# 海克X3Q嵌入式软件方案设计文档

拟制：叶大鹏

审核：叶大鹏

海克智动科技开发有限公司

版权所有 侵权必究

修订记录

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | **版本** | **修订说明** | **修订人** |
| 2020.10.11 | V1.0 | 第一稿完成 |  |
| 2020.11.11 | V1.1 | 修改若干接口 |  |
|  |  |  |  |
|  |  |  |  |

目录

# 简介

### 范围

本文档描述X3Q产品嵌入式软件框架结构，业务流程，组件的接口和使用方法，组件的内部运行流程以及依赖关系。

文档提供给X3Q开发项目团队以及项目依赖团队使用，包含需求分析人员，方案设计人员，开发及测试人员。

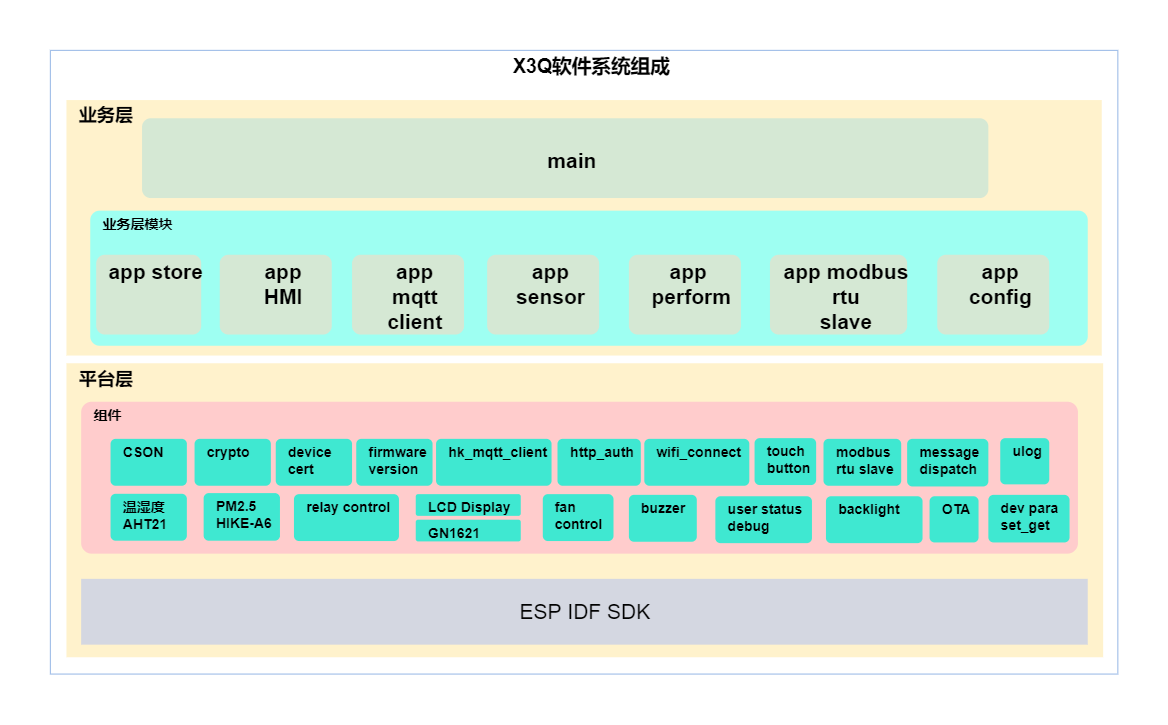
### 背景

附上《xxx需求分析说明文档》

### 缩略语

|  |  |  |
| --- | --- | --- |
| 缩略语 | 英文全名 | 中文解释 |
|  |  |  |
|  |  |  |
|  |  |  |

# 场景和流程

1. 

### 系统组成

### 业务流程图



# 模块(组件)及接口说明

### dev\_cert组件及接口说明

dev\_cert组件主要提供了设备的凭证码保存和读取的功能，凭证码为设备的三元素码：设备码，产品码和设备密钥；设备的批次码；设备配网码：soft ap SSID，soft ap PASSWORD。

dev\_cert组件还提供了验证凭证码完整性的功能，这个功能集合了设备凭证码验证，下载，保存于一体，用户可直接调用此接口，自动完成所有的凭证码操作。

凭证码验证完整性接口的流程见下图：



#### 接口数据结构定义

设备三元素数据结构：

struct dev\_cert\_t

{

   char pid[32];//产品码

   char did[32];//设备码

   char sid[32];//设备密钥

};

设备配网soft ap数据结构：

struct dev\_wifi\_cfg\_ap\_info\_t

{

   char gateway[64];//网关

   char ssid[32];//热点名称

   char password[32];//热点密码

};

#### 读取设备三码接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | int hk\_get\_dev\_cert(struct dev\_cert\_t \*cert) | |
| 描述 | 从flash读取设备的三码 | |
| 参数 | \*cert | 参数类型：输出  设备三码结构体变量回写地址 |
| 返回值 | 0: 读取成功  -1: 读取失败 | |

#### 设置设备三码接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | int hk\_set\_dev\_cert(struct dev\_cert\_t \*cert) | |
| 描述 | 设置设备的三码，并保存到flash | |
| 参数 | \*cert | 参数类型：输入  设备三码结构体变量地址 |
| 返回值 | 0: 设置成功  -1: 设置失败 | |

#### 读取设备批次码接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | int hk\_get\_batch\_code(char \*batch\_code, int \*len) | |
| 描述 | 从flash读取设备的批次码 | |
| 参数 | \*batch\_code | 参数类型：输出  设备批次码字符串回写地址 |
| \*len | 参数类型: 输出  批次码字符串的长度回写地址 |
| 返回值 | 0: 读取成功  -1: 读取失败 | |

#### 设置设备批次码接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | int hk\_set\_dev\_batch\_code(const char \*batch\_code) | |
| 描述 | 设置设备的批次码，并保存到flash | |
| 参数 | \*batch\_code | 参数类型：输入  设备批次码字符串地址 |
| 返回值 | 0: 设置成功  -1: 设置失败 | |

#### 读取设备配网ap信息接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | int hk\_get\_batch\_code(char \*batch\_code, int \*len) | |
| 描述 | 从flash读取设备的配网ap信息 | |
| 参数 | \*ap\_info | 参数类型：输出  配网ap热点信息的结构体回写地址 |
| 返回值 | 0: 读取成功  -1: 读取失败 | |

#### 设置设备配网ap信息接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | int hk\_set\_dev\_ap\_info(struct dev\_wifi\_cfg\_ap\_info\_t \*ap\_info) | |
| 描述 | 设置设备的配网softap信息，并保存到flash | |
| 参数 | \*ap\_info | 参数类型：输入  配网ap信息结构 |
| 返回值 | 0: 设置成功  -1: 设置失败 | |

#### 验证设备信息完整性接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | int hk\_verify\_dev(void) | |
| 描述 | 验证设备信息是否完整 | |
| 参数 | void |  |
| 返回值 | 0: 验证成功  -1: 验证失败 | |

### http\_auth组件及接口说明

http\_auth组件提供mqtt客户端登陆鉴权的功能。

Hike cloud V3.0基础物联网云平台的接入协议为MQTT，设备登录云平台，首先需要通过鉴权接口获取到用户名、密码、端口号、服务器地址等信息。http\_auth模块就是为MQTT提供鉴权服务的接口。

http\_auth组件的原理是，利用设备的三元组和时间戳等参数条件，利用HMAC MD5算法生成鉴权签名，打包为HTTP head，通过HTTP POST的方法，将数据发给鉴权服务器，鉴权服务器通过验证设备合法性后，将结果返回给设备。

下面是一组真实的打包后后head数据：

productCode:PklJRRPdjupp

deviceCode:adnZSv86QjOXAwDwvsyX

timestamp:1606458147024

method:HmacMD5

signature: dfcf8ba4fe3c2a709d923bff408dc0a6

其中signature 生成规则如下: $HmacMD5("productCodePklJRRPdjuppdeviceCodeadnZSv86QjOXAwDwvsyXtimestamp1606458147024methodHmacMD5",${deviceSecret})。

http\_auth组件工作流程图如下：



#### 接口数据结构定义

鉴权时传入的设备三码信息结构体：

struct authcate\_dev\_info

{

    char \*productCode;//产品码

    char \*deviceCode;//设备码

    char \*deviceSecret;//设备密钥

};

鉴权返回结构体：

struct authcate\_res

{

    /\*mqtt\*/

    char host[32];//服务器主机地址

    char user[32];//mqtt登录用户名

    char password[32]; //mqtt登录密码

    char protocol[32];//mqtt底层协议

    char port\_str[8];//端口号，字符串格式

    char uri[64]; //地址路径

    unsigned int port\_int; //端口号，整型

};

#### mqtt客户端登录鉴权接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | int mqtt\_login\_auth(const struct authcate\_dev\_info \*authcate\_dev,    struct authcate\_res \*res) | |
| 描述 | mqtt客户端登录鉴权接口 | |
| 参数 | \*authcate\_dev | 参数类型：输入  鉴权时传入的设备三码信息的变量地址 |
| \*res | 参数类型: 输出  鉴权成功后返回的鉴权结果，写入到此结构体变量 |
| 返回值 | 0: 鉴权成功  非0: 鉴权失败 | |

### hk\_mqtt\_client组件及接口说明

hk\_mqtt\_client组件的主要功能，是为设备接入3.0云平台，实现了mqtt客户端的封装，对外仅暴漏数据点接口，屏蔽了对接云平台繁琐复杂的过程。调用者仅需对指定的参数数据结构进行填充，调用专门的接口即可完成数据的publish；通过注册关心的下发数据点回调事件，即可完成远程数据的接收功能。

3.0云平台接入前需要在平台端定义产品的数据点物模型，物模型一旦确定，就可确定设备端数据点结构体的定义。不同的产品功能不一样，数据点也不一样，为了确保模块代码最大复用率，将数据点结构体分离，单独放在port\_sh.h头文件中。后续不同产品，只需要修改port\_sh.h文件中的数据点结构，配合cson模块提供的数据类型反射接口，完成数据点的反射即可实现模块的复用。

hk\_mqtt\_client组件依赖于SDK的mqtt组件，功能之间耦合性非常低。下面简单举例预设2接口内部运行过程。

hk\_mqtt\_client\_start接口运行流程如下：



hk\_mqtt\_client\_stop接口运行流程如下：



#### 接口数据结构定义

数据点结构体定义：

struct property\_data\_t

{

    struct pm25\_t pm25;

    struct temperature\_t temperature;

    struct humidity\_t humidity;

    struct power\_t power;

    struct loop\_t loop;

    struct mode\_t mode;

    struct fan\_speed\_t fan\_speed;

};

mqtt客户端属性设置接收事件定义：

enum

{

    HK\_MQTT\_RECV\_POWER\_CTRL\_EVENT,//开关事件

    HK\_MQTT\_RECV\_LOOP\_CTRL\_EVENT,//循环事件

    HK\_MQTT\_RECV\_MODE\_CTRL\_EVENT,//模式事件

     HK\_MQTT\_RECV\_FAN\_SPEED\_CTRL\_EVENT,// 风速调节事件

    HK\_RT\_DATA\_NOTIFY\_EVENT,//实时数据上报通知服务

};

#### 创建初始化并开启mqtt客户端接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | void hk\_mqtt\_client\_start(void) | |
| 描述 | 初始化并开启mqtt客户端；  此接口内部会判断批次码和6码是否存在，不存在则阻塞；  接口调用后，如果连接了工厂wifi，会开启批次码烧入任务，超时时间为60s，用来上电烧入或者变更批次码；  此接口需要上电时调用一次；建议此接口放在主干创建其他任务之前；目的是防止没有批次码的设备流出产线； | |
| 参数 | void |  |
| 返回值 | void | |

#### 停止并销毁mqtt客户端接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | void hk\_mqtt\_client\_stop(void) | |
| 描述 | 停止并销毁mqtt客户端，当配网时，可调用此函数，达到释放内存的目的。 | |
| 参数 | void |  |
| 返回值 | void | |

#### 数据点publish接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | int hk\_mqtt\_pub\_property\_data(const struct property\_data\_t \*property\_data) | |
| 描述 | mqtt客户端发布设备的属性状态到服务器。 | |
| 参数 | \*property\_data | 参数类型：输入  设备属性结构体变量地址； |
| 返回值 | 0: 发布成功  -1: 发布失败 | |

#### mqtt客户端接收事件注册回调函数接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | int hk\_mqtt\_recv\_cloud\_data\_event\_register(int32\_t event\_id, void (\*event\_handler)(void \*event\_data)) | |
| 描述 | mqtt客户端接收事件注册回调函数，当接收到服务器下发的数据点时，运行注册的回调函数。 | |
| 参数 | event\_id | 参数类型：输入  事件类型取值：  HK\_MQTT\_RECV\_POWER\_CTRL\_EVENT,//开关事件  HK\_MQTT\_RECV\_LOOP\_CTRL\_EVENT,//循环事件  HK\_MQTT\_RECV\_MODE\_CTRL\_EVENT,//模式事件  HK\_MQTT\_RECV\_FAN\_SPEED\_CTRL\_EVENT,//风速调节事件  HK\_RT\_DATA\_NOTIFY\_EVENT,//实时数据上报通知服务  HK\_MQTT\_THRESHOLD\_SET\_EVENT,//风速阈值设置事件 |
| event\_handler | 参数类型：输入  事件对应的回调函数 |
| 返回值 | 0: 注册成功  -1: 注册失败 | |

### OTA组件及接口说明

OTA组件，利用SDK提供的接口，进行了二次封装。上层应用只需要传入一个很简单的结构体，就能实现固件的下载，验证，固件切换等一系列功能。

本模块为mqtt客户端内部调用，应用层无需关心。

#### 接口数据结构定义

数据点结构体定义：

struct hk\_ota\_t

{

    char fw\_url[256];//固件下载url地址

    char fw\_md5[34];//固件MD5

    char fw\_ver[8];//固件版本号

};

#### hk\_ota\_start接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | void hk\_ota\_start(struct hk\_ota\_t \*hk\_ota) | |
| 描述 | 启动OTA。 | |
| 参数 | \*hk\_ota | 参数类型：输入  OTA固件远程信息结构体变量 |
| 返回值 | void | |

### wifi\_connect组件及接口说明

wifi\_connect组件提供了设备连接工厂wifi、设备soft ap配网、设备断电网络记忆以及断线重连等功能。

本组件规划有2个接口，遵循调用者简单易用，内部高度聚合封装，外部低耦合的设计原则。

wifi扫描列表、筛选工厂wifi、连接上次成功的ssid等功能，被划分到了hk\_wifi\_sta\_init

接口；wifi配网被划分到了hk\_wifi\_distribute\_start接口。

hk\_wifi\_sta\_init接口运行流程图如下：



hk\_wifi\_distribute\_start接口，实现了配网的功能。

设备在工厂生产阶段，会进行6码的下发操作，其中包含了配网时用到的热点的ssid，password和网关等字段。

设备配网的原理是，当需要配网时，设备会临时建立soft ap热点，热点的信息来源于6码中的3个。微信小程序从服务器端查询此设备的配网热点信息后，连接此热点后，通过udp协议将要配置的ap信息发送到设备，设备解析后尝试连接此ap。

设备端接口运行流程图如下：



#### hk\_wifi\_sta\_init接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | void hk\_wifi\_sta\_init(void) | |
| 描述 | 初始化wifi station 模式，扫描ap列表，如果发现工厂ap存在，优先连接工厂ap；  否则连接最后一次连接成功的ap；  此接口需要上电时调用一次。 | |
| 参数 | void |  |
| 返回值 | void | |

#### hk\_wifi\_distribute\_start接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | int hk\_wifi\_distribute\_start(long block\_time\_ms) | |
| 描述 | wifi配置网络，配网成功前，超时时间内，调用此接口的任务将会被阻塞；  配网超时或者配网失败，将会返回；  需要配网的场景，直接调用此接口。 | |
| 参数 | block\_time\_ms | 参数类型：输入  超时时间 |
| 返回值 | 0: 配网成功   -1: 配网失败 | |

### app store模块及接口说明

app store模块，是运行在业务层的数据存储与读取的模块。它负责保存业务层各模块需要保存或者需要读取的数据。它提供一系列接口，方便各模块直接存取数据，而无需关心底层存取的细节。

#### hk\_app\_store\_init接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | int hk\_app\_store\_init(void) | |
| 描述 | 初始化app\_store任务 | |
| 参数 | void |  |
| 返回值 | 0: 初始化成功   -1: 初始化失败 | |

#### hk\_app\_store\_get\_data接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | int hk\_app\_store\_get\_data(const char \*key, void \*data\_structure) | |
| 描述 | 业务层根据键值读取数据的接口 | |
| 参数 | \*key | 参数类型：输入  读取数据的键值地址 |
| \*data\_structure | 参数类型: 输出  读取的结果回写地址 |
| 返回值 | 0: 读取成功  非0: 读取失败 | |

#### hk\_app\_store\_set\_data接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | int hk\_app\_store\_set\_data(const char \*key,  void \*data\_structure,  int size) | |
| 描述 | 业务层根据键值保存数据的接口 | |
| 参数 | \*key | 参数类型：输入  读取数据的键值地址 |
| \*data\_structure | 参数类型: 输入  要保存的数据的地址 |
| size | 要保存的数据的字节长度 |
| 返回值 | 0: 保存成功  非0: 保存失败 | |

### app sensor模块及接口说明

app sensor为业务层运行的模块，它主要提供了设备所有传感器一次性的读取的接口。它与组件层的senor读取最大的区别是，app sensor模块的数据经过了系数调整后的数值。

为方便数据统一分发，结合生产者消费者模式，利用message\_dispatch模块，可以做一个数据生产任务，在任务中将传感器数据分发给那些订阅了传感器主题的消费者任务。具体流程如下：



#### 数据结构定义

系数相关：

struct coefficient\_t

{

     int pm25\_coff;//pm2.5系数

     int tempr\_coff;//温度系数

}

传感器组合值

struct sensor\_t

{

    int pm25\_value;//pm2.5

    float tempr\_value;//温度

    int humi\_value;//湿度

}

#### hk\_app\_sensor\_init接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | int hk\_app\_sensor\_init(void) | |
| 描述 | 初始化app\_sensor,创建传感器读取任务 | |
| 参数 | void |  |
| 返回值 | 0: 初始化成功   -1: 初始化失败 | |

#### hk\_app\_sensor\_get接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | int hk\_app\_sensor\_get(struct sensor\_t \*sensor) | |
| 描述 | 获取修正后传感器的值 | |
| 参数 | \*sensor | 参数类型：输出  传感器数据回写地址 |
| 返回值 | 0: 读取成功   -1: 读取失败 | |

### app modbus rtu slave模块及接口说明

app modbus rtu slave模块,是modbus从机的业务层部分。其主要承当着业务层数据的modbus接口分发及收集工作。

app modbus rtu slave模块，依赖message\_dispatch模块的数据订阅与发布功能，与其他业务层模块进行数据交互。

模块运行流程参考：



#### 数据类型定义

设备配置参数数据结构：

struct dev\_para\_t

{

    uint8\_t default\_product\_type;//默认设备类型

    uint8\_t pm2\_5\_coefficient;//系数

    uint8\_t tempr\_coefficient;

    uint32\_t default\_filter\_val;

    uint32\_t default\_fan\_speed\_lv;

    uint8\_t default\_power\_sta;

    uint8\_t default\_mode;

    uint8\_t default\_loop;

    uint32\_t default\_wifi\_enable;

    uint8\_t default\_exfan\_lv1\_duty;//排风机占空比系数

    uint8\_t default\_exfan\_lv2\_duty;//排风机占空比系数

    uint8\_t default\_exfan\_lv3\_duty;//排风机占空比系数

    uint8\_t default\_exfan\_lv4\_duty;//排风机占空比系数

    uint8\_t default\_exfan\_lv5\_duty;//排风机占空比系数

    uint8\_t default\_infan\_lv1\_duty;//进风机占空比系数

    uint8\_t default\_infan\_lv2\_duty;//进风机占空比系数

    uint8\_t default\_infan\_lv3\_duty;//进风机占空比系数

    uint8\_t default\_infan\_lv4\_duty;//进风机占空比系数

    uint8\_t default\_infan\_lv5\_duty;//进风机占空比系数

};

#### hk\_app\_modbus\_mdp\_rcv接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | void hk\_app\_app\_modbus\_mdp\_get (message\_dispatch\_parameter\_t \* para  ) | |
| 描述 | 获取message dispatch分发过来的数据,此接口用来创建分发数据描述符 | |
| 参数 | \* para | 参数类型：输出  接收的分发的数据参数地址 |
| 返回值 | void | |

#### hk\_app\_app\_modbus\_dev\_para\_dispatch接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | int hk\_app\_app\_modbus\_dev\_para\_dispatch (struct dev\_para\_t \*dev\_para) | |
| 描述 | 设备配置参数分发 | |
| 参数 | \* dev\_para | 参数类型：输入  设备配置参数地址 |
| 返回值 | 0：成功  -1：失败 | |

### app HMI模块及接口说明

app HMI模块，主要功能是人机交换逻辑的实现，包括按键事件注册、设备界面显示逻辑、用户菜单交互逻辑、与其他模块之间的数据分发与数据订阅等。

app HMI模块显示部分下一层依赖于GN1621段码屏驱动组件。由于段码显示屏高度定制的原因，无法实现显示模块像驱动组件那样标准化，所以将显示部分的实现放到了业务逻辑层。

出于便于快速移植的目的，将GN1621的显存进行重新映射，在业务层实现规律化、可循环化、抽象化管理。根据段码屏特点，将显存统一管理后，只要低频率定时刷新即可。

模块将显示和触摸抽象成HMI对象，可以将触摸抽象为HMI对象的输入事件，将显示抽象为HMI对象的输出事件。HMI对象内部可以根据输入输出事件，调用相应的处理方法函数。

模块对外数据同步和通信，依赖message\_dispatch模块。

HMI模块实现流程图如下：



#### 数据结构定义

面板属性数据结构

struct hmi\_property\_t

{

    uint8\_t timming\_val;//倒计时关机

    bool child\_lock\_sta;//童锁状态

    uint32\_t filter\_remain;//剩余滤网

    uint8\_t loop;//新风净化模式

    uint8\_t mode;//手动自动模式

    uint8\_t wifi\_sta;//wifi联网状态

    int8\_t tempr;//温度

    uint8\_t humi;//湿度

    uint16\_t pm2\_5;//pm2.5

    uint8\_t sys\_power;//系统开关

    uint8\_t fan\_speed\_lv;//风速挡位

uint8\_t work\_mode;//工程/正常 模式

uint8\_t user\_set\_step;//用户设置的步骤

    uint8\_t super\_set\_step;//工程设置的步骤

    uint8\_t tempr\_coefficient;//温度系数

    uint8\_t PM2\_5\_coefficient;//PM2.5系数

    uint8\_t rs485\_id;//485地址

    uint8\_t exfan\_lv1\_duty;//排风机占空比系数

    uint8\_t exfan\_lv2\_duty;//排风机占空比系数

    uint8\_t exfan\_lv3\_duty;//排风机占空比系数

    uint8\_t exfan\_lv4\_duty;//排风机占空比系数

    uint8\_t exfan\_lv5\_duty;//排风机占空比系数

    uint8\_t infan\_lv1\_duty;//进风机占空比系数

    uint8\_t infan\_lv2\_duty;//进风机占空比系数

    uint8\_t infan\_lv3\_duty;//进风机占空比系数

    uint8\_t infan\_lv4\_duty;//进风机占空比系数

    uint8\_t infan\_lv5\_duty;//进风机占空比系数

};

#### LCD刷新接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | void hk\_lcd\_flush(struct hmi\_property\_t \*hmi\_p) | |
| 描述 | 根据struct hmi\_property\_t的内容刷新显存，更新显示内容。 | |
| 参数 | \*hmi\_p | 参数类型：输入  lcd属性数据的地址 |
| 返回值 | void | |

#### LCD填充显存接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | void hk\_lcd\_disp\_ram\_fill(void) | |
| 描述 | 以1填满显存，达到全显的功能。 | |
| 参数 | void |  |
| 返回值 | void | |

#### LCD清空显存接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | void hk\_lcd\_disp\_ram\_clear(void) | |
| 描述 | 以0填满显存，达到清屏的功能。 | |
| 参数 | void |  |
| 返回值 | void | |

#### LCD设置某bit显存的接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | uint8\_t hk\_lcd\_disp\_ram\_set\_bit(uint8\_t pos, uint8\_t new\_state) | |
| 描述 | 设置设置某bit显存。 | |
| 参数 | pos | 参数类型：输入  显存的绝对位地址 |
| new\_state | 参数类型：输入  新的状态 |
| 返回值 | 0：成功  -1：失败 | |

#### LCD设置数字显示接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | uint8\_t hk\_lcd\_disp\_ram\_set\_bit( uint8\_t \*addr,  uint8\_t quantity,  uint8\_t mask,  int16\_t value ) | |
| 描述 | 设置数码显示接口。 | |
| 参数 | \*addr | 参数类型：输入  remap后的显示地址 |
| quantity | 参数类型：输入  数码的位数 |
|  | mask | 参数类型：输入  显示掩码，bit0代表的是最左边的数字的掩码状态，0不显示，1显示 |
|  | value | 参数类型：输入  显示数值 |
| 返回值 | 0：成功  -1：失败 | |

#### LCD设置某bit显存的接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | void hk\_hmi\_process(struct hmi\_property\_t \*hmi\_p,  uint8\_t button\_mask) | |
| 描述 | 人机交互处理函数。 | |
| 参数 | \*hmi\_p | 参数类型：输入  lcd属性数据的地址 |
| button\_mask | 参数类型：输入  按键掩码 |
| 返回值 | void | |

### app MQTT client模块及接口说明

app MQTT client模块，完成了本地数据、状态与云端同步的功能。

模块对外数据同步和通信，依赖message\_dispatch模块。

模块运行流程图如下：



#### 数据结构定义

PM2.5风速阈值定义，分发消息时使用。

struct dp\_pm25\_threshold\_t

{

    uint16\_t level\_1;

    uint16\_t level\_2;

    uint16\_t level\_3;

    uint16\_t level\_4;

    uint16\_t level\_5;

}

数据点结构体定义，分发消息或者接收别的模块的消息使用。

struct dp\_property\_data\_t

{

    uint16\_t pm25;

    float temperature;

    uint8\_t humidity;

    uint8\_t power;

    uint8\_t loop;

    uint8\_t mode;

    uint8\_t fan\_speed;

};

#### 云端下发控制设备事件处理函数接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | void hk\_mqtt\_recv\_event\_handler(int32\_t event\_id,  void \*event\_data,  uint8\_t event\_data\_len,  void \*arg) | |
| 描述 | 云端下发控制设备属性事件的注册的回调函数，在此函数中，根据事件的类型，解析出控制数据。 | |
| 参数 | event\_id | 参数类型：输出  事件ID |
| event\_data | 参数类型：输出  事件的数据。 |
| event\_data\_len | 参数类型：输出  事件数据的长度 |
| \*arg | 参数类型：输入  可选参数，不使用时传NULL |
| 返回值 | void | |

### app embedded log模块及接口说明

app embedded log模块，主要功能是提供log输出功能。

模块采用灵活的注册机制，通过注册，实现log输出的底层实现。使用本模块的功能，必须先定义一个处理log输出的底层函数，可以是串口输出，也可以是文件输出。

模块运行流程图如下：



#### 数据结构定义

日志等级枚举定义：

typedef enum {

  ULOG\_TRACE\_LEVEL=100,

  ULOG\_DEBUG\_LEVEL,

  ULOG\_INFO\_LEVEL,

  ULOG\_WARNING\_LEVEL,

  ULOG\_ERROR\_LEVEL,

  ULOG\_CRITICAL\_LEVEL,

  ULOG\_ALWAYS\_LEVEL

} ulog\_level\_t;

日志处理函数订阅错误种类枚举定义：

typedef enum {

  ULOG\_ERR\_NONE = 0,

  ULOG\_ERR\_SUBSCRIBERS\_EXCEEDED,

  ULOG\_ERR\_NOT\_SUBSCRIBED,

} ulog\_err\_t;

#### ulog\_init接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | void ulog\_init(void) | |
| 描述 | 初始化ulog框架 ，调用log接口之前，必须先调用此接口完成框架初始化。 | |
| 参数 | void |  |
| 返回值 | Void | |

#### ulog\_subscribe接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | ulog\_err\_t ulog\_subscribe(ulog\_function\_t fn,  ulog\_level\_t threshold) | |
| 描述 | 订阅注册日志消息处理函数，在调用ulog\_init ()之后，需要调用此接口完成在框架中。 | |
| 参数 | fn | 参数类型：输入  日志处理函数的地址 |
|  | threshold | 参数类型：输入  日志处理函数的等级，取值参考3.11.1小节。函数可根据不同的等级，做出不同的响应，比如颜色控制等。 |
| 返回值 | ULOG\_ERR\_NONE：注册成功；  ULOG\_ERR\_SUBSCRIBERS\_EXCEEDED：注册函数的个数超过限制大小 | |

#### ulog\_unsubscribe接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | ulog\_err\_t ulog\_unsubscribe(ulog\_function\_t fn) | |
| 描述 | 取消订阅注册日志消息处理函数；当不再需要某一个日志处理函数，调用此接口取消订阅注册。 | |
| 参数 | fn | 参数类型：输入  日志处理函数的地址 |
| 返回值 | ULOG\_ERR\_NONE：取消注册成功；  ULOG\_ERR\_NOT\_SUBSCRIBED：函数未被注册过 | |

#### ulog\_message接口

|  |  |  |
| --- | --- | --- |
| 函数原型 | void ulog\_message(ulog\_level\_t severity, const char \*fmt, ...) | |
| 描述 | 日志消息处理，当需要产生log处理，调用此接口 | |
| 参数 | severity | 参数类型：输入  日志等级，取值参考3.11.1 |
| \*fmt | 参数类型：输入  日志格式化表达式 |
| … | 不定参数候补 |
| 返回值 | void | |

#### 接口补充说明

为了便于维护、升级和拓展，此模块所有的接口统一用宏作转接。在基础接口之上另外拓展了几个高级别的接口。宏定义如下：

#define ULOG\_INIT() ulog\_init()

  #define ULOG\_SUBSCRIBE(a, b) ulog\_subscribe(a, b)

  #define ULOG\_UNSUBSCRIBE(a) ulog\_unsubscribe(a)

  #define ulog\_level\_name(a) ulog\_level\_name(a)

  #define ULOG(...) ulog\_message(\_\_VA\_ARGS\_\_)

  #define ULOG\_TRACE(...) ulog\_message(ULOG\_TRACE\_LEVEL, \_\_VA\_ARGS\_\_)

  #define ULOG\_DEBUG(...) ulog\_message(ULOG\_DEBUG\_LEVEL, \_\_VA\_ARGS\_\_)

  #define ULOG\_INFO(...) ulog\_message(ULOG\_INFO\_LEVEL, \_\_VA\_ARGS\_\_)

  #define ULOG\_WARNING(...) ulog\_message(ULOG\_WARNING\_LEVEL, \_\_VA\_ARGS\_\_)

  #define ULOG\_ERROR(...) ulog\_message(ULOG\_ERROR\_LEVEL, \_\_VA\_ARGS\_\_)

  #define ULOG\_CRITICAL(...) ulog\_message(ULOG\_CRITICAL\_LEVEL, \_\_VA\_ARGS\_\_)

  #define ULOG\_ALWAYS(...) ulog\_message(ULOG\_ALWAYS\_LEVEL, \_\_VA\_ARGS\_\_)

另外，为了便于管理log的开关，增加了控制宏#define ULOG\_ENABLED，具体如下：

#ifdef ULOG\_ENABLED

  #define ULOG\_INIT() ulog\_init()

  #define ULOG\_SUBSCRIBE(a, b) ulog\_subscribe(a, b)

  #define ULOG\_UNSUBSCRIBE(a) ulog\_unsubscribe(a)

  #define ulog\_level\_name(a) ulog\_level\_name(a)

  #define ULOG(...) ulog\_message(\_\_VA\_ARGS\_\_)

  #define ULOG\_TRACE(...) ulog\_message(ULOG\_TRACE\_LEVEL, \_\_VA\_ARGS\_\_)

  #define ULOG\_DEBUG(...) ulog\_message(ULOG\_DEBUG\_LEVEL, \_\_VA\_ARGS\_\_)

  #define ULOG\_INFO(...) ulog\_message(ULOG\_INFO\_LEVEL, \_\_VA\_ARGS\_\_)

  #define ULOG\_WARNING(...) ulog\_message(ULOG\_WARNING\_LEVEL, \_\_VA\_ARGS\_\_)

  #define ULOG\_ERROR(...) ulog\_message(ULOG\_ERROR\_LEVEL, \_\_VA\_ARGS\_\_)

  #define ULOG\_CRITICAL(...) ulog\_message(ULOG\_CRITICAL\_LEVEL, \_\_VA\_ARGS\_\_)

  #define ULOG\_ALWAYS(...) ulog\_message(ULOG\_ALWAYS\_LEVEL, \_\_VA\_ARGS\_\_)

#else

  // uLog vanishes when disabled at compile time...

  #define ULOG\_INIT() do {} while(0)

  #define ULOG\_SUBSCRIBE(a, b) do {} while(0)

  #define ULOG\_UNSUBSCRIBE(a) do {} while(0)

  #define ulog\_level\_name(a) do {} while(0)

  #define ULOG(s, f, ...) do {} while(0)

  #define ULOG\_TRACE(f, ...) do {} while(0)

  #define ULOG\_DEBUG(f, ...) do {} while(0))

  #define ULOG\_INFO(f, ...) do {} while(0)

  #define ULOG\_WARNING(f, ...) do {} while(0)

  #define ULOG\_ERROR(f, ...) do {} while(0)

  #define ULOG\_CRITICAL(f, ...) do {} while(0)

  #define ULOG\_ALWAYS(f, ...) do {} while(0)

#endif

这样，当不再使用log接口时，只需要注释掉#define ULOG\_ENABLED，调用此接口的上层无需做任何更改，而且编译后不占用任何资源。