

**FR801xH SDK User Guide**

Bluetooth Low Energy SOC with SIG Mesh

2020-03 v1.0.5

www.freqchip.com

**目录**

[1. 概况 10](#_Toc35436465)

[1.1 FR801xH SDK结构 10](#_Toc35436466)

[1.2 空间分配 10](#_Toc35436467)

[1.2.1 地址空间 10](#_Toc35436468)

[1.2.2 空间分配 11](#_Toc35436469)

[1.3 代码流程 12](#_Toc35436470)

[1.3.1 user\_custom\_parameters函数 12](#_Toc35436471)

[1.3.2 user\_entry\_before\_ble\_init函数 13](#_Toc35436472)

[1.3.3 user\_entry\_after\_ble\_init函数 13](#_Toc35436473)

[1.3.4 睡眠唤醒用户接口 14](#_Toc35436474)

[1.4 \_\_jump\_table 14](#_Toc35436475)

[1.5 SDK 项目工程 15](#_Toc35436476)

[1.6 芯片烧录 15](#_Toc35436477)

[1.6.1 基于Keil + J-Link的烧录方式 16](#_Toc35436478)

[1.6.2 采用PC烧录工具+串口进行烧录 17](#_Toc35436479)

[1.6.3 量产烧录 18](#_Toc35436480)

[2. 低功耗管理 19](#_Toc35436481)

[2.1 工作模式 19](#_Toc35436482)

[2.2 程序运行流程 19](#_Toc35436483)

[2.3 唤醒条件 20](#_Toc35436484)

[3. BLE 协议栈 21](#_Toc35436485)

[3.1 GAP API 21](#_Toc35436486)

[3.1.1 GAP事件 21](#_Toc35436487)

[3.1.1.1 GAP Event Type Defines 21](#_Toc35436488)

[3.1.1.2 GAP Link Established Event 21](#_Toc35436489)

[3.1.1.3 GAP Link Disconnect Event 22](#_Toc35436490)

[3.1.1.4 GAP Link Parameter Update Reject Event 22](#_Toc35436491)

[3.1.1.5 GAP Link Parameter Update Success Event 22](#_Toc35436492)

[3.1.1.6 GAP Advertising Report Event 22](#_Toc35436493)

[3.1.1.7 GAP Peer Feature Event 23](#_Toc35436494)

[3.1.1.8 GAP GATT MTU Event 23](#_Toc35436495)

[3.1.1.9 GAP Security Master Authentication Request Event 23](#_Toc35436496)

[3.1.1.10 GAP Message Event 23](#_Toc35436497)

[3.1.1.11 GAP Advertising Mode Defines 24](#_Toc35436498)

[3.1.1.12 GAP Advertising Type Defines 24](#_Toc35436499)

[3.1.1.13 GAP Advertising Channel Defines 24](#_Toc35436500)

[3.1.1.14 GAP Advertising Filter Mode Defines 25](#_Toc35436501)

[3.1.1.15 GAP Advertising Paramters 25](#_Toc35436502)

[3.1.1.16 GAP Scan Mode Defines 25](#_Toc35436503)

[3.1.1.17 GAP Scan Result Type Defines 25](#_Toc35436504)

[3.1.1.18 GAP Scan Paramters 25](#_Toc35436505)

[3.1.1.19 GAP Pairing Mode Defines 26](#_Toc35436506)

[3.1.1.20 GAP IO Capabilities Defines 26](#_Toc35436507)

[3.1.1.21 GAP Security Parameters 26](#_Toc35436508)

[3.1.2 GAP函数 26](#_Toc35436509)

[3.1.2.1 GAP Set Event Callback Function 26](#_Toc35436510)

[3.1.2.2 GAP Set Advertising Paramters 27](#_Toc35436511)

[3.1.2.3 GAP Set Advertising data 28](#_Toc35436512)

[3.1.2.4 GAP Set Advertising Response data 29](#_Toc35436513)

[3.1.2.5 GAP Start Advertising 29](#_Toc35436514)

[3.1.2.6 GAP Stop Advertising 29](#_Toc35436515)

[3.1.2.7 GAP Start Scanning 29](#_Toc35436516)

[3.1.2.8 GAP Stop Scanning 30](#_Toc35436517)

[3.1.2.9 GAP Connect Request 30](#_Toc35436518)

[3.1.2.10 GAP Cancel Connection Procedure 31](#_Toc35436519)

[3.1.2.11 GAP Disconnect Request 31](#_Toc35436520)

[3.1.2.12 GAP Get Local Address 31](#_Toc35436521)

[3.1.2.13 GAP Set Local Address 31](#_Toc35436522)

[3.1.2.14 GAP Get Connection Status 32](#_Toc35436523)

[3.1.2.15 GAP Get Encryption Status 32](#_Toc35436524)

[3.1.2.16 GAP Set Device Name 32](#_Toc35436525)

[3.1.2.17 GAP Get Device Name 32](#_Toc35436526)

[3.1.2.18 GAP Set Device Appearance 33](#_Toc35436527)

[3.1.2.19 GAP Get Device Appearance 33](#_Toc35436528)

[3.1.2.20 GAP Get Connection Number 33](#_Toc35436529)

[3.1.2.21 GAP Get Link RSSI 33](#_Toc35436530)

[3.1.2.22 GAP Enable RSSI report in real time 34](#_Toc35436531)

[3.1.2.23 GAP Connection Parameters Update Request 34](#_Toc35436532)

[3.2 GATT API 34](#_Toc35436533)

[3.2.1 GATT 事件 34](#_Toc35436534)

[3.2.1.1 GATT Property Bitmap Defines 34](#_Toc35436535)

[3.2.1.2 GATT Operation Defines 35](#_Toc35436536)

[3.2.1.3 GATT Event Type 35](#_Toc35436537)

[3.2.1.4 GATT Operation Complete Event 35](#_Toc35436538)

[3.2.1.5 GATT Message Data 36](#_Toc35436539)

[3.2.1.6 GATT Message Event 36](#_Toc35436540)

[3.2.1.7 GATT Attribute Structure 36](#_Toc35436541)

[3.2.1.8 GATT Service Structure 37](#_Toc35436542)

[3.2.1.9 GATT Client Structure 37](#_Toc35436543)

[3.2.1.10 GATT Client Read 37](#_Toc35436544)

[3.2.1.11 GATT Client Enable Notification 37](#_Toc35436545)

[3.2.1.12 GATT Notification Structure 38](#_Toc35436546)

[3.2.1.13 GATT Indication Structure 38](#_Toc35436547)

[3.2.2 GATT 函数 38](#_Toc35436548)

[3.2.2.1 GATT Add Service 38](#_Toc35436549)

[3.2.2.2 GATT Add Client 39](#_Toc35436550)

[3.2.2.3 GATT Change Service UUID 39](#_Toc35436551)

[3.2.2.4 GATT Change Client UUID 39](#_Toc35436552)

[3.2.2.5 GATT Discover Peer Device All Services 39](#_Toc35436553)

[3.2.2.6 GATT Discover Peer Device Service By UUID 40](#_Toc35436554)

[3.2.2.7 GATT Write Request 40](#_Toc35436555)

[3.2.2.8 GATT Write Command 41](#_Toc35436556)

[3.2.2.9 GATT Enable Notification 41](#_Toc35436557)

[3.2.2.10 GATT Read Request 42](#_Toc35436558)

[3.2.2.11 GATT Notification 43](#_Toc35436559)

[3.2.2.12 GATT Indication 43](#_Toc35436560)

[3.2.2.13 GATT MTU Exchange Request 44](#_Toc35436561)

[3.2.2.14 GATT Get negotiated MTU size 44](#_Toc35436562)

[3.2.2.15 GATT deal message from HOST (condition: RTOS enable) 44](#_Toc35436563)

[3.3 Mesh API 45](#_Toc35436564)

[3.3.1 Mesh事件 45](#_Toc35436565)

[3.3.1.1 Mesh Event Type Defines 45](#_Toc35436566)

[3.3.1.2 Mesh network information updates 45](#_Toc35436567)

[3.3.1.3 Mesh Supported Features 46](#_Toc35436568)

[3.3.1.4 Mesh Provision Output OOB Mode 46](#_Toc35436569)

[3.3.1.5 Mesh Provision Input OOB Mode 46](#_Toc35436570)

[3.3.1.6 Mesh Provision Information 46](#_Toc35436571)

[3.3.1.7 Mesh Provision States 47](#_Toc35436572)

[3.3.1.8 Mesh Publish Message Type 47](#_Toc35436573)

[3.3.1.9 Mesh Response Message type 47](#_Toc35436574)

[3.3.1.10 Mesh Receive Message Type 47](#_Toc35436575)

[3.3.1.11 Mesh Model Structure 48](#_Toc35436576)

[3.3.1.12 Mesh Provision State Change Event 48](#_Toc35436577)

[3.3.1.13 Mesh Model Message Indication 48](#_Toc35436578)

[3.3.1.14 Network Key is NOT updated 49](#_Toc35436579)

[3.3.1.15 Network Key is updated 49](#_Toc35436580)

[3.3.1.16 Application Key is NOT updated 49](#_Toc35436581)

[3.3.1.17 Application Key is updated 49](#_Toc35436582)

[3.3.1.18 Model publication parameter for NOT virtual publication address 50](#_Toc35436583)

[3.3.1.19 Model publication parameter for virtual publication address 50](#_Toc35436584)

[3.3.1.20 Model subscription entry 51](#_Toc35436585)

[3.3.1.21 Key binding entry 51](#_Toc35436586)

[3.3.1.22 Configuration updated indication 51](#_Toc35436587)

[3.3.1.23 Mesh Event Structure 51](#_Toc35436588)

[3.3.2 Mesh函数 52](#_Toc35436589)

[3.3.2.1 Mesh Set Application Callback Function 52](#_Toc35436590)

[3.3.2.2 Mesh Initialization 52](#_Toc35436591)

[3.3.2.3 Mesh Set Runtime 52](#_Toc35436592)

[3.3.2.4 Mesh Start 53](#_Toc35436593)

[3.3.2.5 Mesh Stop 53](#_Toc35436594)

[3.3.2.6 Mesh Model Bind AppKey 53](#_Toc35436595)

[3.3.2.7 Mesh Model Subscribe Group Message 53](#_Toc35436596)

[3.3.2.8 Mesh Add Models 54](#_Toc35436597)

[3.3.2.9 Mesh Publish Message 54](#_Toc35436598)

[3.3.2.10 Mesh Send Response message 54](#_Toc35436599)

[3.3.2.11 Mesh Provision Parameters response 54](#_Toc35436600)

[3.3.2.12 Mesh Provision Authentication Data Response 55](#_Toc35436601)

[3.3.2.13 Mesh Composition Data Response 55](#_Toc35436602)

[3.3.2.14 Mesh Store Information Into Flash 55](#_Toc35436603)

[3.3.2.15 Mesh Clear Information In Flash 55](#_Toc35436604)

[3.4 Security API 56](#_Toc35436605)

[3.4.1 Security 函数 56](#_Toc35436606)

[3.4.1.1 GAP Bond Manager Initializtion 56](#_Toc35436607)

[3.4.1.2 GAP Bond Manager Delete All Bondings 56](#_Toc35436608)

[3.4.1.3 GAP Bond Manager Delete Single Bondging 56](#_Toc35436609)

[3.4.1.4 GAP Set Security Parameters 57](#_Toc35436610)

[3.4.1.5 GAP Sending Pairing Password 57](#_Toc35436611)

[3.4.1.6 GAP Pairing Request 57](#_Toc35436612)

[3.4.1.7 GAP Encrypt Request 58](#_Toc35436613)

[3.4.1.8 GAP Get Bond Status 58](#_Toc35436614)

[3.4.1.9 GAP Security Request 59](#_Toc35436615)

[3.5 BLE Profiles 59](#_Toc35436616)

[3.5.1 HID 59](#_Toc35436617)

[3.5.1.1 HID service 事件 59](#_Toc35436618)

[3.5.1.2 HID service 函数 60](#_Toc35436619)

[3.5.2 DIS 61](#_Toc35436620)

[3.5.2.1 DIS 事件 61](#_Toc35436621)

[3.5.2.2 DIS 函数 62](#_Toc35436622)

[3.5.3 Battery service 62](#_Toc35436623)

[3.5.3.1 BATT 事件 62](#_Toc35436624)

[3.5.3.2 BATT 函数 62](#_Toc35436625)

[3.5.4 OTA 63](#_Toc35436626)

[3.5.4.1 OTA 函数 63](#_Toc35436627)

[4. OSAL API 64](#_Toc35436628)

[4.1 Task API 64](#_Toc35436629)

[4.1.1 Task 函数 64](#_Toc35436630)

[4.1.1.1 OS Task Create 64](#_Toc35436631)

[4.1.1.2 OS Task Delete 64](#_Toc35436632)

[4.1.1.3 OS Message Post 65](#_Toc35436633)

[4.2 Clock API 65](#_Toc35436634)

[4.2.1 Clock 函数 65](#_Toc35436635)

[4.2.1.1 OS Timer Initialization 65](#_Toc35436636)

[4.2.1.2 OS Timer Start 66](#_Toc35436637)

[4.2.1.3 OS Timer Stop 66](#_Toc35436638)

[4.3 Memory API 66](#_Toc35436639)

[4.3.1 Memory 函数 66](#_Toc35436640)

[4.3.1.1 OS Malloc 66](#_Toc35436641)

[4.3.1.2 OS Get Free Heap Size 66](#_Toc35436642)

[4.3.1.3 OS Show Message List 67](#_Toc35436643)

[4.3.1.4 OS Show Kernel Malloc Informationi 67](#_Toc35436644)

[4.3.1.5 OS Show Memory List 67](#_Toc35436645)

[5. MCU外设驱动 68](#_Toc35436646)

[5.1 IO MUX 68](#_Toc35436647)

[5.1.1 普通IO接口 68](#_Toc35436648)

[5.1.1.1 IO 功能设置 68](#_Toc35436649)

[5.1.1.2 IO 上拉设置 69](#_Toc35436650)

[5.1.2 支持低功耗模式的IO接口 69](#_Toc35436651)

[5.1.2.1 IO 使能低功耗模式 69](#_Toc35436652)

[5.1.2.2 IO 关闭低功耗模式 69](#_Toc35436653)

[5.1.2.3 IO 低功耗模式功能设置 70](#_Toc35436654)

[5.1.2.4 IO 低功耗模式输入输出设置 70](#_Toc35436655)

[5.1.2.5 IO 低功耗模式上拉设置 70](#_Toc35436656)

[5.1.2.6 IO 使能低功耗唤醒 71](#_Toc35436657)

[5.1.2.7 IO 低功耗模式中断入口 71](#_Toc35436658)

[5.2 GPIO 72](#_Toc35436659)

[5.2.1 普通GPIO接口 72](#_Toc35436660)

[5.2.1.1 GPIO 输出 72](#_Toc35436661)

[5.2.1.2 GPIO获取当前值 72](#_Toc35436662)

[5.2.1.3 GPIO设置整个port输入输出 72](#_Toc35436663)

[5.2.1.4 GPIO 获取整个port输入输出配置 72](#_Toc35436664)

[5.2.1.5 GPIO 设置单个IO输入输出 73](#_Toc35436665)

[5.2.2 低功耗模式GPIO接口 73](#_Toc35436666)

[5.2.2.1 GPIO 低功耗模式输出值 73](#_Toc35436667)

[5.2.2.2 GPIO 低功耗模式输入值 74](#_Toc35436668)

[5.3 UART 74](#_Toc35436669)

[5.3.1 UART 初始化 74](#_Toc35436670)

[5.3.2 UART 等待发送FIFO为空 74](#_Toc35436671)

[5.3.3 从串口读取数据 75](#_Toc35436672)

[5.3.4 从串口发送数据 75](#_Toc35436673)

[5.3.5 UART 发送一个字节且等待完成 75](#_Toc35436674)

[5.3.6 UART 发送一个字节且立即返回 75](#_Toc35436675)

[5.3.7 UART 发送多个字节且等待完成 76](#_Toc35436676)

[5.3.8 UART 读取特定个数字节 76](#_Toc35436677)

[5.3.9 UART 读取特定个数字节，诺FIFO为空则先返回 76](#_Toc35436678)

[5.3.10 UART0 读数据 76](#_Toc35436679)

[5.3.11 UART0 发数据 77](#_Toc35436680)

[5.3.12 UART1 读数据 77](#_Toc35436681)

[5.3.13 UART1 写数据 77](#_Toc35436682)

[5.4 SPI 78](#_Toc35436683)

[5.4.1 SPI 初始化 78](#_Toc35436684)

[5.4.2 SPI 发送并接收 78](#_Toc35436685)

[5.4.3 SPI 发送 79](#_Toc35436686)

[5.4.4 SPI 接收 79](#_Toc35436687)

[5.5 I2C 79](#_Toc35436688)

[5.5.1 I2C 初始化 79](#_Toc35436689)

[5.5.2 I2C 发送一个字节 79](#_Toc35436690)

[5.5.3 I2C 发送多个字节 80](#_Toc35436691)

[5.5.4 I2C 读取一个字节 80](#_Toc35436692)

[5.5.5 I2C 读取多个字节 80](#_Toc35436693)

[5.6 Timer 81](#_Toc35436694)

[5.6.1 Timer 初始化 81](#_Toc35436695)

[5.6.2 Timer 启动 81](#_Toc35436696)

[5.6.3 Timer 停止 81](#_Toc35436697)

[5.6.4 Timer 获取load值 81](#_Toc35436698)

[5.6.5 Timer 获取当前计数值 82](#_Toc35436699)

[5.6.6 Timer 清中断 82](#_Toc35436700)

[5.7 PWM 82](#_Toc35436701)

[5.7.1 普通 PWM 接口 82](#_Toc35436702)

[5.7.1.1 PWM 初始化 82](#_Toc35436703)

[5.7.1.2 PWM 启动 82](#_Toc35436704)

[5.7.1.3 PWM 停止 83](#_Toc35436705)

[5.7.1.4 PWM 更新参数 83](#_Toc35436706)

[5.7.2 低功耗模式 PWM 接口 83](#_Toc35436707)

[5.7.2.1 低功耗PWM 初始化 83](#_Toc35436708)

[5.7.2.2 低功耗PWM 设置参数 83](#_Toc35436709)

[5.7.2.3 低功耗PWM 启动 84](#_Toc35436710)

[5.7.2.4 低功耗PWM 停止 85](#_Toc35436711)

[5.8 ADC 85](#_Toc35436712)

[5.8.1 ADC 初始化 85](#_Toc35436713)

[5.8.2 ADC 开始采样 85](#_Toc35436714)

[5.8.3 ADC 停止采样 85](#_Toc35436715)

[5.8.4 ADC 读取结果 86](#_Toc35436716)

[5.9 WDT 86](#_Toc35436717)

[5.9.1 WDT 初始化 86](#_Toc35436718)

[5.9.2 WDT 喂狗 86](#_Toc35436719)

[5.9.3 WDT 启动 87](#_Toc35436720)

[5.9.4 WDT 停止 87](#_Toc35436721)

[5.9.5 WDT 中断处理接口 87](#_Toc35436722)

[5.10 RTC 88](#_Toc35436723)

[5.10.1 RTC 初始化 88](#_Toc35436724)

[5.10.2 RTC 启动 88](#_Toc35436725)

[5.10.3 RTC 停止 89](#_Toc35436726)

[5.10.4 RTC 中断处理接口 89](#_Toc35436727)

[5.11 QDEC 89](#_Toc35436728)

[5.11.1 QDEC 初始化 90](#_Toc35436729)

[5.11.2 QDEC 设置引脚 90](#_Toc35436730)

[5.11.3 QDEC 设置清零条件 90](#_Toc35436731)

[5.11.4 QDEC 设置中断阀值 90](#_Toc35436732)

[5.11.5 QDEC 设置中断类型 91](#_Toc35436733)

[5.11.6 QDEC 去抖 91](#_Toc35436734)

[5.11.7 QDEC 读取旋转计数 91](#_Toc35436735)

[5.11.8 QDEC 中断处理接口 91](#_Toc35436736)

[5.12 Key Scan 92](#_Toc35436737)

[5.12.1 Key Scan 结构体定义 92](#_Toc35436738)

[5.12.1.1 Key Scan 参数 92](#_Toc35436739)

[5.12.2 Key Scan 函数 93](#_Toc35436740)

[5.12.2.1 Key Scan 初始化 93](#_Toc35436741)

[5.12.2.2 Key Scan 中断处理接口 93](#_Toc35436742)

[5.13 PMU 94](#_Toc35436743)

[5.13.1 PMU 配置系统电源 94](#_Toc35436744)

[5.13.2 PMU 判断系统是否第一次上电 94](#_Toc35436745)

[5.13.3 PMU 使能中断 94](#_Toc35436746)

[5.13.4 PMU 关闭中断 95](#_Toc35436747)

[5.13.5 PMU 使能Codec供电 95](#_Toc35436748)

[5.13.6 PMU 关闭Codec供电 95](#_Toc35436749)

[5.13.7 PMU 设置LDO\_OUT和IO电压值 96](#_Toc35436750)

[5.13.8 PMU 设置32K时钟源 96](#_Toc35436751)

[5.13.9 PMU 设置内部RC频率 96](#_Toc35436752)

[5.13.10 PMU Charger中断接口 96](#_Toc35436753)

[5.13.11 PMU 低电压监测中断接口 97](#_Toc35436754)

[5.13.12 PMU 高温监测中断接口 98](#_Toc35436755)

[6. OTA 99](#_Toc35436756)

[6.1 OTA profile 99](#_Toc35436757)

[6.2 OTA交互包格式 99](#_Toc35436758)

[6.2.1 OTA主机端的请求包格式（通过write attribute）： 99](#_Toc35436759)

[6.2.1.1 获取新固件的可用存储基地址 100](#_Toc35436760)

[6.2.1.2 获取当前固件版本号 100](#_Toc35436761)

[6.2.1.3 擦除扇区（4KB） 100](#_Toc35436762)

[6.2.1.4 写入数据 100](#_Toc35436763)

[6.2.1.5 重启 100](#_Toc35436764)

[6.2.2 FR801xH的回复包格式（通过Notify attribute）： 100](#_Toc35436765)

[6.2.2.1 获取新固件的可用存储基地址 100](#_Toc35436766)

[6.2.2.2 获取当前固件版本号 101](#_Toc35436767)

[6.2.2.3 擦除扇区（4KB） 101](#_Toc35436768)

[6.2.2.4 写入数据 101](#_Toc35436769)

[6.2.2.5 重启 101](#_Toc35436770)

[6.3 OTA流程 101](#_Toc35436771)

[联系方式 103](#_Toc35436772)

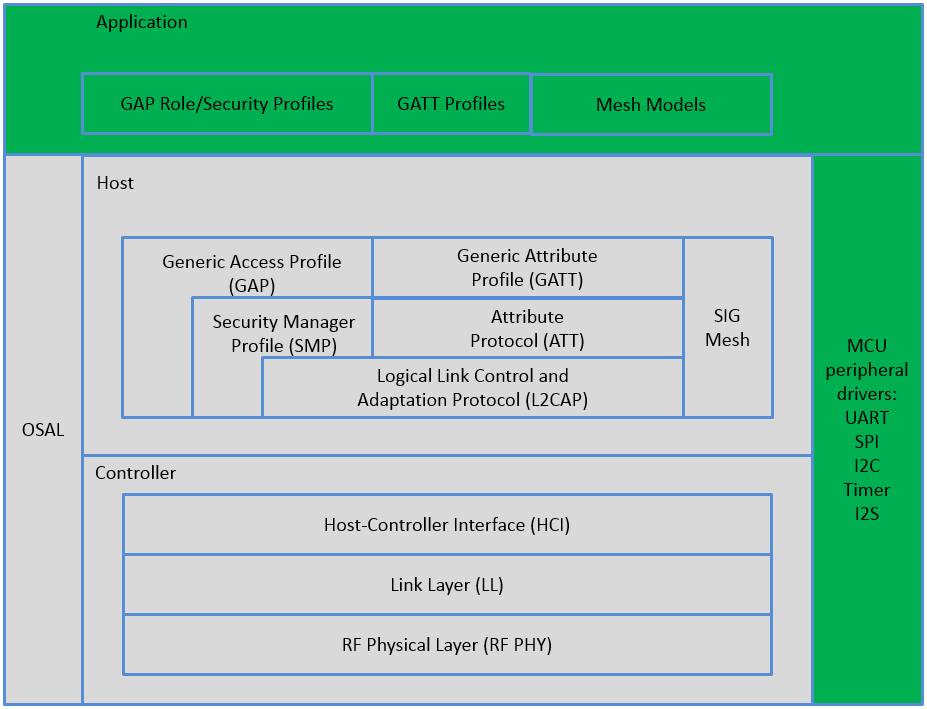
[勘误记录 104](#_Toc35436773)

# 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。

## 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

## 空间分配

### 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。

### 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。

## 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

### 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。

### 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 \*/

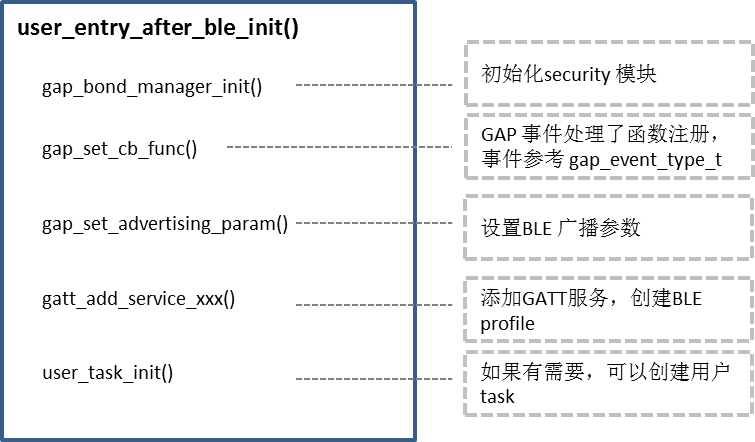
app\_at\_init();

……

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

### 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

### 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。

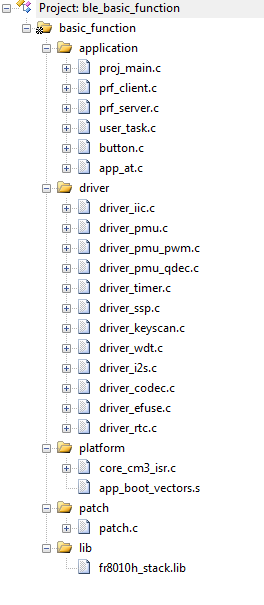
## \_\_jump\_table

\_\_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  SYSTEM\_SYS\_CLK\_48M | 系统工作主频。 |

## 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

## 芯片烧录

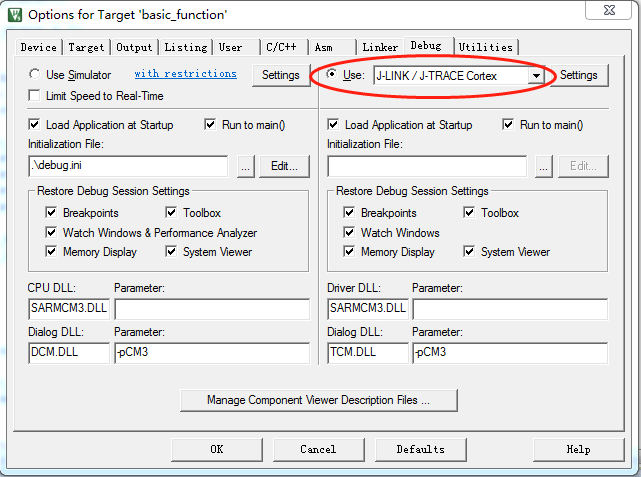
芯片烧录方式主要有两种：

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

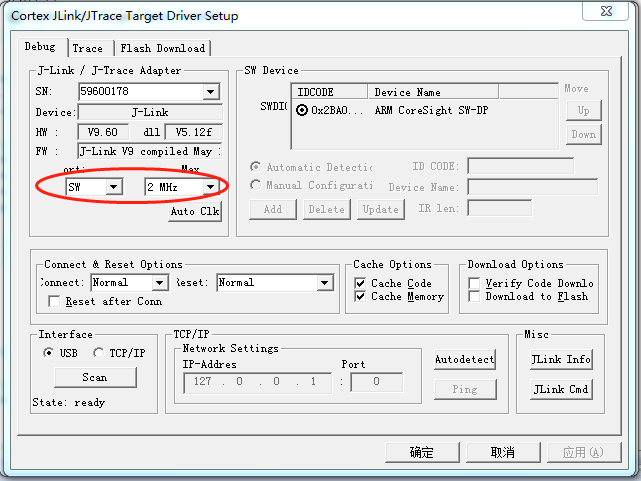
### 基于Keil + J-Link的烧录方式

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

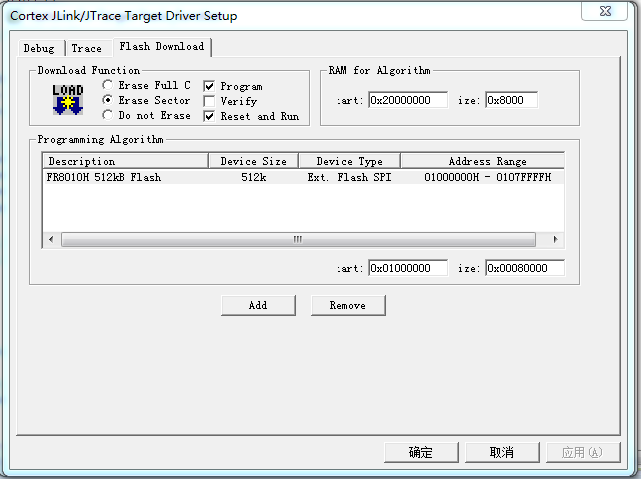
1. 选用J-Link作为调试工具



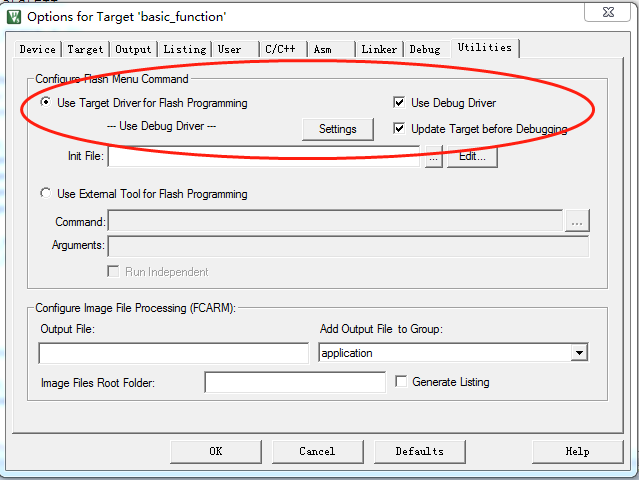
1. 配置调试方式为SW



1. 在flash download选项卡中配置下载选项



1. 配置使用Debug Driver进行flash的烧录

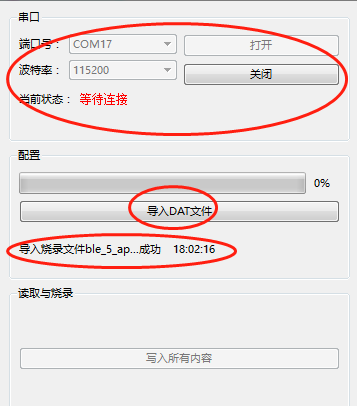


通过以上配置就可以实现在Keil的IDE中进行flash的调试和烧录。

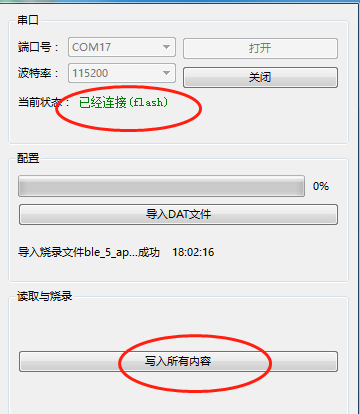
### 采用PC烧录工具+串口进行烧录

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。



1. 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）。
2. 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。

### 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。

# 低功耗管理

## 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。

## 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。

## 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。

# 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。

## GAP API

GAP层的API位于gap\_api.h 文件。

### GAP事件

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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。 |

#### 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。 |

#### 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 |

#### 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 |

#### 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; | | |

#### 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。 |

#### 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 |

#### 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; | | |

#### 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。 |

#### 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 |

#### 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; | | |

### GAP函数

#### 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); | | |

#### 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); | | |

#### 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 |  |  |

#### 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 |  |  |

#### 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); | | |

#### 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 |  |  |

#### 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); | | |

#### 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 |  |  |

#### 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); | | |

#### 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 |  |  |

#### 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 |  |  |

#### 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. | | |

#### 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); | | |

#### 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 |

#### 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 |

#### 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 |  |  |

#### 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 |

#### 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 |  |  |

#### 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 |

#### 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。 |

#### 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 |  |  |

#### 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 |  |  |

#### 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); | | |

## GATT API

### GATT 事件

#### 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 |

#### 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。 |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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。 | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

### GATT 函数

#### 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。 |

#### 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。 |

#### 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 |  |  |

#### 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 |  |  |

#### 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;  } } | | |

#### 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 |  |  |

#### 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 |  |  |

#### 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; } | | |

#### 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; } | | |

#### 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; } | | |

#### 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); | | |

#### 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 |  |  |

#### 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 |  |  |

#### 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 |

#### 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 |  |  |

## Mesh API

### Mesh事件

#### 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。 |
| }; | | |

#### 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 |
| }; | | |

#### 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。 |
| }; |  |  |

#### 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。 |
| }; |  |  |

#### 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。 |
| }; |  |  |

#### 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。 |
| }; |  |  |

#### 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。 |
| }; |  |  |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

#### 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; | | |

### Mesh函数

#### 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 |  |  |

#### 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 |  |  |

#### Mesh Set Runtime

|  |  |  |
| --- | --- | --- |
| **void mesh\_set\_runtime(void)** | | |
| 设置当前时间。 | | |
| **参数：** | | |
| None |  |  |
| **返回：** | | |
| None |  |  |

#### 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 |  |  |

#### Mesh Stop

|  |  |  |
| --- | --- | --- |
| **void mesh\_stop(void)** | | |
| 停止mesh功能。 | | |
| **参数：** | | |
| None |  |  |
| **返回：** | | |
| None |  |  |

#### 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 |  |  |

#### 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 |  |  |

#### 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 |  |  |

#### 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 |  |  |

#### 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 |  |  |

#### 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 |  |  |

#### 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 |  |  |

#### 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 |  |  |

#### 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 |  |  |

#### 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 |  |  |

## Security API

### Security 函数

#### 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); | | |

#### 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 |  |  |

#### 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 |  |  |

#### 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); | | |

#### 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); | | |

#### 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 |  |  |

#### 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 |  |  |

#### 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); | | |

#### 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); | | |

## 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。

### 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。

#### 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; | | |

#### 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 |  |  |

### DIS

Device Information Service。提供设备的产品信息以及软硬件版本。

#### 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。 |

#### DIS 函数

|  |  |  |
| --- | --- | --- |
| **void dis\_gatt\_add\_service(void)** | | |
| 创建device information的profile。 | | |
| **参数：** | | |
| None |  |  |
| **返回：** | | |
| None |  |  |

### Battery service

Battery service 用于实时电池电量监控。

#### 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; | | |

#### 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 |  |  |

### OTA

Over The Air download，空中固件升级。

#### OTA 函数

|  |  |  |
| --- | --- | --- |
| **void ota\_gatt\_add\_service(void)** | | |
| 创建OTA服务的profile。 | | |
| **参数：** | | |
| None |  |  |
| **返回：** | | |
| None |  |  |

# OSAL API

## Task API

### Task 函数

#### 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); } | | |

#### 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 |  |  |

#### 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 |  |  |

## Clock API

### Clock 函数

#### 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。 } | | |

#### 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，1 ~ 0x3FFFFF |
| repeat\_flag | - | 定时器是否重复。 |
| **返回：** | | |
| None |  |  |

#### 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 |  |  |

## Memory API

### Memory 函数

#### 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。 |

#### 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。 |

#### 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 |  |  |

#### 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 |  |  |

#### 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 |  |  |

# MCU外设驱动

## 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

### 普通IO接口

以下API位于driver\_system.h中。

#### 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); | | |

#### 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); | | |

### 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。

#### 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); | | |

#### 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); | | |

#### 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); | | |

#### 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); | | |

#### 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); | | |

#### 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); | | |

#### 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); } | | |

## GPIO

### 普通GPIO接口

#### 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); | | |

#### 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 |

#### 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 |  |  |

#### 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 |

#### 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); | | |

### 低功耗模式GPIO接口

#### 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); } | | |

#### 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) ); } | | |

## UART

### 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 | - |  |

### 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 | - |  |

### 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：** | | |
|  | | |

### 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：** | | |
|  | | |

### 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 | - |  |

### 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 | - |  |

### 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 | - |  |

### 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 | - |  |

### 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 |

### 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 |  |  |

### 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 |  |  |

### 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 |  |  |

### 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 |  |  |

## SPI

### 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); | | |

### 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 |  |  |

### 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 |  |  |

### 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 |  |  |

## I2C

### 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 |  |  |

### 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 |

### 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 |

### 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 |

### 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 |

## Timer

### 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 | - |  |

### 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 |  |  |

### 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 |  |  |

### 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 |

### 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 |

### 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 |  |  |

## PWM

### 普通 PWM 接口

#### 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 |  |  |

#### 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 |  |  |

#### 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 |  |  |

#### 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 |  |  |

### 低功耗模式 PWM 接口

#### 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 |  |  |

#### 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 |  |  |

#### 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); | | |

#### 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 |  |  |

## ADC

以下API维护driver\_adc.h中。

### 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 |  |  |

### 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。 |

### ADC 停止采样

|  |  |  |
| --- | --- | --- |
| **void adc\_disable(void)** | | |
| 停止正在采样的ADC | | |
| **参数：** | | |
| None |  |  |
| **返回：** | | |
| None |  |  |

### 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 |  |  |

## WDT

注意：WDT默认就支持低功耗模式。

### 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 |  |  |

### 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 |  |  |

### WDT 启动

|  |  |  |
| --- | --- | --- |
| **void wdt\_start(void)** | | |
| 启动wdt。超时时间清0。 | | |
| **参数：** | | |
| None |  |  |
| **返回：** | | |
| None |  |  |

### WDT 停止

|  |  |  |
| --- | --- | --- |
| **void wdt\_stop(void)** | | |
| 停止wdt。 | | |
| **参数：** | | |
| None |  |  |
| **返回：** | | |
| None |  |  |

### 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(); } | | |

## RTC

注意：RTC默认支持低功耗模式。

### 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 |  |  |

### 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 |  |  |

### 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 |  |  |

### 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 } | | |

## QDEC

注意：QDEC（旋转编码器）默认支持低功耗模式。

### 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 |  |  |

### 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 |  |  |

### 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 |  |  |

### 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 |  |  |

### 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 |  |  |

### 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 |  |  |

### 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 |

### 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);  } | | |

## Key Scan

注意：Key Scan默认支持低功耗模式。

### Key Scan 结构体定义

#### 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]} 或者{PC[7:4], PD[3:0]}，由row\_map\_sel值决定哪种映射。 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 | 行pin脚映射选择。0，行映射{PD[7:0]}。1，行映射{PC[7:4], PD[3:0]} |
| }keyscan\_param\_t; | | |

### Key Scan 函数

#### 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 |  |  |

#### 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); } | | |

## PMU

### 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 |  |  |

### 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。 |

### 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); } | | |

### 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 |  |  |

### 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 |  |  |

### 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 |  |  |

### 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 |  |  |

### 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 |  |  |

### 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 |

### 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); } | | |

### 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); } | | |

### 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); } | | |

# 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。



## 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。

## OTA交互包格式

本章节的数据格式均采用小端模式。

### 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 |

#### 获取新固件的可用存储基地址

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

#### 获取当前固件版本号

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

#### 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） |

#### 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 |

#### 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） |

### 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 | |

#### 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） | |

#### 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） | |

#### 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） | |

#### 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） |

#### 重启

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

## 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：



# 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](mailto:doc@freqchip.com).

**Website**: [www.freqchip.com](http://www.freqchip.com)

**Sales Email**: [sales@freqchip.com](mailto:sales@freqchip.com)

**Phone**: +86-21-5027-0080

# 勘误记录

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