# 1. LiteOS OC MQTT 抽象组件

概述

为了适应各种各样的使用mqtt接入华为OC的模式，特采用该层次接口，对上提供应用所需的接口，对下允许接入方式的灵活适配。

OC MQTT AL的api接口声明在<oc\_mqtt\_al.h>中，使用相关的接口需要包含该头文件。

配置并连接

对接服务器的所有信息保存在结构体oc\_mqtt\_config\_t中，其定义在oc\_mqtt\_al.h中，如下：

typedef struct

{

en\_oc\_mqtt\_mode boot\_mode; //对接模式：直连模式和bs模式

uint8\_t lifetime; //保持连接时长

char \*server\_addr; //mqtt服务器地址，ip或者域名

char \*server\_port; //mqtt服务器端口

en\_mqtt\_al\_security\_t sec\_type; //当前仅支持crt模式

char \*id; //设备对接ID，默认使用NOTEID（设备标识码）对接

char \*pwd; //设备秘钥

//int device\_mode; //未使用

fn\_oc\_mqtt\_msg\_deal msg\_deal;//接收到数据的回调函数

void \*msg\_deal\_arg; //回调函数参数

}oc\_mqtt\_config\_t;

其中boot\_mode是对接模式，对应华为平台的两种模式：

typedef enum

{

en\_oc\_mqtt\_mode\_bs\_static\_nodeid\_hmacsha256\_notimecheck\_json =0,

en\_oc\_mqtt\_mode\_nobs\_static\_nodeid\_hmacsha256\_notimecheck\_json,

en\_oc\_mqtt\_mode\_last,

}en\_oc\_mqtt\_mode;

本实验中使用的是直连模式，选择第二种。

在配置结构体完成之后，调用配置函数进行配置并连接，API如下：

/\*\*

\* @brief the application use this function to configure the mqtt agent

\*

\* @param[in] param, refer to oc\_mqtt\_config\_t

\*

\* @return code: define by en\_oc\_mqtt\_err\_code while 0 means success

\*/

int oc\_mqtt\_config(oc\_mqtt\_config\_t \*param);

函数参数很清楚，将存放对接信息的结构体指针传入即可，API的返回值是由en\_oc\_mqtt\_err\_code定义的枚举类型，方便定位问题：

typedef enum

{

en\_oc\_mqtt\_err\_ok = 0, ///< this means the status ok

en\_oc\_mqtt\_err\_parafmt, ///< this means the parameter err format

en\_oc\_mqtt\_err\_network, ///< this means the network wrong status

en\_oc\_mqtt\_err\_conversion, ///< this means the mqtt version err

en\_oc\_mqtt\_err\_conclientid, ///< this means the client id is err

en\_oc\_mqtt\_err\_conserver, ///< this means the server refused the service for some reason(likely the id and pwd)

en\_oc\_mqtt\_err\_conuserpwd, ///< bad user name or passwd

en\_oc\_mqtt\_err\_conclient, ///< the client id /user/pwd is right, but does not allowed

en\_oc\_mqtt\_err\_subscribe, ///< this means subscribe the topic failed

en\_oc\_mqtt\_err\_publish, ///< this means subscribe the topic failed

en\_oc\_mqtt\_err\_configured, ///< this means we has configured, please deconfigured it and then do configure again

en\_oc\_mqtt\_err\_noconfigured, ///< this means we have not configure it yet,so could not connect

en\_oc\_mqtt\_err\_noconected, ///< this means the connection has not been built, so you could not send data

en\_oc\_mqtt\_err\_gethubaddrtimeout, ///< this means get the hub address timeout

en\_oc\_mqtt\_err\_sysmem, ///< this means the system memory is not enough

en\_oc\_mqtt\_err\_system, ///< this means that the system porting may have some problem,maybe not install yet

en\_oc\_mqtt\_err\_last,

}en\_oc\_mqtt\_err\_code\_t;

数据上报

连接成功之后，向华为云平台上报数据需要关注两部分：

发布消息主题：这个OC\_MQTT组件会自动帮我们搞定；

发布消息指令：发布时由用户指定；

发布消息内容：在使用JSON数据格式通信时，需要在本地将数据封装为JSON格式，OC\_MQTT组件中已经编写了一个助手程序，帮助我们封装和数据。

数据封装

上报华为云平台的数据格式如下：

{

"msgType": "deviceReq",

"data": [{

"serviceId": "Lightness",

"serviceData": {

"Lightness": 123

}

}]

}

这个数据格式是固定的，用户指定的只有serviceId和service\_data，所以OC\_MQTT提供了一个助手组件，使用时包含如下的头文件即可：

#include <oc\_mqtt\_assistant.h>

在该文件中，对于封装有效数据的结构体是：

typedef struct

{

char \*name; ///< key name

char \*buf; ///< used to storage the key value

int len; ///< how long the key value

en\_value\_type type; ///< the value type

}tag\_key\_value;

name：键名，比如Lightness；

buf：键值，比如123；

len：有效数据缓冲区长度；

type：键值类型；

键值类型已经枚举出了，支持如下三种类型，如下：

typedef enum

{

en\_key\_value\_type\_int = 0,

en\_key\_value\_type\_string,

en\_key\_value\_type\_array,

}en\_value\_type;

上述这个 tag\_key\_value 只是一条有效数据，对于需要同时上报多条数据的情况，助手程序也提供了一个链表，使用时只需要将next指针指向下一条数据即可。

typedef struct

{

void \*next;

tag\_key\_value item;

}tag\_key\_value\_list;

将数据存放在链表及结构体中之后，完成了初步封装，需要再加上上报华为云的信息，整体存放在下面的结构体中：

typedef struct

{

char \*serviceid;

tag\_key\_value\_list \*paralst; //之前封装的有效数据链表

char \*eventtime;

en\_oc\_mqtt\_has\_more hasmore;

}tag\_oc\_mqtt\_report;

其中eventtime可以留空，值为NULL即可，hasmore用来表明是否还有更多的信息，枚举列表如下：

typedef enum

{

en\_oc\_mqtt\_has\_more\_no = 0,

en\_oc\_mqtt\_has\_more\_yes =1,

}en\_oc\_mqtt\_has\_more;

最后，上报数据封装完成，使用如下API将结构体数据格式化为一个cjson\*类型的数据，便于使用：

cJSON \*oc\_mqtt\_json\_fmt\_report(tag\_oc\_mqtt\_report \*report);

上报消息

上面我们封装的消息是一个cjson数据链表，接下来首先包含cJSON组件的头文件：

#include <cJSON.h>

然后调用cJSON组件提供的API将cjson数据链表直接打印为一个不格式化的字符串，方便发送：

(char \*) cJSON\_PrintUnformatted(const cJSON \*item);

最后，调用OC\_MQTT提供的API，上报这个字符串：

/\*\*

\* @brief the application use this function to send message to the default topic(old interface)

\*

\* @param[in] msg:the message to send

\*

\* @param[in] msg\_len:the message length

\*

\* @param[in] qos: defines as the mqtt does

\*

\* @return code: define by en\_oc\_mqtt\_err\_code while 0 means success

\*/

int oc\_mqtt\_report(uint8\_t \*msg,int len, int qos);

msg：刚刚转化完成打印出的字符串指针；

len：字符串长度；

qos：发布消息质量，如下；

发布消息质量枚举值如下(在mqtt\_al.h中)：

/\*\* @brief enum all the qos supported for the application \*/

typedef enum

{

en\_mqtt\_al\_qos\_0 = 0, ///< mqtt QOS 0

en\_mqtt\_al\_qos\_1, ///< mqtt QOS 1

en\_mqtt\_al\_qos\_2, ///< mqtt QOS 2

en\_mqtt\_al\_qos\_err

}en\_mqtt\_al\_qos\_t;

命令接收

配置连接华为云OC平台时，OC\_MQTT组件会自动订阅主题：

/huawei/v1/devices/{NoteId}/command/json

当OC平台发布该主题数据时，OC\_MQTT组件会拉起接收回调函数将数据保存，进而用户解析接收到的JSON数据即可，其中在上一篇文章中测试时，平台发送开启命令，客户端接收到的消息如下：

{

"msgType":"cloudReq",

"serviceId":"Lightness",

"paras":{"LED\_Ctrl":"on"},

"cmd":"LED\_Ctrl",

"hasMore":0,

"mid":10

}

平台发送关闭命令，客户端接收到的消息如下：

{

"msgType":"cloudReq",

"serviceId":"Lightness",

"paras":{"LED\_Ctrl":"off"},

"cmd":"LED\_Ctrl",

"hasMore":0,

"mid":11

}

接下来编写解析任务时可以参考此数据。

下发数据接收回调函数

实现该回调函数：

static int app\_msg\_deal(void \*arg,mqtt\_al\_msgrcv\_t \*msg)

实现的时候，接收数据大小在msg->msg.len中，接收数据在msg->msg.data中。

实现之后在最开始的连接信息结构体中配置即可：

如果下发命令速度较快，可以使用队列接收数据，但是需要注意，使用会占用更多的动态内存空间。

接收数据处理任务

接收数据处理任务需要单独创建一个任务，与接收回调函数之间使用一个信号量进行同步，具体参考下面的示例。

OC\_MQTT组件自动初始化

在SDK目录中的IoT\_LINK\_1.0.0\iot\_link\link\_main.c中可以看到自动初始化函数：

# 2. 配置准备

Makefile配置

因为本次实验用到的组件较多：

AT框架

ESP8266设备驱动

串口驱动框架

cJSON组件

SAL组件

MQTT组件

MBEDTLS组件

OC\_MQTT组件

这些实验代码全部编译下来，有350KB，而小熊派开发板所使用的主控芯片STM32L431RCT6的 Flash 仅有256KB，会导致编译器无法链接出可执行文件，所以要在makefile中修改优化选项，修改为-Os参数，即最大限度的优化代码尺寸，并去掉-g参数，即代码只能下载运行，无法调试，如图：

ESP8266设备配置

在工程目录中的OS\_CONFIG/iot\_link\_config.h文件中，配置ESP8266设备的波特率和设备名称:

WIFI对接信息配置

SDK：C:\Users\Administrator\.icode\sdk\IoT\_LINK\_1.0.0（其中Administrator是实验电脑的用户名）。

在SDK目录中的iot\_link\network\tcpip\esp8266\_socket\esp8266\_socket\_imp.c文件中，配置连接信息：

之后修改同路径下的esp8266\_socket\_imp.mk文件，如图，将 TOP\_DIR 改为 SDK\_DIR ：

修改mbedtls路径配置

在SDK目录中的iot\_link\network\dtls\mbedtls\mbedtls.mk文件中，如图，将 TOP\_DIR 改为 SDK\_DIR ：

修改paho\_mqtt文件路径

在SDK目录中的iot\_link\network\mqtt\paho\_mqtt\paho\_mqtt.mk文件中，如图，将 TOP\_DIR 改为 SDK\_DIR ：

# 3. 上云实验

编写实验文件

在 Demo 文件夹下创建cloud\_test\_demo文件夹，在其中创建oc\_tls\_mqtt\_demo.c文件。

编写以下代码：

#include <osal.h>

#include <oc\_mqtt\_al.h>

#include <oc\_mqtt\_assistant.h>

#include <cJSON.h>

#include <string.h>

#define DEFAULT\_LIFETIME 60

#define DEFAULT\_SERVER\_IPV4 "49.4.93.24"

#define DEFAULT\_SERVER\_PORT "8883"

#define CN\_MQTT\_EP\_NOTEID "321321321321"

#define CN\_MQTT\_EP\_PASSWD "4ac51ec23edeb3eb34e4"

#define recv\_buf\_len 150

static char recv\_buffer[recv\_buf\_len]; //下发数据接收缓冲区

static int recv\_datalen; //表示接收数据长度

osal\_semp\_t recv\_sync; //命令接收回调函数和处理函数之间的信号量

static int app\_msg\_deal(void \*arg,mqtt\_al\_msgrcv\_t \*msg)

{

int ret = -1;

if(msg->msg.len < recv\_buf\_len)

{

//不超过缓冲区，保存数据

memcpy(recv\_buffer,msg->msg.data,msg->msg.len );

recv\_buffer[msg->msg.len] = '\0';

recv\_datalen = msg->msg.len;

//释放信号量，交由数据处理线程进行处理

osal\_semp\_post(recv\_sync);

ret = 0;

}

return ret;

}

static int task\_recv\_cmd\_entry(void \*args)

{

while(1)

{

/\* 阻塞等待信号量 \*/

osal\_semp\_pend(recv\_sync,cn\_osal\_timeout\_forever);

if(strstr(recv\_buffer, "on"))

{

printf("-----------------LED ON !!! --------------------\r\n");

}

else if(strstr(recv\_buffer, "off"))

{

printf("-----------------LED OFF !!! --------------------\r\n");

}

}

return 0;

}

static int task\_report\_msg\_entry(void \*args)

{

int ret = -1;

oc\_mqtt\_config\_t config; // oc\_mqtt 连接信息配置结构体

tag\_key\_value\_list lst; //有效数据结构体

int lightness\_value = 0; //亮度值

tag\_oc\_mqtt\_report report; //上报数据结构体

cJSON\* root = NULL; //存放上报数据的cjson链表

char\* report\_msg = NULL; //存放上报数据的字符串

/\* 设置连接信息 \*/

config.boot\_mode = en\_oc\_mqtt\_mode\_nobs\_static\_nodeid\_hmacsha256\_notimecheck\_json;

config.msg\_deal = app\_msg\_deal;

config.msg\_deal\_arg = NULL;

config.lifetime = DEFAULT\_LIFETIME;

config.server\_addr = DEFAULT\_SERVER\_IPV4;

config.server\_port = DEFAULT\_SERVER\_PORT;

config.id = CN\_MQTT\_EP\_NOTEID;

config.pwd= CN\_MQTT\_EP\_PASSWD;

config.sec\_type = en\_mqtt\_al\_security\_cas;

/\* 配置并对接云平台 \*/

ret = oc\_mqtt\_config(&config);

if(ret != en\_oc\_mqtt\_err\_ok)

{

printf("config and connect error, ret = %d.\r\n", ret);

return -1;

}

else

{

printf("config and connect success.\r\n");

}

/\* 连接成功后，开始上报消息 \*/

while(1)

{

//封装数据链表

lst.item.name = "Lightness";

lst.item.buf = (char\*)&lightness\_value;

lst.item.len = sizeof(lightness\_value);

lst.item.type = en\_key\_value\_type\_int;

lst.next = NULL;

//封装上报数据

report.hasmore = en\_oc\_mqtt\_has\_more\_no;

report.paralst= &lst;

report.serviceid = "Lightness";

report.eventtime = NULL;

//转换为cjson\*类型数据

root = oc\_mqtt\_json\_fmt\_report(&report);

if(NULL != root)

{

//打印为未格式化的字符串（去掉格式符，减小上报数据长度）

report\_msg = cJSON\_PrintUnformatted(root);

//发布消息

ret = oc\_mqtt\_report((uint8\_t \*)report\_msg,strlen(report\_msg),en\_mqtt\_al\_qos\_1);

if(ret != en\_oc\_mqtt\_err\_ok)

{

printf("report fail, ret = %d.\r\n", ret);

return -1;

}

else

{

printf("report success: lightness\_value = %d, msg = %s\r\n", lightness\_value, report\_msg);

}

/\* 一次消息发布完毕 \*/

//释放内存

osal\_free(report\_msg);

cJSON\_Delete(root);

}

//改变亮度值，实际可以替换为采集实际亮度值

lightness\_value++;

//挂起5s

osal\_task\_sleep(5\*1000);

}

}

int standard\_app\_demo\_main()

{

/\* 创建信号量 \*/

osal\_semp\_create(&recv\_sync,1,0);

/\* 创建任务 \*/

osal\_task\_create("task\_reportmsg",task\_report\_msg\_entry,NULL,0x800,NULL,8);

osal\_task\_create("task\_recv\_cmd",task\_recv\_cmd\_entry,NULL,0x400,NULL,8);

return 0;

}

添加路径

在user\_demo.mk中添加如下：

#example for oc\_tls\_mqtt\_demo

ifeq ($(CONFIG\_USER\_DEMO), "oc\_tls\_mqtt\_demo")

user\_demo\_src = ${wildcard $(TOP\_DIR)/targets/STM32L431\_BearPi/Demos/cloud\_test\_demo/oc\_tls\_mqtt\_demo.c}

endif

添加位置如下：

配置.sdkconfig

特别说明：实验时需要关闭shell组件，否则会因动态内存分配失败而导致TLS无法连接到华为OC平台。

上报数据实验结果

编译，下载，在云端的实验现象如下：

在本地的实验现象如下：

命令下发实验结果

在云端下发“on”命令：

在串口助手中可以看到：

下发“off”命令：

在串口助手中可以看到：