快速入门 BlueNRG SDK 固件开发

关键字：BlueNRG SDK， BlueNRG-LP/LPS, 软件架构

1. 前言

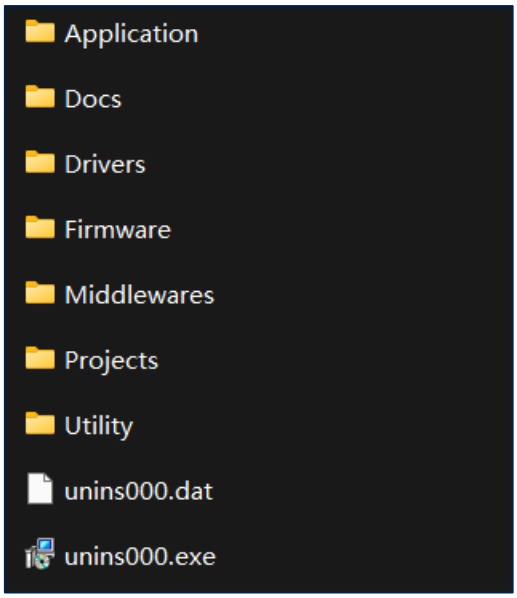
本文档指导用户快速地对 BlueNRG SDK 有一个直观、清晰的认识,了解其软件架构，以便顺利地学会利用 SDK 开发自己的用户固件。

本文档所述 SDK 为 BlueNRG-LP/LPS 芯片的 SDK。阅读本文档前，用户应先了解 BlueNRG-LP/LPS 芯片的一些基本特性，以及其配套开发板的烧录方式。

2. SDK 目录

从 ST 官网下载 SDK 的安装包，成功安装后，即可获得一个 SDK 目录。见图 1：

图1. SDK 目录



各个目录的功能说明见表 1：

表1. SDK 目录说明

文件名	说明
Application	放置一些 PC 应用程序，包括： BLUENRG-X_Wizard，用于配置蓝牙协议栈的参数 BlueNRG-LP_Navigator， 用于一键烧录用户例程 Secure_Bootloader_GUI，用于安全启动
Docs	芯片相关的资料，非常全面！
Drivers	软件驱动层源码
Middlewares	软件中间件层源码
Projects	用户工程，包含软件用户层源码
Firmware	各个例程已经编译好的固件（.hex）
Utility	工具箱目录，包括 Keil pack 等

BlueNRG SDK 安装目录为用户工程师开发 BlueNRG 平台提供了一个便捷的入口，举例来说，有以下几个场景：

1. 硬件工程设计 PCB 前，可通过 Docs 目录找到硬件设计指导文档。完成 PCBA 制作后，可自行使用 Navigator 工具通过串口烧录 Firmware 下的应用固件，验证板子功能。
2. 当工程师想用板子进行功耗、射频测试时，也可在 Firmware 目录下找到合适的已经编译好的固件（测 SOC 蓝牙功耗可用 Beacon，测射频参数可用 DTM）
3. 固件工程师可在 Projects 目录下找到丰富的例程，并且可使用 KEIL、IAR、WiSE 任一个 IDE 打开工程、编译、下载。

3. SDK 例程

SDK Projects 目录包含了以下三类例程：

1. **Periph_Examples**：包含了芯片外设驱动例程。
2. **External_Micro** ：包含了外部单片机的例程，应用于 BlueNRG 芯片在系统中作为协处理器的场景。
3. **BLE_Examples** ：包含了蓝牙相关的所有例程，这些例程的工程特性展示如下：

表2. BLE 例程说明

BLE 例程名	BLE 从机	BLE 主机	OTA/静态协议栈	低功耗	BLE 安全	备注
BLE_ANCS	•			•	•	
BLE_Beacon	•		•	•		协议栈空间占用极少
BLE_Beacon_FlashManagement	•			•		Flash 操作
BLE_Beacon_FreeRTOS	•			•		FreeRTOS
BLE_DirectionFinding						BLE 特性：AOA/AOD
BLE_HID_Peripheral	•			•	•	键盘鼠标
BLE_MultipleConnections	•	•		•	•	多连接
BLE_OTA_ResetManager			•			
BLE_OTA_ServiceManager			•	•		
BLE_PowerControl	•			•		不同蓝牙参数下的功耗
BLE_Power_Consumption	•	•		•		BLE 特性：LE PowerControl
BLE_Privacy	•	•		•	•	BLE 特性：Controller Privacy
BLE_RC_LongRange	•	•		•	•	BLE 特性：Coded PHY
BLE_RemoteControl	•			•	•	
BLE_Security	•			•	•	按键断连，配对方法
BLE_SensorDemo	•		•	•	•	
BLE_SensorDemo_BlueMSapp	•		•	•	•	多传感器
BLE_SensorDemo_Central		•	•	•	•	特定服务发现流程
BLE_SensorDemo_StaticStack			•			
BLE_SerialPort	•	•	•			透传：分别实现主、从
BLE_SerialPort_Master_Slave	•	•				透传：主从一体，先扫描后广播
BLE_StaticStack			•			生成静态协议栈
BLE_Sync	•	•		•		主从时钟精确同步(30us)
BLE_Throughput	•	•				主从吞吐率测试
DTM						支持 ACI/HCI 指令
DTM_basic						
DTM_Updater						

4. 快速实现用户固件功能

本章节指导用户如何快速地在 SDK 中找到相应的 API 接口和位置，以便掌握在 SDK 例程上添加自己的配置和用户逻辑代码的方式。

在对 SDK 提供的例程的功能有所了解之后，假设用户面临的一个开发任务是：

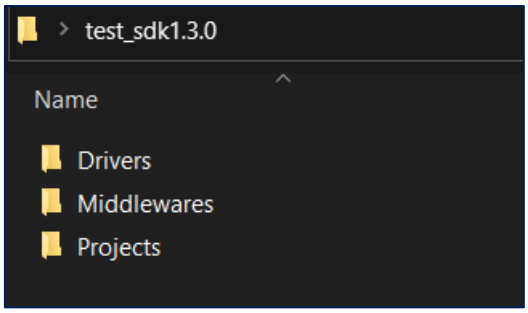
1. BLE 从机功能，包含以下配置：
    a. 一个服务，一个特征，特征具备 Write、Notify 属性
    b. 设备广播名为“Hello”
2. 使用手机 BLE 工具和设备通讯，打印通讯过程产生的数据
3. 自定义协议，实现以下功能：
    a. 控制 LED 亮灭
    b. 定时 1s 上传心跳包，内容为连接后的秒计数值
    c. 每次按键上传按键事件通知

基于以上任务，我们可以选择 BLE_SerialPort 工程作为基础工程并以此来进行固件开发。

4.1. 验证原始工程

在添加用户代码之前，我们最好先验证了原始工程（BLE_SerialPort）的功能，确保开发环境正常。新建一个用户工程目录，比如，test_sdk1.3.0，然后从 SDK 目录拷贝以下文件到我们的用户工程目录，见图 2：

图2. 用户工程目录



打开 Projects > BLE_SerialPort 的 Keil 工程，勾选“Browse Information”选项，以便使能工程内函数的跳转，同时选中 Server 工程配置，见图 3，图 4：

图3. Browse Information

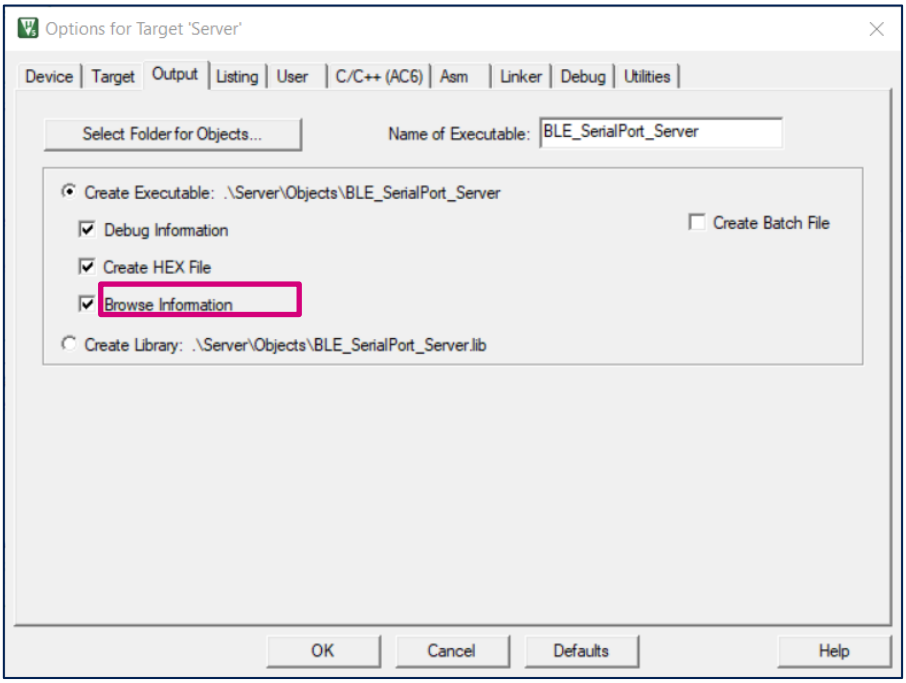
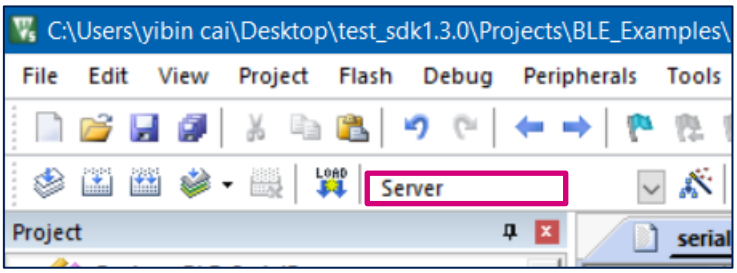
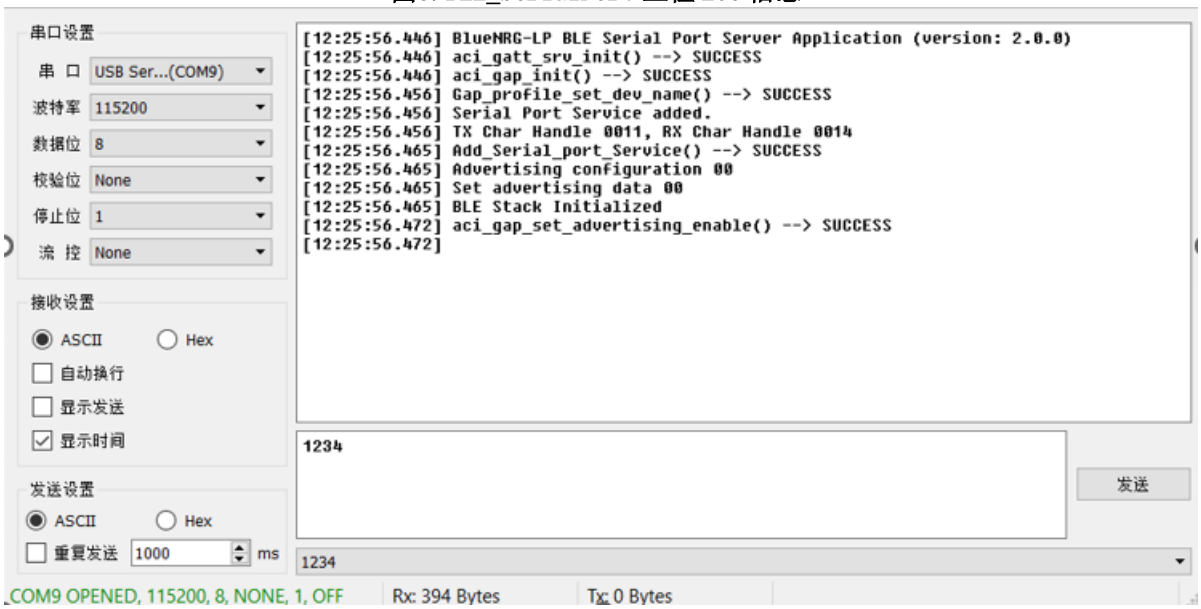


图4. 选中 Server 工程配置



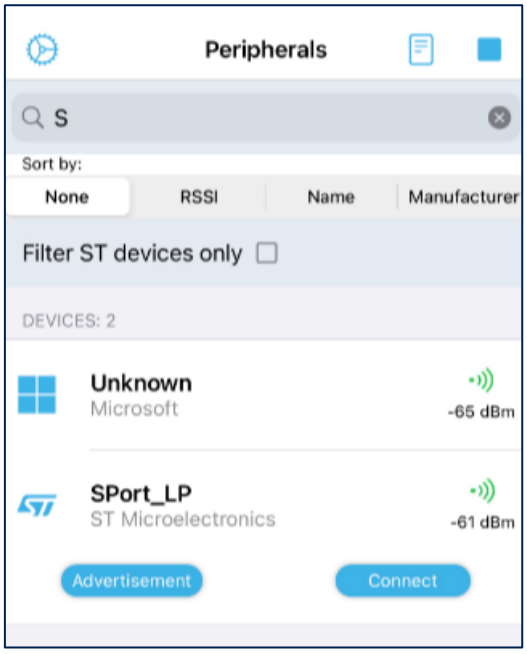
编译、下载工程到开发板，工程运行起来后，应能见到以下打印信息：

图5. BLE_SerialPort 工程 LOG 信息



使用 STBLE Toolbox 工具扫描该设备，应能看到设备名为“Sport_LP”，见图 6：

图6.STBLE Toolbox 扫描页面



至此，原始工程已经正常运行起来了。该工程实现了自定义服务、特征的功能，并能通过串口和手机进行数据的收发。要完成此次开发任务，我们只需在特定位置修改一些代码即可。

4.2. 配置 BLE 从机功能

原始工程有两个特征，一个负责发（TX_CHR_UUID），一个负责收（RX_CHR_UUID）。按照要求，我们需要把他们合并为一个可以收、发的特征。图 6 演示了定义新的特征 UUID 并注释掉旧的两个特征 UUID 的方式：

图7. 特征 UUID

```
#if 1 // test
#define RXTX_CHR_UUID       0x66,0x9a,0x0c,0x20,0x00,0x08,0x96,0x9e,0xe2,0x11,0x9e,0xb1,0xe1,0xf2,0x73,0xd9
#else
#define TX_CHR_UUID         0x66,0x9a,0x0c,0x20,0x00,0x08,0x96,0x9e,0xe2,0x11,0x9e,0xb1,0xe1,0xf2,0x73,0xd9
#define RX_CHR_UUID         0x66,0x9a,0x0c,0x20,0x00,0x08,0x96,0x9e,0xe2,0x11,0x9e,0xb1,0xe2,0xf2,0x73,0xd9
#endif
```

协议栈提供了 ble_gatt_chr_def_t 类型结构体，用户可以定义一个结构体变量并赋值，以此来声明一个特征。图 7 展示了我们所需的特征配置的相关赋值过程，图中可见特征的 UUID 声明，特征 notify、write 属性声明，特征的特征描述符声明等信息。另外，旧的两个特征应该注释掉。

图8. 定义特征

```
static const ble_gatt_chr_def_t serial_port_chars[] = {
#if 1 // test
    {
        .properties = BLE_GATT_SRV_CHAR_PROP_NOTIFY | BLE_GATT_SRV_CHAR_PROP_WRITE_NO_RESP,
        .permissions = BLE_GATT_SRV_PERM_NONE,
        .min_key_size = BLE_GATT_SRV_MAX_ENCRY_KEY_SIZE,
        .uuid = BLE_UUID_INIT_128(RXTX_CHR_UUID),
        .descrs = {
            .descrs_p = &BLE_GATT_SRV_CCCD_DEF_NAME(tx),
            .descr_count = 1U,
        },
    },
#else
    {
        .properties = BLE_GATT_SRV_CHAR_PROP_NOTIFY,
        .permissions = BLE_GATT_SRV_PERM_NONE,
        .min_key_size = BLE_GATT_SRV_MAX_ENCRY_KEY_SIZE,
        .uuid = BLE_UUID_INIT_128(TX_CHR_UUID),
        .descrs = {
            .descrs_p = &BLE_GATT_SRV_CCCD_DEF_NAME(tx),
            .descr_count = 1U,
        },
    },
    {
        .properties = BLE_GATT_SRV_CHAR_PROP_WRITE | BLE_GATT_SRV_CHAR_PROP_WRITE_NO_RESP,
        .permissions = BLE_GATT_SRV_PERM_NONE,
        .min_key_size = BLE_GATT_SRV_MAX_ENCRY_KEY_SIZE,
        .uuid = BLE_UUID_INIT_128(RX_CHR_UUID),
    },
#endif
};
```

将新的特征配置赋值给服务声明，特征的数量修改为 1 个：

图9. 特征声明赋值给服务声明

```
/* Serial port Service definition */
static const ble_gatt_srv_def_t serial_port_service = {
    .type = BLE_GATT_SRV_PRIMARY_SRV_TYPE,
    .uuid = BLE_UUID_INIT_128(SRVC_UUID),
    .chrs = {
        .chrs_p = (ble_gatt_chr_def_t *)serial_port_chars,
        .chr_count = 1U,
    },
};
```

上述服务、特征相关的数据结构配置完毕，我们还需要将这些配置通过 API 传递给协议栈。

首先，需要先定义一个新的句柄，所谓句柄，简单地说，用户在进行数据收发的时候，需要选择在哪个特征上进行数据收发，此时便需要句柄来指定特征（句柄本质上便是 attribute 的 handle 字段）。原来的两个句柄也要注释掉，见图 10

图10. 句柄

```
#if 1 // test
uint16_t RXTXCharHandle;
#else
uint16_t TXCharHandle, RXCharHandle;
#endif
```

然后，将上文定义好的服务、特征通过 aci_gatt_srv_add_service 函数一次性传送给协议栈，协议栈对这些配置进行解析、构建完整的 ATT 属性表，保存在内存中。之后，用户可使用 aci_gatt_srv_get_char_decl_handle 接口获取已分配好的句柄（见图 11），此后的数据交互过程将频繁使用该句柄。

图11. 获取句柄

```
uint8_t Add_Serial_port_Service(void)
{
  uint8_t ret;

    ret = aci_gatt_srv_add_service((ble_gatt_srv_def_t *)&serial_port_service);
    if (ret != BLE_STATUS_SUCCESS)
    {
        goto fail;
    }
  #if 1 // test
    RXTXCharHandle = aci_gatt_srv_get_char_decl_handle((ble_gatt_chr_def_t *)&serial_port_chars[0]);
    printf("Serial Port Service added.\nChar Handle %04X\n", RXTXCharHandle);
  #else
    TXCharHandle = aci_gatt_srv_get_char_decl_handle((ble_gatt_chr_def_t *)&serial_port_chars[0]);
    RXCharHandle = aci_gatt_srv_get_char_decl_handle((ble_gatt_chr_def_t *)&serial_port_chars[1]);

    printf("Serial Port Service added.\nTX Char Handle %04X, RX Char Handle %04X\n",
           TXCharHandle, RXCharHandle);
  #endif
```

至此，GATT 相关的配置已经完成。但是，由于上层大量引用了旧的两个句柄进行数据收发，因此此时编译会出现比较多的错误，此处暂不处理这些错误，先完成其他的 BLE 配置。

接下来修改蓝牙地址，并修改广播名，见图 12：

图12. 蓝牙地址和广播名

```
#define BD_ADDR_SLAVE        0xaa, 0x00, 0x00, 0xE1, 0x80, 0x02


#define LOCAL_NAME  'H','e','l','l','o'
```

广播名的长度改变后，应注意指定其长度：

图13. 广播名长度

```
static uint8_t manuf_data[MANUF_DATA_SIZE] = {
  0x02,AD_TYPE_FLAGS, FLAG_BIT_LE_GENERAL_DISCOVERABLE
  2,                          /* Length of AD type Transmi
  0x0A, 0x00,                 /* Transmission Power = 0 dB
  6,                          /* Length of AD type Complet
  0x09,                       /* AD type Complete Local Na
  LOCAL_NAME,                 /* Local Name */
  13,                         /* Length of AD type Manufac
  0xFF,                       /* AD type Manufacturer info
  0x01,                       /* Protocol version */
  0x05,          /* Device ID: 0x05 = STEVAL-BCN002V1 B
  0x00,                       /* Feature Mask byte#1 */
  0x00,                       /* Feature Mask byte#2 */
  0x00,                       /* Feature Mask byte#3 */
  0x00,                       /* Feature Mask byte#4 */
  0x00                        /* BLE MAC start */
```

LOCAL_NAME 设置的是广播包里的设备名，当设备连接成功后，主机会从 GAP Profile 的 device name 特征里获取另外一个设备名，此处应保持这两个名字一致：

图14. 设备名

```
uint8_t Serial_port_DeviceInit(void)
{
  uint8_t ret;
  uint16_t service_handle;
  uint16_t dev_name_char_handle;
  uint16_t appearance_char_handle;
  uint8_t name[] = {'H', 'e', 'l', 'l', 'o'};
```

图15. 设置设备名

```
/* Set the device name */
ret = Gap_profile_set_dev_name(0, sizeof(name), name);
```

至此，关于蓝牙的应用配置即告完毕。接下来可进行数据通讯相关的配置、实现。

4.3. 和手机进行通讯

上一小节配置完从机功能后，编译会产生大量错误，是因为 BLE 的通讯过程会比较多地引用旧的句柄。循着解决这些编译错误的操作，我们能了解到 BLE 通讯的过程。具体操作如下：

全局搜索旧的发送句柄（TXCharHanlde），我们找到了用于数据发送的协议栈 API，修正之：

图16. 发送 Notify

```
#if SERVER
  #if 1 // test
    ret = aci_gatt_srv_notify(connection_handle, RXTXCharHandle + 1, 0, len,
                                                 (uint8_t *)(cmd + cmd_buff_start));
  #else
    ret = aci_gatt_srv_notify(connection_handle, TXCharHandle + 1, 0, len,
                                                 (uint8_t *)(cmd + cmd_buff_start));
  #endif
```

手机使能订阅后，会通过图 17 的回调函数通知上层。此时应该修改为新的特征句柄，同时，添加一些打印指示 notify 的使能、禁用状态：

图17. 使能订阅回调

```
void aci_gatt_srv_attribute_modified_event(uint16_t Connection_Handle,
                                           uint16_t Attr_Handle,
                                           uint16_t Attr_Data_Length,
                                           uint8_t Attr_Data[])
{
#if ST_OTA_FIRMWARE_UPGRADE_SUPPORT
    OTA_Write_Request_CB(Connection_Handle, Attr_Handle, Attr_Data_Length, Attr_Data);
#endif /* ST_OTA_FIRMWARE_UPGRADE_SUPPORT */
  #if 1 // test
    if(Attr_Handle == RXTXCharHandle + 2)
  #else
    if(Attr_Handle == TXCharHandle + 2)
  #endif
    {
        if(Attr_Data[0] == 0x01)
        {
          APP_FLAG_SET(NOTIFICATIONS_ENABLED);
          printf("[TEST] Notify enabled\r\n");
        }
        else
        {
          APP_FLAG_CLEAR(NOTIFICATIONS_ENABLED);
          printf("[TEST] Notify disabled\r\n");
        }
    }
    else
    {
        printf("Received data from not recognized attribute handle 0x%4X\n",
               Attr_Handle);
    }
}
```

添加新的特征句柄全局变量声明，注释掉旧的：

图18. 句柄声明

```
#if 1 // test
extern uint16_t  RXTXCharHandle;
#else
extern uint16_t  TXCharHandle, RXCharHandle;
#endif
```

关于旧的发送句柄（TXCharHanlde）的问题已经全部解决。继续搜索旧的接收句柄（RXCharHanlde），我们应能找到设备接收手机数据的函数接口，将其中的句柄替换为新的特征句柄，见图 19：

图19. 数据接收回调函数

```
void aci_gatt_srv_write_event(uint16_t Connection_Handle,
                              uint8_t Resp_Needed,
                              uint16_t Attribute_Handle,
                              uint16_t Data_Length,
                              uint8_t Data[])
{
    uint8_t att_error = BLE_ATT_ERR_NONE;
  #if 1
    if(Attribute_Handle == RXTXCharHandle + 1)
  #else
    if(Attribute_Handle == RXCharHandle + 1)
  #endif
    {
        Data_Received(Data_Length, Data);
        att_error = BLE_ATT_ERR_NONE;
    }
```

至此，我们应该能通过全部编译过程，并且已经找到了数据发送、接收的位置。此时可以在这些位置添加数据发送、接收的打印函数。接收数据的用户接口见图 20：

图20. 用户接收 BLE 数据

```
void Data_Received(uint16_t length, uint8_t *data)
{
  printf("\r\n[TEST] Receive data: \r\n");
  for(uint16_t i = 0U; i < length; i++)
  {
    printf("%c", data[i]);
  }
  printf("\r\n");
}
```

关于发送数据，原始工程实现了以下处理流程：

1. 从串口接收数据
2. 解析数据为命令并缓存这些命令
3. 通过轮询的方式，不断将命令缓冲区里的命令发送出去

该处理流程不适用于我们的任务要求，我们需要先取消这部分功能，见图 21

图21. 取消原有的数据处理流程

```
#if 0 // test
  Send_Data_Over_BLE();
#endif
```

然后设计自己的发送函数，见图 22：

图22. 自定义 BLE 数据发送函数

```
uint8_t user_send_data_over_ble(uint16_t handle, uint8_t *p_data, uint16_t data_len)
{
  uint8_t ret;

  if(!APP_FLAG(SEND_DATA) || APP_FLAG(TX_BUFFER_FULL))
    return BLE_STATUS_BUSY;

  if (data_len > (BLE_STACK_DEFAULT_ATT_MTU-3))
    return BLE_STATUS_INVALID_PARAMS;

  ret = aci_gatt_srv_notify(connection_handle, handle + 1, 0, data_len, p_data);

  if (ret == BLE_STATUS_SUCCESS)
  {
    APP_FLAG_CLEAR(SEND_DATA);
    printf("\r\n[TEST] Sent data: \r\n");
    for(uint16_t i = 0U; i < data_len; i++)
    {
      printf("%c", p_data[i]);
    }
    printf("\r\n");
    return ret;
  }

  if(ret == BLE_STATUS_INSUFFICIENT_RESOURCES)
  {
    APP_FLAG_SET(TX_BUFFER_FULL);
  }

  return ret;
}
```

至此，蓝牙的通讯功能已经全部实现完毕。用户可通过：

- Data_Received()接口接收数据
- user_send_data_over_ble()接口发送数据

4.4. 添加其它功能

根据任务要求，我们还需实现下面三个功能：

1. 控制 LED 亮灭
2. 定时 1s 上传心跳包，内容为连接后的秒计数值
3. 每次按键上传按键事件通知

下面开始逐个实现：

4.4.1. 控制 LED 亮灭

首先，实现 LED 亮灭处理函数，

图23. LED 命令处理函数

```
void user_led_process(uint8_t *p_data, uint16_t len)
{
  if (p_data[0] != 0x01) // LED command
  {
    return;
  }

  if (p_data[1] == 0x01)
  {
    BSP_LED_On(BSP_LED1);
  }
  else if (p_data[1] == 0x00)
  {
    BSP_LED_Off(BSP_LED1);
  }
}
```

将其添加到 BLE 数据接收函数处，见图 24。LED 控制功能实现完毕。

图24. 接收数据后进行 LED 命令处理

```
void Data_Received(uint16_t length, uint8_t *data)
{
  printf("\r\n[TEST] Receive data: \r\n");
  for(uint16_t i = 0U; i < length; i++)
  {
    printf("%c", data[i]);
  }
  printf("\r\n");


  user_led_process(data, length);
}
```

4.4.2. 每秒上传心跳包

首先，实现心跳上传处理函数，实现当设备连接后，每秒上传一个 4 字节的计数值，并使用 0xaa 作为命令字。见图 25：

图25. 心跳上传处理

```
void user_heartrate_process(BOOL init_flag)
{
  static uint32_t count = 0;
  uint8_t send_buf[5];

  if (init_flag)
  {
    count = 0;
    return;
  }

  send_buf[0] = 0xaa; // Heart rate event
  memcpy(&send_buf[1], (uint8_t*)&count, 4);

  user_send_data_over_ble(RXTXCharHandle, send_buf, 5);

  count++;
}
```

将心跳处理函数添加到系统任务处理函数 App_Tick 中，App_Tick 会在 main loop 中不断地被调用。心跳包需要在蓝牙连接成功并且使能了订阅之后才可以发送。另外，使用 timeout_flag 变量来控制其每秒只被调用一次，实现方法如下：

图26. 心跳处理

```
void APP_Tick(void)
{
#if CLIENT
  tBleStatus ret;
#endif

  if (APP_FLAG(CONNECTED) && timeout_flag)
  {
    timeout_flag = FALSE;

    if (APP_FLAG(NOTIFICATIONS_ENABLED)) user_heartrate_process(FALSE);
    else                                 user_heartrate_process(TRUE);

  }
```

上述 timeout_flag 变量需要使用每秒循环的软件定时器来周期性置位。软件定时器的应用方式很简单。

首先，实例化一个定时器，并定义超时回调函数：

图27. 软件定时器实例化

```
static BOOL timeout_flag = FALSE;
VTIMER_HandleType second_timer;

void user_second_timer_cb(void* param)
{
  timeout_flag = TRUE;

  HAL_VTIMER_StartTimerMs(&second_timer, 1000);
}
```

然后，在 Serial_port_DeviceInit 函数的末端位置注册超时回调函数并启动定时器。

图28. 启动定时器

```
#endif /* SERVER */

#if CLIENT

  ret = aci_gap_set_scan_configuration(DUPLICATE_FILTER_

  printf("Scan configuration %02X\n", ret);

  ret = aci_gap_set_connection_configuration(LE_1M_PHY_F

  printf("Connection configuration %02X\n", ret);

#endif /* CLIENT */

  second_timer.callback = user_second_timer_cb;
  HAL_VTIMER_StartTimerMs(&second_timer, 1000);


  return BLE_STATUS_SUCCESS;
}
```

4.4.3. 上传按键事件

首先，实现按键回调函数，该函数在按键按下时被调用。发送按键事件前，应检查此时蓝牙是否处理连接、已订阅状态。见图 29：

图29. 按键回调

```
void user_button_callback(void)
{
  printf("[TEST] Button\r\n");
  if (APP_FLAG(CONNECTED) && APP_FLAG(NOTIFICATIONS_ENABLED))
  {
    user_send_data_over_ble(RXTXCharHandle, (uint8_t *)"\x03", 1); // Button event
  }
}
```

按键中断服务函数中调用：

图30. 按键中断服务函数

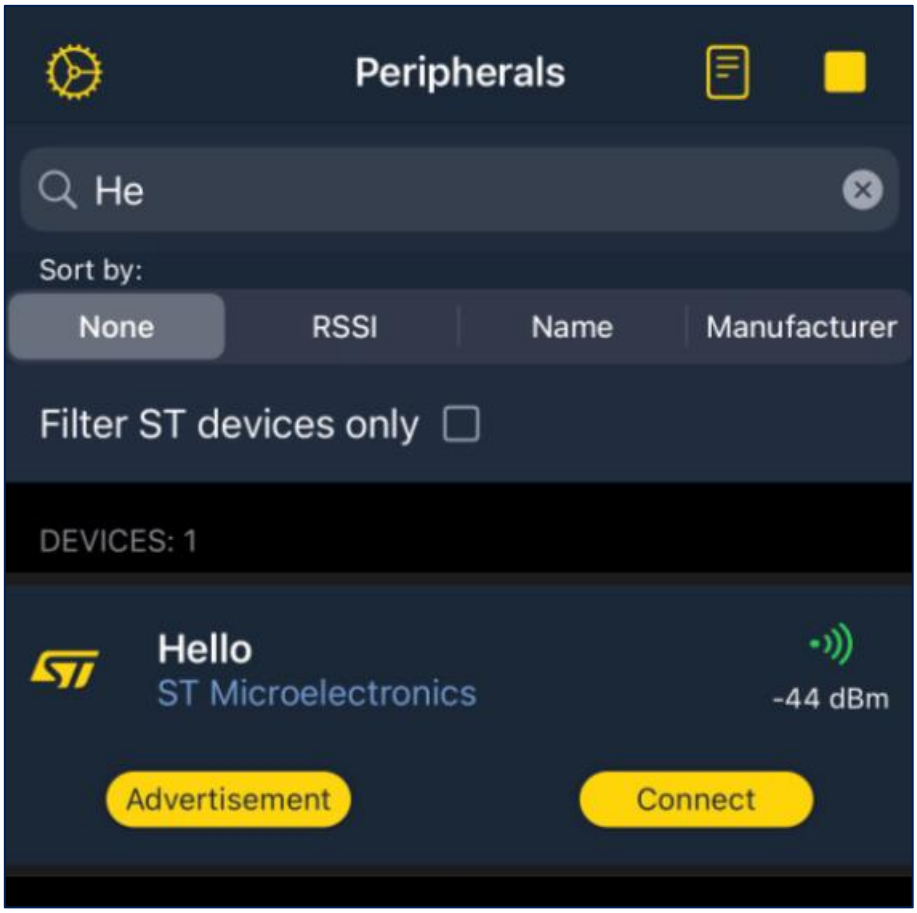
```
void GPIOA_IRQHandler(void)
{
  if (BSP_PB_GetITPendingBit(BSP_PUSH1))
  {
    user_button_callback();
    BSP_PB_ClearITPendingBit(BSP_PUSH1);
  }
}
```

至此，开发任务的要求已经全部实现完毕。接下来进行功能验证。

4.5. 验证功能

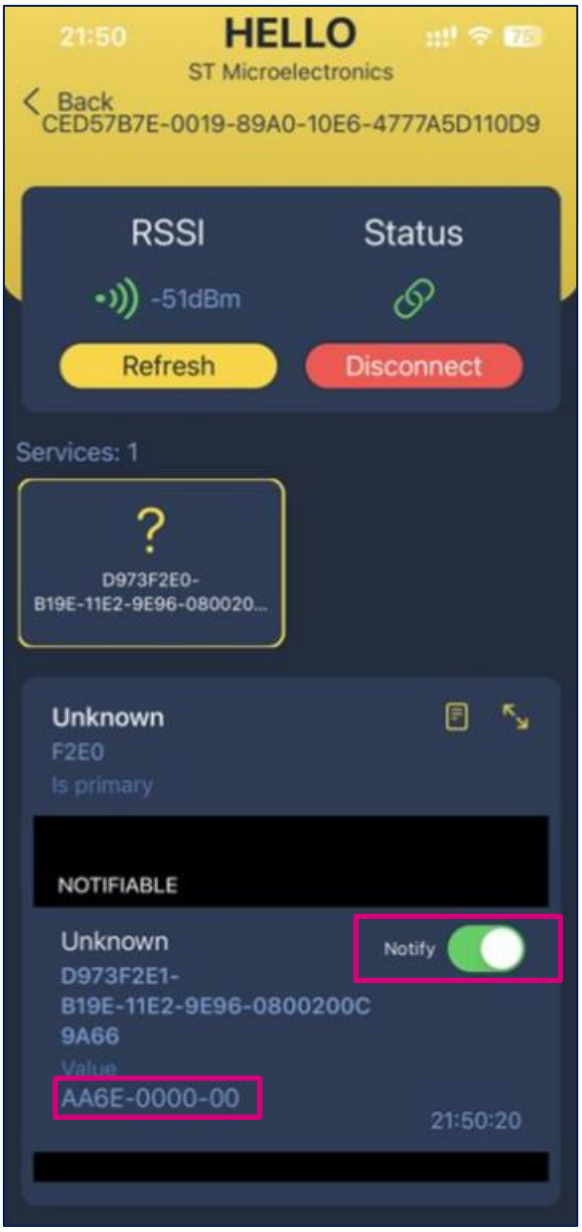
用户工程运行起来后，用 STBLE Toolbox 扫描，可见广播名已经修改过来了。

图31. 用户工程广播名



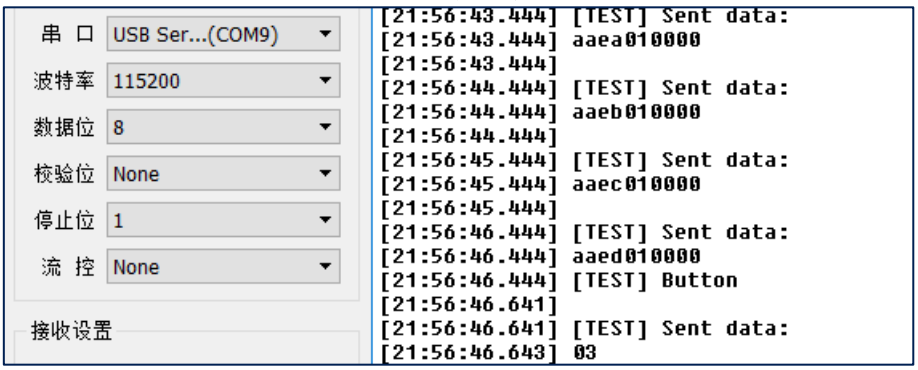
连上设备并点击 Notify 开关以使能订阅，可观察到底部已经开始接收到设备的心跳包数据（以 AA 开头的 5 字节数据），该数据每秒钟变化一次，见图 32：

图32. 使能订阅



通过 LOG 也能观察到心跳包发送情况，此时如果按按键，也能观察到按键事件已经发送：

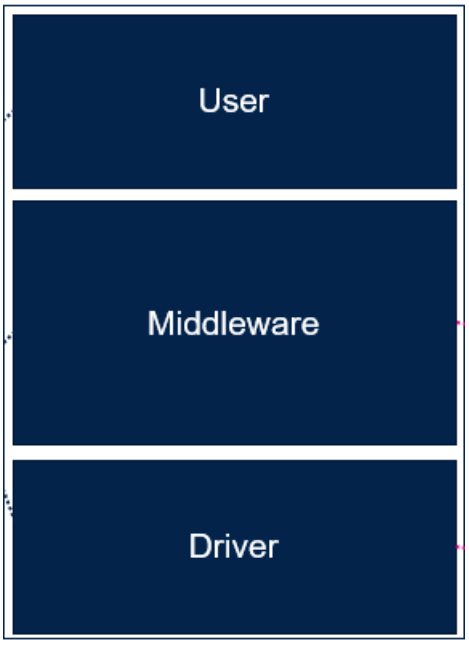
图33. 用户工程 LOG



5. 小结

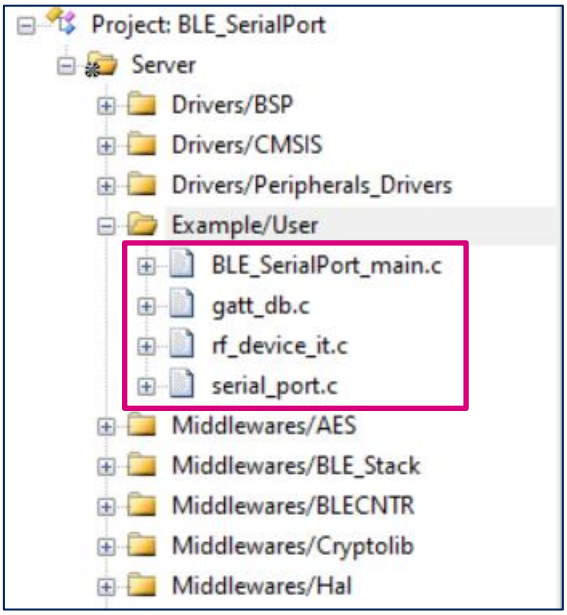
跑完了上述用户任务开发的流程后，相信用户对 BlueNRG SDK 的软件架构应有所理解了。BlueNRG SDK 的软件层次架构为 STM32 典型的三层架构，分别为驱动层、中间层、用户层：

图34. 软件层次架构



上述添加用户功能的整个过程，其实只改动到了用户层的功能，用户层包含以下几个文件：

图35. 用户层文件



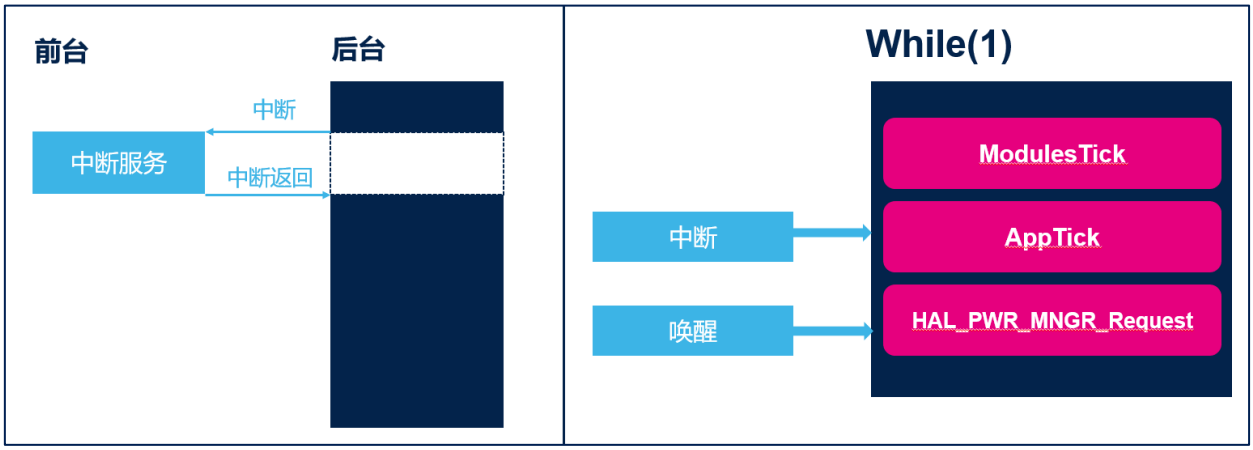
这些文件的含义是：

- serial_port.c，用户应用逻辑的实现
- BLE_SerialPort_main.c，程序入口，程序主流程

- gatt_db.c，BLE GATT 层功能的实现
- rf_device_it.c，存放所有的中断服务函数

上述用户固件的功能，大多都在 serial_port.c 中实现。BLE_SerialPort_main.c 函数则实现了系统的主要流程。简单来说，BlueNRG SDK 的裸机系统即是一个前后台系统。蓝牙事件、按键中断等属于前台处理，负责置位相关标志位和状态，main 函数的 while1 属于后台处理，运行蓝牙协议栈、用户任务处理等后台任务，见图 36：

图36. 系统流程



BlueNRG SDK 中的绝大多数例程都使用了本文档所述的软件架构，即前后台系统。该软件架构比较简单，优点是用户能非常快速地掌握其流程，能够依据本文档的示例快速构建自己的用户功能。缺点是功能比较简单，用户需要在此基础上再添加一个调度器以应对复杂用户功能的要求。

文档中所用到的工具及版本

- 开发板：
  STEVAL-IDB011V2
- SDK：
  STSW-BNRGLP-DK
- APP：
  STBLEToolbox

版本历史

日期	版本	变更
2023 年 08 月 01 日	1.0	首版发布

重要通知 – 请仔细阅读

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

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

ST 不对任何知识产权进行任何明示或默示的授权或许可。

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

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

本文档是 ST 中国本地团队的技术性文章，旨在交流与分享，并期望借此给予客户产品应用上足够的帮助或提醒。若文中内容存有局限或与 ST 官网资料不一致，请以实际应用验证结果和 ST 官网最新发布的内容为准。您拥有完全自主权是否采纳本文档（包括代码，电路图等）信息，我们也不承担因使用或采纳本文档内容而导致的任何风险。

本文档中的信息取代本文档所有早期版本中提供的信息。