life.augmented

AN5289
应用笔记

使用 STM32WB 系列微控制器构建射频应用

引言

本文档一步一步地指导设计人员基于 STM32WB 系列微控制器构建特定 Bluetooth®低功耗或 802.15.4 应用。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、Bluetooth®低功耗技术、802.15.4 OpenThread 协议、Zigbee®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 参考文件

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

[2] RM0434(1) 基于多协议射频 32 位 MCU Arm®的 Cortex®-M4 使用 FPU、 Bluetooth®低功耗耗和 802.15.4 射频解决方案

[3] AN5270(1) STM32WBx5 Bluetooth®低功耗射频接口

[4] UM2442(1) STM32WB HAL 与底层驱动程序说明

[5] UM2288(1) 用于射频性能测量的 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）

[10] AN5506(1) 在 STM32WB 系列上开始使用 Zigbee®

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	Bluetooth®低功耗
CLI	命令行接口
CoAP	受限应用协议
CPU1	Cortex®-M4 内核
CPU2	Cortex®-M0+ 内核
D2D	设备到设备
DUT	被测设备
FUOTA	射频固件更新
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	射频
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 所支持的射频协议栈

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射频协议栈（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 可以提供一个 BT HCI 标准接口，而 CPU1 上可以运行一个不同的 BLE 堆栈。

表 1. STM32WB 系列微控制器支持的射频协议栈

支持的射频协议栈	相关固件
BLE	stm32wb5x_BLE_HCI_AdvScan_fw.bin stm32wb5x_BLE_HCILayer_fw.bin stm32wb5x_BLE_LLD_fw.bin stm32wb5x_BLE_Stack_full_fw.bin stm32wb5x_BLE_Stack_light_fw.bin
Thread	stm32wb5x_Thread_FTD_fw.bin stm32wb5x_Thread_MTD_fw.bin
BLE 和 Thread	stm32wb5x_BLE_Thread_dynamic_fw.bin stm32wb5x_BLE_Thread_static_fw.bin
BLE 和 MAC 802_15_4	stm32wb5x_BLE_Mac_802_15_4_fw.bin
BLE 和 Zigbee	stm32wb5x_BLE_Zigbee_FFD_dynamic_fw.bin stm32wb5x_BLE_Zigbee_FFD_static_fw.bin stm32wb5x_BLE_Zigbee_RFD_dynamic_fw.bin stm32wb5x_BLE_Zigbee_RFD_static_fw.bin stm32wb5x_Zigbee_FFD_fw.bin stm32wb5x_Zigbee_RFD_fw.bin
MAC 802_15_4	stm32wb5x_Mac_802_15_4_fw.bin stm32wb5x_Phy_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

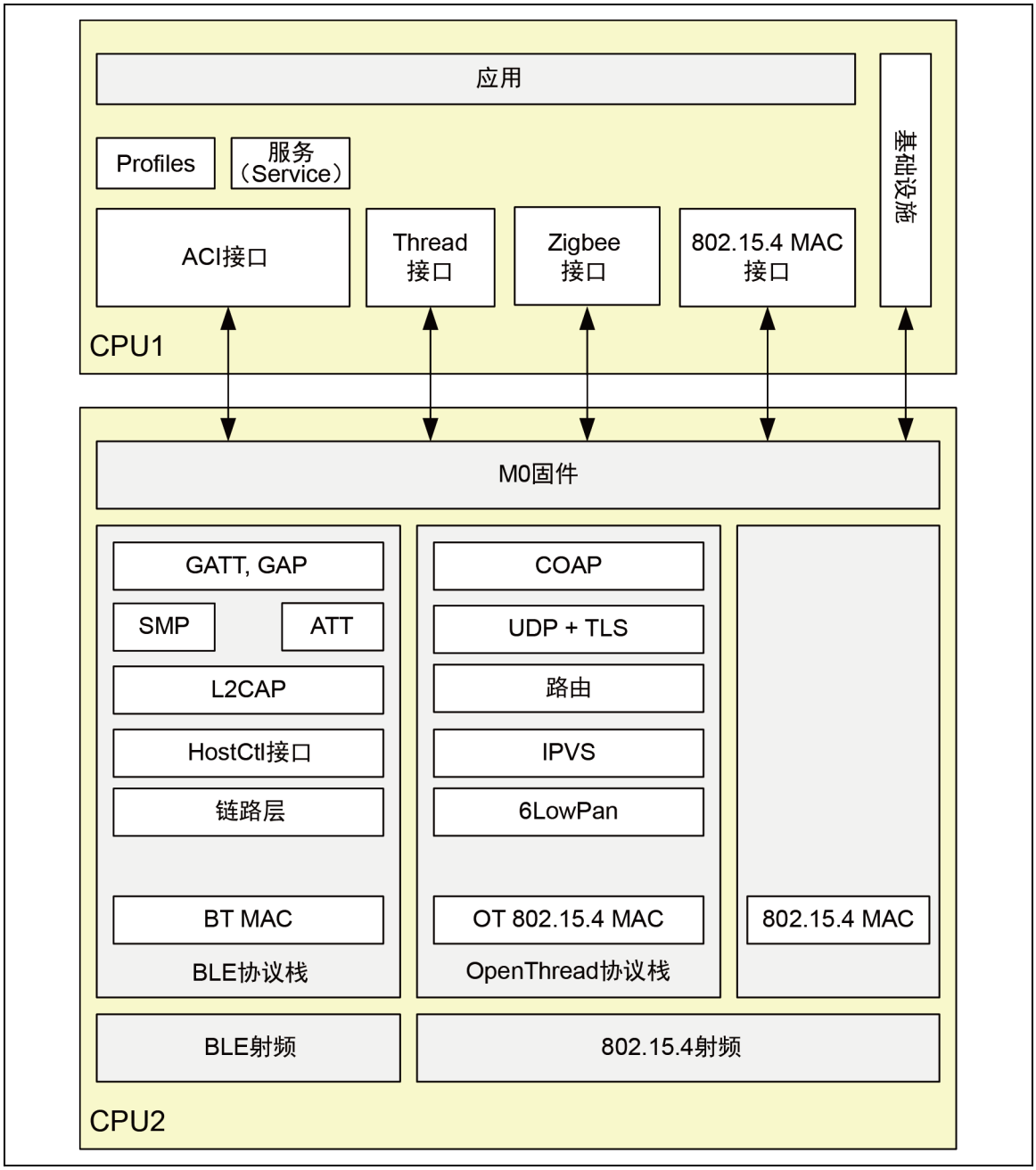
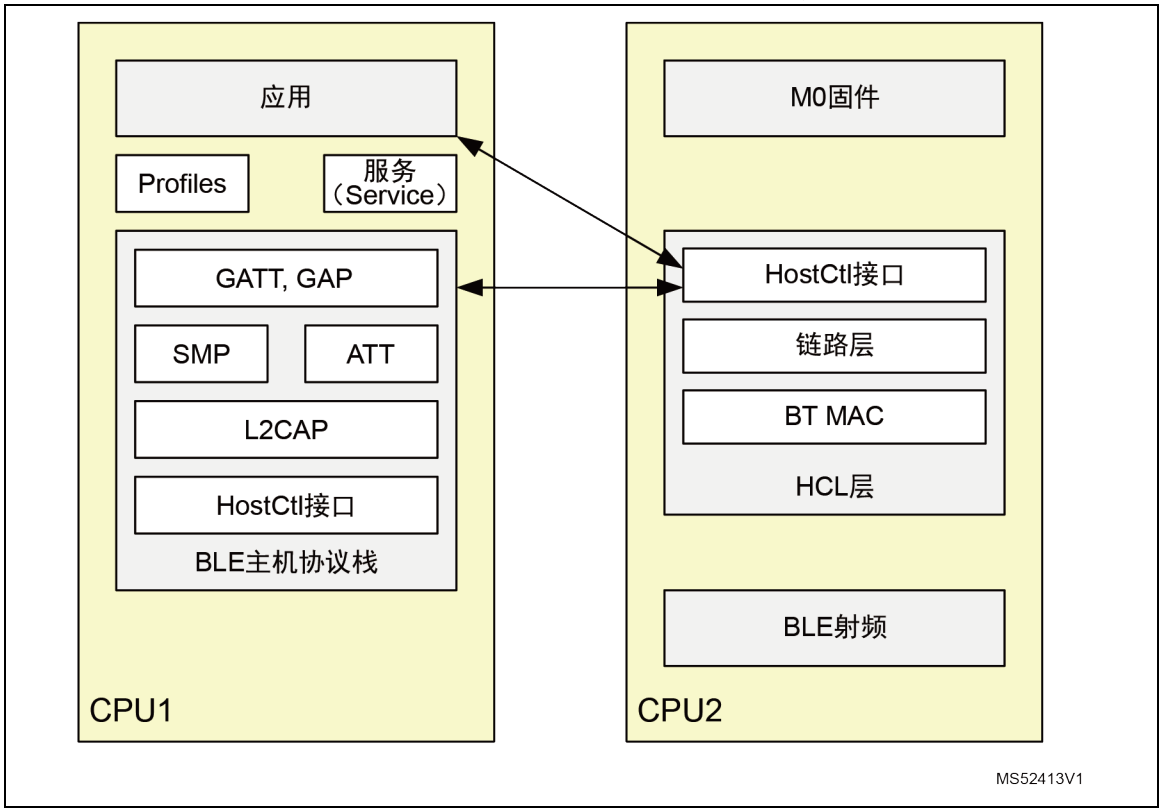


图 2. STM32WB 系列微控制器 BLE HCI 层模型



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。

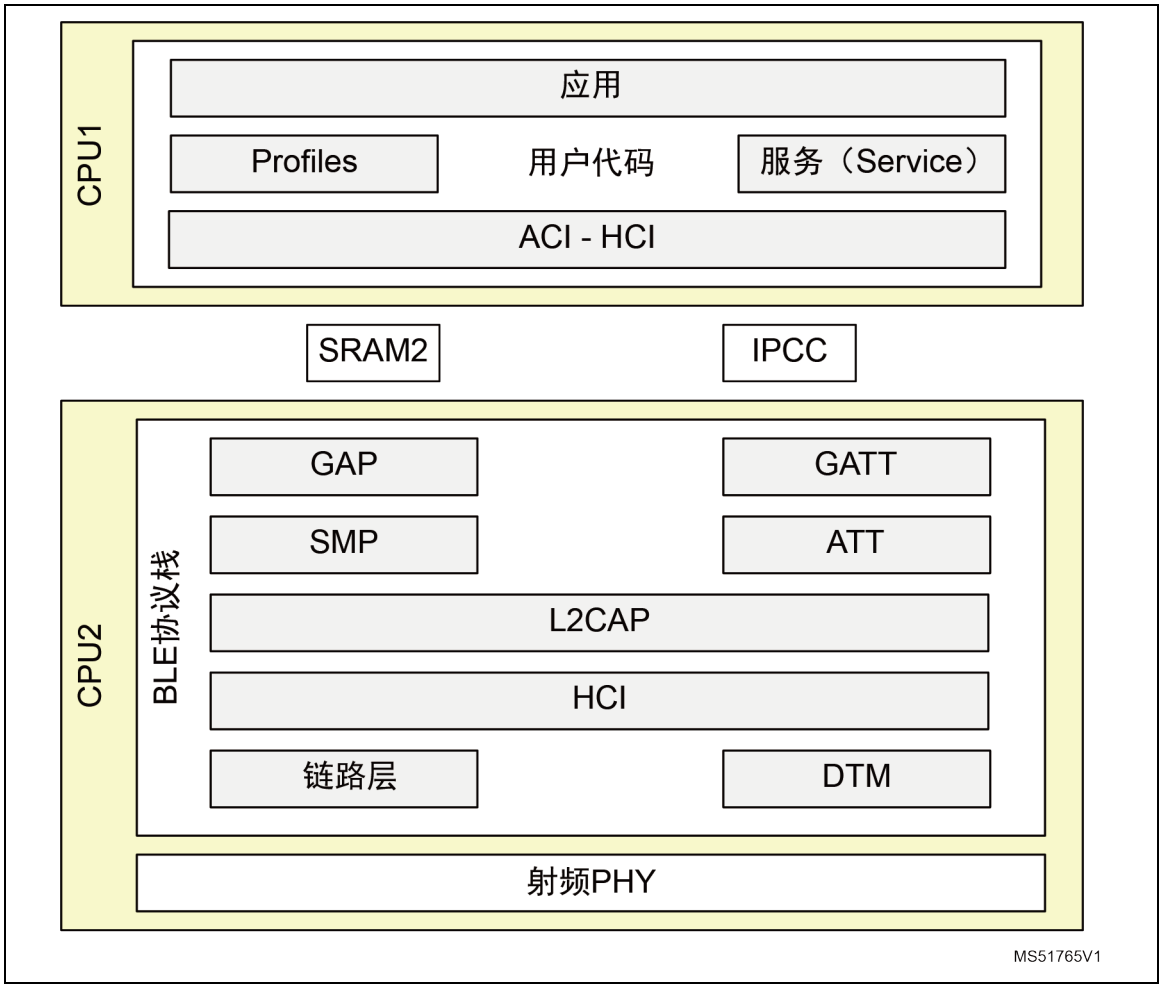
BLE 外设包含了 CPU2 处理器，其中包含用于处理链路层直到 GAP 及 GAP 层的射频协议栈。此外，它还包含了 2.4 GHz 射频部分。

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射频 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
- 邮箱（mailbox）/IPCC，用于连接 BLE 射频协议栈（LL、GAP 和 GATT）

图 3. BLE 应用程序和射频固件架构



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邮箱（mailbox），如*第 13.2 节：邮箱（Mailbox）接口所述*。

邮箱（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邮箱（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邮箱（mailbox）接收到的事件。除了通过邮箱（mailbox）完成 BLE 主机协议栈自适应，用户还必须在可以释放异步数据包时通知邮箱（mailbox）驱动程序。

BLE 主机协议栈不处理系统通道。用户必须实现一个自定义传输层来构建发送到邮箱驱动程序的系统命令缓冲区，并管理从邮箱接收到的事件（包括向邮箱驱动程序释放异步缓冲区的通知），或者也可以使用邮箱扩展驱动程序（如*第 13.3 节：邮箱接口 - 扩展中所描述*）在传输层之上提供一个接口，负责构建系统命令缓冲区和管理系统异步事件。

BLE_TransparentMode 项目可用作使用邮箱（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”）。

内嵌两种射频协议栈（BLE 和 Thread）的 stm32wb5x_BLE_Thread_fw CPU2 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 射频协议栈后从 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 主要原理

- 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“邮箱（mailbox）”进行通信

标准 STM32Cube 交付包包含诸如以下 STM32WB 资源：

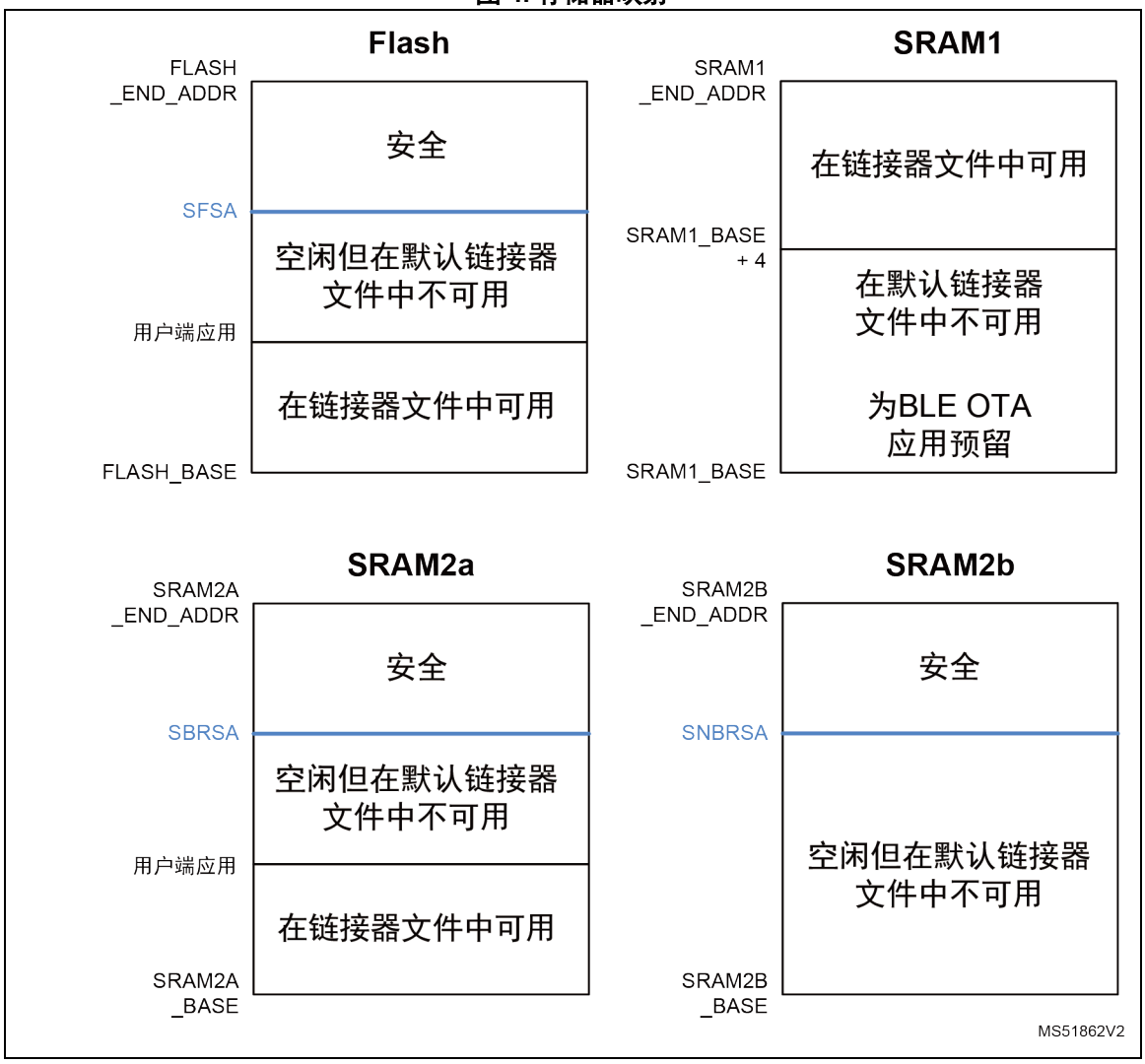
- 用于访问硬件寄存器的 HAL/LL
- BSP
- 中间件（例如：FreeRTOS、USB 设备）。

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
- 定时器服务器，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



Flash、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：

- Flash 存储器的 SFSA
- SRAM2a 的 SBRSA（在待机时保留）
- SRAM2b 的 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 射频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 协处理器射频协议二进制文件的版本说明中获取。RF 射频协议栈的安装地址也是用户 Flash 存储器区域的边界地址。

确保 CPU2 域中包含一些余量，以支持产品生命周期中的更新。

Flash 存储器的边界粒度为 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：

- Flash 存储器：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。当用户应用程序必须进入待机模式时，它必须首先停止所有 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

AES2 保留给 CPU2，CPU1 绝对不能使用/访问它。

AES1 被保留给 CPU1，CPU2 永远不会使用/访问它。CPU2 访问 AES1 的唯一情况是 CPU1 请求在客户密钥存储区域上写入用户密钥。相关内容在[6]中介绍。

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

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 RCC_EXTCFGR RCC_CFGR RCC_SMPSCR
Sem4	停止模式实现的时钟控制机制。
Sem5	RCC_CRRCR RCC_CCIPR
Sem6	被 CPU1 用来防止 CPU2 在 Flash 存储器中写入/擦除数据。
Sem7	被 CPU2 用来防止 CPU1 在 Flash 存储器中写入/擦除数据。
Sem8	确保当 CPU1 正在读取 SRAM2 中的 Thread 持久性数据时，CPU2 不会更新这些数据。
Sem9	确保当 CPU1 正在读取 SRAM2 中的 BLE 持久性数据时，CPU2 不会更新这些数据。

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 上的射频固件更新，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 Sem 0。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 Flash 存储器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 Flash 存储器。

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。该算法的详细信息请见*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。该算法的详细信息请见*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。该实现利用了这样一个事实：当从临界区调用 WFI 时，MCU 被中断请求唤醒，但它不会执行 ISR，而是继续执行 WFI 之后的下一条指令。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

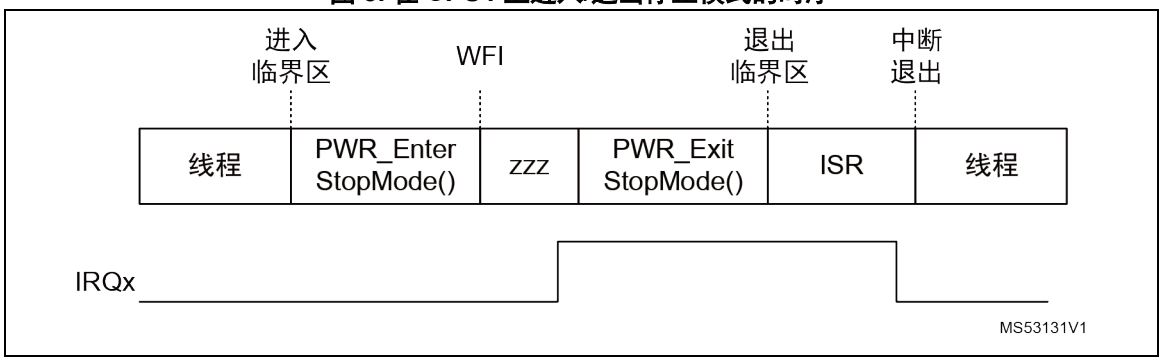


图 6. 在 CPU1 上进入停止模式的算法

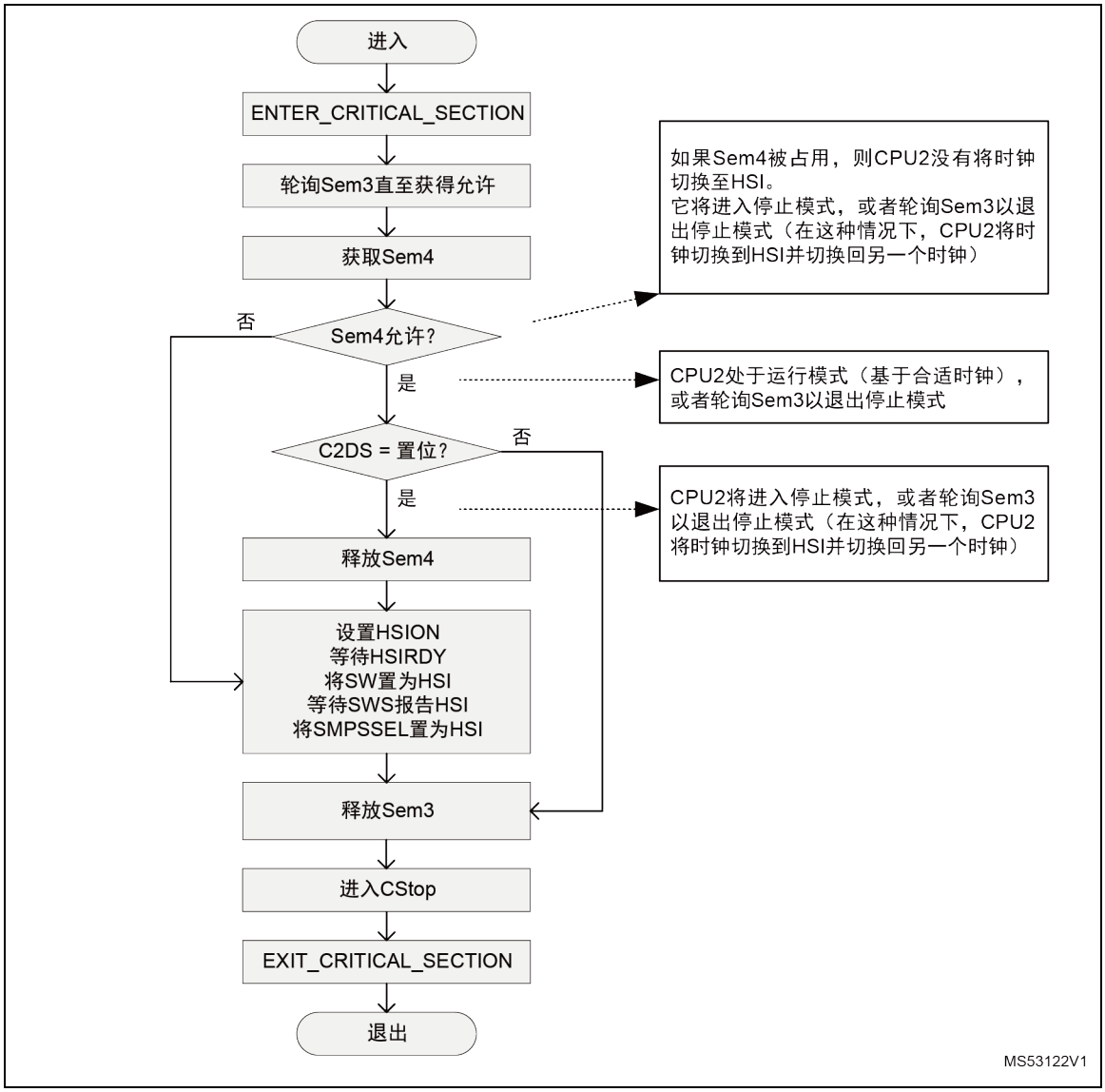
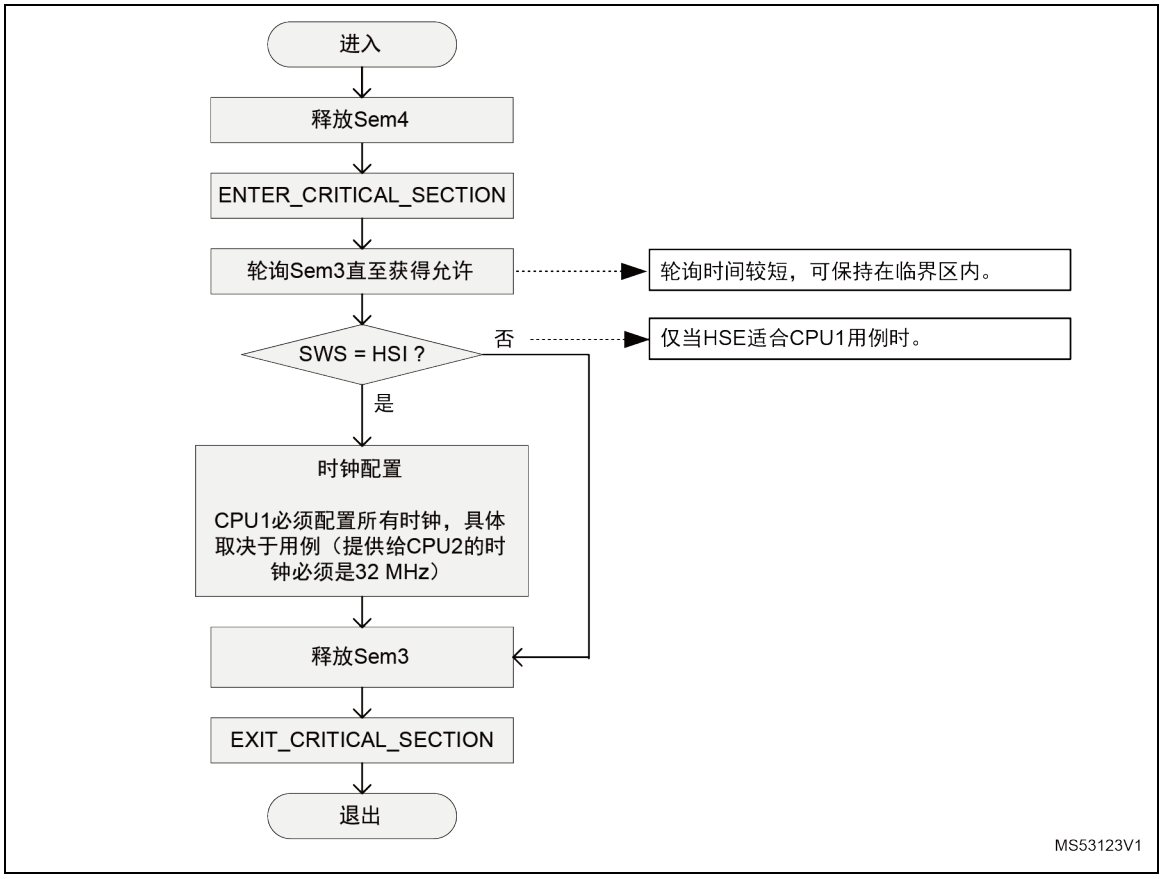


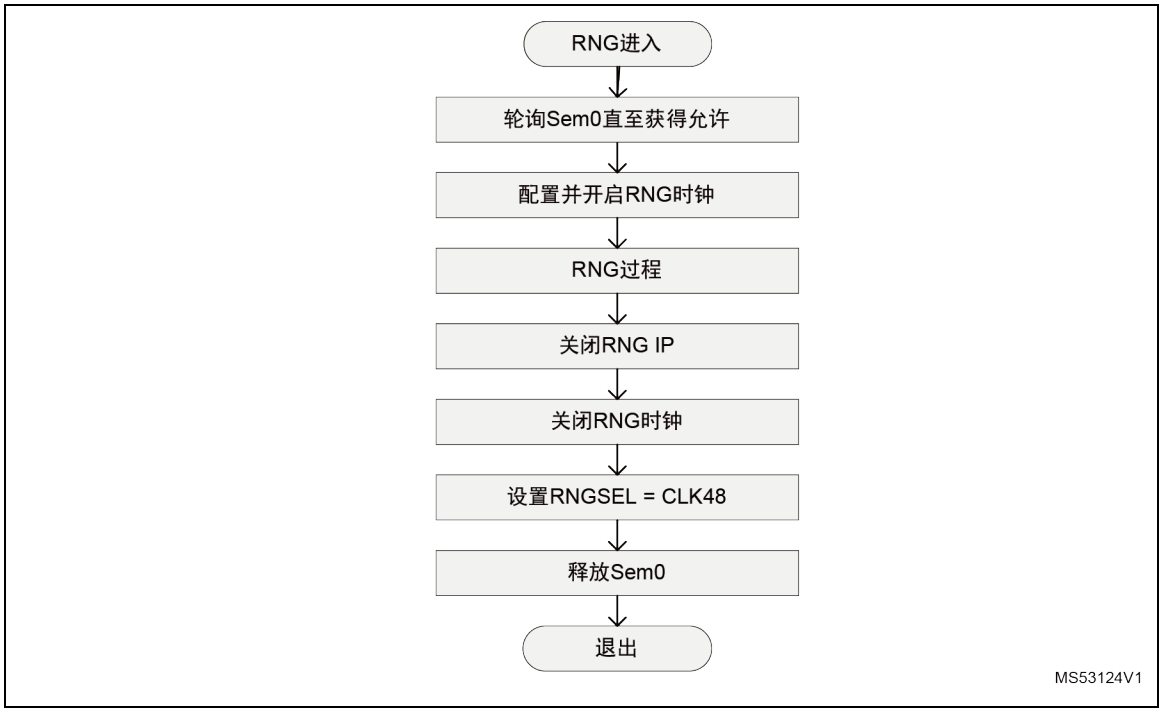
图 7. 在 CPU1 上退出停止模式的算法



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。

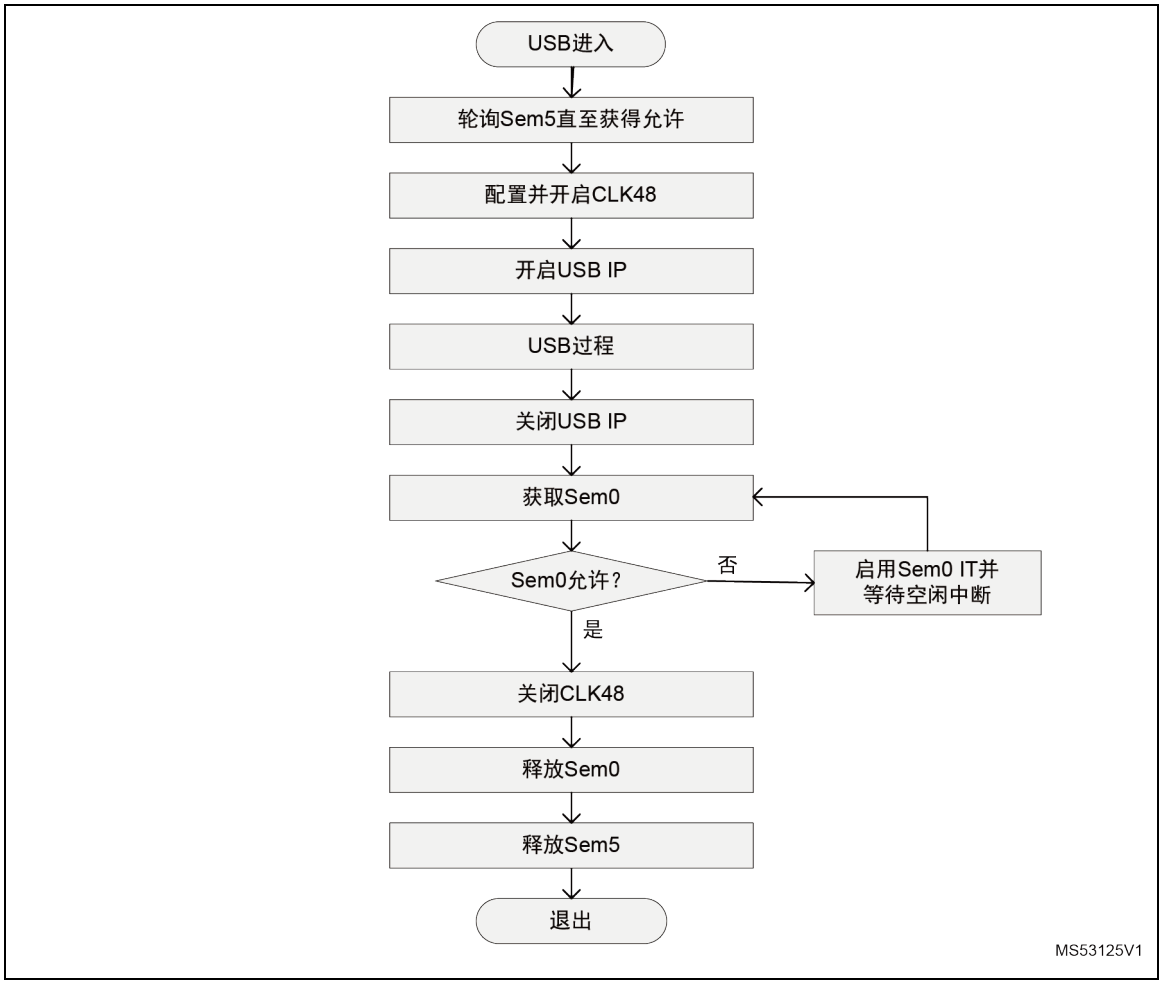
为了避免与 CPU2 形成竞争状态，当 CPU1 需要关闭时钟时，它必须总是先获得 Sem0（即使不使用 RNG IP）。该机制显示在 BLE P-NUCLEO-WB55.USB Dongle 示例（参见*第 7.1.3 节：配置*，以及*图 9*）。这不会影响 CPU2。

图 8. 在 CPU1 上使用 RNG 的算法



注意： 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 IP 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 Flash 存储器中写入或擦除数据。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 Flash 存储器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 Flash 存储器操作的影响。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* 中所示。

Sem8 用于确保当 CPU1 读取 SRAM2 中的 BLE 持久性数据时，CPU2 不会更新这些数据。

CPU2 可以配置为将 BLE 持久性数据存储在 CPU2 的内部 NVM 存储中，也可以存储在用户应用程序提供的 SRAM2 缓冲区中。当请求 CPU2 在 SRAM2 中存储持久数据时，可以使用系统命令 *SHCI_C2_Config()*进行这样的配置，以便在需要时将数据写入该缓冲区。为了从 SRAM2 缓冲区中读取与 CPU1 一致的数据，流程必须：

1. CPU1 获得 Sem8
2. CPU1 从 SRAM2 读取所有持久性数据（大多数时候，目标是将这些数据写入由 CPU1 管理的 NVM 中）
3. CPU1 释放 Sem8

对于该信号量可以保存多长时间，没有时间限制。

Sem9 用于确保当 CPU1 读取 SRAM2 中的 Thread 持久性数据时，CPU2 不会更新这些数据。

CPU2 可以配置为将 Thread 持久性数据存储在 CPU2 的内部 NVM 存储中，也可以存储在用户应用程序提供的 SRAM2 缓冲区中。当请求 CPU2 在 SRAM2 中存储持久数据时，可以使用系统命令 *SHCI_C2_Config()*进行这样的配置，以便在需要时将数据写入该缓冲区。为了从 SRAM2 缓冲区中读取与 CPU1 一致的数据，流程必须：

1. CPU1 获得 Sem9
2. CPU1 从 SRAM2 读取所有持久性数据（大多数时候，目标是将这些数据写入由 CPU1 管理的 NVM 中）
3. CPU1 释放 Sem9

对于该信号量可以保存多长时间，没有时间限制。

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
- 请求要执行的函数
- 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，它提供了一个钩子，应用程序可以进入低功耗模式，或者执行一些其他代码。

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

函数	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。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。
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() 相关联的掩码独立于与 void UTIL_SEQ_Run(UTIL_SEQ_bm_t mask_bm)相关联的掩码。

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()完成信号置位前调用。

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 定时器服务器

定时器服务器具有以下特性：

- 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

定时器服务器提供多个共享 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以相同超时时间自动重启。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器中的时隙。

定时器服务器可与日历同时使用。

4.5.1 实现方法

为了使用定时器服务器，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器。
- 通过 HW_TS_Init()初始化定时器服务器。
- 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

函数	说明
void HW_TS_Init(HW_TS_InitMode_t TimerInitMode, RTC_HandleTypeDef *hrtc);	初始化定时器服务器。
HW_TS_ReturnStatus_t HW_TS_Create(uint32_t TimerProcessID, uint8_t *pTimerId, HW_TS_Mode_t TimerMode, 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(uint8_t TimerID, 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器处理函数。
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(uint32_t TimerProcessID, uint8_t TimerID, 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器更新了下一次中断前的时间片数。

4.5.3 具体接口与行为

定时器服务器提供虚拟定时器，该定时器在系统处于低功耗模式一直到待机模式期间运行。

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）初始化定时器服务器。

TimerInitMode 选择定时器服务器启动模式。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 以使定时器服务器上下文不复位。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

这是定时器服务器返回给调用方的 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，它就没有效果。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器中，以便可以使用不同的值重新启动相同的定时器。

void HW_TS_Start(uint8_t TimerID, uint32_t timeout_ticks)

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器中停止它，然后使用新的 timeout_ticks 值重新启动它。

void HW_TS_Delete(uint8_t TimerID)

从定时器服务器中删除 TimerID。可以将 TimerID 分配给新的虚拟定时器。该 API 可以在正在运行的 TimerID 上调用。在这种情况下，首先将其停止，然后删除。

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 返回在定时器服务器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。它将与 HW_TS_RTC_ReadLeftTicksToCount () API 一起使用。计数器可以在上次调用 HW_TS_RTC_ReadLeftTicksToCount()之后和进入低功耗模式之前被更新。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 ()。

不需要做什么来控制 CPU2 的低功耗模式，只要 CPU2 启动了，射频固件就会自行设置 CPU2 的最佳低功耗模式配置。

CPU2 的低功耗模式选择必须在 CPU2 启动之前被写入 SHUTDOWN，以覆盖应用程序启动而 CPU2 没有启动的情况。在这种情况下，必须重写低功耗模式选择的复位值，以允许设备进入低功耗模式。否则，由于 CPU2 低功耗模式选择的复位值，在 CPU2 启动之前不可能降至停止 0 模式以下。

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

函数	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 模式时调用。

4.7 Flash 存储器管理

STM32WB 的 CPU1 和 CPU2 共享一个单存储区。在写入或擦除 Flash 存储器时，无法从 Flash 存储器中获取指令。

当 CPU 执行 Flash 存储器中的代码时，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 Flash 存储器中的数据）。

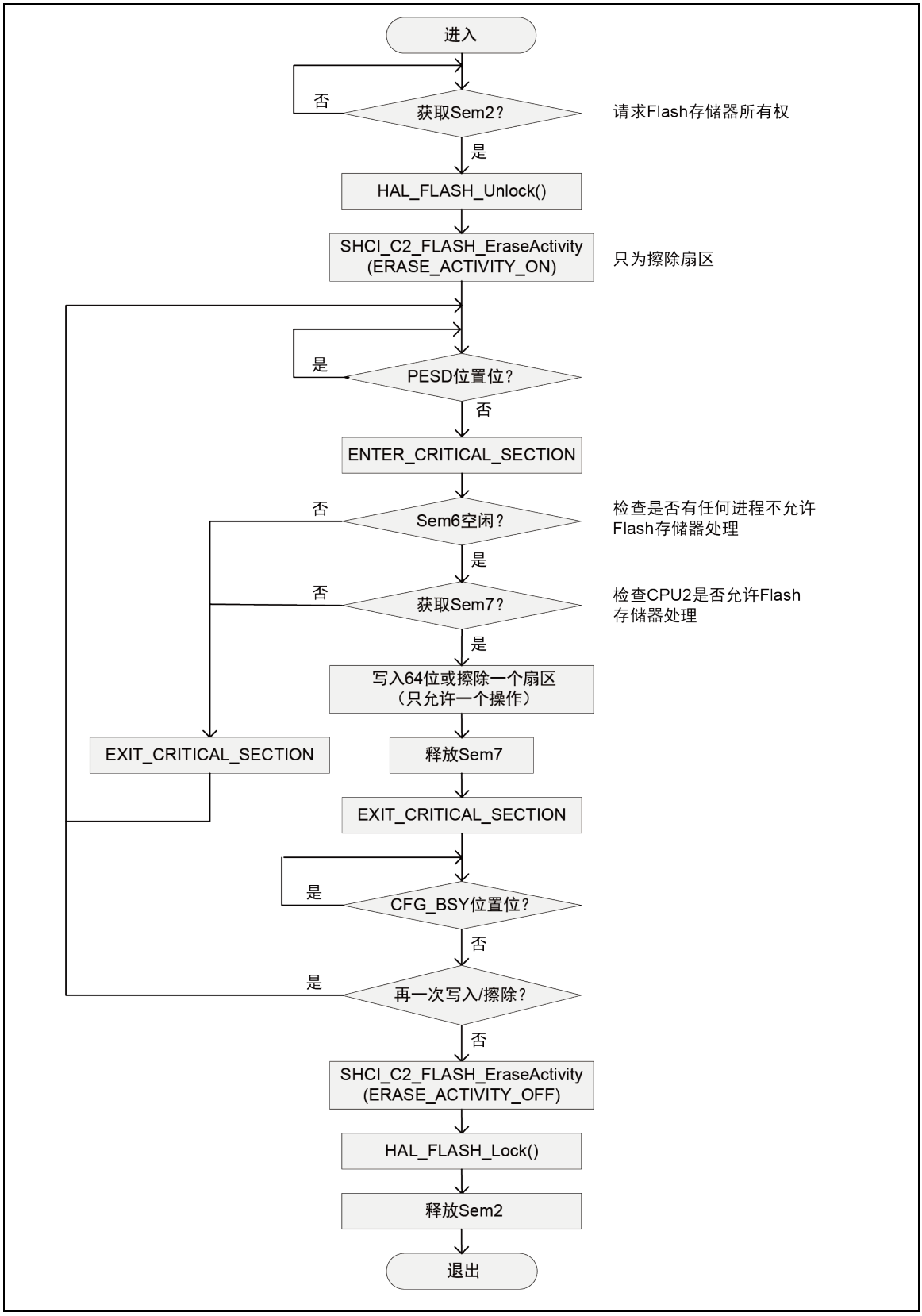
4.7.1 CPU2 时序保护

出于安全原因，禁止 CPU2 执行任何来自 SRAM 的代码。为了保护 CPU2 的时序，它使用 Sem7 来启用或禁用来自 CPU1 的 Flash 存储器操作请求。

CPU1 上的应用必须实现*图 10* 所示的算法才能写入或擦除 Flash 存储器，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）：

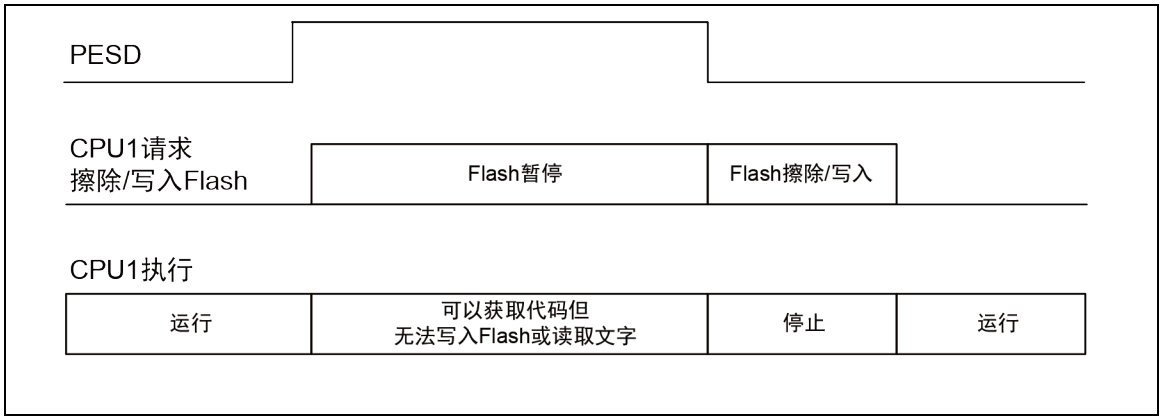
- 在对 Flash 存储器进行任何访问前获取 Sem2，并在不再需要它来访问 IP 时将其释放
- 当用户驱动程序需要擦除扇区时，必须首先发送命令 SHCI_C2_FLASH_EraseActivity(ERASE_ACTIVITY_ON)。在擦除所有相关扇区后，必须发送命令 SHCI_C2_FLASH_EraseActivity(ERASE_ACTIVITY_OFF)，参见*第 4.7.1 节*。
- 当 CPU2 时序保护使用 PESD 位机制（这是默认情况，参见*第 4.7.1 节*）时，FLASH 驱动程序必须轮询 FLASH_SR 寄存器中的 CFGBSY 位或回读存储器，直至值为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

图 10. 在 Flash 存储器中写入/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 Flash 存储器操作与 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 Flash 存储器操作（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 Flash 存储器操作。

当 Sem6 被占用时，表示 CPU2 已经在执行 Flash 存储器操作的过程中（即将开始或刚刚结束）。当需要防止 CPU2 请求任何 Flash 存储器操作时，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 Flash 存储器管理之间的冲突

即使 CPU1 不使用 Sem6 防止 CPU2 启动 Flash 存储器操作，仍然有一些用例中的 CPU2 无法启动擦除操作。

始终可以执行 Flash 存储器写操作，但在 CPU2 上的 NVM 已满时，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 Flash 存储器擦除操作以擦除 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 Flash 存储器擦除操作，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 Flash 存储器擦除操作。
- BLE 连接：连接间隔必须长于 25 ms + 数据包长度，才能执行 Flash 存储器擦除操作。
- 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 Flash 存储器中写入数据：

- 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。信号分配到特定 GPIO 是完全可以从 CPU1 端配置的，BLE IP Core 信号除外，因为它们是由 HW 驱动的。因此，只有在未被应用使用时，才需要启用 BLE IP Core GPIO。每个应用程序的完整配置都在\Core\Src 中的 app_debug.c 文件中完成。

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 - 进入时置位/退出时复位*/

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 用于通过邮箱（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 上运行的是什么射频协议栈，FreeRTOS 低功耗模式在 CPU1 上为所有射频应用程序共享相同的实现。

HAL tick 被映射到 TIM17，以避免与预留给 FreeRTOS 的 systick 冲突。\Applications\BLE\BLE_HeartRateFreeRTOS\Core\Src 中的文件 stm32wbxx_hal_timebase_tim.c 实现 HAL 函数：

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

TIM17 用户中断处理程序 HAL_TIM_PeriodElapsedCallback()在 main.c 中实现，用于增加 HAL 使用的 tick。可以自定义此实现，以选择另一个定时器。

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

当前实现是使用运行在 RTC 上的定时器服务器。定时器选择可以通过重新实现以下函数来更改：

- 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()：**当从中间件调用 hci_notify_asynch_evt()时，此函数必须在后台调用。hci_user_evt_proc()不能在 hci_notify_asynch_evt()里面调用，因为它可以从 IPCC 中断上下文调用。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。

4.10 设备信息表

一旦从 CPU2 接收到 System Ready Event（系统就绪事件），就可以从 SRAM2A 读取设备信息表（DIT）。

uint32_t 安全启动版本	安全启动版本
uint32_t FUS 版本	FUS 版本
uint32_t FUS 内存大小	FUS 需要的内存
uint32_t FusInfo	已保留 - 设为 0
uint32_t 射频固件版本	射频固件版本
uint32_t 射频固件内存大小	射频固件需要的内存
uint32_t InfoStack	射频固件信息

当由 FUS（参见*[6]*）或射频固件填充时，DIT 具有不同的映射。

系统命令 SHCI_GetWirelessFwInfo()可以解码这两个 DIT 映射。

DIT 地址可以在 SRAM2 的开始处找到（+ IPCCDBA 偏移量 – 用户选项字节）。除非用户修改，否则 IPCCDBA 总是被设置为 0，因此 DIT 地址可以在 SRAM2A 的第一个地址处找到。

图 12. 版本和内存信息的格式

版本	主要版本								次要版本								子版本								分支				构建			
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

存储器容量	SRAM2b（1 KB扇区的数量）								SRAM2a（1 KB扇区的数量）								保留								Flash存储器（4 KB扇区的数量）							
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

对于所有正式版本，Build 信息总是不同于 0。

Branch 信息仅供内部使用。

仅使用 InfoStack LSB，它提供 CPU2 上运行的射频固件的信息，即：

- INFO_STACK_TYPE_BLE_STANDARD: 0x01
- INFO_STACK_TYPE_BLE_HCI: 0x02
- INFO_STACK_TYPE_BLE_LIGHT: 0x03
- INFO_STACK_TYPE_BLE_BEACON: 0x04
- INFO_STACK_TYPE_THREAD_FTD: 0x10
- INFO_STACK_TYPE_THREAD_MTD: 0x11
- INFO_STACK_TYPE_ZIGBEE_FFD: 0x30
- INFO_STACK_TYPE_ZIGBEE_RFD: 0x31
- INFO_STACK_TYPE_MAC: 0x40
- INFO_STACK_TYPE_BLE_THREAD_FTD_STATIC: 0x50
- INFO_STACK_TYPE_BLE_THREAD_FTD_DYAMIC: 0x51
- INFO_STACK_TYPE_802154_LLD_TESTS：0x60
- INFO_STACK_TYPE_802154_PHY_VALID：0x61
- INFO_STACK_TYPE_BLE_PHY_VALID: 0x62
- INFO_STACK_TYPE_BLE_LLD_TESTS: 0x63
- INFO_STACK_TYPE_BLE_RLV: 0x64
- INFO_STACK_TYPE_802154_RLV：0x65
- INFO_STACK_TYPE_BLE_ZIGBEE_FFD_STATIC: 0x70
- INFO_STACK_TYPE_BLE_ZIGBEE_RFD_STATIC: 0x71
- INFO_STACK_TYPE_BLE_ZIGBEE_FFD_DYNAMIC: 0x78
- INFO_STACK_TYPE_BLE_ZIGBEE_RFD_DYNAMIC: 0x79
- INFO_STACK_TYPE_RLV: 0x80

4.11 ECCD 错误管理

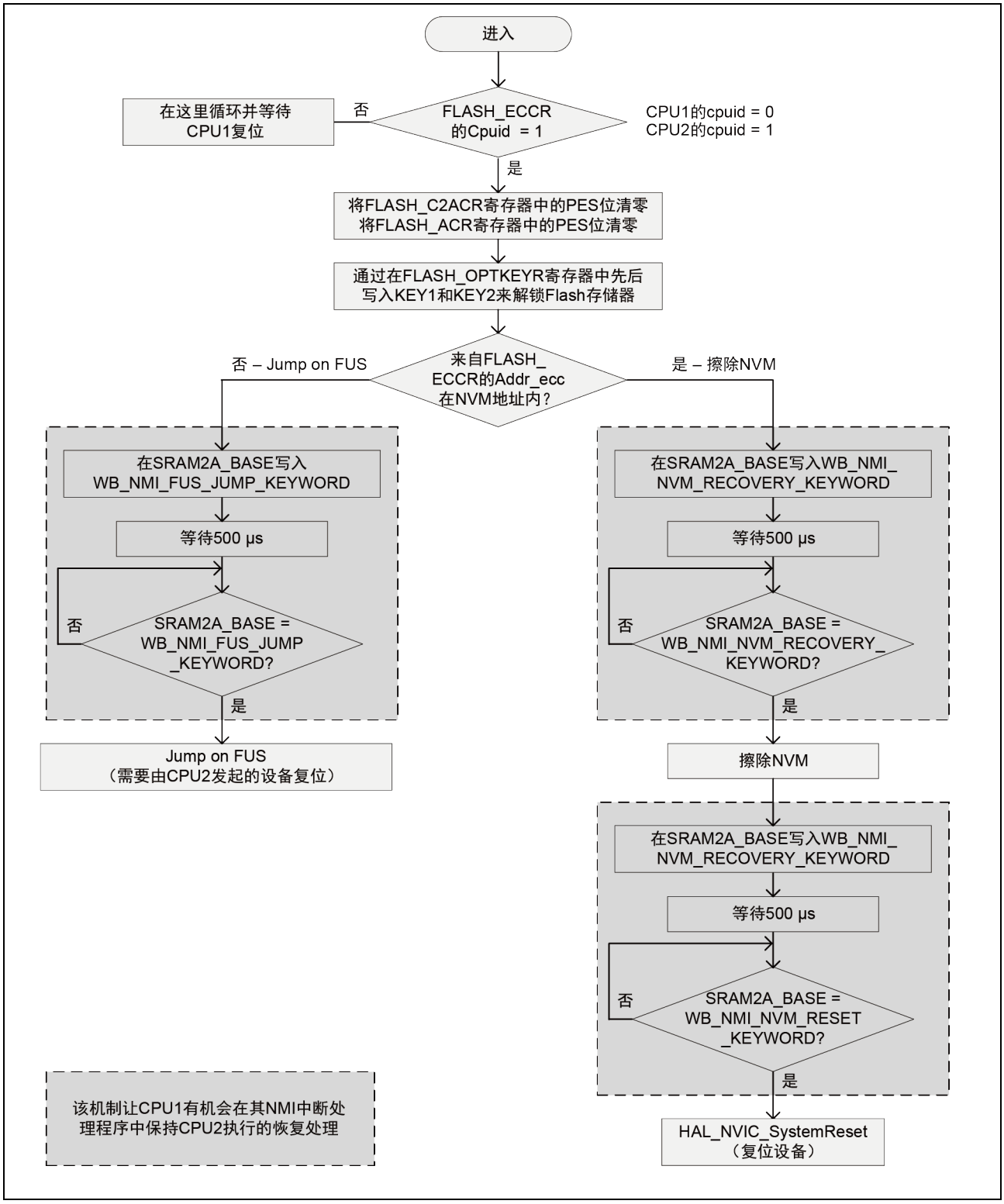
可以因为一个 ECCDFlash 存储器错误（无论是在 NVM 数据段，还是在代码数据段）而产生一个 NMI 中断。

当从 NVM 数据段生成 ECCD 时，CPU2 可以擦除 NVM 以消除错误。

当从代码数据段生成 ECCD 时，CPU2 必须在 FUS 上重新启动以请求安装新的射频固件。

当出现 ECCD 错误时，两个 CPU 都会产生 NMI。*图 13*（WB_NMI_NVM_RECOVERY_KEYWORD = 0xAFB449C9，WB_NMI_RESET_KEYWORD = 0x8518C6F2、以及 WB_NMI_FUS_JUMP_KEYWORD = 0x7E3FF448）中所示的算法描述了 CPU2 管理 ECCD 错误的方式，以及允许 CPU1 保持 CPU2 错误处理的机制。

图 13. 中断处理程序中的 ECC 管理



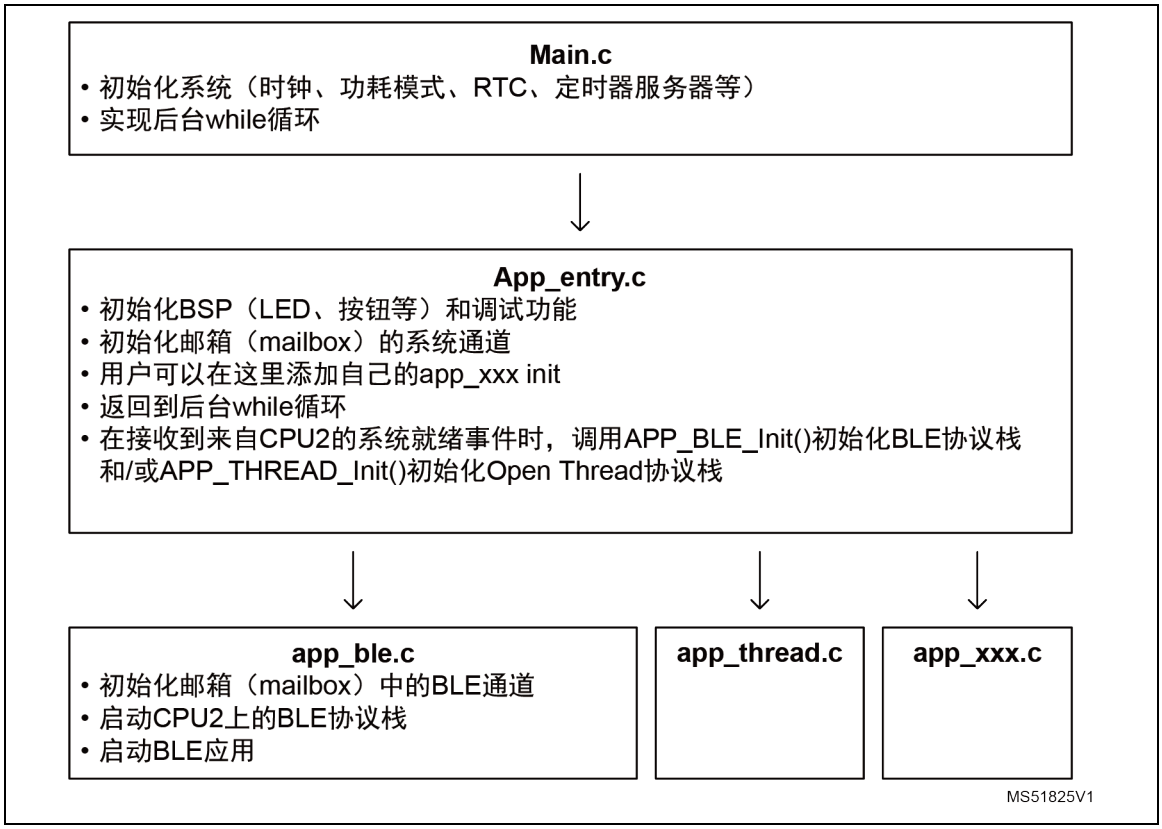
5 系统初始化

5.1 一般概念

所有应用都从三组文件开始（参见*图 14*）：

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 32 MHz）
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 14. 系统初始化



5.2 CPU2 启动

在启动时，CPU2 运行最小的初始化集合，以使所有受支持的系统特性变得可用。在初始化结束时，CPU2 通过系统通道向 CPU1 报告 System ready（系统就绪）事件。此时，CPU1 可以向 CPU2 发送任何系统命令，包括管理安全 CPU2Flash 存储器中的客户密钥存储（CKS）中的密钥所需的全部命令。

在 CPU2 启动阶段，会涉及一些共享资源：

- RNG 外设填充一个池（当操作需要一些随机数时使用），这样当 RNG IP 已经被 CPU1 使用时，这些随机数可以立即可用。RNG 外设访问需要获得 Sem0。此外，CPU2 会尝试获得 Sem5。如果该尝试成功，它将打开 HSI48 振荡器，并将 RNG IP 的 48 MHz 时钟选择配置为 HSI48。这需要假设将 RCC 配置为 RNG IP 提供 48 MHz 时钟，而不是 LSI 或 LSE。在后一种情况下，无论如何都要完成前一步（即使不相关），且 RNG 对所选的时钟进行操作。当 Sem5 繁忙时，CPU2 不会改变 RNG 时钟配置中的任何内容，而是使用当前配置。当射频堆栈启动后，CPU2 会关闭 HSI48 振荡器。这需要 Sem5 可用，并且只尝试一次获取 Sem5。

- 检查 NVM 一致性，如果已损坏，则重新格式化。该操作需要擦除 Flash 存储器扇区。启动时对 NVM 的访问符合在内存上启动任意进程的通用规则。它首先需要获取 Sem2 以取得 Flash 存储器的所有权，CPU1 有能力使用 Sem6 进行任何操作。

所有这些步骤都必须在发送系统就绪事件之前完成，因此当 CPU2 需要任何信号量时，它会进行轮询，直到该信号量空闲为止。

CPU2 可以执行其启动序列，直到它报告系统就绪事件，无需外部 HSE 或 LSE 振荡器。

一旦 CPU2 报告了系统就绪事件，所有系统命令都会得到支持，无需任何外部 HSE 或 LSE 振荡器。

射频协议栈启动时，需要 HSE 和 LSE 振荡器。如果功耗不是问题，则可以去掉外部 LSE 振荡器，并将设备配置为使用 HSE (32 MHz) / 1024 (= 32.768 kHz)。

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
- 配置发射功率
  - 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。 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。 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。设备的定向可连接模式仅持续 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_discover y_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_discover y_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”。

表 7. GAP 中央设备 API（接上页）

API	说明
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。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_connecti on_establish_proc()	启动通用连接流程。在扫描者过滤策略设置为“接受所有广播数据包”的情况下，设备启用控制器扫描，并使用事件回调 hci_le_advertising_report_event()将扫描结果中的所有设备发送至上层。
aci_gap_start_selective_connec tion_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，并在扫描者过滤策略设置为“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，将基于 GATT 客户端-服务器架构进行应用数据交换。

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。 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，以通过使用其 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。 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。 通过 aci_att_read_by_type_resp_event()事件回调提供流程响应。

表 8. GATT 客户端 API（接上页）

API	说明
aci_gatt_disc_all_char_of_service()	启动发现给定服务的所有特征的 GATT 流程。 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。 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。 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. 频带外
3. 密钥输入（Passkey entry）
4. Bluetooth 4.2 以后通过数字比较（仅安全连接）

确定安全密钥的计算方法：

- 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。 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。 当设备绑定后，密钥存储在非易失性存储区。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。 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，则指令返回 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。

表 10. 安全信息命令（接上页）

指令	说明
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，会生成 HCI_L2CAP_CONNECTION_UPDATE_RESP_EVENT 事件。

6.10 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。

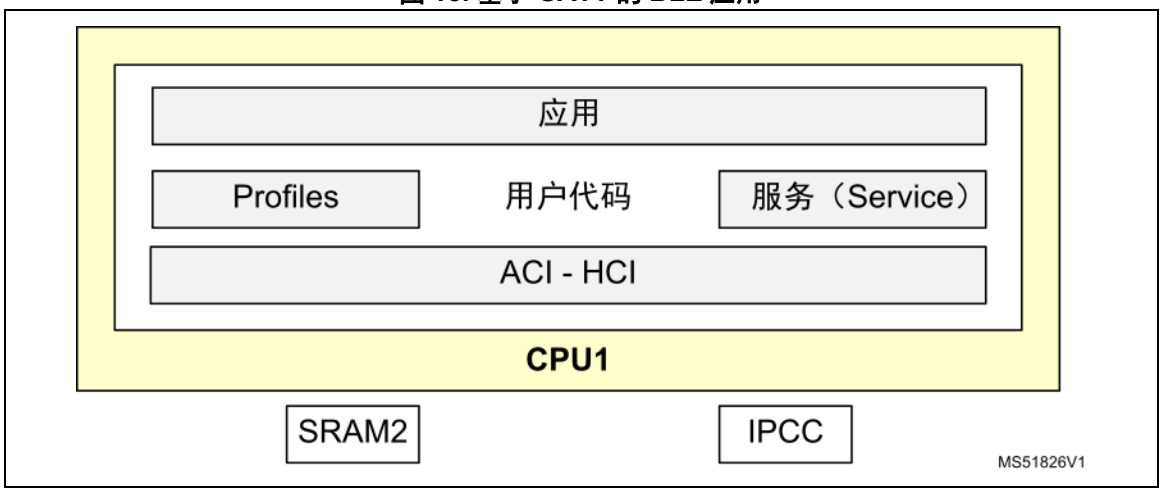
所有 GATT 客户端程序都必须在 aci_gatt_proc_complete_event()上以成功或发生错误的方式完成，这样 GATT 客户端才能启动新的流程。

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：

- 意法半导体特定应用
  - 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 15. 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。

STM32WB 依据蓝牙核心规范 v5.0 [第 6 卷 F 部分]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

函数	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]*）包含发射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 透传模式应用原理

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。

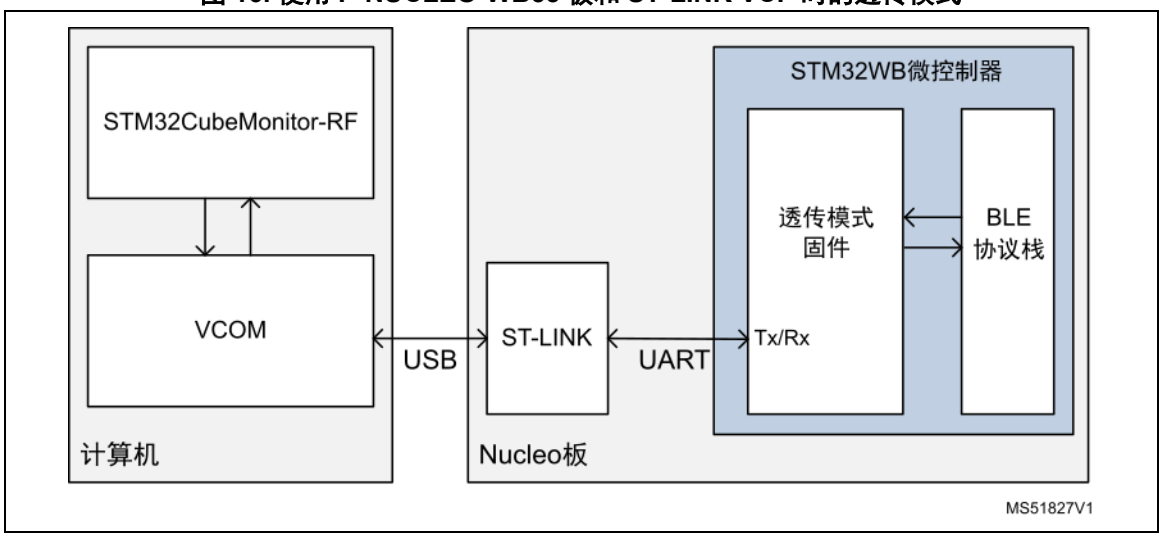
使用 app_conf.h 完成 UART 接口和配置选择：

#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。

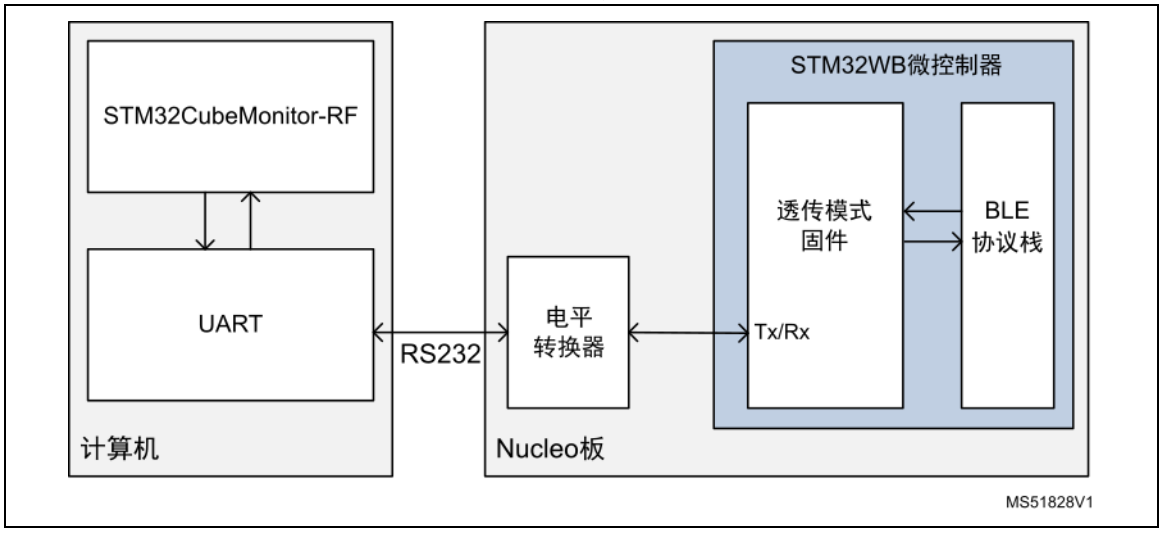
图 16. 使用 P-NUCLEO-WB55 板和 ST-LINK VCP 时的透传模式



P-NUCLEO-WB55 加密狗板没有提供 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 STM3232WB 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 17. 使用 P-NUCLEO-WB55 板和电平转换器时的透传模式

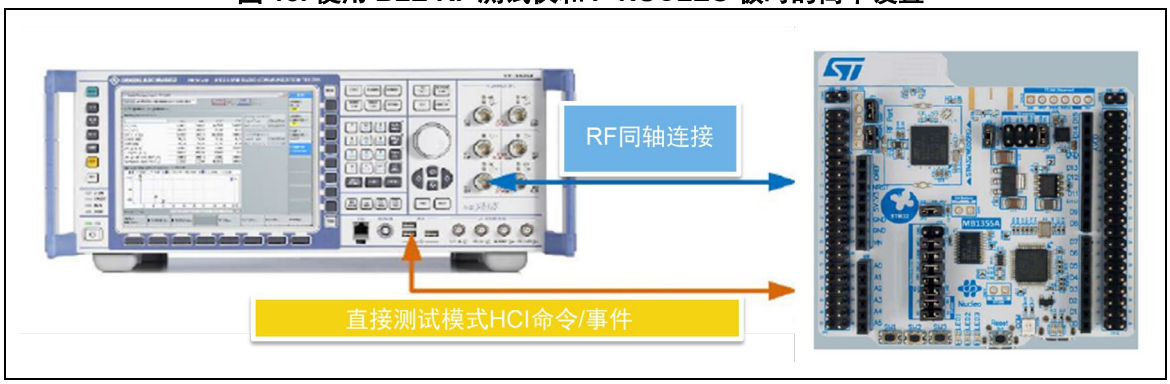


7.1.4 RF 认证 - 应用实现

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 18* 展示了一个简单的设置。

图 18. 使用 BLE RF 测试仪和 P-NUCLEO 板时的简单设置

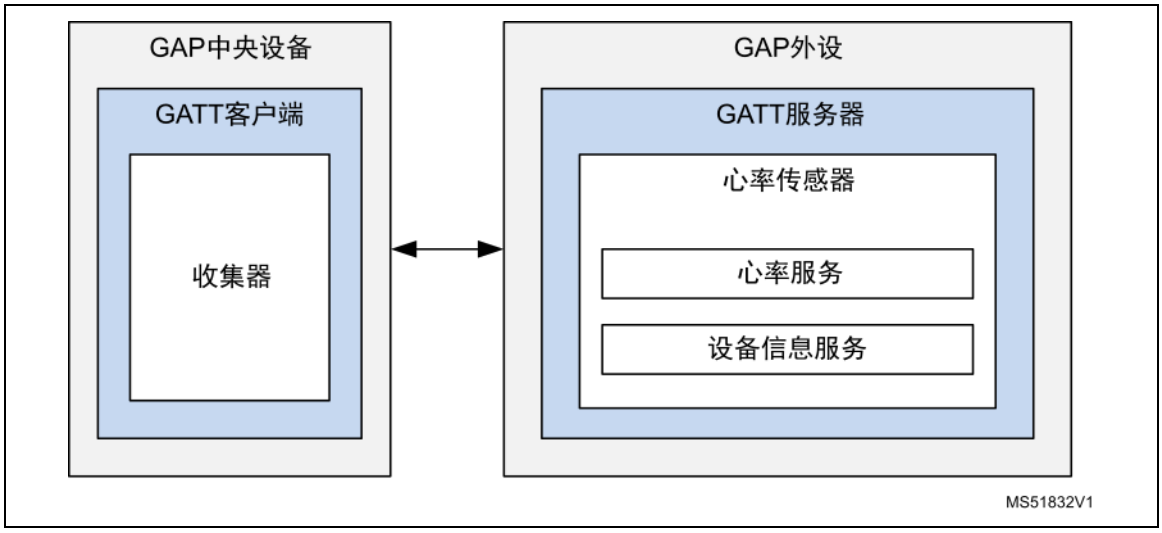


7.2 心率传感器应用

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 19. 心率配置文件结构

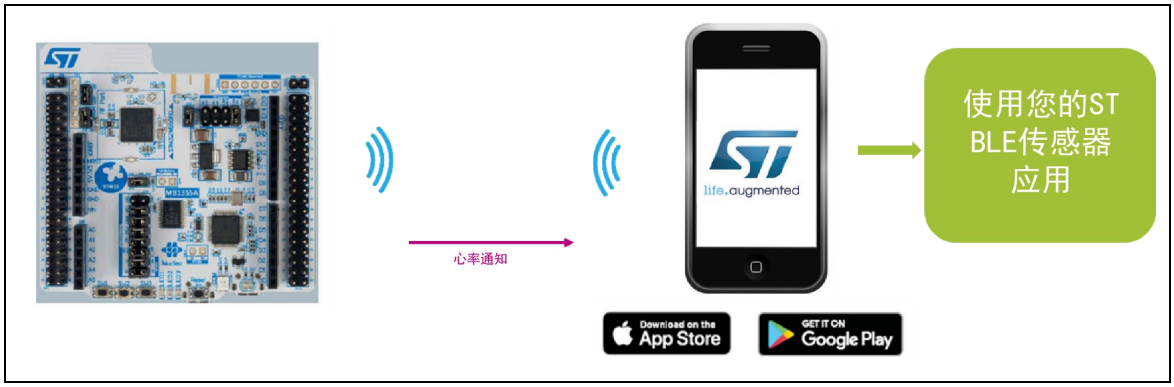


STM32WBCube_FW_WB_V1.0.0 版附带心率传感器的示例。

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 20* 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。

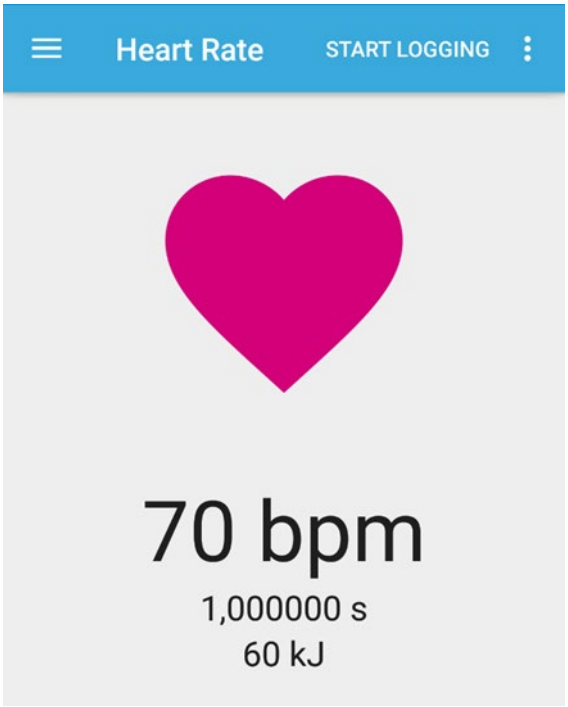
图 20. 使用 BLE RF 测试仪和 P-NUCLEO 板时的简单设置



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 sensor 移动应用连接到您的心率应用

图 21. 智能手机 - ST BLE sensor 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

心率服务函数：
Middlewares\STM32_WPAN\ble\core\Src\blesvc\hrs.c

表 14. 心率服务函数

函数	说明
Service Init - HRS_Init()	– 将心率事件句柄注册到服务控制器 – 初始化服务 UUID
aci_gatt_add_serv	– 添加心率服务作为主要服务 – 初始化心率测量特征
aci_gatt_add_char	– 添加心率特征 – 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 – 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 – 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 – 心率控制点特征值 – 复位能量消耗命令，然后： 发送正常状态的 aci_gatt_write_response()。通知 HRS 应用程序复位损耗的能量或发送带有错误的 aci_gatt_write_response()。
EVT_BLUE_GATT_ATTRIBUTE_MODIFIED	– 心率测量特征描述值 – 启用或禁用通知 – 将测量通知通知 HRS 应用

服务实现的目的是将心率服务和选定的特征注册到 BLE 协议栈 GATT 数据库中。

```
/**
  * @brief      心率服务初始化
  * @param      无
  * @retval     无
*/
void HRS_Init(void)
{
REGISTER HEART RATE EVENT HANDLER
? SVCCTL_RegisterSvcHandler(HeartRate_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）
添加射频重启请求特征（可选）
}

- 管理 HR 服务专用的 GATT 事件

```
/**
  * @brief 心率服务事件处理程序
  * @Param 事件：保存事件的缓存区地址
  * @retval Ack：返回 GATT 事件是否已被管理
*/
static SVCCTL_EvtAckStatus_t HeartRate_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
}
```

服务控制器函数：
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()

HR 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()
- 配置并启动广播：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

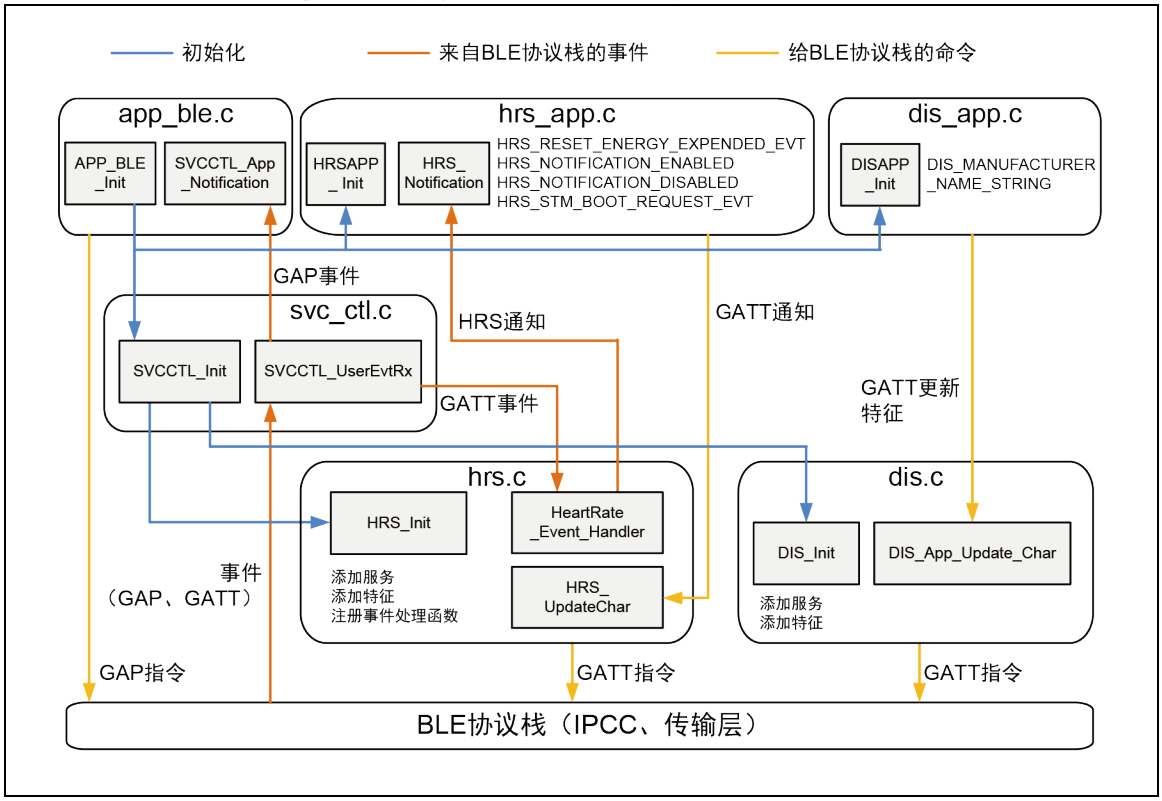
心率传感器应用控制：
Applications\BLE\BLE_HeartRate\STM32_WPAN\App\hrs_app.c

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

函数	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 22. 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. 符合蓝牙 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
FLAGS	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。

```
/* 待广播的本地名称*/
static const char local_name[] = { AD_TYPE_COMPLETE_LOCAL_NAME, 'P', '2', 'P', 'S', 'R', 'V', '1' };

/* 待广播的制造商数据和传统数据 */
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 启动 -MSB */
    0x00,
    0x00,
    0x00,
    0x00,
    0x00, /* BLE MAC 停止 */
};

/* 本地设备 BD 地址*/ const uint8_t *bd_addr;
bd_addr = SVCCTL_GetBdAddress();

/* BLE MAC 更新，用于广播制造商数据*/
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（从设备连接间隔最小值）；
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, /* 使用白名单 */
sizeof(local_name), (uint8_t*) local_name,
```

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

  /* 使用制造商特定信息更新广播数据*/
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 23* 和*图 24* 所示的演示结果。

图 23. P2P 服务器至客户端的演示

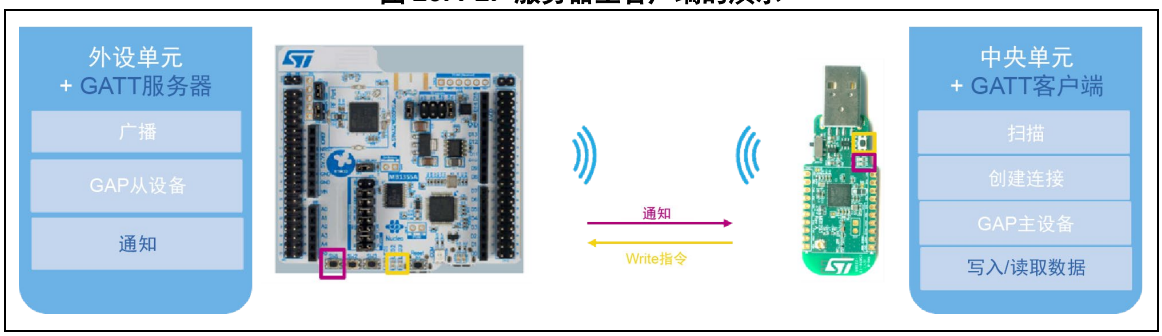
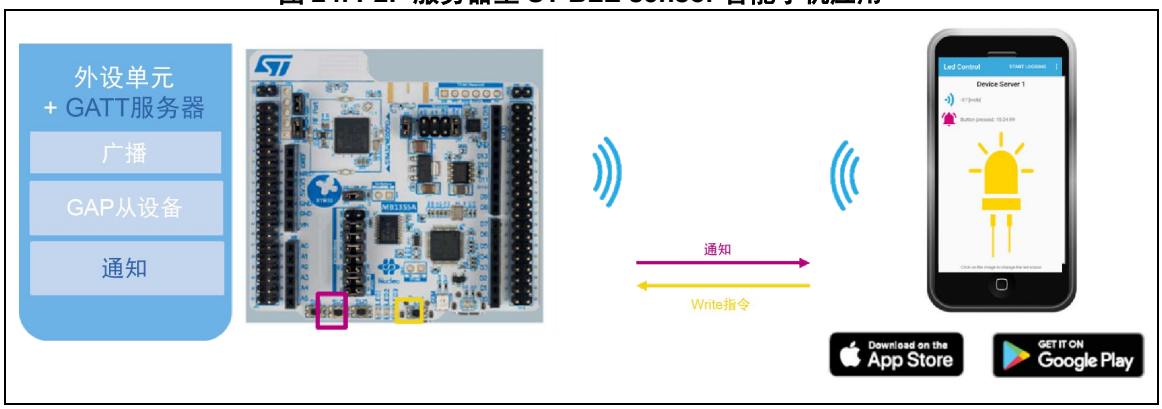


图 24. P2P 服务器至 ST BLE sensor 智能手机应用



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

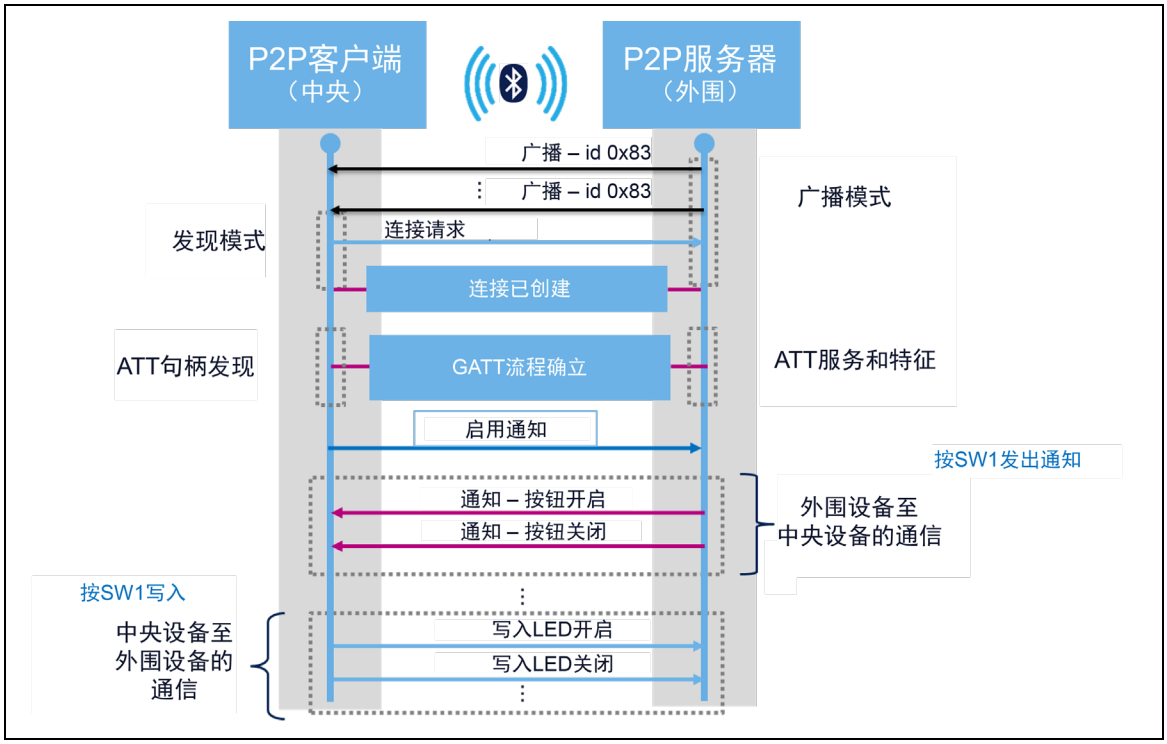
组	服务	特征	大小	模式	UUID
LED 按钮控制	P2P 服务	-	-	-	0000FE40-cc7a-482a-984a-7fed5b3e58f
	-	写	2	读/写	0000FE41-8e22-4541-9d4c-21edae82ed19
	-	通知	2	通知	0000FE42-8e22-4541-94dc-21edae82ed19

表 21. P2P 规范

写	八位组 LSB	0	1
	名称	设备选择	LED 控制
	值	– 0x01：P2P 服务器 1 – 0x02：P2P 服务器 2 – 0x0x：P2P 服务器 x – 0x00：全部	– 0x00 LED 关闭 – 0x01 LED 开启 – 0x02 Thread 切换
通知	八位组 LSB	0	1
	名称	设备选择	按钮
	值	– 0x01：P2P 服务器 1 – 0x02：P2P 服务器 2 – 0x0x：P2P 服务器 x	– 0x00 关闭 – 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 25* 描述了数据交换流程。

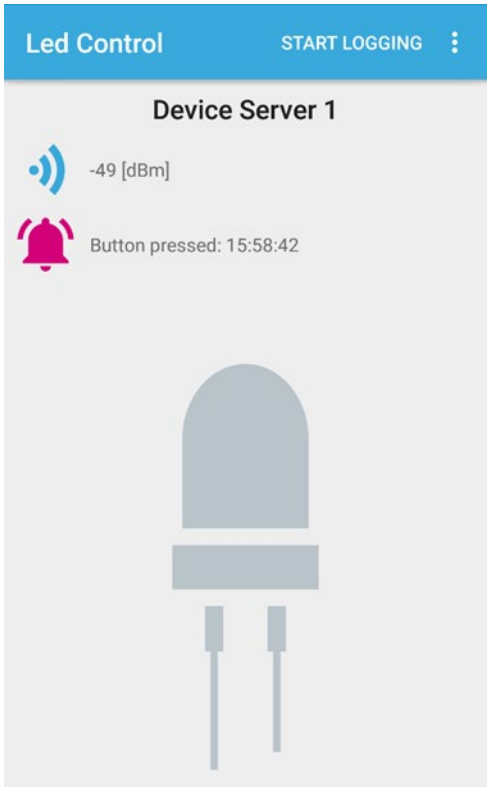
图 25. P2P 服务器/客户端通信序列



7.4.2 P2P 服务器应用

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

图 26. 连接到 ST BLE sensor 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 射频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 服务函数：
Middlewares\STM32_WPAN\ble\core\Src\blesvc\p2p_stm.c

表 22. P2P 服务函数

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

P2P 服务器应用程序控制：
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)
{
Switch (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）。

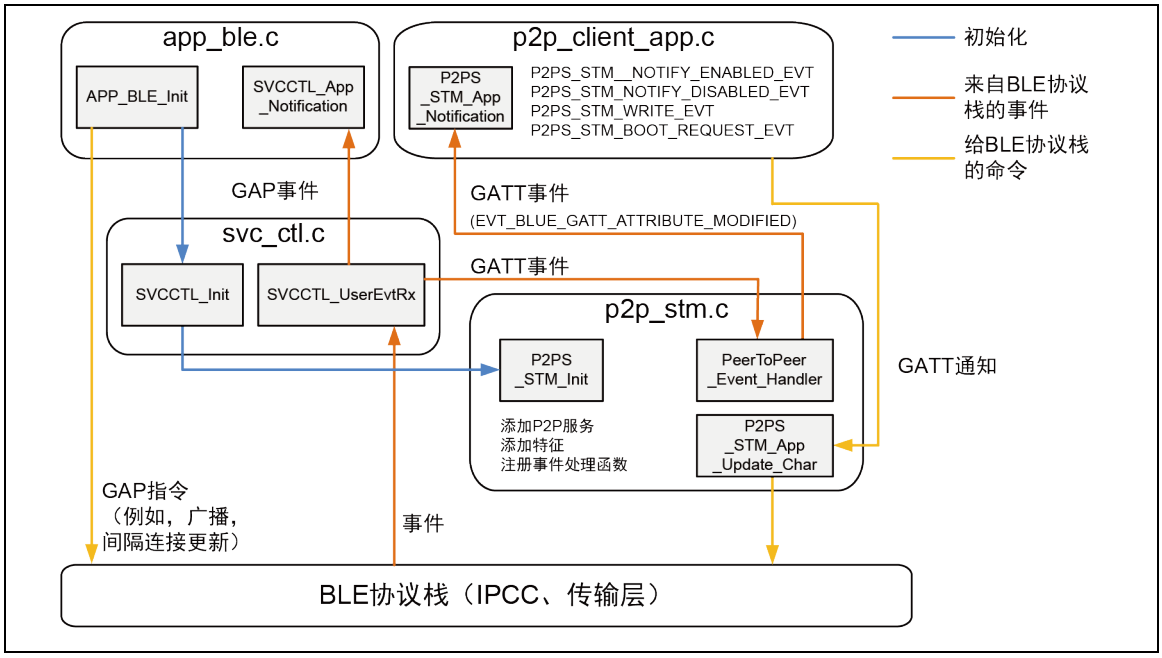
```
P2PS_Send_Notification ()
void P2PS_Send_Notification(void)
{
  if(P2P_Server_App_Context.ButtonControl.ButtonStatus == 0x00){
    P2P_Server_App_Context.ButtonControl.ButtonStatus=0x01;
  } else {
    P2P_Server_App_Context.ButtonControl.ButtonStatus=0x00;
  }

  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 27. 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;
```

- 发起与发现的任何 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()
- 启动发现过程并管理远程 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 : 发现 TX 的描述符 - 写入特性\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]);
/* 存储特征句柄，而非属性句柄 */
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

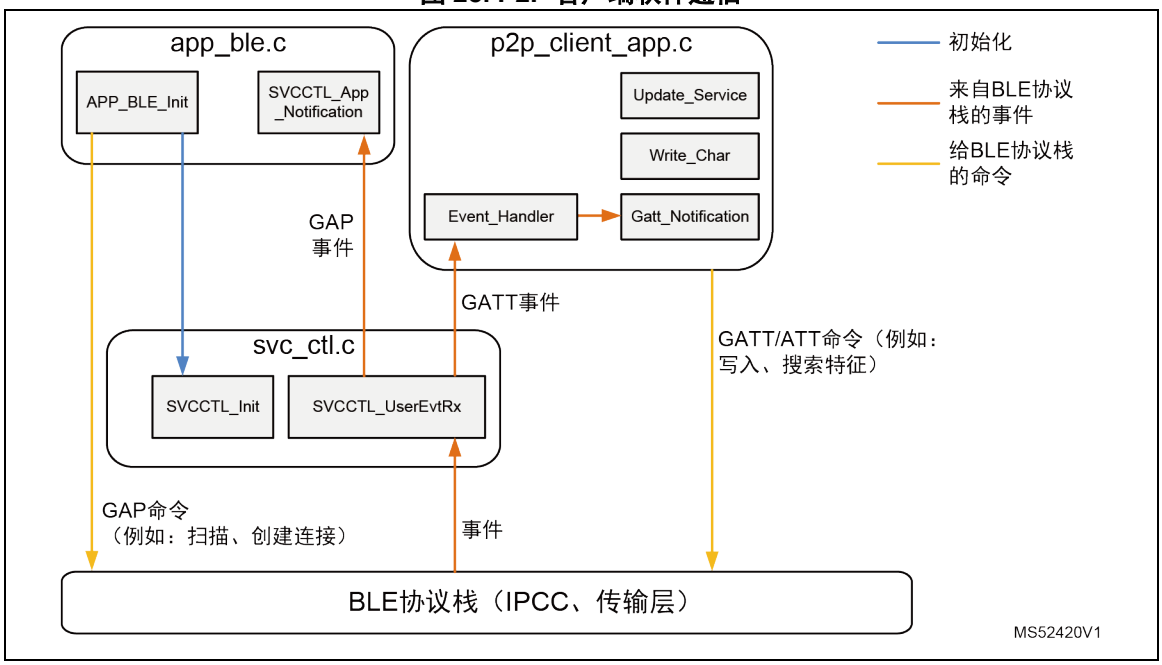
```
tBleStatus Write_Char(uint16_t UUID, uint8_t Service_Instance, uint8_t*pPayload){
tBleStatus ret = BLE_STATUS_INVALID_PARAMS;
uint8_t index;
索引 = 0；
while((index < BLE_CFG_CLT_MAX_NBR_CB) && (aP2PClientContext[index].state
!= APP_BLE_IDLE)){
switch(UUID){
case P2P_WRITE_CHAR_UUID:
ret =aci_gatt_write_without_resp(aP2PClientContext[index].connHandle,
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 28. 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 射频协议栈、CPU1 应用或配置二进制文件：

- 它要求永不删除应用的前六个 Flash 存储器扇区（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 射频固件
  - 在 CPU1 用户 Flash 存储器中的任意地址下载用户数据。

7.5.1 CPU1 用户 Flash 存储器映射

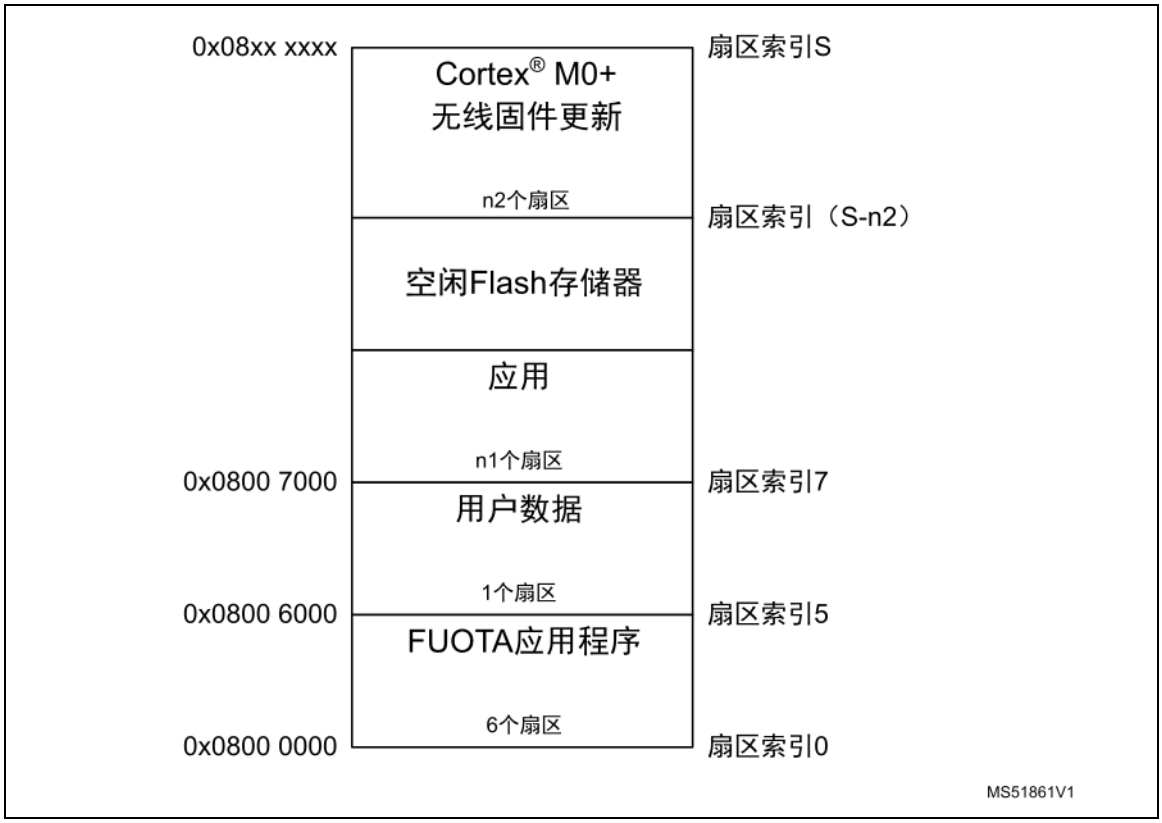
FUOTA BLE 应用不能自我更新但能够：

- 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 29. FUOTA 存储器映射



7.5.2 BLE FUOTA 应用启动

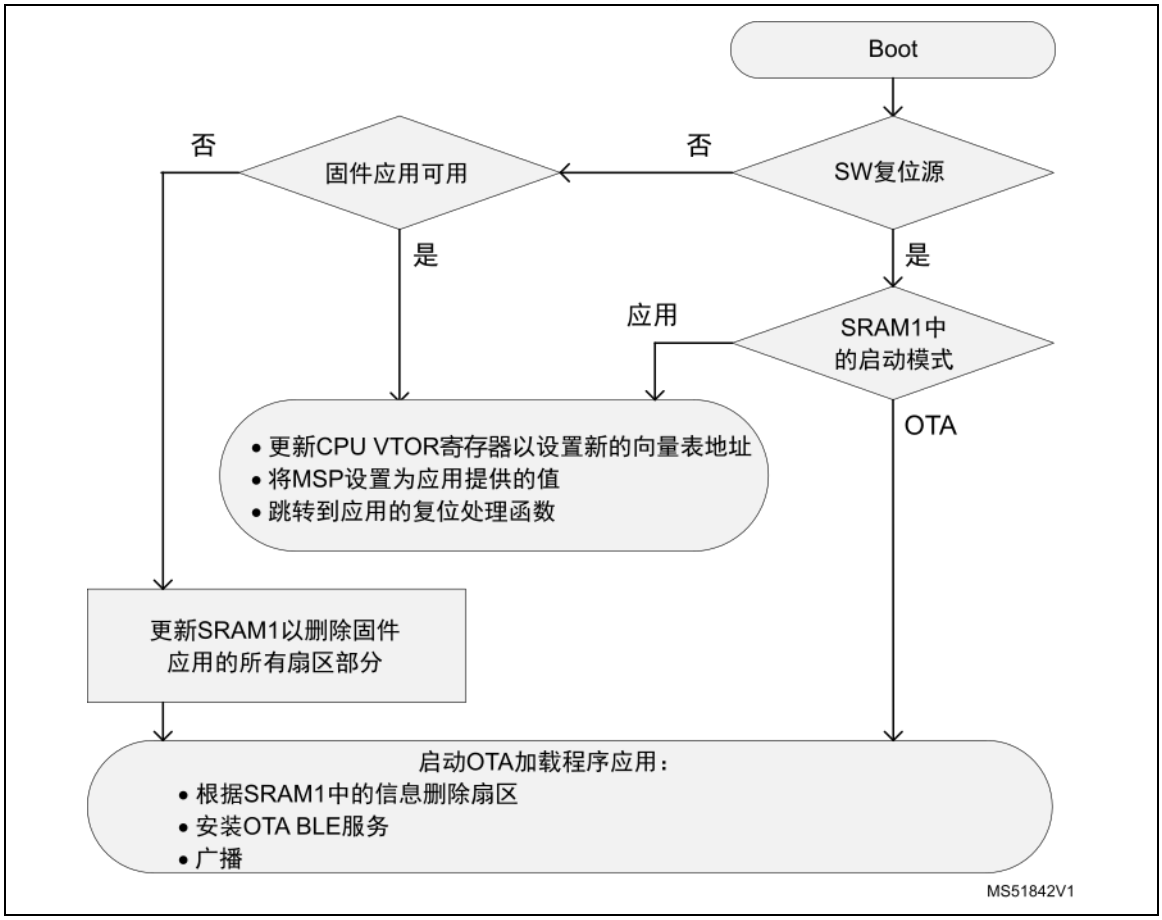
在编译并加载项目 BLE_Ota 后，应用可以：

- 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 射频协议栈、CPU1 应用或用户数据固件更新）。

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

图 30. 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

组	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	写入	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 – 0x01：开始 M0+文件上传 – 0x02：开始 M0+文件上传 – 0x07：文件上传完成 – 0x08：取消上传	0x007000

表 25. 文件上传确认重启请求特征规范

八位组 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的 FUOTA GATT 服务和特征应用程序（7 扇区没有应用程序二进制文件）
- 应用已在运行，支持重启请求特征

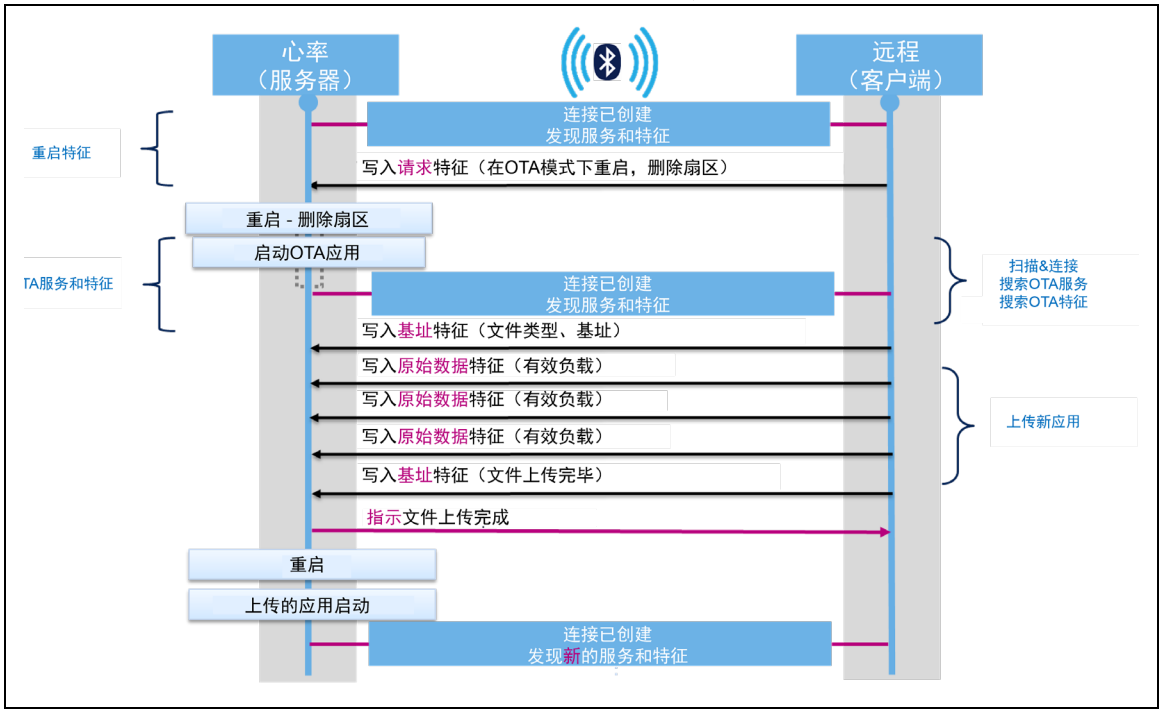
表 27. 重启请求特征规范

Octects LSB	0	1	2
名称	启动模式	扇区索引	要擦除的扇区数
值	– 0x00 应用程序 – 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 Flash 存储器中。
9. 通过基址特征确认文件传输结束。
10. 所接收文件的确认是由文件上传确认特征指示的。
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 31. 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

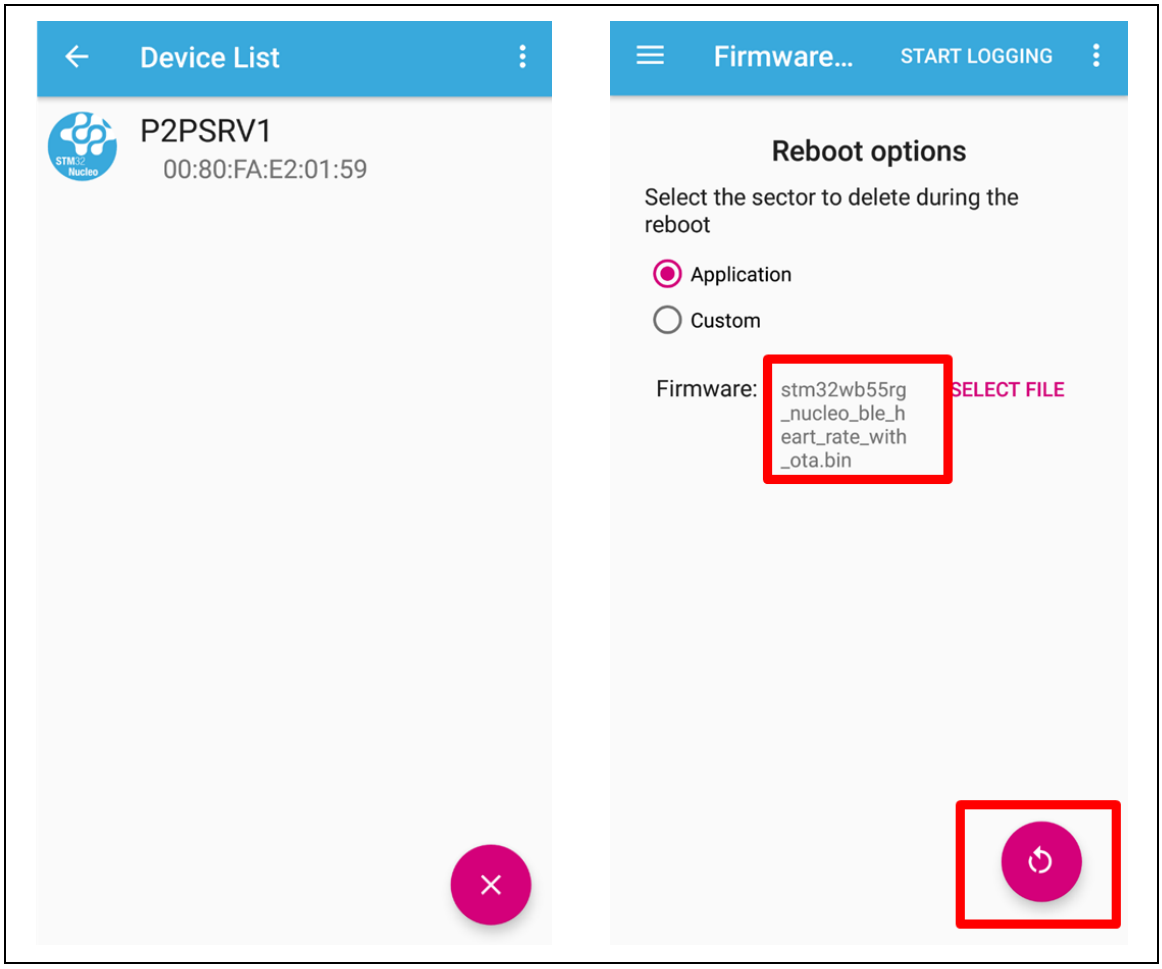
两个工程的广播元素中均包含 OTA 重启请求位掩码。它是远程设备（scanner）快速获取重启请求特征信息存在的一种方式。

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。

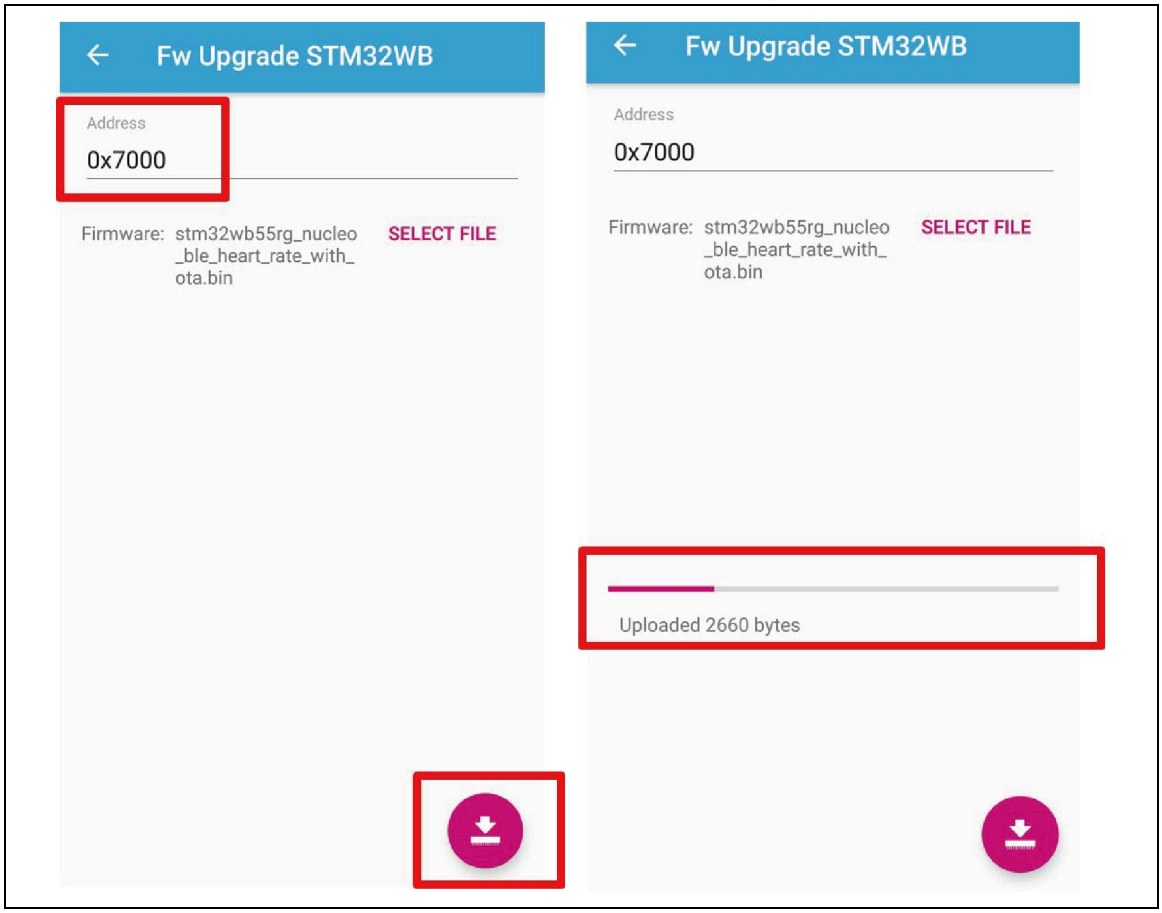
- 编译 BLE_Ota 项目并加载到地址 0x0800 0000
- 编译 BLE_p2pServer_ota 项目并加载到地址 0x0800 7000
- 重启设备
  - 在此阶段，P2P 服务器广播其存在。
- 通过 ST BLE Sensor 移动应用发现并连接到 P2P 服务器
- 移动到重启面板
- 选择二进制文件“BLE_HeartRate_ota”（在演示前复制到智能手机内存）
- 点击上传
  - 在此阶段，重启请求特征用于提供要擦除的扇区和下一个重启阶段（FUOTA 应用）的相关信息。

图 32. 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 33. P2P 服务器 - 应用固件更新



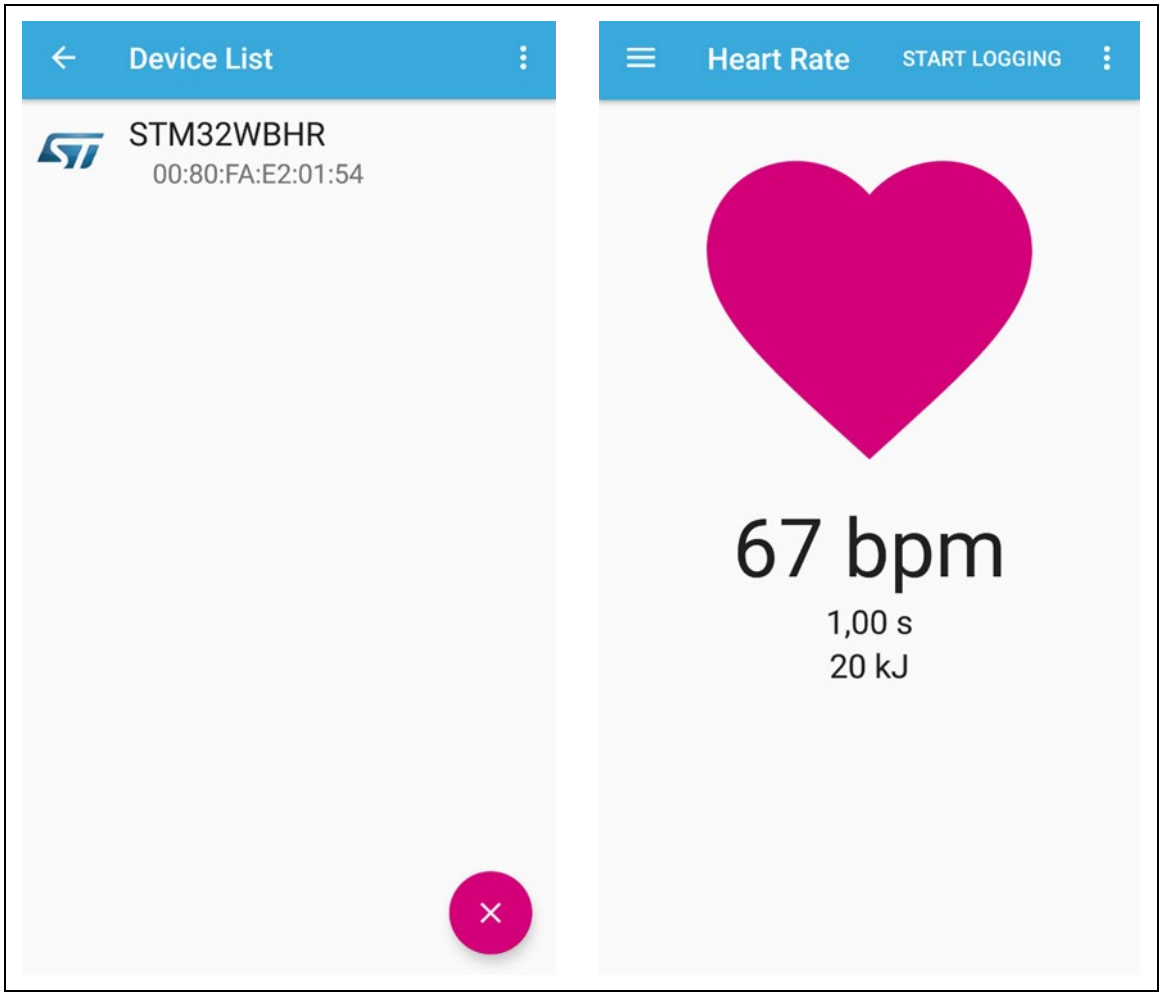
在上传结束后，执行重启流程以启动新应用。

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。

注意： *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 34. 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 定义 OTA 重启特征

```
/*****************************************************************************
*****
  * 空中功能（OTA） - STM 专有

*****************************************************************************
****/
#define BLE_CFG_OTA_REBOOT_CHAR           1 /**< 重启 OTA 模式特征 */
```

- app_ble.c

```
/**
    * 初始化 ADV - Ad 制造商元素 - 支持 OTA 位掩码
  */
#if(BLE_CFG_OTA_REBOOT_CHAR != 0)
```

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

- p2p_stm.c 添加特征（中间件）

```
#if(BLE_CFG_OTA_REBOOT_CHAR != 0)
      /**
        *       添加启动请求特征
        */
      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 接收 GATT 层的请求并通知应用（中间件）

```
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 管理重启请求（应用）

```
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。

在刷写新的应用时，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。

希望开发自己的产品（基于公共设备地址）的客户必须从 IEEE 注册机构获得 IEEE 分配的 48 位通用 LAN MAC 地址，而不是使用已经分配给意法半导体的地址。

关于 BLE 设备如何能够生成随机地址的详细信息，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 Flash 存储器或 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射频微处理器发送命令 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 Flash 存储器中。

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 如何设置 IRK（身份根密钥）和 ERK（加密根密钥）

实际实现的解决方案具有固定的 IRK 和 ERK 值。

如果在启用了绑定和隐私特性后发生了 HW 复位，如果 IRK 是随机生成的，则不可能解码私有地址。

如何写入用于派生长期密钥（LTK）和连接签名解析密钥（CSRK）的 IRK 和 ERK：

定义：CFG_BLE_IRK and CFG_BLE_ERK

Declare: static const uint8_t BLE_CFG_IR_VALUE[16] = CFG_BLE_IRK;

Declare: static const uint8_t BLE_CFG_ER_VALUE[16] = CFG_BLE_ERK;

写入 IRK:

```
aci_hal_write_config_data(CONFIG_DATA_IR_OFFSET, CONFIG_DATA_IR_LEN,
(uint8_t*)BLE_CFG_IR_VALUE);
```

写入 ERK:

```
aci_hal_write_config_data(CONFIG_DATA_ER_OFFSET, CONFIG_DATA_ER_LEN,
(uint8_t*)BLE_CFG_ER_VALUE);
```

另一种解决方案是在每个设备上生成一个随机密钥，并在第一次启动后将其存储在 Cortex M4 侧的 Flash 存储器中。该值不会丢失，会一直使用，密钥因设备而异。

7.6.3 如何将任务添加到调度器

- 声明任务 ID - app_conf.h -

```
/**< 在该列表中添加可能发送 ACI/HCI 指令的所有任务 */
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,      /**< 应为列表中的最后一项 */
} CFG_Task_Id_With_HCI_Cmd_t;

/**< 在该列表中添加从不发送 ACI/HCI 指令的所有任务 */
typedef enum
{
  CFG_FIRST_TASK_ID_WITH_NO_HCICMD = CFG_LAST_TASK_ID_WITH_HCICMD - 1,
/**< 应为列表中的第一项 */
    CFG_TASK_SYSTEM_HCI_ASYNCH_EVT_ID,
    CFG_LAST_TASK_ID_WITHO_NO_HCICMD
/**< 应为列表中的最后一项 */
} 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);
```

7.6.4 如何使用定时器服务器

- 创建带有回调函数的定时器

```
/**
   * 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);
}
```

7.6.5 如何启动 BLE 协议栈 - SHCI_C2_BLE_Init()

```
SHCI_C2_Ble_Init_Cmd_Packet_t ble_init_cmd_packet =
    {
      {{0,0,0}},                                  /**< 未使用的包头 */
      {0,                                         /** pBleBufferAddress 未使用 */
      0,                                          /** BleBufferSize 未使用 */
      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_OPTIONS,
      0,
      CFG_BLE_MAX_COC_INITIATOR_NBR,
      CFG_BLE_MIN_TX_POWER,
      CFG_BLE_MAX_TX_POWER}
    };
```

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 服务特征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。

为了计算 CFG_BLE_ATT_VALUE_ARRAY_SIZE，需要添加与默认 GATT 服务相关的特征。

默认情况下，存在两个必须包含的服务，具有专用特征：

- 通用接入服务：服务 UUID 0x1800，具有三个强制特征：
  - 设备名称 UUID 0x2A00.
  - 外观 UUID 0x2A01。
  - 外设首选的连接参数。UUID 0x2A04.
- 通用属性服务。UUID 0x1801，具有一个可选特征：
  - 变更的服务 UUID 0x2A05。

每个特征对 attrValueArrSize 值的贡献如下：

- 特征值长度加上：
  - 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）

每个描述符对 attrValueArrSize 值的贡献如下：

- 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 DIVC(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: 最大属性大小。
  */
#define DEFAULT_MAX_ATT_SIZE               (512)

/**
  * PREP_WRITE_X_ATT(max_att)：
  * 当使用的 ATT_MTU 值等于 DEFAULT_ATT_MTU (23)时，为了写入大小为 max_att 的特征计算需要的准备写入请求数量。
  *
  */
#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 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（规范允许的最坏情况）

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。

- 外部晶体 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_OPTIONS

这是一个 8 位参数，每个位启用/禁用一个选项：

- 位 0：
  - 1：仅限 LL
  - 0：LL + 主机
- 位 1：
  - 1：无服务变更描述
  - 0：有服务变更描述
- 位 2：
  - 1：设备名称（只读）
  - 0：设备名称（读/写）
- 位 3：
  - 1：支持扩展广播
  - 0：不支持扩展广播
- 位 4：
  - 1：支持 CS Algo #2
  - 0：不支持 CS Algo #2
- 位 5 和 6：保留（必须保持为 0）
- 位 7：
  - 1：LE 功率等级 1
  - 0：LE 功率等级 2 和 3

7.6.6 NVM 中的 BLE GATT DB 和安全记录

NVM 中的 GATT 记录包括：

- 每个服务：
  - 2 字节的句柄
  - 3 个字节的 UUID（16 位），或 17 个字节的 UUID（128 位）
- 每个特征属性：
  - 2 字节的句柄
  - 3 个字节的 UUID（16 位），或 17 个字节的 UUID（128 位）
  - 2 个字节的 CCCD 值（仅当属性为 CCCD 时）

总数与 size_of_gatt 对应。NVM 中的安全记录是固定的，等于 80 字节，与 size_of_sec 记录对应。

7.6.7 如何计算 NVM 中可以存储的最大绑定设备数量

可存储在 NVM 中的绑定设备数量的计算公式为

N = (total_size_of_nvm – 1) / [(size_of_sec_record + 1) + (size_of_gatt_record + 1)] 其中 total_size_of_nvm = 507 个字（4 个字节）。

7.6.8 NVM 写访问

出现以下情况时，执行 NVM 写入：

- 启用绑定（参数在 aci_gap_set_authentication_requirement 中设置）
- 在第一次连接时，请求并完成配对。在接收到 ACI_GAP_PAIRING_VSEVT_CODE 后，如果 connection_interval > 25 ms 并且无 RF 活动，则安全数据存储在 NVM 中。
- 在重新连接时，如果使用 force_rebond 参数请求配对（强制重新生成配对密钥），如果 connection_interval > 25 ms 且没有 RF 活动，则新的安全数据将存储在 NVM 中。
- 如果已启用绑定功能，当链路断开时，GATT 信息会写入服务器端的 NVM 中。

7.6.9 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：400 ms
- Min_CE_Length = 0 and Max_CE_Length: x280 (400 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）

7.6.10 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 22* 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()将回调注册到服务控制器

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，回调通知服务控制器 GATT 已处理或被忽略。

可以返回的三个值：

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，且不仅仅是不再向没有确认 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。

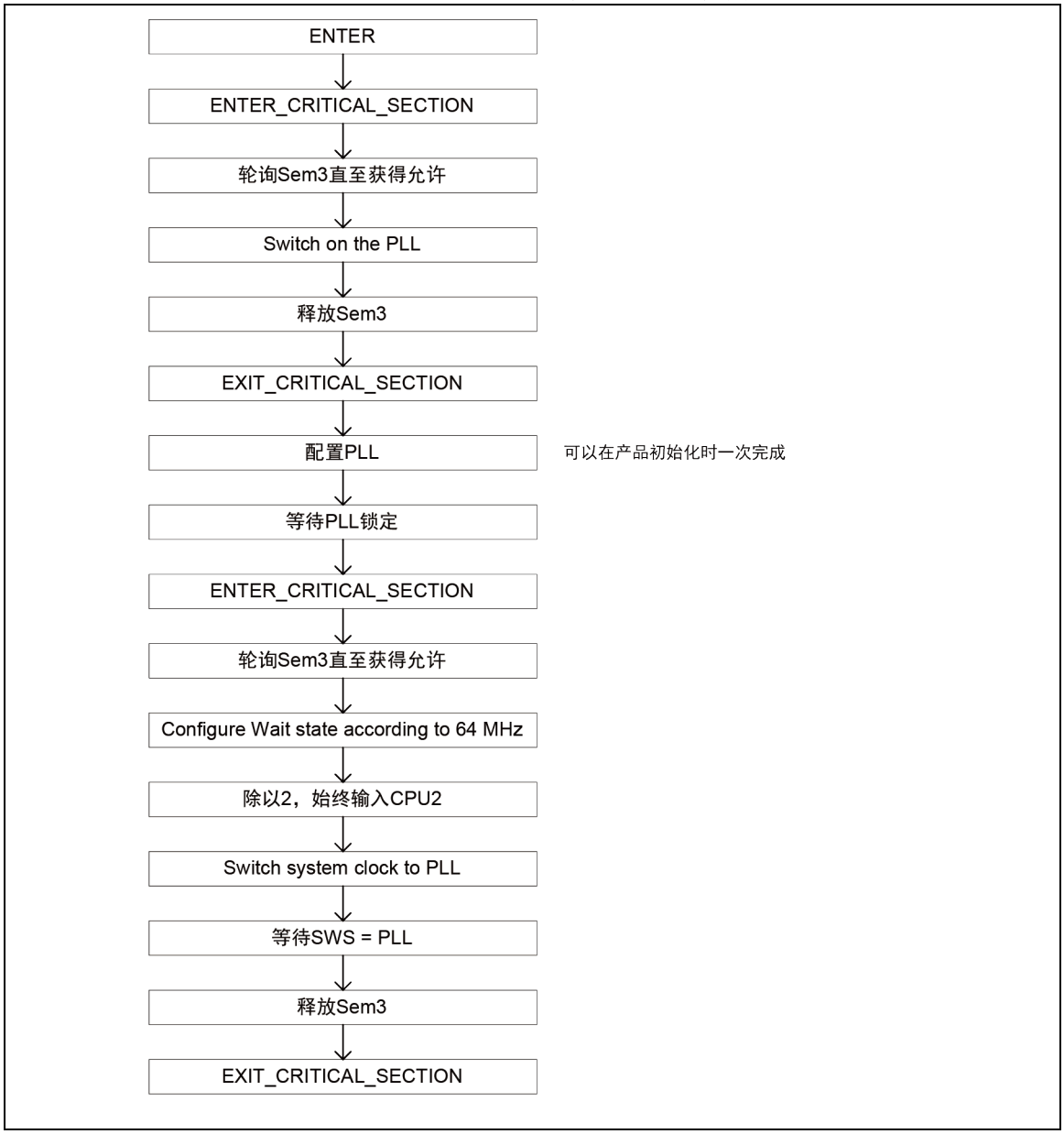
void Custom_xxx_Notification(Custom_xxx_Notification_t *p_notification)

此 API 用于向应用报告（若相关）BLE 服务接收到的 GATT 事件。

7.6.11 如何从 32 MHz 切换到 64 MHz

如需从 32 MHz 切换到 64 MHz，遵循*图 35* 中所示的算法。

图 35. 从 32 MHz 切换到 64 MHz 的算法



7.6.12 如何在退出低功耗模式时重新使能 PLL

如果需要在退出低功耗模式时重启 PLL（必须先配置 PLL），可在 ExitLowPower_1 用户代码部分增加如下代码：

/* 用户代码开始 iExitLowPower_1 */

```
  /* 开启 HSE */
  LL_RCC_HSE_Enable();
  while(!LL_RCC_HSE_IsReady());

  /* 开启 PLL */
  LL_RCC_PLL_Enable();

  /* 等待 PLL 被锁定 */
  while(!LL_RCC_PLL_IsReady());

  /* 根据 64 MHz 配置等待状态 */
  __HAL_FLASH_SET_LATENCY(FLASH_LATENCY_3);

  /* 按除以 2 馈送 CPU2 的时钟划分 */
  LL_C2_RCC_SetAHBPrescaler(RCC_SYSCLK_DIV2);

  /* 将系统时钟切换到 PLL */
  LL_RCC_SetSysClkSource(LL_RCC_SYS_CLKSOURCE_PLL);

  /* 等待 SWS = PLL */
  while (LL_RCC_GetSysClkSource() != LL_RCC_SYS_CLKSOURCE_STATUS_PLL);

/* 用户代码结束 ExitLowPower_1 */
```

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 协处理器进行通信。STM32WB 器件上的等效物理层是邮箱（mailbox），如*第 13.2 节：邮箱（Mailbox）接口*所述。

邮箱（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邮箱（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邮箱（mailbox）接收到的事件。除了通过邮箱（mailbox）完成 BLE 主机协议栈自适应，用户还必须在可以释放异步数据包时通知邮箱（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来构建发送到邮箱驱动程序的系统命令缓冲区，并管理从邮箱接收到的事件（包括向邮箱驱动程序释放异步缓冲区的通知），或者也可以使用邮箱扩展驱动程序（如*第 13.3 节：邮箱接口 - 扩展*中所描述）在提供的传输层之上提供一个接口，负责构建系统命令缓冲区和管理系统异步事件。

BLE_TransparentMode 项目可用作使用邮箱（mailbox）在 BLE HCI 层协处理器之上构建应用（如*第 13.2 节：邮箱（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 M0 固件可供使用，如*表 28* 所示。

表 28. Thread 可以使用的 M0 固件

CPU2 固件库	特性	备注
stm32wb5x_Thread_FTD_fw.bin	FTD： 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）。Thread 角色在*第 13.9.5 节*中描述。
stm32wb5x_Thread_MTD_fw.bin	MTD： 最低功能 Thread 设备	设备只能充当终端设备或休眠终端设备。 MTD 配置比 FTD 配置需要的内存更少。
stm32wb5x_BLE_Thread_static_fw.bin	静态并发模式	设备将两个堆栈（BLE 和 Thread）内嵌在一个二进制文件中，用于静态并发模式。
stm32wb5x_BLE_Thread_dynamic_fw.bin	动态并发模式	设备将两个堆栈（BLE 和 Thread）内嵌在一个二进制文件中，用于动态并发模式。

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。

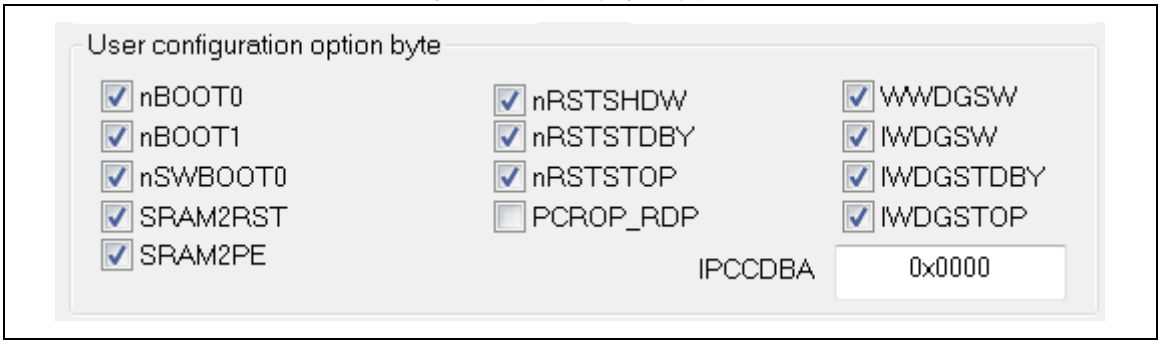
这两个应用程序在 STM32WB 固件包中提供，并带有相关的 readme.txt 文件。

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 Static 模式
- 使用正确的选项字节。

图 36. 用户选项字节设置



小心： 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 内部的射频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

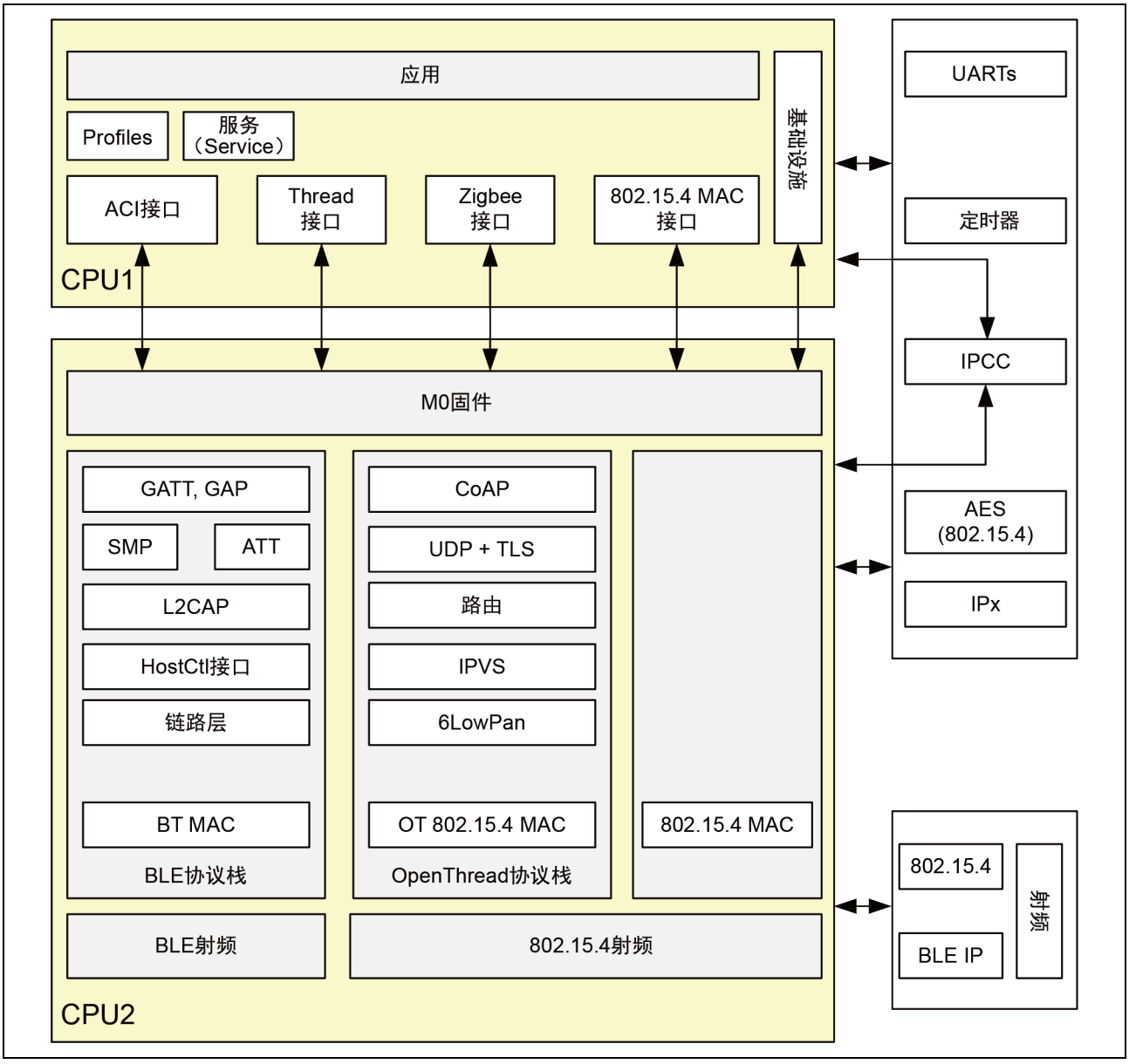
文件名	配置
openthread_api_config_ftd.h	将与 Thread FTD CPU2 固件一起使用
openthread_api_config_mtd.h	将与 Thread MTD CPU2 固件一起使用
openthread_api_config_concurrent.h	将与静态并发模式 CPU2 固件一起使用

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 37 显示了有 BLE 和 Thread 两个射频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 37. 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。

当应用程序调用 OpenThread 函数时，一则同步消息通过 IPCC 发送到 CPU2。与此函数相关的参数存储在共享内存中。

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 38*）。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 39* 所示。

图 38. OpenThread 函数调用

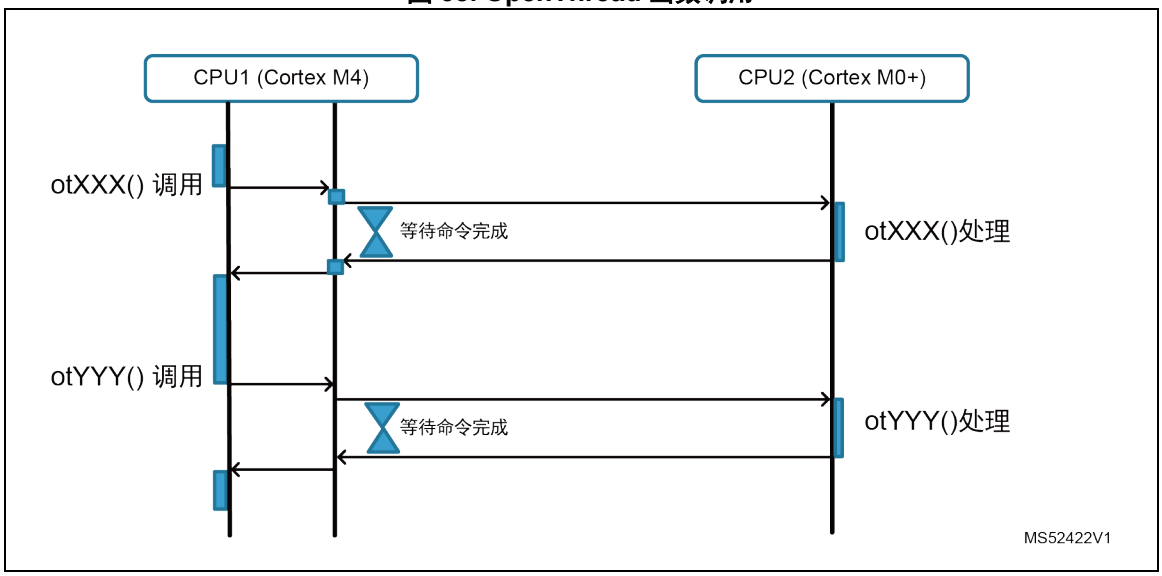


图 39. OpenThread 回调

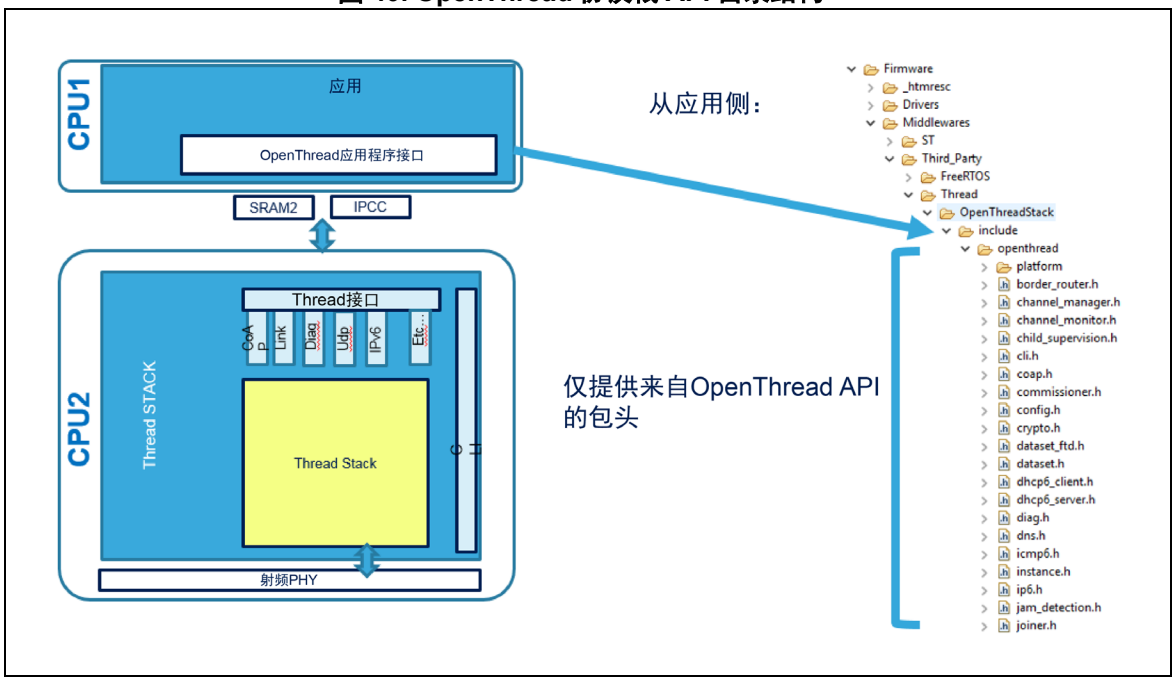
9.6 OpenThread API

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 40. 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 OpenThread API。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)

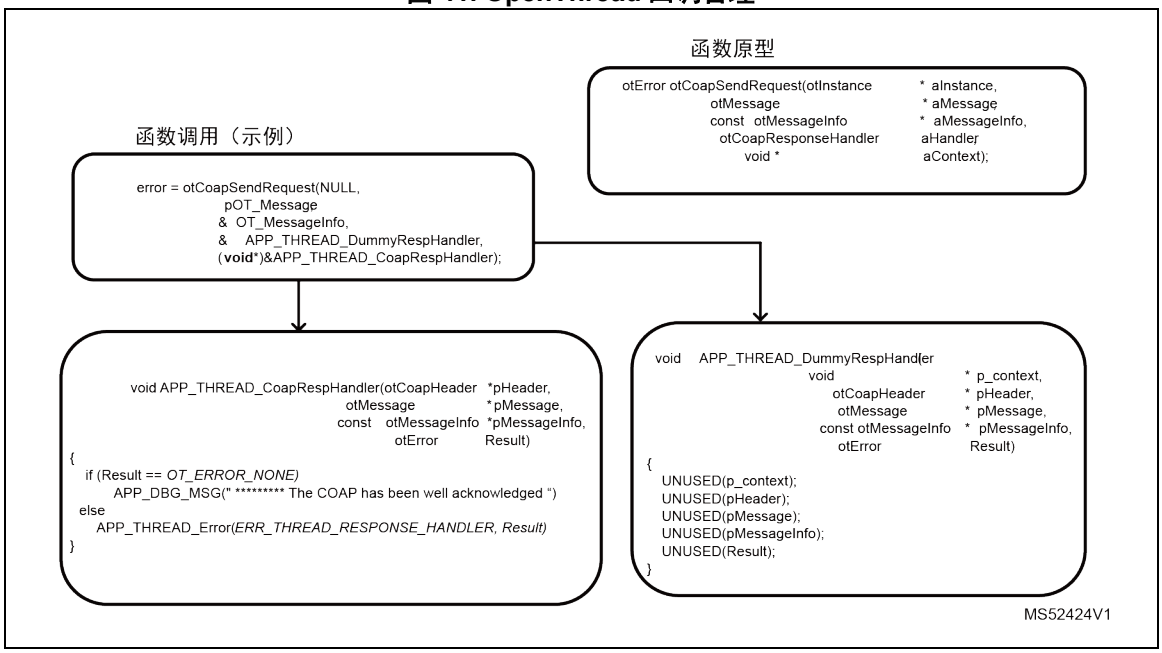
在 STWM32WB Thread 实现中，OpenThread 实例直接位于 CPU2 固件的开头。CPU1 无需注意此参数，它总是设置为 NULL（参见以下代码段中的粗体部分）。

```
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 41* 所示。

图 41. OpenThread 回调管理



注意：　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 Flash 存储器中。由应用程序决定何时必须保存数据（例如：配网阶段后或网络配置后）。

注意：　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 Flash 存储器中擦除。

注意：　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()：读取与加载的射频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 Flash 存储器中，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。只有在配网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。只有在配网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。

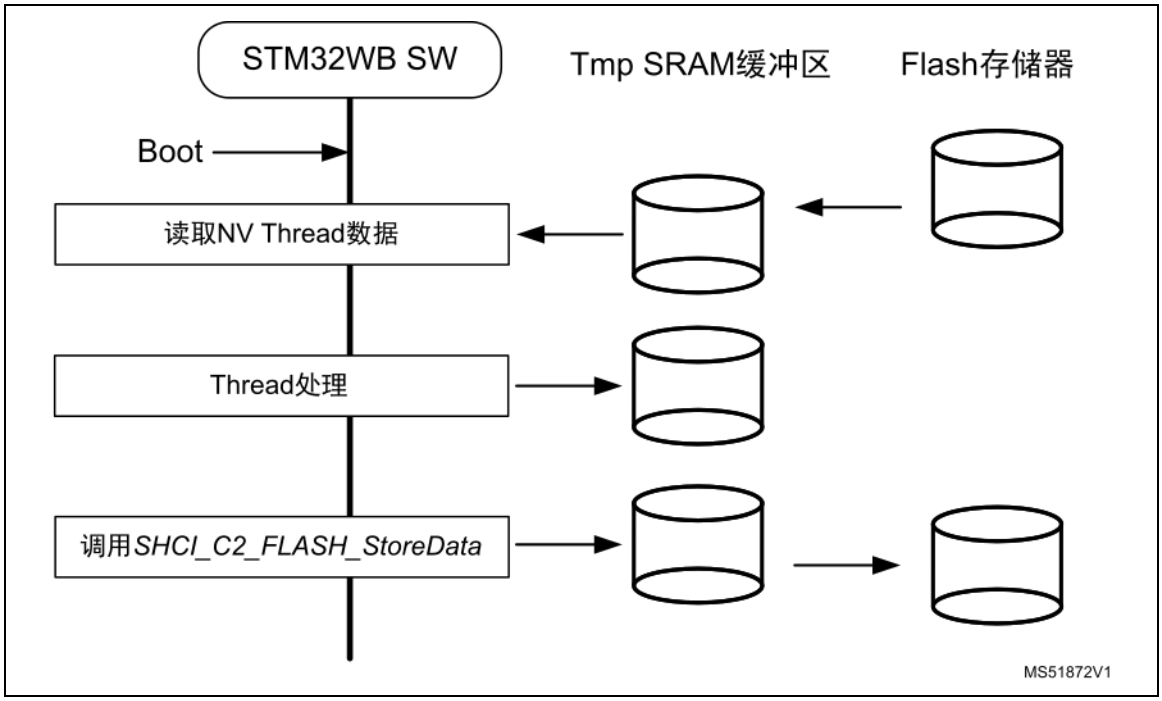
5. 子设备信息：

   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 Flash 存储器中的非易失性 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 42*）。使用函数 SHCI_C2_FLASH_StoreData()将这些非易失性数据强制复制到 Flash 存储器中，具体取决于应用。由于此操作会阻止对 Flash 存储器（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）。

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

图 42. 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：APP_THREAD_DeviceConfig()

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

- 如果板件是该 Thread 网络上的第一个设备，该节点将变为主导设备（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。

注意： *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

Coap 资源的结构如下：

```
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;
```

该结构可以按照固件包中提供的不同 Thread Coap 应用程序示例进行初始化

```
#define C_RESSOURCE                 “light”
static otCoapResource OT_Ressource = {C_RESSOURCE,
APP_THREAD_CoapRequestHandler,"MyOwnContext", NULL};
```

每个设备可以并行分配最多 100 个不同资源。

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* 相关示例）。

10.4 配网

配网要求一个设备的角色为配网设备，另一个设备的角色为加入设备。配网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）。

入网设备是准备加入 Thread 网络的设备。

Thread 配网设备用于验证网络上的设备。它不传输，也不拥有 Thread 网络凭证，例如主密钥。

本文档涵盖了基础的基于 Mesh 的配网，无外部配网设备或边界路由器。

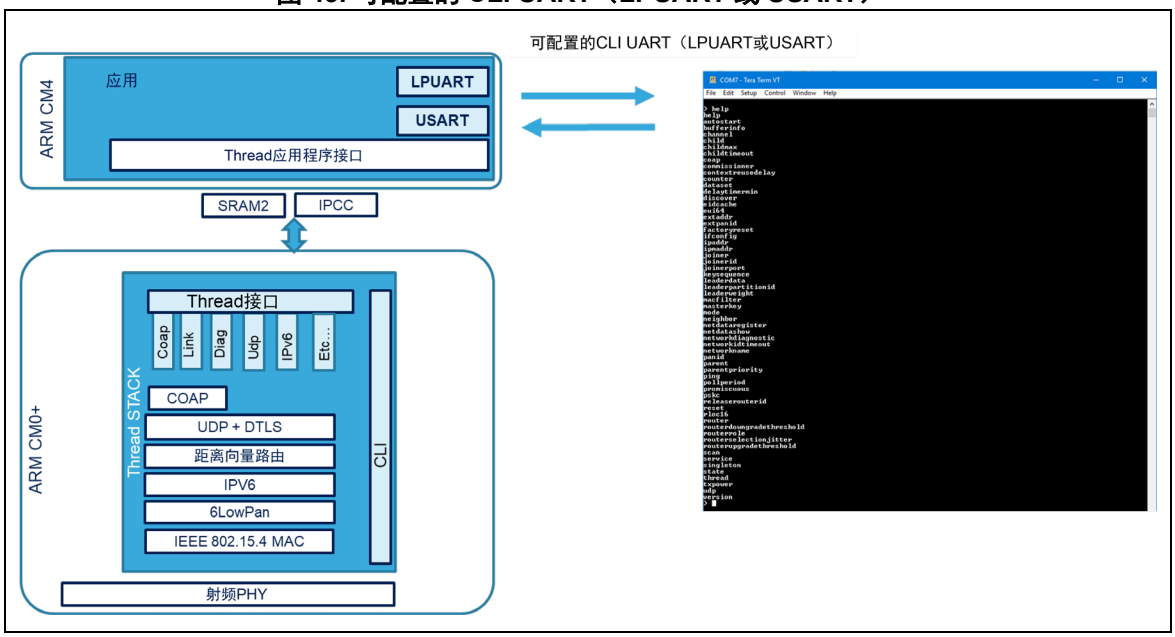
Thread_Commissioning 应用演示了简单的配网示例。

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。

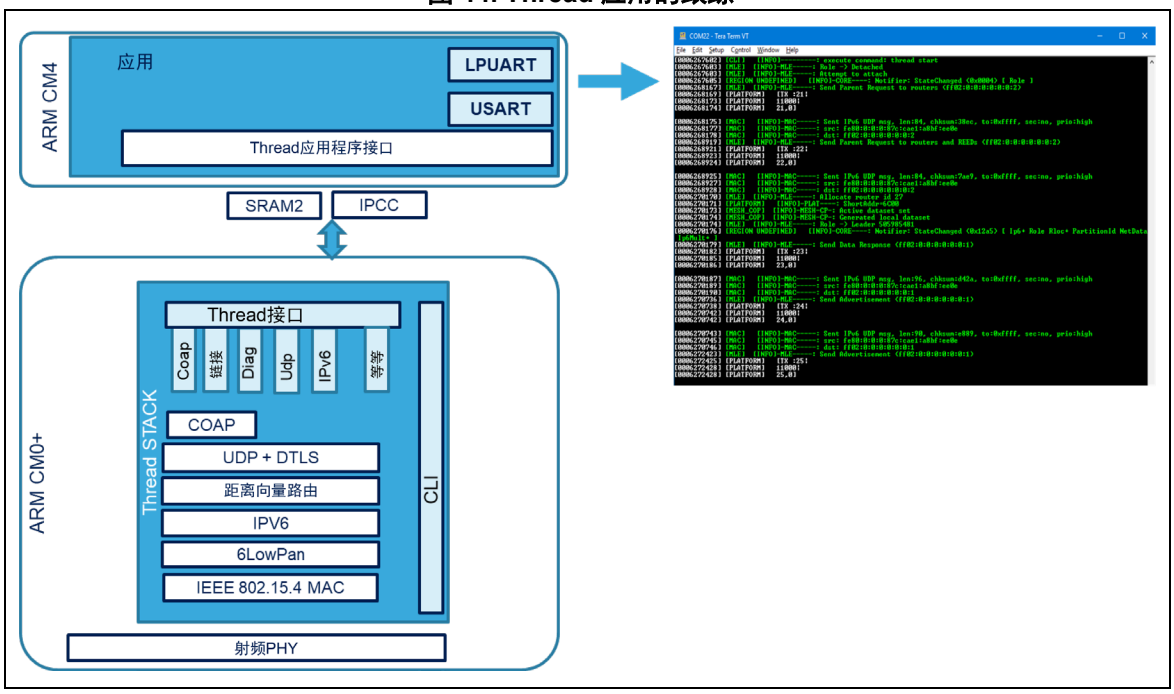
图 43. 可配置的 CLI UART（LPUART 或 USART）



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 44. 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。

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：

- 板件 1 → 板件 2 → (…)→ 板件 n → 板件 1 → 。

每块板都关联了一个特定的 IPv6 地址：

- fdde:ad00:beef:0:442f:ade1:3fc:1f3a for board number 1
- 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

该应用演示了配网设备与入网设备之间的配网过程。

它展示了设备如何通过配网过程将其 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。

一个设备作为配网设备，另一个作为入网设备。

在该应用中，配网设备接受其 Thread 网络中的新入网设备。

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。

注意： *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，一块板处于主导设备（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。

注意： *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）。

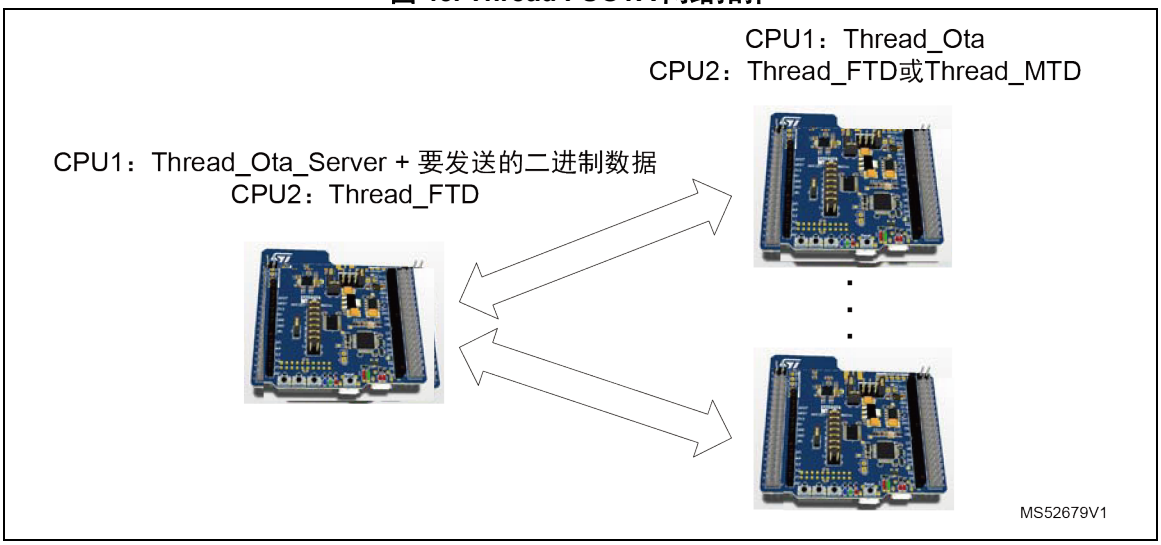
在该阶段，这两块板属于同一个 Thread 网络，设备 2 每秒向设备 1 发送 CoAP 多播请求以点亮/熄灭其蓝色 LED。

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 射频协处理器二进制文件。

图 45. 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 45*）：

- 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

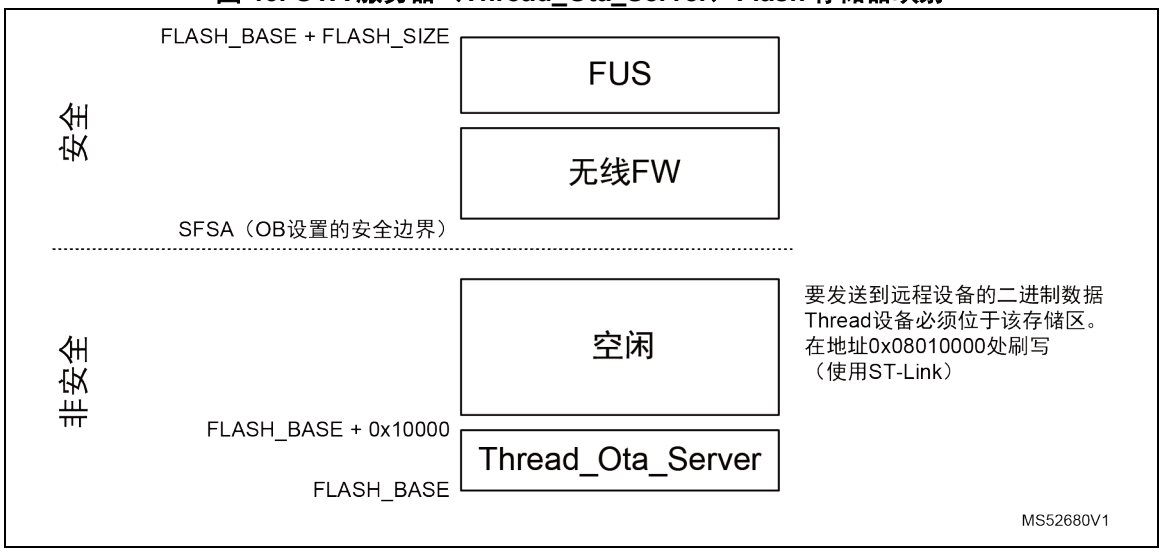
服务器端

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 46*）。

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)

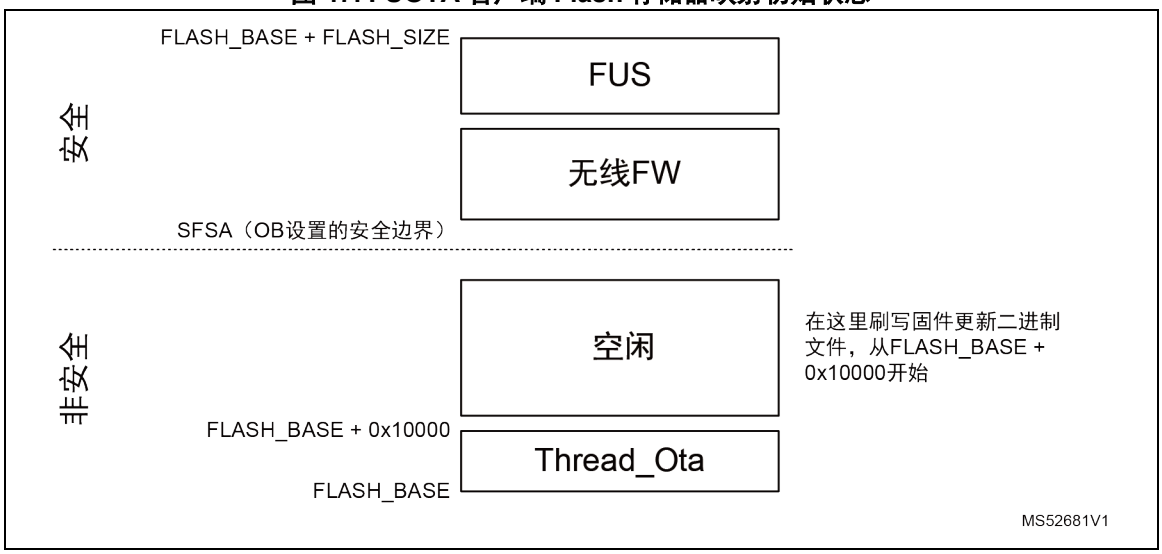
图 46. OTA 服务器（Thread_Ota_Server）Flash 存储器映射



客户端

在客户端，收到来自服务器的二进制数据前的 Flash 存储器如*图 47* 所示。

图 47. FUOTA 客户端 Flash 存储器映射初始状态



接收到来自服务器端的二进制数据后，针对 CPU1 的二进制传输和 CPU2 的二进制传输分别进行 Flash 存储器更新，如*图 48* 和*图 49* 所示。

图 48. CPU1 二进制数据传输后的 FUOTA 服务器 Flash 存储器映射

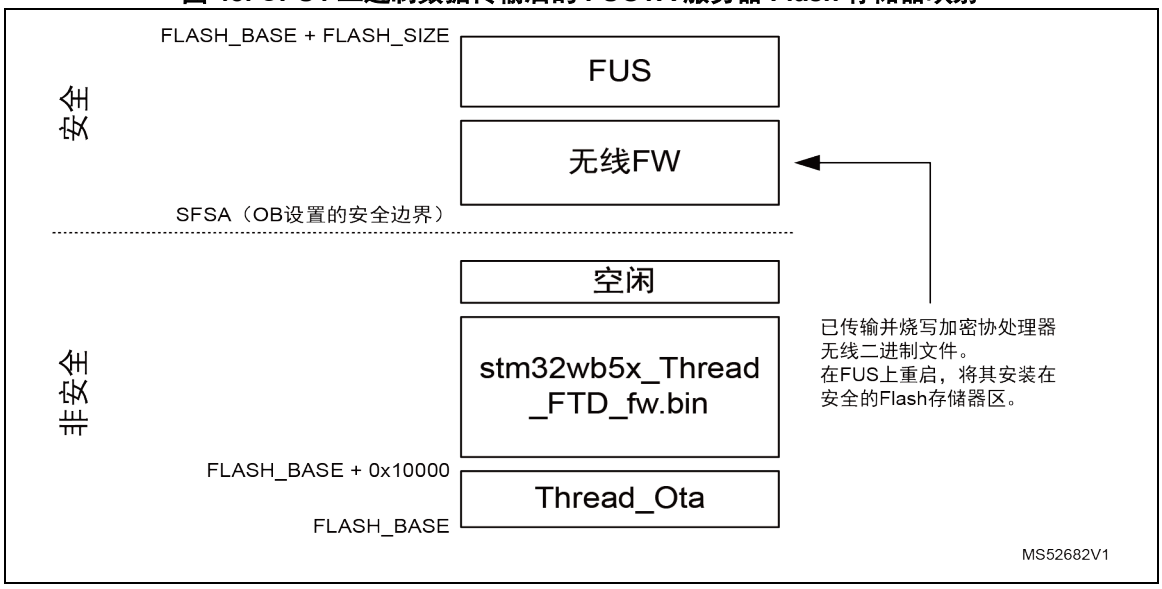
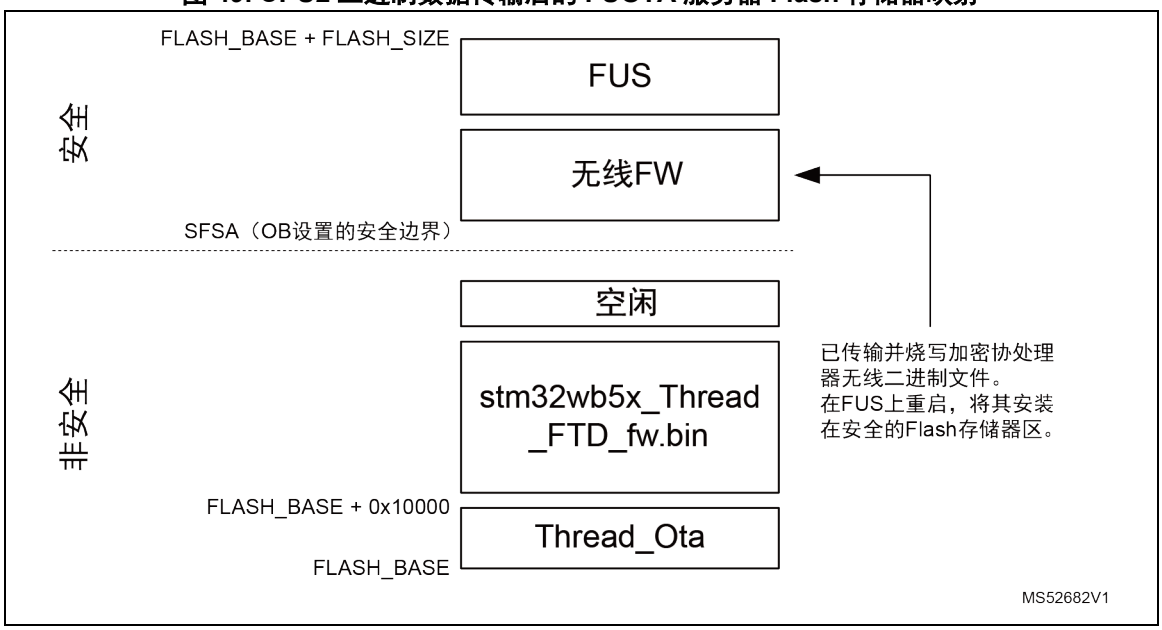


图 49. CPU2 二进制数据传输后的 FUOTA 服务器 Flash 存储器映射

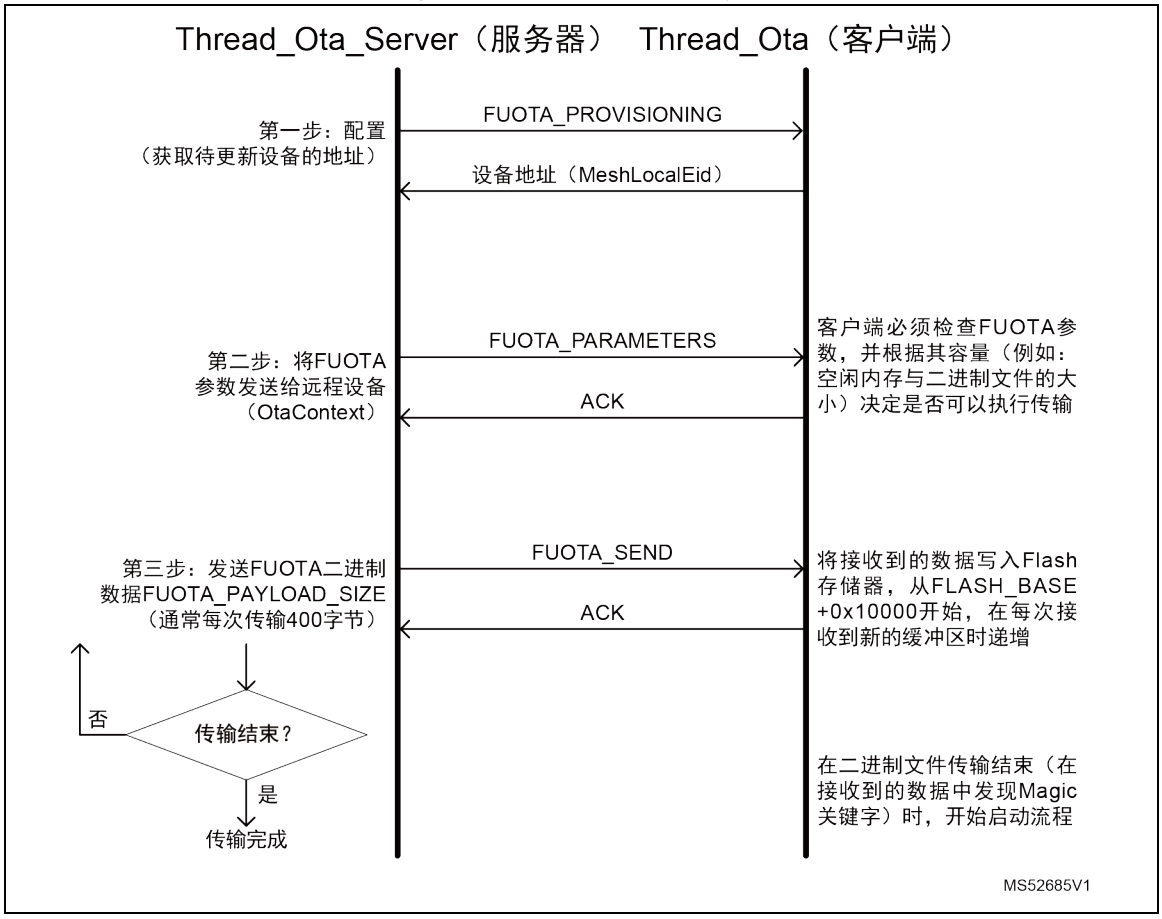


11.8.3 Thread FUOTA 协议

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

图 50 详细描述了执行固件更新传输的步骤。

图 50. 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 Flash 存储器中时的起始地址
   – 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 Flash 存储器。

当找到 Magic 关键字时，表示这是要发送的最后一个缓冲区。

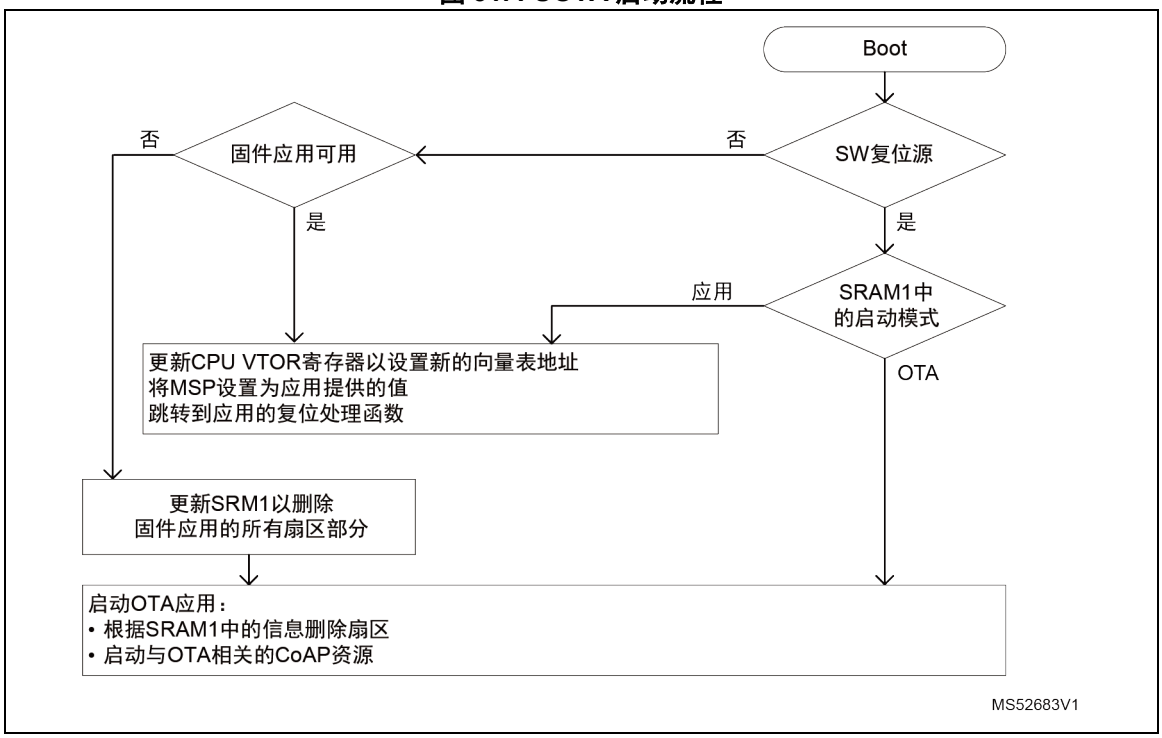
11.8.4 FUOTA 应用启动流程

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 协处理器射频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 51* 所示的过程，从而跳转到 OTA 特定的应用（例如：Thread_Coap_Generic_Ota）：

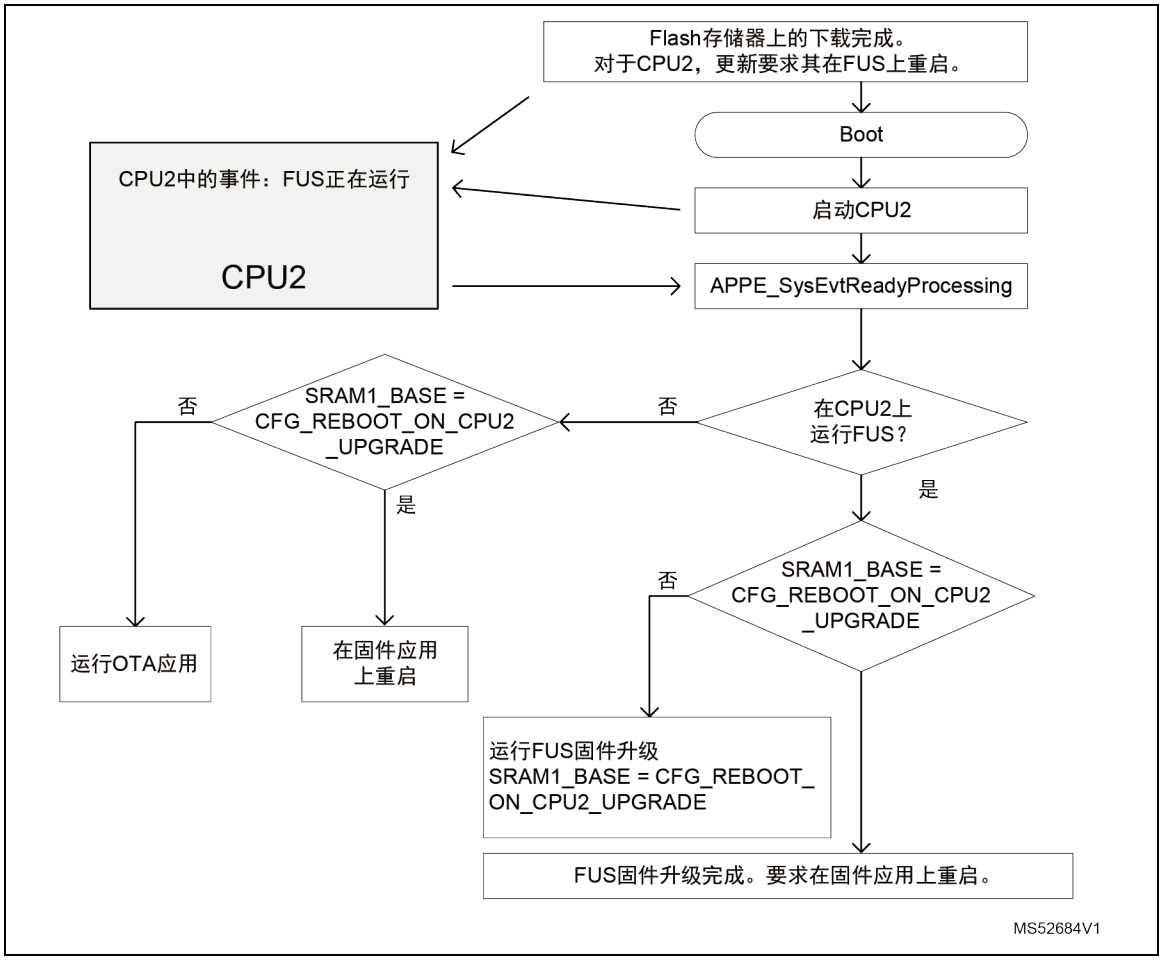
图 51. FUOTA 启动流程



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 52. 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 指向地址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

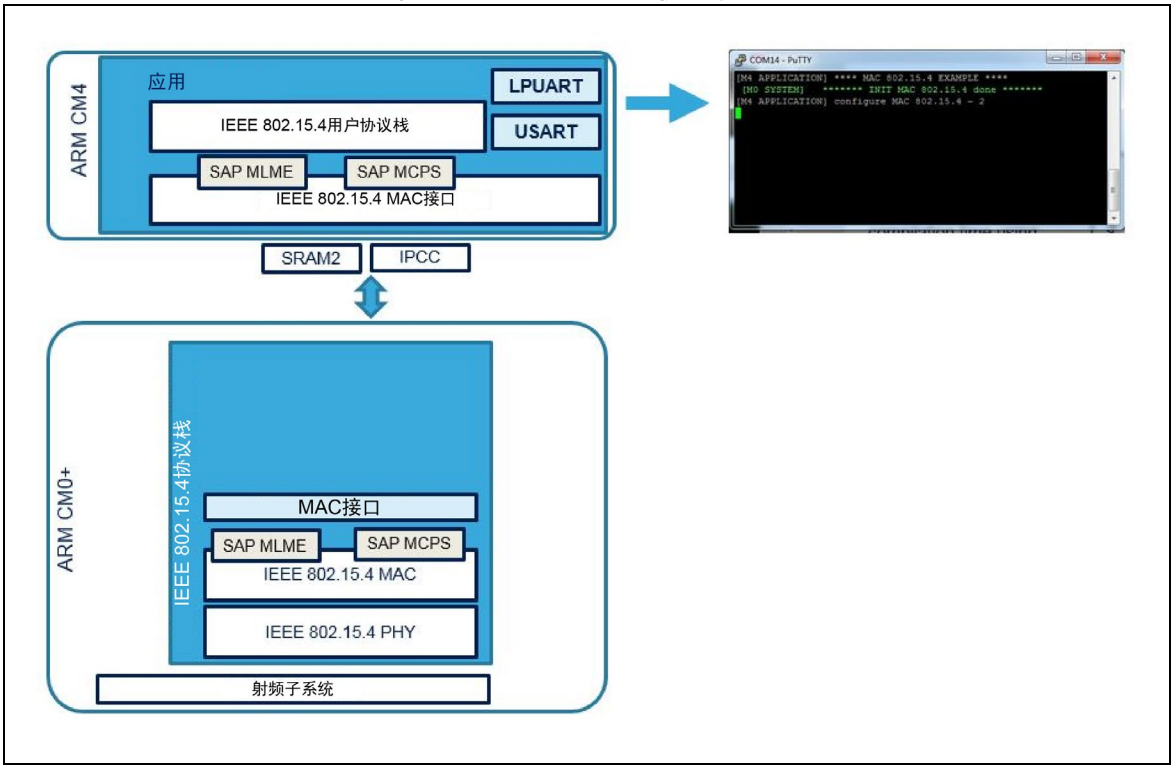
MAC IEEE Std 802.15.4-2011 层运行在 CPU2 内核（射频协议处理器）的 MAC 专用固件上。MAC 层依赖于处理 RF 子系统组件的 PHY 层。

由于此实现以二进制格式提供并在 CPU2 上运行，因此 MAC API 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 53 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 53. MAC 802.15.4 软件架构

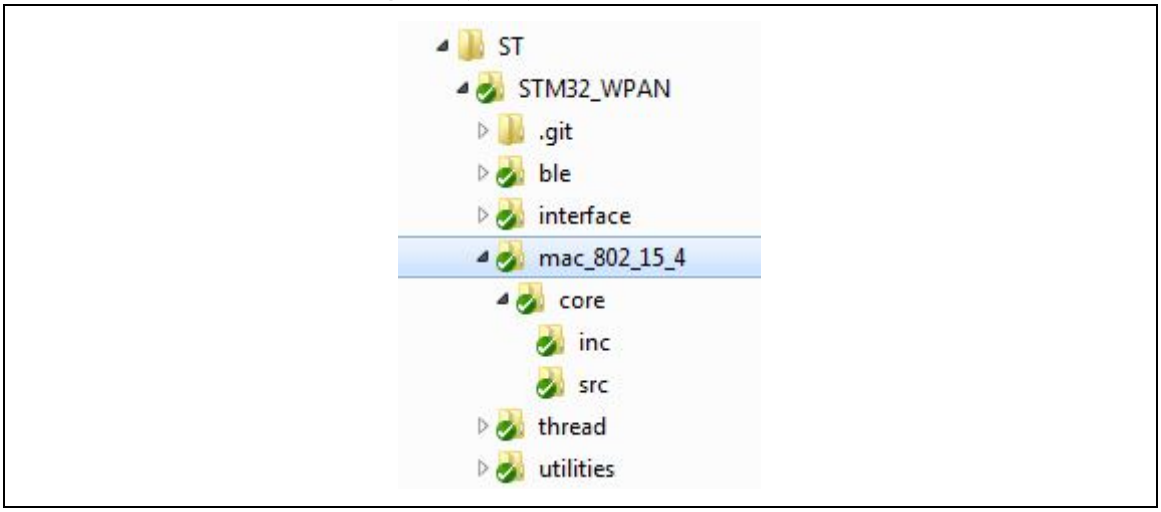


12.3 API

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 54*）下获取。

图 54. 应用内核专用的 MAC API



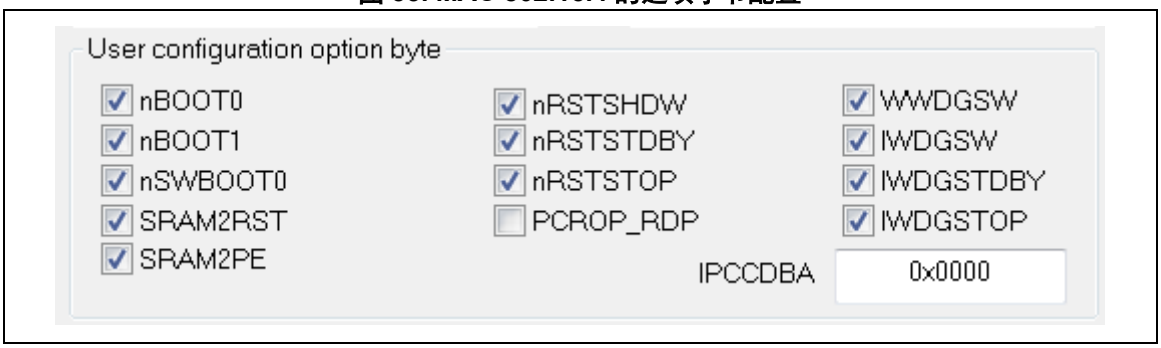
此实现记录在 STM32WB 固件包中目录 Firmware\Middlewares\ST\STM32_WPAN\mac_802_15_4 下的文件 STM32WBxx_MAC_802_15_4_User_Manual.chm 中。具体的原语说明见 IEEE Std 802.15.4-2011 文档。

12.4 如何开始

12.4.1 板配置

确保选项字节设置如*图 55* 所示。

图 55. MAC 802.15.4 的选项字节配置



12.4.2 MAC 射频协议处理器 CPU2 固件

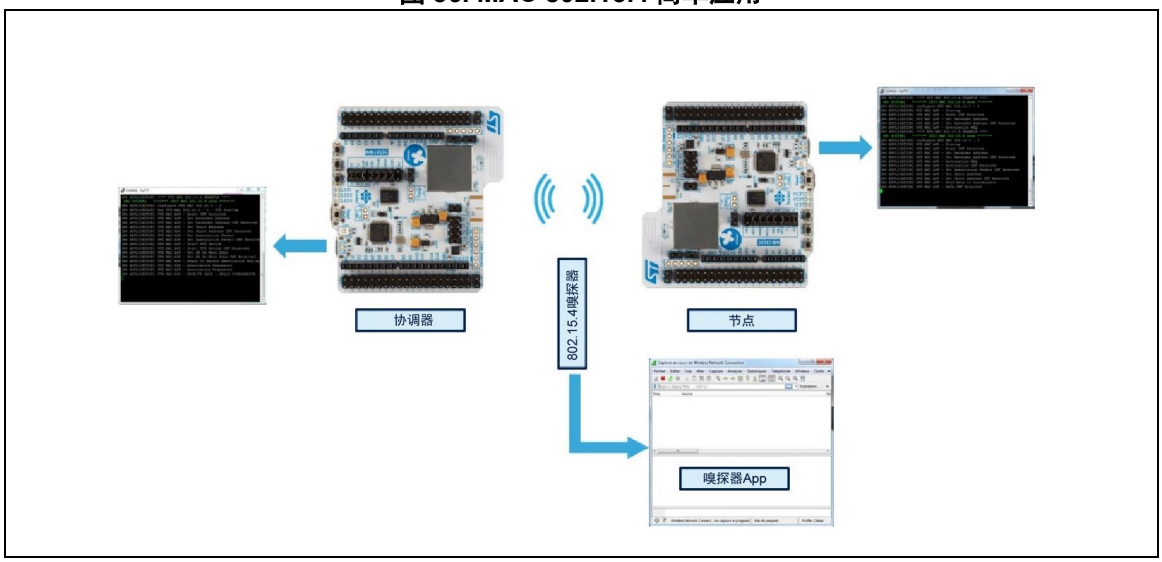
用户首先需要为 CPU2 射频协议内核下载合适的专用 MAC 固件二进制文件，参见 STM32WB 固件包中目录 Firmware\Projects\STM32WB_Copro_Wireless_Binaries 下的文件 Release_Notes.html。

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 MAC API 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：

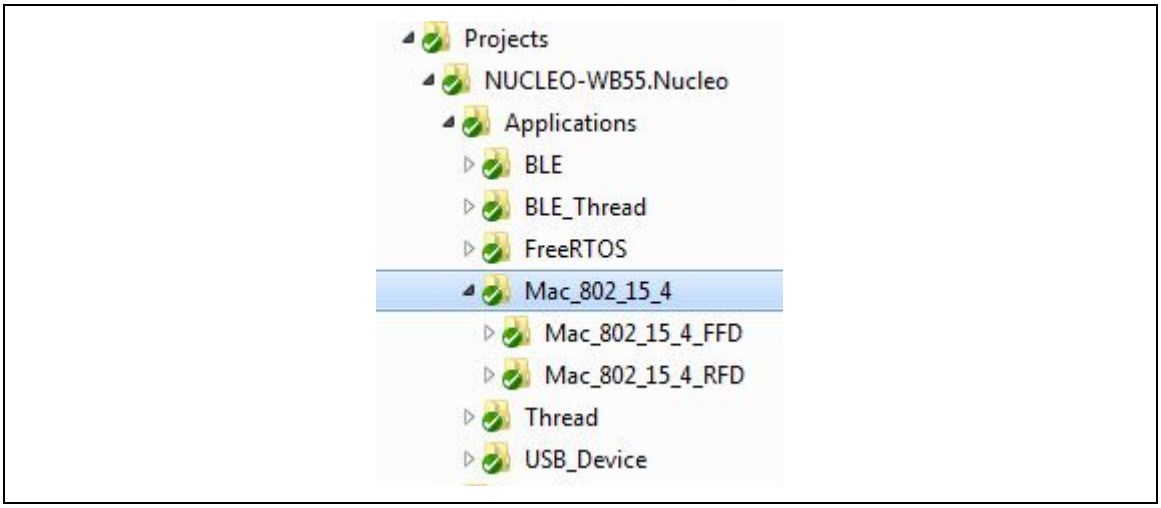
- Mac_802_15_4_FFD：展示如何实现简单的 802.15.4 协调器。此设备按关联请求管理网络，并根据节点需求获取或提供数据。
- Mac_802_15_4_RFD：展示如何实现简单的 802.15.4 节点。此设备向协调器发送关联请求。在被寻址的协调器对请求作出肯定响应后，节点接收其新的短地址，然后向协调器发送数据。

图 56. MAC 802.15.4 简单应用



这两个应用程序（专门用于 Nucleo STM32WB 板）可从 NUCLEO-WBxx.Nucleo 应用程序 Mac_802_15_4 目录获得（参见*图 57*）。

图 57. 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。然后用户可以使用喜欢的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
- 奇偶校验：无
- 流控：XON/XOFF。

运行两个应用会显示如*图 58* 至 *60* 所示的超级终端。

图 58. 协调器启动

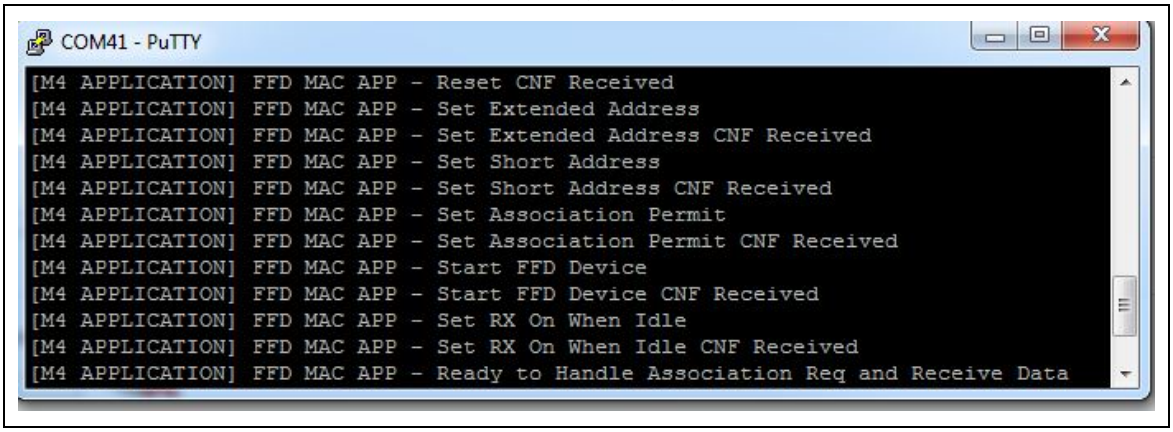


图 59. 节点启动、请求关联和数据发送

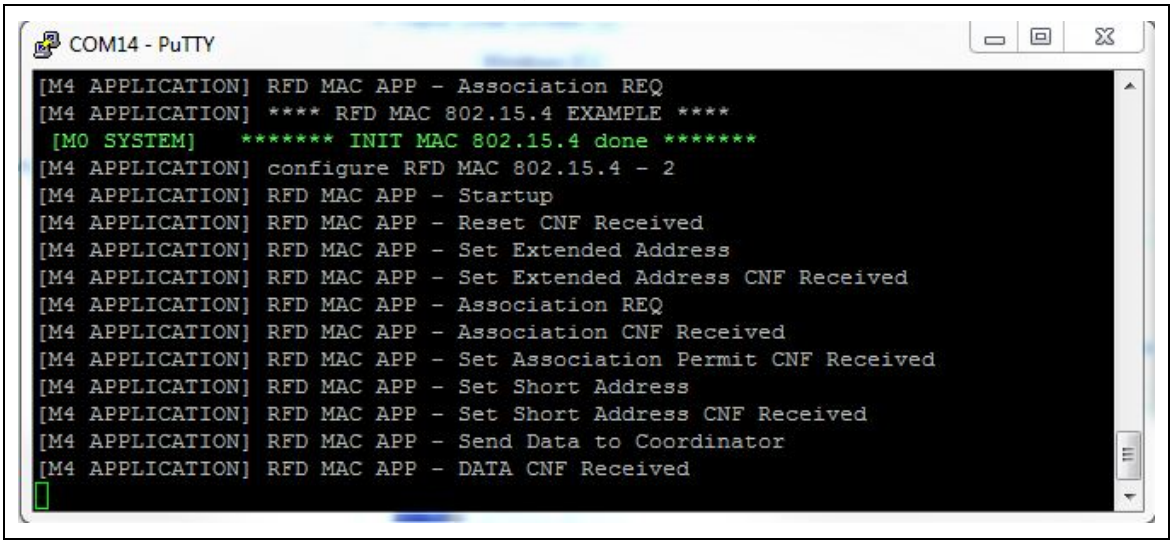
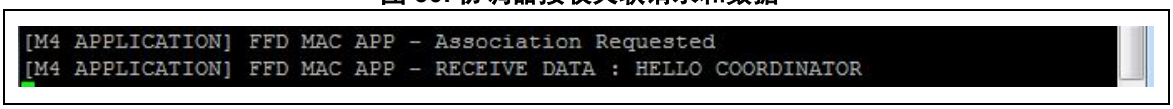


图 60. 协调器接收关联请求和数据



12.4.5 MAC IEEE Std 802.15.4-2011 系统

这是目前已实现的 MAC 系统命令。

SHCI_C2_MAC_802_15_4_Init()启动射频处理器（CPU2）上的 MAC 层和 RF 子系统。

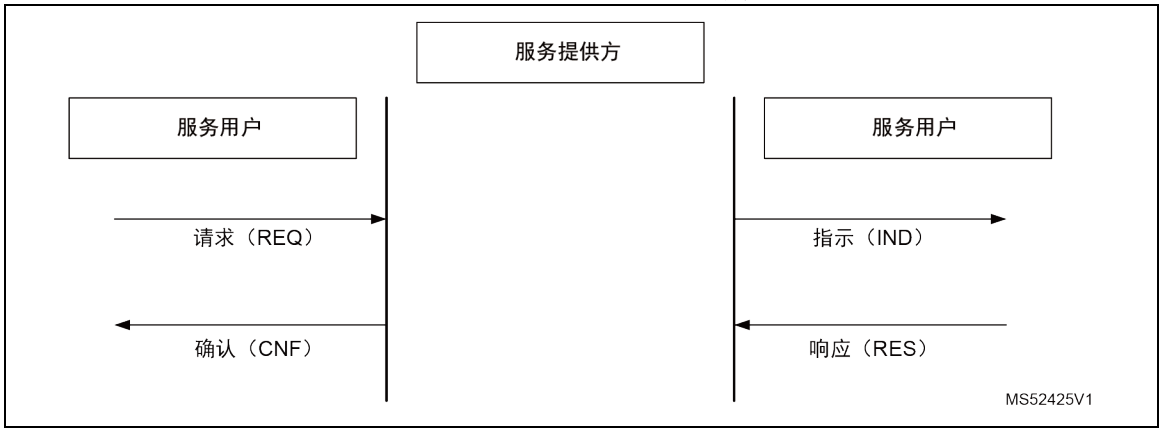
MAC 层无法保证非易失性数据的安全。确保这些数据保存在 Flash 存储器中和恢复以后要使用的数据都取决于上层应用。

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原语。这个抽象层，在 MAC IEEE Std 802.15.4-2011 规范文档中描述，如*图 61* 所示。

图 61. MAC 802.15.4 抽象层



给出的 API 允许用户调用 REQ 和 RES 原语，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,0 x08);
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 回调初始化 */
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：

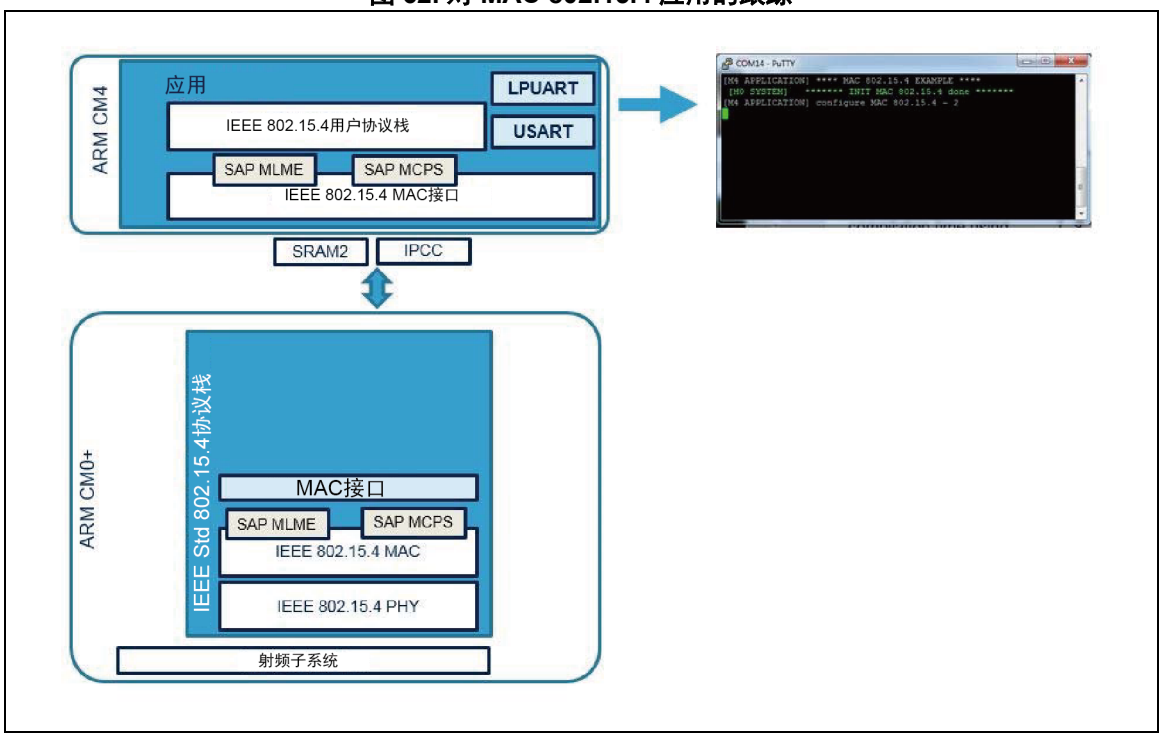
对于来自 MAC 层的数据指示信息，使用 macCbConfig.mcpsDataIndCb 调用 APP_MAC_mcpsDataIndCb 回调，可按以下方式实现该回调，以便检索 MAC_dataInd_t 结构（app_mac_802-15-4_process.c）承载的指示数据：

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

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

}
```

图 62. 对 MAC 802.15.4 应用的跟踪

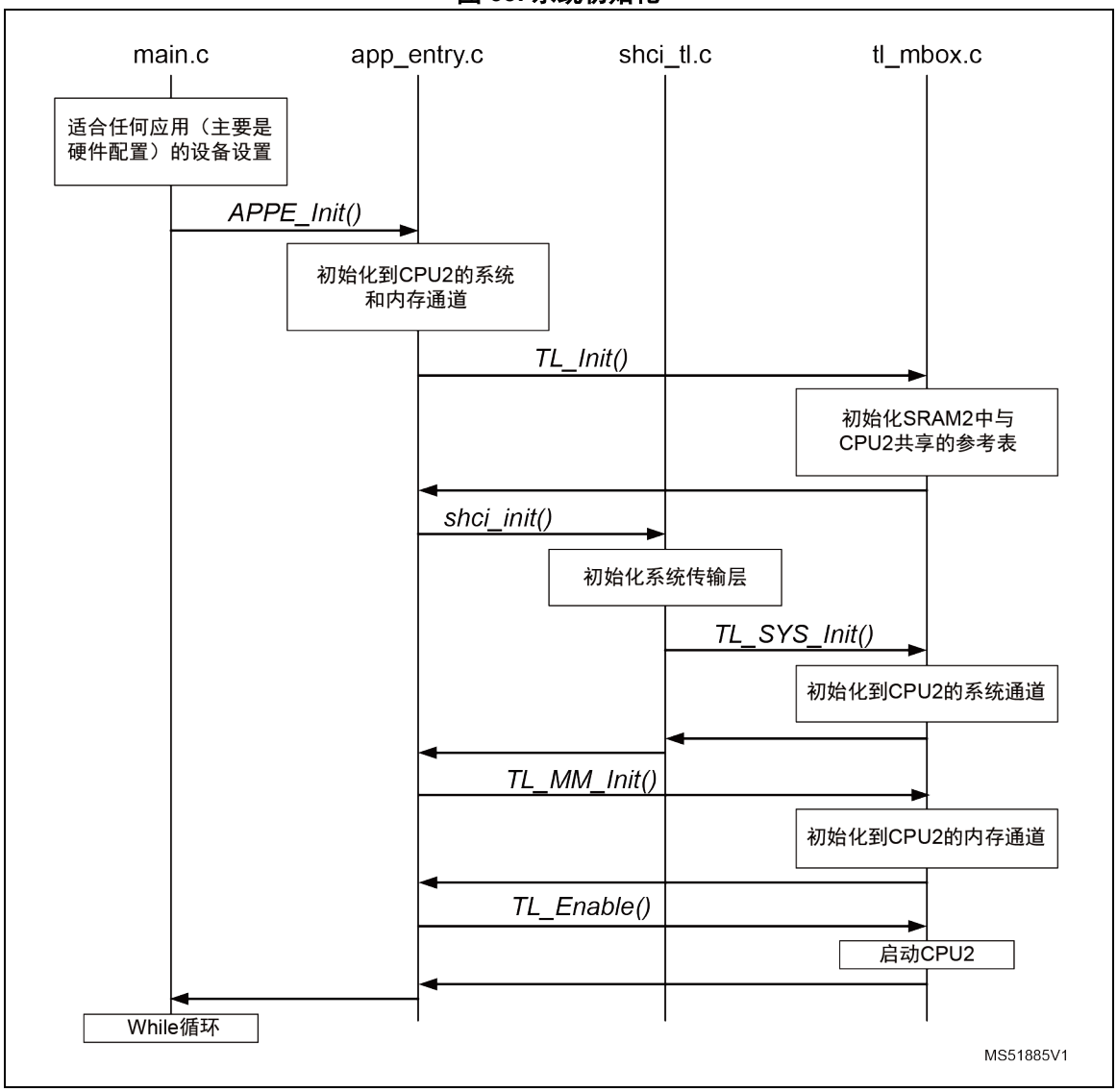


13 附录

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 63. 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 64. 系统就绪事件通知

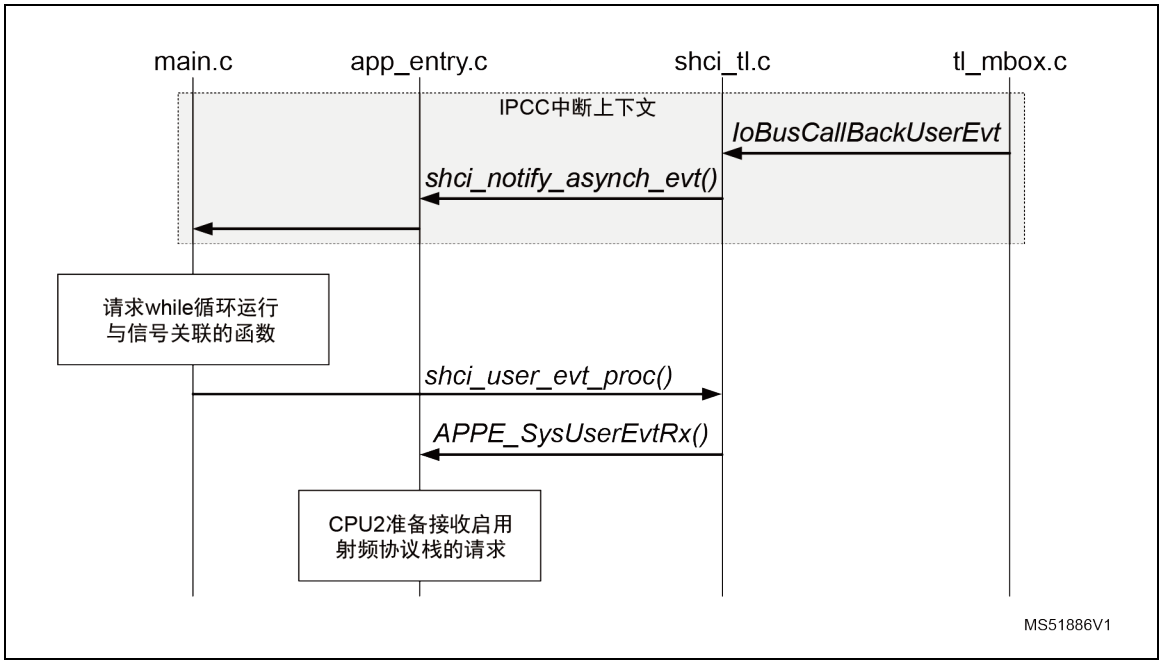
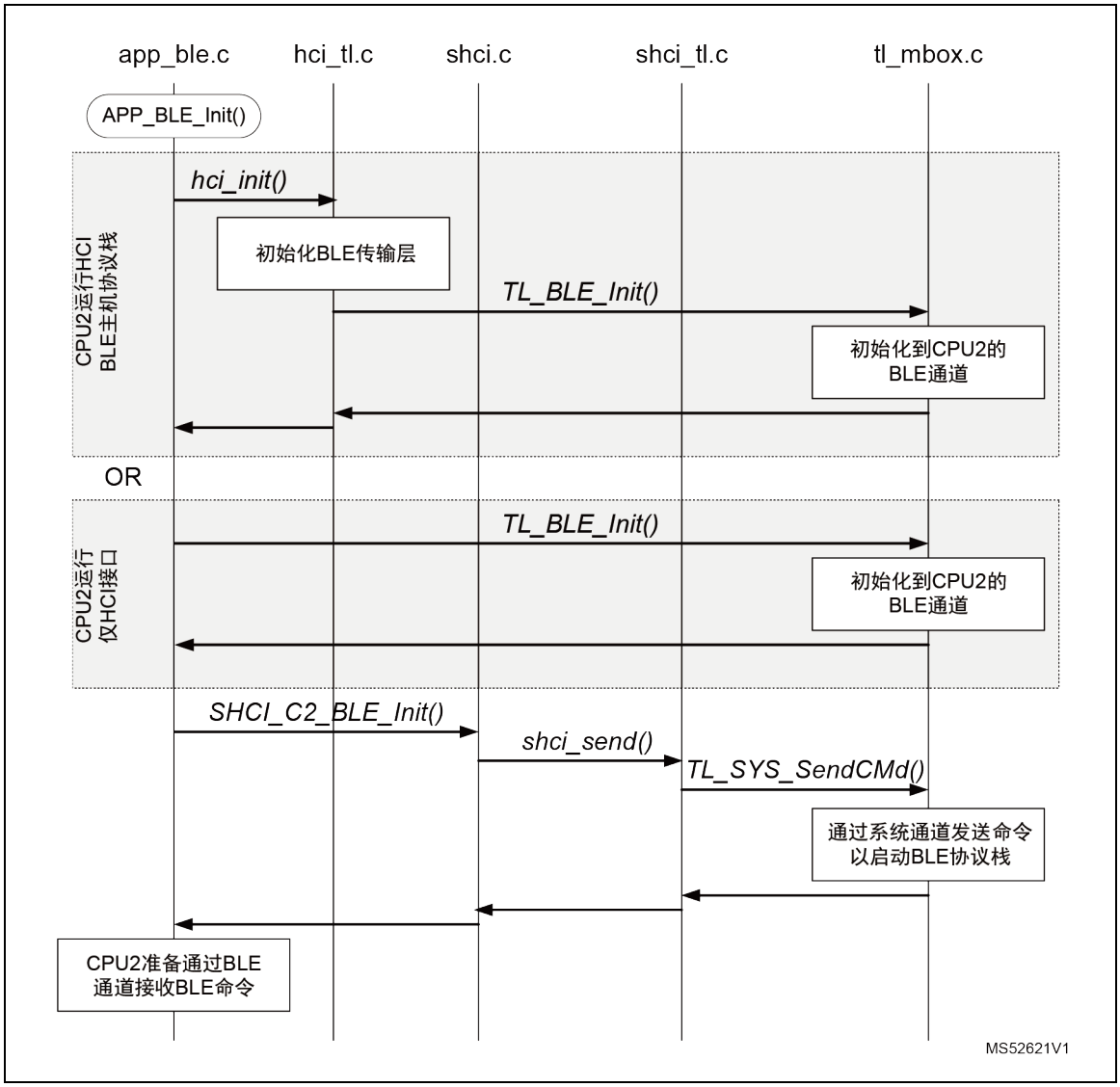


图 65. BLE 初始化



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 邮箱（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 接口之上使用开源 BLE 协议栈时必须使用。CPU2 必须在 1 秒内响应所有命令。

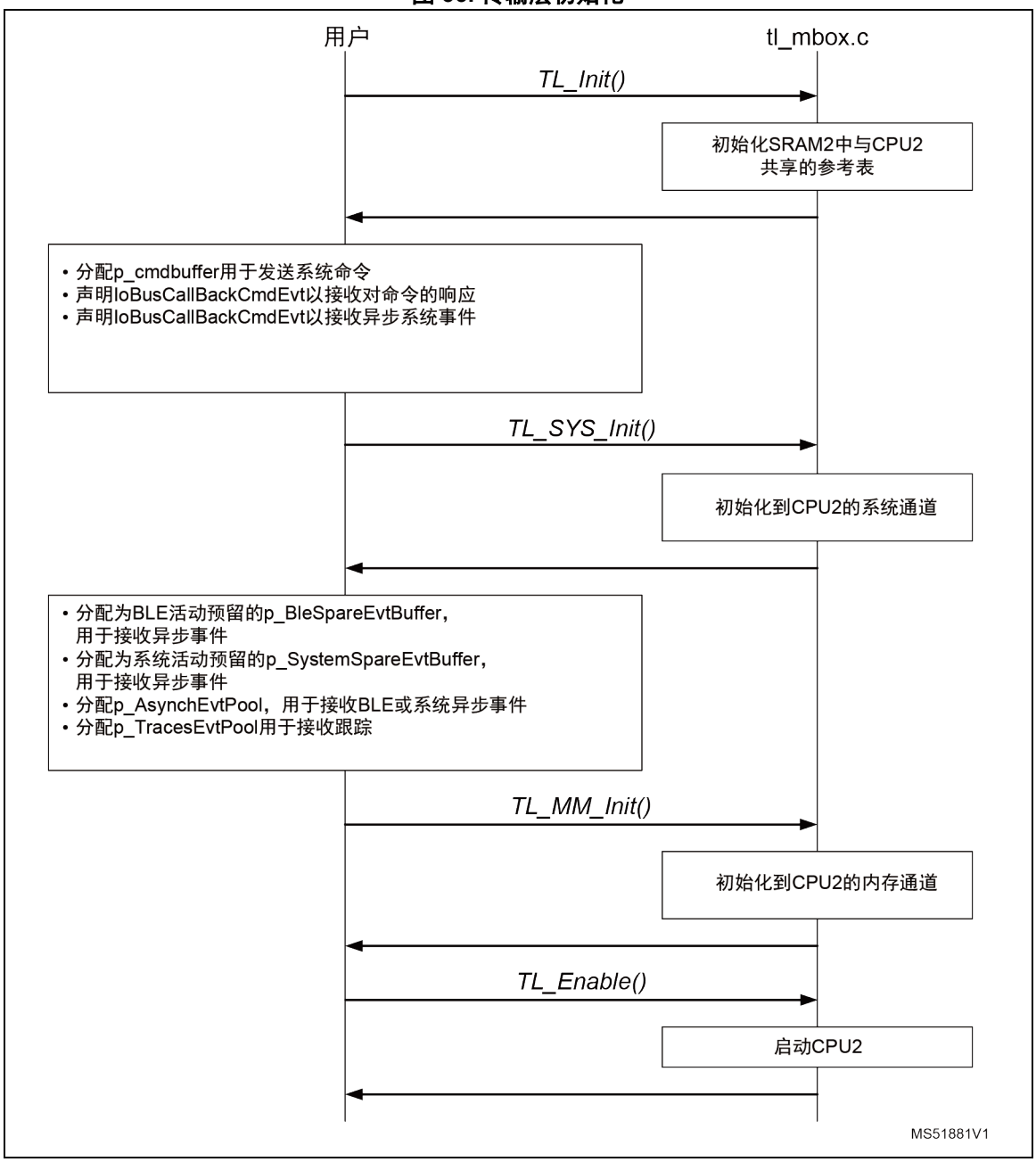
13.2.1 接口 API

表 30. 接口 API

函数	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 66. 传输层初始化



void TL_Init(void):

这是要发送的第一个命令。它初始化邮箱（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邮箱（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。

此命令初始化邮箱（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邮箱（mailbox）驱动程序专门用来报告 BLE 异步事件（p_BleSpareEvtBuffer）的缓冲区、要被邮箱（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。

此命令初始化邮箱（mailbox）驱动程序中的内存通道。

void TL_Enable(void):

在完全初始化邮箱（mailbox）驱动程序后，发送此命令启动 CPU2。

图 67. BLE 通道初始化

int32_t TL_BLE_Init(void* pConf):

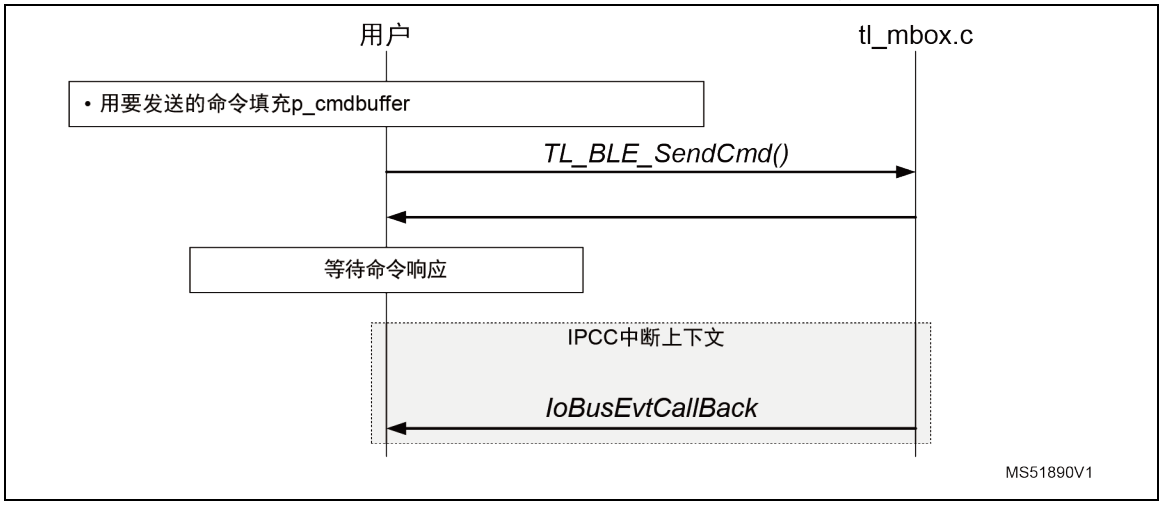
用户必须首先分配要被邮箱（mailbox）驱动程序用来发送 BLE 命令（p_cmdbuffer）的缓冲区、要被邮箱（mailbox）驱动程序用来发送 ACL 数据包（p_AclDataBuffer）的缓冲区以及要用来接收 BLE 事件（IoBusEvtCallBack）和 ACL 数据包确认（IoBusAclDataTxAck）的两个回调函数。

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 68. 通过邮箱（mailbox）发送的 BLE 命令

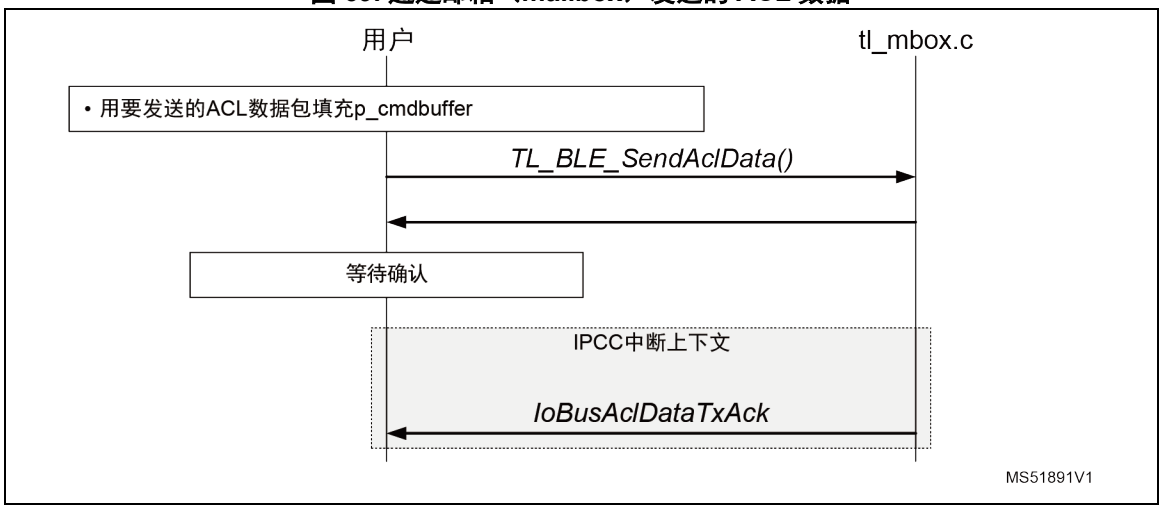


int32_t TL_BLE_SendCmd(uint8_t* buffer, uint16_t size):

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

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 69. 通过邮箱（mailbox）发送的 ACL 数据



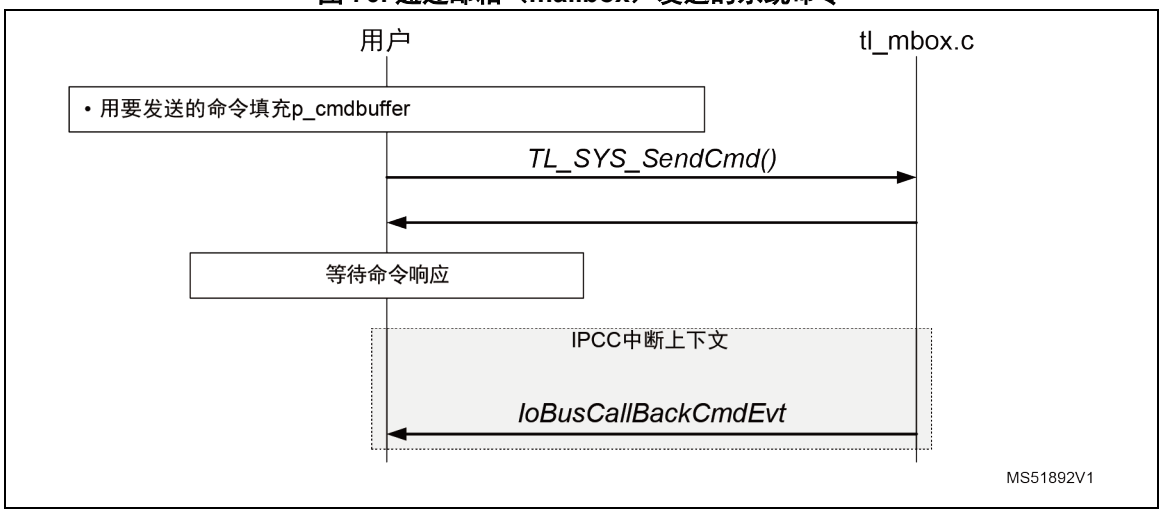
int32_t TL_BLE_SendAclData(uint8_t* buffer, uint16_t size):

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 70. 通过邮箱（mailbox）发送的系统命令

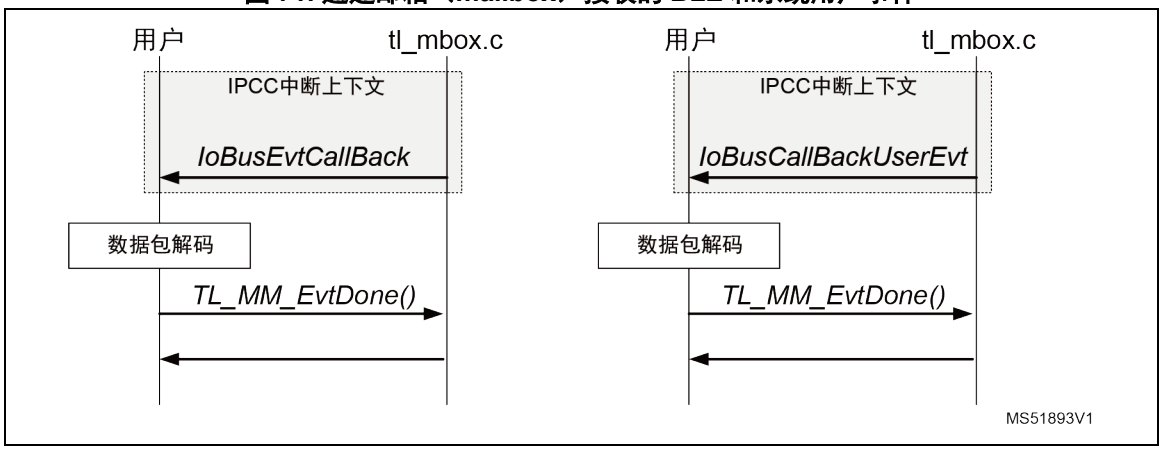


int32_t TL_SYS_SendCmd(uint8_t* buffer, uint16_t size)

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 71. 通过邮箱（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 邮箱（Mailbox）接口 - 扩展

当要发送的命令被用户构建到要通过邮箱发送的缓冲区中时，适合使用邮箱（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，当只使用邮箱（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 邮箱（Mailbox）接口与更高级别的 SHCI 接口混合使用，以便在连接到系统邮箱（Mailbox）接口时编码/解码系统数据包。这就是“邮箱（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邮箱（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邮箱（mailbox）驱动程序。

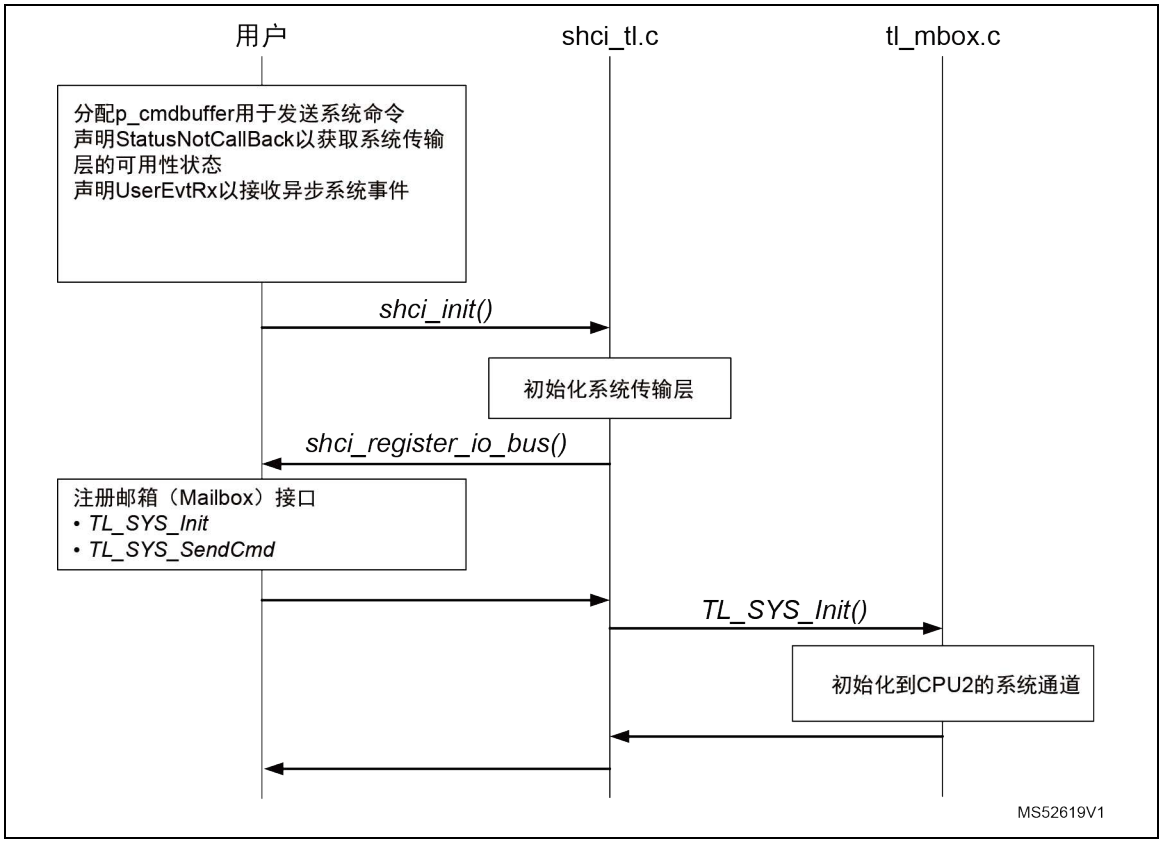
两个 API TL_SYS_Init() 和 TL_SYS_SendCmd() 以及两个回调 IoBusCallBackCmdEvt 和 IoBusCallBackUserEvt 均在传输层使用和实现， 不能再单独使用。

表 31. 接口 API

函数	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)	将邮箱（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 72. 系统传输层初始化



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

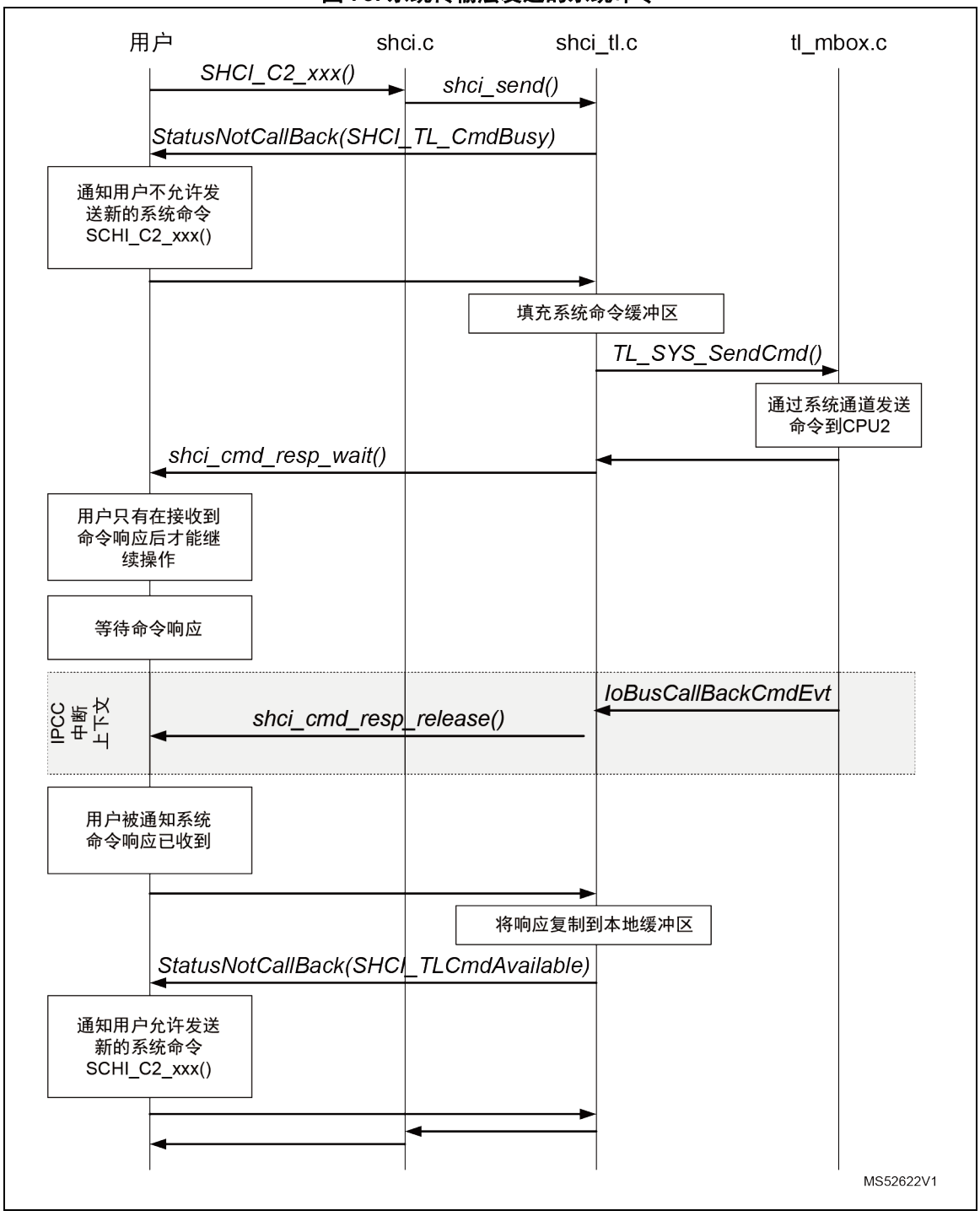
用户必须首先分配要被邮箱（mailbox）驱动程序用来发送系统命令（p_cmdbuffer）的缓冲区，以及要用来接收用户异步系统事件（UserEvtRx）和传输层可用性通知（StatusNotCallBack）的两个回调。

此命令初始化传输层和邮箱（mailbox）驱动程序中的系统通道。

void shci_register_io_bus(tSHciIO* fops);:

此命令将邮箱（mailbox）驱动程序注册到系统传输层。

图 73. 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线程发送系统命令的多线程应用中。

当 status = 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。

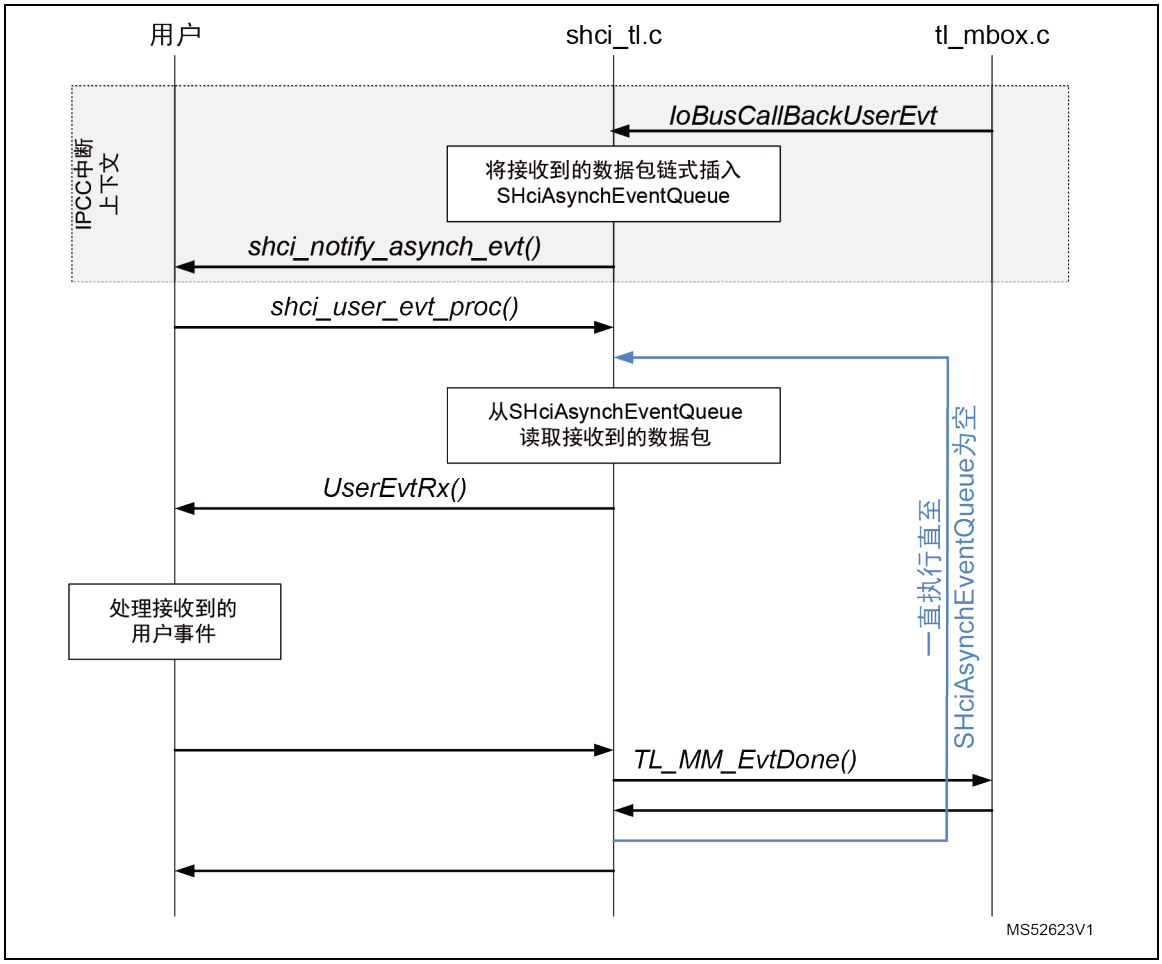
void shci_cmd_resp_release(uint32_t flag):

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 74. 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。

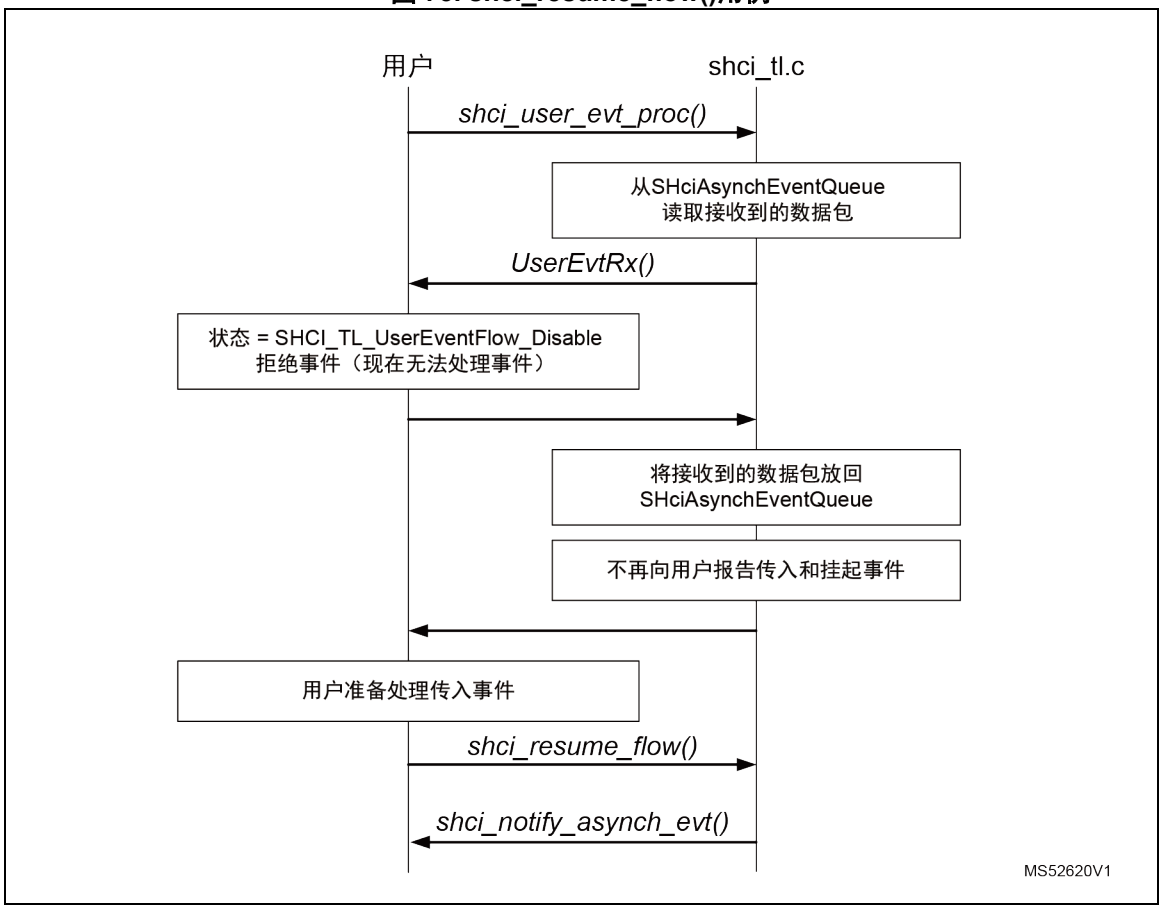
pData 是保存下列参数的结构地址：

```
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 75. 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，系统传输层不释放系统事件，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 接口与邮箱（mailbox）之间收发命令。

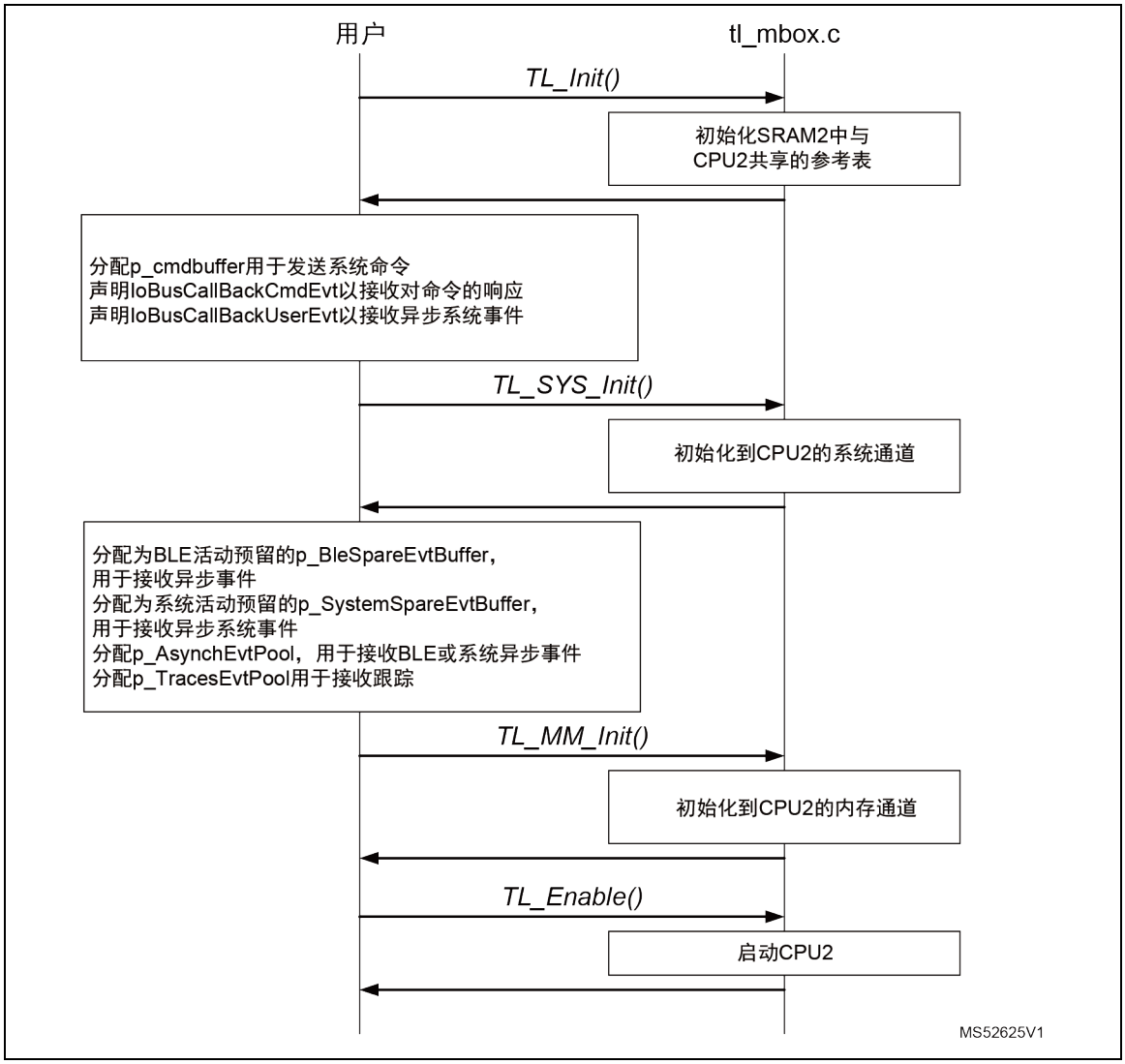
新接口基本为 *“邮箱（Mailbox）接口 - 扩展”*，此处实现了 HCI 传输层。使用 ACI 接口时，应用程序不再使用低级邮箱（Mailbox）接口。

表 32. BLE 传输层接口

函数	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);	将邮箱（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 76. BLE 传输层初始化



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

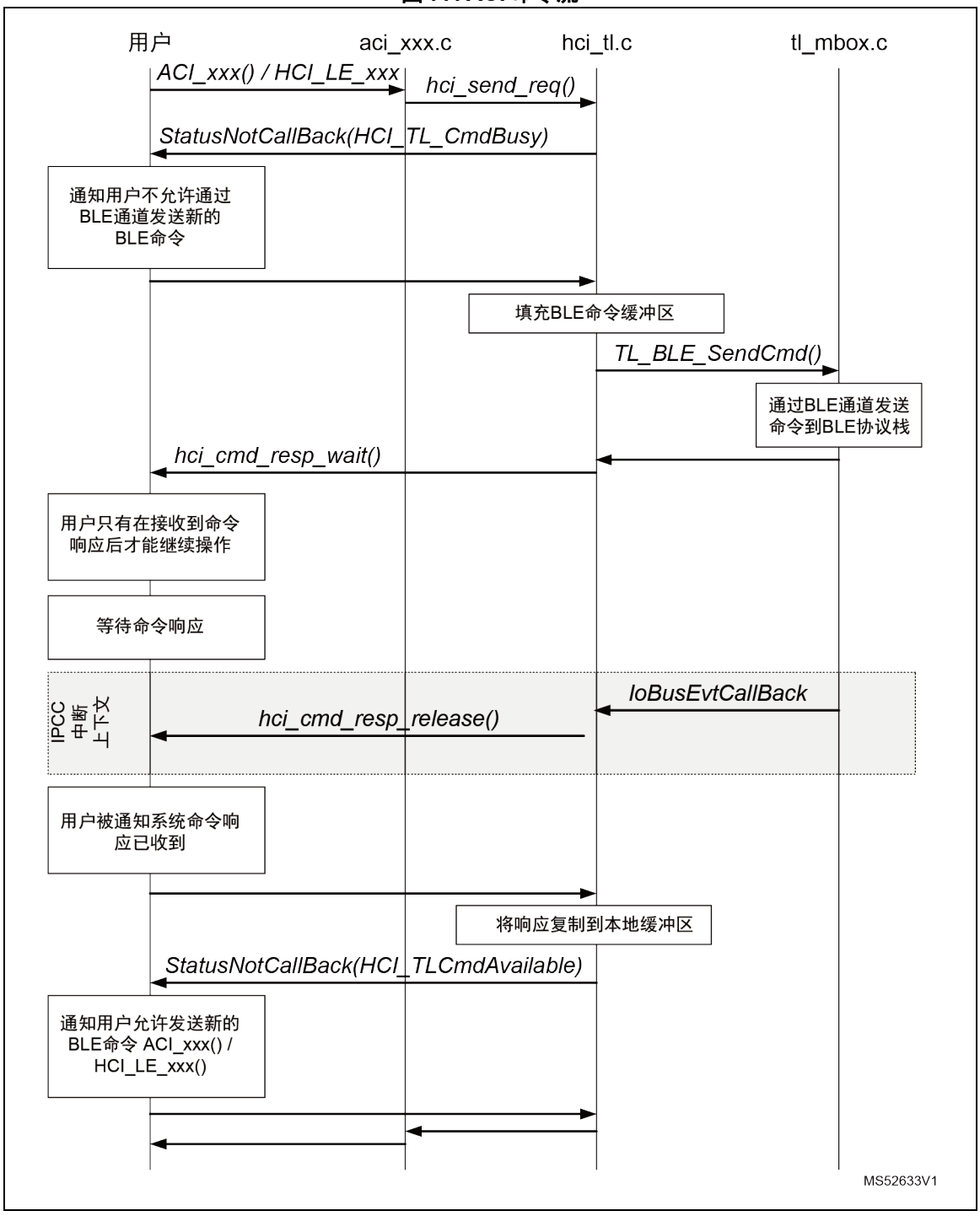
用户必须首先分配要被邮箱（mailbox）驱动程序用来发送 BLE 命令（p_cmdbuffer）的缓冲区，以及要用来接收用户异步系统事件（UserEvtRx）和传输层可用性通知（StatusNotCallBack）的两个回调。

此命令初始化 HCI 传输层和邮箱（mailbox）驱动程序中的 BLE 通道。

void hci_register_io_bus(tSHciIO* fops);:

此命令将邮箱（mailbox）驱动程序注册到 HCI 传输层。

图 77. 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 status = 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。

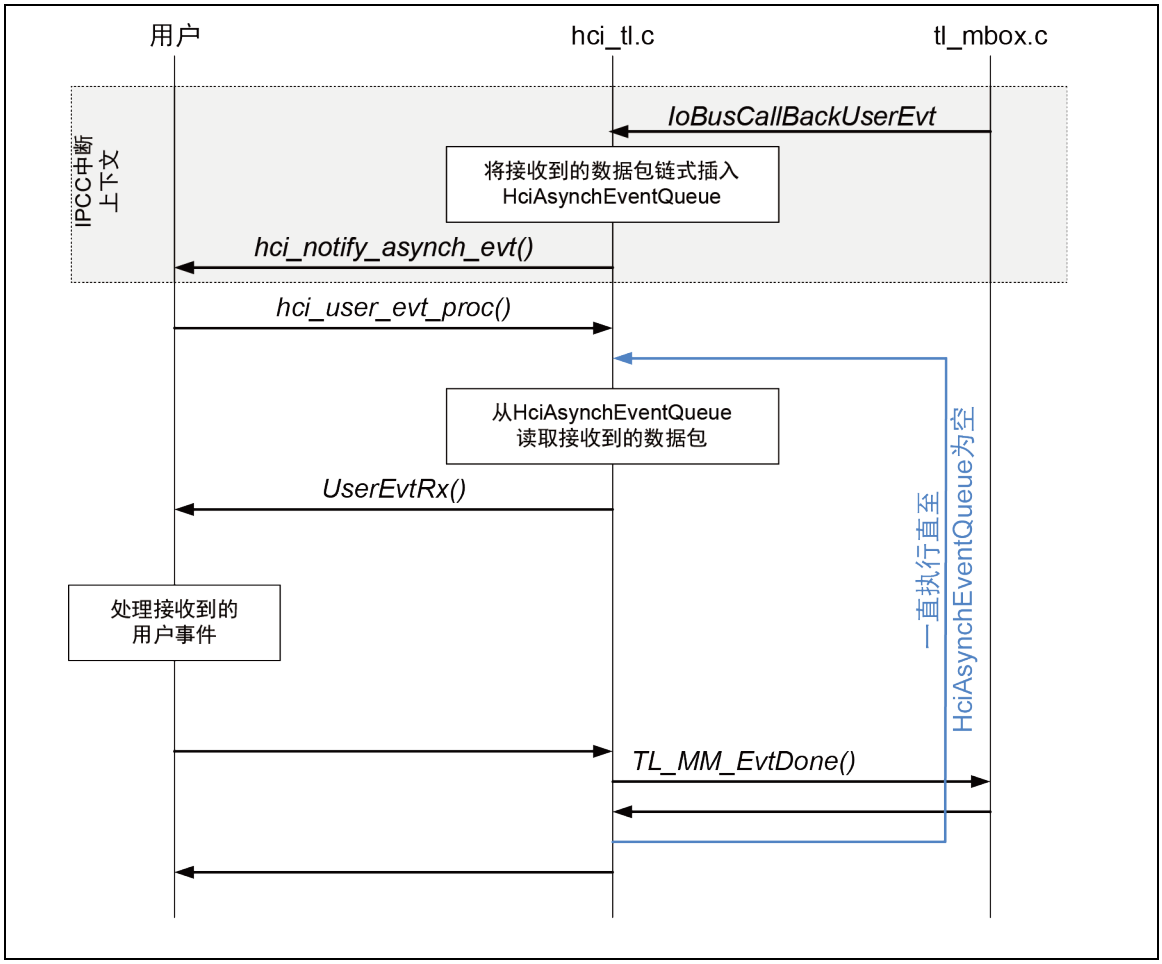
void hci_cmd_resp_release(uint32_t flag):

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 78. 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。

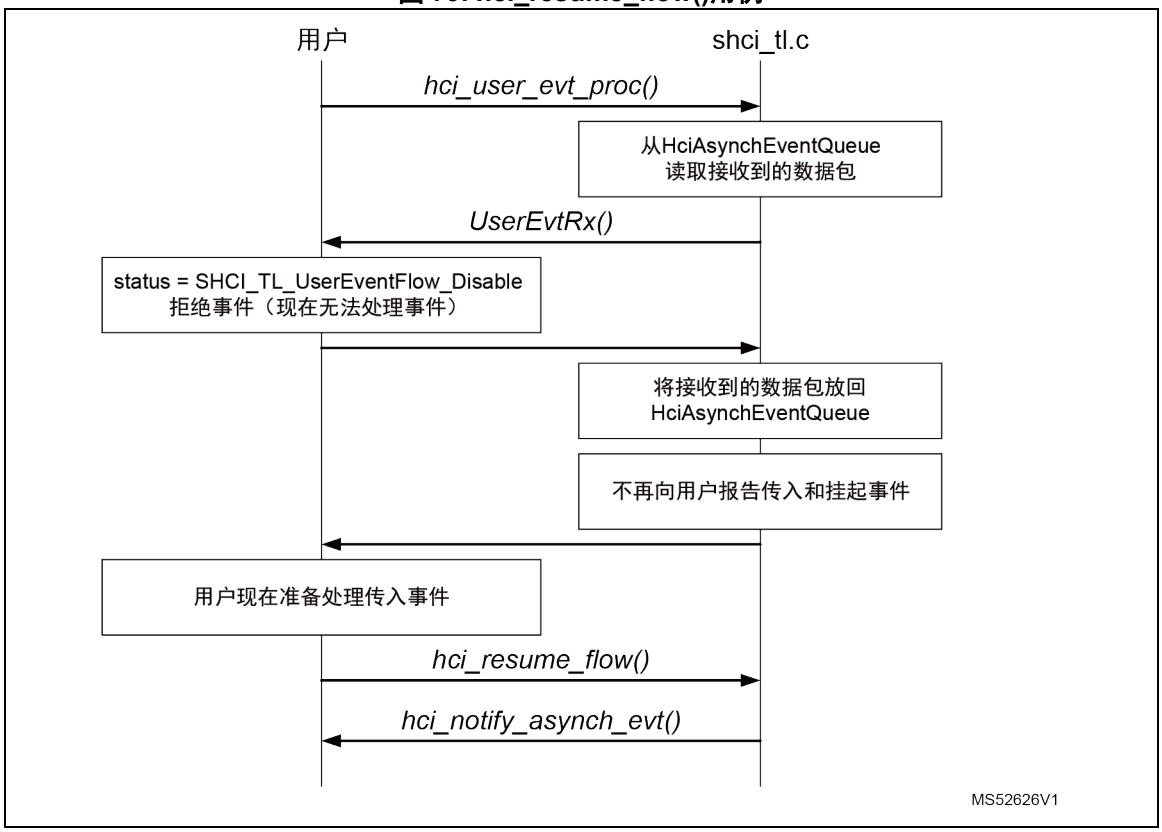
pData 是保存下列参数的结构地址：

```
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 79. 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 status 参数设置为 HCI_TL_UserEventFlow_Disable。在这种情况下，HCI 传输层不释放 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 命令

表 33. 安全接口命令(1)

指令	代码	说明
SHCI_C2_FUS_GetState()	0xFC52	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。请参考*第 7.6.9 节：如何尽可能提高数据吞吐量*，以了解更多细节。
SHCI_C2_THREAD_Init()	0xFC67	请参考*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 Flash 存储器擦除操作。这将允许 CPU2 针对擦除操作启用时序保护。
SHCI_C2_CONCURRENT_SetMode()	0xFC6A	请参考*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	请参考*第 12.4.5 节：MAC IEEE Std 802.15.4-2011 system*。
SHCI_C2_Reinit()(2)	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 上运行的射频协议栈和 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(1)（接上页）

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 Flash 存储器操作的影响。如果不发送此命令，CPU2 将使用 PESD。
SHCI_C2_LLD_BLE_Init	0xFC74	初始化 CPU2 上的 BLE LLD 接口。
SHCI_C2_Config	0xFC75	发送系统配置到 CPU2。非强制。

1. 关于系统命令描述的更多详细信息可以在 STM32WB 固件包中的包头 shci.h 中找到。
2. 详细描述如下表所示。

SHCI_C2_Reinit()

在复位时，CPU2 使用 TL_Enable() 命令启动，当它完成初始化后，会报告 SHCI_SUB_EVT_CODE_READY 系统事件。一旦启动，它就不能进行复位以重新启动它的启动顺序。

因此，当应用程序从已经启动了 CPU2 的主引导应用程序启动时，应用程序永远不会收到 SHCI_SUB_EVT_CODE_READY 系统事件，因为该事件已经报告给了主引导应用程序。

SHCI_C2_Reinit()必须由主引导应用程序（如果它已经启动了 CPU2）在跳转到将要接收 SHCI_SUB_EVT_CODE_READY 系统事件的主应用程序之前发送。

当 CPU2 收到 SHCI_C2_Reinit()命令，将执行以下步骤

- 执行 SEV()和 WFE()指令以设置和清除事件寄存器
- 为 line41 启用 EXTI 上升沿（CPU2 的 C1SEV 中断）
- 将命令响应发送到 CPU1
- 执行 WFE 并等待 CPU1 事件
- 从 WFE 唤醒时，重新启动启动代码

该命令不复位硬件，但要求 CPU2 从其复位向量重新启动。

13.5.2 事件

表 34 中列出的事件可以使用 SHCI_C2_Config()系统命令来启用/禁用

表 34. 用户系统事件

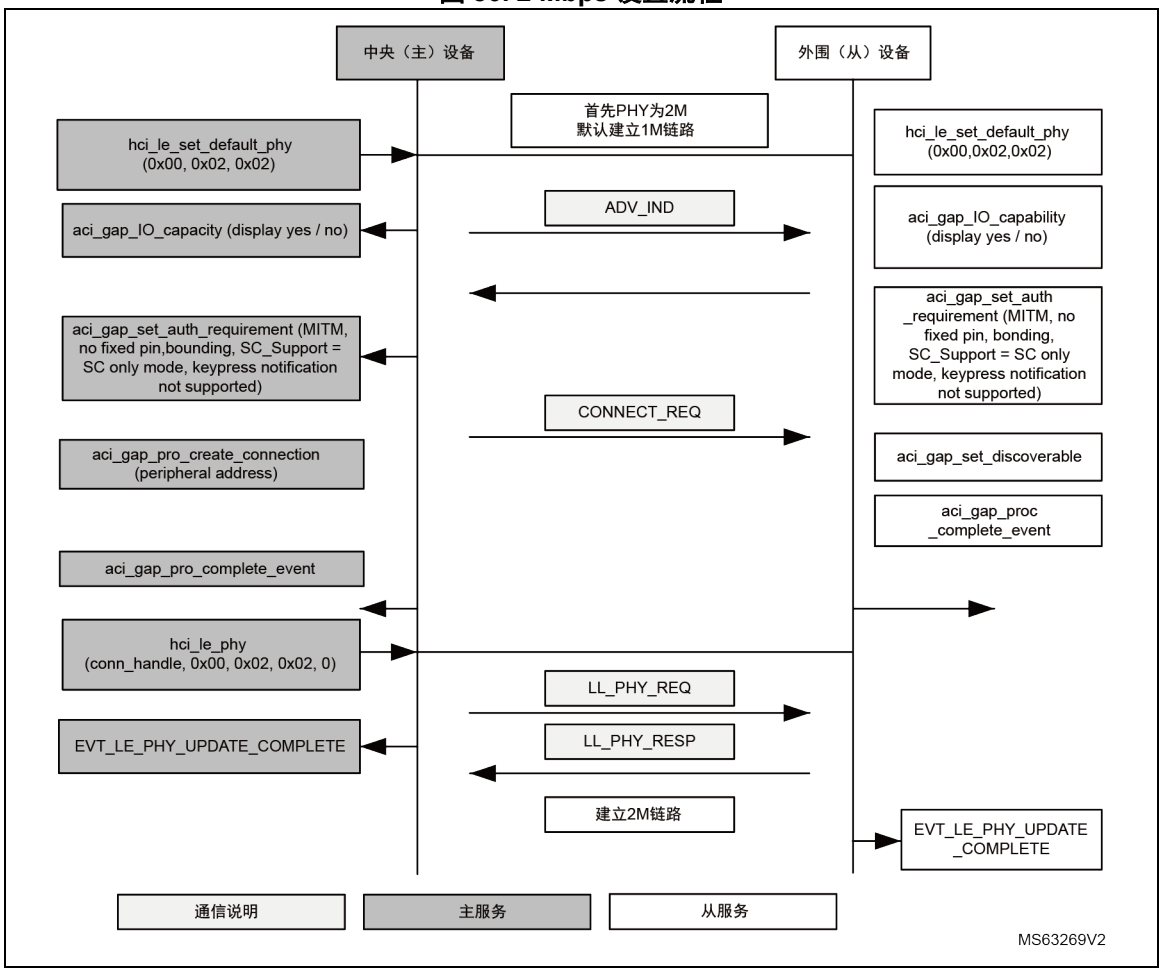
事件	代码	说明
SHCI_SUB_EVT_CODE_READY	0x9200	在 CPU2 启动并准备接收命令后立即返回。
SHCI_SUB_EVT_ERROR_NOTIF	0x9201	报告 CPU2 中的错误。
SHCI_SUB_EVT_BLE_NVM_RAM_UPDATE	0x9202	当 CPU2 在 SHCI_C2_Config()命令请求时将 BLE NVM 数据写入 SRAM 时返回。
SHCI_SUB_EVT_THREAD_NVM_RAM_UPDATE	0x9203	当 CPU2 在 SHCI_C2_Config() 命令请求时将 THREAD NVM 数据写入 SRAM 时返回。
SHCI_SUB_EVT_NVM_START_WRITE	0x9204	当 CPU2 开始对 CPU2 进行 Flash 存储器写入操作时返回。
SHCI_SUB_EVT_NVM_END_WRITE	0x9205	当 CPU2 向 CPU2 的 Flash 存储器成功写入数据时返回。
SHCI_SUB_EVT_NVM_START_ERASE	0x9206	当 CPU2 开始对 CPU2 进行 Flash 存储器擦除操作时返回。
SHCI_SUB_EVT_NVM_END_ERASE	0x9207	当 CPU2 成功擦除 CPU2Flash 存储器中的数据时返回。

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 80*。

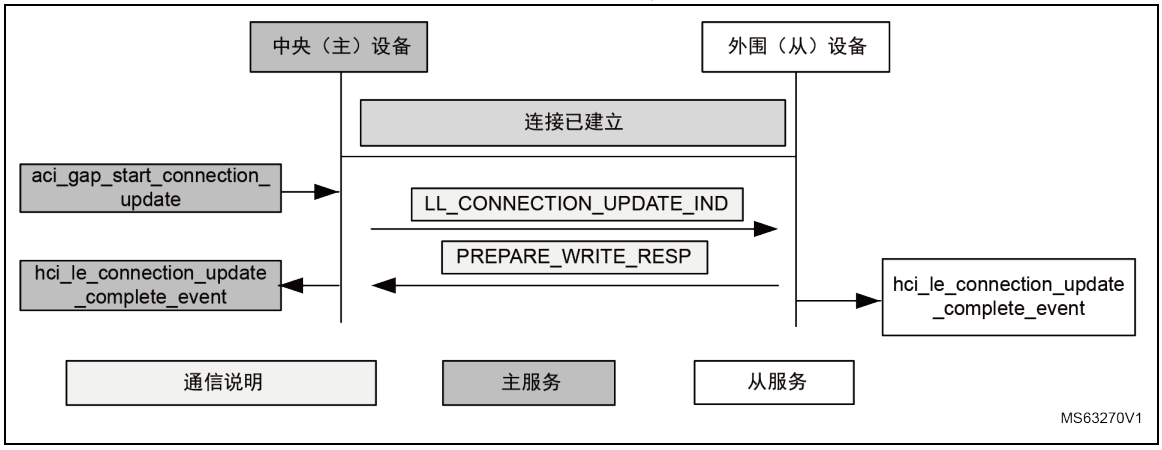
图 80. 2 Mbps 设置流程



13.7 BLE - 连接更新流程

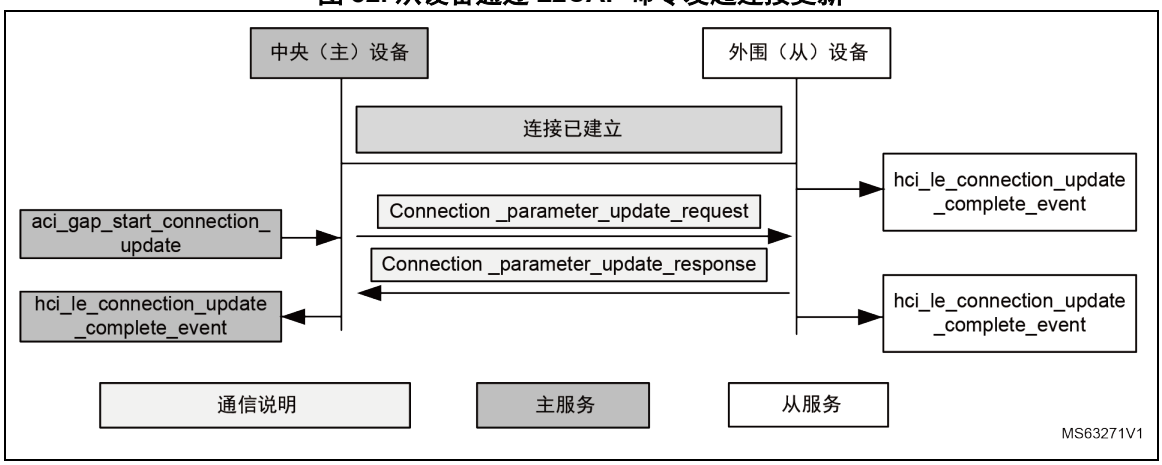
在建立连接后，主设备可以用 aci_gap_start_connection_update 命令更新连接参数。

图 81. 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 82. 从设备通过 L2CAP 命令发起连接更新



13.8 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 83. 数据包结构

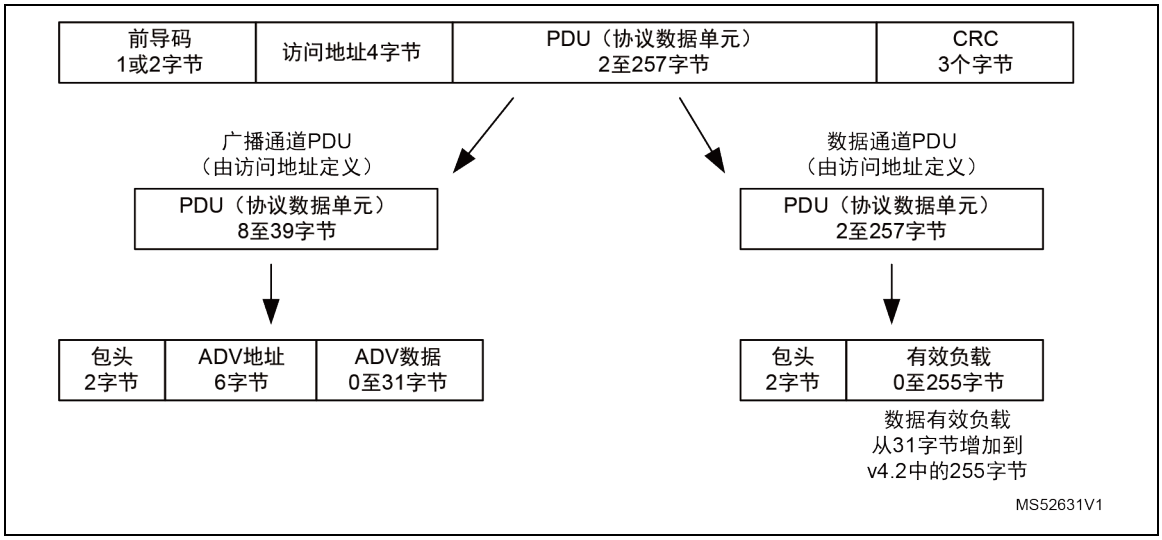
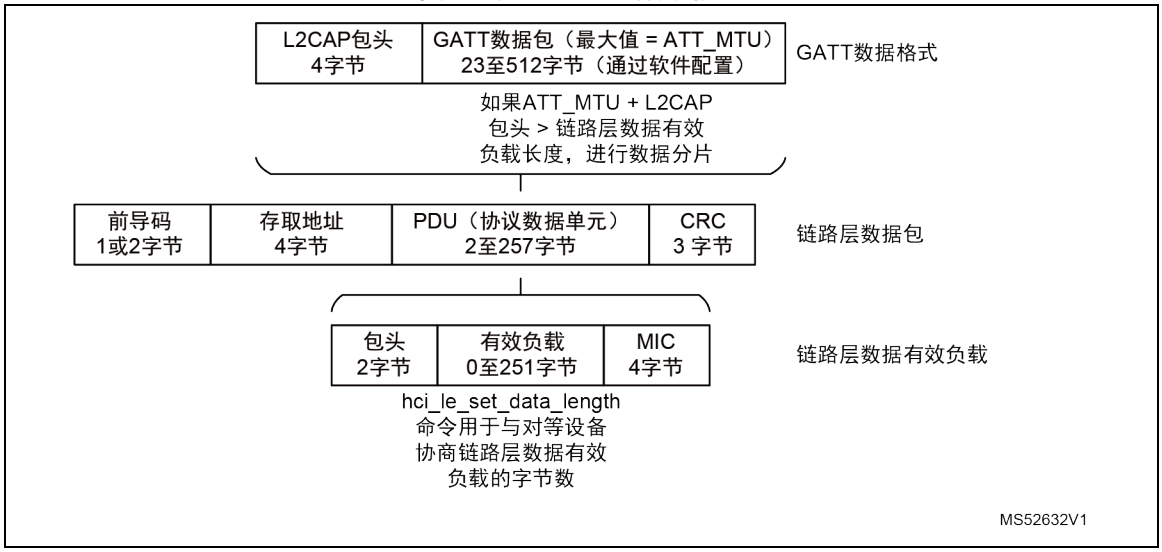


图 84. 应用 GATT 数据格式



13.9 Thread 概述

13.9.1 引言

Thread 协议栈是可靠、高性价比、低功耗的射频 D2D 通信的开放标准。它专为需要基于 IP 的网络且可以在协议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。

13.9.2 主要特征

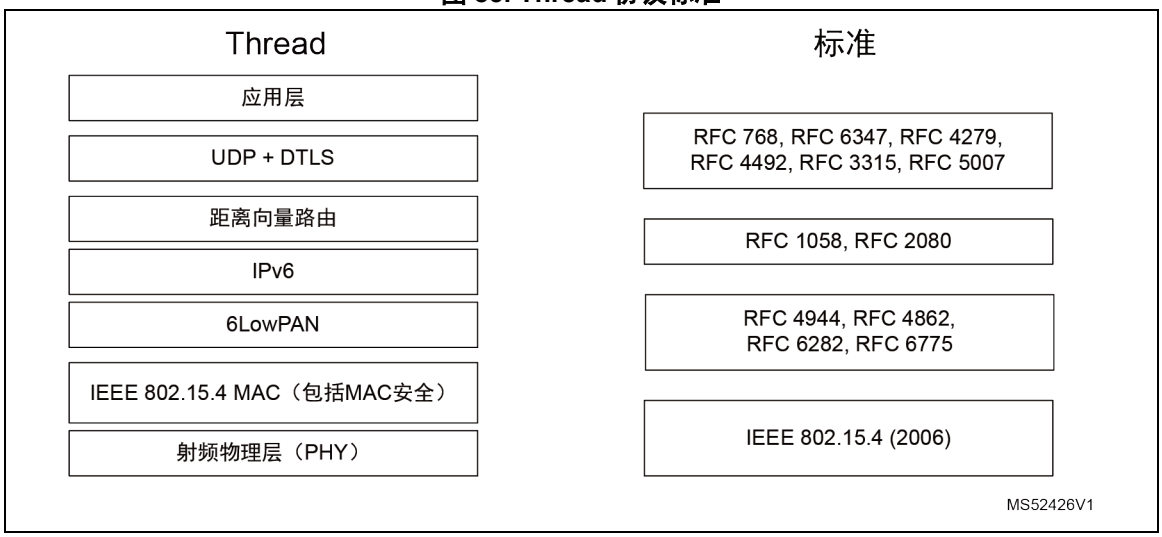
Thread 面向智能家居应用，例如环境控制、恒温器、报警、能源管理、智能锁和智能照明设备。此标准的一个主要优势是它基于 IPv6，因此可以轻松地将任何 Thread 网络连接到任何其他 IPV6 应用。它的另一大优势是它基于真正的 Mesh 网络。此网络一经部署，将十分稳健可靠。例如，当路由失败时，通过找到通往目的地的新路径，系统能够完成自身的自动重新配置。通过 Mesh 网络，设备之间可以进行更长距离的通信。

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。

13.9.3 协议层

Thread 基于成熟并经过充分验证的标准，如*图 85* 所示。

图 85. Thread 协议标准

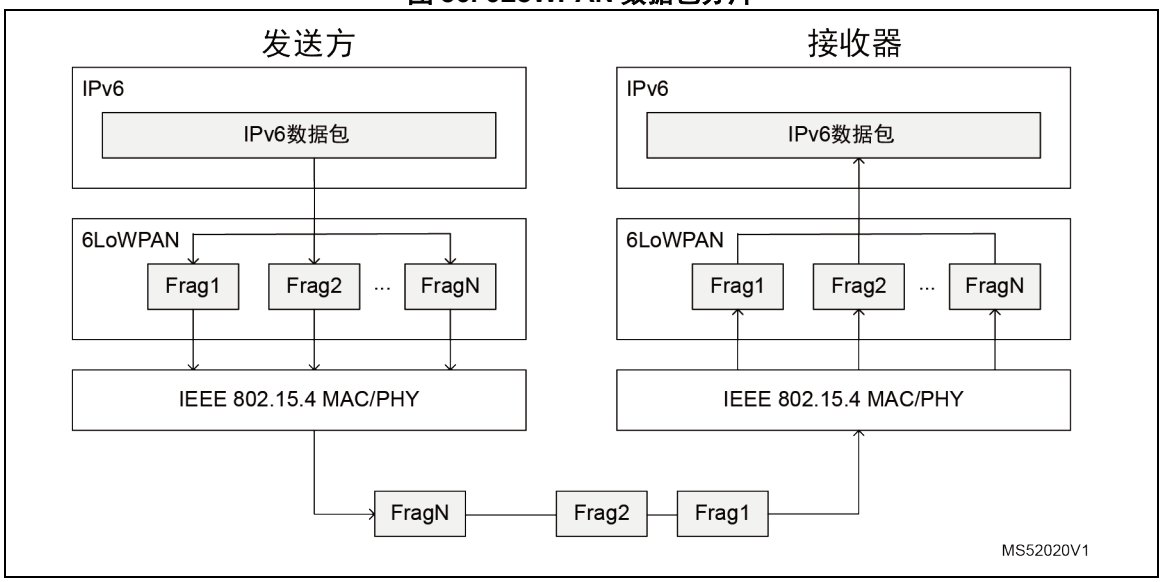


从底层开始：

- 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 射频层使用 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 的低功耗射频个域网”。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 86. 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。

13.9.4 网状拓扑

Thread 支持 Mesh 网络。如*图 87* 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配网服务
  - 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 87. Thread 网络拓扑

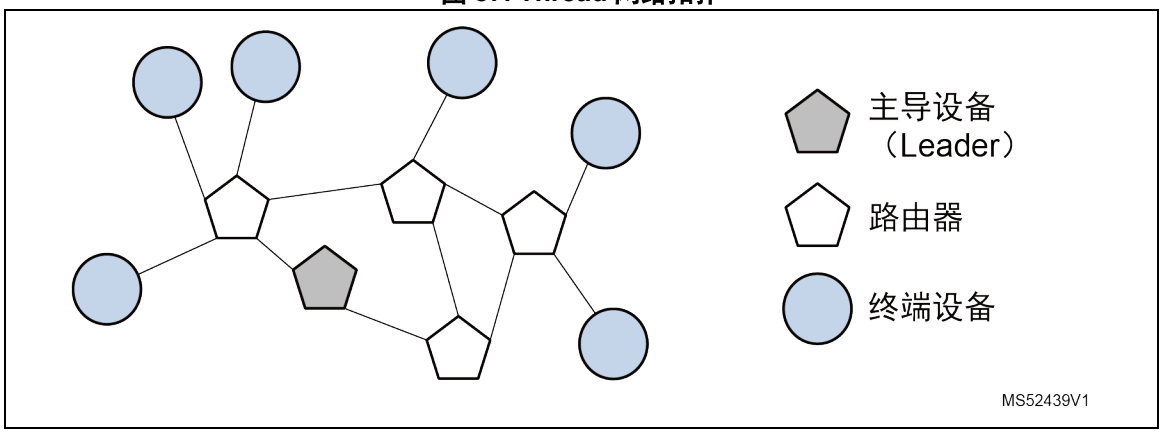
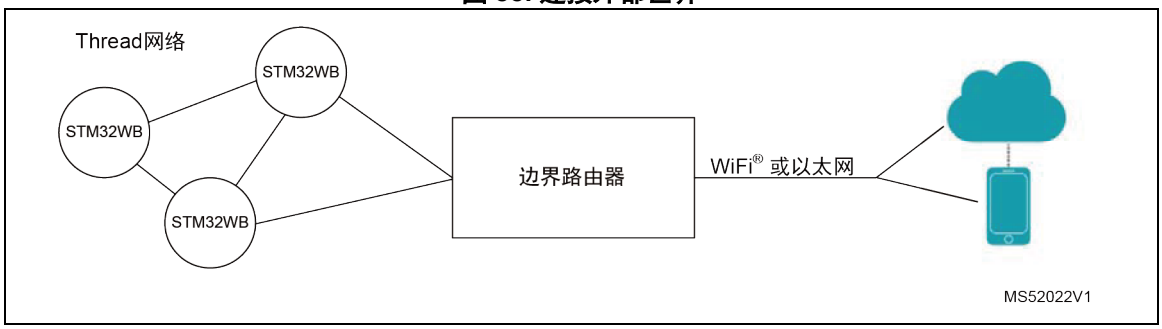


图 88. 连接外部世界



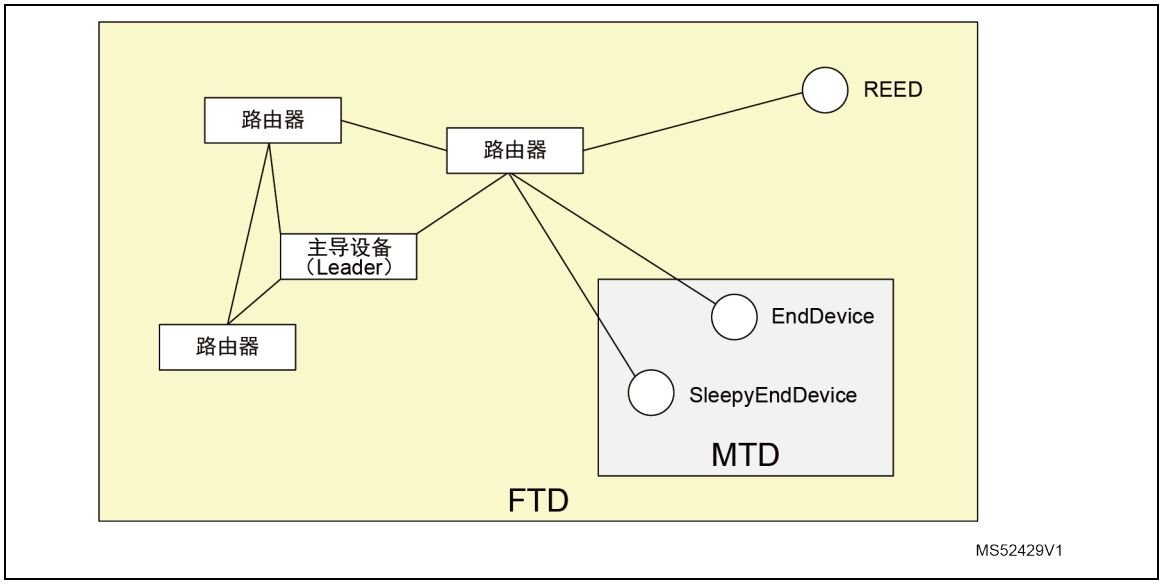
13.9.5 Thread 配置

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 Flash）。MTD 只充当终端设备或休眠终端设备。

图 89. 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*“前言”*，*第 4.2 节：内存映射*，*第 4.3 节：共享外设*，*第 9.2 节：如何启动*，*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*，以及*第 12.4.3 节：MAC 应用处理器固件*。 增加了*第 11.8 节：Thread FUOTA* 及其分段。 更新了*图 4：存储器映射*。 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*，以及*第 7.6.1 节：如何设置蓝牙设备地址*。 增加了*第 4.7 节：Flash 存储器管理*，*第 4.8 节：CPU 的调试信息*，*第 4.9 节：FreeRTOS 低功耗*及其分段。 更新了*表 2：信号量*，*表 33：系统接口命令*，以及*表 34：用户系统事件*。 更新了*图 10：在 Flash 存储器中写入/擦除数据的算法*，*图 22：心率项目 - 中间件与用户应用之间的交互*，*图 27：P2P 服务器软件通信*。 对整个文档进行少量文字修订。
2020 年 10 月 20 日	4	更新了*第 4.3 节：共享外设*，*第 5 节：系统初始化*，*第 7.6.5 节：如何启动 BLE 协议栈 - SHCI_C2_BLE_Init()*，*第 10.3.1 节：创建 otCoapResource*，以及*第 13.5.1 节：指令*。 更新了*图 10：在 Flash 存储器中写入/擦除数据的算法*。 更新了*表 33：系统接口命令*，以及*表 34：用户系统事件*。 增加了*第 4.10 节：设备信息表*，以及*第 5.2 节：CPU2 启动*。
2021 年 4 月 20 日	5	更新了*“前言”*，*第 1 节：参考文献*，*第 4.3 节：共享外设*，*第 7.6.5 节：如何启动 BLE 协议栈 - SHCI_C2_BLE_Init()*，*第 7.6.9 节：如何尽可能提高数据吞吐量*，以及*第 13.2 节：邮箱（Mailbox）接口*。 更新了*图 1：STM32WB 系列微控制器支持的协议*，*图 37：软件架构*，以及*图 78：BLE 用户事件接收流程*。 更新了*表 1：STM32WB 系列微控制器支持的栈*，*表 2：信号量*，*表 9：安全命令*，以及*表 28：Thread 可以使用的 M0 固件*。 增加了*第 4.11 节：ECCD 错误管理*。 删除了之前的*第 6.10 节：写入或读取长本地或远程值*，*第 13.8 节：BLE - 安全程序*及其子分段。

表 35. 文档版本历史

日期	版本	变更
2021 年 12 月 14 日	6	增加了*第 7.6.2 节：如何设置 IRK（身份根密钥）和 ERK（加密根密钥）*，*第 7.6.6 节：NVM 中的 BLE GATT DB 和安全记录*，以及*第 7.6.7 节：如何计算 NVM 中可以存储的最大绑定设备数量*。 更新了*第 7.6.1 节：如何设置蓝牙设备地址*，*CFG_BLE_ATT_VALUE_ARRAY_SIZE*，*第 13.2 节：Mailbox interface*，*SHCI_C2_xxx()*，以及 *ACI_xxx() / HCI_LE_xxx()*。 更新了*表 31：接口 API*。
2022 年 7 月 15 日	7	更新了*第 4.3 节：共享外设*，*第 4.6 节：低功耗管理器*，以及*第 7.2.2 节：STM32WB 心率传感器应用 - 中间件应用*。 增加了*第 7.6.8 节：NVM 写访问*。 更新了*图 22：心率项目 - 中间件与用户应用之间的交互*。 对整个文档进行少量文字修订。
2022 年 11 月 24 日	8	更新了 *CFG_BLE_OPTIONS*。 增加了*第 7.6.11 节：如何从 32 MHz 切换到 64 MHz*，以及*第 7.6.12 节：如何在退出低功耗模式时重新使能 PLL*。 对整个文档进行少量文字修订。