Programming manual

STM32WB BLE 堆栈编程指南

介绍

本文档的主要目的是为开发人员提供一些有关如何使用 STM32WB BLE 堆栈 API 和相关事件回调开发低功耗蓝牙 (BLE) 应用程序的参考编程指南。

该文档描述了 STM32WB 蓝牙低功耗堆栈库框架、API 接口和事件回调，允许访问 STM32WB 片上系统提供的蓝牙低功耗功能。

本编程手册还提供了一些关于低功耗蓝牙 (BLE) 技术的基本概念，以便将 STM32WB BLE 堆栈 API、参数和相关事件回调与 BLE 协议堆栈功能相关联。用户必须对 BLE 技术及其主要特性有基本的了解。

有关 STM32WB 系列和低功耗蓝牙规范的更多信息，请参阅本文档末尾的第 6 节参考文档。

STM32WB 是一款超低功耗蓝牙低功耗 (BLE) 单模网络处理器，符合蓝牙规范 v5.2，支持主从角色。

本手册的结构如下：

- 低功耗蓝牙 (BLE) 技术的基础知识
- STM32WB BLE 堆栈库 API 和事件回调概述
- 如何使用 STM32WB 库 API 和事件回调设计应用程序（使用“switch case”事件处理程序而不是使用事件回调框架给出了一些示例）

1 General information

This document applies to the STM32WB Series dual-core Arm®-based microcontrollers.

Note: *Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.*

2 蓝牙低功耗技术

蓝牙低功耗 (BLE) 无线技术由蓝牙特别兴趣小组 (SIG) 开发，以实现使用纽扣电池运行数年的超低功耗标准。

经典蓝牙技术是作为一种无线标准而开发的，允许更换连接便携式和/或固定电子设备的电缆，但由于其快速跳跃、面向连接的行为和相对复杂的连接过程，它无法实现极端水平的电池寿命。

蓝牙低功耗设备仅消耗标准蓝牙产品的一小部分功率，并使带有纽扣电池的设备能够无线连接到标准蓝牙设备。

Figure 1. 启用蓝牙低功耗技术的纽扣电池设备

蓝牙低功耗技术广泛用于传输少量数据的传感器应用：

- 汽车
- 运动和健身
- 卫生保健
- 娱乐
- 家庭自动化
- 安全性和邻近性

2.1 BLE 堆栈架构

蓝牙低功耗技术已被蓝牙核心规范 4.0 版正式采用（参见第 6 节参考文档）。

低功耗蓝牙技术在 2.4 至 2.485 GHz 的未经许可的工业、科学和医疗 (ISM) 频段中运行，该频段在大多数国家/地区都可用且未经许可。 它使用扩频、跳频、全双工信号。 蓝牙低功耗技术的主要特点是：

- 稳健性
- 表现
- 可靠性
- 互操作性
- 低数据速率
- 低电量

特别是，蓝牙低功耗技术的创建目的是一次传输非常小的数据包，同时比基本速率/增强数据速率/高速消耗更少的功率
(BR/EDR/HS) 设备。
蓝牙低功耗技术旨在解决两种替代实施方式：

- 智能设备
- 智能就绪设备

智能设备仅支持 BLE 标准。 它用于以低功耗和纽扣电池为关键点的应用（作为传感器）。

智能就绪设备支持 BR/EDR/HS 和 BLE 标准（通常是移动设备或笔记本电脑设备）。

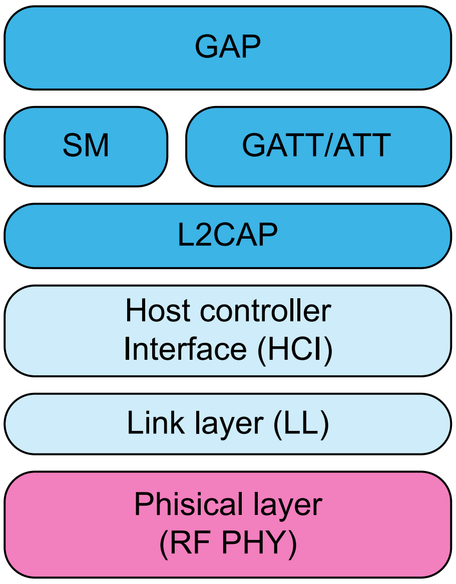
低功耗蓝牙堆栈由两个组件组成：

- 管理员
- 主人

控制器包括物理层和链路层。

Host包括逻辑链路控制和适配协议（L2CAP）、安全管理器（SM）、属性协议（ATT）、通用属性配置文件（GATT）和通用访问配置文件（GAP）。 这两个组件之间的接口称为主机控制器接口 (HCI)。

Figure 2. 蓝牙低功耗堆栈架构



蓝牙规范 v4.1、v4.2、v5.0、v5.1 和 v5.2 已发布，具有新的支持特性：

- **STM32WB v4.1** 支持的当前功能：
  - 同时支持多个角色
  - 支持同步广告和扫描
  - 支持同时成为最多两个主站的从站
  - 隐私 V1.1
  - 低占空比定向广告
  - 连接参数请求程序
  - 32 位 UUID
  - L2CAP 面向连接的通道
- **STM32WB V4.2** 支持的当前功能：
  - **LE** 数据长度扩展
  - 地址解析
  - LE 隐私 1.2
  - LE 安全连接
- **V5.0** 支持的 **STM32WB** 当前功能：
  – LE 2M 物理层

2.2 物理层

物理层是 1 Mbps 自适应跳频高斯频移键控 (GFSK) 无线电或 2Mbit/s 2 级高斯频移键控 (GFSK)。 它在 2400-2483.5 MHz 的免许可 2.4 GHz ISM 频段中运行。 许多其他标准使用此频段：IEEE 802.11、IEEE 802.15。

BLE 系统使用 40 个射频通道 (0-39)，间隔为 2 MHz。 这些射频通道的频率集中在：

$$240 + k * 2MHz,\ where\ k = 0.39 \quad (1)$$

有两种渠道类型：

1. 使用三个固定 RF 频道（37、38 和 39）的广播频道：

   A. 广播通道数据包

   B. 用于可发现性/可连接性的数据包

   C. 用于广播/扫描

2. 数据物理通道使用其他 37 个射频通道用于连接设备之间的双向通信。

Table 1. BLE RF 通道类型和频率

频道索引	射频中心频率	渠道类型
37	2402 MHz	Advertising channel
0	2404 MHz	Data channel
1	2406 MHz	Data channel
….	….	Data channel
10	2424 MHz	Data channel
38	2426 MHz	Advertising channel
11	2428 MHz	Data channel
12	2430 MHz	Data channel
….	….	Data channel
36	2478 MHz	Data channel
39	2480 MHz	Advertising channel

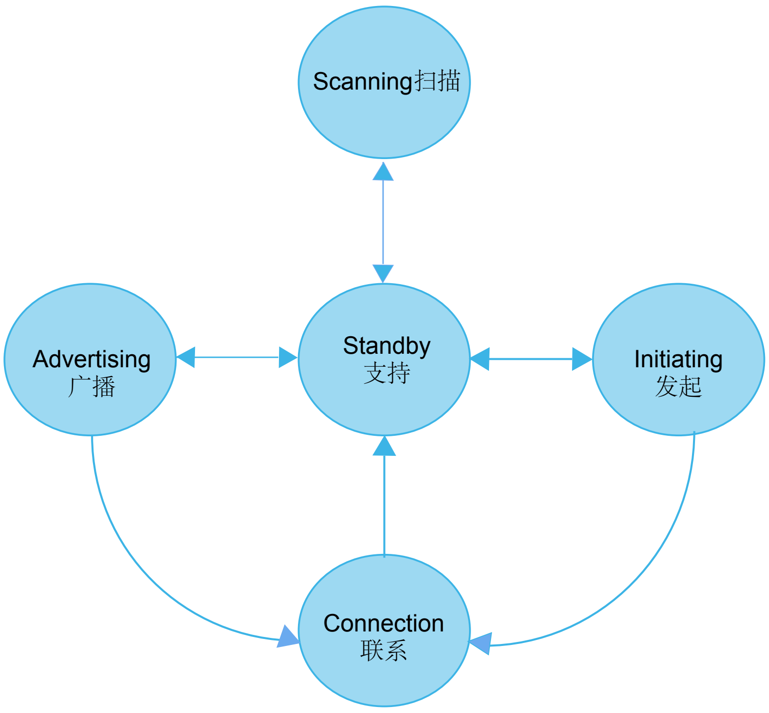
BLE 是一种自适应跳频 (AFH) 技术，它只能使用所有可用频率的一个子集，以避免其他非自适应技术使用的所有频率。 这允许通过使用特定的跳频算法从坏信道移动到已知好的信道，该算法确定要使用的下一个好的信道。

2.3 链路层 (LL)

链路层 (LL) 定义了两个设备如何使用无线电在彼此之间传输信息。

链路层定义了一个具有五种状态的状态机：

Figure 3. 链路层状态机



- 待机：设备不发送或接收数据包
- 广播：设备在广播频道中播放广播（称为广播主设备）
- 扫描：设备寻找广播设备（称为扫描设备）
- 发起：设备发起与广播设备的连接
- 连接：发起方设备处于主机角色：它与处于从机角色的设备进行通信并定义传输时间
- 广播设备处于从属角色：它与处于主角色的单个设备进行通信

2.3.1 BLE数据包

数据包是由一个设备传输并由一个或多个其他设备接收的标记数据。 BLE 数据包结构描述如下。

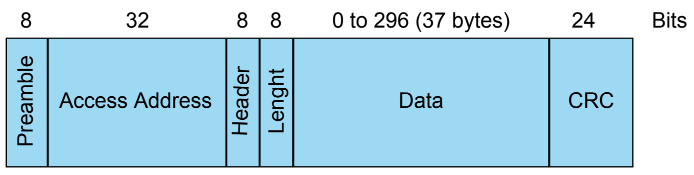
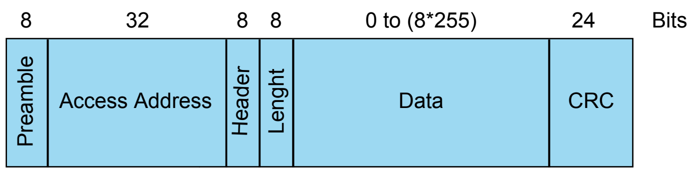


Figure 4. 数据包结构

低功耗蓝牙 BLE 规范 v4.2 定义了 LE 数据包长度扩展功能，该功能将 LE 的链路层 PDU 从 27 字节的数据有效载荷扩展到 251 字节。

Figure 5. 具有LE数据包长度扩展功能的包结构



长度字段的范围为 0 到 255 个字节。 使用加密时，数据包末尾的消息完整性代码 (MIC) 为 4 个字节，因此实际最大可用负载大小为 251 个字节。

- 前导：RF 同步序列
- 访问地址：32 位，广告或数据访问地址（用于标识物理层通道上的通信数据包）
- Header：其内容取决于包类型（广播或数据包）
- 广播包头：

Table 2. 广播数据头内容

广播包类型	Reserved	发送地址类型	接收地址类型
(4 bits)	(2 bits)	(1 bit)	(1 bit)

- 广告包类型定义如下：

Table 3. 广播包类型

Packet type	Description	Notes
ADV_IND	可连接的无向广播	由广播在希望另一台设备连接到它时使用。 设备可以被扫描设备扫描，或者在连接请求接收时作为从设备进入连接。
ADV_DIRECT_IND	可连接的定向广播	由广播在希望特定设备连接到它时使用。 ADV_DIRECT_IND 数据包仅包含广播地址和发起者地址。
ADV_NONCONN_IND	不可连接的无向广播	当广播想要向所有设备提供一些信息，但不希望其他设备向其询问更多信息或连接到它时，由广播使用。 设备只是在相关通道上发送广播数据包，但它不想被任何其他设备连接或扫描。
ADV_SCAN_IND	可扫描的无向广播	供希望允许扫描从中获取更多信息的广播使用。 设备无法连接，但可发现广告数据和扫描响应数据。
SCAN_REQ	扫描请求	由处于扫描状态的设备用于向广播请求附加信息。
SCAN_RSP	扫描响应	由广播设备用于向扫描设备提供附加信息。
CONNECT_REQ	连接请求	由发起设备发送到处于可连接/可发现模式的设备。

广播事件类型决定了允许的响应：

Table 4. 广播事件类型和允许的响应

广播活动类型	允许响应	
	SCAN_REQ	CONNECT_REQ
ADV_IND	YES	YES
ADV_DIRECT_IND	NO	YES
ADV_NONCONN_IND	NO	NO
ADV_SCAN_IND	YES	NO

- 数据包头：

Table 5. 数据包头内容

链路层标识符	下一个序列号	序列号	更多数据	Reserved
(2 bits)	(1 bit)	(1 bit)	(1 bit)	(3 bits)

下一个序列号 (NESN) 位用于执行数据包确认。 它通知接收设备有关发送设备希望它发送的下一个序列号。 重新传输数据包，直到 NESN 与发送数据包中的序列号 (SN) 值不同。

在当前连接事件期间，更多数据位用于向设备发信号通知发送设备有更多数据准备好发送。

有关广告和数据头内容和类型的详细说明，请参阅第 6 节参考文档中的蓝牙规范 [Vol 2]。

- 长度：数据字段上的字节数

Table 6. 数据包长度字段和有效值

	长度字段位
广播包	6 位，有效值从 0 到 37 个字节
数据包	5 位，有效值从 0 到 31 个字节 8 位，有效值从 0 到 255 字节，带 LE 数据包长度扩展

- 数据或有效载荷：是实际传输的数据（广告数据、扫描响应数据、连接建立数据或连接过程中发送的应用数据）
- CRC（24 位）：用于保护数据免受误码。 它是根据标头、长度和数据字段计算的

2.3.2 广播状态

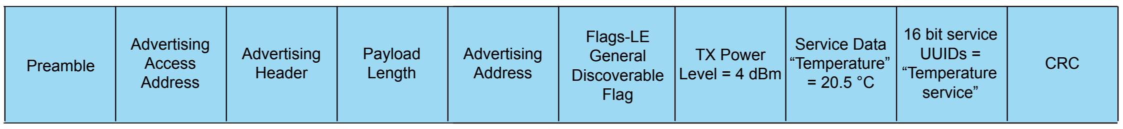
广播状态允许链路层传输广播数据包，并以扫描响应响应来自那些正在主动扫描的设备的扫描请求。

广播客户设备可以通过停止广播进入待机状态。

每次设备进行广播时，它都会在三个广播通道中的每一个上发送相同的数据包。 这三个数据包序列称为“广播事件”。 两个广播事件之间的时间称为广播间隔，可以从 20 毫秒到每 10.28 秒。

广播数据包示例列出了设备实现的服务 UUID（通用可发现标志、tx power = 4dbm、服务数据 = 温度服务和 16 位服务 UUID）。

Figure 6. 带有广播类型标志的广播包


Preamble
Advertising Access Address
Advertising Header
Payload Length
Advertising Address
Flags-LE General Discoverable Flag
TX Power Level = 4 dBm
Service Data “Temperature” = 20.5 °C
16 bit service UUIDs = “Temperature service”
CRC

标志 AD 类型字节包含以下标志位：
- 有限的可发现模式（位 0）
- 一般可发现模式（位 1）
- 不支持 BR/EDR（位 2，在 BLE 上为 1）
- 同时将 LE 和 BR/EDR 连接到同一设备（控制器）（位 3）
- 同时将 LE 和 BR/EDR 连接到同一设备（主机）（位 4）

如果任何位非零，则标志 AD 类型包含在广播数据中（不包含在扫描响应中）。
在启用广播之前，可以设置以下广播参数：

- 广播间隔
- 广播地址类型
- 广播设备地址
- 广播渠道图：应该使用三个广播渠道中的哪一个

- 广播过滤政策：
  - 处理来自白名单中设备的扫描/连接请求
  - 处理所有扫描/连接请求（默认广播过滤策略）
  - 处理所有设备的连接请求，但只扫描白名单中的请求
  - 处理来自所有设备的扫描请求，但只处理白名单中的连接请求

白名单是设备控制器用来过滤设备的存储设备地址列表。 白名单内容在使用过程中不可修改。 如果设备处于广播状态并使用白名单过滤设备（扫描请求或连接请求），则必须禁用广播模式才能更改其白名单。

2.3.3 扫描状态

有两种类型的扫描：

- 被动扫描：它允许从广播设备接收广告数据
- 主动扫描：当收到广播时，设备可以发回扫描请求包，以便从广播那里得到扫描响应。 这允许扫描仪设备从广播设备获得附加信息。

可以设置以下扫描参数：

- 扫描类型（被动或主动）
- 扫描间隔：控制器应该多久扫描一次
- 扫描窗口：对于每个扫描间隔，它定义了设备扫描了多长时间
- 扫描过滤策略：它可以接受所有的广播包（默认策略）或只接受白名单上的那些。

一旦设置了扫描参数，就可以启用设备扫描。 扫描仪设备的控制器在广告报告事件内向上层发送任何接收到的广告数据包。 此事件包括此广告数据包的广告商地址、广告商数据和接收到的信号强度指示 (RSSI)。 RSSI 可与包含在广告数据包中的发射功率电平信息一起使用，以确定信号的路径损耗并确定设备的距离：

路径损耗 = Tx 功率 – RSSI

2.3.4 连接状态

当要传输的数据比广告数据所允许的更复杂或需要两个设备之间的双向可靠通信时，就会建立连接。

当发起者设备从它想要连接的广告设备接收到广告数据包时，它可以向广告者设备发送连接请求数据包。 此数据包包括建立和处理两个设备之间的连接所需的所有必需信息：

- 连接中使用的访问地址，以识别物理链路上的通信
- CRC 初始化值
- 传输窗口大小（第一个数据包的定时窗口）
- 发送窗口偏移（发送窗口开始）
- 连接间隔（两个连接事件之间的时间）
- 从属延迟（从属在强制侦听之前可以忽略连接事件的次数）
- 监督超时（在认为链路丢失之前两个正确接收的数据包之间的最大时间）
- 通道映射：37 位（1 = 好； 0 = 坏）
- 跳频值（5 到 16 之间的随机数）
- 睡眠时钟精度范围（用于确定连接事件时从设备的不确定性窗口）

有关连接请求数据包的详细说明，请参阅蓝牙规范 [Vol 6]。 表 7 总结了允许的时间范围。 连接请求
时间间隔：

Table 7. 连接请求时间间隔

范围	Min.	Max.	注释
发送窗口大小	1.25 毫秒	10 毫秒	-
发送窗口偏移	0	连接间隔	1.25 毫秒的倍数
连接间隔	7.5 毫秒	4 秒	1.25 毫秒的倍数
监督超时	100 毫秒	32 秒	10 毫秒的倍数

发送窗口在连接请求数据包结束加上发送窗口偏移量加上 1.25 ms 的强制延迟后开始。当传输窗口开始时，从设备进入接收器模式并等待来自主设备的数据包。如果在这段时间内没有收到任何数据包，从设备将退出接收器模式，并在稍后再次尝试连接间隔。建立连接后，主机必须在每次连接事件时向从机发送数据包，以允许从机向主机发送数据包。可选地，从设备可以跳过给定数量的连接事件（从设备延迟）。

连接事件是从上一个连接事件开始到下一个连接事件开始之间的时间。

一个 BLE slave 设备只能连接一个 BLE master 设备，但是一个 BLE master 设备可以连接多个 BLE slave 设备。在蓝牙 SIG 上，主设备可以连接的从设备数量没有限制（这受到特定使用的 BLE 技术或堆栈的限制）。

2.4 主机控制器接口 (HCI)

主机控制器接口 (HCI) 层通过软件 API 或硬件接口（例如：SPI、UART 或 USB）提供主机和控制器之间的通信方式。 它来自标准的蓝牙规范，带有用于低功耗特定功能的新附加命令。

2.5 逻辑链路控制和适配层协议 (L2CAP)

逻辑链路控制和适配层协议 (L2CAP) 支持更高级别的协议复用、数据包分段和重组操作以及服务质量信息的传送。

2.6 属性协议 (ATT)

属性协议 (ATT) 允许设备将某些数据（称为属性）公开给另一个设备。 暴露属性的设备称为服务器，使用它们的对等设备称为客户端。
属性是具有以下组件的数据：

- 属性句柄：它是一个 16 位值，用于标识服务器上的属性，允许客户端在读取或写入请求中引用该属性
- 属性类型：由通用唯一标识符 (UUID) 定义，它决定了值的含义。标准 16 位属性 UUID 由蓝牙 SIG 定义
- 属性值：长度为 (0 ~ 512) 个八位字节
- 属性权限：它们由使用该属性的每个上层定义。它们指定读取和/或写入访问以及通知和/或指示所需的安全级别。使用属性协议无法发现权限。有不同的权限类型：
  – 访问权限：它们确定可以对属性执行哪些类型的请求（可读可写，可读可写）
  – 身份验证权限：它们确定属性是否需要身份验证。如果出现身份验证错误，客户端可以尝试使用安全管理器对其进行身份验证并发回请求
  – 授权权限（无授权、授权）：这是服务器的一个属性，它可以授权客户端访问或不访问一组属性（客户端无法解决授权错误）

Table 8. 属性示例

属性句柄	属性类型	属性值	属性权限
0x0008	“Temperature UUID”	“Temperature Value”	“Read only, no authorization, no authentication”

- “Temperature UUID” 由“Temperature characteristic” 规范定义，它是一个有符号的 16 位整数。

属性集合称为数据库，它始终包含在属性服务器中。
属性协议定义了一组方法协议来发现、读取和写入对等设备上的属性。 它实现了属性服务器和属性客户端之间的点对点客户端-服务器协议，如下所示：

- 服务器角色
– 包含所有属性（属性数据库）
– 接收请求、执行、响应命令
– 表示，当数据改变时通知一个属性值
- 客户角色
– 与服务器对话
– 发送请求，等待响应（它可以访问（读取），更新（写入）数据）
– 确认指示

服务器暴露的属性可以被客户端发现、读取和写入，它们可以被服务器指示和通知，如表9中所述。属性协议消息。

Table 9. 属性协议信息

协议数据单元（PDU信息）。	发送人	描述
请求	Client	客户端询问服务器（它总是引起响应）
响应	Server	服务器发送响应来自客户端的请求
命令	Client	客户端向服务器发出命令（无响应）
通知	Server	服务器通知客户端新值（无确认）
指示	Server	服务器向客户端指示新值（它总是导致确认）
确认	Client	对表示的确认

2.7 安全经理 (SM)

蓝牙低能量链路层通过使用计数器模式和CBC-MAC（密码块链-信息验证码）算法以及128位AES块密码来支持加密和验证。CBC-MAC（密码块链-信息验证码）算法和128位AES块密码，支持加密和验证。(AES-CCM)。当在一个连接中使用加密和认证时，一个4字节的消息完整性检查 (MIC)被附加到数据通道PDU的有效载荷上。
加密被应用于PDU有效载荷和MIC字段。
当两个设备想在连接期间加密通信时，安全管理器使用配对程序。该程序允许两个设备通过交换它们的身份信息进行认证，以创建安全密钥，作为信任关系或（单一）安全连接的基础。有一些方法可以用来执行配对程序。其中一些方法提供保护，以防止

- 中间人（MITM）攻击：一个设备能够监测并修改或添加新的信息到两个设备之间的通信渠道。一个典型的情况是，一个设备能够连接到每个设备，并通过与每个设备的通信来充当其他设备的角色。
- 被动窃听攻击：通过嗅探设备监听其他设备的通信。

蓝牙低能量规格v4.0或v4.1的配对，也称为LE传统配对，根据设备的IO能力，支持以下方法。刚好工作，密码输入和带外（OOB）。
在蓝牙低能量规范v4.2中，定义了LE安全连接配对模型。新的安全模型的主要特点是。

1. 密钥交换过程使用椭圆曲线 Diffie-Hellman (ECDH) 算法：这允许通过不安全的通道交换密钥并防止被动窃听攻击（通过嗅探设备秘密监听其他设备的通信）

2. LE 传统配对已有的 3 种方法中添加了一种称为“数值比较”的新方法

根据设备 IO 能力选择配对程序。 共有三种输入功能。
共有三种输入功能：

- 无输入
- 能够选择是/否
- 能够使用键盘输入数字

有两种输出能力：

- 无输出
- 数字输出：能够显示六位数

下表显示了可能的 IO 功能组合

Table 10. BLE 设备上输入/输出功能的组合

	No output	Display
No input	No input, no output	Display only
Yes/No	No input, no output	Display yes/no
Keyboard	Keyboard only	Keyboard display

LE 传统配对

LE 传统配对算法使用并生成 2 个密钥：

- 临时密钥 (TK)：用于生成短期密钥 (STK) 的 128 位临时密钥
- 短期密钥 (STK)：用于在配对后加密连接的 128 位临时密钥

配对过程是一个三阶段的过程。

第一阶段：配对特征交换

两个连接的设备通过使用配对请求消息来传达它们的输入/输出能力。 此消息还包含说明带外数据是否可用以及身份验证要求的位。 第 1 阶段交换的信息用于选择在第 2 阶段生成 STK 使用哪种配对方法。

阶段 2：短期密钥 (STK) 生成

配对设备首先使用以下密钥生成方法之一定义临时密钥 (TK)

1. 带外（OOB）方法，使用带外通信（例如NFC）进行TK协议。 它提供身份验证（MITM 保护）。 仅当两个设备上都设置了带外位时才选择此方法，否则必须使用设备的 IO 功能来确定可以使用哪种其他方法（Passkey Entry 或 Just Works）

2. Passkey 输入方法：用户在设备之间传递六位数字作为 TK。 它提供身份验证（MITM 保护）

3. 有效：此方法不提供身份验证和针对中间人 (MITM) 攻击的保护

Passkey 和 Just Works 方法之间的选择是根据下表定义的 IO 能力完成的。

Table 11. 用于计算临时密钥 (TK) 的方法

	Display only	Display yes/no	Keyboard only	No input, no output	Keyboard display
Display Only	Just Works	Just Works	Passkey Entry	Just Works	Passkey Entry
Display Yes/No	Just Works	Just Works	Passkey Entry	Just Works	Passkey Entry
Keyboard Only	Passkey Entry	Passkey Entry	Passkey Entry	Just Works	Passkey Entry
No Input No Output	Just Works	Just Works	Just Works	Just Works	Just Works
Keyboard Display	Passkey Entry	Passkey Entry	Passkey Entry	Just Works	Passkey Entry

阶段 3：用于计算临时密钥 (TK) 的传输特定密钥分发方法

阶段 2 完成后，最多可以通过使用 STK 密钥加密的消息分发三个 128 位密钥：

1. 1.长期密钥（LTK）：用于生成用于链路层加密和认证的128位密钥
2. 2.连接签名解析密钥（CSRK）：一个128位的密钥，用于在ATT层进行数据签名和验证
3. 3. 身份解析密钥（IRK）：用于生成和解析随机地址的 128 位密钥

LE 安全连接

LE 安全连接配对方法使用并生成一个密钥：

- 长期密钥 (LTK)：128 位密钥，用于在配对和后续连接后加密连接

配对过程分为三个阶段：

第一阶段：配对特征交换

两个连接的设备通过使用配对请求消息来传达它们的输入/输出能力。此消息还包含一点说明是否带外数据可用以及身份验证要求。阶段 1 中交换的信息用于选择在阶段 2 中使用哪种配对方法。

阶段 2：长期密钥 (LTK) 生成

配对过程由发起设备启动，发起设备将其公钥发送给接收设备。接收设备用它的公钥回复。所有配对方法都完成了公钥交换阶段

（OOB 除外）。每个设备都会生成自己的椭圆曲线 Diffie-Hellman (ECDH) 公私密钥对。每个密钥对包含一个私有（秘密）密钥和一个公共密钥。密钥对应该在每个设备上只生成一次，并且可以在执行配对之前计算。

支持以下配对密钥生成方法：

1. 带外（OOB）方法，使用带外通信来设置公钥。如果配对请求/响应中的带外位至少由一个设备设置，则选择此方法，否则必须使用设备的 IO 能力来确定可以使用哪种其他方法（Passkey entry，Just Works或数字比较）

2. Just Works：此方法未经身份验证，不提供任何针对中间人 (MITM) 攻击的保护

3.密码输入法：此方法是经过认证的。用户传递六个数字。这个六位数的值是设备认证的基础

4.数值比较：此方法是经过认证的。两种设备的 IO 功能都设置为显示是/否或键盘显示。两台设备计算一个六位数的确认值，在两台设备上显示给用户：要求用户通过输入是或否来确认是否存在匹配。如果在两个设备上都选择是，则配对成功。这种方法允许用户确认他的设备与正确的设备连接，在有多个设备的上下文中，这些设备不能有不同的名称

可能的方法中的选择基于下表。

Table 12. 将 IO 功能映射到可能的密钥生成方法

发起者/响应者	仅显示	显示是/否	仅键盘	无输入无输出	键盘显示
仅显示	Just Works	Just Works	Passkey Entry	Just Works	Passkey Entry
显示是/否	Just Works	Just Works (LE legacy) Numeric comparison (LE secure connections)	Passkey Entry	Just Works	Passkey Entry (LE legacy) Numeric comparison (LE secure connections)
仅键盘	Passkey Entry	Passkey Entry	Passkey Entry	Just Works	Passkey Entry
无输入无输出	Just Works	Just Works	Just Works	Just Works	Just Works
键盘显示	Passkey Entry	Passkey Entry (LE legacy) Numeric comparison (LE secure connections)	Passkey Entry	Just Works	Passkey Entry (LE legacy) Numeric comparison (LE secure connections)

Note: 如果可能的密钥生成方法不提供与安全属性匹配的密钥（已认证 - MITM 保护或未认证 - 无 MITM 保护），则设备发送配对失败命令，错误代码为“认证要求”。
阶段 3：传输特定的密钥分发
主从之间交换以下密钥：

- 用于验证未加密数据的连接签名解析密钥 (CSRK)
- 用于设备身份和隐私的身份解析密钥 (IRK)

当存储已建立的加密密钥以用于将来的身份验证时，设备将被绑定。

数据签名

还可以使用 CSRK 密钥通过未加密的链路层连接传输经过身份验证的数据：在 ATT 层的数据有效负载之后放置一个 12 字节的签名。 签名算法还使用了一个计数器来防止重放攻击（一个外部设备，它可以简单地捕获一些数据包，然后按原样发送，而无需了解数据包内容：接收器设备只需检查数据包计数器并丢弃它，因为它 帧计数器小于最近收到的好数据包）。

2.8 隐私

始终使用相同地址（公共或静态随机）进行广告的设备可以被扫描仪跟踪。 这可以通过启用广告设备上的隐私功能来避免。 在启用隐私的设备上，使用私有地址。 私有地址有两种：

- 不可解析的私有地址
- 可解析的私有地址

不可解析的私有地址是完全随机的（除了两个最高有效位）并且无法解析。因此，使用不可解析私有地址的设备无法被先前未配对的那些设备识别。可解析的私有地址有一个 24 位的随机部分和一个散列部分。散列来自随机数和 IRK（身份解析密钥）。因此，只有知道这个 IRK 的设备才能解析地址并识别设备。 IRK 在配对过程中分配。

这两种类型的地址都经常更改，从而增强了设备身份的机密性。在 GAP 发现模式和程序期间不使用隐私功能，而仅在 GAP 连接模式和程序期间使用。

在 v4.1 之前的低功耗蓝牙堆栈上，私有地址由主机解析和生成。

在蓝牙 v4.2 中，隐私功能已从 1.1 版更新到 1.2 版。在低功耗蓝牙协议栈 v4.2 上，控制器可以使用主机提供的设备身份信息解析和生成私有地址。

外设

不可连接模式下启用隐私的外围设备使用不可解析或可解析的私有地址。要连接到中央，只有在使用主机隐私时才应使用无向可连接模式。 如果使用控制器隐私，设备也可以使用定向连接模式。 在可连接模式下，设备使用可解析的私有地址。

无论使用不可解析的还是可解析的私有地址，它们都会在每间隔 15 分钟后自动重新生成。 设备不会将设备名称发送到广播数据。

中央

启用隐私的中心执行主动扫描，仅使用不可解析或可解析的私人地址。 要连接到外围设备，如果主机，则应使用一般的连接建立过程

隐私已启用。 通过基于控制器的隐私，可以使用任何连接过程。 中心使用一个可解析的私有地址作为发起者的设备地址。 每隔 15 分钟就会重新生成一个新的可解析或不可解析的私有地址。

广播装置

启用隐私的广播装置使用不可解析或可解析的私有地址。 每间隔 15 分钟后自动生成新地址。 广播装置不应向广播数据发送名称或唯一数据。

观察员

启用隐私的观察者使用不可解析或可解析的私有地址。 每间隔 15 分钟后自动生成新地址。

2.8.1 设备过滤

蓝牙 LE 提供了一种减少设备响应数量以降低功耗的方法，因为这意味着控制器和上层之间的传输和交互更少。过滤是通过白名单实现的。当启用白名单时，那些不在此列表中的设备将被链路层忽略。

在蓝牙 4.2 之前，不能使用设备过滤，而远程设备使用隐私。由于引入了链路层隐私，可以在检查是否在白名单中之前解析远程设备身份地址。

通过将“Filter_Duplicates”模式设置为 1，用户可以激活 LL 级别的广告过滤。它的工作原理如下所述。

LL 维护两组，每组四个缓冲区：一组用于四个不利指示地址，另一组用于四个扫描响应地址。

当接收到一个广告指示包时，它的地址（6 个字节）与四个存储的地址进行比较。如果它与四个地址之一匹配，则丢弃该数据包。如果不匹配，则向上层报告该指示并将其地址存储在缓冲区中，同时从缓冲区中删除最旧的地址。

相同的过程分别适用于扫描响应。

2.9 通用属性配置文件 (GATT)

通用属性配置文件（GATT）定义了使用 ATT 协议的框架，用于服务、特征、描述符发现、特征读取、写入、指示和通知。
在 GATT 上下文中，当两个设备连接时，有两个设备角色：

- GATT 客户端：设备通过读取、写入、通知或指示操作访问远程 GATT 服务器上的数据
- GATT 服务器：设备在本地存储数据并向远程 GATT 客户端提供数据访问方法

设备可以同时作为 GATT 服务器和 GATT 客户端。
设备的 GATT 角色与主从角色在逻辑上是分开的。 主、从角色定义了 BLE 无线电连接的管理方式，GATT 客户端/服务器角色由数据存储和数据流决定。
因此，从（外围）设备必须是 GATT 服务器，而主（中央）设备不必是 GATT 客户端。
由 ATT 传输的属性封装在以下基本类型中：

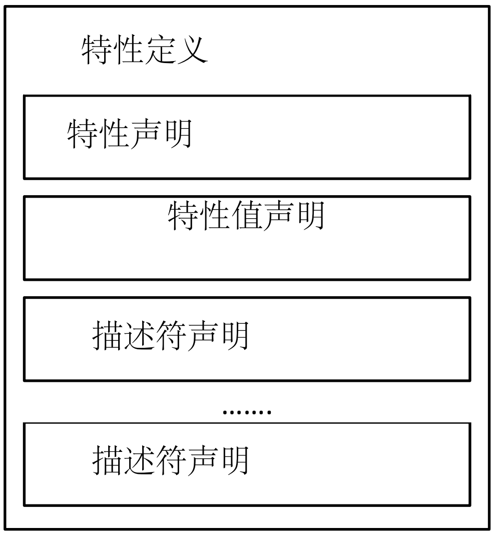
1. 特征（有相关描述符）
2. 服务（主要、次要和包括）

2.9.1 特征属性类型

特性是一种属性类型，它包含单个值和任意数量的描述特性值的描述符，这些描述符可以使用户能够理解它。
一个特性公开了该值所代表的数据类型，该值是否可以被读取或写入，如何配置该值以指示或通知，并说明该值的含义。
特征具有以下组成部分：

1.特性声明
2. 特性值
3. 特征描述符

Figure 7. 特征定义示例



特性声明是一个定义如下的属性：

Table 13. 特性声明

属性句柄	属性类型	属性值	属性权限
0xNNNN	0x2803 (特征属性类型的 UUID)	特征值属性（读取、广播、写入、无响应写入、通知、指示等）。 确定如何使用特征值或如何访问特征描述符	只读， 无认证，无授权
		特征值属性句柄	
		特征值 UUID（16 或 128 位）	

特性声明包含特性的值。 该值是特征声明后的第一个属性：

Table 14. 特性值

特性值	属性类型	属性值	属性权限
0xNNNN	0xuuuu – 16 位或 128 位用于特征 UUID	特性值	更高层配置文件或实现特定

2.9.2 特征描述符类型

特征描述符用于描述特征值，为特征添加特定的“含义”，使其易于用户理解。 以下特征描述符可用：

1. 特征扩展属性：允许在特征中添加扩展属性
2. 特征用户描述：它使设备能够将文本字符串与特征相关联
3. 客户端特性配置：如果特性可以被通知或指示，则为必填项。客户端应用程序必须编写此特征描述符以启用特征通知或指示（前提是特征属性允许通知或指示）
4.服务器特性配置：可选描述符
5、特性表示格式：允许通过格式、指数、单位名称空间、描述等字段定义特性值表示格式，以便正确显示相关值（例如oC格式的温度测量值）
6、特征聚合格式：允许聚合多种特征呈现格式。有关特征描述符的详细说明，请参阅蓝牙规范。

2.9.3 服务属性类型

服务是一组特性，它们一起运行以向应用配置文件提供全局服务。 例如，健康温度计服务包括温度测量值和测量之间的时间间隔的特性。 服务或主要服务可以引用称为次要服务的其他服务。
服务定义如下：

Table 15. 服务声明

属性句柄	属性类型	属性值	属性权限
0xNNNN	0x2800 – “主要服务”的 UUID 或 0x2801 – “次要服务”的 UUID	0xuuuu – 16 位或 128 位服务 UUID	只读， 无认证，无授权

服务包含服务声明并且可能包含定义和特征定义。 服务包括在服务声明和服务器的任何其他属性之后的声明。

Table 16. 包括声明

Attribute handle	Attribute type	Attribute value			Attribute permissions
0xNNNN	0x2802 (UUID for include attribute type)	Include service attribute handle	End group handle	Service UUID	Read only, no authentication, no authorization

“包含服务属性句柄”是包含的二级服务的属性句柄，“端组句柄”是包含的二级服务中最后一个属性的句柄。

2.9.4 GATT 程序

通用属性配置文件 (GATT) 定义了一组标准程序，允许发现服务、特征、相关描述符以及如何使用它们。
可以使用以下程序：

- 发现程序（表 17. 发现程序和相关响应事件）
- 客户端启动的程序（表 18. 客户端启动的程序和相关响应事件）
- 服务器启动的过程（表 19. 服务器启动的过程和相关响应事件）

Table 17. 发现程序和相关响应事件

程序	响应事件
发现所有主要服务	按组响应阅读
按服务 UUID 发现主要服务	按类型查找值响应
查找包含的服务	按类型读取响应事件
发现服务的所有特征	按类型响应读取
通过 UUID 发现特征	按类型响应读取
发现所有特征描述符	查找信息响应

Table 18. 客户发起的程序和相关的响应事件

Procedure	Response events
读取特征值	读取响应事件
通过 UUID 读取特征值	读取响应事件
读取长特征值	读取 blob 响应事件
读取多个特征值	读取响应事件
写入特征值无响应	没有事件生成
签名写无响应	没有事件生成
写入特征值	写响应事件。
写长特征值	准备写响应 执行写响应
可靠写入	准备写响应 执行写响应

Table 19. 服务器启动的过程和相关的响应事件

Procedure	Response events
通知	没有事件生成
适应症	确认事件

有关 GATT 程序和相关响应事件的详细说明，请参阅第 6 节参考文档中的蓝牙规范。

2.10 通用访问配置文件(GAP)

蓝牙系统定义了一个由所有蓝牙设备实现的基本配置文件，称为通用访问配置文件 (GAP)。 此通用配置文件定义了蓝牙设备的基本要求。
下表描述了四种 GAP 配置文件角色：

Table 20. GAP 角色

角色[1]	描述	发射机	接收者	典型例子
广播装置	发送广播事件	M	O	发送温度值的温度传感器
观察员	接收广播事件	O	M	温度显示器，只接收和显示温度值
外围设备	永远是从机。 它处于可连接的广播模式。 支持所有LL控制程序； 加密是可选的	M	M	手表
中央	永远是主机。 它从不做广播。 它支持主动或被动扫描。 支持所有LL控制程序； 加密是可选的	M	M	手机

1. 1. M = 强制； O = 可选

在 GAP 上下文中，定义了两个基本概念：

- GAP 模式：将设备配置为长时间以特定方式运行。 有四种 GAP 模式类型：广播、可发现、可连接和可绑定类型
- GAP 程序：它将设备配置为在特定的有限时间内执行单个操作。 有四种 GAP 程序类型：观察者、发现、连接、绑定程序

可以同时使用不同类型的可发现和可连接模式。 定义了以下 GAP 模式：

Table 21. GAP广播模式

Mode	Description	Notes	GAP role
广播模式	设备仅使用链路层广告通道和数据包广播数据（它不会在 Flags AD 类型上设置任何位）	使用观察程序的设备可以检测广播数据	广播装置

Table 22. GAP 可发现模式

Mode	Description	Notes	GAP role
Non-discoverable mode	It cannot set the limited and general discoverable bits on flags AD type	It cannot be discovered by a device performing a general or limited discovery procedure	Peripheral
Limited discoverable mode	It sets the limited discoverable bit on flags AD type	It is allowed for about 30 s. It is used by devices with which user has recently interacted. For example, when a user presses a button on the device	Peripheral
General discoverable mode	It sets the general discoverable bit on flags AD type	It is used when a device wants to be discoverable. There is no limit on the discoverability time	Peripheral

Table 23. **GAP connectable modes**

Mode	Description	Notes	GAP role
Non-connectable mode	It can only use ADV_NONCONN_IND or ADV_SCAN_IND advertising packets	It cannot use a connectable advertising packet when it advertises	Peripheral
Direct connectable mode	It uses ADV_DIRECT advertising packet	It is used from a peripheral device that wants to connect quickly to a central device. It can be used only for 1.28 seconds, and it requires both peripheral and central devices addresses	Peripheral
Undirected connectable mode	It uses the ADV_IND advertising packet	It is used from a device that wants to be connectable. Since ADV_IND advertising packet can include the flag AD type, a device can be in discoverable and undirected connectable mode at the same time. Connectable mode is terminated when the device moves to connection mode or when it moves to non-connectable mode	Peripheral

Table 24. **GAP bondable modes**

Mode	Description	Notes	GAP role
Non-bondable mode	It does not allow a bond to be created with a peer device	No keys are stored from the device	Peripheral
Bondable mode	Device accepts bonding request from a Central device.		Peripheral

The following GAP procedures are defined in Table 25. GAP observer procedure:

Table 25. **GAP observer procedure**

Procedure	Description	Notes	Role
Observation procedure	It allows a device to look for broadcaster devices data	-	Observer

Table 26. **GAP discovery procedures**

Procedure	Description	Notes	Role
Limited discoverable procedure	It is used for discovery peripheral devices in limited discovery mode	Device filtering is applied based on flag AD type information	Central
General discoverable procedure	It is used for discovery peripheral devices in general ad limited discovery mode	Device filtering is applied based on flag AD type information	Central
Name discovery procedure	It is the procedure to retrieve the “Bluetooth Device Name” from connectable devices		Central

Table 27. **GAP connection procedures**

Procedure	Description	Notes	Role
Auto connection establishment procedure	Allows the connection with one or more devices in the directed connectable mode or the undirected connectable mode	It uses white lists	Central
General connection establishment procedure	Allows a connection with a set of known peer devices in the directed connectable mode or the undirected connectable mode	It supports private addresses by using the direct connection establishment procedure when it detects a device with a private address during the passive scan	Central
Selective connection establishment procedure	Establish a connection with the host selected connection configuration parameters with a set of devices in the white list	It uses white lists and it scans by this white list	Central
Direct connection establishment procedure	Establish a connection with a specific device using a set of connection interval parameters	General and selective procedures use it	Central
Connection parameter update procedure	Updates the connection parameters used during the connection		Central
Terminate procedure	Terminates a GAP procedure		Central

Table 28. **GAP bonding procedures**

Procedure	Description	Notes	Role
Bonding procedure	Starts the pairing process with the bonding bit set on the pairing request		Central

For a detailed description of the GAP procedures, refer to the Bluetooth specifications.

2.11 BLE profiles and applications

A service collects a set of characteristics and exposes the behaviour of these characteristics (what the device does, but not how a device uses them). A service does not define characteristic use cases. Use cases determine which services are required (how to use services on a device). This is done through a profile which defines which services are required for a specific use case:

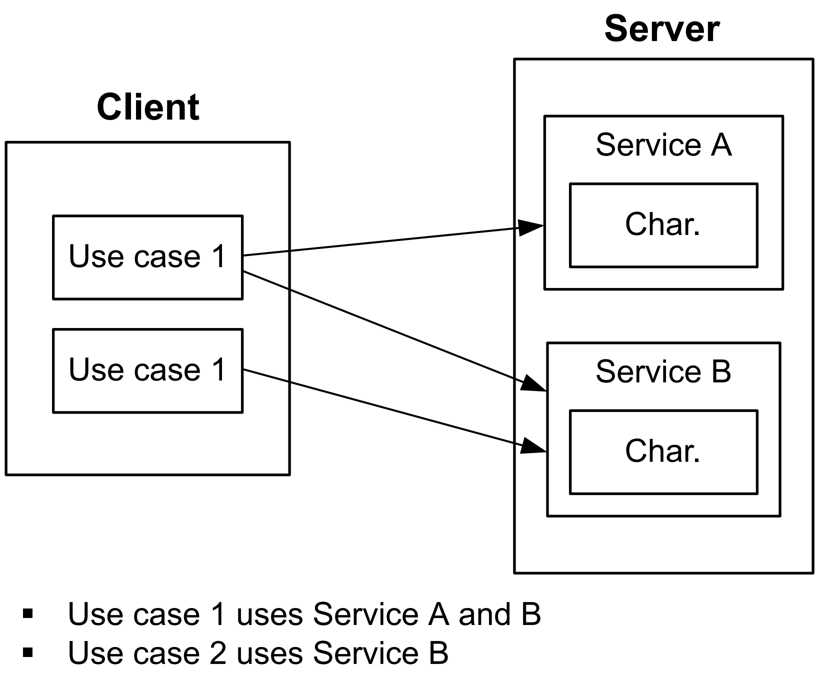
- Profile clients implement use cases
- Profile servers implement services

Standard profiles or proprietary profiles can be used. When using a non-standard profile, a 128-bit UUID is required and must be generated randomly.

Currently, any standard Bluetooth SIG profile (services, and characteristics) uses 16-bit UUIDs. Services, characteristics specification and UUID assignation can be downloaded from the following SIG web pages:

- https://developer.bluetooth.org/gatt/services/Pages/ServicesHome.aspx
- https://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicsHome.aspx

Figure 8. **Client and server profiles**


Server
Client
Service A
Char.
Use case 1
Use case 1
Service B
Char.
▪ Use case 1 uses Service A and B
▪ Use case 2 uses Service B

2.11.1 Proximity profile example

This section simply describes the proximity profile target, how it works and required services:

Target

- When a device is close, very far, far away:
  - Causes an alert

How it works

- If a device disconnects, it causes an alert
- Alert on link loss: «Link Loss» service
  - If a device is too far away
  - Causes an alert on path loss: «Immediate Alert» and «Tx Power» service
- «Link Loss» service
  - «Alert Level» characteristic
  - Behavior: on link loss, causes alert as enumerated
- «Immediate Alert» service
  - «Alert Level» characteristic
  - Behavior: when written, causes alert as enumerated
- «Tx Power» service
  - «Tx Power» characteristic
  - Behavior: when read, reports current Tx Power for connection

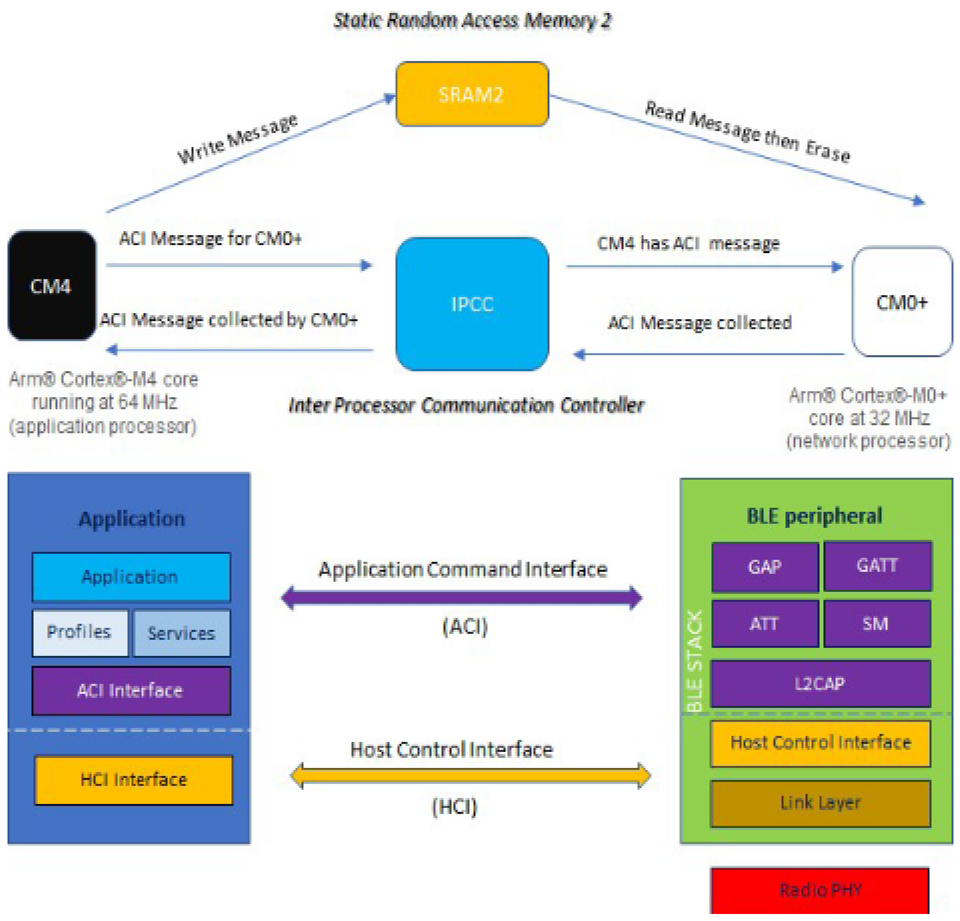
3 STM32WB Bluetooth low energy stack

STM32WB devices are network co-processors which provide high-level interface to control its Bluetooth low energy functionalities. This interface is called ACI (application command interface). STM32WB devices embed on Arm Cortex-M0, respectively and securely, the Bluetooth Smart protocol stack. As a consequence, no BLE library is required on the external micro-controller Arm Cortex-M4. The Inter Process Communication Controller (IPCC) interface communication protocol allows Cortex-M4 micro-controller to send and receive ACI commands to microcontroller Cortex-M0 co-processor. Current secure Bluetooth low energy (BLE) stack is based on standard C library, in binary format. Before sending any BLE command, the Cortex-M4 shall first send the system command SHCI_C2_BLE_Init() to the Cortex-M0 to start the BLE stack. Please, refer to AN5289 for more description of the system command and BLE startup flow.

The BLE binary library provides the following functionalities:

- Stack APIs for BLE stack initialization, BLE stack application command interface (HCI command prefixed with hci_, and vendor specific command prefixed with aci_), Sleep timer access and BLE stack state machines handling
- Stack event callbacks inform user application about BLE stack events and sleep timer events
- Provides interrupt handler for radio IP

Figure 9. **STM32WB stacks architecture and interface between secure Arm Cortex-M0 and Arm Cortex-M4**



3.1 BLE stack library framework

The BLE stack library framework allows commands to be sent to the STM32WB SoC BLE stack and it also provides definitions of BLE event callbacks. The BLE stack APIs utilize and extend the standard HCI data format defined within the Bluetooth specifications.

The provided set of APIs supports the following commands:

- Standard HCI commands for controller as defined by Bluetooth specifications
- Vendor Specific (VS) HCI commands for controller
- Vendor Specific (VS) ACI commands for host (L2CAP, ATT, SM, GATT, GAP)

The reference ACI interface framework is provided within STM32WB kits software package(refer to Section 6 Reference documents). The ACI interface framework contains the code that is used to send ACI commands between both STM32WB network processors: Arm® Cortex®-M0 (network processor) and Arm® Cortex®-M4 core running at 64 MHz (application processor). It also provides definitions of device events. The ACI framework interface is defined by the following header files:

Table 29. **BLE application stack library framework interface**

File	Description
ble_hci_le.h	HCI library functions prototypes and error code definition.
ble_events.h	Header file that contains commands and events for STM32WB FW stack
ble_gatt_aci.h	Header file for GATT server definition
ble_l2cap_aci.h	Header file with L2CAP commands for STM32WB FW stack
ble_gap_aci.h	Header file for STM32WB GAP layer
ble_hal_aci.h	Header file with HCI commands for STM32WB FW stack
ble_types.h	Header file with ACI definitions for STM32WB FW stack

4 Design an application using the STM32WB BLE stack

This section provides information and code examples about how to design and implement a Bluetooth low energy application on a STM32WB device using the BLE stack v2.x binary library.

User implementing a BLE application on a STM32WB device has to go through some basic and common steps:

1. Initialization phase and main application loop
2. STM32WB events and events Callback setup
3. Services and characteristic configuration (on GATT server)
4. Create a connection: discoverable, connectable modes and procedures
5. Security (pairing and bonding)
6. Service and characteristic discovery
7. Characteristic notification/indications, write, read
8. Basic/typical error conditions description

Note: In the following sections, some user applications “defines” are used to simply identify the device Bluetooth low energy role (central, peripheral, client and server).

Table 30. User application defines for BLE device roles

Define	Description
GATT_CLIENT	GATT client role
GATT_SERVER	GATT server role

4.1 Initialization phase and main application loop

The following main steps are required for properly configure the STM32WB devices.

1. Initialize the HAL library:
   a. Configure the Flash prefetch, instruction and Data caches.
   b. Configures the SysTick to generate an interrupt each 1 millisecond, which is clocked by the MSI (at this stage, the clock is not yet configured and thus the system is running from the internal MSI at 4 MHz).
   c. Set NVIC Group Priority to 4.
   d. Calls the HAL_MspInit() callback function defined in user file"stm32wbxx_hal_msp.c" to do the global low level hardware initialization
2. Configure the system clock
3. Configure the peripheral clocks
4. Configure the system power mode
5. Initialize all configured peripherals
6. APPE_Init() :
   a. Configure the system power mode
   b. Initialize the timer server
   c. Init Debug
   d. Initialize all transport layers
7. Add a while(1) loop calling UTIL_SEQ_Run(UTIL_SEQ_DEFAULT)
   a. Sequencer where user actions/events are processed (advertising, connections, services andcharacteristics discovery, notification and related events).

The following pseudocode example illustrates the required initialization steps:

```
int main(void)
{/* Reset of all peripherals, Initializes the Flash interface and the Systick. */ HAL_Init();
/* USER CODE BEGIN Init */ Reset_Device(); Config_HSE();
/* USER CODE END Init */
/* Configure the system clock */ SystemClock_Config();
/* USER CODE BEGIN SysInit */ PeriphClock_Config();
Init_Exti(); /**< Configure the system Power Mode */
/* USER CODE END SysInit */
/* Initialize all configured peripherals */ MX_GPIO_Init(); MX_DMA_Init(); MX_RF_Init();
MX_RTC_Init(); APPE_Init();
/* Infinite loop */ while(1){
UTIL_SEQ_Run( UTIL_SEQ_DEFAULT ); }
}/* end main() */
```

Note:
1. *When performing the GATT_Init() & GAP_Init() APIs, STM32WB stack always add two standard services: Attribute Profile Service (0x1801) with Service Changed Characteristic and GAP Service (0x1800) with Device Name and Appearance characteristics.*
2. *The last attribute handles reserved for the standard GAP service is 0x000B when no privacy or host-based privacy is enabled on aci_gap_init() API, 0x000D when controller-based privacy is enabled on aci_gap_init() API.*

Table 31. GATT, GAP default services

Default services	Start handle	End handle	Service UUID
Attribute profile service	0x0001	0x0004	0x1801
Generic access profile (GAP) service	0x0005	0x000B	0x1800

Table 32. GATT, GAP default characteristics

Default services	Characteristic	Attribute handle	Char property	Char value handle	Char UUID	Char value length (bytes)
Attribute profile service						
	Service changed	0x0002	Indicate	0x0003	0x2A05	4
Generic access profile (GAP) service	-	-	-	-	-	-
-	Device came	0x0006	Read|write without response| write| authenticated signed writes	0x0007	0x2A00	8
-	Appearance	0x0008	Read|write without response| write| authenticated signed writes	0x0009	0x2A01	2
-	Peripheral preferred connection parameters	0x000A	Read| write	0x000B	0x2A04	8
-	Central address resolution[1]	0x000C	Readable without authentication or authorization. Not writable	0x000D	0x2AA6	1

1. *It is added only when controller-based privacy (0x02) is enabled on aci_gap_init() API.*

The `aci_gap_init()` role parameter values are as follows:

Table 33. **aci_gap_init() role parameter values**

Parameter	Role parameter values	Note
Role	0x01:Peripheral 0x02: Broadcaster 0x04: Central 0x08: Observer	The role parameter can be a bitwise OR of any of the supported values (multiple roles simultaneously support)

enable_Privacy	0x00 for disabling privacy;
	0x01 for enabling privacy;
	0x02 for enabling controller-based host privacy
		-

device_name_char_len	-	It allows the length of the device name characteristic to be indicated.

For a complete description of this API and related parameters refer to the Bluetooth LE stack APIs and event documentations, in Section 6 Reference documents.

4.1.1 BLE addresses

The following device addresses are supported from the STM32WB devices:

- Public address
- Random address
- Private address

Public MAC addresses (6 bytes- 48 bits address) uniquely identifies a BLE device, and they are defined by Institute of Electrical and Electronics Engineers (IEEE).

The first 3 bytes of the public address identify the company that issued the identifier and are known as the Organizationally Unique Identifier (OUI). An Organizationally Unique Identifier (OUI) is a 24-bit number that is purchased from the IEEE. This identifier uniquely identifies a company and it allows a block of possible public addresses to be reserved (up to 2^24 coming from the remaining 3 bytes of the public address) for the exclusive use of a company with a specific OUI.

An organization/company can request a new set of 6 bytes addresses when at least the 95% of previously allocated block of addresses have been used (up to 2^24 possible addresses are available with a specific OUI).

If the user wants to program his custom MAC address, he has to store it on a specific device Flash location used only for storing the MAC address. Then, at device power-up, it has to program this address on the radio by calling a specific stack API.

A valid preassigned MAC address is defined in the OTP. A specific public address can be set by the application.

The ACI command to set the MAC address is ACI_HAL_WRITE_CONFIG_DATA (opcode 0xFC0C) with command parameters as follow:

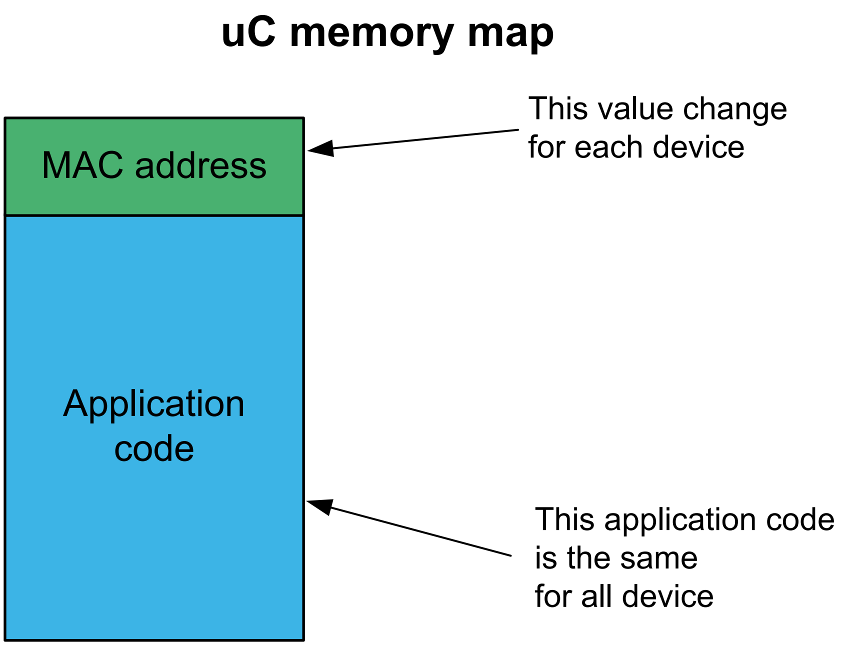
- Offset: 0x00 (0x00 identify the BTLE public address, i.e. MAC address)
- Length: 0x06 (Length of the MAC address)
- Value: 0xaabbccddeeff (48 bit array for MAC address)

The command ACI_HAL_WRITE_CONFIG_DATA should be sent before starting BLE operations (after each power-up or reset).

The following pseudocode example illustrates how to set a public address:

```
uint8_t bdaddr[] = {0x12, 0x34, 0x00, 0xE1, 0x80, 0x02};
ret=aci_hal_write_config_data(CONFIG_DATA_PUBADDR_OFFSET,CONFIG_DATA_PUBAD DR_LEN, bdaddr);
if(ret)PRINTF("Setting address failed.\n")}
```

Figure 10. **BLE MAC address storage**


uC memory map
MAC address
This value change for each device
Application code
This application code is the same for all device

The STM32WB devices do not have a valid preassigned MAC address, but a unique serial number (read only for the user).The unique serial number is a six byte value stored at address 0x100007F4: it is stored as two words (8 bytes) at address 0x100007F4 and 0x100007F8 with unique serial number padded with 0xAA55.

The static random address is generated and programmed at very 1st boot of the device on the dedicated Flash area. The value on Flash is the actual value the device uses: each time the user resets the device the stack checks if valid data are on the dedicated Flash area and it uses it (a special valid marker on FLASH is used to identify if valid data are present). If the user performs mass erase, the stored values (including marker) are removed so the stack generates a new random address and stores it on the dedicated flash.

Private addresses are used when privacy is enabled and according to the Bluetooth low energy specification. For more information about private addresses, refer to Section 2.7 Security manager (SM).

4.1.2 Set tx power level

During the initialization phase, the user can also select the transmitting power level using the following API:

```
aci_hal_set_tx_power_level(high, power level)
```

Follow a pseudocode example for setting the radio transmit power in high power and -2 dBm output power:

```
ret= aci_hal_set_tx_power_level (1,4);
```

For a complete description of this API and related parameters refer to the Bluetooth LE stack APIs and event documentation, in Section 6 Reference documents.

4.2 Services and characteristic configuration

In order to add a service and related characteristics, a user application has to define the specific profile to be addressed:

1. Standard profile defined by the Bluetooth SIG organization. The user must follow the profile specification and services, characteristic specification documents in order to implement them by using the related defined Profile, Services and Characteristics 16-bit UUID (refer to Bluetooth SIG web page: www.bluetooth.org/en-%20us/specification/adopted-specifications).
2. Proprietary, non-standard profile. The user must define its own services and characteristics. In this case, 128-bit UIDS are required and must be generated by profile implementers (refer to UUID generator web page: www.famkruithof.net/uuid/uuidgen).

A service can be added using the following command:

```
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);
```

This command returns the pointer to the service handle (`Service_Handle`), which is used to identify the service within the user application. A characteristic can be added to this service using the following command:

```
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);
```

This command returns the pointer to the characteristic handle (`Char_Handle`), which is used to identify the characteristic within the user application.

The following pseudocode example illustrates the steps to be followed to add a service and two associated characteristic to a proprietary, non-standard profile.

```
/* Service and characteristic UUIDs variables.*/
Service_UUID_t service_uuid;
Char_UUID_t char_uuid;


tBleStatus Add_Server_Services_Characteristics(void)
{
    tBleStatus ret = BLE_STATUS_SUCCESS;
    /*
    The following 128bits UUIDs have been generated from the random UUID
    generator:
    D973F2E0-B19E-11E2-9E96-0800200C9A66: Service 128bits UUID
    D973F2E1-B19E-11E2-9E96-0800200C9A66: Characteristic_1 128bits UUID
    D973F2E2-B19E-11E2-9E96-0800200C9A66: Characteristic_2 128bits UUID
    */
    /*Service 128bits UUID */
    const uint8_t uuid[16] =
    {0x66,0x9a,0x0c,0x20,0x00,0x08,0x96,0x9e,0xe2,0x11,0x9e,0xb1,0xe0,0xf2,0x73,0xd9};
    /*Characteristic_1 128bits UUID */
    const uint8_t charUuid_1[16] =
    {0x66,0x9a,0x0c,0x20,0x00,0x08,0x96,0x9e,0xe2,0x11,0x9e,0xb1,0xe1,0xf2,0x73,0xd9};
    /*Characteristic_2 128bits UUID */
    const uint8_t charUuid_2[16] =
    {0x66,0x9a,0x0c,0x20,0x00,0x08,0x96,0x9e,0xe2,0x11,0x9e,0xb1,0xe2,0xf2,0x73,0xd9};
    Osal_MemCpy(&service_uuid.Service_UUID_128, uuid, 16);
    /* Add the service with service_uuid 128bits UUID to the GATT server
    database. The service handle Service_Handle is returned.
    */
    ret = aci_gatt_add_service(UUID_TYPE_128, &service_uuid, PRIMARY_SERVICE,
                               6, &Service_Handle);
    if(ret != BLE_STATUS_SUCCESS) return(ret);
    Osal_MemCpy(&char_uuid.Char_UUID_128, charUuid_1, 16);

    /* Add the characteristic with charUuid_1128bitsUUID to the service
        Service_Handle. This characteristic has 20 as Maximum length of the
        characteristic value, Notify properties(CHAR_PROP_NOTIFY), no security
        permissions(ATTR_PERMISSION_NONE), no GATT event mask (0), 16 as key
        encryption size, and variable-length characteristic (1).
        The characteristic handle (CharHandle_1) is returned.
        */
    ret = aci_gatt_add_char(Service_Handle, UUID_TYPE_128, &char_uuid, 20,
                            CHAR_PROP_NOTIFY, ATTR_PERMISSION_NONE, 0,16, 1,
                            &CharHandle_1);
    if (ret != BLE_STATUS_SUCCESS) return(ret);
    Osal_MemCpy(&char_uuid.Char_UUID_128, charUuid_2, 16);

    /* Add the characteristic with charUuid_2 128bits UUID to the service
       Service_Handle. This characteristic has 20 as Maximum length of the
       characteristic value, Read, Write and Write Without Response properties,
       no security permissions(ATTR_PERMISSION_NONE), notify application when
       attribute is written (GATT_NOTIFY_ATTRIBUTE_WRITE) as GATT event mask ,
       16 as key encryption size, and variable-length characteristic (1). The
       characteristic handle (CharHandle_2) is returned.
    */
     ret = aci_gatt_add_char(Service_Handle, UUID_TYPE_128, &char_uuid, 20,
                            CHAR_PROP_WRITE|CHAR_PROP_WRITE_WITHOUT_RESP,
                            ATTR_PERMISSION_NONE, GATT_NOTIFY_ATTRIBUTE_WRITE,
                            16, 1, &&CharHandle_2);
    if (ret != BLE_STATUS_SUCCESS)return(ret) ;
}/*end Add_Server_Services_Characteristics() */
```

4.3 Create a connection: discoverable and connectable APIs

In order to establish a connection between a BLE GAP central (master) device and a BLE GAP peripheral (slave) device, the GAP discoverable/connectable modes and procedures can be used as described in Table 34. GAP mode APIs, Table 35. GAP discovery procedure APIs and Table 36. Connection procedure APIs and by using the related BLE stack APIs provided.

GAP peripheral discoverable and connectable modes APIs

Different types of discoverable and connectable modes can be used as described by the following APIs:

Table 34. **GAP mode APIs**

API	Supported advertising event types	Description
aci_gap_set_discoverable()	0x00: connectable undirected advertising (default)	Sets the device in general discoverable mode.
	0x02: scannable undirected advertising	The device is discoverable until the device issues the `aci_gap_set_non_discoverable()` API.
	0x03: non-connectable undirected advertising	
aci_gap_set_limited_discoverable()	0x00: connectable undirected advertising (default)	Sets the device in limited discoverable mode. The device is discoverable for a maximum period of TGAP (lim_adv_timeout) = 180 seconds. The advertising can be disabled at any time by calling `aci_gap_set_non_discoverable()` API.
	0x02: scannable undirected advertising	
	0x03: non-connectable undirected advertising	
aci_gap_set_non_discoverable()	NA	Sets the device in non- discoverable mode. This command disables the LL advertising and sets the device in standby state.
aci_gap_set_direct_connectable()	NA	Sets the device in direct connectable mode. The device is directed connectable mode only for 1.28 seconds. If no connection is established within this duration, the device enters non-discoverable mode and advertising has to be enabled again explicitly.
aci_gap_set_non_connectable()	0x02: scannable undirected advertising	Puts the device into non- connectable mode.
	0x03: non-connectable undirected advertising	
aci_gap_set_undirect_connectable ()	NA	Puts the device into undirected connectable mode.

Table 35. **GAP discovery procedure APIs**

API	Description
aci_gap_start_limited_discovery_proc()	Starts the limited discovery procedure. The controller is commanded to start active scanning. When this procedure is started, only the devices in limited discoverable mode are returned to the upper layers.
aci_gap_start_general_discovery_proc()	Starts the general discovery procedure. The controller is commanded to start active scanning.

Table 36. **Connection procedure APIs**

API	Description
aci_gap_start_auto_connection_establish_proc()	Starts the auto connection establishment procedure. The devices specified are added to the white list of the controller and a create connection call is made to the controller by GAP with the initiator filter policy set to “use whitelist to determine which advertiser to connect to”.
aci_gap_create_connection()	Starts the direct connection establishment procedure. A create connection call is made to the controller by GAP with the initiator filter policy set to “ignore whitelist and process connectable advertising packets only for the specified device”.
aci_gap_start_general_connection_establish_proc()	Starts a general connection establishment procedure. The device enables scanning in the controller with the scanner filter policy set to “accept all advertising packets” and from the scanning results, all the devices are sent to the upper layer using the event `callback hci_le_advertising_report_event()`.
aci_gap_start_selective_connection_establish_proc()	It starts a selective connection establishment procedure. The GAP adds the specified device addresses into white list and enables scanning in the controller with the scanner filter policy set to “accept packets only from devices in white list”. All the devices found are sent to the upper layer by the event callback `hci_le_advertising_report_event()`.
aci_gap_terminate_gap_proc()	Terminate the specified GAP procedure.

4.3.1 Set discoverable mode and use direct connection establishment procedure

The following pseudocode example illustrates only the specific steps to be followed to let a GAP peripheral device be in general discoverable mode, and for a GAP central device direct connect to it through a direct connection establishment procedure.

Note: It is assumed that the device public address has been set during the initialization phase as follows:

```
uint8_t bdaddr[] = {0x12, 0x34, 0x00, 0xE1, 0x80, 0x02};
ret=aci_hal_write_config_data(CONFIG_DATA_PUBADDR_OFFSET,CONFIG_DATA_PUBAD DR_LEN, bdaddr);
if(ret != BLE_STATUS_SUCCESS)PRINTF("Failure.\n");

/*GAP Peripheral: general discoverable mode (and no scan response is sent)
*/

void GAP_Peripheral_Make_Discoverable(void )
{
 tBleStatus ret;
const charlocal_name[]=
{AD_TYPE_COMPLETE_LOCAL_NAME,'S','T','M','3','2','W','B','x','5','T','e','s','t'};/* disable
scan response: passive scan */ hci_le_set_scan_response_data (0,NULL);

 /* disable scan response: passive scan */
 hci_le_set_scan_response_data (0,NULL);

 /* Put the GAP peripheral in general discoverable mode:
    Advertising_Type: ADV_IND(undirected scannable and connectable);
    Advertising_Interval_Min: 100;
    Advertising_Interval_Max: 100;
    Own_Address_Type: PUBLIC_ADDR (public address: 0x00);
    Adv_Filter_Policy: NO_WHITE_LIST_USE (no whit list is used);
    Local_Name_Lenght: 14
    Local_Name: STM32WBx5Test;
    Service_Uuid_Length: 0 (no service to be advertised); Service_Uuid_List: NULL;
    Slave_Conn_Interval_Min: 0 (Slave connection internal minimum value);
    Slave_Conn_Interval_Max:  0 (Slave connection internal maximum value).
    */

  ret = aci_gap_set_discoverable(ADV_IND, 100, 100, PUBLIC_ADDR,
                              NO_WHITE_LIST_USE,
                              sizeof(local_name),
                              local_name,
                              0, NULL, 0, 0);
  if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");
} /* end GAP_Peripheral_Make_Discoverable() */

/*GAP Central: direct connection establishment procedure to connect to the
GAP Peripheral in discoverable mode
*/

void GAP_Central_Make_Connection(void)

{
 /*Start the direct connection establishment procedure to the GAP
   peripheral device in general discoverable mode using the
   following connection parameters:
   LE_Scan_Interval: 0x4000;
   LE_Scan_Window: 0x4000;
   Peer_Address_Type: PUBLIC_ADDR (GAP peripheral address type: public
   address);
   Peer_Address: {0xaa, 0x00, 0x00, 0xE1, 0x80, 0x02};
   Own_Address_Type:
   PUBLIC_ADDR (device address type);
   Conn_Interval_Min: 40 (Minimum value for the connection event
   interval);
   Conn_Interval_Max: 40 (Maximum value for the connection event
   interval);
   Conn_Latency: 0 (Slave latency for the connection in a number of
   connection events);
   Supervision_Timeout: 60 (Supervision timeout for the LE Link);
   Minimum_CE_Length: 2000 (Minimum length of connection needed for the
   LE connection);
   Maximum_CE_Length: 2000 (Maximum length of connection needed for the LE connection).

  */

   tBDAddr GAP_Peripheral_address = {0xaa, 0x00, 0x00, 0xE1, 0x80, 0x02};
   ret= aci_gap_create_connection(0x4000, 0x4000, PUBLIC_ADDR,
                                  GAP_Peripheral_address,PUBLIC_ADDR, 40,
                                  40,
                                  0, 60, 2000 , 2000);
   if(ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");

}/* GAP_Central_Make_Connection(void )*/
```

Note:

1. *If `ret = BLE_STATUS_SUCCESS` is returned, on termination of the GAP procedure, the event callback `hci_le_connection_complete_event()` is called, to indicate that a connection has been established with the GAP_Peripheral_address (same event is returned on the GAP peripheral device).*
2. *The connection procedure can be explicitly terminated by issuing the API `aci_gap_terminate_gap_proc()`.*
3. *The last two parameters `Minimum_CE_Length` and `Maximum_CE_Length` of the `aci_gap_create_connection()` are the length of the connection event needed for the BLE connection. These parameters allows user to specify the amount of time the master has to allocate for a single slave so they must be wisely chosen. In particular, when a master connects to more slaves, the connection interval for each slave must be equal or a multiple of the other connection intervals and user must not overdo the connection event length for each slave. Refer to Section 5 BLE multiple connection timing strategy for detailed information about the timing allocation policy.*

4.3.2 Set discoverable mode and use general discovery procedure (active scan)

The following pseudocode example illustrates only the specific steps to be followed to let a GAP Peripheral device be in general discoverable mode, and for a GAP central device start a general discovery procedure in order to discover devices within its radio range.

Note: It is assumed that the device public address has been set during the initialization phase as follows:

```
    uint8_t bdaddr[] = {0x12, 0x34, 0x00, 0xE1, 0x80, 0x02};
    ret = aci_hal_write_config_data(CONFIG_DATA_PUBADDR_OFFSET,
                                    CONFIG_DATA_PUBADDR_LEN,
                                    bdaddr);
    if (ret != BLE_STATUS_SUCCESS)PRINTF("Failure.\n");

/* GAP Peripheral:general discoverable mode (scan responses are sent):
*/
void GAP_Peripheral_Make_Discoverable(void)
{
   tBleStatus ret;
   const char local_name[] =
{AD_TYPE_COMPLETE_LOCAL_NAME,'S','T','M','3','2','W','B','x','5',}; /* As scan
response data, a proprietary 128bits Service UUID is used.
     This 128bits data cannot be inserted within the advertising packet
     (ADV_IND) due its length constraints (31 bytes).
     AD Type description:
     0x11: length
     0x06: 128 bits Service UUID type
     0x8a,0x97,0xf7,0xc0,0x85,0x06,0x11,0xe3,0xba,0xa7,0x08,0x00,0x20,0x0c,
     0x9a,0x66: 128 bits Service UUID
   */
   uint8_t ServiceUUID_Scan[18]=
{0x11,0x06,0x8a,0x97,0xf7,0xc0,0x85,0x06,0x11,0xe3,0xba,0xa7,0x08,0x00,0x2,0x0c,0x9a,0x66};
/* Enable scan response to be sent when GAP peripheral receives scan
   requests from GAP Central performing general
   discovery procedure(active scan) */

 hci_le_set_scan_response_data(18,ServiceUUID_Scan);
 /* Put the GAP peripheral in general discoverable mode:
   Advertising_Type: ADV_IND (undirected scannable and connectable);
Advertising_Interval_Min: 100;
   Advertising_Interval_Max: 100;
   Own_Address_Type: PUBLIC_ADDR (public address: 0x00); Advertising_Filter_Policy:
NO_WHITE_LIST_USE (no whit list is used);
   Local_Name_Length: 8
   Local_Name: STM32WB;
   Service_Uuid_Length: 0 (no service to be advertised); Service_Uuid_List: NULL;
   Slave_Conn_Interval_Min: 0 (Slave connection internal minimum value);
Slave_Conn_Interval_Max: 0 (Slave connection internal maximum value).
  */
  ret = aci_gap_set_discoverable(ADV_IND, 100, 100, PUBLIC_ADDR,
                                NO_WHITE_LIST_USE,sizeof(local_name),
                                local_name, 0, NULL, 0, 0);
 if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");

} /* end GAP_Peripheral_Make_Discoverable() */

/*GAP Central: start general discovery procedure to discover the GAP peripheral device in
discoverable mode */
void GAP_Central_General_Discovery_Procedure(void)
{
tBleStatus ret;

/* Start the general discovery procedure(active scan) using the following
   parameters:
   LE_Scan_Interval: 0x4000;
   LE_Scan_Window: 0x4000;
   Own_address_type: 0x00 (public device address);
   Filter_Duplicates: 0x00 (duplicate filtering disabled);
*/
ret =aci_gap_start_general_discovery_proc(0x4000,0x4000,0x00,0x00);
if (ret != BLE_STATUS_SUCCESS)PRINTF("Failure.\n");
}
```

The responses of the procedure are given through the event callback `hci_le_advertising_report_event()`.The end of the procedure is indicated by `aci_gap_proc_complete_event()` event callback with `Procedure_Code` parameter equal to `GAP_GENERAL_DISCOVERY_PROC (0x2)`.

```
 /* This callback is called when an advertising report is received */
 void hci_le_advertising_report_event(uint8_t Num_Reports,
                                      Advertising_Report_t
                                      Advertising_Report[])
{
     /* Advertising_Report contains all the expected parameters.
        User application should add code for decoding the received
        Advertising_Report event databased on the specific evt_type
        (ADV_IND, SCAN_RSP, ..)
     */

     /* Example: store the received Advertising_Report fields */
     uint8_t bdaddr[6];

     /* type of the peer address (PUBLIC_ADDR,RANDOM_ADDR) */
     uint8_t bdaddr_type = Advertising_Report[0].Address_Type;

    /* event type (advertising packets types) */
    uint8_t evt_type = Advertising_Report[0].Event_Type ;

    /* RSSI value */
    uint8_t RSSI = Advertising_Report[0].RSSI;

    /* address of the peer device found during discovery procedure */
    Osal_MemCpy(bdaddr, Advertising_Report[0].Address,6);

    /* length of advertising or scan response data */
    uint8_t data_length = Advertising_Report[0].Length_Data;

    /* data_length octets of advertising or scan response data formatted are
       on Advertising_Report[0].Data field: to be stored/filtered based on
       specific user application scenario*/

} /* hci_le_advertising_report_event() */
```

In particular, in this specific context, the following events are raised on the GAP central `hci_le_advertising_report_event ()` , as a consequence of the GAP peripheral device in discoverable mode with scan response enabled:

1. `Advertising Report` event with advertising packet type (evt_type =ADV_IND)
2. `Advertising Report` event with scan response packet type (evt_type =SCAN_RSP)

Table 37. ADV_IND event type

Event type	Address type	Address	Advertising data	RSSI
0x00 (ADV_IND)	0x00 (public address)	0x0280E1003 412	0x02, 0x01, 0x06, 0x08, 0x0A, 0x53, 0x54, 0x4D, 0x33, 0x32, 0x57, 0x42, 0x78, 0x35	0xCE

The advertising data can be interpreted as follows (refer to Bluetooth specification version in Section 6 Reference documents):

Table 38. **ADV_IND advertising data**

Flags AD type field	Local name field	Tx power level
0x02: length of the field 0x01: AD type flags 0x06: 0x110 (Bit 2: BR/EDR Not supported; bit 1: general discoverable mode)	0x09: length of the field 0x0A: complete local name type 0x53, 0x54, 0x4D, 0x33, 0x32, 0x57, 0x48, 0x78, 0x35: STM32WB	0x02: length of the field 0x0A: Tx power type 0x08: power value

Table 39. **SCAN_RSP event type**

Event type	Address type	Address	Scan response data	RSSI
0x04 (SCAN_RS P)	0x01 (random address)	0x0280E1003412	0x12,0x66,0x9A,0x0C, 0x20,0x00,0x08,0xA7,0xBA,0xE3,0x11,0x06,0x85,0xC0,0xF7,0x97,0x8A,0x06,0x11	0xDA

The scan response data can be interpreted as follows (refer to Bluetooth specifications):

Table 40. **Scan response data**

Scan response data
0x12: data length 0x11: length of service UUID advertising data; 0x06: 128 bits service UUID type; 0x66,0x9A,0x0C,0x20,0x00,0x08,0xA7,0xBA,0xE3,0x11,0x06,0x85,0xC0,0xF7,0x97,0x8A: 128-bit service UUID

4.4 BLE stack events and event callbacks

In order to handle ACI events in its application, the user can choose between two different methods:

- Use nested "switch case" event handler
- Use event callbacks framework

Based on its own application scenario, the user has to identify the required device events to be detected and handled and the application specific actions to be done as consequence of such events.

When implementing a BLE application, the most common and widely used device events are the ones related to the discovery, connection, terminate procedures, services and characteristics discovery procedures, attribute modified events on a GATT server and attribute notification/ indication events on a GATT client.

Table 41. **BLE stack: main events callbacks**

Event callback	Description	Where
hci_disconnection_complete_event()	A connection is terminated	GAP central/ peripheral
hci_le_connection_complete_event()	Indicates to both of the devices forming the connection that a new connection has been established	GAP central/ peripheral
aci_gatt_attribute_modified_event()	Generated by the GATT server when a client modifies any attribute on the server, if event is enabled	GATT server
aci_gatt_notification_event()	Generated by the GATT client when a server notifies any attribute on the client	GATT client

Event callback	Description	Where
aci_gatt_indication_event()	Generated by the GATT client when a server indicates any attribute on the client	GATT client
aci_gap_pass_key_req_event()	Generated by the Security manager to the application when a passkey is required for pairing. When this event is received, the application has to respond with the aci_gap_pass_key_resp() API	GAP central/ peripheral
aci_gap_pairing_complete_event()	Generated when the pairing process has completed successfully or a pairing procedure timeout has occurred or the pairing has failed	GAP central/ peripheral
aci_gap_bond_lost_event()	Event generated when a pairing request is issued, in response to a slave security request from a master which has previously bonded with the slave. When this event is received, the upper layer has to issue the command aci_gap_allow_rebond() to allow the slave to continue the pairing process with the master	GAP peripheral
aci_att_read_by_group_type_resp_event()	The Read-by-group type response is sent in reply to a received Read-by-group type request and contains the handles and values of the attributes that have been read	GATT client
aci_att_read_by_type_resp_event()	The Read-by-type response is sent in reply to a received Read-by-type Request and contains the handles and values of the attributes that have been read	GATT client
aci_gatt_proc_complete_event()	A GATT procedure has been completed	GATT client
hci_le_advertising_report_event	Event given by the GAP layer to the upper layers when a device is discovered during scanning as a consequence of one of the GAP procedures started by the upper layers	GAP central

For a detailed description about the BLE events, and related formats refer to the STM32WB Bluetooth LE stack APIs and events documentation, in Section 6 Reference documents.

The following pseudocode provides an example of events callbacks handling some of the described BLE stack events (disconnection complete event, connection complete event, GATT attribute modified event , GATT notification event):

```
/* This event callback indicates the disconnection from a peer device.
   It is called in the BLE radio interrupt context.
*/
void hci_disconnection_complete_event(uint8_t Status,
                                      uint16_t Connection_Handle,
                                      uint8_t Reason)
{
    /* Add user code for handling BLE disconnection complete event based on
       application scenario.
    */
}/* end hci_disconnection_complete_event() */

/* This event callback indicates the end of a connection procedure.
*/
void hci_le_connection_complete_event(uint8_t Status,
                                      uint16_t Connection_Handle,
                                      uint8_t Role,
                                      uint8_t Peer_Address_Type,
                                      uint8_t Peer_Address[6],
                                      uint16_t Conn_Interval,
                                      uint16_t Conn_Latency,
                                      uint16_t Supervision_Timeout,
                                      uint8_t Master_Clock_Accuracy)

{
    /* Add user code for handling BLE connection complete event based on
       application scenario.
    */

/* Store connection handle */
 connection_handle = Connection_Handle;
 …
}/* end hci_le_connection_complete_event() */
```

```
#if GATT_SERVER

/* This event callback indicates that an attribute has been modified from a
   peer device.
*/
void aci_gatt_attribute_modified_event(uint16_t Connection_Handle,
                                       uint16_t Attr_Handle,
                                       uint16_t Offset,
                                       uint8_t Attr_Data_Length,
                                       uint8_t Attr_Data[])
{
    /* Add user code for handling attribute modification event based on
       application scenario.
    */
    ...
} /* end aci_gatt_attribute_modified_event() */

#endif /* GATT_SERVER */

#if GATT_CLIENT
/* This event callback indicates that an attribute notification has been
 received from a peer device.
*/
void aci_gatt_notification_event(uint16_t Connection_Handle,
                                 uint16_t Attribute_Handle,
                                 uint8_t Attribute_Value_Length,
                                 uint8_t Attribute_Value[])
{
 /* Add user code for handling attribute modification event based on
    application scenario.
    */
…
} /* end aci_gatt_notification_event() */
#endif /* GATT_CLIENT */
```

4.5 Security (pairing and bonding)

This section describes the main functions to be used in order to establish a pairing between two devices (authenticate the device identity, encrypt the link and distribute the keys to be used on next re-connections).

To successfully pair with a device, IO capabilities have to be correctly configured, depending on the IO capability available on the selected device.

`aci_gap_set_io_capability(io_capability)` should be used with one of the following io_capability values:

```
0x00: 'IO_CAP_DISPLAY_ONLY'
0x01: 'IO_CAP_DISPLAY_YES_NO',
0x02: 'KEYBOARD_ONLY'
0x03: 'IO_CAP_NO_INPUT_NO_OUTPUT'
0x04: 'IO_CAP_KEYBOARD_DISPLAY’
```

PassKey Entry example with 2 STM32WB devices: Device_1, Device_2

The following pseudocode example illustrates only the specific steps to be followed to pair two devices by using the PassKey entry method.

As described in Table 11. Methods used to calculate the temporary key (TK), Device_1, Device_2 have to set the IO capability in order to select PassKey entry as a security method.

On this particular example, "Display Only" on Device_1 and "Keyboard Only" on Device_2 are selected, as follows:

```
/*Device_1:
*/ tBleStatus ret;\
ret= aci_gap_set_io_capability(IO_CAP_DISPLAY_ONLY);
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");

/*Device_2:
*/ tBleStatus ret;
ret= aci_gap_set_io_capability(IO_CAP_KEYBOARD_ONLY);
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");
```

Once the IO capability are defined, the `aci_gap_set_authentication_requirement()` should be used to set all the security authentication requirements the device needs (MITM mode (authenticated link or not), OOB data present or not, use fixed pin or not, enabling bonding or not).

The following pseudocode example illustrates only the specific steps to be followed to set the authentication requirements for a device with: “MITM protection , No OOB data, don’t use fixed pin”: this configuration is used to authenticate the link and to use a not fixed pin during the pairing process with PassKey Method.

```
ret=aci_gap_set_authentication_requirement(BONDING,/*bonding is
                                                         enabled */
                                           MITM_PROTECTION_REQUIRED,
                                           SC_IS_SUPPORTED,/*Secure connection
                                                              supported
                                                              but optional */
                                           KEYPRESS_IS_NOT_SUPPORTED,
                                           7, /* Min encryption key size */
                                           16, /* Max encryption
                                                  key size */
                                           0x01, /* fixed pin is not used*/
                                           0x123456, /* fixed pin */
                                           0x00 /* Public Identity address type */);
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");
```

Once the security IO capability and authentication requirements are defined, an application can initiate a pairing procedure as follows:

1. By using `aci_gap_slave_security_req()` on a GAP peripheral (slave) device (it sends a slave security request to the master):

```
tBleStatus ret;
ret= aci_gap_slave_security_req(conn_handle,
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");
```

- Or by using the `aci_gap_send_pairing_req()` on a GAP central (master) device.

Since the no fixed pin has been set,once the paring procedure is initiated by one of the two devices, BLE device calls the `aci_gap_pass_key_req_event()` event callback (with related connection handle) to ask the user application to provide the password to be used to establish the encryption key. BLE application has to provide the correct password by using the `aci_gap_pass_key_resp(conn_handle,passkey)` API.

When the `aci_gap_pass_key_req_event()` callback is called on Device_1, it should generate a random pin and set it through the `aci_gap_pass_key_resp()` API, as follows:

```
void aci_gap_pass_key_req_event(uint16_t Connection_Handle)
{
   tBleStatus ret;
   uint32_t pin;
   /*Generate a random pin with an user specific function */
   pin = generate_random_pin();
   ret= aci_gap_pass_key_resp(Connection_Handle,pin);
   if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");
}
```

Since the Device_1, I/O capability is set as “Display Only”, it should display the generated pin in the device display. Since Device_2 , I/O capability is set as “Keyboard Only”, the user can provide the pin displayed on Device_1 to the Device_2 though the same `aci_gap_pass_key_resp()` API, by a keyboard.

Alternatively, if the user wants to set the authentication requirements with a fixed pin 0x123456 (no pass key event is required), the following pseudocode can be used:

```
tBleStatus ret;

ret= aci_gap_set_auth_requirement(BONDING, /* bonding is
                                  enabled */
                                  MITM_PROTECTION_REQUIRED,
                                  SC_IS_SUPPORTED, /* Secure
                                  connection supported
                                  but optional */
                                  KEYPRESS_IS_NOT_SUPPORTED,
                                  7, /* Min encryption
                                  key size */
                                  16, /* Max encryption
                                  key size */
                                  0x00, /* fixed pin is used*/
                                  0x123456, /* fixed pin */
                                  0x00 /* Public Identity address
                                                      type */);
if (ret != BLE_STATUS_SUCCESS) PRINTF("Failure.\n");
```

Note:

1. *When the pairing procedure is started by calling the described APIs (`aci_gap_slave_security_req()` or`aci_gap_send_pairing_req()` and the value `ret= BLE_STATUS_SUCCESS` is returned, on termination of the procedure, a `aci_gap_pairing_complete_event()` is returned to the event callback to indicate the pairing status:*
   - *0x00: Success*
   - *0x01: SMP timeout*
   - *0x02: Pairing failed*

   The pairing status is given from the status field of the `aci_gap_pairing_complete_event()`

   The reason parameter provides the pairing failed reason code in case of failure (0 if status parameter returns success or timeout).
2. *When 2 devices get paired, the link is automatically encrypted during the first connection. If bonding is also enabled (keys are stored for a future time), when the 2 devices get connected again, the link can be simply encrypted (without no need to perform again the pairing procedure).User applications can simply use the same APIs, which do not perform the paring process but just encrypt the link:*
   - *`aci_gap_slave_security_req )` on the GAP peripheral (slave) device or*
   - *`aci_gap_send_pairing_req()` on the GAP central (master) device.*
3. *If a slave has already bonded with a master, it can send a slave security request to the master to encrypt the link. When receiving the slave security request, the master may encrypt the link, initiate the pairing procedure, or reject the request. Typically, the master only encrypts the link, without performing the pairing procedure. Instead, if the master starts the pairing procedure, it means that for some reasons, the master lost its bond information, so it has to start the pairing procedure again. As a consequence, the slave device calls the `aci_gap_bond_lost_event()`event callback to inform the user application that it is not bonded anymore with the master it was previously bonded. Then, the slave application can decide to allow the security manager to complete the pairing procedure and re-bond with the master by calling the command `aci_gap_allow_rebond()`, or just close the connection and inform the user about the security issue.*
4. *Alternatively, the out-of-band method can be selected by calling the `aci_gap_set_oob_data()` API. This implies that both devices are using this method and they are setting the same OOB data defined through an out of band communication (example: NFC).*
5. *Moreover, the “secure connections” feature can be used by setting to 2 the SC_Support field of the `aci_gap_set_authentication_requirement()` API.*

4.5.1 Flow charts on pairing procedure: Pairing request by Master sequence (Legacy)

Flow charts on pairing procedure: Pairing request by Master sequence (Legacy)

The following flow chart illustrates specific steps to be followed from Master to create a security link in Legacy mode

It is assumed that the device public has been set during the initialization phase as follows:

```
Initialization:
Aci_gap_set_IO_capability(keyboard/display)
Aci_gap_set_auth_requirement(MITM,fixed pin,bonding=1,SC_Support=0x00)
```

Figure 11. **Pairing request initiated by master sequence (Legacy) 1/3**

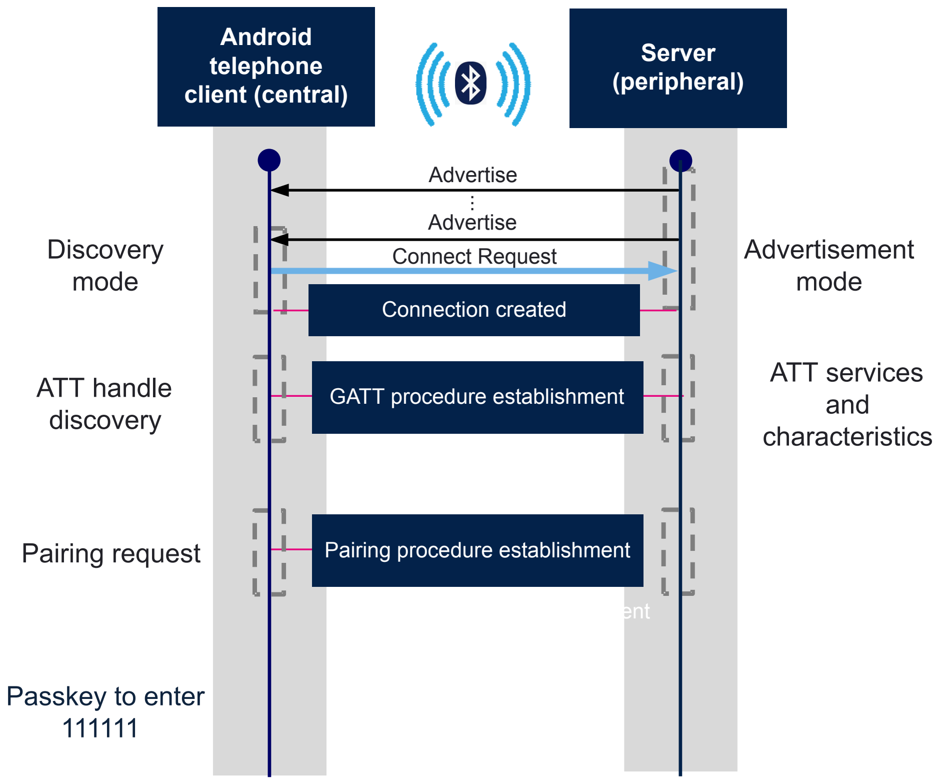


Figure 12. **Pairing request initiated by master sequence (Legacy) 2/3**

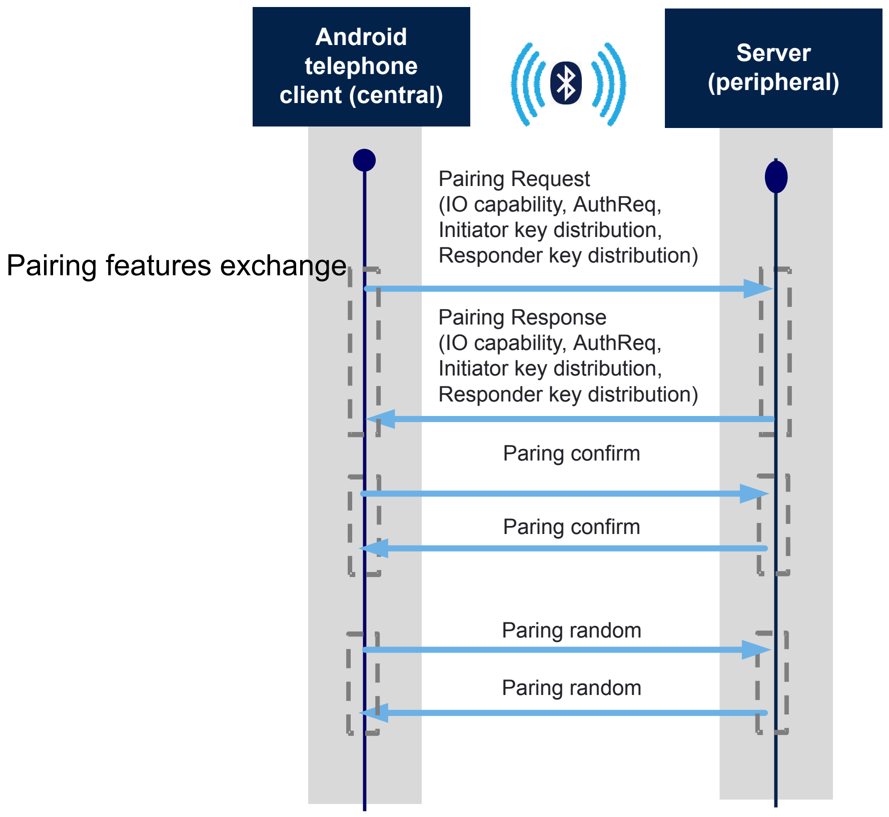
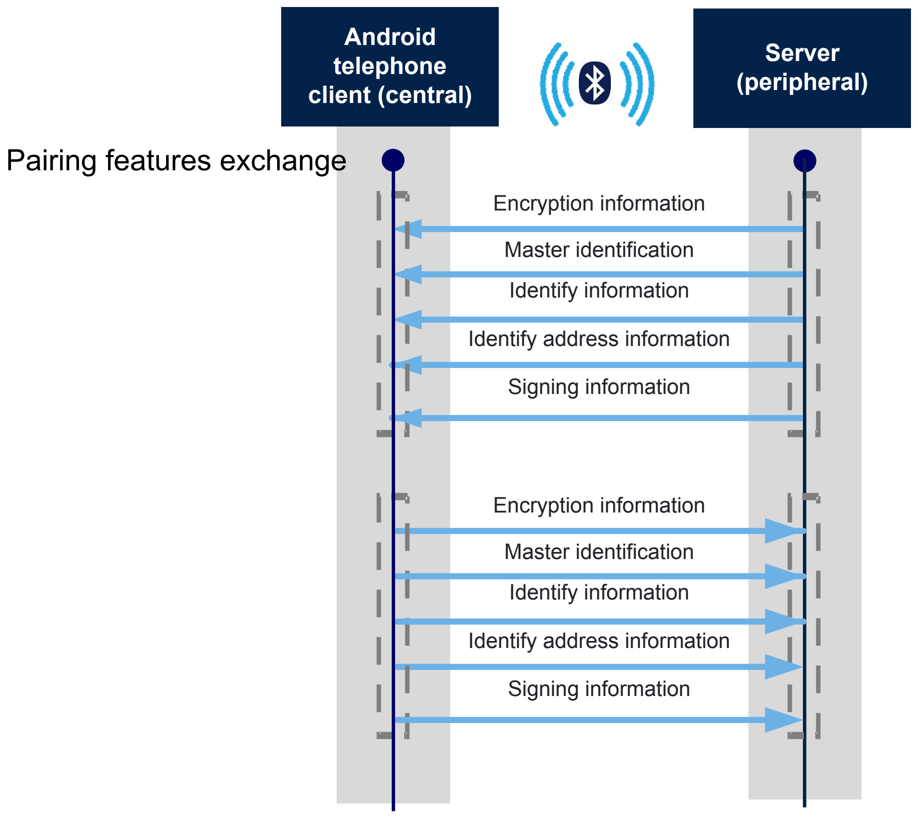

Android telephone client (central)
Server (peripheral)
Pairing features exchange
Pairing Request (IO capability, AuthReq, Initiator key distribution, Responder key distribution)
Pairing Response (IO capability, AuthReq, Initiator key distribution, Responder key distribution)
Paring confirm
Paring confirm
Paring random
Paring random

Figure 13. **Pairing request initiated by master sequence (Legacy) 3/3**



4.5.2 Flow charts on pairing procedure: Pairing request by Master sequence (Secure)

Flow charts on pairing procedure: Pairing request by Master sequence (Secure)

The following flow chart illustrates specific steps to be followed from Master to create a security link in secure mode.

It is assumed that the device public has been set during the initialization phase as follows:

```
Initialization:
Aci_gap_set_IO_capability(display_yes_no)
Aci_gap_set_auth_requirement(MITM,no
fixed pin,bonding=1,SC only mode)
```

Figure 14. Pairing request initiated by master sequence (Secure Connection) 1/3

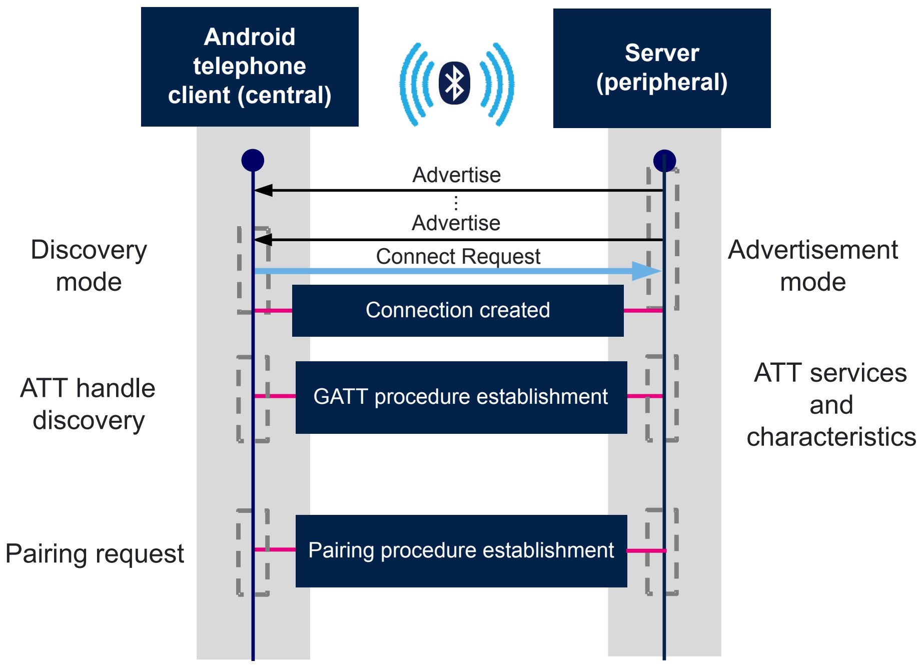


Figure 15. Pairing request initiated by master sequence (Secure Connection) 2/3

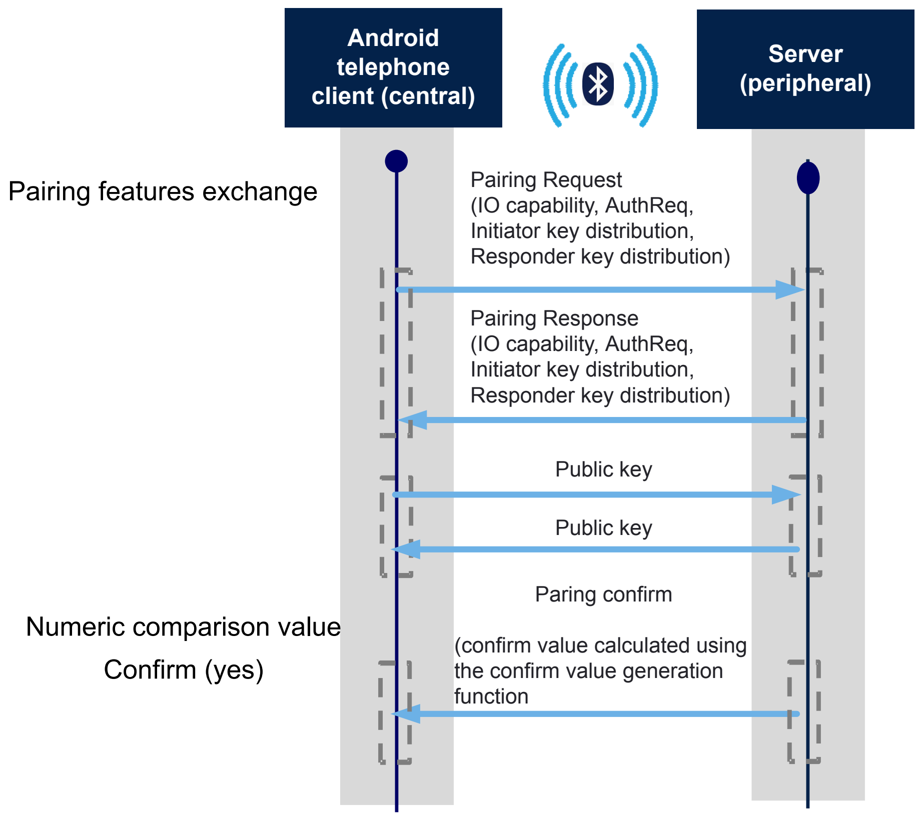
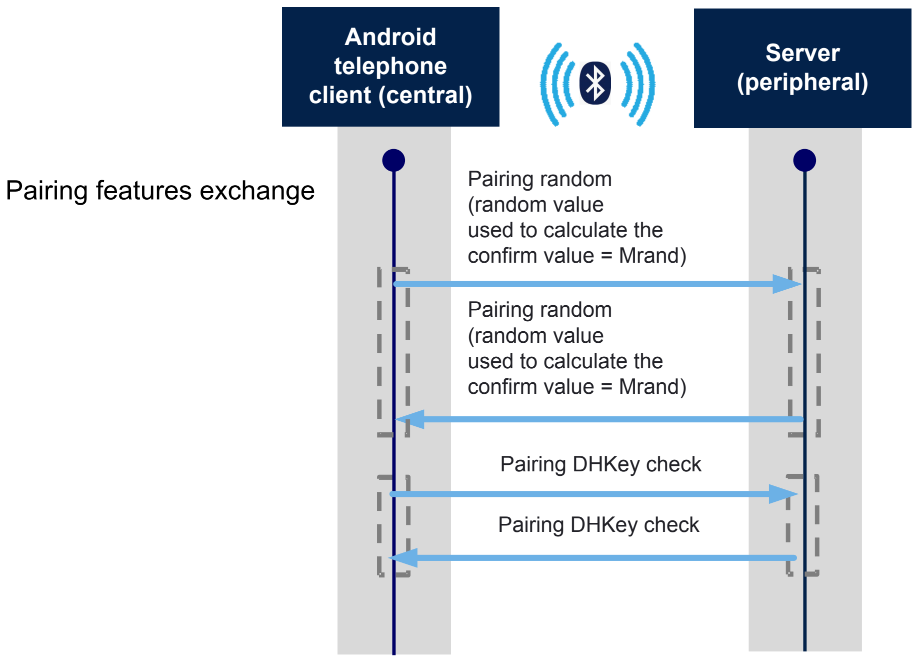

Android telephone client (central)
Server (peripheral)
Pairing features exchange
Pairing Request (IO capability, AuthReq, Initiator key distribution, Responder key distribution)
Pairing Response (IO capability, AuthReq, Initiator key distribution, Responder key distribution)
Public key
Public key
Paring confirm
Numeric comparison value
Confirm (yes)
(confirm value calculated using the confirm value generation function

Figure 16. Pairing request initiated by master sequence (Secure Connection) 3/3


Android telephone client (central)
Server (peripheral)
Pairing features exchange
Pairing random (random value used to calculate the confirm value = Mrand)
Pairing random (random value used to calculate the confirm value = Mrand)
Pairing DHKey check
Pairing DHKey check

4.5.3 Flow charts on pairing procedure: Pairing request by Slave sequence (secure)

Flow charts on pairing procedure: Pairing request by Slave sequence (Secure).

The following flow chart illustrates specific steps to be followed from Master to create a security link in security mode

It is assumed that the device public has been set during the initialization phase as follows:

Initialization:

Aci_gap_set_IO_capability(display_yes_no)

Aci_gap_set_auth_requirement(MITM,no

fixed pin,bonding=1,SC only mode)

```
Initialization:
Aci_gap_set_IO_capability(display_yes_no)
Aci_gap_set_auth_requirement(MITM,no
fixed pin,bonding=1,SC only mode)
```

Figure 17. Pairing request initiated by slave sequence (Secure Connection) 1/2

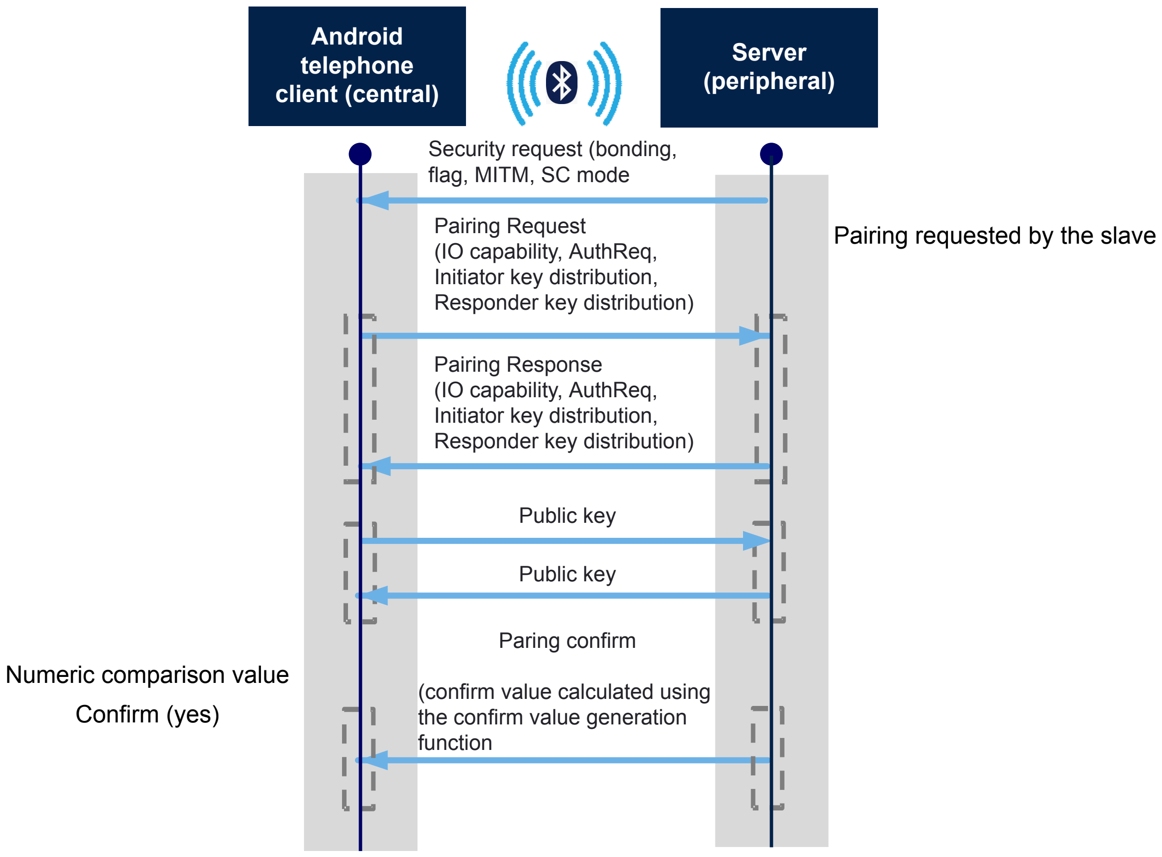
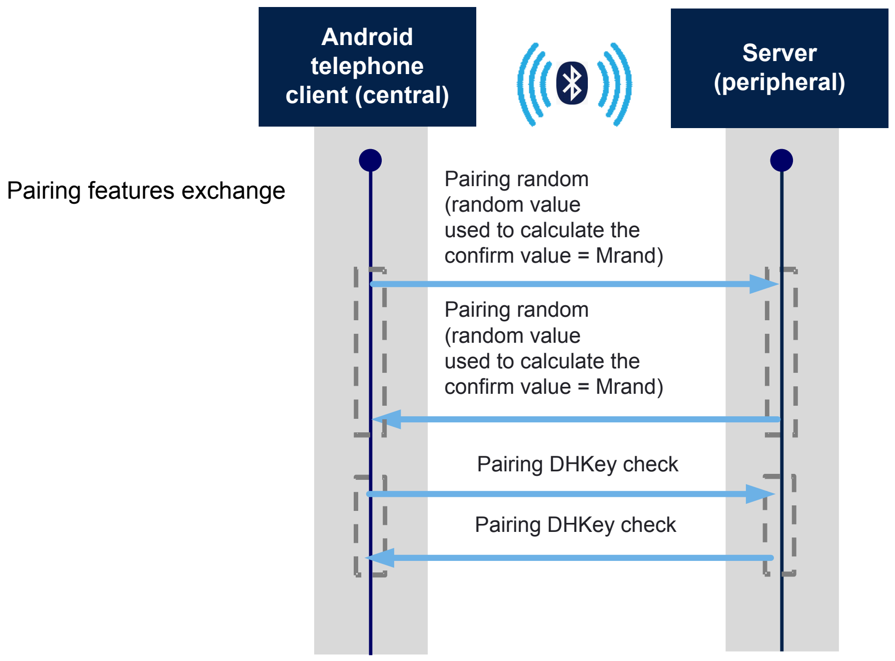

Android telephone client (central)
Server (peripheral)
Security request (bonding, flag, MITM, SC mode
Pairing requested by the slave
Pairing Request (IO capability, AuthReq, Initiator key distribution, Responder key distribution)
Pairing Response (IO capability, AuthReq, Initiator key distribution, Responder key distribution)
Public key
Public key
Paring confirm
(confirm value calculated using the confirm value generation function
Numeric comparison value
Confirm (yes)

Figure 18. **Pairing request initiated by slave sequence (Secure Connection) 2/2**


Android telephone client (central)
Server (peripheral)
Pairing features exchange
Pairing random (random value used to calculate the confirm value = Mrand)
Pairing random (random value used to calculate the confirm value = Mrand)
Pairing DHKey check
Pairing DHKey check

4.6 Service and characteristic discovery

This section describes the main functions allowing a STM32WB GAP central device to discover the GAP peripheral services and characteristics, once both devices are connected.

The P2PServer service & characteristics with related handles is used as reference service and characteristics on the following pseudo-code examples.

Further, it is assumed that a GAP central device (P2PClient application) is connected to a GAP peripheral device running the P2PServer application. The GAP central device uses the service and discovery procedures to find the GAP Peripheral P2PServer service and characteristics. The GAP central device is running the P2PClient application.

Table 42. **BLE sensor profile demo services and characteristic handle**

Service	Characteristic	Service / characteristic handle	Characteristic value handle	Characteristic client descriptor configuration handle	Characteristic format handle
Peer To Peer	NA	0x000C	NA	NA	NA
-	LED	0x000D	0x000E	NA	NA
-	Button	0x000F	0x0010	0x0011	NA

Note: The different attribute value handles are due to the last attribute handle reserved for the standard GAP service. On the following example, the STM32WB GAP peripheral P2PServer service is defining only the LED characteristic and Button characteristic. For detailed information about tP2Pserver refer to Section 6 Reference documents.

A list of the service discovery APIs with related description as follows:

Table 43. Service discovery procedures APIs

Discovery service API	Description
aci_gatt_disc_all_primary_services()	This API starts the GATT client procedure to discover all primary services on the GATT server. It is used when a GATT client connects to a device and it wants to find all the primary services provided on the device to determine what it can do.
aci_gatt_disc_primary_service_by_uuid()	This API starts the GATT client procedure to discover a primary service on the GATT server by using its UUID. It is used when a GATT client connects to a device and it wants to find a specific service without the need to get any other services.
aci_gatt_find_included_services()	This API starts the procedure to find all included services. It is used when a GATT client wants to discover secondary services once the primary services have been discovered.

The following pseudocode example illustrates the `aci_gatt_disc_all_primary_services() API`:

```
/*GAP Central starts a discovery all services procedure:
conn_handle is the connection handle returned on
hci_le_advertising_report_event() event callback
*/
if (aci_gatt_disc_all_primary_services(conn_handle) !=BLE_STATUS_SUCCESS)
{
   PRINTF("Failure.\n");

}
```

The responses of the procedure are given through the `aci_att_read_by_group_type_resp_event()` event callback. The end of the procedure is indicated by `aci_gatt_proc_complete_event()` event callback() call.

```
/* This event is generated in response to a Read By Group Type
Request: refer to aci_gatt_disc_all_primary_services() */
void aci_att_read_by_group_type_resp_event(uint16_t Conn_Handle,
                                           uint8_t
Attr_Data_Length,
                                           uint8_t Data_Length,
                                           uint8_t Att_Data_List[]);

{
/*
 Conn_Handle: connection handle related to the response;
 Attr_Data_Length: the size of each attribute data;
 Data_Length: length of Attribute_Data_List in octets;
 Att_Data_List: Attribute Data List as defined in Bluetooth Core
 specifications. A sequence of attribute handle, end group handle,
 attribute value tuples: [2 octets for Attribute Handle, 2
 octets End Group Handle, (Attribute_Data_Length - 4 octets) for
 Attribute Value].
*/
/* Add user code for decoding the Att_Data_List field and getting
the services attribute handle, end group handle and service uuid
*/
}/* aci_att_read_by_group_type_resp_event() */
```

In the context of the sensor profile demo, the GAP central application should get three read by group type response events (through related `aci_att_read_by_group_type_resp_event()` event callback), with the following callback parameters values.

First read by group type response event callback parameters:

```
Connection_Handle: 0x0801 (connection handle);
Attr_Data_Length: 0x06 (length of each discovered service data: service
handle, end group handle,service uuid);
Data_Length: 0x0C (length of Attribute_Data_List in octets
Att_Data_List: 0x0C bytes as follows:
```

Table 44. First read by group type response event callback parameters

Attribute handle	End group handle	Service UUID	Notes
0x0001	0x0004	0x1801	Attribute profile service (GATT_Init() addsit). Standard 16-bit service UUID
0x0005	0x000B	0x1800	GAP profile service (GAP_Init() adds it). Standard 16-bit service UUID.

Second read by group type response event callback parameters:

```
Conn_Handle: 0x0801 (connection handle);
Attr_Data_Length: 0x14 (length of each discovered service data:
service handle, end group handle,service uuid);
Data_Length: 0x14 (length of Attribute_Data_List in octets);
Att_Data_List: 0x14 bytes as follows:
```

Table 45. Second read by group type response event callback parameters

Attribute handle	End group handle	Service UUID	Notes
0x000C	0x0012	0x02366E80CF3A11E19AB4 0002A5D5C51B	Acceleration service 128-bit service proprietary UUID

Third read by group type response event callback parameters:

```
Connection_Handle: 0x0801 (connection handle);
Attr_Data_Length: 0x14 (length of each discovered service data:
service handle, end group handle, service uuid);
Data_Length: 0x14 (length of Attribute_Data_List in octets);
Att_Data_List: 0x14 bytes as follows:
```

Table 46. Third read by group type response event callback parameters

Attribute handle	End group handle	Service UUID	Notes
0x0013	0x0019	0x42821A40E47711E282D00 002A5D5C51B	Environmental service 128-bit service proprietary UUID

In the context of the sensor profile demo, when the discovery all primary service procedure completes, the `aci_gatt_proc_complete_event()` event callback is called on GAP central application, with the following parameters

```
Conn_Handle: 0x0801 (connection handle;
Error_Code: 0x00
```

4.6.1 Characteristic discovery procedures and related GATT events

A list of the characteristic discovery APIs with associated description as follows:

Table 47. **Characteristics discovery procedures APIs**

Discovery service API	Description
`aci_gatt_disc_all_char_of_service ()`	This API starts the GATT procedure to discover all the characteristics of a given service
`aci_gatt_disc_char_by_uuid ()`	This API starts the GATT the procedure to discover all the characteristics specified by a UUID
`aci_gatt_disc_all_char_desc ()`	This API starts the procedure to discover all characteristic descriptors on the GATT server

In the context of the BLE sensor profile demo, follow a simple pseudocode illustrating how a GAP central application can discover all the characteristics of the acceleration service (refer to Table 45. Second read by group type response event callback parameters):

```
uint16_t service_handle= 0x000C;
uint16_t end_group_handle = 0x0012;
```

```
/*GAP Central starts a discovery all the characteristics of a service
procedure: conn_handle is the connection handle returned on
hci_le_advertising_report_event()eventcallback */
if(aci_gatt_disc_all_char_of_service(conn_handle,
                                    service_handle,/* Servicehandle */
                                    end_group_handle/* End group handle
                                                   */
                                    );) != BLE_STATUS_SUCCESS)
{
   PRINTF("Failure.\n");
}
```

The responses of the procedure are given through the`aci_att_read_by_type_resp_event()` event callback. The end of the procedure is indicated by `aci_gatt_proc_complete_event()` event callback call.

```
/* This event is generated in response to aci_att_read_by_type_req(). Refer to
aci_gatt_disc_all_char() API */
```

```
void aci_att_read_by_type_resp_event(uint16_t Connection_Handle ,
                                    uint8_t Handle_Value_Pair_Length,
                                    uint8_t Data_Length,
                                    uint8_t Handle_Value_Pair_Data[])
{

/*
     Connection_Handle: connection handle related to the response;
     Handle_Value_Pair_Length: size of each attribute handle-value
                              Pair;
     Data_Length: length of Handle_Value_Pair_Data in octets.
     Handle_Value_Pair_Data: Attribute Data List as defined in
     Bluetooth Core specifications. A sequence of handle-value pairs: [2
     octets for Attribute Handle, (Handle_Value_Pair_Length - 2 octets)
     for Attribute Value].
*/
/* Add user code for decoding the Handle_Value_Pair_Data field and
     get the characteristic handle, properties,characteristic value handle,
     characteristic UUID*/
 */

}/* aci_att_read_by_type_resp_event() */
```

In the context of the BLE sensor profile demo, the GAP central application should get two read type response events (through related `aci_att_read_by_type_resp_event()` event callback), with the following callback parameter values.

First read by type response event callback parameters:

```
conn_handle : 0x0801 (connection handle);
Handle_Value_Pair_Length: 0x15 length of each discovered
characteristic data: characteristic handle, properties,
characteristic value handle, characteristic UUID;
Data_Length: 0x16(length of the event data);
Handle_Value_Pair_Data: 0x15 bytes as follows:
```

Table 48. **First read by type response event callback parameters**

Characteristic handle	Characteristic properties	Characteristic value handle	Characteristic UUID	Note
0x000D	0x10 (notify)	0x000E	0xE23E78A0CF4A11E18FFC0002A5D5C51B	Free fall characteristic 128-bit characteristic proprietary UUID

Second read by type response event callback parameters:

```
conn_handle : 0x0801 (connection handle);
Handle_Value_Pair_Length: 0x15 length of each discovered
characteristic data: characteristic handle, properties,
characteristic value handle, characteristic UUID;
Data_Length: 0x16(length of the event data);
Handle_Value_Pair_Data: 0x15 bytes as follows:
```

Table 49. **Second read by type response event callback parameters**

Characteristic handle	Characteristic properties	Characteristic value handle	Characteristic UUID	Note
0x0010	0x12 (notify and read)	0x0011	0x340A1B80CF4B11E1AC360002A5D5C51B	Acceleration characteristic 128- bit characteristic proprietary UUID

In the context of the sensor profile demo, when the discovery all primary service procedure completes, the `aci_gatt_proc_complete_event()` event callback is called on GAP central application, with the following parameters:

```
Connection_Handle: 0x0801 (connection handle);
Error_Code: 0x00.
```

Similar steps can be followed in order to discover all the characteristics of the environment service (Table 42. BLE sensor profile demo services and characteristic handle).

4.7 Characteristic notification/indications, write, read

This section describes the main functions to get access to BLE device characteristics.

Table 50. **Characteristic update, read, write APIs**

Discovery service API	Description	Where
aci_gatt_update_char_value_ext()	If notifications (or indications) are enabled on the characteristic, this API sends a notification (or indication) to the client.	GATT server
aci_gatt_read_char_value()	It starts the procedure to read the attribute value.	GATT client

Discovery service API	Description	Where
aci_gatt_write_char_value()	It starts the procedure to write the attribute value (when the procedure is completed, a GATT procedure complete event is generated).	GATT client
aci_gatt_write_without_resp()	It starts the procedure to write a characteristic value without waiting for any response from the server.	GATT client
aci_gatt_write_char_desc()	It starts the procedure to write a characteristic descriptor.	GATT client
aci_gatt_confirm_indication()	It confirms an indication. This command has to be sent when the application receives a characteristic indication.	GATT client

In the context of the P2PServer demo, follow a part of code the GAP Central application should use in order to configure the Button characteristics client descriptor configuration for notification:

```
/* Enable the Button characteristic client descriptor configuration for notification */
aci_gatt_write_char_desc(aP2PClientContext[index].connHandle,
aP2PClientContext[index].P2PNotificationDescHandle,
2,
uint8_t *)&enable);
```

Once the characteristic notification has been enabled from the GAP central, the GAP peripheral can notify a new value for the free fall and acceleration characteristics as follows:

```
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 USHED \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;
}
tBleStatus P2PS_STM_App_Update_Char(uint16_t UUID, uint8_t *pPayload)
{
tBleStatus result = BLE_STATUS_INVALID_PARAMS;
switch(UUID)
{
case P2P_NOTIFY_CHAR_UUID:
result = aci_gatt_update_char_value(aPeerToPeerContext.PeerToPeerSvcHdle,
aPeerToPeerContext.P2PNotifyServerToClientCharHdle,
0, /* charValOffset */
2, /* charValueLen */
(uint8_t *) pPayload);
break;
default:
break;
}
return result;
}/* end P2PS_STM_Init() */
```

On GAP Central, Event_Handler (EVT_VENDOR as main event), the EVT_BLUE_GATT_NOTIFICATION is raised on reception of the characteristic notification (Button) from the GAP Peripheral device.

```
static SVCCTL_EvtAckStatus_t Event_Handler(void *Event)
{
SVCCTL_EvtAckStatus_t return_value;
hci_event_pckt *event_pckt;
evt_blue_aci *blue_evt;
P2P_Client_App_Notification_evt_t Notification;
return_value = SVCCTL_EvtNotAck;
event_pckt = (hci_event_pckt *)(((hci_uart_pckt*)Event)->data);
switch(event_pckt->evt) {
case EVT_VENDOR:
{
blue_evt = (evt_blue_aci*)event_pckt->data;
switch(blue_evt->ecode) {
….
case EVT_BLUE_GATT_NOTIFICATION:
{
aci_gatt_notification_event_rp0 *pr = (void*)blue_evt->data;
uint8_t index;
index = 0;
while((index < BLE_CFG_CLT_MAX_NBR_CB) &&
(aP2PClientContext[index].connHandle != pr->Connection_Handle))
index++;
if(index < BLE_CFG_CLT_MAX_NBR_CB) {
if ( (pr->Attribute_Handle == aP2PClientContext[index].P2PNotificationCharHdle) &&
(pr->Attribute_Value_Length == (2)) )
{
Notification.P2P_Client_Evt_Opcode = P2P_NOTIFICATION_INFO_RECEIVED_EVT;
Notification.DataTransfered.Length = pr->Attribute_Value_Length;
Notification.DataTransfered.pPayload = &pr->Attribute_Value[0];
Gatt_Notification(&Notification);
/* INFORM APPLICATION BUTTON IS PUSHED BY END DEVICE */
}
}
}
break;/* end EVT_BLUE_GATT_NOTIFICATION */
….
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 APPLICATION CLIENT : NOTIFICATION RECEIVED - LED OFF \n\r");
APP_DBG_MSG(" \n\r");
} else {
APP_DBG_MSG(" -- P2P APPLICATION CLIENT : NOTIFICATION RECEIVED - LED ON\n\r");
APP_DBG_MSG(" \n\r");
BSP_LED_On(LED_BLUE);
}
break;
}
default : break;
}
….
}
```

4.7.1 Getting access to BLE device long characteristics.

This section describes the main functions for getting access to BLE device long characteristics.

Table 51. **Characteristic update, read, write APIs for long Value**

Characteristic handling API	Description	API call side	Events to be used on client side
Aci_gatt_read_long_char_value()	Reads a long characteristic value.	GATT client	ACI_GATT_READ_EXT_EVENT (mask = 0x00100000)
Aci_gatt_write_long_char_value()	Writes a long characteristic value.	GATT client	ACI_ATT_EXEC_WRITE_RESP_EVENT (mask = 0x00001000) ACI_ATT_PREPARE_WRITE_RESP_EVENT (mask = 0x00000800)
Aci_gatt_update_char_value_ext()	Version of aci_gatt_update_char_value to support update of long attribute up to 512 bytes and indicate selectively the generation of indication/notification.	GATT server	ACI_GATT_NOTIFICATION_EXT_EVENT (mask = 0x00400000) or ACI_GATT_INDICATION_EXT_EVENT (mask = 0x00200000)
Aci_gatt_read_handle_value()	Reads the value of the attribute handle specified from the local GATT database.	GATT server	-

1. Characteristics are long when char_length > ATT_MTU – 4
2. Limitation due to the stack interface of events: event parameters length is an 8-bit value.

Read long distant data (client side)

To avoid limitation 2, new events have been added: ACI_GATT_READ_EXT_EVENT
(to be enabled with the following mask: 0x00100000 using aci_gatt_set_event_mask command)
It will replace 3 events:

```
ACI_ATT_READ_RESP_EVENT (1)
ACI_ATT_READ_BLOB_RESP_EVENT (2)
ACI_ATT_READ_MULTIPLE_RESP_EVENT (3)
```

Generated in response to:

```
Aci_gatt_read_char_value (1)
Aci_gatt_read_long_char_value (2)
Aci_gatt_read_multiple_char_value (3)
```

(condition ATT_MTU > sum of the multiple characteristics total length

Write long distant data (client side)

```
Aci_gatt_write_long_char_value()
```

The length of the data to be written is limited to 245 (with ATT_MTU = 251)

Read long local data (server side)

```
Aci_gatt_read_handle_value()
```

This command needs to be called several times.

Write long local data (server side)

```
ACI_GATT_NOTIFICATION_EXT_EVENT
```

(to be enabled with the following mask : 0x00400000 using aci_gatt_set_event_mask command)

In response to:

```
Aci_gatt_update_char_value_ext
```

command

How to use aci_gatt_update_char_value_ext:

When

```
ATT_MTU > (BLE_EVT_MAX_PARAM_LENGTH – 4) i.e ATT_MTU > 251
```

, two commands are necessary.

First command:

```
Aci_gatt_update_char_value_ext (conn_handle, Service_handle, TxCharHandle,
Update_Type = 0x00,
Total_length,
Value_offset,
Param_length,
&payload)
```

Second command

```
Aci_gatt_update_char_value_ext (conn_handle, Service_handle, TxCharHandle,
Update_Type = 0x01,
Total_length,
Value_offset = Param_length,
param_length2,
(&payload) + param_length)
```

After second command, a notification of total length is sent on the air and is received through ACI_GATT_NOTIFICATION_EXT_EVENT events.

The data can be re-assembled depending on the offset parameter of ACI_GATT_NOTIFICATION_EXT_EVENT event. Bit 15 is used as flag: when set to 1it indicates that more data are to come (fragmented event in case of long attribute data)

Idem for: ACI_GATT_INDICATION_EXT_EVENT (to be enabled with the following mask : 0x00200000 using aci_gatt_set_event_mask command)

In response to: Aci_gatt_update_char_value_ext() command.

In this case Update_Type = 0x00 for the first command, and Update_Type = 0x02 for the second command.

If we take an example of long data transfer:

Once the characteristics notification has been enabled from the GAP Central, the GAP peripheral can notify a new value:

```
static void SendData( void )
{
tBleStatus status = BLE_STATUS_INVALID_PARAMS;
uint8_t crc_result;
if( (DataTransferServerContext.ButtonTransferReq != DTS_APP_TRANSFER_REQ_OFF)
&& (DataTransferServerContext.NotificationTransferReq != DTS_APP_TRANSFER_REQ_OFF)
&& (DataTransferServerContext.DtFlowStatus != DTS_APP_FLOW_OFF) )
{
/*Data Packet to send to remote*/
Notification_Data_Buffer[0] += 1;
/* compute CRC */
crc_result = APP_BLE_ComputeCRC8((uint8_t*) Notification_Data_Buffer,
(DATA_NOTIFICATION_MAX_PACKET_SIZE - 1));
Notification_Data_Buffer[DATA_NOTIFICATION_MAX_PACKET_SIZE - 1] = crc_result;
DataTransferServerContext.TxData.pPayload = Notification_Data_Buffer;
//DataTransferServerContext.TxData.Length = DATA_NOTIFICATION_MAX_PACKET_SIZE; /*
DATA_NOTIFICATION_MAX_PACKET_SIZE */
DataTransferServerContext.TxData.Length = Att_Mtu_Exchanged-10;
status = DTS_STM_UpdateChar(DATA_TRANSFER_TX_CHAR_UUID, (uint8_t *)
&DataTransferServerContext.TxData);
if (status == BLE_STATUS_INSUFFICIENT_RESOURCES)
{
DataTransferServerContext.DtFlowStatus = DTS_APP_FLOW_OFF;
(Notification_Data_Buffer[0])-=1;
}
else
{
UTIL_SEQ_SetTask(1 << CFG_TASK_DATA_TRANSFER_UPDATE_ID, CFG_SCH_PRIO_0);
}
}
return;
}
tBleStatus DTS_STM_UpdateChar( uint16_t UUID , uint8_t *pPayload )
{
tBleStatus result = BLE_STATUS_INVALID_PARAMS;
switch (UUID)
{
case DATA_TRANSFER_TX_CHAR_UUID:
result = TX_Update_Char((DTS_STM_Payload_t*) pPayload);
break;
default:
break;
}
return result;
}/* end DTS_STM_UpdateChar() */
static tBleStatus TX_Update_Char( DTS_STM_Payload_t *pDataValue )
{
tBleStatus ret;
/**
* Notification Data Transfer Packet
*/
/* Total length corresponds to total length of data that will be sent through notification
Value offset corresponds to the offset of the value to modify Param length corresponds to
the length of the value to be modify at the offset defined previously */
```

On GAP Client, DTC_Event_Handler (EVT_VENDOR as main event), the EVT_BLUE_GATT_NOTIFICATION_EXT is raised on reception of the characteristic notification (Button) from the GAP Peripheral device.

```
static SVCCTL_EvtAckStatus_t DTC_Event_Handler(void *Event)
{
SVCCTL_EvtAckStatus_t return_value;
hci_event_pckt *event_pckt;
evt_blue_aci *blue_evt;
P2P_Client_App_Notification_evt_t Notification;
return_value = SVCCTL_EvtNotAck;
event_pckt = (hci_event_pckt *)(((hci_uart_pckt*)Event)->data);
switch(event_pckt->evt)
{
case EVT_VENDOR:
{
blue_evt = (evt_blue_aci*)event_pckt->data;
switch(blue_evt->ecode)
{
….
case EVT_BLUE_GATT_NOTIFICATION_EXT:
{
aci_gatt_notification_event_rp0 *pr = (void*)blue_evt->data;nnnn
uint8_t index;
index = 0;
while((index < BLE_CFG_CLT_MAX_NBR_CB) &&
(aP2PClientContext[index].connHandle != pr->Connection_Handle))
index++;
if(index < BLE_CFG_CLT_MAX_NBR_CB)
{
if ( (pr->Attribute_Handle == aP2PClientContext[index].P2PNotificationCharHdle) &&
(pr->Attribute_Value_Length == (2)) )
{
Notification.P2P_Client_Evt_Opcode = P2P_NOTIFICATION_INFO_RECEIVED_EVT;
Notification.DataTransfered.Length = pr->Attribute_Value_Length;
Notification.DataTransfered.pPayload = &pr->Attribute_Value[0];
Gatt_Notification(&Notification);
/* INFORM APPLICATION BUTTON IS PUSHED BY END DEVICE */
}
}
}
break;/* end EVT_BLUE_GATT_NOTIFICATION */
```

4.8 End to end RX flow control using GATT

It is possible to benefit from an optimized RX flow control when using GATT to receive data from a peer.

Typically, the peer device uses several times the GATT write procedure to send the data by packets to a local device GATT characteristic. The user application of the local device then receives the packets through successive GATT events (ACI_GATT_ATTRIBUTE_MODIFIED_EVENT).

To get an RX flow control the user application needs to set the AUTHOR_WRITE flag when creating the characteristic using the ACI_GATT_ADD_CHAR primitive. The user application is then informed of each peer write tentative before it is executed by means of a dedicated event (ACI_GATT_WRITE_PERMIT_REQ_EVENT). The user application just needs to answer to that event with the ACI_GATT_WRITE_RESP primitive (Write_status = 0). If the user application takes time to answer to this event (e.g. it is still processing the previous data packet), this will have the effect of blocking the local GATT and then block the peer when the local internal RX ACL data FIFO is full (the size of this FIFO depending on the BLE stack configuration).

4.9 Basic/typical error condition description

On the STM32WB BLE stack APIs framework, the `tBleStatus` type is defined in order to return the STM32WB stack error conditions. The error codes are defined within the header file “ble_status.h”.

When a stack API is called, it is recommended to get the API return status and to monitor it in order to track potential error conditions.

BLE_STATUS_SUCCESS (0x00) is returned when the API is successfully executed. For a list of error conditions associated to each ACI API refer to the STM32WB Bluetooth LE stack APIs and event documentation, in Section 6 Reference documents

4.10 BLE simultaneously master, slave scenario

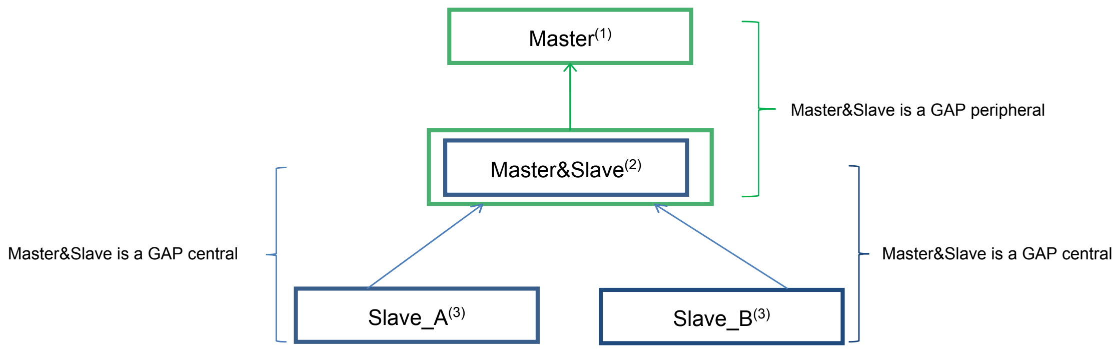
The STM32WB BLE stack supports multiple roles simultaneously. This allows the same device to act as master on one or more connections (up to eight connections are supported), and to act as a slave on another connection.

The following pseudo code describes how a BLE stack device can be initialized to support central and peripheral roles simultaneously:

```
uint8_t role= GAP_PERIPHERAL_ROLE | GAP_CENTRAL_ROLE;
ret= aci_gap_init(role, 0, 0x07, &service_handle,
&dev_name_char_handle, &appearance_char_handle);
```

A simultaneous master and slave test scenario can be targeted as follows:

Figure 19. BLE simultaneous master and slave scenario


Master(1)
Master&Slave is a GAP peripheral
Master&Slave(2)
Master&Slave is a GAP central
Master&Slave is a GAP central
Slave_A(3)
Slave_B(3)

1. (1) BLE GAP central
2. (2) BLE GAP central & peripheral
3. (3) BLE GAP peripheral

Step 1. One BLE device (called Master&Slave) is configured as central and peripheral by setting role as `GAP_PERIPHERAL_ROLE |GAP_CENTRAL_ROLE` on `GAP_Init()` API . Let's also assume that this device also defines a service with a characteristic.

Step 2. Two BLE devices (called Slave_A, Slave_B) are configured as peripheral by setting role as `GAP_PERIPHERAL_ROLE` on `GAP_Init()` API. Both Slave_A and Slave_B define the same service and characteristic as Master&Slave device.

Step 3. One BLE device (called Master) is configured as central by setting role as `GAP_CENTRAL_ROLE` on `GAP_Init()API`.

Step 4. Both Slave_A and Slave_B devices enter discovery mode as follows:

```
ret =aci_gap_set_discoverable(Advertising_Type=0x00,
                              Advertising_Interval_Min=0x20,
                              Advertising_Interval_Max=0x100,
                              Own_Address_Type= 0x0;
                              Advertising_Filter_Policy= 0x00;
                              Local_Name_Length=0x05,
                              Local_Name=[0x08,0x74,0x65,0x73,0x74],
                              Service_Uuid_length = 0;
                              Service_Uuid_length = NULL;
                              Slave_Conn_Interval_Min = 0x0006,
                              Slave_Conn_Interval_Max = 0x0008);
```

Step 5. Master&Slave device performs a discovery procedure in order to discover the peripheral devices Slave_A and Slave_B:

```
ret = aci_gap_start_gen_disc_proc (LE_Scan_Interval=0x10,
                                   LE_Scan_Window=0x10,
                                   Own_Address_Type = 0x0,
                                   Filter_Duplicates = 0x0);
```

The two devices are discovered through the advertising report events notified with the `hci_le_advertising_report_event()` event callback.

Step 6. Once the two devices are discovered, Master&Slave device starts two connection procedures (as central) to connect, respectively, to Slave_A and Slave_B devices:

```
/* Connect to Slave_A:Slave_Aaddress type and address have been found
   during the discovery procedure through the Advertising Report events.
*/
ret= aci_gap_create_connection(LE_Scan_Interval=0x0010,
                               LE_Scan_Window=0x0010
                               Peer_Address_Type= "Slave_A address type"
                               Peer_Address= "Slave_A address,
                               Own_Address_Type = 0x0;
                               Conn_Interval_Min=0x6c,
                               Conn_Interval_Max=0x6c,
                               Conn_Latency=0x00,
                               Supervision_Timeout=0xc80,
                               Minimum_CE_Length=0x000c,
                               Maximum_CE_Length=0x000c);
```

```
/* Connect to Slave_B:Slave_Baddress type and address have been found
   during the discovery procedure through the Advertising Report events.
*/
ret= aci_gap_create_connection(LE_Scan_Interval=0x0010,
                               LE_Scan_Window=0x0010,
                               Peer_Address_Type= "Slave_B address type",
                               Peer_Address= "Slave_B address",
                               Own_Address_Type = 0x0;
                               Conn_Interval_Min=0x6c,
                               Conn_Interval_Max=0x6c,
                               Conn_Latency=0x00,
                               Supervision_Timeout=0xc80,
                               Minimum_CE_Length=0x000c,
                               Maximum_CE_Length=0x000c);
```

Step 7. Once connected, Master&Slave device enables the characteristics notification, on both of them, using the `aci_gatt_write_char_desc()` API. Slave_A and Slave_B devices start the characteristic notification by using the `aci_gatt_upd_char_val()` API.

Step 8. At this stage, Master&Slave device enters discovery mode (acting as peripheral):

```
/*Put Master&Slave device in Discoverable Mode with Name = 'Test' =
[0x08,0x74,0x65,0x73,0x74*/
ret =aci_gap_set_discoverable(Advertising_Type=0x00,
                             Advertising_Interval_Min=0x20,
                             Advertising_Interval_Max=0x100,
                             Own_Address_Type= 0x0;
                             Advertising_Filter_Policy= 0x00;
                             Local_Name_Length=0x05,
                             Local_Name=[0x08,0x74,0x65,0x73,0x74],
                             Service_Uuid_length = 0;
                             Service_Uuid_List = NULL;
                             Slave_Conn_Interval_Min = 0x0006,
                             Slave_Conn_Interval_Max = 0x0008);
```

Since Master&Slave device also acts as a central device, it receives the notification event related to the characteristic values notified from, respectively, Slave_A and Slave_B devices.

Step 9. Once Master&Slave device enters discovery mode, it also waits for the connection request coming from the other BLE device (called Master) configured as GAP central. Master device starts discovery procedure to discover the Master&Slave device:

```
    ret = aci_gap_start_gen_disc_proc(LE_Scan_Interval=0x10,
                                      LE_Scan_Window=0x10,
                                      Own_Address_Type = 0x0,
                                      Filter_Duplicates = 0x0);
```

Step 10. Once the Master&Slave device is discovered, Master device starts a connection procedure to connect to it:

```
/* Master device connects to Master&Slave device: Master&Slave
   address type and address have been found during the discovery
   procedure through the Advertising Report events */
   ret= aci_gap_create_connection(LE_Scan_Interval=0x0010,
                                  LE_Scan_Window=0x0010,
                            Peer_Address_Type= "Master&Slave address type",
                            Peer_Address= " Master&Slave address",
                                  Own_Address_Type = 0x0;
                                  Conn_Interval_Min=0x6c,
                                  Conn_Interval_Max=0x6c,
                                  Conn_Latency=0x00,
                                  Supervision_Timeout=0xc80,
                                  Minimum _CE_Lenght=0x000c
                                  Maximum_CE_Length=0x000c);
```

Master&Slave device is discovered through the advertising report events notified with the `hci_le_advertising_report_event()` event callback.

Step 11. Once connected, Master device enables the characteristic notification on Master&Slave device using the `aci_gatt_write_char_desc()` API.

Step 12. At this stage, Master&Slave device receives the characteristic notifications from both Slave_A, Slave_B devices, since it is a GAP central and, as GAP peripheral, it is also able to notify these characteristic values to the Master device.

4.11 Bluetooth low energy privacy 1.2

BLE stack v2.x supports the Bluetooth low energy privacy 1.2.

Privacy feature reduces the ability to track a specific BLE by modifying the related BLE address frequently. The frequently modified address is called the private address and the trusted devices are able to resolve it.

In order to use this feature, the devices involved in the communication need to be previously paired: the private address is created using the devices IRK exchanged during the previous pairing/bonding procedure.

There are two variants of the privacy feature:

1. Host-based privacy private addresses are resolved and generated by the host
2. Controller-based privacy private addresses are resolved and generated by the controller without involving the host after the Host provides the controller device identity information.

When controller privacy is supported, device filtering is possible since address resolution is performed in the controller (the peer's device identity address can be resolved prior to checking whether it is in the white list).

4.11.1 Controller-based privacy and the device filtering scenario

On STM32WB, with `aci_gap_init()` API supports the following options for the `privacy_enabled` parameter:

- 0x00: privacy disabled
- 0x01: host privacy enabled
- 0x02: controller privacy enabled

When a slave device wants to resolve a resolvable private address and be able to filter on private addresses for reconnection with bonded and trusted devices, it must perform the following steps:

1. Enable privacy controller on `aci_gap_init()`: use 0x02 as `privacy_enabled`parameter.

2. Connect, pair and bond with the candidate trusted device using one of the allowed security methods: the private address is created using the devices IRK.

3. Call the `aci_gap_configure_whitelist()` API to add the address of bonded device into the BLE device controller's whitelist.

4. Get the bonded device identity address and type using the `aci_gap_get_bonded_devices()` API.

5. Add the bonded device identity address and type to the list of address translations used to resolve resolvable private addresses in the controller, by using the `aci_gap_add_devices_to_resolving_list()` API.

6. The device enters the undirected connectable mode by calling the `aci_gap_set_undirected_connectable()` API with `Own_Address_Type = 0x02` (resolvable private address) and `Adv_Filter_Policy` = 0x03 (allow scan request from whitelist only, allow connect request from whitelist only).

7. When a bonded master device performs a connection procedure for reconnection to the slave device, the slave device is able to resolve and filter the master address and connect with it.

4.11.2 Resolving addresses

After a reconnection with a bonded device, it is not strictly necessary to resolve the address of the peer device to encrypt the link. In fact, STM32WB stack automatically finds the correct LTK to encrypt the link.

However, there are some cases where the peer's address must be resolved. When a resolvable privacy address is received by the device, it can be resolved by the host or by the controller (i.e. link layer).

Host-based privacy

If controller privacy is not enabled, a resolvable private address can be resolved by using `aci_gap_resolve_private_addr()`. The address is resolved if the corresponding IRK can be found among the stored IRKs of the bonded devices. A resolvable private address may be received when STM32WB are in scanning, through `hci_le_advertising_report_event()`, or when a connection is established, through `hci_le_connection_complete_event()`.

Controller-based privacy

If the resolution of addresses is enabled at link layer, a resolving list is used when a resolvable private address is received. To add a bonded device to the resolving list, the `aci_gap_add_devices_to_resolving_list()` has to be called. This function searches for the corresponding IRK and adds it to the resolving list.

When privacy is enabled, if a device has been added to the resolving list, its address is automatically resolved by the link layer and reported to the application without the need to explicitly call any other function. After a connection with a device, the `hci_le_enhanced_connection_complete_event()` is returned. This event reports the identity address of the device, if it has been successfully resolved (if the `hci_le_enhanced_connection_complete_event()` is masked, only the `hci_le_connection_complete_event()` is returned).

When scanning, the `hci_le_advertising_report_event()` contains the identity address of the device in advertising if that device uses a resolvable private address and its address is correctly resolved. In that case, the reported address type is 0x02 or 0x03. If no IRK can be found that can resolve the address, the resolvable private address is reported. If the advertiser uses directed advertisement, the resolved private address is reported through the `hci_le_advertising_report_event()` or through the `hci_le_direct_advertising_report_event()` if it has been unmasked and the scanner filer policy is set to 0x02 or 0x03.

4.12 ATT_MTU and exchange MTU APIs, events

ATT_MTU is defined as the maximum size of any packet sent between a client and a server:

- default ATT_MTU value: 23 bytes

This determines the current maximum attribute value size when the user performs characteristic operations (notification/write max. size is ATT_MTU-3).

The client and server may exchange the maximum size of a packet that can be received using the exchange MTU request and response messages. Both devices use the minimum of these exchanged values for all further communications:

```
tBleStatus aci_gatt_exchange_config(uint16_t Connection_Handle);
```

In response to an exchange MTU request, the `aci_att_exchange_mtu_resp_event()` callback is triggered on both devices:

```
void aci_att_exchange_mtu_resp_event(uint16_t Connection_Handle, uint16_t
                                     Server_RX_MTU);
```

`Server_RX_MTU` specifies the ATT_MTU value agreed between the server and client.

4.13 LE data packet length extension APIs and events

On BLE specification v4.2, packet data unit (PDU) size has been increased from 27 to 251 bytes. This allows data rate to be increased by reducing the overhead (header, MIC) needed on a packet. As a consequence, it is possible to achieve: faster OTA FW upgrade operations, more efficiency due to less overhead.

The STM32WB stack supports LE data packet length extension features and related APIs, events:

- HCI LE APIs (API prototypes)
  - `hci_le_set_data_length()`
  - `hci_le_read_suggested_default_data_length()`
  - `hci_le_write_suggested_default_data_length()`
  - `hci_le_read_maximum_data_length()`
- HCI LE events (events callbacks prototypes)
  - `hci_le_data_length_change_event()`

`hci_le_set_data_length()` API allows the user's application to suggest maximum transmission packet size (`TxOctets`) and maximum packet (`TxTime`) transmission time to be used for a given connection:

```
tBleStatus hci_le_set_data_length(uint16_t Connection_Handle,
                                  uint16_t TxOctets,
                                  uint16_t TxTime);
```

The supported TxOctets value is in the range [27-251] and the TxTime is provided as follows: (TxOctets +14)*8.

Once `hci_le_set_data_length()` API is performed on a STM32WB device after the device connection, if the connected peer device supports LE data packet length extension feature, the following event is raised on both devices:

```
hci_le_data_length_change_event(uint16_t Connection_Handle,
                                uint16_t MaxTxOctets,
                                uint16_t MaxTxTime,
                                uint16_t MaxRxOctets,
                                uint16_t MaxRxTime)
```

This event notifies the host of a change to either the maximum link layer payload length or the maximum time of link layer data channel PDUs in either direction (TX and RX). The values reported (`MaxTxOctets, MaxTxTime, MaxRxOctets, MaxRxTime`) are the maximum values that are actually used on the connection following the change.

4.14 STM32WB LE 2M PHY

Introduced in the Bluetooth Core Specification Version 5.0, LE 2M PHY allows the physical layer to operate at higher data rate up to 2Mb/s. LE 2M PHY double data rate versus standard LE 1M PHY, this reduce power consumption using same transmit power. The transmit distance will be lower relative to LE 1M PHY, due to the increased symbol rate. Within STM32WB stack, both LE 1M PHY and LE 2M PHY are supported, and it is up to Application to select default PHY requirement. Application can initiate change PHY parameters at any point of time and as often as required, with different PHY parameters on each connection channel selected (via connection handle). And since STM32WB handles asymmetric connection, Application can also use different PHYs in each direction of connection RX and TX (via connection handle). PHY negotiation is transparent at Application side and depends on remote feature capabilities. STM32WB stack supports followings commands:

- HCI_LE_SET_DEFAULT_PHY: to allow the host to specify its preferred for TX & RX PHY parameters.
- HCI_LE_SET_PHY: to allow the host to set PHY preferences for current connection (identified by the connection handle) for TX & RX PHY parameters.
- HCI_LE_READ_PHY: to hallow the host to read TX & RX PHY parameters on current connection(identify by connection handle).

4.15 STM32WB formula for converting RSSI raw value in dBm

This section explain how the application could read remote device RSSI (receiving signal strength indicator) values reported by STM32WB radio measurement. In order to convert these received level values in dBm, the user must follow the following steps:

- READ RSSI SPI programmation:

```
globalParameters.rssiLevel[0] = 0x84; /* Read command of 3 data bytes */
globalParameters.rssiLevel[1] = SPI_RSSI0_DIG_OUT_ADD; /* 3 bytes reading: 2 for
RSSImeasurement + 1 for AGC (SPI_AGC_DIG_OUT_ADD) */
globalParameters.rssiLevel[5] = 0; /* Last byte shall be 0 to force the BLE core to
stop reading */
```

- RSSI read intoIRQ:

```
if (BLUE_CTRL->RADIO_CONFIG == ((uint32_t)(&(globalParameters.rssiLevel[5])+2)&
0xffff)) /* check BLE Core has properly incremented the pointer */
{
globalParameters.rssiValid = globalParameters.rssiLevel[2] +
globalParameters.rssiLevel[3] <<8) + (globalParameters.rssiLevel[4] <<16);
globalParameters.current_action_packet->rssi =  IPBLE_LLD_Read_RSSI_dBm() ;
}*/
```

- RSSI Code Conversion in dBm:

```
int32_t IPBLE_LLD_Read_RSSI_dBm(void)
{
  int i = 0 ; int rsi_dbm;
  while(i < 100 && (BLUE_CTRL->RADIO_CONFIG & 0x10000) != 0)
{
    rsi_dbm = i++;
}
int rssi_int16 = globalParameters.rssiValid & 0xFFFF ; /* First 2 bytes contain Rssi
measured on the received signal */
  int reg_agc = (globalParameters.rssiValid >> 16) & 0xFF ; /* Third byte is the AGC
value used for the RX */
if(rssi_int16 == 0 || reg_agc > 0xb)
    rsi_dbm = 127 ;
}
else
{
rsi_dbm = reg_agc * 6 -127 ;
while(rssi_int16 > 30)
    {
rsi_dbm = rsi_dbm + 6 ;
rssi_int16 = rssi_int16 >> 1 ;
    }
    rsi_dbm = rsi_dbm + ((417*rssi_int16 + 18080)>>10) ;
}
return rsi_dbm ;
}
```

5 BLE multiple connection timing strategy

This section provides an overview of the connection timing management strategy of the STM32WB stack when multiple master and slave connections are active.

5.1 Basic concepts about Bluetooth low energy timing

This section describes the basic concepts related to the Bluetooth low energy timing management related to the advertising, scanning and connection operations.

5.1.1 Advertising timing

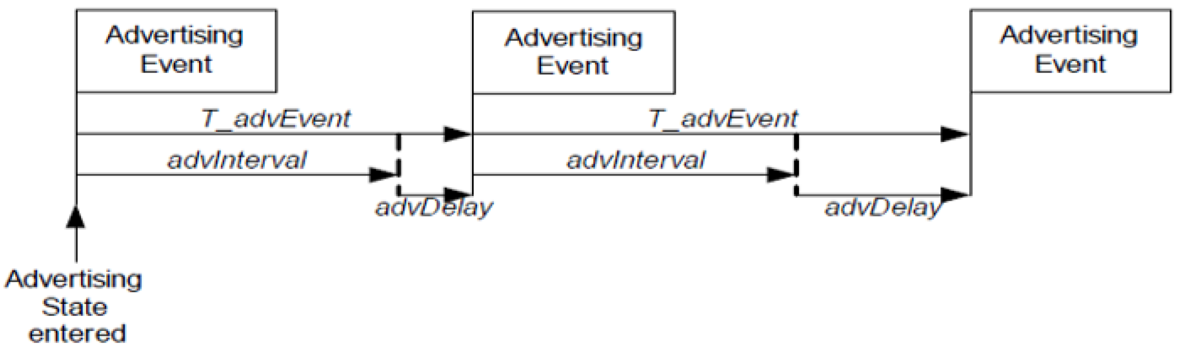
The timing of the advertising state is characterized by 3 timing parameters, linked by this formula:

T_advEvent = advInterval + advDelay

where:

- T_advEvent: time between the start of two consecutive advertising events; if the advertising event type is either a scannable undirected event type or a non-connectable undirected type, the advInterval shall not be less than 100 ms; if the advertising event type is a connectable undirected event type or connectable directed event type used in a low duty cycle mode, the advInterval can be 20 ms or greater.
- advDelay: pseudo-random value with a range of 0 ms to 10 ms generated by the link layer for each advertising event.

Figure 20. **Advertising timings**



5.1.2 Scanning timing

The timing of the scanning state is characterized by 2 timing parameters:

- scanInterval: defined as the interval between the start of two consecutive scan windows
- scanWindow: time during which link layer listens to on an advertising channel index

The scanWindow and scanInterval parameters are less than or equal to 10.24 s.

The scanWindow is less than or equal to the scanInterval.

5.1.3 Connection timing

The timing of connection events is determined by 2 parameters:

- connection event interval (*connInterval*): time interval between the start of two consecutive connection events, which never overlap; the point in time where a connection event starts is named an *anchor point*.

At the anchor point, a master starts transmitting a data channel PDU to the slave, which in turn listens to the packet sent by its master at the anchor point.

The master ensures that a connection event closes at least T_IFS=150 µs (inter frame spacing time, i.e. time interval between consecutive packets on the same channel index) before the anchor point of next connection event.

The connInterval is a multiple of 1.25 ms in the range of 7.5 ms to 4.0 s.

- *slave latency* (*connSlaveLatency*): allows a slave to use a reduced number of connection events. This parameter defines the number of consecutive connection events that the slave device is not required to listen to the master.

When the host wants to create a connection, it provides the controller with the maximum and minimum values of the connection interval (*Conn_Interval_Min*, *Conn_Interval_Max*) and connection length (*Minimum_CE_Length*, *Maximum_CE_Length*) thus giving the controller some flexibility in choosing the current parameters in order to fulfill additional timing constraints e.g. in the case of multiple connections.

5.2 BLE stack timing and slot allocation concepts

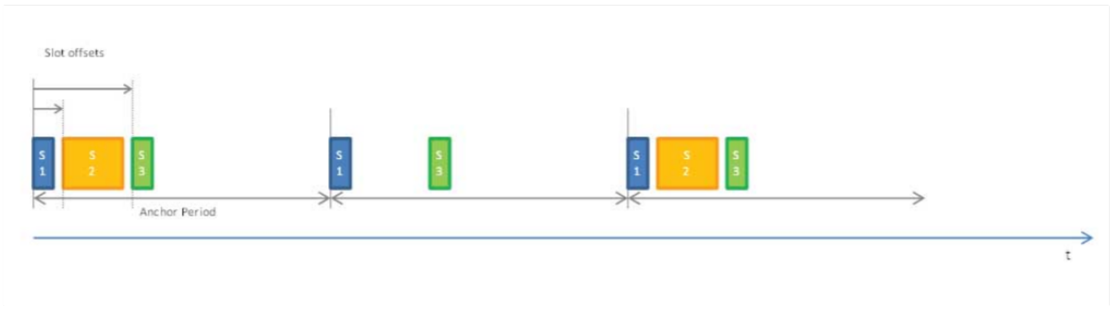
The STM32WB BLE stack adopts a time slotting mechanism in order to allocate simultaneous master and slave connections. The basic parameters, controlling the slotting mechanism, are indicated in the table below:

Table 52. **Timing parameters of the slotting algorithm**

Parameter	Description
Anchor period	Recurring time interval inside which up to 8 connection slots can be allocated. Among these 8 slots, only 1 at a time may be a scanning or advertising slot (they are mutually exclusive)
Slot duration	Time interval inside which a full event (i.e. advertising or scanning, and connection) takes place; the slot duration is the time duration assigned to the connection slot and is linked to the maximum duration of a connection event
Slot offset	Time value corresponding to the delay between the beginning of an anchor period and the beginning of the connection slot
Slot latency	Number representing the actual utilization rate of a certain connection slot in successive anchor periods. (For instance, a slot latency equal to ‘1’ means that a certain connection slot is actually used in each anchor period; a slot latency equal to n means that a certain connection slot is actually used only once every n anchor periods)

Timing allocation concept allows a clean time to handle multiple connections but at the same time imposes some constraints to the actual connection parameters that the controller can accept. An example of the time base parameters and connection slot allocation is shown in the figure below

Figure 21. **Example of allocation of three connection slots**



Slot #1 has offset 0 with respect to the anchor period, slot #2 has slot latency = 2, all slots are spaced by 1.25 ms guard time.

5.2.1 Setting the timing for the first master connection

The time base mechanism above described, is actually started when the first master connection is created. The parameters of such first connection determine the initial value for the anchor period and influence the timing settings that can be accepted for any further master connection simultaneous with the first one.

In particular:

- The initial anchor period is chosen equal to the mean value between the maximum and minimum connection period requested by the host
- The first connection slot is placed at the beginning of the anchor period
- The duration of the first connection slot is set equal to the maximum of the requested connection length

Clearly, the relative duration of such first connection slot compared to the anchor period limits the possibility to allocate further connection slots for further master connections.

5.2.2 Setting the timing for further master connections

Once that the time base has been configured and started as described above, then the slot allocation algorithm tries, within certain limits, to dynamically reconfigure the time base to allocate further host requests.

In particular, the following three cases are considered:

1. The current anchor period falls within the *Conn_Interval_Min* and *Conn_Interval_Max* range specified for the new connection. In this case no change is applied to the time base and the connection interval for the new connection is set equal to the current anchor period.
2. The current anchor period in smaller than the *Conn_Interval_Min* required for the new connection. In this case the algorithm searches for an integer number *m* such that: *Conn_Interval_Min* ≤ *Anchor_Period* × *m* ≤ *Conn_Interval_Max*

   If such value is found then the current anchor period is maintained and the connection interval for the new connection is set equal to *Anchor_Period·m*with slot latency equal to *m*.
3. The current anchor period in larger than the *Conn_Interval_Max* required for the new connection. In this case the algorithm searches for an integer number *k* such that:

   $$Conn\_Interval\_Min \le \frac{Anchor\_Period}{k} \le Conn\_Interval\_Max$$

   If such value is found then the current anchor period is reduced to:

   $$\frac{Anchor\_Period}{k}$$

   The connection interval for the new connection is set equal to:

   $$\frac{Anchor\_Period}{k}$$

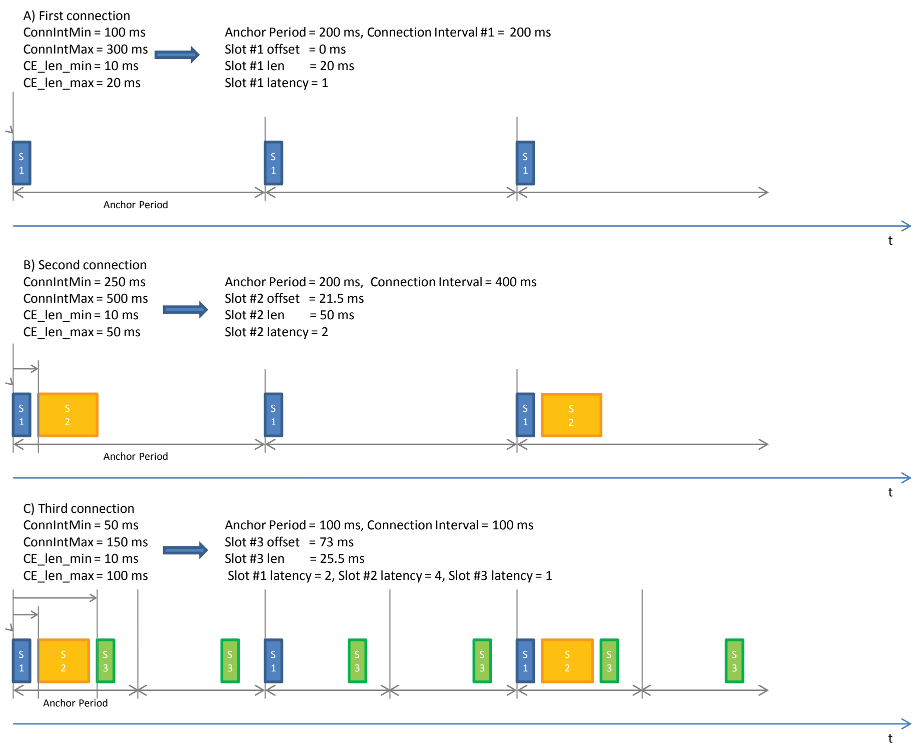
   and the slot latency for the existing connections is multiplied by a factor *k*. Note that in this case the following conditions must also be satisfied:
   - *Anchor_Period/k* must be a multiple of 1.25 ms
   - *Anchor_Period/k* must be large enough to contain all the connection slots already allocated to the previous connections

Once that a suitable anchor period has been found according to the criteria listed above, then a time interval for the actual connection slot is allocated therein. In general, if enough space can be found in the anchor period, the algorithm allocates the maximum requested connection event length otherwise reduces it to the actual free space.

When several successive connections are created, the relative connection slots are normally placed in sequence with a small guard interval between (1.5 ms); when a connection is closed this generally results in an unused gap between two connection slots. If a new connection is created afterwards, then the algorithm first tries to fit the new connection slot inside one of the existing gaps; if no gap is wide enough, then the connection slot is placed after the last one.

Figure 22. Example of timing allocation for three successive connections shows an example of how the time base parameters are managed when successive connections are created.

Figure 22. **Example of timing allocation for three successive connections**



5.2.3 Timing for advertising events

The periodicity of the advertising events, controlled by *advInterval*, is computed based on the following parameters specified by the slave through the host in the `HCI_LE_Set_Advertising_parameters` command:

- *Advertising_Interval_Min, Advertising_Interval_Max;*
- *Advertising_Type*;

if *Advertising_Type* is set to high duty cycle directed advertising, then advertising interval is set to 3.75 ms regardless of the values of *Advertising_Interval_Min* and *Advertising_Interval_Max*; in this case, a timeout is also set to 1.28 s, that is the maximum duration of the advertising event for this case.

In all other cases the advertising interval is chosen equal to the mean value between (*Advertising_Interval_Min* + 5 ms) and (Advertising_Interval_Max + 5 ms). The advertising has not a maximum duration as in the previous case, but it is stopped only if a connection is established, or upon explicit request by host.

The length of each advertising event is set by default by the SW to be equal to 14.6 ms (i.e. the maximum allowed advertising event length) and it cannot be reduced.

Advertising slots are allocated within the same time base of the master slots (i.e. scanning and connection slots). For this reason, the advertising enable command to be accepted by the SW when at least one master slot is active, the advertising interval has to be an integer multiple of the actual anchor period.

5.2.4 Timing for scanning

Scanning timing is requested by the master through the following parameters specified by the host in the HCI_LE_Set_Scan_parameters command:

- LE_Scan_Interval: used to compute the periodicity of the scan slots
- LE_Scan_Window: used to compute the length of the scan slots to be allocated into the master time base

Scanning slots are allocated within the same time base of the other active master slots (i.e. connection slots) and of the advertising slot (if there is one active).

If there is already an active slot, the scan interval is always adapted to the anchor period.

Every time the `LE_Scan_Interval` is greater than the actual anchor period, the SW automatically tries to subsample the `LE_Scan_Interval` and to reduce the allocated scan slot length (up to ¼ of the `LE_Scan_Window`) in order to keep the same duty cycle required by the host, given that scanning parameters are just recommendations as stated by BT official specifications (v.4.1, vol.2, part E, §7.8.10).

5.2.5 Slave timing

The slave timing is defined by the Master when the connection is created so the connection slots for slave links are managed asynchronously with respect to the time base mechanism described above. The slave assumes that the master may use a connection event length as long as the connection interval.

The scheduling algorithm adopts a round-robin arbitration strategy any time a collision condition is predicted between a slave and a master slot. In addition to this, the scheduler may also impose a dynamic limit to the slave connection slot duration to preserve both master and slave connections.

In particular:

- If the end of a master connection slot overlaps the beginning of a slave connection slot then master and slave connections are alternatively preserved/canceled
- If the end of a slave connection slot overlaps the beginning of a master connection slot then the slave connection slot length is hard limited to avoid such overlap. If the resulting time interval is too small to allow for at least a two packets to be exchanged then round-robin arbitration is used.

5.3 Master with multiple slaves connection guidelines

The following guidelines should be followed to properly handle multiple master and slave connections using the STM32WB devices:

1. Avoid over-allocating connection event length: choose *Minimum_CE_Length* and *Maximum_CE_Length* as small as possible to strictly satisfy the application needs. In this manner, the allocation algorithm allocates several connections within the anchor period and reduces the anchor period, if needed, to allocate connections with a small connection interval.
2. For the first master connection:
   a. If possible, create the connection with the shortest connection interval as the first one so to allocate further connections with connection interval multiple of the initial anchor period.
   b. If possible, choose *Conn_Interval_Min* = *Conn_Interval_Max* as multiple of 10 *ms*to allocate further connections with connection interval sub multiple by a factor 2, 4 and 8 (or more) of the initial anchor period being still a multiple of 1.25 *ms*.
3. For additional master connections:
   a. Choose *ScanInterval* equal to the connection interval of one of the existing master connections
   b. Choose *ScanWin* such that the sum of the allocated master slots (including Advertising, if active) is lower than the shortest allocated connection interval
   c. Choose *Conn_Interval_Min* and *Conn_Interval_Max* such that the interval contains either:
      ◦ a multiple of the shortest allocated connection interval
      ◦ a sub multiple of the shortest allocated connection interval being also a multiple of 1,25 *ms*
   d. Choose *Maximum_CE_Length =Minimum_CE_Length* such that the sum of the allocated master slots (including Advertising, if active) plus *Minimum_CE_Length* is lower than the shortest allocated connection interval
4. Every time you start advertising:
   a. If direct advertising, choose Advertising_Interval_Min = Advertising_Interval_Max = integer multiple of the shortest allocated connection interval
   b. If not direct advertising, choose Advertising_Interval_Min = Advertising_Interval_Max such that (Advertising_Interval_Min + 5ms) is an integer multiple of the shortest allocated connection interval
5. Every time you start scanning:
   a. Every time you start scanning: a) choose ScanInterval equal to the connection interval of one of the existing master connections
   b. Choose ScanWin such that the sum of the allocated master slots (including advertising, if active) is lower than the shortest allocated connection interval

6. Keep in mind that the process of creating multiple connections, then closing some of them and creating new ones again, over time, tends to decrease the overall efficiency of the slot allocation algorithm. In case of difficulties in allocating new connections, the time base can be reset to the original state closing all existing connections.

5.4 Master with multiple slaves connection formula

The STM32WB BLE stack multiple master/slave feature offers the capability for one device (called Master_Slave in this context), to handle up to 8 connections at the same time, as follows:

1. Master of multiple slaves:
   – Master_Slave connected up to 8 slaves devices (Master_Slave device is not a slave of any other master device)
2. Simultaneously advertising/scanning and master of multiple slaves:
   a. Master_Slave device connected as a slave to one master device and as a master up to 7 slaves devices
   b. Master_Slave device connected as a slave to two master devices and as a master up to 6 slaves devices

In order to address the highlighted scenarios, the user must properly defines the advertising/scanning and connection parameters to calculate the optimized anchor period allowing the required multiple Master_Slave connection scenario to be handled.

A specific formula allows the required advertising/scanning and connection parameters to be calculated on the highlighted scenarios, where one device (Master_Slave) manages up to Num_Masters master devices, up to Num_Slaves slave devices and performs advertising and scanning with Scan_window length.

The following formula is defined:

- **GET_Master_Slave_device_connection_parameters(Num_Masters, Num_Slaves, Scan_Window, Sleep_Time)**

User is requested to provide the following input parameters, based on its specific application scenario:

Table 53. **Input parameters to define Master_Slave device connection parameters**

Input parameter	Description	Allowed range	Notes
Num_Masters	Number of master devices to which the master/slave should be connected as slave, including the non-connectable advertising	[0-2]	If 0, master device is not slave of any other master device. It can connect up to 8 slave devices at the same time
Num_Slaves	Number of slave devices to which the master/slave should be connected as master	[0 – Allowed_Slaves]	The max. number of slave devices depends on how many master devices Master_Slave device is expected to be connected: Allowed_Slaves = 8 - Num_Masters
Scan_Window	Master_Slave device scan window length in ms	[2.5 - 10240] ms	This input value defines the minimum selected scanning window for Master_Slave device
Sleep_time	Additional time (ms) to be added to the minimum required anchor period	[0-N] ms	0: no additional time is added to the minimum anchor period (which defines the optimized configuration for throughput)

When the user selects Sleep_Time = 0, the **GET_Master_Slave_device_connection_parameters()** formula defines the optimized Master_Slave device connections parameters in order to satisfy the required multiple connection scenarios and keeping the best possible data throughput. If user wants to enhance the power consumption profile, he can add a specific time through the Sleep_Time parameter, which leads to increase the device connection parameters with a benefit on power consumption but with lower data throughput.

Based on the provided input parameters, the formula calculates the following Master_Slave device connections parameters:

- Connection_Interval
- CE_Length
- Advertising_Interval

- Scan_Interval
- Scan_Window
- AnchorPeriodLength

Table 54. **Output parameters for Master_Slave device multiple connections**

Output parameter	Description	Allowed range/ time(ms)	How to use
Connection_Interval	Connection event interval minimum value for the connection event interval	Values: 0x0006 (7.50 ms) ... 0x0C80 (4000.00 ms).Time = N * 1.25 ms	Value to be used for the Conn_Interval_Min, Conn_Interval_Max parameters of created connections APIs (i.e.: ACI_GAP_CREATE_CONNECTION())
CE_Length	Length of connection needed for this LE connection.	Time = N * 0.625 ms	Value to be used for the Minimum_CE_Length, Maximum_CE_Length parameters of created connections APIs (i.e.: ACI_GAP_CREATE_CONNECTION())
Advertising_Interval	Advertising interval	Values: 0x0020 (20.000 ms) ... 0x4000 (10240.000 ms). Time = N * 0.625 ms	Value to be used for the Advertising_Interval_Min, Advertising_Interval_Max parameters of discovery mode, connectable mode APIs (i.e.: ACI_GAP_SET_DISCOVERABLE(), ..)
Scan_Interval	Scanning interval	Values: 0x0004 (2.500 ms) ... 0x4000 (10240.000 ms) Time = N * 0.625 ms	Value to be used for the LE_Scan_Interval parameter of discovery procedures (i.e.: ACI_GAP_CREATE_CONNECTION(), ACI_GAP_START_GENERAL_DISCOVERY_PROC(), ..)
Scan_Window	Scanning window	Values: 0x0004 (2.500 ms) ... 0x4000 (10240.000 ms) Time = N * 0.625 ms	Value to be used for the LE_Scan_Window parameter of discovery procedures (i.e.: ACI_GAP_CREATE_CONNECTION(), ACI_GAP_START_GENERAL_DISCOVERY_PROC(), ..)
AnchorPeriodLength	Minimum time interval used to represent all the periodic master slots associated to Master_Slave device		It is calculated from **GET_Master_Slave_device_connection_parameters()** formula based on input parameters, and it used to define the device connection output parameters

Assumptions: the formula defines internally the number of packets, at maximum length, that can be exchanged to each slave per connection interval.

6 Reference documents

Table 55. Reference documents

Name	Title/description
AN5289	Building wireless applications with STM32WB Series microcontrollers
AN5379	Examples of AT commands on STM32WB Series microcontrollers
AN5270	STM32WB Bluetooth Low Energy (BLE) wireless interface
AN5155	STM32Cube MCU Package examples for STM32WB Series
Bluetooth specifications	Specification of the Bluetooth system (v4.0, v4.1, v4.2, v5.0, v5.1, 5.2)
AN5378	STM32WB Series microcontrollers bring-up procedure
AN5071	STM32WB Series microcontrollers ultra-low-power features overview

7 List of acronyms and abbreviations

This section lists the standard acronyms and abbreviations used throughout the document.

Table 56. List of acronyms

Term	Meaning
ACI	Application command interface
ATT	Attribute protocol
BLE	Bluetooth low energy
BR	Basic rate
CRC	Cyclic redundancy check
CSRK	Connection signature resolving key
EDR	Enhanced data rate
DK	Development kits
EXTI	External interrupt
GAP	Generic access profile
GATT	Generic attribute profile
GFSK	Gaussian frequency shift keying
HCI	Host controller interface
IFR	Information register
IRK	Identity resolving key
ISM	Industrial, scientific and medical
LE	Low energy
L2CAP	Logical link control adaptation layer protocol
LTK	Long-term key
MCU	Microcontroller unit
MITM	Man-in-the-middle
NA	Not applicable
NESN	Next sequence number
OOB	Out-of-band
PDU	Protocol data unit
RF	Radio frequency
RSSI	Received signal strength indicator
SIG	Special interest group
SM	Security manager
SN	Sequence number
USB	Universal serial bus
UUID	Universally unique identifier
WPAN	Wireless personal area networks

Revision history

Table 57. **Document revision history**

Date	Revision	Changes
02-Jul-2020	1	Initial release
11-Dec-2020	2	Added: • Section 4.5.1 Flow charts on pairing procedure: Pairing request by Master sequence (Legacy) • Section 4.5.2 Flow charts on pairing procedure: Pairing request by Master sequence (Secure) • Section 4.5.3 Flow charts on pairing procedure: Pairing request by Slave sequence (secure) • Section 4.8 End to end RX flow control using GATT • Section 4.15 STM32WB formula for converting RSSI raw value in dBm Updated: • Section 2.8.1 Device filtering
11-Feb-2021	3	Updated: • Section Introduction

Contents

List of tables

List of figures