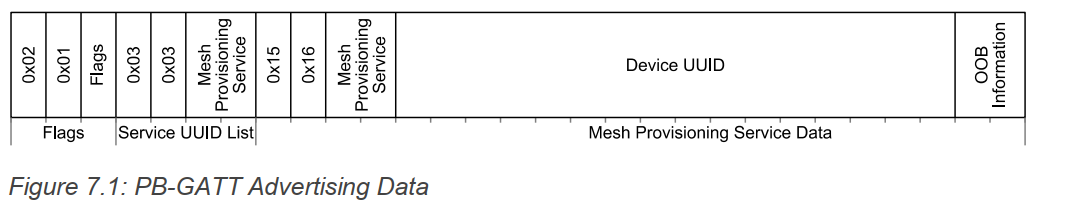
1. 节点需要交叉广播Unprovisioned Device Beacon广播包和PB-GATT的广播包。设备上电后，PB-GATT的广播包每次广播时长是40ms，广播间隔是1S。

PB-GATT 广播数据格式如下：

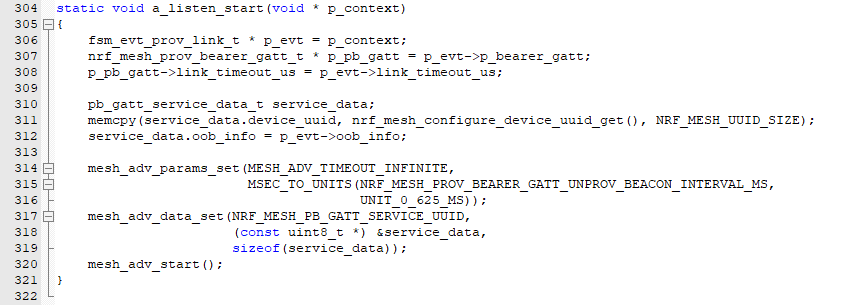


（Mesh Profile 1.0 - 7.1.2.2.1 Advertising）

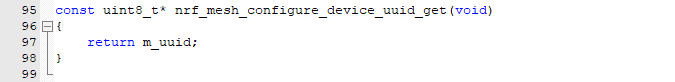
表1 Mesh Beacon⼴播包各个字段含义

|  |  |  |
| --- | --- | --- |
| Field | Size(Octets) | Notes |
| Flag | 1 | 长度，  02代表后面还有2个字节 |
| 1 | 类型，  0x01表示类型是Flags(Assigned Number) |
| 1 | Flags的值，0x02表示LE General Discoverable Mode |
| Service UUID List | 1 | 长度，0x03表示后面还有3个字节 |
| 1 | 类型，0x03表示 Complete List of 16-bit Service Class UUIDs |
| 2 | Mesh Provisioning Service, 0x1827 |
|  | 1 | 长度，0x15表示后面还有21字节 |
| 1 | 类型， 0x16表示Service Data – 16-bit UUID |
| 2 | Mesh Provisioning Service, 0x1827 |
| Device UUID | 16 | 设备UUID,  参考Unprovisioned Device beacon 的广播包中的UUID字段 |
| OOB Information | 2 | bit0 Other  bit1 Electronic / URI  bit2 2D machine-readable code  bit3 Bar code  bit4 Near Field Communication (NFC)  bit5 Number  bit6 String  bit7 Reserved f or Future Use  bit8 Reserved f or Future Use  bit9 Reserved f or Future Use  bit10 Reserved f or Future Use  bit11 On box  bit12 Inside box  bit13 On piece of paper  bit14 Inside manual  bit15 On device  Def ault：0x0000 |

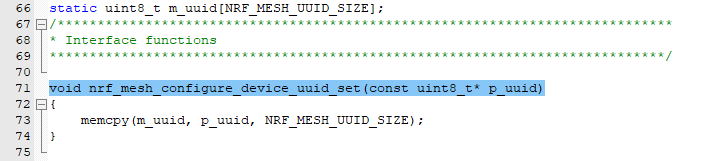
在Nordic Mesh 中，在nrf\_mesh\_prov\_bearer\_gatt.c文件的函数static void a\_listen\_start(void \* p\_context)中进行 Mesh Beacon 数据的填充：



如上图，Beacon数据中的device\_uuid在311行调用nrf\_mesh\_configure\_device\_uuid\_get()获取然后进行填充，



device\_uuid 是通过nrf\_mesh\_configure.c 的void nrf\_mesh\_configure\_device\_uuid\_set(const uint8\_t\* p\_uuid)函数进行预配置：



Beacon数据中oob\_info通过上下文指针进行填充，如312行代码所示，

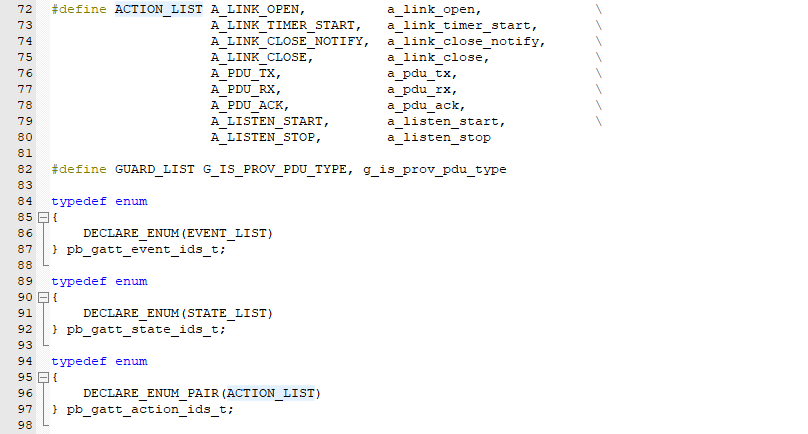
最后将填充的buff（service\_data）通过void mesh\_adv\_data\_set(uint16\_t service\_uuid, const uint8\_t \* p\_service\_data, uint8\_t length);如第317行所示。

参数中的NRF\_MESH\_PB\_GATT\_SERVICE\_UUID = 0x1827 对应上面中的Mesh Provisioning Service；

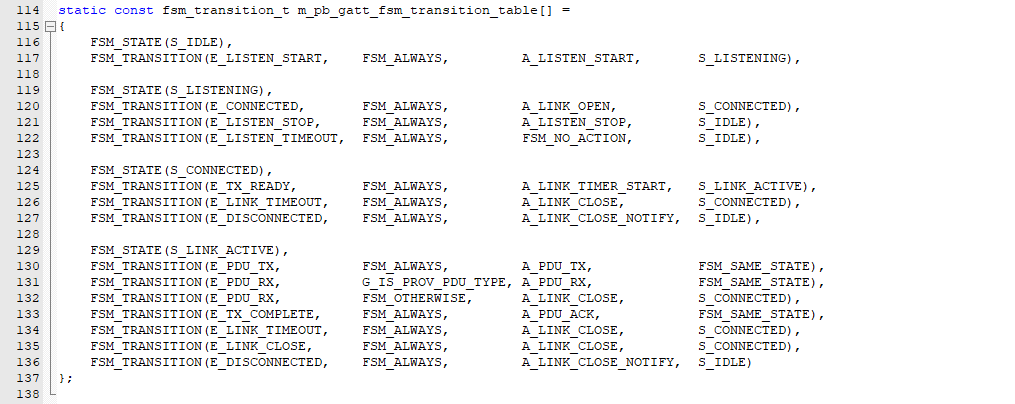
PB-ADV广播包数据初始化过程探究：

PB-ADV 广播包数据在nrf\_mesh\_prov\_bearer\_gatt.c文件的static void a\_listen\_start(void \* p\_context)函数中填充，但是何时被调用呢？经研究：

1. 在nrf\_mesh\_prov\_bearer\_gatt.c文件中，如下图所示：

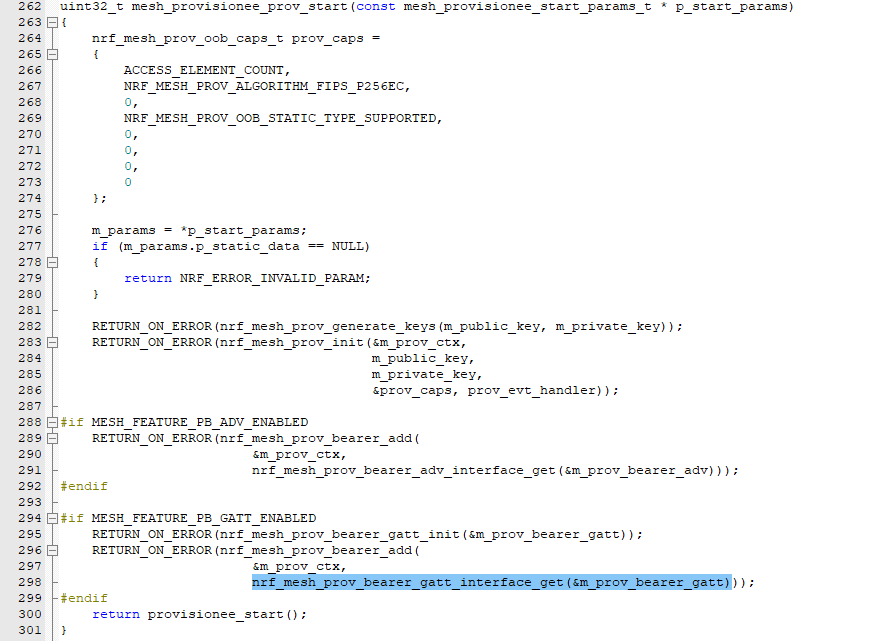


1. 通过上图第72行宏定义和97行枚举结构体，广播数据包的填充函数a\_listen\_start 和 A\_LISTEN\_START形成枚举对，形成绑定关系。



在上图中，A\_LISTEN\_START和E\_LISTEN\_START形成绑定关系。E\_LISTEN\_START为事件对象。

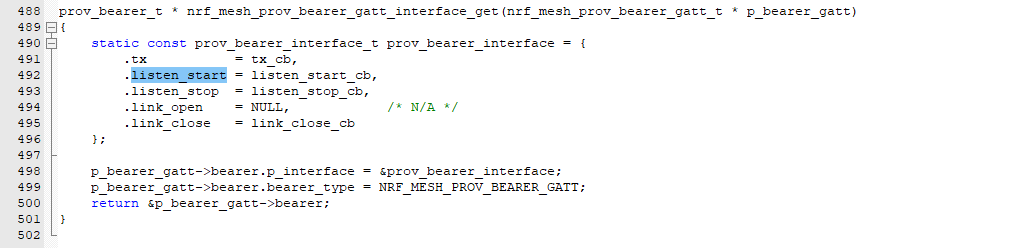
1. 在mesh\_provisionee.c文件中，uint32\_t mesh\_provisionee\_prov\_start(const mesh\_provisionee\_start\_params\_t \* p\_start\_params)函数调用了nrf\_mesh\_prov\_bearer\_gatt\_interface\_get(&m\_prov\_bearer\_gatt)



1. 在nrf\_mesh\_prov\_bearer\_gatt.c文件中，prov\_bearer\_t \* nrf\_mesh\_prov\_bearer\_gatt\_interface\_get(nrf\_mesh\_prov\_bearer\_gatt\_t \* p\_bearer\_gatt)函数把

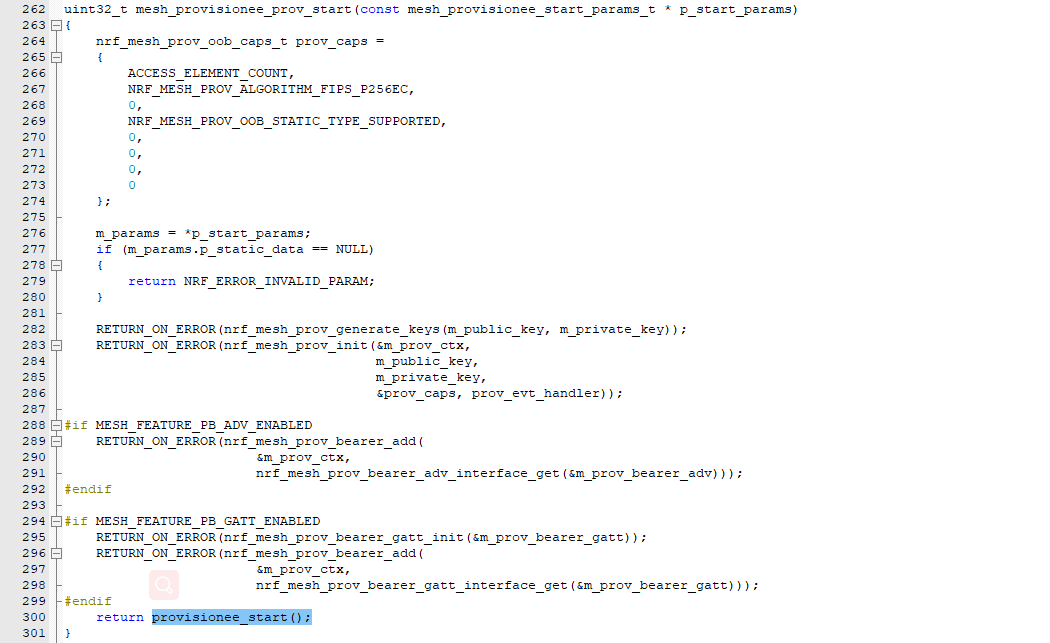
static uint32\_t listen\_start\_cb(prov\_bearer\_t \* p\_bearer, const char \* p\_uri, uint16\_t oob\_info, uint32\_t link\_timeout\_us)回调函数赋予回调函数结构体

prov\_bearer\_interface的成员listen\_start：



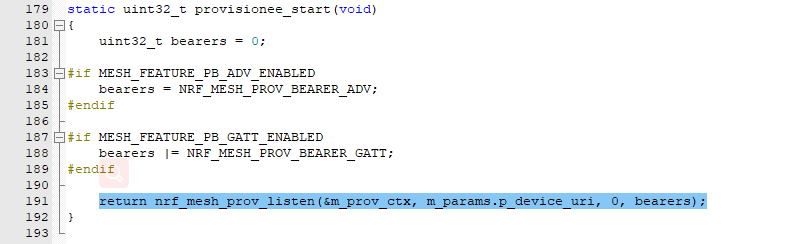
在static uint32\_t listen\_start\_cb(prov\_bearer\_t \* p\_bearer, const char \* p\_uri, uint16\_t oob\_info, uint32\_t link\_timeout\_us)函数中，会传递一个事件，事件的对象为E\_LISTEN\_START，传递成功之后，就会执行上面绑定的a\_listen\_start函数，从而完成对PB-GATT 的Mesh Beacon 广播包的填充。

1. 执行完第④步之后，在mesh\_provisionee.c文件中会执行provisionee\_start();



1. 配置完相关参数后，执行provisionee\_start();

在mesh\_provisionee.c文件中，在static uint32\_t provisionee\_start(void)函数中调用了return nrf\_mesh\_prov\_listen(&m\_prov\_ctx, m\_params.p\_device\_uri, 0, bearers);



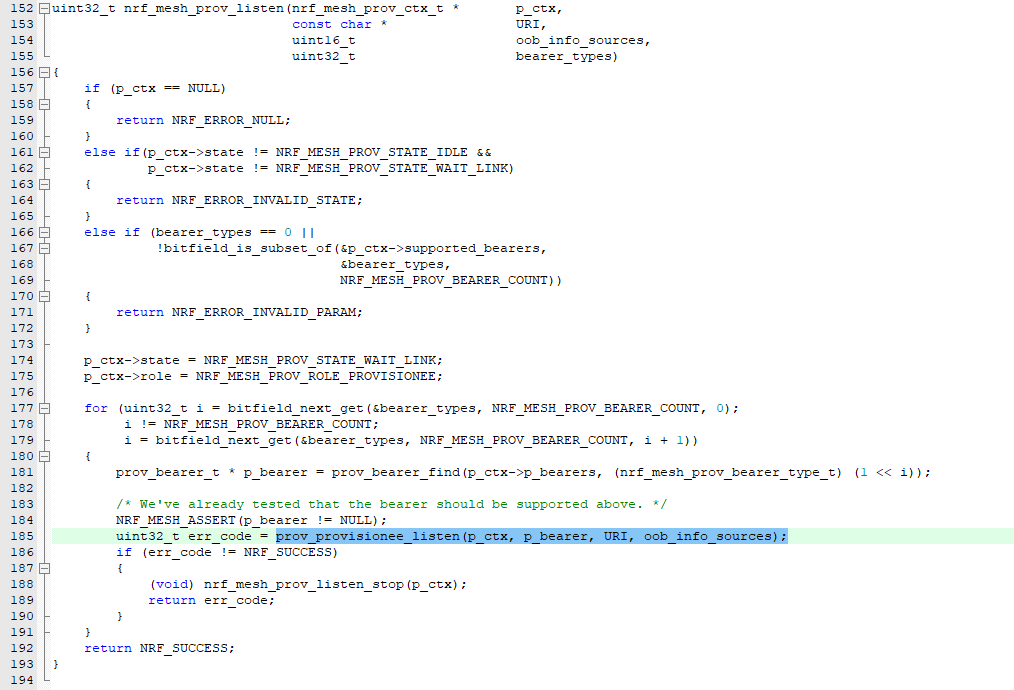
1. 在nrf\_mesh\_prov.c文件中，uint32\_t nrf\_mesh\_prov\_listen(nrf\_mesh\_prov\_ctx\_t \* p\_ctx,

const char \* URI,

uint16\_t oob\_info\_sources,

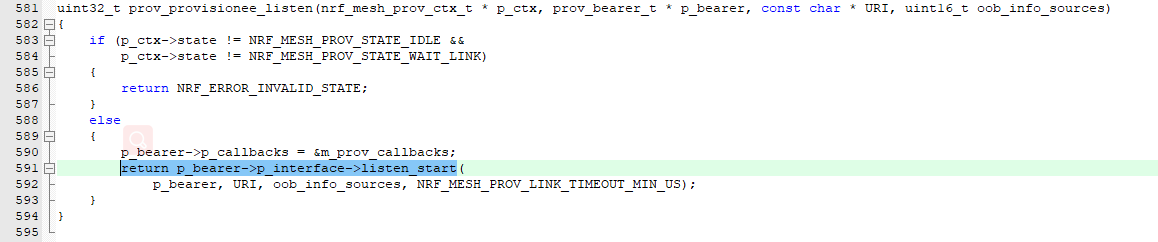
uint32\_t bearer\_types)

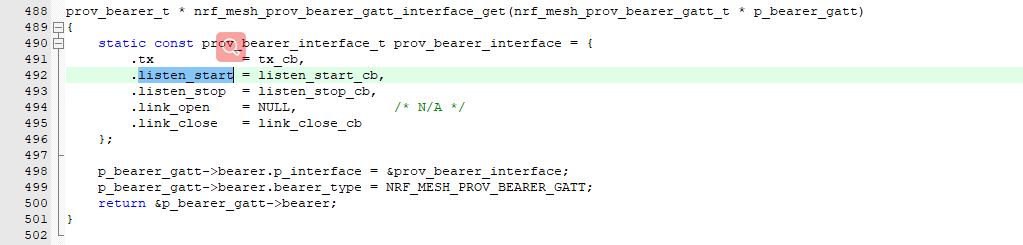
调用了prov\_provisionee\_listen(p\_ctx, p\_bearer, URI, oob\_info\_sources);



1. 在prov\_provisionee.c 文件中，

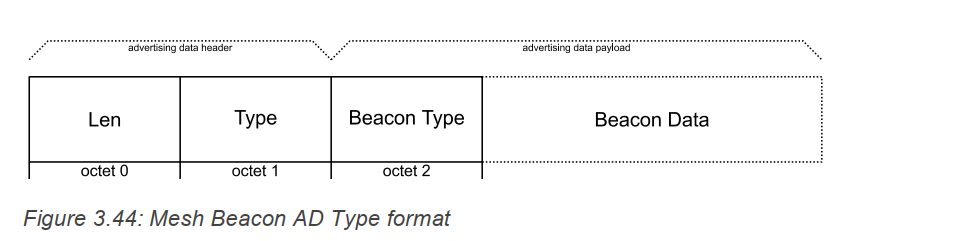
uint32\_t prov\_provisionee\_listen(nrf\_mesh\_prov\_ctx\_t \* p\_ctx, prov\_bearer\_t \* p\_bearer, const char \* URI, uint16\_t oob\_info\_sources)调用了之前④设置的回调函数：





最后开始执行listen\_start\_cb函数并发送E\_LISTEN\_START事件，从而执行a\_listen\_start函数，完成PB-GATT的Mesh Beacon广播包的数据填充并进行广播。

PB-ADV 的Mesh Beacon 广播包格式如下：



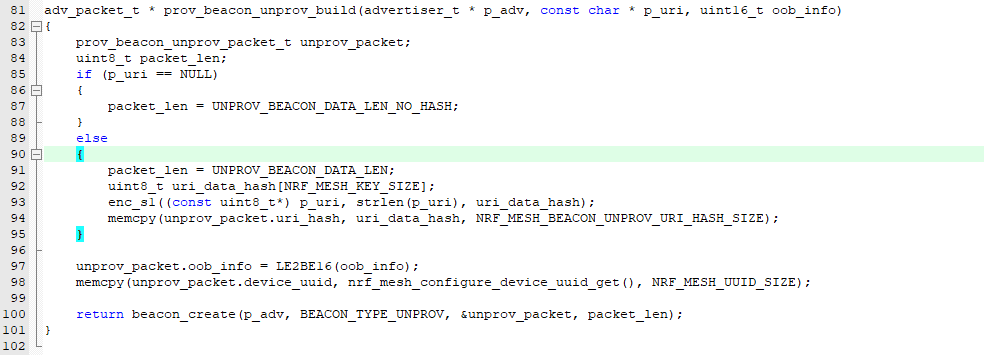
Mesh Beacon 广播包各个字段含义：

|  |  |  |
| --- | --- | --- |
| Field | Size(Octets) | Notes |
| Len | 1 | 长度 |
| Type | 1 | 0x2B |
| Beacon Type | 1 | 0x00 |
| Device UUID | 16 | 设备UUID |
| OOB Information | 2 | bit0 Other  bit1 Electronic / URI  bit2 2D machine-readable code  bit3 Bar code  bit4 Near Field Communication (NFC)  bit5 Number  bit6 String  bit7 Reserved f or Future Use  bit8 Reserved f or Future Use  bit9 Reserved f or Future Use  bit10 Reserved f or Future Use  bit11 On box  bit12 Inside box  bit13 On piece of paper  bit14 Inside manual  bit15 On device  Def ault:0x0000 |
| URI Hash | 4 | 网址（Optional） |

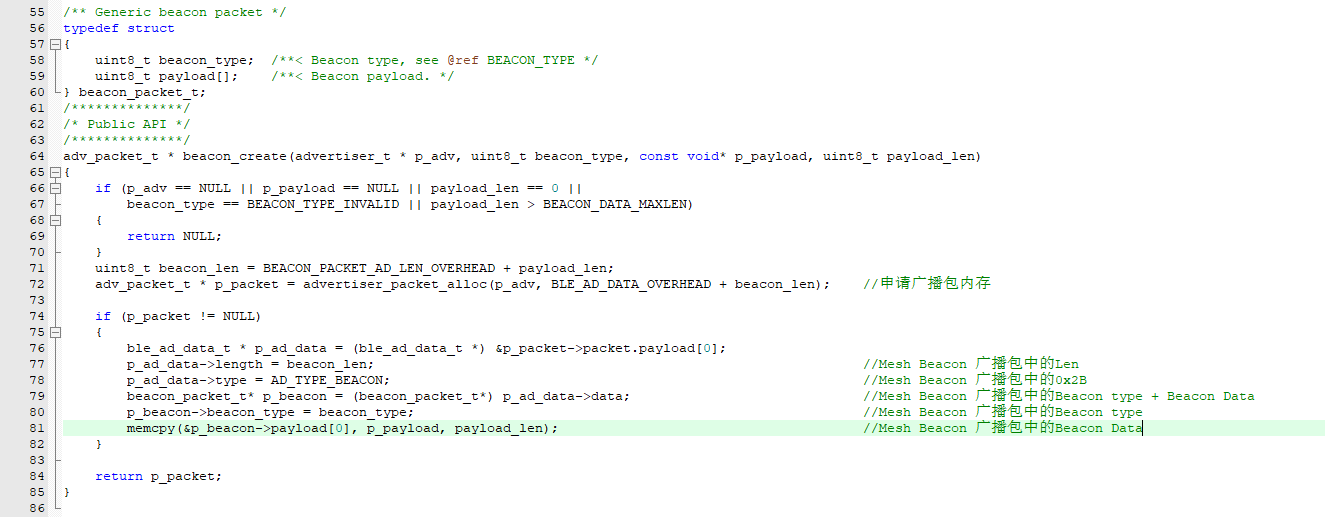
Device UUID 格式

|  |  |  |
| --- | --- | --- |
| Field | Size(Octets) | Notes |
| CID | 2 | 公司ID,设置为0x01A8: Taobao |
| PID | 1 | Bit0-3蓝⽛⼴播包版本号，  ⽬前是0x01  bit4 0：⼀型⼀密  1：⼀机⼀密，  默认是⼀机⼀密  bit5 是否⽀持OTA，  默认是⽀持  bit6-7 00 BLE4.0  01 BLE4.2  10 BLE5.0  11 BLE5.0以上  默认是01 |
| ProductID | 4 | 阿里巴巴平台颁发，一型一号 |
| MAC地址 | 6 | 阿里巴巴平台颁发，一机一号 |
| RFU | 3 | Reserved for future use |

在Nordic Mesh SDK中，在prov\_beacon.c 文件的adv\_packet\_t \* prov\_beacon\_unprov\_build(advertiser\_t \* p\_adv, const char \* p\_uri, uint16\_t oob\_info)对上述数据包进行填充：



第87 行和第91行根据情况填充了Len，第97行填充了oob\_information, 第98行填充了Device\_uuid, 最后调用beacon\_create函数进行整合，参数BEACON\_TYPE\_UNPROV为Type。

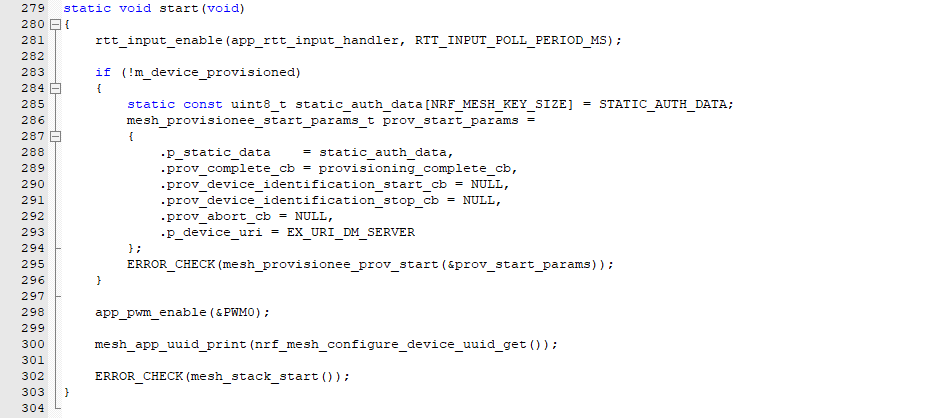


最后广播包填充如上图。

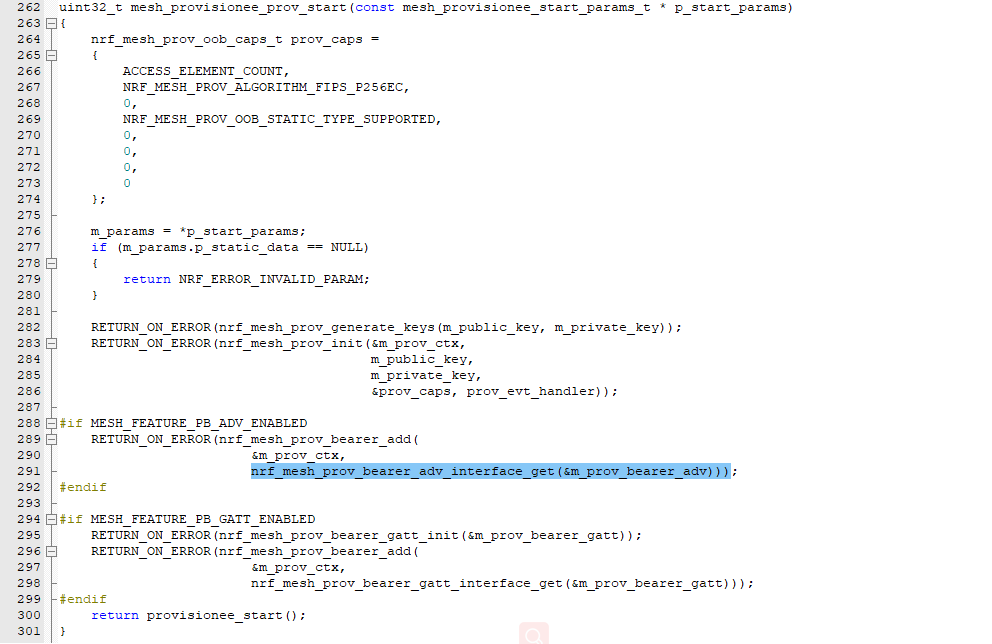
PB-ADV广播包数据初始化过程探究：

在Nordic Mesh 例程中：

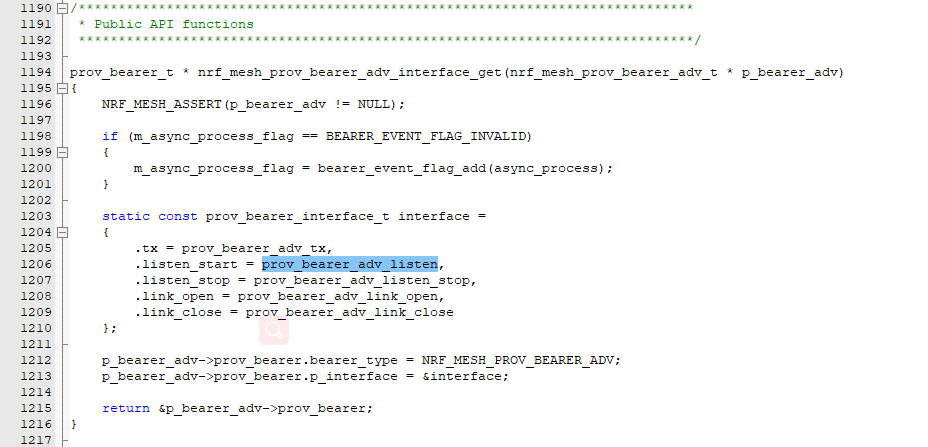
1. 在main.c 中，在static void start(void) 函数中，调用了mesh\_provisionee\_prov\_start(&prov\_start\_params)。



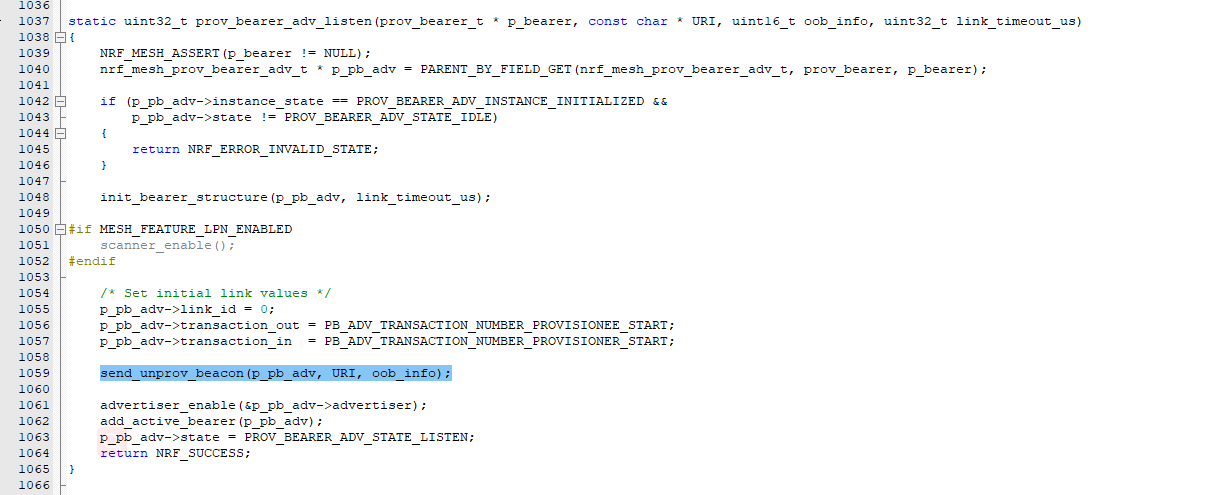
②在nrf\_mesh\_prov\_bearer\_gatt.c中，uint32\_t mesh\_provisionee\_prov\_start(const mesh\_provisionee\_start\_params\_t \* p\_start\_params)函数中调用了nrf\_mesh\_prov\_bearer\_adv\_interface\_get(&m\_prov\_bearer\_adv)))。



1. 在prov\_bearer\_adv.c 文件中，prov\_bearer\_t \* nrf\_mesh\_prov\_bearer\_adv\_interface\_get(nrf\_mesh\_prov\_bearer\_adv\_t \* p\_bearer\_adv)设置了prov\_bearer\_adv\_listen作为回调函数。

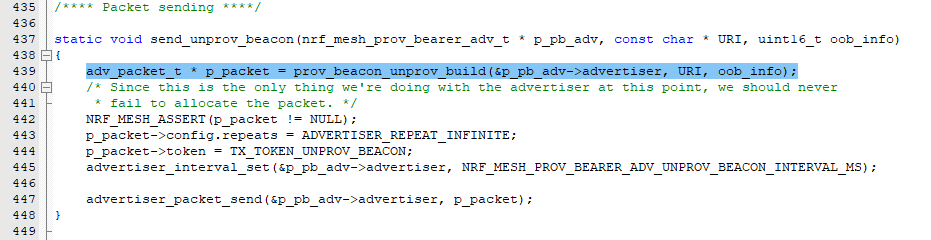


④在prov\_bearer\_adv.c文件中，static uint32\_t prov\_bearer\_adv\_listen(prov\_bearer\_t \* p\_bearer, const char \* URI, uint16\_t oob\_info, uint32\_t link\_timeout\_us)函数调用了send\_unprov\_beacon(p\_pb\_adv, URI, oob\_info);

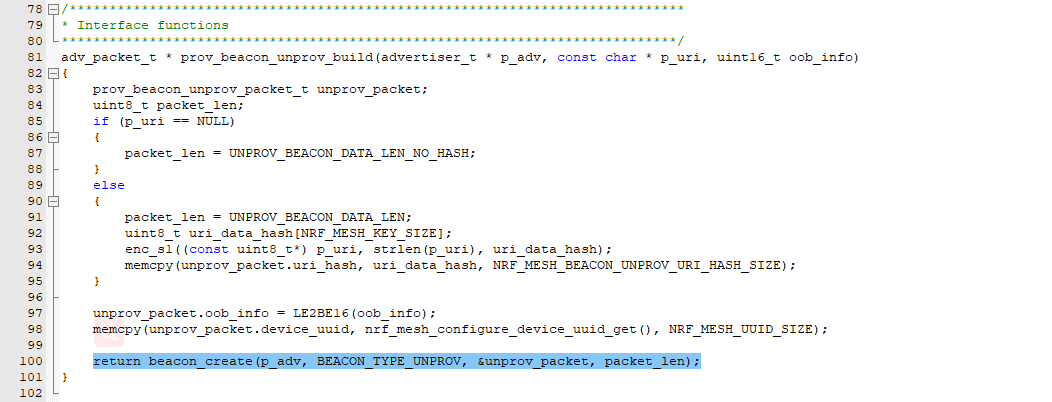


⑤在prov\_bearer\_adv.c文件中，static void send\_unprov\_beacon(nrf\_mesh\_prov\_bearer\_adv\_t \* p\_pb\_adv, const char \* URI, uint16\_t oob\_info)函数调用了

adv\_packet\_t \* p\_packet = prov\_beacon\_unprov\_build(&p\_pb\_adv->advertiser, URI, oob\_info);



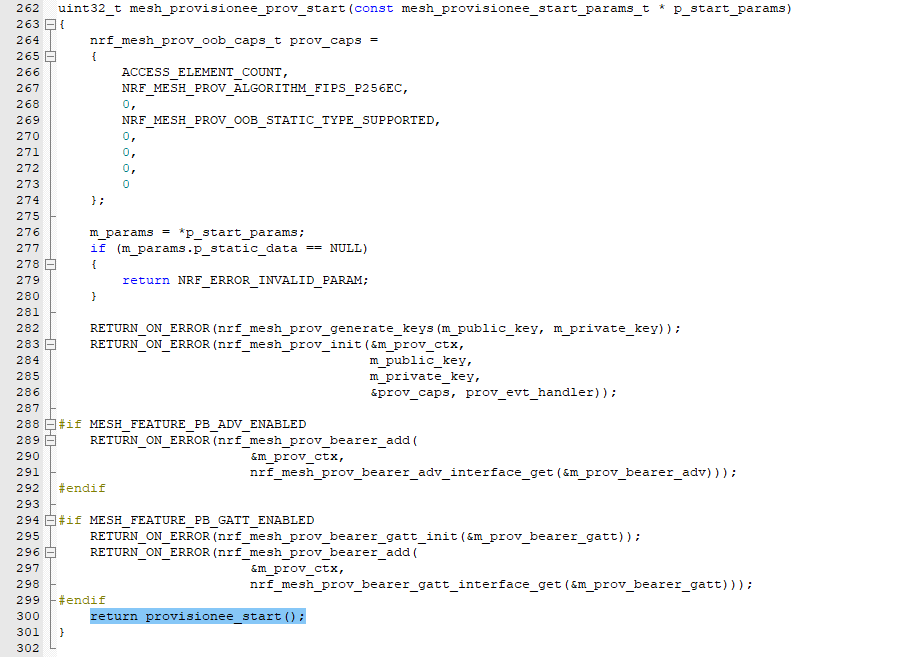
⑥在prov\_beacon.c文件中，adv\_packet\_t \* prov\_beacon\_unprov\_build(advertiser\_t \* p\_adv, const char \* p\_uri, uint16\_t oob\_info)调用了 return beacon\_create(p\_adv, BEACON\_TYPE\_UNPROV, &unprov\_packet, packet\_len);



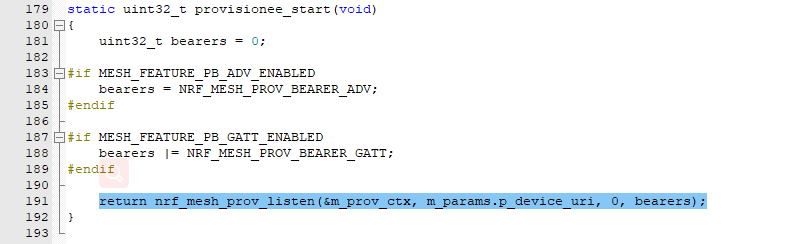
⑦在beacon\_create(p\_adv, BEACON\_TYPE\_UNPROV, &unprov\_packet, packet\_len);中进行PB-ADV数据包的填充。

在以上配置过程中，在②中，只是设置了回调函数，以下探究调用过程：

Ⅰ在上面步骤中的②中，配置完相关参数后，执行provisionee\_start();



Ⅱ在mesh\_provisionee.c文件中，在static uint32\_t provisionee\_start(void)函数中调用了return nrf\_mesh\_prov\_listen(&m\_prov\_ctx, m\_params.p\_device\_uri, 0, bearers);



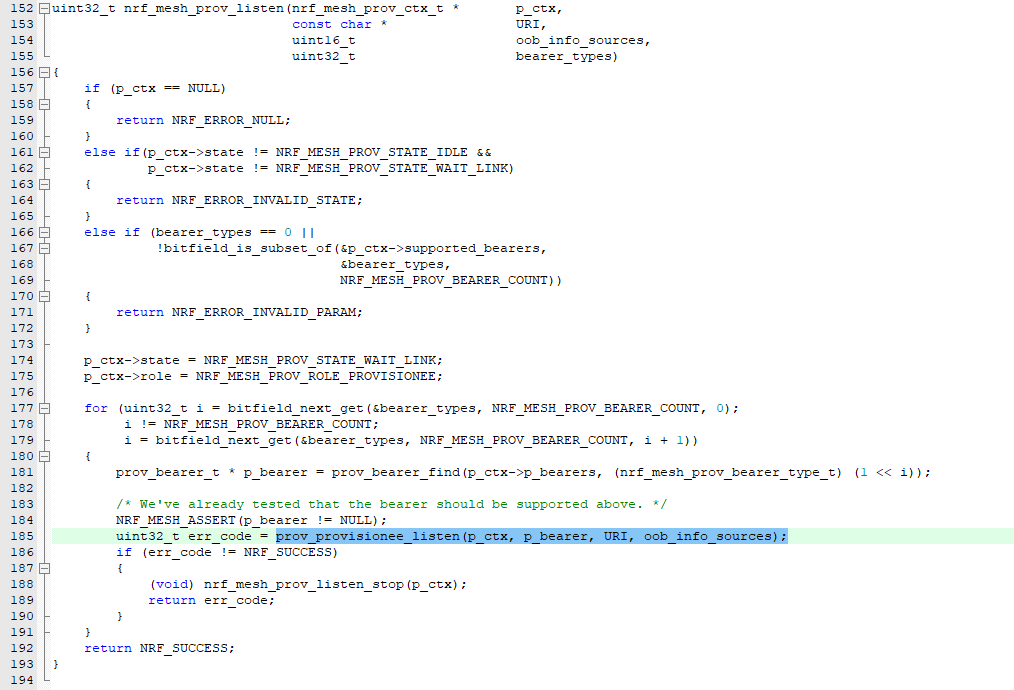
Ⅲ 在nrf\_mesh\_prov.c文件中，uint32\_t nrf\_mesh\_prov\_listen(nrf\_mesh\_prov\_ctx\_t \* p\_ctx,

const char \* URI,

uint16\_t oob\_info\_sources,

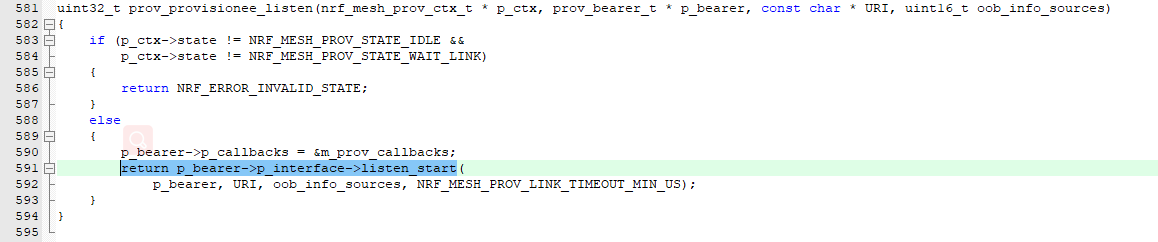
uint32\_t bearer\_types)

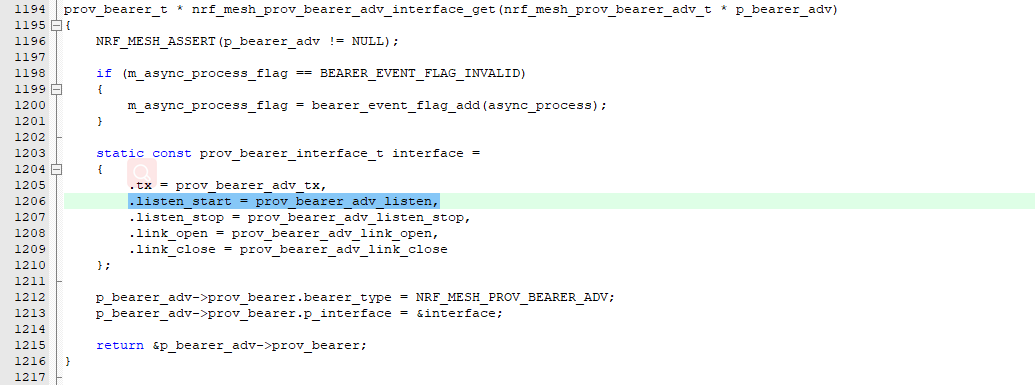
调用了prov\_provisionee\_listen(p\_ctx, p\_bearer, URI, oob\_info\_sources);



Ⅳ 在prov\_provisionee.c 文件中，

uint32\_t prov\_provisionee\_listen(nrf\_mesh\_prov\_ctx\_t \* p\_ctx, prov\_bearer\_t \* p\_bearer, const char \* URI, uint16\_t oob\_info\_sources)调用了之前③设置的回调函数：

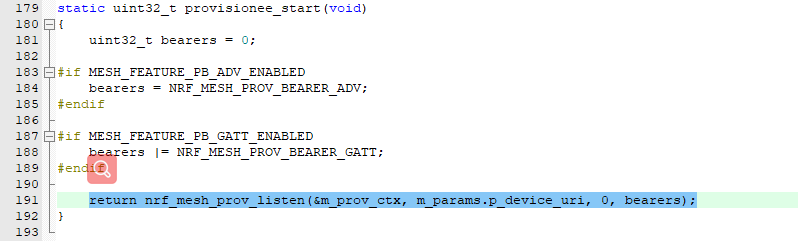




最后开始执行③以后的步骤。

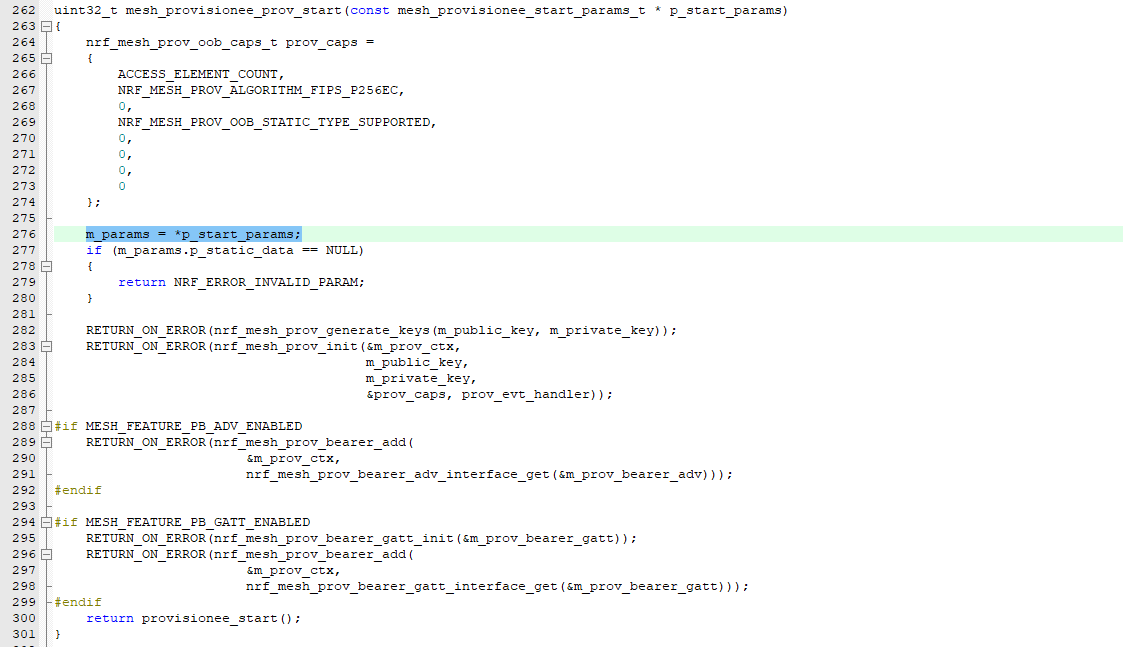
相关参数的配置：

在mesh\_provisionee.c 文件中，由static uint32\_t provisionee\_start(void)函数启动Mesh beacon 广播数据的填充并进行广播。

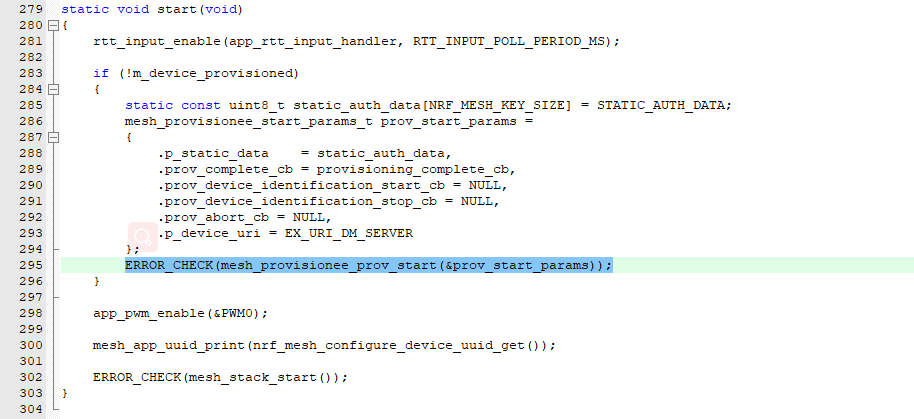
****

URI参数的配置：

在上图中可看出，URI参数由m\_params结构体的p\_device\_uri成员决定，m\_params在mesh\_provisionee.c文件的uint32\_t mesh\_provisionee\_prov\_start(const mesh\_provisionee\_start\_params\_t \* p\_start\_params)的函数进行赋值：



上面的函数在main.c文件的static void start(void)的函数进行调用：

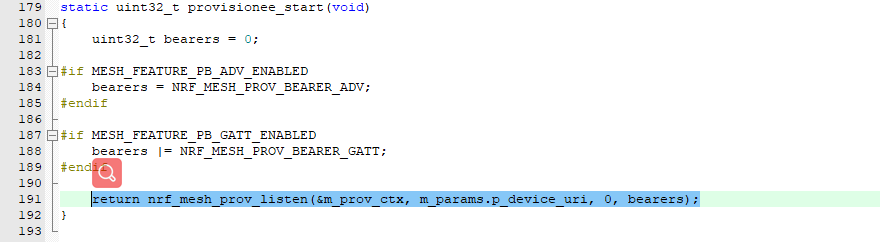


结构体prov\_start\_params就是后面要用的m\_params的原型，其成员p\_device\_uri就是URI参数。

OOB Information:

OOB 信息在mesh\_provisionee.c文件中的static uint32\_t provisionee\_start(void)

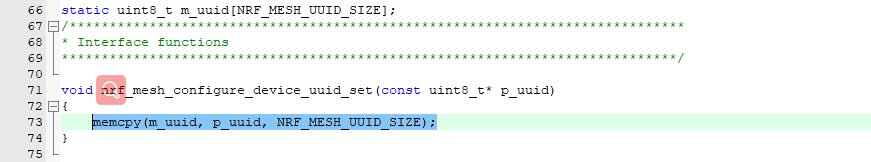
进行调用：

****

目前设置为0，应灵活配置。

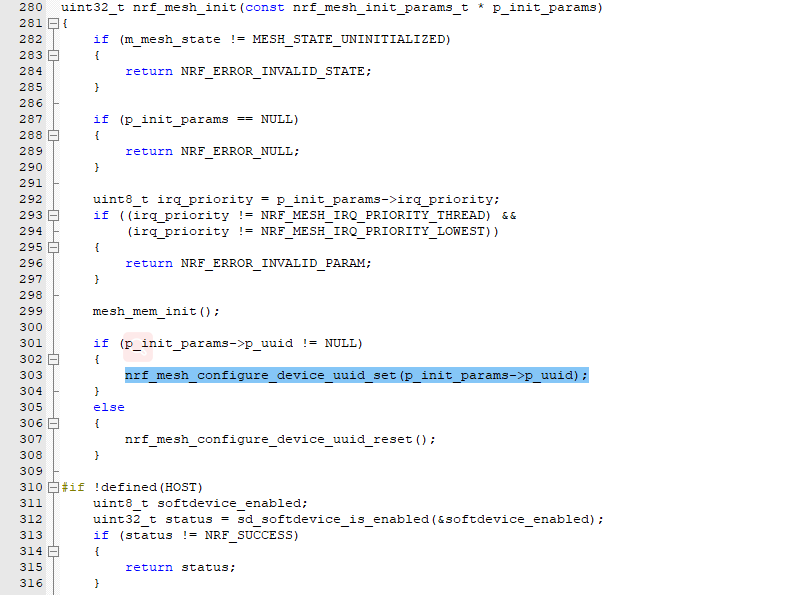
Device\_uuid配置：

通过nrf\_mesh\_configure.c文件中的void nrf\_mesh\_configure\_device\_uuid\_set(const uint8\_t\* p\_uuid)进行配置：

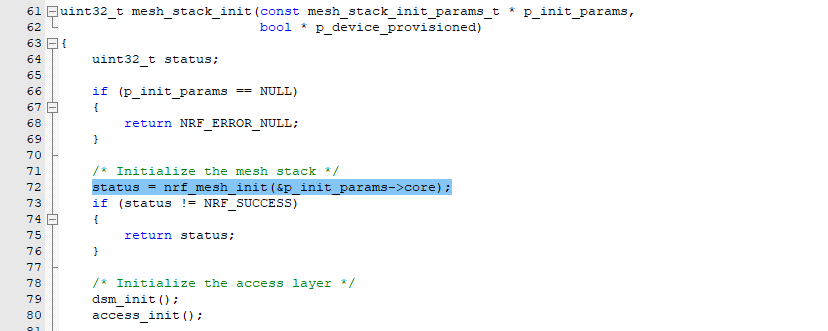


在nrf\_mesh.c文件中的uint32\_t nrf\_mesh\_init(const nrf\_mesh\_init\_params\_t \* p\_init\_params)函数调用了

nrf\_mesh\_configure\_device\_uuid\_set(p\_init\_params->p\_uuid);

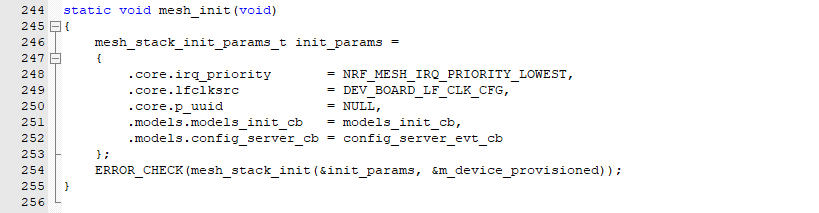


uint32\_t nrf\_mesh\_init(const nrf\_mesh\_init\_params\_t \* p\_init\_params)函数在nrf\_mesh.c文件的uint32\_t mesh\_stack\_init(const mesh\_stack\_init\_params\_t \* p\_init\_params,bool \* p\_device\_provisioned)被调用：



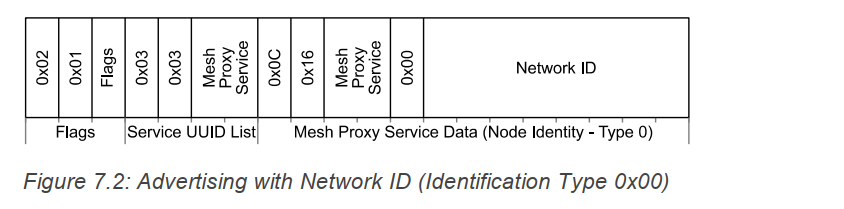
uint32\_t mesh\_stack\_init(const mesh\_stack\_init\_params\_t \* p\_init\_params,

bool \* p\_device\_provisioned)函数在main.c文件中被static void mesh\_init(void)函数调用：



由上图可看出，init\_params结构体的成员core.p\_uuid就是用于设置device\_uuid的变量。

为了支持Proxy特性，需要设置节点的Proxy Feature为Enable，那么节点会广播如下图所示广播包：



参考Mesh Profile 7.2.2.2.2 *Advertising with Network ID*

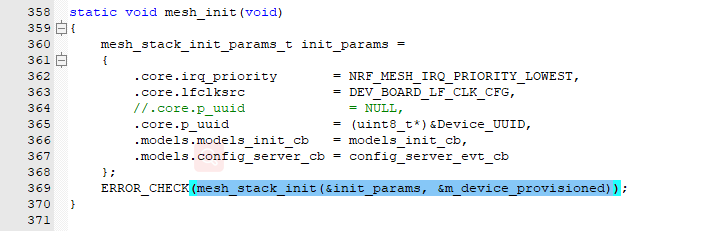
各字段含义如下：

|  |  |  |
| --- | --- | --- |
| Field | Size(Octets) | Notes |
| Flags | 1 | 长度，  0x02表示后面还有2个字节 |
| 1 | 类型，  0x01 表示类型是Flags( Assigned Number) |
| 1 | Flags的值，0x02表示LE General Discover Mode |
| Service UUID List | 1 | 长度，0x03表示后面还有3个字节 |
| 1 | 类型，0x03表示Complete List of 16-bit Service  Class UUIDs |
| 2 | Mesh Proxy Service, 0x1828 |
|  | 1 | 长度，  0x0C表示后面还有12字节 |
| 1 | 类型，0x16表示Service Data 16-bit UUID |
| 2 | Mesh Proxy Service, 0x1828 |
| Identification Type | 1 | 实体类型， 0x00表示Network ID 类型 |
| Network ID | 8 | 参考蓝牙Mesh协议，Network ID = k3(NetKey) |

在Nordic Mesh SDK中，通过以下过程，完成以上数据的填充和广播：

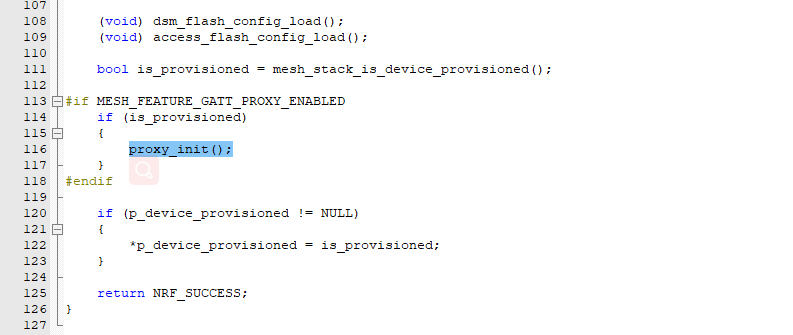
1. 在main.c文件的static void mesh\_init(void)函数中调用

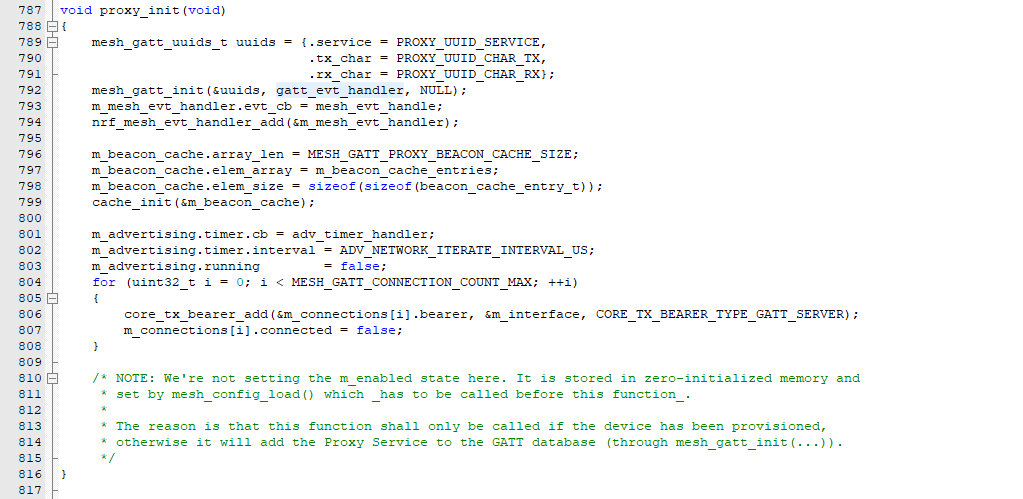
mesh\_stack\_init(&init\_params, &m\_device\_provisioned)



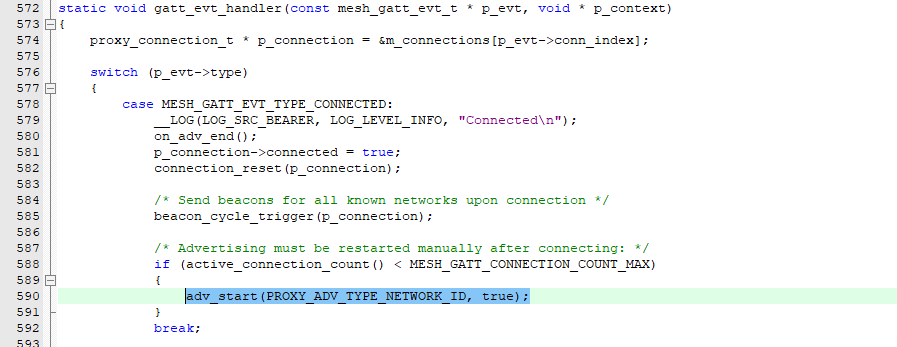
1. 在mesn\_stack.c文件的函数uint32\_t mesh\_stack\_init(const mesh\_stack\_init\_params\_t \* p\_init\_params, bool \* p\_device\_provisioned)调用了

proxy\_init()进行Proxy初始化：

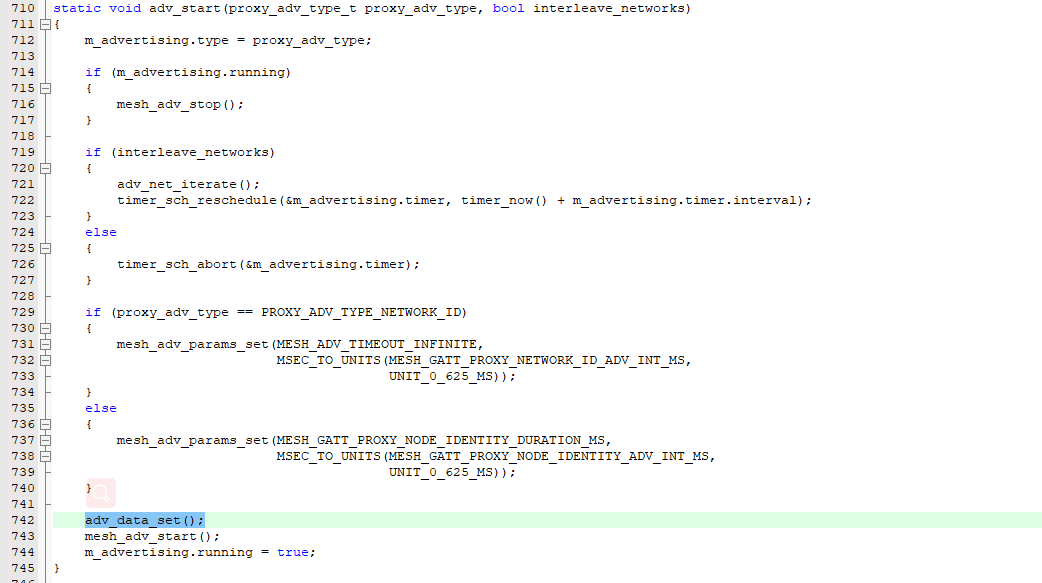




在上图的第792行，mesh\_gatt\_init(&uuids, gatt\_evt\_handler, NULL)绑定了gatt\_evt\_handler回调处理函数：



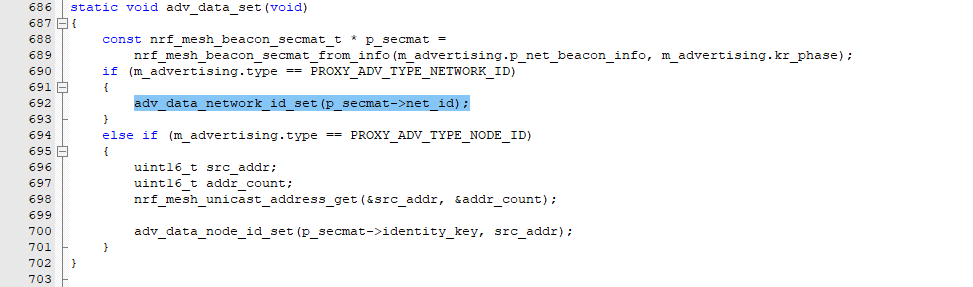
1. 如上图所示，当设备连接成功时，产生MESH\_GATT\_EVT\_TYPE\_CONNECTED事件，然后执行第590行的函数，填充Network ID包，



在proxy.c文件中的static void adv\_start(proxy\_adv\_type\_t proxy\_adv\_type, bool interleave\_networks)函数中执行如图第742行。

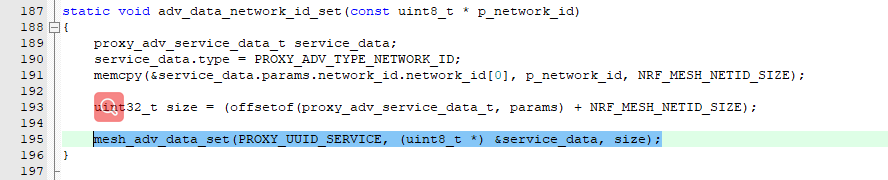
1. 在proxy.c文件中，static void adv\_data\_set(void)函数执行

adv\_data\_network\_id\_set(p\_secmat->net\_id)。填充对应的数据

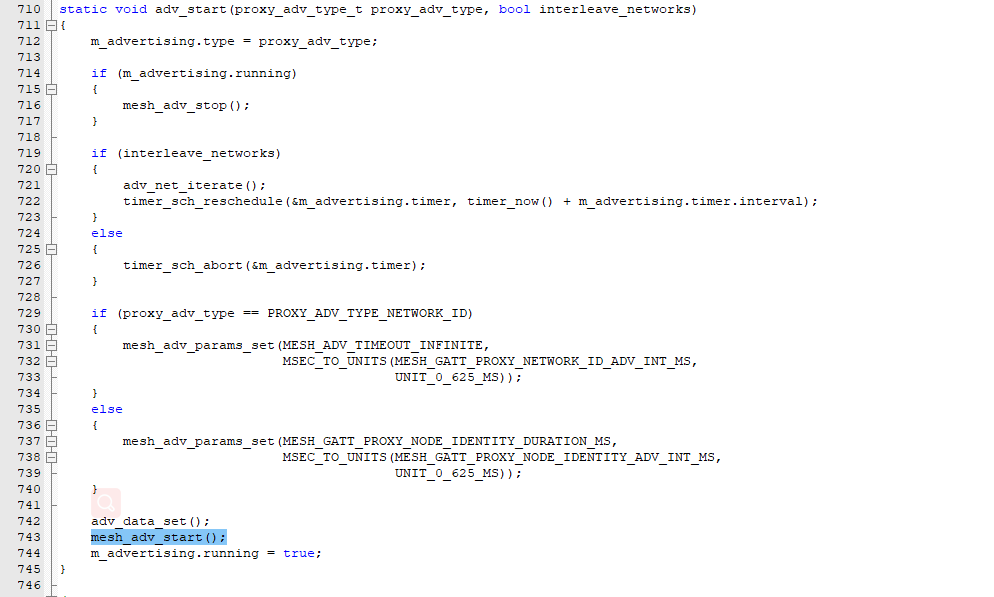


1. 在proxy.c文件，static void adv\_data\_network\_id\_set(const uint8\_t \* p\_network\_id)函数调用了SDK中GATT广播包的填充函数

void mesh\_adv\_data\_set(uint16\_t service\_uuid, const uint8\_t \* p\_service\_data, uint8\_t length)，跟前面的PB-GATT 广播包调用一样的填充函数。



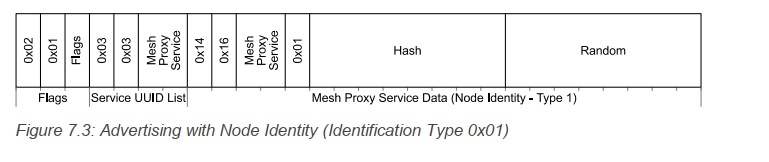
1. 然后回到第③步执行mesh\_adv\_start();发送广播包



为了支持Proxy.c 特性，且节点的Proxy Feature 设置为Enable，节点还会在以下两种情况下广播Node Identity包：

1. 节点通过PB-GATT入网后立即开始广播，每次广播时长是40ms，广播间隔2S
2. 节点在Configuration 阶段设置Node Identity state是Enable的时候开始连续广播，60秒后停止广播。

Node Identity广播包如下图所示：



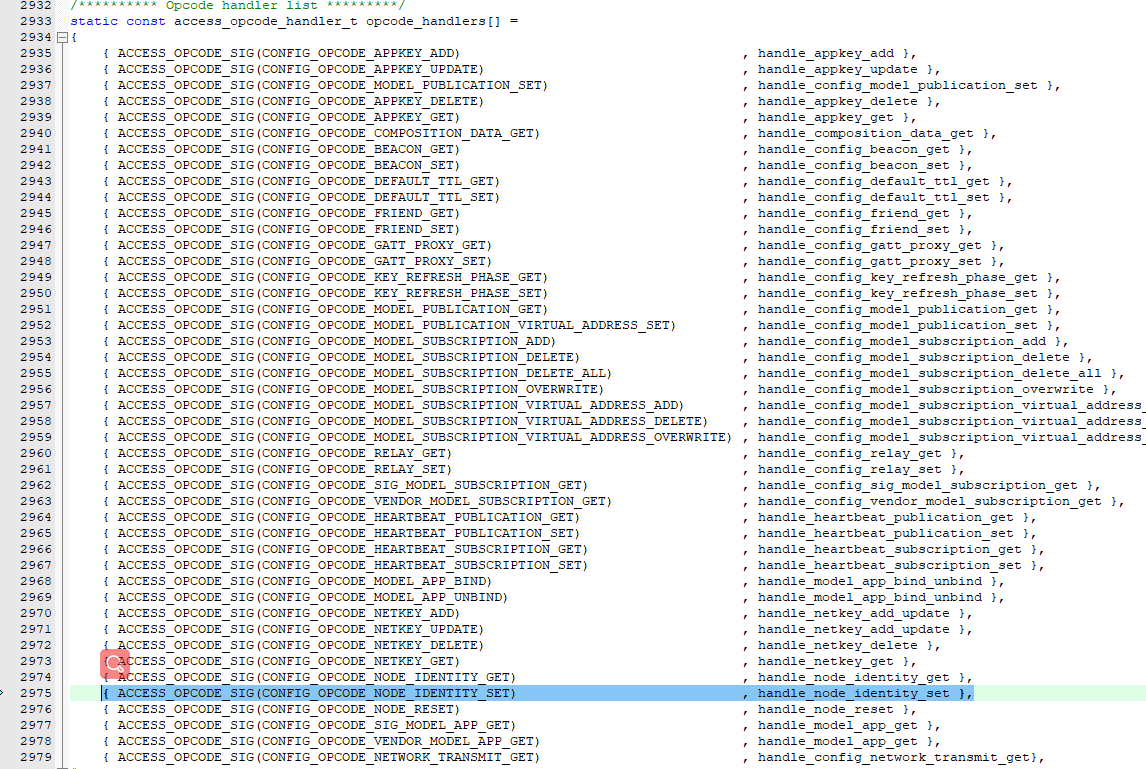
参考 Mesh Profile 7.2.2.2.3 *Advertising with Node Identity*

具体的字段含义如下表：

|  |  |  |
| --- | --- | --- |
| Field | Size(Octets) | Notes |
| Flags | 1 | 长度，  0x02表示后面还有2个字节 |
| 1 | 类型，  0x01 表示类型是Flags (Assigned Number) |
| 1 | Flags的值，0x02表示LE General Discoverable Mode |
| Service UUID List | 1 | 长度，0x03表示后面还有3字节 |
| 1 | 类型，0x03表示Complete List of 16-bit Service Class UUIDs |
| 2 | Mesh Proxy Service, 0x1828 |
|  | 1 | 长度，  0x14表示后面还有20字节 |
| 1 | 类型，0x16表示Service Data 16-bit UUID |
| 2 | Mesh Proxy Service, 0x1828 |
| Identification Type | 1 | 实体类型，0x01表示节点Node Identty类型 |
| Hash | 8 | 参考蓝牙协议，  Hash = e(IdentityKey,  Padding|Random|Address) mod 264 |
| Random | 8 | 随机数 |

在2的情况中。Nordic Mesh SDK 在config\_server.c文件的

static const access\_opcode\_handler\_t opcode\_handlers结构体中，绑定了命令和处理函数：

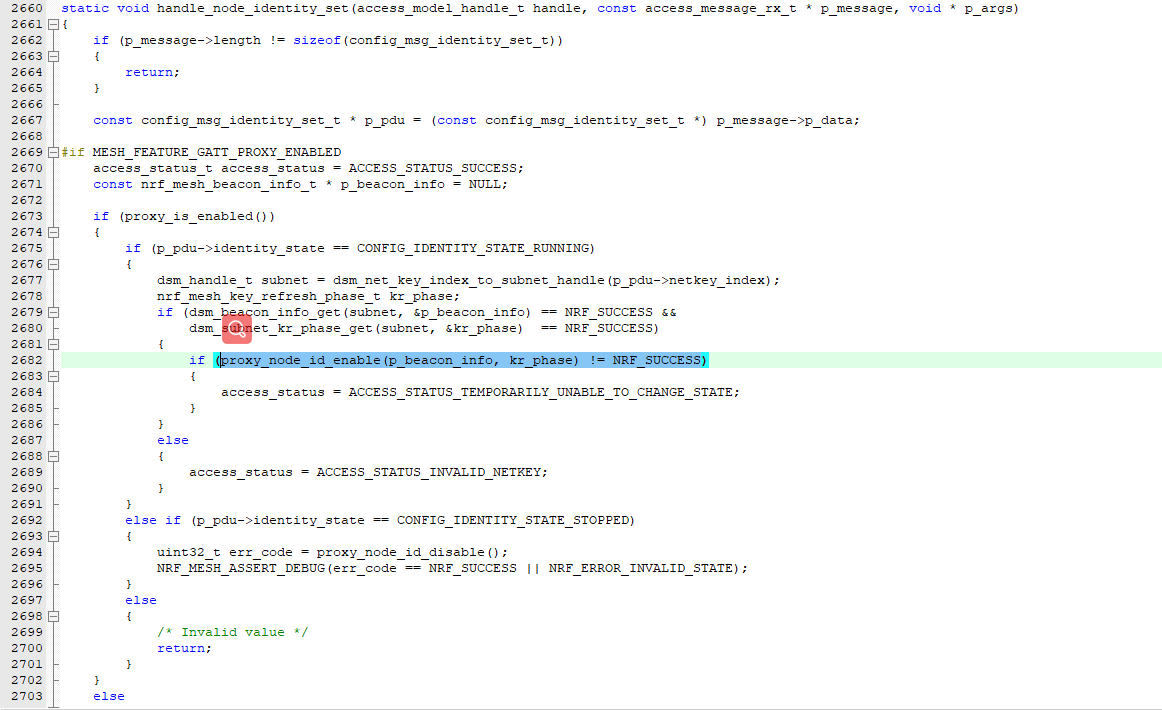


在上图中可以看到CONFIG\_OPCODE\_NODE\_IDENTITY\_SET命令码绑定了处理函数

handle\_node\_identity\_set。

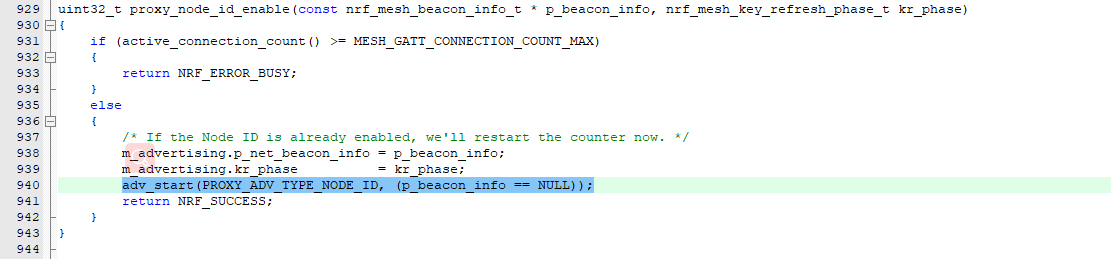
在config\_server.c文件中，定义了

static void handle\_node\_identity\_set(access\_model\_handle\_t handle, const access\_message\_rx\_t \* p\_message, void \* p\_args)

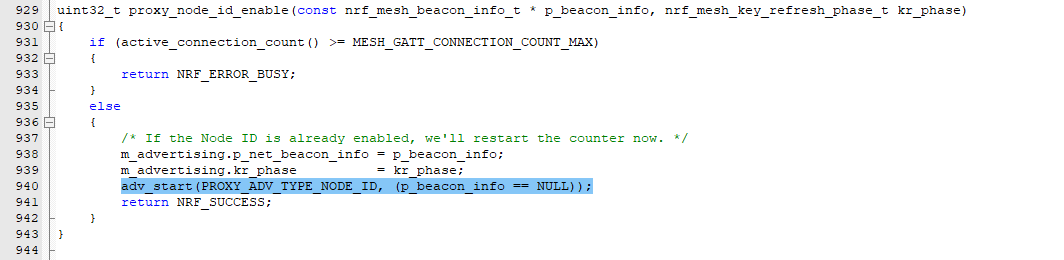


1. 当收到CONFIG\_OPCODE\_NODE\_IDENTITY\_SET命令时，且设置Node Identity State 是Enable的时候，执行第2682行代码，执行函数

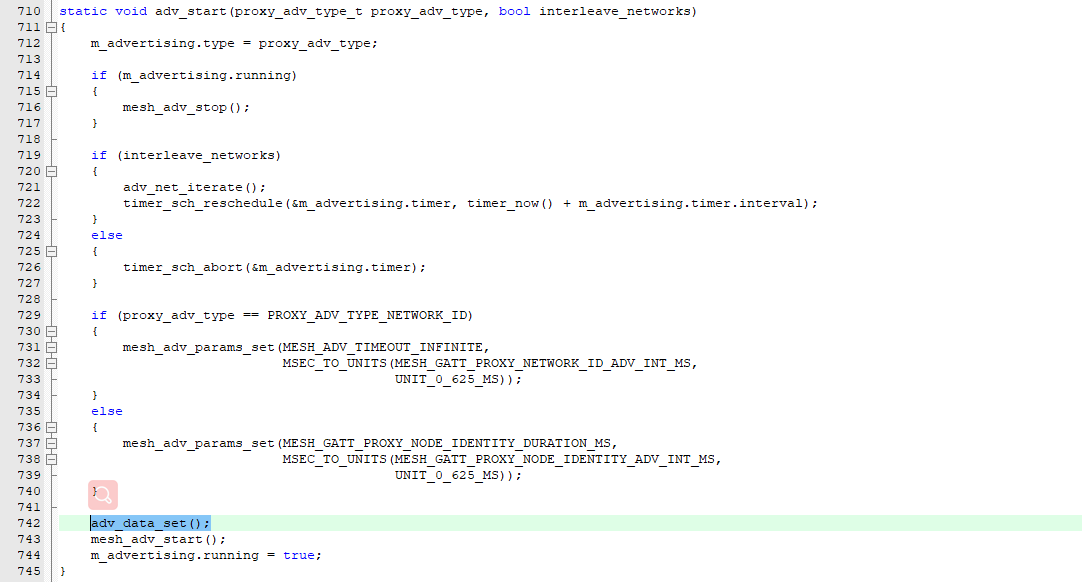
uint32\_t proxy\_node\_id\_enable(const nrf\_mesh\_beacon\_info\_t \* p\_beacon\_info, nrf\_mesh\_key\_refresh\_phase\_t kr\_phase)。



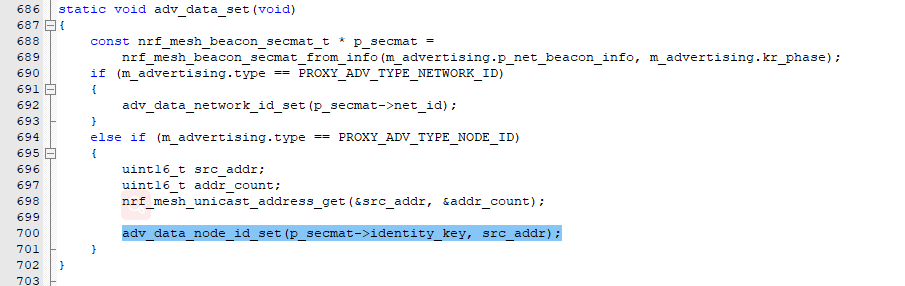
1. 在proxy.c文件中，uint32\_t proxy\_node\_id\_enable(const nrf\_mesh\_beacon\_info\_t \* p\_beacon\_info, nrf\_mesh\_key\_refresh\_phase\_t kr\_phase)函数调用了adv\_start(PROXY\_ADV\_TYPE\_NODE\_ID, (p\_beacon\_info == NULL));



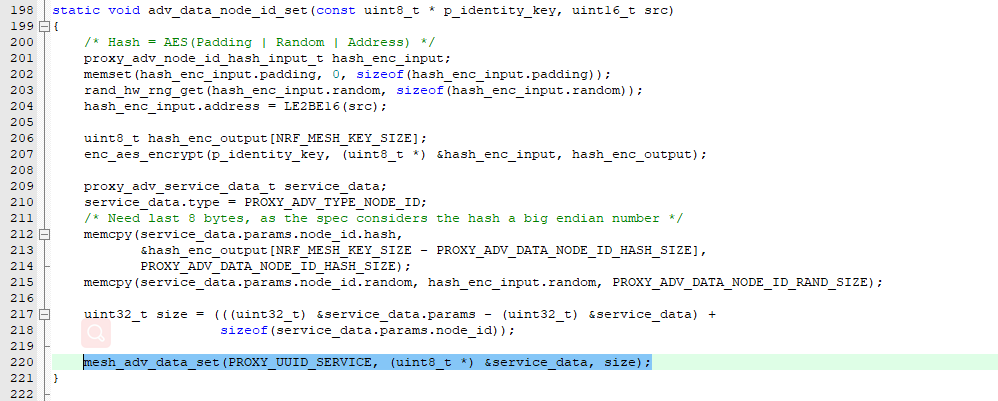
1. 在proxy.c文件中，static void adv\_start(proxy\_adv\_type\_t proxy\_adv\_type, bool interleave\_networks)函数调用static void adv\_data\_set(void)



1. 在proxy.c文件中，static void adv\_data\_set(void)函数调用adv\_data\_node\_id\_set(p\_secmat->identity\_key, src\_addr);



1. 在proxy.c文件中，static void adv\_data\_node\_id\_set(const uint8\_t \* p\_identity\_key, uint16\_t src)函数调用mesh\_adv\_data\_set(PROXY\_UUID\_SERVICE, (uint8\_t \*) &service\_data, size)完成对广播数据的填充:



最后回到③中执行mesh\_adv\_start();进行广播：

