沁恒低功耗蓝牙 MESH 软件开发参考手册

V1. 1

2022 年 7 月 14 日

目录

1. 概述 5
1.1 低功耗蓝牙 mesh 角色功能介绍 5
2. 开发平台 6
2.1 芯片概述 6
2.2 软件概述 6
3. 配置 MESH 协议栈 7
3.1 CONFIG_MESH_ADV_BUF_COUNT_DEF 7
3.2 CONFIG_MESH_RPL_COUNT_DEFRPL 7
3.3 CONFIG_MESH_ALLOW_RPL_CYCLE 7
3.4 CONFIG_MESH_IVU_DIVIDER_DEF 7
3.5 CONFIG_MESH_PROXY_FILTER_DEF 7
3.6 CONFIG_MESH_MSG_CACHE_DEF 7
3.7 CONFIG_MESH_APPKEY_COUNT_DEF 7
3.8 CONFIG_MESH_MOD_KEY_COUNT_DEF 7
3.9 CONFIG_MESH_MOD_GROUP_COUNT_DEF 8
3.10 CONFIG_MESH_ALLOW_SAME_ADDR 8
3.11 CONFIG_MESH_UNSEG_LENGTH_DEF 8
3.12 CONFIG_MESH_RX_SDU_DEF 8
3.13 CONFIG_MESH_SECTOR_COUNT_DEF 8
3.14 CONFIG_MESH_SECTOR_SIZE_DEF 8
3.15 CONFIG_MESH_NVS_ADDR_DEF 8
3.16 CONFIG_MESH_RPL_STORE_RATE_DEF 8
3.17 CONFIG_MESH_SEQ_STORE_RATE_DEF 8
3.18 CONFIG_MESH_STORE_RATE_DEF 9
3.19 CONFIG_MESH_FRIEND_SUB_SIZE_DEF 9
3.20 CONFIG_MESH_QUEUE_SIZE_DEF 9
3.21 CONFIG_MESH_FRIEND_RECV_WIN_DEF 9
3.22 CONFIG_MESH_LPN_REQ_QUEUE_SIZE_DEF 9
3.23 CONFIG_MESH_LPN_POLLINTERVAL_DEF 9
3.24 CONFIG_MESH_LPN_POLLTIMEOUT_DEF 9
3.25 CONFIG_MESH_LPN_RECV_DELAY_DEF 9
3.26 CONFIG_MESH_RETRY_TIMEOUT_DEF 9
3.27 CONFIG_MESH_PROV_NODE_COUNT_DEF 9
3.28 CONFIG_MESH_RF_ACCESSADDRESS 10
4. 初始化模型功能 11
4.1 初始化 ROOT 模型 11
4.1.1 定义应用密钥和订阅地址 11
4.1.2 定义 CONFIG 服务模型用户配置字 11
4.1.3 定义操作码和对应处理函数 13
4.2 初始化厂商自定义模型 13
4.2.1 厂商 ID 以及模型 ID. 13
4.2.2 定义厂商模型应用密钥和订阅地址 13

4.2.3 定义厂商模型操作码和对应处理函数 13
4.2.4 定义模型配置字 14
4.2.5 初始化函数 16
5. 配网流程 17
5.1 普通节点 17
5.1.1 发送未配网广播 17
5.1.2 普通节点配网回调 17
5.1.2.1 link_open 回调 18
5.1.2.2 link_close 回调 18
5.1.2.3 prov_complete 回调 18
5.1.2.4 prov_reset 回调 18
5.1.2.5 HEALTH 模型回调 19
5.1.3 普通节点自配网 19
5.2 中心节点 19
5.2.1 中心节点自配网 19
5.2.2 中心节点配网回调 20
5.2.2.1 prov_complete 回调 20
5.2.2.2 unprov_recv 回调 21
5.2.2.3 link_open 回调 22
5.2.2.4 link_close 回调 22
5.2.2.5 node_added 回调 22
6. 管理网络 24
6.1 通过配网者远程管理网络 24
6.1.1 添加应用密钥 24
6.1.2 绑定应用密钥到指定模型 24
6.1.3 添加订阅地址到指定模型 24
6.1.4 删除节点命令 25
6.1.5 例程配网者自配网后对自身模型的配置流程 26
6.1.6 例程配网者对新入网设备的配置流程 27
6.2 本地管理自身网络信息 29
6.2.1 添加应用密钥到本地网络信息 29
6.2.2 绑定应用密钥到本地模型 29
6.2.3 添加订阅地址到本地模型 30
6.2.4 删除本地配网信息 30
7. 沁恒自定义透传模型 31
7.1 发送数据 31
7.1.1 发送参数 31
7.1.2 发送函数返回值 31
7.1.3 发送函数使用示例 32
7.1.4 发送数据 API 33
7.2 接收数据 34
7.2.1 沁恒自定义透传模型操作码处理函数 34
7.2.2 沁恒自定义透传模型回调 35
8. 低功耗功能与朋友关系 37

8.1 朋友节点 37
8.2 低功耗节点 37
9. 连接手机 39
9.1 代理配网功能 39
9.2 通过独立 ble 功能连接手机 39
修订记录 41

1. 概述

1.1 低功耗蓝牙 mesh 角色功能介绍

成为蓝牙 mesh 网络中一员的设备被称为节点(Node)，蓝牙 mesh 规格定义了节点可能拥有的特性。具有这些特性中的一个或多个，即表示节点可以在网络中扮演相应的特殊角色。

朋友功能：朋友节点能够存储发往相关低功耗节点的消息，随后再进行转发。

低功耗功能：低功耗节点功率受限，可借助朋友节点的支持，在蓝牙 mesh 网络中有效地运作，借此大幅降低电量消耗。

代理功能：代理节点可通过承载层（广播承载层或 GATT 承载层）接受信息，并通过另一个广播承载层或 GATT 承载层重新发送消息。通常可用于连接处于相同网络配置中的手机，或者用于使用手机作为管理网络的配网者的场景。

中继功能：中继节点可通过 adv（广播承载层），接收并重新发送蓝牙 mesh 消息，中继特性能让蓝牙 mesh 消息在设备之间实现多次跳跃，最多可进行 127 跳，传送距离可超过两台设备之间直接进行无线电传输的范围，从而覆盖整个网络。

蓝牙 mesh 网络采用一种称为“网络泛洪（flooding）”的方式来发布和中继消息。消息不会通过某一进程进行路由，也不会沿着由一系列特定设备构成的特定路径来进行传输。相反，传输范围内的所有设备都会接收消息，负责中继的设备能将消息转发至其传输范围内的所有其他设备。

为了避免数据堵塞，所有数据包都包含一个称为 TTL（生存次数）的字段，它可用于限制消息中继的跳数。网络中的设备也可通过此了解传到其他每台设备之间的跳数。这能够让设备将 TTL 设置为最佳值，从而避免不必要的中继操作。

同时每台设备都包含消息缓存，以确定自身是否已经中继过该消息。如果是，则会立即丢弃该消息。

本手册所提到的例程均默认具有中继功能，其他功能都有独立的例程演示。

2. 开发平台

2.1 芯片概述

CH58x 是集成 BLE 无线通讯的 32 位 RISC-V 微控制器。片上集成 2Mbps 低功耗蓝牙通讯模块，两个全速 USB 主机和设备控制器收发器，2 个 SPI，RTC 等丰富外设资源。本手册均以 CH58x 开发平台举例说明，本司其他低功耗蓝牙芯片同样可参考此手册。

2.2 软件概述

本手册所引用的全部代码均来源于 CH58xEVT 软件包。

除 mesh 相关例程之外，还包括丰富的外设例程以及 BLE 蓝牙应用例程。

3. 配置 MESH 协议栈

找到任意 mesh 例程中的 APP/include 目录下，打开 app_mesh_config.h 文件，在参数配置中，可以修改所有 mesh 协议栈内使用的参数，以下为对一些常用参数的说明。

3.1 CONFIG_MESH_ADV_BUF_COUNT_DEF

Net 数据缓存个数：上层每调用单次发送，会占用一个 Net 数据缓存，底层发送完成后自动释放。如果当前缓存已满，应用层调用发送会返回失败。

3.2 CONFIG_MESH_RPL_COUNT_DEFRPL

数据缓存个数：每收到一个来自不同网络地址的数据包，则需要占用一个 RPL 数据缓存，如果缓存已满，则无法再接收新的网络地址发来的数据包。开发时应该考虑到需要与多少不同的设备通讯，保证 RPL 数据缓存大于需要通讯的设备数量。

3.3 CONFIG_MESH_ALLOW_RPL_CYCLE

是否使能 RPL 循环：网络中每一个设备发送的数据都会被其他设备记录其对应 RPL 信息，存储 RPL 信息将占用 RAM 以及 FLASH，如果资源有限，可以使能此宏定义，底层将会用新的 RPL 信息自动覆盖旧数据，并且不存储到 FLASH，每次重启将清空。

注意：使能此功能后，将无法抵御很久之前的重放攻击。

3.4 CONFIG_MESH_IVU_DIVIDER_DEF

IV Update State Timer 基于 96H 的分频系数：多长时间进行一次 IV 更新，最长 96H 最短 1H。应用层无需关心。

3.5 CONFIG_MESH_PROXY_FILTER_DEF

代理黑白名单功能存储个数：仅对代理节点生效。当类型为白名单时，决定代理节点能使用代理功能与多少不同客户端连接。当类型为黑名单时，决定代理节点不能与多少客户端连接。

3.6 CONFIG_MESH_MSG_CACHE_DEF

消息贮存个数：每收到一个新的数据包都会占用一个消息贮存，贮存满后会自动清除最早的消息贮存。

3.7 CONFIG_MESH_APPKEY_COUNT_DEF

APP key 个数：模型（节点）需要有 APP key 才可以发送接收数据，配网者可以给不同的模型（节点）分配不同的 APP key，模型（节点）间如果使用的 APP key 不同，则无法互相通讯。

3.8 CONFIG_MESH_MOD_KEY_COUNT_DEF

可存储的模型密钥数量：每个模型可以存储的 APP key 数量。

3.9 CONFIG_MESH_MOD_GROUP_COUNT_DEF

可存储的订阅地址数量：每个模型可以存储的订阅地址数量。

3.10 CONFIG_MESH_ALLOW_SAME_ADDR

是否允许一个网络中存在同地址的节点（使能后分包功能不可用）：此配置属于特殊应用，如所有自配网使用相同的网络地址，不同设备的区分全部由应用层管理的应用。

3.11 CONFIG_MESH_UNSEG_LENGTH_DEF

不分包消息支持的长度：MESH 里单包发送有长度限制，默认值为 7，超过限制则需要分包，分包发送需要来回应答，通讯延迟比较高，所以如果有大数据传输，建议修改此长度，同网络里所有设备配置需统一。注意如果需要与其他厂商的 SIG MESH 通讯，则不能修改此值。

3.12 CONFIG_MESH_RX_SDU_DEF

每个接收的分包消息的最大字节数：能够接收的最大分包数据长度，如果收到超过此长度的分包数据则不会接收。

3.13 CONFIG_MESH_SECTOR_COUNT_DEF

NVS 存储使用扇区个数：MESH 协议栈会存储配网信息在 dataflash 中，每个扇区大小为 512B，如果做配网者，需要将很多个节点配进自身网络，则需要增加扇区个数。

3.14 CONFIG_MESH_SECTOR_SIZE_DEF

NVS 存储扇区大小：初始化时会自动检测当前配置需要的单个扇区空间是否满足要求，当增加 RPL 缓存、增加存储密钥个数、增加配网数量等等都会需要更大的扇区空间，如果初始化返回“ENOSPC”错误，则需要加大此配置。

3.15 CONFIG_MESH_NVS_ADDR_DEF

NVS 存储首地址：MESH 协议栈存储配网信息在 dataflash 中的首地址，注意 dataflash 的分配。

3.16 CONFIG_MESH_RPL_STORE_RATE_DEF

RPL 更新超过多少次后存储：默认如果接收到 60 次来自同一地址的不同数据包，则存储一次当前 RPL 缓存到 NVS 中。

3.17 CONFIG_MESH_SEQ_STORE_RATE_DEF

SEQ 更新超过多少次后存储：MESH 协议栈每次发送数据都会更新 SEQ 值，默认每发送 60 次不同的数据包后存储一次当前 SEQ 值到 NVS 中。每次重新上电初始化协议栈会将 NVS 中存储的 SEQ 值加上这里的配置值使用并存储 SEQ 值到 NVS 中。

3.18 CONFIG_MESH_STORE_RATE_DEF

其他信息更新后存储的超时时长(s)：其他信息如 KEY 更新、MODEL 更新等会等待 2 秒后存储，防止连续更新损耗 flash 寿命。

3.19 CONFIG_MESH_FRIEND_SUB_SIZE_DEF

朋友节点支持的订阅个数：朋友节点会代替低功耗节点存储发往低功耗节点的数据，包括低功耗节点订阅的地址。

3.20 CONFIG_MESH_QUEUE_SIZE_DEF

朋友节点存储的消息队列大小：朋友节点存储发往低功耗节点的数据的缓存大小。

3.21 CONFIG_MESH_FRIEND_RECV_WIN_DEF

朋友节点接收窗口大小(ms)：低功耗节点在向朋友节点发送 POLL 请求数据后，进入接收状态的接收窗口大小，时间越短功耗越低但可能导致接收率变低。

3.22 CONFIG_MESH_LPN_REQ_QUEUE_SIZE_DEF

低功耗节点的请求消息队列大小：低功耗节点要求的朋友节点需要支持的消息队列大小的最小值。

3.23 CONFIG_MESH_LPN_POLLINTERVAL_DEF

低功耗节点的请求消息间隔(100ms)：低功耗节点每隔多久唤醒一次并向朋友节点请求数据。

3.24 CONFIG_MESH_LPN_POLLTIMEOUT_DEF

低功耗节点的请求消息超时时长(100ms)：低功耗节点在连续多久向朋友节点请求数据失败之后，认为朋友关系断开。

3.25 CONFIG_MESH_LPN_RECV_DELAY_DEF

低功耗节点支持的接收延迟(ms)：低功耗节点在向朋友节点发送 POLL 请求数据后，等待多久进入接收状态。

3.26 CONFIG_MESH_RETRY_TIMEOUT_DEF

朋友关系重建等待时长(s)：当低功耗节点与上一个朋友节点断开连接后，等待多久之后向周围发送建立朋友关系的请求。

3.27 CONFIG_MESH_PROV_NODE_COUNT_DEF

配网发起者支持的配网设备节点个数：配网者例程需要修改此参数时，注意需要同时修改 RPL 数据缓存个数，如果出现重新上电信息丢失的现象，还需要修改 NVS 存储使用扇区个数。每增加一个节点，需要占用 50 字节 NVS 存储空间

3.28 CONFIG_MESH_RF_ACCESSADDRESS

ADV_RF 配置：SIG MESH 默认使用 BLE 广播信道，由于广播信道数据繁多，建议使用非广播的其他 37 个信道通讯，同时使用与 BLE 广播不同的接入地址。注意需要在同一个网络里的所有设备，这里的配置需相同。

4. 初始化模型功能

MESH 的所有通信都必须通过模型来完成，每种模型定义一系列消息类型，包含多个操作码及其对应处理函数，一个或者多个模型组合为一个元素。每一个元素拥有一个独立的网络地址，一个设备可以包含一个或者多个元素。

当协议栈收到一包数据时，会解析数据发送的目的地址（recv_dst），并且与自身包含的所有元素的地址对比。如果存在元素与目的地址匹配，则解析数据内的操作码（opcode），并且与此元素包含的模型定义的操作码对比。如果存在模型与数据内操作码匹配，则收下数据包并执行对应操作码的处理函数。

每个模型都可以绑定独立的应用密钥（APP_KEY）和订阅地址（SUB_ADDR）。

应用密钥用于收发数据的加密，模型必需至少拥有一个应用密钥，否则无法使用，密钥可通过配网者下发添加并绑定到模型（见配网流程），或者配网者下发添加后自行绑定（见接入天猫精灵），也可以自行添加绑定（见配网流程中的自配网）。

订阅地址用于分组管理模型，多个模型可以同时订阅同一个订阅地址，当 MESH 网络里有发往此订阅地址的消息，所有订阅该地址的模型都将收到此消息。订阅地址不需要强制拥有，可通过配网者下发添加到模型（见配网流程），也可以自行添加（见接入天猫精灵）。

4.1 初始化 ROOT 模型

Root 模型为 SIG MESH 协议规范中已经定义的模型，模型 ID 为两字节。下面的代码中包含三种模型：CONFIG 服务模型、HEALTH 服务模型、通用开关模型。

```
1. static struct bt_mesh_model root_models[] = {
2.     BLE_MESH_MODEL_CFG_SRV(cfg_srv_keys, cfg_srv_groups, &cfg_srv),
3.     BLE_MESH_MODEL_HEALTH_SRV(health_srv_keys, health_srv_groups, &health_srv,
   &health_pub),
4.     BLE_MESH_MODEL(BLE_MESH_MODEL_ID_GEN_ONOFF_SRV, gen_onoff_op, NULL, gen_ono
   ff_srv_keys, gen_onoff_srv_groups, NULL),
5. };
```

4.1.1 定义应用密钥和订阅地址

每个模型都有其独立的应用密钥和订阅地址，例如上述代码中 cfg_srv_keys 为 CONFIG 服务模型绑定的应用密钥，cfg_srv_groups 为 CONFIG 服务模型添加的订阅地址。这两个数组由应用层定义，当协议栈收到相应配置命令时，会修改其值，用户也可以自行修改，注意修改后还需要将其更新到 NVS 中（见接入天猫精灵）

```
1. uint16_t cfg_srv_keys[CONFIG_MESH_MOD_KEY_COUNT_DEF] = {BLE_MESH_KEY_UNUSED};
2. uint16_t cfg_srv_groups[CONFIG_MESH_MOD_GROUP_COUNT_DEF] = {BLE_MESH_ADDR_UNASS
   IGNED};
```

4.1.2 定义 CONFIG 服务模型用户配置字

用户可在此配置各项功能的开启和关闭（例程已通过 mesh_config.h 里的宏定义自动配置，正常情况这里无需改动），修改默认 TTL 值，配置底层发送和中转重传次数和间隔（实际发送间隔会在此基础上加上 10ms 内的随机延迟，由协议栈自动完成），添加 CONFIG 服务模型状态回调函数 cfg_srv_rsp_handler。

```
1. static struct bt_mesh_cfg_srv cfg_srv = {
2. #if(CONFIG_BLE_MESH_RELAY)
```

```
    .relay = BLE_MESH_RELAY_ENABLED,
#endif
    .beacon = BLE_MESH_BEACON_ENABLED,
#if(CONFIG_BLE_MESH_FRIEND)
    .frnd = BLE_MESH_FRIEND_ENABLED,
#endif
#if(CONFIG_BLE_MESH_PROXY)
    .gatt_proxy = BLE_MESH_GATT_PROXY_ENABLED,
#endif
    /* Default TTL is 3 */
    .default_ttl = 3,
    /* The transport layer sends data for 7 retries at an interval of 10ms (without internal random delay) */
    .net_transmit = BLE_MESH_TRANSMIT(7, 10),
    /* The transport layer relay data for 7 retries at an interval of 10ms (without internal random delay) */
    .relay_retransmit = BLE_MESH_TRANSMIT(7, 10),
    .handler = cfg_srv_rsp_handler,
};
```

每当收到例如“添加应用密钥”“添加订阅地址”等等命令时，都会进入 CONFIG 服务模型状态回调函数告知用户当前命令的类型以及执行状态。可通过此回调判断设备是否已经配置完成。下述代码中的回调函数里只列举了部分常用命令，更多命令类型可以在协议栈头文件的 BLE_MESH_MODEL_OP 定义中查看。

```
static void cfg_srv_rsp_handler( const cfg_srv_status_t *val )
{
    if(val->cfgHdr.status)
    {
        APP_DBG("warning opcode 0x%02x", val->cfgHdr.opcode);
        return;
    }
    if(val->cfgHdr.opcode == OP_APP_KEY_ADD)
    {
        APP_DBG("App Key Added");
    }
    else if(val->cfgHdr.opcode == OP_MOD_APP_BIND)
    {
        APP_DBG("Vendor Model Binded");
    }
    else if(val->cfgHdr.opcode == OP_MOD_SUB_ADD)
    {
        APP_DBG("Vendor Model Subscription Set");
    }
    else
    {
```

```
22.         APP_DBG("Unknow opcode 0x%02x", val->cfgHdr.opcode);
23.     }
24. }
```

4.1.3 定义操作码和对应处理函数

CONFIG 模型和 HEALTH 模型的 ID 和操作码及其处理函数由库封好，用户只需要注册对应回调查看执行状态即可。其他添加的模型则需要用户自行定义，例如本章节开头初始化定义的 GEN_ONOFF 模型，用户在初始化时需要传入其操作码结构体数组指针，其定义如下。

```
1.  const struct bt_mesh_model_op gen_onoff_op[] = {
2.      {BLE_MESH_MODEL_OP_GEN_ONOFF_GET, 0, gen_onoff_get},
3.      {BLE_MESH_MODEL_OP_GEN_ONOFF_SET, 2, gen_onoff_set},
4.      {BLE_MESH_MODEL_OP_GEN_ONOFF_SET_UNACK, 2, gen_onoff_set_unack},
5.      BLE_MESH_MODEL_OP_END,
6.  };
```

第一列为操作码；第二列为此操作命令有效数据的最小长度，当收到对应数据时，协议栈会自动判断有效数据长度，如果不满足此处定义的最小长度，则不会上报应用层；第三列为用户定义的对应处理函数。这里的有效数据长度格式以及操作码由 SIG MESH 官方定义，详细可查看《MshMDLv1.0.1.pdf》，官方操作码在协议栈头文件中有全部定义。

4.2 初始化厂商自定义模型

除了官方已定义的标准模型之外，用户可自行添加厂商定义模型，这里以沁恒自定义透传服务模型为例，演示初始化流程以及使用方式。

```
1.  struct bt_mesh_model vnd_models[] = {
2.      BLE_MESH_MODEL_VND_CB(CID_WCH, BLE_MESH_MODEL_ID_WCH_SRV, vnd_model_srv_op,
     NULL, vnd_model_srv_keys, vnd_model_srv_groups, &vendor_model_srv, NULL),
3.  };
```

4.2.1 厂商 ID 以及模型 ID

初始化厂商模型时，需要提供厂商 ID，可在蓝牙 SIG 官网中查询对应公司的 ID 号，沁恒的厂商 ID 为 0x07D7，即上述代码中的 CID_WCH。

沁恒自定义透传模型分为服务模型（BLE_MESH_MODEL_ID_WCH_SRV），以及客户端模型（BLE_MESH_MODEL_ID_WCH_CLI），分别对应两套操作码以及处理函数。

4.2.2 定义厂商模型应用密钥和订阅地址

同 ROOT 模型一样，厂商模型也有其独立的应用密钥和订阅地址，可由配网者管理，也可以自行管理。

```
1.  uint16_t vnd_model_srv_keys[CONFIG_MESH_MOD_KEY_COUNT_DEF] = {BLE_MESH_KEY_UNUS
    ED};
2.  uint16_t vnd_model_srv_groups[CONFIG_MESH_MOD_GROUP_COUNT_DEF] = {BLE_MESH_ADDR
    _UNASSIGNED};
```

4.2.3 定义厂商模型操作码和对应处理函数

沁恒自定义透传模型包含五个操作码：

```
1.  #define OP_VENDOR_MESSAGE_TRANSPARENT_CFM    BLE_MESH_MODEL_OP_3(0xCB, CID_WCH)
```

```
#define OP_VENDOR_MESSAGE_TRANSPARENT_WRT    BLE_MESH_MODEL_OP_3(0xCC, CID_WCH)
#define OP_VENDOR_MESSAGE_TRANSPARENT_ACK    BLE_MESH_MODEL_OP_3(0xCD, CID_WCH)
#define OP_VENDOR_MESSAGE_TRANSPARENT_IND    BLE_MESH_MODEL_OP_3(0xCE, CID_WCH)
#define OP_VENDOR_MESSAGE_TRANSPARENT_MSG    BLE_MESH_MODEL_OP_3(0xCF, CID_WCH)
```

其中 WRT 只能由客户端模型发送，服务模型回复 CFM 用于应答 WRT，IND 只能由服务模型发送，客户端模型回复 ACK 用于应答 IND。MSG 属于通用操作码，客户端和服务模型都可以使用。

通常建议使用 MSG 用于透传数据，使用最简单，但要注意 MSG 无应答机制，无法保证对方一定能收到。

WRT 和 IND 为有应答传输，当用户调用发送后，会启动一个超时任务，若超时没有收到对应的 CFM 或者 ACK 应答，则会通过回调函数上报发送失败。由于需要等待超时，所以同时必需等到应答或者超时后才能调用下一次发送。

由此可见服务模型只可能收到三种操作码数据，分别是 MSG、WRT 和 ACK。这里由于数据内容长度非固定值，所以长度判断都填 0。

```
const struct bt_mesh_model_op vnd_model_srv_op[] = {
    {OP_VENDOR_MESSAGE_TRANSPARENT_MSG, 0, vendor_message_srv_trans},
    {OP_VENDOR_MESSAGE_TRANSPARENT_WRT, 0, vendor_message_srv_write},
    {OP_VENDOR_MESSAGE_TRANSPARENT_ACK, 0, vendor_message_srv_ack},
    BLE_MESH_MODEL_OP_END,
};
```

4.2.4 定义模型配置字

模型配置字用于管理沁恒自定义服务模型的当前状态，这里只给 tid 赋初值，以及注册操作码对应处理的状态回调，结构体内其他的值将在后续初始化回调中以及使用过程中赋值。

```
struct bt_mesh_vendor_model_srv vendor_model_srv = {
    .srv_tid.trans_tid = 0xFF,
    .handler = vendor_model_srv_rsp_handler,
};
```

vendor_model_srv 的结构体定义如下

```
struct bt_mesh_vendor_model_srv
{
    struct bt_mesh_model          *model;
    uint32_t                       op_req;
    uint32_t                       op_pending;
    struct vendor_model_srv_tid    srv_tid;
    vendor_model_srv_rsp_handler_t handler;
};
```

其中 model 为所属模型结构体指针；op_req 用于记录当前发送的操作码，op_pending 用于记录当前操作期待的应答操作码，此两者只有用需要应答的操作时才会使用到；srv_tid 为记录当前收到的数据包 ID，发送方每调用一次发送数据函数，对应操作码的数

据包 ID 会加一，最大值 255，之后循环，当发送数据时开启了应用层的重传功能，此 ID 可用于屏蔽收到的重复数据。

注意：当多个服务互相通信时，需要修改例程默认对此 srv_tid 判断的判断流程，可自行添加其他判断逻辑。

handler 为应用层注册的此模型内操作码回调函数，在沁恒自定义服务模型的源文件的操作码处理函数中可以看到，处理完数据后，会调用这个回调函数通知应用层收到的内容和来源网络地址。这里注册的回调函数为 vendor_model_srv_rsp_handler。

```
1.  static void vendor_model_srv_rsp_handler(const vendor_model_srv_status_t *val)

2.  {
3.      if(val->vendor_model_srv_Hdr.status)
4.      {
5.          // 有应答数据传输 超时未收到应答
6.          APP_DBG("Timeout opcode 0x%02x", val->vendor_model_srv_Hdr.opcode);
7.          return;
8.      }
9.      if(val->vendor_model_srv_Hdr.opcode == OP_VENDOR_MESSAGE_TRANSPARENT_MSG)
10.     {
11.         // 收到透传数据
12.         APP_DBG("len %d, data 0x%02x from 0x%04x", val-
    >vendor_model_srv_Event.trans.len, val-
    >vendor_model_srv_Event.trans.pdata[0], val-
    >vendor_model_srv_Event.trans.addr);
13.     }
14.     else if(val-
    >vendor_model_srv_Hdr.opcode == OP_VENDOR_MESSAGE_TRANSPARENT_WRT)
15.     {
16.         // 收到 write 数据
17.         APP_DBG("len %d, data 0x%02x from 0x%04x", val-
    >vendor_model_srv_Event.write.len, val-
    >vendor_model_srv_Event.write.pdata[0], val-
    >vendor_model_srv_Event.write.addr);
18.     }
19.     else if(val-
    >vendor_model_srv_Hdr.opcode == OP_VENDOR_MESSAGE_TRANSPARENT_IND)
20.     {
21.         // 发送的 indicate 已收到应答
22.     }
23.     else
24.     {
25.         APP_DBG("Unknow opcode 0x%02x", val->vendor_model_srv_Hdr.opcode);
26.     }
27. }
```

在例程的回调中，除了上述代码中的打印外，还有对于应用层删除节点命令的处理等

其他判断，这里的命令协议是由用户自行拟定的，例程为演示作用。

4.2.5 初始化函数

在初始化函数中，给模型配置字赋值，并注册服务模型的 TMOS 任务处理函数。

```
1. static int vendor_model_srv_init(struct bt_mesh_model *model)
2. {
3.     vendor_model_srv = model->user_data;
4.     vendor_model_srv->model = model;
5.
6.     vendor_model_srv_TaskID = TMOS_ProcessEventRegister(
7.         vendor_model_srv_ProcessEvent);
8.     return 0;
9. }
```

5. 配网流程

MESH 设备需要经过配网之后才可以通信，简单的讲，配网就是一个分配网络地址，告知网络信息，下发网络密钥的过程。所有设备只能与有相同网络密钥的设备互相通信，上一章的应用密钥则是在网络密钥基础上再次加密。

普通节点在配网成功后，需要接收到中心节点下发的应用密钥才能通信，因此例程在下述流程的基础上，增加了检测应用密钥下发是否成功的功能。

当普通节点配网成功后，检测自身是否拥有应用密钥，如果超过一段时间未收到配网者下发的应用密钥，则认为自身配网失败，自行恢复出厂设置。

同样，中心节点在成功配网了一个普通节点后，会检查下发应用密钥的流程是否正确执行，如果未能收到普通节点的应答，则认为该节点配网失败，自行删除本地存储的对应节点信息

5.1 普通节点

普通节点本身没有密钥，也没有网络地址，需要由中心节点（配网者）将其加入网络。

如有特殊应用，也可以通过自配网的方式，不需要中心节点的参与。

5.1.1 发送未配网广播

普通节点初始化完成后，调用 prov_enable 函数，会进入未配网广播模式，并开启扫描，期望收到对应配网命令。未配网广播会包含设备的 16 字节 uuid 信息，可以通过修改 dev_uuid 数组改变广播内容。

```
1.  static uint8_t dev_uuid[16] = {0};
```

例程默认在初始化之后就会调用 prov_enable，随后会一直处于蓝牙开启状态，功耗较高，如果有低功耗考虑的产品，可以修改为自行管理何时开启。

```
1.  static void prov_enable(void)
2.  {
3.      if(bt_mesh_is_provisioned())
4.      {
5.          return;
6.      }
7.
8.      // Make sure we're scanning for provisioning inviations
9.      bt_mesh_scan_enable();
10.     // Enable unprovisioned beacon sending
11.     bt_mesh_beacon_enable();
12.
13.     if(CONFIG_BLE_MESH_PB_GATT)
14.     {
15.         bt_mesh_proxy_prov_enable();
16.     }
17. }
```

5.1.2 普通节点配网回调

普通节点配网相关回调共有四种，在配网参数结构体中可以看到。除此之外还有一种常见回调为 HEALTH 模型回调。

```
static const struct bt_mesh_prov app_prov = {
    .uuid = dev_uuid,
    .link_open = link_open,
    .link_close = link_close,
    .complete = prov_complete,
    .reset = prov_reset,
};
```

5.1.2.1 link_open 回调

当收到配网命令时，首先便会进入 link_open 回调。Link 是配网时建立的临时连接，用于提供一个数据通道给配网者和待配网设备通信。所以除了 prov_reset 回调之外，其余回调都会在保持 link 连接的时候上报。

```
static void link_open(bt_mesh_prov_bearer_t bearer)
{
    APP_DBG("");
}
```

5.1.2.2 link_close 回调

当 link 连接关闭时会进入 link_close 回调，同时通过参数获取关闭的原因。出现以下情况时会关闭 link：配网成功、配网数据交互错误、接收超时、用户主动取消。参数对应原因可以参考协议栈头文件 link close reason 的宏定义。超时时长固定为 60s。

```
static void link_close(bt_mesh_prov_bearer_t bearer, uint8_t reason)
{
    if(reason != CLOSE_REASON_SUCCESS)
        APP_DBG("reason %x", reason);
}
```

5.1.2.3 prov_complete 回调

只有在 link 连接过程中才会进入此回调，意味着当前配网已完成，设备可以进行正常通信，通常在此回调之后就会进入 link_close 回调，关闭原因为成功。

```
static void prov_complete(uint16_t net_idx, uint16_t addr, uint8_t flags, uint32_t iv_index)
{
    APP_DBG("");
}
```

在配网完成回调中，可以获取当前网络的网络密钥序号，自身网络地址，当前网络中网络密钥更新标志，以及当前网络 IV 序号。其中网络密钥序号会在发送数据时需要，网络地址可用于用户管理网络。

5.1.2.4 prov_reset 回调

配网信息重置回调，会在收到重置节点命令后上报或者用户自行调用清除配网信息 API 后上报（见第六章管理网络）。可在此回调中重新使能配网，回到未配网广播状态。

```
static void prov_reset(void)
{
    APP_DBG("");
    prov_enable();
```

```
}
```

5.1.2.5 HEALTH 模型回调

此模型回调有两种，分别是注意开启回调和注意关闭回调。配网开始后便会进入注意开启回调，用户可在此时用灯光闪烁、震动、蜂鸣等方式，辨认当前进入配网模式的设备，随后在配网命令中包含的超时时间后，会进入注意关闭回调，用户可在此时关闭之前打开的用于辨认的功能。

```
/* Attention on */
void app_prov_attn_on(struct bt_mesh_model *model)
{
    APP_DBG("app_prov_attn_on");
}

/* Attention off */
void app_prov_attn_off(struct bt_mesh_model *model)
{
    APP_DBG("app_prov_attn_off");
}
```

5.1.3 普通节点自配网

普通节点自配网的方式与中心节点的自配网方式相同（见 5.2.1 中心节点自配网），并且还需要自行配置自身的模型网络信息，方式与中心节点配置自身模型的流程相同（见 6.1.5 例程配网者自配网后对自身模型的配置流程）。

EVT 提供了普通节点自配网的例程，用户可直接在例程上修改。

注意：例程自配网所需的网络密钥是固化在代码中的，假设需要组建两个网络，但是使用了同一套代码，则会导致此两个网络相互干扰无法使用。

注意：由于网络地址不再由配网者分配，用户需自行确保同一网络中所有设备的网络地址各不相同，或者通过使能 CONFIG_MESH_ALLOW_SAME_ADDR 配置，所有设备使用相同网络地址，通过应用层添加判断来区分消息来源和目的地。

5.2 中心节点

中心节点即配网者，每个网络同时只可以有一个配网者角色，可以将设备添加到自身网络，并管理网络里的设备。

5.2.1 中心节点自配网

中心节点在第一次上电时，会先执行自配网流程，将自身加入网络，作为网络里的第一个设备，随后才可以使能配网者角色的功能。

自配网需要用户定义自身网络的网络密钥、网络密钥序号、设备密钥、网络地址、网络密钥更新状态标志，以及 IV 序号。每个网络都应拥有独立的网络密钥，否则互相的消息会扰乱网络通信；网络密钥序号用于映射网络密钥，后续发送数据时需要用到；中心节点的网络地址通常为 0x0001，也可以自行分配，注意网络地址的范围为 0x0001~0x7FFF；其余参数通常情况下无需关心。

```
static const uint8_t self_prov_net_key[16] = {
    0x00, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef,
    0x00, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef,
```

```
};
static const uint8_t self_prov_dev_key[16] = {
    0x00, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef,
    0x00, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef,
};
const uint16_t self_prov_net_idx = 0x0000;
const uint32_t self_prov_iv_index = 0x00000000;
const uint16_t self_prov_addr = 0x0001;
const uint8_t  self_prov_flags = 0x00;
```

在初始化完成之后，即可调用 bt_mesh_provision 函数进行自配网。配网成功后，会进入配网完成回调。

```
err = bt_mesh_provision(self_prov_net_key, self_prov_net_idx, self_prov_flags,
self_prov_iv_index, self_prov_addr, self_prov_dev_key);
if(err)
{
    APP_DBG("Self Privisioning (err %d)", err);
    return;
}
```

5.2.2 中心节点配网回调

中心节点常用的配网相关回调类型有五种，其参数定义如下：

```
static const struct bt_mesh_prov app_prov = {
    .uuid = dev_uuid,
    .link_open = link_open,
    .link_close = link_close,
    .complete = prov_complete,
    .unprovisioned_beacon = unprov_recv,
    .node_added = node_added,
};
```

5.2.2.1 prov_complete 回调

配网完成回调，调用自配网 API 后会进入此回调，例程默认在此回调内使能配网者角色功能，随后将自身添加进用户管理的节点结构体数组内，并开始配置自身模型流程（见管理网络）

```
static void prov_complete(uint16_t net_idx, uint16_t addr, uint8_t flags, uint32_t iv_index)
{
    int     err;
    node_t *node;

    APP_DBG("");

    err = bt_mesh_provisioner_enable(BLE_MESH_PROV_ADV);
    if(err)
    {
```

```
        APP_DBG("Unabled Enable Provisoner (err:%d)", err);
    }

    node = node_get(addr);
    if(!node || !node->fixed)
    {
        node = node_cfg_process(node, net_idx, addr, ARRAY_SIZE(elements));
        if(!node)
        {
            APP_DBG("Unable allocate node object");
            return;
        }

        node->cb = &local_cfg_cb;
        local_stage_set(node, LOCAL_APPKEY_ADD);
    }
}
```

5.2.2.2 unprov_recv 回调

收到未配网广播回调：配网者在使能配网功能后，会解析未配网节点发送的广播，并通过 unprov_recv 回调上报应用层。例程默认会在此回调中发送配网请求。

```
static void unprov_recv(bt_mesh_prov_bearer_t bearer, const uint8_t uuid[16], bt_mesh_prov_oob_info_t oob_info, const unprivison_info_t *info)
{
    APP_DBG("");
    int err;

    if(bearer & BLE_MESH_PROV_ADV)
    {
        err = bt_mesh_provision_adv(uuid, self_prov_net_idx, BLE_MESH_ADDR_UNASSIGNED, 5);
        if(err)
        {
            APP_DBG("Unable Open PB-ADV Session (err:%d)", err);
        }
    }
}
```

回调的参数里包含未配网设备的信息，通常只关心 16 字节 uuid，通过 uuid 判断该设备是否为需要配网的设备，随后调用 bt_mesh_provision_adv 向未配网设备发送配网请求，进入配网流程。

```
int bt_mesh_provision_adv( const uint8_t uuid[16], uint16_t net_idx, uint16_t addr, uint8_t attention_duration );
```

此函数需要填入需要配网的设备的 uuid，想要配网所属的网络密钥序号，指定未配网设备入网后的网络地址，以及未配网设备配网时引起注意的持续时长。

指定网络地址可以由用户自行分配，如果分配的地址不合法或者已被占用，则会返回

失败；也可以如上述代码中一样填入宏定义 BLE_MESH_ADDR_UNASSIGNED，协议栈会自动分配符合要求的网络地址。在之后的流程中可以获取到配网成功设备的网络地址。

引起注意的持续时长对应 5.1.2.5 HEALTH 模型回调章节中的注意回调超时。

5.2.2.3 link_open 回调

当调用发起配网请求函数后，如果正确接收到未配网设备的应答，将会进入此回调，表示当前 link 建立完毕，开始进行配网数据交互。

```
1.  static void link_open(bt_mesh_prov_bearer_t bearer)
2.  {
3.      APP_DBG("");
4.  }
```

5.2.2.4 link_close 回调

当 link 连接关闭时会进入 link_close 回调，同时通过参数获取关闭的原因。出现以下情况时会关闭 link：配网成功、储存空间不足、配网数据交互错误、接收超时（60s）、用户主动取消。通常当回调参数为 CLOSE_REASON_SUCCESS 时，代表配网成功，随后会进入 node_add 回调。

```
1.  static void link_close(bt_mesh_prov_bearer_t bearer, uint8_t reason)
2.  {
3.      APP_DBG("reason %x", reason);
4.      if(reason == CLOSE_REASON_RESOURCES)
5.      {
6.          // 存储的节点已满，可选择 停止发起配网 或 清除全部节点 或 按地址清除节点(注意应用层管理的节点也需要对应清除)
7.          bt_mesh_provisioner_disable(BLE_MESH_PROV_ADV, TRUE);
8.          //bt_mesh_node_clear();node_init();
9.          //bt_mesh_node_del_by_addr(app_nodes[1].node_addr);
10.     }
11.     else
12.     {
13.         bt_mesh_provisioner_enable(BLE_MESH_PROV_ADV);
14.     }
15. }
```

5.2.2.5 node_added 回调

当把一个新设备配网完成后，协议栈会保存该节点的信息，并将其存入 NVS 中（如果使能了 NVS 功能），随后会通过此回调告知应用层节点已添加。回调参数包含该节点所属的网络序号、节点网络地址、以及节点拥有的元素数量。

```
1.  static void node_added(uint16_t net_idx, uint16_t addr, uint8_t num_elem)
2.  {
3.      node_t *node;
4.      APP_DBG("");
5.
6.      node = node_get(addr);
7.      if(!node || !node->fixed)
8.      {
```

```
9.          node = node_cfg_process(node, net_idx, addr, num_elem);
10.         if(!node)
11.         {
12.             APP_DBG("Unable allocate node object");
13.             return;
14.         }
15.
16.         node->cb = &node_cfg_cb;
17.         node_stage_set(node, NODE_APPKEY_ADD);
18.     }
19. }
```

例程默认会在此回调中将节点添加到用户管理的结构体数组中，然后开始远端节点配置流程。

6. 管理网络

配网者角色通过 CONFIG 客户端模型向网络中的节点发送配置命令，所有的配置命令码可以在协议栈头文件的 BLE_MESH_MODEL_OP 定义中找到。配网者角色调用的配置命令 API 可以在协议栈头文件的 cfg_cli 相关函数声明中找到，这里只介绍常用的几种命令。

6.1 通过配网者远程管理网络

6.1.1 添加应用密钥

新入网的设备只有 CONFIG 模型可以通信，其他模型则必需绑定应用密钥才可以使用。应用密钥一般由配网者下发给设备。

```
1. int bt_mesh_cfg_app_key_add( uint16_t net_idx, uint16_t addr, uint16_t key_net_
   idx, uint16_t key_app_idx, const uint8_t app_key[16] );
```

前两个参数分别为需要配置的设备所属的网络序号以及网络地址，后面三个参数分别为应用密钥所属的网络序号，应用密钥序号，以及应用密钥。

在 MESH 网络里，同一个设备为了方便管理 16 字节的密钥，通常会使用序号来映射密钥，比如网络密钥与网络序号一一对应，应用密钥与应用密钥序号一一对应。

6.1.2 绑定应用密钥到指定模型

当收到添加应用密钥的命令后，设备会保存收到的应用密钥，还需要将应用密钥绑定到模型上，该模型才可以通信。

设计不同模型的命令通常会分为两种，一种为配置 SIG 标准模型，另一种为配置厂商自定义模型，区别为参数多了一个厂商 ID。

```
1. int bt_mesh_cfg_mod_app_bind( uint16_t net_idx, uint16_t addr, uint16_t elem_ad
   dr, uint16_t mod_app_idx, uint16_t mod_id );
2.
3. int bt_mesh_cfg_mod_app_bind_vnd( uint16_t net_idx, uint16_t addr, uint16_t ele
   m_addr, uint16_t mod_app_idx, uint16_t mod_id, uint16_t cid );
```

同样，前两个参数分别为需要配置的设备所属的网络序号以及网络地址，后面三个参数分别为模型所属的元素地址（通常与网络地址相等），模型需要绑定的应用密钥序号，模型 ID（以及厂商 ID）。

例程默认每个设备可以保存三个不同的应用密钥，每个模型可以绑定一个密钥，模型之间必需绑定相同的密钥才可以通信。

6.1.3 添加订阅地址到指定模型

订阅地址可用于分组管理，当给多个模型添加相同的订阅地址后，如果向此订阅地址发送消息，则所有添加此订阅地址的模型都将收到消息。

此命令也分为两种，一种为配置 SIG 标准模型，另一种为配置厂商自定义模型，区别为参数多了一个厂商 ID。参数与绑定应用密钥到指定模型命令一样，只是把应用密钥序号改成要添加的订阅地址。

```
1. int bt_mesh_cfg_mod_sub_add( uint16_t net_idx, uint16_t addr, uint16_t elem_add
   r, uint16_t sub_addr, uint16_t mod_id );
2.
3. int bt_mesh_cfg_mod_sub_add_vnd( uint16_t net_idx, uint16_t addr, uint16_t elem
   _addr, uint16_t sub_addr, uint16_t mod_id, uint16_t cid );
```

订阅地址可使用的范围为 0xC000~0xFEFF。除此之外还有发送消息时可使用的四种特殊地址：0xFFFC-所有代理节点、0xFFFD-所有朋友节点、0xFFFE-所有使能中继功能节点、0xFFFF-所有节点。

6.1.4 删除节点命令

配网者可以通过删除节点命令告知设备清除自身配网信息，以将其移出自身网络。

删除节点命令的参数只需要配置的设备所属的网络序号以及网络地址。

```
int bt_mesh_cfg_node_reset( uint16_t net_idx, uint16_t addr );
```

在 CONFIG 客户端模型回调中可以获取到删除节点命令的状态，并且需要再调用清除本地保存的节点信息函数（bt_mesh_node_del_by_addr）删除自身存储的已删除节点的信息。

```
static void cfg_cli_rsp_handler(const cfg_cli_status_t *val)
{
    node_t *node;
    APP_DBG("");

    // 删除节点的应答,由于有可能节点已被删除所以收不到应答，所以不管是否应答一律算成功。注意如果节点未在线则节点自身不会收到删除命令
    if(val->cfgHdr.opcode == OP_NODE_RESET)
    {
        if(reset_node_addr != BLE_MESH_ADDR_UNASSIGNED)
        {
            bt_mesh_node_del_by_addr(reset_node_addr);
            node = node_get(reset_node_addr);
            node->stage.node = NODE_INIT;
            node->node_addr = BLE_MESH_ADDR_UNASSIGNED;
            node->fixed = FALSE;
            APP_DBG("node reset complete");
        }
        return;
    }

    node = node_unblock_get();
    if(!node)
    {
        APP_DBG("Unable find Unblocked Node");
        return;
    }

    if(val->cfgHdr.status == 0xFF)
    {
        APP_DBG("Opcode 0x%04x, timeout", val->cfgHdr.opcode);
        goto end;
    }

    node->cb->rsp(node, val);
```

```
35.
36. end:
37.     tmos_start_task(App_TaskID, APP_NODE_EVT, K_SECONDS(1));
38. }
```

由于当被删除节点收到删除命令并应答后，就会执行删除配网信息流程，随后便无法通信，所以无法保证删除命令的应答可以正确被配网者所接收。所以当配网者未收到应答时，有两种可能，一种是被删除节点不在通信范围内，另一种是被删除节点已被删除，但未能收到应答。为防止出现已删除的节点信息占用配网者储存空间，所以例程里即使未收到应答，也按照成功删除流程处理。

如果用户需要保证删除正确执行，可以自行拟定上层协议，使待删除节点通过调用本地 API 删除自身配网信息。（参考本地管理网络章节）

6.1.5 例程配网者自配网后对自身模型的配置流程

配网者自配网后，会执行下述代码流程，依次向自身需要通信的模型发送两种配置命令：添加应用密钥，以及绑定应用密钥到指定模型。每一个命令都会通过 CONFIG 客户端模型回调（cfg_cli_rsp_handler）上报该配置命令执行的状态，同时只能执行一条命令，只有上一条成功后，才会执行下一条。

```
1.  static BOOL local_stage(void *p1)
2.  {
3.      int     err;
4.      BOOL    ret = FALSE;
5.      node_t *node = p1;
6.
7.      switch(node->stage.local)
8.      {
9.          case LOCAL_APPKEY_ADD:
10.             err = bt_mesh_cfg_app_key_add(node->net_idx, node-
    >node_addr, self_prov_net_idx, self_prov_app_idx, self_prov_app_key);
11.             if(err)
12.             {
13.                 APP_DBG("Unable to adding Application key (err %d)", err);
14.                 ret = 1;
15.             }
16.             break;
17.
18.         case LOCAL_MOD_BIND_SET:
19.             err = bt_mesh_cfg_mod_app_bind_vnd(node->net_idx, node-
    >node_addr, node-
    >node_addr, self_prov_app_idx, BLE_MESH_MODEL_ID_WCH_CLI, CID_WCH);
20.             if(err)
21.             {
22.                 APP_DBG("Unable to Binding vendor Model (err %d)", err);
23.                 ret = 1;
24.             }
25.             break;
```

```
26.
27.         default:
28.             ret = 1;
29.             break;
30.     }
31.
32.     return ret;
33. }
```

例程会在每一次命令执行成功后的 cfg_cli_rsp_handler 回调中，调用此 node_rsp 函数，在此函数中会打印当前命令状态信息，并设置下一次命令。

```
1.  static void local_rsp(void *p1, const void *p2)
2.  {
3.      node_t                 *node = p1;
4.      const cfg_cli_status_t *val = p2;
5.
6.      switch(val->cfgHdr.opcode)
7.      {
8.          case OP_APP_KEY_ADD:
9.              APP_DBG("local Application Key Added");
10.             local_stage_set(node, LOCAL_MOD_BIND_SET);
11.             break;
12.         case OP_MOD_APP_BIND:
13.             APP_DBG("local vendor Model Binded");
14.             local_stage_set(node, LOCAL_CONFIGURATIONED);
15.             break;
16.         default:
17.             APP_DBG("Unknown Opcode (0x%04x)", val->cfgHdr.opcode);
18.             return;
19.     }
20. }
```

6.1.6 例程配网者对新入网设备的配置流程

在配网者将一个设备成功配网后，会执行下述代码的流程，依次向新入网的设备发送以下三种配置命令：添加应用密钥、绑定应用密钥到指定模型、添加订阅地址到指定模型。每一个命令都会通过 CONFIG 客户端模型回调（cfg_cli_rsp_handler）上报该配置命令执行的状态，同时只能执行一条命令，只有上一条成功后，才会执行下一条。

```
1.  static BOOL node_stage(void *p1)
2.  {
3.      int     err;
4.      BOOL    ret = FALSE;
5.      node_t *node = p1;
6.
7.      switch(node->stage.node)
8.      {
9.          case NODE_APPKEY_ADD:
```

```
            err = bt_mesh_cfg_app_key_add(node->net_idx, node->node_addr, self_prov_net_idx, self_prov_app_idx, self_prov_app_key);
            if(err)
            {
                APP_DBG("Unable to adding Application key (err %d)", err);
                ret = TRUE;
            }
            break;

        case NODE_MOD_BIND_SET:
            err = bt_mesh_cfg_mod_app_bind_vnd(node->net_idx, node->node_addr, node->node_addr, self_prov_app_idx, BLE_MESH_MODEL_ID_WCH_SRV, CID_WCH);
            if(err)
            {
                APP_DBG("Unable to Binding vendor Model (err %d)", err);
                ret = TRUE;
            }
            break;

            // 设置模型订阅
        case NODE_MOD_SUB_SET:
            err = bt_mesh_cfg_mod_sub_add_vnd(node->net_idx, node->node_addr, node->node_addr, vendor_sub_addr, BLE_MESH_MODEL_ID_WCH_SRV, CID_WCH);
            if(err)
            {
                APP_DBG("Unable to Set vendor Model Subscription (err %d)", err);
                ret = TRUE;
            }
            break;

        default:
            ret = TRUE;
            break;
    }

    return ret;
}
```

例程会在每一次命令执行成功后的 cfg_cli_rsp_handler 回调中，调用此 node_rsp 函数，在此函数中会打印当前命令状态信息，并设置下一次命令。

```
static void node_rsp(void *p1, const void *p2)
{
```

```
    node_t                  *node = p1;
    const cfg_cli_status_t *val = p2;

    switch(val->cfgHdr.opcode)
    {
        case OP_APP_KEY_ADD:
            APP_DBG("node Application Key Added");
            node_stage_set(node, NODE_MOD_BIND_SET);
            break;
        case OP_MOD_APP_BIND:
            APP_DBG("node vendor Model Binded");
            node_stage_set(node, NODE_MOD_SUB_SET);
            break;
        case OP_MOD_SUB_ADD:
            APP_DBG("node vendor Model Subscription Set");
            node_stage_set(node, NODE_CONFIGURATIONED);
            break;
        default:
            APP_DBG("Unknown Opcode (0x%04x)", val->cfgHdr.opcode);
            return;
    }
}
```

6.2 本地管理自身网络信息

为了节约配网后配置的时长，提高多设备入网的效率，例程提供一些接口，供用户自行修改本地网络信息，在天猫精灵和普通节点自配网相关例程里可以看到。

6.2.1 添加应用密钥到本地网络信息

协议栈提供了自行添加应用密钥的 API（bt_mesh_app_key_set），用于将应用密钥以及对应的应用密钥序号添加到设备

```
    status = bt_mesh_app_key_set(app_nodes->net_idx, self_prov_app_idx, self_prov_app_key, FALSE);
    if( status )
    {
        APP_DBG("Unable set app key");
    }
```

6.2.2 绑定应用密钥到本地模型

在初始化模型的章节里讲过，每个模型都有其应用密钥数组和订阅地址数组，而绑定的本质上就是将需要操作的应用密钥序号和订阅地址添加到对应数组里，随后调用协议栈 API 将其储存起来。

注意这里的应用密钥序号一定要是已有的，也就是说，只有当应用密钥以及对应的应用密钥序号通过配网者或者自行添加的方式添加到设备后，设备端才可以通过本地的方式将对应应用密钥序号绑定到模型。

```
        // 添加应用密钥到第 2 个标准模型
```

```
        root_models[2].keys[0] = (uint16_t)0x0000;
        bt_mesh_store_mod_bind(&root_models[2]);

        // 添加应用密钥到第 0 个厂商自定义模型
        vnd_models[0].keys[0] = (uint16_t)0x0000;
        bt_mesh_store_mod_bind(&vnd_models[0]);
```

6.2.3 添加订阅地址到本地模型

添加订阅地址的方式与绑定应用密钥相同，订阅地址取值由用户自行指定。

```
        // 添加订阅地址到第 2 个标准模型
        root_models[2].groups[0] = (uint16_t)0xC000;
        root_models[2].groups[1] = (uint16_t)0xCFFF;
        bt_mesh_store_mod_sub(&root_models[2]);

        // 添加订阅地址到第 0 个厂商自定义模型
        vnd_models[0].groups[0] = (uint16_t)0xC000;
        vnd_models[0].groups[1] = (uint16_t)0xCFFF;
        bt_mesh_store_mod_sub(&vnd_models[0]);
```

6.2.4 删除本地配网信息

上文说过，使用配置命令删除节点的方式，可能存在应答收不到的情况，导致配网者无法正确识别设备是否正常删除。

如果需要用到上述使用场景，用户可以通过上层制定协议，自行确认需要执行删除后，配网者直接调用清除本地保存的节点信息函数（bt_mesh_node_del_by_addr）删除自身存储的已删除节点的信息，设备端通过调用此章节的删除本地配网信息函数，即可做到与远程控制删除相同的效果。

```
void bt_mesh_reset( void );
```

调用此 API 后，协议栈处理完流程会调用 prov_reset 回调通知应用层（见 5.1.2.4 prov_reset 回调）。用户可在此回调中进行下一步操作。

例程提供了这种用户自定协议删除节点的流程演示，通过沁恒自定义透传模型发送删除命令、命令应答和通知网络所有节点删除关于被删除节点信息的命令，并启动任务延迟调用删除本地配网信息 API，用户可参考例程演示，添加或删除上层协议。

7. 沁恒自定义透传模型

本章节演示如何使用沁恒厂商自定义透传模型收发数据。

7.1 发送数据

在沁恒自定义透传服务和客户端的源文件里，都能找到透传函数的原型，其定义分别如下：（有应答的传输函数原型相差不大，可直接参考）

```
int vendor_message_srv_send_trans(struct send_param *param, uint8_t *pData, uint16_t len)

int vendor_message_cli_send_trans(struct send_param *param, uint8_t *pData, uint16_t len)
```

7.1.1 发送参数

函数的参数里有数据指针和长度，还有一个发送参数的结构体，结构体的定义可以在源文件同名的头文件中可以找到：

```
struct send_param
{
    uint16_t net_idx;
    uint16_t app_idx;
    uint16_t addr;
    uint8_t  tid;
    uint8_t  trans_cnt;
    int32_t  period;
    int32_t  rand;
    uint8_t  send_ttl;

    void                          *cb_data;
    const struct bt_adv_trans_cb *cb;
};
```

其中 net_idx 和 app_idx 为加密消息所使用的网络密钥序号和应用密钥序号，可以填写网络信息里存储的值，如果不填，则会自动使用已储存的第一个序号。

addr 为此消息发往的设备对应的网络地址，也可以填写订阅地址发给一组设备。

tid 为包序号，便于重传时区分收到重复的包。

tran_cnt 为重传次数,period 为重传间隔，rand 为发送延迟。如果设置了 tran_cnt，或者 rand 不为 0，则不可连续调用此函数，会返回 busy 错误，直到发送完成后才可以继续发送。反之如果这两个参数都为 0，则可以连续调用此函数直到底层缓存已满（默认十包）。

send_ttl 为此包数据在网络中生存次数，即可被转发中继的次数，默认不填则使用配置的默认值。

cb 为发送回调函数，包含了发送开始回调和发送结束回调，cb_data 为此回调函数的的回调参数。例程默认未使用这两个参数，如需要可自行添加。

7.1.2 发送函数返回值

函数的返回值为负值，其定义在协议栈头文件中可以看到：

```
#define NOERR                           (MESH_ERROR_BASE_NUM + 0)
#define EINVAL                          (MESH_ERROR_BASE_NUM + 1)
#define EALREADY                        (MESH_ERROR_BASE_NUM + 2)
#define ESRCH                           (MESH_ERROR_BASE_NUM + 3)
#define EBUSY                           (MESH_ERROR_BASE_NUM + 4)
#define ENOTCONN                        (MESH_ERROR_BASE_NUM + 5)
#define EAGAIN                          (MESH_ERROR_BASE_NUM + 6)
#define ENOBUFS                         (MESH_ERROR_BASE_NUM + 7)
#define ENOENT                          (MESH_ERROR_BASE_NUM + 8)
#define ENOMEM                          (MESH_ERROR_BASE_NUM + 9)
#define EEXIST                          (MESH_ERROR_BASE_NUM + 10)
#define EIO                             (MESH_ERROR_BASE_NUM + 11)
#define EDEADLK                         (MESH_ERROR_BASE_NUM + 12)
#define ESPIPE                          (MESH_ERROR_BASE_NUM + 13)
#define EACCES                          (MESH_ERROR_BASE_NUM + 14)
#define ENXIO                           (MESH_ERROR_BASE_NUM + 15)
#define ENOSPC                          (MESH_ERROR_BASE_NUM + 16)
#define EBADMSG                         (MESH_ERROR_BASE_NUM + 17)
#define E2BIG                           (MESH_ERROR_BASE_NUM + 18)
#define ENOTSUP                         (MESH_ERROR_BASE_NUM + 19)
#define EADDRINUSE                      (MESH_ERROR_BASE_NUM + 20)
#define EMSGSIZE                        (MESH_ERROR_BASE_NUM + 21)
#define ECANCELED                       (MESH_ERROR_BASE_NUM + 22)
#define ETIMEDOUT                       (MESH_ERROR_BASE_NUM + 23)
#define EADDRNOTAVAIL                   (MESH_ERROR_BASE_NUM + 24)
```

这里只介绍常见的几种错误码：EINVAL－参数错误（可能是填写的参数有误，也有可能内部一些参数无效，比如未找到可用的应用密钥等等）；EBUSY-当前已有一个流程正在执行，需要等待上一个流程执行完毕才可以调用；ENOBUFS-发送缓存已满；ENOMEM-内存占用已满，需要扩大给蓝牙库分配的大小。

7.1.3 发送函数使用示例

厂商自定义模型的例程提供了透传通道的发送函数，注意函数的返回值为负值，此处打印的返回值需要转换为负值再参考头文件。

```
static int vendor_model_srv_send(uint16_t addr, uint8_t *pData, uint16_t len)
{
    struct send_param param = {
        .app_idx = vnd_models[0].keys[0], // 此消息使用的 app key，如无特定则使用第 0 个 key
        .addr = addr,           // 此消息发往的目的地地址
        .trans_cnt = 0x01,                  // 此消息的用户层发送次数
        .period = K_MSEC(400),              // 此消息重传的间隔，建议不小于(200+50*TTL)ms，若数据较大则建议加长
        .rand = (0),                        // 此消息发送的随机延迟
        .tid = vendor_srv_tid_get(),        // tid，每个独立消息递增循环，srv 使用 128~191
```

```
        .send_ttl = BLE_MESH_TTL_DEFAULT, // ttl，无特定则使用默认值
    };
//    return vendor_message_srv_indicate(¶m, pData, len);  // 调用自定义模型服务的有应答指示函数发送数据，默认超时 2s
    return vendor_message_srv_send_trans(¶m, pData, len); // 或者调用自定义模型服务的透传函数发送数据，只发送，无应答机制
}
```

发送函数会在按键触发的回调函数里调用。

```
void keyPress(uint8_t keys)
{
    APP_DBG("%d", keys);

    switch(keys)
    {
        default:
        {
            int status;
            uint8_t data[8] = {0, 1, 2, 3, 4, 5, 6, 7};
            // 发往配网者节点
            status = vendor_model_srv_send(0x0001, data, 8);
            if(status)
            {
                APP_DBG("send failed %d", status);
            }
            break;
        }
    }
}
```

7.1.4 发送数据 API

如果用户不想使用例程演示的发送流程，可以直接调用协议栈的发送 API 接口函数：

```
int bt_mesh_model_send( struct bt_mesh_model *model,
    struct bt_mesh_msg_ctx *ctx, struct net_buf_simple *msg,
    const struct bt_mesh_send_cb *cb, void *cb_data );
```

其中 model 指向发送使用的模型，ctx 为消息参数，包含密钥使用信息和目的地发送方式等等（见 7.2.1 沁恒自定义透传模型操作码处理函数），msg 为消息内容，cb 与 cb_data 分别为回调函数及其回调参数。

ctx 为消息参数中有个 send_rel 参数，接收时不会用到，发送数据时如果将此位置 1，则表示此消息走底层分包传输流程，需要收到对方应答才算发送完成，可以在 cb 中的 end 回调中获取此消息成功发送还是超时未收到应答的状态返回。

msg 的组建可以参考下述代码的形式，依次添加操作码和数据内容到结构体中。

```
    NET_BUF_SIMPLE_DEFINE(msg, APP_MAX_TX_SIZE + 8);
    net_buf    buf;

    buf.__buf = tmos_msg_allocate(len + 8);
```

```
    buf.size = len + 4;
    bt_mesh_model_msg_init(&(buf.b),
                           OP_VENDOR_MESSAGE_TRANSPARENT_WRT);
    net_buf_simple_add_mem(&(buf.b), pData, len);
    net_buf_simple_add_mem(&msg, buf.data, buf.len);

    err = bt_mesh_model_send(model, &ctx, &msg, cb, cb_data);
```

7.2 接收数据

当模型收到数据，首先进入初始化模型的操作码对应处理函数，随后会在处理函数中调用回调函数上报给应用层（见 4.2.4 定义模型配置字）。

7.2.1 沁恒自定义透传模型操作码处理函数

通常用户无需关心此处理函数，但如果需要获取除了数据内容等其他信息，或者需要管理删除 tid，则需要修改例程的处理函数。

这里以无应答传输操作码的处理函数为例，在此函数里，会判断数据 tid 序号是否重复，不重复则将数据的来源地址填写到回调参数，把数据内容去掉开头一字节的 tid 后的内容填写到回调参数，最后调用回调通知应用层。

```
static void vendor_message_srv_trans(struct bt_mesh_model *model, struct
      bt_mesh_msg_ctx *ctx, struct net_buf_simple *buf)
{
    vendor_model_srv_status_t vendor_model_srv_status;
    uint8_t                  *pData = buf->data;
    uint16_t                  len = buf->len;

    if(pData[0] != vendor_model_srv->srv_tid.trans_tid)
    {
        vendor_model_srv->srv_tid.trans_tid = pData[0];
        // 开头为 tid
        pData++;
        len--;
        vendor_model_srv_status.vendor_model_srv_Hdr.opcode =
            OP_VENDOR_MESSAGE_TRANSPARENT_MSG;
        vendor_model_srv_status.vendor_model_srv_Hdr.status = 0;
        vendor_model_srv_status.vendor_model_srv_Event.trans.pdata = pData;
        vendor_model_srv_status.vendor_model_srv_Event.trans.len = len;
        vendor_model_srv_status.vendor_model_srv_Event.trans.addr = ctx-
>addr;
        if(vendor_model_srv->handler)
        {
            vendor_model_srv->handler(&vendor_model_srv_status);
        }
    }
}
```

处理函数的参数里，model 指向所属的模型，ctx 为此包数据的参数，buf 包含数据详细内容。

数据包的参数结构体 bt_mesh_msg_ctx 在协议栈头文件可以找到，其定义如下：

```
struct bt_mesh_msg_ctx
{
    /** NetKey Index of the subnet to send the message on. */
    uint16_t net_idx;

    /** AppKey Index to encrypt the message with. */
    uint16_t app_idx;

    /** Remote address. */
    uint16_t addr;

    /** Destination address of a received message. Not used for sending. */
    uint16_t recv_dst;

    /** RSSI of received packet. Not used for sending. */
    int8_t recv_rssi;

    /** Received TTL value. Not used for sending. */
    uint8_t recv_ttl :7;

    /** Force sending reliably by using segment acknowledgement */
    uint8_t send_rel :1;

    /** TTL, or BLE_MESH_TTL_DEFAULT for default TTL. */
    uint8_t send_ttl;
};
```

这里对可能用到的参数进行说明：recv_rssi 为接收数据信号强度，为负值，可以由此知道当前节点距离最近的中继节点的信号质量，值越大越好。recv_ttl 为接收数据剩余生存次数，如果发送的 TTL 为固定值，则可以由此知道数据在网络中传输到此设备经过多少次中继（延迟）。

7.2.2 沁恒自定义透传模型回调

例程里沁恒自定义透传服务和客户端模型的回调的定义分别如下：

```
static void vendor_model_srv_rsp_handler(const vendor_model_srv_status_t
      *val)

static void vendor_model_cli_rsp_handler(const vendor_model_cli_status_t
      *val)
```

回调参数里的结构体两者基本相同，这里只举一例：

```
typedef struct
{
    struct vendor_model_srv_EventHdr vendor_model_srv_Hdr;
```

```
    union vendor_model_srv_Event_t   vendor_model_srv_Event;
} vendor_model_srv_status_t;
```

其中 vendor_model_srv_Hdr 包含了触发回调的操作码以及执行状态：

```
struct vendor_model_srv_EventHdr
{
    uint8_t  status;
    uint32_t opcode;
};
```

vendor_model_srv_Event 分为两种结构体，一种是无应答传输，一种为有应答传输，例程里这两者结构相同，包含收到的数据内容、长度和来源网络地址，只是名称不一样，这里只举一例：

```
struct bt_mesh_vendor_model_srv_trans
{
    uint8_t *pdata;
    uint16_t len;
    uint16_t addr;
};

union vendor_model_srv_Event_t
{
    struct bt_mesh_vendor_model_srv_trans trans;
    struct bt_mesh_vendor_model_write     write;
};
```

8. 低功耗功能与朋友关系

默认节点都会开启中继的功能，而具有中继功能的节点，蓝牙是一直处于开启状态，其功耗是很高的，无法使用小电池长时间使用。如果需要用小电池供电的节点，需要开启低功耗功能。

Mesh 协议提供了一种低功耗节点方案，每一个低功耗节点，都必须能够与一个开启朋友功能的节点双向点对点通信，在低功耗节点入网后，不会立刻进入低功耗状态，而是先与周围的朋友节点建立朋友关系后，才能正常在网络中通信。

当网络中有发往低功耗节点的消息时，往往低功耗节点处于休眠状态无法立即接收，此时由建立朋友关系的朋友节点，代替低功耗节点把消息接收到自己的朋友缓存队列中。当低功耗节点唤醒后，会主动向朋友节点发送 poll 请求，随机朋友节点会把之前缓存的消息依次发送给低功耗节点。等到所有消息发送完成，低功耗节点再次进入下一轮休眠。

当低功耗节点需要发送数据时，与正常节点发送数据的流程相同，只需要醒来调用发送即可。

8.1 朋友节点

例程提供了在普通节点基础上使能了朋友功能的程序，建议开发时直接在此例程上修改，其与普通节点源文件的区别仅添加了朋友关系建立回调，其定义如下：

```
1.  static void friend_state(uint16_t lpn_addr, uint8_t state)
2.  {
3.      if(state == FRIEND_FRIENDSHIP_ESTABLISHED)
4.      {
5.          APP_DBG("friend friendship established, lpn addr 0x%04x", lpn_addr);
6.      }
7.      else if(state == FRIEND_FRIENDSHIP_TERMINATED)
8.      {
9.          APP_DBG("friend friendship terminated, lpn addr 0x%04x", lpn_addr);
10.     }
11.     else
12.     {
13.         APP_DBG("unknow state %x", state);
14.     }
15. }
```

朋友节点与低功耗节点交互的流程都在底层完成，包括缓存数据等，应用层只能获取到朋友关系的建立和断开的状态。

注意：朋友节点每支持一个低功耗节点，都需要分配更多的内存，通过增大 MESH_MEM 数组的大小，保证初始化成功即可。

8.2 低功耗节点

例程提供了在普通节点基础上使能了低功耗功能的程序，建议开发时直接在此例程上修改，低功耗节点添加了朋友关系回调，其定义如下：

```
1.  static void lpn_state(uint8_t state)
2.  {
```

```
    if(state == LPN_FRIENDSHIP_ESTABLISHED)
    {
        APP_DBG("lpn friendship established");
    }
    else if(state == LPN_FRIENDSHIP_TERMINATED)
    {
        APP_DBG("lpn friendship terminated");
    }
    else
    {
        APP_DBG("unknow state %x", state);
    }
}
```

除此之外还多调用了一个使能低功耗功能的 API 接口函数（bt_mesh_lpn_set）。只有当配网成功并且配网者成功下发应用密钥绑定到模型后，才会调用设置低功耗功能的函数，

```
        if(bt_mesh_app_key_find(vnd_models[0].keys[0]) != NULL)
        {
            // 配网后启动 lpn，开始发送建立 friend 请求
#if(CONFIG_BLE_MESH_LOW_POWER)
            bt_mesh_lpn_set(TRUE);
            APP_DBG("Low power enable");
#endif /* LPN */
        }
```

当启动了低功耗功能后，会周期性向周围朋友节点发送建立朋友关系的请求（周期为 CONFIG_MESH_RETRY_TIMEOUT_DEF）。当成功建立朋友节点后，将会进入朋友关系回调，随后定期向朋友节点请求数据（周期为 CONFIG_MESH_LPN_POLLINTERVAL_DEF）。

由于低功耗节点只有在定期唤醒请求数据的时候才能接收到消息，所以相对于普通节点而言，其接收延迟会相对更久，请求数据的间隔越大，接收延迟越大。

同样，低功耗节点的功耗也与请求数据的间隔相关，间隔越大，功耗越低。当使用低功耗功能时，需要评估功耗与接收延迟，选择请求数据间隔的最佳值。

9. 连接手机

由于配网者每连接一个节点，都需要占用自身资源，可以配置的节点数量有限，与 mesh 协议允许的 32767 个节点相比相差许多，所以可以使用手机作为配网者，走 mesh 协议组建更大的网络，也可以使用预先自配网的方式，突破配网器的限制，再用手机通过 ble 协议连接任意节点控制整个网络等等。

9.1 代理配网功能

Mesh 中配网流程可以通过 adv（广播）和 gatt（ble 连接）两种方式完成，第五章说明的通过配网者方式入网是通过 adv 通道，而代理配网则是通过 gatt 方式入网。通常将代理配网功能用于手机 APP 控制设备入网。

例程提供了使能了代理功能（proxy）的程序，并且默认支持一个通用开关模型，可以使用手机 app 连接配网控制。App 可以使用 nRF_mesh，或者其他支持 SIG 标准 mesh 的 APP。

9.2 通过独立 ble 功能连接手机

如果有 ble 开发经验，可选择单独通过 ble 连接手机，自行拟定通信协议，实现手机控制 mesh 设备与管理 mesh 网络。

例程提供了三种已经整合好的 mesh+ble 连接的例程，这里以沁恒自定义透传普通节点为例。两种功能初始化的流程与单独使用时一致，对于应用层来说可以简单认为为两套程序互不干涉，只有在有数据交互的时候，才会同时涉及两边。例程在沁恒自定义透传服务模型收到数据时，会通过 ble 通知的方式(peripheralChar4Notify)，转发给已连接的主机（手机），源文件的模型数据回调函数如下：

```
1.  static void vendor_model_srv_rsp_handler(const vendor_model_srv_status_t *val)

2.  {
3.      if(val->vendor_model_srv_Hdr.status)
4.      {
5.          // 有应答数据传输 超时未收到应答
6.          APP_DBG("Timeout opcode 0x%02x",
               val->vendor_model_srv_Hdr.opcode);
7.          return;
8.      }
9.      if(val->vendor_model_srv_Hdr.opcode == OP_VENDOR_MESSAGE_TRANSPARENT_MSG)
10.     {
11.         // 收到透传数据
12.         APP_DBG("len %d, data 0x%02x from 0x%04x",
              val->vendor_model_srv_Event.trans.len,
13.                 val->vendor_model_srv_Event.trans.pdata[0],
14.                 val->vendor_model_srv_Event.trans.addr);
15.         // 转发给主机(如果已连接)
16.         peripheralChar4Notify(val->vendor_model_srv_Event.trans.pdata, val-
    >vendor_model_srv_Event.trans.len);
17.     }
```

```
    else if(val->vendor_model_srv_Hdr.opcode == OP_VENDOR_MESSAGE_TRANSPARENT_WRT)
    {
        // 收到 write 数据
        APP_DBG("len %d, data 0x%02x from 0x%04x",
          val->vendor_model_srv_Event.write.len,
                val->vendor_model_srv_Event.write.pdata[0],
                val->vendor_model_srv_Event.write.addr);
        // 转发给主机(如果已连接)
        peripheralChar4Notify(val->vendor_model_srv_Event.write.pdata, val->vendor_model_srv_Event.write.len);
    }
    else if(val->vendor_model_srv_Hdr.opcode == OP_VENDOR_MESSAGE_TRANSPARENT_IND)
    {
        // 发送的 indicate 已收到应答
    }
    else
    {
        APP_DBG("Unknow opcode 0x%02x", val->vendor_model_srv_Hdr.opcode);
    }
}
```

同理，手机发往设备的信息，可以解析后转为配网者的控制命令，也可以通过沁恒自定义透传数据模型把数据发给其他节点。

修订记录

版本	时间	修订内容
V1.0	2022/3/16	版本发布
V1.1	2022/7/14	新增配置描述 同步例程更新对应描述

版本声明与免责声明