# 涂鸦ZigBee SDK 开发使用手册

版本记录：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本 | 编写/修订说明 | 修订人 | 修订日期 | 备注 |
| 1.0.0 | 创建文档 | 邓亮 | 2019.01.08 | 初稿 |
| 1.0.1 | 修订文档 | 祝盼飞 | 2019.04.03 | 添加应用程序开发章节 |
| 1.0.2 | 修订文档 | 张鹏 | 2019.04.06 |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[涂鸦ZigBee SDK 开发使用手册 1](#_Toc5462601)

[概述 3](#_Toc5462602)

[1. SDK框架 3](#_Toc5462603)

[1.1 目录结构介绍 3](#_Toc5462604)

[2. HAL开发接口 4](#_Toc5462605)

[2.1 IO输入输出 4](#_Toc5462608)

[2.2 串口 6](#_Toc5462609)

[2.3 硬件定时器 7](#_Toc5462610)

[3. 协议开发接口 8](#_Toc5462611)

[3.1 设备注册 8](#_Toc5462613)

[3.2 网络管理 8](#_Toc5462614)

[3.2.1 设备入网 8](#_Toc5462624)

[3.2.2 设备离网 8](#_Toc5462625)

[3.2.3 心跳管理 8](#_Toc5462626)

[3.2.4 网络状态 9](#_Toc5462627)

[3.3 场景管理 9](#_Toc5462628)

[4. 应用开发接口 10](#_Toc5462629)

[4.1 事件管理 10](#_Toc5462633)

[4.2 网络状态回调 10](#_Toc5462634)

[4.3 接收zigbee数据 10](#_Toc5462635)

[4.4 发送zigbee数据 10](#_Toc5462636)

[5. 应用层序开发 11](#_Toc5462637)

[5.1 注册对应函数 11](#_Toc5462641)

[5.2 配置固件信息 11](#_Toc5462642)

[5.3 应用开发实例 12](#_Toc5462643)

[5.3.1 APP框架介绍 12](#_Toc5462649)

[5.3.2 新建工程 13](#_Toc5462650)

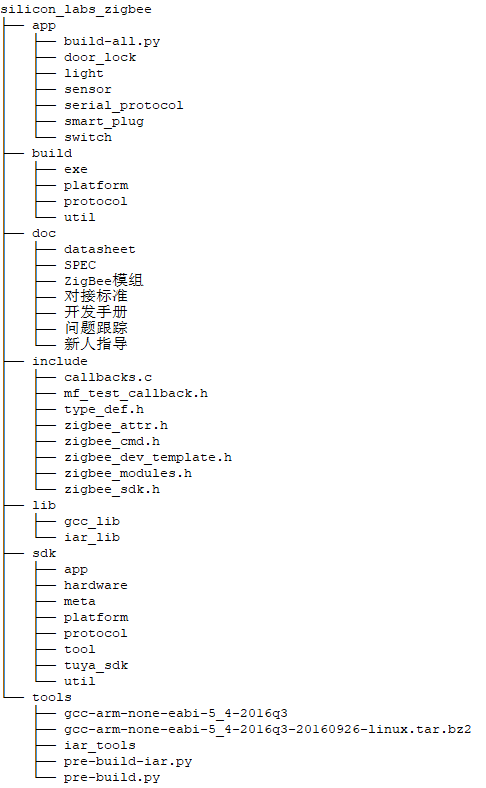
[5.3.3 编译 14](#_Toc5462651)

## 概述

涂鸦ZigBee SDK开发使用手册文档主要针对涂鸦ZigBee 3.0相关产品开发使用，该SDK基于Silicon Labs的ZigBee3.0标准协议SDK的基础上，对相关业务进行了抽象和二次封装。主要内容包含HAL接口封装、ZigBee协议相关功能以及应用开发三个部分。

# SDK框架

## 目录结构介绍



* app： 应用工程路径，用户开发应用，在该目录下面对应的产品品类里开发；
* doc： ZigBee相关文档资料；
* include：sdk对外的头文件和模板文件，包含sdk封装的接口的定义等；
* lib： sdk编译生成的库文件，分为iar库和gcc库， 不同编译器下又分为release 和debug库；
* sdk： zigbee的sdk源码路径， 其中tuya\_sdk为抽象和封装的中间层；
* tools： 编译使用到的相关工具和脚本；

# HAL开发接口



## IO输入输出

通用GPIO port口定义；

typedef enum {

PORT\_A = 0x00,

PORT\_B,

PORT\_C,

PORT\_D,

PORT\_E,

PORT\_F,

PORT\_H,

PORT\_I,

PORT\_J,

PORT\_K

}GPIO\_PORT\_T;

通用GPIO pin口定义；

typedef enum {

PIN\_0 = 0x00,

PIN\_1,

PIN\_2,

PIN\_3,

PIN\_4,

PIN\_5,

PIN\_6,

PIN\_7,

PIN\_8,

PIN\_9,

PIN\_10,

PIN\_11,

PIN\_12,

PIN\_13,

PIN\_14,

PIN\_15,

}GPIO\_PIN\_T;

通用GPIO输入输出模式定义；

//GPIO\_Mode\_TypeDef

typedef enum {

GPIO\_MODE\_INPUT\_HIGH\_IMPEDANCE = 0,

GPIO\_MODE\_INPUT\_PULL,

GPIO\_MODE\_OUTPUT\_PP,

GPIO\_MODE\_OUTPUT\_OD,

GPIO\_MODE\_OUTPUT\_OD\_PULL\_UP,

GPIO\_MODE\_OUTPUT\_OD\_PULL\_DOWN,

}GPIO\_MODE\_T;

通用GPIO输出高低电平模式定义；

typedef enum {

GPIO\_DOUT\_LOW = 0,

GPIO\_DOUT\_HIGH = 1,

}GPIO\_DOUT\_T;

GPIO配置；

typedef struct {

GPIO\_PORT\_T port;//

GPIO\_PIN\_T pin;//

GPIO\_MODE\_T mode;// 输入输出模式

GPIO\_DOUT\_T out;// 输出电平

GPIO\_LEVEL\_T drive\_flag; //什么电平有效

} gpio\_config\_t;

void gpio\_output\_init(gpio\_config\_t \*config, uint8\_t sum)；

描述：GPIO口输出初始化；

config：IO口配置信息；

sum： IO配置信息组数；

void gpio\_button\_init(gpio\_config\_t \*config, uint8\_t sum, uint32\_t jitter\_time, key\_func\_t key\_func)；

描述：GPIO口输入初始化；

config： IO口配置信息；

sum： IO配置信息组数；

jitter\_time： 按键防反跳时间；

key\_func： 按键中断回调函数；

## 串口

void user\_uart\_init(user\_uart\_config\_t \*config); //出口初始化，包含硬件信息和接收回调函数

void user\_uart\_send(UART\_ID\_T uart\_id, uint8\_t\* data, uint16\_t data\_len);//串口发送

void user\_uart\_disbale(UART\_ID\_T uart\_id);//串口禁用

typedef enum {

UART\_ID\_UART0 = 0,

UART\_ID\_UART1

}UART\_ID\_T; //串口ID

typedef enum {

USART\_PARITY\_NONE = 0, /\*\*< No parity. \*/

USART\_PARITY\_EVEN, /\*\*< Even parity. \*/

USART\_PARITY\_ODD, /\*\*< Odd parity. \*/

} USART\_PARITY\_T; //串口检验

typedef enum {

USART\_STOPBITS\_HALF = 0, /\*\*< 0.5 stop bits. \*/

USART\_STOPBITS\_ONE, /\*\*< 1 stop bits. \*/

USART\_STOPBITS\_ONEANDAHALF, /\*\*< 1.5 stop bits. \*/

USART\_STOPBITS\_TWO /\*\*< 2 stop bits. \*/

} USART\_STOPBITS\_T; //串口停止位

typedef enum {

USART\_DATABITS\_8BIT = 0, /\*\*< 8 bits databits. \*/

USART\_DATABITS\_9BIT, /\*\*< 9 bits databits. \*/

} USART\_DATABITS\_T; //串口数据位

typedef enum {

UART\_PIN\_TYPE\_DEFAULT = 0,

UART\_PIN\_TYPE\_CONFIG

}UART\_PIN\_TYPE\_T; //串口IO配置

typedef struct {

GPIO\_PORT\_T port;

GPIO\_PIN\_T pin;

}GPIO\_PORT\_PIN\_T; //通用IO描述

typedef void(\*uart\_callback)(uint8\_t \*data, uint16\_t len); //串口回调函数格式

typedef struct {

UART\_ID\_T uart\_id;

UART\_PIN\_TYPE\_T pin\_type;

GPIO\_PORT\_PIN\_T tx;

GPIO\_PORT\_PIN\_T rx;

uint32\_t baud\_rate; //波特率，直接填写数值

USART\_PARITY\_T parity;

USART\_STOPBITS\_T stop\_bits;

USART\_DATABITS\_T data\_bits;

uart\_callback func;

}user\_uart\_config\_t; //串口配置结构

## 硬件定时器

创建一个硬件定时器函数

TIMER\_ID\_T timer\_hardware\_start\_100us(

uint32\_t t, //延时时间，单位100us

TIMER\_RELOAD\_FLAG\_T flag, //是否周期性执行

hardware\_timer\_func\_t func //时间达到后调用的回调函数

)；

//取消定时器

void timer\_hardware\_stop\_100us(TIMER\_ID\_T id)；//提前停止并释放该定时器

数据结构描述：

typedef enum {

V\_TIMER0 = 0,

V\_TIMER1,

V\_TIMER2,

V\_TIMER3,

V\_TIMER\_ERR = 0xFF //代表定时器id错误

}TIMER\_ID\_T;

typedef enum {

HARDWARE\_TIMER\_AUTO\_RELOAD\_ENABLE = 0, //周期执行标志

HARDWARE\_TIMER\_AUTO\_RELOAD\_DISABLE //一次性执行标志

}TIMER\_RELOAD\_FLAG\_T;

typedef void (\*hardware\_timer\_func\_t)(TIMER\_ID\_T); //定时器回调函数，参数为定时器id

获取当前系统时钟ticks（ms）；

uint32\_t dev\_current\_millisecond\_ticks\_get(void)；

# 协议开发接口



## 设备注册

void dev\_register\_zg\_ep\_infor (

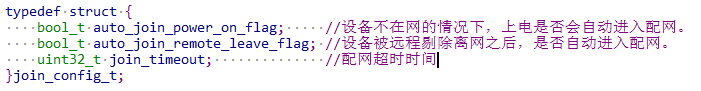
dev\_description\_t \*ep\_desc, //zigbee endpoint的设备描述结构体列表

uint8\_t ep\_sums //zigbee endpoint的个数

)；

void dev\_zg\_join\_config(join\_config\_t \*cfg)//设备为路由设备还是睡眠设备，上电是否需要启动配网，远程离网之后，是否需要

## 网络管理





### 设备入网

bool dev\_zigbee\_join\_start(uint32\_t join\_timeout)；

描述：设备开始入网；

join\_timeout：入网超时时间，单位为ms；

返回值： true或者false；

### 设备离网

EmberStatus dev\_zigbee\_leave(void)；

描述： 设备立即离网；

void dev\_zigbee\_delay\_leave(uint32\_t t)；

描述：设备延时一段时间之后离网；

t：延时离网时间吗，单位为ms；

### 心跳管理

bool dev\_heartbeat\_set(HEARTBEAT\_TYPE\_E type, uint16\_t duration)；

描述： 设置设备心跳类型和心跳周期；

type：0：applicationVersion， 1：batteryVoltage；

duration：心跳周期，单位为ms；

返回值： true或者false；

### 网络状态

typedef enum {

NET\_POWER\_ON\_LEAVE, //上电设备未组网

NET\_POWER\_ON\_ONLINE, //上电设备已组网

NET\_JOIN\_START, //开始组网

NET\_JOIN\_TIMEOUT, //组网超时

NET\_JOIN\_OK, //组网成功

NET\_LOST, //网络失联

NET\_REMOTE\_LEAVE, //远程删除设备

NET\_LOCAL\_LEAVE, //本地删除设备

}NET\_EVT\_T;

NET\_ST\_T nwk\_state\_get(void)；

描述： 设备当前网络状态获取；

返回值：设备当前的网络状态；

## 场景管理

原理：当增加情景时，保存用户自定义数据，调用情景时读取该数据给用户

网络收到增加场景命令时调用

void dev\_scene\_add\_callback(

uint8\_t endpoint, //需要增加场景的endpoint

uint8\_t \*out\_data, //用户自定义数据，比如该endpoint的设备状态

uint8\_t \*in\_out\_len //传入外层buffer的长度和传出out\_data的长度

)；

网络收到执行场景时调用

void dev\_scene\_recall\_callback(

uint8\_t endpoint, //需要执行场景的endpoint

const scene\_save\_data\_t \*in\_data //用户曾经增加情景时保存的数据

)；

添加场景之前是否移除endpoint下所有场景， 用于场景开关添加场景时使用，返回false：不移除所有， 返回true：移除所有； 默认返回false；

bool zigbee\_sdk\_scene\_remove\_before\_add(void)；

场景唤醒函数，当场景开关recall scene时使用

bool dev\_scene\_recall\_send\_command(uint16\_t endpoint, uint16\_t groupId, uint8\_t sceneId)；

# 应用开发接口



## 事件管理

typedef enum {

DEV\_EVT\_1 = 23,

DEV\_EVT\_2,

DEV\_EVT\_3,

DEV\_EVT\_4,

DEV\_EVT\_5,

DEV\_EVT\_6,

DEV\_EVT\_7,

DEV\_EVT\_8

}DEV\_EVT\_T; //用户事件id

typedef void (\*timer\_func\_t)(uint8\_t); //事件回调函数样式，参数是事件id

extern void dev\_timer\_stop(uint8\_t evt); //停止事件

extern void dev\_timer\_start\_with\_callback(uint8\_t evt, uint32\_t t, timer\_func\_t func); //启动事件并安装事件回调函数

extern void dev\_timer\_start(uint8\_t evt, uint32\_t t); //启动事件。如果已经安装回调函数，再起启动可以不用再次安装回调函数

## 网络状态回调

void nwk\_state\_changed\_callback(NET\_ST\_T state)；

描述：网络状态改变callback， 当设备的网络状态改变之后，会调用这个callback接口；

State： 改变之后的网络状态，类型为NET\_ST\_T；

## 接收zigbee数据

ZCL\_CMD\_RET\_T dev\_msg\_recv\_callback(dev\_msg\_t \*dev\_msg);

描述：zigbee收到数据后调用该函数，dev\_msg 含有zigbee发送的具体命令数据

## 发送zigbee数据

void dev\_zigbee\_send\_data(

dev\_send\_data\_t \*send\_data, //发送的数据

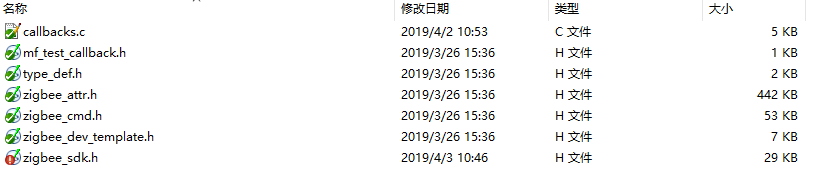
send\_result\_func\_t fun, //发送成功或者失败的回调函数

uint32\_t send\_timeout //发送数据的最大超时，单位毫秒

);

# 应用层序开发





callback.c 应用工程接口使用样例文件

mf\_test\_callback.h 产测相关命令定义

zigbee\_attr.h zigbee 标准属性相关定义

zigbee\_cmd.h zigbee 标准命令相关定义

zigbee\_dev\_template.h 常见zigbee设备定义模板

zigee\_sdk.h 涂鸦sdk提供的所有API接口

## 注册对应函数

#include "zigbee\_sdk.h"

dev\_power\_on\_init(); //上电初始化

dev\_system\_on\_init();//协议栈和系统基本组件启动后初始化

dev\_recovery\_factory();//恢复出厂设置用户回调

dev\_mf\_test\_callback();//生产测试用户回调

dev\_msg\_recv\_callback();//收数据回调

dev\_scene\_add\_callback();//增加情景回调

dev\_scene\_recall\_callback();//调用情景回调

nwk\_state\_changed\_callback();//网络状态改变回调

user\_evt1\_handle(); //用户自定义事件1处理函数

user\_evt2\_handle(); //用户自定义事件2处理函数

user\_evt3\_handle(); //用户自定义事件3处理函数

.....

## 配置固件信息

固件信息配置在对应工程下的package.json文件里进行配置，配置项如下所示；

package.json

{

"fimwareInfo": {

"name": "oem\_si32\_zg\_plug\_10A\_USB\_dlx", //固件名称

"description": "zigbee switch common", //固件描述

"version": "1.0.9", //软件版本号

"bv\_version": "1.0", //SDK版本号

"ic": "efr32mg13p732gm48", // zigbee 芯片型号

"ota\_image\_type":"0x1602", //OTA升级识别使用，目前固定

"manufacture\_id":"0x1002", //OTA升级识别使用，目前固定

"model\_id":"TS0105", //设备组网快速识别设备类型使用

"pid": "ZHahfZRP", //设备默认产品id，产品经理分配

"manufacture\_name": "TUYATEC-" //产品id前缀，网关识别用什么协议来交互

}

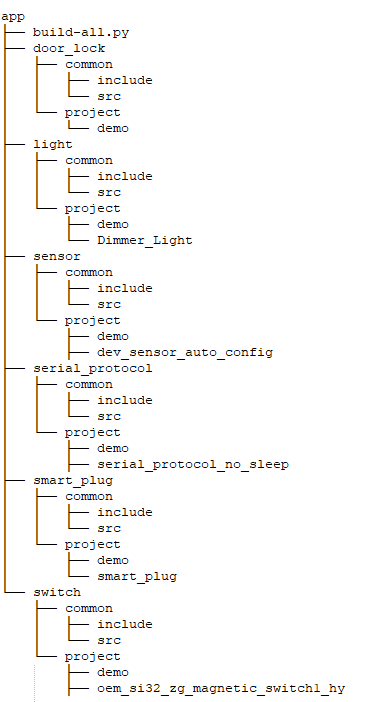
…  
}

## 应用开发实例



### APP框架介绍

App框架图如下所示;

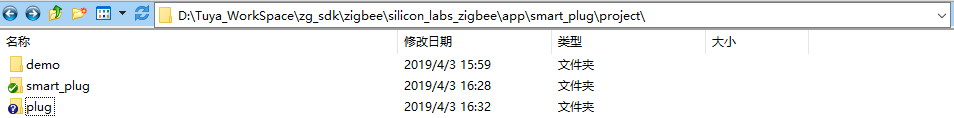


* build-all.py: 编译所有应用工程
* door\_lock: 门锁品类产品应用工程
* light ： 照明品类产品应用工程
* sensor: 传感器类产品应用工程
* serial\_protocol:串口对接类产品应用工程
* smart\_plug: 插座类产品应用工程
* switch: 开关类产品应用工程

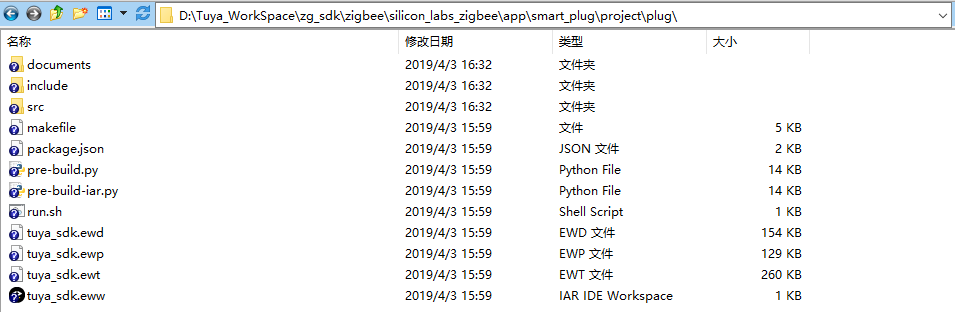
不同的品类文件下有common和project两个文件夹；common文件夹下包含include和src两个文件，这两个文件夹做为该品类公共资源池文件夹，分别可以存放公共的C文件和H文件；project文件下为对应的应用工程文件，同时project下有一个demod空工程，可用于新建工程使用。

### 新建工程

以创建一个插座工程为例，进入smart\_plug/project路径，复制project文件夹下的demo文件夹，并且重命名为plug。



进入plug文件夹，可以看到整个工程概况。



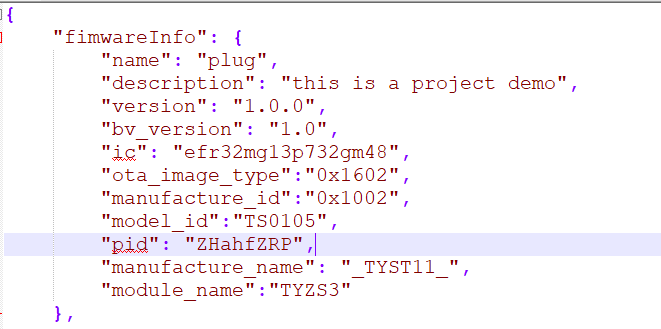
document： 工程相关的文档

include： 头文件

src： C文件

package.json： 设备的信息（PID，modeID，版本号、IO配置等设备信息都在该文件进行修改）

打开package.json文件，修改plug设备信息和IO配置信息；修改完之后，保存文件。



### 编译

**GCC下编译**

GCC下编译在工程文件下运行run.sh脚本即可， 具体命令行如下：

#./run.sh clean //make clean

#./run.sh build //make all with release

#./run.sh build 0 //make all with release

#./run.sh build 1 //make all with debug

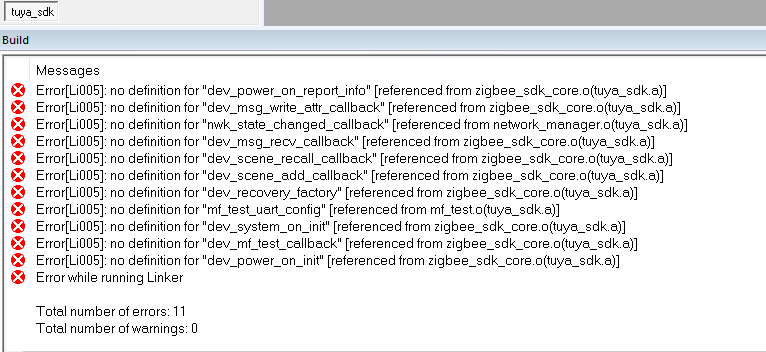
如果第一次编译报错的话需要注意下文件权限和系统PATH路径的配置， 具体修改方法如下：

文件权限：进入silicon\_lib\_zigbee主目录下运行chmod –R 777 \*

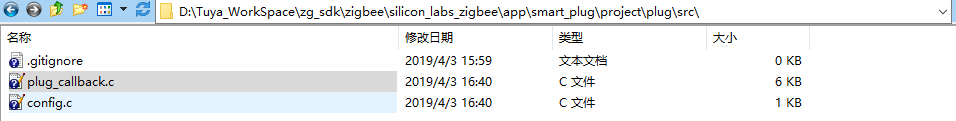
PATH修改：进入silicon\_lib\_zigbee主目录下运行 export PATH="/your\_git\_path/zigbee/silicon\_labs\_zigbee/tools/gcc-arm-none-eabi-5\_4-2016q3/bin:$PATH"

**IAR下编译**

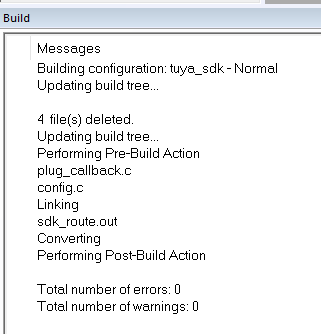
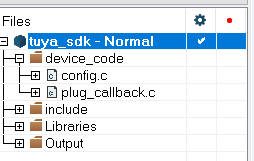
打开工程，全编译工程，第一次编译会报错，需要手动将src下的c文件添加到IAR工程里。



打开src文件夹，经过一次全编译之后，src文件夹会生成config.c和plug\_callback.c两个文件。



将plug\_callback.c文件添加到工程中。再次编译即可。



打开plug\_callback.c，开始你的开发吧。

插座由1个按键，1个led灯，1个配网灯和1个继电器控制口构成。

1. 输入管脚配置

按键为输入管脚，定义输入管脚

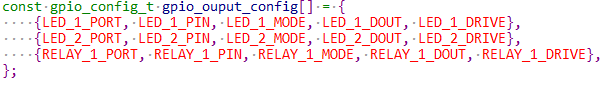
填充输入管脚表格



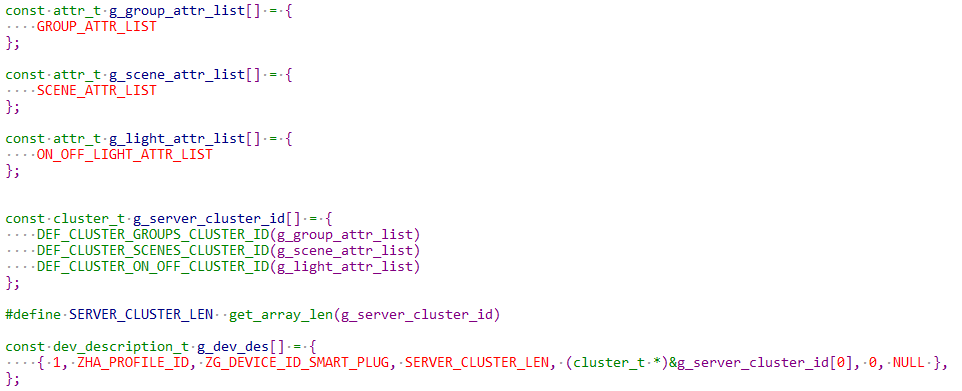
1. 输出管脚配置

Led控制，配网灯控制，继电器控制为输出管脚，定义输出管脚

填充输出管脚表格



硬件信息填充完成之后，我们需要对插座这个设备进行设备定义，需要配置设备类型（路由设备或者睡眠设备）和入网策略。设备定义配置，主要确定设备有几个endpoint，每个endpoint下有哪些cluster以及哪些属性等。入网策略配置，确定设备上电是否需要重新组网，远程离网之后，是否需要进行配网以及心跳周期等。



填充完注册信息之后，就可以在设备上电初始化函数中void dev\_power\_on\_init(void)，实现设备的注册，设备的入网策略，输入输出硬件IO的初始化操作的配置。



void dev\_power\_on\_init(void)函数，内部注册了一个按键处理函数static void \_\_dev\_key\_handle(uint32\_t key\_id, key\_st\_t key\_st, uint32\_t push\_time)，按键处理的业务都可以在该注册函数下面进行开发。