沁恒低功耗蓝牙
软件开发参考手册

V1.3

2021 年 7 月 3 日

目录

前言

本手册针对沁恒低功耗蓝牙的软件开发进行了简单介绍。包括了软件的开发平台，软件开发的基本框架和低功耗蓝牙协议栈等。为方便理解，本手册均以 CH58x 芯片作为例子进行介绍，本司其他低功耗蓝牙芯片的软件开发同样可参考本手册。

CH58x 是一颗 RISC-V 芯片，片上集成两个独立全速 USB 主机和设备控制器及收发器、12-bit ADC，触摸按键检测模块、RTC、电源管理等。有关 CH58x 的详细信息，请参考 CH583DS1.PDF 手册文档。

1. 概述

1.1 介绍

蓝牙自 4.0 版本支持两种无线技术：

· 蓝牙基本速率/增强数据速率（通常称为 BR/EDR 经典蓝牙）

· 低功耗蓝牙

低功耗蓝牙协议的创建旨在于一次传输非常小的数据包，因此与经典蓝牙相比功耗大大下降。

可支持经典蓝牙和低功耗蓝牙的设备称之为双模设备，如移动手机。仅支持低功耗蓝牙的设备称之为单模设备。这些设备主要用于低功耗的应用，如使用纽扣电池供电的应用。

1.2 低功耗蓝牙协议栈基本介绍

低功耗蓝牙协议栈结果如图 1-1 所示。

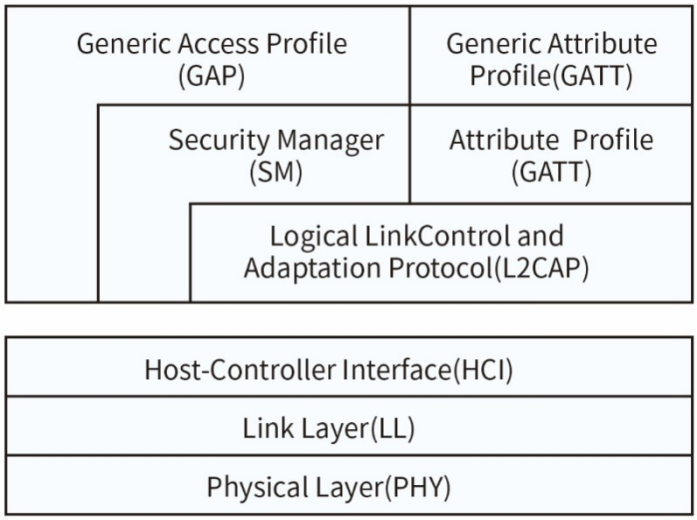


图 1-1

协议栈由 Host（主机协议层）和 Controller（控制协议层）组成，并且这两部分一般是分开执行的。

配置和应用程序是在协议栈的通用访问配置文件（GAP）和通用属性配置文件（GATT）层中实现的。

物理层（PHY）是 BLE 协议栈最底层，它规定了 BLE 通信的基础射频参数，包括信号频率、调制方案等。

物理层是在 2.4GHz 频道，使用高斯频移键控（GFSK - Gauss frequency Shift Keying）技术进行调制。

BLE 5.0 的物理层有三种实现方案，分别是 1Mbps 的无编码物理层、2Mbps 的无编码物理层和 1Mbps 的编码物理层。其中 1Mbps 的无编码物理层与 BLE 4.0 系列协议的物理层兼容，另外两种物理层则分别扩展了通信速率和通信距离。

链路层（LinkLayer）控制设备处于准备（standby）、广播（advertising）、监听/扫描（scanning）、发起（initiating）、连接（connected）这五种状态中一种。围绕这几种状

态，BLE 设备可以执行广播和连接等操作，链路层定义了在各种状态下的数据包格式、时序规范和接口协议。

通用访问协议（Generic Access Profile）是BLE 设备内部功能对外接口层。它规定了三个方面：GAP 角色、模式和规程、安全问题。主要管理蓝牙设备的广播，连接和设备绑定。

广播者——不可以连接的一直在广播的设备

观察者——可以扫描广播设备，但不能发起建立连接的设备

从机——可以被连接的广播设备，可以在单个链路层连接中作为从机

主机——可以扫描广播设备并发起连接，在单个链路层或多链路层中作为主机

逻辑链路控制协议（Logical Link Control and Adaptation Protocol）是主机与控制器直接的适配器，提供数据封装服务。它向上连接应用层，向下连接控制器层，使上层应用操作无需关心控制器的数据细节，。

安全管理协议（Security Manager）提供配对和密匙分发服务，实现安全连接和数据交换。

属性传输协议（Attribute Protocol）定义了属性实体的概念，包括UUID、句柄和属性值，规定了属性的读、写、通知等操作方法和细节。

通用属性规范(Generic Attribute Profile)定义了使用ATT的服务框架和协议的结构，两个设备应用数据的通信是通过协议栈的GATT层实现。

GATT 服务器——为GATT 客户端提供数据服务的设备

GATT 客户端——从GATT 服务器读写应用数据的设备

2. 开发平台

2.1 概述

CH58x 是集成 BLE 无线通讯的 32 位 RISC-V 微控制器。片上集成 2Mbps 低功耗蓝牙通讯模块，两个全速 USB 主机和设备控制器收发器，2 个 SPI，RTC 等丰富外设资源。本手册均以 CH58x 开发平台举例说明，本司其他低功耗蓝牙芯片同样可参考此手册。

2.2 配置

CH58x 是真正的单芯片解决方案，控制器，主机，配置文件和应用程序均在 CH58x 上实现。可参考 Central 和 Peripheral 例程。

2.3 软件概述

软件开发包包括以下六个主要组件：

- TMOS
- HAL
- BLE Stack
- Profiles
- RISC-V Core
- Application

软件包提供了四个 GAP 配置文件：

- Peripheralrole
- Centralrole
- Broadcasterrole
- Observerrole

同时也提供了一些 GATT 配置文件以及应用程序。

详细请参考 CH58xEVT 软件包。

3. 任务管理系统（TMOS）

3.1 概述

低功耗蓝牙协议栈以及应用均基于 TMOS（Task Management Operating System），TMOS 是一个控制循环，通过 TMOS 可设置事件的执行方式。对于一个任务，独一无二的任务 ID，任务的初始化以及任务下可执行的事件都是不可或缺的。

3.2 任务初始化

首先，为保证 TMOS 持续运行，需要在 main()的最后循环执行 TMOS_SystemProcess()。而初始化任务需要调用 tmosTaskID TMOS_ProcessEventRegister(pTaskEventHandlerFn eventCb) 函数，将事件的回调函数注册到 TMOS 中，并生成唯一的 8 位任务 ID。不同的任务初始化后任务 ID 依次递增，而任务 ID 越小任务优先级最高。协议栈任务必须具有最高的优先级。

```
halTaskID = TMOS_ProcessEventRegister( HAL_ProcessEvent );
```

3.3 任务事件及事件的执行

在任务初始化完成之后，TMOS 就会在循环中轮询任务事件，事件的标志存储在 16 位变量中，其中每一位在同一任务中对应一个唯一的事件。需要注意的是（0x8000）不可被定义，因为其对应的是 SYS_EVENT_MSG，即系统消息传递事件。详情请参照 3.5 节。

任务执行的基本结构如图 3.1 所示，TMOS 根据任务的优先级轮询其是否有事件需要执行，若有事件则执行相应的回调函数。当一个循环结束之后，若还有时间空余，系统会进入空闲或睡眠模式。

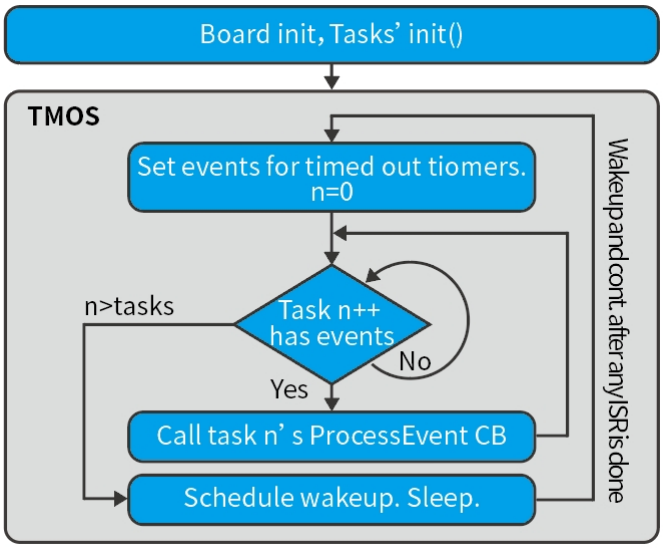


图 3.1 TMOS 任务管理示意图

如在想要在 HAL 层定义一个 TEST 事件，可以在 HAL 层的任务初始化完成之后，在任务回调函数中添加事件 HAL_TEST_EVEN,其基本格式如下：

```
if ( events & HAL_TEST_EVENT )
{
  PRINT( "* \n" );
```

```
    return events ^ HAL_TEST_EVENT;
  }
```

事件执行完成后需要返回对应的 16 位事件变量以清除事件，防止重复处理同一事件，以上代码通过 return events ^ HAL_TEST_EVENT;清除了 HAL_TEST_EVENT 标志。

事件添加完成后，调用 tmos_set_event(halTaskID, HAL_TEST_EVENT)函数即可立即执行对应的事件。其中 halTaskID 为选择执行的任务，HAL_TEST_EVENT 为任务下对应的事件。

```
    tmos_start_task( halTaskID, HAL_TEST_EVENT, 1000 );
```

若不想立即执行某个事件，可调用 tmos_start_task(tmosTaskID taskID, tmosEvents event, tmosTimer time)函数，其功能与 tmos_set_event 类似，区别在于在设置好想要执行的任务的任务 ID 及事件标志后，还需添加第三个参数：事件执行的超时时间。即事件会在达到超时时间后执行。那么在事件中定义下次任务执行的时间即可定时循环执行某个事件。

```
  if ( events & HAL_TEST_EVENT )
  {
    PRINT( "* \n" );
    tmos_start_task( halTaskID, HAL_TEST_EVENT, MS1_TO_SYSTEM_TIME( 100
0 ) );
    return events ^ HAL_TEST_EVENT;
  }
```

此时 TMOS 中仅有一个定时时间 HAL_TEST_EVENT，系统在执行完这个时间后便会进入空闲模式，若打开睡眠功能，则会进入睡眠模式。其实际效果图如图 3.2。

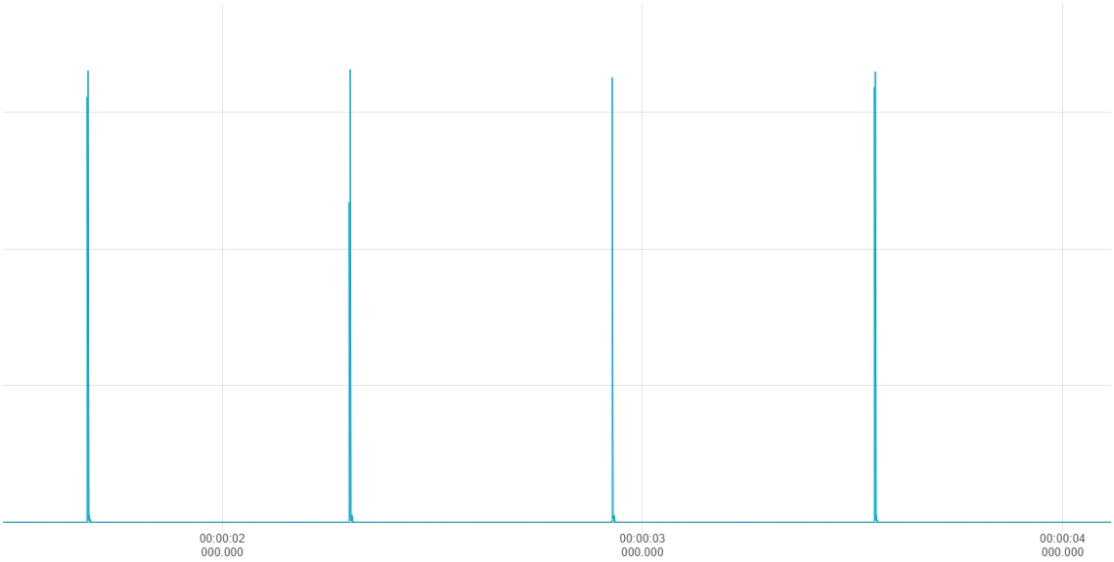


图 3.2 定时任务

3.4 内存管理

TMOS 单独使用一块内存，用户可以自定义内存的使用地址以及大小。TMOS 中的任务事件管理均使用此内存，可通过开启蓝牙绑定功能，加密功能来分析内存的使用情况。

由于低功耗蓝牙的协议栈也是用此内存，所以需要在最大预期工作条件下对其进行测试。

3.5 TMOS 数据传递

TMOS 为不同的任务传递提供了一种接收和发送数据的通信方案。数据的类型是任意的，且在内存足够的情况下长度也可以是任意的。

可按照以下步骤发送一个数据：

1. 使用 tmos_msg_allocate()函数为发送的数据申请内存，若申请成功，则返回内存地址，若失败则返回 NULL。
2. 将数据拷贝到内存中。
3. 调用 tmos_msg_send()函数向指定的任务发送数据的指针。

```
// Register Key task ID
HAL_KEY_RegisterForKeys( centralTaskId )
// Send the address to the task
 msgPtr = ( keyChange_t * ) tmos_msg_allocate( sizeof(keyChange_t));
 if ( msgPtr )
 {
   msgPtr->hdr.event = KEY_CHANGE;
   msgPtr->state = state;
   msgPtr->keys = keys;
   tmos_msg_send( registeredKeysTaskID, ( uint8 * ) msgPtr );
  }
```

数据发送成功后，SYS_EVENT_MSG 被置为有效，此时系统将执行 SYS_EVENT_MSG 事件，在实践中通过调用 tmos_msg_receive()函数检索数据。在数据处理完成后，必须使用 tmos_msg_deallocate()函数释放内存。详情请参考例程。

假设将消息送给 central 的任务 ID，central 的系统事件将会接收到此消息。

```
uint16 Central_ProcessEvent( uint8 task_id, uint16 events ){
if ( events & SYS_EVENT_MSG )  {
    uint8 *pMsg;
    if ( (pMsg = tmos_msg_receive( centralTaskId )) != NULL ){
   central_ProcessTMOSMsg( (tmos_event_hdr_t *)pMsg );
   // Release the TMOS message
     tmos_msg_deallocate( pMsg );
}
```

查询到 KEY_CHANGE 的事件：

```
static void central_ProcessTMOSMsg( tmos_event_hdr_t *pMsg )
{
    switch ( pMsg->event ){
         case KEY_CHANGE:{
       ...
```

4. 应用与协议

4.1 概述

低功耗蓝牙 EVT 例程包括一个简单的 BLE 工程：Peripheral。将此工程烧录置 CH58x 芯片中便可实现一个简单的低功耗蓝牙从机设备。

4.2 工程预览

在 MounRiverStudio 左侧窗口可看到工程文件：

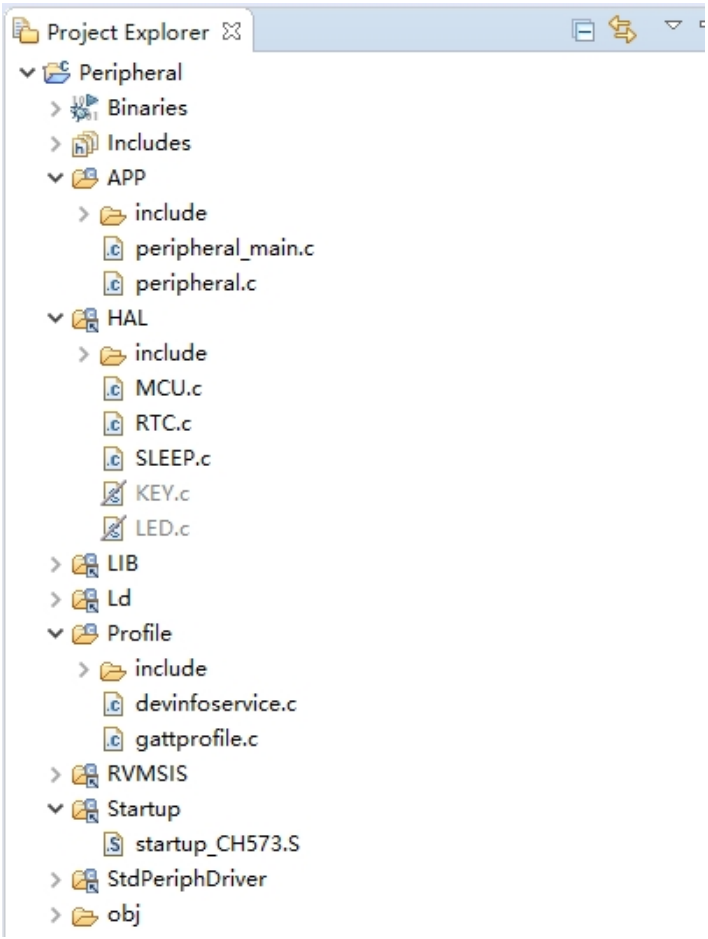


图 4.1 工程文件

文件可分为以下几类：

1. APP - 有关应用的源文件和头文件均可放置于此。
2. HAL - 此文件夹中为 HAL 层的源代码与头文件，HAL 层即蓝牙协议栈与芯片硬件驱动交互层。
3. LIB - 为低功耗蓝牙的协议栈库文件。
4. LD - 链接脚本。
5. Profile - 该文件从包含 GAP 角色配置文件，GAP 安全配置文件和 GATT 配置文件的源代码与头文件，同时也包含 GATT 服务所需的头文件。详情请参考第 5 节。
6. RVMSIS - RISC-V 内核访问的源代码与头文件。
7. Startup - 启动文件。

8. StdPeriphDriver – 包括芯片外设的底层驱动文件。
9. obj – 编译器生成的文件，包括 map 文件与 hex 文件等。

4.3 始于 main()

Main()函数为程序运行的起点，此函数首先对系统时钟进行初始化；然后配置 IO 口状态，防止浮空状态导致工作电流不稳定；接着初始化串口进行打印调试，最后初始化 TMOS 以及低功耗蓝牙。

```
int main( void )
{
#if (defined (DCDC_ENABLE)) && (DCDC_ENABLE == TRUE)
  PWR_DCDCCfg( ENABLE );
#endif
  SetSysClock( CLK_SOURCE_PLL_60MHz );
  GPIOA_ModeCfg( GPIO_Pin_All, GPIO_ModeIN_PU );
  GPIOB_ModeCfg( GPIO_Pin_All, GPIO_ModeIN_PU );
#ifdef DEBUG
  GPIOA_SetBits(bTXD1);
  GPIOA_ModeCfg(bTXD1, GPIO_ModeOut_PP_5mA);
  UART1_DefInit( );
#endif
  PRINT("%s\n",VER_LIB);
  CH58X_BLEInit( );
  HAL_Init( );
  GAPRole_PeripheralInit( );
  Peripheral_Init( );
  while(1){
    TMOS_SystemProcess( );
  }
}
```

4.4 应用初始化

4.4.1 低功耗蓝牙库初始化

低功耗蓝牙库初始化函数 CH58X_BLEInit(),通过配置参数 bleConfig_t 配置库的内存，时钟，发射功率等参数，然后通过 BLE_LibInit()函数将配置参数传进库中。

4.4.2 HAL 层初始化

注册 HAL 层任务，对硬件参数进行初始化，如 RTC 时钟，睡眠唤醒，RF 校准等。

```
void HAL_Init()
{
  halTaskID = TMOS_ProcessEventRegister( HAL_ProcessEvent );
  HAL_TimeInit();
#if (defined HAL_SLEEP) && (HAL_SLEEP == TRUE)
  HAL_SleepInit();
```

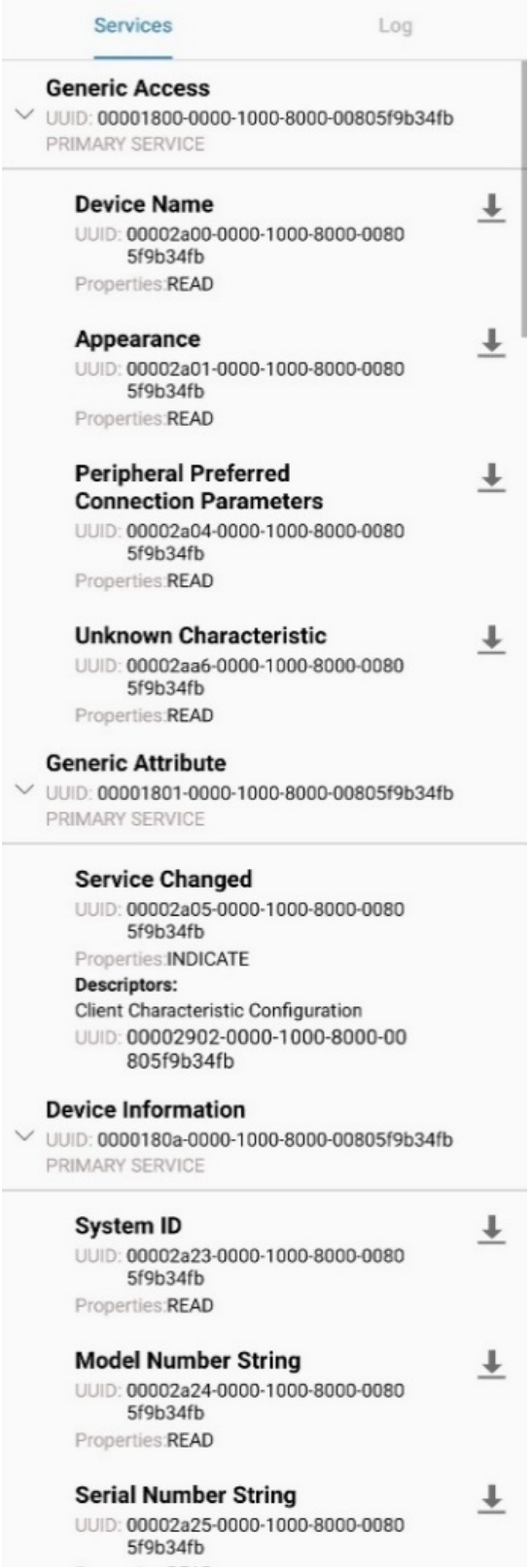
```
#endif
#if (defined HAL_LED) && (HAL_LED == TRUE)
  HAL_LedInit( );
#endif
#if (defined HAL_KEY) && (HAL_KEY == TRUE)
  HAL_KeyInit( );
#endif
#if ( defined BLE_CALIBRATION_ENABLE ) && ( BLE_CALIBRATION_ENABLE == T
RUE )
  tmos_start_task( halTaskID, HAL_REG_INIT_EVENT, MS1_TO_SYSTEM_TIME( B
LE_CALIBRATION_PERIOD ) );    // 添加校准任务，单次校准耗时小于 10ms
#endif
 tmos_start_task( halTaskID, HAL_TEST_EVENT, 1000 );    // 添加一个测试任
务
}
```

4.4.3 低功耗蓝牙从机初始化

此过程包括两个部分：

1. GAP 角色的初始化，此过程由低功耗蓝牙库完成；
2. 低功耗蓝牙从机应用初始化，包括从机任务的注册，参数配置（如广播参数，连接参数，绑定参数等），GATT 层服务的注册，以及回调函数的注册。详见 5.5.3.2 节。

图 4.2 显示了例程 Peripheral 的完整属性表，可作为低功耗蓝牙通讯时的参考。详细信息请参考第五章。


Services
Log
Generic Access
UUID: 00001800-0000-1000-8000-00805f9b34fb
PRIMARY SERVICE
Device Name
UUID: 00002a00-0000-1000-8000-0080 5f9b34fb
Properties:READ
Appearance
UUID: 00002a01-0000-1000-8000-0080 5f9b34fb
Properties:READ
Peripheral Preferred Connection Parameters
UUID: 00002a04-0000-1000-8000-0080 5f9b34fb
Properties:READ
Unknown Characteristic
UUID: 00002aa6-0000-1000-8000-0080 5f9b34fb
Properties:READ
Generic Attribute
UUID: 00001801-0000-1000-8000-00805f9b34fb
PRIMARY SERVICE
Service Changed
UUID: 00002a05-0000-1000-8000-0080 5f9b34fb
Properties:INDICATE
Descriptors:
Client Characteristic Configuration
UUID: 00002902-0000-1000-8000-00 805f9b34fb
Device Information
UUID: 0000180a-0000-1000-8000-00805f9b34fb
PRIMARY SERVICE
System ID
UUID: 00002a23-0000-1000-8000-0080 5f9b34fb
Properties:READ
Model Number String
UUID: 00002a24-0000-1000-8000-0080 5f9b34fb
Properties:READ
Serial Number String
UUID: 00002a25-0000-1000-8000-0080 5f9b34fb

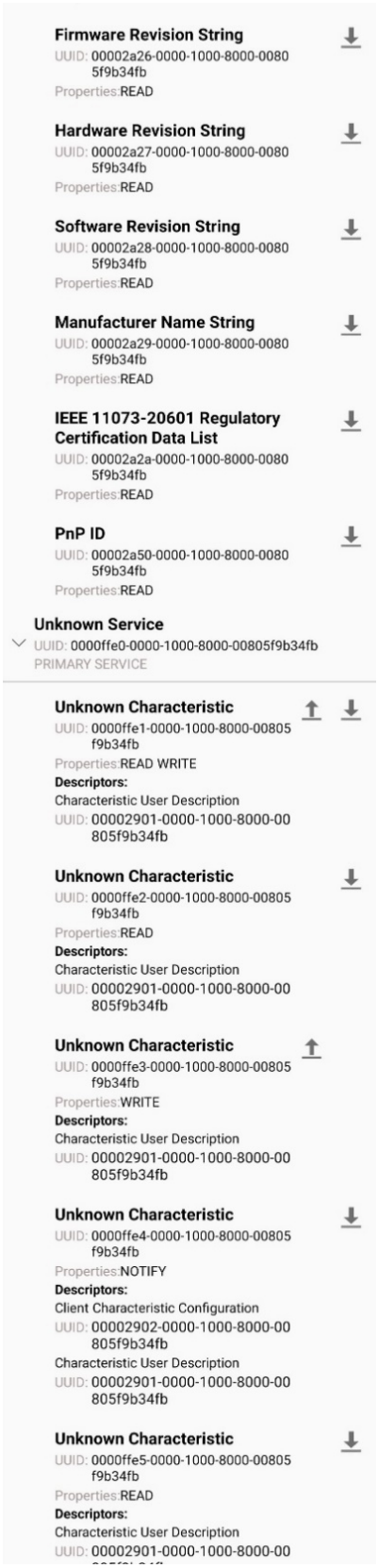

Firmware Revision String
UUID: 00002a26-0000-1000-8000-00805f9b34fb
Properties:READ
Hardware Revision String
UUID: 00002a27-0000-1000-8000-00805f9b34fb
Properties:READ
Software Revision String
UUID: 00002a28-0000-1000-8000-00805f9b34fb
Properties:READ
Manufacturer Name String
UUID: 00002a29-0000-1000-8000-00805f9b34fb
Properties:READ
IEEE 11073-20601 Regulatory Certification Data List
UUID: 00002a2a-0000-1000-8000-00805f9b34fb
Properties:READ
PnP ID
UUID: 00002a50-0000-1000-8000-00805f9b34fb
Properties:READ
Unknown Service
UUID: 0000ffe0-0000-1000-8000-00805f9b34fb
PRIMARY SERVICE
Unknown Characteristic
UUID: 0000ffe1-0000-1000-8000-00805f9b34fb
Properties:READ WRITE
Descriptors:
Characteristic User Description
UUID: 00002901-0000-1000-8000-00805f9b34fb
Unknown Characteristic
UUID: 0000ffe2-0000-1000-8000-00805f9b34fb
Properties:READ
Descriptors:
Characteristic User Description
UUID: 00002901-0000-1000-8000-00805f9b34fb
Unknown Characteristic
UUID: 0000ffe3-0000-1000-8000-00805f9b34fb
Properties:WRITE
Descriptors:
Characteristic User Description
UUID: 00002901-0000-1000-8000-00805f9b34fb
Unknown Characteristic
UUID: 0000ffe4-0000-1000-8000-00805f9b34fb
Properties:NOTIFY
Descriptors:
Client Characteristic Configuration
UUID: 00002902-0000-1000-8000-00805f9b34fb
Characteristic User Description
UUID: 00002901-0000-1000-8000-00805f9b34fb
Unknown Characteristic
UUID: 0000ffe5-0000-1000-8000-00805f9b34fb
Properties:READ
Descriptors:
Characteristic User Description
UUID: 00002901-0000-1000-8000-00

图 4.2 属性表

5. 低功耗蓝牙协议栈

5.1 概述

低功耗蓝牙协议栈的代码在库文件中，并不会提供原代码。但是使用者应该了解这些层的功能，因为他们直接与应用程序进行交互。

5.2 通用访问配置文件（GAP）

5.2.1 概述

低功耗蓝牙协议栈的 GAP 层定义了设备以下几种状态，如图 5.1 所示

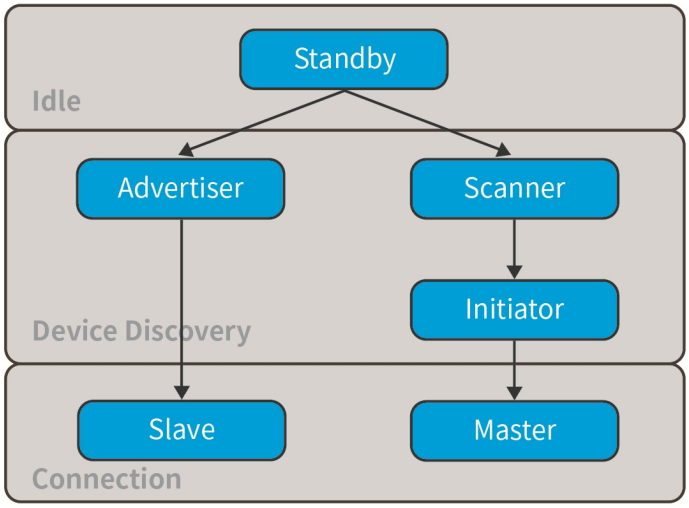


图 5.1 GAP 状态

其中：

Standby：低功耗蓝牙协议栈未启用的空闲状态；

Advertiser：设备使用特定的数据进行广播，广播中可包含设备的名称、地址等数据。广播可表明此设备可被连接。

Scanner：当接收到广播数据后，扫描设备发送扫描请求包给广播者，广播者会返回扫描相应包。扫描者会读取广播者的信息并且判断其是否可以连接。此过程描述了发现设备的过程。

Initiator：建立连接时，连接发起者必须指定用于连接的设备地址，如果地址匹配，则会与广播者建立连接。连接发起者在建立连接时将初始化连接参数。

MasterorSlave：如果设备在连接前是广播者，则其在连接时是从机；如果设备在连接前是发起者，则其在连接后为主机。

5.2.1.1 连接参数

此节描述了连接建立时的连接参数，这些连接参数主机和从机均可修改。

· 连接间隔（ConnectionInterval） - 低功耗蓝牙采用的是跳频方案，设备在特定的时间在特点的通道上发送和接收数据。两个设备的一次数据发送与接收成为一个连接事件。连接间隔则为两个连接事件之间的时间，其时间单位为 1.25ms。连接间隔的范围为 7.5ms~4s。

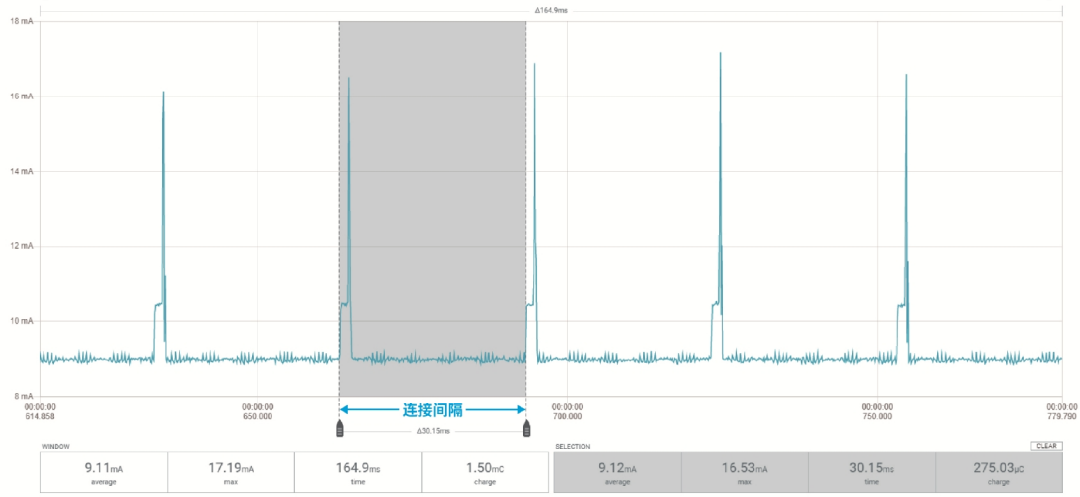


图 5.2 连接事件与连接间隔

不同的应用可能会需要不同的连接间隔，较小的连接间隔会减少数据的相应时间，相应的也会增大功耗。

·从设备延迟（SlaveLatency）－此参数可以让从机跳过部分连接事件。如果设备没有数据需要用发送，那么从机延时可以跳过连接事件并在连接事件期间停止射频，从而降低功耗。从设备延时的值表示可以跳过的最大事件数，其范围为 0~499。但需保证有效连接间隔小于 16s。关于有效连接间隔请参考 5.2.1.2

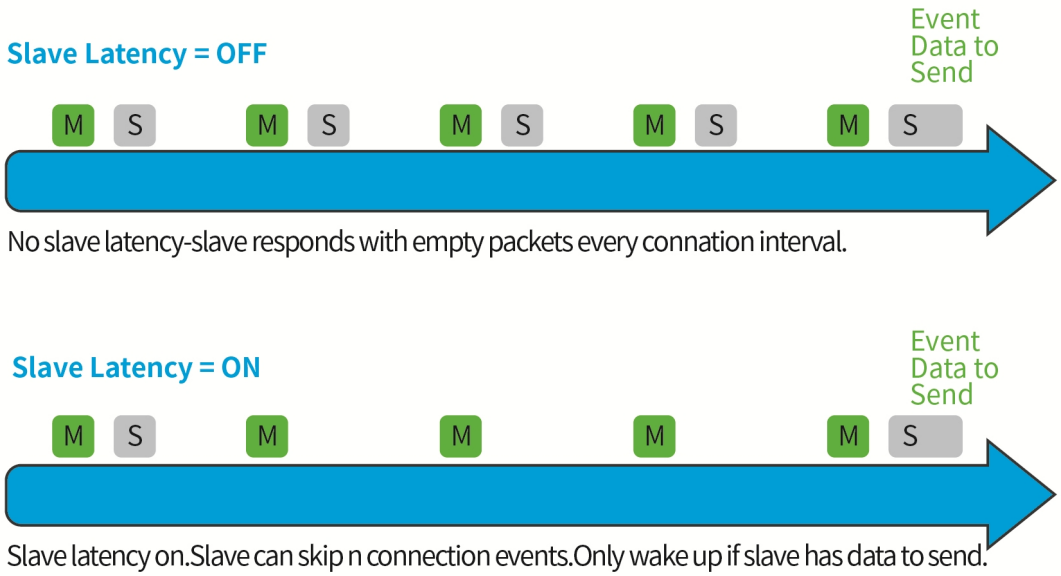


图 5.3 从设备延时

·监督超时（SupervisionTime-out） -此参数为两个有效连接事件间的最大时间。如果超过此时间没有有效的连接事件，则认为连接已断开，设备退回未连接状态。监督超时的范围为 10（100ms）~3200（32s）。超时时间必须大于有效连接间隔。

5.2.1.2 有效连接间隔

在从设备延迟启用且没有数据传输的情况下，有效连接间隔为两个连接事件之间的时间。若不启用从设备延迟或其值为 0 的情况下，有效连接间隔即配置的连接间隔。

其计算公式如下：

$$有效连接间隔 = (连接间隔) \times （1 + 从设备延迟）$$

当

连接间隔：为 80（100ms）

从设备延迟：4

有效连接间隔：$(100\text{ms}) \times (1 + 4) = 500\text{ms}$

那么在没有数据传输的情况下，从机将每隔 500ms 发起一次连接事件。

5.2.1.3 连接参数注意事项

合理的连接参数有助于优化低功耗蓝牙的功耗以及性能，以下总结了连接参数设置中的折衷方案：

减少连接间隔将：

- 增加主机从机功耗
- 增加两个设备之间的吞吐量
- 减少数据往返两设备所需的时间

增加连接间隔将：

- 减少主机从机功耗
- 减少两个设备之间的吞吐量
- 增大数据往返两设备所需的时间

减少从设备延迟或将其设置为 0 将：

- 增加从机的的功耗
- 减少主机向从机发送数据所需要的时间

增加从设备延迟将：

- 当没有数据需要发送给主机时，减少从机的功耗
- 增加主机向从机发送数据所需要的时间

5.2.1.4 连接参数更新

在一些应用中，从机可能需要在连接期间根据应用程序更改连接参数。从机可以将连接参数更新请求发送至主机以更新连接参数。对于蓝牙 4.0，协议栈的 L2CAP 层将处理此请求。

该请求包含以下参数：

- 最小连接间隔
- 最大连接间隔
- 从设备延迟
- 监督超时

这些参数为从机所请求的连接参数，但是主机可以拒绝该请求。

5.2.1.5 终止连接

主机或从机可以出于任何原因终止连接。当任一设备启用终止连接时，另一设备必须在两设备断开连接之前作出终止连接的响应。

5.2.2 GAP 抽象层

应用程序可以通过调用 GAP 层的 API 函数来实现响应的 BLE 功能，如广播与连接。

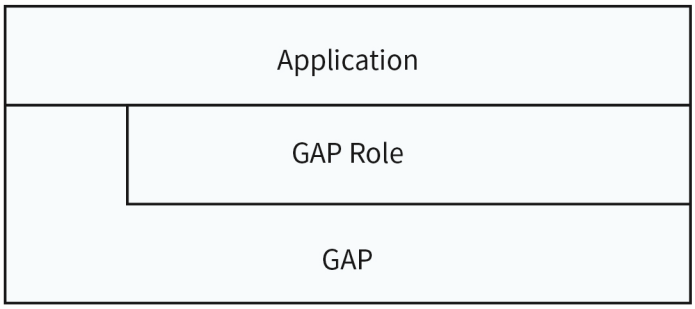


图 5.4 GAP 抽象层

5.2.3 GAP 层配置

GAP 层大部分功能都是在库中实现，用户可以在 CH58xBLE_LIB.h 中找到相应的函数声明。

第 8.1 节定义了 GAP API,可通过 GAPRole_SetParameter()和 GAPRole_GetParamenter()来设置和检测参数，如广播间隔，扫描间隔等。GAP 层配置示例如下：

```
// Setup the GAP Peripheral Role Profile
{
  uint8 initial_advertising_enable = TRUE;
  uint16 desired_min_interval = DEFAULT_DESIRED_MIN_CONN_INTERVAL;
  uint16 desired_max_interval = DEFAULT_DESIRED_MAX_CONN_INTERVAL;
 GAPRole_SetParameter( GAPROLE_MIN_CONN_INTERVAL, sizeof( uint16 ), &
desired_min_interval );
  GAPRole_SetParameter( GAPROLE_MAX_CONN_INTERVAL, sizeof( uint16 ),
&desired_max_interval );

}
```

5.3 GAPRole 任务

正如 4.4 节所描述的那样，GAPRole 是一个单独的任务(GAPRole_PeripheralInit)，GAPRole 大部代码在蓝牙库中运行，从而简化应用层程序。该任务在初始化期间由应用程序启动和配置。且存在回调，应用程序可向 GAPRole 任务注册回调函数。

根据设备的配置，GAP 层可以运行以下四种角色：

- 广播者（Broadcaster） - 仅广播无法被连接
- 观察者（Observer） -仅扫描广播而无法建立连接
- 外围设备（Peripheral） -可广播且可作为从机在链路层建立连接
- 中心设备（Central） - 可扫描广播也可作为主机在链路层建立单个或多个连接

下面介绍外围设备（Peripheral）以及中心设备（Central）这两个角色。

5.3.1 外围设备角色（PeripheralRole）

初始化外围设备的常规步骤如下：

1. 初始化 GAPRole 参数，如以下代码所示。

```
// Setup the GAP Peripheral Role Profile
{
   uint8 initial_advertising_enable = TRUE;
```

```
    uint16 desired_min_interval = DEFAULT_DESIRED_MIN_CONN_INTERVAL;
    uint16 desired_max_interval = DEFAULT_DESIRED_MAX_CONN_INTERVAL;

     // Set the GAP Role Parameters
    GAPRole_SetParameter( GAPROLE_ADVERT_ENABLED, sizeof( uint8 ), &i
    nitial_advertising_enable );
    GAPRole_SetParameter( GAPROLE_SCAN_RSP_DATA, sizeof ( scanRspData
     ), scanRspData );
    GAPRole_SetParameter( GAPROLE_ADVERT_DATA, sizeof( advertData ),
    advertData );
    GAPRole_SetParameter( GAPROLE_MIN_CONN_INTERVAL, sizeof( uint16 )
, &desired_min_interval );
     GAPRole_SetParameter( GAPROLE_MAX_CONN_INTERVAL, sizeof( uint16
), &desired_max_interval );
  }

  // Set the GAP Characteristics
  GGS_SetParameter( GGS_DEVICE_NAME_ATT, GAP_DEVICE_NAME_LEN, attDev
iceName );

  // Set advertising interval
  {
    uint16 advInt = DEFAULT_ADVERTISING_INTERVAL;

    GAP_SetParamValue( TGAP_DISC_ADV_INT_MIN, advInt );
    GAP_SetParamValue( TGAP_DISC_ADV_INT_MAX, advInt );
  }
```

2. 初始化 GAPRole 任务，包括将函数指针传递给应用程序回调函数。

```
if ( events & SBP_START_DEVICE_EVT ){
// Start the Device
  GAPRole_PeripheralStartDevice( Peripheral_TaskID, &Peripheral_Bond
MgrCBs, &Peripheral_PeripheralCBs );
  return ( events ^ SBP_START_DEVICE_EVT );
}
```

3. 从应用层发送 GAPRole 命令

应用层进行连接参数更新。

```
 // Send connect param update request
 GAPRole_PeripheralConnParamUpdateReq(peripheralConnList.connHandle,
                                 DEFAULT_DESIRED_MIN_CONN_INTERVAL,
                                 DEFAULT_DESIRED_MAX_CONN_INTERVAL,
                                DEFAULT_DESIRED_SLAVE_LATENCY,
                                 EFAULT_DESIRED_CONN_TIMEOUT,
```

```
                                    Peripheral_TaskID);
```

协议栈接收到命令，执行参数更新操作，并返回相应状态。

4. GAPRole 任务将协议栈与 GAP 有关的事件传递给应用层。
   蓝牙协议栈接收到连接断开的命令并传递给 GAP 层。
   GAP 层收到命令，直接通过回调函数传递给应用层。

```
static void peripheralStateNotificationCB( gapRole_States_t newState
, gapRoleEvent_t * pEvent )
{
    switch ( newState )
    {
        . . .
        case GAPROLE_ADVERTISING:
        if( pEvent->gap.opcode == GAP_LINK_TERMINATED_EVENT )
        {
            . . .
```

5.3.2 中心设备角色（CentralRole）

初始化中心设备的常规操作如下：

1. 初始化 GAPRole 参数，如以下代码所示。

```
uint8 scanRes = DEFAULT_MAX_SCAN_RES;
GAPRole_SetParameter( GAPROLE_MAX_SCAN_RES, sizeof(uint8), &scanRes
);
```

2. 初始化 GAPRole 任务，包括将函数指针传递给应用程序回调函数。

```
if ( events & START_DEVICE_EVT )
{
        // Start the Device
GAPRole_CentralStartDevice( centralTaskId, &centralBondCB, &central
RoleCB );
    return ( events ^ START_DEVICE_EVT );
}
```

3. 从应用层发送 GAPRole 命令
   应用层调用应用函数，发送 GAP 命令。

```
GAPRole_CentralStartDiscovery( DEFAULT_DISCOVERY_MODE,
                               DEFAULT_DISCOVERY_ACTIVE_SCAN,
                               DEFAULT_DISCOVERY_WHITE_LIST );
```

   GAP 层发送命令给蓝牙协议栈，协议栈收到命令，执行扫描操作，并返回相应状态

4. GAPRole 任务将协议栈与 GAP 有关的事件传递给应用层。
   蓝牙协议栈接收到连接断开的命令并传递给 GAP 层。
   GAP 层收到命令，直接通过回调函数传递给应用层。

```
static void centralEventCB( gapRoleEvent_t *pEvent )
{
```

```
        switch ( pEvent->gap.opcode )
        {
        . . .
            case GAP_DEVICE_DISCOVERY_EVENT:
            {
                uint8 i;
               // See if peer device has been discovered
                for ( i = 0; i < centralScanRes; i++ )
                {
          if ( tmos_memcmp( PeerAddrDef, centralDevList[i].addr , B_ADD
R_LEN ) )
                break;
         }
        . . .
```

5.4 GAP 绑定管理

GAPBondMgr 协议处理低功耗蓝牙连接中的安全管理，使得某些数据仅在经过身份验证后才能被读写。

表 5.1 GAP 绑定管理术语

术语	描述
配对（Pairing）	密钥交互的过程
加密（Encryption）	配对后数据被加密，或是重新加密
验证（Authentication）	配对过程以中间人（MITM：Manin theMiddle）保护完成
绑定（Bonding）	将加密密钥存储在非易失性存储器中，用于下一个加密序列
授权（Authorization）	除身份验证外，还进行附加的应用程序级密钥交换
无线之外（OOB）	密钥不是通过无线的方式交换，而是通过诸如串口或 NFC 等其他来源进行交换。这也提供了 MITM 保护。
中间人（MITM）	中间人保护。这样可以防止监听无线传输的密钥来破解加密
直接连接(JustWorks)	无中间人的配对方式。

建立安全连接的一般过程如下：

1. 密钥配对（包括以下两种方式）。
   A. JustWorks，通过无线发送密钥
   B. MITM，通过中间人发送密钥
2. 通过密钥加密连接。
3. 绑定密钥，存储密钥。
4. 当再次连接时，使用存储的密钥加密连接。

5.4.1 关闭配对

```
 uint8 pairMode = GAPBOND_PAIRING_MODE_NO_PAIRING;
  GAPBondMgr_SetParameter( GAPBOND_PERI_PAIRING_MODE, sizeof ( uint8
), &pairMode );
```

当关闭配对，协议栈将拒绝任何配对尝试。

5.4.2 直接配对配对但不绑定

```
uint8 mitm = FALSE;
uint8 bonding = FALSE;
uint8 pairMode = GAPBOND_PAIRING_MODE_WAIT_FOR_REQ;
GAPBondMgr_SetParameter( GAPBOND_PERI_PAIRING_MODE, sizeof ( uint8 )
, &pairMode );
GAPBondMgr_SetParameter( GAPBOND_PERI_MITM_PROTECTION, sizeof ( uint
8 ), &mitm );
GAPBondMgr_SetParameter( GAPBOND_PERI_BONDING_ENABLED, sizeof ( uint
8 ), &bonding );
```

需要注意的是，开启配对功能，还需要配置设备的 IO 功能，即设备是否支持显示输出和键盘输入,若设备实际不支持键盘输入,但是配置为通过设备输入密码,则无法建立配对。

```
uint8 ioCap = GAPBOND_IO_CAP_DISPLAY_ONLY;
GAPBondMgr_SetParameter( GAPBOND_PERI_IO_CAPABILITIES, sizeof ( uint
8 ), &ioCap );
```

5.4.3 通过中间人配对绑定

```
uint32 passkey = 0; // passkey "000000"
uint8 pairMode = GAPBOND_PAIRING_MODE_WAIT_FOR_REQ;
uint8 mitm = TRUE;
uint8 bonding = TRUE;
uint8 ioCap = GAPBOND_IO_CAP_DISPLAY_ONLY;
GAPBondMgr_SetParameter( GAPBOND_PERI_DEFAULT_PASSCODE, sizeof ( ui
nt32 ), &passkey );
GAPBondMgr_SetParameter( GAPBOND_PERI_PAIRING_MODE, sizeof ( uint8
), &pairMode );
GAPBondMgr_SetParameter( GAPBOND_PERI_MITM_PROTECTION, sizeof ( uin
t8 ), &mitm );
GAPBondMgr_SetParameter( GAPBOND_PERI_IO_CAPABILITIES, sizeof ( uin
t8 ), &ioCap );
 GAPBondMgr_SetParameter( GAPBOND_PERI_BONDING_ENABLED, sizeof ( uin
t8 ), &bonding );
```

使用中间人进行配对并绑定，通过 6 位密码生成密钥。

5.5 通用属性配置文件（GATT）

GATT 层供应用程序在两个连接设备之间进行数据通讯，数据以特征的形式传递和储存。在 GATT 中，当两个设备连接时，它们将各种扮演以下两种角色之一：

- GATT 服务器 – 该设备提供 GATT 客户端读取或写入特征数据库。
- GATT 客户端 – 该设备从 GATT 服务器读写数据。

图 5.5 显示了低功耗蓝牙服务器和客户端的关系，其中外围设备（低功耗蓝牙模块）为 GATT 服务器，中央设备（智能手机）为 GATT 客户端。

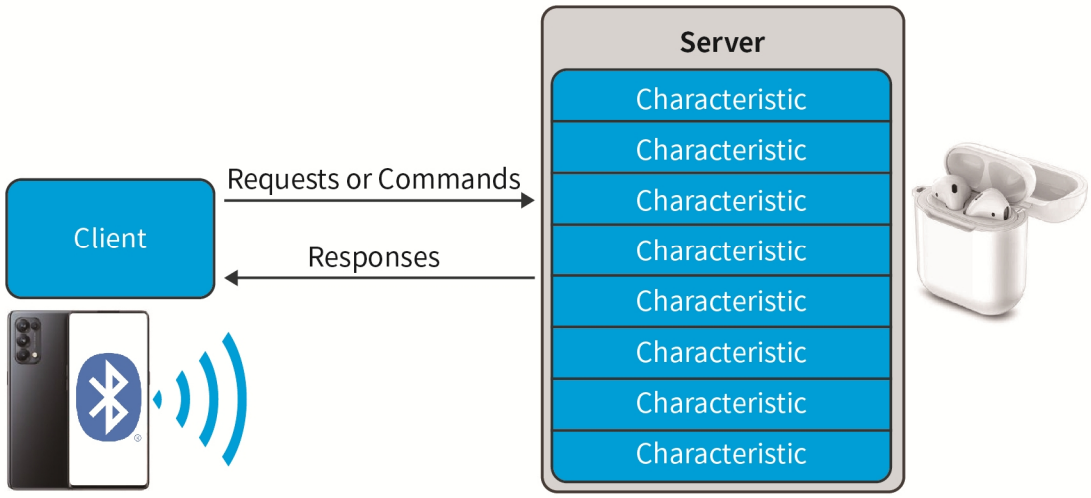


图 5.5 GATT 服务器和客户端

通常 GATT 的服务器和客户端角色是独立于 GAP 的中央设备外围设备角色。外围设备可以是 GATT 客户端或服务器，中央设备也可以是 GATT 服务器或客户端。设备还可以同时充当 GATT 服务器或客户端。

5.5.1 GATT 特征及属性

典型特征由下一属性构成：

- 特征值（Characteristic Value）：此值为特征的数据值。
- 特征声明（Characteristic Declaration）：存储特征值的属性，位置以及类型。
- 客户端特征配置（Client Characteristic Configuration）：通过此配置，GATT 服务器可以配置要发送到 GATT 服务器的属性（notified），或配置要发送到 GATT 服务器的属性并期望得到确认（indicated）。

GATT 服务器可以配置需要发送到 GATT 服务器的属性（notified），或者发送到 GATT 服务器并且期望得到一个回应（indicated）。

- 特征用户描述（CharacteristicUserDescription）：描述特征值的 ASCII 字符串。

这些属性存储在 GATT 服务器的属性表中，以下特征与每个属性相关联：

- 句柄（Handle） -表中属性的索引，每个属性都有唯一的句柄。
- 类型（Type） - 此属性指示属性代表什么，称为通用唯一标识符（UUID）。部分 UUID 由 BluetoothSIG 定义，其他一些 UUID 可由用户自定义。
- 权限（Permissions） -用于限制 GATT 客户端访问该属性的值的权限与方式。
- 值（pValue） - 指向属性值的指针，在初始化后其长度无法改变。最大大小 512 字节。

5.5.2 GATT 客户端抽象层

GATT 客户端没有属性表，应为客户端是接收信息而非提供信息。GATT 层大多数接口直接来自应用程序。

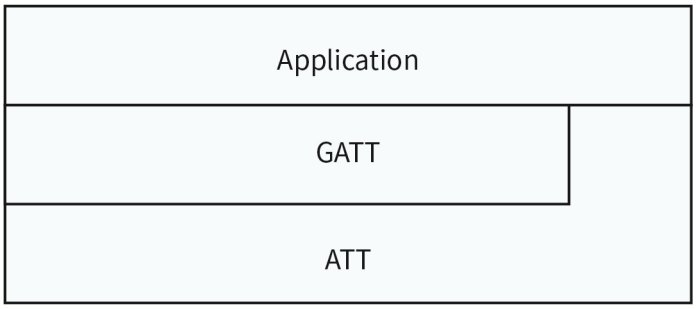


图 5.6 GATT 客户端抽象层

5.5.2.1 GATT 层的应用

此节描述了如何直接在应用中使用 GATT 客户端。可以在例程序 Central 找到相应的源码。

1. 初始化 GATT 客户端。

```
// Initialize GATT Client
GATT_InitClient();
```

2. 注册相关信息以接收传入的 ATT 指示和通知。

```
// Register to receive incoming ATT Indications/Notifications
GATT_RegisterForInd( centralTaskId );
```

3. 执行客户端的程序，如 GATT_WriteCharValue()，即向服务器发送数据。

```
bStatus_t GATT_WriteCharValue( uint16 connHandle, attWriteReq_t *pReq, uint8 taskId )
```

4. 应用程序接收并处理 GATT 客户端的响应，以下为“写”操作的响应。

首先协议栈接收到写响应，并通过任务 TMOS 消息发送至应用层。

```
uint16 Central_ProcessEvent( uint8 task_id, uint16 events )
{
    if ( events & SYS_EVENT_MSG )
    {
        uint8 *pMsg;
        if ( (pMsg = tmos_msg_receive( centralTaskId )) != NULL )
        {
        central_ProcessTMOSMsg( (tmos_event_hdr_t *)pMsg );
    ...
```

应用层的任务查询到 GATT 消息：

```
static void central_ProcessTMOSMsg( tmos_event_hdr_t *pMsg )
{
    switch ( pMsg->event )
    {
        case GATT_MSG_EVENT:
        centralProcessGATTMsg( (gattMsgEvent_t *) pMsg );
```

根据接收内容，应用层可作出相应功能：

```
static void centralProcessGATTMsg( gattMsgEvent_t *pMsg )
{
    ...
    else if ( ( pMsg->method == ATT_WRITE_RSP ) ||
    ( ( pMsg->method == ATT_ERROR_RSP ) &&
      ( pMsg->msg.errorRsp.reqOpcode == ATT_WRITE_REQ ) ) )
    {
    //Application
    ...
```

应用处理完成后清除消息：

```
    // Release the TMOS message
  tmos_msg_deallocate( pMsg );
}
// return unprocessed events
return (events ^ SYS_EVENT_MSG);
  }
```

5.5.3 GATT 服务器抽象层

作为 GATT 服务器，大多数 GATT 功能都能通过 GATTServApp 进行配置。

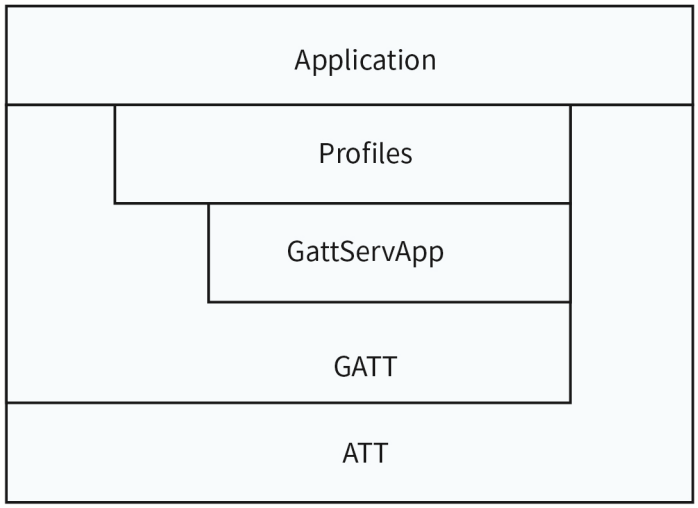


图 5.7 GATT 服务器抽象层

GATT 的使用规范如下：

1. 创建 GATT 配置文件（GATT Profile）对 GATTServApp 模块进行配置。
2. 使用 GATTServApp 模块中的 API 接口对 GATT 层进行操作。

5.5.3.1 GATTServApp 模块

GATTServApp 模块用于存储和管理应用程序的属性表，各种配置文件都是使用该模块将其特征值添加到属性表中。其功能包括：查找特定的属性，读取客户端的特征值以及修改客户端的特征值。详细请参考 API 章节。

每一次初始化，应用程序都会使用 GATTServApp 模块添加服务构建 GATT 表。每一个服

务的内容包括：UUID，值，权限以及读/写权限。图 5.8 描述了 GATTServApp 模块添加服务的过程。

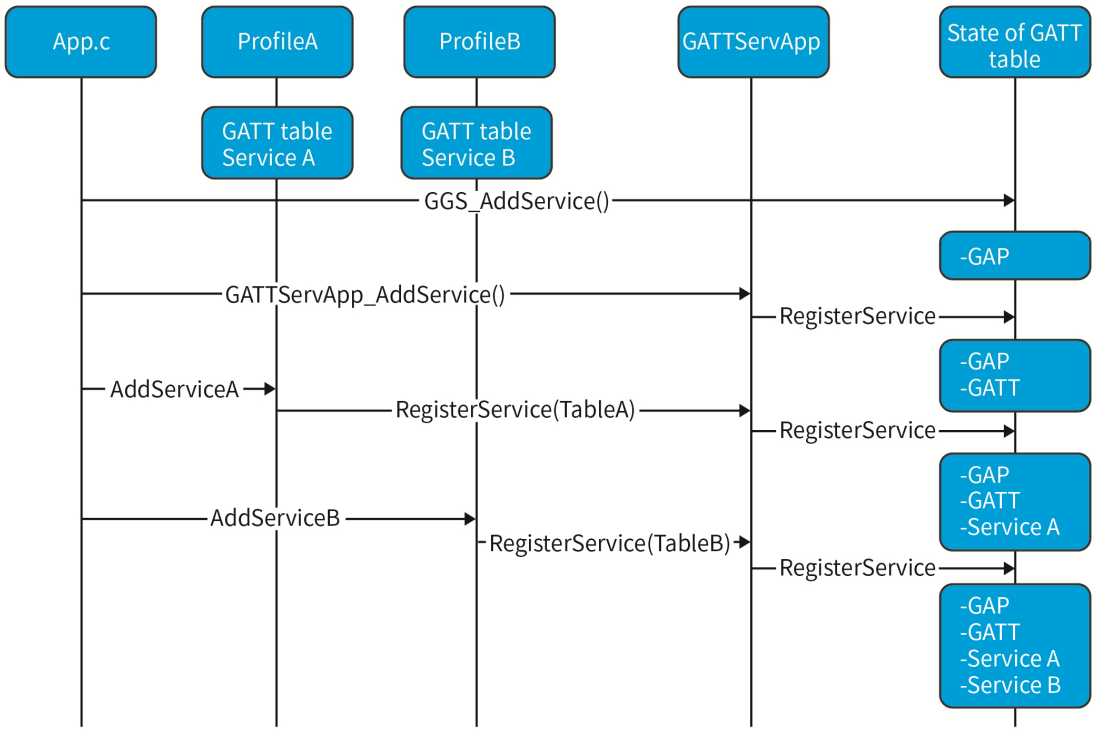


图 5.8 属性表初始化

GATTServApp 的初始化可在 Peripheral_Init()函数中找到。

```
// Initialize GATT attributes
GGS_AddService( GATT_ALL_SERVICES );            // GAP
GATTServApp_AddService( GATT_ALL_SERVICES );    // GATT attributes
DevInfo_AddService();                           // Device Informatio
n Service
SimpleProfile_AddService( GATT_ALL_SERVICES );  // Simple GATT Profi
le
```

5.5.3.2 配置文件架构

本节介绍了配置文件的基本架构，并提供了 Peripheral 工程中的 GATTProfile 的使用示例。

5.5.3.2.1 创建属性表

每一个服务必须定义一个固定大小的属性表，用于传递给 GATT 层。

在 Peripheral 工程中，定义如下：

```
static gattAttribute_t simpleProfileAttrTbl[] =
...
```

每一个属性的格式如下：

```
typedef struct attAttribute_t
{
    gattAttrType_t type; //!< Attribute type (2 or 16 octet UUIDs)
```

```
        uint8 permissions;    //!< Attribute permissions
  uint16 handle;        //!< Attribute handle - assigned internally by a
ttribute server
  uint8* pValue;        //!< Attribute value - encoding of the octet arr
    ay is defined in
                        //!< the applicable profile. The maximum length
    of an attribute
                        //!< value shall be 512 octets.
    } gattAttribute_t;
```

属性中的各个元素：

- type -与属性相关的 UUID。

```
typedef struct
{
    uint8 len;          //!< Length of UUID (2 or 16)
    const uint8 *uuid; //!< Pointer to UUID
} gattAttrType_t;
```

其中 len 可以是 2bytes 也可以是 16bytes。*uuid 可以是指向保存在蓝牙 SIG 中的数字也可以是用于自定义的 UUID 指针。

- Permission - 配置 GATT 客户端设备是否可以访问属性的值。可配置的权限如下：
  - GATT_PERMIT_READ //可读
  - GATT_PERMIT_WRITE // 可写
  - GATT_PERMIT_AUTHEN_READ // 需身份验证读
  - GATT_PERMIT_AUTHEN_WRITE //需身份验证写
  - GATT_PERMIT_AUTHOR_READ // 需授权读
  - GATT_PERMIT_ENCRYPT_READ // 需加密读
  - GATT_PERMIT_ENCRYPT_WRITE // 需加密写
- Handle - GATTServApp 分配的句柄，句柄是按照顺序自动分配的。
- pValue -指向属性值的指针。在初始化后其长度无法改变。最大大小 512 字节。

下面创建 Peripheral 工程中的属性表：

首先创建服务属性：

```
  // Simple Profile Service
{
    { ATT_BT_UUID_SIZE, primaryServiceUUID }, /* type */
    GATT_PERMIT_READ,                         /* permissions */
    0,                                        /* handle */
    (uint8 *)&simpleProfileService            /* pValue */
},
```

此属性为 Bluetooth SIG 定义的主要服务 UUID(0x2800)。GATT 客户端必须读取此属性，所以将权限设置为可读。pValue 是指向服务的 UUID 的指针，自定义为 0xFFE0。

```
// Simple Profile Service attribute
static CONST gattAttrType_t simpleProfileService = { ATT_BT_UUID_SIZ
E, simpleProfileServUUID };
```

然后创建特征的声明，值，用户说明，以及客户端特征配置，5.5.1 节对此进行了介绍。

```
    // Characteristic 1 Declaration
  {
    { ATT_BT_UUID_SIZE, characterUUID },
    GATT_PERMIT_READ,
    0,
    &simpleProfileChar1Props
  },
```

属性特征声明(Characteristic Declaration)的类型需要设置成 BluetoothSIG 定义的特征 UUID 值（0x2803），而 GATT 客户端必须读取此 UUID，所以其权限设置为可读。声明的值指的是此特征的属性，为可读可写。

```
// Simple Profile Characteristic 1 Properties
static uint8 simpleProfileChar1Props = GATT_PROP_READ | GATT_PROP_WR
ITE;


 // Characteristic Value 1
  {
     { ATT_BT_UUID_SIZE, simpleProfilechar1UUID },
     GATT_PERMIT_READ | GATT_PERMIT_WRITE,
     0,
     simpleProfileChar1
  },
```

特征值中，类型设置为自定义的 UUID（0xFFF1），由于特征值的属性是可读可写的，所以设置值的权限为可读可写。pValue 指向实际值的位置，如下：

```
// Characteristic 1 Value
static uint8 simpleProfileChar1[SIMPLEPROFILE_CHAR1_LEN] = { 0 };

// Characteristic 1 User Description
  {
     { ATT_BT_UUID_SIZE, charUserDescUUID },
       GATT_PERMIT_READ,
      0,
      simpleProfileChar1UserDesp
  },
```

用户说明中，类型设置为 Bluetooth SIG 定义的特征 UUID 值（0x2901），其权限设置为可读。值为用户自定义的字符串，如下：

```
// Simple Profile Characteristic 1 User Description
static uint8 simpleProfileChar1UserDesp[] = "Characteristic 1\0";

// Characteristic 4 configuration
 {
     { ATT_BT_UUID_SIZE, clientCharCfgUUID },
```

```
        GATT_PERMIT_READ | GATT_PERMIT_WRITE,
        0,
        (uint8 *)simpleProfileChar4Config
    },
```

该类型须设置为 Bletooth SIG 定义的客户端特征配置 UUID（0x2902），GATT 客户端必须要对此进行读写，所以权限设置为可读可写。pValue 指向客户端特征配置数值的地址。

```
static gattCharCfg_t simpleProfileChar4Config[4];
```

5.5.3.2.2 添加服务

首先需要给客户端特征配置即 Client Characteristic Configuration（CCC）开辟空间。

```
static gattCharCfg_t simpleProfileChar4Config[4];
```

然后初始化 CCC 数组。

```
// Initialize Client Characteristic Configuration attributes
GATTServApp_InitCharCfg( INVALID_CONNHANDLE, battLevelClientCharCfg
);
```

最后通过 GATTServApp 注册配置文件。

```
 // Register GATT attribute list and CBs with GATT Server App
status = GATTServApp_RegisterService( battAttrTbl,
                                      GATT_NUM_ATTRS( battAttrTbl ),
                                      GATT_MAX_ENCRYPT_KEY_SIZE,
                                      &battCBs );
```

5.5.3.2.3 注册应用程序回调函数

在 Peripheral 工程中，每当 GATT 客户端写入特征值时，GATTProfile 就会调用应用程序回调。要使用回调函数，首先需要在初始化时设置回调函数。

```
bStatus_t SimpleProfile_RegisterAppCBs( simpleProfileCBs_t *appCallback
s )
{
  if ( appCallbacks )
  {
    simpleProfile_AppCBs = appCallbacks;

    return ( SUCCESS );
  }
  else
  {
    return ( bleAlreadyInRequestedMode );
  }
}
```

回调函数如下：

```
// Callback when a characteristic value has changed
```

```
    typedef void (*simpleProfileChange_t)( uint8 paramID );

    typedef struct
    {
      simpleProfileChange_t         pfnSimpleProfileChange;  // Called wh
en characteristic value changes
    } simpleProfileCBs_t;
```

回调函数必须指向此类型的应用程序，如下：

```
    // Simple GATT Profile Callbacks
    static simpleProfileCBs_t Peripheral_SimpleProfileCBs =
    {
      simpleProfileChangeCB    // Charactersitic value change callback
    };
```

5.5.3.2.4 读写回调函数

当配置文件被读写时，需要有相应的回调函数，其注册方法与应用程序回调函数一致，具体可参考 Peripheral 工程。

5.5.3.2.5 配置文件的获取与设置

配置文件包含读取与写入特征功能，图 5.9 描述了应用程序设置配置文件参数的逻辑。

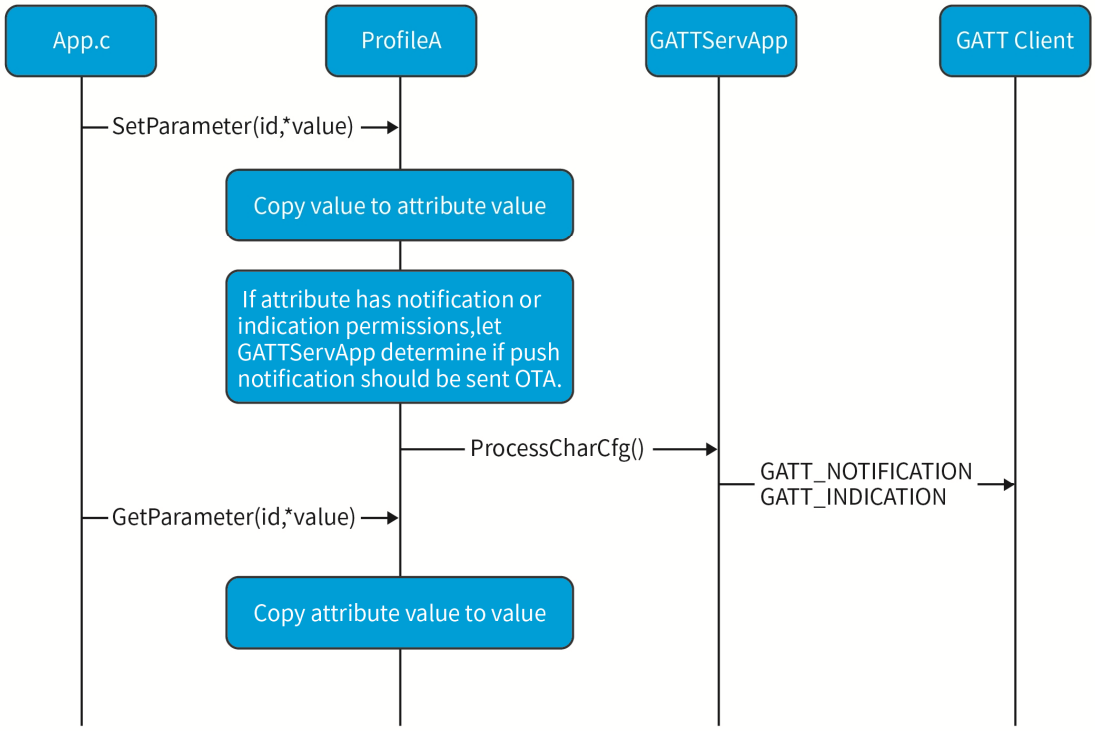


图 5.9 获取和设置配置文件参数

应用程序代码如下：

```
     SimpleProfile_SetParameter( SIMPLEPROFILE_CHAR1, SIMPLEPROFILE_CHAR
1_LEN, charValue1 );
```

5.6 逻辑链路控制和适配协议

逻辑链路控制和适配协议层（L2CAP）位于 HCI 层的顶部，并在主机的上层（GAP 层，GATT 层，应用层等）与下层协议栈之间传输数据。这个上层具有 L2CAP 的分割和重组功能，使更高层次的协议和应用能够以 64KB 的长度发送和接收数据包。它还能够处理协议的多路复用，以提供多种连接和多个连接类型(通过一个空中接口)，同时提供服务质量支持和成组通讯。CH58x 蓝牙协议栈支持有效最大 MTU 为 247。

5.7 主机与控制器交互

HCI（HostControllerInterface），它连接主机和控制器，将主机的操作转化成 HCI 指令传给控制器。BLE CoreSpec 支持的 HCI 种类有四种：UART，USB，SDIO，3-Wire UART。

对于全协议栈的单颗蓝牙芯片，只需要调用 API 接口函数即可，此时 HCI 为函数调用以及回调；对于只有控制器的产品，即使用主控芯片对 BLE 芯片进行操作，BLE 芯片作为外挂芯片接到主控芯片上。此时主控芯片只需要通过标准的 HCI 指令（通常是 UART）与 BLE 芯片进行交互。

本指南所讨论的 HCI 均是函数调用与函数回调。

6. 创建一个 BLE 应用程序

6.1 概述

阅读了前面的章节后，您应该了解如何实现低功耗蓝牙应用程序。这一章将介绍如何开始编写低功耗蓝牙应用程序，以及一些注意事项。

6.2 配置蓝牙协议栈

首先需要确定此设备的角色，我们提供以下几种角色：

- Central
- Peripheral
- Broadcaster
- Observer

选择不同的角色需要调用不同的角色初始化 API，详细参考 5.3 节。

6.3 定义低功耗蓝牙行为

使用低功耗蓝牙协议栈的 API 定义系统行为，如添加配置文件，添加 GATT 数据库，配置安全模式等等。详见第五章。

6.4 定义应用程序任务

确保应用程序中包含对协议栈的回调函数以及来自 TMOS 的事件处理程序。可参考第三章中所介绍的添加其他任务。

6.5 应用配置文件

在 config.h 文件中配置 DCDC 使能，RTC 的时钟，睡眠功能，MAC 地址，低功耗蓝牙协议栈使用 RAM 大小等等，详见 config.h 文件。

需要注意的是，WAKE_UP_RTC_MAX_TIME 即等待 32M 晶体稳定的时间。此稳定时间与晶体，电压，稳定等因素影响。您需要在唤醒时间上添加一个缓冲的时间，以提高稳定性。

6.6 在低功耗蓝牙工作期间限制应用程序处理

由于低功耗蓝牙协议的时间依赖性，控制器必须在每个连接事件或广播事件来临之前就进行处理。如果未及时处理，则会导致重传或者连接断开。且 TOMS 不是多线程的，所以当低功耗蓝牙有事务处理时，必须停止其他任务让控制器处理。所以确保应用程序中不要占用大量的事件，如需复杂的处理，请参考 3.3 节将其拆分。

6.7 中断

在低功耗蓝牙工作期间，需要通过 RTC 定时器计算时间，所以在此期间，不要禁止全局中断，且单个中断服务程序所占用的时间不宜过长，否则长期打断低功耗蓝牙工作会导致连接断开。

7. 创建一个简单的 RF 应用程序

7.1 概述

RF 应用程序是基于 RF 的发送与接收的 PHY，实现 2.4GHz 频段的无线通讯。与 BLE 的区别在于，RF 应用程序并没有建立 BLE 的协议。

7.2 配置协议栈

首先初始化蓝牙库：

```
CH58X_BLEInit( );
```

然后配置此设备的角色为 RF Role：

```
RF_RoleInit( );
```

7.3 定义应用程序任务

注册 RF 任务，初始化 RF 功能，以及注册 RF 的回调函数：

```
    taskID = TMOS_ProcessEventRegister( RF_ProcessEvent );
    rfConfig.accessAddress = 0x71764129;  // 禁止使用 0x55555555 以及
0xAAAAAAAA ( 建议不超过 24 次位反转，且不超过连续的 6 个 0 或 1 )
    rfConfig.CRCInit = 0x555555;
  rfConfig.Channel = 8;
    rfConfig.Frequency = 2480000;
    rfConfig.LLEMode = LLE_MODE_BASIC|LLE_MODE_EX_CHANNEL|LLE_MODE_NON_R
SSI; // 使能 LLE_MODE_EX_CHANNEL 表示选择 rfConfig.Frequency 作为通信频点
    rfConfig.rfStatusCB = RF_2G4StatusCallBack;
    state = RF_Config( &rfConfig );
```

7.4 应用配置文件

在 config.h 文件中配置 DCDC 使能，RTC 的时钟，睡眠功能，MAC 地址，低功耗蓝牙协议栈使用 RAM 大小等等，详见 config.h 文件。

需要注意的是，WAKE_UP_RTC_MAX_TIME 即等待 32M 晶体稳定的时间。此稳定时间与晶体，电压，稳定等因素影响。您需要在唤醒时间上添加一个缓冲的时间，以提高稳定性。

7.5 RF 通信

7.5.1 Basic 模式

在 Basic 模式只需保持接收方一直处于接收模式，即调用 RF_RX()函数。但是需要注意的是，在接收到数据后，需要再次调用 RF_RX()函数使设备再次处于接收模式，且不要直接在 RF_2G4StatusCallBack()回调函数中调用 RF 收发函数，这样可能会造成其状态混乱。

通信示意图如下：

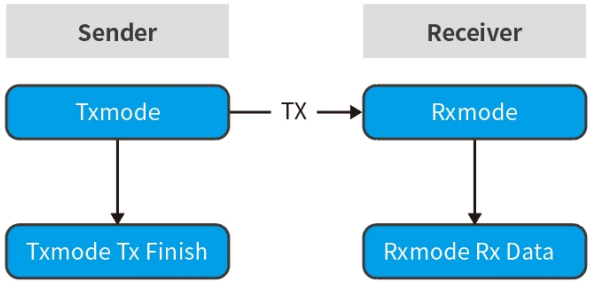


图 7.1 Basic 模式通信示意图

RF 发送的 API 为 RF_Tx(),详情请参照 8.6 节。

RF 接收到数据则会进入调函数 RF_2G4StatusCallBack()，在回调函数中获取接收到的数据。

7.5.2 Auto 模式

由于 Basic 模式仅为单向传输，用户便无法得知此次通信是否成功，由此产生 Auto 模式。

Auto 模式在 Basic 模式的基础上，增加了接收响应的机制，即接收方在接收到数据后，会向发送方发送数据，以通知发送方已成功接收数据。

通信示意图如下：

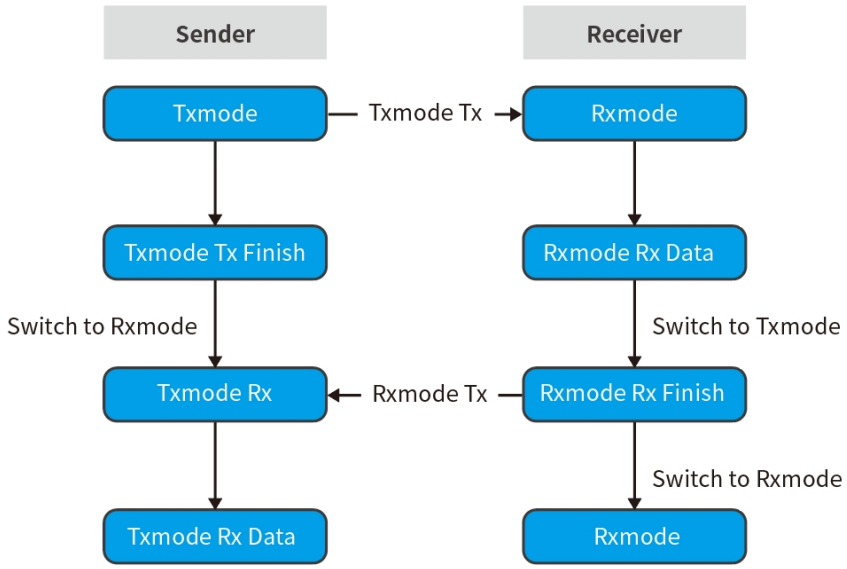


图 7.2 Auto 模式通信示意图

Auto 模式下，RF 发送完数据后会自动切换至接收模式，此接收模式超时时间为 3ms，若 3ms 内未接收到数据则会关闭接收模式。接收到的数据，以及超时状态均通过回调函数返回至应用层。

8. API

8.1 GAP API

8.1.1 指令

```
bStatus_t GAP_SetParamValue( uint16 paramID, u16 paramValue )
```

参数	描述
paramID	参数的 ID,参考 8.1.2
paramValue	新的参数值
返回	SUCCESS 或 INVALIDPARAMETER（无效参数 ID）

```
uint16 GAP_GetParamValue( uint16 paramID )
```

参数	描述
paramID	参数的 ID,参考 8.1.2
返回	GAP 的参数值若参数 ID 无效返回 0xFFFF

8.1.2 配置参数

以下为常用的参数 ID，详细的参数 ID 请参考 CH58xBLE.LIB.h。

参数 ID	描述
TGAP_GEN_DISC_ADV_MIN	通用广播模式的广播时长，单位:0.625ms（默认值：0）
TGAP_LIM_ADV_TIMEOUT	限时可发现广播模式的广播时长，单位:1s（默认值：180）
TGAP_DISC_ADV_INT_MIN	最小广播间隔，单位:0.625ms（默认值：160）
TGAP_DISC_ADV_INT_MAX	最大广播间隔，单位:0.625ms（默认值：160）
TGAP_DISC_SCAN	扫描时长，单位:0.625ms（默认值：16384）
TGAP_DISC_SCAN_INT	扫描间隔，单位:0.625ms（默认值：16）
TGAP_DISC_SCAN_WIND	扫描窗口，单位:0.625ms（默认值：16）
TGAP_CONN_EST_SCAN_INT	建立连接的扫描间隔，单位:0.625ms（默认值：16）
TGAP_CONN_EST_SCAN_WIND	建立连接的扫描窗口，单位:0.625ms（默认值：16）
TGAP_CONN_EST_INT_MIN	建立连接的最小连接间隔，单位:1.25ms（默认值：80）
TGAP_CONN_EST_INT_MAX	建立连接的最大连接间隔，单位:1.25ms（默认值：80）
TGAP_CONN_EST_SUPERV_TIMEOUT	建立连接的连接管理超时时间，单位:10ms（默认值：2000）
TGAP_CONN_EST_LATENCY	建立连接的从设备延迟（默认值：0）

8.1.3 事件

本节介绍了 GAP 层相关的事件，可以在 CH58xBLE_LIB.h 文件中找到相关声明。其中一些事件是直接传递给应用程序，一些是由 GAPRole 和 GAPBondMgr 处理。

无论是传递给哪一层，它们都将作为带标头的 GAP_MSG_EVENT 传递：

```
    typedef struct
    {
      tmos_event_hdr_t  hdr;           //!< GAP_MSG_EVENT and status
     uint8 opcode;                    //!< GAP type of command. Ref: @re
f GAP_MSG_EVENT_DEFINES
    } gapEventHdr_t;
```

以下为常用事件名称以及事件传递消息的格式。详细请参考 CH58xBLE_LIB.h。

·GAP_DEVICE_INIT_DONE_EVENT：当设备初始化完成置此事件。

```
    typedef struct
    {
      tmos_event_hdr_t  hdr;              //!< GAP_MSG_EVENT and status
      uint8 opcode;                       //!< GAP_DEVICE_INIT_DONE_EVEN
    T
      uint8 devAddr[B_ADDR_LEN];          //!< Device's BD_ADDR
      uint16 dataPktLen;                  //!< HC_LE_Data_Packet_Length
      uint8 numDataPkts;                  //!< HC_Total_Num_LE_Data_Pack
    ets
    } gapDeviceInitDoneEvent_t;
```

·GAP_DEVICE_DISCOVERY_EVENT：设备发现过程完成时置此事件。

```
    typedef struct
    {
        tmos_event_hdr_t  hdr; //!< GAP_MSG_EVENT and status
        uint8 opcode;          //!< GAP_DEVICE_DISCOVERY_EVENT
        uint8 numDevs;         //!< Number of devices found during scan
        gapDevRec_t *pDevList; //!< array of device records
    } gapDevDiscEvent_t;
```

·GAP_END_DISCOVERABLE_DONE_EVENT：当广播结束时置此事件。

```
    typedef struct
    {
       tmos_event_hdr_t  hdr; //!< GAP_MSG_EVENT and status
     uint8 opcode;          //!< GAP_END_DISCOVERABLE_DONE_EVENT
    } gapEndDiscoverableRspEvent_t;
```

·GAP_LINK_ESTABLISHED_EVENT：建立连接后置此事件。

```
    typedef struct
    {
      tmos_event_hdr_t  hdr;     //!< GAP_MSG_EVENT and status
      uint8 opcode;              //!< GAP_LINK_ESTABLISHED_EVENT
      uint8 devAddrType;         //!< Device address type: @ref GAP_ADDR
_TYPE_DEFINES
      uint8 devAddr[B_ADDR_LEN]; //!< Device address of link
```

```
    uint16 connectionHandle;   //!< Connection Handle from controller
used to ref the device
    uint8 connRole;            //!< Connection formed as Master or Sla
ve
    uint16 connInterval;       //!< Connection Interval
    uint16 connLatency;        //!< Connection Latency
    uint16 connTimeout;        //!< Connection Timeout
    uint8 clockAccuracy;       //!< Clock Accuracy
  } gapEstLinkReqEvent_t;
```

- GAP_LINK_TERMINATED_EVENT：连接断开后置此事件。

```
  typedef struct
  {
    tmos_event_hdr_t  hdr;   //!< GAP_MSG_EVENT and status
    uint8 opcode;            //!< GAP_LINK_TERMINATED_EVENT
    uint16 connectionHandle; //!< connection Handle
    uint8 reason;            //!< termination reason from LL
    uint8 connRole;
  } gapTerminateLinkEvent_t;
```

- GAP_LINK_PARAM_UPDATE_EVENT：接收到参数更新事件后置此事件。

```
  typedef struct
  {
    tmos_event_hdr_t hdr;     //!< GAP_MSG_EVENT and status
    uint8 opcode;             //!< GAP_LINK_PARAM_UPDATE_EVENT
    uint8 status;             //!< bStatus_t
    uint16 connectionHandle;  //!< Connection handle of the update
    uint16 connInterval;      //!< Requested connection interval
    uint16 connLatency;       //!< Requested connection latency
    uint16 connTimeout;       //!< Requested connection timeout
  } gapLinkUpdateEvent_t;
```

- GAP_DEVICE_INFO_EVENT：在发现设备期间发现设备置此事件。

```
  typedef struct
  {
    tmos_event_hdr_t  hdr;    //!< GAP_MSG_EVENT and status
    uint8 opcode;             //!< GAP_DEVICE_INFO_EVENT
    uint8 eventType;          //!< Advertisement Type: @ref GAP_ADVERT
ISEMENT_REPORT_TYPE_DEFINES
    uint8 addrType;           //!< address type: @ref GAP_ADDR_TYPE_DE
FINES
    uint8 addr[B_ADDR_LEN];   //!< Address of the advertisement or SCA
N_RSP
    int8  rssi;               //!< Advertisement or SCAN_RSP RSSI
```

```
    uint8 dataLen;            //!< Length (in bytes) of the data field
 (evtData)
    uint8 *pEvtData;          //!< Data field of advertisement or SCAN
_RSP
} gapDeviceInfoEvent_t;
```

8.2 GAPRole API

8.2.1 GAPRoleCommon Role API

8.2.1.1 指令

```
bStatus_t GAPRole_SetParameter( uint16 param, uint8 len, void *pValu
e )
```

设置 GAP 参数。

参数	描述
param	配置参数 ID，详见 8.2.1.2 节
len	写入的数据长度
pValue	指向设置参数值的指针。该指针取决于参数 ID，并将被强制转换成合适的数据类型。
返回	SUCCESS INVALIDPARAMETER：参数无效 bleInvalidRange：参数长度无效 blePending：上次参数更新未结束 bleIncorrectMode：模式错误

```
bStatus_t GAPRole_GetParameter( uint16 param, void *pValue )
```

获取 GAP 参数。

参数	描述
param	配置参数 ID，详见 8.2.1.2 节
pValue	指向获取参数的位置的指针。该指针取决于参数 ID，并将被强制转换成合适的数据类型。
返回	SUCCESS INVALIDPARAMETER：参数无效

```
bStatus_t GAPRole_TerminateLink( uint16 connHandle )
```

断开当前 connHandle 指定的连接。

参数	描述
connHandle	连接句柄
返回	SUCCESS bleIncorrectMode：模式错误

```
bStatus_t GAPRole_ReadRssiCmd( uint16 connHandle )
```

读取当前 connHandle 指定连接的 RSSI 值。

参数	描述
connHandle	连接句柄
返回	SUCCESS 0x02:无有效连接

8.2.1.2 常用可配置参数

参数	读/写	大小	描述
GAPROLE_BD_ADDR	只读	uint8	设备地址
GAPRPLE_ADVERT_ENABLE	可读可写	uint8	使能或关闭广播，默认使能
GAPROLE_ADVERT_DATA	可读可写	≤240	广播数据，默认全 0。
GAPROLE_SCAN_RSP_DATA	可读可写	≤240	扫描应答数据，默认全 0
GAPROLE_ADV_EVENT_TYPE	可读可写	uint8	广播类型，默认可连接非定向广播
GAPROLE_MIN_CONN_INTERVAL	可读可写	uint16	最小连接间隔，范围：1.5ms~4s，默认 8.5ms。
GAPROLE_MAX_CONN_INTERVAL	可读可写	uint16	最大连接间隔，范围：1.5ms~4s，默认 8.5ms。

8.2.1.3 回调函数

```
/**
 * Callback when the device has read an new RSSI value during a connect
ion.
 */
typedef void (*gapRolesRssiRead_t)(uint16 connHandle, int8 newRSSI )
```

此函数为读取 RSSI 的回调函数，其指针从指向应用程序，以便 GAPRole 可以将事件返回给应用程序。传递方式如下：

```
// GAP Role Callbacks
static gapCentralRoleCB_t centralRoleCB =
{
  centralRssiCB,       // RSSI callback
  centralEventCB,       // Event callback
  centralHciMTUChangeCB       // MTU change callback
};
```

8.2.2 GAPRolePeripheral Role API

8.2.2.1 指令

```
bStatus_t GAPRole_PeripheralInit( void )
```

蓝牙从机 GAPRole 任务初始化。

参数	描述
返回	SUCCESS bleInvalidRange：参数超出范围

```
bStatus_t GAPRole_PeripheralStartDevice( uint8 taskid, gapBondCBs_t
*pCB,gapRolesCBs_t *pAppCallbacks )
```

蓝牙从机设备初始化。

参数	描述
taskid	tmos 分配的任务 ID
pCB	绑定回调函数，包括密钥回调，配对状态回调
pAppCallbacks	GAPRole 回调函数，包括设备的状态回调，RSSI 回调，参数更新回调
返回	SUCCESS bleAlreadyInRequestedMode:设备已经初始化过

```
bStatus_t GAPRole_PeripheralConnParamUpdateReq( uint16 connHandle,
                                                uint16 minConnInterval,
                                                uint16 maxConnInterval,
                                                uint16 latency,
                                                uint16 connTimeout,
                                                uint8 taskId)
```

蓝牙从机连接参数更新。

注与 GAPRole_UpdateLink()不同，此为从机与主机协商连接参数，而 GAPRole_UpdateLink()是主句直接配置连接参数。

参数	描述
connHandle	连接句柄
minConnInterval	最小连接间隔
maxConnInterval	最大连接间隔
latency	从设备延迟事件数
connTimeout	连接超时
taskID	toms 分配的任务 ID
返回	SUCCESS：参数上传成功 BleNotConnected：无连接所以参数无法更新 bleInvalidRange：参数错误

8.2.2.2 回调函数

```
// GAP Role Callbacks
static gapRolesCBs_t Peripheral_PeripheralCBs =
{
  peripheralStateNotificationCB,  // Profile State Change Callbacks
  peripheralRssiCB,                    // When a valid RSSI is read f
rom controller (not used by application)
  peripheralParamUpdateCB
};
```

从机状态回调函数：

```
/**
* Callback when the device has been started.  Callback event to
* the Notify of a state change.
 */
```

```
void (*gapRolesStateNotify_t)( gapRole_States_t newState ,gapRoleEvent_t * pEvent);
```

其中，状态分为以下几种：

- GAPROLE_INIT //等待启动
- GAPROLE_STARTED //初始化完成但是未未广播
- GAPROLE_ADVERTISING //正在广播
- GAPROLE_WAITING //设备启动了但是未广播，此时正在等待再次广播
- GAPROLE_CONNECTED //连接状态
- GAPROLE_CONNECTED_ADV //连接状态且在广播
- GAPROLE_ERROR//无效状态，若为此状态表明错误

从机参数更新回调函数：

```
/**
* Callback when the connection parameteres are updated.
*/
typedef void (*gapRolesParamUpdateCB_t)( uint16 connHandle,
                                         uint16 connInterval,
                                         uint16 connSlaveLatency,
                                          uint16 connTimeout );
```

参数更新成功调用此回调函数。

8.2.3 GAPRole Central Role API

8.2.3.1 指令

```
bStatus_t GAPRole_CentralInit( void )
```

主机 GAPRole 任务初始化。

参数	描述
返回	SUCCESS bleInvalidRange：参数超出范围

```
bStatus_t GAPRole_CentralStartDevice( uint8 taskid, gapBondCBs_t *pCB, gapCentralRoleCB_t *pAppCallbacks )
```

主机设备初始化。

参数	描述
taskid	tmos 分配的任务 ID
pCB	绑定回调函数，包括密钥回调，配对状态回调
pAppCallbacks	GAPRole 回调函数，包括设备的状态回调，RSSI 回调，参数更新回调
返回	SUCCESS bleAlreadyInRequestedMode:设备已经初始化过

```
bStatus_t GAPRole_CentralStartDiscovery( uint8 mode, uint8 activeScan, uint8 whiteList )
```

主机扫描参数配置。

参数	描述

mode	扫描模式，分为： DEVDISC_MODE_NONDISCOVERABLE：不作设置 DEVDISC_MODE_GENERAL：扫描通用可发现设备 DEVDISC_MODE_LIMITED：扫描有限的可发现设备 DEVDISC_MODE_ALL：扫描所有
activeScan	TRUE 为使能扫描
whiteList	TRUE 为只扫描白名单设备
返回	SUCCESS

```
bStatus_t GAPRole_CentralCancelDiscovery( void )
```

主机停止扫描。

参数	描述
返回	SUCCESS bleInvalidTaskID：没有任务正在扫描 bleIncorrectMode：不在扫描模式

```
bStatus_t GAPRole_CentralEstablishLink( uint8 highDutyCycle, uint8 whiteList, uint8 addrTypePeer, uint8 *peerAddr )
```

与对端设备进行连接。

参数	描述
highDutyCycle	TURE 使能高占空比扫描
whiteList	TURE 使用白名单
addrTypePeer	对端设备的地址类型，包括： ADDRTYPE_PUBLIC：BD_ADDR ADDRTYPE_STATIC：静态地址 ADDRTYPE_PRIVATE_NONRESOLVE：不可解析私有地址 ADDRTYPE_PRIVATE_RESOLVE：可解析的私有地址
peerAddr	对端设备地址
返回	SUCCESS：成功连接 bleIncorrectMode：无效的配置文件 bleNotReady：正在进行扫描 bleAlreadyInRequestedMode：暂时无法处理 bleNoResources：连接过多

8.2.3.2 回调函数

```
// GAP Role Callbacks
static gapCentralRoleCB_t centralRoleCB =
{
  centralRssiCB,       // RSSI callback
  centralEventCB,       // Event callback
  centralHciMTUChangeCB       // MTU change callback
};
```

主机 RSSI 回调函数：

```
/**
 * Callback when the device has read an new RSSI value during a connect
ion.
 */
typedef void (*gapRolesRssiRead_t)(uint16 connHandle, int8 newRSSI )
```

主机事件回调函数：

```
/**
* Central Event Callback Function
*/
typedef void (*pfnGapCentralRoleEventCB_t) (  gapRoleEvent_t *pEvent
 ); //!< Pointer to event structure.
```

回调事件可参考 8.1.3 节。

MTU 交互回调函数：

```
typedef void (*pfnHciDataLenChangeEvCB_t)
(
  uint16 connHandle,
  uint16 maxTxOctets,
  uint16 maxRxOctets
);
```

即与低功耗蓝牙交互的数据包大小。

8.3 GATT API

8.3.1 指令

8.3.1.1 从机指令

```
bStatus_t GATT_Indication( uint16 connHandle, attHandleValueInd_t *p
Ind, uint8 authenticated, uint8 taskId )
```

服务器向客户端指示一个特征值并期望属性协议层确认已经成功收到指示。

需要注意的是，当返回失败时需要释放内存。

参数	描述
connHandle	连接句柄
pInd	指向要发送的指令
authenticated	是否需要经过身份验证的连接
taskId	tmos 分配的任务 ID

```
bStatus_t GATT_Notification( uint16 connHandle, attHandleValueNoti_t
 *pNoti, uint8 authenticated )
```

服务器向客户端通知特征值，但不期望任何属性协议层确认已经成功收到通知。

需要注意的是，当返回失败时需要释放内存。

参数	描述
connHandle	连接句柄
pInd	指向要通知的指令
authenticated	是否需要经过身份验证的连接

8.3.1.2 主机指令

```
bStatus_t GATT_ExchangeMTU( uint16 connHandle, attExchangeMTUReq_t *pReq, uint8 taskId )
```

当客户端支持的值大于属性协议默认 ATT_MTU 的值时，客户端使用此程序将 ATT_MTU 设置为两个设备均可支持的最大可能值。

参数	描述
connHandle	连接句柄
pReq	指向要发送的指令
taskID	通知的任务的 ID

```
bStatus_t GATT_DiscAllPrimaryServices( uint16 connHandle, uint8 taskId )
```

发现服务器上的所有主服务。

参数	描述
connHandle	连接句柄
taskID	通知的任务的 ID

```
bStatus_t GATT_DiscPrimaryServiceByUUID( uint16 connHandle, uint8 *pUUID, uint8 len, uint8 taskId )
```

当仅知道 UUID 时，客户端可以通过此函数发现服务器上的主服务。由于主服务在服务器上可能存在多个，所以被发现的主服务有 UUID 标识。

参数	描述
connHandle	连接句柄
pUUID	指向要查找的服务器的 UUID 的指针
len	值的长度
taskID	通知的任务的 ID

```
bStatus_t GATT_FindIncludedServices( uint16 connHandle, uint16 startHandle, uint16 endHandle, uint8 taskId )
```

客户端使用此函数在服务器上查找次服务。查找的服务由服务句柄范围标识。

参数	描述
connHandle	连接句柄
startHandle	起始句柄
endHandle	结束句柄
taskID	通知的任务的 ID

```
bStatus_t GATT_DiscAllChars( uint16 connHandle, uint16 startHandle,
uint16 endHandle, uint8 taskId )
```

当仅知道服务句柄范围时，客户端可使用此函数在服务器上查找所有特征声名。

参数	描述
connHandle	连接句柄
startHandle	起始句柄
endHandle	结束句柄
taskID	通知的任务的 ID

```
bStatus_t GATT_DiscCharsByUUID( uint16 connHandle, attReadByTypeReq_
t *pReq, uint8 taskId )
```

当服务句柄范围和特征 UUID 已知时，客户端可使用此函数在服务器上发现特征。

参数	描述
connHandle	连接句柄
pReq	指向要发送的请求的指针
taskID	通知的任务的 ID

```
bStatus_t GATT_DiscAllCharDescs( uint16 connHandle, uint16 startHand
le, uint16 endHandle, uint8 taskId )
```

当知道特征的句柄范围时，客户端可使用此程序在特征定义中查找所有特征描述符 AttributeHandles 和 AttributeTypes。

参数	描述
connHandle	连接句柄
startHandle	起始句柄
endHandle	结束句柄
taskID	通知的任务的 ID

```
bStatus_t GATT_ReadCharValue( uint16 connHandle, attReadReq_t *pReq,
 uint8 taskId )
```

当客户端知道特征句柄后，可用此函数从服务器读取特征值。

参数	描述
connHandle	连接句柄
pReq	指向要发送的请求的指针
taskID	通知的任务的 ID

```
bStatus_t GATT_ReadUsingCharUUID( uint16 connHandle, attReadByTypeRe
q_t *pReq, uint8 taskId )
```

当客户端只知道特征的 UUID 而不知道特征的句柄的时候，可使用此函数从服务器读取特征值。

参数	描述
connHandle	连接句柄
pReq	指向要发送的请求的指针
taskID	通知的任务的 ID

```
bStatus_t GATT_ReadLongCharValue( uint16 connHandle, attReadBlobReq_t *pReq, uint8 taskId )
```

读取服务器特征值，但特征值比单个读取响应协议中可发送的长度长。

参数	描述
connHandle	连接句柄
pReq	指向要发送的请求的指针
taskID	通知的任务的 ID

```
bStatus_t GATT_ReadMultiCharValues( uint16 connHandle, attReadMultiReq_t *pReq, uint8 taskId )
```

从服务器读取多个特征值。

参数	描述
connHandle	连接句柄
pReq	指向要发送的请求的指针
taskID	通知的任务的 ID

```
bStatus_t GATT_WriteNoRsp( uint16 connHandle, attWriteReq_t *pReq )
```

当客户端已知特征句柄可向服务器写入特征而不需要确认写入是否成功。

参数	描述
connHandle	连接句柄
pReq	指向要发送的命令的指针

```
bStatus_t GATT_SignedWriteNoRsp( uint16 connHandle, attWriteReq_t *pReq )
```

当客户端知道特征句柄且 ATT 确认未加密时，可用此函数向服务器写入特征值。仅当 Characteristic Properties 认证位使能且服务器和客户端均建立绑定。

参数	描述
connHandle	连接句柄
pReq	指向要发送的命令的指针

```
bStatus_t GATT_WriteCharValue( uint16 connHandle, attWriteReq_t *pReq, uint8 taskId )
```

当客户端知道特征句柄，此函数可将特征值写入服务器。仅能写入特征值的第一个八位数据。此函数会返回写过程是否成功。

参数	描述
connHandle	连接句柄
pReq	指向要发送的请求的指针
taskID	通知的任务的 ID

```
bStatus_t GATT_WriteLongCharDesc( uint16 connHandle, attPrepareWriteReq_t *pReq, uint8 taskId )
```

当客户端知道特征值句柄但是特征值长度大于单个写入请求属性协议中定义的长度时，可使用此函数。

参数	描述
connHandle	连接句柄
pReq	指向要发送的请求的指针
taskID	通知的任务的 ID

8.3.2 返回

- SUCCESS（0x00）：指令按预期执行。
- INVALIDPARAMETER（0x02）：无效的连接句柄或请求字段。
- MSG_BUFFER_NOT_AVAIL（0x04）：HCI 缓冲区不可用。请稍后重试。
- bleNotConnected（0x14）：设备未连接。
- blePending（0x17）：
  - 当返回到客户端功能时，服务器或 GATT 子过程正在进行中，有待处理的响应。
  - 返回服务器功能时，来自客户端的确认待处理。
- bleTimeout（0x16）：上一个事务超时。重新连接之前，无法发送 ATT 或 GATT 消息。
- bleMemAllocError（0x13）：发生内存分配错误
- bleLinkEncrypted（0x19）：链接已加密。不要在加密的链接上发送包含身份验证签名的 PDU。

8.3.3 事件

应用程序通过 TMOS 的消息（GATT_MSG_EVENT）接收协议栈的事件。

以下为常用事件名称以及事件传递消息的格式。详细请参考 CH58xBLE_LIB.h。

- ATT_ERROR_RSP：

```
typedef struct
{
    uint8 reqOpcode; //!< Request that generated this error response
    uint16 handle;   //!< Attribute handle that generated error response
  uint8 errCode;   //!< Reason why the request has generated error response
} attErrorRsp_t;
```

- ATT_EXCHANGE_MTU_REQ：

```
typedef struct
{
    uint8 flags; //!< 0x00 - cancel all prepared writes.
                 //!< 0x01 - immediately write all pending prepared values.
} attExecuteWriteReq_t;
```

- ATT_EXCHANGE_MTU_RSP：

```
typedef struct
{
    uint16 serverRxMTU; //!< Server receive MTU size
```

```
} attExchangeMTURsp_t;
```

- ATT_READ_REQ:

```
typedef struct
{
  uint16 handle; //!< Handle of the attribute to be read (must be fi
rst field)
} attReadReq_t;
```

- ATT_READ_RSP:

```
typedef struct
{
    uint16 len;    //!< Length of value
  uint8 *pValue; //!< Value of the attribute with the handle given (
0 to ATT_MTU_SIZE-1)
} attReadRsp_t;
```

- ATT_WRITE_REQ:

```
typedef struct
{
  uint16 handle; //!< Handle of the attribute to be written (must be
 first field)
  uint16 len;    //!< Length of value
  uint8 *pValue; //!< Value of the attribute to be written (0 to ATT
_MTU_SIZE-3)
  uint8 sig;     //!< Authentication Signature status (not included
(0), valid (1), invalid (2))
  uint8 cmd;     //!< Command Flag
} attWriteReq_t;
```

- ATT_WRITE_RSP:
- ATT_HANDLE_VALUE_NOTI:

```
typedef struct
{
  uint16 handle; //!< Handle of the attribute that has been changed
(must be first field)
  uint16 len;    //!< Length of value
  uint8 *pValue; //!< Current value of the attribute (0 to ATT_MTU_S
IZE-3)
} attHandleValueNoti_t;
```

- ATT_HANDLE_VALUE_IND:

```
typedef struct
{
```

```
  uint16 handle; //!< Handle of the attribute that has been changed
(must be first field)
  uint16 len;    //!< Length of value
  uint8 *pValue; //!< Current value of the attribute (0 to ATT_MTU_S
IZE-3)
} attHandleValueInd_t;
```

- ATT_HANDLE_VALUE_CFM:
  - Empty msg field

8.3.4 GATT 指令与相应的 ATT 事件

ATT 响应事件	GATT API 调用
ATT_EXCHANGE_MTU_RSP	GATT_ExchangeMTU
ATT_FIND_INFO_RSP	GATT_DiscAllCharDescs GATT_DiscAllCharDescs
ATT_FIND_BY_TYPE_VALUE_RSP	GATT_DiscPrimaryServiceByUUID
ATT_READ_BY_TYPE_RSP	GATT_PrepareWriteReq GATT_ExecuteWriteReq GATT_FindIncludedServices GATT_DiscAllChars GATT_DiscCharsByUUID GATT_ReadUsingCharUUID
ATT_READ_RSP	GATT_ReadCharValue GATT_ReadCharDesc
ATT_READ_BLOB_RSP	GATT_ReadLongCharValue GATT_ReadLongCharDesc
ATT_READ_MULTI_RSP	GATT_ReadMultiCharValues
ATT_READ_BY_GRP_TYPE_RSP	GATT_DiscAllPrimaryServices
ATT_WRITE_RSP	GATT_WriteCharValue GATT_WriteCharDesc
ATT_PREPARE_WRITE_RSP	GATT_WriteLongCharValue GATT_ReliableWrites GATT_WriteLongCharDesc
ATT_EXECUTE_WRITE_RSP	GATT_WriteLongCharValue GATT_ReliableWrites GATT_WriteLongCharDesc

8.3.5 ATT_ERROR_RSP 错误码

- ATT_ERR_INVALID_HANDLE（0x01）：给定的属性句柄值在此属性服务器上无效。
- ATT_ERR_READ_NOT_PERMITTED（0x02）：无法读取属性。
- ATT_ERR_WRITE_NOT_PERMITTED（0x03）：无法写入属性。
- ATT_ERR_INVALID_PDU（0x04）：属性 PDU 无效。
- ATT_ERR_INSUFFICIENT_AUTHEN(0x05):该属性需要进行身份验证才能被读取或写入。
- ATT_ERR_UNSUPPORTED_REQ（0x06）：属性服务器不支持从属性客户端收到的请求。

- ATT_ERR_INVALID_OFFSET（0x07）：指定的偏移量超出了属性的末尾。
- ATT_ERR_INSUFFICIENT_AUTHOR（0x08）：该属性需要授权才能被读取或写入。
- ATT_ERR_PREPARE_QUEUE_FULL（0x09）：准备写入的队列过多。
- ATT_ERR_ATTR_NOT_FOUND（0x0A）：在给定的属性句柄范围内找不到属性。
- ATT_ERR_ATTR_NOT_LONG（0x0B）：无法使用读取 Blob 请求或准备写入请求来读取或写入属性。
- ATT_ERR_INSUFFICIENT_KEY_SIZE（0x0C）：用于加密此链接的加密密钥大小不足。
- ATT_ERR_INVALID_VALUE_SIZE（0x0D）：属性值长度对该操作无效。
- ATT_ERR_UNLIKELY（0x0E）：请求的属性请求遇到了一个不太可能发生的错误，并且未能按要求完成。
- ATT_ERR_INSUFFICIENT_ENCRYPT（0x0F）：该属性需要加密才能读取或写入。
- ATT_ERR_UNSUPPORTED_GRP_TYPE（0x10）：属性类型不是更高层规范定义的受支持的分组属性。
- ATT_ERR_INSUFFICIENT_RESOURCES（0x11）：资源不足，无法完成请求。

8.4 GATTServApp API

8.4.1 指令

```
void GATTServApp_InitCharCfg( uint16 connHandle, gattCharCfg_t *charCfgTbl )
```

初始化客户端特性配置表。

参数	描述
connHandle	连接句柄
charCfgTbl	客户端特征配置表
返回	无

```
uint16 GATTServApp_ReadCharCfg( uint16 connHandle, gattCharCfg_t *charCfgTbl )
```

读取客户端的特征配置。

参数	描述
connHandle	连接句柄
charCfgTbl	客户端特征配置表
返回	属性值

```
uint8 GATTServApp_WriteCharCfg( uint16 connHandle, gattCharCfg_t *charCfgTbl, uint16 value )
```

向客户端写入特征配置。

参数	描述
connHandle	连接句柄
charCfgTbl	客户端特征配置表
value	新的值
返回	SUCCESS FAILURE

```
bStatus_t GATTServApp_ProcessCCCWriteReq( uint16 connHandle,
                                          gattAttribute_t *pAttr,
                                          uint8 *pValue,
                                          uint16 len,
                                          uint16 offset,
                                          uint16 validCfg );
```

处理客户端特征配置写入请求。

参数	描述
connHandle	连接句柄
pAttr	指向属性的指针
pvalue	指向写入的数据的指针
len	数据长度
offset	写入的第一八位数据的偏移量
validCfg	有效配置
返回	SUCCESS FAILURE

8.5 GAPBondMgr API

8.5.1 指令

```
bStatus_t GAPBondMgr_SetParameter( uint16 param, uint8 len, void *pValue )
```

设置绑定管理的参数。

参数	描述
param	配置参数
len	写入长度
pValue	指向写入数据的指针
返回	SUCCESS INVALIDPARAMETER：无效参数

```
bStatus_t GAPBondMgr_GetParameter( uint16 param, void *pValue )
```

获取绑定管理的参数。

参数	描述
param	配置参数
pValue	指向读出数据的地址
返回	SUCCESS INVALIDPARAMETER：无效参数

```
bStatus_t GAPBondMgr_PasscodeRsp( uint16 connectionHandle, uint8 status, uint32 passcode )
```

响应密码请求。

参数	描述
connectionHandle	连接句柄

status	SUCCESS：密码可用 其他详见 SMP_PAIRING_FAILED_DEFINES
passcode	整数值密码
返回	SUCCESS：绑定记录已找到且更改 bleIncorrectMode：未发现连接

8.5.2 配置参数

常用配置参数如下表所示，详细的参数 ID 请参考 CH58xBLE.LIB.h。

参数 ID	读写	大小	描述
GAPBOND_PERI_PAIRING_MODE	可读 可写	uint8	配对的方式，默认为： GAPBOND_PAIRING_MODE_WAIT_FOR_REQ
GAPBOND_PERI_DEFAULT_PASSCODE		uint32	默认的中间人保护密钥，范围：0-999999，默认为 0。
GAPBOND_PERI_MITM_PROTECTION	可读 可写	uint8	中间人（MITM）保护。默认为 0，关闭中间人保护。
GAPBOND_PERI_IO_CAPABILITIES	可读 可写	uint8	I/O 能力，模默认为： GAPBOND_IO_CAP_DISPLAY_ONLY，即设备仅能现实。
GAPBOND_PERI_BONDING_ENABLED	可读 可写	uint8	如启用，则在配对过程中请求绑定。默认为 0，不请求绑定。

8.6 RF PHY API

8.6.1 指令

```
bStatus_t RF_RoleInit( void )
```

RF 协议栈初始化。

参数	描述
返回	SUCCESS：初始化成功

```
bStatus_t RF_Config( rfConfig_t *pConfig )
```

RF 参数配置。

参数	描述
pConfig	指向配置参数的指针
返回	SUCCESS

```
bStatus_t RF_Rx( u8 *txBuf, u8 txLen, u8 pktRxType, u8 pktTxType )
```

RF 接受数据函数：将 RF PHY 配置为接受状态，接收到数据后需重新配置。

参数	描述
txBuf	自动模式下，指向 RF 收到数据后返回的数据的指针
txLen	自动模式下，RF 收到数据后返回的数据的长度（0-251）
pkRxType	接受的数据包类型（0xFF：接受所有类型数据包）
pkTxType	自动模式下，RF 收到数据后返回的数据的数据包类型
返回	SUCCESS

```
bStatus_t RF_Tx( u8 *txBuf, u8 txLen, u8 pktTxType, u8 pktRxType )
```

RF 发送数据函数。

参数	描述
txBuf	指向 RF 发送数据的指针
txLen	RF 发送数据的数据长度（0-251）
pkTxType	发送的数据包的类型
pkRxType	自动模式下，RF 发送数据后接收数据的数据类型（0xFF：接受所有类型数据包）
返回	SUCCESS

```
bStatus_t RF_Shut( void )
```

关闭 RF，停止发送或接收。

参数	描述
返回	SUCCESS

8.6.2 配置参数

RF 配置参数 rfConfig_t 描述如下：

参数	描述
LLEMode	LLE_MODE_BASIC：Basic 模式，发送或接受结束后进入空闲模式 LLE_MODE_AUTO：Auto 模式，在发送完成后自动切换至接收模式 LLE_MODE_EX_CHANNEL：切换至 Frequency 配置频段 LLE_MODE_NON_RSSI：将接收数据的第一字节设置为包类型
Channel	RF 通信通道（0-39）
Frequency	RF 通信频率（2400000KHz-2483500KHz），建议不使用超过 24 次位翻转且连续的 0 或 1 不超过 6 个
AccessAddress	RF 通信地址
CRCInit	CRC 初始值
RFStatusCB	RF 状态回调函数

8.6.3 回调函数

```
void RF_2G4StatusCallBack( uint8 sta , uint8 crc, uint8 *rxBuf )
```

RF 状态回调函数，发送或接收完成均会进入此回到函数。

参数	描述
sta	RF 收发状态
crc	数据包状态校验，1：数据 CRC 校验错误；2：数据包类型错误；其他数据校验正确
rxBuf	指向接收到的数据的指针
返回	NULL

修订记录

版本	时间	修订内容
V1.0	2021/3/22	版本发布
V1.1	2021/4/19	增加 RF 使用示例及 API
V1.2	2021/5/28	内容勘误，添加 RF 描述
V1.3	2021/7/3	1. 名称调整； 2. 修改前言以及开发平台相关描述； 3. 图 3.1 勘误。

版本声明与免责声明

参考手册中可能包含少量由于疏忽造成的错误。已发现的会定期勘误，并在再版中更新和避免出现此类错误。