AN0077
应用笔记

BLE Application Note

前言

这篇应用笔记描述了使用AT32WB415的无线蓝牙模块来自定义BLE相关的功能，如何进行无线蓝牙模块和MCU之间的沟通，以及MCU在接收到来自蓝牙模块的需求后，如何做出对应的行为。本应用笔记会概述如何添加自定义的服务与特征到蓝牙模块的Profile上，简介AT command的协议，以及在MCU端该如何处理这些来自无线蓝牙模块的需求命令。

这篇文档也描述了如何透过AT command来控制无线蓝牙模块的功能，让用户可以在不修改代码的情况下，改变BLE端的基本配置。

支持型号列表：

支持型号	AT32WB415

目录

表目录

图目录

1 Bluetooth 简介

Bluetooth®技术取得惊人成功的一个关键原因是它为开发者提供了巨大的灵活性。Bluetooth 技术提供了两种无线电选择，为开发者提供了一套多功能的全栈式、适合目的的解决方案，以满足对无线连接不断扩大的需求。

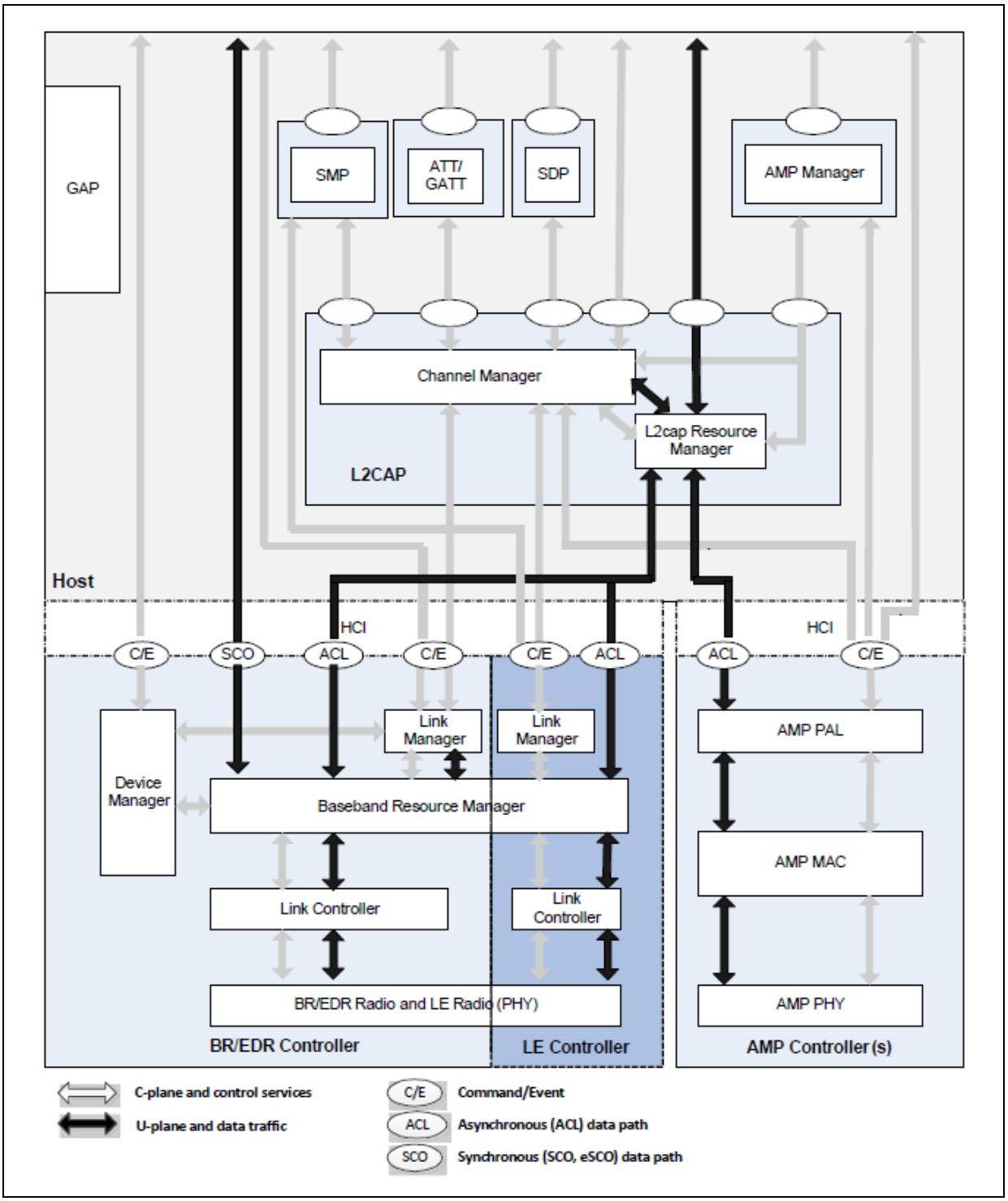
无论产品是在智能手机和扬声器之间传输高质量的音频，还是在平板电脑和医疗设备之间传输数据，或是在楼宇自动化解决方案中的数千个节点之间发送消息，Bluetooth 低能耗(LE)和基本速率/增强数据速率(BR/EDR)无线电台都是为了满足全球开发人员的独特需求而设计的。

在本应用中，我们着眼在低耗能上（以下皆称BLE），经典蓝牙（以下皆称BR/EDR）并不在此应用笔记中讨论，有兴趣的用户可以自行上Bluetooth SIG官方网站了解BR/EDR的内容。

BLE无线电是为非常低的功率操作而设计的。BLE无线电在2.4GHz非授权ISM频段的40个信道上传输数据，为开发者提供了巨大的灵活性，以构建满足其市场独特连接要求的产品。BLE支持多种通信拓扑结构，从点对点扩展到广播，最近又扩展到广播。mesh，使Bluetooth 技术能够支持创建可靠的、大规模的设备网络。虽然最初以其设备通信功能而闻名，但BLE现在也被广泛用作设备定位技术，以满足对高精度室内定位服务日益增长的需求。最初支持简单的存在和接近功能，BLE现在支持Bluetooth®方向查找，并很快支持高精度距离测量。

以下对BLE架构做个简介，请参考图1。

图 1. Bluetooth 核心系统架构



在本应用中，代码中修改的部份，都是在Host的区块，而Controller的部份只使用到LE的区块，整个BLE system都是由无线蓝牙模块来实现。实际上会修改到的部份，是位于Host区块中的GAP与GATT两个小区块，以下会就这两个小区块进行说明，让使用者对修改这两个区块会影响到什么有个初步的了解。

1.1 GAP

GAP是通用访问配置文件(Generic Access Profile)的首字母缩写，用来控制蓝牙的connection和advertising，设备对外界的可见性，并确定两个设备如何(或不能)相互沟通。

1.1.1 装置角色

GAP定义了设备的各种角色，但要记住的两个关键概念是中央(Central)设备和外围(Peripheral)设备。

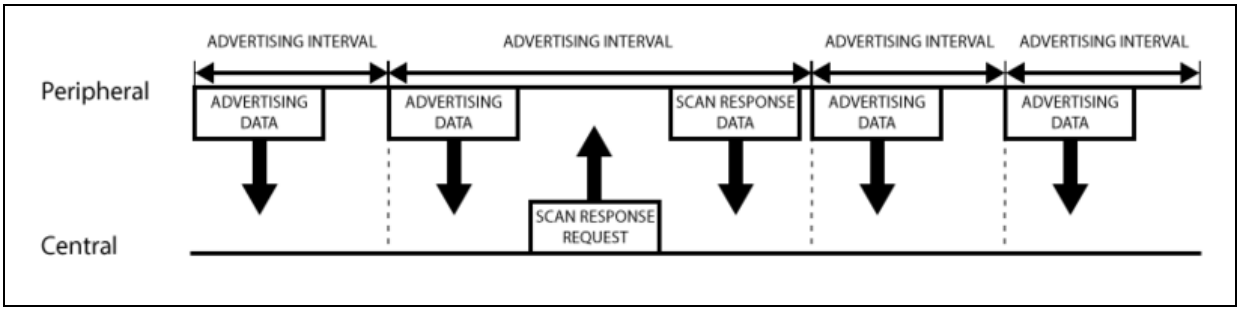
外围设备是小型，低功耗，资源有限的设备，中央设备通常是您连接的移动电话或平板计算机，具有更强的处理能力和内存。

1.1.2 广播和扫描响应数据

有两种方法可以通过GAP发送广播。Advertising Data payload和Scan Response payload。两个有效载荷都可以包含多达31byte的数据，但只有Advertising Data payload是强制性的，它将不断从设备传出，让范围内的中央设备知道它存在。

Scan Response payload是中央设备可以请求的可选次要payload，并且允许设备设计者在Advertising Data payload中容纳更多信息，例如设备名称的字符串等。

图 2. 广播与响应



1.1.3 广播网络拓朴

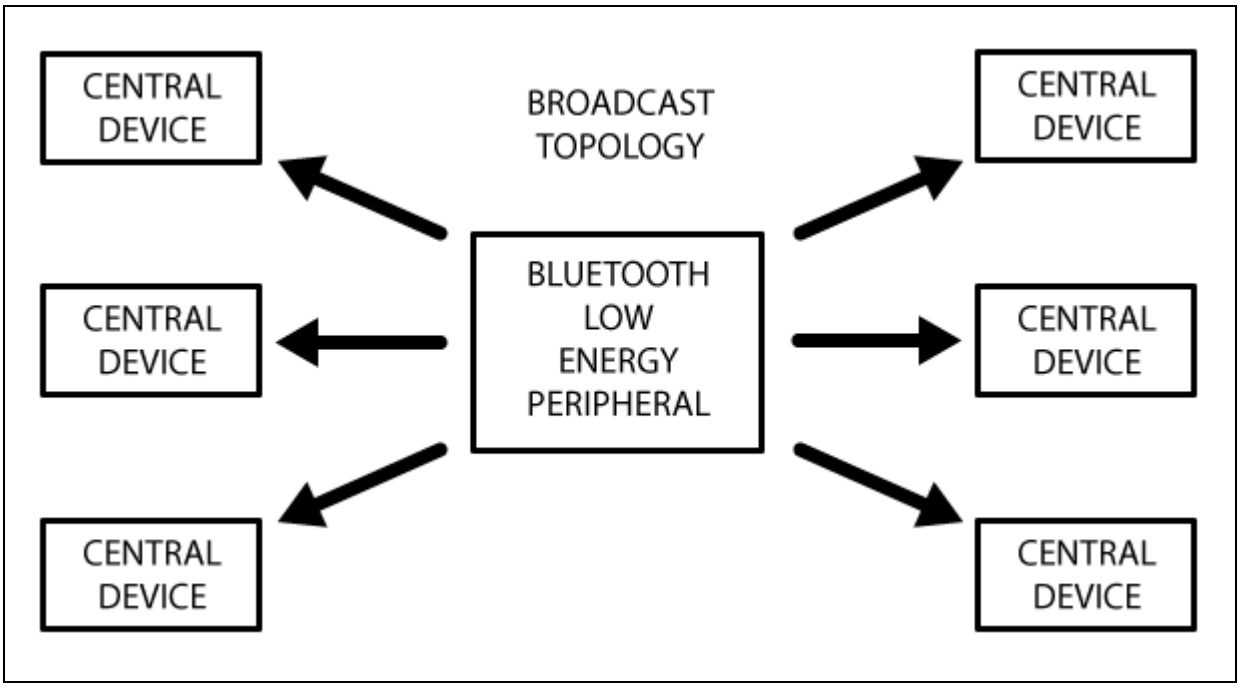
虽然大多数外围设备都在广播自己，以便可以建立连接并且可以使用GATT服务和特征(characteristic)，允许在两个方向上交换更多数据，但是在某些情况下您只想要广播数据。

这里的主要用例是您希望外围设备一次向多个设备发送数据。这只能使用advertising packet，因为在连接模式下发送和接收的数据只能由这两个连接的设备看到。

通过在31 byte payload中包含少量自定义数据，您可以使用低成本蓝牙低功耗外设将数据单向发送到扫描范围内的任何设备，如下图所示。 这被称为蓝牙低功耗广播。

在外围设备和中央设备之间建立连接后，Advertising Process通常会停止，通常将无法再发送advertising packet，并且将使用GATT服务和特征(characteristic)在两个方向进行通讯。

图 3. 广播网络拓朴



1.2 GATT

GATT定义了两个蓝牙低功耗设备使用称为服务(Service)和特征(Characteristic)的概念来回传输数据的方式。它使用称为属性协议(ATT, Attribute Protocol)的通用数据协议，该协议用于在一个简单的查询表中存储服务，特性和相关数据，表中每个条目使用16位ID。

一旦在两个设备之间建立专用连接，GATT即可发挥作用，这意味着已经完成了由GAP管理的广播流程。

GATT和connection最重要的是connection是独占(exclusive)的。这意味着BLE外围设备一次只能连接到一个中央设备！只要外围设备连接到中央设备，它就会停止自我宣传，而其他设备将无法再查看或连接到它，直到现有连接中断。

建立连接也是允许双向通讯的唯一方式，其中中央设备可以向外围设备发送有意义的数据，反之亦然。

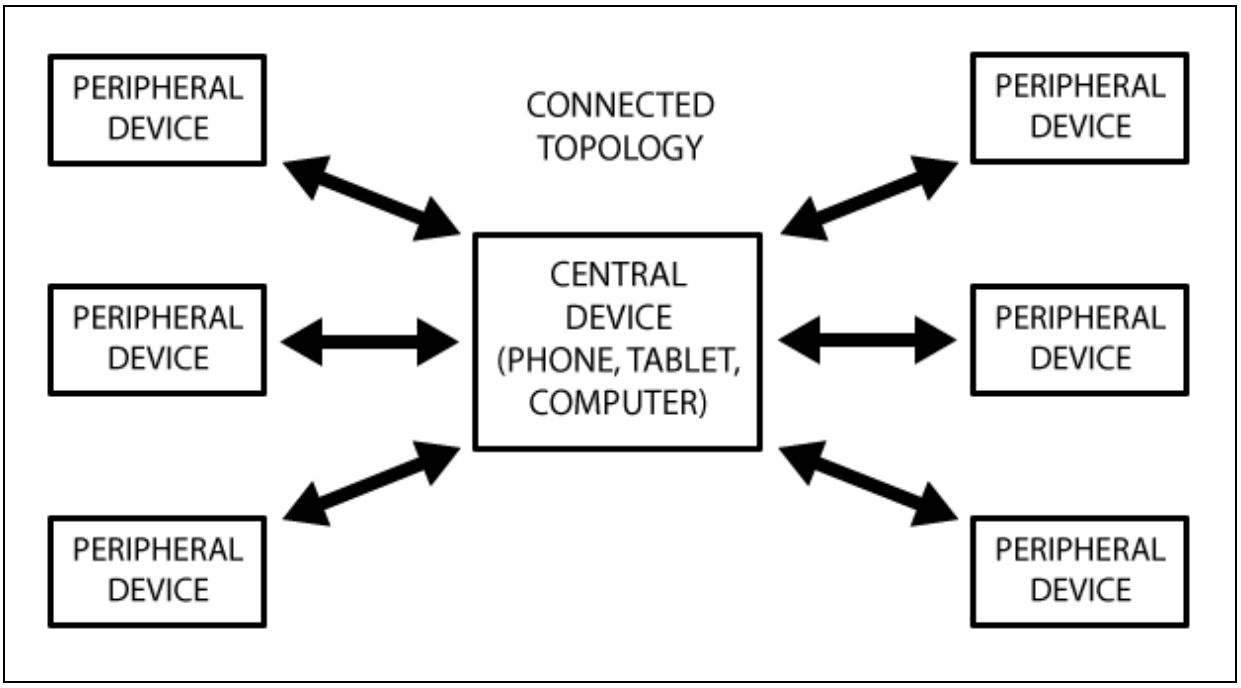
1.2.1 连线网络拓朴

下图应说明蓝牙低功耗设备在连接环境中的工作方式。 外围设备一次只能连接到一个中央设备（例如移动电话），但中央设备可以连接到多个外围设备。

如果需要在两个外围设备之间交换数据，则需要在所有消息都通过中央设备的情况下实施自定义邮箱系统。

然而，一旦在外围设备和中央设备之间建立连接，就可以在两个方向上进行通讯，这与仅使用广告数据和GAP的单向广播方法不同。

图 4. 连线网络拓朴



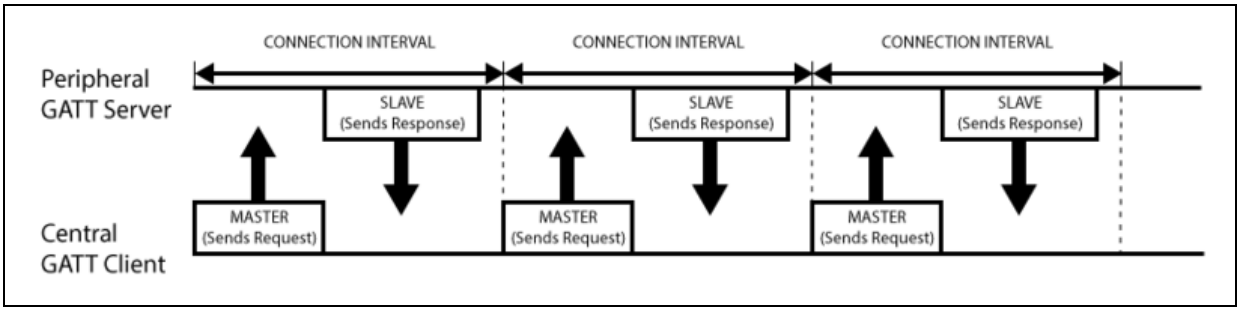
1.2.2 GATT Transactions

使用GATT理解的一个重要概念是服务器/客户端关系。外围设备称为GATT服务器，它保存ATT查找数据和服务及特性定义，以及GATT客户端（电话/平板计算机），它向此服务器发送请求。所有事务都由接收GATT服务器响应的GATT客户端开始。

建立连接时，外围设备将向中央设备建议“连接间隔(Connection Interval)”，并且中央设备将尝试重新连接每个连接间隔以查看是否有可用的新数据等。重要的是要记住，这个连接间隔实际上只是一个建议！ 您的中央设备可能无法兑现请求，因为它正在忙于与其他外围设备通信，或者所需的系统资源不可用。

下图应说明外围设备（GATT服务器）和中央设备（GATT客户端）之间的数据交换过程，主设备启动每个事务。

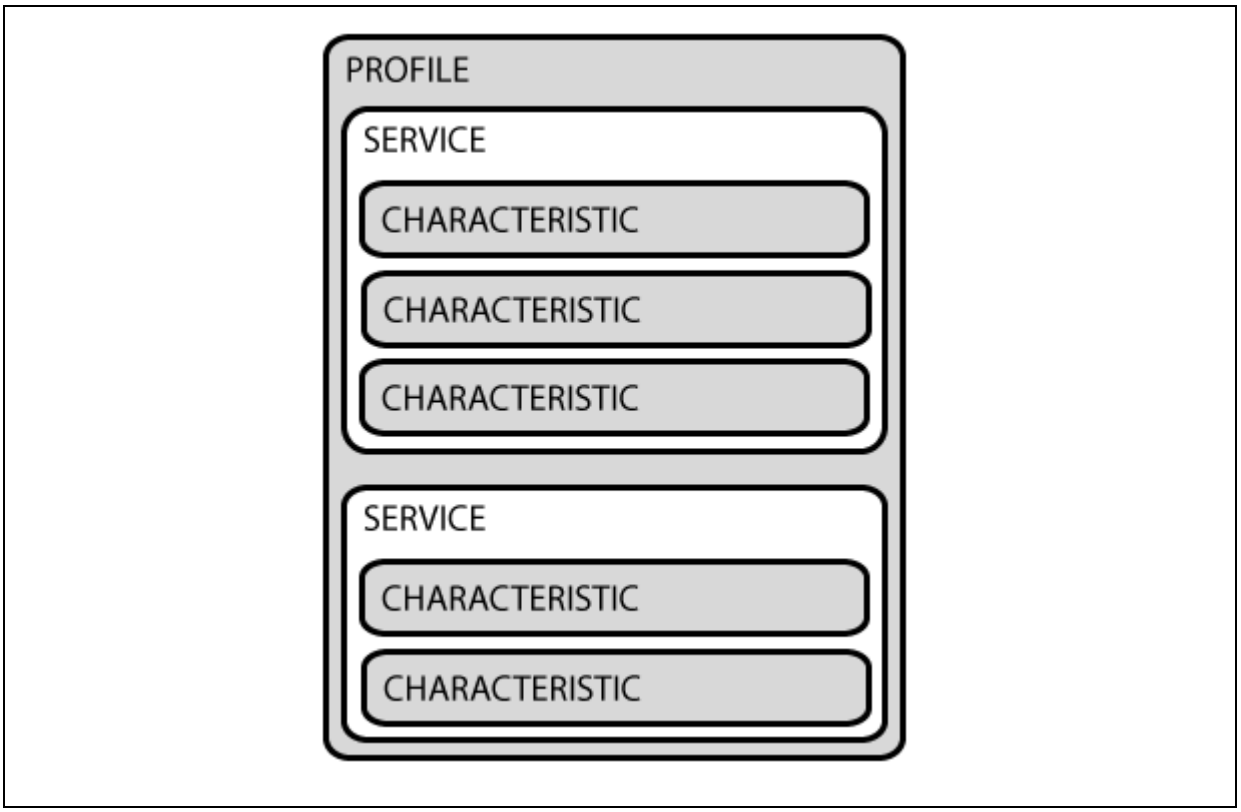
图 5. GATT Transactions



1.2.3 Services and Charateristics

BLE的GATT事务基于名为Profile，Service和Characteristic的高级嵌套物件，如下图所示：

图 6. Profile 架构



1.2.3.1 Profile

BLE 外围设备实际上本身不存在 Profile，它是简单的由 Bluetooth SIG 或外围设备设计人员编译的预定义服务集合。 例如，心率 Profile 结合了心率服务和设备信息服务。可在此处查看官方采用基于 GATT 的 Profile 完整列表：Profiles Overview。

1.2.3.2 Service

Service用于将数据分解为逻辑实体，并包含称为Characteristic的特定数据块。Service可以具有一个或多个Characteristic，并且每个Service通过称为UUID的唯一数字ID将其自身与其他服务区分开来，UUID可以是16位(对于正式采用的BLE服务)或128位(对于自定义服务)。

可以在Bluetooth Developer Portal的“Service”页面上看到正式采用的BLE服务的完整列表。 例如，如果您查看心率服务，我们可以看到这个官方采用的服务具有0x180D的16位UUID，并且包含多达3个Characteristic，但只有第一个是必需的：心率测量，身体传感器位置和心率控制点(Heart Rate Measurement, Body Sensor Location and Heart Rate Control Point)。

1.2.3.3 Charateristics

GATT事务中的最低级别概念是特性(Characteristic)，其封装单个数据点(尽管它可能包含相关数据的数组，例如来自3轴加速度计的X / Y / Z值等)。

与Service类似，每个Characteristic通过预定义的16位或128位UUID进行区分，您可以自由使用Bluetooth SIG定义的标准Characteristic或定义您自己的自定义Characteristic，只有您的外围设备和SW可以理解。

例如，心率测量特性对于心率服务是强制性的，并且使用UUID为0x2A37。它以描述HRM数据格式的单个8位值开始(无论数据是UINT8还是UINT16等)，并且继续包括与该配置byte匹配的心率测量数

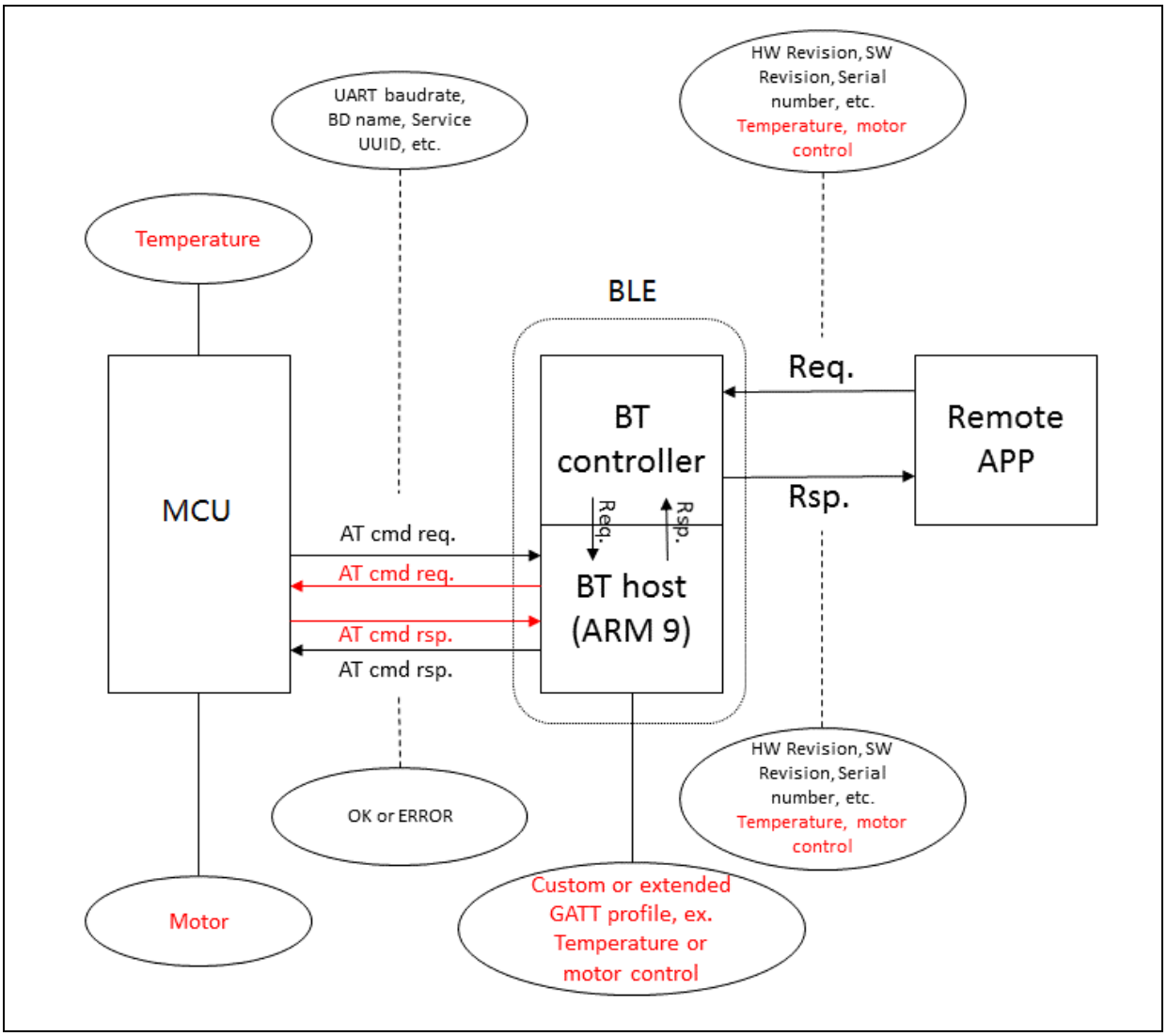
据。

Characteristic是与BLE外围设备交互的要点，因此理解这一概念非常重要。 它们还用于将数据发送回BLE外围设备，因为您还可以写入Characteristic。 您可以使用自定义“UART Service”和两个Characteristic实现简单的UART类型接口，一个用于TX通道，另一个用于RX通道，其中一个特性可能配置为只读，另一个特性具有写权限。

1.3 系统框架

AT32WB415实际上是由MCU及无线蓝牙模块(BLE)两颗芯片组合而成，中间透过UART界面沟通，蓝牙芯片收到来自远程APP的请求后，透过AT command，跟MCU获取需要的信息或是执行某些动作；或者是由MCU端发送请求，透过UART发送AT command，变更BLE端的配置。无论是哪个方向发出请求，用户都可以根据需求扩增AT command，实现各种控制方法。

图 7. 系统框架图



2 添加自定义服务到无线蓝牙模块

在本例程中，已经有GATT必须的相关服务了，可以透过远程的APP取得这些服务内容，但若要实现用户想要的功能，此时就必须要自定义服务了。本应用中已经帮用户写好了一个自定义的服务，后续用户想要新增其他服务，可以参照此例程去建立。

另外，该工程是ARM9的工程，需要安装Legacy Support才能够编译，用户可根据自己的环境在以下路径进行下载：www2.keil.com/mdk5/legacy/。

2.1 添加档案到工程中

新增一个自定义服务，需要8个文件，这边以LED Switch为例：

- led_switch.c
- led_switch.h
- led_switch_task.c
- led_switch_task.h
- app_led_switch.c
- app_led_switch.h

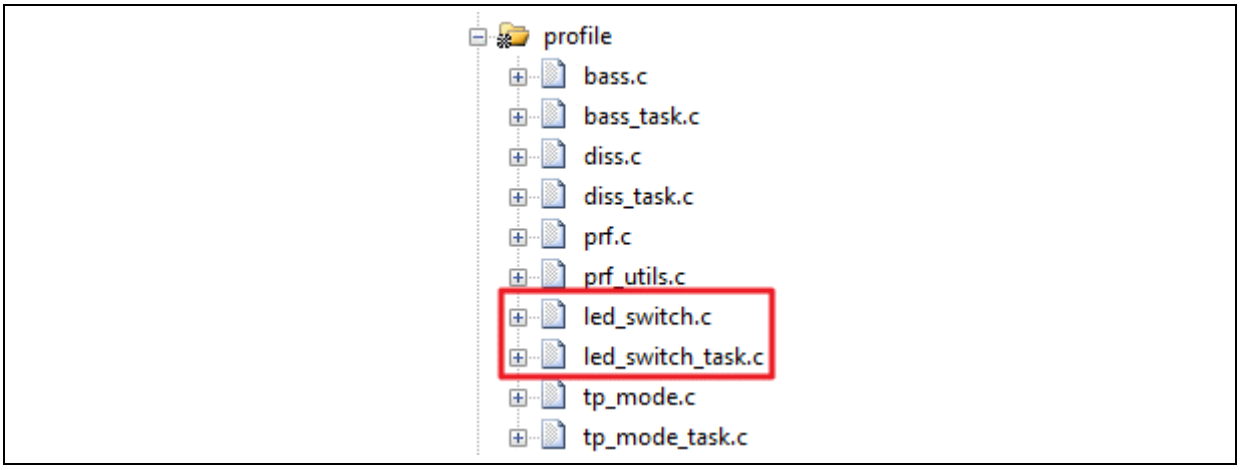
需要将以上文件放到以下目录，这个地方需要自行建立文件夹：

- led_switch.c和led_switch_task.c放入sdk\ble_stack\com\profiles\led_switch\src
- led_switch.h和led_switch_task.h放入sdk\ble_stack\com\profiles\led_switch\api
- app_led_switch.c和app_led_switch.h放入projects\ble_app_gatt\app

2.2 在工程中配置档案

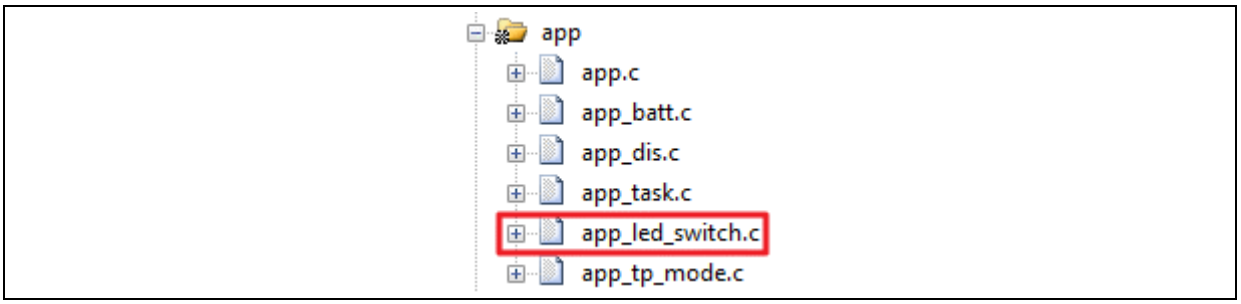
1. 开启 Keil 将 led_switch.c 和 led_switch_task.c 加入 profile 的目录下

图 8. profile 目录下的文件



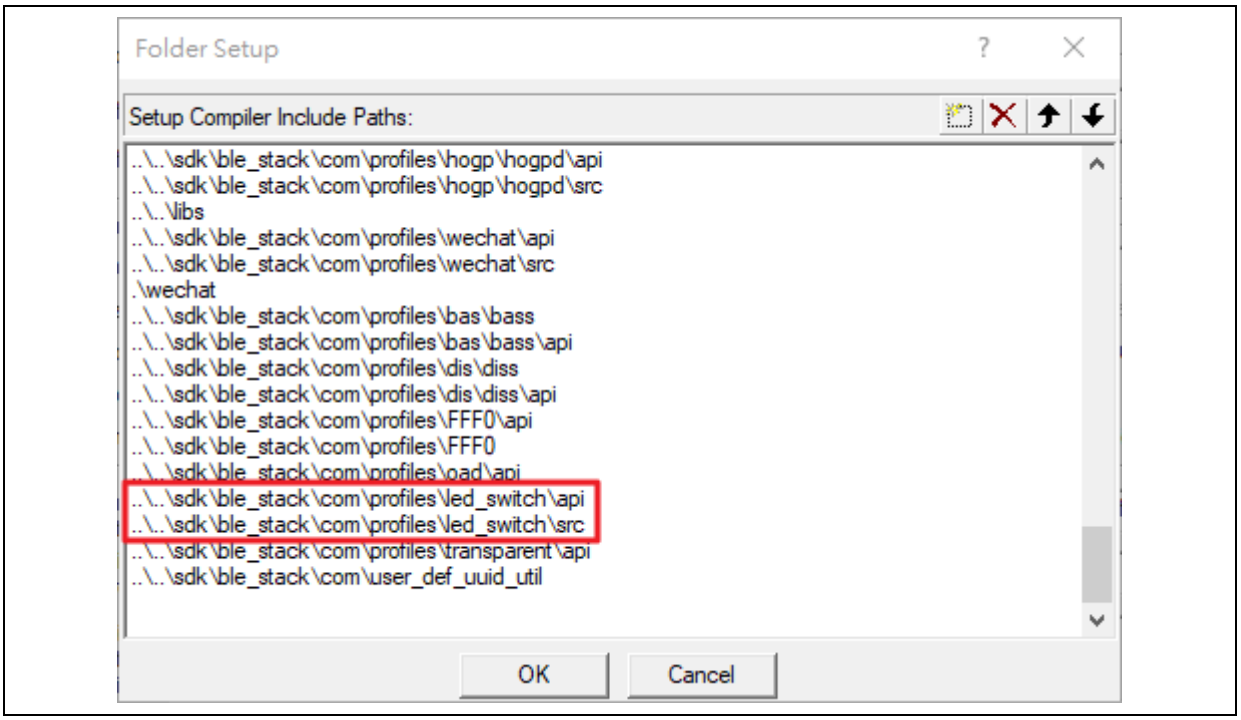
2. 将 app_led_switch.c 加入 app 目录下

图 9. app 目录下的文件



3. 在 Keil C/C++选项卡的 Include Paths 中添加对应的 profile 路径

图 10. Include Paths



2.3 将自定义服务添加至当前软件架构

1. 在 app_task.c 文件中找到 appm_msg_handler 函数，新增处理 led_switch ID 的 message case

图 11. 处理新增 task ID 的入口点

```
case (TASK_ID_LED_SWITCH):
{
    // Call the LED Switch Module
    msg_pol = appm_get_handler(&app_led_switch_table_handler, msgid, param, src_id);
}
break;
```

2. 在 app.c 中新增项目到服务列表 appm_svc_list

图 12. 新增服务列表项目

```
/// List of service to add in the database
enum appm_svc_list
{

    APPM_SVC_UART_TP,

#if BLE_LED_SWITCH_SERVER
    APPM_SVC_LED_SWITCH,
#endif
#if BLE_FFF0_SERVER
    APPM_SVC_FFF0,
#endif
    APPM_SVC_DIS,
    APPM_SVC_BATT,
#if BLE_OADS_SERVER
    APPM_SVC_OADS,
#endif
    APPM_SVC_LIST_STOP,
};
```

3. 在 app.c 中新增函数列表以建立 database

图 13. 函数列表

```
/// List of functions used to create the database
static const appm_add_svc_func_t appm_add_svc_func_list[APPM_SVC_LIST_STOP] =
    {

        (appm_add_svc_func_t)app_uart_add_uarts,
#if BLE_LED_SWITCH_SERVER
        (appm_add_svc_func_t)app_led_switch_add_switches,
#endif
#if BLE_FFF0_SERVER
        (appm_add_svc_func_t)app_fff0_add_fff0s,
#endif
        (appm_add_svc_func_t)app_dis_add_dis,
        (appm_add_svc_func_t)app_batt_add_bas,
#if BLE_OADS_SERVER
        (appm_add_svc_func_t)app_oad_add_oads,
#endif
};
```

4. 在 app.c 中找到 appm_init 函数，在其中添加 app_led_switch_init 函数

图 14. 初始化自定义服务

```
    // Device Information Module
    app_dis_init();

    // Battery Module
    app_batt_init();

#if BLE_OADS_SERVER
    app_oads_init();
#endif

#if BLE_LED_SWITCH_SERVER
    app_led_switch_init();
#endif

#ifdef UART_TP_MODE
    app_uart_tp_init();
#endif
```

5. 在 rwip_task.h 中的 TASK_API_ID 中新增自定义服务的 ID

图 15. 新增 task ID

```
    TASK_ID_FCC0S = 74, // FFC0 PROFILE SERVICE TASK

    TASK_ID_FEE0S = 75,

    TASK_ID_LED_SWITCH = 76, // LED Switch Profile Serivice Task

    TASK_ID_UART_TP = 77, // UART Transparent Profile Serivice Task
```

6. 在 prf.c 中添加 led_switch_prf_itf_get 函数的调用

图 16. 调用 led_switch_prf_itf_get()

```
static const struct prf_task_cbs * prf_itf_get(uint16_t task_id)
{
    const struct prf_task_cbs* prf_cbs = NULL;

    switch(KE_TYPE_GET(task_id))
    {

    #if (BLE_UART_TP_SERVER)
        case TASK_ID_UART_TP:
        prf_cbs = uart_tp_prf_itf_get();
        break;
        #endif // (BLE_UART_TP_SERVER)


    #if (BLE_LED_SWITCH_SERVER)
        case TASK_ID_LED_SWITCH:
        prf_cbs = led_switch_prf_itf_get();
        break;
        #endif // (BLE_LED_SWITCH_SERVER)
```

7. 在 prf.c 中添加 led_switch_prf_itf_get 函数的声明

图 17. 声明 led_switch_prf_itf_get()

```
#if (BLE_LED_SWITCH_SERVER)
extern const struct prf_task_cbs* led_switch_prf_itf_get(void);
#endif // (BLE_LED_SWITCH_SERVER)
```

8. 在 rwprf_config.h 中增加以下定义

图 18. 打开自定义服务的宏及列为伺服 profile 的条件

```
473  /// LED Switch Profile server role
474  #if defined(CFG_PRF_LED_SWITCH)
475  #define BLE_LED_SWITCH_SERVER 1
476  #else
477  #define BLE_LED_SWITCH_SERVER 0
478  #endif // defined(CFG_PRF_LED_SWITCH)
479
480  /// UART Transparent Profile server role
481  #if defined(CFG_PRF_UART_TP)
482  #define BLE_UART_TP_SERVER 1
483  #else
484  #define BLE_UART_TP_SERVER 0
485  #endif // defined(BLE_UART_TP_SERVER)
486
487  /// BLE_CLIENT_PRF indicates if at least one client profile is present
488  #if (BLE_PROX_MONITOR || BLE_FINDME_LOCATOR || BLE_HT_COLLECTOR || BLE_BP_COLLECTOR || BLE_HR_COLLECTOR || BLE_DIS_CLIENT ||
489       BLE_TIP_CLIENT || BLE_SP_CLIENT || BLE_BATT_CLIENT || BLE_GL_COLLECTOR || BLE_HID_BOOT_HOST || BLE_HID_REPORT_HOST ||
490       BLE_RSC_COLLECTOR || BLE_CSC_COLLECTOR || BLE_CP_COLLECTOR || BLE_LN_COLLECTOR || BLE_AN_CLIENT || BLE_PAS_CLIENT ||
491       BLE_IPS_CLIENT || BLE_ENV_CLIENT || BLE_WSC_CLIENT || BLE_ANCS_CLIENT)
492  #define BLE_CLIENT_PRF 1
493  #else
494  #define BLE_CLIENT_PRF 0
495  #endif //(BLE_PROX_MONITOR || BLE_FINDME_LOCATOR ...)
496
497  /// BLE_SERVER_PRF indicates if at least one server profile is present
498  #if (BLE_PROX_REPORTER || BLE_FINDME_TARGET || BLE_HT_THERMOM || BLE_BP_SENSOR || BLE_TIP_SERVER || BLE_HR_SENSOR || \
499       BLE_DIS_SERVER || BLE_SP_SERVER || BLE_BATT_SERVER || BLE_HID_DEVICE || BLE_GL_SENSOR || BLE_RSC_SENSOR || \
500       BLE_CSC_SENSOR || BLE_CP_SENSOR || BLE_LN_SENSOR || BLE_AN_SERVER || BLE_PAS_SERVER || BLE_IPS_SERVER || \
501       BLE_ENV_SERVER || BLE_WSC_SERVER || BLE_UDS_SERVER || BLE_BCS_SERVER || BLE_WPT_SERVER || BLE_PLX_SERVER || \
502       /*BLE_FFF0_SERVER ||*/ BLE_FFE0_SERVER || BLE_FEE0_SERVER || BLE_LED_SWITCH_SERVER || BLE_UART_TP_SERVER)
503  #define BLE_SERVER_PRF 1
504  #else
505  #define BLE_SERVER_PRF 0
506  #endif //(BLE_PROX_REPORTER || BLE_FINDME_TARGET ...)
```

BLE_LED_SWITCH_SERVER 这个宏定义在 led_switch.c, led_switch.h, led_switch_task.c, led_switch_task.h 中都有用到，只有当这个宏被打开后，编译器才会把自定义的服务编译进去。

2.4 BLE 接口说明

1. Led_switch 服务实现一个可读写的特征，其 UUID 及相关属性如下表所示

表 1. 自定义服务的特征

UUID	特征权限	可发送/接收数据长度
0xA00112AC-4202-56BE-EE11-193BA0C45D9E	Read/Write without response	1 Byte

需要在 led_switch 的 ATT database 中去设定权限

图 19. 自定义服务的 ATT database

```
/// Full LED Switch Database Description - Used to add attributes into the database
const struct attm_desc_128 led_switch_att_db[LED_SWTICH_IDX_NB] =
    {
        // Device Information Service Declaration
        [LED_SWITCH_IDX_SVC] = {ATT_BT_UUID_128(ATT_DECL_PRIMARY_SERVICE), PERM(RD, ENABLE), 0, 0},
        // Manufacturer Name Characteristic Declaration
        [LED_SWTICH_IDX_CHAR] = {ATT_BT_UUID_128(ATT_DECL_CHARACTERISTIC), PERM(RD, ENABLE), 0, 0},
        // Manufacturer Name Characteristic Value
        [LED_SWTICH_IDX_VAL] = {LED_ATT_BT_UUID_128(ATT_LED_SWITCH_CHAR_UUID), PERM(RD, ENABLE) | PERM(WRITE_COMMAND, ENABLE),
                                PERM(UUID_LEN, UUID_128) | PERM(RI, ENABLE) | ATT_UUID_128_LEN, LED_SWITCH_VAL_MAX_LEN},
};
```

结构体的第二个参数可以设定该服务或是特征的权限，权限的定义如下

表 2. 权限定义

代码符号	意义
RD	Read
WRITE_REQ	Write
WRITE_COMMAND	Write without response
NTF	Notification
IND	Indication

2. 数据发送函数位于 led_switch_task.c 中，这边使用 gattc_write_req_ind_handler()函数来实现

图 20. 数据发送函数

```
static int gattc_write_req_ind_handler(ke_msg_id_t const msgid, struct gattc_write_req_ind const *param,
                                       ke_task_id_t const dest_id, ke_task_id_t const src_id)
{
    struct gattc_write_cfm * cfm;
    uint8_t status = GAP_ERR_NO_ERROR;
    //Send AT command
    UART_SEND_DATA(AT_CMD_IO_SET, param->value[0]);

    cfm = KE_MSG_ALLOC(GATTC_WRITE_CFM, src_id, dest_id, gattc_write_cfm);
    cfm->handle = param->handle;
    cfm->status = status;
    ke_msg_send(cfm);

    return (KE_MSG_CONSUMED);
}
```

3. 数据接收函数位于 app_led_switch.c 中，这边使用 led_switch_value_req_ind_handler()来实作，里面可以透过 switch 去增加其他 case，用户可以参考范例去实做

图 21. 数据接收函数

```
static int led_switch_value_req_ind_handler(ke_msg_id_t const msgid,
                                            struct led_switch_value_req_ind const *param,
                                            ke_task_id_t const dest_id,
                                            ke_task_id_t const src_id)
{
    const uint8_t led_on = 1, led_off = 0;
    // Initialize length
    uint8_t len = 0;
    // Pointer to the data
    uint8_t *data = NULL;

    at_rsp_content *rsp_content;

    // Check requested value
    if (uart_tp_mode_flag == false)
    {
        switch (param->value)
        {
        case LED_ON_OFF_STATUS:
        {
            // AT command
            UART_SEND_DATA(AT_CMD_IO_GET);
            // Wait for response
            rsp_content = at_wait_for_rsp();
            // Set information
            len = APP_LED_SWITCH_LEN;
            if (rsp_content->data[4] == 0x31)
            {
                data = (uint8_t *)&led_off;
            }
            else
            {
                data = (uint8_t *)&led_on;
            }
        }
        break;

        default:
            ASSERT_ERR(0);
            break;
        }
    }
```

3 AT command

3.1 概述

海斯命令集（Hayes command set），又称 AT 命令集（AT command set），原本是为了海斯智能型 300 调制解调器所开发的一种命令语言。这些命令集是由许多短的字符串组成长的命令，用于代表拨号、挂号以及改变通讯参数的动作。大部分的调制解调器都跟随海斯命令集所制定的规则。
海斯指令可以被区分为四个群组：

1. 基本指令集：一个大写字符跟着一个数值，例如：M1
2. 延伸指令集：一个"&"以及一个大写字符跟着一个数值，这是基本指令集的延伸，例如：&M1
3. 特殊指令集：通常用一个倒斜线（"\"）或一个百分比符号（"%"）；这很广泛使用在调制解调器制造厂商
4. 缓存器指令集：Sr＝n 其中 r 代表是缓存器的编号，n 代表是要指定给缓存器的数值

3.2 无线蓝牙模块命令

在本应用笔记中，只使用到基本指令集的部份。一些重要的字符用于调制解调器初始化：

1) AT - "Attention"：告知调制解调器后面跟着是调制解调器指令，每一行以 AT 为起始
2) Z - 重新设定（reset）调制解调器回到初始状态
3) （a comma） - 使软件暂停一秒钟，若有多个逗点则代表暂停许多秒
4) ^M - 传送一个终止符（Carriage Return）给调制解调器，这是一个控制字符（当传送此字符其实是传送一个字节，内容为 ASCII 的 CR）

以下列出本应用中有实作的 AT command 列表：

表 3. AT command set list(send from MCU)

		MCU 发送	蓝牙模块回复	备注
错误或是没有支持的命令			ERROR	当无线蓝牙模块收到不 support 或是错误的 command，回复 ERROR，MCU/无线蓝牙模块重新发送新的 AT command e.g. MCU 发送 ATT 这个错误 command 发送：ATT 回复：ERROR
Startup 测试：AT		AT	OK	A. 用于确认无线蓝牙模块是否 ready B. MCU 收到回复 OK，确认开始 AT command，避免无线蓝牙模块上电还没有完成 initial，MCU 就开始送 AT command 造成误动作 e.g. 测试无线蓝牙模块是否在 AT command mode 发送：AT 回复：OK
修改 UART baud rate，存入 Flash	9600bps	AT+BAUD1	OK9600	A. default baud rate：9,600bps B. 无线蓝牙模块在回复baud
	19200bps	AT+BAUD2	OK19200	
	38400bps	AT+BAUD3	OK38400	

中： AT+BAUD	57600bos	AT+BAUD4	OK57600	rate后，将新设定入Flash，用新的baud rate与MCU通讯，重新上电继续设定的baud rate 通讯 C. 无线蓝牙模块在回复baud rate后，马上改用新的baud rate做通讯 e.g.改baud rate为115,200bps 发送：AT+BAUD5 回复：OK115200 之后用115,200bps 通讯，无线蓝牙模块重新上电reset后用115,200bps 通讯
	115200bps	AT+BAUD5	OK115200	
修改 UART baud rate，存入 SRAM 中： AT+BAUDS	9600bps	AT+BAUDS1	OK9600	A. 无线蓝牙模块在回复baud rate后，改用新的baud rate与MCU通讯，重新上电恢复用Flash设定 baud rate 通讯 B. 无线蓝牙模块在回复baud rate后，马上改用新的baud rate做通讯 e.g.改 baud rate 为 19,200bps 发送：AT+BAUDS2 回复：OK19200 之后用 19200bps 通讯，无线蓝牙模块重新上电 reset 后用存在 flash 的 baud rate 通讯
	19200bps	AT+BAUDS2	OK19200	
	38400bps	AT+BAUDS3	OK38400	
	57600bos	AT+BAUDS4	OK57600	
	115200bps	AT+BAUDS5	OK115200	
修改蓝牙(BD name)名称，存入 Flash 中： AT+NAME		AT+NAMExxxx	OKxxxx	A. default name : SerialSPP B. 无线蓝牙模块在回复 BD Name 后，将新设定入 Flash，用新的 BD Name 做 advertising，重新上电继续用新 BD name 通讯 C. 最多支持 20 个 char 的 BD name 更改 e.g. 改 BD name 为 Serial-GATT 发送：AT+NAMESerial-GATT 回复：OKSerial-GATT 之后用 Serial-GATT 蓝牙名称做广播，无线蓝牙模块重新上电 reset 后用蓝牙名称维持用 Serial-GATT
修改蓝牙(BD name)名称，存入 SRAM 中： AT+NAMES		AT+NAMESxxx	OKxxxx	A. 无线蓝牙模块在回复 BD Name 后，用新的 BD Name 做 advertising，重新上电恢复用 Flash 内设定的 BD name 通讯

				B. 最多支持 20 个 char 的 BD name 更改 e.g. 改 BD name 为 Serial-GATT 发送：AT+NAMESSerial-GATT 回复：OKSerial-GATT 之后用 Serial-GATT 蓝牙名称，无线蓝牙模块重新上电 reset 后用蓝牙名称恢复用存于 Flash 内设定的 BD name
更改广播间隔时间存入 Flash 中：AT+ADVI	100ms	AT+ADVI1	OK100	A. default Advertising interval: 100ms B. 无线蓝牙模块在回复 OK 后，设定存入 Flash，改用新的 interval 做广播封包，重新上电继续用新的时间设定做广播 e.g. 改 Advertising interval 为 100ms 发送：AT+ADVI1 回复：OK100 之后广播间隔时间用 100ms，无线蓝牙模块重新上电 reset 后维持用 100ms 的广播间隔时间
	250ms	AT+ADVI2	OK250	
	500ms	AT+ADVI3	OK500	
	1600ms	AT+ADVI4	OK1600	
	3200ms	AT+ADVI5	OK3200	
更改广播间隔时间存入 SRAM 中：AT+ADVIS	100ms	AT+ADVIS1	OK100	无线蓝牙模块在回复 OK 后，改用新的 interval 做广播封包，重新上电时恢复用 Flash 内的间隔时间 e.g. 改 Advertising interval 为 100ms 发送：AT+ADVIS1 回复：OK100 之后广播间隔时间用 100ms，无线蓝牙模块重新上电 reset 后恢复用 Flash 内设定的广播间隔时间
	250ms	AT+ADVIS2	OK250	
	500ms	AT+ADVIS3	OK500	
	1600ms	AT+ADVIS4	OK1600	
	3200ms	AT+ADVIS5	OK3200	
读取 Flash：AT+RFLASH		AT+RFLASHad	OKadda	A. default 256byte data value :FF ad(address)：1char da(data)：1 char B. 无线蓝牙模块在回复 OK 带 address 与对应的 data e.g. 读取 address：00 的 data 发送：AT+RFLASH00 回复：OK00FF 读取 address：00 的 data 是 FF
写入 Flash：AT+WFLASH		AT+WFLASHad ,da	OKadda	A. default reserved for MCU accessing 256byte data :FF

			ad(address)：1char da(data)：1 char B. 无线蓝牙模块在回复 OK 带 address 与对应的 data e.g. 写入 address：00 的 data：AA 发送：AT+WFLASH00AA 回复：OK00AA 写入 address：00 的 data 是 AA

表 4. AT command set list(send from BLE)

	BLE 发送	MCU 回复	备注
读取远程 IO 电平：AT+IOGET	AT+IOGET	OKIOx	X 为 0 或 1，0 代表低电平，1 代表高电平
写入远程 IO 电平：AT+IOSET	AT+IOSETx	OKIOx	X 为 0 或 1，0 代表低电平，1 代表高电平

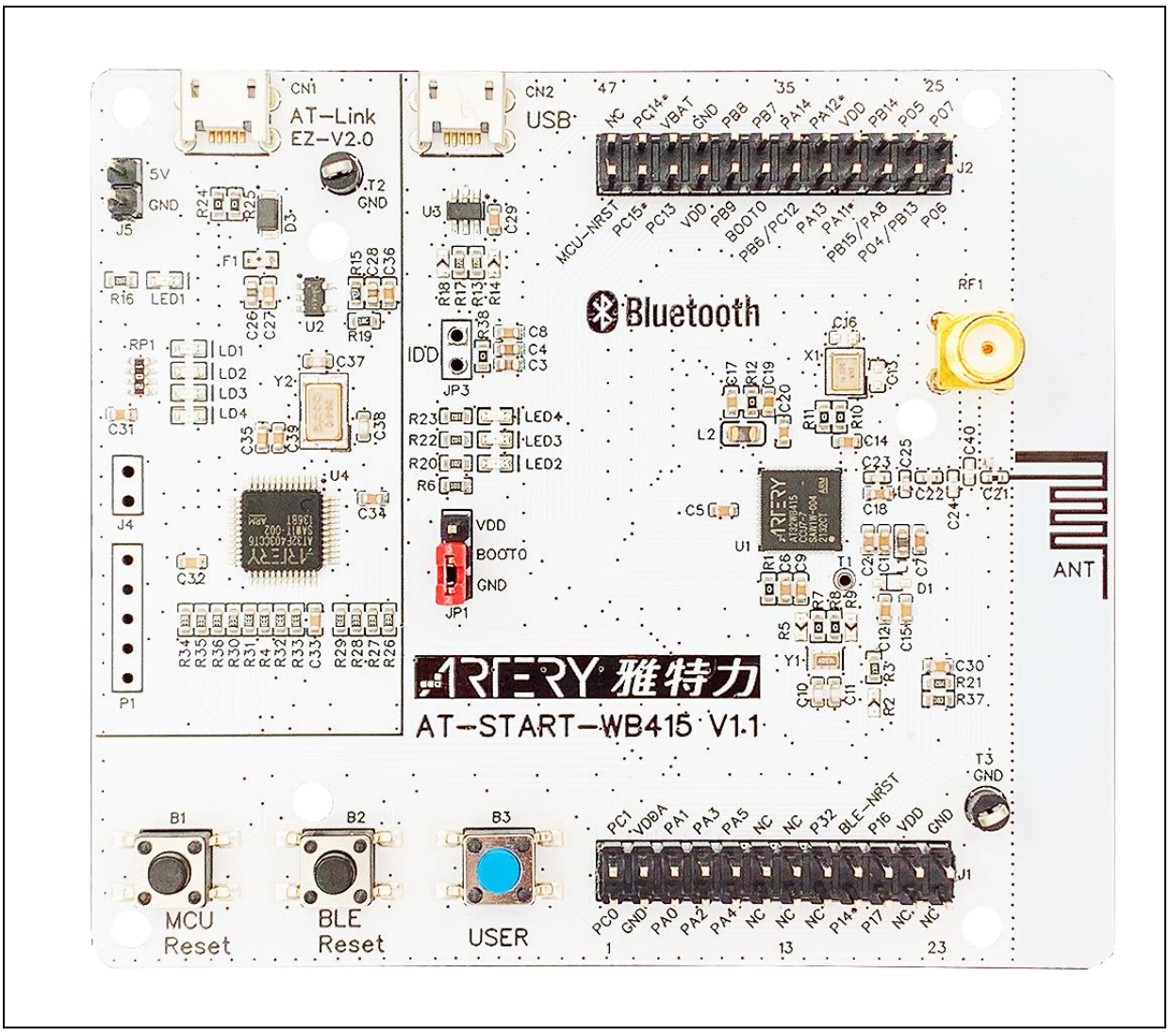
4 BLE 案例使用

本案例将展示如何通过 BLE 完成手机端对 AT32WB415 的操作，包括 IO 控制以及 IO 数据读取。

4.1 硬件资源

1) AT-START-WB415 Board
2) Smartphone with LightBlue APP
3) Micro USB cable

图 22. AT-START-WB415 Board



4.2 软件资源

4.2.1 MCU 执行操作

IO 控制及数据读取是指对 MCU 上的外设资源进行操作，在代码中需要由用户先初始化，并编写收到命令后该执行的功能，本应用笔记以 GPIO 的控制为例，用户可以依此架构进行后续的开发。

1. 首先配置相应的 GPIO，这里使用 AT-START-WB415 上的 LED2(PB7)作为被控制引脚

图 23. 初始化 LED 函数

```
/**
  * @brief  configure led gpio
  * @param  led: specifies the led to be configured.
  * @retval none
  */
void at32_led_init(led_type led)
{
  gpio_init_type gpio_init_struct;

  /* enable the led clock */
  crm_periph_clock_enable(led_gpio_crm_clk[led], TRUE);

  /* set default parameter */
  gpio_default_para_init(&gpio_init_struct);

  /* configure the led gpio */
  gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
  gpio_init_struct.gpio_out_type  = GPIO_OUTPUT_PUSH_PULL;
  gpio_init_struct.gpio_mode = GPIO_MODE_OUTPUT;
  gpio_init_struct.gpio_pins = led_gpio_pin[led];
  gpio_init_struct.gpio_pull = GPIO_PULL_NONE;
  gpio_init(led_gpio_port[led], &gpio_init_struct);
}
```

2. 编写读出及写入 LED 的代码。

图 24. 写入 LED

```
void at32_led_on(led_type led)
{
  if(led > (LED_NUM - 1))
    return;
  if(led_gpio_pin[led])
    led_gpio_port[led]->clr = led_gpio_pin[led];
}

void at32_led_off(led_type led)
{
  if(led > (LED_NUM - 1))
    return;
  if(led_gpio_pin[led])
    led_gpio_port[led]->scr = led_gpio_pin[led];
}
```

图 25. 读出 LED

```
203  flag_status gpio_input_data_bit_read(gpio_type *gpio_x, uint16_t pins)
204  {
205    flag_status status = RESET;
206
207    if(pins != (pins & gpio_x->idt))
208    {
209      status = RESET;
210    }
211    else
212    {
213      status = SET;
214    }
215
216    return status;
217  }
```

3. 在主循环中调用 at_cmd_handler 函数，来完成对 AT Command 的解析，以及针对不同命令执行相应的操作。

图 26. 调用写入及读出 GPIO 的函数

```
void at_cmd_handler(void)
{
  uint8_t msg_id = SIZEOFMSG-1, i;
  if(recv_cmp_flag == SET)
  {
    for(i = 0; i <= SIZEOFMSG; i++)
    {
      if(memcmp(recv_data, at_cmd_list[i].at_cmd_string, strlen(recv_data)) == 0)
      {
        msg_id = i;
        break;
      }
    }

    switch(at_cmd_list[msg_id].msg_id)
    {
      case AT_CMD_IOSET0:
      {
        printf("AT_CMD_IOSET0\r\n");
        at32_led_off(LED2);
        at_cmd_send(AT_RESULT_OK0);
        break;
      }
      case AT_CMD_IOSET1:
      {
        printf("AT_CMD_IOSET1\r\n");
        at32_led_on(LED2);
        at_cmd_send(AT_RESULT_OK1);
        break;
      }
      case AT_CMD_IOGET:
      {
        printf("AT_CMD_IOGET\r\n");
        if(gpio_output_data_bit_read(GPIOB, GPIO_PINS_7))
        {
          at_cmd_send(AT_RESULT_OK1);
        }
        else
        {
          at_cmd_send(AT_RESULT_OK0);
        }
        break;
      }
      default:
      {
        printf("AT_CMD_ERROR\r\n");
        at_cmd_send(AT_RSP_ERROR);
        break;
      }
    }
    recv_cmp_flag = RESET;
    memset(recv_data, 0, strlen(recv_data));
  }
}
```

4.2.2 无线蓝牙模块接收请求

蓝牙端的命令处理，主要是靠 app.c 中 app_user_entry()这个函数，在 uart_rx_done 这个 flag 立起之后，会进 at_result_to_prefix()进行解析，判断收到的数据是不是 AT command 及判断对应的命令号，之后再根据命令号进入符合的 case，执行相应的请求事件后，再回响应给 MCU 端。

图 27. app_user_entry()在 main loop 中被轮询

```
while(1)
{
  //schedule all pending events
  rwip_schedule();
  app_user_entry();

  // Checks for sleep have to be done with interrupt disabled
  GLOBAL_INT_DISABLE();

  oad_updating_user_section_pro();

  if(wdt_disable_flag==1)
  {
    wdt_disable();
  }
```

图 28. 解析收到的数据

```
if(uart_rx_done == 1)
{
    // uint8_t baud_change = 0;
    uint8_t len;
    uint8_t rsp_code;
    //uint8_t idx;
    extern uint8_t rxdata_buffer_len;
    at_prefix_t *prefix_cmd;
    uint8_t w_flash_buf[2];
    //len = strlen((char*)rxdata_buffer);
    len = rxdata_buffer_len;
    rxdata_buffer_len = 0;
    if(rxdata_buffer[len-1] == '\n')
    {
        //AT command finish
        //UART_PRINTF("finish\r\n");
        memcpy(&AT_cmd_buf[recv_AT_cmd_idx],rxdata_buffer,len);
        //UART_PRINTF("%s\r\n",AT_cmd_buf);
        AT_cmd_len += len;
        recv_AT_cmd_idx = 0;
    }
    else
    {
        //command not finish
        memcpy(&AT_cmd_buf[recv_AT_cmd_idx],rxdata_buffer,len);
        recv_AT_cmd_idx = len;
        AT_cmd_len += len;
        uart_rx_done = 0;
        //UART_PRINTF("not finish\r\n");
        return;

    }

    //dispatch AT-COMMAND
    prefix_cmd = at_result_to_prefix((char*)AT_cmd_buf, AT_cmd_len);
    uart_rx_done = 0;
    without_prefix_len = AT_cmd_len-prefix_cmd->prefix_len;
```

图 29. 选择符合的 case 执行并回复响应

```
switch(prefix_cmd->code)
{
    case AT_RESULT_AT :
         //do nothing

        #ifdef used_BK3432_MCU
        UART_SEND_DATA("@");
        #endif
        UART_SEND_DATA("%s\r\n",get_at_rsp(rsp_code));
    break;
    case AT_RESULT_BAUD1 :
         UART_PRINTF("recv AT_RESULT_BAUD1\r\n");
         #ifdef used_BK3432_MCU
         UART_SEND_DATA("@");
         #endif
         UART_SEND_DATA("%s\r\n",get_at_rsp(rsp_code));
         cpu_delay(15);
         uart_init(9600);
         w_flash_buf[0] = 1;
         save_parameter_to_BK3432_USED_FLASH_AREA(TAG_BAUD,w_flash_buf);


    break;
    case AT_RESULT_BAUD2 :
         #ifdef used_BK3432_MCU
         UART_SEND_DATA("@");
         #endif
         UART_SEND_DATA("%s\r\n",get_at_rsp(rsp_code));
         cpu_delay(15);
         w_flash_buf[0] = 2;
         save_parameter_to_BK3432_USED_FLASH_AREA(TAG_BAUD,w_flash_buf);
         uart_init(19200);

    break;
    case AT_RESULT_BAUD3 :
        #ifdef used_BK3432_MCU
         UART_SEND_DATA("@");
        #endif
        UART_SEND_DATA("%s\r\n",get_at_rsp(rsp_code));
        cpu_delay(15);
        w_flash_buf[0] = 3;
        save_parameter_to_BK3432_USED_FLASH_AREA(TAG_BAUD,w_flash_buf);
        uart_init(38400);

    break;
```

4.2.3 无线蓝牙模块发送请求

发送请求的部份，主要是根据 charateristic 的实作来添加，在本应用中是读出及写入远端 IO 的电平，两者都是透过 UART_SEND_DATA()这个函数发送 AT command 给 MCU，但后续的作法不太一样，因为在本应用中，写入在 Profile 的设定是 Write without response，所以不用去等待 MCU 响应结果；但读出的话因为需要把值添加到 GATT 的 response 之中，所以必须等待 MCU 端响应，在代码中是使用 at_wait_for_rsp()函数等待并获取响应的数据。在得到 MCU 端数据后通过 ke_msg_send()函数将数据回传至手机端。

图 30. 发送写入 IO 的命令

```
static int gattc_write_req_ind_handler(ke_msg_id_t const msgid, struct gattc_write_req_ind const *param,
                                       ke_task_id_t const dest_id, ke_task_id_t const src_id)
{
    struct gattc_write_cfm * cfm;
    uint8_t status = GAP_ERR_NO_ERROR;
    //Send AT command
    UART_SEND_DATA(AT_CMD_IO_SET, param->value[0]);

    cfm = KE_MSG_ALLOC(GATTC_WRITE_CFM, src_id, dest_id, gattc_write_cfm);
    cfm->handle = param->handle;
    cfm->status = status;
    ke_msg_send(cfm);

    return (KE_MSG_CONSUMED);
}
```

图 31. 发送读出 IO 的命令并回传数据

```
static int custom_value_req_ind_handler(ke_msg_id_t const msgid,
                                        struct custom_value_req_ind const *param,
                                        ke_task_id_t const dest_id,
                                        ke_task_id_t const src_id)
{
    // Initialize length
    uint8_t len = 0;
    // Pointer to the data
    uint8_t *data = NULL;

    at_rsp_content* rsp_content;
    //rxdata_buffer_len = 0;
    // Check requested value
    switch (param->value)
    {
        case CUSTOM_REMOTE_IO_STATUS:
        {
            // AT command
            UART_SEND_DATA(AT_CMD_IO_GET);
            // Wait for response
            rsp_content = at_wait_for_rsp();
            // Set information
            len = APP_CUSTOM_REMOTE_IO_LEN;
            if(rsp_content->data[4] == 0x31)
            {
              data = (uint8_t *)APP_CUSTOM_REMOTE_IO_HIGH;
            }
            else
            {
              data = (uint8_t *)APP_CUSTOM_REMOTE_IO_LOW;
            }
        } break;

        default:
            ASSERT_ERR(0);
            break;
    }

    // Allocate confirmation to send the value
    struct custom_value_cfm *cfm_value = KE_MSG_ALLOC_DYN(CUSTOM_VALUE_CFM,
            src_id, dest_id,
            custom_value_cfm,
            len);

    // Set parameters
    cfm_value->value = param->value;
    cfm_value->length = len;
    if (len)
    {
        // Copy data
        memcpy(&cfm_value->data[0], data, len);
    }
    // Send message
    ke_msg_send(cfm_value);

    return (KE_MSG_CONSUMED);
}
```

4.2.4 软件下载

将蓝牙部份及 MCU 部份的代码编译好之后，可以通过 ICP Tool download 到 WB415 开发板，这里需要导入两个文件，分别是 BLE 端代码 wb415_ble_app_merge.bin 和 MCU 端代码 Template.hex，下载流程如下：

1. 通过 USB 将 AT-Link 连接至 PC
2. 打开上位机软件 Artery ICP Programmer Tool，并连接 AT32WB415 芯片
3. 选择 BLE 端代码，在文件信息栏目点击添加按钮，选择欲下载的文档，BLE 端编译后的默认路径在工程项目的 output 文件夹中的 app 文件夹，选择文档 wb415_ble_app_merge.bin，并填入下载起始地址 0x00000000
4. 选择 MCU 端代码，同样在文件信息栏目点击添加按钮，选择欲下载的文档，MCU 端编译后的默认路径在工程项目的 Objects 文件夹中，选择文档 Template.hex
5. 点选下载后，再点击开始下载
6. 完成后，上位机软件会提示下载&校验完成信息

图 32. 上位机软件连接 AT32WB415 芯片

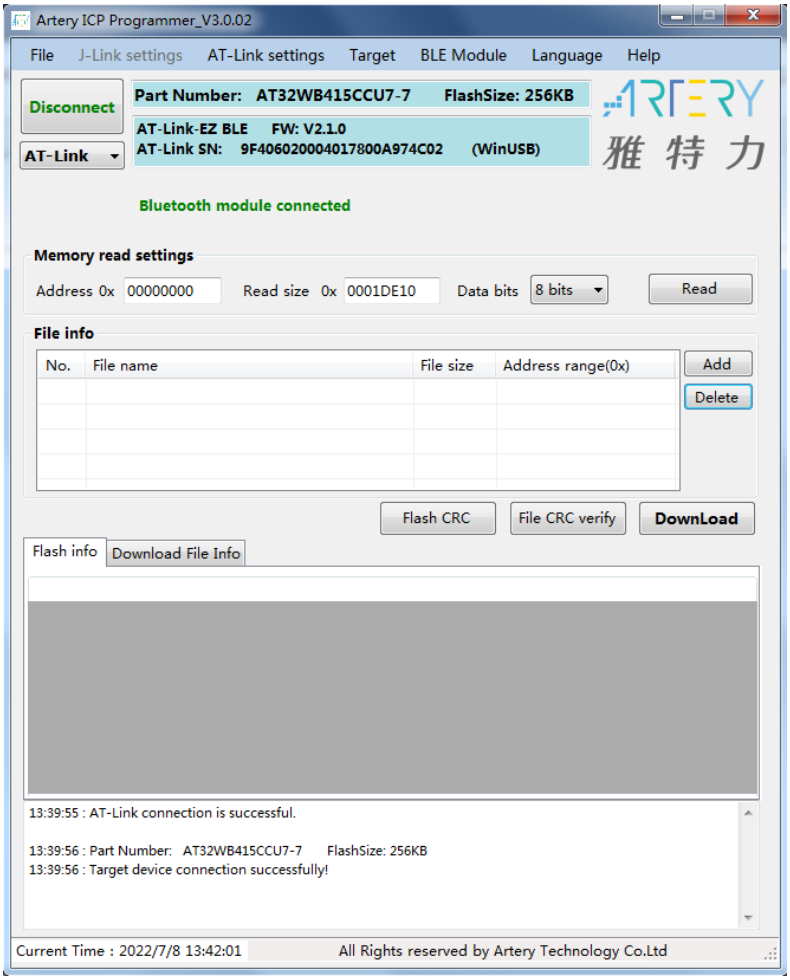


图 33. 点击添加 BLE 文件

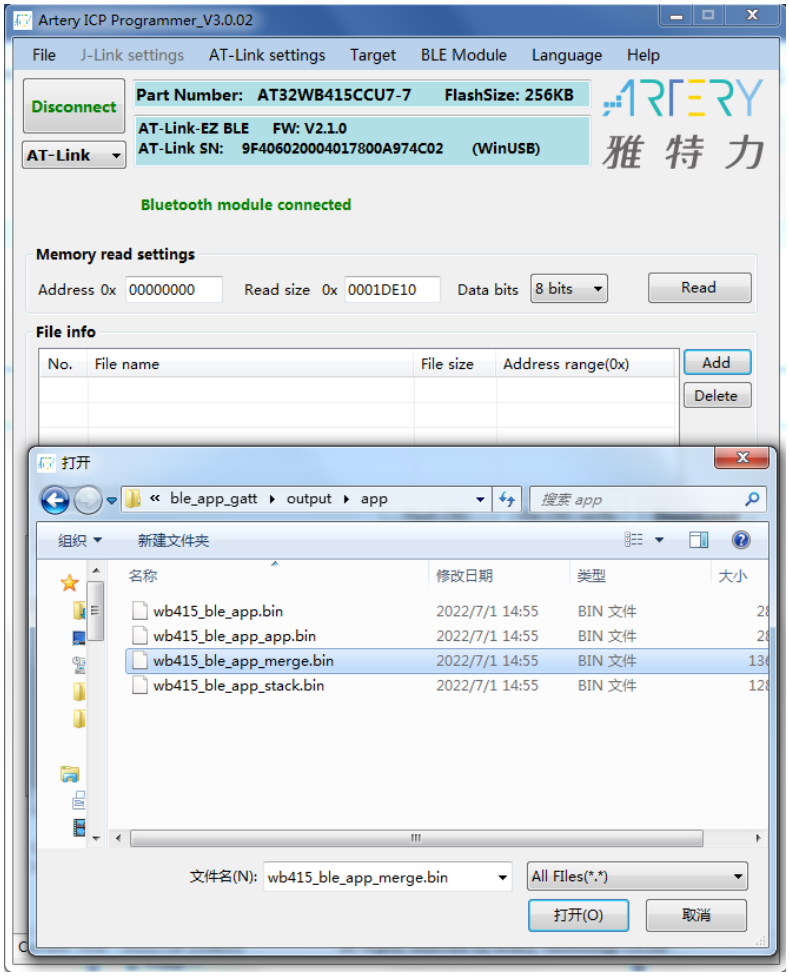


图 34. 修改 BLE 下载起始地址

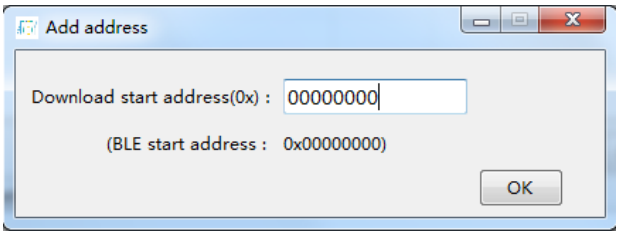


图 35. 点击添加 MCU 文件

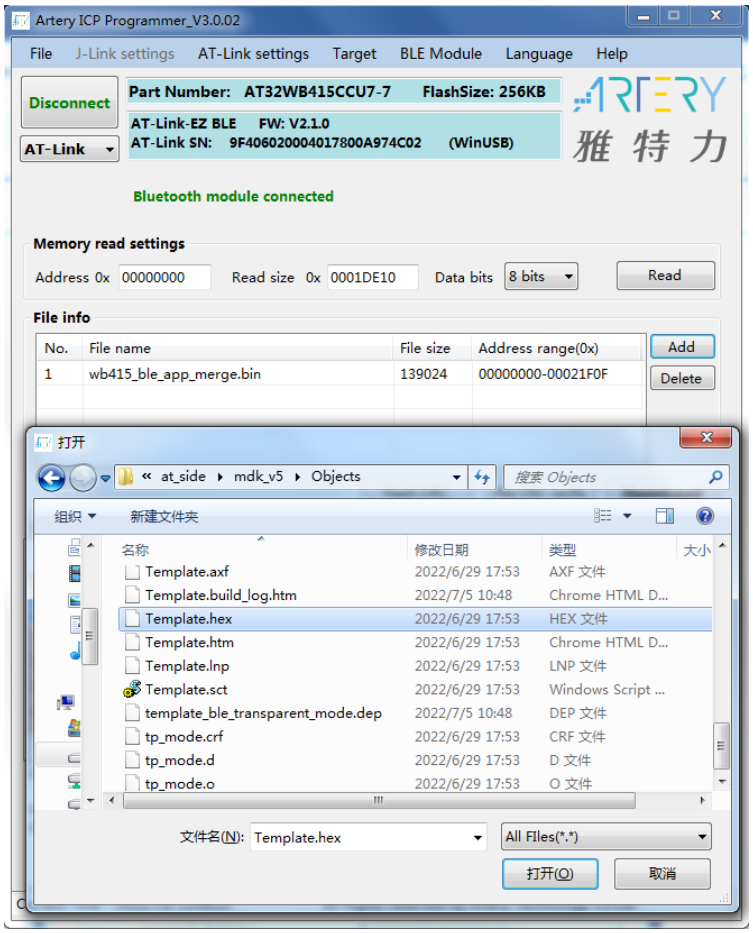


图 36. 点击下载-开始下载

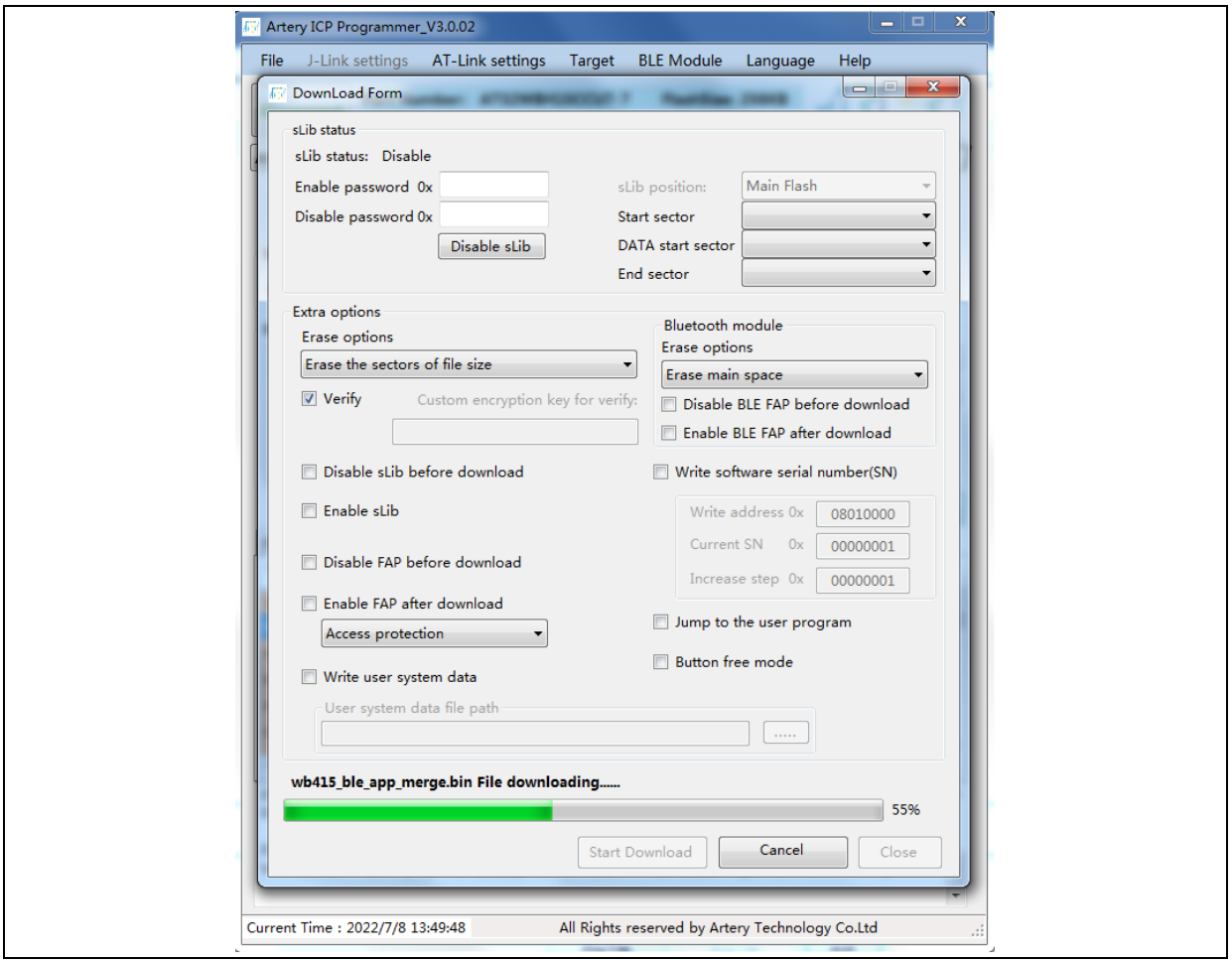
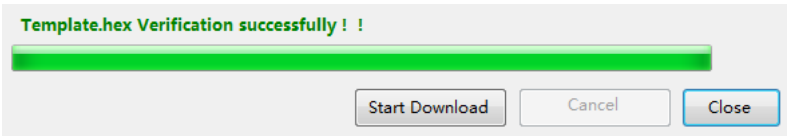


图 37. 下载&校验成功



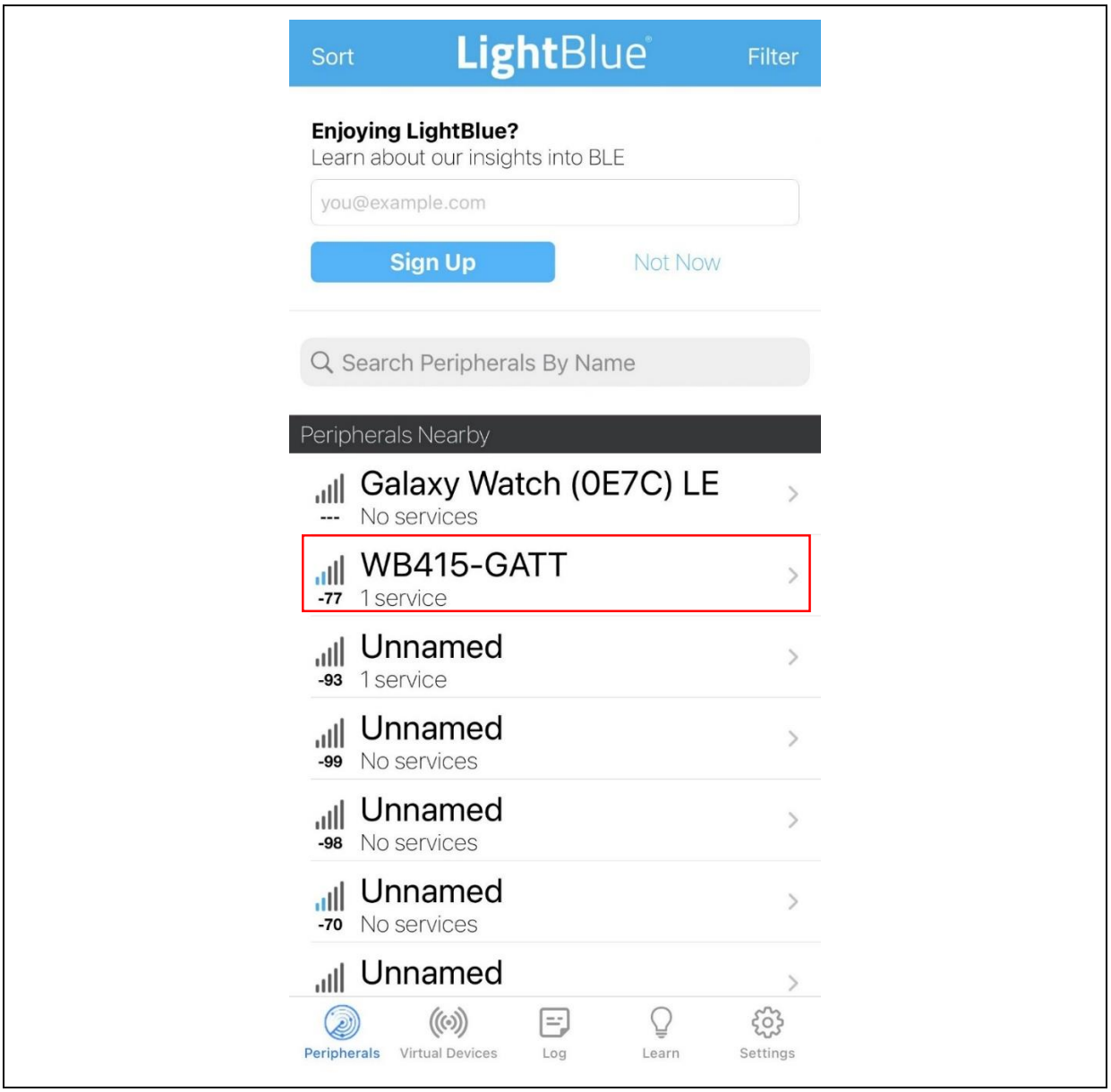
4.3 AT command 模式

为了方便对 BLE 进行操作，需要在手机端安装具有蓝牙设备操作功能的蓝牙工具软件，这里以 LightBlue APP 为例。

通过以下几个步骤即可验证此案例中的 AT command 模式是否正常工作：

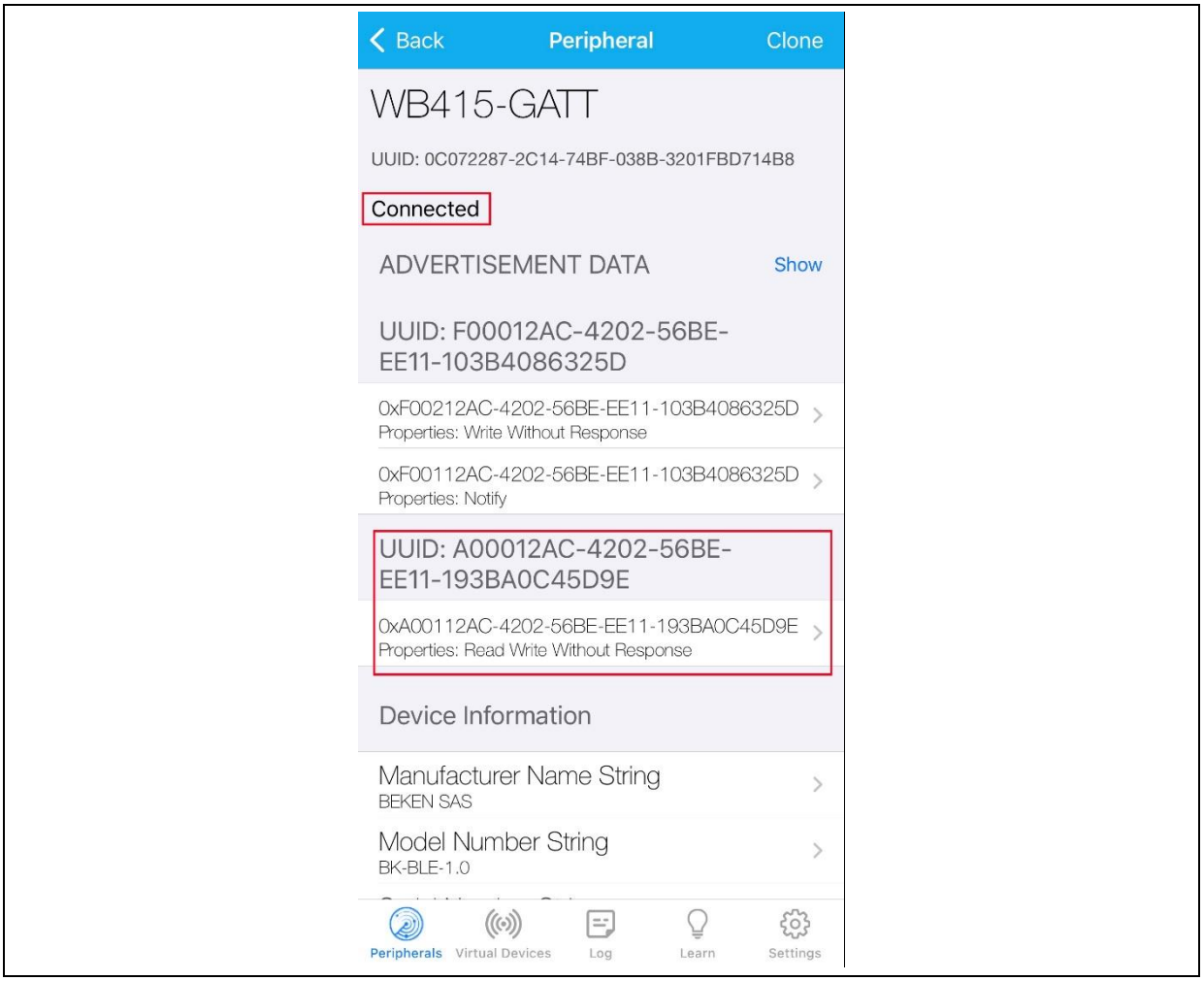
1. 打开 LightBlue APP，找到名为 WB415-GATT 的蓝牙设备，并连接。

图 38. 查找 WB415-GATT



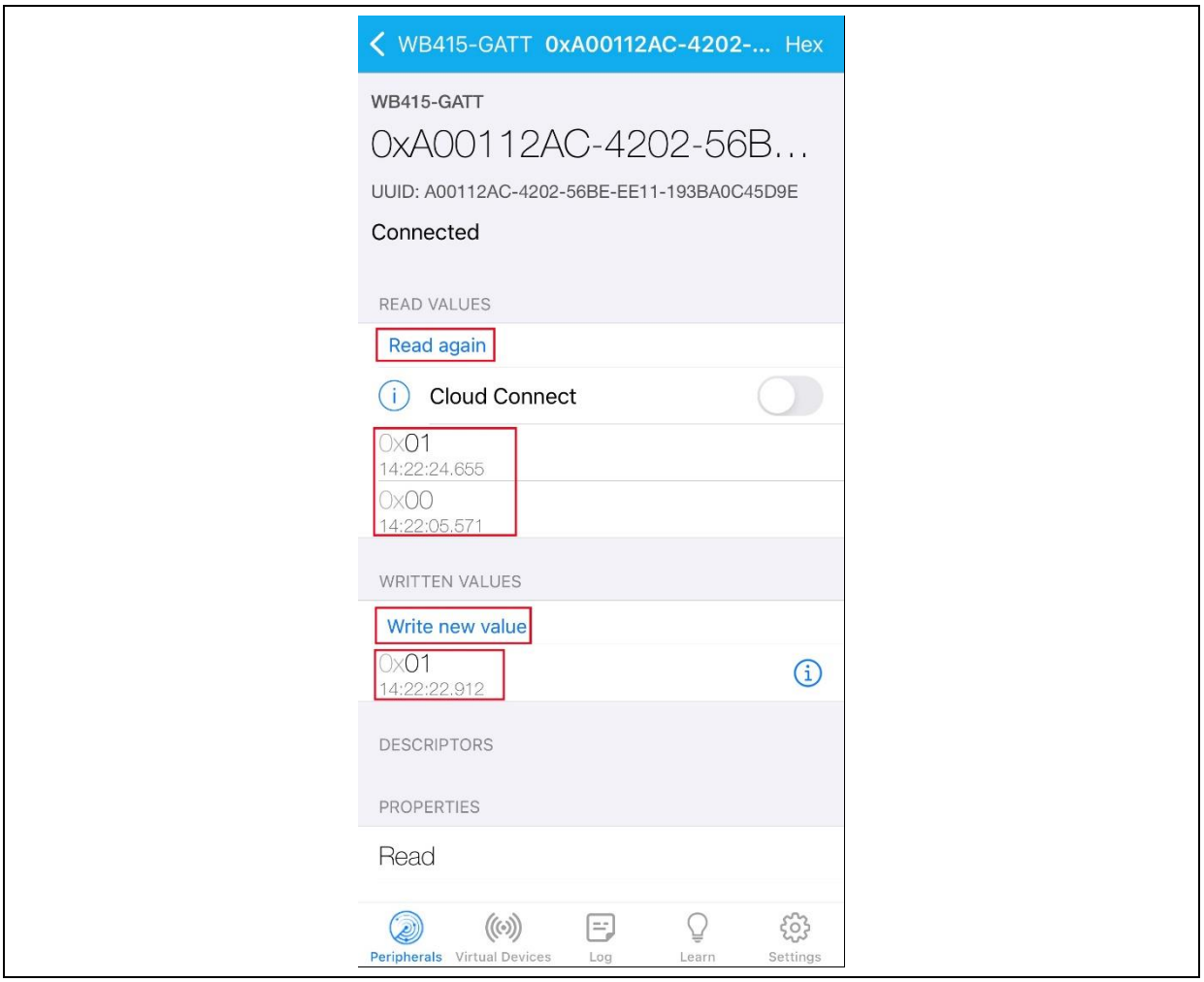
2. 查看连接状态为 connected，点击 UUID 为 0xA00012AC-4202-56BE-EE11-193BA0C45D9E 的服务中 0xA00112AC-4202-56BE-EE11-193BA0C45D9E 这个特征，这是 AT command 模式所用到的服务及特征。

图 39. 连接状态以及 0xA00112AC-4202-56BE-EE11-193BA0C45D9E 特征



3. 可以看到其中有两个功能可以使用，分别是 READ VALUES 和 WRITTEN VALUES。
4. 点击 Read again 按钮，获取 IO 状态数据，返回的数据为 0x00 或 0x01，分别代表 LED 的 off 及 on，用以表示读取 IO 状态的高电平或低电平。
5. 点击 Write new value 按钮，写入 0 或 1 将配置 IO 状态为低电平或高电平，可在 AT-START-WB415 板上看到 LED2 的两种状态，电路设计为低电平点亮 LED2。

图 40. 读写 IO 数据



4.4 透传模式

透传命令简化了用户的开发流程，用户不必自己去实作服务与特征，只需要专注在 MCU 端的应用开发，如果需要其他功能，可以通过定义透传数据的格式来实现，在透传模式下每笔数据的末端不需要加上 CR + LF。本应用笔记提供两种接口的透传模式，其一是透过 WB415 的 USART2 与手机应用对接，其二是使用 USB 接口，实做 custom HID，以下分别对两种透传模式进行说明：

4.4.1 UART 界面

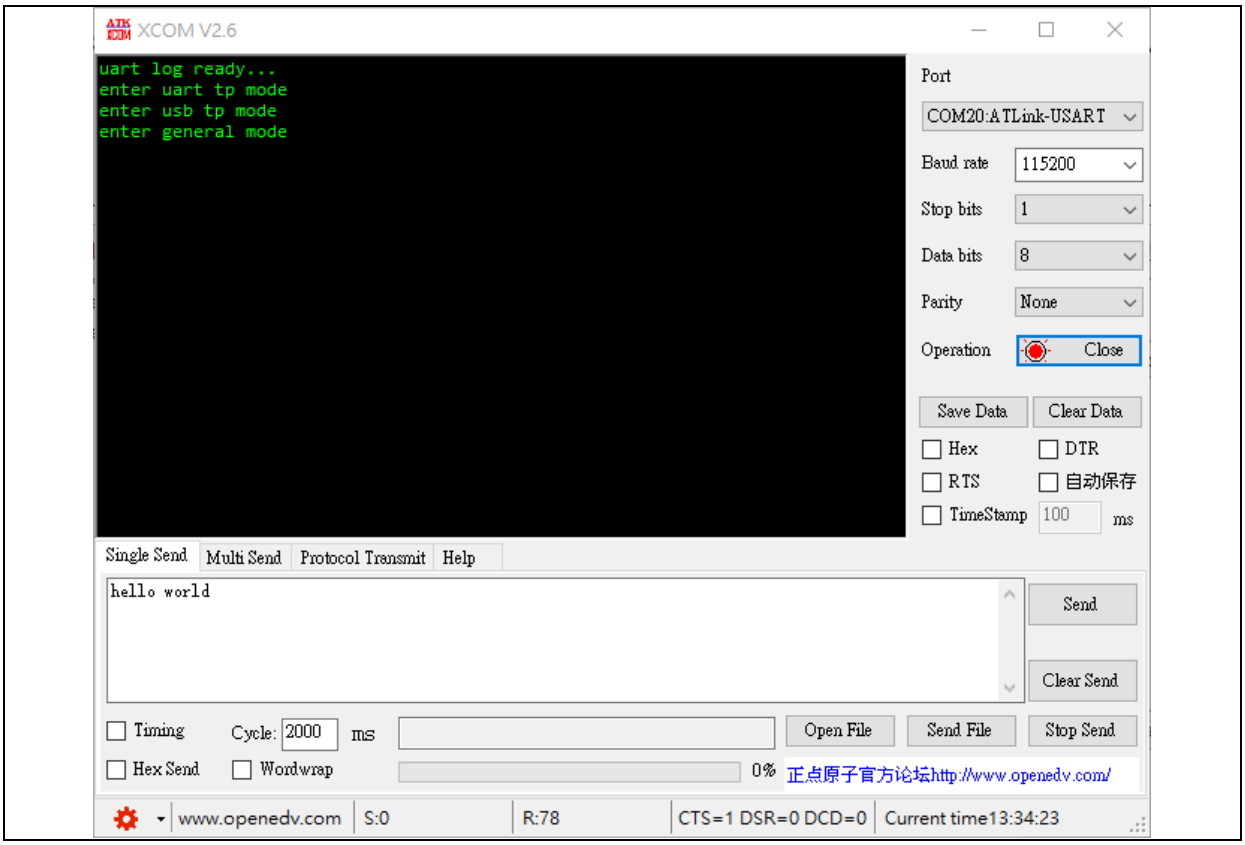
1. AT-START-WB415 上的 USER key 用于切换 AT command 模式和透传模式，透传模式分为 UART 接口和 USB 接口，会透过 USART2_TX(PA2)打印出当前模式，LED3 也能指示当前模式，AT command 模式时 LED3 熄灭，透传模式时 LED3 点亮。

注意：透传模式会与 AT command 模式冲突，故进入透传模式后自定义服务将无法使用。

2. 按下 WB415 上的 USER key，进入透传模式，LED3 会被点亮，也可以透过 USART2 打印的信

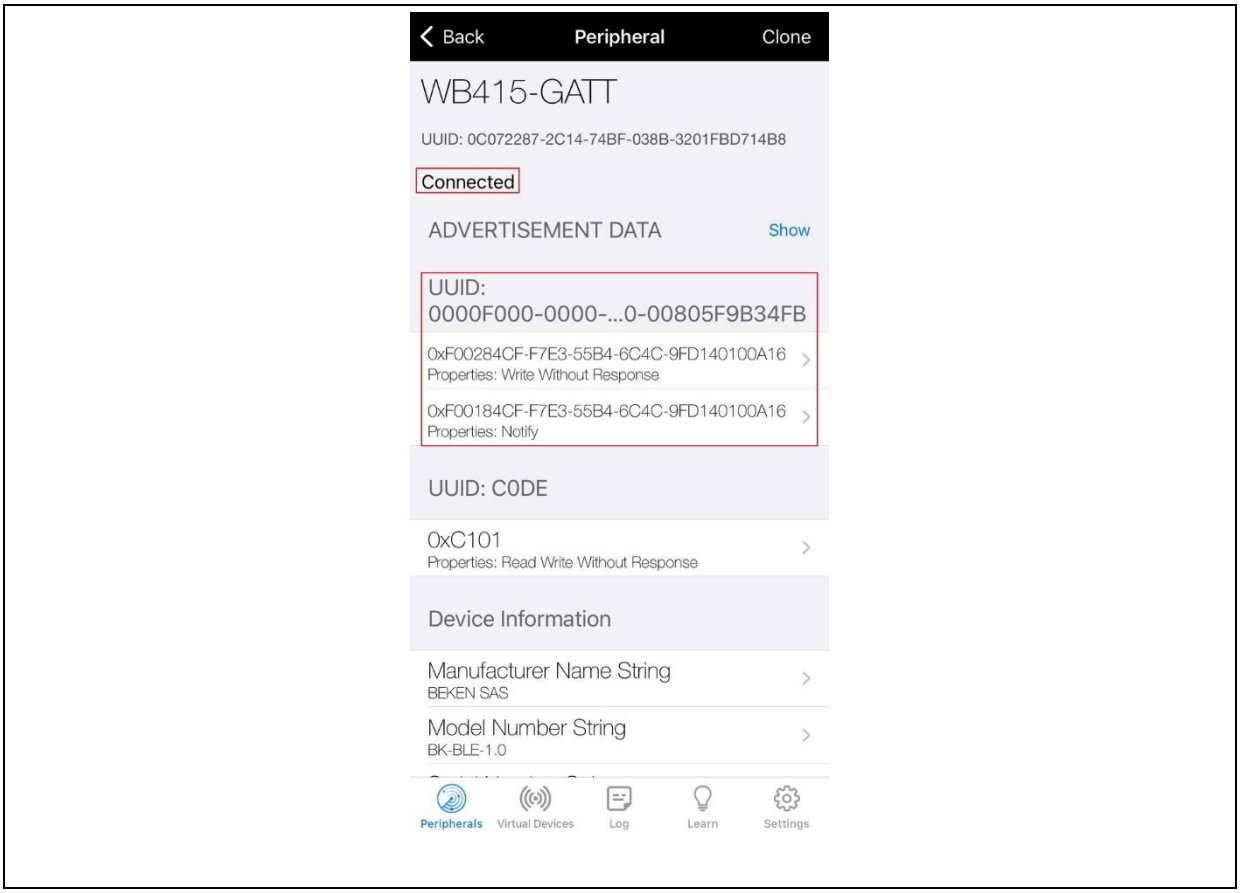
息得知现在是否进入透传模式。

图 41. 切换透传模式



3. 透过 LightBlue 连上 WB415 后可以看到 F000 的服务，里面有 F001 跟 F002 两个特征，这是透传模式会使用到的服务及特征。

图 42. LightBlue 连接 WB415



4. 使用透传模式传数据到 MCU 端，进入 0xF002 这个特征，将右上角的数据模式切换成 UTF-8 String，然后点击 Write new value 输入任意字符串，该字符串会透过 WB415 的 USART2 TX 输出到串口助手上。

图 43. LightBlue 写入数据

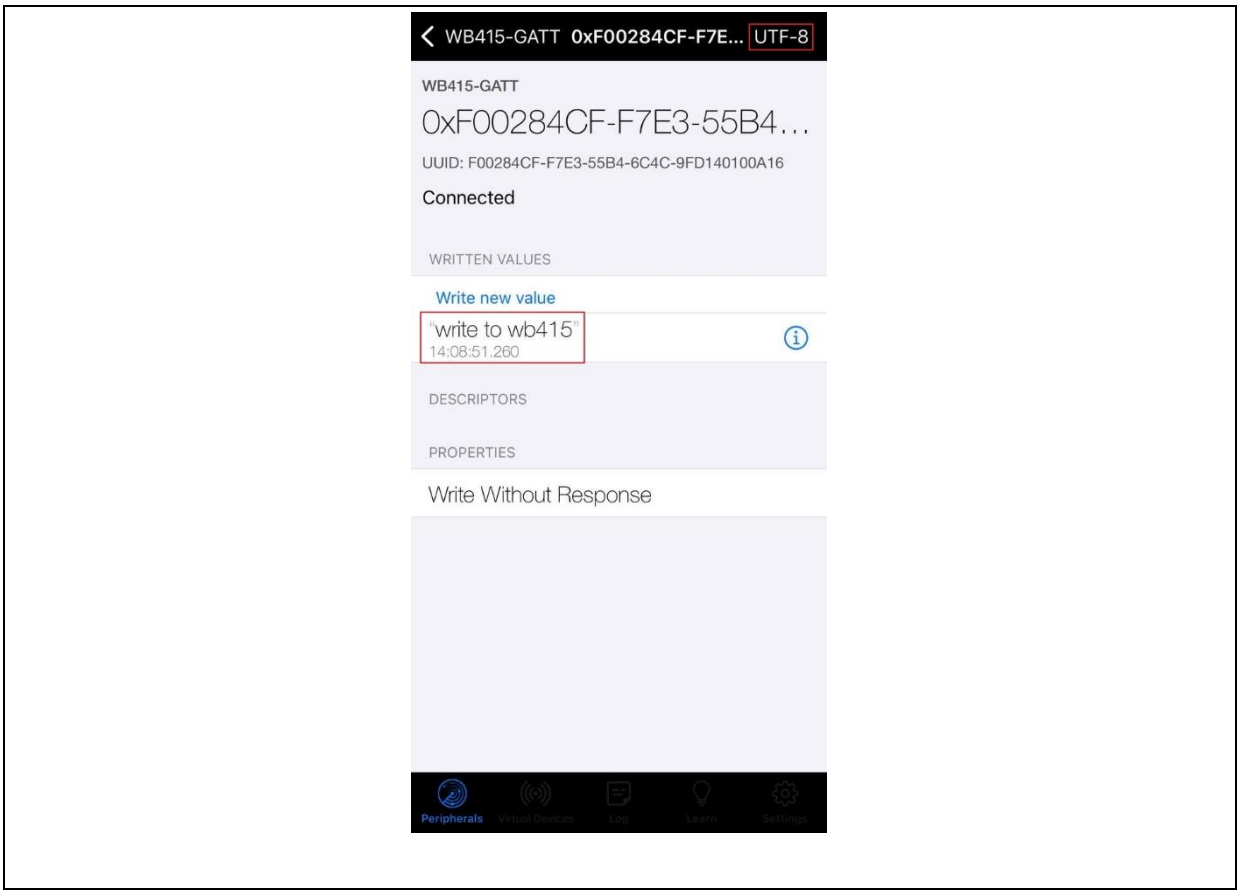
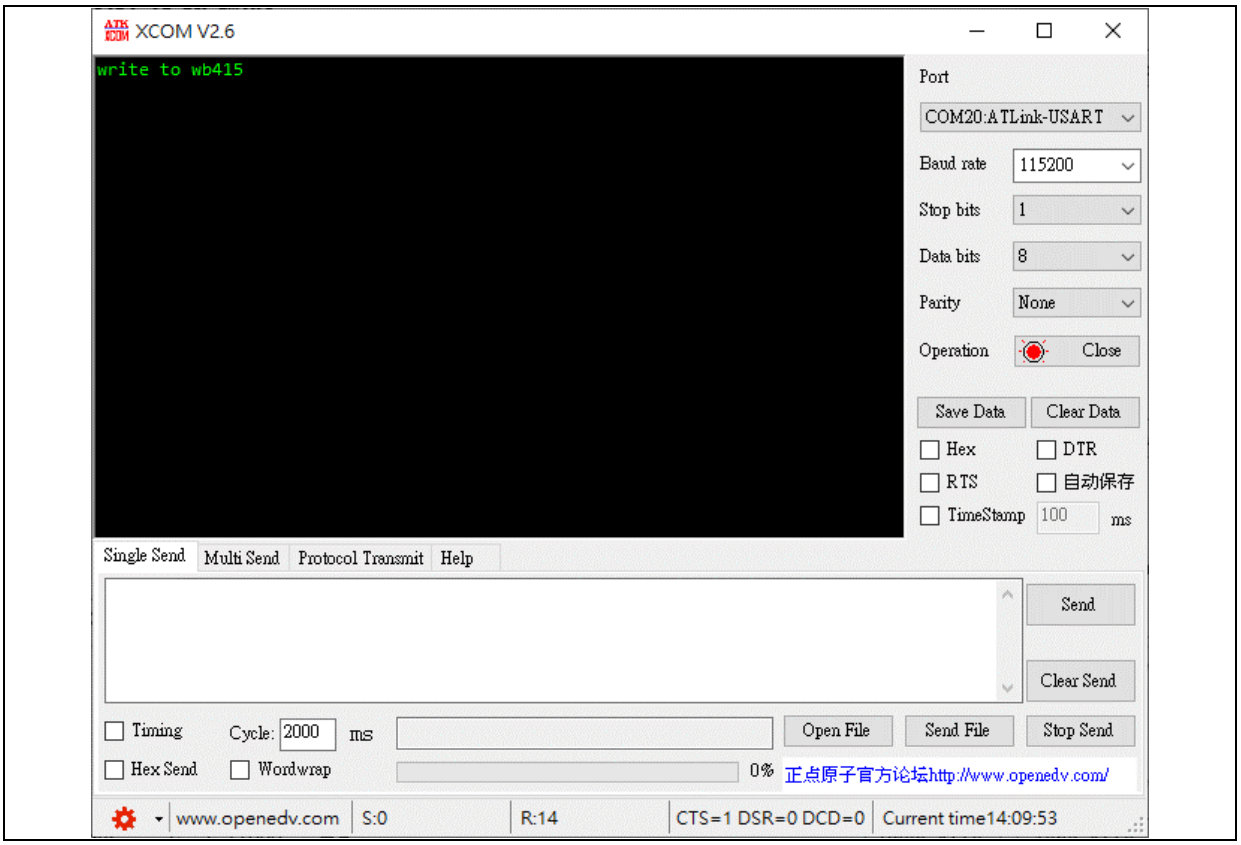


图 44. WB415 打印接收到的数据



5. 使用透传模式传数据到手机端，进入 0xF001 这个特征，按下 Listen for notifications，收到的第一笔数据会是”Notification Start”，然后在串口助手输入完字符串后按下发送，0xF001 即会显示串口助手输入的字符串。

图 45. 输入数据至 WB415

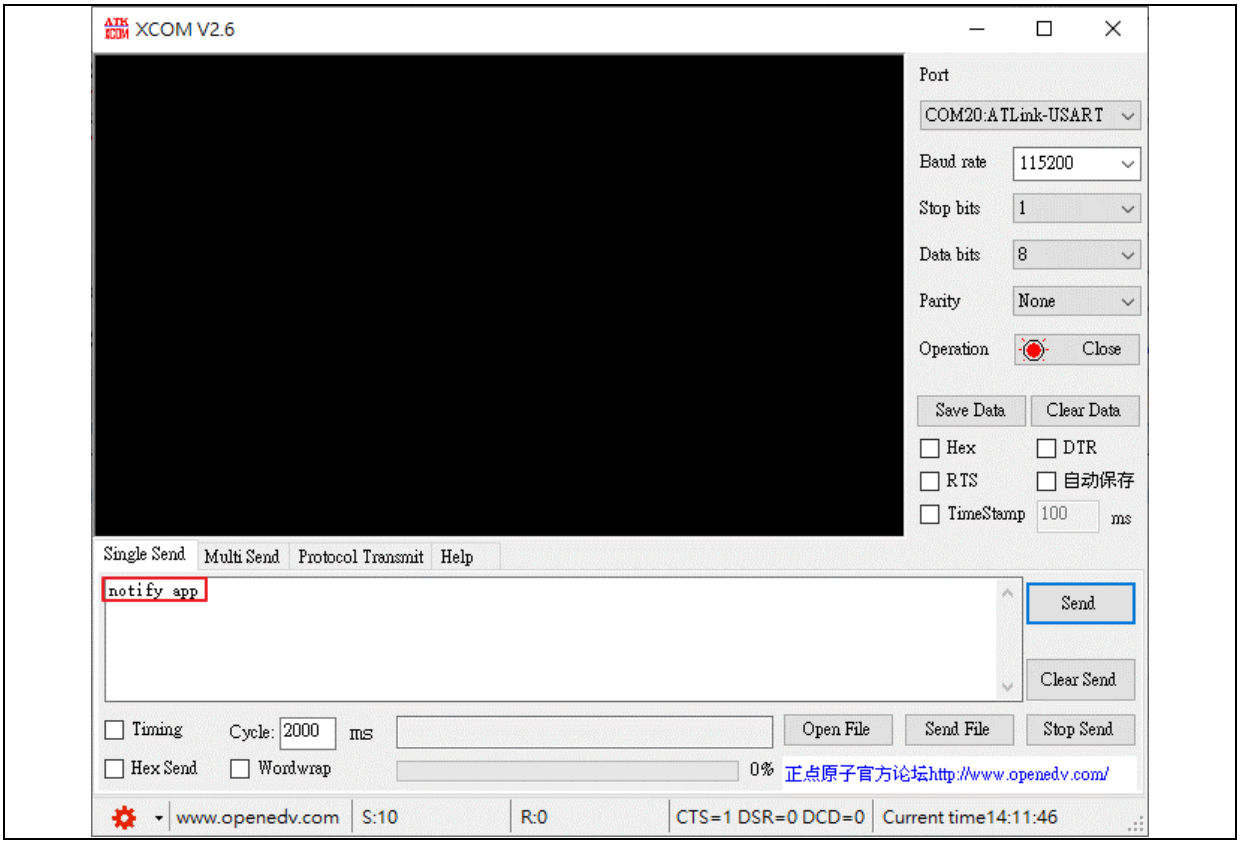
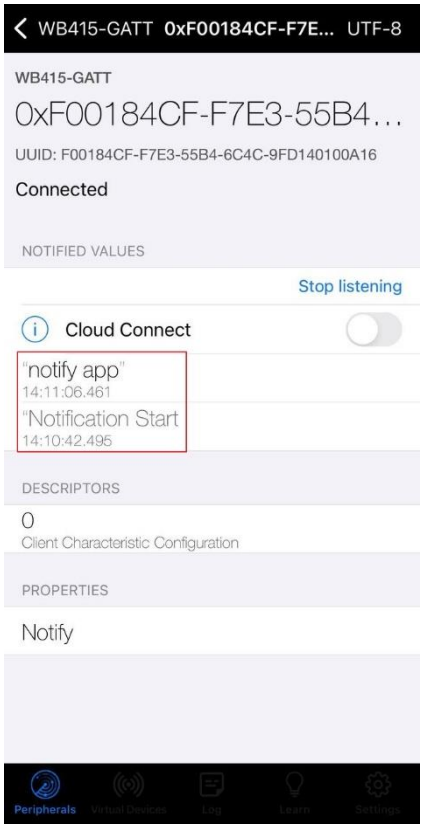


图 46. LightBlue 接收来自 WB415 的数据

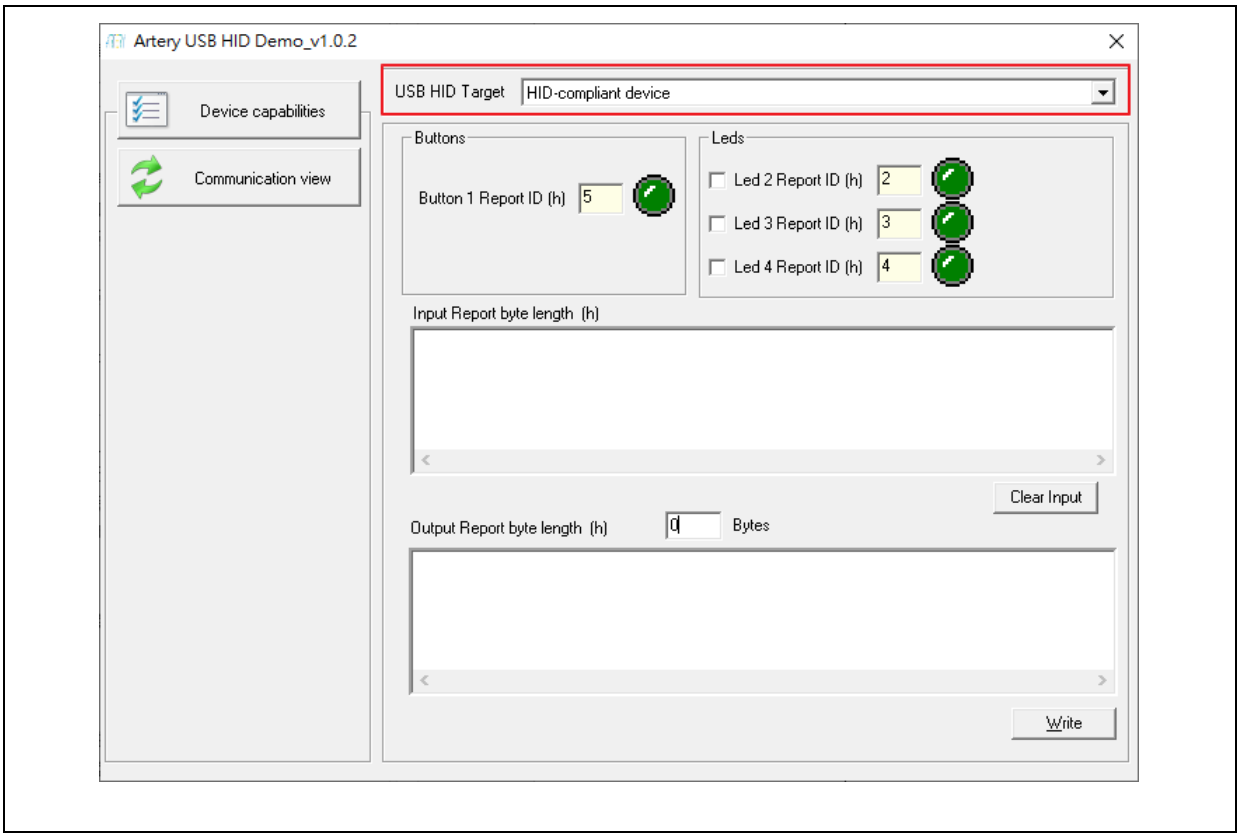


4.4.2 USB 界面

USB 透传模式的 demo 是基于 BSP 中的 custom HID 的 demo 修改而成，细节可以参考 AN0097。在 BT 端完全与 UART 透传模式相同，使用 Artery USB HID Demo 的上位机与 WB415 对接，在收发数据一样使用 0xF001 及 0xF002 两个特征，以下呈现操作的流程：

1. 将 WB415 切换至 USB 透传模式，按压 USER key 可以在 UART, USB 及 General mode 之间切换
2. 将 USB 线接上 WB415 的 USB 孔后，开启上位机工具并选择 USB HID target

图 47. 选择 USB HID Target



3. 发送数据给 APP 前，需要先填入数据长度，然后填入数据后按下 Write，在手机应用上查看 0xF001 的特征，即可以看到接收到的数据

图 48. 填入数据发送给 APP

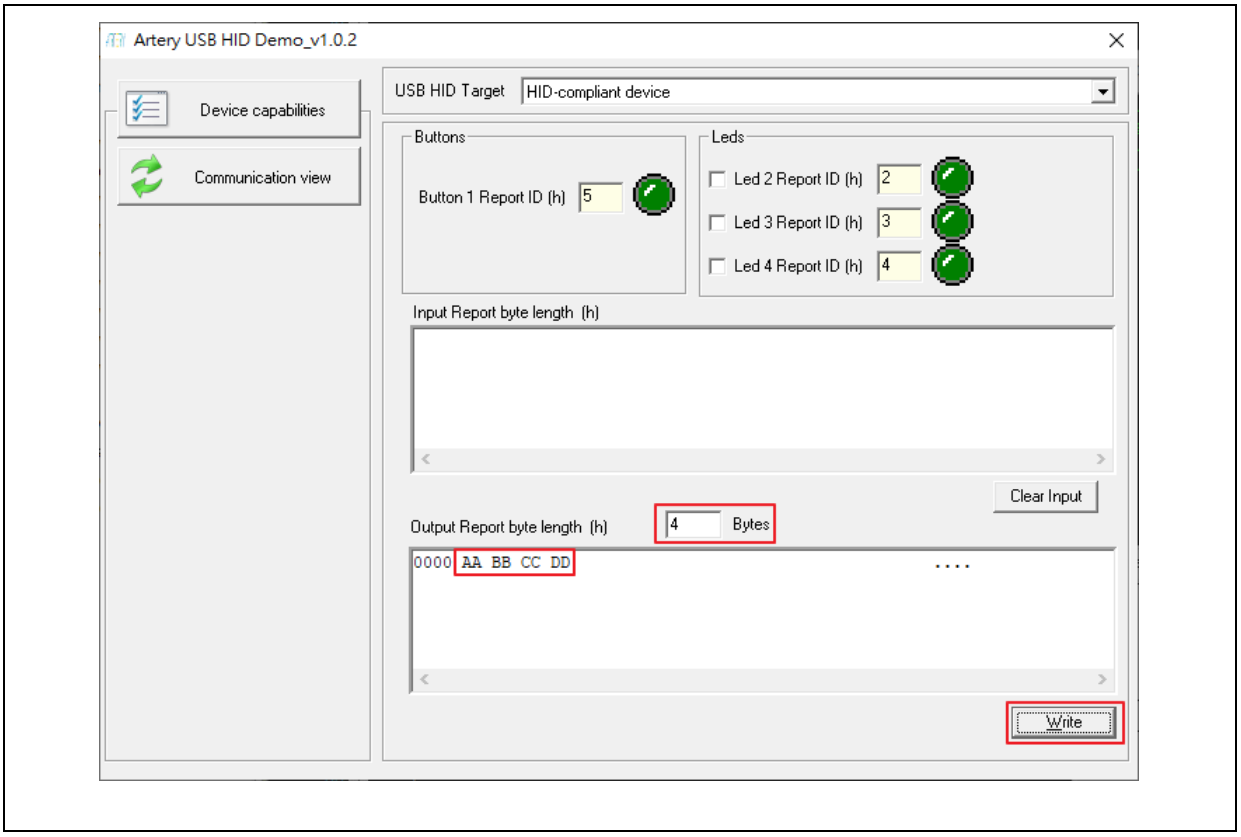
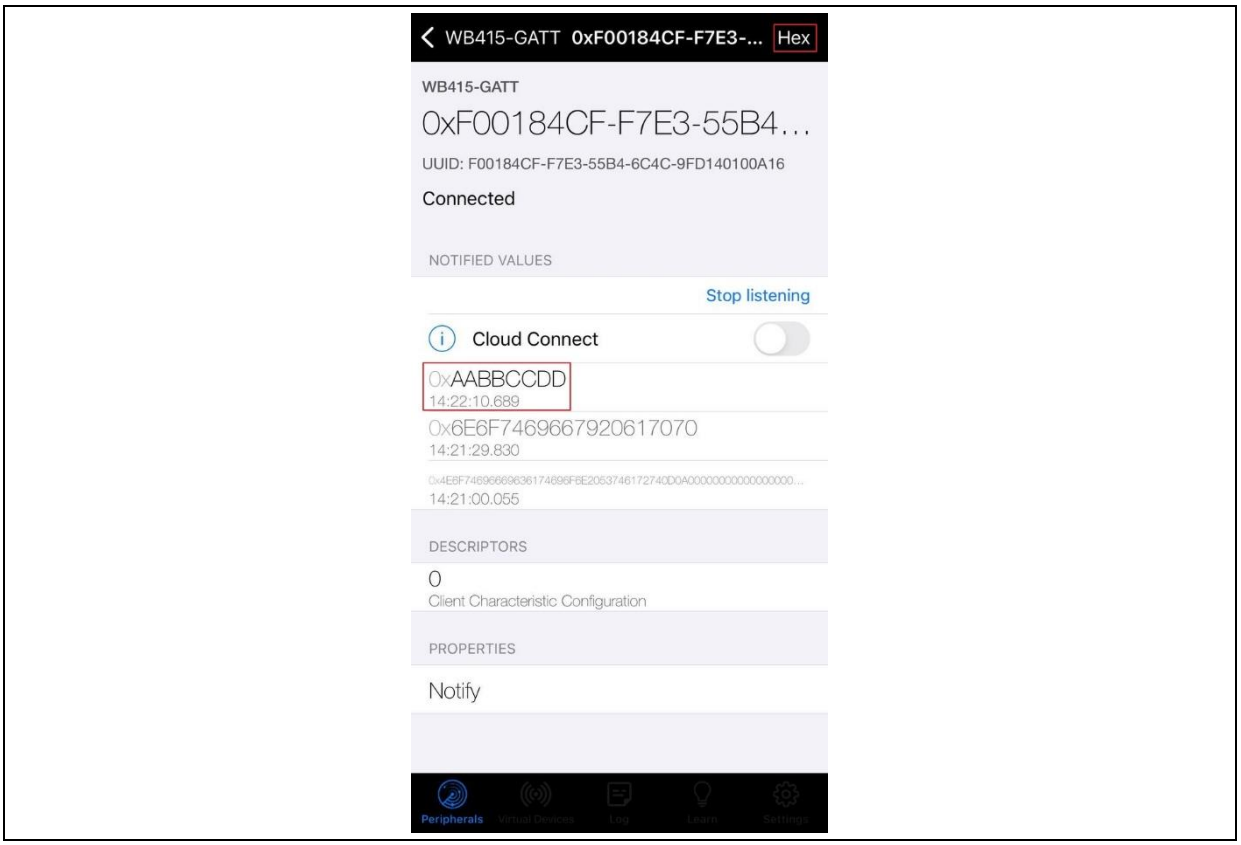


图 49. 手机应用显示出接收到的数据



4. 手机应用发数据给 USB 上位机和 UART 透传模式相同，找到 0xF002 特征并写入数据，即可以在 USB 上位机的 Input Report 字段看到手机发出的数据

图 50. 发送数据到 USB 上位机

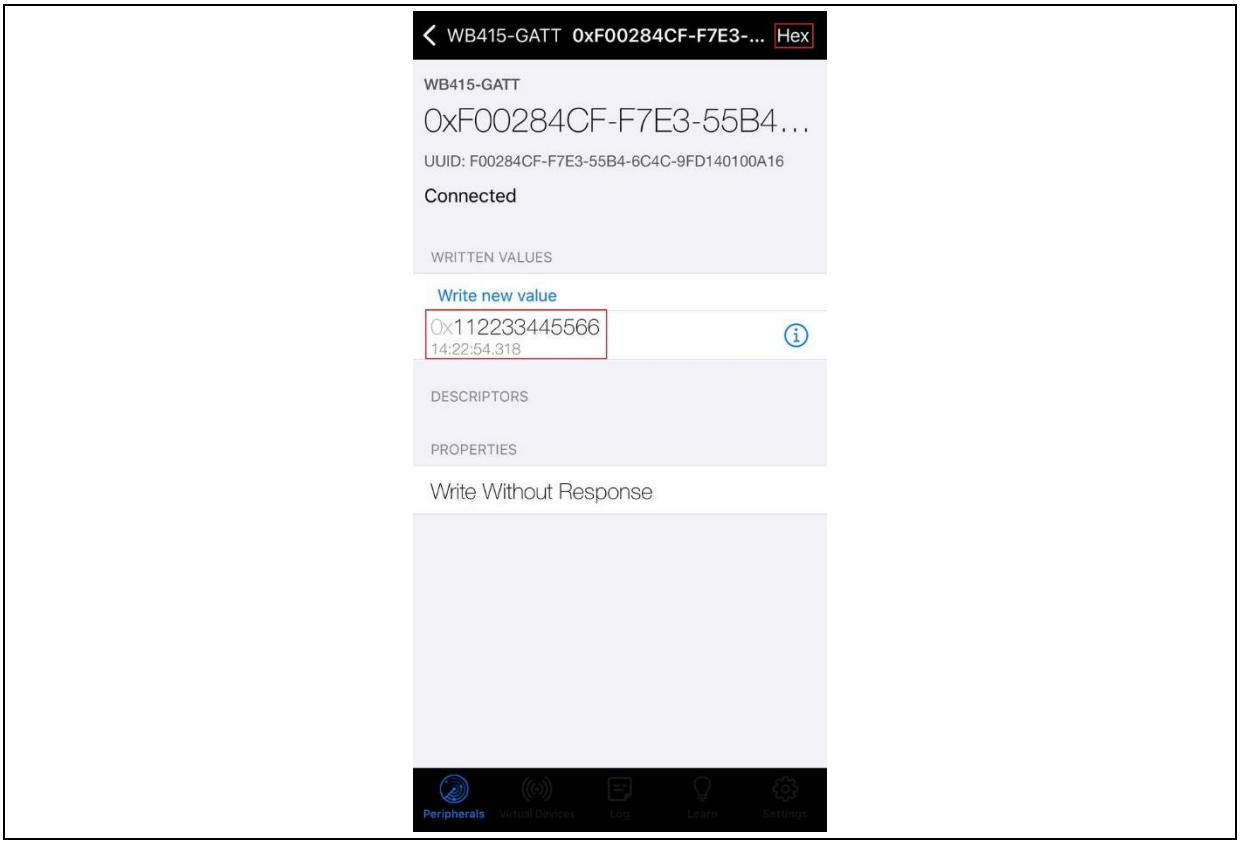
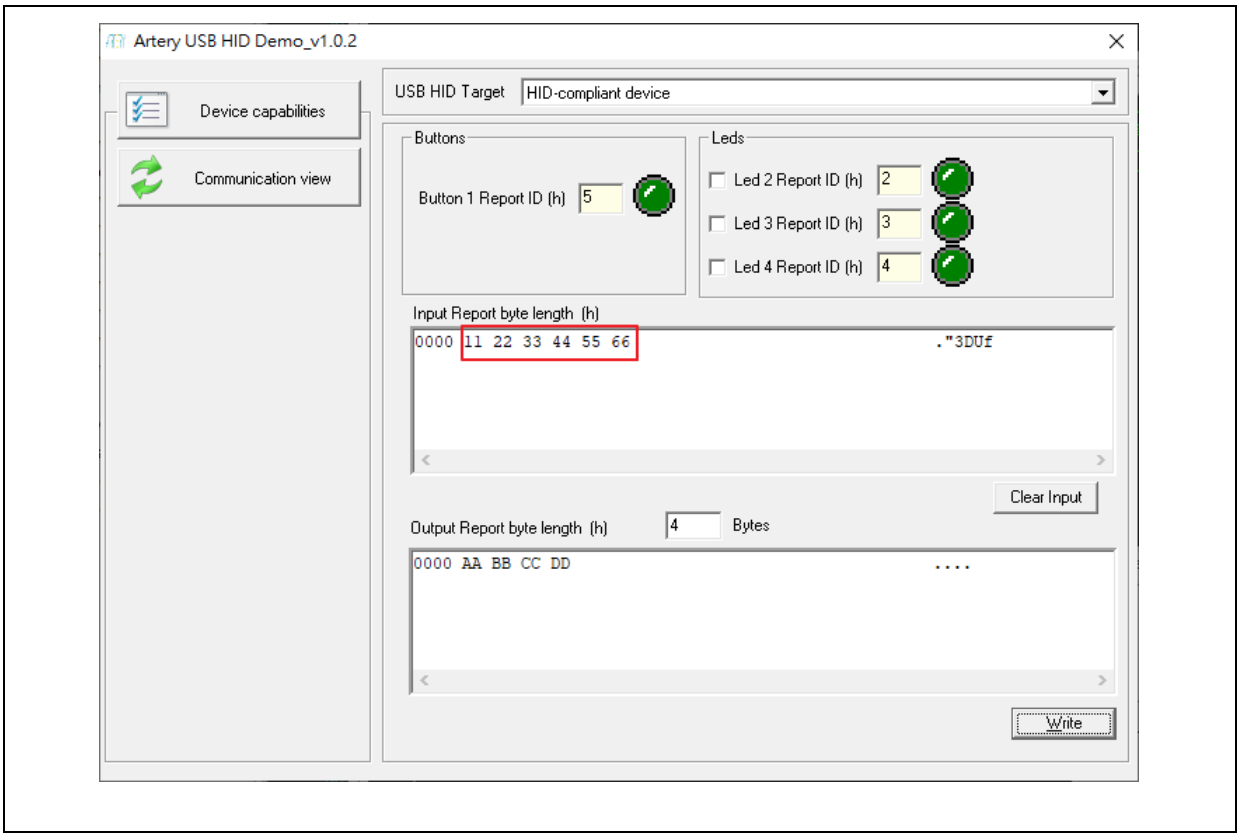


图 51. 上位机的 Input Report 印出接收到的数据内容



5 版本历史

表 5. 文档版本历史

日期	版本	变更
2021.12.30	2.0.0	最初版本
2022.04.18	2.0.1	1. 使用BSP内的函式调用LED ON或LED OFF1 2. 修改BLE Get IO状态时回传的数值，由ASCII的H及L修改为1和0
2022.04.25	2.0.2	1. 更新WB415开发板照片 2. 添加MCU端的代码下载说明
2022.06.15	2.0.3	增加透传模式案例
2022.07.08	2.0.4	更新ICP tool和XCOM界面截图
2022.11.16	2.0.5	新增USB透传模式
2023.08.21	2.0.6	1. 修改custom service为led switch service 2. 同一自定义服务与特征为128-bit的UUID 3. user_config.h可以keil configuration wizard配置

重要通知 – 请仔细阅读

买方自行负责对本文所述雅特力产品和服务的选择和使用，雅特力概不承担与选择或使用本文所述雅特力产品和服务相关的任何责任。

无论之前是否有过任何形式的表示，本文档不以任何方式对任何知识产权进行任何明示或默示的授权或许可。如果本文档任何部分涉及任何第三方产品或服务，不应被视为雅特力授权使用此类第三方产品或服务，或许可其中的任何知识产权，或者被视为涉及以任何方式使用任何此类第三方产品或服务或其中任何知识产权的保证。

除非在雅特力的销售条款中另有说明，否则，雅特力对雅特力产品的使用和/或销售不做任何明示或默示的保证，包括但不限于有关适销性、适合特定用途（及其依据任何司法管辖区的法律的对应情况），或侵犯任何专利、版权或其他知识产权的默示保证。

雅特力产品并非设计或专门用于下列用途的产品：（A）对安全性有特别要求的应用，例如：生命支持、主动植入设备或对产品功能安全有要求的系统；（B）航空应用；（C）航天应用或航天环境；（D）武器，且/或（E）其他可能导致人身伤害、死亡及财产损害的应用。如果采购商擅自将其用于前述应用，即使采购商向雅特力发出了书面通知，风险及法律责任仍将由采购商单独承担，且采购商应独力负责在前述应用中满足所有法律和法规要求。

经销的雅特力产品如有不同于本文档中提出的声明和/或技术特点的规定，将立即导致雅特力针对本文所述雅特力产品或服务授予的任何保证失效，并且不应以任何形式造成或扩大雅特力的任何责任。