FR801xH SDK User Guide

Bluetooth Low Energy SOC with SIG Mesh

2020-03 v1.0.5

www.freqchip.com

FREQCHIP 富芮坤

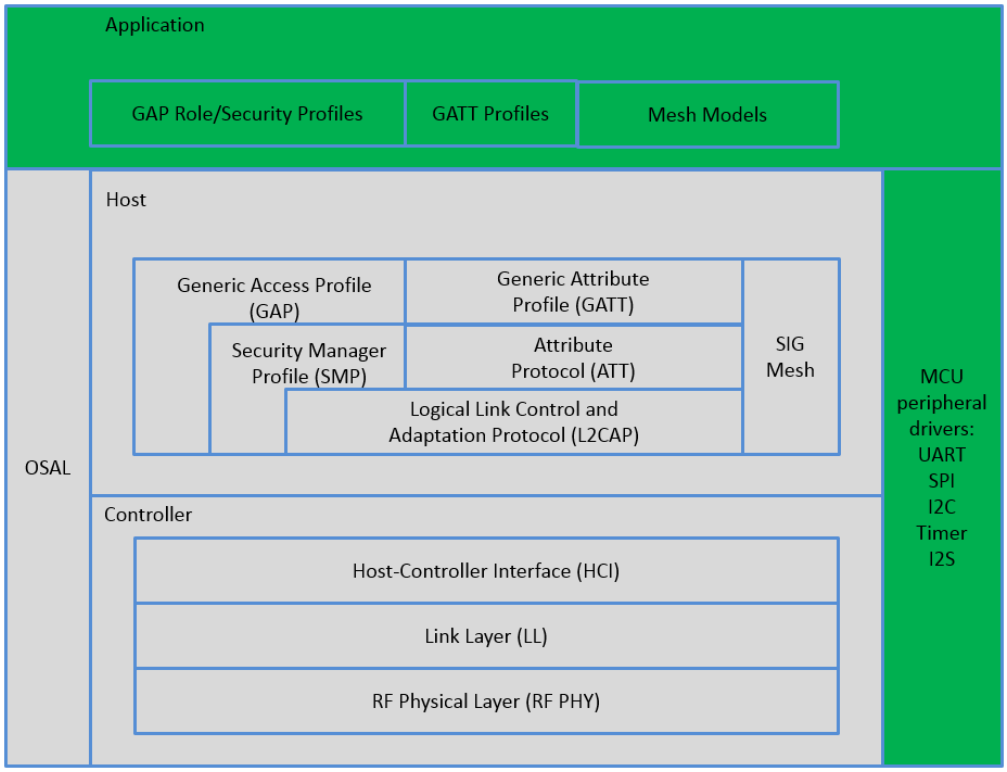
目录

1. 概况

本文档是 FR801xH SDK 的应用开发指导。FR801xH 是单芯片 BLE SOC。FR801xH SDK 是运行于 FR801xH 上的软件包，包含了 BLE 5.0 的完整协议栈，芯片的外设驱动以及操作系统抽象层 OSAL。

1.1 FR801xH SDK 结构

FR801xH SDK 的架构如下图所示。SDK 包含了完整的 BLE 5.0 协议栈，包括完整的 controller，host，profile，SIG Mesh 部分。其中蓝牙协议栈的 controller 和 host 部分以及操作系统抽象层 OSAL 都是以库的形式提供，图中为灰色部分。MCU 外设驱动和 profile，以及应用层的例程代码，都是以源码的形式提供，图中为绿色部分。

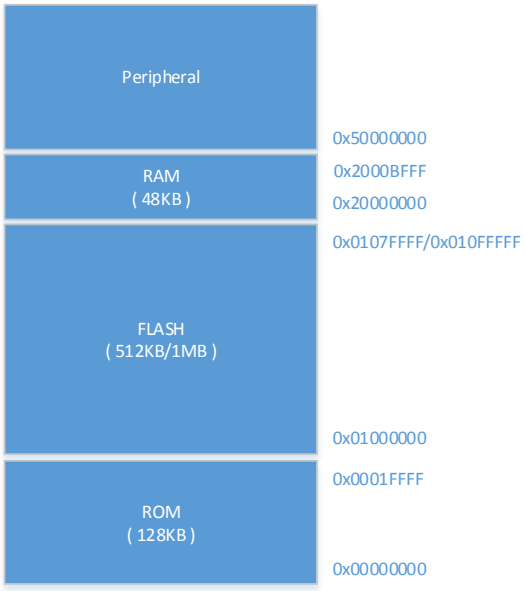


FR801xH SDK 框图

1.2 空间分配

1.2.1 地址空间

FR801xH 的地址空间如下：

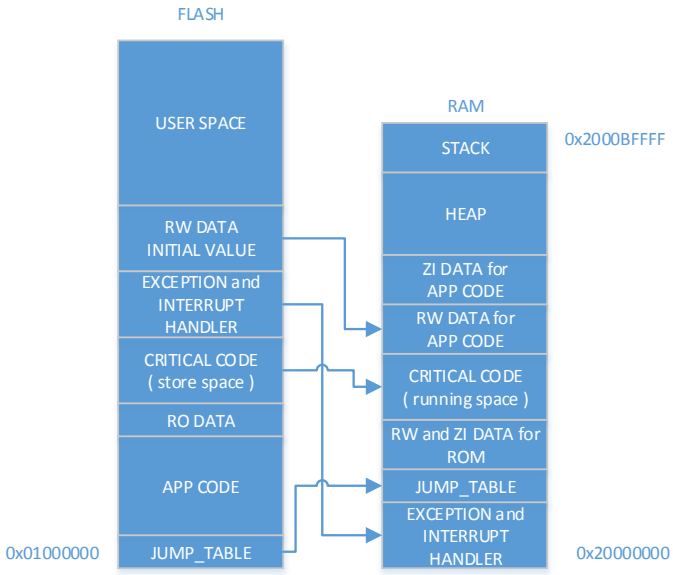


FR801xH 地址空间

其中内置 128KB ROM，主要内容为启动代码、BLE controller 部分协议栈；FLASH 空间用于存储用户程序、用户数据等；RAM 用于存储各种变量、堆栈、重新映射后的中断向量地址、对运行速度较为敏感的代码（中断响应等）等，该空间都支持低功耗的 retention 功能；外设地址空间是各种外设的地址映射，用于进行外设的配置。

1.2.2 空间分配

在 FR801xH 中 FLASH 空间和 RAM 空间的分配由链接脚本指定，具体分配如下：

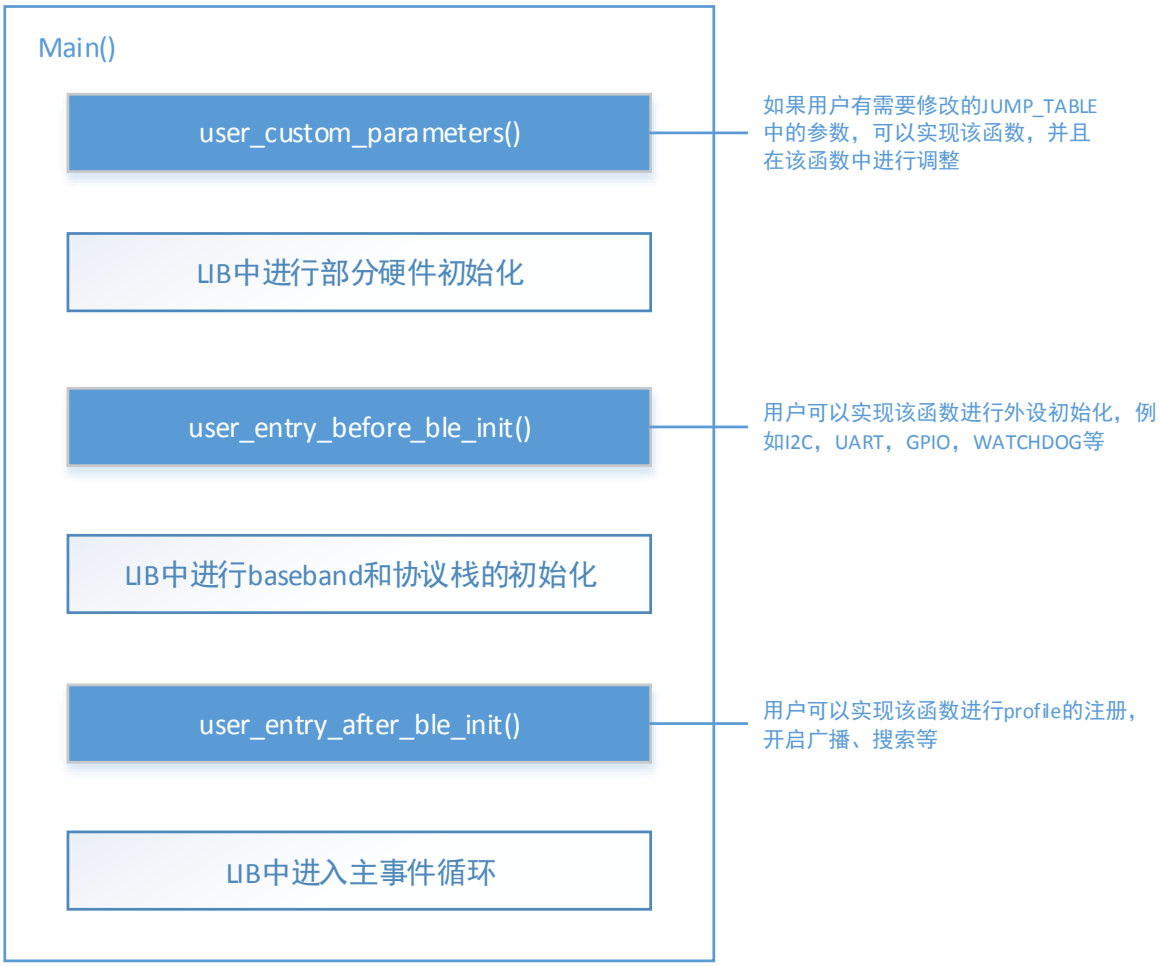


FR801xH flash 和 RAM 空间分配

其中 JUMP_TABLE 存储的是配置信息；APP CODE 和 RO DATA 可以通过 XIP 被 MCU 直接访问；CRITICAL CODE 和 EXCEPTION and INTERRUPT HANDLER 为对运行时间敏感的用户代码，需要在初始化时从 flash 中搬移到 RAM 中；RW DATA 需要进行初始化；ZI 为初始值为 0 的数据段。这些操作均由 SDK 内部进行处理，用户无需做额外操作。HEAP 为动态内存分配空间，SDK 中会根据实际可用空间对内存管理单元进行初始化；STACK 为堆栈空间，生长空间由高到低，大小可由用户指定。

1.3 代码流程

SDK 包含了四大部分，Application 部分，蓝牙协议栈部分，操作系统抽象层 OSAL 部分，还有 MCU 外设驱动部分。整个代码结构比较简单，执行流程也很清晰易懂。SDK 的 main 函数主体入口位于 lib 库中，对于应用层以源码形式开放了一些入口，用于应用开发初始化，基本流程如下图所示：



main 函数入口

1.3.1 user_custom_parameters 函数

该函数示例：

```
void user_custom_parameters(void)
{
    __jump_table.addr.addr[0] = 0x01;
    __jump_table.addr.addr[1] = 0x01;
    __jump_table.addr.addr[2] = 0x01;
    __jump_table.addr.addr[3] = 0x01;
    __jump_table.addr.addr[4] = 0x01;
    __jump_table.addr.addr[5] = 0xc1;
```

```
    __jump_table.image_size = 0x19000;    // 100KB
    __jump_table.firmware_version = 0x00010000;
    __jump_table.system_clk = SYSTEM_SYS_CLK_48M;

    jump_table_set_static_keys_store_offset(0x30000);
}
```

该函数实现了设置本机蓝牙地址、设置程序文件上限、版本信息、配置系统时钟为 48M；配置协议栈中使用的 key（包括 IRK，椭圆曲线加密算法中的 public key 和 privatekey）在 flash 中的保存地址为 0x30000（默认地址即为 0x30000）。用户可以根据实际需求进行相应的配置。

1.3.2 user_entry_before_ble_init 函数

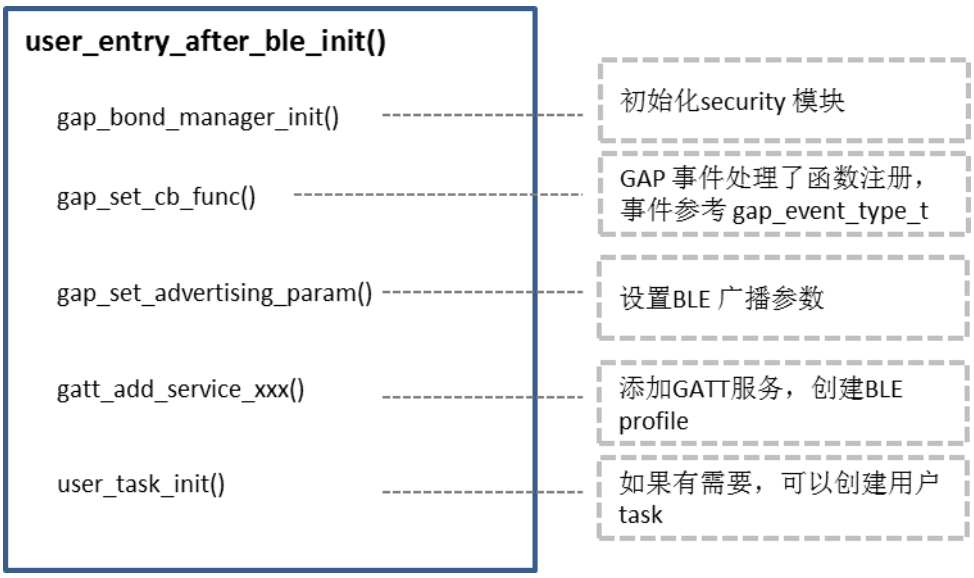
该函数示例：

```
void user_entry_before_ble_init(void)
{
  /* set system power supply in BUCK mode */
  pmu_set_sys_power_mode(PMU_SYS_POW_BUCK);

  pmu_enable_irq(PMU_ISR_BIT_ACOK
                 | PMU_ISR_BIT_ACOFF
                 | PMU_ISR_BIT_ONKEY_PO
                 | PMU_ISR_BIT_OTP
                 | PMU_ISR_BIT_LVD
                 | PMU_ISR_BIT_BAT
                 | PMU_ISR_BIT_ONKEY_HIGH);
                 NVIC_EnableIRQ(PMU_IRQn);

  /* AT command interface */
  app_at_init();
  ......
```

该函数实现了配置芯片供电模式为 BUCK，使能了一系列 PMU 部分的中断，配置了 AT 指令所使用的的串口等

1.3.3 user_entry_after_ble_init 函数

user_entry_after_ble_init 为 BLE stack 在系统中初始化完成并成功创建 BLE stack task 后，用户进行自定义行为的入口，比如可以进行协议栈相关的一些操作，比如下图所示，可以进行 bond manager 的初始化，GAP 事件处理回调函数的注册，BLE 广播参数的设置，GATT service 的创建，用户 task 的创建等。



user_entry_after_ble_init() 函数

1.3.4 睡眠唤醒用户接口

在系统使能睡眠后，LIB 中主代码会判断是否满足进入睡眠条件，针对开始睡眠前和唤醒后分别提供了入口供用户进行自定义系统行为。

1. user_entry_before_sleep_imp

该函数在进入睡眠前被调用，用户可在里面实现控制 GPIO 的状态保持（针对 GPIO 在系统工作和睡眠状态下的控制参见外设驱动章节）等行为。

2. user_entry_after_sleep_imp

在系统唤醒后，用户可以在该函数中重新进行外设的初始化（进入睡眠后外设的状态因为掉电都会丢失）等操作。

1.4 __jump_table

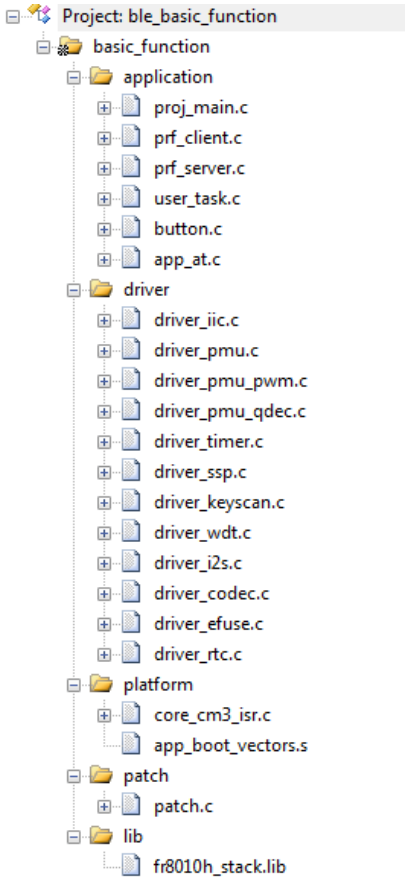
__jump_table 结构体中保存了一些配置信息：

名称	值	功能
image_size	-	用户代码大小，以字节为单位。芯片内置代码在启动时会根据该大小去查找 OTA 的备份区域。例如该值为 0x10000，那么 OTA 双区域在 flash 中的基地址就分别为 0 和 0x10000。因此该值在一个项目中应该保持一致，取值为该项目预计可能占用最大的 flash 空间，且按照 4K 对齐。
firmware_version	-	用户程序的版本号，芯片内置代码在启动时根据版本号来判断 OTA 双区域中的代码哪一个为最新的版本。
addr	-	本机的蓝牙地址
system_clk	SYSTEM_SYS_CLK_12M SYSTEM_SYS_CLK_24M	系统工作主频。

	SYSTEM_SYS_CLK_48M	

1.5 SDK 项目工程

SDK 以源码形式提供了多个项目工程作为参考，用户可以在这些工程上进行自己的应用开发。这些工程采用了同样的目录结构，如下图所示：



工程目录结构

其中 application 用于存放用户应用层的代码，自定义的 profile 等；driver 中为外设驱动；platform 中为异常向量入口和部分异常的处理函数；patch 中为针对 ROM code 中的一些补丁代码；lib 中为封装好的库文件，其中所提供的接口在 gatt_api.h、gap_api.h 等文件中。

在当前的 SDK 中提供了一下几种 sample 工程：

- HID 例程：ble_hid_kbd_mice
- Mesh 例程：ble_mesh
- 主机例程：ble_simple_central
- 从机例程：ble_simple_peripheral

1.6 芯片烧录

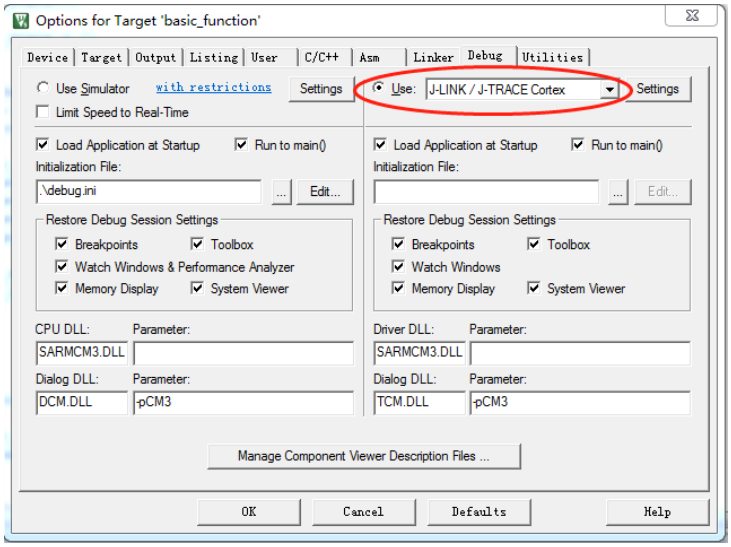
芯片烧录方式主要有两种：

- 基于 Keil + J-Link 的烧录方式
- 采用 PC 烧录工具+串口进行烧录。

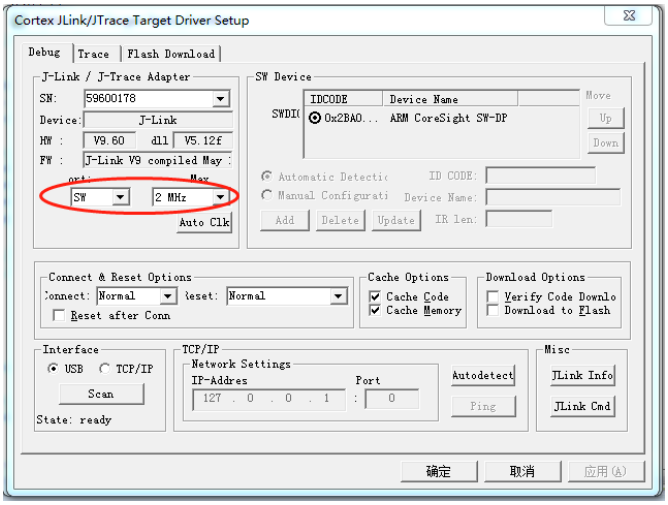
1.6.1 基于 Keil + J-Link 的烧录方式

用户将文件 FR8010H.FLM 存放在 Keil 安装目录下的 ARM\Flash 路径中，然后在 Keil 工程中进行如下配置

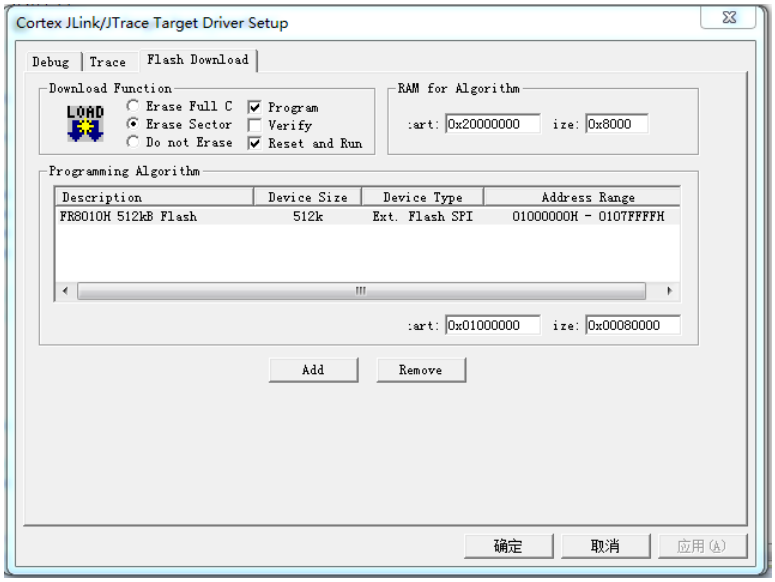
1. 选用 J-Link 作为调试工具



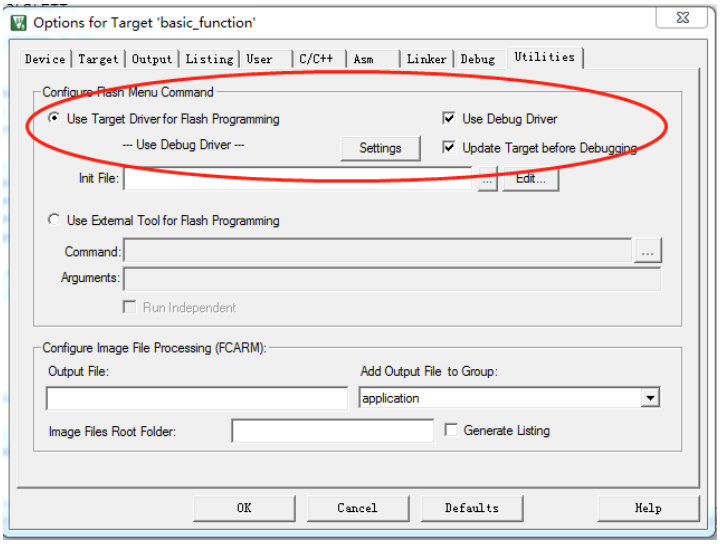
2. 配置调试方式为 SW



3. 在 flash download 选项卡中配置下载选项



4. 配置使用 Debug Driver 进行 flash 的烧录



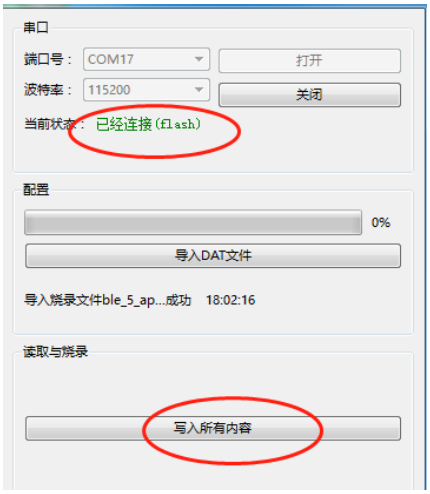
通过以上配置就可以实现在 Keil 的 IDE 中进行 flash 的调试和烧录。

1.6.2 采用 PC 烧录工具+串口进行烧录

在芯片的 J-Link 调试口被复用成别的功能，或者系统会进入睡眠时，则无法通过上述方法进行烧录。这时可以采用 PC 烧录工具加串口的方式进行烧录，这一方法原理是：在芯片一上电时，内部 boot 程序会尝试通过串口与外部工具进行通信，在握手成功之后就可以进行烧录等后续操作。具体操作如下：

1. 打开 PC 端串口烧录工具，选择正确的串口号，导入 DAT 文件（选择要烧录的 bin 文件），然后打开串口，进入等待连接状态。

2. 将串口工具的 TX 连接到芯片 PA2（芯片端的 RX），RX 连接到芯片的 PA3（芯片端的 TX）。
3. 将串口工具的地和供电同时与芯片的地和供电连接，这时芯片与 PC 工具握手成功后在工具端会显示已经连接，然后点击写入所有内容即可将程序烧录到芯片中



注意事项：因为串口工具的 TX 会串电到芯片端，所以接线连接顺序要符合上面的 2 和 3 步骤所描述的流程。

1.6.3 量产烧录

FR801xH 系列芯片有完善的量产烧录工具，可以支持烧录裸片、也可烧录 PCBA，具体实施方式可以联系代理商。

2. 低功耗管理

2.1 工作模式

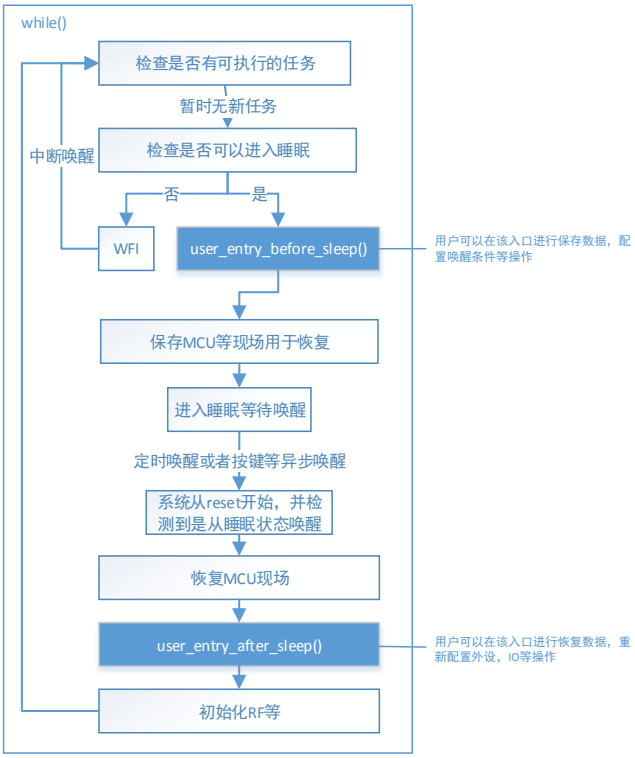
FR801xH 系列芯片在 MCU 正常工作模式，3.3V 供电情况下，工作电流在 2~4mA。为了节省电能，可以进入低功耗模式，FR801xH 支持两种低功耗模式：

工作模式	深睡眠	深睡眠（RAM retention）
RAM	不保持	100% retention
Cache	不保持	可选
数字逻辑（包括外设等资源）	不保持	不保持
PMU（KEYSCAN，QDEC，RTC 等可选）	保持	保持

- 在深睡眠（RAM retention）模式下，系统的底电流大致为 6~7uA，可以被 GPIO，Keyscan，QDEC，RTC 等模块唤醒，且 RAM 中数据保持，这种模式适用于正常连接状态、或者周期性广播状态等。在这种模式下，唤醒时系统会进行现场恢复的操作，包括 MCU 状态，RF 初始化等操作，整个流程会在很短的时间完成。
- 深睡眠模式下系统底电流大致为 3~4uA，唤醒条件与深睡眠（RAM retention）模式下一致，这种模式适用于关机状态。在这种模式下，系统工作流程与正常启动一致。

2.2 程序运行流程

主程序的运行流程如下图：



在该流程中用户在睡眠前和唤醒后各有一个入口可以进行自定义的操作：

1. user_entry_before_sleep

该函数在进入睡眠前被调用，用户可在里面实现控制 GPIO 的状态保持（针对 GPIO 在系统工作和睡眠状态下的控制参见外设驱动章节），配置睡眠唤醒条件等行为。

2. user_entry_after_sleep

在系统唤醒后，用户可以在该函数中重新进行外设的初始化（进入睡眠后外设的状态因为掉电都会丢失）等操作。

2.3 唤醒条件

睡眠的唤醒有同步和异步两种：同步唤醒来自一个硬件 timer，这个 timer 的设置由协议栈中代码完成，主要取决于广播间隔、连接间隔等参数，在应用层代码中无需关注；异步主要来自于 PMU（电源管理单元）的中断信号，PMU 的中断源有：充电器插入拔出、KEYSCAN 模块、Q-DEC 模块、RTC、GPIO 状态监测模块等，这些中断源可以在系统初始化时进行设置。例如：

```
pmu_set_pin_pull(GPIO_PORT_D, (1<<GPIO_BIT_4)|(1<<GPIO_BIT_5), true);
pmu_port_wakeup_func_set(GPIO_PD4|GPIO_PD5);
```

这两行代码可以配置 PMU 中的 GPIO 状态监测模块开始监测 GPIO_PD4 和 GPIO_PD5 的状态，一旦发生电平高低的变化，就可以产生 PMU 中断。如果在睡眠中产生 PMU 中断，则系统会被唤醒，唤醒后可在 PMU 的中断处理函数中进行相应的处理。

3. BLE 协议栈

SDK 里面包含了完整的协议栈，虽然 controller 和 host 部分是以库的形式提供，但给出了接口丰富的 API 提供给上层应用开发调用。Profile 则是以源码的形式提供。

3.1 GAP API

GAP 层的 API 位于 components\ble\include\gap\gap_api.h 文件。

3.1.1 GAP 事件

3.1.1.1 GAP Event Type Defines

// GAP_EVT_TYPE_DEFINES

```
typedef enum
{
    GAP_EVT_ALL_SVC_ADDED                   //!< 所有的 service 都添加完毕。
    GAP_EVT_SLAVE_CONNECT                   //!< 做为 slave 链接建立。
    GAP_EVT_MASTER_CONNECT                  //!< 做为 master 链接建立。
    GAP_EVT_DISCONNECT                      //!<  链接断开，可能是 master 或 slave。
    GAP_EVT_LINK_PARAM_REJECT               //!<  链接参数更新被拒绝。
    GAP_EVT_LINK_PARAM_UPDATE               //!<  链接参数更新成功。
    GAP_EVT_ADV_END                         //!<  广播结束。
    GAP_EVT_SCAN_END                        //!<  扫描结束。
    GAP_EVT_ADV_REPORT                      //!<  找到一个 BLE 设备。
    GAP_EVT_CONN_END                        //!<  主动连接的动作未完成，被主动停止。
    GAP_EVT_PEER_FEATURE                    //!<  收到对端的 feature 特性回复。
    GAP_EVT_MTU                             //!<  mtu 交换完成。
    GAP_EVT_LINK_RSSI                       //!<  收到链接对端的 rssi 值。
    GAP_SEC_EVT_MASTER_AUTH_REQ             //!<  做为 master，收到对端 slave 的建立安全链接的请求。
    GAP_SEC_EVT_MASTER_ENCRYPT              //!<  做为 master，链接加密完成。
    GAP_SEC_EVT_SLAVE_ENCRYPT               //!<  做为 slave，链接加密完成。
}gap_event_type_t;
```

3.1.1.2 GAP Link Established Event

// Link established event structure

```
typedef struct
{
```

```
    uint8_t                 conidx          //!<  链接号。
    mac_addr_t              peer_addr       //!<  链接对端的 mac 地址。
    uint8_t                 addr_type       //!<  链接对端 mac 地址的类型。
    uint16_t                con_interval    //!<  链接握手间隔参数。单位:1.25ms
    uint16_t                con_latency     //!<  链接  lantency 参数。
    uint16_t                sup_to          //!<  链接超时断开的参数。单位:10ms
} conn_peer_param_t;
```

3.1.1.3 GAP Link Disconnect Event

// Link disconnect event structure

```
typedef struct
{
    uint8_t     conidx      //!< 断开链接的链接号
    uint8_t     reason      //!<   断开链接的原因。参考 hl code
} gap_evt_disconnect_t;
```

3.1.1.4 GAP Link Parameter Update Reject Event

// Link parameter update reject event structure

```
typedef struct
{
    uint8_t     conidx      //!< 链接参数更新被拒绝事件对应的链接号
    uint8_t     status      //!<   链接参数更新被拒绝的原因
} gap_evt_link_param_reject_t;
```

3.1.1.5 GAP Link Parameter Update Success Event

// Link parameter update success event structure

```
typedef struct
{
    uint8_t     conidx          //!<  链接参数更新成功事件对应的链接号
    uint16_t    con_interval    //!<  链接参数更新后的握手间隔参数。单位：1.25ms
    uint16_t    con_latency     //!<  链接参数更新后的 lantency 参数。
    uint16_t    sup_to          //!<  链接参数更新后的超时断开参数。单位：10ms
} gap_evt_link_param_update_t;
```

3.1.1.6 GAP Advertising Report Event

// Adv report indication structure

```
typedef struct
{
```

```
    uint8_t          evt_type    //!<   收到广播的类型，见@ GAP_SCAN_EVT_TYPE_DEFINES
    mac_addr_t       src_addr    //!<   广播的 mac 地址。
    int8_t           tx_pwr      //!<   广播的发射功率。
    int8_t           rssi        //!< 广播的 rssi 值。
    uint16_t         length      //!<   广播的数据长度
    uint8_t *        data        //!<   指向广播的数据 buffer 的指针。
} gap_evt_adv_report_t;
```

3.1.1.7 GAP Peer Feature Event

```
// Peer feature rsp structure
typedef struct
{
    uint8_t       conidx              //!< 事件对应的链接号。
    uint8_t       features[8]         //!<   对端 feature 的值。
} gap_evt_peer_feature_t;
```

3.1.1.8 GAP GATT MTU Event

```
// Mtu exchanged event structure
typedef struct
{
    uint8_t       conidx      //!< 事件对应的链接号。
    uint16_t      value       //!<   mtu 交换之后的最终值。
} gattc_mtu_t;
```

3.1.1.9 GAP Security Master Authentication Request Event

```
// Got auth_req event structure
typedef struct
{
    uint8_t       conidx     //!< 事件对应的链接号。
    uint16_t      auth       //!<   是否要进行加密。 0: 不加密。 1: 加密
} gap_sec_evt_master_auth_req_t;
```

3.1.1.10 GAP Message Event

```
// GAP message event structure
typedef struct
{
    gap_event_type_t                type                        //!< GATT event 的类型，见@GAP_EVT_TYPE_DEFINES
    union{
```

conn_peer_param_t	slave_connect	做为 slave 链接建立事件对应参数。
conn_peer_param_t	master_connect	做为 master 链接建立事件对应参数。
gap_evt_disconnect_t	disconnect	链接断开事件对应参数。
gap_evt_link_param_reject_t	link_reject	链接参数更新被拒绝事件对应参数。
gap_evt_link_param_update_t	link_update	链接参数更新成功事件对应参数。
uint8_t	adv_end_status	广播结束事件的原因。
uint8_t	scan_end_status	扫描结束事件的原因。
gap_evt_adv_report_t *	adv_rpt	接收到的扫描广播包。
uint8_t	conn_end_reason	主动连接动作被停止事件对应的原因。
gap_evt_peer_feature_t	peer_feature	收到对端 feature 回复时对应的参数。
gattc_mtu_t	mtu	mtu 交换完毕 事件 对应的参数。
int8_t	link_rssi	收到链接对端 rssi 时的值。
gap_sec_evt_master_auth_req_t	auth_req	收到对端建立安全链接请求时对应的参数。
uint8_t	master_encrypt_conidx	master 加密事件对应的链接号
uint8_t	slave_encrypt_conidx	slave 加密事件对应的链接号

```
  }param;
}gap_event_t;
```

3.1.1.11 GAP Advertising Mode Defines

// GAP_ADV_MODE_DEFINES，对应 gap_adv_param_t 的 adv_mode 变量值。		
GAP_ADV_MODE_UNDIRECT	0x01	//!< 广播为非指向性，可连接，可扫描的。
GAP_ADV_MODE_DIRECT	0x02	//!< 广播为指向性，可连接，不可扫描的。
GAP_ADV_MODE_NON_CONN_NON_SCAN	0x03	//!< 广播为非指向性，不可连接，不可扫描的。
GAP_ADV_MODE_NON_CONN_SCAN	0x04	//!< 广播为非指向性，不可连接，可扫描的。

3.1.1.12 GAP Advertising Type Defines

// GAP_ADDR_TYPE_DEFINES		
GAP_ADDR_TYPE_PUBLIC	0x00	//!< 地址类型为 public。
GAP_ADDR_TYPE_PRIVATE	0x01	//!< 地址类型为 private。
GAP_ADDR_TYPE_RANDOM_RESOVABLE	0x02	//!< 地址类型为 resolvable 随机地址。
GAP_ADDR_TYPE_RANDOM_NONE_RESOVABLE	0x03	//!< 地址类型为非 resolvable 随机地址。

3.1.1.13 GAP Advertising Channel Defines

// GAP_ADVCHAN_DEFINES		
GAP_ADV_CHAN_37	0x01	//!< 广播选择 37 通道
GAP_ADV_CHAN_38	0x02	//!< 广播选择 38 通道
GAP_ADV_CHAN_39	0x04	//!< 广播选择 39 通道
GAP_ADV_CHAN_ALL	0x07	//!< 广播选择所有的 37~39 通道

3.1.1.14 GAP Advertising Filter Mode Defines

// GAP_ADV_FILTER_MODE_DEFINES		
GAP_ADV_ALLOW_SCAN_ANY_CON_ANY	0x00	//!< 广播允许任何设备扫描和链接
GAP_ADV_ALLOW_SCAN_WLST_CON_ANY	0x01	//!< 广播允许任何设备链接，只允许白名单里设备扫描
GAP_ADV_ALLOW_SCAN_ANY_CON_WLST	0x02	//!< 广播允许任何设备扫描，只允许白名单里设备链接
GAP_ADV_ALLOW_SCAN_WLST_CON_WLST	0x03	//!< 广播只允许白名单里设备扫描和链接

3.1.1.15 GAP Advertising Paramters

```
// Gap adv parameter structure
typedef struct
{
  uint8_t       adv_mode        //!<  广播的模式，见@ GAP_ADV_MODE_DEFINES
  uint8_t       adv_addr_type   //!<  广播的 local mac 地址类型。见@ GAP_ADDR_TYPE_DEFINES
  mac_addr_t    peer_mac_addr   //!<  指向性广播模式时对端 mac 地址。
  uint8_t       phy_mode        //!<  广播的 phy 选择。 保留，不用设置。默认 1M
  uint16_t      adv_intv_min    //!< 广播间隔最小值。单位：0.625ms。必须不小于 0x20
  uint16_t      adv_intv_max    //!< 广播间隔最大值。单位：0.625ms。必须不小于 0x20
  uint8_t       adv_chnl_map    //!<  广播的通道选择。见@ GAP_ADVCHAN_DEFINES
  uint8_t       adv_filt_policy //!<  广播的过滤设置。见@ GAP_ADV_FILTER_MODE_DEFINES
}gap_adv_param_t;
```

3.1.1.16 GAP Scan Mode Defines

// GAP_SCAN_MODE_DEFINES		
GAP_SCAN_MODE_GEN_DISC	0x00	//!< 常规主动扫描，能收到 scan rsp 包。
GAP_SCAN_MODE_OBSERVER	0x02	//!< 被动扫描，不能收到 scan rsp 包。

3.1.1.17 GAP Scan Result Type Defines

// GAP_SCAN_EVT_TYPE_DEFINES		
GAP_SCAN_EVT_CONN_UNDIR	0x00	//!< 收到的广播包为非指向性可连接，可扫描包
GAP_SCAN_EVT_CONN_DIR	0x01	//!< 收到的广播包为指向性广播包。
GAP_SCAN_EVT_NONCONN_UNDIR	0x02	//!< 收到的广播包为非指向性不可连接，可扫描包
GAP_SCAN_EVT_SCAN_RSP	0x04	//!< 收到的广播包为 scan rsp

3.1.1.18 GAP Scan Paramters

```
// Gap scan parameters
typedef struct
{
```

```
  uint8_t       scan_mode      //!<   扫描的模式，见@ GAP_SCAN_MODE_DEFINES
  uint8_t       dup_filt_pol   //!<   扫描接收到的包是否要过滤重复包. 1:过滤; 0:不过滤
  uint16_t      scan_intv      //!<   扫描间隔。必须大于等于 scan_window。范围 4~16384
  uint16_t      scan_window    //!<   扫描开窗间隔。范围 4~16384
  uint16_t      duration       //!< 扫描持续时间。单位:10ms。0 表示不会主动结束。
}gap_scan_param_t;
```

3.1.1.19 GAP Pairing Mode Defines

// GAP_PAIRING_MODE_DEFINES		
GAP_PAIRING_MODE_NO_PAIRING	0x00	//!< 不允许配对。
GAP_PAIRING_MODE_WAIT_FOR_REQ	0x01	//!< 需要等待对方的配对请求。

3.1.1.20 GAP IO Capabilities Defines

// GAP_IO_CAP_DEFINES		
GAP_IO_CAP_DISPLAY_ONLY	0x00	//!< ble 设备只能显示 pin 码。用于对端输入 pin 码场合。
GAP_IO_CAP_DISPLAY_YES_NO	0x01	//!< 保留
GAP_IO_CAP_KEYBOARD_ONLY	0x02	//!< ble 设备只能输入 pin 码。用于输入 pin 码场合。
GAP_IO_CAP_NO_INPUT_NO_OUTPUT	0x03	//!< ble 设备没有输入和输出的能力
GAP_IO_CAP_KEYBOARD_DISPLAY	0x04	//!< ble 设备同时有输入和输出的能力

3.1.1.21 GAP Security Parameters

//Gap security parameters

```
typedef struct
{
  bool          mitm           //!<   是否启用 middle mode。即是否需要输入 PIN 码。
  bool          ble_secure_conn //!<   是否启用安全链接。保留。不用填。
  uint8_t       io_cap         //!<   ble 设备的输入输出能力，见@GAP_IO_CAP_DEFINES
  uint8_t       pair_init_mode //!<   是否初始化配对，或等待配对。见@ GAP_PAIRING_MODE_DEFINES
  bool          bond_auth      //!<   是否使能配对时的绑定检查。True -每次绑定会检查双方都需要分发加密
                               key 的要求，并且绑定结束后，会上传加密 key。False -绑定时不会检查双方都
                               需求分发加密 key 的需求，并且绑定结束后，不会上传加密 key。
  uint32_t      password       //!< 启用 middle mode 后的本地 pin 码。
}gap_security_param_t;
```

3.1.2 GAP 函数

3.1.2.1 GAP Set Event Callback Function

void gap_set_cb_func(gap_callback_func_t gap_evt_cb)

注册 GAP 事件在应用层的回调函数。

参数：

gap_evt_cb - 应用层的 GAP 事件处理函数。

返回：

None

示例：

```
void proj_gap_evt_func(gap_event_t *event)
{
    switch(event->type)
    {
        case GAP_EVT_ADV_END:
        {
            co_printf("adv_end,status:0x%02x\r\n",event->param.adv_end_status);
            gap_start_advertising(0);
        }
        break;
    }
}
gap_set_cb_func(proj_gap_evt_func);
```

3.1.2.2 GAP Set Advertising Paramters

void gap_set_advertising_param(gap_adv_param_t *p_adv_param)

设置 BLE 的广播参数。该函数设置广播的参数，但不会开始广播。如果广播参数不需要变更，开始广播前不需要反复调用本函数

参数：

p_adv_param - 指向广播参数结构体的指针。具体见 2.1.1 事件中的 gap_adv_param_t 类型定义。参数取值需满足以下条件：

1 adv_intv_min 需要小于等于 adv_intv_max。

2 adv_intv_min 和 adv_intv_max 的取值范围[0x20,0x4000]

返回：

None

示例：

```
// GAP - Advertisement data (max size = 31 bytes, though this is
// best kept short to conserve power while advertisting)
// GAP-广播包的内容,最长 31 个字节.短一点的内容可以节省广播时的系统功耗.
static uint8_t adv_data[] =
{
  // service UUID, to notify central devices what services are included
  // in this peripheral. 告诉 central 本机有什么服务，但这里先只放一个主要的.
  0x03,   // length of this data
  GAP_ADVTYPE_16BIT_MORE,       // some of the UUID's, but not all
```

```
    0xFF, 0xFE,
};
// GAP - Scan response data (max size = 31 bytes, though this is
// best kept short to conserve power while advertisting)
// GAP-Scan response 内容,最长 31 个字节.短一点的内容可以节省广播时的系统功耗.
static uint8_t scan_rsp_data[] =
{
    // complete name  设备名字
    0x12,      // length of this data
    GAP_ADVTYPE_LOCAL_NAME_COMPLETE,
    'S', 'i', 'm', 'p', 'l', 'e', ' ', 'P', 'e', 'r', 'i', 'p', 'h', 'e', 'r', 'a', 'l',

    // Tx power level  发射功率
    0x02,      // length of this data
    GAP_ADVTYPE_POWER_LEVEL,
    0,       // 0dBm
};
gap_adv_param_t adv_param;
adv_param.adv_mode = GAP_ADV_MODE_UNDIRECT;
adv_param.adv_addr_type = GAP_ADDR_TYPE_PUBLIC;
adv_param.adv_chnl_map = GAP_ADV_CHAN_ALL;
adv_param.adv_filt_policy = GAP_ADV_ALLOW_SCAN_ANY_CON_ANY;
adv_param.adv_intv_min = 300;
adv_param.adv_intv_max = 300;
gap_set_advertising_param(&adv_param);
uint8_t adv_data[]= adv_data;
uint8_t rsp_data[]= scan_rsp_data;
gap_set_advertising_data(adv_data,sizeof(adv_data) );
gap_set_advertising_rsp_data(rsp_data,sizeof(scan_rsp_data) );
gap_start_advertising(0);
```

3.1.2.3 GAP Set Advertising data

void gap_set_advertising_data(uint8_t *p_adv_data, uint8_t adv_data_len)

设置 BLE 的广播数据。该函数设置广播的数据，但不会开始广播。如果广播数据不需要变更，开始广播前不需要反复调用本函数

参数：

p_adv_data	-	指向广播数据 buffer 的指针。
adv_data_len	-	广播数据长度值。取值范围：1 ~ 0x1F

返回：

None

3.1.2.4 GAP Set Advertising Response data

void gap_set_advertising_rsp_data(uint8_t *p_rsp_data, uint8_t rsp_data_len)

设置 BLE 的广播扫描回复数据。该函数设置广播的扫描回复数据，但不会开始广播。如果广播扫描回复数据不需要变更，开始广播前不需要反复调用本函数

参数：

p_scan_rsp_data	-	指向广播扫描回复数据 buffer 的指针。
scan_rsp_data_len	-	广播扫描回复数据长度值。取值范围：1 ~ 0x1F

返回：

None

3.1.2.5 GAP Start Advertising

void gap_start_advertising(uint16_t duration)

开始 BLE 广播。需要在设置完广播参数后调用。广播时间到自动停止时会产生 GAP_EVT_ADV_END 事件。

参数：

duration	-	BLE 广播的时长，单位是 10ms。取值范围 0 ~ 65535。 0：广播一直持续，直到调用停止广播的函数。 其他：广播持续 duration * 10ms 时间，然后自动停止。

返回：

None

示例：

```
gap_start_advertising(0);
```

3.1.2.6 GAP Stop Advertising

void gap_stop_advertising(void)

停止 BLE 广播。广播停止时会产生 GAP_EVT_ADV_END 事件。

参数：

None

返回：

None

3.1.2.7 GAP Start Scanning

void gap_start_scan(gap_scan_param_t *p_scan_param)

开始 BLE 扫描, 用于 central 或者 observer 对 peripheral 或者 advertiser 进行查找。扫描产生的广播包通过 GAP_EVT_ADV_REPORT 获取。

参数：

p_scan_param	-	扫描的参数，具体见 2.1.1 事件中的 gap_scan_param_t 类型定义。参数取值需满足以下条件： 1 scan_intv 必须小于等于 scan_window。 2 scan_intv 和 scan_window 都必须大于等于 0x4。 3 duration 取值范围为 0 ~ 65535。 0：扫描一直持续，直到调用停止扫描的函数。 其他：扫描持续 duration * 10ms 时间，然后自动停止。

返回：

None

示例：

```
gap_scan_param_t scan_param;
scan_param.scan_mode = GAP_SCAN_MODE_GEN_DISC;
scan_param.dup_filt_pol = 0;
scan_param.scan_intv = 32;  //scan event on-going time
scan_param.scan_window = 20;
scan_param.duration = 0;
gap_start_scan(&scan_param);
```

3.1.2.8 GAP Stop Scanning

void gap_stop_scan(void)

停止 BLE 扫描，用于 central 或者 observer 停止对 peripheral 或者 advertiser 正在进行的查找。扫描停止时，会产生 GAP_EVT_SCAN_END 事件。

参数：

None

返回：

None

3.1.2.9 GAP Connect Request

void gap_start_conn(struct mac_addr *addr, uint8_t addr_type, uint16_t min_itvl, uint16_t max_itvl, uint16_t slv_latency, uint16_t timeout)

central 设备发起对 peripheral 设备的 BLE 连接。链接建立时会产生 GAP_EVT_SLAVE_CONNECT 或 GAP_EVT_MASTER_CONNECT 事件。

参数：

addr	-	被连接设备的 BD ADDR。
addr_type	-	被连接设备的 BD ADDR 的地址类型，见@defgroup GAP_ADDR_TYPE_DEFINES。只能选择如下取值： GAP_ADDR_TYPE_PUBLIC ，表示共有地址。 GAP_ADDR_TYPE_PRIVATE，表示私有地址。
min_itvl	-	connection interval 的最小值， 单位是 1.25ms。
max_itvl	-	connection interval 的最大值， 单位是 1.25ms。这个值可以和 min_itvl 相等。
slv_latency	-	允许的 slave latency 的个数。

timeout - supervision timeout，单位是 10ms。

返回：

None

示例：

```
struct mac_addr addr= {{0x0C,0x0c,0x0c,0x0c,0x0c,0x0B}};
 gap_start_conn(&addr, GAP_ADDR_TYPE_PUBLIC, 12, 12, 100, 300);
```

3.1.2.10 GAP Cancel Connection Procedure

void gap_stop_conn(void)

停止正在由 gap_start_conn() 发起的连接的过程，central 设备发起。本函数停止正在进行连接的动作，不是断开已经建立的链接。正在链接的动作被停止后，产生 GAP_EVT_CONN_END 事件。

参数：

None

返回：

None

3.1.2.11 GAP Disconnect Request

void gap_disconnect_req(uint8_t conidx)

断开一个已经建立好的 BLE 连接，可由 central 发起，也可由 peripheral 发起。链接断开时会产生 GAP_EVT_DISCONNECT 事件。

参数：

connidx - 要被断开连接的链接号。链接号从 0 开始一直到 app_config,h 定义的最大链接数-1。

返回：

None

3.1.2.12 GAP Get Local Address

void gap_address_get(struct mac_addr *addr)

获取 ble 设备的 local mac 地址。

参数：

addr - 指向装载 mac 地址值 buff 的指针。获取的 mac 地址被赋值到该指针指向的地址。

返回：

None

示例：

```
struct mac_addr addr;
gap_address_get(&addr);
show_reg(&addr.addr[0], 6, 1);    //printf local mac addr.
```

3.1.2.13 GAP Set Local Address

void gap_address_set(struct mac_addr *addr)

设置 ble 设备的 local mac 地址。

参数：

addr - 要设置的设备 BD ADDR。Mac 地址长度默认是 6 个字节

返回：

None

示例：

```
struct mac_addr addr= {{0x0C,0x0c,0x0c,0x0c,0x0c,0x0B}};
gap_address_set(&addr);
```

3.1.2.14 GAP Get Connection Status

bool gap_get_connect_status(uint8_t conidx)

获取某个链接号是否处于链接已建立的状态。

参数：

connidx - 要查询的链接号。链接号从 0 开始一直到 app_config,h 定义的最大链接数-1。

返回：

bool - True，该链接号处于链接状态
False，该链接号处于断开状态

3.1.2.15 GAP Get Encryption Status

bool gap_get_encryption_status(uint8_t conidx)

获取某个链接号是否处于链接已加密的状态。

参数：

connidx - 要查询的链接号。链接号从 0 开始一直到 app_config,h 定义的最大链接数-1。

返回：

bool - True，该链接号处于已加密状态
False，该链接号处于未加密状态

3.1.2.16 GAP Set Device Name

void gap_set_dev_name(uint8_t *p_name,uint8_t len)

设置 ble 设备的名字，该名字会在查询 gap profile 的 dev name 时候被默认使用。

参数：

p_name - 指向设备名字 buff 的指针。

len - 设备名字的长度，取值范围 1~18。

返回：

None

3.1.2.17 GAP Get Device Name

uint8_t gap_get_dev_name(uint8_t* p_name)

获取设置 ble 设备的名字。

参数：

p_name	-	指向设备名字 buff 的指针。

返回：

uint8_t	名字的长度

3.1.2.18 GAP Set Device Appearance

void gap_set_dev_appearance(uint16_t appearance)

设置 ble 设备的外观，该外观类型会在查询 gap profile 的 dev appearance 时候被默认使用。

参数：

appearance	-	要设置的设备外观。见@defgroup GAP_APPEARANCE_VALUES

返回：

None

3.1.2.19 GAP Get Device Appearance

uint16_t gap_get_dev_appearance(void)

获取 ble 设备的默认外观。

参数：

None

返回：

uint16_t	-	设备的外观。见@defgroup GAP_APPEARANCE_VALUES

3.1.2.20 GAP Get Connection Number

uint8_t gap_get_connect_num(void)

获取当前已建立的链接个数。

参数：

None

返回：

uint8_t	-	已经建立的链接的个数。返回值为 0 ~ app_config,h 定义的最大链接数-1。

3.1.2.21 GAP Get Link RSSI

void gap_get_link_rssi(uint8_t conidx)

发起一次获取链接对端设备 rssi 的动作。获取的 rssi 值通过 gap_event: GAP_EVT_LINK_RSSI 返回。

参数：

connidx	-	要查询的链接号。链接号从 0 开始一直到 app_config,h 定义的最大链接数-1。

返回：

None

3.1.2.22 GAP Enable RSSI report in real time

void gap_set_link_rssi_report(bool enable);

使能或者关闭底层持续上报 rssi，在使能之后，用户层需要定义一个接收函数，且在该函数中不要做过多的操作。需采用如下函数定义名称和参数：

```
__attribute__((section("ram_code"))) void gap_rssi_ind(int8_t rssi, uint8_t conidx)
{
    co_printf("rssi: link=%d, rssi=%d.\r\n", conidx, rssi);
}
```

参数：

enable - 选择使能或者关闭

返回：

None

3.1.2.23 GAP Connection Parameters Update Request

void gap_conn_param_update(uint8_t conidx, uint16_t min_intv, uint16_t max_intv, uint16_t slave_latency, uint16_t supervision_timeout)

发起一次更新链接参数的动作。更新成功会产生 GAP_EVT_LINK_PARAM_UPDATE 事件。更新失败会产生 GAP_EVT_LINK_PARAM_REJECT 事件。

参数：

connidx	-	要进行参数更新的链接号。链接号从 0 开始一直到 app_config,h 定义的最大链接数-1
min_intv	-	最小的链接间隔取值。必须小于等于 max_intv。 取值大于 6。单位：0.625us
max_intv	-	最大的链接间隔取值。取值大于 6。单位：0.625us
slave_latency	-	做为 slave 角色时，允许忽略握手的间隔个数。 slave_latency * max_intv <= 6400
supervision_timeout	-	链接断开前，允许的持续握手失败超时时间。单位：10ms

返回：

None

示例：

```
gap_conn_param_update(event->param.slave_connect.conidx, 12, 12, 0, 500);
```

3.2 GATT API

GATT 层的 API 位于 components\ble\include\gatt\gatt_api.h 中。

3.2.1 GATT 事件

3.2.1.1 GATT Property Bitmap Defines

// GATT_PROP_BITMAPS_DEFINES，对应 gatt_attribute_t 的 prop 变量值。

```
GATT_PROP_READ                                  0x0001    //!< att 权限包含 read
```

GATT_PROP_WRITE	0x0002	//!< att 权限包含 write_cmd 和 write_req
GATT_PROP_AUTHEN_READ	0x0004	//!< att 权限包含加密 read
GATT_PROP_AUTHEN_WRITE	0x0008	//!< att 权限包含加密 write_cmd 和 write_req
GATT_PROP_NOTI	0x0100	//!< att 权限包含 notification
GATT_PROP_INDI	0x0200	//!< att 权限包含 indication
GATT_PROP_WRITE_REQ	0x0400	//!< att 权限包含 write request
GATT_PROP_WRITE_CMD	0x0800	//!< att 权限包含 write with no response

3.2.1.2 GATT Operation Defines

// GATT_OPERATION_NAME，对应 gatt_op_cmp_t 的 operation 变量值。		
GATT_OP_NOTIFY	0x1	//!< att notification 操作。
GATT_OP_INDICA	0x2	//!< att indication 操作。
GATT_OP_PEER_SVC_REGISTERED	0x3	//!< 扫描对端服务操作完成。
GATT_OP_WRITE_REQ	0x5	//!< att write_with_response 操作。
GATT_OP_WRITE_CMD	0x6	//!< att write_without_response 操作。
GATT_OP_READ	0x7	//!< att read 操作。

3.2.1.3 GATT Event Type

```
// GATT message events type define
typedef enum
{
    GATTC_MSG_READ_REQ,             //!< 收到对端 gatt read 请求。
    GATTC_MSG_WRITE_REQ,            //!< 收到对端 gatt write 请求。
    GATTC_MSG_ATT_INFO_REQ,         //!< 收到对端 获取 gatt att info 的请求。
    GATTC_MSG_NTF_REQ,              //!< 收到对端的 notification 的数据。
    GATTC_MSG_READ_IND,             //!< 收到了对端回复读操作的数据。
    GATTC_MSG_CMP_EVT,              //!< 某个 gatt 操作已完成。
    GATTC_MSG_LINK_CREATE,          //!< 链接已创建。
    GATTC_MSG_LINK_LOST             //!< 链接已断开。
}gatt_msg_evt_t;
```

3.2.1.4 GATT Operation Complete Event

```
// GATT message event operation done structure
typedef struct
{
    uint8_t         operation       //!< GATT 操作的类型, see @GATT_OPERATION_NAME
    uint8_t         status          //!< GATT 操作完成的状态
    void *          arg             //!< GATT 操作完成后可能会用到的参数的指针。
}gatt_op_cmp_t;
```

3.2.1.5 GATT Message Data

// GATT message event data structure

```
typedef struct
{
   uint16_t       msg_len          //!< GATT 消息的长度
   void *         p_msg_data       //!< 指向 GATT 消息数据的指针。
}gatt_msg_hdl_t;
```

3.2.1.6 GATT Message Event

// GATT message event structure.

```
typedef struct
{
   gatt_msg_evt_t                   msg_evt     //!< GATT event 的类型
   uint8_t                          conn_idx    //!< Gatt event 对应的链接号
   uint8_t                          svc_id      //!< Gatt event 对应的 service ID
   uint16_t                         att_idx     //!< Gatt evnet 对应的 attribute idx 号
   union{
      gatt_msg_hdl_t                msg         //!< Gatt event 消息类型时对应的结构体
      gatt_op_cmp_t                 op          //!< Gatt event 操作完成类型时对应的结构体
   }param;
}gatt_msg_t;
```

3.2.1.7 GATT Attribute Structure

// BLE attribute define format define.

```
typedef struct
{
   gatt_uuid_t       uuid        //!<Attribute UUID
   uint16_t          prop        //!< Attribute 权限, see @GATT_PROP_BITMAPS_DEFINES
   uint16_t          max_size    //!< Attribute 数据的最大长度。
   uint8_t *         p_data      //!< 指向 Attribute 数据 buffer 的指针
}gatt_attribute_t;
```

说明：

在定义 profile service 的 att 数组，给数组赋值时，max_size 和 p_data 可以按如下情况使用：

1 max_size>0， p_data != NULL。 创建 profile 时内部会分配内存，并拷贝 p_data 的数据到内部。

2 max_size>0， p_data == NULL。创建 profile 时内部不会分配内存，对端如果读取该 att 的值，会产生读操作消息。

3 max_size==0, p_data == NULL。该 att 不能被读。或者读该 att 会返回 Null。

4 max_size==0, p_data != NULL。 该情况不能出现。

针对 att 为 GATT_PRIMARY_SERVICE_UUID 时，需要按情况 1 对该 att idx 赋值。max_size 填 service uuid 的长度，p_data 指向 service_uuid 的 buffer 地址。

p_data 指向的 buffer 推荐定义为 const 类型，节省 ram。

3.2.1.8 GATT Service Structure

// Profile service define format.

```
typedef struct
{
    const gatt_attribute_t *    p_att_tb            //!< 指向 profile service att 定义数组的指针
    uint8_t                     att_nb              //!< profile service att 的个数。
    gatt_msg_handler_t          gatt_msg_handler    //!< profile service 对应的消息接收处理回调函数。
}gatt_service_t;
```

3.2.1.9 GATT Client Structure

// Profile client define format.

```
typedef struct
{
    const gatt_uuid_t           p_att_tb            //!< 指向 profile client att 对应 uuid 的数组的指针
    uint8_t                     att_nb              //!< profile client att 的个数。
    gatt_msg_handler_t          gatt_msg_handler    //!< profile client 对应的消息接收处理回调函数。
}gatt_client_t;
```

3.2.1.10 GATT Client Read

// BLE client read format.

```
typedef struct
{
    uint8_t     conidx      //!< Profile client 读操作对应的链接号。
    uint8_t     client_id   //!< profile client 读操作对应的 client_id。
    uint8_t     att_idx     //!< profile client 读操作对应的 att idx 号
}gatt_client_read_t;
```

3.2.1.11 GATT Client Enable Notification

// BLE client enable ntf format.

```
typedef struct
{
    uint8_t       conidx      //!< Profile client  进行使能 ntf 操作对应的链接号。
    uint8_t       client_id   //!< profile client 进行使能 ntf 操作对应的 client_id。
    uint8_t       att_idx     //!< profile client  进行使能 ntf 操作对应的 att idx 号
}gatt_client_enable_ntf_t;
```

3.2.1.12 GATT Notification Structure

// BLE notification format.

```
typedef struct
{
    uint8_t       conidx      //!< Profile client notification 操作对应的链接号。
    uint8_t       client_id   //!< profile client notification 操作对应的 client_id。
    uint8_t       att_idx     //!< profile client notification 操作对应的 att idx 号
    uint8_t *     p_data      //!<  指向 profile client notification 操作的数据地址的指针。
    uint8_t       data_len    //!< profile client notification 操作的数据的长度。
}gatt_ntf_t;
```

3.2.1.13 GATT Indication Structure

// BLE indication format.

```
typedef struct
{
    uint8_t       conidx      //!< Profile client indication 操作对应的链接号。
    uint8_t       client_id   //!< profile client indication 操作对应的 client_id。
    uint8_t       att_idx     //!< profile client indication 操作对应的 att idx 号
    uint8_t *     p_data      //!<  指向 profile client indication 操作的数据地址的指针。
    uint8_t       data_len    //!< profile client indication 操作的数据的长度。
}gatt_ind_t;
```

3.2.2 GATT 函数

3.2.2.1 GATT Add Service

uint8_t gatt_add_service(gatt_service_t *p_service)

创建一个 profile service。

参数：

p_service - 指向 profile service 数组的指针。

返回：

uint8_t	0xff，创建失败，profile 的个数达到上限。 其他值，创建成功，返回值是 profile 被分配的 svc_id 号。 系统能创建的 profile 个数由 app_config.h 内的宏 USER_PRF_NB 定义。

3.2.2.2 GATT Add Client

uint8_t gatt_add_client(gatt_client_t *p_client)

创建 profile client。系统只能创建一个 client 的服务。用户需要把所有的 att uuid 均放在一个 client 服务数组内。

参数：

p_client	-	指向 profile client 数组的指针。

返回：

uint8_t	0xff，创建失败，profile 的个数达到上限。 其他值，创建成功，返回值是 profile 被分配的 client_id 号。

3.2.2.3 GATT Change Service UUID

void gatt_change_svc_uuid(uint8_t svc_id,uint8_t att_idx,uint8_t *new_uuid,uint8_t uuid_len)

在 profile service 被创建后，更新某个 att 的 uuid。

参数：

svc_id	-	要更改 uuid 的 profile svc_id
att_idx	-	要更改的 att idx。
new_uuid	-	指向新 uuid buffer 的指针
uuid_len	-	新 uuid 的长度

返回：

None

3.2.2.4 GATT Change Client UUID

void gatt_change_client_uuid(uint8_t client_id,uint8_t att_idx,uint8_t *new_uuid,uint8_t uuid_len)

在 profile client 被创建后，更新某个 att 的 uuid。

参数：

client_id	-	要更改 uuid 的 profile client_id
att_idx	-	要更改的 att idx。
new_uuid	-	指向新 uuid buffer 的指针
uuid_len	-	新 uuid 的长度

返回：

None

3.2.2.5 GATT Discover Peer Device All Services

void gatt_discover_all_peer_svc(uint8_t client_id,uint8_t conidx)

在做为 peripheral 设备链接建立后，发起扫描对端所有 service 服务。

参数：

client_id - 要进行扫描的 profile client_id 号。

conidx - 要进行扫描操作的链接号。链接号从 0 开始一直到 app_config,h 定义的最大链接数-1

返回：

None

示例：

```
void proj_gap_evt_func(gap_event_t *event)
{
    switch(event->type)
    {
        case GAP_EVT_MASTER_CONNECT:
        {
            extern uint8_t client_id;
            gatt_discovery_all_peer_svc(client_id,event->param.master_encrypt_conidx);
        }
        break;
    }
}
```

3.2.2.6 GATT Discover Peer Device Service By UUID

void gatt_discover_peer_svc(uint8_t client_id, uint8_t conidx, uint8_t uuid_len, uint8_t *group_uuid)

在做为 peripheral 设备链接建立后，发起扫描对端某一个特定 service uuid 服务

参数：

client_id - 要进行扫描的 profile client_id 号。

conidx - 要进行扫描操作的链接号。链接号从 0 开始一直到 app_config,h 定义的最大链接数-1

uuid_len - 要扫描的对端的 service uuid 的长度

group_uuid - 要扫描的对端的 service uuid。

返回：

None

3.2.2.7 GATT Write Request

void gatt_client_write_req(gatt_client_write_t write_att)

peripheral 的设备向对端进行一次 write with response 操作，需要在 svc 服务扫描完毕后调用。

参数：

write_att - 写操作的结构体。

返回：

None

3.2.2.8 GATT Write Command

void gatt_client_write_cmd(gatt_client_write_t write_att)

peripheral 的设备向对端进行一次 write without response 操作，需要在 svc 服务扫描完毕后调用。

参数：

write_att - 写操作的结构体。

返回：

None

示例：

```
uint16_t client_msg_handler(gatt_msg_t *p_msg)
{
    switch(p_msg->msg_evt)
    {
        case GATTC_MSG_CMP_EVT:
        {
            if(p_msg->param.op.operation == GATT_OP_PEER_SVC_REGISTERED)
            {
                gatt_client_write_t write;
                write.conidx = p_msg->conn_idx;
                write.client_id = client_id;
                write.att_idx = 1;
                write.p_data = "\x1\x2\x3\x4\x5\x6\x7";
                write.data_len = 7;
                gatt_client_write_cmd(write);
            }
        }
        break;
    }
    return 0;
}
```

3.2.2.9 GATT Enable Notification

void gatt_client_enable_ntf(gatt_client_enable_ntf_t ntf_enable_att)

peripheral 的设备使能某个 uuid 的 notification 的功能，需要在 svc 服务扫描完毕后调用。

参数：

ntf_enable_att - 要进行使能 notification 功能的结构体。

返回：

None

示例：

```
uint16_t client_msg_handler(gatt_msg_t *p_msg)
{
    switch(p_msg->msg_evt)
    {
        case GATTC_MSG_CMP_EVT:
        {
            if(p_msg->param.op.operation == GATT_OP_PEER_SVC_REGISTERED)
            {
                gatt_client_enable_ntf_t ntf_enable;
                ntf_enable.conidx = p_msg->conn_idx;
                ntf_enable.client_id = client_id;
                ntf_enable.att_idx = 0;
                gatt_client_enable_ntf(ntf_enable);
            }
        }
        break;
    }
    return 0;
}
```

3.2.2.10 GATT Read Request

void gatt_client_read(gatt_client_read_t read_att)

peripheral 的设备向对端进行一次 read 操作，需要在 svc 服务扫描完毕后调用。读取的内容通过 profile msg 消息 GATTC_MSG_READ_IND 进行上报。

参数：

read_att - 要进行读操作的结构体。

返回：

None

示例：

```
uint16_t client_msg_handler(gatt_msg_t *p_msg)
{
    switch(p_msg->msg_evt)
    {
        case GATTC_MSG_CMP_EVT:
        {
            if(p_msg->param.op.operation == GATT_OP_PEER_SVC_REGISTERED)
            {
                gatt_client_read_t read;
                read.conidx = p_msg->conn_idx;
                read.client_id = client_id;
                read.att_idx = 0;
                gatt_client_read(read);
            }
        }
        break;
    }
    return 0;
}
```

3.2.2.11 GATT Notification

void gatt_notification(gatt_ntf_t ntf_att)

central 设备向对端进行一次 notification 操作，需要在接收到对端 ntf 使能的消息之后，进行调用。

参数：

ntf_att - 要进行 ntf 操作的结构体。

返回：

None

示例：

```
gatt_ntf_t ntf_att;
ntf_att.att_idx = 2;
ntf_att.conidx = p_msg->conn_idx;
ntf_att.svc_id = svc_id;
ntf_att.data_len = 4;
uint8_t tmp[] = "12345";
ntf_att.p_data = tmp;
gatt_notification(ntf_att);
```

3.2.2.12 GATT Indication

void gatt_indication(gatt_ind_t ind_att)

central 设备向对端进行一次 indification 操作，需要在接收到对端 ind 使能的消息之后，进行调用。

参数：

ind_att - 要进行 ind 操作的结构体。

返回：

None

3.2.2.13 GATT MTU Exchange Request

void gatt_mtu_exchange_req(uint8_t conidx)

向对端发起一次 mut 交换的请求。

参数：

conidx - 要进行 mtu 交换的链接号。链接号从 0 开始一直到 app_config,h 定义的最大链接数-1

返回：

None

3.2.2.14 GATT Get negotiated MTU size

uint16_t gatt_get_mtu(uint8_t conidx)

获取当前链接使用的 MTU 打字奥。

参数：

conidx - 要进行 mtu 交换的链接号。链接号从 0 开始一直到 app_config,h 定义的最大链接数-1

返回：

uint16_t - 当前链接使用的 MTU 大小

3.2.2.15 GATT deal message from HOST (condition: RTOS enable)

void gatt_msg_default(os_event_t *msg)

在启用 RTOS 时，用户需要创建一个专门用于处理 GATT 消息的 task，HOST 会向该 task 发送消息，该 task 收到消息后调用本函数用于处理收到的消息。

参数：

msg - 从 host 收到的消息，也就是将要被处理的消息

返回：

None

3.3 Mesh API

Mesh 的相关 API 位于 components\ble\include\mesh\mesh_api.h 中。

3.3.1 Mesh 事件

3.3.1.1 Mesh Event Type Defines

// Mesh event type define

```
enum mesh_event_type_t
{
    MESH_EVT_STARTED,                   //!< Mesh 启动完成。
    MESH_EVT_STOPPED,                   //!< Mesh 功能停止。
    MESH_EVT_RESET,                     //!< 从 provisioner 那里收到一个 reset 命令。
    MESH_EVT_READY,                     //!< Mesh 准备就绪，可以启动。
    MESH_EVT_MODEL_APPKEY_BINDED,       //!< Model 成功和一个 appkey 绑定。
    MESH_EVT_MODEL_GRPADDR_SUBED,       //!< Model 成功订阅一个 group 地址。
    MESH_EVT_PROV_PARAM_REQ,            //!< 从 provisioner 收到 provision parameter request。
    MESH_EVT_PROV_AUTH_DATA_REQ,        //!< 从 provisioner 收到 authentication data request。
    MESH_EVT_PROV_RESULT,               //!< 收到 provision 结果。
    MESH_EVT_UPDATE_IND,                //!< 收到状态更新通知。
    MESH_EVT_RECV_MSG,                  //!< 收到 mesh 的数据
    MESH_EVT_COMPO_DATA_REQ,            //!<从 provisioner 收到 composition data request。
    MESH_EVT_ADV_REPORT,                //!< 收到 BLE 广播包的事件，通知应用层处理。
};
```

3.3.1.2 Mesh network information updates

// Mesh network information update type.

```
enum mesh_update_type_t
{
    MESH_UPD_TYPE_NET_KEY_UPDATED,      //!< Network key 更新
    MESH_UPD_TYPE_NET_KEY_DELETED,      //!< Network key 删除
    MESH_UPD_TYPE_APP_KEY_UPDATED,      //!< Application key 更新
    MESH_UPD_TYPE_APP_KEY_DELETED,      //!< Application key 删除
    MESH_UPD_TYPE_PUBLI_PARAM,          //!< Model publication 参数更新
    MESH_UPD_TYPE_SUBS_LIST,            //!< Model subscription list 更新
    MESH_UPD_TYPE_BINDING,              //!< Model/application key 绑定状态更新
    MESH_UPD_TYPE_STATE,                //!< 状态更新
};
```

3.3.1.3 Mesh Supported Features

// Mesh supported feature type define.

```
enum mesh_feature_t
{
```

```
    MESH_FEATURE_RELAY          = (1<<0),          //!< Relay 模式。
    MESH_FEATURE_PROXY          = (1<<1),          //!< Proxy 模式。
    MESH_FEATURE_FRIEND         = (1<<2),          //!< Friendly 节点模式。
    MESH_FEATURE_LOW_POWER      = (1<<3),          //!< Low Power Node 模式。
    MESH_FEATURE_PB_GATT        = (1<<4),          //!< 支持 GATT bearer。
};
```

3.3.1.4 Mesh Provision Output OOB Mode

```
// Mesh provision out oob mode.
enum mesh_prov_out_oob_t
{
    MESH_PROV_OUT_OOB_BLINK             = 0x0001,   //!< Bit[0]: 闪灯。
    MESH_PROV_OUT_OOB_BEEP              = 0x0002,   //!< Bit[1]: 叫出声。
    MESH_PROV_OUT_OOB_VIBRATE           = 0x0004,   //!< Bit[2]: 抖起来。
    MESH_PROV_OUT_OOB_NUMERIC           = 0x0008,   //!< Bit[3]: 数字显示。
    MESH_PROV_OUT_OOB_ALPHANUMERIC      = 0x0010,   //!< Bit[4]: 字母显示。
                                                    //!< Bit[5-15]: 保留给将来用。
};
```

3.3.1.5 Mesh Provision Input OOB Mode

```
/// Input OOB Action field values
enum mesh_prov_in_oob_t
{
    MESH_PROV_IN_OOB_PUSH               = 0x0001,   //!< Bit[0]: 按。
    MESH_PROV_IN_OOB_TWIST              = 0x0002,   //!< Bit[1]: 转。
    MESH_PROV_IN_OOB_NUMERIC            = 0x0004,   //!< Bit[2]: 数字输入。
    MESH_PROV_IN_OOB_ALPHANUMERIC       = 0x0008,   //!< Bit[3]: 字母输入。
                                                    //!< Bit[4-15]: 保留给将来用。
};
```

3.3.1.6 Mesh Provision Information

```
// Mesh provision information.
enum mesh_prov_info
{
    MESH_PROV_INFO_URI_HASH_PRESENT     = (1 << 0),     //!< 在 unprovisioned device beacon 包中是否显示 URI Hash。
};
```

3.3.1.7 Mesh Provision States

/// State of the provisioning

```
enum mesh_prov_state
{
    MESH_PROV_STARTED,                  //!< Provisioning 过程已经开始。
    MESH_PROV_SUCCEED,                  //!< Provisioning 成功。
    MESH_PROV_FAILED,                   //!< Provisioning 失败。
};
```

3.3.1.8 Mesh Publish Message Type

// Mesh publish message type define

```
typedef struct
{
    uint8_t     element_idx;        //!< Element index。
    uint32_t    model_id;           //!< Model ID。
    uint32_t    opcode;             //!< Operation code (1, 2 or 3 字节长度的 operation code)。
    uint16_t    msg_len;            //!< 数据长度。
    uint8_t     msg[];              //!< 数据内容。
} mesh_publish_msg_t;
```

3.3.1.9 Mesh Response Message type

// Mesh response message type define

```
typedef struct
{
    uint8_t     element_idx;    //!< 数据对应的 Element index
    uint8_t     app_key_lid;    //!< 数据要使用的 App key 的 index
    uint32_t    model_id;       //!< 数据对应的 Model ID
    uint32_t    opcode;         //!< 数据对应的 operation code (1, 2 或者 3 字节长度的 operation code.)
    uint16_t    dst_addr;       //!< 数据对应的目标地址
    uint16_t    msg_len;        //!< 数据长度
    uint8_t     msg[];          //!< 数据内容
} mesh_rsp_msg_t;
```

3.3.1.10 Mesh Receive Message Type

// Mesh receive message type define

```
typedef struct
{
    uint32_t    model_id;           //!< Model ID。
```

```
    uint32_t        opcode;                 //!< Operation code (1, 2 or 3 字节长度的 operation code)。
    uint16_t        msg_len;                //!< 数据长度。
    uint8_t         *p_msg;                 //!< 数据内容。
} mesh_recv_msg_t;
```

3.3.1.11 Mesh Model Structure

// Mesh model struct.

```
typedef struct
{
    uint32_t        model_id;               //!< Model ID。
    uint8_t         model_vendor;           //!< 是 SIG 定义的 model 还是自定义 model。 @MODEL_VENDOR_MODE
    uint8_t         element_idx;            //!< 设备中的 Element index。
} mesh_model_t;
```

3.3.1.12 Mesh Provision State Change Event

// Mesh Provisioning state change indication

```
typedef struct
{
    uint8_t         state;                  //!< Provision 状态。(@see enum mesh_prov_state)
    uint16_t        status;                 //!< Provision 失败时候的原因。
} mesh_prov_result_ind_t;
```

3.3.1.13 Mesh Model Message Indication

// Inform reception of a specific mesh message

```
typedef struct
{
    uint32_t        model_id;               //!< Model id。
    uint8_t         element;                //!< element index。
    uint8_t         app_key_lid;            //!< AppKey Local identifier (Required for a response)。
    int8_t          rssi;                   //!< 收到的 PDU 的 RSSI 值。
    uint8_t         not_relayed;            //!< 1 = 表示消息直接从源设备哪里收到； 0 = 消息是经过 relay 到达。
    uint32_t        opcode;                 //!< Operation code (1, 2 or 3 字节长度的 operation code)。
    uint16_t        src;                    //!< 消息源设备的地址。 (Required for a response)
    uint16_t        msg_len;                //!< 数据长度。
    const uint8_t   *msg;                   //!< 数据内容。
} mesh_model_msg_ind_t;
```

3.3.1.14 Network Key is NOT updated

// Network key information entry structure if network key is not being updated

```
typedef struct
{
   uint8_t          length;          //!< Entry length
   uint8_t          info;            //!< Information
   uint16_t         netkey_id;       //!< NetKey ID
   uint8_t          key[16];         //!< Network Key
} mesh_netkey_t;
```

3.3.1.15 Network Key is updated

// Network key information entry structure if network key is being updated

```
typedef struct
{
   uint8_t          length;          //!< Entry length
   uint8_t          info;            //!< Information
   uint16_t         netkey_id;       //!< NetKey ID
   uint8_t          key[16];         //!< Network Key
   uint8_t          new_key[16];     //!< New network key
}mesh_netkey_upd_t;
```

3.3.1.16 Application Key is NOT updated

// Application key information entry structure if application key is not being updated

```
typedef struct
{
   uint8_t          length;          //!< Entry length
   uint8_t          info;            //!< Information
   uint16_t         netkey_id;       //!< NetKey ID
   uint16_t         appkey_id;       //!< AppKey ID
   uint8_t          key[16];         //!< Network Key
} mesh_appkey_t;
```

3.3.1.17 Application Key is updated

// Application key information entry structure if application key is being updated

```
typedef struct
{
   uint8_t          length;          //!< Entry length
   uint8_t          info;            //!< Information
```

```
    uint16_t        netkey_id;      //!< NetKey ID
    uint16_t        appkey_id;      //!< AppKey ID
    uint8_t         key[16];        //!< Network Key
    uint8_t         new_key[16];    //!< New network key
}mesh_appkey_upd_t;
```

3.3.1.18 Model publication parameter for NOT virtual publication address

// Model publication parameter entry structure if publication address is not a virtual address

```
typedef struct
{
    uint8_t         length;         //!< Entry length
    uint8_t         info;           //!< Information
    uint16_taddress element_addr;   //!< Element address
    uint32_t        model_id;       //!< Model ID
    uint16_t        addr;           //!< Publication address
    uint16_t        appkey_id;      //!< AppKey ID
    uint8_t         ttl;            //!< TTL
    uint8_t         period;         //!< Period
    uint8_t         retx_params;    //!< Retransmission parameters
    uint8_t         friend_cred;    //!< Friend credentials
}mesh_publi_t;
```

3.3.1.19 Model publication parameter for virtual publication address

// Model publication parameter entry structure if publication address is a virtual address

```
typedef struct
{
    uint8_t         length;         //!< Entry length
    uint8_t         info;           //!< Information
    uint16_taddress element_addr;   //!< Element address
    uint32_t        model_id;       //!< Model ID
    uint16_t        addr;           //!< Publication address
    uint16_t        appkey_id;      //!< AppKey ID
    uint8_t         ttl;            //!< TTL
    uint8_t         period;         //!< Period
    uint8_t         retx_params;    //!< Retransmission parameters
    uint8_t         friend_cred;    //!< Friend credentials
    uint8_t         label_uuid[16]; //!< Label UUID
}mesh_publi_virt_t;
```

3.3.1.20 Model subscription entry

// Model subscription list entry structure

```
typedef struct
{
   uint8_t          length;          //!< Entry length
   uint8_t          info;            //!< Information
   uint16_t         element_addr;    //!< Element address
   uint32_t         model_id;        //!< Model ID
   uint8_t          list[];          //!< List
}mesh_subs_t;
```

3.3.1.21 Key binding entry

// Model/Application key binding entry structure

```
typedef struct
{
   uint8_t          length;          //!< Entry length
   uint8_t          info;            //!< Information
   uint16_t         element_addr;    //!< Element address
   uint32_t         model_id;        //!< Model ID
   uint16_t         appkey_ids[];    //!< List of AppKey IDs
}mesh_binding_t;
```

3.3.1.22 Configuration updated indication

// Configuration update indication message structure

```
typedef struct
{
   uint8_t          upd_type;        //!< Update type
   uint8_t          length;          //!< Entry length
   uint8_t          data[];          //!< Entry value
}mesh_update_ind_t;
```

3.3.1.23 Mesh Event Structure

// Mesh event structure

```
typedef struct
{
   enum mesh_event_type_t     type;          //!< Mesh 事件类型，@mesh_event_type_t
   union
   {
```

```
        mesh_prov_result_ind_t      prov_result;          //!< Provision 结果，@mesh_prov_result_ind_t
        void                        *update_ind;          //!< 状态更新，包括当前状态和 key 更新装情况。
        mesh_model_msg_ind_t        model_msg;            //!< Mesh model 的消息。
        uint8_t                     compo_data_req_page;  //!< Mesh composition data request page。
    gap_evt_adv_report_t            adv_report;           //!< BLE 广播包内容。
    } param;
} mesh_event_t;
```

3.3.2 Mesh 函数

3.3.2.1 Mesh Set Application Callback Function

void mesh_set_cb_func(mesh_callback_func_t mesh_evt_cb)

注册 mesh 事件在应用层的回调函数。

参数：

mesh_evt_cb - 应用层的 Mesh 事件处理函数。

返回：

None

3.3.2.2 Mesh Initialization

void mesh_init(enum mesh_feature_t feature, uint32_t store_addr)

初始化 mesh 的功能。

参数：

feature - 应用层的 GAP 事件处理函数。

store_addr - 用于存放 mesh link 信息的 flash 地址。 mesh link 信息包含 network key, app key, binding 信息等。详细参考 4.3.1 章节 mesh_feature_t。

返回：

None

3.3.2.3 Mesh Set Runtime

void mesh_set_runtime(void)

设置当前时间。

参数：

None

返回：

None

3.3.2.4 Mesh Start

void mesh_start(void)

开始运行 mesh。这个必须在 mesh_init()被调用之后执行。

参数：

None

返回：

None

3.3.2.5 Mesh Stop

void mesh_stop(void)

停止 mesh 功能。

参数：

None

返回：

None

3.3.2.6 Mesh Model Bind AppKey

void mesh_model_bind_appkey(uint32_t model_id, uint8_t element, uint8_t app_key_index)

给特定 model 绑定一个 appkey。通常 appkey 是由 provisioner 分发并且绑定，某些情况下如果本地保存或者有预设值 appkey 列表的话，可以通过这个 API 可以从应用层选择一个本地 appkey 进行绑定。

参数：

model_id - 需要被绑定 appkey 的 model。

element - model 归属的 element。

app_key_index - appkey 在 appkey 列表中的序列号。

返回：

None

3.3.2.7 Mesh Model Subscribe Group Message

void mesh_model_sub_group_addr(uint32_t model_id, uint8_t element, uint16_t group_addr)

model 订阅特定的组播地址。用于该 model 接收 group_addr 对应组播内容。

参数：

model_id - 需要操作的 model 的 ID。

element - model 归属的 element。

group_addr - 被订阅的组地址。

返回：

None

3.3.2.8 Mesh Add Models

void mesh_add_model(const mesh_model_t *p_model)

添加 mesh model 到协议栈中。可以用于同时添加多个 model，提前定义好 model 数据表，以数组的形式传递到函数即可。

参数：

p_model - 提前定义好的 mesh model 的数据表，以数组的形式存在。成员结构参考 4.3.1 章节 mesh_model_t。

返回：

None

3.3.2.9 Mesh Publish Message

void mesh_publish_msg(mesh_publish_msg_t *p_publish_msg)

在 mesh 网络中散播一个数据。

参数：

p_publish_msg - 需要 publish 的 message 的数据结构。详细参考 4.3.1 章节 mesh_publish_msg_t。

返回：

None

3.3.2.10 Mesh Send Response message

void mesh_send_rsp(mesh_rsp_msg_t *p_rsp_msg)

在 mesh 网络中发送 response 包。

参数：

p_rsp_msg - 需要发送的 response 的 message 的数据结构。详细参考 4.3.1 章节 mesh_publish_msg_t。

返回：

None

3.3.2.11 Mesh Provision Parameters response

void mesh_send_prov_param_rsp(uint8_t *uuid, uint32_t uri_hash, uint16_t oob_info, uint8_t pub_key_oob, uint8_t static_oob, uint8_t out_oob_size, uint8_t in_oob_size, uint16_t out_oob_action, uint16_t in_oob_action, uint8_t nb_elt, uint8_t info);

对 provision request 的回复。Provision request 来自 provisioner。

参数：

uuid - 设备的 uuid。

uri_hash - uri_hash 值。

oob_info - oob 信息。

pub_key_oob - Public key oob 信息。

static_oob - Static oob 信息。

out_oob_size - Out oob size。

in_oob_size - In oob size。

out_oob_action	-	Out oob action。
in_oob_action	-	In oob action。
nb_elt	-	设备中含有的 element 的数量。
info	-	额外信息，以 bit 位的形式定义。

返回：

None

3.3.2.12 Mesh Provision Authentication Data Response

void mesh_send_prov_auth_data_rsp(uint8_t accept, uint8_t auth_size, uint8_t *auth_data)

回复从 provisioner 发来的 provision authentication data 的请求。用于 provisioning 的过程中。

参数：

accept	-	接收还是拒绝对方。
auth_size	-	authentication data 的长度。
auth_data	-	authentication data 的内容。

返回：

None

3.3.2.13 Mesh Composition Data Response

void mesh_send_compo_data_rsp(uint8_t page, uint8_t *data, uint8_t length)

对 composition data request 的回复。

参数：

page	-	回复的数据所属的 page。
data	-	回复的数据内容。
length	-	回复的数据内容的长度。

返回：

None

3.3.2.14 Mesh Store Information Into Flash

void mesh_info_store_into_flash(void)

把 mesh 网络的信息保存到 flash 中。为了避免频繁对 flash 进行操作，应用层应在在收到 MESH_EVT_UPDATE_IND 事件之后隔至少 2 秒之后再调用此函数。

参数：

None

返回：

None

3.3.2.15 Mesh Clear Information In Flash

void mesh_info_clear(void)

把 mesh 网络保存在 flash 中的信息删除。

参数：

None

返回：

None

3.4 Security API

Security 相关 API 也位于 components\ble\include\gap\gap_api.h 文件中。

3.4.1 Security 函数

3.4.1.1 GAP Bond Manager Initializtion

void gap_bond_manager_init(uint32_t flash_addr,uint32_t svc_flash_addr,uint8_t max_dev_num,bool enable)

初始化绑定管理功能。绑定管理功能启用后，会在链接建立回调事件之前进行绑定地址检查。该函数不能在 user_entry_before_ble_init() 入口函数调用。

参数：

flash_addr	-	指定存储绑定设备秘钥的 flash 地址，地址必须是 4096 整数倍。
svc_flash_addr	-	指定存储绑定设备服务信息的 flash 地址，地址必须是 4096 整数倍。
max_dev_num	-	指定最大能支持的绑定设备的个数。取值 1~ app_config,h 定义的最大链接数。
enable	-	是否启用该功能。 True，启用绑定管理功能。 False，禁止绑定管理功能。

返回：

None

示例：

```
gap_bond_manager_init(0x32000,0x33000,8,true);
```

3.4.1.2 GAP Bond Manager Delete All Bondings

void gap_bond_manager_delete_all(void)

删除所有存储在 flash 内的绑定设备的秘钥和设备服务信息。需要在绑定管理功能初始化之后调用。该函数不能在 user_entry_before_ble_init() 入口函数调用。

参数：

None

返回：

None

3.4.1.3 GAP Bond Manager Delete Single Bondging

void gap_bond_manager_delete(uint8_t *mac_addr, uint8_t addr_type)

删除某个设备存储在 flash 内的绑定设备的秘钥和设备服务信息。需要在绑定管理功能初始化之后调用。该函数不能在

user_entry_before_ble_init() 入口函数调用。

参数：

None

返回：

None

3.4.1.4 GAP Set Security Parameters

void gap_security_param_init(gap_security_param_t *sec_param)

初始化安全绑定操作时的参数。MITM，IO Capbilities，password 等。

参数：

sec_param - 指向安全绑定参数结构体的指针。

返回：

None

示例：

```
gap_security_param_t param =
{
        .mitm = true,
        .ble_secure_conn = false,
        .io_cap = GAP_IO_CAP_DISPLAY_ONLY,
        .pair_init_mode = GAP_PAIRING_MODE_WAIT_FOR_REQ,
        .bond = true,
        .password = 123456,
};
gap_security_param_init(&param);
```

3.4.1.5 GAP Sending Pairing Password

void gap_security_send_pairing_password(uint32_t conidx,uint32_t password)

发送绑定操作时所需的密码，即 pin 码。

参数：

connidx - 要进行回复密码的链接号。链接号从 0 开始一直到 app_config,h 定义的最大链接数-1

password - 回复绑定操作所需的密码。一般为一个小于 999999 的值。

返回：

None

示例：

```
gap_security_send_pairing_password(0,123456);
```

3.4.1.6 GAP Pairing Request

void gap_security_pairing_req(uint8_t conidx)

peripheral 的设备在链接建立后，主动发起一次新的绑定操作。一般来说，如果链接设备的绑定地址检测结果为 false 时，进

行绑定操作。加密成功后，会产生 GAP_SEC_EVT_MASTER_ENCRYPT 事件。

参数：

connidx - 要发起绑定的链接号。链接号从 0 开始一直到 app_config,h 定义的最大链接数-1。

返回：

None

3.4.1.7 GAP Encrypt Request

void gap_security_enc_req(uint8_t conidx)

peripheral 的设备在链接建立后，主动发起一次加密操作。一般来说，如果链接设备的绑定地址检测结果为 true 时，进行加密操作。加密成功后，会产生 GAP_SEC_EVT_MASTER_ENCRYPT 事件。

参数：

connidx - 要发起绑定的链接号。链接号从 0 开始一直到 app_config,h 定义的最大链接数-1。

返回：

None

3.4.1.8 GAP Get Bond Status

bool gap_security_get_bond_status(void)

获取最近一次链接建立后，相连接的对端设备的绑定状态。

参数：

None

返回：

bool True，最近一次链接的对端设备已绑定。
False，最近一次链接的对端设备未绑定。

示例：

```
void proj_gap_evt_func(gap_event_t *event)
{
    switch(event->type)
    {
        case GAP_EVT_MASTER_CONNECT:
        {
            if (gap_security_get_bond_status())
                gap_security_enc_req(event->param.master_connect.conidx);
            else
                gap_security_pairing_req(event->param.master_connect.conidx);
        }
        break;
    }
}
gap_set_cb_func(proj_gap_evt_func);
```

3.4.1.9 GAP Security Request

void gap_security_req(uint8_t conidx)
central 的设备在链接建立后，发出绑定或加密的请求。等 peripheral 设备回应。 **参数：** connidx - 要发送请求的链接号。链接号从 0 开始一直到 app_config,h 定义的最大链接数-1。 **返回：** None

示例：

```
void proj_gap_evt_func(gap_event_t *event)
{
    switch(event->type)
    {
        case GAP_EVT_SLAVE_CONNECT:
        {
            gap_security_req(event->param.slave_connect.conidx);
        }
        break;
    }
}
gap_set_cb_func(proj_gap_evt_func);
```

3.5 BLE Profiles

SDK 里面以源码的形式实现了 BLE 常用的 profile，供用户参考和修改。Profile 可以在 SDK 安装目录 \components\ble\profiles 下找到。

3.5.1 HID

Human Interface Device。HID 主要用于 BLE 鼠标键盘和遥控器的开发。
HID service 的定义在 hid_service.c 文件中。HID service 包含了 service 定义和一系列特征值定义。HID service 的特征值主要包含 report map，HID 的 input, output report。这些特征值的内容可以根据不同 HID 设备呈现出不同内容。SDK 里面给出了一个比较全的 HID service 定义，包含了键盘和鼠标的功能。详见 hid_profile_att_table[] 数组的定义。

3.5.1.1 HID service 事件

// HID Device Defines. 定义 hid 设备。 暂时只支持 keyboard 设备。		
HID_DEV_KEYBOARD	0x01	//!< hid 设备定义为键盘
HID_DEV_MOUSE	0x02	//!< hid 设备定义为鼠标
HID_DEV	HID_DEV_KEYBOARD	//!< 由用户定义 hid server 是什么设备。

// HID information flags. 对应 const 变量 hid_info_value,做为 HID_INFORMATION_UUID 读时的返回值		
HID_FLAGS_REMOTE_WAKE	0x01	//!< hid information 标志位：可远程唤醒
HID_FLAGS_NORMALLY_CONNECTABLE	0x02	//!< hid information 标志位：普通 可连接

// HID protocol mode values. 做为 PROTOCOL_MODE_UUID 读时的返回值		
HID_PROTOCOL_MODE_BOOT	0x01	//!< Boot Protocol Mode
HID_PROTOCOL_MODE_REPORT	0x02	//!< Report Protocol Mode

// HID_REPORT_TYPE_GROUP. 对应 hid_report_ref_t 的 report_type 变量值。		
HID_REPORT_TYPE_INPUT	0x1	//!< report id 包含对端可输入的能力
HID_REPORT_TYPE_OUTPUT	0x2	//!< report id 包含对端可输出的能力
HID_REPORT_TYPE_FEATURE	0x3	//!< report id 包含对端可输出/输入的能力
HID_REPORT_TYPE_WR	0x10	//!< report id 包含向对端 写的能力

// HID report mapping table

```
typedef struct
{
  uint8_t     report_id     //!< Report ID。由 report map 定义.
  uint8_t     report_type   //!< Report Type,。定义 report id 的属性。参见@HID_REPORT_TYPE_GROUP

}hid_report_ref_t;
```

3.5.1.2 HID service 函数

void hid_gatt_add_service(void)

创建 hid server 的 profile。

参数：

None

返回：

None

void hid_gatt_report_notify(uint8_t conidx, uint8_t rpt_id, uint8_t *p_data, uint16_t len)

向某个已建立的链接的某个 reprt_info idx 发送一段数据。

参数：

conidx	-	要进行操作的链接号。链接号从 0 开始一直到 app_config,h 定义的最大链接数-1
rpt_id	-	要进行操作的 hid_rpt_info 数组的序号。hid_rpt_info 数组在 hid_gatt_add_service()创建时赋值。内部默认定义如下。 [0] :赋值为 report id 1，对应鼠标。[1] :赋值为 report id 2，对应 Cosumer Controller。[2]：赋值为 report id 3，对应为 键盘输入，[3] 赋值为 report id 3，对应为键盘输出。
p_data	-	指向发送数据缓存地址的指针。
len	-	发送数据的长度。

返回：

None

3.5.2 DIS

Device Information Service。提供设备的产品信息以及软硬件版本。代码位于 components\ble\profiles\ble_dev_info 中。

3.5.2.1 DIS 事件

// Device information marco defines		
DIS_MANUFACTURER_NAME	"Freqchip"	//!< 设备厂商名字
DIS_MANUFACTURER_NAME_LEN	8	//!< 设备厂商名字长度
DIS_MODEL_NB_STR	"Fr-BLE-1.0"	//!< 设备模组名字 string 定义
DIS_MODEL_NB_STR_LEN	10	//!< 设备模组名字长度
DIS_FIRM_REV_STR	"6.1.2"	//!< 设备固件版本定义
DIS_FIRM_REV_STR_LEN	5	//!< 设备固件版本长度
DIS_SYSTEM_ID	"\x12\x34\x56\xFF\xFE\x9A\xBC\xDE"	//!< 设备 System ID 定义
DIS_SYSTEM_ID_LEN	8	//!< 设备 System ID 长度
DIS_SW_REV_STR	"6.3.0"	//!< 设备软件版本定义
DIS_SW_REV_STR_LEN	5	//!< 设备软件版本长度
DIS_PNP_ID	"\x01\x17\x27\xb0\x32\x10\x24"	//!< 设备 PNP_ID 定义
DIS_PNP_ID_LEN	7	//!< 设备 PNP_ID 长度
DIS_SERIAL_NB_STR	"1.0.0.0-LE"	//!< 设备 系列号定义
DIS_SERIAL_NB_STR_LEN	10	//!< 设备 系列号长度
DIS_HARD_REV_STR	"1.0.0"	//!< 设备硬件版本号 string 定义
DIS_HARD_REV_STR_LEN	5	//!< 设备硬件版本号 string 长度
DIS_IEEE	"\xFF\xEE\xDD\xCC\xBB\xAA"	//!< 设备 IEEE 数据定义
DIS_IEEE_LEN	6	//!< 设备 IEEE 数据长度

// Diss server profile contents		
DIS_MANUFACTURER_NAME_CHAR_SUP	0x0001	//!< diss server 包含 厂商名字
DIS_MODEL_NB_STR_CHAR_SUP	0x0002	//!< diss server 包含 模组名字
DIS_SERIAL_NB_STR_CHAR_SUP	0x0004	//!< diss server 包含 系列号名字
DIS_HARD_REV_STR_CHAR_SUP	0x0008	//!< diss server 包含 硬件版本号
DIS_FIRM_REV_STR_CHAR_SUP	0x0010	//!< diss server 包含 固件版本号
DIS_SW_REV_STR_CHAR_SUP	0x0020	//!< diss server 包含 软件版本号
DIS_SYSTEM_ID_CHAR_SUP	0x0040	//!< diss server 包含 System ID
DIS_IEEE_CHAR_SUP	0x0080	//!< diss server 包含 IEEE 数据
DIS_PNP_ID_CHAR_SUP	0x0100	//!< diss server 包含 PNP_ID
DIS_ALL_FEAT_SUP	0x01FF	//!< diss server 包含 上面所有的信息
DIS_FEATURES		//!< 由用户定义 diss server 包含的设备信息内容。

3.5.2.2 DIS 函数

void dis_gatt_add_service(void)

创建 device information 的 profile。

参数：

None

返回：

None

3.5.3 Battery service

Battery service 用于实时电池电量监控。代码位于 components\ble\profiles\ble_batt 中。

3.5.3.1 BATT 事件

// Battery Server Profile attributes index.

```
typedef struct
{
    IDX_BATT_SERVICE                    //!< Batt server primary
    IDX_BATT_LEVEL_CHAR_DECLARATION     //!< Batt server level charact declaration
    IDX_BATT_LEVEL_CHAR_VALUE           //!< Batt server level charact value
    IDX_BATT_LEVEL_CCCD                 //!< Batt server level charact configuration
    IDX_BATT_NB                         //!< Batt server idx numbers.
}hid_report_ref_t;
```

3.5.3.2 BATT 函数

void batt_gatt_add_service(void)

创建 batt server 的 profile。

参数：

None

返回：

None

void batt_gatt_notify(uint8_t conidx,uint8_t batt_level)

更新电量值到内部。如果链接已建立，则立即发送一次电量等级的 ntf 操作。

参数：

conidx - 要进行操作的链接号。链接号从 0 开始一直到 app_config,h 定义的最大链接数-1

batt_level - 要发送的电池电量百分比的值。

返回：

None

3.5.4 OTA

Over The Air download，空中固件升级。代码位于 components\ble\profiles\ble_ota 中。

3.5.4.1 OTA 函数

void ota_gatt_add_service(void)
创建 OTA 服务的 profile。 **参数：** None **返回：** None

4. OSAL API

OSAL 相关 API 位于 components\modules\os\include 目录中。

4.1 Task API

位于 components\modules\os\include\os_task.h。

4.1.1 Task 函数

4.1.1.1 OS Task Create

uint16_t os_task_create(os_task_func_t task_func)

创建一个任务。最多支持 20 个任务。任务不分优先级。消息按抛送的顺序进行处理。

参数：

task_func - 任务的执行函数。

返回：

uint16_t - 创建任务的 id 号，
0xff，任务创建失败。
其他值，任务创建成功，返回值是任务的 id 号。

示例：

```
uint16_t user_task_id;
static int user_task_func(os_event_t *param)
{
    switch(param->event_id)
    {
        case 0
            break;
        case 1
            break;
    }
    return EVT_CONSUMED;
}
void user_task_init(void)
{
    user_task_id = os_task_create(user_task_func);
}
```

4.1.1.2 OS Task Delete

void os_task_delete(uint8_t task_id)

删除一个已经创建的任务。

参数：

task_id - 创建任务时返回的任务 id 号。

返回：

None

4.1.1.3 OS Message Post

void os_msg_post(uint16_t dst_task_id,os_event_t *evt)

向某个已经创建的目标任务抛一个消息事件。

参数：

dst_task_id - 目标任务的任务 id 号

evt - 指向要抛送的消息事件的指针。

返回：

None

4.2 Clock API

位于 components\modules\os\include\os_timer.h。

4.2.1 Clock 函数

4.2.1.1 OS Timer Initialization

void os_timer_init(os_timer_t *ptimer, os_timer_func_t pfunction, void *parg)

初始化一个软件定时器。最多支持 50 个定时器。使用软件定时器之前，必须调用该函数进行初始化。

参数：

ptimer - 指向软件定时器结构体的指针。

pfunction - 定时器的执行函数。

parg - 定时器执行函数的输入参数指针。

返回：

None

示例：

```
os_timer_t test_timer;
static void test_time_fn(void *param)
{
    co_printf("1s timer\r\n");
}
```

```
void user_timer_init(void)
{
    os_timer_init(&test_timer,test_time_fn, NULL);
    os_timer_start(&test_timer,1000, true);  //启动一个 1s 的定时器。
}
```

4.2.1.2 OS Timer Start

void os_timer_start(os_timer_t *ptimer,uint32_t ms, bool repeat_flag)

启动一个软件定时器。

参数：

ptimer	-	指向软件定时器结构体的指针。
ms	-	定时时间，单位:ms。取值范围，10 ~ 0x3FFFFF
repeat_flag	-	定时器是否重复。

返回：

None

4.2.1.3 OS Timer Stop

void os_timer_stop(os_timer_t *ptimer)

停止一个软件定时器。

参数：

ptimer	-	指向软件定时器结构体的指针。

返回：

None

4.3 Memory API

位于 components\modules\os\include\os_mem.h。

4.3.1 Memory 函数

4.3.1.1 OS Malloc

void *os_malloc(uint32_t size)

向系统 heap 申请分配一段内存。

参数：

size	-	要申请的内存的大小。

返回：

void *	-	指向分配内存地址的指针。

4.3.1.2 OS Get Free Heap Size

void os_get_free_heap_size(void)	
获取系统 heap 剩余的空间大小。	
参数：	
None	
返回：	
None	系统剩余的 heap 空间。

4.3.1.3 OS Show Message List

void show_msg_list(void)
打印当前分配的所有消息的信息。只有在 app_config.h 内定义 USER_MEM_API_ENABLE 之后才能使用。
参数：
None
返回：
None

4.3.1.4 OS Show Kernel Malloc Informationi

void show_ke_malloc(void)
打印当前的内存分配统计信息，包含最大 heap 使用量，剩余 heap 和所有 heap 大小。只有在 app_config.h 内定义 USER_MEM_API_ENABLE 之后才能使用。
参数：
None
返回：
None

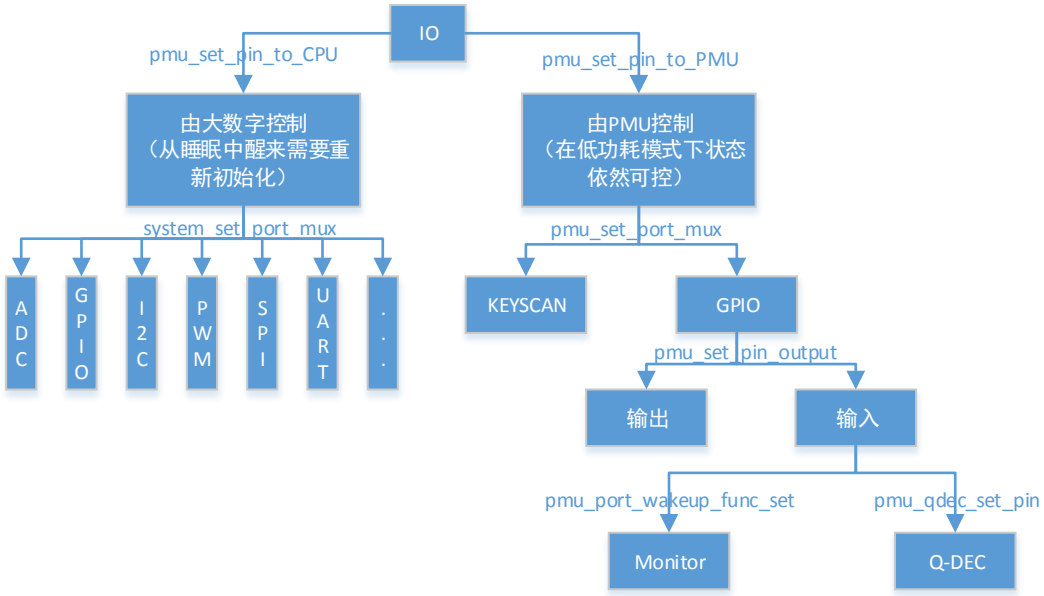
4.3.1.5 OS Show Memory List

void show_mem_list(void)
打印当前所有的内存分配信息。只有在 app_config.h 内定义 USER_MEM_API_ENABLE 之后才能使用。
参数：
None
返回：
None

5. MCU 外设驱动

5.1 IO MUX

FR801xH 系列芯片共有 4 组 IO，每组有 8 路，不同型号的芯片引出的 IO 数量有所不同。每个 IO 可配置为上拉模式，上拉电阻约为 50K 欧姆。IO 的工作状态和模式可选择由大数字（进入低功耗后断电）或者 PMU（进入低功耗模式后继续工作）控制。IO 的控制逻辑和功能配置如下图所示：



IO 逻辑和功能

5.1.1 普通 IO 接口

以下 API 位于 components\driver\include\driver_system.h 中。

5.1.1.1 IO 功能设置

void system_set_port_mux(enum system_port_t port, enum system_port_bit_t bit, uint8_t func)		
设置 IO 由大数字控制时的功能，单次设置一个 IO		
参数：		
port	-	IO 所属的端口组
bit		IO 的 channel 编号
func		所要设置的功能
返回：		
None		
示例：		

```
system_set_port_mux(GPIO_PORT_A, GPIO_BIT_0, PORTA0_FUNC_UART0_RXD);
```

5.1.1.2 IO 上拉设置

void system_set_port_pull(uint32_t port, uint8_t pull)

设置 IO 由大数字控制时的上拉功能，可以一次设置多个 IO

参数：

port	-	IO 所属的端口组
pull		选择是否开启上拉

返回：

None

示例：

```
system_set_port_pull((GPIO_PA0 | GPIO_PA1), true);
```

5.1.2 支持低功耗模式的 IO 接口

以下 API 位于 driver_pmu.h 中，都是在低功耗模式下的 IO 操作。

5.1.2.1 IO 使能低功耗模式

void pmu_set_pin_to_PMU(enum system_port_t port, uint8_t bits)

将某个 pin 脚配置给 pmu 控制。调用 pmu_pwm, pmu_qdec, pmu_gpio 函数前，需要首先调用该函数将对应管脚给 pmu 控制。

参数：

port	-	选择 pin 脚对应的 port 口，一共有 4 个 port 口，PA，PB，PC，PD。参见 enum system_port_t 定义。
bits	-	选择 pin 脚对应的 pin 号码，bit7~bit0 分别代表每个 port 口的 pin7~pin0。每个 bit 位表示该 pin 被选中。

返回：

None

示例：

```
//Select PA0,PA1 to be controlled by PMU
pmu_set_pin_to_PMU(GPIO_PORT_A, GPIO_PA0 | GPIO_PA1);
```

5.1.2.2 IO 关闭低功耗模式

void pmu_set_pin_to_CPU(enum system_port_t port, uint8_t bits)

将某个 pin 脚配置给 CPU 控制。所有管脚默认是被 CPU 控制的。如果希望配置给 PMU 控制的管脚，被 CPU 的外设来控制，需要首先调用该函数将对应管脚给 CPU 控制。

参数：

port	-	选择 pin 脚对应的 port 口，一共有 4 个 port 口，PA，PB，PC，PD。参见 enum system_port_t 定义。
bits	-	选择 pin 脚对应的 pin 号码，bit7~bit0 分别代表每个 port 口的 pin7~pin0。每个 bit 位表示该 pin 被选中。

返回：

None

示例：

```
//Select PA0,PA1 to be controlled by CPU
pmu_set_pin_to_CPU(GPIO_PORT_A, GPIO_PA0 | GPIO_PA1);
```

5.1.2.3 IO 低功耗模式功能设置

void pmu_set_port_mux(enum system_port_t port, uint8_t bit, enum pmu_gpio_mux_t func)

配置某个 pin 脚的 pmu 功能选择。pin 脚对应的 pmu 功能选择由 PMU_IO_MUX 表格决定。

参数：

port - 选择 pin 脚对应的 port 口，一共有 4 个 port 口，PA，PB，PC，PD。参见 enum system_port_t 定义。

bit - 选择 pin 脚对应的 pin 号码，参见 enum system_port_bit_t 定义

func - 选择 pin 脚对应的 pmu 的功能。参见 enum pmu_gpio_mux_t 定义

返回：

None

示例：

```
// configure PB0~PB2 as PMU_PWM function
    pmu_set_port_mux(GPIO_PORT_B,GPIO_BIT_0,PMU_PORT_MUX_PWM);
    pmu_set_port_mux(GPIO_PORT_B,GPIO_BIT_1,PMU_PORT_MUX_PWM);
    pmu_set_port_mux(GPIO_PORT_B,GPIO_BIT_2,PMU_PORT_MUX_PWM);
```

5.1.2.4 IO 低功耗模式输入输出设置

void pmu_set_pin_dir(enum system_port_t port, uint8_t bits, uint8_t dir)

配置某个 pin 脚 pmu 控制时的输入输出选择。

参数：

port - 选择 pin 脚对应的 port 口，一共有 4 个 port 口，PA，PB，PC，PD。参见 enum system_port_t 定义。

bits - 选择 pin 脚对应的 pin 号码，bit7~bit0 分别代表每个 port 口的 pin7~pin0。每个 bit 位表示该 pin 被选中。

dir - 选择 pin 脚对应的输入/输出。只能填以下二值：GPIO_DIR_OUT，表示该 pin 为输出。GPIO_DIR_IN，表示该 pin 为输入。

返回：

None

示例：

```
// configure PA0~PA1 as output
pmu_set_pin_dir(GPIO_PORT_A,BIT(0)|BIT(1), GPIO_DIR_OUT);
```

5.1.2.5 IO 低功耗模式上拉设置

void pmu_set_pin_pull(enum system_port_t port, uint8_t bits, bool flag)

配置某个 pin 脚 pmu 控制时是否内部上拉。

参数：

port - 选择 pin 脚对应的 port 口，一共有 4 个 port 口，PA，PB，PC，PD。参见 enum system_port_t 定义。

bits - 选择 pin 脚对应的 pin 号码，bit7~bit0 分别代表每个 port 口的 pin7~pin0。每个 bit 位表示该 pin 被选中。

flag - 选择 pin 脚是否内部上拉。True，表示该 pin 内部上拉。False，表示该 pin 没有内部上拉。

返回：

None

示例：

```
// configure PA0~PA1 has inner pull
pmu_set_pin_pull(GPIO_PORT_A, GPIO_PA0 | GPIO_PA1, true);
```

5.1.2.6 IO 使能低功耗唤醒

void pmu_port_wakeup_func_set(uint32_t gpios)

设置 PMU 中对 IO 的状态监控功能，该函数内部完成了选择 PMU 控制、IO MUX 选择、设置为输入模式。可以一次设置多个 IO。当被监测的 IO 高低电平发生变化时就可产生 pmu gpio monitor 中断，如果在睡眠状态下发生变化则先产生唤醒信号，同时产生中断。

参数：

gpios - IO 对应的编号

返回：

None

示例：

```
pmu_port_wakeup_func_set(GPIO_PA0|GPIO_PA1);
```

5.1.2.7 IO 低功耗模式中断入口

__attribute__((weak)) void pmu_gpio_isr_ram(void)

pmu_gpio 中断 weak 函数。用于需要重定义来获取中断的入口

参数：

None

返回：

None

示例：

```
void pmu_gpio_isr_ram(void)
{
    uint32_t pmu_int_pin_setting = ool_read32(PMU_REG_PORTA_TRIG_MASK);
    uint32_t gpio_value = ool_read32(PMU_REG_GPIOA_V);

    ool_write32(PMU_REG_PORTA_LAST, gpio_value);
    uint32_t tmp = gpio_value & pmu_int_pin_setting;
    uint32_t pressed_key = tmp^pmu_int_pin_setting;
    co_printf("K:0x%08x\r\n", (pressed_key);
}

void user_entry_after_ble_init(void)
{
```

```
    pmu_port_wakeup_func_set(GPIO_PD5|GPIO_PD4|GPIO_PD3);
}
```

5.2 GPIO

位于 components\driver\include\driver_gpio.h。

5.2.1 普通 GPIO 接口

5.2.1.1 GPIO 输出

void gpio_portX_write(uint8_t value)

设置一组 IO 由大数字控制时的输出值，x 为 a、b、c、d

参数：

value	IO 的输出值

返回：

None

示例：

```
gpio_porta_write(0xFF);
```

5.2.1.2 GPIO 获取当前值

uint8_t gpio_portX_read(void)

获取一组 IO 由大数字控制时的当前值，x 为 a、b、c、d

参数：

None

返回：

uint8_t	-	IO 的当前值

5.2.1.3 GPIO 设置整个 port 输入输出

void gpio_portX_set_dir(uint8_t dir)

设置一组 IO 由大数字控制时的输入输出，x 为 a、b、c、d

参数：

dir	输入输出，每一位对应一个 IO，0：输出；1：输入

返回：

None

5.2.1.4 GPIO 获取整个 port 输入输出配置

uint8_t gpio_portX_get_dir(void)

获取一组 IO 由大数字控制时的输入输出设置，x 为 a、b、c、d

参数：

None

返回：

uint8_t - 当前的输入输出配置

5.2.1.5 GPIO 设置单个 IO 输入输出

void gpio_set_dir(enum system_port_t port, enum system_port_bit_t bit, uint8_t dir)

设置 IO 由大数字控制时的输入输出，一次设置一个 IO

参数：

port - IO 所属的端口组

bit IO 的 channel 编号

dir 输入或者输出

返回：

None

示例：

```
gpio_set_dir(GPIO_PORT_A, GPIO_BIT_0, GPIO_DIR_OUT);
```

5.2.2 低功耗模式 GPIO 接口

5.2.2.1 GPIO 低功耗模式输出值

void pmu_set_gpio_value(enum system_port_t port, uint8_t bits, uint8_t value)

当某个 pin 脚被配置为 pmu gpio 控制，并且是输出模式时，设置该 pin 脚的值。

参数：

port - 选择 pin 脚对应的 port 口，一共有 4 个 port 口，PA，PB，PC，PD。参见 enum system_port_t 定义。

bits - 选择 pin 脚对应的 pin 号码，bit7~bit0 分别代表每个 port 口的 pin7~pin0。每个 bit 位表示该 pin 被选中。

Value - 设置 pin 脚输出值。只能填以下二值：1，该 pin 输出为高。0，该 pin 输出为低。

返回：

None

示例：

```
void pmu_gpio_test(void)
{
    pmu_set_port_mux(GPIO_PORT_A,GPIO_BIT_0,PMU_PORT_MUX_GPIO);
    pmu_set_port_mux(GPIO_PORT_A,GPIO_BIT_1,PMU_PORT_MUX_GPIO);
    pmu_set_pin_to_PMU(GPIO_PORT_A,BIT(0)|BIT(1) );
    pmu_set_pin_dir(GPIO_PORT_A,BIT(0)|BIT(1), GPIO_DIR_OUT);
    pmu_set_pin_pull(GPIO_PORT_A, BIT(0)|BIT(1), true);

    pmu_set_gpio_value(GPIO_PORT_A, BIT(0)|BIT(1), 1);
    co_delay_100us(10);
```

```
    pmu_set_gpio_value(GPIO_PORT_A, BIT(0)|BIT(1), 0);
}
```

5.2.2.2 GPIO 低功耗模式输入值

uint8_t pmu_get_gpio_value(enum system_port_t port, uint8_t bit)

当某个 pin 脚被配置为 pmu gpio 控制，并且是输入模式时，获取该 pin 脚的值。

参数：

port - 选择 pin 脚对应的 port 口，一共有 4 个 port 口，PA，PB，PC，PD。参见 enum system_port_t 定义。

bit - 选择 pin 脚对应的 pin 号码，参见 enum system_port_bit_t 定义

返回：

None

示例：

```
void pmu_gpio_test(void)
{
    pmu_set_port_mux(GPIO_PORT_A,GPIO_BIT_2,PMU_PORT_MUX_GPIO);
    pmu_set_port_mux(GPIO_PORT_A,GPIO_BIT_3,PMU_PORT_MUX_GPIO);
    pmu_set_pin_to_PMU(GPIO_PORT_A,BIT(2)|BIT(3) );
    pmu_set_pin_dir(GPIO_PORT_A,BIT(2)|BIT(3), false);
    co_printf("PA2:%d,PA3:%d\r\n",pmu_get_gpio_value(GPIO_PORT_A,GPIO_BIT_2)
                        ,pmu_get_gpio_value(GPIO_PORT_A,GPIO_BIT_3) );
}
```

5.3 UART

位于 components\driver\include\driver_uart.h。

5.3.1 UART 初始化

void uart_init(uint32_t uart_addr, uint8_t bandrate)

初始化 UART 模块，函数内会清空 fifo，使能接收非空中断

参数：

uart_addr - 初始化的端口，可选 UART0、UART1

bandrate - 配置的波特率，例如 BAUD_RATE_115200

返回：

None -

5.3.2 UART 等待发送 FIFO 为空

void uart_finish_transfers(uint32_t uart_addr)

等待发送 fifo 为空

参数：

uart_addr - 等待的端口，可选 UART0、UART1

返回：

None -

5.3.3 从串口读取数据

void uart_read(uint32_t uart_addr, uint8_t *buf, uint32_t size)

采用阻塞方式从串口接收

参数：

uart_addr - 等待的端口，可选 UART0、UART1

buf - 存放接收数据的指针

size - 需要接收的数据长度

返回：

None -

示例：

5.3.4 从串口发送数据

void uart_write(uint32_t uart_addr, const uint8_t *bufptr, uint32_t size)

采用阻塞方式发送数据到串口

参数：

uart_addr - 等待的端口，可选 UART0、UART1

buf - 待发送数据的指针

size - 待发送的数据长度

返回：

None -

示例：

5.3.5 UART 发送一个字节且等待完成

void uart_putc_noint(uint32_t uart_addr, uint8_t c)

发送一个字符，且等待发送 fifo 为空

参数：

uart_addr - 使用的端口，可选 UART0、UART1

c - 待发送的字符

返回：

None -

5.3.6 UART 发送一个字节且立即返回

void uart_putc_noint_no_wait(uint32_t uart_addr, uint8_t c)

将待发送的字符写入到发送 fifo

参数：

uart_addr	-	使用的端口，可选 UART0、UART1
c	-	待发送的字符

返回：

None	-	

5.3.7 UART 发送多个字节且等待完成

void uart_put_data_noint(uint32_t uart_addr, const uint8_t *d, int size)

发送特定长度的字符，且等待发送 fifo 为空

参数：

uart_addr	-	使用的端口，可选 UART0、UART1
d	-	待发送数据的保存地址指针
size	-	待发送数据的长度

返回：

None	-	

5.3.8 UART 读取特定个数字节

void uart_get_data_noint(uint32_t uart_addr, uint8_t *buf, int size)

获取特定长度的字符

参数：

uart_addr	-	使用的端口，可选 UART0、UART1
buf	-	待接收数据的保存地址指针
size	-	待接收数据的长度

返回：

None	-	

5.3.9 UART 读取特定个数字节，诺 FIFO 为空则先返回

int uart_get_data_nodelay_noint(uint32_t uart_addr, uint8_t *buf, int size)

获取特定长度的字符，当接收 fifo 为空时就会返回

参数：

uart_addr	-	使用的端口，可选 UART0、UART1
buf	-	待接收数据的保存地址指针
size	-	待接收数据的长度

返回：

int	-	实际接收到的数据长度

5.3.10 UART0 读数据

void uart0_read(uint8_t *bufptr, uint32_t size, uart_int_callback callback)

采用中断加回调函数的方式从 UART0 中读取特定长度的数据，需配合内置的 uart0_isr 使用（也就是无需重新实现 UART0 的中断处理函数）。

参数：

bufptr - 待接收数据的保存地址指针

size - 待接收数据的长度

callback - 接收完数据的回调函数

返回：

None

5.3.11 UART0 发数据

void uart0_write(uint8_t *bufptr, uint32_t size, uart_int_callback callback)

采用中断加回调函数的方式用 UART0 中发送特定长度的数据，需配合内置的 uart0_isr 使用（也就是无需重新实现 UART0 的中断处理函数）。

参数：

bufptr - 待发送数据的保存地址指针

size - 待发送数据的长度

callback - 发送完数据的回调函数

返回：

None

5.3.12 UART1 读数据

void uart1_read(uint8_t *bufptr, uint32_t size, uart_int_callback callback)

采用中断加回调函数的方式从 UART0 中读取特定长度的数据，需配合内置的 uart1_isr 使用（也就是无需重新实现 UART1 的中断处理函数）。

参数：

bufptr - 待接收数据的保存地址指针

size - 待接收数据的长度

callback - 接收完数据的回调函数

返回：

None

5.3.13 UART1 写数据

void uart1_write(uint8_t *bufptr, uint32_t size, uart_int_callback callback)

采用中断加回调函数的方式用 UART1 中发送特定长度的数据，需配合内置的 uart1_isr 使用（也就是无需重新实现 UART0 的中断处理函数）。

参数：

bufptr - 待发送数据的保存地址指针
size - 待发送数据的长度
callback - 发送完数据的回调函数

返回：

None

5.4 SPI

位于 components\driver\include\driver_ssp.h。

5.4.1 SPI 初始化

void ssp_init_(uint8_t bit_width, uint8_t frame_type, uint8_t ms, uint32_t bit_rate, uint8_t prescale, void (*ssp_cs_ctrl)(uint8_t))

初始化 SPI 模块

参数：

bit_width - 总线上的数据位宽，取值为 1~8
frame_type - SPI 总线类型，可取值 SSP_FRAME_MOTO、SSP_FRAME_SS、SSP_FRAME_NATTIONAL_M
ms - 主从模式选择，可取值 SSP_MASTER_MODE、SSP_SLAVE_MODE
bit_rate - 需要配置的总线速率
prescale - 基于系统时钟的模块分频比
ssp_cs_ctrl - 自定义的 CS 控制函数

返回：

None

示例：

```
void ssp_cs_ctrl_function(uint8_t op)
{
    if(op == SSP_CS_ENABLE)
    {
        gpio_porta_write(gpio_porta_read() & 0xDF);
    }
    else
    {
        gpio_porta_write(gpio_porta_read() | 0x20);
    }
}
ssp_init_(8, SSP_FRAME_MOTO, SSP_MASTER_MODE, 24000000, 2, ssp_cs_ctrl_function);
```

5.4.2 SPI 发送并接收

void ssp_send_then_recv(uint8_t* tx_buffer, uint32_t n_tx, uint8_t* rx_buffer, uint32_t n_rx)

工作在主模式时，执行先发送一定长度数据，然后接收一定长度数据的操作

参数：

tx_buffer	-	待发送数据的保存地址指针
n_tx	-	待发送数据的长度
rx_buffer	-	待接收数据的保存地址指针
n_rx	-	待接收数据的长度

返回：

None

5.4.3 SPI 发送

void ssp_send_data(uint8_t *buffer, uint32_t length)

工作在主模式时，发送一定长度数据

参数：

buffer	-	待接收数据的保存地址指针
length	-	待接收数据的长度

返回：

None

5.4.4 SPI 接收

void ssp_recv_data(uint8_t *buffer, uint32_t length)

工作在主模式时，接收一定长度数据

参数：

buffer	-	待接收数据的保存地址指针
length	-	待接收数据的长度

返回：

None

5.5 I2C

位于 components\driver\include\driver_iic.h。

5.5.1 I2C 初始化

void iic_init(enum iic_channel_t channel, uint16_t speed, uint16_t slave_addr)

初始化 IIC 模块，默认配置为为 7 位地址模式

参数：

channel	-	初始化对象，可选 IIC_CHANNEL_0、IIC_CHANNEL_1
speed	-	配置总线时钟速率为 speed*1000
slave_addr	-	当本机工作在从机模式时的从机地址

返回：

None

5.5.2 I2C 发送一个字节

uint8_t iic_write_byte(enum iic_channel_t channel, uint8_t slave_addr, uint8_t reg_addr, uint8_t data)

将一个字节数据发送给从机的特定地址

参数：

channel	-	操作对象
slave_addr	-	从机地址
reg_addr	-	操作的从机寄存器地址
data	-	待写入的数据

返回：

uint8_t	-	true：写入成功；false：写入失败

5.5.3 I2C 发送多个字节

uint8_t iic_write_bytes(enum iic_channel_t channel, uint8_t slave_addr, uint8_t reg_addr, uint8_t *buffer, uint16_t length)

将多个字节数据发送给从机的特定地址

参数：

channel	-	操作对象
slave_addr	-	从机地址
reg_addr	-	操作的从机寄存器起始地址
buffer	-	待写入的数据
length	-	待写入的数据长度

返回：

uint8_t	-	true：写入成功；false：写入失败

5.5.4 I2C 读取一个字节

uint8_t iic_read_byte(enum iic_channel_t channel, uint8_t slave_addr, uint8_t reg_addr, uint8_t *buffer)

从从机的特定地址读取一个字节数据

参数：

channel	-	操作对象
slave_addr	-	从机地址
reg_addr	-	操作的从机寄存器地址
buffer	-	读取数据的保存地址

返回：

uint8_t	-	true：读取成功；false：读取失败

5.5.5 I2C 读取多个字节

uint8_t iic_read_bytes(enum iic_channel_t channel, uint8_t slave_addr, uint8_t reg_addr, uint8_t *buffer, uint16_t length)

从从机的特定地址读取多个字节数据

参数：

channel	-	操作对象
slave_addr	-	从机地址
reg_addr	-	操作的从机寄存器起始地址
buffer	-	读取数据的保存地址
length	-	待读取的数据长度

返回：

uint8_t	-	true：读取成功；false：读取失败

5.6 Timer

位于 components\driver\include\driver_timer.h。

5.6.1 Timer 初始化

uint8_t timer_init(uint32_t timer_addr, uint32_t count_us, uint8_t run_mode)

初始化 timer 模块

参数：

timer_addr	-	初始化对象，可选 TIMER0、TIMER1
count_us	-	定时器周期
run_mode	-	单次模式还是周期模式

返回：

None	-	

5.6.2 Timer 启动

void timer_run(uint32_t timer_addr)

开启一个已经初始化好的 timer

参数：

timer_addr	-	操作对象

返回：

None

5.6.3 Timer 停止

void timer_stop(uint32_t timer_addr)

停止一个已经开启的 timer

参数：

timer_addr - 操作对象

返回：

None

5.6.4 Timer 获取 load 值

uint32_t timer_get_load_value(uint32_t timer_addr)

获取当前 timer 的预设计时值

参数：

timer_addr - 操作对象

返回：

uint32_t - timer 的预设值

5.6.5 Timer 获取当前计数值

uint32_t timer_get_current_value(uint32_t timer_addr)

获取当前 timer 的计数值

参数：

timer_addr - 操作对象

返回：

uint32_t - 当天计数值

5.6.6 Timer 清中断

void timer_clear_interrupt(uint32_t timer_addr)

清除中断标志位

参数：

timer_addr - 操作对象

返回：

None

5.7 PWM

位于 components\driver\include\driver_pwm.h。

5.7.1 普通 PWM 接口

5.7.1.1 PWM 初始化

void pwm_init(enum pwm_channel_t channel, uint32_t frequency, uint8_t high_duty)

初始化 pwm。调用任何 pwm 函数前，需要首先调用 该函数。

参数：

channel - 初始化对象，可选 PWM_CHANNEL_0 等

frequency - 设置 PWM 的工作频率

high_duty - 设置 PWM 高电平所占比例，取值为 0~99

返回：

None

5.7.1.2 PWM 启动

void pwm_start(enum pwm_channel_t channel)

开启一个已经初始化好的 PWM

参数：

channel - 操作对象

返回：

None

5.7.1.3 PWM 停止

void pwm_stop(enum pwm_channel_t channel)

停止一个已经开始工作的 PWM

参数：

channel - 操作对象

返回：

None

5.7.1.4 PWM 更新参数

void pwm_update(enum pwm_channel_t channel, uint32_t frequency, uint8_t high_duty)

更新一个正在运行的 PWM 的参数

参数：

channel - 更新对象，可选 PWM_CHANNEL_0 等

frequency - 设置 PWM 的工作频率

high_duty - 设置 PWM 高电平所占比例，取值为 0~99

返回：

None

5.7.2 低功耗模式 PWM 接口

5.7.2.1 低功耗 PWM 初始化

void pmu_pwm_init(void)

初始化 pmu_pwm。调用任何 pmu_pwm 函数前，需要首先调用 该函数。

参数：

None

返回：

None

5.7.2.2 低功耗 PWM 设置参数

void pmu_pwm_set_param(enum system_port_t port, enum system_port_bit_t bit,uint16_t high_count,uint16_t low_count)

设置 pmu_pwm 的周期参数。需要在 pmu_pwm_init()之后调用。pin 脚对应的 pmu_pwm 功能选择由 PMU_IO_MUX 表格决定。

参数：

port	-	pwm 对应的 port 口，参见 enum system_port_t 定义。
bit	-	pwm 对应的 pin 脚号，参见 enum system_port_bit_t 定义。
high_count	-	pwm 在一个周期内输出高电平的持续时间，单位：pmu 时钟周期。
low_count	-	pwm 在一个周期内输出低电平的持续时间，单位：pmu 时钟周期。Pmu 时钟选择由函数 void pmu_set_lp_clk_src(enum pmu_lp_clk_src_t src) 决定。

返回：

None

5.7.2.3 低功耗 PWM 启动

void pmu_pwm_start(enum system_port_t port, enum system_port_bit_t bit,bool repeat_flag,bool reverse_flag)

启动 pmu_pwm。pmu_pwm 在 sleep 情况下，依然能保持运行。pin 脚对应的 pmu_pwm 功能选择由 PMU_IO_MUX 表格决定。

参数：

port	-	pwm 对应的 port 口，参见 enum system_port_t 定义
bit	-	pwm 对应的 pin 号码，参见 enum system_port_bit_t 定义
repeat_flag	-	pwm 是否循环发送标志位。 True - pwm 会循环。 False -pwm 启动后只发送一个周期
reverse_flag	-	pwm 电平是否会翻转发送。True -pwm 从低电平开始。 False -pwm 从高电平开始

返回：

None

示例：

```
    pmu_pwm_init();
// configure PB0~PB2 is controlled by PMU
    pmu_set_pin_to_PMU(GPIO_PORT_B, BIT(0)|BIT(1)|BIT(2));
// configure PB0~PB2 as PMU ouput
    pmu_set_pin_dir(GPIO_PORT_B, BIT(0)|BIT(1)|BIT(2),true);
// configure PB0~PB2 as PMU_PWM function
    pmu_set_port_mux(GPIO_PORT_B,GPIO_BIT_0,PMU_PORT_MUX_PWM);
    pmu_set_port_mux(GPIO_PORT_B,GPIO_BIT_1,PMU_PORT_MUX_PWM);
    pmu_set_port_mux(GPIO_PORT_B,GPIO_BIT_2,PMU_PORT_MUX_PWM);
// set PMU_PWM high count/low count in one period
    pmu_pwm_set_param(GPIO_PORT_B,GPIO_BIT_0,20,10);
    pmu_pwm_set_param(GPIO_PORT_B,GPIO_BIT_1,20,10);
    pmu_pwm_set_param(GPIO_PORT_B,GPIO_BIT_2,20,10);
// start PB0~PB2 PMU_PWM.
    pmu_pwm_start(GPIO_PORT_B,GPIO_BIT_0,1,0);
    pmu_pwm_start(GPIO_PORT_B,GPIO_BIT_1,0,0);
    pmu_pwm_start(GPIO_PORT_B,GPIO_BIT_2,1,1);
    co_delay_100us(10000);
// stop PB0~PB2 PMU_PWM.
    pmu_pwm_stop(GPIO_PORT_B,GPIO_BIT_0);
    pmu_pwm_stop(GPIO_PORT_B,GPIO_BIT_1);
    pmu_pwm_stop(GPIO_PORT_B,GPIO_BIT_2);
```

5.7.2.4 低功耗 PWM 停止

void pmu_pwm_stop(enum system_port_t port, enum system_port_bit_t bit)

停止 pmu_pwm。pin 脚对应的 pmu_pwm 功能选择由 PMU_IO_MUX 表格决定。

参数：

port	-	pwm 对应的 port 口，参见 enum system_port_t 定义。
bit	-	pwm 对应的 pin 脚号，参见 enum system_port_bit_t 定义。

返回：

None

5.8 ADC

位于 components\driver\include\driver_adc.h。

5.8.1 ADC 初始化

void adc_init(struct adc_cfg_t *cfg)

初始化 ADC 模块

参数：

cfg	-	初始化 ADC 的参数，详见 adc_cfg_t 的定义

返回：

None

5.8.2 ADC 开始采样

bool adc_enable(void (*callback)(uint16_t *, uint32_t), uint16_t *buffer, uint32_t length)

ADC 模块初始化之后，就可以调用 adc_enable()进行采样

参数：

callback	-	如果 callback 不为 NULL，采样结束后就会以回调方式返回采样结果。如果 callback 为 NULL，那么采样结束后就以中断模式返回，采样结果需要调用 adc_get_result()来获取。回调方式只能用于 fixed channel 采样模式，不然该函数会返回失败
buffer	-	用于存放采样结果的 buffer
length	-	需要采样的数据长度

返回：

bool	-	使能的结果，成功或失败。

5.8.3 ADC 停止采样

void adc_disable(void)

停止正在采样的 ADC

参数：

None

返回：

None

5.8.4 ADC 读取结果

void adc_get_result(enum adc_trans_source_t src, uint8_t channels, uint16_t *buffer)

获取最近一次的 ADC 采样结果

参数：

src	-	ADC 采集的数据源，详见 adc_trans_source_t 定义
channels	-	ADC 通道
buffer	-	用于存放采样结果的内存空间，顺序摆放

返回：

None

5.9 WDT

位于 components\driver\include\driver_wdt.h。

注意：WDT 默认就支持低功耗模式。

5.9.1 WDT 初始化

void wdt_init(enum wdt_action action, uint8_t delay_s)

初始化 pmu_watchdog。调用任何 pmu_wdt 函数前，需要首先调用 该函数。

参数：

action	-	watch 超时后的行为选择。WDT_ACT_RST_CHIP，直接重启芯片。WDT_ACT_CALL_IRQ，产生中断服务。
delay_s	-	watchdog 触发超时时间。单位：秒。范围：0~0xff

返回：

None

5.9.2 WDT 喂狗

void wdt_feed(void)

给 wdt 喂狗。Wdt 超时时间之内没有喂狗，则产生 wdt 超时行为。

参数：

None

返回：

None

5.9.3 WDT 启动

void wdt_start(void)

启动 wdt。超时时间清 0。

参数：

None

返回：

None

5.9.4 WDT 停止

void wdt_stop(void)

停止 wdt。

参数：

None

返回：

None

5.9.5 WDT 中断处理接口

__attribute__((weak)) void wdt_isr_ram(unsigned int* hardfault_args)

pmu_watchdog 中断 weak 函数。用于需要重定义来获取中断的入口

参数：

hardfault_args - 传入的栈顶指针，用于 debug 产生 watch 的原因。

返回：

None

示例：

```
void wdt_isr_ram(unsigned int* hardfault_args)
{
    co_printf("wdt_rest\r\n\r\n");
    co_printf("PC      = 0x%.8X\r\n",hardfault_args[6]);
    co_printf("LR      = 0x%.8X\r\n",hardfault_args[5]);
    co_printf("R0      = 0x%.8X\r\n",hardfault_args[0]);
    co_printf("R1      = 0x%.8X\r\n",hardfault_args[1]);
    co_printf("R2      = 0x%.8X\r\n",hardfault_args[2]);
    co_printf("R3      = 0x%.8X\r\n",hardfault_args[3]);
    co_printf("R12     = 0x%.8X\r\n",hardfault_args[4]);

    /* reset the system */
    ool_write(PMU_REG_RST_CTRL, ool_read(PMU_REG_RST_CTRL) & (~ PMU_RST_WDT_EN) );
}

void wdt_test(void)
{
    wdt_init(WDT_ACT_CALL_IRQ, 4);
    wdt_start();
}
```

5.10 RTC

位于 components\driver\include\driver_rtc.h。
注意：RTC 默认支持低功耗模式。

5.10.1 RTC 初始化

void rtc_init(void)

初始化 pmu_rtc。调用任何 pmu_rtc 函数前，需要首先调用 该函数。

参数：

None

返回：

None

5.10.2 RTC 启动

void rtc_alarm(enum rtc_idx_t rtc_idx, uint32_t count_ms)

启动某个 rtc 定时器。

参数：

rtc_idx - 要启动的 pmu_rtc。@ref enum rtc_idx_t

count_ms - 要启动的 pmu_rtc 的定时时间。单位 ms。范围 1~4294967

返回：

None

5.10.3 RTC 停止

void rtc_disalarm(enum rtc_idx_t rtc_idx)

停止某个 rtc 定时器。

参数：

rtc_idx - 要启动的 pmu_rtc。@ref enum rtc_idx_t

返回：

None

5.10.4 RTC 中断处理接口

__attribute__((weak)) void rtc_isr_ram(uint8_t rtc_idx)

pmu_rtc 中断 weak 函数。用于需要重定义来获取中断的入口

参数：

rtc_idx - RTC 中断源。RTC_A，RTC_B 分别表示 2 个 RTC 的中断。

返回：

None

示例：

```
void rtc_isr_ram(uint8_t rtc_idx)
{
    if(rtc_idx == RTC_A)
    {

    }
    if(rtc_idx == RTC_B)
    {

    }
}

void rtc_test(void)
{
```

```
    rtc_timer_init();
    rtc_timer_arm(RTC_A,150);          //RTC_A, 150us
    rtc_timer_arm(RTC_B,150);          //RTC_A, 150us
}
```

5.11 QDEC

位于 components\driver\include\driver_qdec.h。
注意：QDEC（旋转编码器）默认支持低功耗模式。

5.11.1 QDEC 初始化

void pmu_qdec_init(void)

初始化 pmu_qdec。调用任何 pmu_qdec 函数前，需要首先调用 该函数。

参数：

None

返回：

None

5.11.2 QDEC 设置引脚

void pmu_qdec_set_pin(enum pmu_qdec_la_pin_t pin_a,enum pmu_qdec_lb_pin_t pin_b,enum pmu_qdec_lc_pin_t pin_c)

设置 pmu_qdec LA，LB，LC 分配到哪个 pin 脚。参见 PMU_IO_MUX 表格。

参数：

pin_a - pmu_qdec 功能 LA 对应的 pin 脚选择。@ref enum pmu_qdec_la_pin_t
pin_b - pmu_qdec 功能 LB 对应的 pin 脚选择。@ref enum pmu_qdec_lb_pin_t
pin_c - pmu_qdec 功能 LC 对应的 pin 脚选择。@ref enum pmu_qdec_lc_pin_t

返回：

None

5.11.3 QDEC 设置清零条件

void pmu_qdec_autorest_cnt_flag(bool flag_read_rest,bool flag_lc_rest)

设置 pmu_qdec 某个旋转方向的计数值的清 0 行为。该计数值大于中断阈值时，产生 Qdec 的中断。

参数：

flag_read_rest - 是否读 cnt 清 0 的标志位。 True，读完旋转 cnt，清 cnt。 False，读完旋转 cnt，不清 cnt

flag_lc_rest - 是否 LC 脚低电平清所有 cnt 值。True，LC 脚低电平，一直清 cnt。False，LC 脚低电平不清 cnt

返回：

None

5.11.4 QDEC 设置中断阀值

void pmu_qdec_set_threshold(uint8_t threshold)

设置 pmu_qdec 产生中断的阈值。当某个方向的旋转计数值大于该值时，产生中断。

参数：

threshold - qdec 旋转计数产生中断的阈值。范围 0~0xff

返回：

None

5.11.5 QDEC 设置中断类型

void pmu_qdec_set_irq_type(enum pmu_qdec_irq_type irq_type)

设置 pmu_qdec 产生中断的类型。

参数：

irq_type - 产生中断的类型。PMU_ISR_QDEC_SINGLE_EN，如果旋转计数值大于 0，就产生中断。
PMU_ISR_QDEC_MULTI_EN，如果旋转计数值大于中断阈值，就产生中断。

返回：

None

5.11.6 QDEC 去抖

void pmu_qdec_set_debounce_cnt(uint8_t cnt)

设置 pmu_qdec 产生旋转计数值的防抖时间。

参数：

cnt - 防抖设置时间，范围 0~0xff。Pmu 防抖检测周期 T 等于 pmu 系统时钟周期，pmu_get_rc_clk(false) 返回值为 pmu 系统时钟频率。Qdec 旋转计数防抖时间 = T * (2+cnt)。

返回：

None

5.11.7 QDEC 读取旋转计数

uint8_t pmu_qdec_get_cnt(enum pmu_qdec_direction dir)

读取 pmu_qdec 当前两方向的旋转计数值。

参数：

dir - 要获取旋转计数值的方向。DIR_A，左旋转。DIR_B，右旋转。

返回：

uint8_t - 旋转计数值。范围 0~0xff

5.11.8 QDEC 中断处理接口

__attribute__((weak)) void qdec_isr_ram(void)

pmu_qdec 中断 weak 函数。用于需要重定义来获取中断的入口

参数：

None

返回：

None

示例：

```
void qdec_isr_ram(void)
{
    co_printf("%x,%x\r\n",pmu_qdec_get_cnt(QDEC_DIR_A),pmu_qdec_get_cnt(QDEC_DIR_B));
}
void qdec_test(void)
{
//set qdec pmu pin configuration
    pmu_set_pin_to_PMU(GPIO_PORT_B,BIT(0)|BIT(1)|BIT(2));
    pmu_set_pin_dir(GPIO_PORT_B,BIT(0)|BIT(1)|BIT(2),false );
    pmu_port_set_mux(GPIO_PORT_B, GPIO_BIT_0, PMU_PORT_MUX_QDEC);
    pmu_port_set_mux(GPIO_PORT_B, GPIO_BIT_1, PMU_PORT_MUX_QDEC);
    pmu_port_set_mux(GPIO_PORT_B, GPIO_BIT_2, PMU_PORT_MUX_QDEC);

    pmu_qdec_init();
//set qdec LA,LB,LC pin mapping
    pmu_qdec_set_pin(PMU_QDEC_LA_PIN_PB0,PMU_QDEC_LB_PIN_PB1,PMU_QDEC_LC_PIN_PB2);
//set qdec rotation cnt clear type
    pmu_qdec_autorest_cnt_flag(true,false);
//set qdec rotation cnt threshold to generat IRQ
    pmu_qdec_set_threshold(5);
//set qdec IRQ generation type
    pmu_qdec_set_irq_type(PMU_ISR_QDEC_SINGLE_EN);
//set qdec rotation cnt generation anti-shake check time
    pmu_qdec_set_debounce_cnt(18);
}
```

5.12 Key Scan

位于 components\driver\include\driver_keyscan.h 中。
注意：Key Scan 默认支持低功耗模式。

5.12.1 Key Scan 结构体定义

5.12.1.1 Key Scan 参数

// KeyScan parameter structer

```
typedef struct
{
    uint8_t     row_en          键盘扫描行 pin 脚选择。Bit7~Bit0 分别映射管脚{PD[7:0]}射。 每个 bit 置位，代表使能该管脚。
    uint32_t    col_en          键盘扫描列 pin 脚选择。Bit19~Bit0 分别映射管脚{PC[3:0]，PB[7:0], PA[7:0]}，每个 bit 置位，代
                                表使能该管脚。
    uint8_t     row_map_sel     目前没用，预留。
}keyscan_param_t;
```

5.12.2 Key Scan 函数

5.12.2.1 Key Scan 初始化

void keyscan_init(keyscan_param_t *param)

初始化并使能 pmu_keyscan。pin 脚对应的 pmu_keyscan 功能选择由 PMU_IO_MUX 表格决定。

参数：

param - 键盘扫描的参数设置。@ref keyscan_param_t

返回：

None

5.12.2.2 Key Scan 中断处理接口

__attribute__((weak)) void keyscan_isr_ram(void)

pmu_keyscan 中断 weak 函数。用于需要重定义来获取中断的入口

参数：

None

返回：

None

示例：

```
void keyscan_isr_ram(void)
{
```

```
    uint32_t value;
    uint8_t reg = PMU_REG_KEYSCAN_STATUS_0;
    for(uint8_t j=0; j<5; j++)
    {
        value = ool_read32(reg + (j<<2));
        if(value)
        {
            LOG_INFO("grp[%d]:0x%08x.\r\n", j, value);
        }
    }
}

void keyscan_test(void)
{
    keyscan_param_t param;
//Row is {PD[7:0]}
    param.row_en = 0xff;
//Col is {PC[3:0]，PB[7:0], PA[7:0]}
    param.col_en = 0xfffff;
    param.row_map_sel = 0x0;
    keyscan_init(&param);
}
```

5.13 PMU

位于 components\driver\include\driver_pmu 中。

5.13.1 PMU 配置系统电源

void pmu_set_sys_power_mode(enum pmu_sys_pow_mode_t mode)

配置系统的供电源。

参数：

mode - 设置系统的供电。只能填以下二值：PMU_SYS_POW_BUCK，系统由 BUCK 供电。PMU_SYS_POW_LDO，系统由 LDO 供电。

返回：

None

5.13.2 PMU 判断系统是否第一次上电

uint8_t pmu_first_power_on(uint8_t clear)

用来获取系统启动是否是第一次上电。

参数：

None

返回：

uint8_t - 系统启动是否因为第一次上电。0，不是。1，是。

5.13.3 PMU 使能中断

void pmu_enable_irq(uint16_t irqs)

使能 pmu 中断。

参数：

irqs - 选择需要使能哪些 pmu 中断，每个 bit 位代表 enum pmu_isr_enable_t 定义的一种中断。

返回：

None

示例：

```
void user_entry_before_ble_init(void)
{
    /* set system power supply in BUCK mode */
    pmu_set_sys_power_mode(PMU_SYS_POW_BUCK);

    /* enable pmu_isr: charge plug in/out; LVD, bat_full, over-temperature protect*/
    pmu_enable_irq(PMU_ISR_BIT_ACOK
                   | PMU_ISR_BIT_ACOFF
                   | PMU_ISR_BIT_OTP
                   | PMU_ISR_BIT_LVD
                   | PMU_ISR_BIT_BAT);
    NVIC_EnableIRQ(PMU_IRQn);
}
```

5.13.4 PMU 关闭中断

void pmu_disable_irq(uint16_t irqs)

禁止 pmu 中断。

参数：

irqs - 选择需要禁止哪些 pmu 中断，每个 bit 位代表 enum pmu_isr_enable_t 定义的一种中断。

返回：

None

5.13.5 PMU 使能 Codec 供电

void pmu_codec_power_enable(void)

使能 pmu 给音频 codec 模块供电

参数：

None

返回：

None

5.13.6 PMU 关闭 Codec 供电

void pmu_codec_power_disable(void)

禁止 pmu 给音频 codec 模块供电

参数：

None

返回：

None

5.13.7 PMU 设置 LDO_OUT 和 IO 电压值

void pmu_set_aldo_voltage(enum pmu_aldo_work_mode_t mode, uint8_t value)

设置 LDO_OUT 和 pin 脚高电平 时的电压值。

参数：

mode - 设置 LDO 输出等于 VBAT，还是由 value 值决定。参见 enum pmu_aldo_work_mode_t。PMU_ALDO_MODE_BYPASS, 表示 LDO 输出等于 VBAT。PMU_ALDO_MODE_NORMAL, 表示 LDO 输出由 value 值决定。

value - LDO 输出由 Value 值决定时，设置 LDO 输出的具体电压值。 待定。。。

返回：

None

5.13.8 PMU 设置 32K 时钟源

void pmu_set_lp_clk_src(enum pmu_lp_clk_src_t src)

设置低功耗模式下的时钟源选择，同时也是 pmu 功能逻辑的时钟源。

参数：

src - 设置 pmu 和低功耗模式时钟源，参见 enum pmu_lp_clk_src_t。PMU_LP_CLK_SRC_EX_32768，时钟源是外部 32768 晶振提供。PMU_LP_CLK_SRC_IN_RC，时钟源是内部 RC 电路提供。

返回：

None

5.13.9 PMU 设置内部 RC 频率

uint32_t pmu_get_rc_clk(uint8_t redo)

获取内部 rc 的时钟频率。

参数：

redo - 是否发起一次 rc 时钟的校准操作。True，发起一次校准计算，并获取 rc 时钟频率，False，不发起校准计算，获取上次的 rc 时钟频率校准值。

返回：

uint32_t - 内部 rc 时钟频率。单位：hz

5.13.10 PMU Charger 中断接口

__attribute__((weak)) void charge_isr_ram(uint8_t type)

充电插拔，充满的中断 weak 函数。用于需要重定义来获取中断的入口

参数：

type - 产生中断的原因。只能为 3 个值。2：充满电，1：充电拔出，0：充电插入。

返回：

None

示例：

```
void charge_isr_ram(uint8_t type)
{
    if(type == 2)
    {
        co_printf("charge full\r\n");
        pmu_disable_isr(PMU_ISR_BAT_EN);
    }
    else if(type == 1)
        co_printf("charge out\r\n");
    else if(type == 0)
    {
        pmu_enable_isr(PMU_ISR_BAT_EN);
        co_printf("charge in\r\n");
    }
}

void user_entry_before_ble_init(void)
{
    pmu_enable_irq(PMU_ISR_BIT_ACOK
                   | PMU_ISR_BIT_ACOF
                   | PMU_ISR_BIT_BAT);
    NVIC_EnableIRQ(PMU_IRQn);
}
```

5.13.11 PMU 低电压监测中断接口

__attribute__((weak)) void lvd_isr_ram(void)

lvd 低电压检测中断 weak 函数。用于需要重定义来获取中断的入口

参数：

None

返回：

None

示例：

```
void lvd_isr_ram(void)
{
    co_printf("lvd\r\n");
    pmu_disable_isr(PMU_ISR_LVD_EN);
}

void user_entry_before_ble_init(void)
{
    pmu_enable_irq( PMU_ISR_BIT_LVD);
    NVIC_EnableIRQ(PMU_IRQn);
}
```

5.13.12 PMU 高温监测中断接口

__attribute__((weak)) void otd_isr_ram(void)

高温检测中断 weak 函数。用于需要重定义来获取中断的入口

参数：

None

返回：

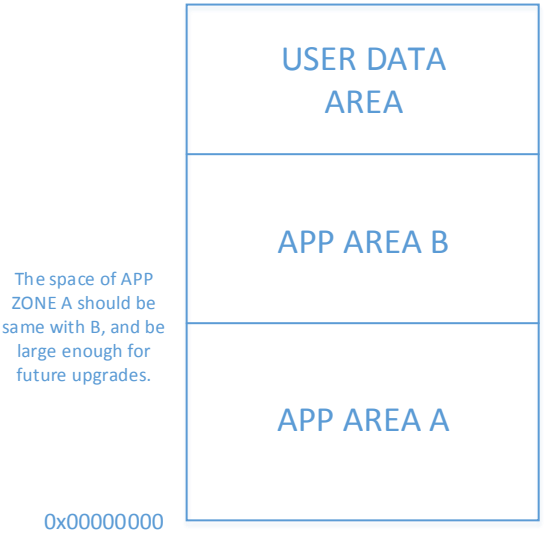
None

示例：

```
void otd_isr_ram(void)
{
    co_printf("otd\r\n");
    pmu_disable_isr(PMU_ISR_OTP_EN);
}

void user_entry_before_ble_init(void)
{
    pmu_enable_irq(PMU_ISR_BIT_OTP);
    NVIC_EnableIRQ(PMU_IRQn);
}
```

6. OTA

在 FR801xH 的 SDK 中集成了一套完整的 OTA profile，用户可以基于此开发手机应用程序等 OTA 主机。FR801xH 采用了双备份的方式进行固件的存储，boot 程序通过固件版本号的方式区分启动后采用哪一块作为运行对象。



6.1 OTA profile

该 profile 的定义如下：

Service UUID：0xFE00

Write attribute UUID: 0xFF01

Notify attribute UUID: 0xFF02

Write attribute 用于 FR801xH 接收来自 OTA 主机的指令，Notify attribute 用于 FR801xH 回复 OTA 主机。在执行写操作时建议采用 write command 方式，以提高写入速度。

该代码位于 components\ble\profiles\ble_ota 中。

6.2 OTA 交互包格式

本章节的数据格式均采用小端模式。

6.2.1 OTA 主机端的请求包格式（通过 write attribute）：

包头		内容
Opcode	**Length（内容）**	**Payload（可选）**
1 字节	2 字节	变长

6.2.1.1 获取新固件的可用存储基地址

包头	
Opcode	**Length（内容）**
0x01	0x00 0x00

6.2.1.2 获取当前固件版本号

包头	
Opcode	**Length（内容）**
0x02	0x06 0x00

6.2.1.3 擦除扇区（4KB）

包头		内容
Opcode	**Length（内容）**	**Base_addr**
0x03	0x06 0x00	Flash 中的基地址偏移（4 字节，小端）

6.2.1.4 写入数据

包头		内容（可选）		
Opcode	**Length（内容）**	**Base_addr**	**Length**	**Payload**
0x05	6+payload_len	Flash 中的基地址偏移（4 字节，小端）	Payload 长度	要写入的内容

6.2.1.5 重启

包头		内容
Opcode	**Length（内容）**	**Payload（可选）**
0x09	0x00 0x00 或 0x10 0x00	写入 bin 文件的 MD5 值（16 字节）

6.2.2 FR801xH 的回复包格式（通过 Notify attribute）：

包头			内容
Result	**Opcode**	**Length（内容）**	**Payload（可选）**
1 字节	1 字节	2 字节	变长

6.2.2.1 获取新固件的可用存储基地址

包头	内容

Result	Opcode	Length（内容）	Payload
0x00	0x01	0x04 0x00	当前运行的代码在 flash 中存储的基地址（4 字节）

6.2.2.2 获取当前固件版本号

包头			内容
Result	Opcode	Length（内容）	Payload
0x00	0x02	0x04 0x00	当前固件版本号（4 字节）

6.2.2.3 擦除扇区（4KB）

包头			内容
Result	Opcode	Length（内容）	Payload
0x00	0x03	0x04 0x00	Flash 中的基地址偏移（4 字节）

6.2.2.4 写入数据

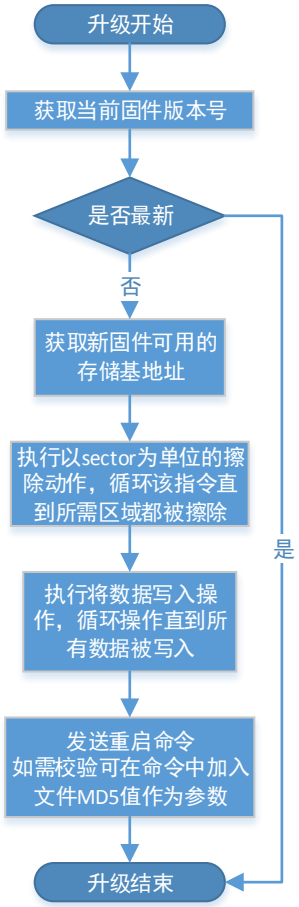
包头			内容	
Result	Opcode	Length（内容）	Base_addr	Length
0x00	0x05	0x06 0x00	写入的基地址偏移（4 字节）	写入的长度（2 字节）

6.2.2.5 重启

包头		
Result	Opcode	Length（内容）
0x00（检验成功）/ 0x01（校验失败）	0x09	0x00 0x00

6.3 OTA 流程

建议采用如下图所示的升级流程，流程中所涉及到的指令在上一节中均有描述：


升级开始
获取当前固件版本号
是否最新
否
是
获取新固件可用的存储基地址
执行以sector为单位的擦除动作，循环该指令直到所需区域都被擦除
执行将数据写入操作，循环操作直到所有数据被写入
发送重启命令
如需校验可在命令中加入文件MD5值作为参数
升级结束

联系方式

Feedback: Freqchip welcomes feedback on this product and this document. If you have comments or suggestions, please send an email to doc@freqchip.com.

Website: www.freqchip.com
Sales Email: sales@freqchip.com
Phone: +86-21-5027-0080

勘误记录

Reversion Number	Reversion Date	Description
V0.1	2019.06.12	Initial Version
V0.8	2019.11.30	Adding most APIs
V1.0	2020.02.16	Adding ADC driver APIs
V1.0.1	2020.02.22	Fix some mismatch with SDK source code
V1.0.2	2020.02.27	Add some GATT APIs
V1.0.3	2020.03.02	Add some GAP APIs
V1.0.4	2020.03.03	Add UART APIs
V1.0.5	2020.03.18	Add GAP RSSI Report API, add code file location for each API paragraph.