# AN5289
# 应用笔记

## 使用STM32WB系列微控制器构建无线应用

## 引言

本应用笔记旨在指导设计者基于STM32WB系列微控制器，完成构建特定低功耗蓝牙®（BLE）或802.15.4应用所需的步骤。它汇集了最重要的信息，并且列出了需要处理的方面。

为了充分利用本文档中的信息进行应用开发，用户必须熟悉STM32微控制器、BLE技术、802.15.4 OpenThread协议和802.15.4 MAC层，并且必须理解诸如低功耗管理和任务调度等系统服务。

# 目录

# 表格索引

# 图片目录

# 1 参考


[1] UM2550(1) 面向STM32WB系列的STM32CubeWB入门

[2] RM0434(1) 基于使用FPU的多协议无线32位MCU Arm® Cortex®-M4蓝牙®低功耗和802.15.4无线解决方案

[3] AN5270(1) STM32WBx5 蓝牙®低功耗（BLE）无线接口

[4] UM2442(1) STM32WB HAL与底层驱动程序说明

[5] UM2288(1) 用于无线性能测量的STM32CubeMonitor-RF软件工具

[6] AN5185(1) STM32WB系列的ST固件升级服务

[7] 蓝牙®规范 蓝牙核心规范（v4.0、v4.1、v4.2和v5.0）

[8] MAC IEEE Std 802.15.4-2011 802_15_4 MAC标准规范

[9] Thread规范 Thread规范V1.1（Thread组）


1. *可从www.st.com 下载*。

# 2 缩写和缩略语列表

| | |
|---|---|
| ACI | 应用命令接口 |
| ATT | 属性协议 |
| BLE | 低功耗蓝牙® |
| CLI | 命令行接口 |
| CoAP | 受限应用协议 |
| CPU1 | Cortex®-M4内核 |
| CPU2 | Cortex®+M0内核 |
| D2D | 器件到器件 |
| DUT | 被测器件 |
| FUOTA | 无线固件更新 |
| FUS | 固件升级服务 |
| GAP | 通用访问配置文件 |
| GATT | 通用属性参数文件 |
| HCI | 主机控制器接口 |
| L2CAP | 逻辑链路控制和适配协议 |
| LTK | 长期密钥 |
| OTA | 无线 |
| PDU | 协议数据单元 |
| P2P | 点对点 |
| RFU | 保留供将来使用 |
| SIG | 技术联盟 |
| SM | 安全管理器 |
| UUID | 通用唯一标识符 |

# 3 软件概述

## 3.1 所支持的无线协议栈

STM32WB系列微控制器基于Arm®(a)内核。

根据目标应用选择要加载的CPU2固件。

STM32WB系列微控制器生态系统支持不同的无线协议栈（参见*表 1*），由应用通过特定接口进行控制，如*图 1*所示。

如*图 2*所示，CPU2可以提供BT HCI标准接口，CPU1上可以运行其他BLE无线协议栈。

**表1. STM32WB系列微控制器支持的无线协议栈**

| 支持的无线协议栈 | 相关固件 |
|---|---|
| BLE | stm32wb5x_BLE_Stack_fw.bin<br>stm32wb5x_BLE_HCILayer_fw.bin |
| Thread | stm32wb5x_Thread_FTD_fw.bin<br>stm32wb5x_Thread_MTD_fw.bin |
| BLE和Thread | stm32wb5x_BLE_Thread_fw.bin |
| MAC 802_15_4 | stm32wb5x_Mac_802_15_4_fw.bin |

a. Arm 是 Arm Limited （或其子公司）在美国和 / 或其他地区的注册商标。

图1. STM32WB系列微控制器支持的协议

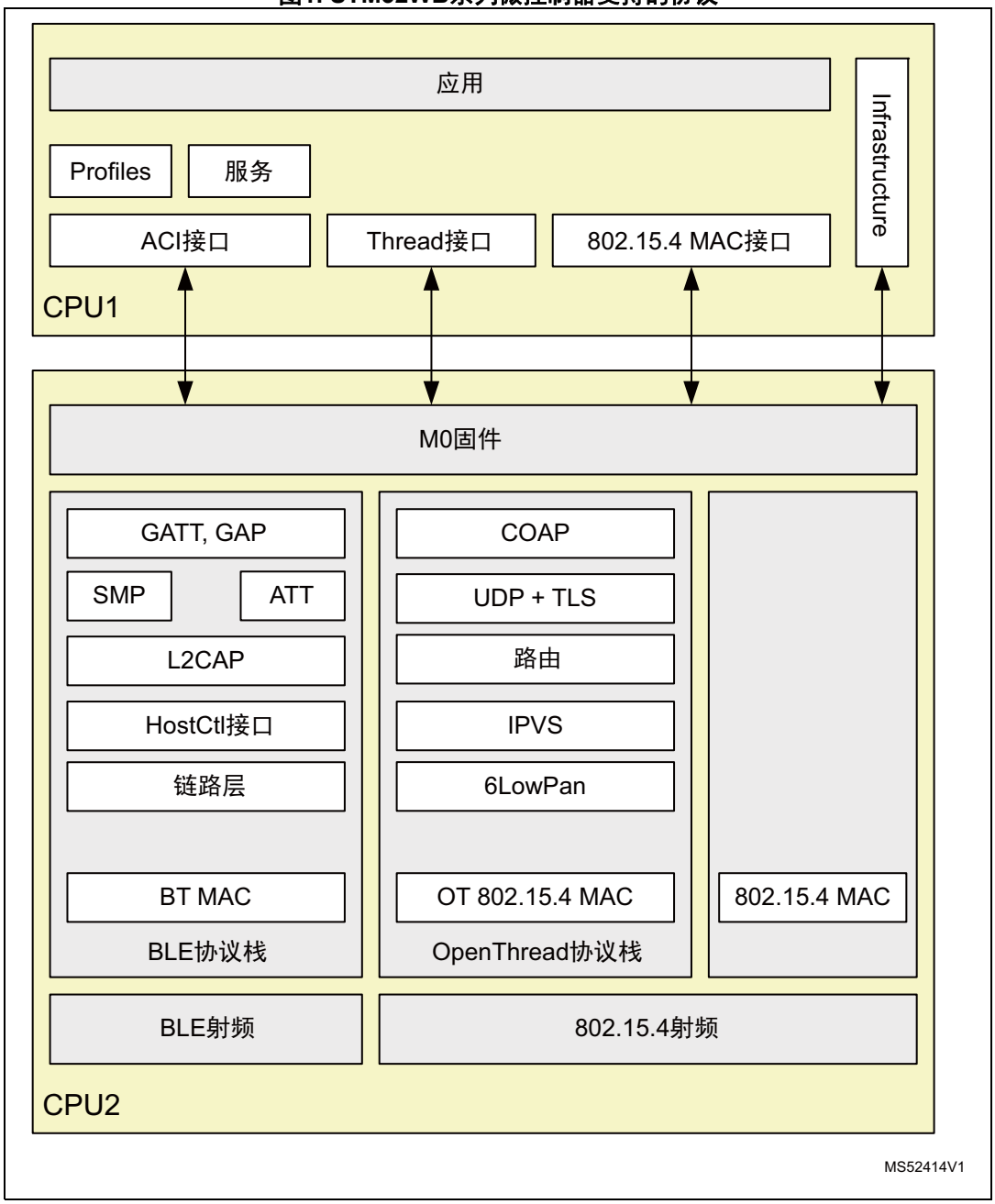
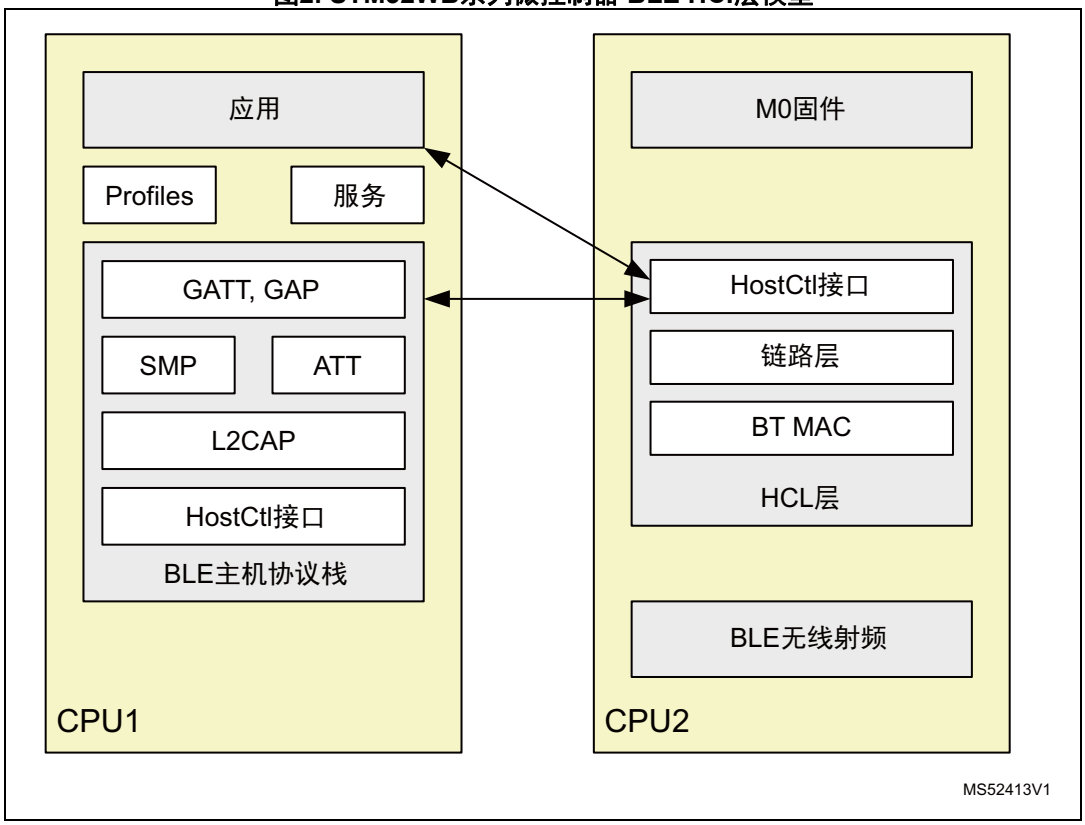


图2. STM32WB系列微控制器 BLE HCI层模型




## 3.2 BLE应用

STM32WB架构分离了BLE配置文件和应用，应用在CPU1上运行，BLE外设提供实时性。

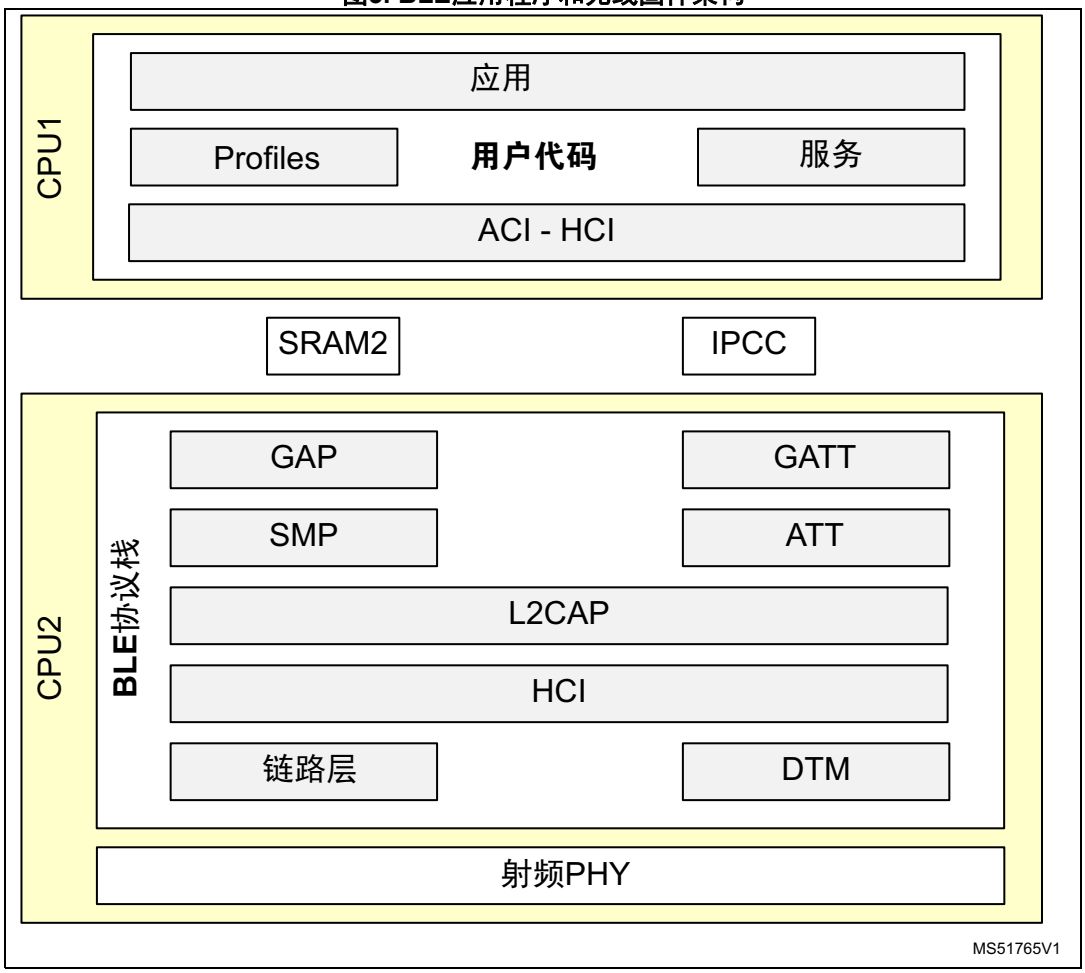
BLE外设包含了CPU2处理器，其中包含用于处理链路层直到GAP及GAP层的无线协议栈。此外，它还包含了2.4 GHz无线射频部分。

CPU1收集并计算要传输到BLE的应用数据。

CPU2包含管理所有实时链路层和无线PHY交互所需的LE控制器和LE主机，包括：

- 低功耗管理器，用于控制低功耗模式
- 调试跟踪工具，用于输出活动的相关信息
- 信箱（Mailbox） / IPCC，用于连接BLE无线协议栈（LL、GAP和GATT）

图3. BLE应用程序和无线固件架构




## 3.3 在HCI层接口之上构建BLE应用

CPU2可用作BLE HCI层协处理器。在这种情况下，用户要么实现自己的HCI应用程序，要么使用现有的开源 BLE 主机协议栈。

大多数BLE主机协议栈使用UART接口与BLE HCI协处理器进行通信。STM32WB系列微控制器的等效物理层是信箱（mailbox），如*第 13.2 节：信箱（Mailbox）接口*所述。

信箱（Mailbox）为BLE通道和系统通道提供了一个接口。BLE主机协议栈负责构建要通过信箱（Mailbox）上BLE通道发送的命令缓冲区，并且必须提供接口用于报告通过信箱（Mailbox）接收到的事件。除了通过信箱（Mailbox）完成BLE主机协议栈自适应，用户还必须在可以发布异步数据包时通知信箱（Mailbox）驱动程序。

系统通道不由 BLE 主机协议栈处理。用户必须实现自定义传输层，以构建要发送到信箱（Mailbox）驱动程序的系统命令缓冲区并管理从信箱（Mailbox）接收到的事件（包括向信箱（Mailbox）驱动程序释放异步缓冲区的通知），或者也可以使用信箱（Mailbox）扩展驱动程序（如*第 13.3 节：信箱（Mailbox）接口 - 扩展*所述），以便在负责构建系统命令缓冲区和管理系统异步事件的传输层之上提供接口。

BLE_TransparentMode项目可用作使用信箱（Mailbox）在BLE HCI层协处理器之上构建应用（如*第 11.2 节：Thread_Coap_DataTransfer*所述）的例子。

## 3.4 Thread应用

OpenThread协议栈运行在CPU2内核上，并在CPU1侧导出一组API，以便构建完整的Thread应用。三个CPU2固件支持Thread协议：

- **sm32wb5x_Thread_FTD_fw**：在这种情况下，设备支持除边界路由器外的所有Thread角色（例如：主导设备（Leader）、路由器、终端设备和休眠终端设备）。
- **stm32wb5x_Thread_MTD_fw**：在这种情况下，设备只能充当终端设备或休眠终端设备）。相比于FTD配置，这种配置更节省存储空间。
- **stm32wb5x_BLE_Thread_fw**：在这种情况下，设备在静态并发模式下同时支持Thread（FTD）和BLE（请参考*第 3.6 节*获取更多信息）。

## 3.5 MAC 802_15_4应用

在下载STM32wb5x_Mac_802_15_4_fw CPU2固件时，CPU1可以直接访问802_15_4 MAC层并在这一层之上构建自己的应用。

## 3.6 BLE和Thread应用并发模式

STM32WB系列微控制器支持“静态并发模式”（也称“开关模式”）。

内嵌两种无线协议栈（BLE和Thread）的stm32wb5x_BLE_Thread_fwCPU2固件可从ST网站*www.st.com*上获取。通过系统应用命令完成从一种协议到另一种协议的切换。在该模式下，系统在激活另一种协议前禁用正在使用的协议。STM32WB器件在完全停止 BLE无线协议栈后从BLE切换至Thread，反之亦然。可能需要几秒钟的时间完成此类过渡，因为每次都需要重新连接网络。

# 4 STM32WB软件架构

## 4.1 主要原则

- 在CPU2上运行的所有代码均以加密二进制数据的形式交付
- 在客户的视角CPU2是个黑盒
- 在CPU1上运行的所有代码均以源代码的形式交付
- CPU之间通过“信箱（Mailbox）”进行通信

标准STM32Cube交付包包含诸如以下STM32WB资源：

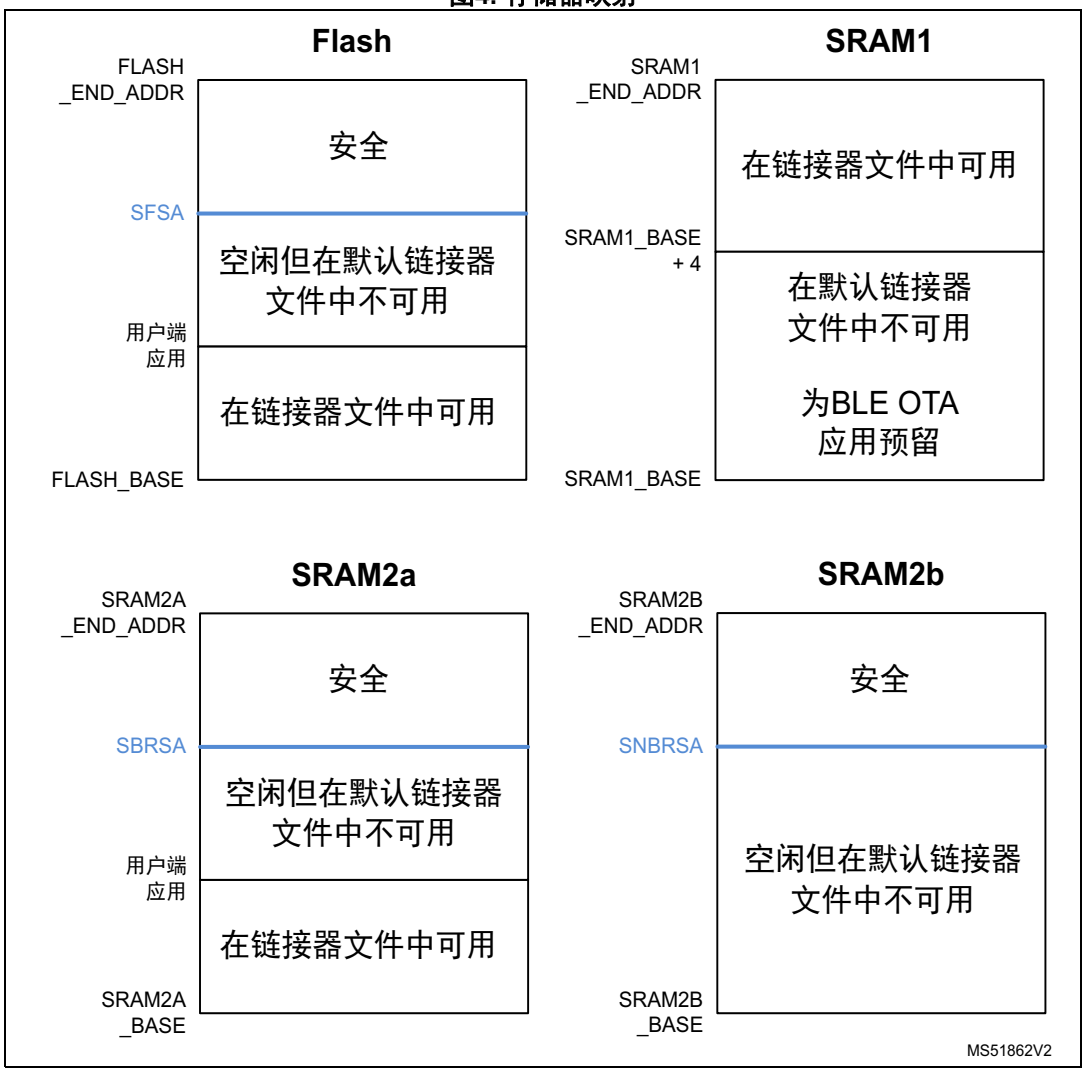
- 用于访问硬件寄存器的HAL/LL
- BSP
- 中间件（例如：FreeRTOS、USB设备）。

此外，下列应用提供高效的系统集成：

- 调度器，用于在后台执行任务并在没有活动时进入低功耗模式
- 定时器服务，为应用提供在RTC上运行的虚拟定时器（在停止和待机模式下）。

## 4.2 存储器映射

图4. 存储器映射




闪存、SRAM2a和SRAM2b存储器包含不能通过CPU1读取或写入的安全存储区。可从选项字节读取每个存储器的安全起始地址，在*图 4*中用蓝色表示：

- 闪存为SFSA
- SRAM2a为SBRSA（在待机时保留）
- SRAM2b为SNBRSA

这些选项字节只能通过运行在CPU2上的FUS写入。这是FUS在每次安装CPU2更新时完成的。

用户应用必须考虑到不同RF无线协议栈版本的可用内存可能不一样。用户应用的可用空间可从STM32WB协处理器无线协议二进制文件的版本说明中获取。RF无线协议栈的安装地址也是用户闪存区域的边界地址。确保CPU2域中包含一些余量，以支持产品生命周期中的更新。

闪存的边界粒度为4 kB，SRAM2a和SRAM2b的为1 kB。

交付的所有BLE/Thread应用（BLE_Thread_Static、BLE_HeartRate_ota和BLE_p2pServer_ota除外）的链接器文件是相同的。选择的可用内存适合提供的所有应用。对于像BLE这样CPU2内存需求较少的应用，可以更新链接器文件，以便为应用分配更多内存。

为了优化专用应用的可用内存，必须按照下列指导方针更新链接器文件：

- 闪存：可用内存的结束地址可以到SFSA地址。当需要更新CPU2时 ，必须有刚好低于安全内存的足够空闲内存用于上传新的加密CPU2固件更新。所需内存大小取决于要更新的CPU2固件（BLE、Thread或 并发 BLE/Thread），参见*[1]*。
- SRAM1：只有BLE_OTA应用要求链接器文件中的前32位不可用 。对于所有其他应用，起始地址可从SRAM1_BASE + 4到SRAM1_BASE。
- SRAM2a：可用内存的结束地址可以到SBRSA地址。当需要CPU2更新支持时，必须有刚好低于安全内存的一些空闲扇区用于支持新的CPU2固件更新（需要更多扇区才稳妥）。
- SRAM2b：SRAM2b不是链接器文件的一部分，因为它对支持Thread协议的任何固件CPU2都是完全安全的。对于只使用BLE的应用，可通过新存储区更新链接器文件，以便将RW数据映射到SRAM2B（从SRAM2B_BASE到SNBRSA地址）。当需要CPU2更新支持时，必须有刚好低于安全内存的一些空闲扇区用于支持新的CPU2固件更新（需要更多扇区才稳妥）。

STOP2是RF激活时支持的最深低功耗模式。当用户应用必须进入待机模式时，必须首先停止所有RF活动，并在退出待机模式时
完全重新初始化CPU2。用户应用可使用整个非安全SRAM2a来存储自己的内容（在待机模式下需要保留的内容）。

## 4.3 共享外设

通过硬件信号量保护两个CPU可同时访问的任何外设。在访问这些外设前，必须先获取相关信号量，然后再发布。

**表2. 信号量**

| 信号量 | 目的 |
|---|---|
| Sem0 | RNG - 所有寄存器 |
| Sem1 | PKA - 所有寄存器 |
| Sem2 | FLASH - 所有寄存器 |
| Sem3 | RCC_CR<br>RCC_EXTCFGR<br>RCC_CFGR<br>RCC_SMPSCR |
| Sem4 | 停止模式实现的时钟控制机制 |
| Sem5 | RCC_CRRCR<br>RCC_CCIPR |
| Sem6 | 被CPU1用来防止CPU2在闪存中写入/擦除数据 |
| Sem7 | 被CPU2用来防止CPU1在闪存中写入/擦除数据 |

如果应用需使用信号量进行任务间控制，建议使用Sem31往下的信号量以兼容未来CPU1上的无线固件更新，这些更新可能是添加需要额外信号量的新特性。

**Sem0**用于在两个CPU之间共享RNG IP。CPU2占用信号量并持续一段时间，这段时间取决于要生成的必要RNG数和RNG源时钟速度。为了缩短获取RNG数的延时，建议首先生成一个RNG数池，在应用取出了一些数后，在池中填充低优先级任务以维持装满状态。Sem0的使用如*图 8*所示。

Sem 0也可以用在USB用例中。当不再使用USB且需要通过应用关闭时，必须在关闭CLK48时钟前获取Sem0。必须这样做的原因是，USB和RNG共享一个时钟，当CPU1需要关闭USB时，CPU2可能正在使用RNG（参见*图 9*）。

**Sem1**用于在两个CPU之间共享PKA IP。

**Sem2**用于在两个CPU之间共享FLASH IP。CPU2占用信号量并持续一段时间，这段时间取决于要在闪存中写入的数据量和要擦除的扇区数。BLE协议栈将配对信息（当绑定启用时）和GATT属性缓存写入闪存。

**Sem3**用于低功耗管理。当有BLE的RF活动时，其被CPU1锁定的时间不得超过500 µs。算法详情见*图 6*和*图 7*。

当一个CPU退出低功耗模式同时另一个CPU进入低功耗模式时，使用**Sem4**处理系统时钟切换时的竞争状态。算法详情见*图 6*和*图 7*。

示例中使用了Sem3和Sem4进入/退出停止模式。

用户必须确保在从停止模式唤醒前和唤醒后执行*图 6*和*图 7*中所示的算法。这些例程（参见*图 5*）通常在调度器或RTOS的IDLE任务中实现。此类实现基于这样一个事实：当从临界区调用WFI时，在发生中断请求时MCU被唤醒，但它不执行ISR，而是执行WFI后的下一条指令。只有在退出临界区后才会执行ISR。

```
PRIMASK = 1; // Mask all interrupts (enter critical section) PWR_EnterStopMode()
WFI
PWR_ExitStopMode()
PRIMASK = 0; // Unmask all interrupts (exit critical section)
```

图5. 在CPU1上进入/退出停止模式的时序

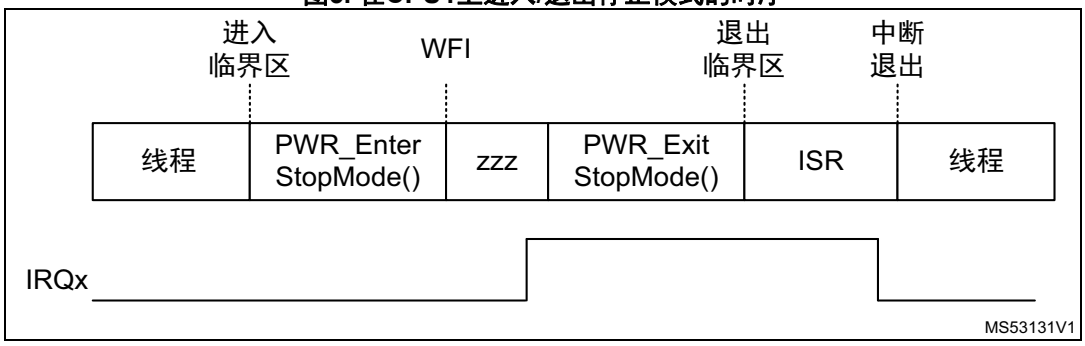


**图6. 在CPU1上进入停止模式的算法**

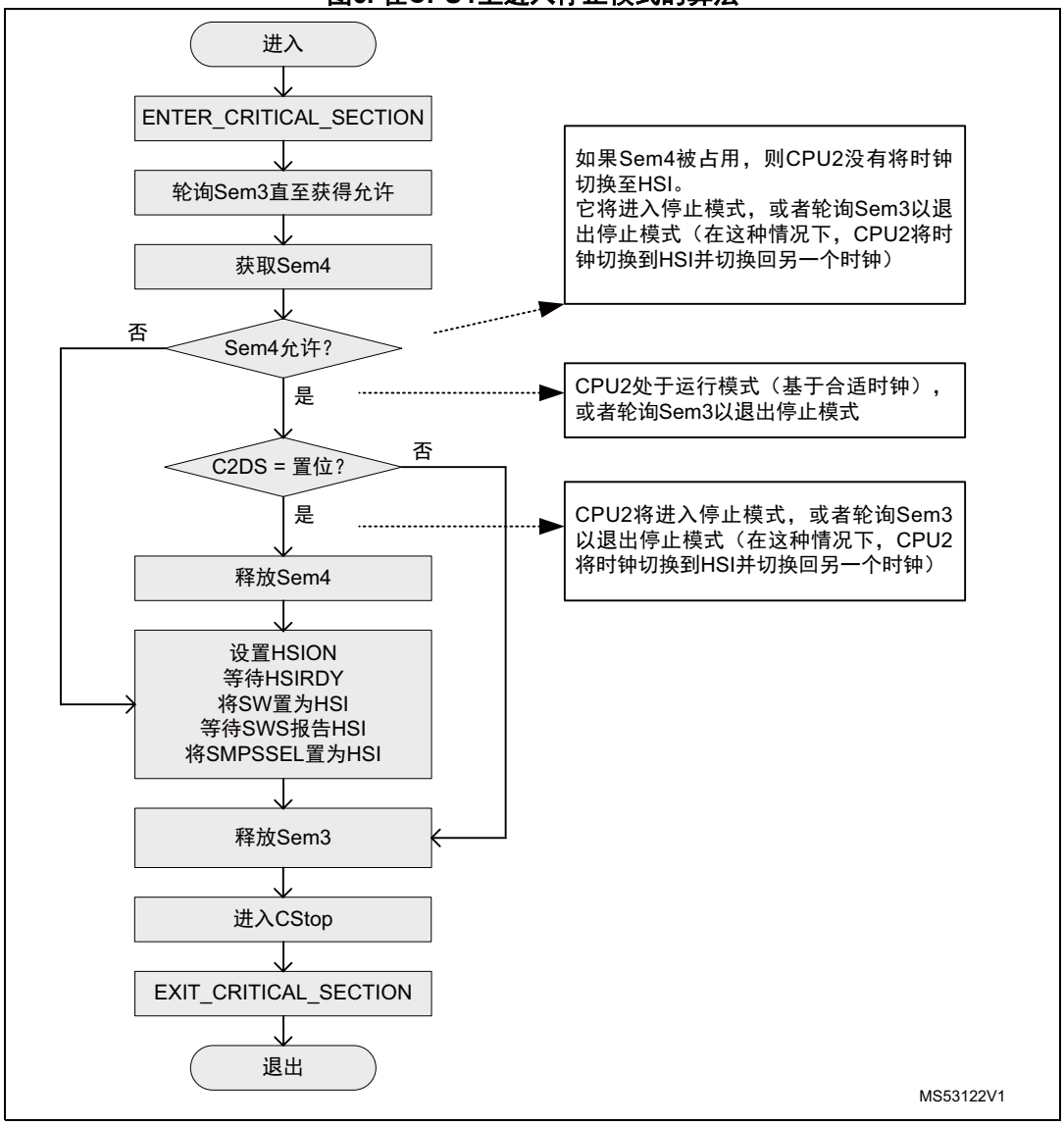
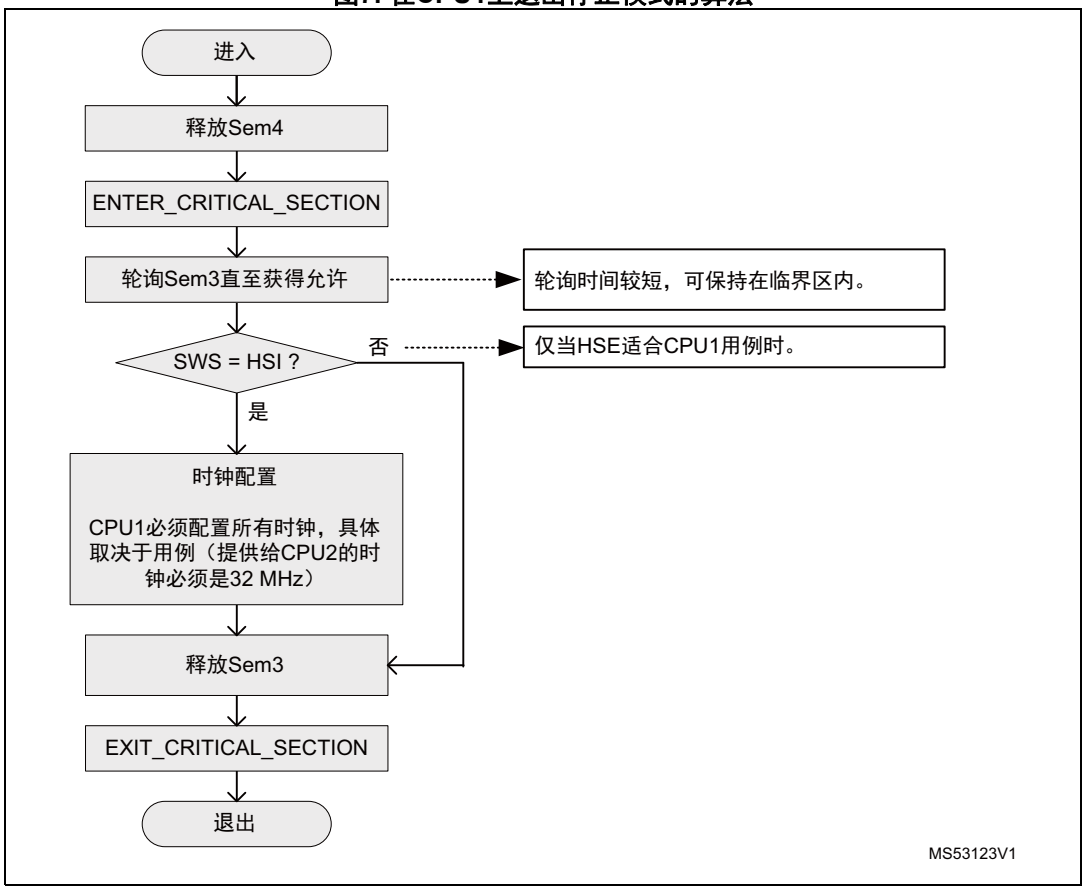


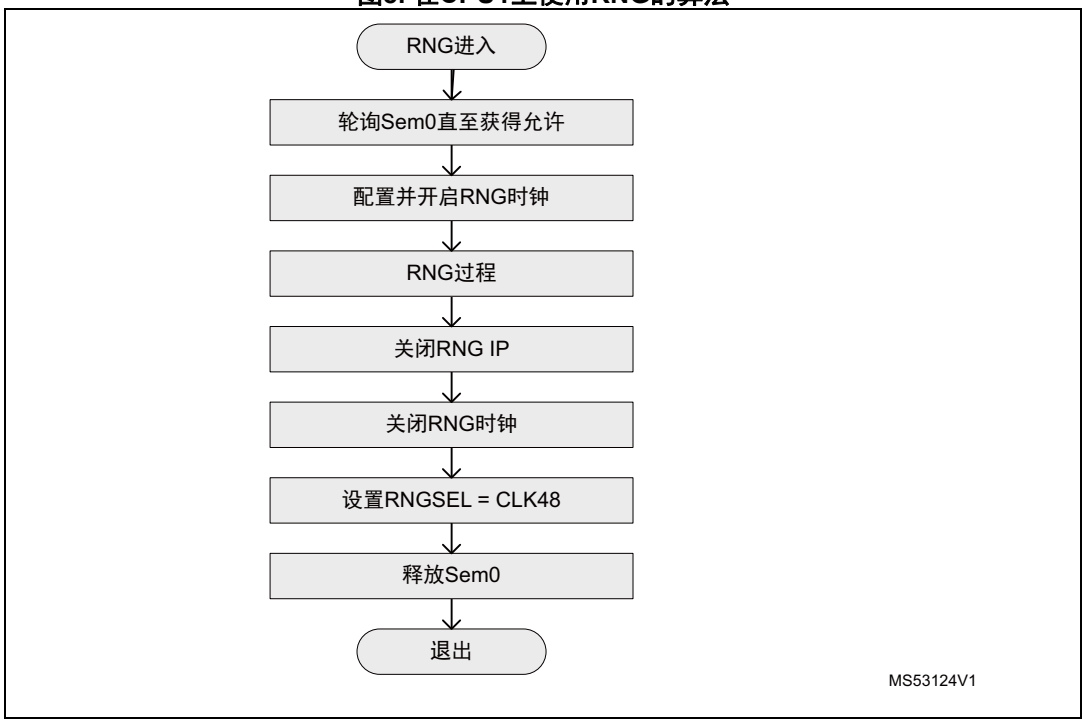
**图7. 在CPU1上退出停止模式的算法**




**Sem5**用于控制RNG/USB CLK48源时钟。仅当使用RNG IP（Sem0）时，CPU2才更新或关闭时钟。

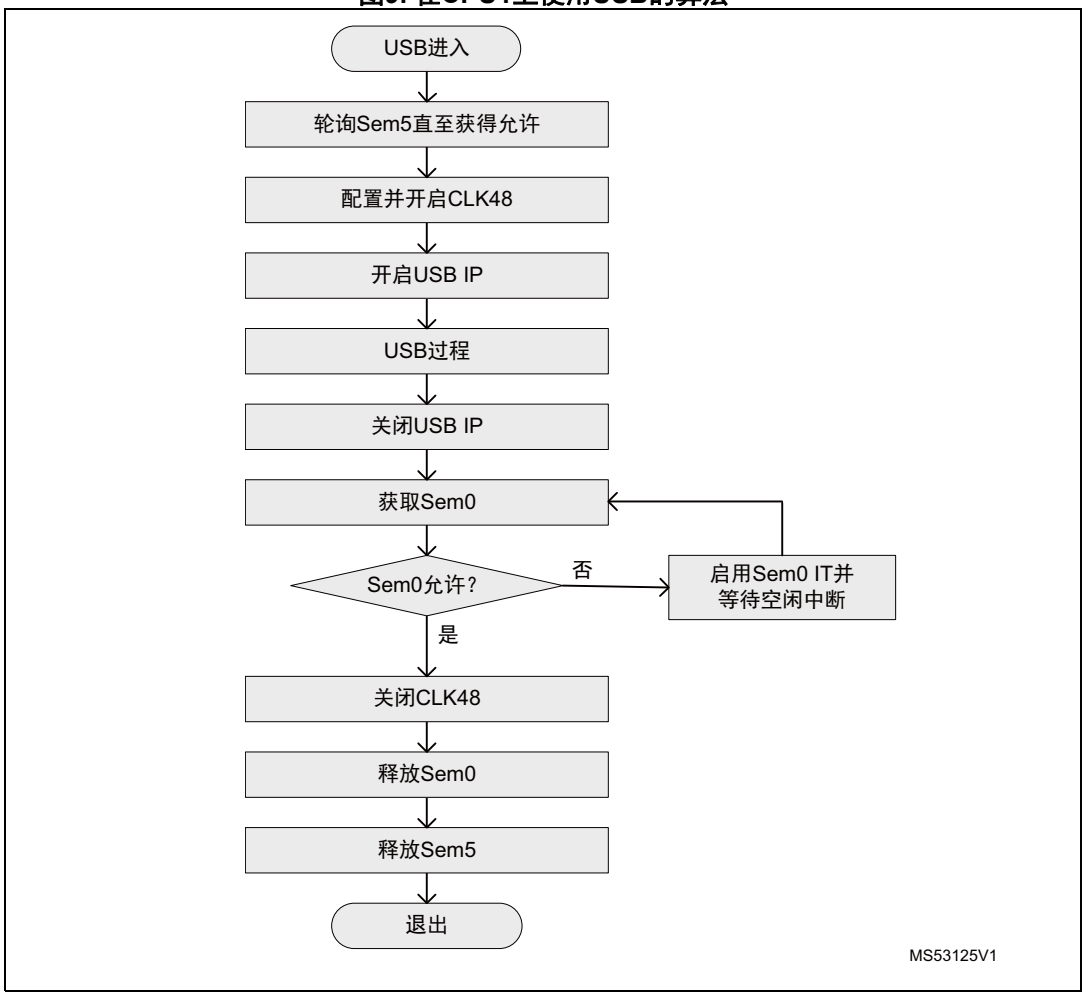
为避免与CPU2形成竞争状态，当CPU1需要关闭时钟时，即使不使用RNG IP，也必须首先获取Sem0。BLE P-NUCLEO-WB55.USB dongle示例（参见*第 7.1.3节：配置*和*图 9*）中展示了这一机制。这不会影响CPU2。

**图8. 在CPU1上使用RNG的算法**




*注：* *不获取Sem5，原因是CPU2不会在没有先获取Sem0的情况下获取Sem5。可以更新此算法，从而在配置RNG时钟源之前获取Sem5。*

**图9. 在CPU1上使用USB的算法**




USB和RNG共享同一个时钟源。在关闭时钟前，USB驱动程序必须先检查CPU2是否需要时钟。为避免与CPU2形成竞争状态，CPU1必须在关闭时钟前获取Sem0（RNG信号量，CPU2不使用USB）。

如果Sem0处于忙碌状态，CPU1必须等到Sem0空闲时才能关闭时钟。必须这样做的原因在于，当CPU1释放USB和CPU2释放RNG同时发生时，会形成竞争状态，导致振荡器持续开启。

**Sem6**用于保护CPU1时序不受CPU2请求的写入/擦除操作的影响。CPU1应获取Sem6，以防止CPU2或其他CPU1进程在闪存中写入或擦除数据。CPU1能够持有信号量的时间长度并无限制，但只要信号量被占用，CPU2就不能在存储器中写入配对或客户端描述符信息。

仅当CPU1能够在完成写入或擦除操作所需的时间内保持停止时，CPU1才必须释放Sem6。

类似于CPU1，CPU2实施*图 10*所示的算法。在闪存中写入或擦除数据时，它尝试获取Sem6，如果成功，将写入/擦除数据并释放信号量。当CPU1需要保护其时序时，它将轮询Sem6直至获取它。

**Sem7**用于保护CPU2时序不受CPU1请求的写入/擦除闪存操作的影响。CPU1必须在写入或擦除前获取Sem7。必须为每次写入或擦除操作获取和释放Sem7，但除去写入/擦除时间外，不得超过0.5 ms。为了符合这一要求，必须在临界区执行代码。该算法如 *图 10*中所述。

## 4.4 调度器

调度器逐一执行注册的函数。它具有下列功能：

- 支持最多32个函数
- 请求执行函数
- 启用/禁用函数执行
- 基于事件接收提供阻塞接口。

调度器提供简单的后台调度功能。它提供hook函数，用于在调度器没有任何挂起的任务要执行时实现安全的低功耗模式（无事件丢失）。 它还为应用提供了一种高效的机制，让应用在继续操作前等待特定事件。当调度器等待特定事件时，它会提供一个hook，
应用程序可以在其中进入低功耗模式或执行一些其他代码。

### 4.4.1 实现方法

为了使用调度器，应用必须：

- 设置支持的最大函数数量（通过定义UTIL_SEQ_CONF_TASK_NBR值来设置）
- 通过UTIL_SEQ_RegTask()注册调度器要支持的函数
- 通过调用UTIL_SEQ_Run()运行后台while循环来启动调度器
- 在需要执行某个函数时调用UTIL_SEQ_SetTask()。

### 4.4.2 接口

**表3. 接口函数**

| 功能 | 说明 |
|---|---|
| void UTIL_SEQ_Idle( void ); | 在没有可执行的代码时调用（临界区 - PRIMASK）。 |
| void UTIL_SEQ_Run( UTIL_SEQ_bm_t mask_bm ) | 请求调度器执行挂起的函数并在mask_bm中启用。 |
| void UTIL_SEQ_RegTask(UTIL_SEQ_bm_t task_id_bm, uint32_t flags, void (*task)( void )) | 注册与调度器中的信号（task_id_bm）关联的函数（任务）。task_id_bm必须有一位置位。 |
| void UTIL_SEQ_SetTask( UTIL_SEQ_bm_t task_id_bm, | 请求执行与task_id_bm关联的函数。<br>调度器只在函数执行完毕时评估task_prio。如果任一时刻有多个挂起的函数，将执行优先级最高（0）的函数。 |
| void UTIL_SEQ_PauseTask( UTIL_SEQ_bm_t task_id_bm ) | 禁止调度器执行与task_id_bm关联的函数。 |
| void UTIL_SEQ_ResumeTask( UTIL_SEQ_bm_t task_id_bm ) | 允许调度器执行与task_id_bm关联的函数。 |
| void UTIL_SEQ_WaitEvt( UTIL_SEQ_bm_t evt_id_bm ) | 请求调度器等待特定事件evt_id_bm，并且在事件通过UTIL_SEQ_SetEvt()置位前不返回。 |

表3. 接口函数（续）

| 功能 | 说明 |
| --- | --- |
| void UTIL_SEQ_SetEvt( UTIL_SEQ_bm_t evt_id_bm ) | 通知调度器发生了evt_id_bm事件（必须首先请求事件）。 |
| void UTIL_SEQ_EvtIdle(UTIL_SEQ_bm_t task_id_bm, UTIL_SEQ_bm_t evt_waited_bm) | 在调度器等待特定事件时调用。 |
| void UTIL_SEQ_ClrEvt( UTIL_SEQ_bm_t evt_id_bm ) | 将挂起事件清零。 |
| UTIL_SEQ_bm_t UTIL_SEQ_IsEvtPend( void ) | 返回挂起事件的evt_id_bm。 |

### 4.4.3 具体接口与行为

调度器是一组while循环，用于在用户请求时调用函数：

```
while(1)
{
    if(task_id1)
    {
        task_id1 = 0;
        Fct1();
    }

    if (task_id2)
    {
        task_id2= 0;
        Fct2();
    }

    __disable_irq();
    If (!(task_id1|| task_id2))
    {
        UTIL_SEQ_Idle();
    }
    __enable_irq();
}
```

void UTIL_SEQ_Run(UTIL_SEQ_bm_t mask_bm)

实现while (1)循环的主体。mask_bm参数是允许调度器执行的函数列表。每个函数与该mask_bm中的一个位相关联。在启动结束时，必须在while (1)循环中调用此API，使mask_bm = (~0) 以允许调度器执行任何挂起的函数。

void UTIL_SEQ_Idle( void )

在调度器没有任何要执行的函数时在临界区之下调用（通过CortexM PRIMASK位置位 - 所有中断均被屏蔽）。应用必须在这里进入低功耗模式。

void UTIL_SEQ_RegTask(UTIL_SEQ_bm_t task_id_bm, uint32_t flags, void (*task)( void ))

通知调度器将与标记task_id_bm关联的函数任务添加到其while循环。

void UTIL_SEQ_SetTask( UTIL_SEQ_bm_t task_id_bm , UTIL_SEQ_bm_t task_prio )

为调度器设置标记task_id_bm以调用关联的函数。

当需要决定要调用的下一个函数时，调度器评估task_prio。只有在当前函数执行完毕时才能进行评估。如果有多个函数设置了标记，将执行优先级较高的函数（0最高）。在以不同优先级实际执行函数前，可多次调用此API。在这种情况下，调度器将记录最高优先级。无论在执行函数前执行了多少次API调用，调度器都只运行一次关联函数。

void UTIL_SEQ_PauseTask( UTIL_SEQ_bm_t task_id_bm ) :

通知调度器不要执行标记task_id_bm的关联函数，即使标记已置位。如果在UTIL_SEQ_PauseTask()之后调用API UTIL_SEQ_SetTask()，将记录请求但不执行函数。与UTIL_SEQ_PauseTask()关联的掩码和与
void UTIL_SEQ_Run(UTIL_SEQ_bm_t mask_bm)关联的掩码无关。

仅当函数的标记置位且在两个掩码中均启用（默认情况）时才能执行函数。

void UTIL_SEQ_ResumeTask( UTIL_SEQ_bm_t task_id_bm ) :

取消UTIL_SEQ_PauseTask()的请求。如果在没有请求UTIL_SEQ_PauseTask()的情况下调用此API，此API无效。

void UTIL_SEQ_WaitEvt( UTIL_SEQ_bm_t evt_id_bm )

此API被调用后，在关联的evt_id_bm信号置位前不会返回。evt_id_bm 32位值中只需有一位置位。在调度器等待此事件时，它在发生事件evt_id_bm时在while循环中调用UTIL_SEQ_EvtIdle()。如果在继续执行前轮询标记，必须用该API替换所有代码。

void UTIL_SEQ_SetEvt( UTIL_SEQ_bm_t evt_id_bm )

只有在已调用UTIL_SEQ_WaitEvt()时才能调用。它将函数UTIL_SEQ_WaitEvt()等待的信号evt_id_bm置位。在函数UTIL_SEQ_WaitEvt()之前调用此API会使对UTIL_SEQ_WaitEvt()的调用立即返回，因为标记已置位。

void UTIL_SEQ_EvtIdle(UTIL_SEQ_bm_t task_id_bm, UTIL_SEQ_bm_t evt_waited_bm)

在API void UTIL_SEQ_WaitEvt()等待UTIL_SEQ_SetEvt()完成信号置位时调用。

在调度器中弱实现此API以调用UTIL_SEQ_Run(0)，这意味着在等待该事件发生时，函数UTIL_SEQ_Idle()允许系统在等待标记时进入低功耗模式。

应用可以实现此API，以将非0参数传递给UTIL_SEQ_Run(mask_bm)。mask_bm中每个置为1的位都要求调度器执行与该标记（通过UTIL_SEQ_SetTask()置位）关联的函数。这意味着如果调用函数UTIL_SEQ_WaitEvt()，在等待请求的事件返回时，它可能执行解掩码函数（如果其标记置位）或调用UTIL_SEQ_Idle()（如果没有任何任务等待调度器执行）。

void UTIL_SEQ_ClrEvt( UTIL_SEQ_bm_t evt_id_bm )

在某些应用中，可以在Evt已置位时需要调用API UTIL_SEQ_WaitEvt()时调用此API。在这种情况下，必须将Evt清零。

UTIL_SEQ_bm_t UTIL_SEQ_IsEvtPend( void ):

此API返回当前挂起的Evt。如果嵌套了多个UTIL_SEQ_WaitEvt()，将返回最后一个，也就是让较深的UTIL_SEQ_WaitEvt()返回到其调用方的那一个。

## 4.5 定时器服务

定时器服务具有以下特性：

- 最多255个虚拟定时器（取决于可用的RAM容量）
- 单发和重复模式
- 停止虚拟定时器并用不同超时值将其重启
- 删除定时器
- 1至$2^{32}$ - 1个时间片的超时

定时器服务提供多个共享RTC唤醒定时器的虚拟定时器。每个虚拟定时器都可以定义为单发或重复定时器。当重复定时器运行到周期末尾时，将通知用户且虚拟定时器在相同超时后自动重启。当单发定时器定时结束时，将通知用户且虚拟定时器被置为挂起状态（即保持寄存状态且随时可以重启）。用户可以停止虚拟定时器并用不同超时值将其重启。当不再需要虚拟定时器时，用户必须将其删除以释放定时器服务中的时隙。

定时器服务可与日历同时使用。

### 4.5.1 实现方法

为了使用定时器服务，应用必须：

- 配置RTC IP。当应用中需要日历时，RTC配置必须与日历设置要求兼容。当不使用日历时，可以优化RTC以便只使用定时器服务。
- 通过HW_TS_Init()初始化定时器服务。
- 实现HW_TS_RTC_Int_AppNot()（可选）。如不实现，在RTC中断处理函数上下文中调用定时器回调。
- 通过HW_TS_Create()创建虚拟定时器。
- 通过HW_TS_Stop()、HW_TS_Start()使用虚拟定时器。
- 在不需要时使用HW_TS_Delete()删除虚拟定时器。

### 4.5.2 接口

**表4. 接口函数**

| 功能 | 说明 |
|---|---|
| void HW_TS_Init(<br>HW_TS_InitMode_t TimerInitMode,<br>RTC_HandleTypeDef *hrtc); | 初始化定时器服务。 |
| HW_TS_ReturnStatus_t HW_TS_Create(<br>uint32_t TimerProcessID,<br>uint8_t *pTimerId,<br>HW_TS_Mode_t TimerMode,<br>HW_TS_pTimerCb_t pTimerCallBack) | 创建虚拟定时器。 |
| void HW_TS_Stop(uint8_t TimerID) | 停止虚拟定时器。 |
| void HW_TS_Start(<br>uint8_t TimerID,<br>uint32_t timeout_ticks) | 启动虚拟定时器。 |
| void HW_TS_Delete(uint8_t TimerID | 删除虚拟定时器。 |
| void HW_TS_RTC_Wakeup_Handler(void); | 将从RTC中断处理函数调用的定时器服务处理函数。 |
| uint16_t HW_TS_RTC_ReadLeftTicksToCount(void); | 返回下一次中断前的时间片计数。 |
| void HW_TS_RTC_Int_AppNot(<br>uint32_t TimerProcessID,<br>uint8_t TimerID,<br>HW_TS_pTimerCb_t pTimerCallBack) | 向应用报告虚拟定时器已超时。 |
| void HW_TS_RTC_CountUpdated_AppNot(void) | 向应用报告定时器服务更新了下一次中断前的时间片数。 |

### 4.5.3 具体接口与行为

定时器服务提供虚拟计时器，这些定时器在系统处于
低功耗模式到待机模式时运行。

void HW_TS_Init(HW_TS_InitMode_t TimerInitMode, RTC_HandleTypeDef *hrtc) :

此命令基于RTC IP配置（必须事先设置）初始化定时器服务。

TimerInitMode选择定时器服务启动模式。如果支持待机模式且设备从待机模式唤醒，将TimerInitMode置为hw_ts_InitMode_Limited以使定时器服务上下文不重置。否则，必须将TimerInitMode置为hw_ts_InitMode_Full以运行完整初始化。

hrtc是Cube HAL RTC句柄。

HW_TS_ReturnStatus_t HW_TS_Create(uint32_t TimerProcessID,
uint8_t *pTimerId,
HW_TS_Mode_t TimerMode,
HW_TS_pTimerCb_t pTimerCallBack) :

pTimerId

这是定时器服务返回给调用方的ID，需要用于停止/启动/删除创建的定时器。

TimerMode

定时器模式可以是单发或重复模式。如果是单发模式，定时器将在达到超时时停止。如果是重复模式，定时器将在每次达到相同超时（事先设定的值）时重启。此模式是定时器创建时就确定不变的。如需更改模式，必须删除该定时器并创建新的定时器。请注意，在这种情况下，新分配的pTimerId可能不一样。

pTimerCallBack

超时时用户回调。

TimerProcessID

它由用户定义，用在HW_TS_RTC_Int_AppNot()中。在创建定时器时，只有调用方知道分配的ID。TimerProcessID是在HW_TS_RTC_Int_AppNot()中通过pTimerCallBack返回的，因此可以在实现HW_TS_RTC_Int_AppNot()时做出相关决策。

void HW_TS_Stop(uint8_t TimerID)

停止定时器TimerID。如果定时器没有运行，则不起作用。定时器 TimerID必须已创建。 当定时器停止时，TimerID 仍然分配在定时器服务中，以便可以使用不同的值重新启动相同的定时器

（具有相同的 TimerMode 和相同的 pTimerCallBack）。

用timeout_ticks值启动定时器TimerID。timeout_ticks的值取决于RTC IP的配置。如果TimerID已经在运行，则首先在定时器服务中将其停止，然后用新的timeout_ticks值重启。

void HW_TS_Delete(uint8_t TimerID)

从定时器服务中删除TimerID。可以将TimerID分配给新的虚拟定时器。此API可能被正在运行的TimerID调用。在这种情况下，首先停止TimerID然后将其删除。

void HW_TS_RTC_Wakeup_Handler(void)

应用必须在RTC中断处理函数中调用此中断处理函数。此处理函数清除RTC和EXTI外设中需要的所有状态标记。

uint16_t HW_TS_RTC_ReadLeftTicksToCount(void)

此API返回在定时器服务生成中断前剩余要计数的时间片数。当系统需要进入低功耗模式和决定要应用的低功耗模式时，可以使用它，具体取决于下一次唤醒的预期发生时间。

当定时器被禁用（列表中没有定时器）时，它返回0xFFFF。

void HW_TS_RTC_Int_AppNot(uint32_t TimerProcessID,
uint8_t TimerID,
HW_TS_pTimerCb_t pTimerCallBack)

此API必须由用户应用来实现。

它在定时器超时时通知应用。此API在RTC唤醒中断上下文中运行，应用可能倾向于将pTimerCallBack作为后台任务来调用，具体取决于在pTimerCallBack中执行的代码量。只要TimerID仅 调用方知道，就可以用TimerProcessID识别此pTimerCallBack所属的模块，并且应用可以评估是否可以在RTC唤醒中断上下文中调用。

void HW_TS_RTC_CountUpdated_AppNot(void):

此API必须由用户应用来实现。

此API通知应用计数器已更新。此API与HW_TS_RTC_ReadLeftTicksToCount () API一起使用。在上一次调用HW_TS_RTC_ReadLeftTicksToCount()后和进入低功耗模式前，计数器可能已更新。此通知为应用提供了解决竞争状态的方法，可在进入低功耗模式前重新评估计数器值。

## 4.6 低功耗管理器

低功耗管理器提供一个简单接口，用于从最多接收32个不同用户请求输入，并计算系统可以使用的最低可能功耗模式。它还在进入或退出低功耗模式前为应用提供了hooks函数。

低功耗管理器具备以下特性：

- 最多支持32个用户请求
- 停止模式和关闭模式（待机和关机）。
- 低功耗模式选择
- 低功耗模式执行
- 在 进入或退出低功耗模式时回调
- 不支持运行模式，当应用必须保持此模式时，不得调用UTIL_LPM_EnterModeSelected()。

### 4.6.1 实现方法

低功耗管理器可以处理最多32个用户的不同低功耗模式请求。

为了使用低功耗管理器，应用必须：

- 创建用户ID
- 随时用定义的用户ID调用UTIL_LPM_SetOffMode()或UTIL_LPM_SetStopMode()，以便设置请求的低功耗模式
- 在后台调用void UTIL_LPM_EnterLowPower()。

### 4.6.2 接口

**表5. 接口函数**

| 功能 | 说明 |
|---|---|
| UTIL_LPM_ModeSelected_t UTIL_LPM_ReadModeSel(void) | 返回将要应用的低功耗模式。 |
| UTIL_LPM_SetOffMode(UTIL_LPM_bm_t lpm_id_bm, UTIL_LPM_State_t state) | 随时为任何用户启用或禁用关闭模式。 |
| void UTIL_LPM_SetStopMode(UTIL_LPM_bm_t lpm_id_bm, UTIL_LPM_State_t state) | 随时为任何用户启用或禁用停止模式。 |
| void UTIL_LPM_EnterLowPower(void) | 进入选定的低功耗模式。 |
| void UTIL_LPM_EnterSleepMode(void) | 在进入睡眠模式前调用的API。 |
| void UTIL_LPM_ExitSleepMode(void) | 在退出睡眠模式时调用的API。 |
| void UTIL_LPM_EnterStopMode(void) | 在进入停止模式前调用的API。 |
| void UTIL_LPM_ExitStopMode(void); | 在退出停止模式时调用的API。 |
| void UTIL_LPM_EnterOffMode(void) | 在进入关闭模式前调用的API。 |
| void UTIL_LPM_ExitOffMode(void); | 在退出关闭模式时调用的API。只在MCU没有按预期进入关闭模式时调用。 |

## 4.7 闪存管理

STM32WB的CPU1和CPU2共享一个单存储区。在写入或擦除闪存时，无法从闪存中获取指令。

当CPU执行闪存中的代码时，如果启动了写入或擦除操作，它会立即停止。

当CPU执行SRAM中的代码时，在写入或擦除操作进行时CPU不会停止（假定它不读取闪存中的数据）。

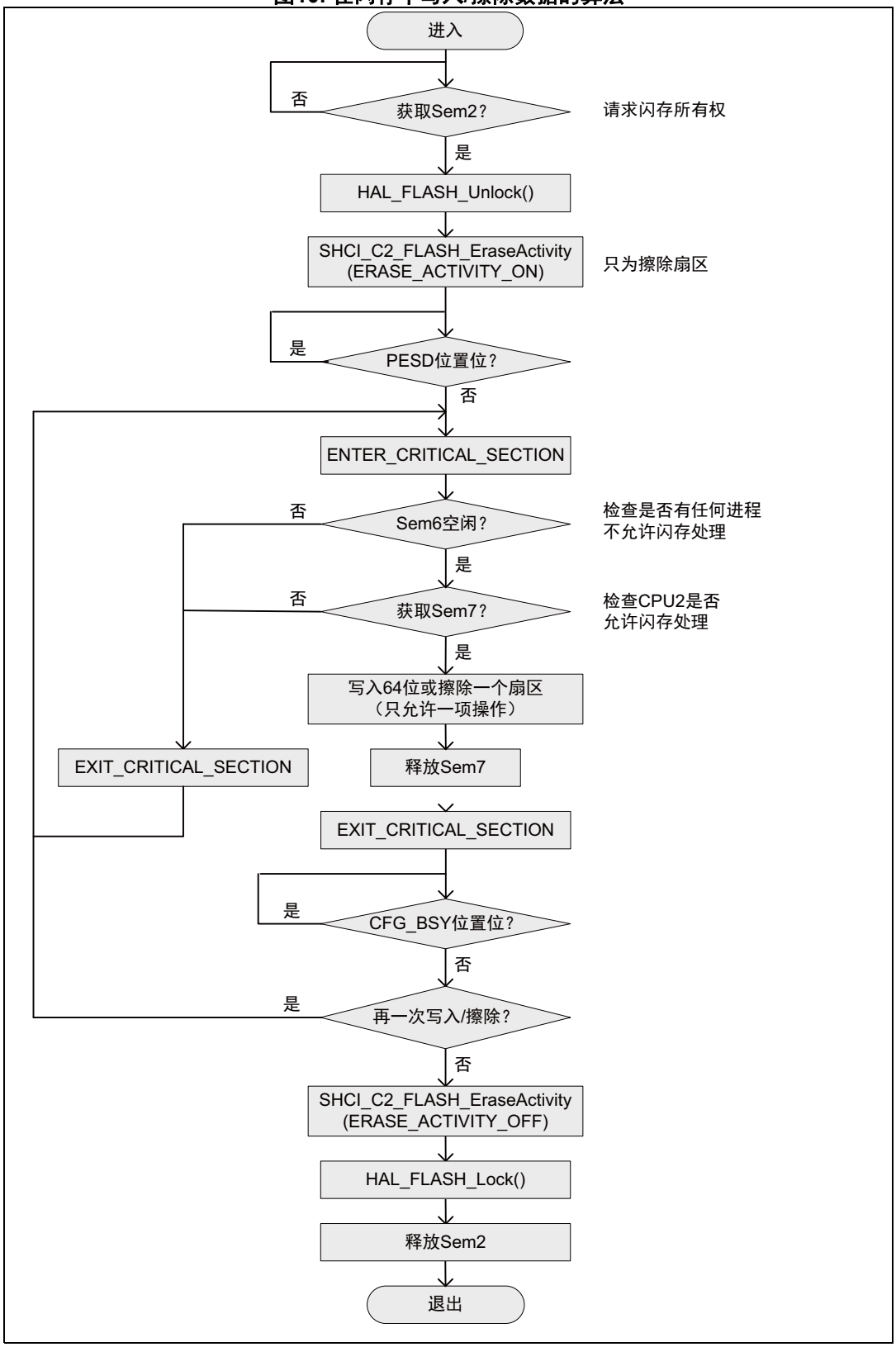
### 4.7.1 CPU2时序保护

出于安全原因，将阻止CPU2执行SRAM中的任何代码。为了保护CPU2时序，它使用Sem7启用或禁用来自CPU1的闪存操作请求。

CPU1上的应用必须实现*图 10*所示的算法才能写入或擦除闪存，还必须在其自身的驱动程序中实现下列操作（在*图 10*定义的临界区之外）：

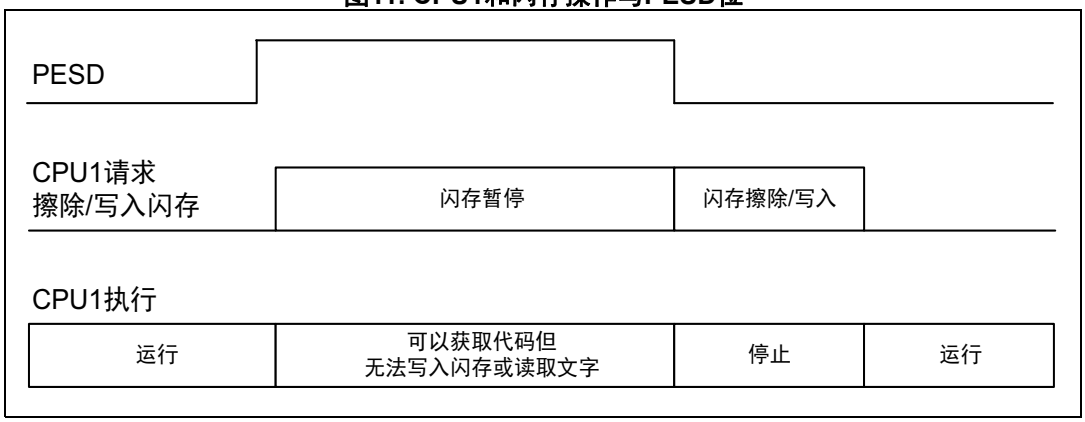
- 在对闪存进行任何访问前获取Sem2，并在不再需要它来访问IP时将其释放
- 当用户驱动程序需要擦除扇区时，必须首先发送命令SHCI_C2_FLASH_EraseActivity(ERASE_ACTIVITY_ON)。在擦除所有相关扇区后，必须发送命令SHCI_C2_FLASH_EraseActivity(ERASE_ACTIVITY_OFF)，参见*第 4.7.1 节*。
- 当CPU2时序保护使用PESD位机制（这是默认情况，参见*第 4.7.1 节*）时，FLASH驱动程序必须轮询FLASH_SR寄存器中的CFGBSY位或回读存储器，直至值为要写入的值。

**图10. 在闪存中写入/擦除数据的算法**



CPU2默认使用PESD位机制（FLASH_SR寄存器）保护其BLE时序，而不是Sem7。尽管Sem7检查是无用的，该算法仍然有效。其缺点在于，如果CPU2在CPU1启动写入或擦除操作的同时将PESD位置位，则CPU1可以获取代码但不能从存储器中读取文字，即使要执行的代码需要此操作。在使用PESD机制时，很难控制CPU1是否停止。此外，CPU2释放PESD位时无中断信号，因此不可能实现异步软件流。

**图11. CPU1和闪存操作与PESD位**




CPU1可通过系统命令SHCI_C2_SetFlashActivityControl()对CPU2使用PESD或Sem7机制保护BLE时序进行配置。虽然可以在任何时间发送此命令，但建议在初始化阶段发送。

CPU2默认保护其时序不受CPU1请求的写操作的影响。当CPU1需要启动擦除操作时，必须先发送系统命令SHCI_C2_FLASH_EraseActivity(ERASE_ACTIVITY_ON)。当其预期不再请求擦除操作时，必须发送系统命令SHCI_C2_FLASH_EraseActivity(ERASE_ACTIVITY_OFF)。不需要为每个擦除操作发送这些命令。建议在请求第一次擦除操作前启用保护，并在执行完最后一次擦除操作后禁用保护。

### 4.7.2 CPU1时序保护

当CPU1需要确保其不会因CPU2请求的闪存操作（写入或擦除）而停止时，它必须占用Sem6。在Sem6被释放前，CPU2不请求任何闪存操作。

当Sem6被占用时，表示CPU2已经在执行闪存操作的过程中（即将开始或刚刚结束）。当需要防止CPU2请求任何闪存操作时，CPU1必须轮询Sem6以获取它。

CPU2使用*图 10*中描述的相同算法。

### 4.7.3 RF活动与闪存管理之间的冲突

即使CPU1不使用Sem6防止CPU2启动闪存操作，仍然有一些用例中的CPU2无法启动擦除操作。

始终可以执行闪存写操作，但在CPU2上的NVM已满时，闪存写请求可能需要擦除操作。在这种情况下，在执行擦除操作前不会写入数据。

当由于CPU1已通过命令SHCI_C2_FLASH_EraseActivity(ERASE_ACTIVITY_ON)发出通知或其自身需要执行闪存擦除操作以擦除NVM中的一些扇区，导致CPU2需要保护其时序不受擦除操作的影响时，从射频活动前25 ms至其结束，期间禁止任何存储器操作。为了在BLE运行时执行闪存擦除操作，应用必须确保射频闲置时间长于25 ms。

- BLE广播：广播间隔必须长于25 ms + 广播数据包长度，才能执行闪存擦除操作。
- BLE连接：连接间隔必须长于25 ms + 数据包长度，才能执行闪存擦除操作。
- 数据吞吐量用例：在发送数据流数据包时，射频保持激活状态，以便在两个连接间隔之间尽可能多地发送数据。因此，射频的闲置时间可能不足以执行擦除操作。当设备为主设备时，可以减小连接事件长度参数（通过aci_gap_create_connection()或aci_gap_start_connection_update()命令），以防止设备将两个连接间隔之间的时间间隔全部占用。当设备为从设备时，必须请求主设备延长连接间隔，以使数据发送只占用两个连接事件之间时间间隔的一部分。

只有在以下情况下，CPU2才需要在闪存中写入数据：

- 在配对期后保存安全信息
- 在断开后保存刚断开的客户端的GATT客户端描述符信息（若已绑定）。

## 4.8 CPU中的调试信息

### 4.8.1 GPIO

可通过GPIO输出CPU2的大多数实时活动，例如后台任务、中断处理函数和BLE IP Core信号。除通过硬件驱动的BLE IP Core信号外，特定GPIO的信号分配完全可以从CPU1侧进行配置。因此，BLE IP Core GPIO只有在不被应用使用时才必须启用。在\Core\Src目录下的文件app_debug.c中完成每个应用的完整配置。

#### HW 信号

aRfConfigList[]表包含由硬件（取决于射频活动）驱动的GPIO列表。需要监测每个信号的四个参数：

```
{ GPIOA, LL_GPIO_PIN_9, 0, 0},     /* DTB13 - Tx/Rx Start */
```

前两个参数定义使用的GPIO（在本例中，PA9用于输出DTB13）。这两个参数不能修改。为了监测信号，板上必须有相关的GPIO。

第三个参数用于启用（1）或禁用（0）信号。所有信号默认置为0。

第四个参数未使用，应保持0不变。

为了监测相关GPIO上的信号，必须将第三个参数置为1并将文件顶部的BLE_DTB_CFG编译器开关置为7。

最有用的信号是 DTB13，它塑造了所有的射频活动。

#### SW 信号

aGpioConfigList []表保存了由软件驱动的GPIO列表。需要监测每个信号的四个参数：
{ GPIOA, LL_GPIO_PIN_0, 0, 0}, /* BLE_ISR - Set on Entry / Reset on Exit */

前两个参数定义用于输出信号的GPIO。这些是完全可配置的参数。用户可以选择应用中未使用的任何GPIO。

第三个参数用于启用（1）或禁用（0）信号。所有信号默认置为0。

第四个参数未使用且必须保持0不变。

为了监测相关GPIO上的一个信号，必须将第三个参数置为1。

### 4.8.2 SRAM2

#### Hardfault

当CPU2进入HardFault中断处理函数时，它可以在运行无限循环前输出不同信息。

它可以设置GPIO（如果已在app_debug.c - aGpioConfigList []中启用）。

它在SRAM2A中写入下列数据：

| | |
|---|---|
| @SRAM2A_BASE: 0x1170FD0F | 识别HardFault问题的关键字 |
| @SRAM2A_BASE + 4 | 生成HardFault的程序计数器值 |
| @SRAM2A_BASE + 8 | 执行生成HardFault的指令时的链接寄存器值 |
| @SRAM2A_BASE + 12 | 执行生成HardFault的指令时的栈指针值 |

#### 安全攻击

当提供给CPU2用于通过信箱（Mailbox）交换数据的缓冲区不在不安全SRAM2中时，CPU2进入无限循环并在SRAM2A_BASE处写入密码0x3DE96F61。

## 4.9 FreeRTOS低功耗

无论 CPU2上运行的是什么无线协议栈，FreeRTOS低功耗模式在 CPU1上为所有无线应用程序共享相同的实现。

HAL定时器被映射到TIM17，不会与为FreeRTOS保留的系统定时器发生冲突。\Applications\BLE\BLE_HeartRateFreeRTOS\Core\Src中的文件stm32wbxx_hal_timebase_tim.c实现以下HAL函数：

- HAL_InitTick()
- HAL_SuspendTick()
- HAL_ResumeTick()

在main.c中实现TIM17用户中断处理函数HAL_TIM_PeriodElapsedCallback()，使用HAL让定时器时间片数递增。可以选择另一个定时器自定义实现。

当FreeRTOS处于空闲模式时，系统定时器关闭并被低功耗定时器取代。\Applications\BLE\BLE_HeartRateFreeRTOS\Core\Src中的文件freertos_port.c实现无时间片模式

- 重新实现vPortSuppressTicksAndSleep()以支持无时间片模式（基于STM32WB提供的低功耗模式）
- 重新实现vPortSetupTimerInterrupt()以启动STM32WB提供的低功耗定时器

当前的实现是使用运行在 RTC 上的 Timer 服务器。可通过重新实现下列函数更改定时器选择：

- LpTimerInit()，用于初始化要使用的低功耗定时器。
- LpTimerCb() 在不仅仅是需要唤醒时使用。在当前的实现中，唤醒时完成的所有操作都是在 vPortSuppressTicksAndSleep() 中退出低功耗模式时实现的，而不是在计时器回调中实现。
- LpTimerStart()，用于在进入低功耗模式前启动低功耗定时器。
- LpGetElapsedTime()，用于返回系统处于低功耗模式的时间。在通过vTaskStepTick()更新FreeRTOS使用的系统定时器时需要用到。

通过LpEnter()进入低功耗模式。当前实现基于所有BLE应用（无论它们是否基于FreeRTOS）中使用的低功耗管理器。LpEnter() 的实现可以自定义。

### BLE

后台要调用的函数数量取决于应用，应用还决定了是从专用任务还是一个共用任务调用每个函数。BLE架构支持上面任何的组合。

任何BLE应用都必须在任务中调用至少两个函数：

- hci_user_evt_proc()：当中间件调用hci_notify_asynch_evt()时，这个函数必须在后台调用。hci_user_evt_proc() 不能在 hci_notify_asynch_evt() 内部调用，因为它可能被 IPCC 中断上下文中调用。在从中间件调用hci_notify_asynch_evt()到在后台调用hci_user_evt_proc()之间，没有时序限制。但是，在某些数据吞吐量用例中，当时间足够短以按通知事件的相同速率读取事件时，性能会更好。当接收到多个 hci_notify_asynch_evt() 时，hci_user_evt_proc() 函数只需要从后台调用一次。 但多调用几次也没问题。hci_user_evt_proc() 来自后台，而 hci_notify_asynch_evt() 只有一次通知或没有通知。

- **shci_user_evt_proc()**：要求与hci_user_evt_proc()和相关通知shci_notify_asynch_evt()相同。请注意，此系统通道上当前没有数据吞吐量。

如果在已有一个挂起的BLE命令时无法发送BLE命令，或在已有一个挂起的系统命令时无法发送系统命令，中间件将提供hook函数，以使应用能够实现信号量机制。

在从中间件调用hci_cmd_resp_wait()时，必须获取信号量，并在收到hci_cmd_resp_release()时释放信号量。在释放信号量前，应用不得从hci_cmd_resp_wait()返回。

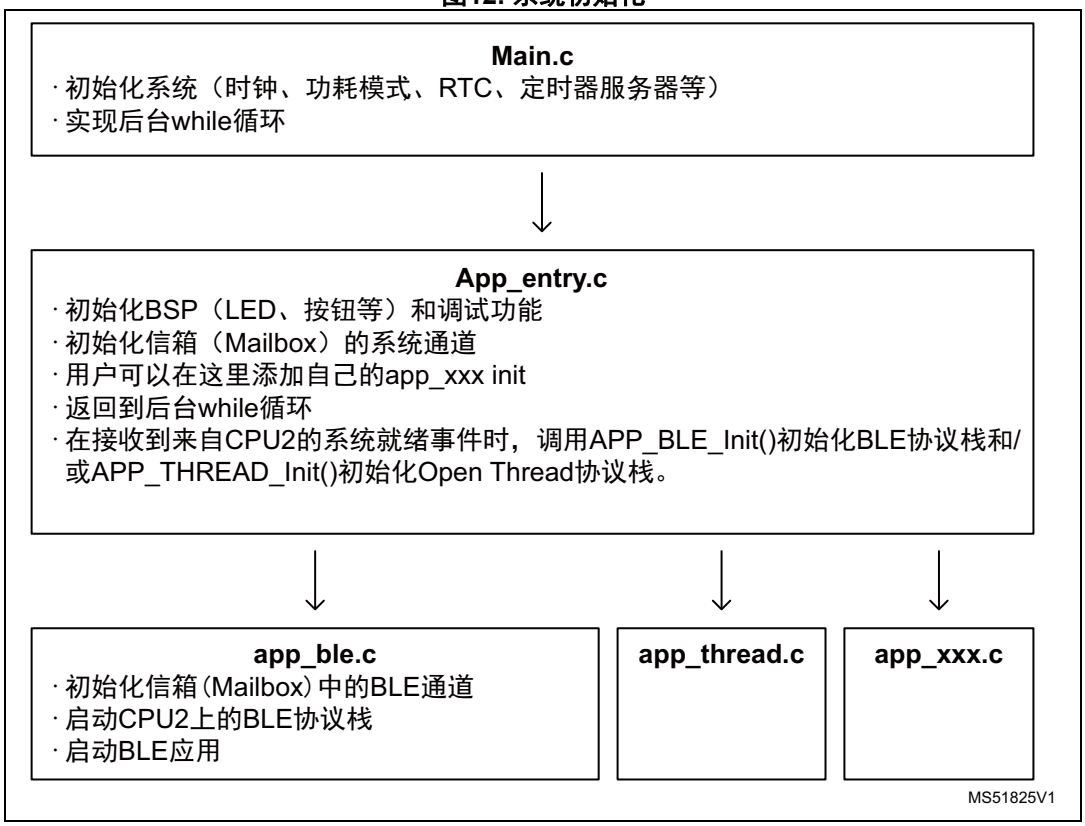
必须使用另一个信号量来处理系统通道上与shci_cmd_resp_wait()/shci_cmd_resp_release() 相同的机制。

# 5 系统初始化

所有应用都从三组文件开始（参见*图 12*）：

1. main.c：包含所有应用程序通用的硬件配置（提供给CPU2的时钟必须始终为32MHz）
2. app_entry.c：任何应用共用的所有软件配置和实现
3. app_ble.c / app_thread.c / app_xxx.c: 应用程序专用文件

**图12. 系统初始化**



# 6 BLE应用的分步设计

本章提供关于如何在STM32WB上设计和实现BLE应用的信息和代码示例。

## 6.1 初始化阶段

应用初始化有几个必要步骤。

- 初始化设备（HAL、复位设备、时钟和电源配置）
- 配置平台（按钮、LED）
- 配置硬件（UART、调试）
- 配置BLE设备公共地址（如使用）：
  – aci_hal_write_config_data() API
- 配置发送功率
  – aci_hal_set_tx_power_level() API
- 初始化BLE GATT层：
  – aci_gatt_init() API
- 根据选择的设备角色初始化BLE GAP层
  – aci_gap_init(“role”)API
- 设置合适的安全I/O能力和验证要求（如果使用了BLE安全）
  – aci_gap_set_io_capability() and aci_gap_set_authentication_requirement() APIs
- 如果设备是GATT服务器，定义需要的服务、特征和特征描述符。
  – aci_gatt_add_service(), aci_gatt_add_char(), aci_gatt_add_char_desc() APIs
- 使用调度器管理任务和低功耗

## 6.2 广播阶段（GAP外围设备）

为了在BLE GAP中央（主）设备和BLE GAP外围（从）设备之间建立连接，必须在外围设备上发起GAP可发现模式。可使用*表 6*中的API。

表6. 广播阶段API描述

| API名称 | 说明 |
|---|---|
| aci_gap_set_discoverable() | 将设备设置为一般可发现模式。<br>设备处于可发现模式，直至设备调用aci_gap_set_non_discoverable() API。 |
| aci_gap_set_limited_discoverable() | 将设备设置为有限可发现模式。设备处于可发现模式的最长时间为TGAP (lim_adv_timeout) = 180秒。<br>随时可通过调用aci_gap_set_non_discoverable() API禁用广播。 |
| aci_gap_set_direct_connectable() | 将设备设置为定向可连接模式。设备处于可直接连接模式的时间只有1.28秒。 如果在此期间没有建立连接，设备将进入不可发现模式，需要重新启动广播才可被发现。 |
| aci_gap_set_non_connectable() | 使设备进入不可连接模式。 |
| aci_gap_set_undirect_connectable() | 使设备进入非定向可连接模式。 |

## 6.3 可发现和可连接阶段（GAP中央设备）

为了在两个设备之间建立连接，GAP中央设备可以发现远程设备，然后向目标设备发起连接。此外，还可以向指定设备发起直接连接。

*表 7*中列出了可用于GAP发现流程的API。

表7. GAP中央设备API

| API | 说明 |
|---|---|
| aci_gap_start_limited_discovery_proc () | 启动有限发现流程。控制器开始主动扫描。只有处于有限可发现模式的设备返回上层。 |
| aci_gap_start_general_discovery_proc () | 启动一般发现流程。控制器开始主动扫描。 |
| 下列API可以用在建立GAP连接的流程中。 | |
| aci_gap_start_auto_connection_establish_proc () | 启动自动连接流程。指定的设备被添加到控制器白名单中，并使用“使用白名单来确定要连接到哪个BLE广播”的发起者过滤策略，发起对 GAP 控制器的连接调用。 |
| aci_gap_create_connection () | 启动直接连接流程。GAP 向控制器发起创建连接调用，发起者过滤策略设置为“忽略白名单并仅处理指定设备的可连接广播数据包”。 |
| aci_gap_start_auto_connection_establish_proc () | 启动自动连接流程。指定的设备被添加到控制器白名单中，并且GAP向控制器发出创建连接调用，发起者过滤策略设置为“使用白名单来确定要连接到哪个广播设备”。 |
| aci_gap_start_general_connection_establish_proc() | 启动通用连接流程。在scanner过滤策略设置为“接受所有广播数据包”的情况下，设备启用控制器扫描，并使用事件callback hci_le_advertising_report_event()将扫描结果中的所有设备发送至上层。 |

表7. GAP中央设备API（续）

| API | 说明 |
|---|---|
| aci_gap_start_selective_connection_establish_proc () | 启动选择性连接流程。GAP将指定设备地址添加到白名单中，并在scanner过滤策略设置为“仅接受来自白名单中的设备的数据包”时启用控制器扫描。通过事件回调hci_le_advertising_report_event()将找到的所有设备发送至上层。 |
| aci_gap_terminate_gap_proc() | 终止指定的GAP流程。 |

## 6.4 服务和特征配置（GATT服务器）

为了添加服务及其相关特征，用户应用从下面的两种配置文件中选择一种：

- 标准配置文件由蓝牙SIG定义。
  用户必须遵循配置文件规范和服务及特征规范文档，以便使用定义的相关配置文件、服务和特征16位UUID进行实现（请参考蓝牙SIG网页）。
- 专有的非标准配置文件。
  用户必须定义自定义服务和特征。在本例中，需要128位UIDS，并且必须由配置文件实现程序生成（请参考UUID生成器网页：www.famkruithof.net）。

可使用以下流程添加服务：

```
aci_gatt_add_service(uint8_t Service_UUID_Type,
Service_UUID_t *Service_UUID,
uint8_t Service_Type,
uint8_t Max_Attribute_Records,
uint16_t *Service_Handle);
```

该流程返回服务句柄的指针（Service_Handle），用于标识用户应用中的该服务。可使用以下流程为该服务添加特征：

```
aci_gatt_add_char(uint16_t Service_Handle,
uint8_t Char_UUID_Type,
Char_UUID_t *Char_UUID,
uint8_t Char_Value_Length,
uint8_t Char_Properties,
uint8_t Security_Permissions,
uint8_t GATT_Evt_Mask,
uint8_t Enc_Key_Size,
uint8_t Is_Variable,
uint16_t *Char_Handle);
```

该流程返回特征句柄的指针（Char_Handle），用于标识用户应用中的该特征。

如果特征所有者处于通知或指示模式并已启用，则GATT服务器端必须使用以下API向GATT客户端发送通知或指示。

aci_gatt_update_char_value()

## 6.5 服务和特征发现（GATT客户端）

在两个设备连接成功后，应用数据交换将基于GATT客户端-服务器架构。

一个设备必须实现F并删除GATT客户端。

GATT客户端使用下列API发现服务和特征，启用/禁用GATT服务器的通知/指示，写入/读取特征，以及确认GATT服务器指示。

**表8. GATT客户端API**

| API | 说明 |
|---|---|
| aci_gatt_disc_all_primary_services () | 启动GATT客户端流程以发现GATT服务器上的所有主要服务。在GATT客户端连接了设备，并想要寻找设备上提供的所有主要服务以确定其功能时使用。<br>通过aci_att_read_by_group_type_resp_event()事件回调提供流程响应。 |
| aci_gatt_disc_primary_service_by_uuid() | 启动GATT客户端流程，以使用其UUID发现GATT服务器上的主要服务。在GATT客户端连接了设备并想要寻找特定服务而无需获取任何其他服务时使用。<br>通过aci_att_find_by_type_value_resp_event()事件回调提供流程响应。 |
| aci_gatt_find_included_services() | 启动寻找所有已包含服务的流程。在GATT客户端已发现主要服务并想要发现次要服务时使用。<br>通过aci_att_read_by_type_resp_event()事件回调提供流程响应。 |
| aci_gatt_disc_all_char_of_service() | 启动发现给定服务的所有特征的GATT流程。<br>通过aci_att_read_by_type_resp_event()事件回调提供流程响应。 |
| aci_gatt_disc_char_by_uuid() | 启动发现UUID指定的所有特征的GATT流程。<br>通过aci_gatt_disc_read_char_by_uuid_resp_event()事件回调提供流程响应。 |
| aci_gatt_disc_all_char_desc() | 启动发现GATT服务器上所有特征描述符的流程。<br>通过aci_att_find_info_resp_event()事件回调提供响应。 |

对于所有指令，通过aci_gatt_proc_complete_event()事件回调指示流程结束。

## 6.6 安全（配对和绑定）

BLE安全模型包含5个安全特性：

1. 配对：创建一个或更多共享密钥的过程。
2. 绑定：为了构成可信设备对，保存配对期间创建的密钥以便在后续连接中使用的行为。
3. 设备验证：确保 两个设备具有相同密钥的确认过程。
4. 加密：提供信息保密性。
5. 信息完整性：防止伪造信息（4字节信息完整性检查或MIC）

BLE使用四种配对方法：

1. 直接工作（Just work）
2. 带外（Out of band）
3. 密钥输入（Passkey entry）
4. 数字比较（Numeric comparison）（仅安全连接）（自蓝牙4.2起）

安全密钥计算的确定方法：

- 传统加密-短期临时密钥（STK）。STK创建用于连接加密。然后，如果绑定，LTK将用于后续的连接。
- 安全连接 - 长期密钥（LTK）。LTK创建用于连接加密。

### 6.6.1 安全模式和级别

LE安全模式1（链路层）：

- 无安全保护 - 1级
- 未验证的加密配对 - 2级
- 已验证的加密配对 - 3级
- 已验证的LE安全连接BT 4.2加密配对 - 4级

已验证的配对：在有中间人（MITM）保护的情况下执行配对

未验证的配对：在没有MITM保护的情况下执行配对

LE安全模式2（ATT层）：不支持

- 有数据签名的未验证配对
- 有数据签名的已验证配对

### 6.6.2 安全命令

在设备初始化阶段，可以使用下列命令初始化安全属性：

aci_gap_set_io_capability()

设置设备的IO能力。只能在设备未处于连接状态时发出该指令。

aci_gap_set_authentication_requirement()

为设备设置验证要求。只能在设备未处于连接状态时发出该指令。

此命令定义绑定模式信息、MITM模式、LE安全连接支持值、按键通知支持值、密钥大小、是否使用固定引脚、其值和身份地址类型。

- SC_Support参数定义LE安全连接支持值。
  - 0x00：不支持安全连接配对（传统配对模式）
  - 0x01：支持安全连接配对，但是可选的
  - 0x02：支持和强制安全连接配对（仅限SC模式），并强制要求

建立连接后，可以启动安全流程：

- 由主设备通过aci_gap_set_pairing_req()启动
  - 发送启动配对过程的SM配对请求。在发出该命令前，必须设置验证要求和IO能力。
  - force_rebond参数值决定了是否发送配对请求，即使设备之前已绑定。
- 由从设备通过aci_gap_slave_security_req()启动
  - 向主设备发送从设备安全请求。必须发出该指令以将从设备的安全要求通知主设备。主设备可以加密链路、启动配对流程或拒绝请求。
  - 在配对过程完成后返回aci_gap_pairing_complete_event事件。

根据SC_Support参数值，设备用*表 9*中列出的命令之一回复安全请求。

**表9. 安全命令**

| 指令 | 说明 |
|---|---|
| aci_gap_pass_key_resp() | 该指令必须由主机发送,以响应aci_gap_pass_key_req_event事件。<br>命令参数包含配对过程中使用的密钥（如果没有固定引脚）。 |
| aci_gap_numeric_comparison_value_confirm_yesno() | 此命令允许用户确认或不确认通过aci_gap_numeric_comparison_value_event 显示的数字比较值<br>当设备被绑定时，密钥存储在非易失性存储区（在BLE链路断开后）。这意味着如果设备之前已绑定且其中一个设备未插电，密钥也不会丢失。<br>如果发送命令aci_gap_set_pairing_req()时force_rebond参数被设置为不强制重新绑定，则无需进行任何其他交换即可完成配对。 |
| 清空安全数据库: | |
| aci_gap_clear_security_db() | 安全数据库中的所有设备都将被删除。 |

### 6.6.3 安全信息命令

**表10. 安全信息命令**

| 指令 | 说明 |
|---|---|
| aci_gap_get_bonded_devices() | 该指令获取已绑定设备的列表。它返回地址数量和对应的地址类型及值。 |
| aci_gap_is_device_bonded() | 该指令检查指令中指定地址所属的设备是否已绑定。如果设备使用可解析私有地址并已绑定，指令将返回ble_status_success。 |
| aci_gap_get_security_level() | 该指令可用于获取设备的当前安全设置。 |
| 如果支持按键通知，使用： | |
| aci_gap_passkey_input() | 此命令允许告诉协议栈检测到的在密码输入期间的输入类型。 |
| 如果支持OOB，使用： | |
| aci_gap_set_oob_data() | 该指令由用户发送，以输入通过OOB通信到达的OOB数据。 |
| aci_gap_get_oob_data() | 此指令由用户发送，用于获取（从协议栈中提取）由协议栈自身生成的OOB数据。 |

## 6.7 隐私特性

BLE隐私特性通过频繁地更改设备地址，降低了在一段时间内跟踪设备的能力。

使用Privacy模式的设备地址可使用IRK（身份解析密钥）进行解析，IRK是配对过程中交换的解析密钥之一。

首先需要在隐私禁用时初始化设备，然后进行连接和配对。

然后，在两个设备上均启用隐私的情况下，在两端发送：

Hci_reset()

aci_gap_init() - 隐私启用

aci_gap_add_devices_to_resolving_list()

此指令用于将一台设备添加到用于解析控制器中可解析私有地址的地址转换列表中。

从中央设备侧，发送：

aci_gap_create_connection()

或

aci_gap_start_auto_connection_establish_proc()

或

aci_gap_start_general_connection_establish_proc() (then aci_gap_create_connection) :
own_address_type = resolvable private address, peer_address_type = public or random

从外围设备侧，发送：

aci_gap_set_discoverable()

或

aci_gap_set_direct_connectable()

或

aci_gap_set_undirected_connectable()

own_address_type = 可解析私有地址

在建立连接后，生成LE增强连接完成事件。

## 6.8 如何使用2 Mbps特性

在设备初始化阶段，可使用以下命令初始化首选TX_PHYS、RX_PHYS值：

**表11. 2 Mbps特性的命令**

| 指令 | 说明 |
|---|---|
| HCI_LE_Set_default_Phy() | 指定要实现的用于接收和发送的首选PHY（没有建立连接）。默认的首选PHY为2M。 |
| 在建立连接后（1M），每个设备都可以发送用于发送和接收的首选PHY： | |
| HCI_LE_Set_Phy() | 允许主机为连接指定首选值。 |
| 在连接期间，可以读取使用的RX或TX PHY： | |
| HCI_LE_Read_Phy() | 读取连接的当前PHY TX和RX。 |

当使用命令HCI_LE_Set_Phy()时，主设备接收到事件hci_le_phy_update_complete。

## 6.9 如何更新连接参数

在建立连接后，可以更新连接参数。

**表12. 专有连接数据**

| 指令 | 说明 |
|---|---|
| 当主设备（中央设备）是更新发起方时： | |
| aci_gap_start_connection_update() | 启动连接更新（仅当角色为主设备时）。当流程完成时，向上层返回HCI_LE_CONNECTION_UPDATE_COMPLETE_EVENT事件。 |
| 当从设备（外围设备）是更新发起方时： | |
| aci_l2cap_connection_parameter_update_req() | 从从设备向主设备发送L2CAP连接参数更新请求。当主设备响应请求（接受或拒绝）时，将生成HCI_L2CAP_CONNECTION_UPDATE_RESP_EVENT事件。 |

## 6.10 写入或读取长本地或远程值

### 6.10.1 写入长远程值

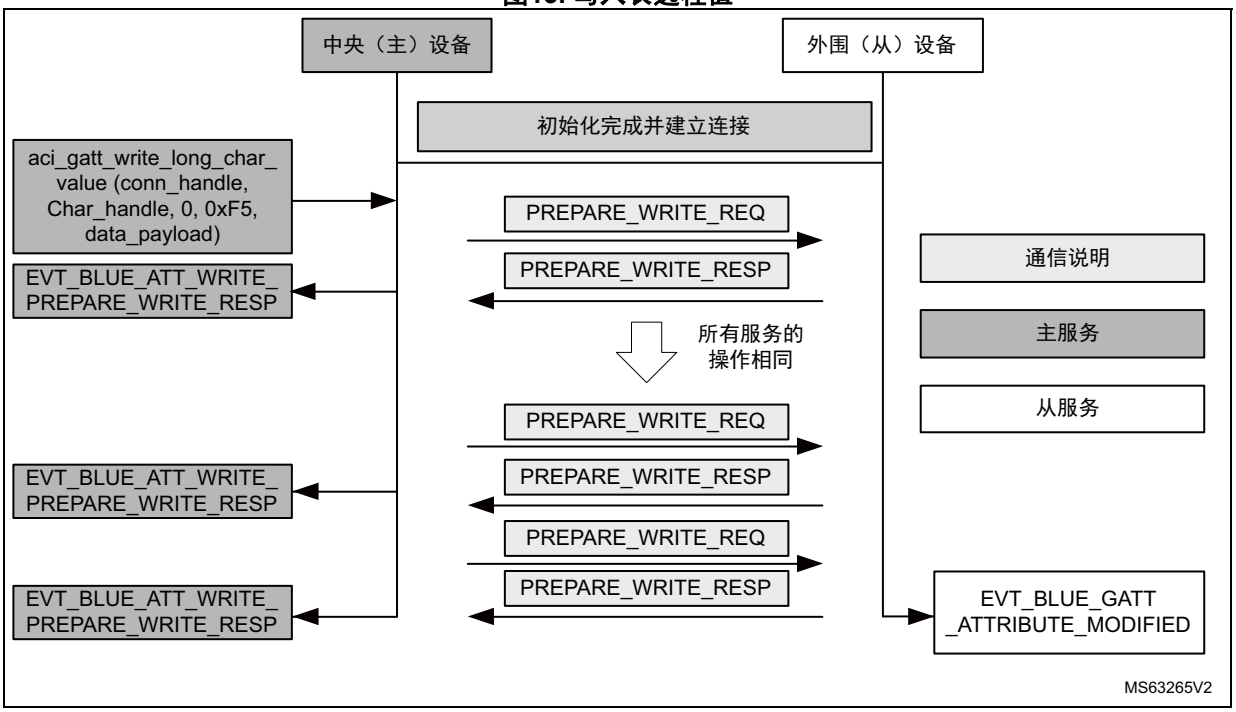
aci_gatt_write_long_char_value(conn_handle, RxCharHandle,offset, chunk_size, (&payload)+offset)

最大数据写入长度为243字节。

在服务器端，创建服务aci_gatt_add_service。

创建特征aci_gatt_add_char，使最大长度=0xFF且属性=CHAR_PROP_WRITE，如*图 13*所示。

**图13. 写入长远程值**

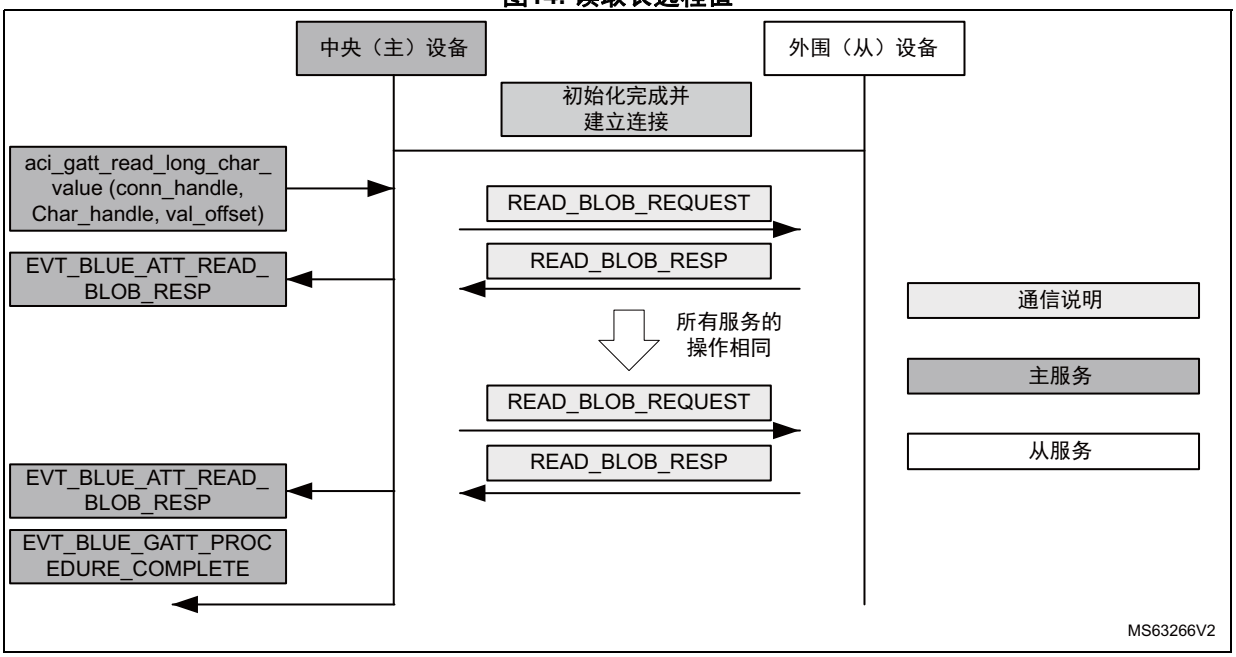


### 6.10.2 读取长远程值

aci_gatt_read_long_char_value(conn_handle, RxCharHandle, offset)

最大数据读取长度为243字节，如*图 14*所示。

**图14. 读取长远程值**



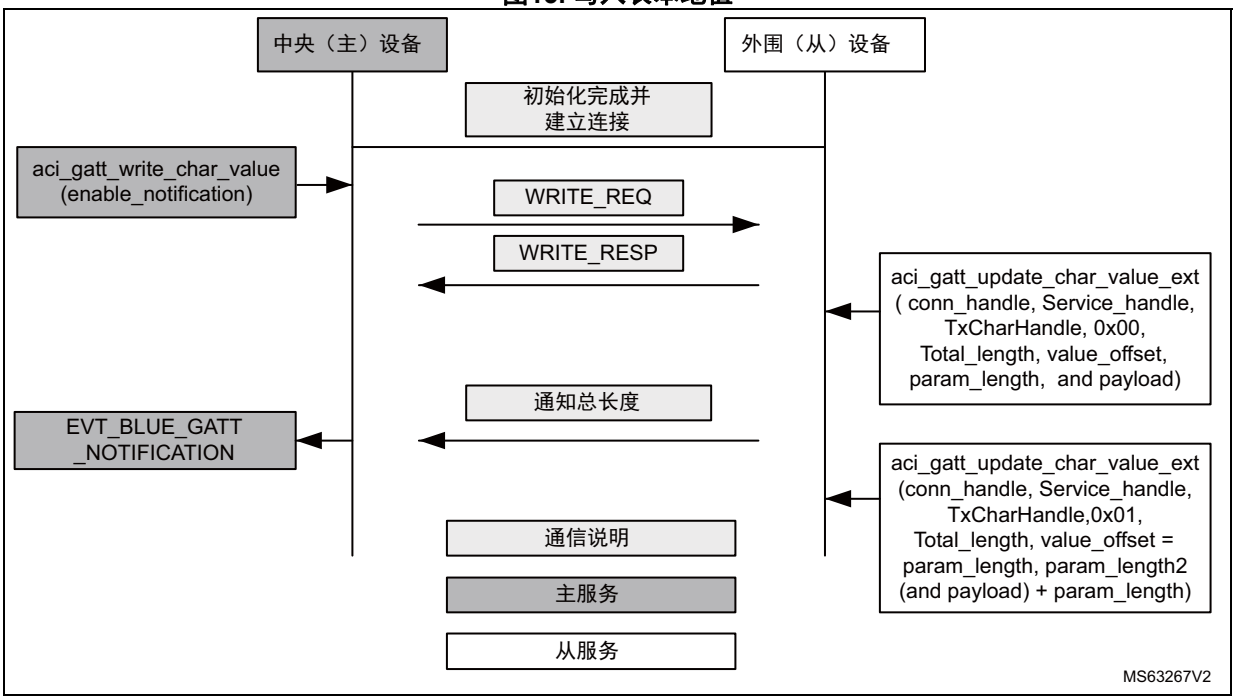
### 6.10.3 写入长本地值

aci_gatt_update_char_value_ext(conn_handle_to_notify, service_handle, char_handle, update_type, char_length, value_offset, value_length, value)

最大数据写入长度为243字节。

如果总长度不超过（ATT_MTU – 3），则发送所有数据，如*图 15*所示。

**图15. 写入长本地值**

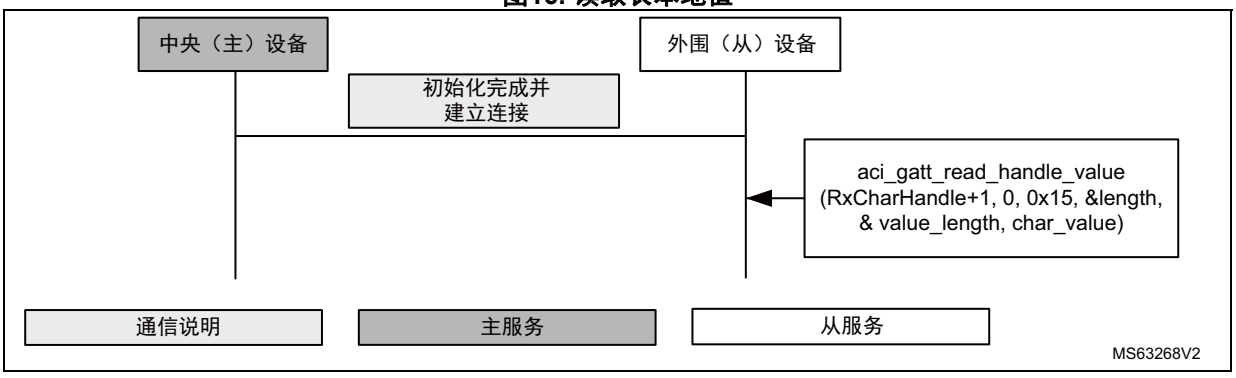



### 6.10.4 读取长本地值

aci_gatt_read_handle_value(RxCharHandle+1, offset, value_length_requested,&length,&value_length,&value)

Value_length_requested不能大于243，如果一个命令不能读取243个以上的字节，必须使用补偿（参见*图 16*）。

**图16. 读取长本地值**



## 6.11 事件和错误代码说明

在调用BLE协议栈API时，获取API返回状态并监测和追踪任何潜在错误状态。

当API成功执行时，返回BLE_STATUS_SUCCESS (0x00)。

通过hci_command_status_event()确认所有命令（HCI - ACI）。

指令状态事件用于指示已接收到Command_Opcode参数描述的指令，并且BLE协议栈控制器当前正在执行此指令的任务。

对于任何问题，状态事件参数都包含了相应的错误代码（参见*[7]*的v5.0第2卷 D部分）。

在GATT客户端，GATT发现流程会因多种原因而失败。在接收到来自GATT服务器的错误响应时生成aci_gatt_error_resp_event()。这并不意味着流程结束并出错，而是该错误是流程本身的一部分。

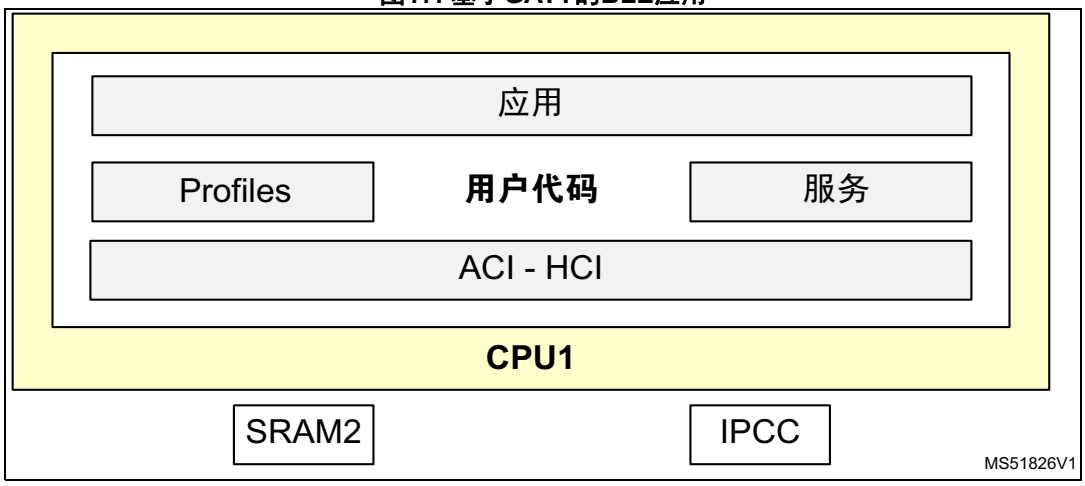
所有 GATT 客户端程序都必须在 aci_gatt_proc_complete_event() 上以成功或发生错误的方式完成，，这样GATT客户端才能启动新的流程。

# 7 基于BT-SIG和专有GATT的BLE应用

本章描述在CPU1上运行的下列应用的规范和实现：

- STM特定的应用
  - 透传模式 - 直接测试模式
- 基于BT-SIG GATT的应用
  - 心率传感器
- 基于ST专有GATT的应用
  - P2P应用（服务器/客户端）
  - FUOTA

**图17. 基于GATT的BLE应用**




## 7.1 透传模式 - 直接测试模式（DTM）

### 7.1.1 目的和范围

为了按照蓝牙规范核心v5.0低功耗控制器卷所述启用直接测试模式（DTM），使用了HCI命令集中的一个命令子集。

DTM用于控制DUT和向测试仪提供报告。根据规范，必须使用下列两种方法中的一种设置DTM：

1. 通过HCI（在STM32WB器件上实现的HCI）
2. 通过2线UART接口。

依据蓝牙核心规范v5.0 [第6卷F部分]， STM32WB支持DTM。

下面是完全符合规范的HCI测试命令：

- HCI_LE_Transmitter_Test
- HCI_LE_Enhanced_Transmitter_Test
- HCI_LE_Receiver_Test
- HCI_LE_Enhanced_Receiver_Test
- HCI_LE_Test_End

接收到的测试数据包数量是函数HCI_LE_Test_End的返回值。

*表 13*中列出了其他可用的函数。

**表13. 直接测试模式函数**

| 功能 | 说明 |
|---|---|
| aci_hal_le_tx_test_packet_number | 此命令提供DTM测试期间发送的测试数据包数。 |
| aci_hal_set_tx_power_level | 此命令用于设置发送输出功率。ACI命令集（参见*[3]*）包含发送功率水平值的完整说明。 |
| aci_hal_tone_start | 此命令用于从STM32WB射频生成连续波形（CW）。 |
| aci_hal_tone_stop | 此命令用于终止CW发射。 |

以下命令序列是从STM32WB射频启用CW传输的典型流程。

hci_reset

aci_hal_set_tx_power_level

aci_hal_tone_start

aci_hal_tone_stop

### 7.1.2 透传模式应用原则

此固件用于：

- 通过UART RX接收命令
- 通过UART TX发送命令
- 通过IPCC与BLE协议栈通信。

CPU1应用程序固件不进行任何解释。

一组命令/事件必须经过STM32WB UART，才能通过透传模式应用控制BLE协议栈。

电平转换器、VCP ST-LINK或应用VCP可用于管理发送和接收。

### 7.1.3 配置

STM32WB可以被视为使用2线UART接口（TXD、RXD）。要使用的CPU1应用为“Ble_TransparentMode”。此固件不解释命令和事件，而是只通过IPCC和选定UART接口（USART1或LPUART1）与BLE协议栈通信。

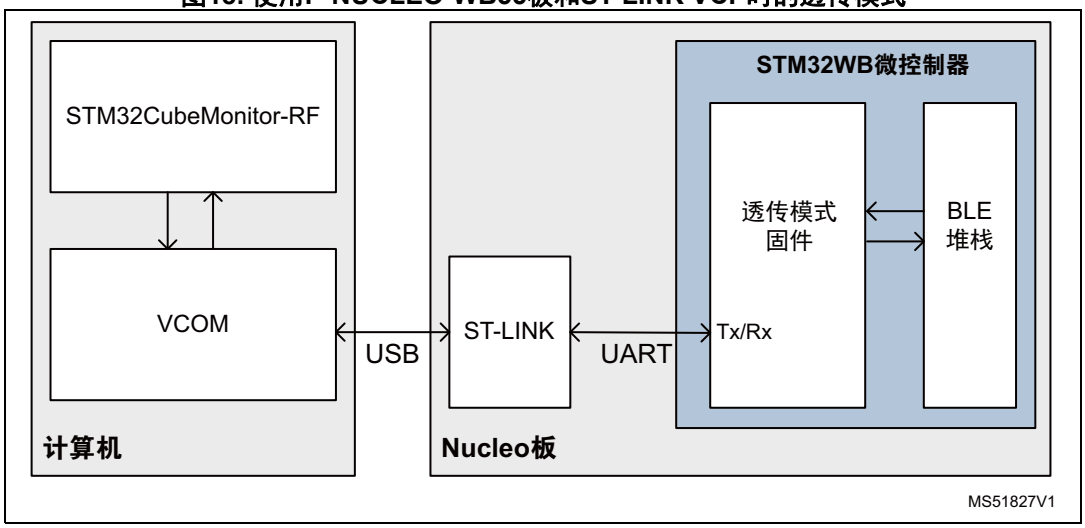
使用app_conf.h完成UART接口和配置选择：
#define CFG_UART_GUI hw_uart1

P-NUCLEO-WB55板包含具有虚拟COM端口能力的ST-LINK。

以下项目被配置为通过ST-LINK的VCP通信：
\Projects\ NUCLEO-WB55.Nucleo\Applications\BLE\Ble_TransparentMode.

UART1（PB6、PB7）连接到P-NUCLEO-WB55板ST-LINK VCP。

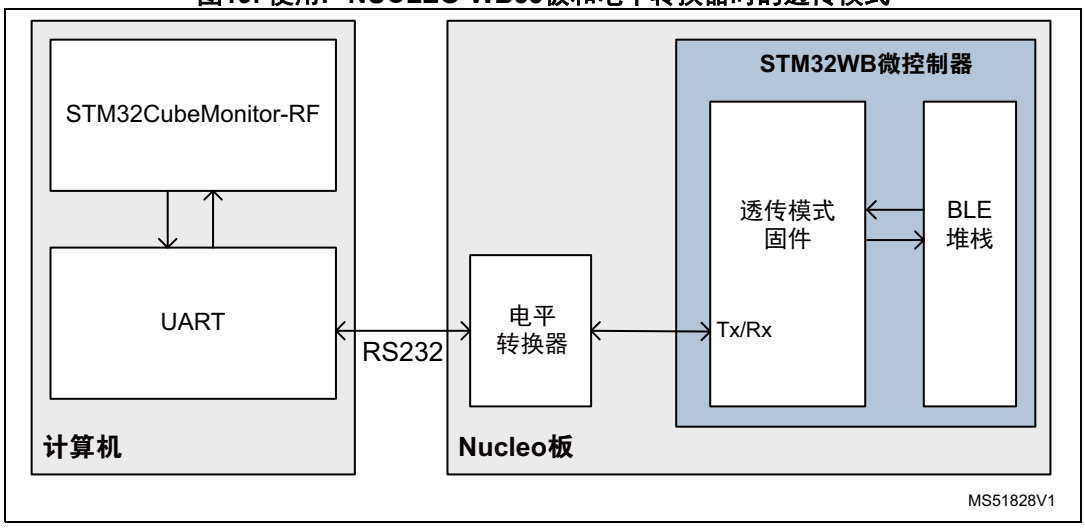
**图18. 使用P-NUCLEO-WB55板和ST-LINK VCP时的透传模式**




P-NUCLEO-WB55 dongle板没有提供ST-LINK。
除了透传模式，项目.\Projects\ NUCLEO-WB55.USBDongle\Applications\BLE\BLE_TransparentModeVCP还包含虚拟COM端口实现。

此外，还可以通过电平转换器（NUCLEO-WB55RG板上未包含）将STM32WB UART接口直接连接到RS232串行通信接口。此方法可用于连接RF测试仪等设备。

图19. 使用P-NUCLEO-WB55板和电平转换器时的透传模式




### 7.1.4 RF认证 - 应用实现

直接测试模式 (DTM) 由蓝牙 SIG 指定，为 BLE 设备提供不同的射频测试选择，包括 USB 或 RS232 接口的远程控制命令。

BLE RF可设置为连续发送或接收模式、有或没有RF调制的评估测试。 *图 20*描述了简单设置。

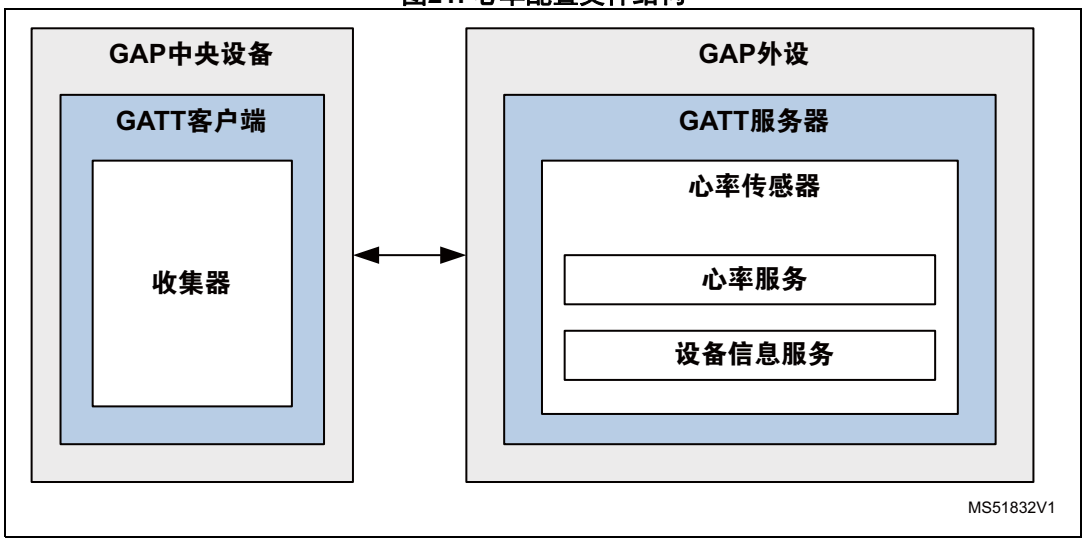
图20. 使用BLE RF测试仪和P-NUCLEO板时的简单设置


## 7.2 心率传感器应用

心率配置文件包含两项操作：

- 收集器：GAP中央设备和GATT客户端接收心率测量值和其他数据
- 心率传感器：GAP外设和GATT服务器提供心率测量值和其他数据。

图21. 心率配置文件结构




STM32WBCube_FW_WB_V1.0.0版附带心率传感器的示例。

本节将描述创建蓝牙SIG心率传感器应用的步骤，该应用旨在每秒从传感器（用于健身应用）发送心率，其创建步骤如下（如*图 22*所示）：

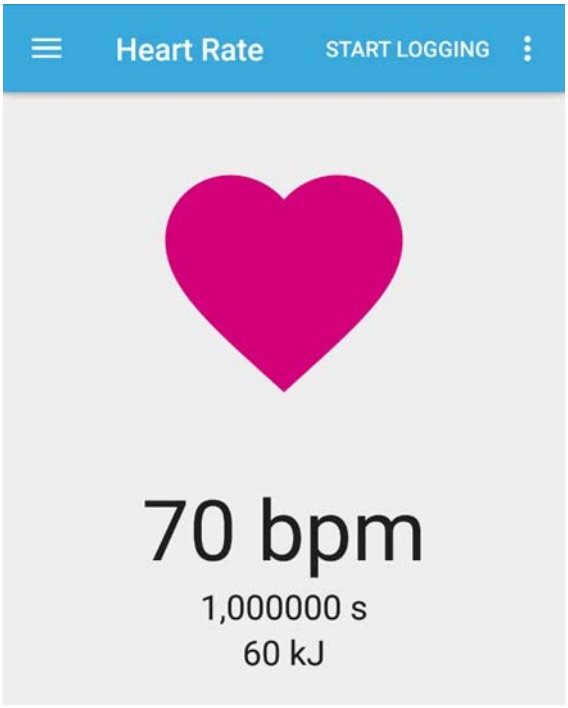
- STM32WB用户应用初始化
- 心率服务实现 - 中间件
- 心率传感器外设 - 用户
- 心率传感器测量值更新 - 用户。

图22. 使用BLE RF测试仪和P-NUCLEO板时的简单设置


### 7.2.1 如何使用STM32WB心率传感器应用

- 打开BLE_HeartRate项目并按照readme.txt中的说明操作
- 将ST BLE传感器移动应用连接到您的心率应用

图23. 智能手机 - ST BLE传感器和心率应用




### 7.2.2 STM32WB心率传感器应用 - 中间件应用

在Middlewares\STM32_WPAN\ble\core\Src\中，用于插入BLE服务的子文件夹为blesvc。

**警告： 不要修改该文件夹中的文件。**

- svc_ctl.c：初始化BLE协议栈并管理应用服务（GATT事件）
- Hrs.c：用于创建：
  - 应用的服务及其特征，
  - 更新服务特征，
  - 接收通知或写指令，以及
  - 在BLE协议栈与应用部分之间建立链接。

对于应用，用于创建特定代码的子文件夹为STM32_WPAN\app

- app_entry.c：初始化BLE传输层和BSP（例如LED和按钮）
- app_ble.c：初始化GAP并管理连接（例如广播和扫描）
- hrs_app.c：初始化GATT并管理应用

## 心率服务功能：Middlewares\STM32_WPAN\ble\core\Src\blesvc\hrs.c

表14. 心率服务功能

| 功能 | 说明 |
|---|---|
| Service Init - HRS_Init() | – 将心率事件句柄注册到服务控制器<br>– 初始化服务UUID |
| aci_gatt_add_serv | – 添加心率服务作为主要服务<br>– 初始化心率测量特征 |
| aci_gatt_add_char | – 添加心率特征<br>– 初始化身体传感器位置特征 |
| aci_gatt_add_char | – 添加身体传感器位置特征<br>– 更新心率测量特征 |
| aci_gatt_update_char_value | – 用指定范围以内的值更新请求的特征<br>– 更新身体传感器位置特征值 |
| aci_gatt_update_char_value | – 用指定范围以内的值更新请求的特征 |
| HeartRate_Event_Handler(void *Event) | – 管理HCI供应商类型事件 |
| EVT_BLUE_GATT_WRITE_PERMIT_REQ | – 服务器接收到写指令<br>– 心率控制点特征值<br>– 重置能量消耗命令，然后：<br>发送正常状态的aci_gatt_write_response()。<br>通知HRS应用重置消耗的能量<br>或发送错误状态的aci_gatt_write_response()。 |
| EVT_BLUE_GATT_ATTRIBUTE_MODIFIED | – 心率测量特征描述值<br>– 启用或禁用通知<br>– 将测量通知通知HRS应用 |

服务实现的目的是：

- 通过BLE协议栈GATT数据库注册心率服务和选定特征

```
/**
 * @brief  Service Heart Rate initialization
 * @参数  无
 * @返回值：无
 */
void HRS_Init(void)
{
REGISTER HEART RATE EVENT HANDLER
?   SVCCTL_RegisterSvcHandler(HearRate_Event_Handler);

REGISTER HEART RATE SERVICE GATT DATABASE TO BLE STACK
添加心率服务
添加心率特征
```

测量值（必要）
身体传感器位置（可选）
心率控制点（可选）
添加无线重启请求特征（可选）
}

- 管理HR服务专用的GATT事件

```
/**
 * @brief Heart Rate Service Event handler
 * @param Event: Address of the buffer holding the Event
 * @retval Ack: Return whether the GATT Event has been managed or not
 */
static SVCCTL_EvtAckStatus_t HearRate_Event_Handler(void *Event)
{
MANAGE GATT EVENT FROM BLE STACK
?  EVT_BLUE_GATT_WRITE_PERMIT_REQ
?  EVT_BLUE_GATT_ATTRIBUTE_MODIFIED

通知用户应用 – HRS_Notification
?  HRS_RESET_ENERGY_EXPENDED_EVT
?  HRS_NOTIFICATION_ENABLED
?  HRS_NOTIFICATION_DISABLED
?  HRS_STM_BOOT_REQUEST_EVT
}
```

- 允许应用将特征更新到BLE协议栈GATT数据库

```
/**
 * @brief  特征更新
 * @param  UUID：特征的UUID
 * @retval BodySensorLocationValue：要写入的新值
 */
tBleStatus HRS_UpdateChar(uint16_t UUID, uint8_t *pPayload)
{
更新身体传感器位置

更新心率测量值
}
```

### 服务控制器功能：Middlewares\STM32_WPAN\ble\core\Src\blesvc\svc_ctl.c

SVCCTL_Init()具有不同功能：

- 调用所有已开发服务的初始化函数
  - HR服务器 - HRS_Init()
- 注册服务事件处理函数
  - SVCCTL_RegisterSvcHandler()
  - 从svc_ctl.c接收GATT事件并将其重定向至应用的函数（hrs_app.c）
- 注册客户端事件处理函数（不适用于HR传感器项目）
  - SVCCTL_RegisterCltHandler()

### 心率传感器应用初始化：Applications\BLE\BLE_HeartRate\STM32_WPAN\App\app_ble.c

心率传感器外设初始化 - APP_BLE_Init()

- 初始化CPU2上的BLE协议栈
  - SHCI_C2_BLE_Init()
- 初始化HCI、GATT和GAP层
  - Ble_Hci_Gap_Gatt_Init()
- 初始化BLE服务
  - SVCCTL_Init()
- 调用心率服务器和设备信息应用初始化
  - HRSAPP_Init()
  - DISAPP_Init()
- 配置并开始广播：ADV参数、本地名称、UUID ...
  - aci_gap_set_discoverable() - 将设备设置为一般可发现模式。
  - aci_gap_update_adv_data() - 在广播数据包添加信息
- 管理GAP事件 - SVCCTL_App_Notification()
  - EVT_LE_CONN_COMPLETE

提供连接间隔 信息、从设备延迟和监控超时

- 提供新的连接信息
  - EVT_LE_CONN_UPDATE_COMPLETE
- 将链路断开及其原因通知应用
  - EVT_DISCONN_COMPLETE
- 通知应用链路是否加密
  - EVT_ENCRYPT_CHANGE

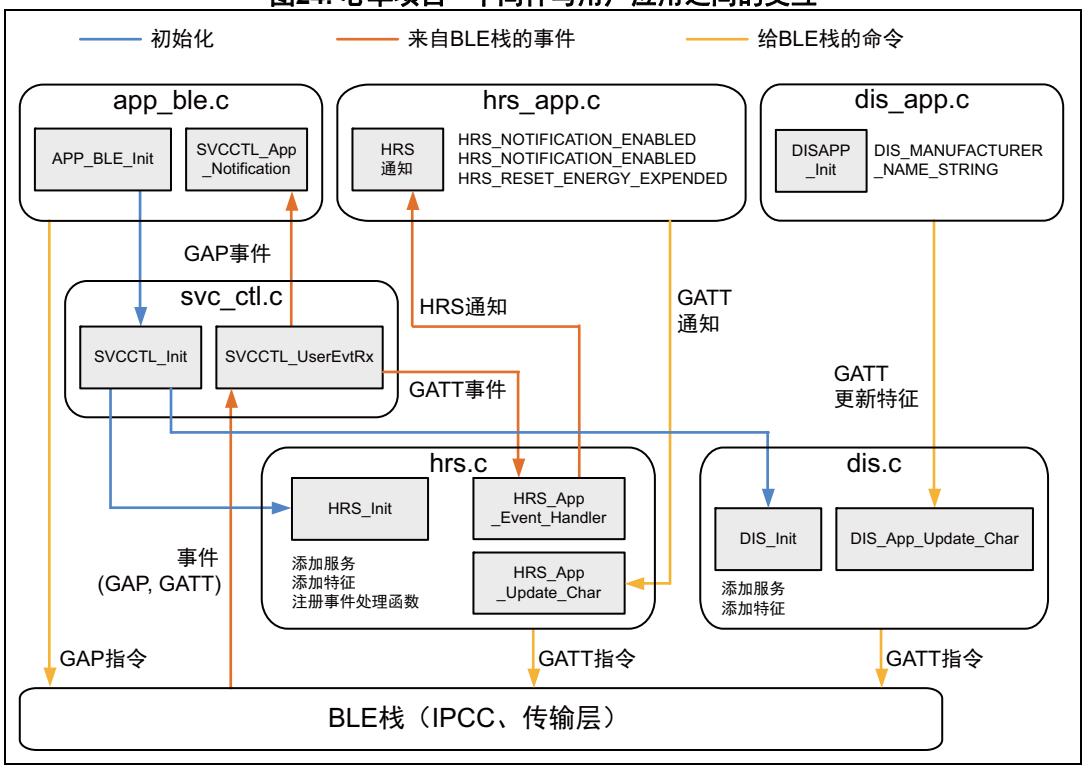
### 心率传感器应用控制：Applications\BLE\BLE_HeartRate\STM32_WPAN\App\hrs_app.c

hrs_app.c文件初始化传感器应用，创建定时器

表15. 心率传感器应用控制

| 功能 | 说明 |
|---|---|
| HRSAPP_Init() | 接收来自BLE协议栈的内部事件并做出响应（GATT层面）。 |
| HRS_Notification() | 调用服务函数以更新特征（通知/写入）。 |
| HRSAPP_Measurement() | - |

图24. 心率项目 - 中间件与用户应用之间的交互




## 7.3 意法半导体专有广播

当设备为外设时，它广播蓝牙地址和广播有效负载（长度为0至31字节）等信息。

广播信息用广播数据元素表示，并经蓝牙SIG标准化：

- 第一个字节：元素长度（不包括长度字节本身）
- 第二个字节：AD类型 - 指定元素中包含什么数据
- AD数据：一个或更多字节，其含义由AD类型定义。

AD类型“0xFF”用于提供制造商特定的数据。

意法半导体专有的基于GATT的应用程序（例如P2P和FUOTA应用程序）的实现建议使用制造商特定的AD类型数据。它是远程设备（Scanner）过滤外围设备并访问请求应用程序的一种方式。

表16. 蓝牙5核心规范 第3卷C部分规定的AD结构

| 字段名称 | 类型 | 长度 | 记录大小 |
|---|---|---|---|
| TX_POWER_LEVEL | 0x0A | 2 | 3 |
| COMPLETE_NAME | 0x09 | 8 | 9 |
| MANUF_SPECIFIC | 0xFF | 13 | 14 |
| 标志 | 0x01 | 2 | 3 |

表17. STM32WB制造商特定的数据

| Octect | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 名称 | 长度 | 类型 | Ver | DevID | A组特性 | | B组特性 | | 公共设备地址（48位），可选 | | | | | |
| 值 | 0x0D | 0xFF | 0x01 | 0xXX | RFU | | 0xXXXX | | 0xXXXXXXXXXX | | | | | |

## B组特性

- 位掩码 Thread：用于广播Thread切换特征的存在。
- 位掩码 OTA重启请求：用于广播BLE重启特征的存在。

表18. B组特性 - 位掩码

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RFU | Thread 支持 | OTA重启请求 | RFU | | | | | | | | | | | | |

表19. 设备ID枚举

| ID | HW |
|---|---|
| 0x00 | 通用 |
| 0x83 | STM32WB P2P服务器1 |
| 0x84 | STM32WB P2P服务器2 |
| 0x85 | STM32WB P2P路由器 |
| 0x86 | STM32WB FUOTA |

在app_ble.c中的用户应用层面管理广播流程。

下面是BLE_p2p Server项目广播启动的示例。

```
/* Local name to be advertised */
static const char local_name[] = { AD_TYPE_COMPLETE_LOCAL_NAME, 'P', '2', 'P', 'S', 'R', 'V', '1' };

/* manufacturer data & legacy data to be advertised */
uint8_t manuf_data[14] = {
   sizeof(manuf_data)-1, AD_TYPE_MANUFACTURER_SPECIFIC_DATA,
```

```
    0x01/*SKD version */,
    CFG_DEV_ID_P2P_SERVER1 /* STM32WB - P2P Server 1*/,
    0x00 /* GROUP A Feature  */,
    0x00 /* GROUP A Feature */,
    0x00 /* GROUP B Feature */,
    0x00 /* GROUP B Feature */,
    0x00, /* BLE MAC start -MSB */
    0x00,
    0x00,
    0x00,
    0x00,
    0x00, /* BLE MAC stop */
};

/* Local device BD address*/
const uint8_t *bd_addr;
bd_addr = SVCCTL_GetBdAddress();

/* BLE MAC update for Advertising manufacturer data*/
manuf_data[ sizeof(manuf_data)-6] = bd_addr[5];
manuf_data[ sizeof(manuf_data)-5] = bd_addr[4];
manuf_data[ sizeof(manuf_data)-4] = bd_addr[3];
manuf_data[ sizeof(manuf_data)-3] = bd_addr[2];
manuf_data[ sizeof(manuf_data)-2] = bd_addr[1];
manuf_data[ sizeof(manuf_data)-1] = bd_addr[0];

/* 使GAP外设进入一般可发现模式：
Advertising_Type: ADV_IND(undirected scannable and connectable);
Advertising_Interval_Min;
Advertising_Interval_Max;
Own_Address_Type: PUBLIC_ADDR (public address: 0x00);
Adv_Filter_Policy: NO_WHITE_LIST_USE (no whit list is used);
Local_Name_Length
Local_Name:
Service_Uuid_Length: 0 (不广播服务);
Service_Uuid_List: NULL;
Slave_Conn_Interval_Min: 0 (Slave connection internal minimum value);
Slave_Conn_Interval_Max: 0 (从设备连接间隔最大值).
*/
result = aci_gap_set_discoverable(ADV_IND,
CFG_FAST_CONN_ADV_INTERVAL_MIN,
CFG_FAST_CONN_ADV_INTERVAL_MAX,
PUBLIC_ADDR,
NO_WHITE_LIST_USE, /* use white list */
sizeof(local_name), (uint8_t*) local_name,
```

```
0,
NULL,
0, 0);

 /* Update Advertising data with manufacturer specific information*/
result = aci_gap_update_adv_data(sizeof(manuf_data), (uint8_t*) manuf_data);
```

总是将结果与BLE_STATUS_SUCCESS (0x00)进行比较。

## 7.4 专有P2P应用

可使用三个组成部分来演示不同数据通信类型：

1. P2P服务器工程
2. P2P客户端工程
3. 智能手机应用

不同组成部分的组合得到如*图 25*和*图 26*所示的演示结果。

**图25. P2P服务器至客户端的演示**

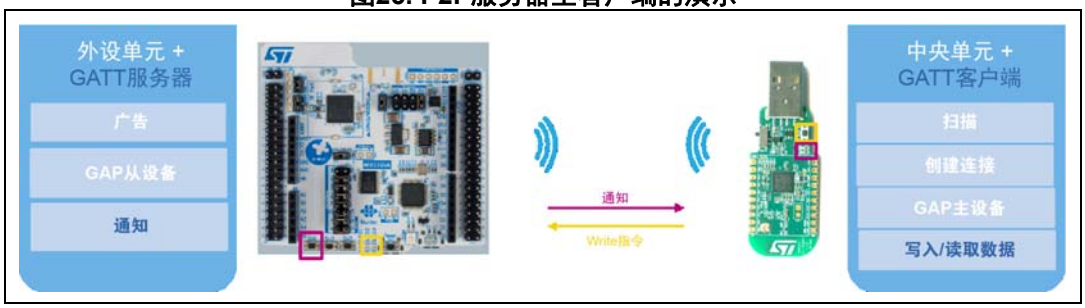
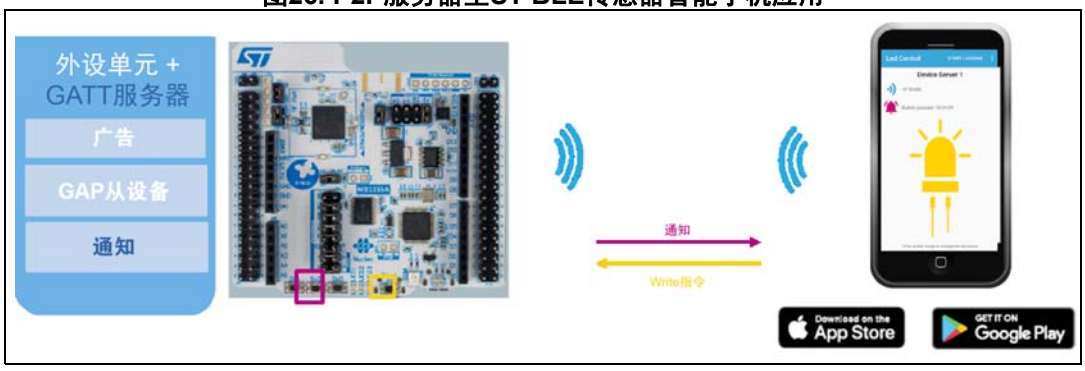



**图26. P2P服务器至ST BLE传感器智能手机应用**




### 7.4.1 P2P服务器规范

必须使用P2P服务器应用演示点到点通信。它作为外设，具有下列GATT服务和特征。

表20. P2P服务和特征UUID

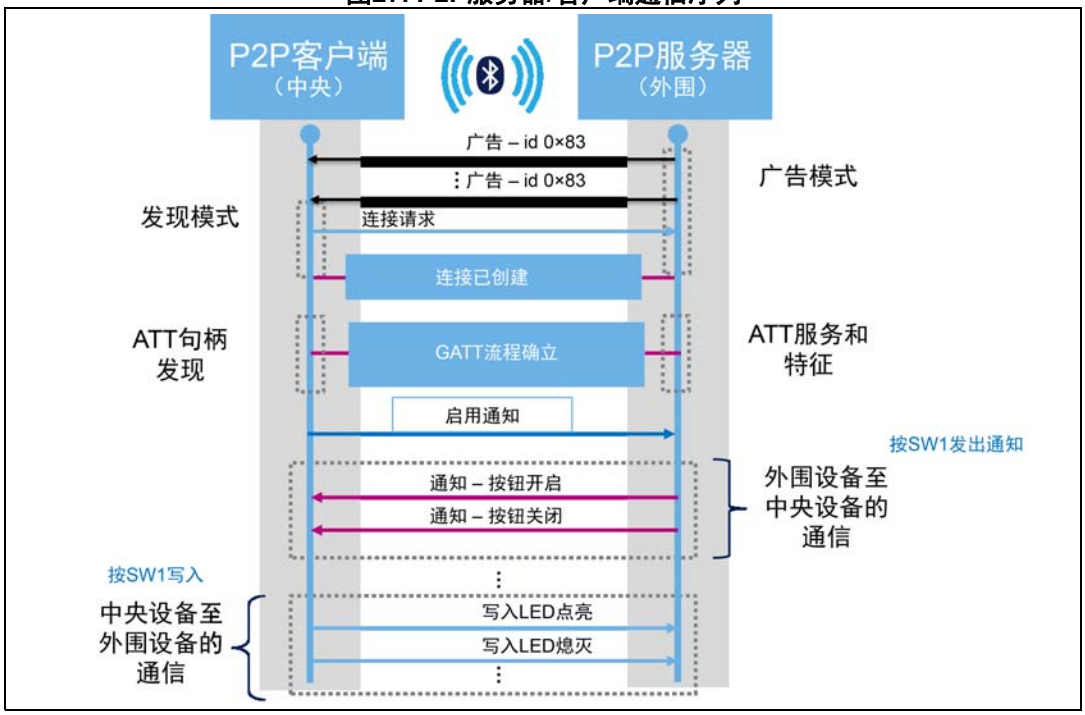
| 组 | 服务 | 特征 | 大小 | 模式 | UUID |
|---|---|---|---|---|---|
| LED按钮控制 | P2P服务 | - | - | - | 0000FE40-cc7a-482a-984a-7fed5b3e58f |
| | - | 写 | 2 | 读/写 | 0000FE41-8e22-4541-9d4c-21edae82ed19 |
| | - | 通知 | 2 | 通知 | 0000FE42-8e22-4541-94dc-21edae82ed19 |

表21. P2P规范

| | | | |
|---|---|---|---|
| **Write** | Octets LSB | 0 | 1 |
| | 名称 | 设备选择 | LED控制 |
| | 值 | – 0x01：P2P服务器1<br>– 0x02：P2P服务器2<br>– 0x0x：P2P服务器x<br>– 0x00：全部 | – 0x00 LED关闭<br>– 0x01 LED开启<br>– 0x02 Thread切换 |
| **通知** | Octets LSB | 0 | 1 |
| | 名称 | 设备选择 | 按钮 |
| | 值 | – 0x01：P2P服务器1<br>– 0x02：P2P服务器2<br>– 0x0x：P2P服务器x | – 0x00 关闭<br>– 0X01 开启 |

使用前，GAP中央设备和GATT客户端设备必须发现并连接到P2P服务器应用。*图 27*描述了数据交换流程。

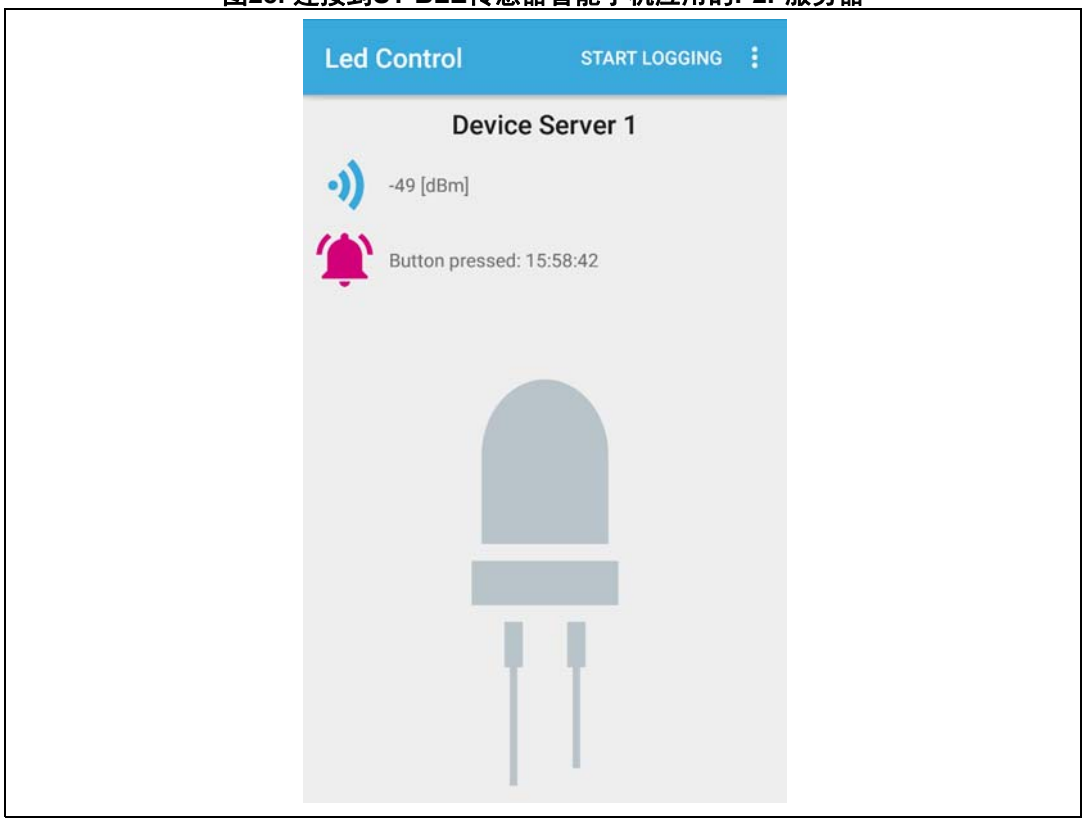
图27. P2P服务器/客户端通信序列



### 7.4.2 P2P服务器应用的使用方法

1. 将ble_P2P_Server工程烧录到P-NUCLEO-WB55板上
2. 烧录完成后，将ST BLE传感器移动应用连接到板上，并使用SW1按钮通知智能手机。

**图28. 连接到ST BLE传感器智能手机应用的P2P服务器**




### 7.4.3 P2P服务器应用 - 中间件应用

使用p2p_stm.c文件创建P2P服务和特征

p2p_stm.c：在应用中创建服务和特征，以便更新特征，接收通知或写指令，以及在BLE无线协议栈与应用部分之间建立链接。

在应用中，用于创建特定代码的子文件夹为“User”

- app_entry.c：初始化BLE传输层和BSP（例如LED和按钮）
- app_ble.c：初始化GAP并管理连接（例如广播和扫描）
- p2p_server_app.c：初始化GATT并管理应用。

### P2P服务功能：Middlewares\STM32_WPAN\ble\core\Src\blesvc\p2p_stm.c

表22. P2P服务功能

| 功能 | 说明 |
|---|---|
| Service Init - P2PS_STM_Init () | – 将PeerToPeer_Event_Handler注册到服务控制器<br>– 初始化服务UUID<br>aci_gatt_add_serv - 添加P2P服务作为主要服务<br>– 初始化P2P写入特征<br>aci_gatt_add_char - 初始化写入特征<br>– 初始化P2P通知特征<br>aci_gatt_add_char - 添加通知特征<br>– 更新通知特征 - P2PS_STM_App_Update_Char()<br>aci_gatt_update_char_value - 用符合规范的值更新通知特征 |
| PeerToPeer_Event_Handler (void *Event) - 管理HCI供应商类型事件： | EVT_BLUE_GATT_ATTRIBUTE_MODIFIED<br>– 接收通知特征描述符值的配置<br>– 启用或禁用通知<br>– 将P2PS_STM__NOTIFY_ENABLED_EVT或P2PS_STM_NOTIFY_DISABLED_EVT通知应用<br>– 接收关于写入特征的数据并通知P2P应用<br>P2PS_STM_WRITE_EVT<br>– 接收关于重启请求特征的数据并通知P2P应用（将用于FUOTA流程）<br>P2PS_STM_BOOT_REQUEST_EVT |

### P2P服务器应用控制：Applications\BLE\BLE_p2pServer\STM32_WPAN\App\p2p_server_app.c

p2p_server_app.c文件

初始化P2P服务器应用，创建定时器
P2PS_APP_Init()

接收来自BLE协议栈的内部事件并做出响应（GATT层面）。
P2PS_STM_App_Notification ()

```
void P2PS_STM_App_Notification(P2PS_STM_App_Notification_evt_t *pNotification)
{
切换（pNotification->P2P_Evt_Opcode）
{
 case P2PS_STM__NOTIFY_ENABLED_EVT:
P2P_Server_App_Context.Notification_Status = 1;
  APP_DBG_MSG("-- P2P APPLICATION SERVER : NOTIFICATION ENABLED\n");
  APP_DBG_MSG(" \n\r");
 break;
```

```
 case P2PS_STM_NOTIFY_DISABLED_EVT:
P2P_Server_App_Context.Notification_Status = 0;
.APP_DBG_MSG("-- P2P APPLICATION SERVER : NOTIFICATION DISABLED\n");
  APP_DBG_MSG(" \n\r");
 break;

 case P2PS_STM_WRITE_EVT:
if(pNotification->DataTransfered.pPayload[0] == 0x00){
.if(pNotification->DataTransfered.pPayload[1] == 0x01)
  {
   BSP_LED_On(LED_BLUE);
   APP_DBG_MSG("-- P2P APPLICATION SERVER  : LED1 ON\n");
   APP_DBG_MSG(" \n\r");
   P2P_Server_App_Context.LedControl.Led1=0x01;
   }
  if(pNotification->DataTransfered.pPayload[1] == 0x00)
  {
   BSP_LED_Off(LED_BLUE);
   APP_DBG_MSG("-- P2P APPLICATION SERVER  : LED1 OFF\n");
   APP_DBG_MSG(" \n\r");
   P2P_Server_App_Context.LedControl.Led1=0x00;
   }
  }
```

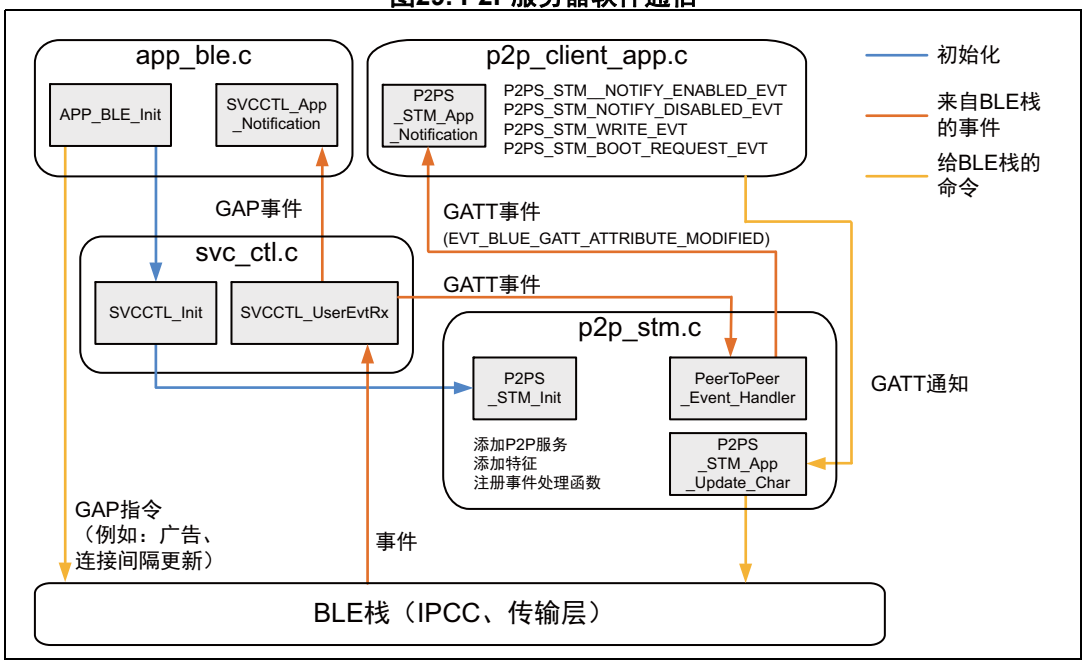
调用服务函数以更新特征（通知）。

```
P2PS_Send_Notification ()
void P2PS_Send_Notification(void)
{
 if(P2P_Server_App_Context.ButtonControl.ButtonStatus == 0x00){
  P2P_Server_App_Context.ButtonControl.ButtonStatus=0x01;
 } else {
  P2P_Server_App_Context.ButtonControl.ButtonStatus=0x00;
 }

 if(P2P_Server_App_Context.Notification_Status){
  APP_DBG_MSG("P2P APPLICATION SERVER  : INFORM CLIENT BUTTON 1 PUSHED \n ");
  APP_DBG_MSG(" \n\r");
  P2PS_STM_App_Update_Char(P2P_NOTIFY_CHAR_UUID, (uint8_t *)
&P2P_Server_App_Context.ButtonControl);
  } else {
  APP_DBG_MSG("P2P APPLICATION SERVER : CAN'T INFORM CLIENT -  NOTIFICATION
DISABLED\n ");
  }
return;
}
```

### 7.4.4 P2P客户端应用 - 中间件应用

图29. P2P服务器软件通信




这里没有为P2P客户端创建服务。只需注册GATT客户端处理函数SVCCTL_RegisterCltHandler()，就能接收应用层面的任何GATT事件通知。

在应用中，用于创建特定代码的子文件夹为“User”

- app_entry.c：初始化BLE传输层和BSP（例如LED和按钮）
- app_ble.c：初始化GAP并管理连接（扫描和连接）
- p2p_client_app.c：初始化GATT并管理GATT客户端应用。

#### P2P客户端 - 扫描和连接

app_ble.c文件

- 执行扫描以搜索任何P2P服务器广播ID：

```
static void Scan_Request( void )
{
 tBleStatus result;
 if (BleApplicationContext.Device_Connection_Status != APP_BLE_CONNECTED_CLIENT)
 {
  BSP_LED_On(LED_BLUE);
  result = aci_gap_start_general_discovery_proc(SCAN_P, SCAN_L, PUBLIC_ADDR, 1);
```

```
 if (result == BLE_STATUS_SUCCESS)
 {
  APP_DBG_MSG(" \r\n\r** START GENERAL DISCOVERY (SCAN) **  \r\n\r");
 } else {
 APP_DBG_MSG("-- BLE_App_Start_Limited_Disc_Req, Failed \r\n\r");
 BSP_LED_On(LED_RED);
 }
}
return;
}
```

- 接收要筛选的ADV事件报告以保存P2P服务器BD地址：J’rric

```
case AD_TYPE_MANUFACTURER_SPECIFIC_DATA: // Manufactureur Specific
 if (adlength >= 7 && le_advertising_event->Advertising_Report[0].Data[k + 2] == 0x01) {
  APP_DBG_MSG("--- ST MANUFACTURER ID --- \n");
  switch (le_advertising_event->Advertising_Report[0].Data[k + 3]) {
   case CFG_DEV_ID_P2P_SERVER1:
    APP_DBG_MSG("-- SERVER DETECTED -- VIA MAN ID\n");
    BleApplicationContext.DeviceServerFound = 0x01;
    SERVER_REMOTE_BDADDR[0] = le_advertising_event->Advertising_Report[0].Address[0];
    SERVER_REMOTE_BDADDR[1] = le_advertising_event->Advertising_Report[0].Address[1];
    SERVER_REMOTE_BDADDR[2] = le_advertising_event->Advertising_Report[0].Address[2];
    SERVER_REMOTE_BDADDR[3] = le_advertising_event->Advertising_Report[0].Address[3];
    SERVER_REMOTE_BDADDR[4] = le_advertising_event->Advertising_Report[0].Address[4];
    SERVER_REMOTE_BDADDR[5] = le_advertising_event->Advertising_Report[0].Address[5];
   break;
```

- 初始化与发现的任何P2P服务器的连接：

```
static void Connect_Request( void )
{
 tBleStatus result;
 APP_DBG_MSG("\r\n\r** CREATE CONNECTION TO SERVER **  \r\n\r");
 if (BleApplicationContext.Device_Connection_Status != APP_BLE_CONNECTED_CLIENT) {
  result = aci_gap_create_connection(
SCAN_P,
SCAN_L,
PUBLIC_ADDR, SERVER_REMOTE_BDADDR,
PUBLIC_ADDR,
CONN_P1,
```

```
CONN_P2,
0,
SUPERV_TIMEOUT,
CONN_L1,
CONN_L2);
```

- 在建立连接后启动“服务发现流程”：

```
case EVT_LE_CONN_COMPLETE:
/**
* 连接已建立
*/
connection_complete_event = (hci_le_connection_complete_event_rp0 *) meta_evt->data;
BleApplicationContext.BleApplicationContext_legacy.connectionHandle =
connection_complete_event->Connection_Handle;
BleApplicationContext.Device_Connection_Status = APP_BLE_CONNECTED_CLIENT;

APP_DBG_MSG("\r\n\r** CONNECTION EVENT WITH SERVER \n");
handleNotification.P2P_Evt_Opcode = PEER_CONN_HANDLE_EVT;
handleNotification.ConnectionHandle =
 BleApplicationContext.BleApplicationContext_legacy.connectionHandle;
P2PC_APP_Notification(&handleNotification);
result = aci_gatt_disc_all_primary_services(
BleApplicationContext.BleApplicationContext_legacy.connectionHandle);
if (result == BLE_STATUS_SUCCESS) {
 APP_DBG_MSG("\r\n\r** GATT SERVICES & CHARACTERISTICS DISCOVERY  \n");
 APP_DBG_MSG("* GATT :  Start Searching Primary Services \r\n\r");
}
```

在这一步，所有GATT事件都被传输到在p2p_client_app.c中管理的GATT客户端事件处理函数。

### P2P客户端 - 应用控制 - GATT客户端通信

p2p_client_app.c文件：

- 初始化P2P客户端应用并注册客户端事件处理函数
  - P2PC_APP_Init()
  - SVCCTL_RegisterCltHandler()
- 启动发现流程并管理远程P2P服务器特征
  - aci_gatt_disc_all_char_of_service()
  - aci_gatt_disc_all_char_desc()
  - aci_gatt_write_char_desc()

```
case APP_BLE_DISCOVER_SERVICES:
APP_DBG_MSG("P2P_DISCOVER_SERVICES\n");
break;
case APP_BLE_DISCOVER_CHARACS:
APP_DBG_MSG("* GATT : Discover P2P Characteristics\n");
```

```
aci_gatt_disc_all_char_of_service(aP2PClientContext[index].connHandle,
aP2PClientContext[index].P2PServiceHandle,
aP2PClientContext[index].P2PServiceEndHandle);
break;
case APP_BLE_DISCOVER_WRITE_DESC:
APP_DBG_MSG("* GATT : Discover Descriptor of TX - Write Characteritic\n");
aci_gatt_disc_all_char_desc(aP2PClientContext[index].connHandle,
aP2PClientContext[index].P2PWriteToServerCharHdle,
aP2PClientContext[index].P2PWriteToServerCharHdle+2);
break;
case APP_BLE_DISCOVER_NOTIFICATION_CHAR_DESC:
APP_DBG_MSG("* GATT : Discover Descriptor of Rx - Notification Characteritic\n");
aci_gatt_disc_all_char_desc(aP2PClientContext[index].connHandle,
aP2PClientContext[index].P2PNotificationCharHdle,
aP2PClientContext[index].P2PNotificationCharHdle+2);
break;
case APP_BLE_ENABLE_NOTIFICATION_DESC:
APP_DBG_MSG("* GATT : Enable Server Notification\n");
aci_gatt_write_char_desc(aP2PClientContext[index].connHandle,
aP2PClientContext[index].P2PNotificationDescHandle,
2,
(uint8_t *)&enable);
aP2PClientContext[index].state = APP_BLE_CONNECTED_CLIENT;
break;
case APP_BLE_DISABLE_NOTIFICATION_DESC :
APP_DBG_MSG("* GATT : Disable Server Notification\n");
aci_gatt_write_char_desc(aP2PClientContext[index].connHandle,
aP2PClientContext[index].P2PNotificationDescHandle,
2,
(uint8_t *)&enable);
aP2PClientContext[index].state = APP_BLE_CONNECTED_CLIENT;
break;
```

- 管理GATT事件以查找和注册远程设备特征句柄
  - SVCCTL_EvtAckStatus_t Event_Handler()

```
uuid = UNPACK_2_BYTE_PARAMETER(&pr->Attribute_Data_List[idx]);
if(uuid == P2P_SERVICE_UUID){
APP_DBG_MSG("-- GATT : P2P_SERVICE_UUID FOUND - connection handle 0x%x \n",
aP2PClientContext[index].connHandle);
aP2PClientContext[index].P2PServiceHandle = UNPACK_2_BYTE_PARAMETER(&pr-
>Attribute_Data_List[idx-16]);
aP2PClientContext[index].P2PServiceEndHandle = UNPACK_2_BYTE_PARAMETER (&pr-
>Attribute_Data_List[idx-14]);
aP2PClientContext[index].state = APP_BLE_DISCOVER_CHARACS ;
}
```

```
uuid = UNPACK_2_BYTE_PARAMETER(&pr->Handle_Value_Pair_Data[idx]);
/* store the characteristic handle not the attribute handle */
handle = UNPACK_2_BYTE_PARAMETER(&pr->Handle_Value_Pair_Data[idx-14]);
if(uuid == P2P_WRITE_CHAR_UUID){
APP_DBG_MSG("-- GATT : WRITE_UUID FOUND - connection handle 0x%x\n",
aP2PClientContext[index].connHandle);
aP2PClientContext[index].state = APP_BLE_DISCOVER_WRITE_DESC;
aP2PClientContext[index].P2PWriteToServerCharHdle = handle;
}
else if(uuid == P2P_NOTIFY_CHAR_UUID){
APP_DBG_MSG("-- GATT : NOTIFICATION_CHAR_UUID FOUND  - connection handle 0x%x\n",
aP2PClientContext[index].connHandle);
aP2PClientContext[index].state = APP_BLE_DISCOVER_NOTIFICATION_CHAR_DESC;
aP2PClientContext[index].P2PNotificationCharHdle = handle;
}
```

在发现P2P服务器服务和特征句柄后，应用能够：

- 使用“写入”特征控制远程设备

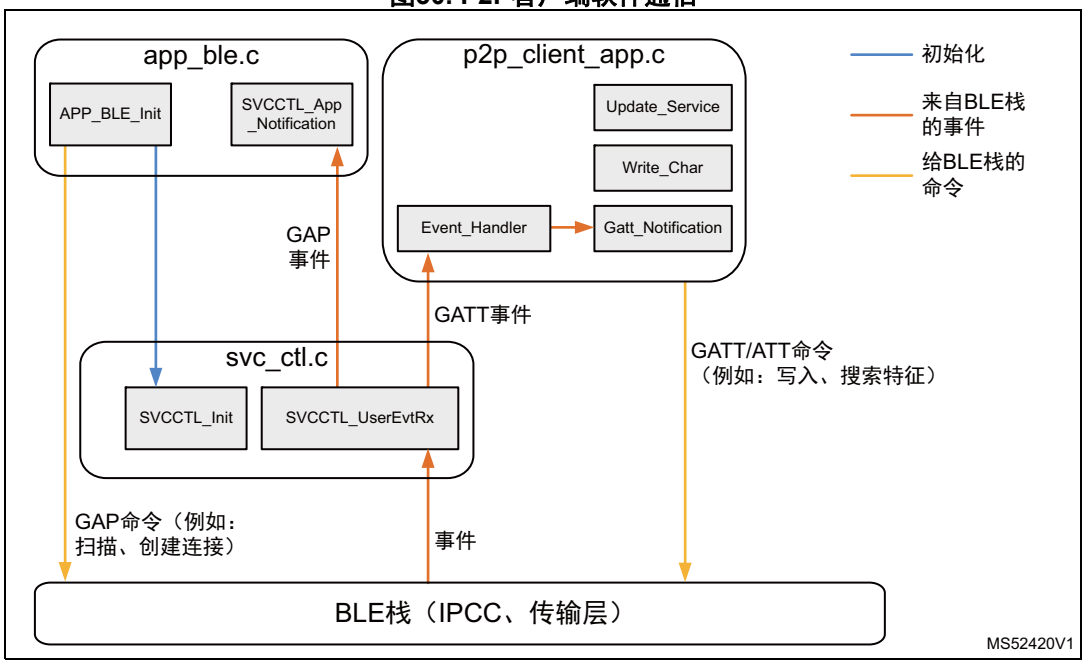
```
tBleStatus Write_Char(uint16_t UUID, uint8_t Service_Instance, uint8_t *pPayload){
tBleStatus ret = BLE_STATUS_INVALID_PARAMS;
uint8_t index;
索引 = 0；
while((index < BLE_CFG_CLT_MAX_NBR_CB) && (aP2PClientContext[index].state !=
APP_BLE_IDLE)){
switch(UUID){
case P2P_WRITE_CHAR_UUID:
ret =aci_gatt_write_without_resp(aP2PClientContext[index].connHandle,
aP2PClientContext[index].P2PWriteToServerCharHdle,
2, /* charValueLen */
(uint8_t *)  pPayload);
break;
```

- 通过“通知”特征接收通知

```
void Gatt_Notification(P2P_Client_App_Notification_evt_t *pNotification){
switch(pNotification->P2P_Client_Evt_Opcode){
case P2P_NOTIFICATION_INFO_RECEIVED_EVT: {
P2P_Client_App_Context.LedControl.Device_Led_Selection=pNotification-
>DataTransfered.pPayload[0];
switch(P2P_Client_App_Context.LedControl.Device_Led_Selection) {
case 0x01 : {
P2P_Client_App_Context.LedControl.Led1=pNotification->DataTransfered.pPayload[1];
if(P2P_Client_App_Context.LedControl.Led1==0x00){
BSP_LED_Off(LED_BLUE);
APP_DBG_MSG(" -- P2P CLIENT : NOTIFICATION RECEIVED - LED OFF \n\r");
} else {
BSP_LED_On(LED_BLUE);
APP_DBG_MSG(" -- P2P CLIENT : NOTIFICATION RECEIVED - LED ON\n\r");
```

```
}
break;
}
```

**图30. P2P客户端软件通信**




## 7.5 FUOTA应用程序

FUOTA是一个独立的应用，能够安装BLE服务以下载新的CPU2无线协议栈、CPU1应用或配置二进制文件：

- 它要求永不删除应用的前六个闪存扇区（FUOTA应用的写入位置）。
- FUOTA应用能够：
  - 更新整个CPU1应用
  - 下载要通过FUS应用的CPU2无线固件
  - 下载CPU1用户闪存中任何地址的用户数据。

### 7.5.1 CPU1用户闪存映射

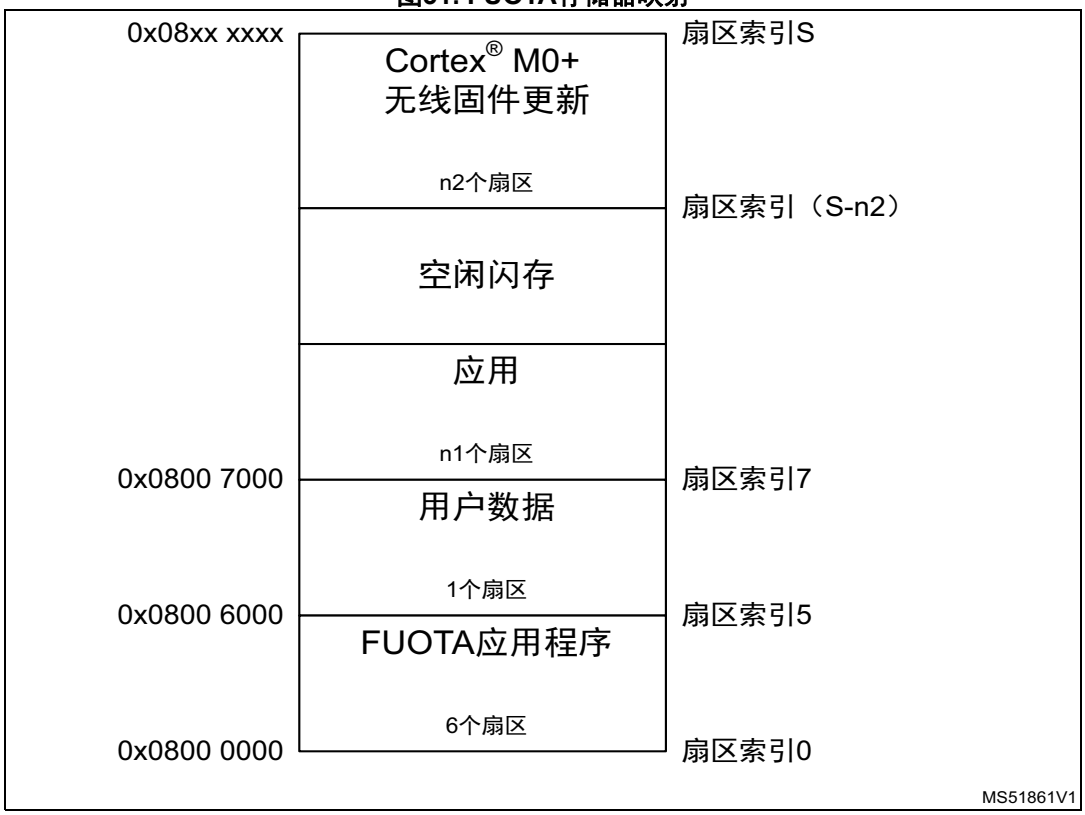
FUOTA BLE应用不能自我更新但能够：

- 跳转到现有应用（扇区索引7）
- 运行并安装意法半导体专有FUOTA GATT服务和特征，以便上传远程设备指定区域中的任何数据。

用户数据区可用于更新部分应用配置。

应用区包含应用的独立二进制文件。它可以通过FUOTA应用进行全面更新。

**图31. FUOTA存储器映射**




### 7.5.2 BLE FUOTA应用启动

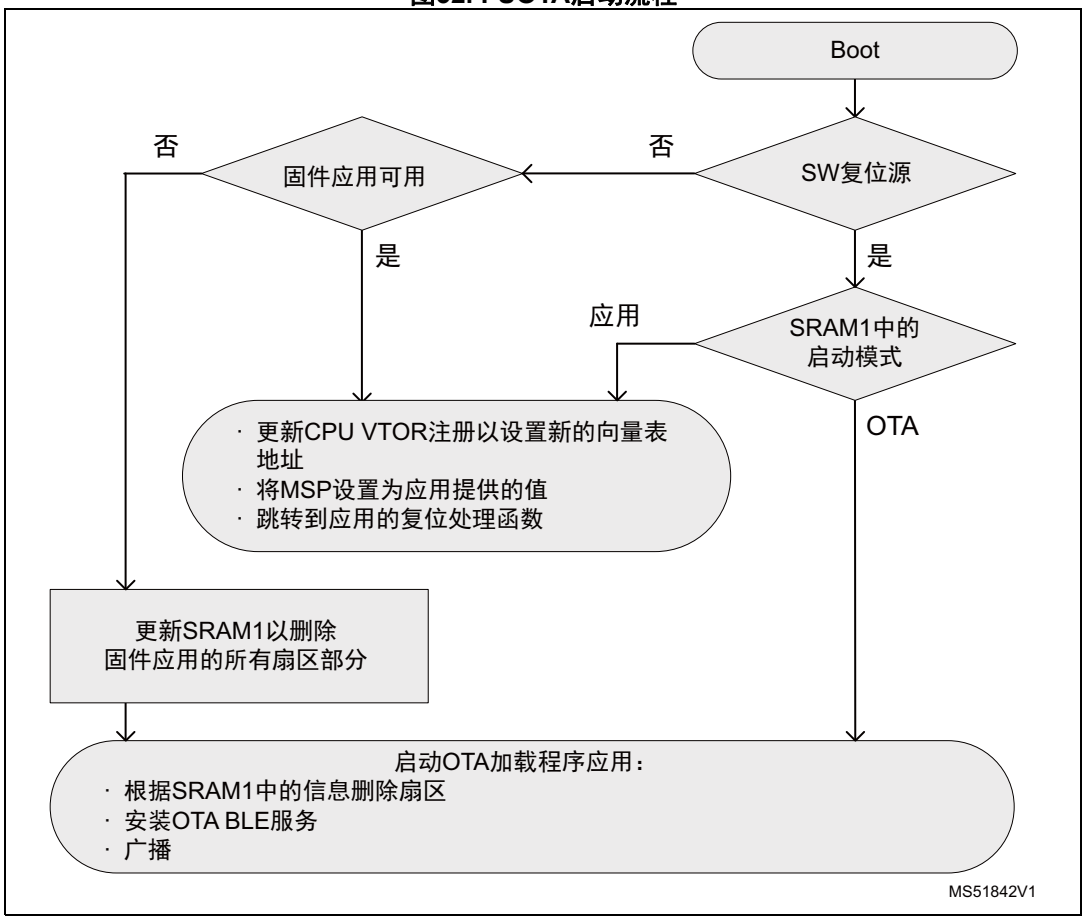
在编译并加载项目BLE_Ota后，应用可以：

- 在应用扇区（扇区索引7）有二进制代码时跳转到可用应用
  - 没有更多与BLE_Ota应用相关的活动
- 或者启动意法半导体专有FUOTA GATT服务和特征广播：
  - 本地名称为“STM_OTA”的广播数据（AD）元素
  - 设备ID为 “STM32WB固件更新OTA应用”的制造商AD元素

第二种情况允许远程设备上传新的二进制文件（CPU2无线协议栈、CPU1应用或用户数据固件更新）。

*注：* *如果只使用意法半导体专有FUOTAGATT服务和特征，则必须擦除应用扇区（自扇区7起）。*

图32. FUOTA启动流程




### 7.5.3 BLE FUOTA服务和特征规范

BLE FUOTA应用（BLE_Ota项目）导出为GATT服务，具有下列特征：

- 基址，提供新的二进制文件的存储位置
- 文件上传 重启确认，在上传新的二进制文件后确认应用重启
- OTA原始数据，用于传输数据（分割成数据包的二进制文件）

表23. FUOTA服务和特征UUID

| 产品部 | 服务 | 特征 | 大小 | 模式 | UUID |
|---|---|---|---|---|---|
| LED按钮控制 | OTA FW 更新 | - | - | - | 0000FE20-cc7a-482a-984a-7f2ed5b3e58f |
| | - | 基址 | 4 字节 | Write | 0000FE22-8e22-4541-9d4c-21edae82ed19 |
| | - | 文件上传重启确认 | 1 字节 | 指示 | 0000FE23-8e22-4541-9d4c-21edae82ed19 |
| | - | OTA 原始数据 | 20 字节 | 写入，无响应 | 0000FE24-8e22-4541-9d4c-21edae82ed19 |

表24. 基址特征规范

| LSB位 | [0:7] | [8:31] |
|---|---|---|
| 名称 | 动作 | 地址 |
| 值 | – 0x00：停止所有上传<br>– 0x01：开始M0+文件上传<br>– 0x02：开始M0+文件上传<br>– 0x07：文件上传完成<br>– 0x08：取消上传 | 0x007000 |

表25. 文件上传确认重启请求特征规范

| Octects LSB | 0 |
|---|---|
| 名称 | 指示 |
| 值 | 0x01重启 |

表26. 原始数据特征规范

| Octects LSB | 0 | 1 | ... | 19 |
|---|---|---|---|---|
| 名称 | 原始数据 | | | |
| 值 | 文件数据 | | | |

### 7.5.4 上传新的CPU1应用二进制文件的流程说明示例

上传新的二进制文件的流程有两种：

- 只加载意法半导体专有FUOTA GATT服务和特征应用。（扇区7上无应用二进制文件）
- 应用已在运行，支持重启请求特征。

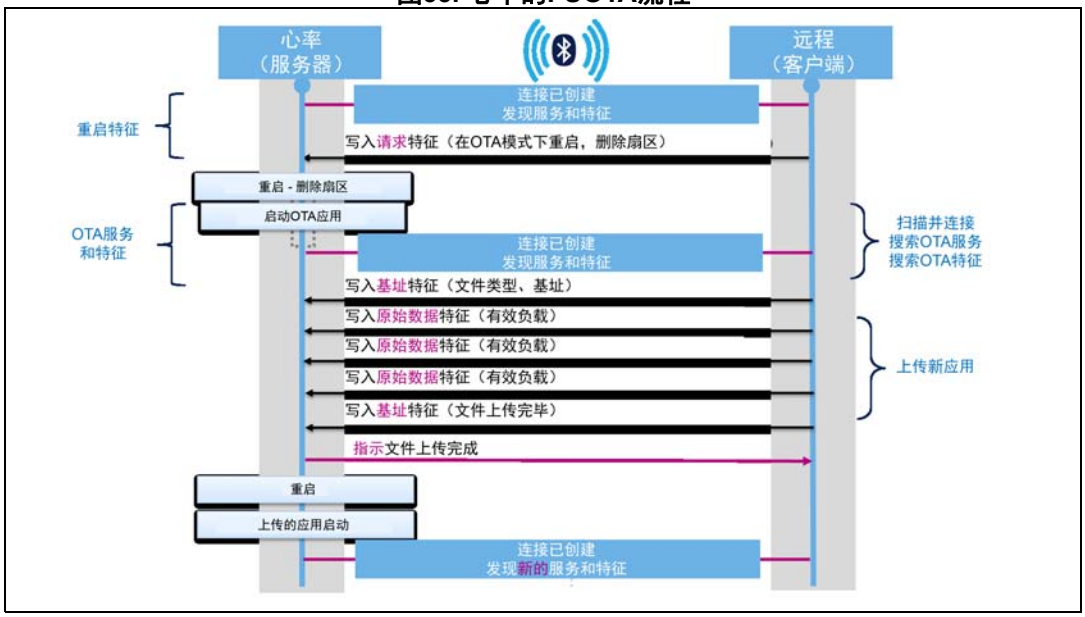
表27. 重启请求特性规范

| Octects LSB | 0 | 1 | 2 |
|---|---|---|---|
| 名称 | 自举模式 | 扇区索引 | 要擦除的扇区数 |
| 值 | – 0x00应用程序<br>– 0x01 FUOTA应用程序 | 07 → 0x0800 7000 | 0x00 ... 0xFF |

从包含重启请求特征的应用开始，CPU1应用的更新流程如下：

1. BLE应用包含重启特征。
2. 在建立GAP连接后，远程GATT客户端设备搜索服务和特征（发现重启请求特征）。
3. 然后，为了切换到FUOTA应用，远程设备将启动模式选项和要擦除的扇区的信息写入重启请求特征。
4. 在此阶段，断开BLE 连接，以便重启意法半导体专有FUOTA GATT服务和特征应用。
5. 通过重启特征提供的信息擦除应用扇区，意法半导体专有FUOTA GATT服务和特征应用开始广播。
6. 远程设备必须建立新的连接以发现 FUOTA服务和特征。
7. 基址特征用于发起新二进制文件的上传。
8. 所有数据均通过原始数据特征进行传输，并在收到后直接写入闪存中。
9. 通过基址特征确认文件传输结束。
10. 通过文件上传确认特征确认文件已接收。
11. 在此阶段， FUOTA应用检查新二进制文件的完整性，并重启以启动已上传的新应用。
12. 如果无法确保应用完整性，将擦除应用扇区，重启FUOTA应用。

**图33. 心率的FUOTA流程**



### 7.5.5 智能手机的应用示例

在工程中实现重启请求特征

- BLE_HeartRate_ota
- BLE_P2pServer_ota

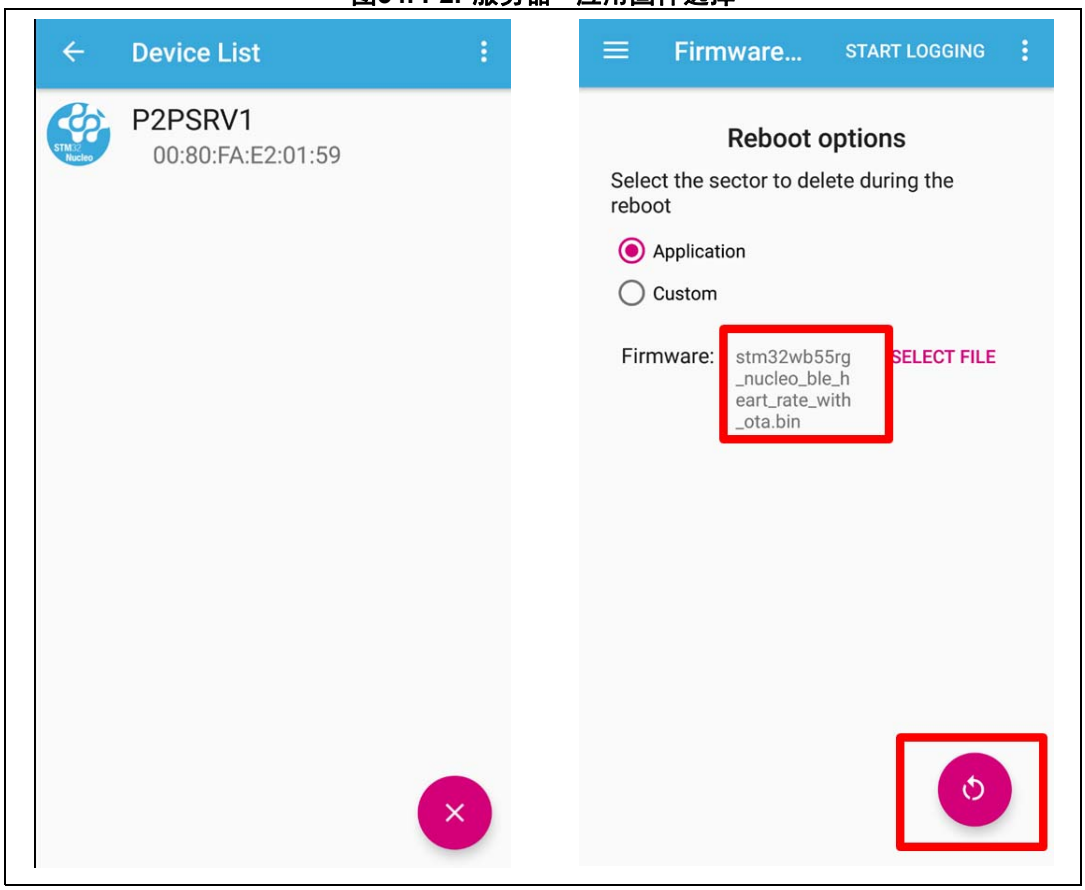
两个工程的广播元素中均包含OTA重启请求位掩码。它是远程设备（scanner）快速获取重启请求特征信息存在的一种方式。

ST BLE Sensor移动应用支持此重启请求特征的检测。

例如，从P2P服务器应用更新到心率应用。

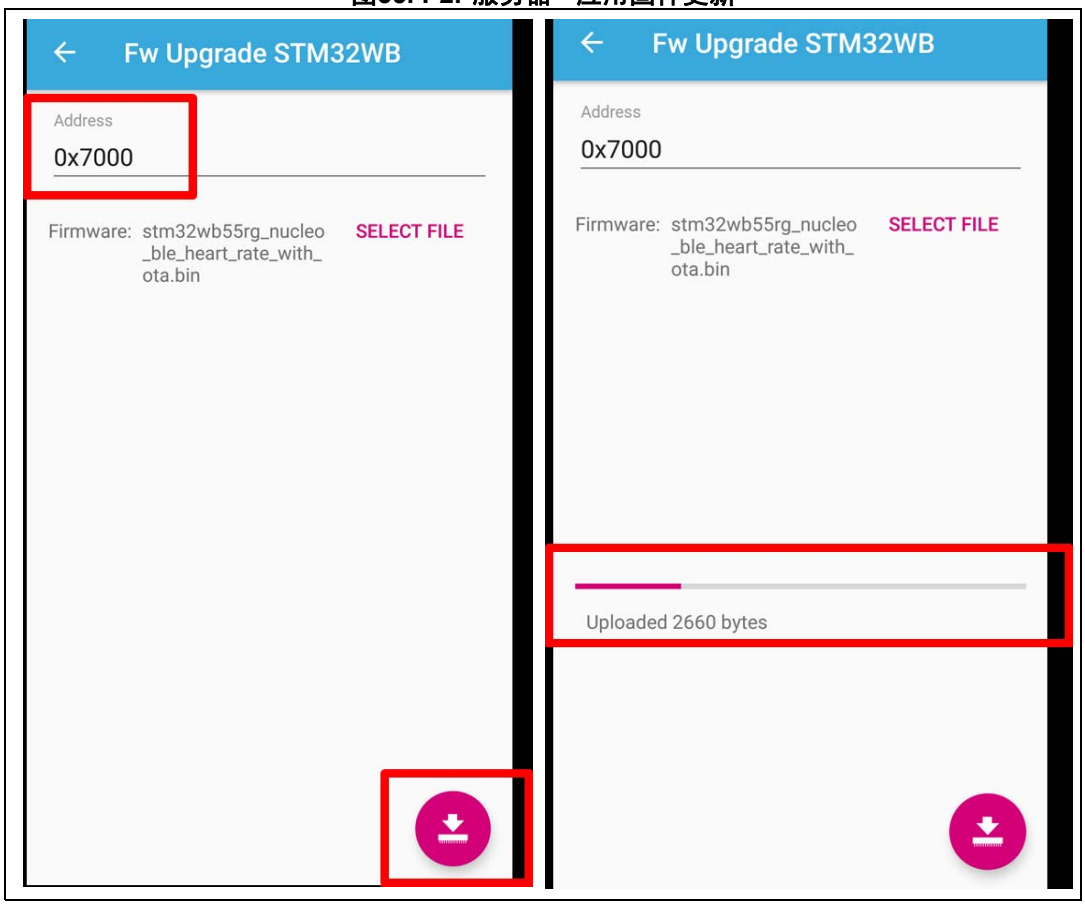
- 编译BLE_Ota项目并加载到地址0x0800 0000
- 编译BLE_p2pServer_ota项目并加载到地址0x0800 7000
- 重启设备
  - 在此阶段，P2P服务器广播其存在。
- 通过ST BLE Sensor移动应用发现并连接到P2P服务器
- 移动到重启面板
- 选择二进制文件“BLE_HeartRate_ota”（在演示前复制到智能手机内存）
- 点击上传
  - 在此阶段，重启请求特征用于提供要擦除的扇区和下一个重启阶段（FUOTA应用）的相关信息。

图34. P2P服务器 - 应用固件选择




在重启后，选择上传应用二进制文件的 地址。默认地址为0x7000（扇区7 - 应用）。在此阶段，如有必要，仍然可以修改要上传的二进制文件。

图35. P2P服务器 - 应用固件更新



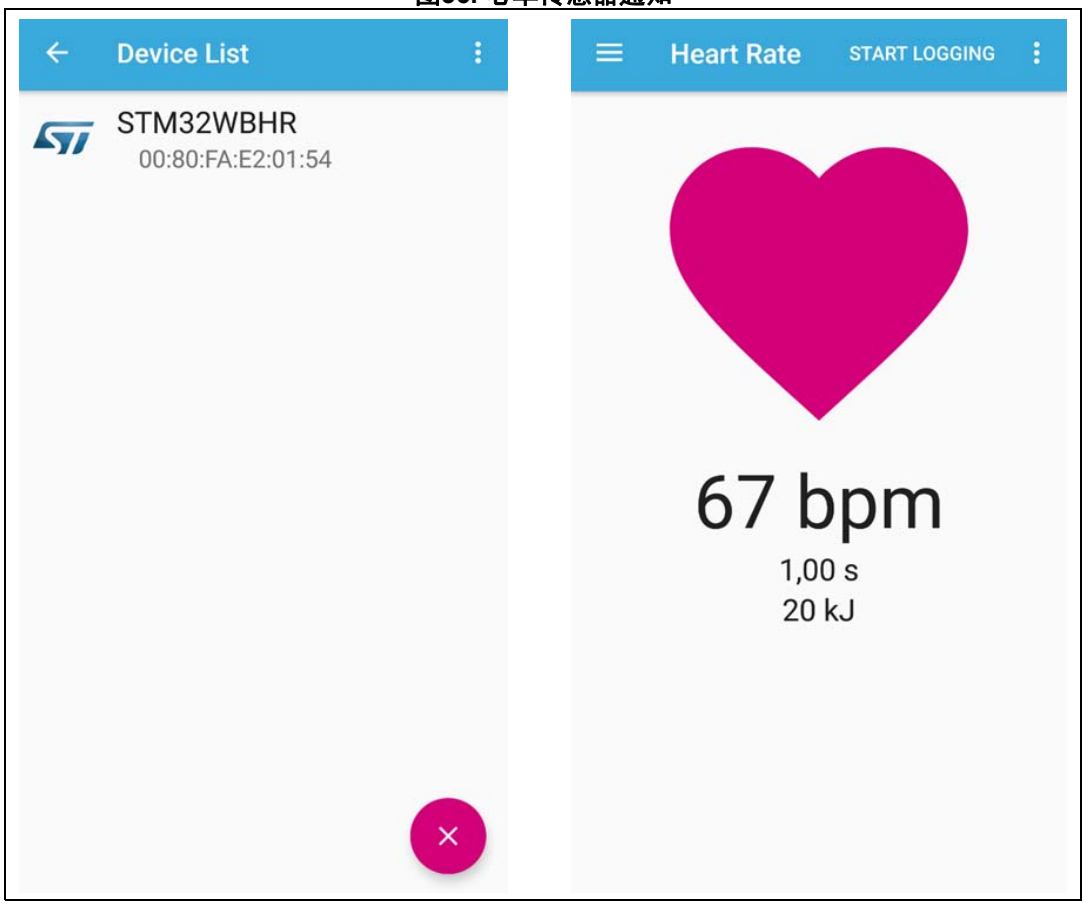

在上传结束后，执行重启流程以启动新应用。

然后，运行新的扫描流程，以发现心率传感器广播数据包并与其连接。

在连接设备后，传感器通知心率测量值。

*注：* *智能手机应用将GATT数据库与远程蓝牙地址关联起来。为了解决此问题，将FUOTA应用广播地址增加1。*

图36. 心率传感器通知




### 7.5.6 如何使用重启请求特征

无论使用什么应用，都可以将重启请求特征整合到服务中，以便以FUOTA应用模式重启应用。

以“BLE_HeartRate_ota”和“BLE_p2pServer_ota”为例，必须在地址0x0800 0700处加载应用，配置如下：

- ble_conf.h to define the OTA Reboot characteristics

```
/******************************************************************************
 * 空中功能（OTA） - STM专利
 ******************************************************************************/
#define BLE_CFG_OTA_REBOOT_CHAR          1 /**< 重启OTA模式特性 */
```

- app_ble.c

```
/**
  * Initialization of ADV - Ad Manufacturer Element - Support OTA Bit Mask
 */
#if(BLE_CFG_OTA_REBOOT_CHAR != 0)
```

```
  manuf_data[sizeof(manuf_data)-8] = CFG_FEATURE_OTA_REBOOT;
#endif
```

- p2p_stm.c to add the characteristics (middleware)

```
#if(BLE_CFG_OTA_REBOOT_CHAR != 0)
   /**
    *  Add Boot Request Characteristic
    */
   aci_gatt_add_char(aPeerToPeerContext.PeerToPeerSvcHdle,
                     BM_UUID_LENGTH,
                     (Char_UUID_t *)BM_REQ_CHAR_UUID,
                     BM_REQ_CHAR_SIZE,
                     CHAR_PROP_WRITE_WITHOUT_RESP,
                     ATTR_PERMISSION_NONE,
                     GATT_NOTIFY_ATTRIBUTE_WRITE,
                     10,
                     0,
                     &(aPeerToPeerContext.RebootReqCharHdle));
#endif
```

- p2p_stm.c to receive the request at GATT level and inform the application (middleware)

```
else if(attribute_modified->Attr_Handle == (aPeerToPeerContext.RebootReqCharHdle + 1))
{
BLE_DBG_P2P_STM_MSG("-- GATT : REBOOT REQUEST RECEIVED\n");
Notification.P2P_Evt_Opcode = P2PS_STM_BOOT_REQUEST_EVT;
Notification.DataTransfered.Length=attribute_modified->Attr_Data_Length;
Notification.DataTransfered.pPayload=attribute_modified->Attr_Data;
P2PS_STM_App_Notification(&Notification);
```

- p2p_server_app.c to manage the reboot request (application)

```
void P2PS_STM_App_Notification(P2PS_STM_App_Notification_evt_t *pNotification)
{
  switch(pNotification->P2P_Evt_Opcode)
  {
#if(BLE_CFG_OTA_REBOOT_CHAR != 0)

    case P2PS_STM_BOOT_REQUEST_EVT:
      APP_DBG_MSG("-- P2P APPLICATION SERVER : BOOT REQUESTED\n");
      APP_DBG_MSG(" \n\r");

      *(uint32_t*)SRAM1_BASE = *(uint32_t*)pNotification->DataTransfered.pPayload;
      NVIC_SystemReset();
      break;
#endif
```

### 7.5.7 CPU1应用的电源故障恢复机制

在更新CPU1应用时，BLE_ota应用提供电源故障恢复机制。

在CPU1应用固件更新期间被用来管理电源故障的两个标签为：

1. MagicKeywordAddress：必须在要加载的二进制映像的开头0x140处进行映射
2. MagicKeywordvalue：由BLE_ota应用在MagicKeywordAddress执行检查。

在烧录新的应用时，如果连接断开，BLE_ota应用将检测故障并自动擦除已编程扇区。该机制可防止在错误的应用上重启。

```
/**
 * These are the two tags used to manage a power failure during CM4 Application OTA FW Update
 * The MagicKeywordAdress shall be mapped @0x140 from start of the binary image
 * The MagicKeywordvalue is checked in the ble_ota application
 */
PLACE_IN_SECTION("TAG_OTA_END") const uint32_t MagicKeywordValue = 0x94448A29 ;
PLACE_IN_SECTION("TAG_OTA_START") const uint32_t MagicKeywordAddress = (uint32_t)&MagicKeywordValue;
```

```
define region OTA_TAG_region   = mem:[from (__ICFEDIT_region_ROM_start__ + 0x140)   to (__ICFEDIT_region_ROM_start__ + 0x140 + 4)];

define block CSTACK    with alignment = 8, size = __ICFEDIT_size_cstack__   { };
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__     { };

initialize by copy { readwrite };
do not initialize  { section .noinit,
                     section MAPPING_TABLE,
                                        section MB_MEM1 };

place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };

keep { section TAG_OTA_START};
keep { section TAG_OTA_END };
place in OTA_TAG_region   { section TAG_OTA_START };
place in ROM_region   { readonly, last section TAG_OTA_END };
```

## 7.6 应用提示

### 7.6.1 如何设置蓝牙设备地址

所有蓝牙设备都必须有一个唯一标识它们的地址。

STM32WB器件支持下列地址类型：

- 公共地址
- 随机地址（可以是静态或私有地址）。

设备地址可以是公共或随机地址。公共和随机设备地址的长度都是48位，并表示为冒号分隔的十六进制值（例如：AA:BB:CC:DD:EE:FF）。

公共设备地址必须按照IEEE 802-2001标准， 使用从IEEE注册管理机构获得的有效的组织唯一标识符（OUI）进行创建。公共设备地址被称为MAC地址。

关于BLE设备如何能够生成随机地址的详细信息，读者可参考核心规范v5.0。

STM32WB提供64位唯一器件标识

- 24位公司ID（ST的为0x00 80 E1）
- 8位器件ID（STM32WB的为0x05）
- 32位唯一器件编号用于区分每个器件。

如果应用需要使用其他公共地址，必须从合适的机构获取地址，然后将其存储在最终产品的永久存储位置（微控制器闪存或OTP中，或外部存储区）。

在STM32WB初始化阶段，应用必须配置此地址。

设置公共地址的ACI命令为：

tBleStatus aci_hal_write_config_data(uint8_t offset, uint8_t len, const uint8_t *val).

参数设置必须如下所示：

- 偏移：0x00
- 长度：0x06
- 值：指向公共地址值的指针，例如：0xaabbccddeeff（6字节数组）。

在开始任何BLE操作前和每次上电或复位后，应用微处理器必须向无线微处理器发送命令aci_hal_write_config_data，，原因是命令aci_hal_write_config_data不会系统性地将数据保存在闪存中。

下面的伪代码示例描述了如何从应用设置MAC地址：

```
uint8_t bdaddr[] = {0xFF, 0xEE, 0xDD, 0xCC, 0xBB, 0xAA};
ret=aci_hal_write_config_data(0x00, 0x06, bdaddr);
if(ret) { PRINTF(”Setting address failed.\n”)}
```

BLE设备还可以使用随机地址。可以使用tBleStatus aci_hal_read_config_data(uint8_t offset, uint16_t data_len, uint8_t *data_len_out_p, uint8_t *data);命令（偏移参数设置为0x80）从应用读取地址值。

或者，应用可以在每次复位后使用int hci_le_set_random_address(tBDAddr bdaddr)命令从外部主机处理器设置随机地址。如果不通过hci_le_set_random_address命令设置随机地址，则如上文所述由协议栈独立处理地址生成。

STM32WB设备的64位UID可用于导出唯一的BLE 48位设备地址。还可以从OTP寄存器获取BLE 48位设备地址。

```
const uint8_t* BleGetBdAddress( void ) {
 uint8_t *otp_addr;
 const uint8_t *bd_addr;
 uint32_t udn;
 uint32_t company_id;
 uint32_t device_id;
 udn = LL_FLASH_GetUDN();

 if(udn != 0xFFFFFFFF) {
  company_id = LL_FLASH_GetSTCompanyID();
  device_id = LL_FLASH_GetDeviceID();
```

```
    bd_addr_udn[0] = (uint8_t)(udn & 0x000000FF);
    bd_addr_udn[1] = (uint8_t)( (udn & 0x0000FF00) >> 8 );
    bd_addr_udn[2] = (uint8_t)( (udn & 0x00FF0000) >> 16 );
    bd_addr_udn[3] = (uint8_t)device_id;
    bd_addr_udn[4] = (uint8_t)(company_id & 0x000000FF);;
    bd_addr_udn[5] = (uint8_t)( (company_id & 0x0000FF00) >> 8 );

    bd_addr = (const uint8_t *)bd_addr_udn;
  }
  else {
   otp_addr = OTP_Read(0);
    if(otp_addr) {
     bd_addr = ((OTP_ID0_t*)otp_addr)->bd_address;
     }
     else {
      bd_addr = M_bd_addr;
    }
  }
 return bd_addr;
}
```

### 7.6.2 如何将任务添加到调度器

- 声明任务ID - app_conf.h -

```
/**< Add in that list all tasks that may send a ACI/HCI command */
typedef enum
{
    CFG_TASK_ADV_CANCEL_ID,
    CFG_TASK_SW1_BUTTON_PUSHED_ID,
    CFG_TASK_HCI_ASYNCH_EVT_ID,
    CFG_LAST_TASK_ID_WITH_HCICMD,  /**< Shall be LAST in the list */
} CFG_Task_Id_With_HCI_Cmd_t;

/**< Add in that list all tasks that never send a ACI/HCI command */
typedef enum
{
    CFG_FIRST_TASK_ID_WITH_NO_HCICMD = CFG_LAST_TASK_ID_WITH_HCICMD - 1,
/**< Shall be FIRST in the list */
    CFG_TASK_SYSTEM_HCI_ASYNCH_EVT_ID,
    CFG_LAST_TASK_ID_WITHO_NO_HCICMD                    /**< Shall be LAST in the
list */
} CFG_Task_Id_With_NO_HCI_Cmd_t;
#define UTIL_SEQ_CONF_TASK_NBR    CFG_LAST_TASK_ID_WITHO_NO_HCICMD
```

- 通过回调函数 - “取消广播” - app_ble.c注册任务

```
SCH_RegTask(CFG_TASK_ADV_CANCEL_ID, Adv_Cancel);
```

- 启动优先任务 - app_ble.c

```
SCH_SetTask(1 << CFG_TASK_ADV_CANCEL_ID, CFG_SCH_PRIO_0);
```

### 7.6.3 如何使用定时器服务

- 通过回调函数创建定时器

```
/**
  * 创建定时器用于处理LED熄灭
  */
HW_TS_Create(CFG_TIM_PROC_ID_ISR, &(BleApplicationContext.SwitchOffGPIO_timer_Id),
hw_ts_SingleShot, Switch_OFF_GPIO);
```

- 启动具有超时的定时器

```
HW_TS_Start(BleApplicationContext.SwitchOffGPIO_timer_Id, (uint32_t)LED_ON_TIMEOUT);
```

- 停止定时器

```
HW_TS_Stop(BleApplicationContext.SwitchOffGPIO_timer_Id);
```

- 回调函数示例

```
static void Switch_OFF_GPIO(){
 BSP_LED_Off(LED_GREEN);
}
```

### 7.6.4 如何启动BLE协议栈 - SHCI_C2_BLE_Init()

```
SHCI_C2_Ble_Init_Cmd_Packet_t STSPIN233；
 {
   0,0,0,                    /**< Header unused */
   0,                        /** pBleBufferAddress not used */
   0,                        /** BleBufferSize not used */
   CFG_BLE_NUM_GATT_ATTRIBUTES,
   CFG_BLE_NUM_GATT_SERVICES,
   CFG_BLE_ATT_VALUE_ARRAY_SIZE,
   CFG_BLE_NUM_LINK,
   CFG_BLE_DATA_LENGTH_EXTENSION,
   CFG_BLE_PREPARE_WRITE_LIST_SIZE,
   CFG_BLE_MBLOCK_COUNT,
   CFG_BLE_MAX_ATT_MTU,
   CFG_BLE_SLAVE_SCA,
   CFG_BLE_MASTER_SCA,
   CFG_BLE_LSE_SOURCE,
   CFG_BLE_MAX_CONN_EVENT_LENGTH,
   CFG_BLE_HSE_STARTUP_TIME,
   CFG_BLE_VITERBI_MODE,
   CFG_BLE_LL_ONLY,
   0                         /** TODO Should be read from HW */
 };
```

### CFG_BLE_NUM_GATT_ATTRIBUTES

与特定BLE用户应用中可存储在GATT数据库中的所有所需特征（不包括服务）相关的属性记录的最大数量

对于每个特征，属性的数量为2到5，具体取决于特征属性：

- 最少为2个（一个用于声明，一个用于值）
- 为每个附加属性再添加一条记录：通知或指示、广播、扩展属性。

由于在初始化GATT和GAP层时，与标准属性配置文件和GAP服务特性相关的记录会自动增加，总计算值必须增加9。

- 最小值：<用户属性的数量> + 9
- 最大值：取决于用户应用定义的GATT数据库

### CFG_BLE_NUM_GATT_SERVICES

定义GATT数据库中可存储的最大服务数量。注意，GAP和GATT服务是在初始化时自动添加的，因此该参数必须是用户服务的数量+2。

- 最小值：<用户服务的数量> + 2
- 最大值：取决于用户应用定义的GATT数据库

### CFG_BLE_ATT_VALUE_ARRAY_SIZE

属性值的存储区大小

attrValueArrSize值的每个特征属性如下：

- 特征值长度 +：
  - 5字节（当特征UUID为16位时）
  - 19字节（当特征UUID为128位时）
  - 2字节（当特征具有服务器配置描述符时）
  - 2字节 * CFG_BLE_NUM_LINK（当特征具有客户端配置描述符时）
  - 2字节（当特征具有扩展属性时）

attrValueArrSize值的每个描述属性如下：

- 描述符长度

### CFG_BLE_NUM_LINK

支持的最大BLE链路数量

- 最小值：1
- 最大值：8

### CFG_BLE_DATA_LENGTH_EXTENSION

禁用/启用BLE 5.0数据包长度扩展功能

- 禁用：0
- 启用：1

### CFG_BLE_PREPARE_WRITE_LIST_SIZE

支持的 “准备写入请求”的最大数量。可使用下列DEFAULT_PREP_WRITE_LIST_SIZE宏命令计算要求的最小值：

```
#define DIV_CEIL(x, y)       (((x) + (y) - 1) / (y))

/**
 * DEFAULT_ATT_MTU：GATT必须支持的最小mtu值
 * 5.2.1 ATT_MTU, BLUETOOTH SPECIFICATION Version 4.2 [Vol 3, Part G]
 */
#define DEFAULT_ATT_MTU                 (23)
/**
 * DEFAULT_MAX_ATT_SIZE: maximum attribute size.
 */
#define DEFAULT_MAX_ATT_SIZE            (512)

/**
 * PREP_WRITE_X_ATT(max_att)：计算需要的准备写入请求数量
 * 当使用的ATT_MTU值等于DEFAULT_ATT_MTU (23)时，
 * 写入大小为max_att的特征。
 */
#define PREP_WRITE_X_ATT(max_att)       (DIV_CEIL(max_att, DEFAULT_ATT_MTU - 5U) * 2)
/**
 * DEFAULT_PREP_WRITE_LIST_SIZE：默认的最小准备写入列表大小。
 */
#define DEFAULT_PREP_WRITE_LIST_SIZE    PREP_WRITE_X_ATT(DEFAULT_MAX_ATT_SIZE)
```

- 最小值：参见上文的宏命令。
- 最大值：可以指定大于要求的最小值的值，但不建议这样做。

### CFG_BLE_MBLOCK_COUNT

BLE协议栈分配的存储器块数量。可使用下列MBLOCKS_CALC宏命令计算要求的最小值：

```
#define MEM_BLOCK_SIZE                  (32)
/**
 * MEM_BLOCK_X_MTU (mtu)：计算构成ATT所需的内存块数量
```

```
 * ATT_MTU = mtu的数据包。
* 蓝牙规范4.2版“7.2 分片和重组”
 * [Vol 3, Part A]
 */
#define MEM_BLOCK_X_TX (mtu)          (DIV_CEIL((mtu) + 4U, MEM_BLOCK_SIZE) + 1U)
#define MEM_BLOCK_X_RX (mtu, n_link)    ((DIV_CEIL((mtu) + 4U, MEM_BLOCK_SIZE) + 2U) * (n_link) + 1)
#define MEM_BLOCK_X_MTU (mtu, n_link)   (MEM_BLOCK_X_TX(mtu) + MEM_BLOCK_X_RX(mtu, (n_link)))

/**
 * 安全连接所需的最小块数
 */
#define MBLOCKS_SECURE_CONNECTIONS     (4)

/**
 * MBLOCKS_CALC(pw, mtu, n_link)：协议栈需要的最小缓冲区数。
* 这是最小推荐值，它取决于：
 *  - pw：准备写入列表的大小
 *  - mtu：ATT_MTU的大小
 *  - n_link：同步连接的最大数量
 */
#define MBLOCKS_CALC(pw, mtu, n_link)  ((pw) + MAX(MEM_BLOCK_X_MTU(mtu, n_link), (MBLOCKS_SECURE_CONNECTIONS)))
```

- 最小值：参见上文的宏命令。
- 最大值：更大的值可以改善数据吞吐量性能，但会使用更多内存。

### CFG_BLE_MAX_ATT_MTU

支持最大ATT MTU大小。

- 最小值：23
- 最大值：512

### CFG_BLE_SLAVE_SCA

在连接了BLE的从设备模式下，使用睡眠时钟精度（ppm值）计算窗口拓宽（结合主设备在CONNECT_REQ PDU中发送的睡眠时钟精度），请参考BLE 5.0规范第6卷B部分4.5.7和4.2.2节。

- 最小值：0
- 最大值：500（规范允许的最坏可能）

### CFG_BLE_MASTER_SCA

主设备模式下处理的睡眠时钟精度。它用于确定连接和广播事件时序。在从设备使用的CONNEC_REQ PDU中发送给从设备，用于计算窗口拓宽，参见*CFG_BLE_SLAVE_SCA*和*[7]*v5.0第6卷B部分4.5.7和4.2.2节。

可能的值：

- 251 ppm 变为 500 ppm：0
- 151 ppm 变为 250 ppm：1
- 101 ppm 变为 150 ppm：2
- 76 ppm 变为 100 ppm：3
- 51 ppm 变为 75 ppm：4
- 31 ppm 变为 50 ppm：5
- 21 ppm 变为 30 ppm：6
- 0 ppm 变为 20 ppm：7

### CFG_BLE_LSE_SOURCE

32 kHz低速时钟源。

- 外部晶振LSE: 0 - 无校准
- 内部RO（LSI）：1 - 由于该振荡器的精度可能随外部条件（温度）而变化，将每秒执行一次校准以确保对时间敏感的BLE操作正确执行。

### CFG_BLE_MAX_CONN_EVENT_LENGTH

此参数决定了从设备连接事件的最长持续时间。 当达到此持续时间时，从设备关闭当前连接事件（无论主设备在HCI_CREATE_CONNECTION HCI命令中指定的CE_length参数是多少），单位时间长度为625/256 µs（~2.44 µs）。

- 最小值：0（如果指定了0，主设备和从设备在每个连接事件将只执行一次TX-RX交换）。
- 最大值：1638400（4000 ms）。可以指定更大的值（最大值0xFFFFFFFF），但会导致连接时间达到指定的最大值4000 ms。在这种情况下，将不应用该参数，并且不缩短在从设备上计算的预计CE长度。

### CFG_BLE_HSE_STARTUP_TIME

高速（16或32 MHz）晶体振荡器的启动时间，以625/256 µs (~2.44 us)为单位

- 最小值：0
- 最大值：820（~2 ms）。可以指定更大的值，但在协议栈中实现的值被强制为~2 ms。

### CFG_BLE_VITERBI_MODE

BLE LL接收中的Viterbi实现

- 0：启用
- 1：禁用

#### CFG_BLE_LL_ONLY

启用/禁用仅BLE链路层模式 在该模式下，主机协议栈（GAP/GATT/SMP等）代码不存在；只实现链路层且只有链路层可以通过HCI命令进行处理。

- “仅链路层”模式禁用：0
- “仅链路层”模式启用：1

### 7.6.5 如何使数据吞吐量最大化

当GATT服务器通知与下列链路层参数一起使用时，可达到最大数据吞吐量：

- 连接间隔：50 ms
- Min_CE_Length和Max_CE_Lenght：0x50（50 ms）

在建立连接后，主设备发送aci_gatt_exchange_config以获取MAX_ATT_MTU值。

数据交换被限制在（MAX_ATT_MTU – 3）以下，该值对应的是最大通知长度。

- 如果支持，将链路设置为2M

为避免分片，LE数据的最大PDU_length = 247 (251 - 4)：

- 使用hci_le_set_data_length命令hci_le_set_data_length(conn_handle, 251, 2120)

为避免分片：

- 如果MAX_ATT_MTU = 250且le_data_length = 251，则要传输的最大数据长度 = 244 (251 - 4 - 3)
- 如果MAX_ATT_MTU = 156且le_data_length = 251，则要传输的最大数据长度 = 153 (156 - 3)

### 7.6.6 如何添加自定义BLE服务

在所有BLE应用中，可以添加与源代码或二进制文件中提供的现有服务平行的自定义服务。CPU1接收到的所有GAP/GATT事件都将到达服务控制器（\Middlewares\ST\STM32_WPAN\ble\svc\Src中的svc_ctl.c），服务控制器负责初始化所有BLE服务并将GATT事件转发到已注册的BLE服务。*图 24：心率项目 - 中间件与用户应用之间的交互*所示为流程示例。

每个服务都必须有custom_xxx.c和custom_xxx.h。

只能提供三个公共接口给用户：

void Custom_xxx_Init( void )

它在custom_xxx.c中实现并执行下列操作：

- 创建服务和添加特征
- 通过API SVCCTL_RegisterSvcHandler()将回调注册到控制器

必须在应用中实现函数SVCCTL_InitCustomSvc()以调用Custom_xxx_Init()。

使用SVCCTL_RegisterSvcHandler()注册的回调被用于接收来自服务控制器的GATT事件。回调类型必须是SVCCTL_EvtAckStatus_t (*SVC_CTL_p_EvtHandler_t)(void *p_evt)。

根据BLE服务定义，接收到的GATT事件要么只在custom_xxx.c模块中处理，要么必须通过通知Custom_xxx_Notification()转发到应用（大多数情况）。每个GATT事件只与一个BLE服务相关。为避免服务控制器调用所有已注册BLE服务来报告接收到的事件，回调返回服务控制器，而GATT已处理或被忽略。

可以返回3个值：

1. SVCCTL_EvtNotAck：表示GATT事件与该BLE服务无关。服务控制器持续向其他已注册BLE服务报告此GATT事件，直至获得确认。如果所有已注册BLE服务均未确认GATT事件，将通过通知SVCCTL_App_Notification()将其报告给应用。
2. SVCCTL_EvtAckFlowEnable：表示GATT事件已处理且服务控制器不将其报告给其他已注册BLE服务或应用。
3. SVCCTL_EvtAckFlowDisable：表示GATT事件已确认且服务控制器不将其报告给其他已注册BLE服务或应用。但是，GATT事件尚未处理。服务控制器通知传输层不应丢弃此事件。在这种情况下，在调用命令hci_resume_flow()前，传输层将不再报告任何事件。一旦流程继续，将再次报告未确认事件。请注意，所有BLE用户HCI事件均不再报告，且不仅仅是向没有确认GATT事件的BLE服务报告的事件。

tBleStatus Custom_xxx_UpdateChar( Custom_xxx_ChardId_t ChardId, uint8_t * p_payload )

此API被应用用来更新服务器特征。接口ChardId与要发送到BLE协议栈的UUID之间的映射必须在BLE服务中实现。

void Custom_xxx_Notification( Custom_xxx_Notification_t *p_notification )

此API用于向应用报告（若相关）BLE服务接收到的GATT事件。

# 8 在HCI层接口之上构建BLE应用

CPU2可用作BLE HCI层协处理器。在这种情况下，用户必须实现自己的HCI应用或使用现有的开源BLE主机协议栈。

大多数BLE主机协议栈使用UART接口与BLE HCI协处理器通信。STM32WB器件上的等效物理层是信箱（Mailbox），如*第 13.2 节：信箱（Mailbox）接口*所述。

信箱（Mailbox）为BLE和系统通道提供了一个接口。BLE主机协议栈构建要通过信箱（Mailbox）上BLE通道发送的命令缓冲区，并且必须提供接口用于报告通过信箱（Mailbox）接收到的事件。除了通过信箱（Mailbox）完成BLE主机协议栈自适应，用户还必须在可以发布异步数据包时通知信箱（Mailbox）驱动程序。

不通过BLE主机协议栈处理系统通道。用户必须实现自己的传输层，以构建要发送到信箱（Mailbox）驱动程序的系统命令缓冲区并管理从信箱（Mailbox）接收到的事件（包括向信箱（Mailbox）驱动程序释放异步缓冲区的通知），或者使用信箱（Mailbox）扩展驱动程序（如*第 13.3 节：信箱（Mailbox）接口 - 扩展*所述），以便在负责构建系统命令缓冲区和管理系统异步事件的传输层之上提供接口。

BLE_TransparentMode项目可用作使用信箱（Mailbox）在BLE HCI层协处理器之上构建应用（如*第 13.2 节：信箱（Mailbox）接口*所述）的例子。

# 9 Thread

## 9.1 概述

CPU2内核中集成的Thread协议栈由OpenThread提供，是Thread网络协议的开源实现，由Nest发布。

OpenThread提供多个API，可解决协议栈内不同层面的不同服务。 所有这些API（记录在STM32WB固件包中）均导出到CPU1内核上，可以被应用直接使用。

STM32WB固件包附带多个演示如何运行简单Thread应用的示例。为了运行这些应用， 需要下载合适的CPU2固件二进制文件。

有三个主要的MO固件可供使用，如*表 28*所示。

**表28. Thread可以使用的MO固件**

| CPU2固件库 | 特性 | 备注 |
|---|---|---|
| stm32wb5x_Thread_FTD_fw.bin | FTD:<br>全功能Thread设备 | 设备支持除边界路由器外的所有Thread角色（主导设备（Leader）、路由器、终端设备和休眠终端设备）。<br>Thread角色描述见*第 13.10.5 节*。 |
| stm32wb5x_Thread_MTD_fw.bin | MTD:<br>最低功能Thread设备 | 设备只能充当终端设备或休眠终端设备。<br>MTD配置对比FTD配置的优势是需要的内存较少。 |
| stm32wb5x_BLE_Thread_fw.bin | 静态并发模式 | 设备在一个二进制库中集成了两个无线协议栈（BLE和Thread）。 |

## 9.2 如何开始

开始使用Thread的最简单方法是使用下面两个应用：

- Thread_Cli_Cmd：显示如何通过CLI命令控制Thread协议栈。CLI（命令行接口）命令通过UART从HyperTerminal（PC）发送到开发板，可用于创建简单用例。该应用用于运行认证测试（Thread GRL测试工具）。
- Thread_Coap_Generic：需要两块P-NUCLEO-WBxx开发板。它显示了一块板与另一块板交换CoAP信息的情景。在该应用中，一个设备作为主导设备（Leader），另一个设备作为终端设备或路由器。

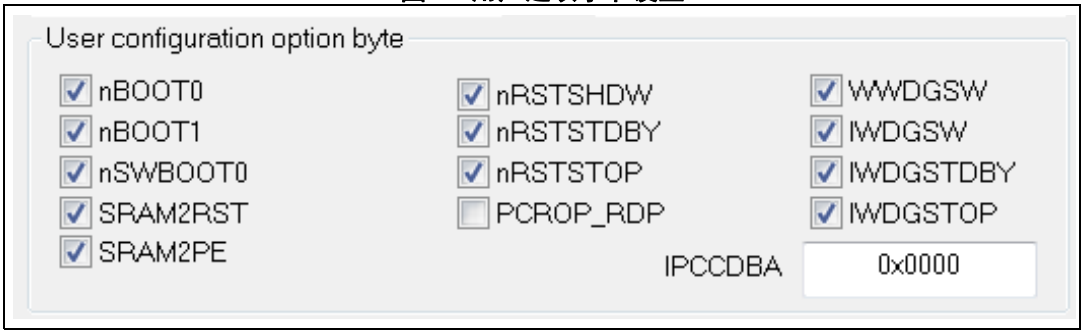
这两个应用程序在 STM32WB 固件包中提供，并带有相关的 readme.txt 文件。

## 9.3 Thread配置

在启动任何Thread应用之前，用户必须：

- 下载合适的固件：Thread MTD、Thread FTD或Thread静态模式
- 使用正确的选项字节。

**图37. 用户选项字节设置**




**注意：** OpenThread协议栈提供了多个编译标记，用于设置不同配置。然而，由于STM32WB内部的无线协议栈以二进制形式交付，因此这些标记是固定的，用户不能修改。选择的标记可以在*表 29*列出的文件中看到。

**表29. Thread配置的文件**

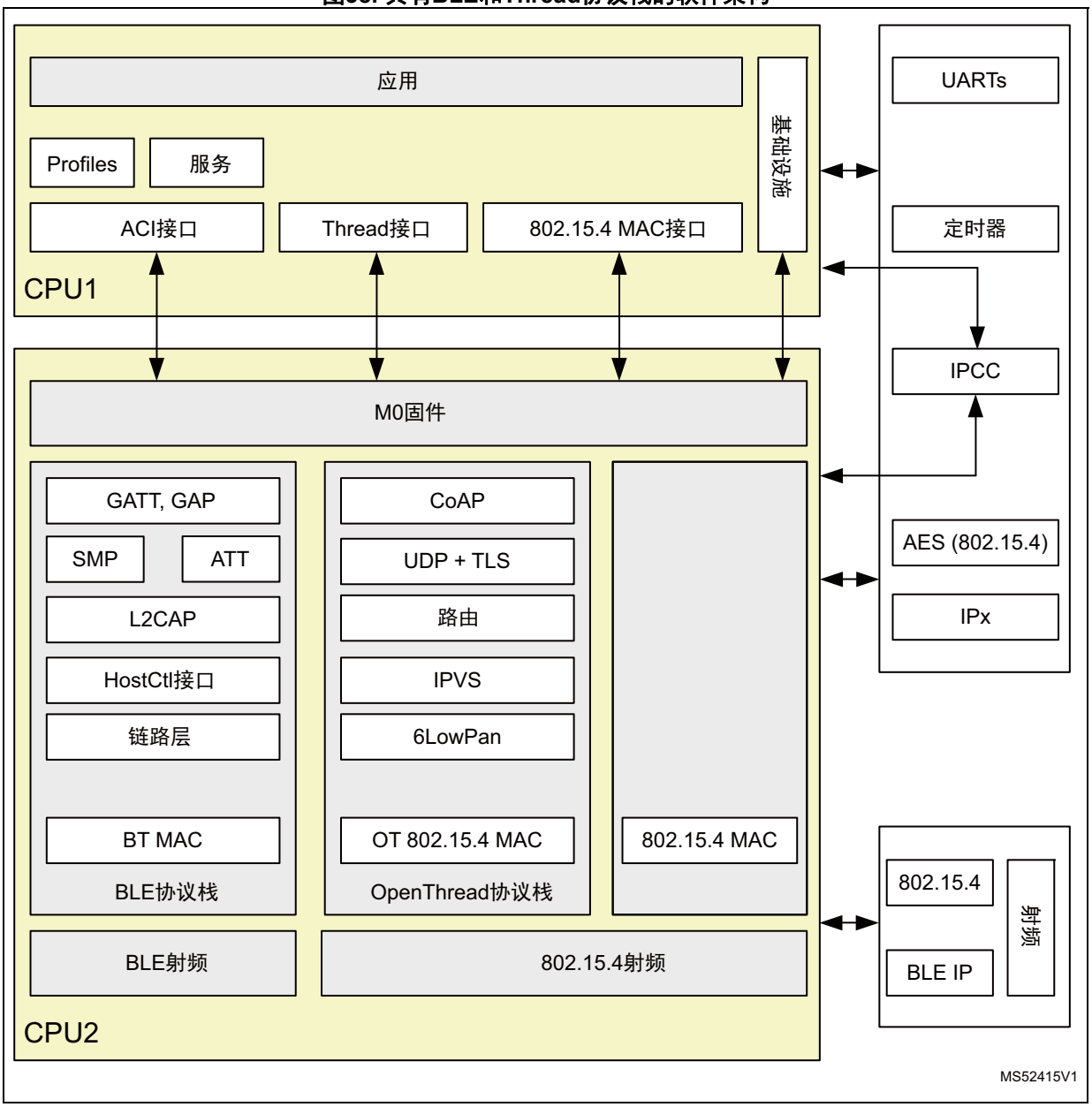
| 文件名 | 配置 |
|---|---|
| openthread_api_config_ftd.h | 将与Thread FTD CPU2固件一起使用 |
| openthread_api_config_mtd.h | 将与Thread MTD CPU2固件一起使用 |
| openthread_api_config_concurrent.h | 将与静态并发模式CPU2固件一起使用 |

在构建Thread应用时，必须根据下载的CPU2固件使用合适的配置文件。此配置文件中的标志用于定义哪些 API 被导出并可用于 CPU1 应用程序。如前所述，用户不得修改这些标志。

## 9.4 架构概述

*图 38*显示了有BLE和Thread两个无线协议栈的整体软件架构。在CPU2上运行的所有代码均以二进制库的形式交付。客户只能访问CPU1内核，并看到在CPU2上运行的固件（相当于一个黑盒）。ACI和Thread接口分别允许用户访问BLE和Thread任务。

图38. 具有BLE和Thread协议栈的软件架构




## 9.5 核间通信

所有OpenThread API均暴露于 CPU1且可用于控制在CPU2上运行的协议栈。STM32WB中间件管理两个内核之间的通信。

当应用调用OpenThread函数时，通过IPCC向CPU2发送同步信息。与该函数相关的参数被保存在共享存储器中。

为确保整个系统保持同步，将暂停OpenThread函数调用，直至命令完成（参见*图 39*）。应用可以注册回调，以便在发生特定事件时得到通知。为确保整个系统保持同步，这些通知也将暂停，如*图 40*所示。

图39. OpenThread函数调用

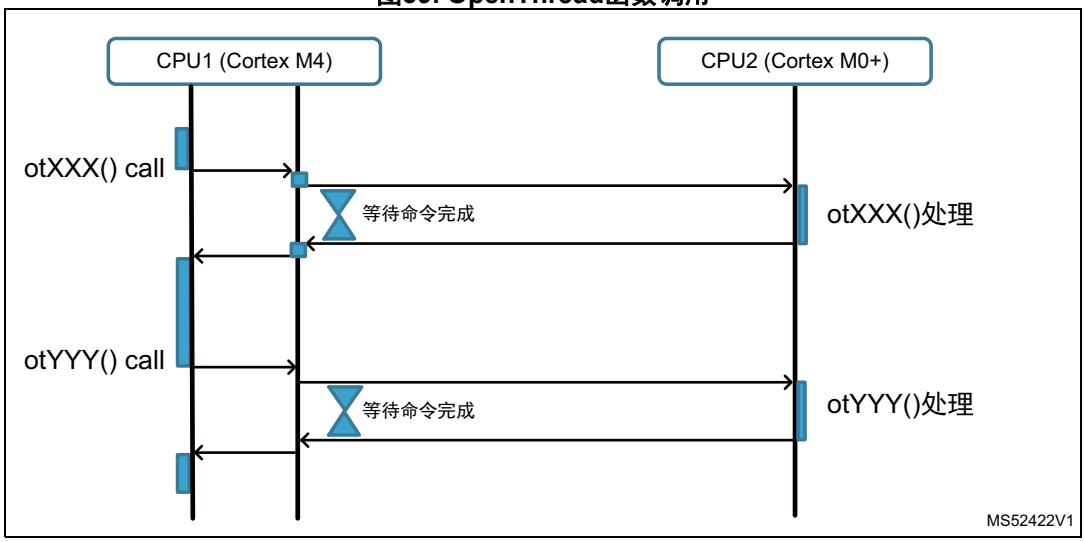



图40. OpenThread回调

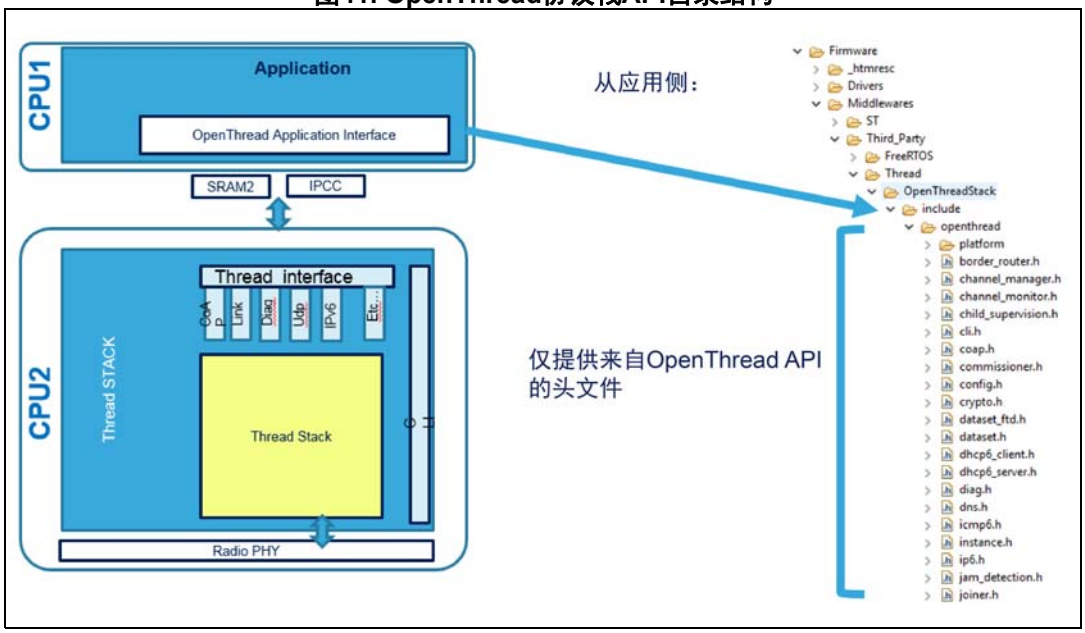

## 9.6 OpenThread API

OpenThread定义了多个API，可在协议栈内寻址不同层面的不同服务：

- 用于管理CoAP服务的函数：otCoapStart()、otCoapSendRequest()
- 用于管理UDP数据报的函数：otUdpOpen()、otUdpConnect()
- 用于管理射频配置的函数：otLinkSetChannel()
- 用于管理IPV6地址的函数：otIp6AddUnicastAddress()

可使用的函数总共有300多个。STM32WB固件包中提供的STM32WBxx_OpenThread_API_User_Manual.chm描述了这些API。

图41. OpenThread协议栈API目录结构




## 9.7 OpenThread API的使用

可以像在一个处理器上运行系统那样使用OpenThreadAPI。Thread接口隐藏了所有多核机制（IPCC、共享存储器），允许CPU1访问在CPU2上运行的OpenThread协议栈。

但是，有两个特性与STM32WB实现OpenThread接口的方式有关，如下文所述。

### 9.7.1 OpenThread实例

许多OpenThread API使用定义了OpenThread实例的参数aInstance作为输入，下面的otThreadSetEnabled()函数示例中用粗体显示了该参数：

otThreadSetEnabled(otInstance ***aInstance***, bool aEnabled)

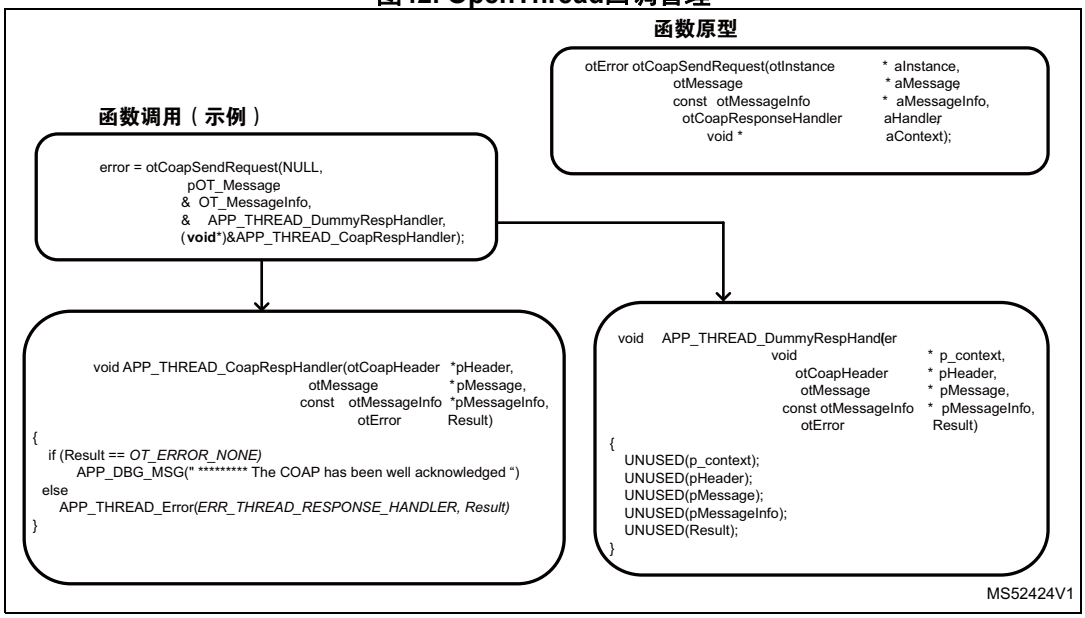
在STWM32WB Thread实现中，OpenThread实例直接位于CPU2固件的开头。CPU1无需注意此参数，它总是设置为NULL（参见以下代码段中的粗体部分）。

```
error = otThreadSetEnabled(NULL, true);
 if (error != OT_ERROR_NONE)
 {
   APP_THREAD_Error(ERR_THREAD_START,error);
 }
}
```

### 9.7.2 OpenThread回调管理

在STWM32WB thread实现中，作为OpenThread函数内部参数传递的回调不遵循标准OpenThread函数的精确原型。这是由于双核架构的限制。应用回调必须在上下文参数中传递，如*图 42*所示。

**图42. OpenThread回调管理**




*注：　了解OpenThread回调管理方式的最简单方式是参考STM32WB固件交付包中提供的不同应用。*

## 9.8 Thread应用的系统命令

有些命令可以从Thread应用调用：

- SHCI_C2_THREAD_Init()：启动Thread协议栈。在初始化阶段结束时调用。
- SHCI_C2_FLASH_StoreData()：将非易失性Thread数据保存在 闪存中。由应用 决定何时必须将数据保存在闪存中（例如：调试阶段后或网络配置后）。

*注：　此操作可能需要几秒钟，只有在没有Thread活动时才能调用。*

- SHCI_C2_FLASH_EraseData()：将非易失性Thread数据从闪存中擦除。

*注：　此操作可能需要几秒钟，只有在没有Thread活动时才能调用。*

- SHCI_C2_CONCURRENT_SetMode()：对并发模式启用或禁用CPU2上的Thread活动。
- SHCI_C2_RADIO_AllowLowPower()：允许或禁止802_15_4射频IP进入低功耗模式。
- SHCI_GetWirelessFwInfo()：读取与加载的无线二进制文件相关的信息。

### 9.8.1 非易失性Thread数据

根据Thread规范，必须将多个值保存在闪存中，以便日后重复使用。这些值与下列实体有关：

1. 有效操作数据集：

   在每次接收到新的有效操作数据集时写入。 只有在调试设备或其他外部实体更新有效操作数据集时才会发生。

2. 待定操作数据集：

   在每次接收到新的待定操作数据集时写入。 只有在调试设备或其他外部实体更新待定操作数据集时才会发生。

3. 网络信息：

   在每次设备角色变化时写入（即：断开的设备、子设备、路由器和主导设备（Leader））。在每次MAC和/或MLE帧计数器值递增至超过特定阈值时写入。

4. 父设备信息：

   在每次有子设备连接到父设备时写入。

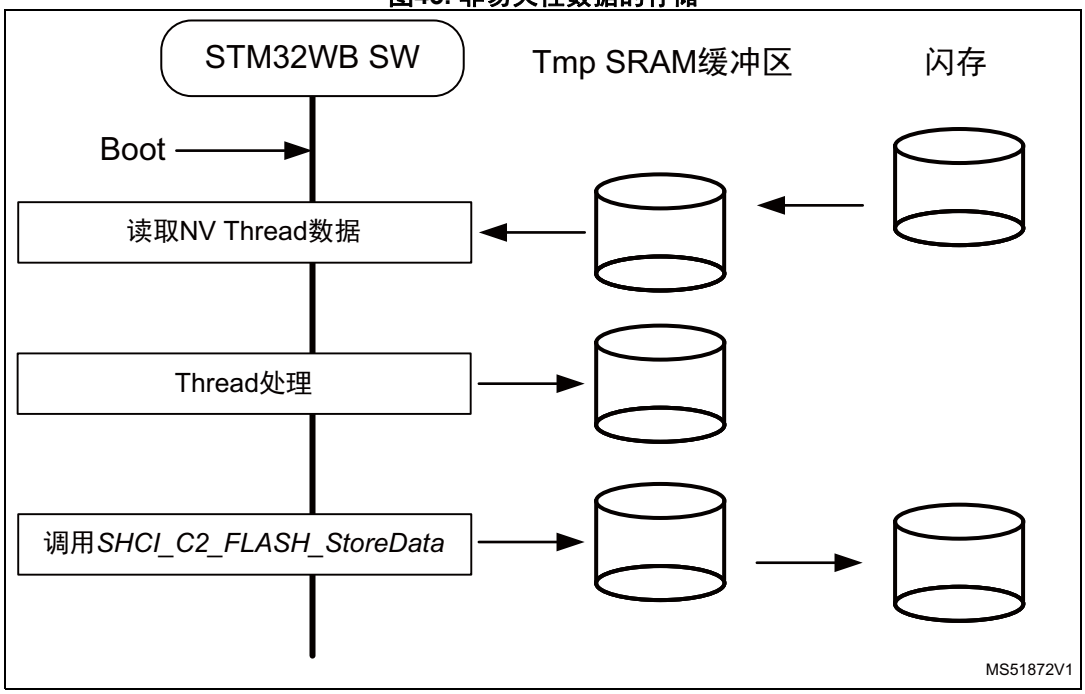
5. 子设备信息：

   在每次将子设备添加到子设备表/从子设备表删除子设备时写入。

在复位后，自动读取闪存中的Thread 数据集。在运行时间，OpenThread定期保存和更新内部SRAM缓冲区中的这些非易失性数据（参见*图 43*）。使用函数SHCI_C2_FLASH_StoreData()将这些非易失性数据强制复制到闪存中，具体取决于应用。由于此操作会阻止对闪存（以及CPU）的访问，必须在没有 实时限制时执行（例如：在Thread停止后）。

*注：* *在调用otInstanceReset ()或otInstanceFactoryReset后自动触发函数SHCI_C2_FLASH_StoreData()。*

**图43. 非易失性数据的存储**



### 9.8.2 低功耗支持

为了达到最小功耗， 必须将设备处于SED（休眠终端设备）模式，并在需要轮询来自其父设备的信息或发送数据时唤醒。设备在大部分时间处于休眠状态并自动进入低功耗模式。低功耗Thread设备可以休眠，并依靠电池供电工作数年。

当系统处于低功耗模式时，一旦应用发送otCmd，系统立即被唤醒并执行命令，然后回到低功耗模式。如果应用连续发送多个otCmd，系统将频繁地唤醒和回到休眠状态。为规避发生多个不必要的短暂唤醒/休眠周期的风险，应用通过函数SHCI_C2_RADIO_AllowLowPower()允许或禁止射频进入低功耗模式。名为Thread_SED_Coap_Multicast的应用中提供了该函数的一个示例。

# 10 OpenThread应用的分步设计

本章提供关于如何在STM32WB上设计和实现OpenThread应用的信息和代码示例。

## 10.1 初始化阶段

STM32WB Thread应用的初始化有几个必要步骤：

- 初始化设备（HAL、复位设备、时钟和电源配置）
- 配置平台（例如：按钮、LED）
- 配置硬件（例如：UART、调试）
- 启动CPU2，然后使其向应用（在CPU1侧）发送系统通知
- 在接收到通知后，应用启动Thread配置。

## 10.2 设置Thread网络

在固件包提供的Thread应用中，总是使用同一个函数APP_THREAD_DeviceConfig()执行Thread网络的设置：

此函数处理以下步骤：

- 擦除持久性Thread参数以便重新开始：otInstanceErasePersistentInfo()
- 注册在节点角色发生变化时OpenThread栈调用的应用回调。（例如，当节点变为路由器时）。使用otSetStateChangedCallback()执行操作。
- 设置通道：otLinkSetChannel()
- 设置PANID：otLinkSetPanId()
- 启用IPv6通信：otIp6SetEnabled()
- 启动CoAP服务器：otCoapStart()
- 将CoAP资源添加到CoAP服务器：otCoapAddResource()
- 启动Thread协议操作：otThreadSetEnabled()

执行上述步骤后：

- 如果此板是该Thread网络上的第一个设备，该节点将变为主导设备（Leader）（例如：Thread的绿色LED点亮）。
- 如果网络中的主导设备（Leader）已存在，该节点将作为路由器或子设备加入网络（Thread 网络示例中的红色LED点亮）。

## 10.3 CoAP请求

受限应用协议（CoAP）是一种专用网络传输协议，适用于受限节点和网络（例如：低功耗有损网络）。

CoAP提供应用端点之间的请求/响应交互模型，支持服务和资源的内置发现，并包含诸如URI和互联网媒体类型等关键网络概念。

*注：* *本节不会详细介绍与CoAP相关的所有OpenThread API，但是会给出主要函数和抽象层（作为STM32WB Thread示例的一部分提供）的概述。*

### 10.3.1 创建otCoapResource

```
typedef struct otCoapResource
{
   const char *          mUriPath; ///< The URI Path string
   otCoapRequestHandler   mHandler; ///< The callback for handling a received request
   void *                mContext; ///< Application-specific context
   struct otCoapResource *mNext;    ///< The next CoAP resource in the list
} otCoapResource;
```

声明如下：

```
static otCoapResource OT_Ressource = {C_RESSOURCE, APP_THREAD_DummyReqHandler,
(void*)APP_THREAD_CoapRequestHandler, NULL};
```

其中：

mUriPath = C_RESSOURCE : name of the resource (eg: “light”)

mHandler = APP_THREAD_DummyReqHandler() : Dummy Request Handler.请参见下方的注释。

mContext = APP_THREAD_CoapRequestHandler() : Handler called when the server receives a COAP request.

mNext = NULL : Not used

*注：* *对于STM32WB实现（双核CPU1/CPU2，具有OpenThread接口封装），必须用虚拟的请求处理函数（不执行任何操作）填充mHandler参数，并将实际的请求处理函数传递至mContext参数。*

### 10.3.2 发送CoAP请求

应用Thread_Coap_Generic使用抽象层来帮助发送CoAP请求。 这是通过以下函数实现的：

```
static void APP_THREAD_CoapSendRequest(otCoapResource* pCoapRessource,
   otCoapType CoapType,
   otCoapCode CoapCode,
   const char *Address,
   uint8_t* Payload,
   uint16_t Size)
```

### 10.3.3 收到CoAP请求

在服务器收到CoAP请求时调用。

CoAP请求处理函数的原型如下：

```
static void APP_THREAD_CoapRequestHandler(otCoapHeader * pHeader,
   otMessage          * pMessage,
   const otMessageInfo  * pMessageInfo)
```

在该函数中，用户可通过调用以下函数读取收到的信息：

otMessageRead()

如果信息类型可确认（使用otCoapHeaderGetType()），则必须发送CoAP响应（参见*第 11.3节：Thread_Coap_Generic*查看示例）。

## 10.4 调试

调试要求一个设备的角色为调试设备，另一个设备的角色为加入设备。调试设备是现有网络中的Thread设备或Thread网络之外充当该角色的设备（例如：移动电话）。

加入设备是准备加入Thread网络的设备。

Thread调试设备用于验证网络上的设备。 它不传输或拥有Thread网络凭证，例如主密钥。

本文档涵盖了基础的基于Mesh的调试，无外部调试设备或边界路由器。

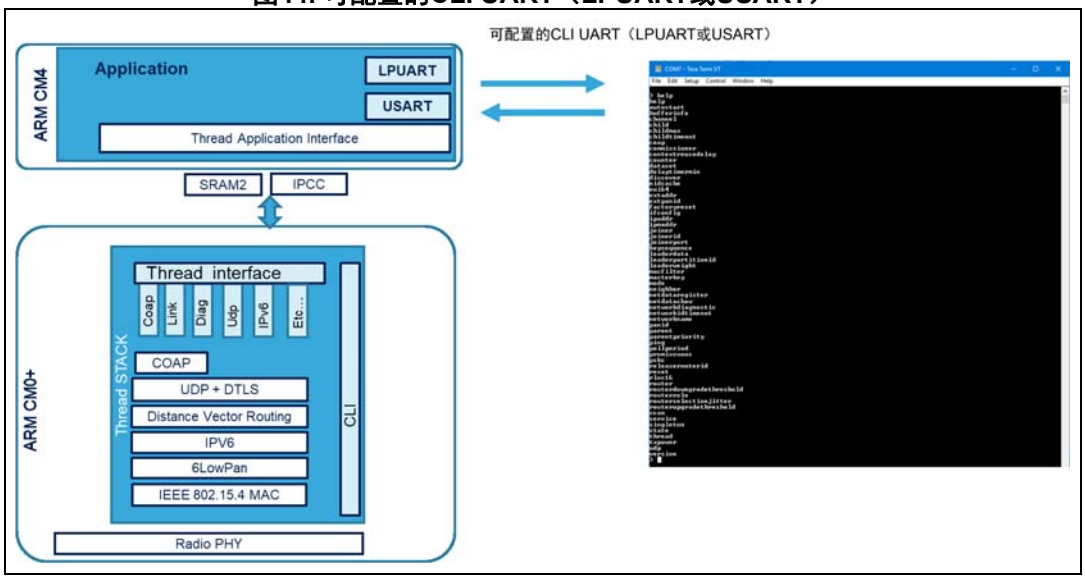
Thread_Commissioning应用演示了简单的调试示例。

## 10.5 CLI

OpenThread协议栈通过命令行接口提供配置和管理API。

认证环境（GRL测试工具）使用CLI执行测试用例。

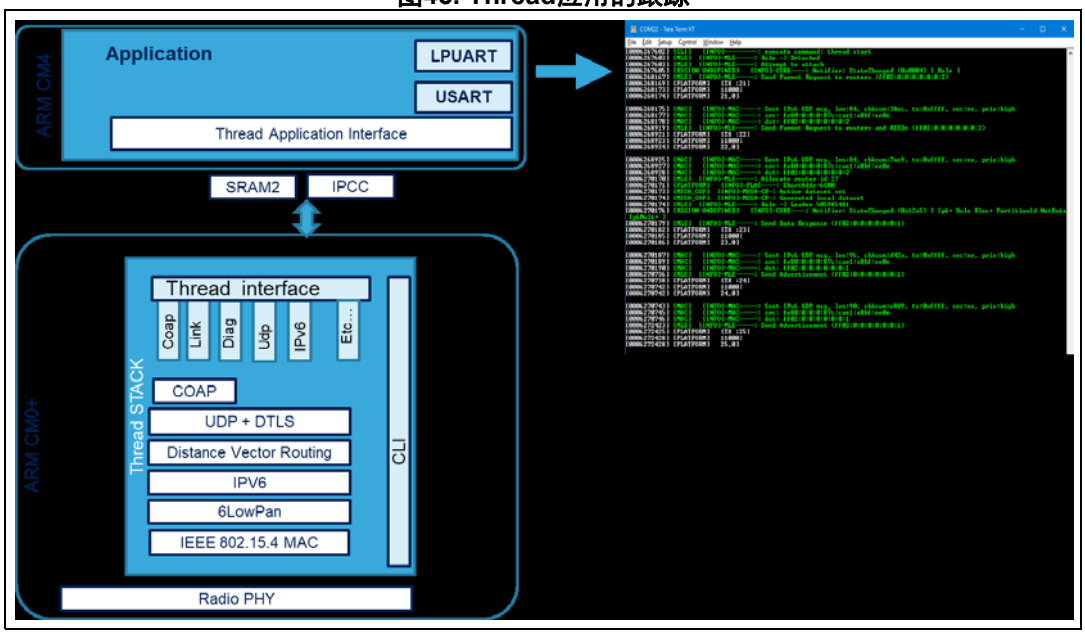
**图44. 可配置的CLI UART（LPUART或USART）**



## 10.6 跟踪

CPU1和CPU2应用的跟踪服务被路由至UART（在编译时使用文件app_conf.h配置）。

**图45. Thread应用的跟踪**




可使用otSetDynamicLogLevel()，通过OpenThread API动态配置OpenThread协议栈跟踪级别。

# 11 STM32WB OpenThread应用

## 11.1 Thread_Cli_Cmd

该应用显示如何通过命令行控制OpenThread协议栈。

通过UART将CLI命令从HyperTerminal（PC）发送到STM32WBxx_NUCLEO板。

该应用用于认证过程。

## 11.2 Thread_Coap_DataTransfer

该应用演示如何使用CoAP信息传输协议传输大数据块。

在该应用中，使用Mesh Local范围和多播寻址类型探查文件传输到的子设备的Mesh Local IP地址。

节点分为两种转发角色：路由器或终端设备。

该应用中使用了两个设备，一个作为主导设备（路由器），另一个作为终端设备（子模式）。

在两块板复位后，其中一块处于主导设备（Leader）模式（绿色LED2点亮），另一块处于子设备模式（红色LED3点亮）。

在一个设备确立子设备模式后，它在多播模式下启动配置流程，以探查主导设备（Leader）的IP地址。

然后，将使用此IP地址在单播模式下启动文件传输流程，蓝色LED发光表示操作成功。

## 11.3 Thread_Coap_Generic

该应用演示了CoAP信息的使用。它提供抽象层用于发送CoAP多播请求。

该应用需要两块STM32WB板。目的是演示两块板彼此交换CoAP信息。在该应用中，一块板作为主导设备（Leader），另一块板作为终端设备或路由器。

## 11.4 Thread_Coap_Multiboard

该应用展示了如何使用CoAP 以单播方式向多块板发送信息。它可以使用2至5块STM32WBxx_NUCLEO板。

该应用的目的是创建一个小型的Thread Mesh网络。

在正确配置设置后，将按以下顺序自动并连续地将CoAP请求从一块板传输到下一块板：

- 板1 → 板2 → (...) → 板n → 板1 → …

每块板都关联了一个特定的IPv6地址：

- fdde:ad00:beef:0:441f:ade1:3fc:1f3a for board number 2
- fdde:ad00:beef:0:442f:ade1:3fc:1f3b for board number 2
- fdde:ad00:beef:0:442f:ade1:3fc:1f3c for board number 3 if present
- fdde:ad00:beef:0:442f:ade1:3fc:1f3d for board number 4 if present
- fdde:ad00:beef:0:442f:ade1:3fc:1f3e for board number 5 if present

## 11.5 Thread_Commissioning

该应用演示了调试设备与连接设备之间的调试过程。

它展示了设备如何通过调试过程将其Thread参数（通道、PAN ID和Masterkey）分配给另一个设备。 。

该应用需要两块STM32WBxx_NUCLEO板。

一个设备作为调试设备，另一个作为连接设备。

在该应用中，调试设备接受其Thread网络中的新连接设备。

## 11.6 Thread_FTD_Coap_Multicast

该应用演示了全功能Thread设备的CoAP多播信息的使用。

*注：* *将与Thread FTD CPU2二进制文件一起使用。*

该应用使用了两个设备，一个作为主导设备（路由器），另一个作为终端设备（子模式）。

在两块板（分别命名为A和B）复位后，一块板处于主导模式（Leader）模式（绿色LED2点亮），另一块板处于子设备模式（红色LED3点亮）。

为了从板A向板B发送CoAP命令，按下板A上的SW1按钮。板B接收到CoAP命令后点亮其蓝色LED1。再次按下同一按钮后，蓝色LED1熄灭。

可以从板B向板A发送相同CoAP。

## 11.7 Thread_SED_Coap_Multicast

该应用演示了从休眠终端设备发送的CoAP多播信息的使用。

*注：* *将与Thread MTD CPU2二进制文件一起使用。*

所述用例需要两块板：

- 一块板在FTD模式下充当主导设备（路由器）（板A）
- 另一块板在MTD模式下充当休眠终端设备（板B）

充当主导设备(Leader)的板必须烧写FTD应用固件：使用CPU2上的应用“Thread_FTD_Coap_Multicast” + Thread FTD二进制文件。

充当休眠终端设备的板必须烧写CPU2上的MTD应用固件“Thread_SED_Coap_Multicast” + Thread MTD二进制文件。

在两块板复位后，一块板（A）自动进入主导设备(Leader)模式（绿色LED2点亮），几秒钟后另一块板（B）进入休眠终端设备模式（红色LED3点亮）。

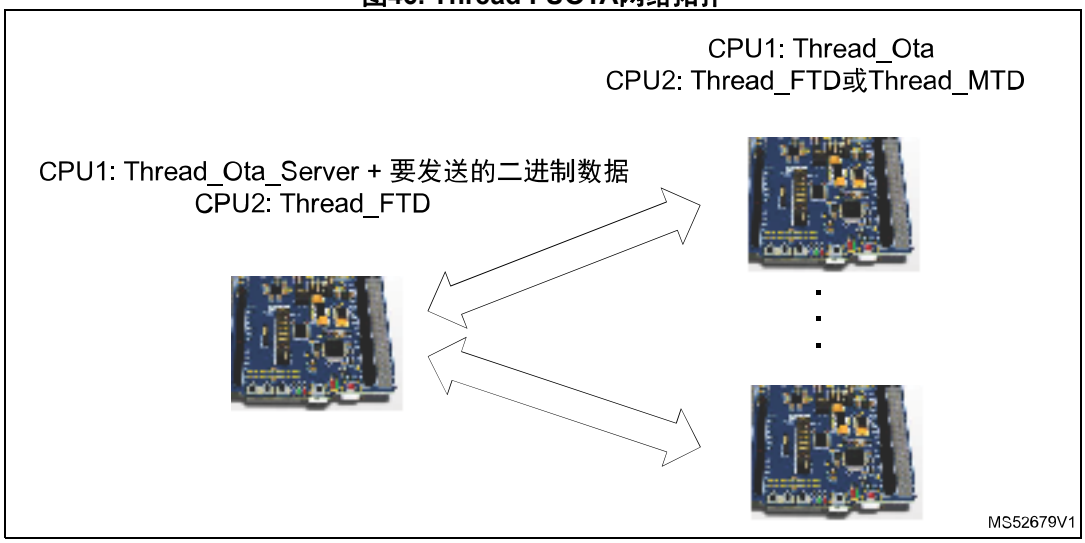
在该阶段，这两块板属于同一个Thread网络，设备2每秒向设备1发送CoAP多播请求以点亮/熄灭其蓝色LED。

## 11.8 Thread FUOTA

### 11.8.1 原理

目的是使用Thread协议更新远程设备上的CPU1应用二进制文件或CPU2无线协处理器二进制文件。

**图46. Thread FUOTA网络拓扑**




该Thread需要至少两块采用Thread协议并运行特定应用的STM32WBxx板（参见*图 46*）

- 一块运行Thread_Ota_Server应用的板
- 一块或更多块运行Thread_Ota应用的板

一次只能在一个设备上执行FUOTA流程。

服务器发起FUOTA配置流程，一个客户端必须响应。 如果多个客户端将逐一更新。。

### 11.8.2 存储器映射

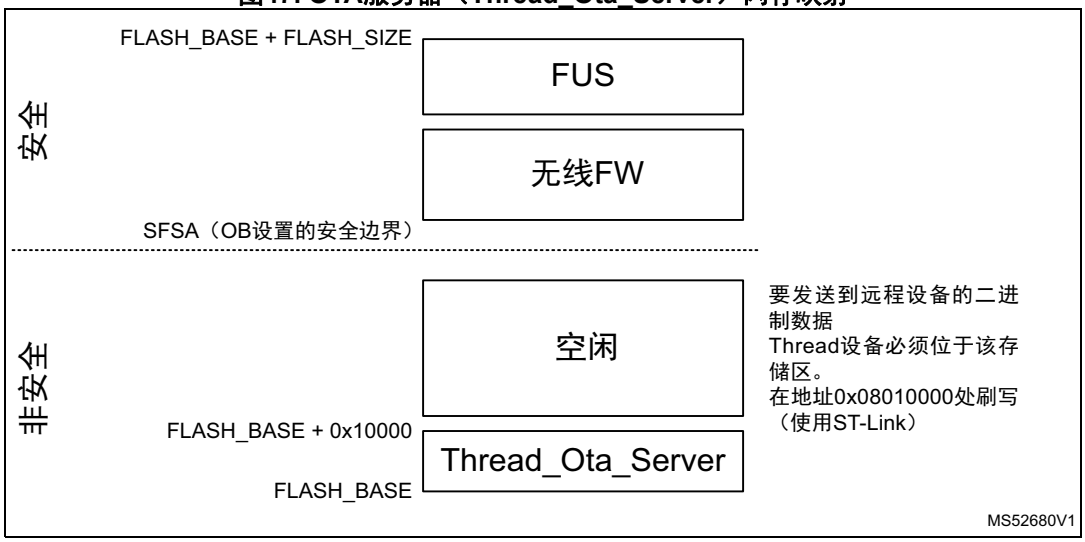
#### 服务器端

必须先将要安装在远程设备上的二进制文件（用于CPU1或CPU2更新）刷写到服务器端的“空闲”存储区（参见*图 47*）。

要传输的二进制文件的大小最大等于：

空闲区大小 = SFSA地址 - (FLASH_BASE - 0x8010000)

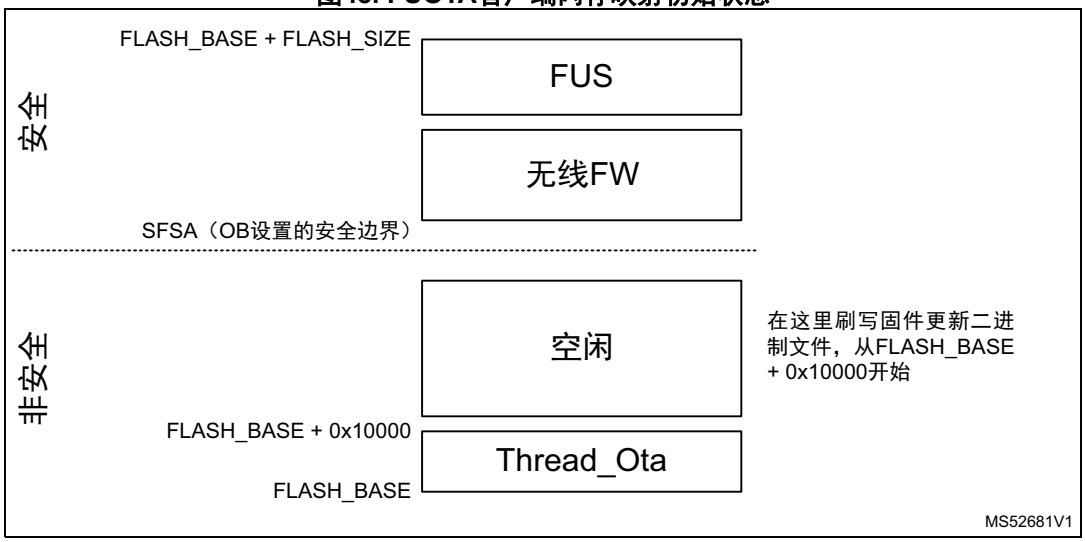
图47. OTA服务器（Thread_Ota_Server）闪存映射




### 客户端

在客户端，收到来自服务器的二进制数据前的闪存如*图 48*所示。

图48. FUOTA客户端闪存映射初始状态



在收到 来自服务器端的二进制数据后，闪存得到更新，如*图 49*和*图 50*（分别对应CPU1二进制数据传输和CPU2二进制数据传输）所示。

**图49. CPU1二进制数据传输后的FUOTA服务器闪存映射**

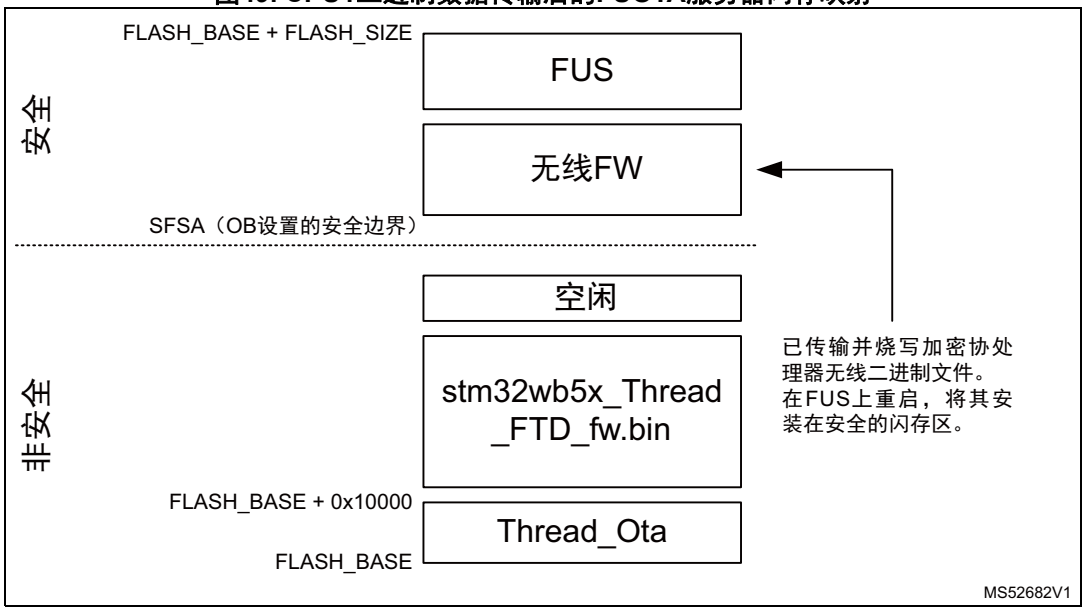
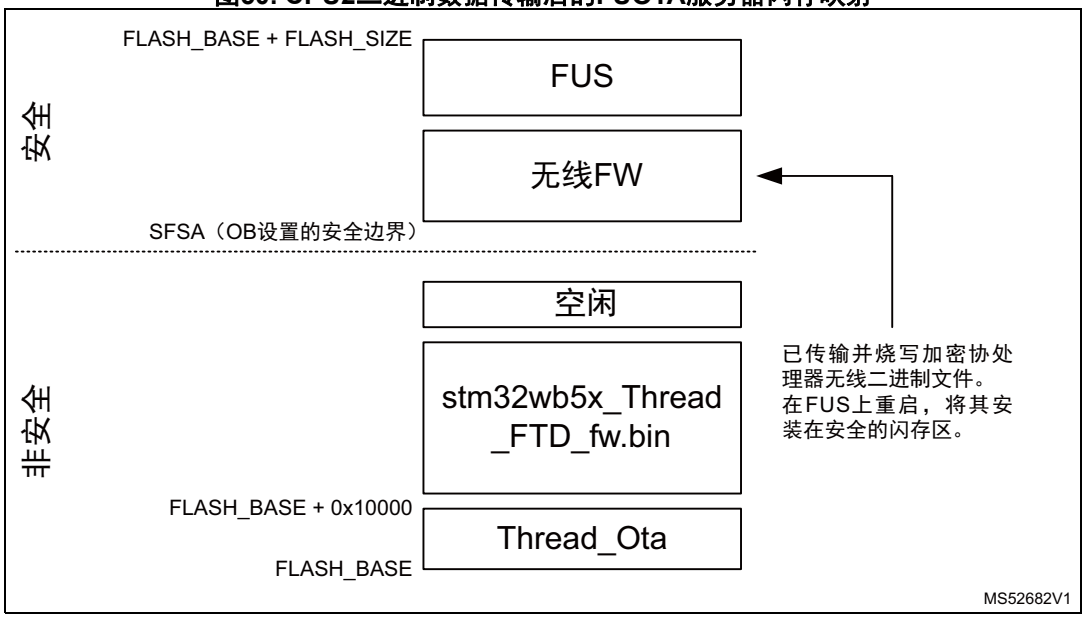



**图50. CPU2二进制数据传输后的FUOTA服务器闪存映射**

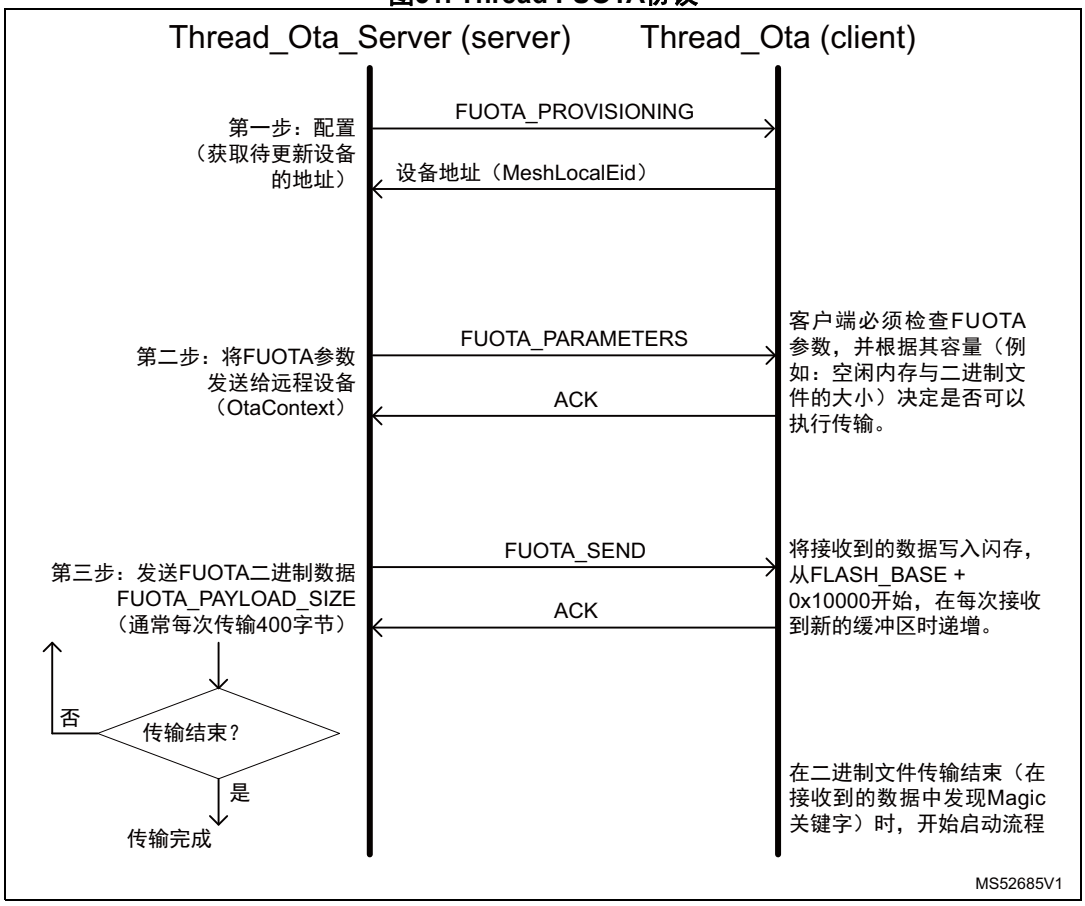


### 11.8.3 Thread FUOTA协议

这是意法半导体专有协议，可基于CoAP请求，使用Thread更新CPU2无线协处理器二进制文件或CPU1固件应用。

*图 51*详细描述了执行固件更新传输的步骤。

**图51. Thread FUOTA协议**




1. 服务器发送信息，以记录处理FUOTA的远程设备的地址。
   服务器对资源发送多播、不可确认的Get CoAP请求：“FUOTA_PROVISIONING”
   远程设备回复以Mesh Local Eid（端点标识符），它标识了Thread接口（与网络拓扑无关）。
2. 将OtaContext数据结构发送给远程设备。它包含：
   - 文件类型：FW_APP更新或FW_COPRO_WIRELESS更新
   - 二进制数据大小：要传输的二进制数据大小（字节数）
   - 基址：远程设备将二进制数据复制到闪存中时的起始地址
   - Magic关键字：指定二进制数据末尾的关键字
3. 将二进制数据传输到远程设备。
   通过大小为FUOTA_PAYLOAD_SIZE（默认为400字节）的缓冲区执行传输。可在服务器端进行配置。
   每次传输时，Thread_Ota_Server等到 远程设备确认后，再继续下一次缓冲区传输。

在远程端，接收到的每个数据缓冲区均被写入闪存。

当找到Magic关键字时，表示这是要发送的最后一个缓冲区。

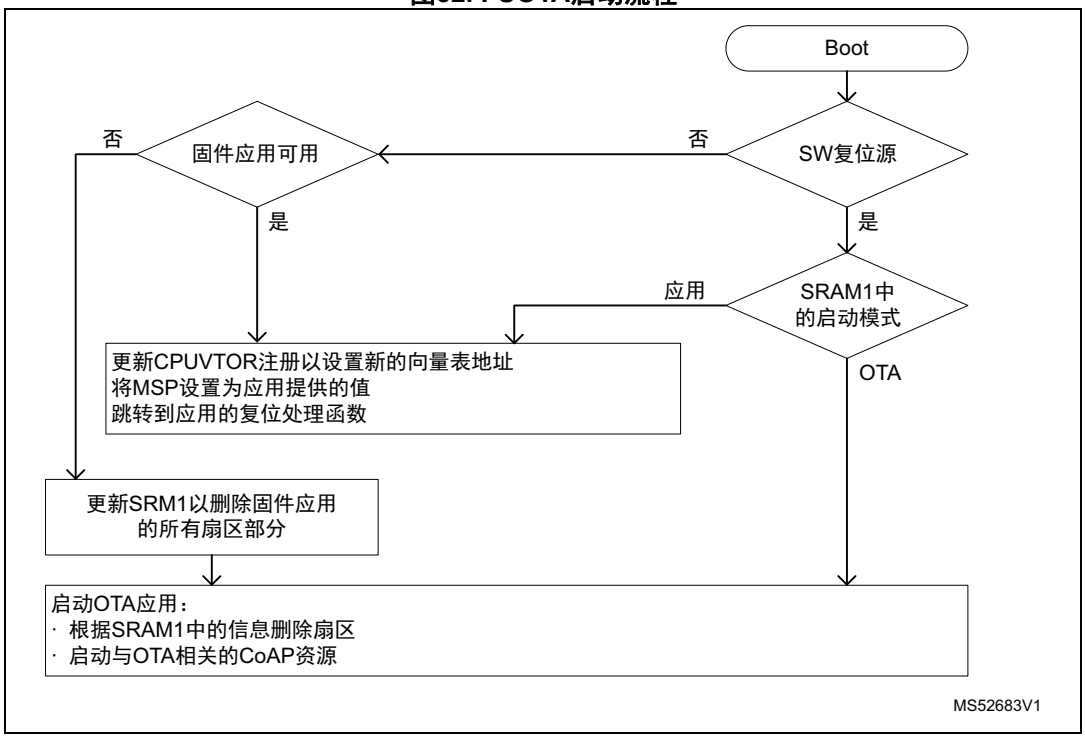
### 11.8.4 FUOTA应用启动流程

在将二进制数据传输到远程设备（Thread FUOTA客户端）后，CPU1应用与CPU2协处理器无线二进制数据的更新启动流程并不相同。

#### CPU1的FUOTA

在客户端，在完成二进制数据传输后，将发生如*图 52*所示的过程，从而跳转到OTA特定的应用（例如：Thread_Coap_Generic_Ota）：

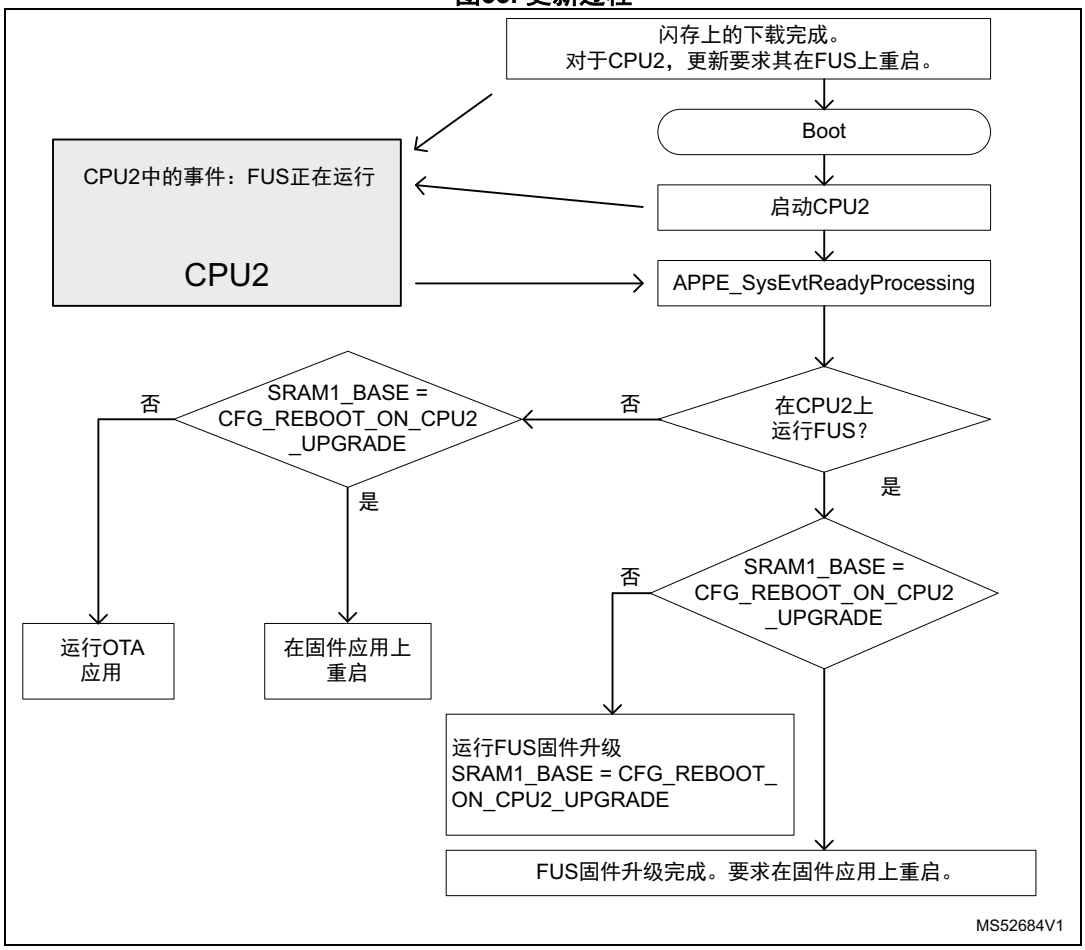
**图52. FUOTA启动流程**




#### CPU2的FUOTA

CPU2更新涉及FUS（固件升级服务）软件组件，该组件负责解密和安装安全的二进制文件。

图53. 更新过程




### 11.8.5 应用

#### Thread_Ota_Server

必须将该应用加载到充当FUOTA服务器的STM32WB 1Nucleo板上。

#### Thread_Ota

必须将该应用加载到充当FUOTA客户端的STM32WB Nucleo板上。

### Thread_Coap_Generic_Ota

该应用与Thread_Coap_Generic几乎相同，区别在于：

- 使用特殊标签（用于管理数据传输结束和数据一致性）：
  - TAG_OTA_END：在thread_ota应用中检查Magic关键字值
  - TAG_OTA_START：应在二进制映像开头的0x140处映射Magic关键字地址

  因此，在0x140处读取的存储内容等于Magic关键字值。
- 必须更新分散加载描述文件以插入上述存储区

  IAR的示例：

  向量表和ROM起始地址移至0x08010000：

```
define symbol __ICFEDIT_intvec_start__ = 0x08010000;
define symbol __ICFEDIT_region_ROM_start__ = 0x08010000;
define region OTA_TAG_region   = mem:[from (__ICFEDIT_region_ROM_start__ + 0x140)   to (__ICFEDIT_region_ROM_start__ + 0x140 + 4)];
keep { section TAG_OTA_START};
keep { section TAG_OTA_END };
place in OTA_TAG_region   { section TAG_OTA_START };
place in ROM_region   { readonly, last section TAG_OTA_END };
```

# 12 MAC IEEE Std 802.15.4-2011

## 12.1 概述

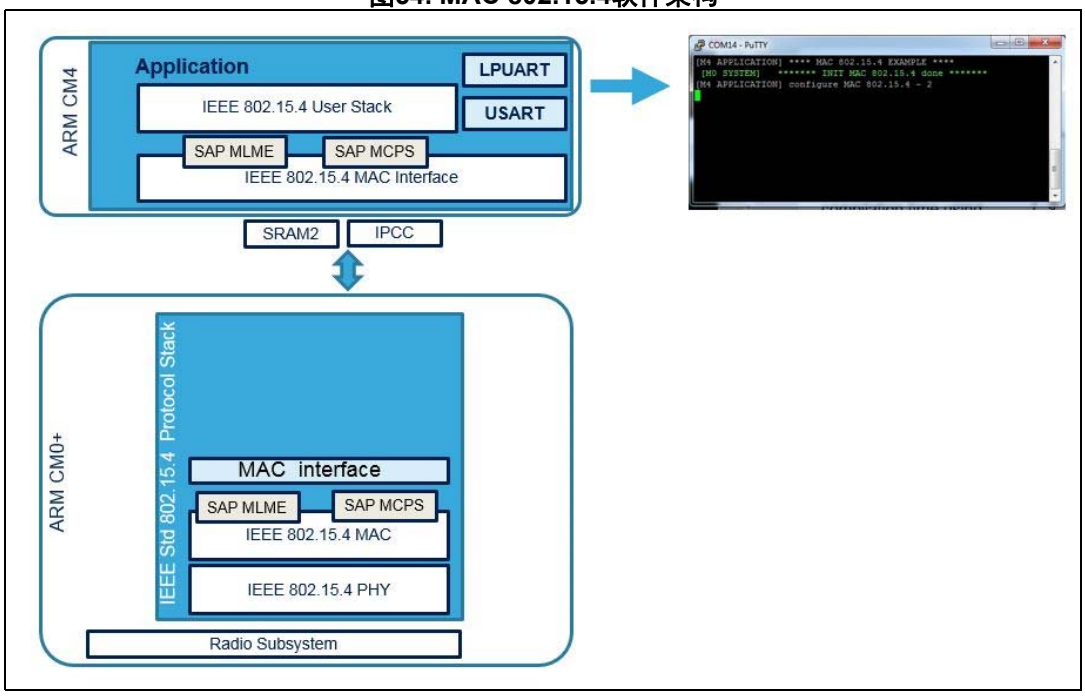
MAC IEEE Std 802.15.4-2011 层运行在 CPU2 内核（无线协议处理器）的 MAC 专用固件上。MAC 层依赖于处理 RF 子系统组件的 PHY 层。

由于此实现以二进制格式提供并在CPU2上运行，因此MACAPI暴露给CPU1内核让用户寻址MAC服务接入点。然后，用户可以将自己的STM32WB设备设置为FFD（全功能设备，即协调器）或 RFD（精简功能设备，即节点），如 IEEE Std 802.15.4-2011 规范文档中所述。

## 12.2 架构

*图 54*显示了当客户希望通过在应用处理器上集成定制解决方案或第三方解决方案来实施内部 802.15.4 网络时使用的 MAC 软件架构。

**图54. MAC 802.15.4软件架构**

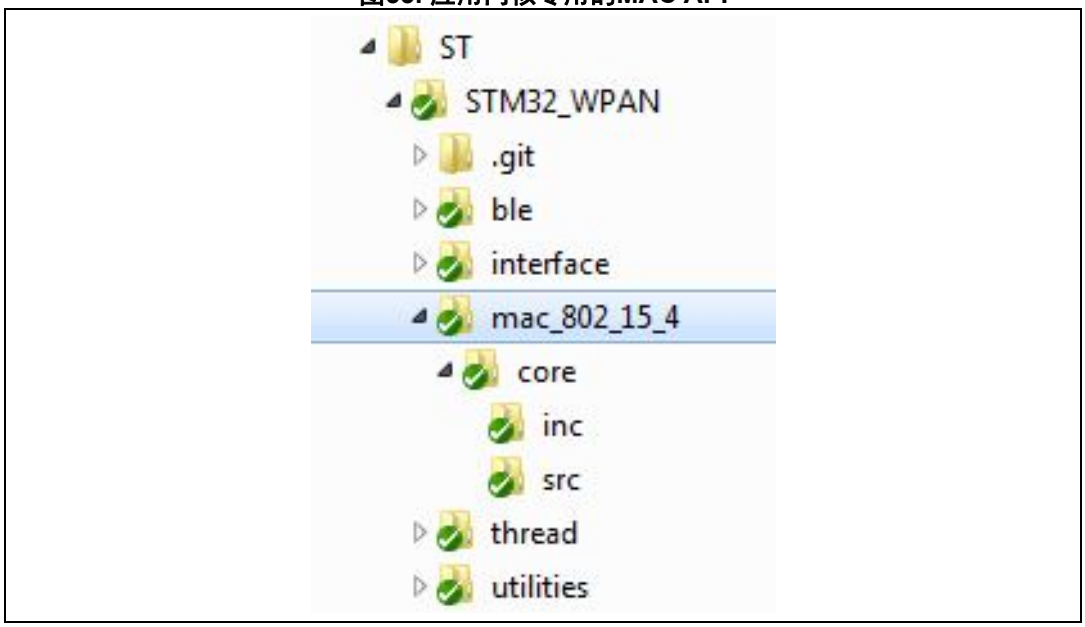



## 12.3 API

MAC IEEE Std 802.15.4-2011规范文档定义了802.15.4网络层与介质访问控制层之间的接口。此API允许用户寻址名为MLME（MAC子层管理实体）的MAC管理实体服务，就像寻址名为MCPS（MAC公共部分子层实体）的MAC数据服务一样。

应用内核专用的MAC API及其关联实现可以从中间件的目录\Middlewares\ST\STM32_WPAN\mac_802_15_4（参见*图 55*）下获取。

图55. 应用内核专用的MAC API




此实现记录在STM32WB固件包中目录Firmware\Middlewares\ST\STM32_WPAN\mac_802_15_4下的文件STM32WBxx_MAC_802_15_4_User_Manual.chm中。具体的基元说明见IEEE Std 802.15.4-2011文档。

## 12.4 如何开始

### 12.4.1 板配置

确保选项字节设置如*图 56*所示。

图56. MAC 802.15.4的选项字节配置

### 12.4.2 MAC无线协议处理器CPU2固件

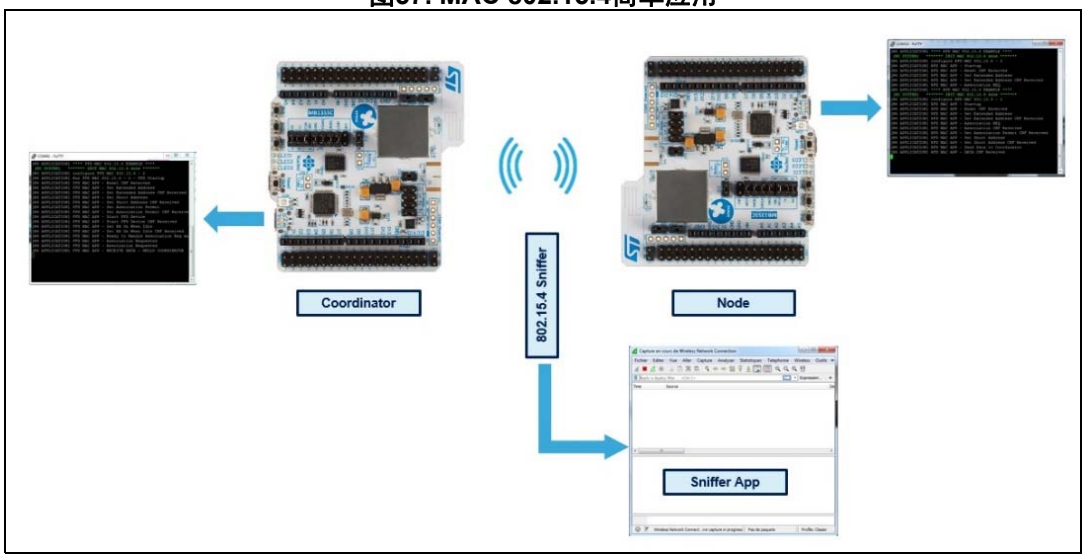
用户首先需要为CPU2无线协议内核下载合适的专用MAC固件二进制文件，参见STM32WB固件包中目录Firmware\Projects\STM32WB_Copro_Wireless_Binaries下的文件Release_Notes.html。

### 12.4.3 MAC应用处理器固件

在实现自定义栈解决方案或集成为CPU1应用内核MACAPI提供的第三方栈之前，用户可通过MAC应用示例加快实现速度，该示例涉及下面两个必须在两块STM32WB板上同步运行的应用：

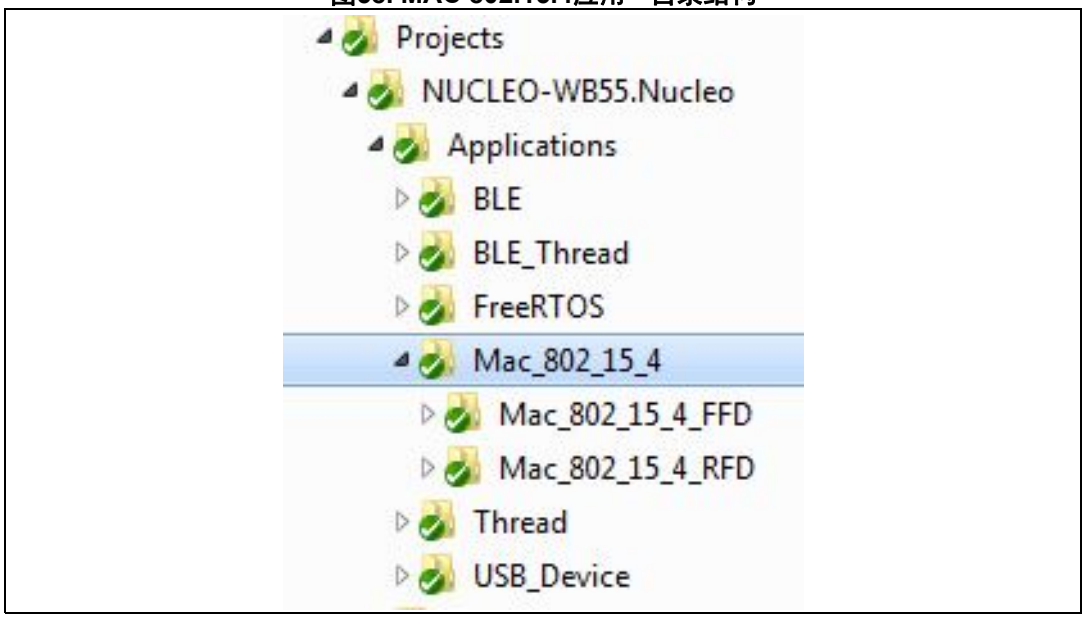
- Mac_802_15_4_FFD：展示如何实现简单的802.15.4协调器。此设备按关联请求管理网络，并根据节点需求获取或提供数据。
- Mac_802_15_4_RFD：展示如何实现简单的802.15.4节点。此设备向协调器发送关联请求。在被寻址的协调器对请求作出肯定响应后，节点接收其新的短地址，然后向协调器发送数据。

**图57. MAC 802.15.4简单应用**




专用于Nucleo STM32WB板的两个应用都可以从NUCLEO-WBxx.Nucleo应用目录Mac_802_15_4下获取（参见*图 58*）。

图58. MAC 802.15.4应用 - 目录结构




readme.txt文件描述了每个802.15.4设备处理的MAC序列。该文件可以从每个根项目获取。

### 12.4.4 输出

用户可以在正确的通道上使用 OTA 嗅探器，在关联阶段监听两块板之间的协商，并在节点注册到由协调器管理的网络中后检测数据交换。

由UART跟踪应用。然后用户可以使用终端仿真器在每个已实现的虚拟COM端口上启动HyperTerminal会话，以便检查每个MAC步骤。

连接控制台的TTY会话配置：

- 波特率：115200
- 数据位：8
- 停止位：1
- 校验位：无
- 流量控制：XON/XOFF。

运行两个应用会显示如图*59*至*61*所示的超级终端。

图59. 协调器启动

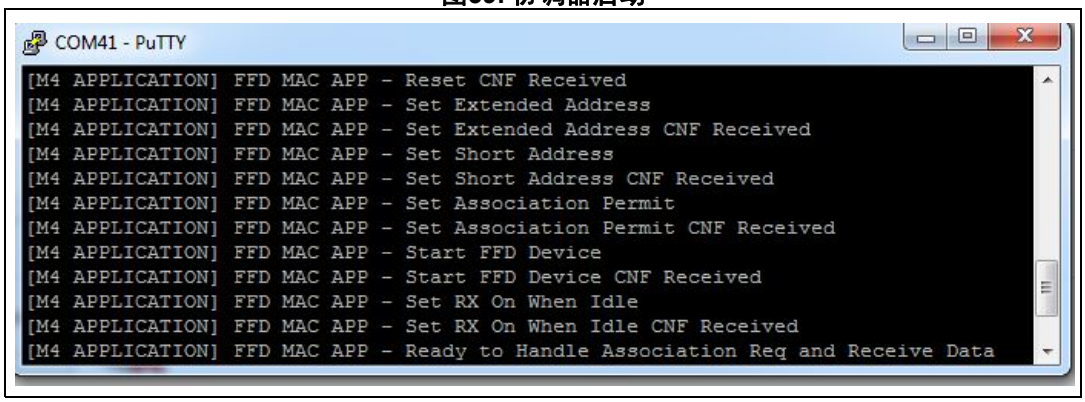



图60. 节点启动、请求关联和数据发送

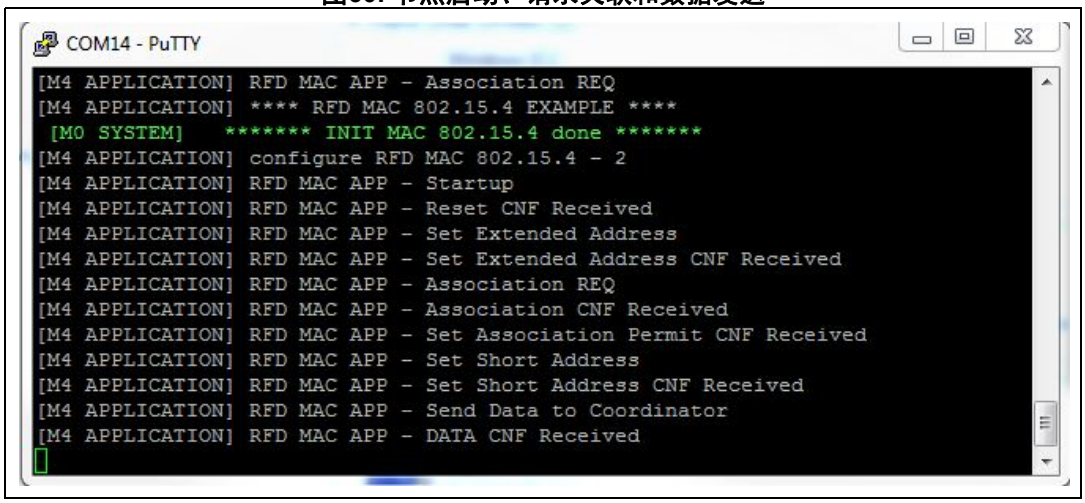
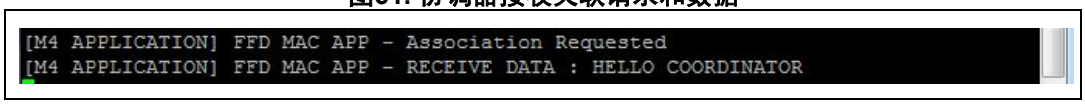



图61. 协调器接收关联请求和数据




### 12.4.5 MAC IEEE Std 802.15.4-2011 system

这是目前已实现的MAC系统命令。

SHCI_C2_MAC_802_15_4_Init()启动无线处理器（CPU2）上的MAC层和RF子系统。

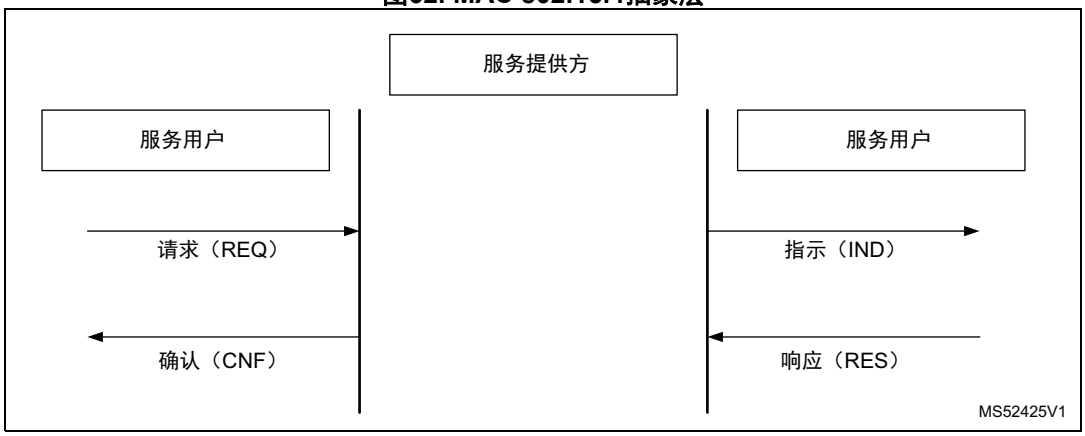
MAC层无法保证非易失性数据的安全。确保这些数据保存在闪存中和恢复以后要使用的数据都取决于上层应用。

不支持低功耗特性。

### 12.4.6 集成建议

MAC层通过实现一个抽象层来提供服务基元。 这个抽象层，在MAC IEEE Std 802.15.4-2011规范文档中描述，如*图 62*所示。

图62. MAC 802.15.4抽象层




给出的API允许用户调用REQ和RES基元，其关联的指定结构已从上层初始化。 为了从MAC层获得通知，必须实现名为MAC指示（IND）或MAC确认（CNF）的自定义调用函数。

### 请求和响应示例

- 设置当前设备的短地址
- 调用MAC_MLMESetReq，使用初始化的SetReq结构保存要设置的短地址。

```
// Set Device Short Address
  uint16_t shortAddr   = 0x1122;
  SetReq.PIB_attribute = g_MAC_SHORT_ADDRESS_c;
  SetReq.PIB_attribute_valuePtr =(uint8_t*) &shortAddr;
  MacStatus = MAC_MLMESetReq( &SetReq );
```

- 响应关联指示

在请求关联后，协调器可能通过向请求方提供短地址的方式做出响应。

- 使用存储属性短地址的初始化 AssociateRes 结构调用 MAC_MLMEAssociateRes。

```
APP_DBG("Srv task : Response to Association Indication");
  MAC_associateRes_t AssociateRes;
uint16_t shortAssociationAddr = 0x3344;
        memcpy(AssociateRes.a_device_address,g_MAC_associateInd.a_device_address,0x08);
memcpy(AssociateRes.a_assoc_short_address,&shortAssociationAddr,0x08);
AssociateRes.security_level = 0x00;
AssociateRes.status = MAC_SUCCESS;

MacStatus = MAC_MLMEAssociateRes(&AssociateRes);
```

- 确认和指示示例

为了从较低的MAC层获得确认或指示信息通知，用户必须在MAC_callbacks_t macCbConfig中注册自定义回调（app_ffd_mac_802_15_4.c中提供了示例）：

```
/* Mac Call Back Initialization */
macCbConfig.mlmeResetCnfCb = APP_MAC_mlmeResetCnfCb;
macCbConfig.mlmeScanCnfCb = APP_MAC_mlmeScanCnfCb;
macCbConfig.mlmeAssociateCnfCb = APP_MAC_mlmeAssociateCnfCb;
macCbConfig.mlmeAssociateIndCb = APP_MAC_mlmeAssociateIndCb;
....
```

- 对数据指示的操作

用户必须实现自定义回调以从MAC服务检索数据：

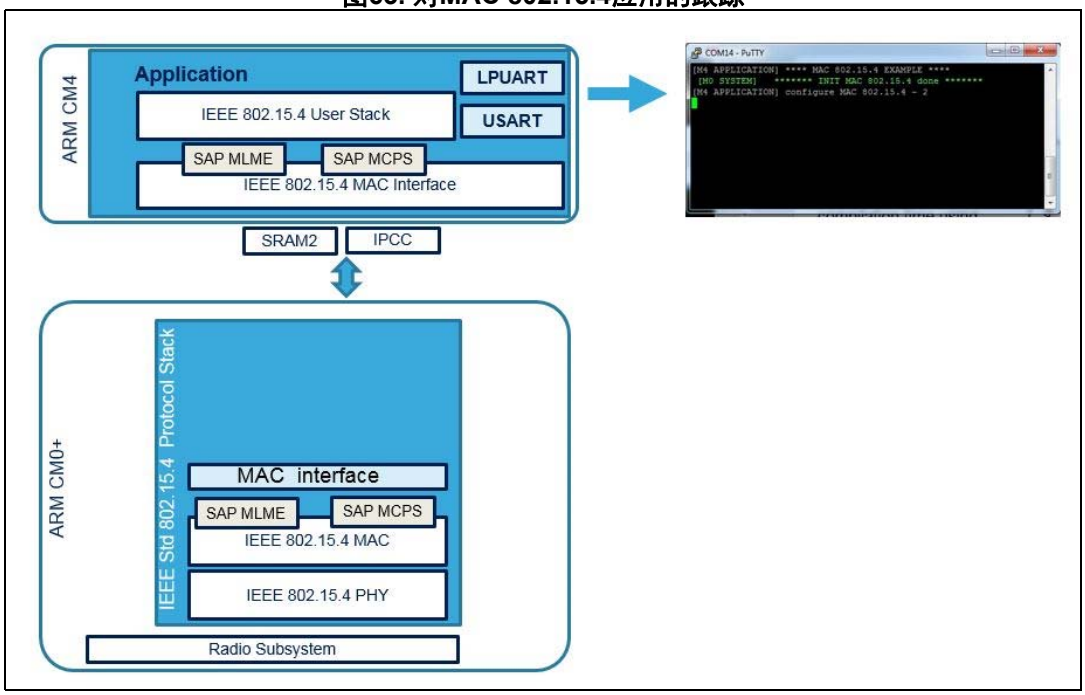
对于来自MAC层的数据指示信息，使用macCbConfig.mcpsDataIndCb调用APP_MAC_mcpsDataIndCb回调，可按以下方式实现该回调，以便检索MAC_dataInd_t结构（app_mac_802-15-4_process.c）承载的指示数据：

```
MAC_Status_t APP_MAC_mcpsDataIndCb( const  MAC_dataInd_t * pDataInd )
{

 memcpy(&g_DataInd,pDataInd,sizeof(MAC_dataInd_t));
 return MAC_SUCCESS;

}
```

**图63. 对MAC 802.15.4应用的跟踪**

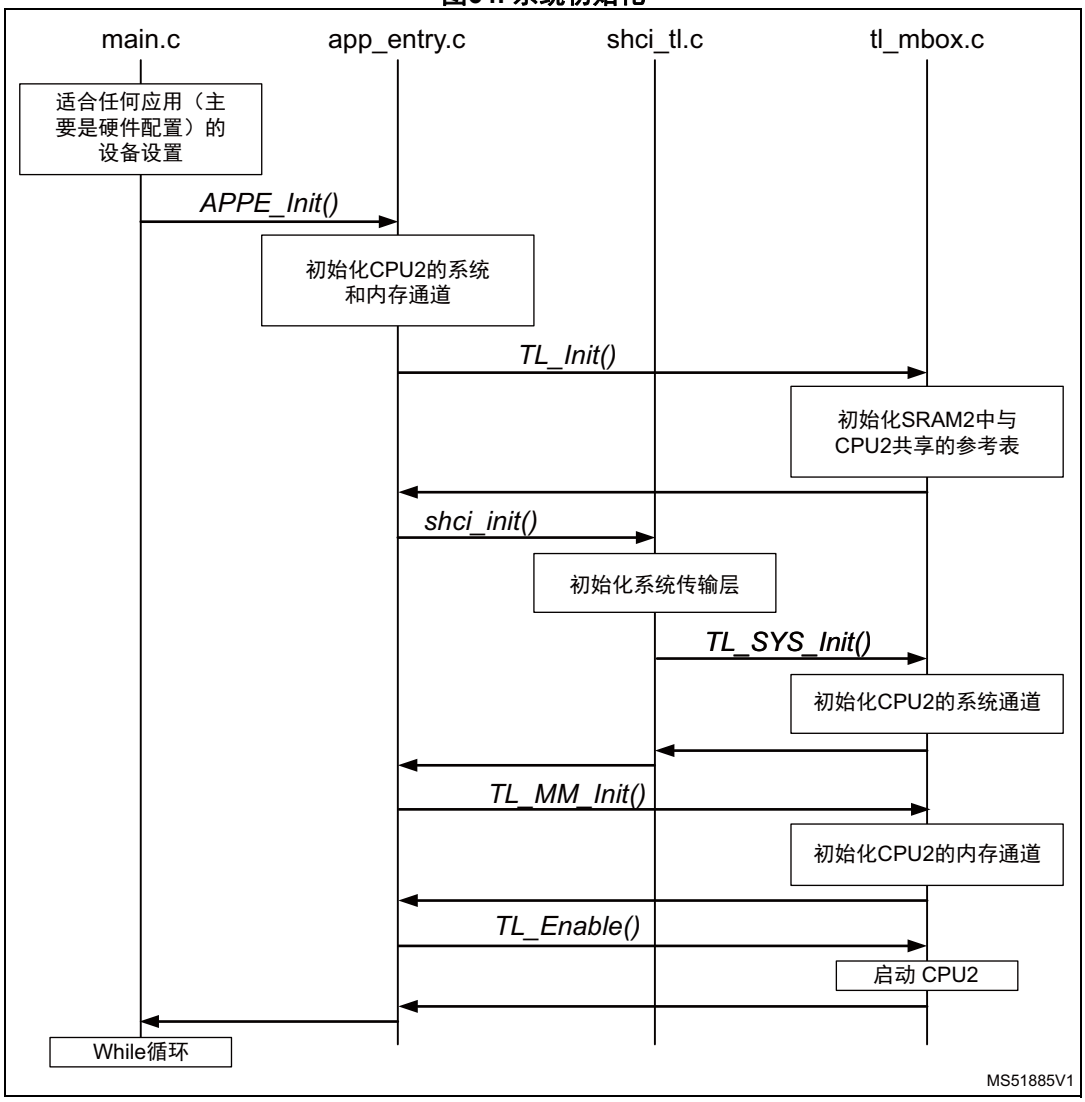


# 13 附录

## 13.1 设备初始化的具体流程

启动时，首先初始化设备，然后初始化通往CPU2的系统通道。此过程结束后，CPU1返回后台while循环，等待CPU2通知它已准备好接收系统命令。CPU1可能会运行其他与RF(CPU2)无关的应用程序初始化。 无论 CPU2 是运行完整 BLE 主机协议栈、仅 HCI 接口还是 OpenThread 协议栈，启动都是相同的。

**图64. 系统初始化**




当CPU2准备接收系统命令时，会向CPU1发送通知。在收到通知shci_notify_asynch_evt()时，用户必须调用shci_user_evt_proc()以允许系统传输层处理事件。通过APPE_SysUserEvtRx()通知用户应用收到系统事件。由于是在IPCC中断处理函数上下文中接收到shci_notify_asynch_evt()，信息会被传递到后台，以便从后台while循环（任何中

断上下文之外）调用shci_user_evt_proc()。无论CPU2运行全功能BLE主机协议栈、仅HCI接口还是OpenThread协议栈，该机制都一样。

**图65. 系统就绪事件通知**

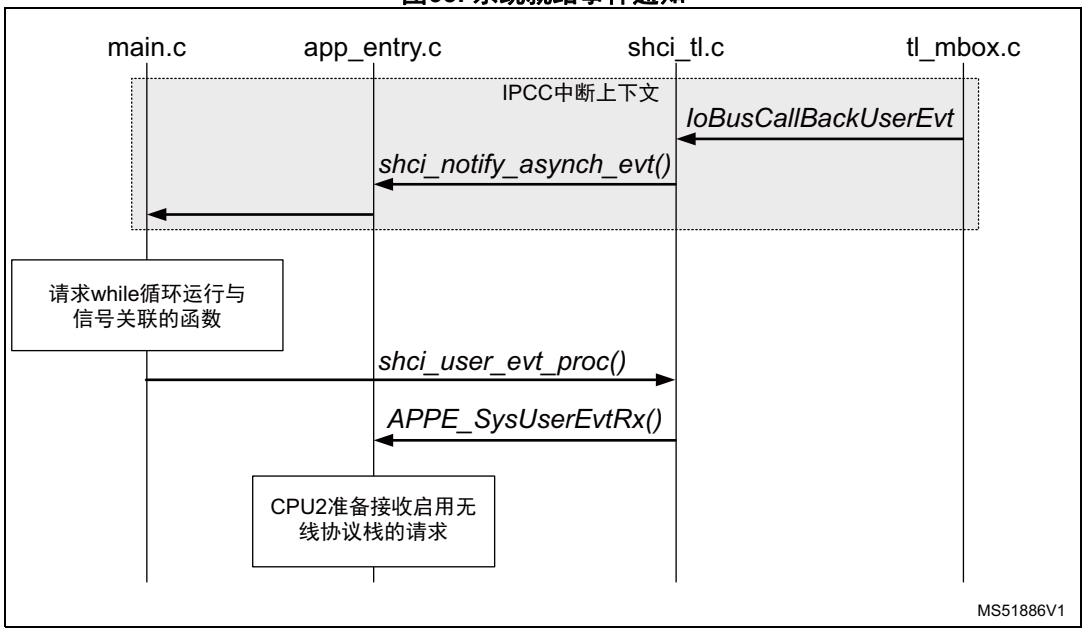
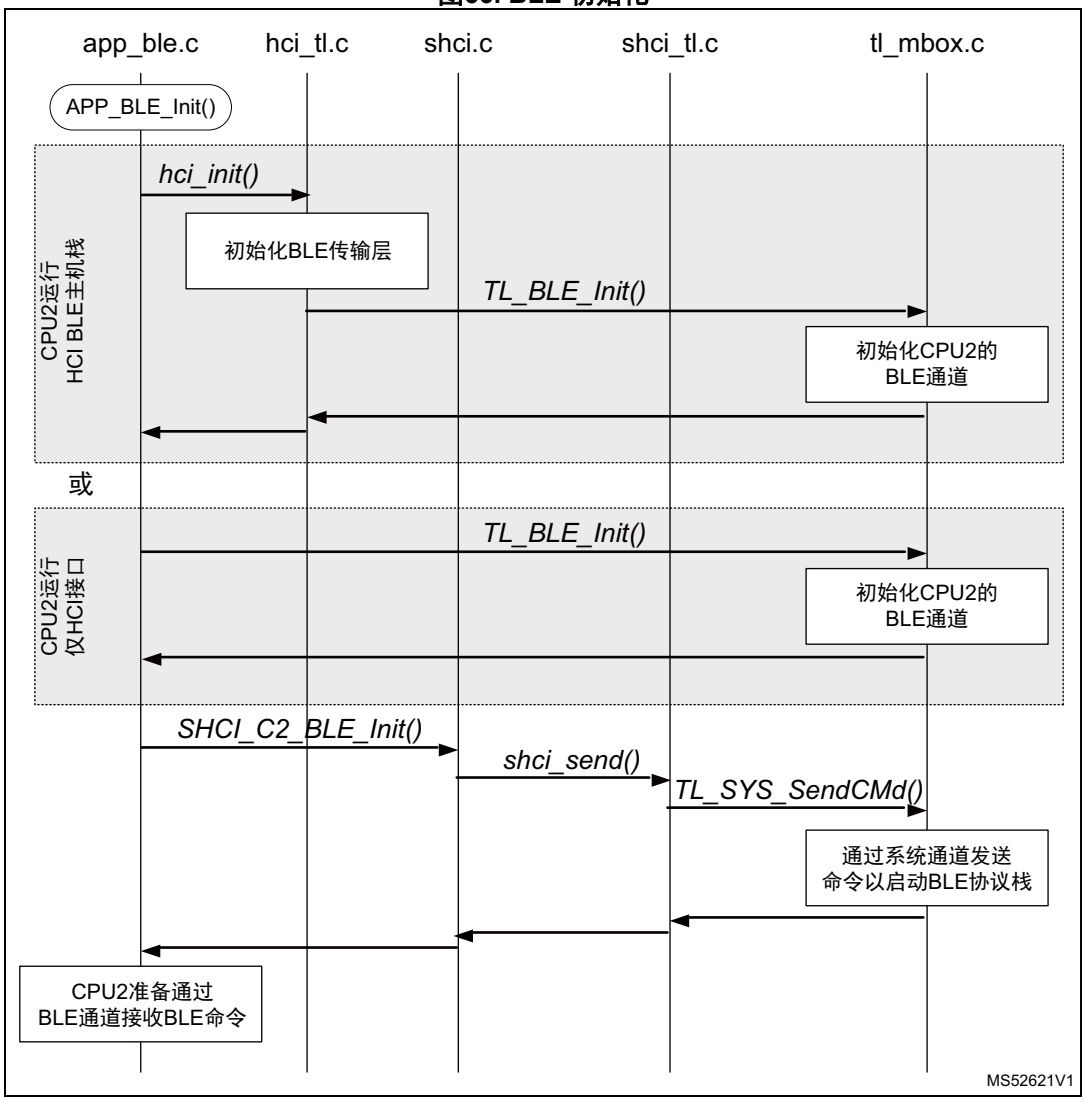


图66. BLE 初始化




在接收到系统事件时，初始化BLE传输层并向CPU2发送系统命令以启动BLE协议栈。在向CPU2发送SHCI_C2_BLE_Init()系统命令后，CPU准备接收BLE命令。

当CPU2仅运行HCI接口时，BLE传输层开始 在CPU1上的主机BLE协议栈中运行。因此，不得使用和初始化提供的BLE传输层。

## 13.2 信箱（Mailbox）接口

此接口是向BLE控制器发送命令时必须使用的最低级别接口。它用在透传模式应用中，并且当在BT SIG HCI接口之上使用BLE协议栈开源时必须使用。

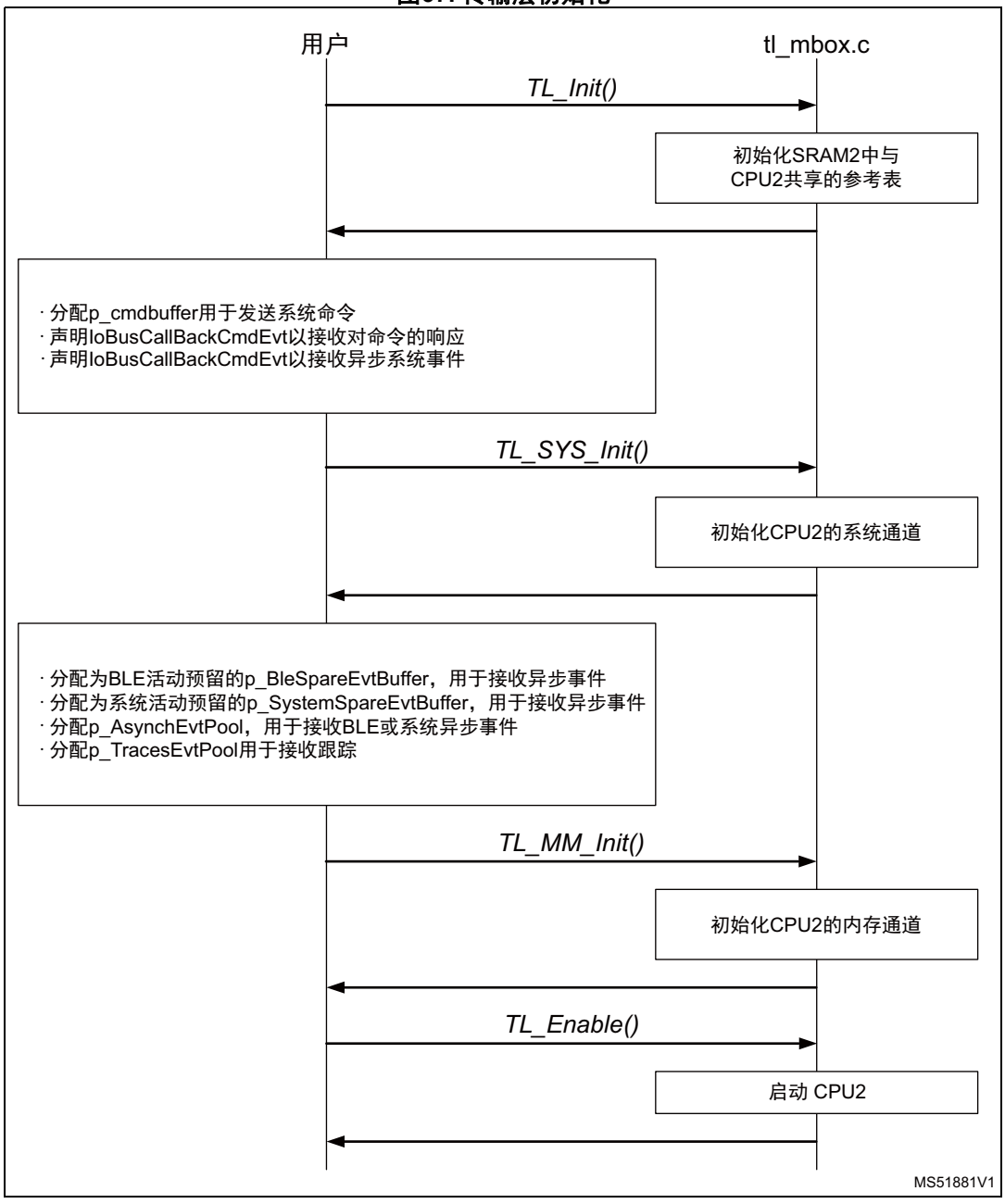
### 13.2.1 接口API

表30. 接口API

| 功能 | 说明 |
|---|---|
| void TL_Init( void ) | 初始化共享存储器 |
| void TL_Enable( void ) | 启用传输层 |
| int32_t TL_SYS_Init( void* pConf ) | 初始化系统通道 |
| int32_t TL_SYS_SendCmd( uint8_t* buffer, uint16_t size ) | 发送系统命令 |
| int32_t TL_BLE_Init( void* pConf ) | 初始化BLE通道 |
| int32_t TL_BLE_SendCmduint8_t* buffer, uint16_t size ) | 发送BLE指令 |
| int32_t TL_BLE_SendAclData( uint8_t* buffer, uint16_t size ) | 发送HCI ACL数据包 |
| void TL_MM_Init( TL_MM_Config_t *p_Config ) | 初始化内存通道 |
| void TL_MM_EvtDone( TL_EvtPacket_t * hcievt ) | 将缓冲区释放给内存通道 |

### 13.2.2 具体接口行为

图67. 传输层初始化




void TL_Init( void ):

这是要发送的第一个命令。它初始化信箱（Mailbox）驱动程序和共享存储器。

int32_t TL_SYS_Init( void* pConf ):

用户必须首先分配要被信箱（Mailbox）驱动程序用来发送系统命令（p_cmdbuffer）的缓冲区，以及要用来接收系统命令响应（IoBusCallBackCmdEvt）和系统异步事件（IoBusCallBackUserEvt）的两个回调函数。

IoBusCallBackCmdEvt 实现新要求，即只在接收到上一个系统命令的响应后才发送新的系统命令。

此命令初始化信箱（Mailbox）驱动程序中的系统通道。

void TL_MM_Init( TL_MM_Config_t *p_Config ):

用户必须首先分配要被信箱（Mailbox）驱动程序专门用来报告BLE异步事件（p_BleSpareEvtBuffer）的缓冲区、要被信箱（Mailbox）驱动程序专门用来报告系统异步事件（p_SystemSpareEvtBuffer）的缓冲区、要被BLE控制器用来报告BLE或系统异步事件的内存池（p_AsynchEvtPool）和要被CPU2用来进行报告跟踪的内存池（p_TracesEvtPool）。

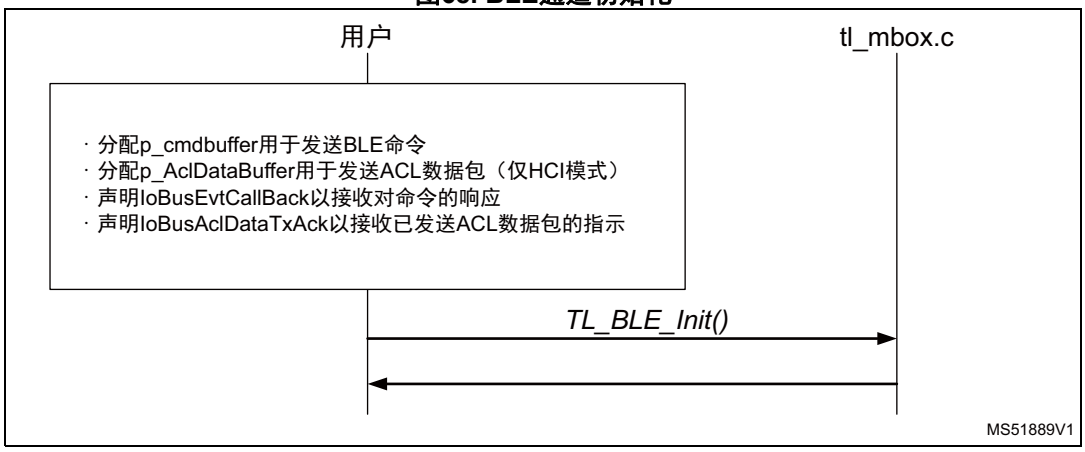
p_BleSpareEvtBuffer和p_SystemSpareEvtBuffer缓冲区用于保证即使在内存池p_AsynchEvtPool为空时，CPU2也始终能够报告BLE或系统事件。

此命令初始化信箱（Mailbox）驱动程序中的内存通道。

void TL_Enable( void ):

在完全初始化信箱（Mailbox）驱动程序后，发送此命令启动CPU2。

**图68. BLE通道初始化**




int32_t TL_BLE_Init( void* pConf ):

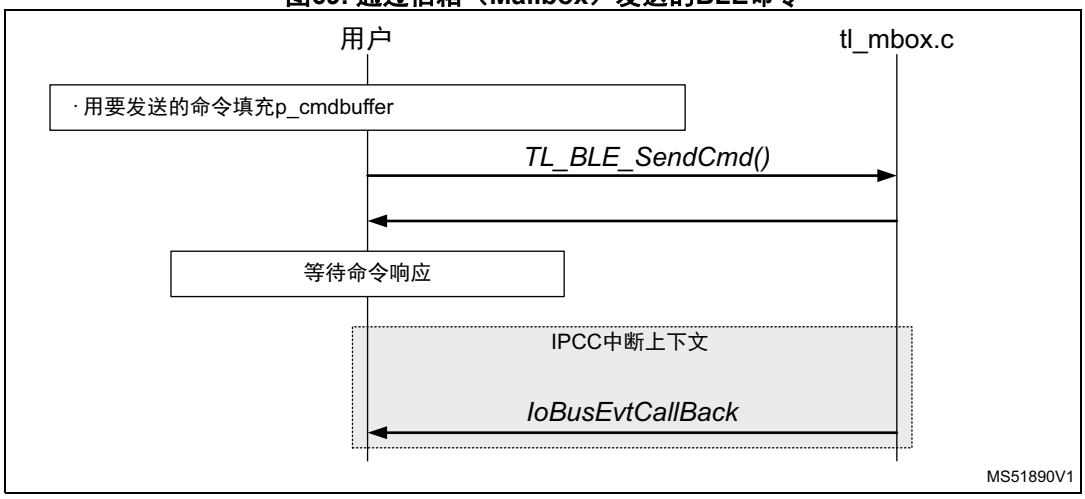
用户必须首先分配要被信箱（Mailbox）驱动程序用来发送BLE命令（p_cmdbuffer）的缓冲区、要被信箱（Mailbox）驱动程序用来发送ACL数据包（p_AclDataBuffer）的缓冲区以及要用来接收BLE事件（IoBusEvtCallBack）和ACL数据包确认（IoBusAclDataTxAck）的两个回调函数。

为了满足只能在命令流（依据BT SIG的规定）允许时发送新BLE命令的要求，必须使用IoBusEvtCallBack。

当不处于仅HCI模式时，不使用p_AclDataBuffer和IoBusAclDataTxAck且必须将它们置为0。

此命令初始化BLE控制器。

**图69. 通过信箱（Mailbox）发送的BLE命令**

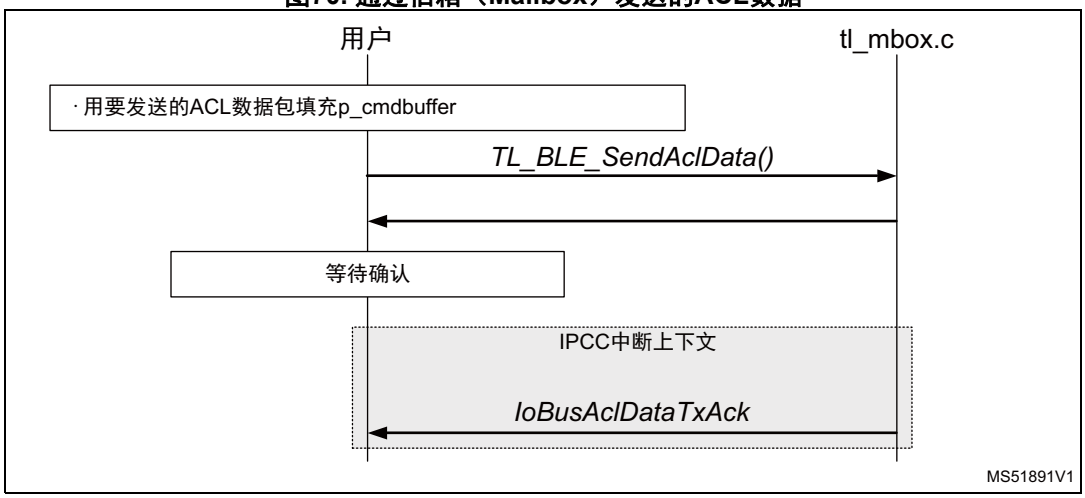



int32_t TL_BLE_SendCmd( uint8_t* buffer, uint16_t size ):

用户必须先用要发送的命令填充缓冲区p_cmdbuffer。不使用参数缓冲区和大小。

用户必须等待通过 IoBusEvtCallBack 接收到的命令响应来检查响应包中的流命令控制，以了解是否可以发送新命令。IoBusEvtCallBack 在 IPCC 中断上下文中异步生成。建议（在 IPCC 中断上下文之外）根据处理负载实施后台机制来解码接收到的数据包。

**图70. 通过信箱（Mailbox）发送的ACL数据**



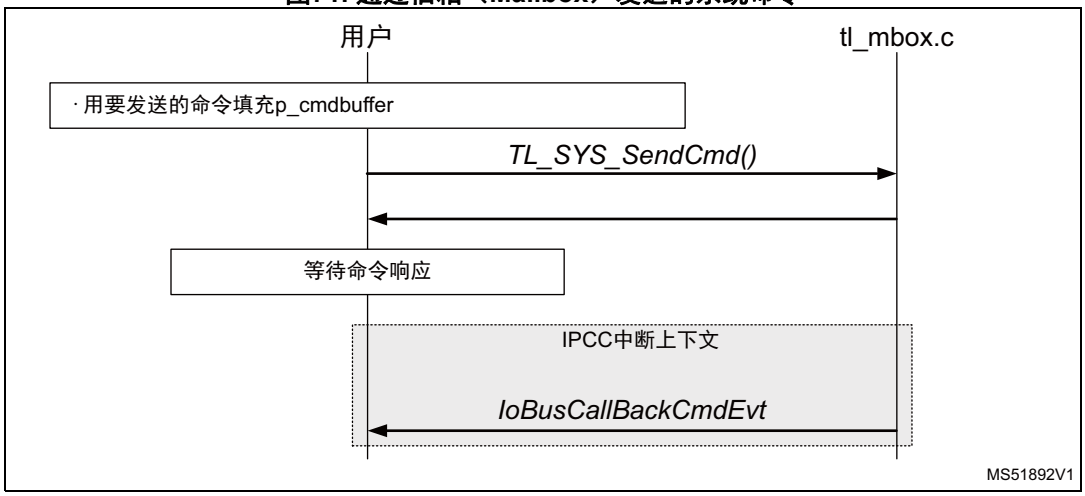

int32_t TL_BLE_SendAclData( uint8_t* buffer, uint16_t size ):

用户必须先用要发送的ACL数据包填充缓冲区p_AclDataBuffer。不使用缓冲区(buffer)和大小(size)参数。

用户只有在通过IoBusAclDataTxAck接收到确认后，才能发送新的ACL数据包。
IoBusAclDataTxAck 在 IPCC 中断上下文中异步生成。建议（在 IPCC 中断上下文之外）根据处理负载实施后台机制来处理确认。

仅HCI模式支持ACL数据包接口。如果支持，可以在BLE命令挂起时发送ACL数据包。BLE命令和ACL数据包不共享资源。

**图71. 通过信箱（Mailbox）发送的系统命令**

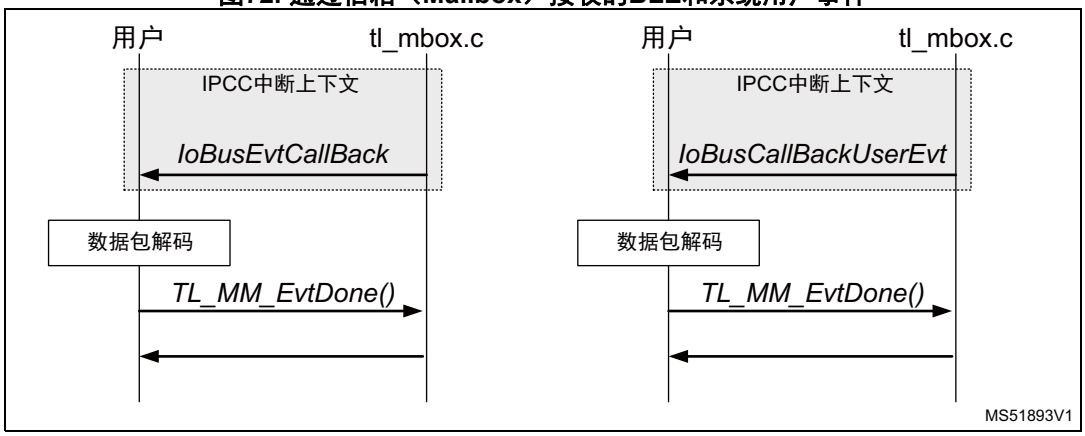



int32_t TL_SYS_SendCmd( uint8_t* buffer, uint16_t size )

用户必须先用要发送的命令填充缓冲区 p_cmdbuffer。不使用缓冲区(buffer)和大小(size)参数。

用户只有在通过IoBusCallBackCmdEvt接收到命令响应后，才能发送新的命令。IoBusCallBackCmdEvt在IPCC中断上下文中异步生成。建议（在IPCC中断上下文之外）根据处理负载实施后台机制来解码接收到的数据包。

**图72. 通过信箱（Mailbox）接收的BLE和系统用户事件**




void TL_MM_EvtDone( TL_EvtPacket_t * hcievt ):

对于以下情况，必须调用此API以将数据包返回到在CPU2上运行的内存管理器：

- 对于每个通过IoBusEvtCallBack（用户BLE事件回调）接收到的不是BLE命令响应的数据包。
- 对于每个通过IoBusCallBackUserEvt（用户系统事件回调）接收到的数据包。

## 13.3 信箱（Mailbox）接口 - 扩展

当要发送的命令被用户构建到要通过信箱（Mailbox）发送的缓冲区中时，适合使用信箱（Mailbox）接口。同样地，用户必须解码接收到的事件数据包，并管理命令流控制，以检查是否可以发送新命令。

当在HCI接口之上使用在CPU1上运行的BLE主机协议栈时，就属于这种情况。在这种情况下，只在HCI模式下使用CPU2。

但是，BLE主机协议栈不支持初始化CPU2所需的系统通道。因此，当只使用信箱（Mailbox）接口时，用户必须构建要发送到CPU2的系统命令包，并且必须管理从CPU2接收到的事件。

可以将简单的BLE信箱（Mailbox）接口与更高级别的SHCI接口混合使用，以便在连接到系统信箱（Mailbox）接口时编码/解码系统数据包。这就是“信箱（Mailbox）接口 - 扩展”的目的。

### 13.3.1 接口API

BLE和存储器接口与简单的信箱（Mailbox）接口相同。

为了使用更高级别的SHCI接口（来自文件shci.h），必须将SHCI传输层初始化并连接到信箱（Mailbox）驱动程序。

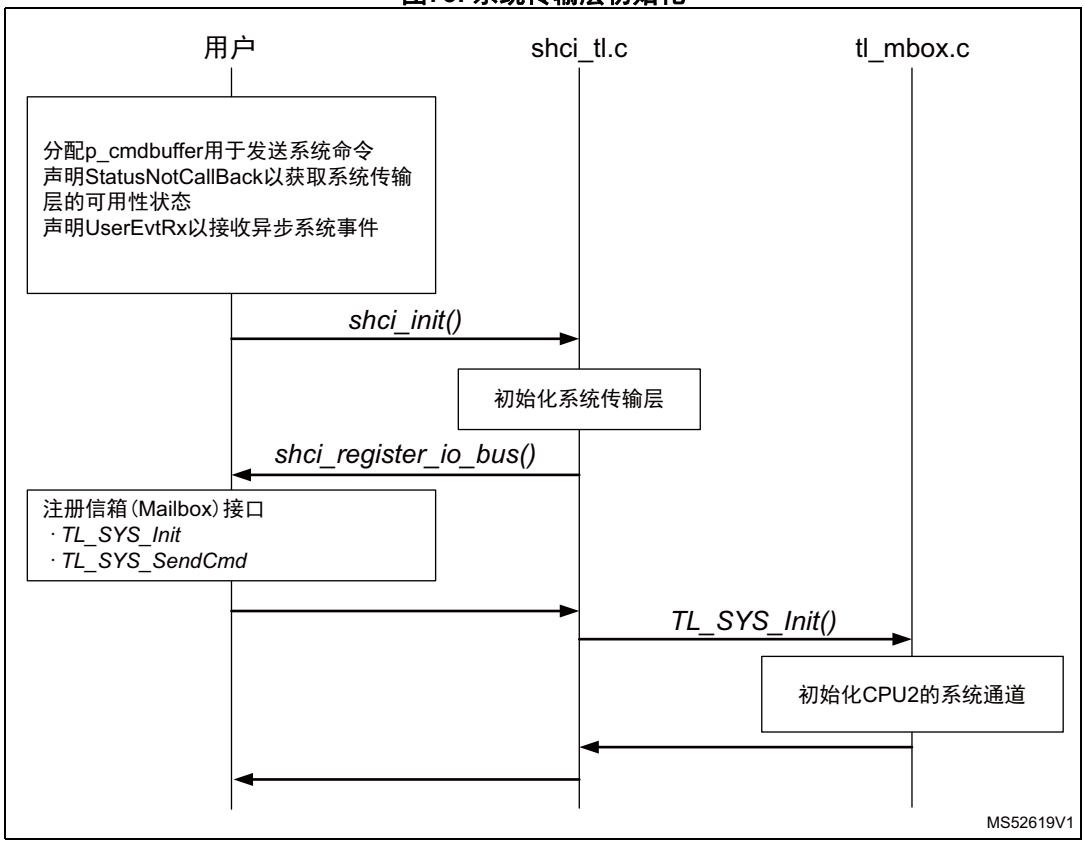
两个API TL_SYS_Init()和TL_SYS_SendCmd()以及两个回调IoBusCallBackCmdEvt和IoBusCallBackUserEvt均在传输层使用和实现， 不能再单独使用。

**表31. 接口API**

| 功能 | 说明 |
|---|---|
| void shci_init(void(* UserEvtRx)(void* pData), void* pConf) | 初始化系统传输层。 |
| void shci_register_io_bus(tSHciIO* fops) | 将信箱（Mailbox）接口注册到系统传输层。 |
| void tcp_echoserver_init(void) | 请求用户调用shci_user_evt_proc |
| void shci_resume_flow(void) | 继续被用户停止的异步用户事件报告。 |
| void shci_cmd_resp_wait(uint32_t timeout) | 等待命令响应。 |
| void shci_cmd_resp_release(uint32_t flag) | 发出已接收到命令响应的通知。 |
| void shci_user_evt_proc(void) | 处理接收到的异步用户事件并调用UserEvtRx。 |

### 13.3.2 具体接口与行为

图73. 系统传输层初始化




void shci_init(void(* UserEvtRx)(void* pData), void* pConf):

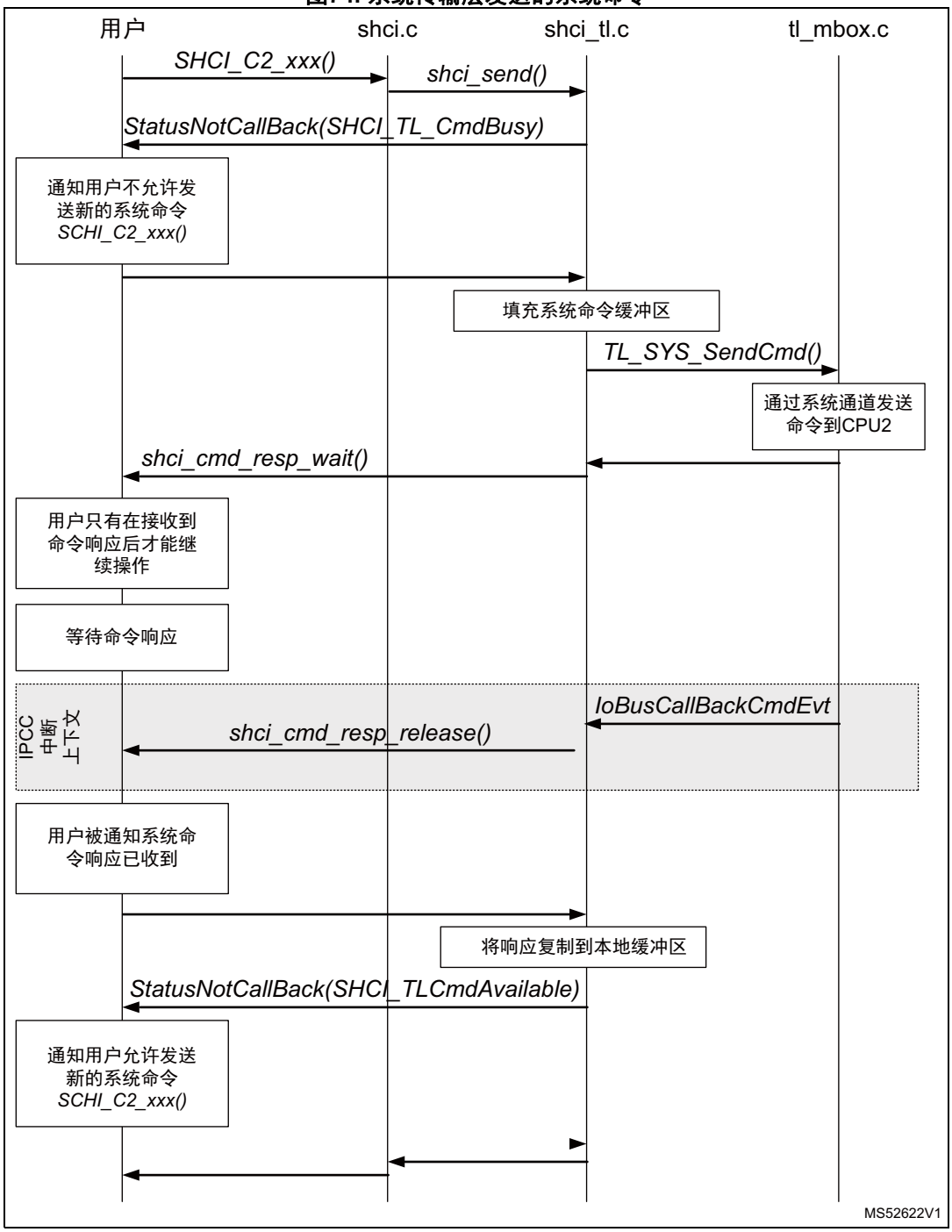
用户必须首先分配要被信箱（Mailbox）驱动程序用来发送系统命令（p_cmdbuffer）的缓冲区，以及要用来接收用户异步系统事件（UserEvtRx）和传输层可用性通知（StatusNotCallBack）的两个回调。

此命令初始化传输层和信箱（Mailbox）驱动程序中的系统通道。

void shci_register_io_bus(tSHciIO* fops);:

此命令将信箱（Mailbox）驱动程序注册到系统传输层。

图74. 系统传输层发送的系统命令




## SHCI_C2_xxx()

文件shci.h中提供了应用可使用的系统命令列表。

void StatusNotCallBack(SHCI_TL_CmdStatus_t status):

这是shci_init()中的已注册回调，用于确认是否可以发送系统命令。它必须用在可以从不同Thread发送系统命令的多Thread应用中。

当状态 = SHCI_TL_CmdBusy时，系统传输层处于忙碌状态，不发送新的系统命令。

void shci_cmd_resp_wait(uint32_t timeout):

应用只能在shci_cmd_resp_wait()中使用此命令，以通知已接收到响应。

参数没有意义。

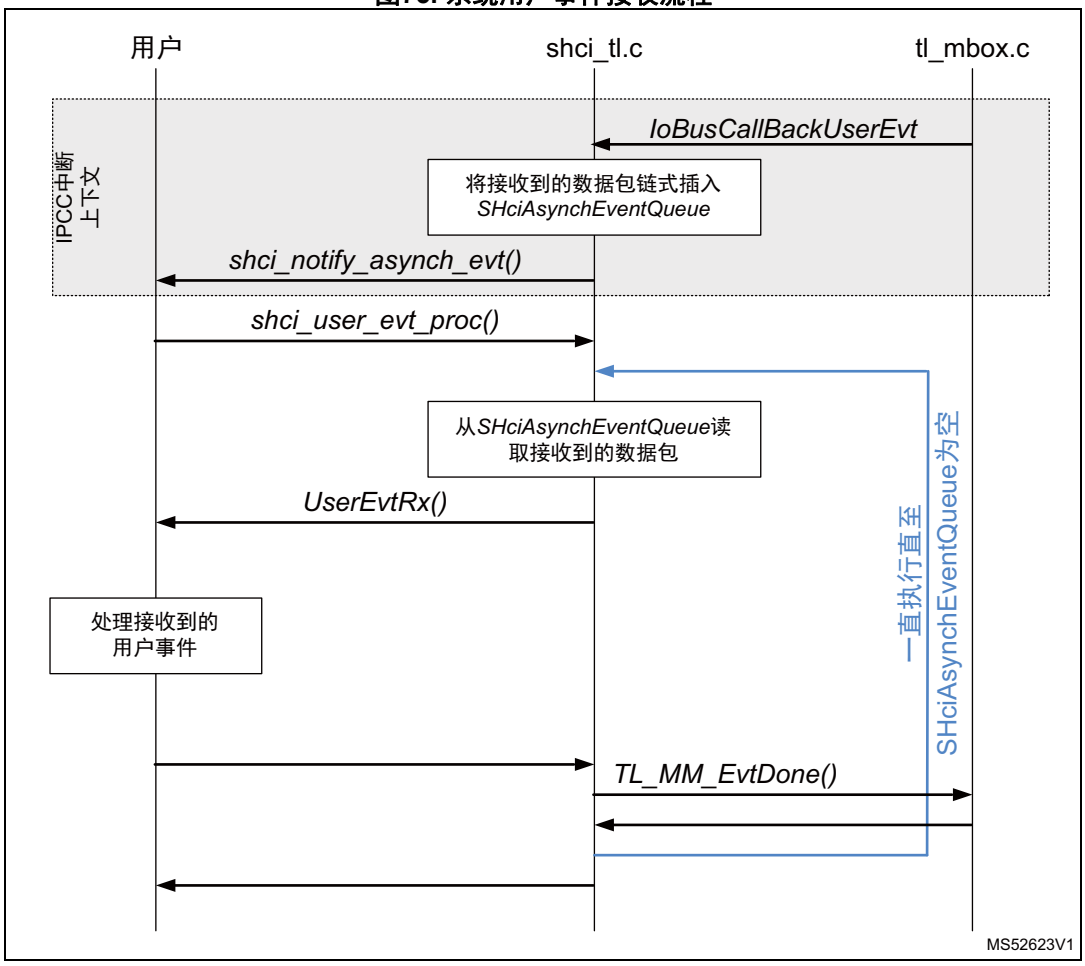
void shci_cmd_resp_wait(uint32_t timeout):

此函数通知用户已接收到挂起系统命令的响应。

它在IPCC中断上下文中调用。在退出该API时，应用可以从API shci_cmd_resp_wait()返回。

参数没有意义。

**图75. 系统用户事件接收流程**




void shci_notify_asynch_evt(void* pdata):

此API通知用户已接收到系统用户事件。用户必须调用shci_user_evt_proc()来处理系统传输层上的通知。由于shci_notify_asynch_evt()通知是从IPCC中断上下文调用的，强烈建议实现一种后台机制来调用shci_user_evt_proc()（在IPP中断上下文之外）。

pdata保存SHciAsynchEventQueue的地址。

void shci_user_evt_proc(void):

此函数通过UserEvtRx()将接收到的事件报告给用户。由于接收到的事件队列SHciAsynchEventQueue是在IPCC中断上下文中填充的，因此可以在用户处理事件时将新事件保存到队列中。为队列中的每个已接收事件调用UserEvtRx()。UserEvtRx()每次返回时，shci_user_evt_proc()处理将缓冲区释放给CPU2内存管理器。

void UserEvtRx (void * pData):

此函数向用户报告 接收到的系统事件。在此函数返回时，将释放保存了已接收事件的缓冲区。

pData是保存下列参数的结构地址：

typedef struct

{

SHCI_TL_UserEventFlowStatus_t status;

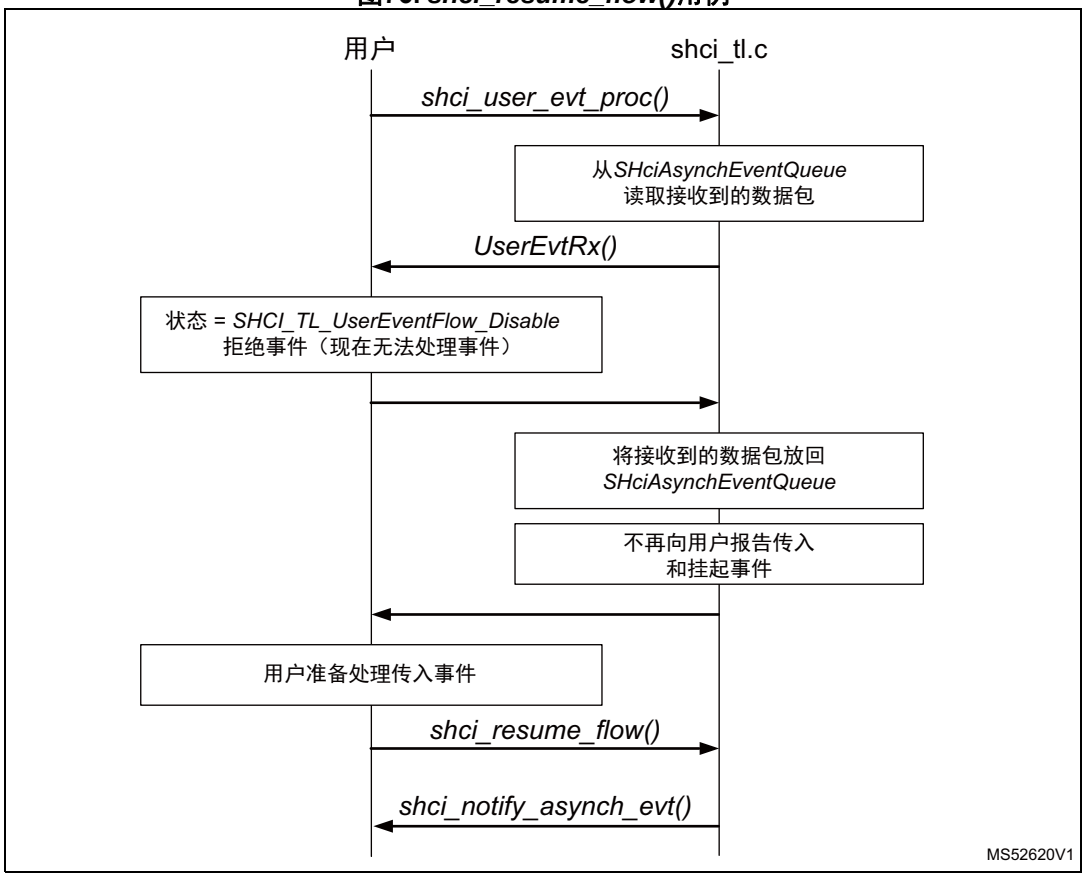
TL_EvtPacket_t *pckt;

} tSHCI_UserEvtRxParam;

pckt：保存了已接收事件的地址。

status：为用户提供了一种方法，用于将已接收数据包尚未处理且不得丢弃的情况通知系统传输层。如果UserEvtRx()返回时用户没有填充，此参数会被设置为SHCI_TL_UserEventFlow_Enable，表示用户处理了接收到的事件。

**图76. *shci_resume_flow()*用例**




void shci_resume_flow( void ):

当用户不能处理传入事件时，必须在从UserEvtRx()返回前将状态参数设置为SHCI_TL_UserEventFlow_Disable。在这种情况下，系统传输层不发布系统事件，并且不报告任何新的传入事件。

当用户准备处理系统事件时，必须发送shci_resume_flow()，以通知系统传输层重新开始报告系统事件。

## 13.4 ACI接口

此接口用于连接在CPU2上运行的BLE协议栈。它提供了全套API，可使用BLE层的全部功能（GATT、GAP和HCI LE）。

通过HCI传输层发送ACI命令。

用于访问所有BLE层（GATT、GAP）的接口位于文件夹\Middlewares\ST\STM32_WPAN\ble\core\Inc\core中。

在使用ACI接口时，BLE控制器必须设置为全协议栈模式。必须在应用中实现HCI传输层，以便在ACI接口与信箱（Mailbox）之间收发命令。

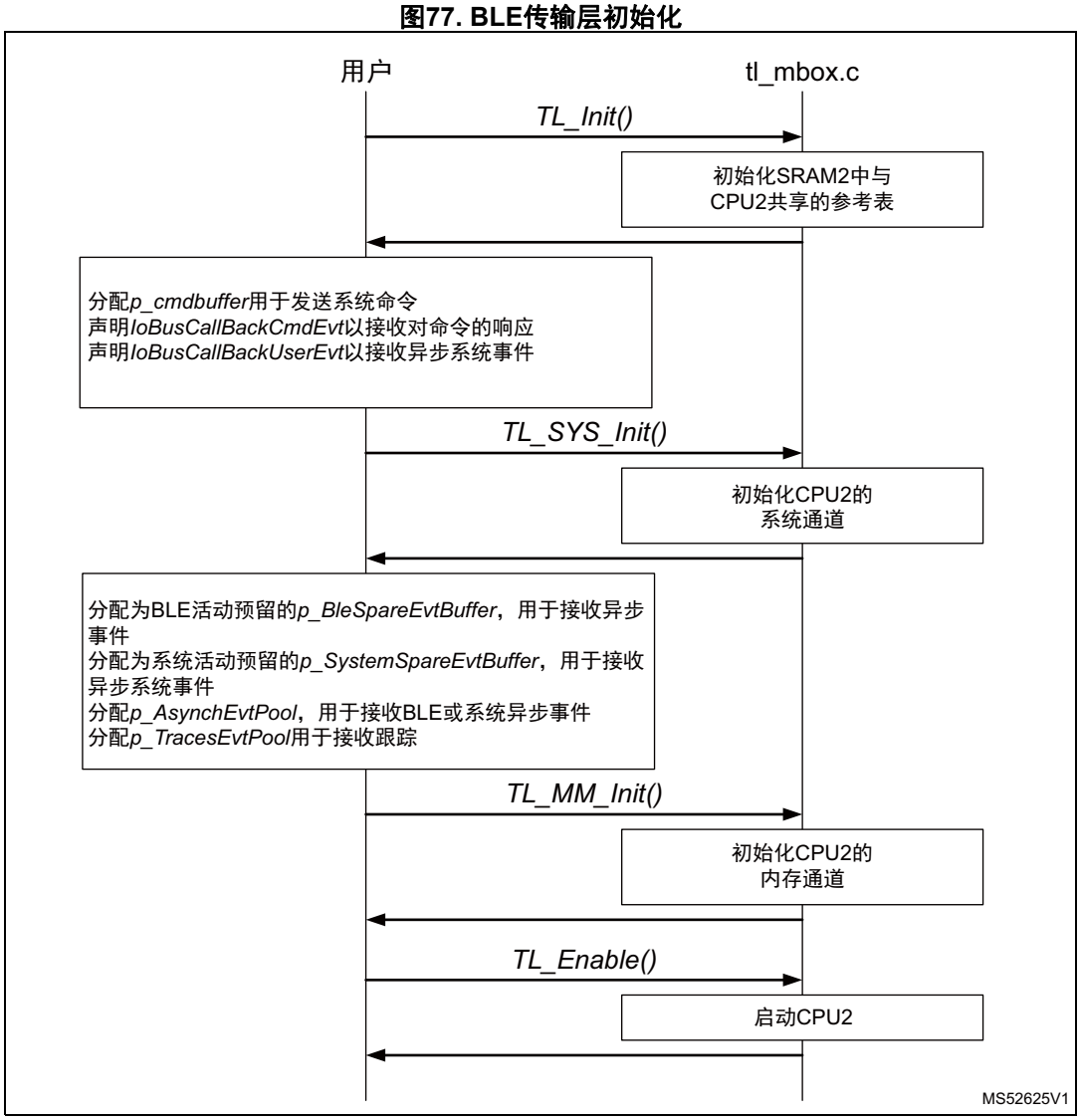
新接口基本上是信箱（Mailbox）接口 - 在实现 HCI 传输层的地方进行了扩展。使用 ACI 接口时，应用程序不再使用低级邮箱(Mailbox)接口。

**表32. BLE传输层接口**

| 功能 | 说明 |
|---|---|
| void hci_init(void(* UserEvtRx)(void* pData), void* pConf); | 初始化BLE传输层 |
| void hci_register_io_bus(tHciIO* fops); | 将信箱（Mailbox）接口注册到BLE传输层 |
| void hci_notify_asynch_evt(void* pdata); | 请求用户调用hci_user_evt_proc |
| void hci_resume_flow (void) | 继续被用户停止的异步用户事件报告 |
| void hci_cmd_resp_wait(uint32_t timeout) | 等待命令响应 |
| void hci_cmd_resp_release(uint32_t flag) | 发出已接收到命令响应的通知 |
| void hci_user_evt_proc(void | 处理接收到的异步用户事件并调用UserEvtRx |

### 13.4.1 具体接口与行为

图77. BLE传输层初始化




void hci_init(void(* UserEvtRx)(void* pData), void* pConf);:

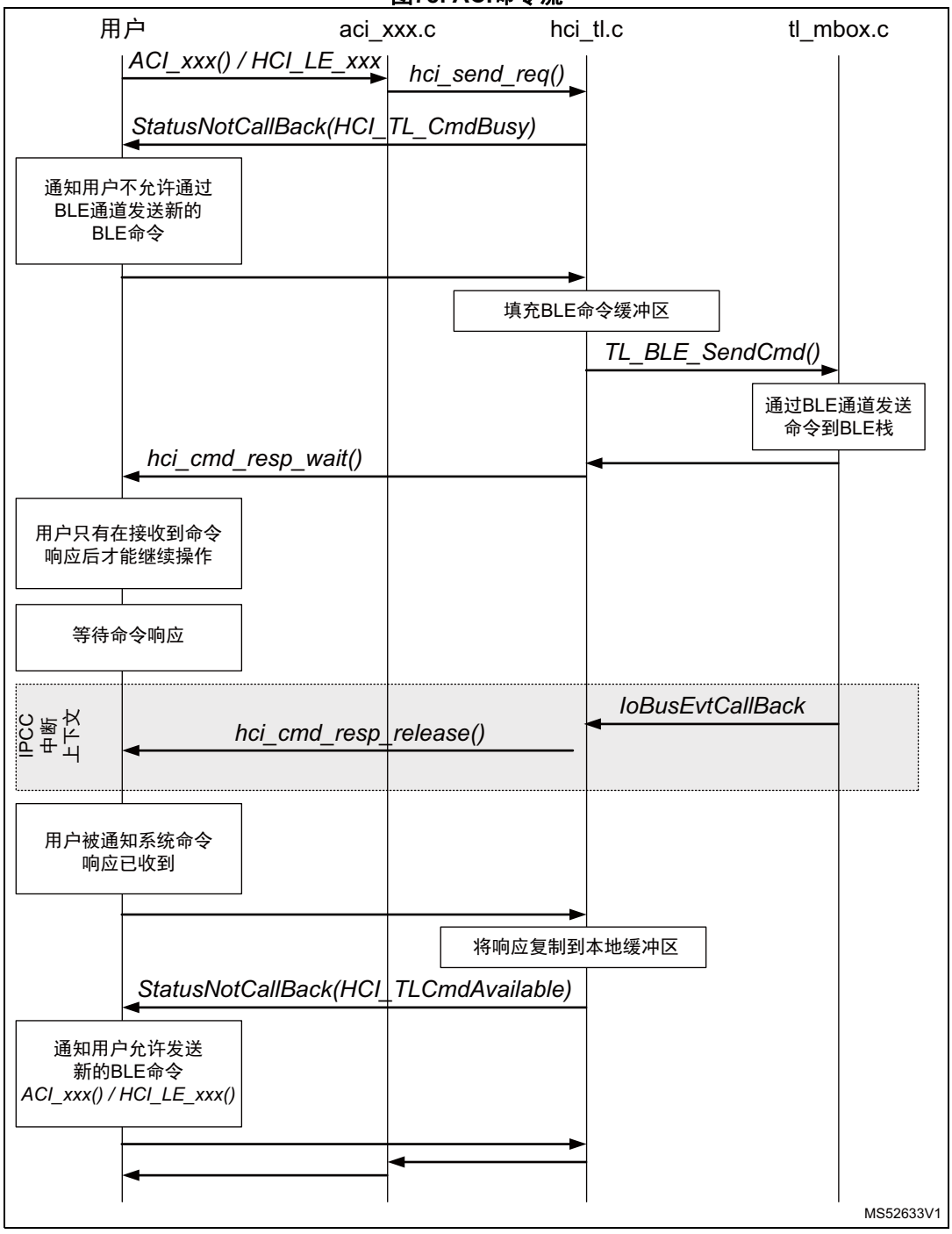
用户必须首先分配要被信箱（Mailbox）驱动程序用来发送BLE命令（p_cmdbuffer）的缓冲区，以及要用来接收用户异步系统事件（UserEvtRx）和传输层可用性通知（StatusNotCallBack）的两个回调。

此命令初始化HCI传输层和信箱（Mailbox）驱动程序中的BLE通道。

void hci_register_io_bus(tSHciIO* fops);:

此命令将信箱（Mailbox）驱动程序注册到HCI传输层。

图78. ACI命令流




## ACI_xxx() / HCI_LE_xxx()

文件夹\Middlewares\ST\STM32_WPAN\ble\core\Inc\core中提供了应用可使用的系统命令列表。

void StatusNotCallBack(HCI_TL_CmdStatus_t status):

这是hci_init()中的已注册回调，用于确认是否可以发送BLE命令。用于多线程应用程序，其中 BLE 命令可以从不同线程发送。

当状态 = HCI_TL_CmdBusy时，HCI传输层处于忙碌状态，不能发送新的BLE命令。

void hci_cmd_resp_wait(uint32_t timeout):

在通过调用hci_cmd_resp_wait()通知接收到响应前，应用不得从该命令返回。

参数没有意义。

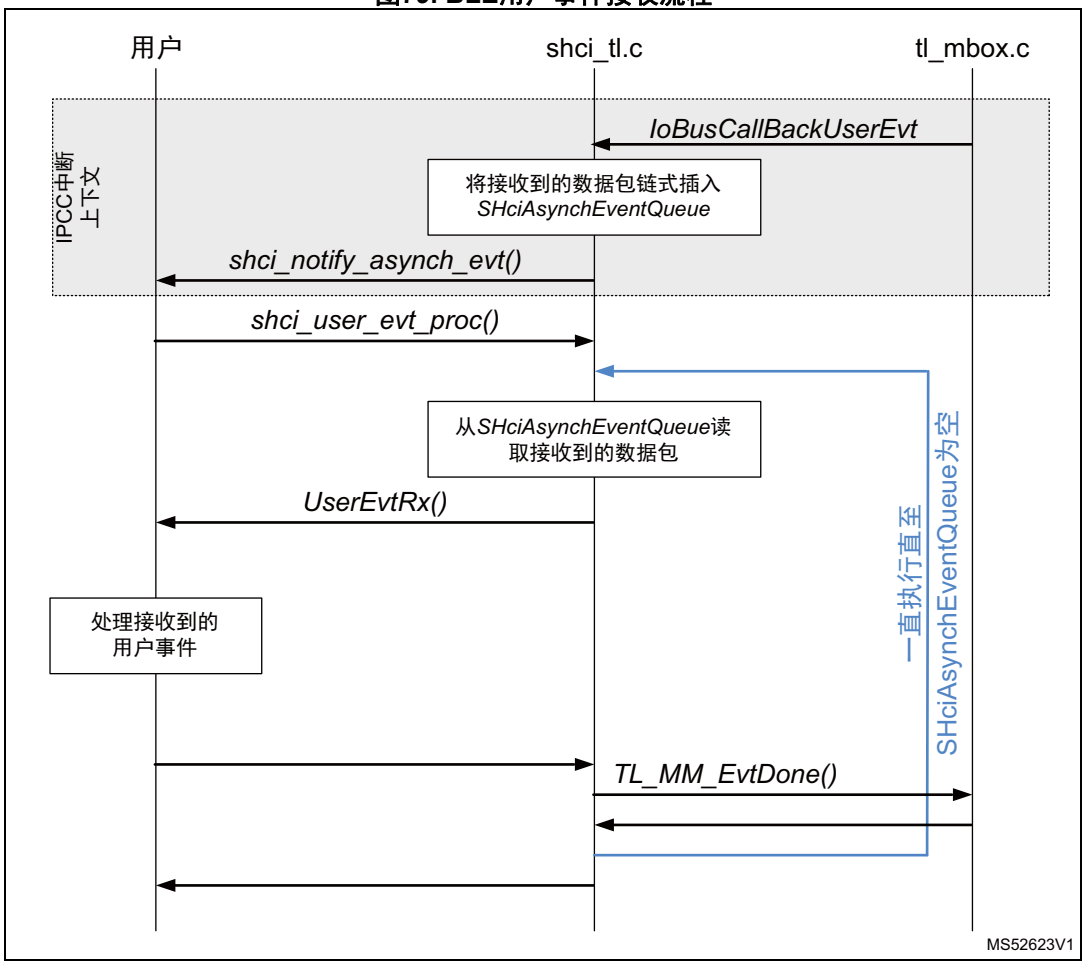
void hci_cmd_resp_wait(uint32_t timeout):

此函数通知用户已接收到挂起BLE命令的响应。

它在IPCC中断上下文中调用。在退出该API时，应用可以从API hci_cmd_resp_wait()返回。

参数没有意义。

**图79. BLE用户事件接收流程**




void hci_notify_asynch_evt(void* pdata):

此API通知用户已接收到BLE用户事件。然后，用户必须调用hci_user_evt_proc()来处理HCI传输层上的通知。由于hci_notify_asynch_evt()通知是从IPCC中断上下文调用的，强烈建议实现一种后台机制来调用hci_user_evt_proc()（在IPP中断上下文之外）。

pdata保存HciCmdEventQueue的地址。

void hci_user_evt_proc(void):

此函数通过UserEvtRx()将接收到的事件报告给用户。由于接收到的事件队列HciCmdEventQueue是在IPCC中断上下文中填充的，因此可以在用户处理事件时将新事件保存到队列中。为队列中的每个已接收事件调用UserEvtRx()。hci_user_evt_proc()处理负责在UserEvtRx() 每次返回时将缓冲区释放给CPU2内存管理器。

void UserEvtRx (void * pData):

此函数报告接收到的BLE用户事件。在此函数返回时，将释放保存了已接收事件的缓冲区。

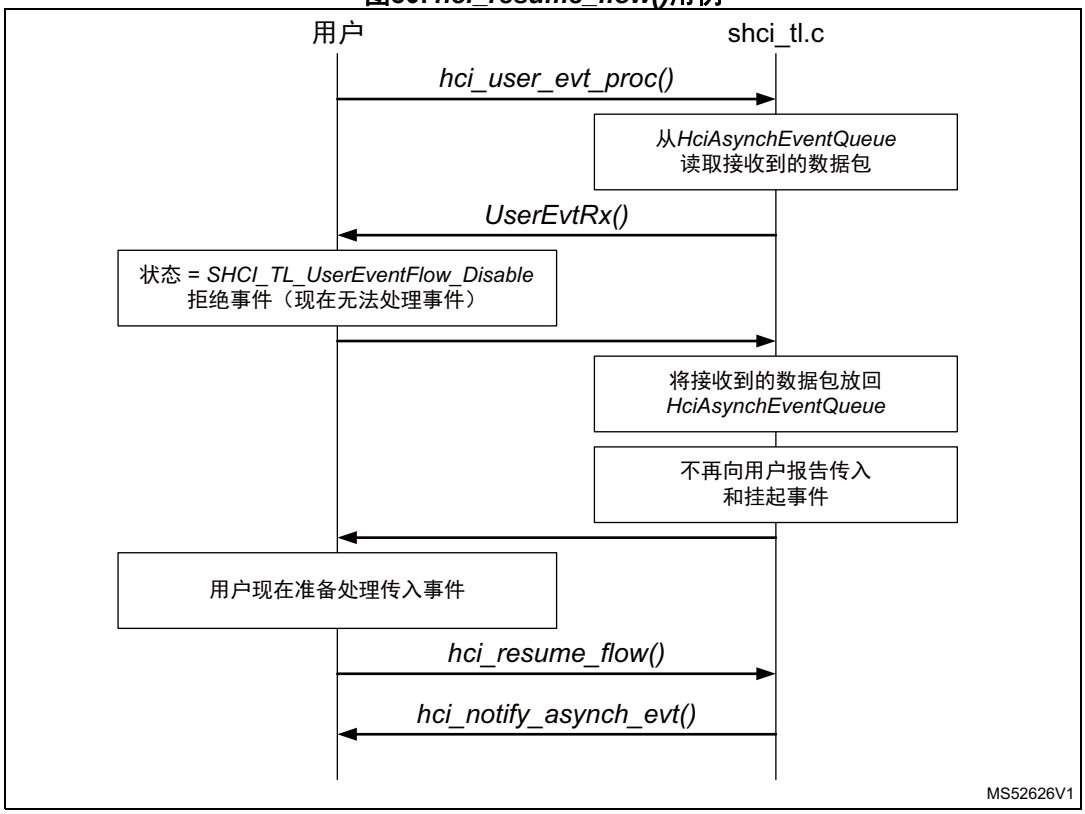
pData是保存下列参数的结构地址：

```
typedef struct
{
HCI_TL_UserEventFlowStatus_t status;
TL_EvtPacket_t *pckt;
} tHCI_UserEvtRxParam;
```

pckt：保存了已接收事件的地址

status：为用户提供了一种方法，用于将已接收数据包尚未处理且不得丢弃的情况通知HCI传输层。如果UserEvtRx()返回时用户没有填充，此参数会被设置为HCI_TL_UserEventFlow_Enable，表示用户处理了接收到的事件。

**图80. *hci_resume_flow()*用例**




void hci_resume_flow( void ):

当用户不能处理传入事件时，必须在从UserEvtRx()返回前将状态参数设置为HCI_TL_UserEventFlow_Disable。在这种情况下，HCI传输层不发布BLE用户事件，并且不报告任何新的传入事件。

当用户准备处理BLE用户事件时，必须发送hci_resume_flow()，以通知HCI传输层重新开始报告BLE用户事件。

## 13.5 STM32WB系统命令与事件

### 13.5.1 指令

表33. 系统接口命令

| 指令 | 代码 | 说明 |
|---|---|---|
| SHCI_C2_FUS_GetState() | 0xFC52 | 请参见*[5]*。 |
| SHCI_C2_FUS_FwUpgrade() | 0xFC54 | |
| SHCI_C2_FUS_FwDelete() | 0xFC55 | |
| SHCI_C2_FUS_UpdateAuthKey() | 0xFC56 | |
| SHCI_C2_FUS_LockAuthKey() | 0xFC57 | |
| SHCI_C2_FUS_StoreUsrKey() | 0xFC58 | |
| SHCI_C2_FUS_LoadUsrKey() | 0xFC59 | |
| SHCI_C2_FUS_StartWs() | 0xFC5A | |
| SHCI_C2_FUS_LockUsrKey() | 0xFC5D | |
| SHCI_C2_BLE_Init() | 0xFC66 | 发送BLE初始化参数。必须在其他ACI命令之前发送。更多详细信息，请参见*第 7.6.5 节：如何使数据吞吐量最大化*。 |
| SHCI_C2_THREAD_Init() | 0xFC67 | 请参见*第 9.8 节：Thread应用的系统命令*。 |
| SHCI_C2_DEBUG_Init | 0xFC68 | 启用CPU1和CPU2上的跟踪功能以及CPU2上的GPIO调试配置。 |
| SHCI_C2_FLASH_EraseActivity | 0xFC69 | 通知CPU2 CPU1可能请求了闪存擦除操作。这将允许CPU2针对擦除操作启用时序保护。 |
| SHCI_C2_CONCURRENT_SetMode() | 0xFC6A | 请参见*第 9.8 节：Thread应用的系统命令*。 |
| SHCI_C2_FLASH_StoreData() | 0xFC6B | |
| SHCI_C2_FLASH_EraseData() | 0xFC6C | |
| SHCI_C2_RADIO_AllowLowPower() | 0xFC6D | |
| SHCI_C2_MAC_802_15_4_Init () | 0xFC6E | 请参见*第 12.4.5 节：MAC IEEE Std 802.15.4-2011 system*。 |
| SHCI_C2_Reinit() | 0xFC6F | 在接收到CPU1上的SEV指令生成的事件时请求CPU2重启其初始化阶段。预计SBSFU会在没有启动任何RF活动时使用此命令。 |
| SHCI_GetWirelessFwInfo() | | 返回在CPU2上运行的无线协议栈和FUS的版本和内存占用量。 |
| SHCI_C2_ZIGBEE_Init() | 0xFC70 | 初始化CPU2上的ZigBee®协议栈。 |

表33. 系统接口命令

| 指令 | 代码 | 说明 |
|---|---|---|
| SHCI_C2_ExtpaConfig() | 0xFC72 | 向CPU2发送要使用的GPIO及其配置，以驱动外部PA引脚的 启用/禁用。 |
| SHCI_C2_SetFlashActivityControl() | 0xFC73 | 请求CPU2使用PESD位或信号量7来保护其时序不受闪存操作的影响。如果不发送此命令，CPU2将使用PESD。 |

### 13.5.2 活动

表34. 用户系统事件

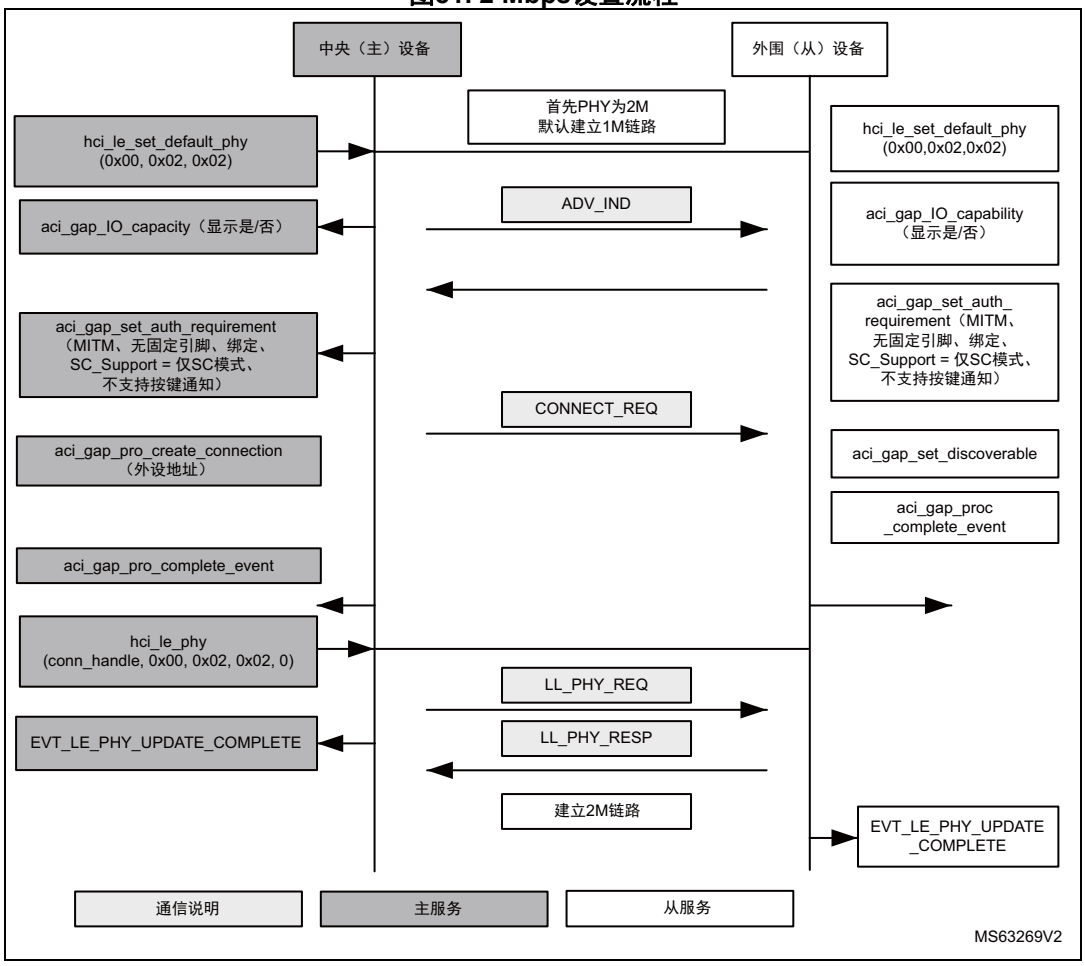
| 事件 | 代码 | 说明 |
|---|---|---|
| SHCI_SUB_EVT_CODE_READY | 0x9200 | 在CPU2启动并准备接收命令后立即返回。 |
| SHCI_SUB_EVT_ERROR_NOTIF | 0x9201 | 报告CPU2中的错误。 |

## 13.6 BLE - 设置2 Mbps链路

在设备初始化阶段，可以初始化首选TX_PHYS、RX_PHYS值。

在以1 Mbps的速率连接后，可以将该链路的PHY更改为2 Mbps，详见*图 81*。

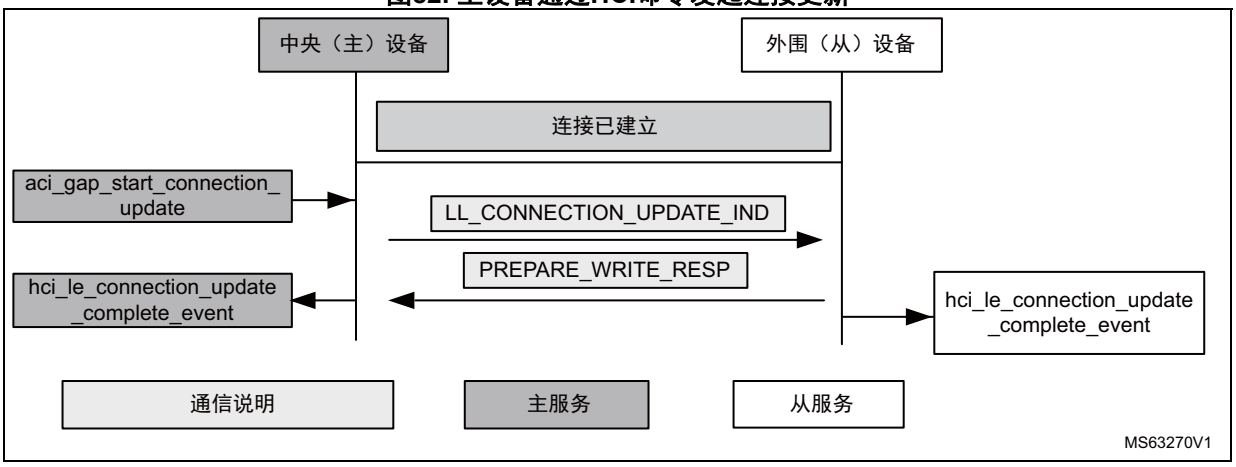
图81. 2 Mbps设置流程




## 13.7 BLE - 连接更新流程

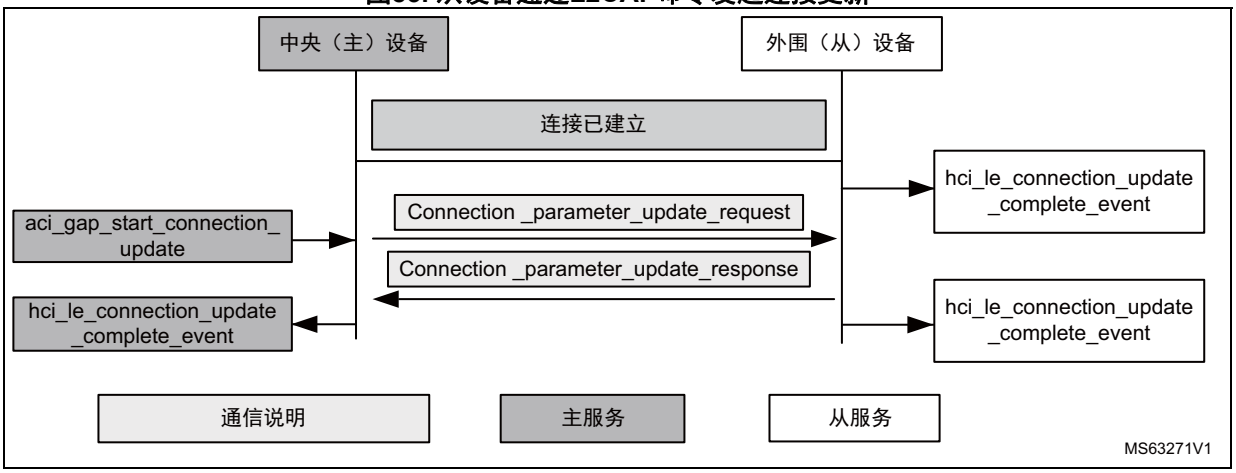
在建立连接后，主设备可以用aci_gap_start_connection_update命令更新连接参数。

**图82. 主设备通过HCI命令发起连接更新**




在建立连接后，从设备可以用aci_l2cap_connection_parameter_update_req命令更新连接参数。

**图83. 从设备通过L2CAP命令发起连接更新**



## 13.8 BLE - 安全流程

### 13.8.1 LE安全模式1级别3

*图 84*、*图 85*和*图 86* 显示了LE安全模式1级别3流程的示例，其中使用了下列初始化参数。

- 对于中央设备：
  - aci_gap_set_IO_capability (keyboard/display)
  - aci_gap_set_auth_requirement (MITM, no fixed pin, SC_Support=0 SC not supported)
- 对于外围设备：
  - aci_gap_set_IO_capability (display only)
  - aci_gap_set_auth_requirement (no MITM, fixed pin=0x1234, SC_Support=0 SC not supported)

**图84. LE安全模式1级别3密钥分配功能**

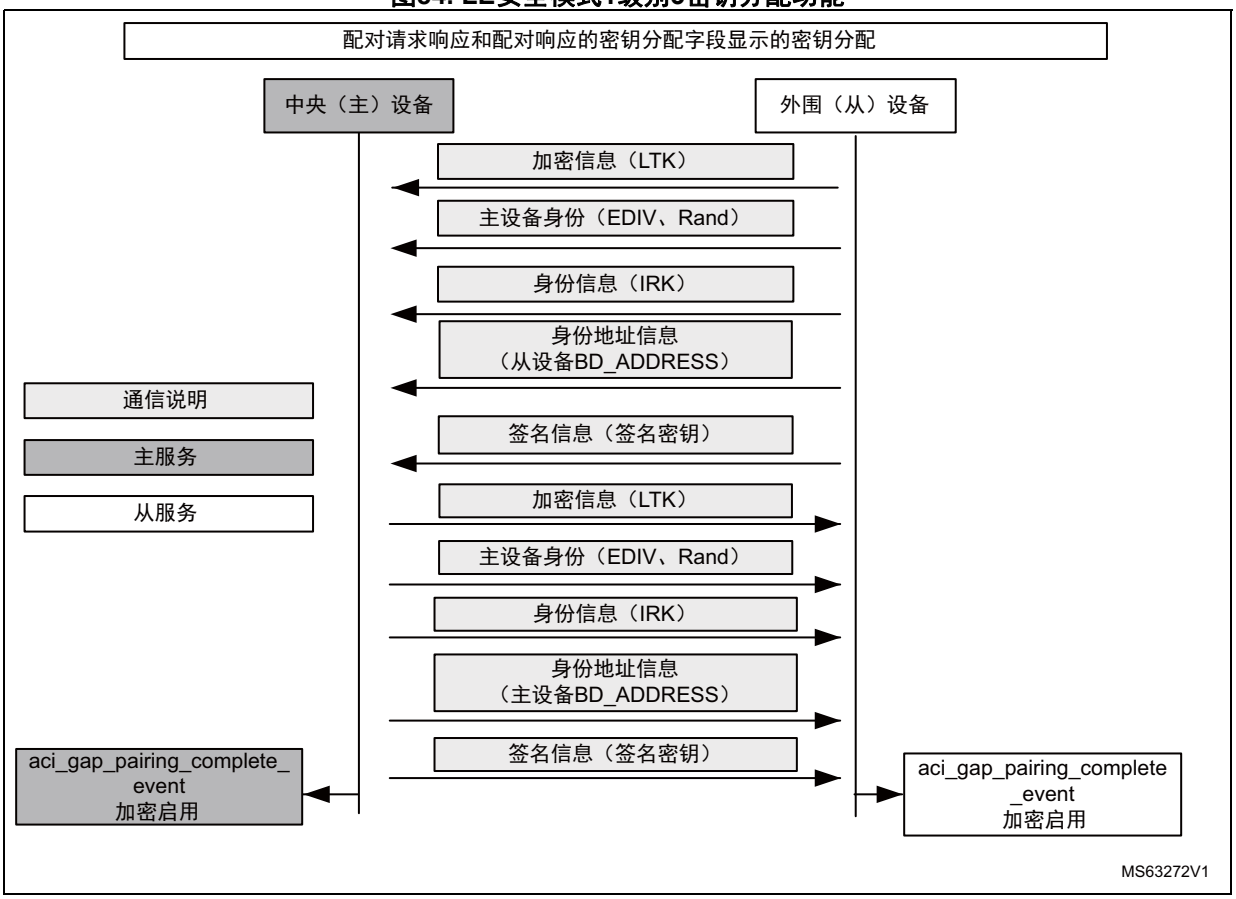


**图85. LE安全模式1级别3配对功能数据交换**

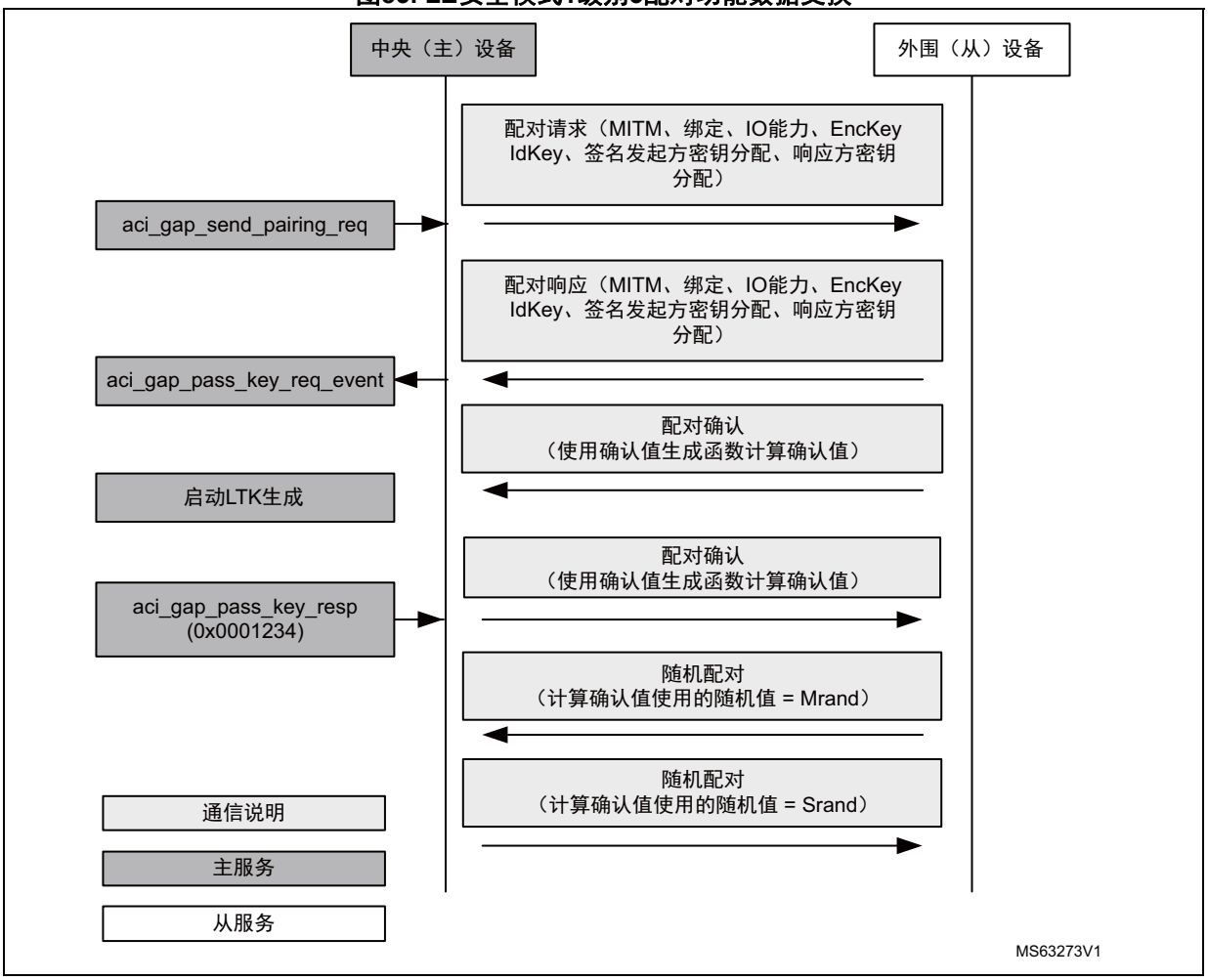
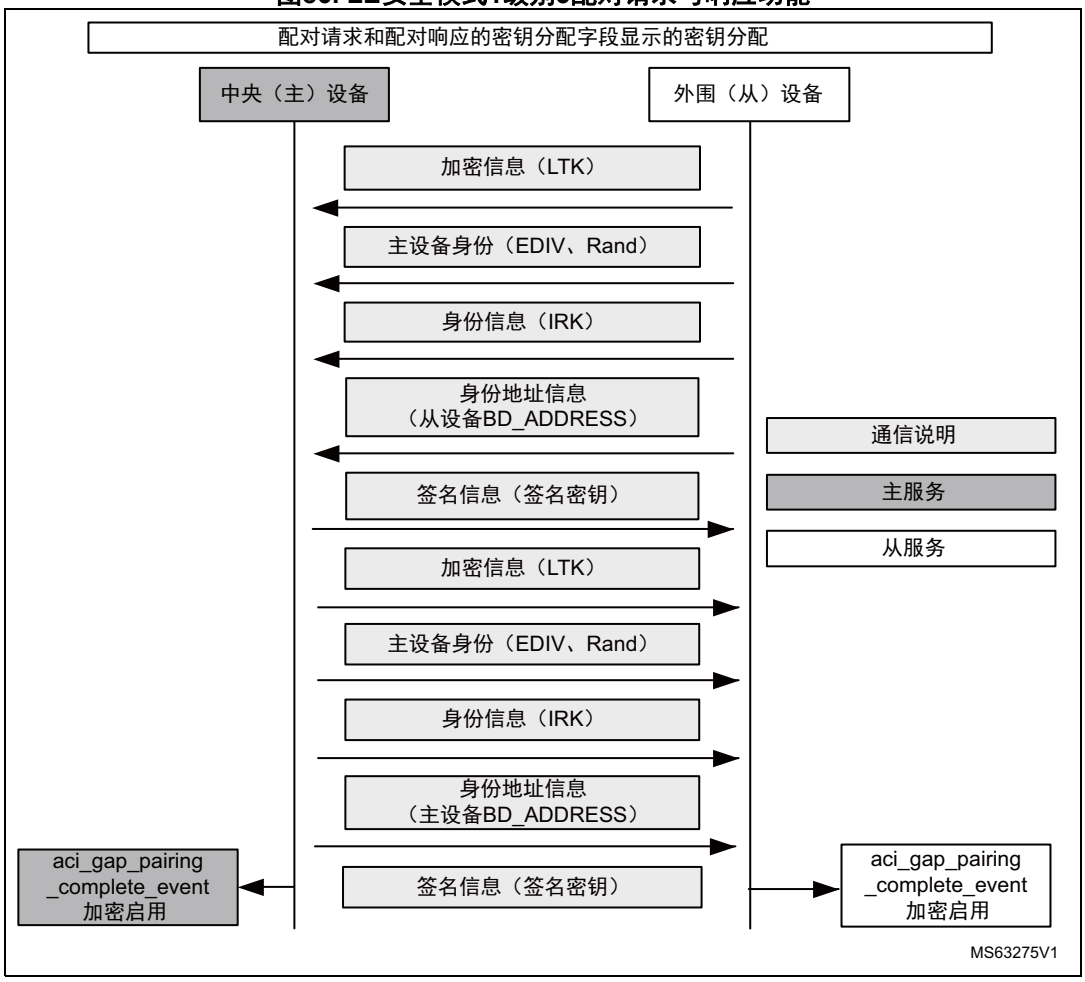


图86. LE安全模式1级别3配对请求与响应功能




## 13.8.2 LE安全模式1级别4

*图 87*、*图 88*和*图 89*显示了安全模式1级别4流程的示例，其中使用了下列初始化参数：

- aci_gap_set_IO_capability (display_yesno)
- aci_gap_set_auth_requirement (MITM, no fixed pin, SC_Support=SC only mode)

图87. LE安全模式1级别4

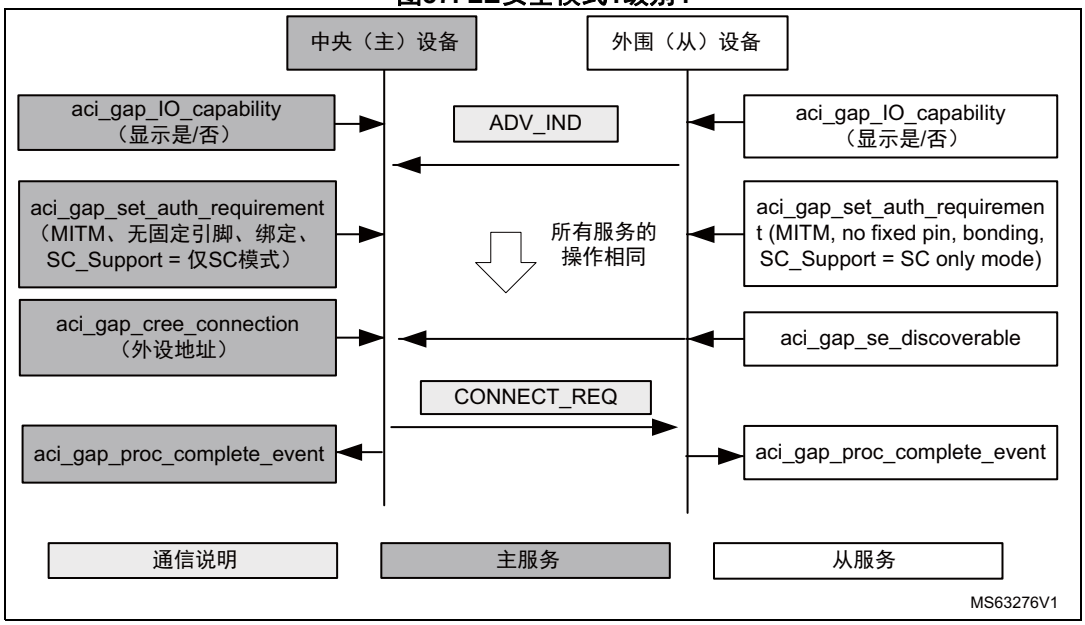



图88. LE安全模式1级别4

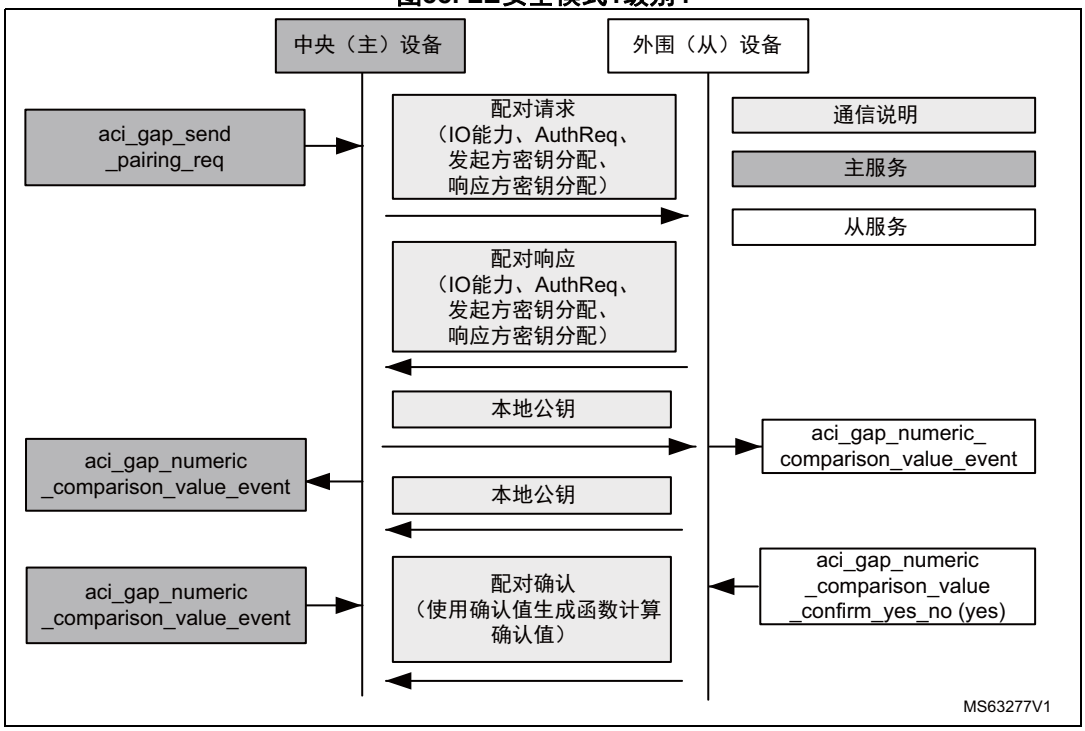
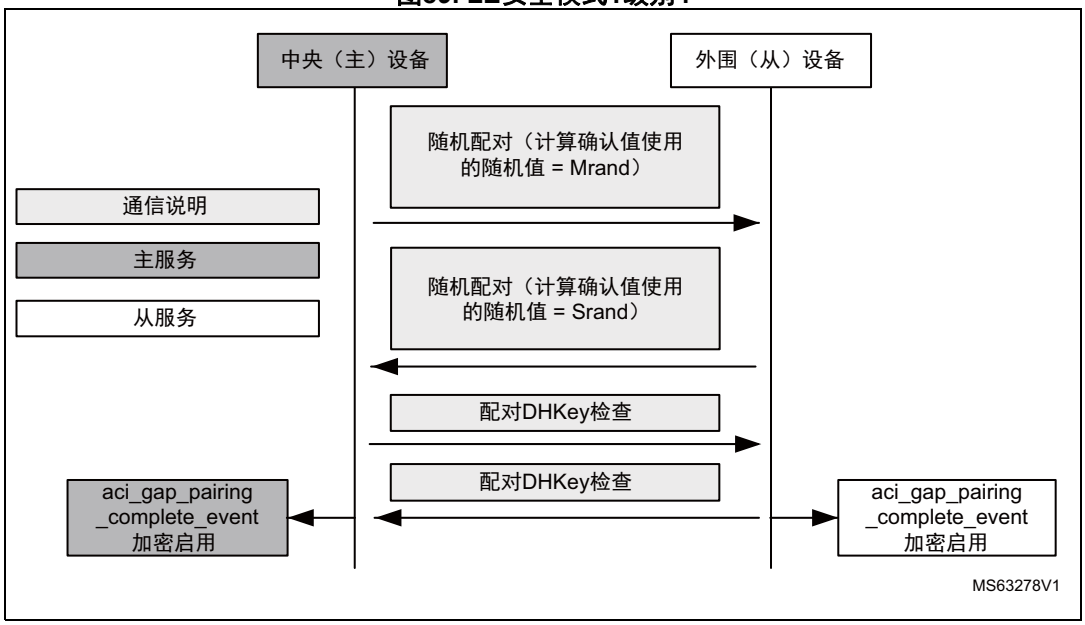


图89. LE安全模式1级别4




### 13.8.3 LE安全模式1级别4 - 按键通知

*图 90*和*图 91*显示了安全模式1级别4流程的示例，其中使用了下列初始化参数：

- aci_gap_set_IO_capability (display_yesno)
- aci_gap_set_auth_requirement (MITM, no fixed pin, SC_Support=SC only mode, keypress notification supported)

图90. LE安全模式1级别4 - 按键通知（1/2）

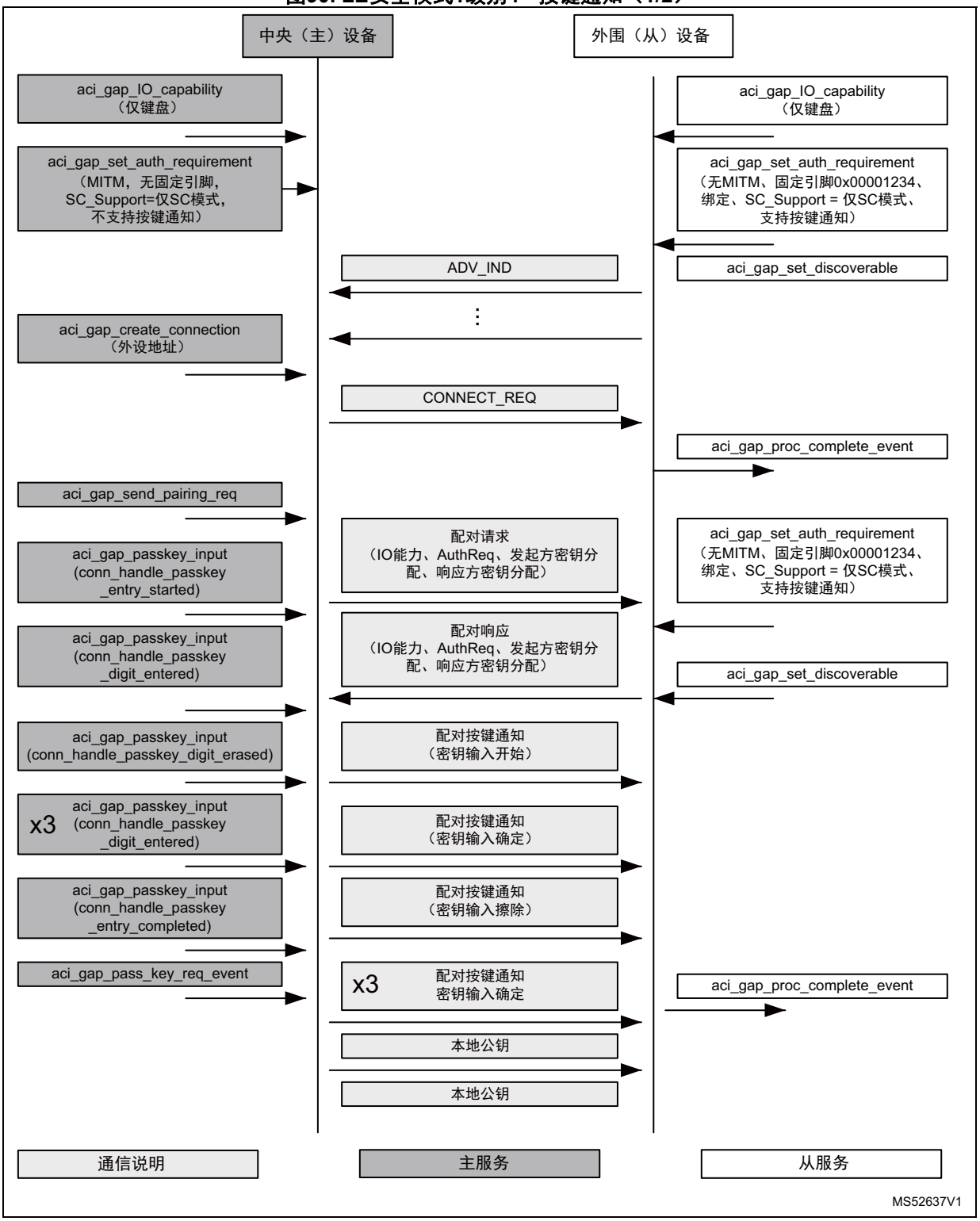
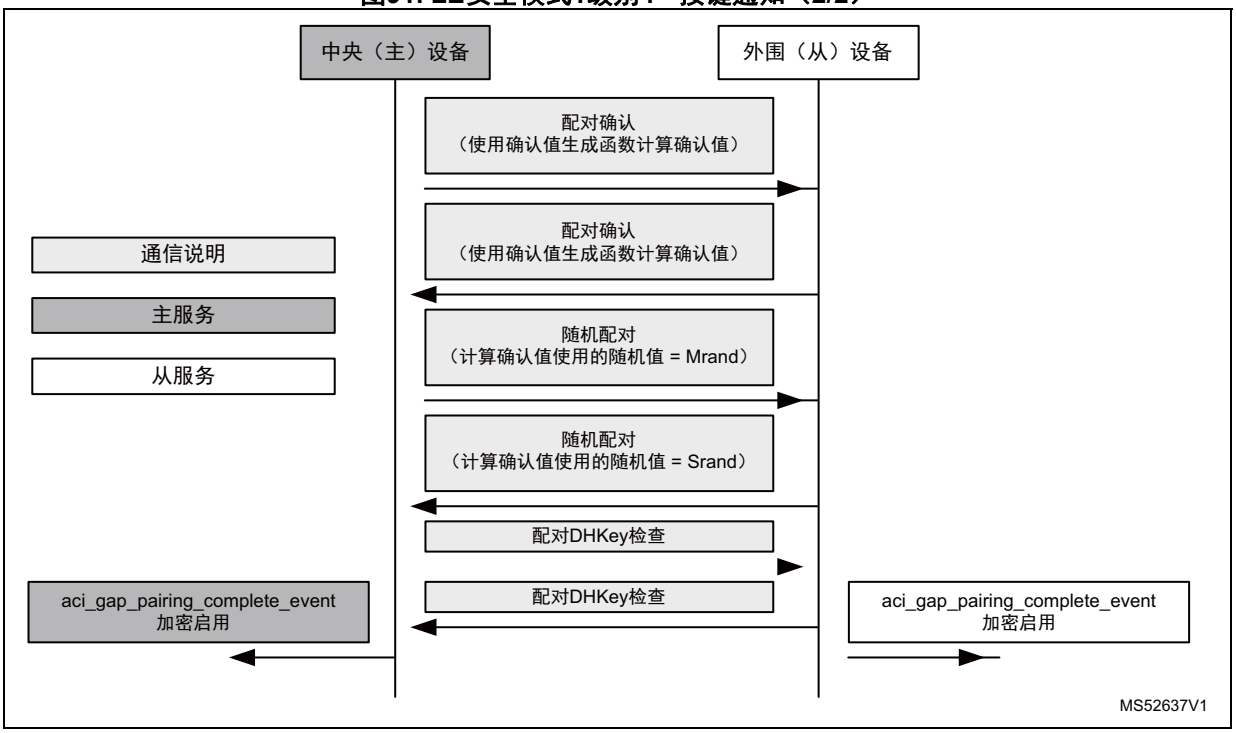


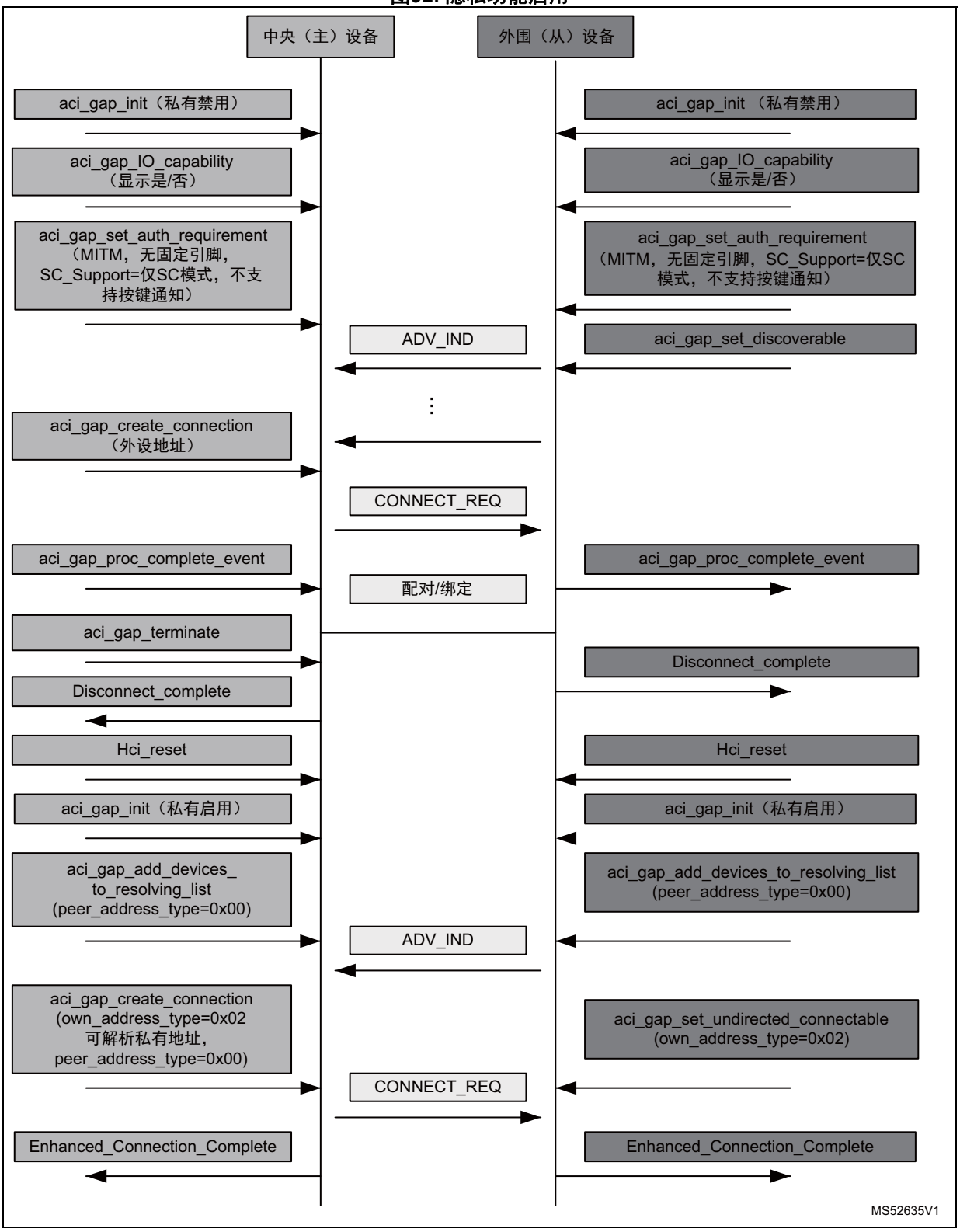
图91. LE安全模式1级别4 - 按键通知（2/2）




### 13.8.4 隐私功能启用

*图 92*显示了启用了隐私功能的设备的连接、配对、绑定、断开、复位和重新初始化流程的示例。

图92. 隐私功能启用


中央（主）设备
外围（从）设备
aci_gap_init（私有禁用）
aci_gap_init （私有禁用）
aci_gap_IO_capability
（显示是/否）
aci_gap_IO_capability
（显示是/否）
aci_gap_set_auth_requirement
（MITM，无固定引脚，
SC_Support=仅SC模式，不支
持按键通知）
aci_gap_set_auth_requirement
（MITM，无固定引脚，SC_Support=仅SC
模式，不支持按键通知）
ADV_IND
aci_gap_set_discoverable
aci_gap_create_connection
（外设地址）
CONNECT_REQ
aci_gap_proc_complete_event
aci_gap_proc_complete_event
配对/绑定
aci_gap_terminate
Disconnect_complete
Disconnect_complete
Hci_reset
Hci_reset
aci_gap_init（私有启用）
aci_gap_init（私有启用）
aci_gap_add_devices_
to_resolving_list
(peer_address_type=0x00)
aci_gap_add_devices_to_resolving_list
(peer_address_type=0x00)
ADV_IND
aci_gap_create_connection
(own_address_type=0x02
可解析私有地址，
peer_address_type=0x00)
aci_gap_set_undirected_connectable
(own_address_type=0x02)
CONNECT_REQ
Enhanced_Connection_Complete
Enhanced_Connection_Complete
MS52635V1

## 13.9 BLE - 链路层数据包

BLE对广播和数据通道数据包使用一种数据包格式。

**图93. 数据包结构**

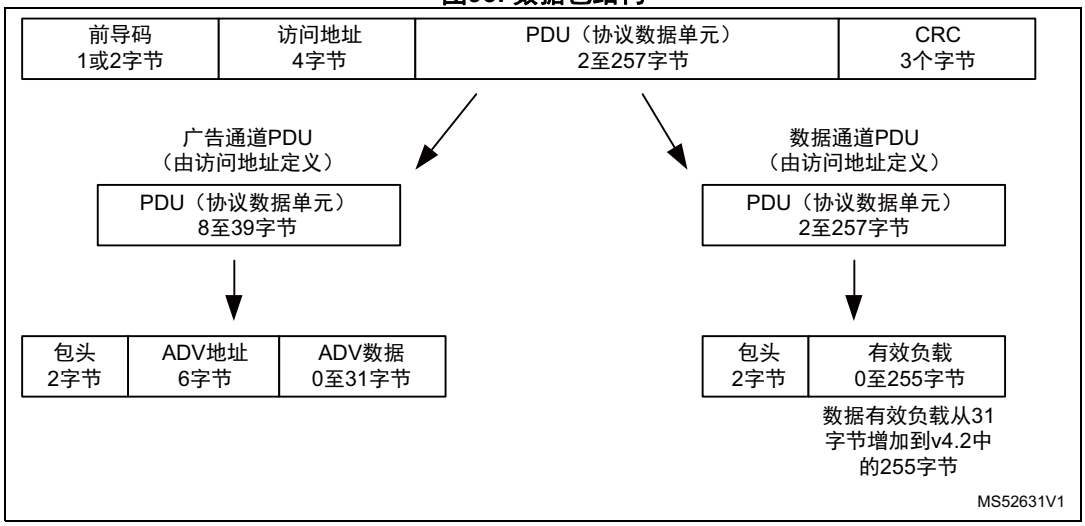
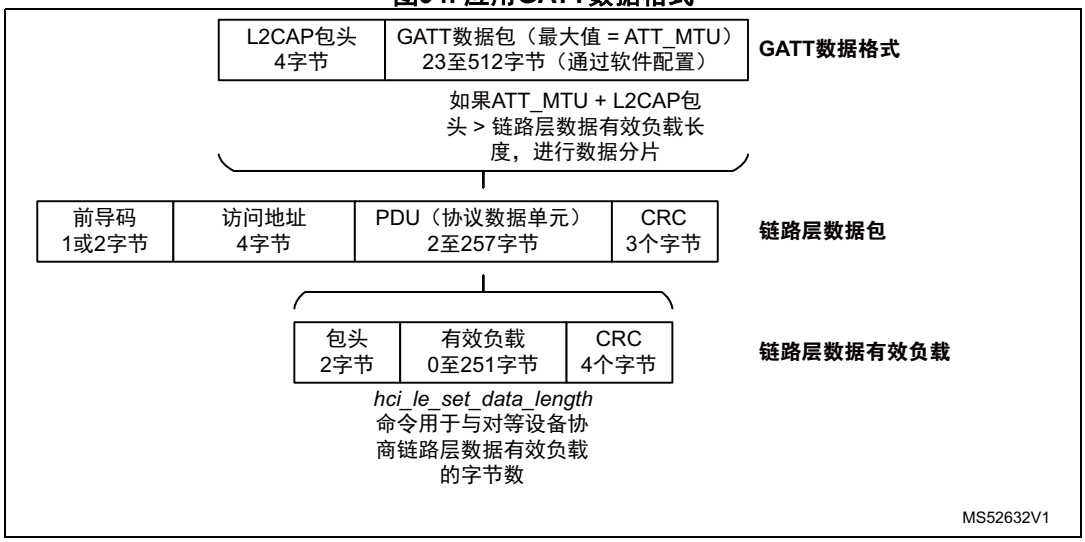



**图94. 应用GATT数据格式**




## 13.10 Thread概述

### 13.10.1 引言

Thread协议栈是可靠、高性价比、低功耗的无线D2D通信的开放标准。它专为需要基于IP的网络且可以在栈上使用各种应用层的联网家居应用而设计。

完整规范（*[9]*）可以在http://threadgroup.org/上找到。

此标准基于在2.4 GHz频带内以250 kbps的速率工作的IEEE 802.15.4 [IEEE802154] PHY（物理）和MAC（介质访问控制）层。

### 13.10.2 主要特性

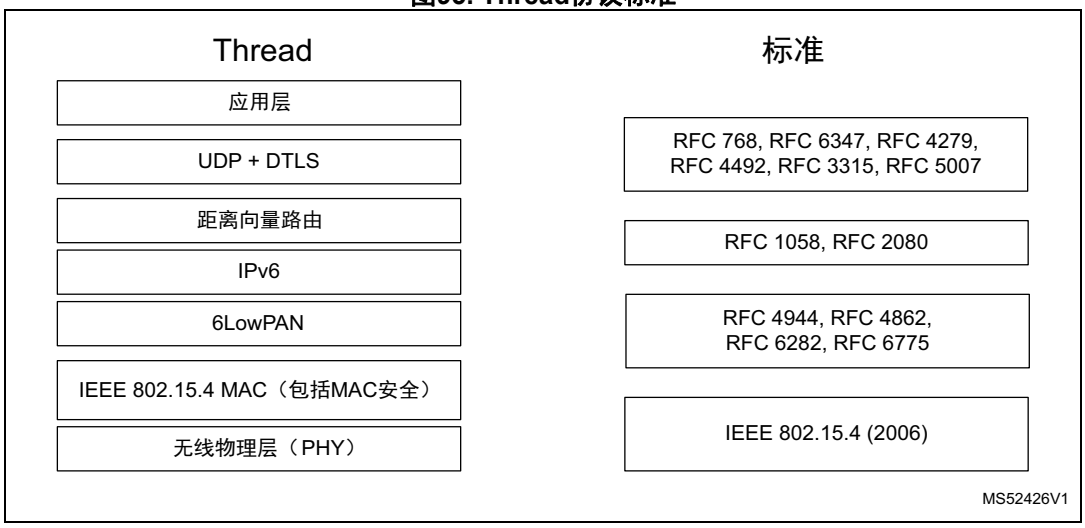
Thread面向智能家居应用，例如环境控制、恒温器、报警、能源管理、智能锁和智能照明设备。此标准的一个主要优势是它基于IPv6，因此可以轻松地将任何Thread网络连接到任何其他IPV6应用。它的另一大优势是它基于真正的Mesh网络。此网络一经部署，将十分稳健可靠。例如，当路由失败时，通过找到通往目的地的新路径，系统能够完成自身的自动重新配置。通过Mesh网络，设备之间可以进行更长距离的通信。

Thread并没有真正定义任何应用层。尽管如此，大多数Thread应用程序使用CoAP来传输数据。例如，CoAP被广泛部署，并且已经在Thread内部本地用于地址解析管理。在STM32WB设备上，CoAP 层向客户公开。

### 13.10.3 协议层

Thread基于成熟并经过充分验证的标准，如*图 95*所示。

**图95. Thread协议标准**

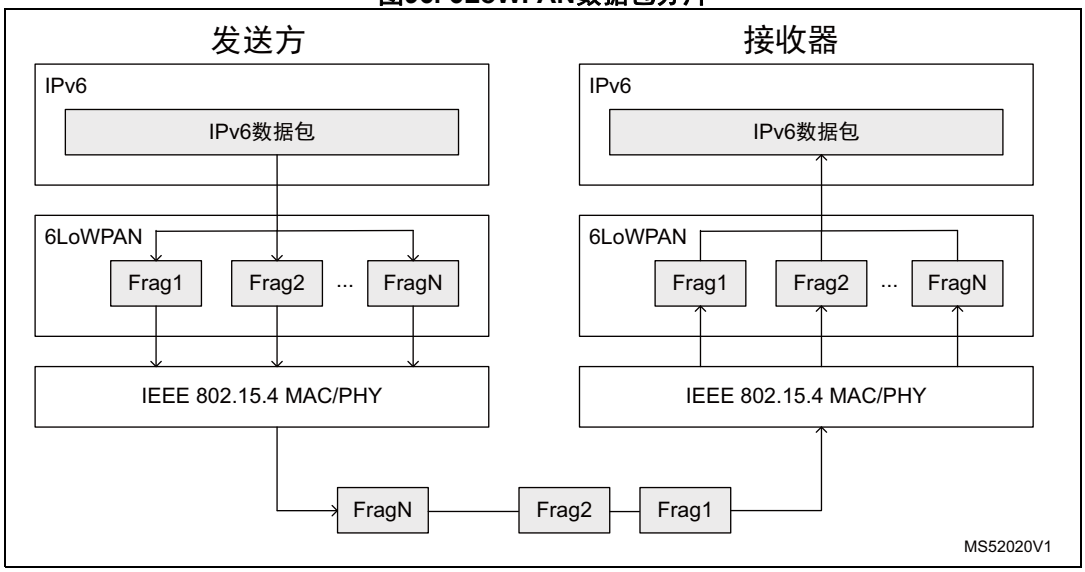



从底层开始：

- MAC层，仅基于IEEE 802.15.4规范（2006年）的子集。它支持2.4 GHz频带内250 kbps的速率。有16个可用通道，为通道11至通道26。在Thread网络内部，只有一个实时使用的通道。MAC无线层使用CSMA机制发送帧。如果传输介质处于忙碌状态，将延迟发送，延迟时间为随机值。该机制降低了两个节点之间发生传输冲突的可能性。在STM32WB上，传输延迟的随机值由硬件直接管理。。

- 6LoWPAN层：6LoWPAN表示“基于IPv6的低功耗无线个域网”。在以太网上，1280字节的IPV6数据包可以轻松地作为一个“单片”帧发送。在MAC层，由于最大数据包大小被限制在127字节，因此不可能这样做。因此，Thread使用6LowPan层。该层实现两种技术：
  - 分片（将数据包分成小块发送，并在接收时重新组合）
  - 包头压缩（在某些情况下，可以将48字节的包头压缩成只有6字节）。

**图96. 6LoWPAN数据包分片**




- IPv6（网际协议第6版），意在取代IPv4。

  IPv4使用32位寻址，IPV6使用128位寻址，提供了数十亿种可能性。除了更大的寻址空间，IPv6还具备其他技术优势，特别是简化了路由流程。

  Thread中定义了多个地址：
  - MeshLocal64：此地址是“拓扑独立的”，这意味着它是稳定的，永远不会改变，无论设备成为路由器或终端设备。MeshLocal64 地址通常是从一台设备 ping 到另一台设备时使用的地址。
  - MeshLocal16：即使应用使用mMeshLocal64，底层协议栈仍将使用mMeshLocal16地址执行路由。Mesh Local包含RLOC字段（路由位置符）。该地址与拓扑有关

（取决于网络和链路质量）。子设备可以决定选择新的父设备，也就选择了新的路由器，并获得了新地址。子设备也可以成为路由器，这取决于使用情况。

  - MeshLinkLocal64：地址以 0xFE80 开头，以 MAC全局/本地位反转的扩展地址结尾。它用于直接点对点链接和 MLE 消息。

- 路由：所有Mesh网络管理都基于MLE（Mesh链路建立）信息。这些信息用于检测相邻设备，配置系统，以及维护整个网络上的路由开销。Thread被认为十分稳健且能够管理动态路由自适应。路由器定期发送广播信息，其中包含下列参数：
  - 发送方与其相邻设备之间的链路质量
  - 访问Thread网络分区中所有路由器的路由开销。

  所有路由器都包含一个表格，其中包含与其所有相邻路由器的UL和DL的链路质量，以及Mesh网络内部存在的所有路由器的路由成本。

  该表格还包含了“下一跃点” 和所谓的“寿命”值，前者定义了如何遍历整个网络，后者表示自上一次接收广播以来的运行时间。

  链路质量是一个0到3之间的值，3表示最佳质量（当记录的信号强度超过20 dB时）。只有四个可能的链路质量值以最大限度地减少与相邻通信链路质 量的开销。充当主导设备的路由器维持着一个额外的数据库，用于跟踪路由器ID分配和与每个路由器关联的扩展地址。
- 应用层：Thread支持CoAP，并且该协议在我们的设计中充当应用层。 CoAP可以被认为是 http 协议的一个非常轻量的版本 它需要的资源要比http少得多，并且开销极低。与http一样，CoAP基于REST模型：服务器通过URL提供资源，客户端使用Get、Put、Post和Delete等请求访问这些资源。URL（统一资源定位符）指定了资源和访问资源的方式。有四种类型的信息：
  - 无需确认的信息
  - 需要确认的信息
  - 确认信息
  - 复位信息。

### 13.10.4 网状拓扑

Thread支持Mesh网络。如*图 97*所示，Thread网络中的设备可以有两种主要角色：

- 路由器
  - 为网络设备转发数据包
  - 为尝试加入网络的设备提供安全调试服务
  - 使其收发器一直维持启用状态
- 终端设备
  - 主要与一个路由器通信
  - 不为其他网络设备转发数据包
  - 可以禁用其收发器以降低功耗

在所有路由器中，总有一个会升级为“主导路由器(Leader)”。Thread主导路由器（Leader）负责管理Thread网络中的一组路由器。。

在所有终端设备中，，可以有休眠终端设备、REED或标准终端设备。

- REED（适任路由器的终端设备）是一种在需要时可以升级为路由器的终端设备。
- 休眠终端设备通常被禁用，偶尔被唤醒以便轮询来自其父设备的信息或发送数据。

Mesh网络的大小是可配置的。最多有32个活动路由器。每个路由器均可以被不同的子设备连接。每个子设备ID均为9位编码，因此理论上每个路由器最多有511个子设备。由于STM32WB上的内存限制，每个路由器的子设备数量被限制为10个。

**图97. Thread网络拓扑**

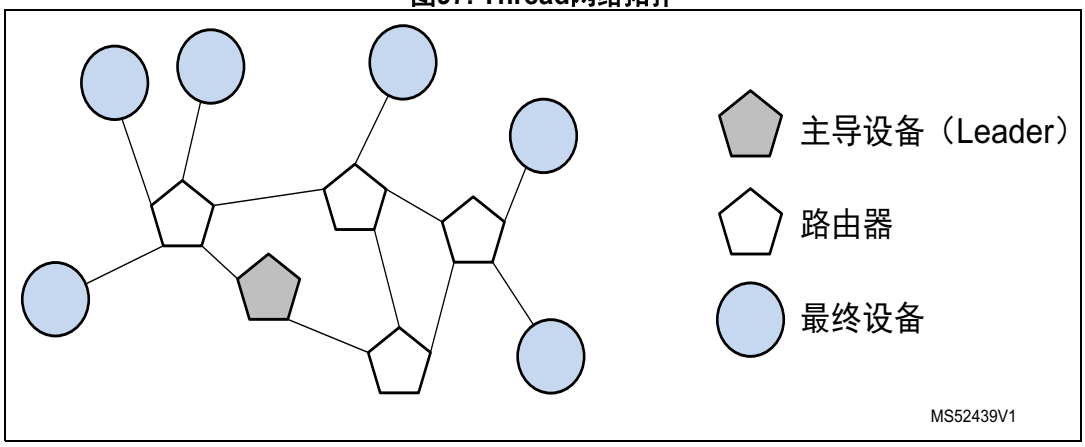
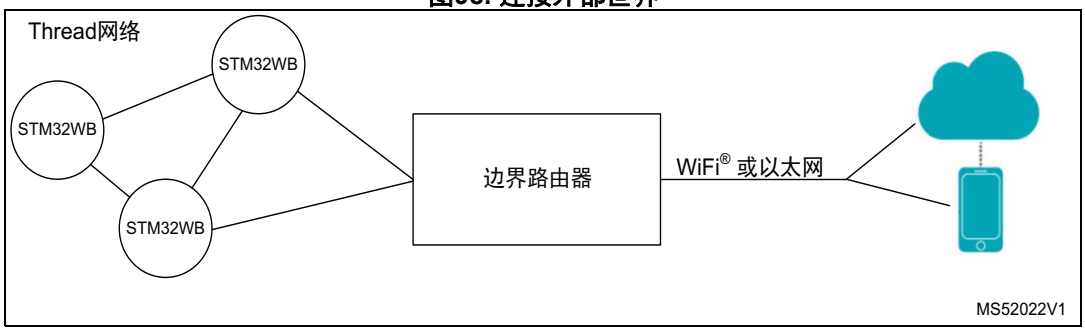



**图98. 连接外部世界**



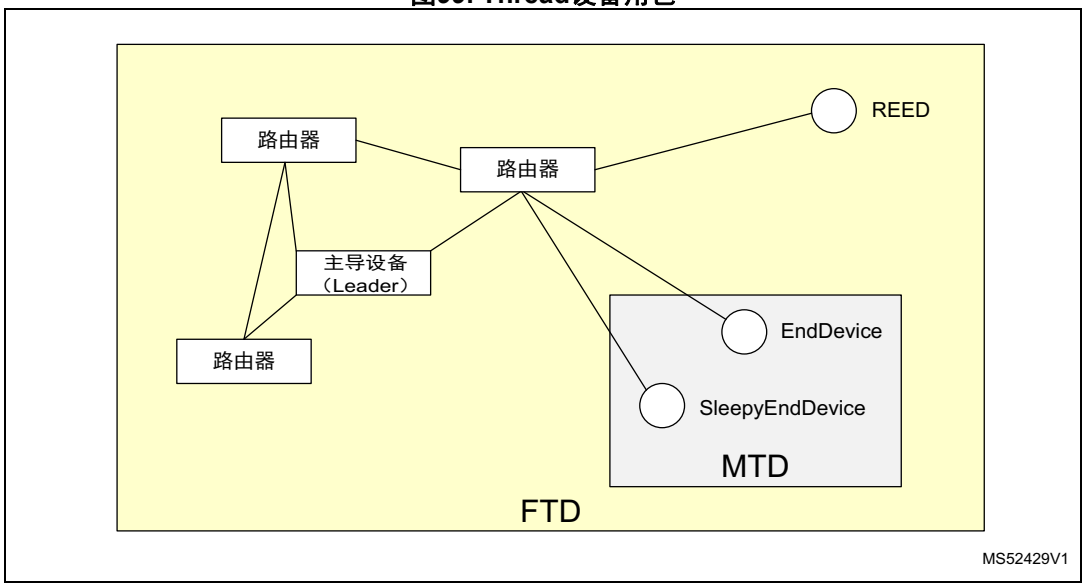

## 13.10.5 Thread配置

在编译OpenThread时，可根据针对的使用情况设置多个选项。对于STM32WB，将考虑两种情况：

1. 全功能Thread设备（FTD）：设备可充当Thread网络中简单的休眠终端设备，但也可充当路由器和主导设备。
2. 最低功能Thread设备（MTD）：设备只充当终端设备或休眠终端设备。

相比于FTD配置，MTD配置需要的存储空间较少（RAM和闪存）。MTD只充当终端设备或休眠终端设备。

图99. Thread设备角色



# 14 结论

基于STM32WB系列微控制器的低功耗蓝牙（BLE）或802.15.4应用要求用户理解专用软件协议和架构。

本文档详细描述了开发人员必须用什么样的方式创建嵌入式应用软件，其中的一个关键因素是遵循系统初始化的正确流程。

# 15 版本历史

表35. 文档版本历史

| 日期 | 版本 | 变更 |
|---|---|---|
| 2019 年 6 月 18 日 | 1 | 初始版本。 |
| 2019年9月26日 | 2 | 更新了*引言*，*第 4.2 节：存储器映射*，*第 4.3 节：共享外设*，*第 9.2 节：如何开始*，*第 9.6 节：OpenThread API*，*第 11.1 节：Thread_Cli_Cmd*，*第 11.4 节：Thread_Coap_Multiboard*，*第 11.5 节：Thread_Commissioning*，*第 12.3 节：API*和*第 12.4.3 节：MAC应用处理器固件*。<br>增加了*第 11.8 节：Thread FUOTA*及其包含的小节。<br>更新了*图 4：存储器映射*。<br>更新了*表 2：信号量*。 |
| 2020年3月23日 | 3 | 更新了*第 4.3 节：共享外设*和*第 7.6.1 节：如何设置蓝牙设备地址*。<br>增加了*第 4.7 节：闪存管理*、*第 4.8 节：CPU 中的调试信息*、*第 4.9 节：FreeRTOS 低功耗* 及其包含的小节。<br>更新了*表 2：信号量*、*表 33：系统接口命令*和*表 34：用户系统事件*。<br>更新了*图 10：在闪存中写入/擦除数据的算法*、*图 24：心率项目 - 中间件与用户应用之间的交互*和*图 29：P2P 服务器软件通信*。<br>对整个文档进行少量文字修订。 |

表36. 中文文档版本历史

| 日期 | 版本 | 变更 |
|---|---|---|
| 2021年 10月 1日 | 1 | 中文初始版本。 |

**重要通知 - 请仔细阅读**

意法半导体公司及其子公司 （“ST”）保留随时对 ST 产品和 / 或本文档进行变更、更正、增强、修改和改进的权利，恕不另行通知。买方在订货之前应获取关于 ST 产品的最新信息。 ST 产品的销售依照订单确认时的相关 ST 销售条款。

买方自行负责对 ST 产品的选择和使用， ST 概不承担与应用协助或买方产品设计相关的任何责任。

ST 不对任何知识产权进行任何明示或默示的授权或许可。

转售的 ST 产品如有不同于此处提供的信息的规定，将导致 ST 针对该产品授予的任何保证失效。

ST 和 ST 徽标是 ST 的商标。若需 ST 商标的更多信息，请参考 *www.st.com/trademarks*。所有其他产品或服务名称均为其各自所有者的财产。

本文档中的信息取代本文档所有早期版本中提供的信息。