TELINK 8267 BLE SDK DEVELOPER HANDLEBOOK

目录

图目录

1 软件组织架构

Telink BLE 8267 软件架构包括 APP 应用层和 BLE stack 协议栈部分，

在 Telink IDE 中导入 sdk 工程后，显示的文件组织结构如////所示。有 3 个主要的顶层文件夹：proj, proj_lib, vendor。

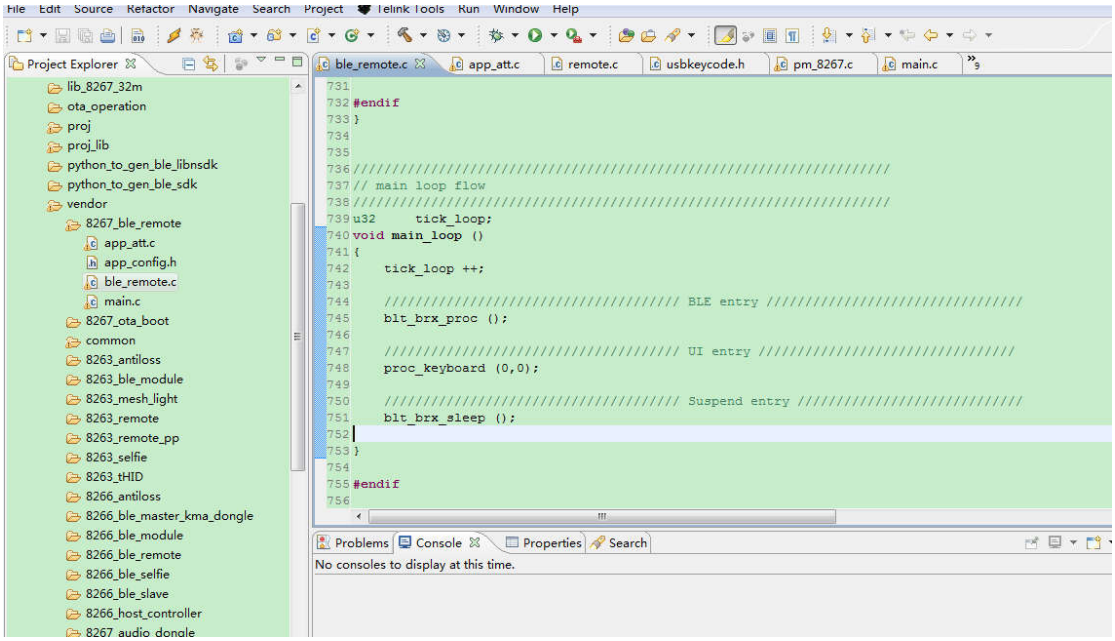

图 1 SDK 文件结构

proj：提供 MCU 相关的外设驱动程序，如 flash，i2c，usb，gpio, uart 等。

proj_lib: 提供 MCU 运行所必需的库文件包括 BLE 协议栈、RF 驱动、PM 驱动等。

vendor：用于存放用户应用层代码，8267 ble remote 示例应用在这个文件夹中。用户新建一个文件夹后，需要最基本的四个文件为：

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。

```
int main (void) {
    cpu_wakeup_init();   //MCU 最基本的硬件初始化，user 不用关注
    clock_init();        //时钟初始化，user 在 app_config.h 中配置先关参数即可
    gpio_init();         //gpio 初始化，user 在 app_config.h 中配置先关参数即可
    deep_wakeup_proc(); //deepsleep 醒来的处理，某些特殊的 case，user 需要处理
    rf_drv_init(0);      //RF 初始化，user 不用关注
    user_init ();        //ble 初始化，整个系统初始化，user 进行设定
    irq_enable();        //开全局中断
    while (1) {
        main_loop ();   //包括 ble 收发处理低功耗管理和 user 的任务
    }
}
```

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。

1.3 app_att.c

service 和 profile 的配置文件，有 Telink 定义的的 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。

1.4 用户文件

如 ble_remote.c：用户文件，用于完成系统的初始化和添加用户的 task UI。

2 MCU 基础模块

这部分主要对 8267 MCU、RF、PM 等硬件模块进行介绍，并详细说明软件设置方法。

2.1 MCU 地址空间

2.1.1 MCU 地址空间分配

Telink 8267 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 的 8M 为外部设备（列如，SRAM、寄存器空间）空间。

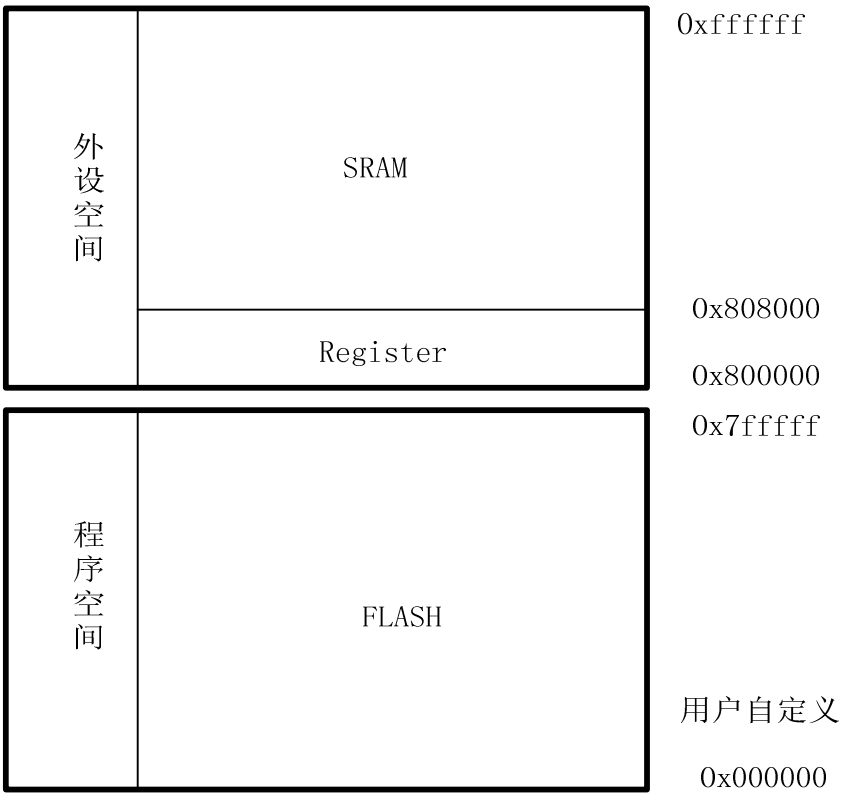

图 2　MCU 地址空间分配

2.1.2　MCU 地址空间访问

程序中对 0x00000 - 0xffffff 地址空间的访问分以下三种情况：

1) 外设空间的读写操作（register 和 sram）直接用指针实现。

```
u8    x = *(volatile u8*)0x800066;    //读 register    0x66 的值
 *(volatile u8*)0x800066 = 0x26;    //给 register    0x66 赋值
u32 = *(volatile u32*)0x808000;                //读 sram    0x8000-0x8003 地址的值
*(volatile u32*)0x808000 = 0x12345678;    //给 sram    0x8000-0x8003 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，其实质也是操作指针，详情请查看 proj/common/compatibility.h 和 proj/common/utility.h。

2) 读 flash 可以使用指针访问和 flash_read_page 函数访问

8267 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 高速缓存区域。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访问 flash 上的内容，如：

u16　x = *(volatile u16*)0x10000;　//读 flash 0x10000-0x10001 两个 byte 内容

也可以使用 flash_read 函数读取 flash 上的内容:

void flash_read_page(u32 addr, u32 len, u8 *buf);

u8 data[6] = {0 };

flash_read_page(0x11000, 6, data); //读 flash 0x11000 开始的 6 个 byte 到 data 数组。

3) 写 flash 只能用 flash_write_page 函数

flash 不能使用指针形式进行写操作，只能用下面函数

flash_write_page(u32 addr, u32 len, u8 *buf);

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。

2.2 时钟模块

系统时钟 system clock 是 MCU 执行程序的时钟，虽然 8267 的系统时钟时钟源有多种（PLL、内部 OSC、内部 RC）但是在 8267 ble SDK 中，我们只用 PLL，因为 PLL 时钟是最精准的。16M 或 12M 的外部晶振经过 MCU 内部的 PLL 硬件模块处理后，获得 192M 的 PLL 时钟（这部分硬件自动处理），通过软件配置相关寄存器进行分频获得低频的系统时钟。

2.2.1 系统时钟配置方法

main.c 中调用 clock_init 函数（详情见 proj/mcu/clock.c），对时钟源和分频系数相关寄存器进行配置，用户只需要在 app_config.h 中配置以下两个参数即可

```
////////////////// Clock  ///////////////////////////////////
#define CLOCK_SYS_TYPE      CLOCK_TYPE_PLL   //时钟源选择PLL
#define CLOCK_SYS_CLOCK_HZ  16000000         //system clock 16M
```

目前 8267 BLE SDK 推荐时钟频率为 16M，我们平时所有的调试都是基于该频率。16M 的时钟足以保证 BLE 的时序，并且功耗不会太大。

2.2.2 软件定时器

在配置好了系统时钟，并经过 clock_init 初始化后，16M 的 system clock 开始运行。基于这个时钟，可以不断读取系统时钟计数器的值，即 system clock tick，该计数器每一个时钟周期加一，长度为 32bit，即每 1/16 us 加一，最小值 0x00000000，最大值 0xffffffff。系统时

钟启动的时候，system clock tick 值为 0，到最大值 0xffffffff 需要的时间为：

(1/16) us * (2^32) 约等于 256 S，每过 256 S system clock tick 转一圈。

同理，如果系统时钟为 32M，那么 system clock tick 1/32 us 加一，转一圈需要

(1/32) us * (2^32) 约等于 128 S。

MCU 在运行程序过程中（包括 MCU 进 suspend），system clock tick 都不会停。

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确性，一般用于对误差要求不是特别苛刻的应用，设计者本身应该对于程序的运行较为清楚，知道程序的运行会在什么时刻去查询计数器是否到达预定的值。后面 PM 部分会对 8267 BLE SDK 的 MCU 运行时序进行详细的介绍。

软件定时器的实现方法为：

1) 启动计时：设置一个 u32 的变量，读取并记录当前 system clock tick，

   u32 start_tick = clock_time(); // clock_time()返回 system clock tick 值。

2) 在程序的某处不断查询当前 system clock 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，那么对于 16M 的系统时钟，查询计时是否到达的写法为：

   if((u32) (clock_time() - start_tick) > 100 * 1000 * 16)

   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，32M 系统时钟查询方法为:

   if((u32) (clock_time() - start_tick) > 100 * 1000 * 32)

   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，不再需要考虑 16 //和 32 的问题

   需要注意的是：由于 16M 时钟转一圈为 256 S，这个查询函数只适用于 256 S 以内的定时，需要超过 256 S，需要在软件上加计数器累计实现（这里不介绍），同理 32M 时钟最大计时 128 S。

   应用举例：A 条件触发（只会触发一次）的 2 S 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

2.3.1 GPIO 定义

8267 共有 6 组 42 个 GPIO，分别为：

GPIO_PA0 - GPIO_PA7、GPIO_PB0 - GPIO_PB7、GPIO_PC0 - GPIO_PC7

GPIO_PD0 - GPIO_PD7、GPIO_PD0 - GPIO_PD7、GPIO_PF0 - GPIO_PF7

程序中需要使用 GPIO 时，必须按照上面的写法定义，详情见 proj/mcu_spec/gpio_8267.h。这 42 个 GPIO 中有 7 个比较特殊，需要注意：PE4-PE7 作为 MSPI 的四个 IO，用于 flash 程序的读写，程序中不能使用；PB0 作为 SWS(single wire slave)用于 debug 和烧写 firmware，程序中一般不能使用；当需要 USB 时 PE2-PE3 作为 DM 和 DP，当不需要 USB 时，可以作为 GPIO 使用，但是程序中一定要将它们配置为一般的 GPIO 功能(因为程序默认试讲它们配置为 USB 功能)。

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。

42 个 GPIO 都包括以下状态：

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

   操作函数：void gpio_set_func(u32 pin, u32 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(u32 pin, u32 value)

   value 1 和 0 分别表示 enable 和 disable

3) datai: data input：当输入使能打开时，该值反应当前该 GPIO 管脚的电平，用于读

取外部电压。

u32 gpio_read(u32 pin)

读到低电压返回值为 0，读到高电压，返回非 0 的值（注意不一定是 1 ！）

4) oe(output enable)：输出使能

void gpio_set_output_en(u32 pin, u32 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，0 输出低电平

void gpio_write(u32 pin, u32 value)

6) 内部上下拉电阻配置：有四种状态:1M 上拉、10K 上拉、100K 下拉和 float。

void gpio_setup_up_down_resistor(u32 gpio, u32 up_down)

up_down 的四种配置为：

PM_PIN_PULLUP_1M

PM_PIN_PULLUP_10K

PM_PIN_PULLDOWN_100K

PM_PIN_UP_DOWN_FLOAT

举例 1：将 GPIO_PA4 配置为输出态，并输出高电平

PA4 默认为一般 GPIO 功能，不再设置 func

```
gpio_set_input_en(GPIO_PA4, 0);
gpio_set_output_en(GPIO_PA4, 1);
gpio_write(GPIO_PA4,1);
```

举例 2：将 GPIO_PC6 配置为输入态，判断是否读到低电平

PC6 默认为一般 GPIO 功能，不再设置 func

```
gpio_set_input_en(GPIO_PC6, 1);
gpio_set_output_en(GPIO_PC6, 0);
if(!gpio_read(GPIO_PC6)){
    ......
}
```

2.3.3 GPIO 的初始化

main.c 中调用 gpio_init 函数，会将 8267 所有的 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，42 个 GPIO 默认状态为：

1) func：除了前面介绍的 7 个为特殊功能，其他 35 个为一般 GPIO 状态

2) ie：全部为 1

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`。

以上详情见 proj/mcu_spec/gpio_8267.h 和 proj/mcu_spec/gpio_default_8267.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 的默认状态使用宏来表示的，这些宏的写法为（以 PA7 的 ie 为例）：

```
#ifndef PA7_INPUT_ENABLE
#define PA7_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 PA7 为例）：

1) 配置 func: #define PA7_FUNC AS_GPIO

2) 配置 ie: #define PA7_INPUT_ENABLE 1

3) 配置 oe: #define PA7_OUTPUT_ENABLE 0

4) 配置 dataO: #define PA7_DATA_OUT 0

5) 配置内部上下拉电阻：#define PULL_WAKEUP_SRC_PA7 PM_PIN_UP_DOWN_FLOAT

GPIO 的初始化总结：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；可以在 user_init 函数中通过 3.3.2 中 GPIO 状态控制函数加以设定；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。需要注意的是：在 app_config.h 中定义和 user_int 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。

3 BLE 模块

3.1 BLE 状态机

目前的 BLE SDK 有两个最基本的状态：广播状态（advertising state）和连接状态（connection state），当加入了低功耗（power management，简称 PM）管理后，增加一个 deepsleep 状态。

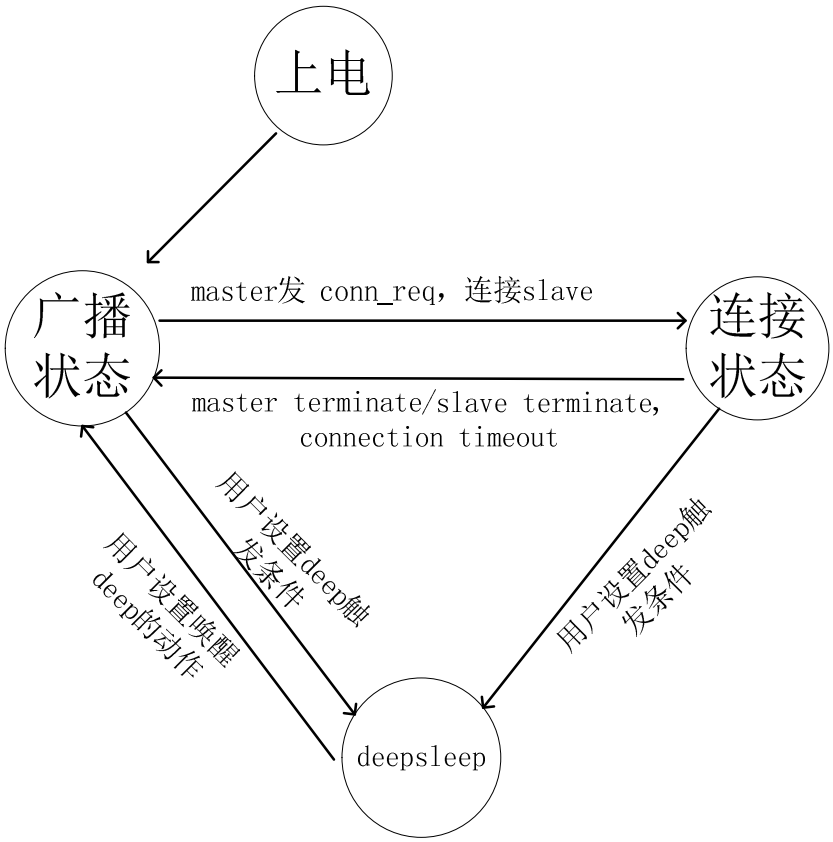

图 3 BLE 状态机

1) slave 上电后自动进入广播状态，在广播 channel 上发送广播包
2) 广播过程中，若有 master 设备发起 connection request(conn_req)，slave 和 master 建立连接，开始维护连接状态和进行数据通信
3) 在连接状态时，有三种情况回到广播状态：
   A. master 发现异常，向 slave 发送 terminate 命令，主动断开连接。slave 收到 terminate 命令，进入广播状态。
   B. slave 发现异常，向 master 发送 terminate 命令，主动断开连接。
   C. 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 supervision timeout，slave 回到广播状态。

4) 加入了低功耗管理后，用户可以自己选择是否设置 slave 从广播状态/连接状态进入 deepsleep 状态以及如何唤醒。

A. 在广播状态时，可以设置一个最大广播时间，当广播时间超过这个时间值时，slave 进入 deepsleep。可以设置唤醒 slave 的动作（如按键触发），唤醒后 slave 继续广播。后面会具体介绍。

B. 在连接状态，可以设置当长时间没有任务或事件发生时让 slave 进入 deepsleep（deepsleep 电流会比 suspend 电流更小，这样做更利于纽扣电池的长时间工作）。可以设置唤醒 slave 的动作（如按键触发），唤醒后先进入广播状态，然后快速重连，再次进入连接状态。

8267 BLE SDK 中用于控制 slave 状态的变量为 blt_state，可以在上层任意访问，注意只能读不能进行写操作，具体的状态用 blt_ll.h 中的宏来表示（虽然定义了 6 个，只有下面两个有意义）。

```
#define          BLT_LINK_STATE_ADV          0
#define          BLT_LINK_STATE_CONN         1
```

由于 user 只能读 blt_state，那么只能根据这个值来判断当前状态。deepsleep 时 MCU 停止运行，所以程序上不需要一个状态来定义。

```
if(blt_state == BLT_LINK_STATE_ADV)   //判断是否广播状态
if(blt_state == BLT_LINK_STATE_CONN)  //判断是否连接状态
```

3.2 BLE 工作时序

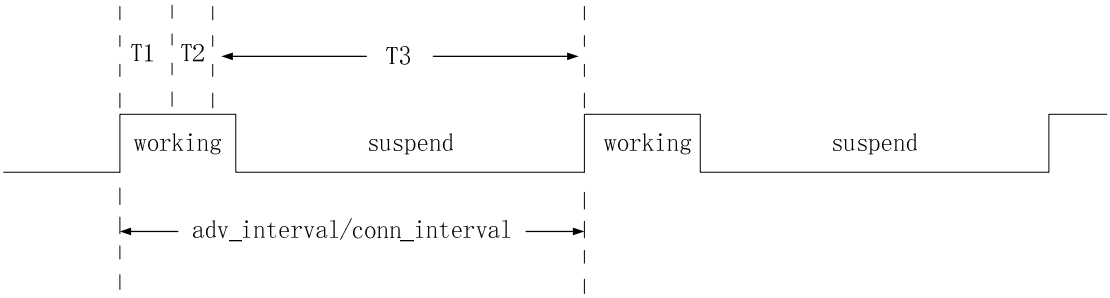

图 4 BLE 工作时序

一个具有低功耗管理的 slave 工作时序如上图所示。BLE slave 在每一个 main_loop 都是一个 adv_interval/conn_interval：当处于广播状态时，每个 main_loop 的时间周期为一个

adv_interval（后面介绍使用 blt_set_adv_interval ()函数设置这个时间）；当 slave 处于连接状态时，每个 main_loop 的时间周期为一个 conn_interval（与 master 的连接参数，后面介绍）。

每个 adv_interval/conn_interval 里，working 的时间占非常小的比例，绝大部分时间处于 suspend 状态（如果去掉功耗管理，软件上在 suspend 时间内会让程序只做纯延时操作，空等）。根据 8267 BLE SDK 的 main_loop 的写法可以将每个 adv_interval/conn_interval 划分为三个部分,T1、T2 和 T3。

```
void main_loop ()
{
    ////////////////////// BLE entry /////////////////////////////
    blt_brx_proc ();
    ////////////////////// UI entry /////////////////////////////
    proc_keyboard (0,0); //仅供参考

    ////////////////////// Suspend entry /////////////////////
    blt_brx_sleep ();
}
```

1) T1 对应 BLE entry 部分的 blt_brx_proc ()函数。每个 main_loop 的开头位置是上一个 interval 的 suspend 醒来的时刻，也就是 blt_brx_sleep ()执行完跳出来的时刻。blt_brx_proc ()函数负责处理 BLE slave 所有 RF 相关工作，在广播状态时处理发广播包、扫描 master 的 scan_req 并回 scan_rsp、扫描 master 的 conn_req 并相应连接等工作，在连接状态时，处理监听 master 的连接包、将 user 的数据发送给 master 等工作。
2) T2 对应 UI entry，是供 user 发挥的部分，user 在这里处理自己的任务并将数据 push 到 BLE 的数据 buffer 里(后面会介绍如何实现)，当 BLE buffer 里有时，在下一个 main_loop 的 blt_brx_proc ()部分，MCU 会自动将数据发送给 master。
3) T3 对应 Suspend entry 的 blt_brx_sleep ()。该函数处理 suspend 的相关操作，包括设置 suspend 时间、设置 suspend 的唤醒源、设置是否在当前 mian_loop 进入 deepsleep 等，程序执行到进入这个函数后，会进入 suspend 直到设定的该醒的时间点才会醒来，此时程序跳出该函数，开始执行下一轮的 main_loop。后面低功耗模块会详细介绍。

3.3 基于事件触发的回调

虽然 8267 BLE SDK 已经尽量将 BLE 基本的 RF 收发、状态切换等工作放在 BLE stack 里完成，user 只需要关注上层的任务，但还是有些事件的发生需要 user 去关注，并在这些事件发生时去做一些相应的操作，以满足上层应用的需要。为了满足这类操作，8267 BLE SDK 提供了相关的回调函数。回调函数的原型说明为：

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

其中 e 是底层事件编号；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，下面会具体说明。

底层事件编号目前暂时最多支持 20 个，在 blt_ll.h 中宏宏来定义，目前有 17 个：

```
#define          BLT_EV_FLAG_BEACON_DONE          0
#define          BLT_EV_FLAG_ADV_PRE              1
#define          BLT_EV_FLAG_ADV_POST             2
#define          BLT_EV_FLAG_SCAN_RSP             3
#define          BLT_EV_FLAG_CONNECT              4
#define          BLT_EV_FLAG_TERMINATE            5
#define          BLT_EV_FLAG_WAKEUP               6
#define          BLT_EV_FLAG_BRX                  7
#define          BLT_EV_FLAG_SLEEP                8
#define          BLT_EV_FLAG_IDLE                 9
#define          BLT_EV_FLAG_EARLY_WAKEUP         10
#define          BLT_EV_FLAG_CHN_MAP_REQ          11
#define          BLT_EV_FLAG_CONN_PARA_REQ        12
#define          BLT_EV_FLAG_CHN_MAP_UPDATE       13
#define          BLT_EV_FLAG_CONN_PARA_UPDATE     14
#define          BLT_EV_FLAG_BOND_START           15
#define          BLT_EV_FLAG_SET_WAKEUP_SOURCE    16
```

初始化的时候注册回调函数的接口为：

```
void blt_register_event_callback (u8 e, blt_event_callback_t p)
```

e 为底层事件编号，用上面的宏来填充即可，p 是回调函数，user 事先定义好的 blt_event_callback_t 函数的指针传入即可。

以 BLT_EV_FLAG_CONNECT 来举例说明注册函数的用法，该事件表示的是：slave 在广播状态发广播包时，master 发送 conn_req 请求连接，slave 收到该 conn_req 并处理，进入 connection 状态，此时 BLE SDK 检查一下 BLT_EV_FLAG_CONNECT 回调事件编号是否已经被注册，若被注册了，就调用注册的函数进行相关的操作。这个操作就是 user 自己定义的，user

希望在 slave 进入连接的时候进行一些记录或是相关的设置。

```
void task_connect (u8 e, u8 *p)
{
    blt_conn_para_request (6, 6, 99, 400); //申请一个新的连接参数
    key_active_tick = clock_time();   //记录conenct发生时的system tick值
}
blt_register_event_callback (BLT_EV_FLAG_CONNECT, &task_connect);   //注册 task_connect
```

根据以上设定，BLE slave 每次收到 conn_req 并进入连接状态时都会调用一次 task_connect 函数。

下面对所有的底层事件进行详细的说明，以便 user 熟练掌握回调函数的用法。每个事件说明其触发的条件，并介绍回调函数 typedef void (*blt_event_callback_t)(u8 e, u8 *p)中回传的指针 p 所指向的数据是什么。

3.3.1 BLT_EV_FLAG_ADV_PRE

1) 事件触发条件：slave 处于广播状态，在每个 T_advEvent 三个 channel 发广播之前，触发该事件。注意触发该事件时，程序先去检测该事件的回调函数是否已经被注册，若没有注册，什么都不做；若已经注册，才去执行回调函数，下面所有事件都是这样。
2) 回调函数回传指针 p：空指针。

3.3.2 BLT_EV_FLAG_ADV_POST

1) 事件触发条件：slave 处于广播状态，在每个 T_advEvent 三个 channel 发广播完成后，触发该事件。
2) 回调函数回传指针 p：空指针。

3.3.3 BLT_EV_FLAG_SCAN_RSP

1) 事件触发条件：slave 处于广播状态，收到 master 的 scan_req，并进行响应，回一个 scan_rsp 时触发该事件。
2) 回调函数回传指针 p：空指针。

3.3.4 BLT_EV_FLAG_CONNECT

1) 事件触发条件：slave 处于广播状态时收到 master 的 conn_req，响应这个请求，建立与 master 的连接进入连接状态时触发该事件。

2) 回调函数回传指针 p：p 指向一片内存，该内存存储了一个结构体如下，p 指向 dma_len 的第一个 byte。4 个 byte 的 dma_len 是 8267 特殊的数据，user 不用去关注，后面剩余的从 type 开始的数据 master 的 conn_req 包所有的内容拷贝而来，如所示为 BLE 协议栈里 conn_req 的格式，请根据此格式进行相应的操作。

```
typedef struct{
    u32 dma_len;
    u8  type;  //RA(1)_TA(1)_RFU(2)_TYPE(4): connect request PDU
    u8  rf_len;       //LEN(6)_RFU(2)
    u8  scanA[6];     //
    u8  advA[6];      //
    u8  accessCode[4];       // access code
    u8  crcinit[3];
    u8  winSize;
    u16 winOffset;
    u16 interval;
    u16 latency;
    u16 timeout;
    u8  chm[5];
    u8  hop;              //sca(3)_hop(5)
}rf_packet_connect_t;
```

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

图 5 BLE 协议栈 conn_req 格式定义

rf_packet_connect_t 前两个 byte 是 16 bit 的 header，后面从 scanA 开始跟上图所示的 conn_req 格式是完全对应的。

3.3.5 BLT_EV_FLAG_TERMINATE

1) 事件触发条件：slave 处于连接状态，master 发送 terminate，主动端开连接，slave 收到 terminate 命令时触发该事件，连接断开，回到广播状态；slave 和 master 的 RF 通信出

现问题，slave 连续一段时间收不到 master 的包，连接超时（supervision timeout）时也触发该事件，连接断开，回到广播状态。

2) 回调函数回传指针 p：p 指向一个 u8 型的变量 blt_conn_terminate，该变量可以来识别该事件是上述哪个条件触发的，若 blt_conn_terminate 为 1 说明 master 发送了一个 terminate 命令导致 slave 触发连接断开，若 blt_conn_terminate 为 0 说明 slave 收不到包超时导致的连接断开。

3.3.6 BLT_EV_FLAG_WAKEUP

SDK 中暂时没定义。

3.3.7 BLT_EV_FLAG_BRX

1) 事件触发条件：

前面介绍过，slave 在执行 blt_brx_sleep 函数时进入 suspend，而在 suspend 之前程序会计算好 suspend 醒来的时间，以保证醒来后能够顺利的收到 master 的包。suspend 醒来后，程序正好退出 blt_brx_sleep 函数，进入下一轮的 main_loop 执行 blt_brx_proc 函数。blt_brx_proc 函数会立刻将 MCU 设置为 RX 听包模式。由于 slave 计算的 master 的发包的时间点是预估的，且随着 interval 的变大、latency 的启用、晶振精度的影响等会导致预估的时间点会有一定的误差，所以设置 RX 听包的时间点离实际 master 包到来的时间点之间会有一些余量，在这个时间内程序上只是空等，不做其他操作。另外 8267 MCU 接收 master 的包然后回包等操作都是底层硬件自动完成，在 RF 数据收发过程中，MCU 几乎不会执行其他什么操作，只是在不断查询底层的硬件的操作是否已经完成。

基于以上两点，我们可以将程序上纯粹等待的时间利用起来，在这个时间内做一些操作并不影响底层的 RF 收发。需要注意到是，一般的数据量较小的 BLE 应用并不需要利用这段时间，在 main_loop 的 UI entry 处做各种任务处理即可满足，对于一些数据量很大，且 user 任务耗时非常大的应用（如音频任务），需要将此事件利用起来，以保证效率和安全性。

blt_brx_sleep 函数内部，当设置了 slave 听包开始后，触发该事件。语音任务需要注册该回调函数，从而在这里完成，而不用在 main_loop 的 UI entry 处理：

```
blt_register_event_callback (BLT_EV_FLAG_BRX, &task_audio);
```

2) 回调函数回传指针 p：p 指向 u16 类型的变量 blt_conn_inst，该变量是 BLE 协议栈里的记录 master 和 slave 连接包事件的计数值 instant。

3.3.8 BLT_EV_FLAG_SLEEP

1) 事件触发条件：slave 执行 blt_brx_sleep 函数，进入该函数后，触发该事件。当 user 需要在 slave 进入 suspend 之前做一些特殊的处理时，可以调用该事件的回调函数。

2) 回调函数回传指针 p：空指针。

3.3.9 BLT_EV_FLAG_IDLE

没有 PM 的时候才会用到，不介绍。

3.3.10 BLT_EV_FLAG_EARLY_WAKEUP

1) 事件触发条件：slave 进入 suspend 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 8267 suspend 状态下的 timer 计时实现的），那么会存在一个问题：如果 suspend 的时间过长，user 的任务必须到 suspend 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 suspend 时间较大，如 400ms、1 S 等（suspend 时间在启用 latency 的时候会达到这些值的），按键就会丢掉。那么需要在 MCU 进入 suspend 之前判断当前 suspend 的时间，如果时间过长，设置 suspend 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 被非 timer 的其他唤醒源提前唤醒时，触发 BLT_EV_FLAG_EARLY_WAKEUP 事件。

2) 回调函数回传指针 p：指向一个 u8 型变量 wakeup_src，wakeup_src 变量记录了被唤醒的这次 suspend 被设置了哪些唤醒源，由 pm_8267.h 可看到四个唤醒源（只用到前三个），唤醒源在 PM 模块详细介绍。

```
enum {
    PM_WAKEUP_PAD   = BIT(4),
     PM_WAKEUP_CORE  = BIT(5),
    PM_WAKEUP_TIMER = BIT(6),
    PM_WAKEUP_COMP  = BIT(7),
};
```

3.3.11 BLT_EV_FLAG_CHN_MAP_REQ

1) 事件触发条件：slave 在连接状态，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。

2) 回调函数回传指针 p：p 指向下面频点列表数组的首地址。

```
u8   blt_conn_chn_map[5]
```

注意回调函数执行时 p 指向的 blt_conn_chn_map 是还没有更新的老 channel map。

blt_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：

blt_conn_chn_map[0]的 bit0-bit7 分别表示 channel0-channel7，

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

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

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

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

3.3.12 BLT_EV_FLAG_CHN_MAP_UPDATE

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

2) 回调函数回传指针 p：p 指向 blt_conn_chn_map 的首地址，此时的 blt_conn_chn_map 是更新以后的新的 map。

3.3.13 BLT_EV_FLAG_CONN_PARA_REQ

3) 事件触发条件：slave 在连接状态，master 需要更新当前连接的参数，发送 LL_CONNECTION_UPDATE_REQ 命令，slave 收到这个请求后触发该事件，此时还没有处理这个请求。

4) 回调函数回传指针 p：p 指向一片内存区域，该区域的值是从 master 的 conn_update_req 包中的数据拷贝而来。

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

图 6　BLE 协议栈 LL_CONNECTION_UPDATE_REQ 格式

p 指针指向上图所示的 WinOffset 开始的地方（Winsize 忽略）

p[1]p[0]:offset

p[3]p[2]:interval

p[5]p[4]:latency

p[7]p[6]:timeout

p[9]p[8]:instant

3.3.14 BLT_EV_FLAG_CONN_PARA_UPDATE

1) 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。

2) 回调函数回传指针 p：p 指向 u32 型的 blt_conn_interval，该值是已经更新的 connection interval 的值，注意是换算为 system tick 的值，如新的 connection interval 为 10ms，那么在 16M 系统时钟下，

blt_conn_interval = 10*1000*16 = 160000

3.3.15 BLT_EV_FLAG_BOND_START

8267 SDK 暂时未使用。

3.3.16 BLT_EV_FLAG_SET_WAKEUP_SOURCE

1) 事件触发条件：slave 执行 blt_brx_sleep 函数时，进入 suspend 之前触发该事件。用户可以在这里添加当前 suspend 的唤醒源（如加上 GPIO CORE 唤醒）、决定是否进入 deepsleep 等。在 PM 模块详细介绍。

2) 回调函数回传指针 p：空指针。

3.4 BLE 初始化设置

基于 8267 BLE SDK 的应用所有的初始化操作都在 user_init ()里进行，包括 BLE 的初始化，PM（power management）的初始化，用户自定义任务的初始化等。

3.4.1 MAC 地址

BLE MAC address 的长度为 6 个 byte，目前 8267 BLE SDK 将 MAC address 的地址设在 flash 的 0x1e000-0x1e005 位置。最终的产品在烧录时，泰凌会通过自己的治具系统将 MAC address 烧在这个区域，具体的 MAC address 需要用户自己从蓝牙组织获取。

目前在调试阶段，SDK 对 MAC address 处理的方法为：slave 设备第一次上电时，读到 flash 的 0x1e000 处的 MAC address 为空，将 MAC address 设为 0xC7E4E3E2E1xx（前 5 个 byte 固定，最后一个 byte 随机生成），然后将这 6 个 byte 写到 flash 的 0x1e000-0x1e005。以后每次上电时，读到 flash 的 0x1e000 处的 MAC address 已经存在，直接使用该地址，从而确保 slave device 的 MAC address 保持一致。相关代码如下，参考本文档之前对 flash 访问的介绍，加以理解。用户也可以根据自己的设计的这个方法进行修改。

```
#define                 CFG_ADR_MAC                 0x1e000
u8    tbl_mac [] = {0xe1, 0xe1, 0xe2, 0xe3, 0xe4, 0xc7};
u32 *pmac = (u32 *) CFG_ADR_MAC;

if (*pmac != 0xffffffff)
{
      memcpy (tbl_mac, pmac, 6);
}
else
{
      tbl_mac[0] = (u8)rand();
      flash_write_page (CFG_ADR_MAC, 6, tbl_mac);
}
```

3.4.2 广播事件的设定

BLE 协议栈里 Advertising Event（简称 Adv Event）如下图所示，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。

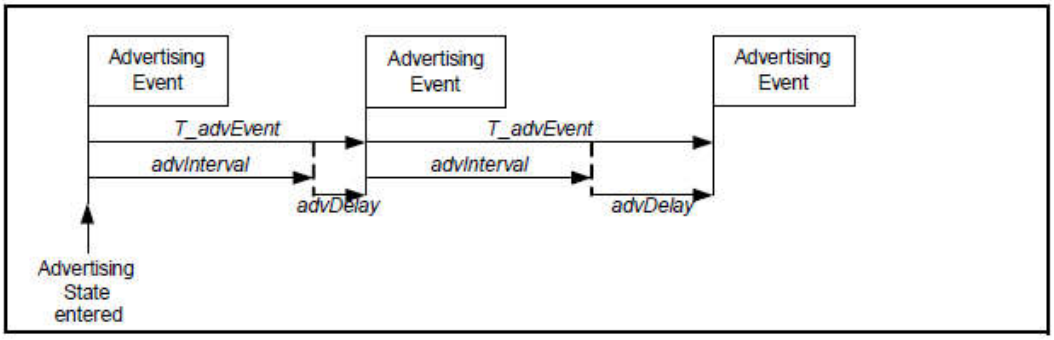

图 7　BLE 协议栈里 Advertising Event

本文档前面图 4　BLE 工作时序图 4 所示的 BLE 工作时序中，每一个 main_loop 的时间（T1+T2+T3）对应一个 T_advEvent。8267 BLE SDK 提供了对 Adv Event 的设定：

1) 广播 channel 的设定：理论上最终的产品 Adv Event 应该在在三个广播 channel 上都发包，以保证 master 设备能够扫描到 slave 设备并建立连接。在前期的调试中，为了方便 BLE sniffer 进行抓包分析（一个 sniffer 只能在一个 channel 上抓包），我们通常将 Adv Event 设为在某个 channel 发包。设定的函数为：

void blt_set_adv_channel (u8 m);

设定的参数有四种选择，在 blt_ll.h 中可以看到，分别代表只在 channel 37 上发包、只在 channel 38 上发包、只在 channel 39 上发包、在三个 channel 上都发包。

```
#define          BLT_ENABLE_ADV_37        BIT(0)
#define          BLT_ENABLE_ADV_38        BIT(1)
#define          BLT_ENABLE_ADV_39        BIT(2)
#define          BLT_ENABLE_ADV_ALL    (BLT_ENABLE_ADV_37 |
                                  BLT_ENABLE_ADV_38 | BLT_ENABLE_ADV_39)
```

目前SDK中默认的设定为 blt_set_adv_channel (BLT_ENABLE_ADV_37)，这样就可以在sniffer上只对channel 37进行抓包。

2) 广播时间的设定

8267 BLE SDK 提供的设定每一个 AdvEvent 时间 T_advEvent 的函数为:

void blt_set_adv_interval (u32 t_us);　//实参的单位为 us

目前 SDK 中默认的设定为 blt_set_adv_interval (30000);

用户可以自己对该值 30 ms 进行修改。根据图 4 所示，由于广播时 MCU working 时间是固定的，所以 T_advEvent 越大，suspend 时间越长，那么整体电流就会越小。T_advEvent 越小，在单位时间内，广播包的数量就会越多，那么被 master scan 并连上的速度就越快（这一点对于 deepsleep 醒来后的快速回连比较重要）。

3.4.3 广播包、scan response 包的格式

8267 BLE SDK 通过下面的的函数将设定 MAC address、广播包和 scan response 包。

blt_init (tbl_mac, tbl_adv, tbl_rsp);

tbl_mac 前面已经介绍。tbl_adv 是广播包的指针，tbl_rsp 是 scan response 的指针，这两个指针需要用户自己再内存上定义数组来实现，用户可以根据 BLE 协议栈和 SDK 中 RF 包参考格式，来完成包格式的定义。

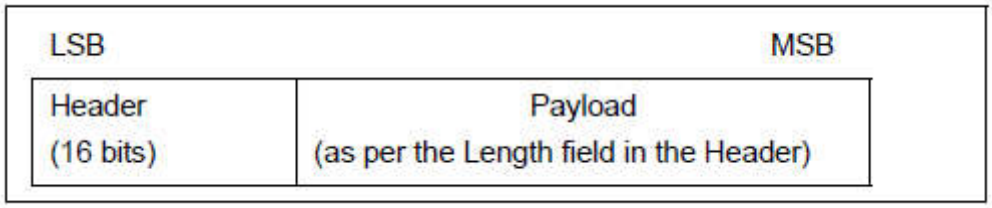

图 8 BLE 协议栈广播包格式

BLE 协议栈里，广播包的格式：前两个 byte 是 head，后面是 Payload

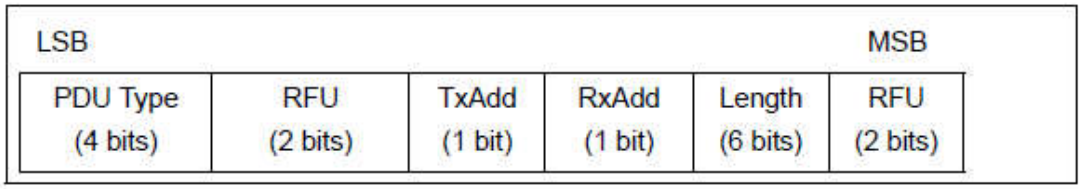

图 9 BLE 协议栈广播包 header 结构

BLE 协议栈里，广播包 heade 结构如上所述，其中我们需要关注的是第一个 bytes 的低 4 bit 的 PDU Type 和第二个 byte 低 6 bit 的 Length，其他 bit 在非特殊情况下都是 0。PDU Type 见下图定义，Length 的值是 PDU 的长度，只要在定义好数据包之后数一下即可。

PDU Type $b_3b_2b_1b_0$	Packet Name
0000	ADV_IND
0001	ADV_DIRECT_IND
0010	ADV_NONCONN_IND
0011	SCAN_REQ
0100	SCAN_RSP
0101	CONNECT_REQ
0110	ADV_SCAN_IND
0111-1111	Reserved

图 10　BLE 协议栈 PDU Type 定义

上图所示的 PDU type，ADV_IND 为非直接广播包，我们的 SDK 使用该类型；SCAN_RSP 为 scan response 包。

```
u8  tbl_adv [] =
        {0x00, 25,
         0xef, 0xe1, 0xe2, 0xe3, 0xe4, 0xe5,       //mac address
         0x05, 0x09, 't', 'H', 'I', 'D',
         0x02, 0x01, 0x05,                         // BLE limited
discoverable mode and BR/EDR not supported
         0x03, 0x19, 0x80, 0x01,                   // 384, Generic
Remote Control, Generic category
         0x05, 0x02, 0x12, 0x18, 0x0F, 0x18,       // incomplete list of
service class UUIDs (0x1812, 0x180F)
        };
```

分析：第一个 0x00 表示的是 PDU Type 为 ADV_IND，第一个 25 是后面 PayLoad 的长度，紧接着的 6 个 byte 的 MAC address 在这里随便怎么写都无所谓，因为后面 blt_init（）函数会将正确的 MAC address 拷贝到这个区域。剩余的数据内容的含义请参考 BLE 标准文档《CSS v4》。注意在广播包中设置设备的名称为"tHID"。

```
u8  tbl_rsp [] =
        {0x0x04, 14,                                    //type len
         0xef, 0xe1, 0xe2, 0xe3, 0xe4, 0xe5,          //mac address
         0x07, 0x09, 't', 'S', 'e', 'l', 'f', 'i'     //scan name " tSelfi"
```

};

分析：第一个 0x04 表示的是 PDU Type 为 SCAN_RSP，第一个 14 是后面 PayLoad 的长度，紧接着的 6 个 byte 的 MAC address 在这里随便怎么写都无所谓，因为后面 blt_init（）函数会将正确的 MAC address 拷贝到这个区域。在 scan_rsp 包中设置设备的名称为"tSelfi"。

上面在广播包和 scan_rsp 包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：

一些设备只看广播包，那么显示的设备名称为"tHID"

一些设备看到广播后，发送 scan_req，并读取回包 scan_rsp，那么显示的设备名称可能就会是"tSelfi"

用户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，对获取设备的 Dev_name，连上后会根据那里的设置来显示设别名称，后面 Attribute Table 部分在介绍。

3.4.4 BLE packet 能量设定

8267 BLE SDK 提供了设定 BLE packet 能量设定的接口:

```
void	rf_set_power_level_index (int level);
```

level 值的选取在 rf_drv_8267.h 中给出了一组枚举变量，从名字可以看出设定的能量值的大小：

```
enum {
RF_POWER_8dBm = 0,
RF_POWER_4dBm = 1,
RF_POWER_0dBm = 2,
RF_POWER_m4dBm    = 3,
RF_POWER_m10dBm   = 4,
RF_POWER_m14dBm   = 5,
RF_POWER_m20dBm   = 6,
RF_POWER_m24dBm   = 8,
RF_POWER_m28dBm   = 9,
RF_POWER_m30dBm   = 10,
RF_POWER_m37dBm   = 11,
RF_POWER_OFF  = 16,
};
```

3.4.5 注册回调函数

根据前面对基于时间触发的回调的介绍，初始化的时候设置好需要回调的函数，在函数里写好需要做哪些操作。如：

```
blt_register_event_callback (BLT_EV_FLAG_CONNECT, &task_connect);
blt_register_event_callback (BLT_EV_FLAG_TERMINATE, &ble_remote_terminate);
blt_register_event_callback (BLT_EV_FLAG_BRX, &task_audio);
blt_register_event_callback (BLT_EV_FLAG_EARLY_WAKEUP, &proc_keyboard);
blt_register_event_callback(BLT_EV_FLAG_CONN_PARA_REQ, &conn_para_update_req_proc);
```

3.4.6 ATT 和 SECURITY 初始化

使用 my_att_init 函数对 ATT table 和 SECURITY 进行初始化。

```
void my_att_init ()
{
    blt_set_att_table ((u8 *)my_Attributes);
#if(BLE_REMOTE_SECURITY_ENABLE)
    blt_smp_func_init ();
#endif
}
```

blt_set_att_table ((u8 *)my_Attributes)将 user 定义的 my_Attributes 设为 BLE slave 的 Attribute Table 即可，在 Attribute 部分详细说明。

blt_smp_func_init ()对 slave 的 Pairing 和 Security 进行初始化，使用 app_config.h 中的宏 BLE_REMOTE_SECURITY_ENABLE 来定义是否对其进行初始化，只要初始化过，后面的 Pairing 和 Security 所有的操作都由 SDK 自动完成，user 不用关注。

3.5 更新连接参数

3.5.1 slave 请求更新连接参数

在 BLE 标准协议栈中，slave 通过 l2cap 层的 CONNECTION PARAMETER UPDATE REQUEST 命令向 master 申请一套新的连接参数，该命令格式如下：

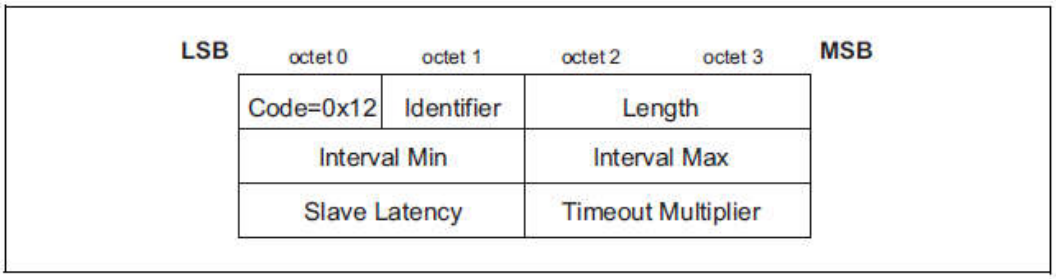

Figure 4.22: Connection Parameters Update Request Packet

图 11　BLE 协议栈中 Connection Para update Req 格式

Code 、Identifier 和 length 的定义见<<Core_v4.1_BLE_spec.pdf>> P1696。

data 区域的四个数据 Interval Min、Interval Max、Slave Latency、Timeout Multiplier 详细说明见<<Core_v4.1_BLE_spec.pdf>> P1718。

8267 BLE stack 提供了 slave 主动申请更新连接参数的接口，用来向 master 发送上面这个 CONNECTION PARAMETER UPDATE REQUEST 请求：

void blt_conn_para_request (u16 min_interval, u16 max_interval, u16 latency, u16 timeout);

以上四个参数跟 CONNECTION PARAMETER UPDATE REQUEST　data 区域中四个参数正好对应。注意 interval min 和 interval max 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）。

只要在程序中调用上面函数，8267 SDK 会根据以上设置的四个参数，自动生成 l2cap 层的 CONNECTION PARAMETER UPDATE REQUEST，命令发送出去。

当 user 调用此函数，在上层申请一套新的连接参数时，8267 BLE SDK 向 master 发送该命令的时间有如下限制：

slave 收到 master 的 conn_req 从广播状态切换到连接状态时，会记下切换时时间点 system　tick 值，该值在 SDK 中用 u32 型变量 blt_conn_start_tick 表示（必要的时候，user 在上层也可以使用该变量进行一些操作，前提是要使用 extern 来声明），若 user 调用 blt_conn_para_request 时的时间点距离 blt_conn_start_tick 已超过 1 S，SDK 会立刻将该命令发出去；若不超过 1 S，会一直等到超过 1 S 的时间才发送此命令。这样做的原因是 slave 刚刚建立和 master 的连接时，master 首先会来读 slave 的 GATT 相关信息，8267 BLE SDK 有意让这些 GATT 信息的交换完成后再申请新的连接，以避免切换连接参数造成的不稳定影响了 GATT 信息的交换。

目前的参考 code 中，建立连接时的 task_connect 回调中申请了新的连接，但是要到 1 S 以后才会发送该命令，user 可以通过 BLE 抓包看到这个。

应用举例：

```
void task_connect (u8 e, u8 *p)
{
    blt_conn_para_request (6, 6, 99, 400);    //interval=7.5 ms latency=99 timeout = 4 S
    key_active_tick = clock_time();
}
```

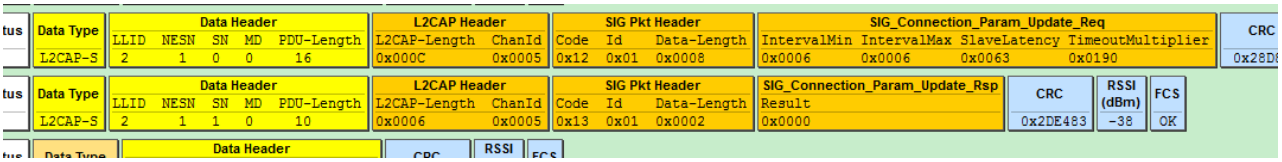

图 12　BLE 抓包 conn para update req 命令

3.5.2　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，详见<<Core_v4.1_BLE_spec.pdf>> P1720。下图所示为改名格式及 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 andriod、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，只能看 user 在平时的 master 兼容性测试中去慢慢总会归纳。图 12 所示抓包显示的 master 接受了申请。

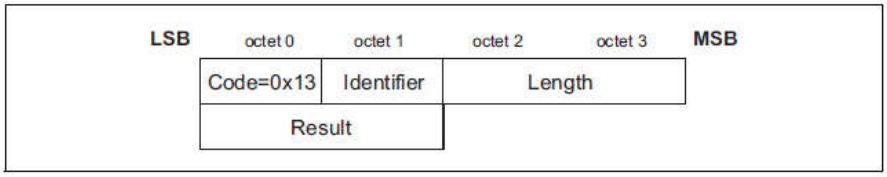

Figure 4.23: Connection Parameters Update Response Packet

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

图 13　BLE 协议栈中 conn para update rsp 格式

3.5.3 master 更新连接

slave 发送 conn para update req，并且 master 回 conn para update rsp 接受申请后，master 回发送 link layer 层的 LL_CONNECTION_UPDATE_REQ 命令，见下图抓包所示，该请求详细定义见<<Core_v4.1_BLE_spec.pdf>> P2514。

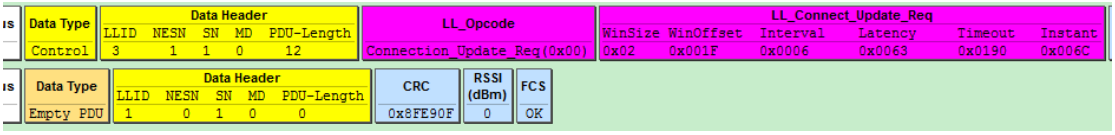

图 14　BLE 协议栈中 ll conn update req 格式

slave 收到此更新请求后，mark 最后一个参数 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 端各个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。

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 值时钟是相等的，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。

3.6 Attribute Protocol (ATT)

3.6.1 Attribute 基本内容

Attribute Protocol 定义了两种角色：server 和 client。 8267 BLE SDK 中，slave 是 server，对应的 andriod、IOS 设备是 client。server 的一组的 Attribute 可被 client 访问。

8267 BLE SDK 中需要了解和掌握的 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 UUID 长度为定义为 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。8267 BLE stack 中已经定义了一些标准的 UUID，分布在以下文件中：proj_lib/ble_l2cap/hids.h、proj_lib/ble_l2cap/gatt_uuid.h、proj_lib/ble_l2cap/service.h。

Telink 私有的一些 profile（OTA、MIC、SPEAKER 等），标准蓝牙里面不支持，在 proj_lib/ble_l2cap/service.h 中定义这些私有设备的 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 GATT 协议获取 slave 的 Attribute Table 后，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 8267 BLE stack 中，Attribute Value 用指针和指针所指区域的长度来描述。

3.6.2 Attribute Table

8267 ble stcak 中，Attribute 的结构体定义为：

```
typedef int (*att_readwrite_callback_t)(void* p);
typedef struct attribute
{
  u8  attNum;
  u8  uuidLen;
  u8  attrLen;
  u8  attrMaxLen;
  u8* uuid;
  u8* pAttrValue;
  att_readwrite_callback_t w;
  att_readwrite_callback_t r;
} attribute_t;
```

结合目前 8267 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，如下截图所示：

```
316 const attribute_t my_Attributes[] = {
317     {50,0,0,0,0,0}, //
318
319     // gatt
320     {7,2,2,2,(u8*)(&my_primaryServiceUUID),     (u8*)(&my_gapServiceUUID), 0},
321     {0,2,1,1,(u8*)(&my_characterUUID),      (u8*)(&my_devNameCharacter), 0},
322     {0,2,sizeof (my_devName), sizeof (my_devName),(u8*)(&my_devNameUUID),           (u8*)(my_devName),
323     {0,2,1,1,(u8*)(&my_characterUUID),      (u8*)(&my_appearanceCharacter), 0},
324     {0,2,sizeof (my_appearance), sizeof (my_appearance),(u8*)(&my_appearanceUIID),  (u8*)(&my_appearan
325     {0,2,1,1,(u8*)(&my_characterUUID),      (u8*)(&my_periConnParamChar), 0},
326     {0,2,sizeof (my_periConnParameters), sizeof (my_periConnParameters),(u8*)(&my_periConnParamUUID),
327
328     ////////////////// all handles below need +7 ////////////////////////////////////////////////////
329     // 1. Device Information Service
330     {3,2,2,2,(u8*)(&my_primaryServiceUUID),     (u8*)(&my_devServiceUUID), 0},
331     {0,2,1,1,(u8*)(&my_characterUUID),      (u8*)(&my_PnPCharacter), 0},
332     {0,2,sizeof (my_PnPtrs), sizeof (my_PnPtrs),(u8*)(&my_PnPUUID),             (u8*)(my_PnPtrs), 0},
333
334     ////////////////////////////////// 4. HID Service ////////////////////////////////////////////////////
335     #if HID_MOUSE_ATT_ENABLE
336     {34,2,2,2,(u8*)(&my_primaryServiceUUID),    (u8*)(&my_hidServiceUUID), 0},
337     #else
338     {27,2,2,2,(u8*)(&my_primaryServiceUUID),    (u8*)(&my_hidServiceUUID), 0},
339     #endif
340
341     //include battery service
342     {0,2,sizeof(include),sizeof(include),(u8*)(&hidIncludeUUID),    (u8*)(include), 0},
343
344     //protocol mode
345     {0,2,1,1,(u8*)(&my_characterUUID),      (u8*)(&protocolModeProp), 0},               //prop
```

图 15　8267 BLE SDK Attribute Table 截图

首先请注意，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，这个 table 里所有内容是只读的，不能改写。

3.6.2.1　attNum

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：

{50,0,0,0,0,0},

attNum = 50 表示当前 Attribute Table 中共有 50 个 Attribute。除了第 0 项这个无效的 Attribute，后面其他所有的 Attribute，都不用再关注 attNum 的值，定义的时候填 0 即可。

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，在 Attribute Table 中数 Attribute 在当前数组中的下表号，就能知道该 Attribute 当前的 Attribute Handle 值。如当前 SDK 中一下几个 Attribute Handle 的值的宏定义的值就是数

出来的。

```
#define          HID_HANDLE_CONSUME_REPORT          21
#define          HID_HANDLE_KEYBOARD_REPORT         25
#define          AUDIO_HANDLE_MIC                   43
```

将 Attribute Table 中所有的 Attribute 数完，输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 50，user 如果添加或删除了 Attribute，需要对此 attNum 进行修改。

3.6.2.2 uuid、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。

1) 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;
{0,2,1,1,(u8*)(&my_characterUUID),      (u8*)(&my_devNameCharacter), 0},
```

UUID=0x2803 在 BLE 中表示 character，uuid 指向 my_devNameCharacter 在 flash 中的地址，uuidLen 为 2，master 来读这个 Attribute 时，UUID 会是 0x2803

2) 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,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00
}
const u8 my_MicUUID[16]         = TELINK_MIC_DATA;
{0,16,1,1,(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

3.6.2.3 pAttrValue、attrLen、attrMaxLen

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 的地址，attrLen 和 attrMaxLen 都是用来反应该数据在 RAM 上的长度，一般代码中将 attrlen 和 attrMaxLen 设置为一样，都等于当前 Attribute Value 在 RAM 上的长度。当 master 读取 slave 某个 Attribute 的 Attribute Value 时， 8267 BLE SDK 从 Attribute 的 pAttrValue 指针指向的区域（flash 或 RAM）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 pAttrValue 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
};

{0,2, sizeof(hidInformation), 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，不会被涉及到写操作，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 和 attrMaxLen 都以实际的 hidInformation 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。

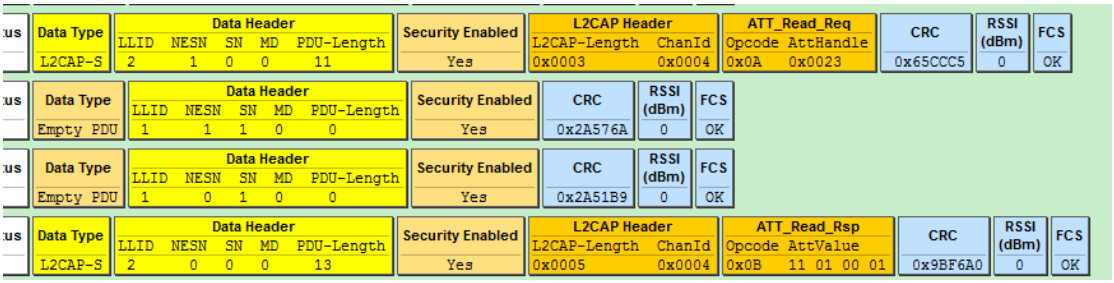

图 16 master 读 hidInformationBLE 抓包

Attribute Handle=40 battery value 的 Attribute，相关代码：

u8 my_batVal[1] = {99};

{0,2,1,1,(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，attrlen 和 attrMaxLen 都是 1。

3.6.2.4 回调函数 w

回调函数 w 是写函数。函数原型：

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

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

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

1) opcode = 0x12, Write Request，见<<Core_v4.1_BLE_spec.pdf>> P2146。

2) opcode = 0x52, Write Command ， 见<<Core_v4.1_BLE_spec.pdf>> P2148。

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

3.4.5.1 Write Request

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

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

Table 3.26: Format of Write Request

图 17 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

Table 3.28: Format of Write Command

图 18 BLE 协议栈中 Write Command

函调函数 w 的 void 型 p 指针指向 master 写命令的具体数值。根据上面所示的 BLE 协议栈中的 master 写命令的格式可知，p 都是指向数据包中 Attribute Value 的开始的地方。可以参考本文档 OTA 数据交互的介绍，来了解回调函数 w 的具体用法。

3.6.2.5 回调函数 r

回调函数 r 是写函数。函数原型：

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

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

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

1) opcode = 0x0a，Read Request，见<<Core_v4.1_BLE_spec.pdf>> P2139。

2) opcode = 0x0C，Read Blob Request， 见<<Core_v4.1_BLE_spec.pdf>> P2140。

slave 收到以上写命令后，如果 user 设置了回调函数 r， slave 收到以上读命令后执行 user 的回调函数 r。 跟回调函数 w 不一样的地方是，只要收到以上两读命令，不管 user 是

否设置了回调读函数，slave 都会自动从 pAttrValue 指针所指向的区域读 attrLen 个值的 value 回给 master。

3.6.3 Attribute PDU

8267 BLE SDK 目前支持的 Attribute PDU 有以下几类：

1) Requests：client 发送给 server 的数据请求。

2) Responses：server 收到 client 的 request 后发送的数据回应。

3) Commands：client 发送给 server 的命令。

4) Notifications：server 发送给 client 的数据。

其他的两类，Indications 和 Confirmations 暂时还没有加进去，后续再加进来。

8267 BLE SDK 支持的以上几种类型 PDU 详细说明如下。

3.6.3.1 Read by Group Type Request、Read by Group Type Response

master 发送 Read by Group 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 Response 回复 Attribute Group 信息。

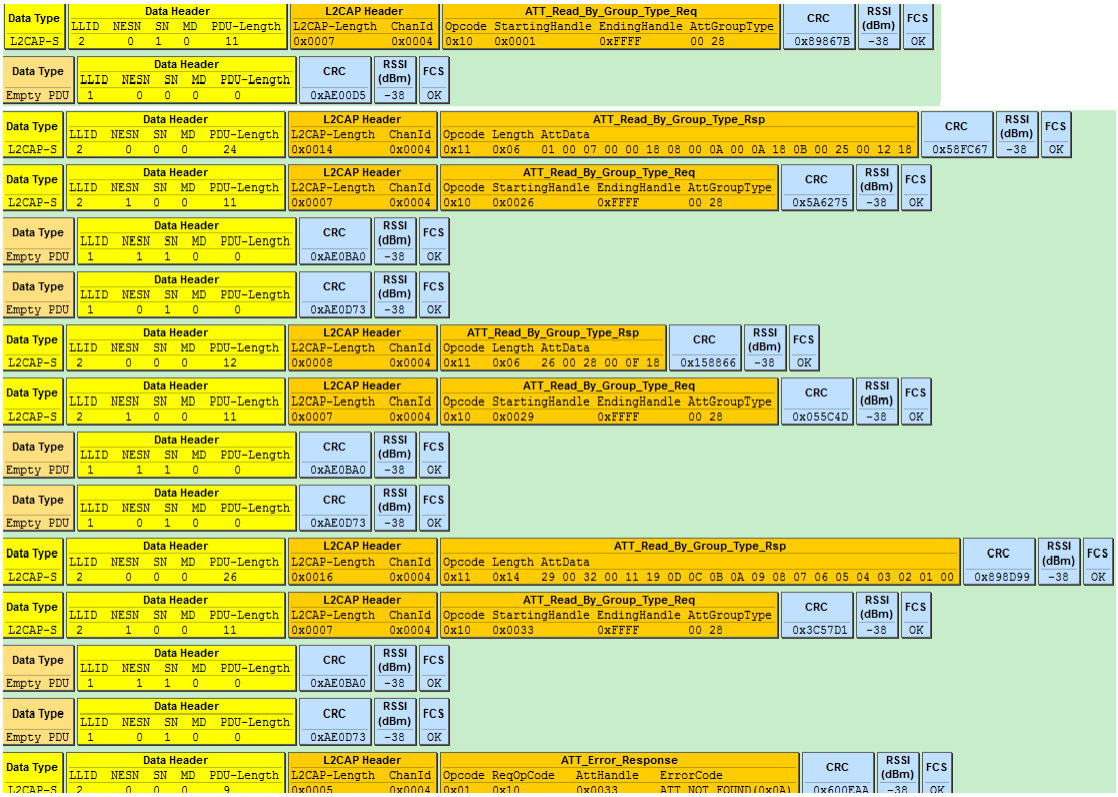

图 19　Read by Group Type Request/Read by Group Type Response

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;
```

参考当前参考 8267 code，slave 的 Attribute table 中有以下几组符合该要求：

1) attHandle 从 0x0001-0x0007 的 Attribute Group，Attribute Value 为 SERVICE_UUID_GENERIC_ACCESS（0x1800）
2) attHandle 从 0x0008-0x000a 的 Attribute Group，Attribute Value 为 SERVICE_UUID_DEVICE_INFORMATION（ 0x180A）
3) attHandle 从 0x000B-0x0025 的 Attribute Group，Attribute Value 为 SERVICE_UUID_HUMAN_INTERFACE_DEVICE（0x1812）
4) attHandle 从 0x0026-0x0028 的 Attribute Group，Attribute Value 为 SERVICE_UUID_BATTERY（0x180F）
5) 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 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 Request。Read by Group Request 和 Read by Group Respnse 的命令详细说明见《Core_v4.1_BLE_spec》P2143。

3.6.3.2　Find by Type Value Request、Find by Type Value Response

master 发送 Read 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 Read by Type Value Response 回复 Attribute。

Read by Type Value Request 和 Read by Type Value Respnse 的命令详细说明见

《Core_v4.1_BLE_spec》P2134。

3.6.3.3 Read by Type Request、Read by Type Response

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 attHanlde 中找到符合 AttributeType 的 Attribute，通过 Read by Type Response 回复 Attribute。

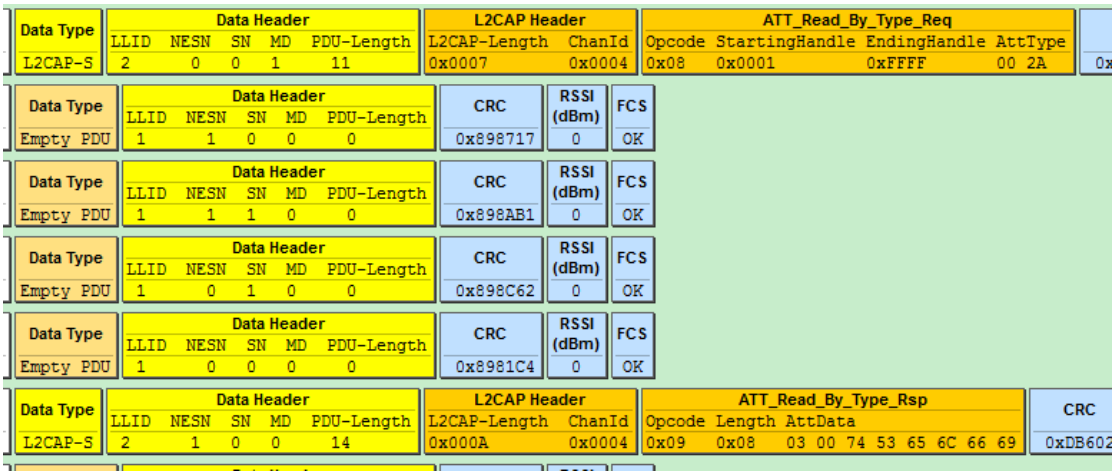

图 20 Read by Type Request/Read by Type Response

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;
{0,2,sizeof (my_devName), 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。

Read by Type Request 和 Read by Type Respnse 的命令详细说明见《Core_v4.1_BLE_spec》P2136。

3.6.3.4 Find information Request、Find information Response

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。

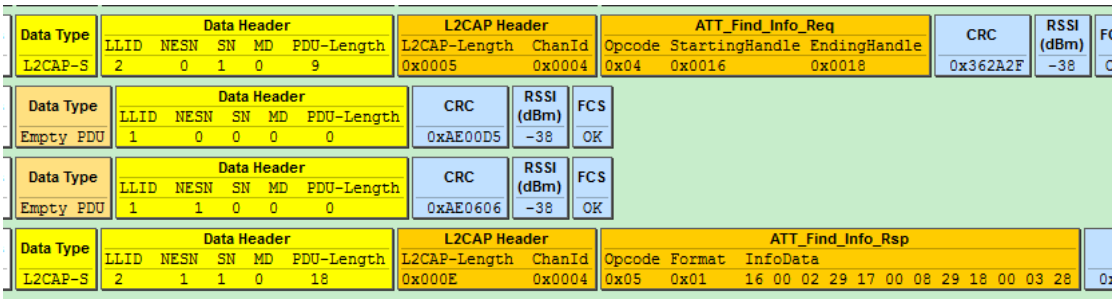

图 21　Find information request/Find information response

Find information request 和 Find information response 的命令详细说明见《Core_v4.1_BLE_spec》P2132。

3.6.3.5　Read Request、Read Response

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。

Read Request 和 Read Response 的命令详细说明见《Core_v4.1_BLE_spec》P2139。

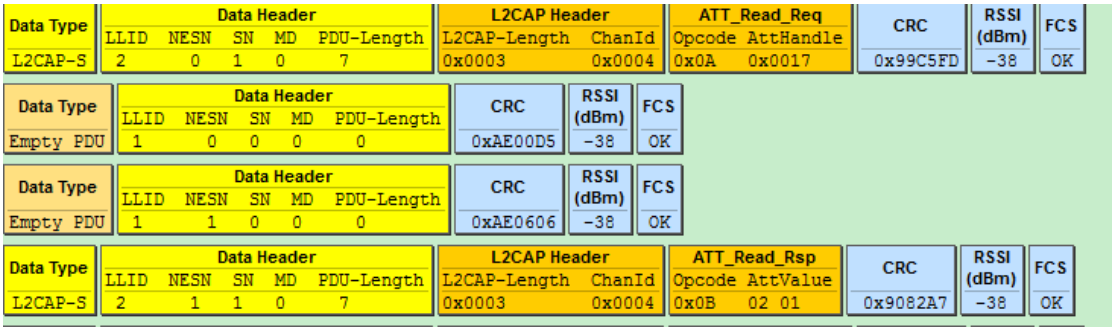

图 22　Read Request/Read Response

3.6.3.6　Read Blob Request、Read Blob Response

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。

Read Blob Request 和 Read Blob Response 的命令详细说明见《Core_v4.1_BLE_spec》P2140。

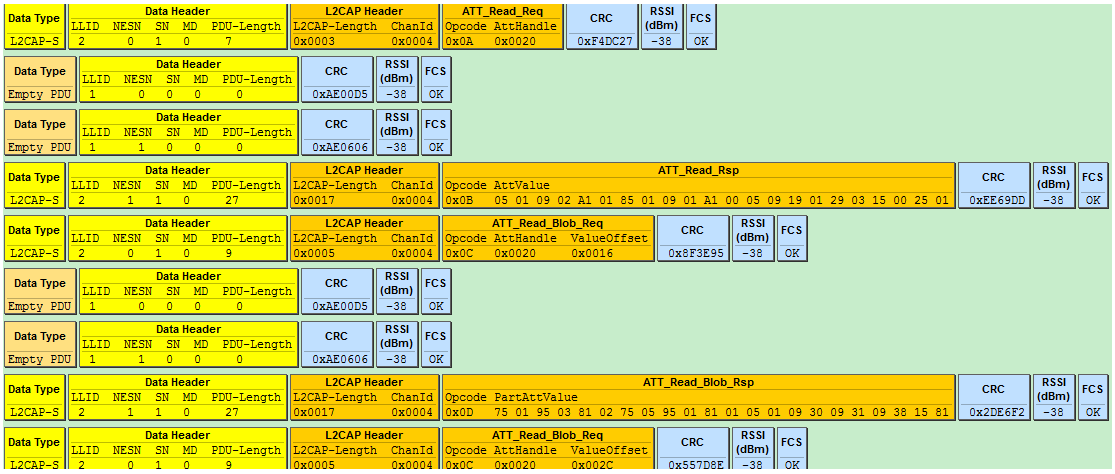

图 23 Read Blob Request/Read Blob Response

3.6.3.7 Exchange MTU Request

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

Exchange MTU Request 和 Exchange MTU Response 的命令详细说明见《Core_v4.1_BLE_spec》P2130。

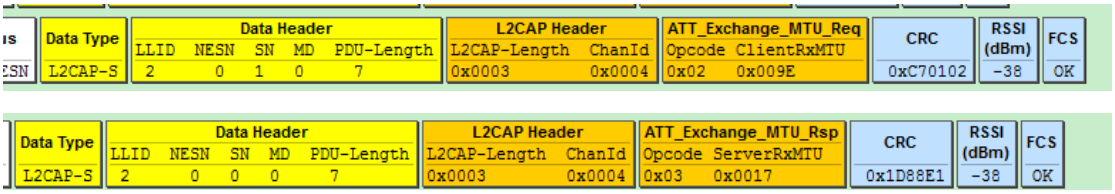

图 24 Exchange MTU Request/Exchange MTU Response

3.6.3.8 Write Request、Write Response

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 attHanlde 为 0x0016 的 Attribute 写入 Attribute Value 为 0x0001，slave 收到后执行该写操作，并回 Write Response。

Write Request 和 Write Response 的命令详细说明见《Core_v4.1_BLE_spec》P2146。

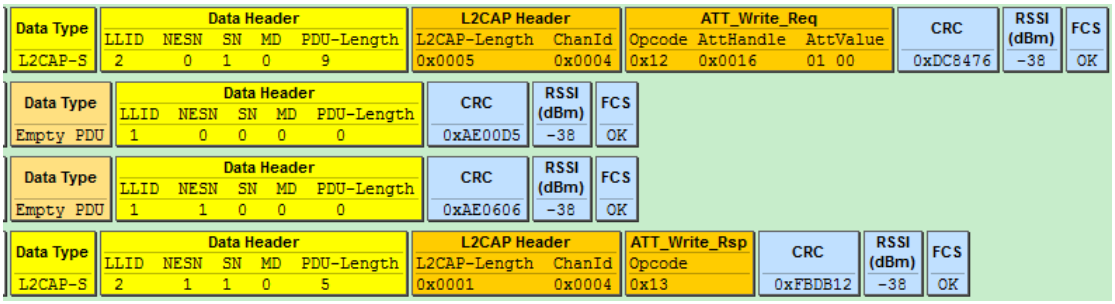

图 25 Write Request/Write Response

3.6.3.9 Write Command

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。Write Command 的命令详细说明见《Core_v4.1_BLE_spec》P2148。

3.6.3.10 Handle Value Notification

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

图 26 BLE 协议栈 handle value notification

上图所示为 ble 协议栈中 handle value notification 的格式，见《Core_v4.1_BLE_spec》P2154。

8267 BLE SDK 提供以下两个接口函数，用于某个 Attribute 的 Handle Value notification：

u8 blt_push_notify (u16 handle, u32 val, int len)

此函数只适用于 notify 的 data 长度不超过 4 个 byte 的情况。handle 为对应 Attribute 的 attHandle，val 为要发送的数据，len 指定发送的数据的字节数。返回值为 len。

假设 handle 为 20，

u8 data1;

blt_push_notify (20, data1, 1); //发送 1 个 byte 的 data1

u16 data2;

blt_push_notify (20, data2, 2); //发送 2 个 byte 的 data2

int data3;

blt_push_notify (20, data3, 4); //发送 4 个 byte 的 data3

blt_push_notify (20, data3, 3); //发送 data3 的前 3 个 byte

u8 blt_push_notify_data (u16 handle, u8 *p, int len)

此接口适用于 notify 的 data 长度理论上无限制，长度小于等于 MTX_SIZE 时，一个包发送，长度大于 MTU_SIZE 时，自动分包发送。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。返回值为 len。

假设 handle 为 20，

int data4;

blt_push_notify_data (20, &data4, 4); //发送 data4 的 4 个 byte

u8 data5[12];

blt_push_notify_data (20, data5, 12); //发送 data5 数组的 12 个值

u8 data6[100];

blt_push_notify_data (20, data6, 100); //发送 data6 数组的 100 个值

4 低功耗管理（PM）

4.1 低功耗模式

8267 MCU 具有三种基本的运行模式：

1) working mode：此时 MCU 正常执行程序，硬件数字模块正常工作，根据程序需要打开相应的模拟模块和 BLE RF 收发器模块。工作电流在 10-30mA 之间。
2) suspend mode：低功耗模式 1，此时程序停止运行，类似一个暂停功能。IC 硬件上大部分的硬件模块都断电，PM 模块维持正常工作。数字和模拟寄存器、内存上所有的数据和状态都 hold 住，不会丢任何数据，此时纯粹的 IC 电流在 7-8 uA 之间。当 suspend 被唤醒后，程序继续执行。
3) deepsleep mode： 低功耗模式 2，此时程序停止运行，IC 上绝大部分的硬件都断电，

PM 硬件模块维持工作。数字寄存器和内存掉电，所有数据不再保存，只有模拟寄存器少数几个寄存器不掉电，可以保存一些必要的信息（pm_8267.h 中的 DEEP_ANA_REG0-DEEP_ANA_REG7）。当 deepsleep 被唤醒时，MCU 将重新开始工作，跟重现上电是一样的效果，程序重新开始进行初始化和执行，可以在 deepsleep 前在 DEEP_ANA_REG0-DEEP_ANA_REG7 寄存器里存入一些信息，那么在初始化的时候读这些寄存器是否存入了预先设置的信息用来判断程序是一个而纯粹的重新上电还是 deepsleep 醒来。在 deepsleep 下纯粹的 IC 电流为 0.7 uA 左右，如果加上内置 flash 的 1 uA 左右，总体电流在 1.7 uA 左右。

8267 BLE SDK 实现低功耗的方法为：在图 4 所示的 BLE 工作时序中，一个 adv_interval/conn_interval 对应一个 main_loop，MCU 处于 working mode 的时间占很小的比例，处理完相应的任务后，即可进入 suspend 状态，由于 suspend 的电流非常低且 suspend 的时间比例占大，整个 mianloop 的平均电流就很低。

当我们不需要 8267 MCU 工作时，可以设置 MCU 进入 deepsleep 模式，使得功耗极低，通过一些特殊的操作将其唤醒并重新开始跑程序。

4.2 低功耗唤醒源

8267 MCU 的低功耗唤醒源源示意图如下，suspend 可被 CORE 和 timer 唤醒，deepsleep 可被 PAD 和 timer 唤醒。8267 SDK 中，我们只关注三种唤醒源：

```
enum {
    PM_WAKEUP_PAD   = BIT(4),
     PM_WAKEUP_CORE  = BIT(5),
    PM_WAKEUP_TIMER = BIT(6),
 };
```

CORE 的唤醒源有四种，但在 8267 SDK 中我们只关注 GPIO 模块触发的 CORE 唤醒，所以我们约定本文档其他地方所述的 CORE 唤醒指的是 GPIO CORE 的唤醒。PAD 模块只能由 GPIO 模块触发，所以我们约定本文档其他地方所述的 PAD 唤醒指的是 GPIO PAD 的唤醒。8267 所有 GPIO 的高低电平都可以通过 CORE 模块唤醒 suspend，8267 所有 GPIO 的高低电平都可以通过 PAD 模块唤醒 deepsleep。

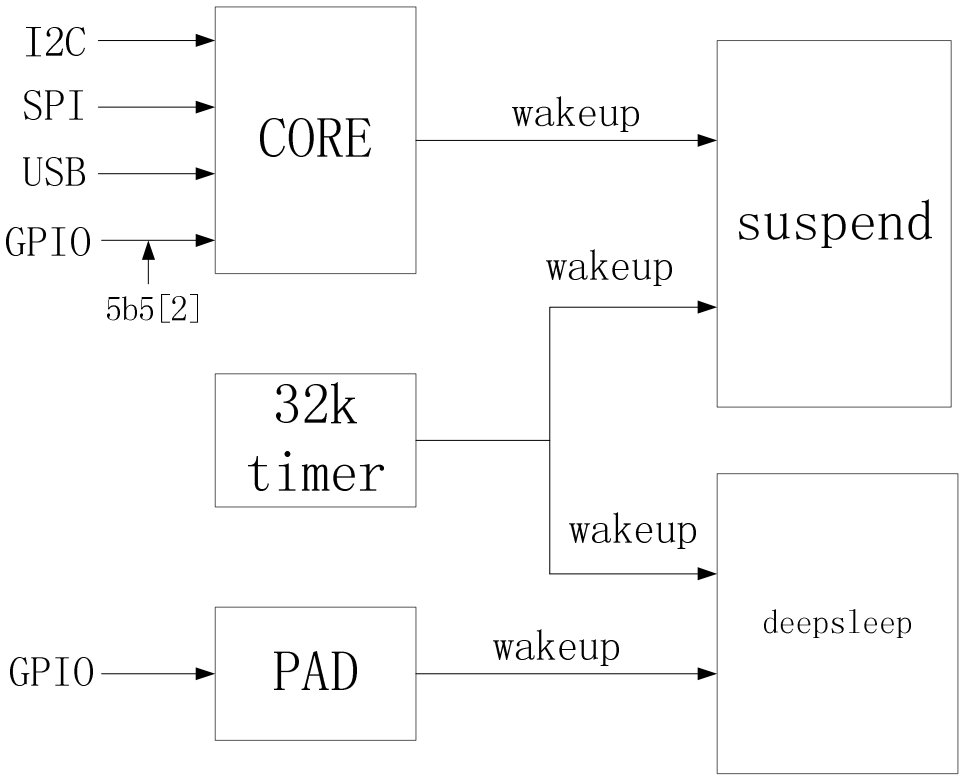

图 27　8267 硬件唤醒源

对于唤醒源 PM_WAKEUP_TIMER，不需要提前设定任何东西；对于唤醒源 PM_WAKEUP_CORE，只关注 GPIO 唤醒，需要提前设定 GPIO 的唤醒使能和唤醒电平极性；对于唤醒源 PM_WAKEUP_PAD，只能由 GPIO 唤醒，需要提前设定 GPIO 的唤醒使能和唤醒电平极性。

GPIO　CORE 的唤醒配置函数为：

```
void gpio_set_wakeup(u32 pin, u32 level, int en);
```

pin 为 GPIO 定义，level：　1 表示高电平唤醒，0 表示低电平唤醒；en: 1 表示 enable，0 表示 disable。举例说明：

```
gpio_set_wakeup(GPIO_PC2,   1,   1);   //GPIO_PC2 CORE 唤醒打开，高电平唤醒
gpio_set_wakeup(GPIO_PC2,   1,   0);   //GPIO_PC2 CORE 唤醒关闭
gpio_set_wakeup(GPIO_PB5,   0,   1);   //GPIO_PB5 CORE 唤醒打开，低电平唤醒
gpio_set_wakeup(GPIO_PB5,   0,   0);   //GPIO_PB5 CORE 唤醒关闭
```

使用 GPIO_CORE 唤醒时，需要将图中所示的数字寄存器 5b5[2]打开，这是 GPIO CORE 硬件模块的总开关，打开该开关的函数为：

```
gpio_core_wakeup_enable_all (int en)   //en:   1 打开总开关，0：关闭总开关
```

GPIO　PAD 的唤醒配置函数为：

void cpu_set_gpio_wakeup (int pin, int pol, int en);

pin 为 GPIO 定义，pol：　1 表示上升沿唤醒，0 表示下降沿唤醒；en: 1 表示 enable，0 表示 disable。举例说明：

cpu_set_gpio_wakeup (GPIO_PC2,　1,　1);　//GPIO_PC2 PAD 唤醒打开，上升沿唤醒

cpu_set_gpio_wakeup (GPIO_PC2,　1,　0);　//GPIO_PC2 PAD 唤醒关闭

cpu_set_gpio_wakeup (GPIO_PB5,　0,　1);　//GPIO_PB5 PAD 唤醒打开，下降沿唤醒

cpu_set_gpio_wakeup (GPIO_PB5,　0,　0);　//GPIO_PB5 PAD 唤醒关闭

4.3　低功耗模式的进入和唤醒

低功耗进入和唤醒的入口函数(pm_8267.h 中)为 cpu_sleep_wakeup，user 可以调用这个函数，让 MCU 进入低功耗模式，并设置相应的唤醒源。在 8267 BLE SDK user 调用 blt_brx_sleep 函数时对 cpu_sleep_wakeup 函数进行了封装，提供了更上层的接口函数 blt_enable_suspend，为了更好的理解，先介绍 cpu_sleep_wakeup 函数。

int　cpu_sleep_wakeup (int deepsleep, int wakeup_src, u32 wakeup_tick);

第一个参数 deepsleep：　0 表示进入 suspend，　1 表示 deepsleep

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_CORE、PM_WAKEUP_TIMER 中的一个或者多个，注意之前所说的 suspend 的唤醒源为 PM_WAKEUP_TIMER 和 PM_WAKEUP_CORE，deepsleep 的唤醒源为 PM_WAKEUP_TIMER 和 PM_WAKEUP_PAD。如果 wakeup_src 为 0，那么进入低功耗后，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，按照前面所说的 system tick 来设置，当 system tick 的值达到这个这个设定的 wakeup_tick 后，低功耗模式被唤醒。wakeup_tick 的值需要根据当前的 system tick 的值，加上需要睡眠的时间换算成的 tick 值，才可以有效的控制睡眠时间。如果没有考虑 system tick，直接对 wakeup_tick 进行设置，唤醒的时间就无法控制，对于 16M 的 system clock，system tick 从 0x00000000 - 0xffffffff 跑一轮需要的时间为 256 S，如

果 wakeup_tick 的值没有设好，最坏的情况需要等到 256 S 才会被唤醒。

举例说明 cpu_sleep_wakeup 的用法：

1) cpu_sleep_wakeup (0, PM_WAKEUP_CORE, 0);

   程序执行该函数时进入 suspend 模式，只能被 GPIO CORE 唤醒

2) cpu_sleep_wakeup (0, PM_WAKEUP_TIMER, clock_time() + 10*CLOCK_SYS_CLOCK_1MS);

   程序执行该函数时进入 suspend 模式，只能被 TIMER 唤醒，唤醒时间为执行该函数时的时间加上 10 ms，所以 suspend 时间为 10 ms。

3) cpu_sleep_wakeup (0, PM_WAKEUP_CORE | PM_WAKEUP_TIMER, clock_time() + 50*CLOCK_SYS_CLOCK_1MS);

   程序执行该函数时进入 suspend 模式，可被 GPIO CORE 和 TIMER 唤醒，Timer 唤醒的时间设置为 50ms，那么如果在 50ms 之前出发了 GPIO 的唤醒动作，MCU 会被 GPIO 唤醒，如果 50ms 之前无 GPIO 动作，MCU 会被 timer 唤醒。

4) cpu_sleep_wakeup (1, PM_WAKEUP_PAD, 0);

   程序执行该函数时进入 deepsleep 模式，可被 GPIO PAD 唤醒。

5) cpu_sleep_wakeup (1, PM_WAKEUP_TIMER, clock_time() + 8* CLOCK_SYS_CLOCK_1S);

   程序执行该函数时进入 deepsleep 模式，可被 Timer 唤醒，唤醒时间执行该函数的 8 s 后。

6) cpu_sleep_wakeup (1, PM_WAKEUP_PAD | PM_WAKEUP_TIMER, clock_time() + 10*CLOCK_SYS_CLOCK_1S);

   程序执行该函数时进入 deepsleep 模式，可被 GPIO PAD 和 Timer 唤醒，Timer 唤醒时间执行该函数的 10 s 后。如果在 10 S 之前出发了 GPIO 动作，被 GPIO 唤醒，否则被 Timer 唤醒。

4.4 低功耗的配置

4.4.1 blt_enable_suspend

由于 8267 SDK 中封装了更上层的低功耗配置接口 blt_enable_suspend，user 实际基本不需要用到 cpu_sleep_wakeup 函数。使用下面接口对 8267 BLE SDK 进行低功耗的配置。

```
u8          blt_enable_suspend (u8 en);
```

该函数的实质是设定一个 mask 变量，在 blt_brx_sleep 函数里面会对这个 mask 进行检查以决定如何调用 cpu_sleep_wakeup 函数。

形参 u8 型 en 是一个 mask，可取以下宏定义的几个值，或者将它们进行“或”操作，返回值是执行该函数设定最新的 mask 时，返回之前老的 mask 值，这样可以保存之前的 mask 值，以便后面进行恢复。

```
#define          SUSPEND_ADV              BIT(0)
#define          SUSPEND_CONN             BIT(1)
#define          DEEPSLEEP_ADV            BIT(2)
#define          DEEPSLEEP_CONN           BIT(3)
#define          SUSPEND_DISABLE          BIT(7)
```

可以对照图 4 所示的 BLE 工作时序来理解。

1) SUSPEND_ADV：当 slave 处于广播状态，在前面 working 时间内处理好各种任务后，执行 blt_brx_sleep 函数时进入 suspend 模式；
2) SUSPEND_CONN：当 slave 处于连接状态，执行 blt_brx_sleep 函数时进入 suspend 模式；
3) SUSPEND_DISABLE：不管在广播状态还是连接状态，执行 blt_brx_sleep 函数时都不进 suspend，程序通过空等（不做任何操作）来耗掉 idle 的时间；
4) DEEPSLEEP_ADV：slave 处于广播状态时，如果在某一个 main_loop 的 working time 内设置了该 mask，执行 blt_brx_sleep 函数时进入 deepsleep 模式。DEEPSLEEP_ADV 只能在 main_loop 里设定，不能提前在 user_init 初始化的设定，一旦设定，程序运行第一个 main_loop 就会进入 deepsleep。
5) DEEPSLEEP_CONN：slave 处于连接状态时，如果在某一个 main_loop 的 working time 内设置了该 mask，执行 blt_brx_sleep 函数时进入 deepsleep 模式。DEEPSLEEP_CONN 只能在 main_loop 里设定，不能提前在 user_init 初始化的设定，一旦设定，slave 响应 master 的 conn_req 后的第一个 mainloop 就会进入 deepsleep。

在 8267 BLE SDK 使用 blt_enable_suspend 配置的 suspend，不管是广播状态还是连接状态，blt_brx_slee 函数里面都自动配置器唤醒源为 PM_WAKEUP_TIMER，且只有这一个唤醒源，唤醒的时间点 wakeup_tick 的值在广播状态有 T_advEvent 计算得到（user 通过 blt_set_adv_interval 函数设定），在连接状态由 connection interval 计算得到。使用

blt_enable_suspend 配置的 deepsleep，SDK 默认没有任何唤醒源，须 user 自己设定 deepsleep 的唤醒源。

基于 blt_enable_suspend 接口，user 可以配置自己的 slave 设备在广播态和连接态的低功耗模式。user 可以在初始化的时候调用这个接口，以确定 slave 设备是否进入 suspend，用以配置整个系统整体的低功耗模式。初始化的时候只能配置 suspend，不能配置 deepsleep，这个原因上面已经解释。如目前 8267 BLE SDK 中参考的配置代码为：

```
blt_enable_suspend (SUSPEND_ADV | SUSPEND_CONN); //广播态和连接态都进 suspend
```

初始化时设定低功耗模式以后，在 main_loop 执行的过程中，可能需要对低功耗模式进行修改，比如硬件上一些任务的时间过长，会超过 main_loop 的时间，此时如果 MCU 进入 suspend 会导致该任务失败（如 I2C 的数据收发、UART 数据收发），user 可以在 UI entry 的地方修改低功耗的配置，让 MCU 在处理这些任务的时候不进 suspend，等到处理好了再重新回到正常的低功耗模式。

```
if(ui_task_done)
{
    blt_enable_suspend (SUSPEND_ADV | SUSPEND_CONN);
}
else
{
    blt_enable_suspend (SUSPEND_DISABLE);
}
```

4.4.2 blt_get_suspend_mask

通过接口获得控制 8267 SDK 低功耗控制的 mask 值：

```
u8  blt_get_suspend_mask (void)  //返回低功耗控制的 mask 值
```

返回的值，也就是用户最近一次设定的值，如：

```
blt_enable_suspend (SUSPEND_ADV);
u8  mask = blt_get_suspend_mask();  //此时 mask 的值是 SUSPEND_ADV
```

4.4.3 blt_set_wakeup_source

user 可以通过 blt_set_wakeup_source 接口设置当前 main_loop 的 suspend/deepsleep 的唤醒源，注意此函数只能在下面的 BLT_EV_FLAG_SET_WAKEUP_SOURCE 的回调函数中使用，在其他地方使用都将无效。

对于三个唤醒源 PM_WAKEUP_PAD、PM_WAKEUP_CORE、PM_WAKEUP_TIMER 的设置，

8267 BLE SDK 的处理方法为：若当前 main_loop 进 suspend，系统默认只设置一个唤醒源 PM_WAKEUP_TIMER，若当前 mian_loop 进 deepsleep，系统默认不设置任何唤醒源。在这两种情况下，如果用户需要添加其他的唤醒源，调用 blt_set_wakeup_source 解决。

```
void blt_set_wakeup_source(int src);
```

调用时，src 设置为 PM_WAKEUP_PAD、PM_WAKEUP_CORE、PM_WAKEUP_TIMER 中的一个或者多个进行“或”操作。假设 user 在 BLT_EV_FLAG_SET_WAKEUP_SOURCE 的回调函数中写法为：

```
blt_set_wakeup_source(x);  //x 为三种唤醒源的一个或者多个的“或”操作
```

那么如果 MCU 进 suspend，实际唤醒源将为：

```
x |  PM_WAKEUP_TIMER
```

如果 MCU 进 deepsleep，实际唤醒源将为：

```
x
```

4.4.4 BLT_EV_FLAG_SET_WAKEUP_SOURCE 事件回调函数

前面介绍基于时间触发的回调中 BLT_EV_FLAG_SET_WAKEUP_SOURCE，该事件是在执行 blt_brx_sleep 函数时，已经按照 user 的设定决定了当前 mian_loop 是否要进 suspend 以及如何唤醒的时候触发，user 可以注册这个回调函数，在此时对低功耗模式进行调整并设置相关的唤醒源。8267 BLE SDK 推荐使用这种方法对运行中的 slave 设备进行低功耗的调整，原因是在此时 SDK 已经计算好了 suspend 醒来的时间点，user 可以根据这个时间点来做一些操作，如果是在 UI entry 的地方，无法获知 suspend 醒来的时间点。

获取当前即将进入的 suspend 醒来的时间点的函数为：

```
u32 blt_get_wakeup_time(void);
```

返回值为醒来时间点的 sysytem tick，即下面函数中设置的 wakeup_tick。

```
int    cpu_sleep_wakeup (int deepsleep, int wakeup_src, u32 wakeup_tick);
```

举例说明 BLT_EV_FLAG_SET_WAKEUP_SOURCE 事件回调函数的用法。下面这个例子是带按键扫描的应用。

```
blt_register_event_callback (BLT_EV_FLAG_SET_WAKEUP_SOURCE,
                             &ble_remote_set_sleep_wakeup);

void    ble_remote_set_sleep_wakeup (u8 e, u8 *p)
```

```
{
    if( blt_state == BLT_LINK_STATE_CONN && ((u32)(blt_get_wakeup_time() - clock_time())) >
                    60 * CLOCK_SYS_CLOCK_1MS){    //suspend time > 30ms.add gpio wakeup
        blt_set_wakeup_source(PM_WAKEUP_CORE);    //gpio CORE wakeup suspend
    }
```

如果当前状态为连接状态，并且当前要进入的 suspend 的唤醒时间点距离当前时间超过 60 ms，那么将 GPIO CORE 的唤醒添加进入，其目的是为了防止 suspend 时间过长，导致使用者的按键动作没有被计时响应或者丢掉这个键。这样设置以后，如果 timer 的唤醒时间点还没到，GPIO 上的按键有动作，触发 MCU 提前唤醒，去处理按键的扫描任务。

```
    //adv 60s, deepsleep
    if( blt_state == BLT_LINK_STATE_ADV && clock_time_exceed(adv_begin_tick,
                                        60 * 1000 * 1000)){
        blt_enable_suspend (blt_get_suspend_mask() | DEEPSLEEP_ADV); //set deepsleep
        blt_set_wakeup_source(PM_WAKEUP_PAD);    //gpio PAD wakeup deesleep
    }
```

如果 slave 在广播状态，且连续广播时间达到 60 S，设置当前 main_loop 进入 deepsleep，设置 depsleep 的唤醒源为 GPIO PAD。判断是否广播超过 60 S 的方法是用软件定时器的方法，对 adv_begin_tick 进行比较，adv_begin_tick 是进入广播时的 system tick，user 需要注意每次进入广播的时候都需要对这个变量取系统时钟的值，包括上电初始化后自动进广播状态和 terminate 的时候回到广播状态（对照图 3　BLE 状态机）。之前的软件定时器有说过 16M 系统时钟时最大的定时时间为 256 S，这里需要注意。

```
    //60s no key, deepsleep
    if( blt_state == BLT_LINK_STATE_CONN && clock_time_exceed(key_active_tick,
        60 * 1000 * 1000) && !key_not_released && !get_mic_enable()){
        blt_enable_suspend (blt_get_suspend_mask() | DEEPSLEEP_CONN); //set deepsleep
        blt_set_wakeup_source(PM_WAKEUP_PAD);    //gpio PAD wakeup deesleep
        analog_write(DEEP_ANA_REG0,CONN_DEEP_FLG);
    }
```

如果 slave 在连接状态、所有的按键都已经释放、没有音频任务、超过最近一次有效的按键事件 60 S 以上，设置当前 main_loop 进入 deepsleep，设置唤醒源为 GPIO PAD，并且在 deepsleep 记忆寄存器 DEEP_ANA_REG0 记录一些数据（具体记录什么由 user 的实际应用决定）。

```
}
```

4.5 低功耗的调试

待补充

4.6 低功耗电流的优化

待补充

5 语音处理

5.1 音频 MIC 初始化

根据硬件上的音频 MIC 相关的接法，软件初始化：

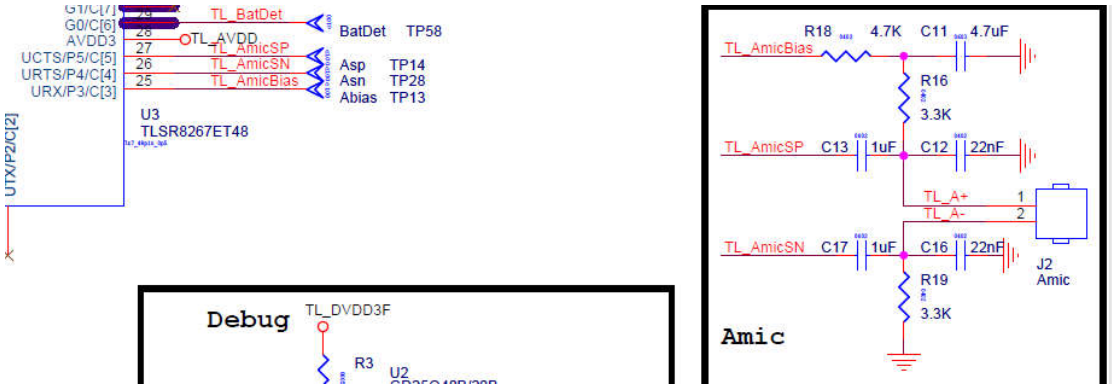

图 28 音频电路

```
config_adc (FLD_ADC_PGA_C45, FLD_ADC_CHN_D0, SYS_16M_AMIC_16K);
//配置硬件MIC模块设置的采样率为16K
gpio_set_output_en (GPIO_PC3, 1);          //AMIC Bias output
gpio_write (GPIO_PC3, 1);

config_mic_buffer ((u32)buffer_mic, TL_MIC_BUFFER_SIZE); //配置音频 buffer
```

前面基于事件触发的回调中详细介绍了 BLT_EV_FLAG_BRX 事件，音频任务由于数据量非常大、时间消耗非常多，放在 BLT_EV_FLAG_BRX 的回调中处理。

```
blt_register_event_callback (BLT_EV_FLAG_BRX, &task_audio);
```

5.2 MIC 采样音频数据处理

5.2.1 音频数据压缩数据量和 RF 传送方法

硬件 MIC 采样出来的原声数据是 pcm 格式的，采用 pcm to adpcm 算法将其压缩为 adpcm 格式，压缩率为 1/4，用以减低 BLE RF 数据量，master 端收到的 adpcm 格式音频数据解压缩还原为 pcm 格式。

8267 硬件 MIC 采样率为 16K，每秒钟 16K 个 sample，每 ms 16 个 sample。8267 采样的时候每个 sample 深度为 16, 16bit = 2 bytes。

每隔 15.5ms，产生 15.5*16=248 个 sample 496 bytes 的原声数据。对这个 496 bytes 进行 pcm 到 adpcm 转换：1/4 压缩为 124 bytes，同时加上 4 个 bytes 的头信息，得到 128 个 bytes 的数据。

128 bytes 的数据，在 L2cap 层上发送给 master，会分成 5 个 packet 上进行，因为每个包最大长度是 27，第一个包必须带 7 个 bytes 的 l2cap 的说明信息：

l2caplen: 2 bytes， chanid： 2 bytes， opcode： 1 byte AttHandle： 2 bytes

下图所示为抓包抓到的音频数据，可以看到第一个包中有 7 个额外的信息，后面紧跟 20 bytes 的音频数据，后面的包 27 bytes 全是音频数据。第一个包只放 20 bytes 的音频数据，后面 4 个包由于是分包，不需要在带 l2cap 说明信息，每个包可以放 27 个 bytes：20 + 27*4 = 128 bytes。

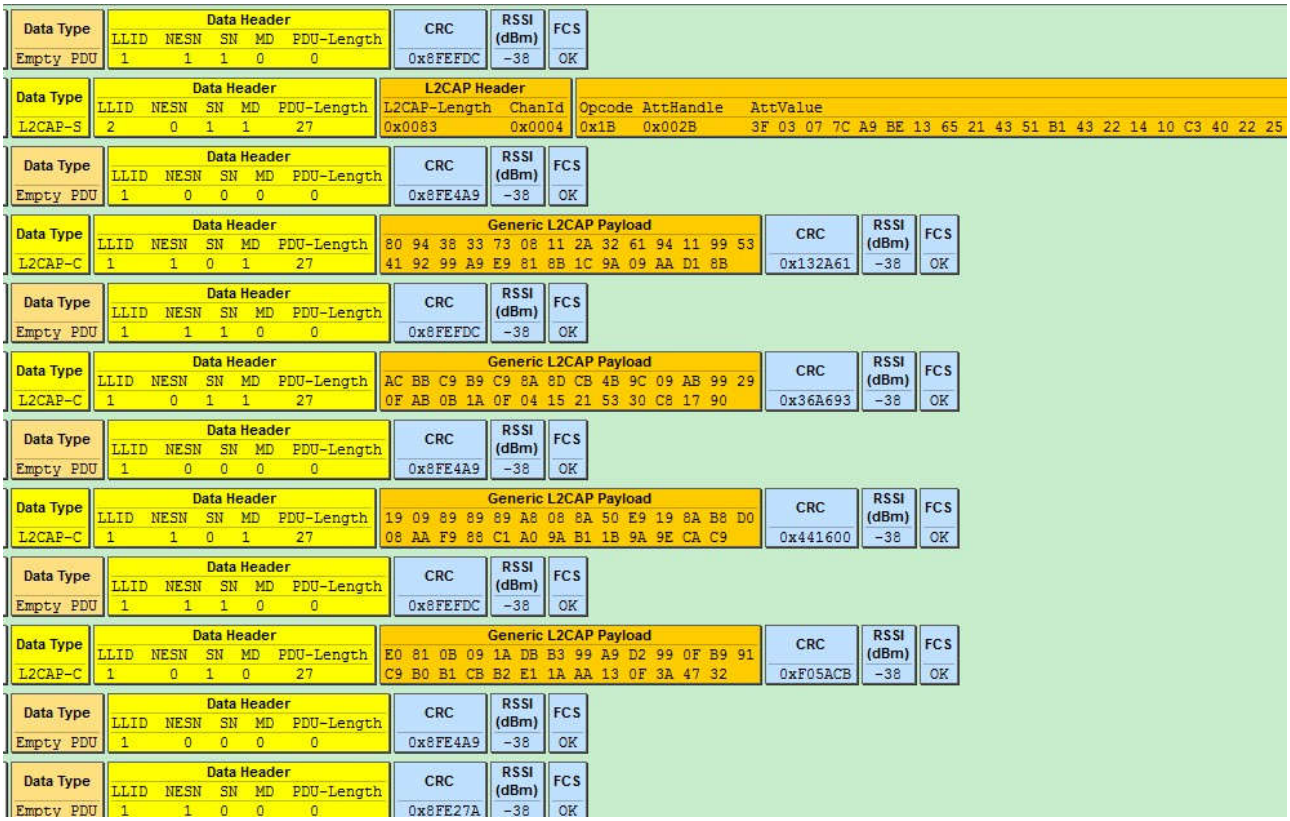

图 29 音频数据抓包

结合前面 ATT 部分的介绍，audio MIC 的 Attribute Table 为

```
{0,2,1,1,(u8*)(&my_characterUUID),     (u8*)(&my_MicProp), 0},
{0,16,1,1,(u8*)(&my_MicUUID),   (u8*)(&my_MicData), 0}, //value
{0,2,sizeof (my_MicName), sizeof (my_MicName),(u8*)(&userdesc_UUID),
```

```
(u8*)(my_MicName), 0},
```

中间那个 Attribute 是负责语音数据传送的，目前它在整个 Attribute Table 中的 AttHandle 的值为 43(0x2B),后面可能会修改。使用 Handle Value Notification 将数据发送给 master。master 只要看到是在 AttHandle 为 0x2B 上的 notification，就可以将 Attribute Value 的值进行拼包成为 128 个 bytes,将拼好的数据放在预先设好的 buffer 里，对其进行解压缩还原为 pcm 格式的 audio 数据。

slave 端的分包和 master 端的拼包都是按照 BLE 协议栈标准的做法,这里不再详细介绍。

5.2.2 音频数据压缩处理

根据以上说明，在 app_config.h 中定义先关的宏：

```
#define    ADPCM_PACKET_LEN                128
#define    TL_MIC_ADPCM_UNIT_SIZE          248
#define    TL_SDM_BUFFER_SIZE              992
#define    TL_MIC_32K_FIR_16K              1

#if TL_MIC_32K_FIR_16K
    #define    TL_MIC_BUFFER_SIZE              1984
#else
    #define    TL_MIC_BUFFER_SIZE              992
#endif
```

config_mic_buffer ((u32)buffer_mic, TL_MIC_BUFFER_SIZE);

每一笔 adpcm 压缩数据量为 248 个 sample，496 个 bytes。由于硬件 MIC 是一直在做采样并把处理过的 pcm 格式数据放到事先设置好的 buffer 上（buffer_mic），将这个 buffer 设置为能够存储 2 笔压缩数据，也就是 496 个 sample，以实现数据的缓冲和保存。直接使用 16K 采样时，496 个 sample 992 个 bytes，TL_MIC_BUFFER_SIZE 为 992；MIC 硬件使用 32K 采样进行 FIR 处理后以 16K 速度将数据放入 buffer 时，可以理解为每个 sample 变成 4 bytes 数据（压缩时只用前两个 bytes 作为一个 16 bit 的原声数据，后两个 bytes 放弃），此时 buffer size 加倍，TL_MIC_BUFFER_SIZE 为 1984。

后面我们对打开 FIR 的情况进行说明，即 TL_MIC_32K_FIR_16K 宏打开，TL_MIC_BUFFER_SIZE 为 1984。

定义 buffer_mic：

```
s16  buffer_mic[TL_MIC_BUFFER_SIZE>>1];   //共 448 个 sample,1984 bytes
```

```
config_mic_buffer ((u32)buffer_mic, TL_MIC_BUFFER_SIZE);
```

上面将配置硬件 MIC 数据输出的 buffer，硬件将采样的数据按照 16K 的速度匀速放入从 buffer_mic 地址开始的内存，向后移动，并且最大长度为 1984，一旦到最大长度，数据回到 buffer_mic 地址开始放数据，且不对内存上的数据进行任何是否已经被读走的判断，直接覆盖老的数据。向 RAM 放数据的过程中，维护一个写指针 reg_audio_wr_ptr（是一个硬件寄存器的值），用于记录当前最新的 audio 数据已经到 RAM 的哪个地址了。

定义一个 buffer_mic_enc，用来存放压缩后的 128 个 bytes 的数据，将 buffer 的 number 设为 4，最多可以缓存 4 笔压缩后的数据。

int buffer_mic_enc[BUFFER_PACKET_SIZE];

BUFFER_PACKET_SIZE 为 128，由于 int 占 4 个 bytes，等同于 128*4 个 signed char。

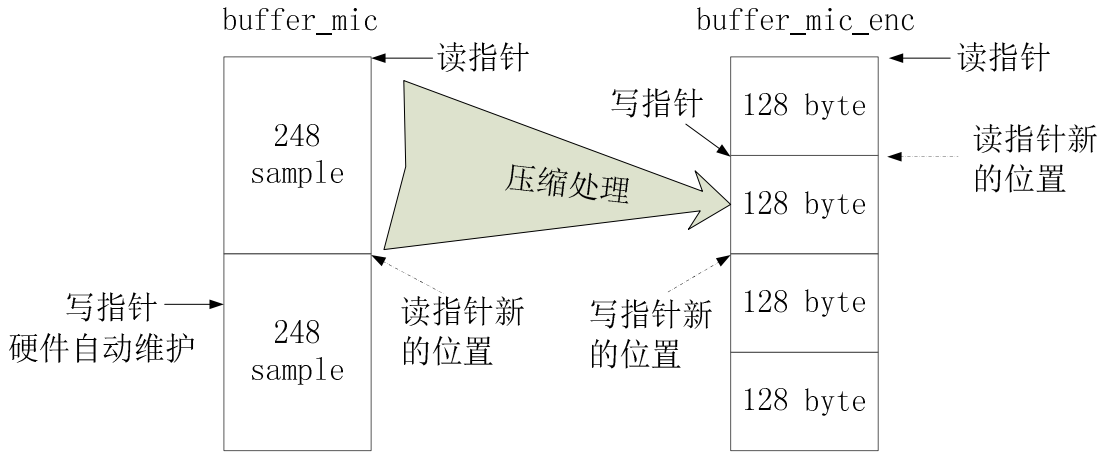

图 30 压缩处理方法

上图所示为压缩处理的基本方法。

buffer_mic 自动维护一个硬件写指针，同时在软件上维护一个读指针，当软件上检测到写指针与读指针中间的差值已经满足 248 个 sample 时，就开始调用压缩处理函数，从读指针开始取出 248 个 sample 的数据压缩为 128 个 bytes，同时将读指针移到图上新的位置，表示最新的未读的数据从新的位置开始。如此循环往复，不断检测是否是有足够的 248 个 sample 的数据，只要达到这个数据量，就开始做压缩处理。由于 248 个 sample 的产生时间为 15.5ms，需要保证程序至少 15.5 ms 才查询一次。由前面的介绍可知，程序在每个 main_loop 只执行一次 tack_audio，那么 main_loop 的时间必须小于 15.5 ms 才能保证音频数据不丢。在连接状态，main_loop 的时间等于 connection interval，所以有音频任务的应用，connection interval 一定要小于 15.5 ms。实际应用中推荐 10 ms 或 7.5 ms，目前的 SDK 使用了 7.5 ms。

buffer_mic_enc 在软件上维护写指针和读指针，当 248 sample 数据压缩为 128 bytes 后，

将这个 128 个 bytes 拷贝到写指针开始的地方，拷贝完之后检查一下这个 buffer 是否溢出，若溢出，将最老的一笔数据放弃（将读指针向后移动 128 bytes 即可）。

压缩后的数据拷贝到 BLE RF 数据发送缓冲区的方法为：检查 buffer_mic_enc 是是否为非空（写指针和读指针相等时为空，不等为非空），若非空，从读指针开始的地址拿出 128 bytes 拷贝到 BLE RF 数据发送缓冲区（若 security 打开时，多一步加密），然后将读指针移到图上所示新的位置。

以上音频数据压缩处理对应的代码为：

```
void   proc_mic_encoder (void)
{
    static u16 buffer_mic_rptr;
    u16 mic_wptr = reg_audio_wr_ptr;
    u16 l = ((mic_wptr<<1) >= buffer_mic_rptr) ? ((mic_wptr<<1) -
buffer_mic_rptr) : 0xffff;

    if (l >=(TL_MIC_BUFFER_SIZE>>2)) {
        log_task_begin (TR_T_adpcm);
        s16 *ps = buffer_mic + buffer_mic_rptr;
        mic_to_adpcm_split ( ps, TL_MIC_ADPCM_UNIT_SIZE,
                    (s16 *)(buffer_mic_enc + (ADPCM_PACKET_LEN>>2) *
                    (buffer_mic_pkt_wptr & (TL_MIC_PACKET_BUFFER_NUM
- 1))), 1);

        buffer_mic_rptr = buffer_mic_rptr ? 0 : (TL_MIC_BUFFER_SIZE>>2);
        buffer_mic_pkt_wptr++;
        int pkts = (buffer_mic_pkt_wptr - buffer_mic_pkt_rptr) &
(TL_MIC_PACKET_BUFFER_NUM*2-1);
        if (pkts > TL_MIC_PACKET_BUFFER_NUM) {
            buffer_mic_pkt_rptr++;
            log_event (TR_T_adpcm_enc_overflow);
        }
        log_task_end (TR_T_adpcm);
    }
}
```

5.3 压缩与解压缩算法

压缩算法调用的函数为：

void mic_to_adpcm_split (signed short *ps, int len, signed short *pds, int start);

ps 指向压缩前数据内存的首地址，对应图 30 中 buffer_mic 的读指针的位置，len 取 TL_MIC_ADPCM_UNIT_SIZE（248），表示 248 个 sample，pds 执行压缩后数据内存的首地址，

对应图 30 中 buffer_mic_enc 写指针的位置。

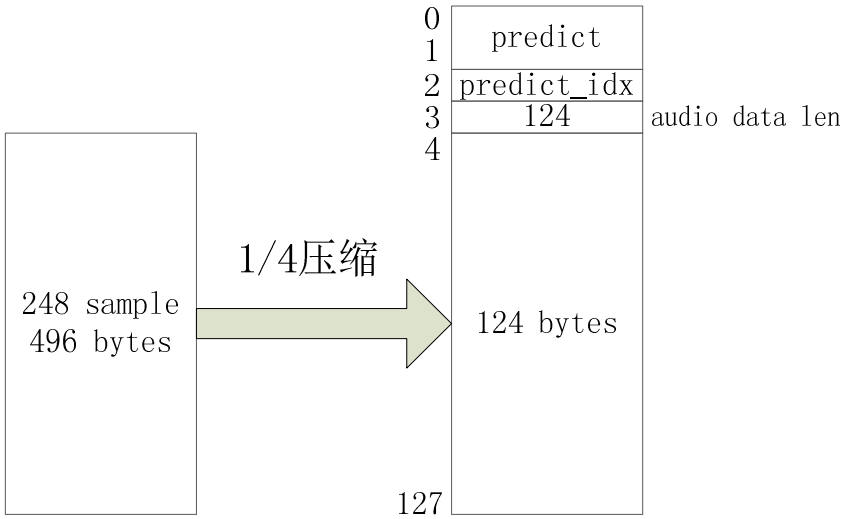

图 31 压缩算法对应数据

如上图所示，压缩后的数据内存的前两个 bytes 存 predict，第三个 byte 存 predict_idx，第 4 个 byte 为当前 adpcm 格式的 audio 数据的有效数据量，也就是 124，后面的 124 个 bytes 由 496d bytes 的原声数据 1/4 压缩而来，压缩的具体算法不介绍，只能能根据这个方法对应解压缩即可。

解压缩算法对应函数为：

void adpcm_to_pcm (signed short *ps, signed short *pd, int len);

ps 指向需要解压缩的数据内存开始的地址，也就是指向 128 bytes 的 adpcm 格式数据，这个地址需要 user 定义 buffer，从 BLE RF 收到的 128 bytes 数据拷贝到该 buffer；pd 指向解压缩后还原的 496 bytes pcm 格式音频数据内存开始的地址，这个需要 user 定义 buffer，播放声音时直接从这 buffer 拿数据；len 与压缩端长度一样，为 248。

解压缩的时候，对应图 31 所示，从前两个 bytes 读到的数据为 predict，第三个 byte 为 predict_idx，第 4 个为 audio 数据有效长度 124，后面的 124 bytes 对应转换为 496 pcm 格式 audio 数据。

5.4 Telink master 端语音数据处理

待补充。

6 OTA

6.1 FLASH 存储结构设计

为了实现 8267 slave 的空中下载，我们需要一个设备作为 BLE OTA master。OTA master 可以是实际与 slave 配套使用的蓝牙设备（须在 APP 实现里 OTA ），也可以使用 telink 的简易 BLE master dongle，下面以 telink 的 kma dongle 作为 ota master 为例说明 8267 实现 OTA 的方法。

由于 8267 芯片支持 flash 多地址启动（从 0、0x20000、其他地址启动），flash 的存储结构比较简单。

注：所有的 firmware 应不大于 64K，即 flash 的 0-0x10000 之间的区域可以存储。

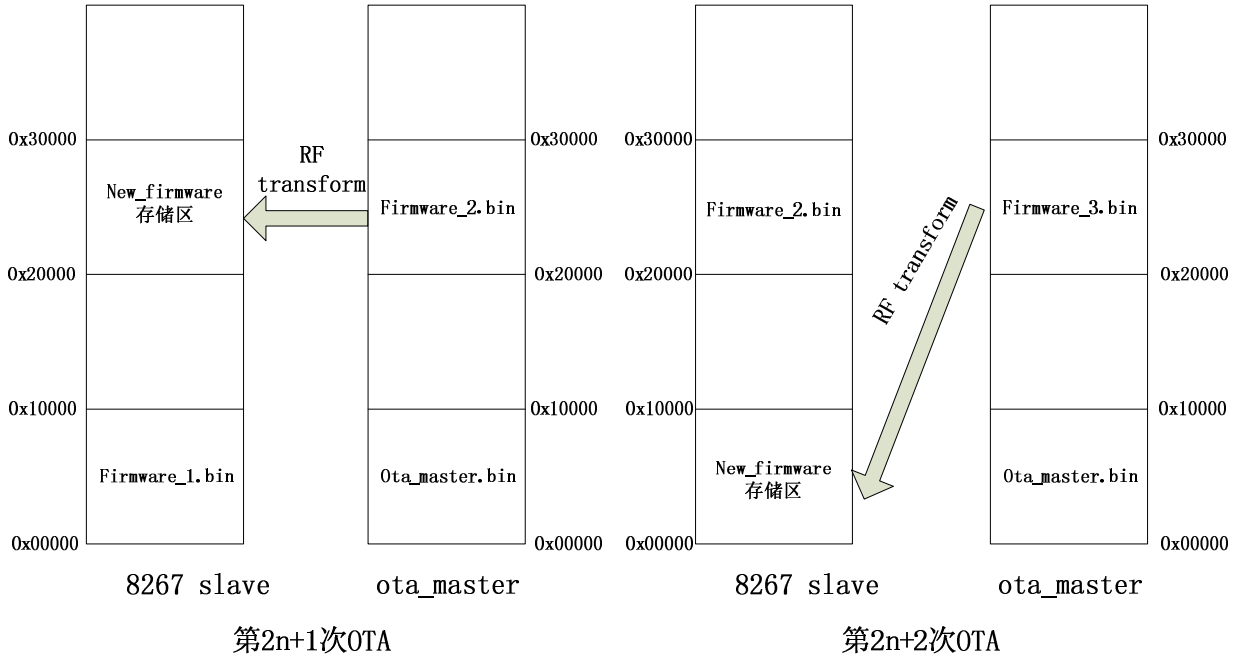

图 32 flash 存储结构

1) ota master 将新的 firmware2 烧写到 0x20000-0x30000 的区域。

2) 第 1 次 OTA：

   A. slave 上电时从 flash 的 0-0x10000 区域读程序启动，运行 firmware1；

   B. firmware1 运行，初始化的时候将 0x20000-0x30000 区域清空，该区域将作为新的 firmware 的存储区。

   C. 启动 OTA，master 通过 RF 将 firmware2 空运到 slave 的 0x20000-0x30000 区域。slave

reboot（重新启动，类似一个断电并重新上电）

3) ota master 将新的 firmware3 烧写到 0x20000-0x30000 的区域。

4) 第 2 次 OTA：

   A. slave 上电时从 flash 的 0x20000-0x30000 区域读程序启动，运行 firmware2；

   B. firmware2 运行，初始化的时候将 0-0x10000 区域清空，该区域将作为新的 firmware 的存储区。

   C. 启动 OTA，master 通过 RF 将 firmware3 空运到 slave 的 0-0x10000 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，(3)代表第 2n+2 次 OTA。

6.2 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 8267 多地址启动机制，由于 OTA 只用到了两个地址启动，这里只说前两个：MCU 上电后，默认从 0 地址启动，首先去读 flash 0x8 的内容，若改值为 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 0x8 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 0x8 和 0x20008 标志位的值，即可指定 MCU 执行 FLASH 哪部分的代码。

对于一个 OTA 功能已经设计好的 8267 SDK，MCU 某一次（2n+1 或 2n+2）上电及 OTA 过程为：

1) MCU 上电，通过读 0x8 和 0x20008 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 开始区域，；若从 0x20000 启动的，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，先发一个 OTA_start 命令通知 slave 进入 OTA 模式。

5) Slave 收到 OTA start 命令后，进入 OTA 模式，等待 master 发 OTA 数据。

6) 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。

7) Slave 接收 OTA 数据，向 ota_program_offset 开始的区域存储。

8) Master 发完 OTA 数据后，发一个 OTA_END 命令。

9) Slave 收到 OTA_END 命令，将新 firmware 区域偏移地址 8（即 ota_program_offset+8）写为 0x4b，将之前老的 firmware 存储区域偏移地址 8 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。

10) 将 slave reboot，新的 firmware 生效。

以上流程 slave 端相关操作在 8267 BLE SDK 中已经实现，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。

6.3 OTA 模式 RF 数据处理

6.3.1 Slave 端 Attribute Table 中 OTA 的处理

首先需要在 Attribute Table 所在的 app_att.c 中添加库里面 ota 的引用

```
#include "../../proj_lib/ble_l2cap/ble_ll_ota.h"
```

其次，在 Attribute Table 中添加 OTA 的相关内容，其中 OTA 数据 Attribute 的 att_readwrite_callback_t 和 att_readwrite_callback_t 分别设为 otaRead 和 otaWrite，将属性设为 Read 和 Write_without_Rsp(Master 通过 Write Command 发数据，不需要 slave 回 ack，速度会更快)。

```
static u8 my_OtaProp = CHAR_PROP_READ | CHAR_PROP_WRITE_WITHOUT_RSP;

{0,2,1,1,(u8*)(&my_characterUUID), (u8*)(&my_OtaProp), 0},
{0,2,1,1,(u8*)(&my_OtaUUID),      (&my_OtaData), &otaWrite, &otaRead},
{0,2,sizeof (my_OtaName), sizeof (my_OtaName),(u8*)(&userdesc_UUID),
                        (u8*)(my_OtaName), 0},
```

在 Attribute Table 中计算 OTA 数据 Attribute 的 handler_value，master 端发 OTA 命令和 OTA 数据时需要用到这个值。

6.3.2 OTA 数据 packet 格式

Master 端通过 L2CAP 层的 Write Command 向 slave 发命令和数据。

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

图 33 BLE 协议栈 Write Command 格式

Attribute Handle 的值为 slave 端 OTA 数据的 hander_value。Attribute Value 长度设为 20，格式见下图。

OTA_cmd

0	1	19
OTA_CMD	invalid data	

OTA_data

0	1	2	17	18	19
adr_index	firmware data:adr_index*16 - adr_index*16+15			CRC	

图 34 OTA 命令和数据的格式

前两个 byte 的范围为 0xff00 - 0xff10 时表示一个 OTA 的命令，命令类型由这两个 byte 的值决定：

1) 0xff00 为 OTA_FW_VERSION，要求获得 slave 当前 firmware 版本号，此命令 SDK 中暂时未处理
2) 0xff01 为 OTA_Start 命令

3) 0xff02 为 OTA_end 命令

前两个 byte 的范围在 0-0x1000 之内表示一个 OTA 数据。由于 firmware size 不超过 64K（0x10000），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 18 个 byte 进行一个 CRC_16 计算得到第一个 CRC 的值，slave 收到 OTA data 后，会对钱 18 个 byte 进行同样的 CRC 计算，只有当计算结果与第 19、20 byte 的 CRC 吻合时，才认为这是一个有效的数据。

6.3.3 master 端 RF transform 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 check 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。

ota master 软件上对 RF transform 的处理流程为：

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) 启动 OTA 开始的一个计时，后面要不断检测该计时是否超过 15 秒，如果超过 15 秒认为 OTA 超时失败，因为 slave 端收到 OTA 数据后会 check CRC，一旦 CRC 错误或者出现其他错误（如烧写 flash 错误），就认为 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。
3) 读取 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 20018-0x2001b 可以读到 firmware 的大小。
4) 向 slave 发一个 OTA start 命令 0xff01，通知 slave 进入 OTA 模式，等待 master 端的 OTA 数据
5) 从 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。
6) firmware 数据发送完毕后，发送 ota_end 命令，通知 slave 所有数据已发送完毕。

7) 检查 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，也就是说 slave 端的 OTA 数据接收全部 OK，OTA 成功。

OTA master 参考代码见本文档附件。CRC_16 计算函数见本文档附件。

6.3.4 slave 端 RF receive 处理方法

按照前面所述，Slave 端在 OTA Attribute 中直接调用 otaWrite 和 otaRead 即可，master 端发送的的 write command 命令过来，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 进行相应的相应，对 OTA 数据进行 CRC check 并烧写到 flash 对应位置。根据 packet 前两个 byte 内容对应的操作为：

1) 0xff00：OTA_FIRMWARE_VERSION 命令，master 要求获得 slave firmware 版本号，这个命令暂时未处理。
2) 0xff01：OTA start 命令，此时 slave 进入 OTA 模式，若用户使用 void ble_setOtaStartCb(ota_startCb_t cb)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，MUC 重启，master 端由于收不到 slave 的 ack 包，也会由于 15 秒超时使得软件发现 OTA 失败。
3) 0-0x1000：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 check，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) 0xff02：OTA end。此时所有的 firmware 数据已经更新完毕，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。

slave 端 otaWrite 函数键 SDK proj_lib/ble_l2cap/ble_ll_ota.c,相关接口见 SDK 中

proj_lib/ble_l2cap/ble_ll_ota.h。

ota_success_1.psd 和 ota_success_2.psd 是两个完整的 OTA 过程抓到的数据包，作为参考。

7 按键扫描

待补充

8 基本调试方法

待补充

9 常见问题解答

待补充

10 附件

10.1 附件 1：OTA master 参考代码

```
typedef struct{
    u32 dma_len;          //won't be a fixed number as previous, should
adjust with the mouse package number

    u8  type;             //RFU(3)_MD(1)_SN(1)_NESN(1)-LLID(2)
    u8  rf_len;              //LEN(5)_RFU(3)
    u16 l2cap;
    u16 chanid;

    u8  att;
    u8  hl;               // assigned by master
    u8  hh;               //

    u8  dat[23];

}rf_packet_att_data_t;


u8 write_ota_cmd[] = {
    0x0b,0,0,0,    //dma_len
    0x02,      //type
```

```
    0x09,      //rf_len
    0x05,0x00,  //l2caplen
    0x04,0x00,  //chanid
    0x52, //opcode = Read Request
    REMOTE_OTA_DATA_HANDLE, 0x00, //handle
    0x00,0x00,  //13  14
};

#define CMD_OTA_FW_VERSION                  0xff00
#define CMD_OTA_START                       0xff01
#define CMD_OTA_END                         0xff02

rf_packet_att_data_t ota_buffer[8];
u32 flash_adr_ota_master = 0x20000;
u8 *p_firmware;
u32 n_firmware = 0;
u32 ota_adr = 0;
u8 host_ota_start;

//UI design: indicate OTA result
void ota_set_result(int status)
{
    host_ota_start = 0;
    pc_tool_send_ota_cmd = 0;
    gpio_write(GPIO_LED_WHITE,status);  //ota fail
    write_reg8(reg_cmd_result,status);  //1 OK  0 fail
    SET_CMD_DONE;
}

void proc_ota ()
{
    static u32 tick_page;
    //add data when master is not in BTX state
    extern u32 ble_master_link_tick;
    extern int  push_tx_fifo_enable;
    if(!clock_time_exceed(ble_master_link_tick,800)
|| !push_tx_fifo_enable){
        return;
    }

    if(host_ota_start == 0)
    {
        gpio_write(GPIO_LED_BLUE,!LED_ON_LEVAL);
```

```
        if(pc_tool_send_ota_cmd){  //UI: pc tool trig OTA mode
            gpio_write(GPIO_LED_BLUE,LED_ON_LEVAL);  //ota begin
            host_ota_start = 1;
        }
    }
    else if (host_ota_start == 1)
    {
        n_firmware = *(u32 *)(flash_adr_ota_master+0x18);
        if(n_firmware > (124<<10)){  //bigger then 124K or 0xffffffff  is
ERR
            host_ota_start = 0;
        }

        p_firmware = (u8 *)flash_adr_ota_master;
        host_ota_start = 2;
        ota_adr = 0;

        //reg_dma_tx_rptr = FLD_DMA_RPTR_CLR;  //clear fifo
        write_ota_cmd[14] = CMD_OTA_START>>8;  //send ota start
        write_ota_cmd[13] = CMD_OTA_START&0xff;
        reg_dma_tx_fifo = (u16) write_ota_cmd;

        tick_page = clock_time ();
    }
    else if (host_ota_start == 2)
    {
        //UI design: indicate OTA fail
        if(clock_time_exceed(tick_page, 15*1000*1000)){  //OTA fail
            ota_set_result(0); //UI design: indicate OTA fial
            return;
        }
        int idx = (ota_adr >> 4) & 7;
        rf_packet_att_data_t *p = &ota_buffer[idx];

        int nlen = ota_adr < n_firmware ? 16 : 0;

        p->type = 2;
        p->l2cap = 7 + nlen;
        p->chanid = 0x04;
        p->att = ATT_OP_WRITE_CMD;
        p->hl = REMOTE_OTA_DATA_HANDLE;
        p->hh = 0x0;
        p->dat[0] = ota_adr>>4;
        p->dat[1] = ota_adr>>12;
```

```
        if(nlen == 16){
            memcpy(p->dat + 2, p_firmware + ota_adr, 16);
        }else{
            p->dat[0] = CMD_OTA_END&0xff;    //ota end cmd
            p->dat[1] = CMD_OTA_END>>8;
            memset(p->dat + 2, 0, 16);
        }
        u16 crc = crc16(p->dat, 2+nlen);
        p->dat[nlen + 2] = crc;
        p->dat[nlen + 3] = crc >> 8;
        p->rf_len = p->l2cap + 4;
        p->dma_len = p->l2cap + 6;

        if (ble_master_add_tx_packet ((u32)p))
        {
            ota_adr += 16;
            if (nlen == 0)  //all data push fifo OK
            {
                host_ota_start = 3;
            }
        }
    }
    else if(host_ota_start == 3){
        if(master_hw_fifo_data_num() == 0 ){ //all data acked,OTA OK
            ota_set_result(1);  //UI design: indicate OTA OK
        }
    }
}
```

10.2 附件 2：crc_16 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;
}
```