT                3 // 传感器重试次数

#endif // SYSTEM_CONFIG_H

```

引脚仅作参考，非最终版本，整合的时候会根据实际情况评估，接线之前先问问 AI，需注意电路连接

引脚	功能	外设	状态	备注
PA0	土壤湿度	ADC1_IN0	✓	必须接 3.3V
PA1	光敏传感器	ADC1_IN1	✓	必须接 3.3V
PA9	蓝牙 TX	USART1_TX	✓	
PA10	蓝牙 RX	USART1_RX	✓	
PB0	风扇 PWM	TIM3_CH3	✓	注意复用功能
PB12	DHT11 数据	GPIO	⚠	需外部上拉，3.3V 电平
PB13	水泵继电器	GPIO	✓	
PB14	风扇继电器	GPIO	✓	
PC13	系统 LED	GPIO	✓	低电平点亮

引脚	功能	外设	状态	备注
PA13	SWDIO	调试	✓	保留
PA14	SWCLK	调试	✓	保留

**关于 PB13 / PB14 的 JTAG 问题：**不用管它，直接当普通 GPIO 用即可。只要不去动 PA13、PA14、PA15、PB3、PB4，调试功能（SWD）就不会受影响

## 2.1.2 系统状态枚举 (system\_state.h)

```
#ifndef SYSTEM_STATE_H
#define SYSTEM_STATE_H

#include <stdbool.h>

// 系统运行状态
typedef enum {
    SYS_STATE_INIT = 0,           // 初始化状态
    SYS_STATE_NORMAL,            // 正常运行
    SYS_STATE_AUTO,              // 自动控制模式
    SYS_STATE_MANUAL,            // 手动控制模式
    SYS_STATE_ERROR,              // 错误状态
    SYS_STATE_CALIBRATING        // 校准状态
} SystemStateEnum;

// 错误码定义
typedef enum {
    ERROR_NONE = 0,
    ERROR_SENSOR_FAILURE,        // 传感器故障
    ERROR_ACTUATOR_FAILURE,      // 执行器故障
    ERROR_COMMUNICATION_FAIL,   // 通信故障
    ERROR_POWER_LOW,             // 电源电压低
    ERROR_WATER_LOW              // 水箱缺水
} ErrorCodeEnum;

// 控制模式
typedef enum {
    MODE_AUTO = 0,               // 自动模式
    MODE_MANUAL,                 // 手动模式
    MODE_CALIBRATION             // 校准模式
} ControlModeEnum;

// 系统状态结构体
typedef struct {
    SystemStateEnum currentState;
    ErrorCodeEnum lastError;
    ControlModeEnum controlMode;
    bool isSystemRunning;
    uint32_t uptimeSeconds;       // 系统运行时间
    uint8_t retryCount;           // 重试计数器
} SystemState;

// 全局状态变量
extern SystemState gSystemState;
```

```

// 函数声明
void SystemState_Init(void);
void SystemState_Update(SystemStateEnum newState);
void SystemState_SetError(ErrorCodeEnum error);
void SystemState_ClearError(void);
const char* SystemState_GetStateString(SystemStateEnum state);
const char* SystemState_GetErrorString(ErrorCodeEnum error);

#endif // SYSTEM_STATE_H

```

## 2.2 传感器模块接口 (Sensor/)

### 2.2.1 传感器数据类型 (sensor\_types.h)

```

#ifndef SENSOR_TYPES_H
#define SENSOR_TYPES_H

// 传感器数据单位
typedef enum {
    UNIT_PERCENT = 0,      // 百分比
    UNIT_CELSIUS,          // 摄氏度
    UNIT_LUX,               // 勒克斯
    UNIT_RAW                 // 原始值
} SensorUnitEnum;

// 传感器状态
typedef enum {
    SENSOR_OK = 0,
    SENSOR_NOT_CONNECTED,
    SENSOR_OUT_OF_RANGE,
    SENSOR_TIMEOUT,
    SENSOR_CHECKSUM_ERROR
} SensorStatusEnum;

// 传感器数据包结构体
typedef struct {
    float value;           // 传感器数值
    SensorUnitEnum unit;   // 数值单位
    SensorStatusEnum status; // 传感器状态
    uint32_t timestamp;    // 时间戳
    float minValue;        // 最小值 (用于校准)
    float maxValue;        // 最大值 (用于校准)
} SensorData;

// 所有传感器数据集合
typedef struct {
    SensorData soilMoisture; // 土壤湿度 (0-100%)
    SensorData temperature;  // 温度 (°C)
    SensorData humidity;    // 湿度 (%)
    SensorData lightIntensity; // 光照强度
    bool allSensorsValid;   // 所有传感器是否有效
    uint32_t lastUpdateTime; // 最后更新时间
} AllSensorData;

```

```
#endif // SENSOR_TYPES_H
```

## 2.2.2 传感器管理器接口 (sensor\_manager.h)

```
#ifndef SENSOR_MANAGER_H
#define SENSOR_MANAGER_H

#include "sensor_types.h"

// 传感器管理器状态
typedef struct {
    bool isInitialized;
    uint8_t readInterval;           // 读取间隔(秒)
    bool autoCalibration;          // 自动校准
    uint32_t totalReadCount;        // 总读取次数
    uint32_t errorCount;            // 错误计数
} SensorManagerStatus;

// 初始化传感器系统
SensorStatusEnum SensorManager_Init(void);

// 读取所有传感器数据
bool SensorManager_ReadAll(AllSensorData* sensorData);

// 读取单个传感器
SensorStatusEnum SensorManager_ReadSoilMoisture(float* moisture);
SensorStatusEnum SensorManager_ReadTemperature(float* temperature);
SensorStatusEnum SensorManager_ReadHumidity(float* humidity);
SensorStatusEnum SensorManager_ReadLightIntensity(float* light);

// 校准函数
bool SensorManager_CalibrateSoilMoisture(float dryValue, float wetValue);
bool SensorManager_CalibrateLightSensor(float minLux, float maxLux);

// 设置读取间隔
void SensorManager_SetReadInterval(uint8_t seconds);

// 获取管理器状态
SensorManagerStatus SensorManager_GetStatus(void);

// 重置传感器统计
void SensorManager_ResetStatistics(void);

#endif // SENSOR_MANAGER_H
```

## 2.2.3 DHT11 温湿度传感器接口 (dht11.h)

```
#ifndef DHT11_H
#define DHT11_H

#include "sensor_types.h"

// DHT11 特定错误码
typedef enum {
```

```

DHT11_OK = 0,
DHT11_NO_RESPONSE,
DHT11_CHECKSUM_ERROR,
DHT11_TIMEOUT_ERROR
} DHT11_StatusEnum;

// DHT11 初始化
DHT11_StatusEnum DHT11_Init(GPIO_TypeDef* port, uint16_t pin);

// 读取温湿度数据
DHT11_StatusEnum DHT11_Read(float* temperature, float* humidity);

// 获取最后一次读取的状态
DHT11_StatusEnum DHT11_GetLastStatus(void);

// 获取读取统计
void DHT11_GetStatistics(uint32_t* successCount, uint32_t* errorCount);

#endif // DHT11_H

```

## 2.2.4 土壤湿度传感器接口 (soil\_moisture.h)

```

#ifndef SOIL_MOISTURE_H
#define SOIL_MOISTURE_H

#include "sensor_types.h"

// 土壤湿度校准结构
typedef struct {
    float dryValue;      // 干燥时的 ADC 值
    float wetValue;      // 湿润时的 ADC 值
    bool isCalibrated;  // 是否已校准
} SoilCalibration;

// 初始化土壤湿度传感器
bool SoilMoisture_Init(ADC_HandleTypeDef* hadc, uint32_t channel);

// 读取土壤湿度
SensorStatusEnum SoilMoisture_Read(float* moisturePercent);

// 校准函数
void SoilMoisture_CalibrateDry(void); // 校准干燥值
void SoilMoisture_CalibrateWet(void); // 校准湿润值
SoilCalibration SoilMoisture_GetCalibration(void);

// 设置自定义校准值
void SoilMoisture_SetCalibration(float dryValue, float wetValue);

#endif // SOIL_MOISTURE_H

```

## 2.3 执行器模块接口 (Actuator/)

### 2.3.1 执行器数据类型 (actuator\_types.h)

```
#ifndef ACTUATOR_TYPES_H
#define ACTUATOR_TYPES_H

// 执行器类型
typedef enum {
    ACTUATOR_RELAY = 0,      // 继电器
    ACTUATOR_PWM,           // PWM 设备
    ACTUATOR_SERVO          // 舵机（预留）
} ActuatorTypeEnum;

// 执行器状态
typedef enum {
    ACTUATOR_OFF = 0,
    ACTUATOR_ON,
    ACTUATOR_ERROR
} ActuatorStateEnum;

// 执行器配置
typedef struct {
    ActuatorTypeEnum type;
    GPIO_TypeDef* port;
    uint16_t pin;
    uint8_t pwmChannel;      // 如果是 PWM 设备
    uint16_t minDutyCycle;   // 最小占空比
    uint16_t maxDutyCycle;   // 最大占空比
} ActuatorConfig;

// 执行器状态信息
typedef struct {
    ActuatorConfig config;
    ActuatorStateEnum currentState;
    uint32_t totalOnTime;    // 总开启时间（秒）
    uint32_t operationCount; // 操作次数
    bool isFaulty;          // 是否故障
} ActuatorStatus;

#endif // ACTUATOR_TYPES_H
```

### 2.3.2 执行器管理器接口 (actuator\_manager.h)

```
#ifndef ACTUATOR_MANAGER_H
#define ACTUATOR_MANAGER_H

#include "actuator_types.h"

// 执行器ID
typedef enum {
    ACTUATOR_ID_PUMP = 0,     // 水泵
    ACTUATOR_ID_FAN,          // 风扇
    ACTUATOR_ID_LIGHT,         // 补光灯（预留）
}
```

```

    ACTUATOR_ID_COUNT          // 执行器总数
} ActuatorIDEnum;

// 初始化执行器系统
bool ActuatorManager_Init(void);

// 控制执行器
bool ActuatorManager_SetState(ActuatorIDEnum id, ActuatorStateEnum state);
bool ActuatorManager_SetPWM(ActuatorIDEnum id, uint16_t dutyCycle); // 0-1000

// 获取执行器状态
ActuatorStateEnum ActuatorManager_GetState(ActuatorIDEnum id);
ActuatorStatus ActuatorManager_GetStatus(ActuatorIDEnum id);

// 安全控制函数
bool ActuatorManager_SafeToggle(ActuatorIDEnum id, uint32_t minInterval);
bool ActuatorManager_CheckOverheat(void); // 检查过热保护

// 重置统计信息
void ActuatorManager_ResetStatistics(ActuatorIDEnum id);

#endif // ACTUATOR_MANAGER_H

```

### 2.3.3 继电器驱动接口 (relay\_driver.h)

```

#ifndef RELAY_DRIVER_H
#define RELAY_DRIVER_H

#include "actuator_types.h"

// 继电器配置
typedef struct {
    GPIO_TypeDef* controlPort; // 控制端口
    uint16_t controlPin; // 控制引脚
    bool activeHigh; // 是否高电平有效
    uint32_t maxOnTime; // 最大开启时间（秒），0 表示无限制
    uint32_t minOffTime; // 最小关闭时间（秒）
} RelayConfig;

// 继电器状态
typedef struct {
    RelayConfig config;
    ActuatorStateEnum state;
    uint32_t lastToggleTime; // 最后切换时间
    bool safetyLock; // 安全锁（防止频繁切换）
} RelayStatus;

// 初始化继电器
bool RelayDriver_Init(RelayConfig* config);

// 继电器控制
bool RelayDriver_Set(RelayConfig* config, ActuatorStateEnum state);
ActuatorStateEnum RelayDriver_GetState(RelayConfig* config);

// 安全控制

```

```

bool RelayDriver_SafeToggle(RelayConfig* config);
bool RelayDriver_CheckSafety(RelayConfig* config);

#endif // RELAY_DRIVER_H

```

## 2.4 控制器模块接口 (Controller/)

### 2.4.1 控制器数据类型 (controller\_types.h)

```

#ifndef CONTROLLER_TYPES_H
#define CONTROLLER_TYPES_H

// 控制参数结构
typedef struct {
    // 土壤湿度控制
    float soilMoistureLow;      // 低阈值（开启水泵）
    float soilMoistureHigh;     // 高阈值（关闭水泵）

    // 温度控制
    float temperatureHigh;      // 高阈值（开启风扇）
    float temperatureLow;        // 低阈值（关闭风扇）

    // 光照控制（预留）
    float lightIntensityLow;    // 低阈值（开启补光）
    float lightIntensityHigh;   // 高阈值（关闭补光）

    // 滞回控制参数
    float hysteresisBand;       // 滞回带宽

    // 时间控制
    uint32_t minPumpInterval;   // 水泵最小间隔（秒）
    uint32_t maxPumpDuration;   // 水泵最大持续时间（秒）
} ControlParams;

// 控制决策结果
typedef struct {
    bool needWatering;          // 需要浇水
    bool needCooling;           // 需要降温
    bool needLighting;          // 需要补光
    char decisionReason[64];    // 决策原因描述
} ControlDecision;

#endif // CONTROLLER_TYPES_H

```

### 2.4.2 核心控制逻辑接口 (controller\_core.h)

```

#ifndef CONTROLLER_CORE_H
#define CONTROLLER_CORE_H

#include "sensor_types.h"
#include "actuator_types.h"
#include "controller_types.h"

// 控制器初始化

```

```

bool ControllerCore_Init(void);

// 主控制循环（在 FreeRTOS 任务或主循环中调用）
void ControllerCore_RunCycle(void);

// 设置控制模式
bool ControllerCore_SetMode(ControlModeEnum mode);

// 手动控制接口
bool ControllerCore_ManualControl(ActuatorIDEnum actuator, ActuatorStateEnum state);
bool ControllerCore_ManualPWM(ActuatorIDEnum actuator, uint16_t dutyCycle);

// 获取控制参数
ControlParams ControllerCore_GetParams(void);
bool ControllerCore_SetParams(ControlParams* newParams);

// 重置为默认参数
void ControllerCore_ResetToDefaults(void);

// 获取控制决策信息
ControlDecision ControllerCore_GetLastDecision(void);

#endif // CONTROLLER_CORE_H

```

### 2.4.3 滞回控制算法接口 (hysteresis\_logic.h)

```

#ifndef HYSTERESIS_LOGIC_H
#define HYSTERESIS_LOGIC_H

// 滞回控制上下文
typedef struct {
    float lastOutputValue;           // 上次输出值
    bool lastOutputState;            // 上次输出状态
    float highThreshold;             // 高阈值
    float lowThreshold;              // 低阈值
    float hysteresisBand;            // 滞回带宽
    uint32_t minStateTime;           // 最小状态保持时间 (ms)
    uint32_t stateStartTime;          // 状态开始时间
} HysteresisContext;

// 初始化滞回控制器
void Hysteresis_Init(HysteresisContext* ctx, float lowThresh, float highThresh,
                      float hysteresis);

// 滞回控制决策
bool Hysteresis_Update(HysteresisContext* ctx, float currentValue);

// 强制设置状态（绕过滞回）
void Hysteresis_ForceState(HysteresisContext* ctx, bool state);

// 获取滞回状态
bool Hysteresis_GetState(HysteresisContext* ctx);
float Hysteresis_GetThresholds(HysteresisContext* ctx, float* low, float* high);

```

```

// 设置最小状态保持时间
void Hysteresis_SetMinStateTime(HysteresisContext* ctx, uint32_t minTimeMs);

#endif // HYSTERESIS_LOGIC_H

```

## 2.4.4 Flash 存储接口 (storage\_flash.h)

```

#ifndef STORAGE_FLASH_H
#define STORAGE_FLASH_H

#include "controller_types.h"

// 存储的数据结构
typedef struct {
    ControlParams controlParams; // 控制参数
    uint32_t magicNumber; // 魔数验证
    uint32_t crc32; // CRC 校验
    uint32_t saveCount; // 保存次数
    uint32_t lastSaveTime; // 最后保存时间
} SystemConfig;

// 存储初始化
bool StorageFlash_Init(void);

// 保存配置到 Flash
bool StorageFlash_SaveConfig(SystemConfig* config);

// 从 Flash 加载配置
bool StorageFlash_LoadConfig(SystemConfig* config);

// 恢复默认设置
bool StorageFlash_RestoreDefaults(void);

// 擦除存储
bool StorageFlash_Erase(void);

// 获取存储状态
bool StorageFlash_IsValid(void);
uint32_t StorageFlash_GetSaveCount(void);

#endif // STORAGE_FLASH_H

```

## 2.5 通信模块接口 (Communication/)

### 2.5.1 通信协议定义 (protocol.h)

```

#ifndef PROTOCOL_H
#define PROTOCOL_H

// 指令类型
typedef enum {
    CMD_GET_SENSOR_DATA = 0x01, // 获取传感器数据
    CMD_GET_ACTUATOR_STATE = 0x02, // 获取执行器状态
    CMD_SET_ACTUATOR = 0x03, // 设置执行器
} Command;

```

```

CMD_SET_PARAMS = 0x04,           // 设置参数
CMD_GET_PARAMS = 0x05,           // 获取参数
CMD_RESET = 0x06,                // 复位
CMD_CALIBRATE = 0x07,           // 校准
CMD_GET_SYSTEM_INFO = 0x08,      // 获取系统信息
CMD_ACK = 0x09,                  // 确认
CMD_ERROR = 0x0A                 // 错误
} CommandTypeEnum;

// 指令结构
#pragma pack(push, 1)
typedef struct {
    uint8_t startByte;           // 起始字节 0xAA
    uint8_t command;             // 指令类型
    uint8_t dataLength;          // 数据长度
    uint8_t data[32];            // 数据
    uint8_t checksum;            // 校验和
    uint8_t endByte;             // 结束字节 0x55
} CommandPacket;
#pragma pack(pop)

// 响应结构
typedef struct {
    CommandTypeEnum command;
    bool success;
    uint8_t data[32];
    uint8_t dataLength;
} Response;

// 协议处理函数
bool Protocol_ParsePacket(uint8_t* buffer, uint16_t length, CommandPacket* packet);
bool Protocol_ValidatePacket(CommandPacket* packet);
uint8_t Protocol_CalculateChecksum(uint8_t* data, uint8_t length);
Response Protocol_ProcessCommand(CommandPacket* packet);

#endif // PROTOCOL_H

```

## 2.5.2 蓝牙通信接口 (bluetooth\_hc05.h)

```

#ifndef BLUETOOTH_HC05_H
#define BLUETOOTH_HC05_H

#include "protocol.h"

// 蓝牙模块状态
typedef enum {
    BT_STATE_DISCONNECTED = 0,
    BT_STATE_CONNECTING,
    BT_STATE_CONNECTED,
    BT_STATE_ERROR
} BluetoothStateEnum;

// 蓝牙配置
typedef struct {

```

```

    char deviceName[16];           // 设备名称
    char pinCode[8];              // 配对码
    uint32_t baudRate;            // 波特率
} BluetoothConfig;

// 蓝牙管理器状态
typedef struct {
    BluetoothStateEnum state;
    bool isPaired;
    uint32_t bytesReceived;
    uint32_t bytesSent;
    uint32_t connectCount;
} BluetoothStatus;

// 初始化蓝牙模块
bool Bluetooth_Init(UART_HandleTypeDef* huart, BluetoothConfig* config);

// 发送数据
bool Bluetooth_SendData(uint8_t* data, uint16_t length);
bool Bluetooth_SendPacket(CommandPacket* packet);

// 接收处理（在串口中断中调用）
void Bluetooth_ReceiveByte(uint8_t byte);
bool Bluetooth_ProcessReceivedData(void);

// 获取状态
BluetoothStatus Bluetooth_GetStatus(void);

// 配置蓝牙模块
bool Bluetooth_SetDeviceName(const char* name);
bool Bluetooth_SetPinCode(const char* pin);

#endif // BLUETOOTH_HC05_H

```

## 2.6 任务调度器接口 (task\_scheduler.h)

```

#ifndef TASK_SCHEDULER_H
#define TASK_SCHEDULER_H

// 任务 ID
typedef enum {
    TASK_ID_SENSOR_READ = 0,      // 传感器读取任务
    TASK_ID_CONTROL_LOGIC,        // 控制逻辑任务
    TASK_ID_COMMUNICATION,        // 通信任务
    TASK_ID_SAFETY_CHECK,         // 安全检查任务
    TASK_ID_STATISTICS,           // 统计任务
    TASK_ID_COUNT                 // 任务总数
} TaskIDEnum;

// 任务状态
typedef enum {
    TASK_READY = 0,
    TASK_RUNNING,
    TASK_WAITING,
    TASK_ERROR
}

```

```

} TaskStateEnum;

// 任务配置
typedef struct {
    TaskIDEnum id;
    char name[16];
    uint32_t intervalMs;      // 执行间隔
    uint32_t lastRunTime;     // 上次执行时间
    TaskStateEnum state;
    bool enabled;             // 是否启用
    void (*taskFunction)(void); // 任务函数
} TaskConfig;

// 初始化任务调度器
bool TaskScheduler_Init(void);

// 添加任务
bool TaskScheduler_AddTask(TaskConfig* config);

// 启动调度器
void TaskScheduler_Start(void);

// 停止调度器
void TaskScheduler_Stop(void);

// 任务执行（在主循环中调用）
void TaskScheduler_Run(void);

// 获取任务状态
TaskStateEnum TaskScheduler_GetTaskState(TaskIDEnum id);

// 启用/禁用任务
void TaskScheduler_EnableTask(TaskIDEnum id, bool enable);

#endif // TASK_SCHEDULER_H

```

## 3. 使用示例

### 3.1 系统初始化示例

```

// main.c 中的初始化代码
int main(void) {
    // HAL 初始化
    HAL_Init();
    SystemClock_Config();

    // 初始化系统状态
    SystemState_Init();

    // 初始化传感器
    SensorManager_Init();
}

```

```

// 初始化执行器
ActuatorManager_Init();

// 初始化控制器
ControllerCore_Init();

// 初始化蓝牙
BluetoothConfig btConfig = {
    .deviceName = "SmartPlant",
    .pinCode = "1234",
    .baudRate = 9600
};
Bluetooth_Init(&huart1, &btConfig);

// 初始化任务调度器
TaskScheduler_Init();

// 添加任务
TaskConfig sensorTask = {
    .id = TASK_ID_SENSOR_READ,
    .name = "SensorRead",
    .intervalMs = 2000,
    .taskFunction = SensorRead_Task
};
Taskscheduler_AddTask(&sensorTask);

// 启动调度器
TaskScheduler_Start();

// 主循环
while (1) {
    TaskScheduler_Run();

    // 空闲时进入低功耗模式（如果支持）
    HAL_Delay(1);
}
}

```

## 3.2 滞回控制使用示例

```

// 在控制逻辑中使用滞回控制
void ControlLogic_Task(void) {
    static HysteresisContext pumpControl;
    static HysteresisContext fanControl;

    // 初始化滞回控制器
    ControlParams params = ControllerCore_GetParams();
    Hysteresis_Init(&pumpControl,
                    params.soilMoistureLow,
                    params.soilMoistureHigh,
                    params.hysteresisBand);

    Hysteresis_Init(&fanControl,
                    params.temperatureLow,
                    params.temperatureHigh,

```

```

    params.hysteresisBand);

// 获取传感器数据
AllSensorData sensorData;
SensorManager_ReadAll(&sensorData);

// 应用滞回控制
bool shouldPump = Hysteresis_Update(&pumpControl,
                                      sensorData.soilMoisture.value);
bool shouldFan = Hysteresis_Update(&fanControl,
                                    sensorData.temperature.value);

// 控制执行器
ActuatorManager_SetState(ACTUATOR_ID_PUMP,
                         shouldPump ? ACTUATOR_ON : ACTUATOR_OFF);
ActuatorManager_SetState(ACTUATOR_ID_FAN,
                         shouldFan ? ACTUATOR_ON : ACTUATOR_OFF);
}

```

### 3.3 蓝牙指令处理示例

```

// 串口接收中断处理
void USART1_IRQHandler(void) {
    if (__HAL_UART_GET_FLAG(&huart1, UART_FLAG_RXNE)) {
        uint8_t byte = huart1.Instance->DR;
        Bluetooth_ReceiveByte(byte);
    }
}

// 主循环中处理蓝牙数据
void Communication_Task(void) {
    if (Bluetooth_ProcessReceivedData()) {
        // 获取接收到的数据包
        CommandPacket packet;
        if (Protocol_ParsePacket(rxBuffer, rxLength, &packet)) {
            // 处理指令
            Response response = Protocol_ProcessCommand(&packet);

            // 发送响应
            if (response.success) {
                CommandPacket ackPacket = {
                    .startByte = 0xAA,
                    .command = CMD_ACK,
                    .dataLength = response.dataLength,
                    .endByte = 0x55
                };
                memcpy(ackPacket.data, response.data, response.dataLength);
                ackPacket.checksum = Protocol_CalculateChecksum(
                    ackPacket.data, ackPacket.dataLength);

                Bluetooth_SendPacket(&ackPacket);
            }
        }
    }
}

```

