目录

图目录

软件组织架构

17H26 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。有 3 个主要的顶层文件夹：proj，proj_lib，vendor。

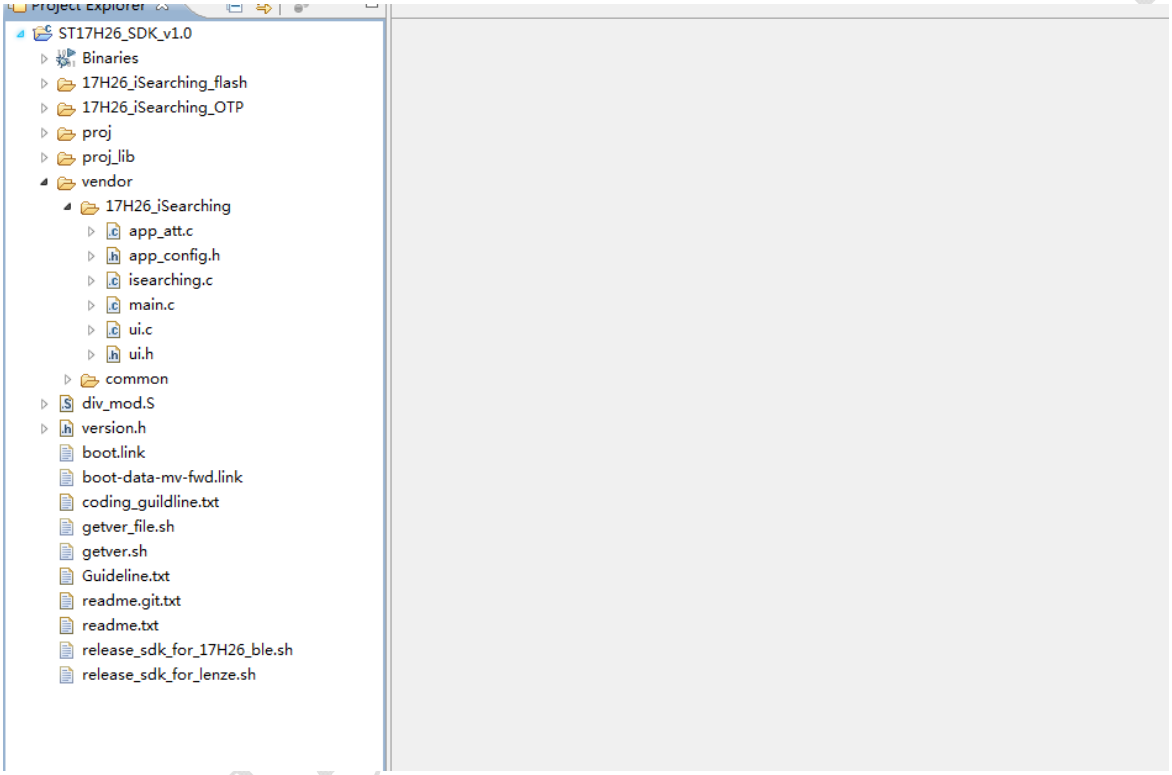

图 0-1 SDK 文件结构

✧ proj：提供 MCU 相关的外设驱动程序，如 flash，i2c，usb，gpio 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，17H26_iSearching 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：

- main.c
- app_config.h
- app_att.c
- 用户文件

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 中配置相关参数即可
    gpio_init();           //gpio 初始化，user 在 app_config.h 中配置相关参数即可
    rf_drv_init(CRYSTAL_TYPE);  //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
#if(MODULE_ADC_ENABLE) // adc 初始化配置，user在app_config.h中配置相关参数
    adc_init();    //  adc 初始化用户不需要修改此项。
#endif
    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 lenze 定义的 Attribute 结构，根据该结构，已提供 GATT、标准 HID 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 17H26_iSearching.c：用户文件，用于完成系统的初始化(user_init)、控制系统运行的过程(main_loop)和添加用户的 task UI。

2 MCU 基础模块

这部分主要对 17H26 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

17H26 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 为外部设备空间（例如：SRAM、寄存器空间）。

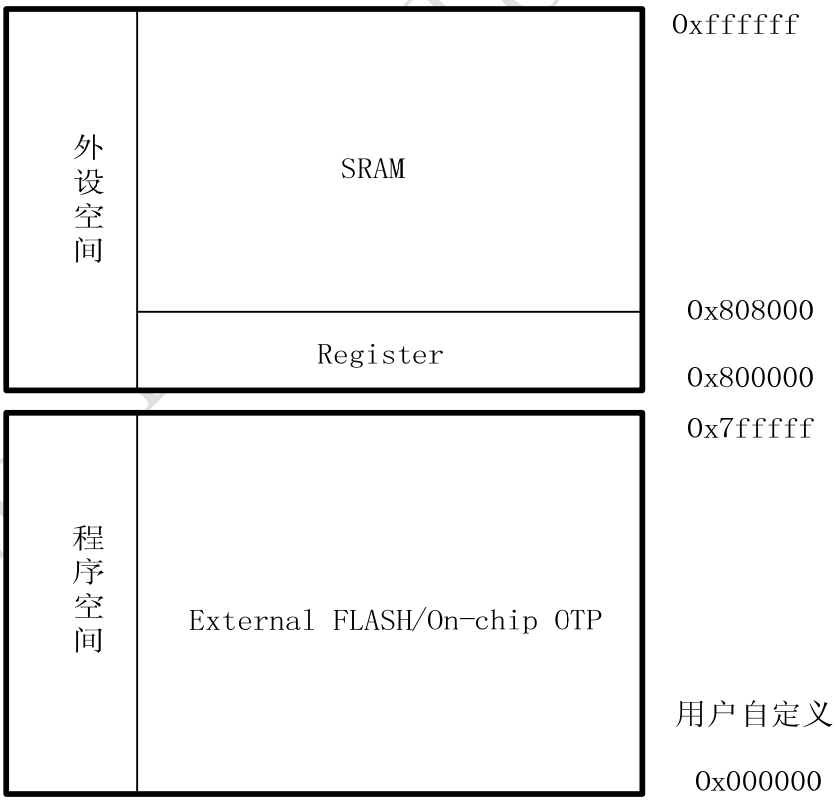

图 2-1 MCU 地址空间分配

2.1.2 MCU 地址空间访问

程序中对 0x000000 - 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。

17h26 firmware 可以选择存储在外部 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) 读 OTP 通过指针访问。

17H26 firmware 也可以选择存储在 OTP 上。17H26 OTP 的空间大小为 16M，即 0x000000 ~ 0x003fff，一般情况下，建议用户通过指针访问 OTP。程序运

行时，只是将 OTP 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 OTP 读到 ram 高速缓存区域。MCU 通过自动控制内部硬件模块，读取 OTP 上的内容。

使用指针的形式访问 OTP 上的内容，如：

```
u16   x = *(volatile u16*)0x3fea;   //读 otp 0x3fea - 0x3feb 两个 byte 内容
```

4) 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。
```

5) 写 OTP：17H26 **不支持**自定义修改 OTP。

2.1.3 SDK FLASH 空间的分配

FLASH 存储信息以一个 sector 的大小（根据不同的 flash 型号，sector 大小不同，通常情况下大小为 4K byte）为基本的单位，因为 flash 的擦除是以 sector 为单位（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。

user 在配置使用自己的 sector 存储信息时，须注意一点：在 32M 系统时钟下，flash_erase_sector 函数擦除一个 sector 大概会需要 20-50 ms 之间，所以不要在 main_loop 运行后调用该函数（若是在连接状态会破坏 BLE 时序）。

2.1.4 SDK OTP 空间的分配

因为 17H26 内置 OTP 大小为 16K，最大寻址空间是 0x3fff，空间使用比较紧张，

另外 17H26 不支持自身程序自主修改 OTP 数据，所以相关的参数的地址定义比较紧凑。但是要遵循的原则是相关地址的设置一定不能与 Firmware 空间冲突。建议如下：

地址	参数定义	长度	描述
0x3fe0	Mac 地址	6	BLE 设备的 Mac 地址
0x3fe8	频偏校准值	1	校准设备频偏，需要通过治具外部校准
0x3fea	TP 值（TP0）	1	校准设备 TP 值，需要通过治具外部校准
0x3feb	TP 值（TP1）	1	校准设备 TP 值，需要通过治具外部校准

注意：以下是 17H26 系统参数地址，用户不允许使用以及占用：

（1） OTP: 0x3fee.

（2） OTP: 0x3ff0~0x3fff.

以上地址都是靠近 OTP 空间的末尾处，尽量给 Firmware 保留足够的使用空间。

2.2 时钟模块

系统时钟 system clock 是 MCU 执行程序的时钟，虽然 17H26 的系统时钟的时钟源有多种（PLL、内部 OSC、内部 RC），但是在 17H26 SDK 中，我们只用 PLL，因为 PLL 时钟是最精准的达到 BLE 时序的要求。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。

17H26 BLE SDK 可以使用两种外部晶振，16M 和 12M，目前默认是使用 12M 晶振，在 app_config.h 中可以配置：

```
//////////////////Extern Crystal Type///////////////////////
#define CRYSTAL_TYPE            XTAL_12M        //  extern 12M crystal
```

可选的两个值为如下所示，在 main 的 rf 初始化时，代入 user 配置的晶体类型。

```
enum{
    XTAL_12M   = 0,
    XTAL_16M    = 1,
};
```

调用接口为main函数中rf_drv_init(CRYSTAL_TYPE);

需要注意的是，外部晶振的类型（12M、16M）与最终程序运行的系统时钟是两个不同的概念：前者是硬件晶体的具体规格，后者是 MCU 运行时的机器周期。不管外部晶振选哪个，MCU 都会通过内部的 PLL 电路倍频处理后获得 192M 的一个基础时钟。根据 user 在 app_config 中配置的系统时钟的频率，在 main 函数的 clock_init() 阶段对 192M 基础时钟进行分频，获得 user 需要的频率。

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  32000000         //system clock 32M
```

目前17H26 BLE SDK推荐时钟频率为32M，我们平时所有的调试都是基于该频率。注意：17H26 的 BLE 时序只能使用 32M 的系统时钟。

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，32M 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 1/32 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：(1/32) us * (2^32) 约等于 128s，system clock tick 每过 128s 转一圈。

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备性，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 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 32M 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 * 32)
```

实际上 SDK 中为了解决系统时钟不同导致的 u32 转换问题，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 32M 时钟转一圈为 128s，这个查询函数只适用于 128s 以内的定时，若超过 128s，需要在软件上加计数器累计实现。

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

17H26 SDK 共有 37 个 GPIO，分别为：

GPIO_GP0 ~GPIO_GP31、GPIO_SWS、GPIO_MCLK、GPIO_MSDO、GPIO_MSDI、GPIO_MSCN。

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_17H26.h。

使用注意：

1. GPIO_MSDO、GPIO_MSDI、GPIO_MSCN、GPIO_MCLK 这四个 IO 为外接 FLASH 时使用的 IO，当硬件使用 flash 存储时，不建议用户使用这四个 IO 做 GPIO 模式使用。
2. GPIO_SWS 用于 debug 程序以及烧录程序使用。通常情况下，不建议用户将此 IO 口用作 GPIO 模式。

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。

37 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。

这里要非常注意，当读到高电平时，返回值不一定是 1，是一个非 0 的值！

详情见下面的 code。

```
static inline u32 gpio_read(u32 pin)
{
    return BM_IS_SET(reg_gpio_in(pin), pin & 0xff);
}
```

所以程序中，不能使用类似 if(gpio_read(GPIO_GP0) == 1)的写法，推荐使用方法是将读到的值取反处理，取反后只有 1 和 0 两种情况：

```
if( !gpio_read(GPIO_GP0) )  //判断高低电平
```

在 17H26 中，要保证 gpio 的输入电压大于系统供电电压的 2/3，才能保证 gpio_read()的返回值非 0。gpio 的输入电压小于 0.7V 时 gpio_read()返回值为 0。所以

在系统供电电压的 2/3 和 0.7V 之间是不确定的状态。 例如：如果系统当前供电电压为 3.0V，大于 2.0V 以上为高，小于 0.7V 为低，介于 0.7V 到 2.0V 为不确定状态。

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) ds(data strength): 控制 gpio 的输出负载能力。

```
void gpio_set_data_strength(u32 pin, u32 value);
```

value 1 和 0 分别表示 ds 设置与不设置。

17H26 每个 gpio 的 ds 设置与不设置时的负载电流可参照 Datasheet 上 GPIO 章节的 Drive Strength 描述。

7) 内部上下拉电阻配置：有 1M 上拉、10K 上拉、100K 下拉和 float（悬空）共四种状态。

对 IO 口的内部上下拉电阻的配置在 app_config.h 中，对相应 IO 的上下拉进行赋值。格式为：

```
PULL_WAKEUP_SRC_GPIO#n#  （n 为 0~31）
```

所有未在 app_config.h 中宏定义的 IO，其默认上下拉电阻为悬空。

四种上下拉电阻的宏定义值为：

PM_PIN_PULLUP_1M

PM_PIN_PULLUP_10K

PM_PIN_PULLDOWN_100K

PM_PIN_UP_DOWN_FLOAT

注意：不是所有的 IO 都有这四种状态，部分 IO 只有下拉 100K 和悬空状态，部分 IO 没有内部上下拉状态。

IO 口对应的内部状态如下：

IO	内部上下拉电阻
GPIO_GP17 ~ GPIO_GP24 GPIO_GP26 GPIO_GP27 GPIO_GP31	PM_PIN_PULLUP_1M PM_PIN_PULLUP_10K PM_PIN_PULLDOWN_100K PM_PIN_UP_DOWN_FLOAT
GPIO_GP0 ~ GPIO_GP16 GPIO_GP32	PM_PIN_PULLDOWN_100K PM_PIN_UP_DOWN_FLOAT
GPIO_MCLK GPIO_MSCN GPIO_MSDO /GPIO_MSDI GPIO_SWS	无内置上下拉电阻

示例：定义 GP2 为 100K 下拉，GP17 为 1M 上拉：

```
#define  PULL_WAKEUP_SRC_GPIO2    PM_PIN_PULLDOWN_100K
#define  PULL_WAKEUP_SRC_GPIO17   PM_PIN_PULLUP_1M
```

GPIO 配置应用举例：

1) 将 GPIO_GP4 配置为输出态，并输出高电平

```
gpio_set_func(GPIO_GP4, AS_GPIO) ;  // GP4 默认为 GPIO 功能，可以不设置
gpio_set_input_en(GPIO_GP4, 0);
```

```
gpio_set_output_en(GPIO_GP4, 1);

gpio_write(GPIO_GP4,1)
```

2) 将 GPIO_GP17 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

```
#define  PULL_WAKEUP_SRC_GPIO17   PM_PIN_PULLUP_1M  //在 app_config.h 中配置。

gpio_set_func(GPIO_GP17, AS_GPIO) ;  // GP17 默认为 GPIO 功能，可以不设置

gpio_set_input_en(GPIO_GP17, 1)

gpio_set_output_en(GPIO_GP17, 0);

if(!gpio_read(GPIO_GP17)){   //是否低电平

    ......

}
```

3) 将 MCLK 配置成 GPIO 模式。

```
gpio_set_func(GPIO_MCLK, AS_GPIO) ;
```

2.3.3 GPIO 的初始化

main.c 中调用 gpio_init 函数，会将 17H26 所有的 GPIO 的状态都初始化一遍。

当用户的 app_config.h 中没有配置的 GPIO 参数时，这个函数会将每个 IO 初始化为默认状态。37 个 GPIO 默认状态为：

1) func：除了前面介绍的 5 个为特殊功能，其他 32 个（GPIO_GP0 ~ GPIO_GP31）均为一般 GPIO 状态。

2) ie：17H26 的 SDK 中，所有的 gpio 的 ie 状态在初始化的时候均为 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_17H26.h 和 proj/mcu_spec/gpio_default_17H26.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 GP7 的 ie 为例）：

```
#ifndef    GPIO7_INPUT_ENABLE
#define    GPIO7_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 GP7 为例）：

1) 配置 func: #define GPIO7_FUNCAS_GPIO

2) 配置 ie： #define GPIO7_INPUT_ENABLE 1

3) 配置 oe: #define GPIO7_OUTPUT_ENABLE 0

4) 配置 dataO：#define GPIO7_DATA_OUT 0

5) 配置内部上下拉电阻：

#define PULL_WAKEUP_SRC_GPIO7 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 2.3.2 节中 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_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。

2.3.4 配置 SWS 上拉防止死机

Lenze 所有的 MCU 都使用 SWS（single wire slave）来 debug 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。我们通过打开内部数字上拉来解决。在 ie 打开且 oe 关闭状态下，只要将 dataO 设置为 1，就是开数字上拉电阻（阻值不定，在 20k-50k 之间）。在 app_config.h 中如下定义即可。

```
//////////////open SWS digital pullup to prevent MCU err, this is must
#define SWS_DATA_OUT          1
```

建议：17H26 **SWS 内部没有上下拉电阻，建议用户使用硬件上外部 10K 上拉。**

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。

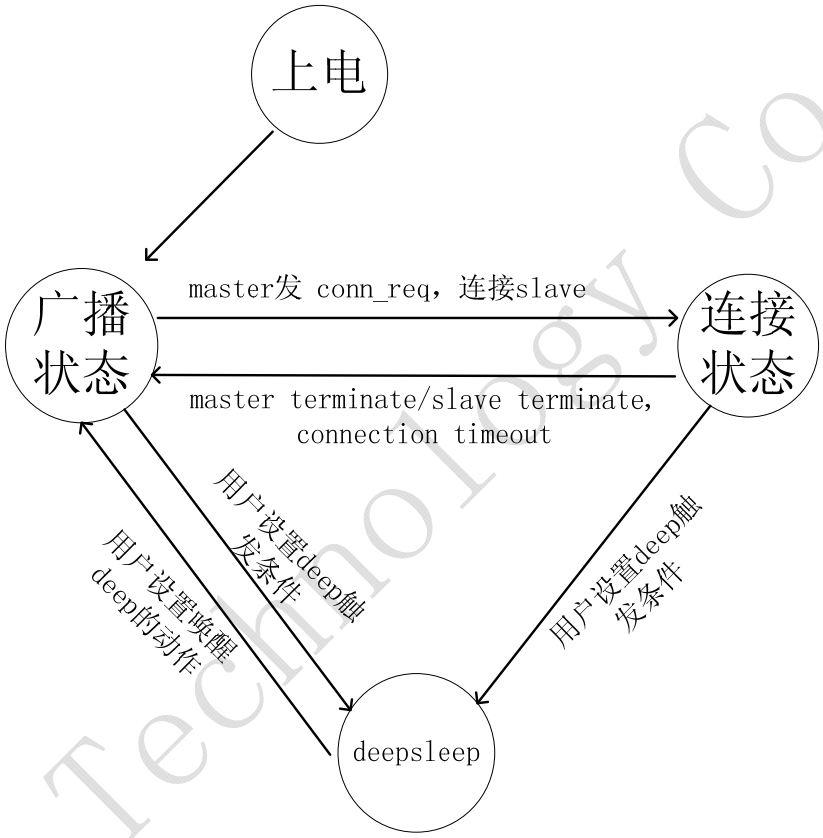

图 3-1 BLE 状态机

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 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。

17H26 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，注意只能读而不能进行写操作，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

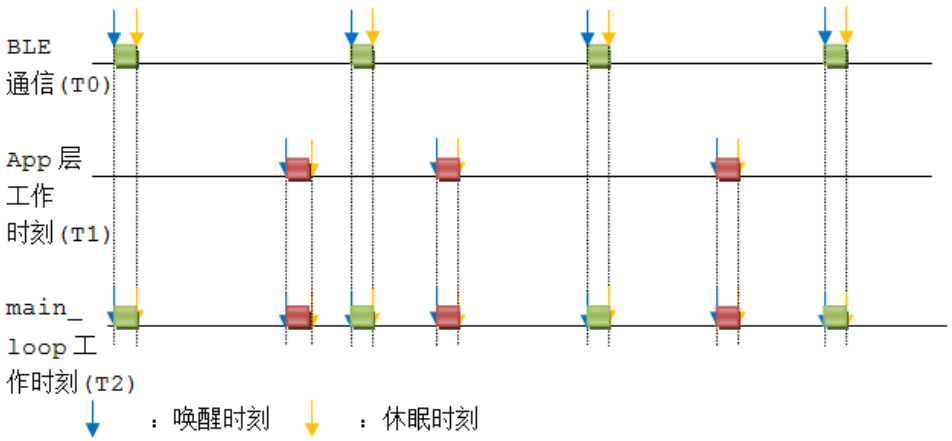

图 3-2 BLE 工作时序

一个具有低功耗管理的 slave 工作时序如上图所示。BLE slave 在每次 main_loop 的时刻（如上图 T2）是 BLE 工作时刻（T0）以及用户层需要的唤醒的时刻（T1）的综合时刻，即 T2 = T0 | T1；BLE T0 时序工作的时刻与设备端 adv_interval 以及 conn_interval 有关，其唤醒时间点为系统自动计算；T1 时间点为用户层所需要的 main_loop 的时间点，用于 UI 层的处理，比如 LED 的闪灯状况，该时间点是设置在睡眠函数 blt_sleep_wakeup()的参数，注意：1.该参数是一个相对的时刻值，比如如果需要在当前时刻的 20ms 之后做 main_loop，函数调用为 blt_sleep_wakeup(clock_time() + 20*CLOCK_SYS_CLOCK_1MS)。 2.当该函数的参数为 0 时，则系统认为用户层对 main_loop 没有需求，系统只会关注 BLE 唤醒的时刻。

如果去掉功耗管理，即 suspend_disable 的情况下，软件上不在会处理 suspend 的状态，系统会一直做 main_loop。强调一点，系统每次 main_loop 虽然会调用 blt_send_adv()或者 blt_brx()函数，系统也不一定会发广播，或者开始 BRX 过程（BRX 过程即使连接状态下的过程），除非到了 BLE 的时序时间。

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。根据 17H26 BLE SDK 的 main_loop 的写法，可以将每个 main_loop 划分为三个部分：T_D1、T_D2 和 T_D3，每个 adv_interval/conn_interval 可能包含一个或多个 main_loop。

```
void main_loop ()

{

    ////////////////////// BLE entry ////////////////////////////


    if (blt_state == BLT_LINK_STATE_ADV)
    {
        blt_send_adv (BLT_ENABLE_ADV_ALL);
    }
    else       // connection state
    {
        blt_brx ();

    }

    ////////////////////// UI entry ////////////////////////////

    proc_ui());



    ////////////////////// Suspend entry ////////////////////

    blt_brx_sleep (0);

}
```

1) T_D1 对应 BLE 时序处理的部分。此部分包含函数 blt_send_adv() 和 blt_brx()，此部分函数负责处理 BLE slave 所有 RF 相关工作：函数 blt_send_adv(int mask)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，参数 mask 用于标志广播使用的 channel，广播 channel 可以是 37、38、39channel 的任意组合；blt_brx()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。注意：为保证 BLE 的时序，在 main_loop 的过程中，程序即使调用了 blt_send_adv()和 blt_brx()也并不一定会做相关的 RF 数据处理，其是否处理数据由系统决定，即如果系统没有到 BLE 时序应该处理收发数据包的时间点，系统会立即退出 blt_send_adv 或者

blt_brx 函数。

2) T_D2 对应 UI entry，是供 user 处理 UI 的的部分，user 在这里处理自己的 UI 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 里有数据时，在下一次的 BLE 的数据交互 Timing 时，MCU 会自动将数据发送给 master。

3) T3 对应 Suspend entry 的 blt_brx_sleep (u32 app_wakeup_tick)。该函数处理在程序 suspend enable 的情况下，用户层需要设置一下 main_loop 的时间以及 suspend 的相关操作，包括设置 suspend 时间、设置 suspend 的唤醒源等。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。在 suspend disable 的情况下，程序进入此函数会马上退出。

为了让 user 更好的了解整个 main_loop 的工作机制，在本章介绍 BLE entry 底层工作机制，在 PM 那一章介绍 Suspend entry 底层工作机制。

3.3 BLE entry 工作机制

BLE entry 的主要工作是处理广播以及连接时的收发包过程，本节介绍这两个过程实现的方式，加深用户对此过程的理解。

3.3.1 广播过程

对于广播过程，17H26 提供了两个广播接口：blr_send_adv 和 blt_beacon_send_adv。毫无疑问，后者对应的是 beacon 设备的广播接口，前者对应的是非 beacon 设备的广播接口。但是一般用户在使用时，建议不要使用此接口。二者的差别主要如下：

a) blt_beacon_send_adv 广播接口占用的 Ram 空间更大，广播处理的时间相对较短，因此功耗会低一些。（这对于过程简单但是功耗要求又比较高的 beacon 设备来说是必要的。）

b) blt_beacon_send_adv 基于事件触发的回调函数相对于 blt_send_adv 多了两个：BLT_EV_FLAG_BEACON_DONE 和 BLT_EV_FLAG_ADV_PRE。后续会介绍。

以下介绍函数实现的伪代码，以及 ble_beacon_send_adv 使用过程中需要注意的地方：

（1） blt_send_adv（int mask）的实现伪代码如下：

```
u8 * blt_send_adv(int mask){
    // 1. 根据广播间隔（adv_interval）判断是否到了广播的时间点，误差 2ms。如果没有到广播时间，则 return 0。否则开始设置参数，进行广播。
```

```
        //2. 设置 RF 参数，进行广播。
        for(i=0; i<3; i++){
            if(mask & (1<<i)){
                //3. 设置发包 channel
                set_ble_channel(i + 37);
                //4. 发送广播包，等待发送完毕。
                //5. 设置接收模式。
                //6. 等待一段时间，判断是否接收到数据
                if(packet_received){
                    //7 收到 scan_req，开始发包。收到 conn_req，开始建立连接。
                }
            }
        }
    }
```

（2） 函数 blt_beacon_send_adv 声明如下：

```
u8*     blt_beacon_send_adv (int mask, u8 tx_pn_init_done);
```

参数“mask”与 blt_send_adv 的参数 mask 的意义相同，都是表示广播的 channel。

参数“tx_pn_init_done”表示是否使用提前计算好 pn 的值计算广播数据。如果使用提前计算好的 pn 的值计算广播数据，则 tx_pn_init_done 置 1，否则置 0。 如何使用计算好的 pn 值得到广播数据，步骤如下（可参考 beacon demonstration）：

1) 分别计算 37,38,39 channel 的 pn 值。

2) 根据相应 channel 的 pn 值以及广播数据（tbl_adv）计算各 channel 的数据。

3) 将各 channel 的指针填进 extern u8* beacon_adv_tx_pn_ptr[]; 该 buffer 包含三

个指针，即 37/38/39 channel 经过 pn 计算的广播数据的指针。blt_beacon_send_adv 会直接发送 beacon_adv_tx_pn_ptr buffer 中对应的指针。 因为每个 channel 对应的 pn_buffer 有 48 byte，每个 channel 对应的 pn 数值又不相同，所以虽然广播数据相同，但是 pn 计算之后的数据差别会比较大，每个 channel 需要有独立的 buffer（RAM 空间）存储该数据，因为占用的 RAM 空间是比较大的。

以上实现过程较为复杂，非 demonstration 实现的功能不建议用户独立使用此函数。

3.3.2 连接过程

连接情况下，blt_brx() 函数实现的伪代码如下：

```
u8 blt_brx ( ){
    //1. 根据 conn_interval 判断是否到 BLE 时序的时间点，如果到了该时间点，则开始进行收发包的测试。如果未到该时间点，则退出该函数。
    //2. 符合 BLE 时序时间点，设置参数，开启收包模式。
    //3. 等待一段时间，判断是否收到 master 发过来的包
    if(packet_received ){
        // 4. 解析数据包。
        // 5. 设置 TX 模式，从 TX buffer 中取包，并且发送出去。如果 tx_buffer 中没有需要发送的数据包，则系统默认发送空包。
    }
}
```

3.4 基于事件触发的回调

虽然 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（比如开始建立连接时刻、断开连接时刻），以满足上层应用的需要。为了满足这类操作，17H26 BLE SDK 提供了相关的系统流程时间回调函数。17H26 的回调函数不需要函数注册，主要通过变量 ll_event_cb_flag 的赋值情况进行相应的处理，并且相应的回调函数名是固定的，目前 17H26 SDK 定义的系统流程回调函数有 3 种，定义以及描述如下：

定义赋值	回调函数名字	描述
BLT_EV_FLAG_CONNECT	**task_connection_established**	连接建立时刻
BLT_EV_FLAG_TERMINATE	**task_connection_terminated**	连接断开时刻
BLT_EV_FLAG_BOND_START	**task_bond_finished**	开始 bond 时刻

赋值的宏定义如下：

```
#define          BLT_EV_FLAG_CONNECT          BIT(2)
#define          BLT_EV_FLAG_TERMINATE        BIT(3)
#define          BLT_EV_FLAG_BOND_START       BIT(4)
```

示例：

a) `ll_event_cb_flag = BLT_EV_FLAG_CONNECT ;`

解析：系统运行中connect事件触发时，**task_connection_established**会被调用，其他函数不会。

b) `ll_event_cb_flag = BLT_EV_FLAG_CONNECT | BLT_EV_FLAG_TERMINATE;`

解析：系统运行过程中，**task_connection_established** 和 **task_connection_terminated** 等函数会被调用，其他不会。

c) `ll_event_cb_flag = 0;`

解析：系统运行过程中，任何函数都不会被调用。

每个回调函数都带有一个参数，该参数的意义与回调函数的意义密切相关。以下对这三种回调函数进行解析。

3.4.1 BLT_EV_FLAG_CONNECT

事件触发时刻： Slave 收到 connection_req 包，并且处理完之后。

回调函数定义：

```
void task_connection_established(rf_packet_connect_t* p)
```

函数参数定义：函数参数是收到的 connection_req 的 packet 的指针，其类型为 rf_packet_connect_t。用户可以通过该指针变量获取 connection_req packet 的值。

3.4.2 BLT_EV_FLAG_TERMINATE

事件触发时刻：连接断开时刻。连接断开包括两种情况：（1）master 端主动发 terminate，注意回调该函数时，slave 已经回复 master terminate 的 ack，但是不会确认 master 是否收到。（2）slave 端一段时间（timeout）内没有收到数据包导致的断开。

回调函数定义：

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

函数参数定义：该函数用于判断是否是因为 master 主动断开。

```
u8 terminate_reason = *(u8*)p;
if(terminate_reason)
{
    // master 主动断开
}else{
    // timeout 导致的断开。
}
```

3.4.3 BLT_EV_FLAG_BOND_START

事件触发时刻：如果 BLE 的系统需要进行 SMP 过程，在系统收到 Start_Encryption_req 命令时，系统会调用此函数。 此函数表明基本配对过程的完成。

回调函数定义：

```
void task_bond_finished(u16* cur_instant);
```

函数参数定义：该函数的参数标志 slave 收到 start_encrystion_req 命令的 instant，类型为 u16 *。

3.4.4 BLT_EV_FLAG_BEACON_DONE

该函数只用于广播函数为 blt_beacon_send_adv 的广播过程中。

事件触发时刻：每次广播发完之后。即一个广播的 Interval 中，发完广播之后，blt_beacon_send_adv 即将退出的时候。以下两种情况不会调用：（1）即使调用了 blt_beacon_send_adv，但是没有到发广播的时间点，函数直接退出，不发出广播包，也不会触发此事件。（2）收到 connect_req 命令时，也不会触发此事件。

回调函数定义：

```
void task_beacon_done(void );
```

3.4.5 BLT_EV_FLAG_ADV_PRE

该函数只用于广播函数为 blt_beacon_send_adv 的广播过程中。

事件触发时刻：每次广播函数执行之前，即发广播之前。但是不包含以下情况：调用 blt_beacon_send_adv，但是并没有到实际的发包时间点。

3.5 BLE 基本参数的配置

基于 17H26 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。

3.5.1 MAC 地址

BLE MAC address 的长度为 6 个 byte，因为 17H26 的空间有限，客户可根据自己的空间使用情况定义 MAC 地址在 Flash/OTP 的位置，尽量做到 Firmware 与 Mac 地址不冲突，具体的 MAC address 需要 user 从蓝牙组织获取。

目前调试阶段，Lenze 的 Demo 对 Mac 地址的使用方法如下，

```
#define  CFG_ADR_MAC                    0x3fe0
u8    tbl_mac [] = {0xe1, 0xe1, 0xe2, 0xe3, 0xe4, 0xc7};
u32 *pmac = (u32 *) CFG_ADR_MAC;
if (*pmac != 0xffffffff)
{
      memcpy (tbl_mac, pmac, 6);
}
```

如果相应的地址位置上的前四个byte值是无效的（等于0xffffffff），那么系统会使用默认的Mac地址值。

3.5.2 广播频点和广播时间间隔

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。每个 adv_interval 可能包含一个或多个 main_loop，且至少有一个 main_loop 的 T_D1 期间会发送广播数据包。

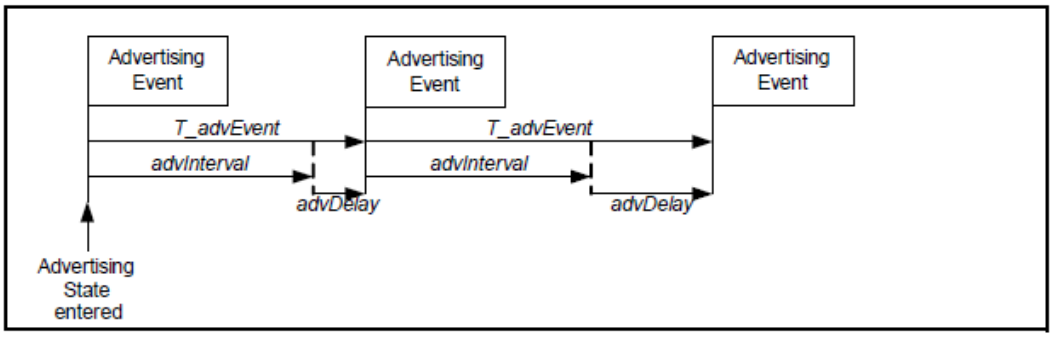

图 3-3 BLE 协议栈里 Advertising Event

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 应该再在三个广播 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。通常情况下，我们会通过发广播函数 blt_send_adv(int mask);的参数进行设置。

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)
```

2) 广播时间的设定

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。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）。

3.5.3 广播包、scan response 包的初始化

SDK 通过下面的的函数设定 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在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。

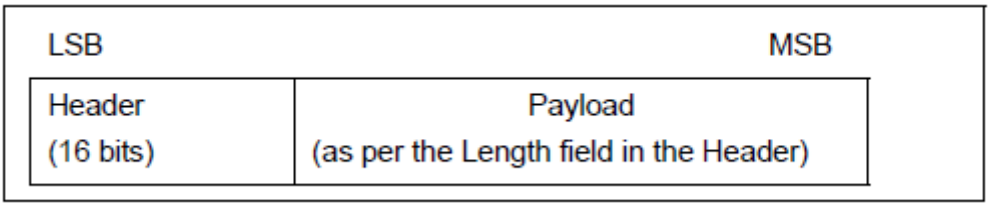

图 3-4 BLE 协议栈广播包格式

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

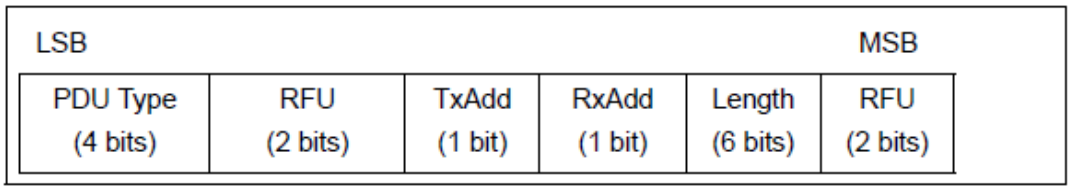

图 3-5 BLE 协议栈广播包 header 结构

BLE 协议栈里，广播包 header 结构如上所述，其中我们需要关注的是第一个 byte 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 3-6 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。

17H26 SDK 的广播包定义成一个数组，其格式如下：

1byte	1byte	6 bytes	N bytes	3 bytes
packet type	adv_len	adv addr	adv payload	crc check

注：

a) 广播包的 packet_type 根据 BLE 协议规定定义。如可连接广播包的 type = 0。

b) adv_len = 6 (adv addr length) + N (adv payload length)。

c) 虽然 adv_len 不包含 CRC check 的三个 byte，但是 adv tbl 的数组大小必须包含这三个 byte，所以 adv tbl 的数组大小必须大于等于 adv_len + 2 + 3。

adv tbl 示例及分析如下：

```
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)
        0, 0, 0,// CRC 部分，此长度不计入packet_length中
        };
```

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，此长度不包含最后三个 byte 的 CRC 计算长度，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"。

17H26 SDK 使用一个数组去定义 scan response，其格式如下：

1byte	1byte	6 bytes	N bytes	3 bytes
packet_type = 0x04	packet_length	packet addr	packet payload	crc check

注：

a) scan response table 的格式与广播包除了 packet_type 不一致之外，其他相同。

b) scan reponse type = 0x4。

c) packet_len = 6 (packet addr length) + N (packet payloadlength)。

d) 与广播包相同，即使 crc_check 的 length 不计算在 payload_length 中，但是 scan response table 的长度也必须包含这三个 byte 的位置，所以 scan response 的长度必须大于等于 packet_length +2 + 3。

scan_reponse 示例以及分析如下：

```
u8  tbl_rsp [] =
        {0x04, 14,                                  //type len
         0xef, 0xe1, 0xe2, 0xe3, 0xe4, 0xe5,        //mac address
         0x07, 0x09, 't', 'S', 'e', 'l', 'f', 'i'   //scan name " tSelfi"
        0, 0, 0,// CRC 部分，此长度不计入packet_length中
        };
```

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，此长度不包含最后三个 byte 的 CRC 计算长度，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备名称，后面 Attribute Table 部分再介绍。

3.5.4 广播包、scan response 包的修改

如上介绍，在 user_init 中使用了 blt_init 设定了广播包和 scan response 包的格式后，在程序的 main_loop 阶段如果需要更改新的广播包（比如需要使用 directable adv），那么用户需要定义新的广播包，并且广播包必须要符合 17H26 广播包的格式定义。

3.5.5 广播事件类型

<<Core_v4.1_BLE_spec.pdf>> P2526 开始的 NON-CONNECTED STATES 部分介绍了四种广播事件类型，如下图所示。

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

图 3-7 BLE 协议栈四种广播事件

3.5.6 BLE packet 能量设定

17H26 BLE SDK 提供了 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_17H26.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_7dBm = 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,
};
```

3.5.7 ATT 和 SECURITY 初始化

my_att_init 函数：该函数一般情况下定义在文件 app_att.c 中，不同的 demo 可能函数名不同，但是其主要功能是相同的，用于初始化 attribute_table 和 SECURITY。

示例：

```
void my_att_init ()
{
```

```
    blt_set_att_table ((u8 *)my_Attributes);

#if(RAM_SECURITY_ENABLE)

        blt_smp_store_in_ram_enable();
    blt_smp_func_init ();
#elif(E2PROM_SECURITY_ENABLE)
    blt_smp_eeprom_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_store_in_ram_enable(); 和 blt_smp_eeprom_enable(); 是根据存储方式选择smp的参数。如果用户使用E2PROM存储smp参数，初始化 blt_smp_eeprom_enable(); 否则初始化blt_smp_store_in_ram_enable(); SDK暂时不支持flash存储SMP参数。

blt_smp_func_init ()对slave的Pairing和Security进行初始化，该函数需要在 blt_smp_store_in_ram_enable(); 或者 blt_smp_eeprom_enable();之后进行。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。

另外，17H26 SDK 暂时不支持 pin code security 类型。

3.6 更新连接参数

3.6.1 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：

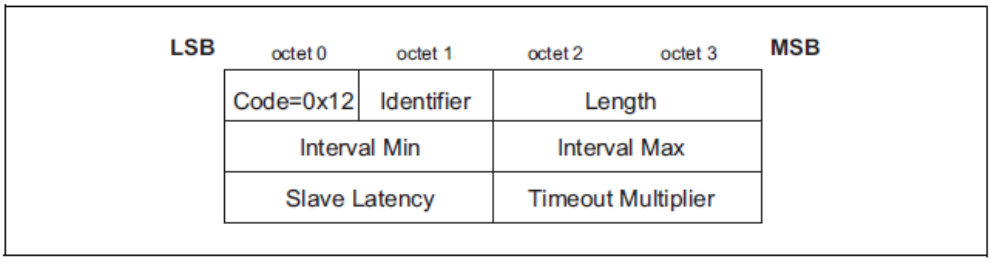

Figure 4.22: Connection Parameters Update Request Packet

图 3-8 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。

17H26 SDK 提供了一套接口用于更新连接参数，接口定义如下：

Function1:

```
void blt_update_conn_para (u16 min_interval, u16 max_interval, u16 latency,
u16 timeout);
```

Function2:

```
void blt_update_parameter_request ();
```

Function1 用于更新需要发送的关于更新连接包的参数，但是并不发送。其四个参数跟 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）。

Function2 用于发送 Function1 更新的数据包，当用户调用 Function2 时，系统会将该更新连接参数的数据发给 master 端。

对于需要使用更新连接参数的用户，需要**注意**以下几点：

1. 因为刚开始连接时，master 需要向 Slave（用户端）读取一些 BLE 协议栈信息，为了不影响其功能，建议用户在连接一段时间后（大概 5s 之后）再发送数据包

到 master 端。

2. 建议用户判断确认当前的连接 interval 不是自己想要的范围再更新参数，即发送连接参数包。

3. 建议用户更新参数时发送更新参数连接数据包的频率不可以太频繁，间隔最好在 5s 以上。

更新参数示例：

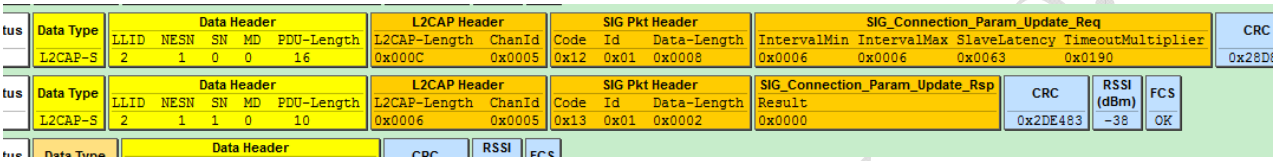

图 3-9 BLE 抓包 conn para update req 命令

3.6.2 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。图 3-9 所示的抓包显示 master 接受了申请。

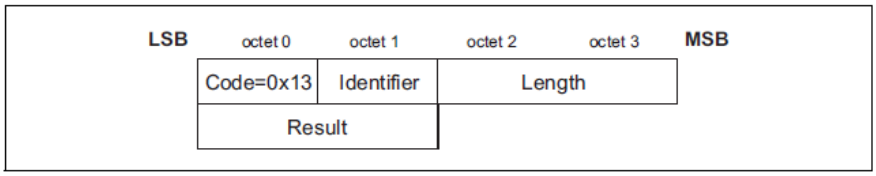

Figure 4.23: Connection Parameters Update Response Packet

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 3-10 BLE 协议栈中 conn para update rsp 格式

3.6.3 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 会发送 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。

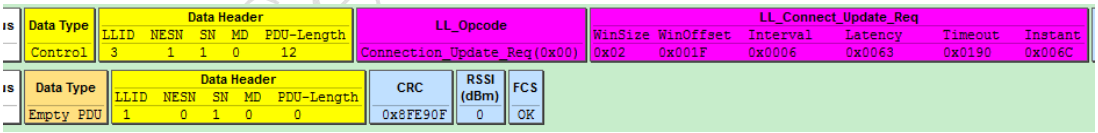

图 3-11 BLE 协议栈中 ll conn update req 格式

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。

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。

3.7 slave 主动断开连接

在连接状态下，slave 可以在 link layer 上主动发送 terminate，申请断开连接。SDK 提供的接口如下。

```
void blt_terminate (void);
```

只有在连接状态下（即 blt_state == BLT_LINK_STATE_CONN），调用上面的函数才有意义。

调用上面函数后，协议栈会在 ble TX fifo 没有需要发送的数据包时，向底层的 ble TX fifo 推送一个 terminate 命令的数据包；如果 ble TX fifo 有数据包时，系统会在 TX buffer 为空时，才会向 TX fifo 推送 terminate 数据包，用户不必要再调用 blt_terminate 函数。一般用户是在 main_loop 的 UI ENTRY 的地方推送该命令的，实际 SDK 最早需要在下一个收包发包点的时候才会将该数据发送出去，并且要再等一个 interval 才会知道 master 是否 ack 了这个包。所以我们的协议栈在发送 terminate 后，不会立刻退出连接状态，会等一会儿 master 的 ack 信号。

用户调用 blt_terminate 后，Master 会在下一个 Interval 回复 Ack 并马上断开连接，但是 Slave 端（SDK）并不会马上断开连接，会等到整个系统 timeout 之后才会认为当前连接断开。 Slave 端认为连接断开之后，会触发事件 BLT_EV_FLAG_TERMINATE，如果用户在 user_init 中使能系统回调函数变量的此项标志，那么在断开连接时，系

统会调用函数

```
void task_connection_terminated(u8* p)。
```

3.8 Attribute Protocol (ATT)

3.8.1 Attribute 基本内容

Attribute Protocol 定义了两种角色：server 和 client。17H26 BLE SDK 中，slave 主要定义的是 server 端.对应的 Android、iOS 设备是 client。Server 端与 client 端的关系是，server 的一组的 Attribute 可被 client 访问。

17H26 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。17H26 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。

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

3.8.2 Attribute Table

17H26 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;  ////节省空间可以去掉
  u8  attrLen;
//  u8  attrMaxLen;   ////节省空间可以去掉
  u16 uuid;
  u8* pAttrValue;

} attribute_t;
```

结合目前 17H26 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：

```
const attribute_t my_Attributes[] = {
    {14,0,0,0}, //

    //////////////////////////////////// 1. GATT Information  ////////////////////////////////////////////////
    // gatt information
    {5,2,GATT_UUID_PRIMARY_SERVICE,     (u8*)(&my_gapServiceUUID)},
    {0,1,GATT_UUID_CHARACTER,       (u8*)(&my_devNameCharacter)},
    {0,sizeof (my_devName), GATT_UUID_DEVICE_NAME,          (u8*)(my_devName)},
    {0,1,GATT_UUID_CHARACTER,       (u8*)(&my_appearanceCharacter)},
    {0,sizeof (my_appearance), 0x2a01,  (u8*)(&my_appearance)},

    //////////////////////////////////// 6. Battery Service  ////////////////////////////////////////////////
    {3,2,GATT_UUID_PRIMARY_SERVICE,     (u8*)(&my_batServiceUUID)},
    {0,1,GATT_UUID_CHARACTER,       (u8*)(&my_batProp)},                //prop
    {0,1,CHARACTERISTIC_UUID_BATTERY_LEVEL,     (u8*)(my_batVal)},  //value

    //////////////////////////////////// 9. Immediate Alert Service//////////////////////////////////////////////
    {3,2,GATT_UUID_PRIMARY_SERVICE, (u8*)(&immediateAlert_serviceUUID)},
    {0,1,GATT_UUID_CHARACTER,       (u8*)(&immediateAlertLevel_prop)},
    {0,1,CHARACTERISTIC_UUID_ALERT_LEVEL,   (u8*)(&immediateAlertLevel_value)},

    //////////////////////////////////// 12. Private Service ////////////////////////////////////////////////
    {3,2,GATT_UUID_PRIMARY_SERVICE, (u8*)(&privateSeviceUUID)},
    {0,1,GATT_UUID_CHARACTER,       (u8*)(&privatekeyNoti_prop)},
    {0,1,0xffe1,    (u8*)(&privateKeyNoti_value)},
};
```

图 3-12 17H26 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。

3.8.2.1 attNum

attNum 有两个作用。

a) 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：

{14,0,0,0},

attNum = 14 表示当前 Attribute Table 中共有 14 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，在 Attribute Table 中数 Attribute 在当前数组中的下标号，就能知道该 Attribute 当前的 Attribute Handle 值。如基于以上 table，如果用户需要 notify battery 相关的数据，那么其所使用的 handle 就是 14。

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，以上 attburite table 中 attNum 为 14，user 如果添加或删除了 Attribute，需要对此 attNum 进行修改。

b) attNum 第二个作用是用于表示当前 service 类型。

如上面截图所示，Attribute Handle 1- Attribute Handle 5 这 5 个 Attribute 是属于 gap service 的描述，它们属于一大类，所以这一组 Attribute 中首个 Attribute 的 UUID 为 0x2800(GATT_UUID_PRIMARY_SERVICE)，且 attribute value 为 0x1800(SERVICE_UUID_GENERIC_ACCESS)用于指明当前的 service 类型，那么它的 attNum 写为 5 后，BLE SDK 底层就知道从 Attribute Handle 1 开始的连续 7 个 Attribute 为 gap service。

同样，上图中的 battery service 的首个 Attribute 的 attNum 设为 3 后，底层会知道从这个 Attribute Handle 开始的连续 3 个 Attribute 都属于 battery service。

```
{3,2,GATT_UUID_PRIMARY_SERVICE,     (u8*)(&my_batServiceUUID)},
```

除了第 0 项 Attribute 和每一类 service 首个用于说明当前 service 类型的 Attribute 外，其他所有的 Attribute 的 attNum 的值都必须设为 0。

3.8.2.2 uuid、uuidLen

按照之前所述，UUID 分两种：BLE 标准的 2 bytes UUID 和 lenze 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;

{0,1,GATT_UUID_CHARACTER,    (u8*)(&my_devNameCharacter)},
```

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。

3.8.2.3 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，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。

3.8.2.4 write_command/write_request 回调函数

下表是 BLE Stack Spec 中对 write_command 和 write_request 的解释：

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 3-13 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 3-14 BLE 协议栈中 Write Command

BLESDK 在收到 write_request/write_command 时可以通过相应的指针去修改相应的变量，如果用户还需要在此时做一些别的操作（比如要给别的变量赋值），就可以通过回调函数 att_write_cb 去操作。

使能系统调用回调函数 att_write_cb 的设置如下：

1) 在 user_init 中，设置变量 l2cap_att_read_write_cb_flag 的值包含 ATT_WRITE_CB_ENABLE，即 (l2cap_att_read_write_cb_flag & ATT_WRITE_CB_ENABLE)!= 0，ATT_WRITE_CB_ENABLE 的值定义如下：

```
#define ATT_WRITE_CB_ENABLE    (1<<1)
```

2) 在回调函数 `int att_write_cb(void*p)`中，根据参数 p 获取 handle 值，根据 handle 值对 write_req 命令进行用户需要的处理。之后会对函数 att_write_cb 进行解析。

系统回调写回调函数的流程如下：

1) 系统收到 write_req 命令后，判断变量 l2cap_att_read_write_cb_flag 是否包含 ATT_WRITE_CB_ENABLE，即判断(l2cap_att_read_write_cb_flag & ATT_WRITE_CB_ENABLE)是否非 0，如果等于 0 ，则会继续进行步骤 3），否则执行步骤 2）。

2) 系统回调函数 att_write_cb，并且获取函数返回值 func_ret，如果 func_ret 等于 ATT_NO_HANDLED（ATT_NO_HANDLED = 0），则系统认为用户未对该 handle 的 write_req/write_cmd 进行处理，继续执行步骤 3）。

3) 系统通过该 handle 对应的 pAttrValue 的对应变量，修改其变量的值。

注意：对应 handle 的变量不可以是 const 类型。如果定义成 const 类型，编译器不会报错，但是在向此变量写值的时候，系统会出现异常。

函数 att_write_cb 的定义及解析：

函数：`int att_write_cb(void*p);`

描述: 该函数用于处理协议栈收到的 write_command 或者 write_request 命令。用户可以用来做自定义的操作。

参数： p – p 的类型实际上是 `rf_packet_att_write_t;` 用户可以在函数中对 p 的类型进行重定义。

3.8.2.5 read_request/read_blog_request 回调函数

一般情况下，系统在收到 read_request 或者 read_blog_request 时，会根据相应的 handle 值，将其对应的变量的值回复给 master。但是如果用户需要在 master 读取相应的变量时，做一些客户自定义的操作，那么用户就会用到 read_request 的回调函

数 att_read_cb。

设置回调函数 att_read_cb 有效的条件如下：

1) 在 user_init 中，设置变量 l2cap_att_read_write_cb_flag 的值包含 ATT_READ_CB_ENABLE，即(l2cap_att_read_write_cb_flag & ATT_READ_CB_ENABLE)!= 0，ATT_READ_CB_ENABLE 的值定义如下：

```
#define ATT_READ_CB_ENABLE    (1<<0)
```

2)定义函数 att_read_cb，其声明为 int att_read_cb(void *p);

系统对函数 att_read_cb 的调用流程如下：

1) 系统协议栈部分收到 read request 命令或者 read blog request 命令时，判断变量 l2cap_att_read_write_cb_flag 是否包含 ATT_READ_CB_ENABLE，即(l2cap_att_read_write_cb_flag & ATT_READ_CB_ENABLE)是否等于 0，如果等于 0，则什么都不做，否则执行步骤 2）。

2)系统执行函数 att_read_cb，并检测返回值 func_ret。

注意：该函数的返回值不能是 1，否则系统不会回复 read_response，继而导致 slave 与 master 断开连接。

回调函数 att_read_cb 解析：

函数：`int att_read_cb(void*p );`

描述: 在系统收到 readrequest 或者 read blog request 时，系统会触发此事件。

参数：p - p 的实际类型是 `rf_packet_att_readBlob_t` 或者 `rf_packet_att_read_t`，用户在实际使用时可重定义此参数类型。

3.8.3 Attribute PDU

17H26BLE 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。

BLE SDK 支持的以上几种类型 PDU 详细说明如下。

3.8.3.1 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。

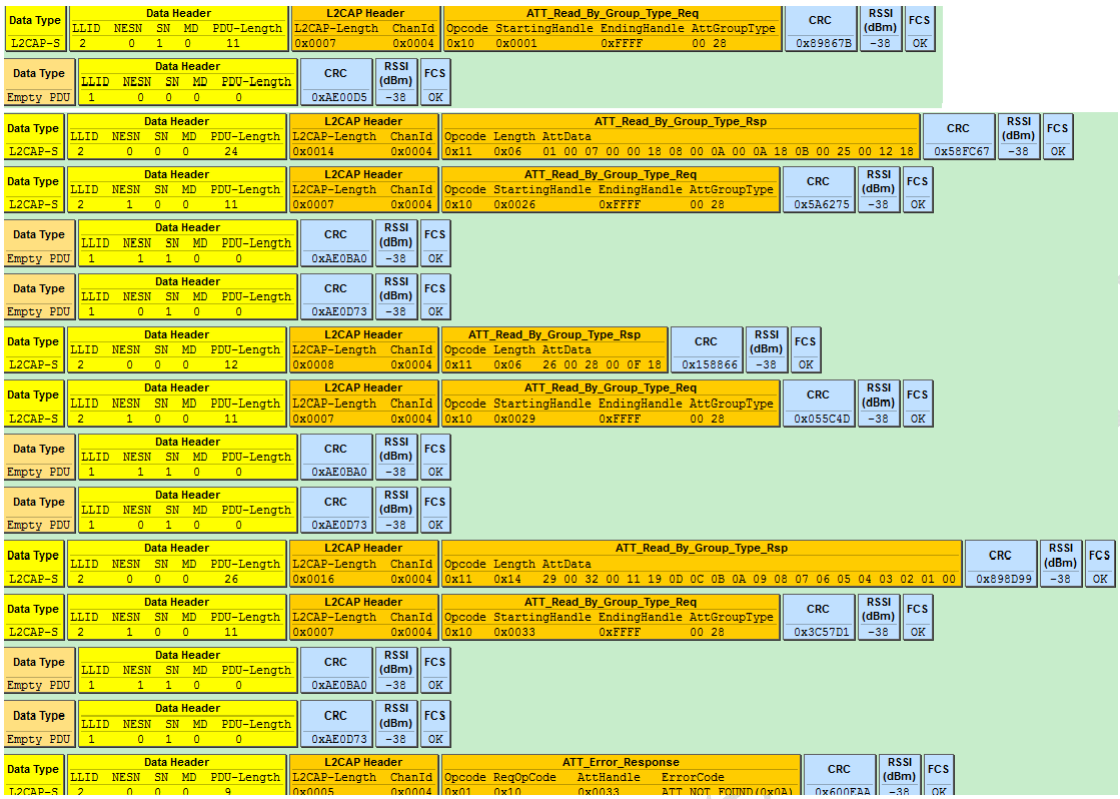

图 3-15 Read by Group Type Request/Read by Group Type Response

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 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）

slave 将以上 4 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 Response 的命令详细说明见《Core_v4.1_BLE_spec》P2143。

3.8.3.2 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 Response 的命令详细说明见《Core_v4.1_BLE_spec》P2134。

3.8.3.3 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。

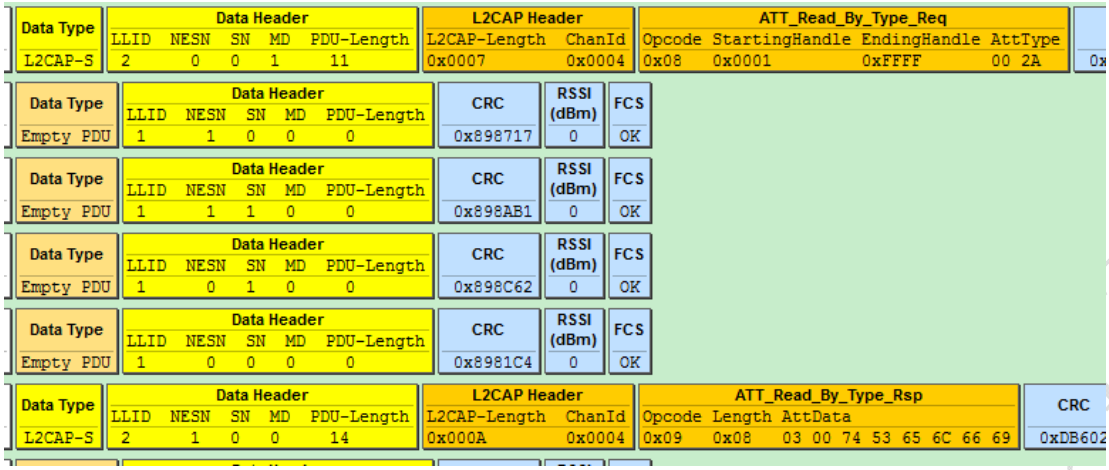

图 3-16 Read by Type Request/Read by Type Response

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,sizeof (my_devName), GATT_UUID_DEVICE_NAME, (u8*)(my_devName)},
```

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 Response 的命令详细说明见《Core_v4.1_BLE_spec》P2136。

3.8.3.4 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。

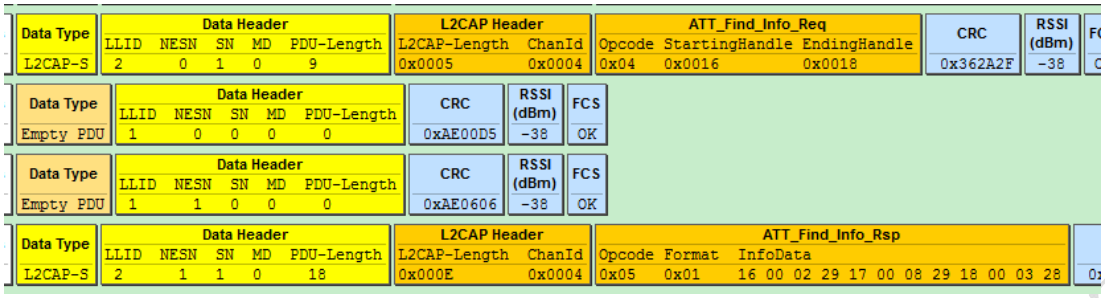

图 3-17 Find information request/Find information response

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

3.8.3.5 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（若设置了使能 att_read_cb，则回调该函数），如下图所示。

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

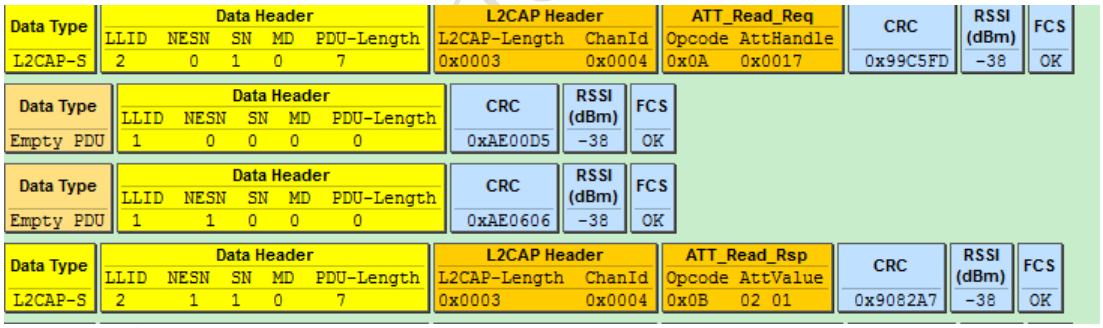

图 3-18 Read Request/Read Response

3.8.3.6 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（若设置使能回调函数 att_read_cb，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。

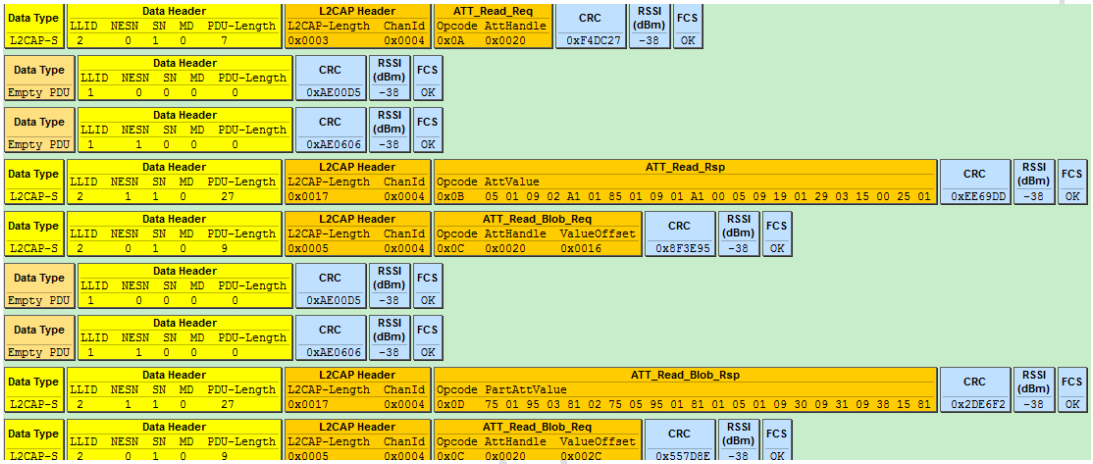

图 3-19 Read Blob Request/Read Blob Response

3.8.3.7 Exchange MTU Request、Exchange MTU Response

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。

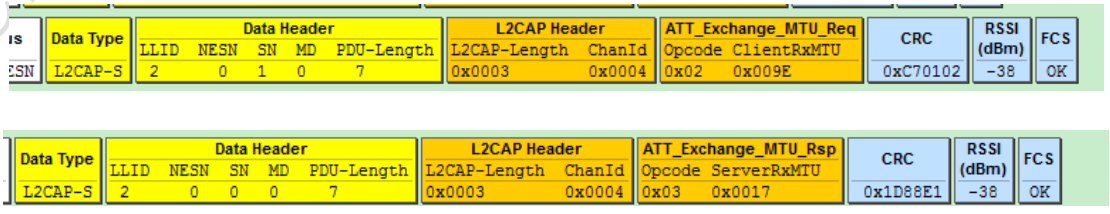

图 3-20 Exchange MTU Request/Exchange MTU Response

3.8.3.8 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 Attribute，根据 user 是否设置了回调函数 att_write_cb 决定数据是使用回调函数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。

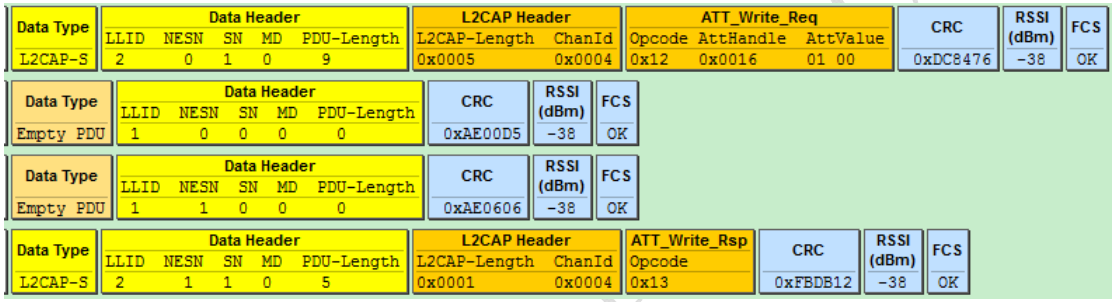

图 3-21 Write Request/Write Response

3.8.3.9 Write Command

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 att_write_cb 决定数据是使用回调函数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。

3.8.3.10 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 3-22 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。

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

1) 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
```

2) 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 MTU_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

17H26 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 10uA 左右。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。当 deepsleep 被唤醒时，MCU 将重新开始工作，跟重新上电是一样的效果，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纯粹的重新上电还是 deepsleep 醒来。在 deepsleep 下纯粹的 IC 电流为 0.5 uA 左右。

BLE SDK 实现低功耗的方法为：在 BLE 工作时序中，在程序不需要处理 BLE 时序以及任何 UI 时，系统都可以进入 suspend 状态，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 占的时间比例大，整个 main_loop 的平均电流就很低。

当我们不需要 17H26 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

17H26 MCU 在 suspend 模式可被 CORE 和 timer 唤醒，deepsleep 可被 PAD 和 timer 唤醒。17H26 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 BLE 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。17H26 所有 GPIO 的高低电平都可以通过 CORE 模块唤醒 suspend，但是只有部分 GPIO 的高低电平可以通过 PAD 模块唤醒 deepsleep。可以通过 Pad 唤醒的 GPIO 是 GP17~GP24、GP26、GP27、GP31 共 11 个 IO 口。

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 PAD 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_GP1,   1,   1);   //GPIO_GP1 CORE 唤醒打开，高电平唤醒
gpio_set_wakeup(GPIO_GP2,   1,   0);   //GPIO_GP1 CORE 唤醒关闭
gpio_set_wakeup(GPIO_GP5,   0,   1);   //GPIO_GP5 CORE 唤醒打开，低电平唤醒
gpio_set_wakeup(GPIO_GP5,   0,   0);   //GPIO_GP5 CORE 唤醒关闭
```

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：

```
cpu_set_gpio_wakeup (GPIO_GP17,   1,   1);   //GPIO_GP17 PAD 唤醒打开，高电平唤醒
cpu_set_gpio_wakeup (GPIO_GP17,   1,   0);   //GPIO_GP17 PAD 唤醒关闭
cpu_set_gpio_wakeup (GPIO_GP18,   0,   1);   //GPIO_GP18 PAD 唤醒打开，低电平唤醒
cpu_set_gpio_wakeup (GPIO_GP18,   0,   0);   //GPIO_GP18 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为 blt_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 BLE SDK，user 调用 blt_brx_sleep 函数时对 blt_sleep_wakeup 函数进行了封装，提供了更上层的接口函数 blt_brx_sleep，为了更好的理解，先介绍 blt_sleep_wakeup 函数。

int blt_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 32M 的 system clock，system tick 从 0x00000000 - 0xffffffff 跑一轮需要的时间为 128 S，如果 wakeup_tick 的值没有设好，最坏的情况需要等到 128 S 才会被唤醒。

举例说明 blt_sleep_wakeup 的用法：

1) blt_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) blt_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) blt_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 之前触发了 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) blt_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) blt_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，唤醒时间为执行该函数的 8 s 后。

6) blt_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 唤醒时间为执行该函数的 10 s 后。如果在 10 S 之前触发了 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 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 blt_sleep_wakeup 函数。使用下面接口 BLE SDK 进行低功耗的配置。

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

函数实现为：

```
u8 blt_enable_suspend (u8 en)
{
    u8 r = blt_suspend_mask;
```

```
    blt_suspend_mask = en;
    return r;
}
```

在 blt_brx_sleep 函数里面会对 blt_suspend_mask 进行检查以决定如何调用 blt_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 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 初始化中设定，否则一旦设定，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 初始化中设定，否则一旦设定，slave 响应 master 的 conn_req 后的第一个 main_loop 就会进入 deepsleep。

在 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_sleep 函数里面都自动配置唤醒源为 PM_WAKEUP_TIMER，且只有这一个唤醒源，唤醒的时间点 wakeup_tick 的值在广播状态由 T_advEvent 计算得到（user 通过 blt_set_adv_interval 函数设定），在连接状态则由 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过。如目前 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），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_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。

对于三个唤醒源 PM_WAKEUP_PAD、PM_WAKEUP_CORE、PM_WAKEUP_TIMER 的设置，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 main_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);
```

函数实现：

```
int             blt_wakeup_src = 0;
void   blt_set_wakeup_source (int src)
{
    blt_wakeup_src = 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：

```
blt_wakeup_src| PM_WAKEUP_TIMER
```

4.5 Suspend entry 基本工作机制

Suspend entry 的函数为 blt_brx_sleep()，该函数内部伪代码写法如下所示。首先

结合前面的 BLE entry 部分，定义两个时间参数：当前状态为广播态时，BLE entry 发第一个广播包的时间称为 T_advertising；当前状态为连接态时，BLE entry 收发数据时，收到第一个 master 的包的时间称为 T_brx。

```
void   blt_brx_sleep ()
{
    if(blt_suspend_mask &SUSPEND_DISABLE)  //当前 loop 不进低功耗模式
    {
           退出blt_brx_sleep
    }
    else if(blt_suspend_mask &(DEEPSLEEP_CONN | DEEPSLEEP_ADV))
    {      //当前 loop 进 deepsleep
        blt_sleep_wakeup (1, blt_wakeup_src , 0); //suspend
    }
    else if(   blt_state == BLT_LINK_STATE_ADV&&
               blt_suspend_mask &SUSPEND_ADV) //当前广播状态，进 suspend
    {
        blt_sleep_wakeup (0, PM_WAKEUP_TIMER,
T_advertising + advInterval); //suspend
    }
    else if(   blt_state == BLT_LINK_STATE_CONN&&
               blt_suspend_mask &SUSPEND_CONN) //当前连接状态，进 suspend
    {
        u32 wakeup_tick;
        if(conn_latency 非 0) //conn_latency != 0
        {
            u16 latency_use;

            //根据系统情况计算得到实际要使用的 latency_use。

            wakeup_tick = T_brx + (latency_use+1) * conn_interval;
```

```
        }
        else  //conn_latency == 0
        {
            wakeup_tick = T_brx + conn_interval;
        }



        blt_sleep_wakeup (0, PM_WAKEUP_TIMER| blt_wakeup_src,
                wakeup_tick);


        if(当前 suspend 是被 GPIO CORE 提前唤醒)
        {
            触发回调 BLT_EV_FLAG_EARLY_WAKEUP
            调整 BLE 时序相关处理
        }
    }

    //清除跟 user 相关的一些低功耗配置，下面三个变量须结合后面的介绍一起看
    blt_wakeup_src = 0;//清除当前loop  user设置的唤醒源
    blt_user_set_no_latency = 0;//清除当前loop  user手动关掉latency
    latency_set_by_user = 0;//清除当前 loop  user 手动设置的 latency


}
```

根据以上 Suspend entry 的逻辑实现，可以归纳出以下几点：

1) user 配置低功耗 disable 时，广播和连接状态都将 mainloop 剩余的时间空跑掉，在下一个发广播点/收 master 包点前退出，以便在 BLE entry 部分正好进行发包/收包处理。

2) user 配置当前 loop 进入 deepsleep 时，会进入 deepsleep，唤醒源上不带 PM_WAKEUP_TIMER，只接收 user 自己设定的唤醒源。由于 wakeup_tick 的值为

0，不支持 user 上层设定定时唤醒。user 如果想要实现进入 deepsleep 后定时唤醒，只能在 UI entry 部分自己调用函数 blt_sleep_wakeup (...)来实现。

3) 广播状态下的 suspend，唤醒源只有 PM_WAKEUP_TIMER，唤醒时间自动计算为下一个广播事件的发广播包时间点。

4) 连接状态下 suspend：

A. 当前系统的 conn_latency 为 0 时，跟广播态处理类似，唤醒时间自动计算为下一个 conn_interval 接收 master 包的时间点；

B. 当 conn_latency 非 0 时，并不是直接使用 conn_latency 来计算唤醒时间点，最终使用 latency_use，suspend 时间为(latency_use+1) * conn_interval。latency_use 计算方法跟当前系统的 conn_latency 有关，也跟 user 在上层设置的一些低功耗优化有关。

4.6 GPIO 唤醒的注意事项

由于 17H26 的 CORE/PAD 唤醒是靠高低电平唤醒的，而不是上升沿下降沿唤醒，所以当配置了 GPIO CORE 或者 PAD 唤醒时，比如设置了某个 GPIO CORE 高电平唤醒 suspend，要确保 MCU 在调用 blt_wakeup_sleep 进入 suspend 时，当前的这个 GPIO 的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 blt_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。同样 GPIO PAD 唤醒也是如此。

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。

针对以上问题，建议用户使用时，如果需要处理此种情况，建议用户在检测到按键 Release 再进入 Deepsleep 模式。并且在进入 suspend 模式时，检测到按键，将 gpio 唤醒取消，否则 MCU 会一直处于全速运行状态，功耗较高。

user 在使用 lenze 的 GPIO CORE/PAD 唤醒时，要注意避免这个问题。