

RTL8762C BLE Stack User Manual

V 1.0.3

2019/3/6

修订历史 (Revision History)

日期	版本	修改	作者	Reviewer
2018/07/13	V1.0.1	Formal version	Berni	Jane
2018/09/12	V1.0.2	Formal version	Berni	Jane
2019/02/25	V1.0.3	Add BLE Privacy Chapter Add BLE Peripheral Privacy Application Add Request Pairing Chapter Add Supported BT Features Chapter Delete DTM Chapter	Berni	

目录

修订历史 (Revision History)	2
目录	3
表目录	7
图目录	9
词汇表	12
1 概述	13
1.1 支持的 BT Features	14
1.2 BLE Profile 架构	15
1.2.1 GAP	15
1.2.2 基于 GATT 的 Profile	16
2 GAP	17
2.1 GAP 结构概述	17
2.1.1 GAP 的位置	17
2.1.2 GAP 的功能	18
2.1.3 GAP 层设备状态	19
2.1.4 GAP 消息	24
2.1.5 GAP Lib	26
2.1.6 APP 消息流	27
2.2 GAP 的初始化和启动流程	29
2.2.1 GAP 参数的初始化	29
2.2.2 GAP 启动流程	35
2.3 BLE GAP 消息	37
2.3.1 概述	37
2.3.2 Device 状态消息	39
2.3.3 Connection 相关消息	41
2.3.4 Authentication 相关消息	42
2.3.5 Extended Advertising 状态消息	46

2.4 BLE GAP 回调函数.....	47
2.4.1 BLE GAP 回调函数消息概述.....	48
2.5 BLE GAP 用例.....	53
2.5.1 Airplane Mode 设置.....	53
2.5.2 本地设备 IRK 的设置.....	54
2.5.3 GAP Service Characteristic 的可写属性	55
2.5.4 本地设备使用 Static Random Address.....	57
2.5.5 Physical (PHY) 设置	59
2.5.6 蓝牙协议栈相关特性设置	62
2.5.7 请求配对	64
2.6 GAP 信息存储	66
2.6.1 FTL 简介	66
2.6.2 本地协议栈信息存储	68
2.6.3 绑定信息存储	69
2.7 BLE Privacy	76
2.7.1 Specification 介绍	76
2.7.2 Privacy 管理模块	79
3 GATT Profile.....	85
3.1 BLE Profile Server	85
3.1.1 概述	85
3.1.2 支持的 Profile 和 Service	86
3.1.3 Profile Server 交互.....	88
3.1.4 Specific Service 的实现	103
3.2 BLE Profile Client.....	111
3.2.1 概述	111
3.2.2 支持的 Clients.....	112
3.2.3 Profile Client Layer	112
3.3 GATT Profile 用例.....	124
3.3.1 ANCS Client.....	124
4 BLE 示例工程.....	126

4.1 BLE Broadcaster Application.....	127
4.1.1 简介	127
4.1.2 工程概述	127
4.1.3 源代码概述	128
4.1.4 APP 的可配置功能	130
4.1.5 测试步骤	130
4.2 BLE Observer Application.....	130
4.2.1 简介	130
4.2.2 工程概述	130
4.2.3 源代码概述	131
4.2.4 APP 的可配置功能	133
4.2.5 测试步骤	133
4.3 BLE Peripheral Application	134
4.3.1 简介	134
4.3.2 工程概述	135
4.3.3 源代码概述	136
4.3.4 APP 的可配置功能	139
4.3.5 测试步骤	140
4.4 BLE Central Application	141
4.4.1 简介	141
4.4.2 工程概述	141
4.4.3 源代码概述	142
4.4.4 APP 的可配置功能	148
4.4.5 用户命令	149
4.4.6 测试步骤	149
4.5 BLE Scatternet Application.....	155
4.5.1 简介	155
4.5.2 工程概述	156
4.5.3 源代码概述	157
4.5.4 APP 的可配置功能	162

4.5.5 用户命令	163
4.6 BLE BT5 Peripheral Application	163
4.6.1 简介	163
4.6.2 工程概述	164
4.6.3 源代码概述	165
4.6.4 APP 的可配置功能	173
4.6.5 测试步骤	173
4.7 BLE BT5 Central Application	174
4.7.1 简介	174
4.7.2 工程概述	175
4.7.3 源代码概述	176
4.7.4 APP 的可配置功能	179
4.7.5 用户命令	182
4.7.6 测试步骤	187
4.8 BLE Application 用户命令	194
4.8.1 用户命令的实现	194
4.8.2 Data UART 连接	196
4.8.3 用户命令	197
4.9 BLE Peripheral Privacy Application	198
4.9.1 简介	198
4.9.2 工程概述	198
4.9.3 Privacy 使用流程图	199
4.9.4 APP 的可配置功能	201
4.9.5 用户命令	202
4.9.6 测试步骤	202
参考文献	204

表目录

表 1-1 支持的 BT Features	14
表 2-1 Advertising 参数设置	32
表 2-2 Authentication 相关消息	42
表 2-3 gap_le.h 相关消息	48
表 2-4 gap_conn_le.h 相关消息.....	48
表 2-5 gap_bond_le.h 相关消息	50
表 2-6 gap_scan.h 相关消息	51
表 2-7 gap_adv.h 相关消息.....	51
表 2-8 gap_dtm.h 相关消息.....	51
表 2-9 gap_vendor.h 相关消息	52
表 2-10 gap_ext_scan.h 相关消息	52
表 2-11 gap_ext_adv.h 相关消息	52
表 3-1 支持的 Profile 列表	86
表 3-2 支持的 service 列表.....	87
表 3-3 Flags 的可选值和描述	104
表 3-4 Flags Value 的选择模式	105
表 3-5 Permissions 的可用值.....	105
表 3-6 Service Table 示例	106
表 3-7 支持的 Clients.....	112
表 3-8 Discovery 状态.....	115
表 3-9 Discovery 结果.....	116
表 4-1 Broadcaster 工程文件列表.....	128
表 4-2 Observer 工程文件列表	131
表 4-3 Peripheral 工程文件列表.....	135
表 4-4 Central 工程文件列表	142
表 4-5 Scatternet 工程文件列表	157
表 4-6 使用 LE Advertising Extensions 的 Peripheral 设备的兼容性.....	164
表 4-7 BT5 Peripheral 工程目录结构.....	165
表 4-8 使用 legacy advertising PDUs 的 Extended Advertising 参数设置.....	169
表 4-9 使用 extended advertising PDUs 的 Extended Advertising 参数设置.....	170
表 4-10 使用 LE Advertising Extensions 的 Central 设备的兼容性	174
表 4-11 BT5 Central 工程文件列表	175

表 4-12 用户命令文件	194
表 4-13 Peripheral Privacy 工程文件列表	199

图目录

图 1-1 软件架构	13
图 1-2 蓝牙 Profile	15
图 1-3 基于 GATT 的 Profile 层级结构	16
图 2-1 GAP 头文件	17
图 2-2 GAP 在 SDK 中的位置	18
图 2-3 Advertising 状态的状态转换	19
图 2-4 Scan 状态的状态转换	20
图 2-5 使用 Extended Scan 时 Scan 状态的状态转换	20
图 2-6 主动的 Connection 状态转换	22
图 2-7 被动的 Connection 状态转换	22
图 2-8 针对一个 advertising set 的 Extended Advertising 状态的转换	23
图 2-9 GAP Lib	26
图 2-10 GAP BT5 Lib	27
图 2-11 APP 消息流	28
图 2-12 GAP 内部初始化流程	36
图 2-13 Config file 中的 LE Link 数目	62
图 2-14 ATT Insufficient Authentication	65
图 2-15 FTL 布局	66
图 2-16 增加一个绑定设备	69
图 2-17 移除一个绑定设备	69
图 2-18 清除所有绑定设备	70
图 2-19 将一个绑定设备设为最高优先级	70
图 2-20 获取最高优先级设备	70
图 2-21 获取最低优先级设备	70
图 2-22 优先级管理示例	71
图 2-23 LE FTL 布局	71
图 2-24 static address 的格式	76
图 2-25 non-resolvable private address 的格式	77
图 2-26 resolvable private address 的格式	77
图 2-27 Transport Specific Key Distribution	78
图 2-28 Resolving List 和 Device White List 的逻辑表示	79
图 2-29 在工程中增加 privacy 管理模块	80

图 3-1 GATT Profile 头文件	85
图 3-2 Profile Server 层级	86
图 3-3 向 Server 添加 Services	89
图 3-4 注册 Service 的流程	89
图 3-5 读 Characteristic Value – 由 Attribute Element 提供 Attribute Value	91
图 3-6 读 Characteristic Value – 由 APP 提供 Attribute Value 且结果未挂起	92
图 3-7 读 Characteristic Value – 由 APP 提供 Attribute Value 且结果挂起	93
图 3-8 Write Characteristic Value – 由 Attribute Element 提供 Attribute Value	95
图 3-9 Write Characteristic Value – 由 APP 提供 Attribute Value 且结果未挂起	95
图 3-10 Write Characteristic Value – 由 APP 提供 Attribute Value 且结果挂起	96
图 3-11 Write Characteristic Value – 写 CCCD 值	97
图 3-12 Write without Response / Signed Write without Response – 由 APP 提供 Attribute Value	100
图 3-13 Write Long Characteristic Value – Prepare Write 流程	100
图 3-14 Write Long Characteristic Values – 结果未挂起的 Execute Write	101
图 3-15 Write Long Characteristic Values – 结果挂起的 Execute Write	101
图 3-16 Characteristic Value Notification	101
图 3-17 Characteristic Value Indication	102
图 3-18 Profile Client 层级	112
图 3-19 向 Profile Client Layer 添加 Specific Clients	114
图 3-20 GATT Discovery 流程	115
图 3-21 Read Characteristic Value by Handle 流程	117
图 3-22 Read Characteristic Value by UUID 流程	118
图 3-23 Write Characteristic Value 流程	118
图 3-24 Write Long Characteristic Value 流程	119
图 3-25 Write Without Response 流程	119
图 3-26 Signed Write without Response 流程	120
图 3-27 Characteristic Value Notification 流程	120
图 3-28 结果未挂起的 Characteristic Value Indication 流程	122
图 3-29 结果挂起的 Characteristic Value Indication 流程	122
图 4-1 Broadcaster 工程目录结构	128
图 4-2 Observer 工程目录结构	131
图 4-3 Peripheral 工程目录结构	135
图 4-4 与 iOS 设备测试	140
图 4-5 Central 工程目录结构	142
图 4-6 app_discov_services()流程图	147

图 4-7 app_discov_services()流程图(F_BT_GATT_SRV_HANDLE_STORAGE)	149
图 4-8 Scatternet 工程目录结构	156
图 4-9 BT5 Peripheral 工程目录结构	165
图 4-10 启动 Extended Advertising 的流程图	166
图 4-11 BT5 Central 工程目录结构	175
图 4-12 重组流程图(GAP_EXT_ADV_EVT_DATA_STATUS_COMPLETE)	181
图 4-13 重组流程图(GAP_EXT_ADV_EVT_DATA_STATUS_MORE)	182
图 4-14 PC 和 EVB 之间的 Data UART 连接	196
图 4-15 串口设置	196
图 4-16 Peripheral Privacy 工程目录结构	199
图 4-17 Peripheral Privacy APP 流程图	200
图 4-18 与 iOS 设备测试	203

词汇表

缩写	含义
ATT	Attribute protocol
BLE	Bluetooth Low Energy
DTM	Direct Test Mode
GAP	Generic Access Profile
GATT	Generic Attribute Profile
L2CAP	Logical Link Control and Adaptation protocol
SDK	Software Development Kit
SMP	Security Manager protocol
SOC	System on Chip

1 概述

Realtek SDK (Software Development Kit, SDK)提供的资料包括 Stack/Profiles 介绍文档、示例 profile 和用户示例程序，旨在帮助用户使用 Realtek SOC (System on Chip, SOC) 设备进行产品开发。SDK 可以促进 BLE (Bluetooth Low Energy, BLE) 应用的快速开发。Profile 作为 SDK 组成模块，封装 BLE 协议栈的实现细节，为应用开发提供用户友好和便于使用的接口。

本文综述 BLE 协议栈接口，其中包括基于 GAP (Generic Access Profile, GAP) 的接口和基于 GATT (Generic Attribute Profile, GATT) 的接口。

SDK 中 BLE 协议栈和 Profile 的架构如图 1-1 所示。

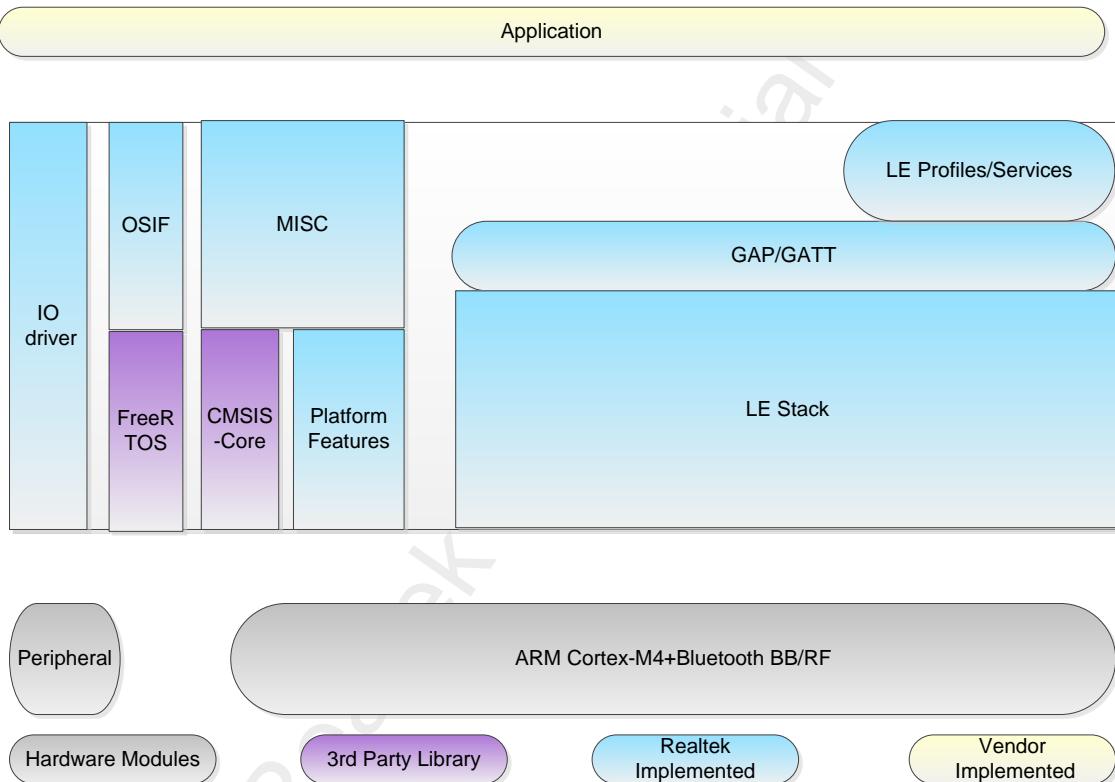


图 1-1 软件架构

1.1 支持的 BT Features

表 1-1 支持的 BT Features

Spec Version	BT Feature	RTL8762C	Remark
BT4.0	Advertiser	Y	
	Scanner	Y	
	Initiator	Y	
	Master	Y	
	Slave	Y	
BT4.1	Low Duty Cycle Directed Advertising	Y	
	LE L2CAP Connection Oriented Channel	Y	
	LE Scatternet	Y	
	LE Ping	Y	
BT4.2	LE Data Packet Length Extension	Y	
	LE Secure Connections	Y	
	Link Layer Privacy (Privacy1.2)	Y	
	Link Layer Extended Filter Policies	Y	
BT5	2 Msym/s PHY for LE	Y	
	LE Long Range	Y	
	High Duty Cycle Non-Connectable Advertising	Y	
	LE Advertising Extensions	Y	
	LE Channel Selection Algorithm #2	Y	

RTL8752C 不支持 BT5 中的特性。

关于 LE Link 数目的内容参见 [LE Link 数目的配置](#)。

BT5 中 LE Advertising Extensions 的兼容性参见 [BLE BT5 Peripheral Application](#) 和 [BLE BT5 Central Application](#)。

1.2 BLE Profile 架构

在蓝牙核心规范 (Bluetooth Core Specification) 中，Profile 的定义不同于 Protocol 的定义。Protocol 被定义为各层协议，例如 Link Layer、Logical Link Control and Adaptation protocol (L2CAP)、Security Manager protocol (SMP) 和 Attribute protocol (ATT)。不同于 Protocol，Profile 从使用蓝牙核心规范中各层协议的角度，定义蓝牙应用互操作性的实现。Profile 定义 Protocol 中的可用特性和功能，以及蓝牙设备互操作性的实现，使蓝牙协议栈适用于各种场景的应用开发。

在蓝牙核心规范中，Profile 和 Protocol 的关联如图 1-2 所示。

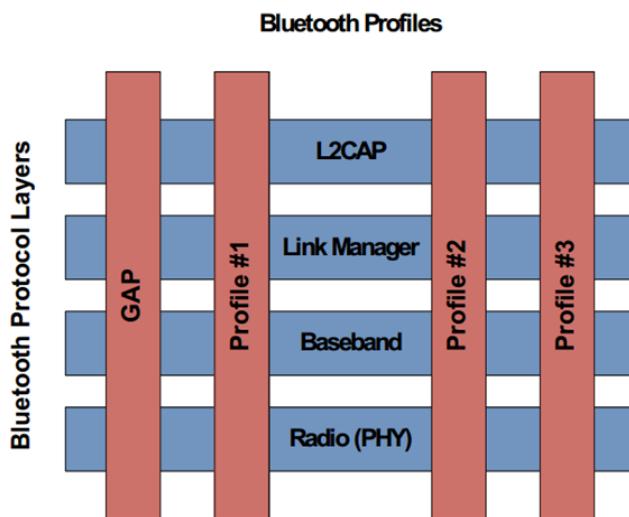


图 1-2 蓝牙 Profile

如图 1-2 所示，Profile 由红色矩形框表示，包括 GAP、Profile #1、Profile #2 和 Profile #3。蓝牙核心规范中的 Profile 分为两种类型：GAP，图中红色矩形框所示的 GAP；基于 GATT 的 Profile，图中红色矩形框所示的 Profile #1、Profile #2 和 Profile #3。

1.2.1 GAP

GAP 是所有的蓝牙设备均需实现的 Profile，用于描述 device discovery、connection、security requirement 和 authentication 的行为和方法。GAP 中的 BLE 部分定义四种角色 (Broadcaster、Observer、Peripheral 和 Central)，用于优化各种应用场景。

Broadcaster 用于只通过广播发送数据的应用；Observer 用于接收广播数据的应用；Peripheral 用于通过广播发送数据并且可以建立链路的应用；Central 用于接收广播数据并且建立一条或多条链路的应用。

1.2.2 基于 GATT 的 Profile

在蓝牙核心规范中，另一种常用的 Profile 是基于 GATT 的 Profile。GATT 分为 server 和 client 两种角色。Server 用于提供 service 数据。Client 可以访问 service 数据。基于 GATT 的 Profile 是基于 server-client 交互结构，适用于不同应用场景，用于蓝牙设备之间的特定数据交互。如图 1-3 所示，Profile 是以 Service 和 Characteristic 的形式组成的。

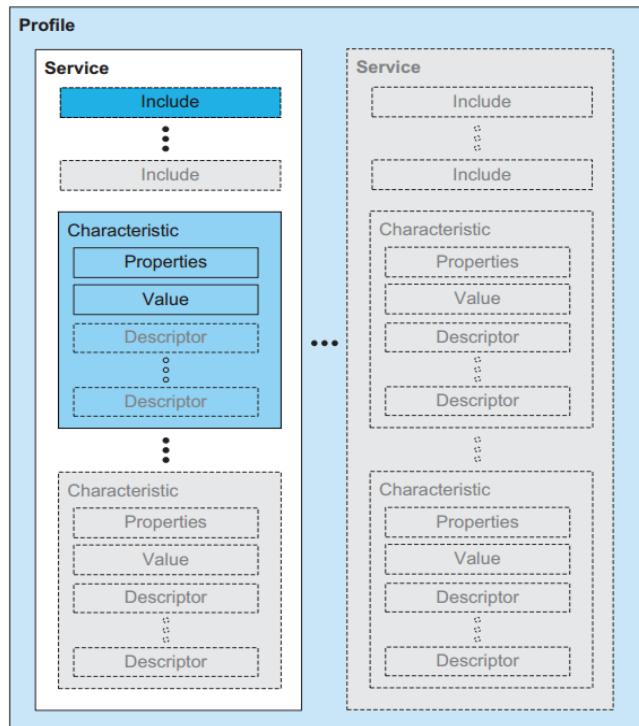


图 1-3 基于 GATT 的 Profile 层级结构

2 GAP

GAP 是所有蓝牙设备均需实现的 Profile，用于描述 device discovery、connection、security requirement 和 authentication 的行为和方法。

GAP 层是在 ROM 中实现的，提供接口给 application 使用。在 SDK 中提供头文件，头文件目录为 sdk\inc\bluetooth\gap。

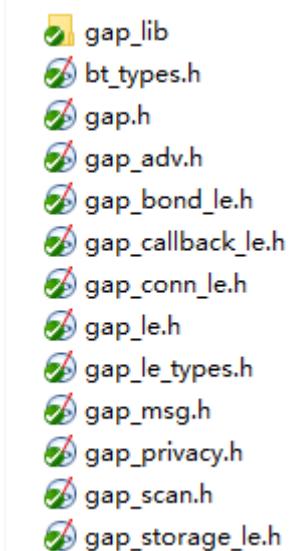


图 2-1 GAP 头文件

GAP 层的内容分为以下几个部分进行介绍：

1. 在章节 [GAP 结构概述](#) 中介绍 GAP 结构。
2. 在章节 [GAP 的初始化和启动流程](#) 中介绍 GAP 参数的配置和 GAP 层内部启动流程。
3. 在章节 [BLE GAP 消息](#) 中介绍 GAP 消息类型的定义和 GAP Message 处理流程。
4. GAP 层使用 GAP 消息回调函数发送消息给 application，在章节 [BLE GAP 回调函数](#) 中介绍关于 GAP 回调函数的内容。
5. 在章节 [BLE GAP 用例](#) 中介绍 GAP 接口的应用示例。
6. 在章节 [GAP 信息存储](#) 中介绍由 GAP 实现的本地信息和设备绑定信息的存储。

2.1 GAP 结构概述

2.1.1 GAP 的位置

GAP 层作为蓝牙协议层的组成模块，如图 2-2 所示，虚线框内的部分为蓝牙协议层。Application 在蓝牙协议层之上，baseband/RF 位于蓝牙协议层之下。GAP 层给 application 提供访问 Upper Stack 的接口。

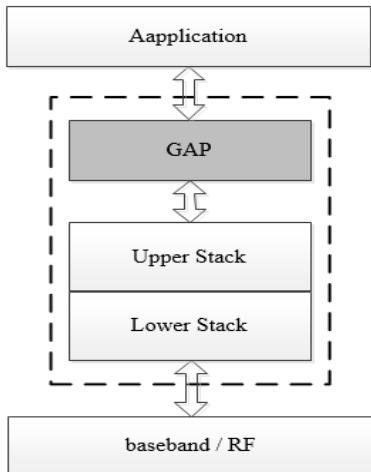


图 2-2 GAP 在 SDK 中的位置

2.1.2 GAP 的功能

GAP 层提供的接口的功能如下所示：

1. Advertising

设置/获取 advertising 参数，启动/停止 advertising。

2. Scan

设置/获取 scan 参数，启动/停止 scan。

3. Connection

设置 connection 参数，创建 connection，终止已建立的 connection，更新 connection 参数。

4. 配对

设置配对参数，启动配对，使用 passkey entry 方式时输入/显示 passkey，删除绑定设备密钥。

5. 密钥管理

根据设备地址和地址类型查找 key entry，保存/加载绑定信息的密钥，解析 random address。

6. 其它

- 1) 设置 GAP 公共参数，例如 device appearance 和 device name
- 2) 获取支持的最大 BLE 链路数目
- 3) 修改 white list
- 4) 生成/设置本地设备 random address
- 5) 配置本地设备 identity address
- 6) 等等

API 不支持多线程，API 的调用和消息处理必须在同一个 task 中。SDK 中提供的 API 分为同步 API 和异步 API。同步 API 的结果由返回值表示，例如 le_adv_set_param()。若 le_adv_set_param()的返回值为

GAP_CAUSE_SUCCESS，APP 成功设置一个 GAP advertising 参数。异步 API 的结果是通过 GAP 消息通知的，例如 le_adv_start()。若 le_adv_start()的返回值为 GAP_CAUSE_SUCCESS，启动 advertising 的请求发送成功，启动 advertising 的结果是通过 GAP 消 GAP_MSG_LE_DEV_STATE_CHANGE 通知 APP 的。

2.1.3 GAP 层设备状态

GAP 层设备状态由 advertising 状态、scan 状态和 connection 状态组成。若使能 LE Advertising Extensions，将使用 extended advertising 状态来替代 advertising 状态。每一个状态都有相应的子状态，本节内容将介绍各子状态。

2.1.3.1 Advertising 状态

Advertising 状态有四个子状态，idle 状态、start 状态、advertising 状态和 stop 状态，在 gap_msg.h 中定义 Advertising 状态的子状态。

```
/* GAP Advertising State */
#define GAP_ADV_STATE_IDLE          0 // Idle, no advertising
#define GAP_ADV_STATE_START         1 // Start Advertising. A temporary state, haven't received the result.
#define GAP_ADV_STATE_ADVERTISING   2 // Advertising
#define GAP_ADV_STATE_STOP          3 // Stop Advertising. A temporary state, haven't received the result.
```

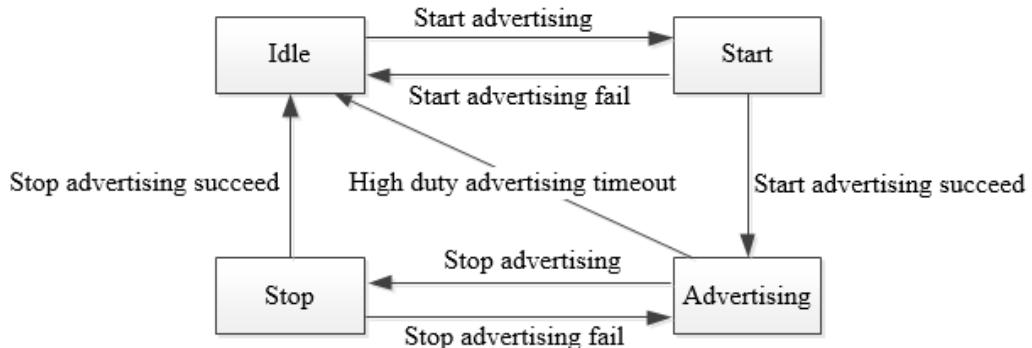


图 2-3 Advertising 状态的状态转换

1. idle 状态

默认状态，不发送 advertisement。

2. start 状态

在 idle 状态启动 advertising 之后，启动 advertising 的流程尚未完成。start 状态为临时状态，若成功启动 advertising，则 Advertising 状态进入 advertising 状态；若启动 advertising 失败，则 Advertising 状态回到 idle 状态。

3. advertising 状态

成功启动 advertising。在此状态下，设备发送 advertisement。若 advertising 类型是 high duty cycle directed

advertising, 一旦超时, Advertising 状态进入 idle 状态。

4. stop 状态

在 advertising 状态停止 advertising 之后, 停止 advertising 的流程尚未完成。stop 状态为临时状态, 若成功停止 advertising, 则 Advertising 状态进入 idle 状态; 若停止 advertising 失败, 则 Advertising 状态回到 advertising 状态。

2.1.3.2 Scan 状态

Scan 状态有四个子状态, idle 状态、start 状态、scanning 状态和 stop 状态, 在 gap_msg.h 中定义 Scan 状态的子状态。

```
/* GAP Scan State */
#define GAP_SCAN_STATE_IDLE          0 //Idle, no scanning
#define GAP_SCAN_STATE_START         1 //Start scanning. A temporary state, haven't received the result.
#define GAP_SCAN_STATE_SCANNING      2 //Scanning
#define GAP_SCAN_STATE_STOP          3 //Stop scanning, A temporary state, haven't received the result
```

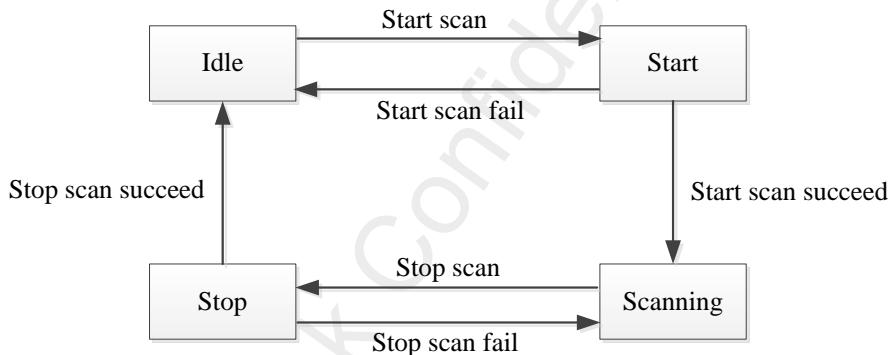


图 2-4 Scan 状态的状态转换

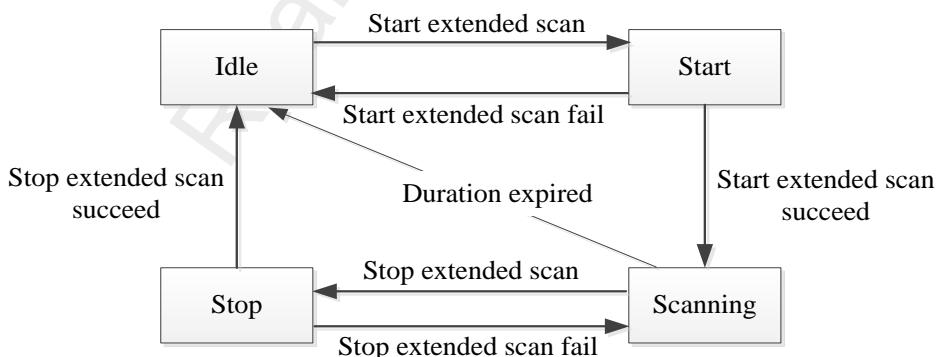


图 2-5 使用 Extended Scan 时 Scan 状态的状态转换

1. idle 状态

默认状态, 不进行 scan。

2. start 状态

在 idle 状态启动 scan 之后，启动 scan 的流程尚未完成。start 状态为临时状态，若成功启动 scan，则 Scan 状态进入 scanning 状态；若启动 scan 失败，则 Scan 状态回到 idle 状态。

3. scanning 状态

成功启动 scan。在此状态下，设备进行 scan，接收 advertisement。使用 Extended Scan 时，若 Duration 参数不为零且 Period 参数为零，一旦 scan 时间达到 Duration，Scan 状态将进入 idle 状态。

4. stop 状态

在 scanning 状态停止 scan 之后，停止 scan 的流程尚未完成。stop 状态为临时状态，若成功停止 scan，则 Scan 状态进入 idle 状态；若停止 scan 失败，则 Scan 状态回到 scanning 状态。

2.1.3.3 Connection 状态

由于支持多链路，当 GAP Connection 状态为 idle 状态时，Link 状态是 connected 或 disconnected。因此，Connection 状态的变化需要结合 GAP Connection 状态和 Link 状态。

GAP Connection 状态的子状态包括 idle 状态和 connecting 状态，在 gap_msg.h 中定义 GAP Connection 状态的子状态。

```
#define GAP_CONN_DEV_STATE_IDLE          0  //!< Idle
#define GAP_CONN_DEV_STATE_INITIATING      1  //!< Initiating Connection
```

注解：当 GAP Connection 状态为 connecting 状态时，application 不能主动创建另一条链路。

Link 状态有四个子状态，disconnected 状态、connecting 状态、connected 状态和 disconnecting 状态，在 gap_msg.h 中定义 Link 状态的子状态。

```
/* Link Connection State */
typedef enum {
    GAP_CONN_STATE_DISCONNECTED, // Disconnected.
    GAP_CONN_STATE_CONNECTING,   // Connecting.
    GAP_CONN_STATE_CONNECTED,    // Connected.
    GAP_CONN_STATE_DISCONNECTING // Disconnecting.
} T_GAP_CONN_STATE;
```

作为 Master 角色主动创建 connection 时的 Connection 状态变化不同于作为 Slave 角色被动接收 connection indication 时的 Connection 状态变化。以下章节分别介绍这两种场景。

2.1.3.3.1 主动的 Connection 状态转换

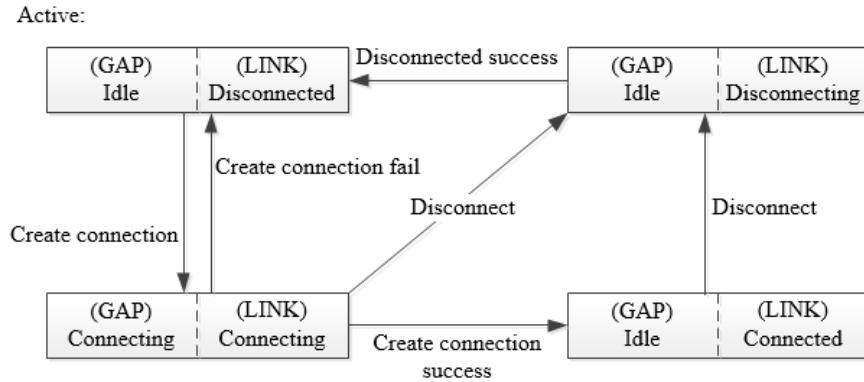


图 2-6 主动的 Connection 状态转换

1. idle 状态 | disconnected 状态

GAP Connection 状态为 idle 状态, Link 状态为 disconnected 状态, 未建立 connection。

2. connecting 状态 | connecting 状态

Master 创建 connection, 创建流程尚未完成。这是临时状态, GAP Connection 状态为 connecting 状态, Link 状态为 connecting 状态。若成功创建 connection, GAP Connection 状态转换为 idle 状态, Link 状态转换为 connected 状态。若创建 connection 失败, GAP Connection 状态回到 idle 状态, Link 状态回到 disconnected 状态。在此状态下, Master 可以断开链路, 此时, GAP Connection 状态回到 idle 状态, Link 状态转换为 disconnecting 状态。

3. idle 状态 | connected 状态

成功创建 connection, GAP Connection 状态为 idle 状态, Link 状态为 connected 状态。

4. idle 状态 | disconnecting 状态

Master 终止 connection, 终止流程尚未完成。这是临时状态, GAP Connection 状态为 idle 状态, Link 状态为 disconnecting 状态。若成功终止 connection, Link 状态转换为 disconnected 状态。

2.1.3.3.2 被动的 Connection 状态转换

Passive:

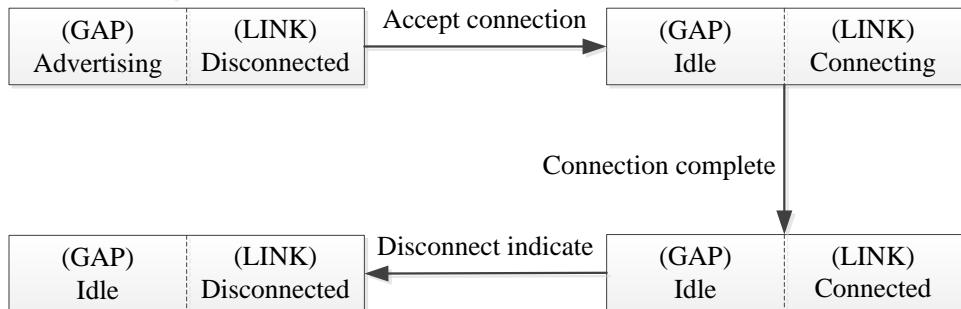


图 2-7 被动的 Connection 状态转换

1. Slave 接受 connection

当 Slave 收到 connect indication 后，GAP Advertising 状态将从 advertising 状态转换为 idle 状态，Link 状态将从 disconnected 状态转换为 connecting 状态。当创建 connection 流程完成之后，Link 状态将进入 connected 状态。

2. 对端设备断开 connection

当对端设备断开 connection 且本地设备收到 disconnect indication 后，本地设备的 Link 状态将从 connected 状态转换为 disconnected 状态。

2.1.3.4 Extended Advertising 状态

不使用 LE Advertising Extensions 时，设备使用 Advertising 状态，可以视为存在一个简化的 advertising set。本节内容仅适用于使用 LE Advertising Extensions 的设备。

由于支持 LE Advertising Extensions，设备可以同时启动或停止多个 advertising set，因此 Extended Advertising 状态的变化需要结合 advertising set。Extended Advertising 状态有四个子状态，idle 状态、start 状态、advertising 状态和 stop 状态，在 gap_ext_adv.h 中定义 Extended Advertising 状态的子状态。

```
/** @brief GAP extended advertising state. */
typedef enum
{
    EXT_ADV_STATE_IDLE,           /**< Idle, no advertising. */
    EXT_ADV_STATE_START,         /**< Start Advertising. A temporary state, haven't received the result. */
    EXT_ADV_STATE_ADVERTISING,   /**< Advertising. */
    EXT_ADV_STATE_STOP,          /**< Stop Advertising. A temporary state, haven't received the result. */
} T_GAP_EXT_ADV_STATE;
```

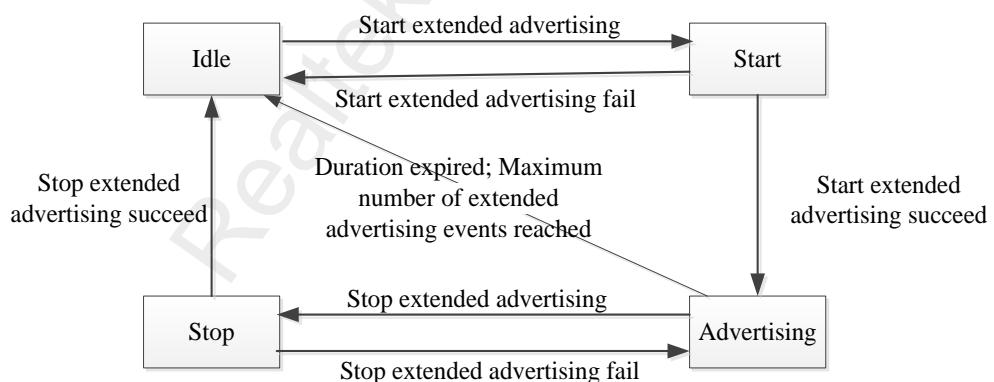


图 2-8 针对一个 advertising set 的 Extended Advertising 状态的转换

1. idle 状态

默认状态，不发送 advertisement。

2. start 状态

在 idle 状态启动 extended advertising 之后，启动 extended advertising 的流程尚未完成。start 状态为临时状态，若成功启动 extended advertising，则 Extended Advertising 状态进入 advertising 状态；若启动 extended

advertising 失败，则 Extended Advertising 状态回到 idle 状态。

3. advertising 状态

成功启动 extended advertising。在此状态下，设备发送 advertisement。若 Duration 参数不为零，一旦 advertising 时间达到 Duration，Extended Advertising 状态将进入 idle 状态。若 Extended Advertising Events 的最大值参数不为零，一旦达到 Extended Advertising Events 的最大值，Extended Advertising 状态将进入 idle 状态。

4. stop 状态

在 advertising 状态停止 extended advertising 之后，停止 extended advertising 的流程尚未完成。stop 状态为临时状态，若成功停止 extended advertising，则 Extended Advertising 状态进入 idle 状态；若停止 extended advertising 失败，则 Extended Advertising 状态回到 advertising 状态。

2.1.4 GAP 消息

GAP 消息包括蓝牙状态消息和 GAP API 消息。蓝牙状态消息用于向 APP 通知蓝牙状态信息，包括 device 状态转换、connection 状态转换以及 bond 状态转换等。GAP API 消息用于向 APP 通知调用 API 后函数的执行状态。每个 API 都有相应的消息，更多关于 GAP 消息的信息参见 [BLE GAP 消息](#) 和 [BLE GAP 回调函数](#)。

2.1.4.1 蓝牙状态消息

在 gap_msg.h 中定义蓝牙状态消息。

```
/* BT status message */

#define GAP_MSG_LE_DEV_STATE_CHANGE          0x01 // Device state change msg type.
#define GAP_MSG_LE_CONN_STATE_CHANGE         0x02 // Connection state change msg type.
#define GAP_MSG_LE_CONN_PARAM_UPDATE        0x03 // Connection parameter update changed msg type.
#define GAP_MSG_LE_CONN_MTU_INFO            0x04 // Connection MTU size info msg type.
#define GAP_MSG_LE_AUTHEN_STATE_CHANGE      0x05 // Authentication state change msg type.
#define GAP_MSG_LE_BOND_PASSKEY_DISPLAY    0x06 // Bond passkey display msg type.
#define GAP_MSG_LE_BOND_PASSKEY_INPUT       0x07 // Bond passkey input msg type.
#define GAP_MSG_LE_BOND_OOB_INPUT          0x08 // Bond passkey oob input msg type.
#define GAP_MSG_LE_BOND_USER_CONFIRMATION   0x09 // Bond user confirmation msg type.
#define GAP_MSG_LE_BOND JUST_WORK           0x0A // Bond user confirmation msg type.

#if F_BT_LE_5_0_AE_ADV_SUPPORT
#define GAP_MSG_LE_EXT_ADV_STATE_CHANGE    0x0B // Extended advertising state change msg type.
#endif
```

2.1.4.2 GAP API 消息

在 gap_callback_le.h 中定义 GAP API 消息，在每个 API 的注释和示例代码中介绍该 API 对应的消息以

及处理方法。

```

/* GAP API message */

#define GAP_MSG_LE MODIFY_WHITE_LIST          0x01 // response msg type for le_modify_white_list
#define GAP_MSG_LE_SET_RAND_ADDR               0x02 // response msg type for le_set_rand_addr
#define GAP_MSG_LE_SET_HOST_CHANN_CLASSIF     0x03 // response msg type for le_set_host_chann_classif
#define GAP_MSG_LE_WRITE_DEFAULT_DATA_LEN     0x04 // response msg type for le_write_default_data_len
#define GAP_MSG_LE_READ_RSSI                  0x10 // response msg type for le_read_rssi
#define GAP_MSG_LE_READ_CHANN_MAP             0x11 // response msg type for le_read_chann_map
#define GAP_MSG_LE_DISABLE_SLAVE_LATENCY      0x12 // response msg type for le_disable_slave_latency
#define GAP_MSG_LE_SET_DATA_LEN               0x13 // response msg type for le_set_data_len
#define GAP_MSG_LE_DATA_LEN_CHANGE_INFO       0x14 // Notification msg type for data length changed
#define GAP_MSG_LE_CONN_UPDATE_IND            0x15 // Indication for le connection parameter update
#define GAP_MSG_LE_CREATE_CONN_IND            0x16 // Indication for create le connection
#define GAP_MSG_LE_PHY_UPDATE_INFO            0x17 // Indication for le phyical update information
#define GAP_MSG_LE_UPDATE_PASSED_CHANN_MAP    0x18 // response msg type for
le_update_passed_chann_map

#define GAP_MSG_LE_REMOTE_FEATS_INFO          0x19 // Information for remote device supported features
#define GAP_MSG_LE_BOND_MODIFY_INFO           0x20 // Notification msg type for bond modify
#define GAP_MSG_LE_KEYPRESS_NOTIFY             0x21 // response msg type for le_bond_keypress_notify
#define GAP_MSG_LE_KEYPRESS_NOTIFY_INFO        0x22 // Notification msg type for le_bond_keypress_notify
#define GAP_MSG_LE_GATT_SIGNED_STATUS_INFO    0x23 // Notification msg type for le signed status
information

#define GAP_MSG_LE_SCAN_INFO                 0x30 // Notification msg type for le scan
#define GAP_MSG_LE_DIRECT_ADV_INFO            0x31 // Notification msg type for le direct adv info
#define GAP_MSG_LE_ADV_UPDATE_PARAM           0x40 // response msg type for le_adv_update_param
#define GAP_MSG_LE_ADV_READ_TX_POWER          0x41 // response msg type for le_adv_read_tx_power
#if F_BT_LE_5_0_AE_SCAN_SUPPORT

//gap_ext_scan.h
#define GAP_MSG_LE_EXT_ADV_REPORT_INFO        0x50 // Notification msg type for le extended adv report
#endif

#if F_BT_LE_5_0_AE_ADV_SUPPORT

//gap_ext_adv.h
#define GAP_MSG_LE_EXT_ADV_START_SETTING      0x60 // response msg type for le_ext_adv_start_setting
#define GAP_MSG_LE_EXT_ADV_REMOVE_SET          0x61 // response msg type for le_ext_adv_remove_set
#define GAP_MSG_LE_EXT_ADV_CLEAR_SET           0x62 // response msg type for le_ext_adv_clear_set
#define GAP_MSG_LE_EXT_ADV_ENABLE              0x63 // response msg type for le_ext_adv_enable
#define GAP_MSG_LE_EXT_ADV_DISABLE             0x64 // response msg type for le_ext_adv_disable
#define GAP_MSG_LE_SCAN_REQ_RECEIVED_INFO      0x65 // Notification msg type for le scan received info
#endif

```

2.1.5 GAP Lib

GAP lib 给 APP 提供 GAP 扩展功能，其中非必需的功能模块对 APP 来说是可选的。

2.1.5.1 GAP 扩展功能

1. LE Advertising Extensions 模块

LE Advertising Extensions 模块提供使用 LE Advertising Extensions 特性的功能，包括 extended advertising、extended scan 和 extended create connection。相关接口定义在 gap_ext_adv.h 和 gap_ext_scan.h 中。[BLE BT5 Peripheral Application](#) 和 [BLE BT5 Central Application](#) 中介绍更多关于 LE Advertising Extensions 的内容。

2. Vendor 功能模块

Vendor 功能模块提供一些厂家自定义的扩展功能，更多信息参见 gap_vendor.h。

2.1.5.2 GAP Lib 的使用方法

为使用 GAP 扩展功能，APP 需要在工程中添加 gap_utils.lib 或 gap_bt5.lib。gap_utils.lib 和 gap_bt5.lib 的区别在于 gap_bt5.lib 包括 LE Advertising Extensions 模块。

GAP Lib 的存储路径为 sdk\bin\gap_utils.lib

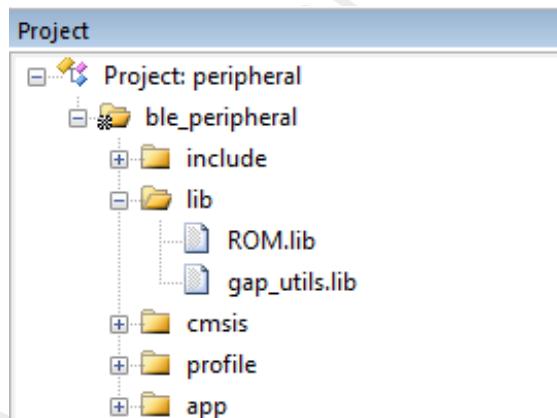


图 2-9 GAP Lib

GAP BT5 Lib 的存储路径为 sdk\bin\gap_bt5.lib

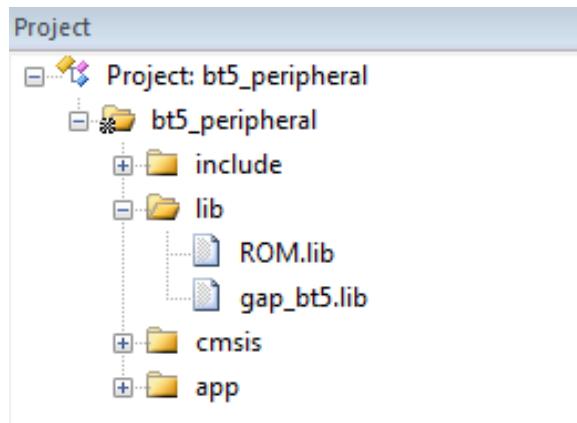


图 2-10 GAP BT5 Lib

在使用扩展功能之前，APP 需要调用 `gap_lib_init()` 初始化 `gap_utils.lib` 或 `gap_bt5.lib`。

```
int main(void)
{
    ...
    gap_lib_init();
    ...
    task_init();
    os_sched_start();
    return 0;
}
```

2.1.6 APP 消息流

APP 消息流如图 2-11 所示，实线框内的步骤是必要步骤，虚线框内的步骤是可选步骤。

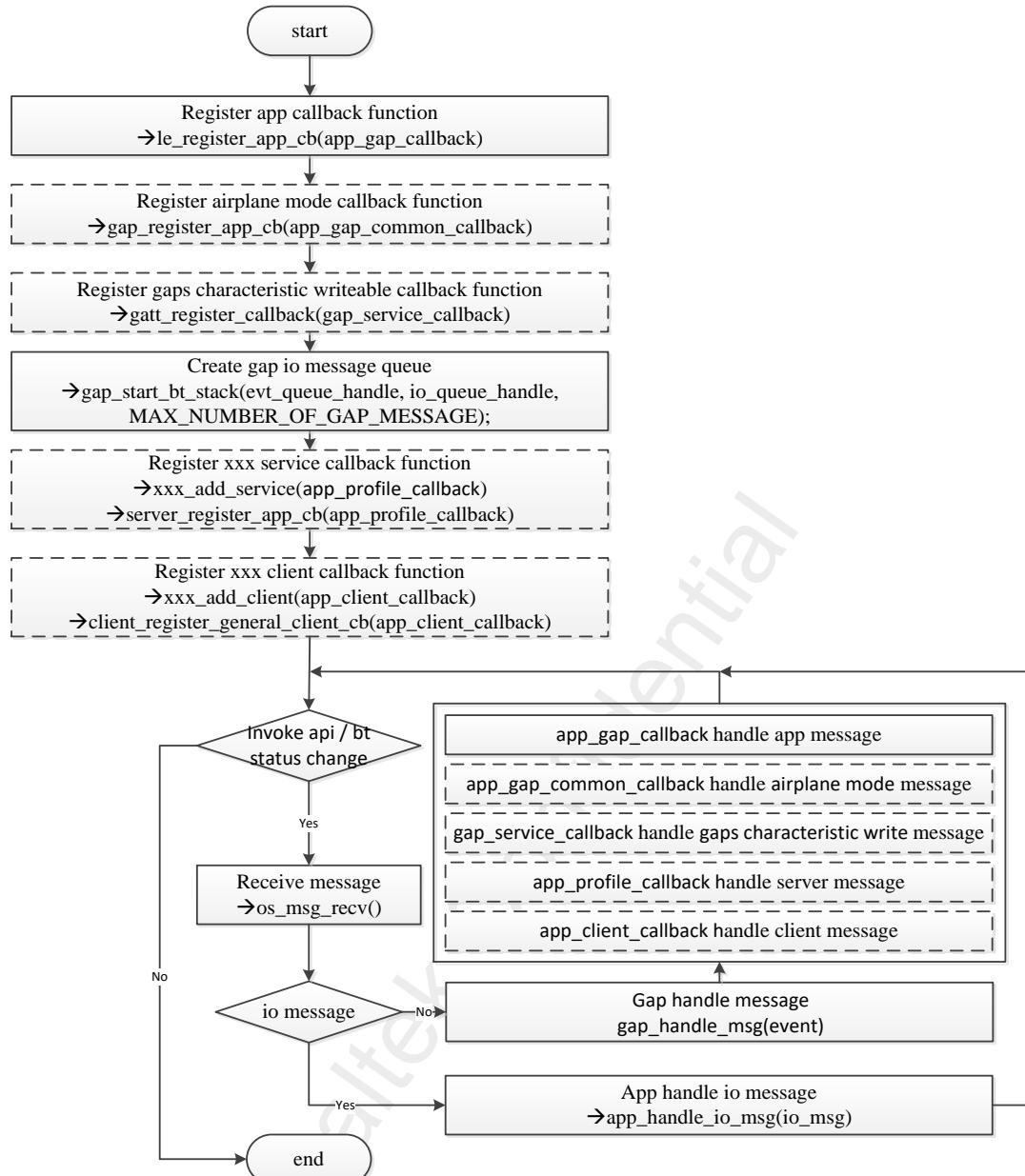


图 2-11 APP 消息流

1. 发送消息给 APP 的两种方法

1) 回调函数

首先，APP 需要注册回调函数。当上行消息送到 GAP 层之后，GAP 层将调用注册的回调函数以通知 APP 处理该消息。

2) 消息队列

首先，APP 需要创建消息队列。当上行消息送到 GAP 层之后，GAP 层将消息发送到消息队列，APP 将循环接收消息队列中的消息。

2. 初始化

1) 注册回调函数

- (1) 为接收 GAP API 消息, APP 需要通过 `le_register_app_cb()`注册 APP 回调函数。
- (2) 为接收 Airplane Mode 消息, APP 需要通过 `gap_register_app_cb()`注册 APP 回调函数。
- (3) 为接收写 GAPS characteristic 的消息, APP 需要通过 `gatt_register_callback()`注册 APP 回调函数。
- (4) 若使用 Peripheral 角色的 APP 需要使用 services, 为接收 server 消息, APP 需要通过 `xxx_add_service()`和 `server_register_app_cb()`注册 service 回调函数。
- (5) 若使用 Central 角色的 APP 需要使用 clients, 为接收 client 消息, APP 需要通过 `xxx_add_client()`注册 client 回调函数。

2) 创建消息队列

为接收蓝牙状态消息, APP 需要通过 `gap_start_bt_stack()`创建 IO 消息队列。

3. 循环接收消息

APP main task 循环接收消息。若收到的事件是发送给 APP 的, APP 收到 IO 消息, 那么 APP 将调用 `app_handle_io_msg()`由 APP 处理该消息。否则, APP 将调用 `gap_handle_msg()`由 GAP 层处理该消息。

4. 消息处理

若消息是由回调函数发送的, 则由初始化过程中注册的函数处理该消息。若消息是由消息队列发送的, 则由另一个函数处理该消息。

2.2 GAP 的初始化和启动流程

本节介绍如何在 `app_le_gap_init()`中配置 LE GAP 参数以及 GAP 内部启动流程。

2.2.1 GAP 参数的初始化

通过修改 `app_le_gap_init()`函数的代码, 在 `main.c` 中实现 GAP 参数的初始化。

2.2.1.1 Device Name 和 Device Appearance 的配置

在 `gap_le.h` 的 `T_GAP_LE_PARAM_TYPE` 中定义参数类型。

2.2.1.1.1 Device Name 的配置

Device Name 的配置用于设置该设备 GAP Service 中 Device Name Characteristic 的值。若在 Advertising 数据中设置 Device Name, 那么 Advertising 数据中的 Device Name 需要与 GAP Service 的 Device Name Characteristic 的值相同, 否则会出现互操作性问题。

```
/** @brief GAP - Advertisement data (max size = 31 bytes, best kept short to conserve power) */
static const uint8_t adv_data[] = {
    /* Flags */
    0x02,           /* length */
    GAP_ADTYPE_FLAGS, /* type="Flags" */
```

```

GAP_ADTYPE_FLAGS_LIMITED | GAP_ADTYPE_FLAGS_BREDR_NOT_SUPPORTED,
/* Service */
0x03,           /* length */
GAP_ADTYPE_16BIT_COMPLETE,
LO_WORD(GATT_UUID_SIMPLE_PROFILE),
HI_WORD(GATT_UUID_SIMPLE_PROFILE),
/* Local name */
0x0F,           /* length */
GAP_ADTYPE_LOCAL_NAME_COMPLETE,
'B', 'L', 'E', '_', 'P', 'E', 'R', 'I', 'P', 'H', 'E', 'R', 'A', 'L',
};

void app_le_gap_init(void)
{
    /* Device name and device appearance */
    uint8_t device_name[GAP_DEVICE_NAME_LEN] = "BLE_PERIPHERAL";
    .....
    /* Set device name and device appearance */
    le_set_gap_param(GAP_PARAM_DEVICE_NAME, GAP_DEVICE_NAME_LEN, device_name);
    .....
}

```

BT Stack 目前支持的 Device Name 字符串的最大长度是 40 字节（包括结束符）。如果 device name 字符串超过 40 字节，则会出现字符串被截断的情况。

#define GAP_DEVICE_NAME_LEN	(39+1)//< Max length of device name, if device name length exceeds it, it will be truncated.
------------------------------------	--

2.2.1.1.2 Device Appearance 的配置

Device Appearance 的配置用于设置该设备 GAP Service 中 Device Appearance Characteristic 的值。若在 Advertising 数据中设置 Device Appearance，那么 Advertising 数据中的 Device Appearance 需要与 GAP Service 的 Device Appearance Characteristic 的值相同，否则会出现互操作性问题。

Device Appearance 用于描述设备的类型，例如键盘、鼠标、温度计、血压计等。在 gap_le_types.h 中定义可以使用的数值。

/** @defgroup GAP_LE_APPEARANCE_VALUES GAP Appearance Values	
* @{	
*/	
#define GAP_GATT_APPEARANCE_UNKNOWN	0
#define GAP_GATT_APPEARANCE_GENERIC_PHONE	64
#define GAP_GATT_APPEARANCE_GENERIC_COMPUTER	128
#define GAP_GATT_APPEARANCE_GENERIC_WATCH	192
#define GAP_GATT_APPEARANCE_WATCH_SPORTS_WATCH	193

示例代码如下所示。

```

/** @brief  GAP - scan response data (max size = 31 bytes) */
static const uint8_t scan_rsp_data[] = {

```

```
0x03,          /* length */
GAP_ADTYPE_APPEARANCE,      /* type="Appearance" */
LO_WORD(GAP_GATT_APPEARANCE_UNKNOWN),
HI_WORD(GAP_GATT_APPEARANCE_UNKNOWN),
};

void app_le_gap_init(void)
{
    /* Device name and device appearance */
    uint16_t appearance = GAP_GATT_APPEARANCE_UNKNOWN;
    .....
    /* Set device name and device appearance */
    le_set_gap_param(GAP_PARAM_APPEARANCE, sizeof(appearance), &appearance);
    .....
}
```

2.2.1.2 Advertising 参数的配置

在 gap_adv.h 的 T_LE_ADV_PARAM_TYPE 中定义 Advertising 参数类型。用户可以配置的 Advertising 参数如下所示：

```
void app_le_gap_init(void)
{
    /* Advertising parameters */
    uint8_t adv_evt_type = GAP_ADTYPE_ADV_IND;
    uint8_t adv_direct_type = GAP_REMOTE_ADDR_LE_PUBLIC;
    uint8_t adv_direct_addr[GAP_BD_ADDR_LEN] = {0};
    uint8_t adv_chann_map = GAP_ADVCHAN_ALL;
    uint8_t adv_filter_policy = GAP_ADV_FILTER_ANY;
    uint16_t adv_int_min = DEFAULT_ADVERTISING_INTERVAL_MIN;
    uint16_t adv_int_max = DEFAULT_ADVERTISING_INTERVAL_MAX;
    .....
    /* Set advertising parameters */
    le_adv_set_param(GAP_PARAM_ADV_EVENT_TYPE, sizeof(adv_evt_type), &adv_evt_type);
    le_adv_set_param(GAP_PARAM_ADV_DIRECT_ADDR_TYPE, sizeof(adv_direct_type), &adv_direct_type);
    le_adv_set_param(GAP_PARAM_ADV_DIRECT_ADDR, sizeof(adv_direct_addr), adv_direct_addr);
    le_adv_set_param(GAP_PARAM_ADV_CHANNEL_MAP, sizeof(adv_chann_map), &adv_chann_map);
    le_adv_set_param(GAP_PARAM_ADV_FILTER_POLICY, sizeof(adv_filter_policy), &adv_filter_policy);
    le_adv_set_param(GAP_PARAM_ADV_INTERVAL_MIN, sizeof(adv_int_min), &adv_int_min);
    le_adv_set_param(GAP_PARAM_ADV_INTERVAL_MAX, sizeof(adv_int_max), &adv_int_max);
    le_adv_set_param(GAP_PARAM_ADV_DATA, sizeof(adv_data), (void *)adv_data);
    le_adv_set_param(GAP_PARAM_SCAN_RSP_DATA, sizeof(scan_rsp_data), (void *)scan_rsp_data);
}
```

adv_evt_type 表示 Advertising 类型，不同类型的 advertising 需要不同的参数，如表 2-1 所示。

表 2-1 Advertising 参数设置

adv_evt_type	GAP_ADTYPE_ ADV_IND	GAP_ADTYPE_ ADV_HDC_DIR	GAP_ADTYPE_ ADV_SCAN_IN	GAP_ADTYPE_ ADV_NONCON	GAP_ADTYPE_AD V_LDC_DIRECT_I
		ECT_IND	D	N_IND	ND
adv_int_min	Y	Ignore	Y	Y	Y
adv_int_max	Y	Ignore	Y	Y	Y
adv_direct_type	Ignore	Y	Ignore	Ignore	Y
adv_direct_addr	Ignore	Y	Ignore	Ignore	Y
adv_chann_map	Y	Y	Y	Y	Y
adv_filter_policy	Y	Ignore	Y	Y	Ignore
allow establish link	Y	Y	N	N	Y

2.2.1.3 Scan 参数的配置

在 gap_scan.h 的 T_LE_SCAN_PARAM_TYPE 中定义 Scan 参数类型。用户可以配置的 Scan 参数如下所示：

```
void app_le_gap_init(void)
{
    /* Scan parameters */
    uint8_t scan_mode = GAP_SCAN_MODE_ACTIVE;
    uint16_t scan_interval = DEFAULT_SCAN_INTERVAL;
    uint16_t scan_window = DEFAULT_SCAN_WINDOW;
    uint8_t scan_filter_policy = GAP_SCAN_FILTER_ANY;
    uint8_t scan_filter_duplicate = GAP_SCAN_FILTER_DUPLICATE_ENABLE;

    .....
    /* Set scan parameters */
    le_scan_set_param(GAP_PARAM_SCAN_MODE, sizeof(scan_mode), &scan_mode);
    le_scan_set_param(GAP_PARAM_SCAN_INTERVAL, sizeof(scan_interval), &scan_interval);
    le_scan_set_param(GAP_PARAM_SCAN_WINDOW, sizeof(scan_window), &scan_window);
    le_scan_set_param(GAP_PARAM_SCAN_FILTER_POLICY, sizeof(scan_filter_policy),
                      &scan_filter_policy);
    le_scan_set_param(GAP_PARAM_SCAN_FILTER_DUPLICATES, sizeof(scan_filter_duplicate),
                      &scan_filter_duplicate);
}
```

参数描述：

1. **scan_mode** - T_GAP_SCAN_MODE
2. **scan_interval** - scan interval 的取值范围： 0x0004 - 0x4000 (单位为 625us)
3. **scan_window** - scan window 的取值范围： 0x0004 - 0x4000 (单位为 625us)
4. **scan_filter_policy** - T_GAP_SCAN_FILTER_POLICY

5. *scan_filter_duplicate* - T_GAP_SCAN_FILTER_DUPLICATE。该参数用于决定是否过滤重复的 Advertising 数据，当 scan_filter_policy 参数为 GAP_SCAN_FILTER_DUPLICATE_ENABLE 时，将在协议栈中过滤重复的 Advertising 数据，且不会通知 APP。

2.2.1.4 Bond Manager 参数的配置

在 gap.h 的 T_GAP_PARAM_TYPE 中和 gap_bond_le.h 的 T_LE_BOND_PARAM_TYPE 中定义参数类型。用户可以配置的 Bond Manager 参数如下所示：

```
void app_le_gap_init(void)
{
    /* GAP Bond Manager parameters */
    uint8_t auth_pair_mode = GAP_PAIRING_MODE_PAIRABLE;
    uint16_t auth_flags = GAP_AUTHEN_BIT_BONDING_FLAG;
    uint8_t auth_io_cap = GAP_IO_CAP_NO_INPUT_NO_OUTPUT;
    uint8_t auth_oob = false;
    uint8_t auth_use_fix_passkey = false;
    uint32_t auth_fix_passkey = 0;
    uint8_t auth_sec_req_enable = false;
    uint16_t auth_sec_req_flags = GAP_AUTHEN_BIT_BONDING_FLAG;
    .....
    /* Setup the GAP Bond Manager */
    gap_set_param(GAP_PARAM_BOND_PAIRING_MODE, sizeof(auth_pair_mode), &auth_pair_mode);
    gap_set_param(GAP_PARAM_BOND_AUTHEN_REQUIREMENTS_FLAGS, sizeof(auth_flags), &auth_flags);
    gap_set_param(GAP_PARAM_BOND_IO_CAPABILITIES, sizeof(auth_io_cap), &auth_io_cap);
    gap_set_param(GAP_PARAM_BOND_OOB_ENABLED, sizeof(auth_oob), &auth_oob);
    le_bond_set_param(GAP_PARAM_BOND_FIXED_PASSKEY, sizeof(auth_fix_passkey), &auth_fix_passkey);
    le_bond_set_param(GAP_PARAM_BOND_FIXED_PASSKEY_ENABLE, sizeof(auth_use_fix_passkey),
                      &auth_use_fix_passkey);
    le_bond_set_param(GAP_PARAM_BOND_SEC_REQ_ENABLE, sizeof(auth_sec_req_enable),
                      &auth_sec_req_enable);
    le_bond_set_param(GAP_PARAM_BOND_SEC_REQ_REQUIREMENT, sizeof(auth_sec_req_flags),
                      &auth_sec_req_flags);
}
```

参数描述：

1. *auth_pair_mode* - 决定设备是否处于可配对模式。
 - 1) GAP_PAIRING_MODE_PAIRABLE: 设备处于可配对模式。
 - 2) GAP_PAIRING_MODE_NO_PAIRING: 设备处于不可配对模式。
2. *auth_flags* - 表示要求的 security 属性的位域。
 - 1) GAP_AUTHEN_BIT_NONE
 - 2) GAP_AUTHEN_BIT_BONDING_FLAG
 - 3) GAP_AUTHEN_BIT_MITM_FLAG

- 4) GAP_AUTHEN_BIT_SC_FLAG
 - 5) GAP_AUTHEN_BIT_KEYPRESS_FLAG
 - 6) GAP_AUTHEN_BIT_FORCE_BONDING_FLAG
 - 7) GAP_AUTHEN_BIT_SC_ONLY_FLAG
3. **auth_io_cap** - T_GAP_IO_CAP, 表示设备的输入输出能力。
 4. **auth_oob** - 表示是否使能 Out of Band (OOB)。
 - 1) true : 设置 OOB 标志位
 - 2) false : 未设置 OOB 标志位
 5. **auth_use_fix_passkey** - 表示当配对方法为 passkey entry 且本地设备需要生成 passkey 时, 是使用随机生成的 passkey 还是固定的 passkey。
 - 1) true : 使用固定的 passkey
 - 2) false : 使用随机生成的 passkey
 6. **auth_fix_passkey** - 配对时使用的固定 passkey 的值, 当 auth_use_fix_passkey 参数为 true 时 auth_fix_passkey 参数是有效的。
 7. **auth_sec_req_enable** - 决定在建立 connection 之后, 是否发起配对流程。
 8. **auth_sec_req_flags** - 表示要求的 security 属性的位域。

2.2.1.5 LE Advertising Extensions 参数的配置

本节内容仅适用于使用 LE Advertising Extensions 的设备。首先，设备需要配置 GAP_PARAM_USE_EXTENDED_ADV 以使用 LE Advertising Extensions。然后，根据设备使用的 GAP 角色配置 extended advertising 相关参数或 extended scan 相关参数。具体信息参见 [BLE BT5 Peripheral Application](#) 和 [BLE BT5 Central Application](#)。

2.2.1.5.1 配置 GAP_PARAM_USE_EXTENDED_ADV

为使用 LE Advertising Extensions, APP 必须配置 GAP_PARAM_USE_EXTENDED_ADV 为 true。

```
void app_le_gap_init(void)
{
    /* LE Advertising Extensions parameters */
    bool use_extended = true;
    .....
    /* Use LE Advertising Extensions */
    le_set_gap_param(GAP_PARAM_USE_EXTENDED_ADV, sizeof(use_extended), &use_extended);
    .....
}
```

2.2.1.5.2 Extended Advertising 相关参数的配置

适用于 Peripheral 角色或 Broadcaster 角色, 更多信息参见 [BLE BT5 Peripheral Application](#)。

2.2.1.5.3 Extended Scan 相关参数的配置

适用于 Central 角色或 Observer 角色，更多信息参见 [BLE BT5 Central Application](#)。

2.2.1.6 其它参数的配置

2.2.1.6.1 GAP_PARAM_SLAVE_INIT_GATT_MTU_REQ 的配置

```
void app_le_gap_init(void)
{
    uint8_t slave_init_mtu_req = false;
    .....
    le_set_gap_param(GAP_PARAM_SLAVE_INIT_GATT_MTU_REQ, sizeof(slave_init_mtu_req),
                     &slave_init_mtu_req);
    .....
}
```

该参数仅适用于 Peripheral 角色，决定在建立 connection 后是否主动发送 exchange MTU request。

2.2.2 GAP 启动流程

1. 在 main() 中初始化 GAP

```
int main(void)
{
    .....
    le_gap_init(APP_MAX_LINKS);
    gap_lib_init();
    app_le_gap_init();
    app_le_profile_init();
    .....
}
```

- 1) [le_gap_init\(\)](#) - 初始化 GAP 并设置 link 数目
- 2) [gap_lib_init\(\)](#) - 初始化 gap_utils.lib 或 gap_bt5.lib
- 3) [app_le_gap_init\(\)](#) - GAP 参数的初始化
- 4) [app_le_profile_init\(\)](#) - 初始化基于 GATT 的 Profiles

2. 在 app task 中启动蓝牙协议层

```
void app_main_task(void *p_param)
{
    uint8_t event;
    os_msg_queue_create(&io_queue_handle, MAX_NUMBER_OF_IO_MESSAGE, sizeof(T_IO_MSG));
    os_msg_queue_create(&evt_queue_handle, MAX_NUMBER_OF_EVENT_MESSAGE, sizeof(uint8_t));
    gap_start_bt_stack(evt_queue_handle, io_queue_handle, MAX_NUMBER_OF_GAP_MESSAGE);
    .....
}
```

APP 需要调用 `gap_start_bt_stack()` 来启动蓝牙协议层和 GAP 初始化流程。

3. GAP 内部初始化流程

GAP 内部初始化流程如图 2-12 所示：

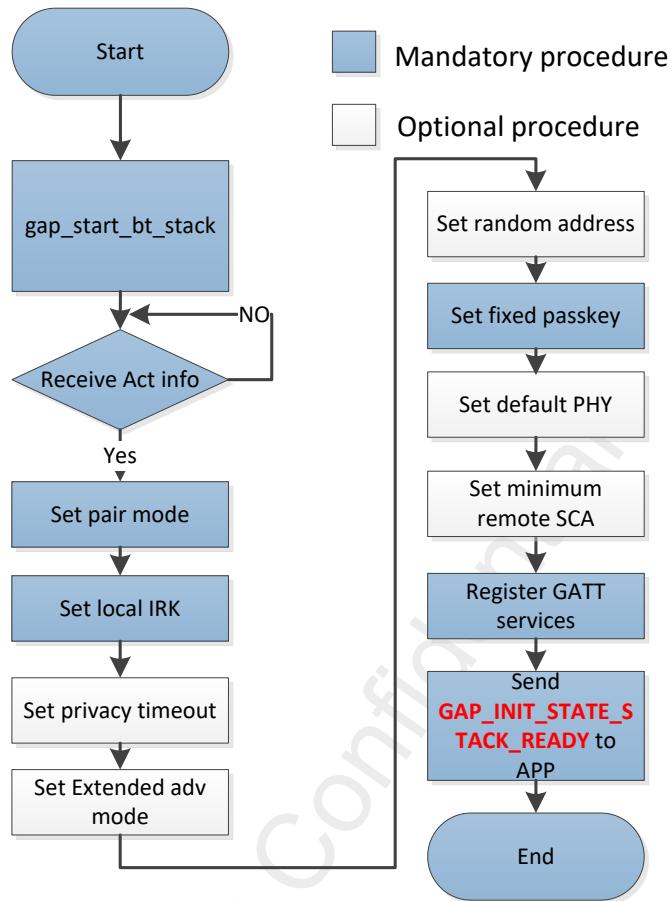


图 2-12 GAP 内部初始化流程

流程图描述：

- 1) **gap_start_bt_stack**: 通过发送注册请求启动 upper stack 初始化流程。
- 2) **Receive act info**: stack 已准备就绪。
- 3) **Set pair mode**: 必要步骤。

设置参数 `GAP_PARAM_BOND_PAIRING_MODE`、

`GAP_PARAM_BOND_AUTHEN_REQUIREMENTS_FLAGS`、`GAP_PARAM_BOND_IO_CAPABILITIES` 和 `GAP_PARAM_BOND_OOB_ENABLED`。

- 4) **Set local IRK**: 必要步骤

设置本地设备的 Identity Resolving Key (IRK)。若 `GAP_PARAM_BOND_GEN_LOCAL_IRK_AUTO` 为 true，则 GAP 层将使用自动生成的 IRK，并将其存入 flash。否则，GAP 层将使用通过 `GAP_PARAM_BOND_SET_LOCAL_IRK` 设置的值，默认值为全零。

- 5) **Set privacy timeout**: 可选步骤

若 APP 调用 `le_privacy_set_param()` 设置 `GAP_PARAM_PRIVACY_TIMEOUT`，则在初始化流程中 GAP

层将设置 Resolvable Private Address Timeout。

6) **Set extended adv mode:** 可选步骤

若 APP 调用 le_set_gap_param() 设置 GAP_PARAM_USE_EXTENDED_ADV，则在初始化流程中 GAP 层将设置 LE Advertising Extensions 模式。

7) **Set random address:** 可选步骤

若 APP 调用 le_set_gap_param() 设置 GAP_PARAM_RANDOM_ADDR，则在初始化流程中 GAP 层将设置 random address。

8) **Set fixed passkey:** 必要步骤

设置 GAP_PARAM_BOND_FIXED_PASSKEY 和 GAP_PARAM_BOND_FIXED_PASSKEY_ENABLE 的值。

9) **Set default physical:** 可选步骤

若 APP 调用 le_set_gap_param() 设置 GAP_PARAM_DEFAULT_PHYS_PREFER、GAP_PARAM_DEFAULT_TX_PHYS_PREFER 或 GAP_PARAM_DEFAULT_RX_PHYS_PREFER，则在初始化流程中 GAP 将设置默认 physical。

10) **Set minimum remote SCA:** 可选步骤

若 APP 调用 le_set_gap_param() 设置 GAP_PARAM_SET_Rem_MIN_SCA，则在初始化流程中 GAP 层将设置默认 remote Sleep Clock Accuracy (SCA) 的最小值。

11) **Register GATT services:** 必要步骤

注册基于 GATT 的 services。

12) **Send GAP_INIT_STATE_STACK_READY to APP:** GAP 初始化流程已完成

2.3 BLE GAP 消息

2.3.1 概述

本节介绍 BLE GAP 消息模块，在 gap_msg.h 中定义 GAP 消息类型和消息数据结构。BLE GAP 消息可以分为以下四种类型：

- *Device 状态消息*
- *Connection 相关消息*
- *Authentication 相关消息*
- *Extended Advertising 状态消息*

BLE GAP 消息处理流程如下所示：

1. APP 调用 *gap_start_bt_stack()* 初始化 BLE GAP 消息模块，初始化代码如下所示：

```
void app_main_task(void *p_param)
{
    uint8_t event;
```

```

os_msg_queue_create(&io_queue_handle, MAX_NUMBER_OF_IO_MESSAGE, sizeof(T_IO_MSG));
os_msg_queue_create(&evt_queue_handle, MAX_NUMBER_OF_EVENT_MESSAGE, sizeof(uint8_t));
gap_start_bt_stack(evt_queue_handle, io_queue_handle, MAX_NUMBER_OF_GAP_MESSAGE);

.....
}

```

2. GAP 向 io_queue_handle 发送 GAP 消息，APP task 接收到 GAP 消息，并调用 app_handle_io_msg() 来处理该消息。(事件：EVENT_IO_TO_APP，类型：IO_MSG_TYPE_BT_STATUS)

```

void app_main_task(void *p_param)
{
    .....
    while (true)
    {
        if (os_msg_recv(evt_queue_handle, &event, 0xFFFFFFFF) == true)
        {
            if (event == EVENT_IO_TO_APP)
            {
                T_IO_MSG io_msg;
                if (os_msg_recv(io_queue_handle, &io_msg, 0) == true)
                {
                    app_handle_io_msg(io_msg);
                }
            }
            .....
        }
    }
}

```

3. GAP 消息处理函数如下所示：

```

void app_handle_io_msg(T_IO_MSG io_msg)
{
    uint16_t msg_type = io_msg.type;
    switch (msg_type)
    {
        case IO_MSG_TYPE_BT_STATUS:
        {
            app_handle_gap_msg(&io_msg);
        }
        break;
    default:
        break;
    }
}

void app_handle_gap_msg(T_IO_MSG *p_gap_msg)
{

```

```

T_LE_GAP_MSG gap_msg;
uint8_t conn_id;
memcpy(&gap_msg, &p_gap_msg->u.param, sizeof(p_gap_msg->u.param));

APP_PRINT_TRACE1("app_handle_gap_msg: subtype %d", p_gap_msg->subtype);
switch (p_gap_msg->subtype)
{
case GAP_MSG_LE_DEV_STATE_CHANGE:
{
    app_handle_dev_state_evt(gap_msg.msg_data.gap_dev_state_change.new_state,
                            gap_msg.msg_data.gap_dev_state_change.cause);
}
break;
.....
}

```

2.3.2 Device 状态消息

2.3.2.1 GAP_MSG_LE_DEV_STATE_CHANGE

该消息用于通知 GAP Device 状态(T_GAP_DEV_STATE)， GAP Device 状态包括以下五种子状态：

- **gap_init_state** : GAP 初始化状态
- **gap_adv_state** : GAP Advertising 状态
- **gap_adv_sub_state**: GAP Advertising 子状态， 该状态仅适用于 gap_adv_state 为 GAP_ADV_STATE_IDLE 的情况。
- **gap_scan_state** : GAP Scan 状态
- **gap_conn_state** : GAP Connection 状态

消息数据结构为 T_GAP_DEV_STATE_CHANGE。

```

/** @brief Device State.*/
typedef struct
{
    uint8_t gap_init_state: 1; //!< @ref GAP_INIT_STATE
    uint8_t gap_adv_sub_state: 1; //!< @ref GAP_ADV_SUB_STATE
    uint8_t gap_adv_state: 2; //!< @ref GAP_ADV_STATE
    uint8_t gap_scan_state: 2; //!< @ref GAP_SCAN_STATE
    uint8_t gap_conn_state: 2; //!< @ref GAP_CONN_STATE
} T_GAP_DEV_STATE;

/** @brief The msg_data of GAP_MSG_LE_DEV_STATE_CHANGE.*/
typedef struct
{

```

```
T_GAP_DEV_STATE new_state;  
uint16_t cause;  
} T_GAP_DEV_STATE_CHANGE;
```

示例代码如下所示：

```
void app_handle_dev_state_evt(T_GAP_DEV_STATE new_state, uint16_t cause)  
{  
    APP_PRINT_INFO4("app_handle_dev_state_evt: init state %d, adv state %d, scan state %d, cause 0x%x",  
                    new_state.gap_init_state, new_state.gap_adv_state,  
                    new_state.gap_scan_state, cause);  
  
    if (gap_dev_state.gap_init_state != new_state.gap_init_state)  
    {  
        if (new_state.gap_init_state == GAP_INIT_STATE_STACK_READY)  
        {  
            APP_PRINT_INFO0("GAP stack ready");  
        }  
    }  
  
    if (gap_dev_state.gap_scan_state != new_state.gap_scan_state)  
    {  
        if (new_state.gap_scan_state == GAP_SCAN_STATE_IDLE)  
        {  
            APP_PRINT_INFO0("GAP scan stop");  
        }  
        else if (new_state.gap_scan_state == GAP_SCAN_STATE_SCANNING)  
        {  
            APP_PRINT_INFO0("GAP scan start");  
        }  
    }  
  
    if (gap_dev_state.gap_adv_state != new_state.gap_adv_state)  
    {  
        if (new_state.gap_adv_state == GAP_ADV_STATE_IDLE)  
        {  
            if (new_state.gap_adv_sub_state == GAP_ADV_TO_IDLE_CAUSE_CONN)  
            {  
                APP_PRINT_INFO0("GAP adv stoped: because connection created");  
            }  
            else  
            {  
                APP_PRINT_INFO0("GAP adv stoped");  
            }  
        }  
        else if (new_state.gap_adv_state == GAP_ADV_STATE_ADVERTISING)  
        {  
            APP_PRINT_INFO0("GAP adv start");  
        }  
    }  
}
```

```
        }
    }
    gap_dev_state = new_state;
}
```

2.3.3 Connection 相关消息

2.3.3.1 GAP_MSG_LE_CONN_STATE_CHANGE

该消息用于通知 Link 状态 (T_GAP_CONN_STATE), 其数据结构为 T_GAP_CONN_STATE_CHANGE。
示例代码如下所示:

```
void app_handle_conn_state_evt(uint8_t conn_id, T_GAP_CONN_STATE new_state, uint16_t disc_cause)
{
    APP_PRINT_INFO4("app_handle_conn_state_evt: conn_id %d old_state %d new_state %d, disc_cause 0x%x",
                    conn_id, gap_conn_state, new_state, disc_cause);

    switch (new_state)
    {
        case GAP_CONN_STATE_DISCONNECTED:
        {
            if ((disc_cause != (HCI_ERR | HCI_ERR_REMOTE_USER_TERMINATE))
                && (disc_cause != (HCI_ERR | HCI_ERR_LOCAL_HOST_TERMINATE)))
            {
                APP_PRINT_ERROR1("app_handle_conn_state_evt: connection lost cause 0x%x", disc_cause);
            }
            le_adv_start();
        }
        break;

        case GAP_CONN_STATE_CONNECTED:
        {
            .....
        }
        break;
    default:
        break;
    }

    gap_conn_state = new_state;
}
```

2.3.3.2 GAP_MSG_LE_CONN_PARAM_UPDATE

该消息用于通知 connection 参数更新状态，更新状态包括三个子状态：

- **GAP_CONN_PARAM_UPDATE_STATUS_PENDING:** 若本地设备调用

le_update_conn_param()更新 Connection 参数，当 Connection 参数更新请求成功但未收到 connection update complete event 时，GAP 层将发送该状态消息。

- **GAP_CONN_PARAM_UPDATE_STATUS_SUCCESS**: 更新成功。
- **GAP_CONN_PARAM_UPDATE_STATUS_FAIL**: 更新失败，参数 cause 表示失败原因。

消息数据结构为 T_GAP_CONN_PARAM_UPDATE。示例代码如下所示：

```
void app_handle_conn_param_update_evt(uint8_t conn_id, uint8_t status, uint16_t cause)
{
    switch (status)
    {
        case GAP_CONN_PARAM_UPDATE_STATUS_SUCCESS:
            .....
            break;
        case GAP_CONN_PARAM_UPDATE_STATUS_FAIL:
            .....
            break;
        case GAP_CONN_PARAM_UPDATE_STATUS_PENDING:
            .....
            break;
    }
}
```

2.3.3.3 GAP_MSG_LE_CONN_MTU_INFO

该消息用于通知 exchange MTU (Maximum Transmission Unit) procedure 已完成。exchange MTU procedure 的目的是更新 client 和 server 之间交互数据包的最大长度，即更新 ATT_MTU。消息数据结构为 T_GAP_CONN_MTU_INFO，示例代码如下所示：

```
void app_handle_conn_mtu_info_evt(uint8_t conn_id, uint16_t mtu_size)
{
    APP_PRINT_INFO2("app_handle_conn_mtu_info_evt: conn_id %d, mtu_size %d", conn_id, mtu_size);
}
```

2.3.4 Authentication 相关消息

配对方法与 Authentication 消息的对应关系如表 2-2 所示：

表 2-2 Authentication 相关消息

Pairing Method	Message
Just Works	GAP_MSG_LE_BOND_JUST_WORK
Numeric Comparison	GAP_MSG_LE_BOND_USER_CONFIRMATION
Passkey Entry	GAP_MSG_LE_BOND_PASSKEY_INPUT GAP_MSG_LE_BOND_PASSKEY_DISPLAY

2.3.4.1 GAP_MSG_LE_AUTHEN_STATE_CHANGE

该消息表示新的 Authentication 状态。

- **GAP_AUTHEN_STATE_STARTED** : Authentication 流程已开始。
- **GAP_AUTHEN_STATE_COMPLETE**: Authentication 流程已完成，参数 cause 表示 Authentication 的结果。

消息数据结构为 T_GAP_AUTHEN_STATE，示例代码如下所示：

```
void app_handle_authen_state_evt(uint8_t conn_id, uint8_t new_state, uint16_t cause)
{
    APP_PRINT_INFO2("app_handle_authen_state_evt:conn_id %d, cause 0x%x", conn_id, cause);
    switch (new_state)
    {
        case GAP_AUTHEN_STATE_STARTED:
        {
            APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_STARTED");
        }
        break;
        case GAP_AUTHEN_STATE_COMPLETE:
        {
            if (cause == GAP_SUCCESS)
            {
                APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_COMPLETE pair
success");
            }
            else
            {
                APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_COMPLETE pair
failed");
            }
        }
        break;
    default:
        break;
    }
}
```

2.3.4.2 GAP_MSG_LE_BOND_PASSKEY_DISPLAY

该消息用于表示配对方法为 Passkey Entry，且本地设备需要显示 Passkey。

在本地设备显示 Passkey，且对端设备需要输入相同的 Passkey。一旦收到该消息，APP 可以在用户终

端界面显示 Passkey（处理 Passkey 的方法取决于 APP），此外 APP 需要调用 le_bond_passkey_display_confirm()确认是否与对端设备配对。

消息数据结构为 T_GAP_BOND_PASSKEY_DISPLAY。示例代码如下所示：

```
void app_handle_gap_msg(T_IO_MSG *p_gap_msg)
{
    .....
    case GAP_MSG_LE_BOND_PASSKEY_DISPLAY:
    {
        uint32_t display_value = 0;
        conn_id = gap_msg.msg_data.gap_bond_passkey_display.conn_id;
        le_bond_get_display_key(conn_id, &display_value);
        APP_PRINT_INFO1("GAP_MSG_LE_BOND_PASSKEY_DISPLAY:passkey %d", display_value);
        le_bond_passkey_display_confirm(conn_id, GAP_CFM_CAUSE_ACCEPT);
    }
    break;
}
```

2.3.4.3 GAP_MSG_LE_BOND_PASSKEY_INPUT

该消息用于表示配对方法为 Passkey Entry，且本地设备需要输入 Passkey。

对端设备会显示 Passkey，且本地设备需要输入相同的 Passkey。一旦收到该消息，APP 需要调用 le_bond_passkey_input_confirm()确认是否与对端设备配对。

消息数据结构为 T_GAP_BOND_PASSKEY_INPUT。示例代码如下所示：

```
void app_handle_gap_msg(T_IO_MSG *p_gap_msg)
{
    .....
    case GAP_MSG_LE_BOND_PASSKEY_INPUT:
    {
        uint32_t passkey = 888888;
        conn_id = gap_msg.msg_data.gap_bond_passkey_input.conn_id;
        APP_PRINT_INFO1("GAP_MSG_LE_BOND_PASSKEY_INPUT: conn_id %d", conn_id);
        le_bond_passkey_input_confirm(conn_id, passkey, GAP_CFM_CAUSE_ACCEPT);
    }
    break;
}
```

2.3.4.4 GAP_MSG_LE_BOND_OOB_INPUT

该消息用于表示配对方法为 OOB。

本地设备需要提供与对端设备交换获得的 OOB 数据。APP 需要调用 le_bond_oob_input_confirm()确认是否与对端设备配对。

消息数据结构为 T_GAP_BOND_OOB_INPUT。示例代码如下所示：

```
void app_handle_gap_msg(T_IO_MSG *p_gap_msg)
{
    .....
    case LE_GAP_MSG_TYPE_BOND_OOB_INPUT:
    {
        uint8_t oob_data[GAP_OOB_LEN] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
        conn_id = gap_msg.msg_data.gap_bond_oob_input.conn_id;
        APP_PRINT_INFO0("GAP_MSG_LE_BOND_OOB_INPUT");
        le_bond_set_param(GAP_PARAM_BOND_OOB_DATA, GAP_OOB_LEN, oob_data);
        le_bond_oob_input_confirm(conn_id, GAP_CFM_CAUSE_ACCEPT);
    }
    break;
}
```

2.3.4.5 GAP_MSG_LE_BOND_USER_CONFIRMATION

该消息用于表示配对方法为 Numeric Comparison。

在本地设备和对端设备均显示需要校验的数值，用户需要确认该数值是否相同。APP 需要调用 le_bond_user_confirm()确认是否与对端设备配对。

消息数据结构为 T_GAP_BOND_USER_CONF。示例代码如下所示：

```
void app_handle_gap_msg(T_IO_MSG *p_gap_msg)
{
    .....
    case GAP_MSG_LE_BOND_USER_CONFIRMATION:
    {
        uint32_t display_value = 0;
        conn_id = gap_msg.msg_data.gap_bond_user_conf.conn_id;
        le_bond_get_display_key(conn_id, &display_value);
        APP_PRINT_INFO1("GAP_MSG_LE_BOND_USER_CONFIRMATION: passkey %d",
                        display_value);
        le_bond_user_confirm(conn_id, GAP_CFM_CAUSE_ACCEPT);
    }
    break;
}
```

2.3.4.6 GAP_MSG_LE_BOND_JUST_WORK

该消息用于表示配对方法为 Just Work。APP 需要调用 le_bond_just_work_confirm()确认是否与对端设备配对。

消息数据结构为 T_GAP_BOND_JUST_WORK_CONF。示例代码如下所示：

```
void app_handle_gap_msg(T_IO_MSG *p_gap_msg)
{
    .....
    case GAP_MSG_LE_BOND_JUST_WORK:
    {
        conn_id = gap_msg.msg_data.gap_bond_just_work_conf.conn_id;
        le_bond_just_work_confirm(conn_id, GAP_CFM_CAUSE_ACCEPT);
        APP_PRINT_INFO0("GAP_MSG_LE_BOND_JUST_WORK");
    }
    break;
}
```

2.3.5 Extended Advertising 状态消息

2.3.5.1 GAP_MSG_LE_EXT_ADV_STATE_CHANGE

该消息用于通知 Extended Advertising 状态 (T_GAP_EXT_ADV_STATE)。

消息数据结构为 T_GAP_EXT_ADV_STATE_CHANGE。示例代码如下所示：

```
void app_handle_ext_adv_state_evt(uint8_t adv_handle, T_GAP_EXT_ADV_STATE new_state, uint16_t cause)
{
    for(int i = 0; i < APP_MAX_ADV_SET; i++)
    {
        if(ext_adv_state[i].adv_handle == adv_handle)
        {
            APP_PRINT_INFO2("app_handle_ext_adv_state_evt: adv_handle = %d oldState = %d",
                           ext_adv_state[i].adv_handle, ext_adv_state[i].ext_adv_state);
            ext_adv_state[i].ext_adv_state = new_state;
            break;
        }
    }
    APP_PRINT_INFO2("app_handle_ext_adv_state_evt: adv_handle = %d newState = %d",
                   adv_handle, new_state);

    switch (new_state)
    {
        /* device is idle */
        case EXT_ADV_STATE_IDLE:
        {
            APP_PRINT_INFO2("EXT_ADV_STATE_IDLE: adv_handle %d, cause 0x%x", adv_handle, cause);
        }
        break;

        /* device is advertising */
    }
}
```

```
case EXT_ADV_STATE_ADVERTISING:  
{  
    APP_PRINT_INFO2("EXT_ADV_STATE_ADVERTISING: adv_handle %d, cause 0x%x",  
adv_handle, cause);  
}  
break;  
  
default:  
break;  
}  
}
```

2.4 BLE GAP 回调函数

本节介绍 BLE GAP 回调函数，GAP 层使用注册的回调函数发送消息给 APP。

不同于 BLE GAP 消息，回调函数是直接被 GAP 层调用的。因此，不建议在回调函数内执行耗时操作，耗时操作会使底层处理流程延缓和暂停，在某些情况下会引发异常。若 APP 在收到 GAP 层发送的消息后确实需要执行耗时操作，在 APP 处理该消息时，可以通过 APP 回调函数将该消息发送到 APP 的队列，APP 回调函数将在把消息发送到队列之后结束，因此该操作不会延缓底层处理流程。

BLE GAP 回调函数的使用方法如下所示：

1. 注册回调函数

```
void app_le_gap_init(void)  
{  
    ....  
    le_register_app_cb(app_gap_callback);  
}
```

2. 处理 GAP 回调函数消息

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)  
{  
    T_APP_RESULT result = APP_RESULT_SUCCESS;  
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;  
    switch (cb_type)  
    {  
        case GAP_MSG_LE_DATA_LEN_CHANGE_INFO:  
            APP_PRINT_INFO3("GAP_MSG_LE_DATA_LEN_CHANGE_INFO: conn_id %d, tx octets 0x%x,  
                           max_tx_time 0x%x",  
                           p_data->p_le_data_len_change_info->conn_id,  
                           p_data->p_le_data_len_change_info->max_tx_octets,  
                           p_data->p_le_data_len_change_info->max_tx_time);  
            break;  
        ....  
    }  
}
```

{}

2.4.1 BLE GAP 回调函数消息概述

本节介绍 GAP 回调函数消息，在 gap_callback_le.h 中定义 GAP 回调函数消息类型和消息数据。由于 GAP 层提供的大部分接口都是异步的，因此 GAP 层使用回调函数发送响应信息给 APP。例如，APP 调用 le_read_rssi() 读取 Received Signal Strength Indication (RSSI)，若返回值为 GAP_CAUSE_SUCCESS，则表示成功发送请求。此时，APP 需要等待 GAP_MSG_LE_READ_RSSI 消息以获取结果。

BLE GAP 回调函数消息的详细信息如下所示：

1. gap_le.h 相关消息

表 2-3 gap_le.h 相关消息

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE MODIFY_WHITE_LIST	T_LE MODIFY_WHITE_LIST_RSP *p_le_modify_white_list_rsp;	le_modify_white_list
GAP_MSG_LE_SET RAND_ADDR	T_LE_SET RAND_ADDR_RSP *p_le_set_rand_addr_rsp;	le_set_rand_addr
GAP_MSG_LE_SET_HOST_CHANN_CLA SSIF	T_LE_SET_HOST_CHANN_CLASSIF _RSP *p_le_set_host_chann_classif_rsp;	le_set_host_chann_classifi
GAP_MSG_LE_VENDOR_SET_MIN_REM _SCA	T LE CAUSE le_cause;	le_vendor_set_rem_min_sca

2. gap_conn_le.h 相关消息

表 2-4 gap_conn_le.h 相关消息

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_READ_RSSI	T LE READ RSSI RSP *p_le_read_rssi_rsp;	le_read_rssi
GAP_MSG_LE_READ_CHANN_MAP	T LE READ CHANN MAP RSP *p_le_read_chann_map_rsp;	le_read_chann_map
GAP_MSG_LE_DISABLE_SLAVE_LATEN CY	T LE DISABLE SLAVE LATENCY_R SP *p_le_disable_slave_latency_rsp;	le_disable_slave_latency
GAP_MSG_LE_SET_DATA_LEN	T LE SET DATA LEN RSP *p_le_set_data_len_rsp;	le_set_data_len
GAP_MSG_LE_DATA_LEN_CHANGE_IN FO	T LE DATA LEN CHANGE INFO *p_le_data_len_change_info;	
GAP_MSG_LE_CONN_UPDATE_IND	T LE CONN UPDATE IND *p_le_conn_update_ind;	

GAP_MSG_LE_CREATE_CONN_IND	T_LE_CREATE_CONN_IND *p_le_create_conn_ind;	
GAP_MSG_LE_PHY_UPDATE_INFO	T_LE_PHY_UPDATE_INFO *p_le_phy_update_info;	
GAP_MSG_LE_UPDATE_PASSED_CHAN_N_MAP	T_LE_UPDATE_PASSED_CHANN_MA P_RSP *p_le_update_passed_chann_map_rsp;	le_update_passed_chann_map
GAP_MSG_LE_REMOTE_FEATS_INFO	T_LE_REMOTE_FEATS_INFO *p_le_remote_feats_info;	
GAP_MSG_LE_SET_CONN_TX_PWR	T_LE_CAUSE le_cause;	le_set_conn_tx_power

1) GAP_MSG_LE_DATA_LEN_CHANGE_INFO

该消息用于向 APP 通知在 Link Layer 的发送或接收方向其最大 Payload 长度或数据包的最大传输时间的变化。

2) GAP_MSG_LE_CONN_UPDATE_IND

该消息仅适用于 Central 角色。当对端设备请求更新 Connection 参数时，GAP 层将通过回调函数发送该消息并检查回调函数的返回值。因此，APP 可以返回 APP_RESULT_ACCEPT 以接受参数更新，或者返回 APP_RESULT_REJECT 以拒绝参数更新。

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    .....
    case GAP_MSG_LE_CONN_UPDATE_IND:
        APP_PRINT_INFO5("GAP_MSG_LE_CONN_UPDATE_IND: conn_id %d, conn_interval_max 0x%x,
                        conn_interval_min 0x%x, conn_latency 0x%x, supervision_timeout 0x%x",
                        p_data->p_le_conn_update_ind->conn_id,
                        p_data->p_le_conn_update_ind->conn_interval_max,
                        p_data->p_le_conn_update_ind->conn_interval_min,
                        p_data->p_le_conn_update_ind->conn_latency,
                        p_data->p_le_conn_update_ind->supervision_timeout);
        /* if reject the proposed connection parameter from peer device, use APP_RESULT_REJECT. */
        result = APP_RESULT_ACCEPT;
        break;
}
```

3) GAP_MSG_LE_CREATE_CONN_IND

该消息仅适用于 Peripheral 角色。由 APP 决定是否建立 connection。当 central 设备发起 connection 时，默认情况下，GAP 层不会发送该消息并直接接受 connection。若 APP 希望使用该功能，需要将 GAP_PARAM_HANDLE_CREATE_CONN_IND 设为 true。示例代码如下所示：

```
void app_le_gap_init(void)
{
    .....
```

```

uint8_t handle_conn_ind = true;
le_set_gap_param(GAP_PARAM_HANDLE_CREATE_CONN_IND, sizeof(handle_conn_ind),
                 &handle_conn_ind);
}

T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    .....
    case GAP_MSG_LE_CREATE_CONN_IND:
        /* if reject the connection from peer device, use APP_RESULT_REJECT. */
        result = APP_RESULT_ACCEPT;
        break;
}

```

4) GAP_MSG_LE_PHY_UPDATE_INFO

该消息表示 Controller 已经切换正在使用的发射机 PHY 或接收机 PHY。

5) GAP_MSG_LE_REMOTE_FEATS_INFO

在 connection 建立成功后，controller 会主动读取对端设备的 feature。读取结果会通过该消息通知给 application。

3. gap_bond_le.h 相关消息

表 2-5 gap_bond_le.h 相关消息

Callback type(cb_type)	Callback data (p_cb_data)	Reference API
GAP_MSG_LE_BOND MODIFY_INFO	T LE_BOND MODIFY_INFO *p_le_bond_modify_info;	
GAP_MSG_LE_KEYPRESS_NOTIFY	T LE_KEYPRESS NOTIFY_RSP *p_le_keypress_notify_rsp;	le_bond_keypress_notif y
GAP_MSG_LE_KEYPRESS_NOTIFY_INFO	T LE_KEYPRESS NOTIFY_INFO *p_le_keypress_notify_info;	
GAP_MSG_LE_GATT_SIGNED_STATUS_INFO	T LE_GATT SIGNED STATUS_INFO *p_le_gatt_signed_status_info;	

1) GAP_MSG_LE_KEYPRESS_NOTIFY_INFO

该消息表示 SMP 已收到 keypress notification。

2) GAP_MSG_LE_GATT_SIGNED_STATUS_INFO

该消息表示 GATT signed 状态信息。

3) GAP_MSG_LE_BOND MODIFY_INFO

该消息用于向 APP 通知绑定信息已变更，更多信息参见 [LE 密钥管理](#)。

4. gap_scan.h 相关消息

表 2-6 gap_scan.h 相关消息

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_SCAN_INFO	T_LE_SCAN_INFO *p_le_scan_info;	le_scan_start
GAP_MSG_LE_DIRECT_ADV_INFO	T_LE_DIRECT_ADV_INFO *p_le_direct_adv_info;	

1) GAP_MSG_LE_SCAN_INFO

Scan 状态为 GAP_SCAN_STATE_SCANNING, 当蓝牙协议层收到 advertising 数据或 scan response 数据时, GAP 将发送该消息以通知 APP。

2) GAP_MSG_LE_DIRECT_ADV_INFO

Scan 状态为 GAP_SCAN_STATE_SCANNING 且 Scan Filter Policy 为 GAP_SCAN_FILTER_ANY_RPA 或 GAP_SCAN_FILTER_WHITE_LIST_RPA, 当蓝牙协议层收到 directed advertising 数据包且其 initiator 地址为 Resolvable Private Address (RPA), 无法解析该 RPA 时, GAP 层将发送该消息以通知 APP。

5. gap_adv.h 相关消息

表 2-7 gap_adv.h 相关消息

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_ADV_UPDATE_PARAM	T_LE_ADV_UPDATE_PARAM_RSP *p_le_adv_update_param_rsp;	le_adv_update_param
GAP_MSG_LE_ADV_READ_TX_POWER	T_LE_ADV_READ_TX_POWER_RSP *p_le_adv_read_tx_power_rsp;	le_adv_read_tx_power
GAP_MSG_LE_ADV_SET_TX_POWER	T_LE_CAUSE le_cause;	le_adv_set_tx_power
GAP_MSG_LE_VENDOR_ONE_SHOT_ADV	T_LE_CAUSE le_cause;	le_vendor_one_shot_adv

6. gap_dtm.h 相关消息

表 2-8 gap_dtm.h 相关消息

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_DTM_RECEIVER_TEST	T_LE_CAUSE le_cause;	le_dtm_receiver_test
GAP_MSG_LE_DTM_TRANSMITTER_TEST	T_LE_CAUSE le_cause;	le_dtm_transmitter_test
GAP_MSG_LE_DTM_TEST_END	T_LE_DTM_TEST_END_ RSP *p_le_dtm_test_end_rsp;	le_dtm_test_end

GAP_MSG_LE_DTM_ENHANCED_RECEIVER_TE ST	T_LE_CAUSE le_cause;	le_dtm_enhanced_receiver_te st
GAP_MSG_LE_DTM_ENHANCED_TRANSMITTER TEST	T_LE_CAUSE le_cause;	le_dtm_enhanced_transmitter _test

7. GAP Lib 相关消息

GAP Lib 提供扩展功能。

1) gap_vendor.h 相关消息

表 2-9 gap_vendor.h 相关消息

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_VENDOR_ADV_3_DATA_ENABLE	T_LE_CAUSE le_cause;	le_vendor_adv_3_data_enable
GAP_MSG_LE_VENDOR_ADV_3_DATA_SET	T_LE_VENDOR_ADV_3_DATA_SET_RSP *p_le_vendor_adv_3_data_set_rsp;	le_vendor_adv_3_data_set
GAP_MSG_LE_VENDOR_DROP_ACL_DATA	T_LE_CAUSE le_cause;	le_vendor_drop_acl_data
GAP_MSG_GAP_SW_RESET	T_LE_CAUSE le_cause;	gap_sw_reset_req

2) gap_ext_scan.h 相关消息(仅适用于 gap_bt5.lib)

表 2-10 gap_ext_scan.h 相关消息

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_EXT_ADV_REPORT_INFO	T_LE_EXT_ADV_REPORT_INFO *p_le_ext_adv_report_info;	le_ext_scan_start

(1) GAP_MSG_LE_EXT_ADV_REPORT_INFO

使用 LE Advertising Extensions 且 Scan 状态为 GAP_SCAN_STATE_SCANNING，当蓝牙协议层收到 advertising 数据或 scan response 数据时，GAP 层将发送该消息以通知 APP。

3) gap_ext_adv.h 相关消息 (仅适用于 gap_bt5.lib)

表 2-11 gap_ext_adv.h 相关消息

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_EXT_ADV_START_SETTING_RSP	T_LE_EXT_ADV_START_SETTING_RSP *p_le_ext_adv_start_setting_rsp;	le_ext_adv_start_setting
GAP_MSG_LE_EXT_ADV_REMOVE_SET	T_LE_EXT_ADV_REMOVE_SET_RSP *p_le_ext_adv_remove_set_rsp;	le_ext_adv_remove_set

GAP_MSG_LE_EXT_ADV_CLEAR_SET	T_LE_EXT_ADV_CLEAR_SET_RSP *p_le_ext_adv_clear_set_rsp;	le_ext_adv_clear_set
GAP_MSG_LE_EXT_ADV_ENABLE	T_LE_CAUSE le_cause;	le_ext_adv_enable
GAP_MSG_LE_EXT_ADV_DISABLE	T_LE_CAUSE le_cause;	le_ext_adv_disable
GAP_MSG_LE_SCAN_REQ_RECEIVED_IN FO	T_LE_SCAN_REQ_RECEIVED_INFO *p_le_scan_req_received_info;	le_ext_adv_enable

(1) GAP_MSG_LE_EXT_ADV_START_SETTING

该消息表示本地设备为一个 advertising set 设置 extended advertising 相关参数的操作已完成。

(2) GAP_MSG_LE_EXT_ADV_REMOVE_SET

该消息表示本地设备移除一个 advertising set 的操作已完成。

(3) GAP_MSG_LE_EXT_ADV_CLEAR_SET

该消息表示本地设备清除所有 advertising sets 的操作已完成。

(4) GAP_MSG_LE_EXT_ADV_ENABLE

该消息表示本地设备启动一个或多个 advertising sets 的 extended advertising 的操作已完成。

(5) GAP_MSG_LE_EXT_ADV_DISABLE

该消息表示本地设备停止一个或多个 advertising sets 的 extended advertising 的操作已完成。

(6) GAP_MSG_LE_SCAN_REQ_RECEIVED_INFO

调用 le_ext_adv_set_adv_param() 使能 scan request notifications 且 Extended Advertising 状态为 EXT_ADV_STATE_ADVERTISING，当蓝牙协议层收到 scan request，GAP 层将使用该消息通知 APP。

2.5 BLE GAP 用例

本节介绍如何使用 BLE GAP 接口，以下为一些典型用例。

2.5.1 Airplane Mode 设置

示例代码位于 BLE Scatternet Application 中。

1. 注册 GAP Common 回调函数

APP 需要调用 gap_register_app_cb() 以注册回调函数。

```
void app_le_gap_init(void)
{
    .....
    gap_register_app_cb(app_gap_common_callback);
}
```

2. GAP Common 回调函数消息处理

```
void app_gap_common_callback(uint8_t cb_type, void *p_cb_data)
```

```
{  
    T_GAP_CB_DATA cb_data;  
    memcpy(&cb_data, p_cb_data, sizeof(T_GAP_CB_DATA));  
    APP_PRINT_INFO1("app_gap_common_callback: cb_type = %d", cb_type);  
    switch (cb_type)  
    {  
        case GAP_MSG_WRITE_AIRPLAN_MODE:  
            APP_PRINT_INFO1("GAP_MSG_WRITE_AIRPLAN_MODE: cause 0x%x",  
                           cb_data.p_gap_write_airplan_mode_rsp->cause);  
            break;  
        case GAP_MSG_READ_AIRPLAN_MODE:  
            APP_PRINT_INFO2("GAP_MSG_READ_AIRPLAN_MODE: cause 0x%x, mode %d",  
                           cb_data.p_gap_read_airplan_mode_rsp->cause,  
                           cb_data.p_gap_read_airplan_mode_rsp->mode);  
            break;  
        default:  
            break;  
    }  
    return;  
}
```

该回调函数用于处理 GAP_MSG_WRITE_AIRPLAN_MODE 和 GAP_MSG_READ_AIRPLAN_MODE 消息。

3. 相关 API

gap_write_airplan_mode()用于写 airplane mode。

gap_read_airplan_mode()用于读 airplane mode。

```
static T_USER_CMD_PARSE_RESULT cmd_wairplane(T_USER_CMD_PARSED_VALUE *p_parse_value)  
{  
    T_GAP_CAUSE cause;  
    uint8_t mode = p_parse_value->dw_param[0];  
    cause = gap_write_airplan_mode(mode);  
    return (T_USER_CMD_PARSE_RESULT)cause;  
}  
  
static T_USER_CMD_PARSE_RESULT cmd_rairplane(T_USER_CMD_PARSED_VALUE *p_parse_value)  
{  
    T_GAP_CAUSE cause;  
    cause = gap_read_airplan_mode();  
    return (T_USER_CMD_PARSE_RESULT)cause;  
}
```

2.5.2 本地设备 IRK 的设置

IRK 是用于生成和解析 RPA 的 128 位密钥。本地设备 IRK 的默认值为全零。

若设备支持 RPA 的生成且生成 RPA 作为其本地地址，在分发 IRK 时该设备必须发送包含有效 IRK 的 Identity Information。

若设备不支持生成 RPA 作为本地地址，在分发 IRK 时该设备发送包含 IRK 为全零的 Identity Information。此时，本地设备不会使用 RPA，APP 不需要设置 IRK。

GAP 层提供两种设置本地设备 IRK 的方法。

1. 自动生成本地设备的 IRK

若 APP 希望使用该功能，那么 APP 需要将 GAP_PARAM_BOND_GEN_LOCAL_IRK_AUTO 设为 true。当 GAP 启动时，GAP 层会调用 flash_load_local_irk() 以载入本地设备 IRK。若载入失败，则会自动生成 IRK，然后调用 flash_save_local_irk() 保存本地设备 IRK。因此，使用自动生成本地设备 IRK 功能时，APP 不能使用 flash_load_local_irk() 和 flash_save_local_irk()。

```
void app_le_gap_init(void)
{
    .....
    uint8_t irk_auto = true;
    le_bond_set_param(GAP_PARAM_BOND_GEN_LOCAL_IRK_AUTO, sizeof(uint8_t), &irk_auto);
}
```

2. APP 生成本地设备的 IRK

默认情况下，GAP 层会使用该方法，默认 IRK 为全零。APP 可以调用 le_bond_set_param() 设置 GAP_PARAM_BOND_SET_LOCAL_IRK 以更改本地设备 IRK。

```
void app_le_gap_init(void)
{
    T_LOCAL_IRK le_local_irk = {0, 1, 0, 5, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 9};
    le_bond_set_param(GAP_PARAM_BOND_SET_LOCAL_IRK, GAP_KEY_LEN, le_local_irk.local_irk);
}
```

该参数仅在 GAP 启动过程中是有效的，因此，APP 需要在调用 gap_start_bt_stack() 之前设置 GAP_PARAM_BOND_SET_LOCAL_IRK。

APP 可以生成本地设备 IRK，并调用 flash_save_local_irk() 保存 IRK。之后 APP 可以调用 flash_load_local_irk() 以载入本地设备 IRK，然后设置 GAP_PARAM_BOND_SET_LOCAL_IRK 以更改本地设备 IRK。

2.5.3 GAP Service Characteristic 的可写属性

示例代码位于 BLE Scatternet Application 中。

GAP service 的 Device Name characteristic 和 Device Appearance characteristic 具有可选的可写属性，该可写属性默认是关闭的。APP 可以通过调用 gaps_set_parameter() 设置 GAPS_PARAM_APPEARANCE_PROPERTY 和 GAPS_PARAM_DEVICE_NAME_PROPERTY 来配置可写属性。

1. 可写属性的配置

```
void app_le_gap_init(void)
{
    uint8_t appearance_prop = GAPS_PROPERTY_WRITE_ENABLE;
    uint8_t device_name_prop = GAPS_PROPERTY_WRITE_ENABLE;
    T_LOCAL_APPEARANCE appearance_local;
    T_LOCAL_NAME local_device_name;
    if (flash_load_local_appearance(&appearance_local) == 0)
    {
        gaps_set_parameter(GAPS_PARAM_APPEARANCE, sizeof(uint16_t),
                           &appearance_local.local_appearance);
    }
    if (flash_load_local_name(&local_device_name) == 0)
    {
        gaps_set_parameter(GAPS_PARAM_DEVICE_NAME, GAP_DEVICE_NAME_LEN,
                           local_device_name.local_name);
    }
    gaps_set_parameter(GAPS_PARAM_APPEARANCE_PROPERTY, sizeof(appearance_prop),
                       &appearance_prop);
    gaps_set_parameter(GAPS_PARAM_DEVICE_NAME_PROPERTY, sizeof(device_name_prop),
                       &device_name_prop);
    gatt_register_callback(gap_service_callback);
}
```

2. GAP Service 回调函数消息处理

APP 需要调用 gatt_register_callback() 以注册回调函数，该回调函数用于处理 GAP service 消息。

```
T_APP_RESULT gap_service_callback(T_SERVER_ID service_id, void *p_para)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_GAPS_CALLBACK_DATA *p_gap_data = (T_GAPS_CALLBACK_DATA *)p_para;
    APP_PRINT_INFO2("gap_service_callback conn_id = %d msg_type = %d\n", p_gap_data->conn_id,
                    p_gap_data->msg_type);
    if (p_gap_data->msg_type == SERVICE_CALLBACK_TYPE_WRITE_CHAR_VALUE)
    {
        switch (p_gap_data->msg_data.opcode)
        {
            case GAPS_WRITE_DEVICE_NAME:
            {
                T_LOCAL_NAME device_name;
                memcpy(device_name.local_name, p_gap_data->msg_data.p_value,
                       p_gap_data->msg_data.len);
                device_name.local_name[p_gap_data->msg_data.len] = 0;
                flash_save_local_name(&device_name);
            }
            break;
        }
    }
}
```

```
case GAPS_WRITE_APPEARANCE:  
{  
    uint16_t appearance_val;  
    T_LOCAL_APPEARANCE appearance;  
    LE_ARRAY_TO_UINT16(appearance_val, p_gap_data->msg_data.p_value);  
    appearance.local_appearance = appearance_val;  
    flash_save_local_appearance(&appearance);  
}  
break;  
default:  
break;  
}  
}  
return result;  
}
```

APP 需要将 device name 和 device appearance 保存到 Flash，具体内容参见[本地协议栈信息存储](#)。

2.5.4 本地设备使用 Static Random Address

示例代码位于 BLE Scatternet Application 中。

在 advertising、scanning 和 connection 时，默认使用的 local address type 是 Public Address，可以配置为 Static Random Address。

1. Random address 的生成和存储

APP 第一次可以调用 le_gen_rand_addr()生成 static random address。然后将生成的地址存储到 Flash。如果 Flash 已经存储过 random 地址，则可以直接使用。然后调用 le_set_gap_param() 的 GAP_PARAM_RANDOM_ADDR 来设置 random address。

2. 设置 Identity Address

Stack 默认使用 public address 作为 Identity Address。APP 需要调用 le_cfg_local_identity_address()将 Identity Address 修改为 static random address。不正确的 Identity Address 设置会导致配对后无法重连。

3. 设置 local address type

Peripheral 角色或 Broadcaster 角色调用 le_adv_set_param()设置 local address type 以使用本地设备的 Static Random Address。Central 角色或 Observer 角色调用 le_scan_set_param()设置 local address type 以使用本地设备的 Static Random Address。示例代码如下所示：

```
void app_le_gap_init(void)  
{  
    .....  
    T_APP_STATIC_RANDOM_ADDR random_addr;  
    bool gen_addr = true;  
    uint8_t local_bd_type = GAP_LOCAL_ADDR_LE_RANDOM;  
    if (app_load_static_random_address(&random_addr) == 0)
```

```
{  
    if (random_addr.is_exist == true)  
    {  
        gen_addr = false;  
    }  
}  
  
if (gen_addr)  
{  
    if (le_gen_rand_addr(GAP_RAND_ADDR_STATIC, random_addr.bd_addr) == GAP_CAUSE_SUCCESS)  
    {  
        random_addr.is_exist = true;  
        app_save_static_random_address(&random_addr);  
    }  
}  
  
le_cfg_local_identity_address(random_addr.bd_addr, GAP_IDENT_ADDR RAND);  
le_set_gap_param(GAP_PARAM_RANDOM_ADDR, 6, random_addr.bd_addr);  
//only for peripheral,broadcaster  
le_adv_set_param(GAP_PARAM_ADV_LOCAL_ADDR_TYPE, sizeof(local_bd_type), &local_bd_type);  
//only for central,observer  
le_scan_set_param(GAP_PARAM_SCAN_LOCAL_ADDR_TYPE, sizeof(local_bd_type), &local_bd_type);  
.....  
}
```

Central 角色调用 le_connect()设置 local address type 以使用本地设备的 Static Random Address。示例代码如下所示：

```
static T_USER_CMD_PARSE_RESULT cmd_condev(T_USER_CMD_PARSED_VALUE *p_parse_value)  
{  
    .....  
#if F_BT_LE_USE_STATIC_RANDOM_ADDR  
    T_GAP_LOCAL_ADDR_TYPE local_addr_type = GAP_LOCAL_ADDR_LE_RANDOM;  
#else  
    T_GAP_LOCAL_ADDR_TYPE local_addr_type = GAP_LOCAL_ADDR_LE_PUBLIC;  
#endif  
    .....  
    cause = le_connect(GAP_PHYS_CONN_INIT_1M_BIT,  
                       dev_list[dev_idx].bd_addr,  
                       (T_GAP_REMOTE_ADDR_TYPE)dev_list[dev_idx].bd_type,  
                       local_addr_type,  
                       1000);  
    .....  
}
```

2.5.5 Physical (PHY) 设置

示例代码位于 BLE Scatternet Application 中。

LE 中必须实现符号速率为 1 mega symbol per second (Msym/s), 一个 symbol 表示一个 bit, 支持的比特率为 1 megabit per second (Mb/s), 即 **LE 1M PHY**。1 Msym/s 符号速率可以选择支持纠错编码, 即 **LE Coded PHY**, 共有两种编码方案: S = 2, 即两个 symbol 表示一个 bit, 支持的比特率为 500 Kb/s; S = 8, 即八个 symbol 表示一个 bit, 支持的比特率为 125 Kb/s。若支持可选符号速率 2 Msym/s, 比特率为 2 Mb/s, 即 **LE 2M PHY**。2 Msym/s 符号速率只支持未编码的数据, LE 1M PHY 和 LE 2M PHY 统称为 LE Uncoded PHYs^[1]。

1. 设置 Default PHY

APP 可以指定其发射机 PHY 和接收机 PHY 的优先值, 用于随后所有建立在 LE transport 上的 connection。

```
void app_le_gap_init(void)
{
    uint8_t phys_prefer = GAP_PHYS_PREFER_ALL;
    uint8_t tx_phys_prefer = GAP_PHYS_PREFER_1M_BIT | GAP_PHYS_PREFER_2M_BIT |
                           GAP_PHYS_PREFER_CODED_BIT;
    uint8_t rx_phys_prefer = GAP_PHYS_PREFER_1M_BIT | GAP_PHYS_PREFER_2M_BIT |
                           GAP_PHYS_PREFER_CODED_BIT;
    le_set_gap_param(GAP_PARAM_DEFAULT_PHYS_PREFER, sizeof(phys_prefer), &phys_prefer);
    le_set_gap_param(GAP_PARAM_DEFAULT_TX_PHYS_PREFER, sizeof(tx_phys_prefer), &tx_phys_prefer);
    le_set_gap_param(GAP_PARAM_DEFAULT_RX_PHYS_PREFER, sizeof(rx_phys_prefer), &rx_phys_prefer);
}
```

2. 读取 connection 的 PHY 类型

成功建立 connection 后, APP 可以调用 le_get_conn_param() 以读取 TX PHY 和 RX PHY 类型。

```
void app_handle_conn_state_evt(uint8_t conn_id, T_GAP_CONN_STATE new_state, uint16_t disc_cause)
{
    .....
    switch (new_state)
    {
        case GAP_CONN_STATE_CONNECTED:
        {
            .....
            data_uart_print("Connected success conn_id %d\r\n", conn_id);
#if F_BT_LE_5_0_SET_PHY_SUPPORT
            uint8_t tx_phy;
            uint8_t rx_phy;
            le_get_conn_param(GAP_PARAM_CONN_RX_PHY_TYPE, &rx_phy, conn_id);
            le_get_conn_param(GAP_PARAM_CONN_TX_PHY_TYPE, &tx_phy, conn_id);
            APP_PRINT_INFO2("GAP_CONN_STATE_CONNECTED: tx_phy %d, rx_phy %d", tx_phy,
                           rx_phy);
#endif
    }
```

```
        }
        break;
}
```

3. 检查对端设备的 Features

成功建立 connection 后，蓝牙协议层会读取对端设备的 Features。GAP 层将通过 GAP_MSG_LE_REMOTE_FEATS_INFO 向 APP 通知对端设备的 Features，APP 可以检查对端设备是否支持 LE 2M PHY 或 LE Coded PHY。

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;
    switch (cb_type)
    {
#if F_BT_LE_5_0_SET_PHY_SUPPORT
        case GAP_MSG_LE_REMOTE_FEATS_INFO:
        {
            uint8_t remote_feats[8];
            APP_PRINT_INFO3("GAP_MSG_LE_REMOTE_FEATS_INFO: conn id %d, cause 0x%x,
                            remote_feats %b",
                            p_data->p_le_remote_feats_info->conn_id,
                            p_data->p_le_remote_feats_info->cause,
                            TRACE_BINARY(8, p_data->p_le_remote_feats_info->remote_feats));
            if (p_data->p_le_remote_feats_info->cause == GAP_SUCCESS)
            {
                memcpy(remote_feats, p_data->p_le_remote_feats_info->remote_feats, 8);
                if (remote_feats[LE_SUPPORT_FEATURES_MASK_ARRAY_INDEX1] &
                    LE_SUPPORT_FEATURES_LE_2M_MASK_BIT)
                {
                    APP_PRINT_INFO0("GAP_MSG_LE_REMOTE_FEATS_INFO: support 2M");
                }
                if (remote_feats[LE_SUPPORT_FEATURES_MASK_ARRAY_INDEX1] &
                    LE_SUPPORT_FEATURES_LE_CODED_PHY_MASK_BIT)
                {
                    APP_PRINT_INFO0("GAP_MSG_LE_REMOTE_FEATS_INFO: support CODED");
                }
            }
        }
        break;
#endif
    }
}
```

4. 切换 PHY

le_set_phy()用于设置 connection (由 conn_id 确定) 的 PHY preferences，而 Controller 有可能不能成功切换 PHY (例如对端设备不支持请求的 PHY)或者认为当前 PHY 更好。

```
static T_USER_CMD_PARSE_RESULT cmd_setphy(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    uint8_t conn_id = p_parse_value->dw_param[0];
    uint8_t all_phys;
    uint8_t tx_phys;
    uint8_t rx_phys;
    T_GAP_PHYS_OPTIONS phy_options = GAP_PHYS_OPTIONS_CODED_PREFER_S8;
    T_GAP_CAUSE cause;
    if (p_parse_value->dw_param[1] == 0)
    {
        all_phys = GAP_PHYS_PREFER_ALL;
        tx_phys = GAP_PHYS_PREFER_1M_BIT;
        rx_phys = GAP_PHYS_PREFER_1M_BIT;
    }
    else if (p_parse_value->dw_param[1] == 1)
    {
        all_phys = GAP_PHYS_PREFER_ALL;
        tx_phys = GAP_PHYS_PREFER_2M_BIT;
        rx_phys = GAP_PHYS_PREFER_2M_BIT;
    }
    .....
    cause = le_set_phy(conn_id, all_phys, tx_phys, rx_phys, phy_options);
    return (T_USER_CMD_PARSE_RESULT)cause;
}
```

5. PHY 的更新

GAP_MSG_LE_PHY_UPDATE_INFO 用于向 APP 通知 Controller 使用的发射机 PHY 或接收机 PHY 的更新结果。

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;
    switch (cb_type)
    {
#if F_BT_LE_5_0_SET_PHY_SUPPORT
        case GAP_MSG_LE_PHY_UPDATE_INFO:
            APP_PRINT_INFO4("GAP_MSG_LE_PHY_UPDATE_INFO:conn_id %d, cause 0x%x, rx_phy %d,
                            tx_phy %d",
                            p_data->p_le_phy_update_info->conn_id,
                            p_data->p_le_phy_update_info->cause,
```

```

    p_data->p_le_phy_update_info->rx_phy,
    p_data->p_le_phy_update_info->tx_phy);
break;
#endif
}
}

```

2.5.6 蓝牙协议栈相关特性设置

蓝牙协议栈相关特性的配置分为 LE Link 数目的配置和 API 的参数配置。

示例代码位于 **BLE Scatternet** 工程中。

2.5.6.1 LE Link 数目的配置

由于支持多链路以及 APP 的不同需求, 用户需要配置 LE Link 数目。本节以 **BLE scatternet** 工程为例, 介绍 LE Link 数目的配置。

假定 **BLE scatternet** 工程同时支持一条作为 master 角色的 link 和一条作为 slave 角色的 link, 配置方法如下所示。

1. 在 Config file 中配置 LE Link 数目

用户需要使用 MPTool 配置 Config file 中的 LE Link 数目, 可配置参数如下所示:

- Maximum LE Link Number: LE Link 数目, 最大值为 4
- Master Link Number: 作为 master 角色的 link 数目, 最大值为 4
- Slave Link Number: 作为 slave 角色的 link 数目, 最大值为 3

BLE scatternet 工程同时支持一条作为 master 角色的 link 和一条作为 slave 角色的 link。因此, Maximum LE Link Number 不能小于 2, Master Link Number 不能小于 1, Slave Link Number 不能小于 1。配置如图 2-13 所示。

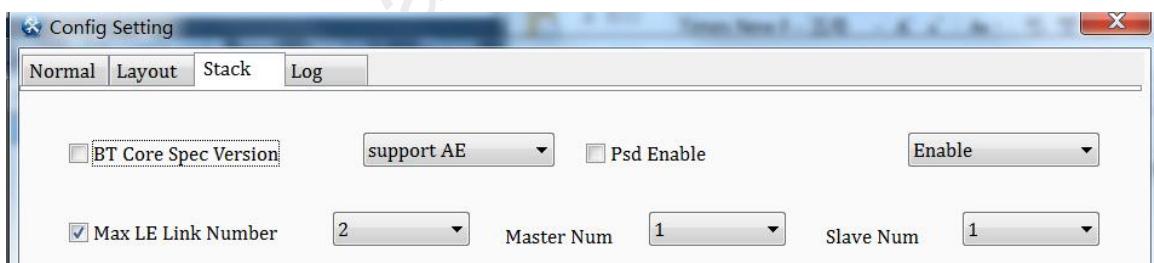


图 2-13 Config file 中的 LE Link 数目

2. 在 APP 中配置 LE 绑定设备数目的最大值

由于支持多链路, 设备需要存储多个对端设备的绑定信息。**BLE scatternet** 工程将 LE 绑定设备数目的最大值配置为 2, 更多关于配置的内容参见 [API 的参数配置](#)。

3. 在 APP 中初始化 LE Link 数目

如 [GAP 启动流程](#)所述, APP 调用 `le_gap_init()`以初始化 GAP 并设置 link 数目。`le_gap_init()`设置的 link 数目必须小于或等于 Config file 配置的 Maximum LE Link Number。若 `le_gap_init()`设置的 link 数目大于 Config file 配置的 Maximum LE Link Number, link 数目的初始化会失败, `le_gap_init()`的返回值为 false。

在 **BLE scatternet** 工程中, link 数目是通过宏定义 APP_MAX_LINKS 配置的。

```
int main(void)
{
    .....
    le_gap_init(APP_MAX_LINKS);
    .....
}
```

Config file 中 Maximum LE Link Number 被配置为 2, 因此头文件 app_flags.h 中的 APP_MAX_LINKS 被配置为 2。

```
/** @brief  Config APP LE link number */
#define APP_MAX_LINKS 2
```

2.5.6.2 API 的参数配置

APP 可以使用 gap_config.h 中的 API, 以配置蓝牙协议栈相关特性, 例如 LE 绑定设备数目的最大值和 CCCD 数目的最大值。APP 需要按照以下配置方法使用 API。

1. 在 otp_config.h 中增加宏定义

```
/*=====
*      upperstack configuration
=====*/
#define BT_STACK_CONFIG_ENABLE

#ifndef BT_STACK_CONFIG_ENABLE
void bt_stack_config_init(void);
#endif
```

2. 在 main.c 中配置蓝牙协议栈相关特性

LE 绑定设备数目最大值的默认值为 1。由于支持多链路, APP 使用 `gap_config_max_le_paired_device()` 配置 LE 绑定设备数目的最大值。

```
.....
#include <gap_config.h>
#include <otp_config.h>
.....

#ifndef BT_STACK_CONFIG_ENABLE
#include "app_section.h"

APP_FLASH_TEXT_SECTION void bt_stack_config_init(void)
{
```

```
    gap_config_max_le_paired_device(APP_MAX_LINKS);  
}  
#endif
```

2.5.7 请求配对

iOS 系统未提供启动 security 流程的接口。若 slave 设备希望与 iOS 设备配对，那么 slave 设备需要请求配对。

在 SDK 中提供两种本地设备启动 security 流程的方法，以及一种本地设备作为 GATT Server 请求 iOS 启动 security 流程的方法。

2.5.7.1 配置 GAP_PARAM_BOND_SEC_REQ_ENABLE

GAP_PARAM_BOND_SEC_REQ_ENABLE 参数决定在连线建立成功时是否启动 security 流程。若该参数设置为 true，当连线建立成功时，GAP 层将自动启动 security 流程。

```
void app_le_gap_init(void)  
{  
    .....  
    uint8_t auth_sec_req_enable = true;  
    uint16_t auth_sec_req_flags = GAP_AUTHEN_BIT_BONDING_FLAG;  
    le_bond_set_param(GAP_PARAM_BOND_SEC_REQ_ENABLE, sizeof(auth_sec_req_enable),  
    &auth_sec_req_enable);  
    le_bond_set_param(GAP_PARAM_BOND_SEC_REQ_REQUIREMENT, sizeof(auth_sec_req_flags),  
    &auth_sec_req_flags);  
    .....  
}
```

2.5.7.2 调用 le_bond_pair 函数

APP 可以调用 le_bond_pair() 以启动 security 流程。当 LE 链路状态为 GAP_CONN_STATE_CONNECTED 时，APP 才能调用 le_bond_pair()。

```
void app_handle_conn_state_evt(uint8_t conn_id, T_GAP_CONN_STATE new_state, uint16_t disc_cause)  
{  
    .....  
    switch (new_state)  
    {  
        .....  
        case GAP_CONN_STATE_CONNECTED:  
        {  
            le_bond_pair(conn_id);  
        }  
        break;  
    }
```

```

default:
    break;
}
}

```

2.5.7.3 Service 的 security 要求

GATT Profile 流程用于访问信息，该信息可能要求 client 是认证的或有加密的连线，才能对 characteristic 进行读或写操作。

若 service 有 security 要求，client 发送请求进行读或写操作时，物理链路是未认证或未加密的，那么 server 必须发送 Error Response。希望进行读或写操作的 client 使用 GAP authentication 流程，使物理链路经过认证，然后 client 可以发送请求进行读或写操作。

示例如图 2-14 所示，当 iOS 设备收到未认证或未加密的 Error Response，iOS 设备将启动 security 流程。

49	16:56:13.328 964 100	ATT Read (Client Characteristic Configuration; Notifications=? , Indications=? ..)	OK
49	16:56:13.328 964 100	ATT Read Transaction (Client Characteristic Configuration; Insufficient ...)	OK
49	16:56:13.328 964 100	ATT Read Request Packet (Client Characteristic Configuration)	OK
50	16:56:13.332 973 400	ATT Error Response Packet [Insufficient Authentication]	OK
51	16:56:13.369 071 800	SMP Pairing Feature Exchange (Keyboard Display, Bonding, MITM, SC > No...)	OK
51	16:56:13.369 071 800	SMP Pairing Request (Keyboard Display, Bonding, MITM, SC, Int=EncK...)	OK
52	16:56:13.376 591 600	SMP Pairing Response (No Input No Output, Bonding, SC, Int=IdKey, R...)	OK

图 2-14 ATT Insufficient Authentication

Attribute Element 是 service 的基本单元，其结构体定义在 gatt.h 中。

```

typedef struct
{
    uint16_t      flags;           /* < Attribute flags @ref GATT_ATTRIBUTE_FLAG */
    uint8_t       type_value[2 + 14]; /* < 16 bit UUID + included value or 128 bit UUID */
    uint16_t      value_len;       /* < Length of value */
    void         *p_value_context; /* < Pointer to value if @ref ATTRIB_FLAG_VALUE_INCL
                                    and @ref ATTRIB_FLAG_VALUE_APPL not set */
    uint32_t      permissions;     /* < Attribute permission @ref GATT_ATTRIBUTE_PERMISSIONS */
} T_ATTRIB_APPL;

```

Permissions 参数用于定义 attribute 的权限，更多信息参见 [Attribute Element](#)。

Characteristic 的认证要求示例代码如下所示，对其进行读或写操作时要求认证的链路。

```

/* client characteristic configuration .. 5*/
{
    (ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_CCCD_APPL),           /* wFlags */
    {                                                       /* bTypeValue */
        LO_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
        HI_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),

```

```

/* NOTE: this value has an instantiation for each client, a write to */
/* this attribute does not modify this default value: */
LO_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT), /* client char. config. bit field */
HI_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT)
},
2,                                     /* bValueLen */
NULL,
(GATT_PERM_READ_AUTHEN_REQ | GATT_PERM_WRITE_AUTHEN_REQ)           /* */
wPermissions */
},

```

Service 的 security 要求示例代码位于 **hids_kb.c** 中。

2.6 GAP 信息存储

在 `gap_storage_le.h` 中定义常量和函数原型。本地协议栈信息和绑定信息保存在 FTL 中，更多关于 FTL 的信息参见 [FTL 简介](#)。

2.6.1 FTL 简介

BT stack 和 user application 使用 FTL 作为抽象层保存或载入 flash 中的数据。

2.6.1.1 FTL 布局

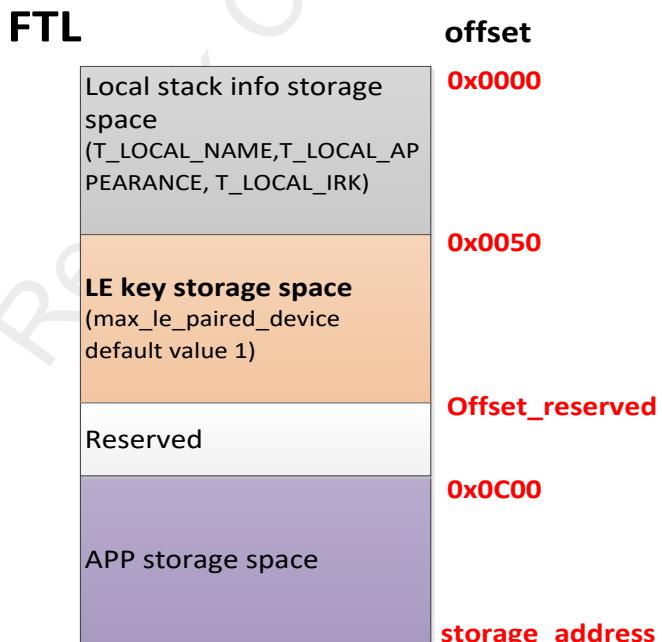


图 2-15 FTL 布局

FTL 可以分为以下四个区域：

1. Local stack information storage space
 - 1) 地址范围: 0x0000 - 0x004F
 - 2) 该区域用于存储本地协议栈信息，包括 device name、device appearance 和本地设备 IRK。更多信息参见[本地协议栈信息存储](#)。
2. LE key storage space
 - 1) 地址范围: 0x0050 - (Offset_reserved - 1)
 - 2) 该区域用于存储 LE 密钥信息，更多信息参见[绑定信息存储](#)。
 - 3) Offset_reserved 是可变值
$$\text{Offset_reserved} = 0x0064 + \text{max_le_paired_device} * \text{device_block_size}$$
 - 4) max_le_paired_device: 该参数表示存储绑定信息设备的最大数目，可以通过 gap_config_max_le_paired_device() 配置，默认值为 1，更多信息参见[API 的参数配置](#)。
3. Reserved space
 - 1) 地址范围: Offset_reserved - 0x0BFF
 - 2) 该区域为保留空间，因为 max_le_paired_device 参数是可配置的。LE key storage space 可以使用该区域存储密钥信息。
4. APP storage space
 - 1) 地址范围: 0x0C00 – storage_address, 更多信息参见关于 memory 的文档^[2]。
 - 2) APP 可以使用该区域存储信息。
 - 3) APP 可以调用 ftl_save() 和 ftl_load() 以访问该存储空间。

2.6.1.2 FTL APIs

APP 调用 ftl_save() 保存数据到 FTL。

APP 调用 ftl_load() 从 FTL 载入数据。

```
#define FTL_APP_START_ADDR (3 * 1024)  
static inline uint32_t ftl_save(void *pdata, uint16_t offset, uint16_t size)  
{  
    return ftl_save_to_storage(pdata, offset + FTL_APP_START_ADDR, size);  
}  
static inline uint32_t ftl_load(void *pdata, uint16_t offset, uint16_t size)  
{  
    return ftl_load_from_storage(pdata, offset + FTL_APP_START_ADDR, size);  
}
```

ftl_save() 从 FTL_APP_START_ADDR 开始写 FTL, ftl_load() 从 FTL_APP_START_ADDR 开始读 FTL。

当使用 ftl_save() 和 ftl_load() 时，偏移可以从 0 开始。示例代码如下所示：

```
/* Define start offset of the flash to save static random address. */  
#define APP_STATIC_RANDOM_ADDR_OFFSET 0
```

```
uint32_t app_save_static_random_address(T_APP_STATIC_RANDOM_ADDR *p_addr)
{
    APP_PRINT_INFO0("app_save_static_random_address");
    return ftl_save(p_addr, APP_STATIC_RANDOM_ADDR_OFFSET,
                    sizeof(T_APP_STATIC_RANDOM_ADDR));
}

uint32_t app_load_static_random_address(T_APP_STATIC_RANDOM_ADDR *p_addr)
{
    uint32_t result;
    result = ftl_load(p_addr, APP_STATIC_RANDOM_ADDR_OFFSET,
                      sizeof(T_APP_STATIC_RANDOM_ADDR));
    APP_PRINT_INFO1("app_load_static_random_address: result 0x%x", result);
    if (result)
    {
        memset(p_addr, 0, sizeof(T_APP_STATIC_RANDOM_ADDR));
    }
    return result;
}
```

2.6.2 本地协议栈信息存储

2.6.2.1 Device Name 存储

GAP 层目前支持的 device name 字符串的最大长度是 40 字节（包括结束符）。

flash_save_local_name() 用于保存 device name 到 FTL。

flash_load_local_name() 用于从 FTL 载入 device name。

若 GAP service 的 Device Name characteristic 是可写的，APP 可以调用该函数保存 device name。示例代码参见 [GAP Service Characteristic 的可写属性](#)。

2.6.2.2 Device Appearance 存储

Device Appearance 用于描述设备的类型，例如键盘、鼠标、温度计、血压计等。

flash_save_local_appearance() 用于保存 device appearance 到 FTL。

flash_load_local_appearance() 用于从 FTL 载入 device appearance。

若 GAP service 的 Device Appearance characteristic 是可写的，APP 可以调用该函数保存 device appearance。示例代码参见 [GAP Service Characteristic 的可写属性](#)。

2.6.2.3 本地设备 IRK 存储

IRK 是用于生成和解析 RPA 的 128 bit 密钥。

`flash_save_local_irk()`用于保存本地设备 IRK 到 FTL。

`flash_load_local_irk()`用于从 FTL 载入本地设备 IRK。

2.6.3 绑定信息存储

2.6.3.1 绑定设备优先级管理

GAP 层实现绑定设备优先级管理机制，其中优先级控制模块被保存在 FTL 中。LE 设备有存储空间和优先级控制模块。

优先级控制模块包括以下两部分：

- **bond_num**: 已保存绑定设备的数目
- **bond_idx** 数组: 已保存绑定设备的索引数组。GAP 层可以根据绑定设备索引查找到其在 FTL 中的起始偏移。

优先级管理包括如下操作：

1. 添加一个绑定设备

GAP LE API: 不对外提供，仅供内部使用。

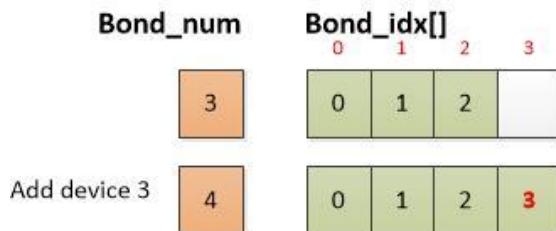


图 2-16 增加一个绑定设备

2. 移除一个绑定设备

GAP LE API: `le_bond_delete_by_idx()` 或 `le_bond_delete_by_bd()`

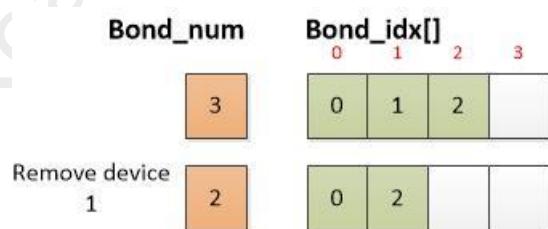


图 2-17 移除一个绑定设备

3. 清除所有绑定设备

GAP LE API: `le_bond_clear_all_keys()`

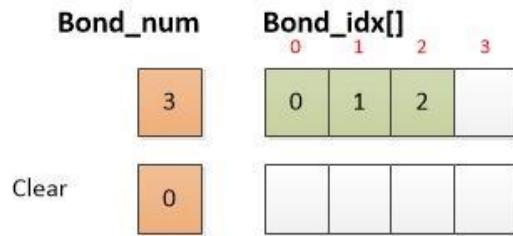


图 2-18 清除所有绑定设备

4. 将一个绑定设备设为最高优先级

GAP LE API: le_set_high_priority_bond()

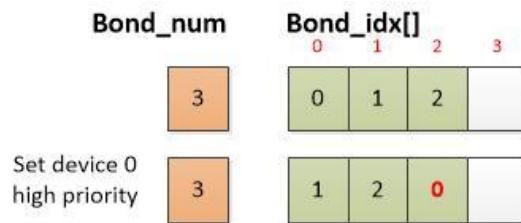


图 2-19 将一个绑定设备设为最高优先级

5. 获取最高优先级设备

最高优先级设备为 `bond_idx[bond_num - 1]`。

GAP LE API: le_get_high_priority_bond()

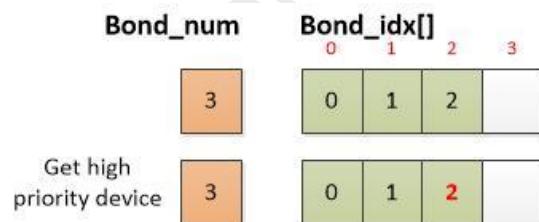


图 2-20 获取最高优先级设备

6. 获取最低优先级设备

最低优先级设备为 `bond_idx[0]`。

GAP LE API: le_get_low_priority_bond()

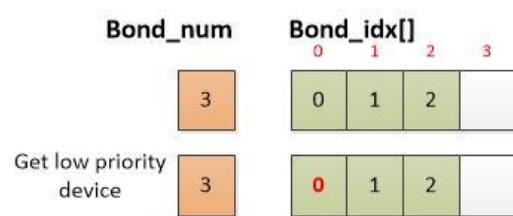


图 2-21 获取最低优先级设备

优先级管理示例如图 2-22 所示。

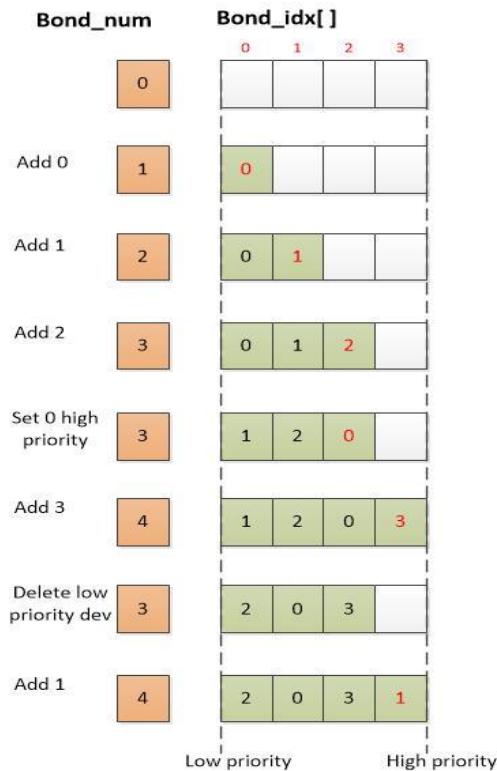


图 2-22 优先级管理示例

2.6.3.2 BLE 密钥存储

BLE 密钥信息存储在 LE key storage space, LE FTL 布局如图 2-23 所示。

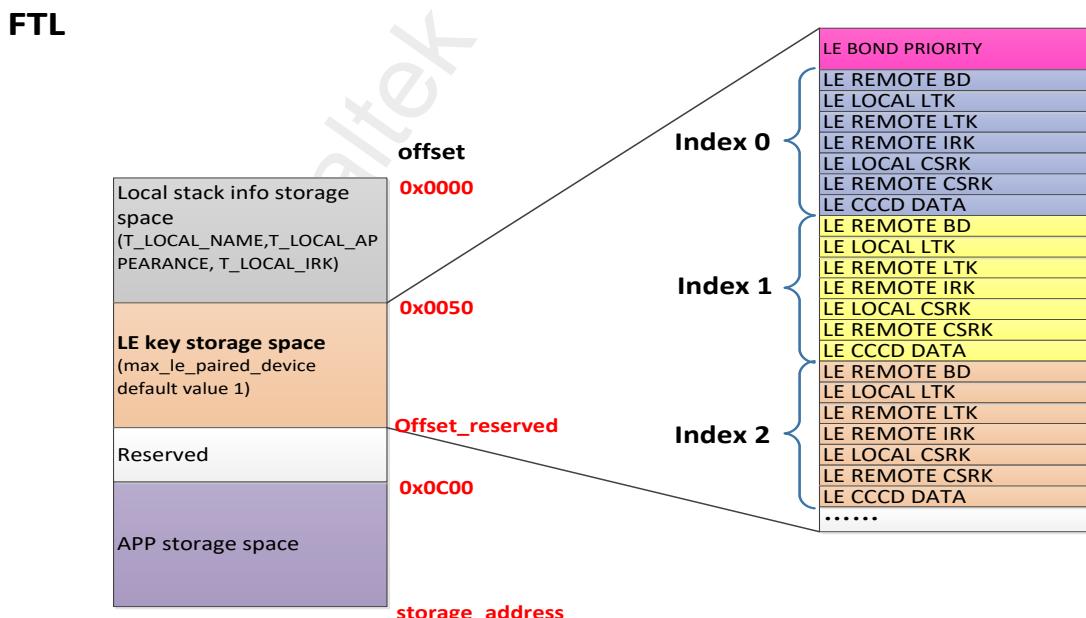


图 2-23 LE FTL 布局

LE key storage space 可以分为以下两部分：

1. **LE BOND PRIORITY:** LE 优先级控制模块，更多信息参见[绑定设备优先级管理](#)。
2. **Bonded device keys storage block:** 设备索引 0、索引 1 等等。
 - 1) LE REMOTE BD: 保存对端设备地址
 - 2) LE LOCAL LTK: 保存本地设备 Long Term Key (LTK)
 - 3) LE REMOTE LTK: 保存对端设备 LTK
 - 4) LE REMOTE IRK: 保存对端设备 IRK
 - 5) LE LOCAL CSRK: 保存本地设备 Connection Signature Resolving Key (CSRK)
 - 6) LE REMOTE CSRK: 保存对端设备 CSRK
 - 7) LE CCCD DATA: 保存 Client Characteristic Configuration declaration (CCCD)数据

2.6.3.2.1 配置

LE key storage space 的大小与以下两个参数有关：

1. LE 绑定设备数目的最大值
 - 1) 默认值为 1
 - 2) 可以使用 `gap_config_max_le_paired_device()` 配置，更多信息参见[API 的参数配置](#)
2. CCCD 数目的最大值
 - 1) 默认值为 16
 - 2) 可以使用 `gap_config_ccc_bits_count()` 配置，更多信息参见[API 的参数配置](#)

2.6.3.2.2 LE Key Entry 结构体

GAP 层使用结构体 `T_LE_KEY_ENTRY` 管理绑定设备。

```
#define LE_KEY_STORE_REMOTE_BD_BIT 0x01
#define LE_KEY_STORE_LOCAL_LTK_BIT 0x02
#define LE_KEY_STORE_REMOTE_LTK_BIT 0x04
#define LE_KEY_STORE_REMOTE_IRK_BIT 0x08
#define LE_KEY_STORE_LOCAL_CSRK_BIT 0x10
#define LE_KEY_STORE_REMOTE_CSRK_BIT 0x20
#define LE_KEY_STORE_CCCD_DATA_BIT 0x40
#define LE_KEY_STORE_LOCAL_IRK_BIT 0x80

/** @brief LE key entry */
typedef struct
{
    bool is_used;
    uint8_t idx;
    uint16_t flags;
    uint8_t local_bd_type;
    uint8_t app_data;
    uint8_t reserved[2];
    T_LE_REMOTE_BD remote_bd;
    T_LE_REMOTE_BD resolved_remote_bd;
```

```
} T_LE_KEY_ENTRY;
```

参数描述：

- **is_used** - 是否使用
- **idx** - 设备索引，GAP 使用 idx 查找绑定设备信息在 FTL 的存储位置
- **flags** - LE Key Storage Bits，表示密钥是否存在位域
- **local_bd_type** - 配对过程中使用的 local address type, T_GAP_LOCAL_ADDR_TYPE
- **remote_bd** - 对端设备地址
- **resolved_remote_bd** - 对端设备的 identity address

2.6.3.2.3 LE 密钥管理

当本地设备与对端设备配对或本地设备与绑定设备加密时，GAP 层将发送 GAP_MSG_LE_AUTHEN_STATE_CHANGE 消息向 APP 通知 authentication 状态的变化。

```
void app_handle_authen_state_evt(uint8_t conn_id, uint8_t new_state, uint16_t
                                 cause)
{
    APP_PRINT_INFO2("app_handle_authen_state_evt:conn_id %d, cause 0x%x", conn_id, cause);
    switch (new_state)
    {
        case GAP_AUTHEN_STATE_STARTED:
        {
            APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_STARTED");
        }
        break;
        case GAP_AUTHEN_STATE_COMPLETE:
        {
            if (cause == GAP_SUCCESS)
            {
                APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_COMPLETE
                                pair success");
            }
            else
            {
                APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_COMPLETE
                                pair failed");
            }
        }
        break;
        .....
    }
}
```

GAP_MSG_LE_BOND MODIFY_INFO 用于向 APP 通知绑定信息已变更。

```
typedef struct
{
    T_LE_BOND MODIFY_TYPE type;
    P_LE_KEY_ENTRY p_entry;
} T LE_BOND MODIFY_INFO;

T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;
    switch (cb_type)
    {
        case GAP_MSG LE_BOND MODIFY_INFO:
            APP_PRINT_INFO1("GAP_MSG LE_BOND MODIFY_INFO: type 0x%x",
                            p_data->p_le_bond_modify_info->type);
            break;
        .....
    }
}
```

绑定信息变更类型的定义如下所示：

```
typedef enum {
    LE_BOND_DELETE,
    LE_BOND_ADD,
    LE_BOND_CLEAR,
    LE_BOND_FULL,
    LE_BOND_KEY_MISSING,
} T LE_BOND MODIFY_TYPE;
```

1. LE_BOND_DELETE

LE_BOND_DELETE 表示绑定信息已被删除，当满足以下条件时会发送该消息：

- 1) 调用 le_bond_delete_by_idx()
- 2) 调用 le_bond_delete_by_bd()
- 3) 链路加密失败
- 4) 密钥存储空间已满，因此最低优先级的绑定信息会被删除。

2. LE_BOND_ADD

LE_BOND_ADD 表示新增绑定设备，仅在第一次与对端设备配对时发送该消息。

3. LE_BOND_CLEAR

LE_BOND_CLEAR 表示所有绑定消息已被删除，在调用 le_bond_clear_all_keys()后会发送该消息。

4. LE_BOND_FULL

LE_BOND_FULL 表示密钥存储空间已满，当参数 GAP_PARAM_BOND_KEY_MANAGER 设为 true 时发送该消息。此时，GAP 层不会自动删除绑定信息。设为 false 时不会发送该消息，GAP 层将删除最低

优先级的绑定信息，存储当前的绑定信息，然后发送 LE_BOND_DELETE 消息。

5. LE_BOND_KEY_MISSING

LE_BOND_KEY_MISSING 表示链路加密失败且密钥失效，当参数 GAP_PARAM_BOND_KEY_MANAGER 设为 true 时发送该消息。此时，GAP 层不会自动删除绑定信息。设为 false 时不会发送该消息，GAP 层将删除绑定信息，发送 LE_BOND_DELETE 消息。

2.6.3.2.4 GAP 层内部 BLE 设备优先级管理

1. 与新设备配对

1) 密钥存储空间未满

(1) GAP 层将在优先级控制模块添加绑定设备，发送 LE_BOND_ADD 消息给 APP。该新增设备具有最高优先级。

2) 密钥存储空间已满

(1) 当 GAP_PARAM_BOND_KEY_MANAGER 为 true 时，GAP 层发送消息 LE_BOND_FULL 给 APP。

(2) 当 GAP_PARAM_BOND_KEY_MANAGER 为 false 时，GAP 层将从优先级控制模块移除最低优先级的绑定设备，发送 LE_BOND_DELETE 消息。GAP 层将在优先级控制模块添加绑定设备，发送 LE_BOND_ADD 消息给 APP。该新增设备具有最高优先级。

2. 与绑定设备加密成功

GAP 层将该绑定设备设为最高优先级。

3. 与绑定设备加密失败

1) 当 GAP_PARAM_BOND_KEY_MANAGER 为 true 时，GAP 层将发送 LE_BOND_KEY_MISSING 给 APP。

2) 当 GAP_PARAM_BOND_KEY_MANAGER 为 false 时，GAP 层将从优先级控制模块移除该绑定设备，发送 LE_BOND_DELETE 给 APP。

2.6.3.2.5 APIs

```
/* gap_storage_le.h */
P_LE_KEY_ENTRY le_find_key_entry(uint8_t *bd_addr, T_GAP_REMOTE_ADDR_TYPE bd_type);
P_LE_KEY_ENTRY le_find_key_entry_by_idx(uint8_t idx);
uint8_t le_get_bond_dev_num(void);
P_LE_KEY_ENTRY le_get_low_priority_bond(void);
P_LE_KEY_ENTRY le_get_high_priority_bond(void);
bool le_set_high_priority_bond(uint8_t *bd_addr, T_GAP_REMOTE_ADDR_TYPE bd_type);
bool le_resolve_random_address(uint8_t *unresolved_addr, uint8_t *resolved_addr,
                               T_GAP_IDENT_ADDR_TYPE *resolved_addr_type);
bool le_get_cccd_data(T_LE_KEY_ENTRY *p_entry, T_LE_CCCD *p_data);
bool le_gen_bond_dev(uint8_t *bd_addr, T_GAP_REMOTE_ADDR_TYPE bd_type,
                     T_GAP_LOCAL_ADDR_TYPE local_bd_type,
```

```
uint8_t ltk_length, uint8_t *local_ltk, T_LE_KEY_TYPE  
key_type, T_LE_CCCD *p_cccd);  
bool le_set_privacy_info(T_LE_KEY_ENTRY *p_entry, T_LE_PRIVACY_INFO *p_privacy_info);  
bool le_get_privacy_info(T_LE_KEY_ENTRY *p_entry, T_LE_PRIVACY_INFO *p_privacy_info);  
bool le_check_privacy_bond(T_LE_KEY_ENTRY *p_entry);  
  
/* gap_bond_le.h */  
void le_bond_clear_all_keys(void);  
T_GAP_CAUSE le_bond_delete_by_idx(uint8_t idx);  
T_GAP_CAUSE le_bond_delete_by_bd(uint8_t *bd_addr, T_GAP_REMOTE_ADDR_TYPE bd_type);
```

2.7 BLE Privacy

2.7.1 Specification 介绍

BLE 支持 privacy 特性，通过修改蓝牙设备地址，降低一段时间后追踪 LE 设备的可能性。

以便使用 privacy 特性的设备可以与已知设备回连，该 device address，也称为 private address 一定是可以被其它设备解析的。private address 是使用绑定流程中交换的设备 IRK 生成的。

2.7.1.1 Device Address

设备是使用 device address 来识别的。device address 可能是 public device address 或 random device address。

random device address 有以下两种子类型：

- Static address
- Private address

private address 有以下两种子类型：

- Non-resolvable private address
- Resolvable private address

2.7.1.1.1 Static address

static address 的格式如图 2-24 所示。

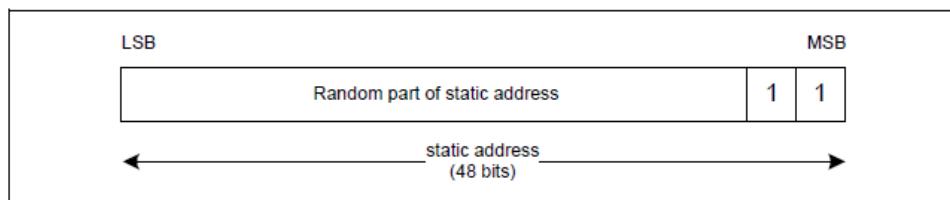


图 2-24 static address 的格式

2.7.1.1.2 Non-resolvable private address

non-resolvable private address 的格式如图 2-25 所示。

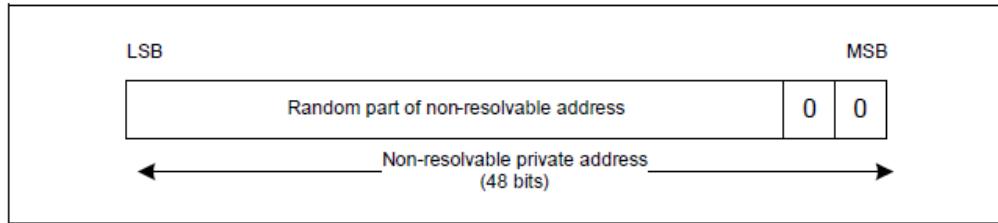


图 2-25 non-resolvable private address 的格式

2.7.1.1.3 Resolvable private address

resolvable private address 的格式如图 2-26 所示。

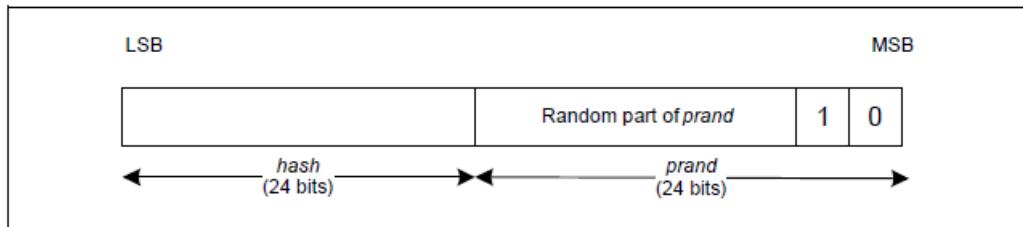


图 2-26 resolvable private address 的格式

设备的 Identity Address 是设备在发送的数据包中使用的 Public Device Address 或 Random Static Device Address。若设备使用 Resolvable Private Address，该设备必须有一个 Identity Address。

为了生成一个 resolvable private address，设备一定要有 Local Identity Resolving Key (IRK) 或 Peer Identity Resolving Key (IRK)。resolvable private address 必须由 IRK 和随机生成的 24-bit 数生成。

resolvable private address 可以被相应设备的 IRK 解析。若 resolvable private address 已被解析，设备可以关联该地址与对端设备。

2.7.1.2 IRK 和 Identity Address

Security Manager (SM) 使用密钥分发方法，以在无线通信中实现身份认证和加密功能。IRK 是用于生成和解析 resolvable private address 的 128 位密钥。若 master 已收到 slave 的 IRK，那么 master 可以解析 slave 的 resolvable private address。若 slave 已收到 master 的 IRK，那么 slave 可以解析 master 的 resolvable private address。privacy 概念仅防范未被分发 IRK 的设备。

Identity Information 用于分发 IRK。全为零的 Identity Resolving Key 数据字段表示设备没有有效的 resolvable private address。

Identity Address Information 用于分发 Public Device Address 或 Random Static Device Address。图 2-27 展示 Master 和 Slave 分发所有密钥和值的示例。

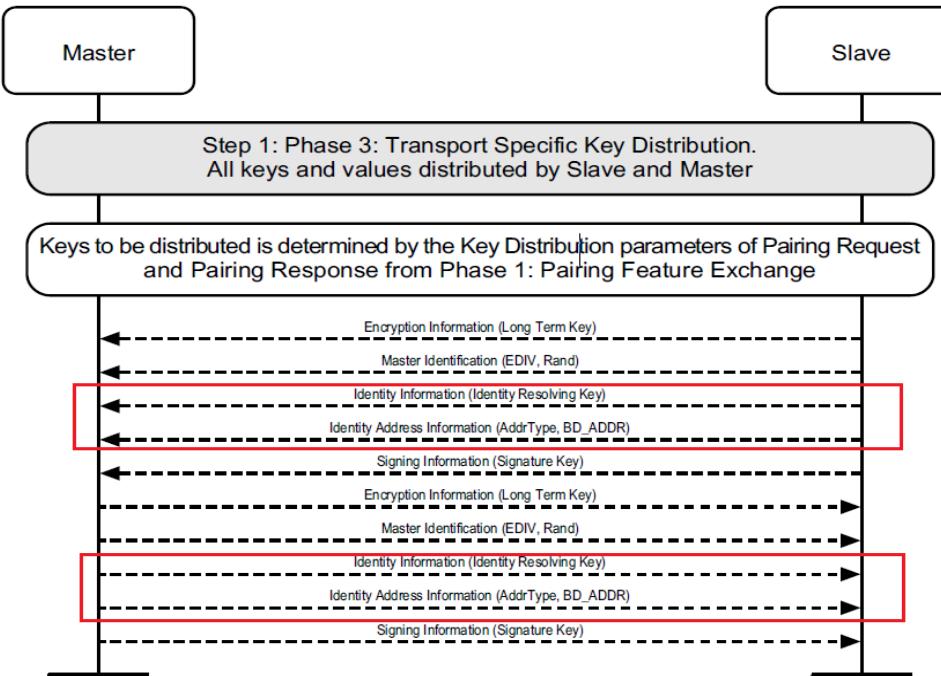


图 2-27 Transport Specific Key Distribution

2.7.1.3 Resolving List 和 Resolution

2.7.1.3.1 Resolving List 和 White List

通过增加和删除 device identity，Host 维护一个 resolving list。一个 device identity 由对端设备的 identity address 以及本地设备和对端设备的 IRK 对组成。

当 Controller 执行 address resolution，若 Host 的操作涉及 resolving list 中的一个对端设备，那么 Host 使用对端设备的 identity address。同样地，倘若对端的设备地址已被解析，所有从 Controller 送往 Host 的 event 将使用对端的 device identity。

在 Controller 中执行 address resolution 时，可以实现设备过滤。因为在检查设备是否存在与 White List 之前，对端的 device identity address 可以被解析出来。

图 2-28 展示 Controller resolving list 和 Controller white list 之间关系的逻辑表示。

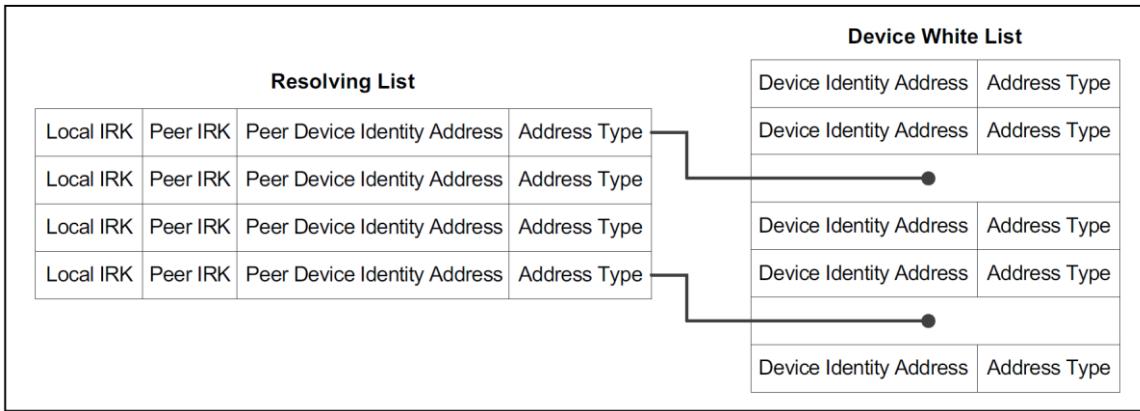


图 2-28 Resolving List 和 Device White List 的逻辑表示

一旦 resolvable private address 被解析出来, Host 设置的 White List 和 filter policies 应用于相应的 identity address。

2.7.1.3.2 Address Resolution

若 Controller 中的 address resolution 已启用, 当 Controller 收到本地或对端的 resolvable private address, Controller 将使用 resolving list。

除了以下场景, 其它时间均可启用 address resolution:

- 已启用 advertising
- 已启用 scanning
- 正在创建连线

当 Controller 中的 address resolution 已启用, 从 Host 至 Controller, 涉及 resolving list 中对端设备的操作, Host 必须使用对端设备的 identity address。同样地, 倘若对端的设备地址已被解析, 所有从 Controller 送往 Host 的 event 将使用对端的 device identity。

2.7.2 Privacy 管理模块

privacy 管理模块的开发是基于头文件 gap_privacy.h 的。若用户需要使用 privacy 管理模块, 那么用户不能使用头文件 gap_privacy.h。因而, 用户可以使用头文件 privacy_mgnt.h 用于开发 privacy 相关应用。

privacy 管理模块的路径如下所示:

- 源文件: sdk\src\ble\privacy\privacy_mgnt.c
- 头文件: sdk\src\ble\privacy\privacy_mgnt.h

用户需要在工程中添加相关文件, 如图 2-29 所示。

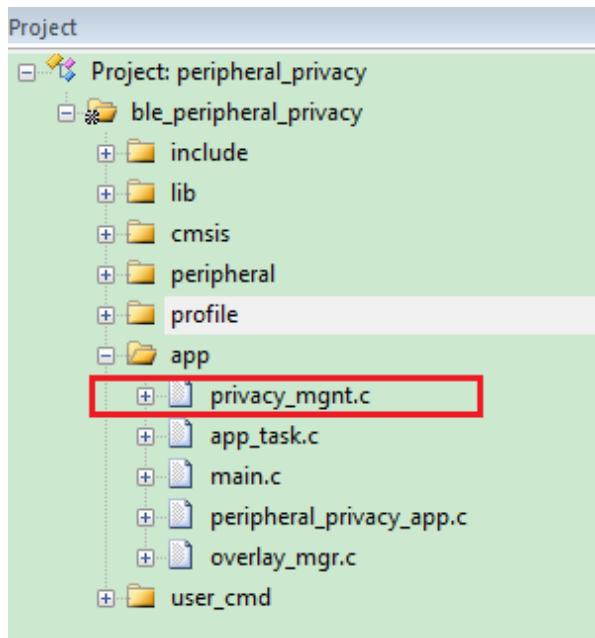


图 2-29 在工程中增加 privacy 管理模块

2.7.2.1 APIs

在头文件 `privacy_mgnt.h` 中定义 privacy 管理模块的接口。

```
/* privacy_mgnt.h APIs */
typedef void(*P_FUN_PRIVACY_STATE_CB)(T_PRIVACY_CB_TYPE type, T_PRIVACY_CB_DATA cb_data);
void privacy_init(P_FUN_PRIVACY_STATE_CB p_fun, bool whitelist);
T_PRIVACY_STATE privacy_handle_resolv_list(void);
void privacy_handle_bond_modify_msg(T_LE_BOND MODIFY_TYPE type, T_LE_KEY_ENTRY *p_entry,
                                     bool handle_add);
bool privacy_add_device(T_LE_KEY_ENTRY *p_entry);
T_GAP_CAUSE privacy_set_addr_resolution(bool enable);
T_GAP_CAUSE privacy_read_peer_resolv_addr(T_GAP_REMOTE_ADDR_TYPE peer_address_type,
                                           uint8_t *peer_address);
T_GAP_CAUSE privacy_read_local_resolv_addr(T_GAP_REMOTE_ADDR_TYPE peer_address_type,
                                            uint8_t *peer_address);
```

2.7.2.2 Privacy 管理模块的使用方法

示例代码位于 **BLE peripheral privacy** 工程中。

本节介绍如何使用 privacy 管理模块。

2.7.2.2.1 初始化

APP 需要调用 `privacy_init()`, 以初始化 privacy 管理模块并注册回调函数。

```
void app_le_gap_init(void)
```

```
{
.....
/* register gap message callback */
le_register_app_cb(app_gap_callback);
#if APP_PRIVACY_EN
    privacy_init(app_privacy_callback, true);
#endif
}
```

privacy_init()的参数 whitelist 用于配置对 white list 的管理。若 whitelist 参数为 true，当 privacy 管理模块修改 resolving list 时，privacy 管理模块会管理 white list。若 whitelist 参数为 false，APP 需要管理 white list。

回调函数用于处理 privacy 管理模块发送的消息。

```
void app_privacy_callback(T_PRIVACY_CB_TYPE type, T_PRIVACY_CB_DATA cb_data)
{
    APP_PRINT_INFO1("app_privacy_callback: type %d", type);
    switch (type)
    {
        case PRIVACY_STATE_MSGTYPE:
            app_privacy_state = cb_data.privacy_state;
            APP_PRINT_INFO1("PRIVACY_STATE_MSGTYPE: status %d", app_privacy_state);
            break;
        case PRIVACY_RESOLUTION_STATUS_MSGTYPE:
            app_privacy_resolution_state = cb_data.resolution_state;
            APP_PRINT_INFO1("PRIVACY_RESOLUTION_STATUS_MSGTYPE: status %d",
app_privacy_resolution_state);
            break;
        default:
            break;
    }
}
```

2.7.2.2 Resolving List 的管理

privacy 管理模块的关键功能是当绑定信息变化时，用于管理 resolving list 和 white list。white list 的管理是可选特性，privacy_init()的参数 whitelist 用于配置对 white list 的管理。

当 APP 处理 GAP_MSG_LE_BOND MODIFY_INFO 消息时，APP 必须调用函数 privacy_handle_bond_modify_msg()。

privacy 管理模块将根据绑定信息的修改类型管理 resolving list，流程如下所示：

- LE_BOND_DELETE: 在 resolving list 删除该设备
- LE_BOND_ADD: 在 resolving list 中加入该设备
- LE_BOND_CLEAR: 清除 resolving list

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
```

```

T_APP_RESULT result = APP_RESULT_SUCCESS;
T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;
switch (cb_type)
{
#if APP_PRIVACY_EN
    case GAP_MSG_LE_BOND MODIFY_INFO:
        APP_PRINT_INFO1("GAP_MSG LE BOND MODIFY_INFO: type 0x%x",
                        p_data->p_le_bond_modify_info->type);
        privacy_handle_bond_modify_msg(p_data->p_le_bond_modify_info->type,
                                        p_data->p_le_bond_modify_info->p_entry, true);
        break;
#endif
.....
}

```

当 address resolution 已启用，在以下场景，不能执行对 resolving list 的修改流程

- 已启用 advertising
- 已启用 scanning
- 正在创建连线

当 address resolution 未启用，resolving list 的修改流程可以在任何时候执行。

当 gap 设备状态为 idle 状态时，APP 必须调用函数 privacy_handle_resolv_list()。 privacy_handle_resolv_list()用于处理挂起的 resolving list 修改流程。

```

void app_handle_dev_state_evt(T_GAP_DEV_STATE new_state, uint16_t cause)
{
    APP_PRINT_INFO3("app_handle_dev_state_evt: init state %d, adv state %d, cause 0x%x",
                   new_state.gap_init_state, new_state.gap_adv_state, cause);
#if APP_PRIVACY_EN
    if ((new_state.gap_init_state == GAP_INIT_STATE_STACK_READY)
        && (new_state.gap_adv_state == GAP_ADV_STATE_IDLE)
        && (new_state.gap_conn_state == GAP_CONN_DEV_STATE_IDLE))
    {
        privacy_handle_resolv_list();
    }
#endif
.....
}

```

回调类型 PRIVACY_STATE_MSGTYPE 用于通知 resolving list 修改流程的状态。回调数据为 T_PRIVACY_STATE。

```

typedef enum
{
    PRIVACY_STATE_INIT, //!< Privacy management module is not initialization.
    PRIVACY_STATE_IDLE, //!< Idle. No pending resolving list modification procedure.
    PRIVACY_STATE_BUSY //!< Busy. Resolving list modification procedure is not completed.
}

```

{ T_PRIVACY_STATE;

2.7.2.2.3 Address Resolution

若对端设备使用 resolvable private address, APP 希望使用 white list 过滤该对端设备。APP 需要调用函数 `privacy_set_addr_resolution()`以启用 address resolution。

若 APP 希望与新的设备配对, APP 需要调用函数 `privacy_set_addr_resolution()`以禁用 address resolution。当 address resolution 已启用, 本地设备不能与不属于 resolving list 的设备建立连线。

示例代码如下所示。

```
void app_adv_start(void)
{
    uint8_t adv_evt_type = GAP_ADTYPE_ADV_IND;
    uint8_t adv_filter_policy = GAP_ADV_FILTER_ANY;
#if APP_PRIVACY_EN
    T_LE_KEY_ENTRY *p_entry;
    p_entry = le_get_high_priority_bond();

    if (p_entry == NULL)
    {
        /* No bonded device, send connectable undirected advertisement without using whitelist*/
        app_work_mode = APP_PAIRABLE_MODE;
        adv_filter_policy = GAP_ADV_FILTER_ANY;
        if (app_privacy_resolution_state == PRIVACY_ADDR_RESOLUTION_ENABLED)
        {
            privacy_set_addr_resolution(false);
        }
    }
    else
    {
        app_work_mode = APP_RECONNECTION_MODE;
        adv_filter_policy = GAP_ADV_FILTER_WHITE_LIST_ALL;
        if (app_privacy_resolution_state == PRIVACY_ADDR_RESOLUTION_DISABLED)
        {
            privacy_set_addr_resolution(true);
        }
    }
#endif
    le_adv_set_param(GAP_PARAM_ADV_EVENT_TYPE, sizeof(adv_evt_type), &adv_evt_type);
    le_adv_set_param(GAP_PARAM_ADV_FILTER_POLICY, sizeof(adv_filter_policy), &adv_filter_policy);
    le_adv_start();
}
```

函数 `privacy_set_addr_resolution()`用于启用 Controller 中对 resolvable private address 的解析功能。这将使得 Controller 收到本地或对端的 resolvable private address 时, Controller 将使用 resolving list。

除了以下场景，其它时间均可启用调用该函数：

- 已启用 advertising
- 已启用 scanning
- 正在创建连线

回调类型 PRIVACY_RESOLUTION_STATUS_MSGTYPE 用于通知 address resolution 的状态。回调数据为 T_PRIVACY_ADDR_RESOLUTION_STATE。

```
/** @brief Define the privacy address resolution state */

typedef enum
{
    PRIVACY_ADDR_RESOLUTION_DISABLED,
    PRIVACY_ADDR_RESOLUTION_DISABLING,
    PRIVACY_ADDR_RESOLUTION_ENABLING,
    PRIVACY_ADDR_RESOLUTION_ENABLED
} T_PRIVACY_ADDR_RESOLUTION_STATE;
```

3 GATT Profile

在 SDK 中提供基于 GATT specification 的 GATT Profile APIs。 基于 GATT 的 Profile 的实现分为两种类型：Profile-Server 和 Profile-Client。

Profile-Server 是基于 GATT 的 Profile 在 server 端的实现的公共接口，更多信息参见 [BLE Profile Server](#)。

Profile-Client 是基于 GATT 的 Profile 在 client 端的实现的公共接口，更多信息参见 [BLE Profile Client](#)。

GATT Profile Layer 已经在 ROM 中实现，在 SDK 中通过头文件提供接口给 APP 使用。GATT Profile 头文件路径为 sdk\inc\bluetooth\profile。

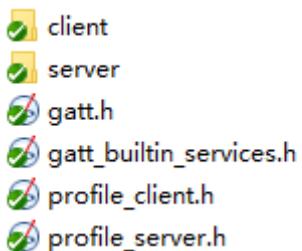


图 3-1 GATT Profile 头文件

3.1 BLE Profile Server

3.1.1 概述

Server 是可以接收来自 client 的 command 和 request 且发送 response、indication 和 notification 给 client 的设备。GATT Profile 定义作为 GATT server 和 GATT client 的 BLE 设备的交互方式。Profile 可能包含一个或多个 GATT services，service 是一组 characteristics 的集合，因而 GATT server 展示的是其 characteristics。

用户可以使用 Profile Server 导出的 APIs 实现 specific service。Profile server 层级如图 3-2 所示。Profile 包括 profile server layer 和 specific service。位于 protocol stack 之上的 profile server layer 封装供 specific service 访问 protocol stack 的接口，因此针对 specific service 的开发不涉及 protocol stack 的细节，使开发变得更简单和清晰。基于 profile server layer 的 specific service 是由 application layer 实现的，specific service 由 attribute value 组成并提供接口供 APP 发送数据。

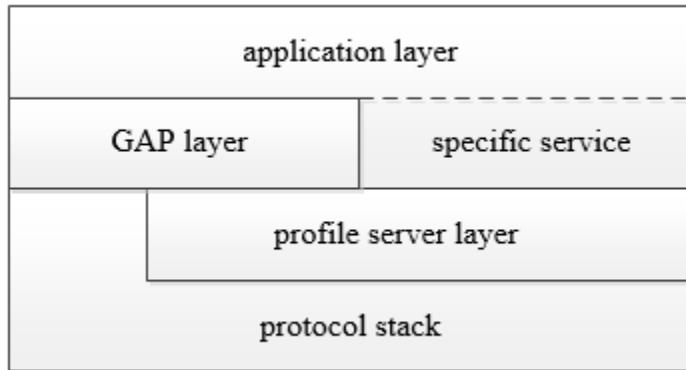


图 3-2 Profile Server 层级

3.1.2 支持的 Profile 和 Service

支持的 Profile 列表如表 3-1 所示。

表 3-1 支持的 Profile 列表

Abbr.	Definition	GATT server	GATT client
GAP	Generic Access Profile	Server role shall support GAS(M)	client role has no claim
PXP	Proximity Profile	Proximity Reporter role shall support LLS(M), IAS(O), TPS(O)	Proximity Monitor role has no claim
ScPP	Scan Parameters Profile	Scan Server role shall support ScPS(M)	Scan Client role has no claim
HTP	Health Thermometer Profile	Thermometer role shall support HTS(M), DIS(M)	Collector role has no claim
HRP	Heart Rate Profile	Heart Rate Sensor role shall support HRS(M), DIS(M)	Collector role has no claim
LNP	Location and Navigation Profile	LN Sensor role shall support LNS(M), DIS(O), BAS(O)	Collector role has no claim
WSP	Weight Scale Profile	Weight Scale role shall support WSS(M), DIS(M), BAS(O)	Weight Scale role shall support WSS(M), DIS(M), BAS(O)
GLP	Glucose Profile	Glucose Sensor role shall support GLS(M), DIS(M)	Collector role has no claim
FMP	Fine Me Profile	Find Me Target role shall support IAS(M)	Find Me Locator role has no claim
HOGP	HID over GATT Profile	HID Device shall support HIDS(M), BAS(M), DIS(O), ScPS(O)	Boot Host has no claim
RSCP	Running Speed and Cadence	RSC Sensor role shall support RSCS(M),	Collector role has no claim

	Profile	DIS(M)	
CSCP	Cycling Speed and Cadence Profile	CSC Sensor role shall support CSCS(M), DIS(M)	Collector role has no claim
IPSP	Internet Protocol Support Profile	Node role shall support IPSS(M)	Router role has no claim
NOTE:			
M: mandatory			
O: optional			

支持的 service 列表如表 3-2 所示。

表 3-2 支持的 service 列表

Abbr.	Definition	Files
GATTS	Generic Attribute Service	gatt_builtin_services.h
GAS	Generic Access Service	gatt_builtin_services.h
BAS	Battery Service	bas.c, bas.h bas_config.h
DIS	Device Information Service	dis.c, dis.h dis_config.h
ScPS	Scan Parameters Service	sps.c, sps.h sps_config.h
HIDS	Human Interface Device Service	hids.c, hids.h hids_kb.c, hids_kb.h hids_ms.c, hids_ms.h
TPS	Tx Power Service	tps.c, tps.h
IAS	Immediate Alert Service	ias.c, ias.h
LLS	Link Loss Service	lls.c, lls.h
HTS	Health Thermometer Service	hts.c, hts.h
HRS	Heart Rate Service	hrs.c, hrs.h
LNS	Location and Navigation Service	lns.c, lns.h
WSS	Weight Scale Service	wss.c, wss.h
RSCS	Running Speed and Cadence Service	rscs.c, rscs.h
CSCS	Cycling Speed and Cadence Service	cscs.c, cscs.h cscs_config.h
GLS	Glucose Service	gls.c, gls.h gls_config.h
IPSS	Internet Protocol Support Service	ipss.c, ipss.h

3.1.3 Profile Server 交互

Profile server layer 处理与 protocol stack layer 的交互，并提供接口用于设计 specific service。Profile Server 交互包括向 server 添加 service、读取 characteristic value、写入 characteristic value、characteristic value notification 和 characteristic value indication。

3.1.3.1 添加 Service

Protocol stack 维护通过 profile server layer 添加的所有 services 的信息。首先，需要通过 server_init() 接口初始化 service attribute table 的总数目。

```
void app_le_profile_init(void)
{
    server_init(2);
    simp_srv_id = simp_ble_service_add_service(app_profile_callback);
    bas_srv_id = bas_add_service(app_profile_callback);
    server_register_app_cb(app_profile_callback);
    ...
}
```

Profile server layer 提供 server_add_service() 接口用于向 profile server layer 添加 service。

```
T_SERVER_ID simp_ble_service_add_service(void *p_func)
{
    if (false == server_add_service(&simp_service_id,
                                    (uint8_t *)simple_ble_service_tbl,
                                    sizeof(simple_ble_service_tbl),
                                    simp_ble_service_cbs))
    {
        APP_PRINT_ERROR0("simp_ble_service_add_service: fail");
        simp_service_id = 0xff;
        return simp_service_id;
    }
    pfn_simp_ble_service_cb = (P_FUN_SERVER_GENERAL_CB)p_func;
    return simp_service_id;
}
```

图 3-3 表示一个 server 包含多个 service tables，向该 server 添加 service 的情况。

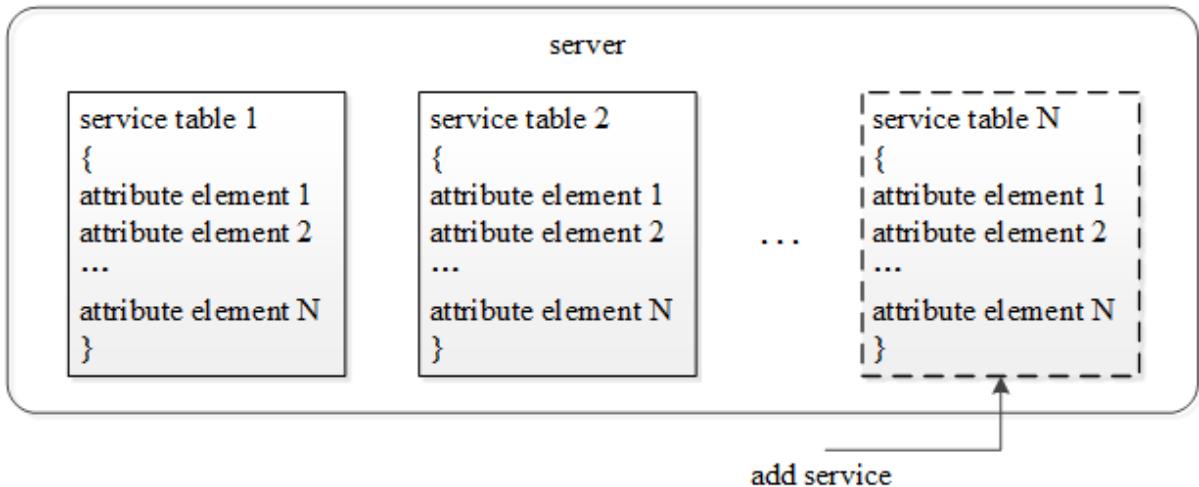


图 3-3 向 Server 添加 Services

当 service 添加到 profile server layer 后, GAP 初始化流程会注册所有 services, 一旦完成注册流程, GAP 层会发送 PROFILE_EVT_SRV_REG_COMPLETE 消息。

注册 service 的流程如图 3-4 所示。

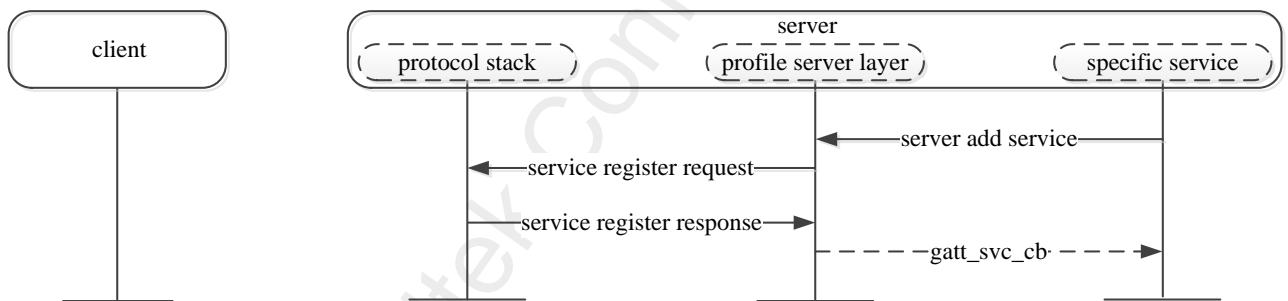


图 3-4 注册 Service 的流程

在 GAP 初始化过程中, 通过向 protocol stack 发送 service 注册请求以启动 service 的注册流程, 然后注册所有已添加 services。若 server 通用回调函数不为 NULL, 一旦最后一个服务被成功注册, profile server layer 将通过注册的回调函数 app_profile_callback() 向 APP 发送 PROFILE_EVT_SRV_REG_COMPLETE 消息。

```

T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    if (service_id == SERVICE_PROFILE_GENERAL_ID)
    {
        T_SERVER_APP_CB_DATA *p_param = (T_SERVER_APP_CB_DATA *)p_data;
        switch (p_param->eventId)
        {
    
```

```

case PROFILE_EVT_SRV_REG_COMPLETE:// srv register result event.
    APP_PRINT_INFO1("PROFILE_EVT_SRV_REG_COMPLETE: result %d",
                    p_param->event_data.service_reg_result);
    break;
...
}

```

3.1.3.2 Service 的回调函数

3.1.3.2.1 Server 通用回调函数

Server 通用回调函数用于向 APP 发送事件，其中包含 service 注册完成事件和通过 characteristic value notification 或 indication 发送数据完成事件。通过 server_register_app_cb() 初始化该回调函数。在 profile_server.h 中定义 server 通用回调函数。

3.1.3.2.2 Specific Service 回调函数

为访问由 specific service 提供的 attribute value，需要在 specific service 中实现回调函数，该回调函数用于处理来自 client 的 read/write attribute value 和更新 CCCD 数值的流程。通过 server_add_service() 初始化该回调函数。在 profile_server.h 中定义回调函数的结构体。

```

/* service related callback functions struct */
typedef struct {
    P_FUN_GATT_READ_ATTR_CB read_attr_cb;           // Read callback function pointer
    P_FUN_GATT_WRITE_ATTR_CB write_attr_cb;          // Write callback function pointer
    P_FUN_GATT_CCCD_UPDATE_CB cccd_update_cb;        // update cccd callback function pointer
} T_FUN_GATT_SERVICE_CBS;

```

read_attr_cb: 读 attribute 回调函数，当 client 发送 attribute read request 时，该回调函数用于获取 specific service 提供的 attribute value。

write_attr_cb: 写 attribute 回调函数，当 client 发送 attribute write request 时，该回调函数用于写入 specific service 提供的 attribute value。

cccd_update_cb: 更新 CCCD 数值回调函数，用于通知 specific service，service 中相应 CCCD 数值已被 client 写入。

```

const T_FUN_GATT_SERVICE_CBS simp_ble_service_cbs =
{
    simp_ble_service_attr_read_cb, // Read callback function pointer
    simp_ble_service_attr_write_cb, // Write callback function pointer
    simp_ble_service_cccd_update_cb // CCCD update callback function pointer
};

```

3.1.3.2.3 Write Indication Post Procedure 回调函数

Write indication post procedure 回调函数用于在处理 client 的 write request 之后执行一些后续流程。该回调函数是在 write attribute 回调函数中被初始化的。若无后续流程需要执行，write attribute 回调函数中的

p_write_post_proc 指针必须为 NULL。在 profile_server.h 中定义 Write indication post procedure 回调函数。

3.1.3.3 Characteristic Value Read

该流程用于从 server 读取 characteristic value。有四个子流程可以用于读取 characteristic value，包括 read characteristic value、read using characteristic UUID、read long characteristic values 和 read multiple characteristic values。一个可读的 attribute 必须配置可读 permission。根据不同的 attribute flag，可以从 service 或 APP 读取 attribute value。

3.1.3.3.1 由 Attribute Element 提供 Attribute Value

flag 为 ATTRIB_FLAG_VALUE_INCL 的 attribute 会涉及该流程。

```
{
    ATTRIB_FLAG_VALUE_INCL,                                /* flags */
    {
        LO_WORD(0x2A04),                                    /* type_value */
        HI_WORD(0x2A04),
        100,
        200,
        0,
        LO_WORD(2000),
        HI_WORD(2000)
    },
    5,                                                 /* bValueLen */
    NULL,
    GATT_PERM_READ                                     /* permissions */
},
```

该流程中各层之间的交互如图 3-5 所示，protocol stack layer 将从 attribute element 中读取 attribute value，并直接在 read response 中响应该 attribute value。

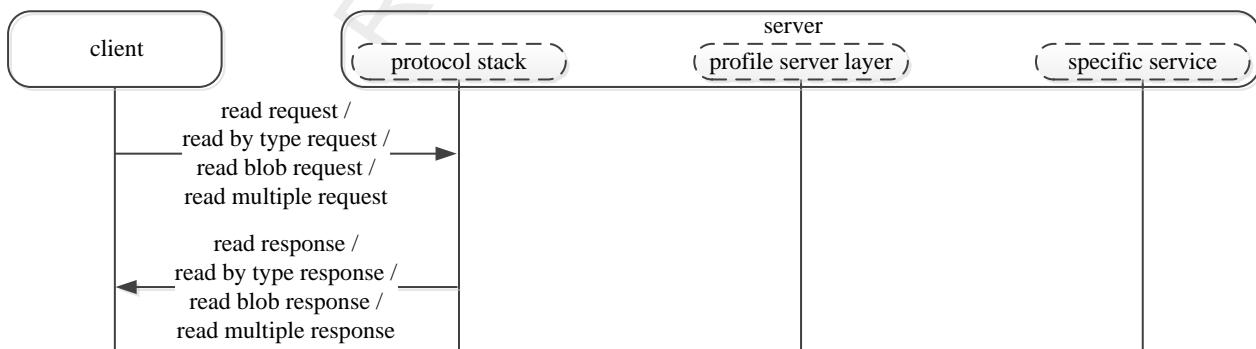


图 3-5 读 Characteristic Value – 由 Attribute Element 提供 Attribute Value

3.1.3.3.2 由 APP 提供 Attribute Value 且结果未挂起

flag 为 ATTRIB_FLAG_VALUE_APPL 的 attribute 会涉及该流程。

```
{
    ATTRIB_FLAG_VALUE_APPL,
    {
        LO_WORD(GATT_UUID_CHAR_SIMPLE_V1_READ),
        HI_WORD(GATT_UUID_CHAR_SIMPLE_V1_READ)
    },
    0,
    NULL,
    GATT_PERM_READ
},
```

该流程中各层之间的交互如图 3-6 所示。当本地设备收到 read request 时，protocol stack 将发送 read indication 给 profile server layer，profile server layer 将调用 read attribute 回调函数获取 specific service 中的 attribute value。然后，profile server layer 通过 read confirmation 将数据传递给 protocol stack。

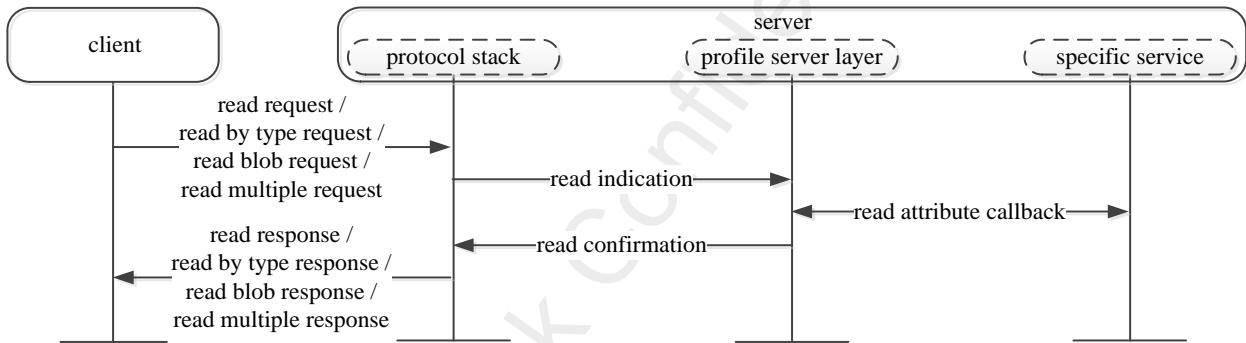


图 3-6 读 Characteristic Value – 由 APP 提供 Attribute Value 且结果未挂起

示例代码如下所示，app_profile_callback()的返回结果必须为 APP_RESULT_SUCCESS。

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    ...
    else if (service_id == simp_svr_id)
    {
        TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;
        switch (p_simp_cb_data->msg_type)
        {
            case SERVICE_CALLBACK_TYPE_READ_CHAR_VALUE:
            {
                if (p_simp_cb_data->msg_data.read_value_index == SIMP_READ_V1)
```

```

    {
        uint8_t value[2] = {0x01, 0x02};
        APP_PRINT_INFO0("SIMP_READ_V1");
        simp_ble_service_set_parameter(
            SIMPLE_BLE_SERVICE_PARAM_V1_READ_CHAR_VAL, 2, &value);
    }
}
break;
}
...
return app_result;
}

```

3.1.3.3.3 由 APP 提供 Attribute Value 且结果挂起

flag 为 ATTRIB_FLAG_VALUE_APPL 的 attribute 会涉及该流程。

由于 APP 提供的 attribute value 不能立即被读取, specific service 需要调用 server_attr_read_confirm() 传递 attribute value。该流程中各层之间的交互如图 3-7 所示。

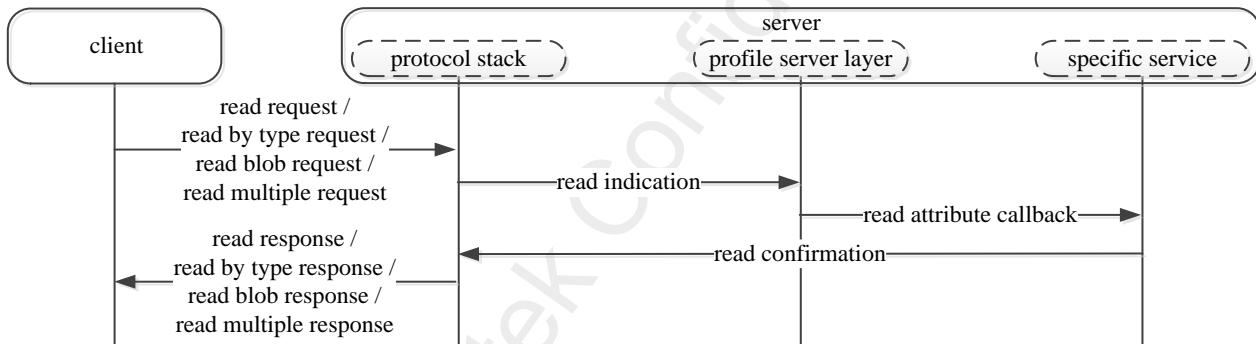


图 3-7 读 Characteristic Value - 由 APP 提供 Attribute Value 且结果挂起

示例代码如下所示, app_profile_callback()的返回结果必须为 APP_RESULT_PENDING。

```

T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_PENDING;
    ...
    else if (service_id == simp_srv_id)
    {
        TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;
        switch (p_simp_cb_data->msg_type)
        {
            case SERVICE_CALLBACK_TYPE_READ_CHAR_VALUE:
            {
                if (p_simp_cb_data->msg_data.read_value_index == SIMP_READ_V1)

```

```

    {
        uint8_t value[2] = {0x01, 0x02};
        APP_PRINT_INFO0("SIMP_READ_V1");
        simp_ble_service_set_parameter(
            SIMPLE_BLE_SERVICE_PARAM_V1_READ_CHAR_VAL, 2, &value);
    }
}

break;
}

...
return app_result;
}

```

3.1.3.4 Characteristic Value Write

该流程用于向 server 写入 characteristic value。有四个子流程可以用于写入 characteristic value，包括 write without response、signed write without response、write characteristic value 和 write long characteristic values。

3.1.3.4.1 Write Characteristic Value

1. 由 Attribute Element 提供 Attribute Value

flag 为 ATTRIB_FLAG_VOID 的 attribute 会涉及该流程。

```

uint8_t cha_val_v8_011[1] = {0x08};
const T_ATTRIB_APPL gatt_dfindme_profile[] = {
    .....
    /* handle = 0x000e Characteristic value -- Value V8 */
    {
        ATTRIB_FLAG_VOID,                                /* flags */
        {                                                 /* type_value */
            LO_WORD(0xB008),
            HI_WORD(0xB008),
        },
        1,                                              /* bValueLen */
        (void *)cha_val_v8_011,
        GATT_PERM_READ | GATT_PERM_WRITE               /* permissions */
    },
    .....
}

```

该流程中各层之间的交互如图 3-8 所示，write request 用于请求 server 写 attribute value，且在写操作结束之后直接发送 write response。

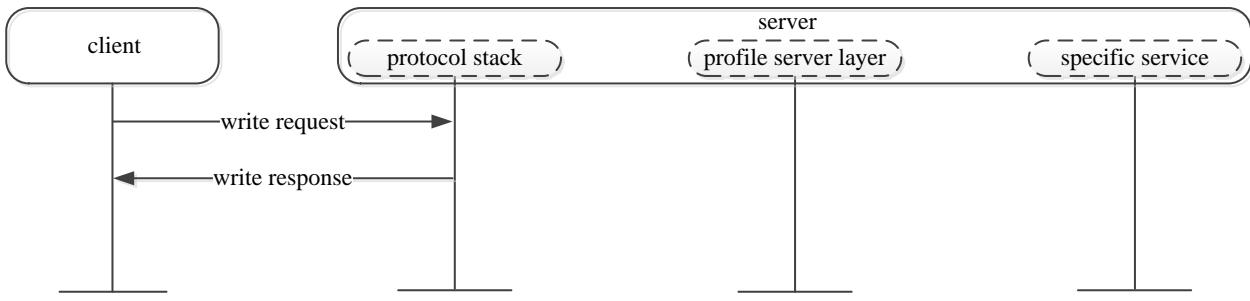


图 3-8 Write Characteristic Value – 由 Attribute Element 提供 Attribute Value

2. 由 APP 提供 Attribute Value 且结果未挂起

flag 为 ATTRIB_FLAG_VALUE_APPL 的 attribute 会涉及该流程。

该流程中各层之间的交互如图 3-9 所示，当本地设备收到 write request，protocol stack 将发送 write request indication 给 profile server layer，profile server layer 将调用 write attribute callback 写入 specific service 中的 attribute value。Profile server layer 将通过 write request confirmation 返回写入结果。

若在 profile server layer 返回 write confirmation 之后 server 需要执行后续流程，回调函数指针 write_ind_post_proc()不为 NULL 时，将调用该回调函数。

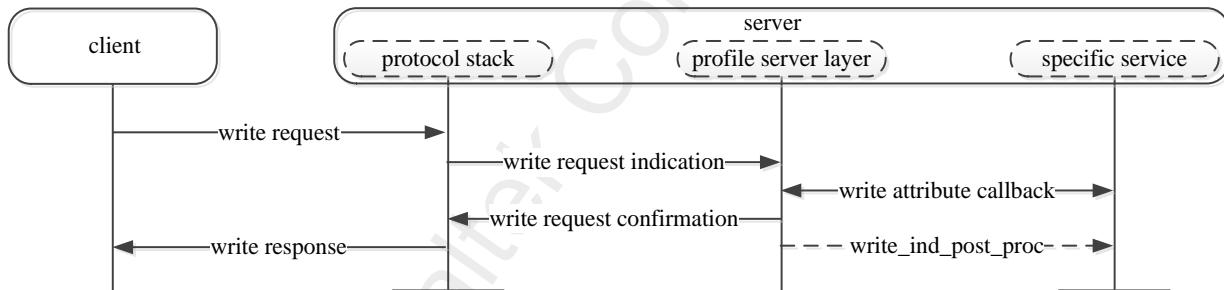


图 3-9 Write Characteristic Value – 由 APP 提供 Attribute Value 且结果未挂起

由 server_add_service()注册的 srv_cbs 回调函数通知 APP， write_type 为 WRITE_REQUEST。示例代码如下所示，app_profile_callback()的返回结果必须为 APP_RESULT_SUCCESS。

```

T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    ...
    else if (service_id == simp_svr_id)
    {
        TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;
        switch (p_simp_cb_data->msg_type)
        {

```

```

case SERVICE_CALLBACK_TYPE_WRITE_CHAR_VALUE:
{
    switch (p_simp_cb_data->msg_data.write.opcode)
    {
        case SIMP_WRITE_V2:
        {
            APP_PRINT_INFO2("SIMP_WRITE_V2: write type %d, len %d",
                p_simp_cb_data->msg_data.write.write_type,
                p_simp_cb_data->msg_data.write.len);
        }
        ...
    }
    return app_result;
}

```

3. 由 APP 提供 Attribute Value 且结果挂起

flag 为 ATTRIB_FLAG_VALUE_APPL 的 attribute 会涉及该流程。

若写 attribute value 流程不能立即结束, specific service 将会调用 server_attr_write_confirm()。该流程中各层之间的交互如图 3-10 所示。Write indication post procedure 为可选流程。

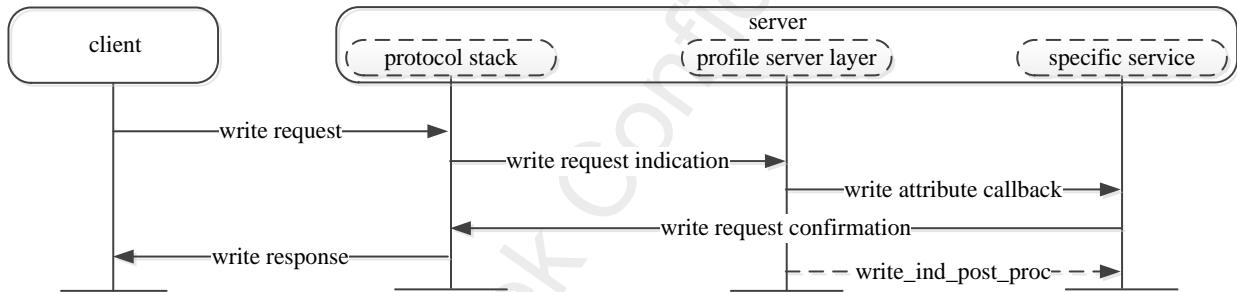


图 3-10 Write Characteristic Value – 由 APP 提供 Attribute Value 且结果挂起

由 server_add_service()注册的 srv_cbs 回调函数通知 APP, write_type 为 WRITE_REQUEST。示例代码如下所示, app_profile_callback()的返回结果必须为 APP_RESULT_PENDING。

```

T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_PENDING;
    ...
    else if (service_id == simp_srv_id)
    {
        TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;
        switch (p_simp_cb_data->msg_type)
        {
            case SERVICE_CALLBACK_TYPE_WRITE_CHAR_VALUE:
            {
                switch (p_simp_cb_data->msg_data.write.opcode)

```

```

    {
        case SIMP_WRITE_V2:
        {
            APP_PRINT_INFO2("SIMP_WRITE_V2: write type %d, len %d",
                            p_simp_cb_data->msg_data.write.write_type,
                            p_simp_cb_data->msg_data.write.len);
        }
        ...
    }

    return app_result;
}

```

4. 写 CCCD 值

若本地设备收到 client 的 write request 以写 CCCD, protocol stack 将更新 CCCD 信息, profile server layer 通过更新 CCCD 回调函数向 APP 通知 CCCD 信息已更新。该流程中各层之间的交互如图 3-11 所示。

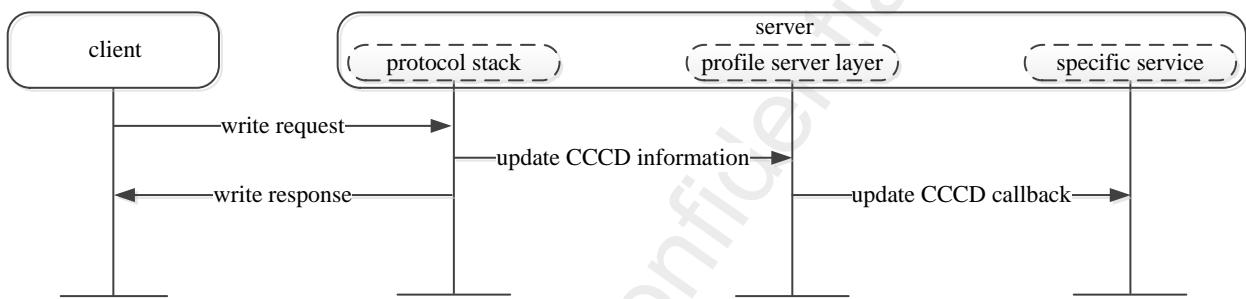


图 3-11 Write Characteristic Value - 写 CCCD 值

```

void simp_ble_service_cccd_update_cb(uint8_t conn_id, T_SERVER_ID service_id, uint16_t index,
                                      uint16_t cccbits)
{
    TSIMP_CALLBACK_DATA callback_data;
    bool is_handled = false;
    callback_data.conn_id = conn_id;
    callback_data.msg_type = SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION;
    APP_PRINT_INFO2("simp_ble_service_cccd_update_cb: index = %d, cccbits 0x%x", index, cccbits);
    switch (index)
    {
        case SIMPLE_BLE_SERVICE_CHAR_NOTIFY_CCCD_INDEX:
        {
            if (cccbits & GATT_CLIENT_CHAR_CONFIG_NOTIFY)
            {
                // Enable Notification
                callback_data.msg_data.notification_indification_index = SIMP_NOTIFY_INDICATE_V3_ENABLE;
            }
        }
        else

```

```
{  
    // Disable Notification  
    callback_data.msg_data.notification_index = SIMP_NOTIFY_INDICATE_V3_DISABLE;  
}  
is_handled = true;  
}  
break;  
case SIMPLE_BLE_SERVICE_CHAR_INDICATE_CCCD_INDEX:  
{  
    if (cccbits & GATT_CLIENT_CHAR_CONFIG_INDICATE)  
    {  
        // Enable Indication  
        callback_data.msg_data.notification_index = SIMP_NOTIFY_INDICATE_V4_ENABLE;  
    }  
    else  
    {  
        // Disable Indication  
        callback_data.msg_data.notification_index = SIMP_NOTIFY_INDICATE_V4_DISABLE;  
    }  
    is_handled = true;  
}  
break;  
default:  
    break;  
}  
/* Notify Application. */  
if (pfn_simp_ble_service_cb && (is_handled == true))  
{  
    pfn_simp_ble_service_cb(service_id, (void *)&callback_data);  
}  
}
```

由 server_add_service() 注册的 srv_cbs 回调函数通知 APP , msg_type 为 SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION。

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)  
{  
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;  
    ...  
    else if (service_id == simp_srv_id)  
    {  
        TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;  
        switch (p_simp_cb_data->msg_type)  
        {  
            switch (p_simp_cb_data->msg_type)
```

```
{  
    case SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION:  
        {  
            switch (p_simp_cb_data->msg_data.notification_indification_index)  
            {  
                case SIMP_NOTIFY_INDICATE_V3_ENABLE:  
                    {  
                        APP_PRINT_INFO0("SIMP_NOTIFY_INDICATE_V3_ENABLE");  
                    }  
                break;  
                case SIMP_NOTIFY_INDICATE_V3_DISABLE:  
                    {  
                        APP_PRINT_INFO0("SIMP_NOTIFY_INDICATE_V3_DISABLE");  
                    }  
                break;  
                case SIMP_NOTIFY_INDICATE_V4_ENABLE:  
                    {  
                        APP_PRINT_INFO0("SIMP_NOTIFY_INDICATE_V4_ENABLE");  
                    }  
                break;  
                case SIMP_NOTIFY_INDICATE_V4_DISABLE:  
                    {  
                        APP_PRINT_INFO0("SIMP_NOTIFY_INDICATE_V4_DISABLE");  
                    }  
                break;  
                default:  
                    break;  
            }  
        }  
    break;  
}  
...  
return app_result;  
}
```

3.1.3.4.2 Write without Response / Signed Write without Response

Write without Response / Signed Write without Response 和 write characteristic value 流程的区别在于 server 不会发送写入结果给 client。

1. 由 APP 提供 Attribute Value

flag 为 ATTRIB_FLAG_VALUE_APPL 的 attribute 会涉及该流程。

该流程中各层之间的交互如图 3-12 所示。当本地设备收到 write command 或 signed write command, 由 server_add_service() 注册的回调函数 write_attr_cb() 将会被调用。

由 `server_add_service()` 注册的 `srv_cbs` 回调函数将会通知 APP , `write_type` 为 `WRITE_WITHOUT_RESPONSE` 或 `WRITE_SIGNED_WITHOUT_RESPONSE`。

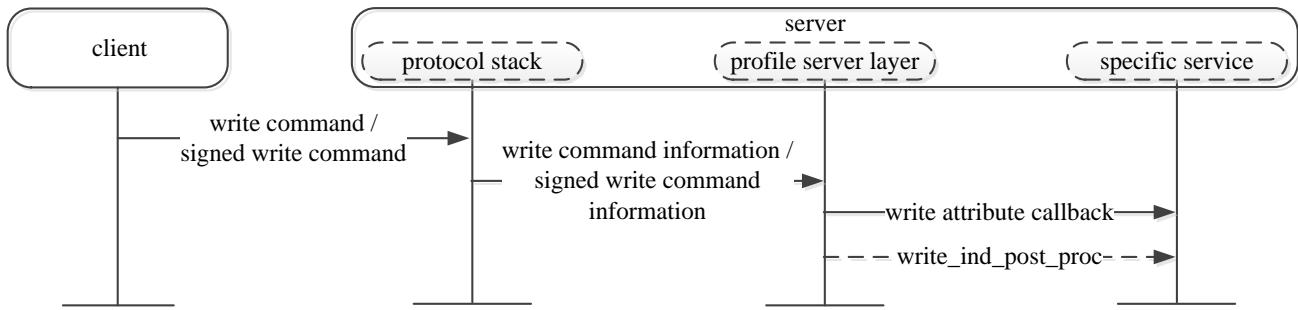


图 3-12 Write without Response / Signed Write without Response – 由 APP 提供 Attribute Value

3.1.3.4.3 Write Long Characteristic Values

1. Prepare Write

若 characteristic value 的长度大于 write request 支持的 characteristic value 的最大长度(ATT_MTU - 3), client 将使用 prepare write request。需要被写入的值将先存储在 profile server layer, 然后 profile server layer 将处理 prepare write request indication, 并返回 prepare write confirmation。该流程中各层之间的交互如图 3-13 所示。

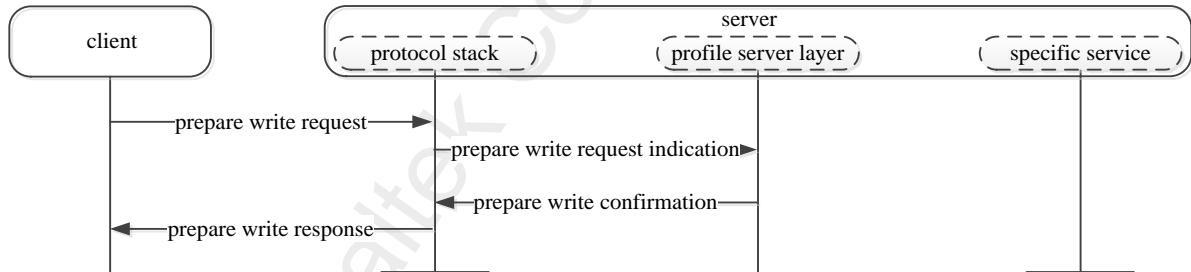


图 3-13 Write Long Characteristic Value - Prepare Write 流程

2. 结果未挂起的 Execute Write

在发送 prepare write request 之后, execute write quest 用于完成写入 attribute value 的流程。由 `server_add_service()` 注册的 `srv_cbs` 回调函数通知 APP, `write_type` 为 `WRITE_LONG`。Write indication post procedure 为可选流程。该流程中各层之间的交互如图 3-14 所示。

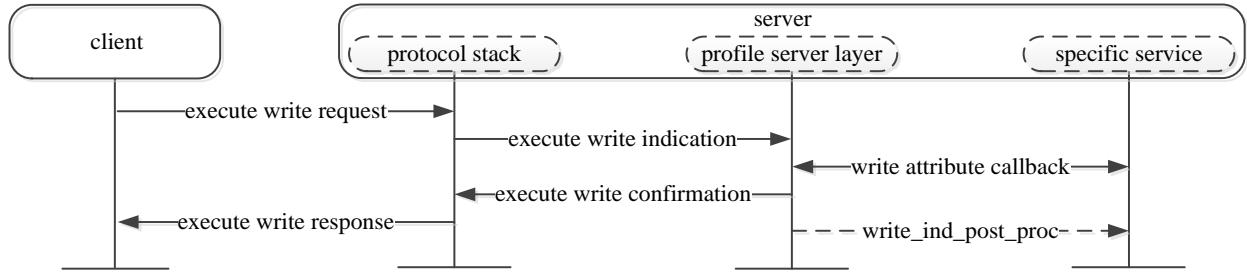


图 3-14 Write Long Characteristic Values- 结果未挂起的 Execute Write

3. 结果挂起的 Execute Write

若写入操作不能立即完成，specific service 将调用 server_exec_write_confirm()。Write indication post procedure 为可选流程。该流程中各层之间的交互如图 3-15 所示。

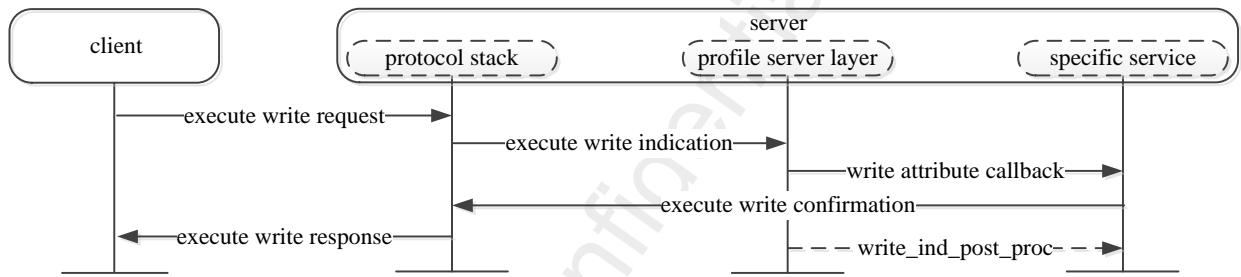


图 3-15 Write Long Characteristic Values- 结果挂起的 Execute Write

3.1.3.5 Characteristic Value Notification

Server 用该流程通知 client 一个 characteristic value。Server 主动调用 server_send_data()发送数据，在发送流程完成之后，会通过 server 通用回调函数通知 APP。该流程中各层之间的交互如图 3-16 所示。

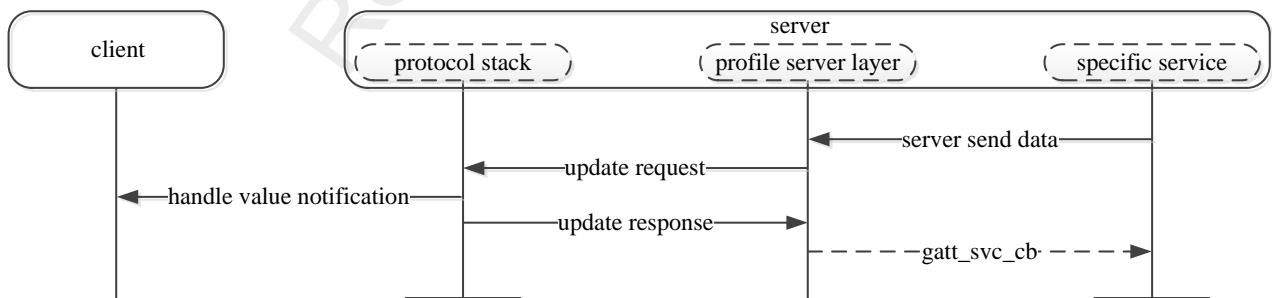


图 3-16 Characteristic Value Notification

```
bool simp_ble_service_send_v3_notify(uint8_t conn_id, T_SERVER_ID service_id, void *p_value,
```

```

        uint16_t length)
{
    APP_PRINT_INFO0("simp_ble_service_send_v3_notify");
    // send notification to client
    return server_send_data(conn_id, service_id, SIMPLE_BLE_SERVICE_CHAR_V3_NOTIFY_INDEX, p_value,
                           length, GATT_PDU_TYPE_ANY);
}

```

3.1.3.6 Characteristic Value Indication

Server 用该流程给 client 指示一个 characteristic value。一旦收到 indication，client 必须用 confirmation 响应。在 server 收到 handle value confirmation 之后，会通过 server 通用回调函数通知 APP。该流程中各层之间的交互如图 3-17 所示。

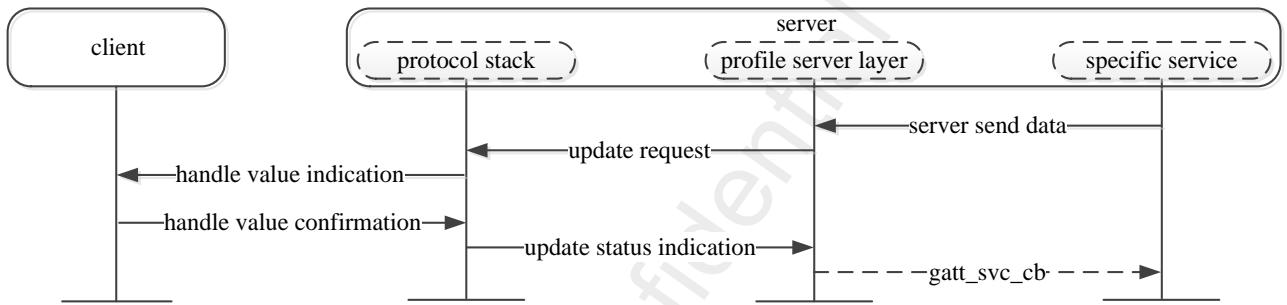


图 3-17 Characteristic Value Indication

```

bool simp_ble_service_send_v4_indicate(uint8_t conn_id, T_SERVER_ID service_id, void *p_value,
                                         uint16_t length)
{
    APP_PRINT_INFO0("simp_ble_service_send_v4_indicate");
    // send indication to client
    return server_send_data(conn_id, service_id, SIMPLE_BLE_SERVICE_CHAR_V4_INDICATE_INDEX,
                           p_value, length, GATT_PDU_TYPE_ANY);
}

```

在收到 handle value confirmation 后，将调用 app_profile_callback()。

```

T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    if (service_id == SERVICE_PROFILE_GENERAL_ID)
    {
        T_SERVER_APP_CB_DATA *p_param = (T_SERVER_APP_CB_DATA *)p_data;
        switch (p_param->eventId)
        {
            ...
        case PROFILE_EVT_SEND_DATA_COMPLETE:

```

```
APP_PRINT_INFO5("PROFILE_EVT_SEND_DATA_COMPLETE: conn_id %d, cause 0x%x,
service_id %d, attrib_idx 0x%x, credits %d",
                p_param->event_data.send_data_result.conn_id,
                p_param->event_data.send_data_result.cause,
                p_param->event_data.send_data_result.service_id,
                p_param->event_data.send_data_result.attrib_idx,
                p_param->event_data.send_data_result.credits);
if (p_param->event_data.send_data_result.cause == GAP_SUCCESS)
{
    APP_PRINT_INFO0("PROFILE_EVT_SEND_DATA_COMPLETE success");
}
else
{
    APP_PRINT_ERROR0("PROFILE_EVT_SEND_DATA_COMPLETE failed");
}
break;
default:
    break;
}
```

3.1.4 Specific Service 的实现

为实现一个用例，一个 Profile 需要包含一个或多个 services，而 service 由 characteristics 组成，每个 characteristic 包括必选的 characteristic value 和可选的 characteristic descriptor。因而，service、characteristic 以及 characteristic 的组成成分（例如，值和 descriptors）包含 Profile 数据，并存储于 server 的 attributes 中。

以下为开发 specific service 的步骤：

1. 定义 Service 和 Profile Spec
2. 定义 Service Attribute Table
3. 定义 Service 与 APP 之间的接口
4. 定义 `xxx_add_service()`, `xxx_set_parameter()`, `xxx_notify()`, `xxx_indicate()` 等 API
5. 用 `T_FUN_GATT_SERVICE_CBS` 实现回调函数 `xxx_ble_service_cbs`

本节内容以 simple BLE service 为例，简单介绍如何实现 specific service。具体细节参见 `simple_ble_service.c` 和 `simple_ble_service.h` 中的源代码。

3.1.4.1 定义 Service 和 Profile Spec

为实现 specific service，定义 Service 和 Profile Spec。

3.1.4.2 定义 Service Attribute Table

由 attribute elements 组成的 service 是通过 service table 定义的，一个 service table 可以由多个 services 组成。

3.1.4.2.1 Attribute Element

Attribute Element 是 service 的基本单元，其结构体定义在 gatt.h 中。

```
typedef struct {
    uint16_t    flags;           /**< Attribute flags @ref GATT_ATTRIBUTE_FLAG */
    uint8_t     type_value[2 + 14]; /*< 16 bit UUID + included value or 128 bit UUID */
    uint16_t    value_len;       /**< Length of value */
    void        *p_value_context; /*< Pointer to value if @ref ATTRIB_FLAG_VALUE_INCL
                                and @ref ATTRIB_FLAG_VALUE_APPL not set */
    uint32_t    permissions;    /**< Attribute permission @ref GATT_ATTRIBUTE_PERMISSIONS */
} T_ATTRIB_APPL;
```

1. Flags

Flags 的可选值和描述见表 3-3。

表 3-3 Flags 的可选值和描述

Option Values	Description
ATTRIB_FLAG_LE	Used only for primary service declaration attributes if GATT over BLE is supported
ATTRIB_FLAG_VOID	Attribute value is neither supplied by application nor included following 16bit UUID. Attribute value is pointed by p_value_context and value_len shall be set to the length of attribute value.
ATTRIB_FLAG_VALUE_INCL	Attribute value is included following 16 bit UUID
ATTRIB_FLAG_VALUE_APPL	Application has to supply attribute value
ATTRIB_FLAG_UUID_128BIT	Attribute uses 128 bit UUID
ATTRIB_FLAG_ASCII_Z	Attribute value is ASCII_Z string
ATTRIB_FLAG_CCCD_APPL	Application will be informed if CCCD value is changed
ATTRIB_FLAG_CCCD_NO_FILTER	Application will be informed about CCCD value when CCCD is written by client, no matter it is changed or not

注：

ATTRIB_FLAG_LE 仅适用于 type 为 primary service declaration 的 attribute，表示 primary service 允许通过 LE 链路访问。

ATTRIB_FLAG_VOID、**ATTRIB_FLAG_VALUE_INCL** 和 **ATTRIB_FLAG_VALUE_APPL** 三者之一必须在 attribute element 中使用。

ATTRIB_FLAG_VALUE_INCL 表示 attribute value 被置于 type_value 的最后 14 字节 (type_value 的前 2 个字节用于保存 UUID)，且 value_len 是放入最后 14 字节区域的字节数目。由于 type_value 提供 attribute

value, p_value_context 指针为 NULL。

ATTRIB_FLAG_VALUE_APPL 表示由 APP 提供 attribute value。只要协议栈涉及对该 attribute value 的操作，协议栈将与 APP 进行交互以完成相应处理流程。由于 attribute value 是由 APP 提供的，type_value 仅保存 UUID，value_len 为 0，且 p_value_context 指针为 NULL。

ATTRIB_FLAG_VOID 表示 attribute value 既不放置于 type_value 的最后 14 个字节，也不由 APP 提供。此时，type_value 仅保存 UUID，p_value_context 指针指向 attribute value，value_len 表示 attribute value 的长度。

表 3-4 展示 flags value 与 read attribute 流程使用的 actual value 之间的关联。

表 3-4 Flags Value 的选择模式

	APPL	APPL ASCII_Z	INCL	INCL ASCII_Z	VOID	VOID ASCII_Z
If set	value_len	Any(NULL)	Any(NULL)	Strlen(value)	Strlen(value)	Strlen(value)
	type_value+2	Any(NULL)	Any(NULL)	value	value	Any(NULL)
	p_value_context	Any(NULL)	Any(NULL)	Any(NULL)	Any(NULL)	value
Actual get by read attribute process	Actual length	Reply by application	Reply by application	Strlen(value)	Strlen(value)+1	Strlen(value)
	Actual value	Reply by application	Reply by application	value	Value + '\0'	Value

APPL: ATTRIB_FLAG_VALUE_APPL

VOID: ATTRIB_FLAG_VOID

INCL: ATTRIB_FLAG_VALUE_INCL

ASCII_Z: ATTRIB_FLAG_ASCII_Z

2. Permissions

Attribute 的 Permissions 指定 read 或 write 访问需要的安全级别，同样包括 notification 或 indication。Permissions 的值表示该 attribute 的 permission。Attribute permissions 是 access permissions、encryption permissions、authentication permissions 和 authorization permissions 的组合，其可用值如表 3-5 所示。

表 3-5 Permissions 的可用值

Types	Permissions
Read Permissions	GATT_PERM_READ
	GATT_PERM_READ_AUTHEN_REQ
	GATT_PERM_READ_AUTHEN_MITM_REQ
	GATT_PERM_READ_AUTHOR_REQ
	GATT_PERM_READ_ENCRYPTED_REQ
	GATT_PERM_READ_AUTHEN_SC_REQ
Write Permissions	GATT_PERM_WRITE
	GATT_PERM_WRITE_AUTHEN_REQ

	GATT_PERM_WRITE_AUTHEN_MITM_REQ
	GATT_PERM_WRITE_AUTHOR_REQ
	GATT_PERM_WRITE_ENCRYPTED_REQ
	GATT_PERM_WRITE_AUTHEN_SC_REQ
	GATT_PERM_NOTIF_IND
	GATT_PERM_NOTIF_IND_AUTHEN_REQ
Notify/Indicate Permissions	GATT_PERM_NOTIF_IND_AUTHEN_MITM_REQ
	GATT_PERM_NOTIF_IND_AUTHOR_REQ
	GATT_PERM_NOTIF_IND_ENCRYPTED_REQ
	GATT_PERM_NOTIF_IND_AUTHEN_SC_REQ

3.1.4.2.2 Service Table

Service 包含一组 attributes，其被称之为 service table。一个 service table 包含各种类型的 attributes，例如 service declaration、characteristic declaration、characteristic value 和 characteristic descriptor declaration。

Service table 的示例如表 3-6 所示，在 ble_peripheral 示例工程的 simple_ble_service.c 中实现。

表 3-6 Service Table 示例

Flags	Attribute Type	Attribute Value	Permission
INCL LE	<<primary service declaration>>	<<simple profile UUID – 0xA00A>>	read
INCL	<<characteristic declaration>>	Property(read)	read
APPL	<<characteristic value>>	UUID(0xB001),Value not defined here	read
VOID ASCII_Z	<<Characteristic User Description>>	UUID(0x2901) Value defined in p_value_context	read
INCL	<<characteristic declaration>>	Property(write write without response)	read
APPL	<<characteristic value>>	UUID(0xB002), Value not defined here	write
INCL	<<characteristic declaration>>	Property(notify)	read
APPL	<<characteristic value>>	UUID(0xB003), Value not defined here	none
CCCD_APPL	<<client characteristic configuration descriptor>>	Default CCCD value	read write
INCL	<<characteristic declaration>>	Property(indicate)	read
APPL	<<characteristic value>>	UUID(0xB004), Value not defined here	none
CCCD_APPL	<<client characteristic configuration descriptor>>	Default CCCD value	read write

注：

引号中的参数为 UUID 的值，其定义在蓝牙核心规范中或由用户自定义。

LE 为 ATTRIB_FLAG_LE 的缩写。

INCL 为 ATTRIB_FLAG_VALUE_INCL 的缩写。

APPL 为 ATTRIB_FLAG_VALUE_APPL 的缩写。

Service table 的示例代码如下所示：

```
const T_ATTRIB_APPL simple_ble_service_tbl[] =  
{  
    /* <<Primary Service>>, .. */  
    {  
        (ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_LE),      /* flags */  
        {  
            LO_WORD(GATT_UUID_PRIMARY_SERVICE),  
            HI_WORD(GATT_UUID_PRIMARY_SERVICE),  
            LO_WORD(GATT_UUID_SIMPLE_PROFILE),           /* service UUID */  
            HI_WORD(GATT_UUID_SIMPLE_PROFILE)  
        },  
        UUID_16BIT_SIZE,                            /* bValueLen */  
        NULL,                                     /* p_value_context */  
        GATT_PERM_READ                            /* permissions */  
    },  
    /* <<Characteristic>> demo for read */  
    {  
        ATTRIB_FLAG_VALUE_INCL,                  /* flags */  
        {  
            LO_WORD(GATT_UUID_CHARACTERISTIC),  
            HI_WORD(GATT_UUID_CHARACTERISTIC),  
            GATT_CHAR_PROP_READ,                   /* characteristic properties */  
            /* characteristic UUID not needed here, is UUID of next attrib. */  
        },  
        1,                                         /* bValueLen */  
        NULL,  
        GATT_PERM_READ                            /* permissions */  
    },  
    {  
        ATTRIB_FLAG_VALUE_APPL,                  /* flags */  
        {  
            LO_WORD(GATT_UUID_CHAR_SIMPLE_V1_READ),  
            HI_WORD(GATT_UUID_CHAR_SIMPLE_V1_READ)  
        },  
        0,                                         /* bValueLen */  
        NULL,  
        GATT_PERM_READ                            /* permissions */  
    },
```

```

{

    ATTRIB_FLAG_VOID | ATTRIB_FLAG_ASCII_Z,           /*flags*/
    {                                                 /*type_value*/
        LO_WORD(GATT_UUID_CHAR_USER_DESCR),
        HI_WORD(GATT_UUID_CHAR_USER_DESCR),
    },
    (sizeof(v1_user_descr) - 1),                     /*bValueLen*/
    (void *)v1_user_descr,
    GATT_PERM_READ          /*permissions*/
},

/* <<Characteristic>> demo for write */
{

    ATTRIB_FLAG_VALUE_INCL,                         /*flags*/
    {                                                 /*type_value*/
        LO_WORD(GATT_UUID_CHARACTERISTIC),
        HI_WORD(GATT_UUID_CHARACTERISTIC),
        (GATT_CHAR_PROP_WRITE | GATT_CHAR_PROP_WRITE_NO_RSP) /*characteristic properties*/
    }
    /* characteristic UUID not needed here, is UUID of next attrib. */
},

1,                                              /*bValueLen*/
NULL,
GATT_PERM_READ          /*permissions*/
},

{

    ATTRIB_FLAG_VALUE_APPL,                         /*flags*/
    {                                                 /*type_value*/
        LO_WORD(GATT_UUID_CHAR_SIMPLE_V2_WRITE),
        HI_WORD(GATT_UUID_CHAR_SIMPLE_V2_WRITE)
    },
    0,                                              /*bValueLen*/
NULL,
GATT_PERM_WRITE          /*permissions*/
},

/* <<Characteristic>>, demo for notify */
{

    ATTRIB_FLAG_VALUE_INCL,                         /*flags*/
    {                                                 /*type_value*/
        LO_WORD(GATT_UUID_CHARACTERISTIC),
        HI_WORD(GATT_UUID_CHARACTERISTIC),
        (GATT_CHAR_PROP_NOTIFY)           /*characteristic properties*/
    }
    /* characteristic UUID not needed here, is UUID of next attrib. */
},

1,                                              /*bValueLen*/

```

```

        NULL,
        GATT_PERM_READ
    },
    {
        ATTRIB_FLAG_VALUE_APPL, /* flags */
        /* type_value */
        LO_WORD(GATT_UUID_CHAR_SIMPLE_V3_NOTIFY),
        HI_WORD(GATT_UUID_CHAR_SIMPLE_V3_NOTIFY)
    },
    0, /* bValueLen */
    NULL,
    GATT_PERM_NONE /* permissions */
},
/* client characteristic configuration */
{
    ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_CCCD_APPL, /* flags */
    /* type_value */
    LO_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
    HI_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
    /* NOTE: this value has an instantiation for each client, a write to */
    /* this attribute does not modify this default value: */ /* */
    LO_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT), /* client char. config. bit field */
    HI_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT)
},
2, /* bValueLen */
NULL,
(GATT_PERM_READ | GATT_PERM_WRITE) /* permissions */
},
/* <<Characteristic>> demo for indicate */
{
    ATTRIB_FLAG_VALUE_INCL, /* flags */
    /* type_value */
    LO_WORD(GATT_UUID_CHARACTERISTIC),
    HI_WORD(GATT_UUID_CHARACTERISTIC),
    (GATT_CHAR_PROP_INDICATE) /* characteristic properties */
    /* characteristic UUID not needed here, is UUID of next attrib. */
},
1, /* bValueLen */
NULL,
GATT_PERM_READ /* permissions */
},
{
    ATTRIB_FLAG_VALUE_APPL, /* flags */
    /* type_value */

```

```

        LO_WORD(GATT_UUID_CHAR_SIMPLE_V4_INDICATE),
        HI_WORD(GATT_UUID_CHAR_SIMPLE_V4_INDICATE)
    },
    0,                                     /* bValueLen */
    NULL,                                    /* permissions */
},
/* client characteristic configuration */
{
    ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_CCCD_APPL,           /* flags */
    {
        /* type_value */
        LO_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
        HI_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
        /* NOTE: this value has an instantiation for each client, a write to */
        /* this attribute does not modify this default value: */             */
        LO_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT), /* client char. config. bit field */
        HI_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT)
    },
    2,                                     /* bValueLen */
    NULL,
    (GATT_PERM_READ | GATT_PERM_WRITE)      /* permissions */
},
);

```

3.1.4.3 定义 Service 与 APP 之间的接口

当 service 的 attribute value 被读取或写入时，将通过 APP 注册的回调函数通知 APP。以 simple BLE service 为例，定义类型为 TSIMP_CALLBACK_DATA 的数据结构以保存需要通知的结果。

```

typedef struct {
    uint8_t          conn_id;
    T_SERVICE_CALLBACK_TYPE msg_type;
    TSIMP_UPSTREAM_MSG_DATA msg_data;
} TSIMP_CALLBACK_DATA;

```

msg_type 表示操作类型是读操作、写操作或更新 CCCD 操作。

```

typedef enum {
    SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION = 1,
    SERVICE_CALLBACK_TYPE_READ_CHAR_VALUE = 2,
    SERVICE_CALLBACK_TYPE_WRITE_CHAR_VALUE = 3,
} T_SERVICE_CALLBACK_TYPE;

```

msg_data 保存读操作、写操作或更新 CCCD 操作的数据。

3.1.4.4 定义 xxx_add_service(), xxx_set_parameter(), xxx_notify(), xxx_indicate() 等 API

xxx_add_service()用于向 profile server layer 添加 service table，注册针对 attribute 的读操作、写操作或更新 CCCD 操作的回调函数。

xxx_set_parameter()用于供 APP 设置 service 相关数据。

xxx_notify() 用于发送 notification 数据。

xxx_indicate()用于发送 indication 数据。

3.1.4.5 用 T_FUN_GATT_SERVICE_CBS 实现回调函数 xxx_ble_service_cbs

xxx_ble_service_cbs 用于处理 client 的读操作、写操作或更新 CCCD 操作。

```
const T_FUN_GATT_SERVICE_CBS simp_ble_service_cbs = {
    simp_ble_service_attr_read_cb, // Read callback function pointer
    simp_ble_service_attr_write_cb, // Write callback function pointer
    simp_ble_service_cccd_update_cb // CCCD update callback function pointer
};
```

在 **xxx_ble_service_add_service()**中调用 **server_add_service()**以注册该回调函数。

```
T_SERVER_ID simp_ble_service_add_service(void *p_func)
{
    if (false == server_add_service(&simp_service_id,
        (uint8_t *)simple_ble_service_tbl,
        sizeof(simple_ble_service_tbl),
        simp_ble_service_cbs))
    {
        APP_PRINT_ERROR0("simp_ble_service_add_service: fail");
        simp_service_id = 0xff;
        return simp_service_id;
    }
    pfn_simp_ble_service_cb = (P_FUN_SERVER_GENERAL_CB)p_func;
    return simp_service_id;
}
```

3.2 BLE Profile Client

3.2.1 概述

Profile 的 client 接口给开发者提供 discover services、接收和处理 indication 和 notification、给 GATT Server 发送 read/write request 的功能。

图 3-18 展示 Profile client 层级。Profile 包括 profile client layer 和 specific profile client。位于 protocol stack 之上的 profile client layer 封装供 specific client 访问 protocol stack 的接口，因此针对 specific client 的开发不涉及 protocol stack 的细节，使开发变得更简单和清晰。基于 profile client layer 的 specific client 是由 application layer 实现的，其实现不同于 specific server 的实现。profile client 不涉及 attribute table，提供收集和获取信息的功能，而不是提供 service 和信息。

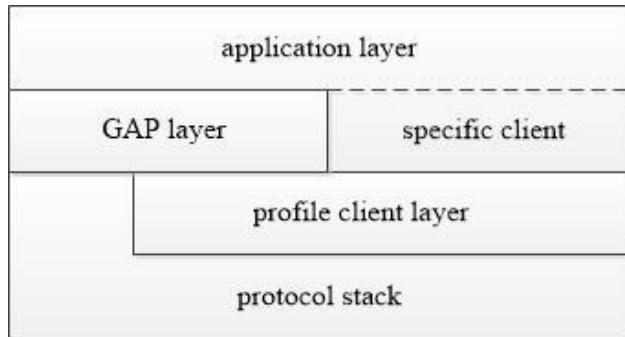


图 3-18 Profile Client 层级

3.2.2 支持的 Clients

支持的 clients 如表 3-7 所示。

表 3-7 支持的 Clients

Terms	Definitions	Files
GAP Client	Attribute Service Client	gaps_client.c gaps_client.h
BAS Client	Battery Service Client	bas_client.c bas_client.h
ANCS Client	Apple Notification Center Service Client	ancs_client.c ancs_client.h
SIMP Client	Simple BLE Service Client	simple_ble_client.c simple_ble_client.h
IPSS Client	Internet Protocol Support Service Client	ipss_client.c ipss_client.h

3.2.3 Profile Client Layer

Profile Client Layer 处理与 protocol stack layer 的交互，提供接口用于设计 specific client。Client 将对 server 进行 discover services 和 discover characteristics、读取和写入 attribute、接收和处理 notifications 和 indications 相关操作。

3.2.3.1 Client 通用回调函数

Client 通用回调函数用于向 APP 发送 client_all_primary_srv_discovery() 的结果，client_id 为 CLIENT_PROFILE_GENERAL_ID。该回调函数由 client_register_general_client_cb() 初始化。

```
void app_le_profile_init(void)
{
    client_init(3);
    .....
    client_register_general_client_cb(app_client_callback);
}

static T_USER_CMD_PARSE_RESULT cmd_srvdis(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    uint8_t conn_id = p_parse_value->dw_param[0];
    T_GAP_CAUSE cause;
    cause = client_all_primary_srv_discovery(conn_id, CLIENT_PROFILE_GENERAL_ID);
    return (T_USER_CMD_PARSE_RESULT)cause;
}

T_APP_RESULT app_client_callback(T_CLIENT_ID client_id, uint8_t conn_id, void *p_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    APP_PRINT_INFO2("app_client_callback: client_id %d, conn_id %d",
                    client_id, conn_id);
    if (client_id == CLIENT_PROFILE_GENERAL_ID)
    {
        T_CLIENT_APP_CB_DATA *p_client_app_cb_data = (T_CLIENT_APP_CB_DATA *)p_data;
        switch (p_client_app_cb_data->cb_type)
        {
            case CLIENT_APP_CB_TYPE_DISC_STATE:
            .....
        }
    }
}
```

若 APP 不使用 client_id 为 CLIENT_PROFILE_GENERAL_ID 的 client_all_primary_srv_discovery(), APP 不需要注册该通用回调函数。

3.2.3.2 Specific Client 回调函数

3.2.3.2.1 添加 Client

Protocol client layer 维护所有添加的 specific clients 的信息。首先，初始化需要添加的 client table 的总数目，profile client layer 提供 client_init() 接口用于初始化 client table 数目。

Protocol client layer 提供 client_register_spec_client_cb() 接口以注册 specific client 回调函数。图 3-19 展

示 client layer 包含多个 specific client tables，添加 specific client 的情况。

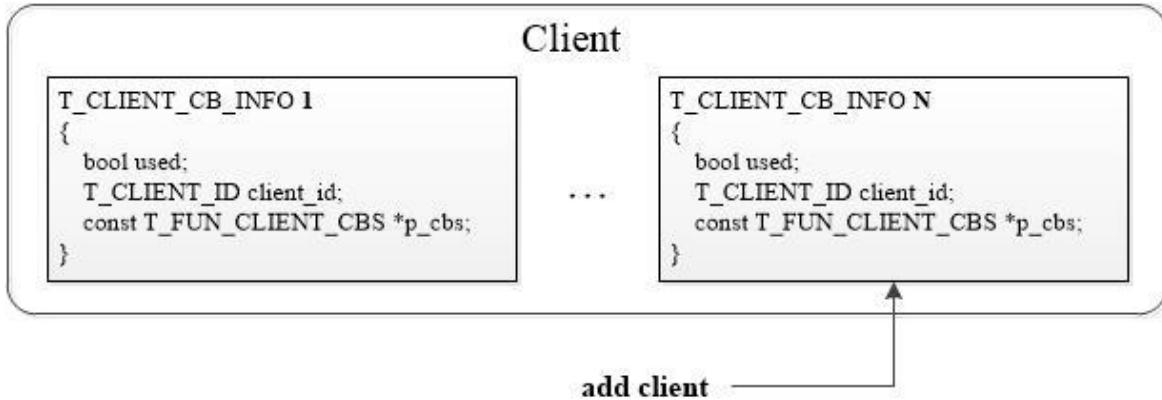


图 3-19 向 Profile Client Layer 添加 Specific Clients

APP 向 profile client layer 添加 specific client 之后，APP 将记录每个已添加 specific client 对应返回的 client id，以实现后续的数据交互流程。

3.2.3.2.2 回调函数

需要在 specific client 模块实现 specific client 回调函数，在 profile_client.h 中定义 specific client 回调函数数据结构。

```
typedef struct {
    P_FUN_DISCOVER_STATE_CB    discover_state_cb;    ///Discovery state callback function pointer
    P_FUN_DISCOVER_RESULT_CB   discover_result_cb;   ///Discovery result callback function pointer
    P_FUN_READ_RESULT_CB       read_result_cb;       ///Read response callback function pointer
    P_FUN_WRITE_RESULT_CB      write_result_cb;      ///Write result callback function pointer
    P_FUN_NOTIFY_IND_RESULT_CB notify_ind_result_cb; ///Notify Indication callback function pointer
    P_FUN_DISCONNECT_CB        disconnect_cb;        ///Disconnection callback function pointer
} T_FUN_CLIENT_CBS;
```

discover_state_cb: discovery 状态回调函数，用于向 specific client 模块通知 client_xxx_discovery 的 discover 状态。

discover_result_cb: discovery 结果回调函数，用于向 specific client 模块通知 client_xxx_discovery 的 discover 结果。

read_result_cb: read 结果回调函数，用于向 specific client 模块通知 client_attr_read() 或 client_attr_read_using_uuid() 的读取结果。

write_result_cb: write 结果回调函数，用于向 specific client 模块通知 client_attr_write() 的写入结果。

notify_ind_result_cb: notification 或 indication 回调函数，用于向 specific client 模块通知收到来自 server 的 notification 或 indication 数据。

disconnect_cb: disconnection 回调函数，用于向 specific client 模块通知一条 LE 链路已断开。

3.2.3.3 Discovery 流程

若本地设备不保存 server 的 handle 信息,那么在与 server 建立 connection 后, client 通常会执行 discovery 流程。Specific client 需要调用 `client_xxx_discovery()` 启动 discovery 流程, 然后 specific client 需要处理回调函数 `discover_state_cb()` 中的 discovery 状态以及回调函数 `discover_result_cb()` 中的 discovery 结果。

该流程中各层中间的交互如图 3-20 所示。

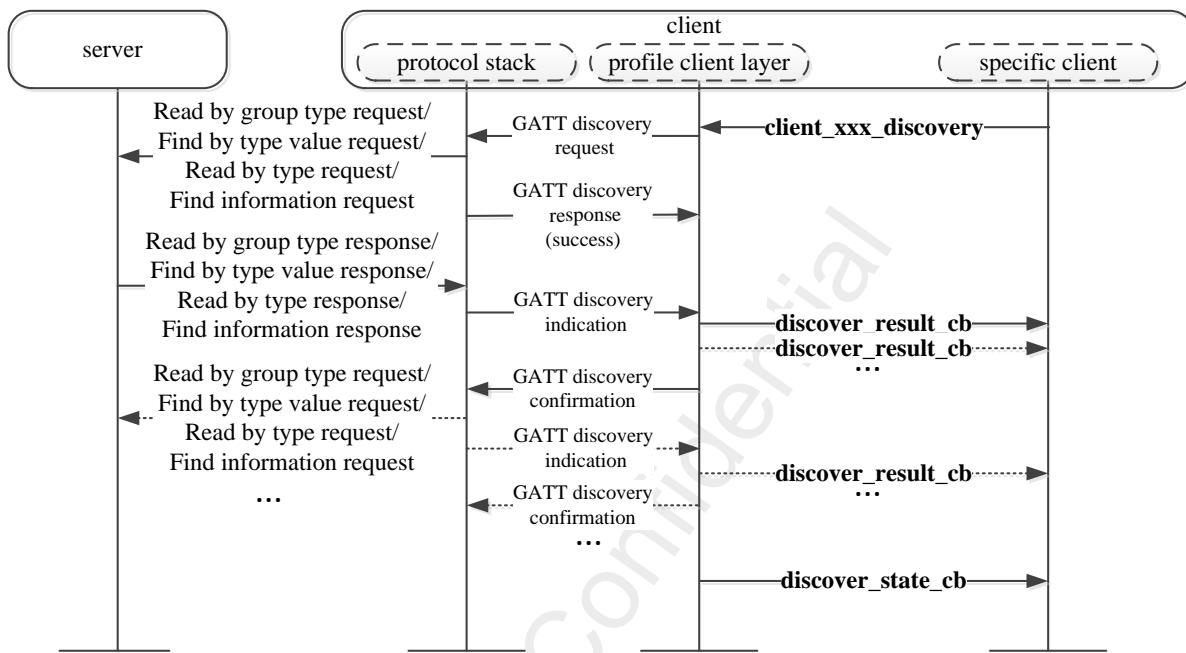


图 3-20 GATT Discovery 流程

3.2.3.3.1 Discovery 状态

表 3-8 Discovery 状态

Reference API	T_DISCOVERY_STATE
<code>client_all_primary_srv_discovery()</code>	DISC_STATE_SRV_DONE, DISC_STATE_FAILED
<code>client_by_uuid_srv_discovery()</code>	DISC_STATE_SRV_DONE DISC_STATE_FAILED
<code>client_by_uuid128_srv_discovery()</code>	DISC_STATE_SRV_DONE DISC_STATE_FAILED
<code>client_all_char_discovery()</code>	DISC_STATE_CHAR_DONE DISC_STATE_FAILED
<code>client_all_char_descriptor_discovery()</code>	DISC_STATE_CHAR_DESCRIPTOR_DONE DISC_STATE_FAILED
<code>client_relationship_discovery()</code>	DISC_STATE_RELATION_DONE DISC_STATE_FAILED

client_by_uuid_char_discovery()	DISC_STATE_CHAR_UUID16_DONE DISC_STATE_FAILED
client_by_uuid128_char_discovery()	DISC_STATE_CHAR_UUID128_DONE DISC_STATE_FAILED

3.2.3.3.2 Discovery 结果

表 3-9 Discovery 结果

Reference API	T_DISCOVERY_RESULT_TYPE	T_DISCOVERY_RESULT_DATA
client_all_primary_srv_discovery()	DISC_RESULT_ALL_SRV_UUID16	T_GATT_SERVICE_ELEM16 *p_srv_uuid16_disc_data;
client_all_primary_srv_discovery()	DISC_RESULT_ALL_SRV_UUID128	T_GATT_SERVICE_ELEM128 *p_srv_uuid128_disc_data;
client_by_uuid_srv_discovery(), client_by_uuid128_srv_discovery()	DISC_RESULT_SRV_DATA	T_GATT_SERVICE_BY_UUID_ELEM *p_srv_disc_data;
client_all_char_discovery()	DISC_RESULT_CHAR_UUID16	T_GATT_CHARACT_ELEM16 *p_char_uuid16_disc_data;
client_all_char_discovery()	DISC_RESULT_CHAR_UUID128	T_GATT_CHARACT_ELEM128 *p_char_uuid128_disc_data;
client_all_char_descriptor_discovery()	DISC_RESULT_CHAR_DESC_UUID16	T_GATT_CHARACT_DESC_ELEM16 *p_char_desc_uuid16_disc_data;
client_all_char_descriptor_discovery()	DISC_RESULT_CHAR_DESC_UUID128	T_GATT_CHARACT_DESC_ELEM128 *p_char_desc_uuid128_disc_data;
client_relationship_discovery()	DISC_RESULT_RELATION_UUID16	T_GATT_RELATION_ELEM16 *p_relation_uuid16_disc_data;
client_relationship_discovery()	DISC_RESULT_RELATION_UUID128	T_GATT_RELATION_ELEM128 *p_relation_uuid128_disc_data;
client_by_uuid_char_discovery()	DISC_RESULT_BY_UUID16_CHARACTER	T_GATT_CHARACT_ELEM16 *p_char_uuid16_disc_data;
client_by_uuid_char_discovery()	DISC_RESULT_BY_UUID128_CHARACTER	T_GATT_CHARACT_ELEM128 *p_char_uuid128_disc_data;

3.2.3.4 Characteristic Value Read

该流程用于读取 server 的 characteristic value。在 profile client layer 有两个子流程可用于读取 characteristic value：Read Characteristic Value by Handle 和 Read Characteristic Value by UUID。

3.2.3.4.1 Read Characteristic Value by Handle

当 client 已知 Characteristic Value Handle 时，该流程可用于读取 server 的 characteristic value。Read Characteristic Value by Handle 流程包含三个阶段，Phase 2 为可选阶段：

1. Phase 1: 调用 `client_attr_read()` 以读取 characteristic value。
2. Phase 2: 可选阶段。若 characteristic value 的长度大于 (ATT_MTU - 1)字节，Read Response 仅包含 characteristic value 的前(ATT_MTU - 1)字节，之后则使用 Read Long Characteristic Value 流程读取 characteristic value。
3. Phase 3: Profile client layer 调用 `read_result_cb()`以返回读取结果。

该流程中各层之间的交互如图 3-21 所示。

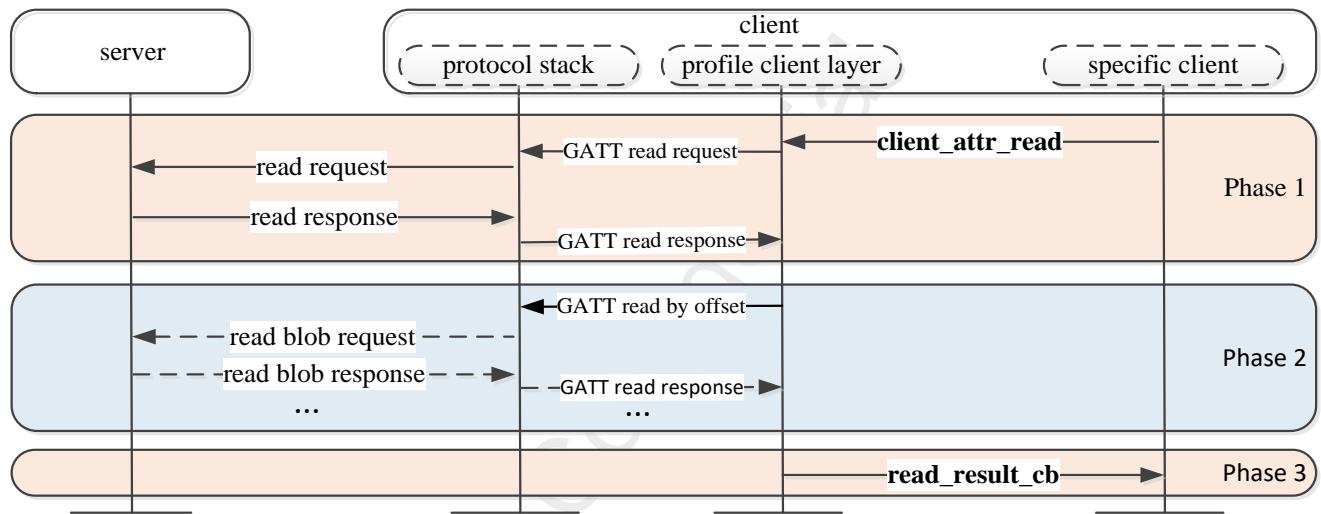


图 3-21 Read Characteristic Value by Handle 流程

3.2.3.4.2 Read Characteristic Value by UUID

当 client 仅已知 UUID 时，该流程可用于读取 server 的 characteristic value。Read Characteristic Value by UUID 流程包含三个阶段，Phase 2 为可选阶段：

1. Phase 1: 调用 `client_attr_read_using_uuid()`以读取 characteristic value。
2. Phase 2: 可选阶段。若 characteristic value 的长度大于 (ATT_MTU - 4)字节，Read by Type Response 仅包含 characteristic value 的前(ATT_MTU - 4)字节，之后则使用 Read Long Characteristic Value 流程读取 characteristic value。
3. Phase 3: Profile client layer 调用 `read_result_cb()`以返回读取结果。

该流程中各层之间的交互如图 3-22 所示。

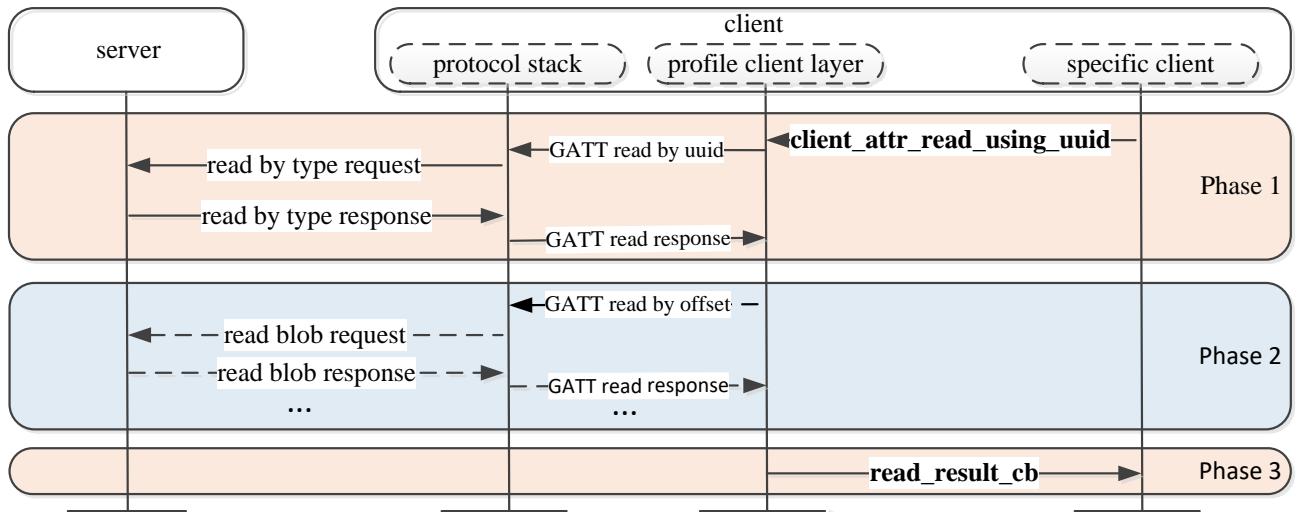


图 3-22 Read Characteristic Value by UUID 流程

3.2.3.5 Characteristic Value Write

该流程用于写入 server 的 characteristic value。在 profile client layer 有四个子流程可用于写入 characteristic value: Write without Response、Signed Write without Response、Write Characteristic Value 和 Write Long Characteristic Values。

3.2.3.5.1 Write Characteristic Value

当 client 已知 Characteristic Value Handle 时，该流程可用于写入 server 的 characteristic value。当 characteristic value 的长度小于或等于 (ATT_MTU - 3)字节，将使用该流程。否则，将使用 Write Long Characteristic Values 流程。

该流程中各层之间的交互如图 3-23 所示。

Value length <= mtu_size - 3

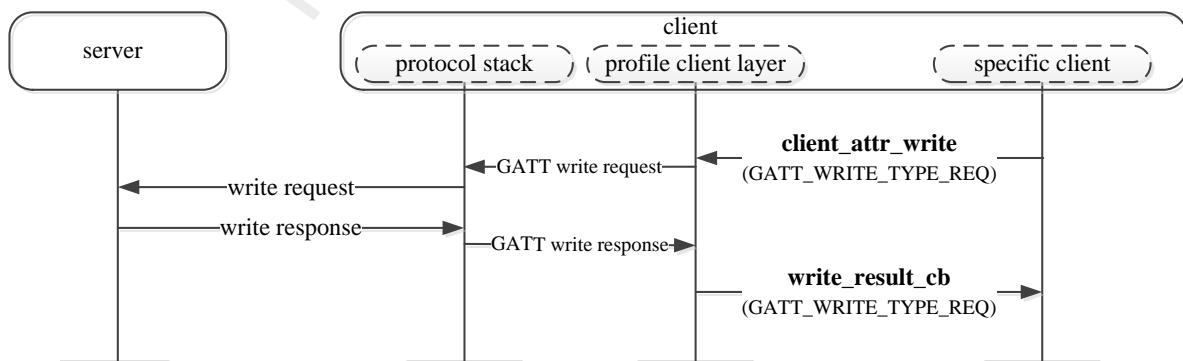


图 3-23 Write Characteristic Value 流程

3.2.3.5.2 Write Long Characteristic Values

当 client 已知 Characteristic Value Handle，且 characteristic value 的长度大于 (ATT_MTU - 3)字节时，该流程可用于写入 server 的 characteristic value。

该流程中各层之间的交互如图 3-24 所示。

Value length > mtu_size -3

Value length <= 512

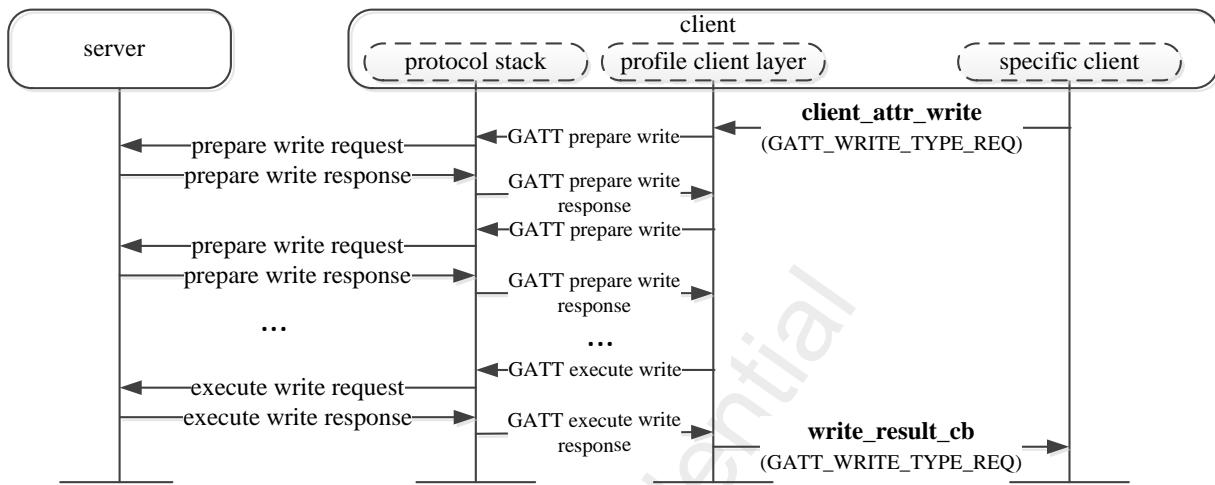


图 3-24 Write Long Characteristic Value 流程

3.2.3.5.3 Write Without Response

当 client 已知 Characteristic Value Handle，且 client 不需要写入操作成功执行的应答时，该流程可用于写入 server 的 characteristic value。characteristic value 的长度小于或等于 (ATT_MTU - 3)字节。

该流程中各层之间的交互如图 3-25 所示。

Value length <= mtu_size -3

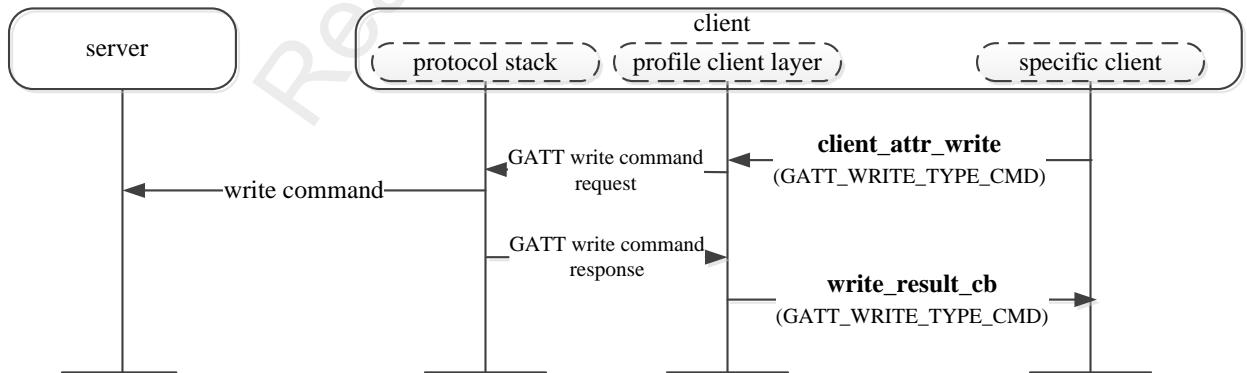


图 3-25 Write Without Response 流程

3.2.3.5.4 Signed Write without Response

当 client 已知 Characteristic Value Handle，且 ATT Bearer 未加密时，该流程可用于写入 server 的 characteristic value。该流程仅适用于 Characteristic Properties 的 authenticated 位已使能，client 与 server 设备已绑定的情况。characteristic value 的长度小于或等于 (ATT_MTU - 15)字节。

该流程中各层之间的交互如图 3-26 所示。

Value length <= mtu_size - 15

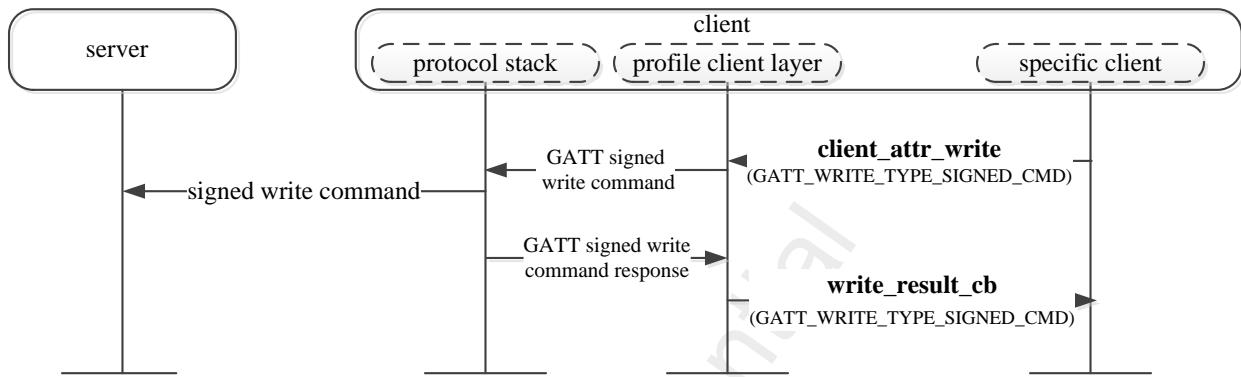


图 3-26 Signed Write without Response 流程

3.2.3.6 Characteristic Value Notification

该流程适用于 server 已被配置为向 client 通知 characteristic value，且不需要成功接收 notification 的应答的情况。

该流程中各层之间的交互如图 3-27 所示。

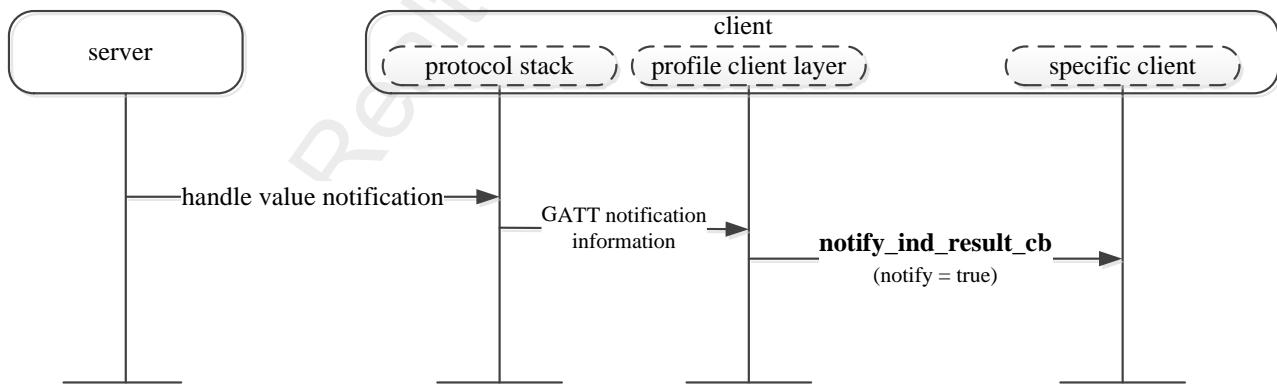


图 3-27 Characteristic Value Notification 流程

profile client layer 未存储 service handle 信息，因此 profile client layer 无法确定发送该 notification 的 specific client。profile client layer 将调用所有注册的 specific clients 回调函数，因此 specific client 需要检查

是否需要处理该 notification。

示例代码如下所示：

```
static T_APP_RESULT bas_client_notify_ind_cb(uint8_t conn_id, bool notify, uint16_t handle,
                                             uint16_t value_size, uint8_t *p_value)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    T_BAS_CLIENT_CB_DATA cb_data;
    uint16_t *hdl_cache;
    hdl_cache = bas_table[conn_id].hdl_cache;
    cb_data.cb_type = BAS_CLIENT_CB_TYPE_NOTIF_IND_RESULT;

    if (handle == hdl_cache[HDL_BAS_BATTERY_LEVEL])
    {
        cb_data.cb_content.notify_data.battery_level = *p_value;
    }
    else
    {
        return APP_RESULT_SUCCESS;
    }
    if (bas_client_cb)
    {
        app_result = (*bas_client_cb)(bas_client, conn_id, &cb_data);
    }
    return app_result;
}
```

3.2.3.7 Characteristic Value Indication

该流程适用于 server 已被配置为向 client 通知 characteristic value，且需要成功接收 indication 的应答的情况。

1. 结果未挂起的 Characteristic Value Indication

回调函数 `notify_ind_result_cb()` 的返回结果不为 `APP_RESULT_PENDING`。该流程中各层之间的交互如图 3-28 所示。

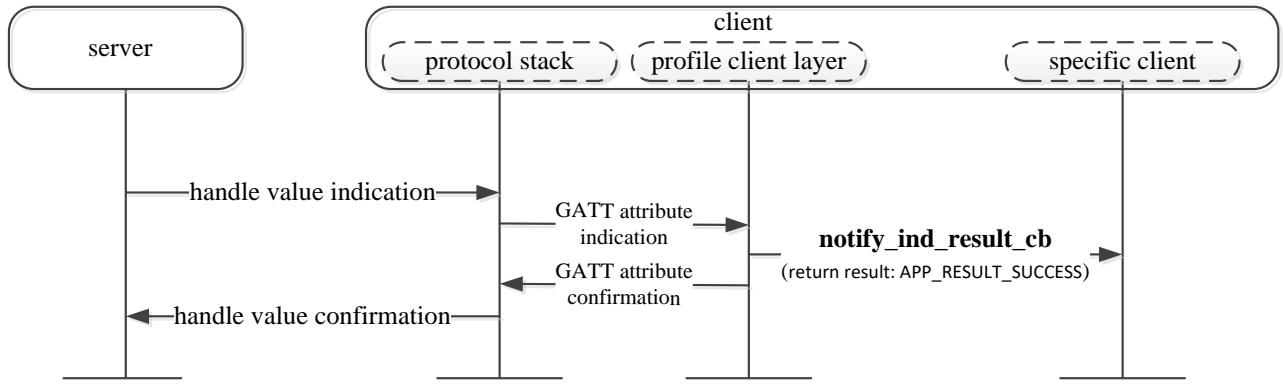


图 3-28 结果未挂起的 Characteristic Value Indication 流程

2. 结果挂起的 Characteristic Value Indication

回调函数 `notify_ind_result_cb()` 的返回结果为 `APP_RESULT_PENDING`。APP 需要调用 `client_attr_ind_confirm()`以发送 confirmation。该流程中各层之间的交互如图 3-29 所示。

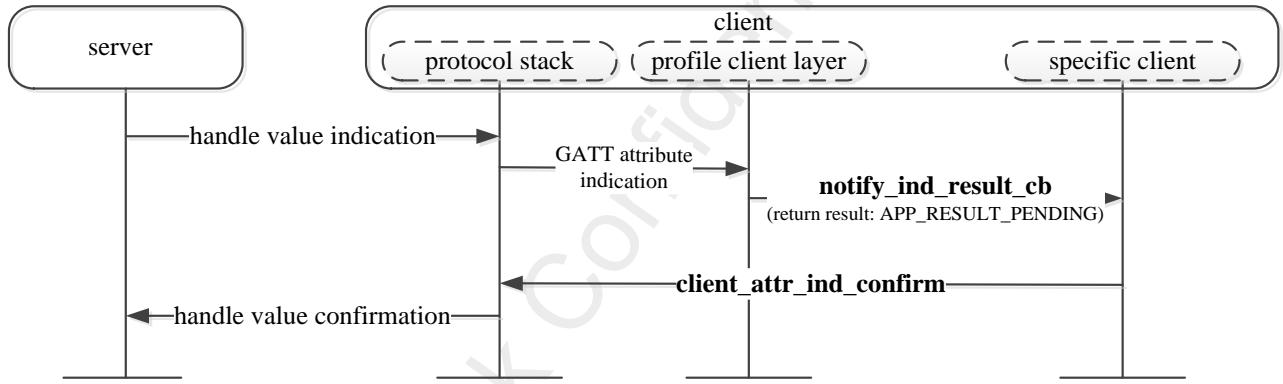


图 3-29 结果挂起的 Characteristic Value Indication 流程

profile client layer 未存储 service handle 信息，因此 profile client layer 无法确定发送该 indication 的 specific client。profile client layer 将调用所有注册的 specific clients 回调函数，因此 specific client 需要检查是否需要处理该 indication。

示例代码如下所示：

```

static T_APP_RESULT simp_ble_client_notif_ind_result_cb(uint8_t conn_id, bool notify,
    uint16_t handle, uint16_t value_size, uint8_t *p_value)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    T_SIMP_CLIENT_CB_DATA cb_data;
    uint16_t *hdl_cache;
    hdl_cache = simp_table[conn_id].hdl_cache;
    cb_data.cb_type = SIMP_CLIENT_CB_TYPE_NOTIF_IND_RESULT;
    if (handle == hdl_cache[HDL_SIMBLE_V3_NOTIFY])

```

```

{
    cb_data.cb_content.notif_ind_data.type = SIMP_V3_NOTIFY;
    cb_data.cb_content.notif_ind_data.data.value_size = value_size;
    cb_data.cb_content.notif_ind_data.data.p_value = p_value;
}
else if (handle == hdl_cache[HDL_SIMBLE_V4_INDICATE])
{
    cb_data.cb_content.notif_ind_data.type = SIMP_V4_INDICATE;
    cb_data.cb_content.notif_ind_data.data.value_size = value_size;
    cb_data.cb_content.notif_ind_data.data.p_value = p_value;
}
else
{
    return app_result;
}
/* Inform application the notif/ind result. */
if (simp_client_cb)
{
    app_result = (*simp_client_cb)(simp_client, conn_id, &cb_data);
}
return app_result;
}

```

3.2.3.8 Sequential Protocol

3.2.3.8.1 Request-response protocol

许多 ATT PDUs 是 sequential request-response protocol。一旦 client 向 server 发送 request，在收到该 server 发送的 response 之前，client 不能发送 request。Server 发送的 indication 同样是 sequential request-response protocol。

以下流程均是 sequential request-response protocol：

- Discovery 流程
- Read Characteristic Value By Handle 流程
- Read Characteristic Value By UUID 流程
- Write Characteristic Value 流程
- Write Long Characteristic Values 流程

在当前流程完成之前，APP 不能启动其它流程。否则，其它流程会启动失败。

当建立 connection 成功后时，蓝牙协议层可能会发送 exchange MTU request。GAP 层将发送 GAP_MSG_LE_CONN_MTU_INFO 消息以通知 APP，exchange MTU 流程已完成。在收到 GAP_MSG_LE_CONN_MTU_INFO 消息后，APP 可以启动以上流程。

```
void app_handle_conn_mtu_info_evt(uint8_t conn_id, uint16_t mtu_size)
```

```
{  
    APP_PRINT_INFO2("app_handle_conn_mtu_info_evt: conn_id %d, mtu_size %d", conn_id, mtu_size);  
    app_discov_services(conn_id, true);  
}
```

3.2.3.8.2 Commands

在 ATT 中，不要求 response 的 command 没有流控机制。

- Write Without Response
- Signed Write Without Response

由于资源有限，蓝牙协议层对 commands 采用流控机制。

在 GAP 层中维护 credits 数目实现对 Write Command 和 Signed Write Command 的流控，在收到蓝牙协议层的响应之前，允许 APP 发送 credits 笔 command。蓝牙协议层能够缓存 credits 笔 command 的数据。

- 当 profile client layer 向协议层发送 command 时，credits 数目减 1
- 当 command 发送给 server 时，协议层会发送响应给 profile client layer，credits 数目加 1
- credits 数目大于 0 时，才可以发送 command

回调函数 write_result_cb() 可以通知当前的 credits 数目。APP 也可以将参数类型设为 GAP_PARAM_LE_REMAIN_CREDITS，调用 le_get_gap_param() 函数获取 credits 数目。

```
void test(void)  
{  
    uint8_t wds_credits;  
    le_get_gap_param(GAP_PARAM_LE_REMAIN_CREDITS, &wds_credits);  
}
```

3.3 GATT Profile 用例

本节介绍 GATT Profile 接口的使用方法，提供一些典型用例。

3.3.1 ANCS Client

示例代码位于 BLE peripheral 工程中。

在 ancs.c 和 ancs.h 中定义 ANCS 程序。若其它 APP 需要使用该功能，流程如下所示。

1. 将 ancs.h 和 ancs.c 拷贝到 APP 目录中。
2. 将参数 auth_sec_req_enable 配置为 true。

```
void app_le_gap_init(void)  
{  
#if F_BT_ANCS_CLIENT_SUPPORT  
    uint8_t auth_sec_req_enable = true;  
#else  
    uint8_t auth_sec_req_enable = false;  
#endif
```

```
uint16_t auth_sec_req_flags = GAP_AUTHEN_BIT_BONDING_FLAG;  
.....  
}
```

3. 在 app_le_profile_init()中初始化 ANCS Client。

```
void app_le_profile_init(void)  
{  
    .....  
#if F_BT_ANCS_CLIENT_SUPPORT  
    client_init(1);  
    ancs_init(APP_MAX_LINKS);  
#endif  
}
```

4. 当 authentication 流程完成后，启动 ANCS discovery 流程。

```
void app_handle_authen_state_evt(uint8_t conn_id, uint8_t new_state, uint16_t cause)  
{  
    .....  
    case GAP_AUTHEN_STATE_COMPLETE:  
    {  
        if (cause == GAP_SUCCESS)  
        {  
#if F_BT_ANCS_CLIENT_SUPPORT  
            ancs_start_discovery(conn_id);  
#endif  
            APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_COMPLETE  
                            pair success");  
        }  
        else  
        {  
            APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_COMPLETE  
                            pair failed");  
        }  
    }  
    break;  
    .....  
}
```

5. 在 ancs.c 中自定义 ANCS 程序。

4 BLE 示例工程

使用 LE physical transport 的设备可以定义为四种 GAP 角色，SDK 提供相应的示例 APP 供用户参考。

1. Broadcaster
 - 1) 发送 advertisement
 - 2) 不能建立 connection
 - 3) 示例 APP: **BLE Broadcaster Application**
2. Observer
 - 1) 对 advertisement 进行 scan
 - 2) 不能发起 connection
 - 3) 示例 APP: **BLE Observer Application**
3. Peripheral
 - 1) 发送 advertisement
 - 2) 作为 slave 角色建立一条 LE 链路
 - 3) 示例 APP: **BLE Peripheral Application**
4. Central
 - 1) 对 advertisement 进行 scan
 - 2) 作为 master 角色发起 connection
 - 3) 示例 APP: **BLE Central Application**
5. 多重角色
 - 1) Broadcaster、Observer、Peripheral 和 Central
 - 2) 示例 APP: **BLE Scatternet Application**
6. 使用 LE Advertising Extensions 的 Peripheral
 - 1) 使用 LE Advertising Extensions 发送 advertisement
 - 2) 同时使能一个或多个 advertising set
 - 3) 设定 advertising set 使能的 duration 或最大的 extended advertising events 数目
 - 4) 使用 extended advertising PDUs 传输更多数据 (使用 LE Advertising Extensions 的 observer 或 central 作为对端设备)
 - 5) 可以作为 slave 角色建立一条 LE 链路, PHY 为 LE 1M PHY; PHY 为 LE 2M PHY 或 LE Coded PHY (使用 LE Advertising Extensions 的 central 作为对端设备)
 - 6) 示例 APP: **BLE BT5 Peripheral Application**
7. 使用 LE Advertising Extensions 的 Central
 - 1) 在 primary advertising channel (LE 1M PHY or/and LE Coded PHY) 对 advertising 数据包进行 extended scan

- 2) 设定 scan 的 duration 或 period scan
 - 3) 可以作为 master 角色发起一条 LE 链路, PHY 为 LE 1M PHY; PHY 为 LE 2M PHY 或 LE Coded PHY (使用 LE Advertising Extensions 的 peripheral 作为对端设备)
 - 4) 示例 APP: *BLE BT5 Central Application*
8. Peripheral + Privacy
- 1) 发送 advertisement
 - 2) 作为 slave 角色建立一条 LE 链路
 - 3) 当对端设备使用 resolvable private address 时, 可以使用 white list
 - 4) 示例 APP: *BLE Peripheral Privacy Application*

4.1 BLE Broadcaster Application

4.1.1 简介

本节内容是 BLE broadcaster application 的概述。BLE broadcaster 工程实现简单的 BLE broadcaster 设备, 可以作为开发基于 broadcaster 角色的 APP 的框架。

1. Broadcaster 角色的特征:

- 1) 发送 advertising 数据包
- 2) 不能建立 connection

2. 可配特征:

- 1) DLPS

配置 F_BT_DLPS_EN 打开该功能(默认打开)

4.1.2 工程概述

本节内容介绍工程的路径和结构, 相关文件路径如下所示:

- 工程路径为 sdk\board\evb\ble_broadcaster
- 工程源代码路径为 sdk\src\sample\ble_broadcaster

工程目录结构如图 4-1 所示:

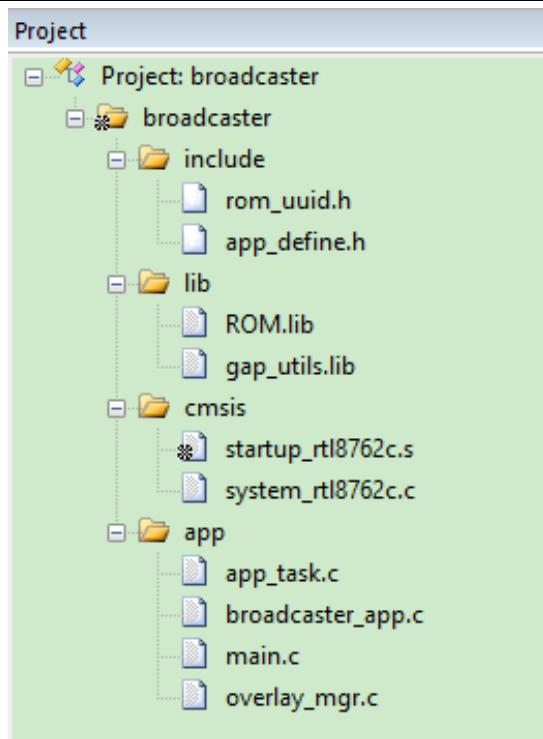


图 4-1 Broadcaster 工程目录结构

文件列表可分为以下四类。

表 4-1 Broadcaster 工程文件列表

Directory	Description
include	rom_uuid.h: ROM UUID header files. User need not modify.
lib	The protocol stack and gap library file. User need not modify.
cmsis	The cmsis source code. User need not modify.
app	The application source code.

4.1.3 源代码概述

以下章节介绍该 APP 的重要组成部分。

4.1.3.1 初始化

当上电或芯片重启后，main()函数会被调用，并执行以下初始化函数：

```

int main(void)
{
    board_init();
    le_gap_init();
    gap_lib_init();
    app_le_gap_init();
  
```

```
pwr_mgr_init();
task_init();
os_sched_start();
return 0;
}
```

app_le_gap_init()函数用于初始化 GAP 参数，用户可通过修改以下参数来自定义 APP。更多信息参见 [Advertising 参数的配置](#)。

```
void app_le_gap_init(void)
{
    /* Advertising parameters */
    uint8_t adv_evt_type = GAP_ADTYPE_ADV_NONCONN_IND;
    uint8_t adv_direct_type = GAP_REMOTE_ADDR_LE_PUBLIC;
    uint8_t adv_direct_addr[GAP_BD_ADDR_LEN] = {0};
    uint8_t adv_chann_map = GAP_ADVCHAN_ALL;
    uint8_t adv_filter_policy = GAP_ADV_FILTER_ANY;
    uint16_t adv_int_min = DEFAULT_ADVERTISING_INTERVAL_MIN;
    uint16_t adv_int_max = DEFAULT_ADVERTISING_INTERVAL_MAX;
    .....
}
```

使用 broadcaster 角色时，设备只能通过 non-connectable advertising 发送数据。因此 adv_evt_type 参数必须配置为 GAP_ADTYPE_ADV_NONCONN_IND 或 GAP_ADTYPE_ADV_SCAN_IND。更多关于 GAP 的初始化和启动流程的信息参见 [GAP 的初始化和启动流程](#)。

4.1.3.2 GAP 消息处理

一旦收到 GAP message，app_handle_gap_msg()将会被调用。更多关于 GAP message 的信息参见 [BLE GAP 消息](#)。

当收到 GAP_INIT_STATE_STACK_READY 消息时，broadcaster APP 将调用 le_adv_start()启动 advertising。若此时 BLE broadcaster application 在 evolution board 上运行，设备发送 non-connectable advertising 数据包。

```
void app_handle_dev_state_evt(T_GAP_DEV_STATE new_state, uint16_t cause)
{
    APP_PRINT_INFO3("app_handle_dev_state_evt: init state %d, adv state %d, cause 0x%x",
                   new_state.gap_init_state, new_state.gap_adv_state, cause);
    if (gap_dev_state.gap_init_state != new_state.gap_init_state)
    {
        if (new_state.gap_init_state == GAP_INIT_STATE_STACK_READY)
        {
            APP_PRINT_INFO0("GAP stack ready");
            /*stack ready*/
    }
}
```

```
        le_adv_start();  
    }  
}  
}
```

4.1.4 APP 的可配置功能

在 app_flags.h 中定义 APP 的可配置功能。

```
/** @brief Config DLPS: 0-Disable DLPS, 1-Enable DLPS */  
#define F_BT_DLPS_EN           1
```

关于 DLPS 的信息参见 DLPS 相关文档^[3]。

4.1.5 测试步骤

首先，编译并将 BLE Broadcaster application 下载到 evolution board。BLE Broadcaster application 的基本功能如上所述，为实现其它复杂功能，用户可以参考 SDK 提供的使用手册和源代码进行开发。当 BLE Broadcaster application 在 evolution board 上运行时，设备发送 non-connectable advertising 数据包。用户可以使用 air sniffer 或 observer application 查看 advertising 数据包。

4.1.5.1 与 BLE Observer Application 设备测试

测试步骤参见[与 BLE Broadcaster Application 设备测试](#)。

4.2 BLE Observer Application

4.2.1 简介

本节内容是 BLE observer application 的概述。BLE observer 工程实现简单的 BLE observer 设备，可以作为开发基于 observer 角色的 APP 的框架。

1. Observer 角色的特征:

- 1) 接收 advertising 数据包
- 2) 不能发起 connection

2. 可配特征:

- 1) DLPS
配置 F_BT_DLPS_EN 打开该功能(默认打开)

4.2.2 工程概述

本节内容介绍工程的路径和结构，相关文件路径如下所示：

- 工程路径为 sdk\board\evb\ble_observer
- 工程源代码路径为 sdk\src\sample\ble_observer

工程目录结构如图 4-2 所示：

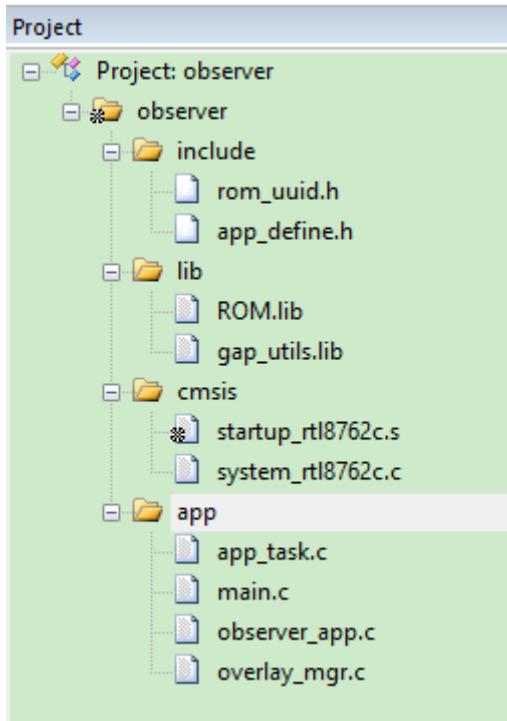


图 4-2 Observer 工程目录结构

文件列表可分为以下四类。

表 4-2 Observer 工程文件列表

Directory	Description
include	rom_uuid.h: ROM UUID header files. User need not modify.
lib	The protocol stack and gap library file. User need not modify.
cmsis	The cmsis source code. User need not modify.
app	The application source code.

4.2.3 源代码概述

以下章节介绍该 APP 的重要组成部分。

4.2.3.1 初始化

当上电或芯片重启后， main() 函数会被调用，并执行以下初始化函数：

```
int main(void)
{
```

```
board_init();
le_gap_init();
gap_lib_init();
app_le_gap_init();
pwr_mngr_init();
task_init();
os_sched_start();
return 0;
}
```

app_le_gap_init()函数用于初始化 scan 参数，用户可通过修改以下参数来自定义 APP。更多信息参见 [Scan 参数的配置](#)。

```
void app_le_gap_init(void)
{
    /* Scan parameters */
    uint8_t scan_mode = GAP_SCAN_MODE_PASSIVE;
    uint16_t scan_interval = DEFAULT_SCAN_INTERVAL;
    uint16_t scan_window = DEFAULT_SCAN_WINDOW;
    uint8_t scan_filter_policy = GAP_SCAN_FILTER_ANY;
    uint8_t scan_filter_duplicate = GAP_SCAN_FILTER_DUPLICATE_ENABLE;
    .....
}
```

更多关于 GAP 的初始化和启动流程的信息参见 [GAP 的初始化和启动流程](#)。

4.2.3.2 GAP 消息处理

一旦收到 GAP message，app_handle_gap_msg()将会被调用。更多关于 GAP message 的信息参见 [BLE GAP 消息](#)。

当收到 GAP_INIT_STATE_STACK_READY 消息时，observer APP 将调用 le_scan_start()启动 scan。若此时 BLE observer application 在 evolution board 上运行，设备将启动 scan。

```
void app_handle_dev_state_evt(T_GAP_DEV_STATE new_state, uint16_t cause)
{
    if (gap_dev_state.gap_init_state != new_state.gap_init_state)
    {
        if (new_state.gap_init_state == GAP_INIT_STATE_STACK_READY)
        {
            APP_PRINT_INFO0("GAP stack ready");
            /*stack ready*/
            le_scan_start();
        }
    }
    .....
}
```

4.2.3.3 GAP 回调函数处理

app_gap_callback()用于处理 GAP 回调函数消息，更多关于 GAP 回调函数的信息参见 [BLE GAP 回调函数](#)。

当设备处于 scanning 模式时，设备可以接收 advertising 数据。当收到 advertising 数据或 scan response 数据时，GAP 将发送 GAP_MSG_LE_SCAN_INFO 消息给 APP。

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;

    switch (cb_type)
    {
        case GAP_MSG_LE_SCAN_INFO:
            APP_PRINT_TRACE5("GAP_MSG_LE_SCAN_INFO: bd_addr %s, bdtype %d, event 0x%x, rssi %d,
                len %d",
                TRACE_BDADDR(p_data->p_le_scan_info->bd_addr),
                p_data->p_le_scan_info->remote_addr_type,
                p_data->p_le_scan_info->adv_type,
                p_data->p_le_scan_info->rssi,
                p_data->p_le_scan_info->data_len);

            /* User can split interested information by using the function as follow. */
            app_parse_scan_info(p_data->p_le_scan_info);
            break;
    }
}
```

app_parse_scan_info()函数是解析 advertising 数据和 scan response 数据的示例函数。

4.2.4 APP 的可配置功能

在 app_flags.h 中定义 APP 的可配置功能。

```
/** @brief Config DLPS: 0-Disable DLPS, 1-Enable DLPS */
#define F_BT_DLPS_EN 1
```

用户可以通过修改宏定义的值以打开或关闭该功能，可以将相关代码拷贝到其它 APP。

关于 DLPS 的信息参见 DLPS 相关文档^[3]。

4.2.5 测试步骤

首先，编译并将 BLE Observer application 下载到 evolution board。BLE Observer application 的基本功能如上所述，为实现其它复杂功能，用户可以参考 SDK 提供的使用手册和源代码进行开发。

可以通过 DebugAnalyser Tool 获取以下 log。当 BLE Observer application 在 evolution board 上运行时，

BLE 设备发送将启动 scan。Log 如下所示：

```
[APP] ***GAP scan start
```

若本地设备收到 advertising 数据或 scan response 数据, APP 将收到 GAP_MSG_LE_SCAN_INFO 消息。

Log 如下所示：

```
[APP] GAP_MSG_LE_SCAN_INFO: bd_addr CC::17::73::36::05::CF, bdtype 1, event 0x0, rssi -81, len 31  
[APP] app_parse_scan_info: AD Structure Info: AD type 0x1, AD Data Length 1  
[APP] ***GAP_ADTYPE_FLAGS: 0x4
```

4.2.5.1 与 BLE Broadcaster Application 设备测试

测试前准备两块 evolution boards, 分别运行 BLE Observer Application 和 BLE Broadcaster Application。

使用 broadcaster application 的设备的 log 如下所示：

```
[APP] ***GAP adv start
```

当收到 broadcaster application 的广播后, observer application 的 log 如下所示。

```
[APP] GAP_MSG_LE_SCAN_INFO: bd_addr 00::80::25::49::78::43, bdtype 0, event 0x3, rssi -18, len 20  
[APP] app_parse_scan_info: AD Structure Info: AD type 0x1, AD Data Length 1  
[APP] ***GAP_ADTYPE_FLAGS: 0x4  
[APP] app_parse_scan_info: AD Structure Info: AD type 0x9, AD Data Length 15  
[APP] ***GAP_ADTYPE_LOCAL_NAME_XXX: BLE_BROADCASTER
```

4.3 BLE Peripheral Application

4.3.1 简介

本节内容是 BLE peripheral application 的概述。BLE peripheral 工程实现简单的 BLE peripheral 设备, 可以作为开发基于 peripheral 角色的 APP 的框架。

1. Peripheral 角色的特征:

- 1) 发送 advertising 数据包
- 2) 作为 slave 角色建立 LE 链路

2. 可配特征:

- 1) DLPS
配置 F_BT_DLPS_EN 打开该功能(默认打开)
- 2) Link 数目

app_flags.h 中的 APP_MAX_LINKS(默认支持 1 条 link)

3) 支持的 GATT services

GAP Service 和 GATT Service (mandatory)、Battery Service、Simple BLE Service

4) 支持的 GATT clients

ANCS Client: 配置 F_BT_ANCS_CLIENT_SUPPORT 打开该功能(默认关闭)

4.3.2 工程概述

本节内容介绍工程的路径和结构，相关文件路径如下所示：

- 工程路径为 sdk\board\evb\ble_peripheral
- 工程源代码路径为 sdk\src\sample\ble_peripheral

工程目录结构如图 4-3 所示：

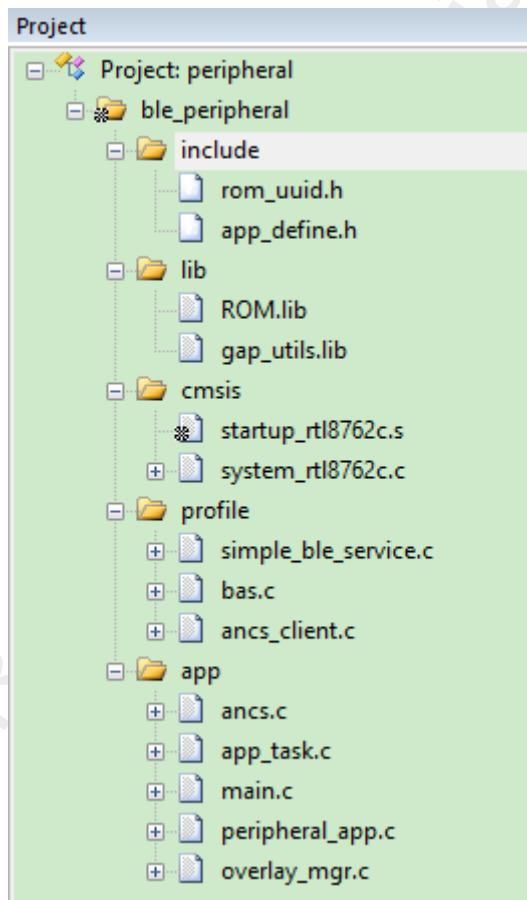


图 4-3 Peripheral 工程目录结构

文件列表分为以下五类。

表 4-3 Peripheral 工程文件列表

Directory	Description
-----------	-------------

include	rom_uuid.h: ROM UUID header files. User need not modify.
lib	The protocol stack and gap library file. User need not modify.
cmsis	The cmsis source code. User need not modify.
profile	The GATT profiles source code.
app	The application source code.

4.3.3 源代码概述

以下章节介绍该 APP 的重要组成部分。

4.3.3.1 初始化

当上电或芯片重启后，main()函数会被调用，并执行以下初始化函数：

```
int main(void)
{
    board_init();
    le_gap_init(APP_MAX_LINKS);
    gap_lib_init();
    app_le_gap_init();
    app_le_profile_init();
    pwr_mgr_init();
    task_init();
    os_sched_start();
    return 0;
}
```

GAP 和 GATT Profiles 初始化流程如下所示：

1. le_gap_init() - 初始化 GAP 并设置 link 数目
2. gap_lib_init() - 初始化 GAP lib，更多信息参见 [GAP Lib](#)
3. app_le_gap_init() - GAP 参数的初始化，用户可以通过修改以下参数自定义 application
 - 1) [Device Name 和 Device Appearance 的配置](#)
 - 2) [Advertising 参数的配置](#)
 - 3) [Bond Manager 参数的配置](#)
 - 4) [其它参数的配置](#)
4. app_le_profile_init() - 初始化基于 GATT 的 Profile

更多关于 GAP 的初始化和启动流程的信息参见 [GAP 的初始化和启动流程](#)。

4.3.3.2 GAP 消息处理

一旦收到 GAP message，app_handle_gap_msg()将会被调用。更多关于 GAP message 的信息参见 [BLE](#)

GAP 消息。

当收到 GAP_INIT_STATE_STACK_READY 消息时, peripheral APP 将调用 le_adv_start()启动 advertising。若此时 BLE peripheral application 在 evolution board 上运行, 设备将是可连接的。对端设备可以对 peripheral 设备进行 scan, 并创建 connection。

```
void app_handle_dev_state_evt(T_GAP_DEV_STATE new_state, uint16_t cause)
{
    if (gap_dev_state.gap_init_state != new_state.gap_init_state)
    {
        if (new_state.gap_init_state == GAP_INIT_STATE_STACK_READY)
        {
            APP_PRINT_INFO0("GAP stack ready");
            /*stack ready*/
            le_adv_start();
        }
    }
    .....
}
```

当 peripheral APP 收到 GAP_CONN_STATE_DISCONNECTED 消息时, APP 将调用 le_adv_start()启动 advertising。在连接断开之后, peripheral APP 将恢复为可连接状态。

```
void app_handle_conn_state_evt(uint8_t conn_id, T_GAP_CONN_STATE new_state, uint16_t disc_cause)
{
    switch (new_state)
    {
        case GAP_CONN_STATE_DISCONNECTED:
        {
            if ((disc_cause != (HCI_ERR | HCI_ERR_REMOTE_USER_TERMINATE))
                && (disc_cause != (HCI_ERR | HCI_ERR_LOCAL_HOST_TERMINATE)))
            {
                APP_PRINT_ERROR1("app_handle_conn_state_evt: connection lost cause 0x%x", disc_cause);
            }
            le_adv_start();
        }
        break;
    }
    .....
}
```

4.3.3.3 GAP 回调函数处理

app_gap_callback()用于处理 GAP 回调函数消息, 更多关于 GAP 回调函数的信息参见 [BLE GAP 回调函数](#)。

4.3.3.4 Profile 消息回调函数

当 APP 使用 xxx_add_service 注册 specific service 时，APP 需要注册回调函数以处理 specific service 的消息。APP 需要调用 server_register_app_cb 注册回调函数以处理 profile server layer 的信息。

APP 可以针对不同的 services 注册不同的回调函数，也可以注册通用回调函数以处理 specific services 和 profile server layer 的消息。

app_profile_callback()是通用回调函数，根据 service id 区分不同的 services。

```
void app_le_profile_init(void)
{
    server_init(2);
    simp_svr_id = simp_ble_service_add_service(app_profile_callback);
    bas_svr_id = bas_add_service(app_profile_callback);
    server_register_app_cb(app_profile_callback);
}
```

1. 通用 profile server 回调函数

SERVICE_PROFILE_GENERAL_ID 是 profile server layer 使用的 service id。profile server layer 使用的消息包含以下两种消息类型：

- 1) PROFILE_EVT_SRV_REG_COMPLETE: 在 GAP 启动流程中完成 service 注册流程。
- 2) PROFILE_EVT_SEND_DATA_COMPLETE: profile server layer 使用该消息向 APP 通知发送 notification/indication 的结果。

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    if (service_id == SERVICE_PROFILE_GENERAL_ID)
    {
        T_SERVER_APP_CB_DATA *p_param = (T_SERVER_APP_CB_DATA *)p_data;
        switch (p_param->eventId)
        {
            case PROFILE_EVT_SRV_REG_COMPLETE:// srv register result event.
                APP_PRINT_INFO1("PROFILE_EVT_SRV_REG_COMPLETE: result %d",
                               p_param->event_data.service_reg_result);
                break;
            case PROFILE_EVT_SEND_DATA_COMPLETE:
                break;
        }
    }
}
```

2. Battery Service

bas_svr_id 是 battery service 的 service id。

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
```

```
T_APP_RESULT app_result = APP_RESULT_SUCCESS;  
.....  
else if (service_id == bas_srv_id)  
{  
    T_BAS_CALLBACK_DATA *p_bas_cb_data = (T_BAS_CALLBACK_DATA *)p_data;  
    switch (p_bas_cb_data->msg_type)  
    {  
        case SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION:  
        .....  
    }  
}
```

3. Simple BLE Service

simp_srv_id 是 simple BLE service 的 service id。

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)  
{  
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;  
    .....  
    else if (service_id == simp_srv_id)  
    {  
        TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;  
        switch (p_simp_cb_data->msg_type)  
        {  
            case SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION:  
            .....  
        }  
    }  
}
```

4.3.4 APP 的可配置功能

在 app_flags.h 中定义 APP 的可配置功能。

```
/** @brief Config APP LE link number */  
#define APP_MAX_LINKS 1  
/** @brief Config DLPS: 0-Disable DLPS, 1-Enable DLPS */  
#define F_BT_DLPS_EN 1  
/** @brief Config ANCS Client: 0-Not built in, 1-Open ANCS client function */  
#define F_BT_ANCS_CLIENT_SUPPORT 0  
#define F_BT_ANCS_APP_FILTER (F_BT_ANCS_CLIENT_SUPPORT & 1)  
#define F_BT_ANCS_GET_APP_ATTR (F_BT_ANCS_CLIENT_SUPPORT & 0)  
/** @brief Config ANCS Client debug log: 0-close, 1-open */  
#define F_BT_ANCS_CLIENT_DEBUG (F_BT_ANCS_CLIENT_SUPPORT & 0)
```

4.3.4.1 DLPS

关于 DLPS 的信息参见 DLPS 相关文档^[3]。

4.3.4.2 ANCS Client

配置 F_BT_ANCS_CLIENT_SUPPORT 以使用 ANCS Client, 更多关于 ANCS Client 的信息参见 [ANCS Client](#)。

4.3.5 测试步骤

首先, 编译并将 BLE Peripheral application 下载到 evolution board。BLE Peripheral application 的基本功能如上所述, 为实现其它复杂功能, 用户可以参考 SDK 提供的使用手册和源代码进行开发。

当 BLE Peripheral application 在 evolution board 上运行时, 设备将是可连接的。对端设备可以对 peripheral 设备进行 scan, 并创建 connection。在连接断开之后, peripheral APP 将恢复为可连接状态。

4.3.5.1 与 iOS 设备测试

步骤简介: 基于 iOS 的设备与 BLE 兼容, 因此可以 discover 运行 BLE Peripheral Application 的设备。推荐从 App Store 下载 BLE 相关的 APP(例如 LightBlue) 以执行 scan 和 connection 测试。

测试步骤: 在 iOS 设备上运行 LightBlue 进行 scan, 与 BLE_PERIPHERAL 设备创建 connection, 如图 4-4 所示:

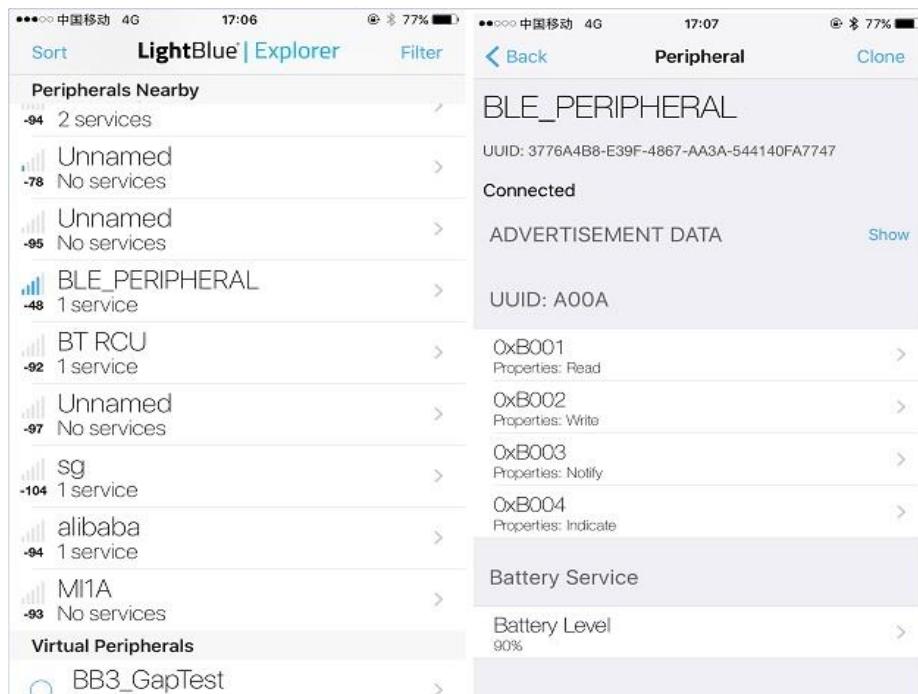


图 4-4 与 iOS 设备测试

4.3.5.2 与 BLE Central Application 设备测试

可以使用 BLE central application 与 BLE peripheral application 进行测试，更多关于 central application 的信息参见 [BLE Central Application](#)。测试流程参见 [与 BLE Peripheral Application 设备测试](#)。

4.4 BLE Central Application

4.4.1 简介

本节内容是 BLE central application 的概述。BLE central 工程实现简单的 BLE central 设备，可以作为开发基于 central 角色的 APP 的框架。

1. Central 角色的特征：

- 1) 对 advertising 数据包进行 scan
- 2) 作为 master 角色发起 connection
- 3) 与多个 peripheral 设备创建 connection

2. 可配特征：

- 1) Link 数目
app_flags.h 中的 APP_MAX_LINKS(默认为 2 条 link)
- 2) GATT services 的 handle 信息的存储
配置 F_BT_GATT_SRV_HANDLE_STORAGE 打开该功能(默认关闭)
- 3) 支持的 GATT services
GAP Service 和 GATT Service (mandatory)
- 4) 支持的 GATT clients
GAP Service Client、Simple BLE Service Client、Battery Service Client

4.4.2 工程概述

本节内容介绍工程的路径和结构，相关文件路径如下所示：

- 工程路径为 sdk\board\evb\ble_central
- 工程源代码路径为 sdk\src\sample\ble_central

工程目录结构如图 4-5 所示：

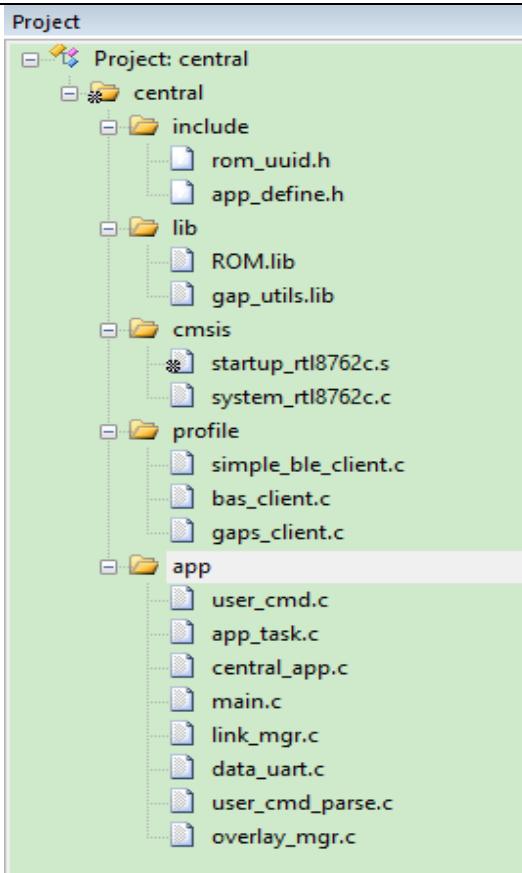


图 4-5 Central 工程目录结构

文件列表分为以下五类。

表 4-4 Central 工程文件列表

Directory	Description
include	rom_uuid.h: ROM UUID header files. User need not modify.
lib	The protocol stack and gap library file. User need not modify.
cmsis	The cmsis source code. User need not modify.
profile	The GATT profiles source code.
app	The application source code.

4.4.3 源代码概述

以下章节介绍该 APP 的重要组成部分。

4.4.3.1 初始化

当上电或芯片重启后，main()函数会被调用，并执行以下初始化函数：

```
int main(void)
{
```

```
board_init();
le_gap_init(APP_MAX_LINKS);
gap_lib_init();
app_le_gap_init();
app_le_profile_init();
pwr_mngr_init();
task_init();
os_sched_start();

return 0;
}
```

GAP 和 GATT Profiles 初始化流程如下所示：

1. le_gap_init() - 初始化 GAP 并设置 link 数目
2. gap_lib_init() - 初始化 GAP lib，更多信息参见 [GAP Lib](#)
3. app_le_gap_init() - GAP 参数的初始化，用户可以通过修改以下参数自定义 application
 - 1) [Device Name 和 Device Appearance 的配置](#)
 - 2) [Scan 参数的配置](#)
 - 3) [Bond Manager 参数的配置](#)
4. app_le_profile_init() - 初始化基于 GATT 的 Profile

更多关于 GAP 的初始化和启动流程的信息参见 [GAP 的初始化和启动流程](#)。

4.4.3.2 多链路管理

在 link_mngr.h 中定义 APP 链路管理模块。

```
typedef struct {
    T_GAP_CONN_STATE          conn_state;
    uint8_t                    discovered_flags;
    uint8_t                    srv_found_flags;
    T_GAP_REMOTE_ADDR_TYPE   bd_type;
    uint8_t                    bd_addr[GAP_BD_ADDR_LEN];
} T_APP_LINK;
extern T_APP_LINK app_link_table[APP_MAX_LINKS];
```

app_link_table 用于保存链路相关信息。

4.4.3.3 GAP 消息处理

一旦收到 GAP message， app_handle_gap_msg() 将会被调用。更多关于 GAP message 的信息参见 [BLE GAP 消息](#)。

当收到 GAP_MSG_LE_CONN_STATE_CHANGE 消息时， app_handle_conn_state_evt() 将更新 app_link_table 中的信息。

```
void app_handle_conn_state_evt(uint8_t conn_id, T_GAP_CONN_STATE new_state, uint16_t disc_cause)
{
    if (conn_id >= APP_MAX_LINKS)
    {
        return;
    }

    APP_PRINT_INFO4("app_handle_conn_state_evt: conn_id %d, conn_state(%d -> %d), disc_cause 0x%x",
                    conn_id, app_link_table[conn_id].conn_state, new_state, disc_cause);
    app_link_table[conn_id].conn_state = new_state;
    switch (new_state)
    {
        case GAP_CONN_STATE_DISCONNECTED:
        {
            if ((disc_cause != (HCI_ERR | HCI_ERR_REMOTE_USER_TERMINATE))
                && (disc_cause != (HCI_ERR | HCI_ERR_LOCAL_HOST_TERMINATE)))
            {
                APP_PRINT_ERROR2("app_handle_conn_state_evt: connection lost, conn_id %d, cause 0x%x",
                                conn_id, disc_cause);
            }

            data_uart_print("Disconnect conn_id %d\r\n", conn_id);
            memset(&app_link_table[conn_id], 0, sizeof(T_APP_LINK));
        }
        break;

        case GAP_CONN_STATE_CONNECTED:
        {
            le_get_conn_addr(conn_id, app_link_table[conn_id].bd_addr,
                            &app_link_table[conn_id].bd_type);
            data_uart_print("Connected success conn_id %d\r\n", conn_id);
        }
        break;
        default:
        break;
    }
}
```

4.4.3.4 GAP 回调函数处理

app_gap_callback()用于处理 GAP 回调函数消息，更多关于 GAP 回调函数的信息参见 [BLE GAP 回调函数](#)。

4.4.3.5 Profile 消息回调函数

当 APP 使用 xxx_add_client() 注册 specific client 时，APP 需要注册回调函数以处理 specific client 的消息。

APP 可以针对不同的 clients 注册不同的回调函数，也可以注册通用回调函数以处理 specific clients 的消息。

app_client_callback() 是通用回调函数，根据 client id 区分不同的 clients。

```
void app_le_profile_init(void)
{
    client_init(3);
    gaps_client_id = gaps_add_client(app_client_callback, APP_MAX_LINKS);
    simple_ble_client_id = simp_ble_add_client(app_client_callback, APP_MAX_LINKS);
    bas_client_id = bas_add_client(app_client_callback, APP_MAX_LINKS);
}
```

更多信息参见 [BLE Profile Client](#)。

1. GAP service client

gaps_client_id 是 GAP service client 的 client id。

```
T_APP_RESULT app_client_callback(T_CLIENT_ID client_id, uint8_t conn_id, void *p_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    APP_PRINT_INFO2("app_client_callback: client_id %d, conn_id %d",
                    client_id, conn_id);
    if (client_id == gaps_client_id)
    {
        T_GAPS_CLIENT_CB_DATA *p_gaps_cb_data = (T_GAPS_CLIENT_CB_DATA *)p_data;
        switch (p_gaps_cb_data->cb_type)
        {
            case GAPS_CLIENT_CB_TYPE_DISC_STATE:
                .....
        }
    }
}
```

2. Battery Service Client

bas_client_id 是 battery service client 的 client id。

```
T_APP_RESULT app_client_callback(T_CLIENT_ID client_id, uint8_t conn_id, void *p_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    APP_PRINT_INFO2("app_client_callback: client_id %d, conn_id %d",
                    client_id, conn_id);
    else if (client_id == bas_client_id)
    {
        T_BAS_CLIENT_CB_DATA *p_bas_cb_data = (T_BAS_CLIENT_CB_DATA *)p_data;
        switch (p_bas_cb_data->cb_type)
```

```
{  
    case BAS_CLIENT_CB_TYPE_DISC_STATE:  
    .....  
}
```

3. Simple BLE Service Client

simple_ble_client_id 是 simple BLE service client 的 client id。

```
T_APP_RESULT app_client_callback(T_CLIENT_ID client_id, uint8_t conn_id, void *p_data)  
{  
    T_APP_RESULT result = APP_RESULT_SUCCESS;  
    APP_PRINT_INFO2("app_client_callback: client_id %d, conn_id %d",  
                    client_id, conn_id);  
    else if (client_id == simple_ble_client_id)  
    {  
        T_SIMP_CLIENT_CB_DATA *p_simp_client_cb_data = (T_SIMP_CLIENT_CB_DATA *)p_data;  
        uint16_t value_size;  
        uint8_t *p_value;  
        switch (p_simp_client_cb_data->cb_type)  
        {  
            case SIMP_CLIENT_CB_TYPE_DISC_STATE:  
            .....  
        }  
    }  
}
```

4.4.3.6 Discover GATT Services 流程

当收到消息 LE_GAP_MSG_TYPE_CONN_MTU_INFO 时，central application 将自动 discover services。在 app_discov_services() 中定义 Discover GATT Services 流程，具体信息参见 central_app.c 中的 app_discov_services()。

```
void app_handle_conn_mtu_info_evt(uint8_t conn_id, uint16_t mtu_size)  
{  
    APP_PRINT_INFO2("app_handle_conn_mtu_info_evt: conn_id %d, mtu_size %d", conn_id, mtu_size);  
    app_discov_services(conn_id, true);  
}
```

app_discov_services (uint8_t conn_id, bool start) 的流程图如图 4-6 所示：

app_discov_services() flow chart

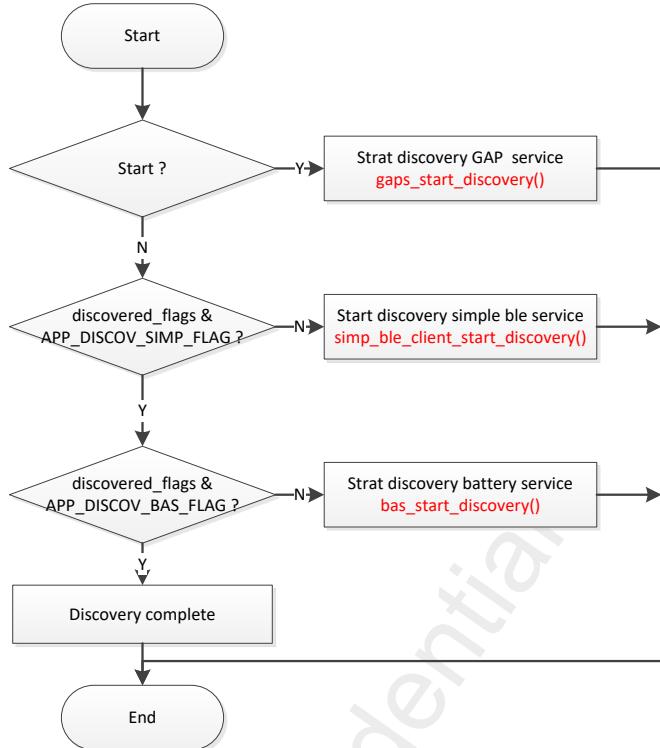


图 4-6 app_discov_services()流程图

当收到消息 DISC_GAPS_DONE 、 DISC_SIMP_DONE 和 DISC_BAS_DONE 时， APP 将调用 app_discov_services()。相关代码如下所示：

```

T_APP_RESULT app_client_callback(T_CLIENT_ID client_id, uint8_t conn_id, void *p_data)
{
    .....
    case DISC_GAPS_DONE:
        app_link_table[conn_id].discovered_flags |= APP_DISCOV_GAPS_FLAG;
        app_link_table[conn_id].srv_found_flags |= APP_DISCOV_GAPS_FLAG;
        app_discov_services(conn_id, false);
        /* Discovery Simple BLE service procedure successfully done. */
        APP_PRINT_INFO0("app_client_callback: discover gaps procedure done.");
        break;
    .....
    case DISC_SIMP_DONE:
        /* Discovery Simple BLE service procedure successfully done. */
        app_link_table[conn_id].discovered_flags |= APP_DISCOV_SIMP_FLAG;
        app_link_table[conn_id].srv_found_flags |= APP_DISCOV_SIMP_FLAG;
        app_discov_services(conn_id, false);
        APP_PRINT_INFO0("app_client_callback: discover simp procedure done.");
        break;
    .....
  
```

```

case DISC_BAS_DONE:
    /* Discovery BAS procedure successfully done. */
    app_link_table[conn_id].discovered_flags |= APP_DISCOV_BAS_FLAG;
    app_link_table[conn_id].srv_found_flags |= APP_DISCOV_BAS_FLAG;
    app_discov_services(conn_id, false);
    APP_PRINT_INFO0("app_client_callback: discover bas procedure done");
    break;
    .....
}

```

4.4.4 APP 的可配置功能

在 app_flags.h 中定义 APP 的可配置功能。

```

/** @brief Config APP LE link number */
#define APP_MAX_LINKS 2

/** @brief Config GATT services storage: 0-Not save, 1-Save to flash
 *
 * If configure to 1, the GATT services discovery results will save to the flash.
 */
#define F_BT_GATT_SRV_HANDLE_STORAGE 0

```

用户可以通过修改宏定义的值以打开或关闭该功能，可以将相关代码拷贝到其它 APP。

4.4.4.1 GATT services 的 handles 存储

相关代码如下所示：

```

typedef struct {
    uint8_t      srv_found_flags;
    uint8_t      bd_type;
    uint8_t      bd_addr[GAP_BD_ADDR_LEN];
    uint32_t     reserved;
    uint16_t     gaps_hdl_cache[HDL_GAPS_CACHE_LEN];
    uint16_t     simp_hdl_cache[HDL_SIMBLE_CACHE_LEN];
    uint16_t     bas_hdl_cache[HDL_BAS_CACHE_LEN];
} T_APP_SRVS_HDL_TABLE;

uint32_t app_save_srvs_hdl_table(T_APP_SRVS_HDL_TABLE *p_info)
uint32_t app_load_srvs_hdl_table(T_APP_SRVS_HDL_TABLE *p_info)

```

app_save_srvs_hdl_table()用于将 handles 信息保存到 Flash 中。

app_load_srvs_hdl_table()用于从 Flash 中载入 handles 信息。

app_discov_services()流程图如图 4-7 所示：

app_discov_services() flow chart

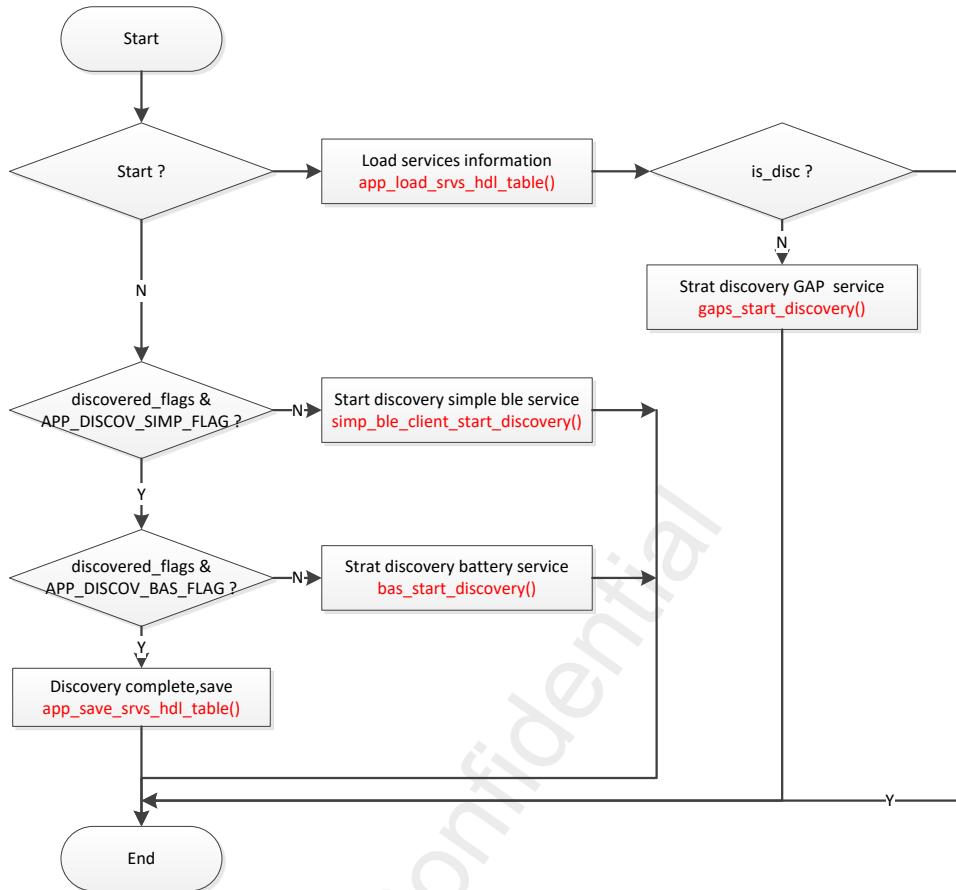


图 4-7 app_discov_services()流程图(F_BT_GATT_SRV_HANDLE_STORAGE)

4.4.5 用户命令

使用 BLE Central Application 时，需要通过个人电脑的 Data UART 输入命令进行交互。更多信息参见 [BLE Application 用户命令](#)。

4.4.6 测试步骤

首先，编译并将 BLE Central application 下载到 evolution board，通过 MPTool 配置链路数目。关于 LE Link 数目的内容参见 [LE Link 数目的配置](#)。BLE Central application 的基本功能如上所述，为实现其它复杂功能，用户可以参考 SDK 提供的使用手册和源代码进行开发。

通过使用 [BLE Application 用户命令](#)中描述的命令，GAP Central 角色和 GATT client 的流程如下所示。
通过 DebugAnalyser Tool 可获取以下 log。

4.4.6.1 与 BLE Peripheral Application 设备测试

可以使用 BLE Peripheral Application 与 BLE Central Application 进行测试，更多关于 peripheral 的信息参见 [BLE Peripheral Application](#)。

1. 一条链路测试

首先，编译并将 BLE Peripheral Application 下载到 evolution board。使用 MPTool 将蓝牙地址设置为"x00 x11 x22 x33 x44 x00"。

1) Scanning

步骤描述：查找附近处于可发现状态的 BLE 设备

步骤如下所示：

scan 0 - 启动 scan，查看附近处于可发现状态的 BLE 设备的信息

```
Log tool shows :
```

```
[APP] !**GAP scan start
```

```
[APP] GAP_MSG_LE_SCAN_INFO: bd_addr 00::11::22::33::44::00, bdtype 0, adv_type 0x0, rssi  
-17, data_len 23
```

stopscan - 停止 scan

```
Log tool shows :
```

```
[APP] !**GAP scan stop
```

showdev - 显示 scan 设备列表，该列表是通过 simple BLE service 的 UUID 过滤的

```
Serial port assistant tool shows :
```

```
Advertising and Scan response: filter uuid = 0xA00A dev list
```

```
RemoteBd[0] = [00:11:22:33:44:00] type = 0
```

2) Connection 和 Reconnection

步骤描述：与 peripheral 设备创建 connection，断开 connection

步骤如下所示：

conde 0 - 发起 connection

```
Serial port assistant tool shows :
```

```
Connected success conn_id 0
```

showcon - 显示 connection 信息

Serial port assistant tool shows :

ShowCon conn_id 0 state 0x00000002 role 1

RemoteBd = [00:11:22:33:44:00] type = 0

active link num 1, idle link num 1

conupdreq 0 1 - 发起 connection 参数更新请求

Log tool shows :

[APP] !**app_handle_conn_param_update_evt update success:conn_id 0, conn_interval 0xa, conn_slave_latency 0x0, conn_supervision_timeout 0x3e8

disc 0 - 与 peripheral 设备断开 connection

Serial port assistant tool shows :

Disconnect conn_id 0

condev 0 - 与 peripheral 设备重新建立 connection

Serial port assistant tool shows :

Connected success conn_id 0

3) Pair

步骤描述：与 peripheral 设备配对

步骤如下所示：

sauth 0 - 发送 authentication 请求

Serial port assistant tool shows :

Pair success

bondinfo - 显示绑定信息

Serial port assistant tool shows :

bond_dev[0]: bd 0x001122334400, addr_type 0, flags 0x000000bf

4) GATT services

步骤描述：GATT services 相关步骤

步骤如下所示：

gaphdl 0 - 打印获取的 GAP service 的 handle 列表

Serial port assistant tool shows :

- >Index 0 – Handle 0x00000005
- >Index 1 – Handle 0x0000000d
- >Index 2 – Handle 0x00000007
- >Index 3 – Handle 0x00000009
- >Index 4 – Handle 0x0000000d
- >Index 5 – Handle 0x00000000

gapread 0 0 - 读取 device name

Log tool shows :

```
[APP] **GAPS_READ_DEVICE_NAME: device name BLE_PERIPHERAL.
```

gapread 0 1 - 读取 device appearance

Log tool shows :

```
[APP] **GAPS_READ_APPEARANCE: appearance 0
```

Battery service 相关步骤:

bashdl 0 - 打印获取的 battery service 的 handle 列表

Serial port assistant tool shows :

- >Index 0 – Handle 0x00000019
- >->Index 1 – Handle 0x0000ffff
- >->Index 2 – Handle 0x0000001b
- >->Index 3 – Handle 0x0000001c

basread 0 0 – 读取 battery level 的值

Log tool shows :

```
[APP] **BAS_READ_BATTERY_LEVEL: battery level 90
```

basread 0 1 - 读取 battery level characteristic 的 CCCD

Log tool shows :

```
[APP] **BAS_READ_NOTIFY: notify 0
```

bascccd 0 1 - 写入 battery level characteristic 的 CCCD

Log tool shows :

```
[APP] !**BAS_WRITE_NOTIFY_ENABLE: write result 0x0
```

Simple BLE Service 相关步骤:

simpdh 0 - 打印获取的 simple BLE service 的 handle 列表

Serial port assistant tool shows :

```
->Index 0 – Handle 0x0000000e  
->Index 1 – Handle 0x00000018  
->Index 2 – Handle 0x00000010  
->Index 3 – Handle 0x00000012  
->Index 4 – Handle 0x00000014  
->Index 5 – Handle 0x00000015  
->Index 6 – Handle 0x00000017  
->Index 7 – Handle 0x00000018
```

simpread 0 0 0 - 通过 handle 读取 V1 characteristic 的值

Log tool shows :

```
[APP] !**SIMP_READ_V1_READ: value_size 2, value 01-02
```

simpread 0 0 1 - 通过 UUID 读取 V1 characteristic 的值

Log tool shows :

```
[APP] !**SIMP_READ_V1_READ: value_size 2, value 01-02
```

simpread 0 1 0 - 读取 V3 characteristic 的 CCCD

Log tool shows :

```
[APP] !**SIMP_READ_V3_NOTIFY_CCCD: notify 0
```

simpread 0 2 0 - 读取 V4 characteristic 的 CCCD

Log tool shows :

```
[APP] !**SIMP_READ_V4_INDICATE_CCCD: indicate 0
```

simpccc 0 0 1 - 写入 V3 characteristic 的 CCCD

Log tool shows :

```
[APP] !**SIMP_WRITE_V3_NOTIFY_CCCD: write result 0x0
```

simpwritev2 0 1 10 - 使用 write request 写入 V2 characteristic

Log tool shows :

```
[APP] !**SIMP_WRITE_V2_WRITE: write result 0x0
```

2. 两条链路测试

首先，编译并将 BLE Peripheral Application 下载到两块 evolution board。使用 MPTool 将蓝牙地址分别设置为"x00 x11 x22 x33 x44 x00"和"x00 x11 x22 x33 x44 x01"。

测试步骤：

scan 0 - 启动 scan，查看附近处于可发现状态的 BLE 设备的信息

Log tool shows :

```
[APP] !**GAP scan start
```

```
[APP] GAP_MSG_LE_SCAN_INFO: bd_addr 00::11::22::33::44::00, bdtype 0, adv_type 0x0, rssi -17, data_len 23
```

```
[APP] GAP_MSG_LE_SCAN_INFO: bd_addr 00::11::22::33::44::01, bdtype 0, adv_type 0x0, rssi -17, data_len 23
```

stopscan - 停止 scan

Log tool shows :

```
[APP] !**GAP scan stop
```

showdev - 显示 scan 设备列表，该列表是通过 simple BLE service 的 UUID 过滤的

Serial port assistant tool shows :

```
Advertising and Scan response: filter uuid = 0xA00A dev list
```

```
RemoteBd[0] = [00:11:22:33:44:00] type = 0
```

```
RemoteBd[1] = [00:11:22:33:44:01] type = 0
```

condev 0 - 发起与 index 为 0 的设备的 connection

Serial port assistant tool shows :

```
Connected success conn_id 0
```

condev 1 - 发起与 index 为 1 的设备的 connection

Serial port assistant tool shows :

Connected success conn_id 1

sauth 0 - 发送 authentication 请求(conn_id=0)

Serial port assistant tool shows :

Pair success

sauth 1 - 发送 authentication 请求(conn_id=1)

Serial port assistant tool shows :

Pair success

gapread 0 0 - 读取 device name (conn_id=0)

Log tool shows :

[APP] !**GAPS_READ_DEVICE_NAME: device name BLE_PERIPHERAL.

gapread 1 0 - 读取 device name (conn_id=1)

Log tool shows :

[APP] !**GAPS_READ_DEVICE_NAME: device name BLE_PERIPHERAL.

4.5 BLE Scatternet Application

4.5.1 简介

本节内容是 BLE scatternet application 的概述。BLE scatternet 工程支持多种角色，包括 broadcaster、observer、peripheral 和 central 角色，可以作为开发基于多种角色的 APP 的框架。

1. Scatternet 拓扑特征:

- 1) 在同一时间，作为 slave 角色与多个 master 建立 connection
- 2) 在同一时间，作为 master 角色和 slave 角色
- 3) 不支持角色转换

2. 可配特征:

- 1) Link 数目
app_flags.h 中的 APP_MAX_LINKS(默认为 2 条 link)
- 2) 支持的 GATT services

GAP Service 和 GATT Service (mandatory)、Simple BLE Service 和 Battery Service

3) 支持的 GATT clients

GAP Service Client、Simple BLE Service Client 和 Battery Service Client

4) Airplane mode 设置

配置 F_BT_AIRPLANE_MODE_SUPPORT 以打开该功能(默认关闭)

5) GAP Service characteristic 的可写属性

配置 F_BT_GAPS_CHAR_WRITEABLE 以打开该功能(默认关闭)

6) static random address 的使用

配置 F_BT_LE_USE_STATIC_RANDOM_ADDR 以打开该功能(默认关闭)

7) Physical channel 设置

配置 F_BT_LE_5_0_SET_PHY_SUPPORT 以打开该功能(默认关闭)

4.5.2 工程概述

本节内容介绍工程的路径和结构，相关文件路径如下所示：

- 工程路径为 sdk\board\evb\ble_scatternet
- 工程源代码路径为 sdk\src\sample\ble_scatternet

工程目录结构如图 4-8 所示：

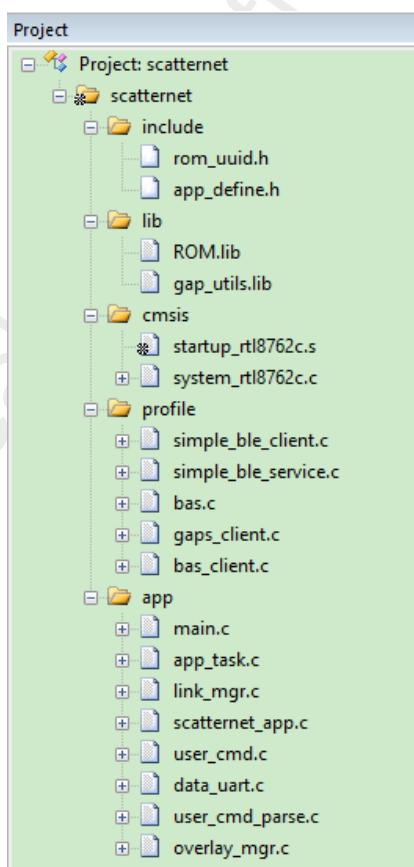


图 4-8 Scatternet 工程目录结构

文件列表分为以下五类。

表 4-5 Scatternet 工程文件列表

Directory	Description
include	rom_uuid.h: ROM UUID header files. User need not modify.
lib	The protocol stack and gap library file. User need not modify.
cmsis	The cmsis source code. User need not modify.
profile	The GATT profiles source code.
app	The application source code.

4.5.3 源代码概述

以下章节介绍该 APP 的重要组成部分。

4.5.3.1 初始化

当上电或芯片重启后，main()函数会被调用，并执行以下初始化函数：

```
int main(void)
{
    board_init();
    le_gap_init(APP_MAX_LINKS);
    gap_lib_init();
    app_le_gap_init();
    app_le_profile_init();
    pwr_mgr_init();
    task_init();
    os_sched_start();
    return 0;
}
```

GAP 和 GATT Profiles 初始化流程如下所示：

1. le_gap_init() - 初始化 GAP 并设置 link 数目
2. gap_lib_init() - 初始化 GAP lib，更多信息参见 [GAP Lib](#)
3. app_le_gap_init() - GAP 参数的初始化，用户可以通过修改以下参数自定义 application
 - 1) [Device Name 和 Device Appearance 的配置](#)
 - 2) [Advertising 参数的配置](#)
 - 3) [Scan 参数的配置](#)
 - 4) [Bond Manager 参数的配置](#)
4. app_le_profile_init() - 初始化基于 GATT 的 Profile

更多关于 GAP 的初始化和启动流程的信息参见 [GAP 的初始化和启动流程](#)。

4.5.3.2 多链路管理

在 link_mgr.h 中定义 APP 链路管理模块。

```
typedef struct {
    T_GAP_CONN_STATE          conn_state;
    T_GAP_REMOTE_ADDR_TYPE   bd_type;
    uint8_t                   bd_addr[GAP_BD_ADDR_LEN];
} T_APP_LINK;
extern T_APP_LINK app_link_table[APP_MAX_LINKS];
```

app_link_table 用于保存链路相关信息。

4.5.3.3 GAP 消息处理

一旦收到 GAP message， app_handle_gap_msg() 将会被调用。更多关于 GAP message 的信息参见 [BLE GAP 消息](#)。

当收到 GAP_MSG_LE_CONN_STATE_CHANGE 消息时， app_handle_conn_state_evt() 将更新 app_link_table 中的信息。

```
void app_handle_conn_state_evt(uint8_t conn_id, T_GAP_CONN_STATE new_state,
                                uint16_t disc_cause)
{
    if (conn_id >= APP_MAX_LINKS)
    {
        return;
    }
    APP_PRINT_INFO4("app_handle_conn_state_evt: conn_id %d, conn_state(%d -> %d), disc_cause 0x%x",
                   conn_id, app_link_table[conn_id].conn_state, new_state, disc_cause);
    app_link_table[conn_id].conn_state = new_state;
    switch (new_state)
    {
        case GAP_CONN_STATE_DISCONNECTED:
        {
            if ((disc_cause != (HCI_ERR | HCI_ERR_REMOTE_USER_TERMINATE))
                && (disc_cause != (HCI_ERR | HCI_ERR_LOCAL_HOST_TERMINATE)))
            {
                APP_PRINT_ERROR2("app_handle_conn_state_evt: connection lost, conn_id %d, cause 0x%x",
                               conn_id, disc_cause);
            }
            data_uart_print("Disconnect conn_id %d\r\n", conn_id);
            memset(&app_link_table[conn_id], 0, sizeof(T_APP_LINK));
        }
        break;
    }
}
```

```

case GAP_CONN_STATE_CONNECTED:
{
}
break;
default:
    break;
}
}

```

4.5.3.4 GAP 回调函数处理

`app_gap_callback()`用于处理 GAP 回调函数消息，更多关于 GAP 回调函数的信息参见 [BLE GAP 回调函数](#)。

```

T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;
    switch (cb_type)
    {
        /* common msg */
        case GAP_MSG_LE_READ_RSSI:
            APP_PRINT_INFO3("GAP_MSG_LE_READ_RSSI:conn_id 0x%x cause 0x%x rssi %d",
                           p_data->p_le_read_rssi_rsp->conn_id,
                           p_data->p_le_read_rssi_rsp->cause,
                           p_data->p_le_read_rssi_rsp->rssi);
            break;
    }
}

```

4.5.3.5 Profile Client 消息回调函数

当 APP 使用 `xxx_add_client` 注册 specific client 时，APP 需要注册回调函数以处理 specific client 的消息。APP 可以针对不同的 clients 注册不同的回调函数，也可以注册通用回调函数以处理 specific clients 的消息。

`app_client_callback()` 是通用回调函数，根据 client id 区分不同的 clients。

```

void app_le_profile_init(void)
{
    client_init(3);
    gaps_client_id = gaps_add_client(app_client_callback, APP_MAX_LINKS);
    simple_ble_client_id = simp_ble_add_client(app_client_callback, APP_MAX_LINKS);
    bas_client_id = bas_add_client(app_client_callback, APP_MAX_LINKS);
    client_register_general_client_cb(app_client_callback);
}

```

```
}

T_APP_RESULT app_client_callback(T_CLIENT_ID client_id, uint8_t conn_id, void
                                *p_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    APP_PRINT_INFO2("app_client_callback: client_id %d, conn_id %d",
                    client_id, conn_id);
    if (client_id == CLIENT_PROFILE_GENERAL_ID)
    {
        T_CLIENT_APP_CB_DATA *p_client_app_cb_data = (T_CLIENT_APP_CB_DATA *)p_data;
        switch (p_client_app_cb_data->cb_type)
        {
            case CLIENT_APP_CB_TYPE_DISC_STATE:
                .....
            }
    }
    else if (client_id == gaps_client_id)
    {
        T_GAPS_CLIENT_CB_DATA *p_gaps_cb_data = (T_GAPS_CLIENT_CB_DATA *)p_data;
        switch (p_gaps_cb_data->cb_type)
        {
            case GAPS_CLIENT_CB_TYPE_DISC_STATE:
                .....
        }
    }
}
```

Scatternet APP 支持的 GATT profile clients 包括 GAP Service Client、Simple BLE Service Client 和 Battery Service Client。

4.5.3.6 Profile Service 消息回调函数

当 APP 使用 xxx_add_service 注册 specific service 时，APP 需要注册回调函数以处理 specific service 的消息。APP 需要调用 server_register_app_cb 注册回调函数以处理 profile server layer 的信息。

APP 可以针对不同的 services 注册不同的回调函数，也可以注册通用回调函数以处理 specific services 和 profile server layer 的消息。

app_profile_callback()是通用回调函数，根据 service id 区分不同的 services。

```
void app_le_profile_init(void)
{
    server_init(2);
    simp_srv_id = simp_ble_service_add_service(app_profile_callback);
    bas_srv_id = bas_add_service(app_profile_callback);
    server_register_app_cb(app_profile_callback);
}
```

更多信息参见 [BLE Profile Server](#)。

1. 通用 profile server 回调函数

SERVICE_PROFILE_GENERAL_ID 是 profile server layer 使用的 service id。profile server layer 使用的消息包含以下两种消息类型：

- 1) PROFILE_EVT_SRV_REG_COMPLETE: 在 GAP 启动流程中完成 service 注册流程。
- 2) PROFILE_EVT_SEND_DATA_COMPLETE: profile server layer 使用该消息向 APP 通知发送 notification/indication 的结果。

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    if (service_id == SERVICE_PROFILE_GENERAL_ID)
    {
        T_SERVER_APP_CB_DATA *p_param = (T_SERVER_APP_CB_DATA *)p_data;
        switch (p_param->eventId)
        {
            case PROFILE_EVT_SRV_REG_COMPLETE:// srv register result event.
                APP_PRINT_INFO1("PROFILE_EVT_SRV_REG_COMPLETE: result %d",
                                p_param->event_data.service_reg_result);
                break;
            case PROFILE_EVT_SEND_DATA_COMPLETE:
                ....
        }
    }
}
```

2. Battery Service

bas_srv_id 是 battery service 的 service id。

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    .....
    else if (service_id == bas_srv_id)
    {
        T_BAS_CALLBACK_DATA *p_bas_cb_data = (T_BAS_CALLBACK_DATA *)p_data;
        switch (p_bas_cb_data->msg_type)
        {
            case SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION:
                .....
        }
    }
}
```

3. Simple BLE Service

simp_srv_id 是 simple BLE service 的 service id。

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
```

```
T_APP_RESULT app_result = APP_RESULT_SUCCESS;  
.....  
else if (service_id == simp_srv_id)  
{  
    TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;  
    switch (p_simp_cb_data->msg_type)  
    {  
        case SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION:  
            {  
                switch (p_simp_cb_data->msg_data.notification_indification_index)  
                {  
                    case SIMP_NOTIFY_INDICATE_V3_ENABLE:  
                        .....  
                }  
            }  
    }  
}
```

4.5.4 APP 的可配置功能

在 app_flags.h 中定义 APP 的可配置功能。

```
/** @brief Config APP LE link number */  
  
#define APP_MAX_LINKS 2  
/** @brief Config airplane mode support: 0-Not built in, 1-built in, use user command to set*/  
#define F_BT_AIRPLANE_MODE_SUPPORT 0  
/** @brief Config device name characteristic and appearance characteristic property: 0-Not writeable, 1-writeable,  
save to flash*/  
#define F_BT_GAPS_CHAR_WRITEABLE 0  
/** @brief Config set physical: 0-Not built in, 1-built in, use user command to set*/  
#define F_BT_LE_5_0_SET_PHY_SUPPORT 0  
/** @brief Config local address type: 0-public address, 1-static random address */  
#define F_BT_LE_USE_STATIC_RANDOM_ADDR 0
```

用户可以通过修改宏定义的值以打开或关闭该功能，可以将相关代码拷贝到其它 APP。

4.5.4.1 Airplane mode 设置

配置 F_BT_AIRPLANE_MODE_SUPPORT 以打开该功能，具体信息参见 [Airplane Mode 设置](#)。

4.5.4.2 GAP Service characteristic 的可写属性

配置 F_BT_GAPS_CHAR_WRITEABLE 以打开该功能，具体信息参见 [GAP Service Characteristic 的可写属性](#)。

4.5.4.3 static random address 的使用

配置 F_BT_LE_USE_STATIC_RANDOM_ADDR 以打开该功能，具体信息参见[本地设备使用 Static Random Address](#)。

4.5.4.4 Physical channel 设置

配置 F_BT_LE_5_0_SET_PHY_SUPPORT 以打开该功能，具体信息参见[Physical \(PHY\) 设置](#)。

4.5.5 用户命令

使用 BLE Scatternent Application 时，需要通过个人电脑的 Data UART 输入命令进行交互。更多信息参见 BLE Application 用户命令。对 peripheral 角色和 central 角色的测试步骤参见[BLE Peripheral Application](#) 和 [BLE Central Application](#) 中的测试步骤。

4.6 BLE BT5 Peripheral Application

4.6.1 简介

本节内容是 BLE BT5 peripheral application 的概述。BLE BT5 peripheral 工程实现简单的使用 extended advertising 的 BLE BT5 peripheral 设备，可以作为开发基于 broadcaster 角色或 peripheral 角色的 APP 的框架。

1. Peripheral 角色的特征:

- 1) 使用 LE Advertising Extensions 发送 advertisement
- 2) 同时使能一个或多个 advertising set
- 3) 设定 advertising set 使能的 duration 或最大的 extended advertising events 数目
- 4) 使用 extended advertising PDUs 传输更多数据 (使用 LE Advertising Extensions 的 observer 或 central)
- 5) 可以作为 slave 角色建立一条 LE 链路，PHY 为 LE 1M PHY、LE 2M PHY 或 LE Coded PHY (使用 LE Advertising Extensions 的 central)

2. 可配特征:

- 1) DLPS

配置 F_BT_DLPS_EN 打开该功能(默认打开)

- 2) Link 数目

app_flags.h 中的 APP_MAX_LINKS(默认支持 1 条 link)

- 3) Advertising PHY

app_flags.h 中的 ADVERTISING_PHY(默认认的 ADVERTISING_PHY 为 APP_PRIMARY_1M_SECONDARY_2M, primary advertising PHY 是 LE 1M PHY, secondary advertising

PHY 是 LE 2M PHY。)

3. 兼容性:

使用 LE Advertising Extensions 的 peripheral 设备可以设置广播的持续时间或 extended advertising event 的最大数目，使能或禁用一个或多个 advertising set。

若使用 extended advertising PDUs，可以扩展 adv data 或 scan response data 的长度，以及使用 LE Coded PHY 增大可以建立连线的距离。关于 extended advertising PDUs 和 legacy advertising PDUs 的信息参见 [GAP Extended Advertising 相关参数的配置](#)。

若 Peripheral 设备使用 LE Advertising Extensions，用户需要考虑与不同蓝牙版本的对端设备的兼容性，与不同蓝牙版本的对端设备的兼容性如表 4-6 所示。

表 4-6 使用 LE Advertising Extensions 的 Peripheral 设备的兼容性

BT 5 Feature	Advertising PDUs	BT 4.0	BT 4.1	BT 4.2	BT 5.0 (not use LE Advertising)	BT 5.0 (use LE Advertising)
LE Advertising Extensions	extended advertising PDUs	N	N	N	N	Y
LE Advertising Extensions	legacy advertising PDUs	Y	Y	Y	Y	Y

4.6.2 工程概述

本节内容介绍工程的路径和结构，相关文件路径如下所示：

- 工程路径为 sdk\board\evb\ble_bt5_peripheral
- 工程源代码路径为 sdk\src\sample\ble_bt5_peripheral

工程目录结构如图 4-9 所示：

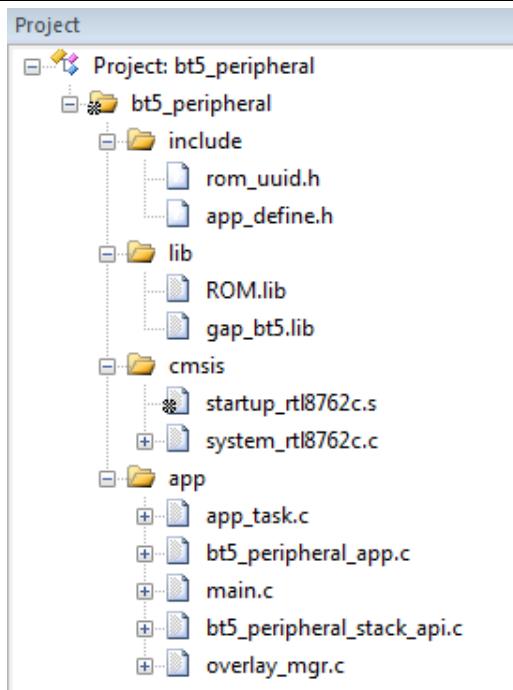


图 4-9 BT5 Peripheral 工程目录结构

文件列表可以分为以下四类。

表 4-7 BT5 Peripheral 工程目录结构

Directory	Description
include	rom_uuid.h: ROM UUID header files. User need not modify.
lib	The protocol stack and gap library file. User need not modify.
cmsis	The cmsis source code. User need not modify.
app	The application source code.

4.6.3 源代码概述

由于支持 LE Advertising Extensions, 设备可以使用一个或多个 advertising sets。APP 需要创建 advertising handle 以识别 advertising set, extended advertising 相关参数是针对指定 advertising set 配置的。使用 LE Advertising Extensions 时, APP 可以通过修改 advertising event properties 选择发送 legacy advertising PDUs (**BLE Peripheral Application** 只能发送 legacy advertising PDUs) 或 extended advertising PDUs。

启动 extended advertising 的流程如图 4-10 所示。根据 advertising event properties, 可以使用可选步骤, 更多信息参见 **GAP Extended Advertising 相关参数的配置**。以下章节将介绍流程的具体实现。

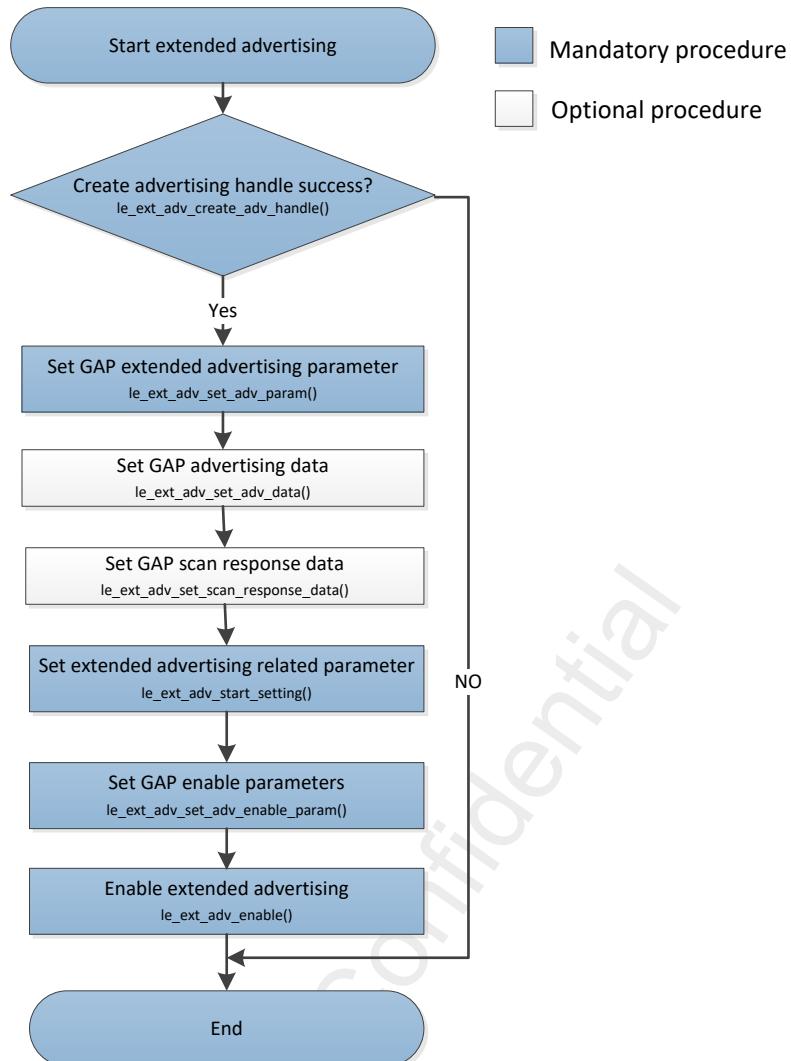


图 4-10 启动 Extended Advertising 的流程图

4.6.3.1 初始化

当上电或芯片重启后，main()函数会被调用，并执行以下初始化函数：

```

int main(void)
{
    board_init();
    le_gap_init(APP_MAX_LINKS);
    gap_lib_init();
    app_le_gap_init();
    pwr_mgr_init();
    task_init();
    os_sched_start();

    return 0;
}

```

GAP 初始化流程如下所示：

1. le_gap_init() - 初始化 GAP 并设置 link 数目
2. gap_lib_init() - 初始化 GAP BT5 lib, 更多信息参见 [GAP Lib](#)
3. app_le_gap_init() - GAP 参数的初始化, 用户可以通过修改以下参数自定义 application
 - 1) [Device Name 和 Device Appearance 的配置](#)
 - 2) [Bond Manager 参数的配置](#)
 - 3) [LE Advertising Extensions 参数的配置](#)
 - 4) [其它参数的配置](#)

更多关于 GAP 的初始化和启动流程的信息参见 [GAP 的初始化和启动流程](#)。

若 BT5 peripheral application 需要使用基于 GATT 的 Profiles, 参见 [BLE Peripheral Application 的 Profile 消息回调函数](#)。

4.6.3.1.1 GAP Extended Advertising 相关参数的配置

BT5 peripheral application 调用 le_init_ext_adv_params_ext_conn()设置 GAP 参数, 以使用 extended advertising PDUs 发送 Connectable Undirected Advertising 数据包。首先, BT5 peripheral application 调用 le_ext_adv_create_adv_handle()创建 advertising handle 以识别 advertising set。然后针对该 advertising set, 设置 GAP extended advertising 参数和 advertising 数据。示例代码如下所示:

```
void app_le_gap_init(void)
{
    .....
    /* Initialize extended advertising related parameters */
    le_init_ext_adv_params_ext_conn();
    .....
}

void le_init_ext_adv_params_ext_conn(void)
{
    T_LE_EXT_ADV_EXTENDED_ADV_PROPERTY adv_event_prop =
    LE_EXT_ADV_EXTENDED_ADV_CONN_UNDIRECTED;
    uint32_t primary_adv_interval_min = DEFAULT_ADVERTISING_INTERVAL_MIN;
    uint32_t primary_adv_interval_max = DEFAULT_ADVERTISING_INTERVAL_MAX;
    uint8_t primary_adv_channel_map = GAP_ADVCHAN_ALL;
    T_GAP_LOCAL_ADDR_TYPE own_address_type = GAP_LOCAL_ADDR_LE_PUBLIC;
    T_GAP_REMOTE_ADDR_TYPE peer_address_type = GAP_REMOTE_ADDR_LE_PUBLIC;
    uint8_t p_peer_address[6] = {0};
    T_GAP_ADV_FILTER_POLICY filter_policy = GAP_ADV_FILTER_ANY;
    uint8_t tx_power = 0;
    T_GAP_PHYS_PRIM_ADV_TYPE primary_adv_phy;
    uint8_t secondary_adv_max_skip = 0;
    T_GAP_PHYS_TYPE secondary_adv_phy;
```

```
uint8_t adv_sid = 0;
bool scan_req_notification_enable = false;

/* Initialize primary advertisement PHY and secondary advertisement PHY */
if (ADVERTISING_PHY == APP_PRIMARY_1M_SECONDARY_2M)
{
    primary_adv_phy = GAP_PHYS_PRIM_ADV_1M;
    secondary_adv_phy = GAP_PHYS_2M;
}
else if (ADVERTISING_PHY == APP_PRIMARY_CODED_SECONDARY_CODED)
{
    primary_adv_phy = GAP_PHYS_PRIM_ADV_CODED;
    secondary_adv_phy = GAP_PHYS_CODED;
}

/* Initialize extended advertising parameters */
adv_handle = le_ext_adv_create_adv_handle();
if(adv_handle == APP_IDLE_ADV_SET)
{
    return;
}
if (adv_set_num < APP_MAX_ADV_SET)
{
    ext_adv_state[adv_set_num].adv_handle = adv_handle;
}
le_ext_adv_set_adv_param(adv_handle, adv_event_prop, primary_adv_interval_min, primary_adv_interval_max,
                        primary_adv_channel_map, own_address_type, peer_address_type, p_peer_address,
                        filter_policy, tx_power, primary_adv_phy, secondary_adv_max_skip,
                        secondary_adv_phy, adv_sid, scan_req_notification_enable);

/* Initialize extended advertising data(max size = 245 bytes)*/
le_ext_adv_set_adv_data(adv_handle, sizeof(ext_adv_data), (uint8_t *)ext_adv_data);
}
```

在 gap_ext_adv.h 中提供关于 extended advertising 的 APIs，更多信息参见 gap_ext_adv.h。adv_event_prop 定义 advertising 的 properties，不同 adv_event_prop 的 advertising 需要不同的参数。根据使用的 advertising PDUs，advertising event properties 可分为两类：使用 legacy advertising PDUs 的 advertising event properties 和 extended advertising PDUs 的 advertising event properties。

使用 legacy advertising PDUs 的 advertising event properties 如表 4-8 所示。若使用 legacy advertising PDUs，advertising 数据或 scan response 数据不能超过 31 字节。

表 4-8 使用 legacy advertising PDUs 的 Extended Advertising 参数设置

adv_event_prop	LE_EXT_A	LE_EXT_A	LE_EXT_A	LE_EXT_A	LE_EXT_ADV
	DV_LEGACY_A	DV_LEGACY_A		DV_LEGACY_A	_LEGACY_ADV_C
	DV_CONN_SCA	DV_CONN_HIG	DV_SCAN_UND	DV_NON_SCAN	NON_LOW_DUTY
	N_UNDIRECTE	H_DUTY_DIRE	IRECTED	_NON_CONN_U	_DIRECTED
	D	CTED		NDIRECTED	
adv_handle	Y	Y	Y	Y	Y
primary_adv_int_erval_min	Y	Ignore	Y	Y	Y
primary_adv_int_erval_max	Y	Ignore	Y	Y	Y
primary_adv_channel_map	Y	Y	Y	Y	Y
peer_address_type	Ignore	Y	Ignore	Ignore	Y
p_peer_address	Ignore	Y	Ignore	Ignore	Y
filter_policy	Y	Ignore	Y	Y	Ignore
primary_adv_phy	LE 1M PHY				
secondary_adv_phy	Ignore	Ignore	Ignore	Ignore	Ignore
allow establish link	Y	Y	N	N	Y
Advertising data	Y	N	Y	Y	N
Scan response data	Y	N	Y	N	N

使用 extended advertising PDUs 的 advertising event properties 如表 4-9 所示。在 bt5_peripheral_stack_api.h 和 bt5_peripheral_stack_api.c 中提供关于使用 extended advertising PDUs 的 extended advertising 参数设置的示例代码。若只使用一个 advertising set, advertising 数据或 scan response 数据的最大长度如表 4-9 所示。

表 4-9 使用 extended advertising PDUs 的 Extended Advertising 参数设置

	LE_EXT	LE_EXT	LE_EXT	LE_EXT	LE_EXT_A	LE_EXT_A
	_ADV_EXTE	_ADV_EXTE	_ADV_EXTE	_ADV_EXTE	DV_EXTENDE	DV_EXTENDE
adv_event_pro	NDED_ADV_	NDED_ADV_	NDED_ADV_	NDED_ADV_	D_ADV_SCAN_	D_ADV_SCAN_
p	NON_SCAN_	NON_SCAN_	NON_CONN	NON_CONN	UNDIRECTED	DIRECTED
	NON_CONN	NON_CONN	CONN_UNDI	CONN_DIRE		
	_UNDIRECT	_DIRECTED	RECTED	CTED		
	ED					
adv_handle	Y	Y	Y	Y	Y	Y
primary_adv_	Y	Y	Y	Y	Y	Y
interval_min						
primary_adv_	Y	Y	Y	Y	Y	Y
interval_max						
primary_adv_	Y	Y	Y	Y	Y	Y
channel_map						
peer_address_type	Ignore	Y	Ignore	Y	Ignore	Y
p_peer_addresses	Ignore	Y	Ignore	Y	Ignore	Y
filter_policy	Y	Ignore	Y	Ignore	Y	Ignore
primary_adv_phy	LE 1M PHY LE Coded PHY	LE 1M PHY LE Coded PHY	LE 1M PHY LE Coded PHY			
secondary_ad_v_phy	LE 1M PHY LE 2M PHY LE Coded PHY	LE 1M PHY LE 2M PHY LE Coded PHY	LE 1M PHY LE 2M PHY LE Coded PHY			
allow establish link	N	N	Y	Y	N	N
Advertising data	Y 1024 bytes	Y 1024 bytes	Y 245 bytes	Y 239 bytes	N	N
Scan response data	N	N	N	N	Y 991 bytes	Y 991 bytes

4.6.3.2 GAP 消息处理

一旦收到 GAP message， app_handle_gap_msg() 将会被调用。更多关于 GAP message 的信息参见 [BLE GAP 消息](#)。

当收到 GAP_INIT_STATE_STACK_READY 消息时，BT5 peripheral application 将调用 le_ext_adv_start_setting() 设置 extended advertising 相关参数。EXT_ADV_SET_AUTO 表示将根据 advertising event properties 自动设置针对指定 advertising set 的 extended advertising 相关参数(包括 advertising 参数、advertising 数据和 scan response 数据)。否则，application 需要参考表 4-8 或表 4-9 根据 advertising event properties 设置 flags。

```
void app_handle_dev_state_evt(T_GAP_DEV_STATE new_state, uint16_t cause)
{
    APP_PRINT_INFO3("app_handle_dev_state_evt: init state %d, adv state %d, cause 0x%x",
                    new_state.gap_init_state, new_state.gap_adv_state, cause);
    if (gap_dev_state.gap_init_state != new_state.gap_init_state)
    {
        if (new_state.gap_init_state == GAP_INIT_STATE_STACK_READY)
        {
            APP_PRINT_INFO0("GAP stack ready");
            /* Stack ready, set extended advertising related parameters */
            le_ext_adv_start_setting(adv_handle, EXT_ADV_SET_AUTO);
        }
    }
    gap_dev_state = new_state;
}
```

在 app_gap_callback() 调用 le_ext_adv_enable() 之后，BT5 peripheral application 将收到 extended advertising 状态消息。更多关于处理 extended advertising 状态消息的信息参见 [Extended Advertising 状态消息](#)。

当收到 GAP_CONN_STATE_DISCONNECTED 消息时，BT5 peripheral application 将调用 le_ext_adv_enable() 以便能指定 advertising set。在连接断开之后，bt5 peripheral application 将恢复为可连接状态。

```
void app_handle_conn_state_evt(uint8_t conn_id, T_GAP_CONN_STATE new_state, uint16_t disc_cause)
{
    APP_PRINT_INFO4("app_handle_conn_state_evt: conn_id %d old_state %d new_state %d, disc_cause 0x%x",
                    conn_id, gap_conn_state, new_state, disc_cause);
    switch (new_state)
    {
        case GAP_CONN_STATE_DISCONNECTED:
        {
            if ((disc_cause != (HCI_ERR | HCI_ERR_REMOTE_USER_TERMINATE))
```

```

    && (disc_cause != (HCI_ERR | HCI_ERR_LOCAL_HOST_TERMINATE)))
{
    APP_PRINT_ERROR1("app_handle_conn_state_evt: connection lost cause 0x%x", disc_cause);
}
/* Enable one advertising set */
le_ext_adv_enable(1, &adv_handle);
}
break;
.....
}
gap_conn_state = new_state;
}

```

4.6.3.3 GAP 回调函数处理

`app_gap_callback()`用于处理 GAP 回调函数消息，更多关于 GAP 回调函数的信息参见 [BLE GAP 回调函数](#)。

在协议栈准备就绪之后调用 `le_ext_adv_start_setting()`，设置 extended advertising 参数的结果将在函数 `app_gap_callback()` 中以回调类型 `GAP_MSG_LE_EXT_ADV_START_SETTING` 返回。若返回结果表示参数设置成功，APP 将使能 extended advertising。

首先，APP 设置 GAP extended advertising 使能参数，以决定是否通过使用 duration 或最大的 extended advertising events 数目停止 extended advertising。APP 调用 `le_ext_adv_enable()` 对一个 advertising set 使能 extended advertising。示例代码如下所示：

```

T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;

    switch (cb_type)
    {
        case GAP_MSG_LE_EXT_ADV_START_SETTING:
            APP_PRINT_INFO3("GAP_MSG_LE_EXT_ADV_START_SETTING:cause 0x%x, flag 0x%x,
                            adv_handle %d", p_data->p_le_ext_adv_start_setting_rsp->cause,
                            p_data->p_le_ext_adv_start_setting_rsp->flag, p_data->p_le_ext_adv_start_setting_rsp->adv_handle);

            if (p_data->p_le_ext_adv_start_setting_rsp->cause == GAP_CAUSE_SUCCESS)
            {
                /* Initialize enable parameters */
                le_init_ext_adv_enable_params(p_data->p_le_ext_adv_start_setting_rsp->adv_handle);
                /* Enable one advertising set */
                le_ext_adv_enable(1, &p_data->p_le_ext_adv_start_setting_rsp->adv_handle);
            }
    }
}

```

```
break;  
.....  
}
```

函数 app_handle_gap_msg()用于处理 extended advertising 状态消息。使能 advertising set 的结果将在 app_gap_callback()中以回调类型 GAP_MSG_LE_EXT_ADV_ENABLE 返回。示例代码如下所示：

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)  
{  
    T_APP_RESULT result = APP_RESULT_SUCCESS;  
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;  
  
    switch (cb_type)  
    {  
        case GAP_MSG_LE_EXT_ADV_ENABLE:  
            APP_PRINT_INFO1("GAP_MSG_LE_EXT_ADV_ENABLE:cause 0x%x", p_data->le_cause.cause);  
            break;  
        .....  
    }  
}
```

当 BLE BT5 Peripheral Application 在 evolution board 上运行时，设备将是可连接的。对端设备可以对 peripheral 设备进行 scan，并创建 connection。

4.6.4 APP 的可配置功能

在 app_flags.h 中定义 APP 的可配置功能。

```
/** @brief Configure Advertising PHY */  
#define ADVERTISING_PHY APP_PRIMARY_1M_SECONDARY_2M  
/** @brief Config APP LE link number */  
#define APP_MAX_LINKS 1  
/** @brief Config DLPS: 0-Disable DLPS, 1-Enable DLPS */  
#define F_BT_DLPS_EN 1
```

4.6.4.1 DLPS

关于 DLPS 的信息参见 DLPS 相关文档^[3]。

4.6.5 测试步骤

首先，编译并将 BLE BT5 peripheral application 下载到 evolution board。BLE BT5 peripheral application 的基本功能如上所述，为实现其它复杂功能，用户可以参考 SDK 提供的使用手册和源代码进行开发。

当 BLE BT5 peripheral application 在 evolution board 上运行时，设备将是可连接的。对端设备可以对 peripheral 设备进行 scan，并创建 connection。在连接断开之后，BLE BT5 peripheral application 将恢复为可连接状态。

4.6.5.1 与 BLE BT5 Central Application 设备测试

可以使用 BLE BT5 central application 与 BLE BT5 peripheral application 进行测试, 更多关于 BT5 central application 的信息参见 [BLE BT5 Central Application](#)。测试流程参见 [与 BLE BT5 Peripheral Application 设备测试](#)。

4.7 BLE BT5 Central Application

4.7.1 简介

本节内容是 BLE BT5 central application 的概述。BLE BT5 central 工程实现简单的使用 LE Advertising Extensions 的 BLE BT5 central 设备, 可以作为开发基于 central 角色或 observer 角色的 APP 的框架。

1. Central 角色的特征:

- 1) 在 primary advertising channel (LE 1M PHY or/and LE Coded PHY) 对 advertising 数据包进行 extended scan
- 2) 设定 scan 的 duration 或 period scan
- 3) 可以作为 master 角色发起一条 LE 链路, PHY 为 LE 1M PHY、LE 2M PHY 或 LE Coded PHY (使用 LE Advertising Extensions 的 peripheral)

2. 可配特征:

- 1) Link 数目
app_flags.h 中的 APP_MAX_LINKS(默认支持 1 条 link)

3. 兼容性:

使用 LE Advertising Extensions 的 central 设备可以与使用 legacy advertising PDUs 或 extended advertising PDUs 的设备通信, 可以设置 scan 的持续时间。

若使用 extended advertising PDUs, 与不同蓝牙版本的对端设备的兼容性如表 4-10 所示。

表 4-10 使用 LE Advertising Extensions 的 Central 设备的兼容性

BT 5 Feature	BT 4.0	BT4.1	BT 4.2	BT 5.0 (not use LE Advertising Extensions)	BT 5.0 (use LE Advertising Extensions)
LE Advertising Extensions	Y	Y	Y	Y	Y

4.7.2 工程概述

本节内容介绍工程的路径和结构，相关文件路径如下所示：

- 工程路径为 sdk\board\evb\ble_bt5_central
- 工程源代码路径为 sdk\src\sample\ble_bt5_central

工程目录结构如图 4-11 所示：

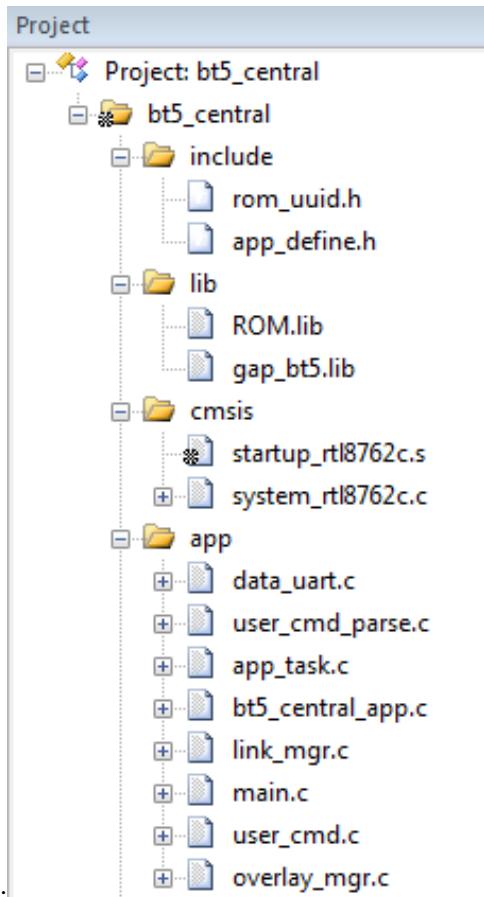


图 4-11 BT5 Central 工程目录结构

文件列表分为以下四类。

表 4-11 BT5 Central 工程文件列表

Directory	Description
include	rom_uuid.h: ROM UUID header files. User need not modify.
lib	The protocol stack and gap library file. User need not modify.
cmsis	The cmsis source code. User need not modify.
app	The application source code.

4.7.3 源代码概述

以下章节介绍该 APP 的重要组成部分。

4.7.3.1 初始化

当上电或芯片重启后，main()函数会被调用，并执行以下初始化函数：

```
int main(void)
{
    board_init();
    le_gap_init(APP_MAX_LINKS);
    gap_lib_init();
    app_le_gap_init();
    pwr_mgr_init();
    task_init();
    os_sched_start();

    return 0;
}
```

GAP 初始化流程如下所示：

1. le_gap_init() - 初始化 GAP 并设置 link 数目
2. gap_lib_init() - 初始化 GAP BT5 lib，更多信息参见 [GAP Lib](#)
3. app_le_gap_init() - GAP 参数的初始化，用户可以通过修改以下参数自定义 application
 - 1) [Device Name 和 Device Appearance 的配置](#)
 - 2) [Bond Manager 参数的配置](#)
 - 3) [LE Advertising Extensions 参数的配置](#)

更多关于 GAP 的初始化和启动流程的信息参见 [GAP 的初始化和启动流程](#)。

若 BT5 central application 需要使用基于 GATT 的 Profiles，参见 [BLE Central Application](#) 的 [Profile 消息回调函数](#) 和 [Discover GATT Services 流程](#)。

若 BT5 central application 需要使用多条链路，参见 [BLE Central Application](#) 中的 [多链路管理](#)。

4.7.3.2 GAP 消息处理

一旦收到 GAP message，app_handle_gap_msg()将会被调用。更多关于 GAP message 的信息参见 [BLE GAP 消息](#)。

当收到 GAP_MSG_LE_CONN_STATE_CHANGE 消息时，APP 将更新 link 状态。

```
void app_handle_conn_state_evt(uint8_t conn_id, T_GAP_CONN_STATE new_state, uint16_t disc_cause)
{
    APP_PRINT_INFO4("app_handle_conn_state_evt: conn_id %d, conn_state(%d -> %d), disc_cause 0x%x",
}
```

```
conn_id, gap_conn_state, new_state, disc_cause);
```

```

switch (new_state)
{
case GAP_CONN_STATE_DISCONNECTED:
{
    if ((disc_cause != (HCI_ERR | HCI_ERR_REMOTE_USER_TERMINATE))
        && (disc_cause != (HCI_ERR | HCI_ERR_LOCAL_HOST_TERMINATE)))
    {
        APP_PRINT_ERROR1("app_handle_conn_state_evt: connection lost cause 0x%x", disc_cause);
    }
}
data_uart_print("Disconnect conn_id %d\r\n", conn_id);
break;

case GAP_CONN_STATE_CONNECTED:
{
    uint8_t tx_phy;
    uint8_t rx_phy;
    uint16_t conn_interval;
    uint16_t conn_latency;
    uint16_t conn_supervision_timeout;
    uint8_t remote_bd[6];
    T_GAP_REMOTE_ADDR_TYPE remote_bd_type;

    le_get_conn_param(GAP_PARAM_CONN_INTERVAL, &conn_interval, conn_id);
    le_get_conn_param(GAP_PARAM_CONN_LATENCY, &conn_latency, conn_id);
    le_get_conn_param(GAP_PARAM_CONN_TIMEOUT, &conn_supervision_timeout, conn_id);
    le_get_conn_addr(conn_id, remote_bd, (uint8_t *)&remote_bd_type);
    APP_PRINT_INFO5("GAP_CONN_STATE_CONNECTED:remote_bd %s, remote_addr_type %d,
conn_interval 0x%x, conn_latency 0x%x, conn_supervision_timeout 0x%x",
                    TRACE_BDADDR(remote_bd), remote_bd_type,
                    conn_interval, conn_latency, conn_supervision_timeout);

    le_get_conn_param(GAP_PARAM_CONN_TX_PHY_TYPE, &tx_phy, conn_id);
    le_get_conn_param(GAP_PARAM_CONN_RX_PHY_TYPE, &rx_phy, conn_id);
    data_uart_print("Connected success conn_id %d, tx_phy %d, rx_phy %d\r\n", conn_id, tx_phy, rx_phy);
}
break;

default:
break;
}
gap_conn_state = new_state;
```

{

4.7.3.3 GAP 回调函数处理

app_gap_callback()用于处理 GAP 回调函数消息，更多关于 GAP 回调函数的信息参见 [BLE GAP 回调函数](#)。若设备启动 extended scan 之后收到 advertising 数据或 scan response 数据，GAP 层将使用 GAP_MSG_LE_EXT_ADV_REPORT_INFO 消息通知 APP。示例代码如下所示：

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;

    APP_PRINT_TRACE1("app_gap_callback: cb_type = %d", cb_type);

    switch (cb_type)
    {
        case GAP_MSG_LE_EXT_ADV_REPORT_INFO:
            APP_PRINT_INFO6("GAP_MSG_LE_EXT_ADV_REPORT_INFO:connectable %d, scannable %d,
direct %d, scan response %d, legacy %d, data status 0x%x",
            p_data->p_le_ext_adv_report_info->event_type & GAP_EXT_ADV_REPORT_BIT_CONNECTABLE_ADV,
            p_data->p_le_ext_adv_report_info->event_type & GAP_EXT_ADV_REPORT_BIT_SCANNABLE_ADV,
            p_data->p_le_ext_adv_report_info->event_type & GAP_EXT_ADV_REPORT_BIT_DIRECTED_ADV,
            p_data->p_le_ext_adv_report_info->event_type & GAP_EXT_ADV_REPORT_BIT_SCAN_RESPONSE,
            p_data->p_le_ext_adv_report_info->event_type & GAP_EXT_ADV_REPORT_BIT_USE_LEGACY_ADV,
            p_data->p_le_ext_adv_report_info->data_status);

            APP_PRINT_INFO5("GAP_MSG_LE_EXT_ADV_REPORT_INFO:event_type 0x%x, bd_addr %s,
addr_type %d, rssi %d, data_len %d",
            p_data->p_le_ext_adv_report_info->event_type,
            TRACE_BDADDR(p_data->p_le_ext_adv_report_info->bd_addr),
            p_data->p_le_ext_adv_report_info->addr_type,
            p_data->p_le_ext_adv_report_info->rssi,
            p_data->p_le_ext_adv_report_info->data_len);

            APP_PRINT_INFO5("GAP_MSG_LE_EXT_ADV_REPORT_INFO:primary_phy %d, secondary_phy %d,
adv_sid %d, tx_power %d, peri_adv_interval %d",
            p_data->p_le_ext_adv_report_info->primary_phy,
            p_data->p_le_ext_adv_report_info->secondary_phy,
            p_data->p_le_ext_adv_report_info->adv_sid,
            p_data->p_le_ext_adv_report_info->tx_power,
            p_data->p_le_ext_adv_report_info->peri_adv_interval);

            APP_PRINT_INFO2("GAP_MSG_LE_EXT_ADV_REPORT_INFO:direct_addr_type 0x%x,
direct_addr %s",
            p_data->p_le_ext_adv_report_info->direct_addr_type,
            TRACE_BDADDR(p_data->p_le_ext_adv_report_info->direct_addr));
    }
}
```

```

link_mngr_add_device(p_data->p_le_ext_adv_report_info->bd_addr,
                     p_data->p_le_ext_adv_report_info->addr_type);

#if APP_RECOMBINE_ADV_DATA
    if (!(p_data->p_le_ext_adv_report_info->event_type &
GAP_EXT_ADV_REPORT_BIT_USE_LEGACY_ADV))
    {
        /* If the advertisement uses extended advertising PDUs, recombine advertising data. */
        app_handle_ext_adv_report(p_data->p_le_ext_adv_report_info->event_type,
                               p_data->p_le_ext_adv_report_info->data_status, p_data->p_le_ext_adv_report_info->bd_addr,
                               p_data->p_le_ext_adv_report_info->data_len, p_data->p_le_ext_adv_report_info->p_data);
    }
#endif
break;
.....
}

```

4.7.4 APP 的可配置功能

在 app_flags.h 中定义 APP 的可配置功能。

```

/** @brief Config APP LE link number */
#define APP_MAX_LINKS 1
/** @brief Configure APP to recombine advertising data: 0-Disable recombine advertising data feature, 1- recombine
advertising data */
#define APP_RECOMBINE_ADV_DATA 0

```

用户可以通过修改宏定义的值以打开或关闭该功能，可以将相关代码拷贝到其它 APP。

4.7.4.1 Advertising 数据重组

Advertising 数据的重组实现对一组 advertising 数据或 scan response 数据的重组，在下次重组之前会删除已重组的 advertising 数据。该重组流程可以作为开发同时重组多组 advertising data 的框架。

在 GAP 初始化时申请用于重组 advertising 数据的内存，示例代码如下所示。

```

void app_le_gap_init(void)
{
    .....
    /* Allocate memory for recombining advertising data */
#if APP_RECOMBINE_ADV_DATA
    ext_adv_data = os_mem_zalloc(RAM_TYPE_DATA_ON, sizeof(T_EXT_ADV_DATA));
    ext_adv_data->flag = false;
#endif

```

```
.....  
}
```

消息数据结构为 T_EXT_ADV_DATA。

```
typedef struct  
{  
    uint8_t      bd_addr[GAP_BD_ADDR_LEN];           /**< remote BD */  
    bool         flag;                                /**< flag of recombining advertising data, true: recombining,  
                                                       false: waiting extended advertising PDUs */  
    uint16_t     event_type;                          /**< advertising event type */  
    uint16_t     data_len;                            /**< length of recombined advertising data */  
    uint8_t      p_data[APP_MAX_EXT_ADV_TOTAL_LEN];   /**< recombined advertising data */  
} T_EXT_ADV_DATA;
```

BT5 central application 在 app_gap_callback() 中处理 GAP_MSG_LE_EXT_ADV_REPORT_INFO 消息，该消息表示收到 advertising 数据或 scan response 数据，处理流程参见 [GAP 回调函数处理](#)。若该 extended advertising report 表示使用的是 extended advertising PDU，APP 将调用 app_handle_ext_adv_report() 以重组 advertising 数据。以下章节描述 advertising 数据的重组流程，相关代码参见 bt5_central_app.c 的 app_handle_ext_adv_report() 函数。

4.7.4.1.1 GAP_EXT_ADV_EVT_DATA_STATUS_COMPLETE

GAP_EXT_ADV_EVT_DATA_STATUS_COMPLETE 表示数据是完整的。Data status 为 GAP_EXT_ADV_EVT_DATA_STATUS_COMPLETE 的 advertising report 的处理流程如图 4-12 所示。

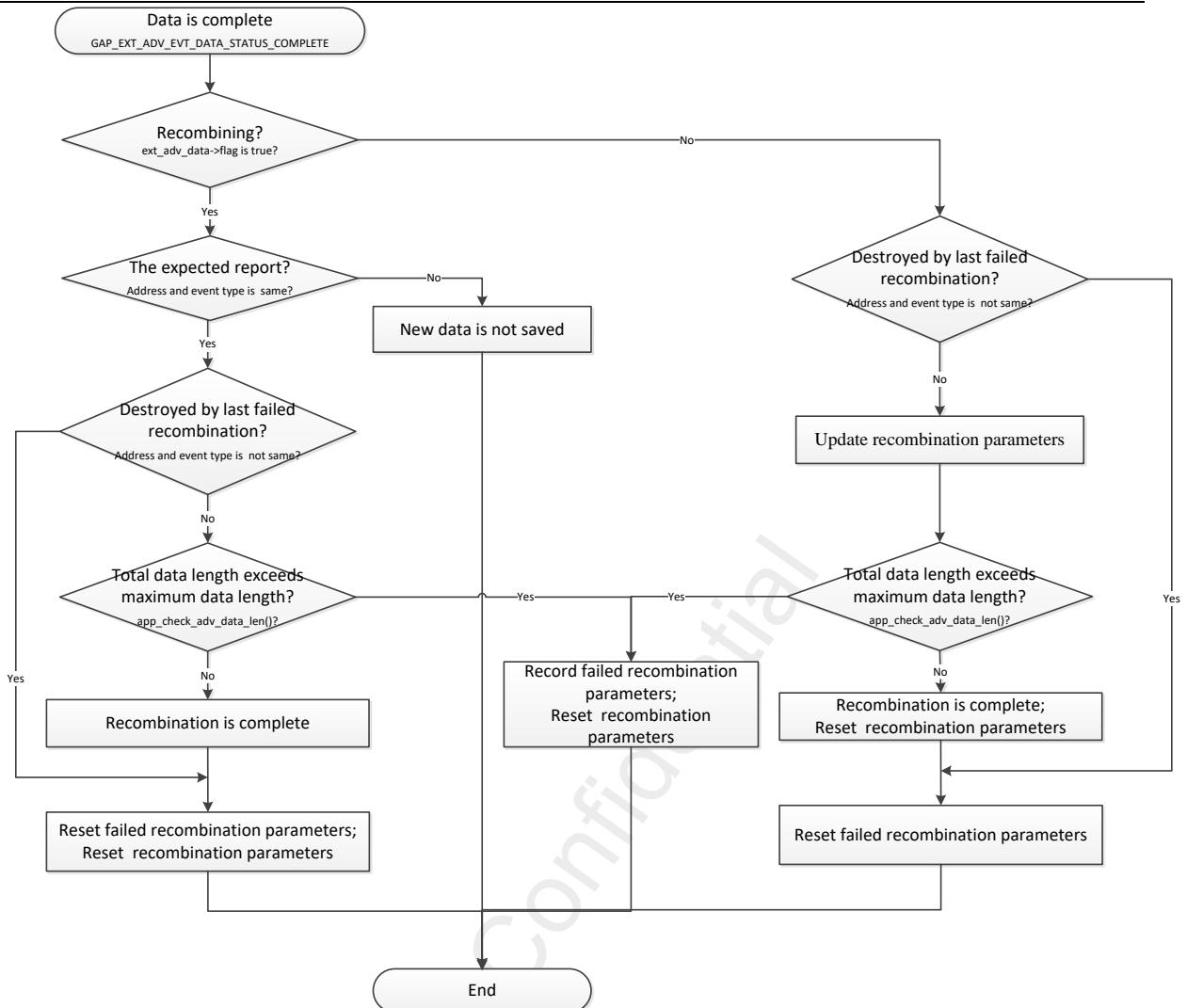


图 4-12 重组流程图(GAP_EXT_ADV_EVT_DATA_STATUS_COMPLETE)

4.7.4.1.2 GAP_EXT_ADV_EVT_DATA_STATUS_MORE

GAP_EXT_ADV_EVT_DATA_STATUS_MORE 表示数据是不完整的且将会收到更多的数据。Data status 为 GAP_EXT_ADV_EVT_DATA_STATUS_MORE 的 advertising report 的处理流程如图 4-13 所示。

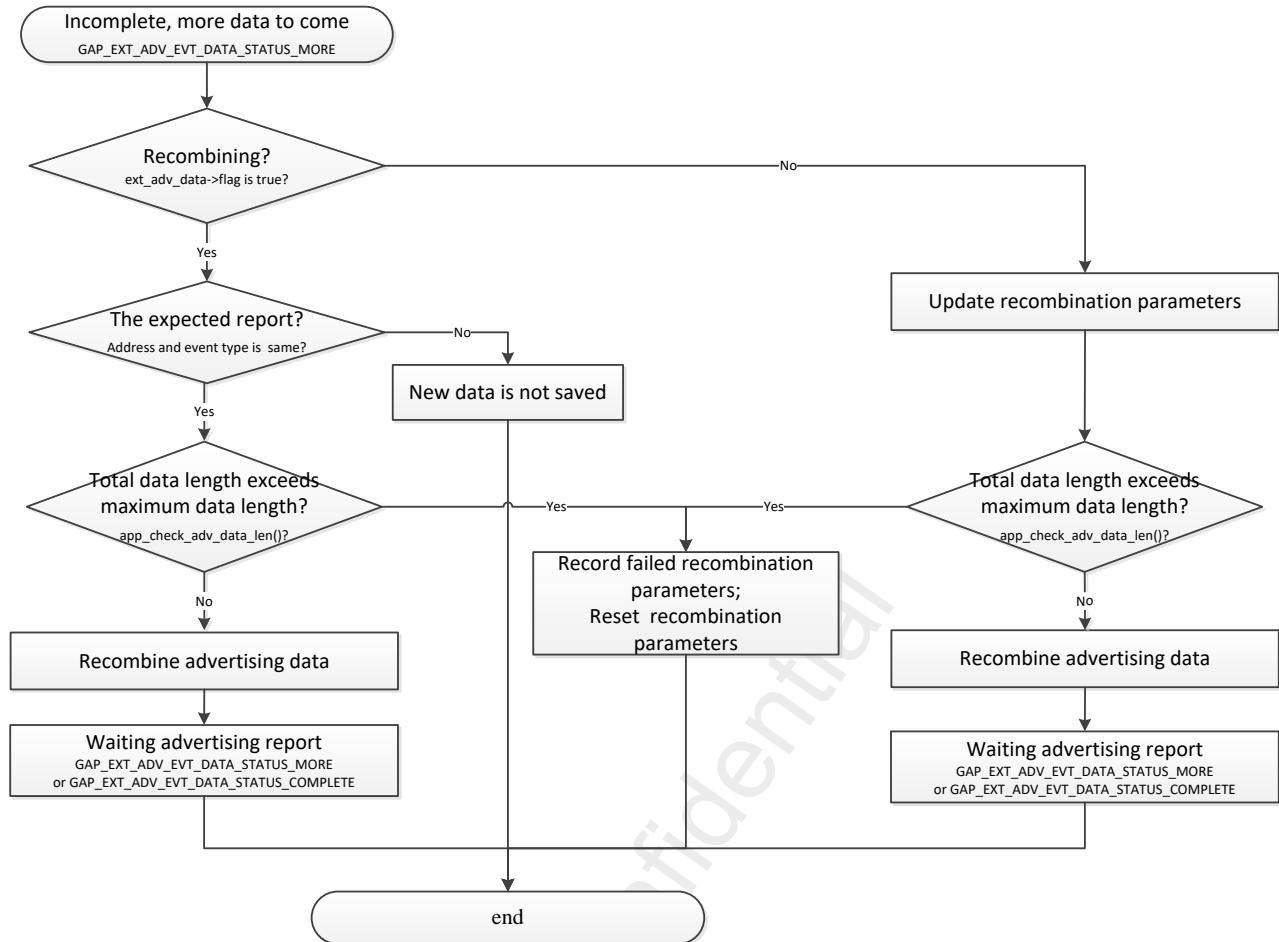


图 4-13 重组流程图(GAP_EXT_ADV_EVT_DATA_STATUS_MORE)

在处理 data status 为 GAP_EXT_ADV_EVT_DATA_STATUS_MORE 的 advertising report 之后，APP 需要等待 advertising report 以完成重组。data status 为 GAP_EXT_ADV_EVT_DATA_STATUS_COMPLETE 的 advertising report 表示重组已完成。

4.7.4.1.3 GAP_EXT_ADV_EVT_DATA_STATUS_TRUNCATED

GAP_EXT_ADV_EVT_DATA_STATUS_TRUNCATED 表示数据是不完整，数据是截断的。若具有截断数据的 advertising report 是 APP 等待的 report，APP 将通过重置重组参数以终止当前的重组流程，并等待 advertising report 以启动下一次重组流程。

4.7.5 用户命令

使用 BLE BT5 Central Application 时，需要通过个人电脑的 Data UART 输入命令进行交互。更多信息参见 [BLE Application 用户命令](#)。以下章节介绍 BT5 central application 使用的命令。

4.7.5.1 Extended Scanning 命令

Extended scanning 命令包括 escan 和 stopescan

escan 命令用于启动 extended scanning。首先，APP 调用 le_ext_scan_set_param()以初始化 extended scan 参数，调用 le_ext_scan_set_phy_param()以初始化 extended scan PHY 参数。然后，APP 调用 le_ext_scan_start()以启动 extended scanning。escan 命令的实现如下所示。

```
static T_USER_CMD_PARSE_RESULT cmd_escan(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    .....
    /* Initialize extended scan parameters */
    le_ext_scan_set_param(GAP_PARAM_EXT_SCAN_LOCAL_ADDR_TYPE, sizeof(own_address_type),
                          &own_address_type);
    le_ext_scan_set_param(GAP_PARAM_EXT_SCAN_PHYS, sizeof(scan_phys),
                          &scan_phys);
    le_ext_scan_set_param(GAP_PARAM_EXT_SCAN_DURATION, sizeof(ext_scan_duration),
                          &ext_scan_duration);
    le_ext_scan_set_param(GAP_PARAM_EXT_SCAN_PERIOD, sizeof(ext_scan_period),
                          &ext_scan_period);
    le_ext_scan_set_param(GAP_PARAM_EXT_SCAN_FILTER_POLICY, sizeof(ext_scan_filter_policy),
                          &ext_scan_filter_policy);
    le_ext_scan_set_param(GAP_PARAM_EXT_SCAN_FILTER_DUPLICATES, sizeof(ext_scan_filter_duplicate),
                          &ext_scan_filter_duplicate);

    /* Initialize extended scan PHY parameters */
    le_ext_scan_set_phy_param(LE_SCAN_PHY_LE_1M, &extended_scan_param[0]);
    le_ext_scan_set_phy_param(LE_SCAN_PHY_LE_CODED, &extended_scan_param[1]);

    /* Enable extended scan */
    cause = le_ext_scan_start();
    return (T_USER_CMD_PARSE_RESULT)cause;
}
```

在 gap_ext_scan.h 中定义 extended scanning 相关 APIs 的使用方法。

不同于 BLE central application，BLE BT5 central application 可以使用 Duration 参数和 Period 参数决定 scanning mode。若 Duration 参数为零，设备将持续 scanning 直到停止 scanning。若 Duration 参数和 Period 参数均不为零，在一个 scan period 内，设备的 scan 时间为 Duration 参数，且 scan period 将持续到停止 scanning。若 Duration 参数不为零且 Period 参数为零，设备持续 scanning 的时间不能超过 Duration 参数。

不同于 BLE central application，BLE BT5 central application 可以接收使用 legacy advertising PDUs 的 advertising 数据包，以及 primary advertising channel 为 LE 1M PHY or/and LE Coded PHY 的 extended advertising PDUs。

escan 命令的定义如下所示。

```
const T_USER_CMD_TABLE_ENTRY user_cmd_table[] =  
{  
    .....  
    {  
        "escan",  
        "escan [scan_mode] [scan_phys]\r\n",  
        "Start extended scan\r\n"  
        "[scan_mode]: 0-(continue scanning until scanning is disabled)\r\n"  
        "1-(scan for the duration within a scan period, and scan periods continue until scanning is  
disabled)\r\n"  
        "2-(continue scanning until duration has expired)\r\n"  
        "[scan_phys]: set scan PHYs to 1(LE 1M PHY), 4(LE Coded PHY) or 5(LE 1M PHY and LE Coded  
PHY)\r\n"  
        "sample: escan 0 4\r\n",  
        cmd_escan  
    },  
    .....  
}
```

命令 stopescan 通过调用 le_ext_scan_stop()停止 extended scanning，其实现和用法如下所示。

```
static T_USER_CMD_PARSE_RESULT cmd_stopescan(T_USER_CMD_PARSED_VALUE *p_parse_value)  
{  
    T_GAP_CAUSE cause;  
    cause = le_ext_scan_stop();  
    return (T_USER_CMD_PARSE_RESULT)cause;  
}  
const T_USER_CMD_TABLE_ENTRY user_cmd_table[] =  
{  
    .....  
    {  
        "stopescan",  
        "stopescan\r\n",  
        "Stop extended scan\r\n",  
        cmd_stopescan  
    },  
    .....  
}
```

4.7.5.2 Extended Create Connection 命令

Extended Create Connection 命令包括 condev 和 disc。

使用 LE Advertising Extensions 时，APP 可以使用 condev 命令创建 connection。Initiating PHYs 参数表示接收 advertising 数据包的 primary advertising channel 的 PHY(s)，以及 connection 参数对应的 PHYs。Primary advertising channel 包括 LE 1M PHY 和 LE Coded PHY，因此 Initiating PHYs 参数必须至少有一个允许在

primary advertising channel 进行 scan 的 PHY 的 bit 置 1。若 peripheral 使用 extended advertising PDUs 且 secondary advertising PHY 为 LE 1M PHY、LE 2M PHY 或 LE Coded PHY，设备可以作为 master 角色在 LE 1M PHY、LE 2M PHY 或 LE Coded PHY 上建立 connection。

condev 命令的实现和用法如下所示。

```
static T_USER_CMD_PARSE_RESULT cmd_condev(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    uint8_t dev_idx = p_parse_value->dw_param[0];
    if (dev_idx < dev_list_count)
    {
        T_GAP_CAUSE cause;
        T_GAP_LE_CONN_REQ_PARAM conn_req_param;
        T_GAP_LOCAL_ADDR_TYPE local_addr_type = GAP_LOCAL_ADDR_LE_PUBLIC;
        uint8_t init_phys = 0;
        conn_req_param.scan_interval = 0x10;
        conn_req_param.scan_window = 0x10;
        conn_req_param.conn_interval_min = 80;
        conn_req_param.conn_interval_max = 80;
        conn_req_param.conn_latency = 0;
        conn_req_param.supv_tout = 1000;
        conn_req_param.ce_len_min = 2 * (conn_req_param.conn_interval_min - 1);
        conn_req_param.ce_len_max = 2 * (conn_req_param.conn_interval_max - 1);

        le_set_conn_param(GAP_CONN_PARAM_1M, &conn_req_param);
        le_set_conn_param(GAP_CONN_PARAM_2M, &conn_req_param);
        le_set_conn_param(GAP_CONN_PARAM_CODED, &conn_req_param);

        uint32_t input_phys = p_parse_value->dw_param[1];
        switch (input_phys)
        {
            case 0x001:
                init_phys = GAP_PHYS_CONN_INIT_1M_BIT;
                break;
            case 0x011:
                init_phys = GAP_PHYS_CONN_INIT_2M_BIT | GAP_PHYS_CONN_INIT_1M_BIT;
                break;
            case 0x100:
                init_phys = GAP_PHYS_CONN_INIT_CODED_BIT;
                break;
            case 0x101:
                init_phys = GAP_PHYS_CONN_INIT_CODED_BIT | GAP_PHYS_CONN_INIT_1M_BIT;
                break;
            case 0x110:
                init_phys = GAP_PHYS_CONN_INIT_CODED_BIT | GAP_PHYS_CONN_INIT_2M_BIT;
                break;
        }
    }
}
```

```

        break;

    case 0x11:
        init_phys = GAP_PHYS_CONN_INIT_CODED_BIT |
                    GAP_PHYS_CONN_INIT_2M_BIT |
                    GAP_PHYS_CONN_INIT_1M_BIT;
        break;

    default:
        break;
    }

cause = le_connect(init_phys, dev_list[dev_idx].bd_addr,
                   (T_GAP_REMOTE_ADDR_TYPE)dev_list[dev_idx].bd_type,
                   local_addr_type,
                   1000);

.....
}

const T_USER_CMD_TABLE_ENTRY user_cmd_table[] =
{
    .....
    {
        "condev",
        "condev [idx] [init_phys]\r\n",
        "Connect to remote device: use showdev to show idx\r\n\r\n"
        "[idx]: use cmd showdev to show idx before use this cmd\r\n\r\n"
        "[init_phys]: bit 0(LE 1M PHY) and bit 2(LE Coded PHY), at least one bit is set to one\r\n\r\n"
        "sample: condev 0 0x100\r\n",
        cmd_condev
    },
    .....
};

命令 disc 通过调用 le_disconnect() 终止已建立的 connection，示例代码如下所示。

```

```

static T_USER_CMD_PARSE_RESULT cmd_disc(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    uint8_t conn_id = p_parse_value->dw_param[0];
    T_GAP_CAUSE cause;
    cause = le_disconnect(conn_id);
    return (T_USER_CMD_PARSE_RESULT)cause;
}

```

4.7.5.3 重组 advertising 数据命令

若宏定义 APP_RECOMBINE_ADV_DATA 为 1，重组 advertising 数据命令包括 sadvdata 和 radvdata。

若用户需要开发关于 advertising 数据重组的功能，参见以下章节以及 [Advertising 数据重组](#)。

sadvdata 命令用于停止 advertising 数据的重组，在 scanning 被停止之后失效。若停止 scanning，在下一次 scanning 时将使能 advertising 数据重组。示例代码如下所示。

```
static T_USER_CMD_PARSE_RESULT cmd_sadvdata(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    ext_adv_data->flag = true;
    ext_adv_data->data_len = 0;
    memset(ext_adv_data->bd_addr, 0, GAP_BD_ADDR_LEN);
    return (RESULT_SUCESS);
}

void app_handle_dev_state_evt(T_GAP_DEV_STATE new_state, uint16_t cause)
{
    .....
    if (gap_dev_state.gap_scan_state != new_state.gap_scan_state)
    {
        if (new_state.gap_scan_state == GAP_SCAN_STATE_IDLE)
        {
            .....
            /* Reset flags of recombining advertising data when stop scanning */
#if APP_RECOMBINE_ADV_DATA
            ext_adv_data->flag = false;
            ext_adv_data->data_len = 0;
            memset(fail_bd_addr, 0, GAP_BD_ADDR_LEN);
#endif
#endif
        }
        .....
    }

    gap_dev_state = new_state;
}
```

命令 radvdata 用于启动或重置 advertising 数据重组，示例代码如下所示。

```
static T_USER_CMD_PARSE_RESULT cmd_radvdata(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    ext_adv_data->flag = false;
    ext_adv_data->data_len = 0;
    memset(fail_bd_addr, 0, GAP_BD_ADDR_LEN);
    return (RESULT_SUCESS);
}
```

4.7.6 测试步骤

首先，编译并将 BLE BT5 Central application 下载到 evolution board，通过 MPTool 配置链路数目。关

于 LE Link 数目的内容参见 [LE Link 数目的配置](#)。BLE BT5 Central application 的基本功能如上所述，为实现其它复杂功能，用户可以参考 SDK 提供的使用手册和源代码进行开发。

通过使用[用户命令](#) 中描述的命令，LE Advertising Extensions 的部分流程如下所示。通过 DebugAnalyser Tool 可获取 log。

4.7.6.1 与 BLE BT5 Peripheral Application 设备测试

可以使用 BLE BT5 peripheral application 与 BLE BT5 central application 进行测试，更多关于 peripheral 的信息参见 [BLE BT5 Peripheral Application](#)。根据 BT5 peripheral application 的配置，将测试步骤分为两类：Primary Advertising Channel 是 LE 1M PHY，Secondary Advertising Channel 是 LE 2M PHY；Primary Advertising Channel 是 LE Coded PHY，Secondary Advertising Channel 是 LE Coded PHY。

1. Primary Advertising Channel 是 LE 1M PHY，Secondary Advertising Channel 是 LE 2M PHY

首先，将宏定义 ADVERTISING_PHY 设为 APP_PRIMARY_1M_SECONDARY_2M，编译并将 BLE BT5 peripheral application 下载到 evolution board。使用 MPTool 将蓝牙地址设置为"x00 x11 x22 x33 x44 x00"。

1) Extended Scanning

步骤描述：查找附近处于可发现状态的 BLE 设备

步骤如下所示：

[escan 2 5](#) - Scan mode 设置为 2，设备持续 scanning 的时间不能超过 Duration 参数。Scan PHYs 设置为 5 (LE 1M PHY 和 LE Coded PHY)。启动 extended scanning 且 scanning 时间不超过 Duration，查看附近处于可发现状态且 primary advertising channel 是 LE 1M PHY 或 LE Coded PHY 的 BLE 设备的信息。如下所示，当 scanning 时间超过 Duration，设备将停止 extended scanning。

Log tool shows :

```
18-05-08#13:56:06.541 161 60234 [APP] !**GAP scan start  
18-05-08#13:56:11.562 255 65256 [APP] !**GAP scan stop
```

串口助工具同样显示 scan 状态。

Serial port assistant tool shows :

```
escan 2 5  
GAP scan start  
GAP scan stop
```

若 BT5 central 设备使用 escan 将 scan mode 设置为 0 或 1，BT5 central 设备可以使用 [stopescan](#) 命令停止 extended scanning。

宏定义 APP_RECOMBINE_ADV_DATA 为 1 时，来自 BT5 peripheral 设备的第一个 advertising report 如下所示。该 advertising report 表示 BT5 peripheral 设备使用 extended advertising PDUs 发送 Connectable Undirected Advertising，且 primary advertising PHY 是 LE 1M PHY，secondary advertising PHY 是 LE 2M PHY。

由于会收到更多的数据，BT5 central设备启动重组流程并等待更多的数据。

Log tool shows :

```
[APP] app_gap_callback: cb_type = 0x50
[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:connectable 1, scannable 0, direct 0, scan
response 0, legacy 0, data status 0x1
[APP]      !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:event_type      0x1,      bd_addr
00::11::22::33::44::00, addr_type 0, rssi -45, data_len 229
[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:primary_phy 1, secondary_phy 2, adv_sid 0,
tx_power 127, peri_adv_interval 0
[APP]      !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:direct_addr_type   0x0,      direct_addr
00::00::00::00::00::00
[APP] !**app_handle_ext_adv_report: Old ext_adv_data->flag is 0, data status is 0x1
[APP] !**app_handle_ext_adv_report:First Data from bd_addr 00::11::22::33::44::00, data length is
229, and waiting more data
[APP] !**app_handle_ext_adv_report: New ext_adv_data->flag is 1
```

来自 BT5 peripheral 设备的最后一个 advertising report 如下所示。该 advertising report 表示数据是完整的。BT5 central 设备完成 advertising 数据重组，来自 BT5 peripheral 设备的 advertising 数据长度为 245 字节。

Log tool shows :

```
[APP] app_gap_callback: cb_type = 0x50
[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:connectable 1, scannable 0, direct 0, scan
response 0, legacy 0, data status 0x0
[APP]      !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:event_type      0x1,      bd_addr
00::11::22::33::44::00, addr_type 0, rssi -45, data_len 16
[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:primary_phy 1, secondary_phy 2, adv_sid 0,
tx_power 127, peri_adv_interval 0
[APP]      !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:direct_addr_type   0x0,      direct_addr
00::00::00::00::00::00
[APP] !**app_handle_ext_adv_report: Old ext_adv_data->flag is 1, data status is 0x0
[APP] !**app_handle_ext_adv_report: Data from bd_addr 00::11::22::33::44::00 is complete, event
type is 0x1, total data length is 245
[APP] !**app_handle_ext_adv_report: First five datas are 0x2, 0x1, 0x5, 0x13, 0x9
[APP] !**app_handle_ext_adv_report: Last five datas are 0xd5, 0xd6, 0xd7, 0xd8, 0xd9
[APP] !**app_handle_ext_adv_report: New ext_adv_data->flag is 0
```

showdev - 显示 scan 设备列表

Serial port assistant tool shows :

dev list

RemoteBd[0] = [97:23:89:45:67:10]

RemoteBd[1] = [00:11:22:33:44:04]

RemoteBd[2] = [00:11:22:33:44:00]

RemoteBd[3] = [00:11:22:33:44:02]

2) Connection 和 Reconnection

步骤描述：与 peripheral 设备创建 connection，断开 connection

步骤如下所示：

condev 2 x001 - 对 primary advertising channel 为 LE 1M PHY 的 connectable advertisements 进行 scan，并发起 connection。Secondary advertising channel 是 LE 2M PHY，因此 TX PHY 和 RX PHY 类型均为 LE 2M PHY。

Serial port assistant tool shows :

condev 2 x001

Connected success conn_id 0, tx_phy 2, rx_phy 2

showcon - 显示 connection 信息

Serial port assistant tool shows :

showcon

ShowCon conn_id 0 state 0x00000002 role 1

RemoteBd = [00:11:22:33:44:00] type = 0

active link num 1, idle link num 0

disc 0 - 与 peripheral 设备断开 connection

Serial port assistant tool shows :

disc 0

Disconnect conn_id 0, dis_cause 0x00000116

condev 2 x001 - 与 peripheral 设备重新建立 connection

Serial port assistant tool shows :

condev 2 x001

Connected success conn_id 0, tx_phy 2, rx_phy 2

2. Primary Advertising Channel 是 LE Coded PHY, Secondary Advertising Channel 是 LE Coded PHY

首先, 将宏定义 ADVERTISING_PHY 设为 APP_PRIMARY_CODED_SECONDARY_CODED, 编译并将 BLE BT5 peripheral application 下载到 evolution board。使用 MPTool 将蓝牙地址设置为"x00 x11 x22 x33 x44 x00"。

1) Extended Scanning

步骤描述: 查找附近处于可发现状态的 BLE 设备

步骤如下所示:

escan 0 5 - Scan mode 设置为 0, 设备持续 scanning 直到停止 scanning。Scan PHYs 设置为 5 (LE 1M PHY 和 LE Coded PHY)。启动 extended scanning, 查看附近处于可发现状态且 primary advertising channel 是 LE 1M PHY 或 LE Coded PHY 的 BLE 设备的信息。

Log tool shows :

```
[APP] !**GAP scan start
```

串口助手机具同样显示 scan 状态。

Serial port assistant tool shows :

```
escan 0 5
```

```
GAP scan start
```

stopescan - 停止 extended scanning

Log tool shows :

```
[APP] !**GAP scan stop
```

串口助手机具同样显示 scan 状态。

Serial port assistant tool shows :

```
stopescan
```

```
GAP scan stop
```

宏定义 APP_RECOMBINE_ADV_DATA 为 1 时, 来自 BT5 peripheral 设备的第一个 advertising report 如下所示。该 advertising report 表示 BT5 peripheral 设备使用 extended advertising PDUs 发送 Connectable Undirected Advertising, 且 primary advertising PHY 是 LE Coded PHY, secondary advertising PHY 是 LE Coded PHY。由于会收到更多的数据, BT5 central 设备启动重组流程并等待更多的数据。

Log tool shows :

```
[APP] app_gap_callback: cb_type = 0x50
[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:connectable 1, scannable 0, direct 0, scan
response 0, legacy 0, data status 0x1
[APP]     !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:event_type      0x1,      bd_addr
00::11::22::33::44::00, addr_type 0, rssi -41, data_len 229
[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:primary_phy 3, secondary_phy 3, adv_sid 0,
tx_power 127, peri_adv_interval 0
[APP]     !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:direct_addr_type  0x0,      direct_addr
00::00::00::00::00::00
[APP] !**app_handle_ext_adv_report: Old ext_adv_data->flag is 0, data status is 0x1
[APP] !**app_handle_ext_adv_report:First Data from bd_addr 00::11::22::33::44::00, data length is
229, and waiting more data
[APP] !**app_handle_ext_adv_report: New ext_adv_data->flag is 1
```

来自 BT5 peripheral 设备的最后一个 advertising report 如下所示。该 advertising report 表示数据是完整的。BT5 central 设备完成 advertising 数据重组，来自 BT5 peripheral 设备的 advertising 数据长度为 245 字节。

Log tool shows :

```
[APP] app_gap_callback: cb_type = 0x50
[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:connectable 1, scannable 0, direct 0, scan
response 0, legacy 0, data status 0x0
[APP]     !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:event_type      0x1,      bd_addr
00::11::22::33::44::00, addr_type 0, rssi -41, data_len 16
[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:primary_phy 3, secondary_phy 3, adv_sid 0,
tx_power 127, peri_adv_interval 0
[APP]     !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:direct_addr_type  0x0,      direct_addr
00::00::00::00::00::00
[APP] !**app_handle_ext_adv_report: Old ext_adv_data->flag is 1, data status is 0x0
[APP] !**app_handle_ext_adv_report: Data from bd_addr 00::11::22::33::44::00 is complete, event
type is 0x1, total data length is 245
[APP] !**app_handle_ext_adv_report: First five datas are 0x2, 0x1, 0x5, 0x13, 0x9
[APP] !**app_handle_ext_adv_report: Last five datas are 0xd5, 0xd6, 0xd7, 0xd8, 0xd9
[APP] !**app_handle_ext_adv_report: New ext_adv_data->flag is 0
```

showdev - 显示 scan 设备列表

Serial port assistant tool shows :

dev list

.....

RemoteBd[3] = [47:78:39:48:32:1a]

RemoteBd[4] = [00:11:22:33:44:00]

2) Connection 和 Reconnection

步骤描述：与 peripheral 设备创建 connection，断开 connection

步骤如下所示：

condev 4 x100 - 对 primary advertising channel 为 LE Coded PHY 的 connectable advertisements 进行 scan，并发起 connection。Secondary advertising channel 是 LE Coded PHY，因此 TX PHY 和 RX PHY 类型均为 LE Coded PHY。

Serial port assistant tool shows :

condev 4 x100

Connected success conn_id 0, tx_phy 3, rx_phy 3

showcon - 显示 connection 信息

Serial port assistant tool shows :

showcon

ShowCon conn_id 0 state 0x00000002 role 1

RemoteBd = [00:11:22:33:44:00] type = 0

active link num 1, idle link num 0

disc 0 - 与 peripheral 设备断开 connection

Serial port assistant tool shows :

disc 0

Disconnect conn_id 0, dis_cause 0x00000116

condev 4 x100 - 与 peripheral 设备重新建立 connection

Serial port assistant tool shows :

condev 4 x100

Connected success conn_id 0, tx_phy 3, rx_phy 3

4.8 BLE Application 用户命令

BLE Central Application、BLE Scatternet Application 和 BLE BT5 Central Application 支持用户命令，需要通过个人电脑的 Data UART (PC 端的 USB 端口通过 USB 转串口模块连接到 evolution board) 输入命令进行交互。

4.8.1 用户命令的实现

文件描述如下所示。

表 4-12 用户命令文件

File name	Description
data_uart.c	Initialize Data UART and print data through data UART.
data_uart_dlps.c	Data UART DLPS initialization.
user_cmd_parse.c	Used to parse user command from lower Data UART data and execute right commands.
user_cmd.c	Define user commands

源代码路径为 sdk\src\ mcu\module\data_uart_cmd。

1. 初始化

data_uart_init()用于初始化 data UART, user_cmd_init()用于初始化用户命令模块。

```
void app_main_task(void *p_param)
{
    uint8_t event;
    .....
    data_uart_init(evt_queue_handle, io_queue_handle);
    user_cmd_init(&user_cmd_if, "central");
    .....
}
```

2. Data UART RX 处理

```
void app_handle_io_msg(T_IO_MSG io_msg)
{
    uint16_t msg_type = io_msg.type;
    uint8_t rx_char;

    switch (msg_type) {
        .....
        case IO_MSG_TYPE_UART:
            /* We handle user command informations from Data UART in this branch. */
            rx_char = (uint8_t)io_msg.subtype;
            user_cmd_collect(&user_cmd_if, &rx_char, sizeof(rx_char), user_cmd_table);
    }
}
```

```
    break;
default:
    break;
}
}
```

user_cmd_collect()用于收集命令字符并执行命令。用户可以从串口助手工具输入命令，命令使用 Enter 作为结束。

3. Data UART TX 处理

Data UART TX 用于输出信息。

```
void app_handle_conn_state_evt(T_IO_MSG io_msg)
{
    .....
case GAP_CONN_STATE_CONNECTED: {
    le_get_conn_addr(conn_id, app_link_table[conn_id].bd_addr,
                    &app_link_table[conn_id].bd_type);
    data_uart_print("Connected success conn_id %d\r\n", conn_id);
}
break;
}
```

data_uart_print()用于通过 Data UART 输出信息，用户可以在串口助手工具查阅信息。

4. 定义用户命令表

```
static T_USER_CMD_PARSE_RESULT cmd_conupdreq(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    T_GAP_CAUSE cause;
    uint8_t conn_id = p_parse_value->dw_param[0];
    uint16_t conn_interval_min = p_parse_value->dw_param[1];
    uint16_t conn_interval_max = p_parse_value->dw_param[2];
    uint16_t conn_latency = p_parse_value->dw_param[3];
    uint16_t supervision_timeout = p_parse_value->dw_param[4];
    cause = le_update_conn_param(conn_id,
                                conn_interval_min,
                                conn_interval_max,
                                conn_latency,
                                supervision_timeout,
                                2 * (conn_interval_min - 1),
                                2 * (conn_interval_max - 1)
                                );
    return (T_USER_CMD_PARSE_RESULT)cause;
}

const T_USER_CMD_TABLE_ENTRY user_cmd_table[] = {
    /****** Common cmd ******/
    {
```

```

    "conupdreq",
    "conupdreq [conn_id] [interval_min] [interval_max] [latency] [supervision_timeout]\n\r",
    "LE connection param update request\n\r"
    sample: conupdreq 0 0x30 0x40 0 500\n\r",
    cmd_conupdreq
},
.....
}

```

4.8.2 Data UART 连接

BLE Central Application、BLE Scatternet Application 和 BLE BT5 Central Application 默认使用 P3_0 作为 Data UART 的 TX 引脚，使用 P3_1 作为 Data UART 的 RX 引脚。

每个 APP 都有一份 board.h 文件，其中包含 Data UART 引脚的定义。

```
#define DATA_UART_TX_PIN    P3_0
#define DATA_UART_RX_PIN    P3_1
```

可以根据硬件环境配置 Data UART 引脚。

PC 端通过 USB 转串口模块连接到运行 BLE Central Application、BLE Scatternet Application 或 BLE BT5 Central Application 的 evolution board 的 Data UART，如图 4-14 所示。

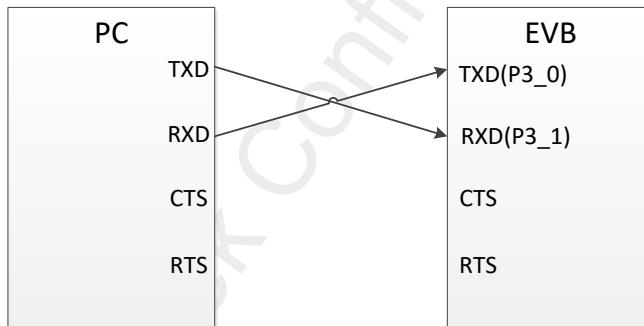


图 4-14 PC 和 EVB 之间的 Data UART 连接

Data UART 的波特率设置为 115200，PC 端串口助手工具的参数设置如图 4-15 所示。

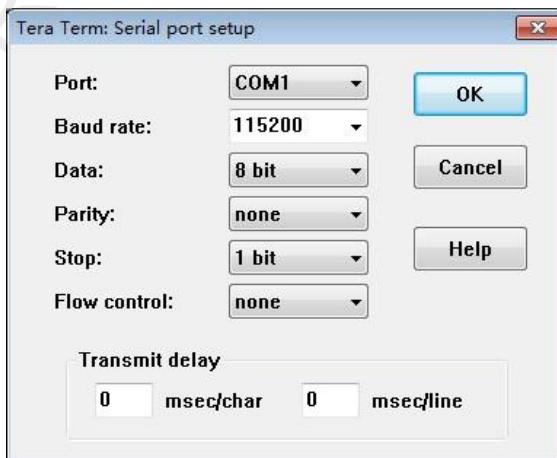


图 4-15 串口设置

4.8.3 用户命令

本节介绍 BLE Central Application 和 BLE Scatternet Application 用户命令的使用方法。以"conupdreq [conn_id] [type]"为例，"conupdreq"是命令名称，与后续参数之间以空格分隔。用户可以在 PC 的串口助手工具输入命令，然后按 ENTER 或点击“Send”发送命令。推荐的串口助手工具是 Tera Term。

BLE Central Application 的用户命令表如下所示：

```
/** @brief User command table */

const T_USER_CMD_TABLE_ENTRY user_cmd_table[] = {
    /***** Common cmd *****/
    {
        "conupdreq",
        "conupdreq [conn_id] [interval_min] [interval_max] [latency] [supervision_timeout]\r\n",
        "LE connection param update request\r\n",
        sample: conupdreq 0x30 0x40 0 500\r\n",
        cmd_conupdreq
    },
    {
        "showcon",
        "showcon\r\n",
        "Show all devices connecting status\r\n",
        cmd_showcon
    },
    {
        "disc",
        "disc [conn_id]\r\n",
        "Disconnect to remote device\r\n",
        cmd_disc
    },
    {
        "authmode",
        "authmode [auth_flags] [io_cap] [sec_enable] [oob_enable]\r\n",
        "Config authentication mode\r\n",
        [auth_flags]:authentication req bit field: bit0-(bonding), bit2-(MITM), bit3-(SC)\r\n,
        [io_cap]:set io Capabilities: 0-(display only), 1-(display yes/no), 2-(keyboard only), 3-(no IO), 4-(keyboard
            display)\r\n,
        [sec_enable]:Start smp pairing procedure when connected: 0-(disable), 1-(enable)\r\n,
        [oob_enable]:Enable oob flag: 0-(disable), 1-(enable)\r\n,
        sample: authmode 0x5 2 1 0\r\n",
        cmd_authmode
    },
    .....
    /* MUST be at the end: */
}
```

```
{  
    0,  
    0,  
    0,  
    0  
}  
};
```

4.9 BLE Peripheral Privacy Application

4.9.1 简介

本节内容是 privacy 管理模块的使用示例。BLE peripheral privacy 工程是基于 BLE peripheral 工程实现的 privacy 管理模块使用示例。

更多关于 peripheral 角色的内容参见 [BLE Peripheral Application](#)。

更多关于 privacy 管理模块的内容参见 [Privacy 管理模块](#)。

4.9.2 工程概述

本节内容介绍工程的路径和结构，相关文件路径如下所示：

- 工程路径为 sdk\board\evb\ble_peripheral_privacy
- 工程源代码路径为 sdk\src\sample\ble_peripheral_privacy

工程目录结构如图 4-16 所示：

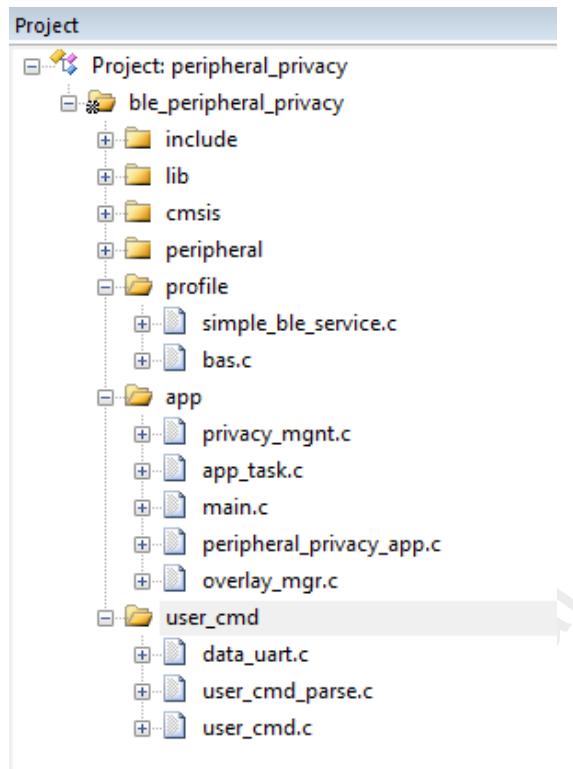


图 4-16 Peripheral Privacy 工程目录结构

文件列表分为以下类别。

表 4-13 Peripheral Privacy 工程文件列表

Directory	Description
include	rom_uuid.h: ROM UUID header files. User need not modify.
lib	The protocol stack and gap library file. User need not modify.
cmsis	The cmsis source code. User need not modify.
profile	The GATT profiles source code.
app	The application source code.
user_cmd	The user command source code.

4.9.3 Privacy 使用流程图

在 APP 中有两种模式。处于配对模式的设备可以与任何设备建立连线。处于配对模式的设备必须禁用 address resolution 和 white list filter policy。处于回连模式的设备只能与在 resolving list 和 white list 中的设备建立连线。若 APP 希望过滤使用 resolvable private address 的设备，APP 必须调用函数 privacy_set_addr_resolution() 以启用 address resolution。

更多关于 privacy 管理模块的信息参见 [Privacy 管理模块](#)。

peripheral privacy application 的流程图如图 4-17 所示。

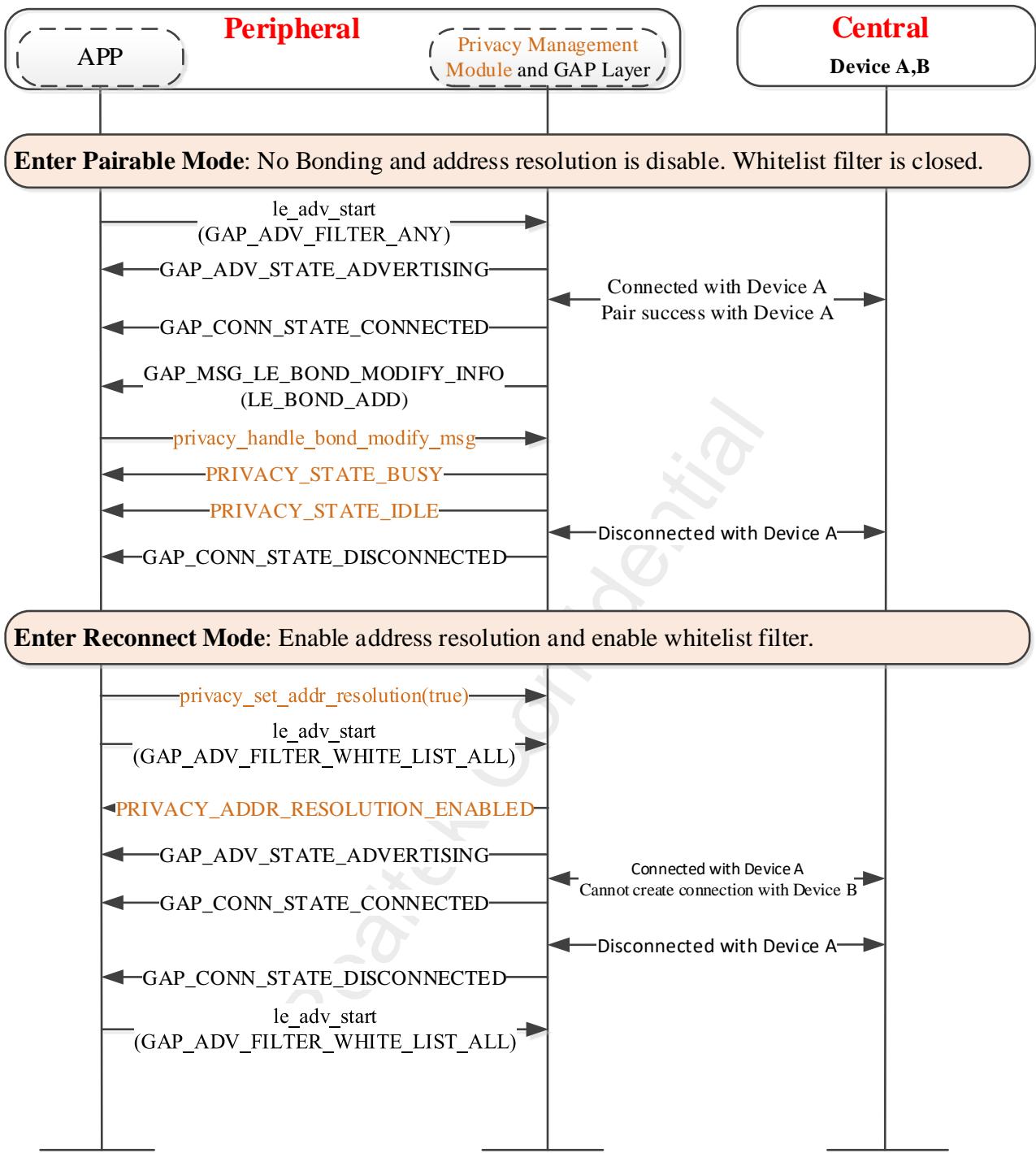


图 4-17 Peripheral Privacy APP 流程图

4.9.4 APP 的可配置功能

在 app_flags.h 中定义 APP 的可配置功能。

```
/** @brief Configure APP LE link number */
#define APP_MAX_LINKS 1

/** @brief User command: 0-Close user command, 1-Open user command */
#define USER_CMD_EN 1

/** @brief Configure Privacy1.2 feature: 0-Closed, 1-Open */
#define APP_PRIVACY_EN 1

#if APP_PRIVACY_EN
/** @brief Configure the authentication requirement of simple_ble_service.c */
#define SIMP_SRV_AUTHEN_EN 1
#endif
```

4.9.4.1 Privacy 配置

所有 privacy 相关的代码通过宏定义 APP_PRIVACY_EN 分隔。

4.9.4.2 Service 的 Security 要求

所有 service security 相关的代码通过宏定义 SIMP_SRV_AUTHEN_EN 分隔。

若 APP 希望使用 privacy，那么 APP 需要启用 security。

更多信息参见 [Service 的 security 要求](#)。

```
/* client characteristic configuration */
{
    ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_CCCD_APPL, /* flags */
    {
        LO_WORD(GATT_UUID_CHAR_CLIENT_CONFIG), /* type_value */
        HI_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
        /* NOTE: this value has an instantiation for each client, a write to */
        /* this attribute does not modify this default value: */ /* */
        LO_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT), /* client char. config. bit field */
        HI_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT)
    },
    2, /* bValueLen */
    NULL,
#endif SIMP_SRV_AUTHEN_EN
    (GATT_PERM_READ_AUTHEN_REQ | GATT_PERM_WRITE_AUTHEN_REQ) /* permissions */
#else
    (GATT_PERM_READ | GATT_PERM_WRITE) /* permissions */
#endif
},
```

4.9.5 用户命令

使用 BLE Peripheral Privacy Application 时，需要通过个人电脑的 Data UART 输入命令进行交互。更多信息参见 [BLE Application 用户命令](#)。以下章节介绍 Peripheral Privacy application 使用的命令。

支持的用户命令如下所示：

```
const T_USER_CMD_TABLE_ENTRY user_cmd_table[] =  
{  
    /***** Common cmd *****/  
    {  
        "disc",  
        "disc [conn_id]\n\r",  
        "Disconnect to remote device\n\r",  
        cmd_disc  
    },  
    {  
        "bondclear",  
        "bondclear\n\r",  
        "Clear all bonded devices information\n\r",  
        cmd_bondclear  
    },  
    ....  
};
```

命令 disc 用于断开 LE 链路。

命令 bondclear 用于清除所有的绑定信息。

4.9.6 测试步骤

首先，编译并将 BLE Peripheral Privacy application 下载到 evolution board。

当 BLE Peripheral Privacy application 在 evolution board 上运行时，设备将是可连接的。对端设备可以对 peripheral 设备进行 scan，并创建 connection。在连接断开之后，Peripheral Privacy APP 将恢复为可连接状态。

4.9.6.1 与 iOS 设备测试

步骤简介：基于 iOS 的设备与 BLE 兼容，因此可以 discover 运行 BLE Peripheral Application 的设备。推荐从 App Store 下载 BLE 相关的 APP (例如 LightBlue) 以执行 scan 和 connection 测试。

测试步骤：在 iOS 设备上运行 LightBlue 进行 scan，与 BLE_PRIVACY 设备创建 connection，如图 4-18 所示。

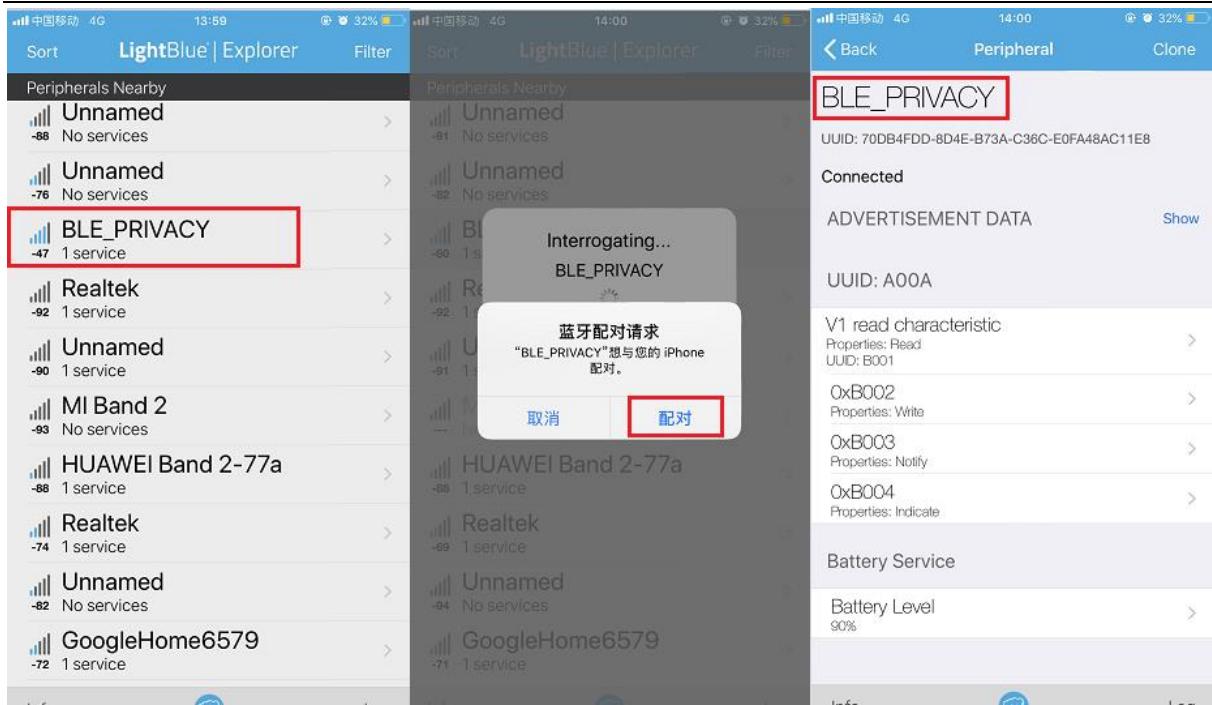


图 4-18 与 iOS 设备测试

参考文献

[1] Bluetooth SIG. Core_v5.0 [M]. 2016, 169.

[2] RTL8762C Memory User Guide

[3] RTL8762C Deep Low Power State

Realtek Confidential