

Telink

# 泰凌 tl\_ble\_sdk

## SDK 开发手册

AN-22063001-C3

Ver1.0.2

2025.06.25

### Keyword

多连接, 低功耗蓝牙 (BLE), Master/Central/主机, Slave/Perpheral/从机, SMP, PM, Host, Controller, HCI

### Brief

本文档为泰凌微电子 BLE 多连接 SDK V4.044 版本的开发指南, 适用泰凌 B91, B92, TL721x, TL321x, TL322x 系列芯片。



**Published by**  
**Telink Semiconductor**

**11F, Building 1, 61 Shengxia Road,  
Pudong District, Shanghai, China**

**© Telink Semiconductor  
All Rights Reserved**

#### **Legal Disclaimer**

This document is provided as-is. Telink Semiconductor reserves the right to make improvements without further notice to this document or any products herein. This document may contain technical inaccuracies or typographical errors. Telink Semiconductor disclaims any and all liability for any errors, inaccuracies or incompleteness contained herein.

Copyright © 2025 Telink Semiconductor (Shanghai) Co., Ltd.

#### **Information**

For further information on the technology, product and business term, please contact Telink Semiconductor Company [www.telink-semi.com](http://www.telink-semi.com)

For sales or technical support, please send email to the address of:

[telinksales@telink-semi.com](mailto:telinksales@telink-semi.com)

[telinksupport@telink-semi.com](mailto:telinksupport@telink-semi.com)



## 修订历史

版本	发布时间	修改内容
V1.0.0	2022-06	初次发布
V1.0.1	2024-09	添加 Flash 写保护章节
V1.0.2	2025-06	SDK 版本对齐到 V4.0.4.4



# Contents

<b>修订历史</b>	3
<b>1 SDK 介绍</b>	<b>10</b>
1.1 适用 IC . . . . .	10
1.2 软件组织架构 . . . . .	11
1.2.1 vendor . . . . .	12
1.2.1.1 main.c . . . . .	13
1.2.1.2 app_config.h . . . . .	13
1.2.1.3 application files . . . . .	13
1.2.1.4 BLE stack entry . . . . .	13
1.3 版本号 . . . . .	14
<b>2 MCU 基础模块</b>	<b>15</b>
2.1 MCU 地址空间 . . . . .	15
2.1.1 Flash . . . . .	15
2.1.2 SRAM . . . . .	16
2.1.3 MCU 地址空间分配 . . . . .	17
2.1.4 Flash 和 SRAM 空间分配 . . . . .	17
2.1.4.1 各个段的介绍 . . . . .	18
2.2 时钟模块 . . . . .	22
2.2.1 System Timer 的使用 . . . . .	23
2.3 中断嵌套 . . . . .	24
2.3.1 中断嵌套功能简述 . . . . .	24
2.3.2 中断嵌套的使用 . . . . .	26
2.3.2.1 App 普通中断 . . . . .	26
2.3.2.2 App 高级中断 . . . . .	26
2.3.3 中断使用限制 . . . . .	26
<b>3 BLE 模块</b>	<b>27</b>
3.1 BLE SDK 软件架构 . . . . .	27
3.1.1 标准 BLE SDK 软件架构 . . . . .	27
3.1.2 Telink BLE SDK 软件架构 . . . . .	28
3.1.2.1 Telink BLE Multiple Connection Controller . . . . .	28
3.1.2.2 Telink BLE Multiple Connection Whole Stack (Controller+Host) . . . . .	29
3.2 Controller . . . . .	31
3.2.1 Connection Number 配置 . . . . .	31
3.2.1.1 supportedMaxCentralNum & supportedMaxPeripheralNum . . . . .	31
3.2.1.2 appMaxCentralNum & appMaxPeripheralNum . . . . .	31
3.2.1.3 currentMaxCentralNum & currentMaxPeripheralNum . . . . .	32
3.2.2 Link Layer 状态机 . . . . .	32
3.2.2.1 Link Layer 状态机初始化 . . . . .	32
3.2.2.2 Link Layer 状态组合 . . . . .	33
3.2.3 Link Layer 时序 . . . . .	36
3.2.3.1 Standby state 时序 . . . . .	36
3.2.3.2 Scanning only, no Advertising, no Connection 时序 . . . . .	37
3.2.3.3 Advertising only, no Scanning, no Connection 时序 . . . . .	37
3.2.3.4 Advertising, Scanning, no Connection 时序 . . . . .	38



3.2.3.5 Connection, Advertising, Scanning 时序 . . . . .	38
3.2.3.6 Connection, no Advertising, no Scanning 时序 . . . . .	39
3.2.4 ACL TX FIFO & ACL RX FIFO . . . . .	40
3.2.4.1 ACL TX FIFO 定义及设置 . . . . .	40
3.2.4.2 ACL RX FIFO 定义及设置 . . . . .	46
3.2.4.3 RX overflow 分析 . . . . .	47
3.2.5 MTU 和 DLE 概念及使用方法 . . . . .	48
3.2.5.1 MTU 和 DLE 概念说明 . . . . .	48
3.2.5.2 MTU 和 DLE 自动交互方法 . . . . .	49
3.2.5.3 MTU 和 DLE 手动交互方法 . . . . .	50
3.2.6 Coded PHY/2M PHY . . . . .	50
3.2.6.1 Code PHY/2M PHY API 介绍 . . . . .	50
3.2.7 Channel Selection Algorithm #2 . . . . .	51
3.2.8 Controller API . . . . .	51
3.2.8.1 BLE MAC address 初始化 . . . . .	52
3.2.8.2 bls_ll_setAdvData . . . . .	52
3.2.8.3 bls_ll_setScanRspData . . . . .	53
3.2.8.4 bls_ll_setAdvParam . . . . .	53
3.2.8.5 bls_ll_setAdvEnable . . . . .	56
3.2.8.6 blc_ll_setAdvCustomedChannel . . . . .	57
3.2.8.7 blc_ll_setScanParameter . . . . .	57
3.2.8.8 blc_ll_setScanEnable . . . . .	58
3.2.8.9 blc_ll_createConnection . . . . .	59
3.2.8.10 blc_ll_setCreateConnectionTimeout . . . . .	61
3.2.8.11 blc_ll_setAclCentralConnectionInterval . . . . .	61
3.2.8.12 blc_ll_setAutoExchangeDataLengthEnable . . . . .	61
3.2.8.13 blc_ll_sendDateLengthExtendReq . . . . .	62
3.2.8.14 blc_ll_setDataLengthReqSendingTime_after_connCreate . . . . .	62
3.2.8.15 blc_ll_disconnect . . . . .	62
3.2.8.16 rf_set_power_level_index . . . . .	63
3.2.8.17 Whitelist & Resolvinglist . . . . .	63
3.3 Host Controller Interface . . . . .	65
3.3.1 HCI 软件架构 . . . . .	65
3.3.1.1 H4 Protocol . . . . .	67
3.3.1.2 H5 Protocol . . . . .	68
3.3.1.3 HCI Transport Control . . . . .	74
3.3.1.4 Controller HCI . . . . .	75
3.3.2 Controller Event . . . . .	77
3.3.2.1 Controller HCI Event 分类 . . . . .	77
3.3.2.2 常用 Controller HCI event . . . . .	78
3.3.2.3 常用 HCI LE event . . . . .	79
3.4 Host . . . . .	82
3.4.1 L2CAP . . . . .	82
3.4.1.1 注册 L2CAP 数据处理函数 . . . . .	82
3.4.1.2 更新连接参数 . . . . .	83
3.4.2 ATT & GATT . . . . .	85
3.4.2.1 GATT 基本单位 Attribute . . . . .	85



3.4.2.2 Attribute and ATT Table . . . . .	86
3.4.2.3 GATT Service Security . . . . .	96
3.4.2.4 Attribute PDU and GATT API . . . . .	98
3.4.3 GAP . . . . .	110
3.4.3.1 GAP 初始化 . . . . .	110
3.4.3.2 GAP Event . . . . .	110
3.4.4 SMP . . . . .	115
3.4.4.1 SMP 安全等级 . . . . .	115
3.4.4.2 SMP 参数配置 . . . . .	116
3.4.4.3 SMP 流程配置 . . . . .	119
3.4.4.4 SMP 配对方法 . . . . .	121
3.4.4.5 SMP Storage . . . . .	123
3.4.5 Device Manage & Simple SDP . . . . .	127
3.5 LE Advertising Extensions . . . . .	130
3.5.1 扩展广播 (Extended Advertising) . . . . .	130
3.5.1.1 广播集 (Advertising Sets) . . . . .	133
3.5.1.2 Extended Advertising 相关的 API 介绍 . . . . .	133
3.5.2 周期广播 (Periodic Advertising) . . . . .	135
3.5.2.1 Periodic Advertising 相关的 API 介绍 . . . . .	137
3.5.3 扩展扫描 (Extended SCAN) . . . . .	138
3.5.3.1 Extended SCAN 相关的 API 介绍 . . . . .	139
3.5.4 周期扫描 (Periodic SCAN) . . . . .	139
3.5.4.1 periodic scan 相关的 API 介绍 . . . . .	140
3.5.5 Periodic Advertising Sync Transfer (PAST) . . . . .	141
3.5.5.1 PAST 模式 1 . . . . .	142
3.5.5.2 PAST 模式 2 . . . . .	142
3.5.5.3 PAST 相关 API 介绍 . . . . .	143
3.5.6 Periodic Advertising with Response (PAwR) . . . . .	144
3.5.6.1 PAwR 基本原理 . . . . .	144
3.5.6.2 PAwR 同步 . . . . .	146
3.5.6.3 PAwR 相关 API 介绍 . . . . .	146
<b>4 低功耗管理 . . . . .</b>	<b>149</b>
4.1 低功耗驱动 . . . . .	149
4.1.1 低功耗模式 . . . . .	149
4.1.2 低功耗唤醒源 . . . . .	150
4.1.3 低功耗模式的进入和唤醒 . . . . .	151
4.1.4 低功耗唤醒后运行流程 . . . . .	153
4.1.5 API pm_is MCU_deepRetentionWakeup . . . . .	156
4.2 BLE 低功耗管理 . . . . .	156
4.2.1 BLE PM 初始化 . . . . .	156
4.2.2 BLE PM for Link Layer . . . . .	156
4.2.2.1 Sleep for advertising "only advertising" . . . . .	157
4.2.2.2 Sleep for scanning "only scanning" . . . . .	157
4.2.2.3 Sleep for connection . . . . .	158
4.2.3 相关变量 . . . . .	158
4.2.4 API blc_pm_setSleepMask . . . . .	159
4.2.5 API blc_pm_setWakeupSource . . . . .	160



4.2.6 API blc_pm_setDeepsleepRetentionType . . . . .	160
4.2.7 API blc_pm_setDeepsleepRetentionEnable . . . . .	161
4.2.8 API blc_pm_setDeepsleepRetentionThreshold . . . . .	161
4.2.9 PM 软件处理流程 . . . . .	162
4.2.9.1 blc_sdk_main_loop . . . . .	162
4.2.9.2 blt_sleep_process . . . . .	163
4.2.10 API blc_pm_getWakeupSystemTick . . . . .	164
4.2.11 API blc_pm_setDeepsleepRetentionEarlyWakeupTiming . . . . .	165
4.3 GPIO 唤醒的注意事项 . . . . .	169
4.3.1 唤醒电平有效时无法进入 sleep mode . . . . .	169
4.4 应用层定时唤醒 . . . . .	169
<b>5 低电检测 . . . . .</b>	<b>171</b>
5.1 低电检测的重要性 . . . . .	171
5.2 低电检测的实现 . . . . .	171
5.2.1 低电检测的注意事项 . . . . .	171
5.2.1.1 建议使用 GPIO 输入通道 . . . . .	172
5.2.1.2 只能使用差分模式 . . . . .	172
5.2.1.3 不同的 ADC 任务需要切换 . . . . .	173
5.2.2 低电检测的单独使用 . . . . .	173
5.2.2.1 低电检测初始化 . . . . .	173
5.2.2.2 低电检测处理 . . . . .	175
5.2.2.3 低压报警 . . . . .	175
5.2.3 低电检测和 Amic Audio . . . . .	176
<b>6 Flash 写保护 . . . . .</b>	<b>177</b>
6.1 Flash 写保护的重要性 . . . . .	177
6.2 Flash 写保护的实现 . . . . .	177
6.2.1 Flash 写保护的使用 . . . . .	177
6.2.1.1 Flash 写保护的初始化 . . . . .	177
6.2.1.2 Flash 写保护处理 . . . . .	178
6.2.1.3 加锁和解锁操作 . . . . .	180
<b>7 OTA . . . . .</b>	<b>181</b>
7.1 FLASH 存储架构 . . . . .	181
7.1.1 传统存储架构 . . . . .	181
7.1.2 Secure Boot 存储架构 . . . . .	182
7.2 OTA 更新流程 . . . . .	184
7.2.1 传统 OTA 更新流程 . . . . .	184
7.2.2 Secure Boot OTA 更新流程 . . . . .	185
7.2.3 修改 Firmware size 和 boot address . . . . .	186
7.3 OTA 模式 RF 数据处理 . . . . .	187
7.3.1 Attribute Table 中 OTA 的处理 . . . . .	187
7.3.2 OTA Protocol . . . . .	188
7.3.3 RF Transfer 处理方法 . . . . .	194
7.4 Q&A . . . . .	203
<b>8 按键扫描 . . . . .</b>	<b>204</b>
8.1 键盘矩阵 . . . . .	204
8.2 Keyscan and Keymap . . . . .	205
8.2.1 Keyscan . . . . .	205



8.2.2 Keymap & kb_event . . . . .	206
8.3 Keyscan Flow . . . . .	207
8.4 Repeat Key 处理 . . . . .	209
<b>9 LED 管理 . . . . .</b>	<b>211</b>
9.1 LED 任务相关调用函数 . . . . .	211
9.2 LED 任务的配置和管理 . . . . .	211
9.2.1 定义 led event . . . . .	211
9.2.2 LED Event 的优先级 . . . . .	212
<b>10 软件定时器 (Software Timer) . . . . .</b>	<b>214</b>
10.1 Timer 初始化 . . . . .	214
10.2 Timer 的查询处理 . . . . .	214
10.3 添加定时器任务 . . . . .	216
10.4 删除定时器任务 . . . . .	217
10.5 Demo . . . . .	217
<b>11 功能参考 Demo . . . . .</b>	<b>220</b>
11.1 feature_backup . . . . .	220
11.2 feature_2M_coded_phy . . . . .	222
11.3 feature_gatt_api . . . . .	224
11.4 feature_ll_more_data . . . . .	226
11.5 feature_dle . . . . .	227
11.6 feature_smp . . . . .	229
11.6.1 Peripheral 和 Central 均不使能 SMP . . . . .	229
11.6.2 Legacy Just Works . . . . .	231
11.6.3 Secure Connections Just Works . . . . .	233
11.6.4 Legacy Passkey Entry Input . . . . .	235
11.6.5 Legacy Passkey Entry Display . . . . .	236
11.6.6 Legacy OOB . . . . .	237
11.6.7 Secure Connections Passkey Entry . . . . .	238
11.6.8 Secure Connections Numeric Comparison . . . . .	240
11.6.9 Secure Connections OOB . . . . .	241
11.6.10 异常处理 . . . . .	242
11.6.10.1 按键没有响应 . . . . .	242
11.6.10.2 LED 灯没有点亮 . . . . .	243
11.7 feature_whitelist . . . . .	244
11.8 feature_soft_timer . . . . .	246
11.9 feature_ota . . . . .	247
11.10 feature_l2cap_coc . . . . .	248
11.11 feature_ext_adv . . . . .	250
11.12 feature_ext_scan . . . . .	251
11.13 feature_per_adv . . . . .	252
11.14 feature_per_adv_sync . . . . .	253
<b>12 其他模块 . . . . .</b>	<b>254</b>
12.1 24MHz 晶体外部电容 . . . . .	254
12.2 32KHz 时钟源选择 . . . . .	254
12.3 PA . . . . .	255
12.4 PhyTest . . . . .	256
12.4.1 PhyTest API . . . . .	256



12.4.2 PhyTest demo . . . . .	256
<b>13 调试方法 . . . . .</b>	<b>258</b>
13.1 GPIO 模拟 UART 打印 . . . . .	258
13.2 BDT 工具读取全局变量的值 . . . . .	258
13.3 BDT 工具的 Memory Access 功能 . . . . .	259
13.4 BDT 工具读 PC 指针 . . . . .	259
13.5 Debug IO . . . . .	260
13.6 USB my_dump_str_data . . . . .	261
13.7 JTAG 使用 . . . . .	261
13.7.1 Diagnostic Report . . . . .	261
13.7.2 Target Configuration . . . . .	263
13.7.3 Flash Programming . . . . .	264
<b>14 附录 . . . . .</b>	<b>266</b>
14.1 附录 1: crc16 算法 . . . . .	266
14.2 附录 2: 检查 stack 是否溢出 . . . . .	266
14.2.1 原理 . . . . .	266
14.2.2 方法 . . . . .	266



# 1 SDK 介绍

tl\_ble\_sdk 提供 BLE 多连接应用的参考代码，用户可以在此基础上开发自己的应用程序。多连接指多个（大于等于 1）Central 或 Peripheral 角色共存，比如自身同时作为 4 个 Central 和 3 个 Peripheral（简称 C4P3）。目前协议栈支持到最多 4 个 Central，和最多 4 个 Peripheral，不支持角色自由转化。

从 tl\_ble\_sdk V4.0.4.4 版本开始，SDK 与 Handbook 同时 Release，该版 Handbook 的内容对应 tl\_ble\_sdk V4.0.4.4。

从 tl\_ble\_sdk V4.0.4.2 版本开始，发布 SDK 从官网 Wiki 移到了 Gitee 和 GitHub 上，代码下载或 Clone 不需要登录账号。为了及时收到 SDK 的更新，建议登录后，关注该仓库。

The screenshot shows the Gitee repository page for 'tl\_ble\_sdk'. At the top right, there is a 'Follow' button with a dropdown menu. The 'Follow' option is checked with a red circle around it. Other options in the menu include 'Unfollow', 'Starred 4', 'Fork 1', and three other less descriptive options.

Figure 1.1: Gitee 关注

The screenshot shows the GitHub repository page for 'tl\_ble\_sdk'. On the right side, there is a dropdown menu for notifications. The 'All Activity' option is checked with a red circle around it. Other options include 'Participating and @mentions', 'Ignore', and 'Custom'.

Figure 1.2: GitHub 关注

## 1.1 适用 IC

当前 tl\_ble\_sdk 适用 B91、B92、TL321X、TL721X、TL322x 系列的 MCU。



## 1.2 软件组织架构

在 Telink IoT Studio 中导入 tl\_ble\_sdk 后，显示的文件组织结构如下图所示（以 B91 为例）。顶层文件夹有 8 个：algorithm, application, boot, common, drivers, proj\_lib, stack, vendor。

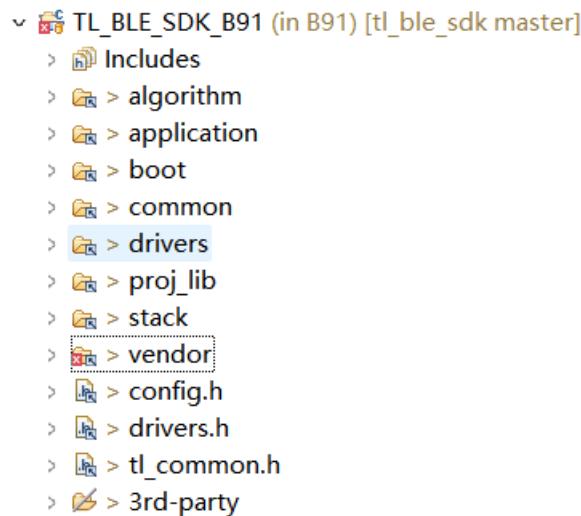


Figure 1.3: SDK 文件结构

**algorithm:** 提供一些通用的算法，如 aes\_ccm。大多数算法对应的 C 文件被封装在库文件中，只留对应的头文件。

**application:** 提供一些通用的应用处理程序，如 usb, keyboard 等。

**boot:** 提供 MCU 的 software bootloader 和链接脚本.link 文件。cstartup\_ 提供了 MCU 上电启动或 deep/deep retention sleep 唤醒后的汇编处理程序，及 main 函数的跳转，为后面 C 程序的运行做准备。boot\_general.link 定义了链接时不同段的地址映射。

**common:** 提供一些通用的跨平台定义，如 assert, BIT(x) 等。

**drivers:** 提供 MCU 外设驱动程序，如 Clock、Flash、I2C、USB、GPIO、UART 等，与 tl\_platform\_sdk 的 release 版本对应，可参考 doc/tl\_platform\_sdk\_Release\_Note.md。

**proj\_lib:** 存放 SDK 运行所必需的库文件。BLE 协议栈、RF 驱动、PM 驱动等文件，被封装在库文件中。不同的芯片对应使用不同的库文件。

**stack:** 存放 BLE 协议栈相关的头文件。源文件被编译到库文件里面，对于用户不可见。

**vendor:** 用于存放示例代码，或者用户的应用层代码。



### 1.2.1 vendor

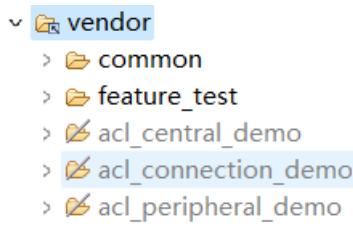


Figure 1.4: tl\_ble\_sdk 的 Demo 工程

tl\_ble\_sdk 提供了 3 个基础 demo：

- **acl\_peripheral\_demo**: 广播，自身作为 Peripheral 被其他 Central 连接的参考代码，默认配置为 COP4，使能 SMP，Deep Retention 休眠。
- **acl\_central\_demo**: 扫描，自身作为 Central 与其他 Peripheral 建立连接的参考代码，默认配置为 C4P0，使能 SMP，没有配置休眠。
- **acl\_connection\_demo**: 同时广播和扫描，自身可同时作为 Peripheral 或 Central 与其他设备连接的参考代码，默认配置为 C4P4，使能 SMP，Suspend 休眠。

feature\_test/ 下是各个功能的参考代码，使用方式可参考[Feature Demo](#) 章节。

common/ 下是应用层通用模块，如

- **boards/**: 该文件夹下是该 SDK 所支持适配的所有开发板通用配置。
- **battery\_check.c/.h**: 提供了低供电保护的处理方案，详情可参考[低电检测](#) 章节。
- **ble\_flash.h**: 声明了 Flash 相关接口、以及定义，如 Flash Map。
- **blt\_soft\_timer.c/.h**: 提供了软件定时器的实现方案。
- **default\_config.h**: 对应用层配置宏做了默认的定义，用户可以在 app\_config.h 中做定义，覆盖默认定义。
- **device\_manage.c/.h**: 连接设备信息的管理（例如：connection handle, attribute handle, BLE device address, address type 等）。
- **simple\_sdp.c/.h**: 该文件提供了 Central role 简单的 SDP (Service Discovery Protocol) 实现方案。
- **tlkapi\_debug.c/.h**: 提供了 Debug 日志的接口实现。

下面以 acl\_connection\_demo 为例来讲解 demo 文件结构，文件构成如下图所示：

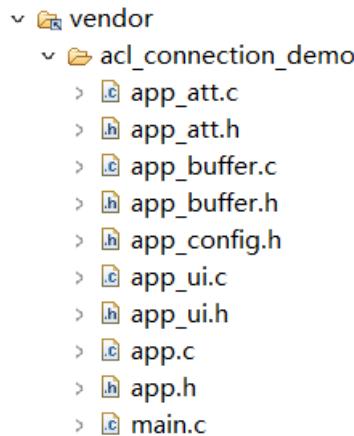


Figure 1.5: acl\_connection\_demo 文件结构

### 1.2.1.1 main.c

main.c 文件中包含 main 函数和中断处理函数。

main 函数是程序执行的入口，包含了系统正常工作所需的配置，建议用户不要对它进行任何修改。其中 tl\_ble\_sdk 支持主频 (CCLK) 最低为 32M。

中断处理函数是系统触发中断时候的入口函数。

### 1.2.1.2 app\_config.h

这是用户配置文件，用于对整个系统的相关参数（例如：BLE 参数，GPIO 配置，低功耗使能/失能，加密使能/失能等）进行配置。

下文介绍各个模块时会对 app\_config.h 中的各个参数的含义进行详细说明。

### 1.2.1.3 application files

**app.c/.h:** 用户主文件，用于完成 BLE 系统初始化、数据处理、低功耗处理等。

**app\_att.c/.h:** 提供了 GATT service 表和 profile 文件，GATT service 表中已提供了标准的 GATT 服务、标准的 GAP 服务、标准的 HID 服务以及一些私有服务等。用户可以参考这些添加自己的 service 和 profile。

**app\_ui.c/.h:** 主要提供按键处理接口和逻辑。

**app\_buffer.c/.h:** 该文件用于定义 stack 各层使用的 buffer，例如：LinkLayer TX & RX buffer、L2CAP layer MTU TX & RX buffer、HCI TX & RX buffer 等。

**app\_freertos.c/.h:** 当使能 FREERTOS\_ENABLE 宏时，将会调用该文件内的接口，运行 FreeRTOS 操作系统。

### 1.2.1.4 BLE stack entry

BLE 中断处理入口函数是 blc\_sdk\_irq\_handler()。

BLE 逻辑和数据处理入口函数是 blc\_sdk\_main\_loop ()，它负责处理 BLE 协议栈相关的数据处理和事件上报。



## 1.3 版本号

在旧版本的 tl\_ble\_sdk 中，使用 sdk\_version.txt 对版本号进行记录。该方法已经废除。

用户可以通过以下函数来获取当前 SDK 版本信息，该 API 在 V4.0.4.4 版本中功能进行了调整。

```
unsigned char blc_get_sdk_version(unsigned char *pbuf,unsigned char number);
```

参数 pbuf 是指向存储版本信息数组的指针，参数 number 是该数组的长度，返回值为版本信息字符串实际需要的长度。如果返回值为 0，代表传入的数组长度不够，需要扩大。

使用时，用户需先定义一个大数组，用以获取版本信息，将该数组及其大小作为参数传入函数，之后进行返回值和字符串的打印，参考代码如下：

```
u8 sdk_ver[180] = {0};  
u8 sdk_ver_len = blc_get_sdk_version(sdk_ver, 180);  
tlkapi_printf(1, "Version Info[%d]:%s\n", sdk_ver_len, sdk_ver);
```

运行后，打印内容类似 "Version Info[95]:V4.0.4.4\_P0001 C0.0 Develop 105b2c862 Thu Jun 19 21:09:30 2025 +0800 Dirty 2025-06-19 23:29:58"，则实际版本信息所需的字符串数组大小为 95 Bytes，代码可以修改为：

```
u8 sdk_ver[95] = {0};  
u8 sdk_ver_len = blc_get_sdk_version(sdk_ver, 95);  
tlkapi_printf(1, "Version Info[%d]:%s\n", sdk_ver_len, sdk_ver);
```



## 2 MCU 基础模块

### 2.1 MCU 地址空间

#### 2.1.1 Flash

各款 MCU 所支持的 Flash 大小，请参考对应的[Datasheet](#)。

程序会烧录在 Flash 中从 0x0 开始的位置。

Flash 读、写的基本单位为 byte，用户通过调用 `flash_write_page()/flash_read_page()`，实现读写操作。

**说明：**受 Flash 设计的限制，Flash 读、写驱动层面的基本单位为 page(256 bytes)，所以这里的接口名称为 `xx_page`。

Flash 擦除的基本单位为 sector (4K bytes)，通过调用 `flash_erase_sector()` 来实现一整个 sector 的擦除。

**注意：**写之前必须要先擦除，Flash 里面一个 page 为 256 byte，`flash_write_page` 函数支持跨 page 的写操作。

`tl_ble_sdk` 在初始化时通过调用 `blc_readFlashSize_autoConfigCustomFlashSector()` 读取 Flash 的 MID，得到 Flash 大小，进行底层 Flash Map 的自动配置，可参考下图及 `vendor/common/ble_flash.h` 的定义（注意替换下文中的 [Flash Size]）：

1M Flash only for one core MCU			2M Flash			4M Flash			16M Flash		
one core		MAC	one core		MAC	one core		MAC	one core		MAC
0xFF000	MAC		0x1FF00	MAC		0x3FF000	MAC		0xFFFF00	MAC	
0xFE000	Calibration		0x1FE00	Calibration		0x3FE000	Calibration		0xFE000	Calibration	
0xFD000	Reserved for future		0x1FD00	Reserved for future	0	0x3FD000	Reserved for future		0xFFD00	Reserved for future	0
0xFC000			0x1FC00			0x3FC000			0xFFC00		
0xFB000			0x1FB00			0x3FB000			0xFFB00		
0xFA000	Secure Boot Descriptor (16K)		0x1FA00	Secure Boot Descriptor (16K)		0x3FA000	Secure Boot Descriptor (16K)		0xFFA00	Secure Boot Descriptor (16K)	
0xF9000			0x1F900			0x3F9000			0xFF900		
0xF8000			0x1F800	0		0x3F8000			0xFF800	0	
0xF7000	SMP pairing and key information area (16K)		0x1F700			0x3F7000			0xFF700		
0xF6000			0x1F600			0x3F6000			0xFF600		
0xF5000			0x1F500			0x3F5000			0xFF500		
0xF4000			0x1F400			0x3F4000			0xFF400		
0xF3000	Bonding Peripheral GATT service critical		0x1F300	SW align area		0x3F3000	SW align area		0xFF300	SW align area	
0xF2000			0x1F200			0x3F2000			0xFF200		
0x1F100			0x1F100			0x3F1000			0xFF100		
0x1F000			0x1F000			0x3F0000			0xFF000		
0x1EF00			0x1EF00	SMP pairing and key information area (16K)		0x3EF000	SMP pairing and key information area (16K)		0xFFE00	SMP pairing and key information area (16K)	
0x1EE00			0x1EE00			0x3EE000			0xFFE00		
0x1ED00			0x1ED00			0x3ED000			0xFFE00		
0x1EC00			0x1EC00			0x3EC000			0xFFE00		
0x1EB00			0x1EB00	Bonding Peripheral GATT service critical		0x3EB000	Bonding Peripheral GATT service critical		0xFFE00	Bonding Peripheral GATT service critical	
0x1EA00			0x1EA00			0x3EA000			0xFFE00		

Figure 2.1: Flash Map

- `CFG_ADR_MAC_[Flash Size]_FLASH`: BLE MAC 地址保存的位置，使用方式参考 `blc_initMacAddress()`。
  - 对于 B91，如果 Flash 中这段内容为全 0xFF，会随机生成一段以 Telink Company ID 开头的 MAC 地址。
  - 对于 B92、TL321x、TL721x、TL322x，如果 Flash 中这段内容为全 0xFF，会从 Efuse/OTP 中对应位置读取内置的 MAC 地址。
- `CFG_ADR_CALIBRATION_[Flash Size]_FLASH`: RF 校准参数保存起始位置，相关代码参考 `user_calib_freq_offset()`。



- FLASH\_ADR\_SMP\_PAIRING\_[Flash Size]\_FLASH: SMP 配对信息存储起始位置，占 16K，其存储结构参考 stack/ble/host/smp/smp\_storage.h 中的 smp\_param\_save\_t，细节参考下文SMP章节。
- FLASH\_SDG\_ATT\_ADDRESS\_[Flash Size]\_FLASH: 作为 Client，做了 simple SDP 后，将对端 Server 的 ATT 信息存储在 Flash 中的位置。

**注意：**以上定义的区域都是占用了整个 sector，不允许用户使用这些 sector 做其他用途。

另外，如果用户在 B92、TL321x 或 TL721x 上配置了 Secure Boot，一块固定的 Flash 区域将用于做 Secure Boot 的配置，见上图。用户如果不需要 Secure Boot，可以自定义这几个 sector 的存储空间。

## 2.1.2 SRAM

各款 MCU 所支持的 SRAM 大小，请参考对应的[Datasheet](#)。I-SRAM 可以放指令和数据，D-SRAM 只能放数据。

对于 B92、TL721x 系列 MCU 的不同型号会有 SRAM 资源的差异，需要用户针对性地根据实际的 I-SRAM 和 D-SRAM 配置 boot/[Chip Name]/cstartup\_[Chip Name].S 文件中对 IRAM 和 DRAM 大小的定义。以 B92 为例：

```
#define SRAM_SIZE      SRAM_128K

#if (SRAM_SIZE == SRAM_256K)
    .equ __IRAM_2_EN,   1
    .equ __DRAM_1_EN,   0
    .equ __DRAM_2_EN,   0
    .equ __DRAM_DIS,   1
#elif (SRAM_SIZE == SRAM_384K)
    .equ __IRAM_2_EN,   1
    .equ __DRAM_1_EN,   1
    .equ __DRAM_2_EN,   0
    .equ __DRAM_DIS,   0
#elif (SRAM_SIZE == SRAM_512K)
    .equ __IRAM_2_EN,   1
    .equ __DRAM_1_EN,   1
    .equ __DRAM_2_EN,   1
    .equ __DRAM_DIS,   0
#else
    .equ __IRAM_2_EN,   0
    .equ __DRAM_1_EN,   0
    .equ __DRAM_2_EN,   0
    .equ __DRAM_DIS,   1
#endif
```

需要注意，I-SRAM 和 D-SRAM 配置只有 cstartup\_[Chip Name].S 中列出的固定几种配置，并不是用户可以自由组合的。



### 2.1.3 MCU 地址空间分配

各款 MCU 的 Memory Map, 请参考对应的[Datasheet](#), 以 B91 的 TLSR9218A 为例, 在 Datasheet 的第[4.1.1 SRAM](#)章节, 介绍了它的 Memory Map。SDK 主要关注以下几个地址空间:

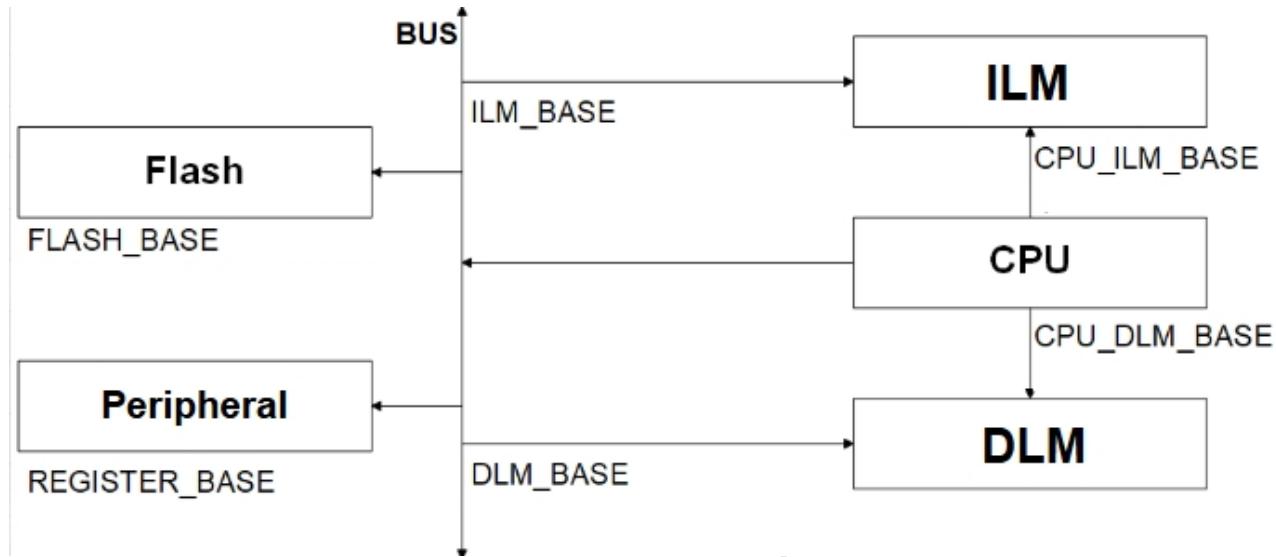


Figure 2.2: MCU 地址空间分配

- Flash 空间的地址范围 (FLASH): 0x20000000~0x21000000
- 寄存器的地址范围 (USB、AUDIO、ZB 等): 0x80100000~0x82000000
- CPU 访问 I-SRAM 的地址范围 (ILM\_CPU): 0x00000000~0x00020000
- CPU 访问 D-SRAM 的地址范围 (DLM\_CPU): 0x00080000~0x000A0000
- 总线访问 I-SRAM 的地址范围 (ILM): 0xC0000000~0xC0020000
- 总线访问 D-SRAM 的地址范围 (DLM): 0xC0200000~0xC0220000

**说明:** 从上图看到, I-SRAM 或 D-SRAM 可以通过 CPU 访问, 也可以通过总线 (如 DMA, Swire) 访问, 通过不同的访问地址来区分访问的方式。从上图可知, CPU 也可以通过总线访问。

### 2.1.4 Flash 和 SRAM 空间分配

tl\_ble\_sdk 会通过调用 `blc_app_setDeepsleepRetentionSramSize()` 根据实际 SRAM 要使用的 retention 大小, 来配置 deepsleep retention 的模式 (如 DEEPSLEEP\_MODE\_RET\_SRAM\_LOW32K)。下图是以 B91 为例, 在 deepsleep retention 32K 模式下, I-SRAM 和 D-SRAM 都同时使用的情况下对应的 SRAM 和 Firmware 空间分配说明。

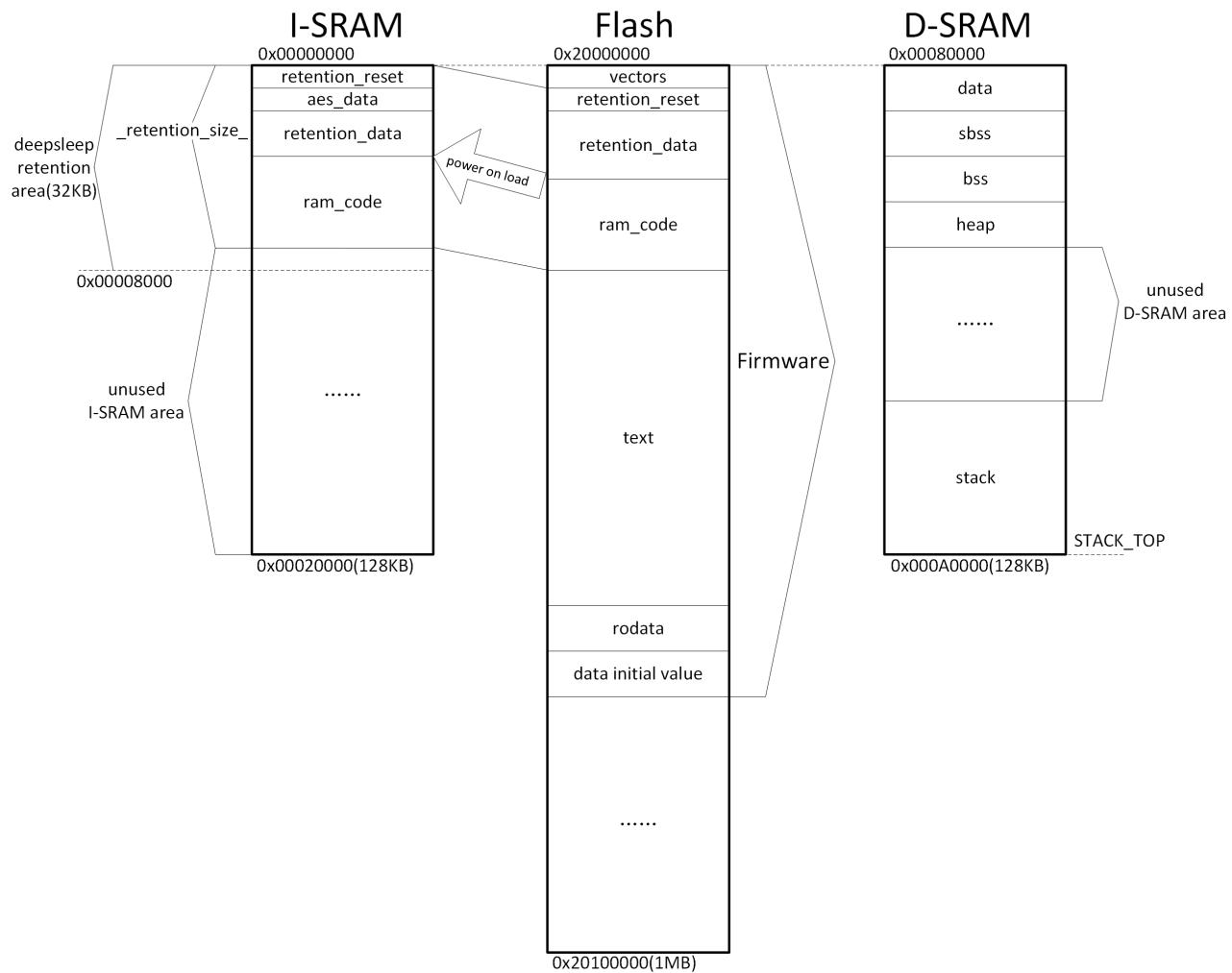


Figure 2.3: SRAM 空间分配 & Firmware 空间分配

上图的 SRAM 空间分配相关的文件有 cstartup\_B91.S 和 boot\_general.link。

编译生成的 Firmware 下载到 Flash 中，其内容包括 vectors、retention\_reset、retention\_data、ram\_code、text、rodata 和 data initial value。

上电启动后，会将 Flash 中的部分内容搬运到 I-SRAM 和 D-SRAM。

I-SRAM 中包括 retention\_reset、aes\_data、retention\_data、ram\_code、和 unused I-SRAM area。I-SRAM 中的 retention\_reset / retention\_data / ram\_code 是 Flash 中 retention\_reset / retention\_data / ram\_code 的拷贝。

D-SRAM 中包括 data、sdata、bss、sbss、heap、unused D-SRAM area 和 stack。D-SRAM 中 data 的初始值是 Flash 中 data initial value。

## 2.1.4.1 各个段的介绍

### 2.1.4.1.1 data、sdata 和 data initial value

“data” 或 “sdata” 段存放初值非 0 的全局变量。sdata 是 small data 的简写。其初值存储在 Flash Firmware 中



的“data initial value”段，在运行 bootloader 时，会将初值拷贝到变量对应的 SRAM 地址。

#### 2.14.1.2 sbss and bss

“sbss”段和“bss”段存放程序未初始化或初始化为 0 的全局变量，即 initial value 为 0 的全局变量。sbss 是 small bss 的简写。在运行 bootloader 时，会直接将该段 SRAM 全设为 0。

#### 2.14.1.3 text

“text”段是程序中的函数默认编译到的段。访问“text”段的指令需要通过 I-Cache，将需要执行的指令先加载到 I-Cache 中才可以被执行。

#### 2.14.1.4 rodata

“rodata”段存放的是程序中定义的可读、不能改写的变量，位于 Flash 中，如用关键字“const”定义的变量。

#### 2.14.1.5 ram\_code

“ram\_code”段是 Firmware 中需要常驻内存的 code，上电后 MCU 会从 Flash 拷贝到 I-SRAM 的 ram\_code area 上，内容包含 SDK 中加了关键字 `_attribute_ram_code_xxx` 的函数，比如 `rf_irq_handler` 函数：

```
_attribute_ram_code_ void rf_irq_handler(void);
```

函数常驻内存有三个原因：

- 某些函数由于涉及到和 Flash MSPI 四根管脚的时序复用，必须常驻内存，如果放到 Flash 中就会出现时序冲突，造成死机，如 Flash 操作相关所有函数；
- 放到 ram 中执行的函数每次被调用时不需要从 Flash 重新读取，可以节省时间，所以对于一些执行时间有要求的函数可以放到常驻内存，提高执行效率。SDK 中将 BLE 时序相关的一些经常要执行的函数常驻到内存，大大降低执行时间，节省功耗；

**注意：** tl\_ble\_sdk 支持中断嵌套的功能，详细内容可以先参考[中断嵌套](#)章节，如果用户新增了 LEV3 优先级的中断入口函数，就必须将它放到“ram\_code”段。

用户如果需要将某个函数常驻内存，可以仿照上面的 `rf_irq_handler`，在函数上添加关键字 `_attribute_ram_code_`，编译之后就能在 `objdump` 文件中看到该函数在 `ram_code` 段了。

#### 2.14.1.6 retention\_data

“retention\_data”段是 Firmware 中需要常驻内存的数据，上电或 normal deepsleep 唤醒后 MCU 会从 Flash 拷贝到 I-SRAM 的 retention\_data area 上，deepsleep retention 休眠后，I-SRAM 的指定 retention 区域数据不掉电保持（参考定义 `pm_sleep_mode_e`）。

程序中的全局变量默认会分配在“data”段、“sdata”段、“sbss”段或“bss”段，并未限定存放在 I-RAM 的 retention 区域，进入 deepsleep retention 会丢失。



如果希望一些特定的变量在 deepsleep retention mode 期间能够保持，需要将它们分配到“retention\_data”段，方法是在定义变量时添加关键字 `_attribute_data_retention_`。以下为几个示例：

```
_attribute_data_retention_ int AA;
_attribute_data_retention_ unsigned int BB = 0x05;
_attribute_data_retention_ int CC[4];
_attribute_data_retention_ unsigned int DD[4] = {0,1,2,3};
```

#### 2.14.1.7 vectors and retention\_reset

“vectors”段和“retention\_reset”段都是汇编文件 cstartup\_.S 对应的程序，是软件启动代码（software boot-loader）。其中 vectors 段是在 Flash 起始地址，retention\_reset 段是在 I-SRAM 的起始地址。芯片上电或 deepsleep 醒来后从 Flash 的起始地址开始执行，deepsleep retention 醒来从 I-SRAM 的起始地址开始执行。

#### 2.14.1.8 aes\_data

“aes\_data”段存放硬件 AES 模块的缓存数据，aes\_data 段在 I-SRAM 上，并且长度固定为 32Bytes，用户不可更改。在运行 bootloader 时，会将这段区域全设为 0。

#### 2.14.1.9 retention\_size

I-SRAM 的“retention\_reset + aes\_data + retention\_data + ram\_code”共 4 段按顺序排布在 I-SRAM 的前面，它们的总大小为“\_retention\_size\_”。MCU 上电或 normal deepsleep 醒来后，程序在执行过程中只要不进 normal deepsleep（只有 suspend/deepsleep retention），“\_retention\_size\_”的内容就一直保持在 I-SRAM 上，MCU 无须再从 Flash 上读取。

评估“\_retention\_size\_”的方法是根据 objdump 文件开头的 ‘Sections’，使用“ram\_code”段的 ‘Size’ 与 ‘VMA’ 相加，就是实际的“\_retention\_size\_”大小，比如下图的“\_retention\_size\_”大小为 0x5b02 + 0xf00，约 26.5KB。

24 Sections:						
Idx	Name	Size	VMA	LMA	File off	Align
0	.vectors	00000166	20000000	20000000	00001000	2**2
		CONTENTS, ALLOC, LOAD, READONLY, CODE				
1	.retention_reset	00000126	00000000	20000168	00002000	2**2
		CONTENTS, ALLOC, LOAD, READONLY, CODE				
2	.aes_data	00000020	00000128	20000290	00002126	2**2
		ALLOC				
3	.retention_data	00000da0	00000148	20000290	00002148	2**2
		CONTENTS, ALLOC, LOAD, DATA				
4	.ram_code	00005b02	00000f00	20001030	00002f00	2**8
		CONTENTS, ALLOC, LOAD, READONLY, CODE				

Figure 2.4: \_retention\_size\_ 大小



如果用户不希望浪费太多的“\_retention\_size\_”，可以将之前不在 retention\_data/ram\_code 的 variable (变量) /function (函数) 通过添加对应的关键字，适当增加切换到 retention\_data/ram\_code 中。放在 retention\_data 中的 variable 也可以节省初始化时间以降低功耗。

如果用户选择的配置使用的是 deepsleep retention 32K mode，但定义的“\_retention\_size\_”超过所定义的 32K，比如下图的“\_retention\_size\_”大小为 0x5b02 + 0x3700，约 36.5KB，超出 32K 部分的 ram\_code 会因为进入 deepsleep retention 模式而丢失内容。

24 Sections:						
25	Idx	Name	Size	VMA	LMA	File off Algn
26	0	.vectors	00000166	20000000	20000000	00001000 2**2
27			CONTENTS, ALLOC, LOAD, READONLY, CODE			
28	1	.retention_reset	00000126	00000000	20000168	00002000 2**2
29			CONTENTS, ALLOC, LOAD, READONLY, CODE			
30	2	.aes_data	00000020	00000128	20000290	00002126 2**2
31			ALLOC			
32	3	.retention_data	000035a0	00000148	20000290	00002148 2**2
33			CONTENTS, ALLOC, LOAD, DATA			
34	4	.ram_code	00005b02	00003700	20003830	00005700 2**8
35			CONTENTS, ALLOC, LOAD, READONLY, CODE			

Figure 2.5: \_retention\_size\_ 大小

用户可以通过下面的方式修改：

1. 减少所定义“\_attribute\_data\_retention\_”段或“\_attribute\_ram\_code\_”段属性的内容。
2. 根据芯片支持情况，选择切换为 deepsleep retention 更大 Size 的模式。

#### 2.14.1.10 Cache

Cache 为高速缓存，分为 I-Cache 和 D-Cache 两块，大小各是固定的 8KB，访问地址用户不可见。

默认 Cache 是打开的，在 cstartup\_.S 文件里配置：

```
/* Enable I/D-Cache */
csrr t0, mcache_ctl
ori t0, t0, 1 #/I-Cache
ori t0, t0, 2 #/D-Cache
csrw mcache_ctl, t0
fence.i
```

常驻内存的 code 可以直接从 SRAM 中读取并执行，但 Firmware 中可以常驻 SRAM 的 code 只是一部分，剩下绝大部分都还在 Flash 中。根据程序的局部性原理，可以将一部分 Flash code 加载到 Cache 中，如果当前需要执行的 code 在 Cache 里，直接从 Cache 读取并执行；如果不在 Cache 中，则从 Flash 读取 code 加载到 Cache 再执行。

Firmware 的“text”和“rodata”段是没有放到 SRAM 中，这部分 code 符合程序局部性原理，需要一直被 load 到 Cache 中才能被执行。



由于 Cache 比较大，所以不允许用户使用指针形式读 Flash，因为指针形式读 Flash 的数据被缓存在 Cache 里，如果 Cache 里这个数据没有被其他内容覆盖时，即使该位置处的 Flash 数据已经被改写，又有新的访问该数据的请求发生，此时 MCU 会直接用 Cache 里缓存的内容作为结果。

#### 2.14.1.11 heap

“heap”区域是分配给堆，堆是向上增长的，一般我们设置在 bss 后面不用的空间，如果调用了 sprintf/malloc/free 这类内存管理函数，这些函数会调用 \_sbrk 函数进行堆内存的分配，\_sbrk 会通过 \_end 符号确定从哪里开始分配堆空间，在 link 文件的定义如下。

```
PROVIDE (_BSS_VMA_END = .);
...
. = ALIGN(8);
/* end is the starting address of the heap, the heap grows upward */
_end = .;
PROVIDE (end = .);
```

#### 2.14.1.12 stack

对于 128K 的 D-SRAM，“stack”是从最高地址 0x000A0000 开始的，其方向为从下往上延伸，即 stack 指针 SP 在数据入栈时自减，数据出栈时自加。

如果 stack 使用过大，那么可能会出现栈溢出的情况，与.bss 段发生重合，导致程序运行错误。关于查看 stack 的原理和方法见附录 2。

以 B91 为例，boot\_general.link 文件中定义了栈顶位置 \_STACK\_TOP：

```
PROVIDE (_STACK_TOP = 0x00a0000); /*Need to prevent stack overflow*/
```

在 cstartup\_B91.S 文件里初始化了堆栈指针 sp 寄存器：

```
/* Initialize stack pointer */
la    t0, _STACK_TOP
mv    sp, t0
```

如果用户希望 128K 的 D-SRAM 空间全部留给用户使用或者 D-SRAM 不使用，可以将 SDK 占用的数据和指令都放到 I-SRAM。

## 2.2 时钟模块

时钟模块参考各 MCU 的[Datasheet](#)中的 Clock 章节。



初始化时调用 API `blc_app_system_init()` 配置 `pll_clk/cclk/hclk/pclk/clk_msp`, 以 TL321x 为例:

```

Project Explorer < main.c >
131 * @return ... none.
132 */
133 INLINE void blc_app_system_init(void)
134 {
135 #if (MCU_CORE_TYPE == MCU_CORE_B91)
136 #elif (MCU_CORE_TYPE == MCU_CORE_B92)
137 #elif (MCU_CORE_TYPE == MCU_CORE_TL721X)
138 #elif (MCU_CORE_TYPE == MCU_CORE_TL321X)
139     sys_init(DCDC_1P25_LDO_1P8, VBAT_MAX_VALUE_GREATER_THAN_3V6, INTERNAL_CAP_XTAL24M);
140     pm_update_status_info(1);
141     gpio_set_up_down_res(GPIO_SWS, GPIO_PIN_PULLUP_1M);
142     wd_32k_stop();
143     wd_stop();
144     PLL_192M_CCLK_24M_HCLK_12M_PCLK_12M_MSP_48M;
145 #elif (MCU_CORE_TYPE == MCU_CORE_TL751X)
146 }
147 */
148 /**
149 * @brief ... This is main function
150 */

```

**注意:** 多连接 SDK 的 CCLK 至少配置 24M 系统时钟, 其他时钟无法满足多连接 SDK 的运行。

B91 的 System Timer 的频率是固定的 16MHz, B92、TL321x、TL721x、TL322x 的 System Timer 的频率是固定的 24MHz。由于 System Timer 是 BLE 计时的基准, SDK 中所有 BLE 时间相关的参数和变量, 在涉及到时间的表达时, 都是用“`SYSTEM_TIMER_TICK_xxx`”的方式, 如下的数值来表示 s、ms 和 us (以 TL321x 为例):

```

enum
{
    SYSTEM_TIMER_TICK_1US = 24,
    SYSTEM_TIMER_TICK_1MS = 24000,
    SYSTEM_TIMER_TICK_1S = 24000000,

    SYSTEM_TIMER_TICK_625US = 15000, //625*24
    SYSTEM_TIMER_TICK_1250US = 30000, //1250*24
};

```

SDK 中以下几个 API 都是跟 System Timer 相关的一些操作, 这些 API 内部已经使用上面类似 “`xxx_TIMER_TICK_xxx`” 的方式来表示时间, 用户操作这些 API 时, 根据形参提示输入 us 或 ms 就可以。

```

void delay_us(unsigned int microsec);
void delay_ms(unsigned int millisec);
_Bool clock_time_exceed(unsigned int ref, unsigned int us)

```

## 2.2.1 System Timer 的使用

main 函数中 `sys_init` 初始化完成后, System Timer 就开始工作, 用户可以读取 System Timer 计数器的值 (简称 System Timer tick)。

System Timer tick 每一个时钟周期加一, 其长度为 32bit, 最小值 0x00000000, 最大值 0xffffffff。System Timer 刚启动的时候, tick 值为 0。B91 每 1/16us 加 1, 到最大值 0xffffffff 需要的时间为:  $(1/16) \text{ us} * (2^{32})$  约等于 268 秒, 每过 268 秒 System Timer tick 转一圈。B92 每 1/24us 加 1, 到最大值 0xffffffff 需要的时间为:  $(1/24) \text{ us} * (2^{32})$  约等于 178 秒, 每过 178 秒 System Timer tick 转一圈。

MCU 在运行程序过程中 system tick 不会停止。

System Timer tick 的读取可以通过 `clock_time()` 函数获得:



```
u32 current_tick = clock_time();
```

## 2.3 中断嵌套

### 2.3.1 中断嵌套功能简述

tl\_ble\_sdk 包含的芯片支持中断嵌套功能，先说明下三个概念：中断优先级，中断阈值，中断抢占。

- (1) 中断优先级是每个中断的等级，在初始化中断的时候需要配置；
- (2) 中断阈值是指响应中断的阈值，只有中断优先级高于中断阈值的中断才会被触发；
- (3) 中断抢占是指当两个中断的优先级都高于中断阈值，如果当前较低优先级的中断正在被响应，较高优先级的中断可以被触发，抢占较低优先级的中断，执行完较高优先级的中断后再继续执行较低优先级的中断。

**注意：**

中断嵌套功能默认是打开的，并且中断阈值默认为 0。

中断优先级可以设置的范围 1~3，中断优先级目前只能支持最高设置到 3，数字越大优先级越高，优先级的枚举如下：

```
typedef enum{
    IRQ_PRI_LEV0,//Never interrupt
    IRQ_PRILEV1,
    IRQ_PRILEV2,
    IRQ_PRILEV3,
}irq_priority_e;
```

如下图所示，BLE SDK 已经规划了 3 种类型优先级的中断，用户必须按照这样的分类来使用。中断优先级 LEV1 的中断等级最低，分配给用户定义的 APP 普通中断。中断优先级 LEV2 的中断等级在中间，强制分配给 BLE 中断，用户定义的中断不能使用 LEV2。中断优先级 LEV3 的中断等级最高，一般情况下不建议使用，只有某些特殊场合需要实时响应的情况才使用，也是分配给用户定义的 APP 高级中断。



LEV3

APP Advanced Interrupt

LEV2

BLE Interrupt("rf\_irq" and  
"stimer\_irq")

LEV1

APP Normal Interrupt

## 0 ----- Interrupt Threshold

BLE SDK 在初始化的 `blc_ll_initBasicMCU` 里已经将 BLE 中断（"rf\_irq" 和 "stimer\_irq"）的中断优先级设置为 `IRQ_PRI_LEVEL2`，并且中断阈值设置为 0（LEV1~LEV3 优先级的中断都可以被触发）。

用户定义的 APP 普通中断，需要将中断优先级设置为 `IRQ_PRI_LEVEL1`，不用限制执行时间，BLE 中断和 APP 高级中断会抢占 APP 普通中断。

如果用户有 APP 高级中断的需求，需要将中断优先级设置为 `IRQ_PRI_LEVEL3`。在使用 APP 高级中断时需注意：

- 中断处理函数必须放到 `ram_code` 段
- 中断处理函数中不允许访问 Flash
- 中断处理函数执行时间小于 50us

在执行 Flash 空间的擦除、读写操作函数时，会将中断阈值设置为 1，执行完 Flash 操作函数后再将中断阈值设置为 0，因此在读写 Flash 操作过程中允许 BLE 中断和用户 APP 高级中断插入。如果 BLE 中断和用户 APP 高级中断函数存放在 Flash 中，Flash 预取指操作和读写 Flash 操作会出现时序冲突，造成死机。如果 BLE 中断和用户 APP 高级中断函数中存在读写 Flash 的操作，多个读写 Flash 操作也会出现时序冲突，造成死机。因此需将 BLE 中断和用户 APP 高级中断函数放在 `ram_code` 段，以及函数内禁止访问 Flash。由于用户 APP 高级中断会抢占 BLE 中断和 APP 普通中断，所以用户也必须限制高级中断函数执行时间小于 50us 以免影响到 BLE 中断。



### 2.3.2 中断嵌套的使用

#### 2.3.2.1 App 普通中断

比如用户想设置一个 PWM 的 APP 普通中断，在配置中断的时候定义中断优先级为 IRQ\_PRI\_LEVEL1，方法如下。

```
plic_set_priority(IRQ16_PWM, IRQ_PRI_LEVEL1);
```

中断响应函数类型不限。

```
void pwm_irq_handler(void)
{
    .....
}
```

#### 2.3.2.2 App 高级中断

比如用户想设置一个 Timer0 的 APP 高级中断，在配置中断的时候定义中断优先级为 IRQ\_PRI\_LEVEL3，方法如下。

```
plic_set_priority(IRQ4_TIMER0, IRQ_PRI_LEVEL3);
```

中断响应函数必须定义为 ram\_code 段，方法如下。

```
_attribute_ram_code_ void timer0_irq_handler(void)
{
    .....
}
```

### 2.3.3 中断使用限制

BLE 中断要求及时响应，所以无论是优先级为 IRQ\_PRI\_LEVEL3 的 APP 中断，还是用户关闭全局中断，都限制最长时间为 50us，需要用户特别注意。



## 3 BLE 模块

本手册以 Bluetooth Core Specification 6.0 版本为参考。

### 3.1 BLE SDK 软件架构

#### 3.1.1 标准 BLE SDK 软件架构

根据 Bluetooth Core Specification，一个比较标准的 BLE SDK 架构如下图所示。

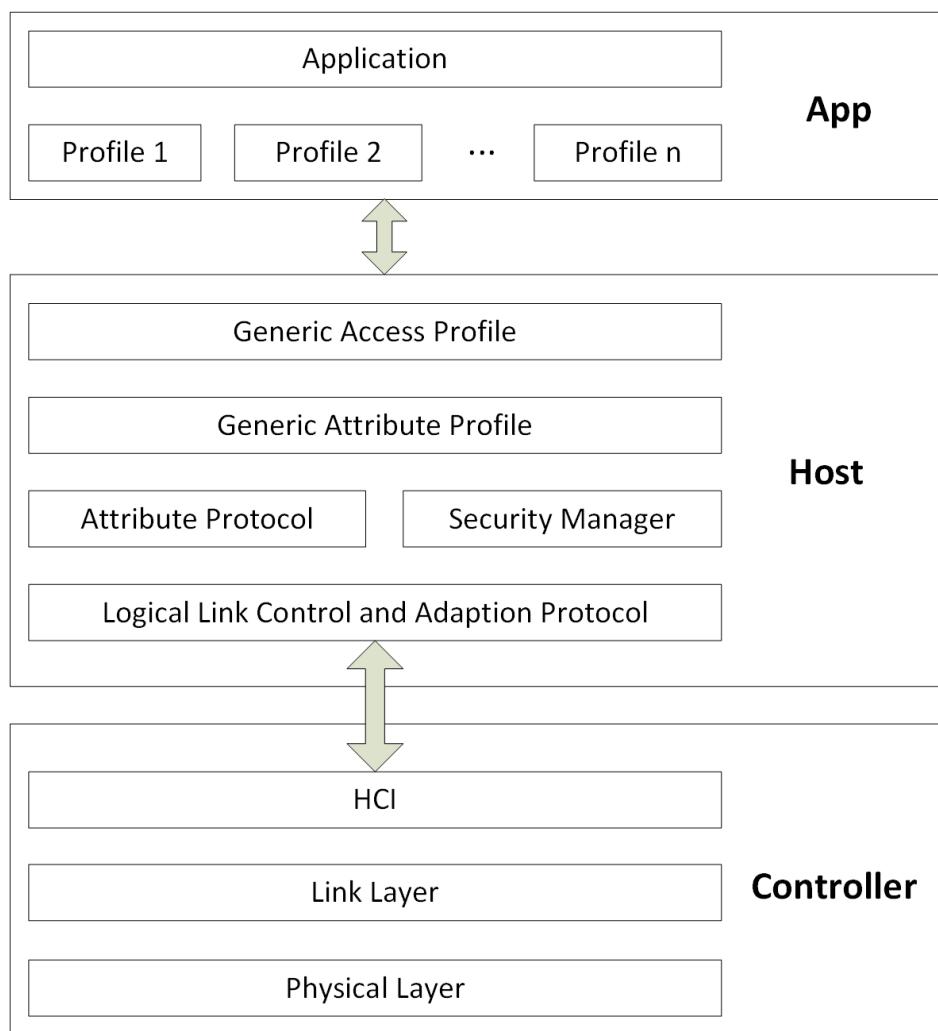


Figure 3.1: BLE SDK 软件架构

在上图所示的架构中，BLE 协议栈分为 Host 和 Controller 两部分。

- Controller 作为 BLE 底层协议，包括 Physical Layer (PHY) 和 Link Layer (LL)。Host Controller Interface (HCI) 是 Controller 与 Host 的唯一通信接口，Controller 与 Host 所有的数据交互都通过该接口完成。



- Host 作为 BLE 上层协议，协议上有 Logic Link Control and Adaption Protocol (L2CAP)、Attribute Protocol (ATT)、Security Manager Protocol (SMP)，Profile 包括 Generic Access Profile (GAP)、Generic Attribute Profile (GATT)。
- 应用层（APP）包含 user 自己相关应用代码和各种 service 对应的 Profile，user 通过 GAP 去控制访问 Host。Host 通过 HCI 与 Controller 完成数据交互，如下图所示。

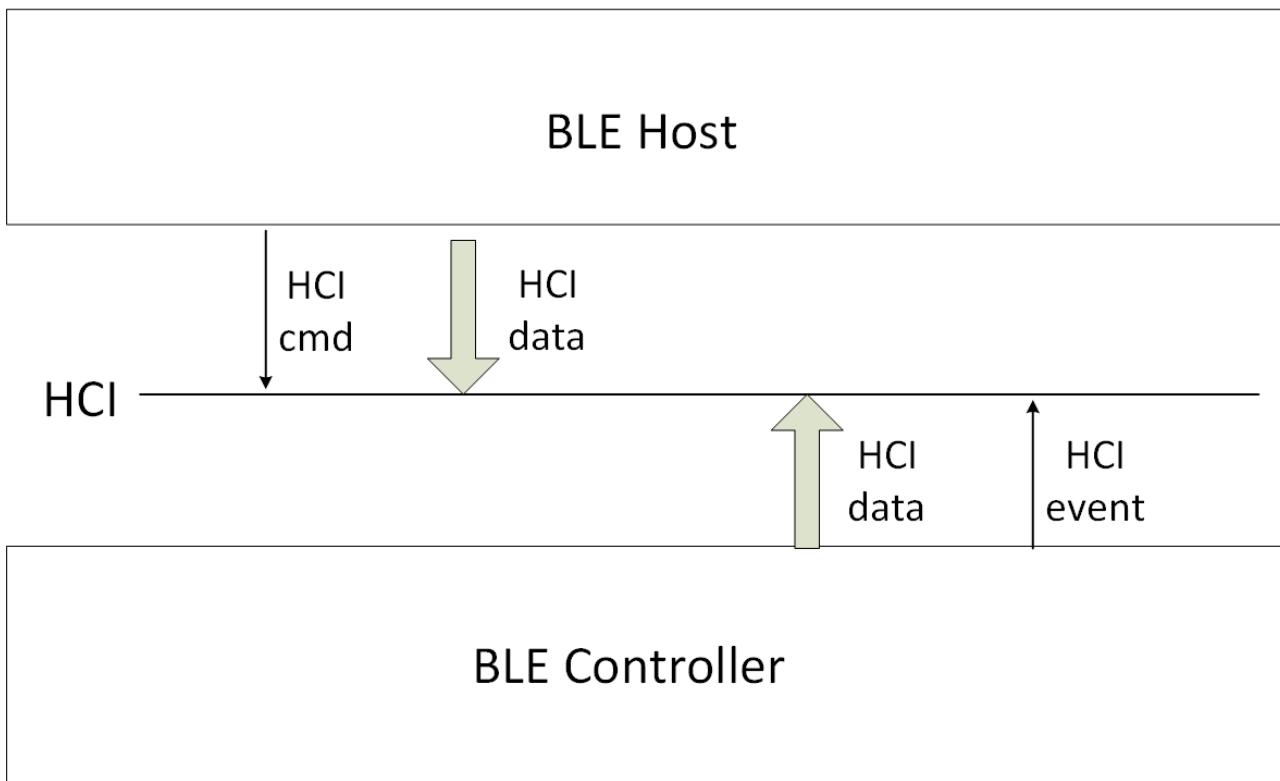


Figure 3.2: Host 和 Controller 的 HCI 数据交互

- (1) BLE Host 通过 HCI cmd 去操作设置 Controller。
- (2) Controller 通过 HCI 向 Host 上报各种 HCI event。
- (3) Host 将需要发送给对方设备的数据通过 HCI 传递到 Controller，Controller 将数据直接丢到 Physical Layer 进行发送。
- (4) Controller 在 Physical Layer 收到的 RF 数据，先判断是属于 Link Layer 的数据还是 Host 的数据：如果是 Link Layer 的数据，直接处理数据；如果是 Host 的数据，则通过 HCI 将数据传到 Host。

### 3.1.2 Telink BLE SDK 软件架构

#### 3.1.2.1 Telink BLE Multiple Connection Controller

tl\_ble\_sdk 支持标准的 BLE controller，包括 HCI、PHY (Physical Layer) 和 LL (Link Layer)。

tl\_ble\_sdk 包含 Link Layer 的五种标准状态(standby、advertising、scanning、initiating、connection)，connection 状态下同时支持最多 4 个 Central role 和 4 个 Peripheral role。

controller 架构图如下：

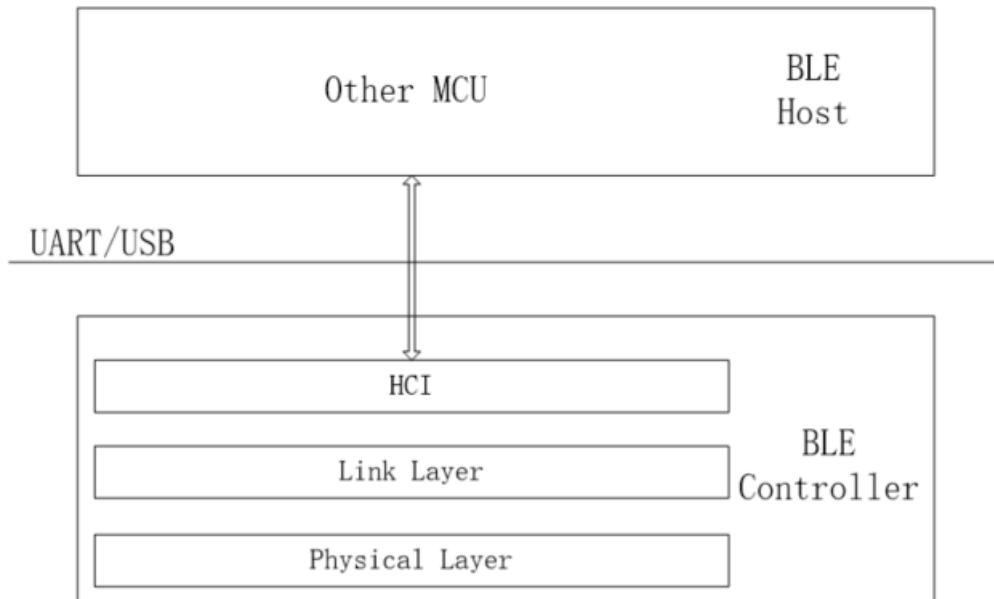


Figure 3.3: Telink HCI 架构

### 3.1.2.2 Telink BLE Multiple Connection Whole Stack (Controller+Host)

tl\_ble\_sdk 提供 BLE Multiple Connection Whole Stack(Controller + Host) 参考设计，只有对于 Central SDP (service discovery) 无法做到完全支持，后面的章节会具体介绍。

Telink BLE stack 架构会对上面标准的结构做一些简化处理，使得整个 SDK 的系统资源开销（包括 Sram、运行时间、功耗等）最小，其架构如下图所示。SDK 中提供的 demo 都是基于该架构。

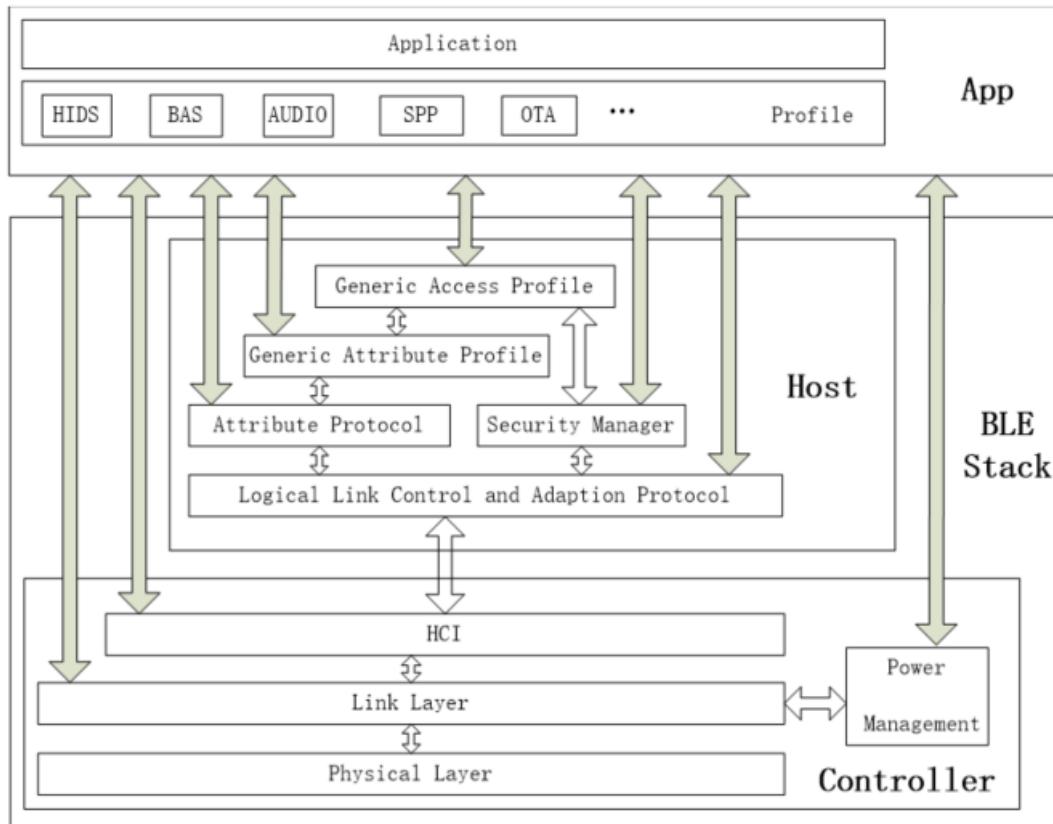


Figure 3.4: tl\_ble\_sdk Whole Stack 架构

图中实心箭头所示的数据交互是 user 可以通过各种接口来操作控制的，会提供 user API。空心箭头是协议栈内部完成的数据交互，user 无法参与。

HCI 是 Controller 与 Host 的数据通信接口（和 L2CAP 层对接），但不是唯一的接口，APP 应用层也可以直接与 Link Layer 进行数据交互。Power Management (PM) 低功耗管理单元被内嵌到 Link layer，应用层可以调用 PM 相关接口进行功耗管理的设置。

考虑到效率，应用层与 Host 的数据交互不通过 GAP 来访问控制，协议栈在 ATT、SMP 和 L2CAP 都提供了相关接口，可以和应用层直接交互。但是 Host 所有 Event 需要通过 GAP 层和应用层交互。

Host 层以 Attribute Protocol 为基础，实现了 Generic Attribute Profile (GATT)。应用层基于 GATT，定义 user 自己需要的各种 profile 和 service。该 BLE SDK 提供几个基本的 profile，包括 HIDS、BAS、OTA 等。

下面基于这个架构对 BLE 多连接协议栈各部分做一些基本的介绍，并给出各层的 user API。

其中 Physical Layer 完全由 Link Layer 控制，且不需要应用层任何的参与，这部分不介绍。

虽然 Host 与 Controller 的部分数据交互还是靠 HCI 来完成，但基本都是 Host 和 Controller 协议栈完成，应用层几乎不参与，只需要在 L2CAP 层注册 HCI 数据回调处理函数就行了，所以对 HCI 部分也不做介绍。



## 3.2 Controller

### 3.2.1 Connection Number 配置

#### 3.2.1.1 supportedMaxCentralNum & supportedMaxPeripheralNum

tl\_ble\_sdk 将 Connection Central role 最大数量称为 supportedMaxCentralNum，将 Connection Peripheral role 最大数量称为 supportedMaxPeripheralNum，他们是由 library 决定的，如下表所示：

Table 3.1: 支持最大主从数与 library 对应关系

IC	library	supportedMaxCentral-Num	supportedMaxPeripheralNum
B91	liblt_9518	4	4
B92	liblt_9528	4	4
TL721X	liblt_TL721X	4	4
TL321X	liblt_TL321X	4	4

SDK 可以通过下面的 API 查询当前 Stack 支持的 Central 和 Peripheral 数量。

```
int blc_ll_getSupportedMaxConnNumber(void);
```

#### 3.2.1.2 appMaxCentralNum & appMaxPeripheralNum

在 supportedMaxCentralNum 和 supportedMaxPeripheralNum 已经确定的前提下，用户可以通过下面 API 来设置自己应用上想要的最大 Central 和 Peripheral 数量，分别称为 appMaxCentralNum 和 appMaxPeripheralNum。

```
ble_sts_t blc_ll_setMaxConnectionNumber(int max_master_num, int max_slave_num);
```

这个 API 只允许在初始化的时候调用，即 Link Layer 运行之前就需要确定好相关连接数，不允许后面再修改。

用户的 appMaxCentralNum 和 appMaxPeripheralNum 必须小于或等于 supportedMaxCentralNum 和 supportedMaxPeripheralNum。

参考例程设计上在初始化的时候都使用了该 API：

```
blc_ll_setMaxConnectionNumber(ACL_CENTRAL_MAX_NUM, ACL_PERIPH_MAX_NUM);
```

用户需要在 app\_config.h 中定义自己的 appMaxCentralNum 和 appMaxPeripheralNum，即 SDK 中的 ACL\_CENTRAL\_MAX\_NUM 和 ACL\_PERIPH\_MAX\_NUM。



#define ACL_CENTRAL_MAX_NUM	4
#define ACL_PERIPH_MAX_NUM	4

appMaxCentralNum 和 appMaxPeripheralNum 能够节省 MCU 的各种资源，比如针对 C4P4 的 library，用户如果只需要用到 C3P2，将 ACL\_CENTRAL\_MAX\_NUM 和 ACL\_PERIPH\_MAX\_NUM 分别设为 3 和 2 后：

#### (1) 节省 SRAM 资源

Link Layer TX Central FIFO 和 TX Peripheral FIFO、L2CAP Central MTU buffer 和 L2CAP Peripheral MTU buffer 都是根据 appMaxCentralNum 和 appMaxPeripheralNum 来分配的，所以可以节省一些 Sram 资源。具体请参考文档后面 TX FIFO 相关的介绍。

#### (2) 节省时间资源和功耗

对于 C4P4，Stack 必须等到 currentCentralNum 为 4 时才会停止 Scan 动作，必须等到 currentPeripheralNum 为 4 时才会停止 Advertising 动作。而对于 C3P2，Stack 等到 currentCentralNum 为 3 时才会停止 Scan 动作，currentPeripheralNum 为 2 时就会停止 Advertising 动作，这样就少了不必要的 Scan 和 Advertising，能够节省 PHY 层带宽，也能降低 MCU 功耗。

### 3.2.1.3 currentMaxCentralNum & currentMaxPeripheralNum

用户定义了 appMaxCentralNum 和 appMaxPeripheralNum 后，确定了 Link Layer 运行时创建的 Central 和 Peripheral 最大数量。但 Central 和 Peripheral 在某一时刻的数量还是不确定的，比如 appMaxCentralNum 为 4 时，任何时刻 Central 的数量可能是 0,1,2,3,4。

SDK 提供了以下 3 个 API，供用户实时查询当前 Link Layer 上的 Central 和 Peripheral 数量。

```
int blc_ll_getCurrentConnectionNumber(void); //Central + Peripheral connection number  
int blc_llGetCurrentCentralRoleNumber(void); //Central role number  
int blc_llGetCurrentPeripheralRoleNumber(void); //Peripheral role number
```

### 3.2.2 Link Layer 状态机

用户可以先参考 Telink B91 BLE Single Connection SDK 中 Link Layer 状态机的介绍，Link Layer 5 个基本状态都是支持的，将 Connection state 再分为 Connection Peripheral role 和 Connection Central role 的话，Link Layer 在任意时刻一定是且只能是以下 6 个状态中的 1 个：Standby、Advertising、Scanning、Initiating、Connection Peripheral role、Connection Central role。

而对于 tl\_ble\_sdk，由于要同时支持多个 Central 和 Peripheral，Link Layer 无法做到在某一时刻只处于某一种状态，必须是几种状态的组合。

tl\_ble\_sdk 的 Link Layer 状态机比较复杂，只做一个大致的介绍，能够满足用户对底层的基本理解以及相应 API 的使用。

#### 3.2.2.1 Link Layer 状态机初始化

tl\_ble\_sdk 将每个基本状态按照模块化的设计，对需要使用的模块，需要提前初始化。

MCU 的初始化是必须的，API 如下：



```
void     blc_ll_initBasicMCU (void);
```

Standby 模块的添加 API 如下，这个是必须的，所有的 BLE 应用都需要初始化。

```
void     blc_ll_initStandby_module (u8 *public_addr);
```

实参 public\_addr 是 BLE public mac address 的指针。

其他几个状态（Advertising、Scanning、ACL Central、ACL Peripheral）对应模块的初始化 API 分别如下：

```
void     blc_ll_initLegacyAdvertising_module(void);
void     blc_ll_initLegacyScanning_module(void);
void     blc_ll_initAclConnection_module(void);
void     blc_ll_initAclCentralRole_module (void);
void     blc_ll_initAclPeripheralRole_module(void);
```

### 3.2.2.2 Link Layer 状态组合

Initiating 状态相对比较简单，当 Scan 状态需要对某个广播设备发起连接时，Link Layer 进入 Initiating 状态，在一定的时间内（这个时间称为 create connection timeout）要么建立连接成功，多出一个 Connection Central role，要么建立连接失败，Link Layer 重新回到 Scanning 状态。为了简化 Link Layer 状态机的介绍，更方便用户的理解，下面的介绍中都忽略 Initiating 这个短暂的临时状态。

tl\_ble\_sdk Link Layer 状态机可以从两个角度去描述，一是 Advertising 和 Peripheral 的转换；二是 Scanning 和 Central 的转换；这两个角度之间互不影响。

以 C1P1 为例分析，假设用户的 appMaxCentralNum 和 appMaxPeripheralNum 都是 1。C1P1 Advertising 和 Peripheral 切换的状态机如下：

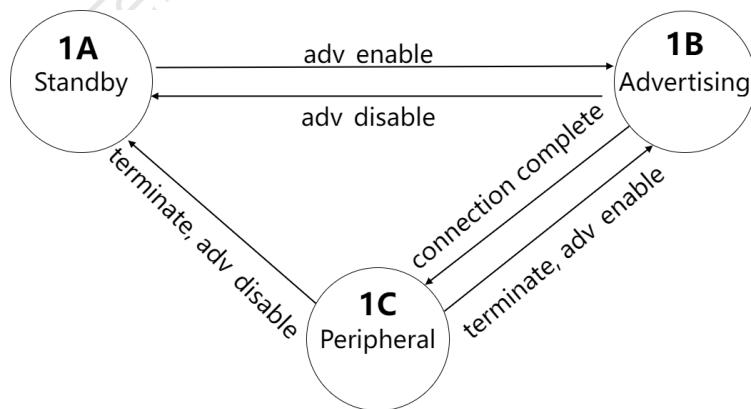
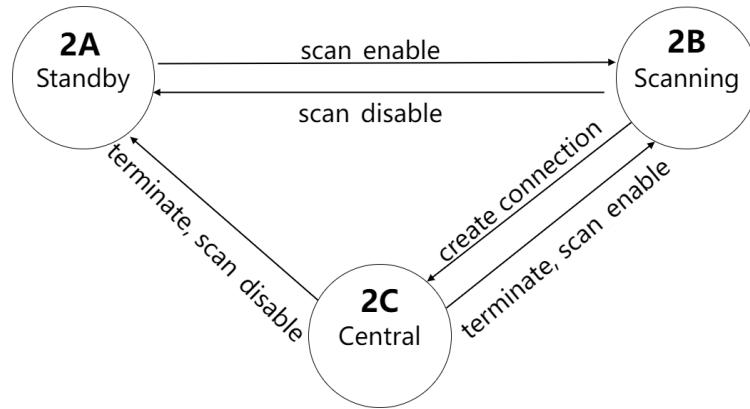


Figure 3.5: C1P1 Advertising 和 Peripheral 切换

图中 adv\_enable 和 adv\_disable 指的是条件发生时，用户最后一次调用 blc\_ll\_setAdvEnable(adv\_enable) 设定的状态。

C1P1 Scanning 和 Central 切换的状态机如下：

**Figure 3.6: C1P1 Scanning 和 Central 切换**

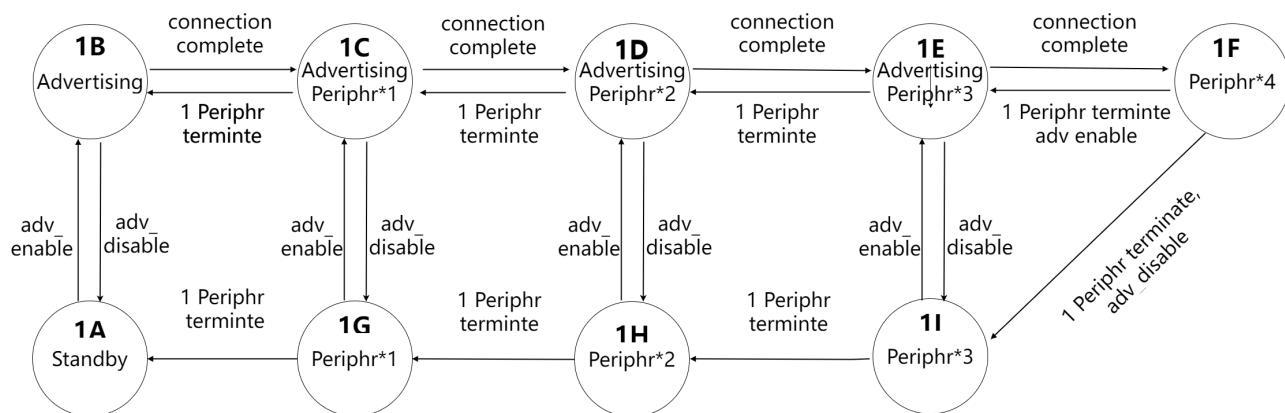
图中 scan\_enable 和 scan\_disable 指的是条件发生时，用户最后一次调用 bIc\_ll\_setScanEnable(scan\_enable, filter\_duplicate) 设定的状态。

Advertising 和 Peripheral、Scanning 和 Central 都各自有 3 种状态，由于这二者之间逻辑完全独立，互相不影响，那么最终 Link Layer 组合状态共有  $3 \times 3 = 9$  种，如下表所示：

**Table 3.2: C1P1 Link Layer 组合状态**

	2A	2B	2C
1A	Standby	Scanning	Central
1B	Advertising	Advertising + Scanning	Advertising + Central
1C	Peripheral	Peripheral + Scanning	Peripheral + Central

以一个较为复杂的 C4P4 分析，假设用户的 appMaxCentralNum 和 appMaxPeripheralNum 分别是 4 和 4，C4P4 Advertising 和 Peripheral 切换的状态机如下：

**Figure 3.7: C4P4 Advertising 和 Peripheral 切换**

C4P4 Scanning 和 Central 切换的状态机如下：

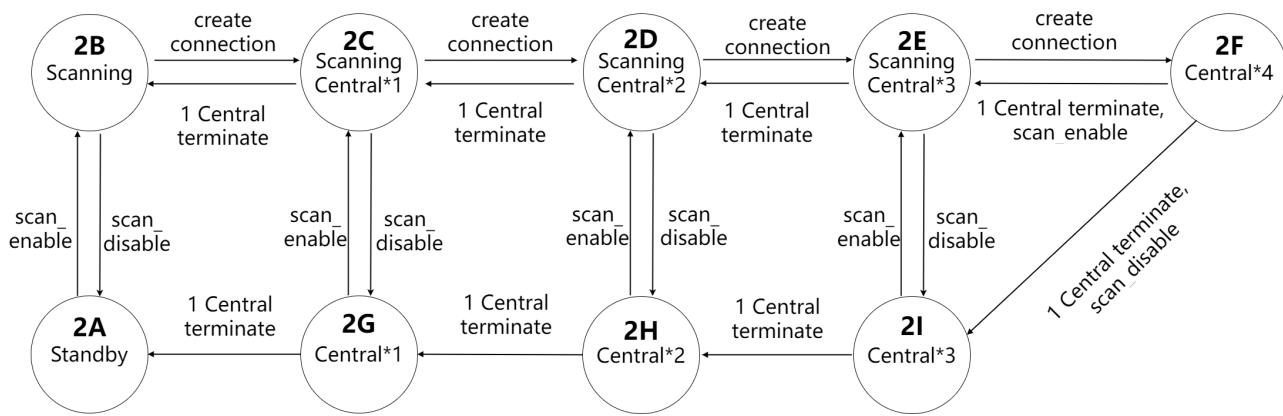


Figure 3.8: C4P4 Scanning 和 Central 切换

Advertising 和 Peripheral 有 9 种可能的状态, Scanning 和 Central 有 9 种可能的状态, 由于这二者之间逻辑完全独立, 互相不影响, 那么最终 Link Layer 组合状态共有  $9 \times 9 = 81$  种, 如下表所示:

Table 3.3: C4P4 Link Layer 组合状态

/	2A	2B	2C	2D	2E	2F	2G	2H	2I
1A	Standby	Scanning	Scanning Central*1	Scanning Central*2	Scanning Central*3	Central*4	Central*1	Central*2	Central*3
1B	Adv	Adv	Adv	Adv	Adv	Adv	Adv	Adv	Adv
		Scanning	Scanning Central*1	Scanning Central*2	Scanning Central*3				
1C	Adv	Adv	Adv	Adv	Adv	Adv	Adv	Adv	Adv
	Periph*1	Periph*1	Periph*1	Periph*1	Periph*1	Periph*1	Periph*1	Periph*1	Periph*1
	Scanning	Scanning Central*1	Scanning Central*2	Scanning Central*3		Central*4	Central*1	Central*2	Central*3
1D	Adv	Adv	Adv	Adv	Adv	Adv	Adv	Adv	Adv
	Periph*2	Periph*2	Periph*2	Periph*2	Periph*2	Periph*2	Periph*2	Periph*2	Periph*2
	Scanning	Scanning Central*1	Scanning Central*2	Scanning Central*3		Central*4	Central*1	Central*2	Central*3
1E	Adv	Adv	Adv	Adv	Adv	Adv	Adv	Adv	Adv
	Periph*3	Periph*3	Periph*3	Periph*3	Periph*3	Periph*3	Periph*3	Periph*3	Periph*3
	Scanning	Scanning Central*1	Scanning Central*2	Scanning Central*3		Central*4	Central*1	Central*2	Central*3
1F	Periph*4	Periph*4	Periph*4	Periph*4	Periph*4	Periph*4	Periph*4	Periph*4	Periph*4
	Scanning	Scanning Central*1	Scanning Central*2	Scanning Central*3		Central*4	Central*1	Central*2	Central*3
1G	Periph*1	Periph*1	Periph*1	Periph*1	Periph*1	Periph*1	Periph*1	Periph*1	Periph*1
	Scanning	Scanning Central*1	Scanning Central*2	Scanning Central*3		Central*4	Central*1	Central*2	Central*3



/	2A	2B	2C	2D	2E	2F	2G	2H	2I
1H	Periph*2	Periph*2	Periph*2	Periph*2	Periph*2	Periph*2	Periph*2	Periph*2	Periph*2
	Scanning	Scanning	Scanning	Scanning	Central*4	Central*1	Central*2	Central*3	
	Central*1	Central*2	Central*3						
1I	Periph*3	Periph*3	Periph*3	Periph*3	Periph*3	Periph*3	Periph*3	Periph*3	Periph*3
	Scanning	Scanning	Scanning	Scanning	Central*4	Central*1	Central*2	Central*3	
	Central*1	Central*2	Central*3						

如果用户的 appMaxCentralNum/appMaxPeriphNum 不是 C1P1 或者 C4P4，请根据以上分析方法去分析。

前面介绍了 supportedMaxCentralNum / supportedMaxPeripheralNum 和 appMaxCentralNum / appMaxPeripheralNum 的概念，对应上面状态机组合表里面 Central 和 Peripheral 的个数，再定义两个概念 currentCentralNum 和 currentPeripheralNum，表示当前时刻 Link Layer 实际 Central 和 Peripheral 的数量，比如在上面表中 ‘1D2E’ 组合状态时，currentCentralNum 为 3，currentPeripheralNum 为 2。

### 3.2.3 Link Layer 时序

Link Layer 时序比较复杂，这里只介绍一些最基本的知识，足以让用户理解，并合理使用相关 API。

Link Layer 5 种基本的单状态 Standby、Advertising、Scanning、Initiating、Connection，忽略只有 Central create connection 时才用到的短暂的 Initiating，我们只对剩余 4 种状态的时序做简单的介绍。

本节我们以 C4P4 状态为例来说明，假设 appMaxCentralNum 和 appMaxPeripheralNum 分别是 4 和 4。

各个子状态 (Advertising、Central0 ~ Central3、Peripheral0 ~ Peripheral3、Scanning、UI task) 会以下图为指示：

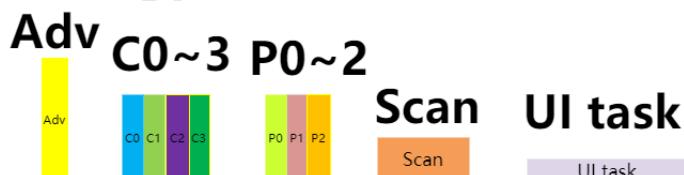


Figure 3.9: 各个子状态指示

#### 3.2.3.1 Standby state 时序

对应 C4P4 在表 3.3 中的 1A2A 状态。

当 Link Layer 处于 Idle state 时，Link Layer 和 Physical Layer 没有任何任务要处理，blc\_sdk\_main\_loop 函数完全不起作用，也不会产生任何中断。可以认为 UI entry (UI task) 占据了整个 main\_loop 的时间。

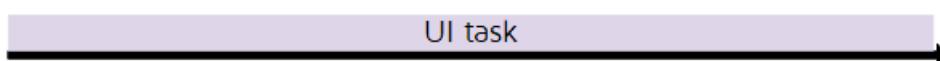


Figure 3.10: Standby 状态时序



### 3.2.3.2 Scanning only, no Advertising, no Connection 时序

对应 C4P4 在表 3.3 中的 1A2B 状态。

此时只需要处理 Scanning 状态，Scan 的效率最高。Link Layer 根据 Scan interval 去切换 channel 37/38/39，时序图如下：

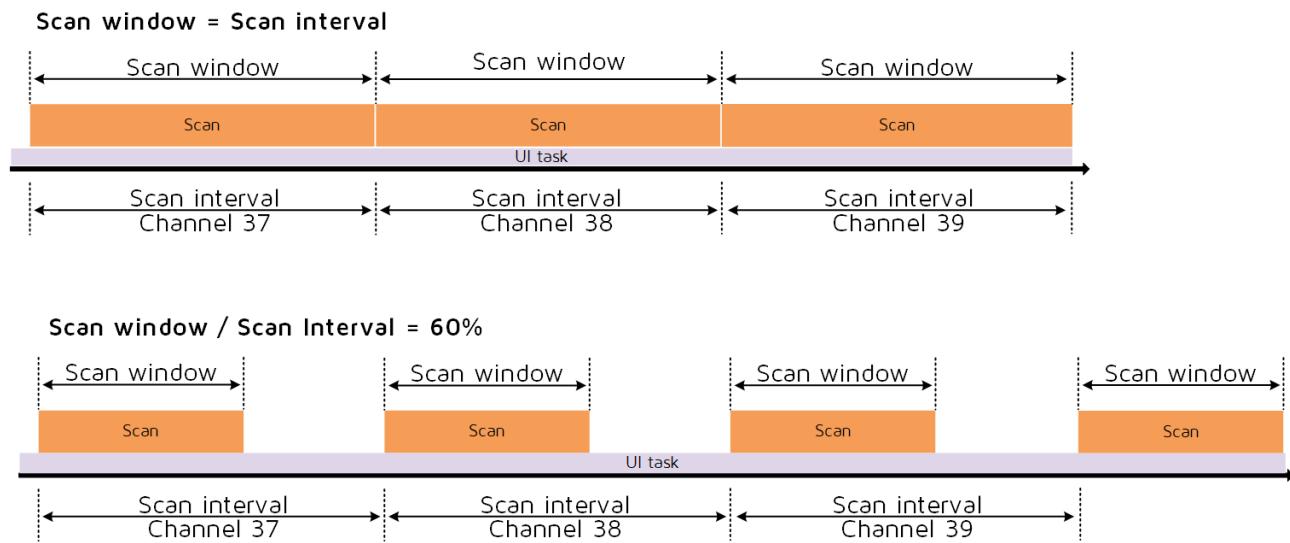


Figure 3.11: Scanning 状态时序分配

根据 Scan window 的大小决定真实的 Scan 时间，如果 Scan window 等于 Scan interval，所有的时间都在 Scan；如果 Scan window 小于 Scan interval，在 Scan interval 里面选择从最前面开始和 Scan window 相等的时间来进行 Scan。

图上所示的 Scan window 大约是 Scan interval 的 60%，在前 60% 的时间里，Link Layer 处于 Scanning 状态，PHY 层在收包，同时用户可以利用这段时间在 main\_loop 中执行自己的 UI task。后 40% 的时间不是 Scanning 状态，PHY 层停止工作，用户可以利用这段时间在 main\_loop 中执行自己的 UI task，对于后面将要介绍的低功耗管理的设计，这段时间也可以让 MCU 进入 suspend 以降低整机功耗。

### 3.2.3.3 Advertising only, no Scanning, no Connection 时序

对应 C4P4 在表 3.3 中的 1B2A 状态。

根据 Adv interval 将 Advertising Event 分配到时间轴上，时序图如下：

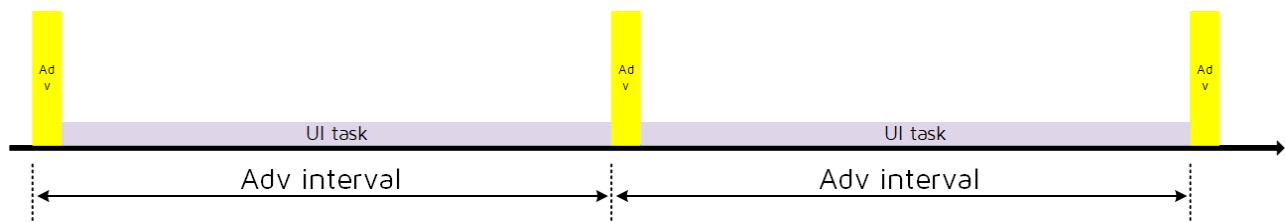


Figure 3.12: Advertising 状态时序分配



Adv Event 的所有细节参考 Telink B91 BLE Single Connection SDK Handbook 中 Adv Event 的详细介绍即可，二者是一样的。

用户可以利用非 Adv 时间在 main\_loop 中执行自己的 UI task。

### 3.2.3.4 Advertising, Scanning, no Connection 时序

对应 C4P4 在表 3.3 中的 1B2B 状态。

首先根据 Adv interval 先将 Advertising Event 分配到时间轴上，然后再分配 Scanning，时序图如下：

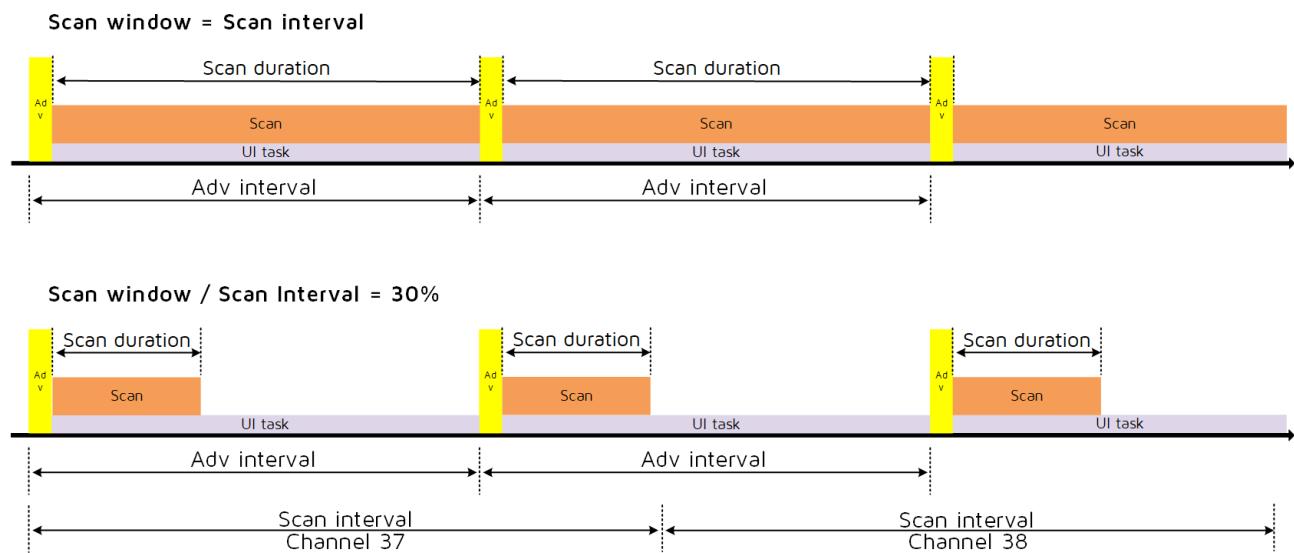


Figure 3.13: Advertising + Scanning 状态时序分配

由于应用上对广播的时间准确性要求比扫描更高，此时 Adv Event 具有较高优先级，先分配好 Adv Event 的时序，然后将 Adv Event 之间的剩余时间用来做 Scan，同时用户可以利用这段剩余时间在 main\_loop 中执行自己的 UI task。当用户设置的 Scan window 等于 Scan interval 时，图中的 Scan duration 会填满剩余时间；当用户设置的 Scan window 小于 Scan interval 时，Link Layer 会自动计算，得到一个 Scan duration 满足以下条件： $\text{Scan duration}/(\text{Adv interval} + \text{rand\_dly}) \leq \text{Scan window}/\text{Scan interval}$ 。

### 3.2.3.5 Connection, Advertising, Scanning 时序

Connection 连接的数量还没达到设定的最大值，此时仍然有 advertising 和 scanning 状态存在。

下图对应 C4P4 在表 3.3 中的 1C2C 状态。

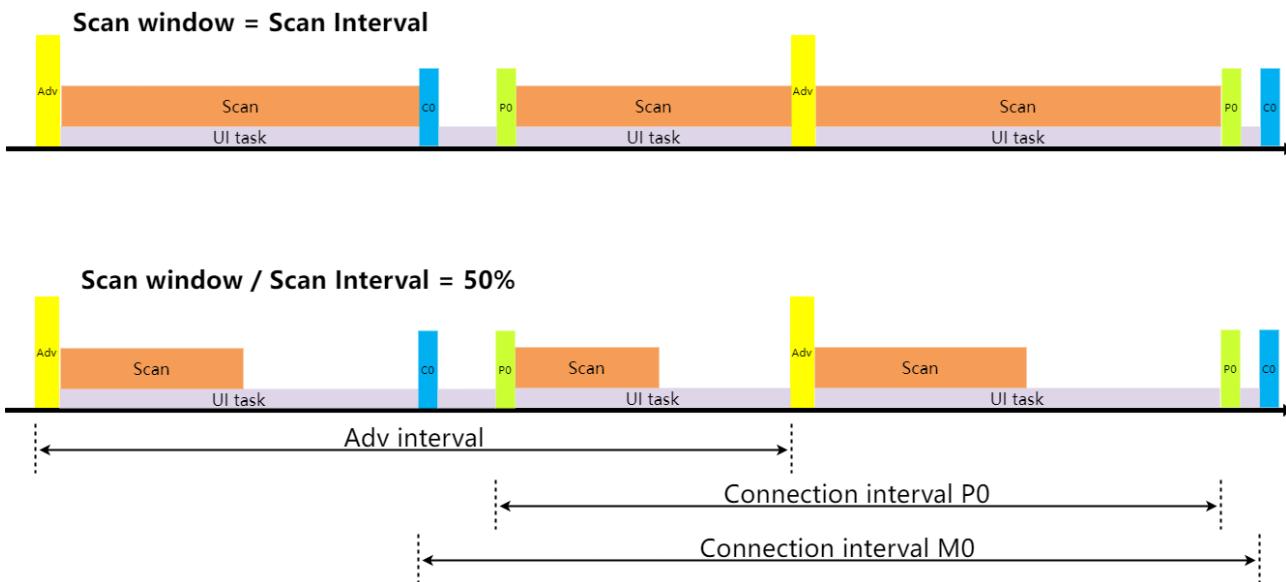


Figure 3.14: C4P4 1C2C 状态时序分配

首先进行连接任务的分配（不管是 Central 还是 Peripheral），会按照各自连接的时序进行分配。如果多个任务占用了同一个时间段而发生冲突，则会按照优先级高低进行分配，高优先级抢占。舍弃的任务会自动增加优先级，以保证不会一直被丢弃。

然后进行 adv 任务的分配，原则是：

- (1) 和上次 adv 事件的时间间隔要大于设置的最小 adv interval 时间。
- (2) 和下一个任务之间的时间大于一定值 (3.75ms)，因为 adv 完成需要一定的时间。
- (3) 分配的时间段没有其他连接任务占用。

最后进行 scan 任务的分配，原则是：只要两个任务之间有比较充足的时间，这段时间就会分配给 scan 任务，同样也会根据 Scan window/Scan interval 来确认 scan 的百分比。

### 3.2.3.6 Connection, no Advertising, no Scanning 时序

Connection 连接数量已经达到了设定的最大值，此时不存在 advertising 和 scanning 状态。

下图对应 C4P4 在表 3.3 中的 1G2H 状态。

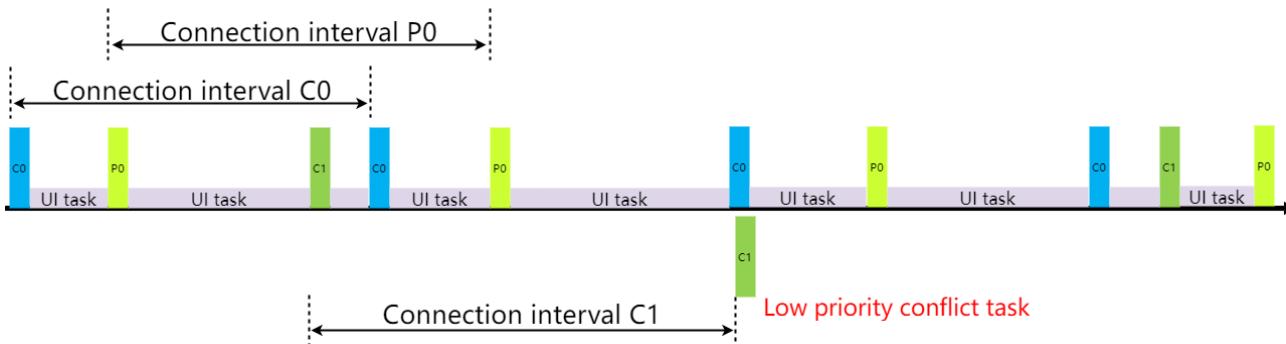


Figure 3.15: C4P4 1G2H 状态时序分配



下图对应 C4P4 在表 3.3 中的 1E2F 状态。

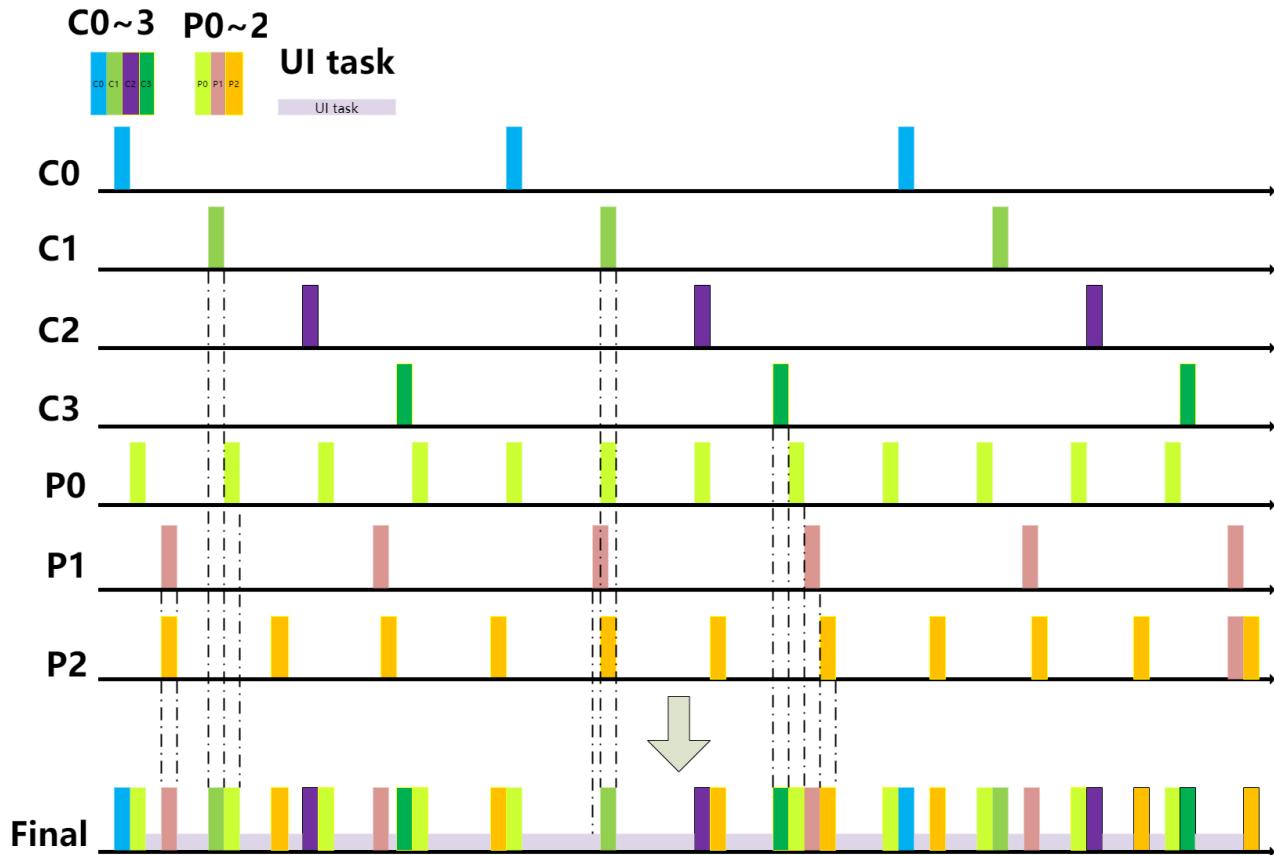


Figure 3.16: C4P4 1I2F 状态时序分配

此时只有连接任务，按照各自连接的时序进行任务分配。如果发生了冲突，则以优先级高低进行分配，高优先级任务抢占，被放弃的任务会自动增加优先级，增加下次冲突时抢占的几率。

### 3.2.4 ACL TX FIFO & ACL RX FIFO

#### 3.2.4.1 ACL TX FIFO 定义及设置

应用层和 BLE Host 所有的数据最终都需要通过 Controller 的 Link Layer 完成 RF 数据的发送，在 Link Layer 中根据 user 设置的连接数量，定义了相应的 TX FIFO。

ACL TX FIFO 的定义如下：

```
u8 app_acl_cen_tx_fifo[ACL_CENTRAL_TX_FIFO_SIZE * ACL_CENTRAL_TX_FIFO_NUM *  
    ↵ ACL_CENTRAL_MAX_NUM] = {0};  
u8 app_acl_per_tx_fifo[ACL_PERIPH_RX_FIFO_SIZE * ACL_PERIPH_RX_FIFO_NUM *  
    ↵ ACL_PERIPH_MAX_NUM] = {0};
```

ACL Central 和 ACL Peripheral 的 TX FIFO 分开定义。以 ACL Peripheral 为例说明，ACL Central 原理一样，类推即可。



数组 app\_acl\_per\_tx\_fifo 的大小与三个宏相关：

- (1) ACL\_PERIPH\_MAX\_NUM 是最大连接数量，即 appMaxPeripheralNum。用户可以根据需要在 app\_config.h 中修改这个值。
- (2) ACL\_PERIPH\_TX\_FIFO\_SIZE 是每个 sub\_buffer 的 size，与 ACL Peripheral 发送数据可能的最大值相关。在 SDK 中使用如下宏定义实现。

```
#define ACL_PERIPH_TX_FIFO_SIZE      CAL_LL_ACL_TX_FIFO_SIZE(ACL_PERIPH_MAX_TX_OCTETS)
```

其中 CAL\_LL\_ACL\_TX\_FIFO\_SIZE 是一个公式，跟 MCU 的实现方式有关，不同的 MCU 计算方法可能会不一样，可以参考 app\_buffer.h 中注释说明了解细节。

ACL\_PERIPH\_MAX\_TX\_OCTETS 是用户定义的 PeripheralMaxTxOctets。如果客户用到 DLE (Data Length Extension)，这个值需要相应的修改；默认值为 27 对应不使用 DLE 时的最小值，用来节省 Sram。具体细节请参考 app\_buffer.h 中的注释说明，以及 Bluetooth Core Specification 中的描述。

- (3) ACL\_PERIPH\_TX\_FIFO\_NUM, sub\_buffer 的 number，该值的选取请参考 app\_buffer.h 中的注释说明。该值与客户应用中数据发送量有一定关系，如果数据发送量大且实时性要求较高时，可以考虑 number 大一些。

user 根据实际情况去定义 TX FIFO，并且 Central TX FIFO 和 Peripheral TX FIFO 分开进行定义，这样一是为每个 connection 的数据分别缓存在各自的 TX FIFO 中，各个 connection 之间 TX 数据不会相互干扰；二是也可以根据实际情况，灵活定义 TX FIFO 的大小，相应的减少 ram 的消耗。比如：

- Peripheral 需要 DLE 功能，而 Central 不需要 DLE，这样就可以分别定义 FIFO，节省 ram 空间。关于 DLE 的讲解，请参考[3.2.5 MTU 和 DLE 章节](#)。
- 比如客户实际使用的是 3 主 2 从，客户就可以只定义 3 个 Central tx fifo，2 个 Peripheral tx fifo，从而减少 ram 的使用，节省 ram 空间：

```
#define ACL_CENTRAL_MAX_NUM          3  
#define ACL_PERIPH_MAX_NUM           2
```

下面我们以图来描述一下各种状态下 TX FIFO 的设置，让大家有一个更直观的认识。这里以 B91 为例，其他芯片是同样的原理，类推即可。

- (1) C4P4，ACL Central、ACL Peripheral 都不使用 DLE

假设相关的定义如下：

```
#define ACL_CENTRAL_MAX_NUM          4  
#define ACL_PERIPH_MAX_NUM           4  
#define ACL_CENTRAL_MAX_TX_OCTETS    27  
#define ACL_PERIPH_MAX_TX_OCTETS     27  
  
#define ACL_CENTRAL_TX_FIFO_SIZE     CAL_LL_ACL_TX_FIFO_SIZE(ACL_CENTRAL_MAX_TX_OCTETS)  
#define ACL_CENTRAL_TX_FIFO_NUM       8  
#define ACL_PERIPH_TX_FIFO_SIZE      CAL_LL_ACL_TX_FIFO_SIZE(ACL_PERIPH_MAX_TX_OCTETS)  
#define ACL_PERIPH_TX_FIFO_NUM        8
```



则 TX FIFO 定义为下面的值：

```
u8 app_acl_cen_tx_fifo[40 * 8 * 4] = {0};  
u8 app_acl_per_tx_fifo[40 * 8 * 4] = {0};
```

图示如下：

每一个 connection 都对应一个 tx fifo，并且每一个 connection fifo 的数量都是 8 (0 ~ 7)，0 ~ 7 的 size 都是一样的 (40B)：

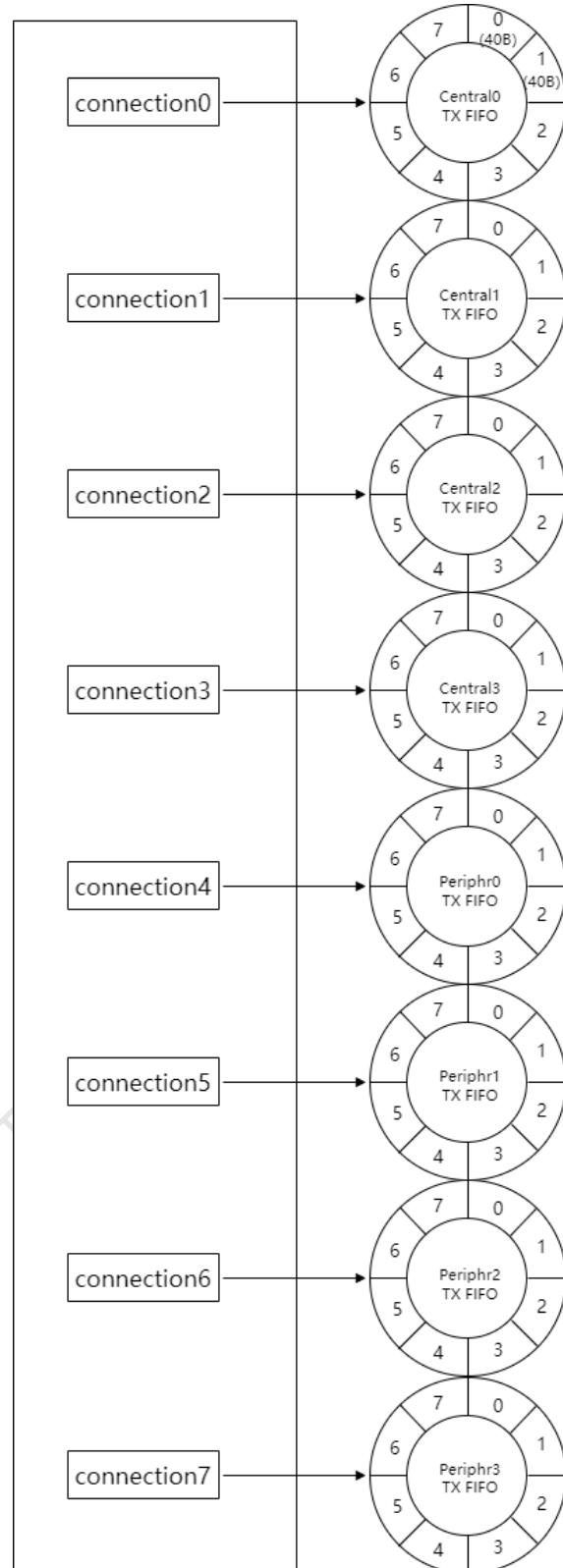


Figure 3.17: TX FIFO 默认设置

(2) C4P4, ACL Central 使用了 DLE 且 CentralMaxTxOctets 为最大值 251, ACL Peripheral 不使用 DLE。

假设相关的定义如下：

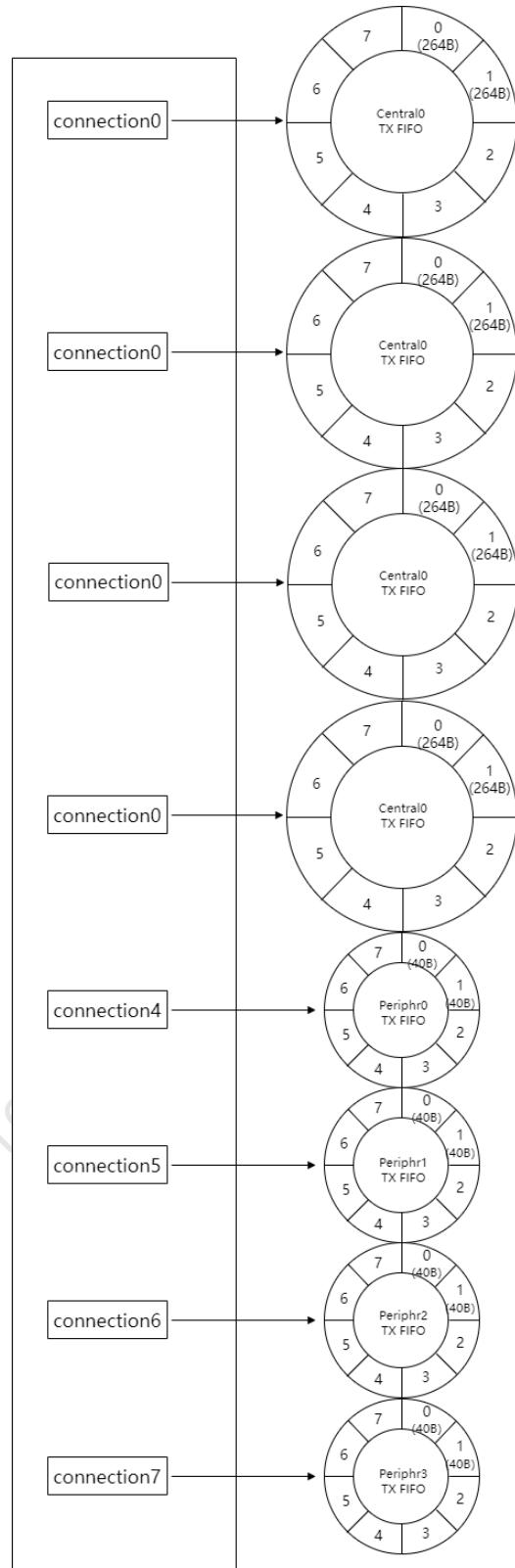


#define ACL_CENTRAL_MAX_NUM	4
#define ACL_PERIPH_MAX_NUM	4
#define ACL_CENTRAL_MAX_TX_OCTETS	251
#define ACL_PERIPH_MAX_TX_OCTETS	27
#define ACL_CENTRAL_TX_FIFO_SIZE	CAL_LL_ACL_TX_FIFO_SIZE(ACL_CENTRAL_MAX_TX_OCTETS)
#define ACL_CENTRAL_TX_FIFO_NUM	8
#define ACL_PERIPH_TX_FIFO_SIZE	CAL_LL_ACL_TX_FIFO_SIZE(ACL_PERIPH_MAX_TX_OCTETS)
#define ACL_PERIPH_TX_FIFO_NUM	8

则 TX FIFO 定义为下面的值：

```
u8 app_acl_cen_tx_fifo[264 * 8 * 4] = {0};  
u8 app_acl_per_tx_fifo[40 * 8 * 4] = {0};
```

从图可以看出，Central 和 Peripheral 每个 connection 的 fifo 数量是一样的，都是 8 个 (0~7)。但是 Central 每个 fifo size 是 264B，而 Peripheral 的 fifo size 是 40B。



**Figure 3.18:** Central 使用 DLE，Peripheral 不使用 DLE 的 buffer 情况

(3) C3P2，ACL Central、ACL Peripheral 都不使用 DLE

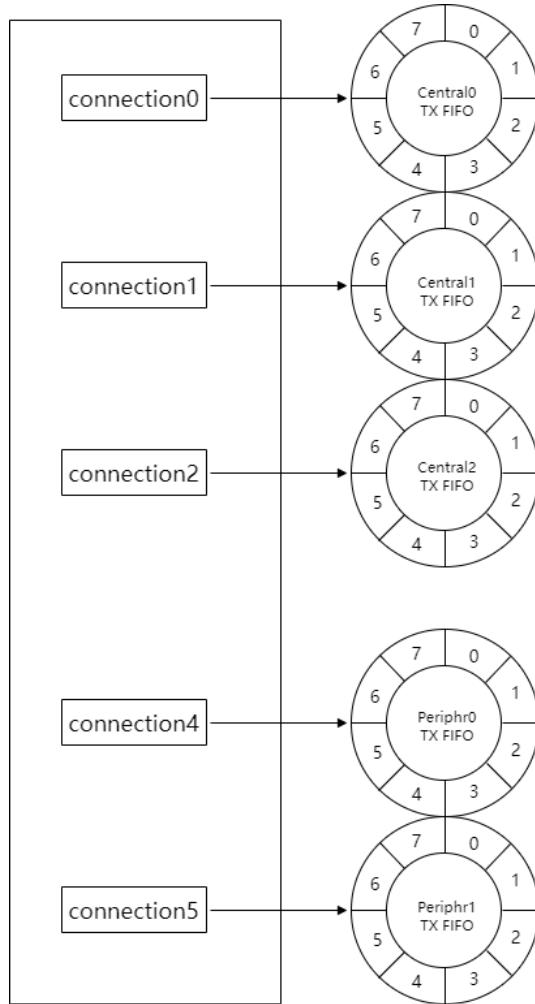


Figure 3.19: 3 个 Central, 2 个 Peripheral 的 buffer 情况

### 3.2.4.2 ACL RX FIFO 定义及设置

ACL RX FIFO 的定义如下：

```
u8 app_acl_rx_fifo[ACL_RX_FIFO_SIZE * ACL_RX_FIFO_NUM] = {0};
```

ACL Central 和 ACL Peripheral 共用 ACL RX FIFO。以 ACL Peripheral 为例说明，ACL Central 原理一样，类推即可。

数组 app\_acl\_rx\_fifo 的大小与二个宏相关：

- (1) ACL\_RX\_FIFO\_SIZE 是 buffer size，与 ACL Peripheral 接收数据可能的最大值相关。在 SDK 中使用如下宏定义实现。

```
#define ACL_RX_FIFO_SIZE          CAL_LL_ACL_RX_FIFO_SIZE(ACL_CONN_MAX_RX_OCTETS)
```

其中 CAL\_LL\_ACL\_RX\_FIFO\_SIZE 是一个公式，跟 MCU 的实现方式有关，不同的 MCU 计算方法可能会不一样，可以参考 app\_buffer.h 中注释说明了解细节。



ACL\_CONN\_MAX\_RX\_OCTETS 是用户定义的 PeripheralMaxRxOctets。如果客户用到 DLE (Data Length Extension)，这个值需要相应的修改；默认值为 27 对应不使用 DLE 时的最小值，用来节省 Sram。具体细节请参考 app\_buffer.h 中的注释说明，以及 Bluetooth Core Specification 中的详细描述。

- (2) ACL\_RX\_FIFO\_NUM, buffer number, 该值的选取请参考 app\_buffer.h 中的注释说明。该值与客户应用中数据接收量有一定关系，如果数据接收量大且实时性要求较高时，可以考虑 number 大一些。

假设 C4P4 ACL RX FIFO 的定义如下：

```
#define ACL_CENTRAL_MAX_NUM          4  
#define ACL_PERIPH_MAX_NUM          4  
  
#define ACL_CONN_MAX_RX_OCTETS      27  
#define ACL_RX_FIFO_SIZE           CAL_LL_ACL_RX_FIFO_SIZE(ACL_CONN_MAX_RX_OCTETS)  
#define ACL_RX_FIFO_NUM            16
```

对应的 ACL RX FIFO 分配如下图所示：

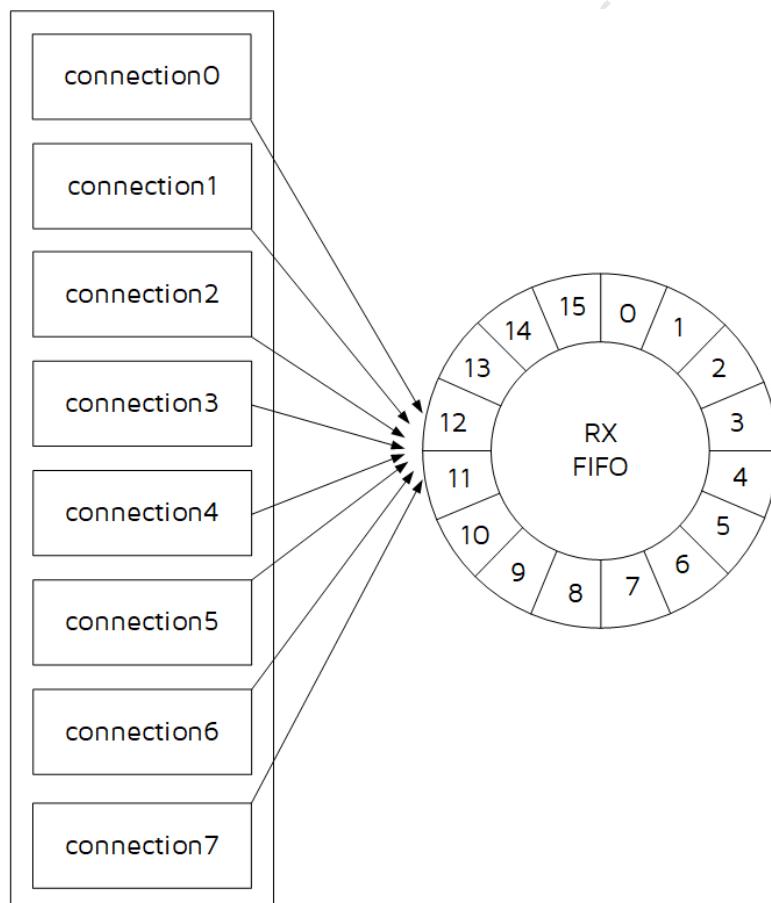


Figure 3.20: ACL RX buffer

### 3.2.4.3 RX overflow 分析

参考 Telink B91 BLE Single Connection Handbook 上的介绍，原理一样。



### 3.2.5 MTU 和 DLE 概念及使用方法

#### 3.2.5.1 MTU 和 DLE 概念说明

Bluetooth Core Specification 从 4.2 版本开始增加了 data length extension (DLE)。

tl\_ble\_sdk Link Layer 上支持 data length extension，且 rf\_len 长度支持到 Bluetooth Core Specification 上最大长度 251 bytes。详情请参考 Bluetooth Core Specification V5.4, Vol 6, Part B, 2.4.2.21 LL\_LENGTH\_REQ and LL\_LENGTH\_RSP。

在具体讲解之前，我们需要搞清楚 MTU 和 DLE 的概念是什么。先看一张图：

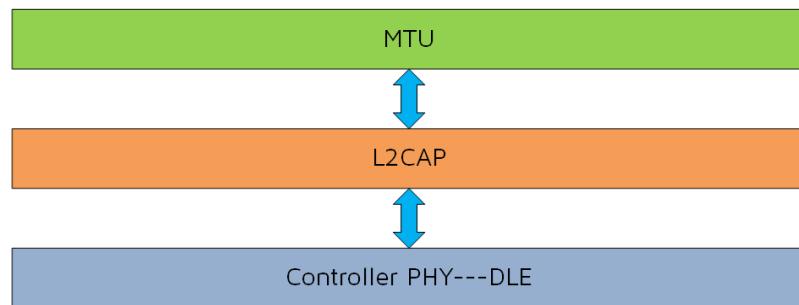


Figure 3.21: MTU 与 DLE 概念

下图是 MTU 和 DLE 所包含的内容：

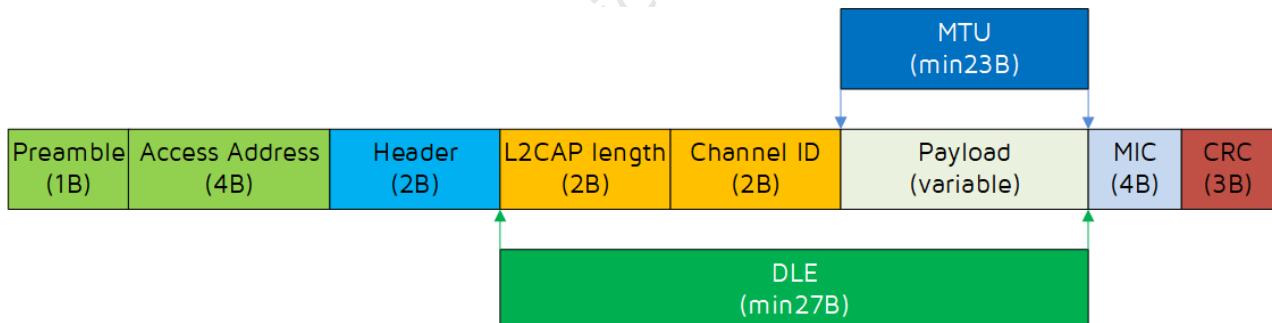


Figure 3.22: MTU 与 DLE 包含内容

- MTU 代表最大传输单元，在计算机网络中用于定义可以由特定协议发送的协议数据单元 (PDU, Protocol Data Unit) 的最大大小。
- Attribute MTU (规范定义的 ATT\_MTU) 是客户端和服务器之间可以发送的最大 ATT payload 大小。Bluetooth Core Specification 规定 MTU 最小值是 23 bytes。

所谓的 DLE，就是 data length extension 数据长度扩展。Bluetooth Core Specification 规定 DLE 最小值是 27 bytes。

如果要想在一个 packet 中携带更多数据，就需要 Central 和 Peripheral 之间进行协商，通过 LL\_LENGTH\_REQ 和 LL\_LENGTH\_RSP 交互 DLE 大小。

Bluetooth Core Specification 规定 DLE 长度最小为 27 bytes，最大是 251 bytes。251bytes 是因为 rf length 字段是一个 byte，能表示的最大长度是 255，如果是加密链路，还需要 4 bytes MIC 字段： $251 + 4 = 255$ 。



### 3.2.5.2 MTU 和 DLE 自动交互方法

User 如果需要使用 data length extension 功能，按如下步骤设置。SDK 中也提供了相应的 MTU&DLE 使用 demo，参考 feature\_test 工程中的 feature\_dle。

在 vendor/feature\_test/feature\_config.h 中定义宏：

```
#define FEATURE_TEST_MODE TEST_LL_DLE
```

#### (1) MTU size exchange

首先需要进行 MTU 的交互，只需要修改 CENTRAL\_ATT\_RX\_MTU 和 PERIPH\_ATT\_RX\_MTU 即可，它们分别代表 ACL Central 和 ACL Peripheral 的 RX MTU size。默认为最小的值是 23，改为想要的值即可。CAL\_MTU\_BUFF\_SIZE 是固定的计算公式，不能修改。

MTU size exchange 的流程确保双方最小值 MTU size 生效，防止 peer device 在 BLE L2cap 层无法处理长包，并且大于等于 23。

<code>#define CENTRAL_ATT_RX_MTU</code>	23
<code>#define PERIPH_ATT_RX_MTU</code>	23
<code>#define CENTRAL_L2CAP_BUFF_SIZE</code>	<code>CAL_L2CAP_BUFF_SIZE(CENTRAL_ATT_RX_MTU)</code>
<code>#define PERIPH_L2CAP_BUFF_SIZE</code>	<code>CAL_L2CAP_BUFF_SIZE(PERIPH_ATT_RX_MTU)</code>

然后初始化里面调用以下 API 分别设置 ACL Central 和 ACL Peripheral 的 RX MTU size。**注意：**这两个 API 需要放到 blc\_gap\_init() 之后才会生效。

```
blc_att_setCentralRxMTUSize(CENTRAL_ATT_RX_MTU);  
blc_att_setPeripheralRxMTUSize(PERIPH_ATT_RX_MTU);
```

MTU size exchange 的实现，请参考本文档“ATT & GATT”部分的详细说明—[Exchange MTU Request, Exchange MTU Response](#)，也可以参考 feature\_test 工程中 feature\_dle 的写法。

#### (2) 设置 connMaxTxOctets 和 connMaxRxOctets

其次需要设置 Central 和 Peripheral 的 DLE 的大小，参考 ACL TX FIFO 和 ACL RX FIFO 的介绍，只需要修改下面几个宏即可，27 改为想要的值。如果这些数值不是默认的 27，stack 在连接后会自动进行 DLE 的交互（也可以使用 API 禁用该功能，需要的时候再进行 DLE 交互）。

<code>#define ACL_CONN_MAX_RX_OCTETS</code>	27
<code>#define ACL_CENTRAL_MAX_TX_OCTETS</code>	27
<code>#define ACL_PERIPH_MAX_TX_OCTETS</code>	27

然后初始化里面调用以下 API 分别设置 ACL Central 和 ACL Peripheral 的 rx DLE size 和 tx DLE size。



```
ble_sts_t    blc_ll_setAclConnMaxOctetsNumber(u8 maxRxOct, u8 maxTxOct_Central, u8
↪  maxTxOct_Peripheral)
```

### (3) 收发长包的操作

请 user 先参考本文档“ATT & GATT”部分的一些说明，包括 Handle Value Notification 和 Handle Value Indication，Write request 和 Write Command 等。

在以上 3 个步骤都正确完成的基础上，可以开始收发长包。

发长包调用 ATT 层的 Handle Value Notification 和 Handle Value Indication 对应的 API 即可，分别如下所示，将要发送的数据地址和长度分别带入下面的形参 “\*p” 和 “len” 即可。

```
ble_sts_t    blc_gatt_pushHandleValueNotify (u16 connHandle, u16 attHandle, u8 *p, int len);
ble_sts_t    blc_gatt_pushHandleValueIndicate (u16 connHandle, u16 attHandle, u8 *p, int len);
```

收长包只要处理 Write request 和 Write Command 对应的回调函数 “w” 即可，在回调函数里，引用形参指针指向的数据。

### 3.2.5.3 MTU 和 DLE 手动交互方法

如果 user 由于某种特殊情况，不希望 stack 自动交互 MTU/DLE，SDK 也提供了相应 API，由 user 根据具体情况来决定何时进行 MTU/DLE 的交互。手动交互大部分的设置和自动交互一致，不同的处理如下描述。

对于 MTU，在初始化的时候调用 API blc\_att\_setCentralRxMTUSize(23) 和 blc\_att\_setPeripheralRxMTUSize(23) 将初始的 MTU 设置为最小值 23，stack 比较后不会进行自动交互。当 user 需要进行 MTU 交互时，再调用这两个 API (blc\_att\_setCentralRxMTUSize 和 blc\_att\_setPeripheralRxMTUSize)，将 MTU 设置为实际值。然后再调用 API blc\_att\_requestMtuSizeExchange() 触发 MTU 交互即可。

对于 DLE，可以在初始化的时候使用 API blc\_ll\_setAutoExchangeDataLengthEnable(0) 来 disable 自动交互，等到需要交互的时候再调用 API blc\_ll\_sendDataLengthExtendReq() 触发 DLE 交互即可。这两个 API 的具体说明参考[controller API 章节](#)。

### 3.2.6 Coded PHY/2M PHY

Coded PHY 和 2M PHY 是《Core\_v5.0》新增加的 feature，很大程度上扩展了 BLE 的应用场景，Coded PHY 包含 S2(500 kbps) 和 S8(125 kbps) 以适应更远距离的应用，2M PHY(2 Mbps) 大大提高了 BLE 带宽。2M PHY/Coded PHY 可以使用在广播通道，也可以用在连接状态下的数据通道。

#### 3.2.6.1 Code PHY/2M PHY API 介绍

##### (1) API blc\_ll\_init2MPhyCodedPhy\_feature()

```
void blc_ll_init2MPhyCodedPhy_feature(void);
```

用于使能 Coded PHY/2M PHY。



## (2) API blc\_ll\_setPhy()

```
ble_sts_t blc_ll_setPhy( u16 connHandle, le_phy_prefer_mask_t all_phys, le_phy_prefer_type_t  
↪ tx_phys, le_phy_prefer_type_t rx_phys, le_ci_prefer_t phy_options);
```

Bluetooth Core Specification 标准接口，详细请参考 Bluetooth Core Specification V5.4, Vol 4, Part E, 7.8.49 LE Set PHY command。用于触发 local device 主动申请 PHY exchange。若 PHY exchange 的判定结果为可以使用新的 PHY，则 Central 设备会触发 PHY update，新的 PHY 很快会生效。

**connHandle:** Central/Peripheral connHandle 根据实际情况填写，参考“Connection Handle”。

其他参数请参考 Bluetooth Core Specification 定义、结合 SDK 上枚举类型定义和 demo 用法去理解。

### 3.2.7 Channel Selection Algorithm #2

Channel Selection Algorithm #2 是 Bluetooth Core Specification V5.0 中新添加的 Feature，拥有更强的抗干扰能力，有关信道选择算法的具体说明请参考 Bluetooth Core Specification V5.4, Vol 6, Part B, 4.5.8.3 Channel Selection algorithm #2。

tl\_ble\_sdk 中，CSA #2 默认是关闭的，用户如果选择使用此 Feature，可以手动打开，需要在 user\_init() 调用下面的 API 使能。

```
void blc_ll_initChannelSelectionAlgorithm_2_feature(void);
```

只有 local device 和 peer Central/Peripheral device 都同时支持 CSA #2(ChSel 字段都设置为 1)，CSA #2 才能用于连接。

### 3.2.8 Controller API

所示的标准 BLE 协议栈架构中，应用层是无法与 Controller 的 Link Layer 直接数据交互的，必须通过 Host 把数据往下发，最终由 Host 通过 HCI 把控制命令传送给 Link Layer。Host 通过 HCI 接口下发的所有 Controller 控制命令都在 Bluetooth Core Specification 中严格定义了，详情请参照 Bluetooth Core Specification V5.4, Vol 4, Part E: Host Controller Interface Functional Specification。

tl\_ble\_sdk 采用 Whole Stack 架构，应用层跨越 Host 直接操作设置 Link Layer，但使用的 API 都是严格按照 Bluetooth Core Specification 标准的 HCI 部分的 API。

Controller API 的声明在 stack/ble/controller 目录下的头文件中，根据 Link Layer 状态机功能的分类分为 ll.h、leg\_adv.h、leg\_scan.h、leg\_init.h、acl\_Peripheral.h、acl\_Central.h、acl\_conn.h 等，user 可以根据 Link Layer 的功能去寻找，比如跟 legacy advertising 相关功能的 API 就应该都在 leg\_adv.h 中声明。

在 stack/ble/ble\_common.h 中定义了枚举类型 ble\_sts\_t，该类型作为 SDK 中大多数 API 的返回值类型，只有调用 API 的设置参数都正确且被协议栈接受时，才会返回 BLE\_SUCCESS (值为 0)；返回的其他非 0 值都表示设置错误，且每一个不同的值都对应一种错误类型。后面的 API 具体说明中，会列举每一个 API 所有可能的返回值，并解释各个错误返回值的具体错误原因。

这个返回值类型 ble\_sts\_t 不仅限于 Link Layer 的 API，对 Host 层一些 API 也适用。



### 3.2.8.1 BLE MAC address 初始化

本文档中的 BLE MAC address 最基本的类型包括 public address 和 random static address。

调用如下 API 获得 public address 和 random static address：

```
void blc_initMacAddress(int flash_addr, u8 *mac_public, u8 *mac_random_static);
```

flash\_addr 填 flash 上存储 MAC address 的地址即可，参考文档前面的介绍[2.1.4 SDK Flash 空间的分配](#)。如果不需 random static address，上面获取的 mac\_random\_static 忽略即可。

BLE public MAC address 成功获取后，调用 Link Layer 初始化的 API，将 MAC address 传入 BLE 协议栈：

```
blc_ll_initStandby_module (mac_public); //mandatory
```

### 3.2.8.2 blc\_ll\_setAdvData

详情请参照 Bluetooth Core Specification V5.4 Vol 4, Part E, 7.8.7 LE Set Advertising Data command。

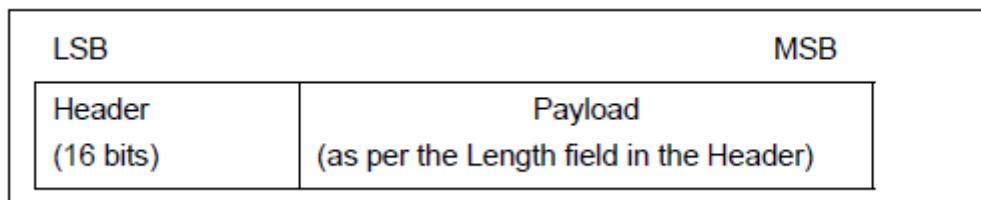


Figure 3.23: 协议栈广播包格式

BLE 协议栈里，广播包的格式如上图所示，前两个 byte 是 Header，后面是 Advertising PDU，最多 37 bytes，包含 AdvA (6B) 和 AdvData(最大 31B)。

下面的 API 用于设置 AdvData 部分的数据：

```
ble_sts_t blc_ll_setAdvData (u8 *data, u8 len);
```

data 指针指向 PDU 的首地址，len 为数据长度。

返回类型 ble\_sts\_t 可能返回的结果如下表所示：

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	操作成功

返回值 ble\_sts\_t 只有 BLE\_SUCCESS，API 不会进行参数合理性检查，user 需要注意设置参数的合理性。

user 可以在初始化的时候调用该 API 设置广播数据，也可以于程序运行时在 main\_loop 里随时调用该 API 来



修改广播数据。

### 3.2.8.3 bIIS\_ll\_setScanRspData

详情请参照 Bluetooth Core Specification V5.4, Vol 4, Part E, 7.8.8 LE Set Scan Response Data command。

类似于上面广播包 PDU 的设置，scan response PDU 的设置使用 API：

```
ble_sts_t bIc_ll_setScanRspData (u8 *data, u8 len);
```

data 指针指向 PDU 的首地址，len 为数据长度。

返回类型 ble\_sts\_t 可能返回的结果如下表所示：

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	操作成功

返回值 ble\_sts\_t 只有 BLE\_SUCCESS，API 不会进行参数合理性检查，user 需要注意设置参数的合理性。

user 可以在初始化的时候调用该 API 设置 scan response data，也可以于程序运行时在 main\_loop 里随时调用该 API 来修改 scan response data。

### 3.2.8.4 bIIS\_ll\_setAdvParam

详情请参照 Bluetooth Core Specification V5.4, Vol 4, Part E, 7.8.5 LE Set Advertising Parameters command。

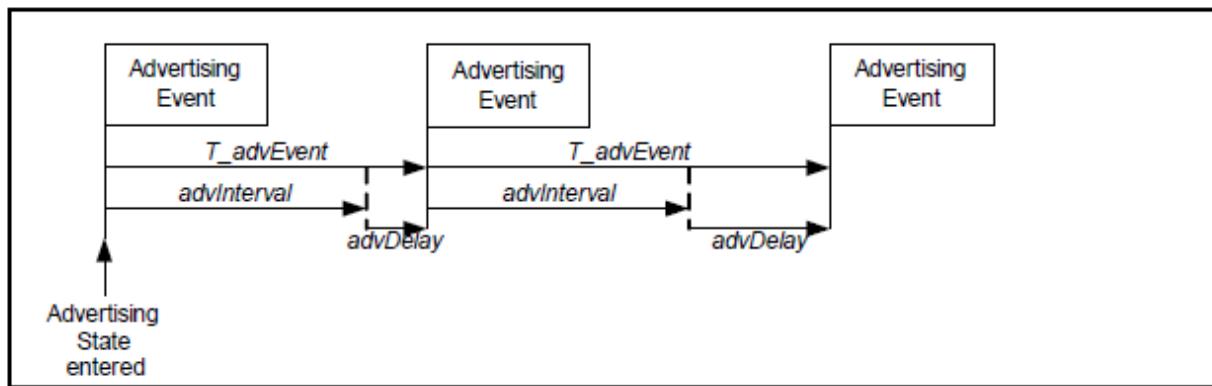


Figure 3.24: 协议栈里 Advertising Event

BLE 协议栈里 Advertising Event (简称 Adv Event) 如上图所示，指的是在每一个 T\_advEvent，Peripheral 进行一轮广播，在三个广播 channel (channel 37、channel 38、channel 39) 上各发一个包。

下面的 API 对 Adv Event 相关的参数进行设置。



```
ble_sts_t blc_ll_setAdvParam( adv_inter_t intervalMin, adv_inter_t intervalMax,
adv_type_t advType, own_addr_type_t ownAddrType,
u8 peerAddrType, u8 *peerAddr, adv_chn_map_t adv_channelMap, adv_fp_type_t advFilterPolicy);
```

### (1) intervalMin & intervalMax

设置广播时间间隔 adv interval 的范围，以 0.625 ms 为基本单位，范围在 20 ms ~ 10.24 s 之间，并且 intervalMin 小于等于 intervalMax。

SDK 在连接态下广播间隔使用 intervalMin，在非连接态下广播间隔使用 intervalMax。

若设置的 intervalMin > intervalMax，intervalMin 会被强制等于 intervalMax。

根据不同广播包的类型，intervalMin 和 intervalMax 的值有一些限定，请参照 (Vol 6/Part B/ 4.4.2.2 "Advertising Events")。

### (2) advType

参考 Bluetooth Core Specification，四种基本的广播事件类型如下：

Advertising Event Type	PDU used in this advertising event type	Allowable response PDUs for advertising event	
		SCAN_REQ	CONNECT_REQ
Connectable Undirected Event	ADV_IND	YES	YES
Connectable Directed Event	ADV_DIRECT_IND	NO	YES*
Non-connectable Undirected Event	ADV_NONCONN_IND	NO	NO
Scannable Undirected Event	ADV_SCAN_IND	YES	NO

Table 4.1: Advertising event types, PDUs used and allowable response PDUs

Figure 3.25: BLE 协议栈四种广播事件

上图中 Allowable response PDUs for advertising event 部分用 YES 和 NO 说明了各种类型广播事件是否对其他设备的 Scan request 和 Connect Request 进行响应，如：第一个 Connectable Undirected Event 对 Scan request 和 Connect Request 都能响应，而 Non-connectable Undirected Event 对它们都不响应。

注意第二个 Connectable Directed Event 对 Connect Request 响应那个“YES”右上角加了“\*”号，表示它只要收到匹配的 Connect Request，就一定会响应，而不会被 whitelist 过滤。剩下的 3 个“YES”表示可以响应相应的请求，但实际需要依赖于 whitelist 的设置，根据 whitelist 的过滤条件来决定最终是否响应，后面的 whitelist 中会详细介绍。

以上四种广播事件中，Connectable Directed Event 又分为 Low Duty Cycle Directed Advertising 和 High Duty Cycle Directed Advertising，这样一共能够得到五种广播事件类型，如下定义 (stack/ble/ble\_common.h)：



```
/* Advertisement Type */
typedef enum{
    ADV_TYPE_CONNECTABLE_UNDIRECTED      = 0x00, // ADV_IND
    ADV_TYPE_CONNECTABLE_DIRECTED_HIGH_DUTY = 0x01, //ADV INDIRECT_IND (high duty cycle)
    ADV_TYPE_SCANNABLE_UNDIRECTED        = 0x02 //ADV_SCAN_IND
    ADV_TYPE_NONCONNECTABLE_UNDIRECTED   = 0x03, //ADV_NONCONN_IND
    ADV_TYPE_CONNECTABLE_DIRECTED_LOW_DUTY = 0x04, //ADV INDIRECT_IND (low duty cycle)
}adv_type_t;
```

默认最常用的广播类型为 ADV\_TYPE\_CONNECTABLE\_UNDIRECTED。

#### (3) ownAddrType

指定广播地址类型时，ownAddrType 4 个可选的值如下：

```
typedef enum{
    OWN_ADDRESS_PUBLIC = 0,
    OWN_ADDRESS_RANDOM = 1,
    OWN_ADDRESS_RESOLVE_PRIVATE_PUBLIC = 2,
    OWN_ADDRESS_RESOLVE_PRIVATE_RANDOM = 3,
}own_addr_type_t;
```

这里只介绍前两个参数。

OWN\_ADDRESS\_PUBLIC 表示广播的时候使用 public MAC address，实际地址来自 MAC address 初始化时 API blc\_initMacAddress(flash\_sector\_mac\_address, mac\_public, mac\_random\_static) 的设置。

OWN\_ADDRESS\_RANDOM 表示广播的时候使用 random static MAC address，该地址来源于下面 API 设定的值：

```
ble_sts_t blc_ll_setRandomAddr(u8 *randomAddr);
```

#### (4) peerAddrType & \*peerAddr

当 advType 被设置为直接广播包类型 directed adv (ADV\_TYPE\_CONNECTABLE\_DIRECTED\_HIGH\_DUTY 和 ADV\_TYPE\_CONNECTABLE\_DIRECTED\_LOW\_DUTY) 时，peerAddrType 和 \*peerAddr 用于指定 peer device MAC Address 的类型和地址。

当 advType 为其他类型时，peerAddrType 和 \*peerAddr 的值都无效，可以设定为 0 和 NULL。

#### (5) adv\_channelMap

设定广播 channel，可以选择 channel 37、38、39 中任意一个或多个。adv\_channelMap 的值可设置如下 3 个或它们中任意或组合。

```
typedef enum{
    BLT_ENABLE_ADV_37    =     BIT(0),
    BLT_ENABLE_ADV_38    =     BIT(1),
    BLT_ENABLE_ADV_39    =     BIT(2),
    BLT_ENABLE_ADV_ALL   = (BLT_ENABLE_ADV_37 | BLT_ENABLE_ADV_38 | BLT_ENABLE_ADV_39),
}adv_chn_map_t;
```



## (6) advFilterPolicy

用于设定发送广播包时，对其他设备的 scan request 和 connect request 采取的过滤策略。过滤的地址需要提前存储到 whitelist 中。在后面 whitelist 介绍中详细解释。

可设置的 4 种过滤类型如下，若不需要 whitelist 过滤功能，选择 ADV\_FP\_NONE。

```
typedef enum {
    ADV_FP_ALLOW_SCAN_ANY_ALLOW_CONN_ANY = 0x00,
    ADV_FP_ALLOW_SCAN_WL_ALLOW_CONN_ANY = 0x01,
    ADV_FP_ALLOW_SCAN_ANY_ALLOW_CONN_WL = 0x02,
    ADV_FP_ALLOW_SCAN_WL_ALLOW_CONN_WL = 0x03,
    ADV_FP_NONE = ADV_FP_ALLOW_SCAN_ANY_ALLOW_CONN_ANY
} adv_fp_type_t;
```

返回值 ble\_sts\_t 可能出现的值和原因如下表所示：

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	操作成功

返回值 ble\_sts\_t 只有 BLE\_SUCCESS，API 不会进行参数合理性检查，user 需要注意设置参数的合理性。

按照 Bluetooth Core Specification HCI 部分 Host command 的设计，Set Advertising parameters 同时设置了上面的 8 个参数。同时设置的思路也是合理的，因为一些不同的参数之间是有耦合关系的，比如 advType 和 advInterval，在不同的 advType 下，对 intervalMin 和 intervalMax 的范围限定会不一样，所以会有不同的范围检查，如果将 set advType 和 set advInterval 拆成两个不同的 API，彼此间的范围检查就无法控制。

### 3.2.8.5 blc\_ll\_setAdvEnable

详情请参照 Bluetooth Core Specification V5.4, Vol 4, Part E, 7.8.9 LE Set Advertising Enable command。

```
ble_sts_t blc_ll_setAdvEnable(adv_en_t adv_enable);
```

en 为 1 时，Enable Advertising；en 为 0 时，Disable Advertising。

Enable 或 Disable Advertising 的状态机变化可以参考[3.2.2.2 Link Layer 状态组合](#)。

返回值 ble\_sts\_t 可能出现的值和原因如下表所示：

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	操作成功
HCI_ERR_CONN_REJ_LIMITED_RESOURCES	0x0D	appMaxPeripheralNum 为 0，不允许设置



### 3.2.8.6 blc\_ll\_setAdvCustomedChannel

该 API 用于定制特殊的 advertising channel & scanning channel，只对一些非常特殊的应用有意义，如 BLE mesh。

```
void     blc_ll_setAdvCustomedChannel(u8 chn0, u8 chn1, u8 chn2);
```

chn0/chn1/chn2 填需要定制的频点，默认的标准频点是 37/38/39，比如设置 3 个 advertising channel 分别为 2420 MHz、2430 MHz、2450 MHz，可如下调用：

```
blc_ll_setAdvCustomedChannel (8, 12, 22);
```

常规 BLE 应用使用该 API 可以实现这样的功能，如果 user 在一些使用场景希望用到单通道广播 & 单通道扫描，比如将 advertising channel & scanning channel 固定为 39，可如下调用：

```
blc_ll_setAdvCustomedChannel (39, 39, 39);
```

需要注意的是该 API 会同时更改广播和扫描的通道。

### 3.2.8.7 blc\_ll\_setScanParameter

详情请参照 Bluetooth Core Specification V5.4, Vol 4, Part E, 7.8.10 LE Set Scan Parameters command。

```
ble_sts_t    blc_ll_setScanParameter ( scan_type_t scan_type,
u16 scan_interval, u16 scan_window,
own_addr_type_t ownAddrType,
scan_fp_type_t scanFilter_policy);
```

参数解析：

(1) scan\_type

可选择 passive scan 和 active scan，区别是 active scan 会在收到 adv packet 基础上发 scan\_req 以获取设备 scan\_rsp 的更多信息，scan rsp 包也会通过 adv report event 传给 BLE Host；passive scan 不发 scan req。

```
typedef enum {
    SCAN_TYPE_PASSIVE    = 0x00,
    SCAN_TYPE_ACTIVE     = 0x01,
} scan_type_t;
```

(2) scan\_interval/scan window

scan\_interval 设置 Scanning state 时频点的切换时间，单位为 0.625 ms，scan\_window 为扫描窗口时间。如果设置的 scan\_window > scan\_interval，实际的 scan window 设置为 scan\_interval。

底层会根据 scan\_window / scan\_interval 计算出 scan\_percent，以达到降低功耗的目的。Scanning 的具体细节可以参考“3.2.3.2”和“3.2.3.4”和“3.2.3.5”。



## (3) ownAddrType

指定 scan req 包地址类型时，ownAddrType 有 4 个可选的值如下：

```
typedef enum{
    OWN_ADDRESS_PUBLIC = 0,
    OWN_ADDRESS_RANDOM = 1,
    OWN_ADDRESS_RESOLVE_PRIVATE_PUBLIC = 2,
    OWN_ADDRESS_RESOLVE_PRIVATE_RANDOM = 3,
}own_addr_type_t;
```

OWN\_ADDRESS\_PUBLIC 表示 Scan 的时候使用 public MAC address，实际地址来自 MAC address 初始化时 API blc\_initMacAddress(int flash\_addr, u8 mac\_public, u8 mac\_random\_static) 的设置。

OWN\_ADDRESS\_RANDOM 表示 Scan 的时候使用 random static MAC address，该地址来源于下面 API 设定的值：

```
ble_sts_t    blc_llms_setRandomAddr(u8 *randomAddr);
```

## (4) scan filter policy

```
typedef enum {
    SCAN_FP_ALLOW_ADV_ANY=0x00,//except direct adv address not match
    SCAN_FP_ALLOW_ADV_WL=0x01,//except direct adv address not match
    SCAN_FP_ALLOW_UNDIRECT_ADV=0x02,//and direct adv address match initiator's resolvable
    ↵ private MAC
    SCAN_FP_ALLOW_ADV_WL_DIRECT_ADV_MACTH=0x03, //and direct adv address match initiator's
    ↵ resolvable private MAC
} scan_fp_type_t;
```

目前支持的 scan filter policy 为下面 2 个：

SCAN\_FP\_ALLOW\_ADV\_ANY 表示 Link Layer 对 scan 到的 adv packet 不做过滤，直接 report 到 BLE Host。

SCAN\_FP\_ALLOW\_ADV\_WL 则要求 scan 到的 adv packet 必须是在 whitelist 里面的，才 report 到 BLE Host。

返回值 ble\_sts\_t 可能出现的值和原因如下表所示：

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	操作成功

返回值 ble\_sts\_t 只有 BLE\_SUCCESS，API 不会进行参数合理性检查，user 需要注意设置参数的合理性。

### 3.2.8.8 blc\_ll\_setScanEnable

详情请参照 Bluetooth Core Specification V5.4, Vol 4, Part E, 7.8.11 LE Set Scan Enable command。



```
ble_sts_t blc_ll_setScanEnable(scan_en_t scan_enable, dupFilter_en_t filter_duplicate);
```

scan\_enable 参数类型有如下 2 个可选值：

```
typedef enum {
    BLC_SCAN_DISABLE = 0x00,
    BLC_SCAN_ENABLE  = 0x01,
} scan_en_t;
```

scan\_enable 为 1 时，Enable Scanning；scan\_enable 为 0 时，Disable Scanning。

Enable/Disable Scanning 的状态机变化可以参考[3.2.2.2 Link Layer 状态组合](#)。

filter\_duplicate 参数类型有如下 2 个可选值：

```
typedef enum {
    DUP_FILTER_DISABLE = 0x00,
    DUP_FILTER_ENABLE  = 0x01,
} dupFilter_en_t;
```

filter\_duplicate 为 1 时，表示开启重复包过滤，此时对每个不同的 adv packet，Controller 只向 Host 上报一次 adv report event；filter\_duplicate 为 0 时，不开启重复包过滤，对 scan 到的 adv packet 会一直上报给 Host。

返回值 ble\_sts\_t 见下表：

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	操作成功

当设置了 scan\_type 为 active scan ([3.2.8.7 blc\\_ll\\_setScanParameter](#))、Enable Scanning 后，对每个 device，只读一次 scan\_rsp 并上报给 Host。因为每次 Enable Scanning 后，Controller 会对不同设备的 scan\_rsp 进行记录，将它们存储到 scan\_rsp 列表里，确保后面不会再次去读该设备的 scan\_req。

若 user 需要多次上报同一个 device 的 scan\_rsp，可以通过 blc\_ll\_setScanEnable 重复设置 Enable Scanning 实现，因为每次 Enable/Disable Scanning 时，设备的 scan\_rsp 列表都会清 0。

### 3.2.8.9 blc\_ll\_createConnection

详情请参照 Bluetooth Core Specification V5.4, Vol 4, Part E, 7.8.12 LE Create Connection command。

```
ble_sts_t blc_ll_createConnection(u16 scan_interval,u16 scan_window,
    init_fp_type_t initiator_filter_policy,
    u8 adr_type, u8 *mac, u8 own_adr_type,
    u16 conn_min, u16 conn_max,u16 conn_latency,
    u16 timeout, u16 ce_min, u16 ce_max )
```



## (1) scan\_inetrvl/scan window

scan\_interval/scan\_window 在该 API 中暂时没有处理。如果需要设置可以使用[“3.2.8.7 blc\\_ll\\_setScanParameter”](#)。

## (2) initiator\_filter\_policy

指定当前连接设备的策略，可选如下两种：

```
typedef enum {
    INITIATE_FP_ADV_SPECIFY = 0x00, //connect ADV specified by host
    INITIATE_FP_ADV_WL = 0x01, //connect ADV in whiteList
} init_fp_type_t;
```

INITIATE\_FP\_ADV\_SPECIFY 表示连接的设备地址是后面的 adr\_type/mac；

INITIATE\_FP\_ADV\_WL 表示根据 whitelist 里面的设备来连接，此时 adr\_type/mac 无意义。

## (3) adr\_type/ mac

initiator\_filter\_policy 为 INITIATE\_FP\_ADV\_SPECIFY 时，连接地址类型为 adr\_type (BLE\_ADDR\_PUBLIC 或者 BLE\_ADDR\_RANDOM)、地址为 mac[5...0] 的设备。

## (4) own\_addr\_type

指定建立连接的 Central role 使用的 MAC address 类型。ownAddrType 4 个可选的值如下。

```
typedef enum{
    OWN_ADDRESS_PUBLIC = 0,
    OWN_ADDRESS_RANDOM = 1,
    OWN_ADDRESS_RESOLVE_PRIVATE_PUBLIC = 2,
    OWN_ADDRESS_RESOLVE_PRIVATE_RANDOM = 3,
}own_addr_type_t;
```

OWN\_ADDRESS\_PUBLIC 表示连接的时候使用 public MAC address，实际地址来自 MAC address 初始化时 API blc\_llms\_initStandby\_module (mac\_public) 的设置。

OWN\_ADDRESS\_RANDOM 表示连接的时候使用 random static MAC address，该地址来源于下面 API 设定的值：

```
ble_sts_t blc_llms_setRandomAddr (u8 *randomAddr);
```

## (5) conn\_min/ conn\_max/ conn\_latency/ timeout

这 4 个参数规定了建立连接后 Central role 的连接参数，同时这些参数也会通过 connection request 发给 Peripheral，Peripheral 也会是同样的连接参数。

conn\_min/conn\_max 指定 conn interval 的范围，单位为 1.25 ms。如果 appMaxCentralNum > 1，conn\_min/conn\_max 参数无效，SDK 中 Central role conn interval 默认固定为 25 (实际 interval 为 31.25 ms = 25 × 1.25 ms)，这种情况下可以在建立连接前调用 blc\_ll\_setAclCentralConnectionInterval 更改设置；如果 appMaxCentralNum 为 1，SDK 中 Central role conn interval 直接使用 conn\_max 的值。



conn\_latency 指定 connection latency，一般设为 0。

timeout 指定 connection supervision timeout，单位为 10 ms。

(6) ce\_min/ ce\_max

tl\_ble\_sdk 暂未处理 ce\_min/ ce\_max。

返回值列表：

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	操作成功
HCI_ERR_CONN_REJ_LIMTED_RESOURCES	0x0D	Link Layer 已经处于 Initiating state，不再接收新的 create connection 或当前 device 处于 Connection state

API 不会进行参数合理性检查，user 需要注意设置参数的合理性。

### 3.2.8.10 blc\_ll\_setCreateConnectionTimeout

```
ble_sts_t    blc_ll_setCreateConnectionTimeout(u32 timeout_ms);
```

返回值为 BLE\_SUCCESS，timeout\_ms 单位为 ms。

当 blc\_ll\_createConnection 触发进入 Initiating state 后，如果长时间无法建立连接，会触发 Initiate timeout，退出 Initiating state。

tl\_ble\_sdk 默认的 Initiate timeout 时间为 5 秒。如果 User 不希望使用默认的这个时间，可以调用 blc\_ll\_setCreateConnectionTimeout 设置自己需要的 Initiate timeout。

### 3.2.8.11 blc\_ll\_setAclCentralConnectionInterval

```
ble_sts_t    blc_ll_setAclCentralConnectionInterval(u16 conn_interval);
```

返回值为 BLE\_SUCCESS，conn interval 单位为 1.25 ms。

这个 API 设置 Central base connection interval 基准，通过这个基准可以将多个 Central 的时序错开，保证多 Central 同时连接的情况下时序不冲突，提高数据传输效率。实际生效的 Central connection interval 是这个基准的 1/2/3/4/6/8/12 倍。

### 3.2.8.12 blc\_ll\_setAutoExchangeDataLengthEnable



```
void blc_ll_setAutoExchangeDataLengthEnable(int auto_dle_en)
```

auto\_dle\_en: 0, 禁止 stack 自动进行 DLE 交互；1, stack 主动进行 DLE 交互。

默认情况下,如果初始化 DLE 的长度不是 27,stack 在连接后会自动进行 DLE 的交互。如果用户不希望 stack 主动交互,可以使用此 API 关闭。等需要进行交互的时候,由用户自己调用相关 API blc\_ll\_sendDateLengthExtendReq 再进行交互。

**注意:**

如果要禁止 stack 自动交互 DLE, 需要在初始化的时候调用该 API。

### 3.2.8.13 blc\_ll\_sendDateLengthExtendReq

```
ble_sts_t blc_ll_sendDateLengthExtendReq (u16 connHandle, u16 maxTxOct)
```

connHandle: 指定需要更新连接参数的 connection。

maxTxOct: 需要设置的 DLE 长度。

返回类型 ble\_sts\_t 可能返回的结果如下表所示：

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	操作成功

返回值 ble\_sts\_t 只有 BLE\_SUCCESS, API 不会进行参数合理性检查, user 需要注意设置参数的合理性。

### 3.2.8.14 blc\_ll setDataLengthReqSendingTime\_after\_connCreate

```
void blc_ll setDataLengthReqSendingTime_after_connCreate(int time_ms)
```

设置 pending 时间。

用于设置在连接后等待 time\_ms (单位: 毫秒) 后执行 DLE 的交互。

### 3.2.8.15 blc\_ll\_disconnect

```
ble_sts_t blc_ll_disconnect(u16 connHandle, u8 reason);
```

调用此 API 在 Link Layer 上发送一个 terminate 给 peer Central/Peripheral device, 主动断开连接。

connHandle 为当前 connection 的 handle 值。



reason 为断开原因，reason 的设置详请参照 Bluetooth Core Specification V5.4, Vol 1, Part F, 2 Error code descriptions。

若不是系统运行异常导致的 terminate，应用层一般指定 reason 为 HCI\_ERR\_REMOTE\_USER\_TERM\_CONN = 0x13，blc\_ll\_disconnect(connHandle, HCI\_ERR\_REMOTE\_USER\_TERM\_CONN);

调用该 API 主动发起断开连接后，一定会触发 HCI\_EVT\_DISCONNECTION\_COMPLETE 事件，在该事件的回调函数里可以看到对应的 terminate reason 和这个手动设置的 reason 是一样的。

一般情况下直接调用该 API 可以成功发送 terminate 并断连，但也存在一些特殊情况会导致该 API 调用失败，根据返回值 ble\_sts\_t 可以了解对应的错误原因。建议应用层调用该 API 时，检查一下返回值是否为 BLE\_SUCCESS。

返回值如下表。

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	操作成功
HCI_ERR_UNKNOWN_CONN_ID	0x02	connHandle 错误或找不到对应的连接
HCI_ERR_CONN_REJ_LIMITED_RESOURCES	0x3E	大量数据正在发送，暂时无法接受该命令

### 3.2.8.16 rf\_set\_power\_level\_index

tl\_ble\_sdk 提供了 BLE RF packet 能量设定的 API。

API 原型如下：

```
void rf_set_power_level_index (rf_power_level_index_e idx);
```

B91 的 idx 设置参考 drivers/B91/rf.h 中定义的枚举变量 rf\_power\_level\_index\_e，其余芯片可依次类推。

该 API 设定的 RF 发包能量，对广播包和连接包同时有效，且在程序的任意位置都可以设置，实际发包时的能量以时间上最近一次的设置为准。

**注意：**

rf\_set\_power\_level\_index 这个函数内部是对 MCU RF 相关的一些寄存器进行设置，而一旦 MCU 进入 sleep (包括 suspend/deepsleep retention) 后，这些寄存器的值都会丢失。所以 user 需要注意，每次 sleep 唤醒后，这个函数必须得重新设置一遍。比如在 SDK demo 中使用了 BLT\_EV\_FLAG\_SUSPEND\_EXIT 事件回调，来确保每次 suspend 醒来 rf power 都被重新设置一遍。

### 3.2.8.17 Whitelist & Resolvinglist

前面介绍过，Advertising/Scanning/Initiating state 的 filter\_policy 中都涉及到 Whitelist，会根据 Whitelist 中的设备进行相应的操作。实际 Whitelist 概念中包含 Whitelist 和 Resolvinglist 两部分。

通过 peer\_addr\_type 和 peer\_addr 可以判断 peer device 地址类型是否 RPA (resolvable private address)。使用下面的宏判断即可。



```
#define IS_NON_RESOLVABLE_PRIVATE_ADDR(type, addr)  
((type)==BLE_ADDR_RANDOM && (addr[5] & 0xC0) == 0x00)
```

非 RPA 地址才可以存储到 whitelist 中，目前 tl\_ble\_sdk 中 whitelist 最多存储 4 个设备。

Whitelist 相关 API 如下：

```
ble_sts_t blc_ll_clearWhiteList(void);
```

Reset whitelist，返回值为 BLE\_SUCCESS。

```
ble_sts_t blc_ll_addDeviceToWhiteList(u8 type, u8 *addr);
```

添加一个设备到 whitelist，返回值列表：

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	操作成功
HCI_ERR_MEM_CAP_EXCEEDED	0x07	whitelist 已满，添加失败

```
ble_sts_t blc_ll_removeDeviceFromWhiteList(u8 type, u8 *addr);
```

从 whitelist 删除之前添加的设备，返回值为 BLE\_SUCCESS。

RPA (resolvable private address) 设备，需要使用 Resolvinglist。为了节省 ram 使用，目前 tl\_ble\_sdk 中 Resolvinglist 最多存储 2 个设备：

```
#define MAX_RESOLVING_LIST_SIZE 2
```

Resolvinglist 相关 API 如下：

```
ble_sts_t blc_ll_clearResolvingList(void);
```

Reset Resolvinglist。返回值 BLE\_SUCCESS。

```
ble_sts_t blc_ll_setAddressResolutionEnable(addr_res_en_t resolution_en);
```

设备地址解析使用，如果要使用 Resolvinglist 解析地址，一定要打开使能。不需要解析的时候，可以关闭。

```
ble_sts_t blc_ll_addDeviceToResolvingList(ida_type_t peerIdAddrType, u8 *peerIdAddr, u8  
*peer_irk, u8 *local_irk);
```



添加使用 RPA 地址的设备，peerIdAddrType/ peerIdAddr 和 peer\_irk 填 peer device 宣称的 identity address 和 irk，这些信息会在配对加密过程中存储到 Flash 中，user 可以在“[3.4.4 SMP](#)”找到获取这些信息的接口。对于 local\_irk, SDK 暂时没有处理，填 NULL 即可。

```
ble_sts_t    b1c_ll_removeDeviceFromResolvingList(ida_type_t peerIdAddrType, u8 *peerIdAddr);
```

删除之前添加的设备。返回值 BLE\_SUCCESS。

Whitelist/Resolvinglist 实现地址过滤的使用，请参考 feature\_test/feature\_whitelist 工程。

在 vendor/feature\_test/app\_config.h 中定义宏：

```
#define FEATURE_TEST_MODE TEST_WHITELIST
```

### 3.3 Host Controller Interface

HCI(Host Controller Interface) 是 Host 和 Controller 交换数据的桥梁，它定义了 Host 和 Controller 交互的各种数据类型，如：CMD，Event，ACL，SCO，ISO 等。HCI 使得蓝牙 Host 和 Controller 的实现在不同硬件平台上成为可能。HCI 具体内容可以查看 Bluetooth Core Specification V5.4, Vol 4: Host Controller Interface。

HCI Transport 是 HCI 的传输层，负责传输 HCI 各种数据类型的数据。HCI Transport 定义了 HCI 不同数据类型的 Type Indicator，如下图所示。

HCI packet type	HCI packet indicator
HCI Command Packet	0x01
HCI ACL Data Packet	0x02
HCI Synchronous Data Packet	0x03
HCI Event Packet	0x04

Figure 3.26: HCI Packet Type Indicator

#### 3.3.1 HCI 软件架构

tl\_ble\_sdk 中 HCI 软件架构如下图所示。HCI Transport 是 HCI Transport Layer 的软件实现，这部分源码完全向用户开放；Controller HCI 主要实现了 HCI CMD 和 HCI ACL 的解析和处理以及产生 HCI Event，同时为 HCI Transport 提供功能接口。本节将围绕 HCI Transport 和 Controller HCI 接口来详细描述 Telink HCI 的使用方法。

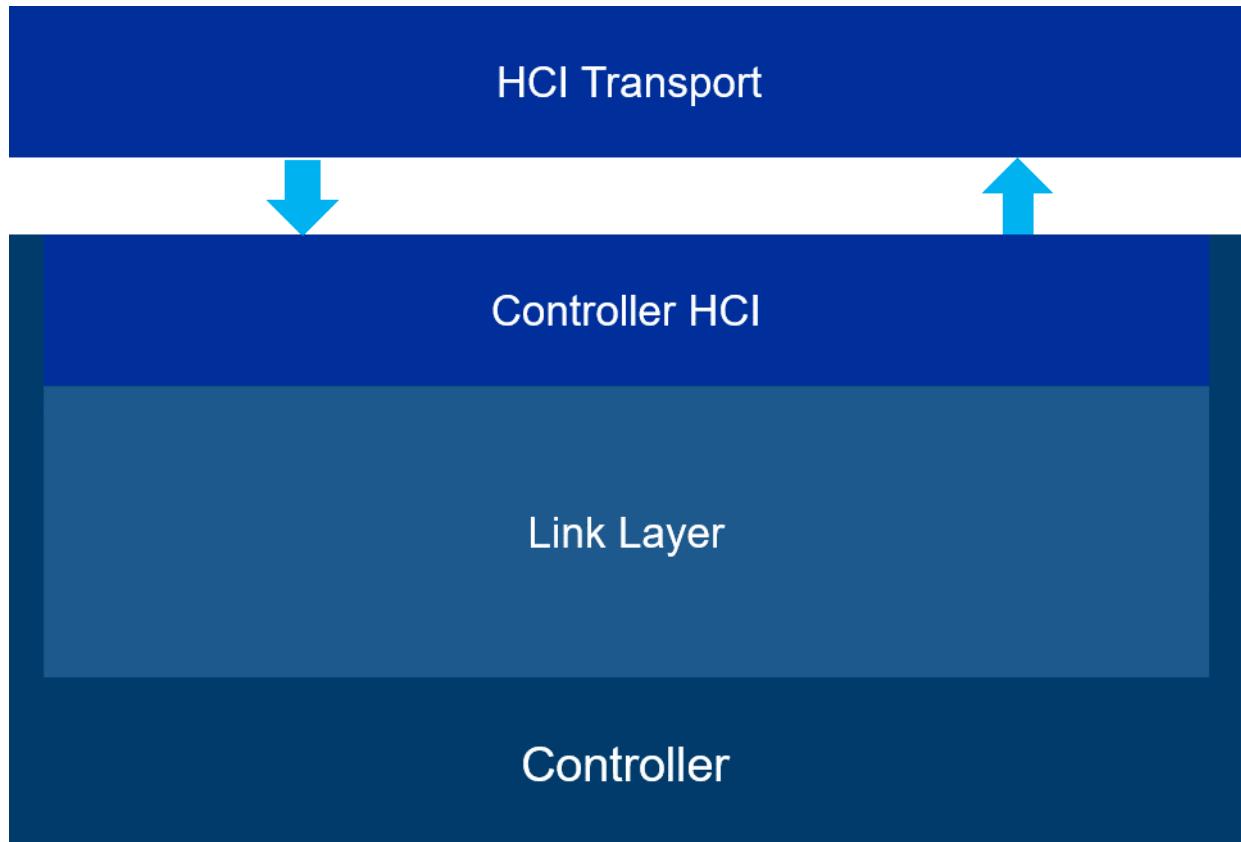


Figure 3.27: Telink HCI 软件架构

HCI Transport 用于传输 HCI 协议包，它不需要解析 HCI 协议包，只需要按照 HCI Type 接收 HCI 协议包，然后交给 Controller HCI 处理即可。HCl Transport 支持多种硬件接口，如：USB，UART，SDIO 等，其中常用的是 UART 接口，tl\_ble\_sdk 目前只提供了 UART 接口的 HCl Transport。Telink BLE HCl transport 的软件实现可以查看 SDK 中的 vendor/common/hci\_transport 文件夹。

HCl UART Transport 支持两种协议，H4 和 H5，这两种协议 tl\_ble\_sdk 都支持，且以源码的形式开放使用。Telink SDK HCl Transport 的软件架构如下图所示。

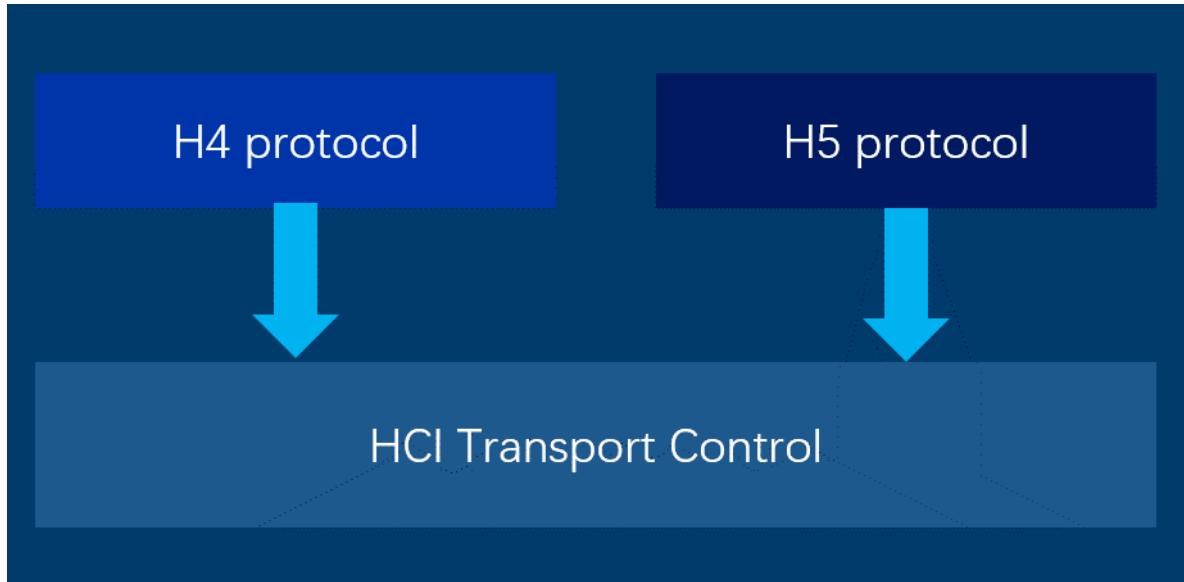


Figure 3.28: Telink HCI Transport 软件架构

H4 Protocol 是 HCI UART Transport H4 协议的软件实现；H5 Protocol 是 HCI UART Transport H5 协议的软件实现；HCI Transport Control 是 HCI Transport 的配置管理层，提供用户使用 HCI Transport 所需要的一切，因此，使用 HCI Transport 的用户只需要关注该层即可。

### 3.3.1.1 H4 Protocol

#### 3.3.1.1.1 H4 PDU

H4 PDU 格式如下图所示。

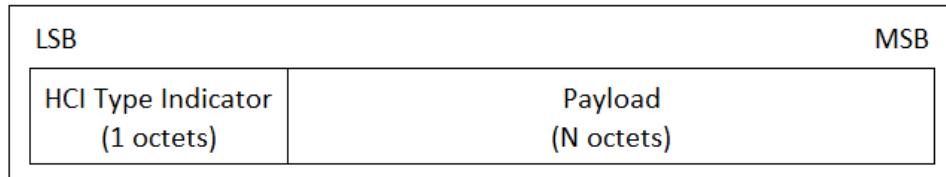


Figure 3.29: H4 PDU 格式

H4 PDU 由 HCI Type Indicator 和 Payload 组成。HCI Type Indicator 指明了 Payload 的内容，HCI Type Indicator 可取的值如下图所示；Payload 可以是 HCI CMD，HCI ACL 和 HCI Event 等 HCI Protocol 包。

Host 发送 H4 PDU 到 Controller，H4 Protocol 软件实现了 H4 PDU 的接收和解析，具体实现可以查看 vendor/common/hci\_transport 文件夹中的 hci\_tr\_h4.c 和 hci\_tr\_h4.h 文件。

用户在使用 H4 协议软件时需要根据需求配置 UART Rx Buffer 大小和 Buffer 个数。可以通过 hci\_tr\_h4.h 文件中的宏 HCI\_H4\_TR\_RX\_BUF\_SIZE 和 HCI\_H4\_TR\_RX\_BUF\_NUM 来配置。实际上，为了方便用户的使用，SDK 会自动计算 H4 UART Buffer Size。用户只需要配置 hci\_tr.h 文件中的宏 HCI\_TR\_RX\_BUF\_SIZE 即可。HCI\_H4\_TR\_RX\_BUF\_NUM 用户一般不需要修改，除非有特殊需求。



## 3.3.1.1.2 H4 API

H4 Protocol 提供了 3 个 API：

```
void HCI_Tr_H4Init(hci_fifo_t *pFifo);
void HCI_Tr_H4RxHandler(void);
void HCI_Tr_H4IRQHandler(void);
```

**void HCI\_Tr\_H4Init(hci\_fifo\_t \*pFifo)**

**功能：**该函是 H4 的初始化，包括 UART，RX Buffer 等。

**参数：**

参数	描述
pFifo	指向 Controller HCI Rx FIFO，用于存储接收并解析成功的 HCI 协议包供 Controller 处理。该参数由 Controller HCI 提供

**说明：**该函数用户一般不需要调用，它由 HCI Transport Control 层调用。

**void HCI\_Tr\_H4RxHandler(void)**

**功能：**该函数实现了 H4 PDU 的解析和处理功能。

**参数：**无。

**说明：**该函数用户一般不需要调用，它由 HCI Transport Control 层调用。

**void HCI\_Tr\_H4IRQHandler(void)**

**功能：**该函数实现了 UART RX/TX 中断处理。

**参数：**无。

**说明：**该函数用户一般不需要调用，它由 HCI Transport Control 层调用。

### 3.3.1.2 H5 Protocol

H5 又称 3wire UART。H5 支持软件流控和重传机制。相较 H4 来说，H5 具有更高的可靠性，但是其传输效率没有 H4 高。H5 具体内容参看 Bluetooth Core Specification V5.4, Vol 4, Part D: Three-wire UART Transport Layer。

H5 PDU(Protocol Data Unit) 在传输之前需要被编码，在解析 H5 PDU 之前需要解码，H5 PDU 的编解码由 Slip Layer 完成。

Telink H5 软件架构如所示。UART 用于数据的收发；Slip 层实现了 H5 PDU 的编码器和解码器，负责 H5 PDU 的编码和解码；H5 Handler 实现了 H5 PDU 的解析和处理、H5 Link 的创建以及流量控制和重传控制。用户可以在 vendor/common/hci\_transport 文件夹中的 hci\_tr\_h5.c, hci\_slip.c, hci\_h5.c 文件中查看 H5 的实现。

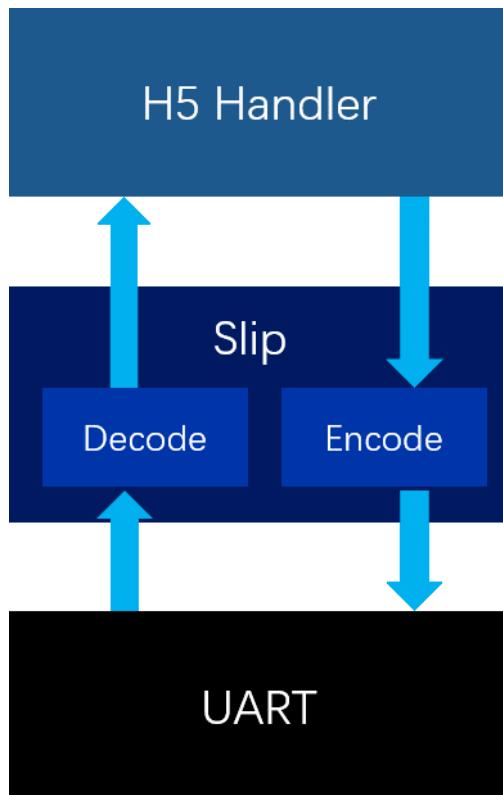


Figure 3.30: H5 软件架构

### 3.3.1.2.1 Slip Layer

#### (1) Slip 编码

Slip layer 编码时会在每个 H5 包的开始和结束的地方放置一个字节的 C0，H5 包中出现的所有 C0 都将被编码成 DB DC 序列；H5 包中出现的所有 DB 将被编码成 DB DD 序列；这里的 DB DC 和 DB DD 被称为 Slip 的转义序列，所有 Slip 的转义序列都始于 DB。Slip 的转义序列表如下图所示。

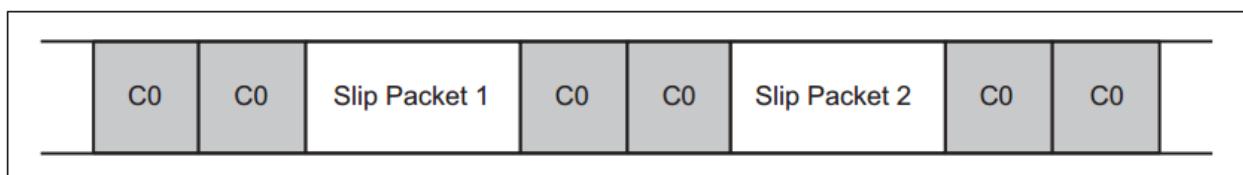


Figure 3.31: HCI Slip 包



SLIP Escape Sequence	Unencoded form	Notes
0xDB 0xDC	0xC0	
0xDB 0xDD	0xDB	
0xDB 0xDE	0x11	Only valid when OOF Software Flow Control is enabled
0xDB 0xDF	0x13	Only valid when OOF Software Flow Control is enabled

Figure 3.32: HCI Slip 转义序列

tl\_ble\_sdk 中 Slip 的编码是通过 API `HCI_Slip_EncodePacket(u8 *p, u32 len)` 来实现的。编码后的数据将存储在 Slip Encode buffer 中。Slip 的 Encode Buffer Size 可以通过宏 `HCI_SLIP_ENCODE_BUF_SIZE` 来设置，为了方便用户使用，SDK 已经对 `HCI_SLIP_ENCODE_BUF_SIZE` 实现了自动计算，不需要用户配置。用户只需要配置 vendor/common/hci\_transport/hci\_tr.h 文件中的宏 `HCI_TR_RX_BUF_SIZE` 即可。

## (2) Slip 解码

当收到 Slip 包以后，通过 Slip 包的起始和结束标志 CO 即可得到一个完整的 Slip 包。然后根据 Slip 的转义字节表将 DB DC 和 DB DD 序列等转化为 CO 和 DB，这样 Slip 就被解码了，接下来就是解析 H5 PDU。

tl\_ble\_sdk 中 Slip 的解码由 `void HCI_Slip_DecodePacket(u8 *p, 32 len)` 来实现，解码后的数据存储在 Slip Decode Buffer 中。Slip 解码 Buffer 的大小可以通过宏 `HCI_SLIP_DECODE_BUF_SIZE` 设置。为了方便用户使用，SDK 已经对 `HCI_SLIP_DECODE_BUF_SIZE` 实现了自动计算，不需要用户配置。用户只需要配置 vendor/common/hci\_transport/hci\_tr.h 文件中的宏 `HCI_TR_RX_BUF_SIZE` 即可。

### 3.3.1.2.2 H5 Handle

H5 Handler 实现了 H5 PDU 的解析和处理、H5 Link 创建以及流量控制和重传控制。

## (1) H5 PDU

H5 PDU(Protocol Data Unit)。H5 PDU 包含 Packet Header，Payload 和可选的 Data Integrity Check 3 个字段。

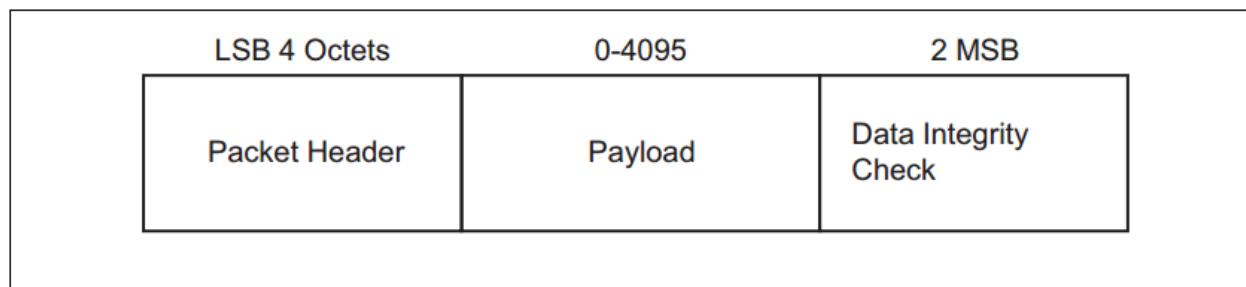


Figure 3.33: H5 PDU

H5 PDU Header 构成如下：

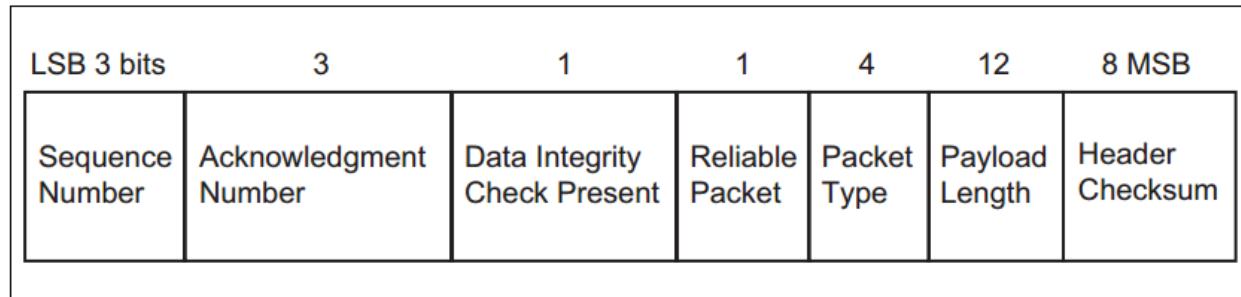


Figure 3.34: H5 PDU Header

**Sequence Number(SEQ):** 对于不可靠包来说，SEQ 应该设置为 0；对于可靠包来说，SEQ 表示包的序号。每收到一个新包，SEQ 应该加 1。SEQ 的范围 0-7。SEQ 不变，表示重传。

**Acknowledgment Number(ACK):** ACK 应该设置为设备期望的下一个包序号，范围 0-7。

**Data Integrity Check Present:** 设置为 1，表示需要对 PDU 的 Payload 字段进行 CRC 校验，也即 PDU 中的 Data Integrity Check 存在，否则不存在

**Reliable Packet:** 设置为 1，使用可靠传输，SEQ 将生效。

**Packet Type:** H5 定义了 8 种包类型：

HCI Packet Type	Packet Type
Acknowledgment packets	0
HCI Command packet	1
HCI ACL Data packet	2
HCI Synchronous Data packet	3
HCI Event packet	4
HCI ISO Data packet	5
Vendor Specific	14
Link Control packet	15
Reserved for future use	All other values

Figure 3.35: H5 Packet Type

**Payload length:** PDU 中 payload 字段的长度。

**Header CheckSum:** H5 PDU Header 的和校验值。

H5 Handler 通过函数 void HCI\_H5\_PacketHandler(u8 \*p, u32 len) 来实现对 H5 PDU 的解析和处理，这个函数是 H5 内部使用的函数，用户不需要调用。

## (2) H5 Link 建立和配置信息交换



Host 和 Controller 交换 H5 数据包之前需要先建立 H5 连接并协商配置信息。H5 link 建立需要使用 SYNC message、SYNC\_RSP message、CONFIG message 和 CONFIG\_RSP message。

初始时，H5 处于 Uninitialized 状态，并且不断发送 SYNC message。当收到 SYNC\_RSP message 以后进入 Initialized 状态，并不断发送 CONFIG message；当收到 CONFIG\_RSP message 以后，H5 进入 Active 状态，此时 H5 link 建立成功，可以收发数据。

Telink H5 初始时以 250 ms 的间隔发送 SYNC message，当收到 CONFIG\_RSP message 以后，进入 Initialized 状态，并以 250 ms 的间隔发送 CONFIG message，当收到 CONFIG\_RSP message 以后，进入连接态。

在 CONFIG message 和 CONFIG\_RSP message 中包含 Host 和 Controller 使用的连接参数，双方取共同部分作为最终的连接参数。H5 连接的配置信息主要包括 Sliding Window Size、OOF Flow Control、Data Integrity Check Type 和 Version。

**Sliding Window Size：**设置不需要立即 ACK 的最大数据包数。当 Sliding Window Size = 1 时，意味着 Controller 发送一包以后，必须等到 Host ACK 以后才能传输下一个包；当 Sliding Window Size = N (N>1) 时，Controller 可以发送 N 包而不需要等待 Host 的 ACK，但是 Controller 发送第 N 包以后，必须等待 Host ACK 才能发送其他数据包。Sliding Window Size 目的是提高 H5 传输效率。目前 Telink SDK 只支持 Sliding Window = 1 的情况，Sliding Window > 1 的情况也很容易扩展。

**OOF Flow Control：**使能软件流控，这个不常用，故不做详述。

**Data Integrity Check Type：**设置为 1，则 H5 PDU 中与 Data Integrity Check 相关的字段都将启用。

**Version：**设置 H5 (3 wire UART) 版本。目前是 v1.0 版本。

tl\_ble\_sdk 中，用户可以通过如下宏来配置 H5 连接参数。

```
#define HCI_H5_SLIDING_WIN_SIZE           1
#define HCI_H5_OOF_FLW_CTRL                HCI_H5_OOF_FLW_CTRL_NONE
#define HCI_H5_DATA_INTEGRITY_LEVEL         HCI_H5_DATA_INTEGRITY_LEVEL_NONE
#define HCI_H5_VERSION                     HCI_H5_VERSION_V1_0
```

### (3) H5 数据交互以及重传

Host 和 Controller 建立 H5 连接以后双方就可以交互数据包了。H5 支持流控和重传机制，流控和重传是通过 H5 PDU Header 中的 SEQ 和 ACK 字段来实现的。下图是 H5 数据交互以及重传的例子。

设备 A 发送 SEQ 为 6，ACK 为 3 的包给设备 B，SEQ 为 6 表示设备 B 期望的包序号是 6，ACK 为 3 表示设备 A 期望的下一个包序号是 3。设备 B 收到设备 A 的包以后，发送 SEQ 为 3，ACK 为 7 的包给设备 A。设备 B 的 SEQ 为 3，是因为设备 A 期望的包是 3；设备 B 的 ACK 为 7，是因为设备 B 收到了序号为 6 的包而期望新包 7，依次类推。

设备 A 发送 SEQ 为 0，ACK 为 5 的包给设备 B，但是由于一些原因设备 B 没有收到这个包，设备 B 由于没有收到设备 A 的包，一段时间以后会重传上一个包。

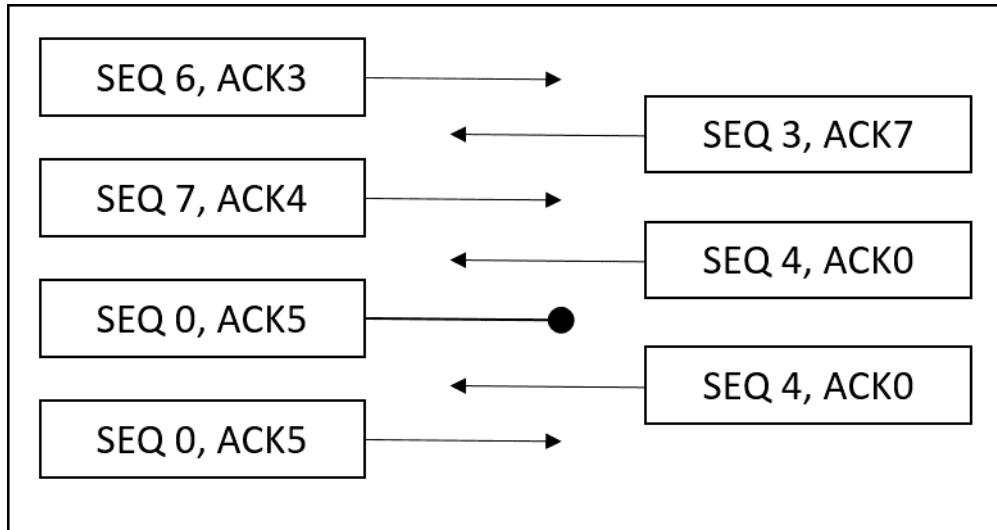


Figure 3.36: H5 Data flow

### 流控:

Host 和 Controller 建立 H5 连接以后，主要交互 Data 包和 pure ACK 包。Pure ACK 包是 H5 Packet Type 为 0 的包。正常情况下，对端设备发送一个 Data 包，本地设备回复一个 Data 包，然后持续进行下去，但是对端和本地设备总会有没有 Data 包可发送的时候，这个时候设备可以发送 pure ACK 包来代替 Data 包，例如：host 使能了 scan，controller 会不断的上报 adv report，controller 会有源源不断的 Data 包，但是 Host 却没有大量的 Data 包可发送，如果 controller 发送 data 包到 host，却收不到 host 的回复，那么就会导致 controller 一直重传，host 将再也收不到新的 adv report，此时 host 需要发送 pure ACK 包来代替 Data 包，相当于回复 controller。这样 host 会不断的收到新的 adv report。

### 重传:

重传有多种情况：本地设备发送 data 包以后，如果在 timeout 到达之前还没收到对端设备的回复 (data 包或者 pure ACK 包)，本地设备会 resend；如果收到对方的 data 包或者 pure ACK 包中的 ACK 值指示需要本地重传的，本地设备将 resend。重传时 SEQ 应该保持不变。

#### 3.3.1.2.3 H5 相关 API

H5 有两个重要的 API：

```
void HCI_H5_Init(hci_fifo_t *pHciRxFifo, hci_fifo_t *pHciTxFifo);
void HCI_H5_Poll(void);
```

##### void HCI\_H5\_Init(hci\_fifo\_t \*pHciRxFifo, hci\_fifo\_t \*pHciTxFifo)

**功能：**该函数用于初始化 H5。

**参数：**

参数	描述
pHciRxFifo	指向 Controller HCI Rx FIFO



参数	描述
pHciTxFifo	指向 Controller HCI Tx FIFO，H5 将接管 HCI Tx FIFO，不需要用户管理，降低了使用难度。 说明：该函数用户一般不需要调用，它由 HCI Transport Control 层调用。

**说明：**该函数用户一般不需要调用，它由 HCI Transport Control 层调用。

#### **void HCI\_H5\_Poll(void)**

**功能：**该函数用于管理 H5 包的解析、发送、resend 和流控。

**参数：**无。

**说明：**该函数用户一般不需要调用，它由 HCI Transport Control 层调用。

### **3.3.1.3 HCI Transport Control**

HCI Transport Control 是 Telink HCI Transport 的集中管理层，它是连接 HCI Transport 和 Controller HCI 的桥梁，同时，它也为用户提供了配置和使用 HCI Transport 所需的宏和 API。对于使用 Telink Controller 工程的用户来说，只需要关注 HCI Transport Control 层即可。HCI Transport Control 提供的宏和 API 可以在 Controller 工程中的 hci\_tr.h 里找到。

#### **3.3.1.3.1 HCI Transport 配置**

HCI Transport Control 提供了一系列的配置宏，下面详细描述。

用户可以通过如下的宏来选择使用的 transport 协议。tl\_ble\_sdk 默认使用 HCI\_TR\_H4。

```
/*! HCI transport layer protocol selection. */
#define HCI_TR_H4          0
#define HCI_TR_H5          1
#define HCI_TR_USB          2 /*!< Not currently supported */
#define HCI_TR_MODE          HCI_TR_H4
```

用户可以通过如下宏设置 Transport Rx Patch 和 Tx Path 上 UART buffer 最大 Size。HCI\_TR\_RX\_BUF\_SIZE 应该设置为最大可能接收到的 HCI 包的大小；HCI\_TR\_TX\_BUF\_SIZE 应该设置为最大可能发送的 HCI 包大小，这对于 H4 和 H5 都适用。例如：最大 Rx HCI ACL 为 27B，最大 Rx HCI Cmd Payload 为 65B，那么 HCI\_TR\_RX\_BUF\_SIZE 应该设置为 1B(HCI Type length) + 4B (HCI ACL Header Length) + MAX(27, 65) = 70B，HCI\_TR\_TX\_BUF\_SIZE 计算方法一样。

```
#define HCI_TR_RX_BUF_SIZE    HCI_RX_FIFO_SIZE
#define HCI_TR_TX_BUF_SIZE    HCI_TX_FIFO_SIZE
/*! HCI UART transport pin define */
#define HCI_TR_RX_PIN          GPIO_PB0
#define HCI_TR_TX_PIN          GPIO_PA2
#define HCI_TR_BAUDRATE        (1000000)
```



HCI\_TR\_RX\_PIN、HCI\_TR\_TX\_PIN 用于设置 UART Tx/Rx 引脚。HCI\_TR\_BAUDRATE 用于设置 UART 波特率。

关于 UART 波特率选择需要注意：由于 BLE 可以采用 1M 和 2M，因此 UART 波特率应该做相应的匹配，否则当传输 ACL Data 量很大时可能会出现 buffer 不够的情形；另外，当波特率很低时，调大 buffer 和 buffer 个数将会消耗较大 RAM，因此需要做好波特率匹配。我们推荐当使用的 BLE 速率是 1M 时，UART 波特率选择最好大于等于 1M；当使用的 BLE 速率是 2M 时，UART 波特率选择最好大于等于 2M。

### 3.3.1.3.2 HCI Transport API

为了方便用户使用，HCl Transport 最终给用户留了必要的 API，实际使用时用户只需要调用这些 API 即可。

```
void HCI_TransportInit(void);
void HCI_TransportPoll(void);
void HCI_TransportIRQHandler(void);
```

#### void HCI\_TransportInit(void)

**功能：**该函数是各种 Transport 协议初始化的封装。用户使用 HCl Transport 功能之前需要通过该函数初始化 HCl Transport。

**参数：**无

#### void HCI\_TransportPoll(void)

**功能：**该函数是对各种 Transport 协议任务处理器的封装，用户需要在 main loop 中调用。

**参数：**无

#### void HCI\_TransportIRQHandler(void)

**功能：**该函数是对各种 Transport 协议使用中断的封装。用户需要在中断中调用该 API。

**参数：**无

### 3.3.1.4 Controller HCl

Controller HCl 接口实现了 HCl 协议包的解析和处理以及产生 HCl Event。HCl 具体内容参看 Bluetooth Core Specification V5.4 Vol 4。本节将讲解几个重要的 API。

Controller HCl 提供必要的 API 供用户使用。

```
ble_sts_t blc_ll_initHciRxFifo(u8 *pRxbuf, int fifo_size, int fifo_number);
ble_sts_t blc_ll_initHciTxFifo(u8 *pTxbuf, int fifo_size, int fifo_number);
ble_sts_t blc_ll_initHciAclDataFifo(u8 *pAclbuf, int fifo_size, int fifo_number);
int blc_hci_handler(u8 *p, int n);
```

#### blc\_ll\_initHciRxFifo(u8 \*pRxbuf, int fifo\_size, int fifo\_number)



**功能：**该函数用于初始化 HCI Rx FIFO。HCI Rx FIFO 可以管理多组 Rx buffer。HCI Rx FIFO 用于存储接收到的 HCI 包。HCI Rx Buffer 需要用户在应用层定义并通过本函数注册。controller 工程中已经定义好了 HCI Rx Buffer，用户可以参看工程中的 app\_buffer.c 和 app\_buffer.h 文件。

**参数：**

参数	说明
pRxbuf	指向 Rx buffer
fifo_size	Rx FIFO 中每个 buffer 的 size，必须是 16 字节对齐
fifo_number	Rx FIFO 中 buffer 的个数，必须是 2 的指数幂

**说明：**用户需要在初始化时调用

#### **bIc\_ll\_initHciTxFifo(u8 \*pTxbuf, int fifo\_size, int fifo\_number)**

**功能：**该函数用于初始化 HCI Tx FIFO。HCI Tx FIFO 可以管理多组 Tx buffer。HCI Tx FIFO 用于存储 Controller 将要发送给 Host 的 HCI Evt 和 HCI ACL。HCI Tx Buffer 需要用户在应用层定义并通过本函数注册。controller 工程中已经定义好了 HCI Tx Buffer，用户可以参看工程中的 app\_buffer.c 和 app\_buffer.h 文件。

**参数：**

参数	描述
pTxbuf	指向 Tx buffer
fifo_size	Tx FIFO 中每个 buffer 的 size，必须是 4 字节对齐
fifo_number	Tx FIFO 中 buffer 的个数，必须是 2 的指数幂

**说明：**用户需要在初始化时调用

#### **bIc\_ll\_initHciAclDataFifo(u8 \*pAclbuf, int fifo\_size, int fifo\_number)**

**功能：**该函数用于初始化 HCI ACL FIFO。HCI ACL FIFO 可以管理多组 ACL buffer。HCI ACL FIFO 用于存储 Host 下发给 Controller 的 ACL Data。HCI ACL Buffer 需要用户在应用层定义并通过本函数注册。controller 工程中已经定义好了 HCI ACL Buffer，用户可以参看工程中的 app\_buffer.c 和 app\_buffer.h 文件。

**参数：**

参数	说明
pAclbuf	指向 Acl buffer
fifo_size	Acl FIFO 中每个 buffer 的 size，必须是 4 字节对齐
fifo_number	Acl FIFO 中 buffer 的个数，必须是 2 的指数幂



说明：用户需要在初始化时调用

```
int blc_hci_handler(u8 *p, int n)
```

功能：该函数是 Controller HCI 包处理器，实现了 HCI CMD 和 ACL 包的解析和处理。

参数：

参数	说明
p	指向接收到的 HCI 协议包 (使用 H4 PDU 格式)
n	未使用，因为 HCI 协议包中包含的 length 信息

说明：该函数被 HCI Transport Control 层调用，一般情况下不需要用户调用。

### 3.3.2 Controller Event

为了满足 user 在应用层对 multiple connection BLE stack 底层一些关键动作的记录和处理，SDK 提供了两种类型的 event 如下图所示：一是 BLE controller 定义的标准的 HCI event；二是 BLE host 定义的一些协议栈流程交互的事件通知型 GAP event（也可以认为是 host event，这部分具体介绍请参考本文档“GAP event”章节）。本小节主要介绍 Controller event。

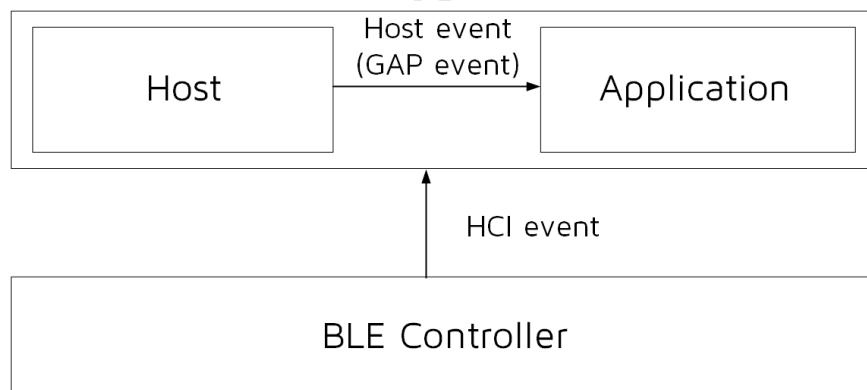


Figure 3.37: BLE SDK Event 架构

注意：

在 Telink B91 BLE Single Connection SDK 上，Telink 提供了自己定义的一套 controller event，和 Bluetooth Core Specification 规定的 HCI event 大部分是一样的，在 tl\_ble\_sdk 中，就去掉了重复部分中 Telink 定义的 event，user 使用标准的 event 即可。

#### 3.3.2.1 Controller HCI Event 分类

Controller HCI event 是按 Bluetooth Core Specification 标准设计的。



如下图 Host + Controller 架构所示，Controller HCI event 是通过 HCI 将 Controller 所有的 event 报告给 Host。

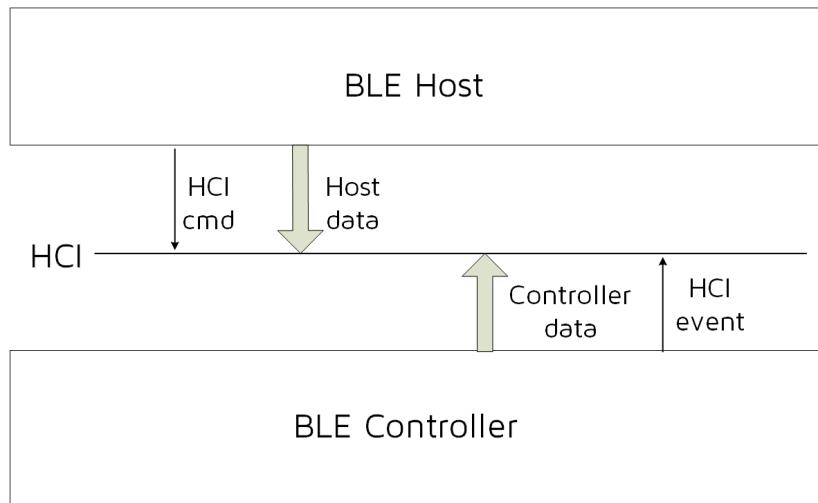


Figure 3.38: Host + Controller 架构

Controller HCI event 的定义，详情请参考 Bluetooth Core Specification V5.4, Vol 4, Part E, 7.7 Events。其中 7.7.65 LE Meta event 指 HCI LE(low energy) event，其他的都是普通的 HCI event。tl\_ble\_sdk 也将 Controller HCI event 分为两类：HCI event 和 HCI LE event。由于 tl\_ble\_sdk 主打低功耗蓝牙，所以对 HCI event 只支持了最基本的几个，而 HCI LE event 绝大多数都支持。

Controller HCI event 相关的宏定义、接口定义等请参考 stack/ble/hci 目录下的头文件。

如果 user 需要在 Host 或 App 层接收 Controller HCI event，首先需要注册 Controller HCI event 的 callback 函数，其次需要将对应 event 的 mask 打开，mask 打开 API 见下面 event 分析。

Controller HCI event 的 callback 函数原型和注册接口分别为：

```
typedef int (*hci_event_handler_t) (u32 h, u8 *para, int n);
void blc_hci_registerControllerEventHandler(hci_event_handler_t handler);
```

callback 函数原型中的 u32 h 是一个标记，底层协议栈多处会用到，user 只需要知道下面一个即可：

```
#define HCI_FLAG_EVENT_BT_STD (1<<25)
```

HCI\_FLAG\_EVENT\_BT\_STD 这个标志表示当前 event 为 Controller HCI event。

Callback 函数原型中 para 和 n 表示 event 的数据和数据长度，该数据和 Bluetooth Core Specification 中定义的一致。用户可参考代码中的如下用法以及 app\_controller\_event\_callback 函数的具体实现。

```
blc_hci_registerControllerEventHandler(app_controller_event_callback);
```

### 3.3.2.2 常用 Controller HCI event

tl\_ble\_sdk 中支持了绝大多数 HCI event，下面介绍客户可能会用到的 event。



```
#define HCI_EVT_DISCONNECTION_COMPLETE          0x05  
#define HCI_EVT_LE_META                         0x3E
```

#### (1) HCI\_EVT\_DISCONNECTION\_COMPLETE

详情请参考 Bluetooth Core Specification V5.4, Vol 4, Part E, 7.7.5 Disconnection Complete event。回调指针指向的数据结构如下：

```
typedef struct {  
    u8      status;  
    u16     connHandle;  
    u8      reason;  
} hci_disconnectionCompleteEvt_t;
```

#### (2) HCI\_EVT\_LE\_META

表示当前是 HCI LE event，根据后面的 sub event code 判断具体的 event 类型。

HCI event 中除了 HCI\_EVT\_LE\_META (HCI\_EVT\_LE\_META 使用 blc\_hci\_le\_setEventMask\_cmd 来打开 event mask)，其他都要通过下面 API 来打开 event mask。

```
ble_sts_t blc_hci_setEventMask_cmd(u32 evtMask); //eventMask: BT/EDR
```

event mask 的定义如下所示：

```
#define HCI_EVT_MASK_DISCONNECTION_COMPLETE      0x000000000010
```

若 user 未通过该 API 设置 HCI event mask，SDK 默认只打开 HCI\_EVT\_MASK\_DISCONNECTION\_COMPLETE 对应的 mask，即保证 Controller disconnect event 的上报。

### 3.3.2.3 常用 HCI LE event

当 HCI event 中 event code 为 HCI\_EVT\_LE\_META，就是 HCI LE event，subevent code 最常用的且 user 可能需要了解的如下，其他的不做介绍。

```
#define HCI_SUB_EVT_LE_CONNECTION_COMPLETE        0x01  
#define HCI_SUB_EVT_LE_ADVERTISING_REPORT        0x02  
#define HCI_SUB_EVT_LE_CONNECTION_UPDATE_COMPLETE 0x03  
#define HCI_SUB_EVT_LE_PHY_UPDATE_COMPLETE        0x0C
```

#### (1) HCI\_SUB\_EVT\_LE\_CONNECTION\_COMPLETE

详情请参考 Bluetooth Core Specification V5.4, Vol 4, Part E, 7.7.65.1 LE Connection Complete event。回调指针指向的数据结构如下：



```
typedef struct {
    u8      subEventCode;
    u8      status;
    u16     connHandle;
    u8      role;
    u8      peerAddrType;
    u8      peerAddr[6];
    u16     connInterval;
    u16     PeripheralLatency;
    u16     supervisionTimeout;
    u8      CentralClkAccuracy;
} hci_le_connectionCompleteEvt_t;
```

## (2) HCI\_SUB\_EVT\_LE\_ADVERTISING\_REPORT

详情请参考 Bluetooth Core Specification V5.4, Vol 4, Part E, 7.7.65.2 LE Advertising Report event。回调指针指向的数据结构如下：

```
typedef struct {
    u8  subcode;
    u8  nreport;
    u8  event_type;
    u8  adr_type;
    u8  mac[6];
    u8  len;
    u8  data[1];
} event_adv_report_t;
```

当 controller 的 Link Layer scan 到正确的 adv packet 后，通过 HCI\_SUB\_EVT\_LE\_ADVERTISING\_REPORT 上报给 Host。

该 event 的数据长度不定 (取决于 adv packet 的 payload)，如下所示，具体数据含义请直接参考 Bluetooth Core Specification。

0x04		0x3e		0x02		
hci event	event code	param len	subevent code	num report	event type	address type[1...i]
address[1...i]					length[1..i]	
data[1...i]					rss[i][1..i]	

Figure 3.39: ADVERTISING\_REPORT 事件包格式

注意：



tl\_ble\_sdk 中的 LE Advertising Report Event 每次只报一个 adv packet, 即上图中的 i 为 1。

### (3) HCI\_SUB\_EVT\_LE\_CONNECTION\_UPDATE\_COMPLETE

详情请参考 Bluetooth Core Specification V5.4, Vol 4, Part E, 7.7.65.3 LE Connection Update Complete event。

当 Controller 上的 connection update 生效时, 向 Host 上报 HCI\_SUB\_EVT\_LE\_CONNECTION\_UPDATE\_COMPLETE。回调指针指向的数据结构如下:

```
typedef struct {
    u8      subEventCode;
    u8      status;
    u16     connHandle;
    u16     connInterval;
    u16     connLatency;
    u16     supervisionTimeout;
} hci_le_connectionUpdateCompleteEvt_t;
```

### (4) HCI\_SUB\_EVT\_LE\_PHY\_UPDATE\_COMPLETE

详情请参考 Bluetooth Core Specification V5.4, Vol 4, Part E, 7.7.65.12 LE PHY Update Complete event。

回调指针指向的数据结构如下:

```
typedef struct {
    u8      subEventCode;
    u8      status;
    u16     connHandle;
    u8      tx_phy;
    u8      rx_phy;
} hci_le_phyUpdateCompleteEvt_t;
```

HCI LE event 需要通过下面的 API 来打开 mask。

```
ble_sts_t blc_hci_le_setEventMask_cmd(u32 evtMask); //eventMask: LE
```

evtMask 的定义也对应上面给出一些, 其他的 event 用户可以在 hci\_const.h 中查到。

```
#define HCI_LE_EVT_MASK_CONNECTION_COMPLETE      0x00000001
#define HCI_LE_EVT_MASK_ADVERTISING_REPORT       0x00000002
#define HCI_LE_EVT_MASK_CONNECTION_UPDATE_COMPLETE 0x00000004
```

若 user 未通过该 API 设置 HCI LE event mask, SDK 默认所有 HCI LE event 都不打开。



## 3.4 Host

### 3.4.1 L2CAP

逻辑链路控制与适配协议通常简称为 L2CAP (Logical Link Control and Adaptation Protocol)，它向上连接应用层，向下连接控制器层，发挥主机与控制器之间的适配器的作用，使上层应用操作无需关心控制器的数据处理细节。

BLE 的 L2CAP 层是经典蓝牙 L2CAP 层的简化版本，它在基础模式下，不执行分段和重组，不涉及流程控制和重传机制，仅使用固定信道进行通信。L2CAP 的简化结构如下图所示，简单说就是将应用层数据分包发给 BLE controller，将 BLE controller 收到的数据打包成不同 CID 数据上报给 host 层。

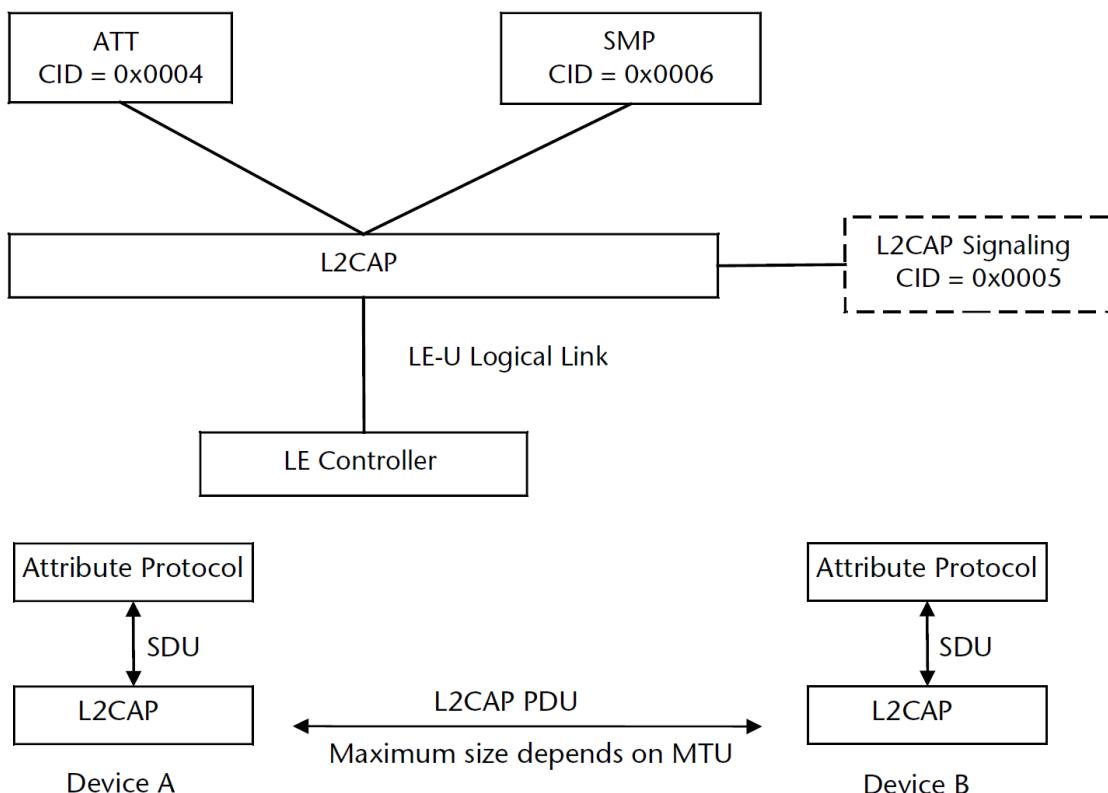


Figure 3.40: BLE L2CAP 结构以及 ATT 组包模型

L2CAP 根据 Bluetooth Core Specification 设计，主要功能是完成 Controller 和 Host 的数据对接，绝大部分都在协议栈底层完成，需要 user 参与的地方很少。user 根据以下几个 API 进行设置即可。

#### 3.4.1.1 注册 L2CAP 数据处理函数

tl\_ble\_sdk 架构中，Controller 的数据通过 HCI 与 Host 对接，从 HCI 到 Host 数据，首先会在 L2CAP 层处理，使用下面 API 注册该处理函数：

```
void blc_hci_registerControllerDataHandler (hci_data_handler_t handle);
```

L2CAP 层处理 Controller 数据的函数为：



```
int     blc_l2cap_pktHandler (u16 connHandle, u8 *raw_pkt);
```

该函数已经在协议栈中实现，它会对接收到的数据进行解析后向上传输给 ATT、SIG 或 SMP。

初始化：

```
blc_hci_registerControllerDataHandler (blt_l2cap_pktHandler);
```

### 3.4.1.2 更新连接参数

(1) Peripheral 请求更新连接参数

在 BLE 协议栈中，Peripheral 通过 L2CAP 层 CONNECTION PARAMETER UPDATE REQUEST 命令向 Central 申请一组新的连接参数。该命令格式如下所示，详情请参照 Bluetooth Core Specification V5.4, Vol 3, Part A, 4.20 L2CAP\_CONNECTION\_PARAMETER\_UPDATE\_REQ (code 0x12)。

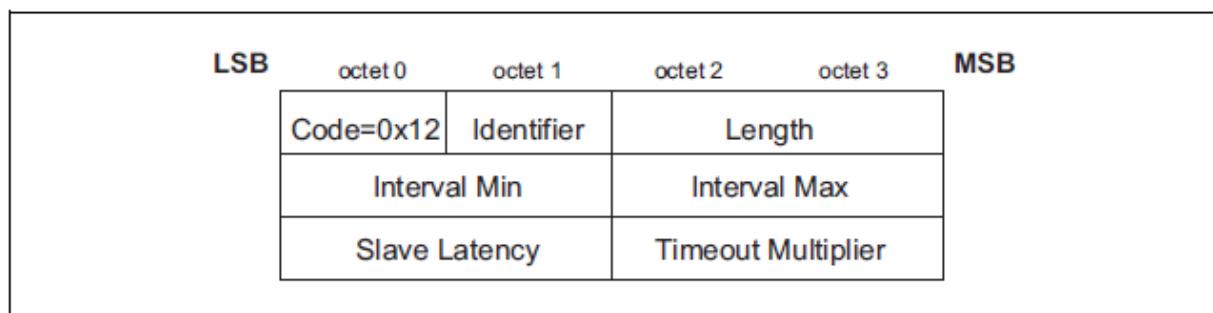


Figure 4.22: Connection Parameters Update Request Packet

Figure 3.41: BLE 协议栈中 Connection Para update Req 格式

tl\_ble\_sdk 提供了 Peripheral 主动申请更新连接参数的 API，用来向 Central 发送 CONNECTION PARAMETER UPDATE REQUEST 命令。

```
void  bls_l2cap_requestConnParamUpdate (u16 connHandle, u16 min_interval, u16 max_interval, u16
→ latency, u16 timeout);
```

该 API 仅限 Peripheral 使用。min\_interval 和 max\_interval 的单位为 1.25 ms，timeout 的单位为 10 ms。

tl\_ble\_sdk 提供了 Peripheral 设置发送更新连接参数请求时间的 API：

```
void bls_l2cap_setMinimalUpdateReqSendingTime_after_connCreate( u16 connHandle, int time_ms )
```

以连接建立时刻为时间基准点，经过 time\_ms 后才会将连接参数更新请求发送出去，不调用该 API，默认设置为 1000 ms。

如果调用 API bls\_l2cap\_requestConnParamUpdate 的时间已经在建立连接后的 time\_ms 之后，则立刻发出连接参数跟新请求。



tus	Data Type	Data Header	L2CAP Header	SIG Pkt Header	SIG_Connection_Param_Update_Req	CRC
L2CAP-S	LLID NESN SN MD PDU-Length	0x000C 0x0005	Code Id Data-Length	IntervalMin IntervalMax SlaveLatency TimeoutMultiplier	0x0006 0x0006 0x0063 0x0190	0x28D8
tus	Data Type	Data Header	L2CAP Header	SIG Pkt Header	SIG_Connection_Param_Update_Rsp	CRC RSSI FCS
L2CAP-S	LLID NESN SN MD PDU-Length	0x0006 0x0005	Code Id Data-Length	Result	0x0000	0x2DE483 -38 OK
tic	Data Type	Data Header	CRC	RSSI	FCS	

Figure 3.42: 抓包显示 conn para update request 和 response

在应用中，SDK 提供了获取连接请求结果的 GAP Event “GAP\_EVT\_L2CAP\_CONN\_PARAM\_UPDATE\_RSP”，用于通知用户 Peripheral 申请的连接参数请求是否被接受，如上图所示，Central 接受了 Peripheral 的 Connection\_Param\_Update\_Req 参数。

其中 app\_host\_event\_callback 函数参考如下：

```
int app_host_event_callback(u32 h, u8 *para, int n)
{
    u8 event = h & 0xFF;
    switch(event){
        .....
        case GAP_EVT_L2CAP_CONN_PARAM_UPDATE_RSP:
        {
            (gap_l2cap_connParamUpdateRspEvt_t*) p= (gap_l2cap_connParamUpdateRspEvt_t*) para;
            if( p->result == CONN_PARAM_UPDATE_ACCEPT ){
                //the LE Central Host has accepted the connection parameters
            }
            else if( p->result == CONN_PARAM_UPDATE_REJECT ){
                //the LE Central Host has rejected the connection parameter
            }
        }
        Break;
        .....
    }
    return 0;
}
```

## (2) Central 回应更新申请

peer Peripheral 申请新的连接参数后，Central 收到该命令，回 CONNECTION PARAMETER UPDATE RESPONSE 命令，详情请参照 Bluetooth Core Specification V5.4, Vol 3, Part A, 4.21 L2CAP\_CONNECTION\_PARAMETER\_UPDATE\_RSP (code 0x13)。

关于实际的 Android、iOS 设备是否接受 user 所申请的连接参数，跟各个厂家 BLE Central 的做法有关，各家标准并不统一。

tl\_ble\_sdk 中，不管是否接受 Peripheral 的参数申请，都使用下面 API 对该申请进行答复：

```
void blc_l2cap_SendConnParamUpdateResponse(u16 connHandle, u8 req_id, conn_para_up_rsp
    ↵ result);
```

该 API 仅限 Central 使用。connHandle 指定当前 connection handle，req->id 是连接参数更新请求中的 Identifier 值，connParaRsp 参考如下：



```
typedef enum{
    CONN_PARAM_UPDATE_ACCEPT = 0x0000,
    CONN_PARAM_UPDATE_REJECT = 0x0001,
}conn_para_up_rsp;
```

tl\_ble\_sdk 中, Central 在判断合适的连接参数请求后, 如果用户注册了 GAP\_EVT\_L2CAP\_CONN\_PARAM\_UPDATE\_REQ, 判断是否同意连接参数更新由用户在事件回调里执行, 如果没有注册 Central 会在底层直接进入连接参数更新流程。

当 GAP\_EVT\_L2CAP\_CONN\_PARAM\_UPDATE\_REQ 生效情况下, 如果用户不同意连接参数请求, 需要在回调里调用 bIc\_l2cap\_SendConnParamUpdateResponse, 并且第三个参数设置为 CONN\_PARAM\_UPDATE\_REJECT。如果用户同意连接参数请求, 需要在回调里先调用 bIc\_l2cap\_SendConnParamUpdateResponse, 并且第三个参数设置为 CONN\_PARAM\_UPDATE\_ACCEPT, 然后再调用 bIm\_l2cap\_processConnParamUpdatePending 进入连接参数更新流程。

```
void bIm_l2cap_processConnParamUpdatePending(u16 connHandle, u16 min_interval, u16 max_interval,
    u16 latency, u16 timeout);
```

### (3) 在 Link Layer 上更新连接参数

Central 可以直接执行连接参数更新, 也可以由 Peripheral 发送 conn para update req, 并且 Central 回 conn para update rsp 接受申请后, 就会有下面的流程。

Central 会发送 link layer 层的 LL\_CONNECTION\_UPDATE\_REQ 命令, 如下图所示。

is	Data Type	Data Header				LL_Opcode		LL_Connect_Update_Req					
	Control	LLID	NESN	SN	MD	PDU-Length		WinSize	WinOffset	Interval	Latency	Timeout	Instant
		3	1	1	0	12	Connection_Update_Req(0x00)	0x02	0x001F	0x0006	0x0063	0x0190	0x006C
is	Data Type	Data Header				CRC	RSSI (dBm)	FCS					
	Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x8FE90F	0	OK				

Figure 3.43: 抓包显示 II conn update req

Peripheral 收到更新请求后, 更新连接参数。Central 和 Peripheral 都会触发 HCI\_SUB\_EVT\_LE\_CONNECTION\_UPDATE\_COMPLETE 的 HCI 事件。

## 3.4.2 ATT & GATT

### 3.4.2.1 GATT 基本单位 Attribute

GATT 定义了两种角色: Server 和 Client。tl\_ble\_sdk 中, Peripheral 设备是 Server, Android、iOS 或 Central 设备是 Client。Server 需要提供多个 service 供 Client 访问。

GATT 的 service 实质是由多个 Attribute 构成, 每个 Attribute 都具有一定的信息量, 当多个不同种类的 Attribute 组合在一起时, 就能够反映出一个基本的 service。

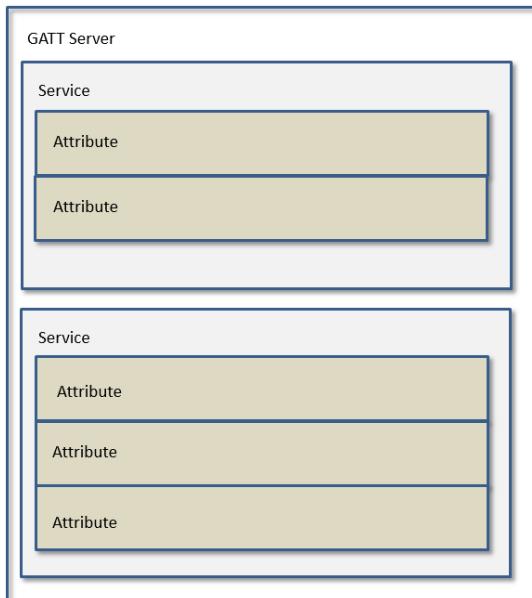


Figure 3.44: Attribute 构成 GATT service

一个 Attribute 的基本内容和属性包括以下：

(1) Attribute Type: UUID

UUID 用来区分每一个 attribute 的类型，其全长为 16 个 bytes。BLE 标准协议中 UUID 长度定义为 2 个 bytes，这是因为 peer device 设备都遵循同一套转换方法，将 2 个 bytes 的 UUID 转换成 16 个 bytes。

user 直接使用蓝牙标准的 2 byte 的 UUID 时，Central 设备都知道这些 UUID 代表的设备类型。SDK 中已经定义了一些标准的 UUID，分布在以下文件中：stack/ble/service/uuid.h。

Telink 私有的一些 profile (OTA、SPP、MIC 等)，标准蓝牙里面不支持，在 stack/ble/service/uuid.h 中定义这些私有的 UUID，长度为 16 bytes。

(2) Attribute Handle

service 拥有多个 Attribute，这些 Attribute 组成一个 Attribute Table。在 Attribute Table 中，每一个 Attribute 都有一个 Attribute Handle 值，用来区分每一个不同的 Attribute。Peripheral 和 Central 建立连接后，Central 通过 Service Discovery 过程解析读取到 Peripheral 的 Attribute Table，并根据 Attribute Handle 的值来对应每一个不同的 Attribute，这样它们后面的数据通信只要带上 Attribute Handle，对方就知道是哪个 Attribute 的数据了。

(3) Attribute Value

每个 Attribute 都有对应的 Attribute Value，用来作为 request、response、notification 和 indication 的数据。在该 SDK 中，Attribute Value 用指针和指针所指区域的长度来描述。

### 3.4.2.2 Attribute and ATT Table

为了实现 Peripheral 端的 GATT service，SDK 设计了一个 Attribute Table，该 Table 由多个基本的 Attribute 组成。基本的 Attribute 的定义为：



```
typedef struct attribute
{
    u16 attNum;
    u8 perm;
    u8 uuidLen;
    u32 attrLen;      //4 bytes aligned
    u8* uuid;
    u8* pAttrValue;
    att_readwrite_callback_t w;
    att_readwrite_callback_t r;
} attribute_t;
```

结合目前该 SDK 给出的参考 Attribute Table 来说明以上各项的含义。Attribute Table 代码见 app\_att.c，如下截图所示：

```
static const attribute_t my_Attributes[] = {

{ATT_END_H - 1, 0,0,0,0,0}, // total num of attribute

// 0001 - 0007 gap
{7,ATT_PERMISSIONS_READ,2,2,(u8*)(&my_primaryServiceUUID), (u8*)(&my_gapServiceUUID), 0},
{0,ATT_PERMISSIONS_READ,2,sizeof(my_devNameCharVal),(u8*)(&my_characterUUID), (u8*)(&my_devNameCharVal), 0},
{0,ATT_PERMISSIONS_READ,2,sizeof(my_devName), (u8*)(&my_devNameUUID), (u8*)(&my_devName), 0},
{0,ATT_PERMISSIONS_READ,2,sizeof(my_appearanceCharVal),(u8*)(&my_characterUUID), (u8*)(&my_appearanceCharVal), 0},
{0,ATT_PERMISSIONS_READ,2,sizeof (my_appearance), (u8*)(&my_appearanceUUID), (u8*)(&my_appearance), 0},
{0,ATT_PERMISSIONS_READ,2,sizeof (my_periConnParamCharVal),(u8*)(&my_characterUUID), (u8*)(&my_periConnParamCharVal), 0},
{0,ATT_PERMISSIONS_READ,2,sizeof (my_periConnParameters),(u8*)(&my_periConnUUID), (u8*)(&my_periConnParameters), 0},


// 0008 - 000b gatt
{4,ATT_PERMISSIONS_READ,2,2,(u8*)(&my_primaryServiceUUID), (u8*)(&my_gattServiceUUID), 0},
{0,ATT_PERMISSIONS_READ,2,sizeof(my_serviceChangeCharVal),(u8*)(&my_characterUUID), (u8*)(&my_serviceChangeCharVal), 0},
{0,ATT_PERMISSIONS_READ,2,sizeof (serviceChangeVal), (u8*)(&serviceChangeUUID), (u8*)(&serviceChangeVal), 0},
{0,ATT_PERMISSIONS_RDWR,2,sizeof (serviceChangeCCC),(u8*)(&clientCharacterCfgUUID), (u8*)(&serviceChangeCCC), 0},


// 000c - 000e device Information Service
{3,ATT_PERMISSIONS_READ,2,2,(u8*)(&my_primaryServiceUUID), (u8*)(&my_devServiceUUID), 0},
{0,ATT_PERMISSIONS_READ,2,sizeof(my_PnCharVal),(u8*)(&my_characterUUID), (u8*)(&my_PnCharVal), 0},
{0,ATT_PERMISSIONS_READ,2,sizeof (my_PnPtrs),(u8*)(&my_PnPUUID), (u8*)(&my_PnPtrs), 0},
```

Figure 3.45: BLE SDK Attribute Table

请注意，Attribute Table 的定义前面加了 const：

```
const attribute_t my_Attributes[ ] = { ... };
```

const 关键字会让编译器将这个数组的数据最终都存储到 flash，以节省 ram 空间。这个 Attribute Table 定义在 Flash 上的所有内容是只读的，不能改写。

(1) attNum

attNum 有两个作用。

attNum 的第一个作用是表示当前 Attribute Table 中所有有效 Attribute 数目，即 Attribute Handle 的最大值，该数目只在 Attribute Table 数组的第 0 项无效的 Attribute 中使用：



```
{57,0,0,0,0,0}, // ATT_END_H - 1 = 57
```

attNum = 57 表示当前 Attribute Table 中共有 57 个 Attribute。

在 BLE 里，Attribute Handle 值从 0x0001 开始，往后加一递增，而数组的下标从 0 开始，在 Attribute Table 里加上上面这个虚拟的 Attribute，正好使得后面每个 Attribute 在数据里的下标号等于其 Attribute Handle 的值。当定义好了 Attribute Table 后，数 Attribute 在当前 Attribute Table 数组中的下标号，就能知道该 Attribute 当前的 Attribute Handle 值。

将 Attribute Table 中所有的 Attribute 数完，数到最后一个的编号就是当前 Attribute Table 中有效 Attribute 的数目 attNum，目前 SDK 中为 57，user 如果添加或删除了 Attribute，需要对此 attNum 进行修改，可以参考 vendor/acl\_connection\_demo/app\_att.h 的枚举 ATT\_HANDLE。

attNum 第二个作用是用于指定当前的 service 由几个 Attribute 构成。

每一个 service 的第一个 Attribute 的 UUID 都必须是 GATT\_UUID\_PRIMARY\_SERVICE(0x2800)，在这个 Attribute 上的 attNum 指定从当前 Attribute 开始往后数，总共有 attNum 个 Attribute 属于该 service 的组成部分。

如上面截图所示，gap service UUID 为 GATT\_UUID\_PRIMARY\_SERVICE 的第一个 Attribute，其 attNum 为 7，则 Attribute Handle 0x0001 ~ Attribute Handle 0x0007 这 7 个 Attribute 是属于 gap service 的描述。

同样，上图中的 HID service 的首个 Attribute 的 attNum 设为 27 后，从这个 Attribute 开始往后连续 27 个 Attribute 都属于 HID service。

除了第 0 项 Attribute 和每一个 service 首个 Attribute 外，其他所有的 Attribute 的 attNum 的值都必须设为 0。

## (2) perm

perm 是 permission 的简写。

perm 用于指定当前 Attribute 被 Client 访问的权限。

权限有以下 10 种，每个 Attribute 的权限都必须为下面的值或它们的组合。

#define ATT_PERMISSIONS_READ	0x01
#define ATT_PERMISSIONS_WRITE	0x02
#define ATT_PERMISSIONS_AUTHEN_READ	0x61
#define ATT_PERMISSIONS_AUTHEN_WRITE	0x62
#define ATT_PERMISSIONS_SECURE_CONN_READ	0xE1
#define ATT_PERMISSIONS_SECURE_CONN_WRITE	0xE2
#define ATT_PERMISSIONS_AUTHOR_READ	0x11
#define ATT_PERMISSIONS_AUTHOR_WRITE	0x12
#define ATT_PERMISSIONS_ENCRYPT_READ	0x21
#define ATT_PERMISSIONS_ENCRYPT_WRITE	0x22

## 注意：

目前 tl\_ble\_sdk 暂不支持授权读和授权写。

## (3) uuid and uuidLen



按照之前所述，UUID 分两种：BLE 标准的 2 bytes UUID 和 Telink 私有的 16 bytes UUID。通过 uuid 和 uuidLen 可以同时描述这两种 UUID。

uuid 是一个 u8 型指针，uuidLen 表示从指针开始的地方连续 uuidLen 个 byte 的内容为当前 UUID。Attribute Table 是存在 flash 上的，所有的 UUID 也是存在 flash 上的，所以 uuid 是指向 flash 的一个指针。

a) BLE 标准的 2 bytes UUID：

如 Attribute Handle = 0x0002 的 devNameCharacter 那个 Attribute，相关代码如下：

```
#define GATT_UUID_CHARACTER          0x2803
static const u16 my_characterUUID = GATT_UUID_CHARACTER;
static const u8 my_devNameCharVal[5] = {
    CHAR_PROP_READ,
    U16_LO(GenericAccess_DeviceName_DP_H), U16_HI(GenericAccess_DeviceName_DP_H),
    U16_LO(GATT_UUID_DEVICE_NAME), U16_HI(GATT_UUID_DEVICE_NAME)
};
{0,ATT_PERMISSIONS_READ,2,sizeof(my_devNameCharVal),(u8*)(&my_characterUUID), (u8*)
    ↵ (my_devNameCharVal), 0},
```

UUID=0x2803 在 BLE 中表示 character，uuid 指向 my\_devNameCharVal 在 flash 中的地址，uuidLen 为 2，peer Central 来读这个 Attribute 时，UUID 会是 0x2803。

b) Telink 私有的 16 bytes UUID：

如 OTA 的 Attribute，相关代码：

```
#define TELINK_SPP_DATA_OTA
0x12,0x2B,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00
static const u8 otaOutUuid[16]      = {TELINK_SPP_DATA_OTA};
static u8 my_OtaData        = 0x00;
{0,ATT_PERMISSIONS_RDWR,16,sizeof(my_OtaData),(u8*)(&my_OtaUUID),   (&my_OtaData), &otaWrite,
    ↵ NULL},
```

uuid 指向 my\_OtaData 在 flash 中的地址，uuidLen 为 16，Central 来读这个 Attribute 时，UUID 会是 0x000102030405060708090a0b0c0d2b12。

#### (4) pAttrValue and attrLen

每一个 Attribute 都会有对应的 Attribute Value。pAttrValue 是一个 u8 型指针，指向 Attribute Value 所在 RAM/Flash 的地址，attrLen 用来反映该数据在 RAM/Flash 上的长度。当 Central 读取 Peripheral 某个 Attribute 的 Attribute Value 时，SDK 从 Attribute 的 pAttrValue 指针指向的区域 (RAM/Flash) 开始，取 attrLen 个数据回给 Central。

UUID 是只读的，所以 uuid 是指向 flash 的指针；而 Attribute Value 可能会涉及到写操作，如果有写操作必须放在 RAM 上，所以 pAttrValue 可能指向 RAM，也可能指向 Flash。

Attribute Handle=0x0027 hid Information 的 Attribute，相关代码：



```

const u8 hidInformation[] =
{
    U16_L0(0x0111), U16_HI(0x0111), // bcdHID (USB HID version), 0x11,0x01
    0x00, // bCountryCode
    0x01 // Flags
};

{0,ATT_PERMISSIONS_READ,2, sizeof(hidInformation),(u8*)(&hidInformationUUID), (u8*)
    ↵ (hidInformation), 0},

```

在实际应用中，hidInformation 4 个 byte 0x01 0x00 0x01 0x11 是只读的，不会涉及到写操作，所以定义时可以使用 const 关键字存储在 Flash 上。pAttrValue 指向 hidInformation 在 flash 上的地址，此时 attrlen 以 hidInformation 实际的长度取值。当 Central 读该 Attribute 时，会根据 pAttrValue 和 attrLen 返回 0x01000111 给 Central。

Central 读该 Attribute 时 BLE 抓包如下图，Central 使用 ATT\_Read\_Req 命令，假定设置要读的 AttHandle = 0x0023 = 35，对应着 SDK 中 Attribute Table 中的 hid information。

Figure 3.46: Central 读 hidInformation 的 BLE 抓包

Attribute Handle=0x002C battery value 的 Attribute，相关代码：

```

u8 my_batVal[1] = {99};
{0,ATT_PERMISSIONS_READ,2,sizeof(my_batVal),(u8*)(&my_batCharUUID), (u8*)(my_batVal), 0}

```

实际应用中，反应当前电池电量的 my\_batVal 值会根据 ADC 采样到的电量而改变，然后通过 Peripheral 主动 notify 或者 Central 主动读的方式传输给 Central，所以 my\_batVal 应该放在内存上，此时 pAttrValue 指向 my\_batVal 在 RAM 上的地址。

### (5) 回调函数 w

回调函数 w 是写函数。函数原型：

```
typedef int (*att_readwrite_callback_t)(void* p);
```

user 如果需要定义回调写函数，须遵循上面格式。回调函数 w 是 optional 的，对某一个具体的 Attribute 来说，user 可以设置回调写函数，也可以不设置回调（不设置回调的时候用空指针 0 表示）。

回调函数 w 触发条件为：当 Peripheral 收到的 Attribute PDU 的 Attribute Opcode 为以下三个时，Peripheral 会检查回调函数 w 是否被设置：



- a) opcode = 0x12, Write Request.
- b) opcode = 0x52, Write Command.
- c) opcode = 0x18, Execute Write Request.

Peripheral 收到以上写命令后，如果没有设置回调函数 w，Peripheral 会自动向 pAttrValue 指针所指向的区域写 Central 传过来的值，写入的长度为 Central 数据包格式中的 l2capLen-3；如果 user 设置了回调函数 w，Peripheral 收到以上写命令后执行 user 的回调函数 w，此时不再向 pAttrValue 指针所指区域写数据。这两个写操作是互斥的，只能有一个生效。

user 设置回调函数 w 是为了处理 Central 在 ATT 层的 Write Request、Write Command 和 Execute Write Request 命令，如果没有设置回调函数 w，需要评估 pAttrValue 所指向的区域是否能够完成对以上命令的处理（如 pAttrValue 指向 flash 无法完成写操作；或者 attrLen 长度不够，Central 的写操作会越界，导致其他数据被错误的改写）。

### 3.4.5.1 Write Request

The *Write Request* is used to request the server to write the value of an attribute and acknowledge that this has been achieved in a *Write Response*.

Parameter	Size (octets)	Description
Attribute Opcode	1	0x12 = Write Request
Attribute Handle	2	The handle of the attribute to be written
Attribute Value	0 to (ATT_MTU-3)	The value to be written to the attribute

Figure 3.47: BLE 协议栈中 Write Request

### 3.4.5.3 Write Command

The *Write Command* is used to request the server to write the value of an attribute, typically into a control-point attribute.

Parameter	Size (octets)	Description
Attribute Opcode	1	0x52 = Write Command
Attribute Handle	2	The handle of the attribute to be set
Attribute Value	0 to (ATT_MTU-3)	The value of be written to the attribute

Figure 3.48: BLE 协议栈中 Write Command



### 3.4.6.3 Execute Write Request

The *Execute Write Request* is used to request the server to write or cancel the write of all the prepared values currently held in the prepare queue from this client. This request shall be handled by the server as an atomic operation.

Parameter	Size (octets)	Description
Attribute Opcode	1	0x18 = Execute Write Request

Figure 3.49: BLE 协议栈中 Execute Write Request

回调函数 w 的 void 型 p 指针指向 Central 写命令的具体数值。实际 p 指向一片内存，内存上的值如下面结构体所示。

```
typedef struct{
    u8 type;
    u8 rf_len;      //User do not use this member, because it may be changed by stack layer.
    u16 l2capLen;
    u16 chanId;

    u8 opcode;
    u16 handle;

    u8 dat[20];
}rf_packet_att_t;
```

p 指向第一个元素 type。写过来的数据有效长度为 l2cap - 3，第一个有效数据为 dat[0]。

```
int my_WriteCallback(u16 connHandle, void * p)
{
    rf_packet_att_t *pw = (rf_packet_att_t *)p;
    int len = pw->l2capLen - 3;
    //add your code
    //valid data is pw->dat[0] ~ pw->dat[len-1]
    return 1;
}
```

上面这个结构体 rf\_packet\_att\_t 所在位置为 stack/ble/ble\_format.h。

注意：

结构体 rf\_packet\_att\_t 内的 rf\_len，用户不要使用，rf\_len 在拼包时有可能会被改写，请使用 l2capLen 换算后再使用。

(6) 回调函数 r



回调函数 r 是读函数。函数原型：

```
typedef int (*att_readwrite_callback_t)(void* p);
```

user 如果需要定义回调读函数，须遵循上面格式。回调函数 r 是 optional 的，对某一个具体的 Attribute 来说，user 可以设置回调读函数，也可以不设置回调（不设置回调的时候用空指针 0 表示）。

回调函数 r 触发条件为：当 Peripheral 收到的 Attribute PDU 的 Attribute Opcode 为以下两个时，Peripheral 会检查回调函数 r 是否被设置：

- a) opcode = 0x0A, Read Request.
- b) opcode = 0x0C, Read Blob Request.

Peripheral 收到以上读命令后，

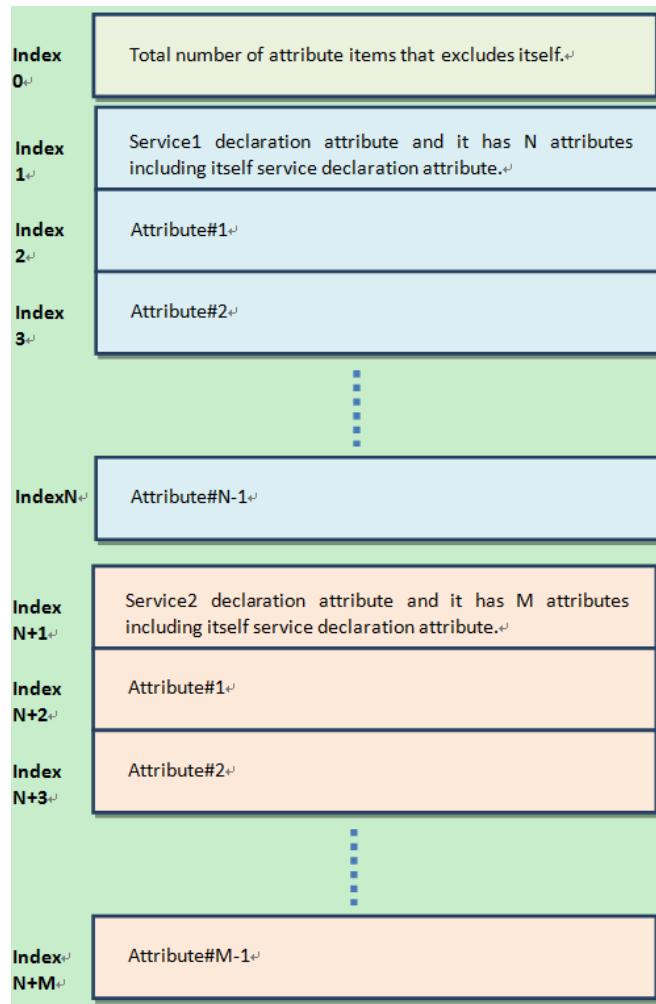
- a) 如果用户设置了回调读函数，执行该函数，根据该函数的返回值决定是否回复 Read Response/Read Blob Response：
  - 若返回值为 1，Peripheral 不回复 Read Response/Read Blob Response 给 Central。
  - 若返回值为其他值，Peripheral 从 pAttrValue 指针所指向的区域读 attrLen 个值用 Read Response/Read Blob Response 回复给 Central。
- b) 如果用户没有设置回调读函数，Peripheral 从 pAttrValue 指针所指向的区域读 attrLen 个值用 Read Response/Read Blob Response 回复给 Central。

如果用户想在收到 Central 的 Read Request/Read Blob Request 后修改即将回复的 Read Response/Read Blob Response 的内容，就可以注册对应的回调函数 r，在回调函数里修改 pAttrValue 指针所指 ram 的内容，并且 return 的值只能是 0。

#### (7) Attribute Table 结构

根据以上对 Attribute 的详细说明，使用 Attribute Table 构造 Service 结构如下图所示。第一个 Attribute 的 atnum 用于指示当前 ATT Table Attribute 的数量，剩余的 Attribute 首先按 Service 分组，每一组的头一条 Attribute 是该 Service 的 declaration，并且使用 atnum 指定后面紧跟的多少条 Attribute 属于该 Service 的具体描述。实际每组 Service 的第一条是一个 Primary Service。

```
#define GATT_UUID_PRIMARY_SERVICE      0x2800    //!< Primary Service
const u16 my_primaryServiceUUID = GATT_UUID_PRIMARY_SERVICE;
```



**Figure 3.50: Service Attribute Layout**

#### (8) ATT table Initialization

GATT & ATT 初始化只需要将应用层的 Attribute Table 的指针传到协议栈即可，提供的 API：

```
void bls_att_setAttributeTable (u8 *p);
```

p 为 Attribute Table 的指针。

#### (9) ATT HANDLE

每一个 attribute 都对应一个 handle，handle 的具体枚举位于 app\_att.h 文件的 **ATT\_HANDLE** 中，如 GAP service 对应 att handle 枚举如下：

```
static const attribute_t my_Attributes[] = {
    ...
    // 0001 - 0007 gap
    {7, ATT_PERMISSIONS_READ, 2, 2, (u8 *)(&my_primaryServiceUUID), (u8 *)(&my_gapServiceUUID), 0, 0},
    {0, ATT_PERMISSIONS_READ, 2, sizeof(my_devNameCharVal), (u8 *)(&my_characterUUID), (u8 *)(&my_devNameCharVal), 0, 0},
    ...
}
```



```
{0, ATT_PERMISSIONS_READ, 2, sizeof(my_devName), (u8 *)(&my_devNameUUID), (u8 *)
    *(size_t)(my_devName), 0, 0},
{0, ATT_PERMISSIONS_READ, 2, sizeof(my_appearanceCharVal), (u8 *)(&size_t)
    (&my_characterUUID), (u8 *)(&size_t)(my_appearanceCharVal), 0, 0},
{0, ATT_PERMISSIONS_READ, 2, sizeof(my_appearance), (u8 *)(&size_t)(&my_appearanceUUID),
    (u8 *)(&size_t)(my_appearance), 0, 0},
{0, ATT_PERMISSIONS_READ, 2, sizeof(my_periConnParamCharVal), (u8 *)(&size_t)
    (&my_characterUUID), (u8 *)(&size_t)(my_periConnParamCharVal), 0, 0},
{0, ATT_PERMISSIONS_READ, 2, sizeof(my_periConnParameters), (u8 *)(&size_t)
    (&my_periConnParamUUID), (u8 *)(&size_t)(&my_periConnParameters), 0, 0},
...
}
```

```
typedef enum
{
    ATT_H_START = 0,
    //////////////////////////////////////////////////////////////////

    //*****
    GenericAccess_PS_H,           //UUID: 2800,   VALUE: uuid 1800
    GenericAccess_DeviceName_CD_H, //UUID: 2803,   VALUE:           Prop: Read / Notify
    GenericAccess_DeviceName_DP_H, //UUID: 2A00,   VALUE: device name
    GenericAccess_Appearance_CD_H, //UUID: 2803,   VALUE:           Prop: Read
    GenericAccess_Appearance_DP_H, //UUID: 2A01,   VALUE: appearance
    CONN_PARAM_CD_H,             //UUID: 2803,   VALUE:           Prop: Read
    CONN_PARAM_DP_H,             //UUID: 2A04,   VALUE: connParameter
    ...
}
```

(10) 如何添加一个新的 att service，下面是一个简单示例

a) 在 app\_att.c 中的 my\_Attributes 数组里加入需要新添加的 service 信息，例子如下：

```
////////////////// CUSTOM DEFINE EXAMPLE ///////////////////
#define TELINK_CUSTOM_DEFINE_EXAMPLE 0x23, 0x23, 0x23, 0x23, 0x23, 0x23, 0x23, 0x23,
    0x23, 0x23, 0x23, 0x23, 0x23, 0x23
static const u8 TelinkCustomDefineExampleUUID[16] =
    WRAPPING_BRACES(TELINK_CUSTOM_DEFINE_EXAMPLE);

static const u8     TelinkBrithday[]     = "20100630";
static const u8     mood_of_today[]      = {'E', 'x', 'c', 'i', 't', 'i', 'n', 'g'};

static const attribute_t my_Attributes[] = {
    .....
    // 0040 - 0042 custom_define_example
    {3, ATT_PERMISSIONS_READ, 2, 16, (u8 *)(&size_t)(&my_primaryServiceUUID), (u8 *)(&size_t)
        (&TelinkCustomDefineExampleUUID), 0, 0},
```



```
{0, ATT_PERMISSIONS_RDWR, 2, sizeof(TelinkBirthday), (u8 *)(size_t)(&my_characterUUID),
↪ (u8 *)(size_t)(TelinkBirthday), 0, 0},
{0, ATT_PERMISSIONS_RDWR, 2, sizeof(mood_of_today), (u8 *)(size_t)(&userdesc_UUID),
↪ (u8 *)(size_t)(&mood_of_today), 0, 0},
};
```

b) 在 app\_att.h 中的 ATT\_HANDLE 里添加新加 service 的枚举

```
typedef enum
{
    ...
    //////////////////////////////////////////////////////////////////
    //***** CUSTOM DEFINE *****
    CUSTOM_DEFINE_PS_H,      //UUID: TELINK_CUSTOM_DEFINE_EXAMPLE,   VALUE: TELINK CUSTOM DEFINE
    ↪ UUID
    TELINK_BRITHDAY_CD_H,   //UUID: 2803,     VALUE: TelinkBirthday      Prop: Read / write
    MOOD_OF_TODAY_DP_H,     //UUID: 2901,     VALUE: mood_of_today
    ...
} ATT_HANDLE;
```

### 3.4.2.3 GATT Service Security

在介绍 GATT Service Security 前，用户可以先了解一下 SMP 相关的内容。

请参考[“SMP”章节](#)相关的详细介绍，了解 LE 配对方式、加密等级等基础知识。

下图是 Bluetooth Core Specification 给出的 GATT 服务安全等级服务请求之间映射关系，详细可以参考《core5.0》(Vol3/Part C/10.3 AUTHENTICATION PROCEDURE)。



Link Encryption State	Local Device's Access Requirement for Service	Local Device Pairing Status			
		No LTK No STK	Unauthenticated LTK or Unauthenticated STK	Authenticated LTK or Authenticated STK	Authenticated LTK with Secure Connections
Unencrypted	<b>None</b>	Request succeeds	Request succeeds	Request succeeds	Request succeeds
	<b>Encryption, No MITM Protection</b>	Error Resp.: Insufficient Authentication	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption
	<b>Encryption, MITM Protection</b>	Error Resp.: Insufficient Authentication	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption
	<b>Encryption, MITM Protection, Secure Connections</b>	Error Resp.: Insufficient Authentication	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption
Encrypted	<b>None</b>	N/A (Not possible to be encrypted without LTK)	Request succeeds	Request succeeds	Request succeeds
	<b>Encryption, No MITM Protection</b>		Request succeeds	Request succeeds	Request succeeds
	<b>Encryption, MITM Protection</b>		Error Resp.: Insufficient Authentication	Request succeeds	Request succeeds
	<b>Encryption, MITM Protection, Secure Connections</b>		Error Resp.: Insufficient Authentication	Error Resp.: Insufficient Authentication	Request succeeds

Table 10.2: Local device responds to a service request

Figure 3.51: 服务请求响应映射关系

用户可以很清楚的看到：

- 第一列跟当前连接的 Peripheral 设备是否处于加密状态下有关；
- 第二列(local Device's Access Requirement for service)则跟用户设置的 ATT 表中特性的权限(Permission Access) 设置有关，如下图所示；
- 第三列又分为 4 个子列，这 4 个子列则对应当前 LE 安全模式 1 下四个级别（具体说就是当前的设备配对状态是否是如下 4 种中的一种：
  - a) No authentication and no encryption
  - b) Unauthenticated pairing with encryption
  - c) Authenticated pairing with encryption
  - d) Authenticated LE Secure Connections



```
/** @defgroup ATT_PERMISSIONS_BITMAPS GAP ATT Attribute Access Permissions Bit Fields
 * @{
 * (See the Core_v5.0 (Vol 3/Part C/10.3.1/Table 10.2) for more information)
 */
#define ATT_PERMISSIONS_AUTHOR          0x10 //Attribute access(Read & Write) requires Authorization
#define ATT_PERMISSIONS_ENCRYPT         0x20 //Attribute access(Read & Write) requires Encryption
#define ATT_PERMISSIONS_AUTHEN         0x40 //Attribute access(Read & Write) requires Authentication(MITM protection)
#define ATT_PERMISSIONS_SECURE_CONN     0x80 //Attribute access(Read & Write) requires Secure Connection
#define ATT_PERMISSIONS_SECURITY       (ATT_PERMISSIONS_AUTHOR | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN | ATT_PERMISSIONS_SECURE_CONN)

//user can choose permission below
#define ATT_PERMISSIONS_READ           0x01 //!< Attribute is Readable
#define ATT_PERMISSIONS_WRITE          0x02 //!< Attribute is Writable
#define ATT_PERMISSIONS_RDWR            (ATT_PERMISSIONS_READ | ATT_PERMISSIONS_WRITE) //!< Attribute is Readable & Writable

#define ATT_PERMISSIONS_ENCRYPT_READ   (ATT_PERMISSIONS_READ | ATT_PERMISSIONS_ENCRYPT) //!< Read requires Encryption
#define ATT_PERMISSIONS_ENCRYPT_WRITE  (ATT_PERMISSIONS_WRITE | ATT_PERMISSIONS_ENCRYPT) //!< Write requires Encryption
#define ATT_PERMISSIONS_ENCRYPT_RDWR   (ATT_PERMISSIONS_RDWR | ATT_PERMISSIONS_ENCRYPT) //!< Read & Write requires Encryption

#define ATT_PERMISSIONS_AUTHEN_READ    (ATT_PERMISSIONS_READ | ATT_PERMISSIONS_AUTHEN) //!< Read requires Authentication
#define ATT_PERMISSIONS_AUTHEN_WRITE   (ATT_PERMISSIONS_WRITE | ATT_PERMISSIONS_AUTHEN) //!< Write requires Authentication
#define ATT_PERMISSIONS_AUTHEN_RDWR    (ATT_PERMISSIONS_RDWR | ATT_PERMISSIONS_AUTHEN) //!< Read & Write requires Authentication

#define ATT_PERMISSIONS_SECURE_CONN_READ (ATT_PERMISSIONS_READ | ATT_PERMISSIONS_SECURE_CONN | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN)
#define ATT_PERMISSIONS_SECURE_CONN_WRITE (ATT_PERMISSIONS_WRITE | ATT_PERMISSIONS_SECURE_CONN | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN)
#define ATT_PERMISSIONS_SECURE_CONN_RDWR (ATT_PERMISSIONS_RDWR | ATT_PERMISSIONS_SECURE_CONN | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN)

#define ATT_PERMISSIONS_AUTHOR_READ    (ATT_PERMISSIONS_READ | ATT_PERMISSIONS_AUTHOR) //!< Read requires Authorization
#define ATT_PERMISSIONS_AUTHOR_WRITE   (ATT_PERMISSIONS_WRITE | ATT_PERMISSIONS_AUTHOR) //!< Write requires Authorization
#define ATT_PERMISSIONS_AUTHOR_RDWR    (ATT_PERMISSIONS_RDWR | ATT_PERMISSIONS_AUTHOR) //!< Read & Write requires Authorization
```

Figure 3.52: ATT Permission 定义

最终 GATT service security 的实现跟 SMP 初始化时的参数配置，包括支持的最高安全级别设置、ATT 表中的特性权限设置等都有关系，而且跟 Central 也有关系，比如 Peripheral 设置的 SMP 能支持的最高等级是 Authenticated pairing with encryption，但是 Central 具备的最高安全等级是 Unauthenticated pairing with encryption，此时如果 ATT 表中某个写特性的权限是 ATT\_PERMISSIONS\_AUTHEN\_WRITE，那么 Central 在写该特性时，我们会回复加密等级不够的错误。

用户可以设定 ATT 表中特性权限实现如下应用：

比如 Peripheral 设备支持的最高安全级别是 Unauthenticated pairing with encryption，但是不想连接后使用发送 Security Request 这种方式去触发 Central 开始配对，那么客户可以将某些具备 notify 属性的客户端特性配置 (Client Characteristic Configuration, 简称 CCC) 属性的权限设置为 ATT\_PERMISSIONS\_ENCRYPT\_WRITE，那么 Central 只有写该 CCC 后，Peripheral 会回复其安全级别不够，这会触发 Central 开启配对加密流程。

#### 注意：

用户设置的安全级别只表示设备能支持的最高安全级别，只要 ATT 表中特性的权限 (ATT Permission) 不超过实际生效的最高级别就可以通过 GATT service security 管控。对于 LE 安全模式 1 中的等级 4 来说，如果用户只设置 Authenticated LE Secure Connections 一种级别，则代表当前设置支持 LE Secure Connections only。

#### 3.4.2.4 Attribute PDU and GATT API

根据 Bluetooth Core Specification，tl\_ble\_sdk 目前支持的 Attribute PDU 有以下几类：

- Requests：client 发送给 server 的数据请求。
- Responses：server 收到 client 的 request 后发送的数据回复。
- Commands：client 发送给 server 的命令。
- Notifications：server 发送给 client 的数据。



- Indications: server 发送给 client 的数据。
- Confirmations: client 对 server Indication 数据的确认。

下面结合之前介绍的 Attribute 结构和 Attribute Table 结构，对 ATT 层所有的 ATT PDU 进行分析。

#### 34.24.1 Read by Group Type Request, Read by Group Type Response

Read by Group Type Request 和 Read by Group Type Response，详情请参考 Bluetooth Core Specification V5.4, Vol 3, Part F, 3.4.4.9 ATT\_READ\_BY\_GROUP\_TYPE\_REQ/3.4.4.10 ATT\_READ\_BY\_GROUP\_TYPE\_RSP。

Central 发送 Read by Group Type Request，在该命令中指定起始和结束的 attHandle，指定 attGroupType。Peripheral 收到该 Request 后，遍历当前 Attribute table，在指定的起始和结束的 attHandle 中找到符合 attGroupType 的 Attribute Group，通过 Read by Group Type Response 回复 Attribute Group 信息。

Data Type	Data Header				L2CAP Header			ATT_Read_By_Group_Type_Req				CRC	RSSI (dBm)	FCS						
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttGroupType	0x0007	0x0004	0x10	0x0001	0xFFFF	00 28	0x89867B	-38	OK
Data Type	Data Header				CRC	RSSI (dBm)	FCS	ATT_Read_By_Group_Type_Rsp				CRC	RSSI (dBm)	FCS						
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE00D5	-38	OK	0x0014	0x0004	0x11	0x06	01 00 07 00 00 18 08 00 0A 00 0A 18 0B 00 25 00 12 18	0x58FC67	-38	OK				
Data Type	Data Header				L2CAP Header	ATT_Read_By_Group_Type_Req			ATT_Read_By_Group_Type_Rsp				CRC	RSSI (dBm)	FCS					
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	0xAE00D5	0x0004	0x10	0x0026	0xFFFF	00 28	0x0007	0x0004	0x11	0x06	01 00 07 00 00 18 08 00 0A 00 0A 18 0B 00 25 00 12 18	0x5A6275	-38	OK	
Data Type	Data Header				CRC	RSSI (dBm)	FCS	ATT_Read_By_Group_Type_Req				CRC	RSSI (dBm)	FCS						
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE0BA0	-38	OK	0x0007	0x0004	0x11	0x06	02 00 28 00 0F 18	0x158866	-38	OK				
Data Type	Data Header				CRC	RSSI (dBm)	FCS	ATT_Read_By_Group_Type_Req				CRC	RSSI (dBm)	FCS						
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE0D73	-38	OK	0x0007	0x0004	0x10	0x029	0xFFFF	00 28	0x055C4D	-38	OK			
Data Type	Data Header				L2CAP Header	ATT_Read_By_Group_Type_Rsp			CRC	RSSI (dBm)	FCS	CRC	RSSI (dBm)	FCS						
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	0x0008	0x0004	0x11	0x06	26 00 28 00 0F 18	0x898D99	-38	OK							
Data Type	Data Header				L2CAP Header	ATT_Read_By_Group_Type_Req			CRC	RSSI (dBm)	FCS	CRC	RSSI (dBm)	FCS						
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x0007	0x0004	0x10	0x0026	0xFFFF	00 28	0x0007	0x0004	0x11	0x06	29 00 32 00 11 19 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00	0x3C57D1	-38	OK	
Data Type	Data Header				L2CAP Header	ATT_Read_By_Group_Type_Rsp			CRC	RSSI (dBm)	FCS	CRC	RSSI (dBm)	FCS						
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE0BA0	-38	OK	0x0007	0x0004	0x10	0x033	0xFFFF	00 28	0x600FAA	-38	OK			
Data Type	Data Header				CRC	RSSI (dBm)	FCS	ATT_Error_Response				CRC	RSSI (dBm)	FCS						
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE0D73	-38	OK	0x0005	0x0004	0x01	0x10	0x0033	ATT NOT FOUND(0xA)	0x600FAA	-38	OK			

Figure 3.53: Read by Group Type Request Read by Group Type Response

上图所示，Central 查询 Peripheral 的 UUID 为 0x2800 的 primaryServiceUUID 的 Attribute Group 信息：

```
#define GATT_UUID_PRIMARY_SERVICE          0x2800
const u16 my_primaryServiceUUID = GATT_UUID_PRIMARY_SERVICE;
```

参考当前 demo code，Attribute table 中有以下几组符合该要求：



- (1) attHandle 从 0x0001 ~ 0x0007 的 Attribute Group, Attribute Value 为 SERVICE\_UUID\_GENERIC\_ACCESS (0x1800)。
- (2) attHandle 从 0x0008 ~ 0x000B 的 Attribute Group, Attribute Value 为 SERVICE\_UUID\_GENERIC\_ATTRIBUTE (0x1801)。
- (3) attHandle 从 0x000C ~ 0x000E 的 Attribute Group, Attribute Value 为 SERVICE\_UUID\_DEVICE\_INFORMATION (0x180A)。
- (4) attHandle 从 0x000F ~ 0x0029 的 Attribute Group, Attribute Value 为 SERVICE\_UUID\_HUMAN\_INTERFACE\_DEVICE (0x1812)。
- (5) attHandle 从 0x002A ~ 0x002D 的 Attribute Group, Attribute Value 为 SERVICE\_UUID\_BATTERY (0x180F)。
- (6) attHandle 从 0x002E ~ 0x0035 的 Attribute Group, Attribute Value 为 TELINK\_SPP\_UUID\_SERVICE (0x10, 0x19, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00)。
- (7) attHandle 从 0x0036 ~ 0x0039 的 Attribute Group, Attribute Value 为 TELINK\_OTA\_UUID\_SERVICE (0x12, 0x19, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00)。

Peripheral 将以上 7 个 GROUP 的 attHandle 和 attValue 的信息通过 Read by Group Type Response 回复给 Central, 最后一个 ATT\_Error\_Response 表明所有的 Attribute Group 都已回复完毕, Response 结束, Central 看到这个包也会停止发送 Read by Group Type Request。

使用下面 API 实现 Read by Group Request:

```
ble_sts_t blc_gatt_pushReadByGroupTypeRequest(u16 connHandle, u16 start_attHandle, u16
    end_attHandle, u8 *uuid, int uuid_len);
```

Read by Group Response 的数据, 在 app\_gatt\_data\_handler 函数里可以读取并处理。

#### 34.24.2 Find by Type Value Request, Find by Type Value Response

Find by Type Value Request 和 Find by Type Value Response, 详情请参考 Bluetooth Core Specification V5.4, Vol 3, Part F, 3.4.3.3 ATT\_FIND\_BY\_TYPE\_VALUE\_REQ/3.4.3.4 ATT\_FIND\_BY\_TYPE\_VALUE\_RSP。

Central 发送 Find by Type Value Request, 在该命令中指定起始和结束的 attHandle, 指定 AttributeType 和 Attribute Value。Peripheral 收到该 Request 后, 遍历当前 Attribute table, 在指定的起始和结束的 attHandle 中找到 AttributeType 和 Attribute Value 相匹配的 Attribute, 通过 Find by Type Value Response 回复 Attribute。

<b>Data Type</b>	<b>Data Header</b>					<b>L2CAP Header</b>			<b>ATT_Find_By_Type_Value_Req</b>					<b>CRC</b>	<b>RSSI (dBm)</b>	<b>FCS</b>	
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttType	AttValue			0x4CEA12	-54	OK
	2	1	1	0	13	0x0009	0x0004	0x06	0x0001	0xFFFF	0x2800	0A 18					
<b>Data Type</b>	<b>Data Header</b>					<b>CRC</b>	<b>RSSI (dBm)</b>	<b>FCS</b>									
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xC4C0E8	-54	OK									
<b>Data Type</b>	<b>Data Header</b>					<b>L2CAP Header</b>			<b>ATT_Find_By_Type_Value_Rsp</b>					<b>CRC</b>	<b>RSSI (dBm)</b>	<b>FCS</b>	
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	HandleInfo						0xF92ED9	-54	OK
	2	1	0	0	9	0x0005	0x0004	0x07	0C 00 0E 00								

**Figure 3.54:** Find by Type Value Request Find by Type Value Response

使用下面 API 实现 Find by Type Value Request:



```
ble_sts_t blc_gatt_pushFindByTypeValueRequest(u16 connHandle, u16 start_attHandle, u16
↪ end_attHandle, u16 uuid, u8 *attr_value, int len);
```

Find by Type Value Response 的数据，在 app\_gatt\_data\_handler 函数里可以读取并处理。

### 34.24.3 Read by Type Request, Read by Type Response

Read by Type Request 和 Read by Type Response，详情请参考 Bluetooth Core Specification V5.4, Vol 3, Part F, 3.4.4.1 ATT\_READ\_BY\_TYPE\_REQ/3.4.4.2 ATT\_READ\_BY\_TYPE\_RSP。

Central 发送 Read by Type Request，在该命令中指定起始和结束的 attHandle，指定 AttributeType。Peripheral 收到该 Request 后，遍历当前 Attribute table，在指定的起始和结束的 attHandle 中找到符合 AttributeType 的 Attribute，通过 Read by Type Response 回复 Attribute。

<b>Data Type</b>	<b>Data Header</b>					<b>L2CAP Header</b>		<b>ATT_Read_By_Type_Req</b>						
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttType		0x	
	2	0	0	1	11	0x0007	0x0004	0x08	0x0001	0xFFFF	00 2A			
<b>Data Type</b>	<b>Data Header</b>					<b>CRC</b>	<b>RSSI (dBm)</b>	<b>FCS</b>						
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x898717	0	OK						
<b>Data Type</b>	<b>Data Header</b>					<b>CRC</b>	<b>RSSI (dBm)</b>	<b>FCS</b>						
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x898AB1	0	OK						
<b>Data Type</b>	<b>Data Header</b>					<b>CRC</b>	<b>RSSI (dBm)</b>	<b>FCS</b>						
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x898C62	0	OK						
<b>Data Type</b>	<b>Data Header</b>					<b>CRC</b>	<b>RSSI (dBm)</b>	<b>FCS</b>						
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x8981C4	0	OK						
<b>Data Type</b>	<b>Data Header</b>					<b>L2CAP Header</b>		<b>ATT_Read_By_Type_Rsp</b>					<b>CRC</b>	
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	Length	AttData		0x		
	2	1	0	0	14	0x000A	0x0004	0x09	0x08	03 00 74 53 65 6C 66 69		DB602		

Figure 3.55: Read by Type Value Request Find by Type Value Response

上图所示，Central 读 attType 为 0x2A00 的 Attribute，Peripheral 中 Attribute Handle 为 0x0003 的 Attribute：

```
static const u8 my_devName[] = {'m', 'u', 'l', 't', 'i', '_', 'c', 'o', 'n', 'n'};
#define GATT_UUID_DEVICE_NAME          0x2A00
const u16 my_devNameUUID = GATT_UUID_DEVICE_NAME;
{0,ATT_PERMISSIONS_READ,2,sizeof(my_devName), (u8*)(&my_devNameUUID), (u8*)(my_devName), 0}
```

Read by Type response 中 length 为 8，attData 中前两个 byte 为当前的 attHandle 0003，后面 6 个 bytes 为对应的 Attribute Value。

使用下面 API 实现 Read by Type Request：

```
ble_sts_t blc_gatt_pushReadByTypeRequest(u16 connHandle, u16 start_attHandle, u16
↪ end_attHandle, u8 *uuid, int uuid_len);
```



Read by Type Response 的数据，在 app\_gatt\_data\_handler 函数里可以读取并处理。

#### 34.244 Find information Request, Find information Response

Find information request 和 Find information response，详情请参考 Bluetooth Core Specification V5.4, Vol 3, Part F, 3.4.3.1 ATT\_FIND\_INFORMATION\_REQ/3.4.3.2 ATT\_FIND\_INFORMATION\_RSP。

Central 发送 Find information request，指定起始和结束的 attHandle。Peripheral 收到该命令后，将起始和结束的所有 attHandle 对应 Attribute 的 UUID 通过 Find information response 回复给 Central。如下图所示，Central 要求获得 attHandle 0x0016 ~ 0x0018 三个 Attribute 的 information，Peripheral 回复这三个 Attribute 的 UUID。

Data Type	Data Header					L2CAP Header			ATT_Find_Info_Req			CRC	RSSI (dBm)	FCS		
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	0x04	0x0016	0x0018	0x362A2F	-38	OK
Data Type	Data Header					CRC	RSSI (dBm)	FCS								
Empty PDU	1	0	0	0	0	0xAE00D5	-38	OK								
Data Type	Data Header					CRC	RSSI (dBm)	FCS								
Empty PDU	1	1	0	0	0	0xAE0606	-38	OK								
Data Type	Data Header					L2CAP Header			ATT_Find_Info_Rsp			CRC	RSSI (dBm)	FCS		
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	Format	InfoData	0x05	0x01	16 00 02 29 17 00 08 29 18 00 03 28	0x99C5FD	-38	OK

Figure 3.56: Find Information Request Find Information Response

#### 34.24.5 Read Request, Read Response

Read Request 和 Read Response，详情请参考 Bluetooth Core Specification V5.4, Vol 3, Part F, 3.4.4.3 ATT\_READ\_REQ/3.4.4.4 ATT\_READ\_RSP。

Central 发送 Read Request，指定某一个 attHandle 为 0x0017，Peripheral 收到后通过 Read Response 回复指定的 Attribute 的 Attribute Value（若设置了回调函数 r，执行该函数），如下图所示。

Data Type	Data Header					L2CAP Header			ATT_Read_Req			CRC	RSSI (dBm)	FCS	
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	0x0A	0x0017	0x99C5FD	-38	OK	
Data Type	Data Header					CRC	RSSI (dBm)	FCS							
Empty PDU	1	0	0	0	0	0xAE00D5	-38	OK							
Data Type	Data Header					CRC	RSSI (dBm)	FCS							
Empty PDU	1	1	0	0	0	0xAE0606	-38	OK							
Data Type	Data Header					L2CAP Header			ATT_Read_Rsp			CRC	RSSI (dBm)	FCS	
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttValue	0x0B	02 01	0x9082A7	-38	OK	

Figure 3.57: Read Request Read Response

使用下面 API 实现 Read Request



```
ble_sts_t blc_gatt_pushReadRequest(u16 connHandle, u16 attHandle);
```

Read Response 的数据，在 app\_gatt\_data\_handler 函数里可以读取并处理。

#### 34.24.6 Read Blob Request, Read Blob Response

Read Blob Request 和 Read Blob Response，详情请参考 Bluetooth Core Specification V5.4, Vol 3, Part F, 3.4.4.5 ATT\_READ\_BLOB\_REQ/3.4.4.6 ATT\_READ\_BLOB\_RSP。

当 Peripheral 某个 Attribute 的 Attribute Value 值的长度超过 MTU\_SIZE (目前 tl\_ble\_sdk 中默认为 23) 时，Central 需要启用 Read Blob Request 来读取该 Attribute Value，从而使得 Attribute Value 可以分包发送。Central 在 Read Blob Request 指定 attHandle 和 ValueOffset，Peripheral 收到该命令后，找到对应的 Attribute，根据 ValueOffset 值通过 Read Blob Response 回复 Attribute Value (若设置了回调函数 r，执行该函数)。

如下图所示，Central 读 Peripheral 的 HID report map (report map 很大，远远超过 23) 时，首先发送 Read Request，Peripheral 回 Read response，将 report map 前一部分数据回给 Central。之后 Central 使用 Read Blob Request，Peripheral 通过 Read Blob Response 回数据给 Central。

The diagram illustrates the sequence of frames for reading a large Attribute Value (e.g., HID report map) from a Peripheral. It shows the following steps:

- Step 1: Read Request (Central to Peripheral)**  
Central sends a Read Request (L2CAP-S frame) with LLID=2, NESN=0, SN=0, MD=0, PDU-Length=7. The L2CAP header includes L2CAP-Length=0x0003 and ChanId=0x0004. The ATT\_Read\_Req field contains Opcode=0x0B, AttHandle=0x0020.
- Step 2: Read Response (Peripheral to Central)**  
Peripheral responds with a Read Response (Empty PDU frame). The L2CAP header includes L2CAP-Length=0xAE00D5 and ChanId=0x0004. The ATT\_Read\_Rsp field contains Opcode=0x0B, AttValue=0x0B 05 01 09 02 A1 01 85 01 09 01 A1 00 05 09 19 01 29 03 15 00 25 01.
- Step 3: Read Blob Request (Central to Peripheral)**  
Central sends a Read Blob Request (L2CAP-S frame) with LLID=2, NESN=1, SN=1, MD=0, PDU-Length=27. The L2CAP header includes L2CAP-Length=0x0017 and ChanId=0x0004. The ATT\_Read\_Blob\_Req field contains Opcode=0x0C, AttHandle=0x0020, ValueOffset=0x0016.
- Step 4: Read Blob Response (Peripheral to Central)**  
Peripheral responds with a Read Blob Response (Empty PDU frame). The L2CAP header includes L2CAP-Length=0xAE0606 and ChanId=0x0004. The ATT\_Read\_Blob\_Rsp field contains Opcode=0x0C, AttHandle=0x0020, ValueOffset=0x0016.
- Step 5: Read Request (Central to Peripheral)**  
Central sends a Read Request (L2CAP-S frame) with LLID=2, NESN=1, SN=1, MD=0, PDU-Length=27. The L2CAP header includes L2CAP-Length=0x0017 and ChanId=0x0004. The ATT\_Read\_Req field contains Opcode=0x0D, PartAttValue=75 01 95 03 81 02 75 05 95 01 81 01 05 01 09 30 09 31 09 38 15 81.
- Step 6: Read Blob Response (Peripheral to Central)**  
Peripheral responds with a Read Blob Response (Empty PDU frame). The L2CAP header includes L2CAP-Length=0x0005 and ChanId=0x0004. The ATT\_Read\_Blob\_Rsp field contains Opcode=0x0C, AttHandle=0x0020, ValueOffset=0x002C.

Figure 3.58: Read Blob Request Read Blob Response

使用下面 API 实现 Read Blob Request

```
ble_sts_t blc_gatt_pushReadBlobRequest(u16 connHandle, u16 attHandle, u16 offset);
```

Read Blob Response 的数据，在 app\_gatt\_data\_handler 函数里可以读取并处理。

#### 34.24.7 Exchange MTU Request, Exchange MTU Response

Exchange MTU Request 和 Exchange MTU Response，详情请参考 Bluetooth Core Specification V5.4, Vol 3, Part F, 3.4.2.1 ATT\_EXCHANGE\_MTU\_REQ/3.4.2.2 ATT\_EXCHANGE\_MTU\_RSP。



如下面所示，Central 和 Peripheral 通过 Exchange MTU Request 和 Exchange MTU Response 获知对方的 MTU size。

Data Type	Data Header					L2CAP Header		ATT_Exchange_MTU_Req		CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	ClientRxMTU	0xC70102	-38	OK

Data Type	Data Header					L2CAP Header		ATT_Exchange_MTU_Rsp		CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	ServerRxMTU	0x1D88E1	-38	OK

Figure 3.59: Exchange MTU Request Exchange MTU Response

当 GATT 层的数据访问过程中出现超过一个 RF 包长度的数据，涉及到 GATT 层分包和拼包时，需要提前和 peer Central/Peripheral 交互双方的 RX MTU size，也就是 MTU size exchange 的过程。MTU size exchange 的目的是为了实现 GATT 层长包数据的收发。

- (1) 用户可以通过注册 GAP event 回调并开启 eventMask：GAP\_EVT\_MASK\_ATT\_EXCHANGE\_MTU 来获取 EffectiveRxMTU，其中：

```
EffectiveRxMTU=min(ClientRxMTU, ServerRxMTU)。
```

本文档“[GAP event](#)”章节会详细介绍 GAP event。

- (2) GATT 层收长包数据的处理。

ServerRxMTU 和 ClientRxMTU 默认为 23，最大 ServerRxMTU/ClientRxMTU 可以支持到和理论值一样（仅受限于 ram 空间）。当应用中需要使用到分包重拼时，使用下面 API 修改 Central 端 RX size：

```
ble_sts_t blc_att_setCentralRxMtuSize(u16 cen_mtu_size);
```

使用下面 API 修改 Peripheral 端 RX size：

```
ble_sts_t blc_att_setPeripheralRxMtuSize(u16 per_mtu_size);
```

返回值列表：

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	操作成功
GATT_ERR_INVALID_PARAMETER	见 SDK 中定义	大于定义的 buffer size，即：mtu_s_rx_fifo 或 mtu_m_rx_fifo

注意：



上述两个 API 设置是 ATT\_Exchange\_MTU\_req/ATT\_Exchange\_MTU\_rsp 命令交互时的 MTU 数值。该数值不能大于实际定义的 buffer size，即变量：mtu\_m\_rx\_fifo[ ] 和 mtu\_s\_rx\_fifo[ ]，这两个 array variable 是在 app\_buffer.c 中定义的。

只要使用上面 API 设置的 MTU 不是默认值 23，连接建立后，SDK 会主动发起 MTU 的交互流程，通过注册 Host 事件 GAP\_EVT\_ATT\_EXCHANGE\_MTU，可以在回调函数中看到 MTU 交互的结果。

#### 34.24.8 Write Request, Write Response

Write Request 和 Write Response，详情请参考 Bluetooth Core Specification V5.4, Vol 3, Part F, 3.4.5.1 ATT\_WRITE\_REQ/3.4.5.2 ATT\_WRITE\_RSP。

Central 发送 Write Request，指定某个 attHandle，并附带相关数据。Peripheral 收到后，找到指定的 Attribute，根据 user 是否设置了回调函数 w 决定数据是使用回调函数 w 来处理还是直接写入对应的 Attribute Value，并回复 Write Response。

下图所示为 Central 向 attHandle 为 0x0016 的 Attribute 写入 Attribute Value 为 0x0001，Peripheral 收到后执行该写操作，并回 Write Response。

Data Type	Data Header					L2CAP Header		ATT_Write_Req			CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue	0xDC8476	-38	OK
	2	0	1	0	9	0x0005	0x0004	0x12	0x0016	01 00			
Data Type	Data Header					CRC	RSSI (dBm)	FCS					
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE00D5	-38	OK					
Data Type	Data Header					CRC	RSSI (dBm)	FCS					
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE0606	-38	OK					
Data Type	Data Header					L2CAP Header		ATT_Write_Rsp			CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode			0xFBDB12	-38	OK
	2	1	1	0	5	0x0001	0x0004	0x13					

Figure 3.60: Write Request Write Response

使用下面 API 实现 Write Request

```
ble_sts_t blc_gatt_pushWriteRequest (u16 connHandle, u16 attHandle, u8 *p, int len);
```

Write Response 的数据，在 app\_gatt\_data\_handler 函数里可以读取并处理。

#### 34.24.9 Write Command

Write Command，详情请参考 Bluetooth Core Specification V5.4, Vol 3, Part F, 3.4.5.3 ATT\_WRITE\_CMD。

Central 发送 Write Command，指定某个 attHandle，并附带相关数据。Peripheral 收到后，找到指定的 Attribute，根据 user 是否设置了回调函数 w 决定数据是使用回调函数 w 来处理还是直接写入对应的 Attribute Value，不回复任何信息。

使用下面 API 实现 Write Command



```
ble_sts_t blc_gatt_pushWriteCommand (u16 connHandle, u16 attHandle, u8 *p, int len);
```

### 34.24.10 Queued Writes

Queued Writes 包含 Prepare Write Request/Response 和 Execute Write Request/Response 等 ATT 协议，详情请参考 Bluetooth Core Specification V5.4, Vol 3, Part F, 3.4.6 Queued writes。

#### 注意：

在使用 Queued Writes 时，需要在初始化的时候调用 API blc\_att\_setPrepareWriteBuffer 来分配 prepare write 的存储 buffer，节省 ram 考虑默认并未进行初始设置。

Prepare Write Request 和 Execute Write Request 可以实现如下两种功能：

- a) 提供长属性值的写入功能。
- b) 允许在一个单独执行的原子操作中写入多个值。

Prepare Write Request 包含 AttHandle、ValueOffset 和 PartAttValue，这和 Read\_Blob\_Req/Rsp 类似。这说明 Client 既可以在队列中准备多个属性值，也可以准备一个长属性值的各个部分。这样，在真正执行准备队列之前，Client 可以确定某属性的所有部分都能写入 Server。

#### 注意：

tl\_ble\_sdk 当前版本仅支持 a) 长属性值写入功能，长属性值最大长度小于等于 244 字节。

如下图所示，Central 向 Peripheral 某个特性写很长的字符串：“I am not sure what a new song”（字节数远远超过 23，使用默认 MTU 情况下）时，首先发送 Prepare Write Request，偏移 0x0000，将“ I am not sure what”部分数据写给 Peripheral，Peripheral 向 Central 回 Prepare Write Response。之后 Central 发送 Prepare Write Request，偏移 0x12，将“ a new song”部分数据写给 Peripheral，Peripheral 向 Central 回 Prepare Write Response。当 Central 将长属性值全部写完成后，发送 Execute Write Request 给 Peripheral，Flags 为 1：表示写立即生效，Peripheral 回复 Execute Write Response，整个 Prepare write 过程结束。

这里我们可以看到 Prepare Write Response 也包含请求中的 AttHandle、ValueOffset 和 PartAttValue。这样做的目的为了数据传递的可靠性。Client 可以对比 Response 和 Request 的字段值，确保准备的数据被正确接收。



Data Type L2CAP-S	Data Header LLID NESN SN MD PDU-Length 2 0 1 0 27	L2CAP Header L2CAP-Length ChanId 0x0017 0x0004	ATT_Prep_Wrt_Rsp Opcode AttHandle ValueOffset PartAttValue 0x17 0x0015 0x0000 49 20 61 6D 20 6E 6F 74 20 73 75 72 65 20 77 68 61 74	
Data Type L2CAP-S	Data Header LLID NESN SN MD PDU-Length 2 0 0 0 20	L2CAP Header L2CAP-Length ChanId 0x0010 0x0004	ATT_Prep_Wrt_Rsp Opcode AttHandle ValueOffset PartAttValue 0x16 0x0015 0x0012 20 61 20 6E 65 77 20 73 6F 6E 67	CRC 0x98D4A6
Data Type Empty PDU	Data Header LLID NESN SN MD PDU-Length 1 1 0 0 0	CRC 0x071388	RSSI (dBm) -54	FCS OK
Data Type Empty PDU	Data Header LLID NESN SN MD PDU-Length 1 1 1 0 0	CRC 0x071E2E	RSSI (dBm) -54	FCS OK
Data Type L2CAP-S	Data Header LLID NESN SN MD PDU-Length 2 0 1 0 20	L2CAP Header L2CAP-Length ChanId 0x0010 0x0004	ATT_Prep_Wrt_Rsp Opcode AttHandle ValueOffset PartAttValue 0x17 0x0015 0x0012 20 61 20 6E 65 77 20 73 6F 6E 67	CRC 0xFF79B4
Data Type L2CAP-S	Data Header LLID NESN SN MD PDU-Length 2 0 0 0 6	L2CAP Header L2CAP-Length ChanId 0x0002 0x0004	ATT_Exec_Wrt_Req Opcode Flags 0x18 0x01	CRC 0x24D166
Data Type Empty PDU	Data Header LLID NESN SN MD PDU-Length 1 1 0 0 0	CRC 0x071388	RSSI (dBm) -54	FCS OK
Data Type Empty PDU	Data Header LLID NESN SN MD PDU-Length 1 1 1 0 0	CRC 0x071E2E	RSSI (dBm) -54	FCS OK
Data Type Empty PDU	Data Header LLID NESN SN MD PDU-Length 1 0 0 0 0	CRC 0x07155B	RSSI (dBm) -54	FCS OK
Data Type L2CAP-S	Data Header LLID NESN SN MD PDU-Length 2 0 1 0 5	L2CAP Header L2CAP-Length ChanId 0x0001 0x0004	ATT_Exec_Wrt_Req Opcode 0x19	CRC 0x430D57
			RSSI (dBm) -54	FCS OK

Figure 3.61: Write Long Characteristic Values 示例

### 34.24.11 Handle Value Notification

Handle Value Notification, 详情请参考 Bluetooth Core Specification V5.4, Vol 3, Part F, 3.4.7.1 ATT\_HANDLE\_VALUE\_NTF。

Parameter	Size (octets)	Description
Attribute Opcode	1	0x1B = Handle Value Notification
Attribute Handle	2	The handle of the attribute
Attribute Value	0 to (ATT_MTU-3)	The current value of the attribute

Table 3.34: Format of Handle Value Notification

Figure 3.62: Bluetooth Core Specification 中 Handle Value Notification

上图所示为 Bluetooth Core Specification 中 Handle Value Notification 的格式。

tl\_ble\_sdk 提供了 API，用于某个 Attribute 的 Handle Value Notification。user 调用这个 API 以将自己需要 notify 的数据 push 到底层的 BLE 软件 fifo，协议栈会在最近的收发包 interval 时将软件 fifo 的数据 push 到硬件 fifo，最终通过 RF 发送出去。

```
ble_sts_t blc_gatt_pushHandleValueNotify(u16 connHandle, u16 attHandle, u8 *p, int len);
```

connHandle 为对应 Connection state 的 connHandle，attHandle 为对应 Attribute 的 attHandle，p 为要发送的连续内存数据的头指针，len 指定发送的数据的字节数。该 API 支持自动拆包功能（根据 EffectiveMaxTxOctets



做分包处理，即链路层 RF RX/TX 最大收发字节数的较小值，DLE 可能会修改该值，默认为 27），可将一个很长的数据拆成多个 BLE RF packet 发送出去，所以 len 可以支持很大。

Link Layer 在 Conn state 时，一般情况下直接调用该 API 可以成功 push 数据到底层软件 fifo，但也存在一些特殊情况会导致该 API 调用失败，根据返回值 ble\_sts\_t 可以了解对应的错误原因。

调用该 API 时，建议用户检查返回值的是否为 BLE\_SUCCESS，若不为 BLE\_SUCCESS，则需要等待一段时间后再次 push。

返回值列表如下：

ble_sts_t	Value	ERR reason
BLE_SUCCESS	0	操作成功
GAP_ERR_INVALID_PARAMETER	0xCO	无效参数
SMP_ERR_PAIRING_BUSY	0xA1	处于配对阶段
GATT_ERR_DATA_LENGTH_EXCEED_MTU_SIZE	0xB5	len 大于 ATT_MTU-3，要发送的数据长度超出了 ATT 层支持的最大数据长度 ATT_MTU
LL_ERR_CONNECTION_NOT_ESTABLISH	0x80	Link Layer 处于 None Conn state
LL_ERR_ENCRYPTION_BUSY	0x82	处于加密阶段，不能发送数据
LL_ERR_TX_FIFO_NOT_ENOUGH	0x81	有大数据量任务在运行，软件 Tx fifo 不够用
GATT_ERR_DATA_PENDING_DUE_TO_SERVICE_DISCOVERY_BUSY	0xB4	处于遍历服务阶段，不能发数据

#### 34.24.12 Handle Value Indication

Handle Value Indication，详情请参考 Bluetooth Core Specification V5.4, Vol 3, Part F, 3.4.7.2 ATT\_HANDLE\_VALUE\_IND。

Parameter	Size (octets)	Description
Attribute Opcode	1	0x1D = Handle Value Indication
Attribute Handle	2	The handle of the attribute
Attribute Value	0 to (ATT_MTU-3)	The current value of the attribute

Table 3.35: Format of Handle Value Indication

Figure 3.63: Handle Value Indication in BLE Spec

上图所示为 Bluetooth Core Specification 中 Handle Value Indication 的格式。

tl\_ble\_sdk 提供 API，用于某个 Attribute 的 Handle Value Indication。user 调用这个 API 以将自己需要 indicate



的数据 push 到底层的 BLE 软件 fifo，协议栈会在最近的收发包 interval 时将软件 fifo 的数据 push 到硬件 fifo，最终通过 RF 发送出去。

```
ble_sts_t blc_gatt_pushHandleValueIndicate (u16 connHandle, u16 attHandle, u8 *p, int len);
```

connHandle 为对应 Connection state 的 connHandle, attHandle 为对应 Attribute 的 attHandle, p 为要发送的连续内存数据的头指针, len 指定发送的数据的字节数。该 API 支持自动拆包功能（根据 EffectiveMaxTxOctets 做分包处理，即链路层 RF RX/TX 最大收发字节数的较小值，DLE 可能会修改该值，默认为 27，下文将介绍其替换 API，见备注），可将一个很长的数据拆成多个 BLE RF packet 发送出去，所以 len 可以支持很大。

Bluetooth Core Specification 里规定了每一个 indicate 的数据，都要等到 client 的 confirm 才能认为 indicate 成功，未成功时不能发送下一个 indicate 数据。

Link Layer 在 Conn state 时，一般情况下直接调用该 API 可以成功 push 数据到底层软件 fifo，但也存在一些特殊情况会导致该 API 调用失败，根据返回值 ble\_sts\_t 可以了解对应的错误原因。

调用该 API 时，建议用户检查返回值的是否为 BLE\_SUCCESS，若不为 BLE\_SUCCESS，则需要等待一段时间后再次 push。

返回值列表如下：

ble_sts_t	Value	ERR reason
BLE_SUCCESS	0	操作成功
GAP_ERR_INVALID_PARAMETER	0xC0	无效参数
SMP_ERR_PAIRING_BUSY	0xA1	处于配对阶段
GATT_ERR_DATA_LENGTH_EXCEED_MTU_SIZE	0xB5	len 大于 ATT_MTU-3，要发送的数据长度超出了 ATT 层支持的最大数据长度 ATT_MTU
LL_ERR_CONNECTION_NOT_ESTABLISH	0x80	Link Layer 处于 None Conn state
LL_ERR_ENCRYPTION_BUSY	0x82	处于加密阶段，不能发送数据
LL_ERR_TX_FIFO_NOT_ENOUGH	0x81	有大数据量任务在运行，软件 Tx fifo 不够用
GATT_ERR_DATA_PENDING_DUE_TO_SERVICE_DISCOVERY_BUSY	0xB4	处于遍历服务阶段，不能发数据
GATT_ERR_PREVIOUS_INDICATE_DATA_HAS_NOT_CONFIRMED	0xB1	前一个 indicate 数据还没有被 Central 确认

#### 34.24.13 Handle Value Confirmation

Handle Value Confirmation，详情请参考 Bluetooth Core Specification V5.4, Vol 3, Part F, 3.4.7.3 ATT\_HANDLE\_VALUE\_CFM。

应用层每调用一次 blc\_gatt\_pushHandleValueIndicate，向 Central 发送 indicate 数据后，Central 会回复一个 confirm，表示对这个数据的确认，然后 Peripheral 才可以继续发送下一个 indicate 数据。



Parameter	Size (octets)	Description
Attribute Opcode	1	0x1E = Handle Value Confirmation

Table 3.36: Format of Handle Value Confirmation

Figure 3.64: BLE 中 Handle Value Confirmation

从上图中可以看出，Confirmation 并不指定是对哪一个具体 handle 的确认，对所有不同 handle 上的 indicate 数据都统一回复一个 Confirmation。

为了让应用层了解发送出去的 indicate data 是否已经被 Confirm，用户可以通过注册 GAP event 回调，并开启相应的 eventMask：GAP\_EVT\_GATT\_HANDLE\_VLAUE\_CONFIRM 来获取 Confirm 事件，本文档“[GAP event”小节](#)会详细介绍 GAP event。

#### 34.24.14 blc\_att\_setServerDataPendingTime\_upon\_ClientCmd

tl\_ble\_sdk 底层在 SDP 过程及 SDP 过后的 data pending time（默认 300 ms）时间内不允许 notify 和 indicate 操作。如果 user 需要改变 data pending time，可以使用此 API。

```
void blc_att_setServerDataPendingTime_upon_ClientCmd(u8 num_10ms);
```

参数以 10 ms 为单位，如参数代入 30，则表示  $30 \times 10 \text{ ms}$ ，即 300 ms。

### 3.4.3 GAP

#### 3.4.3.1 GAP 初始化

tl\_ble\_sdk 中，因为 central 和 peripheral 在一个设备中同时扮演，所以在初始化时就不区分 central 和 peripheral 设备了。

初始化函数：

```
void blc_gap_init(void);
```

由前文我们知道，应用层与 Host 的数据交互不通过 GAP 来访问控制，协议栈在 ATT、SMP 和 L2CAP 都提供了相关接口，可以和应用层直接交互。目前 SDK 的 GAP 层主要处理 host 层上的事件，GAP 初始化主要是注册 host 层事件处理函数入口。

#### 3.4.3.2 GAP Event

GAP event 则是 ATT、GATT、SMP、GAP 等 host 协议层交互过程中产生的事件。从前文我们可以知道，目前 SDK 事件主要分为两大类：Controller event 和 GAP(host) event，其中 controller event 又分为 HCI event 和 LE HCI event。

tl\_ble\_sdk 中新增了 GAP event 处理，主要是协议栈事件分层更加清晰，协议栈处理用户层交互事件更加便捷，特别是 SMP 相关的处理，如 Passkey 的输入，配对结果通知用户等。



如果 user 需要在 App 层接收 GAP event，首先需要注册 GAP event 的 callback 函数，其次需要将对应 event 的 mask 打开。

GAP event 的 callback 函数原型和注册接口分别为：

```
typedef int (*gap_event_handler_t) (u32 h, u8 *para, int n);
void blc_gap_registerHostEventHandler (gap_event_handler_t handler);
```

callback 函数原型中的 u32 h 是 GAP event 标记，底层协议栈多处会用到。

下面列出几个用户可能会用到的事件：

#define GAP_EVT_SMP_PAIRING_BEGIN	0
#define GAP_EVT_SMP_PAIRING_SUCCESS	1
#define GAP_EVT_SMP_PAIRING_FAIL	2
#define GAP_EVT_SMP_CONN_ENCRYPTION_DONE	3
#define GAP_EVT_SMP_TK_DISPALY	4
#define GAP_EVT_SMP_TK_REQUEST_PASSKEY	5
#define GAP_EVT_SMP_TK_REQUEST_OOB	6
#define GAP_EVT_SMP_TK_NUMERIC_COMPARE	7
#define GAP_EVT_ATT_EXCHANGE_MTU	16
#define GAP_EVT_GATT_HANDLE_VLAUE_CONFIRM	17

callback 函数原型中 para 和 n 表示 event 的数据和数据长度，下文将详细说明以上列出的 GAP event。User 可参考 demo code 中如下用法以及 app\_host\_event\_callback 函数的具体实现。

```
blc_gap_registerHostEventHandler( app_host_event_callback );
```

GAP event 需要通过下面的 API 来打开 mask。

```
void blc_gap_setEventMask(u32 evtMask);
```

eventMask 的定义也对应上面给出一些，其他的 event mask 用户可以在 ble/gap/gap\_event.h 中查到。

#define GAP_EVT_MASK_SMP_PAIRING_BEGIN	(1<<GAP_EVT_SMP_PAIRING_BEGIN)
#define GAP_EVT_MASK_SMP_PAIRING_SUCCESS	(1<<GAP_EVT_SMP_PAIRING_SUCCESS)
#define GAP_EVT_MASK_SMP_PAIRING_FAIL	(1<<GAP_EVT_SMP_PAIRING_FAIL)
#define GAP_EVT_MASK_SMP_CONN_ENCRYPTION_DONE	(1<<GAP_EVT_SMP_CONN_ENCRYPTION_DONE)
#define GAP_EVT_MASK_SMP_TK_DISPALY	(1<<GAP_EVT_SMP_TK_DISPALY)
#define GAP_EVT_MASK_SMP_TK_REQUEST_PASSKEY	(1<<GAP_EVT_SMP_TK_REQUEST_PASSKEY)
#define GAP_EVT_MASK_SMP_TK_REQUEST_OOB	(1<<GAP_EVT_SMP_TK_REQUEST_OOB)
#define GAP_EVT_MASK_SMP_TK_NUMERIC_COMPARE	(1<<GAP_EVT_SMP_TK_NUMERIC_COMPARE)
#define GAP_EVT_MASK_ATT_EXCHANGE_MTU	(1<<GAP_EVT_ATT_EXCHANGE_MTU)
#define GAP_EVT_MASK_GATT_HANDLE_VLAUE_CONFIRM	(1<<GAP_EVT_GATT_HANDLE_VLAUE_CONFIRM)

若 user 未通过该 API 设置 GAP event mask，那么当 GAP 相应的 event 产生时将不会通知应用层。

**注意：**



以下论述 GAP event 时，均设定注册了 GAP event 回调，且开启了对应的 eventMask。

### (1) GAP\_EVT\_SMP\_PAIRING\_BEGIN

事件触发条件：当 Peripheral 和 Central 刚刚连接进入 connection state，Peripheral 发送 SM\_Security\_Req 命令后，Central 发送 SM\_Pairing\_Req 请求开始配对，Peripheral 收到这个配对请求命令时，触发该事件，表示配对开始。

Data Type	Data Header				L2CAP Header		SM_Security_Req							
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AuthReq					
	2	1	0	0	6	0x0002	0x0006	0x0B	01					
Data Type	Data Header				L2CAP Header		SM_Pairing_Req							
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	IOCap	OOBDataFlag	AuthReq	MaxEncKeySize	InitKeyDist	RespKeyDist
	2	1	1	0	11	0x0007	0x0006	0x01	0x03	0x00	0x01	0x10	0x02	0x03

Figure 3.65: SMP\_PAIRING\_BEGIN 事件触发条件

数据长度 n: 4。

回传指针 p: 指向一片内存数据，对应如下结构体：

```
typedef struct {
    u16 connHandle;
    u8 secure_conn;
    u8 tk_method;
} gap_smp_pairingBeginEvt_t;
```

connHandle 表示当前连接句柄。

secure\_conn 为 1 表示使用安全加密特性 (LE Secure Connections)，否则将使用 LE legacy pairing。

tk\_method 表示接下来配对使用什么样的 TK 值方式：例如 JustWorks、PK\_Init\_Dsplay\_Resp\_Input、PK\_Resp\_Dsplay\_Init\_Input，Numric\_Comparison 等。

### (2) GAP\_EVT\_SMP\_PAIRING\_SUCCESS

事件触发条件：配对整个流程正确完成时产生该事件，该阶段即为 LE 配对阶段之密钥分发阶段 3 (Key Distribution, Phase 3)，如果有密钥需要分发，则等待双方密钥分发完成后触发配对成功事件，否则直接触发配对成功事件。

数据长度 n: 4。

回传指针 p: 指向一片内存数据，对应如下结构体：

```
typedef struct {
    u16 connHandle;
    u8 bonding;
    u8 bonding_result;
} gap_smp_pairingSuccessEvt_t;
```

connHandle 表示当前连接句柄。

bonding 为 1 表示启用 bonding 功能，否则不启用。



bonding\_result 表示 bonding 的结果：如果没有开启 bonding 功能，则为 0，如果开启了 bonding 功能，则还需要检查加密 Key 是否被正确的存储在 FLASH 中，存储成功为 1，否则为 0。

### (3) GAP\_EVT\_SMP\_PAIRING\_FAIL

事件触发条件：由于 Peripheral 或 Central 其中一个不符合标准配对流程，或者通信中出现报错等异常原因导致配对流程终止。

数据长度 n: 2。

回传指针 p：指向一片内存数据，对应如下结构体：

```
typedef struct {
    u16 connHandle;
    u8 reason;
} gap_smp_pairingFailEvt_t;
```

connHandle 表示当前连接句柄。

reason 表示配对失败的原因，这里列出几个常见的配对失败原因值，其他配对失败原因值我们可以参考 SDK 目录下的“stack/ble/smp/smp\_const.h”文件。

配对失败值的具体含义，详情请参考 Bluetooth Core Specification V5.4, Vol 3, Part H, 3.5.5 Pairing Failed。

```
#define PAIRING_FAIL_REASON_CONFIRM_FAILED      0x04
#define PAIRING_FAIL_REASON_PAIRING_NOT_SUPPORTED 0x05
#define PAIRING_FAIL_REASON_DHKEY_CHECK_FAIL     0x0B
#define PAIRING_FAIL_REASON_NUMERIC_FAILED        0x0C
#define PAIRING_FAIL_REASON_PAIRING_TIMEOUT       0x80
#define PAIRING_FAIL_REASON_CONN_DISCONNECT       0x81
```

### (4) GAP\_EVT\_SMP\_CONN\_ENCRYPTION\_DONE

事件触发条件：Link Layer 加密完成时（Link Layer 收到 Central 发的 start encryption response）触发。

数据长度 n: 3。

回传指针 p：指向一片内存数据，对应如下结构体：

```
typedef struct {
    u16 connHandle;
    u8 re_connect; //1: re_connect, encrypt with previous distributed LTK; 0: pairing ,
    ↵ encrypt with STK
} gap_smp_connEncDoneEvt_t;
```

connHandle 表示当前连接句柄。

re\_connect 为 1 表示快速回连（将使用之前分发的 LTK 加密链路），若该值为 0 则表示当前加密是第一次加密。

### (5) GAP\_EVT\_SMP\_TK\_DISPALY



事件触发条件: Peripheral 收到 Central 发送的 Pairing\_Req 后, 根据对端设备的配对参数和本地设备的配对参数配置, 我们就可以知道接下来配对使用什么样的 TK (pincode) 值方式。如果启用的是 PK\_Resp\_Dsly\_Init\_Input (即: Peripheral 端显示 6 位 pincode 码, Central 端负责输入 6 位 pincode 码) 方式, 则会立即触发。

数据长度 n: 4。

回传指针 p: 指向一个 u32 型变量 tk\_set, 该值即为 Peripheral 需要通知应用层的 6 位 pincode 码, 应用层需要显示该 6 位码值, 参考代码如下:

```
case GAP_EVT_SMP_TK_DISPALY:  
{  
    char pc[7];  
    u32 pinCode = *(u32*)para;  
    sprintf(pc, "%d", pinCode);  
    printf("TK display:%s\n", pc);  
}  
break;
```

pincode 可以通过以下 API 在初始化时进行设置, 如设置为 123456:

```
blc_smp_setDefaultPinCode(123456);
```

如果设置值为 0, 或者没有调用以上 API 进行设置, 则 Pincode 值是随机的。

用户将 Peripheral 上看到的 6 位 pincode 码输入到 Central 设备上 (如手机), 完成 TK 输入, 配对流程得以继续执行。如果用户输入 pincode 错误或者点击取消, 则配对流程失败。

#### (6) GAP\_EVT\_SMP\_TK\_REQUEST\_PASSKEY

事件触发条件: 当 Peripheral 设备启用 Passkey Entry 方式时, 且使用的 PK\_Init\_Dsly\_Resp\_Input 或者 PK\_BOTH\_INPUT 配对方式时, 会触发该事件, 通知用户需要输入 TK 值。用户在收到该事件后就需要通过 IO 输入 TK 值 (超时 30s 如果还未输入则配对失败), 输入 TK 值的 API: blc\_smp\_setTK\_by\_PasskeyEntry 在“[SMP 参数配置”章节](#)有说明。

数据长度 n: 0。

回传指针 p: NULL。

#### (7) GAP\_EVT\_SMP\_TK\_REQUEST\_OOB

事件触发条件: 当 Peripheral 设备启用传统配对 OOB 方式时, 会触发该事件, 通知用户需要通过 OOB 方式输入 16 位 TK 值。用户在收到该事件后就需要通过 IO 输入 16 位 TK 值 (超时 30s 如果还未输入则配对失败), 输入 TK 值的 API: blc\_smp\_setTK\_by\_OOB 在“[SMP 参数配置”章节](#)有说明。

数据长度 n: 0。

回传指针 p: NULL。

#### (8) GAP\_EVT\_SMP\_TK\_NUMERIC\_COMPARE

事件触发条件: Peripheral 收到 Central 发送的 Pairing\_Req 后, 根据对端设备的配对参数和本地设备的配对参数配置我们就可以知道接下来配对使用什么样的 TK (pincode) 值方式, 如果启用的是 Numeric\_Comparison 方式, 则会立即触发。(Numeric\_Comparison 方式即数值比较, 属于 smp4.2 安全加密, Central 和 Peripheral



设备均会弹出显示 6 位 pincode 码以及“YES”和“NO”对话框，用户需要检查两端设备显示的 pincode 是否一致，并需要两端分别确认是否点击“YES”以确认 TK 校验是否通过)。

数据长度 n: 4。

回传指针 p: 指向一个 u32 型变量 pinCode，该值即为 Peripheral 需要通知应用层的 6 位 pincode 码，应用层需要显示该 6 位码值，并提供“YES”和“NO”的确认机制。

#### (9) GAP\_EVT\_ATT\_EXCHANGE\_MTU

事件触发条件：无论是 Central 端发送 Exchange MTU Request，Peripheral 回复 Exchange MTU Response，还是 Peripheral 端发送 Exchange MTU Request，Central 回复 Exchange MTU Response，两种情况下均会触发。

数据长度 n: 6。

回传指针 p: 指向一片内存数据，对应如下结构体：

```
typedef struct {
    u16 connHandle;
    u16 peer_MTU;
    u16 effective_MTU;
} gap_gatt_mtuSizeExchangeEvt_t;
```

connHandle 表示当前连接句柄。

peer\_MTU 表示对端的 RX MTU 值。

effective\_MTU = min(CleintRxMTU, ServerRxMTU), CleintRxMTU 表示客户端的 RX MTU size 值, ServerRxMTU 表示服务端的 RX MTU size 值。Central 和 Peripheral 交互了彼此的 MTU size 后，取两者最小值作为彼此交互的最大 MTU size 值。

#### (10) GAP\_EVT\_GATT\_HANDLE\_VALUE\_CONFIRM

事件触发条件：应用层每调用一次 bls\_att\_pushIndicateData (或者调用 blc\_gatt\_pushHandleValueIndicate)，向 Central 发送 indicate 数据后，Central 会回复一个 confirm，表示对这个数据的确认，Peripheral 收到该 confirm 时触发。

数据长度 n: 0。

回传指针 p: NULL。

### 3.4.4 SMP

Security Manager (SM) 为 LE 设备提供加密所需要的各種 Key，确保数据的机密性。加密链路可以避免第三方“攻击者”拦截、破译或者篡改空中数据原始内容。SMP 详细内容请参考 Bluetooth Core Specification V5.4, Vol 3, Part H: Security Manager Specification。

#### 3.4.4.1 SMP 安全等级

Bluetooth Core Specification V4.2 新增了一种称作安全连接 (LE Secure Connections) 的配对方式，进一步增强了安全性，而此前的配对方式，我们统称传统配对 (LE Legacy Pairing)。



Telink BLE Multiple Connection SDK 提供以下 4 个安全等级，参考 Bluetooth Core Specification V5.4, Vol 3, Part C, 10.2 LE security modes：

- a) No authentication and no encryption (LE security Mode 1 Level 1)
- b) Unauthenticated pairing with encryption (LE security Mode 1 Level 2)
- c) Authenticated pairing with encryption (LE security Mode 1 Level 3)
- d) Authenticated LE Secure Connections (LE security Mode 1 Level 4)

#### 注意：

- 所有连接全部支持到最高安全级别，主从可以配置不同的安全级别；
- 当前不支持不同连接配置为不同的安全级别；
- 本端设备设定的安全级别只表示本端设备可能达到的最高安全级别，想要达到设定的安全级别跟两个因素有关：
  - (a) peer device 设定能支持的最高安全级别  $\geq$  local device 设定能支持的最高安全级别；
  - (b) local device 和 peer device 按照各自设定的 SMP 参数正确处理完配对整个流程（如果存在配对的话）。

举例来说，用户设置 Peripheral 端能够支持的最高安全等级是 Mode 1 Level 3，但是连接 Peripheral 的 Central 设置为不支持配对加密（最高只支持 Mode 1 Level 1），那么连接后 Peripheral 和 Central 不会进行配对流程，Peripheral 实际使用的安全级别是 Mode 1 Level 1。

使用以下 API 设置 local device 能支持的最高安全等级。

```
void blc_smp_setSecurityLevel(le_security_mode_level_t mode_level);
void blc_smp_setSecurityLevel_central(le_security_mode_level_t mode_level);
void blc_smp_setSecurityLevel_periph(le_security_mode_level_t mode_level);
```

#### 说明：

在 tl\_ble\_sdk 中，配置 SMP 相关参数的 API 如没有特别说明，都会有如下 3 种配置形式：

- 统一配置 Central role 和 Peripheral role 参数的 API(...);
- 单独配置所有 Central role 参数的 API\_Central(...);
- 单独配置所有 Peripheral role 参数的 API\_Peripheral(...);

### 3.4.4.2 SMP 参数配置

在调用 GAP 的初始化时，会初始化 SMP，并将 SMP 的参数初始化为默认值：

- 默认支持的最高安全等级：Unauthenticated\_Paring\_with\_Encryption，即 Mode 1 Level 2；
- 默认绑定模式：Bondable\_Mode（参考 blc\_smp\_setBondingMode() API 说明）；
- 默认 IO 能力：IO\_CAPABILITY\_NO\_INPUT\_NO\_OUTPUT；
- 默认配对方式：Legacy Pairing Just Works。



初始化完成后，先通过 SMP 参数配置的 API 配置 SMP 参数，再通过以下 API 将应用层配置的参数带入底层进行初始配置。

```
void blc_smp_smpParamInit(void);
```

下面介绍 SMP 参数配置的相关 API。

```
void blc_smp_setPairingMethods(pairing_methods_t method); //Peripheral()/_Central()
```

该套 API 用于配置 SMP 配对方式，Legacy 或 Secure Connections。

**注意：**

Secure Connection 的安全配对方式，需要 MTU>=65。

```
void blc_smp_setIoCapability(pairing_methods_t method); //Peripheral()/_Central()
```

该套 API 用于配置 SMP IO 能力（见下图），决定 Key 产生的方法，参考 Bluetooth Core Specification V5.4, Vol 3, Part H, 2.3.5.1 Selecting Key Generation Method。

```
/* H: Initiator Capabilities
 * V: Responder Capabilities
 * See the Core_v5.0(Vol_3/Part H/2.3.5.1) for more information.
static const stk_generationMethod_t gen_method_legacy[5 /*Responder IOCap*/][5 /*Initiator IOCap*/] = {
    { JustWorks, JustWorks, PK_Resp_Dsplay_Init_Input, JustWorks, PK_Resp_Dsplay_Init_Input },
    { JustWorks, JustWorks, PK_Resp_Dsplay_Init_Input, JustWorks, PK_Resp_Dsplay_Init_Input },
    { PK_Init_Dsplay_Resp_Input, PK_Init_Dsplay_Resp_Input, PK_BOTH_INPUT, JustWorks, PK_Init_Dsplay_Resp_Input },
    { JustWorks, JustWorks, JustWorks, JustWorks, JustWorks },
    { PK_Init_Dsplay_Resp_Input, PK_Init_Dsplay_Resp_Input, PK_Resp_Dsplay_Init_Input, JustWorks, PK_Init_Dsplay_Resp_Input },
};

static const stk_generationMethod_t gen_method_sc[5 /*Responder IOCap*/][5 /*Initiator IOCap*/] = {
    { JustWorks, JustWorks, PK_Resp_Dsplay_Init_Input, JustWorks, PK_Resp_Dsplay_Init_Input },
    { JustWorks, Numric_Comparison, PK_Resp_Dsplay_Init_Input, JustWorks, Numric_Comparison },
    { PK_Init_Dsplay_Resp_Input, PK_Init_Dsplay_Resp_Input, PK_BOTH_INPUT, JustWorks, PK_Init_Dsplay_Resp_Input },
    { JustWorks, JustWorks, JustWorks, JustWorks, JustWorks },
    { PK_Init_Dsplay_Resp_Input, Numric_Comparison, PK_Resp_Dsplay_Init_Input, JustWorks, Numric_Comparison },
};
```

**Figure 3.66:** 根据不同 IO 能力映射 KEY 产生方法

```
void blc_smp_enableAuthMITM(int MITM_en); //Peripheral()/_Central()
```

该套 API 用于配置 SMP 的 MITM (Man in the Middle) flag，用于提供 Authentication，安全级别在 Mode 1 Level 3 及以上时，要求该参数为 1。其中参数 MITM\_en 的值为 0 对应失能；1 对应使能。

```
void blc_smp_enableOobAuthentication(int OOB_en); //Peripheral()/_Central()
```

该套 API 用于配置 SMP 的 OOB flag，需要安全级别在 Mode 1 Level 3 及以上。其中参数 OOB\_en 的值为 0 对应失能；1 对应使能。

设备会根据本地设备和对端设备的 OOB 及 MITM flag 决定使用 OOB 方式还是根据 IO 能力决定选择什么样的配对方式，参考 Bluetooth Core Specification V5.4, Vol 3, Part H, 2.3.5.1 Selecting Key Generation Method。



```
void blc_smp_setBondingMode(bonding_mode_t mode); //Peripheral()/_Central()
```

该套 API 用于配置是否将 SMP 过程产生的 Key 存在 Flash 中，如果设置为 Bondable\_Mode，用户就可以利用 SMP 信息进行自动回连，回连时不会重新进行配对；如果设置为 Non\_Bondable\_Mode，则产生的 Key 不会存储在 Flash 中，断线之后无法进行自动回连，需要进行重新配对。

```
void blc_smp_enableKeypress(int keyPress_en); //Peripheral()/_Central()
```

该套 API 用于配置是否需使能 Key Press 功能。其中参数 keyPress\_en 的值为 0 对应失能；1 对应使能。

```
void blc_smp_setSecurityParameters(bonding_mode_t mode, int MITM_en, pairing_methods_t
    ↵ method, int OOB_en, int keyPress_en, io_capability_t ioCapablility); //Peripheral()
    ↵ _Central()
```

该套 API 用于整体性配置前述 SMP 参数，各参数和以上 API 分别具有如下对应关系：

parameter	API
mode	void blc_smp_setBondingMode(bonding_mode_t mode);
MITM_en	void blc_smp_enableAuthMITM(int MITM_en);
method	void blc_smp_setPairingMethods(pairing_methods_t method);
OOB_en	void blc_smp_enableOobAuthentication(int OOB_en);
keyPress_en	void blc_smp_enableKeypress(int keyPress_en);
ioCapablility	void blc_smp_setIoCapability(pairing_methods_t method);

```
void blc_smp_setEcdhDebugMode(ecdh_keys_mode_t mode); //periph()/_central()
```

该套 API 用于配置 Security Connections 是否启用椭圆加密密匙的 Debug 密钥对。安全连接配对情况下使用了椭圆加密算法，可以有效避免窃听，但用户无法通过 sniffer 工具解析 BLE 空中包，所以 Bluetooth Core Specification 给出了一组用于 Debug 的椭圆加密私钥/公钥对，只要开启这个模式，部分 BLE sniffer 工具就可以用这个已知的密钥去解密链路。

#### 注意：

Peripheral 和 Central 仅允许一方的密钥配置为 Debug 密钥对，否则连接不具有安全性，失去了配对的意义，协议规定其非法。

```
void blc_smp_setDefaultPinCode(u32 pinCodeInput); //periph()/_central()
```

该套 API 用于配置 Passkey Entry 或 Numeric Comparison 配对方式下 Display 设备显示的默认 Pincode。参数范围在 [0,999999] 之内。



```
u8 blc_smp_setTK_by_PasskeyEntry (u16 connHandle, u32 pinCodeInput); //connHandle 区分连接链路
```

该 API 用于在 Passkey Entry 配对方式下 Input 设备输入 TK 值。返回值 1 代表设置成功，0 代表当前没有要求 Input 设备输入 TK 值。

#### 说明：

这里解释一下 TK, Passkey, Pincode 三者之间的关系，TK (Temporary Key)，临时密钥，作为 SMP 过程中最基础的原始密钥，其产生方式有多种：如 Just Works 默认产生 TK=0；Passkey Entry 方式输入 TK 的值，这个值在应用层被称为 Pincode；OOB 方式通过 OOB 数据生成 TK。可以简单理解为 Pincode 产生 Passkey，Passkey 产生 TK，只不过，这个“产生”并不一定改变其值。

```
u8 blc_smp_setTK_by_OOB (u16 connHandle, u8 *oobData); //connHandle 区分连接链路
```

该 API 用于设置 OOB 配对方式下设备的 OOB 数据。参数 oobData 表示需要设置的 16 位 OOB 数据数组的头指针。返回值 1 代表设置成功，0 代表当前没有要求 Input 设备输入 TK 值。

```
u8 blc_smp_isWaitingToSetTK(u16 connHandle); //connHandle 区分连接链路
```

该 API 用于获取 Passkey Entry 或 OOB 配对方式下，Input 设备是否等待 TK 输入。返回 1 表示等待输入。

```
void blc_smp_setNumericComparisonResult(u16 connHandle, bool YES_or_NO); //connHandle 区分连接链路
```

该 API 用于在 Security Connections 下 Numeric Comparison 配对方式下设置设备输入的 YES 或 NO。当用户确认显示的 6 位数值和对端一致时，可以输入 1 ("YES")，不一致则输入 0 ("NO")。

```
u8 blc_smp_isWaitingToCfmNumericComparison(u16 connHandle); //connHandle 区分连接链路
```

该 API 用于获取 Security Connections 下 Numeric Comparison 配对方式下，设备是否等待输入 Yes or No。返回 1 表示等待输入。

```
int blc_smp_isPairingBusy(u16 connHandle); //connHandle 区分连接链路
```

该 API 用于查询连接是否正在配对中。返回值 0 表示不在配对，1 表示正在配对中。

### 3.4.4.3 SMP 流程配置

- SMP 安全请求 (Security Request) 只有 Peripheral 可以发送，用于主动请求对端 Central 进行配对流程，是 SMP 的可选流程。
- SMP 配对请求 (Pairing Request) 只有 Central 可以发送，用于通知 Peripheral 开始配对流程。

#### 344.3.1 SMP 安全请求



```
blc_smp_configSecurityRequestSending(secReq_cfg newConn_cfg, secReq_cfg reConn_cfg, u16
↪ pending_ms);
```

该 API 用于灵活地配置 Peripheral 发送 Security Request 的时机。

**注意：**

只有连接建立之前调用才有效，建议在初始化时配置。

枚举类型 secReq\_cfg 定义如下：

```
typedef enum {
    SecReq_NOT_SEND = 0,           //连接建立后, Peripheral 不会主动发送 Security Request
    SecReq_IMM_SEND = BIT(0),      //连接建立后, Peripheral 会立即发送 Security Request
    SecReq_PEND_SEND = BIT(1),      //连接建立后, Peripheral 等待 pending_ms (单位毫秒) 后再决定
↪ 是否发送 Security Request
}secReq_cfg;
```

**newConn\_cfg:** 用于配置新的连接。如果 Peripheral 配置为 SecReq\_PEND\_SEND，且在 pending\_ms 前就收到 Central 的 Pairing Request 包，则不会再发送 Security Request。

**reConn\_cfg:** 用于配置回连。配对绑定过的设备，下次再连接的时候（即回连），Central 有时候不一定会主动发起 LL\_ENC\_REQ 来加密链路，此时如果 Peripheral 发 Security Request 可以触发 Central 加密链路。如果 Peripheral 配置为 SecReq\_PEND\_SEND，且在 pending\_ms 之前已经收到 Central 的 LL\_ENC\_REQ 包，则不会再发送 Security Request。

**pending\_ms:** 当 newConn\_cfg 和 reConn\_cfg 任何一项配置为 SecReq\_PEND\_SEND 时，该参数才有作用。

### 344.3.2 SMP 配对请求

```
void blc_smp_configPairingRequestSending( PairReq_cfg newConn_cfg, PairReq_cfg reConn_cfg);
```

该 API 用于灵活地配置 Central 发送 Pairing Request 的时机。

**注意：**

只能在连接之前调用，建议在初始化时配置。

枚举类型 PairReq\_cfg 定义如下：

```
typedef enum {
    PairReq_SEND_upon_SecReq = 0,   // Central 发送 Pairing Request 依赖于收到 Peripheral 发送的
↪ Security Request
    PairReq_AUTO_SEND = 1,          // Central 一经连接便会自动发送 Pairing Request
}PairReq_cfg;
```



### 3.4.4.4 SMP 配对方法

SMP 配对方法主要围绕 SMP 四个安全等级的配置展开。

#### 3.4.4.1 Mode 1 Level 1

设备不支持加密配对，即禁用 SMP 功能，初始化配置：

```
blc_smp_setSecurityLevel(No_Security);
```

#### 3.4.4.2 Mode 1 Level 2

设备最高支持 Unauthenticated\_Paring\_with\_Encryption，如 Legacy Pairing 和 Secure Connections 配对方式下的 Just Works 配对模式。

- LE Legacy Just works 的初始化配置：

```
//blc_smp_setPairingMethods(LE_Legacy_Pairing);      //Default  
//blc_smp_setSecurityLevel_Central(Unauthenticated_Pairing_with_Encryption);    //Default  
blc_smp_smpParamInit();
```

- LE Security Connections Just works 的初始化配置：

```
blc_smp_setPairingMethods(LE_Secure_Connection);  
blc_smp_smpParamInit();
```

#### 3.4.4.3 Mode 1 Level 3

设备最高支持 Authenticated pairing with encryption，如 Legacy Pairing 的 Passkey Entry、Out of Band。该级别需要设备支持 Authentication，Authentication 能确保配对双方身份的合法性。

- LE Legacy Passkey Entry 方式 Display 设备的初始化配置：

```
blc_smp_setSecurityLevel(Authenticated_Pairing_with_Encryption);  
blc_smp_enableAuthMITM(1);  
blc_smp_setIoCapability(IO_CAPABILITY_DISPLAY_ONLY);  
//blc_smp_setDefaultPinCode(123456);  
blc_smp_smpParamInit();
```

或

```
blc_smp_setSecurityLevel(Authenticated_Pairing_with_Encryption);  
blc_smp_setSecurityParameters(Bondable_Mode, 1, LE_Legacy_Pairing, 0, 0,  
    IO_CAPABILITY_DISPLAY_ONLY);  
blc_smp_smpParamInit();
```



这里涉及到显示 TK 的 GAP event: GAP\_EVT\_SMP\_TK\_DISPALY, 请参考[“GAP event”章节](#)。

- LE Legacy Passkey Entry 方式 Input 设备的初始化配置:

```
blc_smp_setSecurityLevel(Authenticated_Pairing_with_Encryption);
blc_smp_enableAuthMITM(1);
blc_smp_setIoCapability(IO_CAPABILITY_KEYBOARD_ONLY);
blc_smp_smpParamInit();
```

或

```
blc_smp_setSecurityLevel(Authenticated_Pairing_with_Encryption);
blc_smp_setSecurityParameters(Bondable_Mode, 1, LE_Legacy_Pairing, 0, 0,
    ↳ IO_CAPABILITY_KEYBOARD_ONLY);
blc_smp_smpParamInit();
```

这里涉及到请求 TK 的 GAP event: GAP\_EVT\_SMP\_TK\_REQUEST\_PASSKEY, 请参考[“GAP event”章节](#)。

用户调用以下 API 来设置 TK:

```
void blc_smp_setTK_by_PasskeyEntry (u16 connHandle, u32 pinCodeInput);
```

- LE Legacy OOB 的初始化配置:

```
blc_smp_setSecurityLevel(Authenticated_Pairing_with_Encryption);
blc_smp_enableOobAuthentication(1);
blc_smp_smpParamInit();
```

或

```
blc_smp_setSecurityLevel_periph(Authenticated_Pairing_with_Encryption);
blc_smp_setSecurityParameters_periph(Bondable_Mode, 1, LE_Legacy_Pairing, 1, 0,
    ↳ IO_CAPABILITY_KEYBOARD_DISPLAY);
blc_smp_smpParamInit();
```

这里涉及到请求 OOB 数据的 GAP event: GAP\_EVT\_SMP\_TK\_REQUEST\_OOB, 请参考[“GAP event”章节](#)。

34444 Mode 1 Level 4

设备最高支持 Authenticated LE Secure Connections, 如 Secure Connections 的 Numeric Comparison、Passkey Entry、Out of Band。

**注意:**

- Secure Connection 的安全配对方式, 需要 MTU>=65。
- Secure Connections Passkey Entry 方式的初始化配置:



与 Legacy Pairing Passkey Entry 基本一致，唯一不同的是需要在初始化最开始的地方设置配对方式为“安全连接配对”：

```
blc_smp_setSecurityLevel(Authenticated_LE_Secure_Connection_Pairing_with_Encryption);
blc_smp_setPairingMethods(LE_Secure_Connection);
...//参考 Mode 1 Level 3 配置方式
```

- Secure Connections Numeric Comparison 的初始化配置：

```
blc_smp_setSecurityLevel(Authenticated_LE_Secure_Connection_Pairing_with_Encryption);
blc_smp_setPairingMethods(LE_Secure_Connection);
blc_smp_enableAuthMITM(1);
blc_smp_setIoCapability(IO_CAPABILITY_DISPLAY_YES_NO);
blc_smp_smpParamInit();
```

或

```
blc_smp_setSecurityLevel_central(Authenticated_LE_Secure_Connection_Pairing_with_Encryption);
blc_smp_setSecurityParameters_central(Bondable_Mode, 1, LE_Secure_Connection, 0, 0,
    IO_CAPABILITY_DISPLAY_YES_NO);
blc_smp_smpParamInit();
```

这里涉及到请求 Yes/No 的 GAP event: GAP\_EVT\_SMP\_TK\_NUMERIC\_COMPARE，请参考[“GAP event”章节](#)。

- Secure Connections OOB 方式，SDK 暂不支持。

### 3.4.4.5 SMP Storage

无论设备作为 Central 还是 Peripheral，在与其他设备进行 SMP 绑定后，需要将一些 SMP 相关的信息保存到 Flash 中，以便在设备重新上电后，可以实现自动回连。该过程称为 SMP Storage。

#### 3.4.5.1 SMP Storage 区域

在 Flash 中用于存储 SMP 绑定信息的区域称为 SMP Storage 区域。

对于 tl\_ble\_sdk, SMP Storage 区域起始位置由宏 FLASH\_ADR\_SMP\_PAIRING 指定 (1M Flash 默认为 0xFA000)。SMP Storage 区域分为 2 个区，分别称为 A 区、B 区，占用的空间相等，由宏 FLASH\_SMP\_PAIRING\_MAX\_SIZE 指定(默认为 0x2000, 即 8K, 所以总 SMP Storage 区域大小为 16K)。每个区的 (FLASH\_SMP\_PAIRING\_MAX\_SIZE-0x10) 偏移量 (默认为 0x1FF0) 位置为“区有效 Flag”，0x3C 代表有效，0xFF 代表未生效。用户可以使用下面的 API 重新配置 SMP Storage 区域：

```
void blc_smp_configPairingSecurityInfoStorageAddressAndSize (int address, int size_byte); //  
    ↵ address and size must be 4K aligned
```

- address : SMP Storage 区域起始地址 (也是 A 区起始地址)；
- size\_byte : 每个 SMP 区的大小，A 区和 B 区大小相等。



下面的 API 用于获取当前 SMP Storage 有效区的起始地址：

```
u32 blc_smp_getBondingInfoCurStartAddr(void);
```

- 返回值：SMP Storage 中当前有效区的起始地址，如 0xFC000。

配对后，默认先从 SMP Storage A 区开始存储 SMP 绑定信息，当 A 区的绑定信息量达到警戒线 (8KB \* 3/4 = 96 Bytes \* 64，也就是最多存储 64 个 Bonding Info) 后，会将其中有效的绑定信息迁移到 B 区，置“区有效 Flag”为 0x3C，并将 A 区清空。同理，当 B 区绑定信息达到警戒线，则切换到 A 区，并清空 B 区。可以通过以下 API 来确认当前的 SMP Storage 有效区的信息量是否达到了警戒线：

```
bool blc_smp_isBondingInfoStorageLowAlarmed(void);
```

- 返回值：0 表示未到警戒线，1 表示已经到了警戒线。

如果需要清空 SMP Storage 中的信息并重置 SMP 绑定信息，建议在非连接态下调用以下 API：

```
void blc_smp_eraseAllBondingInfo(void);
```

#### 344.5.2 Bonding Info

在 SMP Storage 中存储的每一组 SMP 绑定信息，称为一个 Bonding Info 块，SMP Storage 默认依照配对先后依次将 Bonding Info 填入 SMP Storage 区域，参考其结构 smp\_param\_save\_t，得到：

- 一个 Bonding Info 块的大小为 96 Bytes (0x60)；
- Bonding Info 块的第一个 Byte，即 flag 成员，代表该 Bonding Info 块的状态，如果 flag & 0x0F == 0x0A，则说明该 SMP 绑定信息有效；如果 Central 的 Bonding Info 块的 flag 成员的值为 0x00，代表设备已经解绑；如果 flag 的 bit7 为 0，代表支持 RPA，详情参考 RPA 功能章节（SDK 中暂未完整释放该功能）；
- Bonding Info 块的第二个 Byte，即 role\_dev\_idx，代表自身所扮演的角色，如果 bit7 为 1，代表自身为 Central，如果 bit7 为 0，则代表在该连接中扮演 Peripheral 角色；
- SMP 获取的 peer Id Address 和 local/peer IRK 均存在 Bonding Info 块中。

下图为 SMP Storage 内容的一段参考，该段内容表示该 Bonding Info 块有效，设备为 Central 角色：



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000FA430:	65	98	B3	EE	D3	BD	CB	A4	DE	AA	F3	82	4B	B4	0A	BF
000FA440:	10	00	11	11	22	77	66	55	4C	11	2C	42	34	5B	21	55
000FA450:	BF	0A	B4	53	2A	5F	05	32	82	67	79	28	79	2D	B1	82
000FA460:	D2	10	CA	86	B4	FE	9F	98	29	A5	9D	97	DE	70	62	AA
000FA470:	9A	67	FF													
000FA480:	9A	80	00	04	00	00	77	66	55	00	04	00	00	77	66	55
000FA490:	EE	E0	65	1D	C2	2C	A6	45	71	28	6A	5F	30	AE	73	89
000FA4A0:	10	00	11	11	22	77	66	55	9A	E2	3D	D3	59	BA	8E	D7
000FA4B0:	89	73	AE	EC	6F	4C	B9	FC	D6	19	1B	5C	34	DC	1D	03
000FA4C0:	D2	10	CA	86	B4	FE	9F	98	29	A5	9D	97	DE	70	62	AA
000FA4D0:	11	1F	FF													
000FA4E0:	9A	80	00	05	00	00	77	66	55	00	05	00	00	77	66	55
000FA4F0:	DE	E1	8D	4D	DC	07	25	7F	11	C9	0E	4D	63	AF	CD	5F
000FA500:	10	00	11	11	22	77	66	55	72	B2	23	F8	DA	80	EE	36
000FA510:	5F	CD	AF	B9	DA	A6	E5	EB	B8	A5	00	FC	F8	AD	ED	3D
000FA520:	D2	10	CA	86	R4	FF	9F	98	29	A5	9D	97	DE	70	62	AA

Figure 3.67: SMP Storage Bonding Info Central

可以使用以下 API 通过 peer device 的 MAC 地址获取其 Bonding Info：

```
u32 blc_smp_loadBondingInfoByAddr(u8 isCentral, u8 PeripheralDevIdx, u8 addr_type, u8* addr,
    ↵ smp_param_save_t* smp_param_load);
```

- isCentral：自身的角色，0 表示 Peripheral，非 0 表示 Central；
- PeripheralDevIdx：在不涉及多地址功能时，该参数为 0；
- addr\_type：peer device 的地址类型，参考 BLE\_ADDR\_PUBLIC 和 BLE\_ADDR\_RANDOM；
- addr：peer device 的 MAC 地址；
- smp\_param\_load：出参，指向 peer device 对应的 Bonding Info 块。
- 返回值：peer device 对应的 Bonding Info 块在 Flash 中的首地址。

为便于应用层使用，针对 Central 角色，提供一个根据 peer Peripheral 的 MAC 获取其配对状态的 API：

```
u32 blc_smp_searchBondingPeripheralDevice_by_PeerMacAddress( u8 peer_addr_type, u8* peer_addr);
```

- peer\_addr\_type：peer Peripheral 的地址类型，参考 BLE\_ADDR\_PUBLIC 和 BLE\_ADDR\_RANDOM；
- peer\_addr：peer Peripheral 的 MAC 地址。
- 返回值：找到的 Bonding Device 的 Bonding Info 块在 Flash 中的首地址；0 代表未找到有效绑定信息。

使用以下 API 通过 peer device 的 MAC 地址，将其对应的 Bonding Info 删除（实际上，并未删除，只是通过置 flag 使其失效）：

```
int blc_smp_deleteBondingPeripheralInfo_by_PeerMacAddress(u8 peer_addr_type, u8* peer_addr);
```

- peer\_addr\_type：peer Peripheral 的地址类型，参考 BLE\_ADDR\_PUBLIC 和 BLE\_ADDR\_RANDOM；
- peer\_addr：peer Peripheral 的 MAC 地址。
- 返回值：找到并删除的 Bonding Device 的 Bonding Info 块在 Flash 中的首地址；0 代表未找到有效绑定信息。



### 344.5.3 最大绑定数量

对于 tl\_ble\_sdk，默认最多可以保存 8 个有效 peer Peripheral 的 SMP 信息，和 4 个有效 peer Central 的 SMP 信息（“有效”代表设备可以回连成功，也就是 Bonding Info 块的 flag 成员表示当前状态为有效），这在 SDK 中分别称为 Central 和 Peripheral 的最大绑定数量（Bonding Device Max Number）。用户也可以通过下面的 API 重新配置 SMP Storage 的最大绑定数量：

```
ble_sts_t blc_smp_setBondingDeviceMaxNumber ( int Central_max_bonNum, int
↪ Peripheral_max_bondNum);
```

- Central\_max\_bonNum: 自身作为 Central 角色的最大 peer Peripheral 绑定数量，最大为 8，传入参数超过 8 时，返回错误 0xA0 (SMP\_ERR\_INVALID\_PARAMETER)。
- Peripheral\_max\_bondNum: 自身作为 Peripheral 角色的最大 peer Central 绑定数量，最大为 4，传入参数超过 4 时，返回错误 0xA0 (SMP\_ERR\_INVALID\_PARAMETER)。

达到最大绑定数量时，下一个绑定的设备将会顶替掉当前同角色有效设备中最早绑定的设备。具体而言，会将新的设备的 Bonding Info 继续向 Flash 中写入，置 flag 为有效，同时置同角色有效 Bonding Info 中的第一个设备的 flag 为无效。

举例而言，如果设置了 blc\_smp\_setBondingDeviceMaxNumber(8, 4)，当绑定 8 个 peer Peripheral 后，一旦绑定第 9 个 peer Peripheral，最老的那个（第 1 个）peer Peripheral 的 Bonding Info 失效，并向 Flash 中继续存储第 9 个 peer Peripheral 设备的 Bonding Info。

用户可以通过以下 API 获取当前 Peripheral 或 Central 的绑定数量：

```
u8 blc_smp_param_getCurrentBondingDeviceNumber(u8 isCentral, u8 perDevIdx);
```

- isCentral：自身的角色，0 表示 Peripheral，非 0 表示 Central；
- perDevIdx：在不涉及多地址功能时，该参数为 0；
- 返回值：有效绑定设备的数量，isCentral 为 0 时，表示绑定的有效 peer Central 的数量，isCentral 非 0 时，表示绑定的有效 peer Peripheral 的数量。

### 344.5.4 SMP Bonding Info Index

SMP 中为每个 Bonding Device 的绑定信息都分配了一个序号，称为 Bonding Info Index，Bonding Info Index 的值默认根据绑定的先后顺序在 Bonding Device Max Number 中进行分配。如当 Central 的 Bonding Device Max Number 为 2 时，先后配对的两个 peer Peripheral 的 Bonding Info Index 分别为 0 和 1。

这样，除了上述通过 peer device MAC 地址的方式获取 Bonding Info，也可以在已知设备 Bonding Info Index 的情况下，通过 Bonding Info Index 来获取 Bonding Info：

```
u32 blc_smp_loadBondingInfoFromFlashByIndex(u8 isCentral, u8 PeripheralDevIdx, u8 index,
↪ smp_param_save_t* smp_param_load);
```

- isCentral：自身的角色，0 表示 Peripheral，非 0 表示 Central；
- PeripheralDevIdx：在不涉及多地址功能时，该参数为 0；
- index：表示要读取的 Central 或 Peripheral 信息的 Bonding Info Index。
- smp\_param\_load：出参，指向 peer device 对应的 Bonding Info 块。
- 返回值：peer device 对应的 Bonding Info 块在 Flash 中的首地址。



以下 API 用于设置 Bonding Info Index 的分配原则，SDK 中暂时还没有释放该功能，仅用于部分用户的特殊需求：

```
void blc_smp_setBondingInfoIndexUpdateMethod(index_updateMethod_t method);
```

- method：参考 index\_updateMethod\_t，可以设置为根据建立连接的顺序或按照设备绑定的顺序分配 Bonding Info Index 值。

### 3.4.5 Device Manage & Simple SDP

如前面对于 GATT 的描述，在 BLE 中，Peripheral 充当 GATT Server 角色时，会维护一个 GATT Service 的表，表中的每条 Attribute 都对应有个一 Attribute handle 值。而对于 Central，想要获取 Peripheral 的这些信息，需要通过 SDP 过程获取，并进行维护，以供需要时使用。

为了便于用户使用，tl\_ble\_sdk 为用户提供了个连接设备管理方案的实现 Device Manage 以及一个 Central 做 SDP 获取 peer Peripheral 的 GATT Service 表的简单实现。不仅可以为 Central 管理 peer Peripheral 的 GATT Service 表，还可以用于随时通过对端设备的部分信息，调取该设备的其他相关信息。该方案全部以源码的形式提供，用户可以参考 SDK 中 vendor/common/device\_manage.\* 文件及 vendor/common/simple\_sdp.\* 文件。

tl\_ble\_sdk 使用如下数据结构来管理 Attribute handle 和 Connection handle。

```
typedef struct
{
    u16      conn_handle;
    u8       conn_role;           // 0: Central; 1: Peripheral
    u8       conn_state;         // 1: connect; 0: disconnect

    u8      char_handle_valid;   // 1: peer device's attHandle is available; 0: peer
    ↵ device's attHandle not available
    u8      rsvd[3];             // for 4 Byte align

    u8      peer_addrType;
    u8      peer_addr[6];
    u8      peer_RPA;            //RPA: resolvable private address

    u16      char_handle[CHAR_HANDLE_MAX];
}dev_char_info_t;
```

在 SDK 中，使用数组 conn\_dev\_list[] 来记录和维护对端设备的参数，如下图所示。



```
device_manage.h device_manage.c
>
>
>
> /**
> * Used for store information of connected devices.
> *
> * 0 ~ (MASTER_MAX_NUM - 1) is for master,  MASTER_MAX_NUM ~ (MASTER_MAX_NUM + SLAVE_MAX_NUM - 1) s for slave
> *
> * e.g.      MASTER_MAX_NUM    SLAVE_MAX_NUM        master           slave
> *          0                  1                none            conn_dev_list[0]
> *          0                  2                none            conn_dev_list[0..1]
> *          0                  3                none            conn_dev_list[0..2]
> *          0                  4                none            conn_dev_list[0..3]
> *
> *          1                  0                conn_dev_list[0]       none
> *          1                  1                conn_dev_list[0]       conn_dev_list[1]
> *          1                  2                conn_dev_list[0]       conn_dev_list[1..2]
> *          1                  3                conn_dev_list[0]       conn_dev_list[1..3]
> *          1                  4                conn_dev_list[0]       conn_dev_list[1..4]
> *
> *          2                  0                conn_dev_list[0..1]     none
> *          2                  1                conn_dev_list[0..1]     conn_dev_list[2]
> *          2                  2                conn_dev_list[0..1]     conn_dev_list[2..3]
> *          2                  3                conn_dev_list[0..1]     conn_dev_list[2..4]
> *          2                  4                conn_dev_list[0..1]     conn_dev_list[2..5]
> *
> *          3                  0                conn_dev_list[0..2]     none
> *          3                  1                conn_dev_list[0..2]     conn_dev_list[3]
> *          3                  2                conn_dev_list[0..2]     conn_dev_list[3..4]
> *          3                  3                conn_dev_list[0..2]     conn_dev_list[3..5]
> *          3                  4                conn_dev_list[0..2]     conn_dev_list[3..6]
> *
> *          4                  0                conn_dev_list[0..3]     none
> *          4                  1                conn_dev_list[0..3]     conn_dev_list[4]
> *          4                  2                conn_dev_list[0..3]     conn_dev_list[4..5]
> *          4                  3                conn_dev_list[0..3]     conn_dev_list[4..6]
> *          4                  4                conn_dev_list[0..3]     conn_dev_list[4..7]
> */
> _attribute_ble_data_retention_ dev_char_info_t conn_dev_list[DEVICE_CHAR_INFO_MAX_NUM];
>
```

Figure 3.68: conn\_dev\_list 数组定义

当与其他设备建立连接时，在 connection complete event 中通过调用 dev\_char\_info\_insert\_by\_conn\_event() 将对端设备的身份信息存入 conn\_dev\_list[]。

```
int app_le_connection_complete_event_handle(u8 *p)
{
    hci_le_connectionCompleteEvt_t *pConnEvt = (hci_le_connectionCompleteEvt_t *)p;

    if(pConnEvt->status == BLE_SUCCESS) {
        dev_char_info_insert_by_conn_event(pConnEvt);

        if( pConnEvt->role == LL_ROLE_MASTER ) // master role, process SMP and SDP if necessary
        {
            #if (BLE_MASTER_SMP_ENABLE)
            #else
                //manual pairing, device match, add this device to slave mac table
                if(blm_manPair.manual_pair && blm_manPair.mac_type == pConnEvt->peerAddrType && !memcmp(blm_manPair.mac, pConnEvt->peerAddr, 6)){
                    blm_manPair.manual_pair = 0;
                    user_tbl_slave_mac_add(pConnEvt->peerAddrType, pConnEvt->peerAddr);
                }
            #endif
        }
    }
}
```

Figure 3.69: connection completed event handle

如果自己是 Central 角色,且启用了 Simple SDP 功能,会先通过 dev\_char\_info\_search\_peer\_att\_handle\_by\_peer\_mac() 查询对端设备的 GATT Service 表是否已经在 Flash 中,如果在,直接从 Flash 中取出通过 dev\_char\_info\_add\_peer\_att\_handle() 放到 conn\_dev\_list[] 中:



```
app.c [3]
187         user_tbl_slave_mac_add(pConnEvt->peerAddrType, pConnEvt->peerAddr);
188     }
189 #endif
190
191
192 #if (BLE_MASTER_SIMPLE_SD_ENABLE)
193     memset(&cur_sdp_device, 0, sizeof(dev_char_info_t));
194     cur_sdp_device.conn_handle = pConnEvt->connHandle;
195     cur_sdp_device.peer_addrType = pConnEvt->peerAddrType;
196     memcpy(cur_sdp_device.peer_addr, pConnEvt->peerAddr, 6);
197
198     u8 temp_buff[sizeof(dev_att_t)];
199     dev_att_t *pdev_att = (dev_att_t *)temp_buff;
200
201     /* att handle search in flash, if success, load char handle directly from flash, no need SDP again */
202     if( dev_char_info_search_peer_att_handle_by_peer_mac(pConnEvt->peerAddrType, pConnEvt->peerAddr, pdev_att) ){
203         //cur_sdp_device.char_handle[1] = pdev_att->char_handle[1]; //speaker
204         cur_sdp_device.char_handle[2] = pdev_att->char_handle[2]; //OTA
205         cur_sdp_device.char_handle[3] = pdev_att->char_handle[3]; //consume report
206         cur_sdp_device.char_handle[4] = pdev_att->char_handle[4]; //normal key report
207         //cur_sdp_device.char_handle[6] = pdev_att->char_handle[6]; //BLE Module, SPP Server to Client
208         //cur_sdp_device.char_handle[7] = pdev_att->char_handle[7]; //BLE Module, SPP Client to Server
209
210         /* add the peer device att handle value to conn_dev_list */
211         dev_char_info_add_peer_att_handle(&cur_sdp_device);
212     }
213
214 } else
215 {
216     master_sdp_pending = pConnEvt->connHandle; // mark this connection need SDP
217
218 #if (BLE_MASTER_SMP_ENABLE)
219 #else
220     app_register_service(&app_service_discovery); //No SMP, service discovery can initiated now
221 #endif
222
223 }
224 #endif
225 }
226
227
228 return 0;
229 }
```

Figure 3.70: simpleSDP info in Flash

如果不在，将通过 app\_service\_discovery() 来获取。获取到之后，会调用函数 dev\_char\_info\_add\_peer\_att\_handle() 和 dev\_char\_info\_store\_peer\_att\_handle() 将 peer Peripheral GATT Service 表分别存放到 RAM 和 FLASH 中，以便后续使用，如下图所示。

```
app.c [3]
94
95
96 /**
97 * @brief SDP handler.
98 * !!! Note: This is a simple SDP processing implemented by telink.
99 * @param none.
100 * @return none.
101 */
102 void app_service_discovery(void)
103 {
104     att_db_uuid16_t db16[ATT_DB_UUID16_NUM];
105     att_db_uuid128_t db128[ATT_DB_UUID128_NUM];
106     memset(db16, 0, ATT_DB_UUID16_NUM * sizeof(att_db_uuid16_t));
107     memset(db128, 0, ATT_DB_UUID128_NUM * sizeof(att_db_uuid128_t));
108
109     if (master_sdp_pending && host_att_discoveryService(master_sdp_pending, db16, ATT_DB_UUID16_NUM, db128, ATT_DB_UUID128_NUM) == BLE_SUCCESS) // service discovery OK
110     {
111         cur_sdp_device.char_handle[2] = blm_att_findHandleOfUuid128(db128, my_OtaUUID); //OTA
112         cur_sdp_device.char_handle[3] = blm_att_findHandleOfUuid16(db16, CHARACTERISTIC_UUID_HID_REPORT,
113             HID_REPORT_ID_CONSUME_CONTROL_INPUT | (HID_REPORT_TYPE_INPUT<>0)); //consume report(media key report)
114         cur_sdp_device.char_handle[4] = blm_att_findHandleOfUuid16(db16, CHARACTERISTIC_UUID_HID_REPORT,
115             HID_REPORT_ID_KEYBOARD_INPUT | (HID_REPORT_TYPE_INPUT<>0)); //normal key report
116         //cur_sdp_device.char_handle[6] = blm_att_findHandleOfUuid128(db128, TelinkSppDataServer2ClientUUID); //BLE Module, SPP Server to Client
117         //cur_sdp_device.char_handle[7] = blm_att_findHandleOfUuid128(db128, TelinkSppDataClient2ServerUUID); //BLE Module, SPP Client to Server
118
119         /* add the peer device att handle value to conn_dev_list after service discovery is correctly finished */
120         dev_char_info_add_peer_att_handle(&cur_sdp_device);
121
122         /* peer device att handle value store in flash */
123         dev_char_info_store_peer_att_handle(&cur_sdp_device);
124
125         //my_dump_str_data(APP_DUMP_SDP_EN, "OTA handle", (u8*)&cur_sdp_device.char_handle[2], 2);
126         //my_dump_str_data(APP_DUMP_SDP_EN, "CMKEY handle", (u8*)&cur_sdp_device.char_handle[3], 2);
127         //my_dump_str_data(APP_DUMP_SDP_EN, "KBKEY handle", (u8*)&cur_sdp_device.char_handle[4], 2);
128     }
129
130     master_sdp_pending = 0; //service discovery finish
131 }
```

Figure 3.71: service discovery

注意：



SDP 是一个很复杂的部分，对于 tl\_ble\_sdk，由于芯片资源有限，SDP 不能做的像手机那样复杂。这里给出的是一个简单参考。

用户可以通过根据 connHandle dev\_char\_info\_search\_by\_connhandle() 来取用 GATT Service 表中的 Attribute handle，其返回值是 conn\_dev\_list[index] 结构体的指针，指向该 connHandle 所对应的 conn\_dev\_list[] 数组中的那个元素。

## 3.5 LE Advertising Extensions

随着 BLE 应用越来越广泛，功能也随之增加了很多，下面我们来介绍在 core5.0 引入的 LE Advertising Extensions，包含：Extended advertising、Periodic advertising、Extended Scan、Periodic Scan。这些功能的引入也为后面 LE Audio 做了准备，当然这些功能也可以根据实际情况用作其他用途。

### 3.5.1 扩展广播 (Extended Advertising)

BLE core 5.0 之前版本，有一个较大的限制，广播数据的 payload 太小了，只有 31B。但是又不能单纯的增加 payload 来扩容，因为 37、38、39 是广播物理信道，大家都在使用，冲突的概率很大，payload 越长冲突几率越大。所以在 core 5.0 进行了 Advertising Extensions，增加了广播集、周期性广播的概念，既可以解决载荷太小的问题，又能解决冲突概率大的问题。因为使用的其他 37 个 data channel，可以使用跳频机制来减少冲突几率。

**注意：**

core spec 也将最大载荷限制在了 1650B，即 AUX\_\*\* 加上所有对应的 chain packet 的有效数据不超过 1650B。

在《Core\_V4.2》及以前的版本，0 ~ 36 这 37 个信道主要用 LE-ACL 连接的数据信道。在《Core\_V5.0》广播信道定义中，将 37, 38, 39 三个信道定义为主广播信道 (Primary Advertising channel)，其余 37 个信道叫做辅助广播信道或者第二广播信道 (Secondary Advertising channel)，辅助广播信道也可以用于发送或接收广播数据。

对于 extended advertising，在 primary advertising channel 上发送的只有 ADV\_EXT\_IND，在 secondary advertising channel 发送的广播包都是以 AUX\_\*\* 命名。

**注意：**

在 primary advertising channel 上可以使用 1M 和 coded phy，不能使用 2M phy (至少在最新 core spec5.4 版本中仍然是这样规定的)。

Advertising Extensions 也为 LE Audio 的实现打下了基础。因为在建立 ACL 连接初期，就需要知道 peer device 的较多信息，比如 Audio 的一些参数、时序信息、加密信息等等。这些需要较多的广播载荷。

扩展广播包含 ADV\_EXT\_IND、AUX\_ADV\_IND 及对应的 AUX\_CHAIN\_IND。如果需要广播更多数据 (最多 1650 字节)，控制器可以将数据分段并使用 AuxPtr 将各个分段“串联”起来。每个分段都可以在不同的信道上传输。ADV\_EXT\_IND(AuxPtr)→AUX\_ADV\_IND(AuxPtr)→AUX\_CHAIN\_IND(AuxPtr)→AUX\_CHAIN\_IND.....

**扩展广播的核心思想：广播数据可以用数据信道来传输。**

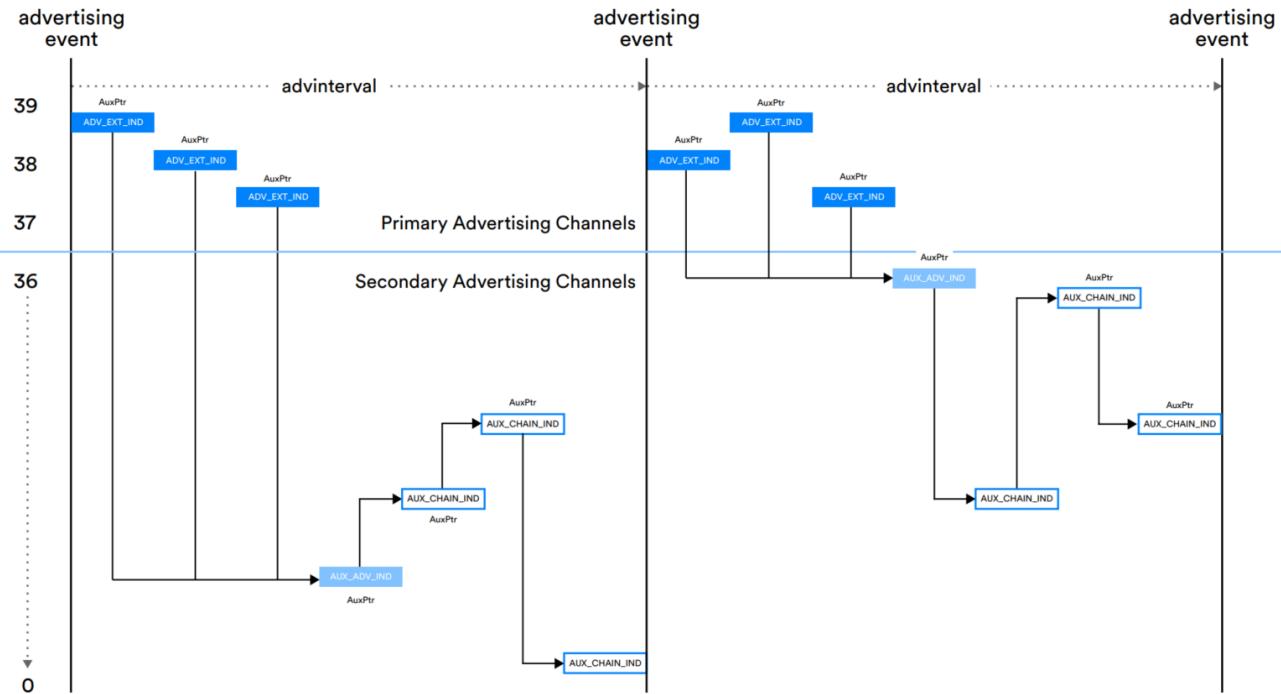


Figure 3.72: 具有包链的扩展广播

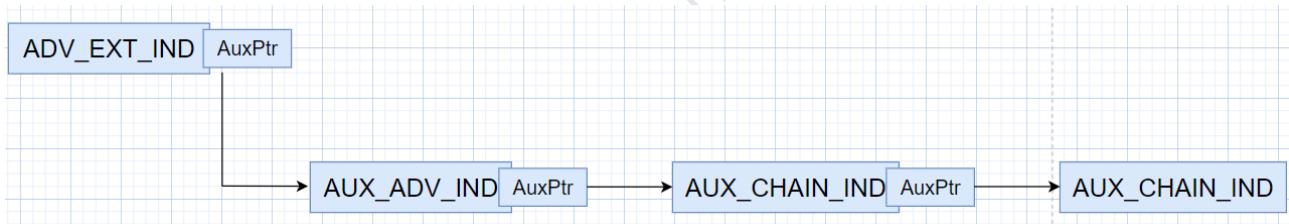


Figure 3.73: 扩展广播



ADV_EXT_IND Packet (AdvDataInfo   AuxPtr[22 (data), @552.062 330 675... Master: "EBQ" DA:D1:5B:AD:A5:55 <-> Slave:... 7 bytes (06 1... OK 9:43:04.769 170 200 LE-LL 0x1
AUX_ADV_IND Packet (DA:D1:5B:AD:A5:55, AdvA   AdvDataInfo   AuxPtr[... Master: "EBQ" DA:D1:5B:AD:A5:55 <-> Slave:... 255 bytes (0C... OK 9:43:04.769 635 000 LE-LL 0x1
AUX_CHAIN_IND Packet (AdvDataInfo   AuxPtr[36 (data), @552.066 770 5... Master: "EBQ" DA:D1:5B:AD:A5:55 <-> Slave:... 198 bytes (06... OK 9:43:04.772 080 100 LE-LL 0x1
AUX_CHAIN_IND Packet (AdvDataInfo   AuxPtr[12 (data), @552.068 765 6... Master: "EBQ" DA:D1:5B:AD:A5:55 <-> Slave:... 198 bytes (06... OK 9:43:04.774 075 200 LE-LL 0x1
AUX_CHAIN_IND Packet (AdvDataInfo   AuxPtr[18 (data), @552.070 760 5... Master: "EBQ" DA:D1:5B:AD:A5:55 <-> Slave:... 198 bytes (06... OK 9:43:04.776 070 100 LE-LL 0x1
AUX_CHAIN_IND Packet (AdvDataInfo   AuxPtr[1 (data), @552.072 755 42... Master: "EBQ" DA:D1:5B:AD:A5:55 <-> Slave:... 198 bytes (06... OK 9:43:04.778 065 000 LE-LL 0x1
AUX_CHAIN_IND Packet (AdvDataInfo   AuxPtr[2 (data), @552.074 750 42... Master: "EBQ" DA:D1:5B:AD:A5:55 <-> Slave:... 198 bytes (06... OK 9:43:04.780 060 000 LE-LL 0x1
AUX_CHAIN_IND Packet (AdvDataInfo   AuxPtr[29 (data), @552.076 745 6... Master: "EBQ" DA:D1:5B:AD:A5:55 <-> Slave:... 198 bytes (06... OK 9:43:04.782 055 200 LE-LL 0x1
AUX_CHAIN_IND Packet (AdvDataInfo   AuxPtr[14 (data), @552.078 740 6... Master: "EBQ" DA:D1:5B:AD:A5:55 <-> Slave:... 198 bytes (06... OK 9:43:04.784 050 200 LE-LL 0x1
AUX_CHAIN_IND Packet (AdvDataInfo   Adv Data, #2'848->) Master: "EBQ" DA:D1:5B:AD:A5:55 <-> Slave:... 70 bytes (03 ... OK 9:43:04.786 045 100 LE-LL 0x1
ADV_EXT_IND Packet (AdvDataInfo   AuxPtr[22 (data), @552.284 630 550... Master: "EBQ" DA:D1:5B:AD:A5:55 <-> Slave:... 7 bytes (06 1... OK 9:43:04.990 570 100 LE-LL 0x1
ADV_EXT_IND Packet (AdvDataInfo   AuxPtr[22 (data), @552.284 630 550... Master: "EBQ" DA:D1:5B:AD:A5:55 <-> Slave:... 7 bytes (06 1... OK 9:43:04.991 020 100 LE-LL 0x1
ADV_EXT_IND Packet (AdvDataInfo   AuxPtr[22 (data), @552.284 630 550... Master: "EBQ" DA:D1:5B:AD:A5:55 <-> Slave:... 7 bytes (06 1... OK 9:43:04.991 470 100 LE-LL 0x1
AUX_ADV_IND Packet (DA:D1:5B:AD:A5:55, AdvA   AdvDataInfo   AuxPtr[... Master: "EBQ" DA:D1:5B:AD:A5:55 <-> Slave:... 255 bytes (0C... OK 9:43:04.991 935 200 LE-LL 0x1
AUX_CHAIN_IND Packet (AdvDataInfo   AuxPtr[36 (data), @552.289 070 5... Master: "EBQ" DA:D1:5B:AD:A5:55 <-> Slave:... 198 bytes (06... OK 9:43:04.994 380 100 LE-LL 0x1

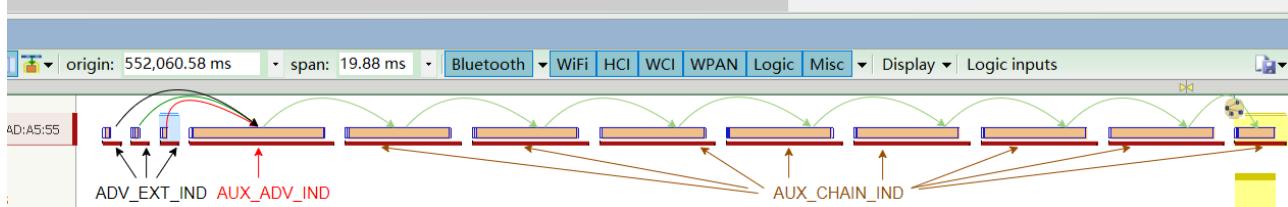


Figure 3.74: 具有包链的扩展广播

所有的扩展广播名称及详细描述：

PDU Name	Description	Channels	PHY(s)	Transmitted By
ADV_EXT_IND	Extended advertising	primary	LE 1M, LE Coded	Peripheral
AUX_ADV_IND	Subordinate extended advertising	secondary	LE 1M, LE 2M, LE Coded	Peripheral
AUX_CHAIN_IND	Additional advertising data	secondary	LE 1M, LE 2M, LE Coded	Peripheral
AUX_SYNC_IND	Periodic advertising synchronization	periodic	LE 1M, LE 2M, LE Coded	Peripheral
AUX_SCAN_REQ	Auxiliary scan request	secondary	LE 1M, LE 2M, LE Coded	Central
AUX_SCAN_RSP	Auxiliary scan response	secondary	LE 1M, LE 2M, LE Coded	Peripheral
AUX_CONNECT_REQ	Auxiliary connect request	secondary	LE 1M, LE 2M, LE Coded	Central
AUX_CONNECT_RSP	Auxiliary connect response	secondary	LE 1M, LE 2M, LE Coded	Peripheral

Figure 3.75: 扩展广播 PDU

legacy advertising 与 extended advertising 比较：



	传统广播	扩展广播	
最大主机广播数据大小	31 字节	1,650 字节	Extended Advertising 支持分段,这使得支持的最大主机广播数据大小增加 50 倍。
最大主机广播数据大小/包	31 字节	254 字节	扩展广播 PDU 使用通用扩展广播负载格式,支持 8 倍大的广告数据字段。
发射通道	37,38,39	0 - 39	Extended Advertising 使用 37 个数据信道作为辅助广播信道。然而, ADV_EXT_IND PDU 类型只能在主要广播信道 (37,38,39)上传输。
PHY支持	LE 1M  LE 2M (不包括 ADV_EXT_IND PDU  LE 编码)	LE 1M	除 ADV_EXT_IND 之外的所有扩展广播 PDU 都可以使用三个 LE PHY 中的任何一个进行传输,ADV_EXT_IND PDU 除外, 它可以使用 LE 1M 或 LE 编码 PHY 进行传输。
最大活动广播数量	1	16	Extended Advertising 包括 Advertising Sets, 它使广播设备能够同时支持多达 16 种不同的广播配置,并根据时间间隔为每个广播集插入广播
通信类型	异步	异步  同步	Extended Advertising 包括 Periodic Advertising, 可在发送器和接收器之间实现广播数据的时间同步通信。

Figure 3.76: 传统广播和扩展广播的总结比较

### 3.5.1.1 广播集 (Advertising Sets)

Extended advertising 引入了广播集概念, 即一个设备可以“同时”进行多个广播, 每个广播可以使用不同的 interval、不同的 PDU data、不同的 PDU type、不同的 phy 等等, 比如可以“同时”发送可连接广播和不可连接广播。不同广播集使用 extended header 中的 ADI 字段进行区分。

注意:

目前 tl\_ble\_sdk 支持创建最大广播集数量是 4 个。

### 3.5.1.2 Extended Advertising 相关的 API 介绍

```
ble_sts_t blc_ll_initExtendedAdvModule_initExtendedAdvSetParamBuffer(u8 *pBuff_advSets, int
← num_advSets);
```

如果使用扩展广播, 需使用该 API 进行初始化。目的是初始化 extended advertising module 及对应广播参数所需要的 buffer 空间。只有进行了相应模块的初始化, 相应的 feature 才会生效, 对应的功能函数才会链接到执行 (bin) 文件中。

pBuff\_advSets: 底层控制 extended advertising 使用的 buffer 空间首地址。stack 会使用该 buffer 来存储 extended advertising 的运行时所需的各种控制变量。每个广播集需要分配一个这样的 buffer 空间。之所以交给上层用户来管控, 是因为用户可以根据实际使用的广播集数量来进行分配, 以节约 RAM size。

num\_advSets: 用户实际使用的广播集数量。note: 底层最大支持 4 个广播集。



```
void blc_ll_initExtendedAdvDataBuffer(u8 *pExtAdvData, int max_len_advData);
```

设置存储 extended advertising data 的 buffer，该 buffer 用来存储 blc\_ll\_setExtAdvData 设置的数据。

pExtAdvData：buffer 首地址。

max\_len\_advData：buffer size。blc\_ll\_setExtAdvData 设置的数据长度不能大于该值。

```
void blc_ll_initExtendedScanRspDataBuffer(u8 *pScanRspData, int max_len_scanRspData);
```

设置存储 scan response data 的 buffer，该 buffer 用来存储 blc\_ll\_setExtScanRspData 设置的数据。

pScanRspData：scan response buffer 首地址。

max\_len\_scanRspData：buffer size。blc\_ll\_setExtScanRspData 设置的数据长度不能大于该值。

```
ble_sts_t blc_ll_setExtAdvParam( u8 adv_handle, advEvtProp_type_t adv_evt_prop,
    ↵ u32 pri_advInter_min,     u32 pri_advInter_max,
    ↵                         adv_chn_map_t pri_advChnMap,   own_addr_type_t ownAddrType,
    ↵ u8 peerAddrType,          u8 *peerAddr,
    ↵                         adv_fp_type_t advFilterPolicy, tx_power_t adv_tx_pow,
    ↵ le_phy_type_t pri_adv_phy, u8 sec_adv_max_skip,
    ↵                         le_phy_type_t sec_adv_phy,   u8 adv_sid,
    ↵ u8 scan_req_noti_en);
```

BLE Spec 标准接口，用于设置 extended advertising 参数，详细请参考《Core\_5.4》(Vol 4/Part E/7.8.53 “LE Set Extended Advertising Parameters Command”), 并结合 SDK 该 API 的注释来理解。

```
ble_sts_t blc_ll_setExtAdvData (u8 adv_handle, int advData_len, u8 *advData);
```

BLE Spec 标准接口，用于设置 extended advertising 发送的数据，详细参考《Core\_5.4》(Vol 4/Part E/7.8.54 “LE Set Extended Advertising Data command”), 并结合 SDK 该 API 的注释来理解。

```
ble_sts_t blc_ll_setExtScanRspData(u8 adv_handle, int scanRspData_len, u8 *scanRspData);
```

BLE Spec 标准接口，用于设置 extended scan response 的数据，详细参考《Core\_5.4》(Vol 4/Part E/7.8.55 “LE Set Extended Scan Response Data command”), 并结合 SDK 该 API 的注释来理解。

```
ble_sts_t blc_ll_setExtAdvEnable(adv_en_t enable, u8 adv_handle, u16 duration, u8
    ↵ max_extAdvEvt);
```

BLE Spec 标准接口，用于打开/关闭 Extended Advertising，详细可参考《Core\_5.4》(Vol 4/Part E/7.8.56 “LE Set Extended Advertising Enable Command”), 并结合 SDK 该 API 的注释来理解。



```
ble_sts_t blc_ll_setAdvRandomAddr(u8 adv_handle, u8* rand_addr);
```

BLE Spec 标准接口，用来设置设备的 random address。详细可参考《Core\_5.4》(Vol 4/Part E/7.8.52 “LE Set Advertising Set Random Address command”), 并结合 SDK 该 API 的注释来理解。

```
ble_sts_t blc_ll_removeAdvSet(u8 adv_handle);
```

BLE Spec 标准接口，用来移除对应的广播集。详细可参考《Core\_5.4》(Vol 4/Part E/7.8.59 “LE Remove Advertising Set command”), 并结合 SDK 该 API 的注释来理解。

```
ble_sts_t blc_ll_clearAdvSets(void);
```

BLE Spec 标准接口，用来移除所有的广播集。详细可参考《Core\_5.4》(Vol 4/Part E/7.8.60 “LE Clear Advertising Sets command”), 并结合 SDK 该 API 的注释来理解。

### 3.5.2 周期广播 (Periodic Advertising)

周期广播也是 core 5.0 引入的概念，如果需要固定周期来发送数据，就需要使用 periodic advertising。周期广播 interval 和 ACL interval 的概念是相同的，每个 interval 使用不同的频点，使用 CSA#2 跳频算法，周期广播使用 37 个二级广播信道 (secondary advertising channels)。

periodic advertising 是：由 ADV\_EXT\_IND AuxPtr 引导出 AUX\_ADV\_IND，由 AUX\_ADV\_IND SyncInfo 引导出 AUX\_SYNC\_IND。如果数据需要 chain 包来继续发送，则由 AUX\_SYNC\_IND AuxPtr 引导出对应的 AUX\_CHAIN\_IND。即：

ADV\_EXT\_IND(AuxPtr) → AUX\_ADV\_IND(SyncInfo) → AUX\_SYNC\_IND(AuxPtr) → AUX\_CHAIN\_IND(AuxPtr)  
→ AUX\_CHAIN\_IND.....

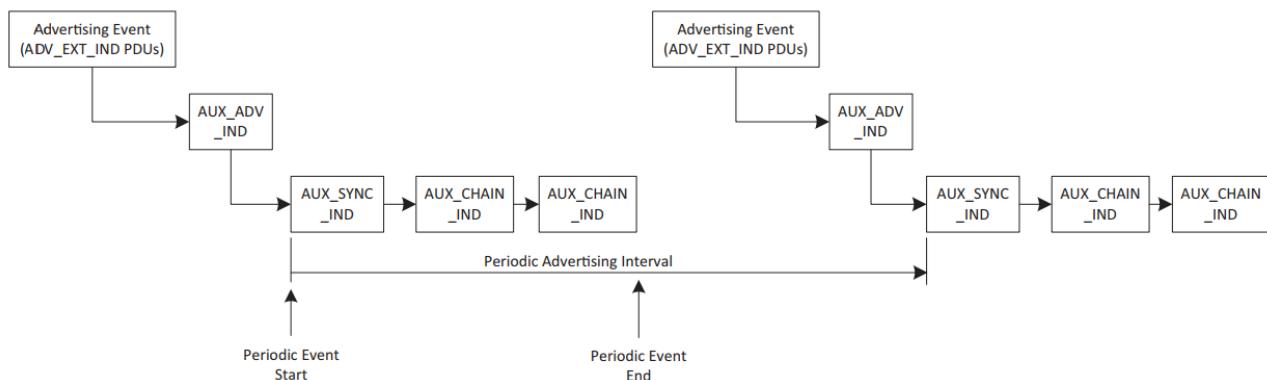


Figure 3.77: 周期广播事件

#### 注意:

- 周期广播引出之后，引出周期广播的 ADV\_EXT\_IND 及 AUX\_ADV\_IND 可以继续发送，也可以停止。如果停止，对于已经同步到周期广播的设备没影响，但是对于还没有同步上的设备或者是刚上电的设备就同步不上对应的周期广播了。是否停止由用户根据实际情况来决定。
- 扩展广播 interval > 周期广播 interval：每个 ADV\_EXT\_IND 之间会有多个 AUX\_SYNC\_IND 出现。扩



展广播 interval < 周期广播 interval: 多个 ADV\_EXT\_IND 最终指向同一个 AUX\_SYNC\_IND。

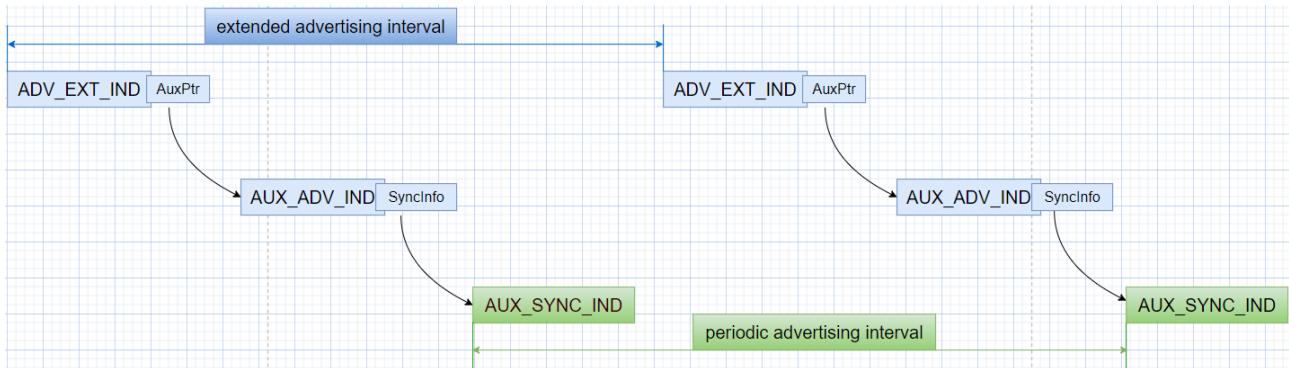


Figure 3.78: 周期广播事件

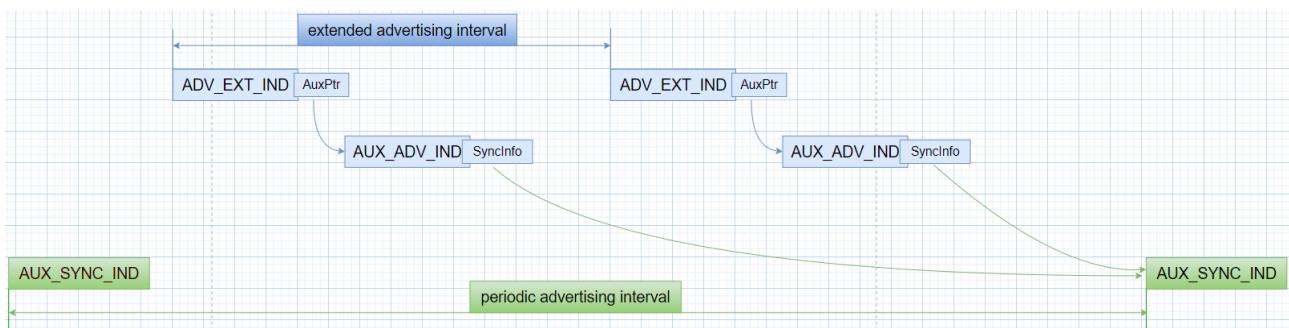


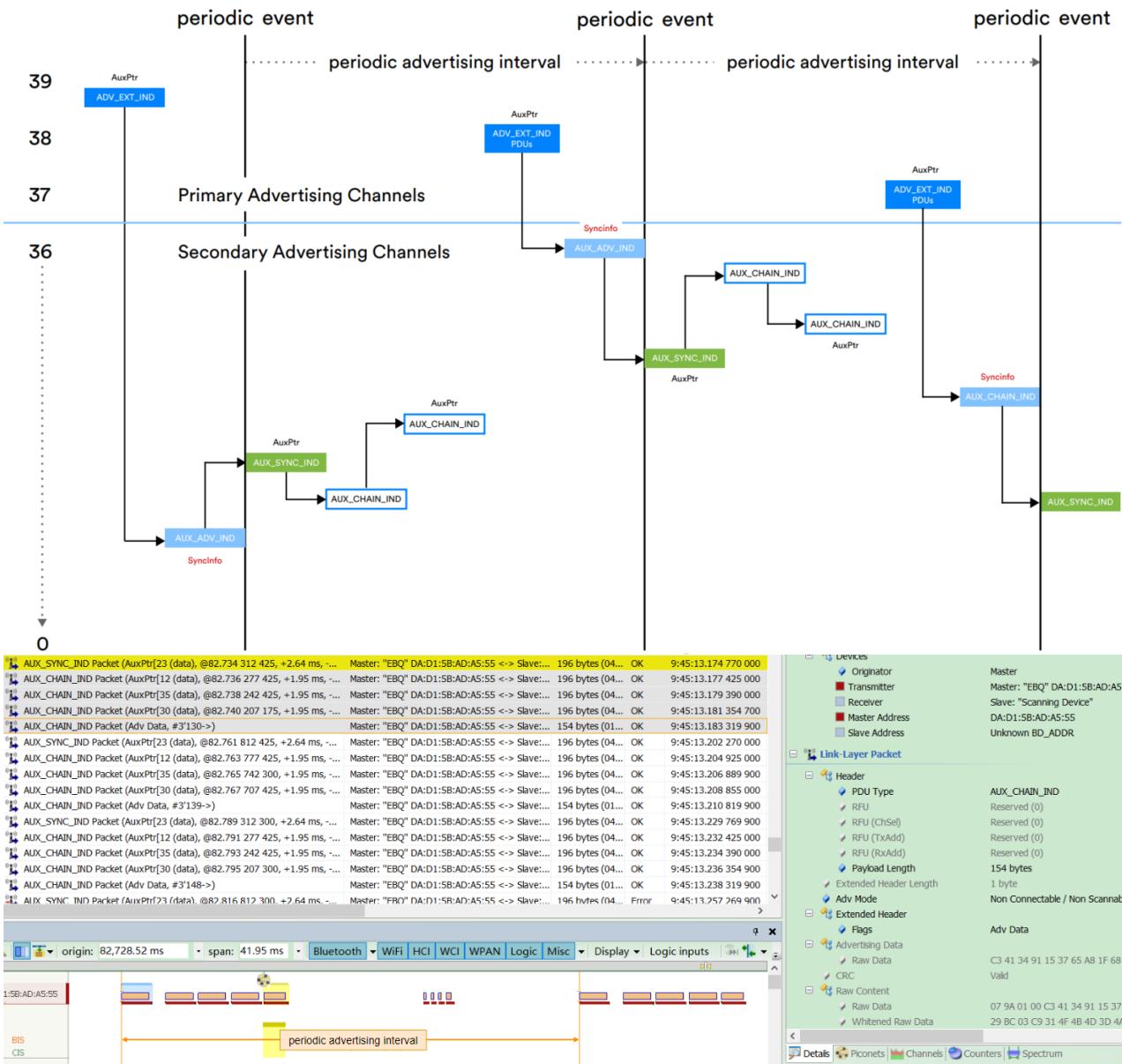
Figure 3.79: 周期广播事件

周期广播可以让扫描设备更节省功耗，因为只需要固定的时间点扫描即可。周期广播是 LE Audio broadcast 解决方案的关键组成部分。

周期广播间隔确定给定广播集的周期广播可以发生的频率。它从 AUX\_SYNC\_IND PDU 的传输开始，然后是一系列零个或多个 AUX\_CHAIN\_IND PDU，如下图所示。

#### 注意:

目前 tl\_ble\_sdk 支持创建最大 2 个周期广播 (4 个广播集中只有 2 个可以引导出周期广播)。



### 3.5.2.1 Periodic Advertising 相关的 API 介绍

PA 广播 (Advertising) 相关的 API：

```
void blc_ll_initPeriodicAdvModule_initPeriodicAdvSetParamBuffer(u8 *pBuff, int num_periodic_adv);
```

初始化 periodic advertising module，只有进行了相应模块的初始化，相应的 feature 才会生效，对应的功能函数才会链接到执行 (bin) 文件中。

初始化 periodic advertising module 所需参数的 buffer 空间。stack 会使用该 buffer 来存储 periodic advertising 运行时所需的各种控制变量。每个广播集需要分配一个这样的 buffer 空间。之所以交由上层用户来管控，是因为用户可以根据实际使用的广播集数量来进行分配，以节约 RAM size。

pBuff: periodic advertising 参数使用的 buffer 空间地址。



num\_periodic\_adv：用户实际使用的周期广播集数量。note：底层最大支持的周期广播集数量是 2 个。

```
ble_sts_t blc_ll_setPeriodicAdvParam(adv_handle_t adv_handle, u16 advInter_min, u16
↪ advInter_max, perd_adv_prop_t property);
```

BLE Spec 标准接口，用于设置 periodic advertising 参数。详细可参考《Core\_5.4》(Vol 4/Part E/7.8.61 "LE Set Periodic Advertising Parameters command")，并结合 SDK 该 API 的注释来理解。

```
void blc_ll_initPeriodicAdvDataBuffer(u8 *perdAdvData, int max_len_perdAdvData);
```

设置存储 periodic advertising data 的 buffer，该 buffer 用来存储 blc\_ll\_setPeriodicAdvData 设置的数据。

```
ble_sts_t blc_ll_setPeriodicAdvData(adv_handle_t adv_handle, u16 advData_len, u8 *advdata);
```

BLE Spec 标准接口，用于设置 periodic advertising 发送的数据，详细参考《Core\_5.4》(Vol 4/Part E/7.8.62 "LE Set Periodic Advertising Data command")，并结合 SDK 该 API 的注释来理解。

```
ble_sts_t blc_ll_setPeriodicAdvEnable(u8 per_adv_enable, adv_handle_t adv_handle);
```

BLE Spec 标准接口，用于打开/关闭 periodic Advertising，详细参考《Core\_5.4》(Vol 4/Part E/7.8.63 "LE Set Periodic Advertising Enable command")，并结合 SDK 该 API 的注释来理解。

### 3.5.3 扩展扫描 (Extended SCAN)

对于传统广播包的获取，传统扫描设备只需要扫描 37、38、39 (Primary Advertising channel) 这个 3 个信道，只需按照扫描窗口和扫描周期在这 3 个信道间来回切换，扫描窗口期间只要收到广播数据就按照协议要求规则处理即可。

但是如果要扫描扩展广播包，需要扫描并获取主广播信道 37/38/39(Primary Advertising channel) 上的 ADV\_EXT\_IND 广播包，然后需要解析出其中是否包含 AuxPtr 信息，如果存在的话，需要获取 AuxPtr 中携带的下一个 Auxiliary 广播包的时序、跳频信息、PHY 等信息，扫描设备需要根据这些信息在合适的窗口监听广播包。如果遇到包链较长的情况，扫描设备继续处理下一个 AuxPtr 直到无 Auxptr 字段为止，收完整包后根据广播包是否是可扫描或可连接以及包类型做相应的处理。蓝牙核心规范包含完整的详细信息，请参考《Core\_V5.4》，Vol 6, Part B **4.4.3 Scanning state**。

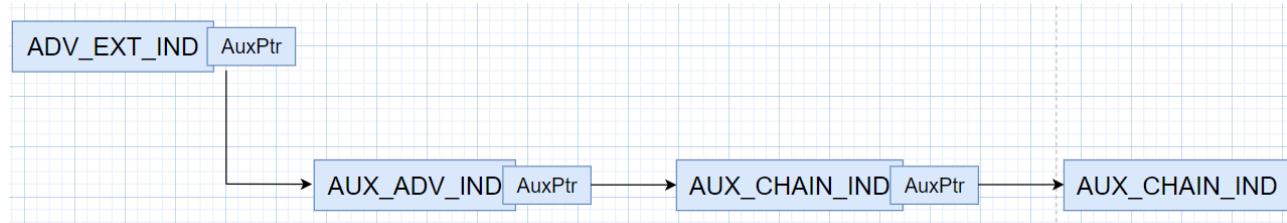


Figure 3.80: 扩展扫描事件

扩展扫描的复杂性在于如何获取辅助信道上的广播包，而广播包可能存在多级引导的情况。



### 3.5.3.1 Extended SCAN 相关的 API 介绍

```
void blc_ll_initExtendedScanning_module(void);
```

初始化 extended scan module，只有进行了相应模块的初始化，相应的 feature 才会生效，对应的功能函数才会链接到执行（bin）文件中。

```
ble_sts_t blc_ll_setExtScanParam ( own_addr_type_t ownAddrType, scan_fp_type_t scan_fp,
↳ scan_phys_t scan_phys, scan_type_t scanType_0, scan_inter_t scanInter_0, scan_wind_t
↳ scanWindow_0, scan_type_t scanType_1, scan_inter_t scanInter_1, scan_wind_t scanWindow_1);
```

BLE Spec 标准接口，用于设置 extended scan 参数。详细可参考《Core\_5.4》(Vol 4/Part E/7.8.64 "LE Set Extended Scan Parameters command")，并结合 SDK 该 API 的注释来理解。

```
ble_sts_t blc_ll_setExtScanEnable (scan_en_t extScan_en, dupe_flg_en_t filter_duplicate,
↳ scan_durn_t duration, scan_period_t period);
```

BLE Spec 标准接口，用于打开/关闭 extended scan。详细可参考《Core\_5.4》(Vol 4/Part E/7.8.65 "LE Set Extended Scan Enable command")，并结合 SDK 该 API 的注释来理解。

### 3.5.4 周期扫描 (Periodic SCAN)

扫描设备可以有以下两种方式中与周期广播序列 (train) 同步：

- (1) 设备本身可以扫描 ADV\_EXT\_IND 及 AUX\_ADV\_IND PDU，并使用 SyncInfo 字段的内容，如：周期广播 interval、时序偏移和要使用的信道等信息来建立与 periodic advertising 的同步。(参考 [周期广播 \(Periodic Advertising\)](#) 和 [扩展扫描 \(Extended SCAN\)](#) 小节)。

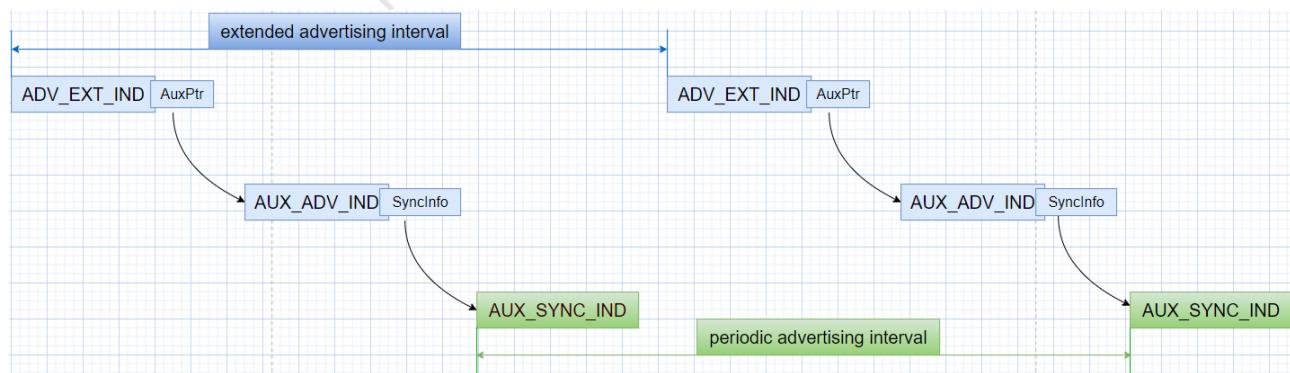


Figure 3.81: 周期广播事件

- (2) 设备可以通过 LE-ACL 连接，从另一个设备接收此信息，不需要在 primary channel 扫描即可与周期广播设备同步。这个过程我们称之为 Periodic Advertising Sync Transfer—PAST。参考[Periodic Advertising Sync Transfer \(PAST\)](#)章节。



### 3.5.4.1 periodic scan 相关的 API 介绍

```
void    blc_ll_initPeriodicAdvertisingSynchronization_module(void);
```

初始化 periodic scan module，只有进行了相应模块的初始化，相应的 feature 才会生效，对应的功能函数才会链接到执行（bin）文件中。

```
ble_sts_t    blc_ll_periodicAdvertisingCreateSync ( option_msk_t options, u8 adv_sid, u8
↪  adv_adrType, u8 *adv_addr, u16 skip, sync_tm_t sync_timeout, u8 sync_cte_type);
```

BLE Spec 标准接口，用于和周期广播同步，开始接收周期广播包。详细可参考《Core\_5.4》(Vol 4/Part E/7.8.67 “LE Periodic Advertising Create Sync command”), 并结合 SDK 该 API 的注释来理解。

```
ble_sts_t    blc_ll_periodicAdvertisingCreateSyncCancel (void);
```

BLE Spec 标准接口，在使用 blc\_ll\_periodicAdvertisingCreateSync 准备同步相应 periodic advertising，但是还未同步上的时候，调用该 API 可以取消同步，但是如果已经同步上，调用该 API 不会有作用，会返回 command disallowed。详细可参考《Core\_5.4》(Vol 4/Part E/7.8.68 “LE Periodic Advertising Create Sync Cancel command”), 并结合 SDK 该 API 的注释来理解。

```
ble_sts_t    blc_ll_periodicAdvertisingTerminateSync (u16 sync_handle);
```

BLE Spec 标准接口，停止与 sync\_handle 指定的 periodic advertising 同步。详细可参考《Core\_5.4》(Vol 4/Part E/7.8.69 “LE Periodic Advertising Terminate Sync command”), 并结合 SDK 该 API 的注释来理解。

```
ble_sts_t    blc_ll_addDeivceToPeriodicAdvertiserList (u8 adv_adrType, u8 *adv_addr, u8 adv_sid);
```

BLE Spec 标准接口，添加指定的设备及广播集 ID 到 Periodic Advertiser List，类似 whitelist。详细内容可参考 (Vol 4/Part E/7.8.70 “LE Add Device To Periodic Advertiser List command”), 并结合 SDK 该 API 的注释来理解。

关于 Periodic sync establishment filter policy 可参考《Core\_5.4》(Vol 6/Part B/4.3.5 Periodic sync establishment filter policy)

```
ble_sts_t    blc_ll_removeDeivceFromPeriodicAdvertiserList (u8 adv_adrType, u8 *adv_addr, u8
↪  adv_sid);
```

BLE Spec 标准接口，从 Periodic Advertiser List 删除指定内容。详细可参考《Core\_5.4》(Vol 4/Part E/7.8.71 “LE Remove Device From Periodic Advertiser List command”), 并结合 SDK 该 API 的注释来理解。

```
ble_sts_t    blc_ll_clearPeriodicAdvertiserList (void);
```

BLE Spec 标准接口，删除 Periodic Advertiser List 中所有内容。详细可参考《Core\_5.4》(Vol 4/Part E/7.8.72 “LE Clear Periodic Advertiser List command”), 并结合 SDK 该 API 的注释来理解。



```
ble_sts_t    blc_ll_readPeriodicAdvertiserListSize (u8 *perdAdvListSize);
```

BLE Spec 标准接口, 读取 Periodic Advertiser List 可以存储信息的最大数量。详细可参考《Core\_5.4》(Vol 4/Part E/7.8.73 "LE Read Periodic Advertiser List Size command"), 并结合 SDK 该 API 的注释来理解。

```
ble_sts_t    blc_ll_periodicAdvertisingReceiveEnable (u16 sync_handle, sync_adv_rcv_en_msk  
↪ enable);
```

BLE Spec 标准接口, 打开或关闭接收到的 periodic advertising 上报给 host。详细可参考《Core\_5.4》(Vol 4/Part E/7.8.88 "LE Set Periodic Advertising Receive Enable command"), 并结合 SDK 该 API 的注释来理解。

### 3.5.5 Periodic Advertising Sync Transfer (PAST)

PAST 是《Core\_V5.1》新增加的特性, 主要用于通知接收设备如何同步到周期广播。蓝牙核心规范包含完整的详细信息, 请参考《Core\_V5.4》, Vol 6, Part B **4.6.23 Periodic Advertising Sync Transfer - Sender** 和 **4.6.24 Periodic Advertising Sync Transfer - Recipient**。



### 3.5.5.1 PAST 模式 1

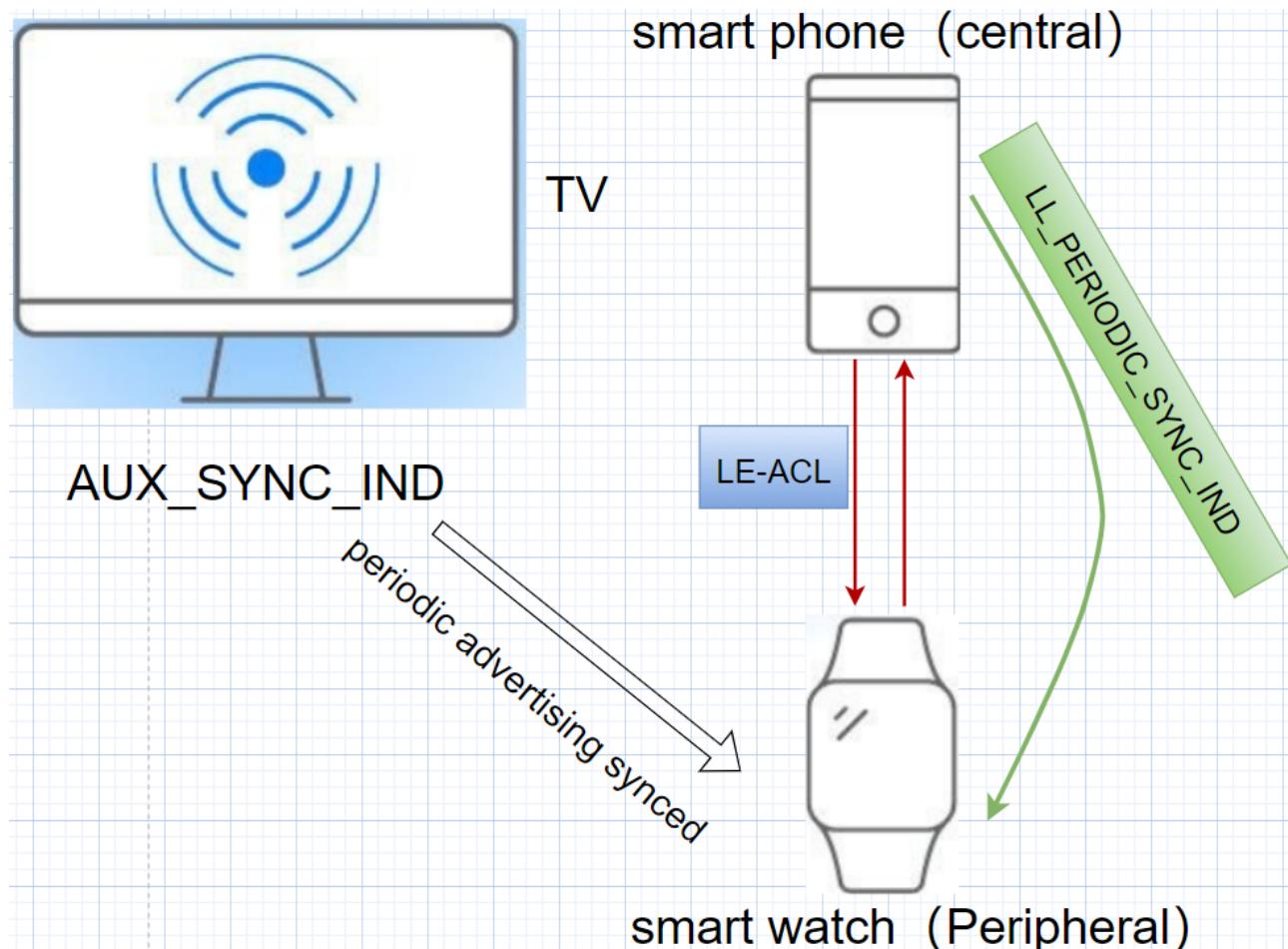


Figure 3.82: PAST 事件

如上图, smart phone 已经与 TV 进行了周期广播同步, 可以收到周期广播数据包 AUX\_SYNC\_IND。同时, smart phone 可与 smart watch 建立 ACL 连接。在没有 PAST 的情况下, 如果 smart watch 希望从 TV 处获得周期广播数据包, 那么 smart watch 需要自行扫描并与 TV 进行周期广播同步。smart watch 在完成这一过程时需要耗费额外的时间和电能, 然而此类设备往往电量有限。

在有 PAST 的情况下, 针对相同的情景, smart phone 能够通过 LE ACL 将周期广播同步信息通过 LL\_PERIODIC\_SYNC\_IND 传输到 smart watch, smart watch 可通过同步信息与 TV 进行周期广播同步。PAST 能够简化同步过程并帮助电量有限的设备节省电能。

### 3.5.5.2 PAST 模式 2

PAST 涉及三方设备: 广播设备 (TV)、辅助器 (smart phone, 作为 Central)、接收设备 (smart watch, 作为 Peripheral), 其中广播设备 (TV) 和辅助器 (smart phone) 在实现上可以是同一个设备, 也可以独立存在。当辅助器 (central) 自身就是广播源时即为 PAST 模式 2, 如下图所示。

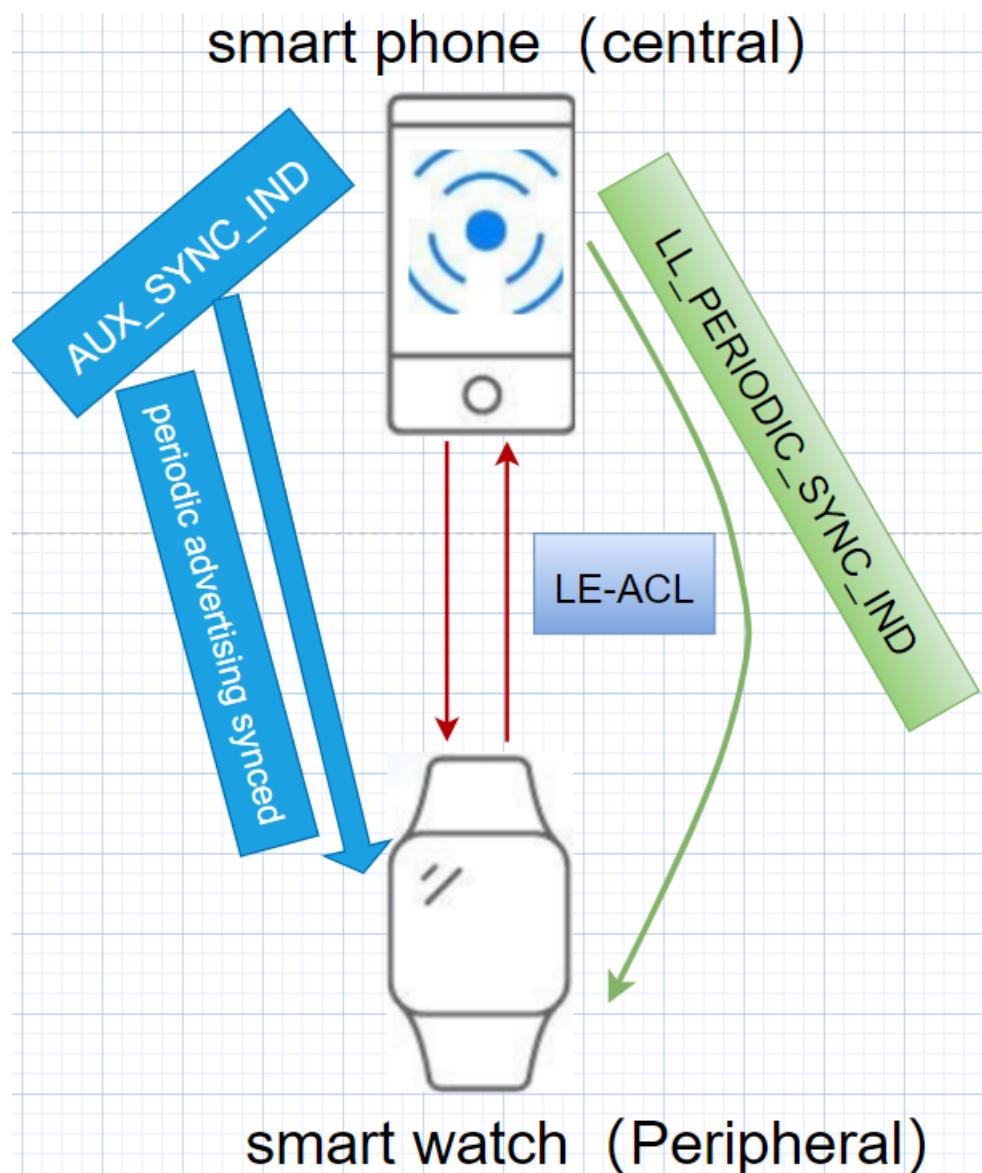


Figure 3.83: PAST 事件

### 3.5.5.3 PAST 相关 API 介绍

```
void     blc_ll_initPAST_module(void);
```

初始化 PAST module，只有进行了相应模块的初始化，相应的 feature 才会生效，对应的功能函数才会链接到执行 (bin) 文件中。

```
ble_sts_t    blc_ll_periodicAdvSyncTransfer(u16 connHandle, u16 serviceData, u16 syncHandle);
```

BLE Spec 标准接口，指示 controller 发送 LL\_PERIODIC\_SYNC\_IND 给到与自己 ACL 连接的设备，具体设备有 API 参数指定。详细可参考《Core\_5.4》(Vol 4/Part E/7.8.89 “LE Periodic Advertising Sync Transfer command”)，并结合 SDK 该 API 的注释来理解。

**注意:**

这个 API 应用场景是 PAST 模式 1, 即广播设备、辅助设备不是同一个设备。syncHandle 是 LE Periodic Advertising Sync Established event 上报上来的, 作为 periodic adv ID。

```
ble_sts_t blc_ll_periodicAdvSetInfoTransfer(u16 connHandle, u16 serviceData, u8
    ↵ advHandle);
```

BLE Spec 标准接口, 指示 controller 发送 LL\_PERIODIC\_SYNC\_IND 给到与自己 ACL 连接的设备, 具体设备有 API 参数指定。详细可参考《Core\_5.4》(Vol 4/Part E/7.8.90 "LE Periodic Advertising Set Info Transfer command"), 并结合 SDK 该 API 的注释来理解。

**注意:**

这个 API 应用场景是 PAST 模式 2, 即广播设备、辅助设备是同一个设备。同一个设备, 不会有 LE Periodic Advertising Sync Established event, 就不会有 syncHandle 来标记的 periodic advertising。但是设备自身是知道 advertising set ID 的, 因此可以通过 advertising set ID 来指定 periodic advertising。

```
ble_sts_t blc_ll_setPeriodicAdvSyncTransferParams(u16 connHandle, u8 mode, u16 skip, u16
    ↵ syncTimeout, u8 cteType);
```

BLE Spec 标准接口, 指示接收设备 (模式 1/2 中的 Peripheral) 接收到 LL\_PERIODIC\_SYNC\_IND 时, controller 该如何处理。可以根据 LL\_PERIODIC\_SYNC\_IND 包含的信息同步到相应的周期广播, 也可以忽略 LL\_PERIODIC\_SYNC\_IND。

详细可参考《Core\_5.4》(Vol 4/Part E/7.8.91 "LE Set Periodic Advertising Sync Transfer Parameters command"), 并结合 SDK 该 API 的注释来理解。

接收设备可能存在多个 ACL 连接, 可使用 connHandle 来指定某个 ACL 连接。

```
ble_sts_t blc_ll_setDftPeriodicAdvSyncTransferParams(u8 mode, u8 skip, u16 syncTimeout, u8
    ↵ cteType);
```

BLE Spec 标准接口, 作用同 **blc\_ll\_setPeriodicAdvSyncTransferParams**。不同之处是: 该 API 对所有的 ACL 连接都起作用。如果某个 ACL 连接需要不同于其他的连接, 可以使用 **blc\_ll\_setPeriodicAdvSyncTransferParams** 来设定特定的 ACL connect (connHandle)。详细可参考《Core\_5.4》(Vol 4/Part E/7.8.92 "LE Set Default Periodic Advertising Sync Transfer Parameters command"), 并结合 SDK 该 API 的注释来理解。

### 3.5.6 Periodic Advertising with Response (PAwR)

PAwR 是《Core\_V5.4》新增加的特性。用于通过周期性广播 (具体内容可参考周期广播章节) 将数据和命令发送给特定的同步设备, 同时可以接收同步设备的响应信息, 目前 PAwR 的典型案例是支持电子货架标签 (ESL) 的部署。

#### 3.5.6.1 PAwR 基本原理

根据功能不同, PAwR 中分为广播者和观察者两种角色。广播者负责进行 PAwR 广播, 向观察者发送控制命令和数据, 同时接收观察者的响应数据。观察者负责监听相关的 PAwR 广播, 并做出响应。PAwR 在周期广播的



基础上，充分利用 periodic event 的间隔时间，将这一段时间分为多个 subevent。如下图所示，每个 subevent 分配有唯一的编号，以 ESL 为例，该编号对应 ESL 设备的 group ID，即具有相同 group ID 的 ESL 设备会同时监听对应的 subevent。

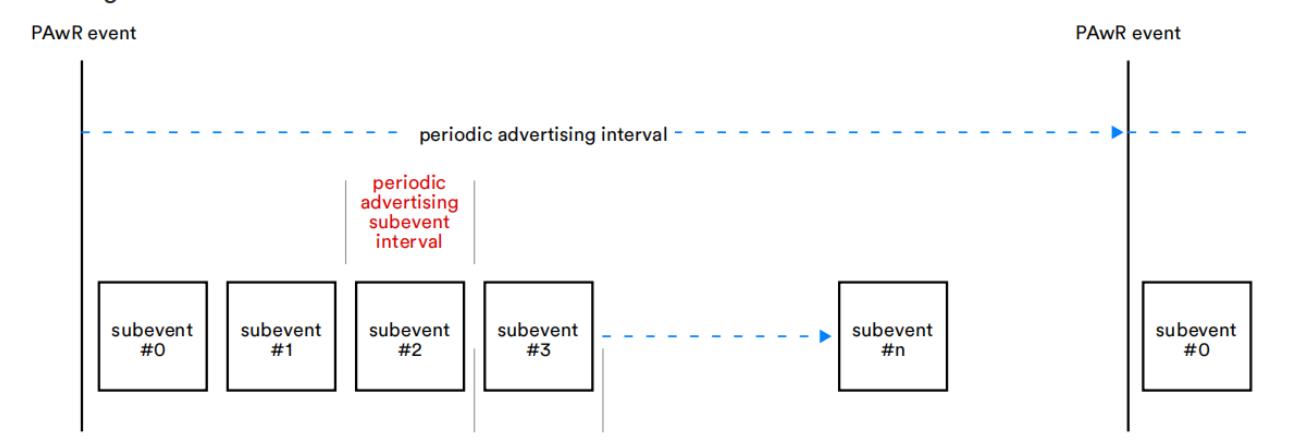


Figure 3.84: PAwR\_subevent

subevent 的结构如下图所示：

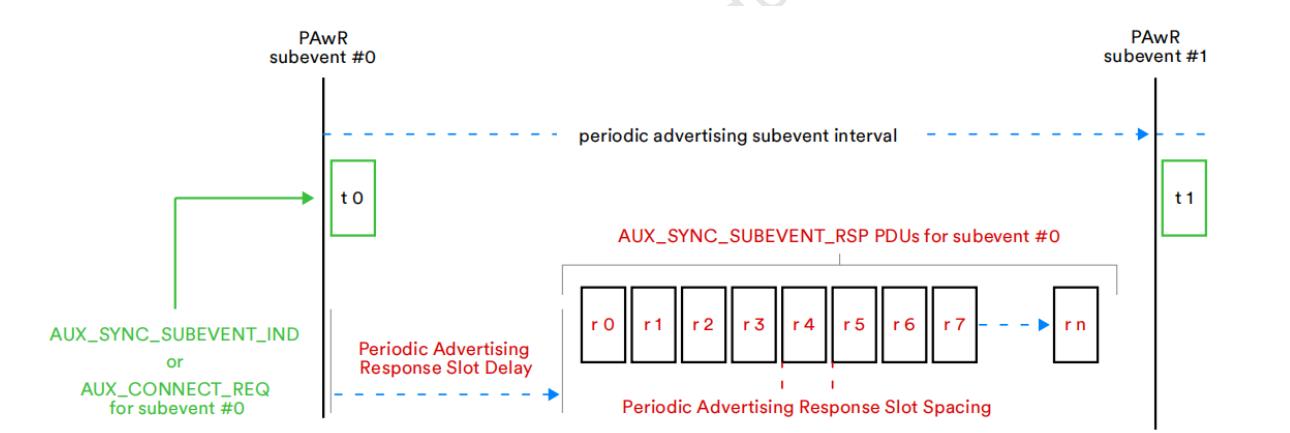


Figure 3.85: PAwR\_rspSlots

在 subevent 起始位置，广播者会发送下面两种类型的同步包：

- AUX\_SYNC\_SUBEVENT\_IND：包含控制命令和数据的同步请求包
- AUX\_CONNECT\_REQ：ACL 连接请求

发送完成后，等待一定的延迟并进入接收状态。可以看到接收的时序中被分为多个 slot，该 slot 被称为响应槽，监听该 subevent 的观察者往往有多个，每个观察者需要在各自对应的响应槽时间内进行响应，响应槽的分配可根据具体场景进行设置。以 ESL 为例，观察者做出响应时的响应槽编号在每一次 subevent 内动态分配，由广播者的同步广播包中的命令顺序和观察者的自身 ID 共同决定，具体分配过程可参考《Electronic Shelf Label Profile》(5.3.1.4.2 "Allocation of response slots to ESLs")。



### 3.5.6.2 PAwR 同步

上面我们提到 PAwR 是在周期广播基础上进行的扩展，subevent 的监听和响应都是在已经建立同步的基础上进行的。为实现同步，作为观察者首先需要知道 PAwR 事件发生周期 (periodic advertising interval)，以及下一次 PAwR 事件发生时刻 (syncPacketWindowOffset)。然后，结合观察者配置的 subevent ID 和响应槽编号，同时需要知道以下信息来确定需要监听和响应的时刻：

- Num\_Subevents：一个周期内的 subevent 数量
- Subevent\_interval：一个 subevent 开始到下一个 subevent 开始的时间
- Response\_Slot\_Delay：从 subevent 开始到第一个响应槽的时间。
- Response\_Slot\_spacing：从一个响应槽开始到下一个响应槽开始的时间
- Num\_Response\_Slots：subevent 中响应槽的数量

上述信息的获取有两种方式，一种是通过观察者设备直接扫描获取，AUX\_ADV\_IND 中 ACAD 部分包含上述信息。第二种是通过 PAST，PAST 是指广播者或第三方设备首先与观察者设备建立 ACL 连接，将包含同步信息的 PAST packet 发送给观察者设备完成同步 (具体内容可参考 PAST 章节)。

### 3.5.6.3 PAwR 相关 API 介绍

#### 3.5.6.3.1 广播端 API

```
ble_sts_t    blc_ll_initPeriodicAdvWrModule_initPeriodicAdvWrSetParamBuffer(u8 *pBuff, int
↪ num_periodic_adv);
void blc_ll_initPeriodicAdvWrDataBuffer(u8 *pSubeventData, int subeventDataLenMax, int
↪ subeventDataCnt);
```

自定义函数，初始化 PAwR 广播者模块和分配的数据空间。

```
//不通过 HCI, host 层直接设置 controller 层相关参数
ble_sts_t    blc_ll_setPeriodicAdvParam_v2(adv_handle_t adv_handle,
                                             u16 advInter_min,
                                             u16 advInter_max,
                                             perfd_adv_prop_t property,
                                             u8 numSubevents,u8 subeventInterval,
                                             u8 responseSlotDelay,
                                             u8 responseSlotSpace,
                                             u8 numResponseSlots);
//通过 HCI command 进行调用
ble_sts_t    blc_hci_le_setPeriodicAdvParam_v2(hci_le_setPeriodicAdvParamV2_cmdParam_t*
↪ pCmdParam);
```

BLE spec 标准接口，对 PAwR 的周期广播参数进行设置。详情请参照《core5.4》(vol4/Part E/7.8.61 “LE Set Periodic Advertising Parameters command”)。



```
ble_sts_t    blc_ll_setPeriodicAdvEnable(u8 per_adv_enable, adv_handle_t adv_handle);
```

BLE spec 标准接口，使能周期广播，PAwR 复用该接口。详情请参照《core5.4》(vol4/Part E/7.8.63 “LE Set Periodic Advertising Enable command”)。

```
//不通过 HCI, host 层直接操作 controler 层 buffer  
ble_sts_t    blc_ll_setPeriodicAdvSubeventData(adv_handle_t adv_handle, u8 num_subevent,  
    ↳ pdaSubevtData_subevtCfg_t* pSubevtCfg);  
//通过 HCI command 进行调用  
ble_sts_t    blc_hci_le_setPeriodicAdvSubeventData(hci_le_setPeridAdvSubeventData_cmdParam_t*  
    ↳ pcmdParam, hci_le_setPeridAdvSubeventDataRetParams_t *pRetParams)
```

BLE spec 标准接口，设置 subevent 中 AUX\_SYNC\_SUBEVENT\_IND 包中数据。详情请参照《core5.4》(vol4/Part E/7.8.125 “LE Set Periodic Advertising Subevent Data command”)。

```
//不通过 HCI, 控制 controler 层建立连接  
ble_sts_t    blc_ll_extended_createConnection_v2 (adv_handle_t adv_handle, u8 subevent,  
    init_fp_t filter_policy, own_addr_type_t ownAddrType, u8 peerAdrType, u8 *peerAddr,  
    ↳ init_phy_t init_phys,  
    scan_inter_t scanInter_0, scan_wind_t scanWindow_0, conn_inter_t conn_min_0, conn_inter_t  
    ↳ conn_max_0, conn_tm_t timeout_0,  
    scan_inter_t scanInter_1, scan_wind_t scanWindow_1, conn_inter_t conn_min_1, conn_inter_t  
    ↳ conn_max_1, conn_tm_t timeout_1,  
    scan_inter_t scanInter_2, scan_wind_t scanWindow_2, conn_inter_t conn_min_2, conn_inter_t  
    ↳ conn_max_2, conn_tm_t timeout_2 );  
//通过 HCI command 进行调用  
ble_sts_t    blc_hci_le_extended_createConnection_v2( hci_le_ext_createConnV2_cmdParam_t *  
    ↳ pCmdParam);
```

BLE spec 标准接口，在对应 subevent 发送 AUX\_CONNECT\_REQ 以建立 ACL 连接。详情请参照《core5.4》(vol4/Part E/7.8.66 “LE Extended Create Connection command”)。

### 3.5.6.3.2 观察端 API

```
ble_sts_t    blc_ll_initPAwRsync_module(int num_pawr_sync);  
ble_sts_t    blc_ll_initPAwRsync_rspDataBuffer(u8 *pdaRspData, int maxLen_pdaRspData);
```

自定义函数，初始化 PAwR 观察者模块和分配的响应数据空间。

```
ble_sts_t    blc_hci_le_setPeriodicSyncSubevent(u16 sync_handle,  
    u16 pda_prop,  
    u8 num_subevent,  
    u8* pSubevent)
```



BLE spec 标准接口，观察者设置自身需要监听的 subevent，可以为多个。详情请参照《core5.4》(vol4/Part E/7.8.127 “LE Set Periodic Sync Subevent command”)。

```
ble_sts_t blc_hci_le_setPAwRsync_rspData( u16 sync_handle,
                                            u16 req_pdaEvtCnt,
                                            u8 req_subEvtCnt,
                                            u8 rsp_subEvtCnt,
                                            u8 rsp_slotIdx,
                                            u8 rspDataLen,
                                            u8* pRspData)
```

BLE spec 标准接口，设置 response 数据。详情请参照《core5.4》(vol4/Part E/7.8.126 “LE Set Periodic Advertising Response Data command”)。



## 4 低功耗管理

低功耗管理 (Low Power Management) 也可以称为功耗管理 (Power Management)，本文档中会简称为 PM。

### 4.1 低功耗驱动

#### 4.1.1 低功耗模式

MCU 正常执行程序时处于 working mode，此时工作电流在 3~7mA 之间。如果需要省功耗需要进入低功耗模式。

低功耗模式 (low power mode) 又称 sleep mode，包括 3 种：suspend mode、deepsleep mode 和 deepsleep retention mode。

Module	suspend	deepsleep retention	deepsleep
Sram	100% keep	first 32K/64K/96K keep, others lost	100% lost
digital register	99% keep	100% lost	100% lost
analog register	100% keep	99% lost	99% lost

上表为 3 种 sleep mode 下 Sram、数字寄存器 (digital register)、模拟寄存器 (analog register) 状态保存的统计说明。

##### (1) Suspend mode (sleep mode 1)

此时程序停止运行，类似一个暂停功能。MCU 大部分硬件模块断电，PM 模块维持正常工作。以 B91 为例，此时 IC 电流在 40-50uA 之间。当 suspend 被唤醒后，程序继续执行。

suspend mode 下所有的 SRAM 和 analog register 都能保存状态，绝大部分 digital register 都保持状态。digital register 中存在少量会掉电的，如 baseband 电路中少量的 digital register，user 需要关注的是 API rf\_set\_power\_level\_index() 设置的寄存器，本文档前面已经介绍，这个 API 需要在每次 suspend 醒来后都重新调用一次。

##### (2) Deepsleep mode (sleep mode 2)

此时程序停止运行，MCU 绝大部分的硬件模块都断电，PM 硬件模块维持工作。在 deepsleep mode 下 IC 电流小于 1uA。如果内置 flash 的 standby 电流出现较大的 1uA 左右，可能导致测量到 deepsleep 为 1~2uA。deepsleep mode wake up 时，MCU 将重新启动，类似于上电的效果，程序会重新开始进行初始化。

Deepsleep mode 下，除了 analog register 上有少数几个 register 能保存状态，其他所有 Sram、digital register、analog register 全部掉电丢失。

##### (3) Deepsleep retention mode (sleep mode 3)

上面的 deepsleep mode，电流很低，但是无法存储 Sram 信息；suspend mode Sram 和 register 可以保持不丢，但是电流偏高。



为了实现一些需要 sleep 时电流很低又要能够确保 sleep 唤醒后能立刻恢复状态的应用场景（比如 BLE 长睡眠维持连接），tl\_ble\_sdk 增加了一种 sleep mode 3：deepsleep with Sram retention mode，简称 deepsleep retention（或 deep retention）。

deepsleep retention mode 也是一种 deepsleep，MCU 绝大部分的硬件模块都断电，PM 硬件模块维持工作。功耗是在 deepsleep mode 基础上增加 retention Sram 消耗的电，电流在 2~3uA 之间。deepsleep retention mode wake up 时，MCU 将重新启动，程序会重新开始进行初始化。

deepsleep retention mode 和 deepsleep mode 在 register 状态保存方面表现一致，几乎全部掉电。deepsleep retention mode 跟 deepsleep mode 相比，Sram 的前 32K/64K/96K 可以保持不掉电，剩余的 Sram 全部掉电。

#### 4.1.2 低功耗唤醒源

tl\_ble\_sdk MCU 的低功耗唤醒源示意图如下，suspend/deepsleep/deepsleep retention 都可以被 GPIO PAD 和 timer 唤醒。tl\_ble\_sdk 中，只关注 2 种唤醒源，如下所示（注意 code 中 PM\_TIM\_RECOVER\_START 和 PM\_TIM\_RECOVER\_END 两个定义不是唤醒源）：

```
typedef enum {
    PM_WAKEUP_PAD          = BIT(3),
    .....
    PM_WAKEUP_TIMER         = BIT(5),
    .....
}pm_sleep_wakeup_src_e;
```

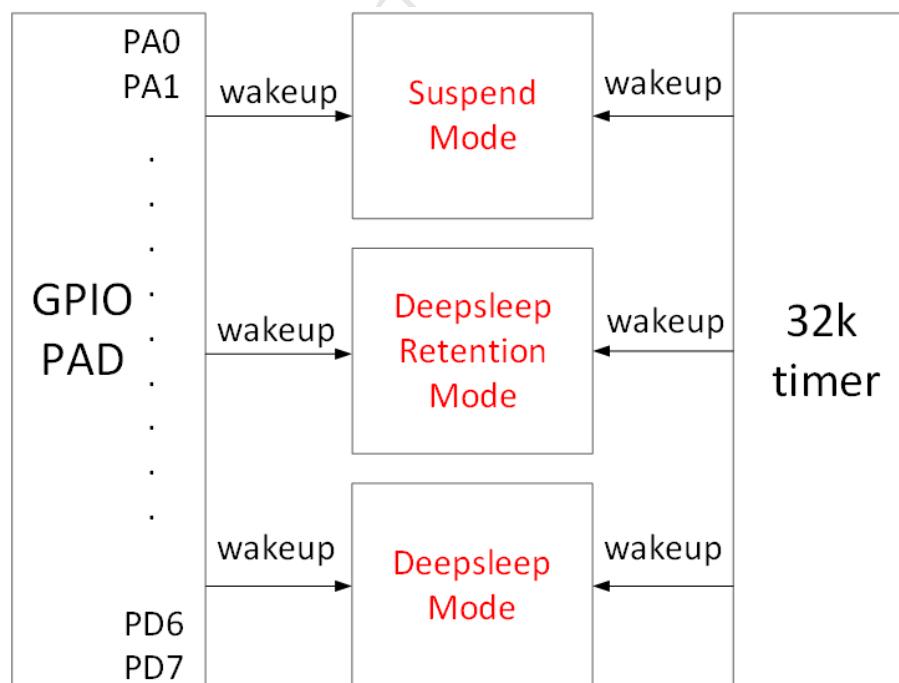


Figure 4.1: MCU 硬件唤醒源

如上图所示，MCU 的 suspend/deepsleep/deepsleep retention 在硬件上有 2 个唤醒源：TIMER、GPIO PAD。



- 唤醒源 PM\_WAKEUP\_TIMER 来自硬件 32k timer (32k RC timer or 32k Crystal timer)。32k timer 在 SDK 中已经被正确初始化，user 在使用时不需要任何配置，只需要在 cpu\_sleep\_wakeup() 中设置该唤醒源即可。
- 唤醒源 PM\_WAKEUP\_PAD 来自 GPIO 模块，除 MSPI 4 个管脚外所有的 GPIO (PAx/PBx/PCx/PDx/PEx) 的高低电平都具有唤醒功能。

配置 GPIO PAD 唤醒 sleep mode 的 API:

```
typedef enum{
    Level_Low=0,
    Level_High,
} GPIO_LevelTypeDef;
void cpu_set_gpio_wakeup (GPIO_PinTypeDef pin, GPIO_LevelTypeDef pol, int en);
```

pin 为 GPIO 定义。

pol 为唤醒极性定义：Level\_High 表示高电平唤醒，Level\_Low 表示低电平唤醒。

en: 1 表示 enable，0 表示 disable。

举例说明：

```
cpu_set_gpio_wakeup (GPIO_PC2, Level_High, 1); //GPIO_PC2 PAD 唤醒打开，高电平唤醒
cpu_set_gpio_wakeup (GPIO_PC2, Level_High, 0); //GPIO_PC2 PAD 唤醒关闭
cpu_set_gpio_wakeup (GPIO_PB5, Level_Low, 1); //GPIO_PB5 PAD 唤醒打开，低电平唤醒
cpu_set_gpio_wakeup (GPIO_PB5, Level_Low, 0); //GPIO_PB5 PAD 唤醒关闭
```

### 4.1.3 低功耗模式的进入和唤醒

在 tl\_ble\_sdk 中 suspend 和 deepsleep retention 由 stack 进行管控，不推荐客户自己设置进入 suspend/deepsleep retention。不过 user 可以设置进入 deepsleep 模式。

设置 MCU 进入睡眠和唤醒的 API 为：

```
int cpu_sleep_wakeup (pm_sleep_mode_e sleep_mode, SleepWakeupSrc_TypeDef wakeup_src,
unsigned int wakeup_tick);
```

- 第一个参数 sleep\_mode：设置 sleep mode，有以下 4 个选择，目前客户可以选择的只有一个：deepsleep mode。（suspend 和 deepsleep retention 由 stack 管控）

```
typedef enum {
    .....
    DEEPSLEEP_MODE          = 0x30,
    .....
}pm_sleep_mode_e;
```

- 第二个参数 wakeup\_src：设置当前的 suspend/deepsleep 的唤醒源，参数只能是 PM\_WAKEUP\_PAD、PM\_WAKEUP\_TIMER 中的一个或者多个。如果 wakeup\_src 为 0，那么进入低功耗 sleep mode 后，无法被唤醒。



- 第三个参数 wakeup\_tick：当 wakeup\_src 中设置了 PM\_WAKEUP\_TIMER 时，需要设置 wakeup\_tick 来决定 timer 在何时将 MCU 唤醒。如果没有设置 PM\_WAKEUP\_TIMER 唤醒，该参数无意义。

wakeup\_tick 的值是一个绝对值，按照本文档前面介绍的 System Timer tick 来设置，当 System Timer tick 的值达到这个设定的 wakeup\_tick 后，sleep mode 被唤醒。wakeup\_tick 的值需要根据当前的 System Timer tick 的值，加上由需要睡眠的时间换算成的绝对时间，才可以有效地控制睡眠时间。如果没有考虑当前的 System Timer tick，直接对 wakeup\_tick 进行设置，唤醒的时间点就无法控制。

由于 wakeup\_tick 是绝对时间，必须在 32bit 的 System Timer tick 能表示的范围之内，所以这个 API 能表示的最大睡眠时间是有限的。目前的设计是最大睡眠时间为 32bit 能表示的最大 System Timer tick 对应时间的 7/8。System Timer tick 最大能表示大概 268s，那么最长 sleep 时间时间为  $268 * 7 / 8 = 234s$ ，即下面 delta\_Tick 不能超过 234s。

```
cpu_sleep_wakeup(SUSPEND_MODE, PM_WAKEUP_TIMER, clock_time() + delta_tick);
```

返回值为当前 sleep mode 的唤醒源的集合，该返回值各 bit 对应表示的唤醒源为：

```
typedef enum {
    .....
    WAKEUP_STATUS_TIMER      = BIT(1),
    WAKEUP_STATUS_PAD        = BIT(3),
    .....
    STATUS_GPIO_ERR_NO_ENTER_PM = BIT(7),
    .....
}pm_wakeup_status_e;
```

- a) WAKEUP\_STATUS\_TIMER 这个 bit 为 1，说明当前 sleep mode 是被 Timer 唤醒。
- b) WAKEUP\_STATUS\_PAD 这个 bit 为 1，说明当前 sleep mode 是被 GPIO PAD 唤醒。
- c) WAKEUP\_STATUS\_TIMER 和 WAKEUP\_STATUS\_PAD 同时为 1 时，表示 Timer 和 GPIO PAD 两个唤醒源同时生效了。
- d) STATUS\_GPIO\_ERR\_NO\_ENTER\_PM 是一个比较特殊的状态，表示当前发生了 GPIO 唤醒错误：比如当设置了某个 GPIO PAD 高电平唤醒，而在这个 GPIO 为高电平的时候尝试调用 cpu\_sleep\_wakeup 进入 suspend，且设置了 PM\_WAKEUP\_PAD 唤醒源。此时会出现无法进入 suspend，MCU 立刻退出 cpu\_sleep\_wakeup 函数，给出返回值 STATUS\_GPIO\_ERR\_NO\_ENTER\_PM。

一般采用如下的形式来控制睡眠时间：

```
cpu_sleep_wakeup (SUSPEND_MODE , PM_WAKEUP_TIMER, clock_time() + delta_Tick);
```

delta\_Tick 是一个相对的时间（比如  $100 * \text{CLOCK\_16M\_SYS\_TIMER\_CLK\_1MS}$ ），加上当前的 clock\_time() 就变成了绝对时间。

举例说明 cpu\_sleep\_wakeup 的用法：

```
cpu_sleep_wakeup (SUSPEND_MODE , PM_WAKEUP_PAD, 0);
```

程序执行该函数时进入 suspend mode，只能被 GPIO PAD 唤醒。



```
cpu_sleep_wakeup (DEEPSLEEP_MODE , PM_WAKEUP_TIMER, clock_time() + 10*  
↪ CLOCK_16M_SYS_TIMER_CLK_1MS);
```

程序执行该函数时进入 deepsleep mode，只能被 Timer 唤醒，唤醒时间为当前时间加上 10ms，所以 deepsleep 时间为 10ms。

```
cpu_sleep_wakeup (DEEPSLEEP_MODE , PM_WAKEUP_PAD | PM_WAKEUP_TIMER,  
clock_time() + 50* CLOCK_16M_SYS_TIMER_CLK_1MS);
```

程序执行该函数时进入 deepsleep 模式，可被 GPIO PAD 和 Timer 唤醒，Timer 唤醒的时间设置为 50ms。如果在 50ms 结束之前触发了 GPIO 的唤醒动作，MCU 会被 GPIO PAD 唤醒；如果 50ms 内无 GPIO 动作，MCU 会被 Timer 唤醒。

```
cpu_sleep_wakeup (DEEPSLEEP_MODE, PM_WAKEUP_PAD, 0);
```

程序执行该函数时进入 deepsleep mode，可被 GPIO PAD 唤醒。

#### 4.1.4 低功耗唤醒后运行流程

当 user 调用 API `cpu_sleep_wakeup()` 后，MCU 进入 sleep mode；当唤醒源触发 MCU 唤醒后，对于不同的 sleep mode，MCU 的软件运行流程不一致。

下面详细介绍 suspend、deepsleep、deepsleep retention 3 种 sleep mode 被唤醒后的 MCU 运行流程。请参考下图。

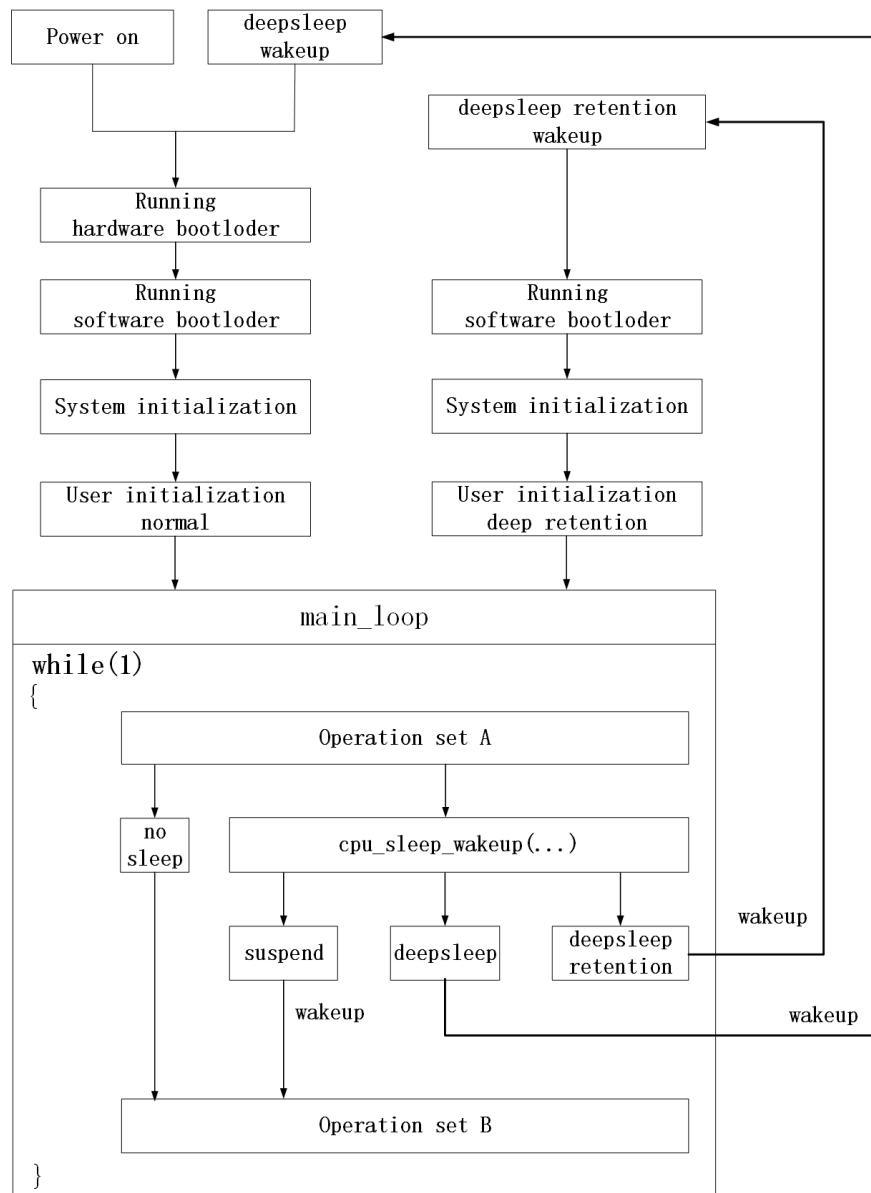


Figure 4.2: Sleep Mode Wakeup Work Flow

MCU 上电 (Power on) 之后，各流程的介绍：

(1) 运行硬件 bootloader (Run hardware bootloader)

MCU 硬件上执行一些固定的动作，这些动作固化在硬件上，软件无法修改。

举几个例子说明一下这些动作，比如：读 flash 的 boot 启动标记，判断当前应该运行的 firmware 是存储在 flash 地址 0 上的，还是在 flash 地址 0x20000 上的（跟 OTA 相关）；读 flash 相应位置的值，判断当前需要从 flash 上拷贝多少数据到 Sram，作为常驻内存的数据（参考第 2 章对 Sram 分配的介绍）。

运行硬件 bootloader 部分由于涉及到 flash 上数据拷贝到 sram，一般执行时间较长，比如拷贝 10K 数据大概耗时 5ms 左右。

(2) 运行软件 bootloader (Run software bootloader)



hardware bootloader 运行结束之后，MCU 开始运行 software bootloader。Software bootloader 就是前面介绍过的 vector 端。

Software bootloader 是为了给后面 C 语言程序的运行设置好内存环境，可以理解为整个内存的初始化。

### (3) 系统初始化 (System initialization)

System initialization 对应 main 函数中 cpu\_wakeup\_init 到 user\_init 之前各硬件模块初始化（包括 cpu\_wakeup\_init、rf\_drv\_init、gpio\_init、clock\_init），设置各硬件模块的数字/模拟寄存器状态。

### (4) 用户初始化 (User initialization)

User initialization 对应 SDK 中函数 user\_init 或 user\_init\_normal/ user\_init\_deepRetn。

### (5) main\_loop

User initialization 完成后，进入 while(1) 控制的 main\_loop。main\_loop 中进入 sleep mode 之前的一系列操作称为“Operation Set A”，sleep 唤醒之后一系列操作称为“Operation Set B”。

对照上图 sleep mode 流程分析。

### (6) no sleep

如果没有 sleep mode，MCU 的运行流程为在 while(1) 中循环，反复执行“Operation Set A” -> “Operation Set B”。

### (7) suspend

如果调用 cpu\_sleep\_wakeup 函数进入 suspend mode，当 suspend 被唤醒后，相当于 cpu\_sleep\_wakeup 函数的正常退出，MCU 运行到“Operation Set B”。

suspend 是最干净的 sleep mode，在 suspend 期间所有的 Sram 数据能保持不变，所有的数字/模拟寄存器状态也保持不变（只有几个特殊的例外）；suspend 唤醒后，程序接着原来的位置运行，几乎不需要考虑任何 sram 和寄存器状态的恢复。suspend 的缺点是功耗偏高。

### (8) deepsleep

如果调用 cpu\_sleep\_wakeup 函数进入 deepsleep mode，当 deepsleep 被唤醒后，MCU 会重新回到 Run hardware bootloader。

可以看出，deepsleep wake\_up 跟 Power on 的流程是几乎一致的，所有的软硬件初始化都得重新做。

MCU 进入 deepsleep 后，所有的 Sram 和数字/模拟寄存器（只有几个模拟寄存器例外）都会掉电，所以功耗很低，MCU 电流小于 1uA。

### (9) deepsleep retention

如果调用 cpu\_sleep\_wakeup 函数进入 deepsleep retention mode，当 deepsleep retention 被唤醒后，MCU 会重新回到 Run software bootloader。

deepsleep retention 是介于 suspend 和 deepsleep 之间的一种 sleep mode。

suspend 因为要保存所有的 sram 和寄存器状态而导致电流偏高；deepsleep retention 不需要保存寄存器状态，Sram 只保留前 32K/64K/96K 不掉电，所以功耗比 suspend 低很多，只有 2uA 左右。

deepsleep wake\_up 后需要把所有的流程重新运行一遍，而 deepsleep retention 可以跳过“Run hardware bootloader”这一步，这是因为 Sram 的前 32K/64K/96K 上数据是不丢的，不需要再从 flash 上重新拷贝一次。但由于 Sram 上 retention area 有限，“run software bootloader”无法跳过，必须得执行；由于 deepsleep



retention 无法保存寄存器状态，所以 system initialization 必须执行，寄存器的初始化需要重新设置。deepsleep retention wake\_up 后的 User initialization deep retention 可以做一些优化改进，和 MCU power on/deepsleep wake\_up 后的 User initialization normal 做区分处理。

#### 4.1.5 API pm\_is MCU\_deepRetentionWakeup

由图“sleep mode wakeup work flow”可以看到，MCU power on、deepsleep wake\_up、deepsleep retention wake\_up 这 3 种情况都需要经过 Run software bootloader、System initialization、User initialization。

在运行 system initialization、user initialization 2 个步骤时，user 需要知道当前 MCU 是否被 deepsleep retention wake\_up 的，以便做一些区分子 power on、deepsleep wake\_up 的设置。PM driver 提供判断是否 deepsleep retention wake\_up 的 API 为：

```
int pm_is_MCU_deepRetentionWakeup(void);
```

return 值为 1，表示 deepsleep retention wake\_up；return 值为 0，表示 power on 或 deepsleep wake\_up。

## 4.2 BLE 低功耗管理

### 4.2.1 BLE PM 初始化

如果使用了低功耗模式，需要将 BLE PM 模块初始化，调用下面 API 即可。

```
void blc_ll_initPowerManagement_module(void);
```

若不需要低功耗模式，不调用此 API，则 PM 相关的代码和变量都不会被编译到程序中，可以节省 firmware size 和 sram size。

### 4.2.2 BLE PM for Link Layer

tl\_ble\_sdk 中对 Legacy advertising state、Scanning state、ACL connection central 和 ACL connection peripheral 做了低功耗管理。

需要说明的是，SDK 目前 peripheral 使用 latency 是有限制条件的，如果不满足限制条件每个 interval 都会进行收发包。即便是作为 peripheral 接受了对方 central 的连接参数，其中 latency 不为 0，SDK 也会按照 latency 为 0 进行 RF 收发数据。

peripheral 使用 latency 的限制条件：

- (1) 只有 Legacy advertising 和 ACL peripheral 任务。
- (2) ACL peripheral 只有 1 个连接（后面 SDK 会优化支持更多的 peripheral 连接）。
- (3) 如果有 Legacy advertising，最小广播间隔需要大于 195ms。

对于 Idle state，SDK 不提供任何低功耗管理。由于此状态不涉及 BLE RF 任何动作（即 blc\_sdk\_main\_loop 函数完全无效），user 可以自行调用 PM driver 去做一些低功耗管理。



注意：用户需调用 API `blc_ll_isBleTaskIdle` 检查 BLE 协议栈是否处于 Idle 状态！

#### 4.2.2.1 Sleep for advertising “only advertising”

当只使能了广播，关闭了 scan 功能，即 Link Layer 处于 advertising state 时，时序如下：

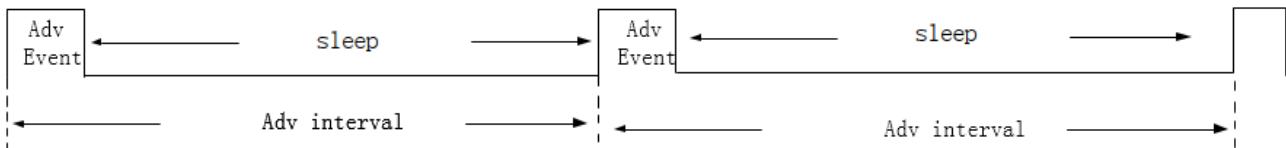


Figure 4.3: sleep 处于 advertising 状态时序

当到达 advertising 时间时，就会从 sleep 唤醒，然后处理广播事件。处理完成后，stack 会判断下一次 Adv Event 的时间点到目前时间的差值，如果满足条件，就会进入 sleep 降低功耗。Adv Event 消耗的时间和具体情况有关，比如：用户只设置了 37channel；ADV packet 长度比较小；在 channel37 或 38 收到了 SCAN\_REQ 或 CONNECT\_IND 等等。

#### 4.2.2.2 Sleep for scanning “only scanning”

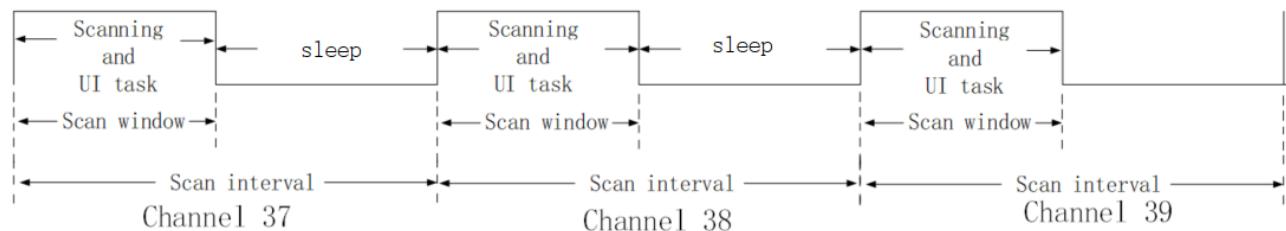


Figure 4.4: sleep for scanning for only scanning

根据 Scan window 的大小决定实际的 scanning 时间，如果 Scan window 等于 Scan interval，所有的时间都在 scanning；如果 Scan window 小于 Scan interval，从 Scan interval 的前面部分开始去分配时间来进行 scanning，等效时间参考 Scan window。

图上所示的 Scan window 大约是 Scan interval 的 40%，在前 40% 的时间里，Link Layer 处于 scanning 状态，PHY 层在收包，同时用户可以利用这段时间在 main\_loop 中执行自己的 UI task。后 60% 的这段时间 MCU 进入 sleep 以降低整机功耗。

设置占比的 API 如下：

```
blc_ll_setScanParameter(SCAN_TYPE_PASSIVE, SCAN_INTERVAL_200MS, SCAN_WINDOW_50MS,  
↪ OWN_ADDRESS_PUBLIC, SCAN_FP_ALLOW_ADV_ANY);
```



#### 4.2.2.3 Sleep for connection

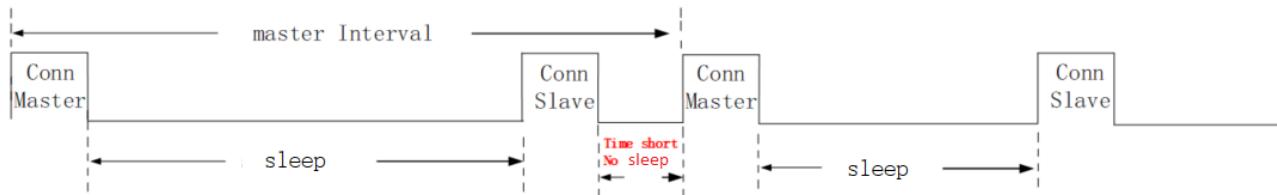


Figure 4.5: sleep for connection

进入 sleep 条件是：

- (1) 下一个任务距离目前任务结束的时间间隔大小；
- (2) RX FIFO 中是否有数据未处理；
- (3) BRX POST 和 BTX POST 执行完成；
- (4) 设备自身没有任何事件 pending。

如果距离下一个任务的时间间隔比较大，且 RX FIFO 没有数据，也没有任何事件 pending，当 BRX POST 或 BTX POST 执行后，底层就会让 MCU 进入 sleep。等到下一个任务即将到来前，timer 唤醒 MCU 开始执行该任务。

#### 4.2.3 相关变量

BLE PM 软件处理流程部分会出现很多变量，用户有必要了解这些变量。

tl\_ble\_sdk 在底层定义了结构体 "st\_llms\_pm\_t"，下面只列出该结构体部分变量（API 介绍时需要用到的变量）。

```
typedef struct {
    u8  deepRt_en;
    u8  deepRet_type;
    u8  wakeup_src;
    u16 sleep_mask;
    u16 user_latency;
    u32 deepRet_thresTick;
    u32 deepRet_earlyWakeupTick;
    u32 sleep_taskMask;
    u32 next_task_tick;
    u32 current_wakeup_tick;
}st_llms_pm_t;

st_llms_pm_t  blmsPm;
```

注意：



上述的结构体变量被封装在 library 中，这里给出定义只是为了方便后面的介绍，用户不允许对这个结构体变量进行任何操作。

下面的介绍中会经常出现类似 “blmsPm.sleep\_mask” 的变量。

#### 4.2.4 API blc\_pm\_setSleepMask

用于配置低功耗管理的 API：

```
void blc_pm_setSleepMask (sleep_mask_t mask);
```

使用 blc\_pm\_setSleepMask 设置 blmsPm.sleep\_mask (默认值为 PM\_SLEEP\_DISABLE)。

这个 API 的源码为：

```
void blc_pm_setSleepMask (sleep_mask_t mask)
{
    u32 r = irq_disable();
    .....
    blmsPm.sleep_mask = mask;
    .....
    u32 r = irq_disable();
}
```

blmsPm.sleep\_mask 的设置，可以选择下面几个值中的一个，也可以选择多个值的“或”操作。

```
typedef enum {
    PM_SLEEP_DISABLE          = 0,
    PM_SLEEP_LEG_ADV          = BIT(0),
    PM_SLEEP_LEG_SCAN         = BIT(1),
    PM_SLEEP_ACL_PERIPH     = BIT(2),
    PM_SLEEP_ACL_CENTRAL     = BIT(3),
    PM_SLEEP_EXT_ADV          = BIT(4),
    PM_SLEEP_CIS_PERIPH     = BIT(8),
    PM_SLEEP_CIS_CENTRAL     = BIT(9),
} sleep_mask_t;
```

PM\_SLEEP\_DISABLE 表示 sleep disable，不允许 MCU 进入 sleep。

PM\_SLEEP\_LEG\_ADV 和 PM\_SLEEP\_LEG\_SCAN 分别用于控制 Legacy advertising state 和 Scanning state 时 MCU 进入 sleep。

PM\_SLEEP\_ACL\_PERIPH 和 PM\_SLEEP\_ACL\_CENTRAL 分别用于控制 ACL connection peripheral 和 ACL connection central 时 MCU 进入 sleep。

该 API 最常用的 2 种情况如下：



(1) blc\_pm\_setSleepMask(PM\_SLEEP\_DISABLE);

MCU 不允许进入 sleep。

(2) blc\_pm\_setSleepMask(PM\_SLEEP\_LEG\_ADV | PM\_SLEEP\_LEG\_SCAN | PM\_SLEEP\_ACL\_PERIPHERAL | PM\_SLEEP\_ACL\_CENTRAL);

MCU 在 Legacy advertising state、Scanning state、ACL connection peripheral 和 ACL connection central 时都允许进入 sleep。

#### 4.2.5 API blc\_pm\_setWakeupSource

user 通过上面的 API blc\_pm\_setSleepMask 设置 MCU 进入 sleep mode (suspend 或 deepsleep retention)，通过下面的 API 可设置 sleep mode 的唤醒源。

```
void blc_pm_setWakeupSource(pm_sleep_wakeup_src_e wakeup_src)
{
    blmsPm.wakeup_src = (u8)wakeup_src;
}
```

wakeup\_src 可以选择唤醒源 PM\_WAKEUP\_PAD。

该 API 设置底层变量 blmsPm.wakeup\_src。

MCU 在 Legacy advertising state、Scanning state、ACL connection central 和 ACL connection peripheral 进入 sleep mode 时，实际的唤醒源为：

```
blmsPm.wakeup_src | PM_WAKEUP_TIMER
```

即 PM\_WAKEUP\_TIMER 是一定会有的，不依赖于 user 的设定，这是为了保证 MCU 一定要在特定的时间点唤醒去处理接下来的 ADV task、SCAN task、central task、peripheral task。

每次调用 blc\_pm\_setWakeupSource 设置唤醒源后，一旦 MCU 进入 sleep mode 被唤醒后，blmsPm.wakeup\_src 会被清 0。

#### 4.2.6 API blc\_pm\_setDeepsleepRetentionType

前面介绍了 deepsleep retention 根据 retention sram size 的差别有分为 32K/64K/96K sram retention。当 sleep mode 中 deepsleep retention mode 生效时，SDK 会根据设置进入相应的 deepsleep retention mode。

以 B91 为例，可选的模式只有以下两种，32K 和 64K，SDK 默认的 deepsleep retention mode 为 DEEPSLEEP\_MODE\_RET\_SRAM\_LOW64K:

```
typedef enum {
    SUSPEND_MODE                  = 0x00,
    DEEPSLEEP_MODE                 = 0x30,
    DEEPSLEEP_MODE_RET_SRAM_LOW32K  = 0x21,
    DEEPSLEEP_MODE_RET_SRAM_LOW64K  = 0x03,
    DEEPSLEEP_RETENTION_FLAG       = 0x0F,
}pm_sleep_mode_e;
```



下面 API 供 user 选择 deepsleep retention mode，鉴于目前 SDK 都是默认使用最大 retention sram size，用户基本上用不到。

```
void blc_pm_setDeepsleepRetentionType(pm_sleep_mode_e sleep_type)
{
    blmsPm.deepRet_type = sleep_type;
}
```

**注意：**

该 API 的调用必须在 blc\_ll\_initPowerManagement\_module 之后才能生效。

#### 4.2.7 API blc\_pm\_setDeepsleepRetentionEnable

该 API 用于使能 deepsleep retention mode。

```
typedef enum {
    PM_DeepRetn_Disable = 0x00,
    PM_DeepRetn_Enable = 0x01,
} deep_retn_en_t;

void blc_pm_setDeepsleepRetentionEnable (deep_retn_en_t en)
{
    blmsPm.deepRt_en = en;
}
```

#### 4.2.8 API blc\_pm\_setDeepsleepRetentionThreshold

在 BLE task 存在，满足以下条件，suspend 才会被自动切换为 deepsleep retention：

```
//判断 sleep mode 是 suspend mode 还是 deepsleep retention mode
pm_sleep_mode_e sleep_M = SUSPEND_MODE;
if( blmsPm.deepRt_en && (u32)(blmsPm.current_wakeup_tick - clock_time() -
    blmsPm.deepRet_thresTick) < BIT(30) ){
    sleep_M = (pm_sleep_mode_e)blmsPm.deepRet_type;
}
```

第一个条件 blmsPm.deepRt\_en，需要调用 API blc\_pm\_setDeepsleepRetentionEnable 使能，前面已经介绍过。

第二个条件 (u32)(blmsPm.current\_wakeup\_tick - clock\_time() - blmsPm.deepRet\_thresTick) < BIT(30)，表示 sleep 的持续时间（即唤醒时间减去实时时间）超过特定的时间阀值时（即 blmsPm.deepRet\_thresTick），MCU 的 sleep mode 才会从 suspend 自动切换为 deepsleep retention。

API blc\_pm\_setDeepsleepRetentionThreshold 用于设置 suspend 切换到 deepsleep retention 触发条件中的时间阀值，这个设计是为了追求更低的功耗。



```
void blc_pm_setDeepsleepRetentionThreshold(u32 threshold_ms)
{
    blmsPm.deepRet_thresTick = threshold_ms * SYSTEM_TIMER_TICK_1MS;
}
```

## 4.2.9 PM 软件处理流程

低功耗管理的软件处理流程，下面将使用代码与伪代码相结合的方式来说明，目的是为了让 user 了解处理流程的所有逻辑细节。

### 4.2.9.1 blc\_sdk\_main\_loop

tl\_ble\_sdk 中，blc\_sdk\_main\_loop 在一个 while(1) 的结构中被反复调用。

```
while(1)
{
    ///////////////// BLE entry /////////////////////////////////
    blc_sdk_main_loop();

    ///////////////// UI entry /////////////////////////////////
    // UI task

    ///////////////// PM entry /////////////////////////////////
    app_process_power_management();
}
```

blc\_sdk\_main\_loop 函数在 while(1) 中不断被执行，BLE 低功耗管理的 code 在 blc\_sdk\_main\_loop 函数中，所以低功耗管理的 code 也是一直在被执行。

下面是 blc\_sdk\_main\_loop 函数中低功耗管理逻辑的实现。

```
void blc_sdk_main_loop (void)
{
    .....
    if( blmsPm.sleep_mask == PM_SLEEP_DISABLE )
    {
        return; // PM_SLEEP_DISABLE, can not enter sleep mode;sleep time
    }

    if( !tick1_exceed_tick2(blmsPm.next_task_tick, clock_time() + PM_MIN_SLEEP_US) )
    {
        return; //too short, can not enter sleep mode.
    }

    if( bltSche.task_mask && (blmsPm.sleep_taskMask & bltSche.task_mask) != bltSche.task_mask )
}
```



```
//是否有 task (adv、scan、central、peripheral)
//sleep_taskMask 是否允许该状态 (adv、scan、central、peripheral) 进入 sleep
{
    return;
}

if ( (brx_post | btx_post | adv_post | scan_post) == 0 )
{
    return; //只能在各个任务完成后才允许进入 sleep
}
else
{
    blt_sleep_process(); //process sleep & wakeup
}
.....
}
```

- (1) 当 blmsPm.sleep\_mask 为 PM\_SLEEP\_DISABLE 时，直接退出，不会执行 blt\_sleep\_process 函数。所以 user 使用 blc\_pm\_setSleepMask(PM\_SLEEP\_DISABLE) 时，低功耗管理的逻辑就会完全失效，MCU 不会进入 sleep，while(1) 的 loop 一直在执行。
- (2) 如果睡眠时间太短，也不会进入 sleep。
- (3) 当存在任务，比如 adv task、scan task、central task、peripheral task，但是如果相应 task 的 sleep\_taskMask 没有使能也不会进入低功耗模式。
- (4) 如果 Adv Event 或 Scan Event 或 Conn state Central role 的 Btx Event 或 Conn state Peripheral role 的 Brx Event 正在执行，blt\_sleep\_process 函数也不会被执行，这是因为此时 RF 的任务正在运行，SDK 需要保证 Adv Event/Scan Event/Btx Event/Brx Event 结束之后才能进 sleep mode。

当以上几个条件都满足时，才去执行 blt\_sleep\_process 函数。

#### 4.2.9.2 blt\_sleep\_process

blt\_sleep\_process 函数的逻辑实现如下所示。

```
void blt_sleep_process (void)
{
    .....
    blmsPm.current_wakeup_tick = blmsPm.next_task_tick;//记录唤醒时间点

    //执行 BLT_EV_FLAG_SLEEP_ENTER 回调函数
    blt_p_event_callback (BLT_EV_FLAG_SLEEP_ENTER, NULL, 0);

    //进入低功耗函数
    u32 wakeup_src = cpu_sleep_wakeup (sleep_M, PM_WAKEUP_TIMER | blmsPm.wakeup_src,
    ↵ blmsPm.current_wakeup_tick);
```



```
//执行 BLT_EV_FLAG_SUSPEND_EXIT 回调函数
blt_p_event_callback (BLT_EV_FLAG_SUSPEND_EXIT, (u8 *)&wakeup_src, 1);

blmsPm.wakeup_src = 0;
.....
}
```

上面是 blt\_sleep\_process 函数的简要流程，这里看到 2 个 sleep 相关 event 回调函数的执行的时机：BLT\_EV\_FLAG\_SLEEP\_ENTER、BLT\_EV\_FLAG\_SUSPEND\_EXIT。

关于怎么进入 sleep mode，最终调用了 driver 中的 API cpu\_sleep\_wakeup：

```
cpu_sleep_wakeup(pm_sleep_mode_e sleep_mode, SleepWakeupSrc_TypeDef wakeup_src, unsigned int
↪ wakeup_tick);
```

唤醒源为 PM\_WAKEUP\_TIMER | blmsPm.wakeup\_src，Timer 唤醒无条件生效，是为了保证 MCU 在下一个 task 到来前唤醒。

blt\_sleep\_process 函数退出时将 blmsPm.wakeup\_src 的值复位，所以需要注意 API blc\_pm\_setWakeupSource 设置唤醒源的生命周期，每次设置的值只对最近一次要进入的 sleep mode 有效。

#### 4.2.10 API blc\_pm\_getWakeupSystemTick

下面的 API 用于获取低功耗管理计算的 sleep 醒来的时间点（System Timer tick），即 T\_wakeup。

```
u32 blc_pm_getWakeupSystemTick (void);
```

T\_wakeup 的计算是在接近 cpu\_sleep\_wakeup 函数处理前，应用层只能在 BLT\_EV\_FLAG\_SLEEP\_ENTER 事件回调函数里才能得到准确的 T\_wakeup。

假设用户在 sleep 时间比较长的情况下，需要按键唤醒。下面我们说明一下设置方法。

我们需要使用 BLT\_EV\_FLAG\_SLEEP\_ENTER 事件回调函数和 blc\_pm\_getWakeupSystemTick。

BLT\_EV\_FLAG\_SLEEP\_ENTER 的回调注册方法如下：

```
blc_ll_registerTelinkControllerEventCallback (BLT_EV_FLAG_SLEEP_ENTER, &app_set_kb_wakeup);
_attribute_ram_code_ void app_set_kb_wakeup (u8 e, u8 *p, int n)
{
    /* sleep time > 100ms. add GPIO wake_up */
    if(((u32)(blc_pm_getWakeupSystemTick() - clock_time())) > 100 * SYSTEM_TIMER_TICK_1MS){
        blc_pm_setWakeupSource(PM_WAKEUP_PAD); //GPIO PAD wake_up
    }
}
```

以上举例，如果 sleep 时间超过 100ms，就添加 GPIO 唤醒。user 可以根据实际情况来调整。

这里只是提供了一个接口，客户根据实际情况来决定是否使用。



#### 4.2.11 API bIc\_pm\_setDeepsleepRetentionEarlyWakeupTiming

该 API 用于设置 Deep Retention 模式下的提前唤醒时间。

芯片在 Deep Retention 模式下唤醒后，从启动文件的 `_IRESET_ENTRY` 入口开始启动，执行必要的初始化代码，从 `_IRESET_ENTRY` 到 `user_init_deepRetn` 函数中的开启中断 (`irq_enable`) 操作，这部分代码消耗的时间称为 Deep Retention 模式的唤醒时间。

唤醒时间会影响 BLE 的时序，且会受到以下因素影响：

1. 用户在开启中断操作前添加的代码；
2. 必要的初始化代码；
3. 主频。

设置提前唤醒时间是为了补偿唤醒时间，通常等于唤醒时间加上 50us 的阈值，唤醒时间的测量主要借助逻辑分析仪来进行测量，可以参考以下步骤：

1. 打开 `_ISTART` 中的 DEBUG IO，该 IO 置高电平时，为 `_IRESET_ENTRY` 的执行时间点；



```
303 .global _IRESET_ENTRY
304 .type _IRESET_ENTRY,function
305
306 .align 2
307 _IRESET_ENTRY:
308 /* Decide whether this is an NMI or cold reset */
309 j _ISTART
310
311 /* BLE SDK use: change from 0x22 to 0x24 */
312 .org 0x24
313
314 _ISTART:
315 #if 1
316 // add debug, PB4 output 1
317 lui t0, 0x80140
318 addi t0, t0, 0x790
319 li t1, 0xef
320 li t2, 0x10
321 sb t1, 0x482(t0)      //0x80140c12  PB oen    = 0xef
322 sb t2, 0x484(t0)      //0x80140c14  PB output = 0x10
323#endif
324
325 /* timer rst enable. */
326 li t0, 0x80140822 /* reg_rst2 register address. */
327 li t1, 0x39        /* bit0=1 timer enable(reset value 0x38). */
328 sb t1, 0x0(t0)     /* (*(volatile unsigned char*)(0x80140822))= 0x39. */
329
330 /* timer clk enable. */
331 li t0, 0x80140826 /* tim_ctrl2 register address. */
332 li t1, 0x31        /* bit0=1 timer enable(reset value 0x30). */
333 sb t1, 0x0(t0)     /* (*(volatile unsigned char*)(0x80140826))= 0x31. */
334
335 /* set timer watchdog value: 10s. */
336 li t0, 0x8014014c /* wt_target register address. */
```

Figure 4.6: image-20250626102143943

2. 打开 `app_config.h` 中的 `DEBUG_GPIO_ENABLE` 开关，这会打开开启中断（`irq_enable`）操作前的 DEBUG IO，该 IO 置高电平时，为开启中断（`irq_enable`）操作的执行时间点；



```
app.c      cststartup_TL321X.S    default_config.h    app_config.h x
80 #elif (MCU_CORE_TYPE == MCU_CORE_TL322X)
81     #define BOARD_SELECT BOARD_322X_EVK_C1T382A20
82 #endif
83
84 ////////////////////////////////////////////////////////////////// UI Configuration //////////////////////////////////////////////////////////////////
85 #define UI_LED_ENABLE      1
86 #define UI_KEYBOARD_ENABLE 1
87 #define UI_BUTTON_ENABLE   0
88
89 ////////////////////////////////////////////////////////////////// DEBUG Configuration //////////////////////////////////////////////////////////////////
90 #define DEBUG_GPIO_ENABLE   1
91
92 #define TLKAPI_DEBUG_ENABLE 1
93 #define TLKAPI_DEBUG_CHANNEL TLKAPI_DEBUG_CHANNEL_GUART
94
95 #define APP_LOG_EN          1
96 #define APP_CONTR_EVT_LOG_EN 1 //controller event|
97 #define APP_HOST_EVT_LOG_EN 1
98 #define APP_KEY_LOG_EN      1
99 #define APP_BUTTON_LOG_EN   1
100
101 #define JTAG_DEBUG_DISABLE  1 //if use JTAG, change this
102
103
104
105 ////////////////////////////////////////////////////////////////// DEEP SAVE FLG //////////////////////////////////////////////////////////////////
106 #define USED_DEEP_ANA_REG PM_ANA_REG_POWER_ON_CLR_BUF1 //u8,can save 8 bit info when deep
107 #define LOW_BATT_FLG      BIT(0)                      //if 1: low battery
108 #define CONN_DEEP_FLG     BIT(1)                      //if 1: conn deep, 0: adv deep
109
110
111 #if FREERTOS_ENABLE
112 ////////////////////////////////////////////////////////////////// PRINT DEBUG INFO //////////////////////////////////////////////////////////////////
```

Figure 4.7: image-20250626102827810



```
app.c x cstartup_TL321X.S default_config.h app_config.h
170 #if (PM_DEEPSLEEP_RETENTION_ENABLE)
171     blc_app_loadCustomizedParameters_deepRetn();
172
173 #if (MCU_CORE_TYPE == MCU_CORE_TL321X) //TL321x Flash protection
174     //blc_ll_initBasicMCU() and blc_ll_recoverDeepRetention() configure stack IRQ.
175     //So we need to disable IRQ first in case the stack IRQ is triggered unexpectedly.
176     irq_disable();
177 #endif
178
179     blc_ll_initBasicMCU(); //mandatory
180
181     blc_ll_recoverDeepRetention();
182
183     DBG_CHNO_HIGH;
184     irq_enable();
185
186 #if (UI_KEYBOARD_ENABLE)
187     ////////// keyboard GPIO wakeup init //////////
188     u32 pin[] = KB_DRIVE_PINS;
189     for (unsigned int i = 0; i < (sizeof(pin) / sizeof(*pin)); i++) {
190         cpu_set_gpio_wakeup(pin[i], WAKEUP_LEVEL_HIGH, 1); //drive pin pad high level wakeup deepsleep
191     }
192 #endif
193
194 #if (BATT_CHECK_ENABLE)
195     adc_hw_initialized = 0;
196 #endif
197
198 #if (TLKAPI_DEBUG_ENABLE)
199     tlkapi_debug_deepRetn_init();
200 #endif
201
202 }
203
204 void app_process_power_management(void)
```

Figure 4.8: image-20250626102915243

3. 使用逻辑分析仪连接以上两个 IO，并设置芯片进入 Deep Retention 模式，可以在逻辑分析仪的页面上观察到唤醒时间。

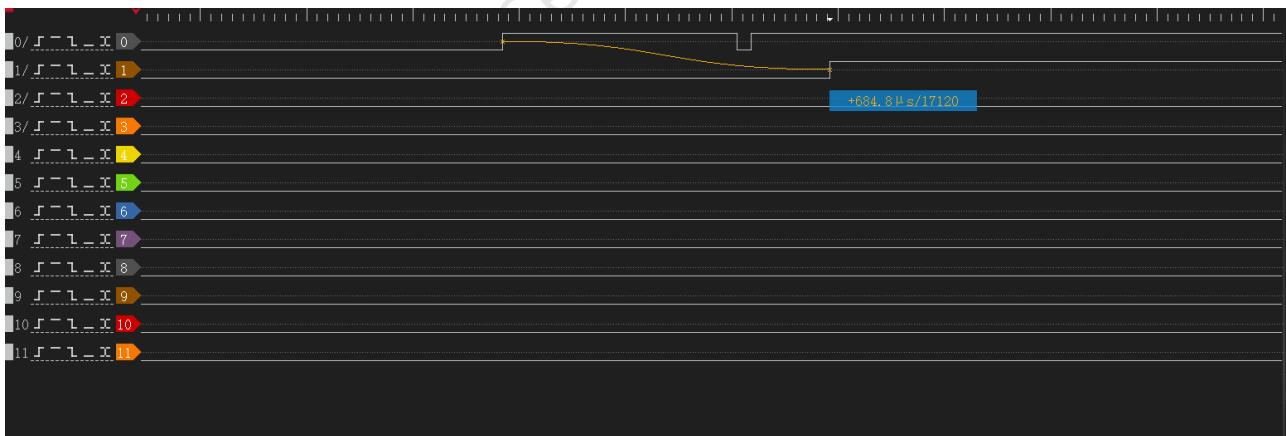


Figure 4.9: image-20250626104331999

将唤醒时间加上 50us 的阈值作为参数，通过 blc\_pm\_setDeepsleepRetentionEarlyWakeupTiming API 来进行设置。



## 4.3 GPIO 唤醒的注意事项

### 4.3.1 唤醒电平有效时无法进入 sleep mode

由于 Telink MCU 的 GPIO 唤醒是靠高低电平唤醒，而不是上升沿下降沿唤醒，所以当配置了 GPIO PAD 唤醒时，比如设置了某个 GPIO PAD 高电平唤醒 suspend，要确保 MCU 在调用 `cpu_sleep_wakeup` 进入 suspend 时，当前的这个 GPIO 读到的电平不能是高电平。若当前已经是高电平了，实际进入 `cpu_sleep_wakeup` 函数里面，触发 suspend 时是无效的，会立刻退出来，即完全没有进入 suspend。

如果出现以上情况，可能会造成意想不到的问题，比如本来想进入 deepsleep 后被唤醒，程序重新执行，结果 MCU 无法进入 deepsleep，导致 code 继续运行，不是我们预想的状态，整个程序的 flow 可能会乱掉。

user 在使用 Telink 的 GPIO PAD 唤醒时，要注意避免这个问题。

如果应用层没有很好的规避这个问题，在调用 `cpu_sleep_wakeup` 函数时发生了 GPIO PAD 唤醒源已经生效的情况，为了防止程序进入不可预知的逻辑，PM driver 做了一些改善：

- (1) suspend & deepsleep retention mode

如果是 suspend 和 deepsleep retention mode，都会很快退出函数 `cpu_sleep_wakeup`，该函数给出的返回值可能出现两种情况：

- PM 模块上检测到了 GPIO PAD 生效的状态，返回 `WAKEUP_STATUS_PAD`；
- PM 模块上没有检测到 GPIO PAD 生效的状态，返回 `STATUS_GPIO_ERR_NO_ENTER_PM`

- (2) deepsleep mode

如果是 deepsleep mode，PM driver 会在底层自动将 MCU reset（此时的 reset 跟 watchdog reset 效果一致），程序回到“Run hardware bootloader”开始重新运行。

## 4.4 应用层定时唤醒

在 BLE task 存在且不考虑 GPIO PAD 唤醒的前提下，一旦进入 sleep mode，只能在 SDK 计算好的时间点 `T_wakeup` 唤醒，user 无法在某一个特定的时间点将 sleep 提前唤醒。为了增加 PM 的灵活性，SDK 增加了应用层定时唤醒的 API 和它的回调函数。

应用层定时唤醒 API：

```
void blc_pm_setAppWakeupLowPower(u32 wakeup_tick, u8 enable);
```

wakeup\_tick 为定时唤醒的 System Timer tick 值；

enable 为 1 时打开该唤醒功能，enable 为 0 时关闭。

应用层定时唤醒发生时，执行 `blc_pm_registerAppWakeupLowPowerCb` 注册的回调函数，其原型和 API 如下：

```
typedef void (*pm_appWakeupLowPower_callback_t)(int);
pm_appWakeupLowPower_callback_t pm_appWakeupLowPowerCb = NULL;
void blc_pm_registerAppWakeupLowPowerCb(pm_appWakeupLowPower_callback_t cb)
```



```
{  
    pm_appWakeupLowPowerCb = cb;  
}
```

以 ACL Conn state Peripheral role 为例：

当 user 使用 blc\_pm\_setAppWakeupLowPower 设置了应用层定时唤醒的 app\_wakeup\_tick，SDK 在进入 sleep 前，会检查 app\_wakeup\_tick 是否在 T\_wakeup 之前。

- (1) 如果 app\_wakeup\_tick 在 T\_wakeup 之前，如下图所示，就会在 app\_wakeup\_tick 触发 sleep 提前唤醒；
- (2) 如果 app\_wakeup\_tick 在 T\_wakeup 之后，MCU 还是会在 T\_wakeup 唤醒。

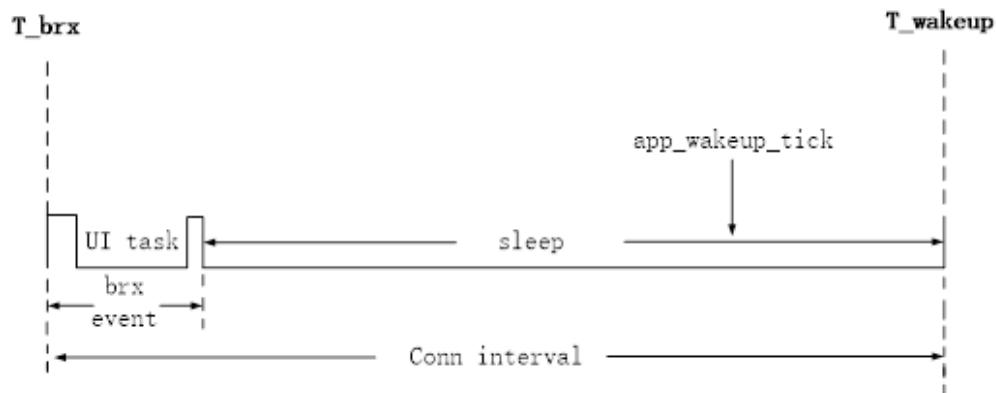


Figure 4.10: Early wake\_up at app\_wakeup\_tick



## 5 低电检测

电池电量检测 (battery power detect/check)，在 Telink BLE SDK 和相关文档中也可能出现其他的名字，包括：电池电量检测 (battery power detect/check)、低电池检测 (low battery detect/check)、低电量检测 (low power detect/check)、电池检查 (battery detect/check) 等。比如 SDK 中相关文件和函数出现 battery\_check、battery\_detect、battery\_power\_check 等命名。

本文档统一以“低电检测 (low battery detect)”这个名称进行说明。

### 5.1 低电检测的重要性

使用电池供电的产品，由于电池电量会逐渐下降，当电压低到一定的值后会引起很多问题：

- a) tl\_ble\_sdk 工作电压的范围为 1.8V~4.3V。当电压低于 1.8V tl\_ble\_sdk 已经无法保证稳定地工作。
- b) 当电池电压较低时，由于电源的不稳定，Flash 的“write”和“erase”操作可能有出错的风险，造成 program firmware 和用户数据被异常修改，最终导致产品失效。根据以往的量产经验，我们将这个可能出风险的低压阈值设定为 2.0V。

根据上面的描述，使用电池供电的产品，必须设定一个安全电压值 (secure voltage)，只有当电压高于这个安全电压的时候才允许 MCU 继续工作；一旦电压低于安全电压，MCU 停止运行，需要立刻被 shutdown (SDK 上使用进入 deepsleep mode 来实现)。

安全电压也称为报警电压，这个电压值的选取，目前 SDK 默认使用 2.0V。如果 user 在硬件电路中出现了不合理的设计，导致电源网络稳定性的恶化，安全电压值还需要继续提高，比如 2.1V、2.2V 等。

对于 Telink BLE SDK 开发实现的产品，只要使用了电池供电，低电检测都必须是该产品整个生命周期实时运行的任务，以保证产品的稳定性。

### 5.2 低电检测的实现

低电检测需要使用 ADC 对电源电压进行测量。user 请参考文档 Data sheet 和 Driver SDK Developer Handbook 相关 ADC 章节，先对 tl\_ble\_sdk 的 ADC 模块进行必要的了解。

低电检测的实现，结合 SDK demo “acl\_central\_demo”给出的实现来说明，参考文件 battery\_check.h 和 battery\_check.c。

必须确保 app\_config.h 文件中宏“BATT\_CHECK\_ENABLE”是被打开的，user 使用低电检测功能时需要注意。

```
#define BATT_CHECK_ENABLE
```

```
1
```

#### 5.2.1 低电检测的注意事项

低电检测是一个基本的 ADC 采样任务，在实现 ADC 采样电源电压时，有一些需要注意的问题，说明如下：



### 5.2.1.1 建议使用 GPIO 输入通道

tl\_ble\_sdk 的采样方式可采用 Vbat 或 GPIO 模拟信号输入的方式进行采样，但 Vbat 通道采样精度较差，对采样精度要求高的场合建议通过外部 GPIO 方式采样。

可用的 GPIO 输入通道为 PB0~PB7、PD0、PD1 对应的 input channel。

```
typedef enum{
    ADC_GPIO_PB0 = GPIO_PB0 | (0x1<<12),
    ADC_GPIO_PB1 = GPIO_PB1 | (0x2<<12),
    ADC_GPIO_PB2 = GPIO_PB2 | (0x3<<12),
    ADC_GPIO_PB3 = GPIO_PB3 | (0x4<<12),
    ADC_GPIO_PB4 = GPIO_PB4 | (0x5<<12),
    ADC_GPIO_PB5 = GPIO_PB5 | (0x6<<12),
    ADC_GPIO_PB6 = GPIO_PB6 | (0x7<<12),
    ADC_GPIO_PB7 = GPIO_PB7 | (0x8<<12),
    ADC_GPIO_PD0 = GPIO_PD0 | (0x9<<12),
    ADC_GPIO_PD1 = GPIO_PD1 | (0xa<<12),
}adc_input_pin_def_e;
```

使用 GPIO input channel 对电源电压进行 ADC 采样，其具体使用方式如下：在硬件电路设计上，将电源直接和 GPIO input channel 连接。ADC 初始化时，将 GPIO 设为高阻态 (ie、oe、output 全部设 0)，此时 GPIO 上的电压等于电源电压，直接进行 ADC 采样即可。

User 可通过“acl\_central\_demo”的 app\_config.h 中的宏切换 GPIO input channel：

```
//若使用 GPIO input channel, 将该值置为 0
#define VBAT_CHANNEL_EN 1
```

demo 中默认选择 PB1 为 GPIO input channel，PB1 作为普通 GPIO 功能，初始化时所有状态 (ie、oe、output) 使用默认状态即可，不做特殊修改，用户若想切换 GPIO，可选择上述定义的 GPIO 输入通道修改下面定义即可。

```
#define GPIO_BAT_DETECT          GPIO_PB1
#define PB1_FUNC                  AS_GPIO
#define PB1_INPUT_ENABLE          0
#define PB1_DATA_OUT              0
#define ADC_INPUT_PIN_CHN         ADC_GPIO_PB1
```

### 5.2.1.2 只能使用差分模式

虽然 tl\_ble\_sdk ADC input mode 同时支持单端模式 (Single Ended Mode) 和差分模式 (Differential Mode)，但由于某些特定的原因，Telink 规定：只能使用差分模式，单端模式不允许使用。

差分模式的 input channel 分为 positive input channel (正端输入通道) 和 negative input channel (负端输入通道)，被测量的电压值为 positive input channel 电压减去 negative input channel 电压得到的电压差。



如果 ADC 采样的 input channel 只有 1 个，使用差分模式时，将当前 input channel 设置为 positive input channel，将 GND 设为 negative input channel。这样二者的电压差和 positive input channel 电压相等。

SDK 中低压检测使用了差分模式，函数接口如下：

```
adc_set_diff_input(ADC_INPUT_PIN_CHN >> 12, GND);
```

### 5.2.1.3 不同的 ADC 任务需要切换

低压检测无法与其他 ADC 任务同时运行，必须采用切换的方式来实现。

## 5.2.2 低电检测的单独使用

在 SDK demo 中，“acl\_central\_demo”、“acl\_connection\_demo”、“acl\_peripheral\_demo”工程中实现了低电检测功能，user 需要在 app\_config.h 中开启低电检测的功能进行使用。用户若在其他 feature\_demo 中使用低电检测功能，可参考已经实现的 demo 中 app\_config.h 的定义和 app.c 中低电检测相关接口调用逻辑。

### 5.2.2.1 低电检测初始化

参考 adc\_bat\_detect\_init 函数的实现。

ADC 初始化的顺序必须满足下面的流程：先 power off (掉电) sar adc，然后配置其他参数，最后 power on (上电) sar adc。所有 ADC 采样的初始化都必须遵循这个流程。

```
_attribute_ram_code_ void adc_bat_detect_init(void)
{
    adc_power_off();           // power off sar adc
    ....                      // add ADC Configuration
    adc_power_on();            // power on sar adc
}
```

Sar adc power on 与 power off 之间的配置，user 尽量不要去修改，使用这些默认的设置就行。User 如果选择了不同的 GPIO input channel，直接修改前面讲述的 app\_config.h 有关宏的定义即可。

adc\_bat\_detect\_init 初始化函数在 app\_battery\_power\_check 中调用的 code 为：

```
if(!adc_hw_initialized){
    adc_hw_initialized = 1;
    adc_bat_detect_init();
}
```

这里使用了一个变量 adc\_hw\_initialized，只有该变量为 0 时调用一次初始化，并将其置 1；该变量为 1 时不再初始化。adc\_hw\_initialized 在下面 API 中也会被操作。



```
void battery_set_detect_enable (int en)
{
    lowBattDet_enable = en;
    if(!en){
        adc_hw_initialized = 0; //need initialized again
    }
}
```

使用了 adc\_hw\_initialized 的设计可以实现的功能有：

- a) 与其他 ADC 任务 (“ADC other task”) 的切换

先不考虑 sleep mode (suspend/deepsleep retention) 的影响，只分析低电检测与其他 ADC 任务的切换。

因为需要考虑低电检测与其他 ADC 任务的切换使用，可能需要 adc\_bat\_detect\_init 被多次执行，所以不能写到 user initialization 中，必须在 main\_loop 里实现。

第一次执行 app\_battery\_power\_check 函数时，adc\_bat\_detect\_init 被执行，且后面不会被反复执行。

一旦“ADC other task”需要执行时，将抢走 ADC 的使用权，确保“ADC other task”初始化时必须调用 battery\_set\_detect\_enable(0)，此时会将 adc\_hw\_initialized 清 0。

等“ADC other task”完成后，交出 ADC 的使用权。app\_battery\_power\_check 再次执行，由于 adc\_hw\_initialized 值为 0，必须再次执行 adc\_bat\_detect\_init，这样就保证了低电检测每次切回来时都会重新初始化。

- b) 对 suspend 和 deepsleep retention 的自适应处理

将 sleep mode 考虑进来。

adc\_hw\_initialized 这个变量使用必须定义成一个“data”段或“bss”段上的变量，不能定义到 retention\_data 上。定义在“data”段或“bss”上可以保证每次 deepsleep retention wake\_up 后在执行 software bootloader (即 cstartup\_xxx.S) 时这个变量会被重新初始化为 0；而 suspend wake\_up 后这个变量可以保持不变。

adc\_bat\_detect\_init 函数里面配置的 register 的共同特征是：在 suspend mode 下不掉电，可以保存状态；在 deepsleep retention mode 下会掉电。

如果 MCU 进入 suspend mode，醒来后再次执行 app\_battery\_power\_check 时，adc\_hw\_initialized 的值和 suspend 之前一致，不需要重新执行 adc\_bat\_detect\_init 函数。

如果 MCU 进入 deepsleep retention mode，醒来后 adc\_hw\_initialized 为 0，必须重新执行 adc\_bat\_detect\_init，ADC 相关的 register 状态需要被重新配置。

adc\_bat\_detect\_init 函数中设定 register 的状态可以在 suspend 期间保持不掉电。

SDK 中对 adc\_bat\_detect\_init 函数添加了关键字“\_attribute\_ram\_code\_”以设置为 ram\_code，最终目的是为了优化长睡眠连接态的功耗。比如对典型的  $10\text{ms} * (99+1) = 1\text{s}$  的长睡眠连接，每 1s 醒来一次，中间的长睡眠使用的是 deepsleep retention mode，那么每次醒来后 adc\_bat\_detect\_init 一定会重新执行一次，加入到 ram\_code 后执行速度会变得更快。

这个“\_attribute\_ram\_code\_”不是必须的。在产品应用中，user 可以根据 deepsleep retention area 的使用情况，结合功耗测试的结果，来决定是否将此函数放入到 ram\_code 中。



### 5.2.2.2 低电检测处理

在 main\_loop 中，调用 user\_battery\_power\_check 函数实现低电检测的处理，相关 code 如下：

```
#if (BATT_CHECK_ENABLE)
    /*The frequency of low battery detect is controlled by the variable lowBattDet_tick, which
     * is executed every
     * 500ms in the demo. Users can modify this time according to their needs.*/
    if(battery_get_detect_enable() && clock_time_exceed(lowBattDet_tick, 500000) ){
        lowBattDet_tick = clock_time();
        user_battery_power_check(BAT_DEEP_THRESHOLD_MV);
    }
#endif
```

battery\_get\_detect\_enable() 返回 lowBattDet\_enable 值，底层低电检测是默认使能，即该值默认为 1，如果用户层面对 BATT\_CHECK\_ENABLE 进行置位，MCU 上电后立刻可以开始低电检测。该变量需要设置成 retention\_data，确保 deepsleep retention 不能修改它的状态。

只有在其他 ADC 任务需要抢占 ADC 使用权时，才能通过调用 battery\_set\_detect\_enable 改变 lowBattDet\_enable 的值：当其他 ADC 任务开始时，调用 battery\_set\_detect\_enable(0)，此时 main\_loop 中不会再调用 user\_battery\_power\_check 函数；在其他 ADC 任务结束后，调用 battery\_set\_detect\_enable(1)，交出 ADC 使用权，此时 main\_loop 中又可以调用 user\_battery\_power\_check 函数。

变量 lowBattDet\_tick 来控制低电检测的频率，Demo 中为每 500ms 执行一次低电检测。User 可以根据自己的需求来修改这个时间值。

user\_battery\_power\_check 函数被放到 ram\_code 上，参考上面对“adc\_bat\_detect\_init”放在 ram\_code 的说明，也是为了节省运行时间，优化功耗。

这个“attribute\_ram\_code”不是必须的。在产品应用中，user 可以根据 deepsleep retention area 的使用情况，结合功耗测试的结果，来决定是否将此函数放入到 ram\_code 中。

```
_attribute_ram_code_ void user_battery_power_check(u16 alarm_vol_mv);
```

### 5.2.2.3 低压报警

user\_battery\_power\_check 的参数为阈值电压，单位为 mV。根据前文介绍，SDK 中默认设置 deepsleep 为 2000mV，在 main\_loop 的低压检测中，当电源电压低于 2000mV 时，进入 deepsleep 模式；当电源电压高于 2200mV 时才会执行唤醒。

tl\_ble\_sdk demo 中使用进入 deepsleep 的方式来实现 shutdown MCU，进入休眠前设置有 LED 闪烁提示，并且设置了按键可以唤醒。

程序被 shutdown 后，进入可被唤醒的 deepsleep mode。此时如果发生按键唤醒，SDK 会在 user initialization(user\_init\_normal()) 的时候先快速做一次低电检测，而不是等到 main\_loop 中检测。这样处理的原因是为了避免应用上的错误，举例说明如下：

在 deepsleep 状态下被按键触发唤醒时（此时工作电压仍然小于唤醒电压），从 main\_loop 的处理来看，需要至少 500ms 的时间才会去做低电检测，即芯片异常工作 500ms。



因为这个原因，SDK 必须在 user initialization 的时候就提前做低电检测，必须在这一步就阻止发生上面的情况。所以在 user initialization 的时候，添加低电检测，SDK 中函数接口为：

```
#if (BATT_CHECK_ENABLE)
    user_battery_power_check(2000);
#endif
```

在 user\_battery\_power\_check 函数中如果是从 deepsleep 状态中唤醒，采用低电检测的阈值电压为参数 alarm\_vol\_mv+200mV，其原因主要是：在 shutdown 模式唤醒后的快速低电检测时，将报警电压稍微调高一些，调高的幅度比低电检测的最大误差稍大，因此需要对唤醒时检测的电压作出提高的设定。一般来说，只有当某次低电检测发现电压低于 2000mV 进入 shutdown 模式后，才会出现恢复电压 2200mV，所以 user 不用担心这个 2200mV 会对实际电压 2V~2.2V 的产品误报低压。

### 5.2.3 低电检测和 Amic Audio

参考低电检测单独使用模式中详细的介绍，对于需要实现 Amic Audio 的产品，只要做好低电检测和 Amic Audio 的切换即可。

按照低电检测单独使用的方式，程序开始运行后，默认低电检测先开启。当 Amic Audio 被触发时，做以下两件事：

- (1) 关闭低电检测

调用 battery\_set\_detect\_enable(0)，告知低电检测模块 ADC 资源已被抢占。

- (2) Amic Audio ADC 初始化

由于使用 ADC 的方式和低电检测不一样，需要对 ADC 重新进行初始化。具体方法参考本文档“Audio”章节的介绍。

Amic Audio 结束时，调用 battery\_set\_detect\_enable(1)，告知低电检测模块 ADC 资源已经被释放。此时低电检测需要重新初始化 ADC 模块，然后开始进行低电检测。

如果是低电检测和其他非 Amic Audio 的 ADC 任务同时存在，其他 ADC 任务的处理可模仿 Amic Audio 的处理流程。

如果是低电检测、Amic Audio、其他 ADC 任务共 3 种任务同时存在，user 可根据“ADC 电路需要切换使用”的原则，参考低电检测和 Amic Audio 切换实现的方法，去自行实现。



## 6 Flash 写保护

Flash 写保护主要是用于保护 Flash 中的用户代码、用户数据和用户配置信息被非授权方读取或篡改。

### 6.1 Flash 写保护的重要性

当电压较低或电源的不稳定，对 Flash 进行操作可能存在出错的风险（尤其是“write”和“erase”操作），造成 firmware 和用户数据被异常修改，最终导致产品失效。根据以往的量产经验，推荐客户将 Flash 写保护默认开启。客户在使用 Flash 写保护时，必须设定一个安全保护区域大小，一方面不能影响正常的协议栈存储信息的写入和擦除（如 SMP 信息存储区域），另一方面需要尽可能的保护到程序和用户数据。

对于 Telink BLE SDK 开发实现的产品，只要使用了电池供电，Flash 写保护最好是该产品整个生命周期实时运行的任务，以保证产品的稳定性和用户数据的安全性。

### 6.2 Flash 写保护的实现

Flash 写保护需要在电源电压大于安全电压阈值。因此使用 Flash 写保护功能时也需要将低压检测功能打开。

Flash 写保护的实现，结合 tl\_ble\_sdk/acl\_central\_demo 给出的实现来说明，参考文件 flash\_prot.h 和 flash\_prot.c。

必须确保 app\_config.h 文件中宏“APP\_FLASH\_PROTECTION\_ENABLE”是被打开的，user 使用 Flash 写保护功能时需要注意。

```
#define APP_FLASH_PROTECTION_ENABLE
```

1

#### 6.2.1 Flash 写保护的使用

在 SDK demo 中，tl\_ble\_sdk 实现了 Flash 写保护功能，user 需要在 app\_config.h 中开启 Flash 写保护功能进行使用。

##### 6.2.1.1 Flash 写保护的初始化

初始化调用 app\_flash\_protection\_operation 函数，第一个参数传入 FLASH\_OP\_EVT\_APP\_INITIALIZATION 事件表示初始化 Flash 写保护，初始化不会使用后面两个参数传入 0 即可，下面是 app\_flash\_protection\_operation 函数中初始化部分。

在初始化 Flash 写保护过程中，样例代码默认对 OTA 过程需要的区域进行加锁（针对 B92，为 000000h-07FFFFh 512K 的空间）

```
void app_flash_protection_operation(u8 flash_op_evt, u32 op_addr_begin, u32 op_addr_end)
{
    if(flash_op_evt == FLASH_OP_EVT_APP_INITIALIZATION) //判断是否为初始化事件
    {
```



```
////读取 Flash 的 mid 信息，根据 mid 信息判断是哪一款 Flash 并调用对应 Flash 加锁的 API  
flash_protection_init();  
  
#if (BLE_OTA_SERVER_ENABLE)  
    //如果 ota 功能使能，根据 ota 启动地址确定 Flash 加锁区域  
    .....  
#else  
    //用户可根据实际情况设置 Flash 上锁区域大小  
    .....  
#endif  
  
//根据上层需要 Flash 加锁区域大小，映射到不同 Flash 型号上的命令，并对 Flash 进行加锁  
.....  
}  
  
.....  
}
```

注意：如果 Flash 容量为 1MB，不能将 1MB 区域都上锁，应保留一些 Flash 区域留给系统数据和用户数据使用。

demo 中初始化时会将 app\_flash\_protection\_operation 函数注册为协议栈里的回调函数，会在 OTA 过程触发不同事件进行处理，主要是清除旧固件以及写入新固件。

```
blc_appRegisterStackFlashOperationCallback(app_flash_protection_operation);
```

### 6.2.1.2 Flash 写保护处理

在 Telink BLE SDK 中 Flash 写保护的处理都集中在 app\_flash\_protection\_operation 函数中，包括应用程序中的所有 Flash 加锁和解锁的情况。如果用户有更多的 flash 操作建议在此函数中进行添加。下面对 app\_flash\_protection\_operation 函数进行介绍：

```
void app_flash_protection_operation(u8 flash_op_evt, u32 op_addr_begin, u32 op_addr_end)  
{  
    .....  
  
    else if(flash_op_evt == FLASH_OP_EVT_STACK_OTA_CLEAR_OLD_FW_BEGIN)  
    {  
        //OTA 清除旧固件开始事件  
        .....  
    }  
    else if(flash_op_evt == FLASH_OP_EVT_STACK_OTA_CLEAR_OLD_FW_END)  
    {  
        //OTA 清除旧固件结束事件  
        .....  
    }  
}
```



```
else if(flash_op_evt == FLASH_OP_EVT_STACK_OTA_WRITE_NEW_FW_BEGIN)
{
    //OTA 写入新固件开始事件
    .....
}

else if(flash_op_evt == FLASH_OP_EVT_STACK_OTA_WRITE_NEW_FW_END)
{
    //OTA 写入新固件结束事件
    .....
}

//如果需要用户应用程序需要添加更多的 Flash 保护操作可以在后面继续添加。
}
```

参数 op\_addr\_begin 表示 Flash 保护区域的起始地址。

参数 op\_addr\_end 表示 Flash 保护区域的结束地址。

参数 flash\_op\_evt 表示 Flash 写保护的事件，包括应用层动作和协议栈事件（OTA 写和擦除），共包含 5 个事件：

#define FLASH_OP_EVT_APP_INITIALIZATION	1
#define FLASH_OP_EVT_STACK_OTA_CLEAR_OLD_FW_BEGIN	10
#define FLASH_OP_EVT_STACK_OTA_CLEAR_OLD_FW_END	11
#define FLASH_OP_EVT_STACK_OTA_WRITE_NEW_FW_BEGIN	12
#define FLASH_OP_EVT_STACK_OTA_WRITE_NEW_FW_END	13

- (1) FLASH\_OP\_EVT\_APP\_INITIALIZATION 为 Flash 写保护初始化事件，对 Flash 进行加锁。
- (2) FLASH\_OP\_EVT\_STACK\_OTA\_CLEAR\_OLD\_FW\_BEGIN 为 OTA 清除旧固件开始事件，需要将 Flash 进行解锁。OTA 清除旧固件开始事件由协议栈底层触发，在“blc\_ota\_initOtaServer\_module”中，在 OTA 中成功后的芯片重启阶段。软件会将擦除整个旧固件，以备下一个新 OTA 流程，如果从“op\_addr\_begin”到“op\_addr\_end”内存地址的任何部分处于锁定区域，则需要解锁 Flash。在 demo 示例代码中，我们为新旧固件保护整个 Flash 区域，所以这里我们不需要判断“op\_addr\_begin”和“op\_addr\_end”，并且必须要进行 Flash 解锁。
- (3) FLASH\_OP\_EVT\_STACK\_OTA\_CLEAR\_OLD\_FW\_END 为 OTA 清除旧固件结束事件，需要将 Flash 进行加锁。OTA 清除旧固件结束事件由堆栈触发，在“blc\_OTA\_initOtaServer\_module”中，擦除旧固件数据完成。在结束事件中忽略“op\_addr\_begin”和“op\_addr\_end”，在 demo 示例代码中，我们需要再次锁定 Flash，因为我们已经在清除旧固件的开始事件时解锁了它。
- (4) FLASH\_OP\_EVT\_STACK\_OTA\_WRITE\_NEW\_FW\_BEGIN 为 OTA 写入新固件开始事件，需要将 Flash 进行解锁。当接收到第一个 OTA 数据 PDU 时，OTA 写入新固件开始事件由堆栈触发，软件将数据写入存放新固件的 Flash 区域，如果从“op\_addr\_begin”到“op\_addr\_end”的 Flash 地址的任何部分处于锁定区域，则需要进行 Flash 解锁。在 demo 示例代码中，我们为新旧固件整个 Flash 区域进行了保护，所以这里我们不需要判断“op\_addr\_begin”和“op\_addr\_end”，直接进行 Flash 解锁。



(5) FLASH\_OP\_EVT\_STACK\_OTA\_WRITE\_NEW\_FW\_END 为 OTA 写入新固件结束事件，需要将 Flash 进行加锁。OTA 写入新固件结束事件由堆栈触发，在 OTA 结束、发生 OTA 错误后或写入新固件数据完成。在结束事件中忽略“op\_addr\_begin”和“op\_addr\_end”，在 demo 示例代码中，我们需要再次锁定 Flash，因为我们在写入新固件的开始事件时已经解锁了它。

在 SDK demo 中针对 Flash 写保护部分只涉及 ota 过程中新旧程序的写入和擦除，用户后续可根据应用情况自行添加其他事件进行 Flash 写保护操作。

目前代码逻辑仅针对固定地址的空间进行写保护，op\_addr\_begin 和 op\_addr\_end 参数目前无效。后续为方便客户使用，会添加针对该参数的使用逻辑。

### 6.2.1.3 加锁和解锁操作

```
//加锁函数, flash_lock_cmd 控制加锁区域, 以 B92 为例, 完整定义参考 flash_mid156085.h 中
↪ mid156085_lock_block_e
void flash_lock(unsigned int flash_lock_cmd)

//解锁函数
void flash_unlock(void)
```

flash 写保护仅支持一段特定、连续区域的保护，加锁函数过程中如果新加锁区域和旧加锁区域不同，首先执行旧加锁区域的解锁操作，再对新加锁区域进行加锁。

用户在使用过程中，可以直接调用加锁和解锁函数进行操作，但是为方便代码管理，建议仅调用 app\_flash\_protection\_operation 进行操作：

- 1) 在 flash\_port.h 中自定义操作码
- 2) app\_flash\_protection\_operation 函数中结合操作码判断编写相应的操作逻辑。



## 7 OTA

tl\_ble\_sdk 支持设备作为 OTA Server 或 OTA Client 进行升级，默认 Peripheral 作为 OTA Server，Central 作为 OTA Client。同一时刻只允许在一个 Peripheral 链路上进行 OTA。

OTA Client 的参考代码可参考[feature\\_ota](#)。OTA Server 的实现代码在协议栈中，不予开放。

tl\_ble\_sdk 支持 Flash 多地址启动：除了 Flash 的首地址地址 0x00000，还支持从 Flash 高地址 0x20000 (128K)、0x40000 (256K)、0x80000 (512K) 读取 Firmware 运行。本文档以高地址 0x20000 为例来介绍 OTA。

### 7.1 FLASH 存储架构

Flash 存储架构分为传统架构 (Secure Boot 功能未使能时) 和 Secure Boot 架构 (Secure Boot 功能使能时)。在两种架构中，如使用启动地址 0x20000 时，SDK 编译出来的 Firmware size 均应不大于 128K，即 Flash 的 0~0x20000 之间的区域存储 Firmware\_1\_，但是由于一些特殊的原因，如果使用启动地址为 0 和 0x20000 交替 OTA 升级，其 Firmware size 不得超过 124K (高地址空间最后的 4KB 都不能使用)；如果超过 124K 必须使用启动地址 0 和 0x40000 交替升级，此时最大 Firmware size 不得超过 252K，如果超过 252K 必须使用启动地址 0 和 0x80000 交替升级，此时最大 Firmware size 不得超过 508K。

Secure Boot 相关说明参考：[Secure Boot Application Note](#)

#### 7.1.1 传统存储架构

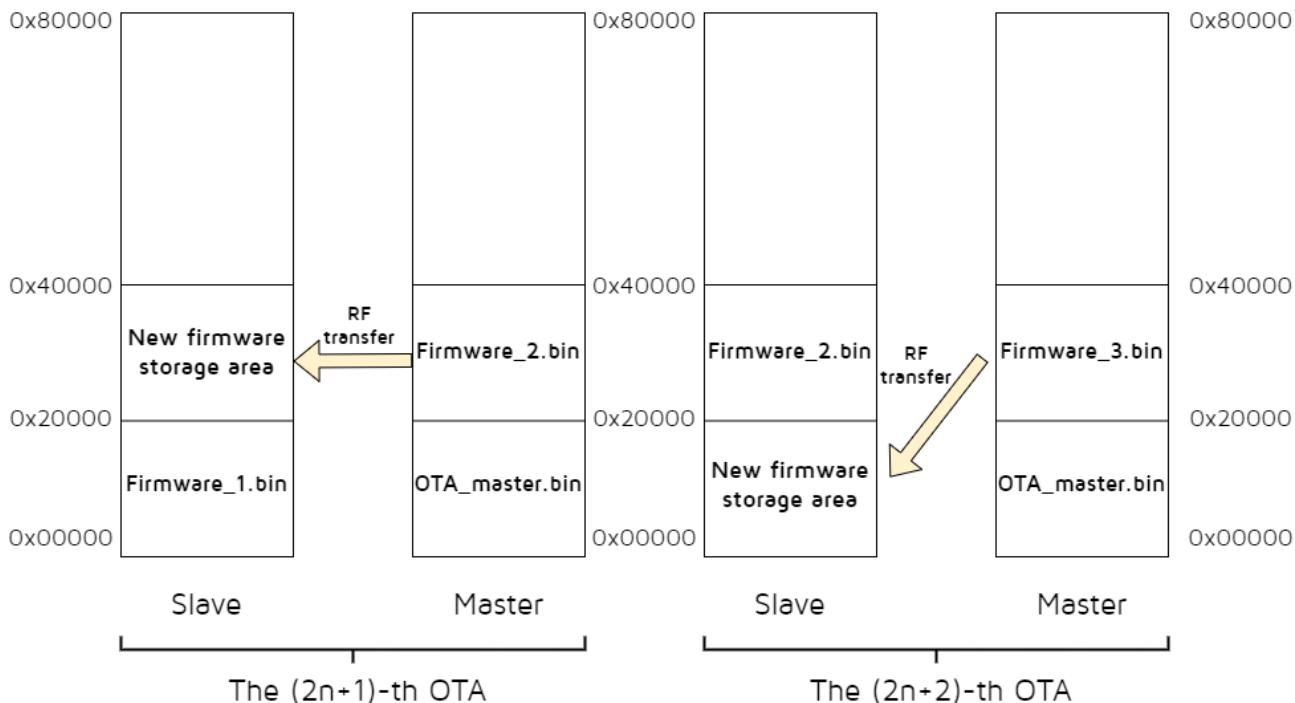


Figure 7.1: 传统 Flash 存储结构

- (1) OTA Client 将新的 firmware\_2 烧写到 0x20000~0x40000 的区域。



(2) 第 1 次 OTA:

- Server 上电时从 flash 的 0~0x20000 区域读程序启动，运行 firmware\_1；
- firmware\_1 运行，初始化的时候将 0x20000~0x40000 区域清空，该区域将作为新的 Firmware 的存储区。
- 启动 OTA，Client 通过蓝牙交互将 firmware\_2 空运到 Server 的 0x20000~0x40000 区域。升级成功后，Server 重启。

(3) 将新的 firmware\_3 烧写到 OTA Client 的 0x20000~0x40000 的区域。

(4) 第 2 次 OTA:

- Server 上电时从 flash 的 0x20000~0x40000 区域读程序启动，运行 firmware\_2；
- firmware\_2 运行，初始化的时候将 0~0x20000 区域清空，该区域将作为新的 Firmware 的存储区。
- 启动 OTA，Client 通过空中包将 firmware\_3 空运到 Server 的 0~0x20000 区域。升级成功后，Server 重启。

(5) 后面的 OTA 过程重复上面 (1) ~ (4) 过程，可理解为 (2) 代表第  $2n+1$  次 OTA，(3) 代表第  $2n+2$  次 OTA。

**注意：**若 B92 芯片使能了 Firmware Encryption，则软件必须调用如下接口配置了 Secure Boot OTA 功能，否则无法进行正常的 Firmware Encryption OTA。

```
ble_sts_t blc_ota_enableFirmwareEncryption(void);
```

### 7.1.2 Secure Boot 存储架构

如果使用 Secure Boot 功能，OTA 模块需参考 feature\_test/feature\_ota 进行特殊设计。作为 Client 端，在 fature\_ota/app\_config 中定义：

```
#define OTA_CLIENT_SEND_SECURE_BOOT_SIGNATURE_ENABLE 1
```

Client 端会向 Server 端发送 Secure Boot Application Note 中 Secure Boot 章节描述的 public key 和 signature 等内容，使 Server 端进行相关验证过程。

作为 Server 端，在 fture\_ota/app\_config 中定义：

```
#define APP_HW_FIRMWARE_ENCRYPTION_ENABLE 1
```

初始化过程将调用 API blc\_ota\_enableFirmwareEncryption 使能 Firmware 加密，MCU 读写 Flash 时自动完成解密过程，功能对应 Secure Boot Application Note 中 Firmware Encryption 章节。

在 fture\_ota/app\_config 中定义：

```
#define APP_HW_SECURE_BOOT_ENABLE 1
```

初始化过程将调用 API blc\_ota\_enableSecureBoot 使能 Secure Boot，MCU 在上电和 OTA 过程中加入对 Firmware 的验证过程，保证程序不被篡改，功能对应 Secure Boot Application Note 中 Secure Boot 章节。



当芯片使能了 Secure Boot，则软件必须调用如下接口配置了 Secure Boot OTA 功能，否则无法进行正常的 Secure Boot OTA。

```
ble_sts_t blc_ota_enableSecureBoot(void);
```

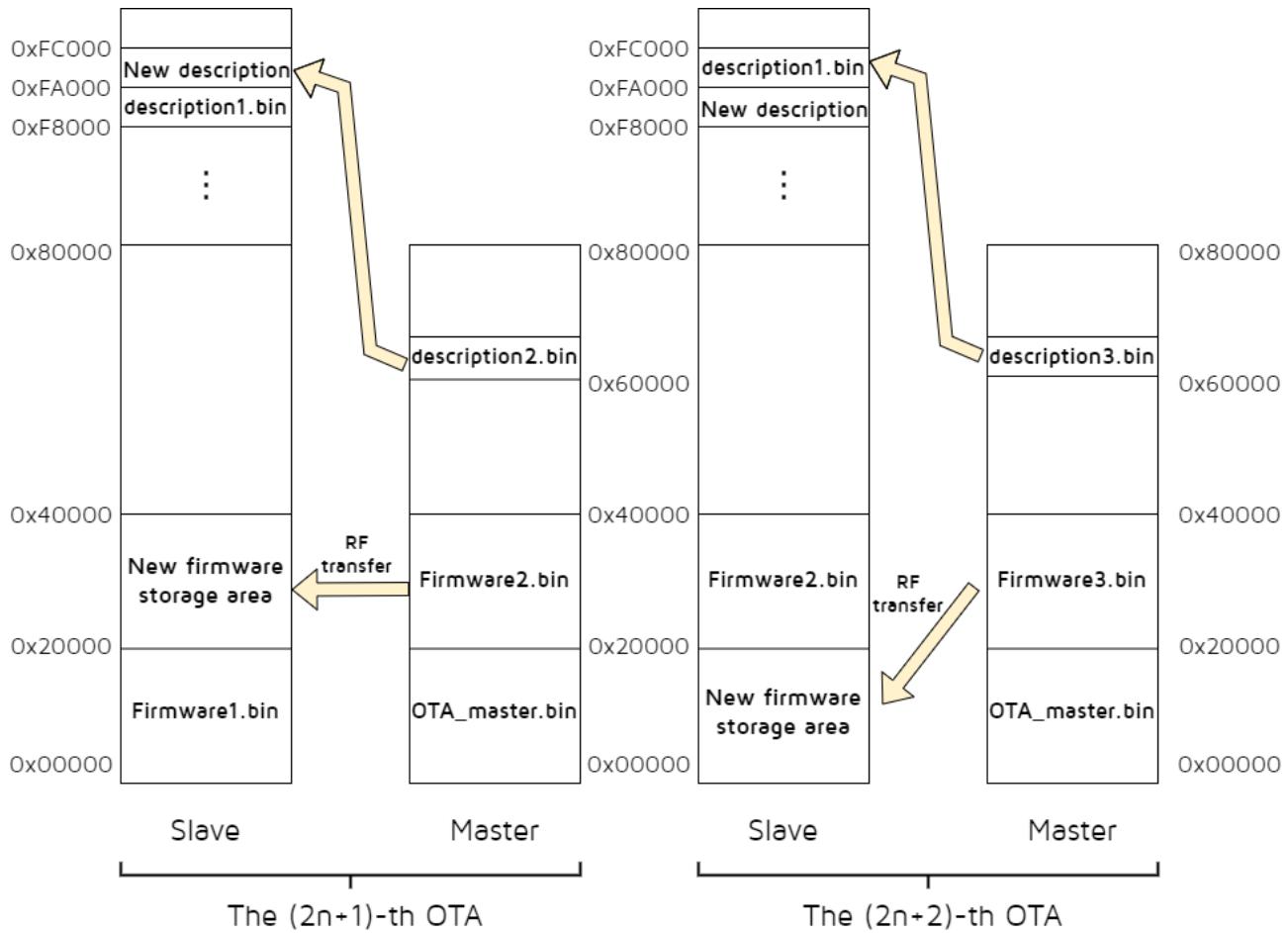


Figure 7.2: Secure Boot Flash 存储结构

以下以 1MB Flash 为例介绍，1MB Flash 中默认的描述符位置为 0xF8000 与 0xFA000，每个描述符占用 8kB 空间。

(1) OTA Client 将新的 firmware\_2 和描述符分别烧写到 0x20000~0x40000、0x60000 的区域。

(2) 第 1 次 OTA：

- Server 上电时从 flash 的描述符 0xF8000 中的程序地址参数 (图中以 0xF8000 中描述符对应程序地址 0x00000 为例)；
- 从对应 0x00000 区域读程序启动，运行 firmware\_1；
- firmware\_1 运行，初始化的时候将 0x20000~0x40000 中程序区域和另一描述符区域 0xFA000~0xFC000 清空，这两个区域将分别作为新 Firmware 和新描述符的存储区。
- 启动 OTA，Client 通过蓝牙将 firmware\_2、描述符空运到 Server 的 0x20000~0x40000、0xFA000 区域。升级成功后，Server 重启。



(3) 将新的 firmware\_3 烧写到 OTA Client 的 0x20000~0x40000 的区域。

(4) 第 2 次 OTA:

- Server 上电时从 flash 的描述符 0xFA000 中的程序地址参数 (图中以 0xFA000 中描述符对应程序地址 0x20000 为例);
- 从对应 0x20000 区域读程序启动, 运行 firmware\_2;
- firmware\_2 运行, 初始化的时候将 0x00000~0x20000 中程序区域和另一描述符区域 0xFA000~0xFC000 清空, 这两个区域将分别作为新 Firmware 和新描述符的存储区。
- 启动 OTA, Client 通过蓝牙将 firmware\_3、描述符空运到 Server 的 0x00000~0x20000、0xF8000 区域。升级成功后, Server 重启。

(5) 后面的 OTA 过程重复上面 (1) ~ (4) 过程, 可理解为 (2) 代表第 2n+1 次 OTA, (3) 代表第 2n+2 次 OTA。

## 7.2 OTA 更新流程

以上面的 FLASH 存储结构为基础, 详细说明 OTA 程序更新的过程。与 Flash 存储架构相对应, OTA 更新流程分为传统流程和 Secure Boot 流程。

下面首先介绍下传统流程。

### 7.2.1 传统 OTA 更新流程

仍以高地址 0x20000 为例, MCU 上电后, 默认从 0 地址启动, 首先去读 flash 0x20 的内容, 若该值为 0x544C4E4B, 则从 0 地址开始搬移代码到 RAM, 并且之后所有的取指都是从 0 地址开始, 即取指地址 = 0+PC 指针的值; 若 0x20 的值不为 0x544C4E4B, MCU 直接去读 0x20020 的值, 若该值为 0x544C4E4B, 则 MCU 从 0x20000 开始搬移代码到 RAM, 并且之后所有的取指都是从 0x20000 地址开始, 即取指地址 = 0x20000+PC 指针的值。

所以只要修改 0x20 和 0x20020 标志位的值, 即可指定 MCU 执行 FLASH 哪部分的代码。

SDK 上某一次 (2n+1 或 2n+2) 上电及 OTA 过程为:

- (1) MCU 上电, 通过读 0x20 和 0x20020 的值和 0x544C4E4B 作比较, 确定启动地址, 然后从对应的地址启动并执行代码。此功能由 MCU 硬件自动完成。
- (2) 程序初始化过程中, 读 MCU 硬件寄存器判断 MCU 是从哪个地址启动:

若从 0 启动, 将 ota\_program\_offset 设为 0x20000, 并将 0x20000 区域非 0xff 的内容全部擦除为 0xff, 表示下一次 OTA 获得的新 Firmware 会存入 0x20000 开始的区域;

若从 0x20000 启动, 将 ota\_program\_offset 设为 0x0, 并将 0x0 区域非 0xff 的内容全部擦除为 0xff, 表示下一次 OTA 获得的新 Firmware 会存入 0x0 开始的区域。

- (3) Server 程序正常运行, OTA Client 上电运行, 二者建立 BLE 连接。
- (4) 在 OTA Client 端 UI 触发进入 OTA 模式 (可以是按键、PC 工具写内存等)。OTA Client 进入 OTA 模式后, 首先需要获取 Server 端 OTA service 数据 Attribute 的 Attribute Handle 的值。
- (5) OTA Client 获取了 Server 端 OTA service 数据 Attribute 的 Attribute Handle 值后, 获取当前 Server 端 Firmware 版本号。

**注意:**



若采用 legacy protocol 则获取版本号需要用户自行实现；若采用 extend protocol 则获取版本号相关操作已实现。关于 legacy 与 extend protocol 的区别用户可参考[下文介绍](#)。

- (6) Client 确定要做 OTA 更新后，先发一个 OTA\_start 命令通知 Server 进入 OTA 模式。
- (7) Server 收到 OTA start 命令后，进入 OTA 模式，等待 Client 发 OTA 数据。
- (8) Client 从 0x20000 开始的区域读预先存储好的 firmware，不间断的向 Server 发送 OTA 数据，直至整个 firmware 都发过去。
- (9) Server 接收 OTA 数据，向 ota\_program\_offset 开始的区域存储。
- (10) Client 端发完所有的 OTA 数据后，检查这些数据 Server 是否都正确收到（调用底层 BLE 的相关函数判断 link layer 的数据是否都被正确 ack）。
- (11) Client 确定所有的 OTA 数据都被 Server 正确收到后，发送一个 OTA\_END 命令。
- (12) Server 收到 OTA\_END 命令，将新 firmware 区域偏移地址 0x20（即 ota\_program\_offset+0x20）写为 0x544C4E4B，将之前老的 firmware 存储区域偏移地址 0x20 的地方改写为 0x00000000，表示下一次程序启动后将从新的区域搬代码执行。
- (13) Server 通过 Handle Value Notification 将 OTA 的结果上报给 Client。
- (14) 将 Server reboot，新的 firmware 生效。

在整个 OTA 更新过程中，Server 会不断检查是否有错包和丢包，同时也会不断检查是否超时（OTA 开始的时候启动一个计时），一旦有错包、丢包或超时，Server 会认为更新失败，并向对方发送失败的原因，使用之前的 Firmware。

以上流程 Server 端相关操作在 SDK 上已经实现，用户不需要添加任何东西，Client 端需要额外的程序设计，后面会详细介绍。

### 7.2.2 Secure Boot OTA 更新流程

首先介绍一下 Secure Boot 中多地址启动机制（以 1M Flash 为例，两个描述符分别对应前两个启动地址 0x00000 和 0x20000）：MCU 上电后，默认从 0xF8000 读取描述符，首先去读 flash 0xF8000 的内容，若该值为 0x544C4E4B 且描述符内容校验通过，则从 0 地址开始搬移代码到 RAM，并且之后所有的取指都是从 0 地址开始，即取指地址 = 0+PC 指针的值；若 0xF8000 的值不为 0x544C4E4B，MCU 直接去读 0xFA000 的值，若该值为 0x544C4E4B 且描述符内容校验通过，则 MCU 从 0x20000 开始搬代码到 RAM，并且之后所有的取指都是从 0x20000 地址开始，即取指地址 = 0x20000+PC 指针的值。

所以只要修改 0xF8000 和 0xFA000 标志位的值，即可指定 MCU 执行 FLASH 哪部分的代码。

SDK 上某一次（2n+1 或 2n+2）上电及 OTA 过程为：

- (1) MCU 上电，通过读 0xF8000 和 0xFA000 的值和 0x544C4E4B 作比较，确定启动地址，完成程序校验，然后从对应的地址启动并执行代码。此功能由 MCU 硬件自动完成。
- (2) 程序初始化过程中，读 MCU 硬件寄存器判断 MCU 是从哪个地址启动：

若从 0 启动，将 ota\_program\_offset 设为 0x20000，并将 0x20000 区域非 0xff 的内容全部擦除为 0xff，表示下一次 OTA 获得的新 firmware 会存入 0x20000 开始的区域；

若从 0x20000 启动，将 ota\_program\_offset 设为 0x0，并将 0x0 区域非 0xff 的内容全部擦除为 0xff，表示下一次 OTA 获得的新 firmware 会存入 0x0 开始的区域。



- (3) Server 程序正常运行，OTA Client 上电运行，并与 Server 建立 BLE 连接。
- (4) 在 OTA Client 端 UI 触发进入 OTA 模式（可以是按键、PC 工具写内存等）。OTA Client 进入 OTA 模式后，首先需要获取 Server 端 OTA service 数据 Attribute 的 Attribute Handle 的值（可以 Server 事先和 Client 约定好，也可以通过 Read\_by\_type 获取这个 handle 值）。
- (5) OTA Client 获取了 Server OTA service 数据 Attribute 的 Attribute Handle 值后，获取当前 Server FLASH 程序的 firmware 版本号。

#### 注意：

若采用 legacy protocol 则获取版本号需要用户自行实现；若采用 extend protocol 则获取版本号相关操作已实现。关于 legacy 与 extend protocol 的区别用户可参考 7.3.2 小节。

- (6) Client 确定要做 OTA 更新后，先发一个 OTA\_start 命令通知 Server 进入 OTA 模式。
- (7) Server 收到 OTA start 命令后，进入 OTA 模式，等待 Client 发 OTA 数据。
- (8) Client 从 0x60000 开始的区域读预先存储好的描述符，将描述符中的 public key 和 signature 发送给 Server。
- (9) Server 收到 public key 和 signature 后，会对 public key 进行校验，如果校验失败，则退出 OTA，如果校验成功将 public key、signature、新的 firmware 运行地址信息等写入到新描述符区域对应位置。
- (10) Client 从 0x20000 开始的区域读预先存储好的 firmware，不间断的向 Server 发送 OTA 数据，直至整个 firmware 都发过去。
- (11) Server 接收 OTA 数据，向 ota\_program\_offset 开始的区域存储。
- (12) Client 端发完所有的 OTA 数据后，检查这些数据 Server 是否都正确收到（调用底层 BLE 的相关函数判断 link layer 的数据是否都被正确 ack）。
- (13) Client 确定所有的 OTA 数据都被 Server 正确收到后，发送一个 OTA\_END 命令。
- (14) Server 收到 OTA\_END 命令，对新 firmware 进行验签，验签通过后将新描述符存储区域偏移地址 0x0 写为 0x544C4E4B，将之前老的描述符存储区域偏移地址 0x0 的地方改写为 0x00000000，表示下一次程序启动后将从新的描述符区域读取执行。
- (15) Server 通过 Handle Value Notification 将 OTA 的结果上报给 Client。
- (16) 将 Server reboot，新的 firmware 生效。

在整个 OTA 更新过程中，Server 会不断检查是否有错包、丢包、超时（OTA 开始的时候启动一个计时），同时也会检查是否公钥错误、验签错误，一旦有错包、丢包、公钥错误、验签错误或超时，Server 会认为更新失败，并向对方发送失败的原因，使用之前的 firmware。

以上流程 Server 端相关操作在 SDK 上已经实现，用户不需要添加任何东西，Client 端需要额外的程序设计，后面会详细介绍。

### 7.2.3 修改 Firmware size 和 boot address

API blc\_ota\_setFirmwareSizeAndBootAddress 支持修改启动地址。这个启动地址指的是 OTA 设计中除了 0 地址外另一个存储 New\_firmware 的地址（只能是 0x20000、0x40000 或 0x80000）。



Firmware_Boot_address	Firmware size (max)/K
0x20000	124
0x40000	252
0x80000	508

SDK 中默认的最大 firmware size 为 252K (由于一些特殊的原因, 启动地址为 0x40000 的 firmware size 不得大于 252K), 对应的启动地址为 0x00000 和 0x40000。这两个值和前文的描述一致, 用户在设置时需要遵循表 7-1 启动地址与 firmware\_size 大小的约束关系, 如果最大 firmware\_size 发生变化, 超过了 124K, 此时需要将启动地址挪到 0x40000 (size 最大不得超过 252K), 同理如果 firmware\_size 超过 252K, 需要将启动地址挪到 0x80000 (size 最大不得超过 508K), 比如最大 firmware size 可能到 200K 用户可以调用 API blc\_ota\_setFirmwareSizeAndBootAddress 来进行设置:

```
ble_sts_t blc_ota_setFirmwareSizeAndBootAddress(int firmware_size_k, multi_boot_addr_e
↪ boot_addr);
```

#### 注意:

该 API 的调用必须在 sys\_init 函数前。

参数 multi\_boot\_addr\_e 表示可供选择的启动地址, 共用三种:

```
typedef enum{
    MULTI_BOOT_ADDR_0x20000 = 0x20000, //128 K
    MULTI_BOOT_ADDR_0x40000 = 0x40000, //256 K
    MULTI_BOOT_ADDR_0x80000 = 0x80000, //512 K
};
```

返回值 ble\_sts\_t 表示设置的状态, 关于该类型的定义可参考 SDK 中 ble\_common.h。

若成功成功, 则返回 BLE\_SUCCESS; 否则返回 SERVICE\_ERR\_INVALID\_PARAMETER。

## 7.3 OTA 模式 RF 数据处理

### 7.3.1 Attribute Table 中 OTA 的处理

Server 端在 Attribute Table 中添加 OTA 的相关内容, 其中 OTA 数据 Attribute 的 att\_readwrite\_callback\_t w 设为 otaWrite, 将属性设为 Read 和 Write\_without\_Rsp。OTA Client 默认采用 Write Command 发数据, 不需要 Server 回 ack, 速度会更), 若 Client 采用 Write Request 发数据, 需要更改 gatt 的特性权限为允许 Server 端响应 (CHAR\_PROP\_WRITE\_WITHOUT\_RSP 更改为 CHAR\_PROP\_WRITE)。

```
// OTA attribute values
static const u8 my_OtaCharVal[19] = {
```

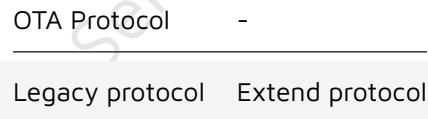


```
CHAR_PROP_READ | CHAR_PROP_WRITE_WITHOUT_RSP,  
U16_LO(OTA_CMD_OUT_DP_H), U16_HI(OTA_CMD_OUT_DP_H),  
TELINK_SPP_DATA_OTA, };  
  
{5,ATT_PERMISSIONS_READ, 2,16,(u8*)(&my_primaryServiceUUID), (u8*)(&my_OtaServiceUUID),  
↪ 0},  
{0,ATT_PERMISSIONS_READ, 2, sizeof(my_OtaCharVal),(u8*)(&my_characterUUID), (u8*)  
↪ (my_OtaCharVal), 0}, //prop  
{0,ATT_PERMISSIONS_RDWR,16,sizeof(my_OtaData),(u8*)(&my_OtaUUID), (&my_OtaData),  
↪ &otaWrite, NULL}, //value  
{0,ATT_PERMISSIONS_RDWR,2,sizeof(otaDataCCC),(u8*)(&clientCharacterCfgUUID), (u8*)  
↪ (otaDataCCC), 0},  
{0,ATT_PERMISSIONS_READ, 2,sizeof (my_OtaName),(u8*)(&userdesc_UUID), (u8*)(my_OtaName), 0},  
↪
```

Client 向 Server 发送 OTA 数据时，实际是向上面第 3 个 Attribute 写数据，Client 需要知道这个 Attribute 在整个 Attribute Table 中的 Attribute Handle。

### 7.3.2 OTA Protocol

目前 OTA 架构对功能进行了扩展并且兼容以前旧版本的协议，整个 OTA 协议包含了 Legacy protocol 和 Extend protocol 两个部分：



#### 注意：

OTA protocol 支持的功能：

- (1) OTA Result feedback function: 该功能不可选，默认添加；
- (2) FirmWare Version Compare function 和 Big PDU function: 该功能可选，可不添加，需要注意一点其中的版本号比较功能在 Legacy protocol 和 Extend protocol 中实现有所区别，具体可参考下文 OTA\_CMD 部分介绍。

下面的介绍均围绕 Legacy 和 Extend protocol 进行介绍。

#### OTA\_CMD 组成

OTA 的 CMD 的 PDU 如下：





## Opcode

Opcode	Name	Use*
0xFF00	CMD_OTA_VERSION	Legacy
0xFF01	CMD_OTA_START	Legacy
0xFF02	CMD_OTA_END	All
0xFF03	CMD_OTA_START_EXT	Extend
0xFF04	CMD_OTA_FW_VERSION_REQ	Extend
0xFF05	CMD_OTA_FW_VERSION_RSP	Extend
0xFF06	CMD_OTA_RESULT	All
0xFF10~0xFF17	CMD_OTA_SB_PUBKEY_SIGN	All

### 注意：

- Use: 识别在 Legacy protocol、Extend protocol 或两者中均可使用的命令；
- Legacy: 只在 Legacy protocol 中使用；
- Extend: 只在 Extend protocol 中使用；
- All: 在 Legacy protocol 和 Extend protocol 中均可使用。
- CMD\_OTA\_SB\_PUBKEY\_SIGN 仅在 Secure Boot 使能时使用

#### (1) CMD\_OTA\_VERSION

该命令为获得 Server 当前 firmware 版本号的命令，用户若采用 OTA Legacy protocol 进行 OTA 升级，可以选择使用，在使用该命令时，可通过 Server 端预留的回调函数来完成 firmware 版本号的传递。

```
void blc_ota_registerOtaFirmwareVersionReqCb(ota_versionCb_t cb);
```

Server 端在收到 CMD\_OTA\_VERSION 命令时会触发该回调函数。

#### (2) CMD\_OTA\_START

该命令为 OTA 升级开始命令，Client 发这个命令给 Server，用来正式启动 OTA 更新。该命令仅供 Legacy Protocol 进行使用，用户若采用 OTA Legacy protocol，则必须使用该命令。

#### (3) CMD\_OTA\_END

该命令为结束命令，OTA 中的 legacy 和 extend protocol 均采用该命令为结束命令，当 Client 确定所有的 OTA 数据都被 Server 正确接收后，发送 OTA end 命令。为了让 Server 再次确定已经完全收到了 Client 所有数据（double check，加一层保险），OTA end 命令后面带 4 个有效的 bytes，后面详细介绍。





- Adr\_index\_max: 最大的 adr\_index 值
- Adr\_index\_max\_xor: Adr\_index\_max 的异或值, 供校验使用
- Reserved: 保留供以后功能扩展使用

#### (4) CMD\_OTA\_START\_EXT

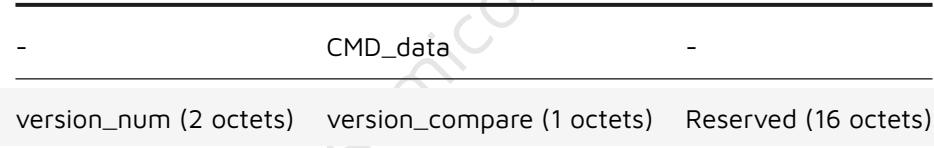
该命令为 extend protocol 中的 OTA 升级开始命令, Client 发这个命令给 Server, 用来正式启动 OTA 更新。用户若采用 OTA extend protocol 则必须采用该命令作为开始命令。



- Length: PDU length
- Version\_compare: 0x01: 开启版本比较功能 0x00: 关闭版本比较功能
- Reserved: 保留供以后扩展使用

#### (5) CMD\_OTA\_FW\_VERSION\_REQ

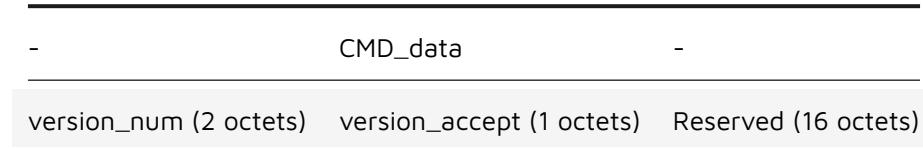
该命令为 OTA 升级过程中的版本比较请求命令, 该命令由 Client 发起给 Server 端, 请求获取版本号和升级许可。



- Version num: Client 端待升级的 Firmware 版本号
- Version compare: 0x01: 开启版本比较功能 0x00: 关闭版本比较功能
- Reserved: 保留供以后扩展使用

#### (6) CMD\_OTA\_FW\_VERSION\_RSP

该命令为版本响应命令, Server 端在收到 Client 发来的版本比较请求命令 (CMD\_OTA\_FW\_VERSION\_REQ) 后, 会将已有的 Firmware 版本号与 Client 端请求升级的版本号进行对比, 确定是否升级, 相关信息通过该命令返回发送给 Client。



- Version num: Server 端当前运行的 Firmware 版本号
- Version\_accept: 0x01: 接受 Client 端升级请求, 0x00: 拒绝 Client 端升级请求
- Reserved: 保留供以后扩展使用

#### (7) CMD\_OTA\_SB\_PUBKEY\_SIGN

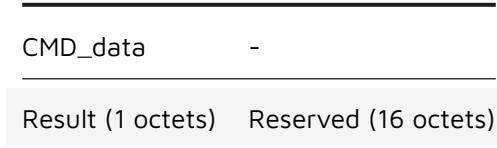


该命令为 public key 和 signature 传输命令，Server 端在收到 Client 发来的版本 public key 和 signature 传输命令 (CMD\_OTA\_SB\_PUBKEY\_SIGN) 后，会将已有的 public key 与 Client 端发送的 public key 进行对比，确定是否允许 OTA。该命令总共分为 8 包发送，每包发送 16 byte。



#### (8) CMD\_OTA\_RESULT

该命令为 OTA 结果返回命令，OTA 结束后 Server 会将结果信息发送给 Client，在整个 OTA 过程中，无论成功或失败，OTA\_result 只会上报一次，用户可根据返回的结果来判断升级是否成功。



Result：OTA 结果信息，所有可能的返回结果如下表所示：

**Table 7.11: OTA 所有可能的返回结果**

Value	Type info
0x00	OTA_SUCCESS success
0x01	OTA_DATA_PACKET_ OTA data packet sequence number error: repeated OTA PDU or lost some OTA PDU SEQ_ERR
0x02	OTA_PACKET_INVALID invalid OTA packet: 1. invalid OTA command; 2. addr_index out of range; 3.not standard OTA PDU length
0x03	OTA_DATA_CRC_ERR packet PDU CRC err
0x04	OTA_WRITE_FLASH_ERR write OTA data to flash ERR
0x05	OTA_DATA_INCOMPLETE lost last one or more OTA PDU
0x06	OTA_FLOW_ERR peer device send OTA command or OTA data not in correct flow
0x07	OTA_FW_CHECK_ERR firmware CRC check error
0x08	OTA_VERSION_ the version number to be update is lower than the current version COMPARE_ERR
0x09	OTA_PDU_LEN_ERR PDU length error: not 16*n, or not equal to the value it declare in "CMD_OTA_START_EXT" packet
0x0a	OTA_FIRMWARE_ firmware mark error: not generated by telink's BLE SDK MARK_ERR
0x0b	OTA_FW_SIZE_ERR firmware size error: no firmware_size; firmware size too small or too big
0x0c	OTA_DATA_PACKET_ time interval between two consequent packet exceed a value(user can adjust this value) TIMEOUT



0x0d OTA\_TIMEOUT OTA flow total timeout

0x0e OTA\_FAIL\_DUE\_TO\_OTA fail due to current connection terminate(maybe connection timeout or local/peer device terminate connection) CONNECTION \_TERMIANTE

0x0f OTA MCU NOT SUPPORTED MCU does not support this OTA mode

0x10 OTA\_LOGIC\_ERR software logic error, please contact FAE of Telink

0x80 OTA\_SECBOOT\_HW\_ERR OTA server device hardware error

0x81 OTA\_SECBOOT\_SYSTEM\_ERR OTA server device system error

0x82 OTA\_SECBOOT\_FUNC\_NOT\_ENABLE OTA server device do not enable secure boot function

0x83 OTA\_SECBOOT\_PUBKEY\_SIGN\_SEQ\_ERR OTA public key & signature sequence number error: repeated or lost

0x84 OTA\_SECBOOT\_PUBKEY\_SIGN\_LEN\_ERR OTA public key & signature data packet length error

0x85 OTA\_SECBOOT\_PUBLIC\_KEY\_ERR OTA client public key not match OTA server device local hash

0x86 OTA\_SECBOOT\_SIGN\_VERIFY\_FAIL OTA signature verification fail

0x87 OTA\_SECBOOT\_WRITE\_DESC\_FAIL write secure boot descriptor fail

0x88 OTA\_SECBOOT\_NEW\_FW\_NOT\_MATCH\_OLD\_FW secure boot function: new firmware not match old firmware 1. old firmware enable secure boot, but new firmware do not enable 2. old firmware do not enable secure boot, but new firmware enable

0x89 OTA\_FWENC\_NEW\_FW\_NOT\_MATCH\_OLD\_FW firmware encryption function: new firmware not match old firmware 1. old firmware enable firmware encryption, but new firmware do not enable 2. old firmware do not enable firmware encryption, but new firmware enable

Other Reserved for future use /

#### 注意：

- 0x80~0x89 只适用于 Secure Boot 模式中。

#### OTA Packet 结构组成：

Client 在采用 WirteCommand 或 WriteRequest 向 Server 端发命令和数据时, ATT 层有关的 Attribute Handle 的值为 Server 端 OTA 数据的 handle\_value。根据 Ble Spec L2CAP 层有关 PDU format 的规范, Attribute Value 长度定义为下图中的 OTA\_DataSize 部分。

DLE_Size							
				MTU_Size			
					OTA_Data_Size		
Rf_len	L2CAP_len	CID	Opcode	Att_Handle	Adr_index	OTA_PDU	CRC
1octet	2octets	2octets	1octet	2octets	2octets	16-240 octets	2octets

Figure 7.3: L2CAP PDU 中对应 OTA packet



- DLE Size: CID + Opcode + Att\_Handle + Adr\_index + OTA\_PDU + CRC
- MTU\_Size: Opcode + Att\_Handle + Adr\_index + OTA\_PDU + CRC
- OTA\_Data\_Size: Adr\_index + OTA\_PDU + CRC

### OTA\_Data 介绍:

Type	Length
Default* + BigPDU*	16octets - 240Octets(n*16,n=1..15)

### 注意:

- Default OTA PDU 长度固定默认大小为 16octets
- BigPDU: OTA PDU 长度可更改范围为 16octets – 240 octets, 且为 16 字节整数倍。

### OTA\_PDU Format

当用户采用 OTA 中的 Extend protocol, 支持 Big PDU, 即可支持长包进行 OTA 升级操作, 减少 OTA 升级的时长, 用户可根据需要在 Client 端自定义设置 PDU 大小。最后两个 byte 是将前面的 Adr\_Index 和 Data 进行一个 CRC\_16 计算得到第一个 CRC 的值, Server 收到 OTA data 后, 会进行同样的 CRC 计算, 只有两者计算的 CRC 吻合时, 才认为这是一个有效数据。

-	OTA PDU	-
Adr_Index (2 octets)	Data(n*16 octets) n=1..15	CRC (2 octets)

(1) PDU 包长度: n = 1

Data : 16 octets

Adr\_Index 与 Firmware address 的映射关系:

Adr_Index	Firmware_address
0x0001	0x0000 - 0x000F
0x0002	0x0010 - 0x001F
.....	.....
XXXX	(XXXX -1)*16 - (XXXX)*16+15

(2) PDU 包长度: n = 2

Data : 32 octets

Adr\_Index 与 Firmware address 的映射关系:

Adr_Index	Firmware_address
0x0001	0x0000 - 0x001F
0x0002	0x0010 - 0x003F
.....	.....
XXXX	(XXXX -1)*32 - (XXXX)*32+31

(3) PDU 包长度:  $n=15$

Data : 240 octets

Adr\_Index 与 Firmware address 的映射关系：

Adr_Index	Firmware_address
0x0001	0x0000 - 0x00EF
0x0002	0x0010 - 0x01DF
.....	.....
XXXX	(XXXX -1)240 - (XXXX)240+239

### 注意：



### 7.3.3 RF Transfer 处理方法

Client 端通过 L2CAP 层的 Write Command 或 Write Request 向 Server 发命令和数据, Spec 规定在收到 Write Request 后必须返回 Write Response。

下面将分别对 Legacy Protocol 和 Extend Protocol、以及 OTA 的 Version Compare 的流程进行介绍，阐述整个 RF Transform 中 Server 和 Client 的交互过程。



### OTA Legacy Protocol 流程

OTA Legacy 兼容 Telink 上一版的 OTA 协议，为更好说明 Server 端和 Client 端整个交互过程，下面采用举例说明：

#### 注意：

- PDU 长度采用默认的 16 octets 大小，不涉及 DLE 长包的操作。
- Firmware compare 功能不选择。

具体的操作流程如下图所示：

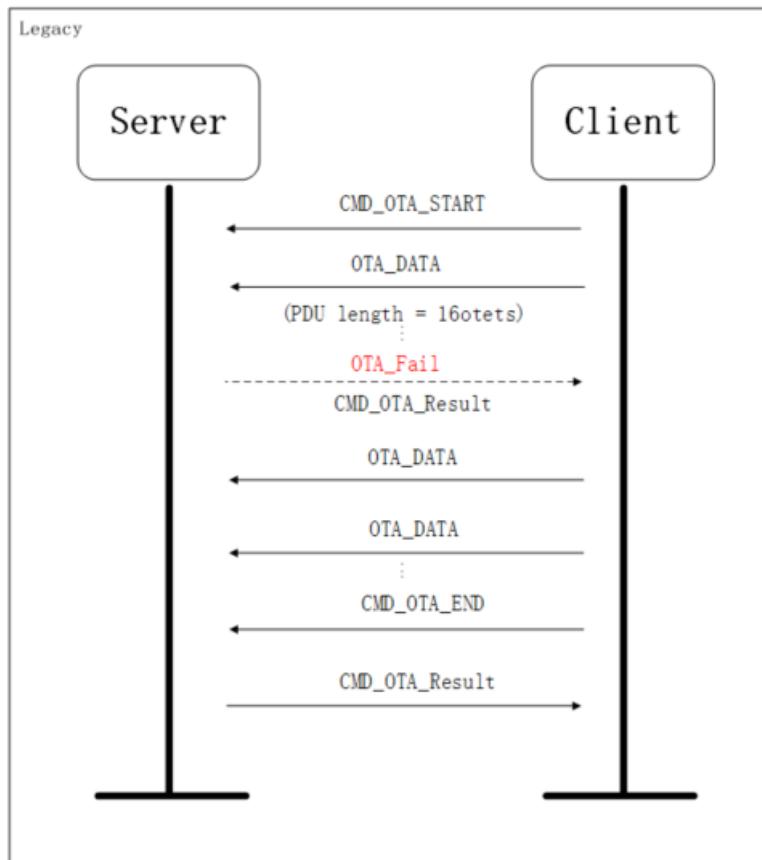


Figure 7.4: OTA Legacy protocol 流程

Client 端首先给 Server 端发送 CMD\_OTA\_START 命令，Server 端在收到命令后开始准备接收 OTA 数据，接着 Client 端开始发送 OTA\_Data，该过程中若出现任何交互失败，Server 端会向 Client 端发送 CMD\_OTA\_Result 即返回错误信息，重新运行原程序但不会进入 reboot，Client 端收到后会立即停止 OTA 数据传输。若 Client 端和 Server 端成功完成了 OTA\_Data 传输，则 Client 端会向 Server 端发送 CMD\_OTA\_END，Server 端在收到后返回结果信息发送 CMD\_OTA\_Result 给 Client 端，并且进入 reboot，运行新的 Firmware。

### OTA Extend Protocol 流程

如前文所述，OTA Extend 与上面介绍的 Legacy 的交互命令存在部分区别，为更好说明 Server 端和 Client 端整个交互过程，下面采用举例说明：

#### 注意：



- PDU 长度采用 64octets 大小，涉及 DLE 长包的操作。
- Firmware compare 功能不选择。

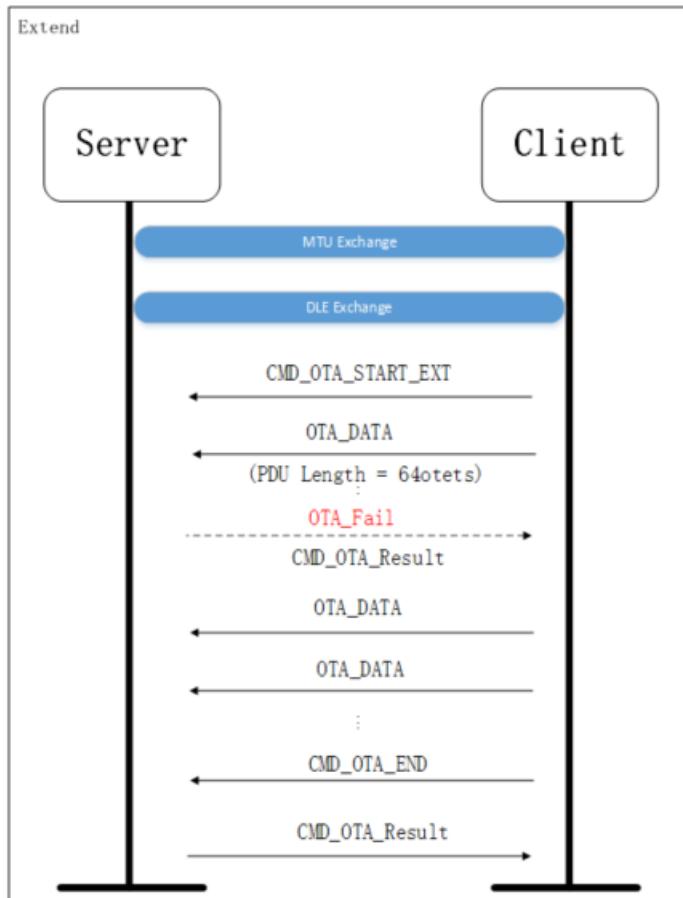


Figure 7.5: OTA Extend Protocol 流程

由于采用了 DLE 长包功能，Client 端首先需要与给 Server 端进行 MTU 和 DLE 交互，然后接下来的流程和前面 Legacy 类似，Client 端向 Server 端发送 CMD\_OTA\_START\_EXT 命令，Server 端在收到命令后开始准备接收 OTA 数据，接着 Client 端开始发送 OTA\_Data，该过程中若出现任何交互失败，Server 端会向 Client 端发送 CMD\_OTA\_Result 即返回错误信息，重新运行原程序但不会进入 reboot，Client 端收到后会立即停止 OTA 数据传输。若 Client 端和 Server 端成功完成了 OTA\_Data 传输，则 Client 端会向 Server 端发送 CMD\_OTA\_END，Server 端在收到后返回结果信息发送 CMD\_OTA\_Result 给 Client 端，并且进入 reboot，运行新的 Firmware。

#### OTA Version Compare 流程

在 Server 端，Extend 和 Legacy Protocol 都具有版本比较功能，其中 Legacy 预留了接口，需要用户自行实现，而 Extend 中已经实现了版本比较的功能，用户可以直接使用。

下面将 Extend 中具有版本比较功能的交互流程进行举例说明：

#### 注意：

- PDU 长度采用 16octets 大小，不涉及 DLE 长包的操作。
- Firmware compare 功能选择 (OTA 待升级版本号为 0x0001, 开启版本比较使能)。

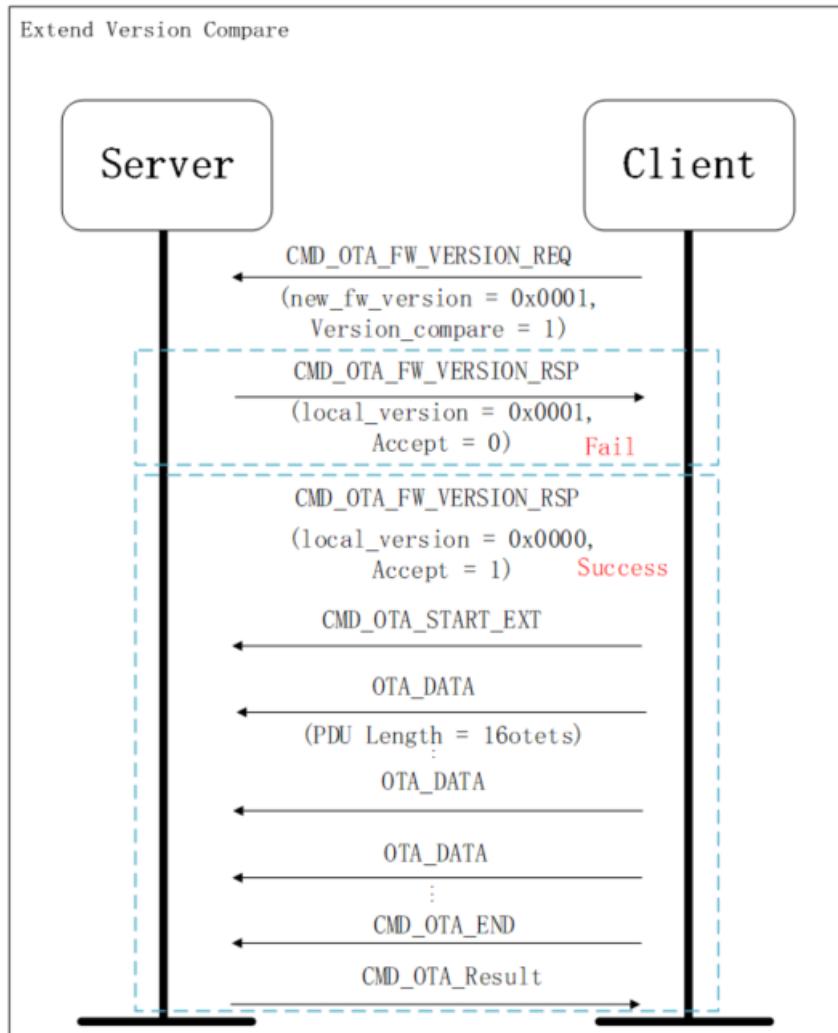


Figure 7.6: OTA Version Compare 流程

在使能版本比较功能后，Client 端首先给 Server 端发送 **CMD\_OTA\_FW\_VERSION\_REQ** 版本比较请求命令，其中发送的 PDU 中包括 Client 端 Firmware 版本号 (`new_fw_version = 0x0001`)，Server 端获取 Client 端的版本号信息并与本地版本号 (`local_version`) 进行对比：

若接收的版本号 (`new_fw_version = 0x0001`) 不大于本地版本号 (`local_version = 0x0001`)，则 Server 端会拒绝 Client 端 OTA 升级请求，发送给 Client 端版本号响应的命令 (**CMD\_OTA\_FW\_VERSION\_RSP**)，发送的信息包括接收参数 (`accept = 0`) 和本地版本号 (`local_version = 0x0001`)，Client 在收到后停止 OTA 相关操作，即当前版本升级不成功。

若接收的版本号 (`new_fw_version = 0x0001`) 大于本地版本号 (`local_version = 0x0000`)，则 Server 端会接收 Client 端 OTA 升级请求，发送给 Client 端版本号响应的命令 (**CMD\_OTA\_FW\_VERSION\_RSP**)，发送的信息包括接收参数 (`accept = 1`) 和本地版本号 (`local_version = 0x0000`)，Client 在收到后开始准备 OTA 升级相关操作，过程与前文类似，即首先向 Server 端发送 **CMD\_OTA\_START** 命令，Server 端在收到命令后开始准备接收 OTA 数据，接着 Client 端开始发送 **OTA\_Data**，该过程中若出现任何交互失败，Server 端会向 Client 端发送 **CMD\_OTA\_Result** 即返回错误信息，重新运行原程序但不会进入 reboot，Client 端收到后会立即停止 OTA 数据传输。若 Client 端和 Server 端成功完成了 **OTA\_Data** 传输，则 Client 端会向 Server 端发送 **CMD\_OTA\_END**，Server 端在收到后返回结果信息发送 **CMD\_OTA\_Result** 给 Client 端，并且进入 reboot，运行新的 Firmware。



## OTA 具体实现

上面介绍了整个 OTA 交互流程，下面举例说明一下 Client 和 Server 具体的数据交互实现：

### 注意：

- OTA Protocol: Legacy Protocol;
- PDU 长度采用 16octets 大小，不涉及 DLE 长包的操作；
- Client 端开启 Firmware compare 功能。

- (1) 检测查询是否有触发进入 OTA 模式的行为，一旦检测到该行为，进入 OTA 模式。
- (2) Client 向 Server 传送 OTA 命令和数据，需要知道 Server 端当前 OTA 数据的 Attribute 的 Attribute Handle 值。

若用户采用事先约定好的方式，直接定义该值；

若没有事先约定好，采用 Read By Type Request 的方式获得这个 Attribute Handle 值。

Telink 所有 BLE SDK 的 OTA data 的 UUID 都是 16bytes，且永远都是下面这个值：

```
#define TELINK_SPP_DATA_OTA {0x12, 0x2B, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,  
↪ 0x03, 0x02, 0x01, 0x00}
```

在 Client 的 Read By Type Request 中将 Type 设置为这 16 个 bytes 的 UUID，Server 端回复的 Read By Type Rsp 中可以查到 OTA UUID 所在的这个 Attribute Handle，如下图所示，Client 可以查到 Attribute Handle 的值为 0x0031。

ATT_Read_By_Type_Req											
Data Type	Data Header				L2CAP Header						
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttType
	2	0	0	0	25	0x0015	0x0004	0x08	0x0001	0xFFFF	12 2B 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00
Data Type	Data Header				CRC	RSSI (dBm)	FCS				
Empty PDU	1	1	1	0	0	0x8FEFDC	0	OK			
Data Type	Data Header				L2CAP Header		ATT_Read_By_Type_Rsp		CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode Length AttData	0x79893F	0	OK
	2	0	1	0	9	0x0005	0x0004	0x09 0x03 31 00 00			

Figure 7.7: Client 通过 Read By Type Request 获取 OTA 的 Attribute Handle

- (3) 获取 Server 当前 Firmware 版本号，决定是否要继续做 OTA 更新（若版本已经最新，不需要更新）。这一步为用户自己选择是否要做。tl\_ble\_sdk Legacy protocol 并没有实现版本号的传送。用户可以使用 write cmd 或 write response 的形式通过 OTA version cmd 向 Server 传送一个获取 OTA version 的请求，Server 端在收到 OTA version 请求的时候在回调函数里将 Server 端的版本号传送给 Client（如手动送一个 NOTIFY/INDICATE 的数据）。

- (4) 启动 OTA 开始的一个计时，后面要不断检测该计时是否超过 30 秒（这只是个参考时间，实际根据用户测试的正常 OTA 需要多少时间后再做评估）。

如果超过 30 秒认为 OTA 超时失败，因为 Server 端收到 OTA 数据后会校验 CRC，一旦 CRC 错误或者出现其他错误（如烧写 flash 错误），就会认为 OTA 失败。

- (5) 读取 Client flash 0x20018~0x2001b 四个字节，确定 Firmware 的 size。

这个 size 是由我们的编译器实现的，假设 Firmware 的 size 为 20k = 0x5000，那么 Firmware 的 0x18<sub>0x1b</sub> 的值为 0x00005000，所以在 0x20018~0x2001b 可以读到 Firmware 的大小。



如下图所示的 bin 文件，0x18 ~0x1b 内容为 0x0000cf94，所以大小为  $0xcf94 = 53140\text{Bytes}$ ，从 0x0000 到 0xcf96。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	F	3	2	2	0	3	4	1	1	A	0	2	0	0	6	0
00000010	2	1	A	8	0	0	0	0	0	9	4	C	F	0	0	0
00000020	4	B	4	E	4	C	5	4	0	0	0	3	B	1	7	D
00000030	9	7	0	2	0	A	E	0	9	3	8	2	0	2	9	3

Figure 7.8: Firmware 示例-开头部分

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F							
0000CF60	0	0	0	E	8	0	3	0	0	2	8	0	0	0	5	2	A	0	1	0	0	0	
0000CF70	0	A	4	8	0	A	0	0	0	2	0	1	0	0	0	0	1	0	2	0	8	2	9
0000CF80	2	0	A	1	0	A	8	0	8	4	8	0	A	9	0	5	2	A	0	1	2	9	FFFF
0000CF90	E	C	6	E	D	D	A	9															

Figure 7.9: Firmware 示例-结尾部分

(6) 向 Server 发一个 OTA start 命令，通知 Server 进入 OTA 模式，等待 Client 端的 OTA 数据，如下图所示。

Data Type	Data Header					L2CAP Header		ATT_Write_Command			CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue	0x61875B	0	OK

Figure 7.10: Client 发 OTA start

(7) 从 Client flash 0x20000 区域开始每次读 16 个 byte 的 Firmware，填入 OTA data packet，设置对应的 adr\_index，并计算 CRC 值，将 packet push 到 TX fifo，一直到 Firmware size 最后一个 16 byte 为止，将 Firmware 所有的数据全部发送给 Server。

数据发送方法如前面介绍，使用 OTA data 的格式，有效数据为 20 bytes，前两个 bytes 放 adr\_index，紧跟 16 个有效的 Firmware 数据，最后两个是前 18 个数据的 CRC 计算值。

注意，如果 Firmware 最后一笔数据不是 16 字节对齐，需要将剩余的部分按 0xff 补对齐，计算 CRC 的时候需要将补充的数据计算进去。

结合上图所示的 bin 文件来详细介绍 OTA 数据如何拼装。

第一笔数据：adr\_index 为 0x00 00，16 个数据为 0x0000 ~0x000f 地址的值，然后这 18 个数据计算 CRC，假设 CRC 结果为 0xXYZW，那么 20bytes 排列为：

0x00 0x00 0xf3 0x22 .... 省略 12 个 bytes..... 0x60 0x15 0xZW 0xXY

第二笔数据：

0x01 0x00 0x21 0xa8 .... 省略 12 个 bytes..... 0x00 0x00 0xJK 0xHI

第三笔数据：

0x02 0x00 0x4b 0x4e .... 省略 12 个 bytes..... 0x81 0x7d 0xNO 0xLM

.....



倒数第二笔数据：

0xf8 0x0c 0x20 0xa1 .... 省略 12 个 bytes..... 0xff 0xff 0xST 0xPQ

最后一笔数据：

0xf9 0x0c 0xec 0x6e 0xdd 0xa9 0xff 0xff 0xff 0xff

0xff 0xff 0xff 0xff 0xff 0xff 0xWX 0xUV

12 “0xff” are added to complement 16 bytes.

12 个 0xff 为补齐的数据。

0xec 0x6e 0xdd 0xa9 为第 3 个 ~ 第 6 个，它是整个 Firmware bin 的 CRC\_32 校验结果。Server 在 OTA 升级过程中会同步计算接收到的整个 bin 的 CRC\_32 校验值，在结束时会与 0xec 0x6e 0xdd 0xa9 进行比较。

0xf9 ~0xff 共 18 个 bytes 的 CRC 计算结果为 0xUVWX。

上面的数据如下图所示：

① ATT Write Command Packet (00010203-0405-0607-0809-0A0B0C0D2B12: 01 FF)  
② ATT Write Command Packet (00010203-0405-0607-0809-0A0B0C0D2B12: 00 00 F3 22 20 34 11 A0 5D 02 63 84 02 00 6F 00 60 15 7B 35)  
③ ATT Write Command Packet (00010203-0405-0607-0809-0A0B0C0D2B12: 01 00 21 A8 00 00 00 00 00 00 94 CF 00 00 00 00 00 00 67 BD)  
④ ATT Write Command Packet (00010203-0405-0607-0809-0A0B0C0D2B12: 02 00 4B 4E 4C 54 00 00 3B 17 97 01 08 E0 93 81 81 7D 74 D4)  
⑤ ATT Write Command Packet (00010203-0405-0607-0809-0A0B0C0D2B12: 03 00 97 02 0A E0 93 82 02 FD 16 81 97 02 00 E0 93 82 59 5B)  
⑥ ATT Write Command Packet (00010203-0405-0607-0809-0A0B0C0D2B12: 04 00 62 FC 73 90 02 80 99 62 F3 A2 02 30 73 10 30 00 63 36)

Figure 7.11: Client OTA 数据 1

① ATT Write Command Packet (00010203-0405-0607-0809-0A0B0C0D2B12: F8 0C 20 A1 0A 08 08 48 0A 09 05 2A 01 29 FF FF FF FF AA 24)  
② ATT Write Command Packet (00010203-0405-0607-0809-0A0B0C0D2B12: F9 0C EC 6E DD A9 FF E3 DF)  
③ ATT Write Command Packet (00010203-0405-0607-0809-0A0B0C0D2B12: 02 FF F9 0C 06 F3)

Figure 7.12: Client OTA 数据 2

- (8) Firmware 数据发送完毕后，检查 BLE link layer 的数据是否已经完全发送出去（因为只有当 link layer 的数据被 Server ack 了，才会认为该数据发送成功）。若完全发送出去，Client 发送一个 ota\_end 命令，通知 Server 所有数据已发送完毕。

OTA end 的 packet 有效字节设为 6 个，前两个为 0xff02，中间的两个 bytes 为新的 Firmware 最大的 adr\_index 值（这个是为了让 Server 端再次确认没有丢掉最后一条或几条 OTA 数据），最后两个 bytes 为中间最大的 adr\_index 值的取反，相当于一个简单的校验。OTA end 不需要 CRC 校验。

以上图所示的 bin 为例，最大的 adr\_index 为 0x0cf9，其取反值为 0xf306，最终的 OTA end 包如上图所示。

- (9) 检查 Client 端 link layer TX fifo 是否为空。若为空，说明之前所有的数据和命令都已成功发送出去，即 Client 端的 OTA 任务已经全部完成。CRC\_16 计算函数见本文档后面的“附录 1：crc16 算法”。

按照前面所述，Server 端在 OTA Attribute 中直接调用 otaWrite 和 otaRead 即可，Client 端发送过来的 write command 命令，BLE 协议栈会自动解析并最终调用到 otaWrite 函数进行处理。

在 otaWrite 函数里对 packet 20 byte 的数据进行解析，首先判断是 OTA CMD 还是 OTA data，对 OTA cmd 进行相应的响应，对 OTA 数据进行 CRC 校验并烧写到 flash 对应位置。

Server 端 OTA 相关的操作为：



### (1) 收到 OTA version 命令 (OTA\_FIRMWARE\_VERSION 命令):

Client 要求获得 Server Firmware 版本号，该 BLE SDK 收到这个命令时，不做处理，只是根据用户是否注册了收到 version 的回调函数，判断是否触发回调函数。

在 ota.h 中看到注册该回调函数的接口为：

```
typedef void (*ota_versionCb_t)(void);
void blc_ota_registerOtaFirmwareVersionReqCb(ota_versionCb_t cb);
```

### (2) 收到 OTA start 命令：

此时 Server 进入 OTA 模式。

若用户使用 blc\_ota\_registerStartCmdCb 函数注册了 OTA start 时的回调函数，则执行此函数，这个函数的目的是让用户在进入 OTA 模式后，修改一些参数状态等，比如将 PM 关掉（使得 OTA 数据传输更加稳定）。另外 Server 启动并维护一个 slave\_adr\_index，初值为 -1，记录最近一次正确 OTA data 的 adr\_index，用于判断整个 OTA 过程中是否有丢包。一旦丢包，认为 OTA 失败，退出 OTA，上报结果，MCU 重启，Client 端由于收不到 Server 的 ack 包，也会由于 OTA 任务超时使得软件发现 OTA 失败。

注册 OTA start 的回调函数：

```
typedef void (*ota_startCb_t)(void);
void blc_ota_registerOtaStartCmdCb(ota_startCb_t cb);
```

用户需要注册这个回调，以便在 OTA start 的时候做一些操作，比如配置 LED 灯的特殊闪烁方式来指示 OTA 正在进行。

另外 Server 这端一旦收到 OTA start 开始 OTA 后，也会启动两个计时，一个是 OTA 整个过程完成的超时时间，目前 SDK 中默认是 30s。如果 30s 之内 OTA 还没有完成，就认为 OTA\_TIMEOUT 失败。实际用户最后需要根据自己的 Firmware 大小（越大越耗时）和 Client 端 BLE 数据带宽（太窄的话会影响 OTA 速度）来修改这个默认的 30s，SDK 提供修改的变量为：

```
ble_sts_t blc_ota_setOtaProcessTimeout(int timeout_second);
ble_sts_t blc_ota_setOtaDataPacketTimeout(int timeout_second);
```

函数 blc\_ota\_setOtaProcessTimeout 的参数 timeout\_second 的单位为秒，默认为 30，范围是 5-1000；

函数 blc\_ota\_setOtaDataPacketTimeout 的参数 timeout\_second 的单位为秒，默认为 5，范围是 1-20；

初始化该变量后，用户可调用下面的的超时函数来进行超时判断处理。

```
void blt_ota_procTimeout(void);
```

另外一个是 receive packet 的超时时间，每收到一次 OTA 数据包都会更新一次，超时时间为 5s，即 5s 内没有收到下一笔数据则认为 OTA\_RF\_PACKET\_TIMEOUT 失败。

### (3) 收到有效的 OTA 数据（前两 bytes 为 0~0x1000）：



这个范围的值表示具体的 OTA data。

每次 Server 收到一个 20 byte 的 OTA data packet，先看 adr\_index 是否等于 slave\_adr\_index 的值加 1。若不等，说明丢包，OTA 失败；若相等，更新 slave\_adr\_index 的值。

然后对前 18 byte 的内容进行 CRC\_16 的校验。若不匹配，OTA 失败；若匹配，则将 16 byte 的有效数据写到 flash 对应位置 ota\_program\_offset+adr\_index\*16 ~ ota\_program\_offset+adr\_index\*16 + 15。在写 flash 的过程中，如果出错，OTA 也失败。

为了保证 OTA 完成后 Firmware 的完整性，在最后还会对整个 Firmware 进行 CRC\_32 校验，与 Client 发送过来同样方法计算得到的校验值进行比较，不相等的话说明中间有数据出错，认为 OTA 失败。

#### (4) 收到 OTA end：

检查 OTA end 包中的 adr\_max 和其取反校验值是否正确。若正确，则 adr\_max 可以用来做 double check。double check 的时候，判断 Server 之前收到的 Client 的数据 index 最大值与该包中的 adr\_max 是否相等。若相等，认为 OTA 成功，若不等，认为丢掉了最后一笔或几笔数据，OTA 不完整。当 OTA 成功的时候，Server 将老的 Firmware 所在地址的 flash 启动标志设为 0，将新的 Firmware 所在地址的 flash 启动标志设为 0x4b，将 MCU reboot。

#### (5) Server 发送 OTA 结果反馈给 Client：

Server 端一旦启动 OTA，不管 OTA 是成功还是失败，最后 Server 都会将结果发送给 Client。如下是 OTA 成功后 Server 发送结果信息示例（长度只有 3 个 byte）：

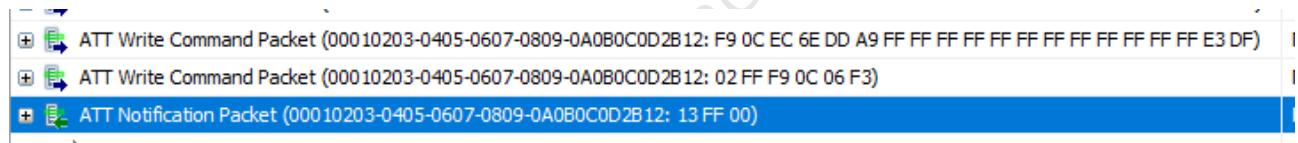


Figure 7.13: Server 将 OTA 成功的结果发送给 Client

#### (6) Server 提供 OTA 状态的回调函数：

Server 端一旦启动 OTA，在 OTA 成功时会将 MCU reboot：

若成功，会在 reboot 前设置 flag 告诉 MCU 再次启动后运行 New\_firmware；

若 OTA 失败，会将错误的新程序擦掉后重新启动，还是运行 Old\_firmware。

在 MCU reboot 前，根据用户是否注册了 OTA 状态回调函数，来决定是否触发该函数。

以下是相关 code：

```
void blc_ota_registerOtaResultIndicationCb (ota_resIndicateCb_t cb);
```

设置了回调函数后，回调函数的参数 result 的 enum 与 OTA 上报的结果一样，第一个 0 是 OTA 成功，其余是不同的失败原因。

OTA 升级成功或失败均会触发该回调函数，实际代码中可以通过该函数的结果返回参数来进行 debug，在 OTA 不成功时，可以读到上面的 result 后，将 MCU 用 while(1) 停住，来了解当前是何种原因导致的 OTA 失败。

注意：



- 1) 如果固件大小超过 256KB，需在 sys\_init() 前调用 API blc\_ota\_setFirmwareSizeAndBootAddress 设置固件大小和多地址启动地址。
- 2) 如果当前程序使能 Flash protection 功能，则不可升级为没有使能 Flash protection 的程序，因为 Flash 已经被加锁，更新后的程序没有解锁操作。

## 7.4 Q&A

- Q: 在编译完成时，从编译日志中看到 bin size is greater 256KB, please refer to handbook!

OTA 功能与多地址启动功能相关，Telink B91 系列芯片的默认启动地址是 0 地址和 256 KB 偏移地址，因此，当编译生成的 Bin Size 超过 256 KB 且需要支持 OTA 功能时，应进行以下配置：

1. 在 main 函数的开头 (`sys_init` 之前)，调用 `blc_ota_setFirmwareSizeAndBootAddress` 来设置**最大固件大小**和多地址启动地址；
2. 关注 `.link` 文件中的 `BIN_SIZE`（即实际的固件大小），避免超过设置的**最大固件大小**。
3. 若 `BIN_SIZE` 超过**最大固件大小**，则应根据用户实际使用的 Flash 和 Flash 规划进行调整，避免出现内存覆盖问题。



## 8 按键扫描

Telink 提供了一套基于行列式扫描的 keyscan 架构，用于按键扫描处理，使用者可以直接使用，也可以自己去实现。

### 8.1 键盘矩阵

定义 drive pin 数组和 scan pin 数组：

```
#define KB_DRIVE_PINS {GPIO_PB3, GPIO_PB5}  
#define KB_SCAN_PINS {GPIO_PB6, GPIO_PB7}
```

keyscan 使用的上下拉电阻都使用 GPIO 的模拟电阻：drive pin 选取下拉 100K，scan pin 选取上拉 10K。那么当没有按键按下时，scan pin 作为输入 GPIO 读到的是被 10K 上拉的高电平。当扫描开始时，在 drive pin 上输出低电平，scan pin 读到低电平，就表示当前列上有按键按下（注意此时 drive pin 不是 float 态，若 output 没打开，scan pin 读到的是 100K 和 10K 的分压电平，还是高）。

定义行列式扫描中，drive pin 输出低电平时 scan pin 扫描到的有效电平。

```
#define KB_LINE_HIGH_VALID 0
```

定义 Row 和 COL 的上下拉：

```
#define MATRIX_ROW_PULL      PM_PIN_PULLDOWN_100K  
#define MATRIX_COL_PULL      PM_PIN_PULLUP_10K  
  
//A<3:0>  
analog_write_reg8(0x17, PULL_WAKEUP_SRC_PA0 | (PULL_WAKEUP_SRC_PA1 << 2) | (PULL_WAKEUP_SRC_PA2  
  << 4) | (PULL_WAKEUP_SRC_PA3 << 6));  
//A<7:4>  
analog_write_reg8(0x18, PULL_WAKEUP_SRC_PA4 | (PULL_WAKEUP_SRC_PA5 << 2) | (PULL_WAKEUP_SRC_PA6  
  << 4) | (PULL_WAKEUP_SRC_PA7 << 6));  
//B<3:0>  
analog_write_reg8(0x19, PULL_WAKEUP_SRC_PB0 | (PULL_WAKEUP_SRC_PB1 << 2) | (PULL_WAKEUP_SRC_PB2  
  << 4) | (PULL_WAKEUP_SRC_PB3 << 6));  
//B<7:4>  
analog_write_reg8(0x1a, PULL_WAKEUP_SRC_PB4 | (PULL_WAKEUP_SRC_PB5 << 2) | (PULL_WAKEUP_SRC_PB6  
  << 4) | (PULL_WAKEUP_SRC_PB7 << 6));  
  
//C<3:0>  
analog_write_reg8(0x1b, PULL_WAKEUP_SRC_PC0 | (PULL_WAKEUP_SRC_PC1 << 2) | (PULL_WAKEUP_SRC_PC2  
  << 4) | (PULL_WAKEUP_SRC_PC3 << 6));  
//C<7:4>  
analog_write_reg8(0x1c, PULL_WAKEUP_SRC_PC4 | (PULL_WAKEUP_SRC_PC5 << 2) | (PULL_WAKEUP_SRC_PC6  
  << 4) | (PULL_WAKEUP_SRC_PC7 << 6));
```



```
//D<3:0>
analog_write_reg8(0x1d, PULL_WAKEUP_SRC_PD0 | (PULL_WAKEUP_SRC_PD1 << 2) | (PULL_WAKEUP_SRC_PD2
    << 4) | (PULL_WAKEUP_SRC_PD3 << 6));
//D<7:4>
analog_write_reg8(0x1e, PULL_WAKEUP_SRC_PD4 | (PULL_WAKEUP_SRC_PD5 << 2) | (PULL_WAKEUP_SRC_PD6
    << 4) | (PULL_WAKEUP_SRC_PD7 << 6));
//E<3:0>
analog_write_reg8(0x1f, PULL_WAKEUP_SRC_PE0 | (PULL_WAKEUP_SRC_PE1 << 2) | (PULL_WAKEUP_SRC_PE2
    << 4) | (PULL_WAKEUP_SRC_PE3 << 6));
//E<7:4>
analog_write_reg8(0x20, PULL_WAKEUP_SRC_PE4 | (PULL_WAKEUP_SRC_PE5 << 2) | (PULL_WAKEUP_SRC_PE6
    << 4) | (PULL_WAKEUP_SRC_PE7 << 6));
```

由于在 gpio\_init 时将 ie 的状态会默认设为 0，scan pin 需要读电平，打开 ie：

```
#define PB1_INPUT_ENABLE      1
#define PB0_INPUT_ENABLE      1
#define PA4_INPUT_ENABLE       1
#define PA0_INPUT_ENABLE       1
#define PE6_INPUT_ENABLE       1
#define PE5_INPUT_ENABLE       1
```

当 MCU 进入 sleep mode 时，需要设置 PAD GPIO 唤醒。设置 drive pin 高电平唤醒，按下按键时，drive pin 读到 100K 和 10K 的分压电平，为 10/11 VCC 的高电平。需要打开 drive pin 的 ie 读取其电平状态：

```
#define PE2_INPUT_ENABLE      1
#define PB4_INPUT_ENABLE       1
#define PB5_INPUT_ENABLE       1
#define PE1_INPUT_ENABLE       1
#define PE4_INPUT_ENABLE       1
```

## 8.2 Keypress and Keymap

### 8.2.1 Keypress

按照上面的配置完成后，在 main\_loop 的 proc\_keyboard 函数中调用下面函数完成 keyscan。

```
u32 kb_scan_key (int numlock_status, int read_key)
```

第一个参数 numlock\_status 在 proc\_keyboard 中调用时设为 0 即可；只有在 deepsleep 醒来的快速扫描按键时才会将其设为 KB\_NUMLOCK\_STATUS\_POWERON，后面的快速扫键中介绍（对应 DEEPBACK\_FAST\_KEYSCAN\_ENABLE）。



第二个参数 read\_key 是 keyscan 函数按键的缓存处理，这个一般用不到，一直设为 1 即可（为 0 时会将按键值缓存在 buffer 里，不报告给上层）。

返回值用于通知 user 当前的按键扫描是否发现矩阵键盘有变化：有变化时，返回 1；无变化时，返回 0。

kb\_scan\_key 这个函数是在 proc\_keyboard 中调用的，根据 BLE 时序可知，main\_loop 的运行时间为 adv\_interval 或 conn\_interval。广播状态时（假设 adv\_interval 为 30ms），每 30ms 做一次 key scan；连接状态时（假设 conn\_interval = 10ms），每 10ms 做一次 key scan。

理论上，当前 key scan 发现矩阵上按键的状态和上次 key scan 的状态不一样时，就认为有变化。

实际代码中开启了一个防抖动滤波处理：只有发现连续两次 key scan 的按键状态一样，且和上一次存储的最新矩阵按键状态不一样时，才认为是一个有效的按键变化。这时返回 1 表示按键有变化，并将矩阵按键的状态通过 kb\_event 结构体反映出来，同时将更新当前的按键状态为最新的矩阵按键状态。这部分对应的代码为 keyboard.c 中的：

```
unsigned int key_debounce_filter( u32 mtrx_cur[], u32 filt_en );
```

上面所说的最新按键状态指的是矩阵上所有按键的按下或松开的状态的集合。上电时，默认的第一次矩阵按键状态为所有按键都是松开的。只要经过防抖动滤波处理后的矩阵按键的状态发生任何变化，返回值都会为 1，否则返回 0。如：按下一个按键返回一个变化；松开一个按键返回一个变化；按下一个键时再按下第二个键返回一个变化；按下两个键时再按下第三个键返回一个变化；按下两个键时松开其中一个键返回一个变化……

### 8.2.2 Keymap & kb\_event

user 在调用 kb\_scan\_key 看到一个按键变化时，通过一个全局的结构体变量 kb\_event 来获取当前的按键状态。

```
#define KB_RETURN_KEY_MAX    6
typedef struct{
    u8 cnt;
    u8 ctrl_key;
    u8 keycode[KB_RETURN_KEY_MAX];
}kb_data_t;
kb_data_t    kb_event;
```

kb\_event 由 8 个 byte 构成：

第一个 cnt 用于指示当前有几个有效的按键被按下；

第二个 ctrl\_key 一般不会用到，只有在做标准的 USB HID keyboard 时才会用到（keymap 中的 keycode 设为 0xe0-0xe7 时会触发，所以 user 千万不要设这 8 个值）。

keycode[6] 用于最多存储当前 6 个被按下按键的 keycode（如果实际按下的键超过 6 个，只有前 6 个能反应出来）。

所有按键对应的 keycode 在 app\_config.h 中定义：



```
#define KB_MAP_NORMAL {      |
    {BTN_UNPAIR, BTN_PAIR }, |
    {CR_VOL_UP, CR_VOL_DN}, |
}
```

在 kb\_scan\_key 函数内部，每次扫描前会将 kb\_event.cnt 清 0，而 kb\_event.keycode[] 这个数组是不清除的。所以每次返回 1 表示有变化时，用 kb\_event.cnt 判断当前矩阵按键上有几个有效的按键。

- a) kb\_event.cnt = 0 时，上一次有效矩阵状态 kb\_event.cnt 肯定是不等于 0 的，但不确定是 1、2 还是 3，这个变化一定是按键释放，不确定是一个键释放还是同时好几个键释放。此时 kb\_event.keycode[] 里面即使有数据，也是无效的，忽略不看。
- b) kb\_event.cnt = 1，可能上次 kb\_event.cnt = 0，那么按键变化是一个键被按下；可能上次 kb\_event.cnt = 2，那么按键变化是两个键中一个被释放；也还有其他可能性，如三个键中两个被同时释放。此时 kb\_event.keycode[0] 表示当前被按下的这个键的键值，后面的 keycode 忽略不看。

user 可以每次在 key scan 前自己将 kb\_event.keycode 清 0，这时就可以根据 kb\_event.keycode 来判断是否有按键变化发生，如下所示。

这个示例只是简单的处理单个按键按下的情况，所以当 kb\_event.keycode[0] 非 0 时，就认为是一个按键被按下，并不去判断是否两个键同时按下或者两个键中的一个释放等复杂的情况。

```
int det_key          = kb_scan_key(0, 1);

if (det_key) {
    key_change_proc();
}
```

## 8.3 Keypress Flow

调用 kb\_scan\_key 时，一个最基本的 keyscan 的流程如下：

- (1) 第一次全矩阵通扫。

将 drive pin 全部输出 drive 电平 (0)，同时读取所有的 scan pin，检查是否能读到有效的电平，并记录哪一列上读到了有效电平（用 scan\_pin\_need 标记有效的列号）。

若不使用第一次全矩阵通扫，直接逐行扫的话，至少要进行所有行的扫描，即使没有按键按下也要每次都逐行扫描，比较耗时间。加入了第一次全矩阵通扫后，若没发现任何列上有按键按下，就可以直接退出 keyscan，在没有按键按下时会节省很多时间。

第一次全矩阵通扫的 code 对应如下：

```
scan_pin_need = kb_key_pressed (gpio);
```

在 kb\_key\_pressed 函数中将所有的行输出低电平，延时 20us 后（延时是为了等待电平稳定后才读 scan pin）。设置了一个 release\_cnt 为 6，当检测到矩阵上的按键按下并全部释放后，并不是立刻就认为没有按键而去逐行扫描了，而是最终缓冲 6 帧，直到发现连续 6 次都是检测到按键全部释放后不再去逐行扫描。实现了一个 key debouce 防抖动的处理。



### (2) 根据全矩阵通扫的结果，逐行扫描。

全矩阵通扫发现有按键按下时，开始逐行扫描，从 ROW0 ~ ROW4 逐行输出有效 drive 电平，读取列上的电平值，找出按键按下的位置。

对应的代码为：

```
unsigned int pressed_matrix[ARRAY_SIZE(drive_pins)] = {0};  
#if (KB_STANDARD_KEYBOARD)  
    kb_k_mp = kb_p_map[0];  
#else  
    kb_k_mp = (kb_k_mp_t *)&kb_map_normal[0];  
#endif  
  
kb_scan_row(0, gpio);  
for (unsigned int i = 0; i <= ARRAY_SIZE(drive_pins); i++) {  
    unsigned int r = kb_scan_row(i < ARRAY_SIZE(drive_pins) ? i : 0, gpio);  
    if (i) {  
        pressed_matrix[i - 1] = r;  
    }  
}
```

在做逐行扫描时使用了一些方法来优化代码执行时间：

一、当对某行 drive 进行扫描时，并不需要读取全部的列 Col0 ~ Col5，根据之前通扫的 scan\_pin\_need 可以知道哪些列上能够读到有效电平，此时只读取已经被标记的列即可。

二、当对每一行 drive 进行扫描时时，需要 20us 左右的等待稳定时间，做了一个缓冲处理，把 20us 的等待时间转化到执行 code 中，节省了这个时间。具体怎么实现不介绍，请 user 自行理解。

最终的矩阵按键状态使用 u32 pressed\_matrix[5] (可看出最多支持 40 列) 来存储，pressed\_matrix[0] 的 bit0~bit5 标记 Row0 上 Col0 ~ Col5 是否有按键，……，pressed\_matrix[4] 的 bit0~bit5 标记 Row4 上 Col0 ~ Col5 是否有按键。

### (3) 对 pressed\_matrix[] 进行防抖动滤波处理

对应代码为：

```
unsigned int key_debounce_filter( u32 mtrc_cur[], u32 filt_en);  
u32 key_changed = key_debounce_filter( pressed_matrix, (numlock_status &  
    KB_NUMLOCK_STATUS_POWERON) ? 0 : 1);
```

当 deepsleep 醒来后快速按键检测时，numlock\_status = KB\_NUMLOCK\_STATUS\_POWERON，此时 filt\_en = 0，不进行滤波，是为了最快速的获取键值。

其他情况下，filt\_en = 1，需要滤波处理。滤波处理的思路是：最近的连续两次 pressed\_matrix[] 一致，且和上一次有效的 pressed\_matrix[] 不一样，才认为是按键矩阵发生了有效的变化，key\_changed = 1。

### (4) 对 pressed\_matrix[] 进行缓存处理



将 pressed\_matrix[] 存入到缓冲区，当 kb\_scan\_key (int numlock\_status, int read\_key) 中的 read\_key 为 1 时，立刻读出缓冲区的数据，当 read\_key 为 0 时，缓冲区数据保存起来，不通知上层，只有等到 read\_key 为 1 时，才能读出之前缓存的数据。

由于我们的 read\_key 永远为 1，这部分可以忽略不计，相当于缓冲区没有起到作用。具体代码不介绍。

(5) 根据 pressed\_matrix[], 查表 KB\_MAP\_NORMAL，返回键值。

对应的函数为 kb\_remap\_key\_code 和 kb\_remap\_key\_row，不具体介绍，user 自行理解。

## 8.4 Repeat Key 处理

以上介绍的最基本的 keyscan 只有在按键状态变化时产生一个变化事件，通过 kb\_event 来读取当前 key 值，就无法实现 repeat key 功能：一个按键一直按着时，需要定时发送一个按键值。

加入 repeat key 处理，在 keyboard.c 中配置相关的宏如下。KB\_REPEAT\_KEY\_ENABLE 用来打开或关闭 repeat key 功能，默认这个功能是关闭的。

```
#ifndef KB_REPEAT_KEY_ENABLE
#define KB_REPEAT_KEY_ENABLE 0
#endif
#ifndef KB_REPEAT_KEY_INTERVAL_MS
#define KB_REPEAT_KEY_INTERVAL_MS 200
#endif
#ifndef KB_REPEAT_KEY_NUM
#define KB_REPEAT_KEY_NUM 4
#endif
```

(1) KB\_REPEAT\_KEY\_ENABLE

用来打开或关闭 repeat key 功能。

若要实现 repeat key，首先要将 KB\_REPEAT\_KEY\_ENABLE 设为 1。

(2) KB\_REPEAT\_KEY\_INTERVAL\_MS

定义 repeat key 的 repeat 时间。

若设为 200ms，表示当一个键被一直按着时，每过 200 ms，kb\_key\_scan 会返回一个变化，并且在 kb\_event 里面给出当前这个按键状态。

(3) KB\_REPEAT\_KEY\_NUM & KB\_MAP\_REPEAT

定义当前需要 repeat 的键值。

KB\_REPEAT\_KEY\_NUM 定义数量；KB\_MAP\_REPEAT 定义一个 map，给出所有需要 repeat 的 keycode，注意这个 map 中 keycode 一定要是 KB\_MAP\_NORMAL 中的值。

应用举例：

如下所示的一个 6\*6 的矩阵按键，四个宏定义实现的功能是：8 个按键 UP、DOWN、LEFT、RIGHT、V+、V-、CHN+、CHN-支持 repeat，每 100ms repeat 一次；其他的按键都不支持 repeat key。



```
#define KB_MAP_NORMAL { \
    {VK_POWER,          VK_LOW_BATT,   VK_TV_PLUS,     VK_TV_MINUS,      VK_IN_OUTPUT,  VK_VOL_UP, }, \
    {VK_VOICE_SEARCH,  VK_PROGRAM,   VK_RETURN,      VK_HOME,        VK_MENU,       VK_EXIT, }, \
    {VK_UP,             VK_CH_UP,     VK_W_MUTE,     VK_LEFT,        VK_CONFIRM,   VK_RIGHT, }, \
    {VK_VOL_DN,         VK_DOWN,      VK_CH_DN,      VK_FAST_BACKWARD, VK_PLAY_PAUSE, VK_1, }, \
    {VK_2,              VK_3,         VK_4,           VK_5,           VK_6,         VK_7, }, \
    {VK_9,              VKPAD_ASTERIX, VK_0,          VK_NUMBER,      VK_W_SRCH,    VK_8, } }

#define KB_REPEAT_KEY_ENABLE 1
#define KB_REPEAT_KEY_INTERVAL_MS 100
#define KB_REPEAT_KEY_NUM 8
#define KB_MAP_REPEAT { VK_UP,           VK_DOWN,      VK_LEFT,      VK_RIGHT, \
    VK_VOL_UP,   VK_VOL_DN,   VK_CH_UP,   VK_CH_DN, }
```

Figure 8.1: Repeat key 应用举例

repeat key 代码的实现这里不介绍，user 自行理解，只要在工程上搜索以上四个宏就可以找到所有代码了。



## 9 LED 管理

### 9.1 LED 任务相关调用函数

该 BLE SDK 提供了一个 led 管理的参考代码，以源码提供，user 可以直接使用这部分的 code 或参考其实现方法自己设计。代码在 vendor/common/blt\_led.c 中，user 在自己的 C file 中 include vendor/common/blt\_led.h 即可。若需要使用，先将下面宏改为 1：

```
#define BLT_APP_LED_ENABLE 0
```

user 需要调用的三个函数为：

```
void device_led_init(u32 gpio,u8 polarity);
int device_led_setup(led_cfg_t led_cfg);
static inline void device_led_process(void);
```

在初始化的时候使用 device\_led\_init(u32 gpio,u8 polarity) 设置当前 LED 对应的 GPIO 和极性。极性设为 1，表示 gpio 输出高电平点亮 LED；极性设为 0，表示低电平点亮 LED。

在 main\_loop 的 UI Entry 部分添加 device\_led\_process 函数，该函数每次检查是否有 LED 任务没有完成 (DEVICE\_LED\_BUSY)，若有任务，去执行相应的操作。

## 9.2 LED 任务的配置和管理

### 9.2.1 定义 led event

使用如下结构体定义一个 led event：

```
typedef struct{
    unsigned short onTime_ms;
    unsigned short offTime_ms;
    unsigned char repeatCount;
    unsigned char priority;
} led_cfg_t;
```

onTime\_ms 和 offTime\_ms 表示当前的 led event 保持亮起的时间 (ms) 和熄灭的时间 (ms)。注意它们是用 unsigned short 定义的，最大 65535。

repeatCount 表示 onTime\_ms 和 offTime\_ms 定义的一亮一灭的动作持续重复多少次。注意它是用 unsigned char 定义的，最大 255。

priority 表示当前 led event 的优先级。

当我们要定义一个长亮或长灭的 led event 时（没有时间限制，也就是 repeatCount 不起作用），将 repeatCount 的值设为 255(0xff)，此时 onTime\_ms 和 offTime\_ms 里面必须一个是 0，一个非 0，根据非 0 来判断是长亮还是长灭。

以下为几个 led event 的示例：



(1) 1 Hz 的频率闪烁 3 秒：亮 500ms，灭 500ms，repeat 3 次。

```
led_cfg_t led_event1 = {500, 500, 3, 0x00};
```

(2) 4 Hz 的频率闪烁 50 秒：亮 125ms，灭 125ms，repeat 200 次。

```
led_cfg_t led_event2 = {125, 125, 200, 0x00};
```

(3) 长亮：onTime\_ms 非 0，offTime\_ms 为 0，repeatCount 为 0xff。

```
led_cfg_t led_event3 = {100, 0, 0xff, 0x00};
```

(4) 长灭：onTime\_ms 为 0，offTime\_ms 非 0，repeatCount 为 0xff。

```
led_cfg_t led_event4 = {0, 100, 0xff, 0x00};
```

(5) 亮 3 秒后熄灭：onTime\_ms 为 1000，offTime\_ms 为 0，repeatCount 为 0x3。

```
led_cfg_t led_event5 = {1000, 0, 3, 0x00};
```

调用 device\_led\_setup 将一个 led\_event 送给 led 任务管理：

```
device_led_setup(led_event1);
```

### 9.2.2 LED Event 的优先级

user 可以在 SDK 里定义多个 led event，LED 在一个时间点只能执行一个 led event。

这个简单的 led 管理没有设置任务列表，当 led 空闲时，led 接受 user 调用 device\_led\_setup 建立的任何 led event；当 led busy 时（前一个 old led event 还没有结束），对于 new led event，对两个 led event 的优先级进行比较。若 new led event 的优先级高于 old led event 的优先级，将 old led event 抛弃，开始执行 new led event；若 new led event 的优先级低于或等于 old led event 的优先级，继续执行 old led event，将 new led event 抛弃（注意：是彻底抛弃，并不会将这个 led event 缓存起来后面再处理）。

user 可以根据以上的 led event 优先级的原则，在自己的应用里定义不同优先级的 led event，实现自己的 led 指示效果。

另外，由于 led 的管理采用了查询的机制，当 DEVICE\_LED\_BUSY 时，不能进入 latency 生效时的 long suspend，如果进入了一个 long suspend（比如  $10\text{ms} * 50 = 500\text{ms}$ ），会导致 onTime\_ms 较小的值（如 250ms）无法得到及时响应，从而影响了 LED 闪烁的效果。

```
#define DEVICE_LED_BUSY (device_led.repeatCount)
```

针对以上问题，需要在 blt\_pm\_proc 中作相应的处理：



```
int user_task_flg = scan_pin_need || key_not_released || DEVICE_LED_BUSY;  
if(user_task_flg){  
    ...  
    bls_pm_setManualLatency(0); // manually disable latency  
    ...  
}
```

Telink Semiconductor



## 10 软件定时器 (Software Timer)

为了方便 user 做一些简单的定时器任务，Telink BLE SDK 提供了 blt software timer demo，并且全部源码提供。user 可以在理解了该 timer 的设计思路后直接使用，也可以自己做一些修改设计。

源代码全部在 vendor/common/blt\_soft\_timer.c 和 blt\_soft\_timer.h 文件中，若需要使用，先将下面宏改为 1：

```
#define BLT_SOFTWARE_TIMER_ENABLE 0 //enable or disable
```

blt soft timer 是基于 system tick 设计的查询式 timer，其准确度无法达到硬件 timer 那么准，且需要保证在 main\_loop 中一直被查询。

我们约定：blt soft timer 的使用场景为定时时间大于 5ms、且对于时间误差要求不是特别高的情况。

blt soft timer 的最大特点是不仅在 main\_loop 中会被查询，也能确保在进入 suspend 后能够及时唤醒并执行 timer 的任务，该设计是基于低功耗唤醒部分介绍的“应用层定时唤醒”实现的。

目前设计上最多同时支持 4 个 timer 运行，实际 user 可以修改下面的宏来实现更多或者更少的 timer：

```
#define MAX_TIMER_NUM 4 //timer max number
```

### 10.1 Timer 初始化

调用下面的 API 进行初始化：

```
void blt_soft_timer_init(void);
```

可以看到源码上初始化只是将 blt\_soft\_timer\_process 注册为应用层提前唤醒的回调函数。

```
void blt_soft_timer_init(void)
{
    bls_pm_registerAppWakeupLowPowerCb(blt_soft_timer_process);
}
```

### 10.2 Timer 的查询处理

blt soft timer 的查询处理使用 blt\_soft\_timer\_process 函数来实现：

```
void blt_soft_timer_process(int type);
```

一方面需要确保在 main\_loop 中如下所示位置一直被调用；另一方面被注册为应用层提前唤醒的回调函数，那么每次在 suspend 中发生定时提前唤醒时，也会快速执行该函数，去处理各 timer 任务。



```
int main_idle_loop(void)
{
    blc_sdk_main_loop();
#if (FEATURE_TEST_MODE == TEST_USER_BLT_SOFT_TIMER)
    blt_soft_timer_process(MAINLOOP_ENTRY);
#endif
}
```

blt\_soft\_timer\_process 的参数中 type 有如下两种情况: 0 表示在 main\_loop 中查询进入, 1 表示发生了 timer 提前唤醒时进入该函数。

#define	MAINLOOP_ENTRY	0
#define	CALLBACK_ENTRY	1

blt\_soft\_timer\_process 的具体实现比较复杂, 基本思路如下:

- (1) 首先检查当前 timer table 中是否还有 user 定义的 timer: 若没有则直接退出, 并关掉应用层定时唤醒;若有 timer 任务, 继续往下运行。

```
if(!blt_timer.currentNum){
    bls_pm_setAppWakeupLowPower(0, 0); //disable
    return;
}
```

- (2) 检查时间上最近的一个 timer 任务是否到达: 若没有到达, 则退出, 否则继续往下运行。设计上会保证 timer 在任何时候都是按照时间排序的, 所以这里只要看时间上最近的 timer 即可。

```
if( !blt_is_timer_expired(blt_timer.timer[0].t, now) ){
    return;
}
```

- (3) 轮询当前所有的 timer 任务, 只要时间达到了就执行 timer 对应的任务。

```
for(int i=0; i<blt_timer.currentNum; i++){
    if(blt_is_timer_expired(blt_timer.timer[i].t ,now) ){ //timer trigger
        if(blt_timer.timer[i].cb == NULL){
        }
        else{
            result = blt_timer.timer[i].cb();
            if(result < 0){
                blt_soft_timer_delete_by_index(i);
            }
            else if(result == 0){
                change_flg = 1;
                blt_timer.timer[i].t = now + blt_timer.timer[i].interval;
            }
        }
    }
}
```



```
        }
    else{ //set new timer interval
        change_flg = 1;
        blt_timer.timer[i].interval = result * SYSTEM_TIMER_1US;
        blt_timer.timer[i].t = now + blt_timer.timer[i].interval;
    }
}
}
```

这里面可以看到对 timer 任务函数的处理：若该函数返回值小于 0，这个 timer 任务会被删掉，后面不再响应；若返回值为 0，则保持上一次的定时值；若返回值大于 0，则以该返回值作为新的定时周期（单位 us）。

- (4) 在上面的第 3 步中，如果 timer 任务表中的任务发生了变化，则之前的时间顺序可能会被破坏，这里再重新排序。

```
if(change_flg){
    blt_soft_timer_sort();
}
```

- (5) 若最近的 timer 任务的响应时间距离现在只剩 3 秒（3s 可以再改大一些）不到，则将该时间设为应用层提前唤醒的时间，否则关闭应用层提前唤醒。

```
if( (u32)(blt_timer.timer[0].t - now) < 3000 * SYSTEM_TIMER_1MS){
    bls_pm_setAppWakeupLowPower(blt_timer.timer[0].t, 1);
}
else{
    bls_pm_setAppWakeupLowPower(0, 0); //disable
}
```

## 10.3 添加定时器任务

使用如下 API 添加定时器任务：

```
typedef int (*blt_timer_callback_t)(void);
int     blt_soft_timer_add(blt_timer_callback_t func, u32 interval_us);
```

func 为定期执行的任务函数；interval\_us 为定时时间，单位为 us。

定时任务 func 的 int 返回值三种处理方法：

- a) 返回值小于 0，则该任务执行后被自动删除。可以使用这个特性来控制定时器执行的次数。
- b) 返回 0，则一直使用之前的 interval\_us 来定时。
- c) 返回值大于 0，则使用该返回值作为新的定时周期，单位 us。



```
int blt_soft_timer_add(blt_timer_callback_t func, u32 interval_us)
{
    u32 now = clock_time();
    if(blt_timer.currentNum >= MAX_TIMER_NUM){ //timer full
        return 0;
    }
    else{
        blt_timer.timer[blt_timer.currentNum].cb = func;
        blt_timer.timer[blt_timer.currentNum].interval = interval_us * SYSTEM_TIMER_TICK_1US;
        blt_timer.timer[blt_timer.currentNum].t = now +
        ← blt_timer.timer[blt_timer.currentNum].interval;
        blt_timer.currentNum++;
        blt_soft_timer_sort();
        bls_pm_setAppWakeupLowPower(blt_timer.timer[0].t, 1);
        return 1;
    }
}
```

代码实现中，可以看到当定时器数量超过最大值时，添加失败。每添加一个新的 timer 任务，必须重新做一下排序，以确保定时器任务在任何时候都是按照时间排序的，时间上最近的那个 timer 任务对应的 index 为 0。

## 10.4 删除定时器任务

除了使用上面返回值小于 0 来自动删除定时器任务，还可以使用下面 API 来指定要删除的定时器任务。

```
int blt_soft_timer_delete(blt_timer_callback_t func);
```

## 10.5 Demo

blt soft timer 的 Demo code 请参考 B91m feature 中 TEST\_USER\_BLT\_SOFT\_TIMER。

```
int gpio_test0(void)
{
    //GPIO toggle to see the effect
    gpio_toggle(GPIO_LED_BLUE);
    return 0;
}

_attribute_ble_data_retention_ static u8 timer_change_flg = 0;

int gpio_test1(void)
{
    //GPIO toggle to see the effect
```



```
gpio_toggle(GPIO_LED_GREEN);
timer_change_flg = !timer_change_flg;
if (timer_change_flg) {
    return 7000;
} else {
    return 17000;
}

int gpio_test2(void)
{
    //GPIO toggle to see the effect
    gpio_toggle(GPIO_LED_WHITE);

    //timer last for 5 second
    if (clock_time_exceed(0, 5000000)) {
        //return -1;
        //blt_soft_timer_delete(&gpio_test2);
    } else {
    }

    return 0;
}

int gpio_test3(void)
{
    //GPIO toggle to see the effect
    gpio_toggle(GPIO_LED_RED);
    ;

    return 0;
}
```

初始化:

```
#if (BLT_SOFTWARE_TIMER_ENABLE)
blt_soft_timer_init();
blt_soft_timer_add(&gpio_test0, 23000); //23ms
blt_soft_timer_add(&gpio_test1, 7000); //7ms <-> 17ms
blt_soft_timer_add(&gpio_test2, 13000); //13ms
blt_soft_timer_add(&gpio_test3, 100000); //100ms
#endif
```

定义了 4 个任务，这 4 个定时任务各有特点：

- (1) gpio\_test0 每 23ms toggle 一次。



- (2) gpio\_test1 使用了 7ms/17ms 两个时间的切换定时。
- (3) gpio\_test2 在 5s 后将自己删掉。代码中有两种方式可以实现这个功能：一是调用 blt\_soft\_timer\_delete(&gpio\_test2)；二是 return -1。
- (4) gpio\_test3 每 100ms toggle 一次。



## 11 功能参考 Demo

本章节将介绍 sdk/vendor/feature\_test 下的各个功能的使用方法与现象，用户可以参考代码实现将所需功能添加到自己的代码中。

### 11.1 feature\_backup

- 功能：BLE 基本功能的演示。包括广播、被动扫描、连接等。同时，该功能演示也可作为用户开发 BLE 应用的相对“干净”的基础版本（默认关闭 SMP，SDP 等功能）。
- 主要硬件：B91 开发板 x 2

以 B91 为例，应用层代码在 tl\_ble\_sdk/vendor/feature\_test/feature\_backup 下，需要修改 tl\_ble\_sdk/vendor/feature\_test/feature\_config.h 中的定义：

```
#define FEATURE_TEST_MODE           TEST_FEATURE_BACKUP
```

来激活这部分代码。代码中设置广播参数、广播内容、扫描响应内容、扫描参数，以及使能广播、使能扫描等配置，请参考初始化代码中的如下内容：

```
/////////// User Configuration for BLE application ///////////
blc_ll_setAdvData( (u8 *)tbl_advData, sizeof(tbl_advData) );
blc_ll_setScanRspData( (u8 *)tbl_scanRsp, sizeof(tbl_scanRsp));
blc_ll_setAdvParam(ADV_INTERVAL_30MS, ADV_INTERVAL_30MS, ADV_TYPE_CONNECTABLE_UNDIRECTED, OWN_ADDRESS_PUBLIC, 0, NULL, BLT_ENABLE_ADV_ALL, ADV_FP_NONE);
blc_ll_setAdvEnable(BLC_ADV_ENABLE); //ADV enable
blc_ll_setScanParameter(SCAN_TYPE_PASSIVE, SCAN_INTERVAL_100MS, SCAN_WINDOW_100MS, OWN_ADDRESS_PUBLIC, SCAN_FP_ALLOW_ADV_ANY);
blc_ll_setScanEnable (BLC_SCAN_ENABLE, DUP_FILTER_DISABLE);
}
```

Figure 11.1: 初始化中配置广播扫描参数

编译，将生成的固件分别烧录到两个开发板中。接电启动后，可通过扫描广播包，扫描到两个设备名为“feature”的广播，可以借由其他 Central 设备或者 Peripheral 设备分别进行连接，也可以使二者互联。要使二者互联，按其中一个开发板的按键 SW4 启动连接，通过 BDT 工具的 Tdebug 选项卡，读取左侧的变量值列表（具体使用方法请参考[“Debug 方法”章节](#)），可以看到该开发板的 acl\_conn\_central\_num 的值为 1：

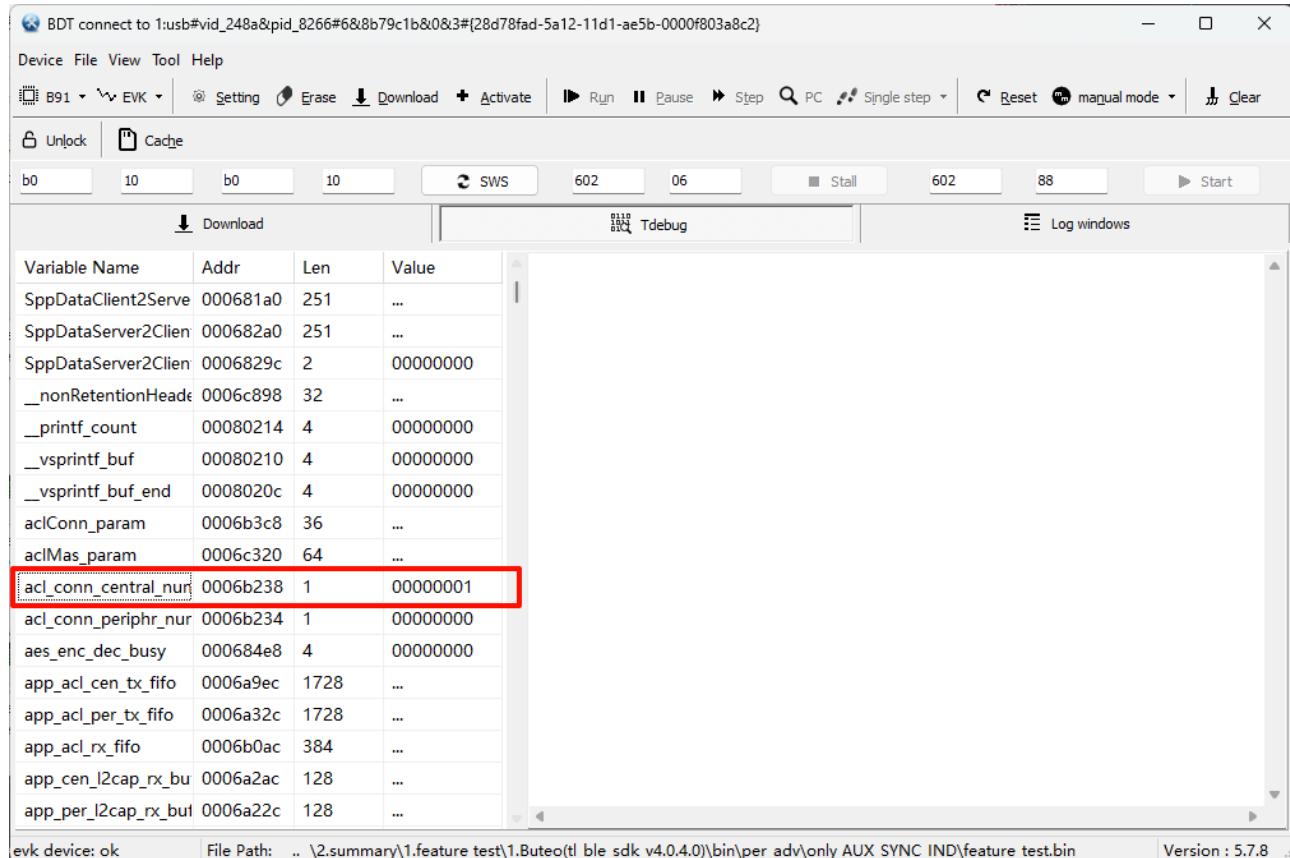


Figure 11.2: acl\_conn\_central\_num=1 in Tdebug

查看另一个开发板的 acl\_conn\_periphrr\_num 变量的值，其值也为 1：

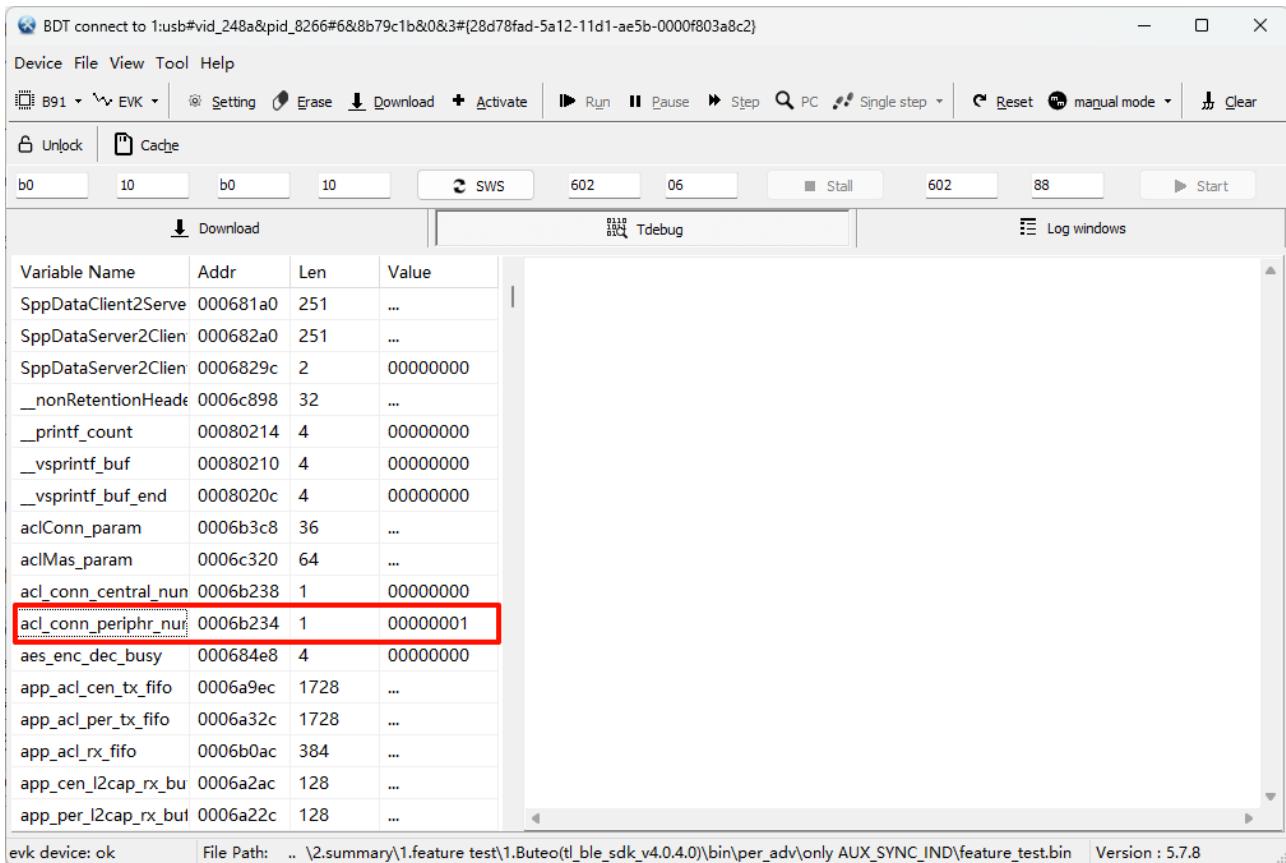


Figure 11.3: acl\_conn\_periph\_num=1 in Tdebug

代表二者连接成功。

## 11.2 feature\_2M\_coded\_phy

- 功能：BLE 1M/2M/Coded PHY 功能演示。
- 主要硬件：B91 开发板 x 2

以 B91 为例，应用层代码在 tl\_ble\_sdk/vendor/feature\_test/feature\_2M\_coded\_phy 下，需要修改 tl\_ble\_sdk/vendor/feature\_test/feature\_config.h 中的定义：

```
#define FEATURE_TEST_MODE           TEST_2M_CODED_PHY_CONNECTION
```

来激活这部分代码，做动态切换 PHY 的演示。

初始化调用 blc\_ll\_init2MPhyCodedPhy\_feature() 来使能 PHY 切换功能。在 main\_loop() 中的 feature\_2m\_phy\_test\_mainloop() 中实现动态切换 PHY 的功能：

(1) Central 建立连接后，每隔 1s 给各 Peripheral 做一次 WriteCmd，有效数据长度为 8Bytes（实际发送间隔还与 Connection Interval 有关）。

(2) Peripheral 建立连接后，每隔 1s 给各个 Central 做一次 Notify，有效数据长度为 8Bytes（实际发送间隔还与 Connection Interval 有关）。



(3) 各个连接每隔 10s 做一次 PHY 的切换，顺序是 Coded\_S8 → 2M → 1M → Coded\_S8...

实际抓包如下：

P...	Time	Item	Transmitter	Receiver
80'690	7:05:24 PM.315 799 125	LLCP Reserved (0x68)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
80'727	7:05:24 PM.616 258 750	ATT Write Command Packet (52: B5 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
80'739	7:05:24 PM.706 029 250	ATT Notification Packet (48: B6 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
80'853	7:05:25 PM.606 259 375	ATT Write Command Packet (52: B6 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
80'872	7:05:25 PM.696 029 625	ATT Notification Packet (48: B7 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
81'003	7:05:26 PM.626 260 125	ATT Write Command Packet (52: B7 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
81'013	7:05:26 PM.686 030 000	ATT Notification Packet (48: B8 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
81'144	7:05:27 PM.616 260 500	ATT Write Command Packet (52: B8 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
81'154	7:05:27 PM.676 260 625	LLCP PHY Request (Tx=LE Coded, Rx=LE Coded)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
81'161	7:05:27 PM.706 030 500	LLCP PHY Response (Tx=LE Coded, Rx=LE Coded)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
81'163	7:05:27 PM.706 513 625	ATT Notification Packet (48: B9 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
81'166	7:05:27 PM.736 260 625	LLCP PHY Update (M->S=LE Coded, S->M=LE Coded, Inst=14688 (+19)   557.306 260 625 (+570.000 ms))	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
81'167	7:05:27 PM.736 530 000	Empty LE Packets (x 37, 539 ms)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
81'274	7:05:28 PM.607 538 875	ATT Write Command Packet (52: B9 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
81'290	7:05:28 PM.696 669 500	ATT Notification Packet (48: BA 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
81'422	7:05:29 PM.627 539 375	ATT Write Command Packet (52: BA 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
81'432	7:05:29 PM.688 408 750	ATT Notification Packet (48: BB 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
81'571	7:05:30 PM.617 539 875	ATT Write Command Packet (52: BB 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02

Figure 11.4: feature\_2M\_coded\_phy Coded Req

P...	Time	Item	Transmitter	Receiver
83'506	7:05:44 PM.627 547 750	ATT Write Command Packet (52: C9 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
83'516	7:05:44 PM.686 678 375	ATT Notification Packet (48: CA 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
83'656	7:05:45 PM.617 548 250	ATT Write Command Packet (52: CA 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
83'670	7:05:45 PM.706 679 125	ATT Notification Packet (48: CB 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
83'789	7:05:46 PM.607 548 750	ATT Write Command Packet (52: CB 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
83'804	7:05:46 PM.696 679 375	ATT Notification Packet (48: CC 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
83'937	7:05:47 PM.627 549 500	ATT Write Command Packet (52: CC 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
83'943	7:05:47 PM.657 549 500	LLCP PHY Request (Tx=LE 2M, Rx=LE 2M)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
83'949	7:05:47 PM.686 680 000	LLCP PHY Response (Tx=LE 2M, Rx=LE 2M)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
83'951	7:05:47 PM.688 610 500	ATT Notification Packet (48: CD 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
83'955	7:05:47 PM.717 549 625	LLCP PHY Update (M->S=LE 2M, S->M=LE 2M, Inst=15'354 (+19)   577.282 649 625 (+570.000 ms))	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
83'956	7:05:47 PM.718 738 875	Empty LE Packets (x 37, 538 ms)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
84'078	7:05:48 PM.616 204 375	ATT Write Command Packet (52: CD 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
84'086	7:05:48 PM.706 011 125	ATT Notification Packet (48: CE 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
84'208	7:05:49 PM.606 205 000	ATT Write Command Packet (52: CE 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
84'225	7:05:49 PM.696 011 750	ATT Notification Packet (48: CF 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
84'356	7:05:50 PM.626 205 375	ATT Write Command Packet (52: CF 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
84'364	7:05:50 PM.686 012 375	ATT Notification Packet (48: D0 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01

Figure 11.5: feature\_2M\_coded\_phy 2M Req



P...	Time	Item	Transmitter	Receiver
11'189	6:57:21 PM.958 829 125	ATT Notification Packet (48: 45 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
11'190	6:57:21 PM.959 081 750	ATT Write Command Packet (52: 45 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
11'334	6:57:22 PM.948 829 750	ATT Notification Packet (48: 46 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
11'335	6:57:22 PM.949 082 500	ATT Write Command Packet (52: 46 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
11'484	6:57:23 PM.938 829 875	LLCP PHY Request (Tx=LE 1M, Rx=LE 1M)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
11'485	6:57:23 PM.939 034 625	LLCP PHY Request (Tx=LE 1M, Rx=LE 1M)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
11'486	6:57:23 PM.939 239 375	ATT Notification Packet (48: 47 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
11'487	6:57:23 PM.939 492 375	ATT Write Command Packet (52: 47 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
11'488	6:57:23 PM.939 745 000	LLCP PHY Response (Tx=LE 1M, Rx=LE 1M)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
11'489	6:57:23 PM.939 949 750	LLCP PHY Update (M->S=LE 1M, S->M=LE 1M, Inst=z'353 (+20)   73.539 949 750 (+600.000 ms))	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
11'493	6:57:23 PM.968 636 875	LLCP PHY Update (M->S=LE 1M, S->M=LE 1M, Inst=z'353 (+19)   73.538 636 875 (+570.000 ms))	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
11'494	6:57:23 PM.968 850 125	Empty LE Packets (x 37, 540 ms)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
11'637	6:57:24 PM.958 861 625	ATT Notification Packet (48: 48 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
11'638	6:57:24 PM.959 211 125	ATT Write Command Packet (52: 48 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
11'793	6:57:25 PM.948 862 000	ATT Notification Packet (48: 49 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
11'794	6:57:25 PM.949 211 625	ATT Write Command Packet (52: 49 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
11'940	6:57:26 PM.938 862 625	ATT Notification Packet (48: 4A 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
11'941	6:57:26 PM.939 212 375	ATT Write Command Packet (52: 4A 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02

Figure 11.6: feature\_2M\_coded\_phy 1M Req

### 11.3 feature\_gatt\_api

- 功能：BLE GATT 指令功能演示与 API 使用方法。tl\_ble\_sdk 中部分实例的 SDP 流程的参考实现中用到了这些指令，用户可以使用该演示代码进行单指令测试。
- 主要硬件：B91 开发板 x 2

以 B91 为例，应用层代码在 tl\_ble\_sdk/vendor/feature\_test/feature\_gatt\_api 下，需要修改 tl\_ble\_sdk/vendor/feature\_test/feature\_config.h 中的定义：

```
#define FEATURE_TEST_MODE           TEST_GATT_API
```

来激活这部分代码。在 app.c 中，通过修改 TEST\_API 的定义来测试不同的 GATT 指令。

编译，将生成的固件分别烧录到两个开发板中。上电，开发板上红灯每隔 2s 做一次亮、灭的切换。通过按其中一个开发板（作为 Central）上的 SW4 按键，触发连接，通过抓包可以看到每隔 2s，Central 发送一个测试指令，Peripheral 会进行相应的回复，回复的实现参考函数 app\_gatt\_data\_handler()。

以下是分别定义 TEST\_API 为不同的指令测试定义时的抓包：



P...	Time	Item	Payl...	Transmitter	Receiver
1	5:02:05 PM.005 955 125	⊕ ⓘ Connectable ("gatt" 55:66:77:00:91:01, Initiator "gatt" 55:66:77:00:9...		Master: "gatt" 55:66:77:00:91:01	Slave: "gatt" 55:66:77:00:91:02
4	5:02:05 PM.019 693 125	⊕ ⓘ Connectable ("gatt" 55:66:77:00:91:02, 1.59 min)		Master: "gatt" 55:66:77:00:91:02	Slave: "Scanning Device"
3'242	5:02:23 PM.139 076 625	⊕ ⓘ SMP Security Request (Bonding)	2 bytes ...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
3'245	5:02:23 PM.155 342 250	⊕ ⓘ Connectable ("gatt" 55:66:77:00:91:01, 1.29 min)		Master: "gatt" 55:66:77:00:91:01	Slave: "Scanning Device"
3'288	5:02:23 PM.326 346 625	⊕ ⓘ ATT Read By Group Type Request Packet (1 - Max Handle, Characterist...		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
3'294	5:02:23 PM.357 366 375	⊕ ⓘ ATT Read By Group Type Response Packet (12 03 00 00 2A > 02 05 00 ...)	18 byte...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
3'297	5:02:23 PM.388 387 125	⊕ ⓘ Empty LE Packets (x 124, 2 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
3'659	5:02:25 PM.326 337 750	⊕ ⓘ ATT Read By Group Type Request Packet (1 - Max Handle, Characterist...		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
3'666	5:02:25 PM.357 357 500	⊕ ⓘ ATT Read By Group Type Response Packet (12 03 00 00 2A > 02 05 00 ...)	18 byte...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
3'671	5:02:25 PM.388 378 250	⊕ ⓘ Empty LE Packets (x 124, 2 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
4'018	5:02:27 PM.326 328 875	⊕ ⓘ ATT Read By Group Type Request Packet (1 - Max Handle, Characterist...		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
4'024	5:02:27 PM.357 348 625	⊕ ⓘ ATT Read By Group Type Response Packet (12 03 00 00 2A > 02 05 00 ...)	18 byte...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
4'031	5:02:27 PM.388 369 375	⊕ ⓘ Empty LE Packets (x 125, 1 retry, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
4'398	5:02:29 PM.326 320 000	⊕ ⓘ ATT Read By Group Type Request Packet (1 - Max Handle, Characterist...		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
4'408	5:02:29 PM.357 339 625	⊕ ⓘ ATT Read By Group Type Response Packet (12 03 00 00 2A > 02 05 00 ...)	18 byte...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
4'412	5:02:29 PM.388 360 375	⊕ ⓘ Empty LE Packets (x 122, 4 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
4'782	5:02:31 PM.326 310 625	⊕ ⓘ ATT Read By Group Type Request Packet (1 - Max Handle, Characterist...		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
4'789	5:02:31 PM.357 330 625	⊕ ⓘ ATT Read By Group Type Response Packet (12 03 00 00 2A > 02 05 00 ...)	18 byte...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
4'793	5:02:31 PM.388 351 750	⊕ ⓘ Empty LE Packets (x 118, 7 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01

Figure 11.7: GATT API test TEST\_READ\_BY\_GROUP\_TYPE\_REQ

P...	Time	Item	Payl...	Transmitter	Receiver
1	5:10:45 PM.000 336 750	⊕ ⓘ Connectable ("gatt" 55:66:77:00:91:02, 9.77 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "Scanning Device"
297	5:10:47 PM.745 293 125	⊕ ⓘ Connectable ("gatt" 55:66:77:00:91:01, Initiator "gatt" 55:66:77:00:9...		Master: "gatt" 55:66:77:00:91:01	Slave: "gatt" 55:66:77:00:91:02
314	5:10:47 PM.839 464 625	⊕ ⓘ SMP Security Request (Bonding)	2 bytes ...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
315	5:10:47 PM.854 043 250	⊕ ⓘ Connectable ("gatt" 55:66:77:00:91:01, 6.92 s)		Master: "gatt" 55:66:77:00:91:01	Slave: "Scanning Device"
322	5:10:47 PM.870 485 250	⊕ ⓘ ATT Find Information Request Packet (1 - Max Handle)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
323	5:10:47 PM.870 788 125	⊕ ⓘ Empty LE Packets (x 119, 8 retries, 2 s)		Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
704	5:10:49 PM.870 476 375	⊕ ⓘ ATT Find Information Request Packet (1 - Max Handle)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
710	5:10:49 PM.901 496 375	⊕ ⓘ ATT Find Information Response Packet (Primary Service > Characteristic...)	20 byte...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
714	5:10:49 PM.932 517 125	⊕ ⓘ Empty LE Packets (x 118, 8 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'064	5:10:51 PM.870 467 750	⊕ ⓘ ATT Find Information Request Packet (1 - Max Handle)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'070	5:10:51 PM.901 487 500	⊕ ⓘ ATT Find Information Response Packet (Primary Service > Characteristic...)	20 byte...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
1'074	5:10:51 PM.932 508 250	⊕ ⓘ Empty LE Packets (x 121, 5 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'451	5:10:53 PM.870 459 125	⊕ ⓘ ATT Find Information Request Packet (1 - Max Handle)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'458	5:10:53 PM.901 478 625	⊕ ⓘ ATT Find Information Response Packet (Primary Service > Characteristic...)	20 byte...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02

Figure 11.8: GATT API test TEST\_FIND\_INFO\_REQ

P...	Time	Item	Payl...	Transmitter	Receiver
1	5:17:54 PM.023 646 375	⊕ ⓘ Connectable ("gatt" 55:66:77:00:91:02, 14.7 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "Scanning Device"
21	5:17:54 PM.223 858 750	⊕ ⓘ Connectable ("gatt" 55:66:77:00:91:01, Initiator "gatt" 55:66:77:00:9...		Master: "gatt" 55:66:77:00:91:01	Slave: "gatt" 55:66:77:00:91:02
42	5:17:54 PM.351 704 375	⊕ ⓘ SMP Security Request (Bonding)	2 bytes ...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
43	5:17:54 PM.364 521 250	⊕ ⓘ Connectable ("gatt" 55:66:77:00:91:01, 14.2 s)		Master: "gatt" 55:66:77:00:91:01	Slave: "Scanning Device"
48	5:17:54 PM.382 265 500	⊕ ⓘ Empty LE Packets (x 88, 1.38 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
331	5:17:55 PM.757 718 625	⊕ ⓘ ATT Find By Type Value Request Packet (1 - Max Handle, PnP ID, Source...)	7 bytes ...	Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
337	5:17:55 PM.788 738 250	⊕ ⓘ ATT Find By Type Value Response Packet (14 - 14)	4 bytes ...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
342	5:17:55 PM.819 759 000	⊕ ⓘ Empty LE Packets (x 125, 1 retry, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
723	5:17:57 PM.757 709 375	⊕ ⓘ ATT Find By Type Value Request Packet (1 - Max Handle, PnP ID, Source...)	7 bytes ...	Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
729	5:17:57 PM.788 729 250	⊕ ⓘ ATT Find By Type Value Response Packet (14 - 14)	4 bytes ...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
733	5:17:57 PM.819 750 000	⊕ ⓘ Empty LE Packets (x 123, 5 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'104	5:17:59 PM.757 700 125	⊕ ⓘ ATT Find By Type Value Request Packet (1 - Max Handle, PnP ID, Source...)	7 bytes ...	Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'111	5:17:59 PM.788 720 375	⊕ ⓘ ATT Find By Type Value Response Packet (14 - 14)	4 bytes ...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
1'187	5:18:00 PM.163 488 875	⊕ ⓘ LLCP LE Power Control Response (Delta=-82 dB, Tx Power=-95 dBm)	33 byte...	Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'192	5:18:00 PM.194 739 250	⊕ ⓘ Empty LE Packets (x 101, 1.56 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'502	5:18:01 PM.757 691 500	⊕ ⓘ ATT End Of Train Value Response (End / 1)	7 bytes ...	Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01

Figure 11.9: GATT API test TEST\_FIND\_BY\_TYPE\_VALUE\_REQ



P...	Time	Item	Payload	Transmitter	Receiver
1	5:19:31 PM.029 298 125	⌚💡 Connectable ("gatt" 55:66:77:00:91:01, Initiator...		Master: "gatt" 55:66:77:00:91:01	Slave: "gatt" 55:66:77:00:91:02
225	5:19:33 PM.484 844 750	⌚💡 Connectable ("gatt" 55:66:77:00:91:02, 30.2 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "Scanning Device"
254	5:19:33 PM.650 559 500	⌚🔑 SMP Security Request (Bonding)	2 bytes (0B 01)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
258	5:19:33 PM.663 049 375	⌚💡 Connectable ("gatt" 55:66:77:00:91:01, 30 s)		Master: "gatt" 55:66:77:00:91:01	Slave: "Scanning Device"
260	5:19:33 PM.681 121 125	⌚➡️ Empty LE Packets (x 118, 1.81 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
605	5:19:35 PM.494 072 125	⌚💡 ATT Read Request Packet (3)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
612	5:19:35 PM.525 091 750	⌚💡 ATT Read Response Packet ("feature")	7 bytes (66 65 61 74 75 72 65)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
616	5:19:35 PM.556 112 625	⌚➡️ Empty LE Packets (x 124, 2 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
963	5:19:37 PM.494 063 125	⌚💡 ATT Read Request Packet (3)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
969	5:19:37 PM.525 083 000	⌚💡 ATT Read Response Packet ("feature")	7 bytes (66 65 61 74 75 72 65)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
974	5:19:37 PM.556 103 625	⌚➡️ Empty LE Packets (x 118, 8 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'332	5:19:39 PM.494 054 125	⌚💡 ATT Read Request Packet (3)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'338	5:19:39 PM.525 074 000	⌚💡 ATT Read Response Packet ("feature")	7 bytes (66 65 61 74 75 72 65)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
1'342	5:19:39 PM.556 094 625	⌚➡️ Empty LE Packets (x 117, 9 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'687	5:19:41 PM.494 044 875	⌚💡 ATT Read Request Packet (3)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'697	5:19:41 PM.525 064 750	⌚💡 ATT Read Response Packet ("feature")	7 bytes (66 65 61 74 75 72 65)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
1'701	5:19:41 PM.556 085 750	⌚➡️ Empty LE Packets (x 119, 6 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
2'037	5:19:43 PM.494 036 000	⌚💡 ATT Read Request Packet (3)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
2'042	5:19:43 PM.525 056 000	⌚💡 ATT Read Response Packet ("feature")	7 bytes (66 65 61 74 75 72 65)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
2'047	5:19:43 PM.556 076 750	⌚➡️ Empty LE Packets (x 117, 9 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01

Figure 11.10: GATT API test TEST\_READ\_REQ

P...	Time	Item	Payload	Transmitter	Receiver
1	5:28:50 PM.004 054 625	⌚💡 Connectable ("gatt" 55:66:77:00:91:01, Initiator...		Master: "gatt" 55:66:77:00:91:01	Slave: "gatt" 55:66:77:00:91:02
202	5:28:52 PM.364 372 625	⌚💡 Connectable ("gatt" 55:66:77:00:91:02, 9.64 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "Scanning Device"
213	5:28:52 PM.442 587 875	⌚🔑 SMP Security Request (Bonding)	2 bytes (0B 01)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
214	5:28:52 PM.457 181 375	⌚💡 Connectable ("gatt" 55:66:77:00:91:01, 9.54 s)		Master: "gatt" 55:66:77:00:91:01	Slave: "Scanning Device"
423	5:28:53 PM.473 145 125	⌚💡 L2CAP SDU (Basic)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
595	5:28:54 PM.379 850 125	⌚💡 ATT Read Blob Request Packet (3 @1)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
601	5:28:54 PM.410 869 750	⌚💡 ATT Read Blob Response Packet (6 bytes)	6 bytes (65 61 74 75 72 65)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
608	5:28:54 PM.441 890 625	⌚➡️ Empty LE Packets (x 126, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
961	5:28:56 PM.379 841 125	⌚💡 ATT Read Blob Request Packet (3 @1)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
969	5:28:56 PM.410 860 875	⌚💡 ATT Read Blob Response Packet (6 bytes)	6 bytes (65 61 74 75 72 65)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
974	5:28:56 PM.441 881 500	⌚➡️ Empty LE Packets (x 125, 1 retry, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'320	5:28:58 PM.379 832 000	⌚💡 ATT Read Blob Request Packet (3 @1)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'326	5:28:58 PM.410 851 625	⌚💡 ATT Read Blob Response Packet (6 bytes)	6 bytes (65 61 74 75 72 65)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
1'332	5:28:58 PM.441 872 250	⌚➡️ Empty LE Packets (x 113, 11 retries, 1.91 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'659	5:29:00 PM.379 823 000	⌚💡 ATT Read Blob Request Packet (3 @1)		Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
1'665	5:29:00 PM.410 842 750	⌚💡 ATT Read Blob Response Packet (6 bytes)	6 bytes (65 61 74 75 72 65)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
1'673	5:29:00 PM.441 863 375	⌚➡️ Emntry I F Packets (x 85, 7 retries, 1.47 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01

Figure 11.11: GATT API test TEST\_READ\_BLOB\_REQ

## 11.4 feature\_ll\_more\_data

- 功能：BLE MD=1 的演示。MD，即 More Data，是数据通道 PDU Header 中的一个标志位 MD flag。同时，该演示也提供用户做吞吐量的测试使用。
- 主要硬件：B91 开发板 x 2

以 B91 为例，应用层代码在 tl\_ble\_sdk/vendor/feature\_test/feature\_ll\_more\_data 下，需要修改 tl\_ble\_sdk/vendor/feature\_test/feature\_config.h 中的定义：

```
#define FEATURE_TEST_MODE           TEST_LL_MD
```

来激活这部分代码。

编译，将生成的固件分别烧录到两个开发板中。上电，按其中一个开发板（作为 Central）的 SW4 按键，触发连接。连接成功后，红灯点亮（如果连接多个 Peripheral，分别点亮：红色、白色、绿色、蓝色灯）。连接成功



后同时按下 Central 的两个按键，可以从抓包中看到 Central 不断向 Peripheral 发送 WriteCmd，其包中的 MD flag 被置为 1，即下一个包已经准备好，即将发送：

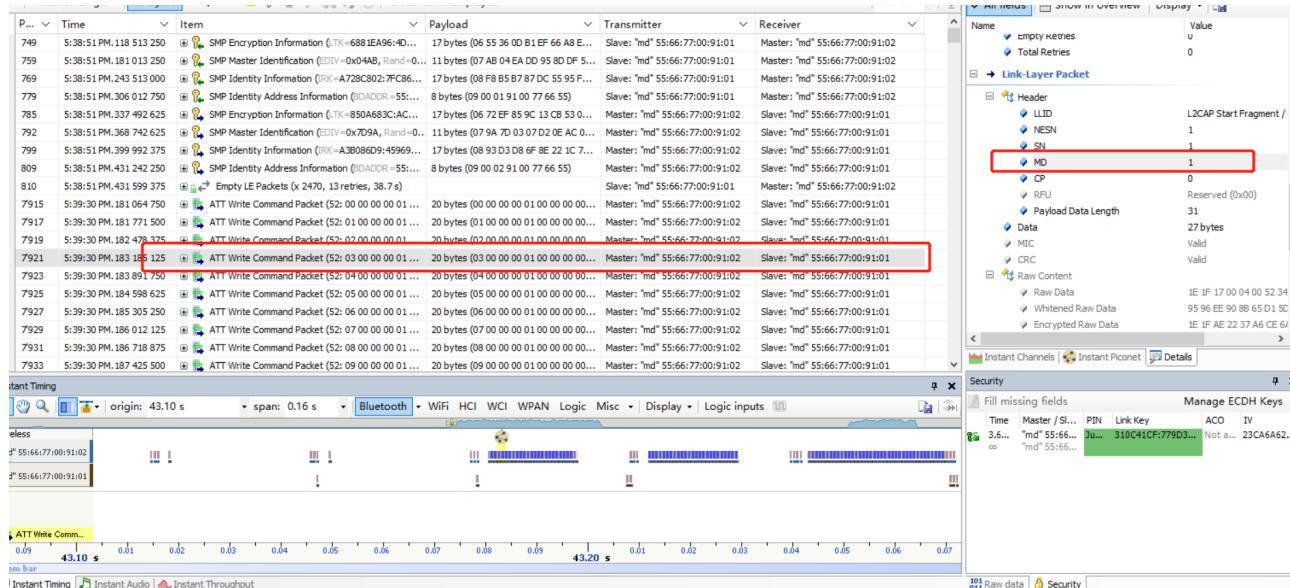


Figure 11.12: feature\_ll\_more\_data WirteCmd

同时按下 Peripheral 的两个按键，可以从抓包中看到 Peripheral 不断向 Central 发送 Notify 包，其中的 MD flag 被置为 1：

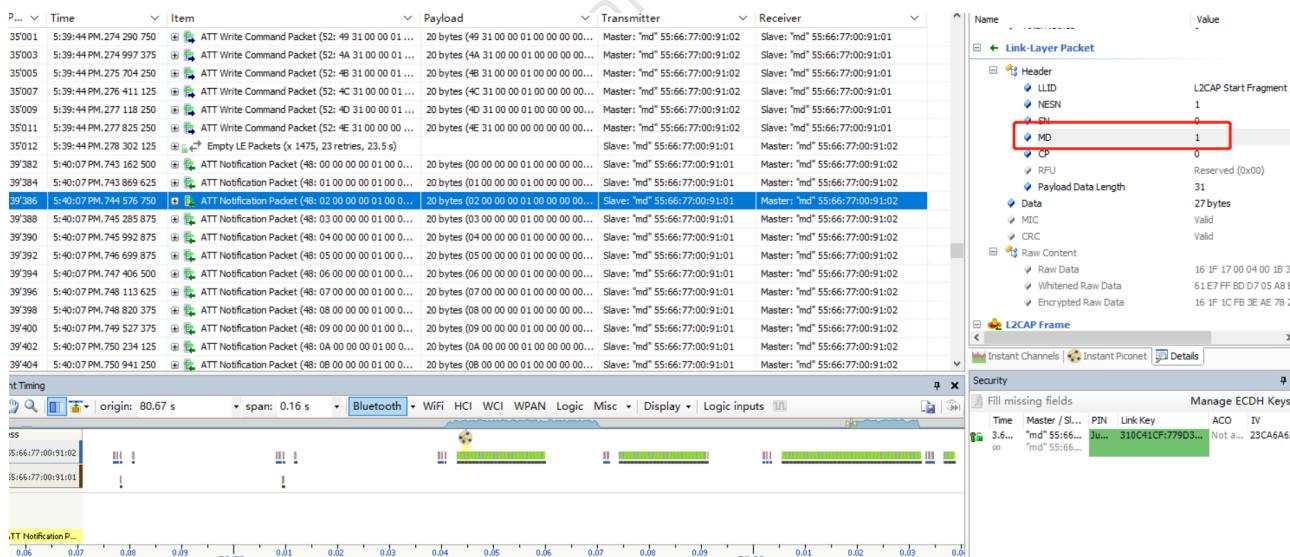


Figure 11.13: feature\_ll\_more\_data Notify

## 11.5 feature\_dle

- 功能：BLE DLE(Data Length Extension) 和 MTU Exchange 以及 L2CAP 分包组包功能演示。
- 主要硬件：B91 开发板 x 2

以 B91 为例，应用层代码在 `tl_ble_sdk/vendor/feature_test/feature_dle` 下，需要修改 `tl_ble_sdk/vendor/feature_test/feature_config.h` 中的定义：

```
#define FEATURE_TEST_MODE TEST_LL_DL
```

来激活这部分代码，做 DLE 和 MTU 的演示。

通过修改 DLE\_LENGTH\_SELECT 的定义来修改 DataLength，演示内容为：

- (1) 通过两个按键同时按下触发测试。
  - (2) Central 触发测试后，每个连接，每隔 1s 发送一个 WriteCmd。
  - (3) Peripheral 触发测试后，每个连接，每隔 1s 发送一个 Notify。

抓包如下：

P...	Time	Item	Payl...	Transmitter	Receiver
186	7:29:39 PM,776 199 250	⌚🔑 SMP Pairing Confirm (Ca=1183DCCD:F3676F57:F0305FAF:A1B32D05)	17 byte...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
191	7:29:39 PM,807 219 125	⌚🔑 SMP Pairing Confirm (Cb=2C9954BC:FA9032AE:60809925:9064F84F)	17 byte...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
196	7:29:39 PM,838 699 250	⌚🔑 SMP Pairing Random (Na=A12FDF8F:DEB97684:C2677E71:AB6C5596)	17 byte...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
201	7:29:39 PM,869 719 000	⌚🔑 SMP Pairing Random (Nb=0E8992DA:6A7356BF:6F203F79:A1FC6519)	17 byte...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
206	7:29:39 PM,901 201 000	⌚ LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, S...	23 byte...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
213	7:29:39 PM,932 221 750	⌚ LLCP Encryption Response (SKD_S=0x51A60FAF408D8DB8, IVs=0x49E9...	13 byte...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
224	7:29:40 PM,025 969 250	⌚ LLCP Start Encryption Request	1 byte (...)	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
228	7:29:40 PM,057 449 750	⌚ LLCP Start Encryption Response	1 byte (...)	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
232	7:29:40 PM,088 469 250	⌚ LLCP Start Encryption Response	1 byte (...)	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
234	7:29:40 PM,088 968 375	⌚ LLCP Length Request (MaxRx=200 bytes, 1.71 ms, MaxTx=200 bytes,...)	9 bytes ...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
236	7:29:40 PM,089 531 625	⌚ ATT Exchange MTU Request Packet		Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
241	7:29:40 PM,119 949 250	⌚ LLCP Length Request (MaxRx=200 bytes, 1.71 ms, MaxTx=200 bytes,...)	9 bytes ...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
243	7:29:40 PM,120 512 625	⌚ ATT Exchange MTU Request Packet		Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
245	7:29:40 PM,121 059 500	⌚ LLCP Length Response (MaxRx=200 bytes, 1.71 ms, MaxTx=200 bytes,...)	9 bytes ...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
247	7:29:40 PM,121 623 000	⌚ ATT Exchange MTU Response Packet		Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
252	7:29:40 PM,150 969 375	⌚ SMP Encryption Information (LTK=B5DCC78F:3F2603EA:3A756A2C:F4...	17 byte...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
254	7:29:40 PM,151 628 875	⌚ LLCP Length Response (MaxRx=200 bytes, 1.71 ms, MaxTx=200 bytes,...)	9 bytes ...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
256	7:29:40 PM,152 192 250	⌚ ATT Exchange MTU Response Packet		Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
262	7:29:40 PM,213 469 375	⌚ SMP Master Identification (EDIV=0xC4F7, Rand=0xA82178CB48BD234)	11 byte...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01

**Figure 11.14:** feature\_dle DLE & MTU Exchange

**Figure 11.15:** Central 和 Peripheral 分别做 WriteCmd 和 Notify

由于 DLE 小于 MTU，可以看到分包的效果：

**Figure 11.16:** 分包发送

## 11.6 feature\_smp

- 功能：BLE SMP (Security Manager Protocol) 功能演示。
  - 主要硬件：B91 开发板 x 2

以 B91 为例，应用层代码在 tl\_ble\_sdk/vendor/feature\_demo/feature\_smp 下，需要修改 tl\_ble\_sdk/vendor/feature\_demo/feature\_config.h 中的定义：

```
#define FEATURE_TEST_MODE TEST_SMP
```

- (1) 由于要演示 SMP 下多个加密配置，为方便用户参考，在 feature\_smp/app.c 中定义 SMP\_TEST\_MODE 宏，用户仅需修改该宏的定义来实现不同加密配置的演示代码，通过全局搜索该定义的方式，得到 porting 相应 SMP 功能的最简代码。
  - (2) 配置 SMP 相关参数时，如果 Central 和 Peripheral 配置相同，则使用不带后缀的 API，如果不同，应按照需求使用带 \_central/\_periph 后缀的 API。演示代码中为了方便用户另作配置进行测试，对 SMP 加密使能的演示，使用带后缀的 API。
  - (3) 为方便观察现象，代码中加入了 LED 指示灯的显示，定义如下：
    - 绿色：ON：Central connected；OFF：Central disconnected.
    - 红色：ON：Peripheral connected；OFF：Peripheral disconnected.
    - 蓝色：ON：Pair Succeeded；OFF：Disconnected，没走 Pair 流程也不亮。
    - 白色：ON：Encryption succeeded；OFF：Disconnected.
  - (4) 在 SMP 中的角色分为 Initiator 和 Responder，分别对应 BLE 中的角色 Central 和 Peripheral。
  - (5) 目前 feature\_smp 中 UART 功能，仅做了 B91\_B92 的适配，如需适配其他芯片，请参考 tl\_platform\_sdk

**提示：** LE Security Connections 的配对方式，要求 MTU>=65。所以，这里没有使用 default\_buffer.h 中的定义。

### 11.6.1 Peripheral 和 Central 均不使能 SMP

Peripheral 和 Central 的 SMP 功能关闭的演示，需在 feature\_smp/app.c 中定义（默认）：



```
#define SMP_TEST_MODE           SMP_TEST_NOT_SUPPORT
```

编译，分别擦除 Flash 后烧录到两个开发板中。按其中一个开发板（作为 Central）的按键 SW4（启动连接），可以看到两个开发板分别亮了绿灯和红灯，代表连接成功：

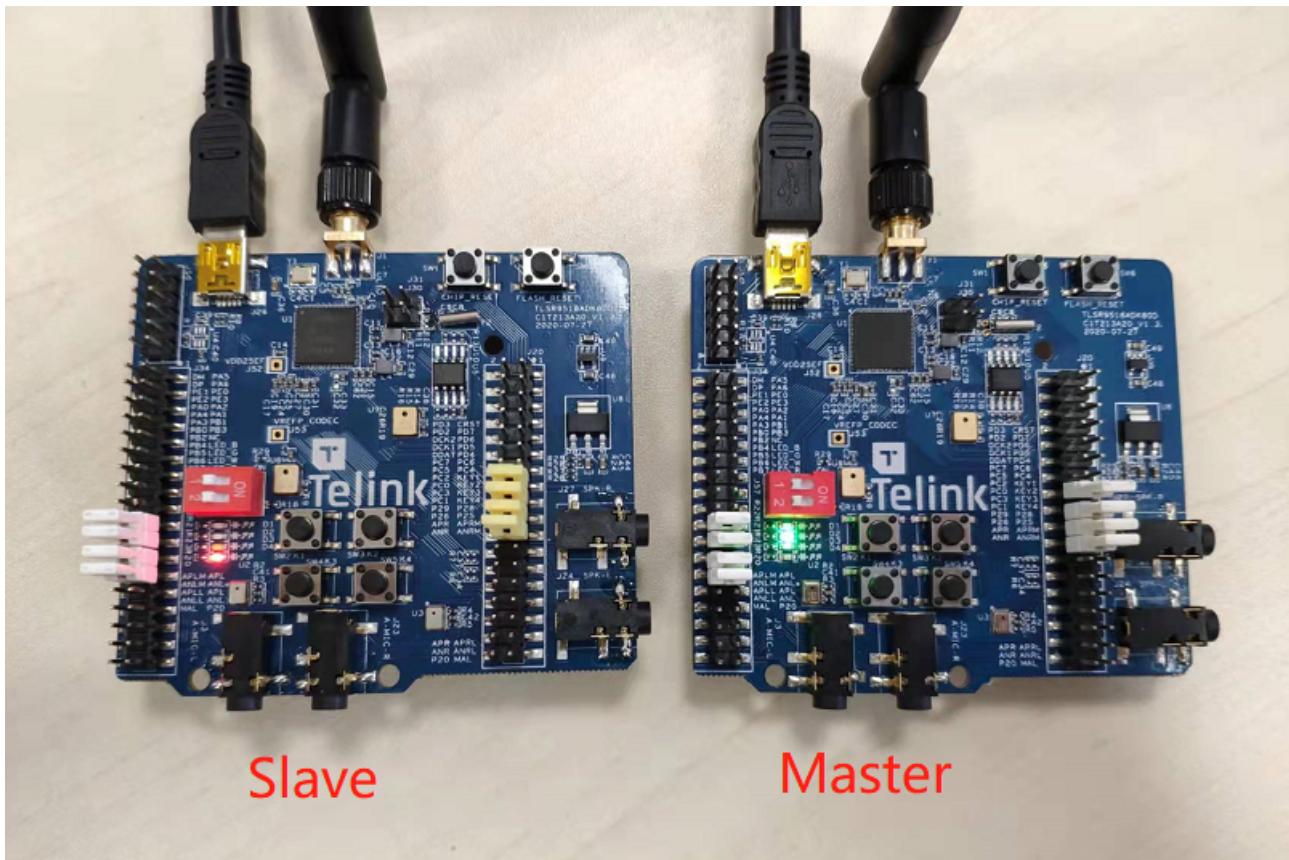


Figure 11.17: SMP 失能时 Peripheral 和 Central 连接成功

抓包如下：

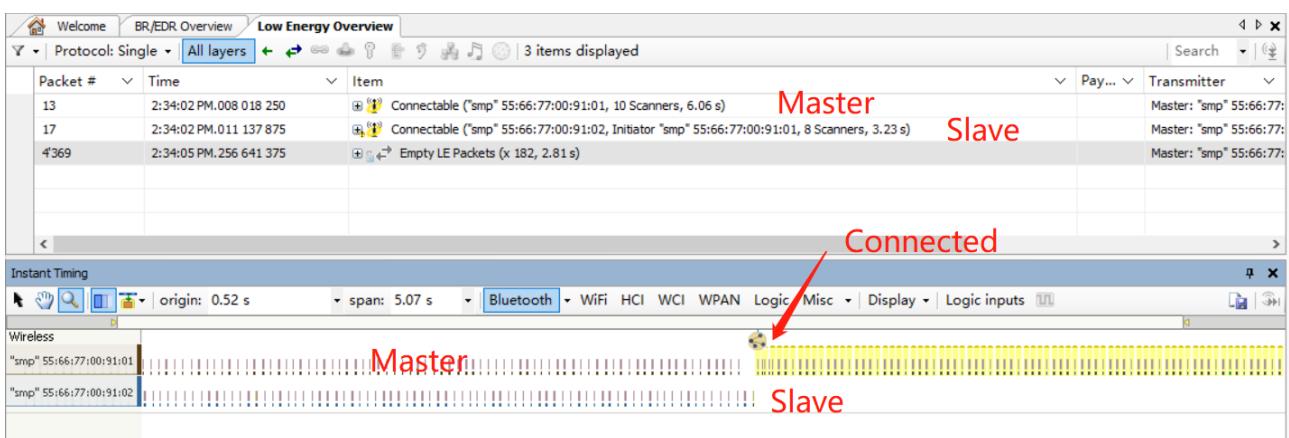


Figure 11.18: SMP 失能时 Peripheral 和 Central 连接成功的抓包



## 11.6.2 Legacy Just Works

Central 和 Peripheral 使能 SMP 功能，以 Legacy Just Works 方式配对的演示。该演示需在 feature\_smp/app.c 中定义：

```
#define SMP_TEST_MODE           SMP_TEST_LEGACY_JW
```

### 说明：

如果用户不希望 Peripheral 建立连接后发 Security Request，应在初始化代码中添加（默认注释掉）：

```
blc_smp_configSecurityRequestSending( SecReq_NOT_SEND, SecReq_NOT_SEND, 0);
```

该 API 仅针对 Peripheral（只有 Peripheral 会发 Security Request），因此不存在后缀为 \_periph/\_central 的相应 API。

### 说明：

如果用户希望 Central 建立连接等待 Peripheral 发出 Security Request 后再发 Pairing Request，应在初始化代码中添加（默认注释掉）：

```
blc_smp_configPairingRequestSending(PairReq_SEND_upon_SecReq, PairReq_SEND_upon_SecReq);
```

该 API 仅针对 Central（只有 Central 会发 Pairing Request），因此不存在后缀为 \_periph/\_central 的相应 API。

编译，分别擦除 Flash 后烧录到两个开发板中。按其中一个开发板（作为 Central）的按键 SW4（启动连接），连接成功后，可以看到 Central 亮绿色、白色、蓝色灯，Peripheral 亮红色、白色、蓝色灯，代表二者 Pair 成功，抓包如下：

P...	Time	Item	Transmitter	Payl...
4	3:04:02 PM.009 471 875	Connectable ("smp" 55:66:77:00:91:01, 5 Scanners, 48 s)	Master: "smp" 55:66:77:00:91:01	
854	3:04:06 PM.082 387 125	Connectable ("smp" 55:66:77:00:91:02, Initiator "smp" 55:66:77:00:91:01, 4 Scanners, 2.38 s)	Master: "smp" 55:66:77:00:91:02	
1'606	3:04:08 PM.478 718 875	SMP Pairing Request (No Input No Output, Bonding, Int=EndKey   IdKey, Rsp=EndKey   IdKey)	Master: "smp" 55:66:77:00:91:01	7 bytes ...
1'607	3:04:08 PM.509 826 250	SMP Security Request (Bonding)	Slave: "smp" 55:66:77:00:91:02	2 bytes ...
1'609	3:04:08 PM.510 335 875	SMP Pairing Response (No Input No Output, Bonding, Int=EndKey   IdKey, Rsp=EndKey   IdKey)	Slave: "smp" 55:66:77:00:91:02	7 bytes ...
1'619	3:04:08 PM.541 219 250	SMP Pairing Confirm (Ca=E8A0DDA3:13196DC1:251D7A6D:9C413836)	Master: "smp" 55:66:77:00:91:01	17 byte...
1'632	3:04:08 PM.572 238 875	SMP Pairing Confirm (Cb=37CB4C8C:722C5B6A:C7833FE0:73171D2)	Slave: "smp" 55:66:77:00:91:02	17 byte...
1'641	3:04:08 PM.603 718 875	SMP Pairing Random (Na=47132D6A:8F9B4F69:E0BCB4D9:97A9515C)	Master: "smp" 55:66:77:00:91:01	17 byte...
1'651	3:04:08 PM.634 739 125	SMP Pairing Random (Nb=CE285F91:EAC9D347:54F2040F:45EEDCDA)	Slave: "smp" 55:66:77:00:91:02	17 byte...
1'657	3:04:08 PM.666 219 250	LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, SKDm=0x4FC0E6552EFC57...)	Master: "smp" 55:66:77:00:91:01	23 byte...
1'664	3:04:08 PM.697 239 250	LLCP Encryption Response (SKDs=0xEACAEF932765AE6, IVs=0x80DEC572)	Slave: "smp" 55:66:77:00:91:02	13 byte...
1'690	3:04:08 PM.790 989 125	LLCP Start Encryption Request	Slave: "smp" 55:66:77:00:91:02	1 byte ...
1'699	3:04:08 PM.822 469 375	LLCP Start Encryption Response	Master: "smp" 55:66:77:00:91:01	1 byte ...
1'713	3:04:08 PM.853 489 250	LLCP Start Encryption Response	Slave: "smp" 55:66:77:00:91:02	1 byte ...
1'728	3:04:08 PM.915 989 375	SMP Encryption Information (LTK=987E0AC4:8F9C8612:01A7515A:10BB889F)	Slave: "smp" 55:66:77:00:91:02	17 byte...
1'742	3:04:08 PM.978 489 750	SMP Master Identification (EDIV=0x6384, Rand=0x22A540EEECF6798)	Slave: "smp" 55:66:77:00:91:02	11 byte...
1'758	3:04:09 PM.040 989 625	SMP Identity Information (IRK=A3B086D9:459694CF:7A1C228E:6FD8D393)	Slave: "smp" 55:66:77:00:91:02	17 byte...
1'777	3:04:09 PM.103 489 625	SMP Identity Address Information (BDADDR=55:66:77:00:91:02)	Slave: "smp" 55:66:77:00:91:02	8 bytes ...
1'788	3:04:09 PM.134 969 875	SMP Encryption Information (LTK=1246783F:DACE1A3C:B5E9E18C:C2FC0409)	Master: "smp" 55:66:77:00:91:01	17 byte...
1'804	3:04:09 PM.166 219 875	SMP Master Identification (EDIV=0xD5E2, Rand=0xFDE22565FD06C8BC)	Master: "smp" 55:66:77:00:91:01	11 byte...
1'813	3:04:09 PM.197 469 875	SMP Identity Information (IRK=A728C802:7FC864DC:FF9555DC:87B7B5F8)	Master: "smp" 55:66:77:00:91:01	17 byte...
1'826	3:04:09 PM.228 720 000	SMP Identity Address Information (BDADDR=55:66:77:00:91:01)	Master: "smp" 55:66:77:00:91:01	8 bytes ...
1'827	3:04:09 PM.229 077 375	Empty LE Packets (x 854, 136 retries, 15.5 s)	Slave: "smp" 55:66:77:00:91:02	

Figure 11.19: Legacy Just Works 的抓包



此时按 Central 或 Peripheral 开发板上的 SW1 按键给开发板重新上电，由于 Central 在扫描到已经绑定过的 Peripheral 设备会自动回连（参考演示代码中 central\_auto\_connect 变量），可以看到灯灭之后又立刻亮了起来，但这一次蓝色灯没有亮，因为回连不走 Pair 流程，直接通过 LTK 走加密流程。

### 说明：

有时候，我们需要不停地跑配对流程，而不希望 Central 和 Peripheral 重新上电后回连跳过 Pair 流程，只需要将 Central 或 Peripheral 任意一方的 Bonding Flag 置位为 0 即可，这里给出两种方法（需要注意，由于此前的测试已经 Bonding 过了，所以使用下面方法演示时，要先擦除 SMP Storage 区域）：

- 方法 1：在初始化配置 Security Parameters 的同时将 Bonding Flag 置位为 0。以 Peripheral 为例（默认注释掉）：

```
blc_smp_setSecurityParameters_periph(Non_Bondable_Mode, 0, LE_Legacy_Pairing, 0, 0,  
↪ IO_CAPABILITY_NO_INPUT_NO_OUTPUT);
```

这里设置了 Peripheral 端的 bonding\_mode 为 Non-Bondable mode，其他设置为 Just Works 默认配置（Central 端同理）。

调用该 API 之后，在 Pair 时，会看到 Peripheral 回复的 Pairing Response 中的 Bonding Flags=0，这样 Central 和 Peripheral 就都不会做 Bonding，也就是不会将配对信息存在 Flash 中。当 Central 或 Peripheral 设备重启之后，他们仍然是“全新的”，并不“认识”彼此：

P...	Time	Item	Transmitter	Payl...
1	1:58:13 PM.015 847 375	Connectable ("smp" 55:66:77:00:91:02, Initiator "smp" 55:66:77:00:91:01, Scanner 01:33:66:66:6...	Master: "smp" 55:66:77:00:91:02	
3	1:58:13 PM.029 066 625	Connectable ("smp" 55:66:77:00:91:01, 2 Scanners, 20 s)	Master: "smp" 55:66:77:00:91:01	
420	1:58:15 PM.477 682 000	SMP Pairing Request (No Input No Output, Bonding, Int=EndKey   IdKey, Rsp=EndKey   IdKey)	Master: "smp" 55:66:77:00:91:01	7 bytes ...
421	1:58:15 PM.508 789 625	SMP Security Request (No Bonding)	Slave: "smp" 55:66:77:00:91:02	2 bytes ...
423	1:58:15 PM.509 298 250	SMP Pairing Response (No Input No Output, Bonding, Int=EndKey   IdKey, Rsp=EndKey   IdKey)	Slave: "smp" 55:66:77:00:91:02	7 bytes ...
430	1:58:15 PM.540 181 750	SMP Pairing Confirm (Ca=91E9A406:BDFAA83:1C6B:070:A38B9B53)	Master: "smp" 55:66:77:00:91:01	17 byte...
436	1:58:15 PM.571 201 875	SMP Pairing Confirm (Cb=7FCCC93C:1CC213CC:F92BB9F2:C47B3801)	Slave: "smp" 55:66:77:00:91:02	17 byte...
441	1:58:15 PM.602 682 375	SMP Pairing Random (Na=3CE7B8DF:8118ECAD:0HB014C9:926D7222)	Master: "smp" 55:66:77:00:91:01	17 byte...
446	1:58:15 PM.633 702 000	SMP Pairing Random (Nb=C0B8A439:48729E06:299ED4DC:A5D92E7D)	Slave: "smp" 55:66:77:00:91:02	17 byte...
452	1:58:15 PM.665 182 625	LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, SKDm=0x33B322B0D62E5A...)	Master: "smp" 55:66:77:00:91:01	23 byte...
459	1:58:15 PM.696 202 500	LLCP Encryption Response (SKDs=0x9FDDBCAB7D668F13A, IVs=0x3FE01BAD)	Slave: "smp" 55:66:77:00:91:02	13 byte...
475	1:58:15 PM.789 952 500	LLCP Start Encryption Request	Slave: "smp" 55:66:77:00:91:02	1 byte ...
481	1:58:15 PM.821 432 250	LLCP Start Encryption Response	Master: "smp" 55:66:77:00:91:01	1 byte ...
487	1:58:15 PM.852 452 375	LLCP Start Encryption Response	Slave: "smp" 55:66:77:00:91:02	1 byte ...
495	1:58:15 PM.914 952 375	SMP Encryption Information (LTK=95EDF16C:1D27:853:7CCB8189:F08C7B28)	Slave: "smp" 55:66:77:00:91:02	17 byte...
507	1:58:15 PM.977 452 625	SMP Master Identification (EDIV=0xB495, Rand=0x705DD20AD4BA7254)	Slave: "smp" 55:66:77:00:91:02	11 byte...
517	1:58:16 PM.039 952 625	SMP Identity Information (IRK=A3B086D9:459694C:F7A1C228E:6FD8D393)	Slave: "smp" 55:66:77:00:91:02	17 byte...
527	1:58:16 PM.102 452 750	SMP Identity Address Information (BDADDR=55:66:77:00:91:02)	Slave: "smp" 55:66:77:00:91:02	8 bytes ...
533	1:58:16 PM.133 933 375	SMP Encryption Information (LTK=69B2ED8A:D44D8F8:51E5419C:C7382777)	Master: "smp" 55:66:77:00:91:01	17 byte...
540	1:58:16 PM.165 183 500	SMP Master Identification (EDIV=0x8F06, Rand=0xA4EBBF830ED38FD)	Master: "smp" 55:66:77:00:91:01	11 byte...
547	1:58:16 PM.196 433 000	SMP Identity Information (IRK=A728C802:7FC864C:FF9555DC:87B7B5F8)	Master: "smp" 55:66:77:00:91:01	17 byte...
553	1:58:16 PM.227 683 250	SMP Identity Address Information (BDADDR=55:66:77:00:91:01)	Master: "smp" 55:66:77:00:91:01	8 bytes ...
554	1:58:16 PM.228 040 125	Empty LE Packets (x 524, 1 retry, 8.19 s)	Slave: "smp" 55:66:77:00:91:02	
182	1:58:28 PM.415 497 500	Connectable ("smp" 55:66:77:00:91:02, Scanner 01:33:66:66:66:66, 4.6 s)	Master: "smp" 55:66:77:00:91:02	

Figure 11.20: Legacy Just Works Peripheral NoBonding 的抓包

如果不调用该 API，默认会使用在 blc\_gap\_init() 中对 Peripheral 和 Central 设置 SecurityParameters 的初始值：



```

mode_level = Unauthenticated_Pairing_with_Encryption;
bond_mode = Bondable_Mode;
MITM_en = 0;
method = LE_Legacy_Pairing;
OOB_en = 0;
keyPress_en = 0;
ioCapablility = IO_CAPABILITY_NO_INPUT_NO_OUTPUT;
ecdh_debug_mode = non_debug_mode;
passKeyEntryDftTK = 0;

```

- 方法 2：在初始化配置 Security Parameters 的同时单独配置 Bonding Mode。以 Central 为例：

```
blc_smp_setBondingMode_central(Non_Bondable_Mode);
```

这里设置了 Central 端的 bonding\_mode 为 Non-Bondable mode (Peripheral 端同理)。抓包可以看到效果和上述方法 1 相同：

P...	Time	Item	Transmitter	Payl...
2	3:37:25 PM.018 833 750	Connectable ("smp" 55:66:77:00:91:02, Initiator "smp" 55:66:77:00:91:01, 3 Scanners, 6.48 s)	Master: "smp" 55:66:77:00:91:02	
143	3:37:25 PM.693 662 750	Connectable ("smp" 55:66:77:00:91:01, 2 Scanners, 26 s)	Master: "smp" 55:66:77:00:91:01	
1'912	3:37:31 PM.514 784 000	SMP Pairing Request (No Input No Output, No Bonding, Int=EndKey   IdKey, Rsp=EndKey   IdKey)	Master: "smp" 55:66:77:00:91:01	7 bytes ...
1'913	3:37:31 PM.515 102 500	SMP Security Request (Bonding)	Slave: "smp" 55:66:77:00:91:02	2 bytes ...
1'923	3:37:31 PM.545 803 500	SMP Pairing Response (No Input No Output, Bonding, Int=EndKey   IdKey, Rsp=EndKey   IdKey)	Slave: "smp" 55:66:77:00:91:02	7 bytes ...
1'933	3:37:31 PM.577 284 125	SMP Pairing Confirm (Ca=56D7895B:747F5881:E5879A2B:3EC8DC6E)	Master: "smp" 55:66:77:00:91:01	17 bytes ...
1'945	3:37:31 PM.608 304 125	SMP Pairing Confirm (Cb=42FCAC5D:F4C89E48:F56009EF:8C49524C)	Slave: "smp" 55:66:77:00:91:02	17 bytes ...
1'953	3:37:31 PM.639 783 625	SMP Pairing Random (Ra=D44F9EAF:0E664E21:FA720B29:573DC600)	Master: "smp" 55:66:77:00:91:01	17 bytes ...
1'960	3:37:31 PM.670 804 250	SMP Pairing Random (Nb=9A27A076:B0C99A89:2FB268C5:533EB5F6)	Slave: "smp" 55:66:77:00:91:02	17 bytes ...
1'968	3:37:31 PM.702 283 750	LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, SKDm=0x6E54C2DDC7142F...)	Master: "smp" 55:66:77:00:91:01	23 bytes ...
1'977	3:37:31 PM.733 303 750	LLCP Encryption Response (SKDs=0x5343A448A04C40C, IVs=0x1961039E)	Slave: "smp" 55:66:77:00:91:02	13 bytes ...
2'006	3:37:31 PM.827 054 125	LLCP Start Encryption Request	Slave: "smp" 55:66:77:00:91:02	1 byte (...)
2'015	3:37:31 PM.858 534 250	LLCP Start Encryption Response	Master: "smp" 55:66:77:00:91:01	1 byte (...)
2'025	3:37:31 PM.889 554 250	LLCP Start Encryption Response	Slave: "smp" 55:66:77:00:91:02	1 byte (...)
2'045	3:37:31 PM.952 054 375	SMP Encryption Information (LTK=C72F523:E59CCFDC:7AE73D90:066BE0A3)	Slave: "smp" 55:66:77:00:91:02	17 bytes ...
2'064	3:37:32 PM.014 554 125	SMP Master Identification (EDIV=0x0120, Rand=0x612AFD78AD76F044)	Slave: "smp" 55:66:77:00:91:02	11 bytes ...
2'087	3:37:32 PM.077 054 375	SMP Identity Information (IRK=A3B86D9:459694CF:7A1C228E:6FD8D393)	Slave: "smp" 55:66:77:00:91:02	17 bytes ...
2'099	3:37:32 PM.139 554 625	SMP Identity Address Information (@DADDR=55:66:77:00:91:02)	Slave: "smp" 55:66:77:00:91:02	8 bytes ...
2'109	3:37:32 PM.171 034 750	SMP Encryption Information (LTK=11ACBFA:5B131B74:AF275E7C:02689355)	Master: "smp" 55:66:77:00:91:01	17 bytes ...
2'120	3:37:32 PM.202 285 125	SMP Master Identification (EDIV=0x205A, Rand=0x1017CF6471C819AE)	Master: "smp" 55:66:77:00:91:01	11 bytes ...
2'133	3:37:32 PM.233 534 875	SMP Identity Information (IRK=A7C802:7FC864DC:FF9555DC:87B7B5F8)	Master: "smp" 55:66:77:00:91:01	17 bytes ...
2'143	3:37:32 PM.264 784 875	SMP Identity Address Information (@DADDR=55:66:77:00:91:01)	Master: "smp" 55:66:77:00:91:01	8 bytes ...
2'144	3:37:32 PM.265 142 000	Empty LE Packets (x 1143, 101 retries, 19.4 s)	Slave: "smp" 55:66:77:00:91:02	
6'897	3:37:48 PM.811 728 875	Connectable ("smp" 55:66:77:00:91:02, Scanner 01:33:66:66:66:66, 2.89 s)	Master: "smp" 55:66:77:00:91:02	

设备重新上电没有自动回连

Figure 11.21: Legacy Just Works Central NoBonding 的抓包

### 11.6.3 Secure Connections Just Works

Central 和 Peripheral 使能 SMP 功能，以 Secure Connections Just Works 方式配对的演示。该演示需在 feature\_smp/app.c 中定义：



#define SMP\_TEST\_MODE

SMP\_TEST\_LESC\_JW

编译，分别擦除 Flash 后烧录到两个开发板中。按其中一个开发板（作为 Central）的按键 SW4（启动连接），连接成功后，可以看到 Central 亮绿色、白色、蓝色灯，Peripheral 亮红色、白色、蓝色灯，代表二者 Pair 成功，抓包如下：

Low Energy Overview			
P...	Time	Item	Transmitter
1	2:17:00 PM.001 633 375	Connectable ("smp" 55:66:77:00:91:01, 4 Scanners, 1.2 min)	Master: "smp" 55:66:77:00:91:01
3'495	2:17:12 PM.025 461 500	Connectable ("smp" 55:66:77:00:91:02, 6 Scanners, 1.22 min)	Master: "smp" 55:66:77:00:91:02
25'580	2:18:11 PM.751 339 875	SMP Pairing Request (No Input No Output, Bonding, SC, Int=EncKey   IdKey, Rsp=EncKey   IdKey)	Master: "smp" 55:66:77:00:91:02
25'581	2:18:11 PM.751 659 875	SMP Security Request (Bonding, SC)	Slave: "smp" 55:66:77:00:91:01
25'594	2:18:11 PM.782 359 625	SMP Pairing Response (No Input No Output, Bonding, SC, Int=IdKey, Rsp=IdKey)	Slave: "smp" 55:66:77:00:91:01
25'616	2:18:11 PM.845 089 625	SMP Pairing Public Key (X=6E68CAA5:2BCC737:SDC7C2B4:F7FC2EDC:02EC260:1F4EE52A:C1FC...)	Master: "smp" 55:66:77:00:91:02
25'635	2:18:11 PM.876 109 375	SMP Pairing Public Key (Debug Key, X=20B003D2:F297BE2C:5E2C83A7:E9F9A5B9:EFF49111:ACF4...)	Slave: "smp" 55:66:77:00:91:01
25'641	2:18:11 PM.878 042 875	SMP Pairing Confirm (Cba=A2A0CA40:512D4211:1DF9CA9E:C744EE20)	Slave: "smp" 55:66:77:00:91:01
25'655	2:18:11 PM.907 589 375	SMP Pairing Random (Na=768C649:E9C5B314:44B86679:A7E60AEF)	Master: "smp" 55:66:77:00:91:02
25'677	2:18:11 PM.969 859 125	SMP Pairing Random (Nb=F496717A:21FEF931:801D51AC:A12B2B17)	Slave: "smp" 55:66:77:00:91:01
25'698	2:18:12 PM.032 588 750	SMP Pairing DH Key Check (C8B199E9:7612DCB8:34885696:448A6BA)	Master: "smp" 55:66:77:00:91:02
25'710	2:18:12 PM.063 608 500	SMP Pairing DH Key Check (18D60413:878E4884:D1F5CASA:5AC29FD8)	Slave: "smp" 55:66:77:00:91:01
25'723	2:18:12 PM.095 088 625	LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, SKDm=0x0D67C70CDF6DB1...)	Master: "smp" 55:66:77:00:91:02
25'734	2:18:12 PM.126 108 500	LLCP Encryption Response (SKDs=0x6DC1B2627C637CB7, IVs=0xC6688A13)	Slave: "smp" 55:66:77:00:91:01
25'768	2:18:12 PM.219 858 125	LLCP Start Encryption Request	Slave: "smp" 55:66:77:00:91:01
25'777	2:18:12 PM.251 338 000	LLCP Start Encryption Response	Master: "smp" 55:66:77:00:91:02
25'790	2:18:12 PM.282 358 000	LLCP Start Encryption Response	Slave: "smp" 55:66:77:00:91:01
25'812	2:18:12 PM.344 857 375	SMP Identity Information (IRK=A728C802:7FC864DC:FF955DC:87B7B5F8)	Slave: "smp" 55:66:77:00:91:01
25'830	2:18:12 PM.407 357 250	SMP Identity Address Information (@IDADDR=55:66:77:00:91:01)	Slave: "smp" 55:66:77:00:91:01
25'846	2:18:12 PM.438 837 125	SMP Identity Information (IRK=A3B086D9:459694CF:7A1C228E:6FD8D393)	Master: "smp" 55:66:77:00:91:02
25'855	2:18:12 PM.470 087 375	SMP Identity Address Information (@IDADDR=55:66:77:00:91:02)	Master: "smp" 55:66:77:00:91:02
25'856	2:18:12 PM.470 445 000	Empty LE Packets (x 785, 13 retries, 12.5 s)	Slave: "smp" 55:66:77:00:91:01

Figure 11.22: SC Just Works 的抓包

关于 Debug Mode 参考[SMP 章节](#)，用户可以根据需求配置为如下三种：

Central	Peripheral	Description
disabled	enabled	演示代码默认配置
enabled	disabled	效果与演示代码类似
disabled	disabled	抓包工具无法自动解析加密包

### 注意：

Peripheral debug mode enable + Central debug mode enable 的组合是不允许的，会出现 SMP Pairing Failed 事件：



P...	Time	Item	Transmitter	Payl...
5	2:44:54 PM.031 107 250	④ Connectable ("smp" 55:66:77:00:91:01, Initiator "smp" 55:66:77:00:91:02, 7 Scanners, 17.1 s)	Master: "smp" 55:66:77:00:91:01	
908	2:44:57 PM.293 257 000	④ Connectable ("smp" 55:66:77:00:91:02, 5 Scanners, 21.4 s)	Master: "smp" 55:66:77:00:91:02	
5'933	2:45:11 PM.136 815 750	④ SMP Pairing Request (No Input No Output, Bonding, SC, Int=EncKey   IdKey, Rsp=EncKey   IdKey)	Master: "smp" 55:66:77:00:91:02	7 bytes ...
5'934	2:45:11 PM.137 133 250	④ SMP Security Request (Bonding, SC)	Slave: "smp" 55:66:77:00:91:01	2 bytes ...
5'943	2:45:11 PM.167 835 500	④ SMP Pairing Response (No Input No Output, Bonding, SC, Int=IdKey, Rsp=IdKey)	Slave: "smp" 55:66:77:00:91:01	7 bytes ...
5'957	2:45:11 PM.199 315 625	④ SMP Pairing Public Key (Debug Key, X=20B003D2:F297BE2C:5E2C83A7:E9F9A5B9:EFF49111:ACF4FDDB:...)	Master: "smp" 55:66:77:00:91:02	65 byte...
5'977	2:45:11 PM.230 335 375	④ SMP Pairing Public Key (Debug Key, X=20B003D2:F297BE2C:5E2C83A7:E9F9A5B9:EFF49111:ACF4FDDB:...)	Slave: "smp" 55:66:77:00:91:01	65 byte...
5'983	2:45:11 PM.232 264 500	④ SMP Pairing Confirm (Cb=33A2614E:2B77C2B7:C295C14D:B191735E)	Slave: "smp" 55:66:77:00:91:01	17 byte...
5'993	2:45:11 PM.261 815 500	④ SMP Pairing Failed (Invalid Parameters)	Master: "smp" 55:66:77:00:91:02	2 bytes ...
5'995	2:45:11 PM.262 323 625	④ SMP Pairing Failed (Unspecified Reason)	Master: "smp" 55:66:77:00:91:02	2 bytes ...
5'996	2:45:11 PM.262 600 625	④ Empty LE Packets (x 472, 1 retry, 7.41 s)	Slave: "smp" 55:66:77:00:91:01	

Figure 11.23: Both DebugMode Pair Failed 的抓包

#### 11.6.4 Legacy Passkey Entry Input

Central 和 Peripheral 使能 SMP 功能，以 Legacy Passkey Entry 配对方式，通过 Input 方法实现的演示。这里演示的 Input 是 Keyboard Input，使用该功能需在 feature\_smp/app.c 中定义：

```
#define SMP_TEST_MODE      SMP_TEST_LEGACY_PKI
```

演示代码中 Initiator 默认设置 Passkey 为 123456，Responder 在收到 GAP\_EVT\_SMP\_TK\_REQUEST\_PASSKEY 事件 10s 内，自动输入 123456 的 Passkey。

编译，分别擦除 Flash 后烧录到两个开发板中。按其中一个开发板（作为 Central）的按键 SW4（启动连接），连接成功后，可以看到 Central 亮绿色，Peripheral 亮红色，停留几秒后二者的白色、蓝色灯相继亮起，代表二者 Pair 成功，抓包如下（框起部分为等待 Peripheral 输入 Passkey）：

P...	Time	Item	Transmitter	Receiver
1	7:38:56 AM.017 320 250	④ Connectable ("smp" 55:66:77:00:91:02, Initiator "smp" 55:66:77:00:91:01, 4.99 s)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
22	7:38:56 AM.249 404 500	④ Connectable ("smp" 55:66:77:00:91:01, 20.4 s)	Master: "smp" 55:66:77:00:91:01	Slave: "Scanning Device"
829	7:39:01 AM.031 142 625	④ SMP Pairing Request (Display Only, Bonding, MITM, Int=EncKey   IdKey, Rsp=EncKey   IdKey)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
830	7:39:01 AM.031 461 500	④ SMP Security Request (Bonding, MITM)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
834	7:39:01 AM.062 164 125	④ SMP Pairing Response (Keyboard Only, Bonding, MITM, Int=EncKey   IdKey, Rsp=EncKey   IdKey)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
839	7:39:01 AM.093 642 125	④ SMP Pairing Confirm (Cb=0BA90E0F6:59A1A768:F639C5D7:022A7D7C)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
840	7:39:01 AM.094 040 000	④ Empty LE Packets (x 412, 6.44 s)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'897	7:39:07 AM.530 919 375	④ SMP Pairing Confirm (Cb=62554804:0B1CC38D:519B6ED2:D46C7112)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'903	7:39:07 AM.562 397 750	④ SMP Pairing Random (Ra=40AF1741:FD2B700C:97FCE6D8:085D9B5B)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'909	7:39:07 AM.593 419 875	④ SMP Pairing Random (Ra=F330447D:EEED74E5:C434F3A6:F395063E)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'916	7:39:07 AM.624 897 625	④ LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, SKDm=0x6EB75150229EC31, I...	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'923	7:39:07 AM.655 919 625	④ LLCP Encryption Response (SKDs=0xB056FB7DD5D292E, IVs=0x90B6AF02)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'938	7:39:07 AM.749 669 875	④ LLCP Start Encryption Request	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'944	7:39:07 AM.781 148 000	④ LLCP Start Encryption Response	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'950	7:39:07 AM.812 169 750	④ LLCP Start Encryption Response	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'960	7:39:07 AM.874 670 000	④ SMP Encryption Information (TK=A6651128:B88821B0:9161A6F3:A6C0536B)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'972	7:39:07 AM.937 170 000	④ SMP Master Identification (EDIV=0xF6F73, Rand=0x15DF9B2BD156054B)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'983	7:39:07 AM.999 670 000	④ SMP Identity Information (IRK=A3B086D9:459694CF:7A1C228E:6FD8D393)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'992	7:39:08 AM.062 170 000	④ SMP Identity Address Information (BDADDR=55:66:77:00:91:02)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'999	7:39:08 AM.093 648 125	④ SMP Encryption Information (TK=15FA4214:A87E2559:C2A9338E:5D08D80E)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'007	7:39:08 AM.124 898 500	④ SMP Master Identification (EDIV=0x246F, Rand=0x0ADF331C9D0D870E0)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'014	7:39:08 AM.156 148 500	④ SMP Identity Information (IRK=A728C802:7FC864DC:FF9555DC:87B7B5F8)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'021	7:39:08 AM.187 398 250	④ SMP Identity Address Information (BDADDR=55:66:77:00:91:01)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'022	7:39:08 AM.187 755 625	④ Empty LE Packets (x 541, 8.44 s)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01

Figure 11.24: Legacy Passkey Entry CDPI 的抓包

代码中在初始化时使用 blc\_smp\_setDefaultPinCode/\_periph/\_central() 来设置 Display 设备的默认 Passkey，



这里设置为 123456。如果没有调用该语句设置默认 Passkey，系统会随机生成一个十进制 000000-999999 的 6 位 Passkey。

此后通过 Input 设备在 main\_loop 中轮询 blc\_smp\_isWaitingToSetTK() 来获取当前是否在等待输入 Passkey，当获取到该需求后，通过 blc\_smp\_setTK\_by\_PasskeyEntry() 来设置 Passkey，应与 Display 设备设置的默认 Passkey 相等。需要注意，在没有获取到 blc\_smp\_isWaitingToSetTK() 的需求前调用 blc\_smp\_setTK\_by\_PasskeyEntry() 设置的 Passkey 是无效的。

另外，在 app\_host\_event\_callback() 中，当获取到 GAP\_EVT\_SMP\_TK\_REQUEST\_PASSKEY GAP 事件时调用 blc\_smp\_setTK\_by\_PasskeyEntry() 也可以成功设置 Input 设备的 Passkey。

### 11.6.5 Legacy Passkey Entry Display

Central 和 Peripheral 使能 SMP 功能，以 Legacy Passkey Entry 配对方式，通过 Display 方法实现的演示。Display 与上一个演示 Input 类似，只是显示和输入的设备角色互换了。使用该功能需在 feature\_smp/app.c 中定义：

```
#define SMP_TEST_MODE           SMP_TEST_LEGACY_PKD
```

演示代码中 Initiator 通过 UART 输入 Passkey，Responder 通过底层随机生成 Passkey 并显示出来，更加贴合实际使用需求。

编译，分别擦除 Flash 后烧录到两个开发板中。给其中一个开发板（作为 Central）的 UART 口 PA3(Tx)，PA4(Rx) 接到电脑串口上，配置波特率 115200 8N1。按 Central 开发板的按键 SW4（启动连接），连接成功后，可以看到 Central 亮绿色灯，Peripheral 亮红色灯，此时 Peripheral 的 Log 输出“TK Display”，将输出的数填入到 Central 的 UART 串口，SDK 默认超时为 30s，须在该时间内将 Passkey 输入成功，可以看到 Central 的 UART 串口返回该值，Log 输出“TK set”后面跟着用户通过 UART 输入的值。SMP 流程继续，白色、蓝色灯相继亮起，代表二者 Pair 成功，抓包如下（框起部分为等待 Central 输入 Passkey）：



P...	Time	Item	Transmitter	Receiver
1	6:08:39 PM.012 992 500	Connectable ("smp" 55:66:77:00:91:01, Scanner 70:DC:AB:F7:A7:FC (Resolvable), 28.8 s)	Master: "smp" 55:66:77:00:91:01	Slave: "Scanning Device"
4	6:08:39 PM.018 719 375	Connectable ("smp" 55:66:77:00:91:02, Initiator "smp" 55:66:77:00:91:01, Scanner 70:DC:AB:F7:A7:FC (Resolvable), 28.8 s)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
750	6:08:43 PM.297 228 625	SMP Pairing Request (Keyboard Only, Bonding, MITM, Int=EndKey   IdKey, Rsp=EndKey   IdKey)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
751	6:08:43 PM.297 545 250	SMP Security Request (Bonding, MITM)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
756	6:08:43 PM.328 250 375	SMP Pairing Response (Keyboard Display, Bonding, MITM, Int=EndKey   IdKey, Rsp=EndKey   IdKey)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
761	6:08:43 PM.359 271 375	Empty LE Packets (x 924, 4 retries, 14.4 s)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
3'177	6:08:57 PM.797 238 375	SMP Pairing Confirm (Ca=C626E58E:2642B915:83B24D2C:0CD3E7A)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
3'183	6:08:57 PM.828 260 375	SMP Pairing Confirm (Cb=17978F39:639F692C:532D8687:A0AB2DA6)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
3'189	6:08:57 PM.859 739 000	SMP Pairing Random (Na=CC95CAF:1EE76639:5DEF99D5:8B41F3D5)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
3'195	6:08:57 PM.890 760 625	SMP Pairing Random (Nb=5178A0CC:F8A85D18:D91318D1:FA21786B)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
3'201	6:08:57 PM.922 238 625	LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, SKDm=0x7EA7876E7735A3C3, I...	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
3'207	6:08:57 PM.953 260 875	LLCP Encryption Response (SKDs=0xC84B63F3758AD2, IVs=0x2F59F10D)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
3'224	6:08:58 PM.047 011 250	LLCP Start Encryption Request	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
3'230	6:08:58 PM.078 489 125	LLCP Start Encryption Response	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
3'236	6:08:58 PM.109 511 125	LLCP Start Encryption Response	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
3'246	6:08:58 PM.172 010 875	SMP Encryption Information (LTK=042DF599:ADFD084D:8C464D84:AF742D3E)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
3'257	6:08:58 PM.234 511 000	SMP Master Identification (EDIV=0x7B41, Rand=0xFB223EEA324C7128)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
3'266	6:08:58 PM.297 011 250	SMP Identity Information (IRK=A3B086D9:459694CF:7A1C228E:6FD8D393)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
3'276	6:08:58 PM.359 511 250	SMP Identity Address Information (BDADDR=55:66:77:00:91:02)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
3'282	6:08:58 PM.390 989 250	SMP Encryption Information (LTK=99C09FAA:4BB2336C:08BACC80:DE14A680)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
3'290	6:08:58 PM.422 239 000	SMP Master Identification (EDIV=0xCA34, Rand=0x2C29077867672F5D4)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
3'298	6:08:58 PM.453 489 500	SMP Identity Information (IRK=A728C802:7FC864DC:FF9555DC:87B7B5F8)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
3'305	6:08:58 PM.484 739 375	SMP Identity Address Information (BDADDR=55:66:77:00:91:01)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
3'306	6:08:58 PM.485 096 500	Empty LE Packets (x 598, 9.34 s)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01

Figure 11.25: Legacy Passkey Entry CIPD 的抓包

### 11.6.6 Legacy OOB

Central 和 Peripheral 使能 SMP 功能，以 Legacy OOB 方式配对的演示。根据蓝牙协议，在 Legacy 配对方式下，当双方均支持 OOB，拥有对方 OOB 数据时，会使用 Legacy OOB 方式进行匹配。使用该功能需在 feature\_smp/app.c 中定义：

```
#define SMP_TEST_MODE      SMP_TEST_LEGACY_OOB
```

这里选择用 Manually 方式通过 UART 输入 OOB data。

编译，分别擦除 Flash 后烧录到两个开发板中。给两个开发板的 UART 口 PA3(Tx)，PA4(Rx) 接到电脑串口上，配置波特率 115200 8N1。按其中一个开发板（作为 Central）的按键 SW4（启动连接），连接成功后，可以看到 Central 亮绿色灯，Peripheral 亮红色灯。此时从双方的 Log 中看到“OOB request”，在 30s 内将自定义的 OOB data 分别填入到 Peripheral 和 Central 的 UART 串口。可以看到二者的白色、蓝色灯相继亮起，代表二者 Pair 成功，抓包如下：



P...	Time	Item	Transmitter	Receiver
1	4:03:11 PM.000 834 125	Connectable ("smp" 55:66:77:00:91:01, Scanner 74:85:51:5E:74:F3 (Resolvable), 16.6 s)	Master: "smp" 55:66:77:00:91:01	Slave: "Scanning Device"
4	4:03:11 PM.009 979 250	Connectable ("smp" 55:66:77:00:91:02, Initiator "smp" 55:66:77:00:91:01, 5.84 s)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
139	4:03:11 PM.785 103 375	LLCP Reserved (0x58)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
608	4:03:14 PM.441 873 250	Unknown LE Transfer (Reserved (0x9D))	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'032	4:03:16 PM.875 538 125	SMP Pairing Request (Keyboard Only, OOB, Bonding, MITM, Int=EndKey   IdKey, Rsp=EndKey   IdKey)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'033	4:03:16 PM.875 857 250	SMP Security Request (Bonding, MITM)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'037	4:03:16 PM.937 809 375	SMP Pairing Response (Keyboard Display, OOB, Bonding, MITM, Int=EndKey   IdKey, Rsp=EndKey   IdKey)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'045	4:03:16 PM.968 830 750	Empty LE Packets (x 385, 19 retries, 6.06 s)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'080	4:03:23 PM.031 790 125	SMP Pairing Confirm (Ca=EEF10862:0827386A:9512209D:CF97D08D)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'087	4:03:23 PM.094 290 000	SMP Pairing Confirm (Cb=4C99C047:A392796D:5DA55C8B:E9116B85)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'098	4:03:23 PM.125 311 875	SMP Pairing Random (Nb=00BC5647:B83C0763:890213B3:3B67CC17)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'104	4:03:23 PM.156 790 250	LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, SKDm=0x790CA9C40652ED1B, I...	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'110	4:03:23 PM.187 812 000	LLCP Encryption Response (SKDs=0x3290993DC771A965, IVs=0xD2D8DDEF)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'127	4:03:23 PM.281 562 000	LLCP Start Encryption Request	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'131	4:03:23 PM.312 582 750	Empty LE Packets (x 2, 80 us)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'133	4:03:23 PM.313 039 750	Encrypted ACL Link Layer Traffic (x 8, 4.22 s)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02

Figure 11.26: Legacy OOB 的抓包

### 11.6.7 Secure Connections Passkey Entry

Central 和 Peripheral 使能 SMP 功能，以 Secure Connections Passkey Entry 方式配对的演示。与 Legacy 对应，Secure Connections 的 Passkey Entry 匹配方式也有 Input 和 Display 两种实现方式。

- 对于 Input，需在 feature\_smp/app.c 中定义：

```
#define SMP_TEST_MODE           SMP_TEST_LESC_PKI
```

编译，分别擦除 Flash 后烧录到两个开发板中。给其中一个开发板（作为 Peripheral）的 UART 口 PA3(Tx)，PA4(Rx) 接到电脑串口上，配置波特率 115200 8N1。按 Central 开发板的按键 SW4（启动连接），连接成功后，可以看到 Central 亮绿色灯，Peripheral 亮红色灯。此时 Central 的 Log 输出“TK display”，将输出的数在 30s 内填入到 Peripheral 的 UART 串口，可以看到 UART 串口返回输入值，Log 显示“TK set”后面跟着输入的 passkey，SMP 流程继续，白色、蓝色灯相继亮起，代表二者 Pair 成功，抓包如下（高亮部分为等待 Peripheral 输入 Passkey）：



P...	Time	Item	Transmitter	Receiver
1	6:25:23 PM.000 271 000	⊕  Connectable ("smp" 55:66:77:00:91:02, Scanner 72:C0:97:F5:A6:85 (Resolvable), 1.43 min)	Master: "smp" 55:66:77:00:91:02	Slave: "Scanning Device"
5	6:25:23 PM.025 245 875	⊕  Connectable ("smp" 55:66:77:00:91:01, Initiator "smp" 55:66:77:00:91:02, Scanner 72:C0:97:F5:A6:8...	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'580	6:25:31 PM.296 809 000	⊕  SMP Pairing Request (Display Only, Bonding, MITM, SC, Int=EncKey   IdKey, Rsp=EncKey   IdKey)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
1'581	6:25:31 PM.297 129 000	⊕  SMP Security Request (Bonding, MITM, SC)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
1'591	6:25:31 PM.359 080 375	⊕  SMP Pairing Response (Keyboard Only, Bonding, MITM, SC, Int=IdKey, Rsp=IdKey)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
1'597	6:25:31 PM.390 558 250	⊕  SMP Pairing Public Key (Debug Key, X=20B003D2:F297BE2C:5E2C83A7:E9F9A5B9:EFF49111:ACF4FDD...	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
1'607	6:25:31 PM.421 580 125	⊕  SMP Pairing Public Key (X=8BA53AD8:1E3DC3D2:97582578:E92343AA:99454744:971883AB:6D5E7A7C...	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
1'616	6:25:31 PM.453 058 125	⊕  SMP Pairing Confirm (Ca=73E18614:E690315D:097B644E:0F7E5477)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
1'617	6:25:31 PM.453 455 500	⊕  Empty LE Packets (x 454, 7.09 s)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'813	6:25:38 PM.546 547 000	⊕  SMP Pairing Confirm (Cb=1C9167F8:D7C28965:B11F13CF:93693F82)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'819	6:25:38 PM.578 024 875	⊕  SMP Pairing Random (Na=11C771C1:9B8C749A:5:9D422608:E9EBE503)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
2'830	6:25:38 PM.640 296 625	⊕  SMP Pairing Random (Nb=3EDDF357:62A33D07:8EEDC0B5:72EBAEDA)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'842	6:25:38 PM.703 024 125	⊕  SMP Pairing Confirm (Ca=3A77F3E1:464B2D75:99A0C7D8:093BFC94)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
2'848	6:25:38 PM.734 045 875	⊕  SMP Pairing Confirm (Cb=4E3FCC81:3BD3E1AF:DE51995E:DB16FDA)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'855	6:25:38 PM.765 524 375	⊕  SMP Pairing Random (Na=085460E2:ACEEDB19:D338FCAB:4A9EE62C)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
2'862	6:25:38 PM.796 545 625	⊕  SMP Pairing Random (Nb=F751F547:3EF4E62C:55FC6872:F2C09816)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'868	6:25:38 PM.828 023 500	⊕  SMP Pairing Confirm (Ca=C265DE01:8B8A1848:1F20391A:A73E55C0)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
2'874	6:25:38 PM.859 045 375	⊕  SMP Pairing Confirm (Cb=E98E2EA3:0F7C407F:3759FC6E:C12D0122)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'880	6:25:38 PM.890 523 375	⊕  SMP Pairing Random (Na=56CFCAF11:F9BB8D03:48288261:0944237F)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
2'887	6:25:38 PM.921 545 125	⊕  SMP Pairing Random (Nb=3A861EAA:3FC00EB7:C9E63736:422DB8A7)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'893	6:25:38 PM.953 022 875	⊕  SMP Pairing Confirm (Ca=A6823F39:1:45FAA8:B36AE266:01B2D088)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
2'899	6:25:38 PM.984 044 875	⊕  SMP Pairing Confirm (Cb=513383D0:1E64CCAC:A08AB92F:4E11E6E5)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'905	6:25:39 PM.015 522 750	⊕  SMP Pairing Random (Na=6596634E:DF6668C2:22EA8AC9:DDBCB86AB)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
2'911	6:25:39 PM.046 544 625	⊕  SMP Pairing Random (Nb=52F1015A:913CB0AC:9A249E85:7C3377F0)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'917	6:25:39 PM.078 022 625	⊕  SMP Pairing Confirm (Ca=D18328F5:70557D4B:B803E4C0:9875658B)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
2'923	6:25:39 PM.109 044 125	⊕  SMP Pairing Confirm (Cb=AE80A5C8:04F63B86:BD014FA4:EC26E6CA)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'929	6:25:39 PM.140 522 250	⊕  SMP Pairing Random (Na=7EE38584:2CA71F81:A6376237:039AB495)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
2'935	6:25:39 PM.171 544 000	⊕  SMP Pairing Random (Nb=67374C69:4E8B7EPD:2A90AB68:3C075F51)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'941	6:25:39 PM.203 021 875	⊕  SMP Pairing Confirm (Ca=718A1F43:0EE8E9E4:F6DEC7E:09D8E0A6)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01

Figure 11.27: SC Passkey Entry CDPI 的抓包

- 对于 Display，需在 feature\_smp/app.c 中定义：

```
#define SMP_TEST_MODE           SMP_TEST_LESC_PKD
```

编译，分别擦除 Flash 后烧录到两个开发板中。按其中一个开发板（作为 Central）的按键 SW4（启动连接），连接成功后，可以看到 Central 亮绿灯，Peripheral 亮红灯。停留几秒后二者的白色、蓝色灯相继亮起，代表二者 Pair 成功，抓包如下（框起部分为等待 Central 输入 Passkey）：



P...	Time	Item	Transmitter	Receiver
1	6:47:54 PM.000 102 250	⊕  Connectable ("smp" 55:66:77:00:91:02, Initiator "smp" 55:66:77:00:91:01, Scanner 76:E7:D0:69:66:D...)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
637	6:48:04 PM.189 301 750	⊕  Connectable ("smp" 55:66:77:00:91:01, 12.7 s)	Master: "smp" 55:66:77:00:91:01	Slave: "Scanning Device"
1'528	6:48:04 PM.654 787 875	⊕  SMP Pairing Request (Keyboard Only, Bonding, MITM, SC, Int=EndKey   IdKey, Rsp=EndKey   IdKey)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'529	6:48:04 PM.655 104 625	⊕  SMP Security Request (Bonding, MITM, SC)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'540	6:48:04 PM.717 059 875	⊕  SMP Pairing Response (Keyboard Display, Bonding, MITM, SC, Int=IdKey, Rsp=IdKey)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'546	6:48:04 PM.748 538 000	⊕  SMP Pairing Public Key (Debug Key, X=20B003D2:F297BE2C:5E2C83A7:E9F9A5B9:EF49111:ACF4FDD...)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'558	6:48:04 PM.779 559 750	⊕  SMP Pairing Public Key (X=BC55B8C9:99D88FA3:6FCC97B9:9E23638F:2A5EB048:515891C1:50FBDF66...)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'567	6:48:04 PM.810 580 625	⊕  Empty LE Packets (x 90, 1.38 s)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'823	6:48:06 PM.186 039 000	⊕  SMP Pairing Confirm (Ca=DE1D9ABC:235E1745:338774AC:8B86E4B6E)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'830	6:48:06 PM.217 061 000	⊕  SMP Pairing Confirm (Cb=15730C94:FF5E39FA:B38D04B8:EACE17E5)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'837	6:48:06 PM.248 539 250	⊕  SMP Pairing Random (Na=D14D480F:1A4427A4:8A997B60:333F492B)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'850	6:48:06 PM.310 810 500	⊕  SMP Pairing Random (Nb=A24E7FA0:C5EE0CBD:EDA7EF0C:97DC19A9)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'863	6:48:06 PM.373 539 250	⊕  SMP Pairing Confirm (Ca=6291D2EC:DFC1D5AE:7CF8E5F:56DCA82E)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'869	6:48:06 PM.404 560 875	⊕  SMP Pairing Confirm (Cb=9868FE16:6DD927E:30258FB7:2F2AE1A1A)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'876	6:48:06 PM.436 039 000	⊕  SMP Pairing Random (Na=070C6FF6:36FD84C6:54B6757E:FE6EA972)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'884	6:48:06 PM.467 061 000	⊕  SMP Pairing Random (Nb=E3306A36:252CD99E:6C9F0839:51763F41)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'891	6:48:06 PM.498 538 750	⊕  SMP Pairing Confirm (Ca=DA693016:B3749A74:A24F97D5:39A13AC2)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'897	6:48:06 PM.529 561 000	⊕  SMP Pairing Confirm (Cb=7AD6378C:2F1827A:8B4C51AD:1D2D1A10)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'904	6:48:06 PM.561 039 000	⊕  SMP Pairing Random (Na=BC690F50:0761CA8D:EAE58958:3A6BEAD)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'911	6:48:06 PM.592 060 875	⊕  SMP Pairing Random (Nb=DE678F41:84A576A5:A9A0EF9F:1DDC7A7E)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'918	6:48:06 PM.623 539 250	⊕  SMP Pairing Confirm (Ca=30F2EE32:23AFD266:BF0840E4:C93080B1)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'925	6:48:06 PM.654 561 250	⊕  SMP Pairing Confirm (Cb=64388E1D:176E49A0:9EC34968:1859D49F)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'931	6:48:06 PM.686 039 250	⊕  SMP Pairing Random (Na=92A54638:EB38871:8FF53688:2072EEA8)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'938	6:48:06 PM.717 060 875	⊕  SMP Pairing Random (Nb=5AE201FA:2BCF6202:8726A6F4:A0958BD3)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'945	6:48:06 PM.748 539 500	⊕  SMP Pairing Confirm (Ca=E5A7E4F2:3964BADD:ECC3B259:4325589F)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'951	6:48:06 PM.779 561 000	⊕  SMP Pairing Confirm (Cb=81125325:0EA8811:EA9EEE:859F8AC3)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'957	6:48:06 PM.811 039 375	⊕  SMP Pairing Random (Na=A0F78DFA:BEEBA44F:D84F15FB:8A8B7FD9)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'963	6:48:06 PM.842 061 375	⊕  SMP Pairing Random (Nb=4C63A979:58FD8AC0:B3C0D6CE:794E1D16)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'970	6:48:06 PM.873 539 375	⊕  SMP Pairing Confirm (Ca=49CD6D9A:39DEC335:83BCA421:936B1FD6)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02

Figure 11.28: SC Passkey Entry CIPD 的抓包

### 11.6.8 Secure Connections Numeric Comparison

Central 和 Peripheral 使能 SMP 功能，以 Secure Connections Numeric Comparison 方式配对的演示。根据蓝牙协议，当使用 Secure Connections 且双方皆为 DisplayYesNo 或 KeyboardDisplay 功能时，会使用 Numeric Comparison 方式进行匹配。使用该功能需在 feature\_smp/app.c 中定义：

```
#define SMP_TEST_MODE           SMP_TEST_LESC_NC
```

编译，分别擦除 Flash 后烧录到两个开发板中。按其中一个开发板（作为 Central）的按键 SW4（启动连接），连接成功后，可以看到 Central 亮绿灯，Peripheral 亮红灯，此时从双方的 Log 中看到“TK display”的值，二者应该相等。分别在 Central 和 Peripheral 开发板上按按键 SW3，代表发送 YES，可以看到二者的白色、蓝色灯相继亮起，代表二者 Pair 成功，抓包如下（框起部分为分别等待 Central 和 Peripheral 进行按键确认）：



P...	Time	Item	Transmitter	Receiver
1	10:18:49 AM.004 797 500	Connectable ("smp" 55:66:77:00:91:01, 21s)	Master: "smp" 55:66:77:00:91:01	Slave: "Scanning Device"
4	10:18:49 AM.009 833 125	Connectable ("smp" 55:66:77:00:91:02, Initiator "smp" 55:66:77:00:91:01, 2.36 s)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
428	10:18:51 AM.392 156 875	SMP Pairing Request (Keyboard Display, Bonding, MITM, SC, Int=EndKey   IdKey, Rsp=EndKey   IdKey)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
429	10:18:51 AM.392 475 500	SMP Security Request (Bonding, MITM, SC)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
439	10:18:51 AM.454 428 625	SMP Pairing Response (Display Yes No, Bonding, MITM, SC, Int=IdKey, Rsp=IdKey)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
445	10:18:51 AM.485 906 875	SMP Pairing Public Key (Debug Key, X=20B003D2:F297BE2C:5E2C83A7:E9F9A5B9:EFF49111:ACF4FDD...)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
455	10:18:51 AM.516 928 625	SMP Pairing Public Key (X=3CAAB646:C553879F:43BE2158:D8EA0548:F9A787C8:19AD56A9:B0F9D507...)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
461	10:18:51 AM.518 855 375	SMP Pairing Confirm (Nb=9BA10FD4:95D7F814:6361D2E9:20A89444)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
467	10:18:51 AM.548 406 625	SMP Pairing Random (Nb=FC1BC417:C4DAA576:D5A88EE:150DE51E)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
480	10:18:51 AM.610 678 750	SMP Pairing Random (Nb=C3A14FEE:D3244A53:9ABEA4FA:IE27193D)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
484	10:18:51 AM.641 699 750	Empty LE Packets (x 563, 1 retry, 8.78 s)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'937	10:19:00 AM.423 412 625	SMP Pairing DH Key Check (E3F33D54:4F349CCE:95CC3006:F411E2C6)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'938	10:19:00 AM.423 810 875	Empty LE Packets (x 142, 2.22 s)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'306	10:19:02 AM.641 935 625	SMP Pairing DH Key Check (B4BAFC08:D02FE6D2:2A1FAA1A:AAE013D6)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'312	10:19:02 AM.673 413 875	LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, SKDm=0x0DF1AA076C1F78C53, I...)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'318	10:19:02 AM.704 435 625	LLCP Encryption Response (SKDs=0xDDED3C7C3E635EB, IVs=0x90B4C9FD)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'334	10:19:02 AM.798 185 750	LLCP Start Encryption Request	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'341	10:19:02 AM.829 664 250	LLCP Start Encryption Response	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'347	10:19:02 AM.860 686 000	LLCP Start Encryption Response	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'357	10:19:02 AM.923 186 000	SMP Identity Information (IRK=A3B086D9:459694CF:7A1C228E:6FD8D393)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'367	10:19:02 AM.985 685 875	SMP Identity Address Information (BDADDR=55:66:77:00:91:02)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'374	10:19:03 AM.017 164 125	SMP Identity Information (IRK=A728C802:7FC864DC:FF9555DC:87B7B5F8)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'382	10:19:03 AM.048 414 125	SMP Identity Address Information (BDADDR=55:66:77:00:91:01)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'383	10:19:03 AM.048 771 375	Empty LE Packets (x 444, 1 retry, 6.94 s)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01

Figure 11.29: SC Numeric Comparison 的抓包

### 11.6.9 Secure Connections OOB

Central 和 Peripheral 使能 SMP 功能，以 Secure Connections OOB 方式配对的演示。根据蓝牙协议，在 SC 配对方式下，当双方均支持 OOB，拥有对方 OOB 数据时，会使用 SC OOB 方式进行匹配。使用该功能需在 feature\_smp/app.c 中定义：

```
#define SMP_TEST_MODE           SMP_TEST_LESC_OOB
```

编译 feature\_test，分别擦除 Flash 后烧录 B92\_ble\_multi\_conn\_sdk/build/B92/feature\_test/feature\_test.bin 到两个开发板中。给两个开发板的 UART 口 PA3(Tx)，PA4(Rx) 接到电脑串口上，配置波特率 115200 8N1。

下载后，开发板初始化会生成 random value 和 confirm value，按其中一个开发板（作为 Central）的按键 SW4（启动连接），连接成功后，可以看到 Central 亮绿色灯，Peripheral 亮红色灯。

```
[2024-05-16 17:26:54.420]# RECV ASCII>
SC OOB data-random value: 4b 00 56 82 94 17 29 d4 df 5d 88 05 6e bc ff e4
SC OOB data-confirm value: 20 8c 06 36 b0 21 f0 b3 60 55 ac 75 e8 9a cc f1
SC OOB Initialize[APP][INI] FEATURE_SMP init:
```

Figure 11.30: random value 和 confirm value

两端的 OOB 功能可以通过 uart 发命令方式将其打开或关闭，打开后即可进行 sc oob 方式的配对。



```
[2024-05-16 17:27:46.724]# SEND HEX>  
31
```

```
[2024-05-16 17:27:46.779]# RECV ASCII>  
[APP][SMP] SC OOB remote used flag:1
```

**Figure 11.31:** oob\_enable

这里将 Central 的 oob flag 置 1, Peripheral 置 0, 此时将 peripheral 的 confirm 值和 value 值一起通过 uart 串口发送给 central, Central 和 Peripheral 均会回复“pairing success”。

```
[2024-05-16 17:29:48.712]# RECV ASCII>  
[APP][SMP] pairing success: 01  
[APP][SMP] Security process done event: 44 00 00
```

**Figure 11.32:** pairing success

## 11.6.10 异常处理

### 11.6.10.1 按键没有响应

(1) 硬件原因：可能是由于按键的跳线帽没有接正确，请参考下图：

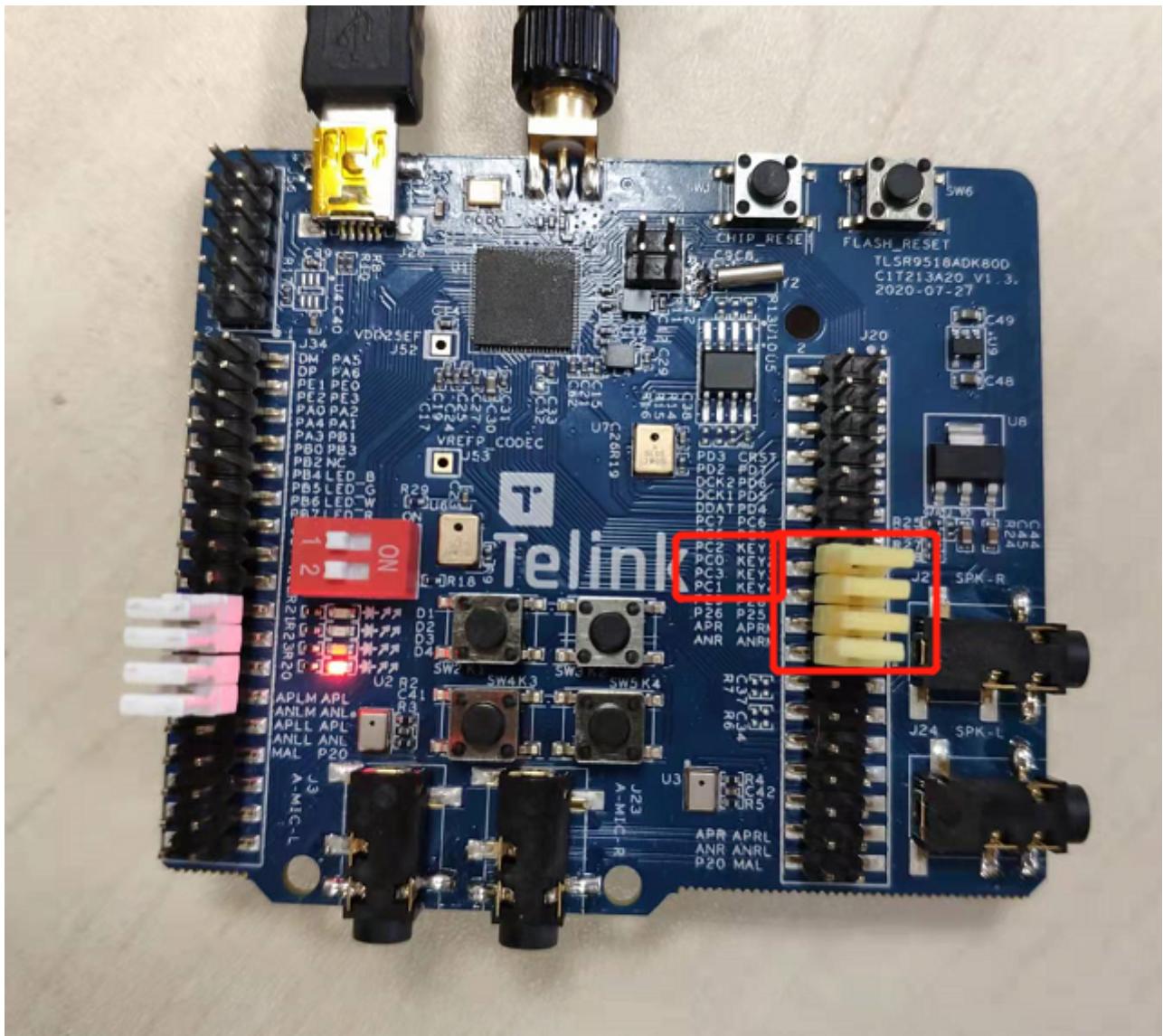


Figure 11.33: Key 跳线帽接错

(2) 环境原因：可能由于开发板烧录后没有 Reset 使程序运行起来。

### 11.6.10.2 LED 灯没有点亮

(1) 硬件原因：可能由于 LED 的跳线帽没有接正确，请参考下图：

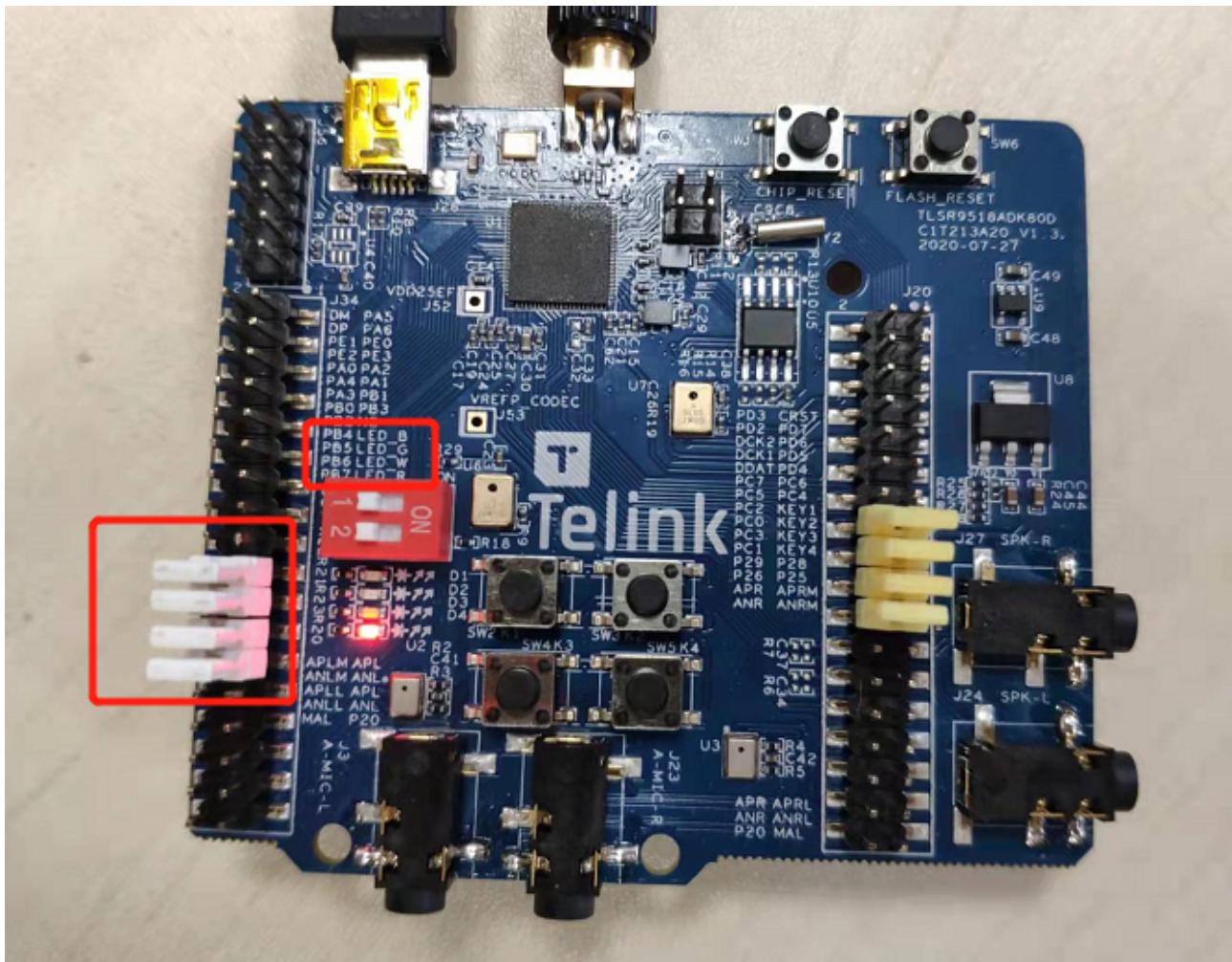


Figure 11.34: LED 跳线帽接错

## 11.7 feature\_whitelist

- 功能：BLE 白名单功能演示。
- 主要硬件：B91 开发板 x 2

以 B91 为例，应用层代码在 tl\_ble\_sdk/vendor/feature\_test/feature\_whitelist 下，需要修改 tl\_ble\_sdk/vendor/feature\_test/feature\_config.h 中的定义：

```
#define FEATURE_TEST_MODE           TEST_WHITELIST
```

来激活这部分代码，做白名单功能的演示。编译并下载到 Central 和 Peripheral 中。烧录完成后，通过按 Central 的 SW4 启动连接，可以看到 Central 和 Peripheral 成功连接，抓包如下：



P...	Time	Item	Payload	Transmitter
1	12:11:12 AM,020 653 375	Connectable ("whitelist" 55:66:77:00:91:02, 8.92 s)		Master: "whitelist" 55:66:77:00:...
256	12:11:14 AM,605 363 125	Connectable ("whitelist" 99:99:99:99:88:33, Initiator "whitelist" 55:66:77:00:91:02, 352 us)		Master: "whitelist" 99:99:99:99:...
260	12:11:14 AM,627 019 875	SMP Security Request (Bonding)	2 bytes (0B 01)	Slave: "whitelist" 99:99:99:99:88...
269	12:11:14 AM,657 581 375	Empty LE Packets (x 395, 9 retries, 6.28 s)		Master: "whitelist" 55:66:77:00:...

Figure 11.35: Central 连接白名单 Peripheral 抓包

通过 BDT 修改 Central 或 Peripheral 的 MAC 地址为其他地址。

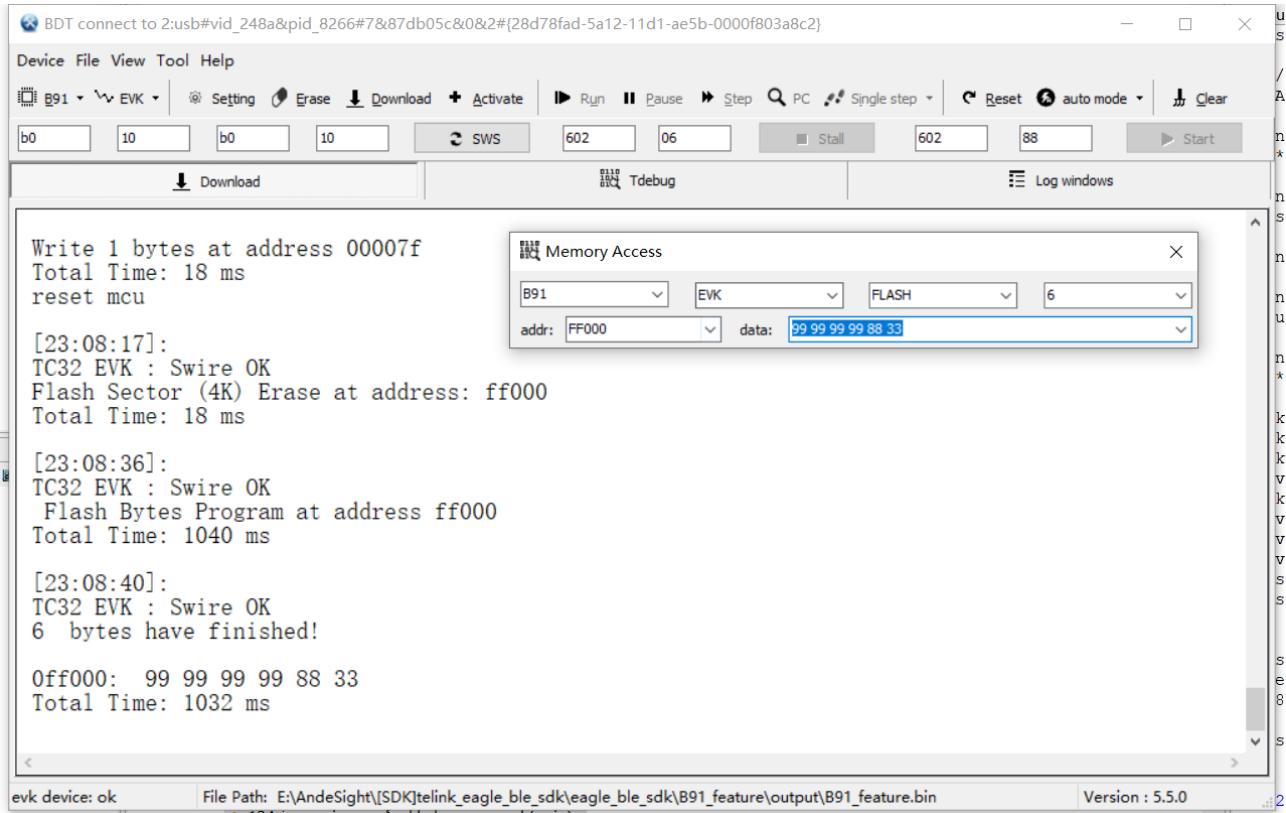


Figure 11.36: 设置白名单设备 MAC 地址

复位，并重新点击 Central 的 SW4 按键，会看到 Central 发出了 Connect Request，Peripheral 却不会响应，以至于 Central 一直尝试连接：



P...	Time	Item	Payload	Transmitter	Receiver
1	12:43:56 AM.001 288 625	Connectable ("whitelist" 55:66:77:00:91:01, Initiator "whit...		Master: "whitelist" 55:66:77:00:91:01	Slave: "whitelist" 55:66:77:00:91:02
1	12:43:56 AM.001 288 625	Connectable Undirected Adv (55:66:77:00:91:01, Name=". 30 bytes (01 91 00 77 66 ...	30 bytes (01 91 00 77 66 ...	Master: "whitelist" 55:66:77:00:91:01	Slave: "Scanning Device"
5'942	12:44:43 AM.648 815 625	Connection Indication Packet (55:66:77:00:91:02 > 55:66:77:00:91:01)	34 bytes (02 91 00 77 66 ...	Slave: "whitelist" 55:66:77:00:91:02	Master: "whitelist" 55:66:77:00:91:01
5'944	12:44:43 AM.679 441 875	Connectable ("whitelist" 55:66:77:00:91:01, Initiator "whit...		Master: "whitelist" 55:66:77:00:91:01	Slave: "whitelist" 55:66:77:00:91:02
5'954	12:44:43 AM.763 094 750	Empty LE (2 retries)	No data	Master: "whitelist" 55:66:77:00:91:02	Slave: "whitelist" 55:66:77:00:91:01
5'943	12:44:43 AM.669 345 125	Empty LE Unit	No data	Master: "whitelist" 55:66:77:00:91:02	Slave: "whitelist" 55:66:77:00:91:01
5'943	12:44:43 AM.669 345 125	Empty LE Packet	No data	Master: "whitelist" 55:66:77:00:91:02	Slave: "whitelist" 55:66:77:00:91:01
5'950	12:44:43 AM.731 844 750	Empty LE Unit	No data	Master: "whitelist" 55:66:77:00:91:02	Slave: "whitelist" 55:66:77:00:91:01
5'954	12:44:43 AM.763 094 750	Empty LE Unit	No data	Master: "whitelist" 55:66:77:00:91:02	Slave: "whitelist" 55:66:77:00:91:01
5'969	12:44:43 AM.893 817 625	Connectable ("whitelist" 55:66:77:00:91:01, Initiator "whit...		Master: "whitelist" 55:66:77:00:91:01	Slave: "whitelist" 55:66:77:00:91:02
5'980	12:44:43 AM.973 718 875	Empty LE (3 retries)	No data	Master: "whitelist" 55:66:77:00:91:02	Slave: "whitelist" 55:66:77:00:91:01
5'992	12:44:44 AM.069 442 625	Connectable ("whitelist" 55:66:77:00:91:01, Initiator "whit...		Master: "whitelist" 55:66:77:00:91:01	Slave: "whitelist" 55:66:77:00:91:02
6'003	12:44:44 AM.118 092 250	Empty LE (3 retries)	No data	Master: "whitelist" 55:66:77:00:91:02	Slave: "whitelist" 55:66:77:00:91:01
6'018	12:44:44 AM.279 442 750	Connectable ("whitelist" 55:66:77:00:91:01, Initiator "whit...		Master: "whitelist" 55:66:77:00:91:01	Slave: "whitelist" 55:66:77:00:91:02
6'029	12:44:44 AM.362 466 875	Empty LE (3 retries)	No data	Master: "whitelist" 55:66:77:00:91:02	Slave: "whitelist" 55:66:77:00:91:01
6'041	12:44:44 AM.453 192 250	Connectable ("whitelist" 55:66:77:00:91:01, Initiator "whit...		Master: "whitelist" 55:66:77:00:91:01	Slave: "whitelist" 55:66:77:00:91:02
6'052	12:44:44 AM.536 216 125	Empty LE (3 retries)	No data	Master: "whitelist" 55:66:77:00:91:02	Slave: "whitelist" 55:66:77:00:91:01
6'064	12:44:44 AM.625 693 125	Connectable ("whitelist" 55:66:77:00:91:01, Initiator "whit...		Master: "whitelist" 55:66:77:00:91:01	Slave: "whitelist" 55:66:77:00:91:02

Figure 11.37: Peripheral 连接非白名单 Central 抓包

因为，Central 和 Peripheral 已经建立过连接，且已将对方的 MAC 地址加入到自己的白名单中。

## 11.8 feature\_soft\_timer

- 功能：Soft Timer（软件定时器）功能演示。并作为 IO Debug 方法的演示参考。
- 主要硬件：B91 开发板，逻辑分析仪或示波器

以 B91 为例，应用层代码在 tl\_ble\_sdk/vendor/feature\_test/feature\_soft\_timer 下，需要修改 tl\_ble\_sdk/vendor/feature\_test/feature\_config.h 中的定义：

```
#define FEATURE_TEST_MODE TEST_SOFT_TIMER
```

来激活这部分代码，并在 app\_config.h 中使能 IO Debug：

```
#define DEBUG_GPIO_ENABLE 1
```

来做 Soft Timer 和 IO Debug 的功能演示。

将开发板上 PE1 接出，作为 PM IO，用于观察休眠唤醒。将 PA3、PBO、PB2、PE0 接出，分别作为 Debug IO 4、5、6、7 的信号显示，参考宏定义 DEBUG\_GPIO\_ENABLE。

编译固件后，烧录到开发板中，上电，逻辑分析仪抓取 Debug IOs 信号，可以看到逻辑分析仪上抓到的 5 个 IO 效果如下：



Figure 11.38: SoftTimer Debug IO capture



从图上可以看出广播间隔为 50ms（左右，底层会有一点动态调整），PA3 每隔 23ms toggle 一次，PBO 以 7ms 和 17ms 交替间隔进行 toggle，PB2 每隔 13ms toggle 一次，PE0 每隔 100ms toggle 一次，参考宏定义 BLT\_SOFTWARE\_TIMER\_ENABLE 所使能的代码。可以看到 Soft Timer 事件触发时，设备都有被提前唤醒，也就是说 Soft Timer 设置的事件是不受休眠影响的。

## 11.9 feature\_ota

- 功能：BLE OTA 功能演示。
- 主要硬件：B91 开发板 x 2

以 B91 为例，应用层代码在 tl\_ble\_sdk/vendor/feature\_test/feature\_ota 下，需要修改 tl\_ble\_sdk/vendor/feature\_test/feature\_config.h 中的定义：

```
#define FEATURE_TEST_MODE           TEST_OTA
```

来激活这部分代码，做 OTA 的演示。

在 app\_config.h 中，使能 BLE\_OTA\_CLIENT\_ENABLE，并且可以选用两种传输协议：extended protocol 和 legacy protocol。extended protocol 是传输速率更快的协议。

```
#define BLE_OTA_CLIENT_ENABLE      1
#if (BLE_OTA_CLIENT_ENABLE)
    //0: OTA extended protocol; 1: OTA legacy protocol
    #define OTA_LEGACY_PROTOCOL     1
#endif
```

编译，得到 Central 的固件，烧录到其中一块开发板上作为 Central。将 Peripheral 的新固件（上电白灯常亮）烧录到 Central 的 OTA 文件地址（demo 中有两个地址，分别是 0x80000 和 0xC0000）。Peripheral 对应的开发板烧录老固件，上电绿灯常亮。

按下 Central 的 SW5 按键，建立连接。连接后按下 SW2 或者 SW4（SW2 对应 0x80000，SW 对应 0xC0000），开始 OTA 流程。Peripheral 接收新固件，传输完成后，Peripheral 自动重启运行新固件，可观察 LED 从绿灯常亮变化为白灯常亮。

抓包可以看到 Central 不停向 Peripheral 发送 WriteCmd：



P...	Time	Item	Payload	Transmitter	Receiver
96'226	6:27:15 PM.235 660 875	Empty LE	No data	Slave: "ota" 55:66:77:00:91:01	Master: "ota" 55:66:77:00:91:02
96'227	6:27:15 PM.236 361 750	Connectable ("ota" 55:66:77:00:91:01, Initiator ...)		Master: "ota" 55:66:77:00:91:01	Slave: "ota" 55:66:77:00:91:02
105'893	6:28:38 PM.148 256 625	SMP Security Request (Bonding)	2 bytes (0B 01)	Slave: "ota" 55:66:77:00:91:01	Master: "ota" 55:66:77:00:91:02
105'894	6:28:38 PM.157 568 000	Empty LE Packets (x 1626, 10 retries, 8.17 s)		Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'083	6:28:46 PM.327 988 875	LLCP Connection Update Indication (WinSz=6,25...)	12 bytes (00 05 04 00 08 00 00 00 90...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'085	6:28:46 PM.328 546 375	ATT Read By Type Request Packet (1 - Max Han...		Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'088	6:28:46 PM.337 759 000	ATT Read By Type Response Packet (00)	3 bytes (38 00 00)	Slave: "ota" 55:66:77:00:91:01	Master: "ota" 55:66:77:00:91:02
108'091	6:28:46 PM.347 989 000	ATT Write Command Packet (56: 03 FF 10 00)	4 bytes (03 FF 10 00)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'104	6:28:46 PM.397 988 750	ATT Write Command Packet (56: 00 00 25 A0 00 ...)	20 bytes (00 00 25 A0 00 00 00 5...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'106	6:28:46 PM.398 663 250	ATT Write Command Packet (56: 01 00 00 00 00 ...)	20 bytes (01 00 00 00 00 00 00 00 ...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'108	6:28:46 PM.399 338 500	ATT Write Command Packet (56: 02 00 4B 4E 4C ...)	20 bytes (02 00 4B 4E 4C 54 00 00 38...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'110	6:28:46 PM.400 013 250	ATT Write Command Packet (56: 03 00 97 02 0A ...)	20 bytes (03 00 97 02 0A E0 93 82 02...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'112	6:28:46 PM.400 688 375	ATT Write Command Packet (56: 04 00 62 FC 73 ...)	20 bytes (04 00 62 FC 73 90 02 80 99...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'114	6:28:46 PM.401 363 125	ATT Write Command Packet (56: 05 00 97 A2 00 E0 93 ...)	20 bytes (05 00 97 A2 00 E0 93 82 02...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'116	6:28:46 PM.402 038 125	ATT Write Command Packet (56: 06 00 B7 02 00 ...)	20 bytes (06 00 B7 02 00 E4 0D 43 23...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'118	6:28:46 PM.402 712 875	ATT Write Command Packet (56: 07 00 12 00 93 ...)	20 bytes (07 00 12 00 93 E2 22 00 73...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'120	6:28:46 PM.403 387 875	ATT Write Command Packet (56: 08 00 00 13 23 10 ...)	20 bytes (08 00 00 13 23 10 97...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'122	6:28:46 PM.404 062 625	ATT Write Command Packet (56: 09 00 00 E0 13 ...)	20 bytes (09 00 00 E0 13 0E AE 08 63...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'124	6:28:46 PM.404 737 625	ATT Write Command Packet (56: 0A 00 53 00 11 ...)	20 bytes (0A 00 53 00 11 03 91 03 C...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01

Figure 11.39: feature\_ota OTA 开始

升级完成，Peripheral 发送给 Central 一个 Notify 指令，代表升级成功，之后发送 Terminate 指令，断开连接。由于二者进行 SMP 连接，Central 存有 Peripheral 的信息，立刻又重新连上：

P...	Time	Item	Payload	Transmitter	Receiver
138'835	6:29:28 PM.709 306 875	ATT Write Command Packet (56: 02 15 08 95 07 07 ...)	20 bytes (02 15 08 95 07 94 06 93 05...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'837	6:29:28 PM.709 981 875	ATT Write Command Packet (56: 03 15 44 00 00 ...)	20 bytes (03 15 44 00 00 00 34 00 00...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'839	6:29:28 PM.710 656 625	ATT Write Command Packet (56: 04 15 00 42 DB ...)	20 bytes (04 15 00 42 DB 42 DB 0E 3...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'841	6:29:28 PM.711 331 750	ATT Write Command Packet (56: 05 15 D7 D8 D9 ...)	20 bytes (05 15 D7 D8 D9 DA DB 0E 2...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'843	6:29:28 PM.712 008 125	ATT Write Command Packet (56: 06 15 D3 D4 D5 ...)	20 bytes (06 15 D3 D4 D5 D6 D8 D9 D...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'845	6:29:28 PM.712 683 000	ATT Write Command Packet (56: 07 15 42 C1 42 ...)	20 bytes (07 15 42 C1 42 0E 00 00 00...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'848	6:29:28 PM.717 957 125	ATT Write Command Packet (56: 08 15 02 01 00 ...)	20 bytes (08 15 02 01 00 00 01 01 00...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'850	6:29:28 PM.718 632 125	ATT Write Command Packet (56: 09 15 18 18 FF ...)	20 bytes (09 15 18 18 FF FF FF FF...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'852	6:29:28 PM.719 307 125	ATT Write Command Packet (56: 0A 15 91 8D A9 ...)	20 bytes (0A 15 91 8D A9 EF FF F...)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'854	6:29:28 PM.719 982 250	ATT Write Command Packet (56: 02 FF 0A 15 F5 ...)	6 bytes (02 FF 0A 15 F5 EA)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'870	6:29:28 PM.758 186 125	ATT Notification Packet (56: 06 FF 00)	3 bytes (06 FF 00)	Slave: "ota" 55:66:77:00:91:01	Master: "ota" 55:66:77:00:91:02
138'872	6:29:28 PM.758 725 250	LLCP Termination (Remote User Terminated Conn...)	2 bytes (02 13)	Slave: "ota" 55:66:77:00:91:01	Master: "ota" 55:66:77:00:91:02
138'873	6:29:28 PM.767 957 000	Empty LE Packets (x 2, 80 us)		Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'875	6:29:28 PM.768 896 375	Connectable ("ota" 55:66:77:00:91:01, Initiator ...)		Master: "ota" 55:66:77:00:91:01	Slave: "ota" 55:66:77:00:91:02
138'905	6:29:29 PM.103 018 500	SMP Security Request (Bonding)	2 bytes (0B 01)	Slave: "ota" 55:66:77:00:91:01	Master: "ota" 55:66:77:00:91:02
138'910	6:29:29 PM.112 330 375	Empty LE Packets (x 719, 5 retries, 3.6 s)		Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
139'916	6:29:32 PM.722 312 750	L2CAP SDU (Basic)		Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
139'917	6:29:32 PM.723 313 625	Empty LE Packets (x 586, 3 retries, 2.94 s)		Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01

Figure 11.40: feature\_ota OTA 完成

## 11.10 feature\_l2cap\_coc

- 功能：BLE COC(Connection Oriented Channel) 连接和数据发送功能演示。
- 主要硬件：B91 开发板 x 2

(1) 将 feature\_config.h 中的 FEATURE\_TEST\_MODE 宏改为 TEST\_L2CAP\_COC：

```
#define FEATURE_TEST_MODE TEST_L2CAP_COC
```

(2) 编译下载和操作：



编译并将生成的固件分别烧录到两块开发板中，上电，按其中一个开发板的 SW5(作为 Master) 按键，触发连接，连接成功 Central 端红灯常亮，Peripheral 端绿灯常亮，然后按任意一个开发板的 SW2 触发 COC 连接，连接成功后两块开发板蓝灯常亮，然后按任意一块开发板的 SW2 发送数据给对方，对方收到数据会 toggle 一次白灯，同时也可以通过串口查看 log，默认 GPIO\_PD4 波特率 1000000。

### (3) 抓包

以下是 COC 连接和发送数据的抓包截图，发送 300 字节的数据。

#### COC 连接请求：

Item	Status	Transmitter	Time	PHY	Name	Value	Hex
Connectable + Scannable Undirected ("CoC" 11:11:11:00:00:04, Initiator "CoC" 11:11:11:00:00:03)	OK	Master: "CoC" 11:11:11:00:00:04, Slave: "CoC" 11:11:11:00:00:03	0.010 015 875	LE 1M	Link-Layer Information	-47.0 dBm (Average) on c...	
ATT Exchange MTU Request Packet	OK	Slave: "CoC" 11:11:11:00:00:04	24.040 002 375	LE 1M	Retransmission Information		
ATT Exchange MTU Response Packet	OK	Master: "CoC" 11:11:11:00:00:03	24.040 287 500	LE 1M	Link-Layer Packet	L2CAP Start Fragment / C...	
ATT Exchange MTU Response Packet	OK	Master: "CoC" 11:11:11:00:00:04	24.071 022 750	LE 1M			
LLCP Length Request (MaxRx=200 bytes, 1.71 ms, MaxTx=200 bytes, 1.71 ms)	OK	Slave: "CoC" 11:11:11:00:00:04	24.071 307 875	LE 1M			
LLCP Length Request (MaxRx=200 bytes, 1.71 ms, MaxTx=200 bytes, 1.71 ms)	OK	Slave: "CoC" 11:11:11:00:00:04	24.071 894 625	LE 1M			
LLCP Length Response (MaxRx=200 bytes, 1.71 ms, MaxTx=200 bytes, 1.71 ms)	OK	Slave: "CoC" 11:11:11:00:00:04	24.102 502 250	LE 1M			
LLCP Length Response (MaxRx=200 bytes, 1.71 ms, MaxTx=200 bytes, 1.71 ms)	OK	Master: "CoC" 11:11:11:00:00:03	24.102 803 750	LE 1M			
Empty LE Packets (x 1067, 16.7 s)	OK	Slave: "CoC" 11:11:11:00:00:04	24.103 105 250	LE 1M			
L2CAP LE Connection Request (Dst=0x0000 (Dynamic), Src=0x0040, MTU=1'000 ...)	OK	Master: "CoC" 11:11:11:00:00:03	40.758 958 500	LE 1M			
L2CAP LE Connection Response (Dst=0x0040, MTU=1'000 bytes, MPS=248 bytes, C...	OK	Slave: "CoC" 11:11:11:00:00:04	40.789 978 875	LE 1M			
Empty LE Packets (x 2987, 1 retry, 46.7 s)	OK	Master: "CoC" 11:11:11:00:00:03	40.820 999 375	LE 1M			
L2CAP SDU (LE, Service=0x0080 (Dynamic), PDU count= 2)	OK	Slave: "CoC" 11:11:11:00:00:04	40.877 643 500	LE 1M			
Empty LE Packets (x 334, 1 retry, 5.21 s)	OK	Slave: "CoC" 11:11:11:00:00:04	40.874 481 270 875	LE 1M			
L2CAP SDU (LE, Service=0x0080 (Dynamic), PDU count= 2)	OK	Slave: "CoC" 11:11:11:00:00:04	42.696 386 250	LE 1M			
Empty LE Packets (x 198, 1 retry, 3.09 s)	OK	Slave: "CoC" 11:11:11:00:00:04	42.700 013 500	LE 1M			
L2CAP SDU (LE, Service=0x0080 (Dynamic), PDU count= 2)	OK	Slave: "CoC" 11:11:11:00:00:04	45.790 131 875	LE 1M			
L2CAP LE Flow Control Credit (Src=0x0040, 5)	OK	Slave: "CoC" 11:11:11:00:00:04	45.821 152 375	LE 1M			
Empty LE Packets (x 198, 3.06 s)	OK	Master: "CoC" 11:11:11:00:00:03	45.852 172 750	LE 1M			
Out Of Sync Packets (Duration= 0 ms, x 1)	OK	Master: "CoC" 11:11:11:00:00:03	49.816 956 750	LE 1M			
L2CAP SDU (LE, Service=0x0080 (Dynamic), PDU count= 2)	OK	Slave: "CoC" 11:11:11:00:00:04	49.845 918 375	LE 1M			
Empty LE Packets (x 46228, 41 retries, 16.7 min)	OK	Slave: "CoC" 11:11:11:00:00:04	49.849 545 875	LE 1M			

Figure 11.41: COC 连接请求抓包

#### COC 连接响应：

Item	Status	Transmitter	Time	PHY	Name	Value	Hex
Connectable + Scannable Undirected ("CoC" 11:11:11:00:00:04, Initiator "CoC" 11:11:11:00:00:03)	OK	Master: "CoC" 11:11:11:00:00:04, Slave: "CoC" 11:11:11:00:00:03	0.010 015 875	LE 1M	Link-Layer Information	-44.5 dBm (Average) on c...	
ATT Exchange MTU Request Packet	OK	Slave: "CoC" 11:11:11:00:00:04	24.040 002 375	LE 1M	Retransmission Information		
ATT Exchange MTU Response Packet	OK	Master: "CoC" 11:11:11:00:00:03	24.040 287 500	LE 1M	Link-Layer Packet	L2CAP Start Fragment / C...	
ATT Exchange MTU Response Packet	OK	Master: "CoC" 11:11:11:00:00:04	24.071 022 750	LE 1M			
LLCP Length Request (MaxRx=200 bytes, 1.71 ms, MaxTx=200 bytes, 1.71 ms)	OK	Slave: "CoC" 11:11:11:00:00:04	24.071 307 875	LE 1M			
LLCP Length Request (MaxRx=200 bytes, 1.71 ms, MaxTx=200 bytes, 1.71 ms)	OK	Slave: "CoC" 11:11:11:00:00:04	24.071 894 625	LE 1M			
LLCP Length Response (MaxRx=200 bytes, 1.71 ms, MaxTx=200 bytes, 1.71 ms)	OK	Slave: "CoC" 11:11:11:00:00:04	24.102 502 250	LE 1M			
LLCP Length Response (MaxRx=200 bytes, 1.71 ms, MaxTx=200 bytes, 1.71 ms)	OK	Master: "CoC" 11:11:11:00:00:03	24.102 803 750	LE 1M			
Empty LE Packets (x 1067, 16.7 s)	OK	Slave: "CoC" 11:11:11:00:00:04	24.103 105 250	LE 1M			
L2CAP LE Connection Request (Dst=0x0000 (Dynamic), Src=0x0040, MTU=1'000 ...)	OK	Master: "CoC" 11:11:11:00:00:03	40.758 958 500	LE 1M			
L2CAP LE Connection Response (Dst=0x0040, MTU=1'000 bytes, MPS=248 bytes, C...	OK	Slave: "CoC" 11:11:11:00:00:04	40.789 978 875	LE 1M			
Empty LE Packets (x 2987, 1 retry, 46.7 s)	OK	Master: "CoC" 11:11:11:00:00:03	40.820 999 375	LE 1M			
L2CAP SDU (LE, Service=0x0080 (Dynamic), PDU count= 2)	OK	Slave: "CoC" 11:11:11:00:00:04	40.877 643 500	LE 1M			
Empty LE Packets (x 334, 1 retry, 5.21 s)	OK	Slave: "CoC" 11:11:11:00:00:04	40.874 481 270 875	LE 1M			
L2CAP SDU (LE, Service=0x0080 (Dynamic), PDU count= 2)	OK	Slave: "CoC" 11:11:11:00:00:04	42.696 386 250	LE 1M			
Empty LE Packets (x 198, 1 retry, 3.09 s)	OK	Slave: "CoC" 11:11:11:00:00:04	42.700 013 500	LE 1M			
L2CAP SDU (LE, Service=0x0080 (Dynamic), PDU count= 2)	OK	Slave: "CoC" 11:11:11:00:00:04	45.790 131 875	LE 1M			
L2CAP LE Flow Control Credit (Src=0x0040, 5)	OK	Slave: "CoC" 11:11:11:00:00:04	45.821 152 375	LE 1M			
Empty LE Packets (x 198, 3.06 s)	OK	Master: "CoC" 11:11:11:00:00:03	45.852 172 750	LE 1M			
Out Of Sync Packets (Duration= 0 ms, x 1)	OK	Master: "CoC" 11:11:11:00:00:03	49.816 956 750	LE 1M			
L2CAP SDU (LE, Service=0x0080 (Dynamic), PDU count= 2)	OK	Slave: "CoC" 11:11:11:00:00:04	49.845 918 375	LE 1M			
Empty LE Packets (x 46228, 41 retries, 16.7 min)	OK	Slave: "CoC" 11:11:11:00:00:04	49.849 545 875	LE 1M			

Figure 11.42: COC 连接请求响应抓包

COC 数据发送 1(SDU=300,MPS=248,L2CAP 层分 2 帧，第一帧在 payload 头部包含 2 字节 SDU length)：

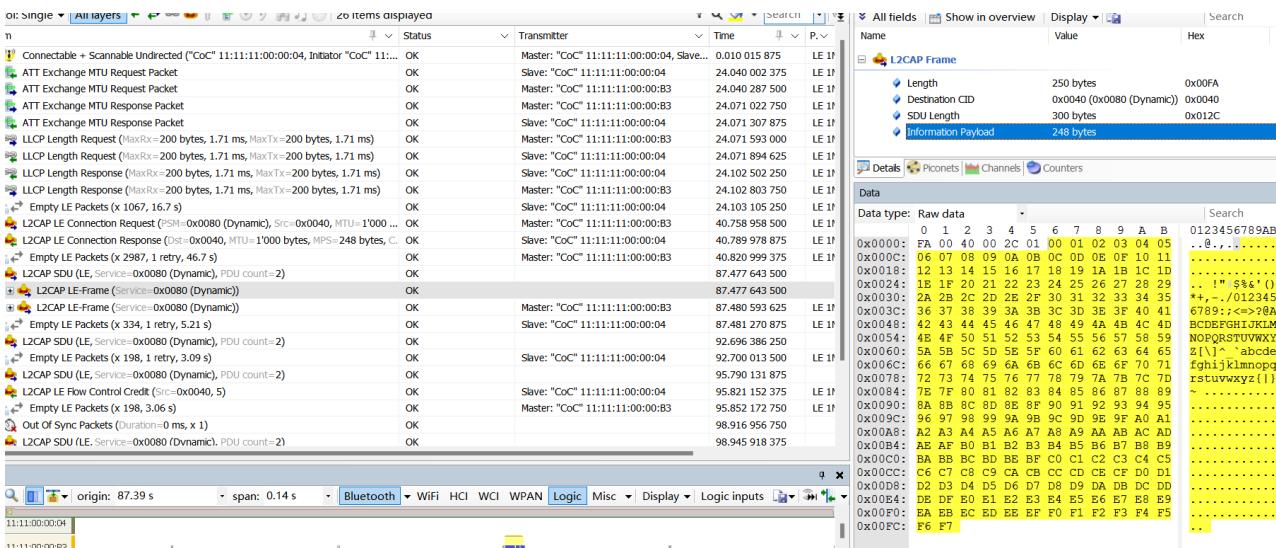


Figure 11.43: COC 数据发送

COC 数据发送 2(SDU=300,MPS=248,L2CAP 层分 2 帧):

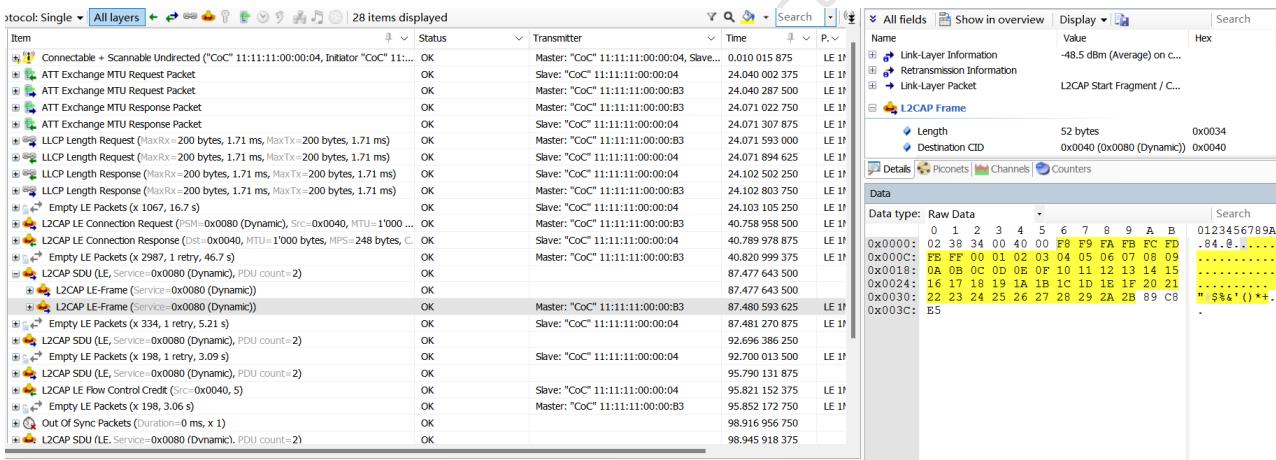


Figure 11.44: COC 数据发送

## 11.11 feature\_ext\_adv

- 功能：扩展广播
- 主要硬件：B92 开发板 ×1

应用层代码在 B92\_ble\_multi\_conn\_sdk/vendor/feature\_test/feature\_ext\_adv 下，需要修改 B92\_ble\_multi\_conn\_sdk/vendor/feature\_test/feature\_config.h 中的定义：

```
#define FEATURE_TEST_MODE TEST_EXT_ADV
```

demo 打开了 TLKAPI\_DEBUG\_CHANNEL\_GSUART，默认使用 GPIO\_PD4 进行模拟 DEBUG 输出，可通过连接串口工具 RX 查看打印信息。



```
#define TLKAPI_DEBUG_CHANNEL           TLKAPI_DEBUG_CHANNEL_GSUART
```

在 ext\_adv\_set.c 中，app\_ext\_adv\_set\_test() 函数实现了多种 extend advertising。通过按键 SW5 循环切换发送的扩展广播内容。

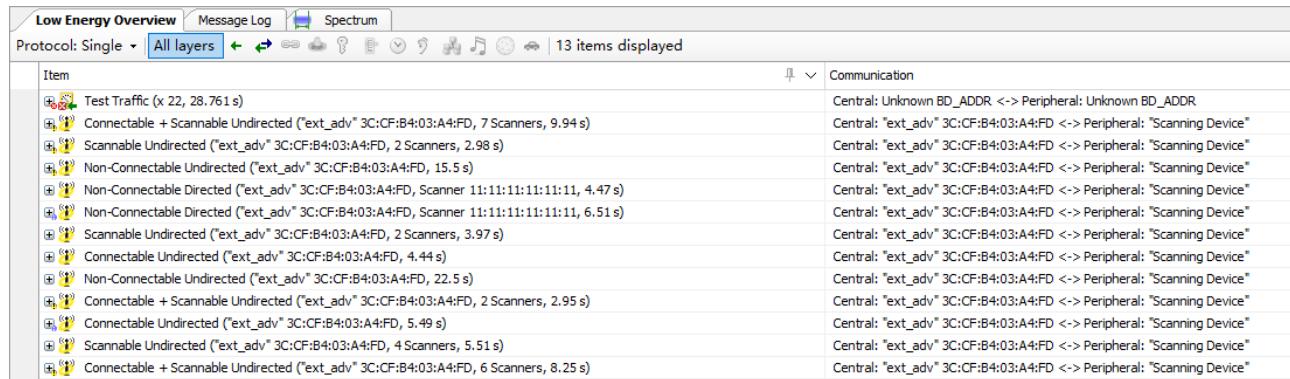


Figure 11.45: EXT ADV

## 11.12 feature\_ext\_scan

- 功能：扩展扫描
- 主要硬件：B92 开发板 ×2

应用层代码在 B92\_ble\_multi\_conn\_sdk/vendor/feature\_test/feature\_ext\_scan 下，需要修改 B92\_ble\_multi\_conn\_sdk/vendor/feature\_test/feature\_config.h 中的定义：

```
#define FEATURE_TEST_MODE           TEST_EXT_SCAN
```

demo 打开了 TLKAPI\_DEBUG\_CHANNEL\_GSUART，默认使用 GPIO\_PD4 进行模拟 DEBUG 输出，可通过连接串口工具 RX 查看打印信息。

```
#define TLKAPI_DEBUG_CHANNEL           TLKAPI_DEBUG_CHANNEL_GSUART
```

在 app\_config.h 中可以配置扫描的目标 phy 类型：

```
#define APP_EXT_SCAN_1M                  0x01  
#define APP_EXT_SCAN_CODED                0x02  
#define APP_EXT_SCAN_1M_CODED              0x04  
#define EXT_SCAN_TEST_MODE                APP_EXT_SCAN_CODED
```

两块开发板一块烧录 ext\_adv 固件作为 peripheral，另一块烧录 ext\_scan 固件作为 central。按下 central 的 SW5 按键，可以连接到正在发出扩展广播的 peripheral。



1 ADV_EXT_IND Packet (AdvDataInfo   AuxPtr[7] (data), @78.575 051 750, +11.7 ms, ->#84232])	Central: "ext_adv" 3C:CF:B4:03:A6:56 <-> Peripheral: "Scanning Device"
1 AUX_ADV_IND Packet (3C:CF:B4:03:A6:56, AdvA   AdvDataInfo   AdvData, #84218->, Local Name, Name="ext_adv", Fla...	Central: "ext_adv" 3C:CF:B4:03:A6:56 <-> Peripheral: "Scanning Device"
1 ADV_EXT_IND Packet (AdvDataInfo   AuxPtr[27] (data), @78.651 411 125, +11.16 ms, ->#84302])	Central: "ext_adv" 3C:CF:B4:03:A6:56 <-> Peripheral: "Scanning Device"
1 ADV_EXT_IND Packet (AdvDataInfo   AuxPtr[27] (data), @78.651 411 125, +10.86 ms, ->#84302])	Central: "ext_adv" 3C:CF:B4:03:A6:56 <-> Peripheral: "Scanning Device"
1 ADV_EXT_IND Packet (AdvDataInfo   AuxPtr[27] (data), @78.651 411 375, +10.56 ms, ->#84302])	Central: "ext_adv" 3C:CF:B4:03:A6:56 <-> Peripheral: "Scanning Device"
1 AUX_ADV_IND Packet (3C:CF:B4:03:A6:56, AdvA   AdvDataInfo   AdvData, #84290->, Local Name, Name="ext_adv", Fla...	Central: "ext_adv" 3C:CF:B4:03:A6:56 <-> Peripheral: "Scanning Device"
1 ADV_EXT_IND Packet (AdvDataInfo   AuxPtr[3] (data), @78.727 970 875, +12.72 ms, ->#84374])	Central: "ext_adv" 3C:CF:B4:03:A6:56 <-> Peripheral: "Scanning Device"
1 ADV_EXT_IND Packet (AdvDataInfo   AuxPtr[3] (data), @78.727 970 875, +12.42 ms, ->#84374])	Central: "ext_adv" 3C:CF:B4:03:A6:56 <-> Peripheral: "Scanning Device"
1 ADV_EXT_IND Packet (AdvDataInfo   AuxPtr[3] (data), @78.727 971 250, +12.12 ms, ->#84374])	Central: "ext_adv" 3C:CF:B4:03:A6:56 <-> Peripheral: "Scanning Device"
1 AUX_ADV_IND Packet (3C:CF:B4:03:A6:56, AdvA   AdvDataInfo   AdvData, #84356->, Local Name, Name="ext_adv", Fla...	Central: "ext_adv" 3C:CF:B4:03:A6:56 <-> Peripheral: "Scanning Device"
1 AUX_CONNECT_RSP Packet (ChSel #2, 3C:CF:B4:03:A4:FD > 3C:CF:B4:03:A6:56, AA:0x8E6C99B1)	Central: "ext_adv" 3C:CF:B4:03:A6:56 <-> Peripheral: "ext_adv" 3C:CF:B4:03:A4:FD
1 AUX_CONNECT_RSP Packet (3C:CF:B4:03:A6:56, AdvA   TargetA)	Central: "ext_adv" 3C:CF:B4:03:A6:56 <-> Peripheral: "ext_adv" 3C:CF:B4:03:A4:FD
1 Test Traffic	Central: Unknown BD_ADDR <-> Peripheral: Unknown BD_ADDR
1 SMP Security Request (Bonding)	Central: "ext_adv" 3C:CF:B4:03:A4:FD <-> Peripheral: "ext_adv" 3C:CF:B4:03:A6:56
1 SMP Pairing Feature Exchange (No Input No Output, Bonding > No Input No Output, Bonding)	Central: "ext_adv" 3C:CF:B4:03:A4:FD <-> Peripheral: "ext_adv" 3C:CF:B4:03:A6:56
1 SMP Short Term Key Generation	Central: "ext_adv" 3C:CF:B4:03:A4:FD <-> Peripheral: "ext_adv" 3C:CF:B4:03:A6:56
1 LLCP Encryption Start (EDIV=0x0000, SKDm=0x8CA88ADFEDE9EEA, IVm=0x90BFDCC2 > SKDs=0x2A49F95F9CA593A2, IVs=0...)	Central: "ext_adv" 3C:CF:B4:03:A4:FD <-> Peripheral: "ext_adv" 3C:CF:B4:03:A6:56
1 SMP Transport Specific Key Distribution (CTK=978D1F01a7A73026:D5D73823:7C20AC00 > EDIV=0x0FB2 & LTK=0x8667A1A4A3...)	Central: "ext_adv" 3C:CF:B4:03:A4:FD <-> Peripheral: "ext_adv" 3C:CF:B4:03:A6:56
1 ATT Read By Type Transaction (1 - Max Handle, Characteristic Declaration: [3=Device Name, Read], [5=Appearance, Read], [7=P...	Central: "ext_adv" 3C:CF:B4:03:A4:FD <-> Peripheral: "ext_adv" 3C:CF:B4:03:A6:56
1 ATT Read By Type Transaction (7 - Max Handle, Characteristic Declaration: [10=Service Changed, Indicate], [14=Prn ID, Read], [...	Central: "ext_adv" 3C:CF:B4:03:A4:FD <-> Peripheral: "ext_adv" 3C:CF:B4:03:A6:56
1 ATT Read By Type Transaction (18 - Max Handle, Characteristic Declaration: [20=Boot Keyboard Input Report, Read, Notify], [2...	Central: "ext_adv" 3C:CF:B4:03:A4:FD <-> Peripheral: "ext_adv" 3C:CF:B4:03:A6:56
1 ATT Read By Type Transaction (25 - Max Handle, Characteristic Declaration: [29=Report, Read, Notify], [31=Report, Read, Wri...	Central: "ext_adv" 3C:CF:B4:03:A4:FD <-> Peripheral: "ext_adv" 3C:CF:B4:03:A6:56
1 ATT Read By Type Transaction (36 - Max Handle, Characteristic Declaration: [39=HID Information, Read], [41=HID Control Point, ...)	Central: "ext_adv" 3C:CF:B4:03:A4:FD <-> Peripheral: "ext_adv" 3C:CF:B4:03:A6:56
1 ATT Read By Type Transaction (44 - Max Handle, Characteristic Declaration: [48=00010203-0405-0607-0809-0A0800CD2810, Read], [49=...	Central: "ext_adv" 3C:CF:B4:03:A4:FD <-> Peripheral: "ext_adv" 3C:CF:B4:03:A6:56
1 ATT Read By Type Transaction (48 - Max Handle, Characteristic Declaration: [57=0010203-0405-0607-0809-0A0800CD2811, R...	Central: "ext_adv" 3C:CF:B4:03:A4:FD <-> Peripheral: "ext_adv" 3C:CF:B4:03:A6:56

Figure 11.46: EXT SCAN

## 11.13 feature\_per\_adv

- 功能：扩展扫描
- 主要硬件：B92 开发板 x1

应用层代码在 B92\_ble\_multi\_conn\_sdk/vendor/feature\_test/feature\_per\_adv 下，需要修改 B92\_ble\_multi\_conn\_sdk/vendor/feature\_test/feature\_config.h 中的定义：

```
#define FEATURE_TEST_MODE TEST_PER_ADV
```

demo 打开了 TLKAPI\_DEBUG\_CHANNEL\_GSUART，默认使用 GPIO\_PD4 进行模拟 DEBUG 输出，可通过连接串口工具 RX 查看打印信息。

```
#define TLKAPI_DEBUG_CHANNEL TLKAPI_DEBUG_CHANNEL_GSUART
```

编译固件烧录到开发板，上电运行，可看到周期广播

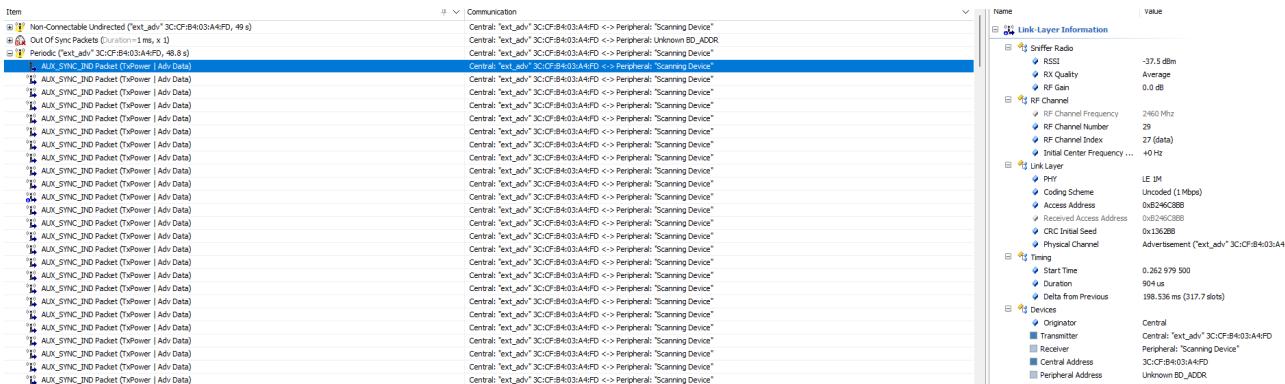


Figure 11.47: PER ADV



## 11.14 feature\_per\_adv\_sync

- 功能：扩展扫描
- 主要硬件：B92 开发板 ×2

应用层代码在 B92\_ble\_multi\_conn\_sdk/ vendor/ feature\_test/ feature\_per\_adv\_sync 下，需要修改 B92\_ble\_multi\_conn\_sdk/vendor/feature\_test/feature\_config.h 中的定义：

```
#define FEATURE_TEST_MODE           TEST_PER_ADV_SYNC
```

demo 打开了 TLKAPI\_DEBUG\_CHANNEL\_GSUART，默认使用 GPIO\_PD4 进行模拟 DEBUG 输出，可通过连接串口工具 RX 查看打印信息。

```
#define TLKAPI_DEBUG_CHANNEL      TLKAPI_DEBUG_CHANNEL_GSUART
```

在 app.c 中，app\_controller\_event\_callback，可以判断 HCI 事件进入不同的事件 callback，在扫描到周期广播后，可以建立同步，可以通过打印观察事件

```
[12:19:46.454]收->◆[APP][INI] feature_per_adv_sync init
[12:21:57.141]收->◆[UI][PAIR] Pair begin:
[12:21:57.389]收->◆[UI][PAIR] Pair end:
[12:22:02.054]收->◆[UI][PAIR] Pair begin:
[12:22:02.245]收->◆[UI][PAIR] Pair end:
[12:22:09.199]收->◆[UI][PAIR] Pair begin:
[12:22:09.358]收->◆[UI][PAIR] Pair end:
[12:22:14.362]收->◆[UI][PAIR] Pair begin:
[12:22:14.518]收->◆blc_ll_createConnection error code :00
[12:22:14.584]收->◆[UI][PAIR] Pair end:
[12:22:14.718]收->◆[APP][EVENT]periodic_adv_sync_established syncHandle:0008 mac:56 A6 03 B4 CF 3C
```

Figure 11.48: PER ADV SYNC



## 12 其他模块

### 12.1 24MHz 晶体外部电容

参考下图中的 24MHz 晶体匹配电容的位置 C1/C4。

SDK 默认使用内部电容（即 ana\_8a<5:0> 对应的 cap）作为 24MHz 晶振的匹配电容，此时 C1/C4 不用焊接电容。使用该方案的优势是：在 Telink 治具上可以测量并调节该电容，使得最终应用产品的频点值达到最佳。

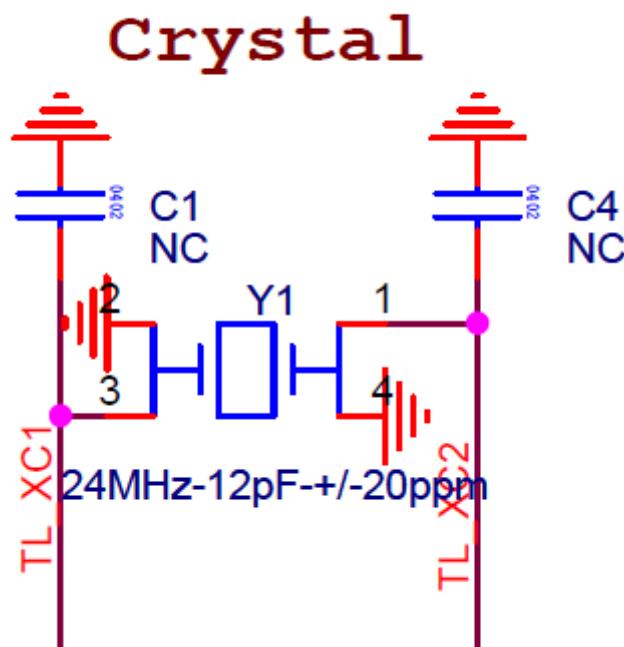


Figure 12.1: 24M 晶体电路

如果需要使用外部焊接电容作为 24M 晶振的匹配电容 (C1/C4 焊接电容)，则只要在 main 函数开始的地方（一定要在 sys\_init 函数之后，blc\_app\_loadCustomizedParameters() 之前）调用下面 API 即可：

```
static inline void blc_app_setExternalCrystalCapEnable(u8 en)
{
    blt_miscParam.ext_cap_en = en;
    analog_write(0x8a, analog_read(0x8a) | 0x80); // disable internal cap
}
```

### 12.2 32KHz 时钟源选择

SDK 默认使用 MCU 内部 32kHz RC 振荡电路，简称 32kHz RC。32kHz RC 的误差比较大，所以对于 suspend 或者 deep retention 时间较长的应用，其时间准确性会差一些。目前 32kHz RC 默认支持的最大长连接不能超



过 3s，一旦超过这个时间，ble\_timing 会出错，造成收包时间点不准确，容易出现收发包 retry，功耗增大，甚至出现断连。

如果用户需要实现更低的连接功耗，包括低功耗睡眠情况下时钟计时更加准确，可以选择使用外部 32k 晶体，简称 32k Pad，目前 SDK 支持该模式。

用户只需要在 main 函数开始的地方（一定要在 sys\_init 函数之前）调用下面两个 API 中的一个：

```
void blc_pm_select_internal_32k_crystal(void);
void blc_pm_select_external_32k_crystal(void);
```

上述 API 分别是选择 32k RC 和 32k Pad 的 API，SDK 默认调用 blc\_pm\_select\_internal\_32k\_crystal 选择的 32k RC，如果需要使用 32k Pad，将其替换成 blc\_pm\_select\_external\_32k\_crystal 即可。

## 12.3 PA

如果需要使用 RF PA，B91 请参考 drivers/B91/ext\_driver/software\_pa.h，其余芯片依次类推。

首先打开下面的宏，默认是关闭的。

```
#ifndef PA_ENABLE
#define PA_ENABLE          0
#endif
```

在系统初始化的时候，调用 PA 的初始化。

```
void rf_pa_init(void);
```

参考代码实现，该初始化里面，将 PA\_TXEN\_PIN 和 PA\_RXEN\_PIN 设为 GPIO 输出模式，初始状态为输出 0。需要 user 定义 TX 和 RX 的 PA 对应的 GPIO：

```
#ifndef PA_TXEN_PIN
#define PA_TXEN_PIN          GPIO_PB2
#endif

#ifndef PA_RXEN_PIN
#define PA_RXEN_PIN          GPIO_PB3
#endif
```

另外将 void (\*rf\_pa\_callback\_t)(int type) 注册为 PA 的回调处理函数，实际它处理了下面 3 种 PA 状态：PA 关、开 TX PA、开 RX PA。

```
#define PA_TYPE_OFF          0
#define PA_TYPE_TX_ON         1
#define PA_TYPE_RX_ON         2
```

User 只需要调用上面的 rf\_pa\_init，app\_rf\_pa\_handler 被注册到底层的回调，BLE 在各种状态时，都会自动调用 app\_rf\_pa\_handler 的处理。



## 12.4 PhyTest

PhyTest 即 PHY test，是指对 BLE controller RF 性能的测试。

详情请参照《Core\_v5.4》(Vol 4/Part E/7.8.28~7.8.30) 和《Core\_v5.4》(Vol 6/Part F "Direct Test Mode" )。

### 12.4.1 PhyTest API

PhyTest 的源码被封装在 library 文件中，提供相关 API 供 user 使用，请参考 stack/ble/controller/phy/phy\_test.h 文件。

```
void      blc_phy_initPhyTest_module(void);

ble_sts_t blc_phy_setPhyTestEnable (u8 en);
bool      blc_phy_isPhyTestEnable(void);

int       blc_phystest_cmd_handler (u8 *p, int n);
```

首先调用 blc\_phy\_initPhyTest\_module 初始化 PhyTest 模块，应用层触发 PhyTest 后，底层自动调用 blc\_phy\_setPhyTestEnable(1) 开启 PhyTest 模式。

PhyTest 是一个特殊的模式，和正常的 BLE 功能是互斥的，一旦进入 PhyTest 模式，广播和连接都不可用。所以运行正常 BLE 功能时不能触发 PhyTest。

PhyTest 结束后，要么直接重新上电，要么调用 blc\_phy\_setPhyTestEnable (0)，此时 MCU 会自动 reboot。使用 blc\_phy\_isPhyTestEnable 可判断当前 PhyTest 是否被触发。

### 12.4.2 PhyTest demo

tl\_ble\_sdk demo “feature\_test”的 app\_config.h 中，测试模式修改为“TEST\_BLE\_PHY”，如下所示：

```
#define FEATURE_TEST_MODE           TEST_BLE_PHY
```

根据物理接口和测试命令格式的不同，PhyTest 可分为两种测试模式，如下所示。

```
#define      PHYTEST_MODE_THROUGH_2_WIRE_UART      1 //Direct Test Mode through a 2-wire
          ↵  UART interface
#define      PHYTEST_MODE_OVER_HCI_WITH_UART        2 //Direct Test Mode over HCI(UART
          ↵  hardware interface)
```

选择 PhyTest 的测试模式，如下定义为 uart 两线模式：

```
#define BLE_PHYTEST_MODE      PHYTEST_MODE_THROUGH_2_WIRE_UART
```

如下定义为 HCI 模式 UART 接口（硬件接口还是 uart） phystest：



```
#define BLE_PHYTEST_MODE          PHYTEST_MODE_OVER_HCI_WITH_UART
```

用户需设置发送和接收回调函数：

```
blc_register_hci_handler (app_phyTest_rxUartCb, app_phyTest_txUartCb);
```

app\_phyTest\_rxUartCb 实现上位机下发的 cmd 的解析和执行，app\_phyTest\_txUartCb 实现将相应的结果和数据反馈给上位机。

编译 feature\_test 生成的 bin 文件直接测试可以通过。user 可研究一下 code 的实现，掌握相关接口的使用。

**注意：**

在验证 PHY test 功能时，需保证没有其他模块使用 RF 功能。



## 13 调试方法

该章节介绍几种在开发过程中常用的调试方法。

### 13.1 GPIO 模拟 UART 打印

为方便用户调试, tl\_ble\_sdk 提供 GPIO 模拟 UART 串口输出调试信息的一种实现, 该方法仅作为一种参考, 并非官方推荐的调试信息输出方法。将例程中 app\_config.h 中的宏 UART\_PRINT\_DEBUG\_ENABLE 定义为 1 后, 就可以在代码中直接使用与 C 语言语法规则一致的 printf 接口进行串口输出。各个应用例程中都有模拟 UART 的 GPIO 的默认配置, 用户可根据需求进行更改:

```
//////////////////////////////////////////////////////////////// PRINT DEBUG INFO //////////////////////////////
#ifndef UART_PRINT_DEBUG_ENABLE
    //the baud rate should not bigger than 115200 when MCU clock is 16M
    //the baud rate should not bigger than 1000000 when MCU clock is 24M
#define PRINT_BAUD_RATE          1000000
#define DEBUG_INFO_TX_PIN        GPIO_PD7
#define PULL_WAKEUP_SRC_PD7      PM_PIN_PULLUP_10K
#define PD7_OUTPUT_ENABLE         1
#define PD7_DATA_OUT             1 //must
#endif
```

Figure 13.1: 模拟 UART 的 GPIO 的定义

一般而言, 只修改波特率与其他每行定义中的 IO 名(图中所有 PD7)。

注意:

- 波特率目前最高支持 1M。
- 由于 GPIO 模拟 UART 串口的打印会被中断打断, 致使模拟的 UART 的时序不准确, 因此在实际使用过程中时而会有打印乱码的情况发生。
- 由于 GPIO 模拟 UART 串口的打印会占用 CPU, 所以不建议在中断中加入打印, 会影响对时序要求较高的中断任务。

### 13.2 BDT 工具读取全局变量的值

Telink 官方提供的 BDT 工具, 不仅可以用来烧录固件, 也可以进行一些线上 Debug, 其中 Tdebug 标签页下可以读到代码中的全局变量的值。用户可以根据需求在代码中添加全局变量, 在 Tdebug 中读取。具体请参考《BDT 用户指南》的 Debug 章节。

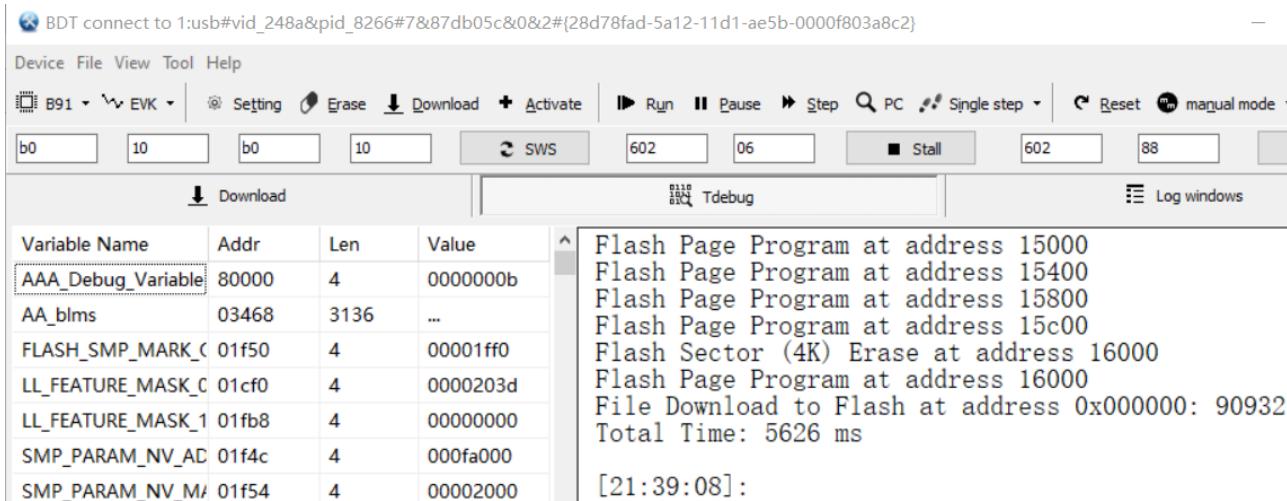


Figure 13.2: BDT 读取全局变量

#### 注意：

- 芯片处于休眠状态时，读出来的值为全 0。
- 该功能依赖于生成的 list 文件（默认 B91 生成 objdump.txt，其余芯片是 xxx.lst），所以用户在向官方提供调试固件时，也应尽可能提供 list 文件，以便 Telink 工程师通过 list 文件读取底层一些变量的值。

### 13.3 BDT 工具的 Memory Access 功能

Telink 官方提供的 BDT 工具，还可以通过 Memory Access 功能，读取 Flash、内存等空间指定位置的内容，也可以写入数据。需要在读取前保证 SWS 处于接通状态。具体请参考《BDT 用户指南》的 Debug 章节。

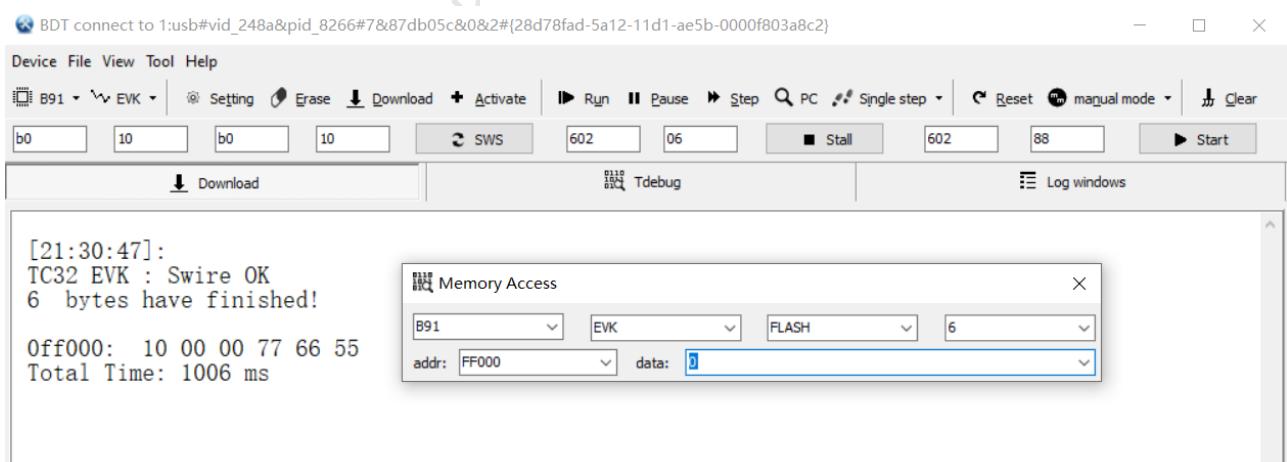


Figure 13.3: BDT Memory Access 功能

### 13.4 BDT 工具读 PC 指针

Telink 官方提供的 BDT 工具，可以用来读取 PC (Program Counter) 指针 (B91 不支持)，这在分析死机问题时，非常有帮助。具体请参考《BDT 用户指南》的 Debug 章节。

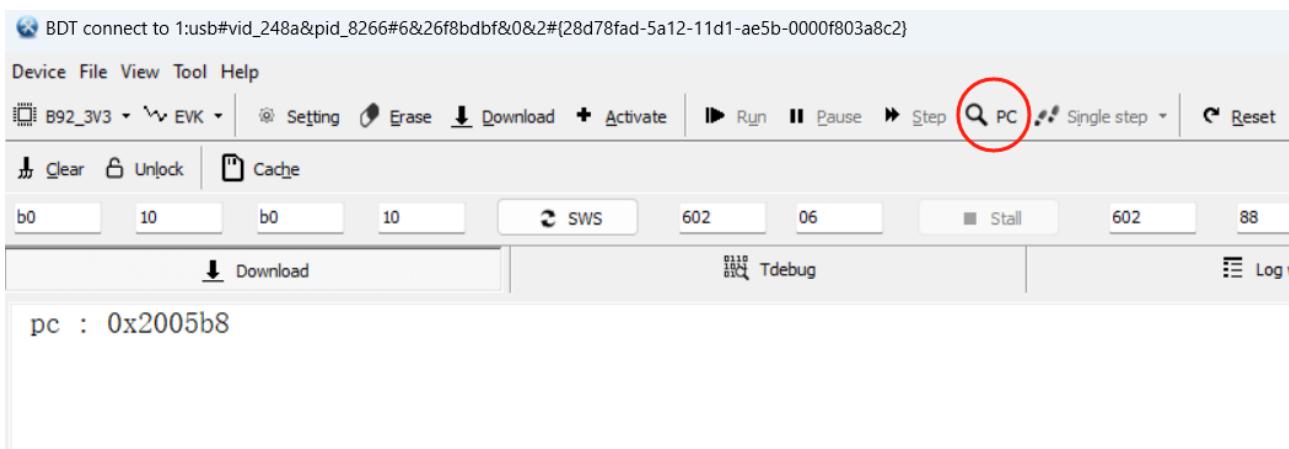


Figure 13.4: BDT PC 指针读取

## 13.5 Debug IO

可以看到在各个例程的 app\_config.h 中都有用宏 DEBUG\_GPIO\_ENABLE 括起来的一段 IO 相关的定义：

```
1 #if (DEBUG_GPIO_ENABLE)
2     #define GPIO_CHN0           GPIO_PE1
3     #define GPIO_CHN1           GPIO_PE2
4     #define GPIO_CHN2           GPIO_PA0
5     #define GPIO_CHN3           GPIO_PA4
6     #define GPIO_CHN4           GPIO_PA3
7     #define GPIO_CHN5           GPIO_PB0
8     #define GPIO_CHN6           GPIO_PB2
9     #define GPIO_CHN7           GPIO_PE0
10
11    #define GPIO_CHN8           GPIO_PA2
12    #define GPIO_CHN9           GPIO_PA1
13    #define GPIO_CHN10          GPIO_PB1
14    #define GPIO_CHN11          GPIO_PB3
15    #define GPIO_CHN12          GPIO_PC7
16    #define GPIO_CHN13          GPIO_PC6
17    #define GPIO_CHN14          GPIO_PC5
18    #define GPIO_CHN15          GPIO_PC4
19
20
21    #define PE1_OUTPUT_ENABLE   1
22    #define PE2_OUTPUT_ENABLE   1
```

Figure 13.5: Debug IO 定义

该功能默认不开启，是给 Telink 工程师内部使用的统一的 Debug IO 定义，通过逻辑分析仪或者示波器抓 IO 的波形进行调试。但用户可以使用类似的方式在应用层添加自己的 Debug IO，参考 common/default\_config.h。

**注意：**



- 官方 release 的 SDK 中，Stack 中的 Debug IO 调试信息以禁用的状态被包含在了库文件中。所以即使用户在应用层定义 DEBUG\_GPIO\_ENABLE 为 1，也不会使能 Stack 中的 Debug IO 调试信息。
- 如果使能 Debug IO，虽然 Stack 中的 Debug IO 不会起效，但是应用层的 Debug IO 会起效，比如 tl\_ble\_sdk 中 CHN14 所代表的 rf\_irq\_handler，CHN15 所代表的 timer\_irq\_handler。

## 13.6 USB my\_dump\_str\_data

在 tl\_ble\_sdk 中可以看到有多处 my\_dump\_str\_data 这个 API 的调用，该功能是 B91 上借助 USB 接口将调试信息输出的一种实现，不是官方推荐的调试信息输出方法，仅作为一种参考，目的是解决在中断中不能通过 GPIO 模拟 UART 进行输出的问题。该功能默认不开启，需要定义 DUMP\_STR\_EN 为 1 才能开启。

## 13.7 JTAG 使用

为了能够使用 JTAG 模块，需要在使用前确保满足以下几点条件：

- JTAG 的四个 GPIO 需要设置成使能模式。
- 如果芯片处于低功耗模式，那么使用 JTAG 前芯片必须退出低功耗模式。
- 如果 JTAG 模式因为 FLASH 中有程序而不能正常使用，需要在使用前用 Telink BDT 工具擦除 FLASH。



Figure 13.6: JTAG 连线说明

### 13.7.1 Diagnostic Report

- (1) 在 Target Manager 里选择 Diagnostic report。

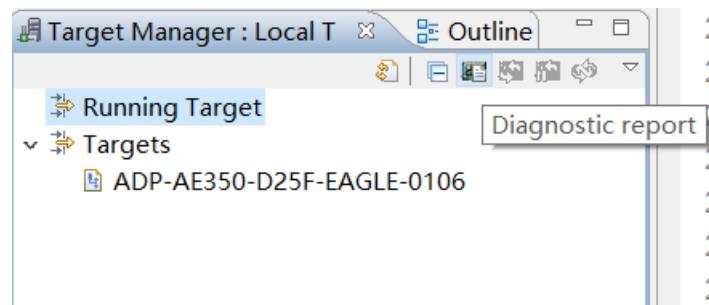


Figure 13.7: Target Manager

(2) 选择 V5 core，不要勾选 SDP (2wires)，我们的 JTAG 暂时不支持 2 线模式，address 输入 0。

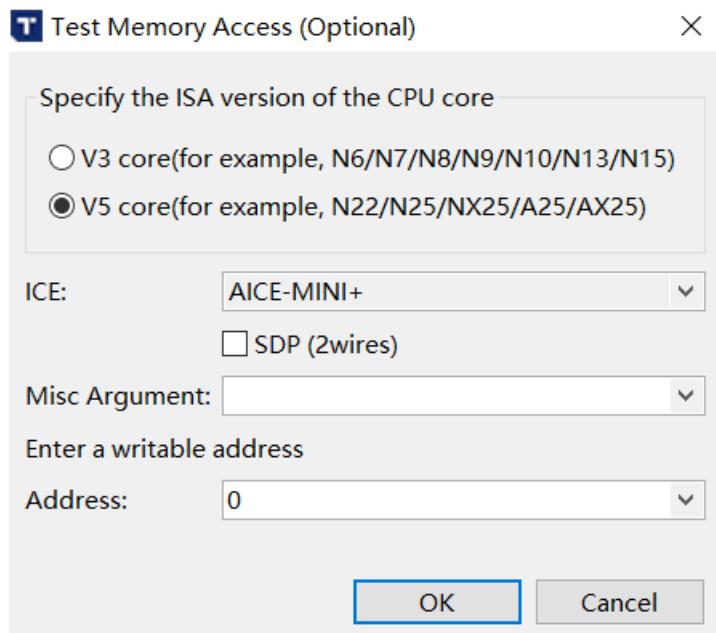


Figure 13.8: Diagnostic report option

(3) 点击 OK，会生成一个 Diagnostic report。



The screenshot shows a terminal-like window titled "ICE Diagnostic Report". It displays a series of diagnostic commands and their results:

```
core0: acsr=0x400000c3
core0: mnvec=0x200000
core0: pc = 0x9270
core0: debug_buffer_size=0x8
REG_SMU=0x0 0000
Testing memory write from addr = 0x0, size:4 words
Testing memory read addr = 0x0, size:4 words
Testing reset_and_halt_one_hart
Hart 0 pc = 0x200000
write 4 words from memory:0x0 to get dmi_busy_delay_count
*****
Diagnostic Report
*****
(PASS) check changing the JTAG frequency ...
(PASS) check JTAG/DTM connectivity ...
(PASS) check that Debug Module (DM) is operational ...
(PASS) check reset-and-debug ...
(PASS) check accessing memory through CPU ...
*****
```

At the bottom right of the window are two buttons: "Copy to Clipboard" and "Close".

Figure 13.9: Diagnostic report

### 13.7.2 Target Configuration

(1) 右击工程文件夹选择“Target Configuration”。

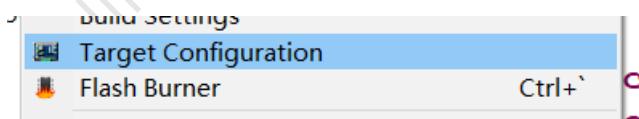


Figure 13.10: Target Configuration Option

(2) 确保没有勾选“SDP (2wires)”。

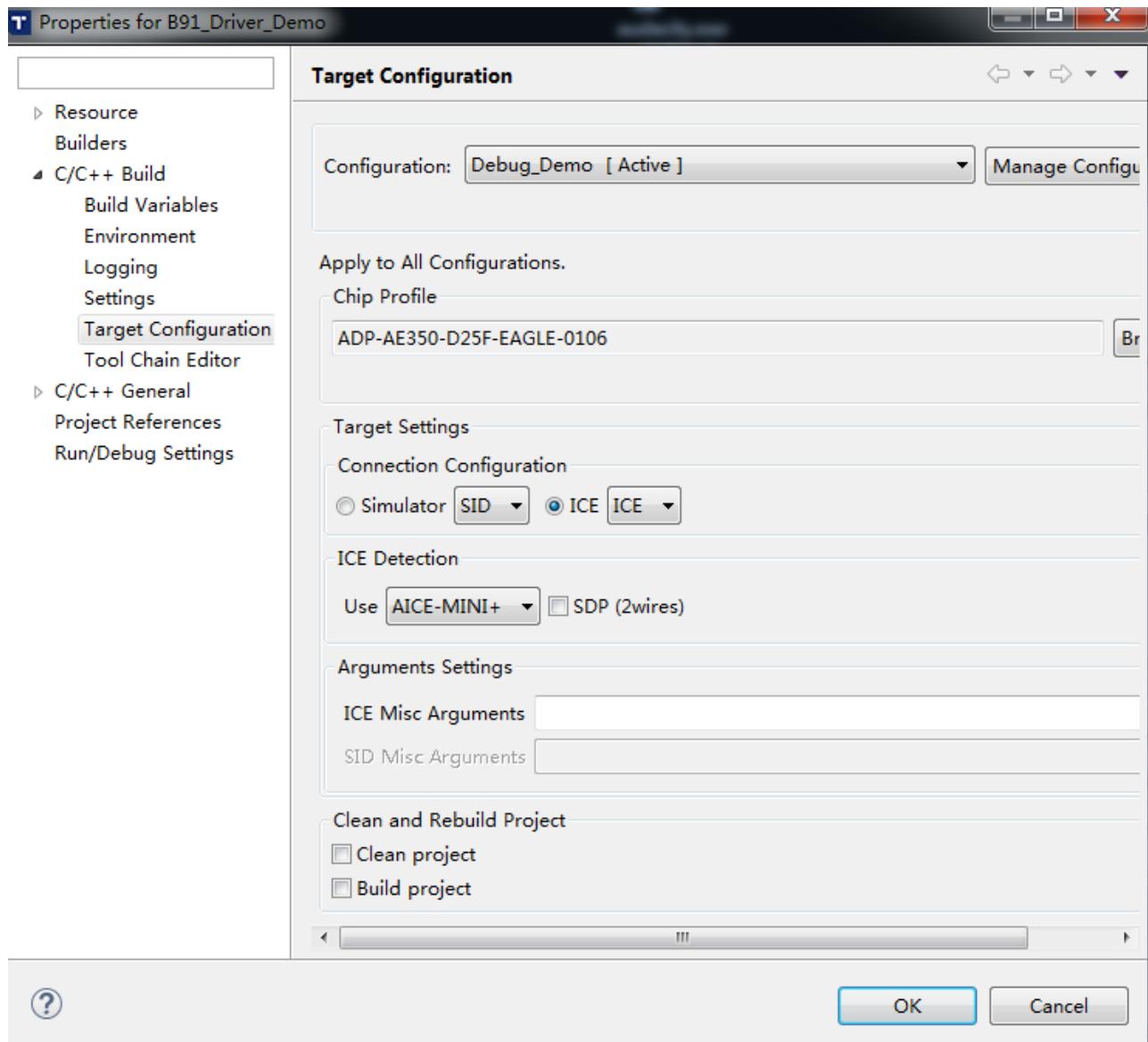


Figure 13.11: Target Configuration

### 13.7.3 Flash Programming

右击工程文件夹选择“Flash Burner”。

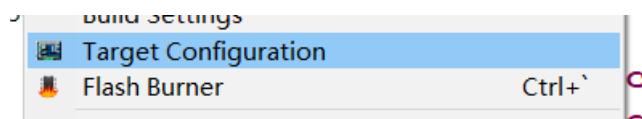


Figure 13.12: Flash Burner Option

(1) 选择 IDE 安装目录下的 SPI\_burn.exe。

(2) 选择需要下载的 bin 文件。

(3) 勾选“Target management”。



- (4) 不要勾选“Target Burn”。
- (5) 勾选“Verification”，如果需要在烧录前擦除 FLASH，可以勾选“erase all”。

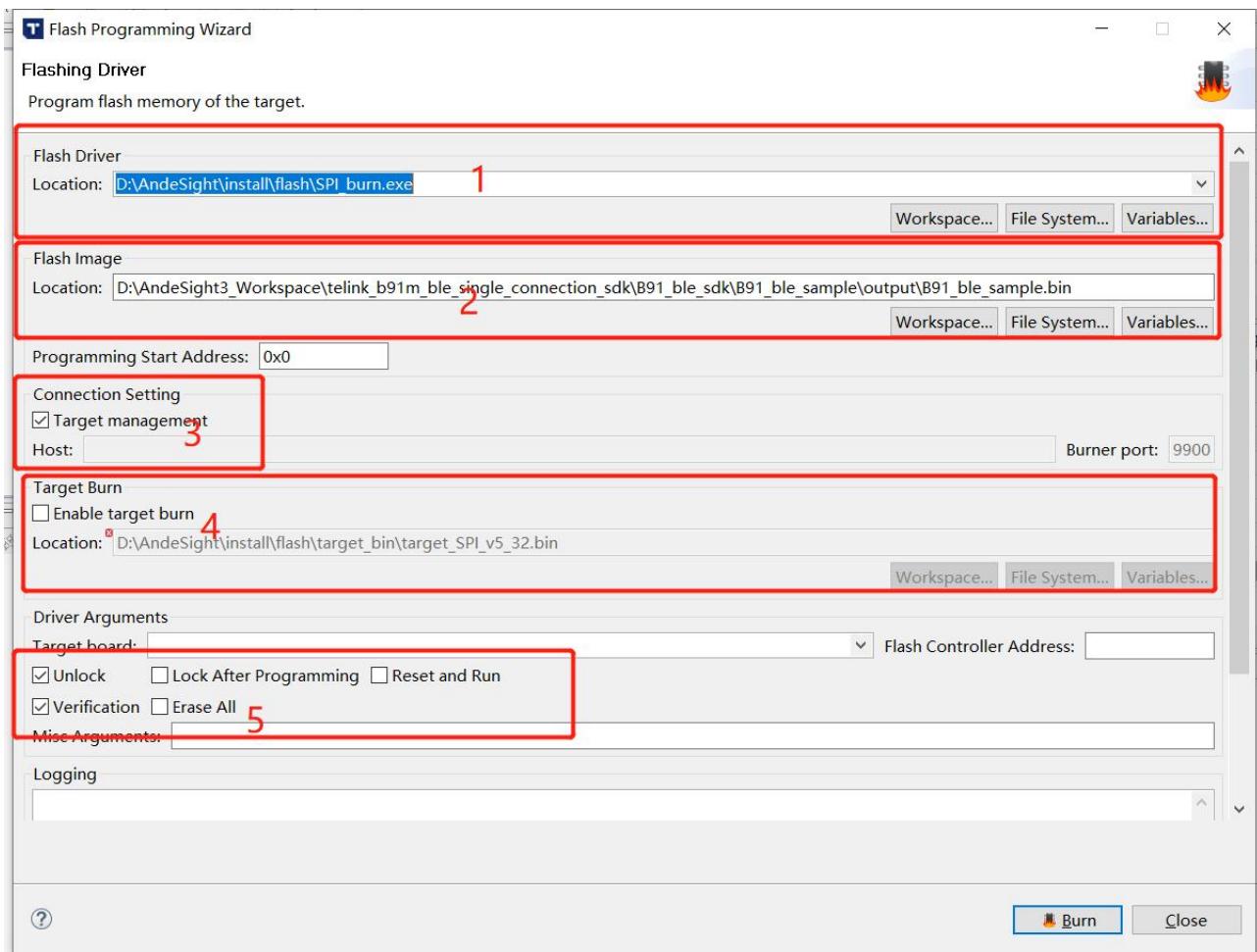


Figure 13.13: Flash Programming

- (6) 点击“Burn”下载 bin 文件，如果出现“Verify sucess”，此时表示烧录成功。

```
Verify success.  
Flash burning done.  
Delete the image copy C:\Users\Admin\.burning  
exitValue: 0  
Spend time: 15s
```

Figure 13.14: Verify sucess



## 14 附录

### 14.1 附录 1：crc16 算法

```
unsigned short crc16 (unsigned char *pD, int len)
{
    static unsigned short poly[2]={0, 0xa001};
    unsigned short crc = 0xffff;
    unsigned char ds;
    int i,j;

    for(j=len; j>0; j--)
    {
        unsigned char ds = *pD++;
        for(i=0; i<8; i++)
        {
            crc = (crc >> 1) ^ poly[(crc ^ ds) & 1];
            ds = ds >> 1;
        }
    }

    return crc;
}
```

### 14.2 附录 2：检查 stack 是否溢出

#### 14.2.1 原理

在 cstartup\_flash.S 中将 stack 的所有内容写为 0x55，在程序运行时会将使用过的栈的数据改写为其他值。通过查看栈被改写的 size 大小，可以判断栈的使用情况，并可以判断栈是否发生溢出。

#### 14.2.2 方法

(1) 打开 boot/cstartup\_flash.S，将 \_FILL\_STK 下的内容使能。

```
_FILL_STK:
#if 1
    lui    t0, 0x55555
    addi   t0, t0, 0x555
    la     t2, _BSS_VMA_END
    la     t3, _STACK_TOP
```



```
_FILL_STK_BEGIN:  
bleu    t3, t2, _MAIN_FUNC  
sw      t0, 0(t2)  
addi    t2, t2, 4  
j       _FILL_STK_BEGIN  
#endif
```

(2) 根据此手册中 2.1.2.1 章节的 SRAM 空间分配确定 stack 的栈底地址。

(3) 使用 BDT 软件将程序的.bin 文件下载完成后，点击 Reset 使程序运行。然后将 slave 与 master 进行连接配对。

(4) 配对完成后，在 BDT 中使用“Tool -> Memory Access”来读取 RAM 中的数据。示例图如下。

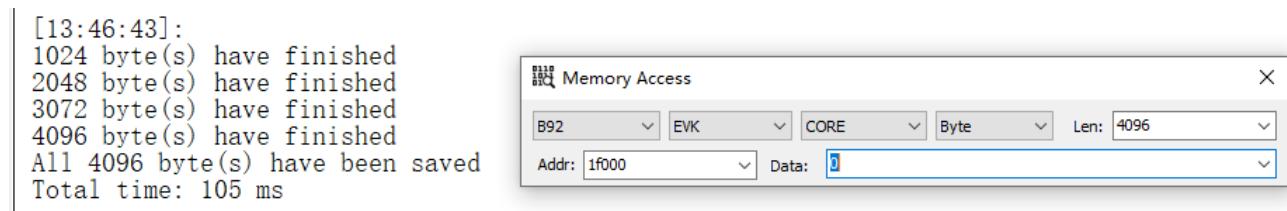


Figure 14.1: 使用 BDT 读取 RAM 数据的设置

(5) 在键盘上按“tab”键可生成一个 Read.bin 文件将数据保存。文件地址为：BDT 安装路径 -> config -> user, Read.bin。

(6) 使用十六进制查看软件打开 Read.bin，如果没有连续的 0x55，说明 stack 溢出到了.bss 段。或者，更准确的方法为，在工程生成的.lst 文件中找到所分配的栈顶地址，如下图所示。然后在 Read.bin 中查看此地址是否被改写为其他内容，若被改写则说明栈溢出。

Idx	Name	Size	VMA	LMA	File off	Align	Flags
0	.vectors	00000162	20000000	20000000	00001000	2**2	CONTENTS, ALLOC, LOAD, READONLY, CODE
1	.retention_reset	00000168	00000000	20000168	00002000	2**2	CONTENTS, ALLOC, LOAD, READONLY, CODE
2	.aes_data	00000020	00000168	200002d0	00002168	2**2	ALLOC
3	.retention_data	00001644	00000188	200002d0	00002188	2**2	CONTENTS, ALLOC, LOAD, DATA
4	.ram_code	00006f02	00001800	20001918	00003800	2**8	CONTENTS, ALLOC, LOAD, READONLY, CODE
5	.text	0000a0a4	20008820	20008820	0000a820	2**2	CONTENTS, ALLOC, LOAD, READONLY, CODE
6	.rodata	00000d60	200128c4	200128c4	000148c4	2**2	CONTENTS, ALLOC, LOAD, READONLY, DATA
7	.eh_frame	000000f8	20013640	20013640	00015640	2**2	CONTENTS, ALLOC, LOAD, READONLY, DATA
8	.data	00000134	00008708	20013738	00016708	2**2	CONTENTS, ALLOC, LOAD, DATA
9	.sbss	00000044	00008840	20013870	00017840	2**2	ALLOC
10	.bss	00000024	00008884	200138b4	00017840	2**2	ALLOC
11	.sdk_version	0000002b	000088a8	2001386c	000168a8	2**2	CONTENTS, ALLOC, LOAD, DATA
12	.debug_info	0005ded8	00000000	00000000	000168d3	2**0	CONTENTS, READONLY, DEBUGGING
13	.debug_abbrev	0000a3e3	00000000	00000000	000747ab	2**0	CONTENTS, READONLY, DEBUGGING
14	.debug_loc	00013c6c	00000000	00000000	0007eb8e	2**0	CONTENTS, READONLY, DEBUGGING

Figure 14.2: 通过.lst 查看栈顶地址