低功耗蓝牙开发者手册

桃芯科技（苏州）有限公司

官网：www.ingchips.com

www.ingchips.cn

邮箱：service@ingchips.com

电话：010-85160285

地址：北京市海淀区知春路 49 号紫金数 3 号楼 8 层 803

深圳市南山区科技园曙光大厦 1009

目录

插图

表格

版本历史

版本	信息	日期
0.5	初始版本	2022-09-07
0.6	增加减速模式、功率控制	2022-09-15
0.7	针对 SDK v8.2 更新	2022-10-31
0.8	增加同步版 API、线程安全 API 等	2022-11-19
0.9	更新 L2CAP，SM 等	2025-02-20
1.0	增加 Controller 的说明	2025-02-28

第一章　简介

欢迎使用 *INGCHIPS* 918XX/916XX 软件开发工具包（SDK）.

本手册将带您了解 *INGCHIPS* 各系列 BLE SoC 芯片上的低功耗蓝牙开发，了解整个协议栈设计思路，熟悉各模块主要接口的使用方法。

本手册侧重从开发者的角度介绍 BLE 的开发，涉及 BLE 规范时采用了更 “实用化” 的描述，不去关注规范里的每一个细节。

SDK 工具可以生成本手册里提到的一些常见代码，如回调函数、事件处理等。

1.1　缩略语及术语

表 1.1: 缩略语

缩略语	说明
BLE	低功耗蓝牙（Bluetooth LE，Bluetooth Low Engergy）
CCCD	客户端特征配置描述符（Client Characteristic Configuration Descriptor）
CTE	定频扩展（Constant Tone Extension）
ECC	椭圆曲线密码学（Elliptic Curve Cryptography）
HCI	Host Controller 接口
IRK	身份解析密钥 (Identity Resolving Key)
L2CAP	逻辑链路控制与适配协议（Logical Link Control and Adaptation Protocol）
LL	链路层（Link Layer）
LMP	BR/EDR 的链路管理协议（Link Manager Protocol）
LTK	长期密钥（Long Term Key）
MIC	消息认证码（Message Integrity Code）
MITM	中间人（Man-In-The-Middle）
OOB	带外（Out Of Band）

缩略语	说明
RSSI	接收信号强度指示（Received Signal Strength Indicator）
SCA	睡眠时钟精度（Sleep Clock Accuracy）
SPSM	简化的协议/服务选择器 (Simplified Protocol/Service Multiplexer)
UUID	通用唯一识别码（Universally Unique IDentifier）

表 1.2: 术语

术语	说明
1M	BLE 使用的一种 PHY，符号速率为 $1Msps$
2M	BLE 使用的一种 PHY，符号速率为 $2Msps$
Characteristic	特征，是服务的组成部分
Coded	BLE 使用的一种 PHY，基于卷积码的信道编码
Controller	BLE 控制器，整个蓝牙协议栈偏低层的部分
Handle	句柄
Host	BLE 主机，整个蓝牙协议栈偏上层的部分
PHY	BLE 的物理层传输方式
Service	服务，由特征组成

蓝牙规范从 v5.3 版本开始更新了部分术语，本手册沿用旧称。表 1.3 是这部分术语的新旧对照。

表 1.3: 新旧术语对照

原名称	新名称	本手册
Master	Central	主角色
Slave	Peripheral	从角色
White List	Accept List	白名单

1.2 参考文档

1. Bluetooth SIG[1]

[1] https://www.bluetooth.com/

2. Controller API Reference
3. Application Note: Direction Finding Solution[2]

[2]https://ingchips.github.io/application-notes/an_aoa/index.html

第二章　概览

2.1　基本原则

1. BLE 协议栈**尽最大努力**执行各项任务，比如
    - 将发射功率设置为 $100dBm$，协议栈将以最大功率进行发射
    - 广播、连接并发时，部分广播事件、连接事件可能因需要执行其它任务而被忽略（跳过）
    - 极端情况下，当协议栈可能会主动终止某些任务并上报相应的事件
2. 连接模式下已进入 BLE 协议栈的数据包不会丢失、总是可以送达，除非连接断开

2.2　协议栈架构

Controller、Host 以两个任务（或者线程）的形式运行，HCI 接口经过特殊设计，尽量减少内存数据的复制。Host 的架构如图 2.1 所示，主要通过 GAP、ATT、GATT Client、SM 等 4 个模块为开发者提供操作接口。

Host 任务的伪代码如下：

```
void host_task(void)
{
    host_init();
    while (true)
    {
        if (recv_msg(msg) != 0) continue;
        process_msg(msg);
```

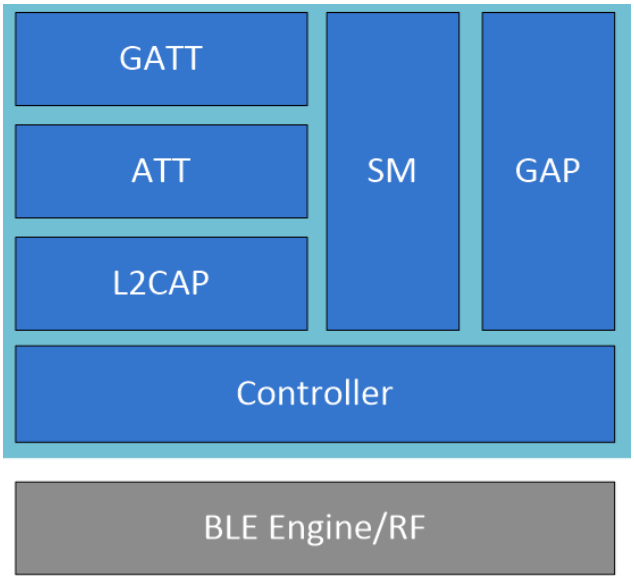

图 2.1: Host 架构

```
    }
}
```

也就是说 Host 任务是由各种消息驱动的。这些消息既包括来自 Controller 的事件、ACL 数据，也包含软件定时器消息和用户发送的消息（btstack_push_user_msg 和 btstack_push_user_runnable）。

process_msg 的伪代码如下：

```
void process_msg(msg)
{
    switch (msg)
    {
    case HCI Event:
        调用各个回调();
        break;
    case ACL 数据:
        调用各个回调();
        break;
    case 软件定时器:
```

```
        超时处理();
        break;
    case 用户消息:
        弹出用户消息事件();
        break;
    case 用户执行体:
        运行用户执行体();
        break;
    }
}
```

各个模块（包含协议内部模块及 App[1]）通过注册回调函数以响应这些事件。有的模块在处理这些消息时又会产生其它的新事件，为了响应这些新事件可以再向这些模块注册回调函数。也就是说，Host 内部各个模块（以及 App）通过消息、回调函数耦合在一起。例如：

- `hci_add_event_handler`

  通过这个函数可以注册一个能够监听所有 HCI 事件的回调；

- `att_server_init`

  向 ATT Server 模块注册用以响应特征读写的回调函数。

2.3 通信模型

通信是指由一地向另一地进行信息的传输与交换，其目的是传输信息、削减另一地的不确定性。对于 BLE 而言，主要有两种通信方式：一对多的广播、一对一的连接。低功耗蓝牙定义了四种角色：广播发送方称为广播者（Broadcaster），接收方称为观察者（Observer）；连接的发起方称为主角色（新名称为中心角色，Central），接受方称为从角色（新名称为外围角色，Periperhal）。

蓝牙规范定义了广播数据的格式、AD 类型，见图 2.2。如，AD 类型 `0x09` 表示设备名称，其后面的 AD 数据域是一个 UTF-8 字符串。另请参阅 “广播数据” 一节。

对于连接模式，低功耗蓝牙定义了特征（Characteristic）和值（Value）这两个概念，进而组织成服务（Service），再由服务组成配置（Profile）。特征的标识用 UUID 标识。客户端发现

[1]如无特殊说明，本文档中的 App 皆指运行在芯片上的蓝牙程序。

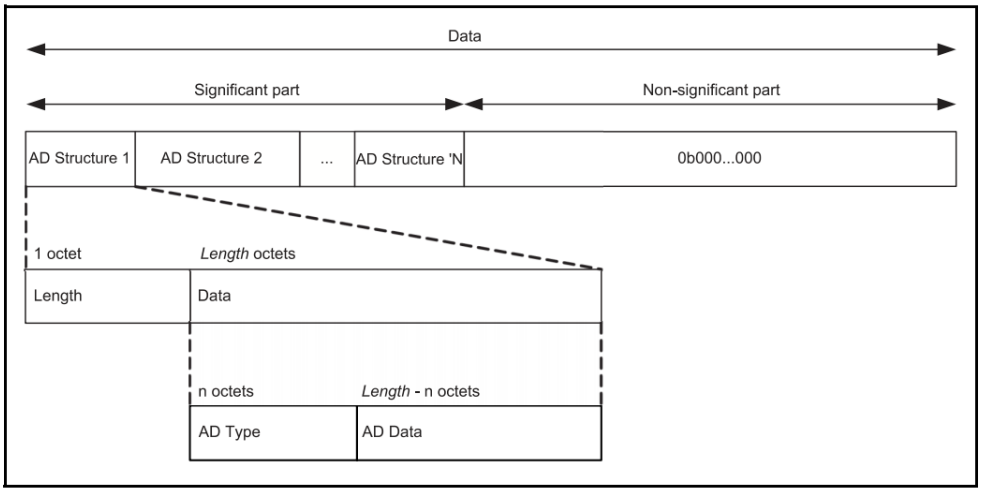

图 2.2: 广播及扫描响应数据包格式

了服务器支持的服务后，就可以读写特征的值，或者订阅特征[2]。显然特征不见得支持所有这些操作，因此，蓝牙核心规范又为特征定义了属性（Property）、描述符（Discriptor）等装饰物，以说明特征所支持的操作、需要的权限。UUID 长度为 16 字节，为了提高传输效率，BLE 定义了一系列特殊的 UUID，它们的区别仅在于 2 个字节，另外 14 个字节相同：

```
0x0000xxxx-0000-1000-8000-00805F9B34FB
```

这两个字节就是所谓的 16-bit UUID，由蓝牙特别兴趣小组公司负责管理、分配[3]。

2.4 回调函数事件包

协议栈的各种回调函数遵循类似的原型，其输入称为事件包：

```
typedef void (*btstack_packet_handler_t) (
    // 事件包类型
    uint8_t packet_type,
    // 关联的信道（一般指蓝牙连接句柄）
    uint16_t channel,
    // 事件包内容
```

[2]指示服务器端主动报告特征的值。

[3]https://www.bluetooth.com/16-bit-uuids-for-sdos/

```
    const uint8_t *packet,
    // 事件包内容的长度
    uint16_t size);
```

包类型 packet_type 的取值如下：

- HCI_EVENT_PACKET：HCI 事件包（**常用**）

  这个类型的事件包是个“大杂烩”，多个模块弹出的事件包都会使用这个类型。

- HCI_ACL_DATA_PACKET：Controller 上报的 ACL 数据

  这个类型的事件包只会被通过 hci_register_acl_packet_handler 注册的 ACL 数据回调函数收到。

- HCI_COMPLETED_SDU_PACKET：来自 LE 信用信道的完整 SDU

  这个类型的事件包只会被通过 l2cap_register_service 注册的 L2CAP 服务回调函数收到。

- L2CAP_EVENT_PACKET：来自 L2CAP 的事件包

  这个类型的事件包只会被通过 l2cap_add_event_handler 注册的 L2CAP 事件回调函数收到。

2.5 事件包的解析

下面只介绍 HCI_EVENT_PACKET 事件包的解析，其它几种类型不常用。

首先使用 hci_event_packet_get_type(packet) 获取事件代码，根据事件代码的不同，后续的处理大不相同。常用的几种事件代码如下。

1. BTSTACK_EVENT_STATE：蓝牙协议栈事件

   一般用于响应协议栈初始化：

```
if (btstack_event_state_get_state(packet) != HCI_STATE_WORKING)
    break;
// App 初始化
```

2. HCI_EVENT_LE_META：BLE 元事件

   这个元事件下辖多个子事件。先通过 hci_event_le_meta_get_subevent_code(packet) 获得子事件代码，然后通过 decode_hci_le_meta_event(packet, sub_event_type) 宏得到子事件的内容。sub_event_type 为子事件内容对应的数据类型，各字段与蓝牙核心规范里的定义一致[4]。

   协议栈的版本及初始化流程导致下列子事件不会出现，只会出现对应的扩展过的、功能更全面的事件：

   - HCI_SUBEVENT_LE_CONNECTION_COMPLETE

     5.3 及以下改用 HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE；5.4 及以上改用 HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE_V2

   - HCI_SUBEVENT_LE_ADVERTISING_REPORT

     改用 HCI_SUBEVENT_LE_EXTENDED_ADVERTISING_REPORT

   协议栈可能报告的子事件及对应的 sub_event_type 类型如下：

   - HCI_SUBEVENT_LE_CONNECTION_UPDATE_COMPLETE

     连接参数更新（le_meta_event_conn_update_complete_t）

   - HCI_SUBEVENT_LE_READ_REMOTE_USED_FEATURES_COMPLETE

     读取对端特性（le_meta_event_read_remote_feature_complete_t）

   - HCI_SUBEVENT_LE_LONG_TERM_KEY_REQUEST

     请求 LTK（le_meta_event_long_term_key_request_t）

   - HCI_SUBEVENT_LE_REMOTE_CONNECTION_PARAMETER_REQUEST_COMPLETE

     远端连接参数请求（le_meta_event_remote_conn_param_request_t）

   - HCI_SUBEVENT_LE_DATA_LENGTH_CHANGE_EVENT

     数据包长度改变（le_meta_event_data_length_changed_t）

   - HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE (5.3 及以下)

     连接建立（le_meta_event_enh_create_conn_complete_t）

   - HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE_V2 (5.4 及以上)

     连接建立（le_meta_event_enh_create_conn_complete_v2_t），与 HCI_SUBEVENT_LE_ENHANCED_CONNEC 相比，增加了 PAwR 的相关信息

[4]私有事件（如 HCI_SUBEVENT_LE_VENDOR_PRO_CONNECTIONLESS_IQ_REPORT）除外。

- HCI_SUBEVENT_LE_DIRECT_ADVERTISING_REPORT

  定向广播报告（le_meta_directed_adv_report_t）

- HCI_SUBEVENT_LE_PHY_UPDATE_COMPLETE

  PHY 更新完成（le_meta_phy_update_complete_t）

- HCI_SUBEVENT_LE_EXTENDED_ADVERTISING_REPORT

  扩展广播报告（le_meta_event_ext_adv_report_t）

- HCI_SUBEVENT_LE_PERIODIC_ADVERTISING_SYNC_ESTABLISHED (5.3 及以下)

  周期广播同步建立（le_meta_event_periodic_adv_sync_established_t）

- HCI_SUBEVENT_LE_PERIODIC_ADVERTISING_SYNC_ESTABLISHED_V2 (5.4 及以上)

  周期广播同步建立（le_meta_event_periodic_adv_sync_established_v2_t），与 HCI_SUBEVENT_LE_PERIODIC_ADVERTISING_SYNC_ESTABLISHED 相比，增加了 PAwR 的相关信息

- HCI_SUBEVENT_LE_PERIODIC_ADVERTISING_REPORT (5.3 及以下)

  周期广播报告（le_meta_event_periodic_adv_report_t）

- HCI_SUBEVENT_LE_PERIODIC_ADVERTISING_REPORT_V2 (5.4 及以上)

  周期广播报告（le_meta_event_periodic_adv_report_v2_t），与 HCI_SUBEVENT_LE_PERIODIC_AD 相比，增加了 PAwR 的相关信息

- HCI_SUBEVENT_LE_PERIODIC_ADVERTISING_SYNC_LOST

  周期广播同步丢失（le_meta_event_periodic_adv_sync_lost_t）

- HCI_SUBEVENT_LE_SCAN_TIMEOUT

  扫描超时

- HCI_SUBEVENT_LE_ADVERTISING_SET_TERMINATED

  广播停止（le_meta_adv_set_terminated_t）

  如果是由于广播是由于建立了连接而停止，那么从这个事件里可以得到广播句柄和连接句柄的对应关系。

- HCI_SUBEVENT_LE_SCAN_REQUEST_RECEIVED

  收到扫描请求（le_meta_scan_req_received_t）

- HCI_SUBEVENT_LE_CHANNEL_SELECTION_ALGORITHM

  信道选择算法（le_meta_ch_sel_algo_t）

- HCI_SUBEVENT_LE_CONNECTIONLESS_IQ_REPORT

  无连接 IQ 报告（le_meta_connless_iq_report_t）

- HCI_SUBEVENT_LE_CONNECTION_IQ_REPORT

  有连接 IQ 报告（le_meta_conn_iq_report_t）

- HCI_SUBEVENT_LE_CTE_REQ_FAILED

  CTE 请求失败（le_meta_cte_req_failed_t）

- HCI_SUBEVENT_LE_PRD_ADV_SYNC_TRANSFER_RCVD (5.3 及以下)

  周期广播转移请求（le_meta_prd_adv_sync_transfer_recv_t）

- HCI_SUBEVENT_LE_PRD_ADV_SYNC_TRANSFER_RCVD_V2 (5.4 及以上)

  周期广播转移请求（le_meta_prd_adv_sync_transfer_recv_v2_t），与 HCI_SUBEVENT_LE_PRD_ADV_SYN
  相比，增加了 PAwR 的相关信息

- HCI_SUBEVENT_LE_REQUEST_PEER_SCA

  对端 SCA 请求完成（le_meta_request_peer_sca_complete_t）

- HCI_SUBEVENT_LE_PATH_LOSS_THRESHOLD

  路损门限报告（le_meta_path_loss_threshold_t）

- HCI_SUBEVENT_LE_TRANSMIT_POWER_REPORTING

  发射功率报告（le_meta_tx_power_reporting_t）

- HCI_SUBEVENT_LE_SUBRATE_CHANGE

  减速模式改变（le_meta_subrate_change_t）

- HCI_SUBEVENT_PRD_ADV_SUBEVT_DATA_REQ

  PAwR 子事件数据请求（le_mete_event_prd_adv_subevent_data_req_t）

- HCI_SUBEVENT_PRD_ADV_RSP_REPORT

  PAwR 子事件响应（le_mete_event_prd_adv_rsp_report_t）

- HCI_SUBEVENT_LE_VENDOR_CONNECTION_ABORTED

  私有连接中止报告（le_meta_event_vendor_connection_aborted_t）

- HCI_SUBEVENT_LE_VENDOR_CHANNEL_MAP_UPDATE

  私有信道集合更新报告（le_meta_event_vendor_channel_map_update_t）

- HCI_SUBEVENT_LE_VENDOR_PRO_CONNECTIONLESS_IQ_REPORT

  私有无连接 IQ 报告（le_meta_pro_connless_iq_report_t）

3. HCI_EVENT_DISCONNECTION_COMPLETE：连接断开事件

   通过 decode_hci_event_disconn_complete(packet) 解析事件内容：

```
typedef struct event_disconn_complete
{
    // 状态码
    uint8_t status;
    // 连接句柄
    uint16_t conn_handle;
    // 原因
    uint8_t reason;
} event_disconn_complete_t;
```

4. HCI_EVENT_COMMAND_COMPLETE：HCI 命令完成事件

   通过 hci_event_command_complete_get_command_opcode(packet) 获得 HCI 命令码。通过 hci_event_command_complete_get_return_parameters(packet) 获得 Controller 返回的参数，其中第 1 个字节为命令完成的状态，0 表示没有错误，详见 “Controller 错误码”。其它参数需要根据命令码做具体分析。

5. HCI_EVENT_COMMAND_STATUS：HCI 命令状态事件

   有些 HCI 命令可以立即完成，得到结果，Controller 会上报相应的 HCI_EVENT_COMMAND_COMPLETE。有些 HCI 命令则需要一定时间才能完成，比如发起连接，Controller 收到这样的命令后不上报 HCI_EVENT_COMMAND_COMPLETE，而是上报 HCI_EVENT_COMMAND_STATUS。

   通过 hci_event_command_status_get_command_opcode(packet) 获得 HCI 命令码。通过 hci_event_command_status_get_status(packet) 获得状态，0 表示没有错误，详见 “Controller 错误码”。

6. BTSTACK_EVENT_USER_MSG：来自 btstack_push_user_msg 的用户消息

2.6 Controller 错误码

表 2.1 列出了 Controller 使用的部分错误码。

表 2.1： Controller 错误码

错误码	含义
0x00	无错误

错误码	含义
0x01	未知的命令
0x02	未知的连接句柄
0x03	硬件错误
0x05	鉴权失败
0x06	PIN 或密钥缺失
0x07	超出内存容量
0x08	连接超时
0x09	连接数目达到极限
0x0B	连接已存在
0x0C	命令不允许
0x11	不支持的特性或参数值
0x12	HCI 命令参数错误
0x13	远端用户断开连接
0x14	远端设备因资源紧张断开连接
0x15	远端设备因关机断开连接
0x16	本机断开连接
0x1A	不支持的远端或 LMP 特性
0x1E	非法的 LMP 或 LL 参数
0x1F	未指定的错误
0x20	不支持的 LMP 或 LL 参数
0x22	LMP 或 LL 响应超时
0x23	LMP 错误（会话冲突）
0x28	时机已过
0x2E	不支持信道分类
0x2F	安全特性不足
0x3A	Controller 正忙
0x3C	定向广播超时
0x3D	因 MIC 错误而断开连接
0x3E	连接无法建立
0x43	到达极限
0x44	Host 已取消
0x45	数据包太长
0x46	太晚了
0x47	太早了

错误码	含义
0xFE	任务调度失败（私有错误码）
0xFF	其它错误（私有错误码）

2.7 ATT 错误码

表 2.2 列出了蓝牙核心规范定义的 ATT 错误码。

表 2.2: ATT 错误码

错误码	宏定义	含义
0x00		无错误。
0x01	ATT_ERROR_INVALID_HANDLE	无效句柄。
0x02	ATT_ERROR_READ_NOT_PERMITTED	指定的特征不可读。
0x03	ATT_ERROR_WRITE_NOT_PERMITTED	指定的特征不可写。
0x04	ATT_ERROR_INVALID_PDU	PDU 不合法。
0x05	ATT_ERROR_INSUFFICIENT_AUTHENTICATION	身份认证不足。在读写该特征之前需要进行身份认证。
0x06	ATT_ERROR_REQUEST_NOT_SUPPORTED	ATT 服务器不支持该请求。
0x07	ATT_ERROR_INVALID_OFFSET	Offset 参数超出范围。
0x08	ATT_ERROR_INSUFFICIENT_AUTHORIZATION	授权不足。在读写该特征之前需要获得授权。
0x09	ATT_ERROR_PREPARE_QUEUE_FULL	Prepare Write 队列已满。
0x0A	ATT_ERROR_ATTRIBUTE_NOT_FOUND	未指定指定的特征。
0x0B	ATT_ERROR_ATTRIBUTE_NOT_LONG	特征太长，无法通过 ATT_READ_BLOB_REQ PDU 读取。
0x0C	ATT_ERROR_INSUFFICIENT_ENCRYPTION_KEY_SIZE	密钥长度太短。
0x0D	ATT_ERROR_INVALID_ATTRIBUTE_VALUE_LENGTH	特征的长度不合法。
0x0E	ATT_ERROR_UNLIKELY_ERROR	不常见的错误。
0x0F	ATT_ERROR_INSUFFICIENT_ENCRYPTION	加密不足。在读写该特征之前需要开启加密。
0x10	ATT_ERROR_UNSUPPORTED_GROUP_TYPE	不支持的组类型。
0x11	ATT_ERROR_INSUFFICIENT_RESOURCES	资源不足，无法完成请求。
0x12	ATT_ERROR_DATABASE_OUT_OF_SYNC	数据库失步。服务器要求客户端重新发现数据库。
0x13	ATT_ERROR_VALUE_NOT_ALLOWED	特征的参数值非法。
0x80~0x9F		应用层错误码。

错误码	宏定义	含义
0xE0~0xFF		Profile 或服务的错误码[5]。

2.8 协议栈 API 错误码

表 2.3 列出了协议栈 API 返回的常见返回值（错误码）。

表 2.3: 协议栈 API 常见返回值（错误码）

错误码	宏定义	含义
0x0		无错误。
0x55	BTSTACK_BUSY	协议栈正忙。
0x56	BTSTACK_MEMORY_ALLOC_FAILED	内存不足，协议栈对象分配失败。
0x57	BTSTACK_ACL_BUFFERS_FULL	内存不足，无法缓存 ACL 数据。
0x59	BTSTACK_LE_CHANNEL_NOT_EXIST	指定的 LE 信道句柄不存在（未连接）。
0x69	L2CAP_SERVICE_ALREADY_REGISTERED	服务已注册。
0x6A	L2CAP_DATA_LEN_EXCEEDS_REMOTE_MTU	数据长度超出远端 MTU。
0x6B	L2CAP_SERVICE_NOT_REGISTERED	服务未注册。
0x6C	L2CAP_CONNECTION_INSUFFICIENT_SECURITY	安全性不足。
0x90	ATT_HANDLE_VALUE_INDICATION_IN_PROGRESS	上个指示（Indiction）流程未完成。
0x93	GATT_CLIENT_NOT_CONNECTED	指定的连接句柄无效（未连接）。
0x94	GATT_CLIENT_BUSY	GATT 客户端正忙。
0x95	GATT_CLIENT_IN_WRONG_STATE	GATT 客户端状态错误（会话冲突）。
0x97	GATT_CLIENT_VALUE_TOO_LONG	GATT 客户端：要写入的数据超过最大长度。
0x98	GATT_CLIENT_CHARACTERISTIC_NOTIFICATION_NOT_SUPPORTED	GATT 客户端：待订阅的特征不支持通知（Notification）。
0x99	GATT_CLIENT_CHARACTERISTIC_INDICATION_NOT_SUPPORTED	GATT 客户端：待订阅的特征不支持指示（Indiction）。

2.9 Controller 特性定义

[5]Core Specification Supplement, Part B, Common Profile and Service Error Codes

表 2.4: BLE 链路层特性定义

比特位置	链路层特性
0	LE Encryption
1	Connection Parameters Request
2	Extended Reject Indication
3	Slave-initiated Features Exchange
4	LE Ping
5	LE Data Packet Length Extension
6	LL Privacy
7	Extended Scanner Filter Policies
8	LE 2M PHY
9	Stable Modulation Index - Transmitter
10	Stable Modulation Index - Receiver
11	LE Coded PHY
12	LE Extended Advertising
13	LE Periodic Advertising
14	Channel Selection Algorithm #2
15	LE Power Class 1
16	Minimum Number of Used Channels Procedure
17	Connection CTE Request
18	Connection CTE Response
19	Connectionless CTE Transmitter
20	Connectionless CTE Receiver
21	Antenna Switching During CTE Transmission (AoD)
22	Antenna Switching During CTE Reception (AoA)
23	Receiving Constant Tone Extensions
24	Periodic Advertising Sync Transfer - Sender
25	Periodic Advertising Sync Transfer - Recipient
26	Sleep Clock Accuracy Updates
27	Remote Public Key Validation
28	Connected Isochronous Stream - Master
29	Connected Isochronous Stream - Slave
30	Isochronous Broadcaster
31	Synchronized Receiver

比特位置	链路层特性
32	Isochronous Channels (Host Support)
33	LE Power Control Request
34	LE Power Control Request
35	LE Path Loss Monitoring
36	Periodic Advertising ADI
37	Connection Subrating
38	Connection Subrating (Host Support)
39	Channel Classification

两个比特位置 33 和 34 合并表示 LE 功控请求特性，只能同为 0 或同为 1。这是因为蓝牙规范 5.2 的特性定义有误，5.3 加以修正，只得以两个比特表示同一特性。

2.10 蓝牙规范版本编号

表 2.5: 蓝牙链路层协议版本号

编号	蓝牙链路层协议版本号
6	4.0
7	4.1
8	4.2
9	5.0
10	5.1
11	5.2
12	5.3
13	5.4

2.11 白名单

在广播、扫描或者建立连接时，都可能用到白名单。操作白名单的 API 共有 3 个：

1. `gap_clear_white_lists`：清空白名单
2. `gap_add_whitelist`：添加一个设备地址
3. `gap_remove_whitelist`：删除一个设备地址

2.12 异步特性

Host API 绝大多数都是异步非阻塞操作，比如调用 `gap_set_ext_adv_enable()` 并不立即使能广播：这个函数只会给 Controller 发送一条 HCI 消息[6]，Controller 接收消息、完成处理之后才会真正开始广播。

这种异步特性可能使得实现某些功能的代码冗长、零散，开发者可以考虑使用其它语言[7]，或者重新封装 Host API，使其变为同步操作（参考 “同步版 API” 一节）。

2.13 线程安全性

Host 是线程**不安全**的，除下列函数之外的所有 API，如无特殊说明都必须在 Host 任务的上下文内调用。当其它任务需要调用 Host API 时，必须触发一段处于 Host 任务上下文内的代码，再在这段代码里调用 Host API。下列函数恰是为该功能设置：

- `btstack_push_user_msg`

  通过 `btstack_push_user_msg` 向 Host 发送一条消息，消息的回调处理处于 Host 任务上下文内，在这里可以调用 Host API。

- `btstack_push_user_runnable`

  通过 `btstack_push_user_runnable` 向 Host 发送一个可执行体（函数指针及参数），Host 任务在其上下文内运行这个可执行体。这个可执行体可以自由调用 Host API。SDK 提供的工具模块 `btstack_mt.c` 利用这个函数实现了一套 “线程安全的 API”。

开发者在其它任务里直接调用 Host API，即便发现功能正常，也仅是偶然现象，无法保证总是正常。

[6] 更准确地说，只是把一条 HCI 消息放入消息队列。

[7] https://ingchips.github.io/blog/2021-01-25-zig-async/

2.14 BLE 设备地址

BLE 设备地址长度为 6 字节，外加 1 个比特表示地址类型。BLE 规范定义了若干种地址类型：

1. 公共地址（Public Address，地址类型为 0）

   指从 IEEE 注册机构（IEEE Registration Authority）获得的全球唯一的 EUI-48 地址。

2. 随机地址（Random Address，地址类型为 1）

   以下几种随机地址类型通过最高的 2 个比特区分。

   1. 静态设备地址（最高 2 个比特为 0b11）

      可随机生成，可以每次上电后重新生成[8]，但是整个上电周期内不能改变。

   2. 私有地址

      共有两种私有地址。

      1. 可解析私有地址（最高 2 个比特为 0b01）
      2. 不可解析私有地址（最高 2 个比特为 0b00）

多数情况下四处广播设备的公共地址或者静态地址显然不是一个好主意，使用私有地址可有效地保护隐私。有了可解析地址的概念后，设备的公共地址、静态地址就从逻辑上变成身份地址（Identity Address）。

INGCHIPS 918xx/916xx 系列芯片没有公共地址，只能通过编程配置随机地址。

使用蓝牙地址时要注意字节顺序。协议栈遵循下面的基本规律：

- API 使用大端模式
- BLE 元事件使用小端模式[9]

使用 `reverse_bd_addr` 可以翻转地址的字节顺序：

[8] 地址改变后，曾与之配对的设备无法自动重连。

[9] 参照蓝牙核心规范。

```
void reverse_bd_addr(
    // 待翻转的地址
    const uint8_t *src,
    // 输出（不能与 src 相同）
    uint8_t * dest);
```

2.15 解析列表与隐私

当使用链路层隐私（LL Privacy）特性[10]时，需要用到解析列表。与之相关的 API 如下：

1. `gap_add_dev_to_resolving_list`：向列表中添加一条记录。一条记录包含 4 项数据：对端设备身份地址及地址类型，本端及对端的 IRK；
2. `gap_remove_dev_from_resolving_list`：删除列表中的一条记录；
3. `gap_clear_resolving_list`：清空列表；
4. `gap_read_peer_resolving_addr`：读取与某条记录关联的对端设备的可解析地址；
5. `gap_read_local_resolving_addr`：读取与某条记录关联的本端设备的可解析地址；
6. `gap_set_addr_resolution_enable`：使能或禁用地址解析功能；
7. `gap_set_resolvable_private_addr_timeout`：设置可解析地址超时时间。超过此时间后，Controller 会重新随机生成可解析地址；
8. `gap_set_privacy_mode`：设置某对端设备的隐私模式。

设备要保护自己的隐私，就必须保证不在任何广播 PDU 里发送其身份地址。使用解析列表里的某个记录时，如果本端 IRK 有效（即不是全部为 0），则本端使用可解析地址，隐私得到保护，反之则只能使用身份地址，泄露隐私；如果对端 IRK 有效，则只接受可解析地址，对端可在保护了隐私的前提下与本端正常通信，反之则只能接受、识别对端的身份地址，对端为了与本端通信必须泄露隐私。如果将隐私模式设置为 `PRIVACY_MODE_DEVICE`，那么当对端 IRK 有效时，Controller 仍会接受对端的身份地址。

也就是说，保证地址解析列表每条记录里的本端和对端的 IRK 都是有效的，并使能地址解析，可有效地保护隐私，详见链路层隐私。`PRIVACY_MODE_DEVICE` 仅可用于对端设备完全无法支持可解析地址的情况。

[10]ING918 芯片家族不支持此特性。

第三章　特性简析

本章从应用角度简要介绍核心规范自 5.0 以来引入的主要新特性。如需了解更多细节，请参考蓝牙核心规范。

3.1　5.0 新特性

3.1.1　新的 PHY

在旧规范中，GFSK 符号速率为 $1Msps$，即 1M PHY。新规范又定义了另外两种 PHY，2M 和 Coded。2M PHY 的符号速率加倍；与 1M PHY 相比，Coded 的符号速率也是 $1Msps$，但加入了 $1/2$ 卷积编码。直接传输卷积编码器的输出，称之为 S=2；经过 $1:4$ 比特映射后再传输，就是 S=8。

1M PHY 传输有效载荷的速率为 $1Mb/s$；2M PHY 传输有效载荷的速率为 $2Mb/s$；S=2 传输包头信息的速率为 $125kb/s$，传输有效载荷的速率为 $500kb/s$；S=8 传输包头信息的速率为 $125kb/s$，传输有效载荷的速率也为 $125kb/s$。

PHY 一般通过 `phy_type_t` 类型表示，这时将 S=2、S=8 统一表示为 Coded。当需要明确区分 S=2、S=8 时，则将它们视为单独的 PHY，与 1M、2M 并列。

用来做什么？

在良好的信道条件下，使用 2M PHY 可以获得更高的通信速率；在较差的信道条件下，Coded PHY 误包率相对较低，可获得略好的通信效果。

3.1.2　扩展广播

在旧规范中，广播只能使用 37、38、39 等 3 个信道。随着 BLE 设备的快速增长，这 3 个信道将变得越来越拥挤。于是新规范将广播 *扩展*到另外的 37 个信道，并将旧版广播称为传统广

播（Legacy advertising）。一个设备可以使用不同的广播参数、地址发送多组扩展广播，每组广播称为一个广播集。

扩展广播由一串广播 PDU 组成[1]，最多可携带 1650 字节的广播数据。第 1 个 PDU 仍位于 37、38、39 等 3 个信道，相同内容重复 3 次[2]。这些 PDU 里含有一种新的长度可变的包头，其中的 AuxPtr 用来通知扫描方何时、何地（信道）接收下一个 PDU，如同一个指向时频域的指针。AuxPtr 包含以下字段：

- 信道号（Channel index）
- 时钟精度（CA）
- 时间偏移的单位（Offset Unit）
- 时间偏移（AUX Offset）
- PHY（AUX PHY）

用来做什么？

相比传统广播，可以减少 37、38、39 等 3 个信道上的碰撞、干扰，传输更大的数据量。需要注意检查扫描方是否支持此特性。

3.1.3 周期广播

顾名思义，周期广播是一种时间上周期性出现的广播，同样可包含一串广播 PDU，第 1 个 PDU 严格地周期性发送，信道也按照既定参数进行跳频。这第一个 PDU 称为 AUX_SYNC_IND，其数据格式与队列中的其它 PDU 没有任何区别。

获得 AUX_SYNC_IND 参数并成功接收一次的过程称为周期广播的同步建立。规范提供两种建立同步的方法，一是借助与周期广播关联的扩展广播；一是基于连接进行 “周期广播同步信息传递”.

与周期广播关联的扩展广播的 PDU 队列里的最后一个 PDU 不含 AuxPtr，但是含有 SyncInfo，携带了关于如何周期性接收 AUX_SYNC_IND 的全部信息。

[1]称为一个 PDU 队列（train）。

[2]AuxPtr 里的时间偏移略有不同。

用来做什么？

用来周期性传递广播数据。

3.1.4 信道选择算法 #2

旧规范里的信道选择算法仅有 1 个参数，`hopIncrement`。假设上一个连接事件所使用的信道的原始编号[3]为 n，那么，本次连接事件所使用的信道的原始编号就是 $(n + hopIncrement) \bmod 37$。

这种跳频算法基本没有随机性，效果不佳。新的信道选择算法 #2 可粗略地写为

```
hash(AccessAddress, EventCounter);
```

即一种类似哈希的算法。

用来做什么？

更好地跳频、提高抗干扰能力。此特性完全于协议栈控制，不需要开发者参与。

3.2 5.1 新特性

3.2.1 CTE

定频扩展（CTE）用于实现寻向，可用于 1M 和 2M PHY，不可用于 Coded PHY。从基带看，CTE 即一连串的数字 1；从射频看，CTE 即一段频率固定为 $(F_t + f_d)$ 的电磁波，F_t 为载波频率，f_d 为 FSK 频率偏差。CTE 信号最长 $160\mu s$，相关信息由 CTEInfo 承载。

CTE 有两种操作方式：AoA 和 AoD。

- AoA：在接收端配置多个天线，估算单天线发送端的方向；
- AoD：在发送端配置多个天线，接收端使用单天线，采集 CTE 信号后计算方向。

[3]将原始编号再映射到当前的信道集合得到实际的信道号。

CTE 既通过周期广播发送，也可以在连接模式下按需发送。

用来做什么？

寻向，进而定位。

3.2.2 周期广播同步信息传递

假设设备 A 与设备 B 之间已经建立连接。如果设备 A 正在发送周期广播或者已经与其它设备发送的周期广播建立同步，现设备 B 也需要与这个周期广播建立了同步，那么设备 A 可以将周期广播的信息传递给设备 B，设备 B 根据这个信息直接开始接收 AUX_SYNC_IND，迅速建立同步。这一传递过程简称 PAST[4]，其关键在于以连接事件为参考，描述周期广播的定时信息。

用来做什么？

通过直接传递参数，加快对方的周期广播同步建立过程。不需要扫描与周期广播相关联的扩展广播。

3.3 5.2 新特性

3.3.1 LE Audio

引入了同步信道及 LC3 音频编码器。暂不支持。

3.3.2 增强的 ATT（EATT）

EATT 支持会话的并发，采用新的流控机制，更加稳健。暂不支持。

3.3.3 路径损耗监测与功率控制

详见 “路损检测与上报” 和 “功率控制”。

[4]Periodic Advertising Sync Transfer。

注意：核心规范 5.2 版本定义的功率控制存在缺陷，应以 5.3 版本为准。

用来做什么？

调整发射功率，兼顾通信质量和功耗。

3.4 5.3 新特性

3.4.1 减速模式

详见 “减速模式”。

用来做什么？

设计该特性的出发点是使用 LE Audio 时，减少常规连接所占用的射频资源。当连接的双方都没有数据需要发送时，减速模式可以减少射频活动，降低功耗；需要数据传输时，又可尽快恢复。

3.5 5.4 新特性

3.5.1 带响应的周期广播（PAwR）

PAwR 的设计目的是提供一种低功耗、双向、一对多的通信方式，适合电子价签等应用场景。与已有的周期广播（PADVB）相比，PAwR 有以下不同之处：

- PADVB 是单向通信，而 PAwR 的接收者可以向广播者发送响应包，是一种双向的、无连接的通信机制；
- PADVB 的同步信息都包含在 SyncInfo 内；而 PAwR 的同步信息除了 SyncInfo，还需要 ACAD 里的数据；
- PADVB 通过广播事件发送，而 PAwR 又定义了子事件，接收发需要同步一个或者多个子广播事件；
- PADVB 里的数据一般变化不频繁；而 PAwR 支持数据频繁变化；

- PAwR 可以给不同的接收者发送不同的数据；
- 对于 PAwR，PAST 是必选项。

如图 3.1 所示，PAwR 将 PADVB 的一个广播周期拆分为多个子事件，每个子事件包含 1 个数接包[5]，及多个响应时隙。为了确定它们之间的定时关系，增加了子事件间隔、响应时隙延迟、响应时隙间隔、响应时隙数目等参数。

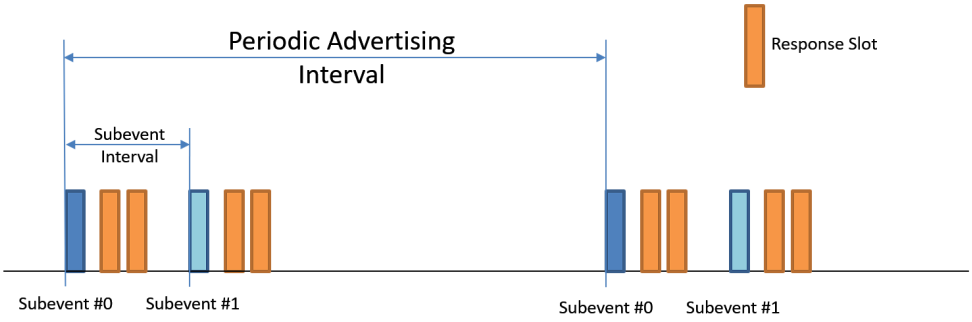

图 3.1: PAwR 子事件与响应

3.5.2 广播数据加密

旧版本的规范仅定义了连接模式下的加密，一直没有涉及广播数据的加密。这一新特性为加密广播数据提供了标准化方案。

广播数据的加密、鉴权沿用 CCM 算法，密钥信息通过新定义的 Encrypted Data Key Material 特征进行交换。加密过的数据再打包到现有的 “AD 结构” 里发送。

此特性可由 App 实现。

3.6 其它

3.6.1 广播时 Coded PHY 选择

当使用 Coded PHY 发送广播时，5.4 以前的 HCI 无法指定 S=2 或者 S=8。5.4 补上了这一 “漏洞”。SDK 一直支持[6] 通过 API `ll_set_adv_coded_scheme` 选择 S=2 或者 S=8。

[5]PAwR 每个子事件、每个响应都只包含一个数据包，不支持 PDU 队列。

[6]https://github.com/ingchips/ING918XX_SDK_SOURCE/blob/75cbf9928711c39e0b234ae11796bc1696111998/bundles/typical/inc/platform_api.h#L265

3.7 6.0 新特性

3.7.1 信道探测

蓝牙信道探测是蓝牙核心规范 6.0 版本中的一项重要创新，主要用于在两个蓝牙设备之间实现安全的定位。该技术测量精度高和安全性强，满足数字钥匙和资产追踪等应用中的高精度和安全性需求。

信道探测显著提升了测量精度，将之前的半米级别精度提升至 10 厘米以内。其核心原理包括两种测量方法：相位测距（PBR）和往返时间测距（RTT）。PBR 利用无线电波的相位特性，通过测量相位差和频率差来计算距离，但存在相位模糊问题，信道间隔 1 MHz 时模糊距离约为 150 米。RTT 则通过测量信号去程和返程时间差来计算距离，能够提供无模糊的长距离测量，但需要精确的计时机制。

为了确保安全性，信道探测采用了多种措施：随机化技术、信号处理防御、攻击检测系统以及 PBR 与 RTT 的交叉验证。这些安全措施有效防止了距离欺骗攻击和中间人攻击。

暂不支持。

3.7.2 基于决策的广告过滤（DBAF）

基于决策的广告过滤（Decision-Based Advertising Filtering, DBAF）旨在提高扫描设备在处理扩展广告时的效率。

DBAF 引入了一种新的扩展广告 PDU 类型，称为 ADV_DECISION_IND，非正式地称为决策 PDU。这种 PDU 旨在替代 ADV_EXT_IND 扩展广播 PDU，并在主信道上传输。

决策 PDU 包含应用程序指定的数据，使扫描器能够更明智地决定相关的 AUX_ADV_IND 辅助数据包是否值得关注。

即将支持。

3.7.3 广播者监控

这个特性所要解决的问题：当 Host 准备连接某个曾扫描到的设备时，这个设备是否还在可连接的范围内、是否已经远离？

这个特性增加了 HCI 命令和事件，用来命令 Controller 监控一个或多个广播集，并上报进入或者离开事件。

即将支持。

3.7.4 ISOAL 增强

ISOAL 引入了一种新的帧模式，能显著降低时间敏感型应用的延迟。

暂不支持。

3.7.5 链路层扩展特性集

随着规范发展，功能日益复杂，原有的特性集合已经用尽。为此引入了扩展特性集以表示更多的特性。

即将支持。

3.7.6 帧间隔更新

一直以来，相邻两个数据包之间的间隔 T_IFS 固定为 $150\mu s$。现在，规范定义一套 T_IFS 更新机制，允许连接双方协商在 $50\mu s$ $1000\mu s$ 范围内更新该参数。对于高吞吐率应用，可以缩小 T_IFS；而对于低速、低功耗应用，可以扩大 T_IFS。

暂不支持。

第四章　GAP - 广播

4.1　概览

支持 4.0 ~ 5.1 规范定义的所有 BLE 广播类型：

- 传统广播（Legacy Adv）
- 扩展广播（Extended Adv）
- 周期广播（Periodic Adv）

传统广播的有效载荷最长为 $31B$，而扩展广播（包括周期广播）每个广播包的有效载荷最长接近 $255B$，每个扩展广播又可包含多个广播包（广播包链条），总有效载荷最长达 $1650B$。

4.1.1　类型

广播有几种不同的属性：

- 可连接：接受对方发来的连接建立请求
- 可扫描：接受对方发来的扫描请求，并回复扫描响应
- 定向：只用于可连接广播，只接受特定方发来的连接建立请求
- 高占空比（High Duty）：以更高的频率[1]重复发送广播数据，常用于实现快速重连（定向可连接广播），最长只持续 $1.28s$。从 5.0 开始，高占空比广播也可用于不可连接广播。

对于传统的扫描响应包，其有效载荷最长同样为 $31B$；扩展的扫描响应包，其有效载荷最长同样为 $1650B$。开发者可以要求协议栈收到扫描请求时上报事件。

[1]广播间隔小于 $3.75ms$。

总结起来，传统广播共有 5 种类型，见表 4.1。

表 4.1: 传统广播类型

类型	PDU 类型	广播数据	扫描响应数据
非定向可连接可扫描广播	ADV_IND	支持	支持
定向可连接广播（非高占空比）	ADV_DIRECT_IND	不支持	不支持
定向可连接广播（高占空比）	ADV_DIRECT_IND	不支持	不支持
非定向可扫描广播	ADV_SCAN_IND	支持	支持
非定向不可连接不可扫描广播	ADV_NONCONN_IND	支持	不支持

4.1.2 过滤策略

对于可连接或可扫描广播，可以只接受某些设备的连接建立请求或扫描请求，这就是所谓的过滤策略。BLE 定义了 4 种策略：

```
typedef enum adv_filter_policy
{
    // 接受所有的连接建立请求或扫描请求
    ADV_FILTER_ALLOW_ALL    = 0x00,
    // 只接受白名单内的扫描请求，接收所有的连接建立请求
    ADV_FILTER_ALLOW_SCAN_WLST_CON_ALL,
    // 只接受白名单内的连接建立请求，接收所有的扫描请求
    ADV_FILTER_ALLOW_SCAN_ALL_CON_WLST,
    // 只接受白名单内的连接建立请求和扫描请求
    ADV_FILTER_ALLOW_SCAN_WLST_CON_WLST
} adv_filter_policy_t;
```

4.1.3 PHY

对于扩展广播，既需要在主广播信道（37/38/39）上发送少量信息，也需要在其它信道（即辅广播信道）上发送，所以需要分别设置主、辅广播信道所使用的 PHY，其中主广播信道只能使用 1M、Coded 等两种 PHY，而辅广播信道 3 种 PHY 皆可。

4.1.4 广播集

从 5.0 开始，BLE 支持并发发送多个广播，每个广播称为一个广播集[2]，由广播集句柄指示。每个广播使用各自独立的参数，包括地址、广播类型、PHY、数据等。开发者可以为广播集指定一个 4 比特长的 SID。

4.1.5 相关事件

- `HCI_SUBEVENT_LE_ADVERTISING_SET_TERMINATED`

  一个广播集停止广播时,HCI 回调会收到 `HCI_SUBEVENT_LE_ADVERTISING_SET_TERMINATED` 事件。这个事件的触发条件如下：

  - 连接建立：此时 `status` 为 `0`，可读取广播句柄和连接句柄的对应关系；
  - 已达到预定的广播时长、次数，自动停止：此时 `status` 为 `0x43`；
  - 极端情况：Controller 无法完成任务处理：此时 `status` 为 `0xFE`。

  注意,高占空比广播停止时不上报此事件,而是上报 `HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLET` 事件。当未能建立连接高占空比广播超时时，`status` 置为 `0x3C`（定向广播超时）。

- `HCI_SUBEVENT_LE_SCAN_REQUEST_RECEIVED`

  开发者使能扫描请求指示后，HCI 回调会收到 `HCI_SUBEVENT_LE_SCAN_REQUEST_RECEIVED` 事件。

- `HCI_SUBEVENT_PRD_ADV_SUBEVT_DATA_REQ`

  发送 PAwR 时，通过该事件向上层应用请求设置子事件数据。该事件的内容为 `le_mete_event_prd_adv_subevent_data_req_t`。应用收到该事件后,可通过 `gap_set_periodic_adv_s` 为子事件设置数据。

- `HCI_SUBEVENT_PRD_ADV_RSP_REPORT`

  收到 PAwR 的响应时，HCI 回调会收到 `HCI_SUBEVENT_PRD_ADV_RSP_REPORT` 事件，其内容为 `le_mete_event_prd_adv_rsp_report_t`。

[2]在不引起混淆的前提下，本手册混用 “广播”、“广播集” 这两个名词。

4.2 使用说明

4.2.1 配置广播

主要用到4个函数,gap_set_adv_set_random_addr、gap_set_ext_adv_para、gap_set_ext_adv_data 和 gap_set_ext_scan_response_data，分别配置随机地址、参数、广播数据和扫描响应数据。参数最复杂的函数是 gap_set_ext_adv_para，其原型为：

```
uint8_t gap_set_ext_adv_para(
    // 广播集句柄
    const uint8_t adv_handle,
    // 属性比特组合
    const adv_event_properties_t properties,
    // 广播间隔
    const uint32_t interval_min,
    const uint32_t interval_max,
    // 使用的主广播信道比特组合（0x7 表示使用全部 3 个主广播信道）
    const adv_channel_bits_t primary_adv_channel_map,
    // 使用的地址类型（随机地址来自 gap_set_adv_set_random_addr）
    const bd_addr_type_t own_addr_type,
    // 设置定向广播的对端地址
    const bd_addr_type_t peer_addr_type,
    const uint8_t *peer_addr,
    // 过滤策略
    const adv_filter_policy_t adv_filter_policy,
    // 发射功率，单位为 dBm
    const int8_t tx_power,
    // 主信道 PHY
    const phy_type_t primary_adv_phy,
    // 是否允许跳过部分辅信道的发送（填 0 表示总是发送）
    const uint8_t secondary_adv_max_skip,
    // 辅信道 PHY
    const phy_type_t secondary_adv_phy,
    // 广播集 SID
    const uint8_t sid,
```

```
    // 使能扫描请求上报
    const uint8_t scan_req_notification_enable);
```

其中，properties 为以下比特的组合：

```
// 可连接广播
#define    CONNECTABLE_ADV_BIT        ...
// 可扫描广播
#define    SCANNABLE_ADV_BIT          ...
// 定向广播
#define    DIRECT_ADV_BIT             ...
// 高频广播
#define    HIGH_DUTY_CIR_DIR_ADV_BIT ...
// 传统广播
#define    LEGACY_PDU_BIT             ...
// 匿名广播
#define    ANONY_ADV_BIT              ...
// 包含发射功率
#define    INC_TX_ADV_BIT             ...
```

对于传统广播，比特组合必须符合表 4.1 的定义。对于扩展广播，不能既可连接又可扫描；不支持高占空比广播。匿名广播中不包含广播者的地址，所以称为 “匿名” 广播。附加 INC_TX_ADV_BIT 比特后，广播内自动包含发射功率，比在载荷内通过 AD 项 “0x0A - «Tx Power Level»” 发送开销更小。

Controller 执行 gap_set_ext_adv_para() 命令时，会重新初始化对应的广播集，数据需要重新设置。

广播数据以空口 PDU 的格式存放，重新配置参数后，PDU 格式可能改变。由于一个扩展广播集可能包含多个广播 PDU，Controller 难以逐个调整 PDU 的数据格式，所以采用了这种重新初始化、由上层应用重新配置数据的方案。

对于带广播数据的广播，使用 gap_set_ext_adv_data 设置广播数据：

```
uint8_t gap_set_ext_adv_data(
  // 广播集句柄
  const uint8_t adv_handle,
  // 数据总长度（对于扩展广播，最长可达 1650）
  uint16_t length,
  // 数据
  const uint8_t *data);
```

对于可扫描广播，使用 gap_set_ext_scan_response_data 设置扫描响应数据：

```
uint8_t gap_set_ext_scan_response_data(
  const uint8_t adv_handle,
  const uint16_t length,
  const uint8_t *data);
```

4.2.2 广播数据

使用 Wizard 里的广播数据编辑器可以方便地编辑数据[3]。广播数据编辑器同时可以生成一些常数，方便开发者编程修改广播数据。下面的例子把蓝牙地址的最末两个字节填充到设备名称的最后 4 个字符里。

1. 用广播数据编辑器生成初始数据（图 4.1）：

```
// 0x01 - «Flags»
2, 0x01,
0x06,

// 0x09 - «Complete Local Name»: name_xxxx
10, 0x09,
0x6E, 0x61, 0x6D, 0x65, 0x5F, 0x78, 0x78, 0x78,
0x78,
```

[3]请参阅 SDK 用户手册。

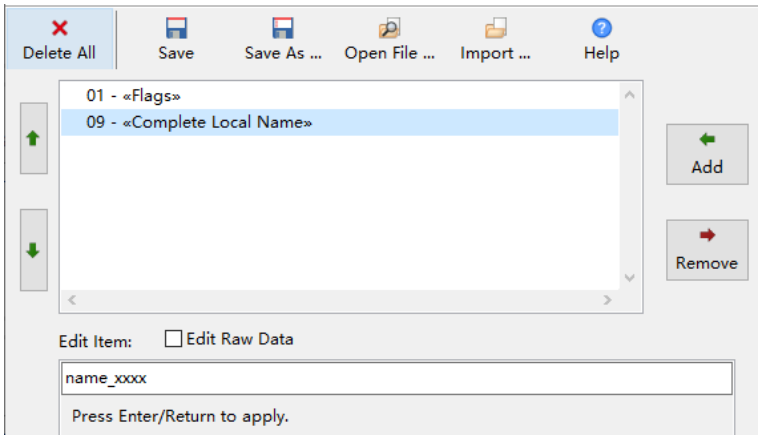

图 4.1: 用广播数据编辑器生成初始数据

```
// Total size = 14 bytes
```

2. 导入广播数据及常数：

```
static uint8_t adv_data[] = {
    #include "../data/advertising.adv"
};
// 这个文件里是编辑器生成的常数
#include "../data/advertising.const"
```

3. 修改广播名称

```
void assign_name(const uint8_t *id_bytes)
{
    char temp[5];
    sprintf(temp, "%02X%02X", id_bytes[0], id_bytes[1]);
    // ADVERTISING_ITEM_OFFSET_COMPLETE_LOCAL_NAME 是编译器自动生成的常数,
    // 表示 "name_xxxx" 在整个数据里的偏移位置
    memcpy(adv_data + ADVERTISING_ITEM_OFFSET_COMPLETE_LOCAL_NAME + 5,
           temp, sizeof(temp) - 1);
}
```

```
// 假设地址存放于 rand_addr
assign_name(&rand_addr[4]);
```

4.2.3 配置周期广播

周期广播总是与一个不可连接、不可扫描的扩展广播绑定。使用 gap_set_ext_adv_para 设置了扩展广播参数后，就可以通过 gap_set_periodic_adv_para 创建相关联的周期广播：

```
uint8_t gap_set_periodic_adv_para(
  // 使用同一个广播集句柄
  const uint8_t adv_handle,
  // 广播周期
  const uint16_t interval_min,
  const uint16_t interval_max,
  // 属性（仅支持 0 或 PERIODIC_ADV_BIT_INC_TX)
  const periodic_adv_properties_t properties);
```

周期广播的数据通过 gap_set_periodic_adv_data 设置，而不是 gap_set_ext_adv_para。

4.2.4 起停广播

通过 gap_set_ext_adv_enable 控制多个广播集的使能、停止状态。

```
uint8_t gap_set_ext_adv_enable(
  // 使能还是停止？
  const uint8_t enable,
  // 广播集数目
  const uint8_t set_number,
  // 每个广播集的使能参数
  const ext_adv_set_en_t *adv_sets);
```

这个函数支持一种快速停止所有广播的用法：gap_set_ext_adv_enable(0, 0, NULL)。除此以外，都需要用 adv_sets 数组表明每个广播集的句柄。

对于使能广播的情况，adv_sets 使用另外两个参数用来控制广播次数：

```
typedef struct ext_adv_set_en
{
    uint8_t handle;
    // 广播持续时间，单位为 10ms。0ms 表示一直广播
    uint16_t duration;
    // 最大广播次数。0 表示一直广播
    uint8_t max_events;
} ext_adv_set_en_t;
```

当 duration 或 max_events 条件满足时，广播就会自动停止。

4.2.5 起停周期广播

周期广播需要使用 gap_set_periodic_adv_enable 控制使能、停止状态：

```
uint8_t gap_set_periodic_adv_enable(
  const uint8_t enable,
  const uint8_t adv_handle);
```

要“完整”地开启周期广播，需要先通过 gap_set_ext_adv_enable 使能关联的扩展广播，再用这个 API 使能周期广播。扩展广播可以独立地关闭[4]。

4.2.6 为周期广播添加 CTE

参考“基于周期广播的 CTE 接收和发送”一节。

4.2.7 带响应的周期广播（PAwR）

4.2.7.1 配置

PAwR 广播也总是与一个不可连接、不可扫描的扩展广播绑定。使用 gap_set_ext_adv_para 设置了扩展广播参数后，就可以通过 gap_set_periodic_adv_para_v2 创建相关联的 PAwR 广

[4]关闭之后，其它设备无法再与该周期广播建立同步。

播，与 gap_set_periodic_adv_para 相比，增加了几个关于 PAwR 的参数：

```
uint8_t gap_set_periodic_adv_para_v2(
  // 使用同一个广播集句柄
  const uint8_t adv_handle,
  // 广播周期
  const uint16_t interval_min,
  const uint16_t interval_max,
  // 属性（仅支持 0 或 PERIODIC_ADV_BIT_INC_TX)
  const periodic_adv_properties_t properties,
  // 子事件个数
  const uint8_t num_subevents,
  // 子事件间隔
  const uint8_t subevent_interval,
  // 第一个响应时隙的延迟
  const uint8_t response_slot_delay,
  // 响应时隙的间隔
  const uint8_t response_slot_spacing,
  // 每个子事件的响应时隙数
  const uint8_t num_response_slots);
```

4.2.7.2 为子事件设置、更新数据

当收到 HCI_SUBEVENT_PRD_ADV_SUBEVT_DATA_REQ 事件时，通过 gap_set_periodic_adv_subevent_data 为子事件设置数据：

```
uint8_t gap_set_periodic_adv_subevent_data(
  // PAwR 的广播集句柄
  uint8_t adv_handle,
  // 要设置的子事件的个数
  uint8_t num_subevents,
  // 每个子事件的数据
  const gap_prd_adv_subevent_data_t *data);
```

每个子事件的数据定义如下：

```
typedef struct
{
    // 子事件序号
    uint8_t         subevent;
    // 本子事件的第一个响应时隙的序号
    uint8_t         rsp_slot_start;
    // 本子事件的响应时隙的个数
    uint8_t         rsp_slot_count;
    // 数据长度
    uint8_t         data_len;
    // 数据
    const uint8_t * data;
} gap_prd_adv_subevent_data_t;
```

这里设置的响应时隙应为 gap_set_periodic_adv_para_v2 所设置的响应时隙的子集。

4.2.7.3 从 PAwR 发起连接

通过 gap_ext_create_connection_v2 在指定的子事件上向指定设备发送连接请求。gap_ext_create_connection_v2 是gap_ext_create_connection 的增强版本，增加了 2 个参数：adv_handle 和 subevent。当这两个参数都是无效值（0xff）时，gap_ext_create_connection_v2 的功能等同于 gap_ext_create_connection。从 PAwR 发起连接时，这两个参数应该为有效值，此时，忽略 filter_policy，以及 phy_configs 里的 phy、scan_int、scan_win。

```
uint8_t gap_ext_create_connection_v2(
  // PAwR 的广播集句柄
  const uint8_t adv_handle,
  // 子事件序号
  const uint8_t subevent,
  const initiating_filter_policy_t filter_policy,
  const bd_addr_type_t own_addr_type,
  const bd_addr_type_t peer_addr_type,
```

```
    const uint8_t *peer_addr,
    const uint8_t initiating_phy_num,
    const initiating_phy_config_t *phy_configs
    );
```

第五章　GAP - 扫描

5.1　概览

接收广播的过程称为扫描。

5.1.1　间隔与窗口

接收机实际进行扫描工作的时机受扫描间隔和窗口两个参数控制，其含义如图 5.1 所示。

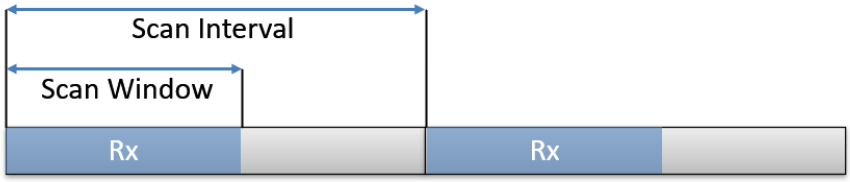

图 5.1: 扫描间隔与扫描窗口

注意：Controller 在执行扫描任务时，遵循最大努力原则，实际真正用于扫描的时机可能并不与两个参数所指定的完全一致。

5.1.2　过滤策略

与广播的 “过滤策略” 类似，扫描也有几种过滤策略：

```
typedef enum scan_filter_policy
{
    // 接收所有广播（定向到其它设备的除外）
    SCAN_ACCEPT_ALL_EXCEPT_NOT_DIRECTED,
```

```
    // 只接收来自白名单内的设备的广播（定向到其它设备的除外）
    SCAN_ACCEPT_WLIST_EXCEPT_NOT_DIRECTED,
    // 接收所有广播（所定向的设备与本设备身份地址不同的除外）
    // SCAN_ACCEPT_ALL_EXCEPT_IDENTITY_NOT_MATCH,
    // 只接收来自白名单内的设备的广播（所定向的设备与本设备身份地址不同的除外）
    // SCAN_ACCEPT_WLIST_EXCEPT_IDENTITY_NOT_MATCH
} scan_filter_policy_t;
```

后两种策略[1]与前两种策略的区别在于判断是否接受定向广播的方法不同。前两种策略只接受满足以下两者之一的定向广播：

1. TargetA 与本设备的地址一致，
2. TargetA 是可解析地址，且可由本设备的 IRK 解析。

而后两种策略接受满足以下两者之一的定向广播：

1. TargetA 与本设备的地址一致，
2. TargetA 是可解析地址。

可见后两种策略对 TargetA 的要求略微宽松。

5.1.3 主动与被动

所谓被动扫描（Passive Scan）指只接收广播数据包，对于可扫描广播也是如此；所谓主动扫描（Active Scan）指除了接收广播数据包之外，对于可扫描广播会主动发送扫描请求并接收扫描响应包。

5.1.4 PHY

由于扩展广播可在主广播信道使用两种 PHY 发送，相应地，扫描时也需要配置扫描哪种 PHY。

[1]后两种策略在规范中称为 Extended filter policy。ING916 和 ING918 不支持后两种策略。

5.2 使用说明

5.2.1 配置参数

在开始扫描之前，需要先通过 gap_set_random_device_address 为设备配置地址，这个地址用于扫描、发起连接等场景。使用 gap_set_ext_scan_para 配置扫描参数：

```
uint8_t gap_set_ext_scan_para(
  // 本设备地址类型
  const bd_addr_type_t own_addr_type,
  // 过滤策略
  const scan_filter_policy_t filter,
  // PHY 配置个数
  const uint8_t config_num,
  // 关于每种 PHY 的参数配置
  const scan_phy_config_t *configs);
```

这里的 PHY 指的是主广播信道所使用的 PHY，只能为 1M 或 Coded。辅广播信道上所有的 PHY 类型全部支持，不需要配置。每种 PHY 的参数配置如下：

```
typedef struct scan_phy_config
{
    // PHY
    phy_type_t phy;
    // 扫描方式：主动或被动
    scan_type_t type;
    // 扫描间隔，单位是 625us
    uint16_t interval;
    // 扫描窗口，单位是 625us
    uint16_t window;
} scan_phy_config_t;
```

当连接、扫描并发时，由于协议栈倾向于优化传输速率，将明显降低扫描任务的调度。需要更多地调度扫描任务时，可调用 LL API `ll_hint_on_ce_len`[2] 提示 Controller 减小连接事件的长度。

5.2.2 起停扫描

使用 `gap_set_ext_scan_enable` 起停扫描：

```
uint8_t gap_set_ext_scan_enable(
  // 开始或者停止扫描
  const uint8_t enable,
  // 是否对数据做去重处理:
  // 0 - 不去重;
  // 1 - 去重;
  // 2 - 去重，但是每个周期复位过滤器
  const uint8_t filter,
  // 持续时间，单位为 10ms
  const uint16_t duration,
  // 周期，单位为 1.28s
  const uint16_t period);
```

`duration` 和 `period` 两个参数的目的是实现每个 `period` 里扫描 `duration` 长的时间。另有两种特殊情况，从开发者的角度总结如下：

1. 持续、一直地扫描：`duration` 和 `period` 两个参数皆置为 0
2. 只扫描一段时间：`duration` 置为扫描时长，`period` 置为 0

Controller 在做数据去重时遵循最大努力原则，受限于存储空间、处理能力，去重可能失效。

5.2.3 处理数据

收到广播包后，HCI 回调函数将收到 `HCI_SUBEVENT_LE_EXTENDED_ADVERTISING_REPORT` 事件。利用 `decode_hci_le_meta_event` 宏可将事件转换为 `le_ext_adv_report_t` 结构体指针：

```
const le_ext_adv_report_t *report =
  decode_hci_le_meta_event(packet, le_meta_event_ext_adv_report_t)->reports;
```

这个结构体的定义如下：

```
typedef struct le_ext_adv_report
{
    // 事件类型比特位组合
    uint16_t        evt_type;
    // 广播者地址类型
    bd_addr_type_t  addr_type;
    // 广播者地址
    bd_addr_t       address;
    // 主信道上用的 PHY
    uint8_t         p_phy;
    // 辅信道上用的 PHY
    uint8_t         s_phy;
    // SID
    uint8_t         sid;
    // 发射功率（单位 dBm）
     int8_t         tx_power;
    // RSSI （单位 dBm）
     int8_t         rssi;
    // 周期广播的间隔（仅对周期广播有效）
    uint16_t        prd_adv_interval;
    // 定向广播的目的地址类型
    bd_addr_type_t  direct_addr_type;
    // 定向广播的目的地址
    bd_addr_t       direct_addr;
    // 广播数据长度
    uint8_t         data_len;
    // 广播数据
    uint8_t         data[0];
} le_ext_adv_report_t;
```

5.2.4 与周期广播同步

发现周期广播后，可以通过 gap_periodic_adv_create_sync 与周期广播同步[3]：

```
uint8_t gap_periodic_adv_create_sync(
  // 过滤策略：目标地址来自白名单还是参数
  const periodic_adv_filter_policy_t filter_policy,
  // SID
  const uint8_t adv_sid,
  // 目标地址类型
  const bd_addr_type_t addr_type,
  // 目标地址
  const uint8_t *addr,
  // 成功接收一次之后，可跳过的数目
  const uint16_t skip,
  // 同步超时（单位 100ms）
  const uint16_t sync_timeout,
  // 周期广播里的 CTE 类型（如果存在）
  const uint8_t sync_cte_type
);
```

成功建立同步后，HCI 回调会收到 HCI_SUBEVENT_LE_PERIODIC_ADVERTISING_SYNC_ESTABLISHED 事件。接收到的周期广播通过 HCI_SUBEVENT_LE_PERIODIC_ADVERTISING_REPORT 事件上报，其内容与 le_ext_adv_report_t 类似。

对于 5.4，HCI 回调会收到 HCI_SUBEVENT_LE_PERIODIC_ADVERTISING_SYNC_ESTABLISHED_V2 事件。与 HCI_SUBEVENT_LE_PERIODIC_ADVERTISING_SYNC_ESTABLISHED 相比，事件内容增加了子事件的相关信息。

周期广播的接收可参考 SDK *Periodic Scanner*。

5.2.5 与 PAwR 同步

当 HCI_SUBEVENT_LE_PERIODIC_ADVERTISING_SYNC_ESTABLISHED_V2 携带的子事件信息显示存在存在子事件时，可通过 gap_set_periodic_sync_subevent 与指定的子事件同步：

[3]即周期性地接收周期广播。

```
uint8_t gap_set_periodic_sync_subevent(
  // 同步
  uint16_t sync_handle,
  // 发送响应时的属性
  // 只能为 0 或者 PERIODIC_ADV_BIT_INC_TX
  uint16_t periodic_adv_properties,
  // 要同步的子事件数
  uint8_t num_subevents,
  // 要同步的每个子事件的序号
  const uint8_t *subevents)
```

与子事件同步后,接收到的每个子事件的数据将通过 HCI_SUBEVENT_LE_PERIODIC_ADVERTISING_REPORT 事件上报，其内容与 HCI_SUBEVENT_LE_PERIODIC_ADVERTISING_REPORT 类似，只是增加了关于子事件的信息。

5.2.5.1 设置、发送响应

通过 gap_set_periodic_adv_rsp_data 在指定的子事件、响应时隙发送响应数据：

```
uint8_t gap_set_periodic_adv_rsp_data(
  // 同步句柄
  uint16_t sync_handle,
  // 所要响应的 PAwR 所在的周期广播事件序号
  uint16_t request_event,
  // 所要响应的 PAwR 所在的子事件序号
  uint8_t request_subevent,
  // 发送响应的子事件序号
  uint8_t rsp_subevent,
  // 发送响应的时隙序号
  uint8_t rsp_slot,
  // 响应的长度
  uint8_t rsp_data_len,
  // 响应数据
  const uint8_t *rsp_data);
```

第六章　GAP - 连接

6.1　概览

连接管理功能也由 GAP 模块提供，包含建立连接、断开连接、读取对端版本、减速模式、功率控制等。

6.2　使用说明

6.2.1　建立连接

建立连接的过程与主动扫描有相似之处：持续尝试接收某种类型的广播，主动发出一个请求。所以，蓝牙协议规定不要并发地执行这两项任务：建立连接前要先停止扫描，反之亦然。

在建立连接之前，需要先通过 gap_set_random_device_address 为设备配置地址，这个地址用于扫描、发起连接等场景。通过 gap_ext_create_connection 建立连接。所连接的目标可以是一个特定的地址，也可以是白名单中的任意一个地址[1]。

```
uint8_t gap_ext_create_connection(
  // 过滤策略：目标地址来自参数还是白名单？
  const initiating_filter_policy_t filter_policy,
  // 本设备地址类型
  const bd_addr_type_t own_addr_type,
  // 目标地址来自参数时，指定目标地址类型
  const bd_addr_type_t peer_addr_type,
  // 目标地址来自参数时，指定目标地址
```

[1]需要连接多个设备，使用白名单方式效率更高。

```
  const uint8_t *peer_addr,
  // 主广播信道的配置个数
  const uint8_t initiating_phy_num,
  // 主广播信道的配置
  const initiating_phy_config_t *phy_configs);
```

另见从 PAwR 发起连接。

主广播信道的配置 initiating_phy_config_t 指定了每种主广播信道 PHY 的扫描参数（此部分与扫描参数 scan_phy_config_t 相同）及连接参数：

```
typedef struct {
    // 同 scan_phy_config_t
    uint16_t scan_int;
    uint16_t scan_win;
    // 最小连接间隔，单位 1.25ms
    uint16_t interval_min;
    // 最大连接间隔，单位 1.25ms
    uint16_t interval_max;
    // 从机延迟（即允许从机跳过多少个连接间隔）
    uint16_t latency;
    // LE 链路超时时间，单位 10ms
    uint16_t supervision_timeout;
    // 关于每个连接间隔内连接事件长度的提示信息，单位 0.625ms
    uint16_t min_ce_len;
    uint16_t max_ce_len;
} conn_para_t;

typedef struct initiating_phy_config
{
    phy_type_t phy;
    conn_para_t conn_param;
} initiating_phy_config_t;
```

关于每个连接间隔内连接事件长度的提示信息（min_ce_len 和 max_ce_len）不会被

传递给从端。Controller 可以借助这个信息更好地调度多种任务。从端 App 可调用 LL API `ll_hint_on_ce_len`[2] 将提示信息告知 Controller。

收到对应的 `HCI_EVENT_COMMAND_STATUS` 事件，并且 `status` 为 0，标志着开始执行连接建立任务。`HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE` 事件标志着连接建立任务的结束。也就是说每次调用这个函数将到达 3 个互斥的终点：

1. 函数返回值非 0；
2. 上报 `HCI_EVENT_COMMAND_STATUS` 事件，并且 `status` 非 0；
3. 上报 `HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE` 事件。

在上一次调用到达终点 3 前，再次调用 `gap_ext_create_connection` 会到达终点 1 或 2。

复杂应用（如多种蓝牙任务并发）中，务必响应 `HCI_EVENT_COMMAND_STATUS` 事件检查建立连接命令是否出错（即到达终点 2）。有时，Controller 会因为无法调度任务而上报 `status` 为 `0x07` 的 `HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE` 事件。对于这种情况，建议 App 延后一段时间再重新尝试建立连接。

6.2.2 取消连接

建立连接需要一定的时间，如果决定不再继续等待，可以通过 `gap_create_connection_cancel` 取消连接建立任务。任务取消后，同样会上报 `HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE` 事件，其中的 `status` 为 `0x02` （未知的连接句柄）。

6.2.3 获取对端版本

通过 `gap_read_remote_info` 可以读取对端协议栈版本。获得版本信息后 Controller 上报 `HCI_EVENT_READ_REMOTE_VERSION_INFORMATION_COMPLETE` 事件。版本的解析方法可参考 SDK *UART GATT Console*。

6.2.4 获取对端特性

通过 `gap_read_remote_used_features` 可以读取对端支持的 BLE 特性。获得特性信息后 Controller 上报 `HCI_SUBEVENT_LE_READ_REMOTE_USED_FEATURES_COMPLETE` 这个子事件。特性的解析方法可参考 SDK *UART GATT Console*。

[2]参考《Controller API Reference》。

6.2.5 设置 PHY

通过 gap_set_phy 可以设置偏好的 PHY。经过与对方的协商生效后，Controller 上报 HCI_SUBEVENT_LE_PHY_UPDATE_COMPLETE 事件。

gap_set_phy 参数详解：

```
uint8_t gap_set_phy(
  // 连接句柄
  const uint16_t con_handle,
  // 置起比特 0 表示在发送方向无偏好
  // 置起比特 1 表示在接收方向无偏好
  // 其它比特保留
  const uint8_t all_phys,
  // 发送方向上的 PHY 偏好（比特 0 为 0 有效）
  const phy_bittypes_t tx_phys,
  // 接收方向上的 PHY 偏好（比特 1 为 0 有效）
  const phy_bittypes_t rx_phys,
  // PHY 的其它选项
  const phy_option_t phy_opt);
```

PHY 偏好 phy_bittypes_t 是几个比特的组合：

表 6.1： PHY 比特组合

比特序号	含义
0	1M PHY
1	2M PHY
2	Coded PHY

phy_option_t 目前用来指示本端 Coded PHY 采用 S2 或 S8。对于对端，可以在对端 App 里调用 ll_set_conn_coded_scheme 选择 S2 或者 S8。默认为 S8。

6.2.6 更新连接参数

连接中的主从角色都可以使用 gap_update_connection_parameters[3]：主角色使用这个函数可以更新连接参数；从角色使用这个函数则是请求主端更新连接参数：

```
int gap_update_connection_parameters(
  // 连接句柄
  hci_con_handle_t con_handle,
  // 建议的最小连接间隔（单位 1.25ms）
  uint16_t conn_interval_min,
  // 建议的最大连接间隔（单位 1.25ms）
  uint16_t conn_interval_max,
  // 建议的从机延迟
  uint16_t conn_latency,
  // 建议的超时时间（单位 10ms）
  uint16_t supervision_timeout,
  // 关于每个连接间隔内连接事件长度的提示信息，单位 0.625ms
  //  （从角色忽略这两个参数）
  uint16_t min_ce_len,
  uint16_t max_ce_len);
```

事件 HCI_SUBEVENT_LE_CONNECTION_UPDATE_COMPLETE 标志着参数更新完成。

6.2.7 减速模式

减速模式的使用方法可参考 SDK *UART GATT Console*。

减速（Subrating）模式为中心设备和外围设备定义了一种统一的节奏，在保证通信的持续性前提下跳过若干连接间隔，减少射频占用、降低功耗。调用 gap_subrate_request[4] 即可在主从两端协商启动减速模式。

[3]对于 v8.2 以下版本，这个函数仅用于主角色，从角色需要使用 l2cap_request_connection_parameter_update 请求更新。

[4]调用之前建议先检查对方是否支持此特性。此 API 无论主从角色都可以调用。

```
uint8_t gap_subrate_request(
  // 连接句柄
  hci_con_handle_t con_handle,
  // 最小减速比
  uint16_t subrate_min,
  // 最大减速比
  uint16_t subrate_max,
  // 最大从延迟（单位：减速后连接间隔）
  uint16_t max_latency,
  // 最小连续传输次数
  uint16_t continuation_number,
  // 超时时间（单位 10ms）
  uint16_t supervision_timeout);
```

例如，将连接 0 的减速比设置为 8，在双方无数据传输时，每 8 个连接间隔对发 1 次空包维持连接：

```
gap_subrate_request(0, 8, 8,
  0, 0, 2000);
```

使用 BLE 空口抓包工具或者高端电流表可观察到这种减速行为，见图 6.1。

`continuation_number` 表示每个减速周期开始时，至少再连续传输多少个连接间隔。如果为 1，在上述配置下当双方无数据传输时，每 8 个连接间隔有 2 个激活；如果为 7，则每个连接间隔有 8 个激活，即都激活。

启用减速模式后，从机延迟的单位从原来的连接间隔变为 (连接间隔 × 减速比)。

减速参数更新后，HCI 回调函数会收到 `HCI_SUBEVENT_LE_SUBRATE_CHANGE` 事件。

当出现通信需求时，可以迅速从减速模式回到连接通信模式，**兼顾功耗与效率**，如图 6.2。

通过 `gap_set_default_subrate` 设置 Controller 所接受的减速模式参数范围。

```
uint8_t gap_set_default_subrate(
  uint16_t subrate_min,
```

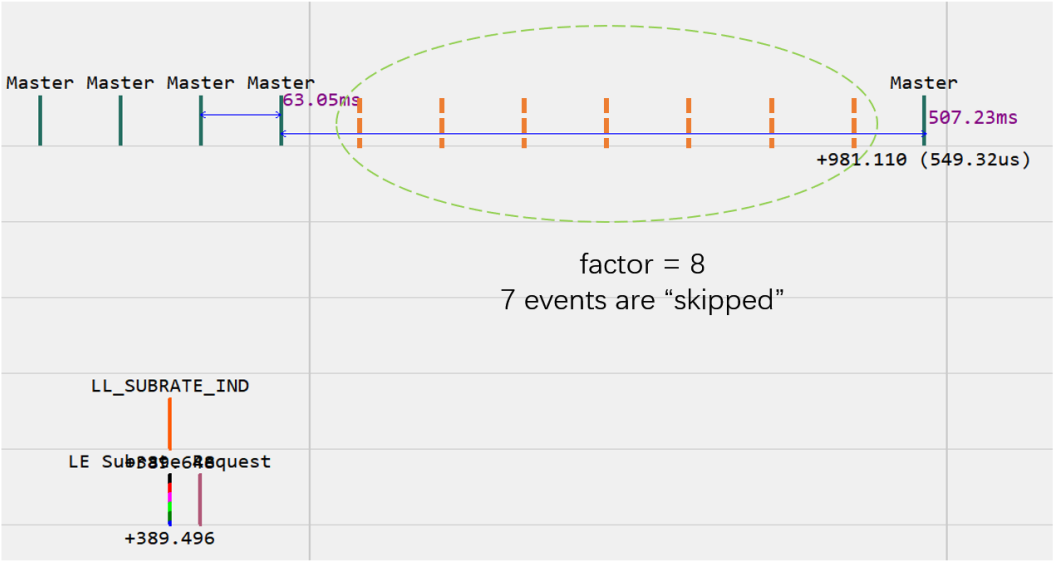

图 6.1: 减速比为 8 时的行为

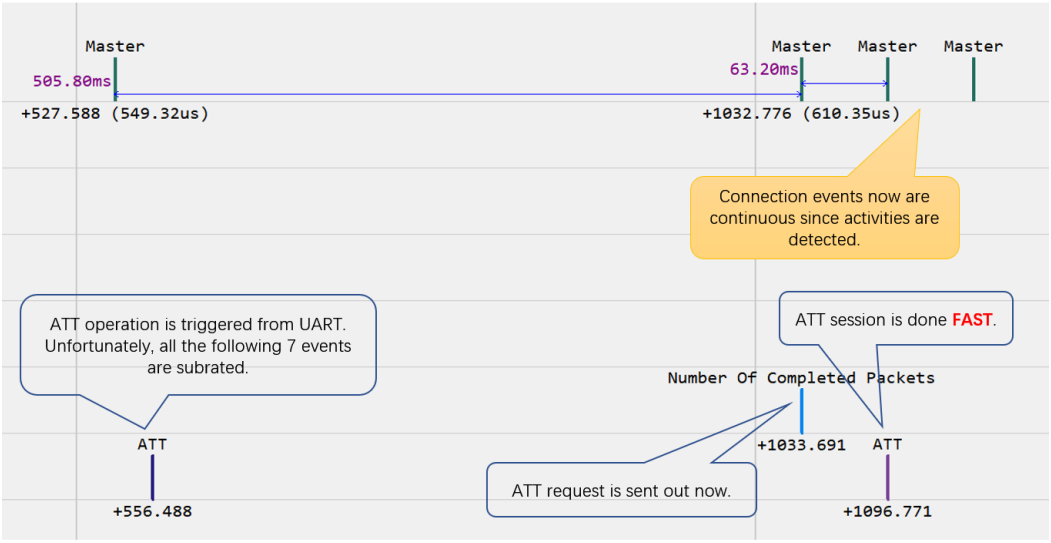

图 6.2: 减速下出现数据通信

```
uint16_t subrate_max,
uint16_t max_latency,
uint16_t continuation_number,
uint16_t supervision_timeout);
```

6.2.8 路损检测与上报

BLE 5.2 为路径损耗定义了 3 种分类（或者分区，zone），高损耗、中损耗和低损耗。Controller 监控损耗情况，当损耗分类发生改变时（图 6.3 中的虚线箭头），上报 HCI_SUBEVENT_LE_PATH_LOSS_THRESHOLD 事件。

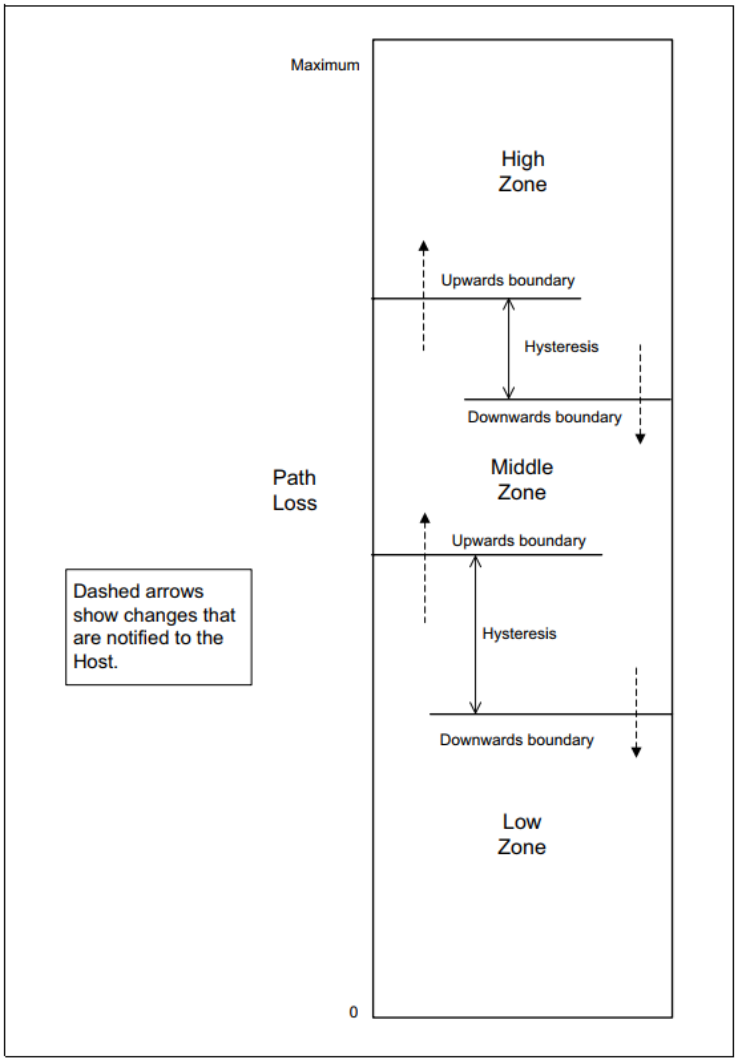

图 6.3: 路径损耗上报

- 配置路损分类参数：

```
uint8_t gap_set_path_loss_reporting_param(
  hci_con_handle_t con_handle,  // 连接句柄
  uint8_t high_threshold,       // 高路损门限
```

```
  uint8_t high_hysteresis,      // 高路损迟滞
  uint8_t low_threshold,        // 低路损门限
  uint8_t low_hysteresis,       // 低路损迟滞
  uint8_t min_time_spent);      // 最小停留时间（单位连接间隔）
```

图 6.3 中的 *upwards boundary* 为 (threshold + hysteresis)，*downwards boundary* 为 (threshold - hysteresis)。

- 使能上报：

```
uint8_t gap_set_path_loss_reporting_enable(
  hci_con_handle_t con_handle,  // 连接句柄
  uint8_t enable                // 使能
);
```

6.2.9 功率控制

功率控制的使用方法可参考 SDK *UART GATT Console*。

- 读取发射功率

  读取本端发射功率：

```
uint8_t gap_read_local_tx_power_level(
  hci_con_handle_t con_handle, // 连接句柄
  unified_phy_type_t phy       // PHY
);
```

从对应的 HCI_EVENT_COMMAND_COMPLETE 事件的返回参数中取得发射功率值：

```
uint8_t Status,               // 状态码
uint8_t Connection_Handle,    // 连接句柄
```

```
uint8_t PHY,
 int8_t Current_TX_Power_Level, // 当前发射功率（dBm）
 int8_t Max_TX_Power_Level      // 最大发射功率（dBm）
```

读取对端发射功率：

```
uint8_t gap_read_remote_tx_power_level(
  hci_con_handle_t con_handle, // 连接句柄
  unified_phy_type_t phy       // PHY
);
```

从 HCI_SUBEVENT_LE_TRANSMIT_POWER_REPORTING 事件中取得发射功率值。

- 设置发射功率

  设置本端发射功率：

```
void ll_set_conn_tx_power(
  uint16_t conn_handle, // 连接句柄
  int16_t tx_power      // 发射功率（dBm）
);
```

调整对端发射功率：

```
void ll_adjust_conn_peer_tx_power(
  uint16_t conn_handle, // 连接句柄
  int8_t delta          // 调整量，正值为增大，负值为减小（dB）
);
```

- 发射功率自动上报

```
uint8_t gap_set_tx_power_reporting_enable(
  hci_con_handle_t con_handle,  // 连接句柄
  uint8_t local_enable,         // 使能本端上报
  uint8_t remote_enable         // 使能对端上报
);
```

第七章　GATT - 服务器

7.1　概览

GATT 服务器[1]为客户端提供服务。协议栈支持多个连接，每个连接的配置（Profile）可以独立设置。需要注意，GATT 服务器和客户端这两个角色与主、从两个角色没有任何关联：一个连接的主角色既可以充当 GATT 的客户端，也可以充当服务器，还可以两种角色一起扮演；一个连接的从角色也是如此。

要使用 GATT 服务器，开发者需要做三件事[2]：

1. 初试化：设置事件回调

```
void att_server_register_packet_handler(
  btstack_packet_handler_t handler);
```

2. 初始化：提供回调函数

```
void att_server_init(
  // 特征的读回调
  att_read_callback_t read_callback,
  // 特征的写回调
  att_write_callback_t write_callback);
```

读回调的类型如下：

[1]在不引起混淆的前提下，本手册混用 ATT 服务器、GATT 服务器，代码里也用 `att_server` 代指 `gatt_server`。

[2]事实上，这几件事已由 Wizard 工具代劳。

```
typedef uint16_t (*att_read_callback_t)(
  // 连接句柄
  hci_con_handle_t con_handle,
  // 特征句柄
  uint16_t attribute_handle,
  // 数据偏移
  uint16_t offset,
  // 缓存
  uint8_t *buffer,
  // 缓存的大小
  uint16_t buffer_size);
```

写回调的类型如下：

```
typedef int (*att_write_callback_t)(
  // 连接句柄
  hci_con_handle_t con_handle,
  // 特征句柄
  uint16_t attribute_handle,
  // 会话模式
  uint16_t transaction_mode,
  // 数据偏移
  uint16_t offset,
  // 缓存
  const uint8_t *buffer,
  // 缓存的大小
  uint16_t buffer_size);
```

将 `con_handle` 和 `attribute_handle` 组合到一起，回调函数就可以确定是在访问哪个 Profile 里的哪个特征。对于长度超过 $(ATT\_MTU - 3)$ 的长值，BLE 支持分块读写模式，相应地，两个回调函数都有一个 `offset` 参数。

关于会话模式 `transaction_mode` 的说明见后文。

3. 适时提供 Profile 数据

```
void att_set_db(
  // 要关联的句柄
  hci_con_handle_t con_handle,
  // Profile 数据库
  const uint8_t *db);
```

协议栈内的 GATT 服务器可以通过 2 种方式获得特征的值，然后传输到客户端：

1. 保存在 Profile 数据库内部的值

   这种方式适用于值不改变的情况，服务器可能自动将值传输到客户端，不需要开发者参与。

2. 借助回调函数 `att_read_callback_t`

   这种方式适用于值动态改变的情况。每当客户端读取值时，协议栈会立即调用回调函数，且 `buffer` 为 `NULL`。这一次调用是为了获取数值长度。回调函数的处理流程又可以分为两种情况：

   - 如果 App 可以立即准备好数据，那么直接返回数值的**总长**；

     之后，协议栈准备内存空间并立即再次调用函数，此时 `buffer` 参数非 `NULL`，回调函数将数据写入 `buffer` 所指向的内存，读取完成；

   - 如果 App 无法立即准备好数据，那么返回 `ATT_DEFERRED_READ` 进入延迟读取模式；

     待数据就绪之后，App 调用 `att_server_deferred_read_response` 将数据传给协议栈，读取完成。

每个特性具有若干属性，见表 7.1。

表 7.1： 特征的属性

属性	说明
ATT_PROPERTY_BROADCAST	允许广播该特性的值
ATT_PROPERTY_READ	允许读取
ATT_PROPERTY_WRITE_WITHOUT_RESPONSE	允许无响应写入
ATT_PROPERTY_WRITE	允许（有响应的）写入
ATT_PROPERTY_NOTIFY	允许通知（Notification）
ATT_PROPERTY_INDICATE	允许指示（Indication）

属性	说明
ATT_PROPERTY_AUTHENTICATED_SIGNED_WRITE	允许带签名的写入
ATT_PROPERTY_EXTENDED_PROPERTIES	支持扩展属性
ATT_PROPERTY_DYNAMIC	为动态特性（即需要使用回调函数）

其中 `DYNAMIC` 为协议栈自定义的属性，只有加上了这个属性，对特性的读写操作才会交由回调函数处理。对于支持写入的特征，由于总是需要通过回调函数处理，必须加上此属性。

对于支持通知（Notification）和/或指示（Indication）的特征，必须带有 Client Characteristic Configuration 描述符（常被简称为 CCCD）：

- 将 `GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTIFICATION` 写入 CCCD 就可以使能通知，
- 将 `GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_INDICATION` 写入 CCCD 就可以使能指示，
- 将 `GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NONE` 写入 CCCD 就可以关闭通知和指示。

通知相当于无响应的上报，而指示相当于有响应的上报。

7.2 使用说明

7.2.1 Profile 数据

Profile 数据[3] 使用了协议栈自定义的数据结构。开发者不需要了解这个数据结构的定义，而是借助图形化工具或者编程地生成。

1. 使用图形化的 Profile 编辑器。

   请参考《SDK 用户手册》。

2. 使用 `att_db_util` 模块。

   调用 `att_db_util_init` 告知 Profile 数据的存储空间。调用 `att_db_util_add_service_uuid??` 添加服务，然后通过 `att_db_util_add_characteristic_uuid??` 添加若干特性。重复上

[3]在手册、工具、代码的不同位置可能使用了不同的名词，如 Profile 数据库、GATT 数据库、GATT 数据等。

述步骤可以添加多个服务。最后调用 `att_db_util_get_size` 查看整个 Profile 数据的大小，相应调整存储空间的大小，再重新编译程序。

上面的 `....._uuid??` 函数都包含 `uuid16` 和 `uuid128` 等两种形式，分别对应 16-bit 的简短 UUID 和 16 字节的完整 UUID。

`att_db_util_add_characteristic_uuid??` 函数返回的是特性的值的句柄。调用 `att_db_util_add_characteristic_uuid??` 时需要注意根据情况添加 `ATT_PROPERTY_DYNAMIC` 属性。对于带有 `ATT_PROPERTY_NOTIFY` 和/或 `ATT_PROPERTY_INDICATE` 属性的特性，`att_db_util_add_characteristic_uuid??` 函数会自动添加 CCCD。设特性的值句柄为 N，那么 CCCD 的句柄为 $N+1$。

7.2.2 实现读回调

一个典型的读回调函数大概是这种样子：

```
static uint16_t att_read_callback(hci_con_handle_t connection_handle,
  uint16_t att_handle, uint16_t offset,
  uint8_t * buffer, uint16_t buffer_size)
{
    switch (att_handle)
    {
    case HANDLE_0:
        if (buffer)
        {
            memcpy(buffer, ...)
            return buffer_size;
        }
        else
            return size of value;
    //...
    default:
        return 0;
    }
}
```

延迟读取的情况：

```
static uint16_t att_read_callback(hci_con_handle_t connection_handle,
  uint16_t att_handle, uint16_t offset,
  uint8_t * buffer, uint16_t buffer_size)
{
    switch (att_handle)
    {
    case HANDLE_0:
        //...
        return ATT_DEFERRED_READ;
    //...
    default:
        return 0;
    }
}
```

延迟读取的数据通过 att_server_deferred_read_response 传递给协议栈：

```
int att_server_deferred_read_response(
  // 连接句柄
  hci_con_handle_t con_handle,
  // 特征句柄
  uint16_t attribute_handle,
  // 指向值的指针
  const uint8_t *value,
  // 值的长度
  uint16_t value_len);
```

SDK *GATT Relay* 演示了延迟读取的具体用法。

7.2.3 实现写回调

当写特性时，可能触发服务器执行某个动作。BLE 为了精度控制动作的执行引入了会话[4]概念：一个会话包含对 1 个或多个特性的写入，最后是一个显式的 “执行” 命令通知服务器开始执行。此外还有一个 “取消” 命令，通知服务器会话不要执行动作、直接终止。相应地，写回调的会话模式 transaction_mode 参数包含四种值：

```
// 无会话的普通写入
#define ATT_TRANSACTION_MODE_NONE      ...
// 带会话的写入
#define ATT_TRANSACTION_MODE_ACTIVE    ...
// 执行
#define ATT_TRANSACTION_MODE_EXECUTE   ...
// 取消
#define ATT_TRANSACTION_MODE_CANCEL    ...
```

对于 NONE 和 ACTIVE 两种模式，都会传入要写入的数据，而 EXECUTE 和 CANCEL 则既不会传入特征句柄，也不传入数据，也就是说这两个命令只关联到连接，施加于整个服务器，而不针对某个特征。

一个典型的写回调函数大概是这种样子：

```
static int att_write_callback(
  hci_con_handle_t connection_handle, uint16_t att_handle,
  uint16_t transaction_mode,
  uint16_t offset, const uint8_t *buffer, uint16_t buffer_size)
{
    处理 EXECUTE 和 CANCEL 两种会话模式并返回;

    //
    switch (att_handle)
    {
    case HANDLE_0:
```

[4]协议栈里称为 “会话”，规范里称为 “队列”。

```
        //...
        return 0;
    //...
    default:
        return 0;
    }
}
```

如果发现错误，可以返回非 0 值告知客户端[5]。

7.2.4 发送通知（Notification）

通过 att_server_notify 发送通知：

```
int att_server_notify(
  // 连接句柄
  hci_con_handle_t con_handle,
  // 特征句柄
  uint16_t attribute_handle,
  // 指向值的指针
  const uint8_t *value,
  // 值的长度
  uint16_t value_len);
```

该函数返回的状态代码及含义如下：

- 0：数据正常进入发送队列；
- BTSTACK_LE_CHANNEL_NOT_EXIST (0x59)：参数 con_handle 指定的连接不存在；
- BTSTACK_ACL_BUFFERS_FULL (0x57)：内部缓存已满，数据未进入发送队列[6]。

[5]仅适用于有响应的写入，无响应的写入无效。

[6]解决方法参考L2CAP 传输队列。

7.2.5 发送指示（Indication）

通过 att_server_indicate 发送指示：

```
int att_server_indicate(
  // 连接句柄
  hci_con_handle_t con_handle,
  // 特征句柄
  uint16_t attribute_handle,
  // 指向值的指针
  const uint8_t *value,
  // 值的长度
  uint16_t value_len);
```

指示的响应通过 ATT_EVENT_HANDLE_VALUE_INDICATION_COMPLETE 事件通知 App，事件里的状态码的定义见ATT 错误码。

该函数返回的状态代码及含义如下：

- 0：数据正常进入发送队列；
- BTSTACK_LE_CHANNEL_NOT_EXIST (0x59)：参数 con_handle 指定的连接不存在；
- BTSTACK_ACL_BUFFERS_FULL (0x57)：内部缓存已满，数据未进入发送队列[7]；
- ATT_HANDLE_VALUE_INDICATION_IN_PROGRESS (0x90):在收到 ATT_EVENT_HANDLE_VALUE_INDICATION_C 事件前，不能发送新的指示，否则 att_server_indicate 将返回此错误码。

7.2.6 响应事件

GATT 服务器模块会弹出以下事件。

- ATT_EVENT_MTU_EXCHANGE_COMPLETE

  这个事件表示 MTU 协商完成。有两个解析函数：

  - att_event_mtu_exchange_complete_get_handle(packet) 获得连接句柄；
  - att_event_mtu_exchange_complete_get_MTU(packet) 获得 MTU 大小。

[7]解决方法参考L2CAP 传输队列。

- ATT_EVENT_HANDLE_VALUE_INDICATION_COMPLETE

  这个事件表示指示发送完成。有三个解析函数：

  - att_event_handle_value_indication_complete_get_conn_handle(packet) 获得连接句柄；
  - att_event_handle_value_indication_complete_get_attribute_handle(packet) 获得特征句柄；
  - att_event_handle_value_indication_complete_get_status(packet) 获得响应状态。

    共有两种状态：0 表示成功送达，ATT_HANDLE_VALUE_INDICATION_TIMEOUT (0x91) 表示超时。

7.2.7 ATT_MTU

除了通过监听 ATT_EVENT_MTU_EXCHANGE_COMPLETE 事件以外，通过 att_server_get_mtu 可以随时获取 ATT_MTU：

```
uint16_t att_server_get_mtu(
  // 连接句柄
  hci_con_handle_t con_handle);
```

特征的值的最大长度为 (ATT_MTU-3)。调用 att_server_indicate 或 att_server_notify 上报数据时，如果超过最大长度，会被自动截短。

第八章　GATT - 客户端

8.1　概览

GATT 客户端的主要功能是发现服务，读写和订阅特征。每个连接使用独立的 GATT 客户端实例。

GATT 客户端的操作几乎都需要先发出数据再等待服务器发回响应，需要较长的时间，所以也引入了会话的概念：如果上一个会话未结束，那么就不允许发起新的会话。每次发起会话时，都要提供一个专门的回调函数。当这个回调函数收到 `GATT_EVENT_QUERY_COMPLETE` 事件时，会话结束。

使用 GATT 客户端 API 时需要注意检查返回值，常见的返回值有：

- 0：正常、成功；
- `BTSTACK_MEMORY_ALLOC_FAILED` (`0x56`)：内存不足，无法创建 GATT 客户端实例
- `GATT_CLIENT_IN_WRONG_STATE` (`0x95`)：会话冲突

  解决方法：等待上一个会话完成后再发起新的会话。通过 `gatt_client_is_ready` 可以查询当前是否可以发起新的会话。
- `GATT_CLIENT_VALUE_TOO_LONG` (`0x97`)：要写入的值超出 MTU 的限制

  解决方法：缩短值的长度；或者检查对比设备的 MTU 能力。
- `BTSTACK_ACL_BUFFERS_FULL` (`0x57`)：内部缓存已满，数据未进入发送队列

  解决方法：参考L2CAP 传输队列。

可选地，开发者可以设置 GATT 客户端事件回调：

```
void gatt_client_register_handler(
  btstack_packet_handler_t handler);
```

这个回调函数会收到以下事件[1]：

- GATT_EVENT_MTU

  这个事件表示 MTU 协商完成。有两个解析函数：

  - gatt_event_mtu_get_handle(packet)

    获得连接句柄；

  - gatt_event_mtu_get_mtu(packet)

    获得 MTU 大小。

- GATT_EVENT_UNHANDLED_SERVER_VALUE [2]

  这个事件表示收到了服务器某特性的值上报，但是相关句柄未通过 gatt_client_listen_for_characterist
  注册回调函数。

  - gatt_event_unhandled_server_value_get_type(packet)

    获得上报类型，即 GATT_EVENT_NOTIFICATION 或 GATT_EVENT_INDICATION；

  - gatt_event_unhandled_server_value_get_value_handle(packet)

    获得特征的句柄；

  - gatt_event_unhandled_server_value_get_size(packet)

    获得上报的值的长度；

  - gatt_event_unhandled_server_value_get_data(packet)

    获得指向上报的值的指针。

8.1.1 句柄范围

特征包含值和若干个描述符，每个描述符也对应于一个句柄，所以一个特征包含从 start_handle 到 end_handle 的一系列句柄。同理，服务也对应一个句柄范围。

本章在不引起误解的前提下，把特征的值的句柄简称为“特征的句柄”。

[1] 以及转发过来的 L2CAP_EVENT_CAN_SEND_NOW 事件。

[2] SDK v8.4.23 及以上版本。

8.2 使用说明

8.2.1 创建客户端

调用 GATT 客户端的绝大多数带有连接句柄参数的 API 时，都会按需自动创建客户端实例。带有连接句柄参数但不会触发创建动作的 API：

- `gatt_client_listen_for_characteristic_value_updates`

通过 `gatt_client_is_ready()` 可显式地 “手动” 创建客户端实例。

8.2.2 发现服务

下列 API 用来发现服务：

- `gatt_client_discover_primary_services`：发现所有服务。
- `gatt_client_discover_primary_services_by_uuid16`：发现指定的服务。
- `gatt_client_discover_primary_services_by_uuid128`：发现指定的服务。

下列 API 用来发现服务内部的特征：

- `gatt_client_discover_characteristics_for_service`：发现一个服务里的所有特征
- `gatt_client_discover_characteristics_for_handle_range_by_uuid16`：在一定句柄范围内发现指定的特征
- `gatt_client_discover_characteristics_for_handle_range_by_uuid16`：在一定句柄范围内发现指定的特征
- `gatt_client_discover_characteristic_descriptors`：发现特征的所有描述符。

以 `gatt_client_discover_primary_services` 的使用为例说明使用方法。这个 API 的原型如下：

```
uint8_t gatt_client_discover_primary_services(
  // 回调
  user_packet_handler_t callback,
  // 连接句柄
  hci_con_handle_t con_handle);
```

其回调函数的例子：

```
static void service_discovery_callback(
    // 事件包类型（忽略）
    uint8_t packet_type,
    // 连接句柄（这个句柄也可以从事件内部获得，故忽略此参数）
    uint16_t _,
    // 事件包
    const uint8_t *packet,
    // 事件包大小
    uint16_t size)
{
    uint16_t con_handle;
    switch (packet[0])
    {
    // 对于发现的每个服务都有一个 QUERY_RESULT
    case GATT_EVENT_SERVICE_QUERY_RESULT:
        {
            const gatt_event_service_query_result_t *result =
                gatt_event_service_query_result_parse(packet);
            // ....
        }
        break;
    case GATT_EVENT_QUERY_COMPLETE:
        // 会话完成
        break;
    }
}
```

为了简化开发，SDK 提供了 *gatt_client_util* 模块，只调用一个函数就可以完成服务发现：

```
struct gatt_client_discoverer *gatt_client_util_discover_all(
    // 连接句柄
    hci_con_handle_t con_handle,
    // 发现完成后的回调
    f_on_fully_discovered on_fully_discovered,
    // 传递给黑调的用户数据
    void *user_data);
```

下面的回调函数演示了如何遍历所有服务、特征：

```
void my_on_fully_discovered(
    // 第一个服务
    service_node_t *first,
    // 用户数据
    void *user_data,
    // 错误码
    int err_code)
{
    if (err_code) ...
    service_node_t *s = first;
    // 遍历服务
    while (s)
    {
        char_node_t *c = s->chars;
        // 遍历服务的所有特征
        while (c)
        {
            desc_node_t *d = c->descs;
            // 遍历特征的所有描述符
            while (d)
            {
                d = d->next;
```

```
            }
            c = c->next;
        }
        s = s->next;
    }
}
```

8.2.3 读取特征

可以通过特征的句柄或者 UUID 读取值：

- gatt_client_read_value_of_characteristic_using_value_handle
- gatt_client_read_value_of_characteristics_by_uuid16
- gatt_client_read_value_of_characteristics_by_uuid128

也可以指定多个句柄批量读取：

- gatt_client_read_multiple_characteristic_values

基于句柄的分块读取：

- gatt_client_read_long_value_of_characteristic_using_value_handle
- gatt_client_read_long_value_of_characteristic_using_value_handle_with_offset

以 gatt_client_read_value_of_characteristic_using_value_handle 说明使用方法。其原型为：

```
uint8_t gatt_client_read_value_of_characteristic_using_value_handle(
  // 回调
  btstack_packet_handler_t callback,
  // 连接句柄
  hci_con_handle_t con_handle,
  // 特征句柄
  uint16_t characteristic_value_handle);
```

其回调函数的例子：

```
void read_characteristic_value_callback(
    // 事件包类型（忽略）
    uint8_t packet_type,
    // 连接句柄（这个句柄也可以从事件内部获得，故忽略此参数）
    uint16_t _,
    // 事件包
    const uint8_t *packet,
    // 事件包大小
    uint16_t size)
{
    switch (packet[0])
    {
    case GATT_EVENT_CHARACTERISTIC_VALUE_QUERY_RESULT:
        {
            uint16_t value_size;
            const gatt_event_value_packet_t *value =
                gatt_event_characteristic_value_query_result_parse(
                  packet, size, &value_size);
            // value->handle 为特征句柄
            // value_size 为值的长度
            // value->value 是指向值的指针
        }
        break;
    case GATT_EVENT_QUERY_COMPLETE:
        // 会话完成
        break;
    }
}
```

8.2.4 写入特征

通过特征的句柄写入值：

- gatt_client_write_value_of_characteristic：有响应的写入
- gatt_client_write_value_of_characteristic_without_response：无响应的写入

基于句柄的分块写入：

- gatt_client_write_long_value_of_characteristic
- gatt_client_write_long_value_of_characteristic_with_offset

其中 gatt_client_write_value_of_characteristic 的原型为：

```
uint8_t gatt_client_write_value_of_characteristic(
    // 回调
    btstack_packet_handler_t callback,
    // 连接句柄
    hci_con_handle_t con_handle,
    // 特征句柄
    uint16_t characteristic_value_handle,
    // 值的长度
    uint16_t length,
    // 指向值的指针
    const uint8_t * data);
```

其回调函数的例子：

```
void write_characteristic_value_callback(
    uint8_t packet_type, uint16_t _,
    const uint8_t *packet, uint16_t size)
{
    switch (packet[0])
    {
    case GATT_EVENT_QUERY_COMPLETE:
        platform_printf(" 特征写入完成。状态码: : %d\n",
            gatt_event_query_complete_parse(packet)->status);
        break;
```

```
    }
}
```

状态码的定义见ATT 错误码。

与之相比，无响应的写入不需要提供回调函数，没有 “会话” 的概念，下一次写入不需要等待上次完成，数据吞吐率更高。

```
uint8_t gatt_client_write_value_of_characteristic_without_response(
    // 连接句柄
    hci_con_handle_t con_handle,
    // 特征句柄
    uint16_t characteristic_value_handle,
    // 值的长度
    uint16_t length,
    // 指向值的指针
    const uint8_t * data);
```

8.2.5 订阅特征

开发者需要完成两个步骤：

1. 调用 gatt_client_listen_for_characteristic_value_updates 添加值更新时的回调函数

```
void gatt_client_listen_for_characteristic_value_updates(
    // 开发者提供一个回调函数结构体
    gatt_client_notification_t * notification,
    // 回调函数
    btstack_packet_handler_t packet_handler,
    // 连接句柄
    hci_con_handle_t con_handle,
    // 特征的值的句柄
    uint16_t value_handle);
```

由于 notification 会被添加到 GATT 客户端实例的回调链表中，所以必须指向一块全局内存，一定**不能**在栈上分配。notification 所指向的内容不需要初始化。

如果 notification 是从动态内存（如堆）里分配的，那么当连接断开时记得释放这块内存，以免泄露。

2. 调用 gatt_client_write_characteristic_descriptor_using_descriptor_handle 向 CCCD 写入使能标志

   这个函数的原型及使用方法与 gatt_client_write_value_of_characteristic 完全一致：

```
uint8_t gatt_client_write_characteristic_descriptor_using_descriptor_handle(
    // 回调函数
    btstack_packet_handler_t callback,
    // 连接句柄
    hci_con_handle_t con_handle,
    // CCCD 的句柄
    uint16_t descriptor_handle,
    // 数据长度（固定为 2）
    uint16_t length,
    // 值为 GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTIFICATION
    // 或   GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_INDICATION
    // 或   GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NONE
    uint8_t  * data);
```

或者调用 gatt_client_write_client_characteristic_configuration 向 CCCD 写入使能标志。这个函数的原型如下，它会自动发现 CCCD 的句柄而不需要通过参数传入：

```
uint8_t gatt_client_write_client_characteristic_configuration(
    // 回调函数
    btstack_packet_handler_t callback,
    // 连接句柄
    hci_con_handle_t con_handle,
    // 待订阅的特征
```

```
    gatt_client_characteristic_t * characteristic,
    // 值为 GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTIFICATION
    // 或   GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_INDICATION
    // 或   GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NONE
    uint16_t configuration);
```

如果待订阅的特征不支持指定的配置，则相应地返回 GATT_CLIENT_CHARACTERISTIC_NOTIFICATION_NO (0x98) 或 GATT_CLIENT_CHARACTERISTIC_INDICATION_NOT_SUPPORTED (0x99)。

8.2.6 ATT_MTU

除了通过监听 GATT_EVENT_MTU 事件以外，通过 gatt_client_get_mtu 可以随时获取 ATT_MTU：

```
// 返回错误码
uint8_t gatt_client_get_mtu(
    // 连接句柄
    hci_con_handle_t con_handle,
    // 输出 MTU
    uint16_t * mtu);
```

特征的值的最大长度为 (ATT_MTU-3)。调用 gatt_client_write_value_of_characteristic_witho 写入值时，如果超过最大长度，会返回错误码：GATT_CLIENT_VALUE_TOO_LONG (0x97)。

8.2.7 自定义 MTU [3]

当 GATT 客户端实例创建时，会自动发起 MTU 协商。开发者可通过以下接口干预 MTU 协商过程。

1. 为协议栈配置 STACK_GATT_CLIENT_DISABLE_MTU_EXCHANGE，关闭 MTU 自动协商功能；
2. 通过 gatt_client_exchange_mtu_request 编程发起 MTU 协商：

[3] SDK v8.4.23 及以上版本。

```
int gatt_client_exchange_mtu_request(
    hci_con_handle_t con_handle, // 连接句柄
    uint16_t mtu);               // MTU 建议值
```

注意：一个连接上 GATT 客户端最多只能发起一次 MTU 协商过程。

第九章　L2CAP

9.1　概览

BLE L2CAP 负责高层协议复用，分包、组包，以及 QoS 信息的传输。

L2CAP 定义了基于信用点的连接。两个设备之间可以建立多个基于信用点的连接，不同连接用 SPSM[1] 区分。SPSM 长度为 16 比特，`0x0001~0x007f` 为规范保留，`0x0080~0x00ff` 可自由使用：

- 对于服务端，SPSM 可以取固定值或根据 GATT Profile 动态设置；
- 对于客户端，应在每次建立连接后借助 GATT 发现服务端的 SPSM。

所谓信用点，指的是接收方允许对方在该连接上发送的 K 帧数目。对方每发送一个 K 帧就消耗一个信用点，当信用点为 0 时停止发送。接收方可以根据需要给对方补充信用点。每个传输单元称为一个 SDU。一个 SDU 可能被切分为多个 K 帧，所以发送一个 SDU 可能会消耗多个信用点。通过信用点可控制双方向的流量。

9.2　使用说明

9.2.1　从端请求更新连接参数

从角色调用 `l2cap_request_connection_parameter_update` 可向主端请求更新连接参数：

[1] 5.1 及更低版本里称为 LE_PSM

```
int l2cap_request_connection_parameter_update(
  // 连接句柄
  hci_con_handle_t con_handle,
  // 请求的最小连接间隔（单位 1.25ms）
  uint16_t conn_interval_min,
  // 请求的最大连接间隔（单位 1.25ms）
  uint16_t conn_interval_max,
  // 请求的从机延迟
  uint16_t conn_latency,
  // 请求的超时时间（单位 10ms）
  uint16_t supervision_timeout);
```

事件 HCI_SUBEVENT_LE_CONNECTION_UPDATE_COMPLETE 标志着参数更新完成。

9.2.2 基于信用点的连接

1. 注册 SPSM 及回调

   调用 l2cap_register_service 注册 SPSM 及对应的回调函数：

```
uint8_t l2cap_register_service(
  btstack_packet_handler_t packet_handler,
  uint16_t psm,  // SPSM
  uint16_t mtu,  // MTU 可填 0
  gap_security_level_t security_level
  );
```

这里的 MTU 最小为 23，可直接填 0，表示采用协议栈所支持的最大 MTU。无论是服务端还是客户端，都需要完成这个注册流程。

这个回调函数将收到以下事件：

- L2CAP_EVENT_CHANNEL_OPENED：连接已打开

  事件参数详见 l2cap_event_channel_opened_t。

- L2CAP_EVENT_CHANNEL_CLOSED：连接已关闭

  事件参数详见 `l2cap_event_channel_closed_t`。

- L2CAP_EVENT_COMPLETED_SDU_PACKET：收到完整 SDU[2]

  事件参数详见 `l2cap_event_complete_sdu_t`。

- L2CAP_EVENT_FRAGMENT_SDU_PACKET：收到 SDU 片断[3]

  事件参数详见 l2cap_event_fragment_sdu_t。

2. 建立连接

   BLE 连接建立并发现服务端的 SPSM 后，客户端就可以调用这个函数发起基于信用点的连接：

```
uint8_t l2cap_create_le_credit_based_connection_request(
  uint16_t credits,   // 赋于对端的初始信用点
  uint16_t psm,       // SPSM
  uint16_t handle,    // 连接句加柄
  uint16_t *local_cid // 本基于信用点的连接的 ID
);
```

3. 发送数据

   连接打开后调用 `l2cap_credit_based_send` 发送数据：

```
int l2cap_credit_based_send(
  uint16_t local_cid,   // 基于信用点的连接的 ID
  const uint8_t *data,  // SDU 数据
  uint16_t len          // SDU 总长度
);
```

   这个函数如果出错，将返回负数错误码：

- `-BTSTACK_LE_CHANNEL_NOT_EXIST`：连接不存在

[2]整个 SDU 只占用一个 K 帧

[3]整个 SDU 占用多个 K 帧这个事件对应其中一个 K 帧

否则这个函数会返回发送出去[4]的数据的长度。当整个 SDU 都已发送时，返回值与 `len` 参数相同，否则小于 `len` 参数[5]。这时，需要等待有可用的信用点或者链路层队列出现空余时：如果返回值为 0，则再次调用 `l2cap_credit_based_send` 重新发送；否则调用 `l2cap_credit_based_send_continue` 继续发送剩余数据：

```
int l2cap_credit_based_send_continue(
  uint16_t local_cid,   // 基于信用点的连接的 ID
  const uint8_t *data,  // SDU 剩余数据
  uint16_t len          // SDU 剩余长度
);
```

在完整发送一个 SDU 前不可调用 `l2cap_credit_based_send` 发送新的 SDU。

4. 为对端补充信用点

   调用 `l2cap_le_send_flow_control_credit` 即可为对端补充信用点：

```
uint8_t l2cap_le_send_flow_control_credit(
  uint16_t local_cid,   // 基于信用点的连接的 ID
  uint16_t credits      // 要补充的点数
);
```

5. 查询信用点数

   调用 `l2cap_get_le_credit_based_connection_credits` 查询当前的信用点情况：

```
uint8_t l2cap_get_le_credit_based_connection_credits(
  uint16_t local_cid,     // 基于信用点的连接的 ID
  uint32_t *peer_credits,
  uint32_t *local_credits
);
```

[4]指进入链路层发送队列。

[5]当信用点用完或者链路层队列已满时，会出现此情况

local_credits 表示对方还能允许本方在该连接上发送多少 K 帧；peer_credits 表示对方的 local_credits。

9.2.3 传输队列

通过 GATT 向对方上报或者写入数据时，数据会存入 Controller 的传输队列。如果传输队列的缓存已满，相关 API 会返回 BTSTACK_ACL_BUFFERS_FULL (0x57)。如果这些数据必须传输到对方、不可丢弃，那么可参考以下几种方法重新尝试发送：

- 响应 HCI_EVENT_NUMBER_OF_COMPLETED_PACKETS 事件，重新尝试发送；
- 要求协议栈产生 “可发送” 事件，在事件中重新尝试发送；

  对于 GATT 服务器，调用 att_server_request_can_send_now_event。待传输队列不满时，GATT 服务器的事件回调函数会收到 ATT_EVENT_CAN_SEND_NOW 事件。也可以调用 att_dispatch_server_request_can_send_now_event[6]，待传输队列不满时，会收到 L2CAP_EVENT_CAN_SEND_NOW 事件。

  对于 GATT 客户端，调用 att_dispatch_client_request_can_send_now_event。待传输队列不满时，GATT 客户端的事件回调函数会收到 L2CAP_EVENT_CAN_SEND_NOW 事件。

  另有 att_server_can_send_now、att_dispatch_client_can_send_now、att_dispatch_server_can_se 等几个以 can_send_now 为后缀的 API，用于查询当前是否可发送数据。这几个 API 一般不需要调用。以 att_server_notify 为例，其伪代码如下：

```
int att_server_notify(con_handle, attribute_handle, value){
    if (con_handle 不存在)
      return BTSTACK_LE_CHANNEL_NOT_EXIST;

    if (!att_dispatch_server_can_send_now(con_handle))
      return BTSTACK_ACL_BUFFERS_FULL;

    return l2cap_send(con_handle, L2CAP_CID_ATTRIBUTE_PROTOCOL,
                      header, value);
}
```

[6] att_server_request_can_send_now_event 是基于 att_dispatch_server_request_can_send_now_event 实现的。

可见，调用 att_server_notify 之前不需要先调用 att_dispatch_server_can_send_now。

- 也可以延迟一段时间后重试。

第十章　安全管理

10.1　概览

安全管理（Security Manager）协议负责配对、认证及加密。连接的主角色在 SM 里扮演发起者（Initiator），从角色扮演应答者（Responder）。SM 涉及多个密钥，其层次关系如图 10.1，位于顶层的是 ER 和 IR，长度都是 $128bits$[1]，$d1$ 是加解密工具箱里的一个工具，DIV 是一个随机数[2]。

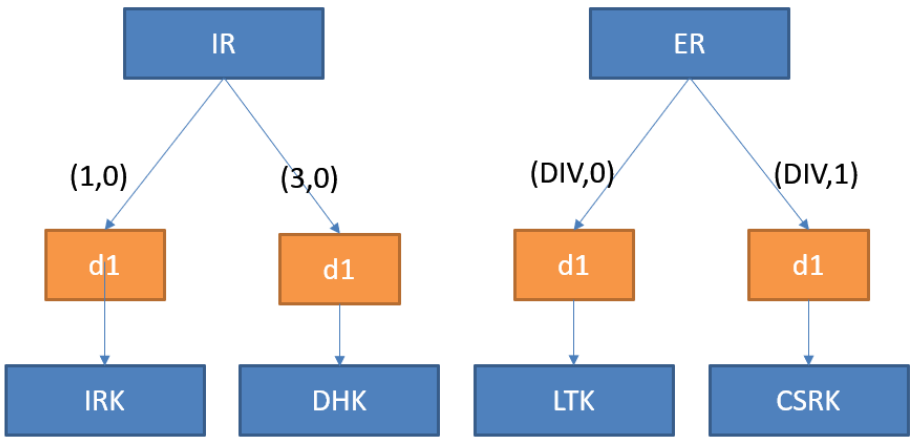

图 10.1: BLE 密钥层次结构

SM 涉及两个重要概念：

- **绑定**：在两个连接的设备之间交换秘密和身份信息以建立相互信赖关系。

  有的设备可能不支持绑定，因此规范定义了两种绑定模式：可绑定、不可绑定。建立绑定时需要使用**配对**流程交换秘密和身份信息。

[1]生成时需要保证具有足够高的熵。

[2]IRK、DHK、LTK、CSRK 等的具体含义及作用请参考蓝牙核心规范。

- **配对**：是一个用户层面的概念，需要用户输入识别码（passkey）。

  两种情况下会发生配对：1) 为了绑定；2) 用于认证未绑定的设备。

10.2 使用说明

10.2.1 初始化

1. 调用 `sm_add_event_handler` 添加 SM 模块事件回调

```
void sm_add_event_handler(
  // 回调函数
  btstack_packet_callback_registration_t * callback_handler);
```

2. 调用 `sm_config` 配置基础数据 ER、IR

```
void sm_config(
  // 是否使能 SM（默认为不使能）
  uint8_t enable,
  // 设备的 IO 能力
  io_capability_t io_capability,
  // 从角色时是否自动发送安全请求
  int   request_security,
  // 持久化数据
  const sm_persistent_t *persistent);
```

这里的 IO 能力 `io_capability` 用于协商配对方法，跟设备的实际 IO 能力无关：

```
typedef enum {
  IO_CAPABILITY_UNINITIALIZED = -1,
  IO_CAPABILITY_DISPLAY_ONLY = 0,    // 只支持显示
  IO_CAPABILITY_DISPLAY_YES_NO,      // 可显示，可输入 YES、NO
  IO_CAPABILITY_KEYBOARD_ONLY,       // 只能输入
```

```
  IO_CAPABILITY_NO_INPUT_NO_OUTPUT, // 无输入、输出能力
  IO_CAPABILITY_KEYBOARD_DISPLAY,   // 可显示，能输入
} io_capability_t;
```

这里的输入能力指可以输入 0 – 9 等 10 个数字，以及 YES、NO；显示能力是指可以显示 6 位十进制识别码（000000 – 999999）[3]。

持久化数据的定义如下：

```
typedef struct sm_persistent
{
  // ER 密钥
  sm_key_t        er;
  // IR 密钥
  sm_key_t        ir;
  // 身份地址
  bd_addr_t       identity_addr;
  // 身份地址类型（BD_ADDR_TYPE_LE_PUBLIC 或 BD_ADDR_TYPE_LE_RANDOM）
  bd_addr_type_t  identity_addr_type;
} sm_persistent_t;
```

3. 通过 sm_set_authentication_requirements 设置认证需求

```
void sm_set_authentication_requirements(
  // SM_AUTHREQ_... 的组合: 是否绑定、是否需要 MITM 保护
  uint8_t auth_req);

// 不可绑定
#define SM_AUTHREQ_NO_BONDING       ...
// 可绑定
#define SM_AUTHREQ_BONDING          ...
// 使能 MITM 保护
```

[3]以及给予用户必要的提示的能力。

```
#define SM_AUTHREQ_MITM_PROTECTION  ...
// 使能 基于 LE 安全连接的配对
#define SM_AUTHREQ_SC               ...
```

10.2.2 使用私有随机地址

如果需要使用私有随机地址，可调用 sm_private_random_address_generation_set_mode 启动 SM 模块的私有地址生成功能：

```
void sm_private_random_address_generation_set_mode(
  // 私有地址的生成方式
  gap_random_address_type_t random_address_type);
```

私有地址的生成方式有三种：

```
typedef enum {
  GAP_RANDOM_ADDRESS_OFF = 0,          // 不生成（默认值）
  GAP_RANDOM_ADDRESS_NON_RESOLVABLE,   // 生成不可解析私有地址
  GAP_RANDOM_ADDRESS_RESOLVABLE,       // 生成可解析私有地址
} gap_random_address_type_t;
```

启动后 SM 模块将尽快产生一个新的地址并弹出 SM_EVENT_PRIVATE_RANDOM_ADDR_UPDATE 事件，之后周期性地重新产生并弹出事件。周期默认为 15 分钟，可通过 sm_private_random_address_generation_s 修改[4]：

```
void sm_private_random_address_generation_set_update_period(
  int period_ms);
```

[4]没必要过于频繁地更新地址。尤其是对于可解析地址，每更新一次，就意味着泄露了一点关于 IRK 的消息。

10.2.3 SM 事件回调

SM 模块的事件回调函数将收到一系列事件，App 的主要工作就在于响应这些事件。

- `SM_EVENT_PRIVATE_RANDOM_ADDR_UPDATE`

  SM 生成了一个新的私有随机地址。App 此时可以更新广播地址[5]，比如：

```
// 更新广播集 0 的随机地址
gap_set_adv_set_random_addr(0,
  sm_private_random_addr_update_get_address(packet));
```

与地址解析、查找有关的 3 个事件：

- `SM_EVENT_IDENTITY_RESOLVING_STARTED`：开始解析某个地址

  使用以下 API 可读取正在解析的地址等信息：

  - `sm_event_identity_resolving_started_get_handle(packet)`：连接句柄
  - `sm_event_identity_resolving_started_get_addr_type(packet)`：正在解析的地址类型
  - `sm_event_identity_resolving_started_get_address(packet, addr)`：正在解析的地址

- `SM_EVENT_IDENTITY_RESOLVING_FAILED`：解析失败

  使用以下 API 可读取解析失败的地址等信息：

  - `sm_event_identity_resolving_failed_get_handle(packet)`：连接句柄
  - `sm_event_identity_resolving_failed_get_addr_type(packet)`：正在解析的地址类型
  - `sm_event_identity_resolving_failed_get_address(packet, addr)`：正在解析的地址

- `SM_EVENT_IDENTITY_RESOLVING_SUCCEEDED`：解析成功

  使用以下 API 可读取解析成功的地址等信息：

[5]注意：当该广播是可连接的并且正在广播时，不允许修改地址。

- sm_event_identity_resolving_succeeded_get_handle(packet)：连接句柄
- sm_event_identity_resolving_succeeded_get_addr_type(packet)：正在解析的地址类型
- sm_event_identity_resolving_succeeded_get_address(packet, addr)：正在解析的地址
- sm_event_identity_resolving_succeeded_get_le_device_db_index(packet)：在“设备数据库”里的序号

例如，通过下面的代码获取解析出的身份地址：

```
uint16_t index =
    sm_event_identity_resolving_succeeded_get_le_device_db_index(packet);
const le_device_memory_db_t *item =
    le_device_db_from_key(index);
printf("RESOLVING_SUCCEEDED: (%d) ", index);
if (item)
    printf_hexdump(item->addr, BD_ADDR_LEN);
else
    printf("ERROR: should not happen");
printf("\n");
```

与配对有关的事件：

- SM_EVENT_JUST_WORKS_REQUEST：JUST_WORKS 请求，等待用户接收或拒绝

  调用 sm_just_works_confirm 接受，sm_bonding_decline 拒绝。

- SM_EVENT_PASSKEY_DISPLAY_NUMBER：显示识别码

  App 收到此事件后显示识别码，并提示用户在对端输入此识别码。

- SM_EVENT_PASSKEY_DISPLAY_CANCEL：不要再显示识别码

  App 收到此事件后应该刷新显示，不再呈现识别码。

- SM_EVENT_PASSKEY_INPUT_NUMBER：提示用户输入识别码

  App 收到此事件后提示用户输入识别码，然后调用 sm_passkey_input 把用户的输入传递到协议栈。调用 sm_bonding_decline 可中止配对。

- SM_EVENT_NUMERIC_COMPARISON_REQUEST：提示用户对比数字识别码

  仅在基于 LE 安全连接的配对时出现。调用 sm_numeric_comparison_confirm 确认数字无误，sm_bonding_decline 否认。

  请注意区分两类涉及数字识别码的配对事件，并了解 Bluetooth SIG 的安全说明[6]：

  – 对于基于 LE 安全连接的配对,配对双方同时显示数字（SM_EVENT_NUMERIC_COMPARISON）用户在两个设备上分别确认显示的数字是否一致；

  – 对于不使用 LE 安全连接的配对（即 LE 传统方式），会在具备显示能力一方显示识别码（SM_EVENT_PASSKEY_DISPLAY_NUMBER），另一方具备输入能力的设备会提示用户输入（SM_EVENT_PASSKEY_INPUT_NUMBER）这个数字完成配对。

SM 状态改变事件：

- SM_EVENT_STATE_CHANGED

  这个事件指示 SM 总状态的变化。使用 decode_hci_event 将其解析为 sm_event_state_changed_t：

```
typedef struct sm_event_state_changed {
  // 连接句柄
  uint16_t conn_handle;
  // 状态变化的原因
  uint8_t reason;
} sm_event_state_changed_t;

const sm_event_state_changed_t *state_changed =
  decode_hci_event(packet, sm_event_state_changed_t);
```

这里的 reason 即 SM 的几种**主要**状态：

```
enum sm_state_t
{
  SM_STARTED,                 // SM 启动
  SM_FINAL_PAIRED,            // 已配对
```

[6]https://www.bluetooth.com/learn-about-bluetooth/key-attributes/bluetooth-security/method-vulnerability/

```
  SM_FINAL_REESTABLISHED,     // 已重新建立
  SM_FINAL_FAIL_PROTOCOL,     // 协议流程错误
  SM_FINAL_FAIL_TIMEOUT,      // 超时
  SM_FINAL_FAIL_DISCONNECT,   // 连接断开
};
```

10.2.4 每个连接的个性化设置

SM API 支持为每个连接进行个性化的设置：

```
void sm_config_conn(
  // 连接句柄
  hci_con_handle_t con_handle,
  // IO 能力
  io_capability_t io_capability,
  // SM_AUTHREQ_... 的组合: 是否绑定、是否需要 MITM 保护等
  uint8_t auth_req);
```

注意，这个 API 只允许在 HCI 事件 `HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE` 的回调里调用。即使 SM 为禁用状态，也可以使用这个 API 为单个连接使能 SM。

10.2.5 地址解析、查找

通过 `sm_address_resolution_lookup` 可以“手动”触发蓝牙设备地址的解析、查找：

```
int sm_address_resolution_lookup(
  uint8_t addr_type,  // 地址类型
  bd_addr_t addr);    // 地址
```

这里的“查找”指是的在“设备数据库”中查找。如果传入的地址可解析私有地址，那么将尝试解析该地址。

这个函数将待查地址加入一个处理队列，如果成功加入，返回值为 0；反之返回非 0 值。解析、查找的结果通过 SM_EVENT_IDENTITY_RESOLVING_FAILED、SM_EVENT_IDENTITY_RESOLVING_SUCCEEDED 等事件上报。

10.2.6 P-256 椭圆曲线

要在 ING916、ING918 上使用基于 LE 安全连接的配对，需要先为 Controller 安装 FIPS 186-4[7] P-256 椭圆曲线计算引擎：

```
void ll_install_ecc_engine(
  // 用于生成一对新的公钥和私钥的回调函数
  f_start_generate_p256_key_pair start_generate_p256_key_pair,
  // 用于计算 DH 交换密钥的回调
  f_start_generate_dhkey start_generate_dhkey);
```

回调函数的实现方法可参考 SDK 里的相关示例。

当不使用 OOB 时，SM 层自动管理公钥、私钥的生成：对于每个基于 LE 安全连接的配对流程都会重新生成私钥。当使用 OOB 时，考虑到可能存在与多个设备配对的情况，私钥的更新由开发者控制：每次调用 `sm_sc_generate_oob_data` 时，重新生成私钥和 OOB 数据，该私钥（及对应的公钥）将应用于后续所有的基于 LE 安全连接的配对流程。此时，为了保护设备的私钥，开发者需要及时重新调用 `sm_sc_generate_oob_data` 刷新私钥和 OOB 数据。参照蓝牙核心规范[8]，最迟应该在 $S + 3F > 8$ 时刷新私钥，其中 S 表示当前私钥参与的配对成功的次数，F 表示当前私钥参与的配对失败的次数。

10.2.7 基于 OOB 数据的配对

- 对于 LE 传统 OOB 配对：

  通过 `sm_register_oob_data_callback` 注册回调函数来为 SM 提供 OOB 数据：

[7] dx.doi.org/10.6028/NIST.FIPS.186-4

[8] BLUETOOTH CORE SPECIFICATION Version 5.4 | Vol 3, Part H, 2.3.6

```
void sm_register_oob_data_callback(
  int (*get_oob_data_callback)(
    uint8_t address_type,
    bd_addr_t addr,
    uint8_t * oob_data));
```

这个回调函数的签名为：

```
// 存在与该对端设备关联的 OOB 数据时返回 1; 否则返回 0.
int (*get_oob_data_callback)(
    // 对端设备的地址类型
    uint8_t address_type,
    // 对端设备的地址
    bd_addr_t addr,
    // 输出: OOB 数据（长度与 sm_key_t 相同）
    uint8_t * oob_data)
```

此 OOB 数据为随机生成。

- 基于 LE 安全连接的 OOB 配对

通过 `sm_register_sc_oob_data_callback` 注册回调函数来为 SM 提供 OOB 数据：

```
void sm_register_sc_oob_data_callback(
  int (*get_sc_oob_data_callback)(
    uint8_t address_type,
    bd_addr_t addr,
    uint8_t *peer_confirm,
    uint8_t *peer_random));
```

这个回调函数的签名为：

```
// 存在与该对端设备关联的 OOB 数据时返回 1; 否则返回 0.
int (*get_sc_oob_data_callback)(
    // 对端设备的地址类型
    uint8_t address_type,
    // 对端设备的地址
    bd_addr_t addr,
    // 输出: OOB 数据 confirm（长度与 sm_key_t 相同）
    uint8_t *peer_confirm,
    // 输出: OOB 数据 random （长度与 sm_key_t 相同）
    uint8_t *peer_random)
```

confirm 是从 random 和公钥等数据中推算得出，通过调用 `sm_sc_generate_oob_data` 可触发生成一组新的 random、confirm 数据：

```
int sm_sc_generate_oob_data(
  void (*callback)(
    const uint8_t *confirm,
    const uint8_t *random));
```

新的 random、confirm 数据生成后，会调用这个回调告知结果。通过带外传输将这组数据传递到对端，对端在 `get_sc_oob_data_callback` 里将这组数据再传递到 SM 完成 OOB 配对。

`sm_sc_generate_oob_data` 函数会返回以下几种值：

- 0: 开始计算新的 OOB 数据
- -1: 上一组 OOB 数据还在计算中

第十一章　Controller

Controller 除了通过 HCI 为 Host 提供服务以外，还提供了一系列接口供开发者直接调用。这些接口有的弥补 HCI 的不足，有的提供标准以外的扩展功能。这些接口与芯片、软件包紧密耦合，不同芯片系列、软件包，所提供的 Controller 接口也不相同，比如 `typical` 软件包不提供不符合规范的 “非标” 接口。

11.1　配置项

11.1.1　功能开关

链路层包含若干功能开关，通过 `platform_config(PLATFORM_CFG_LL_DBG_FLAGS, ...)` 设置。

各功能开关及设置后所产生的效果如下：

- `LL_FLAG_DISABLE_CTE_PREPROCESSING`：关闭 CTE 预处理。

  用于调试。非必要不应设置。

- `LL_FLAG_LEGACY_ONLY_INITIATING`：仅通过传统广播信道建立连接。

  当待连接的设备仅支持或仅使用传统广播时，建议开启，可明显提高连接建立速度。

- `LL_FLAG_LEGACY_ONLY_SCANNING`：仅扫描传统广播。

  当待扫描的设备仅支持或仅使用传统广播时，建议开启，可明显提高扫描效率。

- `LL_FLAG_REDUCE_INSTANT_ERRORS`：尝试减少 “0x28 - 时机已过” 错误码的上报。

  多连接场景当遇到较多 “0x28 - 时机已过” 错误时，可尝试设置。

- `LL_FLAG_DISABLE_RSSI_FILTER`：关闭内部的 RSSI 滤波器

  当需要读取原始 RSSI 时设置。

- LL_FLAG_RSSI_AFTER_CRC：仅从 CRC 正常的数据包获取 RSSI

  当需要更稳定的 RSSI 时设置。

通过 platform_config 时，应把所有需要打开的开关组合起来，一并设置。如：

- 对方设备仅仅支持或仅使用传统广播，组合使用 LL_FLAG_LEGACY_ONLY_INITIATING 和 LL_FLAG_LEGACY_ONLY_SCANNING：

```
platform_config(PLATFORM_CFG_LL_DBG_FLAGS,
      LL_FLAG_LEGACY_ONLY_INITIATING
    | LL_FLAG_LEGACY_ONLY_SCANNING);
```

- 要获得尽可能稳定的 RSSI，那么：

```
platform_config(PLATFORM_CFG_LL_DBG_FLAGS,
      LL_FLAG_RSSI_AFTER_CRC);
```

- 要获得尽可能多的原始 RSSI，那么：

```
platform_config(PLATFORM_CFG_LL_DBG_FLAGS,
      LL_FLAG_DISABLE_RSSI_FILTER);
```

11.1.2 可配参数

链路层还包含若干带有参数值的可配置项，通过 ll_config 配置：

```
void ll_config(ll_config_item_t item, // 项目
    uint32_t value);                  // 值
```

ll_config_item_t 包含以下项目：

- `LL_CFG_SLAVE_LATENCY_PRE_WAKE_UP`：使用从机延迟时，用于预唤醒的时间提前量。

  Controller 在处理连接时，包含两部分工作：主要的处理以 RTOS 任务形式进行，另外少量的工作（配置和触发硬件）在中断中进行。使用从机延迟时，假设按照从机延迟处理流程，需要在 T 时刻的中断里配置和触发硬件，但是在这之前，RTOS 任务有可能一直未能触发，所以需要在 T 时刻之前主动唤醒 RTOS 任务，完成必要的处理，为 T 时刻的中断做好准备。

  参数值的范围为 $1 \sim 255$，单位为 $0.625ms$。默认值为 4。

- `LL_CFG_FEATURE_SET_MASK`：特性集合掩码。

  有时需要 “模仿” 其它 BLE 设备的链路层协议流程，而支持的特性对链路层协议流程影响很大。为此可通过该配置项调整上报给对端设备的链路层特性。例如，模仿只支持加密和 2M PHY 两种特性的设备：

```
const uint8_t feature_mask[8] =
{
    0x01,       // 比特 0: LE Encryption
    0x01,       // 比特 8: LE 2M PHY
};

// 参数值为指向掩码数组的指针
ll_config(LL_CFG_FEATURE_SET_MASK,
          (uintptr_t)feature_mask);
```

  注意，这里只是修改了 Feature Exchange 流程的上报值，对链路层的实际功能没有影响。需要保证设置的掩码数组一直存在，不可释放。

通过 `ll_set_max_conn_number` 设置可能用到的最大连接数：

```
int ll_set_max_conn_number(
    int max_number);
```

比如软件包本身支持的最大连接数为 N，但是应用中实际最多用到 2 个连接，通过 `ll_set_max_conn_number(2)` 可优化多连接时的数据吞吐率。`ll_set_max_conn_number(M)`，$M > N$，并不能增加软件包本身支持的最大连接数。

当一个连接事件中接收到多个 ACL 数据包时，Controller 默认以 4 个[1]为一组上报，而不是每收到一个即上报，以减少任务间的切换开销。通过 `ll_set_conn_acl_report_latency` 可以修改上报频率：

```
void ll_set_conn_acl_report_latency(
    uint16_t conn_handle, // 连接句柄
    int latency);         // 以 latency 个包一组上报
```

当 `latency` 为 0 时，表示总是等到连接事件结束时集中上报；`latency` 为 1 时，每收到一个 ACL 数据包就立即上报。这个参数保存在连接对象内，当连接断开后，配置消失。

11.2 HCI 增强

11.2.1 读取特性和能力

通过 `ll_get_capabilities` 可能直接获取 Controller 支持的特性及各种能力。

```
void ll_get_capabilities(
    ll_capabilities_t *capabilities);
```

`ll_capabilities_t` 的定义如下。

```
typedef struct ll_capabilities
{
    // 链路层支持的特性集合
    const uint8_t *features;
    // 最大广播集数目
    uint16_t adv_set_num;
    // 最大连接数
    uint16_t conn_num;
    // 白名单列表的大小
```

[1]典型值。不同软件包有所不同。

```
    uint16_t whitelist_size;
    // 地址解析列表的大小
    uint16_t resolving_list_size;
    // 周期广播者列表的大小
    uint16_t periodic_advertiser_list_size;
    // 用于检测广播数据重复的过滤器的大小
    uint16_t adv_dup_filter_size;
} ll_capabilities_t;
```

11.2.2 工作状态

通过 `ll_get_states` 可以获取当前 Controller 的工作状态：

```
void ll_get_states(uint32_t *adv_states,
    uint32_t *conn_states,
    uint32_t *sync_states,
    uint32_t *other_states);
```

`adv_status[n]` 中第 `n` 的 `uint32_t` 的第 `i` 比特表示第 $(n \times 32 + i)$ 个广播集的工作状态，为 0 表示未使能，1 表示已使能（正在广播）。`conn_states`、`sync_states` 的含义与之类似。传入这三个数组指针参数时，数组的长度应为最大数目向上转换为 32 的倍数，再除以 32 所得到的商。如从 `ll_get_capabilities` 得知最大广播集数目 `adv_set_num` 为 10，则 `adv_states` 的长度为 1；最大连接数 `conn_num` 为 33，则 `conn_states` 的长度应为 2。

`other_states` 数组长度目前固定为 1，`other_states[0]` 的比特 0 为 1 表示正在扫描；比特 1 为 1 表示正在建立连接。

11.2.3 发射功率

通过 `ll_set_tx_power_range` 可设置发送功率范围。此范围将被用于广播、连接等各种需要发射的场景。

```
void ll_set_tx_power_range(
    int16_t min_dBm, // 最小发射功率（单位：dBm）
    int16_t max_dBm);// 最大发射功率（单位：dBm）
```

此范围限制存在于链路层，对 HCI 命令里指定的发射功率起限制作用。如将 max_dBm 设置为 0 dBm，则通过 gap_set_ext_adv_para 设置广播的发射功率为 3 dBm 时，最终发射功率会被限制为 0 dBm。

显然，最终的发射功率还受硬件实际支持的范围限制，将 max_dBm 设置为 1000 dBm，并不可能得到 1000 dBm 的最大发射功率。

通过 ll_set_conn_tx_power 可以直接调节连接的发射功率：

```
void ll_set_conn_tx_power(
    uint16_t conn_handle, // 连接句柄
    int16_t tx_power);    // 发射功率（单位：dBm）
```

当对端设备支持功率控制特性时，可通过 ll_adjust_conn_peer_tx_power 尝试调整对端的发射功率：

```
void ll_adjust_conn_peer_tx_power(
    uint16_t conn_handle, // 连接句柄
    int8_t delta);        // 功率增量（单位：dB）
```

delta 为正数，表示请求对端增加发射功率，为负数则表示降低发射功率。

11.2.4 编码方式

当需要使用 Coded 编码时，默认采用 S8 方式。

```
typedef enum coded_scheme_e
{
    BLE_CODED_S8,
    BLE_CODED_S2
} coded_scheme_t;
```

通过 ll_set_adv_coded_scheme 修改广播集的 Coded 编码方式：

```
void ll_set_adv_coded_scheme(
    const uint8_t adv_hdl,       // 广播集句柄
    const coded_scheme_t scheme);//  Coded 编码方式
```

此函数需要在 adv_hdl 使能前调用方能生效。

通过 ll_set_initiating_coded_scheme 设置以 Coded 编码发送连接请求时的编码方式：

```
void ll_set_initiating_coded_scheme(
    const coded_scheme_t scheme);
```

此函数需要 gap_ext_create_connection 之前调用方能生效。

使用 ll_set_conn_coded_scheme 修改连接的 Coded 编码方式：

```
void ll_set_conn_coded_scheme(
    uint16_t conn_handle, // 连接句柄
    int ci);              //  Coded 编码方式（同 `coded_scheme_t`）
```

11.2.5 底层广播参数

一个广播事件会在多个主广播信道上各发送一个广播数据包(参见primary_adv_channel_map参数)。通过 ll_legacy_adv_set_interval 可配置相邻两个广播数据包的间隔：

```
void ll_legacy_adv_set_interval(
    uint16_t for_hdc, // 用于高占空比情况（单位：⊠s），默认 1250 ⊠s
    uint16_t not_hdc);// 用于其它情况（单位：⊠s），默认 1500 ⊠s
```

platform_config(PLATFORM_CFG_LL_LEGACY_ADV_INTERVAL, flag) 等效于：

```
ll_legacy_adv_set_interval(flag >> 16, flag & 0xffff);
```

适当减小间隔可以略微降低功耗。

11.2.6 底层连接参数

Controller 调度连接事件时，以连接事件长度（ce_len）为参考。连接建立后的从设备，ce_len 总是初始化为连接间隔，即追求最大吞吐率。对于多连接或多状态并发，这种初始设置并不合适。此时，可通过 ll_hint_on_ce_len 提示 Controller 实际需要的连接事件长度。将 ce_len 调小后，Controller 就可以在连接事件结束后调度其它任务，有效实现多任务并发。

```
void ll_hint_on_ce_len(
    const uint16_t conn_handle, // 连接句柄
    const uint16_t min_ce_len,  // 事件长度的最小值（单位：0.625 ms）
    const uint16_t max_ce_len); // 事件长度的最大值（单位：0.625 ms）
```

这个函数调整已建立的连接调用。

此函数同样适应于主设备。主设备与从设备的不同在于其 ce_len 来自phy_configs 参数。

从机延迟参数可以有效降低速低功耗应用的耗电，但主机端往往把从机延迟设为 0。通过 ll_set_conn_latency 可以为从机**主动**设置从机延迟参数降低功耗：

```
void ll_set_conn_latency(
    uint16_t conn_handle,   // 连接句柄
    int latency);           // 从机延迟参数
```

注意：不建议在使用了减速模式的情况下使用。

通过 ll_get_conn_info 可以读取连接的基础参数：

```
// 函数执行成功时返回 0，否则返回非 0 值。
int ll_get_conn_info(
    const uint16_t conn_handle, // 连接句柄
```

```
    uint32_t *access_addr,      // 接入地址
    uint32_t *crc_init,         // CRC 初始值
    uint8_t *hop_inc);          // 信道选择算法 #1 跳频增量
```

通过 ll_get_conn_events_info 获取未来若干连接事件的参数信息：

```
// 函数执行成功时返回 0，否则返回非 0 值。
int ll_get_conn_events_info(
    const uint16_t conn_handle, // 连接句柄
    int number,                 // 连接事件个数
    uint64_t from_time,         // 时间参考
    uint32_t *interval,         // 输出：连接间隔（单位：⊠s）
    uint32_t *time_offset,      // 输出：第一个连接事件与 `from_time` 的时间差
    uint16_t *event_count,      // 输出：第一个连接事件的事件计数值
    uint8_t *channel_ids);      // 输出：`number` 个物理信道号
```

这个函数输出 from_time 之后 number 个连接事件的物理信道号等参数。注意：这个函数假定从当前时刻到 number 个连接事件结束信道参数不发生变化，并且忽略减速模式参数。当信道参数发生变化时（如连接参数更新、信道集合更新），输出的参数将是不可靠的。

11.2.7 默认天线

当系统配置了多个天线（如使用 CTE、信道探测等特性时）时，通过 ll_set_def_antenna 可设置射频工作时的默认天线 ID：

```
void ll_set_def_antenna(uint8_t ant_id);
```

这个设置直接写入硬件，立即生效。

11.2.8 单信道扫描

扫描广播时，在每个扫描窗口轮流扫描 3 个主广播信道。可通过 ll_scan_set_fixed_channel 限定只扫描其中一个主广播信道。

```
void ll_scan_set_fixed_channel(
    int channel_index);
```

channel_index 可以是 37、38 或 39；也可以是 0，表示解除锁定，恢复为轮流扫描所有主广播信道。

此项设置一直生效。

11.3 连接中止与重建

通过 ll_create_conn 可以跳过广播、连接建立过程，直接创建或者恢复连接：

```
// 函数执行成功时返回 0，否则返回非 0 值。
int ll_create_conn(
    uint8_t role,               // 角色。主（0），从（1）
    uint8_t addr_types,         // 广播者和连接发起者的地址类型
    const uint8_t *adv_addr,    // 广播者地址
    const uint8_t *init_addr,   // 连接发起者的地址
    uint8_t rx_phy,             // 接收 PHY（以主角色为参考）
    uint8_t tx_phy,             // 发射 PHY（以主角色为参考）
    uint32_t access_addr,       // 接入地址
    uint32_t crc_init,          // CRC 初始值
    uint32_t interval,          // 连接间隔（单位：μs）
    uint16_t sup_timeout,       // 超时时间（单位：10 ms）
    const uint8_t *channel_map, // 信道集合（5 个字节，37 个信道的 bitmap）
    uint8_t  ch_sel_algo,       // 信道选择算法（0: 算法 #1; 1: 算法 #2）
    uint8_t  hop_inc,           // 跳频增量（仅用于信道选择算法 #1）
    uint8_t  last_unmapped_ch,  // 上一次未映射信道号（仅用于信道选择算法 #1）
    uint16_t min_ce_len,        // 事件长度的最小值（单位：0.625 ms）
    uint16_t max_ce_len,        // 事件长度的最大值（单位：0.625 ms）
    uint64_t start_time,        // 首个连接事件的起始时间（单位：μs）
    uint16_t event_counter,     // 首个连接事件的事件计数值
    uint16_t slave_latency,     // 从机延迟参数
```

```
    uint8_t  sleep_clk_acc,     // 睡眠时钟精度（仅用于从角色）
    uint32_t sync_window,       // 首个连接事件的同步窗口长度（单位：0.625ms）
    const void *security);      // 链路层安全上下文
```

security 来自 HCI_SUBEVENT_LE_VENDOR_CONNECTION_ABORTED 事件，NULL 表示不加密。

使用 ll_conn_abort 中止连接：

```
// 中止流程成功启动时返回 0，否则返回非 0 值。
int ll_conn_abort(
    uint16_t conn_handle);      // 连接句柄
```

连接中止后，将上报 HCI_SUBEVENT_LE_VENDOR_CONNECTION_ABORTED 事件。

11.4 广播上的 CTE

CTE 私有方案 #2[2] 使用扩展广播发送 CTE。通过 ll_attach_cte_to_adv_set 为已初始化且未使能的广播集附着 CTE：

```
// 函数执行成功时返回 0，否则返回非 0 值。
int ll_attach_cte_to_adv_set(
    uint8_t adv_handle,             // 广播集句柄
    uint8_t cte_type,               // CTE 类型
    uint8_t cte_len,                // CTE 长度（单位：8 μs）
    uint8_t switching_pattern_len,  // 天线切换模板（发送 AoD 时使用）
    const uint8_t *switching_pattern);
```

CTE 类型 cte_type 包含 AoA（0）、AoD 1μs 切换（1）、AoD 1μs 切换（1）。

会导致的该函数失败的几种原因：

[2]更多信息请参考《Application Note: Direction Finding Solution》, https://ingchips.github.io/application-notes/an_aoa/sdk-support.html#proprietary-solution-2

- 指定的广播集未初始化（参见 “广播的配置”）；
- 指定的广播集不是扩展广播；
- 内存不足。

启动了扫描后，通过 ll_scanner_enable_iq_sampling 开始接收扩展广播上附着的 CTE：

```
// 函数执行成功时返回 0，否则返回非 0 值。
int ll_scanner_enable_iq_sampling(
    uint8_t cte_type,                 // 固定填 0
    uint8_t slot_len,                 // 时隙长度
    uint8_t switching_pattern_len,    // 天线切换模板（接收 AoA 时使用）
    const uint8_t *switching_pattern,
    uint8_t slot_sampling_offset,     // 时隙内采样偏移（0..23）
    uint8_t slot_sample_count);       // 时隙内采样数（1..5）
```

slot_len 的取值与 cte_slot_duration_type_t 相同。采样时，从每个时隙的 slot_sampling_offset / 24 µs 处开始，连续采样 slot_sample_count 次，采样频率 24MHz。slot_sampling_offset 与 slot_sample_count 的和必须小于等于 24。建议 slot_sampling_offset 取 12，slot_sample_count 取 1。

如果扫描未开始，则该函数将失败。扫描停止后，CTE IQ 采样同步停止。再次开始扫描时，需要重新调用该函数才能继续采样。

IQ 采样通过 HCI_SUBEVENT_LE_VENDOR_PRO_CONNECTIONLESS_IQ_REPORT 事件上报。

通过 ll_scanner_enable_iq_sampling_on_legacy[3] 可对传统广播的数据部分做 IQ 采样：

```
int ll_scanner_enable_iq_sampling_on_legacy(
    uint16_t sampling_offset,         // 采样起始位置（单位：比特）
    uint8_t cte_type,                 // 固定填 0
    uint8_t cte_time,                 // CTE 时长（单位：8 µs）
    uint8_t slot_len,                 // 时隙长度
    uint8_t switching_pattern_len,    // 天线切换模板（接收 AoA 时使用）
    const uint8_t *switching_pattern,
```

[3] ING918 不支持此功能。

```
    uint8_t slot_sampling_offset,    // 时隙内采样偏移（0..23）
    uint8_t slot_sample_count);      // 时隙内采样数（1..5）
```

与 ll_scanner_enable_iq_sampling 相比，这个函数增加了两个参数：sampling_offset 和 cte_time。sampling_offset 表示采样起始位置，0 对应于 Payload 的第 0 个比特。例如，要采样 ADV_IND 的 AdvData 部分，则把 sampling_offset 取做 $(6 \times 8 =)48$，以跳过长度为 6 个字节的 AdvA。IQ 采样通过 HCI_SUBEVENT_LE_VENDOR_PRO_CONNECTIONLESS_IQ_REPORT 事件上报。[4]。

此功能可能导致系统卡死或者产生 HardFault。务必配合硬件看门狗使用。

11.5 原始包（Raw Packet）对象

Raw Packet 对象是一种不透明的数据结构：

```
struct ll_raw_packet;
```

这部分接口是面向对象的，不同功能的对象使用不同的函数创建。销毁统一使用 ll_raw_packet_free 接口：

```
void ll_raw_packet_free(
    struct ll_raw_packet *packet); // 要销毁的对象
```

必须待工作完成才能销毁。工作完成时通过回调函数通知应用。回调函数类型如下：

```
typedef void (* f_ll_raw_packet_done)(
    struct ll_raw_packet *packet, // 产生回调的对象
    void *user_data);             // 用户数据
```

[4]关于原理、使用方法等更多信息请参考《使用 ING916 定位传统蓝牙设备》, https://ingchips.github.io/blog/2023-03-11-legacy-aoa/

不同功能的对象使用方法类似：1）创建对象；2）设置信道参数；3）设置数据；4）运行；5）在回调函数中读取数据和状态。对象的运行函数，如 `ll_raw_packet_send`，立即返回而不是阻塞到运行完成；只要运行函数返回了表示成功的值，回调函数就必然被调用。运行函数一旦返回了表示成功的值，在完成运行之前（即回调函数被调用前），不允许再调用对象的其它方法。

没有用于“取消运行”的接口。

11.5.1 无响应的包

无响应的 Raw Packet 通信包含两个角色：

- 发送方：在指定的时间发送一包数据，发送完成即任务完成；
- 接收方：在指定的时间窗口内持续接收数据，接收成功一包数据或者接收窗口结束时任务完成。

11.5.1.1 创建与配置

使用 `ll_raw_packet_alloc` 创建对象：

```
struct ll_raw_packet *ll_raw_packet_alloc(
    uint8_t for_tx,                // 角色（0: 接收方; 1: 发送方）
    f_ll_raw_packet_done on_done, // 回调函数
    void *user_data);              // 传给回调函数的用户数据
```

使用 `ll_raw_packet_set_param` 配置信道参数：

```
// 函数执行成功时返回 0，否则返回非 0 值。
int ll_raw_packet_set_param(
    struct ll_raw_packet *packet,
    int8_t tx_power,          // 发射功率（单位: dBm）
    int8_t rf_channel_id,     // 射频信道号
    uint8_t phy,              // PHY
    uint32_t access_addr,     // 接入地址
    uint32_t crc_init);       // CRC 初始值
```

射频信道号 rf_channel_id 范围为 0 ~ 39，当其值为 k 时，信道中心频率为 $(2402 + 2k)MHz$。PHY 的取值见表 11.1。

表 11.1： PHY 类型

取值	含义（发送方）	含义（接收方）
1	1M	1M
2	2M	2M
3	Coded S8	Coded
4	Coded S2	—

接入地址 access_addr 和 CRC 初始值 crc_init 的含义和用法与蓝牙核心规范一致。

11.5.1.2 发送

发送方通过 ll_raw_packet_set_tx_data 设置待发送的数据：

```
// 函数执行成功时返回 0，否则返回非 0 值。
int ll_raw_packet_set_tx_data(
    struct ll_raw_packet *packet,
    uint8_t header,     // 头数据（仅限低 2 比特）
    const void *data,   // 数据
    int size);          // 数据长度（单位：字节。最大 254）
```

如果附着了 CTE，size 最大支持 252 字节。当数据长度过长时，返回 1。

通过 ll_raw_packet_send 在指定的时刻发送：

```
// 可以在指定时刻发送时返回 0，否则返回非 0 值。
int ll_raw_packet_send(
    struct ll_raw_packet *packet,
    uint64_t when); // 发送时刻（单位：μs）
```

当 platform_get_us_time() 等于 when 时，启动发送。当 Controller 无法在指定的时间发送时（如与其它任务时间冲突，时间太迟等），返回非 0 值。ll_raw_packet_send(packet,

platform_get_us_time() + OFFSET)，当 OFFSET 小于一定值 T 时，会因为时间太迟而返回非 0 值。T 值与芯片处理能力有关，一般为 100 到几百微秒。其它接口的 when 参数用法与此相同。

发送完成后，如果需要再次发送相同的数据，直接调用 ll_raw_packet_send 即可，无需重新设置数据。

11.5.1.3 接收

通过 ll_raw_packet_recv 在一定时间窗口内尝试接收数据包：

```
// 可以在指定窗口接收则返回 0，否则返回非 0 值。
int ll_raw_packet_recv(
    struct ll_raw_packet *packet,
    uint64_t when,      // 接收窗口开始的时刻（单位：μs）
    uint32_t rx_window);// 窗口长度（单位：μs）
```

当 platform_get_us_time() 等于 when 时，开始尝试接收。这个函数立即返回，而非阻塞到接收完成再返回。从成功调用 ll_raw_packet_set_tx_data 到收到回调前，不可调用这个对象的其它方法。

在回调函数里调用 ll_raw_packet_get_rx_data 读取接收状态和数据：

```
// 返回接收状态
int ll_raw_packet_get_rx_data(
    struct ll_raw_packet *packet,
    uint64_t *air_time, // 数据包在空口出现的时间
    uint8_t *header,    // 头数据（仅 2 比特）
    void *data,         // 用来接收数据的缓存
    int *size,          // 接收到的数据包的长度
    int *rssi);         // 接收到的数据包的 RSSI
```

返回值解释：

- 0：成功接收，CRC 正确；
- 1：未接收到任何信息（指定的信道上无信号，或者任务未执行）；

- 2：接收到数据，但是出现接入码错误、CRC 错误等；
- 5：数据长度错误；
- 6：帧结构错误。

当出现非 0 错误码，但是错误码不属于 $\{1, 2\}$ 时，`air_time`、`header`、`rssi` 等 3 项输出有效。

11.5.1.4 附着 CTE

CTE 私有方案 #1[5] 使用 Raw Packet 发送 CTE。发送方通过 `ll_raw_packet_set_tx_cte` 附着 CTE：

```
int ll_raw_packet_set_tx_cte(
    struct ll_raw_packet *packet,
    uint8_t cte_type,
    uint8_t cte_len,
    uint8_t switching_pattern_len,
    const uint8_t *switching_pattern);
```

`ll_raw_packet_set_tx_cte` 各参数含义与 `ll_attach_cte_to_adv_set` 类似，不再赘述。需要 `ll_raw_packet_send(...)` 之前调用。

接收方通过 `ll_raw_packet_set_rx_cte` 设置 CTE 接收参数：

```
int ll_raw_packet_set_rx_cte(
    struct ll_raw_packet *packet,
    uint8_t cte_type,
    uint8_t slot_len,
    uint8_t switching_pattern_len,
    const uint8_t *swiching_pattern,
    uint8_t slot_sampling_offset,
    uint8_t slot_sample_count);
```

[5]更多信息请参考《Application Note: Direction Finding Solution》, https://ingchips.github.io/application-notes/an_aoa/sdk-support.html#proprietary-solution-2

ll_raw_packet_set_rx_cte 各参数含义与 ll_scanner_enable_iq_sampling 类似，不再赘述。需要 ll_raw_packet_recv(...) 之前调用。

完成接收后，在回调函数里通过 ll_raw_packet_get_iq_samples 获取 IQ 采样：

```
// 函数执行成功时返回 0，否则返回非 0 值。
int ll_raw_packet_get_iq_samples(
    struct ll_raw_packet *packet,
    void *iq_samples,    // 接收 IQ 采样的缓存
    int *iq_sample_cnt,  // 采样的数量
    int preprocess);     // 是否进行预处理
```

iq_samples 得到的数据顺序为“IQIQIQ……”。preprocess 为 0 时，不进行预处理，直接输出原始 IQ 采样数据，每个分量类型为 int16_t；preprocess 为非 0 时，进行预处理，输出处理后的 IQ 采样数据，每个分量类型为 int8_t。预处理仅支持 slot_sample_count 为 1 的情况。必须为 iq_samples 预留足够的空间。

函数返回的错误码如下：当未收到 CTE 时，该函数返回 1；preprocess 非 0 且 slot_sample_count 大于 1 时，返回 2。

对于发送方,提供了一种发送虚假 CTEInfo 数据域的方法:待调用了 ll_raw_packet_set_tx_cte 之后，通过 ll_raw_packet_set_fake_cte_info 填写虚假 CTEInfo 数据域。在发送时，CTE 按照 ll_raw_packet_set_tx_cte 指定的参数发送，但是数据包内协带的 CTEInfo 数据域却是修改过的。接收方处理 CTE 时将按照这个虚假的 CTEInfo 接收 CTE。

```
// 函数执行成功时返回 0，否则返回非 0 值。
int ll_raw_packet_set_fake_cte_info(
    struct ll_raw_packet *packet,
    uint8_t cte_type,
    uint8_t cte_len);
```

如果对象未通过 ll_raw_packet_set_tx_cte 配置 CTE，则返回 1；如果 CTE 参数错误，则返回 2。其它情况返回 0。

用途举例:通过 ll_raw_packet_set_tx_cte 设置发送 AoD;通过 ll_raw_packet_set_fake_cte 填入 AoA。这样，双方在发送和接收 CTE 时分别切换天线，在天线控制端口可产生严格同步的脉冲信号输出。

11.5.1.5 “裸包” 模式

无响应的 Raw Packet 通信默认使用与蓝牙规范一致的信道白化和 CRC 校验。另外提供一种关闭白化和 CRC 校验的“裸包”模式。在这种模式下，空口数据完全由开发者负责生成和校验，硬件只负责 PREAMBLE 和接入地址的比对。收发双方都通过 `ll_raw_packet_set_bare_mode` 启用“裸包”模式：

```
// 函数执行成功时返回 0，否则返回非 0 值。
int ll_raw_packet_set_bare_mode(
    struct ll_raw_packet *packet,
    uint8_t header, // 在数据包长之前发送的头数据
    int freq_mhz);  // 信道中心频点（单位：MHz）
```

仅支持发送 `header` 里部分比特，建议固定填写 0。当 `freq_mhz` 为 0 时，继续使用 `ll_raw_packet_set_param` 所指定的信道；当其非 0 时，需要注意仍可能对邻近蓝牙信道产生干扰；当其在蓝牙 2.4GHz 频段以外时，行为未知。当输入参数不合理时，这个函数将返回非 0 值。

通过 `ll_raw_packet_set_bare_data` 设置数据：

```
// 函数执行成功时返回 0，否则返回非 0 值。
int ll_raw_packet_set_bare_data(
    struct ll_raw_packet *packet,
    const void *data,       // 数据
    int size,               // 最大 255 字节
    uint32_t crc_value);    // 附加在数据之后的 CRC（低 24 比特）
```

在接收端回调里通过 `ll_raw_packet_get_bare_rx_data` 获取接收状态和数据：

```
// 函数执行成功时返回 0，否则返回非 0 值。
int ll_raw_packet_get_bare_rx_data(
    struct ll_raw_packet *packet,
    uint64_t *air_time,   // 数据包在空口出现的时间
```

```
    uint8_t *header,      // 头数据
    void *data,           // 用来接收数据的缓存
    int *size,            // 接收到的数据包的长度
    int *rssi,            // 接收到的数据包的 RSSI
    uint32_t *crc_value); // 附加在数据之后的 CRC（低 24 比特）
```

在指定的信道、指定的窗口未能与接入地址比对成功时，返回 2；否则返回 0。

11.5.2 带确认的包（Ack-able Packet）

带确认的包可以在通信双方之间可靠地向对方传输一包数据。通信双方分别称为：

- 发起者（Initiator）：在指定的时间开始发送数据，并在一定的时间窗口内等待对方发来的确认和数据包；
- 应答者（Responder）：从指定的时间开始尝试接收数据，如果成功，自动向对方发送确认信息和己方数据包。

在接收数据过程中，如果一方发现 CRC 错误，会自动请求对方重传。

11.5.2.1 创建与配置

通过 `ll_ackable_packet_alloc` 创建带确认的包对象：

```
// 创建成功返回对象指针；否则返回 NULL
struct ll_raw_packet *ll_ackable_packet_alloc(
    uint8_t for_initiator,        // 角色（1: 发起者; 0: 应答者）
    f_ll_raw_packet_done on_done, // 回调函数
    void *user_data);             // 传给回调函数的用户数据
```

使用 `ll_raw_packet_set_param` 配置参数。通过 `ll_ackable_packet_set_tx_data` 设置己方要发送数据（如不设置，则表示要发送的数据长度为 0）：

```
// 函数执行成功时返回 0，否则返回非 0 值。
int ll_ackable_packet_set_tx_data(
    struct ll_raw_packet *packet,
    const void *data, // 数据
    int size);        // 长度（单位：字节），最大 255 字节
```

11.5.2.2 运行

通过 ll_ackable_packet_run 设置对象的运行时机：

```
// 函数执行成功时返回 0，否则返回非 0 值。
int ll_ackable_packet_run(
    struct ll_raw_packet *packet,
    uint64_t when,      // 接收窗口开始的时刻（单位：μs）
    uint32_t window);   // 窗口长度（单位：μs）
```

window 至少应为 0.625 ms，建议取 1.25ms 以上。

收到回调后通过 ll_ackable_packet_get_status 获取己方数据的确认状态和接收到的数据：

```
// 返回对方数据的接收状态
int ll_ackable_packet_get_status(
    struct ll_raw_packet *packet,
    int *acked,         // 己方数据的确认状态
    uint64_t *air_time, // 对方数据包的空口时间
    void *data,         // 对方数据包的数据
    int *size,          // 对方数据包的大小
    int *rssi);         // 对方数据包的 RSSI
```

成功接收到对方数据时这个函数返回 0；否则返回其它错误码，无具体含义。acked 为 1 说明己方的数据被对方成功收到。

acked 是否为 1 与这个函数是否返回 0 没有必然联系。

11.5.3 信道监听

信道监听是在指定的信道上持续接收数据包，直到收到指定的数目或者接收窗口结束。可能的用途：

- 在单一信道上接收广播数据；
- 监听一个连接事件内通信双方的所有数据包；

  理想情况下（合理配置启动时间），收到的第 1 个 PDU 是主角色发送给从角色的，第 2 个是从角色发送给主角色的，依此类推。

11.5.3.1 创建与配置

通过 ll_channel_monitor_alloc 创建信道监听对象：

```
// 创建成功时返回对象指针，否则返回 NULL
struct ll_raw_packet *ll_channel_monitor_alloc(
    int pdu_num,     // 一次监听最多接收的包的数量
    f_ll_raw_packet_done on_done, // 回调函数
    void *user_data);             // 用户数据
```

使用 ll_raw_packet_set_param 配置参数。

11.5.3.2 运行

通过 ll_channel_monitor_run 运行监听对象：

```
// 函数执行成功时返回 0，否则返回非 0 值。
int ll_channel_monitor_run(
    struct ll_raw_packet *packet,
```

```
    uint64_t when,    // 监听启动时间（单位：μs）
    uint32_t window); // 窗长（单位：μs）
```

收到回调后，通过 ll_channel_monitor_check_each_pdu 依次遍历收到的数据包：

```
// 返回 `visitor` 的调用次数
int ll_channel_monitor_check_each_pdu(
    struct ll_raw_packet *packet,
    f_ll_channel_monitor_pdu_visitor visitor, // 访问者回调
    void *user_data);   // 用户数据
```

访问者回调的类型如下：

```
typedef void (* f_ll_channel_monitor_pdu_visitor)(
    int index,          // 包序号
    int status,         // 接收状态
    uint8_t reserved,   // 保留
    const void *data,   // 数据
    int size,           // 数据长度
    int rssi,           // RSSI
    void *user_data);   // 用户数据
```

index 从 0 开始，最大到 pdu_num - 1。status 为 0 表示该包成功接收；不为 0 时，表示接收失败，data、size、rssi 等输出皆无效。

ING918 仅支持获取 rssi。data、size 无输出。

通过 ll_channel_monitor_get_1st_pdu_time 获得第 1 个数据包的空口时间：

```
// 函数执行成功时返回 0，否则返回非 0 值。
int ll_channel_monitor_get_1st_pdu_time(
    struct ll_raw_packet *packet,
    uint64_t *air_time); // 空口时间
```

如果未成功收到数据包，这个函数会失败，返回非 0 值。

11.6 内存管理

Controller 管理了一个单独的堆空间，并对外提供接口供开发者使用。

使用 `ll_malloc` 分配内存：

```
// 分配失败时返回 NULL
void *ll_malloc(
    uint16_t size); // 大小（单位：字节）
```

使用 `ll_free` 释放内存：

```
void ll_free(void *buffer);
```

通过 `ll_get_heap_free_size` 获取当前未分配的空间大小：

```
// 返回当前未分配的空间大小（单位：字节）
int ll_get_heap_free_size(void);
```

过多地从 Controller 分配内存，将影响蓝牙功能。

11.7 低时延接口

11.7.1 ACL 预览

ACL 数据从 Controller 传输到 Host 再通知应用存在一定的时延。如果开发者需要更及时地获取 ACL 数据，可通过 `ll_register_hci_acl_previewer` 注册数据预览回调：

```
void ll_register_hci_acl_previewer(
    f_ll_hci_acl_data_preview preview);
```

回调函数的类型如下：

```
typedef void (*f_ll_hci_acl_data_preview)(
    uint16_t conn_handle,    // 连接句柄
    const uint8_t acl_flags, // ACL 标志位
    const void *data,        // 数据
    int len);                // 数据长度
```

ACL 标志位的含义如下：

- `0x01`：L2CAP 消息的一个接续片断或者空 PDU；
- `0x02`：L2CAP 新消息的第一个片断或者一条完整的 L2CAP 消息。

11.7.2 AES 加密

通过 GAP 接口进行AES-128 加密，存在一定的时延。通过 `ll_aes_encrypt` 可以以阻塞模式立即调用硬件完成加密并得到结果：

```
// 函数执行成功时返回 0，否则返回非 0 值。
int ll_aes_encrypt(
    const uint8_t *key,       // 密钥（小端模式）
    const uint8_t *plaintext, // 明文（小端模式）
    uint8_t *ciphertext);     // 密文（大端模式）
```

当硬件正忙，无法计算时，该函数返回非 0 值。使用时请注意参数的大小端模式与 gap_aes_encrypt 不同。

11.8 “非标”选项

11.8.1 锁频

锁频功能可将后续所有的射频行为（蓝牙广播、扫描、连接，原始包等）全部固定在指定的信道上。通过 ll_lock_frequency 开启锁定并指定频率：

```
void ll_lock_frequency(
    int freq_mhz); // 频率（单位：MHz）
```

锁频是一种底层设置，链路层并不知晓。例如，将广播者锁频在 2402 MHz，扫描者扫描 37 信道时，在一个广播件内并不能同时收到 3 个广播包。这是因为广播者链路层在发送 3 个广播包时仍然按照 37/38/39 设置白化参数，虽然它们最终都在 37 信道发送。

通过 ll_unlock_frequency 解除锁定：

```
void ll_unlock_frequency(void);
```

嵌套 ll_lock_frequency 时，可能产生意外效果，例如：

```
lock(f0);       // 锁定到 f0
    lock(f1);   // 锁定到 f1
    unlock();
    ...         // 此时依然处于锁频状态，锁定于 f1
unlock();
...             // 已解锁
```

11.8.2 自定义参数[6]

通过 ll_set_adv_access_address 替换蓝牙核心规范定义的广播包接入地址：

```
void ll_set_adv_access_address(
    uint32_t acc_addr);
```

通过 ll_override_whitening_init_value 替换蓝牙核心规范定义的白化初始值：

```
void ll_override_whitening_init_value(
    uint8_t override,  // 是否替换
    uint8_t value);    // 白化初始值
```

当 override 为 0 时，恢复使用规范值，value 参数忽略；为 1 时，value 的各比特依次放入移位寄存器的相应位置，即 lfsr[0] 填入最低比特，lfsr[1] 填入 (value >> 1) & 1，依此类推。使用这种表示法，规范为 37 信道定义的白化初始值表示为 0x53。

广播物理信道 PDU 包含长度为 4 个比特的 PDU Type。Controller 接收到的 PDU Type 为预留值时，整个 PDU 会被丢弃。通过 ll_allow_nonstandard_adv_type 可以使能接收一种非标 PDU Type：

```
void ll_allow_nonstandard_adv_type(
    uint8_t allowed, // 是否使能非标 ADV 接收
    uint8_t type);   // 非标类型值
```

当 allowed 为 0 时，功能关闭，type 参数忽略；为 1 时，扫描时会接收 type 类型的广播 PDU 并上报。

规范定义 CTE 比特为 1。通过 ll_set_cte_bit 可修改其定义：

```
void ll_set_cte_bit(
    uint8_t bit);  // CTE 比特（0 或 1）
```

[6]本节功能中，ING918 仅支持自定义连接间隔。

规范定义连接间隔时，单位为 1.25 ms。通过 `ll_set_conn_interval_unit` 可自定义该值：

```
void ll_set_conn_interval_unit(
    uint16_t unit); // （单位：⊠s）默认值：1250
```

此设置应该在连接建立前修改，通信双方必须使用相同的设置。`unit` 允许的最小值依赖硬件处理能力，ING918 为 800 µs 左右。将连接间隔调小，可明显降低通信时延。例如，标准中最小连接间隔为 $7.5ms$；使用 *extension* 包，ING918 连接间隔最小可降为 $(800 \times 1 =)800\mu s$。

11.9 ECC 引擎

当 Controller 未内置 ECC 硬件时，允许应用提供 ECC 引擎，并通过 HCI 为 Host 提供服务。通过 `ll_install_ecc_engine` 定义 ECC 引擎：

```
void ll_install_ecc_engine(
    f_start_generate_p256_key_pair start_generate_p256_key_pair,
    f_start_generate_dhkey start_generate_dhkey);
```

ECC 引擎应该能够安全地保存一个公钥对，并实现两个接口（回调函数），：

- `start_generate_p256_key_pair`：开始生成新的 P256 密钥对；
- `start_generate_dhkey`：开始生成 DHKey。

`f_start_generate_p256_key_pair` 的定义为：

```
typedef void (*f_start_generate_p256_key_pair)(void);
```

这个函数在 Controller 的上下文中调用，应该立即返回。当 P256 密钥对生成或出现错误后，通过 `ll_p256_key_pair_generated` 告知结果：

```
void ll_p256_key_pair_generated(
    int status,              // 是否成功（0: 成功）
    const uint8_t *pub_key); // 生成的公钥
```

status 表示是否成功生成了新的密钥对，0 为成功，其它值为不成功。当成功时，pub_key 包含新生成的公钥，长度为 64 字节，前 32 字节为 X 坐标，后 32 字节为 Y 坐标，大端模式。

f_start_generate_dhkey 的定义为：

```
typedef int  (*f_start_generate_dhkey)(
    int key_type,                   // 本端的密钥类型
    const uint8_t *remote_pub_key); // 远端的公钥
```

key_type 为 0 时，选用引擎生成的私钥；为 1 时使用规范定义的调试用私钥[7]。remote_pub_key 为远端的公钥，长度为 64 字节，前 32 字节为 X 坐标，后 32 字节为 Y 坐标，大端模式。

这个函数也是在 Controller 的上下文中调用，应该立即返回。当 DHKey 生成或出现错误后，通过 ll_p256_key_pair_generated 告知结果：

```
void ll_dhkey_generated(
    int status,              // 是否成功（0: 成功）
    const uint8_t *dh_key); // Diffie Hellman Key
```

status 为 0 时，表示 DHKey 生成成功，其它值为不成功。当成功时，dh_key 包含新生成的 DHKey，长度为 32 字节，大端模式。ECC 引擎必须首先检验远端公钥是否合法，如不合法，直接调用 ll_dhkey_generated，将 status 设为非 0 值。

[7]参见蓝牙核心规范 Vol 3, Part H, 2.3.5.6 LE Secure Connections pairing phase 2.

第十二章　杂项

12.1　接收 CTE

共有四种 CTE 接收、发送方式[1]。以 AoA 为例，各种方式的使用方法如下。

12.1.1　基于连接的 CTE 接收和发送

这种方式的使用可参考 SDK *Central CTE* 和 *Peripehral LED & CTE*。

12.1.1.1　发送方

建立连接后：

1. 调用 `gap_set_connection_cte_tx_param` 配置发送参数：

```
uint8_t gap_set_connection_cte_tx_param(
  // 连接句柄
  const hci_con_handle_t  conn_handle,
  // 允许发送的 CTE 类型组合
  const uint8_t           cte_types,
  // 用于 AoD 发送的天线切换模板的长度
  const uint8_t           switching_pattern_len,
  // 用于 AoD 发送的天线切换模板
  const uint8_t          *antenna_ids
);
```

[1]参考《Application Note: Direction Finding Solution》。

对于 AoA 模式，不需要配置天线切换模板，但是模板的长度必须至少为 2：

```
gap_set_connection_cte_tx_param(
  con_handle, (1 << CTE_AOA), 2, NULL);
```

2. 调用 gap_set_connection_cte_response_enable 使能 CTE 响应：

```
uint8_t gap_set_connection_cte_response_enable(
  // 连接句柄
  const hci_con_handle_t  conn_handle,
  // 是否使能
  const uint8_t           enable);
```

12.1.1.2 接收方

使用 ll_set_def_antenna 配置默认天线。建立连接后：

1. 调用 gap_set_connection_cte_rx_param 配置接收参数：

```
uint8_t gap_set_connection_cte_rx_param(
  // 连接句柄
  const hci_con_handle_t  conn_handle,
  // 是否使能 CTE 采样
  const uint8_t           sampling_enable,
  // 时隙长度
  const cte_slot_duration_type_t slot_durations,
  // 天线切换模板的长度
  const uint8_t           switching_pattern_len,
  // 天线切换模板
  const uint8_t          *antenna_ids);
```

时隙长度共有两种：

```
typedef enum
{
    CTE_SLOT_DURATION_1US = 1,
    CTE_SLOT_DURATION_2US = 2
} cte_slot_duration_type_t;
```

关于天线切换模板的更多信息请参考《Application Note: Direction Finding Solution》。

2. 调用 gap_set_connection_cte_request_enable 开始发送 CTE 请求

   连接模式的 CTE 为按需发送：一方发送一次 CTE 请求，对方就响应一次。

```
uint8_t gap_set_connection_cte_request_enable(
  // 连接句柄
  const hci_con_handle_t  conn_handle,
  // 是否使能
  const uint8_t           enable,
  // 发送 CTE 请求的间隔
  const uint16_t          requested_cte_interval,
  // 请求的 CTE 的长度（范围 2~20，单位 8μs）
  const uint8_t           requested_cte_length,
  // 请求的 CTE 的类型
  const cte_type_t        requested_cte_type);
```

   requested_cte_interval 表示每 requested_cte_interval 个连接间隔发送一次 CTE 请求，0 表示只发送一次。对于 AoA，requested_cte_type 为 CTE_AOA。

3. 响应 HCI_SUBEVENT_LE_CONNECTION_IQ_REPORT 事件

   使用 decode_hci_le_meta_event 解析事件内容：

```
const le_meta_conn_iq_report_t *rpt =
  decode_hci_le_meta_event(packet, le_meta_conn_iq_report_t);
```

   如果 CTE 请求失败（未收到响应），则会收到 HCI_SUBEVENT_LE_CTE_REQ_FAILED 事件。

12.1.2 基于周期广播的 CTE 接收和发送

这种方式的使用可参考 SDK *Periodic Advertiser* 和 *Periodic Scanner*。

12.1.2.1 发送方

使能周期广播后，

1. 调用 gap_set_connectionless_cte_tx_param 配置 CTE 发送参数

```
uint8_t gap_set_connectionless_cte_tx_param(
  // 广播句柄
  const uint8_t       adv_handle,
  // CTE 长度（范围 2~20，单位 8μs）
  const uint8_t       cte_len,
  // CTE 类型（对于 AoA，即 CTE_AOA）
  const cte_type_t    cte_type,
  // 一个周期广播里 CTE 发送次数
  const uint8_t       cte_count,
  // 用于 AoD 发送的天线切换模板的长度
  const uint8_t       switching_pattern_len,
  // 用于 AoD 发送的天线切换模板
  const uint8_t      *antenna_ids);
```

2. 调用 gap_set_connectionless_cte_tx_enable 使能 CTE 发送

```
uint8_t gap_set_connectionless_cte_tx_enable(
  // 广播句柄
  const uint8_t       adv_handle,
  // 是否使能
  const uint8_t       cte_enable);
```

12.1.2.2 接收方

与周期广播建立同步后，调用 gap_set_connectionless_iq_sampling_enable 启动 CTE 接收。

```
uint8_t gap_set_connectionless_iq_sampling_enable(
  // 同步句柄
  const uint16_t       sync_handle,
  // 是否使能采样
  const uint8_t        sampling_enable,
  // 时隙长度
  const cte_slot_duration_type_t slot_durations,
  // 每个周期广播间隔内最多接收多少个 CTE
  const uint8_t        max_sampled_ctes,
  // 天线切换模板长度
  const uint8_t        switching_pattern_len,
  // 天线切换模板
  const uint8_t       *antenna_ids);
```

此后就可以周期性收到 HCI_SUBEVENT_LE_CONNECTIONLESS_IQ_REPORT 事件，利用 decode_hci_le_meta_event 解析事件内容：

```
const le_meta_connless_iq_report_t *rpt =
  decode_hci_le_meta_event(packet, le_meta_connless_iq_report_t);
```

12.1.3 基于私有方式 #1 的 CTE 接收和发送

这种方式最为灵活，需要配置的参数也最多，可参考 SDK *Ext. Raw Packet Tx/Rx*。

12.1.4 基于私有方式 #2 的 CTE 接收和发送

这种方式的使用可参考 SDK *Central CTE* 和 *Peripehral LED & CTE*。

12.1.4.1 发送方

配置一个扩展广播集，属性设置为不可连接、不可扫描。待广播集使能后，调用 `ll_attach_cte_to_adv_set`[2] 为扩展广播附加 CTE。

12.1.4.2 接收方

启动扫描之后，调用 `ll_scanner_enable_iq_sampling`[3] 使能 CTE 采样。之后，通过 `HCI_SUBEVENT_LE_VENDOR_PRO_CONNECTIONLESS_IQ_REPORT` 事件获得 CTE 报告。

12.2 加密与解密

12.2.1 AES-128 加密

通过 `gap_aes_encrypt` 进行 AES-128 加密：

```
uint8_t gap_aes_encrypt(
  const uint8_t *key,           // 密钥（大端模式）
  const uint8_t *plaintext,     // 明文（大端模式）
  gap_hci_cmd_complete_cb_t cb, // 加密完成后的回调
  void *user_data);             // 回调函数的用户数据
```

以 FIPS 197[4] 里的数据为例：

```
PLAINTEXT: 00112233445566778899aabbccddeeff
KEY:       000102030405060708090a0b0c0d0e0f
CIPHERTEXT:69c4e0d86a7b0430d8cdb78070b4c55a
```

写成大端模式的数组形式：

[2]参考《Controller API Reference》。

[3]参考《Controller API Reference》。

[4]https://www.nist.gov/publications/advanced-encryption-standard-aes

```
static const uint8_t plain_text[] =
    {0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
     0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff};
static const uint8_t key[] =
    {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
     0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
static const uint8_t cipher_text[] =
    {0x69, 0xc4, 0xe0, 0xd8, 0x6a, 0x7b, 0x04, 0x30,
     0xd8, 0xcd, 0xb7, 0x80, 0x70, 0xb4, 0xc5, 0x5a};
```

加密：

```
gap_aes_encrypt(key, plain_text, aes_cb, NULL);
```

在回调函数 aes_cb 对比密文如下：

```
void aes_cb(const uint8_t *return_param, const uint8_t *user_data)
{
    uint8_t reversed[16];
    reverse_bytes(return_param + 1, reversed, sizeof(reversed));
    printf("Matched: %d\n",
      memcmp(reversed, cipher_text, sizeof(cipher_text)) == 0);
}
```

这里 return_param 的内容与规范中相应 HCI 命令的 Return Parameters 一致。密文采用小端模式，所以需要反转为大端模式，再与测试数据比较。

在上一次 AES 加密完成前，允许再次调用 gap_aes_encrypt。所有的加密任务保存在一个队列中，顺序完成。

也通过 ll_aes_encrypt 进行 AES-128 加密。使用时务必检查返回值，若不为 0 表示硬件正忙。遇到此情况时可稍后重试。

```
int ll_aes_encrypt(
  const uint8_t *key,       // 密钥（小端模式）
  const uint8_t *plaintext, // 明文（小端模式）
  uint8_t *ciphertext);     // 密文（大端模式）
```

ll_aes_encrypt 与 gap_aes_encrypt 的不同点：

- ll_aes_encrypt 以阻塞模式工作，不支持队列方式
- ll_aes_encrypt 可以更快完成加密
- 数据的大小端

12.2.2 AES-CCM

CCM 是 Cipher Block Chaining-Message Authentication Code (CBC-MAC) 和 Counter 模式（CTR）的组合，同时对数据加密和生成认证信息。通过 gap_start_ccm 启动 AES-CCM 计算[5]：

```
uint8_t gap_start_ccm(
        uint8_t  type,        // 类型: 0 为加密, 1 为解密
        uint8_t  mic_size,    // MIC 长度（4 或 8 字节）
        uint16_t msg_len,     // 消息长度（最长 384 字节）
        uint16_t aad_len,     // AAD 长度（最长 16 字节, 可为 0）
        uint32_t tag,         // 数据标签（任意值）
        const uint8_t *key,   // 密钥
        const uint8_t *nonce, // Nonce（长度 13 字节）
        const uint8_t *msg,   // 消息
        const uint8_t *aad,   // AAD
        uint8_t *out_msg,     // 加解密输出
        gap_hci_cmd_complete_cb_t cb, // 计算完成后的回调
        void *user_data);             // 回调函数的用户数据
```

AAD 为补充认证证数据（Additional Authenticated Data），可为数据提供额外的完整性保护。这个函数使用了自定义的 HCI 消息 HCI_VENDOR_CCM。由于消息数据量大，这条消息仅传递

[5]为防止与 Media Access Controller 混淆，蓝牙规范使用 MIC 代替 MAC。

指针，因此所有的数据（key、nonce、aad、msg、out_msg）必须一直存在，直到计算完成方可释放。加密时，输出到 out_msg 的数据长度为 (msg_len + mic_size)；解密时，msg 传入的数据长度为 (msg_len + mic_size)。

回调函数收到的 return_param 参数转换自 event_vendor_ccm_complete_t *。

例如：

```
static const uint8_t plain_text[] =
    {0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
     0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff};
static const uint8_t key[] =
    {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
     0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

#define MSG_LEN     16
#define MIC_SIZE    4

static const uint8_t nonce[13] = {1};
static uint8_t ccm_enc_out[MSG_LEN + MIC_SIZE];
static uint8_t ccm_dec_out[MSG_LEN];
```

加密：

```
gap_start_ccm(0, MIC_SIZE, MSG_LEN, 0, 0,
            key, nonce, plain_text, NULL,
            ccm_enc_out, ccm_enc_cb, NULL);;
```

在回调函数 ccm_enc_cb 里解密：

```
void ccm_enc_cb(const uint8_t *return_param, const uint8_t *user_data)
{
  const event_vendor_ccm_complete_t *complete =
    (const event_vendor_ccm_complete_t *)return_param;
```

```
  gap_start_ccm(1, MIC_SIZE, MSG_LEN, 0, 0,
        key, nonce, ccm_enc_out, NULL,
        ccm_dec_out, ccm_dec_cb, NULL);
}
```

在回调函数 ccm_dec_cb 里检查 MIC 验证状态并比对数据：

```
void ccm_dec_cb(const uint8_t *return_param, void *user_data)
{
  const event_vendor_ccm_complete_t *complete =
    (const event_vendor_ccm_complete_t *)return_param;
  printf("CCM DEC Status: %d\n", complete->status);
  printf("Matched: %d\n",
    memcmp(ccm_dec_out, plain_text, sizeof(plain_text)) == 0);
}
```

event_vendor_ccm_complete_t 的定义如下：

```
typedef struct
{
    uint8_t  status;   // 状态码
    uint8_t  type;
    uint8_t  mic_size;
    uint16_t msg_len;
    uint16_t aad_len;
    uint32_t tag;
    uint8_t *out_msg;
} event_vendor_ccm_complete_t;
```

状态码在加密时，总是为 0；在解密时，0 表示 MIC 验证通过，非 0 表示失败。其它域与传入 gap_start_ccm 的参数一一对应、相等。

在上一次 AES-CCM 计算完成前，允许再次调用 gap_start_ccm。所有的计算任务保存在一个队列中，顺序完成。

12.3 协议栈配置

协议栈提供若干配置项，通过 btstack_config 配置：

```
void btstack_config(
  // btstack_config_item 的比特组合
  uint32_t flags);
```

每个配置项对应一个比特位，可通过 “或” 运算同时使能多项配置。

```
enum btstack_config_item {
    STACK_ATT_SERVER_ENABLE_AUTO_DATA_LEN_REQ = ...,
    STACK_GATT_CLIENT_DISABLE_AUTO_DATA_LEN_REQ = ...,
    STACK_DISABLE_L2CAP_TIMEOUT = ...,
    STACK_SM_USE_FIXED_CSRK = ...
    STACK_GATT_CLIENT_DISABLE_MTU_EXCHANGE = ...
    ...
};
```

12.3.1 Data Length 与 MTU

低功耗蓝牙进入连接模式后，各层分别协商通信中数据包的大小，对于 ATT 层，由 MTU EXCHANGE 流程实现；对于链路层，由 DATA LENGTH 更新流程实现。

按照规范，进入连接模式后，DATA LENGTH 更新流程可以由主或从设备在任何时刻发起。这导致了一个问题：某些芯片无法处理对方设备 “随时” 发起的 DATA LENGTH 更新流程[6]。为了更新地兼容不同的芯片，协议栈定义了两个配置项：

- STACK_ATT_SERVER_ENABLE_AUTO_DATA_LEN_REQ
- STACK_GATT_CLIENT_DISABLE_AUTO_DATA_LEN_REQ

这两个配置分别控制 GATT Server、Client 在 MTU EXCHANGE 时是否自动发起 DATA LENGTH 更新流程。默认情况下，Servier 不会自动发起更新流程，而 Client 会自动发起。

[6]https://ingchips.github.io/blog/2021-06-02-sdk-6/#%E5%85%BC%E5%AE%B9%E6%80%A7

12.3.2 面向测试

STACK_DISABLE_L2CAP_TIMEOUT 用于防止 L2CAP 超时，STACK_SM_USE_FIXED_CSRK 用来使用固定的 CSRK。

当 GATT 客户端实例创建时，会自动发起 MTU 协商。通过 STACK_GATT_CLIENT_DISABLE_MTU_EXCHANGE 可关闭该功能，MTU 保持初始默认值 ATT_MTU。

这些配置项用于 PTS 测试。

12.4 API 返回值

绝大多数协议栈 API 都带有 uint8_t 型的返回值，0 为成功，非 0 为错误。开发者需要关注这些返回值。

几个例子：

- att_server_notify 的返回值：
  - 0：成功
  - BTSTACK_LE_CHANNEL_NOT_EXIST (0x59)：连接不存在（连接句柄参数错误？）
  - BTSTACK_ACL_BUFFERS_FULL (0x57)：内存已满
- gap_set_ext_adv_para 的返回值：
  - 0：成功
  - BTSTACK_MEMORY_ALLOC_FAILED (0x56)：内存已满

12.5 键值存储接口与实现

12.5.1 键值存储接口

协议栈定义了一个简单的键值存储接口（*kv_storage*），其键（*key*）为 uint8_t，值（*value*）为长度不超过 KV_VALUE_MAX_LEN 的数组。开发者可以在 App 里使用这个接口，*key* 的取值范围应该在 KV_USER_KEY_START 和 KV_USER_KEY_END 之间。这个存储模块的增、删、改、查等功能如下。

- 增/改：kv_put

```
// 如果 key 不存在，为“增”；如果 key 存在，为“改”
int kv_put(
  const kvkey_t key,
  // 值
  const uint8_t *data,
  // 值的长度
  int16_t len);
```

- 删：kv_remove

```
void kv_remove(
  // 键
  const kvkey_t key)
```

- 清空：kv_remove_all

```
void kv_remove_all(void);
```

- 查：kv_get

```
// 返回：指向值的指针
uint8_t *kv_get(
  // 键
  const kvkey_t key,
  // 输出：值的长度
  int16_t *len);
```

这个 API 返回的指针直接指向模块内值的存储空间，允许开发者在不改变值的长度的前提下修改其的内容。修改之后需要调用 kv_value_modified 告知存储模块。

- 遍历：kv_visit

```
void kv_visit(
  // 访问者回调
  f_kv_visitor visitor,
  // 传递给回调的用户参数
  void *user_data);
```

使用这个 API 可以遍历存储内所有的键值对。

- 数据已被修改

```
void kv_value_modified_of_key(
  // 键
  const kvkey_t key);
```

协议栈内实现了这个接口。开发者既可以使用内置的默认实现，也可以自定义实现。

12.5.2 默认的键值存储实现

默认的键值存储实现的总储存大小为 1024 字节。这个实现本身没有数据持久化。持久化需要开发者通过 kv_init[7] 提供回调来实现：

```
void kv_init(
  // 用来保存数据的回调
  f_kv_write_to_nvm f_write,
  // 用来读取（恢复）数据的回调
  f_kv_read_from_nvm f_read);
```

当键值存储模块初始化时，会调用 f_read 恢复之前的数据状态；当存储里的数据更新后，键值存储模块会自动调用 f_write 回调。考虑到 Flash 不宜频繁擦写，键值存储模块通过定时器超时来触发写入。每当数据更新时，复位定时器。

使用这种实现时需要注意以下几点：

[7]只能在 app_main 就调用。

1. 该模块查找一个 *key* 的时间复杂度为 $\mathcal{O}(n)$；
2. 该模块不是线程安全的。

12.5.3 自定义键值存储实现

通过在 app_main 里调用 kv_init_backend 可以自定义键值存储实现：

```
void kv_init_backend(
  const kv_backend_t *backend);
```

其中 kv_backend_t 包含以下回调接口：

```
typedef struct kv_backend
{
    // 清空
    f_kv_remove_all kv_remove_all;
    // 删除
    f_kv_remove     kv_remove;
    // 增/改
    f_kv_put        kv_put;
    // 查
    f_kv_get        kv_get;
    // 遍历
    f_kv_visit      kv_visit;
    // 数据已被修改
    f_kv_value_modified_of_key kv_value_modified_of_key;
} kv_backend_t;
```

注意 app_main 里不能既调用 kv_init 又调用 kv_init_backend，会导致混乱。当 app_main 里既没有调用 kv_init 也没有调用 kv_init_backend 时，会使用默认的键值存储实现，且不支持数据持久化。

12.6 设备数据库

设备数据库模块（*le_device_db*）负责存储、管理设备的绑定信息。这个模块是基于键值存储模块实现的，所能存储的设备个数等于 max(软件包所支持的连接数目, 10)[8]。删、查等接口如下。

- 查：le_device_db_find

```
le_device_memory_db_t *le_device_db_find(
  // 待查设备的地址类型
  const int addr_type,
  // 待查设备的地址
  const bd_addr_t addr,
  // 输出：在数据库里的序号
  int *index);
```

- 删

直接按地址删除：

```
void le_device_db_remove(
  // 待删设备的地址类型
  const int addr_type,
  // 待删设备的地址
  const bd_addr_t addr);
```

根据在数据库里的编号删除：

```
void le_device_db_remove_key(
  // 待删设备在数据库里的编号
  int index);
```

[8]对于 v8.4.12 或更旧的版本，所能存储的设备个数等于软件包所支持的连接数目。

- 遍历

  这个模块支持迭代器遍历：

```
le_device_memory_db_iter_t iter;
le_device_db_iter_init(&iter);
while (le_device_db_iter_next(&iter))
{
    le_device_memory_db_t *cur = le_device_db_iter_cur(&iter);
    // ...
}
```

12.7 同步版 API

SDK 提供的工具模块 *btstack_sync* 里包含几个 GAP、GATT 客户端同步版本的 API。要使用这些 API 必须先初始化同步执行器。

- v8.2 及以上版本（基于 `btstack_push_user_runnable` 实现）

  如下创建同步执行器对象：

```
struct gatt_client_synced_runner *synced_runner =
    gatt_client_create_sync_runner(enable_gap_api);
```

  这里的 `enable_gap_api` 表示是否使能 GAP 同步版 API。

- v8.2 以下版本（基于 `btstack_push_user_msg` 实现）

  - v8.2 以下版本的 *gatt_client_util* 模块仅提供 GATT 客户端同步版本 API，未提供 GAP 同步版 API —— 开发者可参考 v8.2 自行添加。

  1. 基于 `btstack_push_user_msg` 实现一个消息传递函数，

```
// runner 为同步执行器
// msg_id 为同步执行器内部的消息，从其数据类型可看出最多只有 256 种 ID，
// 可以映射到 btstack_push_user_msg 的 uint32_t 型消息 ID 的一个子集里。
void synced_push_user_msg(
    struct gatt_client_synced_runner *runner,
    uint8_t msg_id)
{
    // 把同步执行器消息映射到 USER_MSG_SYNC_MSG_START 以上
    // App 可以使用的消息 ID 范围是 [0..USER_MSG_SYNC_MSG_START - 1]
    btstack_push_user_msg(USER_MSG_SYNC_MSG_START + msg_id,
        runner, 0);
}
```

2. 更新 user_msg_handler 把同步执行器内部的消息交给 gatt_client_sync_handle_msg

```
static void user_msg_handler(uint32_t msg_id, void *data,
    uint16_t size)
{
    switch (msg_id)
    {
    //...
    default:
        if (msg_id >= USER_MSG_SYNC_MSG_START)
        {
            struct gatt_client_synced_runner *runner =
                (struct gatt_client_synced_runner *)data;
            gatt_client_sync_handle_msg(runner,
                msg_id - USER_MSG_SYNC_MSG_START);
        }
    }
}
```

3. 创建同步执行器对象

```
struct gatt_client_synced_runner *synced_runner =
    gatt_client_create_sync_runner(synced_push_user_msg);
```

注意：对于一个连接，最多只能存在一个与之关联的同步执行器。最简单的用法是在初始化时创建唯一一个同步执行器对象[9]。创建多个同步执行器时，最多只能有一个使能了 GAP 同步版 API。

上述准备工作做完，就可以使用同步 API 了。下面这个函数 —— 称为一个同步执行体 —— 使用同步 API 多次读取某个特征的值并打印：

```
// 用 user_data 表示 value_handle
static void demo_synced_api(
    struct gatt_client_synced_runner *synced_runner,
    void *user_data)
{
    uint16_t handle = (uint16_t)(uintptr_t)user_data;
    static uint8_t data[255];
    int n = 5;
    printf("synced read value for %d times:\n", n);

    for (; n > 0; n--)
    {
        uint16_t length = sizeof(data);
        // 注意: 这个函数返回后, 数据就读取完成了
        int err = gatt_client_sync_read_value_of_characteristic(
                synced_runner, mas_conn_handle, handle,
                data, &length);
        printf("[%d]: err = %d:\n", n, err);
        if (err) break;
        // print_value(data, length);
        printf("wait for 200ms...\n", n, err);
        vTaskDelay(pdMS_TO_TICKS(200));
    }
```

[9]为多个连接创建多个同步执行器的优势在于多个 GATT 客户端上的会话可以并发。

```
    printf("done\n\n");
}
```

这种同步执行体不能直接使用，而要交给同步执行器去执行：

```
gatt_client_sync_run(synced_runner, demo_synced_api,
    (void *)(uintptr_t)value_handle);
```

这里，demo_synced_api 函数运行于同步执行器内的线程。gatt_client_sync_run 可以从任意线程调用。

12.7.1 GAP 同步 API

- gap_sync_ext_create_connection：建立连接

```
int gap_sync_ext_create_connection(
    struct btstack_synced_runner *runner,
    const initiating_filter_policy_t filter_policy,
    const bd_addr_type_t own_addr_type,
    const bd_addr_type_t peer_addr_type,
    const uint8_t *peer_addr,
    const uint8_t initiating_phy_num,
    const initiating_phy_config_t *phy_configs,
    uint32_t timeout_ms,
    le_meta_event_enh_create_conn_complete_t *complete);
```

与 gap_ext_create_connection 相比，这个函数是“阻塞”的：函数直到连接成功或者超时才返回。如果调用 gap_create_connection 时出错，则 gap_ext_create_connection 会返回其错误码；其它情况下返回的值即 le_meta_event_enh_create_conn_complete_t 里的 status 字段。也就是说，连接建立成功时返回值为 0 时，超时时返回值为 0x02[10] (未知的连接句柄)。

[10]由规范规定。

complete 参数为输出，保存了 HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE 事件的数据。同使用 gap_ext_create_connection 一样，已有的 HCI 事件回调函数也会收到这个 HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE 事件。

- gap_sync_le_read_channel_map：读取某连接所使用的信道集合

  函数原型如下：

```
int gap_sync_le_read_channel_map(
  struct btstack_synced_runner *runner,
  // 连接句柄
  hci_con_handle_t con_handle,
  // 用前 37 个比特表示的信道集合
  uint8_t channel_map[5]);
```

- gap_sync_read_rssi：读取某连接的 RSSI

  函数原型如下：

```
int gap_sync_read_rssi(
  struct btstack_synced_runner *runner,
  // 连接句柄
  hci_con_handle_t con_handle,
  // RSSI 输出
  int8_t *rssi);
```

- gap_sync_read_phy：读取某连接的 PHY

  函数原型如下：

```
gap_sync_read_phy(
  struct btstack_synced_runner *runner,
  // 连接句柄
  hci_con_handle_t con_handle,
  // 本设备发送方向使用的 PHY
  phy_type_t *tx_phy,
```

```
  // 本设备接收方向使用的 PHY
  phy_type_t *rx_phy);
```

- gap_sync_read_remote_version：读取某连接对端设备的版本

  函数原型如下：

```
int gap_sync_read_remote_version(
  struct btstack_synced_runner *runner,
  // 连接句柄
  hci_con_handle_t con_handle,
  // 蓝牙链路层协议版本编号
  uint8_t *version,
  // 厂家编号
  uint16_t *manufacturer_name,
  // Controller 版本号
  uint16_t *subversion);
```

  蓝牙链路层协议版本号由 Bluethooth SIG 指定，部分编号与版本号的对应关系如表 2.5 所示；厂家编号由厂家申请、Bluethooth SIG 指定[11]；Controller 版本号由 Controller 厂家自行指定。

- gap_sync_read_remote_used_features：读取某连接对端设备使用的蓝牙特性

  函数原型如下：

```
int gap_sync_read_remote_used_features(
  struct btstack_synced_runner *runner,
  // 连接句柄
  hci_con_handle_t con_handle,
  // 蓝牙特性比特图
  uint8_t features[8]);
```

  蓝牙特性的定义参见表 2.4。

[11] https://www.bluetooth.com/specifications/assigned-numbers/

12.7.2 GATT 客户端同步 API

本模块提供了以下同步 API，其原型与异步版本基本类似：

- `gatt_client_sync_discover_all`：同步发现所有服务
- `gatt_client_sync_read_value_of_characteristic`：同步读取特征的值
- `gatt_client_sync_read_characteristic_descriptor`：同步读取特征描述符
- `gatt_client_sync_write_value_of_characteristic`：同步有响应地写入特征的值
- `gatt_client_sync_write_value_of_characteristic_without_response`：同步无响应地写入特征的值[12]
- `gatt_client_sync_write_characteristic_descriptor`：同步写入特征描述符

12.8 线程安全的 API

SDK 提供的工具模块 *btstack_mt* 借助 `btstack_push_user_runnable` 为 Host 常用 API 重新封装了一套线程安全的 API。每个 API 与原始 API 参数完全一致，得到的返回值也相同，只是函数名称增加了表示多线程的 `mt_` 前辍。

每个 `mt_foo` 函数都有一个 `f_btstack_user_runnable` 类型的辅助函数 `foo_0`，调用背后的 `foo` 所需要的参数及保存返回值的变量等都通过第一个参数 `ctx` 传入，伪代码如下：

```
static void foo_0(*ctx, uint16_t _)
{
  ctx->_ret = foo(ctx->param);
  event_set(ctx->_event);
}
```

`mt_foo` 的伪代码如下：

[12]这个函数的原始版本不是严格意义上的异步操作。考虑到在一个同步执行体内可能既会用到有响应的写入，也会用到无响应的写入，加入这个 API 可以带来便利。

```
type mt_foo(param)
{
  if (in Host task)
    return foo(param);
  ctx->param = param;
  ctx->_event = event_create();
  btstack_push_user_runnable(foo_0, &ctx, 0);
  event_wait(ctx->_event);
  event_free(ctx->_event);
  return ctx->_ret;
}
```

由于通用 OS 接口未提供释放的接口，所以这个模块实现了一个事件对象池以模拟事件的释放（伪代码里的 event_free）。

用上述“阻塞”方式实现的线程安全 API，既可以获取实际的返回值，也可以避免复制内存数据。允许这样的用法：

```
void do_some_thing()
{
  uint8_t data[10];
  向 data 写入数据;
  mt_att_server_notify(con_handle,
    att_handle, data, sizeof(data));
}

void any_thread()
{
  do_some_thing();
  // 此时 do_some_thing 里的 data 已被释放
}
```

注意事项：

- 这个模块依赖于一个真正的 RTOS。使用 NoOS 软件包时，必需提供真正的队列及事件支持。

- 封装必然存在开销。对于高性能、高实时性的应用，不推荐使用。反之，对于相对简单的应用，则推荐使用，可以简化代码；
- 不要在中断服务程序内使用这些 API[13]。

12.9 链路层隐私

下面以充分保护隐私为出发点说明链路层隐私特性的使用方法。

12.9.1 将已配对设备添加到解析列表

同白名单类似，地址解析列表也完全由开发者管理。下面的代码演示了如何将设备数据库里的信息添加到解析列表和白名单。

```
static void populate_pairing_data(const uint8_t *local_irk)
{
    gap_clear_resolving_list();
    gap_clear_white_lists();

    le_device_memory_db_iter_t device_db_iter;
    le_device_db_iter_init(&device_db_iter);
    while (le_device_db_iter_next(&device_db_iter))
    {
        const le_device_memory_db_t *dev =
            le_device_db_iter_cur(&device_db_iter);
        gap_add_dev_to_resolving_list(dev->addr,
            dev->addr_type, dev->irk, local_irk);
        gap_add_whitelist(dev->addr, dev->addr_type);
    }
}
```

[13]可以使用 btstack_push_user_runnable。

上面的演示代码假定对于所有的配对设备，本端使用的 IRK 相同。与不同的设备配对时，本端使用不同的 IRK 也是可以的。

12.9.2 广播

12.9.2.1 未配对时

当设备未配对或者期望被新的设备连接时，可事先生成不可解析随机地址或用 IRK 生成可解析地址，将其配置为广播集的随机地址，并使用这个随机地址发送广播，即：

```
gap_set_adv_set_random_addr(..., 生成的地址);
gap_set_ext_adv_para(
  ...
  BD_ADDR_TYPE_LE_RANDOM, // own_addr_type
  ...);
```

12.9.2.2 已配对时

当设备已与唯一的对端设备配对，发送广播等待与其建立连接时，将对端设备的身份地址填入 gap_set_ext_adv_para 的 peer_addr 参数，Controller 从解析列表中查到对应的本端 IRK，使用与之对应的可解析地址（如地址不存在，会自动生成）发送广播：

```
gap_set_ext_adv_para(
  ...
  BD_ADDR_TYPE_LE_RESOLVED_RAN, // own_addr_type
  peer_addr_type,               // peer_addr_type
  peer_addr,                    // peer_addr
  ADV_FILTER_ALLOW_SCAN_WLST_CON_WLST, // adv_filter_policy
  ...);
```

当设备已与多个对端设备配对，发送广播等待与任意对端建立连接时，分为两种情况：

1. 使用了相同的本端 IRK

   由于使用任意对端的身份地址都会检索到这个 IRK，所以可以任意选择一个对端设备的身份地址填入 gap_set_ext_adv_para 的 peer_addr 参数，代码同上。

2. 使用了各不相同的本端 IRK

   可以轮流使用每个对端设备身份地址，填入 gap_set_ext_adv_para 的 peer_addr 参数，发送广播并等待一段时间，如无连接则切换到下一个设备。或者用多个对端设备身份地址同时配置多个广播集。

如果地址解析成功，HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE_.. 事件里的 peer_addr_type 将会是 BD_ADDR_TYPE_LE_RESOLVED_RAN 或者 BD_ADDR_TYPE_LE_RESOLVED_PUB，peer_addr 是解析出的对端的身份地址。HCI_SUBEVENT_LE_SCAN_REQUEST_RECEIVED 事件里的 scanner_addr_type 将会是 BD_ADDR_TYPE_LE_RESOLVED_RAN 或者 BD_ADDR_TYPE_LE_RESOLVED_PUB，scanner_addr 是解析出的扫描者的身份地址。

12.9.3 扫描

可以生成不同于身份地址（如不可解析随机地址或用 IRK 生成可解析地址）的随机地址，并调用 gap_set_random_device_address。

如果需要扫描周围全部设备的广播信息，可如下配置扫描参数：

```
gap_set_ext_scan_para(
  BD_ADDR_TYPE_LE_RANDOM,               // own_addr_type
  SCAN_ACCEPT_ALL_EXCEPT_NOT_DIRECTED,  // filter_policy
  ...);
```

如果只扫描白名单内设备的广播信息，可如下配置扫描参数：

```
gap_set_ext_scan_para(
  BD_ADDR_TYPE_LE_RANDOM,                // own_addr_type
  SCAN_ACCEPT_WLIST_EXCEPT_NOT_DIRECTED, // filter_policy
  ...);
```

此时，Controller 扫描到某设备的广播时，尝试解析地址。对于主动扫描，如果解析成功且通过了白名单，就使用对应于本端 IRK 的可解析地址发送扫描请求。

如果地址解析成功，`HCI_SUBEVENT_LE_EXTENDED_ADVERTISING_REPORT` 事件里的 `addr_type` 将会是 `BD_ADDR_TYPE_LE_RESOLVED_RAN` 或者 `BD_ADDR_TYPE_LE_RESOLVED_PUB`，`address` 是解析出的广播者身份地址。

12.9.4 建立连接

事先生成不可解析随机地址或用 IRK 生成可解析地址，并调用 `gap_set_random_device_address`。

12.9.4.1 未配对时

如果需要连接一个未配对的设备，可如下配置：

```
gap_ext_create_connection(
  INITIATING_ADVERTISER_FROM_PARAM,   // filter_policy
  BD_ADDR_TYPE_LE_RANDOM,             // own_addr_type
  peer_addr_type, // peer_addr_type
  peer_addr,      // peer_addr
  ...);
```

12.9.4.2 已配对时

如果需要连接单一的确定的已配对设备，可如下配置：

```
gap_ext_create_connection(
  INITIATING_ADVERTISER_FROM_PARAM,   // filter_policy
  BD_ADDR_TYPE_LE_RESOLVED_RAN,       // own_addr_type
  peer_addr_type, // peer_addr_type
  peer_addr,      // peer_addr
  ...);
```

如果需要连接白名单内的任一已知设备，可如下配置：

```
gap_ext_create_connection(
  INITIATING_ADVERTISER_FROM_LIST,   // filter_policy
  BD_ADDR_TYPE_LE_RESOLVED_RAN,      // own_addr_type
  ...);
```

如果地址解析成功，HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE_.. 事件里的 peer_addr_type 将会是 BD_ADDR_TYPE_LE_RESOLVED_RAN 或者 BD_ADDR_TYPE_LE_RESOLVED_PUB，peer_addr 是解析出的对端的身份地址。

第十三章　协议栈能力

对于不同的软件包、不同的芯片系列，协议栈能力不同，汇总[1]于表 13.1，表 13.2 和表 13.3。

表 13.1: {*typical*, *extension*, *exp*} 软件包协议栈能力

系列	广播集数目	连接数目	白名单容量	CTE	最大 MTU
ING9188X	8	8	16	✓	247
ING9187X	8	8	16		247
ING9168X	5	5	8	✓	247

表 13.2: {*mass_conn*} 软件包协议栈能力

系列	广播集数目	连接数目	白名单容量	CTE	最大 MTU
ING9188X	8	26	24	✓	247
ING9187X	8	26	24		247
ING9168X	5	10	10	✓	247

表 13.3: {*mini*} 软件包协议栈能力

系列	广播集数目	连接数目	白名单容量	CTE	最大 MTU
ING9188X	1	1	4	✓	247
ING9187X	1	1	4		247
ING9168X	1	1	4	✓	247

通过 `ll_get_capabilities` 和 `btstack_get_capabilities` 可分别获取链路层和 Host 的协议栈能力[2]。

[1]依据 SDK v8.3.7。ING916XX 协议栈能力可能发生变化。

[2]限 SDK v8.4.13 或更高版本。