泰凌 BLE 多连接

SDK 开发手册

AN-22063001-C1

Ver1.0.0

2022.06.30

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.x.x.x 版本的开发指南，适用于 B91/B85m 系列。

Published by
Telink Semiconductor

Bldg 3, 1500 Zuchongzhi Rd,
Zhangjiang Hi-Tech Park, Shanghai, China

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.

Information

For further information on the technology, product and business term, please contact Telink Semiconductor Company www.telink-semi.com

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

telinksales@telink-semi.com

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

Contents

List of Figures

List of Tables

1 SDK 介绍

Telink BLE Multiple Connection SDK(泰凌低功耗蓝牙多连接 SDK) 提供基于 B85m 系列和 B91 系列 MCU 的 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。

两个 SDK 内容基本一致，分别称为 Telink B85m BLE Multiple Connection SDK 和 Telink B91 BLE Multiple Connection SDK，本文介绍时如遇到二者不一致的地方，会进行说明。

除了 BLE Stack 部分，Telink BLE Multiple Connection SDK 中其他功能模块（比如 Flash、时钟、GPIO、IR 等）都和 Telink BLE Single Connection SDK 是一样的，为了避免这些模块的重复介绍，请用户同时准备好对应的 Telink BLE Single Connection SDK Handbook。本文档在后面每一个模块的介绍时都会具体说明当前模块是否和 Telink BLE Single Connection SDK Handbook 中完全一致，如不一致也会详细介绍差异在哪里。

需要准备的 Telink BLE Single Connection SDK handbook 包括 (点击 Handbook 链接，可以打开)：

- Telink B85m BLE Single Connection SDK Handbook
- Telink B91 BLE Signle Connection SDK Handbook。

Telink BLE Multiple Connection SDK 中的 Mulitple 指多个（大于等于 1）Master 或 Slave 角色共存，比如自身同时作为 4 个 Master 和 3 个 Slave（简称 M4S3）。

注意：

> 这些称谓都是从本地设备自身扮演的角色命名。

1.1 适用 IC 介绍

Telink BLE Multiple Connection SDK 适用如下几种型号的 MCU，B85m 系列（8253/8258/8273/8278）和 B91 系列。它们之间的 Flash size 和 SRAM size 不一样，如下所示。需要注意在使用不同 MCU 时，要选择不同的 bootloader 文件（bootloader 文件在1.3 节讲解）。

Table 1.1: MCU 资源情况

MCU	Flash size	SRAM size
8253	512 KB	48 KB
8258	512 KB / 1 MB	64 KB
8273	512 KB	64 KB
8278	1 MB	64 KB
B91	1 MB / 2 MB	128 KB + 128 KB

1.2 软件组织架构

Telink BLE Multiple Connection SDK 软件架构包括应用层和 BLE Stack 两部分。

1.2.1 文件结构

在 IDE 中导入 Telink BLE Multiple Connection SDK 后，显示的文件组织结构如下图所示（以 Telink B85m BLE Multiple Connection SDK 为例）。顶层文件夹有 8 个：algorithm，application，boot，common，drivers，proj_lib，stack，vendor。

> 官方推荐 IDE：B85m 系列使用 Telink IDE，B91 系列使用 Andes Telink_DRS。

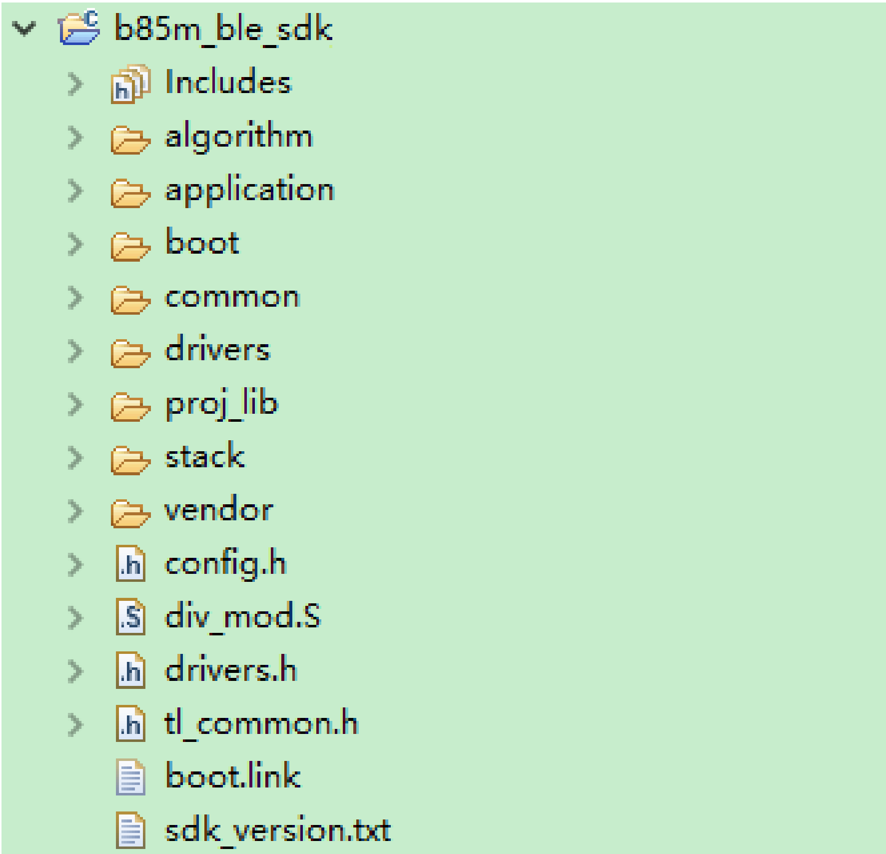


Figure 1.1: SDK 文件结构

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 print、keyboard 等。

boot：提供 MCU 的 software bootloader，即 MCU 上电启动或 deep sleep 唤醒后的汇编处理程序，为后面 C 程序的运行做准备。

common：提供一些通用的跨平台的处理函数，如内存处理函数、字符串处理函数等。

drivers：提供 MCU 外设驱动程序，如 Clock、Flash、I2C、USB、GPIO、UART 等。

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。

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 demo code 或者用户自己的 code。

Telink B85m BLE Multiple Connection SDK 提供了 6 个 demo，分别是 b85m_demo、b85m_controller、b85m_feature、b85m_m1s1、b85m_master_dongle、b85m_slave。这些 demo 工程在 vendor 文件夹下，如下图所示。

> b85m_m1s1 和 b85m_slave 支持 suspend mode 和 deepsleep retention mode，其余 demo 支持 suspend mode，不支持 deepsleep retention mode。

Figure 1.2: B85m 多连接 SDK 的 Demo 工程

Telink B91 BLE Multiple Connection SDK 提供了 5 个 demo，分别是 B91_demo、B91_controller、B91_feature、B91_master_dongle、B91_slave。这些 demo 工程在 vendor 文件夹下，如下图所示。

> B91_slave 默认配置支持 suspend mode 和 deepsleep retention mode，其余 demo 默认配置支持 suspend mode，如果希望支持 deepsleep retention mode 请参考 B91_slave。

Figure 1.3: B91 多连接 SDK 的 Demo 工程

以 b85m_demo 为例来讲解 demo 文件结构，文件构成如下图所示：

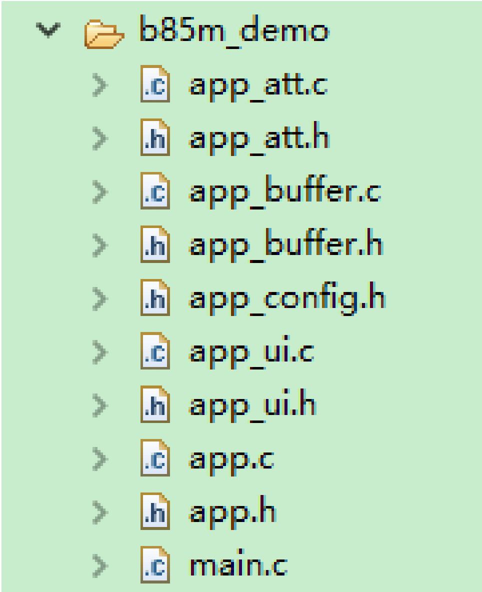


Figure 1.4: b85m_demo 文件结构

1.2.2 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。中断处理函数是系统触发中断时候的入口函数。

```
_attribute_ram_code_ int main(void)
{
    #if (BLE_APP_PM_ENABLE)
        blc_pm_select_internal_32k_crystal();
    #endif

    //MCU 最基本的硬件初始化
    #if (MCU_CORE_TYPE == MCU_CORE_825x)
        cpu_wakeup_init();
    #elif (MCU_CORE_TYPE == MCU_CORE_827x)
        cpu_wakeup_init(DCDC_MODE, EXTERNAL_XTAL_24M);
    #endif

    /* detect if MCU is wake_up from deep retention mode */
    int deepRetWakeUp = pm_is_MCU_deepRetentionWakeup();  //MCU deep retention wakeUp

    clock_init(SYS_CLK_TYPE); //时钟初始化

    rf_drv_init(RF_MODE_BLE_1M); //RF 初始化

    gpio_init(!deepRetWakeUp); //GPIO 初始化
```

```
    if( deepRetWakeUp ){ //MCU wake_up from deepSleep retention mode
        user_init_deepRetn ();
    }
    else{ //MCU power_on or wake_up from deepSleep mode
        user_init_normal ();
    }

    /* load customized freq_offset cap value.*/
    blc_app_loadCustomizedParameters();

    irq_enable();

    while(1)
    {
        main_loop ();//包括 ble 收发包处理和 user 的任务
    }
    return 0;
}
```

1.2.3 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。

后面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。

1.2.4 application file

app.c：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：这个文件提供了 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：该文件主要提供了按键功能。

app_buffer.c：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。

1.2.5 common file

路径在 vendor/common。

blt_soft_timer.c：该文件提供了软件定时器的实现方案。

custom_pair.c：该文件提供了泰凌自定义的一套 pair 的方案。

device_manage.c：该文件主要是连接设备信息的管理（例如：connection handle, attribute handle, BLE device address, address type 等）。这些信息是用户在开发应用时需要使用的。

simple_sdp.c：该文件提供了 Master role 简单的 SDP（Service Discovery Protocol）实现方案。

1.2.6 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 协议栈相关的数据和事件。

1.3 Software bootloader 介绍

Telink BLE Multiple Connection SDK 中，不同型号的 MCU 对应不同的 bootloader 文件。bootloader 文件存放在 boot 文件夹下。Telink 的 bootloader 文件是由两部分构成，link 文件和 cstartup.S 汇编文件。

1.3.1 B85m 多连接的 bootloader 文件

Telink B85m BLE Multiple Connection SDK 的 bootloader 文件如下图所示。

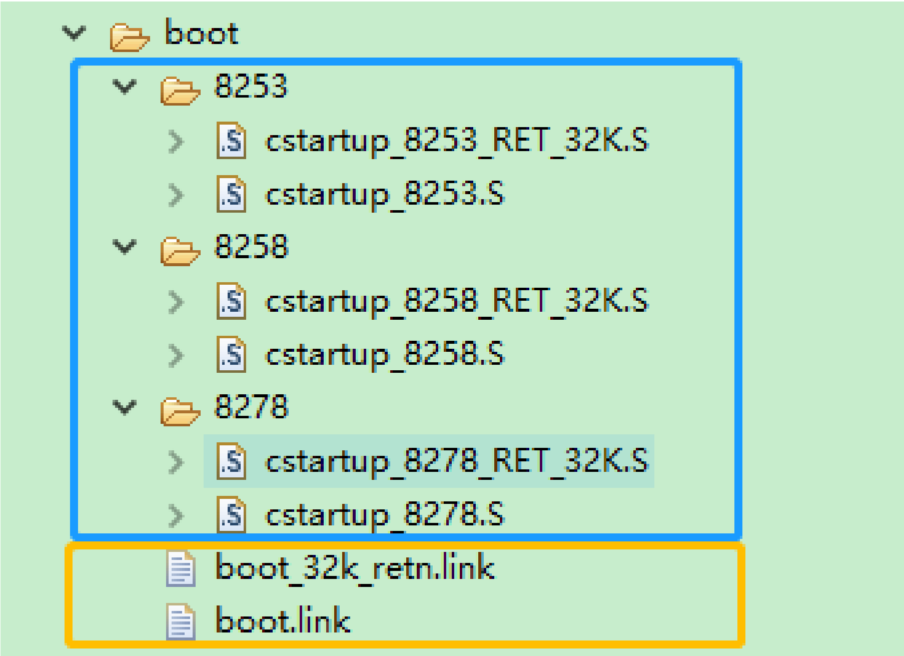


Figure 1.5: B85m 多连接 SDK 的 bootloader 文件

Telink B85m BLE Multiple Connection SDK 中的 link 文件有两个，boot.link（支持 suspend mode）和 boot_32k_retn.link（支持 suspend mode 和 deepsleep retention mode）。

注意：

- 编译工程时，真正生效的 link 文件是 b85m_ble_sdk/boot.link。
- 只有 b85m_slave 和 b85m_m1s1 支持 deepsleep retention mode，使用这两个 demo 时需要复制 b85m_ble_sdk/boot/boot_32k_retn.link 的内容至 b85m_ble_sdk/boot.link。
- 其余 demo 使用时需要复制 b85m_ble_sdk/boot/boot.link 的内容至 b85m_ble_sdk/boot.link。

Telink B85m BLE Multiple Connection SDK 中的 cstartup.S 文件依据芯片 SRAM 资源大小而不同。每种 MCU 对应 2 个 cstartup.S 文件，'RET_32K' 表示支持 deepsleep retention 32K mode。由于 8273 与 8278 的 SRAM 都是 64 KB，所以它们的 cstartup.S 文件完全相同，统一用 8278 的配置。

以 b85m_demo，MCU 选择 8258 为例，用户设置对应的 cstartup.S 文件方法如下：

在 boot/8258 文件夹下找到"cstartup_8258.S"文件，在该文件中找到宏 MCU_STARTUP_8258，然后在工程的属性设置窗口中按下图所示的方式配置即可。

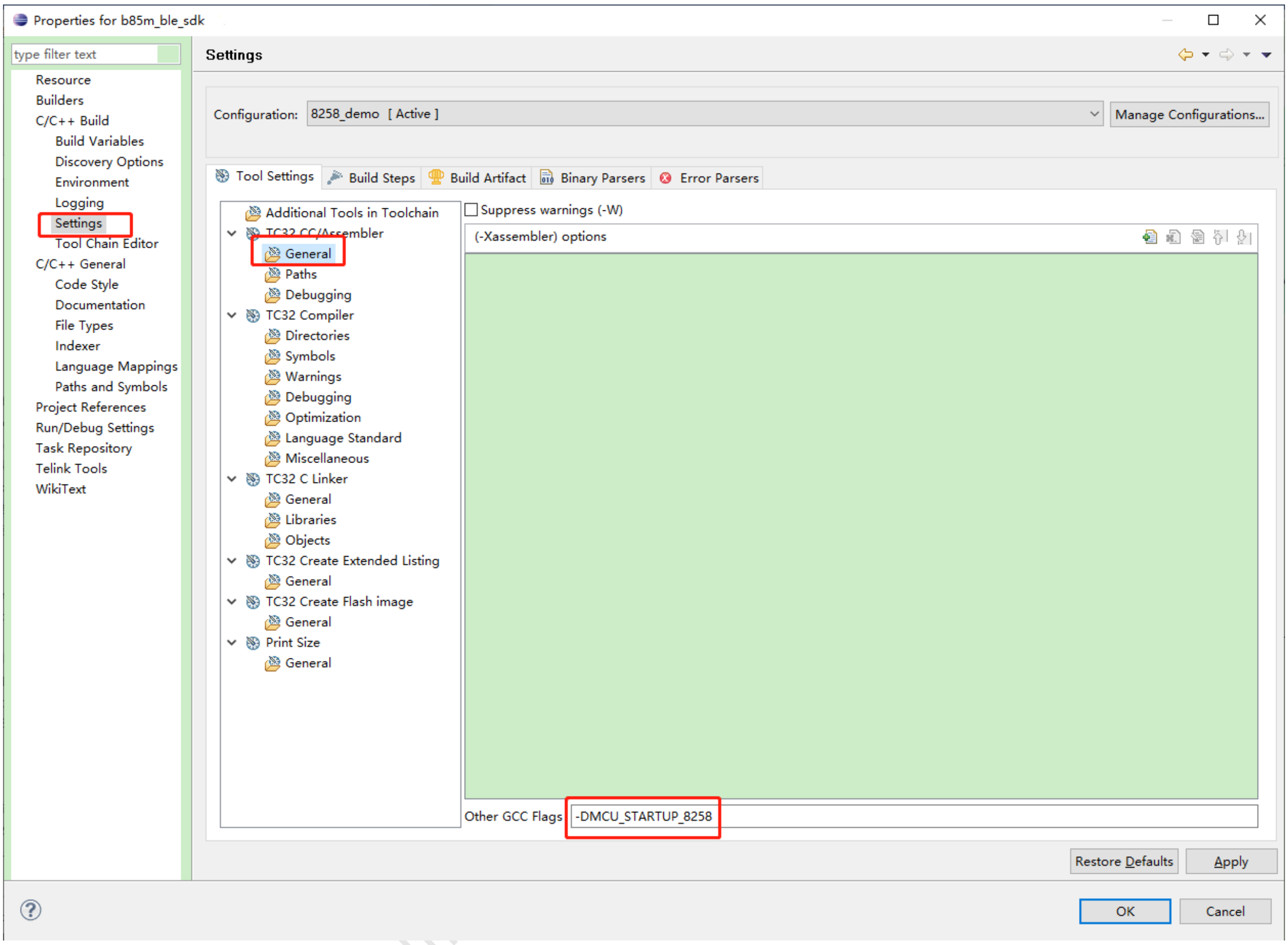


Figure 1.6: cstartup 选择

B85m 系列不同 MCU 的 bootloader 文件选择关系见下表。

Table 1.2: B85m 系列 MCU bootloader 文件选择

MCU	b85m_demo、b85m_controller、b85m_feature、b85m_master_dongle	b85m_m1s1、b85m_slave
8253	boot.link cstartup_8253.S	boot_32k_retn.link cstartup_8253_RET_32K.S
8258	boot.link cstartup_8258.S	boot_32k_retn.link cstartup_8258_RET_32K.S
8273/8278	boot.link cstartup_8278.S	boot_32k_retn.link cstartup_8278_RET_32K.S

1.3.2 B91 多连接的 bootloader 文件

Telink B91 BLE Multiple Connection SDK 的 bootloader 文件如下图所示：

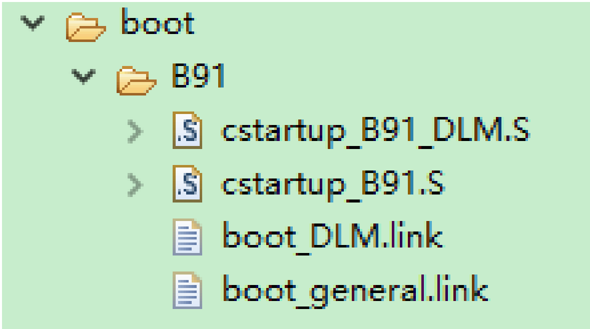


Figure 1.7: B91 多连接 SDK 的 bootloader 文件

默认运行的是 cstartup_B91.S 及 boot_general.link 文件，此时 SDK 会占用 I-SRAM 及 D-SRAM 空间，I-SRAM 中包括 retention_reset、aes_data、retention_data、ramcode、和 unused I-SRAM area。D-SRAM 中包括 data、sbss、bss、heap、unused D-SRAM area 和 stack。

如果想把 128K 的 D-SRAM 空间全部留给用户使用，需要做如下更改。

(1) 将 cstartup_B91.S 文件开始的 #if 1 改成 #if 0。

(2) 将 cstartup_B91_DLM.S 文件开始的 #if 0 改成 #if 1。

(3) 将 eagle_ble_sdk/boot/B91/boot_DLM.link 的内容复制到 eagle_ble_sdk/boot.link 文件中。

注意：

> Telink B91 BLE Multiple Connection SDK 默认配置 deepsleep retention 64K mode。

1.4 library 介绍

1.4.1 B85m 多连接的 library

Telink B85m BLE Multiple Connection SDK 配置 project 对应的 library 如下图所示。

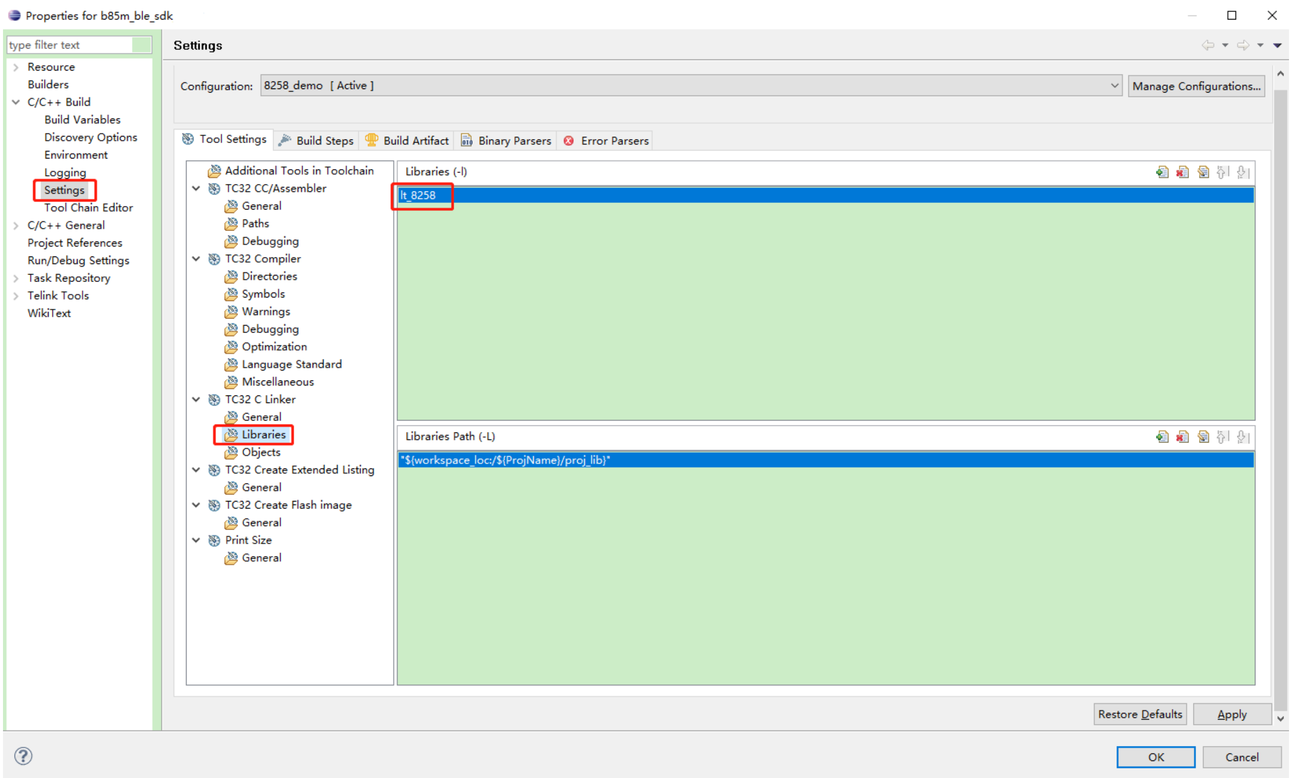


Figure 1.8: B85m 多连接 SDK 选择 project 对应的 library

下图为 B85m 多连接 SDK 的 proj_lib 文件夹目前已经提供的 library。

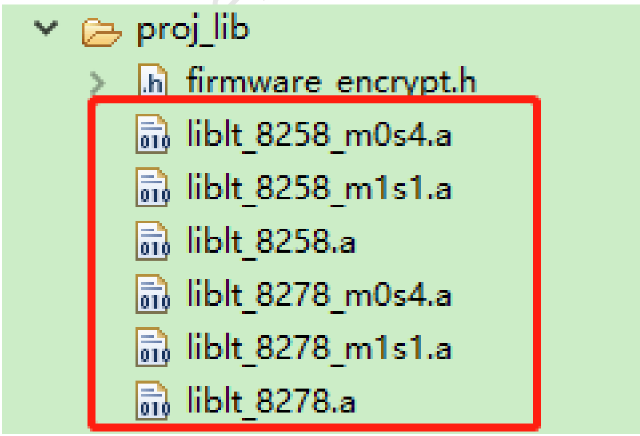


Figure 1.9: B85m 多连接 SDK 提供的 library

下表为 Telink B85m BLE Multiple Connection SDK 不同 MCU、demo 与 library 的适配关系。

Table 1.3: B85m 多连接 SDK 不同 MCU、demo 与 library 适配关系

MCU	b85m_demo、b85m_controller、b85m_feature、b85m_master_dongle	b85m_m1s1	b85m_slave
8253	liblt_8258.a	liblt_8258_m1s1.a	liblt_8258_m0s4.a
8258	liblt_8258.a	liblt_8258_m1s1.a	liblt_8258_m0s4.a
8273	liblt_8278.a	liblt_8278_m1s1.a	liblt_8278_m0s4.a
8278	liblt_8278.a	liblt_8278_m1s1.a	liblt_8278_m0s4.a

1.4.2 B91 多连接的 library

下图为 Telink B91 BLE Multiple Connection SDK 的 proj_lib 文件夹目前已经提供的 library。

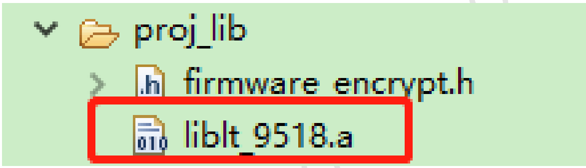


Figure 1.10: B91 多连接 SDK 提供的 library

Telink B91 BLE Multiple Connection SDK 所有的 demo 都使用同一个 library。

2 MCU 基础模块

2.1 MCU 地址空间

2.1.1 MCU 地址空间分配

请参考 Telink BLE Single Connection SDK Handbook 对应章节的介绍，包括 B85m 系列（825x + 827x）和 B91 系列。

2.1.2 SRAM 空间分配

请参考 Telink BLE Single Connection SDK Handbook 对应章节的介绍，包括 B85m 系列（825x + 827x）和 B91 系列。

Telink B85m BLE Multiple Connection SDK ，只有 b85m_m1s1 和 b85m_slave 支持 deepsleep retention 功能，b85m_m1s1 和 b85m_slave 的 Sram 空间分配请参考 Telink B85m BLE Single Connection SDK。其余 demo 只用到 suspend 和 deepsleep 功能，不支持 deepsleep retention 功能的 Sram 空间分配如下图。

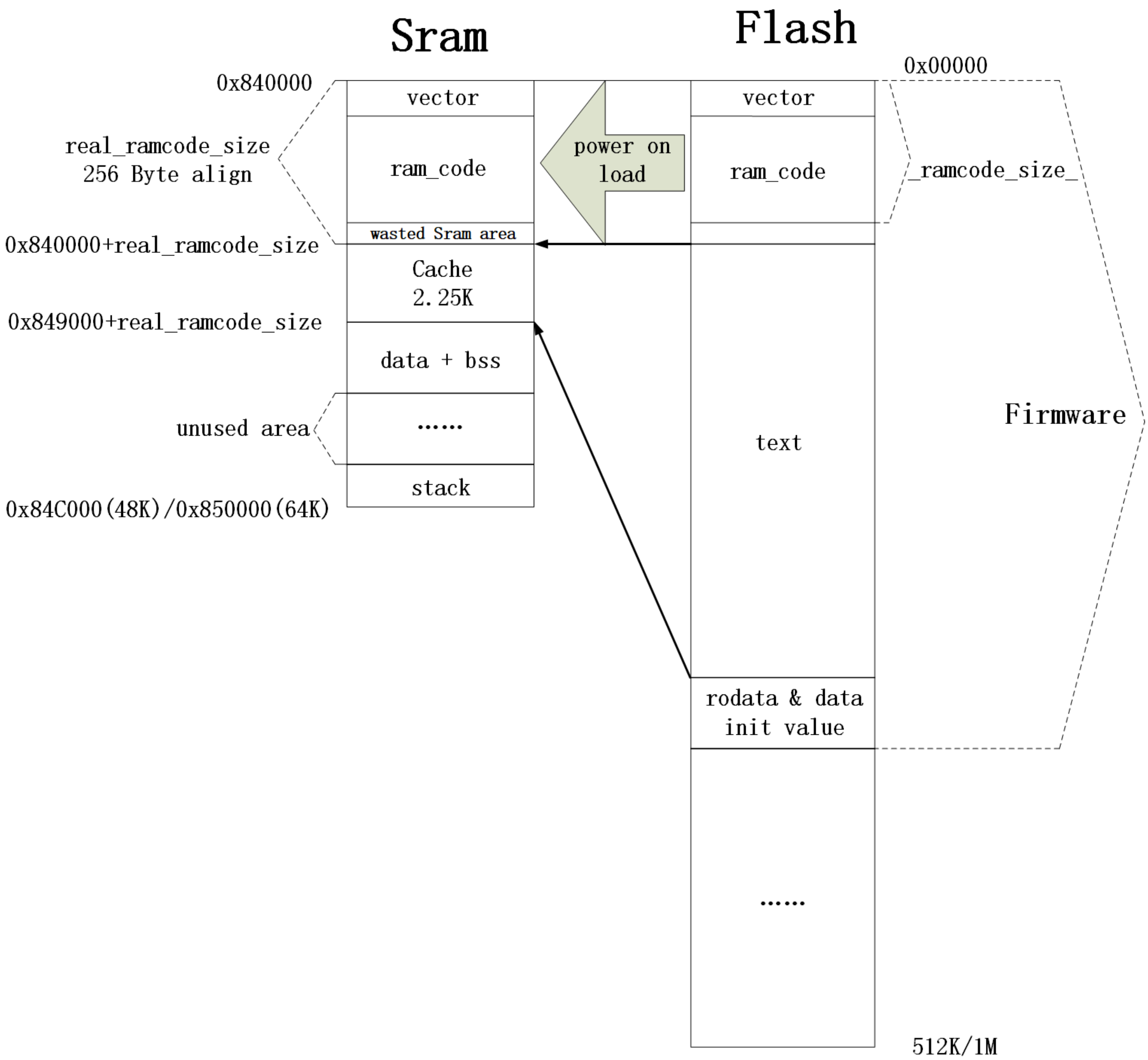


Figure 2.1: 不支持 deepsleep retention 功能的 Sram 空间分配 & Firmware 空间分配

Flash 中 Firmware 包括 vector、ramcode、text、rodata 和 data initial value。

Sram 中包括 vector、ramcode、Cache、data、bss、unused sram area 和 stack。

Sram 中的 vector/ramcode 是 Flash 中 vector/ramcode 的拷贝。

2.1.3 MCU 地址空间访问

请参考 Telink BLE Single Connection SDK Handbook 对应章节的介绍，包括 B85m 系列（825x + 827x）和 B91 系列。

2.1.4 SDK Flash 空间的分配

多连接由于功能相对复杂用到的 flash 存储区域较多，不采用自动读 Flash size 进行 Flash 配置的方法，而是要求用户必须根据自己产品实际情况手动配置 Flash size。所有 demo 里 app_config.h 都只是预留了默认的配置，**用户必须检查确认甚至修改"FLASH_SIZE_CONFIG"的定义。**

```
#define FLASH_SIZE_512K                                    0x80000
#define FLASH_SIZE_1M                                      0x100000
#define FLASH_SIZE_2M                                      0x200000

#define FLASH_SIZE_CONFIG                                  FLASH_SIZE_1M
```

Flash 存储信息以一个 sector 的大小（4K byte）为基本的单位，因为 Flash 的擦除是以 sector 为单位的（擦除函数为 flash_erase_sector），理论上同一类的信息需要存储在一个 sector 里面，不同类的信息需要存放在不同的 sector（防止擦除信息时将其他类的信息误擦除）。所以建议 user 在使用 Flash 存储定制信息时遵循"不同类信息放在不同 sector"的原则。

Telink BLE Multiple Connection SDK 中有四类信息需要存储在 Flash 中，分别是 MAC，校准信息，加密配对信息和 SDP 信息。这些参数在 SDK 中都默认分配了不同的 Flash 空间的。

MAC 和校准信息存放的 Flash 空间会随芯片 Flash 的大小不同而不同，默认情况下，对于 512K Flash 的 B85m 系列 (8253/8258/8273) 芯片来说，MAC 存放在 0x76000 开始的 4K Flash 空间中，校准信息存放在 0x77000 开始的 4K Flash 空间中。对于 512K Flash 的 B91 系列芯片来说，MAC 存放在 0x7F000 开始的 4K Flash 空间中，校准信息存放在 0x7E000 开始的 4K Flash 空间中。对于 1M Flash 的芯片（8258/8278/B91 系列）来说，MAC 存放在 0xFF000 开始的 4K Flash 空间中，校准信息存放在 0xFE000 开始的 4K Flash 空间中。对于 2M Flash 的 B91 系列芯片来说，MAC 存放在 0x1FF000 开始的 4K Flash 空间中，校准信息存放在 0x1FE000 开始的 4K Flash 空间中。

```
/**
 * @brief   512 K Flash MAC address and calibration data area
 */
#if (MCU_CORE_TYPE == MCU_CORE_9518)
    #define CFG_ADR_MAC_512K_FLASH                                  0x7F000
    #define CFG_ADR_CALIBRATION_512K_FLASH                          0x7E000
#else
    #define CFG_ADR_MAC_512K_FLASH                                  0x76000
    #define CFG_ADR_CALIBRATION_512K_FLASH                          0x77000
#endif

/**
 * @brief   1 M Flash MAC address and calibration data area
 */
#define CFG_ADR_MAC_1M_FLASH                                        0xFF000
#define CFG_ADR_CALIBRATION_1M_FLASH                                0xFE000

/**
 * @brief   2 M Flash MAC address and calibration data area
 */
#define CFG_ADR_MAC_2M_FLASH                                        0x1FF000
#define CFG_ADR_CALIBRATION_2M_FLASH                                0x1FE000
```

Figure 2.2: MAC 和校准信息存放的 Flash 空间

加密配对信息和 SDP 信息也存储在独立的 Flash 空间上，SDK 能根据用户设置的 Flash 大小自动配置相应的默认位置。默认情况下，对于 512K Flash 的芯片来说，加密配对信息存放在 0x78000 开始 16K Flash 空间中，自定义绑定信息存放在 0x7C000 开始 4K Flash 空间中，SDP 信息存放在 0x7D000 开始的 8K Flash 空间中。对于 1M Flash 的芯片来说，加密配对信息存放在 0xFA000 开始 16K Flash 空间中，自定义绑定信息存放在 0xF8000 开始 4K Flash 空间中，SDP 信息存放在 0xF6000 开始的 8K Flash 空间中。对于 2M Flash 的芯片来说，加密配对信息存放在 0x1FA000 开始 16K Flash 空间中，自定义绑定信息存放在 0x1F8000 开始 4K Flash 空间中，SDP 信息存放在 0x1F6000 开始的 8K Flash 空间中。

SDK 能根据用户设置的 Flash 大小自动配置相应的 MAC 和校准值存储空间。用户也可以根据自己的需要修改 vendor/common/blt_common.h 中的相应的宏来修改默认的 MAC 和校准值存储空间，此时需要注意是要相应的修改泰凌量产治具上的烧写地址。

```
#if(FLASH_SIZE_CONFIG == FLASH_SIZE_512K)

    /* MAC and calibration data area */
    #define     CFG_ADR_MAC                                 CFG_ADR_MAC_512K_FLASH          //can not change this value
    #define     CFG_ADR_CALIBRATION                         CFG_ADR_CALIBRATION_512K_FLASH  //can not change this value

    #if (MCU_CORE_TYPE == MCU_CORE_9518)
        //TODO
    #else
        /* SMP paring and key information area */
        #ifndef FLASH_ADR_SMP_PAIRING
        #define FLASH_ADR_SMP_PAIRING                           0x78000
        #endif

        #ifndef FLASH_SMP_PAIRING_MAX_SIZE
        #define FLASH_SMP_PAIRING_MAX_SIZE                      (2*4096)  //normal 8K + backup 8K = 16K
        #endif


        /* bonding slave information for custom pair area */
        #ifndef FLASH_ADR_CUSTOM_PAIRING
        #define FLASH_ADR_CUSTOM_PAIRING                        0x7C000
        #endif

        #ifndef FLASH_CUSTOM_PAIRING_MAX_SIZE
        #define FLASH_CUSTOM_PAIRING_MAX_SIZE                   4096
        #endif


        /* bonding slave GATT service critical information area */
        #ifndef FLASH_SDP_ATT_ADRRESS
        #define FLASH_SDP_ATT_ADRRESS                           0x7D000    //for master: store peer slave device's ATT handle
        #endif

        #ifndef FLASH_SDP_ATT_MAX_SIZE
        #define FLASH_SDP_ATT_MAX_SIZE                          (2*4096)   //8K flash for ATT HANLDE storage
        #endif
    #endif
```

Figure 2.3: 512K Flash 默认 Flash 存储地址

```
#elif(FLASH_SIZE_CONFIG == FLASH_SIZE_1M)

    /* MAC and calibration data area */
    #define CFG_ADR_MAC                              CFG_ADR_MAC_1M_FLASH          //can not change this value
    #define CFG_ADR_CALIBRATION                      CFG_ADR_CALIBRATION_1M_FLASH  //can not change this value

    /* SMP paring and key information area */
    #ifndef FLASH_ADR_SMP_PAIRING
    #define FLASH_ADR_SMP_PAIRING                    0xFA000
    #endif

    #ifndef FLASH_SMP_PAIRING_MAX_SIZE
    #define FLASH_SMP_PAIRING_MAX_SIZE               (2*4096)  //normal 8K + backup 8K = 16K
    #endif


    /* bonding slave information for custom pair area */
    #ifndef FLASH_ADR_CUSTOM_PAIRING
    #define FLASH_ADR_CUSTOM_PAIRING                 0xF8000
    #endif

    #ifndef FLASH_CUSTOM_PAIRING_MAX_SIZE
    #define FLASH_CUSTOM_PAIRING_MAX_SIZE            4096
    #endif


    /* bonding slave GATT service critical information area */
    #ifndef FLASH_SDP_ATT_ADRRESS
    #define FLASH_SDP_ATT_ADRRESS                    0xF6000    //for master: store peer slave device's ATT handle
    #endif

    #ifndef FLASH_SDP_ATT_MAX_SIZE
    #define FLASH_SDP_ATT_MAX_SIZE                   (2*4096)   //8K flash for ATT HANLDE storage
    #endif
```

Figure 2.4: 1M Flash 默认存储信息空间

```
#elif(FLASH_SIZE_CONFIG == FLASH_SIZE_2M)
/* MAC and calibration data area */
    #define CFG_ADR_MAC                              CFG_ADR_MAC_2M_FLASH          //can not change this value
    #define CFG_ADR_CALIBRATION                      CFG_ADR_CALIBRATION_2M_FLASH  //can not change this value

    /* SMP paring and key information area */
    #ifndef FLASH_ADR_SMP_PAIRING
    #define FLASH_ADR_SMP_PAIRING                    0x1FA000
    #endif

    #ifndef FLASH_SMP_PAIRING_MAX_SIZE
    #define FLASH_SMP_PAIRING_MAX_SIZE               (2*4096)  //normal 8K + backup 8K = 16K
    #endif


    /* bonding slave information for custom pair area */
    #ifndef FLASH_ADR_CUSTOM_PAIRING
    #define FLASH_ADR_CUSTOM_PAIRING                 0x1F8000
    #endif

    #ifndef FLASH_CUSTOM_PAIRING_MAX_SIZE
    #define FLASH_CUSTOM_PAIRING_MAX_SIZE            4096
    #endif


    /* bonding slave GATT service critical information area */
    #ifndef FLASH_SDP_ATT_ADRRESS
    #define FLASH_SDP_ATT_ADRRESS                    0x1F6000    //for master: store peer slave device's ATT handle
    #endif

    #ifndef FLASH_SDP_ATT_MAX_SIZE
    #define FLASH_SDP_ATT_MAX_SIZE                   (2*4096)   //8K flash for ATT HANLDE storage
    #endif
#endif
```

Figure 2.5: 2M Flash 默认存储信息空间

2.2 时钟模块

请参考 Telink BLE Single Connection SDK Handbook 对应章节的介绍，包括 B85m 系列（825x + 827x）和 B91 系列。

多连接 SDK 与单连接 SDK 不同的地方在于：多连接 SDK 只能使用 CLK_32M 和 CLK_48M 这两种系统时钟，其他时钟速度太慢，无法满足多连接 SDK 的运行。

2.3 GPIO 模块

请参考 Telink BLE Single Connection SDK Handbook 对应章节的介绍，包括 B85m 系列（825x + 827x）。

2.4 中断嵌套

Telink B85m BLE Multiple Connection SDK 不支持中断嵌套，所有的中断统一优先级，先到先响应，具体请参考 Telink B85m Single Connection SDK Handbook 对应章节的介绍。

Telink B91 BLE Multiple Connection SDK 支持中断嵌套功能，请参考 Telink B91 BLE Single Connection SDK Handbook 对应章节的介绍。

3 BLE 模块

本手册以 Bluetooth Core Specification 5.3 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。

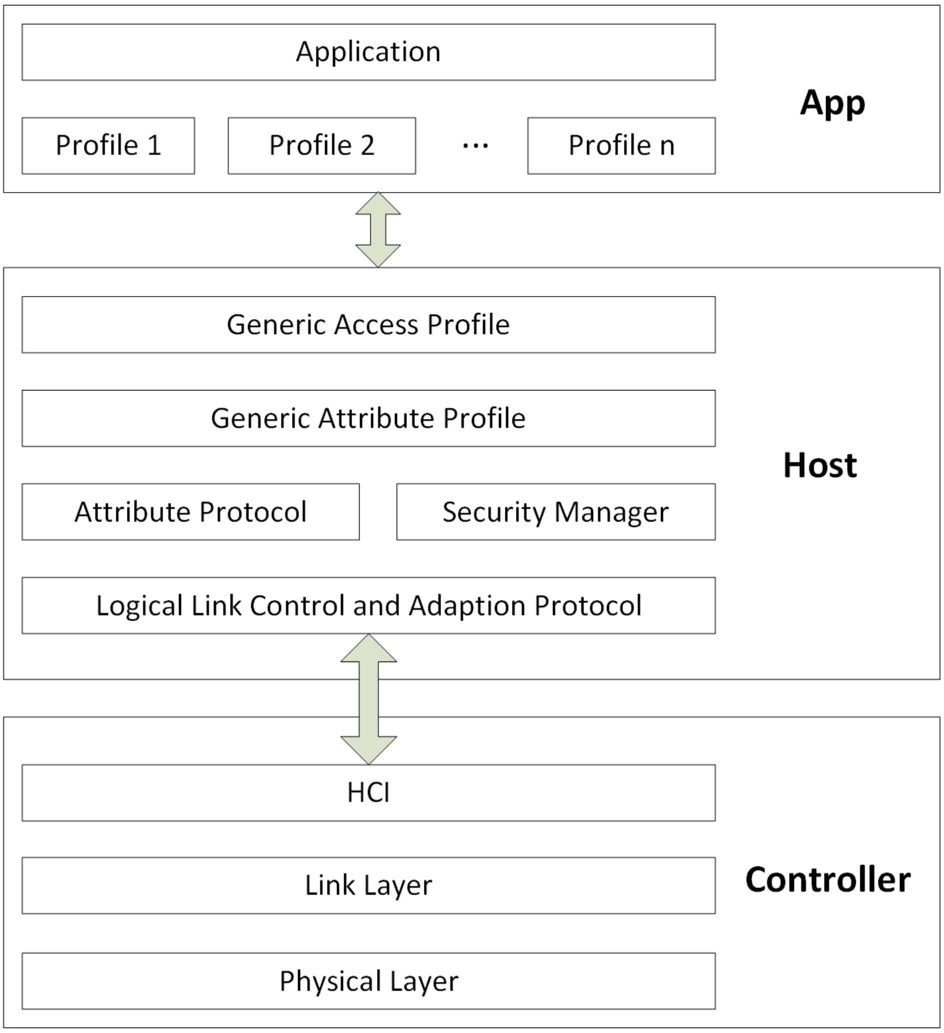


Figure 3.1: BLE SDK 软件架构

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。

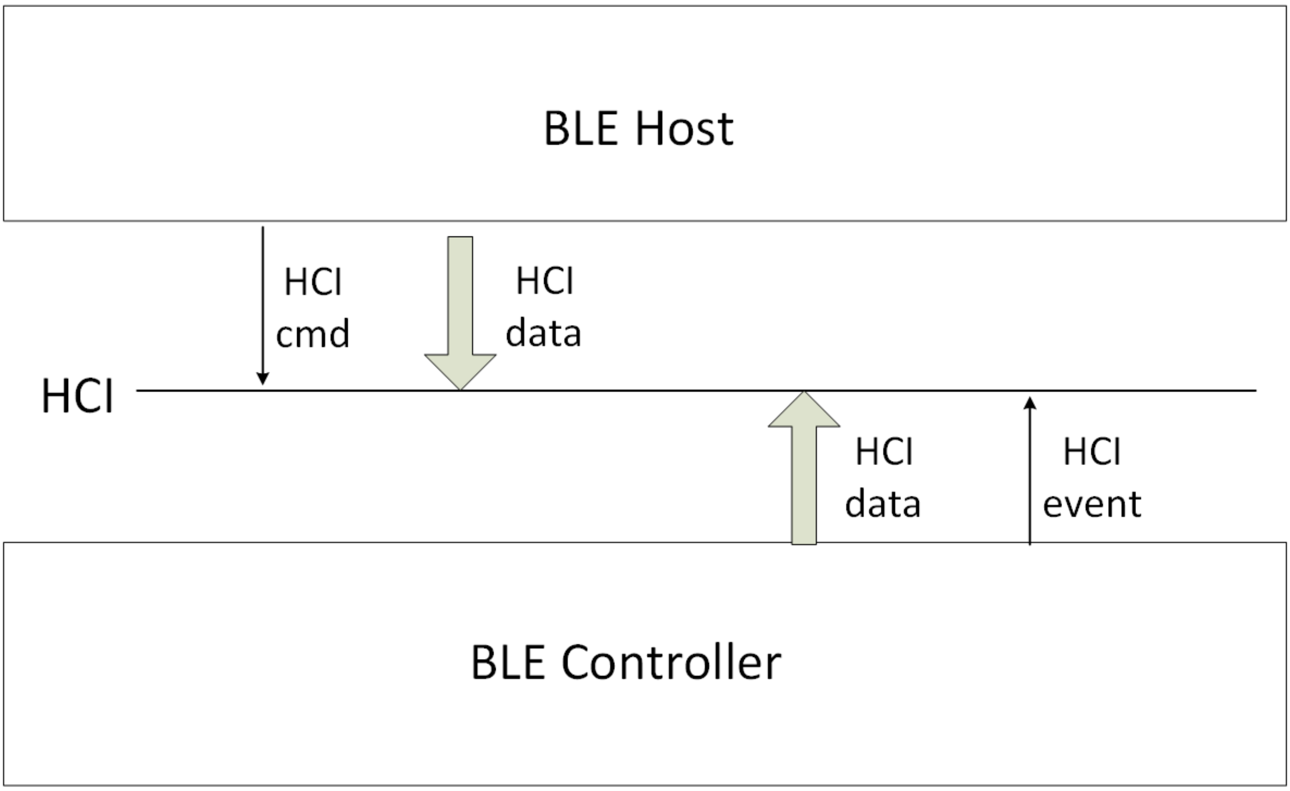


Figure 3.2: Host 和 Controller 的 HCI 数据交互

(1) BLE Host 通过 HCI cmd 去操作设置 Controller，这些 HCI cmd 对应本章后面将要介绍的 controller API。

(2) Controller 通过 HCI 向 host 上报各种 HCI event，本章也会具体介绍。

(3) Host 将需要发送给对方设备的数据通过 HCI 传送到 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

Telink BLE Multiple Connection SDK 支持标准的 BLE controller，包括 HCI、PHY（Physical Layer）和 LL（Link Layer）。

Telink BLE Multiple Connection 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 Master role 和 4 个 Slave role。

controller 架构图如下：

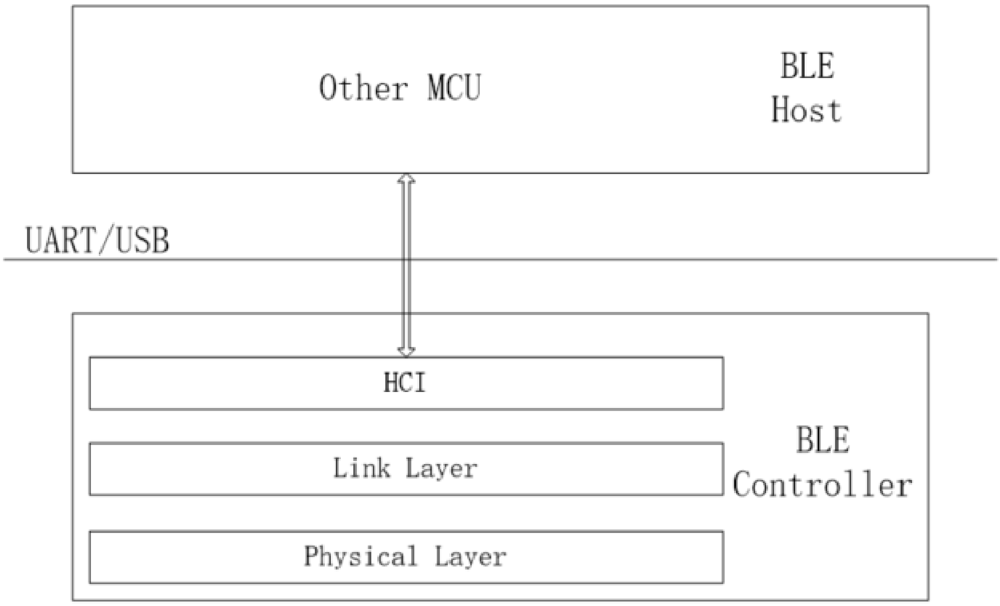


Figure 3.3：Telink HCI 架构

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

Telink BLE Multiple Connection SDK 提供 BLE Multiple Connection Whole Stack(Controller + Host) 参考设计，只有对于 Master 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 b85m_demo、b85_m1s1、b91_demo、b91_slave 都是基于该架构。

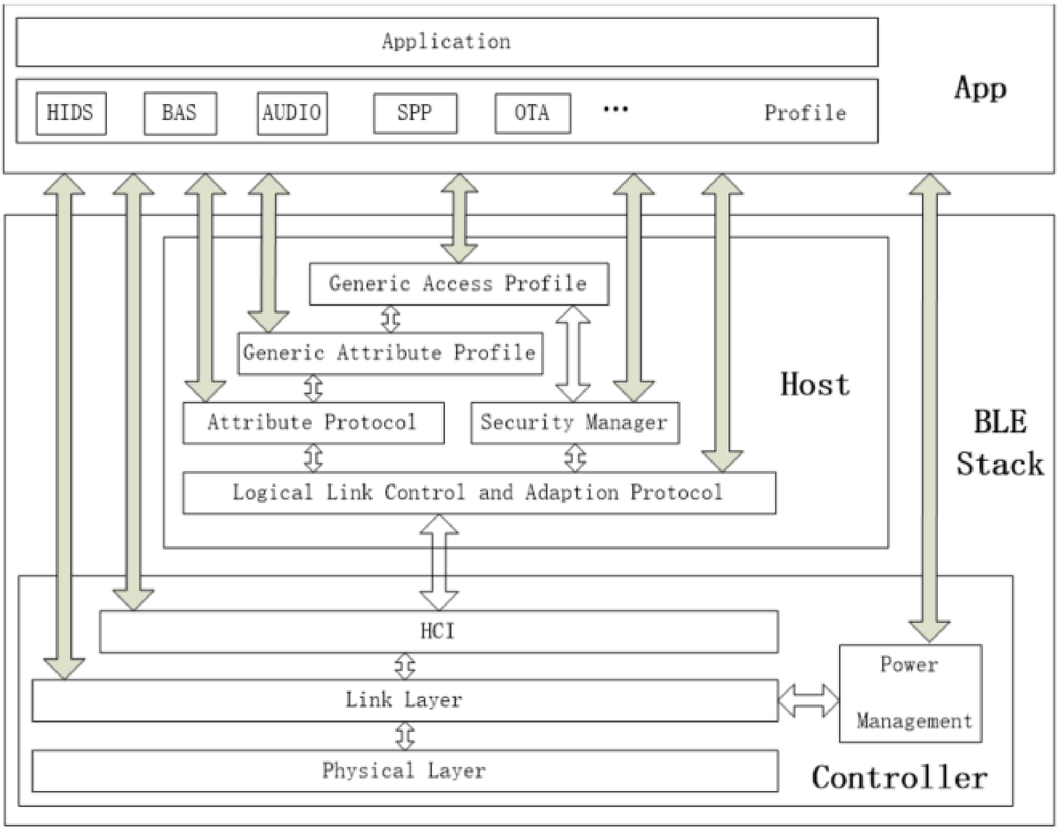


Figure 3.4: Telink BLE Multiple Connection Whole Stack 架构

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 supportedMaxMasterNum & supportedMaxSlaveNum

Telink BLE Multiple Connection SDK 将 Connection Master role 最大数量称为 supportedMaxMasterNum，将 Connection Slave role 最大数量称为 supportedMaxSlaveNum，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	supportedMaxMaster-Num	supportedMaxSlaveNum
B91	liblt_9518	4	4
825x	liblt_8258	4	4
	liblt_8258_m0s4	0	4
	liblt_8258_m1s1	1	1
827x	liblt_8278	4	4
	liblt_8278_m0s4	0	4
	liblt_8278_m1s1	1	1

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

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

3.2.1.2 appMaxMasterNum & appMaxSlaveNum

在 supportedMaxMasterNum 和 supportedMaxSlaveNum 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 Master 和 Slave 数量，分别称为 appMaxMasterNum 和 appMaxSlaveNum。

```
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 appMaxMasterNum 和 appMaxSlaveNum 必须小于或等于 supportedMaxMasterNum 和 supported-MaxSlaveNum。

参考例程设计上在初始化的时候都使用了该 API：

```
blc_ll_setMaxConnectionNumber( MASTER_MAX_NUM, SLAVE_MAX_NUM);
```

用户需要在 app_config.h 中定义自己的 appMaxMasterNum 和 appMaxSlaveNum，即 SDK 中的 MAS-TER_MAX_NUM 和 SLAVE_MAX_NUM。

```
#define MASTER_MAX_NUM                        4
#define SLAVE_MAX_NUM                         4
```

appMaxMasterNum 和 appMaxSlaveNum 能够节省 MCU 的各种资源，比如针对 M4S4 的 library，用户如果只需要用到 M3S2，将 MASTER_MAX_NUM 和 SLAVE_MAX_NUM 分别设为 3 和 2 后：

(1) 节省 SRAM 资源

Link Layer TX Master FIFO 和 TX Slave FIFO、L2CAP Master MTU buffer 和 L2CAP Slave MTU buffer 都是根据 appMaxMasterNum 和 appMaxSlaveNum 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 M4S4，Stack 必须等到 currentMasterNum 为 4 时才会停止 Scan 动作，必须等到 currentSlaveNum 为 4 时才会停止 Advertising 动作。而对于 M3S2，Stack 等到 currentMasterNum 为 3 时才会停止 Scan 动作，currentSlaveNum 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 currentMaxMasterNum & currentMaxSlaveNum

用户定义了 appMaxMasterNum 和 appMaxSlaveNum 后，确定了 Link Layer 运行时创建的 Master 和 Slave 最大数量。但 Master 和 Slave 在某一时刻的数量还是不确定的，比如 appMaxMasterNum 为 4 时，任何时刻 Master 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 Master 和 Slave 数量。

```
int blc_ll_getCurrentConnectionNumber(void);//Master + Slave connection number
int blc_ll_getCurrentMasterRoleNumber(void);//Master role number
int blc_ll_getCurrentSlaveRoleNumber(void);//Slave  role number
```

3.2.2 Link Layer 状态机

用户可以先参考 Telink 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 Slave role 和 Connection Master 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 Slave role、Connection Master role。

而对于 Telink BLE Multiple Connection SDK ，由于要同时支持多个 Master 和 Slave，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。

Telink BLE Multiple Connection 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

Telink BLE Multiple Connection 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_adr);
```

实参 public_adr 是 BLE public mac address 的指针。

其他几个状态（Advertising、Scanning、ACL Master、ACL Slave）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_initAclMasterRole_module (void);
void        blc_ll_initAclSlaveRole_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 Master 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。

Telink BLE Multiple Connection SDK Link Layer 状态机可以从两个角度去描述，一是 Advertising 和 Slave 的转换；二是 Scanning 和 Master 的转换；这两个角度之间互不影响。

以 M1S1 为例分析，假设用户的 appMaxMasterNum 和 appMaxSlaveNum 都是 1。M1S1 Advertising 和 Slave 切换的状态机如下：

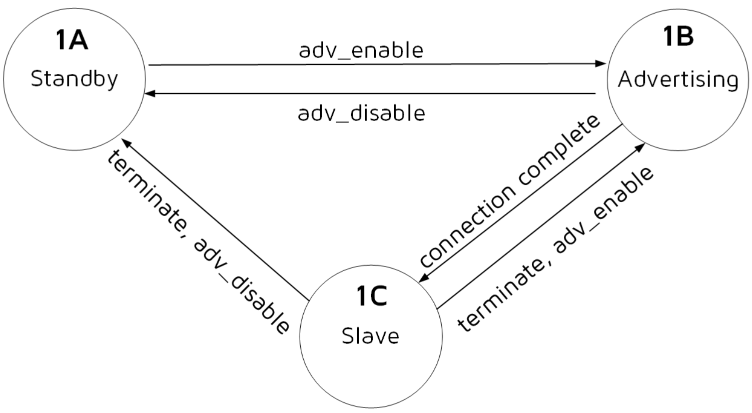


Figure 3.5: M1S1 Advertising 和 Slave 切换

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。

M1S1 Scanning 和 Master 切换的状态机如下：

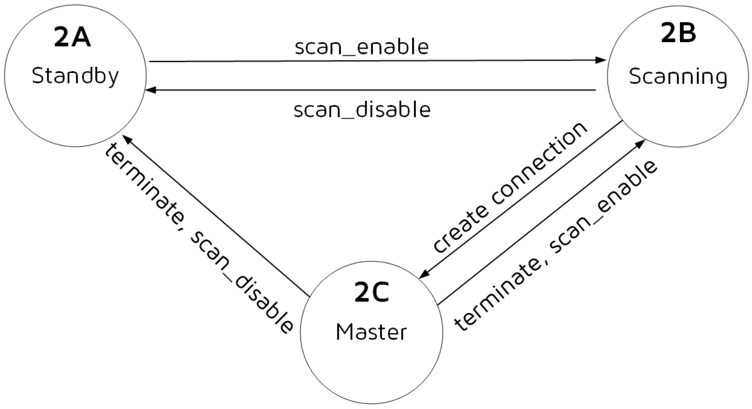


Figure 3.6: M1S1 Scanning 和 Master 切换

图中 scan_enable 和 scan_disable 指的是条件发生时，用户最后一次调用 blc_ll_setScanEnable(scan_enable, filter_duplicate) 设定的状态。

Advertising 和 Slave、Scanning 和 Master 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*3=9 种，如下表所示：

Table 3.2: M1S1 Link Layer 组合状态

	2A	2B	2C
1A	Standby	Scanning	Master
1B	Advertising	Advertising + Scanning	Advertising + Master
1C	Slave	Slave + Scanning	Slave + Master

以一个较为复杂的 M4S4 分析，假设用户的 appMaxMasterNum 和 appMaxSlaveNum 分别是 4 和 4，M4S4 Advertising 和 Slave 切换的状态机如下：

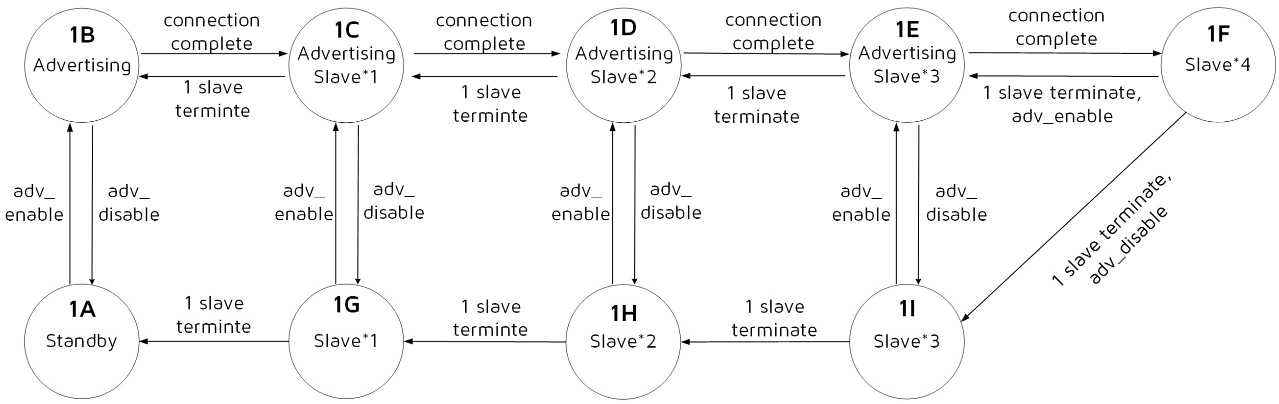


Figure 3.7: M4S4 Advertising 和 Slave 切换

M4S4 Scanning 和 Master 切换的状态机如下：

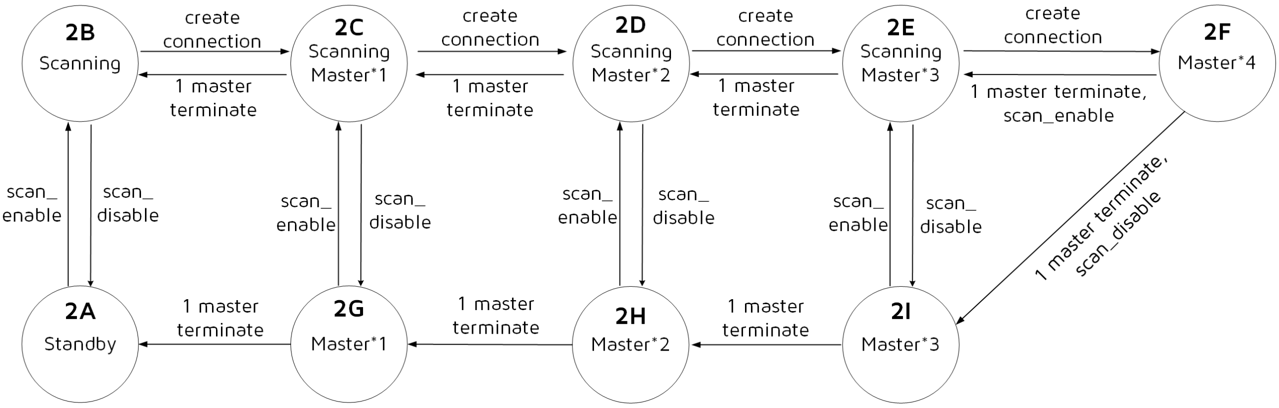


Figure 3.8: M4S4 Scanning 和 Master 切换

Advertising 和 Slave 有 9 种可能的状态，Scanning 和 Master 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*9=81 种，如下表所示：

Table 3.3: M4S4 Link Layer 组合状态

	2A	2B	2C	2D	2E	2F	2G	2H	2I
1A	Standby	Scanning	Scanning Master*1	Scanning Master*2	Scanning Master*3	Master*4	Master*1	Master*2	Master*3
1B	Adv	Adv Scanning	Adv Scanning Master*1	Adv Scanning Master*2	Adv Scanning Master*3	Adv Master*4	Adv Master*1	Adv Master*2	Adv Master*3
1C	Adv Slave*1	Adv Slave*1 Scanning	Adv Slave*1 Scanning Master*1	Adv Slave*1 Scanning Master*2	Adv Slave*1 Scanning Master*3	Adv Slave*1 Master*4	Adv Slave*1 Master*1	Adv Slave*1 Master*2	Adv Slave*1 Master*3
1D	Adv Slave*2	Adv Slave*2 Scanning	Adv Slave*2 Scanning Master*1	Adv Slave*2 Scanning Master*2	Adv Slave*2 Scanning Master*3	Adv Slave*2 Master*4	Adv Slave*2 Master*1	Adv Slave*2 Master*2	Adv Slave*2 Master*3

	2A	2B	2C	2D	2E	2F	2G	2H	2I
1E	Adv Slave*3	Adv Slave*3 Scanning	Adv Slave*3 Scanning Master*1	Adv Slave*3 Scanning Master*2	Adv Slave*3 Scanning Master*3	Adv Slave*3 Master*4	Adv Slave*3 Master*1	Adv Slave*3 Master*2	Adv Slave*3 Master*3
1F	Slave*4	Slave*4 Scanning	Slave*4 Scanning Master*1	Slave*4 Scanning Master*2	Slave*4 Scanning Master*3	Slave*4 Master*4	Slave*4 Master*1	Slave*4 Master*2	Slave*4 Master*3
1G	Slave*1	Slave*1 Scanning	Slave*1 Scanning Master*1	Slave*1 Scanning Master*2	Slave*1 Scanning Master*3	Slave*1 Master*4	Slave*1 Master*1	Slave*1 Master*2	Slave*1 Master*3
1H	Slave*2	Slave*2 Scanning	Slave*2 Scanning Master*1	Slave*2 Scanning Master*2	Slave*2 Scanning Master*3	Slave*2 Master*4	Slave*2 Master*1	Slave*2 Master*2	Slave*2 Master*3
1I	Slave*3	Slave*3 Scanning	Slave*3 Scanning Master*1	Slave*3 Scanning Master*2	Slave*3 Scanning Master*3	Slave*3 Master*4	Slave*3 Master*1	Slave*3 Master*2	Slave*3 Master*3

如果用户的 appMaxMasterNum/appMaxSlaveNum 不是 M1S1 或者 M4S4，请根据以上分析方法去分析。

前面介绍了 supportedMaxMasterNum / supportedMaxSlaveNum 和 appMaxMasterNum / appMaxSlaveNum 的概念，对应上面状态机组合表里面 Master 和 Slave 的个数，再定义两个概念 currentMasterNum 和 currentSlaveNum，表示当前时刻 Link Layer 实际 Master 和 Slave 的数量，比如在上面表中 '1D2E' 组合状态时，currentMasterNum 为 3，currentSlaveNum 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 Master 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 M4S4 状态为例来说明，假设 appMaxMasterNum 和 appMaxSlaveNum 分别是 4 和 4。

各个子状态 (Advertising、Master0 ~ Master3、Slave0 ~ Slave3、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 M4S4 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。

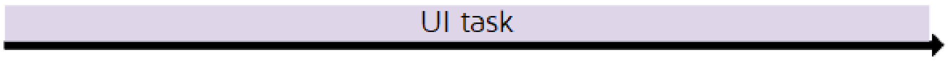


Figure 3.10: Standby 状态时序

3.2.3.2 Scanning only, no Adverting, no Connection 时序

对应 M4S4 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：

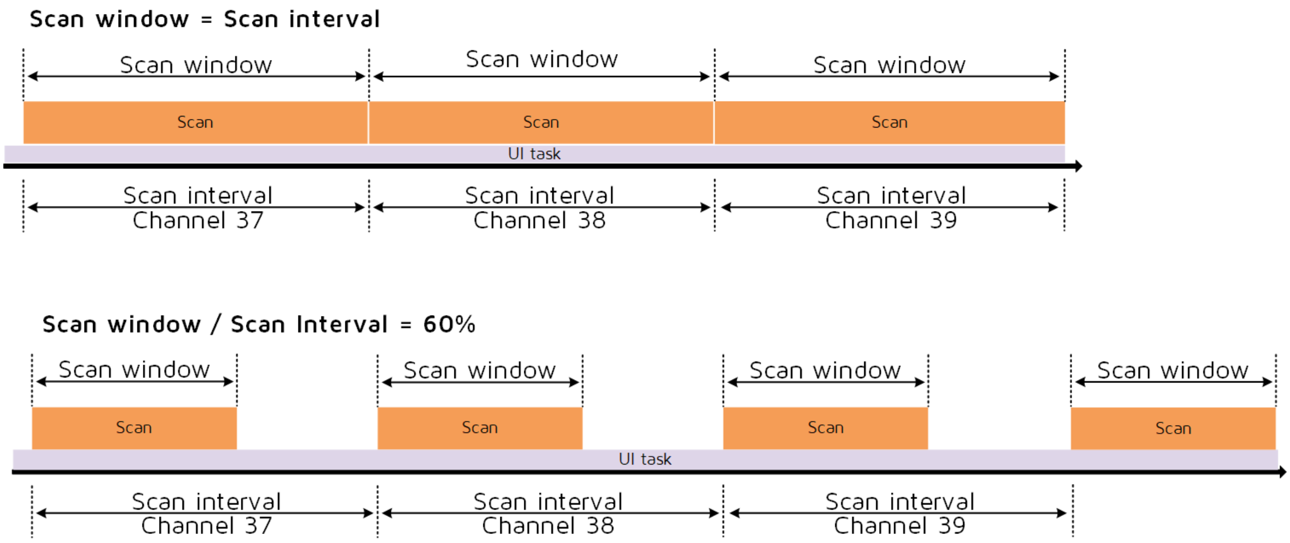


Figure 3.11: Scanning 状态时序分配

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 M4S4 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：

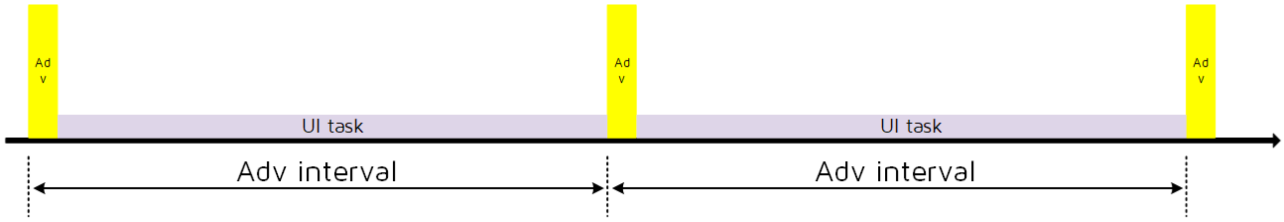


Figure 3.12: Advertisig 状态时序分配

Adv Event 的所有细节参考 Telink 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 M4S4 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：

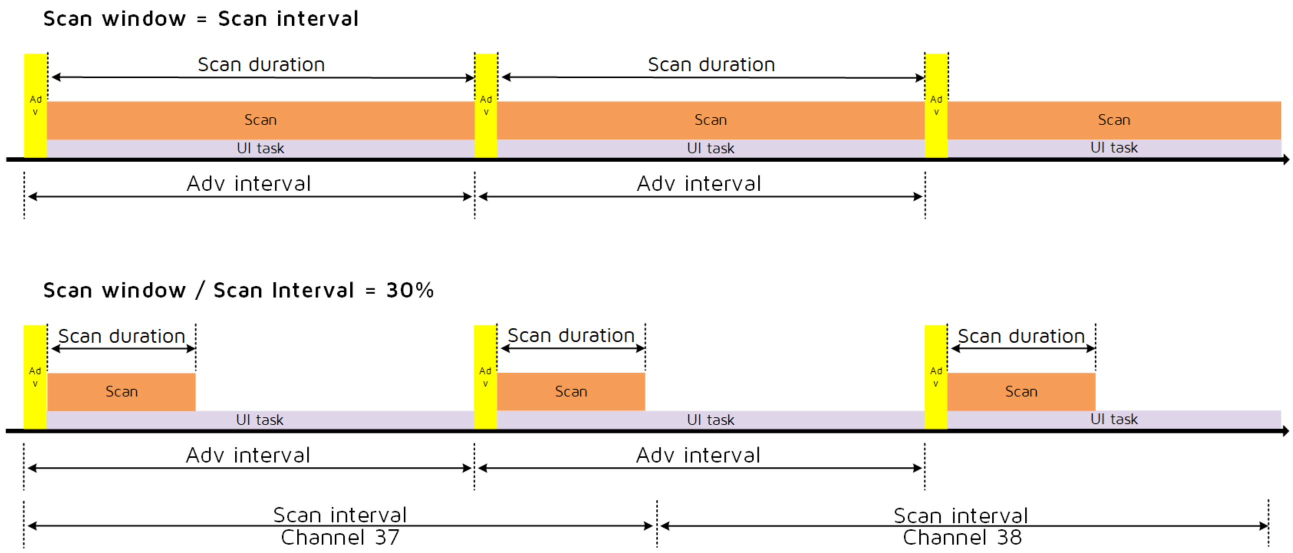


Figure 3.13: Advertising + Scanning 状态时序分配

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：Scan duration/(Adv interval + rand_dly) 尽量等于 Scan window/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 M4S4 在表 3.3 中的 1C2C 状态。

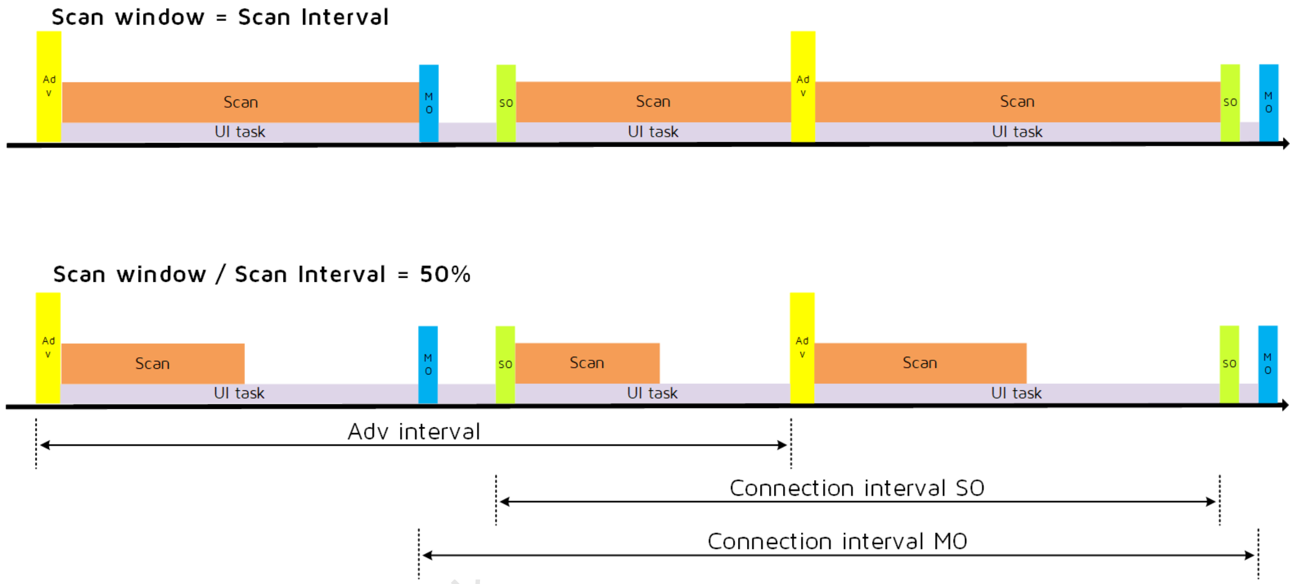


Figure 3.14: M4S4 1C2C 状态时序分配

首先进行连接任务的分配（不管是 Master 还是 Slave），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（stack 限制了最小时间，时间太短不再进行 scan），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 M4S4 在表 3.3 中的 1F2H 状态。

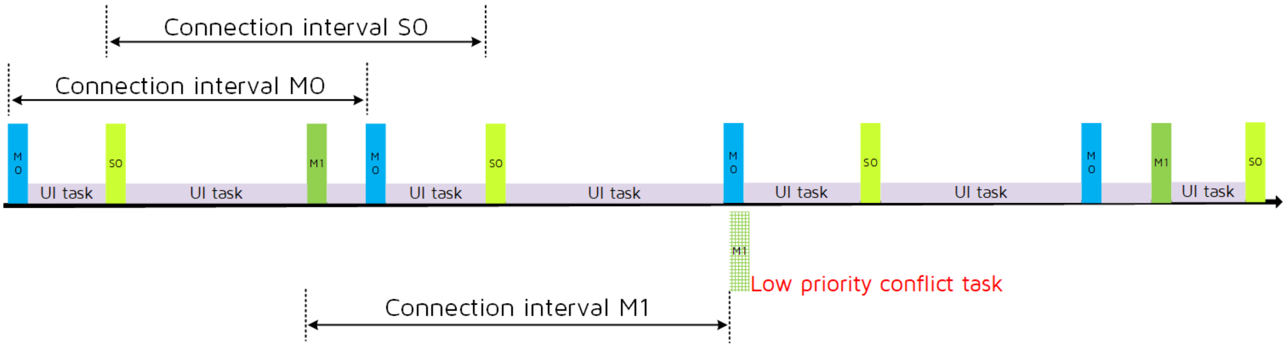


Figure 3.15: M4S4 1F2H 状态时序分配

下图对应 M4S4 在表 3.3 中的 1E2F 状态。

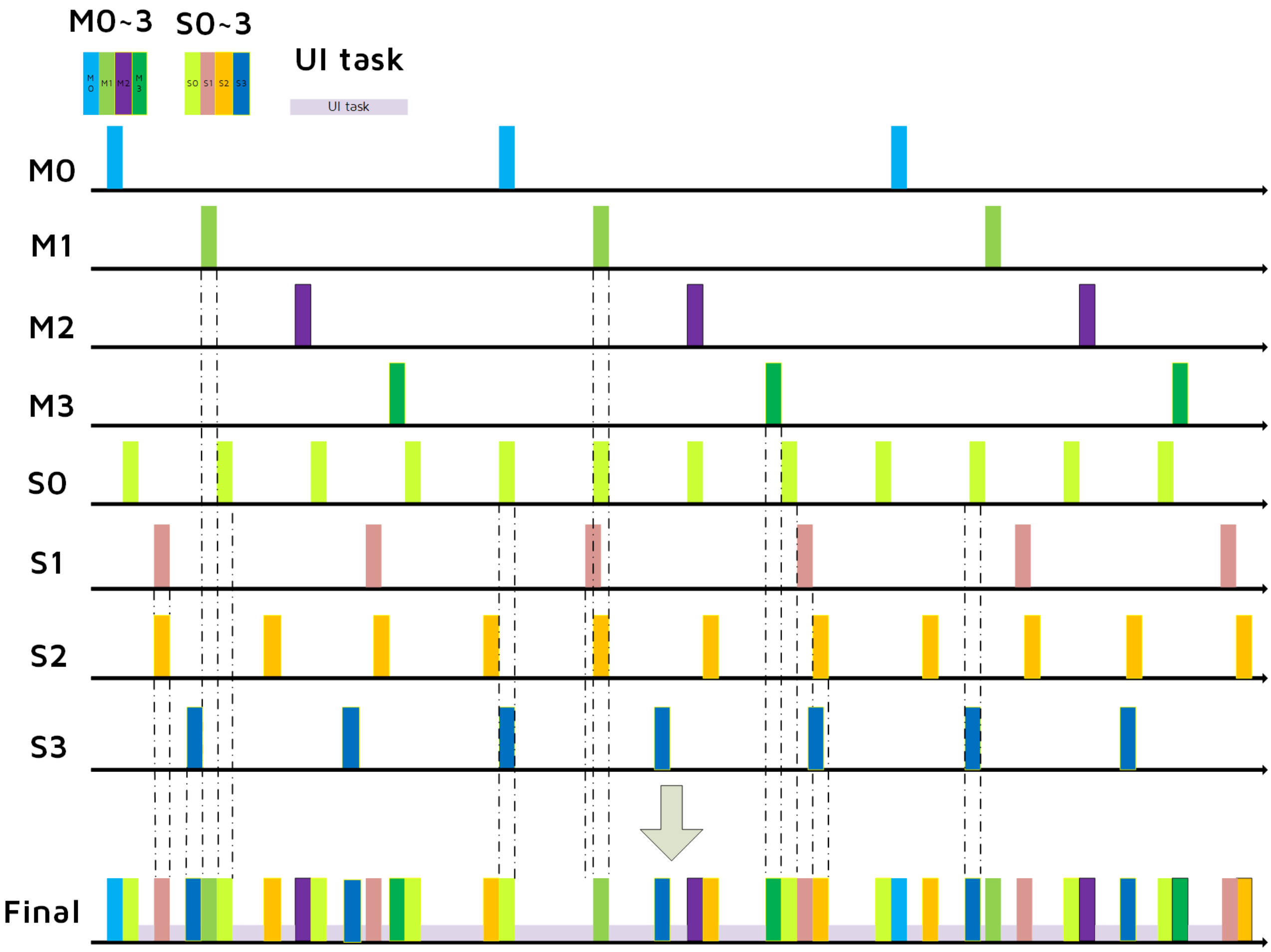


Figure 3.16: M4S4 1E2F 状态时序分配

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_mstTxfifo[ACL_MASTER_TX_FIFO_SIZE * ACL_MASTER_TX_FIFO_NUM * MASTER_MAX_NUM] = {0};
u8  app_acl_slvTxfifo[ACL_SLAVE_TX_FIFO_SIZE * ACL_SLAVE_TX_FIFO_NUM * SLAVE_MAX_NUM] = {0};
```

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

数组 app_acl_slvTxfifo 的大小与三个宏相关：

(1) SLAVE_MAX_NUM 是最大连接数量，即 appMaxSlaveNum。用户可以根据需要在 app_config.h 中修改这个值。

(2) ACL_SLAVE_TX_FIFO_SIZE 是每个 sub_buffer 的 size，与 ACL Slave 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_SLAVE_TX_FIFO_SIZE    CAL_LL_ACL_TX_FIFO_SIZE(ACL_SLAVE_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_SLAVE_MAX_TX_OCTETS 是用户定义的 slaveMaxTxOctets。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_SLAVE_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 Master TX FIFO 和 Slave 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：

- Slave 需要 DLE 功能，而 Master 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 Master tx fifo，2 个 Slave tx fifo，从而减少 ram 的使用，节省 ram 空间：

```
#define MASTER_MAX_NUM                     3
#define SLAVE_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 B85m 为例，其他芯片（如 B91）是同样的原理，类推即可。

(1) M4S4，ACL Master、ACL Slave 都不使用 DLE

假设相关的定义如下：

```
#define MASTER_MAX_NUM                4
#define SLAVE_MAX_NUM                 4
#define ACL_MASTER_MAX_TX_OCTETS      27
#define ACL_SLAVE_MAX_TX_OCTETS       27

#define ACL_MASTER_TX_FIFO_SIZE    CAL_LL_ACL_TX_FIFO_SIZE(ACL_MASTER_MAX_TX_OCTETS)
#define ACL_MASTER_TX_FIFO_NUM     8
#define ACL_SLAVE_TX_FIFO_SIZE     CAL_LL_ACL_TX_FIFO_SIZE(ACL_SLAVE_MAX_TX_OCTETS)
#define ACL_SLAVE_TX_FIFO_NUM      8
```

则 TX FIFO 定义为下面的值：

```
u8  app_acl_mstTxfifo[40 * 8 * 4] = {0};
u8  app_acl_slvTxfifo[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）：

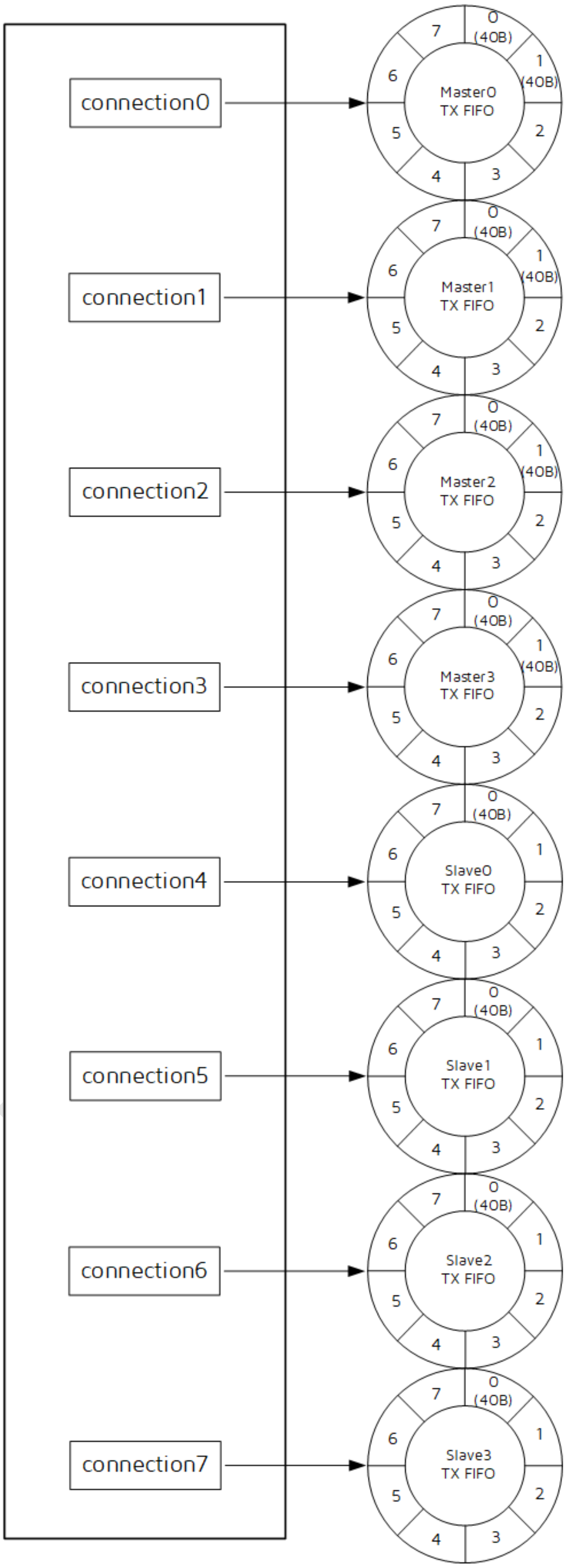


Figure 3.17: TX FIFO 默认设置

(2）M4S4，ACL Master 使用了 DLE 且 MasterMaxTxOctets 为最大值 251，ACL Slave 不使用 DLE。

假设相关的定义如下：

```
#define MASTER_MAX_NUM                  4
#define SLAVE_MAX_NUM                   4
#define ACL_MASTER_MAX_TX_OCTETS        251
#define ACL_SLAVE_MAX_TX_OCTETS         27

#define ACL_MASTER_TX_FIFO_SIZE      CAL_LL_ACL_TX_FIFO_SIZE(ACL_MASTER_MAX_TX_OCTETS)
#define ACL_MASTER_TX_FIFO_NUM       8
#define ACL_SLAVE_TX_FIFO_SIZE       CAL_LL_ACL_TX_FIFO_SIZE(ACL_SLAVE_MAX_TX_OCTETS)
#define ACL_SLAVE_TX_FIFO_NUM        8
```

则 TX FIFO 定义为下面的值：

```
u8  app_acl_mstTxfifo[264 * 8 * 4] = {0};
u8  app_acl_slvTxfifo[40 * 8 * 4] = {0};
```

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

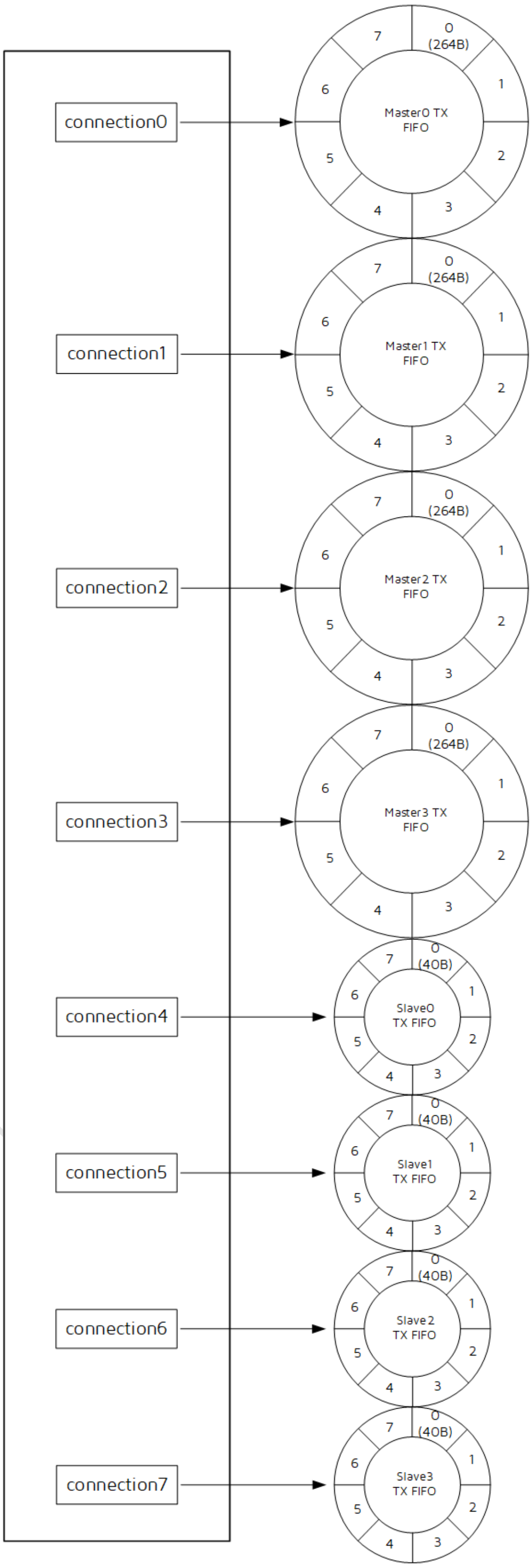


Figure 3.18: Master 使用 DLE，Slave 不使用 DLE 的 buffer 情况

(3) M3S2，ACL master、ACL Slave 都不使用 DLE

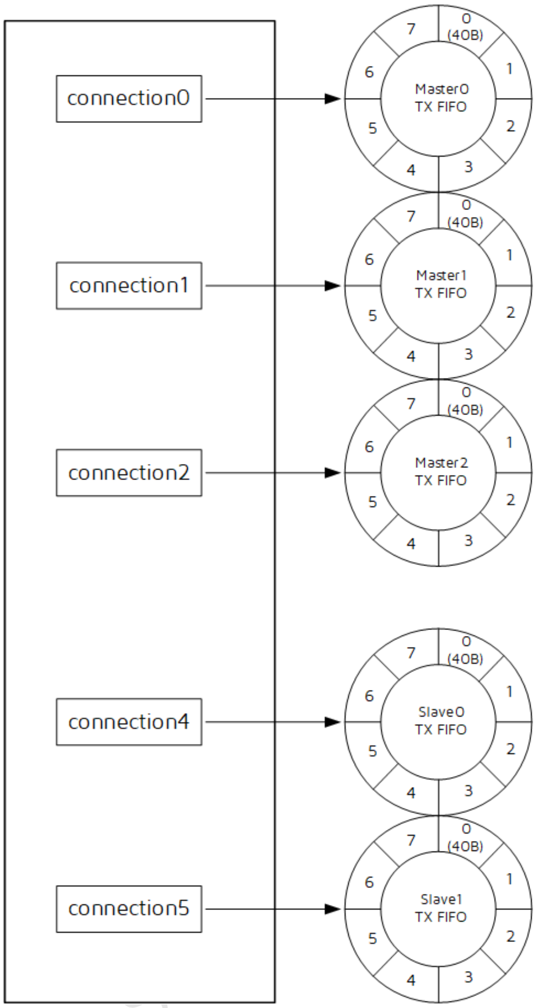


Figure 3.19: 3 个 Master，2 个 Slave 的 buffer 情况

3.2.4.2 ACL RX FIFO 定义及设置

ACL RX FIFO 的定义如下：

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

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

数组 app_acl_rxfifo 的大小与二个宏相关：

(1) ACL_RX_FIFO_SIZE 是 buffer size，与 ACL Slave 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 slaveMaxRxOctets。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 M4S4 ACL RX FIFO 的定义如下：

```
#define MASTER_MAX_NUM                          4
#define SLAVE_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：

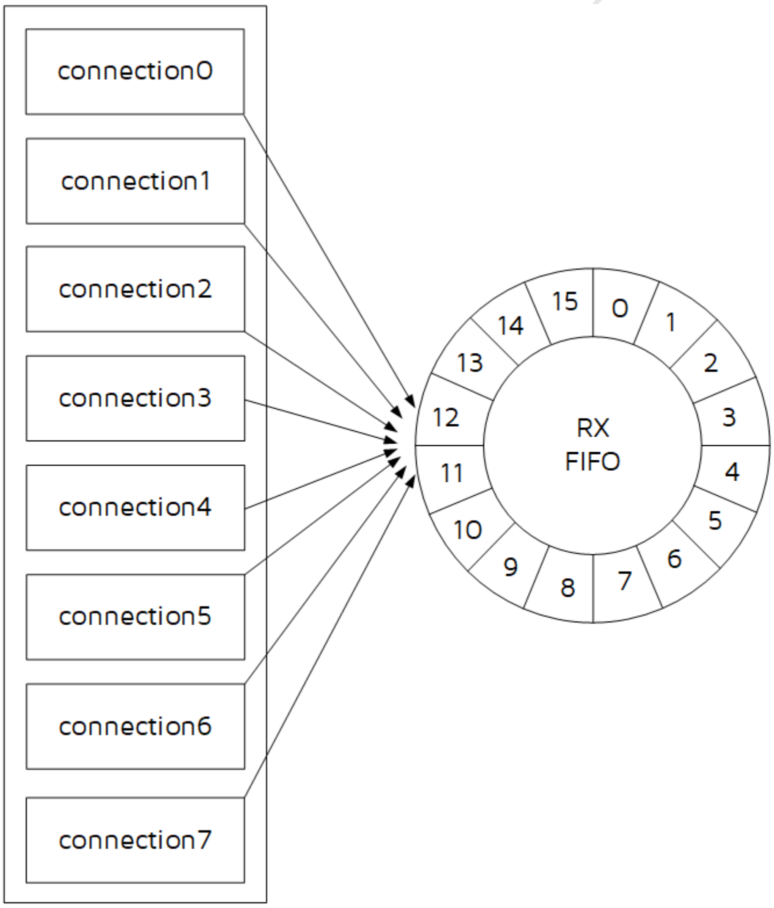


Figure 3.20: ACL RX buffer

3.2.4.3 RX overflow 分析

参考 Telink 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）。

Telink BLE Multiple Connection 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 V5.3, 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：

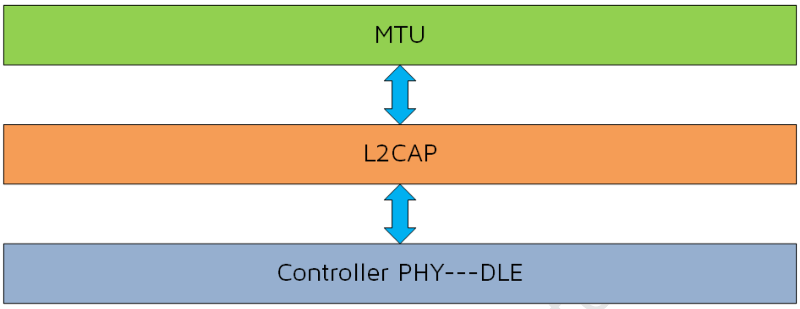


Figure 3.21: MTU 与 DLE 概念

下图是 MTU 和 DLE 所包含的内容：

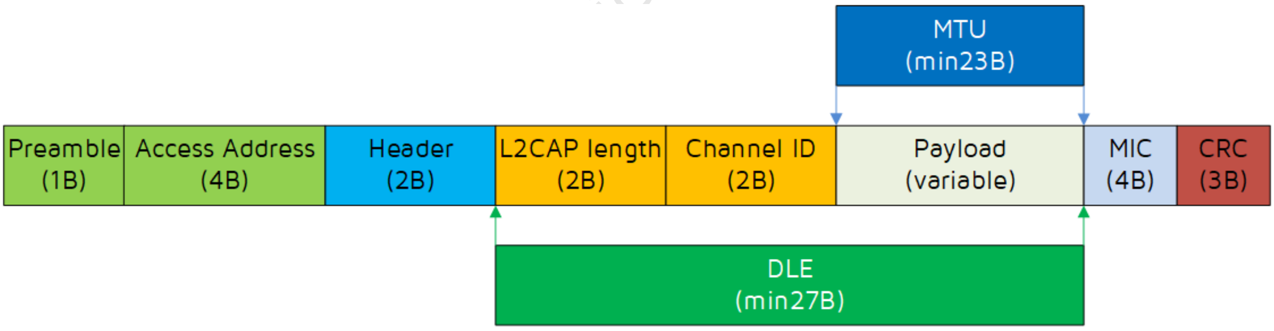


Figure 3.22: MTU 与 DLE 包含内容

- 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 Master 和 Slave 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 b85m_feature/b91_feature 工程中的 feature_dle。

在 vendor/b85m_feature/feature_config.h 或 vendor/b91_feature/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 ATT_MTU_MASTER_RX_MAX_SIZE 和 ATT_MTU_SLAVE_RX_MAX_SIZE 即可，它们分别代表 ACL Master 和 ACL Slave 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。

```
#define ATT_MTU_MASTER_RX_MAX_SIZE    23
#define  MTU_M_BUFF_SIZE_MAX          CAL_MTU_BUFF_SIZE(ATT_MTU_MASTER_RX_MAX_SIZE)

#define ATT_MTU_SLAVE_RX_MAX_SIZE     23
#define MTU_S_BUFF_SIZE_MAX           CAL_MTU_BUFF_SIZE(ATT_MTU_SLAVE_RX_MAX_SIZE)
```

然后初始化里面调用以下 API 分别设置 ACL Master 和 ACL Slave 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_setMasterRxMTUSize(ATT_MTU_MASTER_RX_MAX_SIZE);
blc_att_setSlaveRxMTUSize(ATT_MTU_SLAVE_RX_MAX_SIZE);
```

MTU size exchange 的实现，请参考本文档"ATT & GATT"部分的详细说明—3.4.2.4.7 Exchange MTU Request, Exchange MTU Response，也可以参考 b85m_feature/b91_feature 工程中 feature_dle 的写法。

(2) 设置 connMaxTxOctets 和 connMaxRxOctets

其次需要设置 Master 和 Slave 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）。

```
#define ACL_CONN_MAX_RX_OCTETS          27
#define ACL_MASTER_MAX_TX_OCTETS        27
#define ACL_SLAVE_MAX_TX_OCTETS         27
```

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

```
ble_sts_t   blc_ll_setAclConnMaxOctetsNumber(u8 maxRxOct, u8 maxTxOct_master, u8 maxTxOct_slave)
```

(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_setMasterRxMTUSize(23) 和 blc_att_setSlaveRxMTUSize(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_setMasterRxMTUSize 和 blc_att_setSlaveRxMTUSize），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_sendDateLengthExtendReq () 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 Coded PHY/2M PHY Demo 介绍

Telink BLE Multiple Connection SDK 中，为节省 Sram，Coded PHY/2M PHY 默认是关闭的，用户如果选择使用此 Feature，可以手动打开。

- local device 参考 b85m_feature/feature_2M_coded_phy 或 b91_feature/ feature_2M_coded_phy。

在 feature_config.h 中定义宏：

```
#define FEATURE_TEST_MODE                        TEST_2M_CODED_PHY_CONNECTION
```

- peer device 可以选择使用手机或其他标准的 BLE Master 设备；也可以使用任何一个支持 Coded PHY/2M PHY 的 Telink BLE SDK，编译出 BLE Master 设备运行，比如使用 Telink BLE Single Connection SDK 编译的 xxx_kma_master_dongle 或 Telink BLE Multiple Connection SDK 编译的 xxx_master_dongle。

3.2.6.2 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 V5.3, 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 Master 设备会触发 PHY update，新的 PHY 很快会生效。

connHandle： Master/Slave 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 V5.3, Vol 6, Part B, 4.5.8.3 Channel Selection algorithm #2。

Telink BLE Multiple Connection 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 Master/Slave device 都同时支持 CSA #2(ChSel 字段都设置为 1)，CSA #2 才能用于连接。

Demo 参考 b85m_feature/feature_misc，在 feature_config.h 中定义宏：

```
#define FEATURE_TEST_MODE        TEST_MISC_FUNC
```

在对应工程的 app.c 中定义：

```
#define MISC_FUNC_TEST_MODE      TEST_CSA2
```

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 V5.3, Vol 4, Part E: Host Controller Interface Functional Specification。

Telink BLE Multiple Connection 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_slave.h、acl_master.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。如果不需要 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 bls_ll_setAdvData

详情请参照 Bluetooth Core Specification V5.3, Vol 4, Part E, 7.8.7 LE Set Advertising Data command。

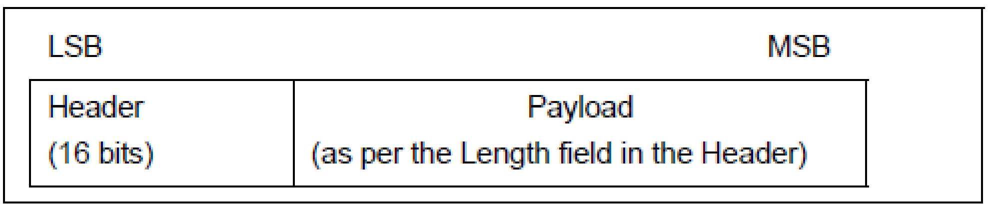


Figure 3.23: 协议栈广播包格式

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 bls_ll_setScanRspData

详情请参照 Bluetooth Core Specification V5.3, 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   blc_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 bls_ll_setAdvParam

详情请参照 Bluetooth Core Specification V5.3, Vol 4, Part E, 7.8.5 LE Set Advertising Parameters command。

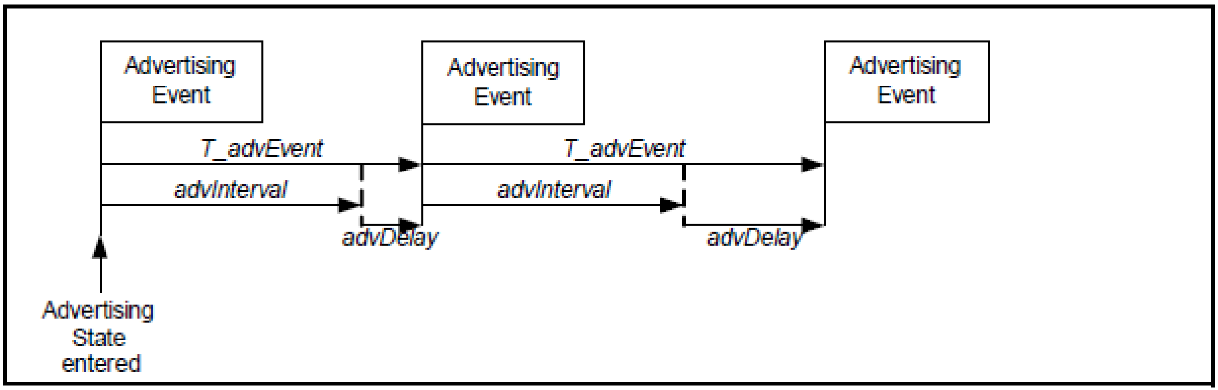


Figure 3.24: 协议栈里 Advertising Event

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，Slave 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 Undi-rected Event	ADV_IND	YES	YES
Connectable Directed Event	ADV_DIRECT_IND	NO	YES*
Non-connectable Undirected Event	ADV_NONCONN_IND	NO	NO
Scannable Undi-rected 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 Undireted 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 bls_ll_setAdvEnable

详情请参照 Bluetooth Core Specification V5.3, 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	appMaxSlaveNum 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 V5.3, 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 V5.3, 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 V5.3, 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_inetrval/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_adr_type

指定建立连接的 Master 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 Master role 的连接参数，同时这些参数也会通过 connection request 发给 Slave，Slave 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 appMaxMasterNum > 1，conn_min/ conn_max 参数无效，SDK 中 Master role conn interval 默认固定为 25(实际 interval 为 31.25 ms = 25 x1.25 ms)，这种情况下可以在建立连接前调用 blc_ll_setAclMasterConnectionInterval 更改设置；如果 appMaxMasterNum 为 1，SDK 中 Master 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

Telink BLE Multiple Connection 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_ LIMITED_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。

Telink BLE Multiple Connection 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_setAclMasterConnectionInterval

```
ble_sts_t   blc_ll_setAclMasterConnectionInterval(u16 conn_interval);
```

返回值为 BLE_SUCCESS，conn interval 单位为 1.25 ms。

这个 API 设置 master base connection interval 基准，通过这个基准可以将多个 master 的时序错开，保证多 master 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 master 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 Master/Slave 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 V5.3, 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

ble_sts_t	Value	ERR Reason
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

Telink BLE Multiple Connection SDK 提供了 BLE RF packet 能量设定的 API，不同芯片函数原型不同：

b85m 的 API 原型如下：

```
void rf_set_power_level_index (RF_PowerTypeDef level);
```

B85m 的 level 值设置参考 drivers/8258(8278)/rf_drv.h 中定义的枚举变量 RF_PowerTypeDef。

b91 的 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。

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 Telink BLE Multiple Connection SDK 中 whitelist 最多存储 4 个设备：

```
#define    MAX_WHITE_LIST_SIZE              4
```

Whitelist 相关 API 如下：

```
ble_sts_t ll_whiteList_reset(void);
```

Reset whitelist，返回值为 BLE_SUCCESS。

```
ble_sts_t ll_whiteList_add(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 ll_whiteList_delete(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 Telink BLE Multiple Connection SDK 中 Resolvinglist 最多存储 2 个设备：

```
#define    MAX_WHITE_IRK_LIST_SIZE          2
```

Resolvinglist 相关 API 如下：

```
ble_sts_t  ll_resolvingList_reset(void);
```

Reset Resolvinglist。返回值 BLE_SUCCESS。

```
ble_sts_t  ll_resolvingList_setAddrResolutionEnable(u8 resolutionEn);
```

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  ll_resolvingList_add(u8 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  ll_resolvingList_delete(u8 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 8258_multi_conn_feature_test 工程(feature_whitelist.c)。

在 vendor/b85m_feature/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 V5.3, 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

Telink BLE Multiple Connection 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。

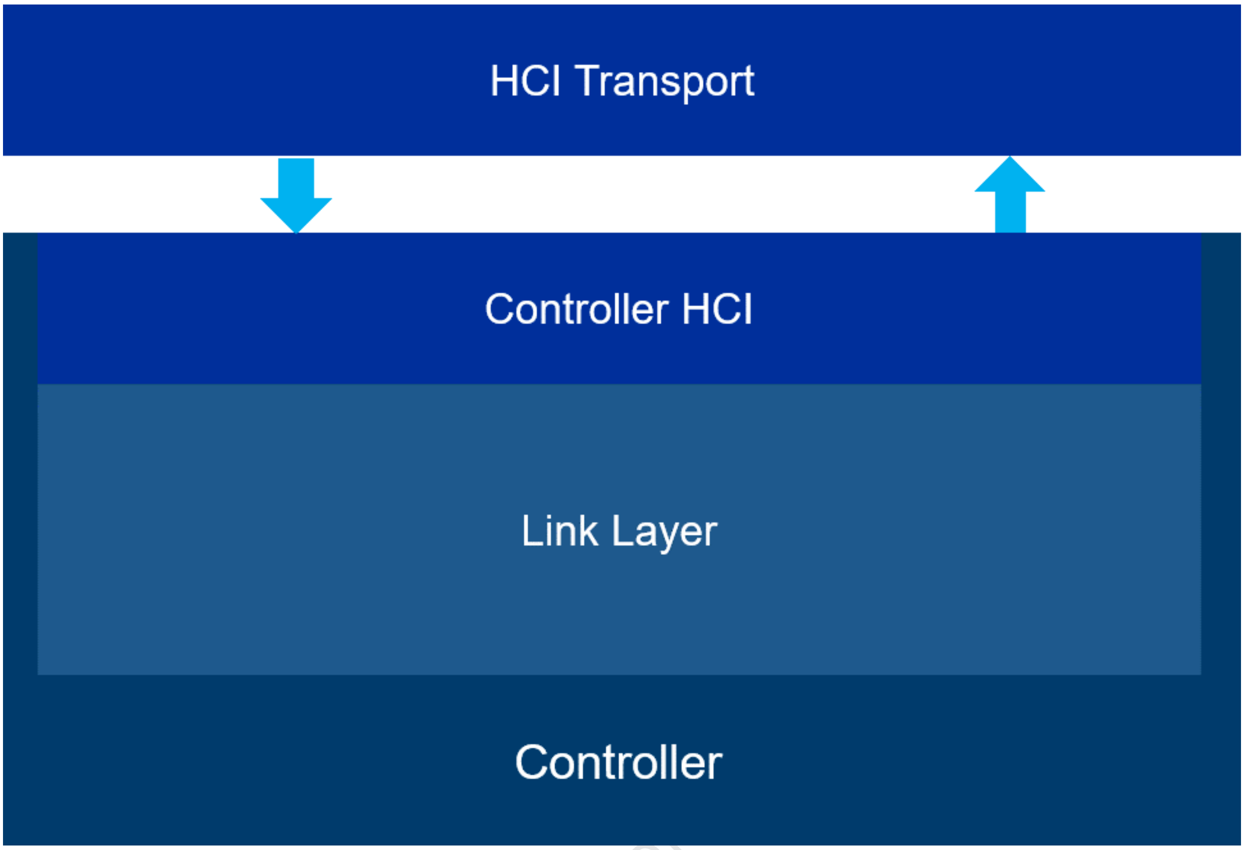


Figure 3.27: Telink HCI 软件架构

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。HCI Transport 支持多种硬件接口，如：USB，UART，SDIO 等，其中常用的是 UART 接口，Telink BLE Multiple Connection SDK 目前只提供了 UART 接口的 HCI Transport。Telink BLE HCI transport 的软件实现可以查看 SDK 中的 vendor/common/hci_transport 文件夹。

HCI UART Transport 支持两种协议，H4 和 H5，这两种协议 Telink BLE Multiple Connection SDK 都支持，且以源码的形式开放使用。Telink SDK HCI Transport 的软件架构如下图所示。

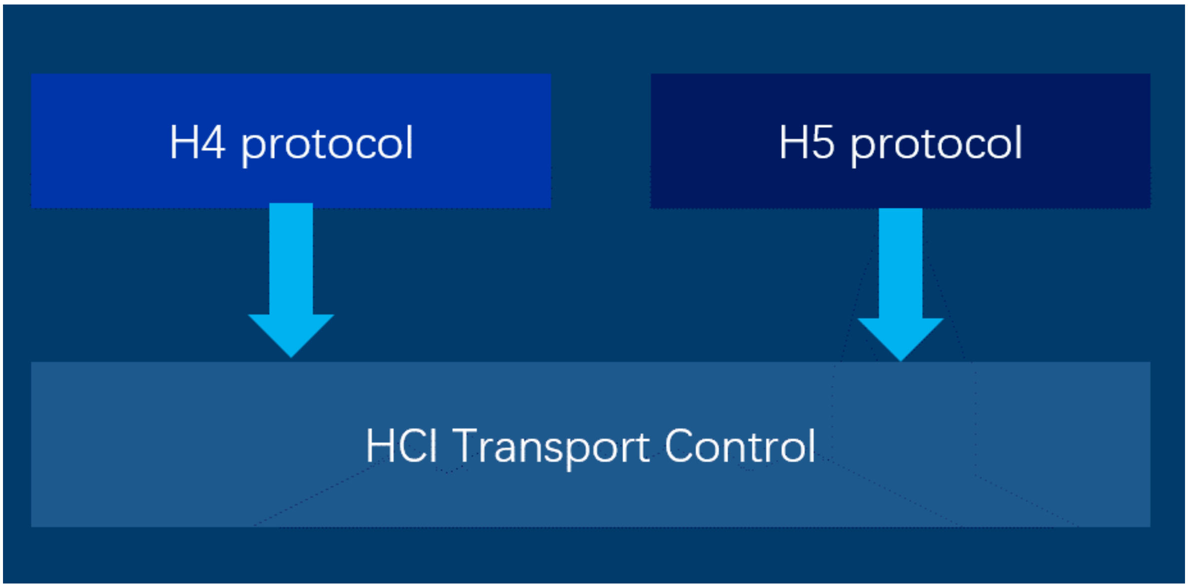


Figure 3.28: Telink HCI Transport 软件架构

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。

LSB	MSB
HCI Type Indicator (1 octets)	Payload (N octets)

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 V5.3, 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。

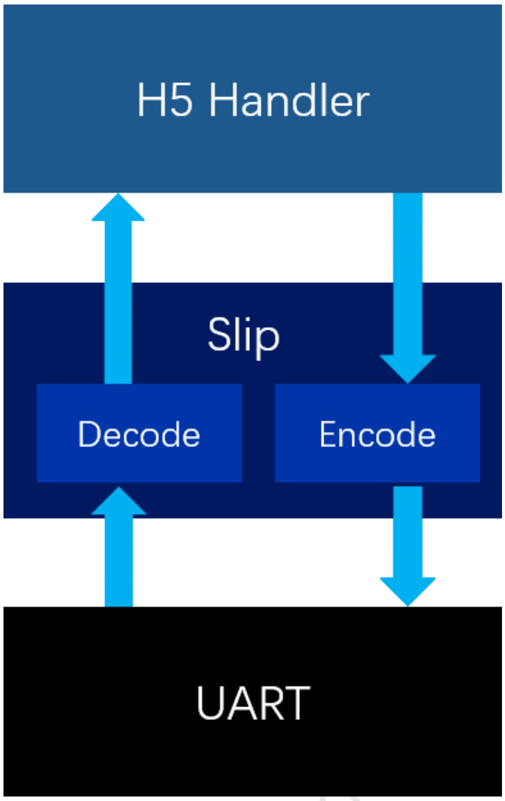


Figure 3.30: H5 软件架构

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 的转义序列都开始于 DB。Slip 的转义序列表如下图所示。

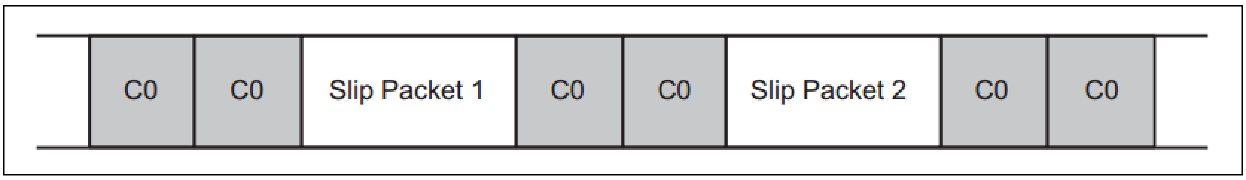


Figure 3.31: HCI Slip 包

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

Telink BLE Multiple Connection 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 C0 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 C0 和 DB，这样 Slip 就被解码了，接下来就是解析 H5 PDU。

Telink BLE Multiple Connection 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。

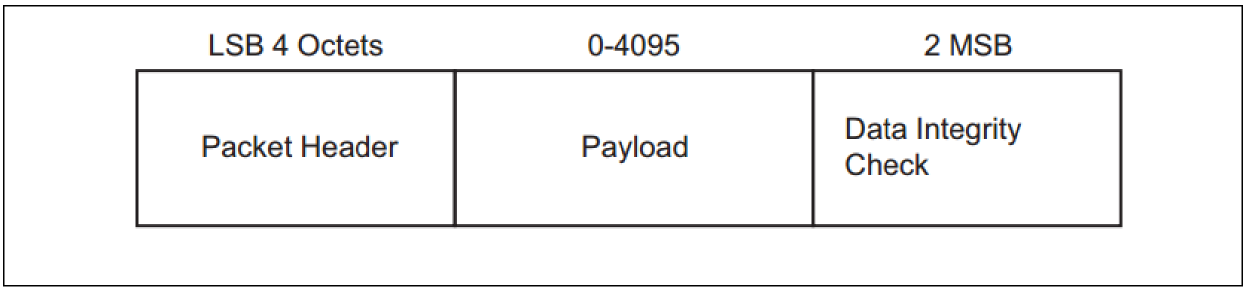


Figure 3.33: H5 PDU

H5 PDU Header 构成如下：

LSB 3 bits	3	1	1	4	12	8 MSB
Sequence Number	Acknowledgment Number	Data Integrity Check Present	Reliable Packet	Packet Type	Payload Length	Header Checksum

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。

Telink BLE Multiple Connection 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。

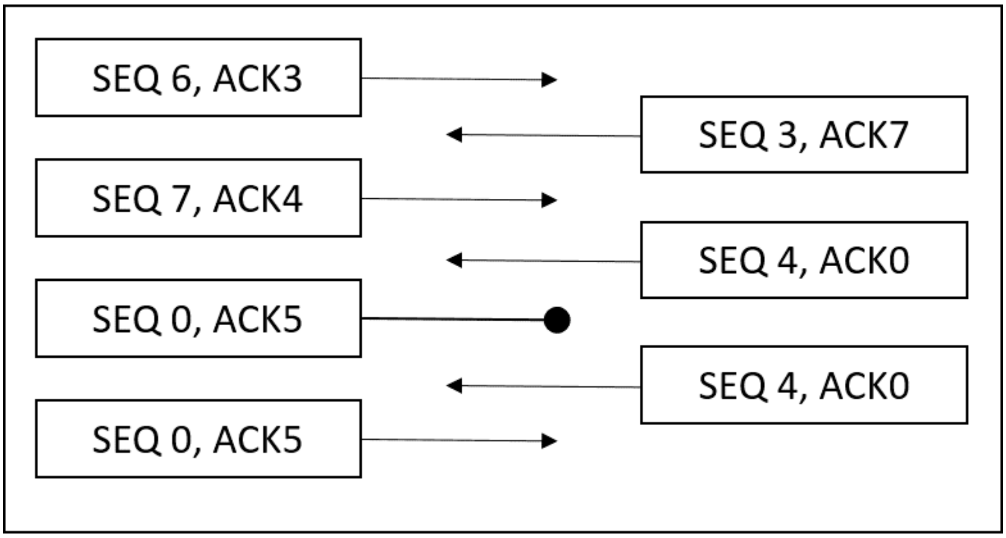


Figure 3.36: H5 Data flow

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。Telink BLE Multiple Connection 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 UART transport Flow Control. */
#define HCI_TR_FLOW_CTRL_EN  0
#if HCI_TR_FLOW_CTRL_EN
/*** RTS/CTS Pin ***/
#define HCI_TR_RTS_PIN       UART_RTS_PB3
#define HCI_TR_CTS_PIN       UART_CTS_PA3
#endif
```

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。HCI_TR_FLOW_CTRL_EN 用于使能 UART 硬件流控。

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，HCI 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 HCI Transport 功能之前需要通过该函数初始化 HCI 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 HCI

Controller HCI 接口实现了 HCI 协议包的解析和处理以及产生 HCI Event。HCI 具体内容参看 Bluetooth Core Specification V5.3 Vol4。本节将讲解几个重要的 API。

Controller HCI 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

blc_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

blc_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 工程介绍

Telink BLE Multiple Connection SDK 专门提供了 Controller 工程方便用户对接其他 Host 使用。本文以 Telink B85m BLE Multiple Connection SDK 中的 Controller 工程为例来介绍，其工程文件结构如下图所示：

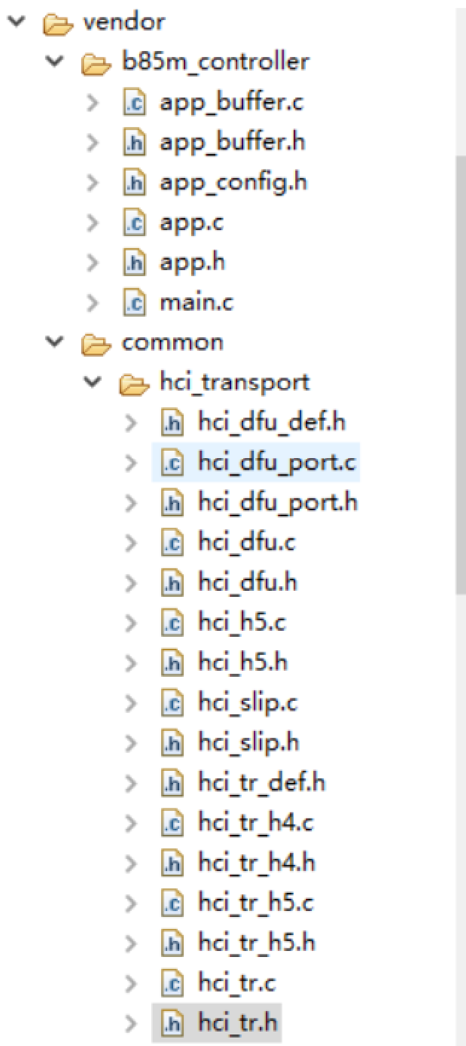


Figure 3.37: Controller 工程文件结构

Controller 工程中的除了 HCI 部分外，其他部分的 code 与其他 demo 工程一样。Demo 工程前面章节有介绍，本节重点对 HCI 相关的 code 做说明。

(1) main.c

main.c 提供了程序入口和全局中断函数入口。程序入口 main 函数实现了硬件和软件模块的初始化以及 while(1) 循环；全局中断函数调用了 ble stack 的中断处理函数和 HCI Transport 的中断处理函数，如下：

```
void irq_handler(void)
{
    blc_sdk_irq_handler ();

    HCI_TransportIRQHandler();
}
```

(2) app.c

Controller HCI 接口和 HCI Transport 的初始化程序在 app.c 文件的 user_init_normal() 函数中被调用，HCI Transport 任务处理在 main_loop() 函数中调用，如下：

```
void user_init_normal(void)
{
     ......

    /* HCI RX FIFO */
    blc_ll_initHciRxFifo(app_bltHci_rxfifo, HCI_RX_FIFO_SIZE, HCI_RX_FIFO_NUM);
    /* HCI TX FIFO */
    blc_ll_initHciTxFifo(app_bltHci_txfifo, HCI_TX_FIFO_SIZE, HCI_TX_FIFO_NUM);
    /* HCI RX ACL FIFO */
    blc_ll_initHciAclDataFifo(app_hci_aclDataFifo,HCI_ACL_DATA_FIFO_SIZE,
 ↪  HCI_ACL_DATA_FIFO_NUM);

    /* HCI Data && Event call-back */
    blc_hci_registerControllerDataHandler (blc_hci_sendAC
    blc_hci_registerControllerEventHandler(blc_hci_send_data);

    //bluetooth event
    blc_hci_setEventMask_cmd (HCI_EVT_MASK_DISCONNECTION_COMPLETE);
    //bluetooth low energy(LE) event, all enable
    blc_hci_le_setEventMask_cmd( 0xFFFFFFFF );
    blc_hci_le_setEventMask_2_cmd( 0x7FFFFFFF );

     ......
     /* HCI Transport initialization */
    HCI_TransportInit();
    blc_ll_register_user_irq_handler_cb(HCI_Tr_IRQHandler);
}

void main_loop(void)
{
    HCI_TransportPoll();

    ......
}
```

(3) app_buffer.c 和 app_buffer.h

这两个文件主要实现了 SDK 所有使用的 buffer 的定义。Controller HCI 部分使用的 buffer 有 3 个，分别是 app_bltHci_rxfifo[]、app_bltHci_txfifo[] 和 app_hci_aclDataFifo[]，这些 buffer 的 size 定义由如下宏控制。

```
#define HCI_TX_FIFO_SIZE                ((2+1+4 + 130 + 4)& ~3)
#define HCI_TX_FIFO_NUM             16

#define HCI_RX_FIFO_SIZE                ((1+4+ 130 +16) & ~15)
#define HCI_RX_FIFO_NUM             4

#define LE_ACL_DATA_PACKET_LENGTH      (28)
```

```
#define HCI_ACL_DATA_FIFO_SIZE      CALCULATE_HCI_ACL_DATA_FIFO_SIZE(LE_ACL_DATA_PACKET_LENGTH)
#define HCI_ACL_DATA_FIFO_NUM         8
```

HCI_TX_FIFO_SIZE：用于设置 HCI Tx Buffer 的 size。应该设置为最大可能发送的 HCI Event payload 和 HCI ACL payload 中的最大者再加上 4 Bytes 的 HCI ACL header、1 Byte 的 HCI type 以及 2Bytes 的 telink 内部使用的内存，然后 4 字节对齐。例如：最大 HCI Event payload = 65B，最大 HCI ACL payload=27B，那么 HCI_TX_FIFO_SIZE = align4(2B(telink) + 1B(HCI Type) + 4B(HCI ACL Header) + MAX(27, 65)) = 72B。

HCI_TX_FIFO_NUM：用于设置 HCI Tx buffer 个数，必须是 2 的指数幂，如 2,4,8...

HCI_RX_FIFO_SIZE：用于设置 HCI Rx buffer 的 size，应该设置为最大可能收到的 HCI Cmd payload 和 HCI ACLpaylaod 中的最大者再加上 4Bytes 的 HCI ACL header 和 1Byte 的 HCI Type，然后 16 字节对齐。例如：最大 HCI Cmd payload = 65B，最大 HCI ACL payload=27B，那么 HCI_RX_FIFO_SIZE = align16(1B(HCI Type) + 4B(HCI ACL Header) + MAX(27, 65)) = 80B。

HCI_RX_FIFO_NUM：用于设置 HCI Rx buffer 个数，必须是 2 的指数幂，如 2,4,8...

LE_ACL_DATA_PACKET_LENGTH：用于设置最大 HCI ACL payload，该值不能超过 252.

HCI_ACL_DATA_FIFO_SIZE：用于设置 ACL Data Buffer Size，用户直接调用宏 CALCULATE_HCI_ACL_DATA_FIFO_SIZE(LE_ACL_DATA_PACKET_LENGTH) 自动计算即可

HCI_ACL_DATA_FIFO_NUM：用于设置 HCI ACL DATA buffer 个数，必须是 2 的指数幂，如 2,4,8...

3.3.3 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。

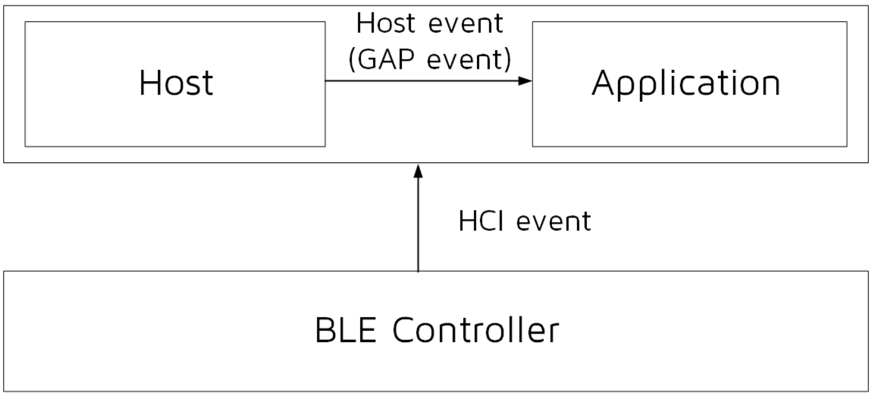


Figure 3.38: BLE SDK Event 架构

注意：

> 在 Telink 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 Telink BLE Multiple Connection 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。

3.3.3.1 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。

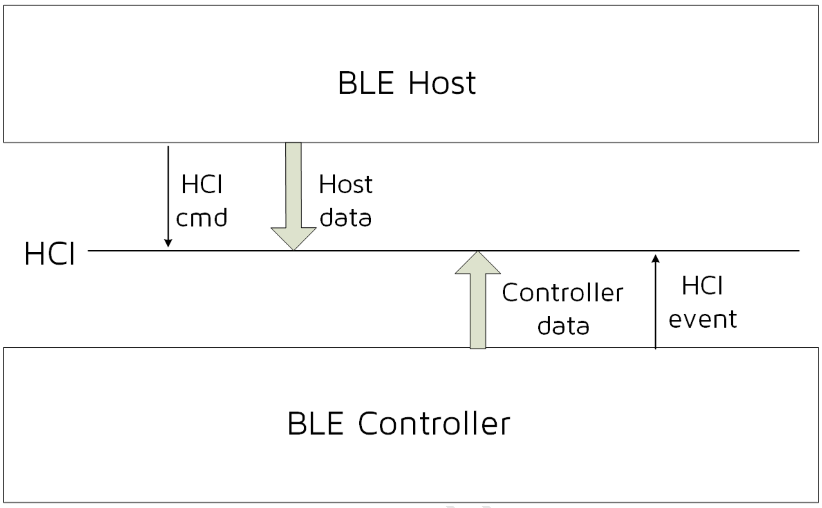


Figure 3.39: Host + Controller 架构

Controller HCI event 的定义，详情请参考 Bluetooth Core Specification V5.3, 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。Telink BLE Multiple Connection SDK 也将 Controller HCI event 分为两类：HCI event 和 HCI LE event。由于 Telink BLE Multiple Connection 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);
```

3.3.3.2 常用 Controller HCI event

Telink BLE Multiple Connection 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 V5.3, 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          0x0000000010
```

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。

3.3.3.3 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 V5.3, 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         slaveLatency;
    u16         supervisionTimeout;
    u8          masterClkAccuracy;
} hci_le_connectionCompleteEvt_t;
```

(2) HCI_SUB_EVT_LE_ADVERTISING_REPORT

详情请参考 Bluetooth Core Specification V5.3, 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]						rssi[1..i]

Figure 3.40: ADVERTISING_REPORT 事件包格式

注意：

> Telink BLE Multiple Connection 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 V5.3, 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 V5.3, 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。

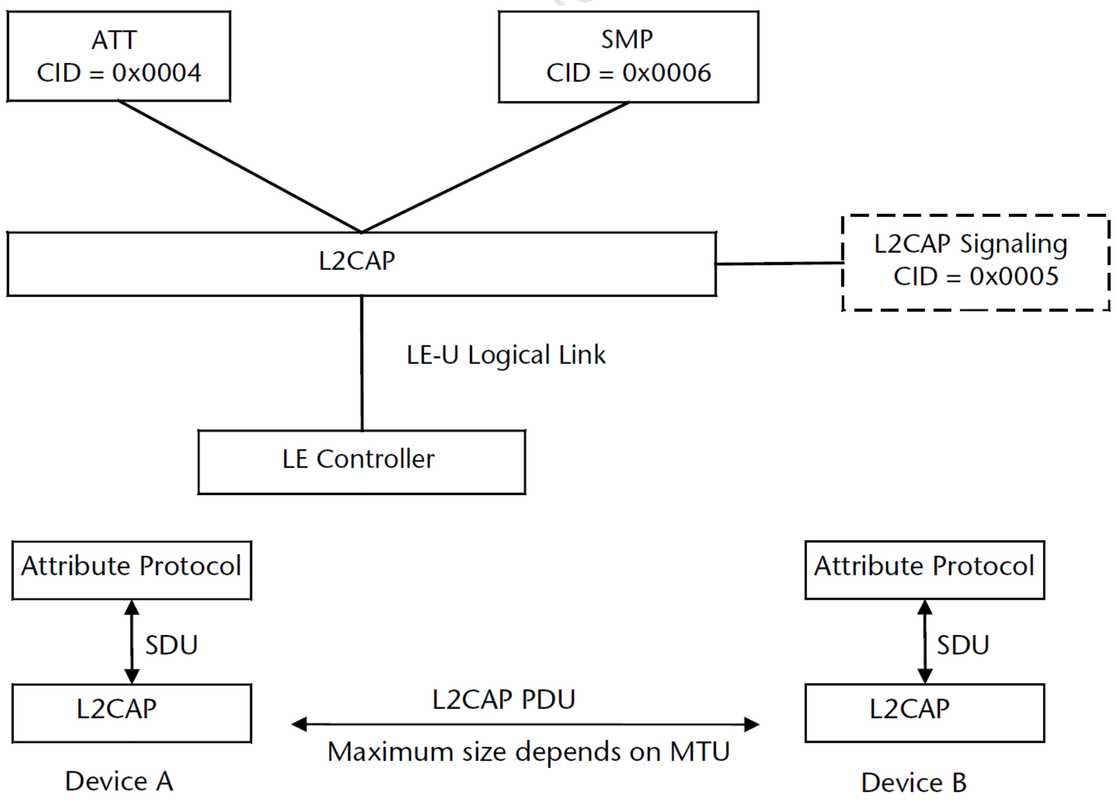


Figure 3.41: BLE L2CAP 结构以及 ATT 组包模型

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

Telink BLE Multiple Connection 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(void *p);
```

L2CAP 层处理 Controller 数据的函数为：

```
int         blt_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) Slave 请求更新连接参数

在 BLE 协议栈中，Slave 通过 L2CAP 层 CONNECTION PARAMETER UPDATE REQUEST 命令向 Master 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 V5.3, Vol 3, Part A, 4.20 L2CAP_CONNECTION_PARAMETER_UPDATE_REQ (code 0x12)。

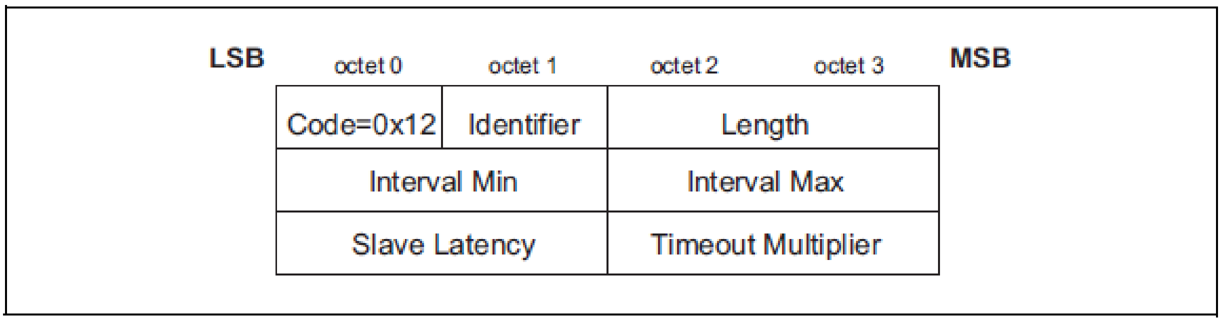


Figure 4.22: Connection Parameters Update Request Packet

Figure 3.42: BLE 协议栈中 Connection Para update Req 格式

Telink BLE Multiple Connection SDK 提供了 Slave 主动申请更新连接参数的 API，用来向 Master 发送 CONNECTION PARAMETER UPDATE REQUEST 命令。

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

该 API 仅限 Slave 使用。min_interval 和 max_interval 的单位为 1.25 ms，timeout 的单位为 10 ms。

Telink BLE Multiple Connection SDK 提供了 Slave 设置发送更新连接参数请求时间的 API：

```c
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。

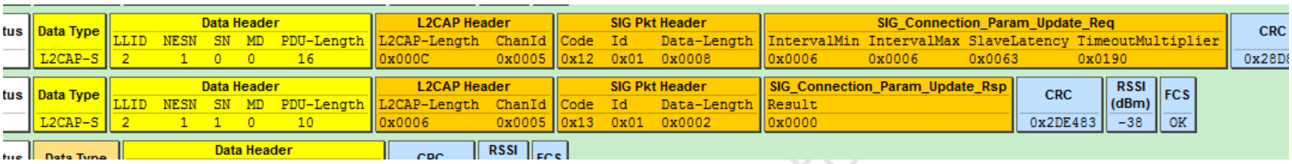


Figure 3.43: 抓包显示 conn para update reqeust 和 response

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 Slave 申请的连接参数请求是否被接受，如上图所示，Master 接受了 Slave 的 Connection_Param_Update_Req 参数。

其中 app_host_event_callback 函数参考如下：

```c
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 Master Host has accepted the connection parameters
            }
            else if( p->result == CONN_PARAM_UPDATE_REJECT ){
                //the LE Master Host has rejected the connection parameter
            }
        }
        Break;
        ......
    }
  return 0;
}
```

(2) Master 回应更新申请

peer Slave 申请新的连接参数后，Master 收到该命令，回 CONNECTION PARAMETER UPDATE RESPONSE 命令，详情请参照 Bluetooth Core Specification V5.3, 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 Master 的做法有关，各家标准并不统一。

Telink BLE Multiple Connection SDK 中，不管是否接受 Slave 的参数申请，都使用下面 API 对该申请进行答复：

```
void blc_l2cap_SendConnParamUpdateResponse(connHandle, req->id, connParaRsp);
```

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

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

Telink BLE Multiple Connection SDK 中，Master 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 Master 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 blc_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 blc_l2cap_SendConnParamUpdateResponse，并且第三个参数设置为 CONN_PARAM_UPDATE_ACCEPT，然后再调用 blm_l2cap_processConnParamUpdatePending 进入连接参数更新流程。

```
void blm_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

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

Master 会发送 link layer 层的 LL_CONNECTION_UPDATE_REQ 命令，如下图所示。

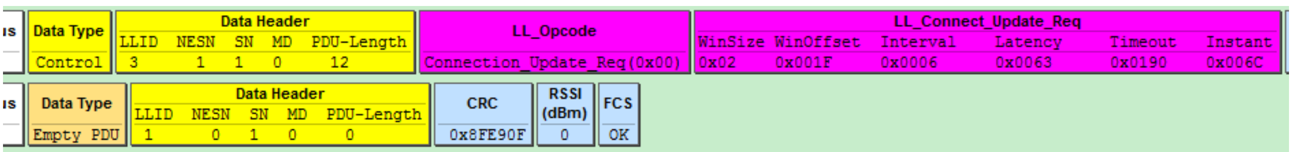


Figure 3.44: 抓包显示 ll conn update req

Slave 收到更新请求后，更新连接参数。Master 和 Slave 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。Telink BLE Multiple Connection SDK 中，Slave 设备是 Server，Android、iOS 或 Master 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。

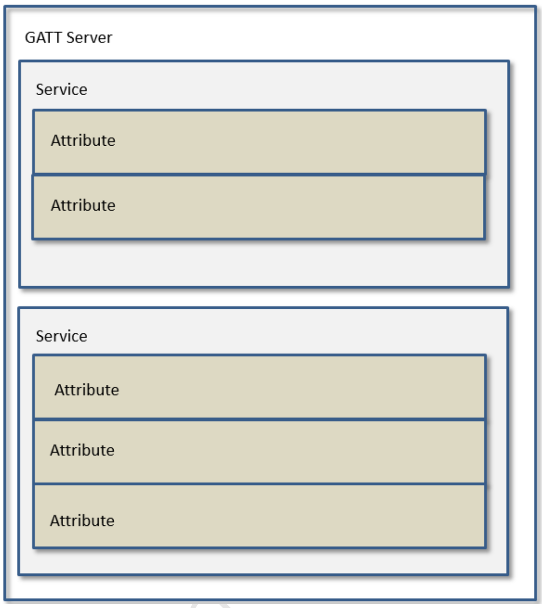


Figure 3.45: Attribute 构成 GATT service

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，Master 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/hids.h、stack/ble/attr/gatt_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/attr/gatt_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。Slave 和 Master 建立连接后，Master 通过 Service Discovery 过程解析读取到 Slave 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 Slave 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_appearanceUIID),     (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_periConnParamUUID),  (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 3.46: 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/b85m_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 Telink BLE Multiple Connection 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] = {
0x12, 0x03, 0x00, 0x00, 0x2A
};
{0,1,2,5,(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 Master 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_MIC_DATA
{0x12,0x2B,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x0}
const u8 my_OtaUUID[16]      = TELINK_SPP_DATA_OTA;
static u8 my_OtaData         = 0x00;
{0,3,16,1,(u8*)(&my_OtaUUID), (&my_OtaData), &otaMyWrite, &otaRead},
```

uuid 指向 my_OtaData 在 flash 中的地址，uuidLen 为 16，Master 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 Master 读取 Slave 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 Master。

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_LO(0x0111), U16_HI(0x0111), // bcdHID (USB HID version), 0x11,0x01
    0x00,                           // bCountryCode
    0x01                            // Flags
};
{0,1,2, 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 Master 读该 Attribute 时，会根据 pAttrValue 和 attrLen 返回 0x01000111 给 Master。

Master 读该 Attribute 时 BLE 抓包如下图，Master 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。

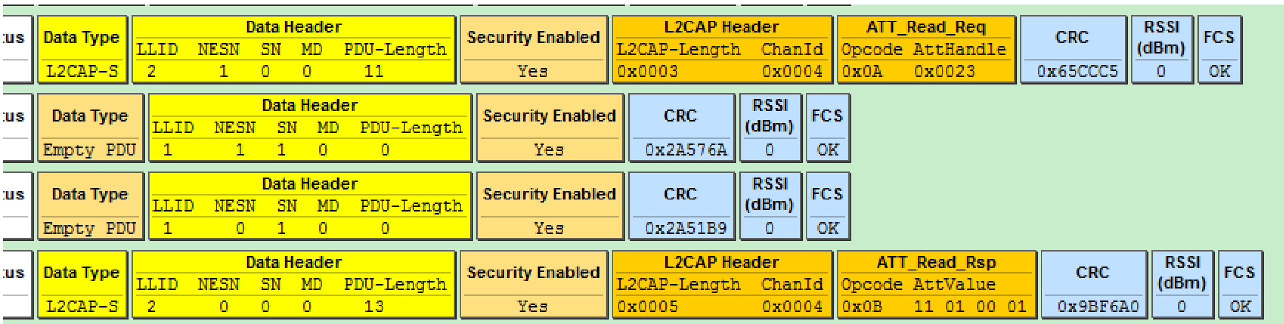


Figure 3.47: Master 读 hidInformation 的 BLE 抓包

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

```
u8        my_batVal[1]  = {99};
    {0,1,2,1,(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 Slave 主动 notify 或者 Master 主动读的方式传输给 Master，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 Slave 收到的 Attribute PDU 的 Attribute Opcode 为以下三个时，Slave 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.

Slave 收到以上写命令后，如果没有设置回调函数 w，Slave 会自动向 pAttValue 指针所指向的区域写 Master 传过来的值，写入的长度为 Master 数据包格式中的 l2capLen-3；如果 user 设置了回调函数 w，Slave 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 Master 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，Master 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 attri-bute

Figure 3.48: 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 attri-bute

Figure 3.49: 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 3.50: BLE 协议栈中 Execute Write Request

回调函数 w 的 void 型 p 指针指向 Master 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 Slave 收到的 Attribute PDU 的 Attribute Opcode 为以下两个时，Slave 会检查回调函数 r 是否被设置：

a) opcode = 0x0A, Read Request.

b) opcode = 0x0C, Read Blob Request.

Slave 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，Slave 不回复 Read Response/Read Blob Response 给 Master。
- 若返回值为其他值，Slave 从 pAttrValue 指针所指向的区域读 attrLen 个值用 Read Response/Read Blob Response 回复给 Master。

b) 如果用户没有设置回调读函数，Slave 从 pAttrValue 指针所指向的区域读 attrLen 个值用 Read Response/Read Blob Response 回复给 Master。

如果用户想在收到 Master 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 attnum 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 attnum 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;
```

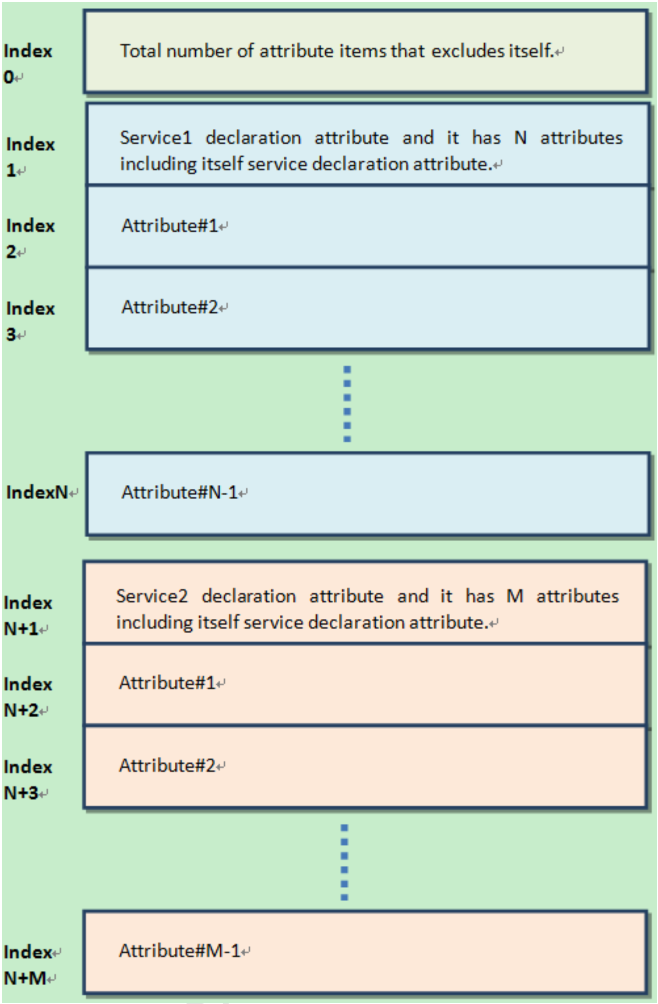


Figure 3.51: Service Attribute Layout

(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。

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	None	Request succeeds	Request succeeds	Request succeeds	Request succeeds
	Encryption, No MITM Protection	Error Resp.: Insufficient Authentication	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption
	Encryption, MITM Protection	Error Resp.: Insufficient Authentication	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption
	Encryption, MITM Protection, Secure Connections	Error Resp.: Insufficient Authentication	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption
Encrypted	None	N/A (Not possible to be encrypted without LTK)	Request succeeds	Request succeeds	Request succeeds
	Encryption, No MITM Protection		Request succeeds	Request succeeds	Request succeeds
	Encryption, MITM Protection		Error Resp.: Insufficient Authentication	Request succeeds	Request succeeds
	Encryption, MITM Protection, Secure Connections		Error Resp.: Insufficient Authentication	Error Resp.: Insufficient Authentication	Request succeeds

Table 10.2: Local device responds to a service request

Figure 3.52: 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 Slave 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_ENCRYPT | ATT_PERMISSIONS_AUTHEN)        //!< Read requires Authe
#define ATT_PERMISSIONS_AUTHEN_WRITE        (ATT_PERMISSIONS_WRITE | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN)        //!< Write requires Autl
#define ATT_PERMISSIONS_AUTHEN_RDWR         (ATT_PERMISSIONS_RDWR  | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN)        //!< Read & Write requi


#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_AUTHEN)       //!< Write requires Authorization
#define ATT_PERMISSIONS_AUTHOR_RDWR         (ATT_PERMISSIONS_RDWR | ATT_PERMISSIONS_AUTHOR)        //!< Read & Write requires Authorization
```

Figure 3.53: 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 Master 也有关系，比如 Slave 设置的 SMP 能支持的最高等级是 Authenticated pairing with encryption，但是 Master 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 Master 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 Slave 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 Master 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 Master 只有写该 CCC 后，Slave 会回复其安全级别不够，这会触发 Master 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，Telink BLE Multiple Connection 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。

3.4.24.1 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 V5.3, Vol 3, Part F, 3.4.4.9 ATT_READ_BY_GROUP_TYPE_REQ/3.4.4.10 ATT_READ_BY_GROUP_TYPE_RSP。

Master 发送 Read by Group Type Request，在该命令中指定起始和结束的 attHandle，指定 attGroupType。Slave 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。

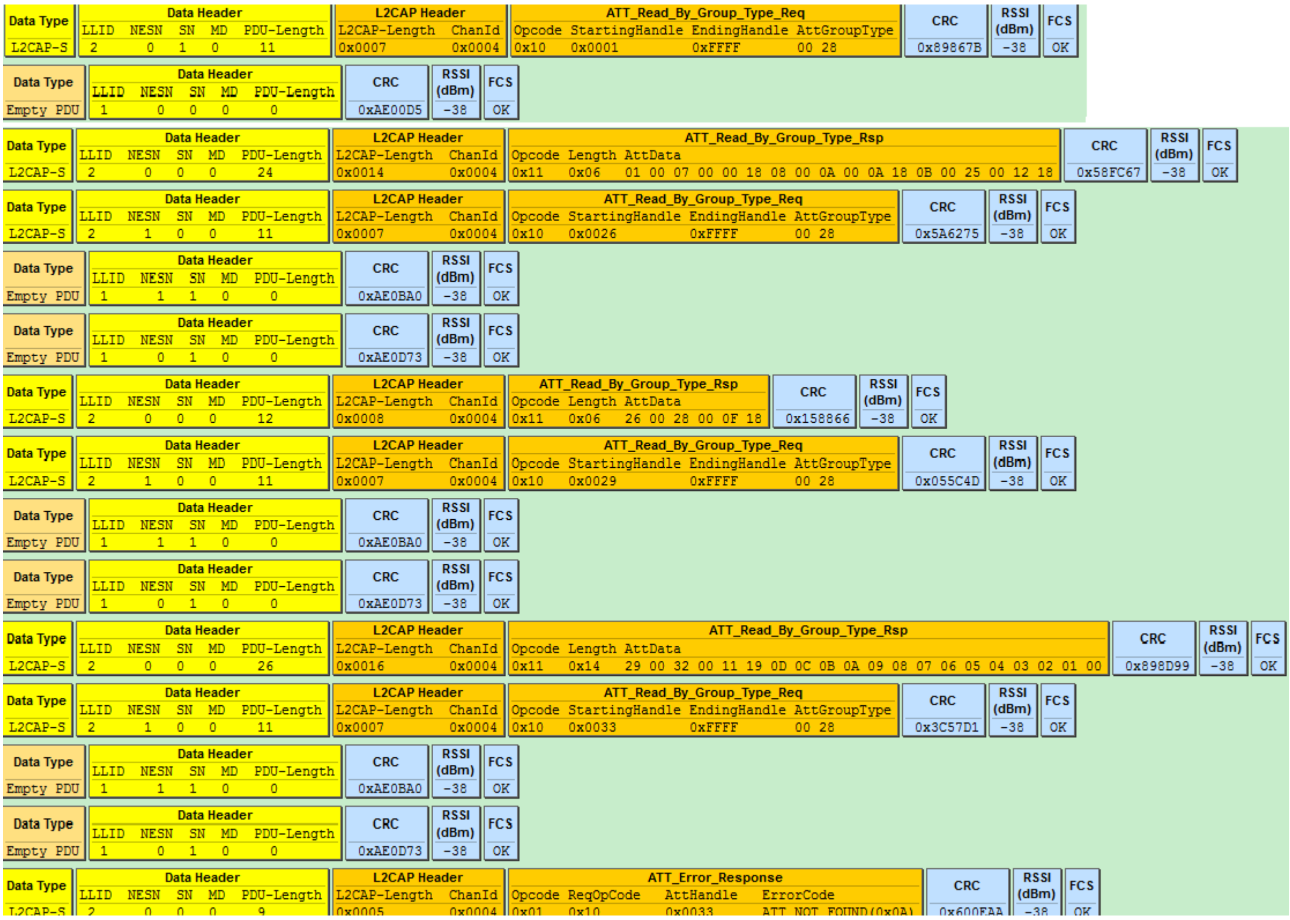


Figure 3.54: Read by Group Type Request Read by Group Type Response

上图所示，Master 查询 Slave 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)。

Slave 将以上 7 个 GROUP 的 attHandle 和 attValue 的信息通过 Read by Group Type Response 回复给 Master，最后一个 ATT_Error_Response 表明所有的 Attribute Group 都已回复完毕，Response 结束，Master 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 V5.3, Vol 3, Part F, 3.4.3.3 ATT_FIND_BY_TYPE_VALUE_REQ/3.4.3.4 ATT_FIND_BY_TYPE_VALUE_RSP。

Master 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。Slave 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。

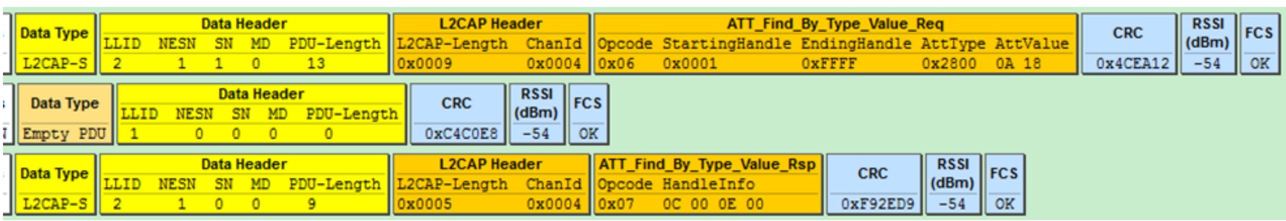


Figure 3.55: Find by Type Value Request Find by Type Value Response

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 V5.3, Vol 3, Part F, 3.4.4.1 ATT_READ_BY_TYPE_REQ/3.4.4.2 ATT_READ_BY_TYPE_RSP。

Master 发送 Read by Type Request，在该命令中指定起始和结束的 attHandle，指定 AttributeType。Slave 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。

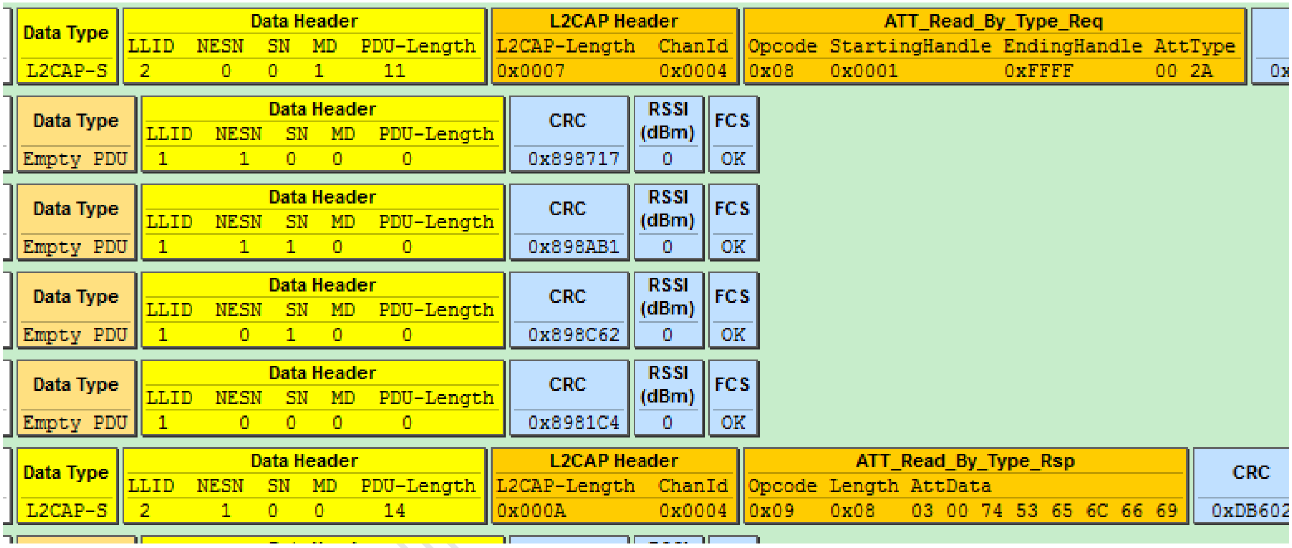


Figure 3.56: Read by Type Value Request Find by Type Value Response

上图所示，Master 读 attType 为 0x2A00 的 Attribute，Slave 中 Attribute Handle 为 0x0003 的 Attribute：

```
const u8 my_devName [] = {'t', 'S', 'e', 'l', 'f', 'i'};
     #define GATT_UUID_DEVICE_NAME          0x2a00
     const u16 my_devNameUUID = GATT_UUID_DEVICE_NAME;
     {0,1,2, 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。

3.4.24.4 Find information Request, Find information Response

Find information request 和 Find information response，详情请参考 Bluetooth Core Specification V5.3, Vol 3, Part F, 3.4.3.1 ATT_FIND_INFORMATION_REQ/3.4.3.2 ATT_FIND_INFORMATION_RSP。

Master 发送 Find information request，指定起始和结束的 attHandle。Slave 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 Master。如下图所示，Master 要求获得 attHandle 0x0016 ~0x0018 三个 Attribute 的 information，Slave 回复这三个 Attribute 的 UUID。

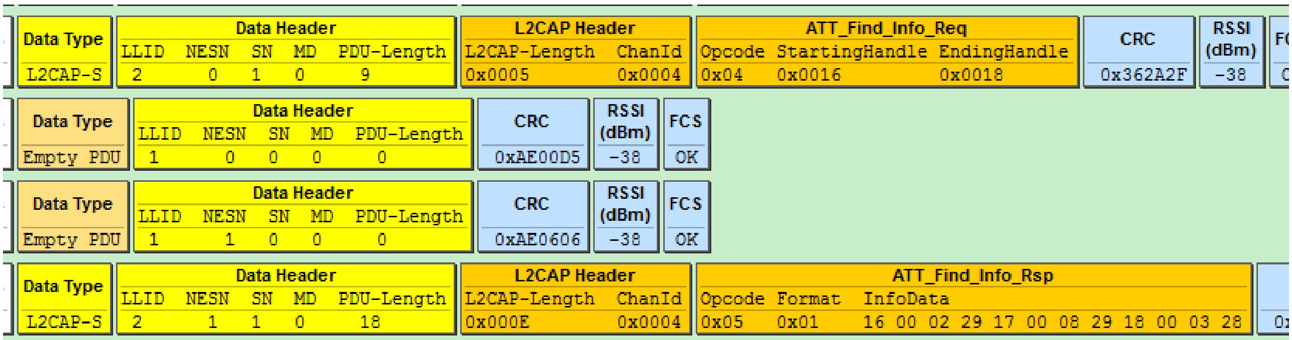


Figure 3.57: Find Information Request Find Information Response

3.4.24.5 Read Request, Read Response

Read Request 和 Read Response，详情请参考 Bluetooth Core Specification V5.3, Vol 3, Part F, 3.4.4.3 ATT_READ_REQ/3.4.4.4 ATT_READ_RSP。

Master 发送 Read Request，指定某一个 attHandle 为 0x0017，Slave 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。

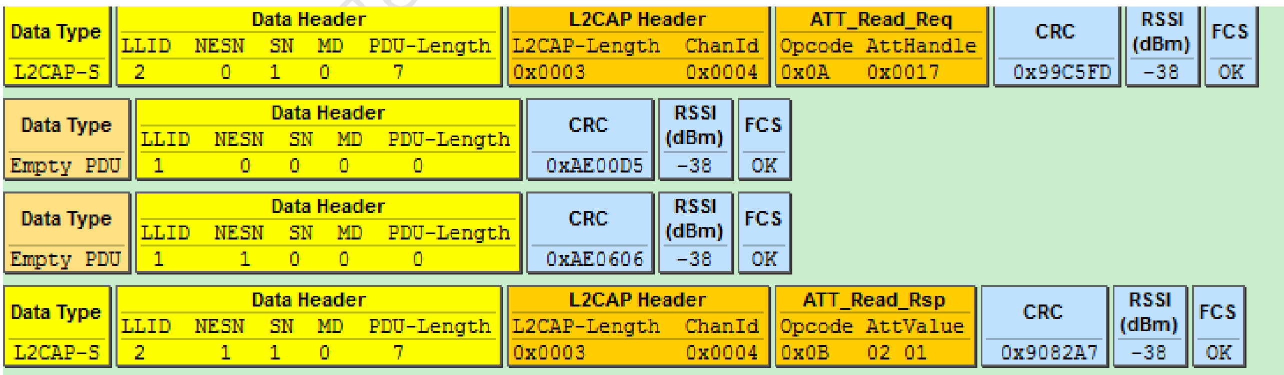


Figure 3.58: Read Request Read Response

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 V5.3, Vol 3, Part F, 3.4.4.5 ATT_READ_BLOB_REQ/3.4.4.6 ATT_READ_BLOB_RSP。

当 Slave 某个 Attribute 的 Attribute Value 值的长度超过 MTU_SIZE（目前 Telink BLE Multiple Connection SDK 中默认为 23）时，Master 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。Master 在 Read Blob Request 指定 attHandle 和 ValueOffset，Slave 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，Master 读 Slave 的 HID report map（report map 很大，远远超过 23）时，首先发送 Read Request，Slave 回 Read response，将 report map 前一部分数据回给 Master。之后 Master 使用 Read Blob Request，Slave 通过 Read Blob Response 回数据给 Master。

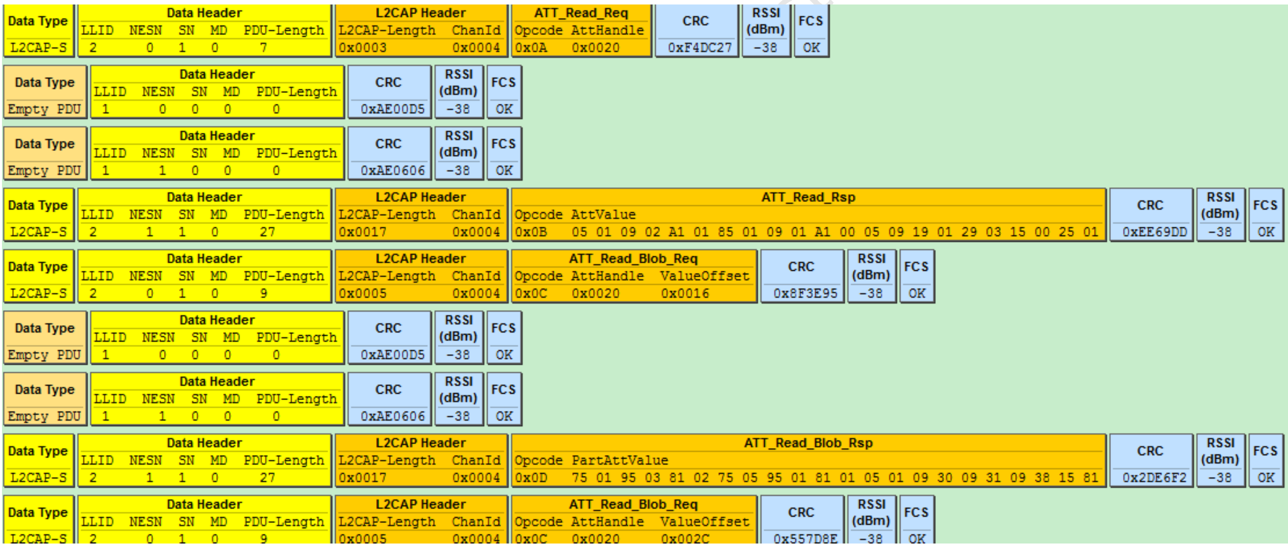


Figure 3.59: Read Blob Request Read Blob Response

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 V5.3, 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，Master 和 Slave 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
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	ClientRxMTU			
L2CAP-S	2	0	1	0	7	0x0003	0x0004	0x02	0x009E	0xC70102	-38	OK

Data Type	Data Header					L2CAP Header		ATT_Exchange_MTU_Rsp		CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	ServerRxMTU			
L2CAP-S	2	0	0	0	7	0x0003	0x0004	0x03	0x0017	0x1D88E1	-38	OK

Figure 3.60: Exchange MTU Request Exchange MTU Response

当 GATT 层的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 Master/Slave 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 Master 端 RX size：

```
ble_sts_t   blc_att_setMasterRxMTUSize(u16 master_mtu_size);
```

使用下面 API 修改 Slave 端 RX size：

```
ble_sts_t   blc_att_setSlaveRxMTUSize(u16 slave_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 V5.3, Vol 3, Part F, 3.4.5.1 ATT_WRITE_REQ/3.4.5.2 ATT_WRITE_RSP。

Master 发送 Write Request，指定某个 attHandle，并附带相关数据。Slave 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 Master 向 attHandle 为 0x0016 的 Attribute 写入 Attribute Value 为 0x0001，Slave 收到后执行该写操作，并回 Write Response。

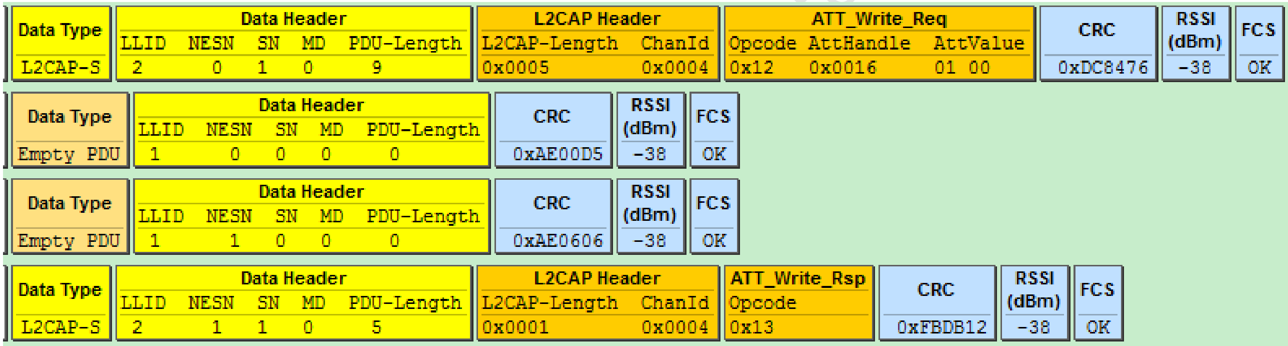


Figure 3.61: Write Request Write Response

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 V5.3, Vol 3, Part F, 3.4.5.3 ATT_WRITE_CMD。

Master 发送 Write Command，指定某个 attHandle，并附带相关数据。Slave 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 V5.3, 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：

> Telink BLE Multiple Connection 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，Master 向 Slave 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 Slave，Slave 向 Master 回 Prepare Write Response。之后 Master 发送 Prepare Write Request，偏移 0x12，将 " a new song" 部分数据写给 Slave，Slave 向 Master 回 Prepare Write Response。当 Master 将长属性值全部写完成后，发送 Execute Write Request 给 Slave，Flags 为 1：表示写立即生效，Slave 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。

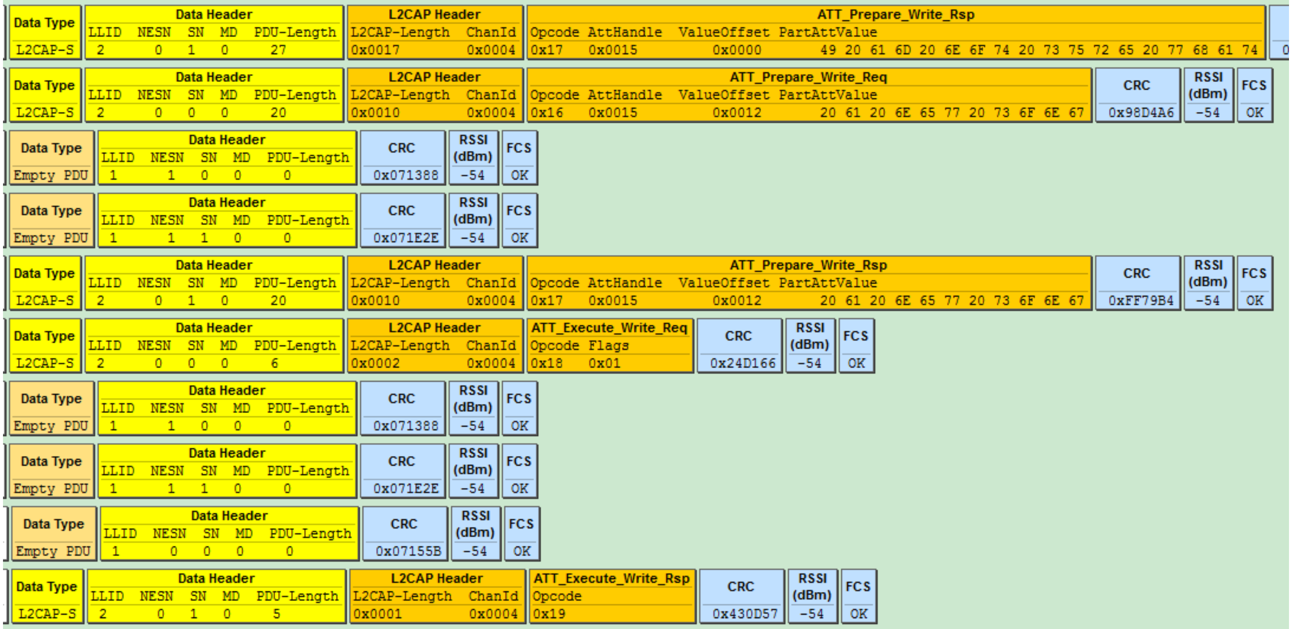


Figure 3.62: Write Long Characteristic Values 示例

34.24.11 Handle Value Notification

Handle Value Notification，详情请参考 Bluetooth Core Specification V5.3，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 3.63: Bluetooth Core Specification 中 Handle Value Notification

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

Telink BLE Multiple Connection 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	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

34.24.12 Handle Value Indication

Handle Value Indication，详情请参考 Bluetooth Core Specification V5.3, 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 3.64: Handle Value Indication in BLE Spec

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

Telink BLE Multiple Connection 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 Master 确认

34.24.13 Handle Value Confirmation

Handle Value Confirmation，详情请参考 Bluetooth Core Specification V5.3, Vol 3, Part F, 3.4.7.3 ATT_HANDLE_VALUE_CFM。

应用层每调用一次 blc_gatt_pushHandleValueIndicate，向 Master 发送 indicate 数据后，Master 会回复一个 confirm，表示对这个数据的确认，然后 Slave 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 3.65: 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

Telink BLE Multiple Connection 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*10 ms，即 300 ms。

3.4.3 GAP

3.4.3.1 GAP 初始化

Telink BLE Multiple Connection 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。

Telink BLE Multiple Connection 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 Slave 和 Master 刚刚连接进入 connection state，Slave 发送 SM_Security_Req 命令后，Master 发送 SM_Pairing_Req 请求开始配对，Slave 收到这个配对请求命令时，触发该事件，表示配对开始。

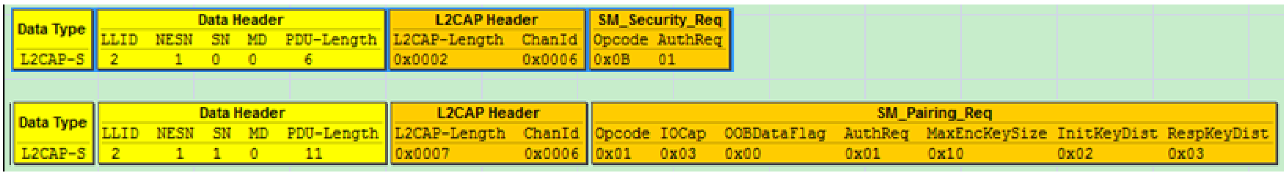


Figure 3.66: SMP_PAIRING_BEGIN 事件触发条件

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_Dsply_Resp_Input、PK_Resp_Dsply_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 Slave 或 Master 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 V5.3, 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_NUMUERIC_FAILED         0x0C
#define PAIRING_FAIL_REASON_PAIRING_TIEMOUT         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 Master 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：Slave 收到 Master 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_Dsply_Init_Input（即：Slave 端显示 6 位 pincode 码，Master 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 Slave 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 Slave 上看到的 6 位 pincode 码输入到 Master 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 Slave 设备启用 Passkey Entry 方式时，且使用的 PK_Init_Dsply_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 Slave 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：Slave 收到 Master 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，Master 和 Slave 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 Slave 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 Master 端发送 Exchange MTU Request，Slave 回复 Exchange MTU Response，还是 Slave 端发送 Exchange MTU Request，Master 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。Master 和 Slave 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_VLAUE_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 Master 发送 indicate 数据后，Master 会回复一个 confirm，表示对这个数据的确认，Slave 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种 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 V5.3, 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 Mulitple Connection SDK 提供以下 4 个安全等级，参考 Bluetooth Core Specification V5.3, 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，每个连接都可以使用不同安全级别进行通信；
> - 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 >= 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 Slave 端能够支持的最高安全等级是 Mode 1 Level 3，但是连接 Slave 的 Master 设置为不支持配对加密（最高只支持 Mode 1 Level 1），那么连接后 Slave 和 Master 不会进行配对流程，Slave 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_slave(le_security_mode_level_t  mode_level);
void blc_smp_setSecurityLevel_master(le_security_mode_level_t  mode_level);
```

说明：

在 Telink BLE Multiple Connection 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 Master role 和 Slave role 参数的 API(...)；
- 单独配置所有 Master role 参数的 API_master(...);
- 单独配置所有 Slave role 参数的 API_slave(...);

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);  //_slave()/_master()
```

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);    //_slave()/_master()
```

该套 API 用于配置 SMP IO 能力（见下图），决定 Key 产生的方式，参考 Bluetooth Core Specification V5.3, 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_Dsply_Init_Input, JustWorks, PK_Resp_Dsply_Init_Input },
    { JustWorks,              JustWorks,              PK_Resp_Dsply_Init_Input, JustWorks, PK_Resp_Dsply_Init_Input },
    { PK_Init_Dsply_Resp_Input, PK_Init_Dsply_Resp_Input, PK_BOTH_INPUT,          JustWorks, PK_Init_Dsply_Resp_Input },
    { JustWorks,              JustWorks,              JustWorks,              JustWorks, JustWorks                },
    { PK_Init_Dsply_Resp_Input, PK_Init_Dsply_Resp_Input, PK_Resp_Dsply_Init_Input, JustWorks, PK_Init_Dsply_Resp_Input },
};

static const stk_generationMethod_t gen_method_sc[5 /*Responder IOCap*/][5 /*Initiator IOCap*/] = {
    { JustWorks,              JustWorks,              PK_Resp_Dsply_Init_Input, JustWorks, PK_Resp_Dsply_Init_Input },
    { JustWorks,              Numric_Comparison,      PK_Resp_Dsply_Init_Input, JustWorks, Numric_Comparison },
    { PK_Init_Dsply_Resp_Input, PK_Init_Dsply_Resp_Input, PK_BOTH_INPUT,          JustWorks, PK_Init_Dsply_Resp_Input },
    { JustWorks,              JustWorks,              JustWorks,              JustWorks, JustWorks                },
    { PK_Init_Dsply_Resp_Input, Numric_Comparison,      PK_Resp_Dsply_Init_Input, JustWorks, Numric_Comparison },
};
```

Figure 3.67: 根据不同 IO 能力映射 KEY 产生方法

```
void    blc_smp_enableAuthMITM(int MITM_en);        //_slave()/_master()
```

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);    //_slave()/_master()
```

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 V5.3, Vol 3, Part H, 2.3.5.1 Selecting Key Generation Method。

```
void    blc_smp_setBondingMode(bonding_mode_t mode);    //_slave()/_master()
```

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);    //_slave()/_master()
```

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);  //_slave()/_master()
```

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);   //_slave()/_master()
```

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：

> Slave 和 Master 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);    //_slave()/_master()
```

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 Slave 可以发送，用于主动请求对端 Master 进行配对流程，是 SMP 的可选流程。
- SMP 配对请求（Pairing Request）只有 Master 可以发送，用于通知 Slave 开始配对流程。

3.4.4.3.1 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 Slave 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，Slave 不会主动发送 Security Request
    SecReq_IMM_SEND = BIT(0),       //连接建立后，Slave 会立即发送 Security Request
    SecReq_PEND_SEND = BIT(1),      //连接建立后，Slave 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 Slave 配置为 SecReq_PEND_SEND，且在 pending_ms 前就收到 Master 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），Master 有时候不一定会主动发起 LL_ENC_REQ 来加密链路，此时如果 Slave 发 Security Request 可以触发 Master 加密链路。如果 Slave 配置为 SecReq_PEND_SEND，且在 pending_ms 之前已经收到 Master 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。

3.4.4.3.2 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 Master 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,   // Master 发送 Pairing Request 依赖于收到 Slave 发送的 Security Request
    PairReq_AUTO_SEND = 1,          // Master 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。

3444.1 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);
```

3444.2 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_master(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();
```

3444.3 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_CAPABLITY_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_CAPABLITY_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_slave(Authenticated_Pairing_with_Encryption);
blc_smp_setSecurityParameters_slave(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。

- 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_setParingMethods(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_setParingMethods(LE_Secure_Connection);
blc_smp_enableAuthMITM(1);
blc_smp_setIoCapability(IO_CAPABLITY_DISPLAY_YESNO);
blc_smp_smpParamInit();
```

或

```
blc_smp_setSecurityLevel_master(Authenticated_LE_Secure_Connection_Pairing_with_Encryption);
blc_smp_setSecurityParameters_master(Bondable_Mode, 1, LE_Secure_Connection, 0, 0,
    IO_CAPABLITY_DISPLAY_YESNO);
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 Master 还是 Slave，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。

3.4.4.5.1 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 Telink BLE Multiple Connection SDK，SMP Storage 区域起始位置由宏 FLASH_ADR_SMP_PAIRING 指定 (1M Flash 默认为 0xFA000, 512K Flash 默认为 0x78000)。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);
```

3.4.4.5.2 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 Master 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 Master，如果 bit7 为 0，则代表在该连接中扮演 Slave 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 Master 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	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	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	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	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	B4	FE	9F	98	29	A5	9D	97	DE	70	62	AA

Figure 3.68: SMP Storage Bonding Info Master

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 isMaster, u8 slaveDevIdx, u8 addr_type, u8* addr,
↪  smp_param_save_t* smp_param_load);
```

- isMaster：自身的角色，0 表示 Slave，非 0 表示 Master；
- slaveDevIdx：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 Master 角色，提供一个根据 peer Slave 的 MAC 获取其配对状态的 API：

```
u32 blc_smp_searchBondingSlaveDevice_by_PeerMacAddress( u8 peer_addr_type, u8* peer_addr);
```

- peer_addr_type：peer Slave 的地址类型，参考 BLE_ADDR_PUBLIC 和 BLE_ADDR_RANDOM;
- peer_addr：peer Slave 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_deleteBondingSlaveInfo_by_PeerMacAddress(u8 peer_addr_type, u8* peer_addr);
```

- peer_addr_type：peer Slave 的地址类型，参考 BLE_ADDR_PUBLIC 和 BLE_ADDR_RANDOM;
- peer_addr：peer Slave 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。

3.4.4.5.3 最大绑定数量

对于 Telink BLE Multiple Connection SDK，默认最多可以保存 8 个有效 peer Slave 的 SMP 信息，和 4 个有效 peer Master 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 Master 和 Slave 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：

```
void blc_smp_setBondingDeviceMaxNumber ( int peer_slave_max,
int peer_master_max);
```

- peer_slave_max：自身作为 Master 角色的最大 peer Slave 绑定数量；
- peer_master_max：自身作为 Slave 角色的最大 peer Master 绑定数量。

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 Slave 后，一旦绑定第 9 个 peer Slave，最老的那个（第 1 个）peer Slave 的 Bonding Info 失效，并向 Flash 中继续存储第 9 个 peer Slave 设备的 Bonding Info。

用户可以通过以下 API 获取当前 Slave 或 Master 的绑定数量:

```
u8 blc_smp_param_getCurrentBondingDeviceNumber(u8 isMasterRole, u8 slaveDevIdx);
```

- isMasterRole：自身的角色，0 表示 Slave，非 0 表示 Master；
- slaveDevIdx：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，isMasterRole 为 0 时，表示绑定的有效 peer Master 的数量，isMasterRole 非 0 时，表示绑定的有效 peer Slave 的数量。

3.4.4.54 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 Master 的 Bonding Device Max Number 为 2 时，先后配对的两个 peer Slave 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 isMaster, u8 slaveDevIdx, u8 index,
↪  smp_param_save_t* smp_param_load);
```

- isMaster：自身的角色，0 表示 Slave，非 0 表示 Master；
- slaveDevIdx：在不涉及多地址功能时，该参数为 0；
- index：表示要读取的 Master 或 Slave 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 Custom Pair

多主多从的设备中，如果 Master 设备 disable 了 SMP，则 SDK 无法自动完成配对和解配对操作，需要在应用层添加配对管理。基于此，Telink 自定义了一套配对与解配对的方案。

如果用户需要使用自定义的配对管理，首先需要初始化该功能，调用如下 API：

```
blc_smp_setSecurityLevel_master(No_Security);//禁用 SMP 功能
user_master_host_pairing_management_init();//自定义方式
```

(1) Flash 存储方法设计

默认使用的 flash 数据区 sector 为 0x7C000 - 0x7CFFF，在 app_config.h 中可以修改宏：

```
#define FLASH_ADR_CUSTOM_PAIRING                0x7C000
#define FLASH_CUSTOM_PAIRING_MAX_SIZE           4096
```

将 flash 0x7C000 开始每 8 个 bytes 划分为一个 area，称 8 bytes area。每个 area 可以存储一个 Slave 的 mac address，其中第一个 byte 是标志位，第二个 byte 为地址类型，后面 6 个为 6 bytes 的 mac address。

```
typedef struct {
        u8 bond_mark;
        u8 adr_type;
        u8 address[6];
} macAddr_t;
```

flash 存储过程中使用依次往后推 8 bytes area 的方法，第一个有效 Slave mac 存储在 0x7C000-0x7C007，将 0x7C000 的第一个 byte 标志位写为 0x5A，表示当前地址有效；当存储第二个有效 mac address 时存储在 0x7C008-0x7C00f，将 0x7C008 打上标记 0x5A；当存储第三个有效 mac address 时存储在 0x7C010-0x7C017，将 0x7C010 打上标记 0x5A。

如果要某个 Slave 设备解配对，多主多从设备需要擦掉这个设备的 MAC address，只需要将之前存储该 MAC address 8 bytes area 的标志位写为 0x00 即可；如擦掉上面三个 device 中的第一个 device，将 0x7C000 写为 0x00 即可。

采用上面这种 8bytes 顺延方法的原因是，程序在运行过程中不能调用 flash_erase_sector 这个函数擦 flash，因为该操作擦一个 sector 4K 的 flash 耗时在 20-200ms 之间，这个时间会引起 BLE 时序错误。

将所有的 Slave MAC 的配对存储和解配对擦除使用 0x5A 和 0x00 标志位来表示，当 8 bytes area 越来越多，可能会占满整个 sector 4K flash 导致出错时，在初始化的时候加了特别处理：从 0x7C000 开始读取 8 bytes area 信息，将所有的有效 MAC address 读到 RAM 中的 Slave MAC table。这过程中检查是否 8 bytes area 太多，太多的话，就擦掉整个 sector，然后将 RAM 中维护的 Slave MAC table 重新写回 0x7C000 开始的 8 bytes area。

(2) Slave MAC table

```
/* define pair slave max num,
   if exceed this max num, two methods to process new slave pairing
   method 1: overwrite the oldest one(telink demo use this method)
   method 2: not allow paring unness unfair happend  */
#define USER_PAIR_SLAVE_MAX_NUM       4  //telink demo use max 4, you can change


typedef struct {
    u8 bond_mark;
    u8 adr_type;
    u8 address[6];
} macAddr_t;


typedef struct {
    u32 bond_flash_idx[USER_PAIR_SLAVE_MAX_NUM];  //mark paired slave mac addres
    macAddr_t bond_device[USER_PAIR_SLAVE_MAX_NUM];  //macAddr_t alreay defined
    u8 curNum;
} user_salveMac_t;

```

Figure 3.69: Slave MAC table

```
user_salveMac_t user_tbl_slaveMac;
```

用上面结构在 RAM 中使用 Slave MAC table 维护所有的配对设备，改变宏 USER_PAIR_SLAVE_MAX_NUM 即可定义自己想要的最多允许几个配对，Telink BLE Multiple Connection SDK 中默认为 4，指维护 4 个设备的配对，user 可以修改这个值。

user_tbl_slaveMac 中 curNum 表示当前 flash 上记录了几个有效的 Slave 设备，bond_flash_idx 数组记录有效地址在 flash 上的 8 bytes area 起始地址相对于 0x7C000 的偏移量（当解配对这个设备时，可以通过这个偏移量找到 8 bytes area 的标志位，将其写为 0x00），bond_device 数组记录 MAC address。

(3) 相关 API 说明

基于上面 FLASH 存储设计和 RAM 中 Slave MAC table 的设计，分别有以下几个 API 可以调用。

a. user_master_host_pairing_management_init

```
void    user_master_host_pairing_management_init(void);
```

用户自定义配对管理初始化函数，启用自定义方式时需要调用该初始化函数.

b. user_tbl_slave_mac_add

```
int user_tbl_slave_mac_add(u8 adr_type, u8 *adr);
```

添加一个 Slave mac，return 1 表示成功，0 表示失败。当有新的设备配对上时，需要调用此函数。

函数先判断当前 flash 和 Slave MAC table 中设备是否已经到达最大值。若没有到最大值，无条件添加到 Slave MAC table，并在 FLASH 的一个 8 bytes area 上存储。

若已经到最大值。涉及到处理的策略问题：是不允许配对还是直接覆盖最老的，Telink demo 的方法是直接覆盖最老的，先使用 user_tbl_slave_mac_delete_by_index(0) 删掉当前设备，再往 Slave mac table 里面加入新的。User 可以根据自己的策略去修改这个函数的实现。

c. user_tbl_slave_mac_search

```
int user_tbl_slave_mac_search(u8 adr_type, u8 * adr)
```

根据 adv report 的设备地址搜索该设备是否已经在 Slave MAC table 中，即判断当前发广播包的设备是否之前已经和 Master 配对上，若是已经配对过的设备可以直接连接。

d. user_tbl_slave_mac_delete_by_adr

```
int user_tbl_slave_mac_delete_by_adr(u8 adr_type, u8 *adr)
```

通过指定地址删除一个配对的设备。

e. user_tbl_slave_mac_delete_by_index

```
void user_tbl_slave_mac_delete_by_index(int index)
```

通过指定 index 删除配对设备。Index 值反映的是设备配对的顺序。如果最大配对个数为 1，配上的那个设备 index 永远为 0；如果如果最大配对个数为 2，第一个配上的设备 index 为 0，第二个配上的设备 index 为 1；依次类推。

f. user_tbl_slave_mac_delete_all

```
void user_tbl_slave_mac_delete_all(void)
```

删除所有配对设备。

(4) 连接和配对

Master 收到 Controller 上报的广播包时，有以下两种情况会和 Slave 进行连接：

a. 调用函数 user_tbl_slave_mac_search 来检查当前设备是否已经和 Master 配对过并且没有被解配对，如果已经配对过，可以自动连接。

```
master_auto_connect = user_tbl_slave_mac_search(pa->adr_type, pa->mac);
if(master_auto_connect) { create connection }
```

b. 若当前广播设备不在 Slave MAC table 里面，不符合自动连接，检查是否满足手动配对条件。SDK 中默认设置了两个手动配对方案，在当前广播设备距离足够近的前提下，一是多主多从设备上配对键被按下；二是当前广播数据是 Telink 定义的配对广播包数据。代码：

```
//manual paring methods 2: special paring ADV data
if(!user_manual_paring){  //special ADV pair data can also trigger pairing
    user_manual_paring = (memcmp(pa->data, telink_adv_trigger_paring, sizeof(telink_adv_trigger_paring)) == 0) && (rssi > -56);
}
```

Figure 3.70: 连接配对代码示例 1

```
if(user_manual_pairing) { create connection }
```

若是手动配对触发的连接建立，在连接成功建立后，即 HCI LE CONECTION ESTABLISHED EVENT 上报时，将当前设备添加到 Slave MAC table 中：

```
//manual paring, device match, add this device to slave mac table
if(blm_manPair.manual_pair && blm_manPair.mac_type == pCon->peer_adr_type && !memcmp(blm_manPair.mac, pCon->mac, 6)){
    blm_manPair.manual_pair = 0;

    user_tbl_slave_mac_add(pCon->peer_adr_type, pCon->mac);
}
```

Figure 3.71: 连接配对代码示例 2

c. 解配对

当解配对条件生效时,多主多从设备先调用 blc_llms_disconnect 断开连接,然后调用 user_tbl_salve_mac_delete_by_adr 函数删除该设备。

3.4.6 Device Manage & Simple SDP

如前面对于 GATT 的描述，在 BLE 中，Slave 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 Master，想要获取 Slave 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，Telink BLE Multiple Connection SDK 为用户提供了一个连接设备管理方案的实现 Device Manage 以及一个 Master 做 SDP 获取 peer Slave 的 GATT Service 表的简单实现。不仅可以为 Master 管理 peer Slave 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。

Telink BLE Multiple Connection 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: master; 1: slave
    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_adrType;
    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。

```
/*
 * 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 3.72: 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[]。

```c
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 3.73: connection completed event handle

如果自己是 Master 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：

```c
                    user_tbl_slave_mac_add(pConnEvt->peerAddrType, pConnEvt->peerAddr);
                }
            #endif


            #if (BLE_MASTER_SIMPLE_SDP_ENABLE)
                memset(&cur_sdp_device, 0, sizeof(dev_char_info_t));
                cur_sdp_device.conn_handle = pConnEvt->connHandle;
                cur_sdp_device.peer_adrType = pConnEvt->peerAddrType;
                memcpy(cur_sdp_device.peer_addr, pConnEvt->peerAddr, 6);

                u8  temp_buff[sizeof(dev_att_t)];
                dev_att_t *pdev_att = (dev_att_t *)temp_buff;

                /* att_handle search in flash, if success, load char_handle directly from flash, no need SDP again */
                if( dev_char_info_search_peer_att_handle_by_peer_mac(pConnEvt->peerAddrType, pConnEvt->peerAddr, pdev_att) ){
                    //cur_sdp_device.char_handle[1] =                                  //Speaker
                    cur_sdp_device.char_handle[2] = pdev_att->char_handle[2];          //OTA
                    cur_sdp_device.char_handle[3] = pdev_att->char_handle[3];          //consume report
                    cur_sdp_device.char_handle[4] = pdev_att->char_handle[4];          //normal key report
                    //cur_sdp_device.char_handle[6] =                                  //BLE Module, SPP Server to Client
                    //cur_sdp_device.char_handle[7] =                                  //BLE Module, SPP Client to Server

                    /* add the peer device att_handle value to conn_dev_list */
                    dev_char_info_add_peer_att_handle(&cur_sdp_device);
                }
                else
                {
                    master_sdp_pending = pConnEvt->connHandle;  // mark this connection need SDP

                    #if (BLE_MASTER_SMP_ENABLE)
                    #else
                        app_register_service(&app_service_discovery);  //No SMP, service discovery can initiated now
                    #endif
                }
            #endif
        }
    }

    return 0;
}
```

Figure 3.74: 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 Slave 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 simple_sdp.c

```c
/**
 * @brief   SDP handler.
 *          !!! Note: This is a simple SDP processing implemented by telink.
 * @param   none.
 * @return  none.
 */
void app_service_discovery (void)
{
    att_db_uuid16_t     db16[ATT_DB_UUID16_NUM];
    att_db_uuid128_t    db128[ATT_DB_UUID128_NUM];
    memset (db16, 0, ATT_DB_UUID16_NUM * sizeof (att_db_uuid16_t));
    memset (db128, 0, ATT_DB_UUID128_NUM * sizeof (att_db_uuid128_t));

    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
    {
        cur_sdp_device.char_handle[2] = blm_att_findHandleOfUuid128 (db128, my_OtaUUID);            //OTA
        cur_sdp_device.char_handle[3] = blm_att_findHandleOfUuid16 (db16, CHARACTERISTIC_UUID_HID_REPORT,
                    HID_REPORT_ID_CONSUME_CONTROL_INPUT | (HID_REPORT_TYPE_INPUT<<8));      //consume report(media key report)
        cur_sdp_device.char_handle[4] = blm_att_findHandleOfUuid16 (db16, CHARACTERISTIC_UUID_HID_REPORT,
                    HID_REPORT_ID_KEYBOARD_INPUT | (HID_REPORT_TYPE_INPUT<<8));             //normal key report
        //cur_sdp_device.char_handle[6] = blm_att_findHandleOfUuid128 (db128, TelinkSppDataServer2ClientUUID);      //BLE Module, SPP Server to Client
        //cur_sdp_device.char_handle[7] = blm_att_findHandleOfUuid128 (db128, TelinkSppDataClient2ServerUUID);      //BLE Module, SPP Client to Server

        /* add the peer device att_handle value to conn_dev_list after service discovery is correctly finished */
        dev_char_info_add_peer_att_handle(&cur_sdp_device);

        /* peer device att_handle value store in flash */
        dev_char_info_store_peer_att_handle(&cur_sdp_device);

        //my_dump_str_data(APP_DUMP_SDP_EN,"OTA handle", (u8*)&cur_sdp_device.char_handle[2], 2);
        //my_dump_str_data(APP_DUMP_SDP_EN,"CMKEY handle", (u8*)&cur_sdp_device.char_handle[3], 2);
        //my_dump_str_data(APP_DUMP_SDP_EN,"KBKEY handle", (u8*)&cur_sdp_device.char_handle[4], 2);
    }

    master_sdp_pending = 0;  //service discovery finish
}
```

Figure 3.75: service discovery

注意： SDP 是一个很复杂的部分，对于 Telink BLE Multiple Connection 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。

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。

Table 4.1: 低功耗模式

Module	suspend	deepsleep retention	deepsleep
Sram	100% keep	first 16K/32K/64K 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。此时 IC 电流 B85m 在 60-70uA 之间，B91 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，包括：

a) 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。

b) B85m IC 还有控制 Dfifo 状态的 digital register。对应 drivers/dfifo.h 中的相关 API，user 在使用这些 API 的时候，必须确保每次 suspend wake_up 后都要重新设置。

(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），B85m 和 B91 增加了一种 sleep mode 3：deepsleep with Sram retention mode，简称 deepsleep retention（或 deep retention）。根据 Sram retention area 的大小不同，B85m 分为 deepsleep retention 16K Sram 和 deepsleep retention 32K Sram，B91 分为 deepsleep retention 32K Sram 和 deepsleep retention 64K Sram。

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 16K（或前 32K）可以保持不掉电，剩余的 Sram 全部掉电。

4.1.2 低功耗唤醒源

B85m 和 B91 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。Telink BLE Multiple Connection 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(4),
     PM_WAKEUP_TIMER = BIT(6),
}SleepWakeupSrc_TypeDef;
```

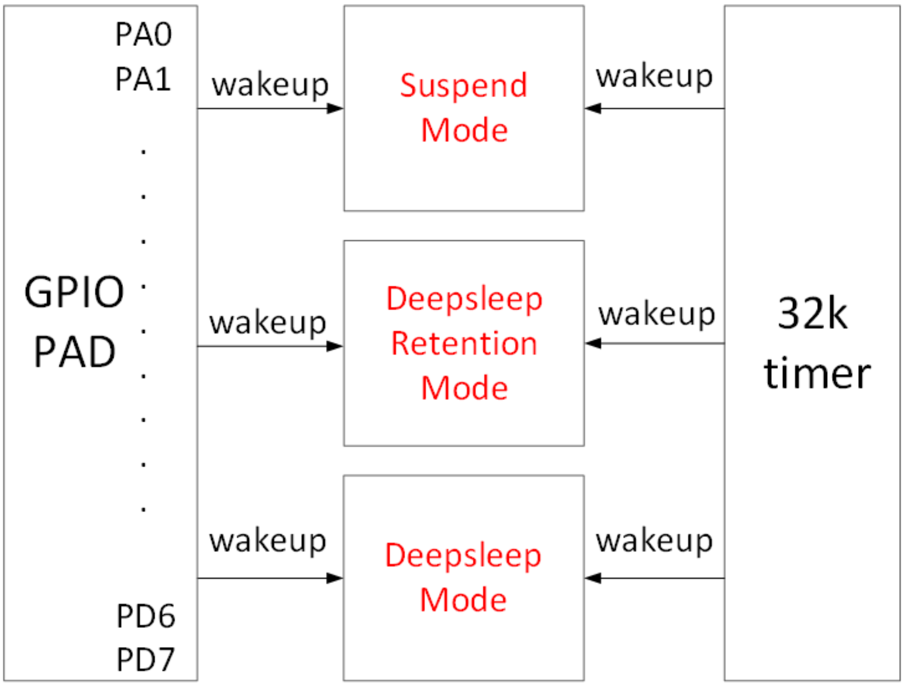


Figure 4.1: MCU 硬件唤醒源

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 Telink BLE Multiple Connection 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 (SleepMode_TypeDef 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            = 0x80,
    ......
}SleepMode_TypeDef;
```

- 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：

```
enum {
        WAKEUP_STATUS_TIMER   = BIT(1),
        WAKEUP_STATUS_PAD     = BIT(3),

        STATUS_GPIO_ERR_NO_ENTER_PM   = BIT(7),
};
```

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* 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。

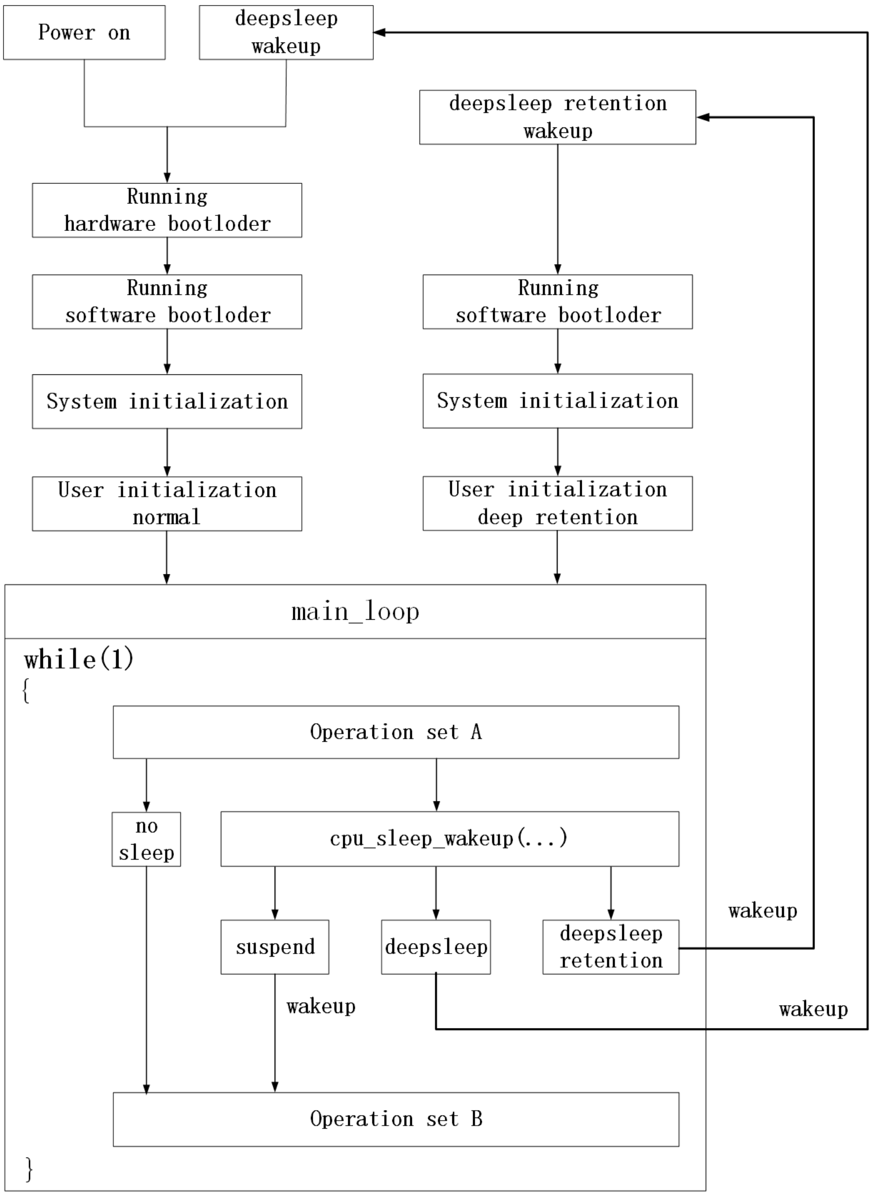


Figure 4.2: Sleep Mode Wakeup Work Flow

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 16K/32K/64K 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 16K/32K/64K 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 initilization 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，以便做一些区分于 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

Telink BLE Multiple Connection SDK 中对 Legacy advertising state、Scanning state、ACL connection master 和 ACL connection slave 做了低功耗管理。

需要说明的是，SDK 目前 slave 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 slave 接受了对方 master 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。

slave 使用 latency 的限制条件：

（1）只有 Legacy advertising 和 ACL slave 任务。

（2）ACL slave 只有 1 个连接（后面 SDK 会优化支持更多的 slave 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。

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：

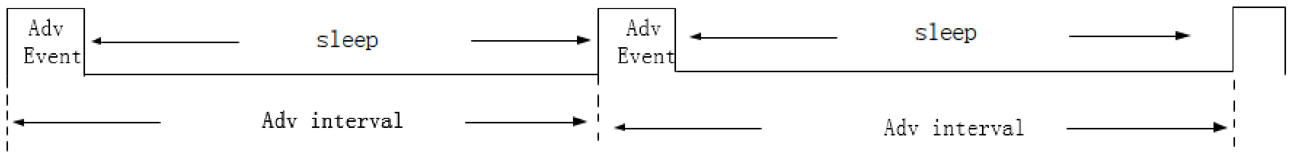


Figure 4.3: sleep 处于 advertising 状态时序

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”

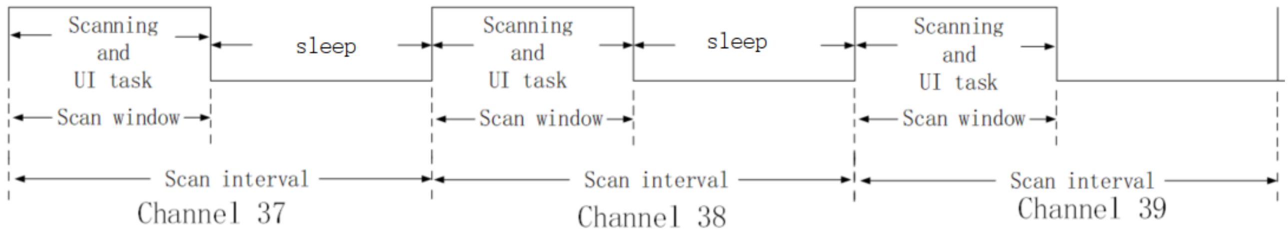


Figure 4.4: sleep for scanning for only scanning

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

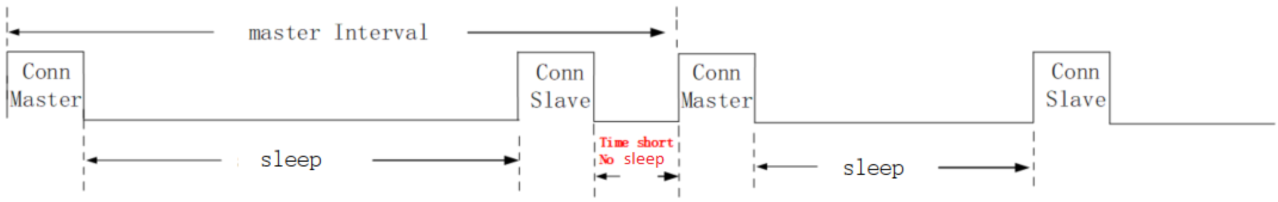


Figure 4.5: sleep for connection

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。Telink BLE Multiple Connection 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_SLAVE      = BIT(2),
    PM_SLEEP_ACL_MASTER     = BIT(3),
}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_SLAVE 和 PM_SLEEP_ACL_MASTER 分别用于控制 ACL connection slave 和 ACL connection master 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_SLAVE | PM_SLEEP_ACL_MASTER);

MCU 在 Legacy advertising state、Scanning state、ACL connection slave 和 ACL connection master 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 (SleepWakeupSrc_TypeDef 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 master 和 ACL connection slave 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、master task、slave 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 16K/32K/64 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。

B85m 可选的模式只有以下两种，16K 和 32K，SDK 默认的 deepsleep retention mode 为 DEEPSLEEP_MODE_RET_SRAM_LOW32K：

```
typedef enum {
    DEEPSLEEP_MODE_RET_SRAM_LOW16K      = 0x43,
    DEEPSLEEP_MODE_RET_SRAM_LOW32K      = 0x07,
}SleepMode_TypeDef;
```

B91 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 {
    DEEPSLEEP_MODE_RET_SRAM_LOW32K      = 0x21,
    DEEPSLEEP_MODE_RET_SRAM_LOW64K      = 0x03,
}SleepMode_TypeDef;
```

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(SleepMode_TypeDef 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
SleepMode_TypeDef  sleep_M = SUSPEND_MODE;
if( blmsPm.deepRt_en && (u32)(blmsPm.current_wakeup_tick - clock_time() -
    blmsPm.deepRet_thresTick) < BIT(30) ){
    sleep_M = (SleepMode_TypeDef)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 thres_ms)
{
    blmsPm.deepRet_thresTick = thres_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

Telink BLE Multiple Connection 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 ////////////////////////////////
    blc_pm_setSleepMask( PM_SUSPEND_ADV | PM_SUSPEND_SCAN | PM_SUSPEND_SLAVE |
 ↪  PM_SUSPEND_MASTER);
}
```

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、master、slave)
    //sleep_taskMask 是否允许该状态 (adv、scan、master、slave) 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 bltmsPm.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、master task、slave 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 Master role 的 Btx Event 或 Conn state Slave 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(SleepMode_TypeDef 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。

4.3 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 Slave 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。

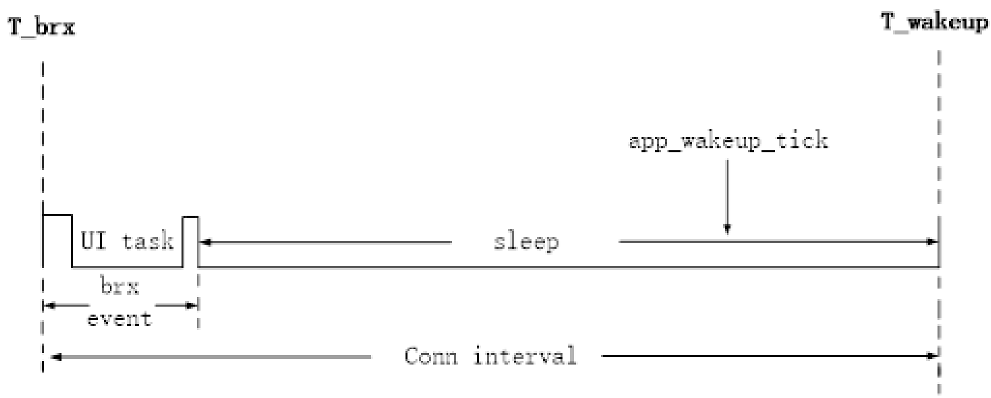


Figure 4.6: Early wake_up at app_wakup_tick

5 低电检测

Telink BLE Multiple Connection SDK 中暂未开放低电检测功能。

6 OTA

关于 OTA，整个流程和单连接完全一致，请参考 Telink BLE Single Connection SDK Handbook 对应章节的介绍，包括 B85m 系列（825x + 827x）和 B91 系列。

相比 Telink BLE Single Connection SDK ，Telink BLE Multiple Connection SDK 增加一个限制：同一时刻只允许在一个 slave connection 上进行 OTA。

7 按键扫描

这部分可以参考 Telink BLE Single Connection SDK Handbook 对应章节的介绍，包括 B85m 系列（825x + 827x）和 B91 系列。

8 LED 管理

这部分请参考 Telink BLE Single Connection SDK Handbook 对应章节的介绍，包括 B85m 系列（825x + 827x）和 B91 系列。

9 软件定时器

这部分请参考 Telink BLE Single Connection SDK Handbook 对应章节的介绍，包括 B85m 系列（825x + 827x）和 B91 系列。

10 功能参考 Demo

本章节将介绍 sdk/vendor/xx_feature 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。

10.1 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 Telink B91 BLE Multiple Connection SDK 为例，应用层代码在 eagle_ble_sdk/vendor/B91_feature/feature_backup 下，需要修改 eagle_ble_sdk/vendor/B91_feature/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 10.1: 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 Master 设备或者 Slave 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 conn_master_num 的值为 1：

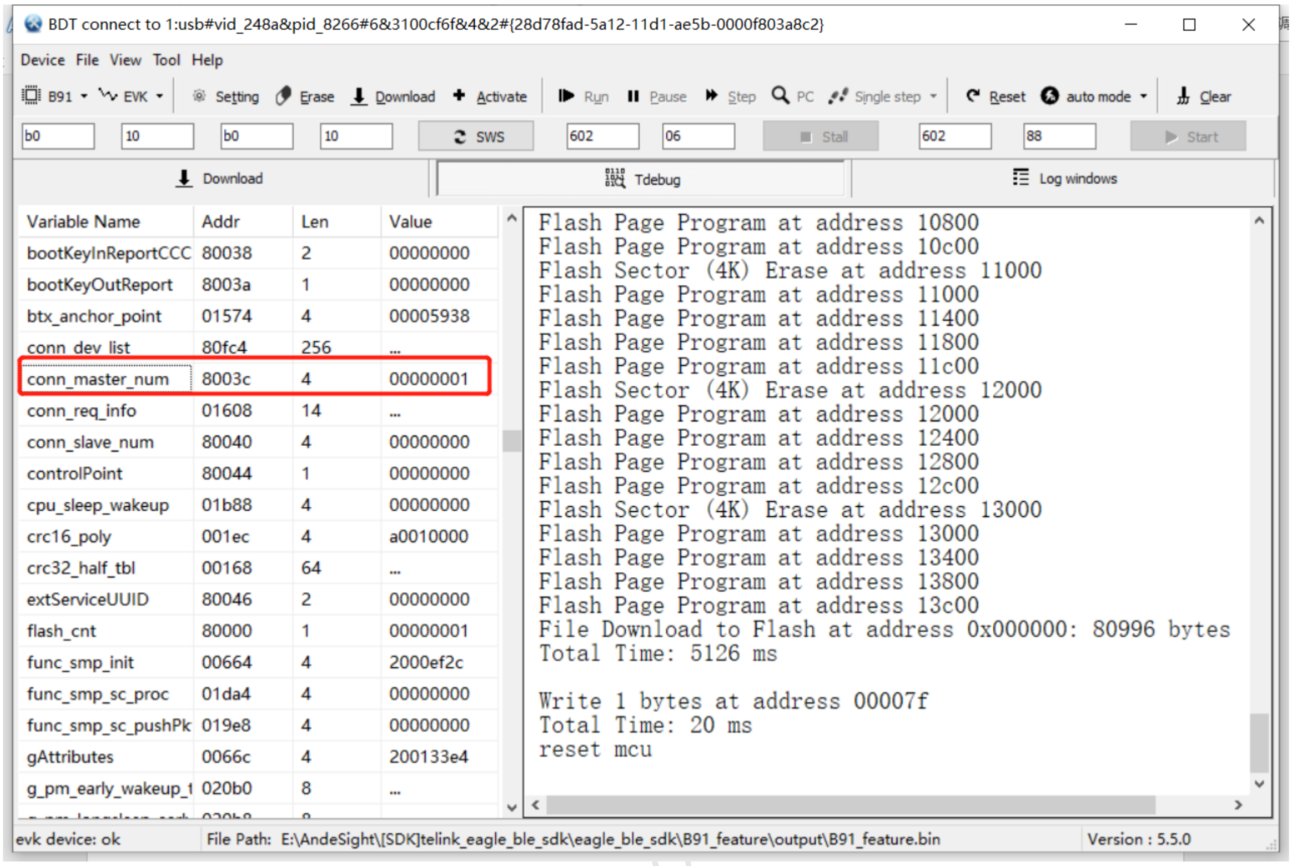


Figure 10.2: conn_master_num=1 in Tdebug

查看另一个开发板的 conn_slave_num 变量的值，其值也为 1：

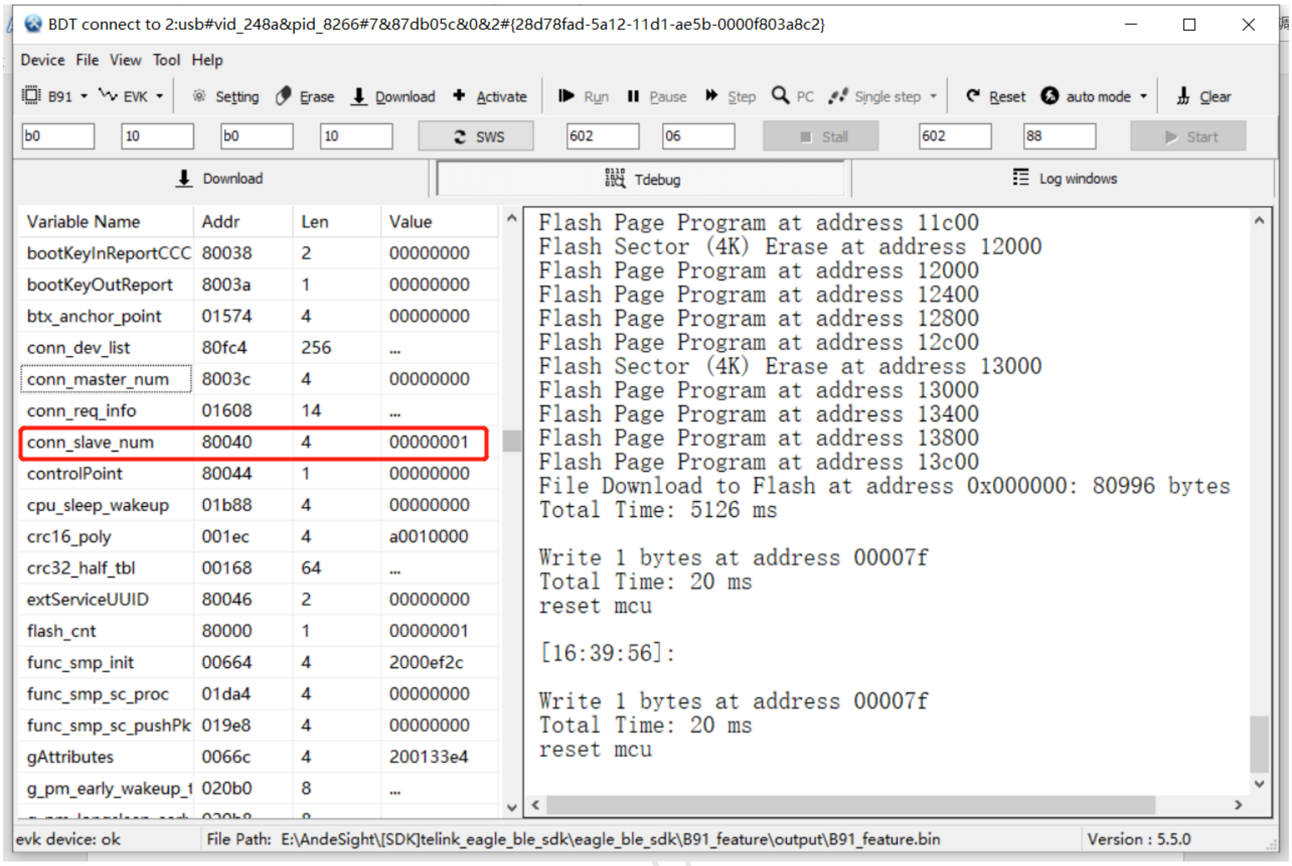


Figure 10.3: conn_slave_num=1 in Tdebug

代表二者连接成功。

10.2 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 Telink B91 BLE Multiple Connection SDK 为例，应用层代码在 eagle_ble_sdk/vendor/B91_feature/feature_2M_coded_phy 下，需要修改 eagle_ble_sdk/vendor/B91_feature/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) Master 建立连接后，每隔 1s 给各 Slave 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) Slave 建立连接后，每隔 1s 给各个 Master 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 (TXs=LE Coded, RXs=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 (TXs=LE Coded, RXs=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=14'688 (+19) | 557.306 260 625 (+570.000...	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 10.4: 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 (TXs=LE 2M, RXs=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 (TXs=LE 2M, RXs=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) | 577.287 549 625 (+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 10.5: 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 (TXs=LE 1M, RXs=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 (TXs=LE 1M, RXs=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 (TXs=LE 1M, RXs=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=2'353 (+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=2'353 (+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 10.6: feature_2M_coded_phy 1M Req

10.3 feature_gatt_api

- 功能：BLE GATT 指令功能演示与 API 使用方法。Telink BLE Multiple Connection 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 Telink B91 BLE Multiple Connection SDK 为例，应用层代码在 eagle_ble_sdk/vendor/B91_feature/feature_gatt_api 下，需要修改 eagle_ble_sdk/vendor/B91_feature/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 Master）上的 SW4 按键，触发连接，通过抓包可以看到每隔 2s，Master 发送一个测试指令，Slave 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 10.7: 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 10.8: 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, Sourc...	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, Sourc...	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, Sourc...	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 Find By Type Value Request Packet (1 - Max Handle, PnP ID, Sourc...	7 bytes ...	Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01

Figure 10.9: 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 10.10: 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	Empty LE 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 10.11: GATT API test TEST_READ_BLOB_REQ

10.4 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 Telink B91 BLE Multiple Connection SDK 为例，应用层代码在 eagle_ble_sdk/vendor/B91_feature/feature_ll_more_data 下，需要修改 eagle_ble_sdk/vendor/B91_feature/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 Master）的 SW4 按键，触发连接。连接成功后，红灯点亮（如果连接多个 Slave，分别点亮：红色、白色、绿色、蓝色灯）。按 Master 的按

键 SW3，可以从抓包中看到 Master 不断向 Slave 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：

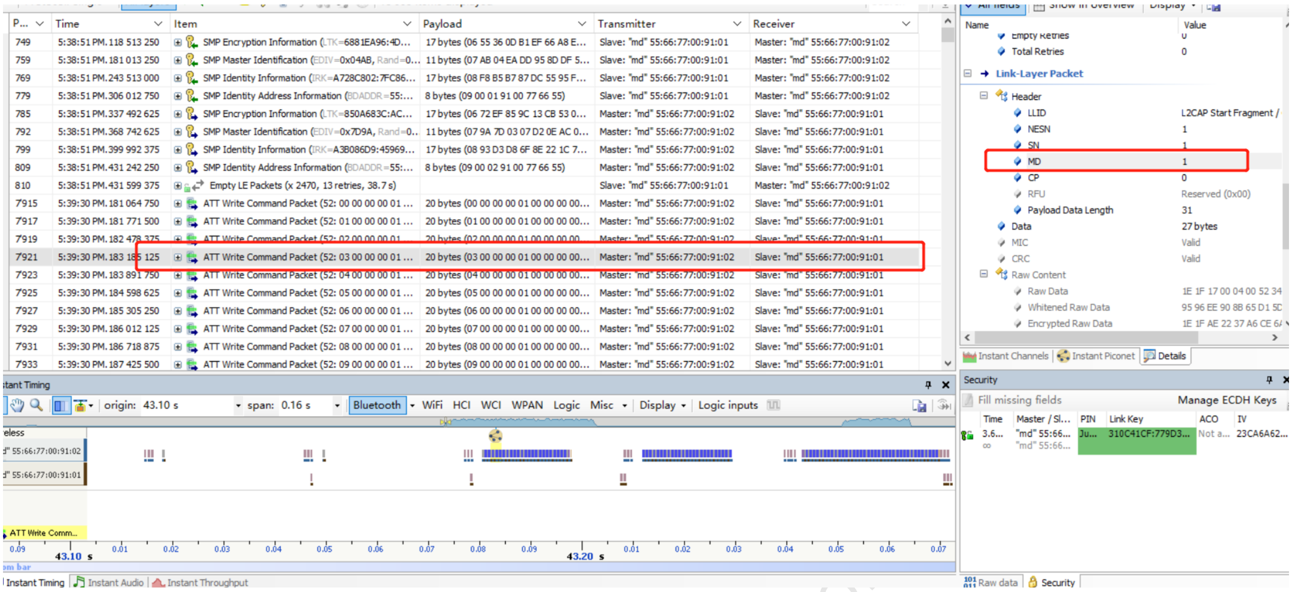


Figure 10.12：feature_ll_more_data WriteCmd

按 Master 的按键 SW2，停止发送。

按 Slave 的 SW3，可以从抓包中看到 Slave 不断向 Master 发送 Notify 包，其中的 MD flag 被置为 1：

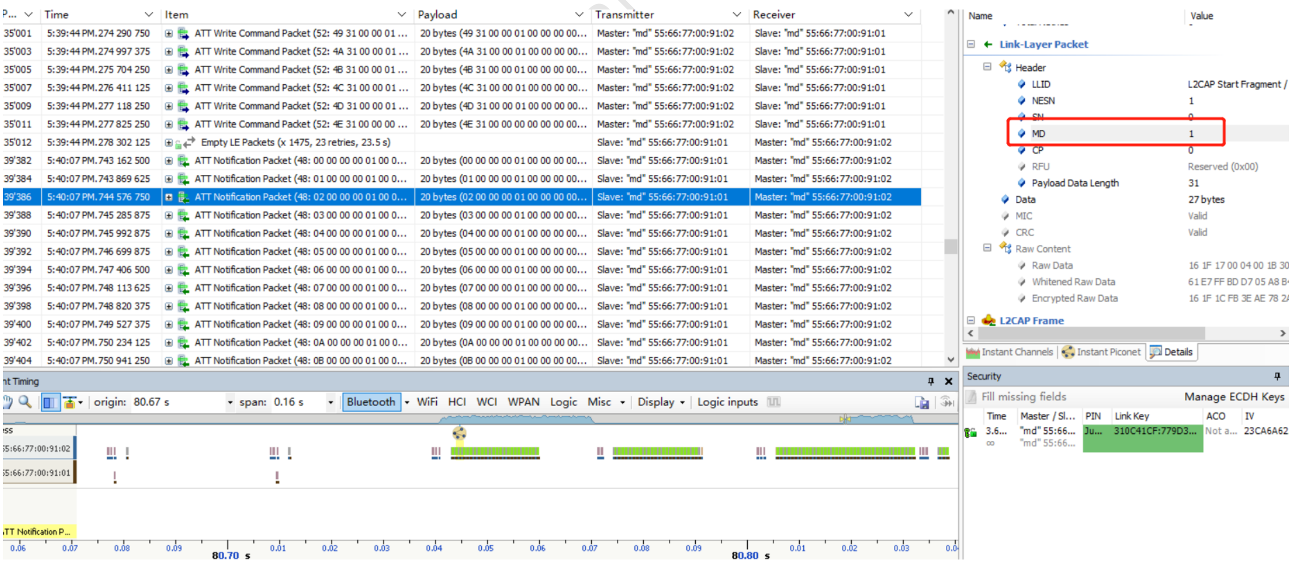


Figure 10.13：feature_ll_more_data Notify

按 Slave 的按键 SW2，停止发送。

10.5 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 Telink B91 BLE Multiple Connection SDK 为例，应用层代码在 eagle_ble_sdk/vendor/B91_feature/feature_dle 下，需要修改 eagle_ble_sdk/vendor/B91_feature/feature_config.h 中的定义：

```
#define FEATURE_TEST_MODE                TEST_LL_DLE
```

来激活这部分代码，做 DLE 和 MTU 的演示。

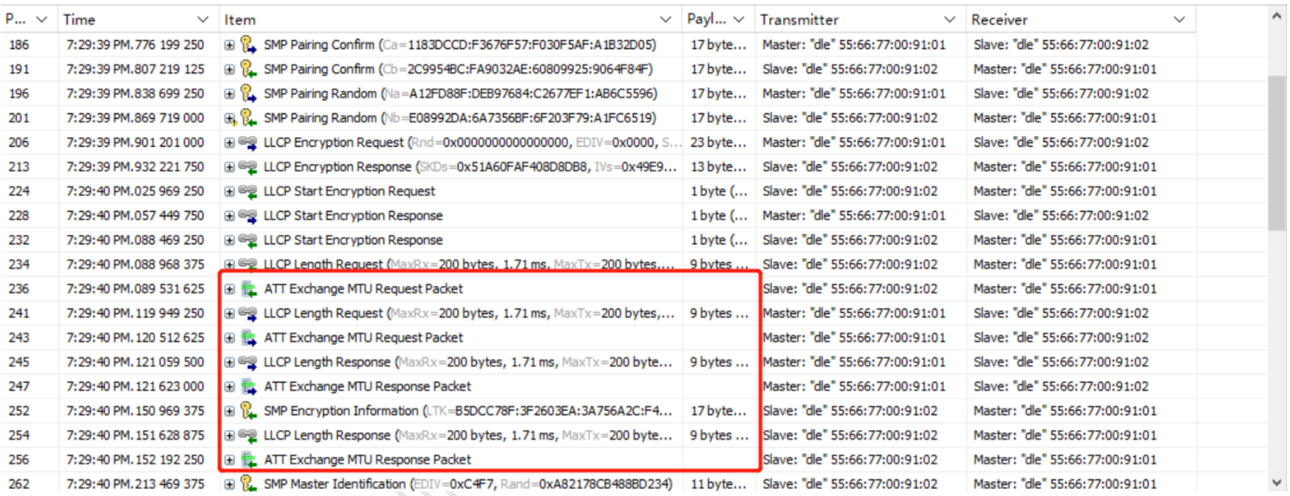
通过修改 DLE_LENGTH_SELECT 的定义来修改 DataLength，演示内容为：

(1) 通过两个按键同时按下触发测试。

(2) Master 触发测试后，每个连接，每隔 1s 发送一个 WriteCmd。

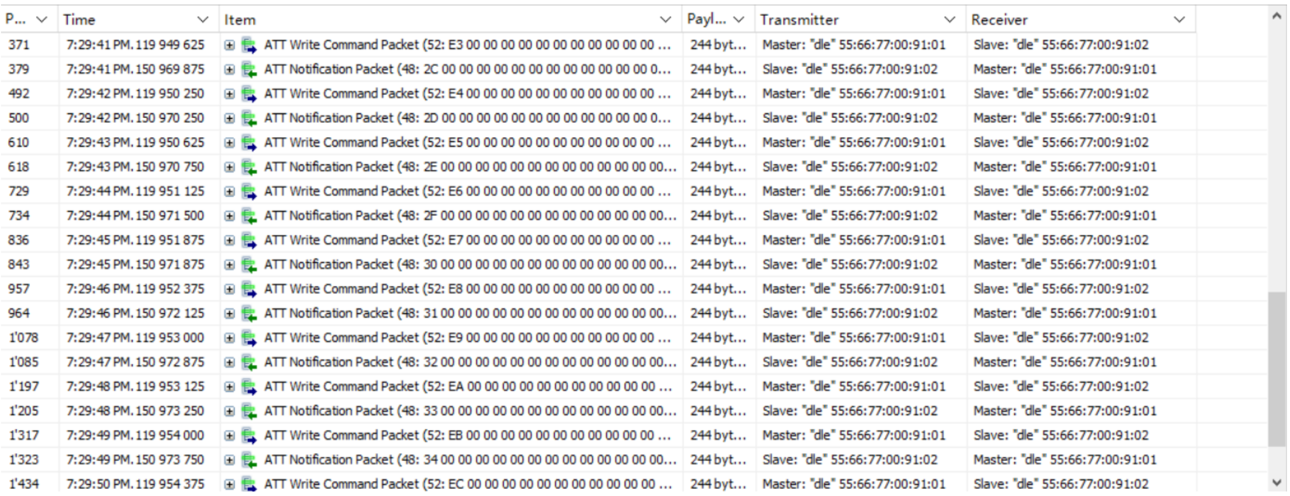
(3) Slave 触发测试后，每个连接，每隔 1s 发送一个 Notify。

抓包如下：



P...	Time	Item	Payl...	Transmitter	Receiver
186	7:29:39 PM.776 199 250	SMP Pairing Confirm (Ca=1183DCCD:F3676F57:F030F5AF: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=A12FD88F:DEB97684:C2677EF1: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=E08992DA: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 (SKDs=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 byte...	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 byte...	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=0xA82178CB488BD234)	11 byte...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01

Figure 10.14: feature_dle DLE & MTU Exchange



P...	Time	Item	Payl...	Transmitter	Receiver
371	7:29:41 PM.119 949 625	ATT Write Command Packet (52: E3 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
379	7:29:41 PM.150 969 875	ATT Notification Packet (48: 2C 00 00 00 00 00 00 00 00 00 00 00 0...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
492	7:29:42 PM.119 950 250	ATT Write Command Packet (52: E4 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
500	7:29:42 PM.150 970 250	ATT Notification Packet (48: 2D 00 00 00 00 00 00 00 00 00 00 00 0...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
610	7:29:43 PM.119 950 625	ATT Write Command Packet (52: E5 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
618	7:29:43 PM.150 970 750	ATT Notification Packet (48: 2E 00 00 00 00 00 00 00 00 00 00 00 00...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
729	7:29:44 PM.119 951 125	ATT Write Command Packet (52: E6 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
734	7:29:44 PM.150 971 500	ATT Notification Packet (48: 2F 00 00 00 00 00 00 00 00 00 00 00 00...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
836	7:29:45 PM.119 951 875	ATT Write Command Packet (52: E7 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
843	7:29:45 PM.150 971 875	ATT Notification Packet (48: 30 00 00 00 00 00 00 00 00 00 00 00 00...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
957	7:29:46 PM.119 952 375	ATT Write Command Packet (52: E8 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
964	7:29:46 PM.150 972 125	ATT Notification Packet (48: 31 00 00 00 00 00 00 00 00 00 00 00 00...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
1'078	7:29:47 PM.119 953 000	ATT Write Command Packet (52: E9 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
1'085	7:29:47 PM.150 972 875	ATT Notification Packet (48: 32 00 00 00 00 00 00 00 00 00 00 00 00...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
1'197	7:29:48 PM.119 953 125	ATT Write Command Packet (52: EA 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
1'205	7:29:48 PM.150 973 250	ATT Notification Packet (48: 33 00 00 00 00 00 00 00 00 00 00 00 00...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
1'317	7:29:49 PM.119 954 000	ATT Write Command Packet (52: EB 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
1'323	7:29:49 PM.150 973 750	ATT Notification Packet (48: 34 00 00 00 00 00 00 00 00 00 00 00 00...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
1'434	7:29:50 PM.119 954 375	ATT Write Command Packet (52: EC 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02

Figure 10.15: Master 和 Slave 分别做 WriteCmd 和 Notify

由于 DLE 小于 MTU，可以看到分包的效果：

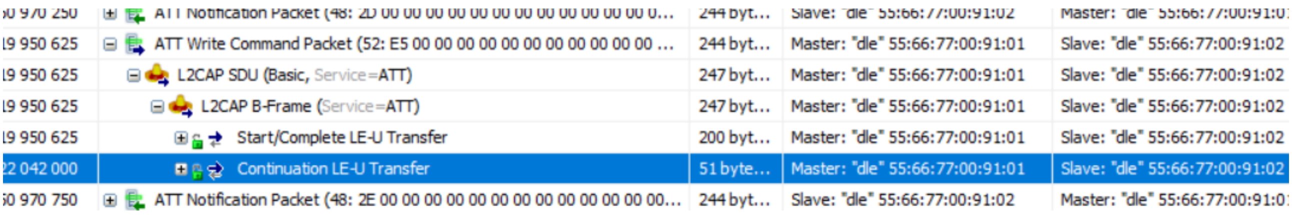


Figure 10.16: 分包发送

10.6 feature_smp

- 功能：BLE SMP (Security Manager Protocol) 功能演示。
- 主要硬件：B91 开发板 x 2

以 Telink B91 BLE Multiple Connection SDK 为例，应用层代码在 eagle_ble_sdk/vendor/B91_feature/feature_smp 下，需要修改 eagle_ble_sdk/vendor/B91_feature/feature_config.h 中的定义：

```
#define FEATURE_TEST_MODE                TEST_SMP
```

来激活这部分代码，默认 Slave 和 Master 的 SMP 功能均不使能。

说明：

(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_config.h 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。

(2) 配置 SMP 相关参数时，如果 Master 和 Slave 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 _master/_slave 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：

a) 绿色：ON：Master connected；OFF: Master disconnected.

b) 红色：ON：Slave connected；OFF: Slave disconnected.

c) 蓝色：ON：Pair Succeeded；OFF：Disconnected，没走 Pair 流程也不亮。

d) 白色：ON：Encryption succeeded；OFF：Disconnected.

(4) 为方便用户调试，代码中默认使用 GPIO 模拟串口的 DEBUG 方式输出 Log，需将 PA0(Tx) 接到显示设备的输入管脚上，波特率配置为 115200。

(5) **提示：** Security Connections 的配对方式，要求 MTU>=65。

10.6.1 Slave 和 Master 均不使能 SMP

关闭 Slave 和 Master 的 SMP 功能的演示，需在 feature_smp/app_config.h 中定义（默认）：

```
#define SMP_TEST_MODE            SMP_TEST_NOT_SUPPORT
```

编译 B91_feature，分别烧录 /eagle_ble_sdk/B91_feature/output/B91_feature.bin 到两个开发板中。按其中一个开发板（作为 Master）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：

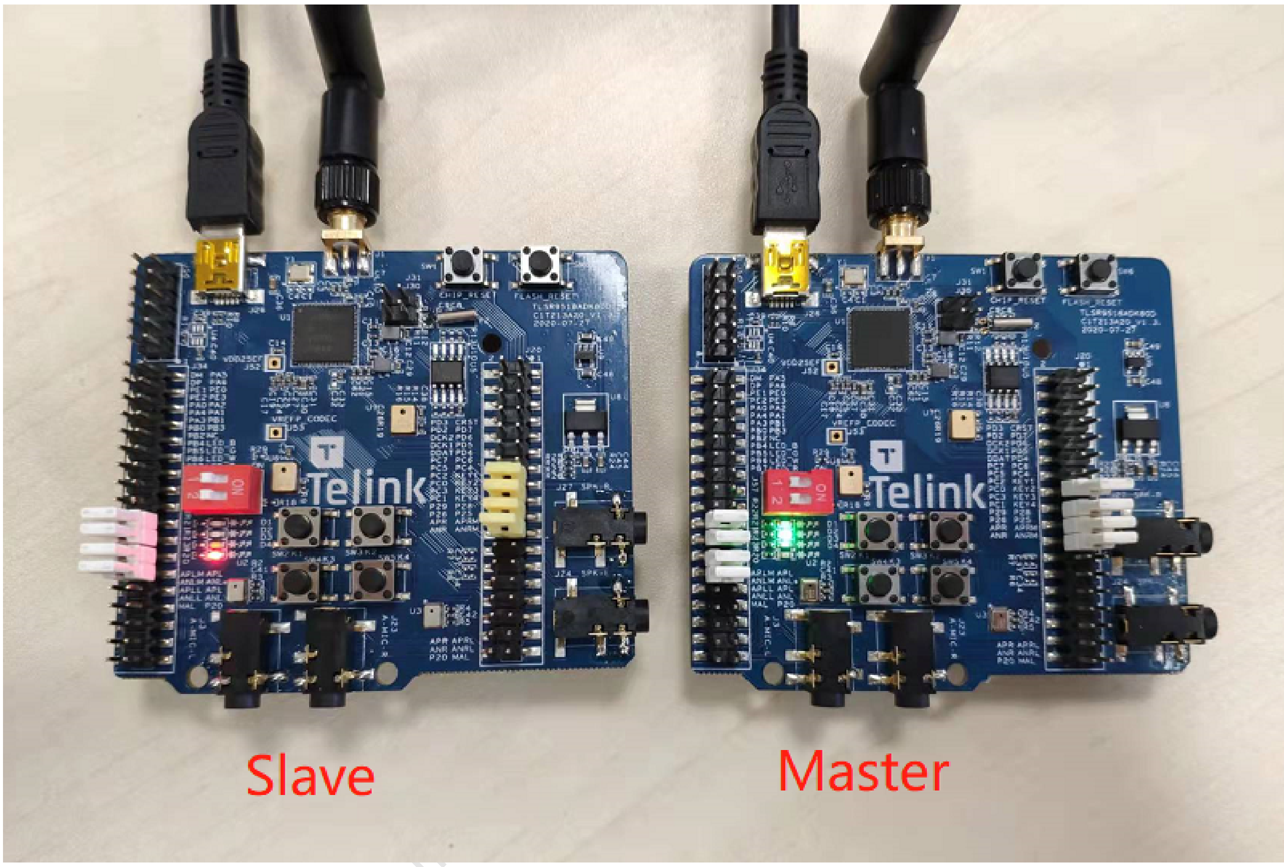


Figure 10.17: SMP 失能时 Slave 和 Master 连接成功

抓包如下：

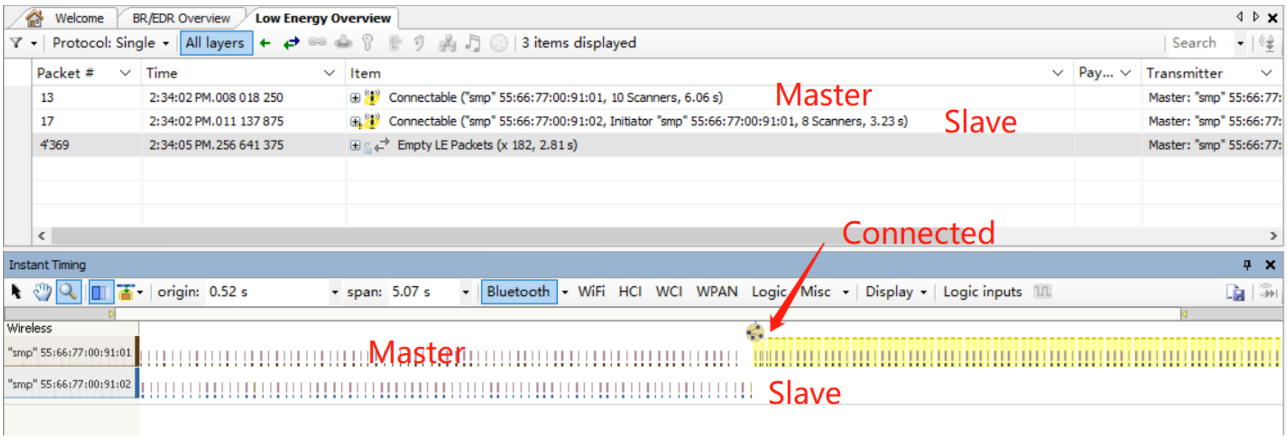


Figure 10.18: SMP 失能时 Slave 和 Master 连接成功的抓包

二者先分别发广播，建立连接后，作为 Master 的设备停止发广播。

说明：

初始化中将设备的 appMaxSlaveNum 设置为 1，当 currentMaxSlaveNum 与 appMaxSlaveNum 相等时，将停止广播。如果希望建立连接后，仍然广播，修改初始化中的以下参数：

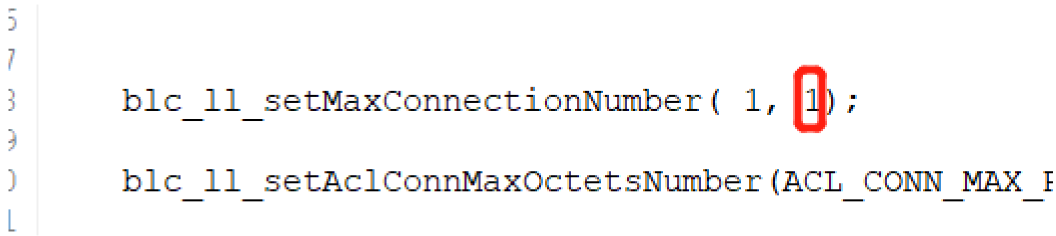


Figure 10.19: 修改 appMaxSlaveNum

则抓包如下：

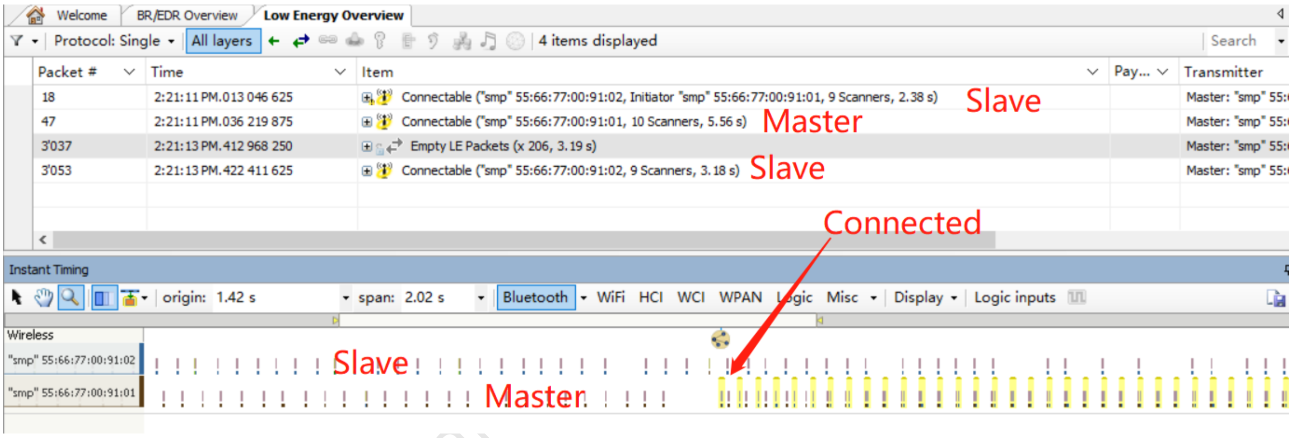


Figure 10.20: appMaxSlaveNum>1 连接成功的抓包

10.6.2 仅使能 Slave SMP

仅 Slave 使能 SMP 功能时的演示。该演示需在 feature_smp/app_config.h 中定义：

```
#define SMP_TEST_MODE            SMP_TEST_ONLY_SLAVE
```

编译 B91_feature，分别烧录 /eagle_ble_sdk/B91_feature/output/B91_feature.bin 到两个开发板中。按其中一个开发板（作为 Master）的按键 SW4（启动连接），从灯效上看，和双方均不使能 SMP 一样。但通过抓包可以看到，这里多了建立连接后 Slave 发送 Security Request，但由于 Master 不支持 SMP，Mater 并不会做回复，如下图：

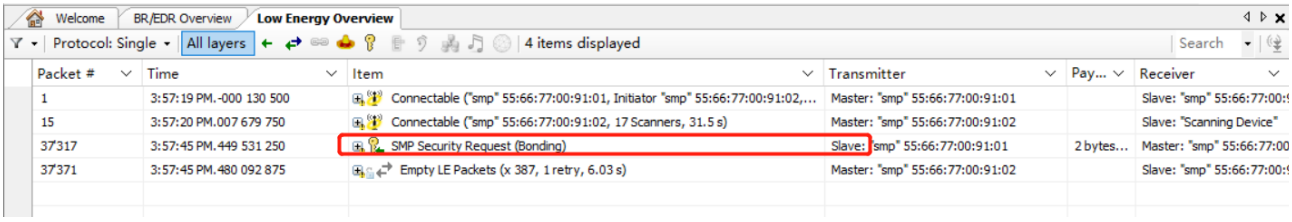


Figure 10.21: 仅 Slave SMP 的抓包

说明：

如果用户不希望 Slave 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 Slave（只有 Slave 会发 Security Request），因此不存在后缀为 _master/_slave 的相应 API。

10.6.3 仅使能 Master SMP

仅 Master 使能 SMP 功能时的演示。该演示需在 feature_smp/app_config.h 中定义：

```
#define SMP_TEST_MODE            SMP_TEST_ONLY_MASTER
```

编译 B91_feature，分别烧录 /eagle_ble_sdk/B91_feature/output/B91_feature.bin 到两个开发板中。按其中一个开发板（作为 Master）的按键 SW4（启动连接），从灯效上看，和双方均不使能 SMP 一样。但通过抓包可以看到，这里多了建立连接后 Master 发送 Pairing Request，但由于 Slave 不支持 SMP，Slave 回复 Pairing Failed，Error Code 对应到 SDK 中的定义是 PAIRING_FAIL_REASON_PAIRING_NOT_SUPPORTED，抓包如下图：

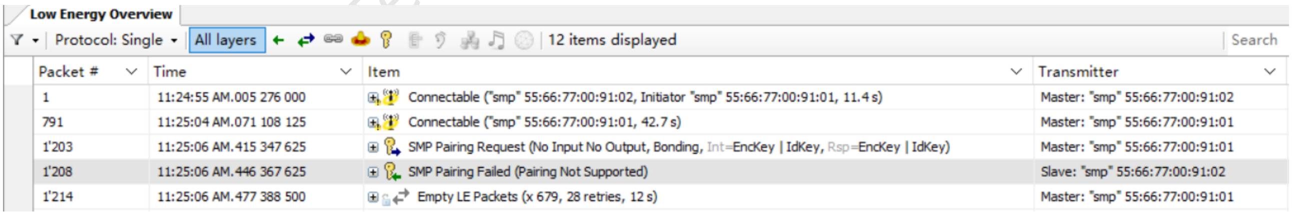


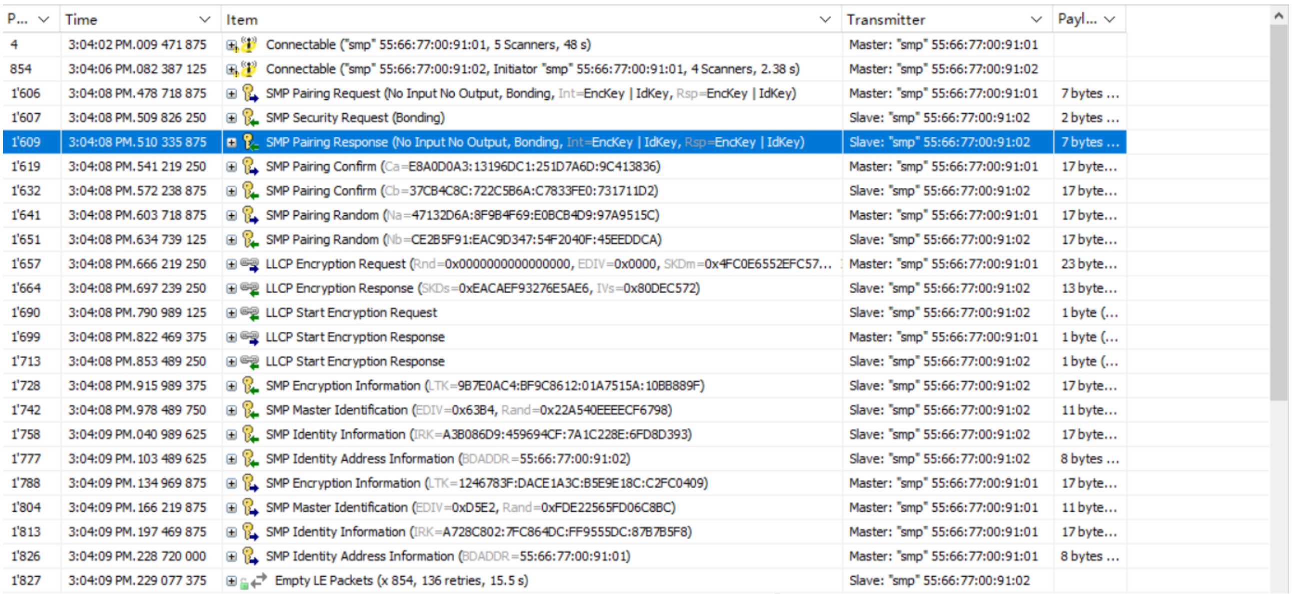
Figure 10.22: 仅 Master SMP 的抓包

10.6.4 Legacy Just Works

Master 和 Slave 使能 SMP 功能，以 Legacy Just Works 方式配对的演示。该演示需在 feature_smp/app_config.h 中定义：

```
#define SMP_TEST_MODE            SMP_TEST_LEGACY_JUST_WORKS
```

编译 B91_feature，分别烧录 /eagle_ble_sdk/B91_feature/output/B91_feature.bin 到两个开发板中。按其中一个开发板（作为 Master）的按键 SW4（启动连接），连接成功后，可以看到 Master 亮绿色、白色、蓝色灯，Slave 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=EncKey | IdKey, Rsp=EncKey | 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=EncKey | IdKey, Rsp=EncKey | IdKey)	Slave: "smp" 55:66:77:00:91:02	7 bytes ...
1'619	3:04:08 PM.541 219 250	SMP Pairing Confirm (Ca=E8A0D0A3:13196DC1:251D7A6D: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:C7833FE0:731711D2)	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=CE2B5F91:EAC9D347:54F2040F:45EEDDCA)	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=0xEACAEF93276E5AE6, 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=9B7E0AC4:BF9C8612:01A7515A:10BB889F)	Slave: "smp" 55:66:77:00:91:02	17 byte...
1'742	3:04:08 PM.978 489 750	SMP Master Identification (EDIV=0x63B4, Rand=0x22A540EEEECF6798)	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 10.23: Legacy Just Works 的抓包

此时按 Master 或 Slave 开发板上的 SW1 按键给开发板重新上电，由于 Master 在扫描到已经 Pair 过的 Slave 设备会自动回连（参考演示代码中 master_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 Master 和 Slave 重新上电后回连跳过 Pair 流程，只需要 Master 或 Slave 任意一方的 Bonding Flag 置位为 0 即可，这里给出两种方法：

- 方法 1：在初始化配置 Security Parameters 的同时将 Bonding Flag 置位为 0。以 Slave 为例（默认注释掉）：

```
blc_smp_setSecurityParameters_slave(Non_Bondable_Mode, 0, LE_Legacy_Pairing,    0, 0,
↪  IO_CAPABILITY_NO_INPUT_NO_OUTPUT);
```

这里设置了 Slave 端的 bonding_mode 为 Non-Bondable mode，其他设置为 Just Works 默认配置，Master 端同理。如果不调用该 API，默认会在 blc_gap_init() 中对 Slave 和 Master 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;
```

调用该 API 之后，在 Pair 时，会看到 Slave 回复的 Pairing Response 中的 Bonding Flags=0，这样 Master 和 Slave 就都不会做 Bonding，也就是不会将配对信息存在 Flash 中。当 Master 或 Slave 设备重启之后，他们仍然是"全新的"，并不"认识"彼此：

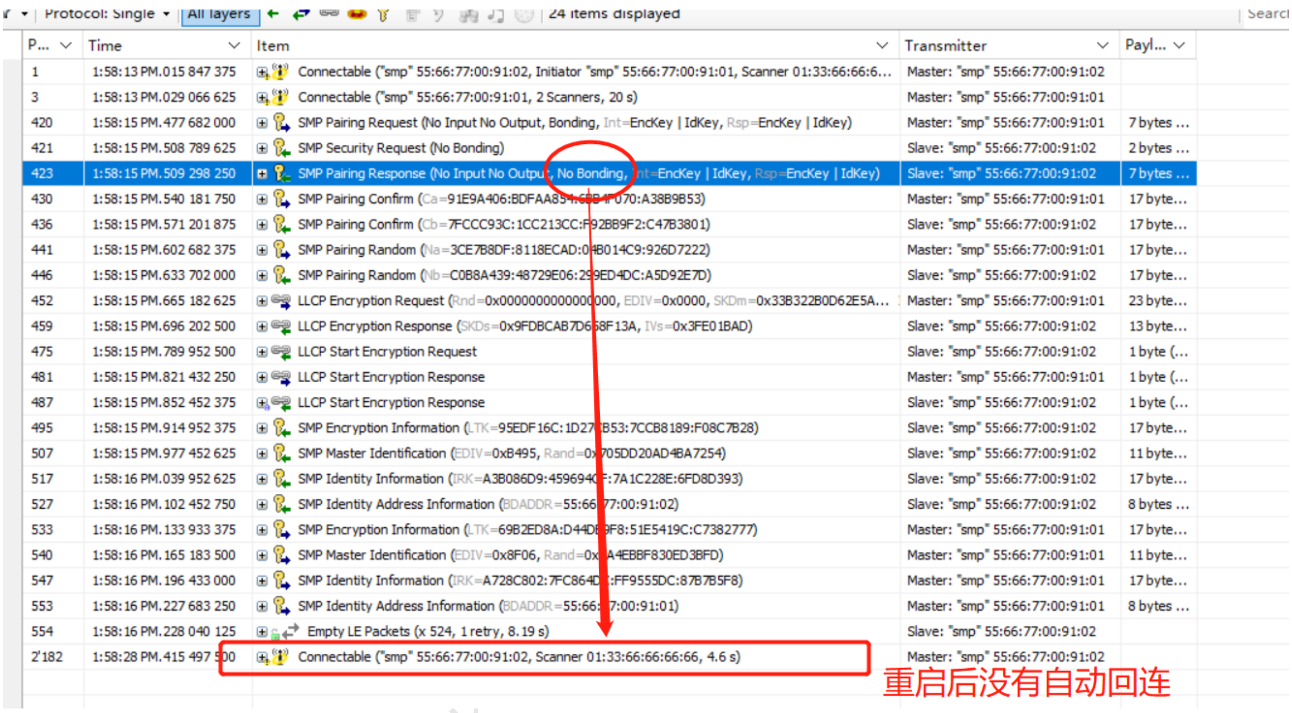
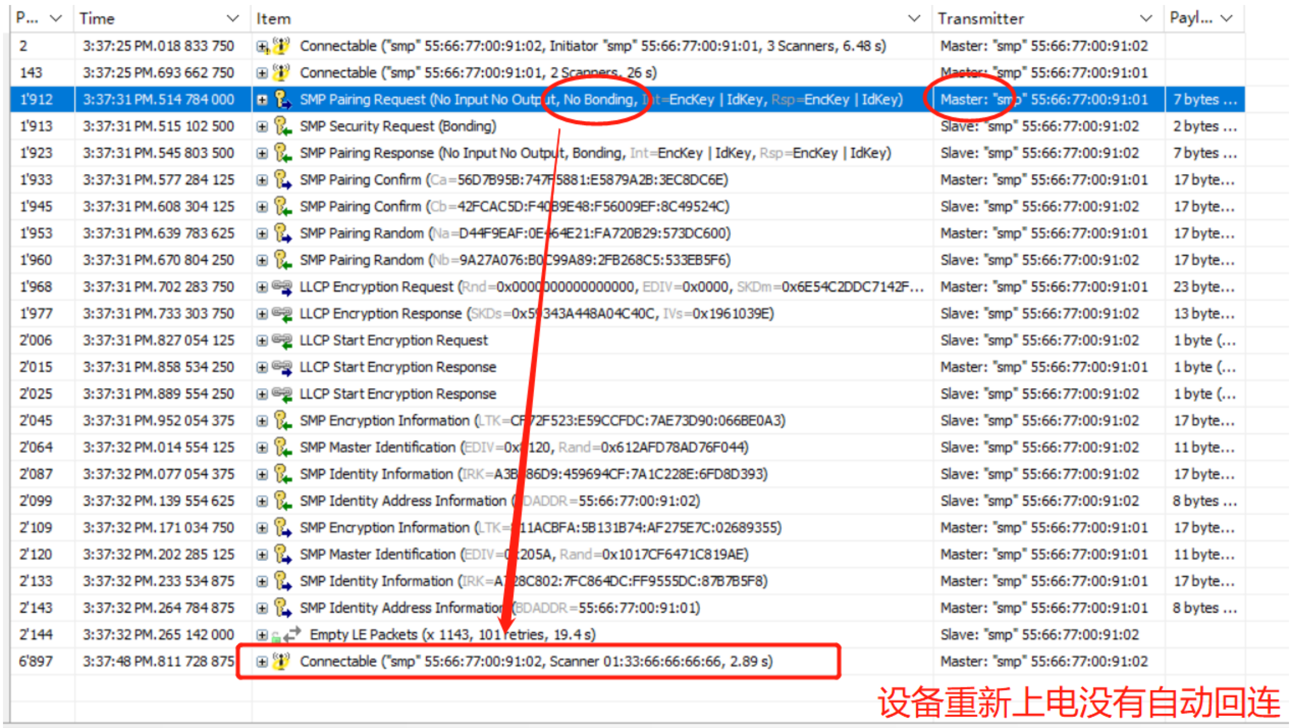


Figure 10.24: Legacy Just Works Slave NoBonding 的抓包

- 方法 2：在初始化配置 Security Parameters 的同时单独配置 Bonding Mode。以 Master 为例：

```
blc_smp_setBondingMode_master(Non_Bondable_Mode);
```

这里设置了 Master 端的 bonding_mode 为 Non-Bondable mode，Slave 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 Outp[illegible]t, No Bonding, I[illegible]t=EncKey | IdKey, Rsp=EncKey | IdKey)	Master: "sm[illegible]" 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=EncKey | IdKey, Rsp=EncKey | IdKey)	Slave: "smp" 55:66:77:00:91:02	7 bytes ...
1'933	3:37:31 PM.577 284 125	SMP Pairing Confirm (Ca=56D7B95B:747F5881:E5879A2B:3EC8DC6E)	Master: "smp" 55:66:77:00:91:01	17 byte...
1'945	3:37:31 PM.608 304 125	SMP Pairing Confirm (Cb=42FCAC5D:F40B9E48:F56009EF:8C49524C)	Slave: "smp" 55:66:77:00:91:02	17 byte...
1'953	3:37:31 PM.639 783 625	SMP Pairing Random (Na=D44F9EAF:0E[illegible]64E21:FA720B29:573DC600)	Master: "smp" 55:66:77:00:91:01	17 byte...
1'960	3:37:31 PM.670 804 250	SMP Pairing Random (Nb=9A27A076:B0[illegible]C99A89:2FB268C5:533EB5F6)	Slave: "smp" 55:66:77:00:91:02	17 byte...
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 byte...
1'977	3:37:31 PM.733 303 750	LLCP Encryption Response (SKDs=0x5[illegible]343A448A04C40C, IVs=0x1961039E)	Slave: "smp" 55:66:77:00:91:02	13 byte...
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=CF[illegible]72F523:E59CCFDC:7AE73D90:066BE0A3)	Slave: "smp" 55:66:77:00:91:02	17 byte...
2'064	3:37:32 PM.014 554 125	SMP Master Identification (EDIV=0x[illegible]120, Rand=0x612AFD78AD76F044)	Slave: "smp" 55:66:77:00:91:02	11 byte...
2'087	3:37:32 PM.077 054 375	SMP Identity Information (IRK=A3B[illegible]86D9:459694CF:7A1C228E:6FD8D393)	Slave: "smp" 55:66:77:00:91:02	17 byte...
2'099	3:37:32 PM.139 554 625	SMP Identity Address Information ([illegible]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=[illegible]11ACBFA:5B131B74:AF275E7C:02689355)	Master: "smp" 55:66:77:00:91:01	17 byte...
2'120	3:37:32 PM.202 285 125	SMP Master Identification (EDIV=0[illegible]205A, Rand=0x1017CF6471C819AE)	Master: "smp" 55:66:77:00:91:01	11 byte...
2'133	3:37:32 PM.233 534 875	SMP Identity Information (IRK=A[illegible]28C802:7FC864DC:FF9555DC:87B7B5F8)	Master: "smp" 55:66:77:00:91:01	17 byte...
2'143	3:37:32 PM.264 784 875	SMP Identity Address Informatio[illegible] (BDADDR=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	

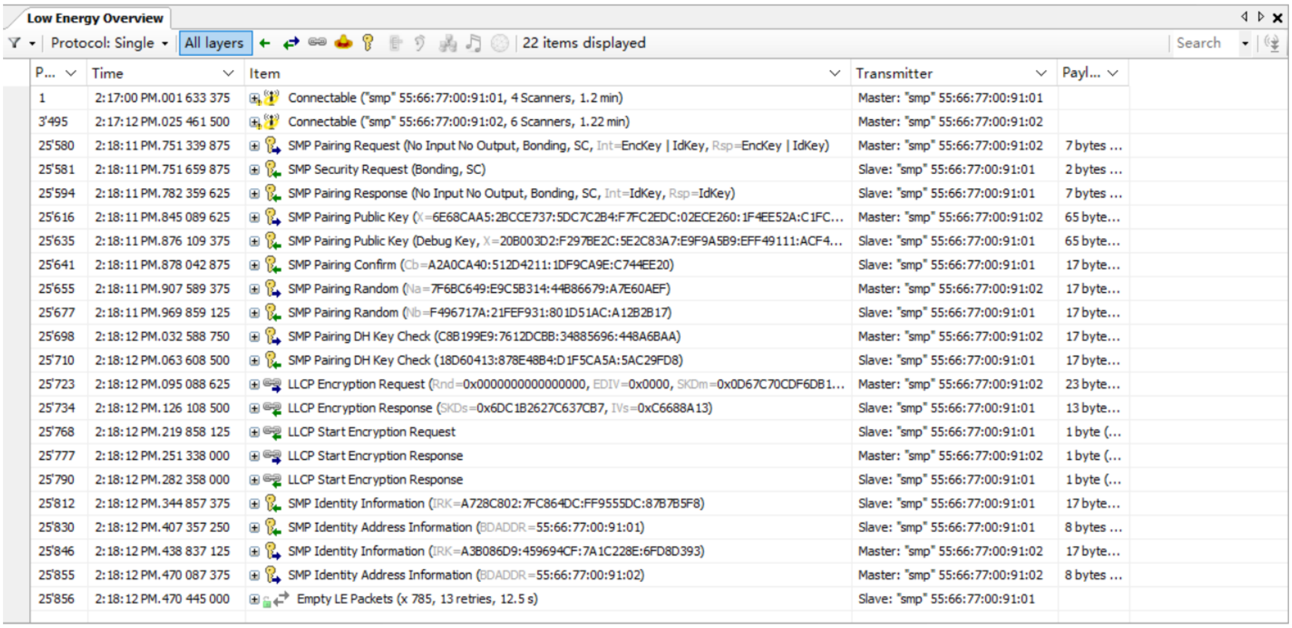
Figure 10.25: Legacy Just Works Master NoBonding 的抓包

10.6.5 Secure Connections Just Works

Master 和 Slave 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_config.h 中定义：

```
#define SMP_TEST_MODE          SMP_TEST_SC_JUST_WORKS
```

编译 B91_feature，分别烧录 /eagle_ble_sdk/B91_feature/output/B91_feature.bin 到两个开发板中。按其中一个开发板（作为 Master）的按键 SW4（启动连接），连接成功后，可以看到 Master 亮绿色、白色、蓝色灯，Slave 亮红色、白色、蓝色灯，代表二者 Pair 成功，抓包如下：



Low Energy Overview

Protocol: Single | All layers | 22 items displayed | Search

P...	Time	Item	Transmitter	Payl...
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	7 bytes ...
25'581	2:18:11 PM.751 659 875	SMP Security Request (Bonding, SC)	Slave: "smp" 55:66:77:00:91:01	2 bytes ...
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	7 bytes ...
25'616	2:18:11 PM.845 089 625	SMP Pairing Public Key (X=6E68CAA5:2BCCE737:5DC7C2B4:F7FC2EDC:02ECE260:1F4EE52A:C1FC...	Master: "smp" 55:66:77:00:91:02	65 byte...
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	65 byte...
25'641	2:18:11 PM.878 042 875	SMP Pairing Confirm (Cb=A2A0CA40:512D4211:1DF9CA9E:C744EE20)	Slave: "smp" 55:66:77:00:91:01	17 byte...
25'655	2:18:11 PM.907 589 375	SMP Pairing Random (Na=7F6BC649:E9C5B314:44B86679:A7E60AEF)	Master: "smp" 55:66:77:00:91:02	17 byte...
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	17 byte...
25'698	2:18:12 PM.032 588 750	SMP Pairing DH Key Check (C8B199E9:7612DCBB:34885696:448A6BAA)	Master: "smp" 55:66:77:00:91:02	17 byte...
25'710	2:18:12 PM.063 608 500	SMP Pairing DH Key Check (18D60413:878E48B4:D1F5CA5A:5AC29FD8)	Slave: "smp" 55:66:77:00:91:01	17 byte...
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	23 byte...
25'734	2:18:12 PM.126 108 500	LLCP Encryption Response (SKDs=0x6DC1B2627C637CB7, IVs=0xC6688A13)	Slave: "smp" 55:66:77:00:91:01	13 byte...
25'768	2:18:12 PM.219 858 125	LLCP Start Encryption Request	Slave: "smp" 55:66:77:00:91:01	1 byte (...
25'777	2:18:12 PM.251 338 000	LLCP Start Encryption Response	Master: "smp" 55:66:77:00:91:02	1 byte (...
25'790	2:18:12 PM.282 358 000	LLCP Start Encryption Response	Slave: "smp" 55:66:77:00:91:01	1 byte (...
25'812	2:18:12 PM.344 857 375	SMP Identity Information (IRK=A728C802:7FC864DC:FF9555DC:87B7B5F8)	Slave: "smp" 55:66:77:00:91:01	17 byte...
25'830	2:18:12 PM.407 357 250	SMP Identity Address Information (BDADDR=55:66:77:00:91:01)	Slave: "smp" 55:66:77:00:91:01	8 bytes ...
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	17 byte...
25'855	2:18:12 PM.470 087 375	SMP Identity Address Information (BDADDR=55:66:77:00:91:02)	Master: "smp" 55:66:77:00:91:02	8 bytes ...
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 10.26: SC Just Works 的抓包

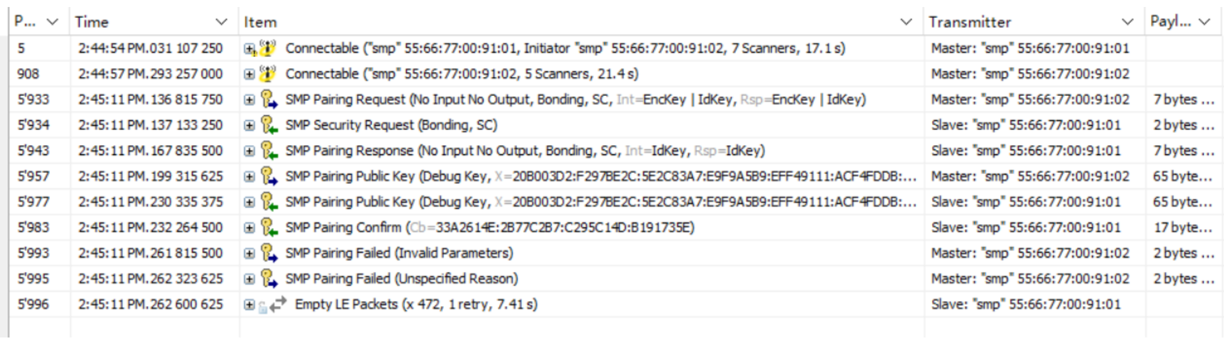
关于 Debug Mode 参考SMP 章节，用户可以根据需求配置为如下三种：

Table 10.1: Debug Mode 配置选项

Master	Slave	Description
debug mode disabled	debug mode enabled	演示代码默认配置
debug mode enabled	debug mode disabled	效果与演示代码类似
debug mode disabled	debug mode disabled	加密后的包抓包工具无法解析

注意：

> Slave debug mode enable + Master 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	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	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	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	SMP Security Request (Bonding, SC)	Slave: "smp" 55:66:77:00:91:01	2 bytes ...
5'943	2:45:11 PM.167 835 500	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	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	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	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	SMP Pairing Failed (Invalid Parameters)	Master: "smp" 55:66:77:00:91:02	2 bytes ...
5'995	2:45:11 PM.262 323 625	SMP Pairing Failed (Unspecified Reason)	Master: "smp" 55:66:77:00:91:02	2 bytes ...
5'996	2:45:11 PM.262 600 625	Empty LE Packets (x 472, 1 retry, 7.41 s)	Slave: "smp" 55:66:77:00:91:01	

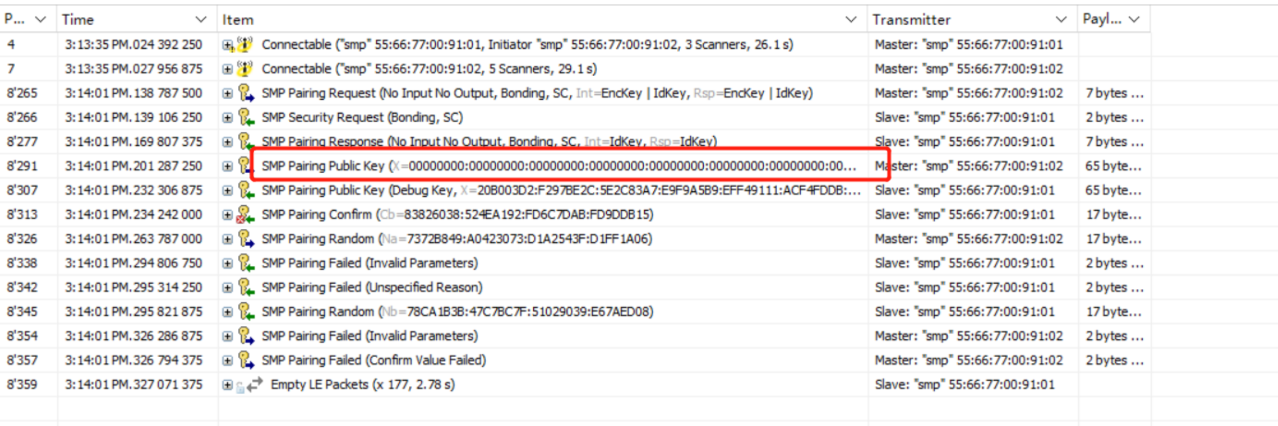
Figure 10.27: Both DebugMode Pair Failed 的抓包

重要提示！

> 在 Telink BLE Multiple Connection SDK V4.0.1.0 版本及版本前，需要调用以下 API（代码中默认注释掉）

```
void    blc_smp_setPairingMethods{_master/_slave}(pairing_methods_t  method);
```

设置 Secure Connection，以避免出现 Public Key 为全 0 的情况：



P...	Time	Item	Transmitter	Payl...
4	3:13:35 PM.024 392 250	Connectable ("smp" 55:66:77:00:91:01, Initiator "smp" 55:66:77:00:91:02, 3 Scanners, 26.1 s)	Master: "smp" 55:66:77:00:91:01	
7	3:13:35 PM.027 956 875	Connectable ("smp" 55:66:77:00:91:02, 5 Scanners, 29.1 s)	Master: "smp" 55:66:77:00:91:02	
8'265	3:14:01 PM.138 787 500	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 ...
8'266	3:14:01 PM.139 106 250	SMP Security Request (Bonding, SC)	Slave: "smp" 55:66:77:00:91:01	2 bytes ...
8'277	3:14:01 PM.169 807 375	SMP Pairing Response (No Input No Output, Bonding, SC, Int=IdKey, Rsp=IdKey)	Slave: "smp" 55:66:77:00:91:01	7 bytes ...
8'291	3:14:01 PM.201 287 250	SMP Pairing Public Key (X=00000000:00000000:00000000:00000000:00000000:00000000:00000000:00...	Master: "smp" 55:66:77:00:91:02	65 byte...
8'307	3:14:01 PM.232 306 875	SMP Pairing Public Key (Debug Key, X=20B003D2:F297BE2C:5E2C83A7:E9F9A5B9:EFF49111:ACF4FDDB:...	Slave: "smp" 55:66:77:00:91:01	65 byte...
8'313	3:14:01 PM.234 242 000	SMP Pairing Confirm (Cb=83826038:524EA192:FD6C7DAB:FD9DDB15)	Slave: "smp" 55:66:77:00:91:01	17 byte...
8'326	3:14:01 PM.263 787 000	SMP Pairing Random (Na=7372B849:A0423073:D1A2543F:D1FF1A06)	Master: "smp" 55:66:77:00:91:02	17 byte...
8'338	3:14:01 PM.294 806 750	SMP Pairing Failed (Invalid Parameters)	Slave: "smp" 55:66:77:00:91:01	2 bytes ...
8'342	3:14:01 PM.295 314 250	SMP Pairing Failed (Unspecified Reason)	Slave: "smp" 55:66:77:00:91:01	2 bytes ...
8'345	3:14:01 PM.295 821 875	SMP Pairing Random (Nb=78CA1B3B:47C7BC7F:51029039:E67AED08)	Slave: "smp" 55:66:77:00:91:01	17 byte...
8'354	3:14:01 PM.326 286 875	SMP Pairing Failed (Invalid Parameters)	Master: "smp" 55:66:77:00:91:02	2 bytes ...
8'357	3:14:01 PM.326 794 375	SMP Pairing Failed (Confirm Value Failed)	Master: "smp" 55:66:77:00:91:02	2 bytes ...
8'359	3:14:01 PM.327 071 375	Empty LE Packets (x 177, 2.78 s)	Slave: "smp" 55:66:77:00:91:01	

Figure 10.28: PublicKey=0 的抓包

10.6.6 Legacy Passkey Entry MDSI

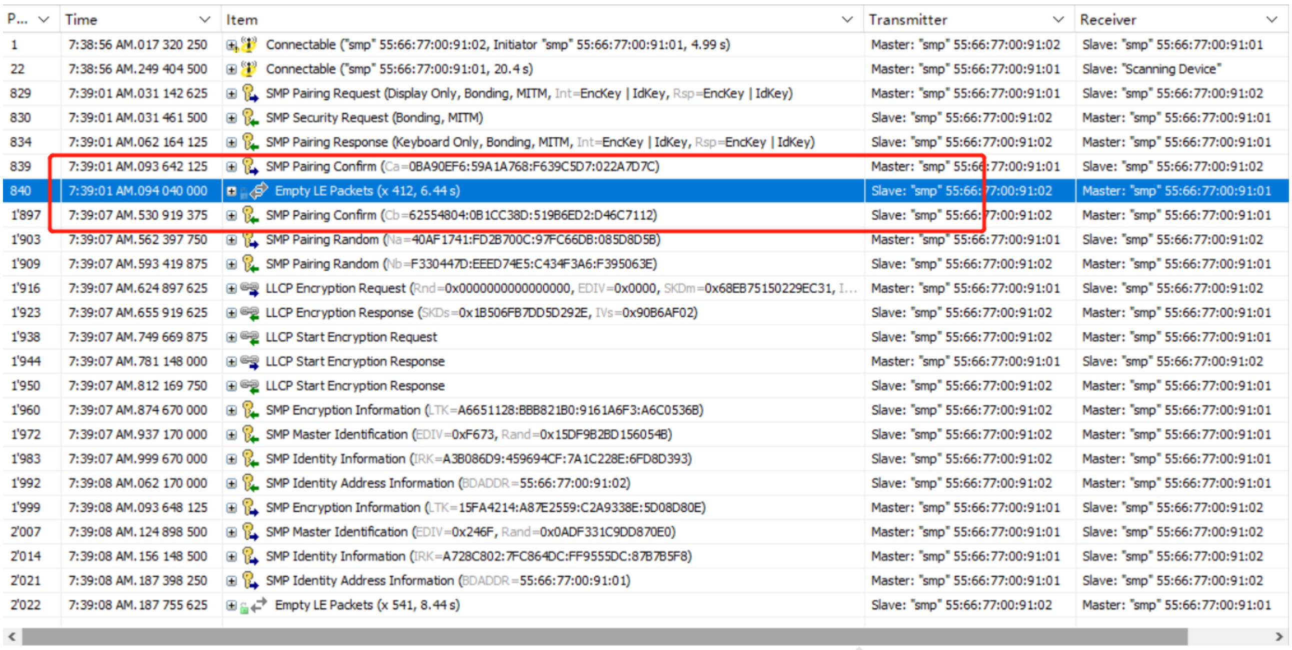
Master 和 Slave 使能 SMP 功能，以 Legacy Passkey Entry 配对方式，通过 MDSI 方法实现的演示。MDSI 代表 Master(Initiator) Displays Slave(Responder) Inputs，这里演示的 Input 是 Keyboard Input，使用该功能需在 feature_smp/app_config.h 中定义：

```
#define SMP_TEST_MODE         SMP_TEST_LEGACY_PASSKEY_ENTRY_MDSI
```

演示代码中提供了两种 Responder 输入 Passkey 的方法，这里演示直接设置默认 Passkey 为 123456 的方式，参考定义：

```
#define PASSKEY_ENTRY_METHOD         PASSKEY_ENTRY_BY_DEFAULT
```

编译 B91_feature，分别烧录 /eagle_ble_sdk/B91_feature/output/B91_feature.bin 到两个开发板中。按其中一个开发板（作为 Master）的按键 SW4（启动连接），连接成功后，可以看到 Master 亮绿色，Slave 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 Slave 输入 Passkey）：



P...	Time	Item	Transmitter	Receiver
1	7:38:56 AM.017 320 250	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	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	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	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	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	SMP Pairing Confirm (Ca=0BA90EF6:59A1A768: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	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	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	SMP Pairing Random (Na=40AF1741:FD2B700C:97FC66DB:085D8D5B)	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	SMP Pairing Random (Nb=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	LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, SKDm=0x68EB75150229EC31, 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	LLCP Encryption Response (SKDs=0x1B506FB7DD5D292E, 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	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	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	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	SMP Encryption Information (LTK=A6651128:BBB821B0:9161A6F3: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	SMP Master Identification (EDIV=0xF673, 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	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	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	SMP Encryption Information (LTK=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	SMP Master Identification (EDIV=0x246F, Rand=0x0ADF331C9DD870E0)	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	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	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	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 10.29: Legacy Passkey Entry MDSI 的抓包

代码中在初始化时使用 blc_smp_setDefaultPinCode() 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。

10.6.7 Legacy Passkey Entry MISD

Master 和 Slave 使能 SMP 功能，以 Legacy Passkey Entry 配对方式，通过 MISD 方法实现的演示。MISD 与上一个演示 MDSI 类似，只是显示和输入的设备角色互换了，MISD 代表 Master(Initiator) Inputs Slave(Responder) Displays。使用该功能需在 feature_smp/app_config.h 中定义：

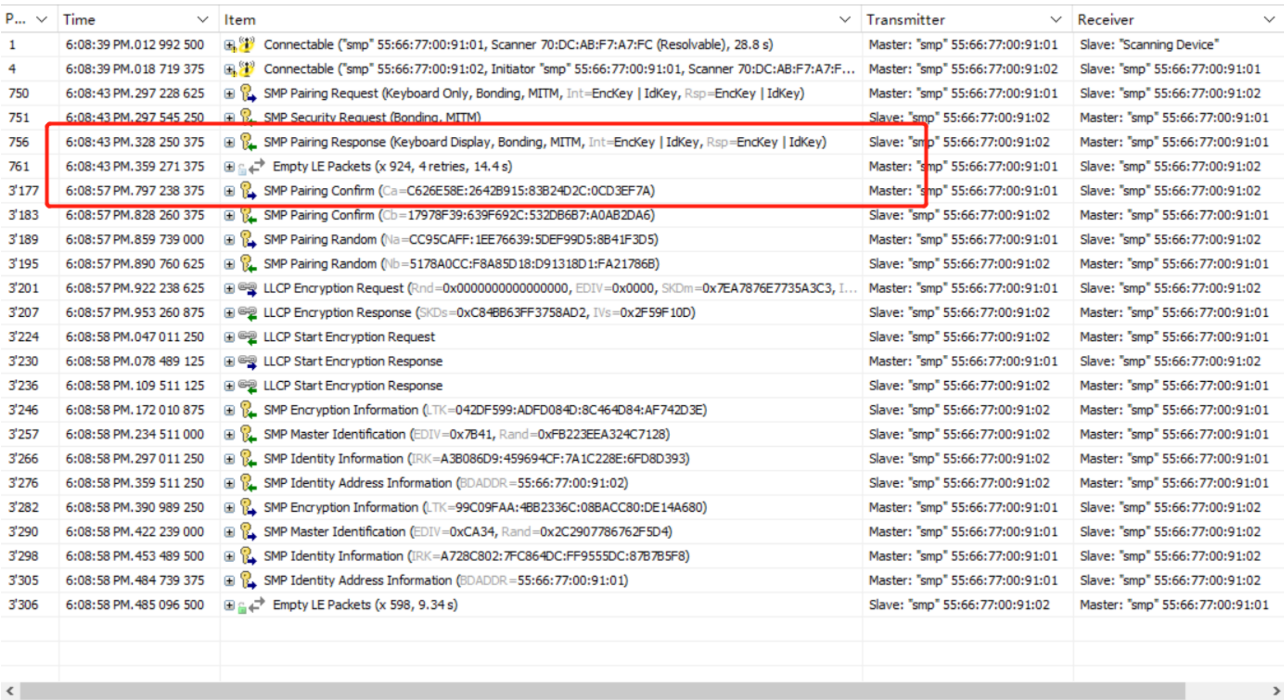
```
#define SMP_TEST_MODE        SMP_TEST_LEGACY_PASSKEY_ENTRY_MISD
```

演示代码中提供了两种 Initiator 输入 Passkey 的方法，这里演示通过 UART 输入 Passkey 的方法，更加贴合实际使用需求，参考定义：

```
#define PASSKEY_ENTRY_METHOD        PASSKEY_ENTRY_MANUALLY
```

编译 B91_feature，分别烧录 /eagle_ble_sdk/B91_feature/output/B91_feature.bin 到两个开发板中。给其中一个开发板（作为 Master）的 UART 口 PD2(Tx)，PD3(Rx) 接到电脑串口上，配置波特率 115200。按 Master

开发板的按键 SW4（启动连接），连接成功后，可以看到 Master 亮绿色灯，Slave 亮红色灯，此时 Slave 的 Log 输出 Display Pincode，将输出的数填入到 Master 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 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 Master 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:F...	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=EncKey | IdKey, Rsp=EncKey | 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=EncKey | IdKey, Rsp=EncKey | 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:0CD3EF7A)	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:532DB6B7: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=CC95CAFF: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=0xC84BB63FF3758AD2, 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=0x2C2907786762F5D4)	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 10.30: Legacy Passkey Entry MISD 的抓包

10.6.8 Legacy Passkey Entry Both Input

Master 和 Slave 使能 SMP 功能，以 Legacy Passkey Entry 配对方式，通过 Both Input 方法实现的演示。Both Input 的方式是由用户在双方输入相同的 Pincode 来实现的。使用该功能需在 feature_smp/app_config.h 中定义：

```
#define SMP_TEST_MODE        SMP_TEST_LEGACY_PASSKEY_ENTRY_BOTH_INPUT
```

演示代码中提供了两种输入 Passkey 的方法，这里演示直接设置默认 Passkey 为 123456 的方式，参考定义：

```
#define PASSKEY_ENTRY_METHOD          PASSKEY_ENTRY_BY_DEFAULT
```

编译 B91_feature，分别烧录 /eagle_ble_sdk/B91_feature/output/B91_feature.bin 到两个开发板中。按其中一个开发板（作为 Master）的按键 SW4（启动连接），连接成功后，可以看到 Master 亮绿灯，Slave 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（框起部分为等待双方输入 Passkey)：

P...	Time	Item	Transmitter	Receiver
1	12:37:08 PM.001 769 750	Connectable ("smp" 55:66:77:00:91:02, Initiator "smp" 55:66:77:00:91:01, 1.65 s)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
4	12:37:08 PM.005 029 125	Connectable ("smp" 55:66:77:00:91:01, 15.6 s)	Master: "smp" 55:66:77:00:91:01	Slave: "Scanning Device"
318	12:37:09 PM.668 638 125	SMP Pairing Request (Keyboard 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
319	12:37:09 PM.668 957 750	SMP Security Request (Bonding, MITM)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
327	12:37:09 PM.699 660 125	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
331	12:37:09 PM.730 681 000	Empty LE Packets (x 364, 5.66 s)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'302	12:37:15 PM.387 391 875	SMP Pairing Confirm (Ca=F97DAD23:70D9D0B0:6FBEE76B:4ABF50FB)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'303	12:37:15 PM.387 788 750	Empty LE Packets (x 252, 3.94 s)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'979	12:37:19 PM.324 666 000	SMP Pairing Confirm (Cb=BE2A022C:C137F8E1:F2716704:035A8220)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'986	12:37:19 PM.356 143 750	SMP Pairing Random (Na=575D9442:A21676C7:D893D0F7:A5BB4616)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'993	12:37:19 PM.387 165 625	SMP Pairing Random (Nb=5647EFBC:69CFC73A:D10786D1:18DAC55B)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'998	12:37:19 PM.418 644 000	LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, SKDm=0xA324E72A74871E57, I...	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'004	12:37:19 PM.449 665 750	LLCP Encryption Response (SKDs=0xCD89EBF03B33A59B, IVs=0x3C2FAC4F)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'019	12:37:19 PM.543 415 625	LLCP Start Encryption Request	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'029	12:37:19 PM.574 893 625	LLCP Start Encryption Response	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'036	12:37:19 PM.605 915 875	LLCP Start Encryption Response	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'045	12:37:19 PM.668 415 750	SMP Encryption Information (LTK=0312BAE9:3C9A926F:8452D384:4D8F900E)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'054	12:37:19 PM.730 915 875	SMP Master Identification (EDIV=0x8F2A, Rand=0x906F3EC0AE5E4609)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'065	12:37:19 PM.793 415 750	SMP Identity Information (IRK=A3B086D9:459694CF:7A1C228E:6FD8D393)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'079	12:37:19 PM.855 915 875	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
2'084	12:37:19 PM.887 394 125	SMP Encryption Information (LTK=0208C117:F7432392:8DC685A2:F0EE1343)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'090	12:37:19 PM.918 644 000	SMP Master Identification (EDIV=0x1051, Rand=0x12B943A4A065C5A9)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'097	12:37:19 PM.949 894 250	SMP Identity Information (IRK=A728C802:7FC864DC:FF9555DC:87B7B5F8)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'104	12:37:19 PM.981 144 125	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
2'105	12:37:19 PM.981 501 125	Empty LE Packets (x 235, 3.66 s)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01

Figure 10.31: Legacy Passkey Entry BothInput 的抓包

10.6.9 Secure Connections Passkey Entry

Master 和 Slave 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 MDSI、MISD、Both Input 三种实现方式。为了便于交叉演示，演示代码中给出 Secure Connections 下 Passkey Entry 的 MISD 方式以 By_Default 方法输入 Passkey，MDSI 和 BothInput 方式用 Manually 方法输入 Passkey。

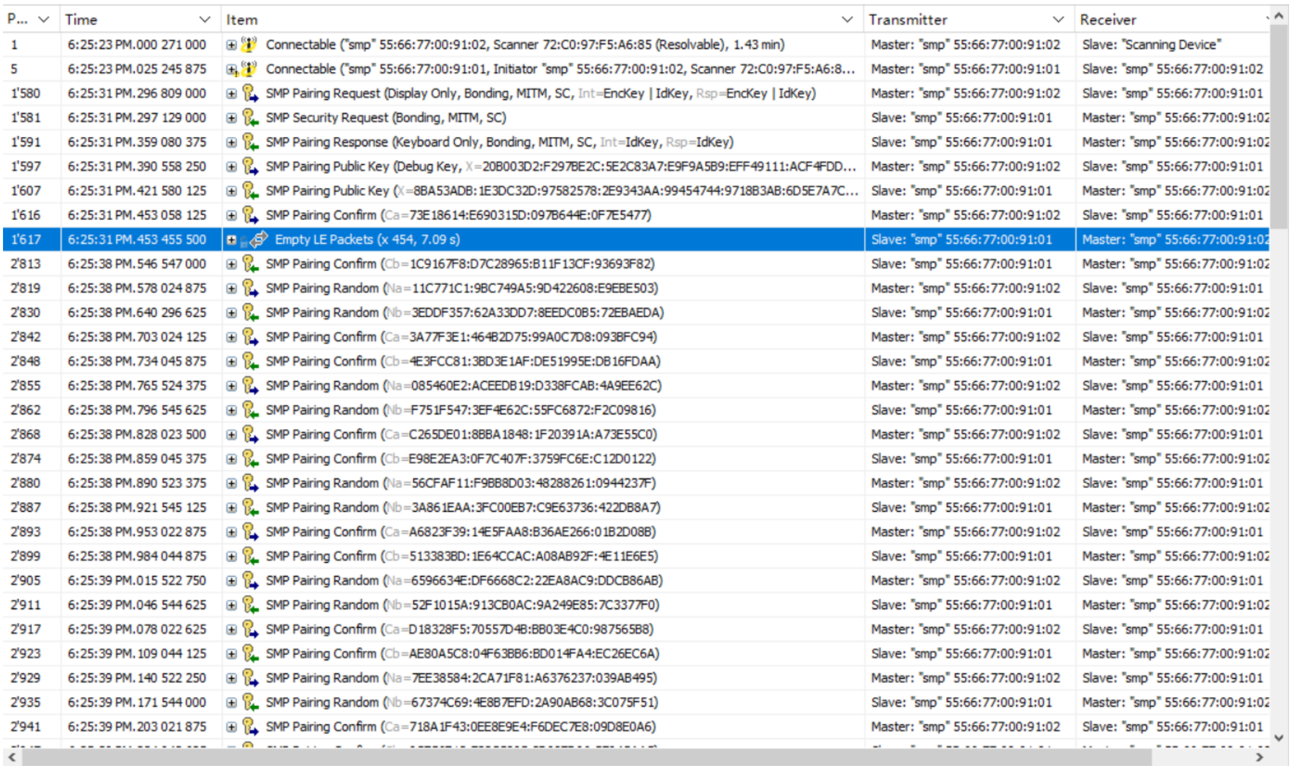
- 对于 MDSI，需在 feature_smp/app_config.h 中定义：

```
#define SMP_TEST_MODE          SMP_TEST_SC_PASSKEY_ENTRY_MDSI
```

Passkey 的输入参考定义：

```
#define PASSKEY_ENTRY_METHOD          PASSKEY_ENTRY_MANUALLY
```

编译 B91_feature，分别烧录 /eagle_ble_sdk/B91_feature/output/B91_feature.bin 到两个开发板中。给其中一个开发板（作为 Slave）的 UART 口 PD2(Tx)，PD3(Rx) 接到电脑串口上，配置波特率 115200。按另一个开发板（作为 Master）的按键 SW4（启动连接），连接成功后，可以看到 Master 亮绿色灯，Slave 亮红色灯。此时 Master 的 Log 输出 Display Pincode，将输出的数在 30s 内填入到 Master 的 UART 串口，可以看到 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 Slave 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=8BA53ADB:1E3DC32D:97582578:2E9343AA:99454744:9718B3AB: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=11C771C1:9BC749A5:9D422608: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=3EDDF357:62A33DD7:8EEDC0B5:72EBAEDA)	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:DB16FDAA)	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=C265DE01:8BBA1848:1F20391A: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=56CFAF11: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=A6823F39:14E5FAA8:B36AE266:01B2D08B)	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=513383BD: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:DDCB86AB)	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:BB03E4C0:987565B8)	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=AE80A5C8:04F63BB6:BD014FA4:EC26EC6A)	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=67374C69:4E8B7EFD: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:F6DEC7E8:09D8E0A6)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01

Figure 10.32: SC Passkey Entry MDSI 的抓包

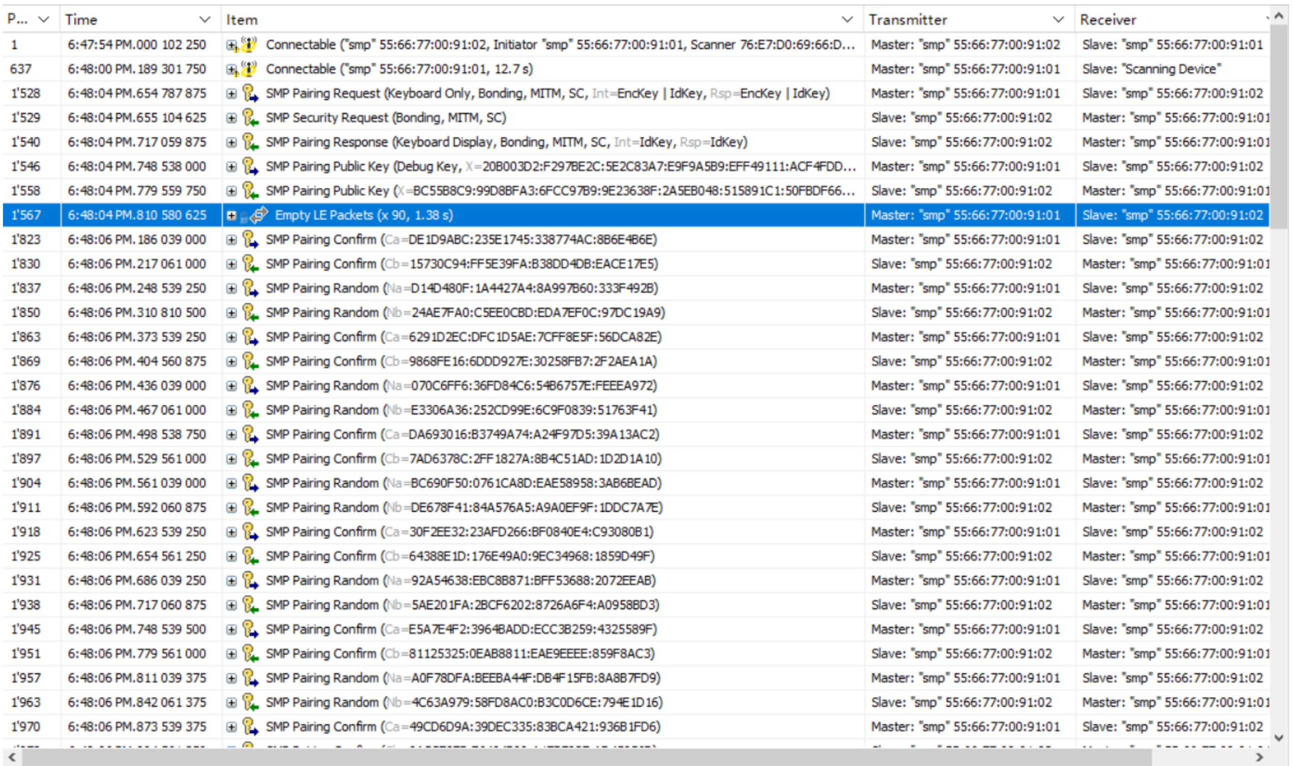
- 对于 MISD，需在 feature_smp/app_config.h 中定义：

```
#define SMP_TEST_MODE        SMP_TEST_SC_PASSKEY_ENTRY_MISD
```

Passkey 的输入参考定义：

```
#define PASSKEY_ENTRY_METHOD          PASSKEY_ENTRY_BY_DEFAULT
```

编译 B91_feature，分别烧录 /eagle_ble_sdk/B91_feature/output/B91_feature.bin 到两个开发板中。按其中一个开发板（作为 Master）的按键 SW4（启动连接），连接成功后，可以看到 Master 亮绿灯，Slave 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 Master 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	6:48:00 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=EncKey | IdKey, Rsp=EncKey | 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:EFF49111: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=BC55B8C9:99D8BFA3:6FCC97B9:9E23638F: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:8B6E4B6E)	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:B38DD4DB: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=24AE7FA0: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:7CFF8E5F: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=9868FE16:6DDD927E:30258FB7:2F2AEA1A)	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:FEEEA972)	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:2FF1827A: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:EAE58958:3AB6BEAD)	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:EBC8B871:BFF53688:2072EEAB)	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=81125325:0EAB8811:EAE9EEEE: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:DB4F15FB: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 10.33: SC Passkey Entry MISD 的抓包

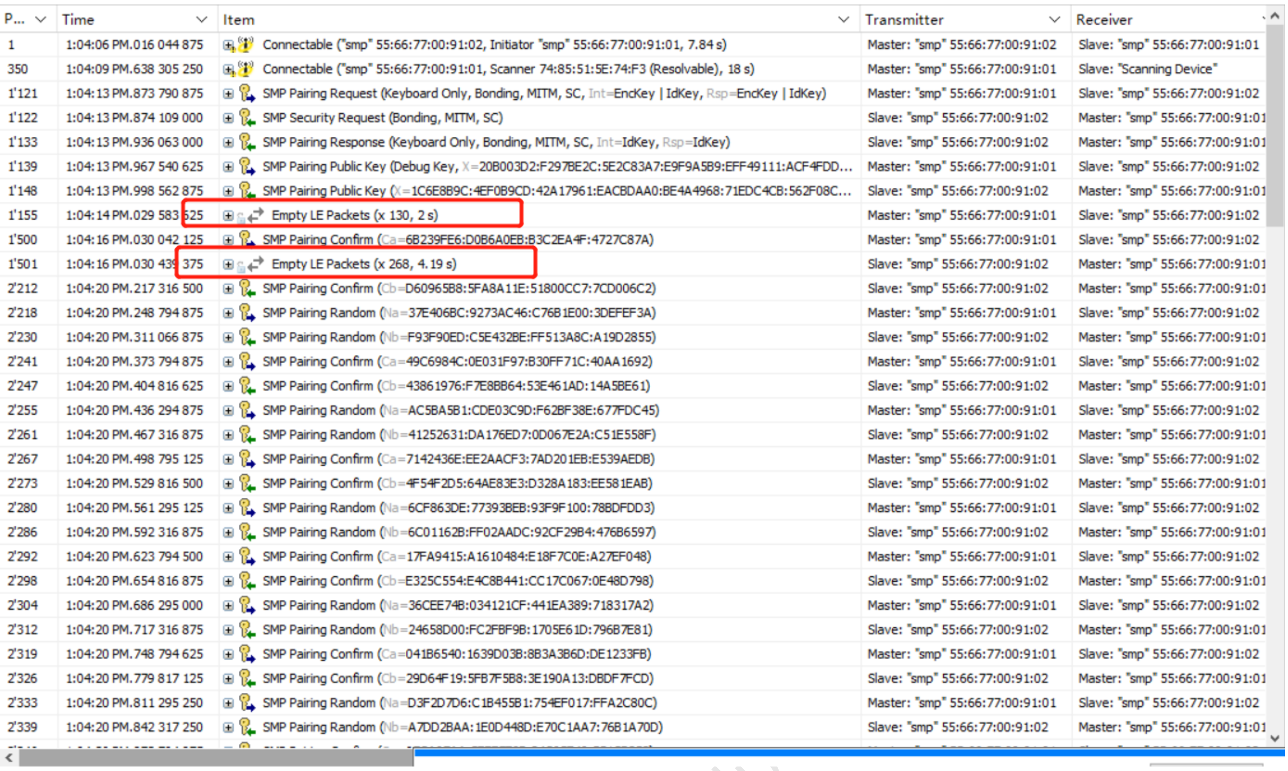
- 对于 Both Input，需在 feature_smp/app_config.h 中定义：

```
#define SMP_TEST_MODE        SMP_TEST_SC_PASSKEY_ENTRY_BOTH_INPUT
```

Passkey 的输入参考定义：

```
#define PASSKEY_ENTRY_METHOD        PASSKEY_ENTRY_MANUALLY
```

编译 B91_feature，分别烧录 /eagle_ble_sdk/B91_feature/output/B91_feature.bin 到两个开发板中。给两个开发板的 UART 口 PD2(Tx)，PD3(Rx) 接到电脑串口上，配置波特率 115200。按其中一个开发板（作为 Master）的按键 SW4（启动连接），连接成功后，可以看到 Master 亮绿色灯，Slave 亮红色灯。在 30s 内将自定义的 Pincode 分别填入到 Slave 和 Master 的 UART 串口。可以看到 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（高亮部分为等待双方输入 Passkey）：



P...	Time	Item	Transmitter	Receiver
1	1:04:06 PM.016 044 875	Connectable ("smp" 55:66:77:00:91:02, Initiator "smp" 55:66:77:00:91:01, 7.84 s)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
350	1:04:09 PM.638 305 250	Connectable ("smp" 55:66:77:00:91:01, Scanner 74:85:51:5E:74:F3 (Resolvable), 18 s)	Master: "smp" 55:66:77:00:91:01	Slave: "Scanning Device"
1'121	1:04:13 PM.873 790 875	SMP Pairing Request (Keyboard Only, Bonding, MITM, SC, Int=EncKey | IdKey, Rsp=EncKey | IdKey)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'122	1:04:13 PM.874 109 000	SMP Security Request (Bonding, MITM, SC)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'133	1:04:13 PM.936 063 000	SMP Pairing Response (Keyboard Only, Bonding, MITM, SC, Int=IdKey, Rsp=IdKey)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'139	1:04:13 PM.967 540 625	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
1'148	1:04:13 PM.998 562 875	SMP Pairing Public Key (X=1C6E8B9C:4EF0B9CD:42A17961:EACBDAA0:BE4A4968:71EDC4CB:562F08C...	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'155	1:04:14 PM.029 583 525	Empty LE Packets (x 130, 2 s)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'500	1:04:16 PM.030 042 125	SMP Pairing Confirm (Ca=6B239FE6:D0B6A0EB:B3C2EA4F:4727C87A)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'501	1:04:16 PM.030 439 375	Empty LE Packets (x 268, 4.19 s)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'212	1:04:20 PM.217 316 500	SMP Pairing Confirm (Cb=D60965B8:5FA8A11E:51800CC7:7CD006C2)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'218	1:04:20 PM.248 794 875	SMP Pairing Random (Na=37E406BC:9273AC46:C76B1E00:3DEFEF3A)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'230	1:04:20 PM.311 066 875	SMP Pairing Random (Nb=F93F90ED:C5E432BE:FF513A8C:A19D2855)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'241	1:04:20 PM.373 794 875	SMP Pairing Confirm (Ca=49C6984C:0E031F97:B30FF71C:40AA1692)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'247	1:04:20 PM.404 816 625	SMP Pairing Confirm (Cb=43861976:F7E8BB64:53E461AD:14A5BE61)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'255	1:04:20 PM.436 294 875	SMP Pairing Random (Na=AC5BA5B1:CDE03C9D:F62BF38E:677FDC45)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'261	1:04:20 PM.467 316 875	SMP Pairing Random (Nb=41252631:DA176ED7:0D067E2A:C51E558F)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'267	1:04:20 PM.498 795 125	SMP Pairing Confirm (Ca=7142436E:EE2AACF3:7AD201EB:E539AEDB)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'273	1:04:20 PM.529 816 500	SMP Pairing Confirm (Cb=4F54F2D5:64AE83E3:D328A183:EE581EAB)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'280	1:04:20 PM.561 295 125	SMP Pairing Random (Na=6CF863DE:77393BEB:93F9F100:78BDFDD3)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'286	1:04:20 PM.592 316 875	SMP Pairing Random (Nb=6C01162B:FF02AADC:92CF29B4:476B6597)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'292	1:04:20 PM.623 794 500	SMP Pairing Confirm (Ca=17FA9415:A1610484:E18F7C0E:A27EF048)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'298	1:04:20 PM.654 816 875	SMP Pairing Confirm (Cb=E325C554:E4C8B441:CC17C067:0E48D798)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'304	1:04:20 PM.686 295 000	SMP Pairing Random (Na=36CEE74B:034121CF:441EA389:718317A2)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'312	1:04:20 PM.717 316 875	SMP Pairing Random (Nb=24658D00:FC2FBF9B:1705E61D:796B7E81)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'319	1:04:20 PM.748 794 625	SMP Pairing Confirm (Ca=041B6540:1639D03B:8B3A3B6D:DE1233FB)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'326	1:04:20 PM.779 817 125	SMP Pairing Confirm (Cb=29D64F19:5FB7F5B8:3E190A13:DBDF7FCD)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'333	1:04:20 PM.811 295 250	SMP Pairing Random (Na=D3F2D7D6:C1B455B1:754EF017:FFA2C80C)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'339	1:04:20 PM.842 317 250	SMP Pairing Random (Nb=A7DD2BAA:1E0D448D:E70C1AA7:76B1A70D)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01

Figure 10.34: SC Passkey Entry Both Input 的抓包

10.6.10 Secure Connections Numeric Comparison

Master 和 Slave 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_config.h 中定义：

```
#define SMP_TEST_MODE        SMP_TEST_SC_NUMERIC_COMPARISON
```

编译 B91_feature，分别烧录 /eagle_ble_sdk/B91_feature/output/B91_feature.bin 到两个开发板中。按其中一个开发板（作为 Master）的按键 SW4（启动连接），连接成功后，可以看到 Master 亮绿灯，Slave 亮红灯，此时从双方的 Log 中看到 Numeric Comparison Pincode 的值，理论上二者应该相等。分别在 Master 和 Slave 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 Master 和 Slave 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, 21 s)	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=EncKey | IdKey, Rsp=EncKey | 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:C553B79F:43BE2158:DBEA0548: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 (Cb=9BA10FD4:95D7F814:6361D2E9:20AB9444)	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 (Na=FC1BC417:C4DAA576:D5AA8BEE: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:1E27193D)	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:DD2FE6D2:2A1FAA1A: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=0xDF1AA076C1F78C53, 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=0xDDED3C7C3E635E5B, 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 10.35: SC Numeric Comparison 的抓包

10.6.11 Legacy OOB

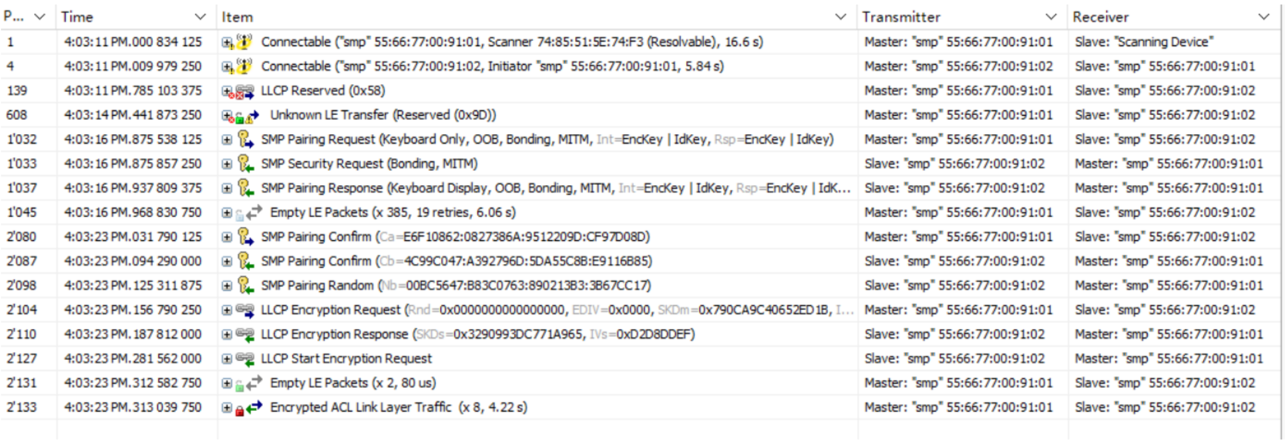
Master 和 Slave 使能 SMP 功能，以 Legacy OOB 方式配对的演示。根据蓝牙协议，在 Legacy 配对方式下，当双方均支持 OOB，拥有对方 OOB 数据时，会使用 Legacy OOB 方式进行匹配。使用该功能需在 feature_smp/app_config.h 中定义：

```
#define SMP_TEST_MODE          SMP_TEST_LEGACY_OOB
```

这里选择用 Manually 方式通过 UART 输入 OOB data，参考定义：

```
#define PASSKEY_ENTRY_METHOD          PASSKEY_ENTRY_MANUALLY
```

编译 B91_feature，分别烧录 /eagle_ble_sdk/B91_feature/output/B91_feature.bin 到两个开发板中。给两个开发板的 UART 口 PD2(Tx)，PD3(Rx) 接到电脑串口上，配置波特率 115200。按其中一个开发板（作为 Master）的按键 SW4（启动连接），连接成功后，可以看到 Master 亮绿灯，Slave 亮红灯，此时从双方的 Log 中看到 Requset OOB。。在 30s 内将自定义的 OOB data 分别填入到 Slave 和 Master 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=EncKey | IdKey, Rsp=EncKey | 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=EncKey | IdKey, Rsp=EncKey | IdK...	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=E6F10862:0827386A: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 10.36: Legacy OOB 的抓包

可以看到这里的加密内容没有被解析，解析需要用 STK，Telink 不提供获取 STK 的接口和方法。但可以通过读 RAM 得到 LTK，这样重连用 LTK 加密后，将 LTK 加载到抓包器中：

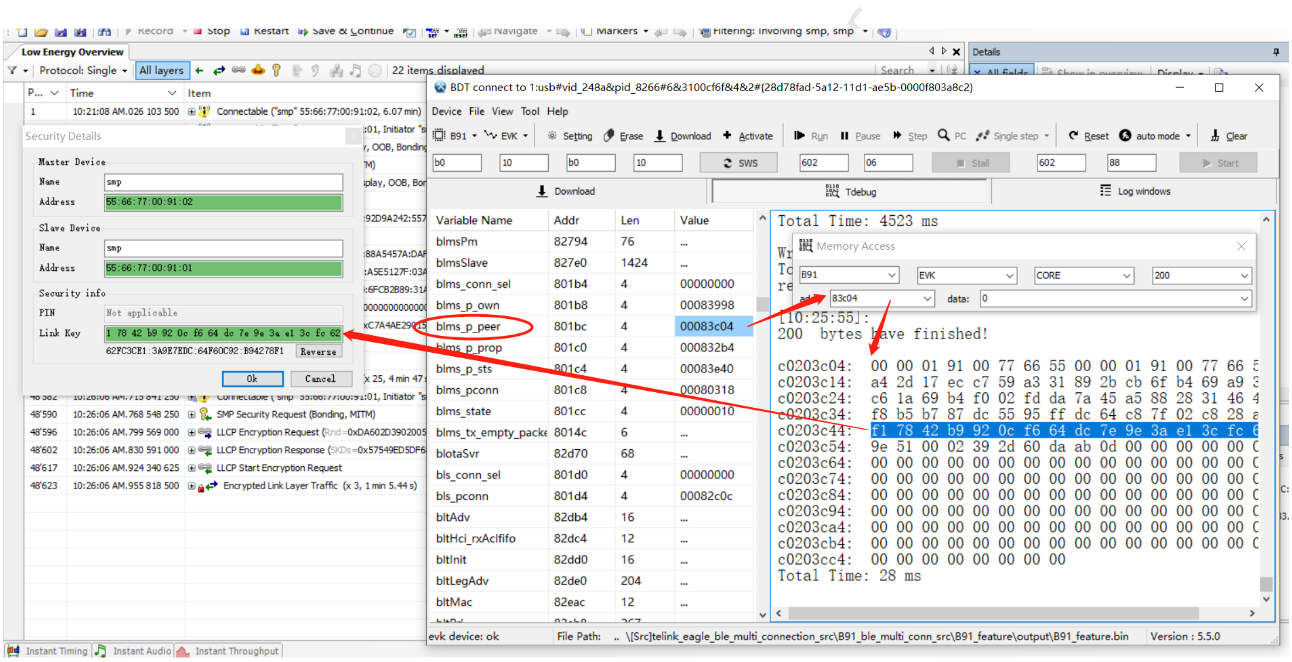


Figure 10.37: Legacy OOB 加载 LTK

就可以解析重连后的加密内容：

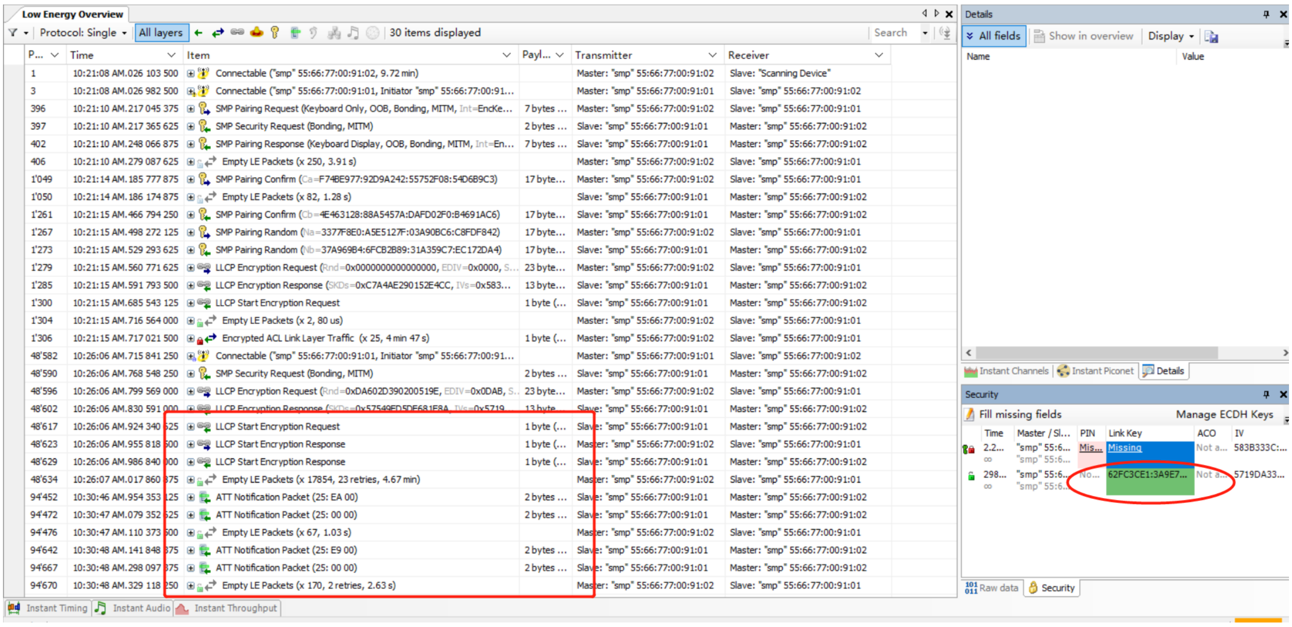


Figure 10.38: Legacy OOB LTK 解析

10.6.12 Secure Connections OOB

Master 和 Slave 使能 SMP 功能，以 Secure Connections OOB 方式配对的演示，SDK 中暂时不支持。

10.6.13 Custom Pair

关闭 Slave 和 Master 的 SMP 功能时，Telink 提供了一种不通过 SMP 实现设备自动回连的方法 —— Custom Pair，这里提供该功能的演示。使用该功能需在 feature_smp/app_config.h 中定义：

```
#define SMP_TEST_MODE        SMP_TEST_CUSTOM_PAIR
```

编译 B91_feature，分别烧录 /eagle_ble_sdk/B91_feature/output/B91_feature.bin 到两个开发板中。按其中一个开发板（作为 Master）的按键 SW4（启动连接），连接成功后，可以看到 Master 亮绿灯，Slave 亮红灯，代表二者连接成功。此时给 Master 或 Slave 重新上电，可以看到另一方的灯熄灭之后，双方的灯很快点亮，代表回连成功。

10.6.14 异常处理

10.6.14.1 按按键没有响应

(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：

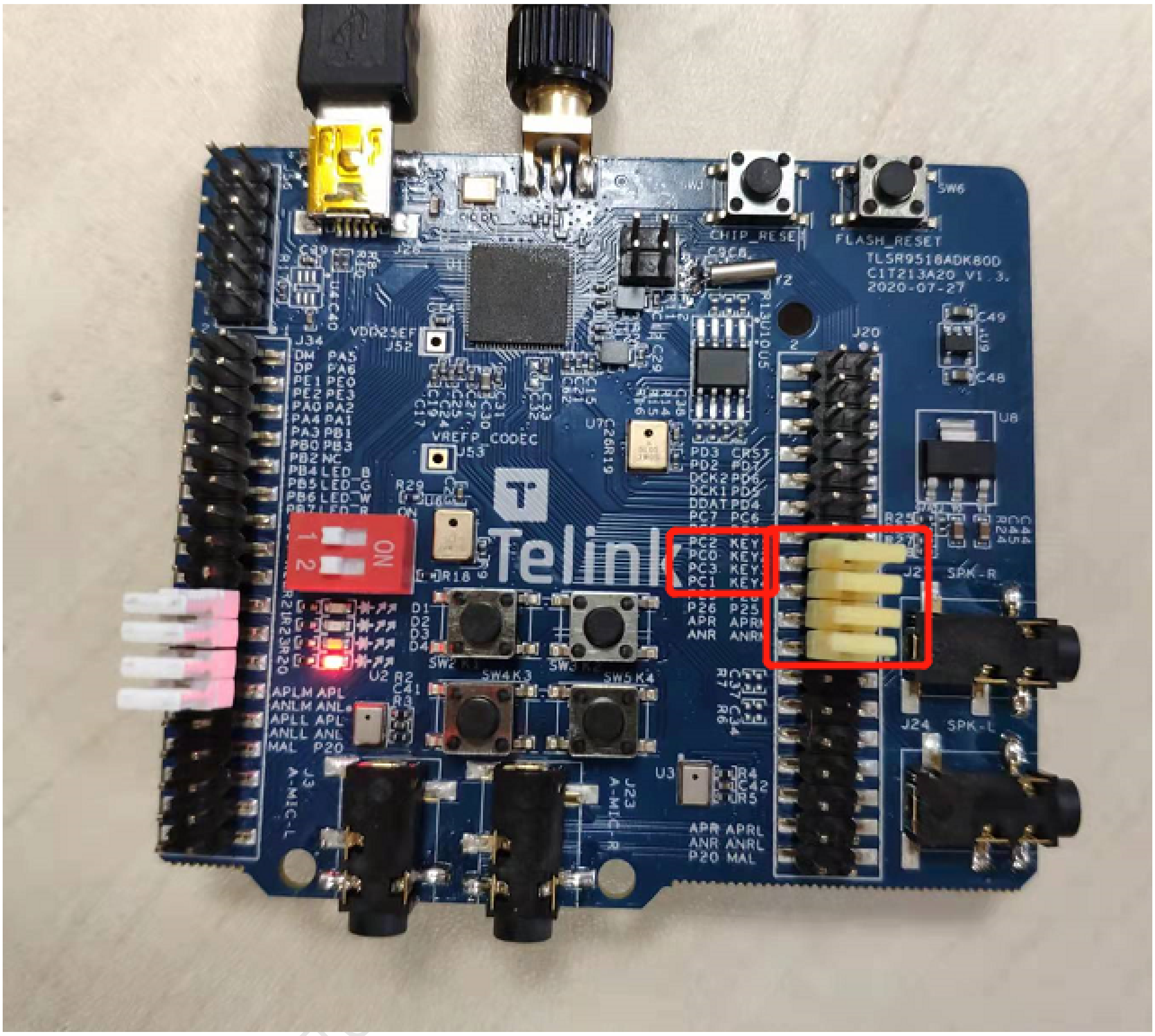


Figure 10.39: Key 跳线帽接错

(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。

10.6.14.2 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：

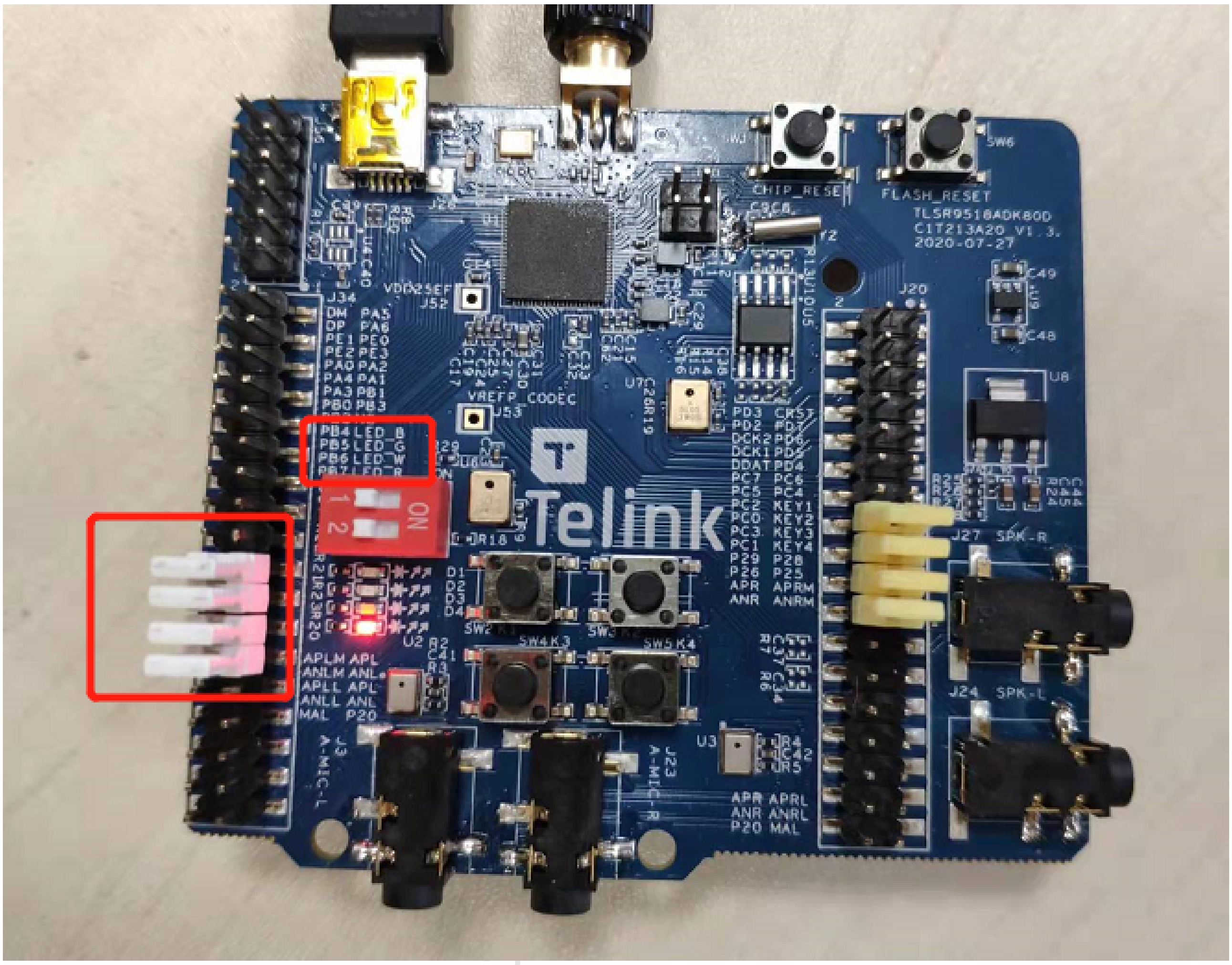


Figure 10.40: LED 跳线帽接错

10.7 feature_ota

- 功能：BLE OTA 功能演示。
- 主要硬件：B91 开发板 x 2

以 Telink B91 BLE Multiple Connection SDK 为例，应用层代码在 eagle_ble_sdk/vendor/B91_feature/feature_ota 下，需要修改 eagle_ble_sdk/vendor/B91_feature/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。

编译固件后，分别烧录到两个开发板中，并给其中一个开发板的 Flash 0x80000 位置烧录 OTA 固件（这里使用以上同一个编译固件）。重新给两个开发板上电后，连续按 Master 的按键 SW2 或 SW3 五次，触发 OTA 测试模式，可以看到蓝色灯和绿色灯三次慢闪，代表切换 OTA 测试模式成功。

按 Master 的按键 SW2 开始 OTA，可以看到两个开发板上的蓝色灯点亮，代表正在 OTA，抓包可以看到 Master 不停向 Slave 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 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 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 3B...	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 ...	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 00 13 ...	20 bytes (08 00 00 00 13 03 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 0B 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:

Packet Start @ti

Figure 10.41: feature_ota OTA 开始

升级完成，Slave 发送给 Master 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，Master 存有 Slave 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 ...	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 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 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.732 313 625	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 10.42: feature_ota OTA 完成

10.8 feature_whitelist

- 功能：BLE 白名单功能演示。
- 主要硬件：B91 开发板 x 2

以 Telink B91 BLE Multiple Connection SDK 为例，应用层代码在 eagle_ble_sdk/vendor/B91_feature/feature_whitelist 下，需要修改 eagle_ble_sdk/vendor/B91_feature/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。

由于 SDK 中待连接 Master 白名单和待连接 Slave 白名单共用同一个白名单列表，所以，这里分别对连接 Master 白名单设备和连接 Slave 白名单设备进行演示。

先编译一个使能白名单功能但不使用白名单的主从一体设备固件（为方便后文引用，称为 NoWhiteList），将初始化代码中的

```
blc_ll_setAdvParam(ADV_INTERVAL_30MS, ADV_INTERVAL_30MS, ADV_TYPE_CONNECTABLE_UNDIRECTED,
↪  OWN_ADDRESS_PUBLIC, 0, NULL, BLT_ENABLE_ADV_ALL, ADV_FP_ALLOW_SCAN_WL_ALLOW_CONN_WL);
```

修改为

```
blc_ll_setAdvParam(ADV_INTERVAL_30MS, ADV_INTERVAL_30MS, ADV_TYPE_CONNECTABLE_UNDIRECTED,
↪  OWN_ADDRESS_PUBLIC, 0, NULL, BLT_ENABLE_ADV_ALL, ADV_FP_ALLOW_SCAN_ANY_ALLOW_CONN_ANY);
```

再将

```
blc_ll_setScanParameter(SCAN_TYPE_PASSIVE, SCAN_INTERVAL_100MS, SCAN_WINDOW_100MS,
↪  OWN_ADDRESS_PUBLIC, SCAN_FP_ALLOW_ADV_WL);
```

修改为

```
blc_ll_setScanParameter(SCAN_TYPE_PASSIVE, SCAN_INTERVAL_100MS, SCAN_WINDOW_100MS,
↪  OWN_ADDRESS_PUBLIC, SCAN_FP_ALLOW_ADV_ANY);
```

编译，NoWhiteList.bin。

10.8.1 Master 设置扫描 Slave 白名单

连接 Slave 白名单设备的演示，就是将要连接的 Slave 的 MAC 地址添加到白名单中，作为 Master 的 Telink 设备通过对扫描到的广播进行地址过滤的方式决定是否进行连接。

基于 NoWhiteList 代码，还原扫描参数配置语句：

```
blc_ll_setScanParameter(SCAN_TYPE_PASSIVE, SCAN_INTERVAL_100MS, SCAN_WINDOW_100MS,
↪  OWN_ADDRESS_PUBLIC, SCAN_FP_ALLOW_ADV_ANY);
```

为

```
blc_ll_setScanParameter(SCAN_TYPE_PASSIVE, SCAN_INTERVAL_100MS, SCAN_WINDOW_100MS,
↪  OWN_ADDRESS_PUBLIC, SCAN_FP_ALLOW_ADV_WL);
```

编译，得到 Master 的固件，烧录到其中一块开发板上作为 Master。另一块开发板烧录 NoWhiteList.bin，并修改 MAC 地址为白名单地址 33:88:99:99:99:99：

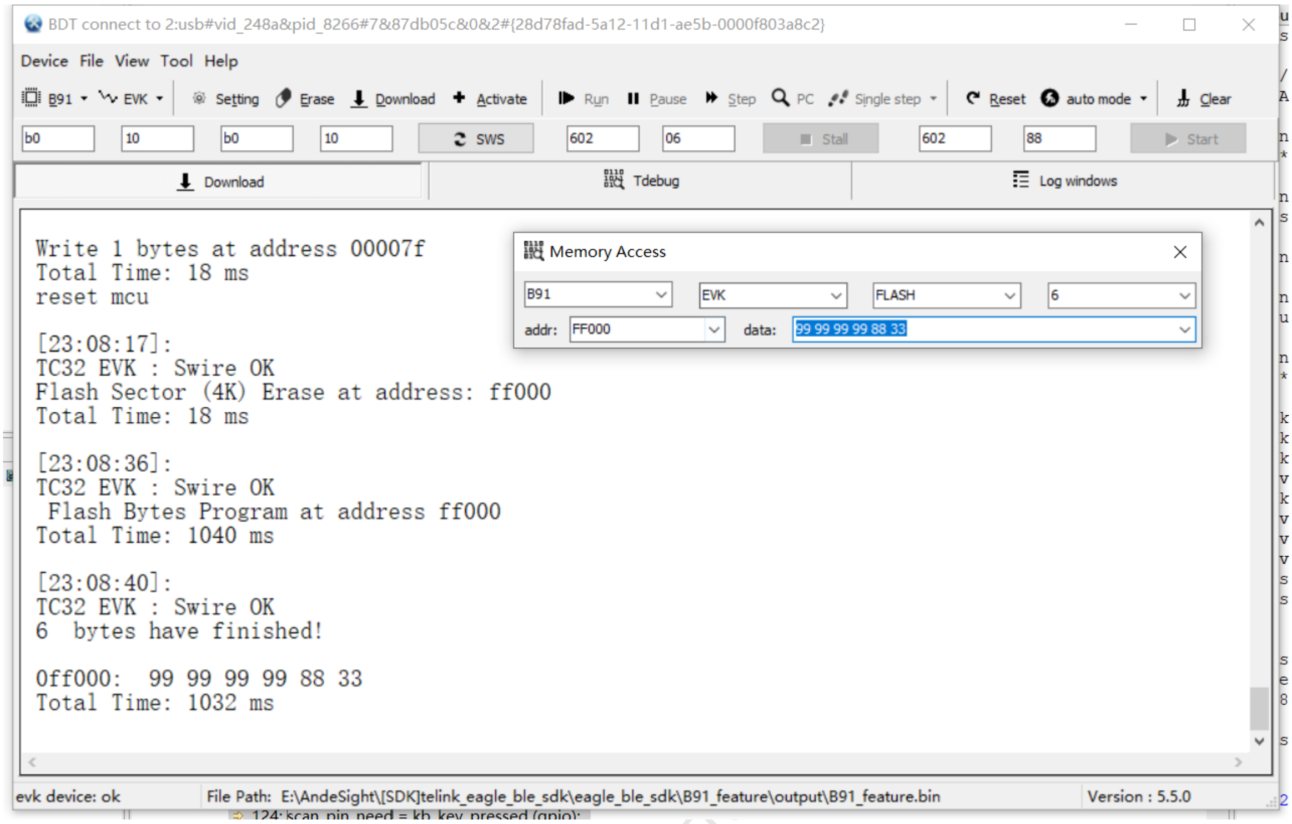


Figure 10.43: 设置白名单设备 MAC 地址

烧录完成后，通过按 Master 的 SW4 启动连接，可以看到 Master 和 Slave 成功连接，抓包如下：

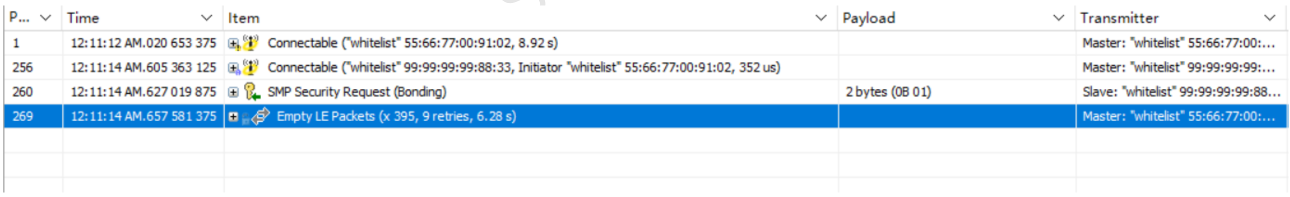


Figure 10.44: Master 连接白名单 Slave 抓包

如果此时 Slave 的 MAC 地址不是 33:88:99:99:99:99，按 Master 的 SW4 按键，是没有响应的，抓包中不会看到建立连接及建立连接之后的包。

10.8.2 Slave 设置 Master 连接白名单

连接 Master 白名单设备的演示，就是将要连接的 Master 的 MAC 地址添加到白名单中，作为 Slave 的 Telink 设备通过对扫描包或连接包的过滤来决定是否进行响应。

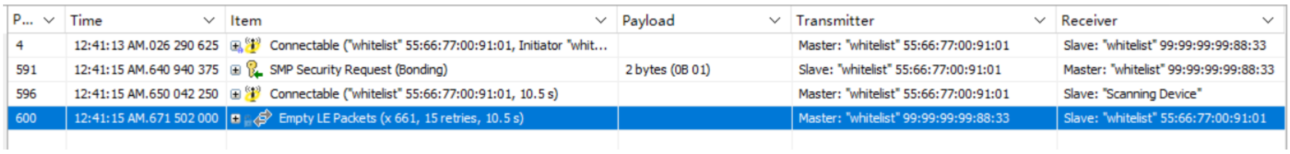
基于 NoWhiteList 代码，还原扫描参数配置语句：

```
blc_ll_setAdvParam(ADV_INTERVAL_30MS, ADV_INTERVAL_30MS, ADV_TYPE_CONNECTABLE_UNDIRECTED,
↪  OWN_ADDRESS_PUBLIC, 0, NULL, BLT_ENABLE_ADV_ALL, ADV_FP_ALLOW_SCAN_ANY_ALLOW_CONN_ANY);
```

为

```
blc_ll_setAdvParam(ADV_INTERVAL_30MS, ADV_INTERVAL_30MS, ADV_TYPE_CONNECTABLE_UNDIRECTED,
↪ OWN_ADDRESS_PUBLIC, 0, NULL, BLT_ENABLE_ADV_ALL, ADV_FP_ALLOW_SCAN_WL_ALLOW_CONN_WL);
```

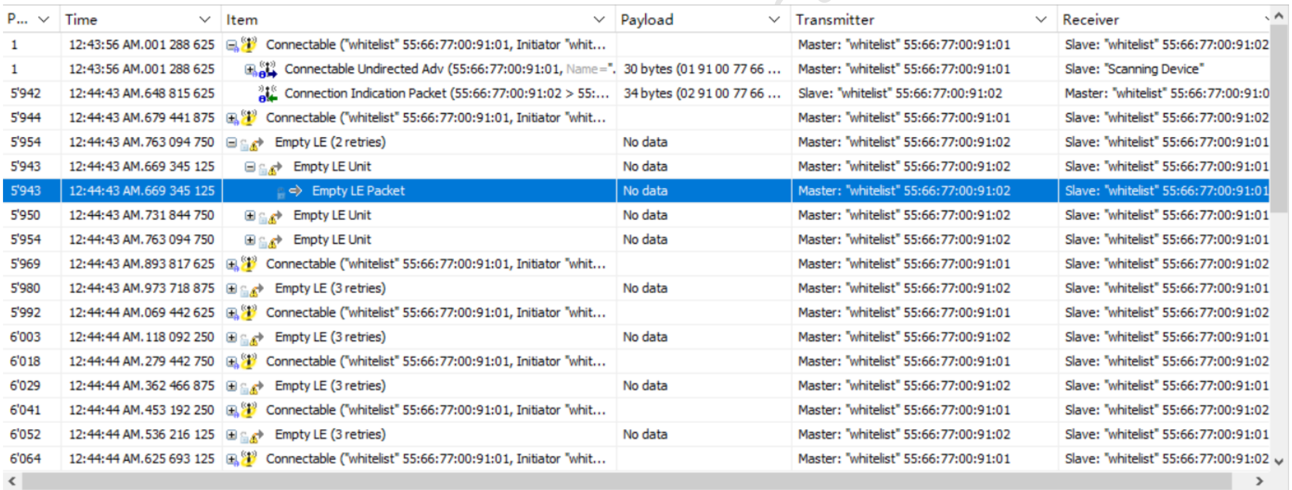
编译，得到 Slave 的固件，烧录到其中一块开发板上作为 Slave。另一块开发板烧录 NoWhiteList.bin，并修改 MAC 地址为白名单地址 33:88:99:99:99:99。烧录完成后，通过按 Master 的 SW4 启动连接，可以看到 Master 和 Slave 成功连接，抓包如下：



P...	Time	Item	Payload	Transmitter	Receiver
4	12:41:13 AM.026 290 625	Connectable ("whitelist" 55:66:77:00:91:01, Initiator "whit...		Master: "whitelist" 55:66:77:00:91:01	Slave: "whitelist" 99:99:99:99:88:33
591	12:41:15 AM.640 940 375	SMP Security Request (Bonding)	2 bytes (0B 01)	Slave: "whitelist" 55:66:77:00:91:01	Master: "whitelist" 99:99:99:99:88:33
596	12:41:15 AM.650 042 250	Connectable ("whitelist" 55:66:77:00:91:01, 10.5 s)		Master: "whitelist" 55:66:77:00:91:01	Slave: "Scanning Device"
600	12:41:15 AM.671 502 000	Empty LE Packets (x 661, 15 retries, 10.5 s)		Master: "whitelist" 99:99:99:99:88:33	Slave: "whitelist" 55:66:77:00:91:01

Figure 10.45: Slave 连接白名单 Master 抓包

如果此时 Master 的 MAC 地址不是 33:88:99:99:99:99，按 Maste 的 SW4 按键，会看到 Master 发出了 Connect Request，Slave 却不会响应，以至于 Master 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 ...	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:...	34 bytes (02 91 00 77 66 ...	Slave: "whitelist" 55:66:77:00:91:02	Master: "whitelist" 55:66:77:00:91:0
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 10.46: Slave 连接非白名单 Master 抓包

10.9 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 Telink B91 BLE Multiple Connection SDK 为例，应用层代码在 eagle_ble_sdk/vendor/B91_feature/feature_soft_timer 下，需要修改 eagle_ble_sdk/vendor/B91_feature/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、PB0、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：

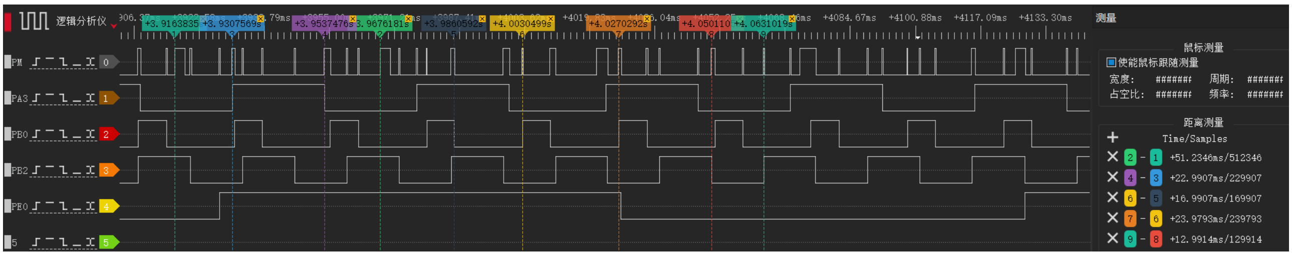


Figure 10.47: SoftTimer Debug IO capture

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，PB0 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 其他模块

11.1 PA

如果需要使用 RF PA，8253/8258 请参考 drivers/8258/rf_pa.h，8273/8278 请参考 drivers/8278/driver_ext/ext_rf.h，B91 系列请参考 drivers/B91/ext_driver/software_pa.h。

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 调试方法

该章节介绍几种在开发过程中常用的调试方法。

12.1 GPIO 模拟 UART 打印

为方便 B85m 与 B91 用户调试，Telink BLE Multiple Connection 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 ///////////////////////////////////////
#if (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 12.1: 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。

12.2 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。

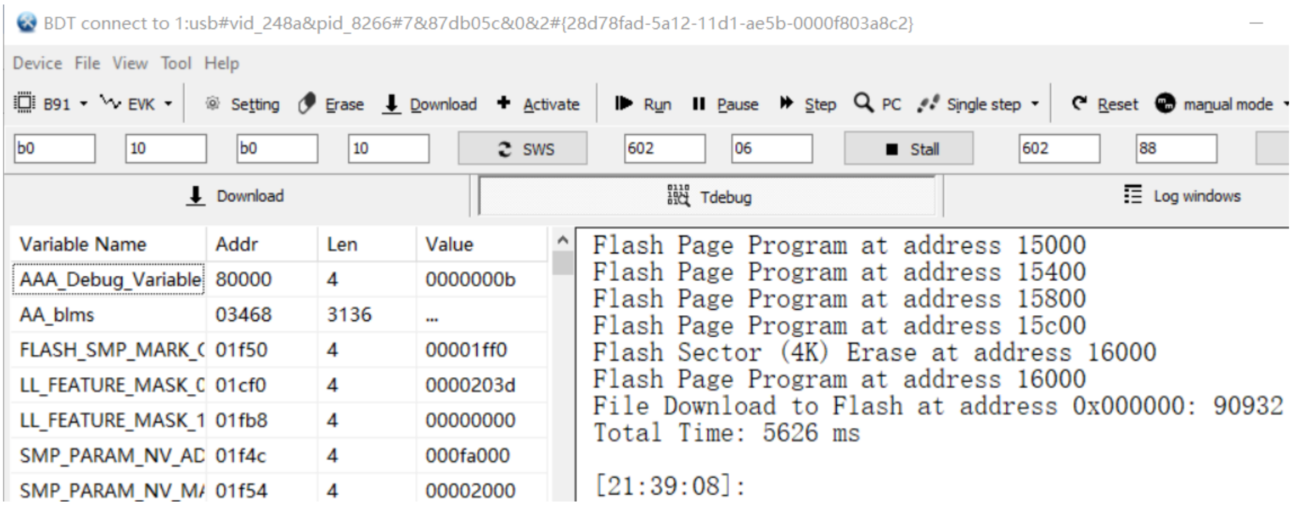


Figure 12.2: BDT 读取全局变量

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 B85m 生成的文件名是 xxx.lst，B91 是 objdump.txt），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。

12.3 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。

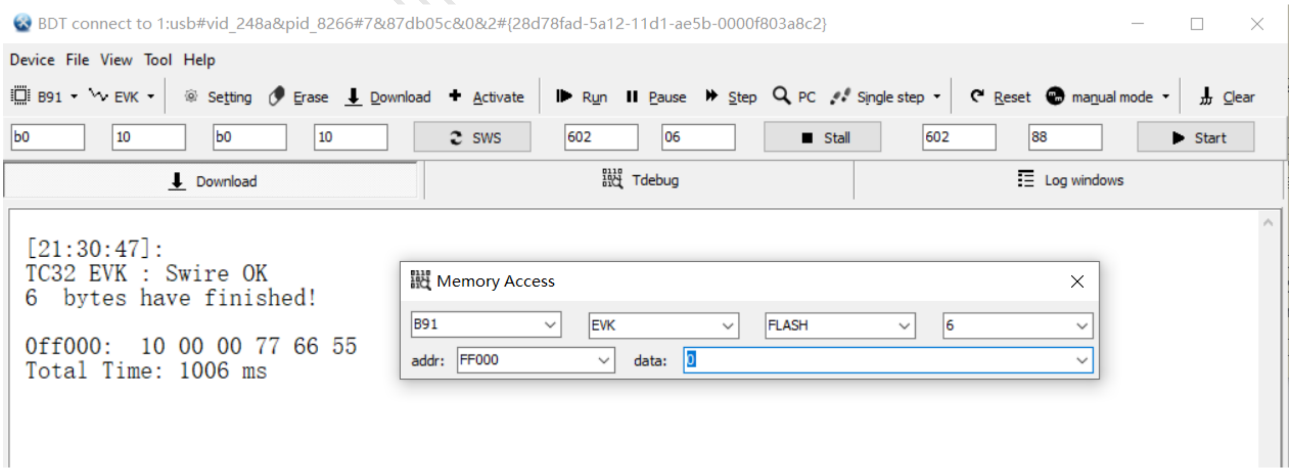


Figure 12.3: BDT Memory Access 功能

12.4 BDT 工具读 PC 指针

Telink 官方提供的 BDT 工具，可以用来读取 B85m 的 PC（Program Counter）指针，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。

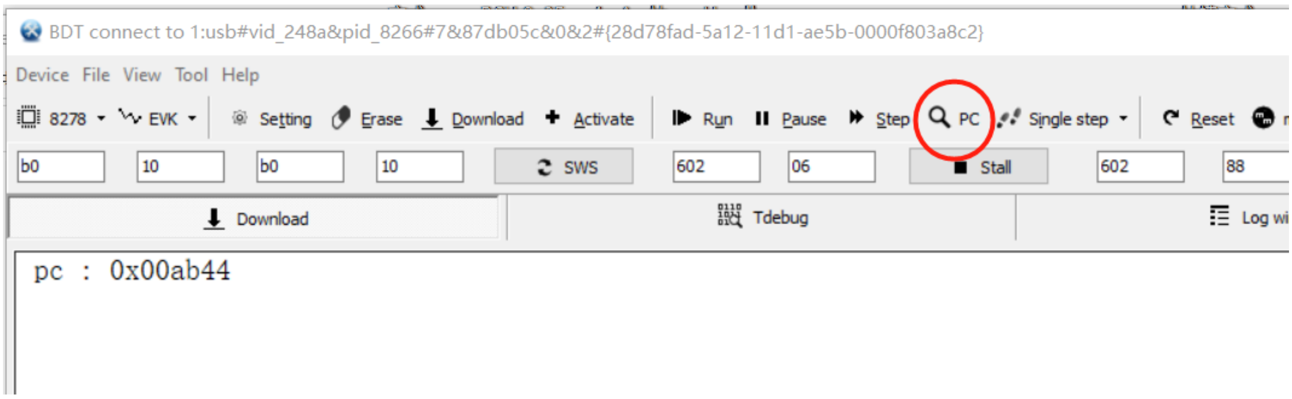


Figure 12.4: BDT PC 指针读取

12.5 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：

```
#if(DEBUG_GPIO_ENABLE)
    #define GPIO_CHN0                               GPIO_PE1
    #define GPIO_CHN1                               GPIO_PE2
    #define GPIO_CHN2                               GPIO_PA0
    #define GPIO_CHN3                               GPIO_PA4
    #define GPIO_CHN4                               GPIO_PA3
    #define GPIO_CHN5                               GPIO_PB0
    #define GPIO_CHN6                               GPIO_PB2
    #define GPIO_CHN7                               GPIO_PE0

    #define GPIO_CHN8                               GPIO_PA2
    #define GPIO_CHN9                               GPIO_PA1
    #define GPIO_CHN10                              GPIO_PB1
    #define GPIO_CHN11                              GPIO_PB3
    #define GPIO_CHN12                              GPIO_PC7
    #define GPIO_CHN13                              GPIO_PC6
    #define GPIO_CHN14                              GPIO_PC5
    #define GPIO_CHN15                              GPIO_PC4


    #define PE1_OUTPUT_ENABLE                       1
    #define PE2_OUTPUT_ENABLE                       1
```

Figure 12.5: 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 Telink B91 BLE Multiple Connection SDK 中 CHN14 所代表的 rf_irq_handler，CHN15 所代表的 stimer_irq_handler。

12.6 USB my_dump_str_data

在 Telink BLE Multiple Connection 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 附录

13.1 附录 1：crc16 算法

```
unsigned shortcrc16 (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;
}
```