# Mesh Message Send Application Note



**V 0.0.1**

**2019/1/29**

# 修订历史（Revision History）

| 日期 | 版本 | 修改 | 作者 | Reviewer |
|---|---|---|---|---|
| **2018/08/31** | Draft v0.1 | 初稿 | bill | |

# 目 录

# 表目录

# 图目录

未找到图形项目表。

# 词汇表

| 缩写 | 含义 |
| --- | --- |
| | |

# 1 背景介绍

mesh 消息发送需要明确的参数有地址、应用密钥和 TTL。地址包含源地址和目的地址，分别标记消息的发送者和接收者。源地址只能是发送者的单播地址，是 Provisioner 分配的固定值，由 stack 自动配置。而目的地址则可以是除 Unassigned address 外任意地址类型，可以是单播，也可以是组播。

mesh 消息参数的设置，需要根据消息类型来区分对待。

当消息是针对请求消息的应答时，目的地址即原请求消息的源地址，使用相同的 AppKey。此种被动消息的目的地址等参数设置也是被动的。

当消息是 mesh 设备主动发出的，spec 里称作 publish。这些参数通常是 Provisioner 利用 Config Model Publication Set 消息灵活配置的，stack 也会自动处理的，不需要 app 自己干预的。例如：一个开关的开关灯消息发送给哪盏灯，是可动态配置的。

如果 provisioner 没有配置 model 的 publish 参数时，mode 发送消息所需参数如何设置也分情况。

情况一：provisioner 中的 configuration client，是它去配置其他设备的 model 的 publish 等等参数的，那谁来来配置它的 publish？provisioner 是网络管理者，自然需要自己去管理自己的消息发送参数。也就是说，人在 provisioner 设备上操作，进行人为干预。目的地址、密钥和 ttl 的指定，全是人为指定的。例如 provisioner 可以给设备 1 配置，也可以给设备 2 配置。SDK user guide 里介绍的 provisioner 工程管理其他设备时，全部是人为指定目的地址等参数的。

情况二：有些 Model 发送消息比较灵活，不想受限于 Publish 行为，例如 ping control model，此时也是人为自由指定。

情况三：model 自己设定 Publish 参数。例如写死预先协商好的目的地址，地址可以是单播地址、组播地址、虚地址，甚至 0xFFFF。

# 2 获取 Publish 参数

publish 相关操作：

**表 2-1 publish 判断**

| | |
|---|---|
| **函数** | **bool mesh_model_pub_check(mesh_model_info_p pmodel_info)** |
| **功能** | 判断是否被 provisioner 设置了 publish 参数 |
| **参数** | pmodel_info：model 信息 |

**表 2-2 publish 提取**

| | |
|---|---|
| **函数** | **mesh_msg_send_cause_t access_cfg(mesh_msg_p pmesh_msg)** |
| **功能** | 提取 default configuration、publish 等参数，为发送消息预置参数。 |
| **参数** | pmesh_msg：待发送的消息 |

publish 的处理都是由 stack 处理的，并保存在 flash 里，上层 app 只需调取即可。

**表 2-3 publish**

```
1. mesh_msg_send_cause_t generic_on_off_stat(mesh_model_info_p pmodel_info, uint16_t
   dst, uint8_t app_key_index, generic_on_off_t present_on_off, bool optional, generic_on_off_t
   target_on_off, generic_transition_time_t remaining_time)
2. {
3.
4.      mesh_msg_t mesh_msg;
5.      mesh_msg.pmodel_info = pmodel_info;
6.      access_cfg(&mesh_msg);
7.      mesh_msg.pbuffer = (uint8_t *)&msg;
8.      mesh_msg.msg_len = msg_len;
9.      if (0 != dst) // dst == 0 时，使用 publish 参数发送
10.     {
11.         mesh_msg.dst = dst;
12.         mesh_msg.app_key_index = app_key_index;
13.     }
14.     return access_send(&mesh_msg);
15. }
```

如果 model 不关心 publish 参数，可以在调用 access_cfg 后，覆盖掉相关参数即可。

**表 2-4 自主参数**

```
1.  static mesh_msg_send_cause_t ping_control_send(uint16_t dst, uint8_t ttl, uint8_t *pmsg,
2.                                                 uint16_t msg_len, uint8_t app_key_index)
3.  {
4.      mesh_msg_t mesh_msg;
5.      mesh_msg.pmodel_info = &ping_control;
6.      access_cfg(&mesh_msg);
7.      mesh_msg.pbuffer = pmsg;
8.      mesh_msg.msg_len = msg_len;
9.      mesh_msg.dst = dst;//强行覆盖
10.     mesh_msg.ttl = ttl; //强行覆盖
11.     mesh_msg.app_key_index = app_key_index; //强行覆盖
12.     return access_send(&mesh_msg);
13. }
```

# 3 获取 Configuration Client 源地址

设备在配网的时候，通常 provisioner 会分发密钥、绑定 model、设置订阅地址等等。此时 provisioner 是用 configuration client model 发消息给 configuration server model，这些消息的源地址即 Provisioner 的地址。

由于 configuration 过程涉及到许多网络参数操作，configuration server 被封装在了 mesh lib 中。尽管封装在 Lib 里了，上层 app 想知道 configuration client 在配置 configuration server 时做了哪些操作，获取 configuration client 地址，可以通过下述方法。

在第一个 element 创建后，向 configuration server 注册新的 callback cfg_server_receive_peek。

**表 3-1 cfg client address**

```
14.     /** create elements and register models */
15.     mesh_element_create(GATT_NS_DESC_UNKNOWN);
16.     cfg_server.model_receive = cfg_server_receive_peek;
```

**表 3-2 cfg server peek**

```
17. bool cfg_server_receive_peek(mesh_msg_p pmesh_msg)
18. {
19.     bool ret = cfg_server_receive(pmesh_msg);
20.     if(ret)
21.     {
22.         /* The configuration client message can be peeked now! */
23.         printi("cfg_server_receive_peek: cfg client addr = 0x%04x", pmesh_msg->src);
24.     }
25.     return ret;
26. }
```

configuration client 并不会一直发消息，设备端需要将 configuration client 地址存到 flash 中，重新上电后恢复，防止地址丢失。注意：不要在 cfg_server_receive_peek 里重复写 flash，只有地址不同的时候，再重新写入。

# 4 AppKey 选择

stack 支持多应用密钥，但 Provisioner 也许只为 device 配置 1 个。

AppKey 会在调用 API access_cfg 时，被配置成 publish AppKey。如果 publish AppKey 没被配置，需要 app 自己选择一个已经绑定的 AppKey。注意，如果 model 没有绑定这个应用密钥，stack 是不允许 model 使用该 AppKey 发送消息的。

根据 spec 规定，每个 AppKey 都有一个唯一的 global index，但实现上被映射成了 local index。Application 发送消息时，使用的是 AppKey 本地索引 local index。stack 通过 API mesh_model_get_available_key 可以获取一个已经绑定的 AppKey 的 local index。如果 model 绑定了多个 AppKey，又不介意使用哪一个，可以调用 API mesh_model_get_available_key 获取一个可用的 local index。如果 model 需要使用特定 AppKey，例如应用指定了 AppKey 的 global index，此时可以通过 API app_key_index_from_global 根据 AppKey 的 global index，获取 local index。

# 5 TTL

TTL 会在调用 API access_cfg 时，被配置成 publish TTL。如果 publish TTL 没被配置，会被配成设备 default TTL。如果 app 想自行修改，在调用 API access_cfg 后修改 TTL 参数。

# 参考文献

[1] Mesh Profile Specification

[2] Mesh Model Specification

[3] RTL8762C Mesh SDK User Guide

# 附录