泰凌 B80 BLE Single Connection

SDK 开发手册

AN-22070701-C1

Ver1.0.0

2022.07.07

Keyword

BLE

Brief

本文档为泰凌微电子 B80 BLE Single Connection SDK 的开发指南，适用于 B80 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	修改内容
V1.0.0	初次发布

Contents

List of Figures

List of Tables

1 SDK 介绍

该 BLE SDK 提供 BLE slave 开发 demo，用户可以在这些 demo 基础上开发自己的应用程序。

1.1 软件组织架构

该 BLE SDK 软件架构包括 APP 应用层和 BLE stack 协议栈部分。

在 IDE 中导入 sdk 工程后，显示的文件组织结构如下图所示。顶层文件夹有 8 个：algorithm，application，boot，common，drivers，proj_lib，stack，vendor。

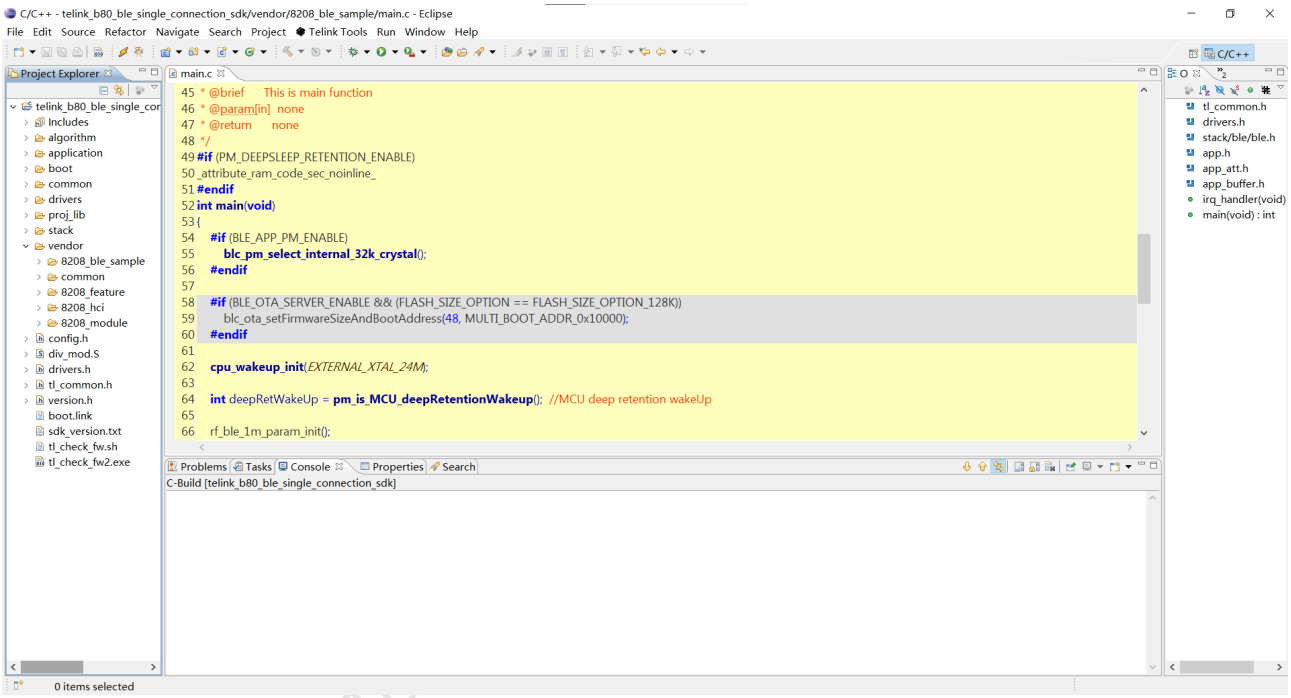


Figure 1.1: SDK 文件结构

- Algorithm：提供加密算法相关的函数。
- Application：提供一些通用的应用处理程序。
- boot：提供芯片的 software bootloader，即 MCU 上电启动或 deepsleep 唤醒后的汇编处理过程，为后面 C 语言程序的运行搭建好环境。
- 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：提供与 MCU 紧密相关的硬件设置和外设驱动程序，如 clock、flash、i2c、usb、gpio、uart 等。
- proj_lib：存放 SDK 运行所必需的库文件（如 liblt_8208.a）。BLE 协议栈、RF 驱动、PM 驱动等文件，被封装在库文件里，用户无法看到源文件。
- stack：存放 BLE 协议栈相关的头文件。源文件被编译到库文件里面，对于用户是不可见的。
- vendor：用于存放用户应用层代码。

1.1.1 main.c

包括 main 函数入口，系统初始化的相关函数，以及无限循环 while(1) 的写法，建议不要对此文件进行任何修改，直接使用固有写法。

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

    #if (BLE_OTA_SERVER_ENABLE && (FLASH_SIZE_OPTION == FLASH_SIZE_OPTION_128K))
        blc_ota_setFirmwareSizeAndBootAddress(48, MULTI_BOOT_ADDR_0x10000);
    #endif

    cpu_wakeup_init(EXTERNAL_XTAL_24M);

    int deepRetWakeUp = pm_is_MCU_deepRetentionWakeup();  //MCU deep retention wakeUp

    rf_ble_1m_param_init();

    clock_init(SYS_CLK_TYPE);

    gpio_init( !deepRetWakeUp );  //analog resistance will keep available in deepSleep mode, so
↪   no need initialize again

    /* load customized freq_offset CAP value and TP value. */
    blc_app_loadCustomizedParameters();

    #if FIRMWARES_SIGNATURE_ENABLE
        blt_firmware_signature_check();
    #endif

    #if (PM_DEEPSLEEP_RETENTION_ENABLE)
        if( deepRetWakeUp ){
            user_init_deepRetn ();
        }
        else
    #endif
        {
        #if(BATT_CHECK_ENABLE)
            blc_app_loadADCParameters();
        #endif
            user_init_normal ();
```

```
        }

    #if (MODULE_WATCHDOG_ENABLE)
        wd_set_interval_ms(WATCHDOG_INIT_TIMEOUT,CLOCK_SYS_CLOCK_1MS);
        wd_start();
    #endif

    irq_enable();

    while (1) {
        #if (MODULE_WATCHDOG_ENABLE)
            wd_clear(); //clear watch dog
        #endif

        main_loop ();
    }
}
```

1.1.2 app_config.h

用户配置文件，用于对整个系统的相关参数进行配置，包括 BLE 相关参数、GPIO 的配置、PM 低功耗管理的相关配置等。

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。

1.1.3 Application File

- app.c：用户主文件，用于完成 BLE 协议栈初始化、数据处理、低功耗处理等。
- app_att.c：service 和 profile 的配置文件，有 Telink 定义的 Attribute 结构，根据该结构，已提供 GATT、标准 HID 和私有的 OTA、MIC 等相关 Attribute，用户可以参考这些添加自己的 service 和 profile。
- app_buffer.c：用于配置 ACL RX FIFO、ACL TX FIFO、L2CAP RX Buffer 等。
- app_ui.c：按键、OTA 等用户任务的处理文件。

1.1.4 BLE Stack Entry

Telink BLE SDK 中 BLE stack 部分 code 的入口函数有两个：

(1) main.c 文件 irq_handler 函数中 BLE 相关中断的处理入口 blc_sdk_irq_handler。

```
_attribute_ram_code_ void rf_irq_handler (void)
{
......
blc_sdk_irq_handler ();
```

```
......
}
```

(2) application file main_loop 中 BLE 逻辑和数据处理的函数入口 blc_sdk_main_loop。

```
void main_loop (void)
{
////////////////////// BLE entry /////////////////////////////
    blc_sdk_main_loop();
////////////////////// UI entry /////////////////////////////
......
////////////////////// PM process ////////////////////
......
}
```

1.2 Applicable IC

适用如下几种 IC 型号，它们属于 B80 系列，均为同一内核，8208A/8208B/8208C 硬件模块基本一致，只是在 flash 方面略有差异。具体如下表所示。

Table 1.1: BLE B80 SDK 支持芯片及资源

IC	Flash size
8208A	无内置 Flash
8208B	内置 128 kB
8208C	内置 512 kB

因为上面 3 颗 IC 的差异主要是 Flash，其他部分是一致的，SDK 文件架构除了 SDK/boot/启动脚本（即 software bootloader 文件）和 boot.link 文件有差异外，其他部分完全共用。

1.3 Software Bootloader 介绍

software bootloader 文件存放在 boot 目录下：

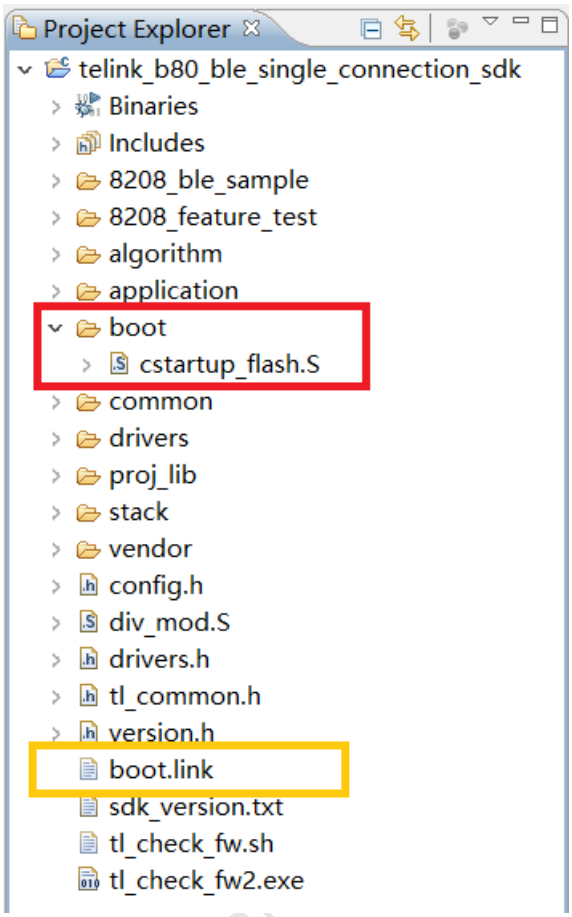


Figure 1.2: bootloader 以及 boot.link 路径

1.4 Demo 介绍

Telink BLE SDK 给用户提供了多个 BLE Slave Demo。

用户通过软硬件 demo 的运行，可以观察到直观的效果。用户也可以在 demo code 上进行修改，完成自己的应用开发。

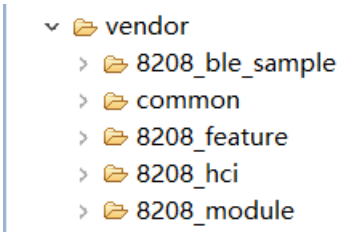


Figure 1.3: BLE SDK 提供的 demo code

BLE slave demo 及区别如下表所示。

Table 1.2: BLE slave 的 demo 简要

Demo	Stack	Application	MCU Function
8208 hci	BLE controller	No	BLE controller, only Advertising and one Slave
8208 module	BLE controller + host	Application 在主控 MCU 上	BLE 透传模组
8208 ble sample	BLE controller + host	最简单的 slave demo，广播和连接功能	主控 MCU
8208 feature	BLE controller + host	各种 feature 的集合	主控 MCU

8208 hci 是一个 BLE slave controller，提供了基于 USB/UART 的 HCI，和其他 MCU 的 host 通信，形成一个完整的 BLE slave 系统。

8208 module 只作为 BLE 透传模组，与主控 MCU 通过 UART 接口通信，一般应用代码写在对方主控 MCU。

8208 module 实现了通过透传模组控制相关状态变化的功能。

8208 ble sample 是一个简化的 slave demo，可以和标准的 IOS/android 设备配对连接。

1.4.1 Feature Demo 和 Driver Demo

8208_feature_test 针对一些常用的 BLE 相关的 feature 给出了 demo code，用户可参考这些 demo 完成自己的功能实现，详情见 code。该文档 BLE 部分会介绍所有的 feature。

在 8208_feature_test 工程里面 feature_config.h 中对宏“FEATURE_TEST_MODE”进行选择性的定义，即可切换到不同 feature 的 Demo。

2 MCU 基础模块

2.1 MCU 地址空间

2.1.1 MCU 地址空间分配

MCU 地址空间分配如下图所示。

Telink B80 MCU 的最大寻址空间为 16M bytes：

- 从 0 到 0x7FFFFF 的 8M 空间为程序空间，即最大程序容量为 8M bytes。
- 0x800000 到 0xFFFFFF 为外部设备空间：0x800000~0x80FFFF 为寄存器空间；0x840000~0x84FFFF 为 64K Sram 空间。

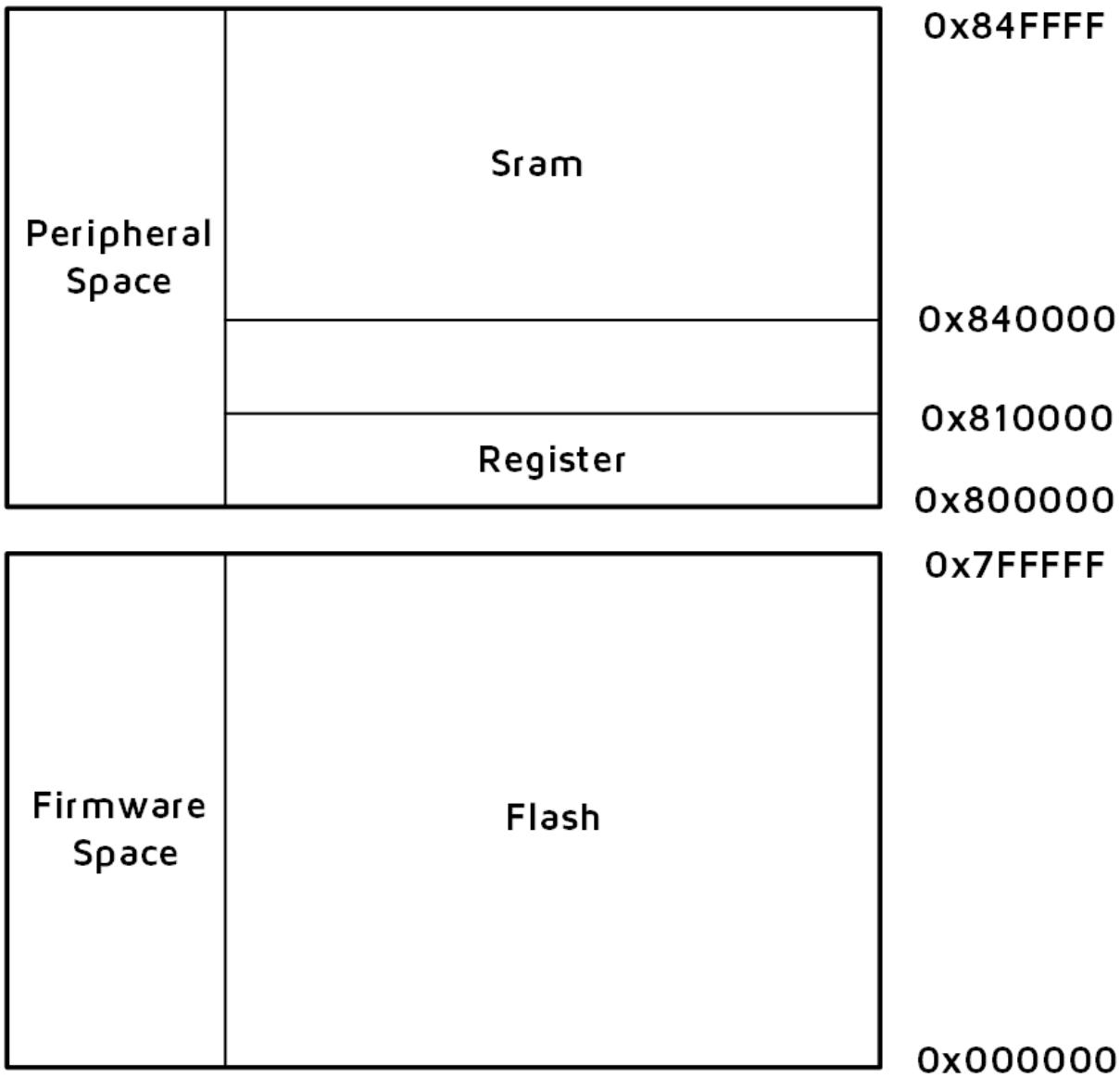


Figure 2.1: MCU 地址空间分配

B80 MCU 物理寻址时，地址线 BIT（23）用于区别程序空间/外设空间：

- 该地址为 0 时，访问程序空间；
- 该地址为 1 时，访问外设空间。

寻址空间为外设空间（BIT(23) 为 1）时，地址线 BIT（18）用于区别 Register 和 Sram：

- 该地址为 0 时，访问 Register；
- 该地址为 1 时，访问 Sram。

2.1.2 SRAM 空间分配

B80 SRAM 的空间分配和低功耗管理部分的 deepsleep retention 功能密切相关，请用户先掌握了解 deepsleep retention 相关知识。

2.1.2.1 SRAM 和 Firmware 空间

对 MCU 地址空间中 SRAM 空间的分配做进一步说明。

16kB Sram 地址空间范围为 0x840000 ~ 0x844000，对应的 Sram 和 Firmware 空间分配如下图所示。

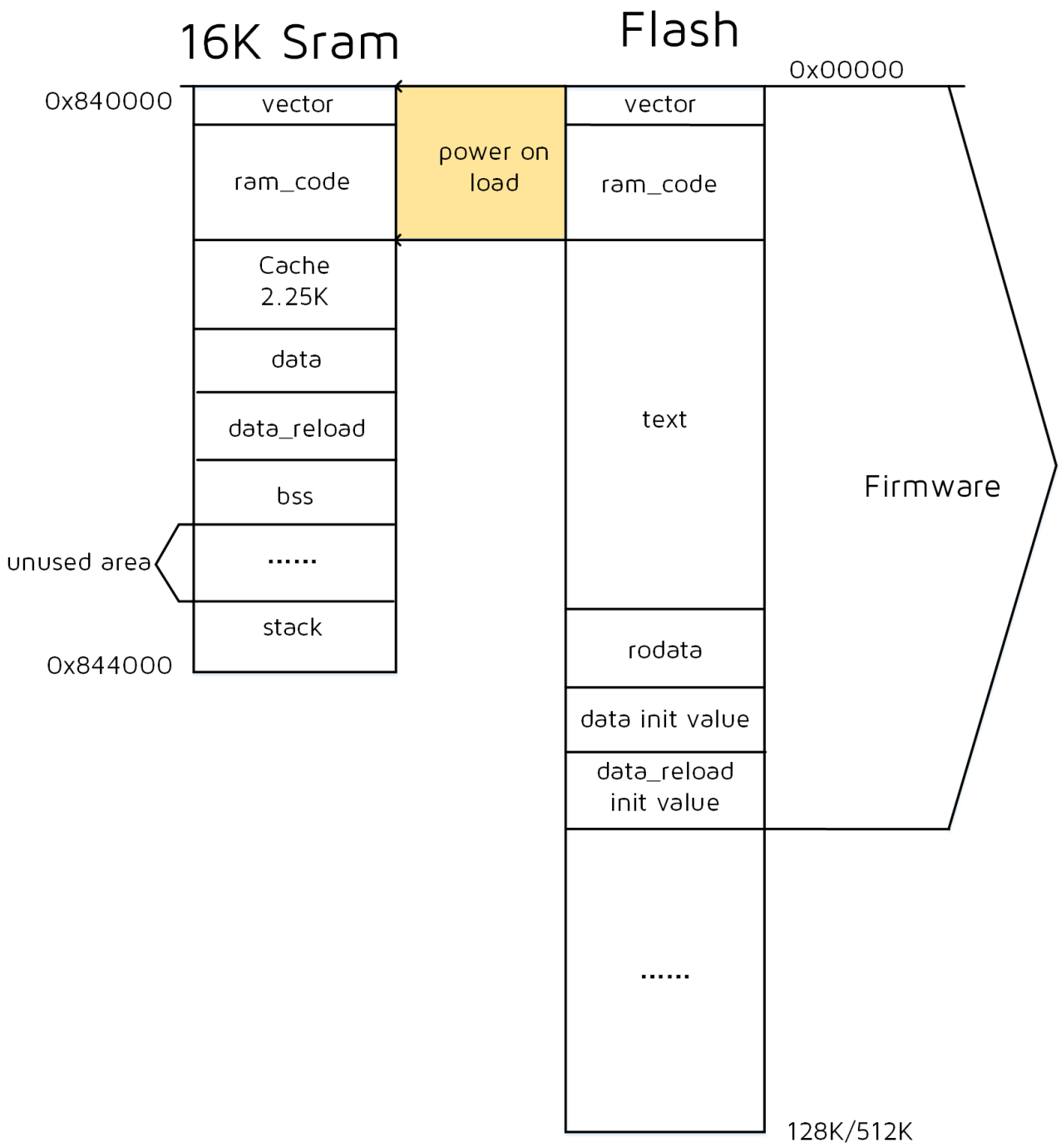


Figure 2.2: Sram 空间分配 & Firmware 空间分配

SDK 中 Sram 空间分配相关的文件有 boot.link 和 cstartup_flash.S。

Flash 中 Firmware 包括 vector、ramcode、retention_data、text、rodata、data init value 和 data_reload init value。

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

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

(1) vectors and ram_code

"vectors"段是汇编文件 cstartup_flash.S 对应的程序，是软件启动代码（software bootloader）。

"ramcode"段是 Flash Firmware 中需要常驻 Ram 的 code,对应 SDK 中所有加了关键字"attribute_ram_code"或者"attribute_ram_code_sec_noinline"的函数，比如 flash_send_cmd 函数：

_attribute_ram_code_sec_noinline_ static void flash_send_cmd(unsigned char cmd);

函数常驻内存有两个原因：

一是某些函数由于涉及到和 Flash MSPI 四根管脚的时序复用，必须常驻内存，如果放到 flash 中就会出现时序冲突，造成死机，如 flash 操作相关所有函数；

二是放到 ram 中执行的函数每次被调用时不需要从 flash 重新读取，可以节省时间，所以对于一些执行时间有要求的函数可以放到常驻内存，提高执行效率。SDK 中将 BLE 时序相关的一些经常要执行的函数常驻到内存，大大降低执行时间，最终达到节省功耗。

用户如果需要将某个函数常驻内存，可以仿照上面 flash_send_cmd，在自己的函数上添加关键字，编译之后就能在 list 文件中看到该函数在 ramcode 段了。

Firmware 中的 vector 和 ramcode 都需要在 MCU 上电时全部搬到 ram 上，编译之后，这两部分加起来的 size 为 _ramcode_size_。_ramcode_size_ 是一个编译器能够认识的变量值，它的计算实现在 boot.link 中，如下所示，编译的结果 _ramcode_size_ 等于 vector 和 ramcode 所有 code 的 size。

```
. = 0x0;
    .vectors :
    {
    *(.vectors)
    *(.vectors.*)
    }
    .ram_code :
    {
    *(.ram_code)
    *(.ram_code.*)
    }
PROVIDE(_ramcode_size_ = . );//计算实际的 ramcode size(vector + ramcode)
```

(2) Cache

Cache 是 MCU 的指令高速缓存，且必须被配置为 Sram 中的一段才可以正常运行。Cache size 是固定的，包括 256 字节的 tag 和 2048 字节的 Instructions cache，总共 0x900 = 2.25K。

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

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

Cache 大小是固定的 2.25K，它在 Sram 中的起始地址为可配置的。

(3) data

"data" 段是 Sram 中存放程序已经初始化的全局变量，即 initial value 非 0 的全局变量。该部分只会在上电或者 deep sleep 唤醒后重新初始化，在 deepsleep retention 和 suspend 期间均保持睡眠前的值。

"data"段紧跟 Cache，起始地址即 Cache 的结束地址。下面为 boot.link 中的代码，直接定义 Sram 上"data"段开始的地址：

```
    . = (0x840000 + ((0x900 + _ramcode_size_align_256_) * __LOAD_FLASH) + (0x400 * __LOAD_DUT)
↪   + (_dstored_ * __LOAD_RAM));
    .data :
```

"data"段是被初始化的全局变量，其初值需要提前存储在 flash 中，即图中所示 Firmware 中的"data initial value"。

(4) data_reload

为了节省 retention 的唤醒起来的 boot 时间，8208 中"data" 段和"bss" 段被设置为 retention 保留，retention 起来不再初始化。另一方面，有些变量就是 retention 回来是有初始化需要的，因此我们增加了"data_reload"段。这个段的特点是：其在 retention 回来后会重新初始化，不再保留睡眠之前的值。

"data_reload"段紧跟"data"段，起始地址即"data"段的结束地址。下面为 boot.link 中的代码，直接定义 Sram 上"data_reload"段开始的地址：

```
    PROVIDE(_rstored_ = _dstored_ + _end_data_ - _start_data_);

    .data_reload :
    AT ( _rstored_ )
```

"data_reload"段是被初始化的全局变量，其初值需要提前存储在 flash 中，即图中所示 Firmware 中的"data_reload init value"。

(5) bss

"bss" 段是 Sram 中存放程序未初始化的全局变量，即 initial value 为 0 的全局变量。该部分只会在上电或者 deep sleep 唤醒后重新初始化，在 deepsleep retention 和 suspend 期间均保持睡眠前的值。

"bss" 段紧跟"data_reload"段，起始地址即"data_reload"段的结束地址。下面为 boot.link 中的代码，直接定义 Sram 上"bss"段开始的地址：

```
    PROVIDE(_data_reload_end_ = . );
    ......
    .bss :
    {
    . = (((. + 3) / 4)*4);
    PROVIDE(_start_bss_ = .);
    ......
```

(6) stack / unused area

对于 16K 的 Sram ，"stack"是从最高地址 0x844000 开始的，其方向为从下往上延伸，即 stack 指针 SP 在数据入栈时自减，数据出栈时自加。

默认情况下，SDK library 使用了 stack 的 size 不超过 256 byte，但由于 stack 的使用 size 取决于 stack 最深位置的地址，所以最终 stack 的使用量跟用户上层的程序设计是有关的。如果用户使用了比较麻烦的递归函数调用，或者在函数里使用了比较大的局部数组变量，或者其他有可能造成 stack 比较深的情况，都会导致最终 stack 的使用 size 变大。

当用户的 Sram 使用较多时，需要明确知道自己的程序使用了多少 stack，这个无法通过 list 文件来分析，只能让应用程序运行起来，确保其运行了程序中所有的可能使用 stack 比较深的 code 后，将 MCU reset，读取 Sram 空间去确定 stack 的使用量。

"unused area"即 bss 段结束地址与 stack 最深地址之间剩余的空间。只有当这个空间存在时，才说明 stack 没有和 bss 冲突，Sram 使用没有问题。如果 stack 最深的地方和 bss 段重合了，则说明 Sram 不够用了。

通过 list 文件可以查出 bss 段结束的地址，也就确定了留给 stack 的最大空间，用户需要分析这个空间是否足够，结合上面说的查看 stack 最深地址，可以知道 Sram 的使用是否超出。下面 demo 中会给出分析方法。

(7) text

"text"段是 Flash Firmware 中所有非 ram_code 函数的集合。程序中的函数如果加了"_attribute_ram_code_"或者"attribute_ram_code_sec_noinline"，会被编译为 ram_code 段，其他没有加这个关键字的函数就全部编译到了"text"段。一般情况下，"text" 段是 Firmware 中最大的空间，远大于 Sram 的 size，所以需要通过 Cache 的缓存功能，将需要执行的 code 先 load 到 Cache 中再可以被执行。

(8) rodata /data init value /data_reload init value

Flash Firmware 中除了 vector、ram_code 和 text，剩余的数据为"rodata" 段、"data init value" 和"data_reload init value"。

"rodata"段是程序中定义的可读、不能改写的数据，是用关键字"const"定义的变量。比如 Slave 中的 ATT table：

```
static const attribute_t my_Attributes[] = ……
```

用户可以在对应的 list 文件中看到"my_Attributes"是在 rodata 段之内。

前面介绍的"data"段是程序中已初始化的全局变量，比如定义全局变量如下：

```
int   testValue = 0x1234;
```

那么编译器就会将初值 0x1234 存放到"data initial value"中，在上电或者 deepsleep 唤醒时，会将该初值拷贝到 testValue 对应的内存地址。

前面介绍的"data_reload"段是程序中定义为 retention 回来重新初始化的全局变量，比如定义全局变量如下：

```
_attribute_data_reload_ int   reloadValue = 0x1234;
```

那么编译器就会将初值 0x1234 存放到"data_reload init value"中，在运行 bootloader 时，会将该初值拷贝到 reloadValue 对应的内存地址。

2.1.2.2 list 文件分析 demo

这里以 8208 ble sample 的默认配置为例，结合"Sram 空间分配 & Firmware 空间分配"来分析。

以下分析中，会出现多处截图，均来自 boot.link、cstartup_flash.S、8208 ble sample.bin 和 8208 ble sample.list，请用户自行查找文件找到截图对应位置。

list 文件中各个 section 的分布情况如下图所示（注意 Algn 字节对齐）：

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .vectors      00000210  00000000  00000000  00008000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .ram_code     00001e5c  00000210  00000210  00008210  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .text         00007098  00002100  00002100  0000a100  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .rodata       00000800  00009198  00009198  00011198  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         00000174  00842a00  00009998  00012a00  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000980  00842b80  00009b18  00012b74  2**4
                  ALLOC
  6 .TC32.attributes 00000010  00000000  00000000  00012b74  2**0
                  CONTENTS, READONLY
  7 .comment      0000001a  00000000  00000000  00012b84  2**0
                  CONTENTS, READONLY
```

Figure 2.3: list 文件 section 统计

根据 section 统计先将需要了解的信息列出来，和后面具体的 section 介绍都会一一对应。

(1) vectors：从 Flash 0 开始，Size 为 0x210，计算出结束地址为 0x210；

(2) ram_code：从 Flash 0x210 开始，Size 为 0x1e5c，计算出结束地址为 0x2100；

(3) text：从 Flash 0x2100 开始，Size 为 0x7098，计算出结束地址为 0x9198；

(4) rodata：从 Flash 0x9198 开始，Size 为 0x800，计算出结束地址为 0x9998；

(5) data：从 Sram 0x842a00 开始，Size 为 0x174，计算出结束地址为 0x842b74；

(6) bss：从 Sram 0x842b80 开始（因为对齐原因不是从 0x842b74 开始），Size 为 0x980，计算出结束地址为 0x843500。

结合前面介绍可知，剩余 Sram 空间为 0x844000 –0x843500= 0xb00 = 2816 byte，减去 stack 需要使用的 256 byte，还剩 2560 byte。

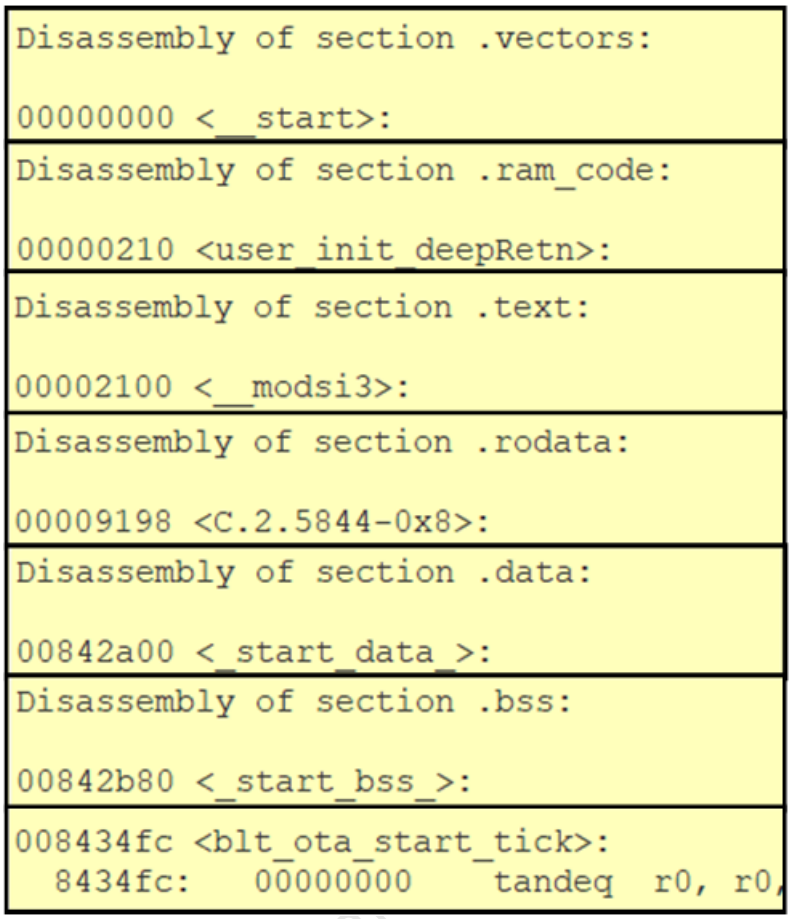


Figure 2.4: list 文件 section 地址

上图为在 list 文件中搜索"section"查到的部分 section 的起始地址后的拼图，结合该图和上面的"list 文件 section 统计"，分析如下：

(1) vector:

"vectors"段在 flash firmware 中起始地址为 0，结束地址为 0x210，size 为 0x210。上电搬移到 Sram 后，在 Sram 上的地址为 0x840000 ~ 0x840210。

(2) ram_code:

"ram_code"段在 flash firmware 中起始地址为 0x210，结束地址为 0x2100。上电搬移到 Sram 后，在 Sram 上的地址为 0x840210 ~ 0x842100。

(3) Cache:

Cache 在 Sram 中地址范围为：0x842100~0x842a00。Cache 的相关信息在 list 文件中不会体现出来。

(4) text:

"text"段在 flash firmware 中起始地址为 0x2100，结束地址为 0x9198，Size 为 0x9198- 0x2100 = 0x7098，和前面 Section 统计中数据一致。

(5) rodata:

"rodata"段起始地址为 text 的结束地址 0x9198，结束地址为 0x9998。

(6) data:

"data"段在 Sram 上起始地址为 Cache 的结束地址 0x842a00，list 文件 Section 统计部分给出的 size 为 0x174。"data"段在 Sram 上的结束地址为 0x842b74。

(7) bss:

"bss"段在 Sram 上起始地址为"data"段的结束地址 0x842b80（16 字节对齐），list 文件 Section 统计部分给出的 size 为 0x980。

"bss"段在 Sram 上的结束地址为 0x843500。

剩余 Sram 空间为 0x844000 – 0x843500 = 0xb00 = 2816 byte，减去 stack 需要使用的 256 byte，还剩 2560 byte。

2.1.3 MCU 地址空间访问

程序中对 0x000000 - 0xFFFFFF 地址空间的访问分以下两种情况。

2.1.3.1 外设空间的读写操作

外设空间（register 和 sram）的读写操作直接用指针访问实现：

```
u8  x = *(volatile u8*)0x800066;  //读 register 0x66 的值
 *(volatile u8*)0x800066 = 0x26;  //给 register 0x66 赋值
u32 y = *(volatile u32*)0x840000;      //读 sram 0x40000-0x40003 地址的值
*(volatile u32*)0x840000 = 0x12345678; //给 sram 0x40000-0x40003 地址赋值
```

程序中使用函数 write_reg8、write_reg16、write_reg32、read_reg8、read_reg16、read_reg32 对外设空间进行读写，其实质是指针操作。更多信息，请参照 drivers/bsp.h。

注意程序中类似 write_reg8(0x40000)/ read_reg16(0x40000) 的操作，其定义如下所示，从中看出 0x800000 的偏移已自动加上（地址线 BIT(23) 为 1），所以 MCU 能够确保访问的是 Register/Sram 空间，而不会去访问 flash 空间。

```
#define REG_BASE_ADDR           0x800000

#define REG_ADDR8(a)            (*(volatile unsigned char*) (REG_BASE_ADDR | (a)))
#define REG_ADDR16(a)           (*(volatile unsigned short*)(REG_BASE_ADDR | (a)))
#define REG_ADDR32(a)           (*(volatile unsigned long*) (REG_BASE_ADDR | (a)))

#define write_reg8(addr,v)      (*(volatile unsigned char*)  (REG_BASE_ADDR | (addr)) =
 ↪  (unsigned char)(v))
#define write_reg16(addr,v)     (*(volatile unsigned short*) (REG_BASE_ADDR | (addr)) =
 ↪  (unsigned short)(v))
#define write_reg32(addr,v)     (*(volatile unsigned long*)  (REG_BASE_ADDR | (addr)) = (v))

#define read_reg8(addr)         (*(volatile unsigned char*) (REG_BASE_ADDR | (addr)))
```

```
#define read_reg16(addr)        (*(volatile unsigned short*)(REG_BASE_ADDR | (addr)))
#define read_reg32(addr)        (*(volatile unsigned long*) (REG_BASE_ADDR | (addr)))
```

这里注意一个内存对齐的问题：如果使用指向 2 字节/4 字节的指针来读写外设空间，一定要确保地址是 2 字节/4 字节对齐的，如果不对齐的话，会发生数据读写错误。如下两种是错误的：

```
u16  x = *(volatile u16*)0x840001;     //0x840001 没有 2 字节对齐
*(volatile u32*)0x840005 = 0x12345678;  //0x840005 没有 4 字节对齐
```

修改为正确的读写操作：

```
u16  x = *(volatile u16*)0x840000;     //0x840000  2 字节对齐
*(volatile u32*)0x840004 = 0x12345678;  //0x840004  4 字节对齐
```

2.1.3.2 Flash 的操作

此部分内容详见第 8 章 Flash 介绍。

2.1.4 SDK Flash 空间的分配

此部分内容详见第 8 章 Flash 介绍。

2.2 时钟模块

2.2.1 System Clock & System Timer

系统时钟（system clock）是 MCU 执行程序的时钟。

系统定时器（System Timer）是一个只读的定时器，为 BLE 的时序控制提供时间基准，同时也可以提供给用户使用。

在 Telink 上一代 IC（826x 系列）上，System Timer 的时钟来源于 system clock，而 B80 系列 IC 上，System Timer 和 system clock 是独立分开的。如下图所示，System Timer 是由外部 24M Crystal Oscillator 经分频得到的 16M。

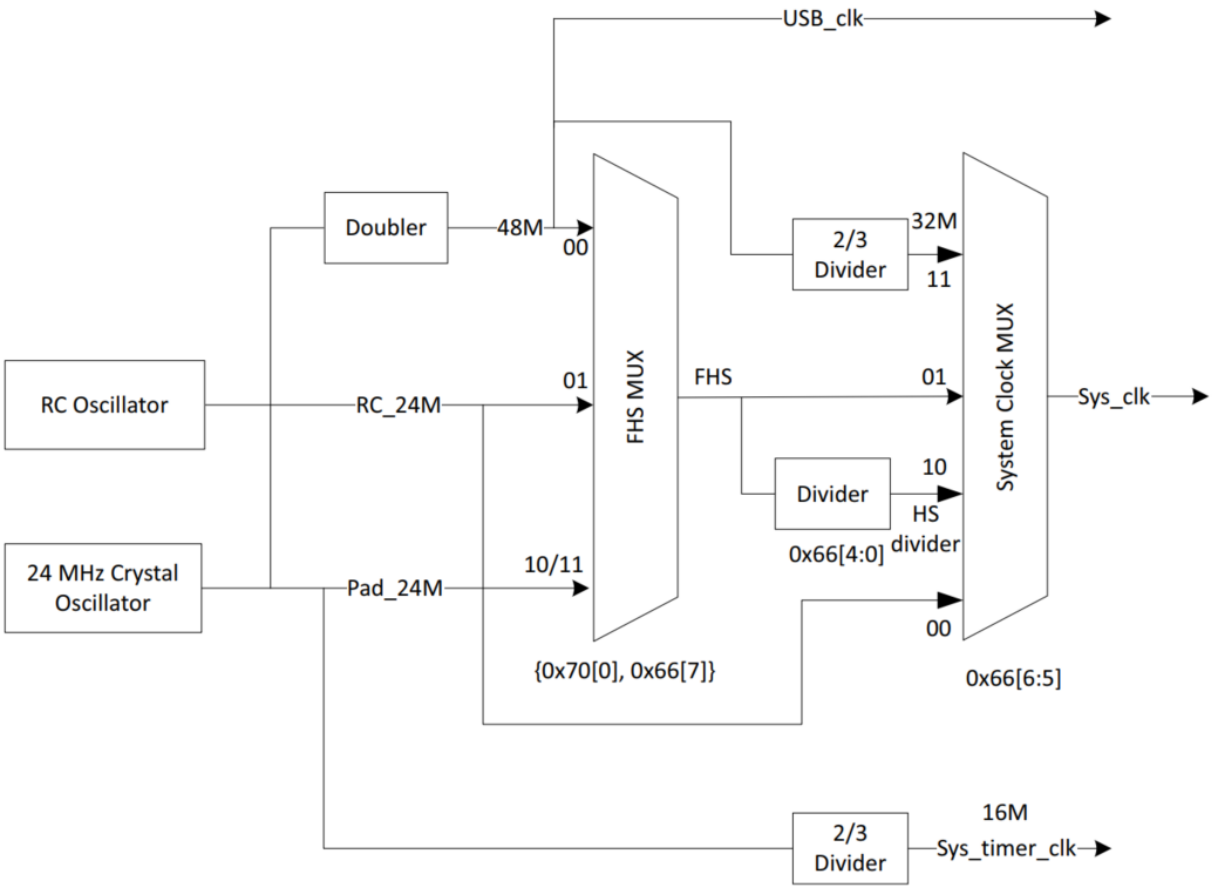


Figure 2.5: System clock & System Timer

图上可以看到，system clock 可以由外部 24M 晶体振荡器经"doubler"电路倍频到 48M 后再分频得到 16M、24M、32M、48M 等，这一类 clock 称为 crystal clock (如 16M crystal system clock、24M crystal system clock)；也可以由 IC 内部 24M RC Oscillator 处理后得到 24M RC clock、32M RC clock、48M RC clock 等。这一类称为 RC clock（BLE SDK 不支持 RC clock)。

在 BLE SDK 中，我们推荐使用 crystal clock。

初始化时调用下面的 API 配置 system clock，在枚举变量 SYS_CLK_TYPEDEF 定义中选择时钟对应的 clock 即可。

```
void clock_init(SYS_CLK_TypeDef SYS_CLK)
```

由于 B80 系列芯片的 System Timer 与 system clock 不一样，用户需要了解 MCU 上各硬件模块的 clock 是来源于 system clock 还是 System Timer。我们以 system clock 为 24M crystal 的情况来进行说明，此时 system clock 为 24M，而 System Timer 是 16M。

在文件 app_config.h 中 system clock 以及对应的 s、ms、us 的定义如下：

```
#define CLOCK_SYS_CLOCK_HZ      24000000
enum{
        CLOCK_SYS_CLOCK_1S = CLOCK_SYS_CLOCK_HZ,
        CLOCK_SYS_CLOCK_1MS = (CLOCK_SYS_CLOCK_1S / 1000),
        CLOCK_SYS_CLOCK_1US = (CLOCK_SYS_CLOCK_1S / 1000000),
};
```

所有时钟源为 system clock 的硬件模块，在设置模块的 clock 时，只能使用上面 CLOCK_SYS_CLOCK_HZ、CLOCK_SYS_CLOCK_1S 等；换言之，如果用户看到模块中 clock 的设置使用的是以上几个定义，说明该模块的时钟源为 system clock。

如 PWM 驱动中 PWM 周期和占空的设置如下，说明 PWM 的时钟源是 system clock。

```
pwm_set_cycle_and_duty(PWM0_ID, (u16) (1000 * CLOCK_SYS_CLOCK_1US),  (u16) (500 *
 ↪  CLOCK_SYS_CLOCK_1US) );
```

System Timer 是固定的 16M，所以对于这个 timer，SDK code 中使用如下的数值来表示 s、ms 和 us。

```
//system timer clock source is constant 16M, never change
enum{
        CLOCK_16M_SYS_TIMER_CLK_1S =  16000000,
        CLOCK_16M_SYS_TIMER_CLK_1MS = 16000,
        CLOCK_16M_SYS_TIMER_CLK_1US = 16,
};
```

SDK 中以下几个 API 都是跟 System Timer 相关的一些操作，所以涉及到这些 API 操作时，都使用上面类似"CLOCK_16M_SYS_TIMER_CLK_xxx" 的方式来表示时间。

```
void sleep_us (unsigned long microsec);
unsigned int clock_time(void);
int clock_time_exceed(unsigned int ref, unsigned int span_us);
#define ClockTime           clock_time
#define WaitUs              sleep_us
#define WaitMs(t)           sleep_us((t)*1000)
```

由于 System Timer 是 BLE 计时的基准，SDK 中所有 BLE 时间相关的参数和变量，在涉及到时间的表达时，都是用"CLOCK_16M_SYS_TIMER_CLK_xxx" 的方式。

2.2.2 System Timer 的使用

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

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

MCU 在运行程序过程中 system tick 不会停止。

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

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

BLE SDK 整个 BLE 时序都是基于 System Timer tick 设计的，程序中也大量使用这个 System Timer tick 来完成各种计时和超时判断，强烈推荐 user 使用这个 System Timer tick 来实现一些简单的定时和超时判断。

比如要实现一个简单的软件定时。软件定时器的实现基于查询机制，由于是通过查询来实现，不能保证非常好的实时性和准备性，一般用于对误差要求不是特别苛刻的应用。实现方法：

(1) 启动计时：设置一个 u32 的变量，读取并记录当前 System Timer tick。

```
u32 start_tick = clock_time();   // clock_time() returns System Timer tick value
```

(2) 在程序的某处不断查询当前 System Timer tick 和 start_tick 的差值是否超过需要定时的时间值。若超过，认为定时器触发，执行相应的操作，并根据实际的需求清除计时器或启动新一轮的定时。

假设需要定时的时间为 100 ms，查询计时是否到达的写法为：

```
if( (u32) ( clock_time() - start_tick) >  100 * CLOCK_16M_SYS_TIMER_CLK_1MS)
```

由于将差值转化为 u32 型，解决了 system clock tick 从 0xffffffff 到 0 这个极限情况。

实际上 SDK 中为了解决系统时钟不同导致和 u32 转换的问题，提供了统一的调用函数，不管系统时钟多少，都可以下面函数进行查询判断：

```
if( clock_time_exceed(start_tick, 100 * 1000))  //第二个参数单位为 us
```

注意：

> 由于 16MHz 时钟转一圈为 268 秒，这个查询函数只适用于 268 秒以内的定时。如果超过 268 秒，需要在软件上加计数器累计实现（这里不介绍）。

应用举例：A 条件触发（只会触发一次）的 2 秒后，程序进行 B() 操作。

```
u32  a_trig_tick;
int  a_trig_flg = 0;
while(1)
{
        if(A){
            a_trig_tick = clock_time();
            a_trig_flg = 1;
        }
        if(a_trig_flg && clock_time_exceed(a_trig_tick,2 *1000 * 1000)){
            a_trig_flg = 0;
            B();
        }
}
```

2.3 GPIO 模块

GPIO 模块的说明请 user 对照 drivers/gpio.h、gpio_default.h、gpio.c 来理解，所有代码都以源码形式提供。

代码中涉及到对寄存器的操作，请参考 datasheet 来理解。

2.3.1 GPIO 定义

B80 系列芯片最多支持 5 组 38 个 GPIO，分别为：GPIO_PA0 ~ GPIO_PA7、GPIO_PB0 ~ GPIO_PB7、GPIO_PC0 ~ GPIO_PC7、GPIO_PD0 ~ GPIO_PD7、GPIO_PE0 ~ GPIO_PE3 和 GPIO_PF0 ~ GPIO_PF1。

注意：

> IC 的 core 部分都是有这 38 个 GPIO，但具体到每一颗 IC 的不同封装上，可能有些 GPIO 并没有引出，所以使用 GPIO 时请以 IC 实际封装出来的 GPIO 管脚为准。

程序中需要使用 GPIO 时，必须按照上面的写法定义，详情见 drivers/gpio.h。

注意：

> 7 个 GPIO 比较特殊，需要注意：
> a）MSPI 的 4 个 GPIO。这 4 个 GPIO 是 MCU 系统总线中主 SPI 总线，用于读写 flash 操作，上电默认为 spi 状态，user 永远不能操作它们，程序中不能使用。这个 4 个 GPIO 为 PE0、PE1、PE2、PE3。
> b）SWS (Single Wire Slave)，用于 debug 和烧写 firmware，上电默认为 SWS 状态，程序中一般不使用。B80 的 SWS 管脚为 PA3。
> c）DM 和 DP，上电默认为 GPIO 状态。当需要 USB 功能时，DM 和 DP 需要使用；当不需要 USB 时，可以作为 GPIO 使用。B80 的 DM、DP 管脚为 PA1、PA2。

2.3.2 GPIO 状态控制

这里只列举用户需要了解的最基本的 GPIO 状态。

(1) func(功能配置：特殊功能/一般 GPIO)，如需要使用输入输出功能，需配置为一般 GPIO。

```
void gpio_set_func(GPIO_PinTypeDef pin, gpio_func_e func);
```

pin 为 GPIO 定义，以下一样。func 可选择 AS_GPIO 或其他特殊功能。

(2) ie(input enable)：输入使能

```
void gpio_set_input_en(GPIO_PinTypeDef pin, unsigned int value);
```

value：1 和 0 分别表示 enable 和 disable。

(3) datai (data input)：当输入使能打开时，该值为当前该 GPIO 管脚的电平，用于读取外部电压。

```
static inline _Bool gpio_read(GPIO_PinTypeDef pin);
```

读到低电压返回值为 0；读到高电压返回值为 1。

```
if( !gpio_read(GPIO_PA0) )  //判断高低电平
```

(4) oe(output enable)：输出使能

```
static inline void gpio_set_output_en(GPIO_PinTypeDef pin, unsigned int value);
```

value 1 和 0 分别表示 enable 和 disable。

(5) datao (data output)：当输出使能打开时，该值为 1 输出高电平，为 0 输出低电平。

```
static inline void gpio_write(GPIO_PinTypeDef pin, unsigned int value);
```

(6) 内部模拟上下拉电阻配置：有 1M 上拉、10K 上拉、100K 下拉 3 种模拟电阻，可配置的状态有 4 种：1M 上拉、10K 上拉、100K 下拉和 float 状态。

```
void gpio_setup_up_down_resistor(GPIO_PinTypeDef gpio, GPIO_PullTypeDef up_down);
```

up_down 的四种配置为：

```
typedef enum {
    PM_PIN_UP_DOWN_FLOAT    = 0,
    PM_PIN_PULLUP_1M        = 1,
    PM_PIN_PULLDOWN_100K    = 2,
    PM_PIN_PULLUP_10K       = 3,
}GPIO_PullTypeDef;
```

在 deepsleep 和 deepsleep retention 状态下，GPIO 的输入输出状态全部失效，但是模拟上下拉电阻仍然有效。

注意：

> 关于 GPIO 在进行功能改变配置的时候，需要按照如下顺序：
> a）开始就是 GPIO 功能，那么需要先将所需要的功能 MUX 配置好后，最后再将 GPIO 功能 disable。
> b）开始是功能 IO，需要改成 GPIO output，先将对应的 IO 的 output 值和 OEN 设置对后，最后 enable GPIO 功能。
> c）开始功能是 IO, 需要改成 GPIO input 并且 IO 上拉，先将 output 设置成 1，OEN 设置为 1（对应 PA 和 PD），其次将 pullup 设置为 1（对应 PB 和 PC），最后 enable GPIO 功能。
> d）将 pullup 设置为 1（对应 PB 和 PC）并且 IO 不上拉，先将 output 设置成 0，OEN 设置为 1（对应 PA 和 PD），其次将 pullup 设置为 0（对应 PB 和 PC），最后 enable GPIO 功能。

GPIO 配置应用举例：

(1) 将 GPIO_PA4 配置为输出态，并输出高电平。

```
gpio_set_func(GPIO_PA4, AS_GPIO) ;  // PA4 默认为 GPIO 功能，可以不设置
gpio_set_input_en(GPIO_PA4, 0);
gpio_set_output_en(GPIO_PA4, 1);
gpio_write(GPIO_PA4,1)
```

(2) 将 GPIO_PC6 配置为输入态，判断是否读到低电平，需要开启上拉，防止 float 电平的影响。

```
gpio_set_func(GPIO_PC6, AS_GPIO) ;  // PC6 默认为 GPIO 功能，可以不设置
gpio_setup_up_down_resistor(GPIO_PC6, PM_PIN_PULLUP_10K);
    gpio_set_input_en(GPIO_PC6, 1)
    gpio_set_output_en(GPIO_PC6, 0);
    if(!gpio_read(GPIO_PC6)){  //是否低电平
        ......
    }
```

(3) 将 PA5、PA6 脚配置成 USB 功能。

```
    gpio_set_func(GPIO_PA2, 0);//DP
    gpio_set_func(GPIO_PA1, 0);//DM
    gpio_set_input_en(GPIO_PA2|GPIO_PA1,1);//DP/DM must set input enable
```

2.3.3 GPIO 的初始化

main.c 中调用 gpio_init 函数，会将除了 MSPI 4 个 GPIO 以外的其他 32 个 GPIO 的状态都初始化一遍。

当用户的 app_config.h 中没有配置 GPIO 参数时，这个函数会将每个 IO 初始化为默认状态。32 个 GPIO 默认状态为：

(1) func

除了 SWS ，其他均为一般 GPIO 状态。

(2) ie

除了 SWS 默认 ie 为 1，其他所有的一般 GPIO 默认 ie 为 0。

(3) oe

全部为 0。

(4) dataO

全部为 0。

(5) 内部上下拉电阻配置

全部为 float。

更多详情请参照 drivers/gpio.h、drivers/gpio_default.h。

如果在 app_config.h 中有配置到某个或某几个 GPIO 的状态，那么 gpio_init 时不再使用默认状态，而是使用用户在 app_config.h 配置的状态。原因是 gpio 的默认状态是使用宏来表示的，这些宏的写法为（以 PA0 的 ie 为例）：

```
#ifndef PA0_INPUT_ENABLE
#define PA0_INPUT_ENABLE        1
#endif
```

当在 app_config 中可以提前定义这些宏，这些宏就不再使用以上这种默认值。

在 app_config.h 中配置 GPIO 状态方法为（以 PA0 为例）：

(1) 配置 func:

```
#define PA0_FUNC                AS_GPIO
```

(2) 配置 ie：

```
#define PA0_INPUT_ENABLE         1
```

(3) 配置 oe:

```
#define PA0_OUTPUT_ENABLE        0
```

(4) 配置 dataO：

```
#define PA0_DATA_OUT             0
```

(5) 配置内部上下拉电阻：

```
#define  PULL_WAKEUP_SRC_PA0            PM_PIN_UP_DOWN_FLOAT
```

GPIO 的初始化总结：

(1) 可以提前在 app_config.h 中定义 GPIO 的初始状态，在 gpio_init 中得以设定；

(2) 可以在 user_init 函数中通过 GPIO 状态控制函数 (gpio_set_input_en 等) 加以设定；

(3) 也可以使用以上两种方式混用：在 app_config.h 中提前定义一些，gpio_init 加以执行，在 user_init 中设定另外一些。

注意：

> 在 app_config.h 中定义和 user_init 中设定同一个 GPIO 的某个状态为不同的值时，根据程序的先后执行顺序，最终以 user_init 中设定为准。

gpio_init 函数的实现如下，anaRes_init_en 的值决定模拟上下拉电阻是否被设置。

```
void gpio_init(int anaRes_init_en)
{
    ......
    // gpio digital status setting
    if(anaRes_init_en){
    gpio_analog_resistance_init();
    }
}
```

参考文档后面低功耗管理的介绍可知，控制 GPIO 模拟上下拉电阻的寄存器在 deepsleep retention 期间是可以保持不掉电的，所以 GPIO 模拟上下拉电阻的状态能在 deepsleep retention mode 下被维持住。

为了确保 deepsleep retention 唤醒之后 GPIO 模拟上下拉电阻的状态不被改变，在 gpio_init 前先要判断当前是否被 deepsleep retention wake_up，根据这个状态去设置 anaRes_init_en 的值，如下面的 code 所示：

```
int deepRetWakeUp = pm_is_MCU_deepRetentionWakeup();
    gpio_init( !deepRetWakeUp );
```

2.3.4 GPIO 数字状态在 deepsleep retention mode 失效

上面介绍的 GPIO 状态控制中，除了模拟上下拉电阻是由模拟寄存器（analog register）控制，其他所有的状态（func、ie、oe、dataO 等）都是被数字寄存器（digital register）控制的。

参考文档后面低功耗管理的介绍可知，deepsleep retention 期间所有 digital register 的状态掉电丢失。

在 Telink 上一代 826x 系列 IC 上，suspend 期间可以用 gpio output 来控制一些外围设备，但到了 B80 上如果 suspend 被切换为 deepsleep retention mode 后，gpio output 状态失效，无法在 sleep 期间准确的控制外围设备。此时可以使用 GPIO 模拟上下拉电阻的状态来代替实现：上拉 10K 代替 gpio output high，下拉 100K 代替 gpio output low。

注意：

> deepsleep retention 期间的 GPIO 状态控制不要使用上拉 1M（上拉电压可能会比供电电压 VCC 低一些）。另外，上拉 10K 的控制中不要使用 PC0~PC7 的上拉 10K（在 deepsleep retention wake_up 时会有短时间的抖动，产生毛刺），其他 GPIO 上拉 10K 都是可以的。

2.3.5 配置 SWS 上拉防止死机

Telink 所有的 MCU 都使用 SWS（single wire slave）来调试和烧录程序。在最终的应用代码上，SWS 这个 pin 的状态为：

(1) function 上设为 SWS，非 GPIO。

(2) ie =1，只有 input enable 时，才可以收到 EVK 发的各种命令，用来操作 MCU。

(3) 其他的配置：oe、dataO 都为 0。

设为以上状态后，可以随时接收 EVK 的操作命令，但同时也带来一个风险：当整个系统的电源抖动很厉害的时候（如发送红外时，瞬间电流可能会冲到接近 100mA），由于 SWS 处于 float 状态，可能会读到一个错误的数据，误以为是 EVK 发来的命令，这个错误的命令可能会导致程序挂掉。

解决上面问题的方法是，将 SWS 的 float 状态修改为输入上拉。通过模拟上拉 1M 电阻来解决。

B80 的 SWS 和 GPIO_PA3 复用，在 drivers/gpio_default.h 中将 PA3 的 1M 上拉开启即可：

```
#ifndef PULL_WAKEUP_SRC_PA7
#define PULL_WAKEUP_SRC_PA7     PM_PIN_PULLUP_1M  //sws pullup
#endif
```

2.4 系统中断

该文档适用 IC 的硬件中断具有以下两个特点：

(1) 所有中断的优先级一样，MCU 不具备中断嵌套的能力

(2) 所有中断共用同一个中断硬件入口，最终会触发软件 irq_handler 函数，在该函数里面通过读取相关中断的状态位来判断对应中断是否被触发。

上面的特点 1 决定了 MCU 响应中断时是按照先来先处理的方式。当第一个中断未处理完时，新的中断产生，无法被立刻响应，进入等待队列，直到前面的中断处理完毕后才会响应。所以，当有 2 个或 2 个以上的中断时，所有的中断都无法做到实时响应，某一个中断响应延时的时间，取决于这个中断触发时 MCU 是否正在处理其他中断以及处理其他中断花费的时间。如下图所示，由于 IRQ2 触发时 IRQ1 正在处理，必须等到 IRQ1 处理结束后才能响应，IRQ 2 delay 的时间最坏情况就是 IRQ1 process 的最大时间。

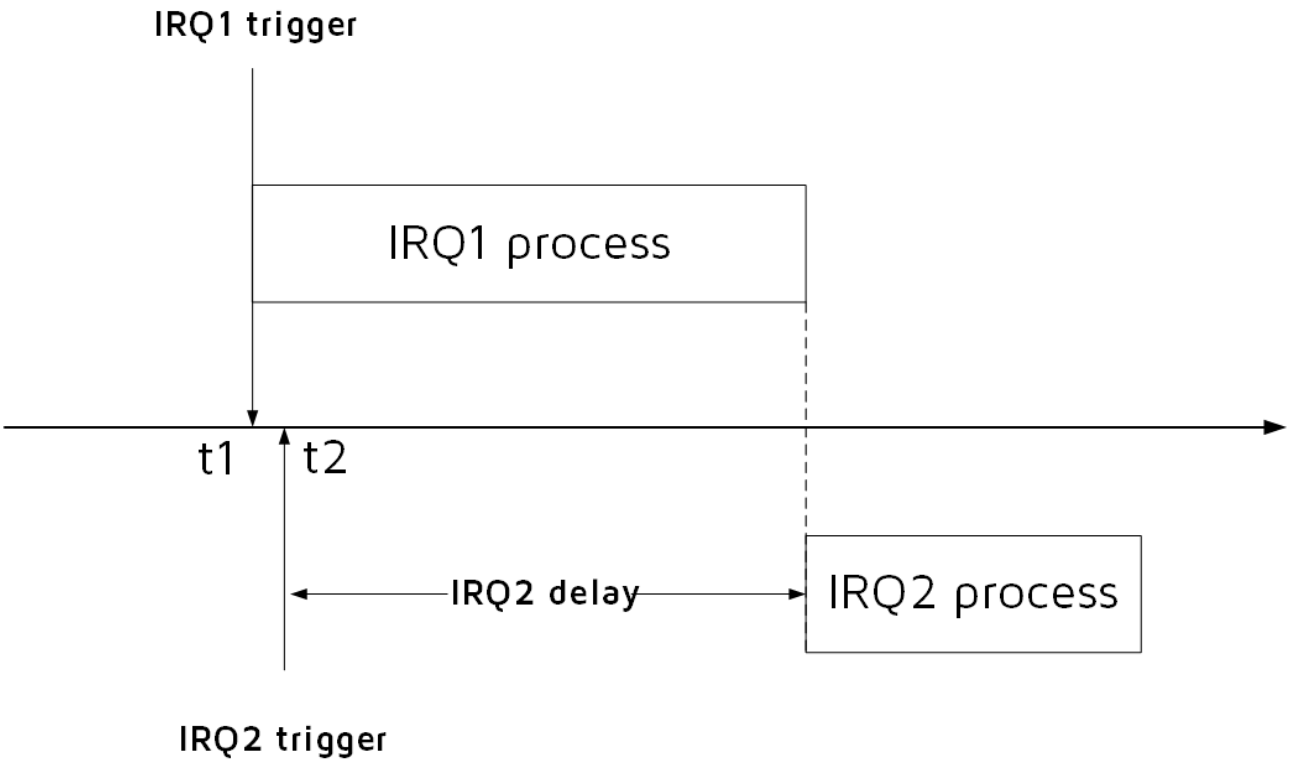


Figure 2.6: IRQ delay

在 BLE SDK 中，使用了 system timer 和 RF 两个系统中断。如果用户没有增加新的中断，则不用考虑两个系统中断的时间问题；如果客户需要增加其他中断（比如 UART、PWM 等），需要考虑的细节如下：

(1) SDK 中 system timer 和 RF 两个系统中断，可能的最大执行时间是 200us。这意味着客户增加的中断可能无法实时响应，且理论上最大可能出现的延迟时间为 200us。

(2) system timer 和 RF 两个系统中断是为了处理 BLE 任务，由于 BLE 的时序较为严格，不能被延时太久。所以客户增加的中断，处理时间不能过长，建议在 50us 以内。如果时间过长，可能出现 BLE 时序同步出错，造成收发包效率低、功耗偏高、BLE 断连等问题。

3 BLE 模块

3.1 BLE SDK 软件架构

3.1.1 标准 BLE SDK 软件架构

根据 BLE spec，一个比较标准的 BLE SDK 架构如下图所示。

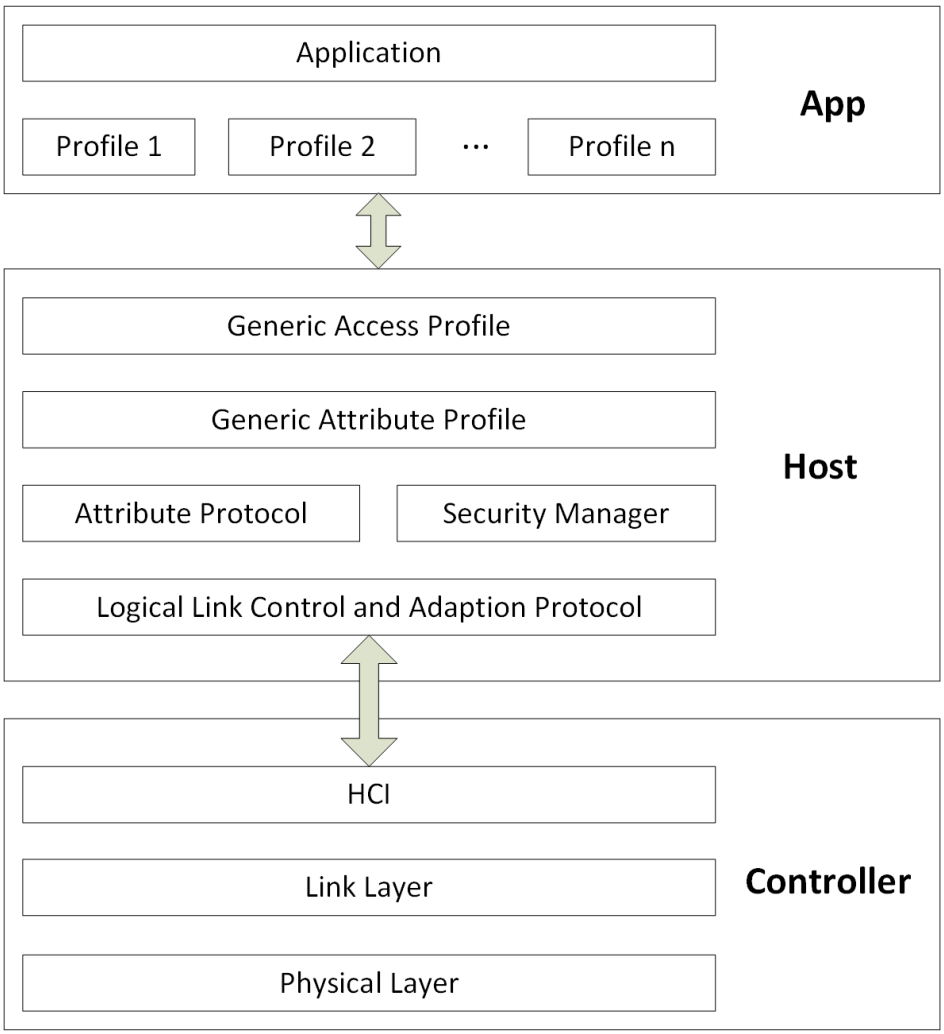


Figure 3.1: BLE SDK 软件架构

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。

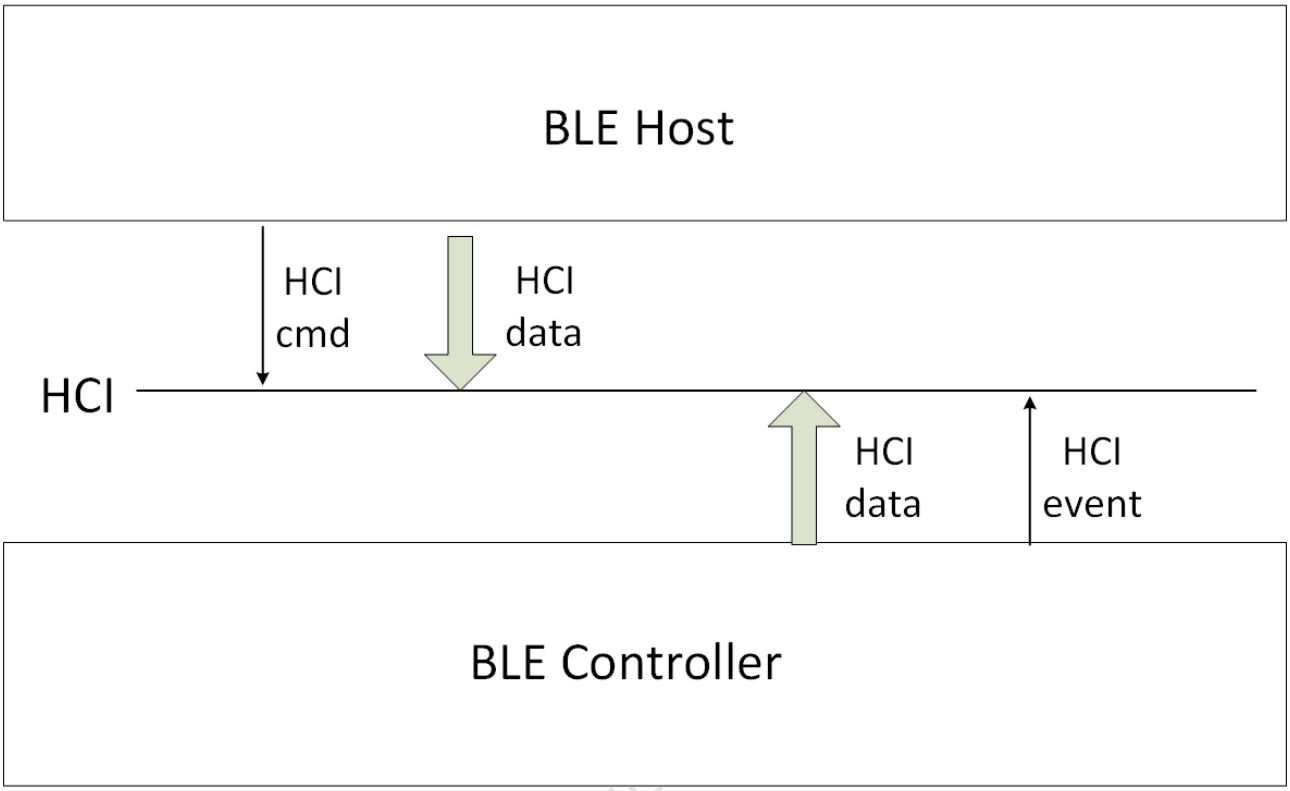


Figure 3.2: Host 和 Controller 的 HCI 数据交互

(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 Controller

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

B80 BLE Single Connection SDK 包含 Link Layer 的 3 种标准状态（standby、advertising、connection），conneciton 状态仅支持 Slave role。B80 BLE Single Connection SDK 中的 Slave role 只是 single connection，即 Link Layer 只能维持一个连接，无法同时多个 Slave 同时存在。

SDK 上的 B80 hci 是一个 BLE slave 的 controller，需要和另外一个运行 BLE Host 的 MCU 协调工作形成一个标准的 BLE Slave 系统，架构图如下。

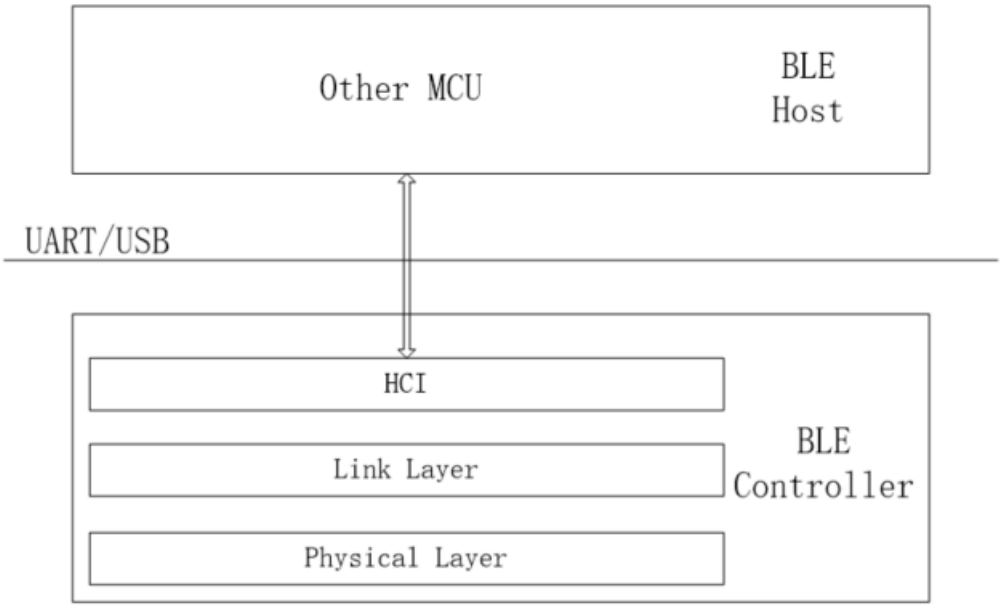


Figure 3.3: Telink HCI 架构

3.1.2.2 Telink BLE Slave

B80 BLE Single Connection SDK 在 BLE host 上，只支持 Slave 部分。

当 user 只需要使用标准的 BLE slave 时，Telink BLE SDK 运行 Host（slave 部分）+ 标准的 Controller，实际的协议栈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 B80 ble sample、B80 module 都是基于该架构。

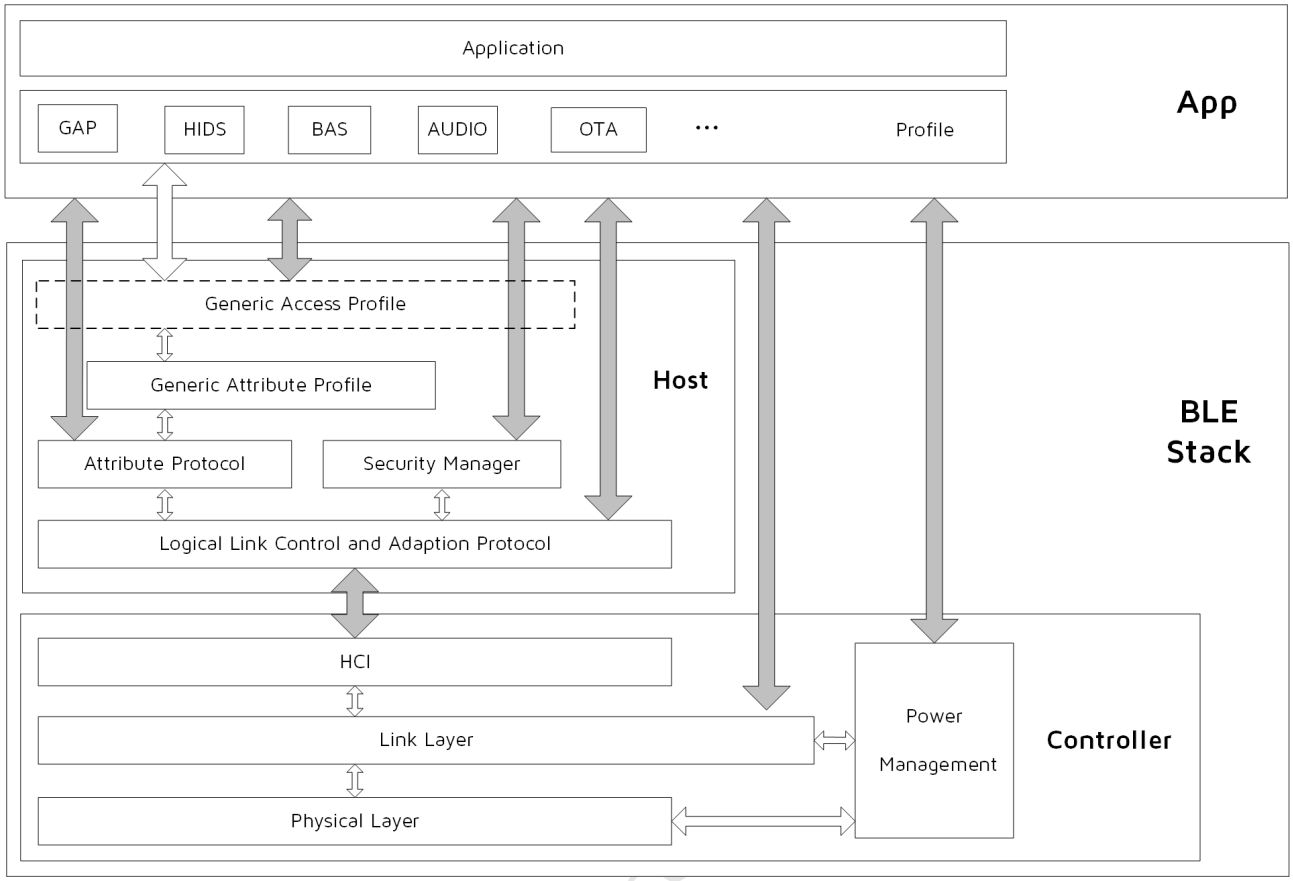


Figure 3.4: Telink BLE Slave 架构

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。

Controller 部分 HCI 仍然是与 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 的事件需要通过 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 demo code 提供几个基本的 profile，包括 HIDS、BAS、OTA 等。

下面基于这个架构对 BLE 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 BLE Controller

3.2.1 BLE Controller 介绍

BLE controller 包括 Physical Layer、Link Layer、HCI 和 Power Management。

Telink BLE SDK 对 Physical Layer（对应 driver 文件中的 rf.h 的 c 文件）完全封装到 library 中，user 不需要了解。Power Management 将在本文档后面的章节中详细介绍，本章只稍微提一下，不做过多介绍。

本章只介绍 Link Layer 和 HCI，且由于 HCI 主要是 interface，其实现的功能也都是为了操作 Link Layer 和获取 Link Layer 的数据，所以重点详细介绍 Link layer，并在介绍 Link Layer 和 API 的过程中对 HCI 相关 interface 一一描述。

3.2.2 Link Layer 状态机

下图为 BLE spec 中 Link Layer 状态机。

更多信息请参照《Core_v4.2》(Vol 6/Part B/1.1 "LINK LAYER STATES")。

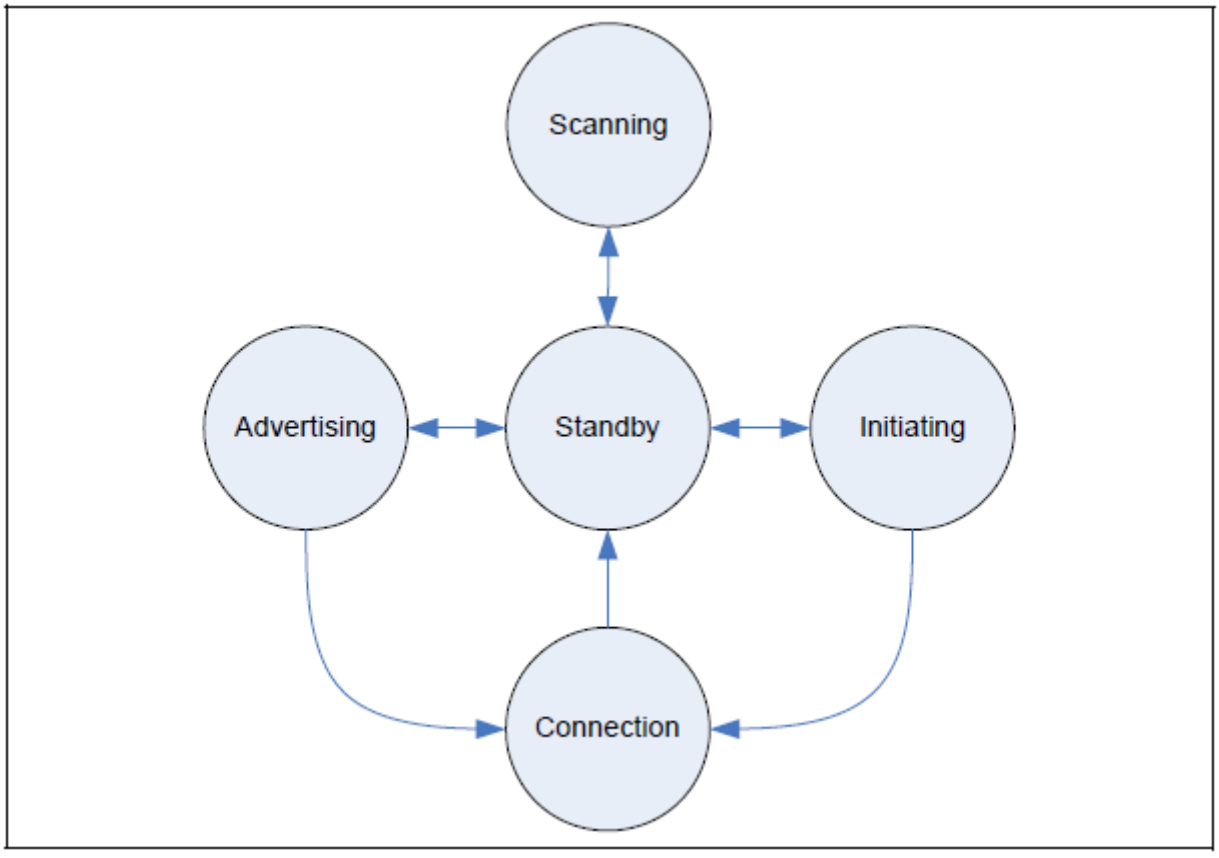


Figure 3.5: BLE Spec 中 Link Layer 状态机

B80 BLE Single Connection SDK Link Layer 状态机如下图所示。

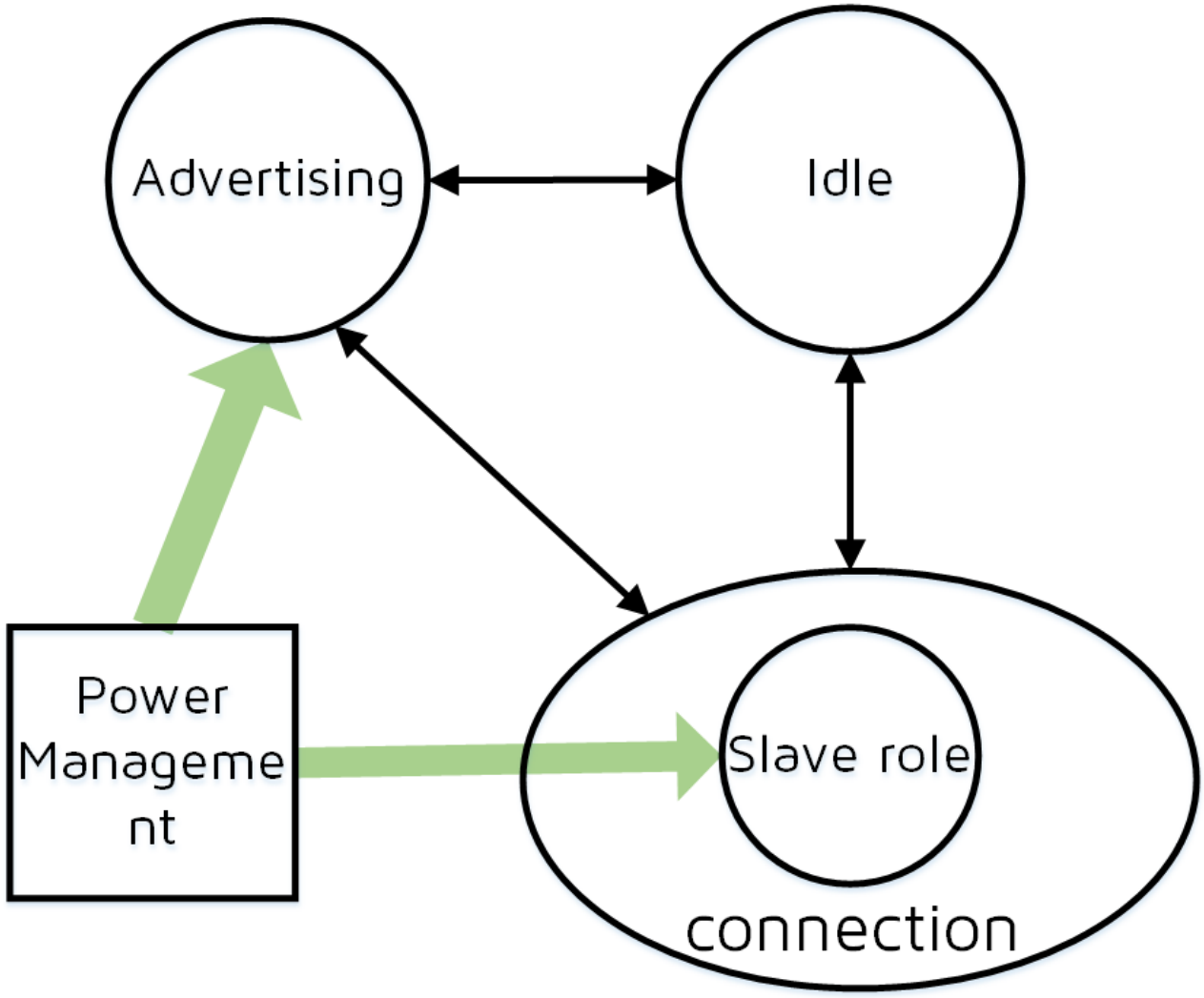


Figure 3.6: Telink Link Layer 状态机

B80 BLE Single Connection SDK 由于仅支持 Slave Role，状态机仅拥有 3 种基本状态：Idle（Standby）、Advertising、Connection（Slave Role）。Connection（Slave Role）在后面会简称为 ConnSlaveRole。

图中 Power Management 并不是 LL 的一种状态，而是功能模块，表示的是 SDK 只对 Advertising 和 ConnSlaveRole 进行了低功耗处理；Idle state 如果需要低功耗，user 在应用层调用相关 API 可以完成。

基于上面 3 种状态，stack/ble/ll/ll.h 中定义了状态机的命名。

```
#define BLS_LINK_STATE_IDLE         0
#define BLS_LINK_STATE_ADV          BIT(0)
#define BLS_LINK_STATE_CONN         BIT(3)
```

Link Layer 状态机的切换都在 BLE stack 底层自动完成，所以 user 在应用层无法去修改状态，只能获取当前状态，调用下面 API 即可，返回值为上面 3 种状态中的 1 种。

```
u8      blc_ll_getCurrentState(void);
```

3.2.3 Link Layer 状态机组合应用

3.2.3.1 Link Layer 状态机初始化

状态机在设计上很灵活，每一个状态都封装成为一个模块，默认情况下只有最基本的 Idle 模块，user 通过添加模块的方式去搭建自己的状态机组合，从而实现不同的应用。比如应用只需要广播，那么只需要添加 Advertising 模块就可以了，剩下的 ConnSlaveRole 模块没有被配置进来。这样的设计的目的是节省 code size，不需要使用的状态的相关代码不会被编译进来。

MCU 的初始化是必须的，API 如下：

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

Idle 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);
```

Advertising、ConnSlaveRole 对应模块的初始化 API 分别如下：

```
void        blc_ll_initAdvertising_module(void);
void        blc_ll_initSlaveRole_module(void);
```

上面 API 中实参 public_adr 是 BLE public mac address 的指针。

User 通过以上几个 API 去配合组合 Link Layer 状态机，下面给出两个常用的组合方式和对应的应用场景。

3.2.3.2 Idle + Advertising

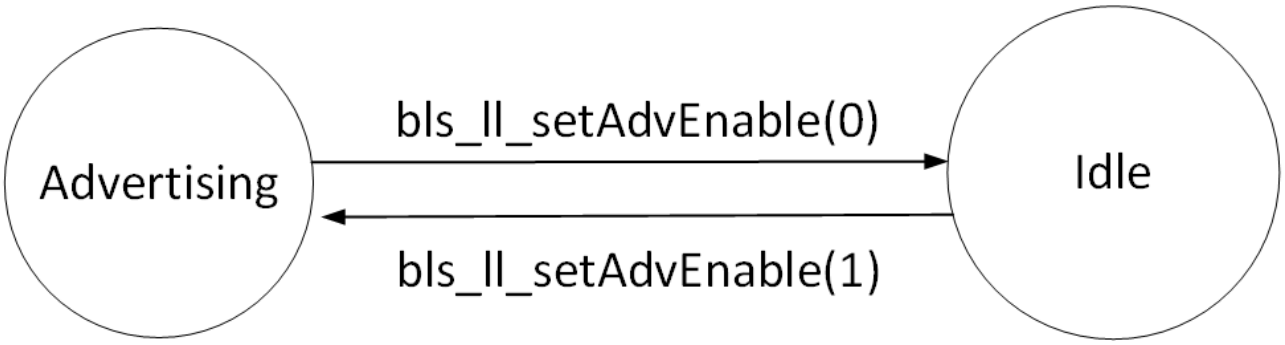


Figure 3.7: Idle + Advertising

上图所示，只初始化 Idle 和 Advertising 模块，使用最基本的 Advertising 功能实现的应用如 beacon 等，单方向广播产品信息。

Link Layer 状态机模块初始化代码为：

```
u8  mac_public[6] = {……};
blc_ll_initBasicMCU();
blc_ll_initStandby_module(mac_public);
blc_ll_initAdvertising_module();
```

Idle 和 Advertising 状态的切换通过 bls_ll_setAdvEnable 来实现。

3.2.3.3 Idle + Advertising + ConnSlaveRole

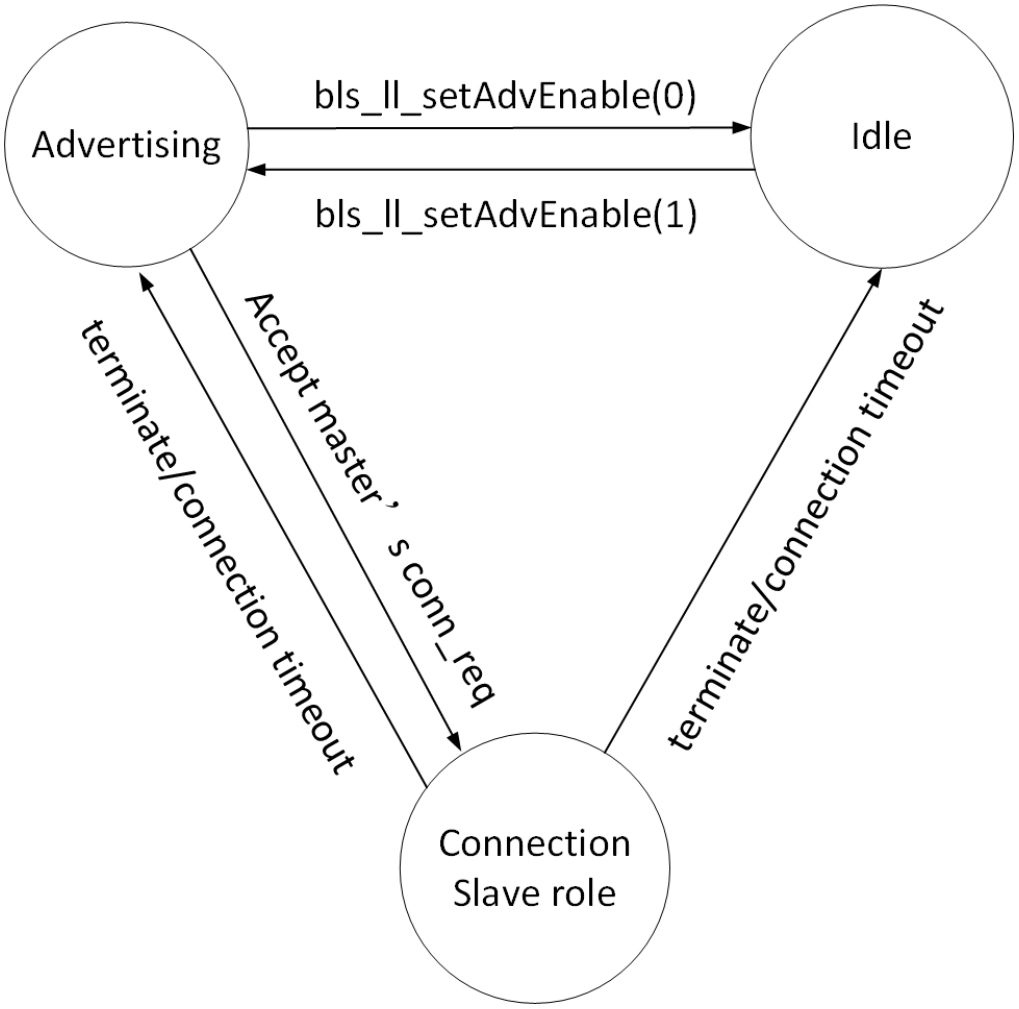


Figure 3.8: BLE Slave LL State

上图所示为一个基本的 BLE slave 应用的 Link Layer 状态机组合，SDK 中 8208 hci/8208 ble sample/8208 module 都是基于该状态机组合。

Link Layer 状态机模块初始化代码为：

```
u8  mac_public[6] = {……};
blc_ll_initBasicMCU();
```

```
blc_ll_initStandby_module(mac_public);
blc_ll_initAdvertising_module();
blc_ll_initSlaveRole_module();
```

该状态机组合下，状态变化情况描述如下。

(1) MCU 上电后，进入 Idle state。Idle state 时将 adv Enable，Link Layer 切换到 Advertising State；adv Disable 时，回到 Idle state。

Adv Enable 和 Disable 通过下面 API bls_ll_setAdvEnable 来控制。

上电后，Link layer 默认处于 Idle 状态，一般需要在 user_init 里面将 Adv Enable，进入 Advertising state。

(2) Link Layer 处于 Idle state 时，Physical Layer 不进行任何 RF 相关动作，不收包也不发包。

(3) Link Layer 处于 Advertising state 时，在广播 channel 上发送广播包。若 master 收到广播包，并发送 conneciton request，Link Layer 收到这个 connection request 后，响应并建立连接，Link Layer 进入 ConnSlaveRole。

(4) Link Layer 处于 ConnSlaveRole 时，有三种情况会回到 Idle State 或者 Advertising state：

- master 向 slave 发送 terminate 命令，断开连接。slave 收到 terminate 命令，退出 ConnSlaveRole。
- slave 向 master 发送 terminate 命令，主动断开连接，退出 ConnSlaveRole。
- slave 的 RF 收包异常或 master 发包异常，导致 slave 长时间收不到包，触发 BLE 的 connection supervision timeout，退出 ConnSlaveRole。

Link layer 的 ConnSlaveRole 退出该 state 时，根据 Adv 是否被 Enable 切换到不同 state：若 Adv 被 enable，Link Layer 重新回到 Advertising state；若 Adv 被 Disable，Link Layer 回到 Idle state。Adv 处于 Enable 或者 Disable 取决于 user 在应用层最后一次调用 bls_ll_setAdvEnable 时设置的值。

3.2.4 Link Layer 时序

结合该 BLE SDK 的 irq_handler 和 main_loop 来说明 Link layer 在各状态的工作时序。

```
_attribute_ram_code_ void irq_handler(void)
{
    ......
    blc_sdk_irq_handler ();
    ......
}

void main_loop (void)
{
////////////////////// BLE entry /////////////////////////////
    blc_sdk_main_loop();
////////////////////// UI entry /////////////////////////////
    ......
}
```

BLE entry 部分 blc_sdk_main_loop 函数负责处理 BLE 协议栈相关的数据和事件。UI entry 是 user 写自己的应用代码的地方。

3.2.4.1 Idle State 时序

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 占据了整个 main_loop 的时序。

3.2.4.2 Advertising State 时序

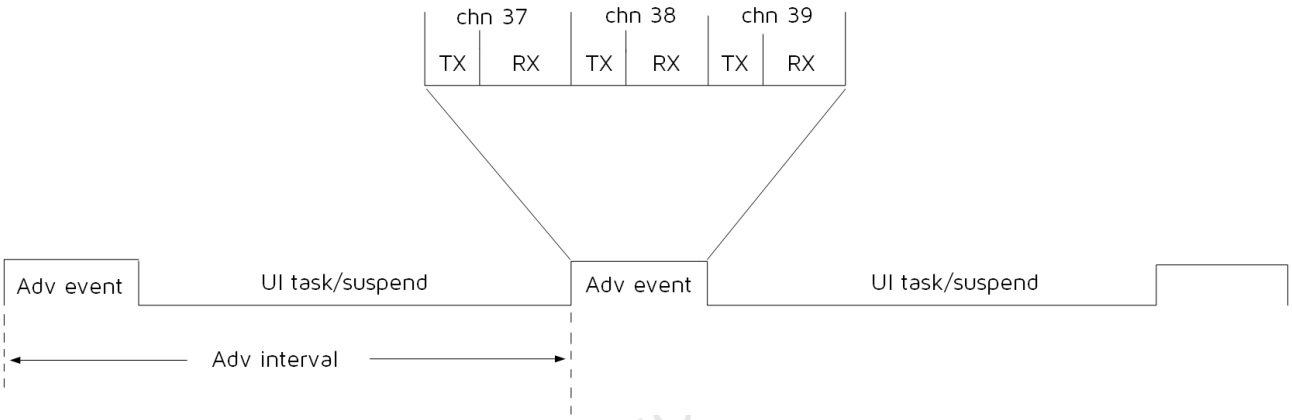


Figure 3.9: Advertising State 时序

Advertising state 时序图如上所示。每一个 adv interval 时间内 Link Layer 触发一个 Adv event。一个典型的 3 个 adv channel 都发包的 Adv event 会在 channel 37、38、39 上都发送一个广播包。每发一个广播包之后都进入收包状态，等待 master 可能的回包，回包有两种：一个是 scan request，收到这个包后再发送一个 scan response 作为回复；另一个是 connect reqeust，收这个包后和 master 建立 ble 连接，进入 Connection state Slave Role。

user 在 main_loop 的 UI entry 部分的 code 执行会在图中所示的 UI task/suspend 部分执行，这部分时间全部可以用来做 UI task，如果需要低功耗的话，可以将多余的时间用来 sleep(suspend/deepsleep retention) 以降低功耗。

在 Advertising state，blc_sdk_main_loop 函数没有太多要处理的任务，只是触发几个跟 Adv 相关的几个回调事件。包括 BLT_EV_FLAG_ADV_DURATION_TIMEOUT、BLT_EV_FLAG_SCAN_RSP、BLT_EV_FLAG_CONNECT 等。

3.2.4.3 Conn State Slave Role 时序

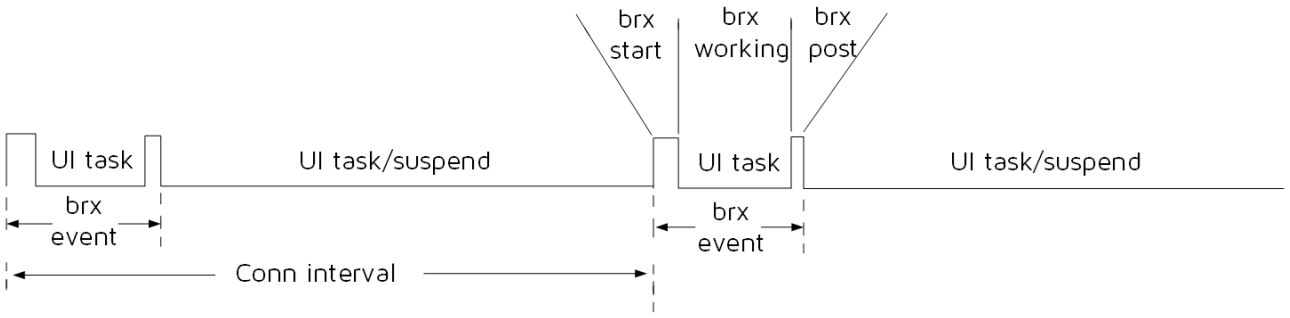


Figure 3.10: Conn State Slave Role 时序

ConnSlaveRole 时序图如上所示。每一个 conn interval 开始的时候，Link Layer 进行一次 BLE 的 RF 包收发过程：先让 PHY 进入收包状态，收到 master 的包后发送一个 ack 包回复，若有 more data，则继续收 master 包并回复，这个过程简称为 brx event。

在该 BLE SDK 中，根据软硬件的工作分配，每个 brx 过程分为 3 个阶段：

(1) brx start 阶段

当 master 发包的时间快要到来时，会由 system tick irq 触发进入 brx start 阶段，在这个中断里 MCU 设置 PHY 的 BLE 状态机进入 brx 状态，底层硬件做好收发包的相关准备，然后退出中断 irq。

(2) brx working 阶段

brx start 结束后，MCU 退出 irq 后，底层硬件进入收包状态，并自动完成收发包的所有工作，不需要软件任何的参与，这个过程称为 brx working 阶段。

(3) brx post 阶段

收发包完成后，brx working 结束，同样由 system tick irq 触发进入中断执行 brx post 阶段。这个阶段主要是协议栈根据 brx working 阶段收发包的情况对 BLE 的一些数据和时序进行相关的处理。

上面三个阶段中 brx start 和 brx post 都是中断完成，而 brx working 阶段不需要软件的参与，此时 UI task 可以正常执行（注意 brx working 阶段 RX、TX、系统定时器中断等处理函数以外的时隙是可以跑 UI task 的）。brx working 时间内，硬件需要进行收发包，所以不能进 sleep(suspend/deepsleep retention)。

整个 conn interval 内除去 brx 过程的时间，可以用来做 UI task，如果需要低功耗的话，可以将多余的时间用来 sleep(suspend/deepsleep retention) 以降低功耗。

在 ConnSlaveRole，blc_sdk_main_loop 需要处理 brx 过程收到的数据。brx working 过程中，实际是在 RX 接收中断 irq 处理中将硬件收到的 master 数据包拷贝出来，这些数据并不会立刻实时处理，而是将数据缓存到软件 RX fifo 中（对应 code 中的 app_acl_rxfifo）。blc_sdk_main_loop 函数会检查软件 RX fifo 中是否有数据，只要有数据就去处理。

blc_sdk_main_loop 对数据包的处理包括：

(1) 数据包的解密

(2) 数据包的解析

解析的数据若发现是属于 master 发给 Link Layer 的控制命令，立即执行该命令，若是 master 发给 Host 层的数据，则会通过 HCI 接口将数据丢到 L2CAP 层处理。

3.2.4.4 Conn State Slave role 时序保护

ConnSlaveRole，每个 interval 需要一个收发包事件，也就是上面的 Brx Event。B80 SDK 中，Brx Event 完全是由中断触发的，所以 MCU 系统总中断需要一直被打开。如果 user 在 Conn state 的一些任务时间较长且必须把系统总中断关闭（比如擦除 Flash），就会造成 Brx Event 被停掉，BLE 的时序很快就乱掉，最终连接断开。对于这种情况 SDK 中提供了一套保护机制，让 user 去停掉 Brx Event 却不破坏 BLE 的时序，user 需要严格根据这个机制来操作。相关 API 如下：

```
int     bls_ll_requestConnBrxEventDisable(void);
void    bls_ll_disableConnBrxEvent(void);
void    bls_ll_restoreConnBrxEvent(void);
```

调用 bls_ll_requestConnBrxEventDisable 来申请关掉 Brx Event。

(1) 该 API 返回值若为 0，表示当前不接受用户的申请，即此时不能停掉 Brx Event。在 Conn state 时的 Brx working 阶段，不能接受申请，返回值为 0，一定要等到一个完整的 Brx Event 结束后，在剩余的 UI task/suspend 时间内才会接受申请。

(2) 该 API 返回非 0 值表示可以接受申请，返回的值是允许停掉 Brx Event 的时间，单位为 ms。该事件值有三种情况：
a）若当前 Link Layer 为 Advertising state 或 Idle state，返回值为 0xffff，即没有 Brx Event，用户关闭系统中断的时间随便多长都可以。
b）若当前为 Conn state，收到了 master 的 update map 或 update connection parameter 且还没有到更新的时间点时，返回时间为更新的时间点减去当前时间。即停掉 Brx Event 的时间不能超过更新的时间点，否则会造成后面所有包收不到，最终断开连接。
c）若当前为 Conn state，且没有 master 的更新请求，返回值为当前 connection supervision timeout 值的一半。比如当前 timeout 为 1S，返回值为 500ms。

user 调用上面的 API 申请停掉 Brx Event，若返回值对应的时间 (ms)，足够自己的任务运行时间，即可进行该任务。在该任务执行之前，调用 API bls_ll_disableConnBrxEvent 停掉 Brx Event。任务结束后，调用 API bls_ll_restoreConnBrxEvent 重新开启 Brx Event 并修复 BLE 时序。

参考使用方法如下。其中具体时间的判断，以测试到的实际任务时间为准。

```
        if(bls_ll_requestConnBrxEventDisable() > 300)
        {

            bls_ll_disableConnBrxEvent();

#if 0  //test 1
            irq_disable();
            DBG_CHN3_HIGH;
            sleep_us(287*1000);
            DBG_CHN3_LOW;
            irq_enable();
#else  //test 2
            DBG_CHN3_HIGH;
            flash_erase_sector(0x40000);
            DBG_CHN3_LOW;
#endif

            bls_ll_restoreConnBrxEvent();


        }
```

Figure 3.11: Scanning in Advertising state 时序

3.2.5 Link Layer 状态机扩展

上面 BLE Link Layer 状态机和工作时序介绍了最基本的几种状态，能够满足 BLE slave 基本应用。

但是考虑到 user 可能会有的一些特殊的应用，B80 BLE SDK 增加了 Conn state Slave role 时还要能够 advertising，下面详细描述。

3.2.5.1 Advertising in ConnSlaveRole

在 Link Layer 处于 ConnSlaveRole 时，可以添加 Advertising feature。

添加 Advertising feature 的 API 为：

```
ble_sts_t   blc_ll_addAdvertisingInConnSlaveRole(void);
```

去掉 Advertising feature 的 API 为：

```
ble_sts_t   blc_ll_removeAdvertisingFromConnSLaveRole(void);
```

以上两个 API ble_sts_t 类型的返回值都是 BLE_SUCCESS。

结合 Advertising 和 ConnSlaveRole 的时序图，当加入 Advertising feature 到 ConnSlaveRole 后，时序图如下。

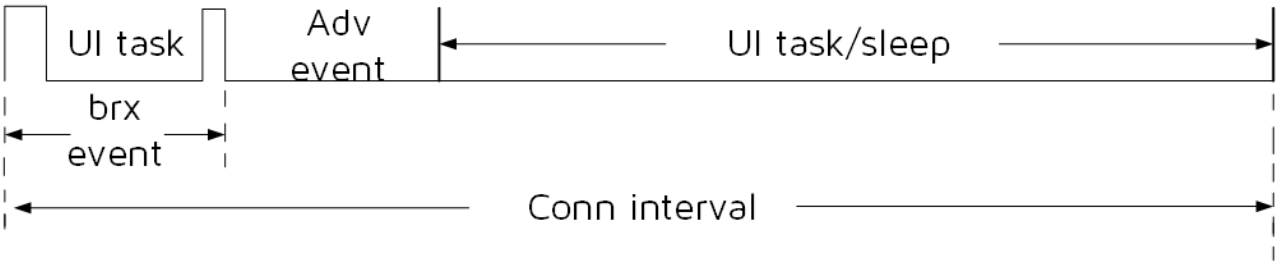


Figure 3.12: Advertising in ConnSlaveRole 时序

当前 Link Layer 还是处于 ConnSlaveRole（BLS_LINK_STATE_CONN），在每个 Conn interval 内 brx event 结束后，立刻执行一次 adv event，然后剩余的时间留给 UI task 或进入 sleep(suspend/deepsleep retention) 节省功耗。

Advertising in ConnSlaveRole 的使用请参考 8208_feature 中的 TEST_ADVERTISING_IN_CONN_SLAVE_ROLE。

3.2.6 Link Layer TX fifo & RX fifo

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 中设计了一个 BLE TX fifo，可以用来缓存传过来的数据，并在 brx/btx 开始后进行数据发送。

Link Layer brx/btx 时收到的所有 peer device 的数据都会先存放在一个 BLE RX fifo 中，然后才上传给 BLE Host 或应用层处理。

BLE TX fifo 和 BLE RX fifo 都在应用层 app_buffer.c 中定义：

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

之后在 app.c 中配置到底层 stack 中：

```
blc_ll_initAclConnTxFifo(app_acl_txfifo, ACL_TX_FIFO_SIZE, ACL_TX_FIFO_NUM);
blc_ll_initAclConnRxFifo(app_acl_rxfifo, ACL_RX_FIFO_SIZE, ACL_RX_FIFO_NUM);
```

其中 ACL_TX_FIFO_SIZE 默认为 64，TX fifo size 默认为 40，除非需要使用 data length extension，否则不允许修改这两个 size。

不管是 TX fifo number 还是 RX fifo number，必须设置为 2 的幂，即 2、4、8、16 等值。User 可以根据自己的需要稍作修改。

RX fifo number 默认为 4，这是一个比较合理的值，能够确保 Link Layer 底层最多缓存 4 个数据包。如果设置的太大，会占用过多的 Sram，如果设置的太小，可能出现数据覆盖的风险：在 brx event 时，Link Layer 很可能在一个 interval 上出现 more data（MD）模式，连续收多个包，如果设置太少的话，很可能在一个 interval 上出现多个包（比如 OTA），而上层对这些数据的响应由于解密时间较长来不及处理，那么就有可能有一些数据被 overflow。

下面举一个 RX overflow 的示例，我们有如下假设：

(1) RX fifo 数量为 8；

(2) 在 brx_event(n) 开启前 RX fifo 的读、写指针分别为 0 和 2；

(3) 在 brx_event(n) 和 brx_event(n+1) 阶段 main_loop 存在任务阻塞，没有及时去取 RX fifo；

(4) 两个 brx_event 阶段都是多包情况。

由上文"Conn state Slave role 时序"小节描述我们知道，在 brx_working 阶段收到的 BLE 数据包只会拷贝到 RX fifo 中（RX fifo 写指针 ++），真正取出 RX fifo 数据进行处理操作是放在 main_loop 阶段的（RX fifo 读指针 ++），我们可以看到第 6 笔数据会覆盖读指针 0 区域。这里需要注意的是在 brx working 阶段的 UI task 时隙是在 RX、TX、系统定时器等中断处理除外的时间。

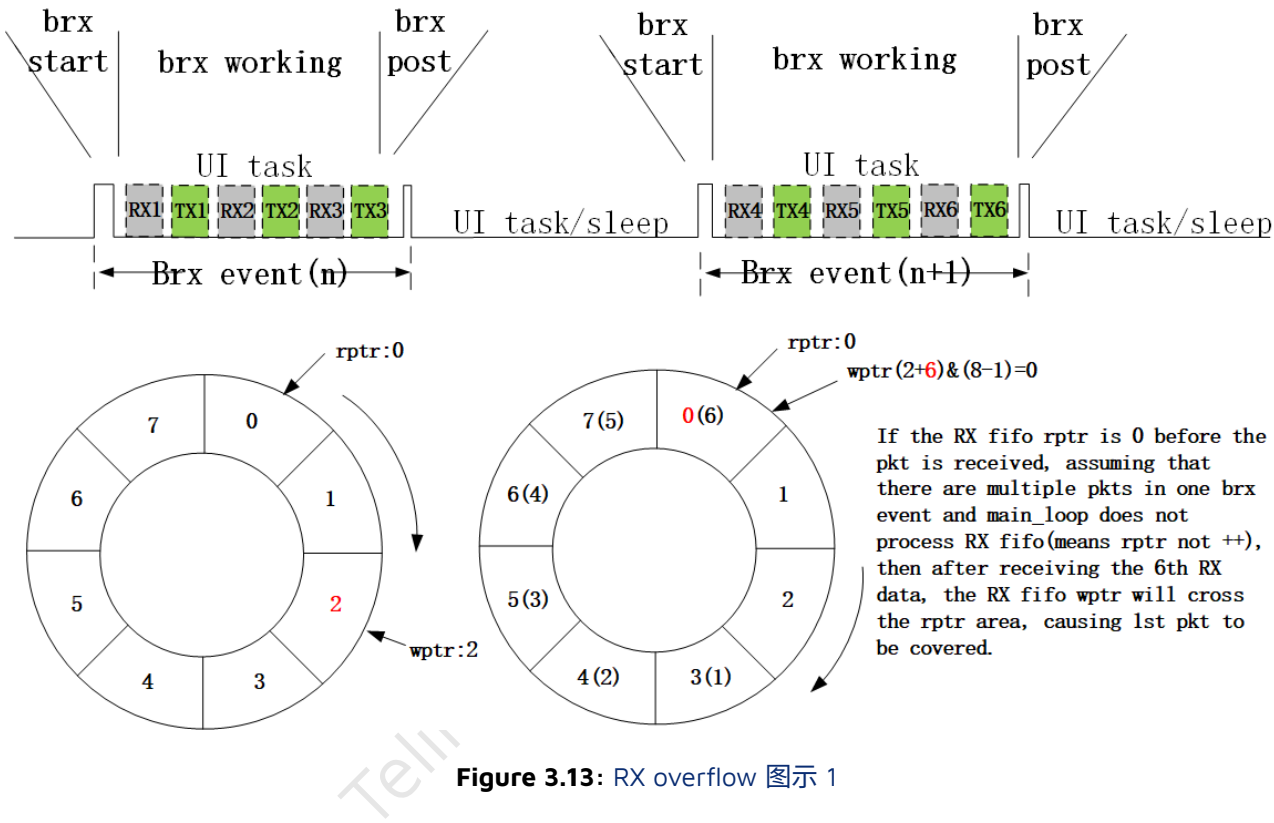


Figure 3.13: RX overflow 图示 1

上面的例子由于间隔了 1 个连接间隔，任务阻塞时间得要足够长，有点极端，下面这个 RX overflow 情形则相对而言出现的概率更高：在一个 brx_event 期间，master 向 slave 写入多笔数据，比如多包数量 7、8 个，这种情况下由于 master 一下子发送了很多数据，slave 来不及进行处理。如下图，读指针只移动了 2 笔，但是写指针移动了 8 笔也会造成数据溢出。

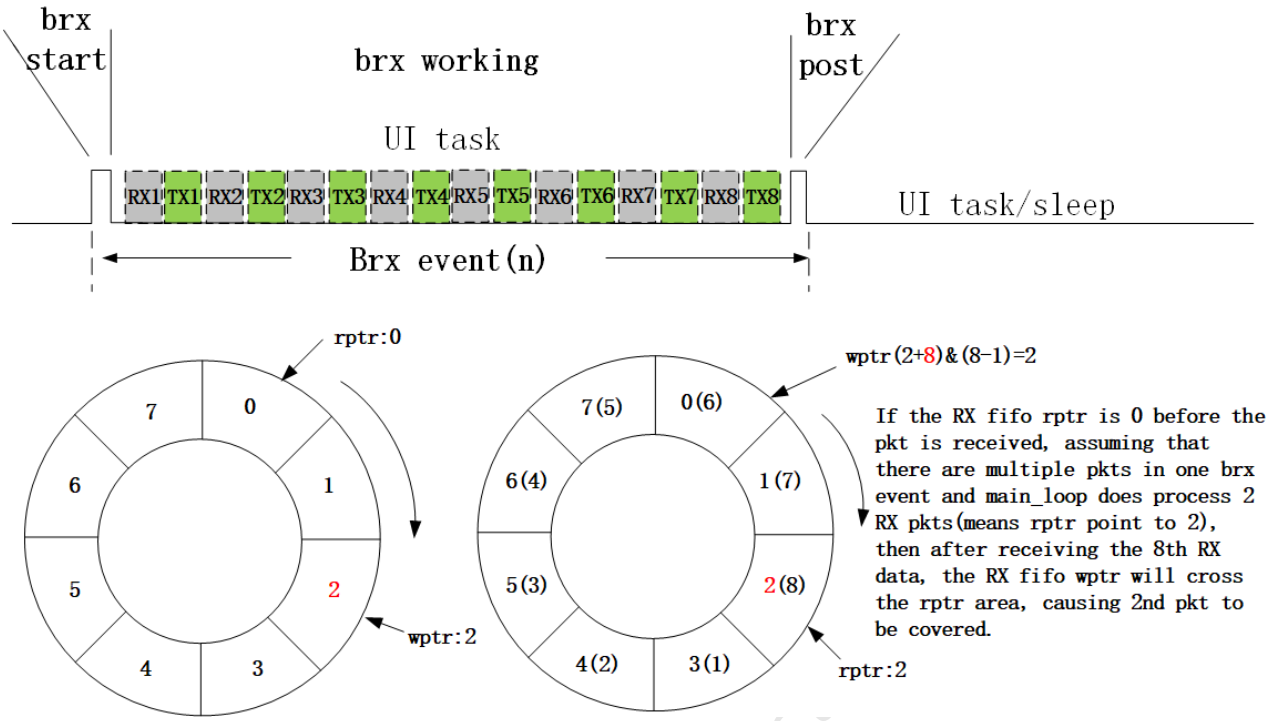


Figure 3.14: RX overflow 图示 2

一旦出现 overflow 导致的数据丢失问题，对加密系统而言，则会出现 MIC failure 断连问题。(旧 SDK 由于 brx event Rx IRQ 将往 Rx fifo 填数据，但是不做数据溢出检查，如果 main_loop 处理 RX fifo 太慢就会导致溢出问题，所以在使用旧 SDK 时，用户需要更多的注意这个风险，避免 master 在一个连接间隔上发太多的数据，注意用户 UI task 处理时间尽量短，避免阻塞问题。)

目前 SDK 新增了 Rx overflow 校验：在 brx Rx IRQ 中检查当前的 RX fifo 写指针和读指针差值是否大于 Rx fifo 数量，一旦发现 Rx fifo 被占满则让 RF 不去 ACK 对方，BLE 协议会确保数据重传，此外 SDK 还提供了 Rx overflow 回调函数以便通知用户，文档后面章节"Telink defined event"会介绍这个回调。

同理如果可能出现一个 interval 多于 8 个有效数据包的话，默认的 8 也就不够用了。

TX fifo number 默认为 8。如果设置太大（如 16）会占用过多的 Sram。

TX fifo 中，SDK 底层 stack 需要用到 2 个，剩下的才完全由应用层使用，TX fifo 为 16 时，应用层只能用 14 个；为 8 时应用层只能用 6 个。

User 在应用层发送数据时（比如调用 blc_gatt_pushHandleValueNotify)，应该先检查一下当前 Link Layer 还有多少 TX fifo 可用。

下面 API 用于判断当前 TX fifo 被占用了多少个，注意不是剩余多少个可用。

```
u8          blc_ll_getTxFifoNumber (void);
```

比如 TX fifo number 默认为 16 时，user 可用为 14 个，所以该 API 返回的值只要小于 14，就是可用的：返回 13 表示还有 1 个可用，返回 0 则表示还有 14 个可用。

TX fifo 使用上，如果客户先看多少个剩余，再决定是否直接 push 数据时，要留一个 fifo，防止发生各种边界问题。

3.2.7 Controller Event

为了满足 user 在应用层对 BLE stack 底层一些关键动作的记录和处理，Telink BLE SDK 提供了三种类型的 event：一是 BLE Controller 定义的标准的 HCI event；二是 Telink 自己定义的一套 event，称为 Telink defined event（前两种类型均属于 Controller event）；三是 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）。

BLE SDK event 架构说明如下图所示，可以看到 HCI event 和 Telink defined event 均属于 Controller event，而 GAP event 属于 BLE host event。下面两小节内容主要介绍 Controller event。

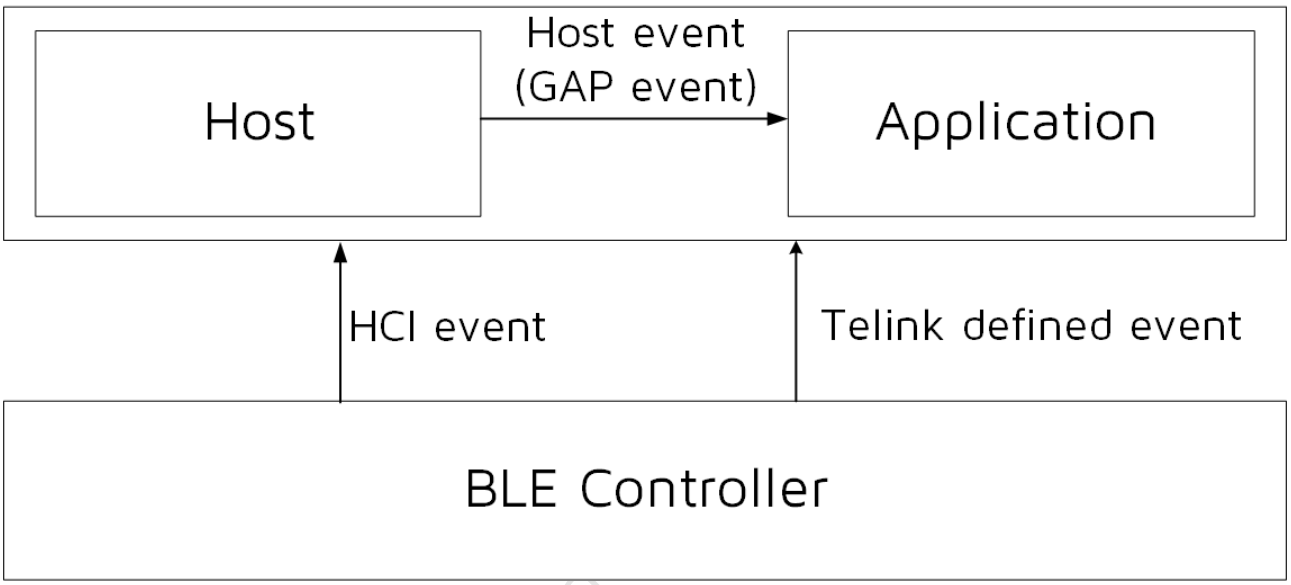


Figure 3.15: BLE SDK Event 架构

3.2.7.1 Controller HCI Event

HCI event 是按 BLE Spec 标准设计的，对于 BLE slave，HCI event 和 Telink defined event 同时可用。

在 BLE slave 上，这两套 event 基本相互独立，只有两个 event 重复定义了：Link Layer 的 connect 和 disconnet event，后面会具体介绍。

User 可以根据自己的需要，在这两套 event 挑选一套使用，也可以两套同时使用。

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。

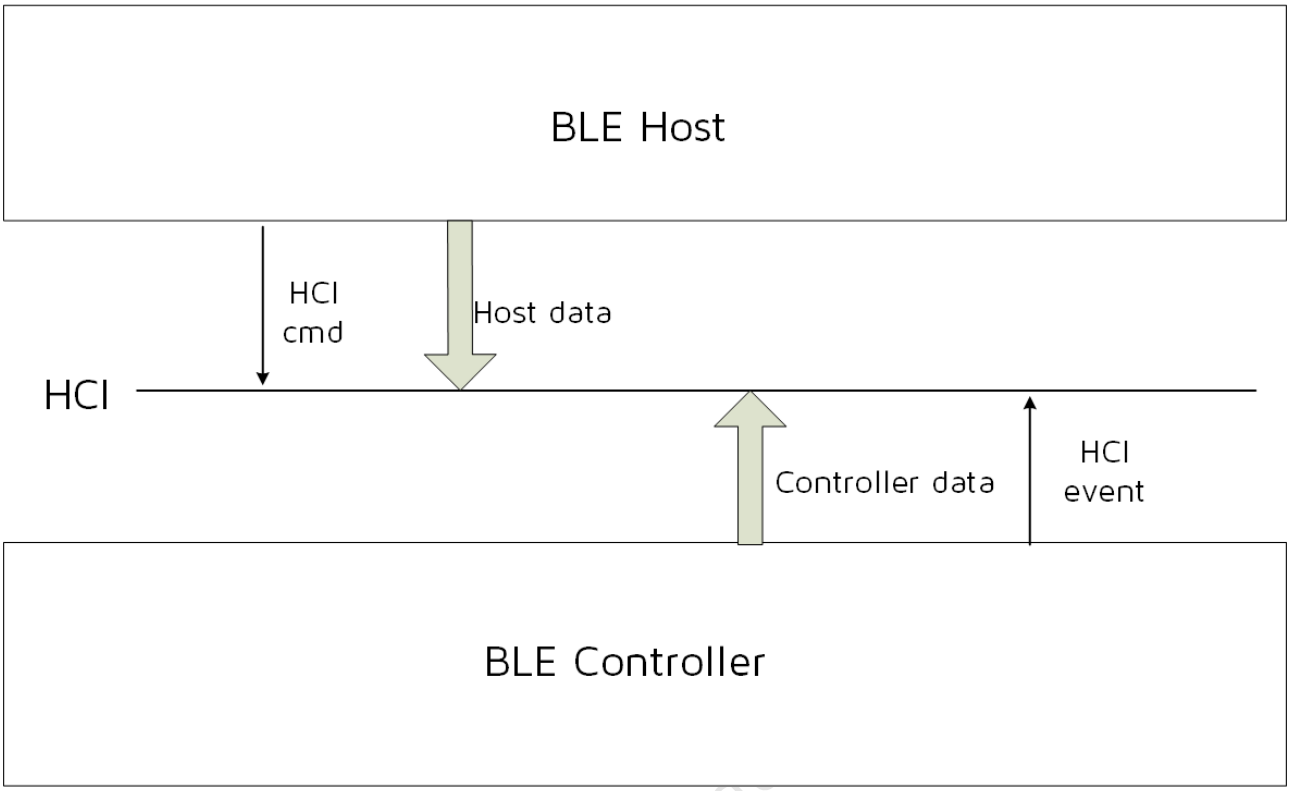


Figure 3.16: HCI Event

Controller HCI event 的定义，详情请参照《Core_v4.2》(Vol 2/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。根据 Spec 的定义，Telink BLE SDK 也将 Controller HCI event 分为两类：HCI Event 和 HCI LE event。由于 Telink BLE SDK 主打低功耗蓝牙，所以对 HCI event 只支持了最基本的几个，而 HCI LE event 绝大多数都支持。

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

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

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 只需要知道以下两个即可：

```
#define         HCI_FLAG_EVENT_TLK_MODULE           (1<<24)
#define         HCI_FLAG_EVENT_BT_STD               (1<<25)
```

HCI_FLAG_EVENT_TLK_MODULE 在后面的 Telink defined event 会再介绍，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 BLE spec 中定义的一致。

3.2.7.2 HCI Event

Telink BLE SDK 中支持了少部分的 HCI event，下面列出了几个 user 可能希望了解的 event。

```
#define HCI_EVT_DISCONNECTION_COMPLETE              0x05
#define HCI_EVT_ENCRYPTION_CHANGE                   0x08
#define HCI_EVT_READ_REMOTE_VER_INFO_COMPLETE       0x0C
#define HCI_EVT_ENCRYPTION_KEY_REFRESH              0x30
#define HCI_EVT_LE_META                             0x3E
```

(1) HCI_EVT_DISCONNECTION_COMPLETE

详情请参照《Core_v5.0》(Vol 2/Part E/7.7.5 "Disconnection Complete Event")。该 event 的总体数据长度为 7，param len 为 4，如下所示，具体数据含义请直接参考 BLE spec。

hci event	event code	param len	status	connection handle	reason
0x04	0x05	4	0x00		

Figure 3.17: Disconnection Complete Event

(2) HCI_EVT_ENCRYPTION_CHANGE and HCI_EVT_ENCRYPTION_KEY_REFRESH

详情请参照《Core_v5.0》(Vol 2/Part E/7.7.8 & 7.7.39)。跟 Controller 加密相关，具体的处理封装到 library 中完成了，这里不介绍细节。

(3) HCI_EVT_READ_REMOTE_VER_INFO_COMPLETE

详情请参照《Core_v5.0》(Vol 2/Part E/7.7.12)。当 Host 使用了 HCI_CMD_READ_REMOTE_VER_INFO 命令 Controller 和 BLE peer device 交互 version 信息，并且收到了 peer device 的 version 后，向 Host 上报该 event。该 event 的总体数据长度为 11，param len 为 8，如下所示，具体数据含义请直接参考 BLE spec。

hci event	event code	param len	status	connection handle	version	manufacture name	subversion
0x04	0x0c	8	0x00				

Figure 3.18: Read Remote Version Information Complete Event

(4) 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，其他都要通过下面 API 来打开 event mask。

```
ble_sts_t   blc_hci_setEventMask_cmd(u32 evtMask);
```

event mask 的定义如下所示：

```
#define HCI_EVT_MASK_DISCONNECTION_COMPLETE                0x0000000010
#define HCI_EVT_MASK_ENCRYPTION_CHANGE                     0x0000000080
#define HCI_EVT_MASK_READ_REMOTE_VERSION_INFORMATION_COMPLETE  0x0000000800
```

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

3.2.7.3 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_CONNECTION_ESTABLISH                0x20
```

(1) HCI_SUB_EVT_LE_CONNECTION_COMPLETE

详情请参照《Core_v5.0》(Vol 2/Part E/7.7.65.1 "LE Connection Complete Event")。当 controller Link Layer 和 peer device 建立 connection 后，上报该 event。该 event 的总体数据长度为 22，param len 为 19，如下所示，具体数据含义请直接参考 BLE spec。

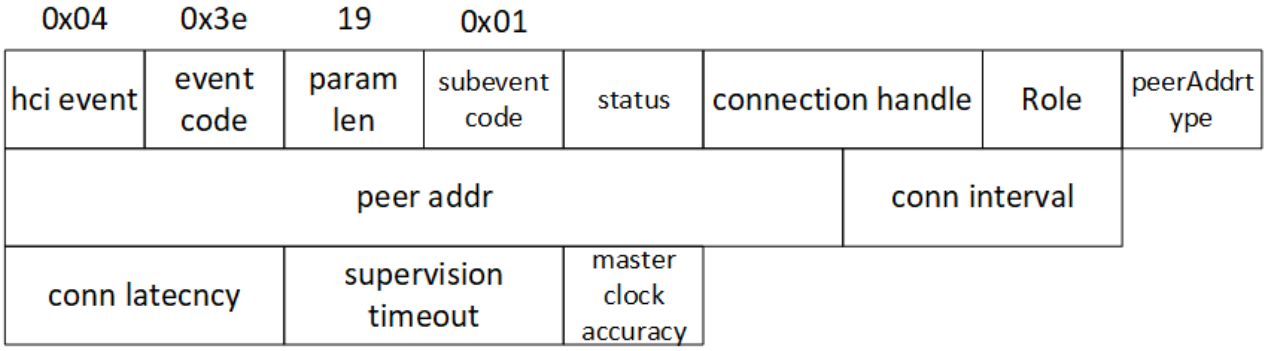


Figure 3.19: LE Connection Complete Event

(2) HCI_SUB_EVT_LE_ADVERTISING_REPORT

详情请参照《Core_v5.0》(Vol 2/Part E/7.7.65.2 "LE Advertising Report Event")。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 BLE spec。

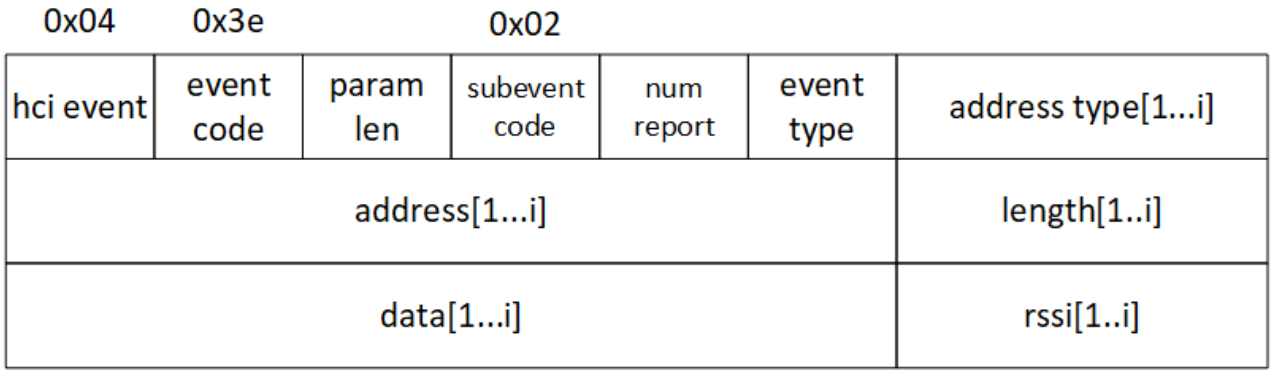


Figure 3.20: LE Advertising Report Event

注意：Teink BLE SDK 中的 LE Advertising Report Event 每次只报一个 adv packet，即上图中的 i 为 1。

(3) HCI_SUB_EVT_LE_CONNECTION_UPDATE_COMPLETE

详情请参照《Core_v5.0》(Vol 2/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。该 event 的总体数据长度为 13，param len 为 10，如下所示，具体数据含义请直接参考 BLE spec。

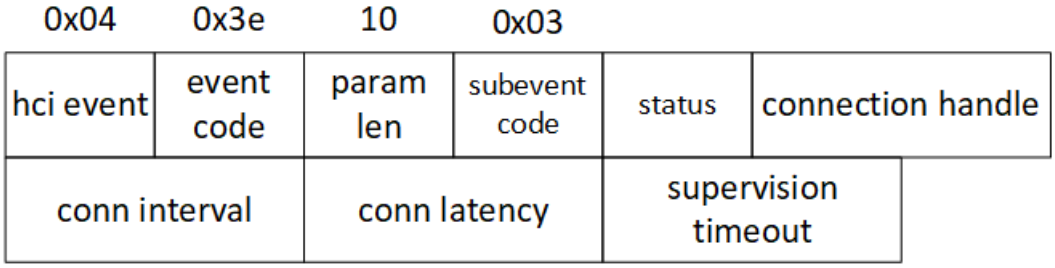


Figure 3.21: LE Connection Update Complete Event

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);
```

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。

3.2.7.4 Telink Defined Event

除了标准的 Controller HCI event，SDK 提供了 Telink defined event。Telink defined event 最多支持 20 个，在 stack/ble/ll/ll.h 中用宏来定义。

目前 SDK 中支持以下回调事件。其中 BLT_EV_FLAG_CONNECT/BLT_EV_FLAG_TERMINATE 可理解为和 HCI event 中的 HCI_SUB_EVT_LE_CONNECTION_COMPLETE /HCI_EVT_DISCONNECTION_COMPLETE 功能上是重复的，只是 event 数据部分定义不同。

```
#define         BLT_EV_FLAG_ADV                     0
#define         BLT_EV_FLAG_ADV_DURATION_TIMEOUT    1
#define         BLT_EV_FLAG_SCAN_RSP                2
#define         BLT_EV_FLAG_CONNECT                 3
#define         BLT_EV_FLAG_TERMINATE               4
#define         BLT_EV_FLAG_LL_REJECT_IND           5
#define         BLT_EV_FLAG_RX_DATA_ABANDOM         6
#define         BLT_EV_FLAG_PHY_UPDATE              7
#define         BLT_EV_FLAG_DATA_LENGTH_EXCHANGE    8
#define         BLT_EV_FLAG_GPIO_EARLY_WAKEUP       9
#define         BLT_EV_FLAG_CHN_MAP_REQ             10
#define         BLT_EV_FLAG_CONN_PARA_REQ           11
#define         BLT_EV_FLAG_CHN_MAP_UPDATE          12
#define         BLT_EV_FLAG_CONN_PARA_UPDATE        13
#define         BLT_EV_FLAG_SUSPEND_ENTER           14
#define         BLT_EV_FLAG_SUSPEND_EXIT            15
```

前面已经介绍，Telink defined event 只在 BLE slave 相关应用中才会被触发。Telink defined event 在 BLE slave 应用上的回调实现，有两种方式。

(1) 第一种方式是每个 event 的回调函数单独注册。这种方式我们称为"independent registration"。

回调函数的原型说明为：

```
typedef void (*blt_event_callback_t)(u8 e, u8 *p, int n);
```

其中 e 是 event number；p 为回调函数执行时，底层传上来相关数据的指针，不同回调函数传上来的指针数据都不一样；n 是回传指针所指的有效数据长度。

注册回调函数的 API 为：

```
void    bls_app_registerEventCallback (u8 e, blt_event_callback_t p);
```

每一个 event 是否响应，取决于应用层是否有对应的回调函数被注册。没有被注册回调函数的 event，不会响应。

(2) 第二种方式是：所有 event 的回调函数共用同一个入口，每个事件是否响应取决于该 event 对应的 event mask 是否被打开。这种方式我们称为"shared event entry"。"shared event entry"方式注册 event 回调是使用和 HCI event 一样的 API：

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

虽然这里共用了 HCI event 的注册回调函数，但二者在实现上有一些区别。HCI event 回调函数里面：

```
h = HCI_FLAG_EVENT_BT_STD  |  hci_event_code;
```

Telink defined event "shared event entry"方式里面：

```
h = HCI_FLAG_EVENT_TLK_MODULE  |  e;
```

其中 e 是 Telink defined event 的 event number。

Telink defined event "shared event entry"方式，类似于 HCI event 的 mask 方法，每一个 event 是否被响应，由下面的 API 来设置其 mask：

```
ble_sts_t   bls_hci_mod_setEventMask_cmd(u32 evtMask);
```

evtMask 的值和 event number 的对应关系为：

```
evtMask = BIT(e);
```

Telink defined event 的两种实现方式,是互斥独立的,只能使用一种。推荐使用方式 1 "independent registration"，SDK 中绝大多数都是用的这种方式；只有 8208_ module 使用了方式 2 "shared event entry"。

在 Telink defined event 的使用上，方式 2 "shared event entry"请参考 project "8208_module"的 demo code。

下面以 connect 和 terminate 事件回调为例，描述这两种方式的 code 实现的方法。

(1) 方式一 "independent registration"

```
void    task_connect (u8 e, u8 *p, int n)
{
        // add connect callback code here
}

void    task_terminate (u8 e, u8 *p, int n)
{
        // add terminate callback code here
}
bls_app_registerEventCallback (BLT_EV_FLAG_CONNECT, &task_connect);
bls_app_registerEventCallback (BLT_EV_FLAG_TERMINATE, &task_terminate);
```

(2) 方式二"shared event entry"

```
int event_handler(u32 h, u8 *para, int n)
{
        if( (h&HCI_FLAG_EVENT_TLK_MODULE)!= 0 ) //module event
        {
            switch(event)
```

```
            {
                case BLT_EV_FLAG_CONNECT:
                {
                    // add connect callback code here
                }
                break;
                case BLT_EV_FLAG_TERMINATE:
                {
                    // add terminate callback code here
                }
            break;
                default:
            break;
            }
        }
}
blc_hci_registerControllerEventHandler(event_handler);
bls_hci_mod_setEventMask_cmd( BIT(BLT_EV_FLAG_CONNECT) | BIT(BLT_EV_FLAG_TERMINATE) );
```

下面对 Controller 所有的事件、事件触发条件、对应的回调函数的参数进行详细的说明。

(1) BLT_EV_FLAG_ADV

该事件目前没有使用。

(2) BLT_EV_FLAG_ADV_DURATION_TIMEOUT

事件触发条件：如果 user 调用 API bls_ll_setAdvDuration 设置了广播时间限制，BLE 协议栈底层启动一个计时，在这个限时达到后，停止广播，同时触发该事件，user 可以在该事件回调函数中进行修改广播事件类型、重新打开广播使能、再次设置广播时间限制等操作。

回传指针 p：空指针 NULL。

数据长度 n：0。

注意：该 event 在 Link Layer 扩展状态中的 advertising in ConnSlaveRole 不触发。

(3) BLT_EV_FLAG_SCAN_RSP

事件触发条件：slave 处于广播状态，收到 master 的 scan request 并响应，回 scan response 时触发该事件。

回传指针 p：空指针 NULL。

数据长度 n：0。

(4) BLT_EV_FLAG_CONNECT

事件触发条件：Link Layer 处于广播状态时收到 master 的连接请求包单元，响应这个请求，进入 Conn state Slave role，触发该事件。

数据长度 n：34。

回传指针 p：p 指向长度为 34 byte 的一片 ram 区域，对应如下图所示的连接请求包单元 PDU。

Payload		
InitA (6 octets)	AdvA (6 octets)	LLData (22 octets)

Figure 2.10: CONNECT_REQ PDU payload

The format of the LLData field is shown in Figure 2.11.

LLData									
AA (4 octets)	CRCInit (3 octets)	WinSize (1 octet)	WinOffset (2 octets)	Interval (2 octets)	Latency (2 octets)	Timeout (2 octets)	ChM (5 octets)	Hop (5 bits)	SCA (3 bits)

Figure 2.11: LLData field structure in CONNECT_REQ PDU's payload

Figure 3.22: 连接请求包单元 PDU

(5) BLT_EV_FLAG_TERMINATE

事件触发条件：Link Layer 状态机中从 Conn state Slave role 退出时触发。

回传指针 p：p 指向一个 u8 类型的变量 terminate_reason，该变量表明当前是什么 reason 导致的 Link Layer 断开连接。

数据长度 n：1。

Conn state Slave role 退出的三种情况对应的 reason 分别如下：

a) slave 和 master 的 RF 通信出现问题（RF 不好或者 master 断电等），slave 连续一段时间收不到 master 的包，连接超时（connection supervision timeout）时触发该事件，连接断开，回到 None Conn state。terminate reason 为 HCI_ERR_CONN_TIMEOUT（0x08）。

b) Master 发送 terminate 主动断开连接，slave 收到 terminate 命令，对这个命令进行 ack 后，触发该事件，连接断开，回到 None Conn state。terminate reason 是 slave 在 Link Layer 上收到的 LL_TERMINATE_IND 控制包中的 Error Code，该 Error Code 是由 master 决定的。常见的 Error code 有 HCI_ERR_REMOTE_USER_TERM_CONN(0x13)、HCI_ERR_CONN_TERM_MIC_FAILURE（0x3D）等。

c) Slave 端调用了 API bls_ll_terminateConnection(u8 reason)，主动断开连接。terminate reason 是该 API 的实参 reason。

(6) BLT_EV_FLAG_LL_REJECT_IND

事件触发条件：Master 在 Link Layer 发送 LL_ENC_REQ（encryption request），且声明了使用之前已经分配好的 LTK，slave 无法找到对应的 LTK，发送 LL_REJECT_IND（or LL_REJECT_EXT_IND），此时触发该事件。

回传指针 p：指向发送的 command(LL_REJECT_IND or LL_REJECT_EXT_IND)。

数据长度 n：1。

更多信息请参照《Core_v5.0》(Vol 6/Part B/2.4.2)。

(7) BLT_EV_FLAG_RX_DATA_ABANDOM

事件触发条件：当 BLE RX fifo overflow 时（参考前文"Link Layer TX fifo & RX fifo"小节），或者在一个连接间隔里收到多包数量 > 设定的多包数量阈值（用户需要调用 API：blc_ll_init_max_md_nums 且参数不为 0，SDK 底层才会做多包数量的检查），触发 BLT_EV_FLAG_RX_DATA_ABANDOM 事件。

回传指针 p：空指针 NULL。

数据长度 n：0。

(8) BLT_EV_FLAG_PHY_UPDATE

事件触发条件：当 slave 或者 master 主动发起 LL_PHY_REQ，更新成功或者失败后触发；或者当 slave 或者 master 被动收到 LL_PHY_REQ，并且 PHY 更新成功后触发。

数据长度 n：1

回传指针 p：指向一个 u8 类型的变量，指示当前连接的 PHY mode。

```
typedef enum {
    BLE_PHY_1M          = 0x01,
    BLE_PHY_2M          = 0x02,
} le_phy_type_t;
```

(9) BLT_EV_FLAG_DATA_LENGTH_EXCHANGE

事件触发条件：Slave 和 Master 交互 Link Layer 最大数据长度时触发，即一方发送 ll_length_req，另一方回复 ll_length_rsp。如果 Slave 主动发送 ll_length_req，需要等收到 ll_length_rsp 时触发；如果 Master 发起 ll_length_req，Slave 回复 ll_length_rsp 后立刻触发。

数据长度 n：12。

回传指针 p：指向一片内存数据，对应如下结构体的前 6 个 u16 的变量。

```
typedef struct {
    u16     connEffectiveMaxRxOctets;
    u16     connEffectiveMaxTxOctets;
    u16     connMaxRxOctets;
    u16     connMaxTxOctets;
    u16     connRemoteMaxRxOctets;
    u16     connRemoteMaxTxOctets;
            ......
}ll_data_extension_t;
```

connEffectiveMaxRxOctets 和 connEffectiveMaxTxOctets 是当前连接上最终允许的 RX 和 TX 最大数据长度；connMaxRxOctets 和 connMaxTxOctets 是设备自己 RX 和 TX 最大数据长度；connRemoteMaxRxOctets 和 connRemoteMaxTxOctets 是对方设备 RX 和 TX 最大数据长度。

connEffectiveMaxRxOctets = min(supportedMaxRxOctets,connRemoteMaxTxOctets);

connEffectiveMaxTxOctets = min(supportedMaxTxOctets, connRemoteMaxRxOctets);

(10) BLT_EV_FLAG_GPIO_EARLY_WAKEUP

事件触发条件：BLE slave 进入 sleep（suspend 或 deepsleep retention）之前，计算好下一次醒来的时间点，到该时间点后醒来（这是由 sleep 状态下的 timer 计时实现的），那么会存在一个问题：如果 sleep 的时间过长，user 的任务必须到 sleep 醒来后才能处理，对于一些实时性要求比较严的应用，可能会出问题。如对于键盘扫描，使用者按键的动作可能很快，为了保证按键不丢同时又要处理按键去抖动，按键扫描的时间间隔在 10~20ms 比较好，此时如果 sleep 时间较大，如 400ms、1s 等（sleep 时间在启用 latency 的时候会达到这些

值），按键就会丢掉。那么需要在 MCU 进入 sleep 之前判断当前睡眠的时间，如果时间过长，设置 suspend/deepsleep retention 可以被使用者的按键动作提前唤醒（后面 PM 模块详细介绍）。

当 suspend/deepsleep retention 在 timer wakeup 时间点之前被 GPIO 提前唤醒时，触发 BLT_EV_FLAG_GPIO_EARLY_WAKEUP 事件。

数据长度 n：1。

回传指针 p：指向一个 u8 型变量 wakeup_status，wakeup_status 变量记录了当前 suspend 哪些唤醒源状态生效了，由 drivers/lib/include/pm.h 可看到如下唤醒状态。

```
enum {
    WAKEUP_STATUS_PAD              = BIT(0),
    WAKEUP_STATUS_CORE             = BIT(1),
    WAKEUP_STATUS_TIMER            = BIT(2),

    STATUS_GPIO_ERR_NO_ENTER_PM    = BIT(7),
    STATUS_ENTER_SUSPEND           = BIT(30),
};
```

以上参数的含义请参考“低功耗管理”部分下的详细说明。

(11) BLT_EV_FLAG_CHN_MAP_REQ

事件触发条件：slave 在 Conn state，master 需要更新当前连接的频点列表，发送一个 LL_CHANNEL_MAP_REQ，slave 收到这个请求后触发该事件，注意是 slave 收到该请求的时候，还并没有处理。

数据长度 n：5。

回传指针 p：p 指向下面频点列表数组的首地址。

unsigned char 类型的 bltc.conn_chn_map[5]，注意回调函数执行时 p 指向的 bltc.conn_chn_map 是还没有更新的老 channel map。

conn_chn_map 用 5 个 byte 表示当前频点列表，采用映射的方法，每个 bit 代表一个 channel：

conn_chn_map[0] 的 bit0-bit7 分别表示 channel0-channel7。

conn_chn_map[1] 的 bit0-bit7 分别表示 channel8-channel15。

conn_chn_map[2] 的 bit0-bit7 分别表示 channel16-channel23。

conn_chn_map[3] 的 bit0-bit7 分别表示 channel24-channel31。

conn_chn_map[4] 的 bit0-bit4 分别表示 channel32-channel36。

(12) BLT_EV_FLAG_CHN_MAP_UPDATE

事件触发条件：slave 在连接状态，收到 LL_CHANNEL_MAP_REQ 命令后到了更新的时间点，将 channel map 进行更新，触发该事件。

回传指针 p：p 指向 conn_chn_map[5] 的首地址，此时的 conn_chn_map 是更新以后新的 map。

数据长度 n：5。

(13) BLT_EV_FLAG_CONN_PARA_REQ

事件触发条件：Slave 设备处于连接态（Conn state Slave role），master 设备需要更新当前连接的参数，发送 LL_CONNECTION_UPDATE_REQ 命令，Slave 设备收到这个请求后触发该事件，此时还没有处理这个请求。

数据长度 n：11。

回传指针 p：p 指向 LL_CONNECTION_UPDATE_REQ 的 PDU，如下图所示的 11 个 byte。

CtrData					
WinSize (1 octet)	WinOffset (2 octets)	Interval (2 octets)	Latency (2 octets)	Timeout (2 octets)	Instant (2 octets)

Figure 2.15: CtrData field of the LL_CONNECTION_UPDATE_REQ PDU

Figure 3.23: LL_CONNECTION_UPDATE REQ Format in BLE Stack

(14) BLT_EV_FLAG_CONN_PARA_UPDATE

事件触发条件：slave 在连接状态，收到 LL_CONNECTION_UPDATE_REQ 后到了该更新的时间点，对连接参数进行更新，触发该事件。

数据长度 n：6。

回传指针 p：p 指向连接参数更新后的新参数，如下

p[0] | p[1]<<8：new connection，以 1.25ms 为 unit

p[2] | p[3]<<8：new connection latency.

p[4] | p[5]<<8：new connection timeout，以 10ms 为 unit

(15) BLT_EV_FLAG_SUSPEND_ENETR

事件触发条件：slave 执行 blc_sdk_main_loop 函数时，进入 suspend 之前触发该事件。

回传指针 p：空指针 NULL。

数据长度 n：0。

(16) BLT_EV_FLAG_SUSPEND_EXIT

事件触发条件：slave 执行 blc_sdk_main_loop 函数时，suspend 唤醒之后触发该事件。

回传指针 p：空指针 NULL。

数据长度 n：0。

注意：

> 实际 SDK 底层执行 cpu_sleep_wakeup 唤醒后执行该回调，且不管是被 gpio 唤醒还是被 timer 唤醒，都会无条件触发该事件。如果同时发生了 BLT_EV_FLAG_GPIO_EARLY_WAKEUP 事件，这两个事件的先后执行顺序请参考"低功耗管理—低功耗管理工作机制"部分伪代码的描述。

3.2.8 Data Length Extension

BLE Spec 从 core_4.2 开始增加了 data length extension（DLE）。

该 BLE SDK Link Layer 上支持 data length extension，且 rf_len 长度支持到 BLE spec 上最大长度 251 bytes。

详情请参照《Core_v5.0》(Vol 6/Part B/2.4.2.21 "LL_LENGTH_REQ and LL_LENGTH_RSP")。

User 如果需要使用 data length extension 功能，按如下步骤设置。

Step 1：设置合适的 TX & RX fifo size

收发长包都需要更大的 TX &RX fifo size，考虑到这些 fifo 会占据大量的 Sram 空间，在设置 fifo size 时应选取最合适的值，避免 Sram 的浪费或者 stack 的溢出。

发长包需要加大 TX fifo size。TX fifo size 至少比 TX rf_len 大 12，且必须按 4 字节对齐，在 app_buffer.h 中直接修改 ACL_CONN_MAX_TX_OCTETS，code 中会自动处理对齐。

```
#define ACL_CONN_MAX_TX_OCTETS          27
```

收长包需要加大 RX fifo size。RX fifo size 至少比 RX rf_len 大 24，且必须按 16 字节对齐，在 app_buffer.h 中直接修改 ACL_CONN_MAX_RX_OCTETS，code 中会自动处理对齐。

```
#define ACL_CONN_MAX_TX_OCTETS          27
```

Step 2：设置合适的 MTU Size

MTU 即最大传输单元用于设置 BLE 中的 L2CAP 层单个数据包中的 payload 大小，att.h 中提供了接口函数 ble_sts_t blc_att_setRxMtuSize(u16 mtu_size); 在 BLE 协议栈初始化时，用户可直接使用该函数进行传参设定 MTU 大小，但是 MTU size 有效值是在 MTU exchange 流程中协商后得到的，MTU size effect =min(MTU_A, MTU_B) MTU_A 为设备 A 所支持的 MTU 大小，MTU_B 为设备 B 所支持的 MTU 大小；另外只有 MTU size 大于 DLE 长度才能充分利用 DLE 的作用来增加链路层数据吞吐率。

MTU size exchange 的实现，请参考本文档" ATT & GATT" 部分的详细说明，也可以参考 8208_feature 里面 DLE demo 的写法。

```
#define MTU_SIZE_SETTING                    247
blc_att_setRxMtuSize(MTU_SIZE_SETTING);
```

Step 3：data length exchange

在收发长包前，一定要确保在 BLE connection 上 data length exchange 这个流程已经完成。data length exchange 流程是 Link Layer 上 LL_LENGTH_REQ 和 LL_LENGTH_RSP 两个包的交互过程，slave 和 master 任何一方都可以主动发起 LL_LENGTH_REQ，另一方回应 LL_LENGTH_RSP。通过这两个包的交互后，master 和 slave 都能知道彼此 TX 和 RX 最大包长，然后取两个最大包长中的较小值，即可确定当前 connection 收发包最大包长。

不管是哪一端发起的 LL_LENGTH_REQ，data length exchange 流程结束时，SDK 都会产生 BLT_EV_FLAG_DATA_LENGTH_EXCHANGE 事件回调（前提是注册了该事件的回调），user 可参考"Telink defined event"部分的说明来了解该事件回调函数各个参数的含义。

在这个 BLT_EV_FLAG_DATA_LENGTH_EXCHANGE 事件回调函数里，可以获得最终的最大 TX 包长和 RX 包长。

在实际应用中，当 8207 作为 BLE slave 设备时，master 端可能会主动发起 LL_LENGTH_REQ，也可能不会主动发起。如果 master 端没有主动发起 LL_LENGTH_REQ，就需要由 slave 端主动发起。SDK 提供主动发起 LL_LENGTH_REQ 的 API 如下：

```
ble_sts_t       blc_ll_exchangeDataLength (u8 opcode, u16 maxTxOct);
```

API 中 opcode 填"LL_LENGTH_REQ"，maxTxOct 填当前设备支持的最大 TX 包长，比如 TX 最大包长为 200 bytes 时，设置如下：

```
blc_ll_exchangeDataLength(LL_LENGTH_REQ , 200);
```

由于 slave 设备并不知道 master 会不会主动发送 LL_LENGTH_REQ，推荐一个参考的方法：注册 BLT_EV_FLAG_DATA_LENGTH_EXCHANGE 事件回调，当 connection 建立后开启一个软件定时器开始计时(比如 2S)，如果在 2S 后这个回调一直没有触发，就说明 master 还没有主动发起 LL_LENGTH_REQ，此时 slave 调用 API blc_ll_exchangeDataLength 主动发起 LL_LENGTH_REQ。

Step 4： MTU size exchange

除了上面 data length exchange 流程，MTU size exchange 的流程必须也要执行，确保大的 MTU size 生效，防止对方设备在 BLE l2cap 层无法处理长包。MTU size 的值需要大于等于 TX & RX 最大包长。MTU size exchange 的实现，请参考本文档" ATT & GATT" 部分的详细说明，也可以参考 8208_feature 里面 demo 的写法。

Step 5： 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。

3.2.9 Controller API

3.2.9.1 Controller API 说明

在 3.1.1 小节所示的标准 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 BLE spec《Core_v5.0》中严格定义了，详情请参照《Core_v5.0》(Vol 2/Part E/ Host Controller Interface Functional Specification)。

Telink BLE SDK 的设计是按照标准的 BLE 架构设计，可以作为一个单独的 Controller 与另外一个运行 Host 协议的系统级联工作，所以所有的操作设置 Link Layer 的 API 都是严格按照 Spec 上 Host command 的数据格式来定义实现的。

虽然 Telink BLE SDK 最终采用了如上图的架构，应用层跨越 Host 直接操作设置 Link Layer，但使用的 API 还是标准的 HCI 部分的 API。下面的 API 具体介绍中会给出 Spec 上对应的 Host command，user 可以参考 Spec 上具体说明加以理解。

BLE spec 上所有的 HCI command，Controller 的处理都会有对应的 HCI command complete event 或 HCI command status event 作为对 Host 层的应答，但在 Telink BLE SDK 中，是分情况处理的。

(1) 对于 8208_hci 类的应用，Telink 的 IC 只作为 BLE controller，需要和其他家的运行 BLE host 的 MCU 协同工作，每个 HCI command 都会有对应的 HCI command complete event 或 HCI command status event 产生。

(2) 对于 8208_sample 应用，BLE Host 和 Controller 都运行在 Telink IC 上，Host 调用 interface 发送 HCI command 给 Controller 时，Controller 全部都能正确收到，不会有丢失的情况，所以 Controller 在处理 HCI command 时不再回复 HCI command complete event 或 HCI command status event。

Controller API 的声明在 stack/ble/ll 和 stack/ble/hci 目录下的头文件中，其中 ll 目录下根据 Link Layer 状态机功能的分类分为 ll.h、ll_adv.h、ll_slave.h、ll_pm.h，user 可以根据 Link Layer 的功能去寻找，比如跟 advertising 相关功能的 API 就应该都在 ll_adv.h 中声明。

3.2.9.2 API 返回类型 ble_sts_t

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。

3.2.9.3 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。

该 BLE SDK 中，调用如下接口获得 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，8208 512K flash 上对应的这个地址为 0x76000。如果不需要 random static address，上面的 mac_random_static 填"NULL"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);
```

3.2.9.4 Link Layer 状态机初始化

结合前面对 Link Layer 状态机的详细介绍，以下几个 API 用于配置搭建 BLE 状态机时各个模块的初始化。

```
void        blc_ll_initBasicMCU (void)
void        blc_ll_initStandby_module (u8 *public_adr);
void        blc_ll_initAdvertising_module(void);
void        blc_ll_initSlaveRole_module(void);
```

3.2.9.5 bls_ll_setAdvData

详情请参照《Core_v5.0》(Vol 2/Part E/ 7.8.7 "LE Set Advertising Data Command")。

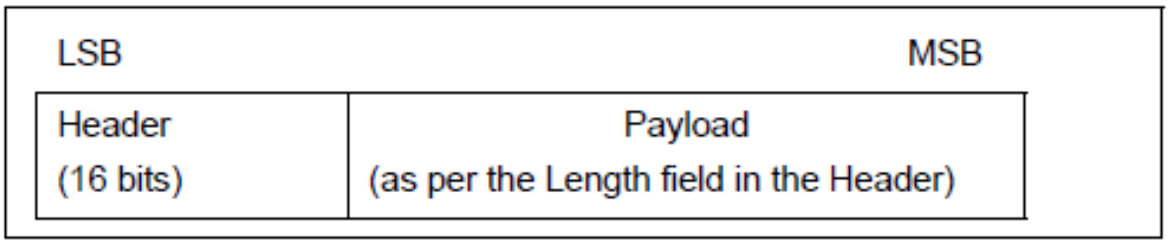


Figure 3.24: BLE 协议栈广播包格式

BLE 协议栈里，广播包的格式如上图所示，前两个 byte 是 head，后面是 Payload（PDU），最多 31 byte。

下面的 API 用于设置 PDU 部分的数据：

```
ble_sts_t   bls_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	添加成功
HCI_ERR_INVALID_HCI_ CMD_PARAMS	0x12	Len 超过最大长度 31

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。

该 BLE SDK 中 feature_backup 工程中定义的 Adv PDU 如下，其中各个字段的含义请参考文档 BLE Spec《CSS v6》(Core Specification Supplement v6.0）中 Data Type Specifcation 的具体说明。

```
const u8    tbl_advData[] = {
     8,  DT_COMPLETE_LOCAL_NAME,             'f','e','a','t','u','r','e',
     2,  DT_FLAGS,                           0x05,
     3,  DT_APPEARANCE,                      0x80, 0x01,
```

```
    5,  DT_INCOMPLT_LIST_16BIT_SERVICE_UUID,   0x12, 0x18, 0x0F, 0x18,
};
```

上面广播数据里，设置广播设备名为“feature”。

3.2.9.6 bls_ll_setScanRspData

详情请参照《Core_v5.0》(Vol 2/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   bls_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	添加成功
HCI_ERR_INVALID_HCI_ CMD_PARAMS	0x12	Len 超过最大长度 31

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。该 BLE SDK 中 8208_ble_sample 工程中定义 scan response data 如下，scan 设备名为“eSample”。各个字段的含义请参考文档 BLE Spec《CSS v6》(Core Specification Supplement v6.0）中 Data Type Specifcation 的具体说明。

```
const u8    tbl_scanRsp [] = {
    8,  DT_COMPLETE_LOCAL_NAME,       'e', 'S', 'a', 'm', 'p', 'l', 'e',
};
```

上面在 advertising data 和 scan response data 中都设置了设备名称且不一样，那么在手机或 IOS 系统上扫描蓝牙设备时，看到的设备名称可能会不一样：

(1) 一些设备只看广播包，那么显示的设备名称为“feature”；

(2) 一些设备看到广播后，发送 scan request，并读取回包 scan response，那么显示的设备名称可能就会是“eSample”。

user 也可以在这两个包中将设备名称写为一样，被扫描时就不会显示两个不同的名字了。

实际上设备被 master 连接后，master 在读设备的 Attribute Table 时，会获取设备的 gap device name，连上设备后也可能会根据那里的设置来显示设备名称。

3.2.9.7 bls_ll_setAdvParam

详情请参照《Core_v5.0》(Vol 2/Part E/ 7.8.5 “LE Set Advertising Parameters Command”)。

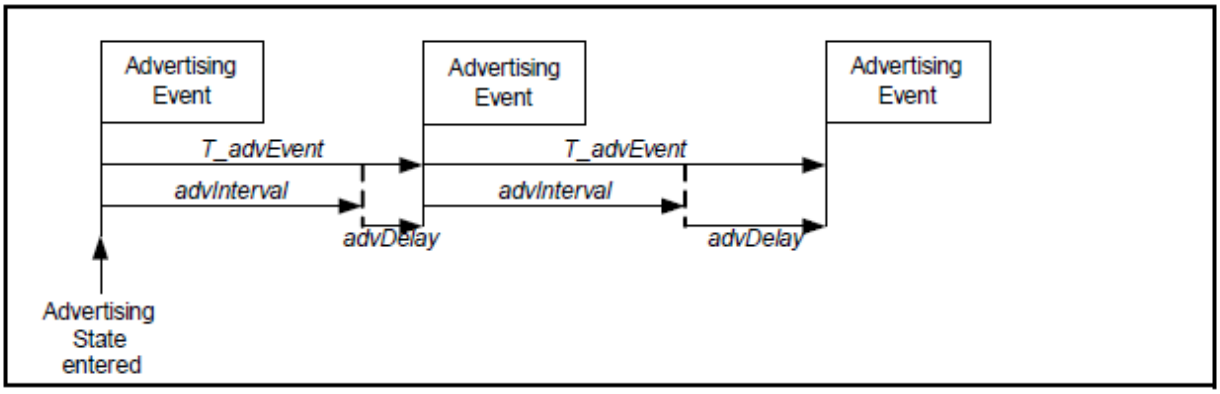


Figure 3.25: BLE 协议栈里 Advertising Event

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   bls_ll_setAdvParam( u16 intervalMin,  u16 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 0.625ms 为基本单位，范围在 20ms ~10.24s 之间，并且 intervalMin 小于等于 intervalMax。

BLE spec 要求 adv interval 不要设一个固定的值，需要有一些随机的变化。Telink BLE SDK 通过 intervalMin 和 intervalMax 设置为不同的值来实现，最终的 adv interval 是在 intervalMin~intervalMax 之间随机变化。若设置的 intervalMin 和 intervalMax 相等，adv interval 会等于固定的 intervalMin，不会变化。

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 BLE Spec，四种基本的广播事件类型如下：

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

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

Figure 3.26: 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 req 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：

```
/* 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。

```
/* Own Address Type */
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:

```
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; //adv_filterPolicy_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	添加成功
HCI_ERR_INVALID_HCI_CMD_PARAMS	0x12	intervalMin 或 intervalMax 的值不符合 BLE spec 的规定

按照 BLE spec 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。

但是考虑到 user 对一些常用的参数可能会经常性的去修改，而又不希望每次都要调用 bls_ll_setAdvParam 同时设置 8 个参数，SDK 对其中 4 个不会跟其他参数有耦合关系的参数，单独封装了 API，以方便 user 的使用。单独封装 3 个 API 如下：

```
ble_sts_t   bls_ll_setAdvInterval(u16 intervalMin, u16 intervalMax);
ble_sts_t   bls_ll_setAdvChannelMap(u8 adv_channelMap);
ble_sts_t   bls_ll_setAdvFilterPolicy(u8 advFilterPolicy);
```

这 3 个 API 的参数与 bls_ll_setAdvParam 中一致。

返回值 ble_sts_t：

(1) bls_ll_setAdvChannelMap 和 bls_ll_setAdvFilterPolicy 会无条件返回 BLE_SUCCESS。

(2) bls_ll_setAdvInterval 返回 BLE_SUCCESS 或 HCI_ERR_INVALID_HCI_CMD_PARAMS。

3.2.9.8 bls_ll_setAdvEnable

详情请参照《Core_v5.0》(Vol 2/Part E/ 7.8.9 "LE Set Advertising Enable Command")。

```
ble_sts_t   bls_ll_setAdvEnable(int en);
```

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

(1) 在 Idle state 时，Enable Advertising，Link Layer 进入 Advertising state。

(2) 在 Advertising state 时，Disable Advertising，Link layer 进入 Idle state。

(3) 在其他 state，Enable Advertising 或 Disable Advertising 都不影响 Link Layer 的 state。

注意：

> 需要注意的是，在任何时候调用该函数，ble_sts_t 无条件返回 BLE_SUCCESS，也就是 adv 相关参数内部会打开或关闭，但只有处于 idle 或 adv state 才会生效。

3.2.9.9 bls_ll_setAdvDuration

```
ble_sts_t   bls_ll_setAdvDuration (u32 duration_us, u8 duration_en);
```

使用 bls_ll_setAdvParam 对广播所有参数设置成功后，使用 bls_ll_setAdvEnable（1）开始广播。若希望对设置好的广播事件进行时间限定，让它持续发送一段时间后就自动关闭，可以调用上面的 API。

duration_en 设为 1 表示开启计时功能，设为 0 表示关闭计时功能。只有在计时功能开启的情况下，duration_us（单位:us）的设置才有意义。

程序从设置的时间点开始计时，一旦超出预设的时间时，停止广播，广播使能（AdvEnable）失效，在 None Conn state 下，会切换到 Idle State。此时会触发 Link Layer 事件 BLT_EV_FLAG_ADV_DURATION_TIMEOUT。

BLE Spec 中规定 ADV_TYPE_CONNECTABLE_DIRECTED_HIGH_DUTY 广播类型，Duration Time 固定为 1.28s，到 1.28s 之后将停止广播。所以对于这种广播类型，调用 bls_ll_setAdvDuration 设置将无效。返回值 ble_sts_t 见下表。

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	添加成功
HCI_ERR_INVALID_HCI_CMD_PARAMS	0x12	ADV_TYPE_CONNECTABLE_DIRECTED_HIGH_DUTY 广播类型不能被设置 Duration Time

如果 user 需要在 Adv Duration Time 到达广播停止后，重新设置广播参数（AdvType、AdvInterval、AdvChannelMap 等），需要在 BLT_EV_FLAG_ADV_DURATION_TIMEOUT 事件的回调函数里设置，并且需要再次调用 bls_ll_setAdvEnable(1) 开始新的广播。

触发 BLT_EV_FLAG_ADV_DURATION_TIMEOUT 需要注意一种特殊的情况：

假设设定了 duration_us 为 2000000，即 2s。

如果 Slave 一直在广播,那么广播时间到达 2s 时会触发 timeout,执行 BLT_EV_FLAG_ADV_DURATION_TIMEOUT 回调。

如果广播时间不到 2s 的情况下（假设在 0.5s 的时候），Slave 和 Master 连接上了，这个 timeout 的计时在底层并没有被清除掉，而是被缓存起来。在进入连接状态 1.5s 的时候，也就是到达实际设定的 timeout 时间点正好 2s 时，由于此时已经处于连接状态，不会去检查广播事件是否超时，也就不会触发 BLT_EV_FLAG_ADV_DURATION_TIMEOUT 回调。

注意：

> 当进入连接状态一段时间（如 10s）后断开连接重新回到广播（adv）状态，在发第一个广播包前，协议栈认为此时的时间超过了之前设定的 2s timeout，触发 BLT_EV_FLAG_ADV_DURATION_TIMEOUT 回调。在这种情况下触发的 BLT_EV_FLAG_ADV_DURATION_TIMEOUT 回调已经远远超过了实际设定的 timeout 时间点。

3.2.9.10 blc_ll_setAdvCustomedChannel

下面 API 用于定制特殊的 advertising channel，只对一些非常特殊的应用有意义，如 BLE mesh，其他常规 BLE 应用不要使用该 API。

```
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 2420MHz、2430MHz、2450MHz，可如下调用：

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

3.2.9.11 rf_set_power_level_index

该 BLE SDK 提供了 BLE RF packet 能量设定的 API:

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

level 值的设置参考 drivers/lib/include/rf_drv.h 中定义的枚举变量 RF_PowerTypeDef。

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。需要注意的是：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，如下 code 所示。

```
void task_suspend_exit (u8 e, u8 *p, int n)
{
    rf_set_power_level_index (MY_RF_POWER_INDEX);
}
bls_app_registerEventCallback (BLT_EV_FLAG_SUSPEND_EXIT, &task_suspend_exit);
```

3.2.9.12 bls_ll_terminateConnection

```
ble_sts_t   bls_ll_terminateConnection (u8 reason);
```

应用层可以调用此 API 在 Link Layer 上发送一个 Terminate 给 master，主动断开连接，reason 为需要指定的断开原因，reason 的设置详请参照《Core_v5.0》(Vol 2/Part D/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
bls_ll_terminateConnection(HCI_ERR_REMOTE_USER_TERM_CONN);
```

Telink BLE SDK 底层 stack 中，只有一个地方会调用该 API 主动 terminate：解密对方设备的数据包，如发现认证数据 MIC 错误，调用 bls_ll_terminateConnection(HCI_ERR_CONN_TERM_MIC_FAILURE)，告知对方解密错误，断开连接。

Slave 调用该 API 主动发起断开连接后，一定会触发 BLT_EV_FLAG_TERMINATE 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。

在 Connection state Slave role 时，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。返回值列表如下。

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	成功
HCI_ERR_CONN_NOT_ ESTABLISH	0x3E	Link Layer 处于非 Connection state Slave role
HCI_ERR_CONTROLLER _BUSY	0x3A	Controller busy（有大量数据正在发送）暂时无法接受该命令。

3.2.9.13 Get Connection Parameters

获取当前连接参数的 Connection Interval、Connection Latency、Connection Timeout 的 API 如下：

```
u16         bls_ll_getConnectionInterval(void);
u16         bls_ll_getConnectionLatency(void);
u16         bls_ll_getConnectionTimeout(void);
```

(1) 若返回值为 0，表示当前 Link Layer 处于 None Conn state，也就没有连接参数。

(2) 若返回为非 0 值表示参数值：

- bls_ll_getConnectionInterval 返回的是实际 conn interval 除以 1.25ms 单位值，比如当前 conn interval 为 10ms，则返回值为 8。
- bls_ll_getConnectionLatency 返回实际 Latency 值。
- bls_ll_getConnectionTimeout 返回的是实际 conn timeout 除以 10ms 的单位值，比如当前 conn timeout 为 1000ms，则返回值为 100。

3.2.9.14 blc_ll_getCurrentState

下面 API 用于获取当前 Link Layer 所处的状态。

```
u8  blc_ll_getCurrentState(void);
```

user 在应用层判断当前状态，如：

```
if(blc_ll_getCurrentState() == BLS_LINK_STATE_ADV)
if(blc_ll_getCurrentState() == BLS_LINK_STATE_CONN)
```

3.2.9.15 blc_ll_getLatestAvgRSSI

当 Link Layer 进入 Slave role 后，通过下面 API 能得到跟自己连接的 peer device 过去一段时间的平均 RSSI。

```
u8      blc_ll_getLatestAvgRSSI(void)
```

返回值 u8 的 rssi_raw 要做一定的转换，rssi_real = rssi_raw- 110，假设返回值为 50，则 rssi = -60 db。

3.2.9.16 Whitelist & Resolvinglist

前面介绍过，Advertising 的 filter_policy 中涉及到 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 SDK whitelist 最多存储 4 个设备：

```
#define     MAX_WHITE_LIST_SIZE         4
```

相关接口：

```
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	添加成功
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 SDK Resolvinglist 最多存储 2 个设备：

```
#define     MAX_WHITE_IRK_LIST_SIZE            2
```

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 可以在文档 SMP 部分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。

3.2.10 2M PHY

3.2.10.1 2M PHY 介绍

2M PHY 是《Core_5.0》新增加的 Feature，很大程度上扩展了 BLE 的应用场景，2M PHY(2Mbps) 大大提高了 BLE 带宽。2M 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。连接状态下的使用方法在本小节中中介绍，广播通道下使用方法在“Extended Advertising”章节中涉及到。

3.2.10.2 2M PHY Demo 介绍

Telink 提供的 B80 BLE SDK 中，为节省 Sram，2M PHY 默认是关闭的，用户如果选择使用此 Feature，可以手动打开，打开方法可以参考 BLE SDK 提供的 Demo。

- Slave 端可参考 Demo “b80_feature_test”

在 vendor/8208_feature/feature_config.h 中定义宏

```
#define FEATURE_TEST_MODE    TEST_PHY_CONN
```

用户也可以选择使用其他厂家的设备，只要支持 2M PHY 即可以和 Telink 的 Slave 设备兼容互联。

3.2.10.3 2M PHY API 介绍

(1) API

```
void blc_ll_init2MPhy_feature(void)
```

用于使能 2M PHY。

(2) 在"Telink defined event" 中增加了一个" BLT_EV_FLAG_PHY_UPDATE"Event, 具体细节请参考前面"Controller Event"章节介绍。

(3) API:

```
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);
```

BLE Spec 标准接口，详细请参考《Core_v5.0》(Vol 2/PartE/7.8.49 "LE Set PHY Command")。

connHandle：BLS_CONN_HANDLE。

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

3.3 BLE Host

3.3.1 BLE Host 介绍

BLE host 包括 L2CAP、ATT、SMP、GATT、GAP 等层，用户层的应用都是基于 host 层实现的。

3.3.2 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 App 层，向下连接 Controller 层，发挥 Host 与 Controller 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。

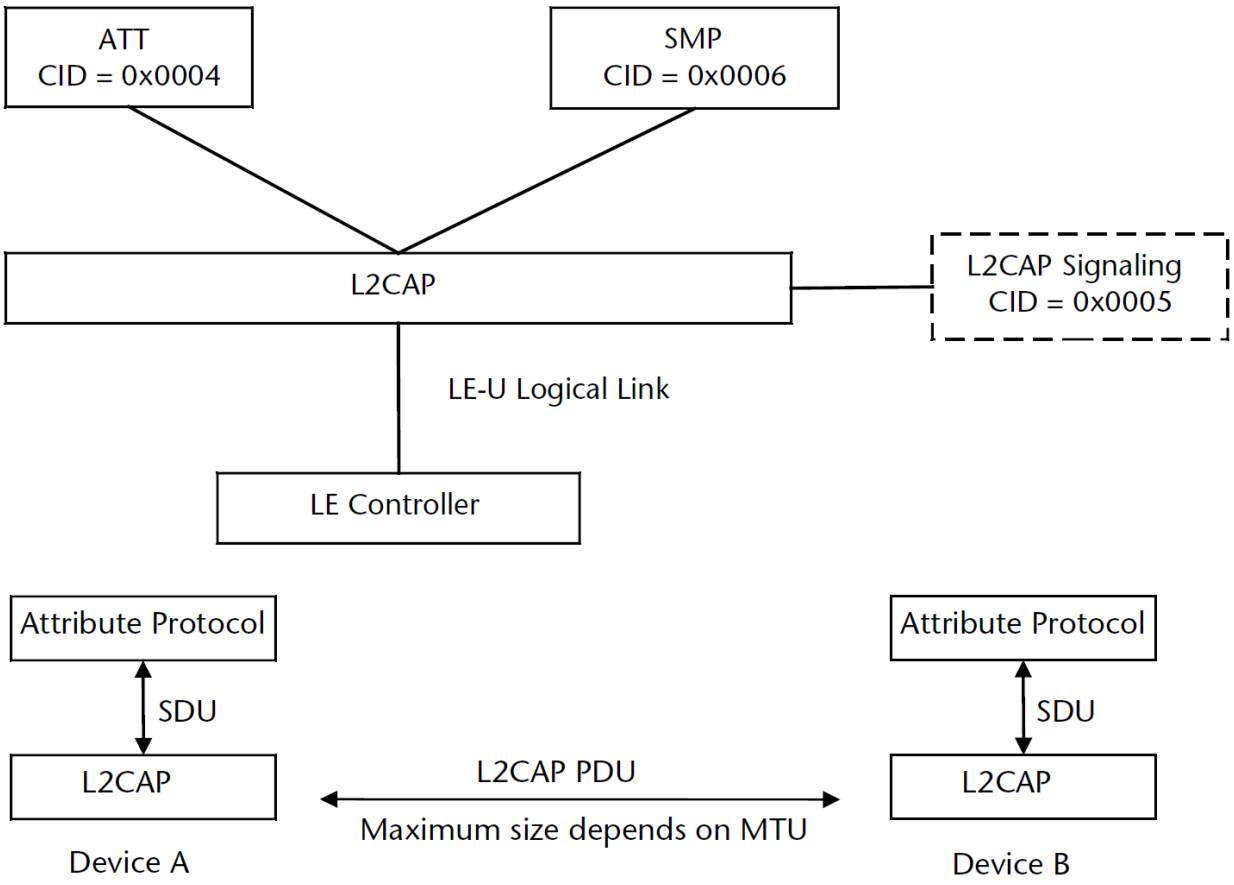


Figure 3.27: BLE L2CAP 结构以及 ATT 组包模型

L2CAP 根据 BLE Spec 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。

3.3.2.1 注册 L2CAP 数据处理函数

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

```
void    blc_l2cap_register_handler (void *p);
```

在 8208_ble_sample/8208_module 等 BLE Slave 应用中，SDK L2CAP 层处理 Controller 数据的函数为：

```
int     blc_l2cap_packet_receive (u16 connHandle, u8 * p);
```

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_l2cap_register_handler (blc_l2cap_packet_receive);
```

在 8208 hci 中，只实现了 slave controller，blc_hci_sendACLData2Host 函数将 controller 数据通过 UART/USB 等硬件接口传送给 BLE Host 设备。

```
int blc_hci_sendACLData2Host (u16 handle, u8 *p)
```

初始化：

```
blc_l2cap_register_handler (blc_hci_sendACLData2Host);
```

3.3.2.2 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 层的 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《Core_v5.0》(Vol 3/Part A/ 4.20 "CONNECTION PARAMETER UPDATE REQUEST")。

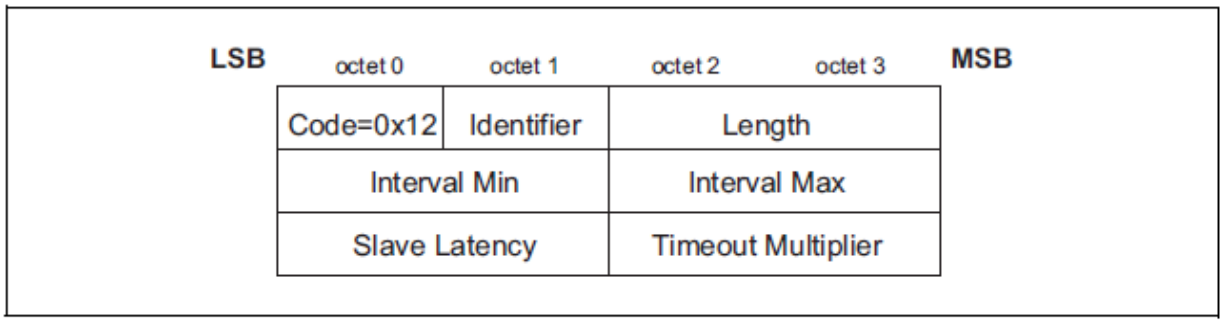


Figure 4.22: Connection Parameters Update Request Packet

Figure 3.28: BLE 协议栈中 Connection Para update Req 格式

该 BLE SDK 在 L2CAP 层上提供了 slave 主动申请更新连接参数的 API，用来向 master 发送上面这个 CONNECTION PARAMETER UPDATE REQUEST 命令。

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

以上四个参数跟 CONNECTION PARAMETER UPDATE REQUEST 的 data 区域中四个参数对应。注意 min_interval 和 max_interval 的值是实际 interval 时间值除以 1.25 ms（如申请 7.5ms 的连接，该值为 6），timeout 的值为实际 supervision timeout 时间值除以 10ms（如 1 s 的 timeout 该值为 100）。

应用举例：在 connection 建立的时候，申请更新连接参数。

```
void    task_connect (u8 e, u8 *p, int n)
    {
        bls_l2cap_requestConnParamUpdate (6, 6, 99, 400);
        bls_l2cap_setMinimalUpdateReqSendingTime_after_connCreate(1000);
    }
```

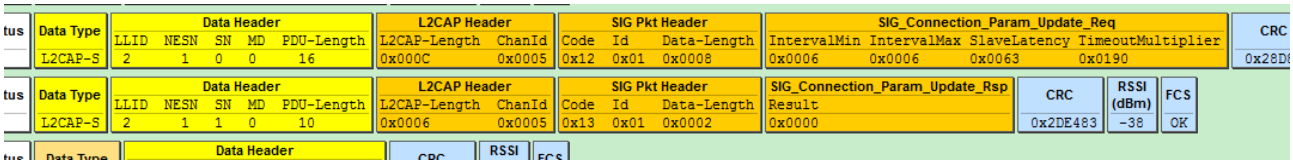


Figure 3.29: 抓包显示 conn para update reqeust 和 response

其中 API:

```
void bls_l2cap_setMinimalUpdateReqSendingTime_after_connCreate(int time_ms)
```

用于设置在连接建立后,slave 设备等待 time_ms(单位:毫秒)后执行 API:bls_l2cap_requestConnParamUpdate，来更新连接参数。用户如果在连接建立后，只调用 bls_l2cap_requestConnParamUpdate，则 slave 设备在连接建立后 1s 再执行发送连接参数更新请求。

(2) master 回应更新申请

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《Core_v5.0》(Vol 3/Part A/ 4.20 "CONNECTION PARAMETER UPDATE RESPONSE")。

下图所示为该命令格式及 result 含义。result 为 0x0000 时表示接受该命令，result 为 0x0001 时表示拒绝该命令。

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，基本上每一家都是不同的，这里没法提供一个统一的标准，只能靠 user 在平时的 master 兼容性测试中去慢慢总结归纳。

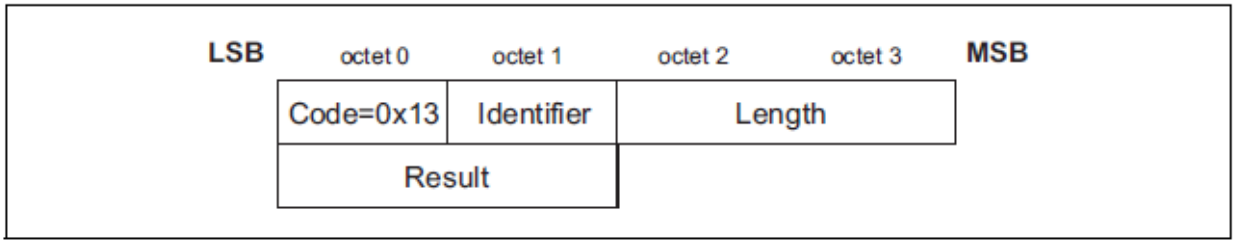


Figure 4.23: Connection Parameters Update Response Packet

The data field is:

- *Result (2 octets)*

  The result field indicates the response to the Connection Parameter Update Request. The result value of 0x0000 indicates that the LE master Host has accepted the connection parameters while 0x0001 indicates that the LE master Host has rejected the connection parameters.

Result	Description
0x0000	Connection Parameters accepted
0x0001	Connection Parameters rejected
Other	Reserved

Figure 3.30: BLE 协议栈中 conn para update rsp 格式

(3) Master 在 Link Layer 上更新连接参数

Slave 发送 conn para update req，并且 master 回 conn para update rsp 接受申请后，master 会发送 link layer 层的 LL_CONNECTION_UPDATE_REQ 命令，如下图所示。

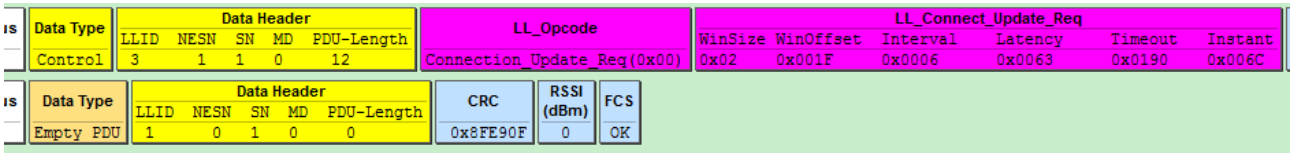


Figure 3.31: 抓包显示 ll conn update req

slave 收到此更新请求后，记下最后一个参数为 master 端的 instant 的值，当 slave 端的 instant 值到达这个值的时候，更新到新的连接参数，并触发回调事件 BLT_EV_FLAG_CONN_PARA_UPDATE。

instant 是 master 和 slave 端各自都维护的连接事件计数值，范围为 0x0000~0xffff，在一个连接中，它们的值一直都是相等的。当 master 发送 conn_req 申请和 slave 连接后，master 开始切换自己的状态（从扫描状态到连接状态），并将 master 端的 instant 清 0。slave 收到 conn_req，从广播状态切换到连接状态，将 slave 端的 instant 清 0。master 和 slave 的每一个连接包都是一个连接事件，两端在 conn_req 后的第一个连接事件，instant 值为 1，第二个连接事件 instant 值为 2，依次往后递增。

当 master 发送 LL_CONNECTION_UPDATE_REQ 时，最后一个参数 instant 是指在标号为 instant 的连接事件时，master 将使用 LL_CONNECTION_UPDATE_REQ 包中前几个连接参数对应值。由于 slave 和 master 的 instant 值始终是相等的，它收到 LL_CONNECTION_UPDATE_REQ 时，在自己的 instant 等于 master 所声明的那个 instant 的连接事件时，使用新的连接参数。这样就可以保证两端在同一个时间点完成连接参数的切换。

3.3.3 ATT and GATT

3.3.3.1 GATT 基本单位 Attribute

GATT 定义了两种角色：Server 和 Client。该 BLE SDK 中，Slave 是 Server，对应的 Android、iOS 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。

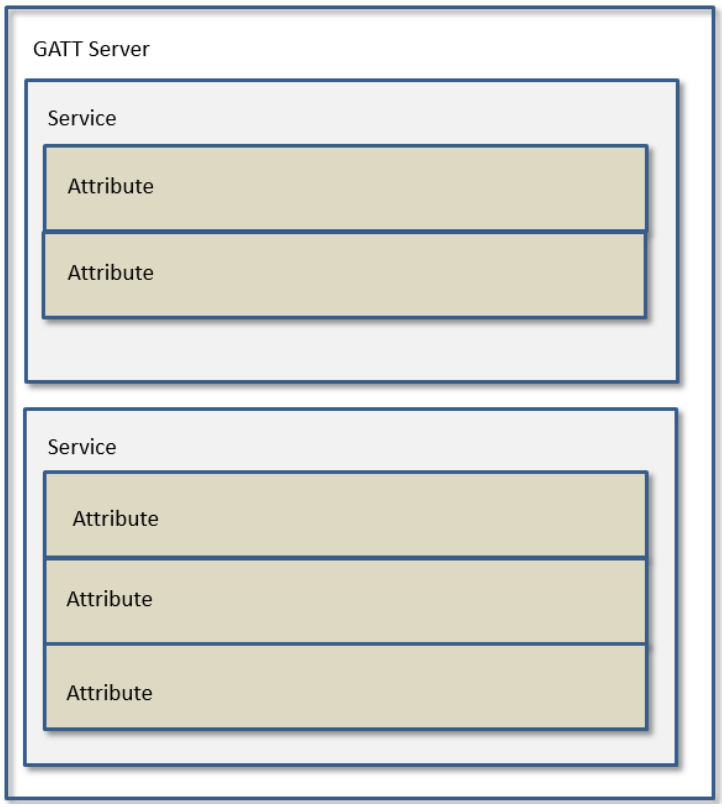


Figure 3.32: Attribute 构成 GATT service

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 master 设备都遵循同一套转换方法，将 2 个 bytes 的 UUID 转换成 16 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。该 BLE 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/service/uuid.h。

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

(2) Attribute Handle

slave 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 BLE SDK 中，Attribute Value 用指针和指针所指区域的长度来描述。

3.3.3.2 Attribute and ATT Table

为了实现 slave 端的 GATT service，该 BLE SDK 设计了一个 Attribute Table，该 Table 由多个基本的 Attribute 组成。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 BLE 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，如下所示部分代码：

```
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},
};
```

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 第一个作用是表示当前 Attribute Table 中有效 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 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。

attNum 第二个作用是用于指定当前的 service 由哪几个 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。

如上面代码所示，gap service UUID 为 GATT_UUID_PRIMARY_SERVICE 的那个 Attribute 上 attNum 为 7，则 Attribute Handle 1~ Attribute Handle 7 这 7 个 Attribute 是属于 gap service 的描述。

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 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 = 2 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，master 来读这个 Attribute 时，UUID 会是 0x2803。

b) Telink 私有的 16 bytes UUID：

如 audio 中 MIC 的 Attribute，相关代码：

```
#define TELINK_MIC_DATA {0x18,0x2B,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,
↪  0x01,0x0}
const u8 my_MicUUID[16]     = TELINK_MIC_DATA;
static u8 my_MicData        = 0x80;
{0,1,16,1,(u8*)(&my_MicUUID), (u8*)(&my_MicData), 0},
```

uuid 指向 my_MicData 在 flash 中的地址，uuidLen 为 16，master 来读这个 Attribute 时，UUID 会是 0x000102030405060708090a0b0c0d2b18。

(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，该 BLE 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=35 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，hid information 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，设置要读的 AttHandle = 0x23 = 35，对应 SDK 中 Attribute Table 中的 hid information。

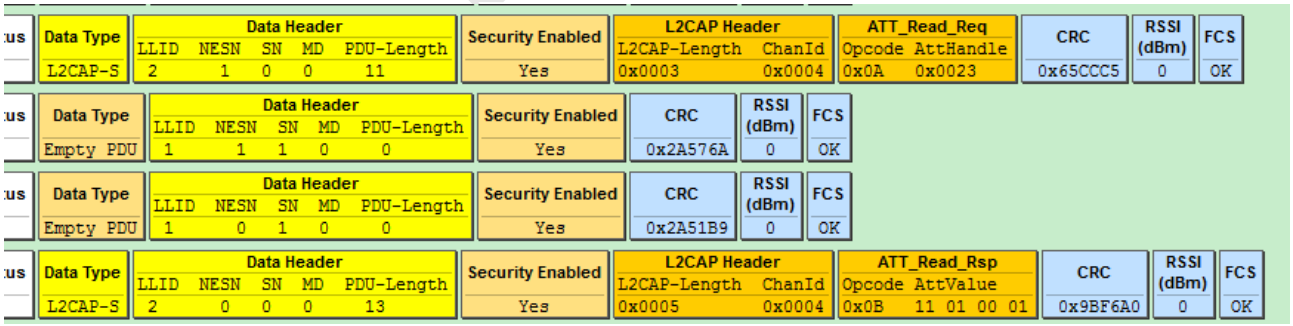


Figure 3.33: master 读 hidInformation 的 BLE 抓包

Attribute Handle=40 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。

此外，当 att 表存在 CHARACTERISTIC_UUID_HID_REPORT_MAP 时，很多设备都要识别操作系统，然后根据操作系统去发送不同的 report map，所以底层在处理时候可以将对应的 pAttrValue 和 attrLen 重新注册，API 接口如下。

```
ble_sts_t bls_att_setHIDReportMap(u8* p,u32 len);
```

如果需要恢复 Attribute Table 中定义的预设值，调用以下 API 接口。

```
ble_sts_t bls_att_resetHIDReportMap();
```

(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)(u16 connHandle, 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 pAttrValue 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)。另外，回调函数 w 的返回值为 0，代表写成功，其他返回值请参考枚举 att_err_t 或者按照协议要求 user 自行定义。

3.4.5.1 Write Request

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

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

Figure 3.34: BLE 协议栈中 Write Request

3.4.5.3 Write Command

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

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

Figure 3.35: 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 3.36: 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;
    u16 l2capLen;
    u16 chanId;
    u8  opcode;
    u16 handle;
    u8  dat[20];
}rf_packet_att_data_t;
```

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

```
int my_WriteCallback (void *p)
{
    rf_packet_att_data_t *pw = (rf_packet_att_data_t *)p;
    int len = pw->l2cap - 3;
    //add your code
    //valid data is pw->dat[0] ~ pw->dat[len-1]
    return 0;
}
```

上面这个结构体 rf_packet_att_data_t 所在位置为 stack/ble/ble_format.h。

(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)(u16 connHandle,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）,connHandle 为 master 和 slave 之间的连接句柄，slave 应用填 BLS_CONN_HANDLE; master 应该填 BLM_CONN_HANDLE。

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 user 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 user 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 user 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
const u16 my_primaryServiceUUID = GATT_UUID_PRIMARY_SERVICE;
```

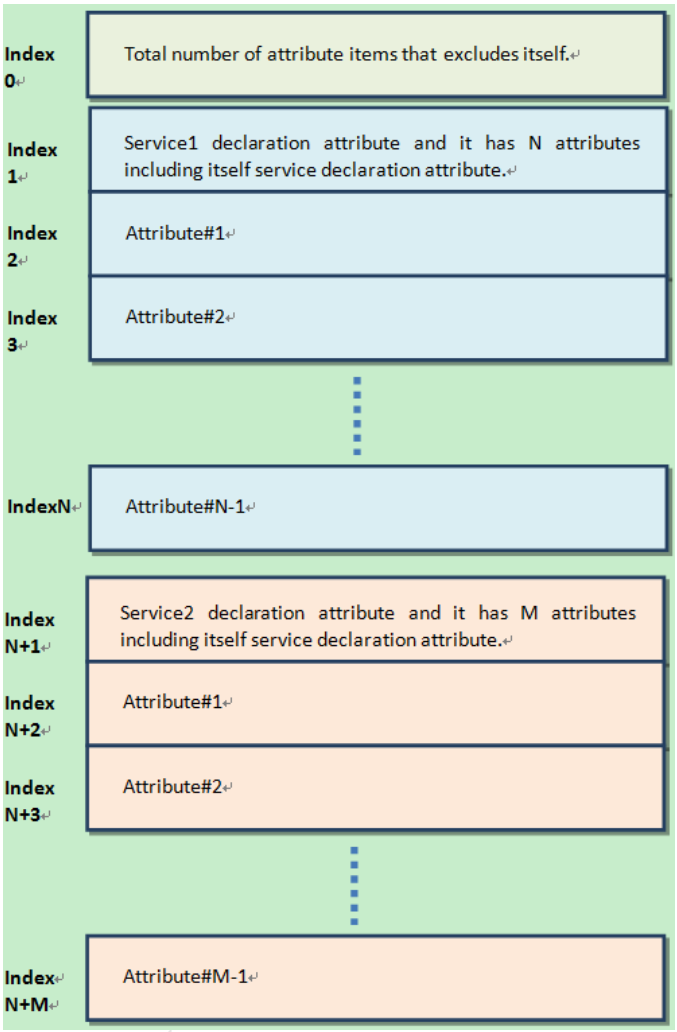


Figure 3.37: Service Attribute Layout

(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。

3.3.3.3 Attribute PDU and GATT API

根据 BLE Spec，该 BLE SDK 目前支持的 Attribute PDU 有以下几类：

- Requests：client 发送给 server 的数据请求。
- Responses：server 收到 client 的 request 后发送的数据回应。
- 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 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。

(1) Read by Group Type Request, Read by Group Type Response

Read by Group Type Request 和 Read by Group Type Response 详请参照《Core_v5.0》(Vol 3/Part F/3.4.4.9 and 3.4.4.10)。

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。

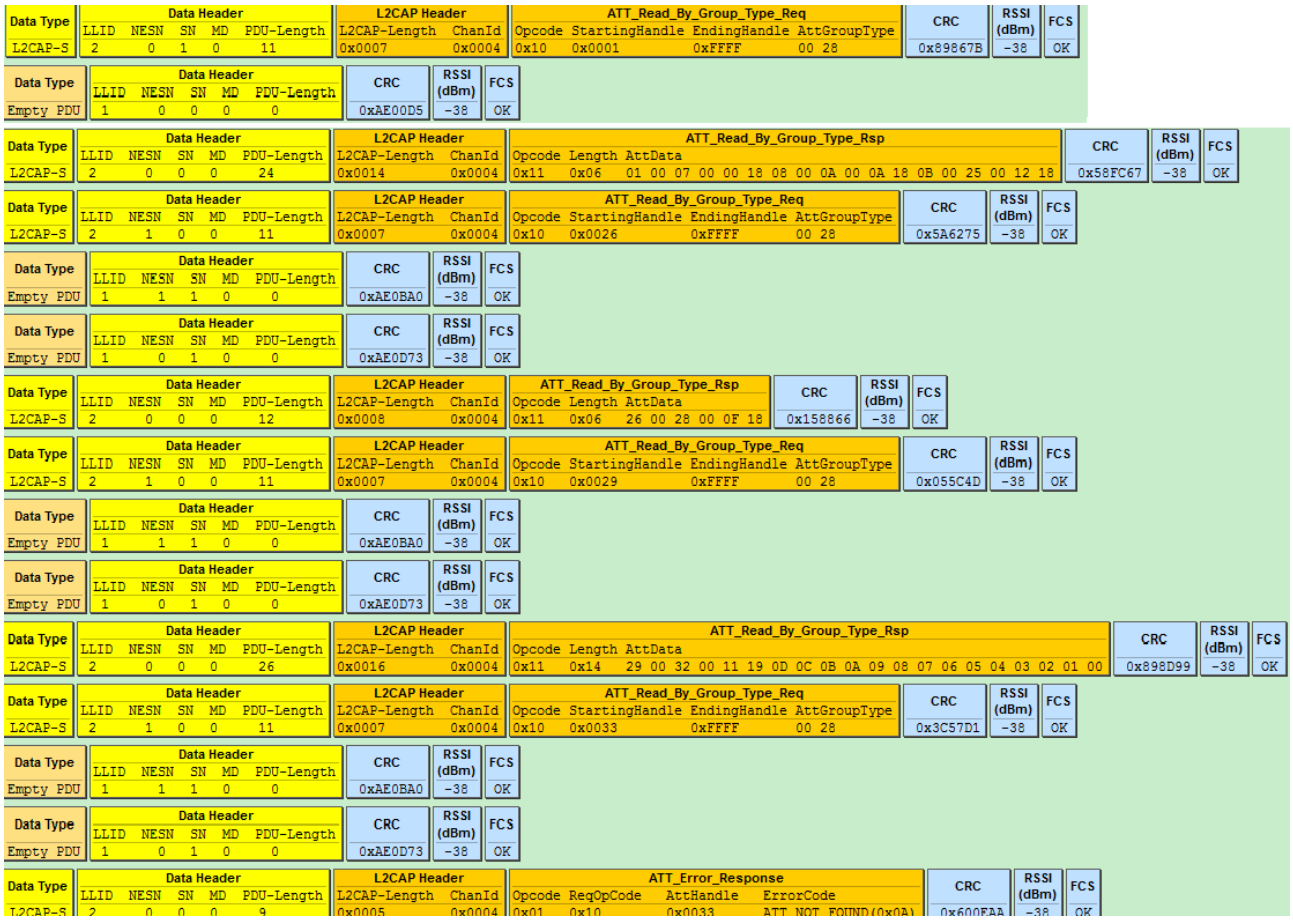


Figure 3.38: Read by Group Type Request Read by Group Type Response

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，slave 的 Attribute table 中有以下几组符合该要求：

a) attHandle 从 0x0001 ~ 0x0007 的 Attribute Group,

Attribute Value 为 SERVICE_UUID_GENERIC_ACCESS (0x1800).

b) attHandle 从 0x0008 ~ 0x000a 的 Attribute Group,

Attribute Value 为 SERVICE_UUID_DEVICE_INFORMATION (0x180A).

c) attHandle 从 0x000B ~ 0x0025 的 Attribute Group,

Attribute Value 为 SERVICE_UUID_HUMAN_INTERFACE_DEVICE (0x1812).

d) attHandle 从 0x0026 ~ 0x0028 的 Attribute Group,

Attribute Value 为 SERVICE_UUID_BATTERY (0x180F).

e) attHandle 从 0x0029 ~ 0x0032 的 Attribute Group,

Attribute Value 为 TELINK_AUDIO_UUID_SERVICE(0x11, 0x19, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00).

slave 将以上 5 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。

(2) Find by Type Value Request, Find by Type Value Response

Find by Type Value Request 和 Find by Type Value Response 详请参照《Core_v5.0》(Vol 3/Part F/3.4.3.3 and 3.4.3.4)。

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。

Data Type	Data Header					L2CAP Header		ATT_Find_By_Type_Value_Req					CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttType	AttValue			
L2CAP-S	2	1	1	0	13	0x0009	0x0004	0x06	0x0001	0xFFFF	0x2800	0A 18	0x4CEA12	-54	OK

Data Type	Data Header					CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length			
Empty PDU	1	0	0	0	0	0xC4C0E8	-54	OK

Data Type	Data Header					L2CAP Header		ATT_Find_By_Type_Value_Rsp		CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	HandleInfo			
L2CAP-S	2	1	0	0	9	0x0005	0x0004	0x07	0C 00 0E 00	0xF92ED9	-54	OK

Figure 3.39: Find by Type Value Request Find by Type Value Response

(3) Read by Type Request, Read by Type Response

Read by Type Request 和 Read by Type Response 详请参照《Core_v5.0》(Vol 3/Part F/3.4.4.1 and 3.4.4.2)。

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。

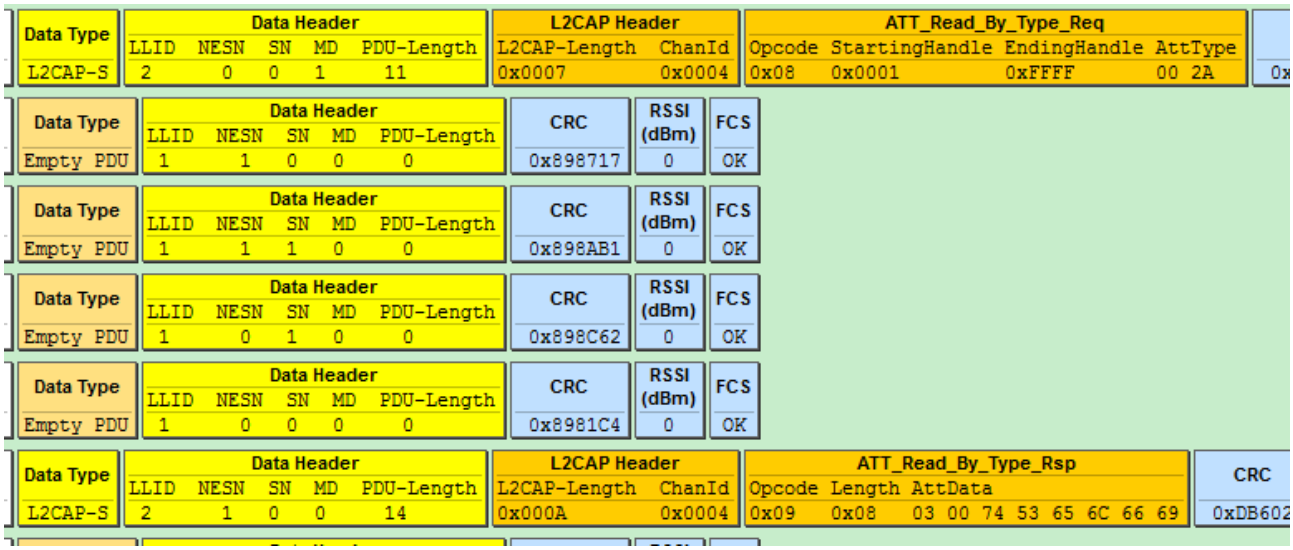


Figure 3.40: Read by Type Value Request Find by Type Value Response

上图所示，master 读 attType 为 0x2A00 的 Attribute，slave 中 Attribute Handle 为 00 03 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。

(4) Find information Request, Find information Response

Find information request 和 Find information response 详请参照《Core_v5.0》(Vol 3/Part F/3.4.3.1 and 3.4.3.2)。

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。

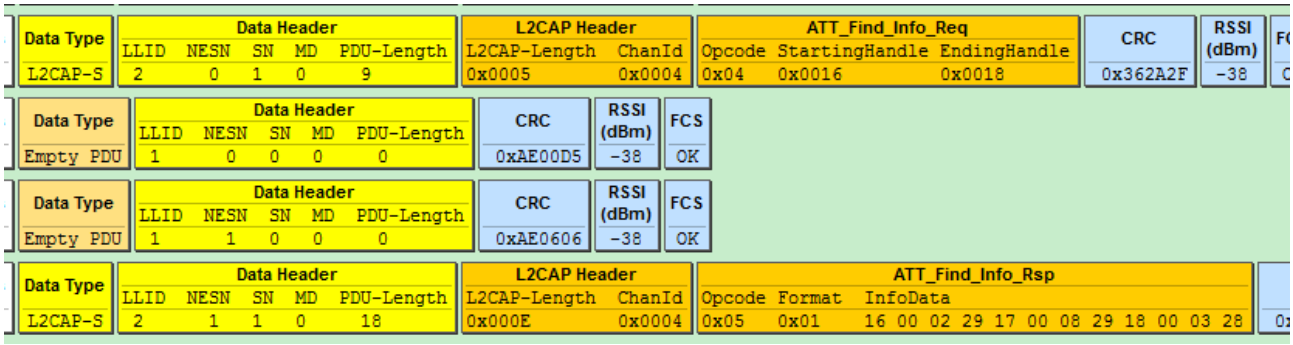


Figure 3.41: Find Information Request Find Information Response

(5) Read Request, Read Response

Read Request 和 Read Response 详请参照《Core_v5.0》(Vol 3/Part F/3.4.4.3 and 3.4.4.4)。

master 发送 Read Request，指定某一个 attHandle，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。

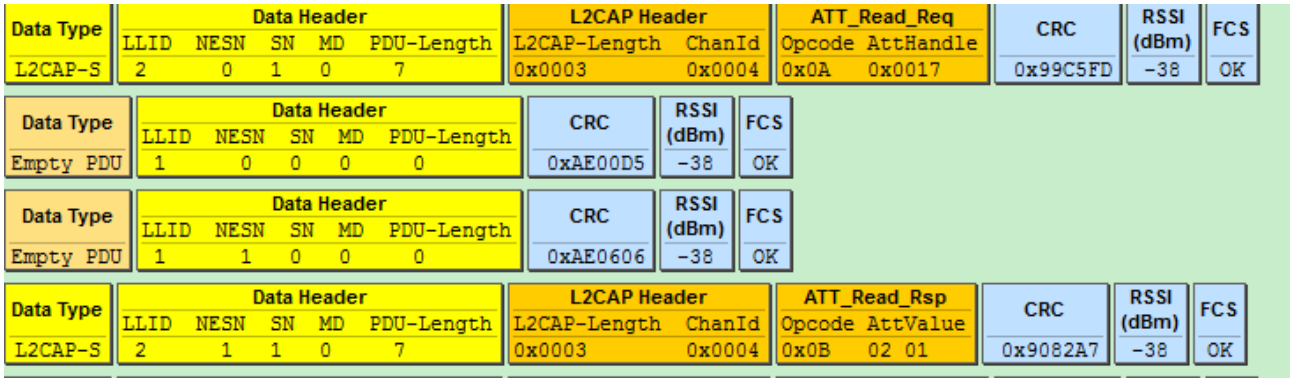


Figure 3.42: Read Request Read Response

(6) Read Blob Request, Read Blob Response

Read Blob Request 和 Read Blob Response 详请参照《Core_v5.0》(Vol 3/Part F/3.4.4.5 and 3.4.4.6)。

当 slave 某个 Attribute 的 Attribute Value 值的长度超过 MTU_SIZE（目前 SDK 中为 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。

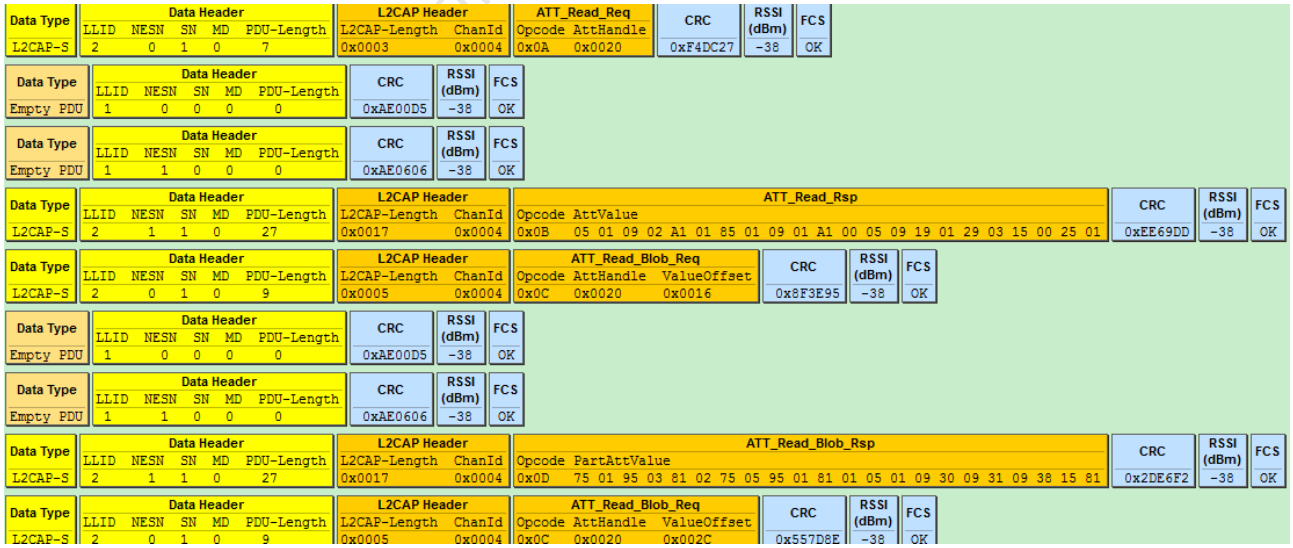
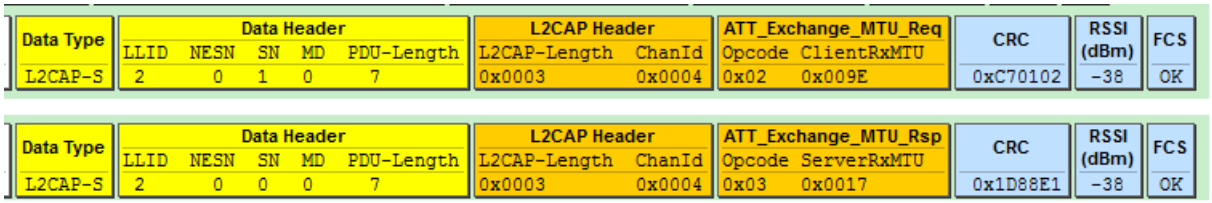


Figure 3.43: Read Blob Request Read Blob Response

(7) Exchange MTU Request, Exchange MTU Response

Exchange MTU Request 和 Exchange MTU Response 详请参照《Core_v5.0》(Vol 3/Part F/3.4.2.1 and 3.4.2.2)。

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 3.44: Exchange MTU Request Exchange MTU Response

当 Telink BLE Slave 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 master 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。

a) 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”小节会详细介绍 GAP event。

b) Slave GATT 层收长包数据的处理。

B80 Slave ServerRxMTU 默认为 23，实际最大 ServerRxMTU 可以支持到 250，即 master 端 250 个 byte 的分包数据在 Slave 端可以正确的完成数据包重拼。当应用中需要使用到 master 的分包重拼时，使用下面 API 先修改 Slave 端的 RX size：

```
ble_sts_t   blc_att_setRxMtuSize(u16 mtu_size);
```

返回值列表：

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	添加成功
GATT_ERR_INVALID_ PARAMETER	见 SDK 中定义	mtu_size 大于最大值 250

如果 Master GATT 层有长包数据需要发送给 Slave，Master 端会主动发起 ATT_Exchange_MTU_req，此时 Slave 回复 ATT_Exchange_MTU_rsp，其中 ServerRxMTU 为上面 API：blc_att_setRxMtuSize 设置的值。如果用户注册了 GAP event，并且开启了 eventMask：GAP_EVT_MASK_ATT_EXCHANGE_MTU，用户可以在 GAP event 回调函数中获取 EffectiveRxMTU 和 Master 端的 ClientRxMTU。

c) Slave GATT 层发长包数据的处理。

当 b80 Slave 需要在 GATT 层发送长包数据时，需要先获取到 Master 的 Client RxMTU，最终的数据长度不能大于 ClientRxMTU。

Slave 先使用 API blc_att_setRxMtuSize 设置自己的 ServerRxMTU，假设设为 158。

```
blc_att_setRxMtuSize (158) ;
```

再调用下面 API 主动发起一个 ATT_Exchange_MTU_req：

```
ble_sts_t    blc_att_requestMtuSizeExchange (u16 connHandle, u16 mtu_size);
```

connHandle 为 Slave conection 的 ID，为 BLS_CONN_HANDLE，mtu_size 为 ServerRxMTU。

```
blc_att_requestMtuSizeExchange(BLS_CONN_HANDLE,  158);
```

Master 收到 ATT_Exchange_MTU_req 后，回复 ATT_Exchange_MTU_rsp，SDK 收到 rsp 后，计算 EffectiveRxMTU。如果用户注册了 GAP event 并且开启了 eventMask：GAP_EVT_MASK_ATT_EXCHANGE_MTU，则 EffectiveRxMTU 和 ClientRxMTU 会上报给用户。

(8) Write Request, Write Response

Write Request 和 Write Response 详请参照《Core_v5.0》(Vol 3/Part F/3.4.5.1 and 3.4.5.2)。

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。

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

Data Type	Data Header					CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length			
Empty PDU	1	0	0	0	0	0xAE00D5	-38	OK

Data Type	Data Header					CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length			
Empty PDU	1	1	0	0	0	0xAE0606	-38	OK

Data Type	Data Header					L2CAP Header		ATT_Write_Rsp	CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode			
L2CAP-S	2	1	1	0	5	0x0001	0x0004	0x13	0xFBDB12	-38	OK

Figure 3.45: Write Request Write Response

(9) Write Command

Write Command 详请参照《Core_v5.0》(Vol 3/Part F/3.4.5.3)。

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。

(10) Queued Writes

Queued Writes 包含 Prepare Write Request/Response 和 Execute Write Request/Response 等 ATT 协议，详请内容可以参照《Core_v5.0》(Vol 3/Part F/3.4.6/Queued Writes)。

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。

备注：当前 SDK 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。如果长度大于 244 字节，则需要调用以下 API 接口，对 prepare write buffer 及长度进行修改：

```
void  blc_att_setPrepareWriteBuffer(u8 *p, u16 len)
```

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。

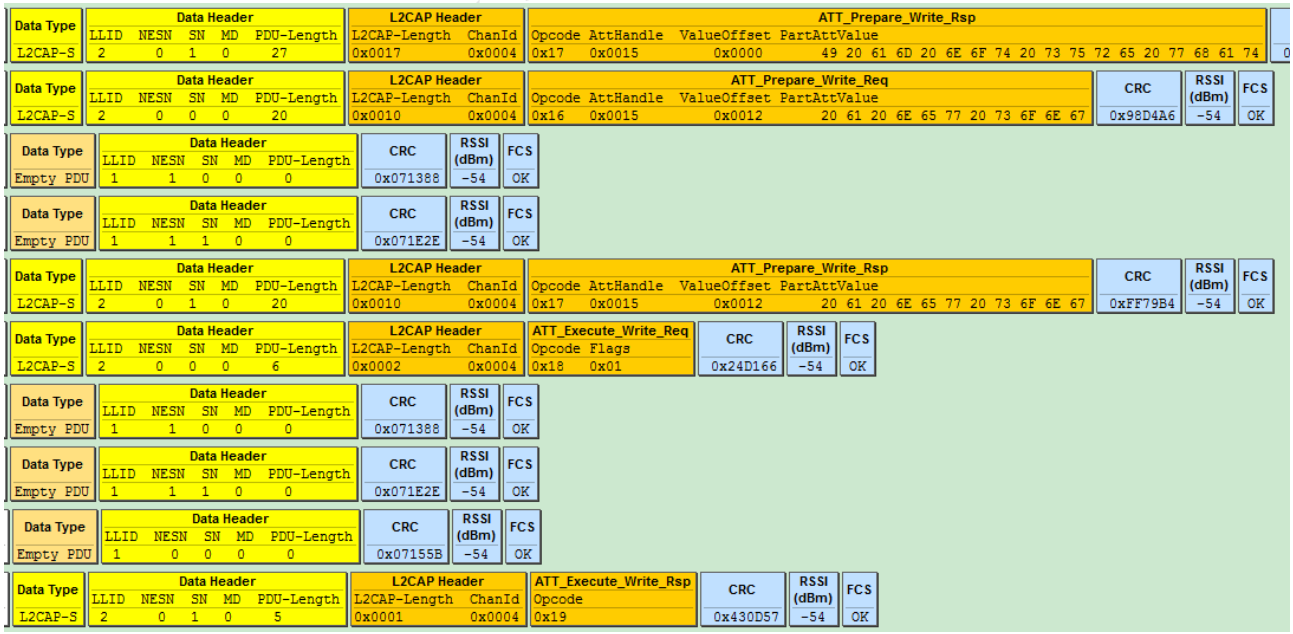


Figure 3.46: Write Long Characteristic Values 示例

(11) Handle Value Notification

Handle Value Notification 详请参照《Core_v5.0》(Vol 3/Part F/3.4.7.1)。

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 3.47: BLE Spec 中 Handle Value Notification

上图所示为 BLE Spec 中 Handle Value Notification 的格式。

该 BLE SDK 提供 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 handle, u8 *p, int len);
```

handle 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 TX 最大发送字节数，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。

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。

建议应用层调用该 API 时，检查一下返回值是否为 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	成功
LL_ERR_CONNECTION_NOT_ ESTABLISH	见 SDK 中定义	Link Layer 处于 None Conn state
SMP_ERR_PAIRING_BUSY	见 SDK 中定义	处于配对阶段，不能发送数据
LL_ERR_ENCRYPTION_BUSY	见 SDK 中定义	处于加密阶段，不能发送数据
LL_ERR_TX_FIFO_NOT_ ENOUGH	见 SDK 中定义	有大数据量任务在运行，软件 Tx fifo 不够用
GATT_ERR_DATA_LENGTH_ EXCEED_MTU_SIZE	见 SDK 中定义	len 大于 ATT_MTU-3 时，发送的数据长度超出了 ATT 层支持的最大数据长度

ble_sts_t	Value	ERR Reason
GATT_ERR_DATA_PENDING_DUE_TO_SERVICE_DISCOVERY _BUSY	见 SDK 中定义	处于遍历服务阶段，不能发数据

(12) Handle Value Indication

Handle Value Indication 详请参照《Core_v5.0》(Vol 3/Part F/3.4.7.2)。

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 3.48: Handle Value Indication in BLE Spec

上图所示为 BLE Spec 中 Handle Value Indication 的格式。

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

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

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 TX 最大发送字节数，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。

BLE Spec 里规定了每一个 indicate 的数据，都要等到 Master 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。建议应用层调用该 API 时，检查一下返回值是否为 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	添加成功
LL_ERR_CONNECTION_NOT_ESTABLISH	见 SDK 中定义	Link Layer 处于 None Conn state

ble_sts_t	Value	ERR Reason
SMP_ERR_PAIRING_BUSY	见 SDK 中定义	处于配对阶段，不能发送数据
LL_ERR_ENCRYPTION_BUSY	见 SDK 中定义	处于加密阶段，不能发送数据
LL_ERR_TX_FIFO_NOT_ENOUGH	见 SDK 中定义	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	见 SDK 中定义	处于遍历服务阶段，不能发数据
GATT_ERR_PREVIOUS_INDICATE_DATA_HAS_NOT_CONFIRMED	见 SDK 中定义	前一个 indicate 数据还没有被 master 确认
GATT_ERR_DATA_LENGTH_EXCEED_MTU_SIZE	见 SDK 中定义	len 大于 ATT_MTU-3 时，发送的数据长度超出了 ATT 层支持的最大数据长度

(13) Handle Value Confirmation

Handle Value Confirmation 详请参照《Core_v5.0》(Vol 3/Part F/3.4.7.3)。

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，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 3.49: BLE Spec 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。

3.3.3.4 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 BLE spec 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 3.50: 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：

(1) No authentication and no encryption

(2) Unauthenticated pairing with encryption

(3) Authenticated pairing with encryption

(4) 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 Aut
#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 3.51: 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，比如我们 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。

GATT 安全级别的示例用户可以参 8208_feature/ feature_gatt_security/app.c。

3.3.4 SMP

Security Manager（SM）在 BLE 中的主要目的是为 LE 设备提供加密所需要的各种 Key，确保数据的机密性。加密链路可以确保避免第三方"攻击者"拦截、破译或者读取空中数据原始内容。SMP 详细内容请用户参考《Core_v5.0》(Vol 3/Part H/ Security Manager Specification)。

3.3.4.1 SMP 安全等级

BLE 4.2 Spec 新增了一种称作安全连接（LE Secure Connections）配对方式，新配对方式在安全性方面得到进一步增强，而 BLE4.2 以前的配对方式我们统称传统配对（LE legacy pairing）。

回顾"GATT service Security"小节，可知本地设备的配对状态类型如下：

Local Device Pairing Status			
No LTK No STK	Unauthenticated LTK or Unauthenticated STK	Authenticated LTK or Authenticated STK	Authenticated LTK with Secure Connections

Figure 3.52: 本地设备配对状态

这四个状态分别对应 LE 安全模式 1 的四个级别, 详情可以参考《Core_v5.0》(Vol 3//Part C/10.2 LE SECURITY MODES)

(1) No authentication and no encryption (LE security mode1 level1)

(2) Unauthenticated pairing with encryption (LE security mode1 level2)

(3) Authenticated pairing with encryption (LE security mode1 level3)

(4) Authenticated LE Secure Connections (LE security mode1 level4)

注意：

> 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）master 对端设定能支持的最高安全级别 >=slave 本端设定能支持的最高安全级别；
> b）本端和对端按照各自设定的 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，用户即便是设置 slave 端能够支持的最高安全等级是 mode1 level3，但是连接 slave 的 master 设置为不支持配对加密（最高只支持 mode1 level1），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 mode1 level1。

用户可以通过如下 API 设置 SM 能支持的最高安全等级：

```
void blc_smp_setSecurityLevel(le_security_mode_level_t  mode_level);
```

枚举类型 le_security_mode_level_t 具体定义介绍如下：

```
typedef enum {
    LE_Security_Mode_1_Level_1 = BIT(0),        No_Authentication_No_Encryption = BIT(0),
↪   No_Security = BIT(0),
    LE_Security_Mode_1_Level_2 = BIT(1),        Unauthenticated_Paring_with_Encryption = BIT(1),
```

```
    LE_Security_Mode_1_Level_3 = BIT(2),        Authenticated_Paring_with_Encryption = BIT(2),
    LE_Security_Mode_1_Level_4 = BIT(3),
 ↪  Authenticated_LE_Secure_Connection_Paring_with_Encryption =BIT(3),
.....
}le_security_mode_level_t;
```

3.3.4.2 SMP 参数配置

Telink BLE SDK 中 SMP 参数配置介绍主要围绕 SMP 四个安全等级的配置展开，slave 的 SMP 功能目前能支持的最高级别是 LE security mode1 level4；master 的 SMP 功能目前支持传统配对方式下的最高级别是 LE security mode1 level2（传统配对 Just Works 方式）。

(1) LE security mode1 level1

安全级别 1 表示设备不支持加密配对，如果需要禁用 SMP 功能，用户只需要在初始化地方需调用如下函数:

```
blc_smp_setSecurityLevel(No_Security);
```

表示设备端不会对当前连接进行配对加密过程，即使对方请求配对加密时，设备端也会拒绝配对加密。一般用于当前设备不支持设备加密配对的过程。如下图，master 发起配对请求，slave 回复 SM_Pairing_Failed。

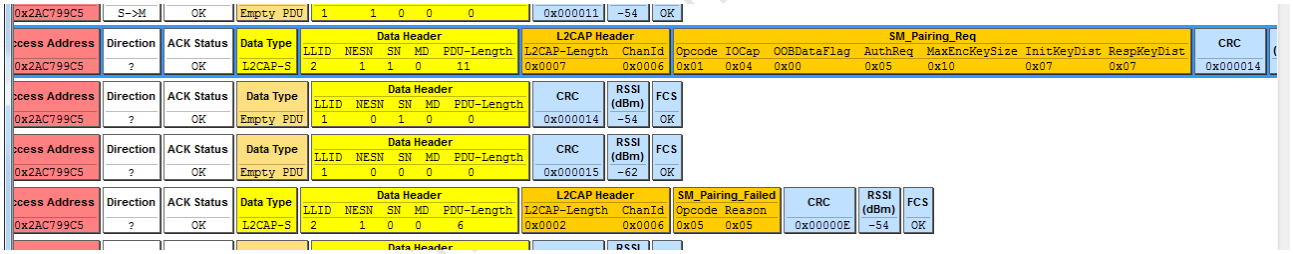


Figure 3.53: 抓包显示 Pairing Disable

(2) LE security mode1 level2

安全级别 2 表示设备最高支持 Unauthenticated_Paring_with_Encryption，如传统配对和安全连接配对方式下的 Just Works 配对模式。

a) 通过前文 SMP 基本概念的描述，我们知道 SM 配对方法包括传统加密和安全连接配对，SDK 提供了如下 API 用于设置是否支持 BLE4.2 新加密特性：

```
void blc_smp_setParingMethods (paring_methods_t  method);
```

枚举类型 paring_methods_t 具体定义介绍如下:

```
typedef enum {
    LE_Legacy_Paring      = 0,    // BLE 4.0/4.2
    LE_Secure_Connection = 1,    // BLE 4.2/5.0/5.1
}paring_methods_t;
```

b) 使用 LE security mode1 level1 以外的安全级别配置就必须要调用如下 API 用于初始化 SMP 各参数配置，包括绑定区域 FLASH 的初始化配置：

```
int blc_smp_peripheral_init (void);
```

如果在初始化阶段只调用了该 API，则 SDK 会使用默认参数去配置 SMP：

- 默认支持的最高安全等级：Unauthenticated_Paring_with_Encryption；
- 默认绑定模式：Bondable_Mode（存储配对加密后分发的 KEY 到 FLASH）；
- 默认 IO 能力是 IO_CAPABILITY_NO_INPUT_NO_OUTPUT。

以上默认参数是按照传统配对 Just Works 模式配置的，所以用户只调用该 API，相当于配置了 LE security mode1 level2，通过 A 和 B 我们知道 LE security mode1 level2 有两种配置：

a) 设备具备在传统配对下 Just works 的初始化配置：

```
blc_smp_peripheral_init();
```

b) 设备具备在安全连接下 Just works 的初始化配置：

```
blc_smp_setParingMethods(LE_Secure_Connection);
blc_smp_peripheral_init();
```

(3) LE security mode1 level3

安全级别 3 表示设备最高支持 Authenticated pairing with encryption，如传统配对模式下的 Passkey Entry、Out of Band 等。

该级别需要设备支持 Authentication，也就是需要通过某种方法确保配对双方身份的合法性，BLE 给出了如下三种 Authentication 方式：

- 有人参与的方式，如设备具备按键或者显示能力，通过一方显示 TK，另一方输入相同的 TK（比如 Passkey Entry）；
- 配对的双方通过非 BLE RF 传输方式交互一些信息，进行后续的配对操作（如 Out of Band，一般通过 NFC 传输 TK）；
- 设备自行协商 TK(如 Just Works,两端设备均使用 TK：0)。需要注意的是第 3 种方式属于 Unauthenticated，所以 Just works 的安全级别对应 LE security mode1 level2。

Authentication 能确保配对双方身份的合法性，提供这种方式的保护又可以称为 MITM（Man in the Middle）中间人保护。

a) 具备 Authentication 的设备需要设置其 MITM flag 或 OOB flag，SDK 提供如下两个 API 用于设置 MITM 和 OOB flag 的值：

```
void blc_smp_enableAuthMITM (int MITM_en);
void blc_smp_enableOobAuthentication (int OOB_en);
```

其中参数 MITM_en、OOB_en 的值为 0 或者 1，0 对应失能；1 对应使能。

b) 根据 Authentication 方式的介绍，SM 提供了三类鉴权方式，这三类方式的选择依赖于配对双方具备的 IO 能力，我们 SDK 提供了如下接口用于配置当前设备具备的 IO 能力：

```
void blc_smp_setIoCapability (io_capability_t ioCapablility);
```

枚举类型 io_capability_t 具体定义介绍如下:

```
typedef enum {
    IO_CAPABILITY_UNKNOWN = 0xff,
    IO_CAPABILITY_DISPLAY_ONLY = 0,
    IO_CAPABILITY_DISPLAY_YESNO = 1,
    IO_CAPABILITY_KEYBOARD_ONLY = 2,
    IO_CAPABILITY_NO_IN_NO_OUT = 3,
    IO_CAPABILITY_KEYBOARD_DISPLAY = 4,
} io_capability_t;
```

c) 传统配对模式下 MITM、OOB flag 使用规则：

		Initiator			
		OOB Set	**OOB Not Set**	**MITM Set**	**MITM Not Set**
Responder	**OOB Set**	Use OOB	Check MITM		
	OOB Not Set	Check MITM	Check MITM		
	MITM Set			Use IO Capabilities	Use IO Capabilities
	MITM Not Set			Use IO Capabilities	Use Just Works

Table 2.6: Rules for using Out-of-Band and MITM flags for LE legacy pairing

Figure 3.54: 传统配对模式下 MITM、OOB flag 使用规则

设备会根据本地设备和对端设备的 OOB 以及 MITM flag 决定使用 OOB 方式还是根据 IO 能力决定选择什么样的 KEY 产生方式。下图是 SDK 根据 IO 能力映射关系选择不同的 KEY 产生方法 (行、列参数类型 io_capability_t)：

```
// 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*/][5 /*Initiator*/] = {
    { 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 },
};

#if SECURE_CONNECTION_ENABLE
static const stk_generationMethod_t gen_method_sc[5 /*Responder*/][5 /*Initiator*/] = {
    { 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 },
};
#endif
```

Figure 3.55: 根据不同 IO 能力映射 KEY 产生方法

这部分具体映射关系可以参考《core5.0》(Vol3/Part H/2.3.5.1 Selecting Key Generation Method)，文档不再展开介绍。

根据以上介绍，我们知道 LE security mode1 level3 有如下几种初始值配置方式：

a) 设备具备传统配对下 OOB 的初始化配置：

```
blc_smp_enableOobAuthentication(1);
blc_smp_peripheral_init(); //SMP 参数配置必须放在该 API 之前
```

这里因为涉及到 OOB 传输 TK 值，SDK 在应用层提供了相关的 GAP event 给用户，请参考"GAP event"章节。提供给用户设置 OOB TK 值的 API 如下：

```
void blc_smp_setTK_by_OOB (u8 *oobData);
```

参数 oobData 表示需要设置的 16 位 TK 值数组的头指针。

b) 设备具备传统配对下 Passkey Entry（PK_Resp_Dsply_Init_Input）的初始化配置：

```
blc_smp_enableAuthMITM(1);
blc_smp_setIoCapability(IO_CAPABILITY_DISPLAY_ONLY);
blc_smp_peripheral_init();//SMP 参数配置必须放在该 API 之前
```

c) 设备具备传统配对下 Passkey Entry（PK_Init_Dsply_Resp_Input 或者 PK_BOTH_INPUT）的初始化配置：

```
blc_smp_enableAuthMITM(1);
blc_smp_setIoCapability(IO_CAPABLITY_KEYBOARD_ONLY);
blc_smp_peripheral_init();//SMP 参数配置必须放在该 API 之前
```

这里因为涉及到用户输入 TK 值，SDK 在应用层提供了相关的 GAP event 给用户，请参考"GAP event"章节。提供给用户设置 Passkey Entry 的 TK 值 API 如下：

```
void blc_smp_setTK_by_PasskeyEntry (u32 pinCodeInput);
```

参数 pinCodeInput 表示设置的 pincode 值，范围在"0~999999"。在 Passkey Entry 方式下，master 显示 TK，slave 需要输入 TK 的情况下使用。

最终设备使用何种 Key 产生方式是基于配对连接的两端设备支持什么样的 SMP 安全等级的，如果 master 只支持安全等级 LE security mode1 level1，那么最终 slave 是不会启用 SMP 功能的，因为 master 不支持配对加密。

(4) LE security mode1 level4

安全级别 4 表示设备最高支持 Authenticated LE Secure Connections，如安全连接配对模式下的 Numeric Comparison、Passkey Entry、Out of Band 等。

根据以上介绍，我们知道 LE security mode1 level4 有如下几种初始值配置方式：

a) 设备具备安全连接配对下 Numeric Comparison 的初始化配置：

```
blc_smp_setParingMethods(LE_Secure_Connection);
blc_smp_enableAuthMITM(1);
blc_smp_setIoCapability(IO_CAPABLITY_DISPLAY_YESNO);
```

这里因为涉及到向用户显示数值比较值，SDK 在应用层提供了相关的 GAP event 给用户，请参考"GAP event"章节。提供给用户设置数值比较结果"YES"或"NO"值的 API 如下：

```
void blc_smp_setNumericComparisonResult(bool YES_or_NO);
```

参数 YES_or_NO：在数值比较配对方式下，用于给用户确认比较两端显示的数值是否一致。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 0："NO"。

b) 设备具备安全连接配对下 Passkey Entry 的初始化配置：

这部分用户初始化代码和 LE security mode1 level3 配置方式 B、C（传统配对 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_setParingMethods(LE_Secure_Connection);
.....//参考 LE security mode1 level3 配置方式 B、C
```

c) 设备具备安全连接配对下 Out of Band 的初始化配置：

该部分目前 SDK 并未实现，所以这里就不再做具体介绍。

(5) 下面再介绍几个额外的 SMP 参数配置相关的 API：

a) SDK 提供是否需要开启绑定功能的 API：

```
void blc_smp_setBondingMode(bonding_mode_t mode);
```

枚举类型 bonding_mode_t 具体定义介绍如下：

```
typedef enum {
    Non_Bondable_Mode = 0,
    Bondable_Mode     = 1,
}bonding_mode_t;
```

配置安全级别非 mode1 level1 的设备，必须要使能绑定功能，SDK 已经默认开启了，所以一般情况下用户不需要再调用该 API。

b) SDK 提供是否需使能 Key Press 功能的 API：

```
void blc_smp_enableKeypress (int keyPress_en);
```

表示 Passkey Entry 期间是否支持为 KeyboardOnly 设备提供一些必要的输入状态信息，因为目前 SDK 不支持该功能，所以参数必须设置为 0。

c) 在安全连接方式下是否启用 Debug 椭圆加密密匙对：

```
void blc_smp_setEcdhDebugMode(ecdh_keys_mode_t mode);
```

枚举类型 ecdh_keys_mode_t 具体定义介绍如下：

```
typedef enum {
    non_debug_mode = 0,//ECDH distribute private/public key pairs
    debug_mode = 1,//ECDH use debug mode private/public key pairs
} ecdh_keys_mode_t;
```

该 API 仅在安全连接配对情况下使用，由于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，这对调试开发就不那么友好了，用户无法通过 sniffer 工具抓取 BLE 空中包，进而进行数据分析调试，所以 BLE spec 也规定给出了一组用于 Debug 的椭圆加密私钥/公钥对，只要开启这个模式，BLE sniffer 工具就可以用已知的密钥去解密链路。

d) 通过如下 API 设置 SM 是否绑定、是否开启 MITM flag、是否支持 OOB、是否支持 Keypress notification，以及支持的 IO 能力（文档前面给的都是单独的配置 API，为了用户设置方便，SDK 也提供了统一配置 API）：

```
void blc_smp_setSecurityParameters (bonding_mode_t mode,int MITM_en,int OOB_en, int keyPress_en,
io_capability_t ioCapablility);
```

前面已经介绍了每个参数含义，这里就不再重复了。

3.3.4.3 SMP 安全请求配置

SMP 安全请求（Security Request）只有 slave 可以发送，所以这部分只对 slave 设备来说。

我们知道配对流程阶段 1 有一个可选的安全请求包（Security Request），该包的目的是使 slave 可以主动触发配对流程的开始。SDK 提供了如下 API 用于灵活地配置 slave 是否在连接后或者回连后立即还是 pending_ms 毫秒后再向 master 发送 Security Request，亦或是不发送 Security Request 以实现不同的配对触发组合：

```
blc_smp_configSecurityRequestSending( secReq_cfg newConn_cfg, secReq_cfg reConn_cfg, u16
↪ pending_ms);
```

枚举类型 secReq_cfg 具体定义介绍如下：

```
typedef enum {
    SecReq_NOT_SEND = 0,
    SecReq_IMM_SEND = BIT(0),
    SecReq_PEND_SEND = BIT(1),
}secReq_cfg;
```

每个参数的意义介绍如下：

- SecReq_NOT_SEND：连接建立后，slave 不会主动发送 Security Request；
- SecReq_IMM_SEND：连接建立后，slave 会立即发送 Security Request；
- SecReq_PEND_SEND：连接建立后，slave 等待 pending_ms（单位毫秒）后再决定是否发送 Security Request

(1) 首次连接，slave 在 pending_ms 毫秒前就收到 master 的 Pairing_request 包也不会再发送 Security Request；

(2) 在回连阶段，pending_ms 毫秒之前如果 master 已经发送 LL_ENC_REQ 加密回连链路，则不再发送 Security Request。

newConn_cfg：用于配置新设备，reConn_cfg：用于配置回连的设备。这里 SDK 在回连时也提供配置是否发配对请求的目的：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 发一下 Security Request 就会去触发 master 主动加密链路，所以 SDK 提供了 reConn_cfg 配置，客户可以根据实际需要配置。

注意：

> 当前函数只能在连接之前调用。建议在初始化的时候调用。

函数 blc_smp_configSecurityRequestSending 的输入参数有如下 9 种组合：

Table 3.10: blc_smp_configSecurityRequestSending 的输入参数组合

Parameter	SecReq_NOT_SEND	SecReq_IMM_SEND	SecReq_PEND_SEND
SecReq_NOT_SEND	第一次连接或者回连都不发 SecReq （参数 pending_ms 无效）	第一次连接不发 SecReq，回连立即发 SecReq （参数 pending_ms 无效）	第一次连接不发 SecReq，回连 pending_ms 毫秒后发 SecReq （* 见前面参数说明）
SecReq_IMM_SEND	第一次连接立即发 SecReq，回连不发 SecReq （参数 pending_ms 无效）	第一次连接或者回连都立即发 SecReq （参数 pending_ms 无效）	第一次连接立即发 SecReq，回连 pending_ms 毫秒后发 SecReq （* 见前面参数说明）
SecReq_PEND_SEND	第一次连接 pending_ms 毫秒后发 SecReq （* 见前面参数说明），回连不发 SecReq	第一次连接 pending_ms 毫秒后发 SecReq （* 见前面参数说明），回连立即发 SecReq	第一次连接或者回连都 pending_ms 毫秒后发 SecReq （* 见前面参数说明）

我们挑其中两组做一下详细说明，其他组合类似，不做具体介绍：

(1) newConn_cfg：SecReq_NOT_SEND

reConn_cfg：SecReq_NOT_SEND

pending_ms：此时该参数不起作用。

newConn_cfg：SecReq_NOT_SEND 表示新设备 slave 不会主动发起 Security Request，只有在对方发起配对请求时响应对方的配对请求。如果对方不发送配对请求，则不会进行加密配对。如下图，在 master 发送配对请求包 SM_Pairing_Req 时，slave 会响应，但是不会主动触发 master 发起配对请求。

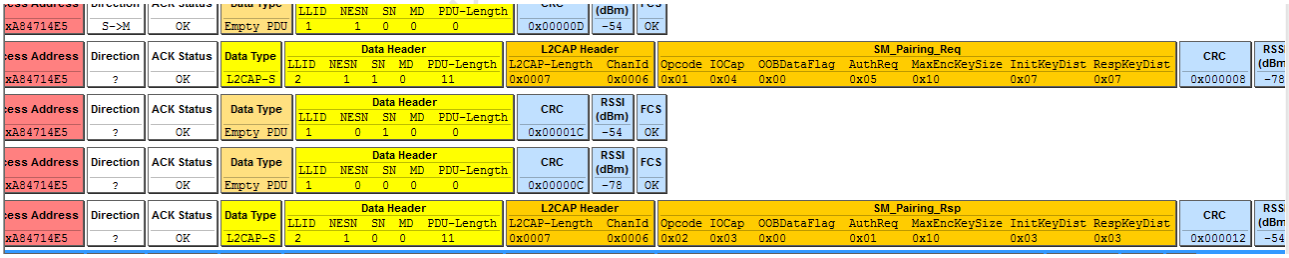


Figure 3.56: 抓包显示 Pairing Peer Trigger

reConn_cfg：SecReq_NOT_SEND 表示设备已经配对，回连时 slave 设备不会发送 Security Request。

(2) newConn_cfg：SecReq_IMM_SEND

reConn_cfg：SecReq_NOT_SEND

pending_ms：此时该参数不起作用。

newConn_cfg：SecReq_IMM_SEND 表示新设备 slave 一经连接便会主动向 master 发 Security Request，以触发 master 开始配对流程。如下图，slave 主动发送 SM_Security_Req 触发 master 发送配对请求：

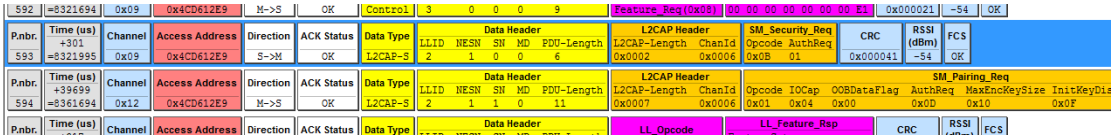


Figure 3.57: 抓包显示 Pairing Conn Trigger

reConn_cfg：SecReq_NOT_SEND 表示回连的时候 slave 不会发送 Security Request。

此外 SDK 还提供了一个单独发送 Security Request 包的 API，用于特殊应用场合，应用层可以随时调用该 API 发送 Security Request 包：

```
int blc_smp_sendSecurityRequest (void);
```

这里需要注意的是，用户如果使用 blc_smp_configSecurityRequestSending 管控安全配对请求包的话，就不要再使用调用 blc_smp_sendSecurityRequest 函数。

3.3.4.4 SMP 绑定信息说明

这里讨论的 SMP 绑定信息是对 slave 设备来说的。用户可以参考 SDK demo "8208_ble_sample"初始化中设置 direct advertising 的代码。

Slave 最多可以同时存储 4 个 master 的配对信息，这 4 个设备都可以回连成功。下面接口用于设定当前存储的最多设备数，不超过 4，如果用户不设置的话，默认值也为 4。

```
#define         SMP_BONDING_DEVICE_MAX_NUM                  4
ble_sts_t blc_smp_param_setBondingDeviceMaxNumber( int device_num);
```

如果设置了 blc_smp_param_setBondingDeviceMaxNumber (4)，配对 4 个设备后，一旦配对第 5 个，就会自动将最老的那个（第 1 个）设备的配对信息删除，然后存储第 5 个设备的配对信息。

如果设置了 blc_smp_param_setBondingDeviceMaxNumber（2），配对 2 个设备后，一旦配对第 3 个，就会自动将最老的那个（第 1 个）设备的配对信息删除，然后存储第 3 个设备的配对信息。

下面 API 用于获取当前 slave 在 flash 上存储的配对成功的 master 设备数量。

```
u8 blc_smp_param_getCurrentBondingDeviceNumber(void);
```

假设返回值为 3，就说明 flash 上目前存储了 3 个配对成功的设备，这 3 个设备都可以回连成功。

(1) 绑定信息存储顺序

和 BondingDeviceNumber 相关的一个概念叫 index。如果当前 BondingDeviceNumber 为 1，那么只有 1 个 bonding 设备，它的 index 为 0；如果 BondingDeviceNumber 为 2，两个设备的 index 分别为 0 和 1。

SDK 提供了两种 index 更新顺序方式：1、根据设备最近连接时间为顺序；2、根据设备配对时间先后为顺序。可以通过如下 API 设置 index 更新方式：

```
void    bls_smp_setIndexUpdateMethod(index_updateMethod_t method);
```

枚举类型 index_updateMethod_t 具体定义介绍如下：

```
typedef enum {
    Index_Update_by_Pairing_Order = 0,     //default value
    Index_Update_by_Connect_Order = 1,
} index_updateMethod_t;
```

下面分别介绍两种 index 更新方式：

a) 根据设备最近连接时间为顺序（Index_Update_by_Connect_Order）

如果 BondingDeviceNumber 为 2，两个设备的 index 分别为 0 和 1。Index 顺序是以最近成功的一次连接为准的，并不是最近一个配对为准：假设 slave 先和 masterA 配对成功，再和 masterB 配对成功，此时 slave 的 flash 存储上 masterA 为 index 0，masterB 为 index1，因为 masterB 是最近的设备。接着让 slave 和 masterA 回连一次成功，这时候 masterA 就成为最近的一个设备，此时 index 0 设备变为 masterB，index1 设备变为 masterA。

如果 BondingDeviceNumber 为 3，三个设备的 index 分别为 0、1、2，0 是最老连接的，2 是最新连接的。

如果 BondingDeviceNumber 为 4，四个设备的 index 分别为 0、1、2、3，此时 3 是最新连接的设备。根据上面的描述，若 slave 连续配对 masterA、B、C、D，此时 masterD 是 index3 设备，若 slave 和 masterB 回连一次，则 master B 成为最新的 index3 设备。

由于 index 是以最近连接时间为顺序的。所以也要注意设备配对超过 4 的情况：若连续配对 masterA、B、C、D，再配对 master E，则 slave 会删除最老的 masterA；但如果在配对 masterA、B、C、D 后，先和 masterA 回连一次，此时顺序变为 B、C、D、A，再配对 masterE 的话 slave 就会删除 masterB 的配对信息。

b) 根据设备配对时间先后为顺序（Index_Update_by_Pairing_Order）

如果 BondingDeviceNumber 为 2，两个设备的 index 分别为 0 和 1。Index 顺序是以配对时间先后为准：假设 slave 先和 masterA 配对成功，再和 masterB 配对成功，此时 slave 的 flash 存储上 masterA 为 index 0，masterB 为 index1，接着让 slave 和 masterA 回连一次成功，此时 index 0 设备依然为 masterA，index1 设备为 masterB。

如果 BondingDeviceNumber 为 4，四个设备的 index 分别为 0、1、2、3，0 是最老配对的设备，3 是最新配对的设备。根据上面的描述，若 slave 连续配对 masterA、B、C、D，此时 masterD 是 index3 设备，期间不管 slave 和 master A、B、C、D 什么顺序回连，index 0、1、2、3 依然分别对应 masterA、B、C、D。

注意：

> 设备配对超过 4 的情况：若连续配对 masterA、B、C、D，再配对 master E，则 slave 会删除最老的 masterA；如果在配对 masterA、B、C、D 后，先和 masterA 回连一次，此时顺序依旧 A、B、C、D，再配对 masterE 的话 slave 会删除 masterA 的配对信息。

(2) 绑定信息格式及相关 API 说明

Master 设备绑定信息存储在 flash 上，其格式为：

```
typedef struct {
    u8      flag;
    u8      peer_addr_type;  //address used in link layer connection
    u8      peer_addr[6];
    u8      peer_key_size;
    u8      peer_id_adrType; //peer identity address information in key distribution, used to
↪   identify
    u8      peer_id_addr[6];
    u8      own_ltk[16];      //own_ltk[16]
    u8      peer_irk[16];
    u8      peer_csrk[16];
}smp_param_save_t;
```

绑定信息共 64byte。

- peer_addr_type 和 peer_addr 是 link layer 上 master 的连接地址，设备 direct adv 时使用这个地址。
- peer_id_adrType/peer_id_addr 和 peer_irk 是 master 在 key distribution 阶段宣称的 identity address 和 irk。

只有当 peer_addr_type 和 peer_addr 是可解析的私有地址 (resolvable private addr，简称 RPA)，且用户需要使用地址过滤时，才需要将相关的信息添加到 resolving list 中，以便 slave 可以解析出（参考 feature_test 中 TEST_WHITELIST 的用法)。

其他参数 user 不用关注。

下面 API 使用 index 从 flash 上获取设备的信息。

```
u32  bls_smp_param_loadByIndex(u8 index, smp_param_save_t* smp_param_load);
```

返回值为 0 表示获取信息失败，非 0 的值为该信息区域 Flash 首地址。比如当前 bonding 设备为 3 时，获取最近的那个连接设备的相关信息：

```
bls_smp_param_loadByIndex(2, …)
```

下面接口使用 master 的地址（link layer 上的连接地址）从 flash 上获取 bonding 的设备的信息。

```
u32  bls_smp_param_loadByAddr(u8 addr_type, u8* addr, smp_param_save_t* smp_param_load);
```

返回值为 0 表示获取信息失败，非 0 的值为该信息区域 Flash 首地址。

下面 API 用于 slave 设备清除本地 FLASH 上存储的所有配对绑定信息：

```
void    bls_smp_eraseAllParingInformation(void);
```

需要注意的是，用户调用该 API 前需要确保设备处于非连接态。

下面 API 可以用于 slave 设备配置绑定信息存储在 FLASH 中的位置：

```
void bls_smp_configParingSecurityInfoStorageAddr(int addr);
```

其中参数 addr 可以根据实际需要修改，用户配置前可以参考文档中"SDK FLASH 空间的分配"章节，决定绑定信息放置在 FLASH 中合适区域。

3.3.4.5 SMP 失败管理

当 SMP 失败时，可以通过回调函数控制继续保持连接或是断开。其实现方法如下：

(1) 注册 gap 层的处理函数，打开事件 mask，事件为 GAP_EVT_SMP_PAIRING_FAIL ，如下：

```
    blc_gap_registerHostEventHandler( app_host_event_callback );
    blc_gap_setEventMask( GAP_EVT_MASK_SMP_PARING_FAIL );
```

(2) 修改处理函数中该 mask 下对应的处理：

```
int app_host_event_callback (u32 h, u8 *para, int n)
{
    u8 event = h & 0xFF;

    switch(event)
    {
        case GAP_EVT_SMP_PAIRING_FAIL:
        {
            gap_smp_paringFailEvt_t* p = (gap_smp_paringFailEvt_t*)para;
            //想要的操作
        }
        break;

        default:
        break;
    }

    return 0;
}
```

3.3.5 GAP

3.3.5.1 GAP 初始化

GAP 初始化分主从角色，slave 使用如下 API 初始化 GAP：

```
void blc_gap_peripheral_init(void);
```

Master 使用如下 API 初始化 GAP：

```
void  blc_gap_central_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。

3.3.5.2 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 Telink defined event。

Telink BLE SDK 中新增了 GAP event 处理，主要是协议栈事件分层更加清晰，协议栈处理用户层交互事件更加便捷，特别是 SMP 相关的处理，如 Passkey 的输入，配对结果通知用户等。

如果 user 需要在 App 层接收 GAP event，首先需要注册 GAP event 的 callback 函数，其次需要将对应 event 的 mask 打开。

GAP event 的 callback 函数原型和注册接口分别为：

```
typedef int (*gap_event_handler_t) (u32 h, u8 *para, int n);
void blc_gap_registerHostEventHandler (gap_event_handler_t  handler);
```

callback 函数原型中的 u32 h 是 GAP event 标记，底层协议栈多处会用到。

下面列出几个用户可能会用到的事件：

```
#define GAP_EVT_SMP_PAIRING_BEGIN                       0
#define GAP_EVT_SMP_PAIRING_SUCCESS                     1
#define GAP_EVT_SMP_PAIRING_FAIL                        2
#define GAP_EVT_SMP_CONN_ENCRYPTION_DONE                3
#define GAP_EVT_SMP_SECURITY_PROCESS_DONE               4
#define GAP_EVT_SMP_TK_DISPALY                          8
#define GAP_EVT_SMP_TK_REQUEST_PASSKEY                  9
#define GAP_EVT_SMP_TK_REQUEST_OOB                      10
#define GAP_EVT_SMP_TK_NUMERIC_COMPARE                  11
#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 8208_feature/feature_gatt_security/app.c 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_NONE                       0x00000000
#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_SECURITY_PROCESS_DONE     (1<<GAP_EVT_SMP_SECURITY_PROCESS_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。

Data Type	Data Header					L2CAP Header		SM_Security_Req	
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AuthReq
L2CAP-S	2	1	0	0	6	0x0002	0x0006	0x0B	01

Data Type	Data Header					L2CAP Header		SM_Pairing_Req					
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	IOCap	OOBDataFlag	AuthReq	MaxEncKeySize	InitKeyDist	RespKeyDist
L2CAP-S	2	1	1	0	11	0x0007	0x0006	0x01	0x03	0x00	0x01	0x10	0x02	0x03

Figure 3.58: master 发起 Pairing_Req

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_paringBeginEvt_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_paringSuccessEvt_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_paringFailEvt_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.h"文件。

配对失败值具体含义可以参照《Core_v5.0》(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_MASK_SMP_SECURITY_PROCESS_DONE

事件触发条件：第一次配对时，在 GAP_EVT_SMP_PAIRING_SUCCESS 事件之后触发，快速回连时，在 GAP_EVT_SMP_CONN_ENCRYPTION_DONE 事件之后触发。

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: paring ,
 ↪  encrypt with STK
} gap_smp_securityProcessDoneEvt_t;
```

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。

(6) 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。

用户 debug 时如需不使用底层随机生成的 6 位 pincode 码，可以手动设置一个用户指定的 pincode 码，例如“123456”，采用以下 API。

```
void    blc_smp_manualSetPinCode_for_debug(u16 connHandle, u32 pinCodeInput);
```

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。

关于 Passkey Entry 应用的实例，用户可以参考 SDK 提供的 demo "vendor/ 8208_feature/ feature_gatt_security/app.c"。

```
case GAP_EVT_SMP_TK_DISPALY:
{
    char pc[7];
    u32 pinCode = *(u32*)para;
    sprintf(pc, "%d", pinCode);
    printf("TK display:%s\n", pc);
}
break;
```

(7) 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 输入能力输入 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。

(8) 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 输入能力输入 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。

(9) 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。

关于数值比较应用的实例，用户可以参考 SDK 提供的 demo "vendor/8208_feature/feature_gatt_security/app.c"。

(10) 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。

(11) GAP_EVT_GATT_HANDLE_VLAUE_CONFIRM

事件触发条件：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，slave 收到该 confirm 时触发。

数据长度 n：0。

回传指针 p：NULL。

4 低功耗管理（PM）

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

8208 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: Sleep mode 说明

Module	suspend	deepsleep retention	deepsleep
Sram	100% keep	100% keep	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 电流在 60~70 uA 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。SDK 中为了降低功耗，在进入 suspend 低功耗处理时已经对部分模块设置了掉电模式，这时该模块的 digital register 也会掉电，唤醒后必须需要重新进行初始化配置，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) 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），增加了一种 sleep mode 3：deepsleep with Sram retention mode，简称 deepsleep retention（或 deep retention）。

Deepsleep retention mode 也是一种 deepsleep，MCU 绝大部分的硬件模块都断电，PM 硬件模块维持工作。功耗是在 deepsleep mode 基础上增加 retention Sram 消耗的电，电流在 2uA 左右。deepsleep 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 可以保持不掉电。

deepsleep mode 和 deepsleep retention mode 中在 analog register 部分都有极少数可以保持不掉电，这些不掉电的 analog register 包括：

a) 控制 GPIO 模拟上下拉电阻的 analog register

通过 API gpio_setup_up_down_resistor 设置的 GPIO 模拟上下拉电阻，或者在 app_config.h 中使用类似如下方式配置的 GPIO 模拟上下拉电阻，可以保持状态。

```
#define PULL_WAKEUP_SRC_PD5     PM_PIN_PULLDOWN_100K
```

参考文档 GPIO 模块的介绍。使用 gpio output 属于 digital register 控制的状态。8208 在 suspend 期间可以用 gpio output 来控制一些外围设备，但被切换为 deepsleep retention mode 后，gpio output 状态失效，无法在 sleep 期间准确的控制外围设备。此时可以使用 GPIO 模拟上下拉电阻的状态来代替实现：上拉 10K 代替 gpio output high，下拉 100K 代替 gpio output low。

b) PM 模块特殊的不掉电 analog regsiter

drivers/lib/include/pm.h 文件中的 DEEP_ANA_REG，如下 code 所示：

```
#define DEEP_ANA_REG0                       0x3a //initial value =0x00
#define DEEP_ANA_REG1                       0x3b //initial value =0x00,system used, user can not
 ↪  use
#define DEEP_ANA_REG2                       0x3c //initial value =0x0f
```

需要注意的是，客户不允许使用 ana_3b，该模拟寄存器留给底层 stack 使用，如果应用层代码有用到该寄存器，需要修改。因为不掉电模拟寄存器数量比较少，建议客户使用其每一个 bit 指示不同的状态位信息。

如下几组不掉电模拟寄存器可能会因为错误的 GPIO 唤醒而丢掉信息，比如 GPIO_PAD 高电平唤醒 deepsleep，但是在调用 cpu_sleep_wakeup 函数前 gpio 已经为高电平，这就会导致错误的 GPIO 唤醒，那么这些模拟寄存器值将会丢失。

```
#define DEEP_ANA_REG6                       0x35 //initial value =0x20
#define DEEP_ANA_REG7                       0x36 //initial value =0x00
#define DEEP_ANA_REG8                       0x37 //initial value =0x00
#define DEEP_ANA_REG9                       0x38 //initial value =0x00
#define DEEP_ANA_REG10                      0x39 //initial value =0xff
```

使用者可以在这几个 analog regsiter 中保存一些重要的的信息，deepsleep/deepsleep retention wake_up 后还可以读到之前存储的值。

4.1.2 低功耗唤醒源

8208 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。该 BLE SDK 中，只关注 2 种唤醒源，如下所示（注意 code 中 PM_TIM_RECOVER_START 和 PM_TIM_RECOVER_END 两个定义不是唤醒源）：

```
typedef enum {
    //available wake-up source for customer
    PM_WAKEUP_PAD              = BIT(0),
    PM_WAKEUP_TIMER            = BIT(2),
}SleepWakeupSrc_TypeDef;
```

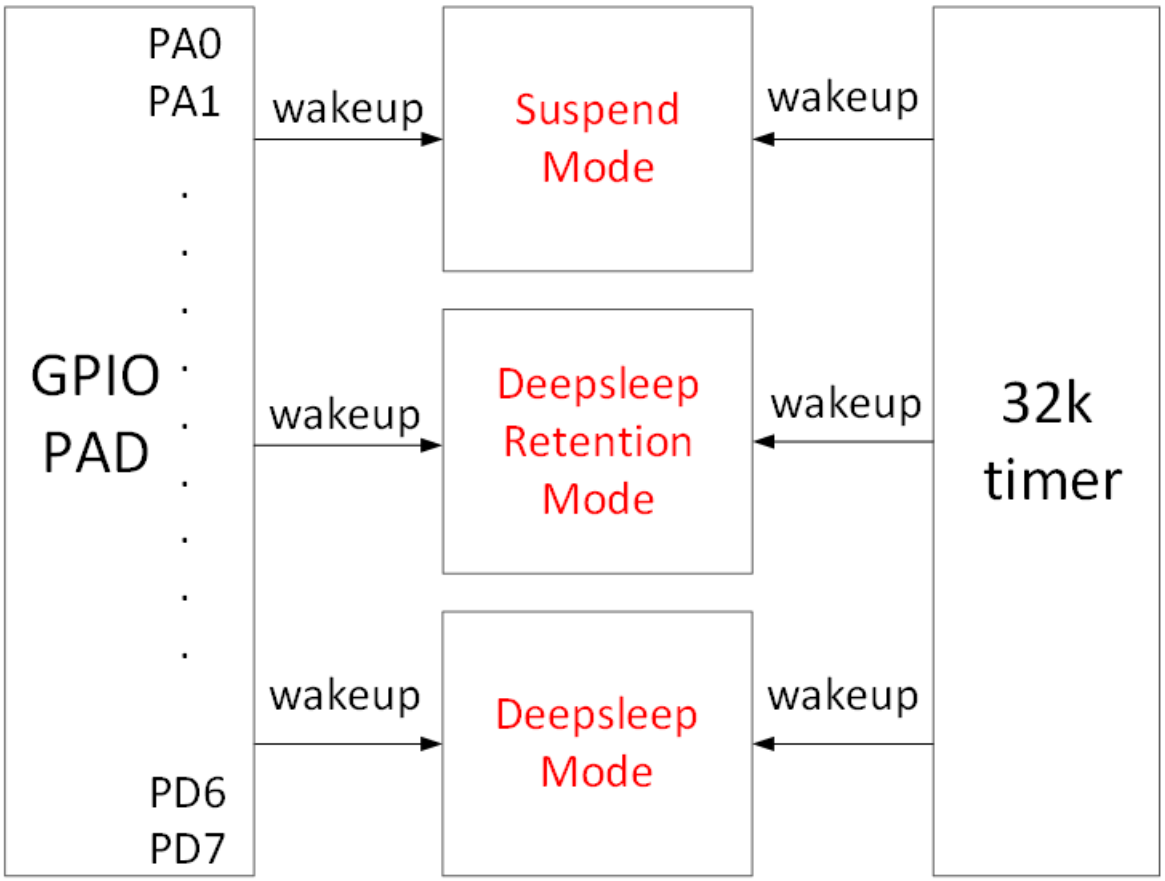


Figure 4.1: B80 MCU 硬件唤醒源

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 外所有的 GPIO 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 =1,
} 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

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 3 个选择，分别表示 suspend mode、deepsleep mode、deepsleep retention 16K Sram。

```
typedef enum {
        SUSPEND_MODE                        = 0x00,
        DEEPSLEEP_MODE                      = 0x30,
        DEEPSLEEP_MODE_RET_SRAM_LOW16K      = 0x01,
}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=234 s，即下面 delta_Tick 不能超过 234 s。

```
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_PAD              = BIT(0),
     WAKEUP_STATUS_TIMER            = BIT(2),
```

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

(1) WAKEUP_STATUS_TIMER 这个 bit 为 1，说明当前 sleep mode 是被 Timer 唤醒。

(2) WAKEUP_STATUS_PAD 这个 bit 为 1，说明当前 sleep mode 是被 GPIO PAD 唤醒。

(3) 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。

(4) 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 (SUSPEND_MODE , PM_WAKEUP_TIMER, clock_time() + 10*
↪  CLOCK_16M_SYS_TIMER_CLK_1MS);
```

程序执行该函数时进入 suspend mode，只能被 Timer 唤醒，唤醒时间为当前时间加上 10 ms，所以 suspend 时间为 10 ms。

```
cpu_sleep_wakeup (SUSPEND_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 suspend 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。

```
cpu_sleep_wakeup (DEEPSLEEP_MODE_RET_SRAM_LOW32K , PM_WAKEUP_TIMER, clock_time() + 8*
↪  CLOCK_16M_SYS_TIMER_CLK_1S);
```

程序执行该函数时进入 deepsleep retention 32K Sram mode，可被 Timer 唤醒，唤醒时间为执行该函数的 8s 后。

```
cpu_sleep_wakeup (DEEPSLEEP_MODE_RET_SRAM_LOW32K , PM_WAKEUP_PAD | PM_WAKEUP_TIMER,clock_time()
↪  + 10* CLOCK_16M_SYS_TIMER_CLK_1S);
```

程序执行该函数时进入 deepsleep retention 32K Sram mode，可被 GPIO PAD 和 Timer 唤醒，Timer 唤醒时间为执行该函数的 10s 后。如果在 10s 结束之前触发了 GPIO 动作，MCU 会被 GPIO PAD 唤醒；如果 10s 内无 GPIO 动作，MCU 会被 Timer 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。

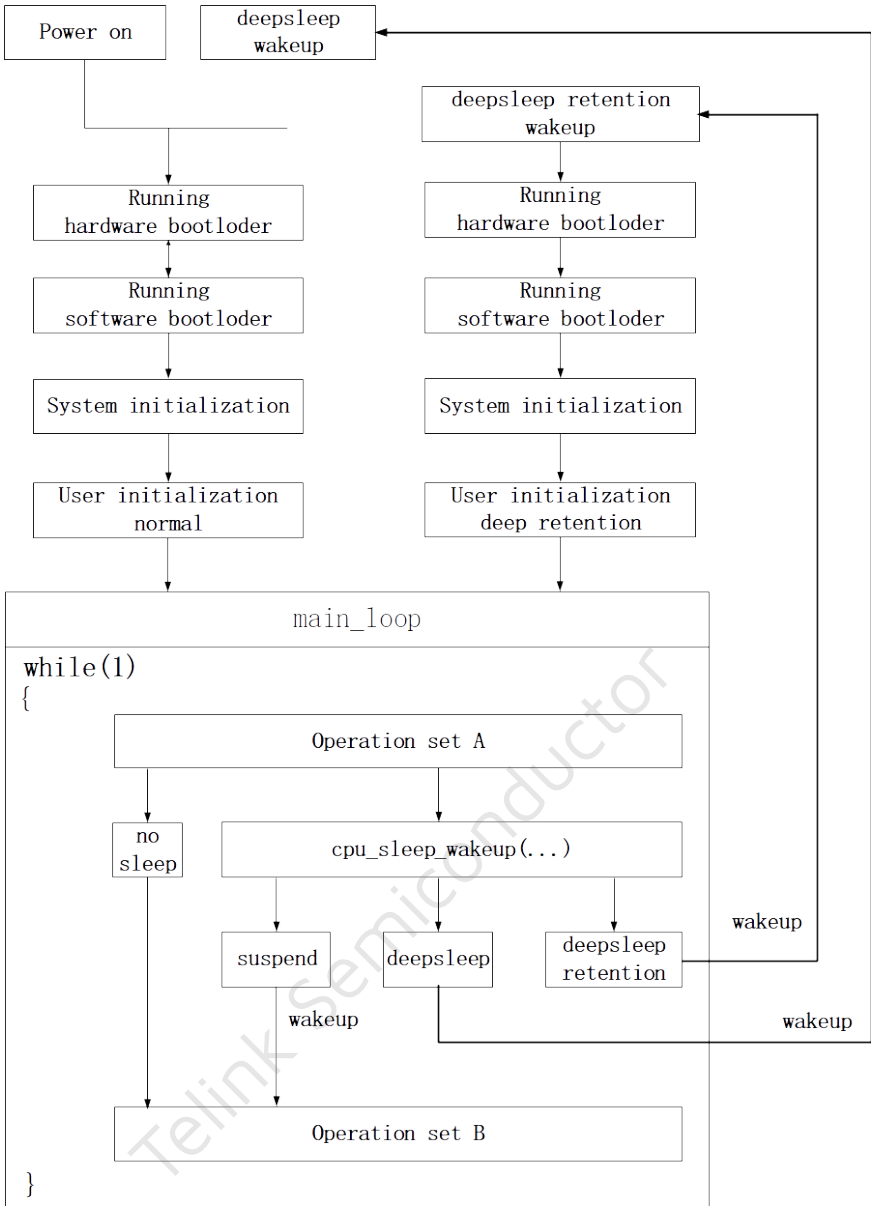


Figure 4.2: Sleep Mode Wakeup Work Flow

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（b80 sdk 的 boot 目录下的.s 汇编程序）。

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 分为 User initialization normal 和 User initialization deep retention 两种，分别对应 SDK 中函数 user_init_normal 和 user_init_deepRetn。对于不使用 deep retention 的工程，比如 kma master dongle，只有 user_init，等同于 user_init_normal。

user_init 和 user_init_normal 需要把所有的初始化都做一遍，时间较长。

user_init_deepRetn 只需要做一些硬件相关的初始化，无需软件初始化，因为在设计的时候将软件初始化涉及到的变量都放到了 MCU retention 区域中，deep retention 期间这些变量保持不变，醒来后无需重设。所以 user_init_deepRetn 是一个加速模式，节省了时间也就节省了功耗。

(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 wakeup work flow，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（或 32K）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）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 initilization 可以做一些优化改进，和 MCU power on/deepsleep wake_up 后的 user initilization 做区分处理，参考本文档后面的介绍。

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

该 BLE SDK 低功耗管理是对 BLE Link Layer 功耗的管理，请参考本文档前面对 Link Layer 的介绍。

SDK 中已对 Advertising state 和 Connection state Slave role 做了低功耗管理，并且提供了一套 API 供 user 调用和配置。

对于 Idle state，SDK 也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。下面 code 为一个简单的 demo：当 Link Layer 处于 Idle state 时，每个 main_loop suspend 10ms。

```
void main_loop (void)
{
    ////////////////////// BLE entry /////////////////////////
    blc_sdk_main_loop();

    ////////////////////// UI entry /////////////////////////
    // add user task
    ////////////////////// PM configuration /////////////////////////
    if(blc_ll_getCurrentState() == BLS_LINK_STATE_IDLE ){  //Idle state
        cpu_sleep_wakeup(SUSPEND_MODE, PM_WAKEUP_TIMER,
clock_time() + 10*CLOCK_16M_SYS_TIMER_CLK_1MS);
    }
    else{
        blt_pm_proc();  //BLE Adv & Conn state
    }
}
```

当 Link Layer 处于 Advertising state 或 Conn state Slave role 时，下图所示为 sleep mode 的时序。

注意：

> 图中 Conn state Slave role 为 connection latency = 0 的情况。

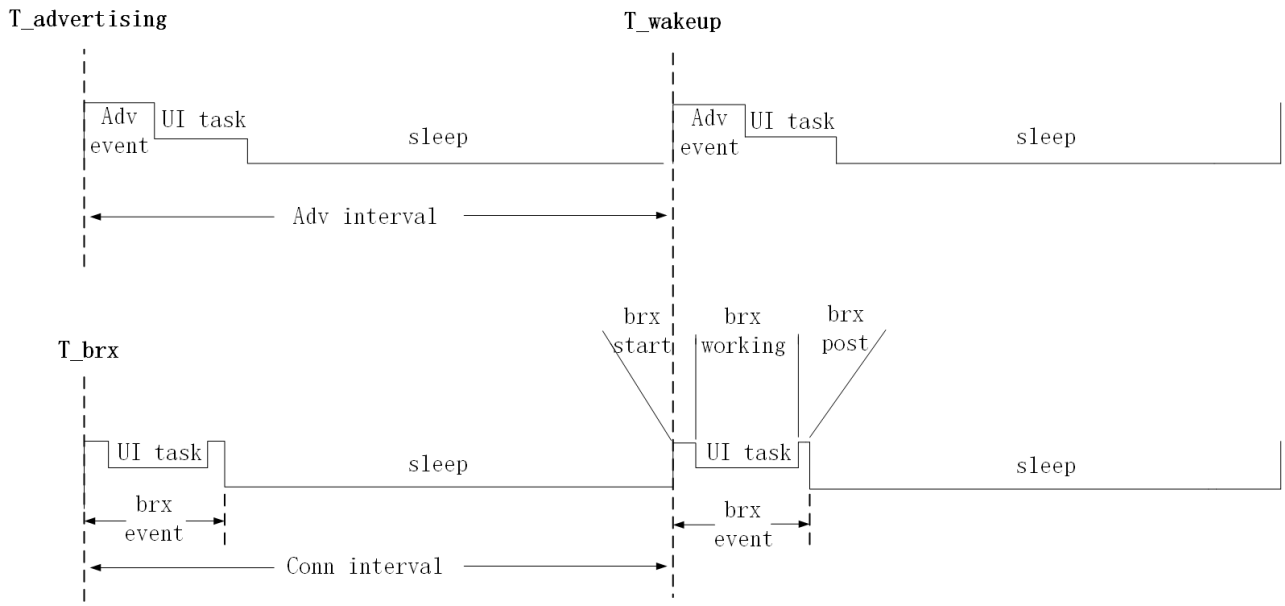


Figure 4.3: Sleep Timing for Advertising State and Conn State Slave Role

(1) 处于 Advertising state 时，每个 Adv Interval 里，Adv Event 时间是必须的，除去 UI task 所占用的时间，剩余时间 MCU 可以进入 sleep mode (suspend/ deepsleep retention)。

图中，第一个 Adv interval 上 Adv event 开始的时间我们定义为 T_advertising；sleep 需要唤醒的时间我们定义为 T_wakeup，T_wakeup 也是下一个 Adv interval 上 Adv event 的开始。T_advertising 和 T_wakeup 在本文档后面的介绍中需要使用到。

(2) 处于 Conn state Slave role 时，每个 Conn interval 内，brx Event (brx start+brx working+brx post) 时间是必须的，除去 UI task 占用的时间，剩余时间 MCU 可以进入 sleep mode (suspend/deepsleep retention)。

图中，第一个 Connection interval 上 Brx event 开始的时间我们定义为 T_brx；sleep 需要唤醒的时间我们定义为 T_wakeup，T_wakeup 也是下一个 Connection interval 上 Brx event 的开始。T_brx 和 T_wakeup 在本文档后面的介绍中需要使用到。

BLE 低功耗管理的实质是对上面两个状态的 sleep 时间进行管理，user 可以决定如何使用这些时间：不进入 sleep、进入 suspend mode 或进入 deepsleep retention mode。

8208 的 sleep mode 分 3 种：suspend、deepsleep、deepsleep retetnion。

对于 sleep mode 中的 suspend 和 deepsleep retention，user 不需要调用 API cpu_sleep_wakeup 来实现。SDK 根据 Link Layer 的状态和低功耗模式，在 BLE 协议栈部分加了一套低功耗管理机制（对应的代码在 blc_sdk_main_loop 中实现）。user 只需要调用相应的 API 即可对低功耗进行配置。

对于 deepsleep，BLE 低功耗管理不包括对它的处理，所以 user 需要手动在应用层调用 API cpu_sleep_wakeup 来进入 deepsleep。手动 deepsleep mode 的使用，可以参考 SDK project "8208_ble_sample" blt_pm_proc 函数中对 deepsleep 的处理。

下面开始对 Advertising state 和 Connection state Slave role 的低功耗管理做详细的介绍。

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。

BLE SDK 中定义了以下结构体和变量，下面只列出结构体中部分变量（API 介绍时需要用到的变量）。

```
typedef struct {
    u8      suspend_mask;
    u8      wakeup_src;
    u16     user_latency;
}st_ll_pm_t;

extern  u32     deepRet_advThresTick;
extern  u32     deepRet_connThresTick;
extern  u32     deepRet_earlyWakeupTick;
```

在文件 ll_pm.c 中定义了如下结构体变量。

```
st_ll_pm_t  bltPm;
```

注意：

> 该文件被封装在 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 "bltPm. suspend_mask" 的变量。

4.2.4 API bls_pm_setSuspendMask

用于配置 Link Layer Advertising state 和 Conn state Slave role 低功耗管理的 API：

```
void    bls_pm_setSuspendMask (u8 mask);
u8      bls_pm_getSuspendMask (void);
```

使用 bls_pm_setSuspendMask 设置 bltPm.suspend_mask（默认值为 SUSPEND_DISABLE）。

这两个 API 的源码为：

```
void  bls_pm_setSuspendMask (u8 mask)
{
        bltPm.suspend_mask = mask;
}
u8  bls_pm_getSuspendMask (void)
{
        return bltPm.suspend_mask;
}
```

bltPm.suspend_mask 的设置，可以选择下面几个值中的一个，或者选择多个值的"或操作"。

```
#define         SUSPEND_DISABLE             0
#define         SUSPEND_ADV                 BIT(0)
#define         SUSPEND_CONN                BIT(1)
#define         DEEPSLEEP_RETENTION_ADV     BIT(2)
#define         DEEPSLEEP_RETENTION_CONN    BIT(3)
```

SUSPEND_DISABLE 表示 sleep disable，不允许 MCU 进入 suspend 和 deepsleep retention。

SUSPEND_ADV 和 DEEPSLEEP_RETENTION_ADV 分别用于控制 Advertising state 时 MCU 进入 suspend 和 deepsleep retention。

SUSPEND_CONN 和 DEEPSLEEP_RETENTION_CONN 分别用于控制 Conn state Slave role 时 MCU 进入 suspend 和 deepsleep retention。

SDK 低功耗 sleep mode 的设计上，deepsleep retention 是 suspend 的替代模式，目的是降低 sleep mode 的功耗。

以 Conn state slave role 为例，SDK 首先得看到 bltPm.suspend_mask 中 SUSPEND_CONN 是否生效，才可以进入 suspend。在可以进入 suspend 的基础上，根据实际情况再结合 bltPm.suspend_mask 中 DEEPSLEEP_RETENTION_CONN 是否生效，才能决定此时 suspend mode 是否被切换为 deepsleep retention mode。

所以如果 user 希望 MCU 进入 suspend，打开 SUSPEND_ADV/SUSPEND_CONN 即可；如果希望 MCU 进入 deepsleep retention mode，必须同时打开 SUSPEND_CONN 和 DEEPSLEEP_RETENTION_CONN。

该 API 最常用的 3 种情况如下：

```
bls_pm_setSuspendMask(SUSPEND_DISABLE);
```

MCU 不允许进入 sleep mode。

```
bls_pm_setSuspendMask(SUSPEND_ADV | SUSPEND_CONN);
```

MCU 在 Advertising state 和 Conn state Slave role 只允许进入 suspend，但是不允许进入 deepsleep retention。

```
bls_pm_setSuspendMask(SUSPEND_ADV | DEEPSLEEP_RETENTION_ADV
                    |SUSPEND_CONN | DEEPSLEEP_RETENTION_CONN);
```

MCU 在 Advertising state 和 Conn state Slave role 允许进入 suspend 和 deepsleep retention，具体进入哪种 sleep mode 由当前 sleep 的时间长度决定，后面会详细介绍。

除了上面 3 种常用的情况，也可以出现一些特殊的用法，如：

```
bls_pm_setSuspendMask(SUSPEND_ADV)
```

只有 Advertising state 可以进入 suspend，Conn state Slave role 不允许进入 sleep mode。

```
bls_pm_setSuspendMask(SUSPEND_CONN | DEEPSLEEP_RETENTION_CONN)
```

只有 Conn state Slave role 可以进入 suspend 或 deepsleep retention，Advertising state 不允许进入 sleep mode。

4.2.5 API bls_pm_setWakeupSource

user 通过上面的 bls_pm_setSuspendMask 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        bls_pm_setWakeupSource(u8 source);
```

source 可以选择唤醒源 PM_WAKEUP_PAD。

该 API 设置底层变量 bltPm.wakeup_src，SDK 中源码为：

```
void    bls_pm_setWakeupSource (u8 src)
{
    bltPm.wakeup_src = src;
}
```

MCU 在 Advertising state 或 Conn state Slave role 进入 sleep mode，实际的唤醒源为：

```
bltPm.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 Event 或 Brx Event。

每次调用 bls_pm_setWakeupSource 设置唤醒源后，一旦 MCU 进入 sleep mode 被唤醒后，bltPm.wakeup_src 会被清 0。

4.2.6 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。

4.2.6.1 blc_sdk_main_loop

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
////////////////////// user PM config ////////////////////////
    //blt_pm_proc();
}
```

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。

```
int blc_sdk_main_loop (void)
{
    ......
    if(bltPm. suspend_mask == SUSPEND_DISABLE) // SUSPEND_DISABLE, can not
    {                                          // enter sleep mode
        return  0;
    }

    if( (Link Layer State == Advertising state)  ||  (Link Layer State == Conn state Slave role)
     ↪ )
    {
        if(Link Layer is in Adv Event or Brx Event) //RF is working, can not enter
        {                                  //sleep mode
```

```
            return  0;
        }
        else
        {
            blt_brx_sleep (); //process sleep & wakeup
        }
    }
    return 0;
}
```

(1) 当 bltPm. suspend_mask 为 SUSPEND_DISABLE 时，直接退出，不会执行 blt_brx_sleep 函数。所以 user 使用 bls_pm_setSuspendMask(SUSPEND_DISABLE) 时，低功耗管理的逻辑就会完全失效，MCU 不会进入低功耗，while(1) 的 loop 一直在执行。

(2) 如果 Advertising State 的 Adv Event 或 Conn state Slave role 的 Brx Event 正在执行，blt_brx_sleep 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/Brx Event 结束之后才能进 sleep mode。

当以上两个条件都不满足时，才去执行 blt_brx_sleep 函数。

4.2.6.2 blt_brx_sleep

blt_brx_sleep 函数的逻辑实现如下所示。

```
void    blt_brx_sleep (void)
{
    if( (Link Layer state == Adv state)&& (bltPm. suspend_mask &SUSPEND_ADV) )
    {   //当前广播状态，允许进 suspend
        T_wakeup = T_advertising + advInterval;
         “BLT_EV_FLAG_SUSPEND_ENTER”  event callback execution
        T_sleep = T_wakeup - clock_time();
        if( bltPm. suspend_mask & DEEPSLEEP_RETENTION_ADV && T_sleep >
        ↪ bltPm.deepRet_advThresTick )
        {   //suspend is automatically switched to deepsleep retention
            cpu_sleep_wakeup (DEEPSLEEP_MODE_RET_SRAM_LOW16K, PM_WAKEUP_TIMER |
↪ bltPm.wakeup_src,T_wakeup); //suspend
            //MCU 被唤醒后 PC 值 reset to 0, 将重新执行 software bootloader //
             ↪ (cstartup_8258_16K.S)、system initialization 等
        }
        else
        {
            cpu_sleep_wakeup (SUSPEND_MODE, PM_WAKEUP_TIMER | bltPm.wakeup_src, T_wakeup);
        }
         “BLT_EV_FLAG_SUSPEND_EXIT”  event callback execution

        if(suspend 是被 GPIO PAD 唤醒)
        {
```

```
            “BLT_EV_FLAG_GPIO_EARLY_WAKEUP” event callback execution
        }
    }
    else if((Link Layer state == Conn state Slave role)&& (SuspendMask&SUSPEND_CONN) )
    {   //当前 Conn state，进 suspend
        if(conn_latency != 0)
        {
            latency_use = bls_calculateLatency();
            T_wakeup = T_brx + (latency_use +1) * conn_interval;
        }
        else
        {
            T_wakeup = T_brx + conn_interval;
        }
         “BLT_EV_FLAG_SUSPEND_ENTER” event callback execution
        T_sleep = T_wakeup - clock_time();
        if(  bltPm. suspend_mask & DEEPSLEEP_RETENTION_CONN &&
T_sleep  >  bltPm.deepRet_connThresTick )
        {   //suspend is automatically switched to deepsleep retention
            cpu_sleep_wakeup (DEEPSLEEP_MODE_RET_SRAM_LOW16K, PM_WAKEUP_TIMER |
↪  bltPm.wakeup_src,T_wakeup); //suspend
            //MCU 被唤醒后 PC 值 reset to 0, 将重新执行 software bootloader//
             ↪  (cstartup_8258_16K.S)、system initialization 等
        }
        else
        {
            cpu_sleep_wakeup (SUSPEND_MODE, PM_WAKEUP_TIMER | bltPm.wakeup_src, T_wakeup);
        }

         “BLT_EV_FLAG_SUSPEND_EXIT” event callback execution
        if(suspend 是被 GPIO PAD 唤醒)
        {
             “BLT_EV_FLAG_GPIO_EARLY_WAKEUP” event callback execution
            调整 BLE 时序相关的处理
        }
    }
    bltPm.wakeup_src = 0;
    bltPm.user_latency = 0xFFFF;
}
```

上面 blt_brx_sleep 函数的流程看起来比较复杂，我们的分析从最简单的情况开始：首先 conn_latency 方面只考虑 conn_latency =0 的情况。其次暂时不考虑 deepsleep retention 生效的情况，此时只有 suspend。当应用层设置 suspend mask 时使用的是 bls_pm_setSuspendMask(SUSPEND_ADV | SUSPEND_CONN) 时，就对应这种情况。

结合文档前面介绍的 controller event，这里看到几个 suspend 相关 event 回调函数的执行的时机：BLT_EV_FLAG_SUSPEND_ENTER、BLT_EV_FLAG_SUSPEND_EXIT、BLT_EV_FLAG_GPIO_EARLY_WAKEUP。

Link Layer Advertising state 时 bltPm. suspend_mask 中 SUSPEND_ADV 生效，或者 Link Layer Conn state slave role 时 bltPm. suspend_mask 中 SUSPEND_CONN 生效，可以进入 suspend。

在 suspend 模式下，最终调用了 driver 中的 API cpu_sleep_wakeup：

```
cpu_sleep_wakeup (SUSPEND_MODE, PM_WAKEUP_TIMER | bltPm.wakeup_src, T_wakeup);
```

唤醒源为 PM_WAKEUP_TIMER | bltPm.wakeup_src，Timer 无条件生效，是为了保证 MCU 一定要在下一个 Adv Event、Brx Event 到来前唤醒。唤醒时间 T_wakeup 请参考本文档前面的图"sleep timing for Advertising state & Conn state Slave role"。

blt_brx_sleep 函数退出时将 bltPm.wakeup_src 和 bltPm.user_latency 的值复位，所以需要注意 API bls_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。同理 API bls_pm_setManualLatency 也一样。

4.2.7 deepsleep retention 的详细分析

引入 deepsleep retention，对上面的软件处理流程继续分析。

当应用层按如下设置时，deepsleep retention mode 被打开。

```
bls_pm_setSuspendMask( SUSPEND_ADV | DEEPSLEEP_RETENTION_ADV | SUSPEND_CONN |
↪  DEEPSLEEP_RETENTION_CONN);
```

4.2.7.1 API blc_pm_setDeepsleepRetentionThreshold

Advertising state/Conn state slave role 中，满足以下条件，suspend 才会被自动切换为 deep retention：

```
if(  bltPm. suspend_mask & DEEPSLEEP_RETENTION_ADV &&T_sleep  > bltPm.deepRet_advThresTick )
if(  bltPm. suspend_mask & DEEPSLEEP_RETENTION_CONN &&T_sleep  >  bltPm.deepRet_connThresTick )
```

第一个条件 bltPm. suspend_mask 中 DEEPSLEEP_RETENTION_ADV 需要生效，前面已经介绍过。

第二个条件 T_sleep > bltPm.deepRet_advThresTick 或 T_sleep > bltPm.deepRet_connThresTick 中，T_sleep = T_wakeup – clock_time() 表示唤醒时间减去实时时间，即 sleep 的持续时间。这个条件的意义是：当 sleep 的持续时间超过特定的时间阈值时，MCU 的 sleep mode 才会被切换为 deep retention。

我们先介绍两个时间阈值设置的 API，如下所示为源码，设置时间单位为 ms。

```
void blc_pm_setDeepsleepRetentionThreshold( u32 adv_thres_ms,
u32 conn_thres_ms)
{
 bltPm.deepRet_advThresTick = adv_thres_ms * CLOCK_16M_SYS_TIMER_CLK_1MS;
 bltPm.deepRet_connThresTick = conn_thres_ms * CLOCK_16M_SYS_TIMER_CLK_1MS;
}
```

API blc_pm_setDeepsleepRetentionThreshold 用于设置 suspend 切换到 deepsleep retention 触发条件中的时间阈值，这个设计是为了追求更低的功耗。

参考前文"sleep wake_up 后运行流程"部分的说明可知，suspend mode wake_up 后，可以立刻回到 suspend 前的环境继续运行。上面软件流程中在 T_wakeup 醒来之后，可以立刻开始执行 Adv Event/Brx Event 任务。

而 deepsleep retention wake_up 后需要回到"Run software bootloader"开始的地方，和 suspend wake_up 相比，需要多运行 3 个步骤（Run software bootloader + System initialization + User initialization），才能再次回到 main_loop 中执行 Adv Event/Brx Event 任务。

以 Conn state slave role 为例，下图表示 sleep mode 分别为 suspend 和 deepsleepretention 时的 timing（时序）& power（功耗）对比。

两个相邻的 Brx event 之间的时间差值 T_cycle 即当前时间周期。将 Brx Event 的功耗平均化，等效电流为 I_brx，持续时间为 t_brx（这里取名 t_brx 是为了和前面已有概念 T_brx 做区分）。Suspend 的底电流为 I_suspend，deep retention 的底电流为 I_deepRet。

"Run software bootloader + System initialization + User initialization"的过程平均电流等效为 I_init，持续的总时间为 T_init。实际的应用中，T_init 的值需要 user 去把控和测量，后面会介绍如何实现。

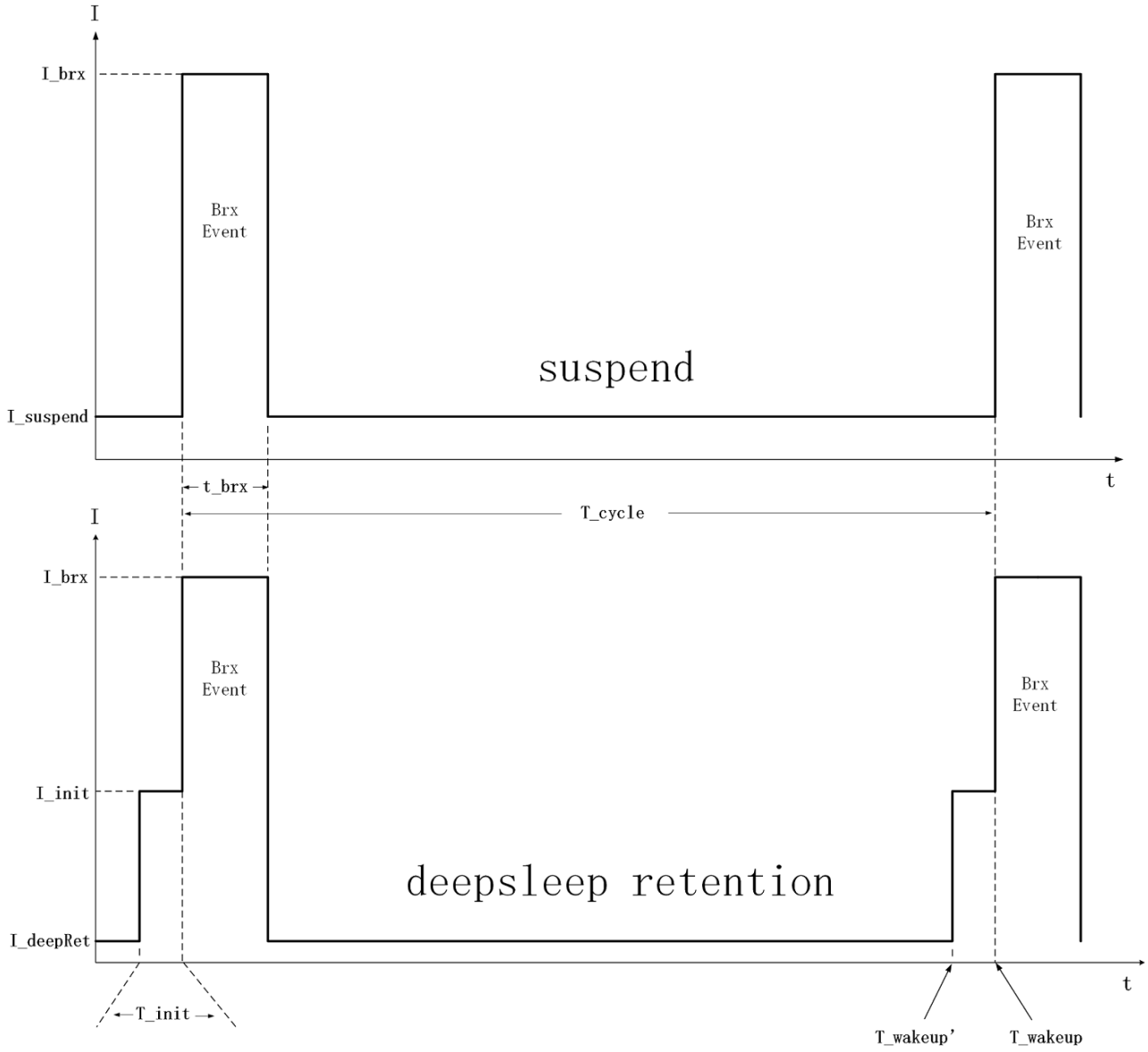


Figure 4.4: Suspend Deep sleep Retention Timing Power

以下是图中几个名词说明。

- T_cycle：两个相邻的 Brx event 之间的时间差值
- I_brx：将 Brx Event 的功耗平均化，等效电流为 I_brx
- t_brx：I_brx 持续时间
- I_suspend：suspend 底电流
- I_deepRet：deep retention 的底电流
- I_init：Software bootloader + System initialization + User initialization 过程的等效平均电流
- T_init：I_init 持续的总时间

那么，使用了 suspend mode 的 Brx 平均功耗为：

I_avgSuspend = I_brx*t_brx + I_suspend* (T_cycle – t_brx)

由于 T_cycle 远大于 t_brx，(T_cycle – t_brx) 约等于 T_cycle。

I_avgSuspend = I_brx*t_brx + I_suspend* T_cycle

使用了 deepsleep retention mode 的 Brx 平均功耗为

I_avgDeepRet = I_brx*t_brx + I_init*T_init + I_deepRet* (T_cycle – t_brx)

= I_brx*t_brx + I_init*T_init + I_ deepRet * T_cycle

比较 I_avgSuspend 和 I_avgDeepRet，去掉相同的 " I_brx*t_brx"，最终比较的部分为：

I_avgSuspend – I_avgDeepRet = I_suspend* T_cycle – I_init*T_init – I_ deepRet * T_cycle

= T_cycle((I_suspend – I_ deepRet) – (T_init*I_init)/T_cycle)

对于功耗调试正确的应用程序 (硬件电路和软件的功耗调试都正确)，最终 (I_suspend - I_ deepRet) 是一个固定的值，比如 suspend 为 30uA，deepsleep retention 为 2uA 时，(I_suspend - I_ deepRet) = 28uA；(T_init*I_init) 最终也是固定的值，比如 I_init 为 3mA，T_init 为 400 us，(T_init*I_init) 为 1200 uA*us

I_avgSuspend – I_avgDeepRet = Tcycle (28 – 1200/Tcycle)

可以看到，当 T_cycle 的值较小时（例子中 T_cycle < 43 ms)，使用 suspend mode 最终的平均功耗更低；当 T_cycle 较大时（T_cycle > 43 ms)，使用 deepsleep retention mode 最终的平均功耗会更低。

注意：

> PM 软件处理流程部分可以看到，当 T_sleep > 43ms 的时候才会将 suspend 自动切换为 deepsleep retention。一般我们认为 MCU working 时间（Brx Event + UI task）比较短，当 T_cycle 较大时，可以认为 T_sleep 约等于 T_cycle。

那么初始化的时候按照如下设置的话，MCU 对于 T_sleep 大于 43ms 的 suspend 自动切换为 deepsleep retention，对于 T_sleep 小于 43ms 的 suspend 还是保持不变。

```
blc_pm_setDeepsleepRetentionThreshold(43, 43);
```

以一个 10ms connection interval * (99 + 1) = 1s 的长连接为例进行说明：

在 Conn state slave role 时，由于应用层的任务、手动 latency 的设置等，会导致 MCU suspend 时可能出现 10ms、20ms、50ms、100ms、1s 等时间值。根据 43ms 的阈值设置，MCU 会自动将 50ms、100ms、1s 等 suspend 切换为 deepsleep retention，而 10ms、20ms 等 suspend 还是维持 suspend，这样的处理可以保证一个最优的功耗。

由于 deepsleep retention 的功耗比 suspend 低，并且"Run software bootloader + System initialization + User initialization" 3 个步骤的存在会多出一部分的功耗，根据以上分析，一定是 T_cycle 大于某个临界值之后，使用 deepsleep retention 才会更省功耗。上面例子中的数值只是简单的 demo，user 在实现功耗优化时需要根据一定的方法先测出上面公式中对应的数值，最终才可以确定临界值。

只要程序设计上参考 SDK 中的 demo，且在 user initialization 这部分没有错误的增加很大的时间消耗，上面的 T_cycle 临界值都不会太大。一般超过 100ms 以上的 T_cycle，使用 deepsleep retention mode 的功耗都会更低。

4.2.7.2 blc_pm_setDeepsleepRetentionEarlyWakeupTiming

参考上面的图"suspend & deepsleep retention timing & power"，可以看到，suspend wake_up 的时间点 T_wakeup 正好是下一个 Brx Event 开始的时间点，对应 BLE master 端开始发包的时间。

deepsleep retention wake_up 的时间点如果也设置为 T_wakeup 的话，就会有问题：此时 MCU 醒来，还需要经过 T_init 的时间（Run software bootloader + System initialization + User initialization 3 个步骤消耗的时间）才能开始 Brx Event，已经错过了 BLE master 端发包的时间点。

为了解决这个问题，MCU wake_up 的时间点需要提前到 T_wakeup'。

T_wakeup' = T_wakeup – T_init

以 Conn state slave role 为例（Advertising state 的处理完全相同），上面 PM 软件处理流程对 blt_brx_sleep 函数中 deepsleep retention wake_up 的时间点进行优化改善，如下：

```
cpu_sleep_wakeup (DEEPSLEEP_MODE_RET_SRAM_LOW32K, PM_WAKEUP_TIMER | bltPm.wakeup_src,
T_wakeup - bltPm.deepRet_earlyWakeupTick);
```

T_wakeup 是 BLE stack 自动计算的时间点，user 只需要知道 T_init 的值，就可以通过下面 API 来设置 deepsleep retention wake_up 的提前量。

```
void blc_pm_setDeepsleepRetentionEarlyWakeupTiming(u32 earlyWakeup_us)
{
    bltPm.deepRet_earlyWakeupTick = earlyWakeup_us *    CLOCK_16M_SYS_TIMER_CLK_1US;
}
```

User 将测量到的 T_init 的值直接设置到上面 API 即可，或者设置的值比 T_init 略大一点，但不能小于这个值。

4.2.7.3 T_init 的优化和测量

这部分的介绍大量用到 ram_code、retention_data、deepsleep retention area 等 Sram 相关的概念，请 user 先参考 Sram 空间分配的详细介绍。

(1) T_init timing

由图"suspend & deepsleep retention timing & power"，结合上面已有的分析可知，对于 T_cycle 较大的情况，sleep mode 使用 deepsleep retention 功耗更低，但这种模式下 T_init 的时间是必须的，无法避开。为了尽量降低长时间睡眠的功耗，需要将 T_init 的时间尽量优化到最小。I_init 的值基本上是保持稳定的，不需要去做优化。

T_init 是 Run software bootloader + System initialization + User initialization 3 个步骤消耗的时间总和，将这 3 个步骤拆开分析，先定义各步骤的时间。

- T_cstatup 为 Run software bootloader 的时间（Run software bootloader 就是执行汇编文件 cstartup_xxx.S）。
- T_sysInit 为执行 system initialization 的时间。
- T_userInit 为执行 user initialization 的时间。

T_init = T_cstatup + T_sysInit + T_userInit

下图为 T_init 的时序图。

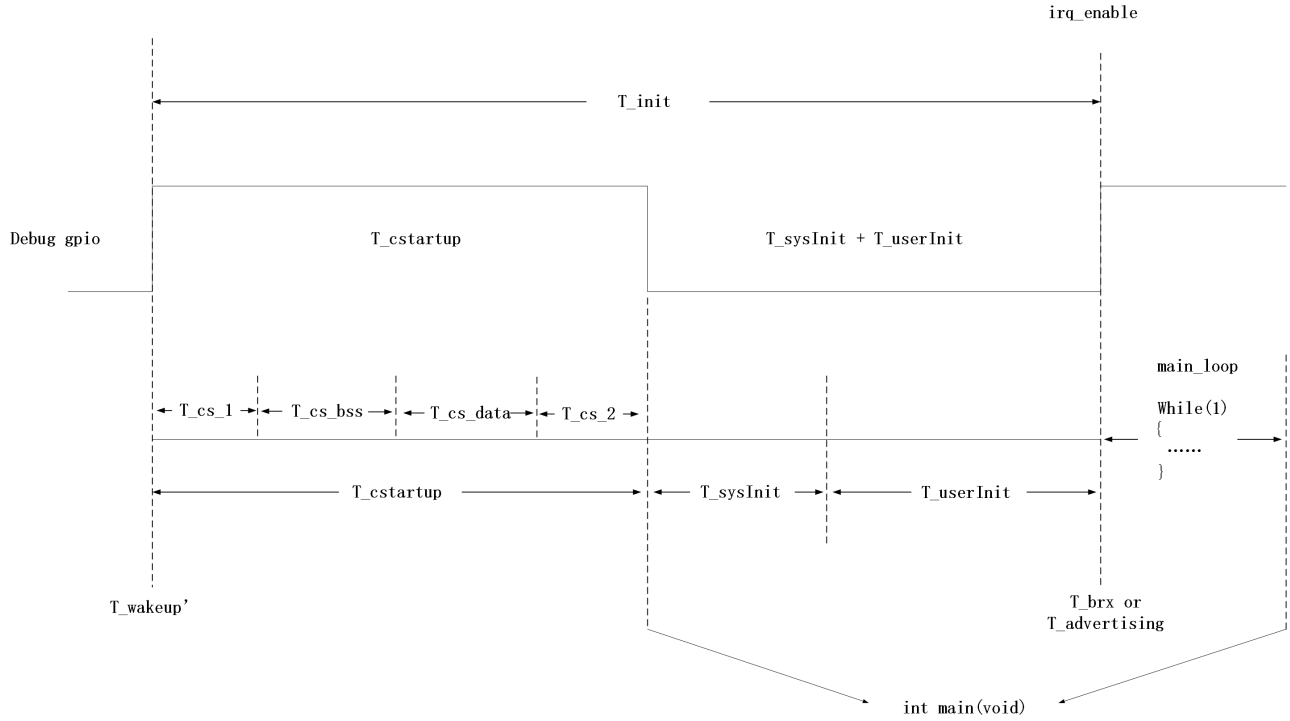


Figure 4.5: T_init Timing

结合前面已有的图和概念继续分析。

T_wakeup 是下一个 Adv Event/Brx Event 开始的时间点，T_wakeup' 是 MCU 提前 wake_up 的时间。

MCU wake_up 后，执行 cstatup_xxx.S，然后跳转到 main 函数，执行 System initialization + User initialization，之后进入 main_loop。一旦进入 main_loop 便可以处理 Adv Event/Brx Event，所以 T_userInit 结束的时间对应 Adv Event/Brx Event 开始的时间 T_brx/T_advertising。图中 irq_enable 这个函数是 T_userInit 和 main_loop 分界线，和 SDK 上 code 是对应的。

SDK 上，T_sysInit 时间包括 sys_init、rf_drv_init、gpio_init、clock_init 执行的时间，这些时间 SDK 已经做了优化，运行时间达到最低。所以 T_sysInit 是一个固定的值，user 不用关注这个时间。SDK 优化这部分时间的方法是将这些初始化函数全部放到 ram_code 中运行。

下面只对 SDK 中 T_cstatup 和 T_userInit 进行详细的介绍。

(2) T_userInit

由前面的介绍可知，user initialization 在 power on、deepsleep wake_up、deepsleep retention wake_up 的时候都需要被执行。

对于没有使用 deepsleep retention mode 的应用来说，user initialization 不需要区分 deepsleep retention wake_up 和 power on/deepsleep wake_up。BLE SDK 中，所有的 user initialization 都是用下面函数即可完成。

```
void user_init(void);
```

对于使用了 deepsleep retention mode 的应用，为了尽量降低功耗，T_userInit 需要最优化。所以需要做区分，deepsleep retention wake_up 时，user initialization 需要尽量快。

user_init 函数中所有的 initialization 可以被分为两类：一是硬件寄存器的初始化，二是 Sram 上逻辑变量的初始化。

根据 deepsleep retention mode 可以保持 Sram 前 16K（32K）不掉电的特性，我们可以将逻辑变量定义成 retention_data，这样的话 deepsleep retention wake_up 时就可以省掉逻辑变量初始化的时间。由于寄存器的状态无法被保持，deepsleep retention wake_up 时寄存器的初始化必须重新执行。

最终的实现方法是 deepsleep retention wake_up 时，执行优化过的 user_init_deepRetn 函数，power on 和 deepsleep wake_up 时执行 user_init_normal。如下 code 所示：

```
int deepRetWakeUp = pm_is_MCU_deepRetentionWakeup();
    if( deepRetWakeUp ){
        user_init_deepRetn ();
    }
    else{
        user_init_normal ();
    }
```

user 可以对比一下这两个函数的实现，下面是 SDK demo "8208_ble_sample" user_init_deepRetn 函数的实现。

```
_attribute_ram_code_ void user_init_deepRetn(void)
{
    blc_ll_initBasicMCU();
    rf_set_power_level_index (MY_RF_POWER_INDEX);
    blc_ll_recoverDeepRetention();
    irq_enable();
}
```

前 2 句 code 是 BLE 初始化中必不可少的相关硬件寄存器的初始化;

blc_ll_recoverDeepRetention 是对 Link Layer 相关软硬件状态的恢复，由 stack 底层处理。

最后 irq_enable 是初始化完成，打开系统中断。

以上几个初始化都属于固定写法，user 不要去修改。

在 SDK demo 基础上，user initialization 如果增加了其他功能，这些新增功能的 initialization 节省时间的原则是：对每一条 initialization 的 code 进行分析，判断出是确定需要 retention 回来重新初始化，还是硬件寄存器的操作。

- 如果是 retention 回来需要重新初始化的全局变量，将相应的变量添加关键字"attribute_data_reload"定义到 "data_reload"段，就可以保证 deepsleep retention wake_up 后重新初始化，不需要该操作放到 user_init_deepRetn 函数中。

- 如果是硬件寄存器的操作，那么必须放到 user_init_deepRetn 函数中，确保硬件状态的正确性。

deepsleep retention wake_up 后的 T_userInit 就是 user_init_deepRetn 函数的执行时间，SDK 中也尽量将这部分的函数放到 ram_code 中以节省运行时间。

在 deepsleep retention area 空间足够的前提下，user 也需要将增加的硬件初始化相关函数放到 ram_code 中。在 B80 BLE SDK 中我们提供了以下 API 用于将 deepsleep retention wake_up 后需要初始化的寄存器操作放到 ram 中，节省唤醒时间和平均功耗，当然刚 API 也会导致 ram 占用增加，user 需要自行评估是否需要调用该 API。

```
void blc_ll_initDeepsleepRetention_module(void)
```

(3) T_cstartup

T_cstartup 是执行 cstartup_xxx.S（比如 cstartup_8258_RET_16K.S）所消耗的时间，请 user 参考 SDK 中 boot 文件。

T_cstartup 按照时间顺序可以被拆成 4 个时间组成：

T_cstartup = T_cs_1 + T_cs_data_reload + T_cs_2

T_cs_1 和 T_cs_2 两个时间是固定的，user 无法修改它们，不需要去关注。

T_cs_data_reload 是 Sram 中"data_reload"段的初始化时间。"data_reload"段是 retention 醒来重新初始化的全局变量，它们的初始值存储在 flash 的"data_reload init value"区域上。"data_reload"段初始化的时间就是 MCU 从 flash "data_reload init value"区域上的初值拷贝到 Sram "data_reload"段的过程。对应的汇编 code 如下：

```
COPY_DATA_RELOAD:
    tloadr      r1, DATA_I+12
    tloadr      r2, DATA_I+16
    tloadr      r3, DATA_I+20
COPY_DATA_RELOAD_BEGIN:
    tcmp        r2, r3
    tjge        COPY_DATA_RELOAD_END
    tloadr      r0, [r1, #0]
    tstorer     r0, [r2, #0]
    tadd        r1, #4
    tadd        r2, #4
    tj          COPY_DATA_RELOAD_BEGIN
```

因为 flash 数据拷贝速度相对比较慢（给个参考：16 byte 数据大概需要 7us 时间），如果"data_reload"段的数据较多，就会造成 T_cs_data_reload 时间偏大，最终导致 T_init 偏大。

SDK 中"data_reload"段数据越少越好。user 可以参考文档前面介绍的方法去查看 list 文件中"data_reload" 段的大小。

(4) T_init 测量

根据以上介绍，对 T_cstartup 和 T_userInit 的时间做优化，将 T_init 优化到最小时间，然后需要测量出 T_init，填到 API blc_pm_setDeepsleepRetentionEarlyWakeupTiming。

T_init 的起点即 T_cstartup 的起点。T_cstartup 的起点是 cstartup_8258_RET_16K.S 文件中 “__reset” 这个点，如下 code 所示。

```
__reset:

#if 0
    @ add debug, PB4 output 1
    tloadr      r1, DEBUG_GPIO    @0x80058a  PB oen
    tmov        r0, #139      @0b 11101111
    tstorerb    r0, [r1, #0]

    tmov        r0, #16         @0b 00010000
    tstorerb    r0, [r1, #1]    @0x800583  PB output
#endif
```

结合图 “ T_init timing” 中 Debug gpio 的示意，在 “__reset” 这里放了 Debug GPIO PB4 输出高的操作，user 只要将“#if 0”改成“#if 1”便可打开 PB4 输出高的操作。

T_cstartup 结束时间是下面“tjl main”的时间点。

```
tjl main
END:    tj  END
```

那么 main 函数开始的时间与 T_cstartup 结束的时间几乎是相等的。在 main 函数一开始的地方让 PB4 输出低，如下所示。注意这个 DBG_CHN4_LOW 依赖于 app_config.h 中 “ DEBUG_GPIO_ENABLE” 的打开。

```
_attribute_ram_code_ int main (void)    //must run in ramcode
{
    DBG_CHN4_LOW;   //debug
    ......
}
```

PB4 的一高一低可以测量出 T_cstartup 的持续时间。

在 user_init_deepRetn 函数里面 T_userInit 结束地方添加 PB4 输出高的操作，这样就可以实现图中 Debug gpio 的效果。User 可以通过示波器、逻辑分析仪等设备测量出 T_init、T_cstartup 的时间。User 在理解 GPIO 操作的基础上，可根据自己的需要，对 Debug gpio 的 code 进行修改，以得到更多时间参数测量的结果，如 T_sysInit、T_userInit 等。

4.2.8 Connection Latency

4.2.8.1 connection Latency 生效时的 Sleep 时序

前面关于 Conn state slave role 的 sleep mode 的介绍（参考图“sleep timing for Advertising state & Conn state Slave role”所示），都是基于 connection latency（简称 conn_latency）没有生效时的前提。

PM 软件处理流程上，T_wakeup = T_brx + conn_interval，对应的 code 如下。

```
if(conn_latency != 0)
    {
        latency_use = bls_calculateLatency();
        T_wakeup = T_brx + (latency_use +1) * conn_interval;
    }
    else
    {
        T_wakeup = T_brx + conn_interval;
    }
```

当 BLE slave 经过 connection parameters update (连接参数更新) 流程，conn_latency 生效后，sleep wake_up 的时间为

T_wakeup = T_brx + (latency_use +1) * conn_interval;

下图所示为一个 conn_latency 生效时的 sleep 时序，此时 latency_use= 2。

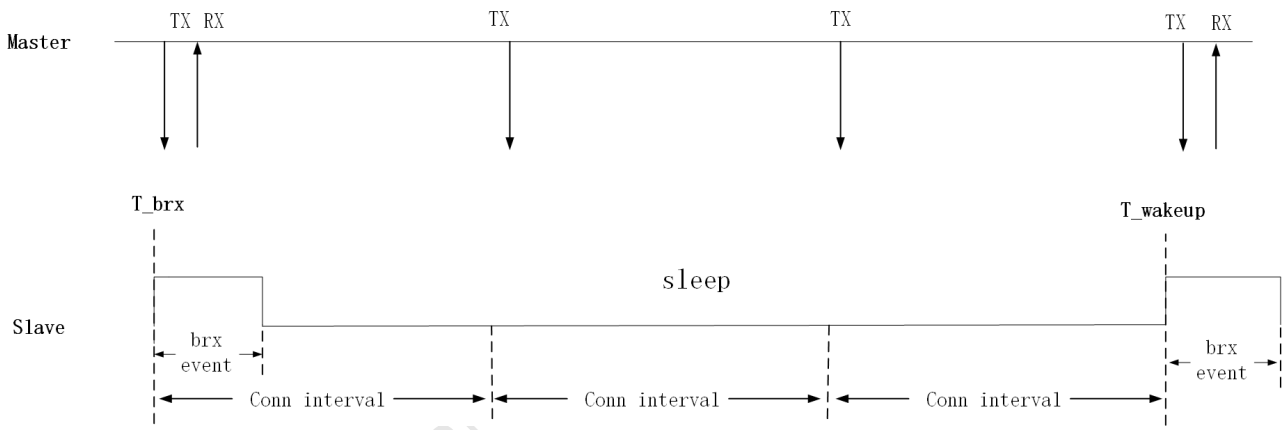


Figure 4.6: Sleep Timing for Valid Conn_latency

conn_latency 没有生效时，sleep 的时间最长不超过 1 个 connection interval (一般都比较小)。由于 conn_latency 的生效，sleep 的时间可能会出现一个比较大的值，如 1s、2s 等，系统功耗可以变得非常低。长 sleep 期间使用功耗更小的 deepsleep retention mode 才变得有意义。

4.2.8.2 latency_use 的计算

当 conn_latency 生效时，T_wakeup 的值是由 latency_use 决定的。说明 latency_use 并不是直接等于 conn_latency。

在 latency_use 计算中，涉及到一个 user_latency，这个是 user 可以设置的值，调用的 API 及其源码为：

```
void bls_pm_setManualLatency(u16 latency)
{
    bltPm.user_latency = latency;
}
```

bltPm.user_latency 这个变量的初值为 0xFFFF。注意 PM 软件处理流程中 blt_brx_sleep 函数最后会强制将它再次复位为 0xFFFF，说明 API bls_pm_setManualLatency 设置的 user_latency 只对最近一次 sleep 管用，每次不同的 sleep 都需要重新设置。

latency_use 的计算过程如下。

首先计算 system latency：

(1) 若当前连接参数中 connection latency 为 0，system latency 为 0。

(2) 若当前连接参数中 connection latency 非 0：

- 若当前系统还有一些任务没有处理完，必须在下一个 connection interval 醒来收发包继续处理（比如还有数据没有发送完、收到 master 的数据还没处理完等等），system latency 为 0。
- 若当前系统已经没有任务需要处理了，则 system latency 等于 connection latency。但是有一个例外，如果收到了 master 的 update map request 或 update connection parameter request 且实际的更新时间点在 (connection latency+1) 个 interval 之前，则实际的 system latency 会强制 MCU 在实际更新时间点之前那个 interval 醒来，确保 BLE 时序的正确。

然后

latency_use = min(system latency, user_latency)

即 latency_use 取 system latency 和 user_latency 中的较小值。

以上逻辑可以看出：如果 user 调用 API bls_pm_setManualLatency 设置的 user_latency 比 system latency 小，user_latency 将会作为最终的 latency_use，否则 system latency 将作为最终的 latency_use。

4.2.9 API bls_pm_getSystemWakeupTick

下面的 API 用于获取低功耗管理计算的 suspend 醒来的时间点（System Timer tick），即 T_wakeup。

```
u32 bls_pm_getSystemWakeupTick(void);
```

从 PM 软件处理流程 blt_brx_sleep 函数中可以看到，T_wakeup 的计算比较晚，已经很接近 cpu_sleep_wakeup 函数了，应用层只能在 BLT_EV_FLAG_SUSPEND_ENTER 事件回调函数里才能得到准确的 T_wakeup。

下面以按键扫描的应用为例，说明 BLT_EV_FLAG_SUSPEND_ENTER 事件回调函数和 bls_pm_getSystemWakeupTick 的用法。

```
bls_app_registerEventCallback (BLT_EV_FLAG_SUSPEND_ENTER, &app_set_kb_wakeup);

void app_set_kb_wakeup(u8 e, u8 *p, int n)
{
    #if (BLE_APP_PM_ENABLE)
    if( blc_ll_getCurrentState() == BLS_LINK_STATE_CONN
        && ((u32)(bls_pm_getSystemWakeupTick() - clock_time())) > 80
↪   *CLOCK_16M_SYS_TIMER_CLK_1MS ){  //suspend time > 30ms.add gpio wakeup
        bls_pm_setWakeupSource(PM_WAKEUP_PAD);  //gpio CORE wakeup suspend
    }
    #endif
}
```

以上这个回调函数要实现的作用是防止按键丢失。

一个正常的人为机械按键动作大概会持续几百毫秒，按的快的时候也会有一两百毫秒。当 user 通过 bls_pm_setSuspendMask 设置了 Advertising state 和 Conn state 都要进入 sleep mode，在 conn_latency 没有生效的前提下，只要 Adv interval 和 conn_interval 的值不是特别大（一般设置在 100ms 以内），sleep 的时间不会超过 Adv interval 和 conn_interval，能够确保按键扫描的频率，就不会丢键。此时不设置 GPIO 唤醒，不让按键动作唤醒 MCU。

但是当 conn_latency 生效后（比如 conn_interval 为 10ms，conn_latency 为 99），可能某次 Conn state 时的 sleep 会持续 1s。这个过程中按键可能会丢掉。在 BLT_EV_FLAG_SUSPEND_ENTER 回调里判断如果当前状态为 Conn state，并且当前要进入的 suspend 的唤醒时间点距离当前时间大于 80 ms，那么将 GPIO PAD 的唤醒添加进去。如果 timer 的唤醒时间点还没到，有按键按下导致 GPIO 上电平发生变化，触发 MCU 提前唤醒，去处理按键的扫描任务，按键就不会丢失。

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 8208 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_wakeup_sleep 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_wakeup_sleep 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_wakeup_sleep 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_wakeup_sleep，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。

针对以上问题，在 SDK demo "8208_ble_sample"中，做了相应的处理。

在 BLT_EV_FLAG_SUSPEND_ENTER 中设置了只有当 suspend 时间超过某个时间时，才会开启 GPIO PAD 唤醒。

```
void app_set_kb_wakeup(u8 e, u8 *p, int n)
{
    #if (BLE_APP_PM_ENABLE)
```

```
    if( blc_ll_getCurrentState() == BLS_LINK_STATE_CONN
        && ((u32)(bls_pm_getSystemWakeupTick() - clock_time())) > 80
↪   *CLOCK_16M_SYS_TIMER_CLK_1MS ){  //suspend time > 30ms.add gpio wakeup
        bls_pm_setWakeupSource(PM_WAKEUP_PAD);  //gpio CORE wakeup suspend
    }
    #endif
}
```

当按键没有释放时，通过手动设置 latency 为 0 或者一个很小的值，使得 sleep 时间较短，确保 sleep 时间不会超过 80ms，那么就不会发生按键按着的时候（drive pin 上有高电平）开启了 GPIO PAD 高电平唤醒。如下代码所示。

```
void blt_pm_proc(void)
{
  #if(BLE_APP_PM_ENABLE)
    #if (PM_DEEPSLEEP_RETENTION_ENABLE)
      bls_pm_setSuspendMask (SUSPEND_ADV | DEEPSLEEP_RETENTION_ADV | SUSPEND_CONN | DEEPSLE
    #else
      bls_pm_setSuspendMask (SUSPEND_ADV | SUSPEND_CONN);
    #endif
    #if (BLE_OTA_SERVER_ENABLE)
      if(ota_is_working)
      {
        bls_pm_setSuspendMask(SUSPEND_DISABLE);
      }
    #endif
    #if(UI_KEYBOARD_ENABLE)
      if(scan_pin_need || key_not_released )
      {
        bls_pm_setManualLatency(0);
      }
    #endif
```

Figure 4.7: 低功耗代码

MCU 进入 deepsleep 的两种情况：

- 一是连续 60s 没有任何事件会进入 deepsleep，这里的事件包括按键被按下，所以此时不会有 drive pin 高电平导致 deepsleep 无法进入；
- 二是卡键 60s 后进入 deepsleep，这时候虽然有 drive pin 上的高电平，SDK 会将卡键的 drive pin 唤醒电平极性取反，设为低电平唤醒，同样避免了这个问题（参考按键扫描章节的卡键处理）。

4.4 BLE 系统低功耗管理参考

在了解了该 BLE SDK 低功耗管理的实现原理基础上，user 可以很灵活地配置自己的低功耗管理，请参考 SDK demo "8208_ble_sample" 低功耗管理的参考 code。下面做一些解释。

在 main_loop 的 PM configuration 部分添加 blt_pm_proc 函数，注意这个函数要放在 main_loop 的最后面，以保证运行时间上最接近 blc_sdk_main_loop。因此 blt_pm_proc 函数里面低功耗管理的配置，需要根据 UI entry 部分各种任务的处理情况来做相应设置。

blt_pm_proc 函数低功耗管理配置几个要点总结如下：

(1) 某些任务需要关闭 sleep mode 时，设置 bltm.suspend_mask 为 SUSPEND_DISABLE。

(2) Advertising state 下连续广播时间达到 60s，设置 MCU 进入 deepsleep，唤醒源为 GPIO PAD（需要在 user initialization 部分提前设置按键 GPIO PAD）。判断是否广播超过 60s 的方法是用软件定时器，用变量 advertise_begin_tick 记录广播开始的 System Timer tick。

设置连续 60s 无广播进入 deepsleep 的目的是为了省功耗，防止 slave 设备没有被 master 连接时还一直在广播。user 需要根据自己的需求，对功耗进行评估后，决定如何处理 advertising state 的时间问题。

(3) Conn state slave role 时，所有的按键都已经释放、没有音频任务、LED 任务等，超过最近一次有效的任务时间 60s 以上，设置 MCU 进入 deepsleep，唤醒源为 GPIO PAD，并且在 deepsleep 记忆寄存器 DEEP_ANA_REG0 标记当前是在连接状态下进入 deepsleep（deepsleep 唤醒后，可以设置快速广播包尽快跟 master 连上）。

设置连续 60s 无有效任务进入 deepsleep 的目的是为了省功耗。实际只要将维持连接的功耗调到很小，也可以不进入 deepsleep。user 需要根据自己的需求和功耗状况决定如何实现。

Conn state slave role 时要进入 deepsleep，先调用 bls_ll_terminateConnection 向 master 发送一个 TERMINATE 命令，等到这个命令被 ack 后（此时会触发 BLT_EV_FLAG_TERMINATE 事件回调函数）再进入 deepsleep。这样做是为了确保 master 收到 salve 主动断连的请求后立刻断开。如果 slave 没有发送断连请求就进入 deepsleep，master 仍然处于连接状态并一直尝试去和 slave 同步，直到 connection timeout 触发。这个 connection timeout 时间可能很大（比如 20s），如果在 20s connection timeout 之前 slave 被唤醒并发广播尝试和 master 建立连接，由于 master 还处于上一次的连接状态中，会导致无法立刻和 slave 建立连接。应用上的体验就是回连速度很慢。

(4) 当有一些任务不能被长时间的 sleep 破坏时，可以手动设置 user_latency 为 0。如 key_not_released、DEVICE_LED_BUSY 时调用 API bls_pm_setManualLatency 将 user_latency 设为 0，那么 latency_use 就是 0，conn_interval 为 10ms 时 sleep 时间不超过 10ms。

(5) 在上面第 4 步的基础上，手动关闭 latency 后，每个 conn_interval 都要醒来，功耗稍微有点高，且按键扫描和 LED 任务的处理并不需要每个 conn_interval 都做一次，此时可以再做一些功耗优化。

LONG_PRESS_KEY_POWER_OPTIMIZE 为 1 时，当按键已经稳定后（key_matrix_same_as_last_cnt > 5），可以手动设置 latency 的值，调用 bls_pm_setManualLatency (3) 后，sleep 的时间不会超过 4 个 conn_interval。conn_interval 为 10 ms 时，每 40 ms 醒来一次处理 LED 和按键扫描。

user 在使用这个优化时，需要自行根据 conn_interval 的值和任务响应时间来评估。

4.5 应用层定时唤醒

在 Advertising state 和 Conn state Slave role，不考虑 GPIO PAD 唤醒前提下，一旦进入 sleep mode，只能在 BLE 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  bls_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 bls_pm_registerAppWakeupLowPowerCb 注册的回调函数，其原型和 API 如下：

```
typedef     void (*pm_appWakeupLowPower_callback_t)(int);
void        bls_pm_registerAppWakeupLowPowerCb(pm_appWakeupLowPower_callback_t cb);
```

以 Conn state Slave role 为例：

当 user 使用 bls_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。

- 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；
- 如果 app_wakeup_tick 在 T_wakeup 之后，MCU 还是会在 T_wakeup 唤醒。

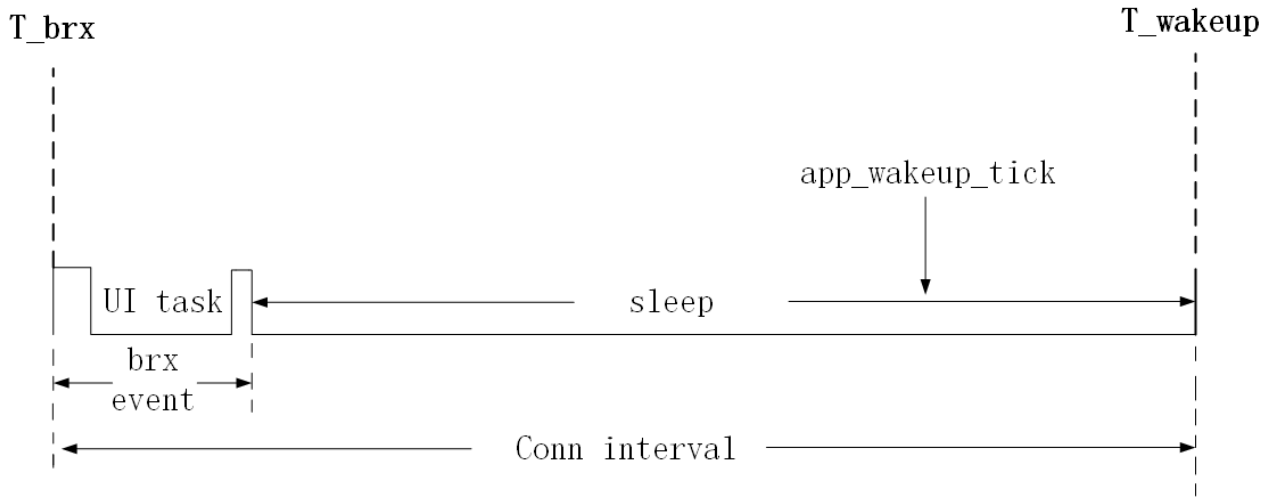


Figure 4.8: EarlyWake_upatapp_wakup_tick

5 低电检测

电池电量检测（battery power detect/check），在 Telink BLE SDK 和相关文档中也可能出现其他的名字，包括：电池电量检测（battery power detect/check）、低电池检测（low battery detect/check）、低电量检测（low power detect/check）、电池检查（battery detect/check）等。比如 SDK 中相关文件和函数出现 battery_check、battery_detect、battery_power_check 等命名。

本文档统一以"低电检测（low battery detect）"这个名称进行说明。

5.1 低电检测的重要性

使用电池供电的产品，由于电池电量会逐渐下降，当电压低到一定的值后会引起很多问题：

a) 8208 工作电压的范围为 1.8V~3.6V。当电压低于 1.8V 时，8208 已经无法保证稳定的工作。

b) 当电池电压较低时，由于电源的不稳定，Flash 的"write"和"erase"操作可能有出错的风险，造成 program firmware 和用户数据被异常修改，最终导致产品失效。根据以往的量产经验，我们将这个可能出风险的低压阀值设定为 2.0V。

根据上面的描述，使用电池供电的产品，必须设定一个安全电压值（secure voltage），只有当电压高于这个安全电压的时候才允许 MCU 继续工作；一旦电压低于安全电压，MCU 停止运行，需要立刻被 shutdown（SDK 上使用进入 deepsleep mode 来实现）。

安全电压也称为报警电压，这个电压值的选取，目前 SDK 默认使用 2.0V。

注意：

> 低压保护阈值数值 2.0V 只是示例、参考值。客户要根据实际情况评估修改这些阈值，如果 user 在硬件电路中出现了不合理的设计，导致电源网络稳定性降低，都要酌情提高安全阈值。

对于 Telink BLE SDK 开发实现的产品，只要使用了电池供电，低电检测都必须是该产品整个生命周期实时运行的任务，以保证产品的稳定性。

低电压检测对于 Flash 操作的保护，在 Flash 介绍的章节中会继续介绍。

5.2 低电检测的实现

低电检测需要使用 ADC 对电源电压进行测量。user 请参考文档 8208 Datasheet 相关 ADC 章节，先对 B80 的 ADC 模块进行必要的了解。

低电检测的实现，结合 SDK demo "8208_ble_sample"给出的实现来说明，参考文件 vendor/common/battery_check.h 和 vendor/common/battery_check.c。

必须确保 app_config.h 文件中宏"BATT_CHECK_ENABLE"是被打开的，这个宏默认是关闭的，user 使用低电检测功能时需要注意。

```
#define BATT_CHECK_ENABLE                   1
```

5.2.1 低电检测的注意事项

低电检测是一个基本的 ADC 采样任务，在实现 ADC 采样电源电压时，有一些需要注意的问题，说明如下：

5.2.1.1 建议使用 VBAT 输入通道

8208 的 ADC 输入通道上支持在"VCC/VBAT" 输入通道上对电源电压进行 ADC 采样，对应下面变量 ADC_InputPchTypeDef 中最后一个"VBAT"。

可用的 GPIO 输入通道为 PB0~PB7、PC4、PA3 对应的 input channel。

```
/*ADC analog positive input channel selection enum*/
typedef enum {
......
    B0P,
    B1P,
    B2P,
    B3P,
    B4P,
    B5P,
    B6P,
    B7P,
    C4P,
    A3P,
......
    VBAT,
}ADC_InputPchTypeDef;
```

目前"8208_ble_sample"选择的是 VBAT 输入通道采样。

5.2.1.2 只能使用差分模式

虽然 8208 ADC input mode 同时支持单端模式（Single Ended Mode）和差分模式（Differential Mode），但由于某些特定的原因，Telink 规定：只能使用差分模式，单端模式不允许使用。

差分模式的 input channel 分为 positive input channel（正端输入通道）和 negative input channel（负端输入通道），被测量的电压值为 positive input channel 电压减去 negative input channel 电压得到的电压差。

如果 ADC 采样的 input channel 只有 1 个，使用差分模式时，将当前 input channel 设置为 positive input channel，将 GND 设为 negative input channel。这样二者的电压差和 positive input channel 电压相等。

SDK 中低压检测使用了差分模式，在 adc_init() 函数中调用 adc_set_input_mode_chn_misc 函数进行差分模式选择，code 如下。

```
void adc_init(void){
...
    adc_set_input_mode_chn_misc(DIFFERENTIAL_MODE);
```

```
...
}
```

5.2.1.3 必须使用 Dfifo 模式获得 ADC 采样值

对于 8208，Telink 规定：只能使用 Dfifo 模式来实现 ADC 采样值的读取。可参考 dirver 中如下函数的实现。

```
unsigned int adc_sample_and_get_result(void);
```

5.2.1.4 不同的 ADC 任务需要切换

参考《8208 Datasheet》可知，ADC 状态机仅有 Misc channel。

5.2.1.5 低电检测初始化

参考 app_battery_power_check 函数里的初始化实现。

ADC 初始化的顺序必须满足下面的流程：先 power off（掉电）sar adc，然后配置其他参数，最后 power on（上电）sar adc。所有 ADC 采样的初始化都必须遵循这个流程。

```
adc_init();
adc_vbat_channel_init();
adc_power_on_sar_adc(1);
```

Sar adc power on 与 power off 之前的配置，user 尽量不要去修改，使用这些默认的设置就行。

app_battery_power_check 初始化函数在 app_battery_power_check 中调用的 code 为：

```
if(!adc_hw_initialized){
        adc_hw_initialized = 1;
        adc_init();
        adc_vbat_channel_init();
        adc_power_on_sar_adc(1);
    }
```

这里使用了一个变量 adc_hw_initialized，只有该变量为 0 时调用一次初始化，并将其置 1；该变量为 1 时不再初始化。adc_hw_initialized 在下面 API 中也会被操作。

```
void battery_set_detect_enable (int en)
{
    lowBattDet_enable = en;
    if(!en){
        adc_hw_initialized = 0;   //need initialized again
    }
}
```

使用了 adc_hw_initialized 的设计可以实现的功能有：

(1) 与其他 ADC 任务（"ADC other task"）的切换

先不考虑 sleep mode (suspend/deepsleep retention) 的影响，只分析低电检测与其他 ADC 任务的切换。

因为需要考虑低电检测与其他 ADC 任务的切换使用，可能出现 adc init 被多次执行，所以不能写到 user initialization 中，必须在 main_loop 里实现。

第一次执行 app_battery_power_check 函数时，adc init 被执行，且后面不会被反复执行。

一旦"ADC other task"需要执行时，将抢走 ADC 的使用权，确保"ADC other task"初始化时必须调用 battery_set_detect_enable(0)，此时会将 adc_hw_initialized 清 0。

等"ADC other task"完成后，交出 ADC 的使用权。app_battery_power_check 再次执行，由于 adc_hw_initialized 值为 0，必须再次执行 adc_vbat_detect_init，这样就保证了低电检测每次切回来时都会重新初始化。

(2) 对 suspend 和 deepsleep retention 的自适应处理

将 sleep mode 考虑进来。

adc_hw_initialized 这个变量使用必须定义成一个"data_reload 段"上的变量，不能定义到"data"段或"bss"段上。定义在"data_reload 段"上可以保证每次 deepsleep retention wake_up 后在执行 software bootloader（即 cstartup_xxx.S）时这个变量会被重新初始化为 0；而 sleep wake_up 后这个变量可以保持不变。

adc init 函数里面配置的 register 的共同特征是：在 suspend mode 下不掉电，可以保存状态；在 deepsleep retention mode 下会掉电。

如果 MCU 进入 suspend mode，醒来后再次执行 app_battery_power_check 时，adc_hw_initialized 的值和 suspend 之前一致，不需要重新执行 adc init 函数。

如果 MCU 进入 deepsleep retention mode，醒来后 adc_hw_initialized 为 0，必须重新执行 adc init，ADC 相关的 register 状态需要被重新配置。

adc init 函数中设定 register 的状态可以在 suspend 期间保持不掉电。

参考文档"低功耗管理"部分对 suspend mode 的说明可知，Dfifo 相关的寄存器在 suspend mode 会掉电，所以以下两句 code 没有放到 adc init 函数中，而是在 app_battery_power_check 函数中，确保每次低电检测前都重新设置。

```
adc_config_misc_channel_buf((u16 *)adc_dat_buf,ADC_SAMPLE_NUM<<2);
dfifo_enable_dfifo2();
```

5.2.1.6 低电检测处理

在 main_loop 中，调用 app_battery_power_check 函数实现低电检测的处理，相关 code 如下：

```
u8      lowBattDet_enable = 1;
_attribute_data_reload_ u8      adc_hw_initialized = 0;   //note: can not be retention variable
void battery_set_detect_enable (int en)
{
    lowBattDet_enable = en;
```

```
    if(!en){
        adc_hw_initialized = 0;   //need initialized again
    }
}
int battery_get_detect_enable (void)
{
    return lowBattDet_enable;
}
if(battery_get_detect_enable() && clock_time_exceed(lowBattDet_tick, 500000) ){
    lowBattDet_tick = clock_time();
    app_battery_power_check(VBAT_ALRAM_THRES_MV);
}
```

lowBattDet_enable 默认值为 1，低电检测是默认允许的，MCU 上电后立刻可以开始低电检测。该变量需要设置成 retention_data，确保 deepsleep retention 不能修改它的状态。

只有在其他 ADC 任务需要抢占 ADC 使用权时，才能改变 lowBattDet_enable 的值：当其他 ADC 任务开始时，调用 battery_set_detect_enable(0)，此时 main_loop 中不会再调用 app_battery_power_check 函数；在其他 ADC 任务结束后，调用 battery_set_detect_enable(1)，交出 ADC 使用权，此时 main_loop 中又可以调用 app_battery_power_check 函数。

通过变量 lowBattDet_tick 来控制低电检测的频率，Demo 中为每 500ms 执行一次低电检测。User 可以根据自己的需求来修改这个时间值。

app_battery_power_check 函数的具体实现看起来比较繁琐，涉及到低电检测的初始化、Dfifo 的准备、数据的获取、数据的处理、低电报警的处理等等。

ADC 采样数据的获取使用了 Dfifo mode，Dfifo 默认采样 8 笔数据，去掉最大最小值后计算平均值。

adc_init 函数里可以看到每个 adc 采样的周期为 10.4us，所以获取数据过程大概 83us。

可以看到 Demo 中宏"ADC_SAMPLE_NUM"可以被修改为 4，缩短 ADC 采样时间到 41us。推荐使用 8 笔数据的方法，计算结果会更加准确。

```
#define ADC_SAMPLE_NUM      8

#if (ADC_SAMPLE_NUM == 4)   //use middle 2 data (index: 1,2)
    u32 adc_average = (adc_sample[1] + adc_sample[2])/2; #elif(ADC_SAMPLE_NUM == 8)     //use
 ↪  middle 4 data (index: 2,3,4,5)
    u32 adc_average = (adc_sample[2] + adc_sample[3] + adc_sample[4] +
adc_sample[5])/4;
#endif
```

5.2.1.7 低压报警

app_battery_power_check 的参数 alram_vol_mv 指定低电检测的报警电压，单位为 mV。根据前文介绍，SDK 中默认设置为 2000mV。在 main_loop 的低压检测中，当电源电压低于 2000mV 时，进入低压范围。

低压报警的处理 demo code 如下所示。低压后必须 shutdown MCU，不能再进行其他工作。

“8208_ble_sample”使用进入 deepsleep 的方式来实现 shutdown MCU，并且设置了按键可以唤醒。

低压报警的处理，除了必须 shutdown 外，user 可以修改其他的报警行为。

下面 code 中，使用 LED 灯做了 3 次快闪，告知产品使用者需要充电或更换电池。

```
u8 battery_check_returnVaule = 0;
if(analog_read(USED_DEEP_ANA_REG) & LOW_BATT_FLG){
    battery_check_returnVaule = app_battery_power_check(VBAT_ALRAM_THRES_MV + 200);  //2.2 V
}
else{
    battery_check_returnVaule = app_battery_power_check(VBAT_ALRAM_THRES_MV);  //2.0 V
}
if(battery_check_returnVaule){
    analog_write(USED_DEEP_ANA_REG,  analog_read(USED_DEEP_ANA_REG)&(~LOW_BATT_FLG));  //clr
}
else{
    #if (UI_LED_ENABLE)  //led indicate
        for(int k=0;k<3;k++){
            gpio_write(GPIO_LED_BLUE, LED_ON_LEVAL);
            sleep_us(200000);
            gpio_write(GPIO_LED_BLUE, !LED_ON_LEVAL);
            sleep_us(200000);
        }
    #endif
    analog_write(USED_DEEP_ANA_REG,  analog_read(USED_DEEP_ANA_REG) | LOW_BATT_FLG);  //mark
    GPIO_WAKEUP_FEATURE_LOW;

    cpu_set_gpio_wakeup (GPIO_WAKEUP_FEATURE, Level_High, 1);  //drive pin pad high wakeup
 ↪  deepsleep

    cpu_sleep_wakeup(DEEPSLEEP_MODE, PM_WAKEUP_PAD, 0);  //deepsleep
}
```

“8208_ble_sample”被 shutdown 后，进入可被唤醒的 deepsleep mode。此时如果发生按键唤醒，SDK 会在 user initialization 的时候先快速做一次低电检测，而不是等到 main_loop 中检测。这样处理的原因是为了避免应用上的错误，举例说明如下：

如果低电报警时 LED 闪烁已经提示了产品使用者，然后进入 deepsleep 又被唤醒，从 main_loop 的处理来看，需要至少 500ms 的时间才会去做低电检测。在 500ms 之前，slave 的广播包已经发很久了，很可能会跟 master 已经连接上了。这样的话，就出现已经低电报警的设备又继续工作的 bug 了。

因为这个原因，SDK 必须在 user initialization 的时候就提前做低电检测，必须在这一步就阻止发生上面的情况。所以在 user initialization 的时候，添加低电检测：

```
if(analog_read(USED_DEEP_ANA_REG) & LOW_BATT_FLG){
            battery_check_returnVaule = app_battery_power_check(VBAT_ALRAM_THRES_MV + 200);  //
 ↪  2.2 V
        }
```

```
        else{
            battery_check_returnVaule = app_battery_power_check(VBAT_ALRAM_THRES_MV);  //2.0 V
        }
```

根据 USED_DEEP_ANA_REG 模拟寄存器的值可以判断是否低电报警 shutdown 被唤醒的情况，此时进行快速低电检测，并且将之前的 2000mV 报警电压提高到 2200mV（称为恢复电压）。提高 200mV 的原因是：

低压检测会有一些误差，无法保证测量结果的准确性和一致性。比如误差在 20mV，可能第一次检测到的电压是 1990mV 进入 shutdown 模式，然后唤醒后在 user initialization 的时候再次检测到的电压值是 2005mV。如果还是以 2000mV 为报警电压的话，还是无法阻止上面描述的 bug。

所以需要在 shutdown 模式唤醒后的快速低电检测时，将报警电压稍微调高一些，调高的幅度比低电检测的最大误差稍大。

只有当某次低电检测发现电压低于 2000mV 进入 shutdown 模式后，才会出现恢复电压 2200mV，所以 user 不用担心这个 2200mV 会对实际电压 2V~2.2V 的产品误报低压。产品使用者看到低压报警指示后，进行充电或更换电池后，满足恢复电压的要求，产品恢复正常使用。

6 OTA

为了实现 8208 BLE slave 的 OTA 功能，需要一个设备作为 BLE OTA master。

OTA master 可以是实际与 slave 配套使用的蓝牙设备（需要在 APP 里实现 OTA），也可以使用 Telink 的 BLE master kma dongle，下面以 Telink 的 BLE master kma dongle 作为 ota master 来详细介绍 OTA，相关 code 实现也可参考多连接 SDK 下的 feature_ota_big_pdu。

8208 支持 Flash 多地址启动：除了 Flash 的首地址 0x00000，128K Flash 还支持从 Flash 高地址 0x10000（64K）读取 firmware 运行，512K Flash 还支持从 Flash 高地址 0x10000（64K）、0x20000（128K）、0x40000（256K）读取 firmware 运行。本文档以高地址 0x20000 为例来介绍 OTA。

6.1 Flash 架构设计和 OTA 流程

6.1.1 FLASH 存储架构

使用启动地址 0x20000 时，SDK 编译出来的 firmware size 应不大于 128K，即 flash 的 0~0x20000 之间的区域存储 firmware；如果超过 128K 必须使用启动地位 0 和 0x40000 交替升级，此时最大 firmware size 不得超过 240K，这是由于 0x7C000 ~ 0x7FFFF 需要存储一些 SDK 相关的信息，详细可以参考 Flash 章节的介绍。另外，对于 128K Flash，使用启动地址为 0 和 0x10000 交替 OTA 升级，其 firmware size 不得超过 48K。

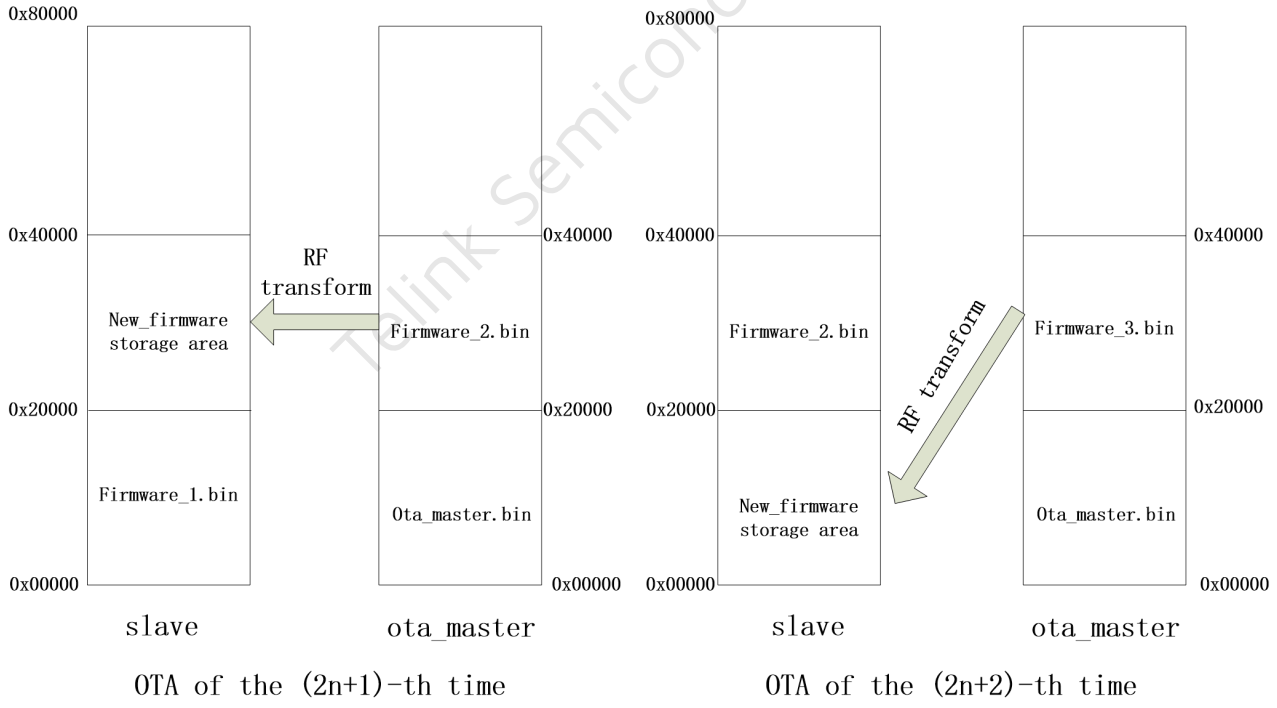


Figure 6.1: Flash 存储结构

(1) ota master 将新的 firmware2 烧写到 0x20000~0x40000 的区域。

(2) 第 1 次 OTA：

- slave 上电时从 flash 的 0~0x20000 区域读程序启动，运行 firmware1；

- firmware1 运行，初始化的时候将 0x20000~0x40000 区域清空，该区域将作为新的 firmware 的存储区。
- 启动 OTA，master 通过 RF 将 firmware2 空运到 slave 的 0x20000~0x40000 区域。slave reboot（重新启动，类似一次断电并重新上电）。

(3) 将新的 firmware3 烧写到 ota master 的 0x20000~0x40000 的区域。

(4) 第 2 次 OTA：

- slave 上电时从 flash 的 0x20000~0x40000 区域读程序启动，运行 firmware2；
- firmware2 运行，初始化的时候将 0~0x20000 区域清空，该区域将作为新的 firmware 的存储区。
- 启动 OTA，master 通过 RF 将 firmware3 空运到 slave 的 0~0x20000 区域。slave reboot。

(5) 后面的 OTA 过程重复上面（1）~（4）过程，可理解为（2）代表第 2n+1 次 OTA，(4) 代表第 2n+2 次 OTA。

6.1.2 OTA 更新流程

以上面的 FLASH 存储结构为基础，详细说明 OTA 程序更新的过程。

首先介绍一下多地址启动机制（只介绍前两个启动地址 0x00000 和 0x20000）：MCU 上电后，默认从 0 地址启动，首先去读 flash 0x08 的内容，若该值为 0x4b，则从 0 地址开始搬移代码到 RAM，并且之后所有的取指都是从 0 地址开始，即取指地址 = 0+PC 指针的值；若 0x08 的值不为 0x4b，MCU 直接去读 0x20008 的值，若该值为 0x4b，则 MCU 从 0x20000 开始搬代码到 RAM，并且之后所有的取指都是从 0x20000 地址开始，即取指地址 = 0x20000+PC 指针的值。

所以只要修改 0x08 和 0x20008 标志位的值，即可指定 MCU 执行 FLASH 哪部分的代码。

SDK 上某一次（2n+1 或 2n+2）上电及 OTA 过程为：

(1) MCU 上电，通过读 0x08 和 0x20020 的值和 0x4b 作比较，确定启动地址，然后从对应的地址启动并执行代码。此功能由 MCU 硬件自动完成。

(2) 程序初始化过程中，读 MCU 硬件寄存器判断 MCU 刚才是从哪个地址启动：

若从 0 启动，将 ota_program_offset 设为 0x20000，并将 0x20000 区域非 0xff 的内容全部擦除为 0xff，表示下一次 OTA 获得的新 firmware 会存入 0x20000 开始的区域；

若从 0x20000 启动，将 ota_program_offset 设为 0x0，并将 0x0 区域非 0xff 的内容全部擦除为 0xff，表示下一次 OTA 获得的新 firmware 会存入 0x0 开始的区域。

(3) Slave 程序正常运行，OTA master 上电运行，并与 slave 建立 BLE 连接。

(4) 在 OTA master 端 UI 触发进入 OTA 模式（可以是按键、PC 工具写内存等）。OTA master 进入 OTA 模式后，首先需要获取 slave OTA service 数据 Attribute 的 Attribute Handle 的值（可以 slave 事先和 master 约定好，也可以通过 read_by_type 获取这个 handle 值）。

(5) OTA master 获取了 slave OTA service 数据 Attribute 的 Attribute Handle 值后，获取当前 slave FLASH 程序的 firmware 版本号。

注意：

> 获取版本号需要 user 自行实现。

(6) master 确定要做 OTA 更新后，先发一个 OTA_start 命令通知 slave 进入 OTA 模式。

(7) Slave 收到 OTA start 命令后，进入 OTA 模式，等待 master 发 OTA 数据。

(8) Master 从 0x20000 开始的区域读预先存储好的 firmware，不间断的向 slave 发送 OTA 数据，直至整个 firmware 都发过去。

(9) Slave 接收 OTA 数据，向 ota_program_offset 开始的区域存储。

(10) master 端发完所有的 OTA 数据后，检查这些数据 slave 是否都正确收到（调用底层 BLE 的相关函数判断 link layer 的数据是否都被正确 ack）。

(11) master 确定所有的 OTA 数据都被 slave 正确收到后，发送一个 OTA_END 命令。

(12) Slave 收到 OTA_END 命令，将新 firmware 区域偏移地址 0x08（即 ota_program_offset+0x08）写为 0x4b，将之前老的 firmware 存储区域偏移地址 0x08 的地方写为 0x00，表示下一次程序启动后将从新的区域搬代码执行。

(13) 将 slave reboot，新的 firmware 生效。

在整个 OTA 更新过程中，slave 会不断检查是否有错包和丢包，同时也会不断检查是否超时（OTA 开始的时候启动一个计时），一旦有错包、丢包或超时，slave 会认为更新失败，程序 reboot，使用之前的 firmware。

以上流程 slave 端相关操作在 SDK 上已经实现，user 不需要添加任何东西，master 端需要额外的程序设计，后面会详细介绍。

6.1.3 修改 Firmware size 和 boot address

API blc_ota_setFirmwareSizeAndBootAddress 同时支持修改启动地址和最大 firmware size。这个启动地址指的是 OTA 设计中除了 0 地址外另一个存储 New_firmware 的地址 (只能是 0x10000、0x20000 或 0x40000)。

Table 6.1: Firmware size 和 boot address 对照

Flash size	Firmware_Boot_address	Firmware size (max)/K
128	0x10000	48
512	0x10000	64
512	0x20000	128
512	0x40000	240

SDK 中默认的最大 firmware size 为 240K，对应的启动地址为 0x00000 和 0x40000。user 可以调用 API blc_ota_setFirmwareSizeAndBootAddress 来进行设置最大 firmware size：

```
ble_sts_t blc_ota_setFirmwareSizeAndBootAddress(int firmware_size_k, multi_boot_addr_e
↪  new_fw_addr);
```

firmware_size_k 的设置一定要 4K byte 对齐，比如 size 为 97K 时需要设为 100K。

参数 multi_boot_addr_e 表示可供选择的启动地址，共有三种：

```
typedef enum{
    MULTI_BOOT_ADDR_0x10000     = 0x10000,  //64 K
    MULTI_BOOT_ADDR_0x20000     = 0x20000,  //128 K
    MULTI_BOOT_ADDR_0x40000     = 0x40000,  //256 K
}multi_boot_addr_e;
```

这个 API 只能在 main 函数中 cpu_wakeup_init 之前调用，否则无效。原因是 cpu_wakeup_init 函数中需要根据 firmware_size 和 boot_addr 的值做一些设置。

如果固件不需要这么大的空间，例如 firmware size 不超过 60kB，则只使用两个 128kB 空间 (0x00000 ~ 0x20000, 0x20000 ~ 0x40000) 的一部分。要使用冗余空间作为数据存储区域，可以执行以下设置。

```
blc_ota_setFirmwareSizeAndBootAddress(60, 0x20000);
```

通过上面的配置，两个 60kB Flash 区域 0x00000 ~ 0x0F000 和 0x20000 ~ 0x2F000 可以用作固件的存储空间，而两个 68kB 的 Flash 区域 0x0F000 ~ 0x20000 和 0x2F000 ~ 0x40000 可以作为用户数据的存储空间。

128kB Flash OTA 中的情况与 512kB 基本相同，只是将偏移量需配置为 0x10000。1) 对于 128kB Flash, SDK 的默认最大支持的固件大小为 48kB。2) 对于 128kB Flash，通过修改 SDK 配置，如下图，固件的最大大小可以达到 56kB，但不会有额外的 Flash 空间来存储用户数据。

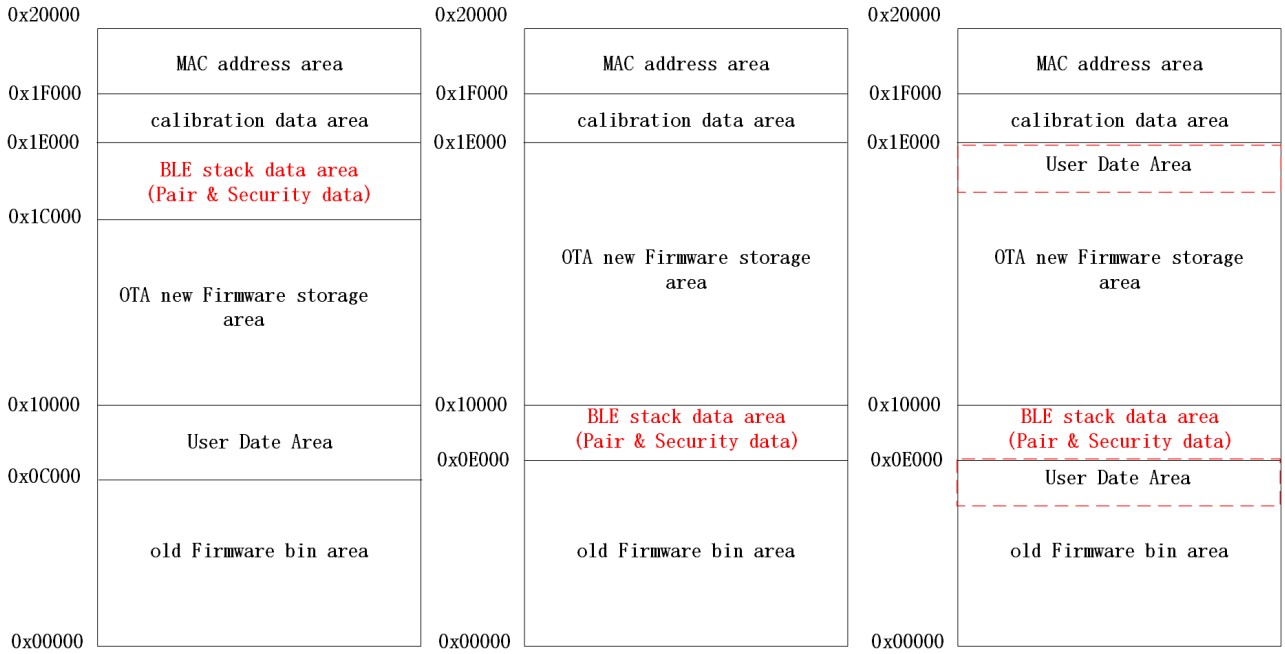


Figure 6.2: 128kB Flash 存储结构

6.2 OTA 模式 RF 数据处理

6.2.1 Attribute Table 中 OTA 的处理

Slave 端在 Attribute Table 中添加 OTA 的相关内容，其中 OTA 数据 Attribute 的 att_readwrite_callback_t r 和 att_readwrite_callback_t w 分别设为 otaRead 和 otaWrite，将属性设为 Read 和 Write_without_Rsp（Telink 的 Master KMA Dongle 默认采用 Write Command 发数据，不需要 slave 回 ack，速度会更快）。

```
// OTA attribute values
static const u8 my_OtaCharVal[19] = {
    CHAR_PROP_READ | CHAR_PROP_WRITE_WITHOUT_RSP,
    U16_LO(OTA_CMD_OUT_DP_H), U16_HI(OTA_CMD_OUT_DP_H),
    TELINK_SPP_DATA_OTA,
};

    {4,ATT_PERMISSIONS_READ, 2,16,(u8*)(&my_primaryServiceUUID),    (u8*)(&my_OtaServiceUUID),
↪   0},
    {0,ATT_PERMISSIONS_READ, 2, sizeof(my_OtaCharVal),(u8*)(&my_characterUUID), (u8*)
↪   (my_OtaCharVal), 0},
    {0,ATT_PERMISSIONS_RDWR,16,sizeof(my_OtaData),(u8*)(&my_OtaUUID),   (&my_OtaData),
↪   &otaWrite, NULL},
    {0,ATT_PERMISSIONS_READ, 2,sizeof (my_OtaName),(u8*)(&userdesc_UUID), (u8*)(my_OtaName), 0},
```

master 向 slave 发送 OTA 数据时，实际是向上面第 3 个 Attribute 写数据，master 需要知道这个 Attribute 在整个 Attribute Table 中的 Attribute Handle。如果 user 使用 master 和 slave 事先约定好 Attribute Handle 值的方法，可以直接在 master 端定义 Attribute Handle 的值。

6.2.2 OTA Protocol

Master 端通过 L2CAP 层的 Write Command 向 slave 发命令和数据。

OTA_CMD 组成：

OTA 的 CMD 的 PDU 如下：

Table 6.2: OTA 的 CMD 的 PDU

OTA Command Payload	
Opcode (2 octet)	invalid data

Opcode：

Table 6.3: CMD 的 opcode

Opcode	Name
0xFF00	CMD_OTA_VERSION
0xFF01	CMD_OTA_START
0xFF02	CMD_OTA_END

(1) CMD_OTA_VERSION

该命令为获得 slave 当前 firmware 版本号的命令，user 可以选择使用。在使用该命令时，可通过 slave 端预留的回调函数来完成 firmware 版本号的传递。

```
void blc_ota_registerOtaFirmwareVersionReqCb(ota_versionCb_t cb);
```

server 端在收到 CMD_OTA_VERSION 命令时会触发该回调函数。

(2) CMD_OTA_START

该命令为 OTA 升级开始命令，master 发这个命令给 slave，用来正式启动 OTA 更新。

(3) CMD_OTA_END

该命令为结束命令，当 master 确定所有的 OTA 数据都被 slave 正确接收后，发送 OTA end 命令。为了让 slave 再次确定已经完全收到了 master 所有数据（double check，加一层保险），OTA end 命令后面带 4 个有效的 bytes，后面详细介绍。

Table 6.4: OTA 结束命令

-	CMD_data	-
Adr_index_max (2 octets)	Adr_index_max_xor (2 octets)	Reserved

- Adr_index_max：最大的 adr_index 值
- Adr_index_max_xor：Adr_index_max 的异或值，供校验使用
- Reserved：保留供以后功能扩展使用

OTA_Data 介绍：

Table 6.5: OTA data

-	OTA PDU	-
Adr_Index (2 octets)	Data(16 octets)	CRC (2 octets)

注意：

> OTA PDU 长度固定大小为 16octets

OTA_PDU Format：

前两个 byte 的范围在 firmware_size_k 之内时，表示一个 OTA 数据。由于 firmware size 不超过 128K (0x20000)，OTA data packet 中每次传送 16 byte 的 firmware 数据，使用的 adr_index 为实际 firmware 地址除以 16 的值。adr_index=0，表示 OTA 数据是 firmware 地址 0x0 ~ 0xF 的值；adr_index=1，表示 OTA 数据是 firmware 地址 0x10 ~ 0x1F 的值。最后两个 byte 是将前面的 Adr_Index 和 Data 进行一个 CRC_16 计算得到第一个 CRC 的值，slave 收到 OTA data 后，会进行同样的 CRC 计算，只有两者计算的 CRC 吻合时，才认为这是一个有效数据。

6.2.3 RF Transfer 处理方法

基于 BLE link layer RF 数据自动 ack 确保所有数据包不丢的前提，OTA 的数据 transform 不检查每一个 OTA 数据是否被 ack，即 master 通过 write command 发一个 ota 数据后，不在软件上检查对方是否有 ack 信息回复，只要 master 端硬件 TX buffer 缓存的待发送数据未达到一定数量，直接将下一笔数据丢进 TX buffer。

下面将对 OTA 具体实现流程进行介绍, 阐述整个 RF Transform 中 Salve 和 Master 的交互过程。

OTA 具体实现：

master 端 OTA 相关的操作为：

(1) 检测查询是否有触发进入 OTA 模式的行为，一旦检测到该行为，进入 OTA 模式。

(2) master 向 slave 传送 OTA 命令和数据，需要知道 slave 端当前 OTA 数据的 Attribute 的 Attribute Handle 值。

若 user 采用事先约定好的方式，直接定义该值；

若没有事先约定好，采用 Read By Type Request 的方式获得这个 Attribute Handle 值。

Telink 所有 BLE SDK 的 OTA data 的 UUID 都是 16bytes，且永远都是下面这个值：

```
#define TELINK_SPP_DATA_OTA     {0x12,0x2B,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,
↪  0x03,0x02,0x01,0x00}
```

在 master 的 Read By Type Request 中将 Type 设置为这 16 个 bytes 的 UUID，slave 端回复的 Read By Type Rsp 中可以查到 OTA UUID 所在的这个 Attribute Handle，如下图所示，master 可以查到 Attribute Handle 的值为 0x0031。

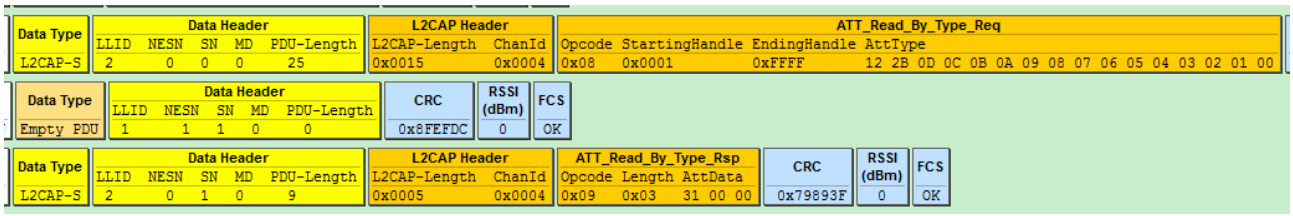


Figure 6.3: master 通过 Read By Type Request 获取 OTA 的 Attribute Handle

(3) 获取 slave 当前 firmware 版本号，决定是否要继续做 OTA 更新（若版本已经最新，不需要更新）。这一步为 user 自己选择是否要做。该 BLE SDK 不提供具体的版本号获取办法，user 可以自行发挥。目前的 B80 BLE SDK 中并没有实现版本号的传送。user 可以使用 write cmd 的形式通过 OTA version cmd 向 slave 传送一个获取 OTA version 的请求，但是 slave 那端在收到 OTA version 请求的时候只提供一个回调函数，user 自己在回调函数里想办法将 slave 端的版本号传送给 master（如手动送一个 NOTIFY/INDICATE 的数据）。

(4) 启动 OTA 开始的一个计时，后面要不断检测该计时是否超过 30 秒（这只是个默认参考时间，实际根据 user 测试的正常 OTA 需要多少时间后再做评估修改）。

如果超过 30 秒认为 OTA 超时失败，因为 slave 端收到 OTA 数据后会校验 CRC，一旦 CRC 错误或者出现其他错误（如烧写 flash 错误），就会认为 OTA 失败，直接程序重启，此时 link layer 无法 ack master，master 端的数据一直发不出去导致超时。

(5) 读取 Master flash 0x20018~0x2001b 四个字节，确定 firmware 的 size。

这个 size 是由我们的编译器实现的，假设 firmware 的 size 为 20k = 0x5000，那么 firmware 的 0x18 ~ 0x1b 的值为 0x00005000，所以在 0x20018 ~ 0x2001b 可以读到 firmware 的大小。

如下图所示的 bin 文件，0x18 ~ 0x1b 内容为 0x0000A164，所以大小为 0xa164 = 41316Bytes，从 0x0000 到 0xa164。

```
00000000 58 80 00 00 00 00 5D 02  4B 4E 4C 54 10 02 88 00
00000010 E6 80 00 00 00 00 00 00  64 A1 00 00 00 00 00 00
00000020 0C 64 81 A2 22 0B 1A 40  C0 06 C0 06 C0 06 C0 06
00000030 C0 06 C0 06 C0 06 C0 06  C0 06 C0 06 C0 06 C0 06
00000040 C0 06 C0 06 C0 06 C0 06  C0 06 C0 06 C0 06 C0 06
00000050 C0 06 C0 06 C0 06 C0 06  C0 06 C0 06 C0 06 C0 06
00000060 C0 06 C0 06 C0 06 C0 06  C0 06 C0 06 C0 06 C0 06
00000070 C0 06 C0 06 C0 06 C0 06  C0 06 C0 06 C0 06 C0 06
```

Figure 6.4: firmware 示例-开头部分

```
0000A0D0 1E 00 00 00 02 09 05 00  04 00 01 00 00 00 0A 00
0000A0E0 00 07 01 00 40 42 0F 00  33 21 12 34 29 78 64 54
0000A0F0 56 07 82 58 09 79 86 19  97 74 24 67 62 42 81 14
0000A100 57 20 42 53 32 37 32 74  02 04 01 03 00 00 00 00
0000A110 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0000A120 02 15 11 00 06 00 00 00  00 00 00 00 00 00 00 00
0000A130 00 00 00 00 00 00 00 00  00 00 00 00 11 00 00 00
0000A140 00 40 07 00 FF FF FF FF  80 C3 C9 01 A1 22 00 00
0000A150 06 00 00 00 FF FF FF FF  FF FF FF FF FF FF FF FF
0000A160 AD CB 7A DE
```

Figure 6.5: firmware 示例-结尾部分

(6) 向 slave 发一个 OTA start 命令，通知 slave 进入 OTA 模式，等待 master 端的 OTA 数据，如下图所示。

Data Type	Data Header					L2CAP Header		ATT_Write_Command			CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue			
L2CAP-S	2	0	0	1	9	0x0005	0x0004	0x52	0x0031	01 FF	0x61875B	0	OK

Figure 6.6: master 发 OTA start

(7) 从 Master flash 0x20000 区域开始每次读 16 个 byte 的 firmware，填入 OTA data packet，设置对应的 adr_index，并计算 CRC 值，将 packet push 到 TX fifo，一直到 firmware size 最后一个 16 byte 为止，将 firmware 所有的数据全部发送给 slave。

数据发送方法如前面介绍，使用 OTA data 的格式，有效数据为 20 bytes，前两个 bytes 放 adr_index，紧跟 16 个有效的 firmware 数据，最后两个是前 18 个数据的 CRC 计算值。

注意，如果 firmware 最后一笔数据不是 16 字节对齐，需要将剩余的部分按 0xff 补对齐，计算 CRC 的时候需要将补充的数据计算进去。

结合上图所示的 bin 文件来详细介绍 OTA 数据如何拼装。

第一笔数据：adr_index 为 0x00 00，16 个数据为 0x0000 ~ 0x000f 地址的值，然后这 18 个数据计算 CRC，假设 CRC 结果为 0xXYZW，那么 20bytes 排列为：

0x00 0x00 0x58 0x80 .... 省略 12 个 bytes..... 0x88 0x00 0xZW 0xXY

第二笔数据：

0x01 0x00 0xE6 0x80 .... 省略 12 个 bytes..... 0x00 0x00 0xJK 0xHI

第三笔数据：

0x02 0x00 0x0C 0x64 .... 省略 12 个 bytes..... 0xC0 0x06 0xNO 0xLM

.....

倒数第二笔数据：

0x15 0x0a 0x06 0x00 .... 省略 12 个 bytes..... 0xff 0xff 0xST 0xPQ

最后一笔数据：

0x16 0x0a 0xad 0xcb 0x7a 0xde **0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff** 0xWX 0xUV

12 个 0xff 为补齐的数据。

0x16 ~0xff 共 18 个 bytes 的 CRC 计算结果为 0xUVWX。

上面的数据如下图所示：

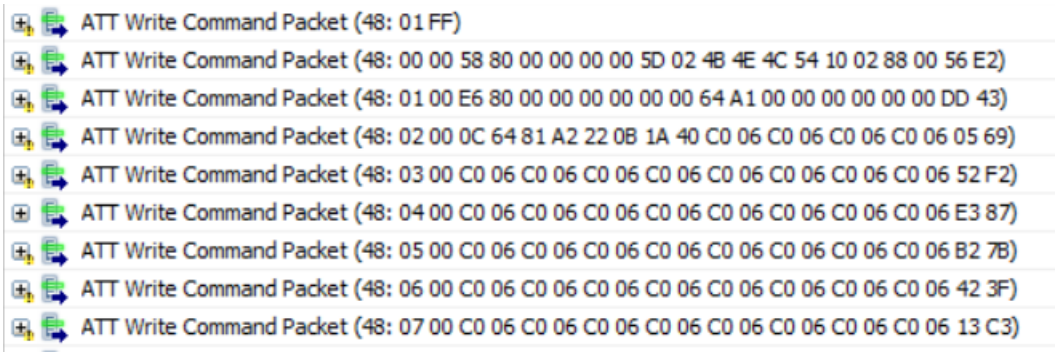


Figure 6.7: master OTA 数据 1

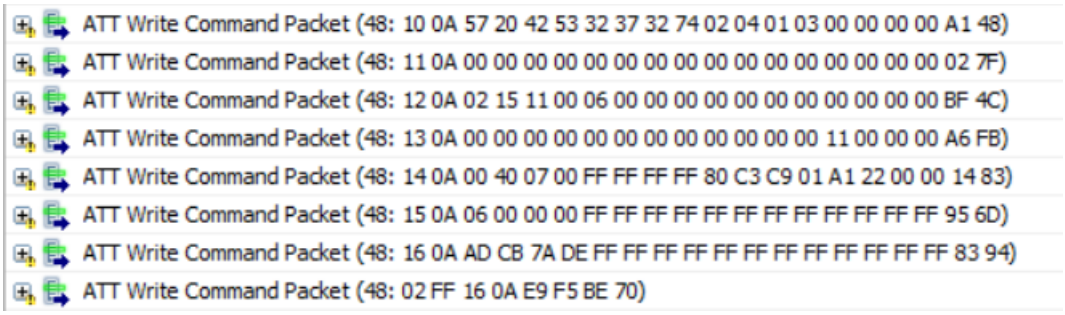


Figure 6.8: master OTA 数据 2

(8) firmware 数据发送完毕后，检查 BLE link layer 的数据是否已经完全发送出去（因为只有当 link layer 的数据被 slave ack 了，才会认为该数据发送成功）。若完全发送出去，master 发送一个 ota_end 命令，通知 slave 所有数据已发送完毕。

OTA end 的 packet 有效字节设为 6 个，前两个为 0xff02，中间的两个 bytes 为新的 firmware 最大的 adr_index 值（这个是为了让 slave 端再次确认没有丢掉最后一条或几条 OTA 数据），最后两个 bytes 为中间最大的 adr_index 值的取反，相当于一个简单的校验。OTA end 不需要 CRC 校验。

以上图所示的 bin 为例，最大的 adr_index 为 0x0a16，其取反值为 0xf5e9，最终的 OTA end 包如上图所示。

(9) 检查 master 端 link layer TX fifo 是否为空。若为空，说明之前所有的数据和命令都已成功发送出去，即 master 端的 OTA 任务已经全部完成。

CRC_16 计算函数见本文档后面的"附录 1：crc16 算法"。

按照前面所述，Slave 端在 OTA Attribute 中直接调用 otaWrite 和 otaRead 即可，master 端发送过来的 write command 命令，BLE 协议栈会自动解析并最终调用到 otaWrite 函数进行处理。

在 otaWrite 函数里对 packet 20 byte 的数据进行解析，首先判断是 OTA CMD 还是 OTA data，对 OTA cmd 进行相应的响应，对 OTA 数据进行 CRC 校验并烧写到 flash 对应位置。

slave 端 OTA 相关的操作为：

(1) 收到 OTA version 命令（OTA_FIRMWARE_VERSION 命令）：

master 要求获得 slave firmware 版本号，该 BLE SDK 收到这个命令时，不做处理，只是根据 user 是否注册了收到 version 的回调函数，判断是否触发回调函数。

在 ota.h 中看到注册该回调函数的接口为：

```
typedef void (*ota_versionCb_t)(void);
void blc_ota_registerOtaFirmwareVersionReqCb(ota_versionCb_t cb);
```

(2) 收到 OTA start 命令：

此时 slave 进入 OTA 模式。

若用户使用 bls_ota_registerStartCmdCb 函数注册了 OTA start 时的回调函数，则执行此函数，这个函数的目的是让用户在进入 OTA 模式后，修改一些参数状态等，比如将 PM 关掉（使得 OTA 数据传输更加稳定）。另外 slave 启动并维护一个 slave_adr_index，初值为-1，记录最近一次正确 OTA data 的 adr_index，用于判断整个 OTA 过程中是否有丢包。一旦丢包，认为 OTA 失败，退出 OTA，上报结果，MCU 重启，master 端由于收不到 slave 的 ack 包，也会由于 OTA 任务超时使得软件发现 OTA 失败。

注册 OTA start 的回调函数：

```
typedef void (*ota_startCb_t)(void);
void blc_ota_registerOtaStartCmdCb(ota_startCb_t cb);
```

user 需要注册这个回调，以便在 OTA start 的时候做一些操作，比如配置 LED 灯的特殊闪烁方式来指示 OTA 正在进行。

另外 slave 这端一旦收到 OTA start 开始 OTA 后，也会启动两个计时，一个是 OTA 整个过程完成的超时时间，目前 SDK 中默认是 30s。如果 30s 之内 OTA 还没有完成，就认为超时失败。实际 user 最后需要根据自己的 firmware 大小（越大越耗时）和 master 端 BLE 数据带宽（太窄的话会影响 OTA 速度）来修改这个默认的 30s，SDK 提供修改的接口为：

```
ble_sts_t blc_ota_setOtaProcessTimeout(int timeout_second);
```

该接口支持的 timeout 时间范围为 4 ~ 1000s，单位为 s。

(3) 收到有效的 OTA 数据：

这个范围的值表示具体的 OTA data。

每次 slave 收到一个 20 byte 的 OTA data packet，先看 adr_index 是否等于 slave_adr_index 的值加 1。若不等，说明丢包，OTA 失败；若相等，更新 slave_adr_index 的值。

然后对前 18 byte 的内容进行 CRC_16 的校验。若不匹配，OTA 失败；若匹配，则将 16 byte 的有效数据写到 flash 对应位置 ota_program_offset+adr_index*16 ~ ota_program_offset+adr_index*16 + 15。在写 flash 的过程中，如果出错，OTA 也失败。

(4) 收到 OTA end：

检查 OTA end 包中的 adr_max 和其取反校验值是否正确。若正确，则 adr_max 可以用来做 double check。double check 的时候，判断 slave 之前收到的 master 的数据 index 最大值与该包中的 adr_max 是否相等。若相等，认为 OTA 成功，若不等，认为丢掉了最后一笔或几笔数据，OTA 不完整。当 OTA 成功的时候，slave 将老的 firmware 所在地址的 flash 启动标志设为 0，将新的 firmware 所在地址的 flash 启动标志设为 0x4b，将 MCU reboot。

(5) slave 提供 OTA 状态的回调函数：

slave 端一旦启动 OTA，在 OTA 成功时会将 MCU reboot：

若成功，会在 reboot 前设置 flag 告诉 MCU 再次启动后运行 New_firmware；

若 OTA 失败，会将错误的新程序擦掉后重新启动，还是运行 Old_firmware。

在 MCU reboot 前，根据 user 是否注册了 OTA 状态回调函数，来决定是否触发该函数。

以下是相关 code：

```
enum{
    OTA_SUCCESS             = 0,            //success
    OTA_PACKET_LOSS,                        //lost one or more OTA PDU
    OTA_DATA_CRC_ERR,                       //packet PDU CRC err
    OTA_WRITE_FLASH_ERR,                    //write OTA data to flash ERR
    OTA_DATA_UNCOMPLETE,                    //lost last one or more OTA PDU
```

```
    OTA_TIMEOUT,                            //OTA flow total timeout
    OTA_FW_CHECK_ERR,                       //firmware CRC check error
    OTA_STEP_ERR,
};

typedef void (*ota_resIndicateCb_t)(int result);

void blc_ota_registerOtaResultIndicationCb (ota_resIndicateCb_t cb);
```

设置了回调函数后，回调函数的参数 result 的 6 个值如上面 enum 描述，第一个是 OTA 成功，其余是不同的失败原因。

由于回调函数执行完后，一定会触发 MCU reboot，实际代码中看到这个状态指示的结果时，并没有太多功能性用途。OTA 升级成功或失败均会触发该回调函数，user 可以通过该函数的结果返回参数来进行 debug，在 OTA 不成功时，可以读到上面的 result 后，将 MCU 用 while(1) 停住，来了解当前是何种原因导致的 OTA 失败。

6.3 OTA 安全性

6.3.1 OTA Service 数据安全

OTA Service 是一种 GATT service，OTA service 安全保护问题就是 BLE GATT service data 安全保护的问题，即数据不被非法访问。根据 BLE Spec 的设计，可以使用的方法包括以下：

(1) 开启 SMP，建议使用尽量高的 Security Level，实现的功能是：只有合法配对的设备，才有访问 OTA server 数据的权限。参考本文档 SMP 的介绍。

比如使用 Security Mode 1 Level 3，使用了 Authentication 和 MITM 的配对，可以有效控制产品 slave 设备和特定的 master 才能配对加密成功以及回连，攻击者无法和 slave 设备加密成功。对被保护的 GATT service data 的读和写加入相应的安全级别设置，攻击者就无法访问到这些数据了。如果使用 Mode 1 Level 4，Secure Connection + Authentication，安全级别更高了。

可能涉及到的 code 包括以下：

```
typedef enum {
    LE_Security_Mode_1_Level_1 = BIT(0),  No_Authentication_No_Encryption          = BIT(0),
 ↪  No_Security = BIT(0),
    LE_Security_Mode_1_Level_2 = BIT(1),  Unauthenticated_Pairing_with_Encryption  = BIT(1),
    LE_Security_Mode_1_Level_3 = BIT(2),  Authenticated_Pairing_with_Encryption    = BIT(2),
    LE_Security_Mode_1_Level_4 = BIT(3),
 ↪  Authenticated_LE_Secure_Connection_Pairing_with_Encryption = BIT(3),
}le_security_mode_level_t;

#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)
```

(2) 使用 whitelist。用户可以使用白名单，只和自己希望连接的 master 设备连接，也可以有效拦截攻击者的连接。

(3) 使用地址隐私保护，local device 和 peer device 使用 resolvable private address(RPA)，有效隐藏对方或我方的身份地址，使用连接更加安全。

6.3.2 OTA RF 传输数据完整性

由于 RF 是不稳定传输，需要一定的保护机制来确保 OTA 过程中 Firmware 的完整性和正确性。

参考前面的介绍可知，OTA master 需要提前将 Firmware 按照一定大小分成多个数据包，每个数据包前 2byte 是包序列号，从 0 开始加 1 递增。

6.3.2.1 OTA PDU CRC16 校验

参考本文前面介绍，在 LinkLayer 数据保护的基础上，OTA 协议上再增加一个 CRC16 校验，使得数据传输更加安全。

6.3.2.2 OTA PDU 序列号检查

OTA master 将 Firmware 拆成若干个 OTA PDU，每个 PDU 都有自己的包序列号。

为了便于说明，假设 Firmware size 为 50K，按照 OTA PDU 16Byte 进行拆分，PDU 数量为 50*1024/16=3200，那么序列号为 0 ~ 3199，即 0x0 ~ 0xC7F。

OTA 开始后，设置预期序列号为 0。每收到一笔 OTA 数据，使用预期序列号和实际序列号对比，只有二者相等才认为流程正确，同时更新预期序列号 +1；如果二者不等，认为失败，结束 OTA。这样的设计可以保证 OTA PDU 的连续性和唯一性。

在 OTA 结束时，可以在 OTA_END 包上可以读到 Firmware 最后一个 OTA PDU 的序列号 0xC7F，用这个序列号跟实际接收的最大序列号进行对比，可以确定 OTA PDU 是否存在尾巴丢失情况。如果实际接收的最大序列号为 0xC7E，则说明 master 漏传了最后一个包，此时 OTA 会失败。

以上设计结合在一起，可以确保 OTA master 端对 Firmware 的正确拆分，并且每一个 OTA PDU 都有效发出。

6.3.2.3 OTA 期间需要关闭低功耗

为了保证 OTA 过程不受低功耗的影响，OTA 开始后需要调用以下 API 关闭低功耗。

```
bls_pm_setSuspendMask(SUSPEND_DISABLE)
```

7 Flash

7.1 Flash 地址分配

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），理论上同一种类的信息需要存储在一个 sector 里面，不同种类的信息需要在不同的 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。系统相关信息 (Customed Value，MAC address，Pair&Sec Info) 默认位置会自适应的根据 flash 真实大小，偏移到 flash 的较为偏后的位置。

如下图是 128K/512K Flash 中各种信息的地址分配，以默认 OTA Firmware 最大 size 不超过 128K 为例来说明，如果用户修改了 OTA Firmware size，则相应地址发生变化，用户可以自行分析。

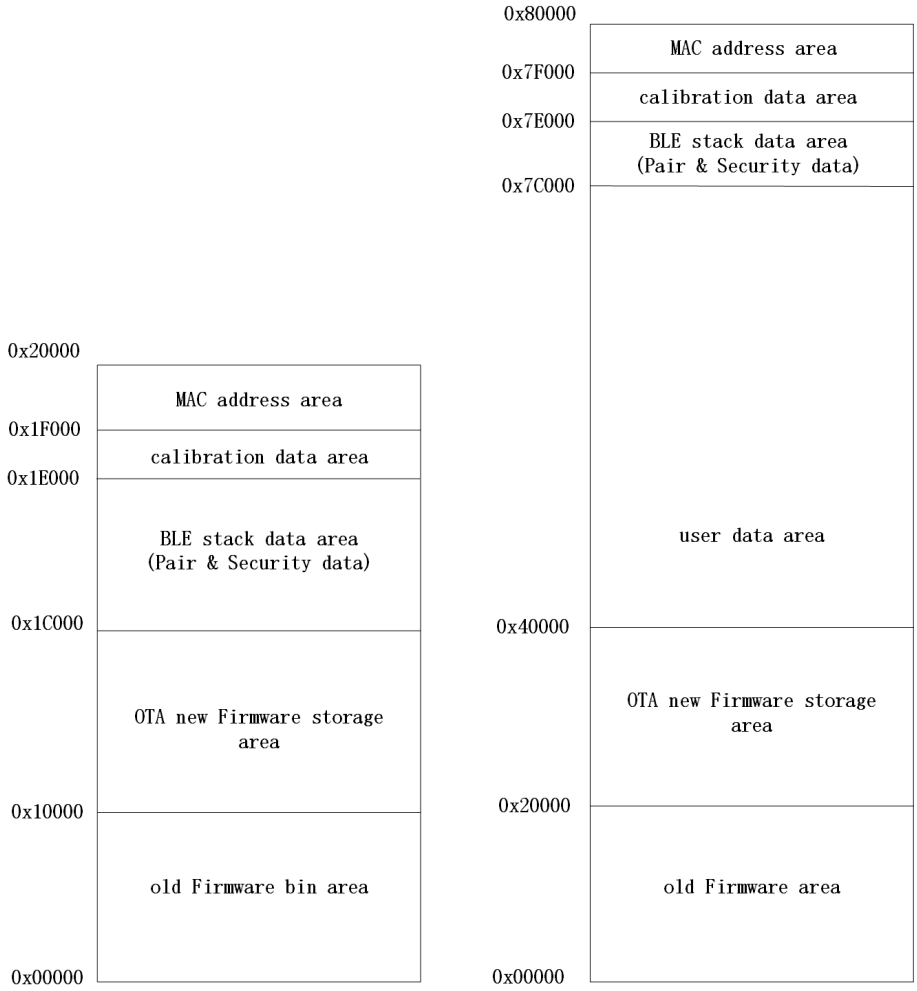


Figure 7.1: 128K/512K FLASH 地址分配

上图所示，其中所有的地址分配都给 user 提供了对应的修改接口，user 可以根据自己需要去规划地址分配。user 可以根据相对应的 Flash size 定义以下宏定义。

```
#define     FLASH_SIZE_OPTION_128K                          0x20000
#define     FLASH_SIZE_OPTION_512K                          0x80000

#define     FLASH_SIZE_OPTION                       FLASH_SIZE_OPTION_128K
```

(1) 当使用 512K Flash 时 0x7F000~0x80000 这个 sector 存储 MAC 地址，当使用 128K Flash 时为 0x1F000~0x20000，实际上 MAC address 6 个 bytes 存储在 0x7F000~0x7F005(0x1F000~0x1F005)，当使用 512K 时，高 byte 的地址存放在 0x7F005，低 byte 地址存放在 0x7F000。比如 FLASH 0x7F000 到 0x7F005 的内容依次为 0x11 0x22 0x33 0x44 0x55 0x66，那么 MAC address 为 0x665544332211。泰凌的量产治具系统会将实际产品的 MAC 地址烧写到 0x7F000（0x1F000）地址，和 SDK 相对应。如果 user 需要修改这个地址，请确保治具系统烧写的地址也作了相应的修改。SDK 中在 user_init 函数里会从 FLASH 的 CFG_ADR_MAC 读取 MAC 地址，这个宏在 stack/ble/blt_config.h 里面修改即可。

```
/*************************** 512 K Flash ***************************/
#ifndef     CFG_ADR_MAC_512K_FLASH
#define     CFG_ADR_MAC_512K_FLASH                          0x7F000
#endif
/*************************** 128 K Flash ****************************/
#ifndef     CFG_ADR_MAC_128K_FLASH
#define     CFG_ADR_MAC_128K_FLASH                          0x1F000
#endif
```

(2) 对于 512K Flash 0x7E000~0x7EFFF 这个 sector 存储 telink MCU 需要校准定制的信息，128K Flash 时为 0x1E000~0x1EFFF。只有这部分的信息不遵循"不同类信息放在不同 sector"的原则，将这个 sector 4096 bytes 按照每 64 bytes 划分为不同的单元，每个单元存储一类校准信息。校准信息可以放在同一个 sector，是因为校准信息在治具烧录的过程中烧到对应地址，实际 firmware 在运行时只能读这些校准信息，而不允许去写，更不允许去擦除。

校准区信息的细节，用户可参考《Telink_IC_Flash Customize Address_Spec》。以相对校准区起始地址的偏移地址对校准信息进行说明。比如偏移地址 0x00 指的是 0x7E000 或 0x1E000。

a) 偏移地址 0x00，1 byte，存储 BLE RF 频偏校准值。

b) 偏移地址 0x40，4 byte，存储 TP 值校准。B80 IC 不需要 TP 校准，这里忽略。

c) 偏移地址 0xC0，存储 ADC Vref 校准值。

d) 偏移地址 0x180，16 byte，存储 Firmware 数字签名，用于防止客户 Firmware 被盗用。

e) 其他，保留。

(3) 512K Flash 0x7C000 ~ 0x7DFFF 这两个 sector 被 BLE 协议栈系统占用，对于 128K Flash 为 0x1C000~0x1DFFF，用来存储配对和加密信息。user 也可以修改这两个 sector 的位置，size 固定为两个 sector 8K，无法修改，可以调用下面函数修改配对加密信息存储的起始地址：

```
void    bls_smp_configpairingSecurityInfoStorageAddr (int addr);
```

(4) 对于 512K Flash0x00000 ~ 0x40000 这段区域为程序空间，128K Flash 时为 0x00000~0x0FFFF。以 512K Flash 举例，0x00000 ~ 0x1FFFF 共 128K 为 Firmware 存储空间；0x20000 ~ 0x3FFFF 128K 为 OTA 更新时存储新 Firmware 的空间，即支持最大 Firmware 空间为 128K。

(5) 剩余的 FLASH 空间全部作为 user 的数据存储空间。

7.2 Flash 操作

Flash 空间的读写操作使用 flash_read_page 和 flash_write_page 函数指针，flash 的擦除使用 flash_erase_sector 函数。

(1) Flash 读写操作

flash 读写操作使用 flash_read_page 和 flash_write_page 函数指针来实现，默认指向 flash_read_data 和 flash_page_program 函数。

```
void flash_read_data(unsigned long addr, unsigned long len, unsigned char *buf);
void flash_page_program(unsigned long addr, unsigned long len, unsigned char *buf);
```

如果需要更改 Flash 读写函数可以通过以下 API 更改。

```
void flash_change_rw_func(flash_hander_t read, flash_hander_t write);
```

flash_read_page 读取 flash 上的内容:

```
u8 data[6] = {0 };
flash_read_page(0x11000, 6, data); //读 flash 0x11000 开始的 6 个 byte 到 data 数组。
```

flash_write_page 对 flash 进行写操作:

```
u8 data[6] = {0x11,0x22,0x33,0x44,0x55,0x66 };
flash_write_page(0x12000, 6, data); //向 flash 0x12000 开始的 6 个 byte 写入 0x665544332211。
```

flash_write_page 函数是对 page 的操作，flash 里面一个 page 为 256 byte，这个函数操作的地址大小最大为 256 byte，不能跨越两个不同 page 范围。

当被操作的地址是一个 page 的首地址时，最大地址为 256 byte，flash_write_page(0x12000, 256 , data) 操作正确，而 flash_write_page(0x12000, 257 , data) 错误，因为最后一个地址不属于 0x12000 所在的 page 了，写操作会失败。

当被操作的地址不是一个 page 的首地址时，更要注意不能出现跨 page 的问题，如 flash_write_page(0x120f0, 20 , data) 就错了，前 16 个地址在 0x12000 这个 page，而后 4 个地址在 0x12100 这个 page。

flash_read_page 不存在上面说的跨 page 的问题，可以一次性读取超过 256 byte 的数据。

注意：

- user 在使用 flash_write_page 函数时，最多只能一次写 16 个 byte, 多了会导致 BLE 中断异常。
- 该限制的原理，请结合"Flash API 对 BLE timing 的影响"部分的介绍。

(2) flash 擦除操作

使用 flash_erase_sector 函数来擦除 flash。

```
void  flash_erase_sector(u32 addr);
```

一个 sector 为 4096 byte，如 0x13000 ~ 0x13fff 是一个完整的 sector。

addr 必须是一个 sector 的首地址，该函数每次擦除整个 sector。

擦除一个 sector 的时间会比较长，16M 系统时钟时，大约需要 30 ~ 100ms 甚至更长时间。

(3) flash 读写和擦除操作对系统中断的影响

上面介绍的三个 flash 操作函数 flash_read_page、flash_write_page、flash_erase_sector 在执行时，都必须先将系统中断关掉 irq_disable()，操作结束后再恢复中断 irq_restore()，其目的是为了保证 flash MSPI 时序操作的完整性和连续性，同时防止中断里又有 flash 操作调用 MSPI 总线而造成硬件资源的重入。

这个 BLE SDK RF 收发包的时序全部由中断来控制的，flash 操作关闭系统中断造成的后果是 BLE 收发包的时序会被破坏，得不到及时响应。

其中 flash_read_page 函数的执行时间不是太长，对 BLE 的中断影响很小，当使用 flash_write_page 时建议在 BLE 连接状态时最长一次写 16Bytes，如果过长的话可能会对 BLE 时序造成影响。所以强烈建议用户在 main_loop 里 BLE 连接状态时，不要连续读写太长的地址。

flash_erase_sector 函数的执行时间为几十到几百个 ms，所以在主程序的 main_loop 里，一旦进入 BLE 连接状态，不允许去调用 flash_erase_sector 函数，否则会破坏 BLE 收发包的时间点，造成连接断开。如果是无法避免在 BLE 连接的时候需要擦除 flash，请根据本文档后面介绍的 Conn state Slave role 时序保护实现方法来操作。

(4) 读 flash 可以使用指针访问来实现

BLE SDK 的 firmware 存储在 flash 上，程序运行时，只是将 flash 前一部分的代码作为常驻内存代码放在 ram 上执行，剩余的绝大部分代码根据程序的局部性原理，在需要的时候从 flash 读到 ram 高速缓存 cache 区域(简称 cache)。MCU 通过自动控制内部 MSPI 硬件模块，读取 flash 上的内容。

可以使用指针的形式读取 flash 上的内容，指针形式读 flash 的原理是 MCU 系统总线访问数据时，当发现数据地址不在常驻内存 ramcode 上，系统总线就会自动切换到 MSPI，通过 MSCN、MCLK、MSDI 和 MSDO 四根线去操作 spi 的时序来获得读取 flash 数据。

以下列出 3 种示例：

```
u16  x = *(volatile u16*)0x10000; //读 flash 0x10000 两个 byte
u8 data[16];
memcpy(data, 0x20000, 16);     //读 flash 0x20000 16 个 byte copy 到 data
if(!memcmp(data, 0x30000, 16)){ //读 flash 0x30000 16 个 byte 和 data 比较
    ......
}
```

user_init 里面读取 flash 上的校准值并设定到对应的 register 时，都是采用指针访问 flash 的方式实现的，请参考 SDK 里函数。

```
static inline void blc_app_loadCustomizedParameters(void);
```

读 flash 可以使用指针形式，但是不能使用指针写 flash（写 flash 只能通过 flash_write_page 来实现）。

需要注意指针读 flash 存在一个容易出问题的地方：只要是通过 MCU 系统总线读到的数据，MCU 都会将这些数据缓存在 cache 里，如果 cache 里这个数据没有被其他内容覆盖时，又有新的访问该数据的请求发生，此时 MCU 会直接用 cache 里缓存的内容作为结果。如果 user 的代码出现下面这种情况：

```
u8  result;
result = *(volatile u16*)0x40000;    //指针读取 flash
u8 data = 0x5A;
flash_write_page(0x40000, 1, &data );
result = *(volatile u16*)0x40000;    //指针读取 flash
if(result != 0x5A){  ..... }
```

flash 地址 0x40000 处本来是 0xff，第 1 次读到的 result 为 0xff，然后写入 0x5A，理论上第 2 次读到的值是 0x5A，但实际程序给出的结果还是 0xff，是从 cache 里拿到的第一次缓存的结果。

注意：

> 如果出现这种多次读同一个地址且这个地址的值会被改写时，千万不要用指针的形式，使用 API flash_read_page 来实现最安全，这个函数读到的结果不会从 cache 里拿之前缓存的值。

改成如下实现才正确：

```
u8  result;
flash_read_page(0x40000, 1, &result );  //API 读取 flash
u8 data = 0x5A;
flash_write_page(0x40000, 1, &data );
```

位置会自适应的根据 flash 真实大小，偏移到 flash 的较为偏后的位置。

7.3 Flash 操作的保护

由于 write flash 和 erase flash 的过程中，需要将地址和数据通过 SPI 总线传递给 Flash，SPI 总线上的电平稳定性非常重要，这些关键的数据一旦错误就会造成不可逆的后果，比如将 firmware 写错或误擦都会造成 firmware 无法再运行，OTA 功能也会失效。

在 Telink 芯片多年的量产经验中，出现过 Flash 在不稳定条件下操作导致的错误情况。不稳定的条件主要包括电源电压偏低、电源纹波过大、系统上其他模块间歇性的耗电导致电源抖动等等。为了避免后续的产品出现类似的操作风险，这里我们介绍一些相关的 Flash 操作保护的方法，客户认真阅读后需要尽量考虑这些问题，增加更多的安全保护机制，才确保产品为稳定性。

7.3.1 低电压检测保护

结合低电保护章节的介绍，需要考虑在所有的 Flash write 和 erase 操作之前都做电压检测，避免出现过低电压下操作 Flash 的情况。另外为了保证系统一直在一个安全的电压下工作，在 main_loop 中也建议定时做低压检测，以保证系统的正常运行。

注意：

> 关于 flash 低压保护，以下多处出现 2.0V、2.2V 等阈值，强调下这些数值只是示例、参考值。客户要根据实际情况评估修改这些阈值，比如单层板、电源波动大等因素，都要酌情提高安全阈值。

以 SDK demo 中的低压检测为例：

Step 1：首先在上电或从 deepsleep 唤醒时，在调用 Flash 函数之前，都要进行一次低压检测，以防止低电压造成的 flash 问题：

```
void user_init_normal(void)
{

    ......

    #if (BATT_CHECK_ENABLE)
        u8 battery_check_returnVaule = 0;
        if(analog_read(USED_DEEP_ANA_REG) & LOW_BATT_FLG){
            battery_check_returnVaule = app_battery_power_check(VBAT_ALRAM_THRES_MV + 200);
        }
        else{
            battery_check_returnVaule = app_battery_power_check(VBAT_ALRAM_THRES_MV);
        }
        if(battery_check_returnVaule){
            ......
        }
        else{
            ......
        }
    #endif

    ......

}
```

Step 2：在 main_loop 中，每隔一定 500ms 需要进行一次低压检测：

```
if(battery_get_detect_enable() && clock_time_exceed(lowBattDet_tick, 500000) ){
    lowBattDet_tick = clock_time();
    u8 battery_check_returnVaule;
    if(analog_read(USED_DEEP_ANA_REG) & LOW_BATT_FLG){
        battery_check_returnVaule=app_battery_power_check(VBAT_ALRAM_THRES_MV + 200);
    }
    else{
        battery_check_returnVaule=app_battery_power_check(VBAT_ALRAM_THRES_MV);
    }
    if(battery_check_returnVaule){
        ......
    }
```

```
    else{
        ......
    }
}
```

考虑 mcu 的工作电压及 flash 的工作电压情况，因此 Demo 设置在 2.0V 以下芯片直接进入 deepsleep，并且一旦芯片检测过低于 2.0V，需要等到电压升至 2.2V，芯片才会恢复正常运行状况。这么设计考虑以下几点：

- 2.0V 时当有其他模块被操作可能拉低电压造成 flash 无法正常工作，因此在 2.0V 以下需要进入 deepsleep，保证芯片不再运行相关模块；
- 当有低电压情况时，需恢复至 2.2V 才能使其他功能正常，这是为了保证电源电压确认在被充电且已有一定电量，此时开始恢复功能可以较为安全。

以上是 SDK Demo 中的定时检测电压及管理方式，用户可以参考来进行设计。

注意：

> 关于 flash 低压保护，以上出现的阈值，只是参考值。客户要根据实际情况评估修改阈值，比如单层板、电源波动大等因素，都要酌情提高安全阈值。

7.3.2 Flash lock 保护

除了上述的定时电压检测与管理方案，强烈建议客户做 Flash 的擦写保护。这是由于在某些情况下，即便做了低压检测结果为安全, 但在检测之后应用层各个模块运行也小概率风险造成 Flash 电源电压的拉低，而导致 Flash 真正操作时其电源电压不满足条件而产生 Flash 的内容被篡改。因此建议客户在程序启动后便进行 Flash 的擦写保护，这样即使有误操作产生，Flash 的内容也会更有保障。

一般建议客户只对程序部分写保护 (Flash 前部分)，这样其余的 Flash 地址仍可以用于用户层的数据存储。这里以 SDK Sample 工程为例讲述如何计算要保护大小与保护的方法。

7.3.2.1 初始化写保护

(1) 计算保护大小：在初始化前，计算要写保护的 flash 地址大小。

(2) 调用 flash_read_mid 判断 flash 类型，根据结果调用相关函数，并根据要保护大小传入对应参数。mid 值对应的相关函数可以在 drivers/flash 目录下找到。

7.3.2.2 OTA 过程中的保护操作

在 OTA 中，由于需要对 flash 进行擦写操作，因此如果上电有写保护的操作，在 OTA 过程中需要将其解锁保护。可以在 OTA_START 回调中进行 flash 解锁保护，步骤如下：

Step 1：首先在初始化函数中如以下方法注册回调函数；

```
blc_ota_registerOtaStartCmdCb (&flash_ota_start);
```

Step 2：在回调函数中，根据之前上电拿到的 flash 类型，调用对应函数进行解锁保护：

```
void flash_ota_start(void)
{
switch(flash_lock_mid)
{
case mid 值:
    对应的 unlock 函数;
    break;

        ...
}
}
```

由于 OTA 结束后，无论成功或者失败，都会重新运行程序，因此在程序开始，由上一节将的 flash_lock 方法再次将程序写保护，形成闭环保证应用的安全性。

7.4 内置 Flash 介绍

7.4.1 Flash 访问时序对 BLE 时序的影响

7.4.1.1 Flash 访问时序

(1) Flash 操作基本时序

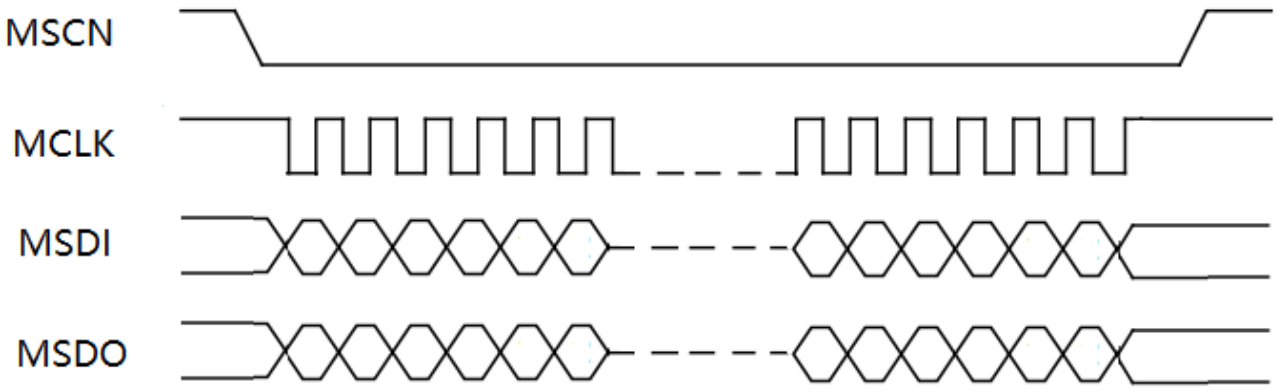


Figure 7.2: Flash 基本操作时序

上图所示为一个典型的 MCU 访问 Flash 时序，MSCN 拉低期间，在 MCLK 控制下，通过 MSDI 和 MSDO 的电平状态变化，完成与 Flash 的数据交互。

Flash 访问时序是 Flash 操作基本时序，MSCN 拉低期间为数据交互，拉高之后结束。所有的 Flash 功能都是以它为基础，复杂的 Flash 功能可以拆分成若干个 Flash 操作基本时序。

每一个 Flash 操作基本时序都是相对独立的，必须等一个操作时序完成之后，才能进行下一轮的操作。

(2) MCU 硬件访问 Flash

Firmware 存储在 Flash 中，MCU 执行程序需要提前从 Flash 中读取指令和数据，结合 2.1.2.1 的介绍可知需要读取的内容为 text 段和"read only data"段。MCU 运行过程中实时读取 Flash 上的指令，所以会不停的启动 Flash 操作基本时序，这个过程是由 MCU 硬件自动控制的，软件不参与。

main_loop 程序运行过程中，如果突然发生中断，进入 irq_handler。即使 main_loop 和 irq_handler 中的程序都在 text 段中，不会出现 Flash 时序冲突，因为都是由 MCU 硬件来完成的，它会做好相关的仲裁和管控工作。

(3) 软件访问 Flash

MCU 硬件访问 Flash 只解决读程序指令和"read only data"问题。如果需要对 Flash 手动进行读写擦等操作，使用 flash driver 中的 flash_read_page、flash_write_page、flash_erase_sector 等 API。查看这几个 API 的具体实现，可以看出是由软件来管控 Flash 操作基本时序，先拉低 MSCN，然后读写数据，最后拉高 MSCN。

(4) Flash 访问时序冲突以及解决方法

由于 Flash 操作基本时序是一个不可分割和破坏的过程，当软件和 MCU 硬件同时访问 Flash 时，由于软件和 MCU 硬件不具备协调和仲裁机制，有可能出现时序冲突。

出现这个时序冲突的场景为：软件在 main_loop 中调用 flash_read_page、flash_write_page、flash_erase_sector 等 API，MSCN 拉低后正在进行数据读写时，发生了中断，irq_handler 中有一些指令存储在 text 段中，MCU 硬件也启动一个新的 Flash 操作基本时序，这个时序就和前面 main_loop 中的时序冲突，造成 MCU 死机等错误。

如下图所示，Software 访问 Flash 结束时，发生中断并响应，MCU 硬件开始访问 Flash。此时 Flash 访问的结果必然会出错。

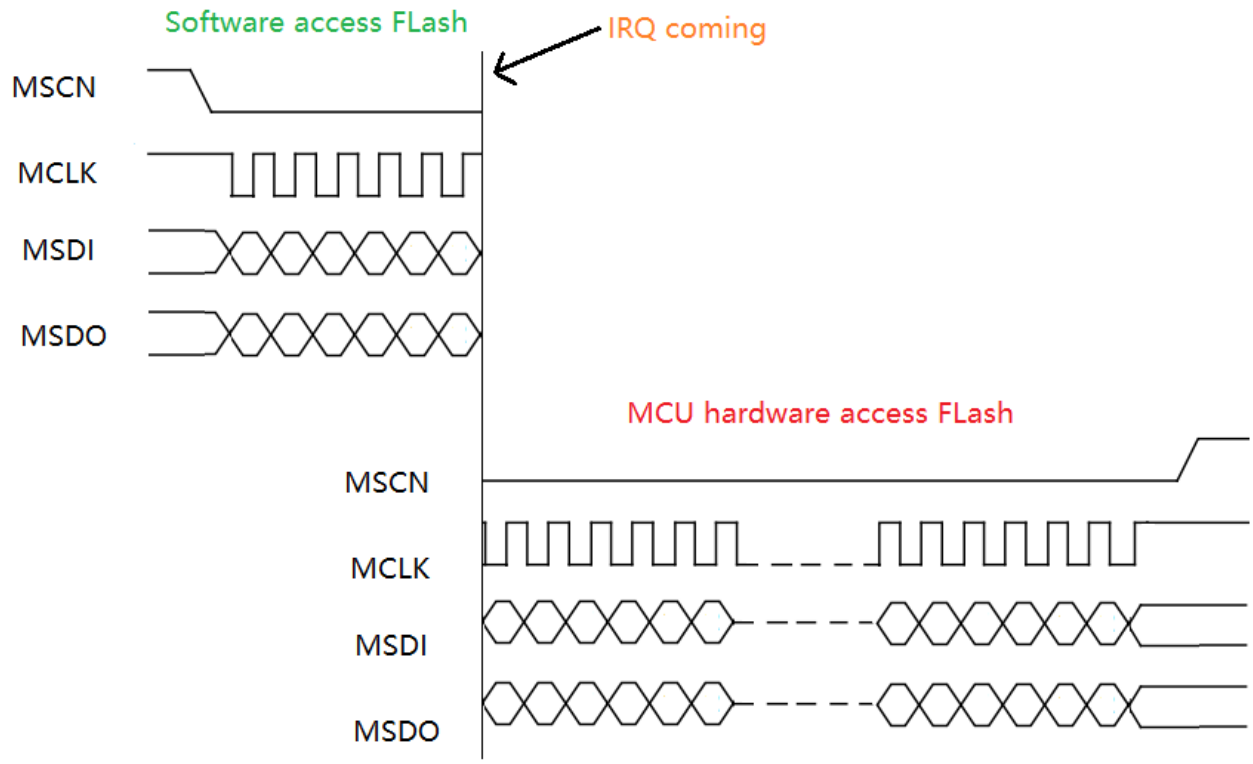


Figure 7.3: 中断导致的 Flash 时序冲突

分析时序冲突的几个必须同时满足的条件，可以得出解决该冲突的方法包括以下几种：

a) main_loop 中不要出现任何软件操作 Flash 时序的 API。这个方法不可行，SDK 和应用上都会出现 flash_write_page 等 API 的使用。

b) irq_handler 函数中所有的程序都提前存储到 ramcode，不依赖任何 text 段和"read_only_data" 段。这个方法也不太好。受限于 8208 芯片的 Sram size，如果所有的中断的 code 都存储到 ramcode，Sram 资源不够用。另外对于 user 做这种限制不容易管控，无法保证 user 中断 code 写的那么严密。

c) 软件操作 Flash 时序的几个 API 中，加保护，关闭中断，不让 irq_handler 响应，Flash 访问结束后，重新恢复中断。

目前 Telink BLE SDK 采用了方法 3，Flash API 中关中断保护。如下 code 所示（中间省略了若干 code），使用 irq_disable 关中断，irq_restore 恢复中断。

```
void flash_mspi_write_ram(unsigned char cmd, unsigned long addr, unsigned char addr_en, unsigned
 ↪  char *data, unsigned long data_len)
{
    unsigned char r = irq_disable();

    ..... //flash access

    irq_restore(r);
}
```

下图是关中断保护 Flash 访问时序的原理示意图。软件访问 Flash 时将中断关闭，中间发生了中断但是不能立刻响应（中断等待），等到软件访问 Flash 时序全部正确结束后，恢复开启中断，此时中断立刻响应，再由 MCU 硬件访问 Flash。

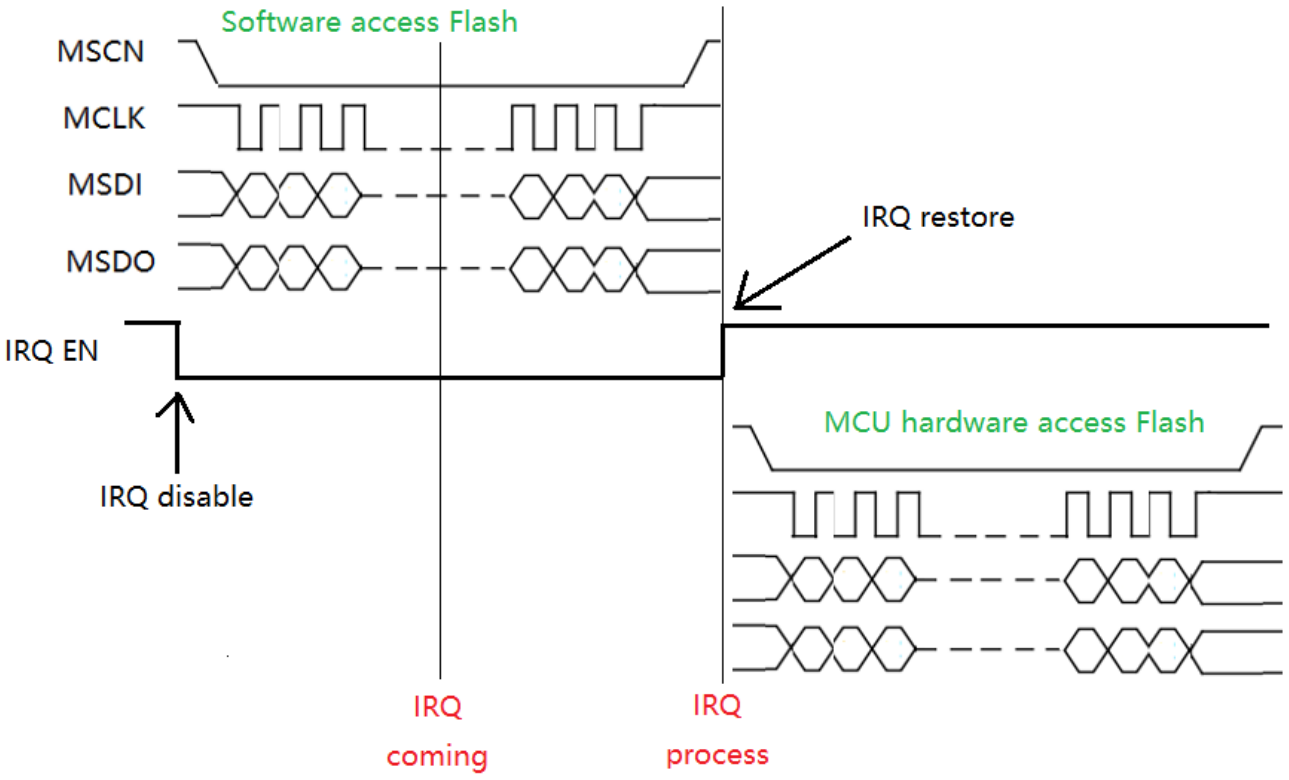


Figure 7.4: 正确的的中断处理与 flash 操作

7.4.1.2 Flash API 对 BLE timing 的影响

前面介绍了使用 Flash API 关中断保护来解决软件和硬件 MCU 访问 Flash 时序冲突的问题。由于关中断会使得所有的中断无法实时响应，排队等待中断恢复后延时执行，需要考虑被延时时间可能带来的副作用。

(1) 关中断对 BLE timing 的影响

结合 BLE timing 的特点来介绍。这个 SDK 中 BLE 连接态的 BTX、BRX 状态机都是由中断任务来完成的。BTX 和 BRX 是类似的实现，以 slave role 的 BRX 为例来说明。

BRX timing 的处理比较复杂，以 BLE slave BRX 出现 more data 时 RX IRQ 的处理为例，如下图所示。SDK 设计中要求软件对每一个 RX IRQ 都要响应，可以被延迟响应，但不能丢掉。如果某个 RX IRQ 被丢掉，触发这个 RX IRQ 的 RX packet 也会丢掉，造成 Linklayer 丢包的错误。

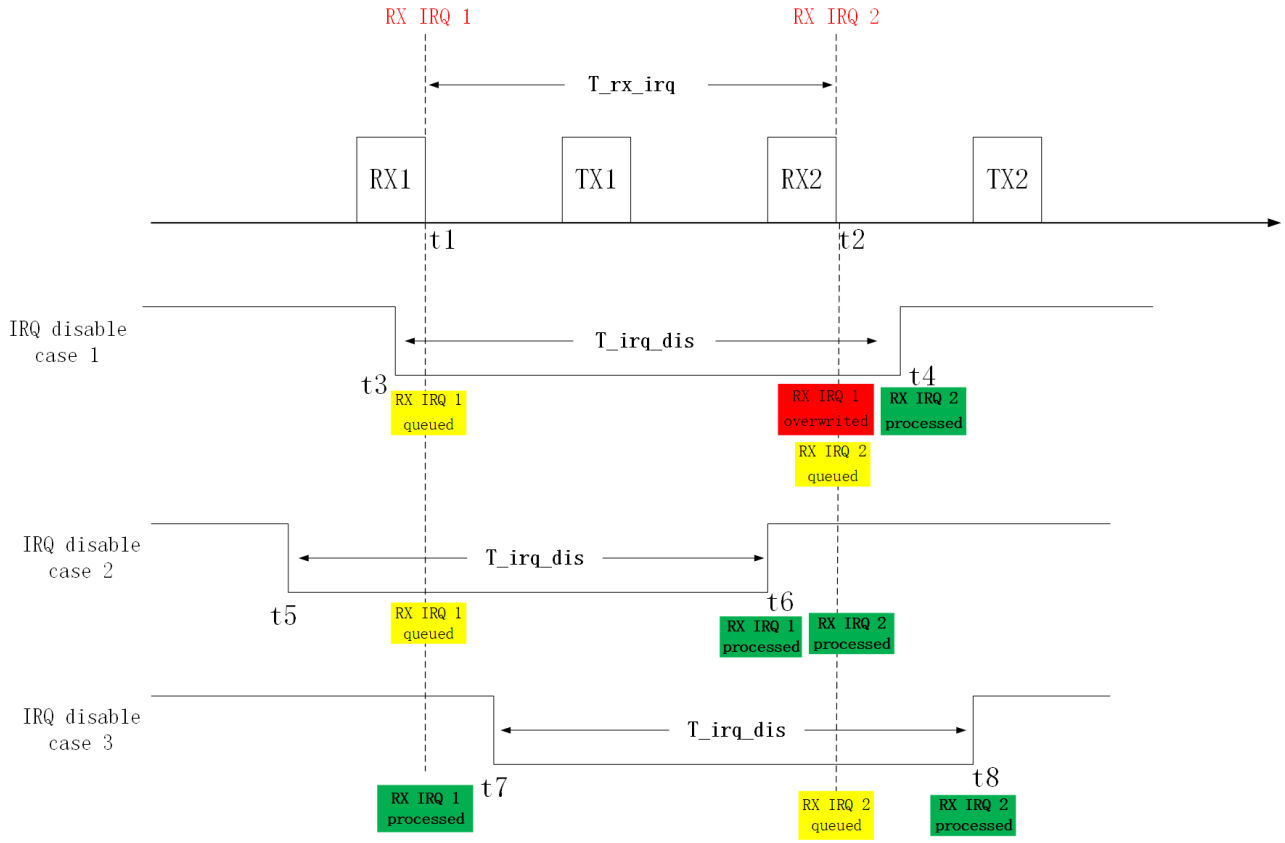


Figure 7.5: Flash 操作对 Link Layer 风险

图中 RX1 在 t1 触发 RX IRQ 1，RX2 在 t2 触发 RX IRQ 2。如果没有关中断发生，中断会在 t1 和 t2 实时响应，软件正确处理 RX packet。

t1 和 t2 时间差为 T_rx_irq，关中断持续时间为 T_irq_dis，T_irq_dis > T_rx_irq。

三种情况 IRQ disable case 1、IRQ disable case 和 IRQ disable case 3 的关中断持续时间都是 T_irq_dis，但关中断的起点和 t1 的相对时间不一样。

IRQ disable case 1，t3 关中断，t4 恢复中断。t3 < t1；t4 > t2。RX IRQ 1 在 t1 无法响应，中断排队等待。RX IRQ 2 在 t2 触发，覆盖 RX IRQ1（因为中断等待队列里只能有一个 RX IRQ），RX IRQ 2 排队等待，在 t4 被正确执行。RX IRQ1 对应的 RX1 丢失，如果 RX1 是一个有效的数据包，Linklayer 就会出错。

IRQ disable case 2 和 IRQ disable case 3，RX IRQ 1 和 RX IRQ 2 虽然被延迟执行，但没有丢掉，不会发生错误。

由以上的例子分析可以得到一个重要结论：

当中断关闭持续时间大于某个安全阈值，可能会发生 Linklayer 错误的风险。

这个安全阈值，跟 SDK 中 Linklayer 时序设计、BLE Spec 时序特点都相关，比例子中的 T_rx_irq 复杂得多。具体细节不详细介绍，这里直接给出安全阈值是 220us。

同样是关中断持续时间 T_irq_dis，上面例子中 IRQ disable case 2 和 IRQ disable case 3，由于关中断发生的时间点不一样，RX IRQ 1 或 RX IRQ2 被延时响应，不会发生 RX IRQ 2 覆盖 RX IRQ1 导致的丢包。即便是 IRQ disable case 1，如果 RX1 和 RX2 是无关紧要的空包，发生丢包也不会造成任何错误。

中断关闭持续时间大于 220us 时，不是一定会出现错误，必须多个条件同时满足，才有可能触发错误，这些条件包括：关中断的时间较长、关中断的时间点和 RX IRQ 发生的时间点符合某种特定的关系、BTX 或 BRX 中出现 more data；连续触发 RX IRQ 的两个 RX packet 都是有效数据包而不是空包等等。所以最终结论是：

中断关闭持续时间大于 220us 时，会出现 linlayer 出错的风险，概率非常低。

BLE SDK Linklayer 的设计，以零风险为目标，即中断关闭持续时间永远要小于 220us 的情况，不给任何出错的机会。

这里额外介绍上面例子 RX packet 丢失的问题。在 Telink BLE SDK 使用生产中，经常遇到客户反馈碰到这个问题：在加密开启的前提下，看到 device 发送一个 reason 为 0x3D（MIC_FAILURE）的 terminate 包，导致断连。

以上分析可知，中断关闭时间过长会导致 RX IRQ 被延迟太长时间进而被覆盖，最终丢包。但 SDK 会正确处理好中断关闭时间的问题，文档后面会详细介绍。更有可能的原因是 user 用到了其他的中断（比如 Uart、USB 等），这些中断响应时的软件执行时间如果过长，跟中断关闭的效果时一样的，也会让 RX IRQ 延迟。这里我们限定一个 user 中断执行的最大安全时间为 100us。

(2) Flash API 关中断保护对 BLE timing 的影响

为了规避软件访问 Flash 和 MCU 硬件访问 Flash 的时序冲突，Flash API 使用了关中断的方法。当中断关闭持续时间大于 220us 时，Linklayer 可能发生出错的风险。为了解决这二者的矛盾，需要关注 Flash API 关中断的最大时间。

受影响的 BLE timing 是 connection state slave role 和 master role。系统初始化和 mainloop 中的 Adver-tiisng state 不受影响。在 mainloop connection state 中，主要关注以下三个 Flash API：flash_read_page、flash_write_page、flash_erase_sector。其他 Flash API 一般不使用或者在初始化的时候才会用到。

a) flash_read_page

经测试验证，flash_read_page 一次性读取的 byte 数量不超过 64 时，时间非常安全，在 220us 以内。超过这个值后会有一定的风险。

强烈建议 user 使用 flash_read_page 读 Flash 时最多读 64 byte，如果超过 64 byte，需要拆成多次调用 flash_read_page 来实现。

b) flash_erase_sector

flash_erase_sector 的时间一般在 10ms ~ 100ms 这个量级，远远超过 220us。所以这个 SDK 要求 user 在 BLE connection state 不要调用 flash_erase_sector。直接调用这个 API，connection 一定会出错。

我们建议 user 使用其他方式来取代 flash_erase_sector 的设计。比如一些应用是为了反复更新在 Flash 上存储的一些关键信息，设计上可以考虑选取一块较大的区域，使用 flash_write_page 不断往后延伸的方法。

BLE slave 应用，对于无法避免的 flash_erase_sector，如果只是偶尔会发生，可以使用 Conn state Slave role 时序保护机制来规避，请参考本文档的详细介绍。

注意，由于时序保护机制非常复杂，对于高频率的 flash_erase_sector，不建议这么用，无法保证在 BLE slave 连接态时反复连接调用这套机制的稳定性。建议 user 尽量从设计上去避开这种情况。

c) flash_write_page

flash_write_page 时间收到多个关键因素的影响，包括：Flash 种类、Flash 工艺、write byte number、高低温等。下面从内置 Flash 的几个种类来详细说明。

7.4.2 内置 Flash API 的使用

根据上一节的介绍，Flash API 中 flash_write_page 跟内置 Flash 的种类相关的。本节结合 8208 已经支持的内置 Flash 详细介绍。

7.4.2.1 GD Flash

GD Flash 属于 ETOX 工艺。

flash_write_page 的消耗时间跟参数 len（即一次性写入的字节数量）相关，接近于一个正比的关系。经过 Telink 内部的详细测试和分析，发现字节数量小于等于 16 时，写入时间能够稳定在 220us 以内；字节数量如果超过 16，会有风险。

对于 GD Flash，要求 flash_write_page 写入字节数量最大值为 16。如果超过 16，比如 32，可以拆成两次写 16 byte。

在 SDK 设计上，两处涉及到 flash_write_page 的地方，一是 SMP 存储配置信息，使用的是每次写入 16 byte；二是 OTA 写入新的 firmware 时，也使用了每次写入 16 byte。OTA 长包设计上，比如每包 240 byte 有效数据，是拆成了 15 次写入（16*15=240）。

强烈建议客户使用 flash_write_page 每次最多写入 16 byte，否则会有跟 BLE timing 冲突的风险。

8 按键扫描

Telink 提供了一套基于行列式扫描的 keyscan 架构，用于按键扫描处理，user 可以直接使用这部分的 code，也可以自己去实现。

8.1 键盘矩阵

如下图所示，这是一个 2*2 的 Key matrix（键盘矩阵）。Row0，Row1 是 2 个 drive pin（驱动管脚），用来输出驱动电平；CoL0，CoL1 是 2 个 scan pin（扫描管脚），用来扫描当前列上是否有按键被按下。

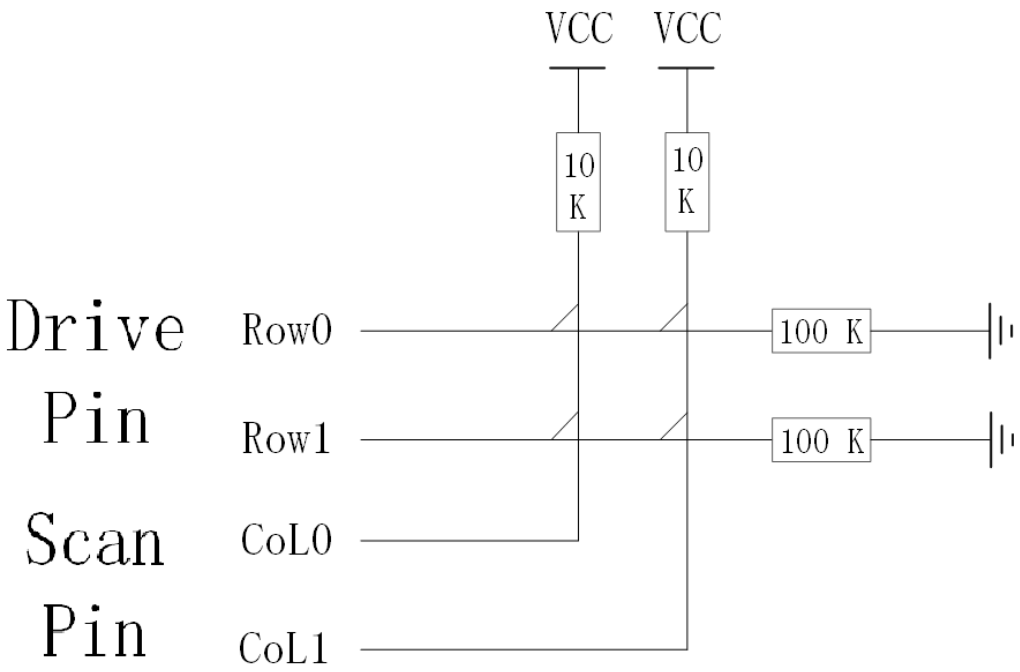


Figure 8.1: 行列式键盘结构

Telink EVK 板上为 2*2 键盘矩阵。在实际产品应用中可能需要更多的按键，请用户自行添加设计，下面以 Telink 提供板载 2*2 矩阵键盘为例来进行说明。结合上图，对 app_config.h 中 keyscan 相关的配置进行详细说明如下。

根据实际的硬件电路，EVK 板上 Row0，Row1 为 GPIO_PF0, GPIO_PF1。CoL0，CoL1 为 GPIO_PA0, GPIO_PD4。

定义 drive pin 数组和 scan pin 数组：

```
#define KB_DRIVE_PINS   {GPIO_PF0, GPIO_PF1}
#define KB_SCAN_PINS    {GPIO_PA0, GPIO_PD4}
```

keyscan 使用的上下拉电阻都使用 GPIO 的模拟电阻：drive pin 选取下拉 100K，scan pin 选取上拉 10K。那么当没有按键按下时，scan pin 作为输入 GPIO 读到的是被 10K 上拉的高电平。当扫描开始时，在 drive pin 上输出低电平，scan pin 读到低电平，就表示当前列上有按键按下（注意此时 drive pin 不是 float 态，若 output 没打开，scan pin 读到的是 100K 和 10K 的分压电平，还是高）。

定义行列式扫描中，drive pin 输出低电平时 scan pin 扫描到的有效电平。

```
#define KB_LINE_HIGH_VALID        0
```

定义 Row 和 COL 的上下拉：

```
#define MATRIX_ROW_PULL          PM_PIN_PULLDOWN_100K
#define MATRIX_COL_PULL          PM_PIN_PULLUP_10K
//drive pin need 100K pulldown
#define PULL_WAKEUP_SRC_PF0      MATRIX_ROW_PULL
#define PULL_WAKEUP_SRC_PF1      MATRIX_ROW_PULL
//scan  pin need 10K pullup
#define PULL_WAKEUP_SRC_PA0      MATRIX_COL_PULL
#define PULL_WAKEUP_SRC_PD4      MATRIX_COL_PULL
```

由于在 gpio_init 时将 ie 的状态会默认设为 0，scan pin 需要读电平，打开 ie：

```
//drive pin open input to read gpio wakeup level
#define PF0_INPUT_ENABLE         1
#define PF1_INPUT_ENABLE         1
```

当 MCU 进入 sleep mode 时，需要设置 PAD GPIO 唤醒。设置 drive pin 高电平唤醒，按下按键时，drive pin 读到 100K 和 10K 的分压电平，为 10/11 VCC 的高电平。需要打开 drive pin 的 ie 读取其电平状态：

```
//scan pin open input to read gpio level
#define PA0_INPUT_ENABLE         1
#define PD4_INPUT_ENABLE         1
```

8.2 Keyscan and Keymap

8.2.1 Keyscan

按照上面的配置完成后，在 main_loop 中调用下面函数完成 keyscan。

```
u32 kb_scan_key (int numlock_status, int read_key)
```

第一个参数 numlock_status 在 main_loop 中调用时设为 0 即可；只有在 deepsleep 醒来的快速扫描按键时才会将其设为 KB_NUMLOCK_STATUS_POWERON，后面的快速扫键中介绍（对应 DEEP-BACK_FAST_KEYSCAN_ENABLE）。

第二个参数 read_key 是 keyscan 函数按键的缓存处理，这个一般用不到，一直设为 1 即可（为 0 时会将按键值缓存在 buffer 里，不报告给上层）。

返回值用于通知 user 当前的按键扫描是否发现矩阵键盘有变化：有变化时，返回 1；无变化时，返回 0。

kb_scan_key 这个函数是在 main_loop 中调用的，根据 BLE 时序可知，main_loop 的运行时间为 adv_interval 或 conn_interval。广播状态时 (假设 adv_interval 为 30ms)，每 30ms 做一次 key scan；连接状态时（假设 conn_interval = 10ms)，每 10ms 做一次 key scan。

理论上，当前 key scan 发现矩阵上按键的状态和上次 key scan 的状态不一样时，就认为有变化。

实际代码中开启了一个防抖动滤波处理：只有发现连续两次 key scan 的按键状态一样，且和上一次存储的最新矩阵按键状态不一样时，才认为是一个有效的按键变化。这时返回 1 表示按键有变化，并将矩阵按键的状态通过 kb_event 结构体反映出来，同时将更新当前的按键状态为最新的矩阵按键状态。这部分对应的代码为 keyboard.c 中的：

```
unsigned int key_debounce_filter( u32 mtrx_cur[], u32 filt_en );
```

上面所说的最新按键状态指的是矩阵上所有按键的按下或松开的状态的集合。上电时，默认的第一次矩阵按键状态为所有按键都是松开的。只要经过防抖动滤波处理后的矩阵按键的状态发生任何变化，返回值都会为 1，否则返回 0。如：按下一个按键返回一个变化；松开一个按键返回一个变化；按下一个键时再按下第二个键返回一个变化；按下两个键时再按下第三个键返回一个变化；按下两个键时松开其中一个键返回一个变化……

8.2.2 Keymap & kb_event

user 在调用 kb_scan_key 看到一个按键变化时，通过一个全局的结构体变量 kb_event 来获取当前的按键状态。

```
#define KB_RETURN_KEY_MAX   6
typedef struct{
    u8 cnt;
    u8 ctrl_key;
    u8 keycode[KB_RETURN_KEY_MAX];
}kb_data_t;
kb_data_t   kb_event;
```

kb_event 由 8 个 byte 构成：

第一个 cnt 用于指示当前有几个有效的按键被按下；

第二个 ctrl_key 一般不会用到，只有在做标准的 USB HID keyboard 时才会用到（keymap 中的 keycode 设为 0xe0 - 0xe7 时会触发，所以 user 千万不要设这 8 个值)。

keycode[6] 用于最多存储当前 6 个被按下按键的 keycode（如果实际按下的键超过 6 个，只有前 6 个能反应出来)。

所有按键对应的 keycode 在 app_config.h 中定义：

```
#define     KB_MAP_NORMAL   {   {CR_VOL_UP,     VK_1},   \
                                {CR_VOL_DN,     VK_2}, }
```

这个 keymap 的格式和 2*2 矩阵结构一致，可以对应设置按键按下后的 keycode，如按下 Row0 和 Col0 两条线交叉的按键，出来的 keycode 为 CR_VOL_UP。

在 kb_scan_key 函数内部，每次扫描前会将 kb_event.cnt 清 0，而 kb_event.keycode[] 这个数组是不清除的。所以每次返回 1 表示有变化时，用 kb_event.cnt 判断当前矩阵按键上有几个有效的按键。

a) kb_event.cnt = 0 时，上一次有效矩阵状态 kb_event.cnt 肯定是不等于 0 的，但不确定是 1、2 还是 3，这个变化一定是按键释放，不确定是一个键释放还是同时好几个键释放。此时 kb_event.keycode[] 里面即使有数据，也是无效的，忽略不看。

b) kb_event.cnt = 1，可能上次 kb_event.cnt =0，那么按键变化是一个键被按下；可能上次 kb_event.cnt = 2，那么按键变化是两个键中一个被释放；也还有其他可能性，如三个键中两个被同时释放。此时 kb_event.keycode[0] 表示当前被按下的这个键的键值，后面的 keycode 忽略不看。

c) kb_event.cnt = 2，可能上次 kb_event.cnt = 0，变化是两个键同时按下；可能上次 kb_event.cnt = 1，一个键被按下时另一个键被按下；可能上次 kb_event.cnt = 3，三个键被按下时，其中一个被释放；其他可能性等等。此时 kb_event.keycode[0] 和 kb_event.keycode[1] 表示当前被按下的两个键的键值，后面的 keycode 忽略不看。

user 可以每次在 key scan 前自己将 kb_event.keycode 清 0，这时就可以根据 kb_event.keycode 来判断是否有按键变化发生，如下所示。

这个示例只是简单的处理单个按键按下的情况，所以当 kb_event.keycode[0] 非 0 时，就认为是一个按键被按下，并不去判断是否两个键同时按下或者两个键中的一个释放等复杂的情况。

```
kb_event.keycode[0] = 0;//clear keycode[0]
int det_key = kb_scan_key (0, 1);
if (det_key)
{
    key_not_released = 1;
    u8 key0 = kb_event.keycode[0];
    if (kb_event.cnt == 2)   //two key press, do  not process
    {
    }
    else if(kb_event.cnt == 1)
    {
key_buf[2] = key0;
         //send key press
        bls_att_pushNotifyData (HID_NORMAL_KB_REPORT_INPUT_DP_H, key_buf, 8);
}
    else  //key release
    {
        key_not_released = 0;
        key_buf[2] = 0;
        //send key release
        bls_att_pushNotifyData (HID_NORMAL_KB_REPORT_INPUT_DP_H, key_buf, 8);
```

```
    }
}
```

8.3 Keyscan Flow

调用 kb_scan_key 时，一个最基本的 keyscan 的流程如下：

(1) 第一次全矩阵通扫。

将 drive pin 全部输出 drive 电平（0），同时读取所有的 scan pin，检查是否能读到有效的电平，并记录哪一列上读到了有效电平（用 scan_pin_need 标记有效的列号）。

若不使用第一次全矩阵通扫，直接逐行扫的话，至少要进行所有行的扫描，即使没有按键按下也要每次都逐行扫描，比较耗时间。加入了第一次全矩阵通扫后，若没发现任何列上有按键按下，就可以直接退出 keyscan，在没有按键按下时会节省很多时间。

第一次全矩阵通扫的 code 对应如下：

```
scan_pin_need = kb_key_pressed (gpio);
```

在 kb_key_pressed 函数中将所有的行输出低电平，延时 20us 后（延时是为了等待电平稳定后才读 scan pin）。设置了一个 release_cnt 为 6，当检测到矩阵上的按键按下并全部释放后，并不是立刻就认为没有按键而不去逐行扫描了，而是最终缓冲 6 帧，直到发现连续 6 次都是检测到按键全部释放后不再去逐行扫描。实现了一个 key debouce 防抖动的处理。

(2) 根据全矩阵通扫的结果，逐行扫描。

全矩阵通扫发现有按键按下时，开始逐行扫描，ROW0，ROW1 逐行输出有效 drive 电平，读取列上的电平值，找出按键按下的位置。

对应的代码为：

```
u32 pressed_matrix[ARRAY_SIZE(drive_pins)] = {0};
kb_scan_row (0, gpio);
for (int i=0; i<=ARRAY_SIZE(drive_pins); i++) {
    u32 r = kb_scan_row (i < ARRAY_SIZE(drive_pins) ? i : 0, gpio);
    if (i) {
        pressed_matrix[i - 1] = r;
    }
}
```

在做逐行扫描时使用了一些方法来优化代码执行时间：

一是当某行 drive 时，并不需要读取全部的列 CoL0，CoL1，根据之前通扫的 scan_pin_need 可以知道哪些列上能够读到有效电平，此时只读取已经被标记的列即可。

二是每一行 drive 时，需要 20us 左右的等待稳定时间，做了一个缓冲处理，把 20us 的等待时间转化到执行 code 中，节省了这个时间。具体怎么实现不介绍，请 user 自行理解。

最终的矩阵按键状态使用 u32 pressed_matrix[5]（可看出最多支持 40 列）来存储，pressed_matrix[0] 的 bit0~bit2 标记 Row0 上 Col0 ~ Col2 是否有按键，……，pressed_matrix[4] 的 bit0~bit2 标记 Row4 上 Col0 ~ Col2 是否有按键。

(3) 对 pressed_matrix[] 进行防抖动滤波处理

对应代码为：

```
unsigned int key_debounce_filter( u32 mtrx_cur[], u32 filt_en );
u32 key_changed = key_debounce_filter( pressed_matrix, (numlock_status &
 ↪  KB_NUMLOCK_STATUS_POWERON) ? 0 : 1);
```

当 deepsleep 醒来后快速按键检测时，numlock_status = KB_NUMLOCK_STATUS_POWERON，此时 filt_en = 0，不进行滤波，是为了最快速的获取键值。

其他情况下，filt_en = 1，需要滤波处理。滤波处理的思路是：最近的连续两次 pressed_matrix[] 一致，且和上一次有效的 pressed_matrix[] 不一样，才认为是按键矩阵发生了有效的变化，key_changed = 1。

(4) 对 pressed_matrix[] 进行缓存处理

将 pressed_matrix[] 存入到缓冲区，当 kb_scan_key (int numlock_status, int read_key) 中的 read_key 为 1 时，立刻读出缓冲区的数据，当 read_key 为 0 时，缓冲区数据保存起来，不通知上层，只有等到 read_key 为 1 时，才能读出之前缓存的数据。

由于我们的 read_key 永远为 1，这部分可以忽略不计，相当于缓冲区没有起到作用。具体代码不介绍。

(5) 根据 pressed_matrix[]，查表 KB_MAP_NORMAL，返回键值。

对应的函数为 kb_remap_key_code 和 kb_remap_key_row，不具体介绍，user 自行理解。

9 软件定时器（Software Timer）

为了方便 user 一些简单的定时器任务，Telink BLE SDK 提供了 blt software timer demo，并且全部源码提供。user 可以在理解了该 timer 的设计思路后直接使用，也可以自己做一些修改设计。

源代码全部在 vendor/common/blt_soft_timer.c 和 blt_soft_timer.h 文件中，若需要使用，先在 feature_config.h 里将宏 FEATURE_TEST_MODE 改为 TEST_USER_BLT_SOFT_TIMER：

```
#define FEATURE_TEST_MODE                       TEST_USER_BLT_SOFT_TIMER
```

blt soft timer 是基于 system tick 设计的查询式 timer，其准确度无法达到硬件 timer 那么准，且需要保证在 main_loop 中一直被查询。

我们预定：blt soft timer 的使用场景为定时时间大于 5ms、且对于时间误差要求不是特别高的情况。

blt soft timer 的最大特点是不仅在 main_loop 中会被查询，也能确保在进入 suspend 后能够及时唤醒并执行 timer 的任务，该设计是基于低功耗唤醒部分介绍的"应用层定时唤醒"实现的。

目前设计上最多同时支持 4 个 timer 运行，实际 user 可以修改下面的宏来实现更多的或者更少的 timer：

```
#define     MAX_TIMER_NUM     4   //timer max number
```

9.1 Timer 初始化

调用下面的 API 进行初始化：

```
void  blt_soft_timer_init(void);
```

可以看到源码上初始化只是将 blt_soft_timer_process 注册为应用层提前唤醒的回调函数。

```
void  blt_soft_timer_init(void){
      bls_pm_registerAppWakeupLowPowerCb(blt_soft_timer_process);
}
```

9.2 Timer 的查询处理

blt soft timer 的查询处理使用 blt_soft_timer_process 函数来实现：

```
void  blt_soft_timer_process(int type);
```

一方面需要在 main_loop 中如下图所示位置确保一直被调用，另一方面被注册为应用层提前唤醒的回调函数，那么每次在 suspend 中发生定时提前唤醒时，也会快速执行该函数，去处理各 timer 任务。

```
void main_loop (void)
{
    blt_soft_timer_process(MAINLOOP_ENTRY);
    blc_sdk_main_loop();
    #if (UI_KEYBOARD_ENABLE)
        proc_keyboard (0, 0, 0);
    #endif
    blt_pm_proc();
}
```

blt_soft_timer_process 的参数中 type 有如下两种情况：0 表示在 main_loop 中查询进入，1 表示发生了 timer 提前唤醒时进入该函数。

```
#define     MAIN_LOOP_ENTRY          0
#define     CALLBACK_ENTRY           1
```

blt_soft_timer_process 的具体实现比较复杂，基本思路如下：

(1) 首先检查当前 timer table 中是否还有 user 定义的 timer：若没有则直接退出，并关掉应用层定时唤醒；若有 timer 任务，继续往下运行。

```
if(!blt_timer.currentNum){
    bls_pm_setAppWakeupLowPower(0, 0);  //disable
    return;
}
```

(2) 检查时间上最近的一个 timer 任务是否到达：若没有达到，则退出，否则继续往下运行。设计上会保证 timer 在任何时候都是按照时间排序的，所以这里只要看时间上最近的 timer 即可。

```
if( !blt_is_timer_expired(blt_timer.timer[0].t, now) ){
    return;
}
```

(3) 轮询当前所有的 timer 任务，只要时间达到了就执行 timer 对应的任务。

```
for(int i=0; i<blt_timer.currentNum; i++){
if(blt_is_timer_expired(blt_timer.timer[i].t ,now) ){ //timer trigger
    if(blt_timer.timer[i].cb == NULL){
    }
    else{
        result = blt_timer.timer[i].cb();
        if(result < 0){
            blt_soft_timer_delete_by_index(i);
        }
        else if(result == 0){
```

```
            change_flg = 1;
            blt_timer.timer[i].t = now + blt_timer.timer[i].interval;
        }
        else{  //set new timer interval
            change_flg = 1;
            blt_timer.timer[i].interval = result * CLOCK_16M_SYS_TIMER_CLK_1US;
            blt_timer.timer[i].t = now + blt_timer.timer[i].interval;
            }
        }
    }
}
```

这里面可以看到对 timer 任务函数的处理：若该函数返回值小于 0，这个 timer 任务会被删掉，后面不再响应；若返回值为 0，保持上一次的定时值；若返回值大于 0，则以该返回值作为新的定时周期（单位 us）。

(4) 在上面的第 3 步中，如果 timer 任务表中的任务发生了变化，则之前的时间顺序可能会被破坏，这里再重新排序。

```
if(change_flg){
    blt_soft_timer_sort();
}
```

(5) 若最近的 timer 任务的响应时间距离现在只剩 3 秒（3s 可以再改大一些）不到，则将该时间设为应用层提前唤醒的时间，否则关闭应用层提前唤醒。

```
if( (u32)(blt_timer.timer[0].t - now) < 3000 *  CLOCK_16M_SYS_TIMER_CLK_1MS){
    bls_pm_setAppWakeupLowPower(blt_timer.timer[0].t,  1);
}
else{
    bls_pm_setAppWakeupLowPower(0, 0);  //disable
}
```

9.3 添加定时器任务

使用如下 API 添加定时器任务：

```
typedef int (*blt_timer_callback_t)(void);
int     blt_soft_timer_add(blt_timer_callback_t func, u32 interval_us);
```

func 为定期执行的任务函数；interval_us 为定时时间，单位为 us。

定时任务 func 的 int 返回值三种处理方法为：

(1) 返回值小于 0，则该任务执行后被自动删除。可以使用这个特性来控制定时器执行的次数。

(2) 返回 0，则一直使用之前的 interval_us 来定时。

(3) 返回值大于 0，则使用该返回值做为新的定时周期，单位 us。

```
int blt_soft_timer_add(blt_timer_callback_t func, u32 interval_us)
{
    int i;
    u32 now = clock_time();
    if(blt_timer.currentNum >= MAX_TIMER_NUM){  //timer full
        return  0;
    }
    else{
        blt_timer.timer[blt_timer.currentNum].cb = func;
        blt_timer.timer[blt_timer.currentNum].interval = interval_us *
↪   CLOCK_16M_SYS_TIMER_CLK_1US;
        blt_timer.timer[blt_timer.currentNum].t = now +
↪   blt_timer.timer[blt_timer.currentNum].interval;
        blt_timer.currentNum ++;
        blt_soft_timer_sort();
        bls_pm_setAppWakeupLowPower(blt_timer.timer[0].t,  1);
        return  1;
    }
}
```

代码实现中，可以看到当定时器数量超过最大值时，添加失败。每添加一个新的 timer 任务，必须重新做一下排序，以确保定时器任务在任何时候都是按照时间排序的，时间上最近的那个 timer 任务对应的 index 为 0。

9.4 删除定时器任务

除了使用上面返回值小于 0 来自动删除定时器任务，还可以使用下面 API 来指定要删除的定时器任务。

```
int     blt_soft_timer_delete(blt_timer_callback_t func);
```

9.5 Demo

blt soft timer 的 Demo code 请参考 feature_test 中 feature_soft_timer。

```
int gpio_test0(void)
{
    DBG_CHN3_TOGGLE;
    return 0;
}
static u8 timer_change_flg = 0;
int gpio_test1(void)
{
    DBG_CHN4_TOGGLE;
```

```
    timer_change_flg = !timer_change_flg;
    if(timer_change_flg){
        return 7000;
    }
    else{
        return 17000;
    }
}
int gpio_test2(void)
{
    DBG_CHN5_TOGGLE;
    if(clock_time_exceed(0, 5000000)){
        //return  -1;
        blt_soft_timer_delete(&gpio_test2);
    }
    return 0;
}
int gpio_test3(void)
{
    DBG_CHN6_TOGGLE;
    return 0;
}
```

初始化：

```
blt_soft_timer_init();
blt_soft_timer_add(&gpio_test0, 23000);//23ms
blt_soft_timer_add(&gpio_test1, 7000); //7ms <-> 17ms
blt_soft_timer_add(&gpio_test2, 13000);//13ms
blt_soft_timer_add(&gpio_test3, 27000);//27ms
```

定义了 4 个任务，这 4 个定时任务各有特点：

(1) gpio_test0 每 23ms toggle 一次。

(2) gpio_test1 使用了 7ms/17ms 两个时间的切换定时。

(3) gpio_test2 在 5s 后将自己删掉。代码中有两种方式都可以实现这个功能：一是调用 blt_soft_timer_delete(&gpio_test2)；二是 return -1。

(4) gpio_test3 每 27ms toggle 一次。

10 软件模拟 UART（Software UART）

为了方便一些 user 的双 uart 任务，除了支持硬件 UART，B80 BLE SDK 还提供了 blt software UART demo，并且全部源码提供。user 可以在理解了该 demo 的设计思路后直接使用，也可以自己做一些修改设计。

源代码全部在 drivers/ext_driver/software_uart.c 和 software_uart.h 文件中，若需要使用，先在 feature_config.h 里将宏 FEATURE_TEST_MODE 改为 TEST_USER_BLT_SOFT_UART：

```
#define FEATURE_TEST_MODE                       TEST_USER_BLT_SOFT_UART
```

blt soft uart TX 基于轮询设计，只有在广播时间和连接时间的间隔之间允许发送。blt soft uart RX 是基于 gpio 中断和 timer 中断设计。

注意：

> 为了减少中断处理任务时间，软件串口工程必须使用 SYS_CLK_48M_Crystal。

10.1 Software UART 初始化

调用下面的 API 进行初始化：

```
soft_uart_rx_handler(app_soft_rx_uart_cb);
soft_uart_RxSetFifo(uart_rx_fifo.p, uart_rx_fifo.size);
soft_uart_init();
```

可以看到 soft_uart_rx_handler 函数将 irq handler 中的 UART RX 处理函数注册为应用层定义的回调函数 app_soft_rx_uart_cb。

```
int app_soft_rx_uart_cb(void)//UART data send to Master,we will handler the data as CMD or DATA
{
    if (((uart_rx_fifo.wptr - uart_rx_fifo.rptr) & 255) < uart_rx_fifo.num) {
        uart_rx_fifo.wptr++;
        unsigned char* p = uart_rx_fifo.p + (uart_rx_fifo.wptr & (uart_rx_fifo.num - 1)) *
        ↪  uart_rx_fifo.size;
        soft_uart_RxSetFifo(p, uart_rx_fifo.size);
    }
    return 0;
}
```

soft_uart_RxSetFifo 函数将 UART 接收 FIFO 注册为上层的定义的 uart_rx_fifo。

```
u8          uart_rx_buf[80 * 4] = {0};
my_fifo_t   uart_rx_fifo = {
    80,
    4,
```

```
    0,
    0,
    uart_rx_buf,};
```

soft_uart_init 函数将 software UART RX 和 TX 使用的 GPIO 口、RX GPIO 中断、software UART 相关的变量和使用到的 timer0 进行初始化。

```
void soft_uart_init(void)
{

    // set software rx io
    gpio_set_func(SOFT_UART_RX_IO, AS_GPIO);
    gpio_set_output_en(SOFT_UART_RX_IO, 0);
    gpio_set_input_en(SOFT_UART_RX_IO, 1);
    gpio_setup_up_down_resistor(SOFT_UART_RX_IO, PM_PIN_PULLUP_10K);
    gpio_set_interrupt(SOFT_UART_RX_IO, POL_FALLING);

    //set software tx io
    gpio_set_func(SOFT_UART_TX_IO , AS_GPIO);
    gpio_setup_up_down_resistor(SOFT_UART_TX_IO, PM_PIN_PULLUP_1M);
    gpio_set_output_en(SOFT_UART_TX_IO,1);//Enable output
    gpio_write(SOFT_UART_TX_IO, 1);// Add this code to fix the problem that the first byte will
↪   be error.

    soft_uart_rece.bit_num = 0x00;
    soft_uart_rece.temp_byte = 0x00;

    soft_uart_rece.stop_count = 0;
    soft_uart_rece.done_count = 0;

    soft_uart_rece.state = SOFT_UART_WAIT;

    soft_uart_rece.mutex_flag = 0;

    soft_uart_rece.time_interval = (1000000 / SOFT_UART_BAUD_RATE) * CLOCK_SYS_CLOCK_1US +
↪   SOFT_UART_OFFSET;
    //SET TIME
    timer0_set_mode(TIMER_MODE_SYSCLK, 0, SOFT_UART_INTERVAL * CLOCK_SYS_CLOCK_1US);
    timer_stop(TIMER0);
}
```

10.2 Software UART TX 处理

blt software UART TX 处理使用 soft_uart_send 函数来实现。user 可以参考模拟串口只发数据的 Demo，配置如下。

```
#define     TEST_RX_TX_RUN                  1
#define     TEST_ONLY_TX_RUN                2

#define     TEST_SOFT_UART_RUN_MODEL        TEST_ONLY_TX_RUN
```

在 main_loop 中如下图所示位置调用 soft_uart_send 函数，去处理各 timer 任务。

```
u8 send_buf[10] = {0xaa,0x55,0xaa,0x55,0xaa,0x55,0xaa,0x55,0xaa,0x55};

void main_loop (void)
{
    ...
    soft_uart_send(send_buf, 10);
    ...
}
```

通过串口上位机工具可以看到发送的 10 Byte 数据。另外，外接逻辑分析仪也可以看到效果。

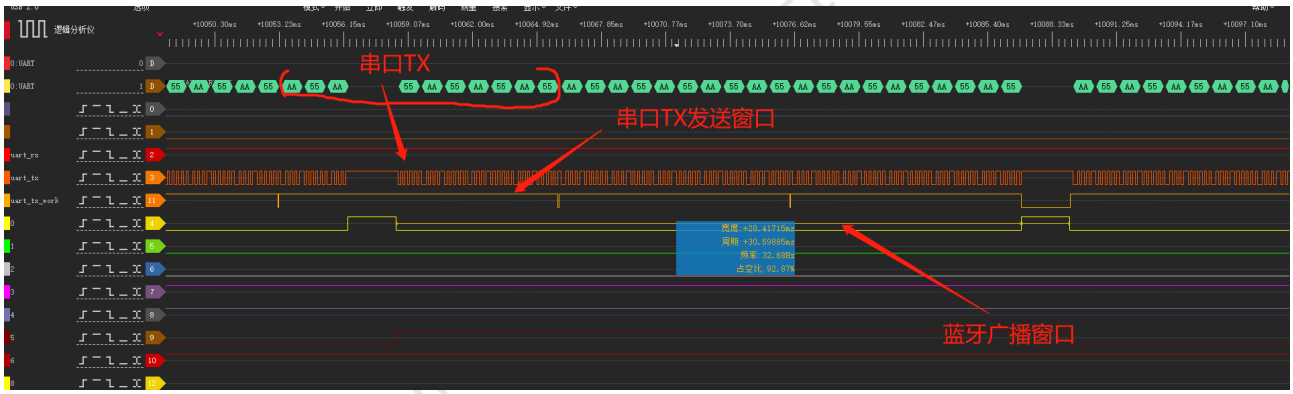


Figure 10.1: 广播态下模拟串口只发数据

blt software UART TX 的具体实现思路如下：

(1) 首先按 Byte 发送数据，发送前检查当前状态机状态。如果当前状态为 advertising 态，跳转到 (2)；如果当前状态为 connection 态，跳转到 (3)；如果当前状态为 standby 态，跳转到 (4)。

```
void soft_uart_send(unsigned char * buf, unsigned char len) {

    unsigned char i;
    for (i = 0; i < len; i++) {

        unsigned char s;
        extern u8 blc_ll_getCurrentState(void);
        s = blc_ll_getCurrentState();
        ...
    }
```

```
}
```

(2) 当前状态为 advertising 态，调用 blc_sdk_adv 检查是否已经到达广播时间，如果到达，先发送广播，之后允许 TX，跳转到 (4)。

```
void soft_uart_send(unsigned char * buf, unsigned char len) {
    ...
    /*
    #define     BLS_LINK_STATE_ADV                  BIT(0)
    */
    if (s == BIT(0)) {
        extern void blc_sdk_adv(void);
        blc_sdk_adv();
    }
    ...
}
```

(3) 当前状态为 connection 态，调用 blc_ll_SoftUartisRfState 检查当前 RF 任务是否影响 TX 任务，如果当前时刻到下一次连接事件大于 UART TX 一个 Byte 数据的时间（SOFT_UART_SEND_ONE_BYTE），允许 TX，跳转到 (4)，否则需要等待下一次连接事件结束。

```
void soft_uart_send(unsigned char * buf, unsigned char len) {
    ...
    /*
    #define         BLS_LINK_STATE_CONN                 BIT(3)
    */
    if (s == BIT(3)) {
        extern void blc_ll_SoftUartisRfState(int acl_margin_us);
        blc_ll_SoftUartisRfState(SOFT_UART_SEND_ONE_BYTE);
    }
    ...
}
```

(4) 在允许 UART TX 的情况下，调用 soft_uart_putchar 函数将第 i 个 Byte 发送。

```
void soft_uart_send(unsigned char * buf, unsigned char len) {
    ...
    soft_uart_putchar(buf[i]);
    ...
}
```

10.3 Software UART RX 处理

blt software UART RX 处理使用 gpio 中断和 timer 中断来实现，RX 动作在 irq handler 中进行。user 可以参考模拟串口收发数据的 Demo，配置如下。

```
#define     TEST_RX_TX_RUN                  1
#define     TEST_ONLY_TX_RUN                2

#define     TEST_SOFT_UART_RUN_MODEL        TEST_RX_TX_RUN
```

在 main_loop 中如下图所示位置去处理收到的 RX 数据，并通过 soft_uart_send 函数再 TX 回去。

```
void main_loop (void)
{
    ...
    if (uart_rx_fifo.wptr != uart_rx_fifo.rptr) {
        u8 *p = uart_rx_fifo.p + (uart_rx_fifo.rptr & (uart_rx_fifo.num - 1))
                * uart_rx_fifo.size;

        soft_uart_send(&p[4], p[0]);

        uart_rx_fifo.rptr++;
    }
    ...
}
```

通过串口上位机工具发送 10 Byte 数据，可以在接受窗口收到下位机发回的数据。另外，外接逻辑分析仪也可以看到效果。

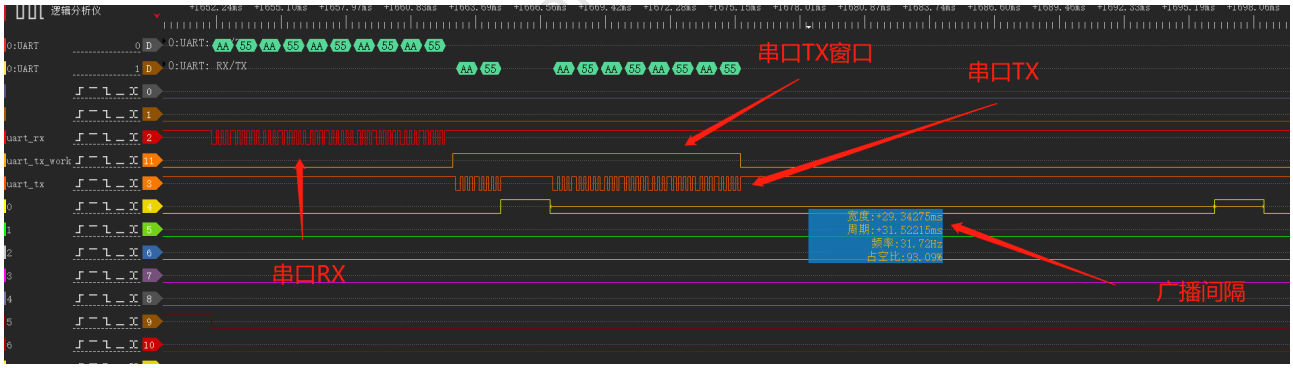


Figure 10.2: 广播态下模拟串口收发数据

blt software UART RX 的具体实现思路如下：

(1) 首先收到上位机发送的数据时，会触发 GPIO irq，如果 software uart state 为 SOFT_UART_WAIT，则关闭 GPIO irq，通过设置 timer irq tick 来触发中断，读取下一次 RX interval 时的 GPIO 输入值，并将当前 software uart state 设置为 SOFT_UART_WORK。

```
_attribute_ram_code_ void soft_uart_irq_handler(void)
{
    ...
```

```
    if ((reg_irq_src & FLD_IRQ_GPIO_EN) == FLD_IRQ_GPIO_EN) {
        reg_irq_src |= FLD_IRQ_GPIO_EN; // clear the relevant irq
        if ((gpio_read(SOFT_UART_RX_IO) == 0)&&(SOFT_UART_WAIT & soft_uart_rece.state)) {
                BM_CLR(reg_gpio_irq_wakeup_en(SOFT_UART_RX_IO), SOFT_UART_RX_IO & 0xff); //
↪  close GPIO irq
                soft_uart_rece.bit_num = 0x00;
                soft_uart_rece.temp_byte = 0x00;
                soft_uart_rece.state &= ~SOFT_UART_DONE_CHECK;
                soft_uart_rece.state &= ~SOFT_UART_WAIT;
                soft_uart_rece.state |= SOFT_UART_WORK;
                soft_uart_rece.done_count = 0;
                timer0_set_mode(TIMER_MODE_SYSCLK, 0, soft_uart_rece.time_interval);
                timer_start(TIMER0);
        }
    }
    ...
}
```

(2) 当触发 timer irq 时，清完 timer irq status，之后判断当前 software uart state，并进行相应操作。如果 state 为 SOFT_UART_WORK，跳转到 (3)；如果 state 为 SOFT_UART_STOP_CHECK，跳转到 (4)；如果 state 为 SOFT_UART_DONE_CHECK，跳转到 (5)。

```
_attribute_ram_code_ void soft_uart_irq_handler(void)
{
    ...
    //time irq
    if (timer_get_interrupt_status(FLD_TMR_STA_TMR0)) {
        timer_clear_interrupt_status(FLD_TMR_STA_TMR0); //clear irq status

        if (soft_uart_rece.state & SOFT_UART_WORK) {
            ...
        } else if (soft_uart_rece.state & SOFT_UART_STOP_CHECK) {
            ...
        } else if (soft_uart_rece.state & SOFT_UART_DONE_CHECK) {
            ...
        }
    }
    ...
}
```

(3) software uart state 为 SOFT_UART_WORK，暂存当前位的值，如果已经接收到 8 位的值，将 software uart state 置为 SOFT_UART_STOP_CHECK。

```
_attribute_ram_code_ void soft_uart_irq_handler(void)
{
    ...
```

```
    if (soft_uart_rece.state & SOFT_UART_WORK) {
        if (1 == gpio_read(SOFT_UART_RX_IO)) { //
            soft_uart_rece.temp_byte |= BIT(soft_uart_rece.bit_num);
        }
        soft_uart_rece.bit_num++;
        if (8 == soft_uart_rece.bit_num) {
            soft_uart_rece.bit_num = 0x00;
            soft_uart_rece.state |= SOFT_UART_STOP_CHECK; //change state
            soft_uart_rece.state &= ~SOFT_UART_WORK;
        }
    }
    ...
}
```

(4) software uart state 为 SOFT_UART_STOP_CHECK，保存之前传输的 Byte 值，之后将 software uart state 置 SOFT_UART_DONE_CHECK 和 SOFT_UART_WAIT, 并调用 soft_uart_RxHandler 进行一些处理，重新打开 RX GPIO irq。

```
_attribute_ram_code_ void soft_uart_irq_handler(void)
{
    ...
    if (soft_uart_rece.state & SOFT_UART_STOP_CHECK) {
        soft_uart_rece.state &= ~SOFT_UART_STOP_CHECK;
        if (1 == gpio_read(SOFT_UART_RX_IO)) { //
            soft_uart_rece.data[soft_uart_rece.data_count + 4] = soft_uart_rece.temp_byte; //len
↪   + buf
            soft_uart_rece.temp_byte = 0x00;
            soft_uart_rece.data_count++;
            if (soft_uart_rece.data_count >= soft_uart_rece.data_size) { //over flow
                soft_uart_rece.data[0] = soft_uart_rece.data_count;
                if (soft_uart_RxHandler)
                    soft_uart_RxHandler();
            }
        }
        soft_uart_rece.state |= SOFT_UART_DONE_CHECK;
        soft_uart_rece.state |= SOFT_UART_WAIT;
        reg_irq_src |= FLD_IRQ_GPIO_EN; // clear the relevant irq
        BM_SET(reg_gpio_irq_wakeup_en(SOFT_UART_RX_IO), SOFT_UART_RX_IO & 0xff); //start io irq
    }
    ...
}
```

(5) software uart state 为 SOFT_UART_DONE_CHECK，清 SOFT_UART_DONE_CHECK 状态并调用 soft_uart_RxHandler 进行一些处理，关闭 timer irq。

```
_attribute_ram_code_ void soft_uart_irq_handler(void)
{
    ...
    if (soft_uart_rece.state & SOFT_UART_DONE_CHECK) {
        soft_uart_rece.done_count++;
        if (UART_RECE_DONE_NUM <= soft_uart_rece.done_count) {
            timer_stop(TIMER0);
            if(soft_uart_rece.data_count > 0){
                soft_uart_rece.data[0] = soft_uart_rece.data_count;
                if (soft_uart_RxHandler)
                    soft_uart_RxHandler();
            }
            soft_uart_rece.state &= ~SOFT_UART_DONE_CHECK;
        }
    }
    ...
}
```

11 Feature Demo 介绍

B80_feature_test 针对一些常用的 BLE 相关的 feature 给出了 demo code，用户可参考这些 demo 完成自己的功能实现，详情见 code。在 B80_feature_test 工程里面 app_config.h 中对宏“FEATURE_TEST_MODE”进行选择性的定义，即可切换到不同 feature test 的 Demo。

```
////////////////// TEST FEATURE SELECTION ////////////////////////////////
#define TEST_FEATURE_BACKUP                         0

//ble link layer test
#define TEST_POWER_ADV                              1
#define TEST_ADVERTISING_IN_CONN_SLAVE_ROLE         2


#define TEST_SDATA_LENGTH_EXTENSION                 3
#define TEST_SLAVE_MD                               4

#define TEST_USER_BLT_SOFT_TIMER                    5

#define TEST_PHY_CONN                               6

#define TEST_BLE_PHY                                7   // BQB PHY_TEST demo
#define TEST_EMI                                    8       // emi test


#define TEST_GATT_SECURITY                          9


#define TEST_USER_BLT_SOFT_UART                     10




#define FEATURE_TEST_MODE                           TEST_FEATURE_BACKUP
```

Figure 11.1: feature test 的测试 Demo 选择

下面介绍每个 Demo 测试方法。

11.1 广播功耗测试

此项主要测试不同广播参数广播时的功耗，用户在测试时可外接万用表测量功耗。在 feature_config.h 需要修改 FEATURE_TEST_MODE 为 TEST_POWER_ADV。

```
#define FEATURE_TEST_MODE             TEST_POWER_ADV
```

在 feature_adv_power 中根据需求修改广播类型及广播参数，app_config.h 中提供了多种广播类型供用户选择，分别为可连接广播及非可连接广播。

```
//TEST_POWER_ADV config start
#define CONNECT_12B_1S_1CHANNEL          0
#define CONNECT_12B_1S_3CHANNEL          1
#define CONNECT_12B_500MS_3CHANNEL       2
#define CONNECT_12B_30MS_3CHANNEL        3

#define UNCONNECT_16B_1S_3CHANNEL        4
#define UNCONNECT_16B_1_5S_3CHANNEL      5
#define UNCONNECT_16B_2S_3CHANNEL        6

#define UNCONNECT_32B_1S_3CHANNEL        7
#define UNCONNECT_32B_1_5S_3CHANNEL      8
#define UNCONNECT_32B_2S_3CHANNEL        9

#define APP_ADV_POWER_TEST_TYPE          UNCONNECT_16B_1S_3CHANNEL
//TEST_POWER_ADV end
```

Figure 11.2: ADV_POWER_TEST_TYPE 的选择

11.1.1 可连接广播功耗测试

在 feature_adv_power 中，以下代码是可连接广播类型。

```
#define CONNECT_12B_1S_1CHANNEL          0
#define CONNECT_12B_1S_3CHANNEL          1
#define CONNECT_12B_500MS_3CHANNEL       2
#define CONNECT_12B_30MS_3CHANNEL        3
```

Demo 默认广播数据长度为 12byte，用户可根据需求修改。

```
//ADV data length: 12 byte
u8 tbl_advData[12] = {0x08, 0x09, 't', 'e', 's', 't', 'a', 'd', 'v',0x02, 0x01, 0x05,};
```

Demo 提供了 1s_1channel，1s_3channel，500ms_3channel，30ms_3channel 的广播参数，用户根据自己需求选择对应的测项即可。

11.1.2 非可连接广播功耗测试

在 feature_adv_power 中，以下代码是非可连接广播类型。

```
#define UNCONNECT_16B_1S_3CHANNEL        4
#define UNCONNECT_16B_1_5S_3CHANNEL      5
#define UNCONNECT_16B_2S_3CHANNEL        6

#define UNCONNECT_31B_1S_3CHANNEL        7
```

```
#define UNCONNECT_31B_1_5S_3CHANNEL     8
#define UNCONNECT_31B_2S_3CHANNEL       9
```

Demo 提供了两种广播数据长度分别为 16byte 和 31byte，用户可根据需求选择。

```
u8 tbl_advData[] = {
                15, 0x09, 't', 'e', 's', 't', 'a', 'd', 'v', '8', '9', 'A', 'B', 'C', 'D', 'E',
               };//ADV data length: 16 byte
u8 tbl_advData[] = {
                30, 0x09, 't', 'e', 's', 't', 'a', 'd', 'v', '8', '9', 'A', 'B', 'C', 'D', 'E',
                 ↪  'F', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D'
            };//ADV data length: max 31 byte
```

Demo 提供了 1s_3channel，1.5s_3channel，2s_3channel 的广播参数，用户根据自己需求选择对应的测项即可。

11.2 GATT Security 测试

用户需要在 feature_config.h 中将 FEATURE_TEST_MODE 为 TEST_GATT_SECURITY。

```
#define FEATURE_TEST_MODE          TEST_GATT_SECURITY
```

由 BLE 模块 ATT&GATT 章节可知，服务列表中每个 Attribute 都定义了读写权限，即配对模式也要达到相应的等级才能去读或者写。如 Demo 的 SPP 服务中：

```
// client to server RX
{0,ATT_PERMISSIONS_READ,2,sizeof(TelinkSppDataClient2ServerCharVal),(u8*)(&my_characterUUID),
 ↪  (u8*)(TelinkSppDataClient2ServerCharVal), 0},    //prop
{0,SPP_C2S_ATT_PERMISSIONS_RDWR,16,sizeof(SppDataClient2ServerData),(u8*)
 ↪  (&TelinkSppDataClient2ServerUUID), (u8*)(SppDataClient2ServerData), &module_onReceiveData},
 ↪  //value
{0,ATT_PERMISSIONS_READ,2,sizeof(TelinkSPPC2SDescriptor),(u8*)&userdesc_UUID,(u8*)
 ↪  (&TelinkSPPC2SDescriptor)},
```

第二个 Attribute 的读写权限定义为：SPP_C2S_ATT_PERMISSIONS_RDWR。

此读写权限由用户去选择，可以选择如下其中一个：

```
#define     SPP_C2S_ATT_PERMISSIONS_RDWR        ATT_PERMISSIONS_RDWR
#define     SPP_C2S_ATT_PERMISSIONS_RDWR        ATT_PERMISSIONS_ENCRYPT_RDWR
#define     SPP_C2S_ATT_PERMISSIONS_RDWR        ATT_PERMISSIONS_AUTHEN_RDWR
#define     SPP_C2S_ATT_PERMISSIONS_RDWR        ATT_PERMISSIONS_SECURE_CONN_RDWR
```

无论选择哪个，当前配对模式要高于或者等于此读写权限等级才能正确的读写服务。

GATT 测试主要测试配对加密的流程，主要分为以下几种方式：

- LE_Security_Mode_1_Level_1, no authentication and no encryption.
- LE_Security_Mode_1_Level_2, unauthenticated paring with encryption.
- LE_Security_Mode_1_Level_3, authenticated paring with encryption-legacy.
- LE_Security_Mode_1_Level_4, authenticated paring with encryption-sc.

用户需要将 app_config.h 根据需求选择相应的配对模式。

```
// LE security mode select
#define     SMP_TEST_MODE                       LE_SECURITY_MODE_1_LEVEL_1
```

下面对每种配对模式进行简要介绍。

11.2.1 LE_Security_Mode_1_Level_1

```
// LE_Security_Mode_1_Level_1, no authentication and no encryption
#define     SMP_TEST_NO_SECURITY                1
```

LE_Security_Mode_1_Level_1 为最简单的配对方式，既不认证也不加密。用户将 app_config.h 的 LE_SECURITY_MODE_1_LEVEL_1 更改为 SMP_TEST_NO_SECURITY。

```
#define     LE_SECURITY_MODE_1_LEVEL_1          SMP_TEST_NO_SECURITY
```

SMP_TEST_MODE 更改为 LE_SECURITY_MODE_1_LEVEL_1。

```
#define     SMP_TEST_MODE                       LE_SECURITY_MODE_1_LEVEL_1
```

11.2.2 LE_Security_Mode_1_Level_2

```
// LE_Security_Mode_1_Level_2, unauthenticated pairing with encryption
#define     SMP_TEST_LEGACY_PAIRING_JUST_WORKS  2 //JustWorks
#define     SMP_TEST_SC_PAIRING_JUST_WORKS      3 //JustWorks
```

LE_Security_Mode_1_Level_2 模式为 just work，只加密不认证。Just work 又分为 legacy just work，sc just work。GATT Security 测试中 app_config.h 的 LE_SECURITY_MODE_1_LEVEL_2 默认为 SMP_TEST_LEGACY_PARING_JUST_WORKS，SMP_TEST_SC_PARING_JUST_WORKS 未做处理，user 可自行按照 API 配置。下面介绍 SMP_TEST_LEGACY_PARING_JUST_WORKS。

11.2.2.1 SMP_TEST_LEGACY_PARING_JUST_WORKS

用户作如下修改：

```
#define     LE_SECURITY_MODE_1_LEVEL_2        SMP_TEST_LEGACY_PARING_JUST_WORKS
```

```
#define     SMP_TEST_MODE                     LE_SECURITY_MODE_1_LEVEL_2
```

示意流程图：

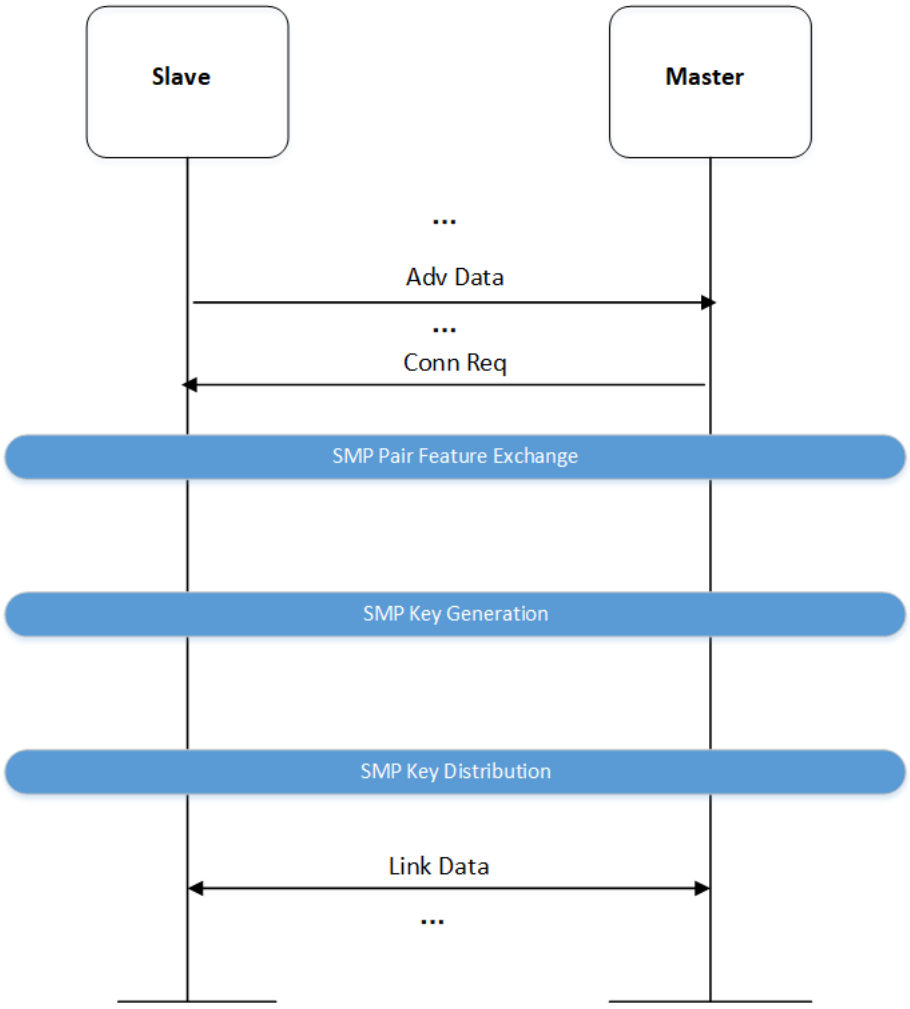


Figure 11.3: Legacy Just Work 流程示意图

11.2.3 LE_Security_Mode_1_Level_3

```
// LE_Security_Mode_1_Level_3, authenticated pairing with encryption
#define     SMP_TEST_LEGACY_PASSKEY_ENTRY_SDMI  4 //PK_Resp_Dsply_Init_Input
#define     SMP_TEST_LEGACY_PASSKEY_ENTRY_MDSI  5 //PK_Init_Dsply_Resp_Input
#define     SMP_TEST_LEGACY_PASSKEY_ENTRY_MISI  6 //PK_BOTH_INPUT, not test
#define     SMP_TEST_LEGACY_PASSKEY_ENTRY_OOB   7 //OOB_Authentication, not test
```

LE_Security_Mode_1_Level_3 为既认证又加密 Legacy 配对方式，根据配对参数设置分为 OOB，PassKey Entry, Numeric Comparison。目前 demo 仅提供了 SMP_TEST_LEGACY_PASSKEY_ENTRY_SDMI 示例代码。下面简单介绍下这种方式。

11.2.3.1 SMP_TEST_LEGACY_PASSKEY_ENTRY_SDMI

用户需要在 app_config.h 中如下修改：

```
#define     LE_SECURITY_MODE_1_LEVEL_3      SMP_TEST_LEGACY_PASSKEY_ENTRY_SDMI

#define     SMP_TEST_MODE       LE_SECURITY_MODE_1_LEVEL_3
```

配对过程中需要 slave 端显示密钥，master 端输入密钥。初始化时，注册了配对相关 gap event。配对信息会通知到 app 层。

```
blc_gap_registerHostEventHandler( app_host_event_callback );
blc_gap_setEventMask( GAP_EVT_MASK_SMP_PAIRING_BEGIN            |  \
                      GAP_EVT_MASK_SMP_PAIRING_SUCCESS          |  \
                      GAP_EVT_MASK_SMP_PAIRING_FAIL               |  \
                      GAP_EVT_MASK_SMP_TK_DISPALY               |  \
                      GAP_EVT_MASK_SMP_CONN_ENCRYPTION_DONE     |  \
                      GAP_EVT_MASK_SMP_SECURITY_PROCESS_DONE);
```

用户需要在收到 GAP_EVT_SMP_TK_DISPALY 消息时打印下当前的密钥信息。

```
int app_host_event_callback (u32 h, u8 *para, int n)
{
    u8 event = h & 0xFF;
    switch(event)
    {
    ...
        case GAP_EVT_SMP_TK_DISPALY:
        {
            char pc[7];
            u32 pinCode = *(u32*)para;
            sprintf(pc, "%d", pinCode);
            printf("TK display:%s\n", pc);
        }
    ...
    }
}
```

基本流程示意图如下：

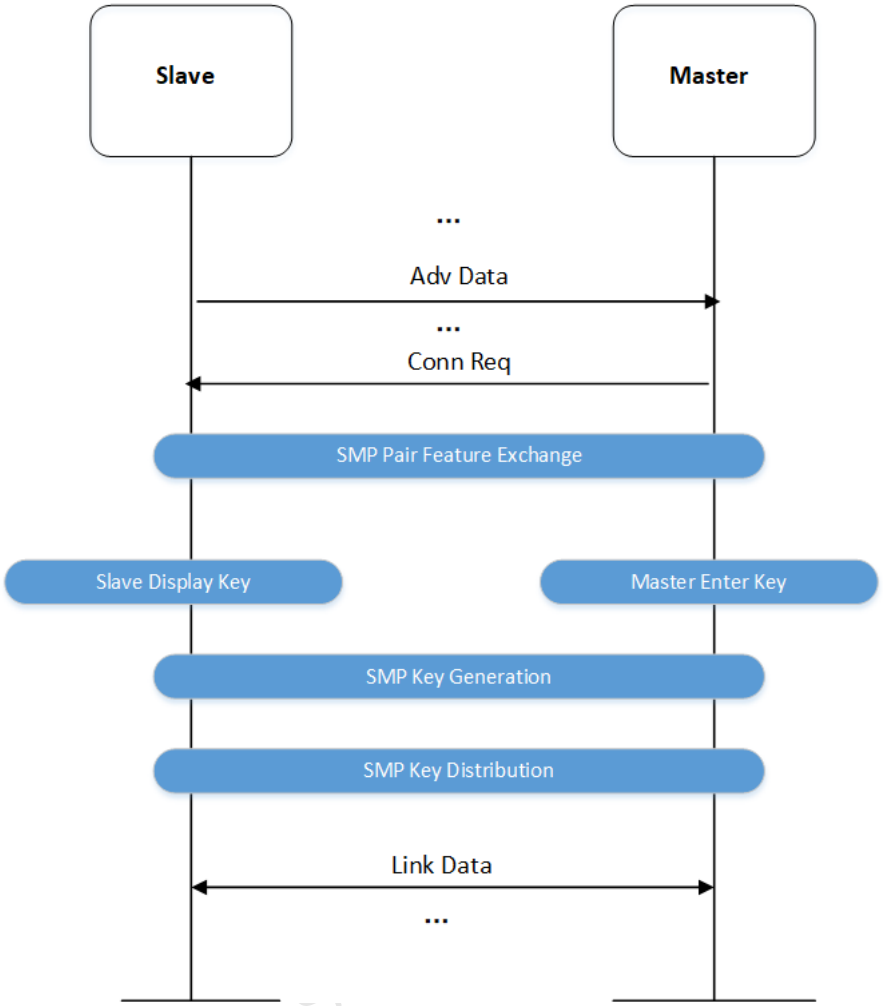


Figure 11.4: Legacy Just Work SDMI 流程示意图

11.2.4 LE_Security_Mode_1_Level_4

```
// LE_Security_Mode_1_Level_4, authenticated pairing with encryption
#define     SMP_TEST_SC_NUMERIC_COMPARISON      8 //Numric_Comparison
#define     SMP_TEST_SC_PASSKEY_ENTRY_SDMI      9 //PK_Resp_Dsply_Init_Input
#define     SMP_TEST_SC_PASSKEY_ENTRY_MDSI      10//PK_Init_Dsply_Resp_Input
#define     SMP_TEST_SC_PASSKEY_ENTRY_MISI      11//PK_BOTH_INPUT, not test
#define     SMP_TEST_SC_PASSKEY_ENTRY_OOB       12//OOB_Authentication, not test
```

LE_Security_Mode_1_Level_4 为既认证又加密 SC 配对方式，根据配对参数设置分为 OOB，PassKey Entry，Numeric Comparison。目前 demo 仅提供了 SMP_TEST_SC_PASSKEY_ENTRY_SDMI 的示例代码。下面简单介绍下这种方式。

11.2.4.1 SMP_TEST_SC_PASSKEY_ENTRY_SDMI

用户需要在 feature_security.c 中如下修改：

```
#define     LE_SECURITY_MODE_1_LEVEL_4      SMP_TEST_SC_PASSKEY_ENTRY_SDMI
```

```
#define     SMP_TEST_MODE       LE_SECURITY_MODE_1_LEVEL_4
```

配对过程中需要 slave 端显示密钥，master 端输入密钥。初始化时，注册了配对相关 gap event。配对信息会通知到 app 层。

```
    blc_gap_registerHostEventHandler( app_host_event_callback );
    blc_gap_setEventMask( GAP_EVT_MASK_SMP_PAIRING_BEGIN            |  \
                          GAP_EVT_MASK_SMP_PAIRING_SUCCESS          |  \
                          GAP_EVT_MASK_SMP_PAIRING_FAIL             |  \
                          GAP_EVT_MASK_SMP_TK_DISPALY               |  \
                          GAP_EVT_MASK_SMP_CONN_ENCRYPTION_DONE     |  \
                          GAP_EVT_MASK_SMP_SECURITY_PROCESS_DONE);
```

用户需要在收到 GAP_EVT_MASK_SMP_TK_DISPLAY 消息时打印下当前的密钥信息。

```
int app_host_event_callback (u32 h, u8 *para, int n)
{
    u8 event = h & 0xFF;
    switch(event)
    {
...
        case GAP_EVT_SMP_TK_DISPALY:
        {
            char pc[7];
            u32 pinCode = *(u32*)para;
            sprintf(pc, "%d", pinCode);
            printf("TK display:%s\n", pc);
        }
        break;
...
    }
}
```

基本流程示意图如下：

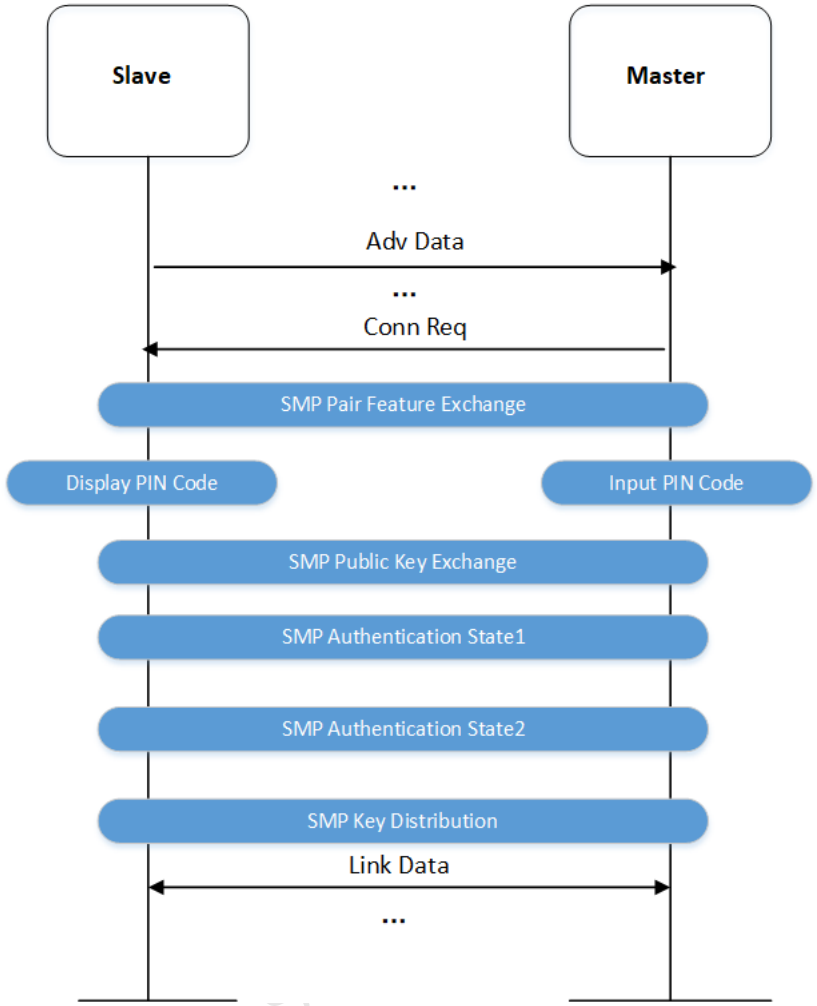


Figure 11.5: SC SDMI 配对流程示意图

11.2.5 GATT Security 测试流程

如当前的配对模式为 LE_SECURITY_MODE_1_LEVEL_3，即既有认证又有加密 Lagacy 配对模式。所以当前读写权限可以如下选择。

```
#define     SPP_C2S_ATT_PERMISSIONS_RDWR        ATT_PERMISSIONS_AUTHEN_RDWR
```

流程示意图如下：

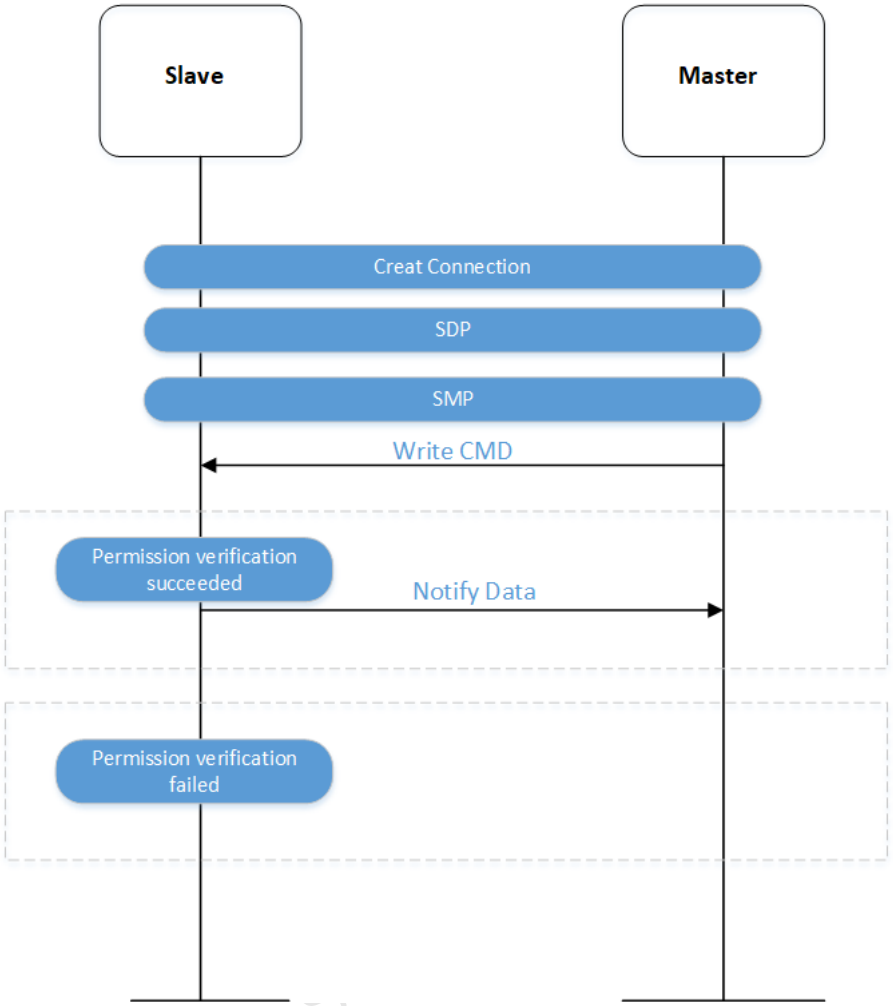


Figure 11.6: Gatt Security 示意图

11.3 DLE 测试

DLE 测试主要测试长包，对应的 feature_config.h 选择，代码如下：

```
#define FEATURE_TEST_MODE          TEST_SDATA_LENGTH_EXTENSION
```

烧录完之后按下 reset，B80 作为 slave 端选择与 master 建立连接，连接成功后会进行 MTU 及 DataLength 交换。

```
blc_att_requestMtuSizeExchange(BLS_CONN_HANDLE, MTU_SIZE_SETTING);
blc_ll_exchangeDataLength(0x14 , ACL_CONN_MAX_TX_OCTETS);//LENGTH_REQ opcode: 0x14
```

交换成功后，slave 会每隔 3.33s 向 master 发送长包数据。

测试流程示意图如下：

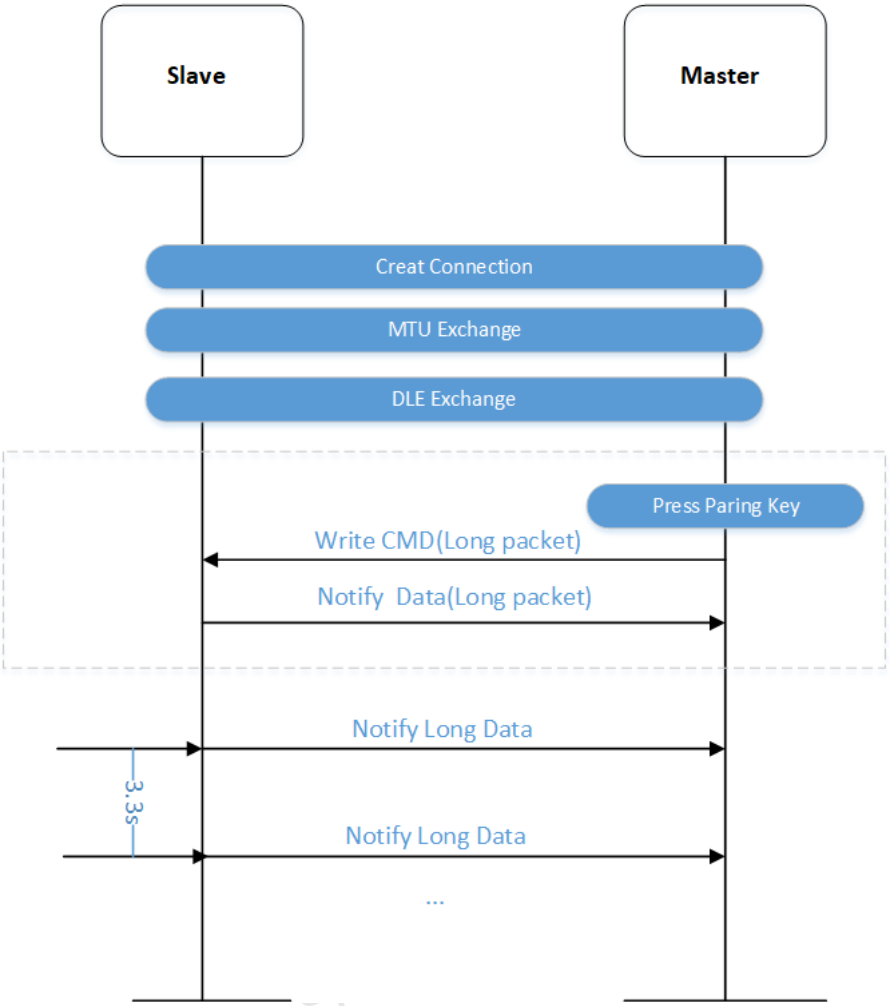


Figure 11.7: DLE 测试流程示意图

注意：

> 需要注意的是，在打开 deepsleep retention 时候，如果将 RX FIFO size 和 TX FIFO size 设置为 200 以上，可能会存在 ram 不够用的情况，user 需要评估修改。

11.4 Soft Timer 测试

可参考软件定时器（Software Timer）一章。

11.5 EMI 测试

feature_emi 用于产生所需的 EMI 测试信号，该例程需配合 EMI_Tool" 和 "Non_Signaling_Test_Tool"工具使用。

11.5.1 协议

通信协议请参考《Telink SoC EMI Test User Guide》。

11.5.2 Demo 说明

B80 中 EMI 测试支持 carrieronly 模式、continue 模式、burst 模式、收包模式。

支持的无线通信方式包括 Ble1M、Ble2M、Ble125K、Ble500K、Zigbee250K。

关于各个模式和功能函数的介绍，user 可参考《Telink Driver SDK Developer Handbook》。

12 其他模块

12.1 24MHz 晶体外部电容

参考下图中的 24MHz 晶体匹配电容的位置 C3/C4。

SDK 默认使用 B80 内部电容（即 ana_8a<5:0> 对应的 cap）作为 24MHz 晶振的匹配电容，此时 C3/C4 不用焊接电容。使用该方案的优势是：在 Telink 治具上可以测量并调节该电容，使得最终应用产品的频点值达到最佳。

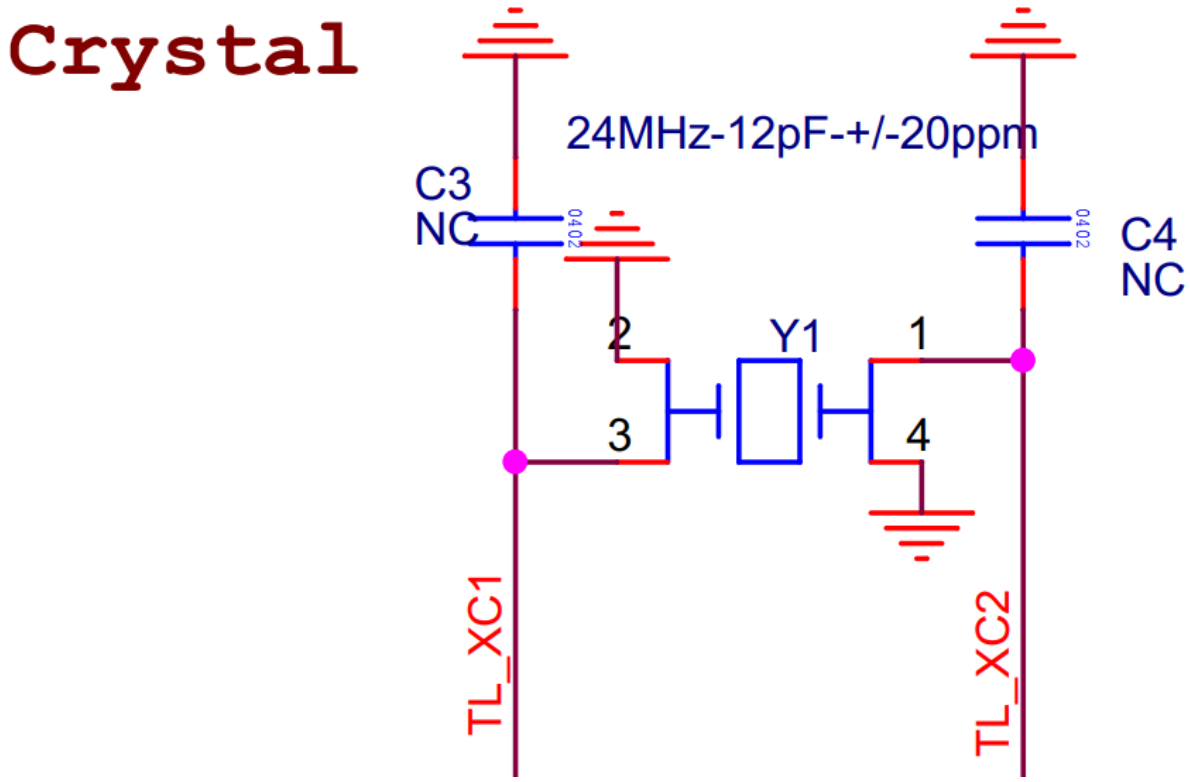


Figure 12.1: 24M 晶体电路

如果需要使用外部焊接电容作为 24M 晶振的匹配电容（C3/C4 焊接电容），则只要在 main 函数开始的地方（一定要在 cpu_wakeup_init() 之后，blc_app_loadCustomizedParameters() 之前）调用下面 API 即可：

```
static inline void blc_app_setExternalCrystalCapEnable(u8  en)
{
    blt_miscParam.ext_cap_en = en;
}
```

只要按要求调用该 API，SDK 会自动处理所有的设置，包括关掉内部匹配电容、不再读取频偏校正值等。

12.2 32KHz 时钟源选择

SDK 支持使用 MCU 内部 32k RC 振荡电路 (简称 32k RC) 或者外部 32k RC 振荡电路 (简称 32k Pad)。使用 32k RC 时，由于误差比较大，所以对于 suspend 或者 deep retention 时间较长的应用，其时间准确性会差一些。目前 32k RC 默认支持的最大长连接不能超过 3s，一旦超过这个时间，ble_timing 会出错，造成收包时间点不准确，容易出现收发包 retry，功耗增大，甚至出现断连。而使用 32k Pad 时，误差则会小很多。

用户只需要在 main 函数开始的地方（一定要在 cpu_wakeup_init 函数之前）调用下面 API：

32k RC 调用：

```
void blc_pm_select_internal_32k_crystal(void);
```

32k Pad 调用：

```
void blc_pm_select_external_32k_crystal (void);
```

12.3 Firmware 数字签名

市场上存在一种产品恶意抄袭的方法。比如 A 客户使用了 Telink 的芯片和 SDK 研发了一款产品。A 客户的竞争对手 B 客户也使用 Telink 的芯片，拿到该产品后，可以复制同样的硬件电路设计。如果该产品的数据烧录总线未被禁用的话，B 客户有可能读到产品 Flash 上完整的 Firmware。B 客户使用同样的软硬件就可以抄袭出这个产品。

为了解决以上安全风险，SDK 支持了软件数字签名功能。其原理是利用了芯片内置 Flash 具有唯一的 UID。产品在治具烧录环节读取内置 Flash 的 16 byte UID，然后和 Firmware 上的内容进行复杂的加密运算，生产一组校验值称为 Signature，存储在 Flash 校准区对应的地址。即：

Signature = Encryption_function (Firmware, Flash_UID)

Signature 跟 Firmware 和 Flash_UID 同时相关。SDK 上程序初始化的时候读取 Firmware 和 Flash_UID 也做相同的计算，得到的结果和治具烧录的 Signature 进行对比验证，如不吻合则认为程序不合法，禁止运行。

需要强调，此功能涉及到的技术环节较多，包括治具上的配合、SDK 上相应的配置等，客户如需要，必须提前和 Telink FAE 沟通确认细节。

下面介绍此功能实现的一些技术细节。

(1) 治具端必须正确配合，包括文件配置、写脚本等，详细内容请参考 Telink testbench 的相关文档和说明，这里不具体介绍。

(2) Signature 存储地址为 Flash Calibration area 的偏移地址 0x180 连续 16 byte。

(3) SDK 上默认此功能是关闭的，如需使用，在 app_config.h 中打开下面的宏：

```
#define FIRMWARES_SIGNATURE_ENABLE          1   //firmware check
```

注意：

> SDK 上只有少数几个 project 在 main 函数初始化里添加了 Firmware 数字签名校验功能（通过搜索 FIRMWARES_SIGNATURE_ENABLE 可以看到）。如果客户使用的 project 上没有添加该功能，请务必从其他 project merge 到自己的 project。

SDK 中 code 如下所示，当数字签名不匹配时需要禁止程序运行。SDK 上使用了最简单的 while(1) 来禁止运行，这只是一个示例写法，请用户自己评估该方法是否满足需求，如不满足可以自行使用其他方法，比如让 MCU 进 deepsleep 睡眠、修改各种 data、bss、stack 段等。

```
void blt_firmware_signature_check(void)
{
        unsigned int flash_mid;
        unsigned char flash_uid[16];
        unsigned char signature_enc_key[16];
        int flag = flash_read_mid_uid_with_check(&flash_mid, flash_uid);

        if(flag==0){  //reading flash UID error
            while(1);
        }

        firmware_encrypt_based_on_uid (flash_uid, signature_enc_key);

          //signature not match
        if(memcmp(signature_enc_key, (u8*)(flash_sector_calibration +
         ↪ CALIB_OFFSET_FIRMWARE_SIGNKEY), 16)){
            while(1);   //user can change the code here to stop firmware running
        }
}
```

(4) 数字签名的计算方法 Encryption_function 是由 Telink 定义的，会同时兼顾 Firmware 内容和 Flash_UID，使用了 AES 128 加密。具体计算细节不公开，打包封库。上面 firmware_encrypt_based_on_uid 函数在 libfirmware_encrypt.a 中实现。

如果客户觉得通用的加密算法不够安全，需要使用自己的加密算法，可联系 Telink FAE 沟通解决。

12.4 SDK 版本信息

为了方便用户获取当前 SDK 和 library 的版本信息，增加了版本信息获取函数。

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

用户调用该接口应该得到至少 5 个字节，前 5 个字节显示 SDK 版本，其余为未来保留。

例如，如果调用该 API 后得到的数字为 {3,4,0,0,1}，则表示 SDK 版本为 3.4.0.0 补丁 1。

13 Debug

13.1 GPIO 模拟 UART_TX 打印方法介绍

为方便用户调试时打印信息，B80 支持 gpio 模拟打印 printf(const char *format, ...)，sprintf(char *buff, const char *format, ...)，需要在 drivers/printf.h 中定义相关信息如下所示：

```
#define DEBUG_MODE 1

#define  DEBUG_BUS      1

#if(DEBUG_BUS==DEBUG_IO)
#define PRINT_BAUD_RATE                 115200      //1M baud rate,should Not bigger than
 ↪  1Mb/s
#define DEBUG_INFO_TX_PIN               GPIO_PD3
#define TX_PIN_GPIO_EN()                gpio_set_func(DEBUG_INFO_TX_PIN , AS_GPIO);
#define TX_PIN_PULLUP_1M()              gpio_setup_up_down_resistor(DEBUG_INFO_TX_PIN,
 ↪  PM_PIN_PULLUP_1M);
#define TX_PIN_OUTPUT_EN()              gpio_set_output_en(DEBUG_INFO_TX_PIN,1)
#define TX_PIN_OUTPUT_REG               (0x503+((DEBUG_INFO_TX_PIN>>8)<<3))
#endif
```

此处默认的波特率为 115200，TX_PIN 为 GPIO_PD3，用户根据自己实际需求更改波特率及 TX_PIN。

如果用户想用更高的波特率（大于 115200，最高支持 1M），需要提高 cclk，至少更改为 24MHz 以上，在 app_config.h 更改 cclk：

```
/////////////////// Clock  //////////////////////////////////
/**
 *  @brief  MCU system clock
 */
#define CLOCK_SYS_CLOCK_HZ      24000000
```

注意：

> printf 打印的信息可能会被中断打断出现乱码，不要在中断里打印！

14 Q&A

Q．如何在 SDK 中新建自己的工程?

A．一般为了保证工程中的各种设置，我们一般会以某一个 demo 为基础，建新的工程。比如我们以 8208_ble_sample 为基础来完成一个项目的新建。

Step 1: 将 code 复制粘贴，并重命名。如下图所示，我们复制 8208_ble_sample 的代码，新命名为 Test_Demo.

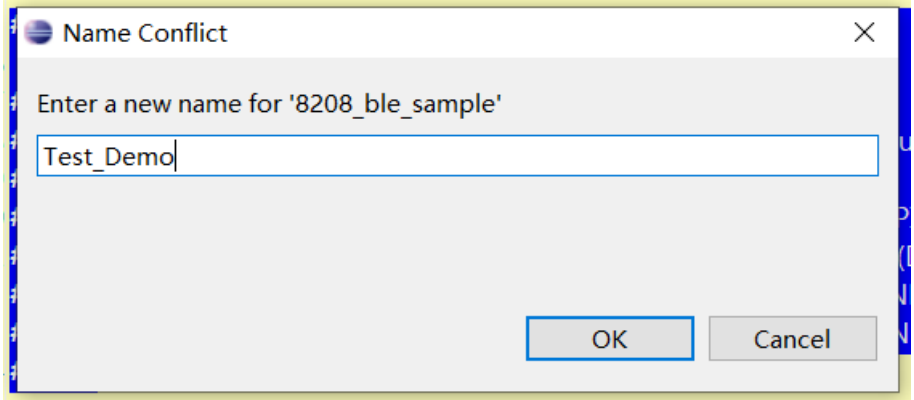


Figure 14.1: 为工程重命名

Step 2: 在项目右键的 Properties -> Settings -> Manage Configurations 中建立新的工程，比如新建 Test_Demo。

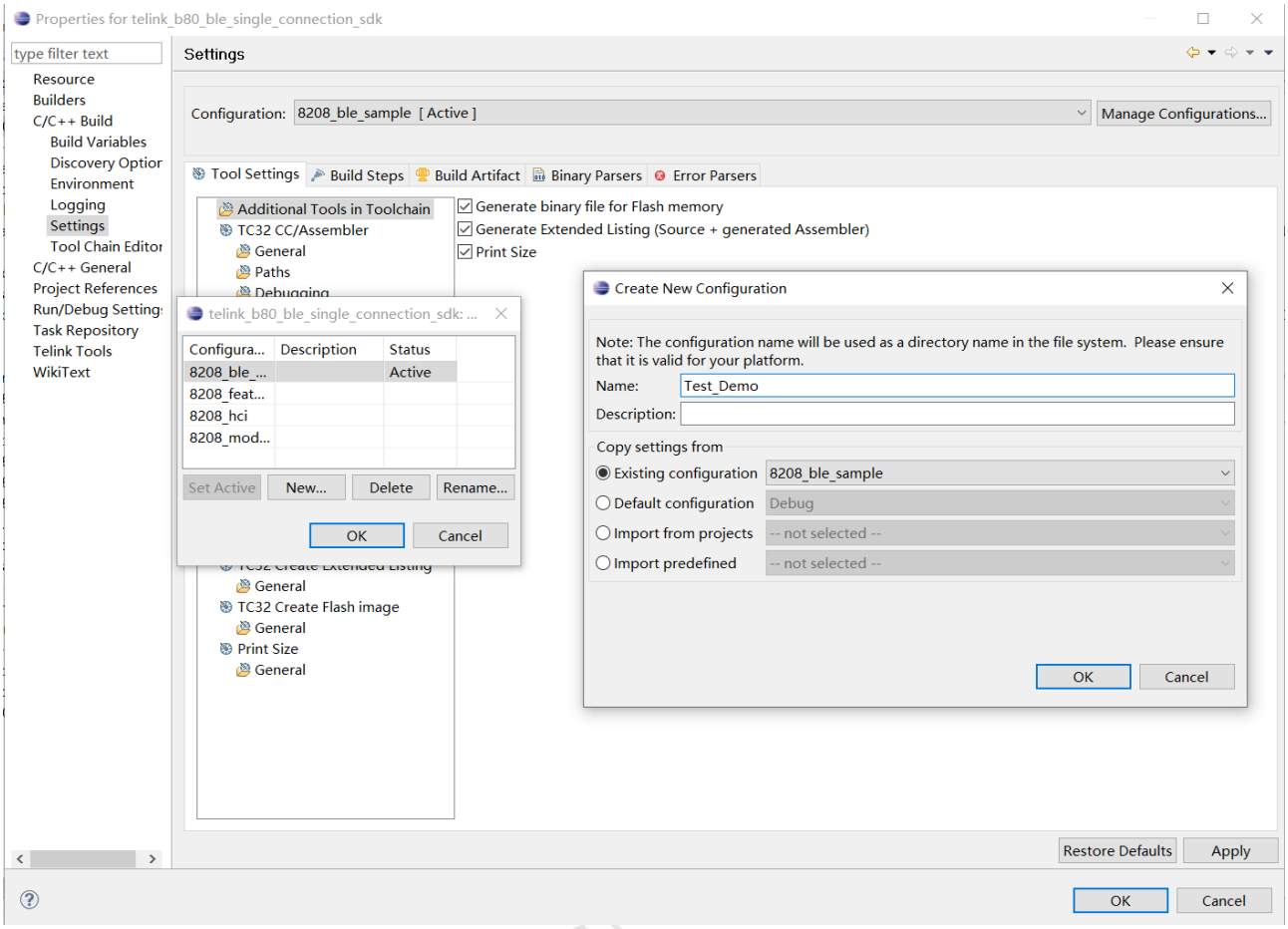


Figure 14.2: 为工程新建配置

点击 OK 之后，可以在项目列表中看到我们新建的工程，如下图。

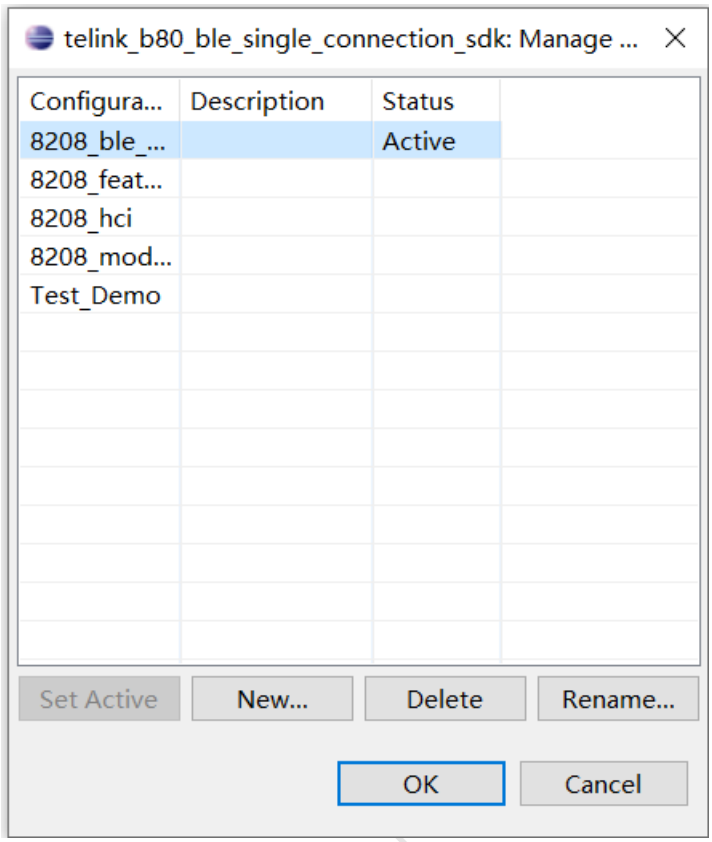


Figure 14.3: 工程列表中新建的工程

Step 3：在 Test_Demo 文件夹右键的 Resource Configurations -> Exclude from Build 中，将 Test_Demo 的设置中除了自己，将其他所有项目打勾。且在其复制源的相同设置中把 Test_Demo 打勾。如下图所示：

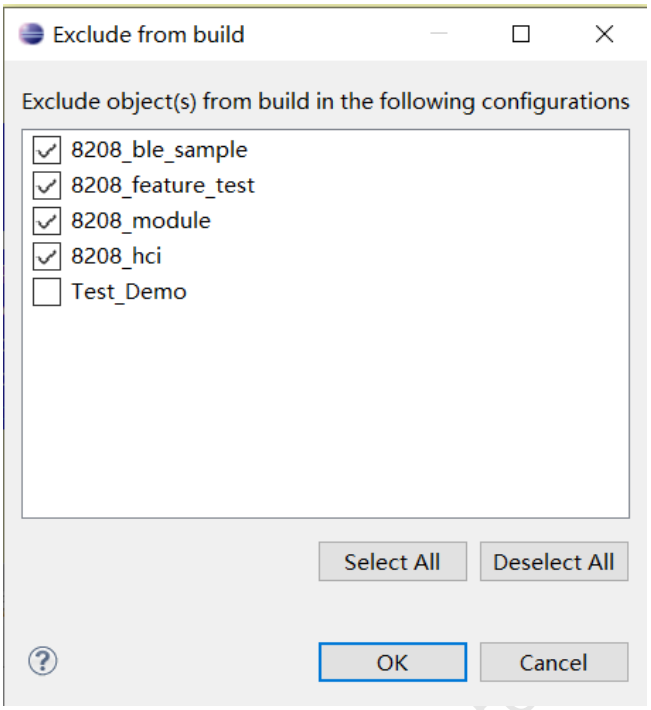


Figure 14.4: Exclude Test_Demo from build

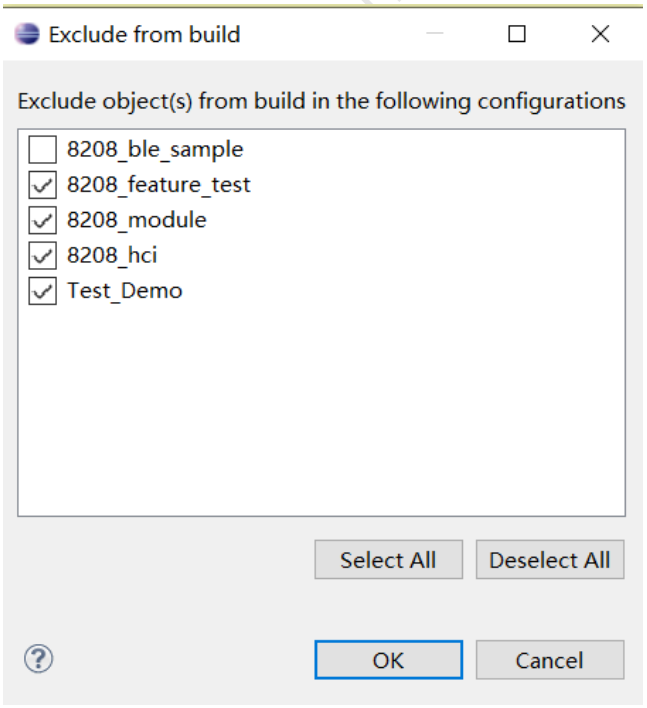


Figure 14.5: Exclude source project from build

Step 4：将 Test_Demo 属性中的 Setting-TC32 Compiler-symbols 修改为新的标志，修改后需点 Apply，如下图所示。

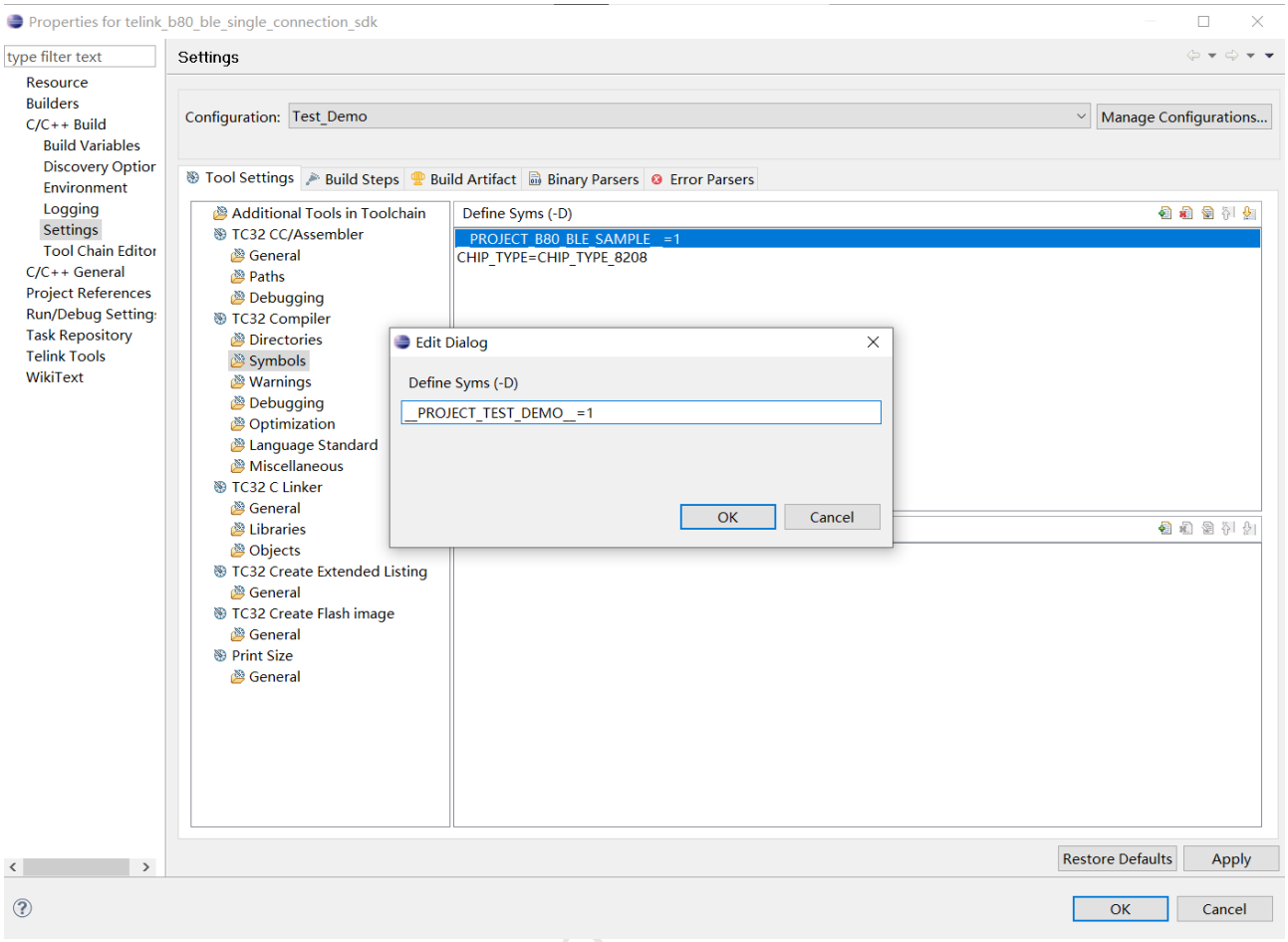


Figure 14.6: 修改编译器标志

Step 5：在 vender/common/user_config.h 中，增加新代码对应的设置包含。如图所示：

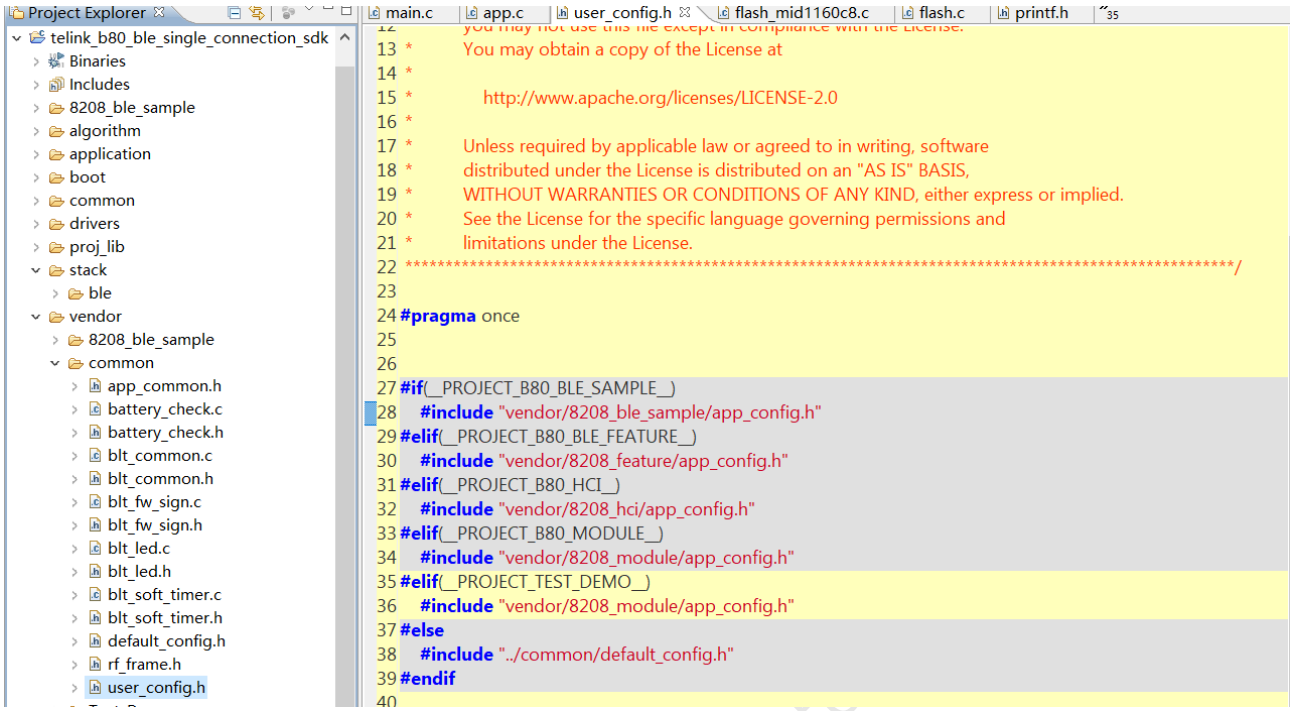


Figure 14.7: 增加新代码对应的设置

至此，新项目已经建好了。可以选择新的工程，进行 clean 和 build 使用了。

15 附录

15.1 附录 1：crc16 算法

```
unsigned short crc16 (unsigned char *pD, int len)
{
static unsigned short poly[2]={0, 0xa001};
unsigned short crc = 0xffff;
unsigned char ds;
int i,j;

for(j=len; j>0; j--)
    {
unsigned char ds = *pD++;
for(i=0; i<8; i++)
        {
            crc = (crc >> 1) ^ poly[(crc ^ ds ) & 1];
            ds = ds >> 1;
        }
    }

return crc;
}
```