低功耗蓝牙开发者手册

Ingchips Technology Co., Ltd.

目录

版本历史 **xiii**

第一章 简介 **1**

1.1 缩略语及术语 1

1.2 参考文档 2

第二章 概览 **3**

2.1 基本原则 3

2.2 协议栈架构 3

2.3 通信模型 5

2.4 回调函数事件包 6

2.5 事件包的解析 7

2.6 Controller 错误码 11

2.7 Controller 特性定义 12

2.8 蓝牙规范版本编号 13

2.9 白名单 14

2.10 异步特性 14

2.11 线程安全性 14

2.12 BLE 设备地址 15

第三章 GAP - 广播 **17**

3.1 概览 17

3.1.1 类型 17

3.1.2 过滤策略 18

3.1.3 PHY 18

3.1.4 广播集 19

3.1.5 相关事件 19

3.2 使用说明 19

3.2.1 配置广播 19

3.2.2 广播数据 21

3.2.3 配置周期广播 23

3.2.4 起停广播 23

3.2.5 起停周期广播 24

3.2.6 为周期广播添加 CTE 24

第四章 GAP - 扫描 **25**

4.1 概览 25

4.1.1 间隔与窗口 25

4.1.2 过滤策略 25

4.1.3 主动与被动 26

4.1.4 PHY 26

4.2 使用说明 26

4.2.1 配置参数 26

4.2.2 起停扫描 27

4.2.3 处理数据 28

4.2.4 与周期广播同步 29

第五章 GAP - 连接 **31**

5.1 概览 31

5.2 使用说明 31

5.2.1 建立连接 31

5.2.2 取消连接 33

5.2.3 获取对端版本 33

5.2.4 获取对端特性 33

5.2.5 设置 PHY 33

5.2.6 更新连接参数 34

5.2.7 减速模式 35

5.2.8 路损检测与上报 37

5.2.9 功率控制 39

第六章 GATT - 服务器 **41**

6.1 概览 41

6.2 使用说明 44

6.2.1 Profile 数据 44

6.2.2 实现读回调 45

6.2.3 实现写回调 47

6.2.4 发送通知（Notification） 48

6.2.5 发送指示（Indication） 48

6.2.6 响应事件 49

6.2.7 ATT_MTU 50

第七章 GATT - 客户端 **51**

7.1 概览 51

7.1.1 句柄范围 52

7.2 使用说明 52

7.2.1 创建客户端 52
7.2.2 发现服务 52
7.2.3 读取特征 55
7.2.4 写入特征 57
7.2.5 订阅特征 59
7.2.6 ATT_MTU 60

第八章 L2CAP **63**
8.1 概览 63
8.2 使用说明 63
8.2.1 从端请求更新连接参数 63

第九章 安全管理 **65**
9.1 概览 65
9.2 使用说明 66
9.2.1 初始化 66
9.2.2 使用私有随机地址 68
9.2.3 SM 事件回调 68
9.2.4 每个连接的个性化设置 71

第十章 杂项 **73**
10.1 接收 CTE 73
10.1.1 基于连接的 CTE 接收和发送 73
10.1.2 基于周期广播的 CTE 接收和发送 76
10.1.3 基于私有方式 #1 的 CTE 接收和发送 77
10.1.4 基于私有方式 #2 的 CTE 接收和发送 77
10.2 兼容性 78
10.2.1 Data Length 与 MTU 78
10.3 API 返回值 79

10.4 键值存储 79
10.5 设备数据库 81
10.6 同步版 API 82
10.6.1 GAP 同步 API 85
10.6.2 GATT 客户端同步 API 88
10.7 线程安全的 API 89

第十一章 协议栈能力 **91**

表格

1.1 缩略语 1
1.2 术语 2
1.3 新旧术语对照 2
2.1 Controller 错误码 11
2.2 BLE 链路层特性定义 12
2.3 蓝牙链路层协议版本号 13
3.1 传统广播类型 18
5.1 PHY 比特组合 34
6.1 特征的属性 43
11.1 {*typical*, *extension*,*exp*} 软件包协议栈能力 91
11.2 {*mass_conn*} 软件包协议栈能力 91

插图

2.1 Host 架构 4
2.2 广播及扫描响应数据包格式 6
3.1 用广播数据编辑器生成初始数据 22
4.1 扫描间隔与扫描窗口 25
5.1 减速比为 8 时的行为 36
5.2 减速下出现数据通信 37
5.3 路径损耗上报 38
9.1 BLE 密钥层次结构 65

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
0.61	修正拼写错误等	2022-09-28
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

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）
HCI	Host Controller 接口
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）
RSSI	接收信号强度指示（Received Signal Strength Indicator）
SCA	睡眠时钟精度（Sleep Clock Accuracy）
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]
2. Controller API Reference
3. Application Note: Direction Finding Solution[2]

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

[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);
```

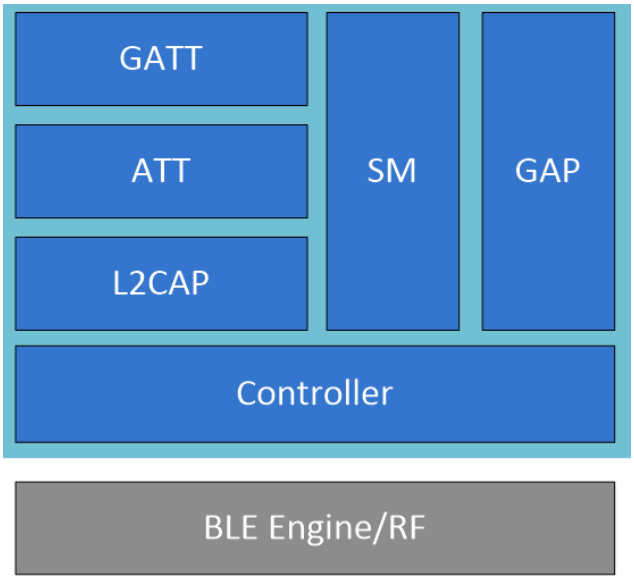

图 2.1: Host 架构

```
    }
}
```

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`）。

`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，本文所说的 App 皆指运行在芯片上的蓝牙程序。

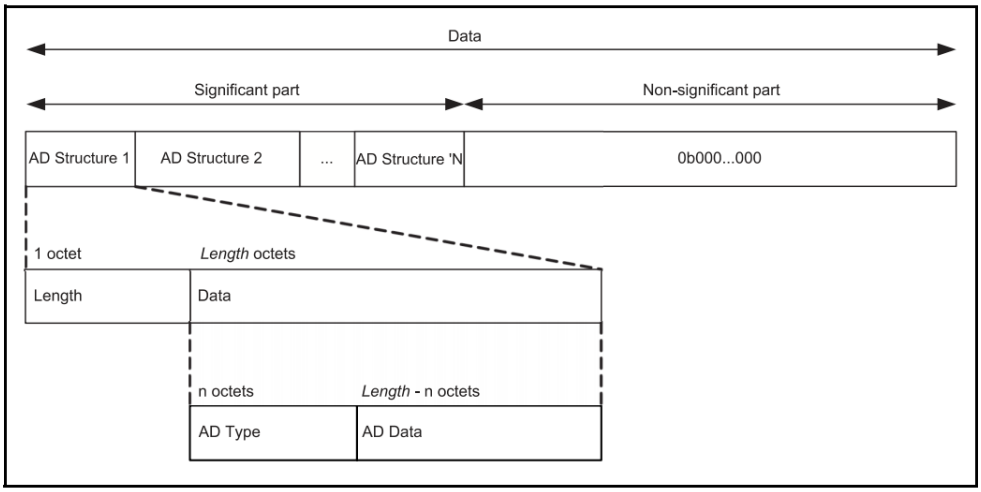

图 2.2: 广播及扫描响应数据包格式

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

  改用 HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE

- 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

  连接建立（le_meta_event_enh_create_conn_complete_t）

- 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）

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

- 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

  周期广播同步建立（le_meta_event_periodic_adv_sync_established_t）

- HCI_SUBEVENT_LE_PERIODIC_ADVERTISING_REPORT

  周期广播报告（le_meta_event_periodic_adv_report_t）

- 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

  周期广播转移请求（le_meta_prd_adv_sync_transfer_recv_t）

- 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_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
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

错误码	含义
0x3E	连接无法建立
0x43	到达极限
0xFE	任务调度失败（私有错误码）
0xFF	其它错误（私有错误码）

2.7 Controller 特性定义

表 2.2: 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)

比特位置	链路层特性
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
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。

2.8 蓝牙规范版本编号

表 2.3: 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

编号	蓝牙链路层协议版本号
9	5.0
10	5.1
11	5.2
12	5.3

2.9 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

2.10 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[5]，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[6]，或者重新封装 Host API，使其变为同步操作（参考 “同步版 API” 一节）。

2.11 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：

[5]更准确地说，只是把一条 HCI 消息放入消息队列。

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

- `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_msg` 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。

2.12 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[7]，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）

[7]地址改变后，曾与之配对的设备无法自动重连。

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[8]

使用 `reverse_bd_addr` 可以翻转地址的字节顺序：

```
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);
```

[8]参照蓝牙核心规范。

第三章　GAP - 广播

3.1　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 $1650B$。

3.1.1　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 3.1。

表 3.1: 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

3.1.2 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;
```

3.1.3 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。

3.1.4 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。

3.1.5 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：

  - 通过 GAP API 停止广播；
  - 连接建立；
  - 已达到预定的广播时长、次数，自动停止；
  - 极端情况：Controller 无法完成任务处理。

- `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。

3.2 使用说明

3.2.1 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,
```

[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。

```
    // 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 3.1 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。

3.2.2 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 3.1）：

```
// 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。

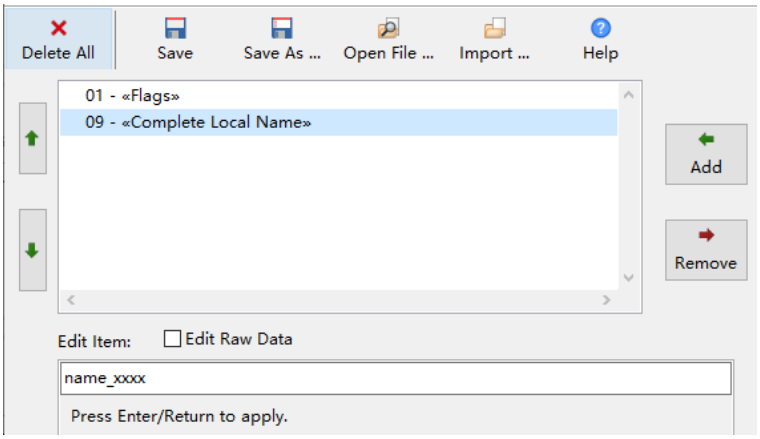

图 3.1: 用广播数据编辑器生成初始数据

```
// 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]);
```

3.2.3 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。

3.2.4 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。

3.2.5 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]。

3.2.6 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]关闭之后，其它设备无法再与该周期广播建立同步。

第四章　GAP - 扫描

4.1　概览

接收广播的过程称为扫描。

4.1.1　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 4.1 所示。

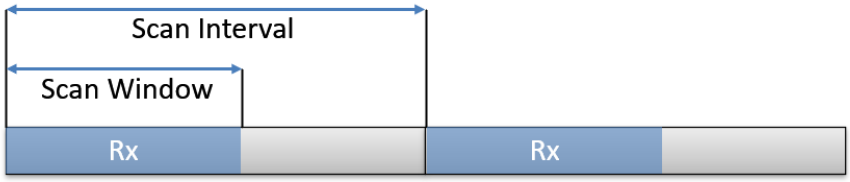

图 4.1: 扫描间隔与扫描窗口

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。

4.1.2　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;
```

4.1.3 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。

4.1.4 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。

4.2 使用说明

4.2.1 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);
```

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;
```

4.2.2 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。

4.2.3 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;
```

4.2.4 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[1]：

```
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,
```

[1]即周期性地接收周期广播。

```
  // 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。

周期广播的接收可参考 SDK *Periodic Scanner*。

第五章　GAP - 连接

5.1　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。

5.2　使用说明

5.2.1　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);
```

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。

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。有时，Controller 会因为无法调度任务而上报 `STATUS=0x07`。对于这种情况，建议 App 延后一段时间再重新尝试建立连接。

5.2.2 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）。

5.2.3 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*。

5.2.4 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*。

5.2.5 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。

[2]参考《Controller API Reference》。

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 5.1: 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。

5.2.6 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：

[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。

```
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
  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。

5.2.7 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。

```
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,
```

[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。

```
  // 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 5.1。

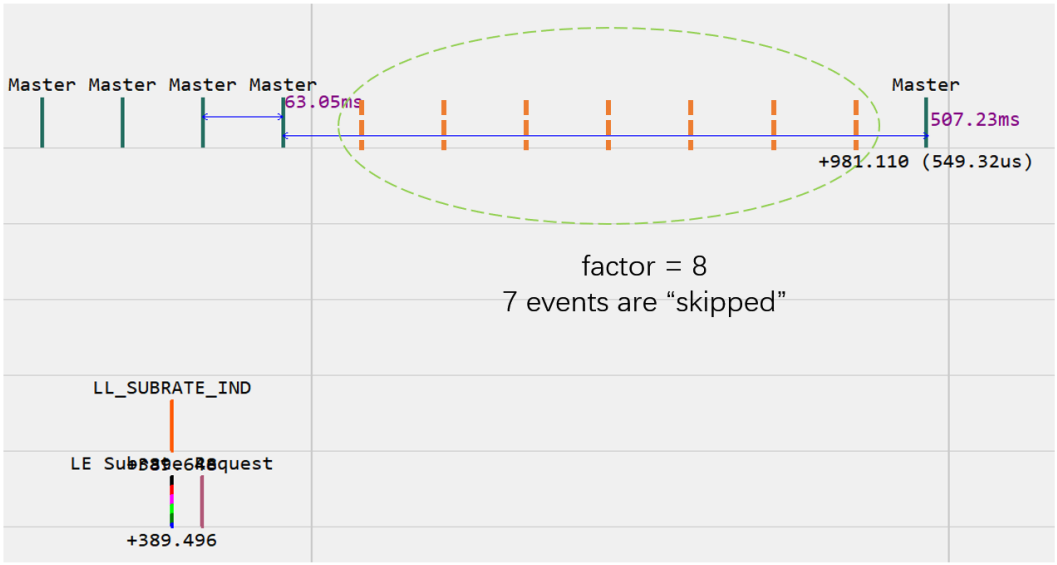

图 5.1: 减速比为 8 时的行为

`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，可以迅速从减速模块回到连接通信模式，**兼顾功耗与效率**，如图 5.2。

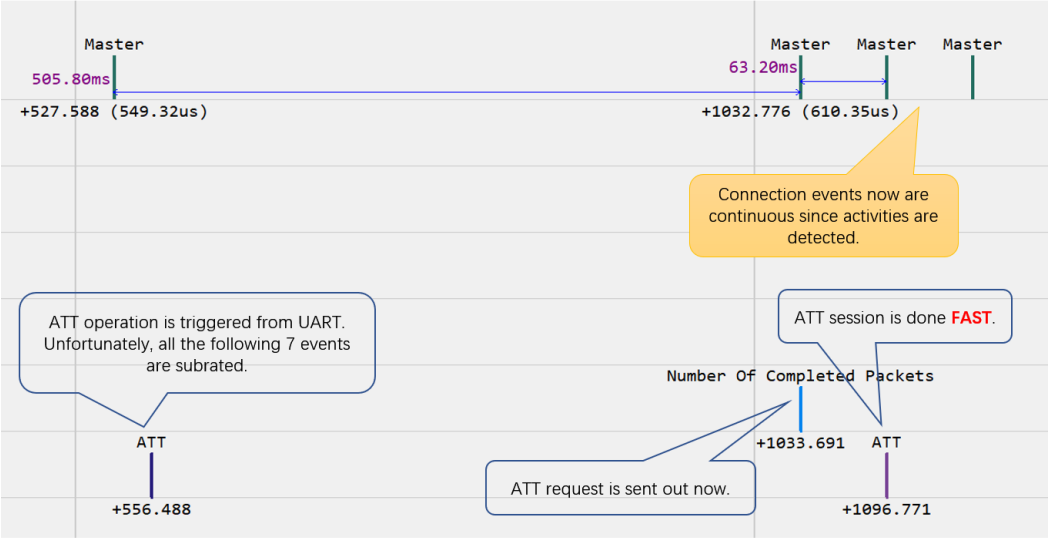

图 5.2: 减速下出现数据通信

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

5.2.8 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 5.3 中的虚线箭头），上报 HCI_SUBEVENT_LE_PATH_LOSS_THRESHOLD 事件。

- 配置路损分类参数：

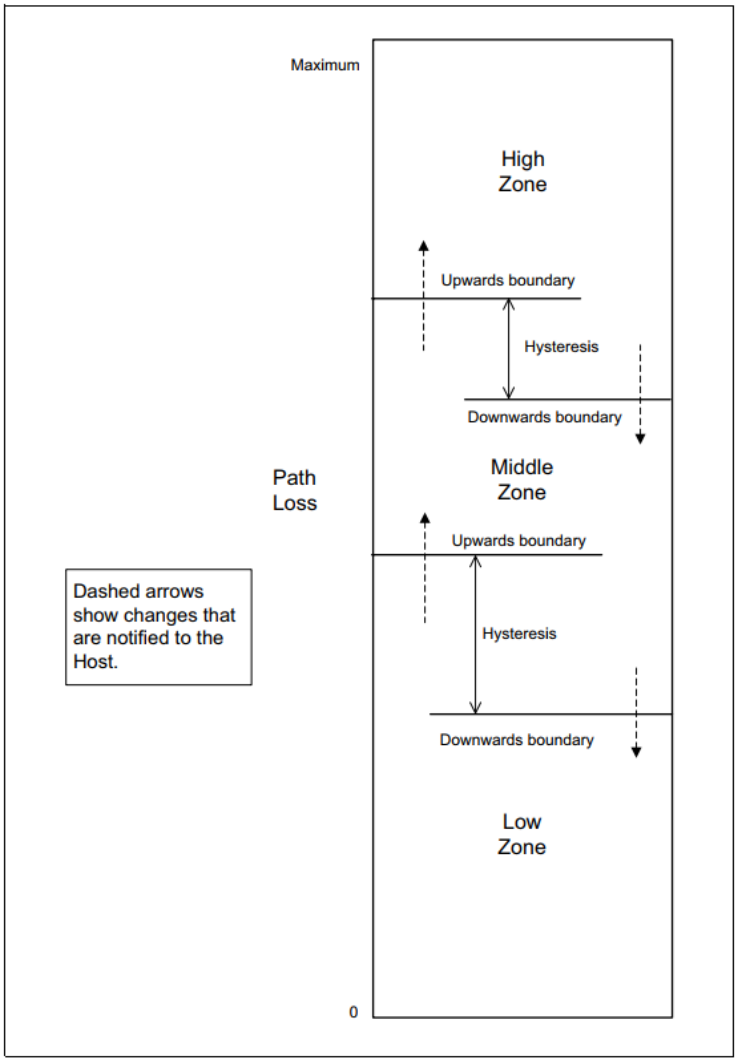

图 5.3: 路径损耗上报

```
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 5.3 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
);
```

5.2.9 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
);
```

第六章　GATT - 服务器

6.1　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，开放者需要做三件事[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：提供写回调函数

```
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 6.1。

表 6.1: 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。

6.2 使用说明

6.2.1 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

[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。

上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 $N+1$。

6.2.2 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。

6.2.3 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]。

6.2.4 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);
```

6.2.5 发送指示（Indication）

通过 att_server_indicate 发送指示：

```
int att_server_indicate(
  // 连接句柄
```

[5]仅适用于有响应的写入，无响应的写入无效。

```
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。

在收到 ATT_EVENT_HANDLE_VALUE_INDICATION_COMPLETE 事件前，不能发送新的指示，否则 att_server_indicate 将返回错误码 ATT_HANDLE_VALUE_INDICATION_IN_PORGRESS。

6.2.6 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。

- 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 表示超时。

6.2.7 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。

第七章　GATT - 客户端

7.1　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`：内存不足，无法创建 GATT 客户端实例
- `GATT_CLIENT_IN_WRONG_STATE`：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`：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。

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);
```

这个回调函数目前只会收到一个事件[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。

7.1.1 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”。

7.2 使用说明

7.2.1 创建客户端

调用 GATT 客户端的绝大多数 API 时，都会按需自动创建客户端实例。不会触发创建动作的 API：

- `gatt_client_listen_for_characteristic_value_updates`

7.2.2 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。

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

- 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;
    }
}
```

7.2.3 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;
    }
}
```

7.2.4 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;
    }
}
```

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);
```

7.2.5 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);
```

7.2.6 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_w
写入值时，如果超过最大长度，会返回错误码：GATT_CLIENT_VALUE_TOO_LONG。

第八章　L2CAP

8.1　概览

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

8.2　使用说明

目前，只有一种情况可能用到 L2CAP API。

8.2.1　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：

```
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。

第九章　安全管理

9.1　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 9.1，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]。

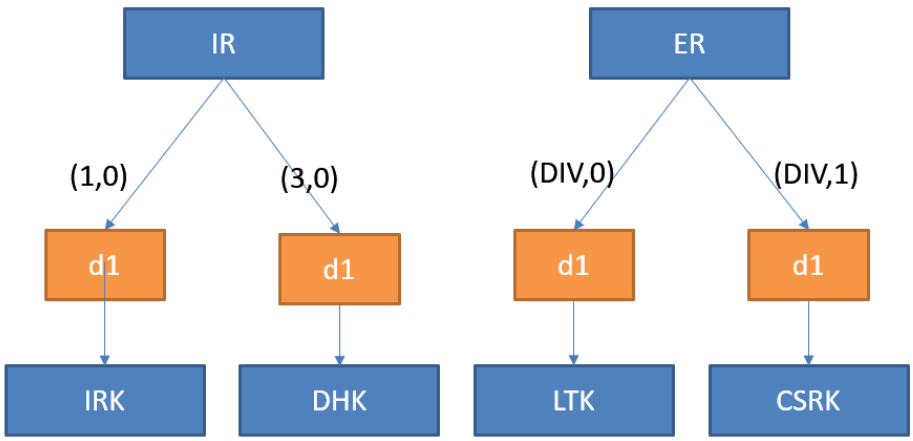

图 9.1: BLE 密钥层次结构

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。

9.2 使用说明

9.2.1 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
#define SM_AUTHREQ_MITM_PROTECTION  ...
```

[3]以及给予用户必要的提示的能力。

9.2.2 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_generatio 修改[4]:

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

9.2.3 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:

[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。

[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。

```
// 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：

  - `sm_event_identity_resolving_succeeded_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_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

与配对有关的 6 个事件：

- `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 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
  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
};
```

9.2.4 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
  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。

第十章　杂项

10.1　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。

10.1.1　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*。

10.1.1.1　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);
```

10.1.1.2 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。

10.1.2 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*。

10.1.2.1 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);
```

10.1.2.2 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);
```

10.1.3 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*。

10.1.4 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*。

10.1.4.1 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。

10.1.4.2 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。

10.2 兼容性

10.2.1 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[4]。为了更新地兼容不同的芯片，协议栈定义了两个配置项：

```
enum btstack_config_item {
    STACK_ATT_SERVER_ENABLE_AUTO_DATA_LEN_REQ = 1,
    STACK_GATT_CLIENT_DISABLE_AUTO_DATA_LEN_REQ = 2,
    //...
};
```

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。

通过 `btstack_config` 配置：

[2]参考《Controller API Reference》。

[3]参考《Controller API Reference》。

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

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

10.3 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`：连接不存在（连接句柄参数错误?）
  - `BTSTACK_ACL_BUFFERS_FULL`：内存已满
- `gap_set_ext_adv_para` 的返回值：
  - 0：成功
  - `BTSTACK_MEMORY_ALLOC_FAILED`：内存已满

10.4 键值存储

协议包含了一个简单的键值存储模块（*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 里使用这个模块，*key* 的取值范围应该在 `KV_USER_KEY_START` 和 `KV_USER_KEY_END` 之间。需要注意以下几点：

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。

这个模块本身没有数据持久化。持久化需要开发者通过 `kv_init`[5] 提供回调来实现：

[5]只能在 `app_main` 就调用。

```
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。整个储存的总大小为 DB_CAPACITY_SIZE。

这个存储模块的增、删、改、查等接口如下。

- 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)
```

- 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。

10.5 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。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);
```

- 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);
    // ...
}
```

10.6 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[6]。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;
```

[6]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。

```
    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));
    }
    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。

10.6.1 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[7] (未知的连接句柄)。

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：

[7] 由规范规定。

```
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 2.3 所示；厂家编号由厂家申请、Bluethooth SIG 指定[8]；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 2.2。

10.6.2 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[9]
- `gatt_client_sync_write_characteristic_descriptor`：同步写入特征描述符

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

[9] 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。

10.7 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_` 前缀。

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：

```
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[10]。

[10]可以使用 `btstack_push_user_runnable`。

第十一章　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 11.1 和表 11.2。

表 11.1: {*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	4	4	8	✓	247

表 11.2: {*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	4	12	12	✓	247

[1]依据 SDK v8.2.0。ING916XX 协议栈能力可能发生变化。