微信蓝牙外设协议 dome

基于 Nordic nRF51822 平台

V0.2

Tencent Confidential

版本	变更	变更人	日期
0.1	初稿	Anqiren、ryanlin	2014/12/15
0.2	提升 pb 包版本、增加软加密等功能	Anqiren	2015/03/03

开发此测试 DEMO 需要使用的资源

Mpbledem2 需要使用的资源				
资源类型	名称	型号	版本	备注
硬件资源	nRF51822 开发板	不限	不限	下面介绍使用的硬件资源
	Jlink 或者 STlink 调试器	不限	不限	使用 SW 调试接口
	串口工具	根据开发板选择型号与类型	\	板上有 TTL~232 电平转换电路可直接选择 232~USB 串口工具或者直接使用串口线（如果直接使用串口线请注意线的类型是直通还是交叉），否则使用 TTL~USB 串口调试工具
软件环境	SoftDevice（ble stack）	\	s110_nrf51822_5.2.1	
	nRF51 SDK	\	version 4.4.0.	
	Keil for ARM	\	V4.0 及以上	
	PC 端日志工具	不限	不限	
	nrfgostudio	\	1.15.1	
资料	微信蓝牙外设协议		1.0.4 正式版	下载地址
	微信蓝牙嵌入式 pb1.0.4		1.0.4	下载地址
辅助工具	AirSyncDebugger			下载地址

*了解更多内容请进入微信硬件平台官方网站

*技术问题可访问微信硬件平台技术论坛

开发板硬件需求说明		
资源类型	数量	说明
GPIO	4	两个按键（复位键 1 个、设备按键 1 个），控制两个 LED 灯（广播 1 个、设备 1 个）。
UART	1	用于打印 log
LED	2	广播和设备信号
32.768KHz 晶振	1	RTC

目录

1. 开发环境搭建.....6
1.1. 安装 keil.....6
1.2. 安装 nRF51 SDK.....6
1.3. 安装 nrfgostudio.....6
2. Mpbledemo2 开发要求.....6
2.1. 需求说明.....6
2.1.1. 该 demo 实现一个蓝牙硬件公众号，详细功能如下：.....6
2.1.2. 该公众号具有简单的两个功能：.....6
2.2. 实现流程图.....7
2.3. 实现协议.....8
2.3.1. 数据包.....8
2.3.1.1. 包头：.....9
2.3.1.2. 包体根据不同的命令，有不同的值。.....9
2.3.2. 命令.....9
2.3.3. Seq.....10
2.4. Demo 实现说明.....10
2.4.1. 点灯/灭灯：.....10
2.4.2. 发送文本数据：.....10
3. 程序规划.....10
3.1. 参考例程.....11
3.2. 需求分析.....11
3.3. 数据流以及模块关系.....12
4. 实现代码.....12
4.1. 代码结构.....12
4.2. Comsource.c & main.c.....13
4.2.1. 程序主函数.....13
4.2.2. int fputc(int ch, FILE *F).....13
4.2.3. 串口初始化与串口事件处理函数.....14
4.2.4. 获取芯片的 MAC 地址.....14
4.2.5. 错误处理与维护.....14
4.2.6. 注册所要使用的设备.....14
4.2.7. 设备接口初始化.....14
4.2.8. 定时器初始化.....14
4.2.9. 按键初始化和按键事件处理.....15
4.2.10. 广播与广播初始化.....15
4.2.11. Ble 事件处理函数.....16
4.2.12. 绑定关系管理器初始化.....17
4.2.13. Blestack 事件调度函数与 blestack 初始化函数.....18
4.2.14. 连接参数初始化函数.....18
4.2.15. Gap 初始化.....19

4.2.16. 安全参数初始化.....19
4.2.17. wechat 服务初始化.....19
4.2.18. 等待事件.....20
4.2.19. 资源初始化函数.....20
4.3. ble_wechat_service.c.....20
4.3.1. 微信服务数据结构体.....20
4.3.2. 微信服务 uuid.....21
4.3.3. 微信服务结构体.....21
4.3.4. 微信服务数据发送函数.....21
4.3.5. 错误处理函数.....22
4.3.6. 微信服务数据接收函数.....22
4.3.7. 微信服务事件处理函数.....23
4.3.8. 添加微信服务.....23
4.3.9. 添加微信服务特征字.....24
4.4. mpbledemo2.c.....26
4.4.1. 相关数据结构体.....26
4.4.2. 计算 MD5 摘要.....28
4.4.3. 初始化函数.....28
4.4.4. 控制函数.....28
4.4.5. 复位函数.....29
4.4.6. 按键事件处理函数.....29
4.4.7. AUTH 函数.....30
4.4.8. INIT 函数.....31
4.4.9. 设备处理 write 事件函数.....31
4.4.10. 设备处理 disconnect 事件函数.....31
4.4.11. 设备处理 ble_evt 事件函数.....32
4.4.12. 设备主流程函数.....32
4.4.13. 设备释放内存函数.....33
4.4.14. 设备生产数据函数.....33
4.4.15. 设备处理数据函数.....37
4.4.16. 设备注册结构定义.....42
4.5. ble_wechat_util.....42
4.5.1. 设备存储链表节点.....42
4.5.2. 注册设备宏.....42
4.5.3. 参数设置宏.....43
4.5.4. 固定头.....43
4.5.5. 通过类型查找设备.....43
4.5.6. 转序函数.....43
4.6. epb.c 与 epb_MmBp.c.....45
4.7. Md5.c.....45
4.8. Aes.c.....45
4.9. Crc32.c.....45
5. 添加设备与设备数据流向.....45
5.1. 添加设备.....45

5.1.1. 定义设备.....45
5.1.2. 把设备添加到服务与系统中.....45
5.1.3. 把设备添加到服务中.....46
5.2. 设备数据流向.....46
5.2.1. 数据接收.....46
5.2.2. 发送数据.....46

1. 开发环境搭建

1.1. 安装 keil

安装步骤略

nRF51822 主控制器使用的是 Cortex-M0 内核，属于 ARM 构架处理器，因此要使用 keil for ARM。

1.2. 安装 nRF51 SDK

打开 nrf51_sdk_v4_4_0_30888.msi 安装文件按照默认步骤安装，SDK 会自动安装到 Keil 安装目录下。\Keil\ARM\Device\Nordic。

在\Keil\ARM\Device\Nordic 目录下有部分 ble 例程可供参考

1.3. 安装 nrfgostudio

打开 nrfgostudio_win-32_1.15.1_installer.msi 或者 nrfgostudio_win-64_1.15.1_installer.msi 安装文件按照默认步骤安装。

在这里 nRFgostudio 主要用来下载 blestack。

2. Mpbledemo2 开发要求

2.1. 需求说明

2.1.1. 该 demo 实现一个蓝牙硬件公众号，详细功能如下：

- 绑定设备，用户使用微信，扫描设备二维码，点击绑定设备。
- 连接设备，用户进入公众号，微信自动连接设备。

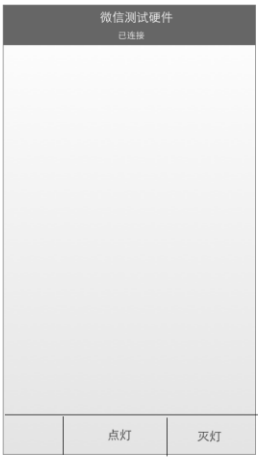
2.1.2. 该公众号具有简单的两个功能：

- 点灯，灭灯：

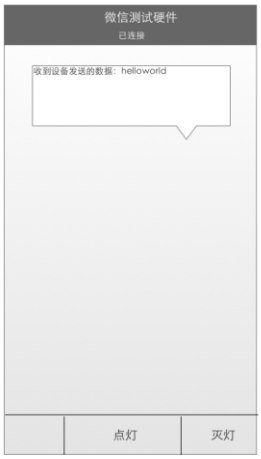
该公众号只有两个菜单：点灯，灭灯。当用户点击“点灯”菜单时，设备上的灯变亮。当用户点击“灭灯”菜单是，设备上的灯灭掉。

- 接收设备的数据：

当设备按下按钮时，向微信发送一个文本消息。服务器收到消息后，下发一条消息到公众号里，内容为：收到设备发送的消息：xxxxx（xxxx 为设备发过来的消息）。

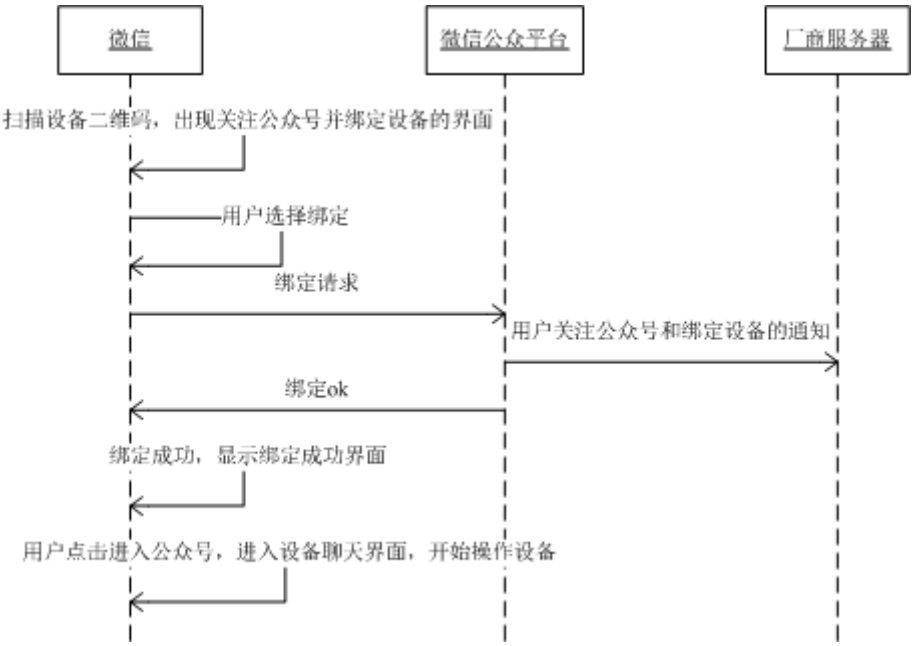


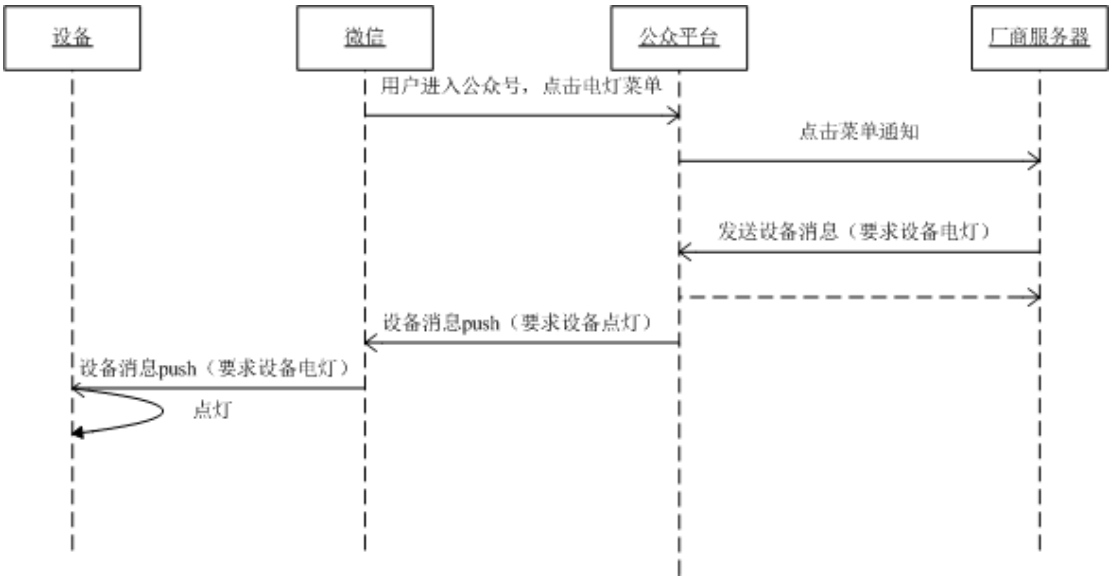
场景1:
用户点击"点灯"菜单后，设备上的灯变亮

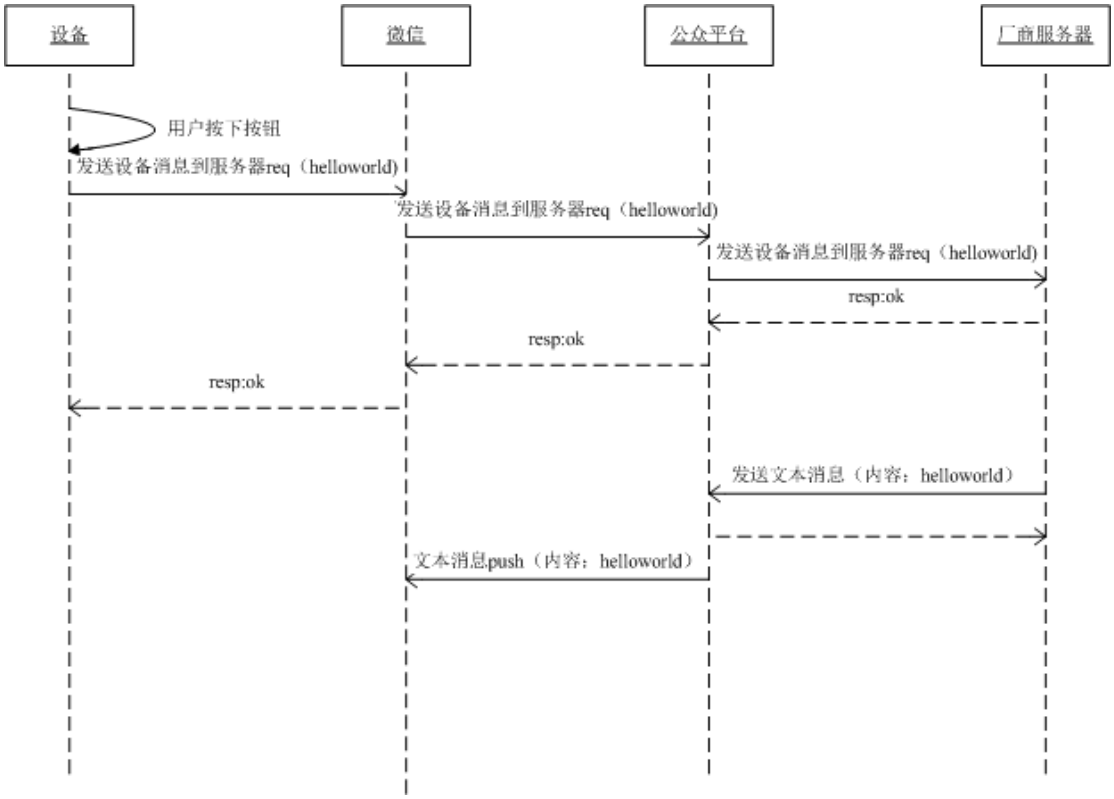


场景2:
服务器收到设备发送的数据后，发送文本消息显示在界面上

2.2. 实现流程图







2.3. 实现协议

2.3.1. 数据包

包由包头，包体组成。

2.3.1.1. 包头：

字段	类型	说明
magicCode	unsigned char[2]	两字节，固定为 fecf
version	unsigned short	两字节，目前填 1
totalLength	unsigned short	两字节，包头加包体长度
cmdId	unsigned short	两字节，命令号
seq	unsigned short	两字节，序列号。
errorCode	unsigned short	两字节，错误码。当命令为 resp 的时候有意义，其他情况下赋值为 0。

2.3.1.2. 包体根据不同的命令，有不同的值。

2.3.2. 命令

命令有三种包：

- Req : req 表示设备向服务器的一个请求，服务器必须回复一个 resp。
- Resp：服务器对设备的请求的回复。
- Push：服务器通知设备。设备不需要回复服务器。

Req 的命令号加上 0x1000 就是对应 resp 的命令号。

命令	sendTextReq（0x01）
含义	发送数据到服务器。
包体	可见字符，utf-8 编码。

命令	sendTextResp（0x1001）
含义	发送数据到服务器的回包。
包体	可见字符，utf-8 编码。可为空。

命令	openLightPush（0x2001）
含义	服务器通知设备点灯。
包体	空。

命令	closeLightPush（0x2002）
含义	服务器通知设备灭灯。
包体	空。

2.3.3. Seq

包头里的 Seq 表示一个命令序号。

- Req 命令：seq 不能为零，设备每发一个 req 请求，必须 seq+1。每个请求里的序号和对应的回包里的序号相同。
- Resp 命令：回包的序号和请求里的序号相同。
- push：序号为零。

2.4. Demo 实现说明

2.4.1. 点灯/灭灯：

- 当设备收到 openLightPush 的时候，设备点灯。
- 当设备收到 closeLightPush 的时候，设备灭灯。

2.4.2. 发送文本数据：

- 当设备上按下按钮的时候，设备发送 sendTextReq 到服务器上，内容为小于 40 字的文本（例如 helloworld）。
- 服务器收到该文本，会回复一个 sendTextResp 给设备，内容为收到的文本（例如 helloworld），并且，下发一条消息给到微信客户端，内容为：收到设备发送的数据：xxxxxx。

3. 程序规划

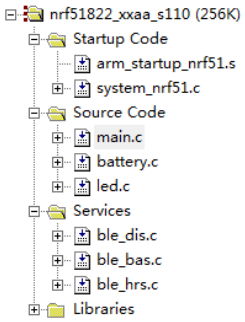
3.1. 参考例程

使用 nordic 低功耗蓝牙开发一般会参考其官方的心率例程，这个例程使用的 API 比较全面，因此这里我们也会参考这个例程。

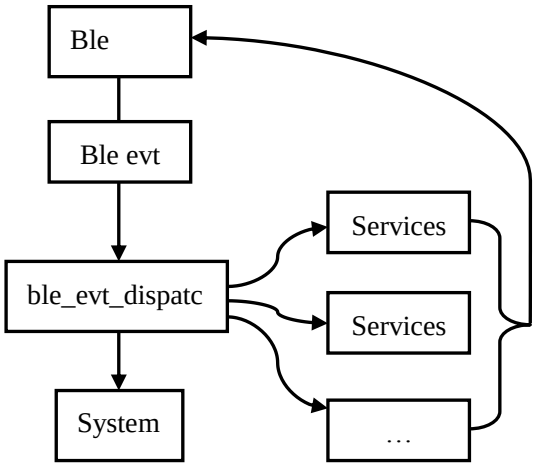
这个例程中主要使用了三个服务：

- Device Information Service，ble_sdk_srv_dis。
- Battery Service，ble_sdk_srv_bas。
- Heart Rate Service，ble_sdk_srv_hrs。

代码结构如下：



简单分析一下其代码，可以得出蓝牙的事件处理流程：

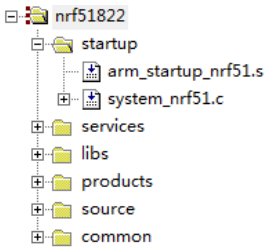


3.2. 需求分析

开发此 demo 需要实现以下内容：

- 外设与微信通信协议（AirSYNC）
- 外设与后台服务器通信协议（Mpbledemo2 开发要求中规定的协议）
- Protobuf 封包、解包以，MD5 摘要计算等

因此做如下代码结构设计：



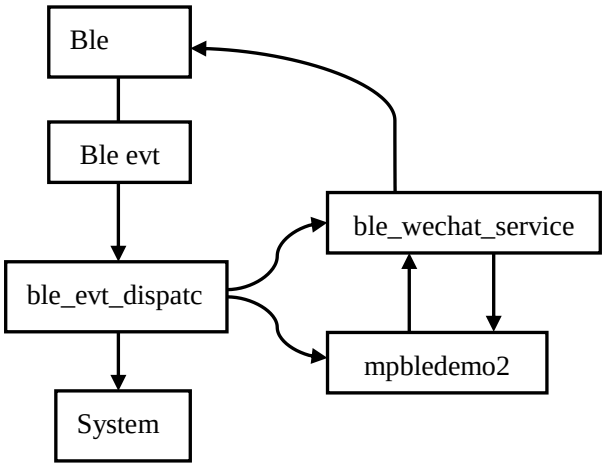
Service 文件夹下放 AirSYNC 通信服务，命名为 ble_wechat_service。

Products 文件夹下放 mpbledemo2 相关内容，命名为 mpbledemo2。
Common 文件夹下放通用内容，如 protobuf、md5 等

3.3. 数据流以及模块关系

AirSYNC 通信（ble_wechat_service）是直接与 blestack 发生数据交流的，因此要把这一部分按普通服务来实现。

Mpbledemo2 的收发数据是通过 ble_wechat_service 来收发数据，但又会去处理一些 ble_evt，因此也可以当作一个服务来对待。但是 mpbledemo2 要注册在 ble_wechat_service 上而不是像普通的服务那样注册在 blestack 上。

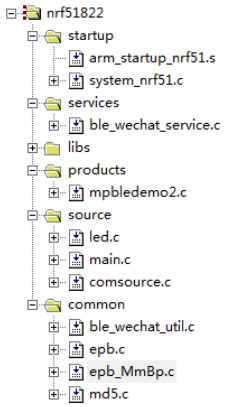


4. 实现代码

注意*代码以官网上参考代码为准，这里的代码仅用来讲解

4.1. 代码结构

整个程序最终代码结构如下：



代码包含 ble_wechat_service.c, mpbledemo2.c, led.c(非必须)， main.c, comsource.c, ble_wechat_util.c, epb.c, epb_MmBp.c, md5.c

在 mpbledemo2.h 中定义了与设备相关的一些信息，如设备类型（DEVICE_TYPE）、

设备名称（DEVICE_ID）、设备密钥等与设备相关的信息。请注意根据自己需要重新定义。

4.2. Comsource.c & main.c

Nordic 官方的心率程序 main.c 非常庞大，在这里我们做了一下拆分与整合，但是其功能是一样的。
这里仅对不同的地方做一下功能说明。

4.2.1. 程序主函数

这是一个简单的一个主函数，初始化所需要的资源，开始广播，然后进入一个主循环。在主循环里等待事件和运行添加设备的主流程。

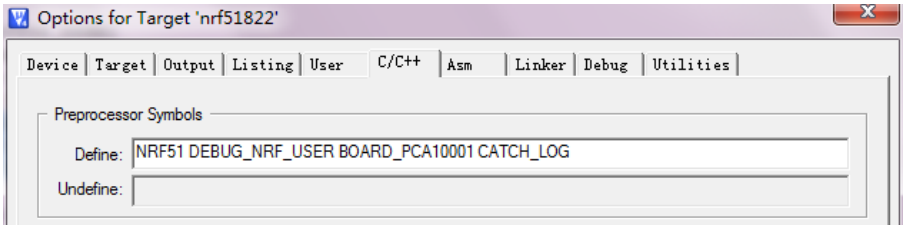
```
int main(void)
{
        resource_init();
        advertising_start();
  for(;;)
  {
        wait_app_event();
        m_mpbledemo2_handler->m_data_main_process_func(&m_ble_wechat);//通过接口调用设备主流程函数，并选择
//设备所使用的服务，这里选择 wechat 服务。
  }
}
```

4.2.2. int fputc(int ch, FILE *F)

为了方便调试，我们把 uart 串口绑定到了 STDIO 上面，这样就方便打印与查看日志。要做的是#include <stdio>，然后在程序里面加上如下代码就可以了。在这段代码里我们没有等待或者查询内容是否发送完毕，因为这是一个不重要的功能。

```
int fputc(int ch, FILE *F)
{
  app_uart_put(ch);
  return ch;
}
```

其中程序中的 log 是用如下方法打印的，如果需要打印 log 需要在程序中或者编译器中 define CATCH_LOG 以添加打印 log 的代码带程序中，如图所示：



程序中的 log 块：

```
#ifdef CATCH_LOG
        printf("\r\n log：log msg");
#endif
```

4.2.3. 串口初始化与串口事件处理函数

static void uart_event_handler(app_uart_evt_t *p_app_uart_event) & void uart_init(void)
这里初始化 nRF51822 的串口用啦打印日志与输入相关信息（输入信息会在之后的版本中使用）。

4.2.4. 获取芯片的 MAC 地址

void get_mac_addr(uint8_t *p_mac_addr)；该函数用来获取当前芯片的 mac 地址。
在程序中我们把获取到的 mac 地址放在全局变量
uint8_t m_addl_adv_manuf_data[BLE_GAP_ADDR_LEN];中，该变量用来初始化广播信息中的 ble_advdata_manuf_data（ios 设备需要）。

4.2.5. 错误处理与维护

错误处理函数与维护函数要根据实际产品自己来定义添加或者移除。

4.2.6. 注册所要使用的设备

```
void register_all_products(void) {
        REGISTER(mpbledemo2);
}
```

4.2.7. 设备接口初始化

在添加设备时会定义一个指向该设备接口（data_handler）的指针：
data_handler *m_mpbledemo2_handler = NULL;
之后所有对设备的操作均通过此接口，但最初这个指针是空的，我们需要在注册列表里（在 resource_init(); 中，需要注册所添加的设备，也就是把设备的接口存储到一个链表里）找到该接口并把指针指向该接口。

```
void    data_handler_init(data_handler** p_data_handler, uint8_t product_type)
{
        if (*p_data_handler == NULL)
        {
                *p_data_handler = get_handler_by_type(product_type);
        }
}
```

4.2.8. 定时器初始化

初始化定时器模块，用开产生并启动应用程序计时器。

```
void timers_init(void)
{
    APP_TIMER_INIT(APP_TIMER_PRESCALER, APP_TIMER_MAX_TIMERS, APP_TIMER_OP_QUEUE_SIZE, false);
}
```

4.2.9. 按键初始化和按键事件处理

按键事件处理函数，当有按键按下时调用该服务函数，该服务函数把执行相应处理后把键值传递给设备按键处理函数。这个函数是在 buttons_init(void)函数中注册的。

```
static void button_pullup_handler(uint8_t pin_no)
{
    NVIC_DisableIRQ(GPIOTE_IRQn);
    switch (pin_no)
    {
        case SYS_RESET:
        {
        #ifdef CATCH_LOG
                printf("\r\n button 0");
        #endif
                NVIC_SystemReset ();
        }
     break;
    }
    uint32_t err_code;
    err_code = m_mpbledemo2_handler->m_data_button_handler_func(&m_ble_wechat, pin_no);
    APP_ERROR_CHECK(err_code);
    NVIC_EnableIRQ(GPIOTE_IRQn);
    return;
}
```

按键初始化函数，该函数初始化 SYS_RESET 和 EVAL_BOARD_BUTTON_1 两个按键作为唤醒按键，并设置为上拉模式（没有外接上拉电阻）。

```
static void buttons_init(void)
{
    static app_button_cfg_t buttons[] =
    {
        {SYS_RESET, false, NRF_GPIO_PIN_PULLUP, button_pullup_handler},
        {MPBLEDEMO2_BUTTON_1, false, NRF_GPIO_PIN_PULLUP, button_pullup_handler}
    };
    APP_BUTTON_INIT(buttons, sizeof(buttons) / sizeof(buttons[0]), BUTTON_DETECTION_DELAY, false);
        uint32_t err_code;
        err_code = app_button_enable();
        APP_ERROR_CHECK(err_code);
}
```

4.2.10. 广播与广播初始化

广播初始化函数用来初始化广播参数以及广播内容等。

```
void advertising_init(void)
{
    uint32_t        err_code;
    ble_advdata_t advdata;
```

```
    uint8_t           flags = BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE;
    // 定义 UUID 数据类型并初始化，UUID 初始化为 BLE_UUID_WECHAT_SERVICE，类型初始化为
BLE_UUID_TYPE_BLE（因为该 UUID 已经向 Bluetooth SIG 申请）。
    ble_uuid_t adv_uuids[] =
    {
        {BLE_UUID_WECHAT_SERVICE,              BLE_UUID_TYPE_BLE}
    };
    // 建立并设置广播初始化数据体。
    memset(&advdata, 0, sizeof(advdata));
    advdata.name_type                = BLE_ADVDATA_FULL_NAME;
    advdata.include_appearance       = false;
    advdata.flags.size               = sizeof(flags);
    advdata.flags.p_data             = &flags;
    advdata.uuids_complete.uuid_cnt = sizeof(adv_uuids) / sizeof(adv_uuids[0]);
    advdata.uuids_complete.p_uuids  = adv_uuids;
//在 advdata 的 p_manuf_specific_data 数据段里加入本地 MAC 地址，并广播（IOS 设备需要）。
    ble_advdata_manuf_data_t          manuf_data;
    manuf_data.company_identifier = COMPANY_IDENTIFIER;
    manuf_data.data.size              = sizeof(m_addl_adv_manuf_data);
    manuf_data.data.p_data            = m_addl_adv_manuf_data;
    advdata.p_manuf_specific_data = &manuf_data;
    //设置广播参数和内容
    err_code = ble_advdata_set(&advdata, NULL);
    APP_ERROR_CHECK(err_code);
    // 设置广播参数（开始广播时需要的内容）。
    memset(&m_adv_params, 0, sizeof(m_adv_params));
    m_adv_params.type            = BLE_GAP_ADV_TYPE_ADV_IND;
    m_adv_params.p_peer_addr = NULL;                                 // Undirected advertisement
    m_adv_params.fp              = BLE_GAP_ADV_FP_ANY;
    m_adv_params.interval      = APP_ADV_INTERVAL;
    m_adv_params.timeout       = APP_ADV_TIMEOUT_IN_SECONDS;
}
```

开始广播，调用该函数，设备开始广播。

```
void advertising_start(void)
{
    uint32_t err_code;
    err_code = sd_ble_gap_adv_start(&m_adv_params);
    APP_ERROR_CHECK(err_code);
    //广播开始 LED0 闪烁
    ed_start();
}
```

4.2.11. Ble 事件处理函数

系统处理 blestack 抛上来的事件。

```
void on_ble_evt(ble_evt_t * p_ble_evt)
{
        uint32_t        err_code;
    switch (p_ble_evt->header.evt_id)
    {
        case BLE_GAP_EVT_CONNECTED:
//连接后保存当前连接号，LED0 停止闪烁
                m_conn_handle = p_ble_evt->evt.gap_evt.conn_handle;
                led_stop();
          break;
        case BLE_GAP_EVT_DISCONNECTED:
//断开连接后存储绑定关系，无效当前连接号，并开始广播。
            err_code = ble_bondmngr_bonded_masters_store();
            APP_ERROR_CHECK(err_code);
            m_conn_handle = BLE_CONN_HANDLE_INVALID;
            NVIC_SystemReset();
            break;
        case BLE_GAP_EVT_SEC_PARAMS_REQUEST:
            err_code = sd_ble_gap_sec_params_reply(m_conn_handle,
                                                   BLE_GAP_SEC_STATUS_SUCCESS,
                                                   &m_sec_params);
            break;
        case BLE_GAP_EVT_TIMEOUT:
//超时后系统关闭
            if (p_ble_evt->evt.gap_evt.params.timeout.src == BLE_GAP_TIMEOUT_SRC_ADVERTISEMENT)
            {
                err_code = sd_power_system_off();
                APP_ERROR_CHECK(err_code);
            }
            break;
        default:
                break;
    }
}
```

4.2.12. 绑定关系管理器初始化

```
static void bond_manager_init(void)
{
    uint32_t                    err_code;
    ble_bondmngr_init_t bond_init_data;
    bond_init_data.flash_page_num_bond     = FLASH_PAGE_BOND;
    bond_init_data.flash_page_num_sys_attr = FLASH_PAGE_SYS_ATTR;
    bond_init_data.evt_handler             = NULL;
```

```
    bond_init_data.error_handler                = bond_manager_error_handler;
    bond_init_data.bonds_delete                 = BOND_DELETE;
    err_code = ble_bondmngr_init(&bond_init_data);
    APP_ERROR_CHECK(err_code);
}
```

4.2.13. Blestack 事件调度函数与 blestack 初始化函数

把所有需要 blestack 事件的函数放到调度函数里，该函数在下面的 blestack 初始化函数里注册。

```
static void ble_evt_dispatch(ble_evt_t * p_ble_evt)
{
    ble_bondmngr_on_ble_evt(p_ble_evt);
    ble_wechat_on_ble_evt(&m_ble_wechat, p_ble_evt, m_mpbledemo2_handler);
    ble_conn_params_on_ble_evt(p_ble_evt);
    m_mpbledemo2_handler->m_data_on_ble_evt_func(&m_ble_wechat,p_ble_evt);
    on_ble_evt(p_ble_evt);
}
```

Blestack 初始化函数

```
static void ble_stack_init(void)
{
    BLE_STACK_HANDLER_INIT(NRF_CLOCK_LFCLKSRC_XTAL_20_PPM,
                           BLE_L2CAP_MTU_DEF,
                           ble_evt_dispatch,
                           false);
}
```

4.2.14. 连接参数初始化函数

根据设备连接需要与策略来设置

```
static void conn_params_init(void)
{
    uint32_t               err_code;
    ble_conn_params_init_t cp_init;
    memset(&cp_init, 0, sizeof(cp_init));
    cp_init.p_conn_params                  = NULL;
    cp_init.first_conn_params_update_delay = FIRST_CONN_PARAMS_UPDATE_DELAY;
    cp_init.next_conn_params_update_delay  = NEXT_CONN_PARAMS_UPDATE_DELAY;
    cp_init.max_conn_params_update_count   = MAX_CONN_PARAMS_UPDATE_COUNT;
    cp_init.start_on_notify_cccd_handle    = BLE_GATT_HANDLE_INVALID;
    cp_init.disconnect_on_fail             = true;
    cp_init.evt_handler                    = NULL;
    cp_init.error_handler                  = conn_params_error_handler;
    err_code = ble_conn_params_init(&cp_init);
    APP_ERROR_CHECK(err_code);
```

```
}
```

4.2.15. Gap 初始化

```
static void gap_params_init(void)
{
    uint32_t                err_code;
    ble_gap_conn_params_t   gap_conn_params;
    ble_gap_conn_sec_mode_t sec_mode;
    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&sec_mode);
    err_code = sd_ble_gap_device_name_set(&sec_mode, (const uint8_t *)DEVICE_NAME, strlen(DEVICE_NAME));
    APP_ERROR_CHECK(err_code);
    memset(&gap_conn_params, 0, sizeof(gap_conn_params));
    gap_conn_params.min_conn_interval = MIN_CONN_INTERVAL;
    gap_conn_params.max_conn_interval = MAX_CONN_INTERVAL;
    gap_conn_params.slave_latency     = SLAVE_LATENCY;
    gap_conn_params.conn_sup_timeout  = CONN_SUP_TIMEOUT;

    err_code = sd_ble_gap_ppcp_set(&gap_conn_params);
    APP_ERROR_CHECK(err_code);
}
```

4.2.16. 安全参数初始化

在安全参数请求事件中调用。

```
static void sec_params_init(void)
{
    m_sec_params.timeout          = SEC_PARAM_TIMEOUT;
    m_sec_params.bond             = SEC_PARAM_BOND;
    m_sec_params.mitm             = SEC_PARAM_MITM;
    m_sec_params.io_caps          = SEC_PARAM_IO_CAPABILITIES;
    m_sec_params.oob              = SEC_PARAM_OOB;
    m_sec_params.min_key_size = SEC_PARAM_MIN_KEY_SIZE;
    m_sec_params.max_key_size = SEC_PARAM_MAX_KEY_SIZE;
}
```

4.2.17. wechat 服务初始化

服务初始化主要是添加微信服务和添加特征字。

```
static void services_init(void)
{
        uint32_t err_code;
        err_code = ble_wechat_add_service(&m_ble_wechat);
        APP_ERROR_CHECK(err_code);
        err_code = ble_wechat_add_characteristics(&m_ble_wechat);
        APP_ERROR_CHECK(err_code);
}
```

4.2.18. 等待事件

此函数会把系统设置到低功耗状态，并等待系统事件。

```
void wait_app_event(void)
{
    uint32_t err_code = sd_app_event_wait();
    APP_ERROR_CHECK(err_code);
}
```

4.2.19. 资源初始化函数

初始化要使用的所有资源，注意初始化的顺序。如先获取 mac 地址再进行广播初始化，先初始化设备接口指针再去调用设备接口函数初始化设备等。

```
void resource_init(void) {
        uart_init();//串口初始化
        timers_init();//定时器初始化
        gpiote_init();//io 初始化
        buttons_init();//按键初始化
        ble_stack_init();//blestack 初始化
        get_mac_addr(m_addl_adv_manuf_data);//获取本地 mac 用来初始化广播信息
        bond_manager_init();//绑定关系管理器初始化
        register_all_products();//注册所有设备，把定义的设备添加到设备列表里
        data_handler_init(&m_mpbledemo2_handler,  PRODUCT_TYPE_MPBLEDEMO2);//设备接口指针初始化。
        APP_ERROR_CHECK(m_mpbledemo2_handler->m_data_init_func());//所使用的设备初始化
        gap_params_init();//gap 参数初始化
        advertising_init();  //广播初始化
        services_init();//服务初始化
        conn_params_init();//连接参数初始化
        sec_params_init();//安全参数初始化
}
```

4.3. ble_wechat_service.c

这里是 wechat_service 的主程序。

4.3.1. 微信服务数据结构体

我们定义了如下的结构体用来发送与接收数据，由于一帧发送的数据量有限，这里我们规定的是#define BLE_WECHAT_MAX_DATA_LEN (GATT_MTU_SIZE_DEFAULT - 3)。因此需要有 offset 来记录当前的数据偏移。

```
typedef struct
{
        uint8_t *data;
        int len;
        int offset;
} data_info;
```

4.3.2. 微信服务 uuid

微信服务必须使用如下 UUID（已向蓝牙协会申请）

```
#define BLE_UUID_WECHAT_SERVICE                         0xFEE7
#define BLE_UUID_WECHAT_WRITE_CHARACTERISTICS           0xFEC7
#define BLE_UUID_WECHAT_INDICATE_CHARACTERISTICS    0xFEC8
#define BLE_UUID_WECHAT_READ_CHARACTERISTICS            0xFEC9
```

4.3.3. 微信服务结构体

```
typedef struct
{
    uint16_t                service_handle;
    ble_gatts_char_handles_t    indicate_handles;
    ble_gatts_char_handles_t    write_handles;
    ble_gatts_char_handles_t    read_handles;
    uint16_t                conn_handle;
} ble_wechat_t;
```

4.3.4. 微信服务数据发送函数

```
int ble_wechat_indicate_data(ble_wechat_t *p_wcs,data_handler *p_data_handler, uint8_t *data, int len)
{
    if (data == NULL || len == 0) {
        return 0;
    }
    if (g_send_data.len != 0 && g_send_data.offset != g_send_data.len) {
        return 0;
    }
    g_send_data.data = data;
    g_send_data.len = len;
    g_send_data.offset = 0;
    return (ble_wechat_indicate_data_chunk(p_wcs,p_data_handler));
}

static int ble_wechat_indicate_data_chunk(ble_wechat_t *p_wcs, data_handler *p_data_handler)
{
    ble_gatts_hvx_params_t hvx_params;
    uint16_t chunk_len = 0;
    chunk_len = g_send_data.len-g_send_data.offset;
    chunk_len = chunk_len > BLE_WECHAT_MAX_DATA_LEN?BLE_WECHAT_MAX_DATA_LEN:chunk_len;
    if (chunk_len == 0)
{
        p_data_handler->m_data_free_func(g_send_data.data, g_send_data.len);
        g_send_data.data = NULL;
        g_send_data.len = 0;
```

```
                g_send_data.offset = 0;
                return -1;
        }
        memset(&hvx_params, 0, sizeof(hvx_params));
        hvx_params.handle = p_wcs->indicate_handles.value_handle;
        hvx_params.p_data = g_send_data.data+g_send_data.offset;
        hvx_params.p_len    = &chunk_len;
        hvx_params.type     = BLE_GATT_HVX_INDICATION;
        g_send_data.offset += chunk_len;
        return sd_ble_gatts_hvx(p_wcs->conn_handle, &hvx_params);
}
```

该函数仅负责发送一帧函数，并记录当前 offset。当成功发送一帧数据之后，ble stack 会抛上来一个确认事件， p_ble_evt->header.evt_id = BLE_GATTS_EVT_HVC。当收到这个事件以后调用下面的 on_indicate_comfirm 函数以发送剩余数据直到数据发送完毕。

```
void on_indicate_comfirm(ble_wechat_t *p_wcs, ble_evt_t * p_ble_evt, data_handler *p_data_handler)
{
        ble_wechat_indicate_data_chunk(p_wcs,p_data_handler);
}
```

4.3.5. 错误处理函数

void wechat_error_chack(ble_wechat_t *p_wcs, data_handler *p_data_handler, int error_code)；这个函数的具体内容要根据具体的产品来定义，该 demo 的做法是打印出错误信息并调用当前设备的错误处理函数 p_data_handler->m_data_error_func(error_code);该函数在设备中定义。

4.3.6. 微信服务数据接收函数

这是微信客户端发起写事件的处理函数，用来接收 push 下来的数据。当接收完所有数据之后会调用设备的 p_data_handler->m_data_consume_func(g_rcv_data.data, g_rcv_data.len)函数，该函数在设备中定义。

```
static void on_write(ble_wechat_t *p_wcs, ble_evt_t * p_ble_evt, data_handler *p_data_handler)
{
        int error_code;
        ble_gatts_evt_write_t * p_evt_write = &p_ble_evt->evt.gatts_evt.params.write;
        int chunk_size = 0;
        if (p_evt_write->handle == p_wcs->write_handles.value_handle &&
                        p_evt_write->len <= BLE_WECHAT_MAX_DATA_LEN)
        {
                if (g_rcv_data.len == 0)
                {
                        BpFixHead *fix_head = (BpFixHead *) p_evt_write->data;
                        g_rcv_data.len = ntohs(fix_head->nLength);
                        g_rcv_data.offset = 0;
```

```
                g_rcv_data.data = (uint8_t *)malloc(g_rcv_data.len);
            }
            chunk_size = g_rcv_data.len - g_rcv_data.offset;
            chunk_size = chunk_size < p_evt_write->len ? chunk_size : p_evt_write->len;
            memcpy(g_rcv_data.data+g_rcv_data.offset, p_evt_write->data, chunk_size);
            g_rcv_data.offset += chunk_size;
            if (g_rcv_data.len <= g_rcv_data.offset) //接收完所有帧
            {
                error_code = p_data_handler->m_data_consume_func(g_rcv_data.data, g_rcv_data.len);
                free(g_rcv_data.data);
                wechat_error_chack(p_wcs, p_data_handler, error_code);
                g_rcv_data.len = 0;
                g_rcv_data.offset = 0;
            }
        }
    }
```

4.3.7. 微信服务事件处理函数

主要处理由 ble stack 抛上来的事件。

```
void ble_wechat_on_ble_evt(ble_wechat_t *p_wcs,ble_evt_t * p_ble_evt, data_handler *p_data_handler) {
        switch (p_ble_evt->header.evt_id)
    {
            case BLE_GAP_EVT_CONNECTED:
//获取当前连接
                p_wcs->conn_handle = p_ble_evt->evt.gap_evt.conn_handle;
                    break;
        case BLE_GATTS_EVT_WRITE://接收数据
            on_write(p_wcs,p_ble_evt,p_data_handler);
            break;
        case BLE_GATTS_EVT_HVC://数据确认
                    on_indicate_comfirm(p_wcs,p_ble_evt,p_data_handler);
                    break;
            case BLE_GAP_EVT_DISCONNECTED://断开连接
                    p_wcs->conn_handle = BLE_CONN_HANDLE_INVALID;
                    break;
        default:
            break;
    }
}
```

4.3.8. 添加微信服务

在 GATTS sercvice 中添加微信服务。

```
uint32_t ble_wechat_add_service(ble_wechat_t *p_wechat)
{
        uint32_t err_code;
        ble_uuid_t ble_wechat_uuid;
        BLE_UUID_BLE_ASSIGN(ble_wechat_uuid, BLE_UUID_WECHAT_SERVICE);
        err_code = sd_ble_gatts_service_add(BLE_GATTS_SRVC_TYPE_PRIMARY, &ble_wechat_uuid,
&p_wechat->service_handle);
        return err_code;
}
```

4.3.9. 添加微信服务特征字

微信服务特征字主要有三个，indicate_char、write_char、read_char，为了支持同一手机上一个app连接了设备后，微信还能搜素并连接到设备的情况，设备需要在微信的service下面，暴露一个read character，内容为6字节的mac地址。当ios上的其他app连接上设备时，设备不会再广播，微信会读取该特征值，以确定是否要连接该设备。

- 添加 indicate char。

```
uint32_t ble_wechat_add_indicate_char(ble_wechat_t *p_wechat)
{
        ble_gatts_char_md_t char_md;
    ble_gatts_attr_md_t cccd_md;
    ble_gatts_attr_t    attr_char_value;
    ble_uuid_t          char_uuid;
    ble_gatts_attr_md_t attr_md;
        char *data = "indicate char";
        memset(&cccd_md, 0, sizeof(cccd_md));
    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&cccd_md.read_perm);
    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&cccd_md.write_perm);
        cccd_md.vloc = BLE_GATTS_VLOC_STACK;
        memset(&char_md, 0, sizeof(char_md));
    char_md.char_props.indicate = 1;
    char_md.p_char_user_desc    = NULL;
    char_md.p_char_pf           = NULL;
    char_md.p_user_desc_md      = NULL;
    char_md.p_cccd_md           = NULL;//&cccd_md;
    char_md.p_sccd_md           = NULL;
BLE_UUID_BLE_ASSIGN(char_uuid, BLE_UUID_WECHAT_INDICATE_CHARACTERISTICS);
        memset(&attr_md, 0, sizeof(attr_md));
    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&attr_md.read_perm);
```

```
    BLE_GAP_CONN_SEC_MODE_SET_NO_ACCESS(&attr_md.write_perm);
    attr_md.vloc            = BLE_GATTS_VLOC_STACK;
    attr_md.rd_auth         = 0;
    attr_md.wr_auth         = 0;
    attr_md.vlen            = 1;
    memset(&attr_char_value, 0, sizeof(attr_char_value));
    attr_char_value.p_uuid              = &char_uuid;
    attr_char_value.p_attr_md           = &attr_md;
    attr_char_value.init_len            = strlen(data);
    attr_char_value.init_offs           = 0;
    attr_char_value.max_len             = BLE_WECHAT_MAX_DATA_LEN;
    attr_char_value.p_value             = (uint8_t *)data;
    return sd_ble_gatts_characteristic_add(p_wechat->service_handle,
                                           &char_md,
                                           &attr_char_value,
                                           &p_wechat->indicate_handles);
}
```

- 添加 write_char

```
static uint32_t ble_wechat_add_write_char(ble_wechat_t *p_wechat)
 {
        //内容见源程序。
}
```

- 添加 read_char

  添加 read_char 时需要获取当前设备的 mac 地址，用来初始化 attr_char_value.p_value。

```
static uint32_t ble_wechat_add_read_char(ble_wechat_t *p_wechat)
{
  //m_addl_adv_manuf_data.
        get_mac_addr(m_addl_adv_manuf_data);
        ble_gatts_char_md_t char_md;
    ble_gatts_attr_t    attr_char_value;
    ble_uuid_t              char_uuid;
    ble_gatts_attr_md_t attr_md;

        memset(&char_md, 0, sizeof(char_md));
    char_md.char_props.read = 1;
    char_md.p_char_user_desc    = NULL;
    char_md.p_char_pf           = NULL;
    char_md.p_user_desc_md      = NULL;
    char_md.p_cccd_md           = NULL;
    char_md.p_sccd_md           = NULL;

        BLE_UUID_BLE_ASSIGN(char_uuid, BLE_UUID_WECHAT_READ_CHARACTERISTICS);
```

```
    memset(&attr_md, 0, sizeof(attr_md));
    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&attr_md.read_perm);
    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&attr_md.write_perm);
    attr_md.vloc        = BLE_GATTS_VLOC_STACK;
    attr_md.rd_auth     = 0;
    attr_md.wr_auth     = 0;
    attr_md.vlen        = 1;

    memset(&attr_char_value, 0, sizeof(attr_char_value));

    attr_char_value.p_uuid      = &char_uuid;
    attr_char_value.p_attr_md   = &attr_md;
    attr_char_value.init_len    = sizeof(m_addl_adv_manuf_data);
    attr_char_value.init_offs   = 0;
    attr_char_value.max_len     = BLE_WECHAT_MAX_DATA_LEN;
    attr_char_value.p_value     = m_addl_adv_manuf_data;

    return sd_ble_gatts_characteristic_add(p_wechat->service_handle,
                                           &char_md,
                                           &attr_char_value,
                                           &p_wechat->read_handles);
}
```

4.4. mpbledemo2.c

mpbledemo2 是与设备相关的文件

4.4.1. 相关数据结构体

- Mpbledemo2 解包错误码，数值可以根据需求自定义。

```
typedef enum
{
  errorCodeUnpackAuthResp = 0x9990,
  errorCodeUnpackInitResp = 0x9991,
  errorCodeUnpackSendDataResp = 0x9992,
  errorCodeUnpackCtlCmdResp = 0x9993,
  errorCodeUnpackRecvDataPush = 0x9994,
  errorCodeUnpackSwitchViewPush = 0x9995,
  errorCodeUnpackSwitchBackgroundPush = 0x9996,
  errorCodeUnpackErrorDecode = 0x9997,
}mpbledemo2UnpackErrorCode;
```

- Mpbledemo2 打包错误码，数值可以根据需求自定义。

```
typedef enum
```

```
{
  errorCodeProduce = 0x9980,
}mpbledemo2PackErrorCode;
```

- 对 Mpbledemo2 的相关命令定义

```
typedef enum
{
  sendTextReq = 0x01,//发送字符
  sendTextResp = 0x1001,//收到字符
  openLightPush = 0x2001,//开灯命令
  closeLightPush = 0x2002,//关灯命令
}BleDemo2CmdID;
```

- Mpbledemo2 数据头格式，具体内容见第二章 Mpbledemo2 开发要求中的实现协议。

```
typedef struct
{
  uint8_t m_magicCode[2];//魔数
  uint16_t m_version;//版本
  uint16_t m_totalLength;//数据总长度（
  uint16_t m_cmdid;//命令号
  uint16_t m_seq;//序号
  uint16_t m_errorCode;//错误吗
}BlueDemoHead;
```

- Mpbledemo2 数据结构体

  这个结构体主要用于本地对设备的控制，要向设备传输某参数时，首先初始化此结构体，然后把这个结构体当作一个参数传递过去。在 mpbledemo2_data_produce_func 函数中使用，后面章节会具体说明。

  如，要进行 auth，init，发送数据等控制。这个可以根据设备需要来定义，如该设备需要发送字符串，就在该结构体中定义了 CString send_msg;

```
typedef struct
{
    int cmd;
    CString send_msg;
} mpbledemo2_info;
```

- Mpbledemo2 状态结构体

  这个结构体用来记录设备的当前状态，以用来判断之后的步骤如何执行。

```
typedef struct
{
    bool wechats_switch_state; //公众账号被切换到前后台的状态
    bool indication_state;//indication char 是否被订阅，true 表示被订阅，false 表示未订阅
    bool auth_state;//auth 状态，true 表示 auth 成功，false 表示为 auth 成功或者未进行 auth。
    bool init_state;// init 状态，true 表示 init 成功，false 表示为 init 成功或者未进行 init。
    bool auth_send;//auth 发送状态，true 表示已经发送 auth 包，false 表示未发送 auth 包。
```

```
    bool init_send;// init 发送状态，true 表示已经发送 init 包，false 表示未发送 init 包。
    unsigned short send_data_seq;//记录发送数据的序列号，非必需
    unsigned short push_data_seq;//记录接受到数据的序列号，非必需
    unsigned short seq;//当前序列号。
}mpbledemo2_state;
```

4.4.2. 计算 MD5 摘要

目前（截至微信 6.0 版本）外设使用的是 DEVICE_TYPE+ DEVICE_ID 字符串的 md5 摘要来进行认证的。当然此摘要可以算好再烧录或者传输到设备中来。这个 Demo 中添加了 md5 摘要算法（虽然 DEVICE_ID 是 hide code，之后 DEVICE_ID 可以通过串口写进来并记录到 flash 中，以方便生产）。

```
int32_t mpbledemo2_get_md5(void)
{
  int32_t error_code;
  char device_type[] = DEVICE_TYPE;
  char device_id[] = DEVICE_ID;
  char argv[sizeof(DEVICE_TYPE) + sizeof(DEVICE_ID) - 1];
  memcpy(argv,device_type,sizeof(DEVICE_TYPE));
/*这里用来合并 DEVICE_TYPE 和 DEVICE_ID 字符串，但是要注意去掉字符串后面的'\0',在操作内存的时候-1 覆盖即可 */
  memcpy(argv + sizeof(DEVICE_TYPE)-1,device_id,sizeof(DEVICE_ID));
  error_code = md5(argv, md5_type_and_id);//md5 摘要
  return error_code;
}
```

要注意的是 md5 须要在 auth 之前获得，因此我们把它放在初始化函数队列里面。

4.4.3. 初始化函数

初始化一个 gpio 用来控制一个 led 灯。

```
void light_init(void)
{
    nrf_gpio_cfg_output(MPBLEDEMO2_LIGHT);
}
```

Mpbledemo2 的初始化函数。

```
int32_t mpbledemo2_init(void)
{
  light_init();
  return (mpbledemo2_get_md5());
}
```

4.4.4. 控制函数

这里的外设与功能比较简单，主要是开关灯的控制。

开灯。

```
void light_on(uint8_t light)
```

```
{
  nrf_gpio_pin_set(light);
}
```

关灯。

```
void light_off(uint8_t light)
{
  nrf_gpio_pin_clear(light);
}
```

4.4.5. 复位函数

这个函数主要在断开连接时使用，用来复位设备某些状态，以便在重新连接时重新进行 auth 和 init 等相关操作。

```
void mpbledemo2_reset()
{
  mpbledemo2Sta.auth_send = false;
  mpbledemo2Sta.auth_state = false;
  mpbledemo2Sta.indication_state = false;
  mpbledemo2Sta.init_send = false;
  mpbledemo2Sta.init_state = false;
  mpbledemo2Sta.send_data_seq = 0;
  mpbledemo2Sta.push_data_seq = 0;
  mpbledemo2Sta.seq = 0
}
```

4.4.6. 按键事件处理函数

这里主要定义了一个按键功能， MPBLEDEMO2_BUTTON_1。当 MPBLEDEMO2_BUTTON_1 按下的时候发送指定的信息。

```
int32_t mpbledemo2_button_handler(void *p_wcs, uint8_t pin_no)
{//首先要根据设备类型获取当前设备，这个会在注册设备的时候进行详细说明。
        ble_wechat_t *p_wcs_t = (ble_wechat_t*)p_wcs;
        if (m_mpbledemo2_handler == NULL) {
                m_mpbledemo2_handler = get_handler_by_type(PRODUCT_TYPE_MPBLEDEMO2);
        }/
        uint8_t *data = NULL;
        int len = 0;
        switch (pin_no)
    {
         case MPBLEDEMO2_BUTTON_1:
                        {
//填指令，这里是发送数据就填 CMD_SENDDAT，其实就是初始化 mpbledemo2_info.cmd。
                                ARGS_ITEM_SET(mpbledemo2_info, m_mpbledemo2_handler->m_data_produce_args, cmd,
CMD_SENDDAT);
//填发送数据长度，这里是发送数据就填 sizeof(SEND_HELLO_WECHAT)，其实就是初始化 mpbledemo2_info.
send_msg.len。
```

```
                    ARGS_ITEM_SET(mpbledemo2_info, m_mpbledemo2_handler->m_data_produce_args,
send_msg.len, sizeof(SEND_HELLO_WECHAT));
//填发送的数据，这里是发送数据就填 SEND_HELLO_WECHAT，其实就是初始化 mpbledemo2_info. send_msg.str。
                    ARGS_ITEM_SET(mpbledemo2_info, m_mpbledemo2_handler->m_data_produce_args,
send_msg.str, SEND_HELLO_WECHAT);
                }
            break;
        default:
            return 0;
    }
  //这里回调数据生产函数，来产生数据，这里也麻烦的使用了回调仅为说明其使用方法，之后在其他地方使用的时候也可以这样使用。
m_mpbledemo2_handler->m_data_produce_func(m_mpbledemo2_handler->m_data_produce_args, &data, &len);
        if(data == NULL)
        {
            return errorCodeProduce;
        }
//发送数据。
        ble_wechat_indicate_data(p_wcs_t,m_mpbledemo2_handler, data, len);
//发送数据之后回调相应的 free 函数，避免内存泄漏。
        m_mpbledemo2_handler->m_data_free_func(data,len);
        return 0;
}
```

4.4.7. AUTH 函数

Auth 函数内容与按键处理函数功能中的 MPBLEDEMO2_BUTTON_1 一样，只不过这里是填的是 CMD_AUTH 指令，让设备进行 auth。

```
int32_t device_auth(ble_wechat_t *p_wcs)
{
  if (m_mpbledemo2_handler == NULL) {
            m_mpbledemo2_handler = get_handler_by_type(PRODUCT_TYPE_MPBLEDEMO2);
        }
        uint8_t *data = NULL;
        int len = 0;
        ARGS_ITEM_SET(mpbledemo2_info, m_mpbledemo2_handler->m_data_produce_args, cmd, CMD_AUTH);
        m_mpbledemo2_handler->m_data_produce_func(m_mpbledemo2_handler->m_data_produce_args, &data, &len);
        if(data == NULL)
        {
            return errorCodeProduce;
        }
                ble_wechat_indicate_data(p_wcs, m_mpbledemo2_handler, data, len);
                m_mpbledemo2_handler->m_data_free_func(data,len);
        return 0;
  }
```

4.4.8. INIT 函数

init 函数内容与 auth 函数一样，只不过这里是填的是 CMD_INIT 指令，让设备进行 init。

```
int32_t   device_init(ble_wechat_t *p_wcs)
  {
        uint8_t *data = NULL;
        int len = 0;
        ARGS_ITEM_SET(mpbledemo2_info, m_mpbledemo2_handler->m_data_produce_args, cmd, CMD_INIT);
        m_mpbledemo2_handler->m_data_produce_func(m_mpbledemo2_handler->m_data_produce_args, &data, &len);
        if(data == NULL)
        {
            return errorCodeProduce;
        }
        ble_wechat_indicate_data(p_wcs, m_mpbledemo2_handler, data, len);
        m_mpbledemo2_handler->m_data_free_func(data,len);
        return 0;
}
```

4.4.9. 设备处理 write 事件函数

当发生写事件的时候，判断 indicate 是否被订阅，如果订阅了则设置状态位为 true，否则为 false。订阅成功之后才能发送数据，如进行 auth。

```
void mpbledemo2_on_write(ble_wechat_t *p_wcs, ble_evt_t * p_ble_evt)
{
  ble_gatts_evt_write_t * p_evt_write = &p_ble_evt->evt.gatts_evt.params.write;
  if (p_evt_write->handle == p_wcs->indicate_handles.cccd_handle&& p_evt_write->len == 2)
        {
            if (ble_srv_is_indication_enabled(p_evt_write->data))
                {
                    mpbledemo2Sta.indication_state = true;
                }
            else
                {
                    mpbledemo2Sta.indication_state = false;
                }
        }
}
```

4.4.10. 设备处理 disconnect 事件函数

设备连接断开时，复位设备，以便在重新连接是重新进行 auth 和 init 等相关操作。

```
void mpbledemo2_on_disconnect(ble_wechat_t *p_wcs, ble_evt_t * p_ble_evt)
{
  mpbledemo2_reset();
}
```

4.4.11. 设备处理 ble_evt 事件函数

处理 ble stack 抛上来的事件，根据事件调用相关函数。

```
void ble_mpbledemo2_on_ble_evt(void *p_wcs, void *p_ble_evt)
{
  ble_wechat_t *p_wcs_t = (ble_wechat_t*)p_wcs;
  ble_evt_t* p_ble_evt_t = (ble_evt_t*)p_ble_evt;
  switch (p_ble_evt_t->header.evt_id)
    {
        case BLE_GAP_EVT_DISCONNECTED:
            mpbledemo2_on_disconnect(p_wcs_t, p_ble_evt);
            break;
        case BLE_GATTS_EVT_WRITE:
            mpbledemo2_on_write(p_wcs_t, p_ble_evt);
            break;
        default:
            break;
    }
}
```

4.4.12. 设备主流程函数

这里主要流程是 auth 和 init。这个函数放在 main 函数的主循环里，并根据设备状态判断是否进行 auth 或者 init 操作。

```
void mpbledemo2_main_process(void *p_wcs)
{
  ble_wechat_t *p_wcs_t = (ble_wechat_t*)p_wcs;
  int error_code;
//如果 indicate 订阅成功&&还未 auth 成功&&还未发送 auth 数据包，就进行 auth 操作。
  if((mpbledemo2Sta.indication_state) && (!mpbledemo2Sta.auth_state) && (!mpbledemo2Sta.auth_send) )
  {
        error_code = device_auth(p_wcs_t);
        APP_ERROR_CHECK(error_code);
//auth 之后把 auth_send 状态设置为 true，以避免重复发 auth 包。
        mpbledemo2Sta.auth_send = true;

  }
//如果 auth 成功了&&还未 init 成功包&&还未发送 init 包，就进行 init 操作。
  if((mpbledemo2Sta.auth_state) && (!mpbledemo2Sta.init_state) && (!mpbledemo2Sta.init_send))
  {
        error_code = device_init(p_wcs_t);
        APP_ERROR_CHECK(error_code);
//init 之后把 init_send 状态设置为 true，以避免重复发 init 包。
        mpbledemo2Sta.init_send = true;
```

```
    }
    return ;
}
```

4.4.13. 设备释放内存函数

调用 mpbledemo2_data_produce_func 后使用。

```
void mpbledemo2_data_free_func(uint8_t *data, int len)
{
        if(data)
        free(data);
        data = NULL;
}
```

4.4.14. 设备生产数据函数

根据 void *args 的指向的结构体（mpbledemo2_info）参数来打包相应的数据包放入 uint8_t **r_data 指向的地址，并把长度放入 int *r_len。

```
void mpbledemo2_data_produce_func(void *args, uint8_t **r_data, int *r_len)
{
//获取 bluedemohead 的长度这是 mpbledemo2 与服务器通信定义的结构头。
        static uint16_t bleDemoHeadLen = sizeof(BlueDemoHead);
//把 args 指向 mpbledemo2_info 类型，以获取其内容。
        mpbledemo2_info *info = (mpbledemo2_info *)args;
// 这里把 BaseRequest 类型 basReq 设置为 null。
        BaseRequest basReq = {NULL};
//获取 BpFixHead 的长度这是微信服务与微信 APP 通信定义的结构头。
        static int fix_head_len = sizeof(BpFixHead);
//初始化 fix_head，BpFixHead 结构见开发文档。
//注意对包外的 2 字节数据转序，接收时也要转序
        BpFixHead fix_head = {0xFE, 1, 0, htons(ECI_req_auth), 0};
//req 包数据不能为 0 首先要对 seq++操作。
        mpbledemo2Sta.seq++;
        switch (info->cmd)
        {//auth 指令。
        case CMD_AUTH:
                {
                        //这里定义了很多版本的 auth 方式，包括：MD5 认证&AES 加密；MD5 认证不加密；MAC 认
证不加密
//根据需要在 mpbledemo2.h 中定义
  #if defined EAM_md5AndAesEnrypt
// MD5 认证&AES 加密模式
//生成加密数据块。
                        uint8_t deviceid[] = DEVICE_ID;
                        static uint32_t seq = 0x00000001;//
                        uint32_t ran = 0x11223344;//为了方便，这里定了一个数据作为随机数，在实际应用时请自行生产
```

随机数据。

```
ran = t_htonl(ran);
seq = t_htonl(seq);
uint8_t id_len = strlen(DEVICE_ID);
uint8_t* data = malloc(id_len+8);
if(!data){printf("\r\nNot enough memory!");return;}
memcpy(data,deviceid,id_len);
memcpy(data+id_len,(uint8_t*)&ran,4);
memcpy(data+id_len+4,(uint8_t*)&seq,4);
uint32_t crc = crc32(0, data, id_len+8);
crc = t_htonl(crc);
memset(data,0x00,id_len+8);
memcpy(data,(uint8_t*)&ran,4);
memcpy(data+4,(uint8_t*)&seq,4);
memcpy(data+8,(uint8_t*)&crc,4);
uint8_t CipherText[16];
AES_Init(key);
AES_Encrypt_PKCS7 (data, CipherText, 12, key);
if(data){free(data);data = NULL;}
AuthRequest authReq = {&basReq, true,{md5_type_and_id, MD5_TYPE_AND_ID_LENGTH},
PROTO_VERSION, AUTH_PROTO, (EmAuthMethod)AUTH_METHOD, true ,{CipherText, CIPHER_TEXT_LENGTH},
false, {NULL, 0}, false, {NULL, 0}, false, {NULL, 0},true,{DEVICE_ID,sizeof(DEVICE_ID)}};
seq++;
#endif

#if defined EAM_macNoEncrypt
//MAC 地址认证不加密模式
static uint8_t mac_address[MAC_ADDRESS_LENGTH];
get_mac_addr(mac_address);
AuthRequest authReq = {&basReq, false,{NULL, 0}, PROTO_VERSION, AUTH_PROTO,
(EmAuthMethod)AUTH_METHOD, false,{NULL, 0}, true, {mac_address, MAC_ADDRESS_LENGTH}, false, {NULL, 0},
false, {NULL, 0},true,{DEVICE_ID,sizeof(DEVICE_ID)}};
#endif

#if defined EAM_md5AndNoEnrypt
//MD5 认证不加密模式
AuthRequest authReq = {&basReq, true,{md5_type_and_id, MD5_TYPE_AND_ID_LENGTH},
PROTO_VERSION, (EmAuthMethod)AUTH_PROTO, (EmAuthMethod)AUTH_METHOD, false ,{NULL, 0}, false,
{NULL, 0}, false, {NULL, 0}, false, {NULL, 0},true,{DEVICE_ID,sizeof(DEVICE_ID)}};
#endif
//数据长度为 auth 结构体长度+固定头的长度。
*r_len = epb_auth_request_pack_size(&authReq) + fix_head_len;
*r_data = (uint8_t *)malloc(*r_len);
//调用相应函数打包 auth 包
```

```
                if(epb_pack_auth_request(&authReq, *r_data+fix_head_len, *r_len-fix_head_len)<0)
                {
                    *r_data = NULL;
                    return;
                }//填写固定头。
                fix_head.nCmdId = htons(ECI_req_auth);
                fix_head.nLength = htons(*r_len);
                fix_head.nSeq = htons(mpbledemo2Sta.seq);
                //连接固定头和打包好的数据。
                memcpy(*r_data, &fix_head, fix_head_len);
                return ;
        }
    //init 同上。
        case CMD_INIT:
            {
                //has challeange
                InitRequest initReq = {&basReq,false, {NULL, 0},true, {challeange, CHALLENAGE_LENGTH}};
                *r_len = epb_init_request_pack_size(&initReq) + fix_head_len;
            #if defined EAM_md5AndAesEnrypt
                uint8_t length = *r_len;
                uint8_t *p = malloc(AES_get_length( *r_len-fix_head_len));
                if(!p){printf("\r\nNot enough memory!");return;}
                *r_len = AES_get_length( *r_len-fix_head_len)+fix_head_len;
            #endif
                //pack data
                *r_data = (uint8_t *)malloc(*r_len);
                if(!(*r_data)){printf("\r\nNot enough memory!");return;}
                if(epb_pack_init_request(&initReq, *r_data+fix_head_len, *r_len-fix_head_len)<0)
                {*r_data = NULL;return;}
                //encrypt body
            #if defined EAM_md5AndAesEnrypt
                AES_Init(session_key);
                AES_Encrypt_PKCS7(*r_data+fix_head_len,p,length-fix_head_len,session_key);//
                memcpy(*r_data + fix_head_len, p, *r_len-fix_head_len);
                if(p)free(p);
            #endif
                fix_head.nCmdId = htons(ECI_req_init);
                fix_head.nLength = htons(*r_len);
                fix_head.nSeq = htons(mpbledemo2Sta.seq);
                memcpy(*r_data, &fix_head, fix_head_len);
                return  ;
            }
        case CMD_SENDDAT:
            {
```

```
//发送数据时就要按照设备与服务器的约定的格式填数据包。
                BlueDemoHead    *bleDemoHead =
(BlueDemoHead*)malloc(bleDemoHeadLen+info->send_msg.len);
                if(bleDemoHead == NULL){printf("\r\n Erroe! not enough memory!");return ;}
//按要求填 bleDemoHead 固定头的相关数据
                bleDemoHead->m_magicCode[0] = MPBLEDEMO2_MAGICCODE_H;
                bleDemoHead->m_magicCode[1] = MPBLEDEMO2_MAGICCODE_L;
                bleDemoHead->m_version = htons( MPBLEDEMO2_VERSION);
                bleDemoHead->m_totalLength = htons(bleDemoHeadLen + info->send_msg.len);
                bleDemoHead->m_cmdid = htons(sendTextReq);
                bleDemoHead->m_seq = htons(mpbledemo2Sta.seq);
                bleDemoHead->m_errorCode = 0;
                /*connect body and head.*/
                /*turn to uint8_t* befort offset.*/
//把要发送的数据当作包体连接到包头上
                memcpy((uint8_t*)bleDemoHead+bleDemoHeadLen, info->send_msg.str, info->send_msg.len);
//把上面的“包头+包体”数据放入发送数据结构体 sendDatReq 的数据段中。
                SendDataRequest sendDatReq = {&basReq, {(uint8_t*) bleDemoHead, (bleDemoHeadLen +
info->send_msg.len)}, false, (EmDeviceDataType)NULL};
                *r_len = epb_send_data_request_pack_size(&sendDatReq) + fix_head_len;
#if defined EAM_md5AndAesEnrypt
                uint16_t length = *r_len;
                uint8_t *p = malloc(AES_get_length( *r_len-fix_head_len));
                if(!p){printf("\r\nNot enough memory!");return;}
                *r_len = AES_get_length( *r_len-fix_head_len)+fix_head_len;
        #endif
                *r_data = (uint8_t *)malloc(*r_len);
//打包要发送的数据结构体
                if(epb_pack_send_data_request(&sendDatReq, *r_data+fix_head_len, *r_len-fix_head_len)<0)
                {
                    *r_data = NULL;
                #if defined EAM_md5AndAesEnrypt
                    if(p){free(p);
                    p = NULL;}
                #endif
                    return;
                }
        #if defined EAM_md5AndAesEnrypt
                //encrypt body
                AES_Init(session_key);
                AES_Encrypt_PKCS7(*r_data+fix_head_len,p,length-fix_head_len,session_key);//Ô Ê¼ÊýÝ ³¤¶È
                memcpy(*r_data + fix_head_len, p, *r_len-fix_head_len);
                if(p){free(p); p = NULL;}
        #endif
```

```
                fix_head.nCmdId = htons(ECI_req_sendData);
                fix_head.nLength = htons(*r_len);
                fix_head.nSeq = htons(mpbledemo2Sta.seq);
                memcpy(*r_data, &fix_head, fix_head_len);
                free(bleDemoHead);
                bleDemoHead = NULL;
                return ;
        }
    }
}
```

4.4.15. 设备处理数据函数

用来处理传递进来的数据（为了更好的说明保留了 log 代码）

```
int mpbledemo2_data_consume_func(uint8_t *data, int len)
{
//把数据指向 BpFixHead 固定头
        BpFixHead *fix_head = (BpFixHead *)data;
        uint8_t fix_head_len = sizeof(BpFixHead);
//打印收到的数据内容好部分包头中的数据
//使用包头中的 2 字节数据时注意转序。
//通过包头的 nCmdId 来判断数据包的类型
        switch(ntohs(fix_head->nCmdId))
        {
            case ECI_none:
                {
                }
                break;
            case ECI_resp_auth:
                {
//如果是 authResp 包，则调用相应的解包函数来解包包体。
                    AuthResponse* authResp;
                    authResp = epb_unpack_auth_response(data+fix_head_len,len-fix_head_len);
                    if(!authResp){return errorCodeUnpackAuthResp;}
//判断是 base_response 中的错误码，如果 auth 成功，则设置 auth 状态为 true。//否则打印相关错误信息。
                    if(authResp->base_response)
                    {
                        if(authResp->base_response->err_code == 0)
                        {
                            mpbledemo2Sta.auth_state = true;
                        }
                        else
                        {
                            epb_unpack_auth_response_free(authResp);
                            return authResp->base_response->err_code;
```

```
                }
            }
            #if defined EAM_md5AndAesEnrypt
            //从 authresp 中取出 sessionkey
                if(authResp->aes_session_key.len)
                {
                    AES_Init(key);
                    AES_Decrypt(session_key,authResp->aes_session_key.data,authResp->aes_session_key
.len,key);
                }
            #endif
                epb_unpack_auth_response_free(authResp);
            }
            break;
//sendDataResp
        case ECI_resp_sendData:
            {
//解包 sendDataResp
            #if defined EAM_md5AndAesEnrypt
                uint32_t length = len- fix_head_len;//加密后数据长度
                uint8_t *p = malloc (length);
                if(!p){printf("\r\nNot enough memory!"); if(data)free(data);data = NULL; return 0;}
                AES_Init(session_key);
                //解密数据
                AES_Decrypt(p,data+fix_head_len,len- fix_head_len,session_key);
                uint8_t temp;
                temp = p[length - 1];//算出填充长度
                len = len - temp;//取加密前数据长度
                memcpy(data + fix_head_len, p ,length -temp);//把明文放回
                if(p){free(p);p = NULL;}
            #endif
            SendDataResponse *sendDataResp;
            sendDataResp = epb_unpack_send_data_response(data+fix_head_len,len-fix_head_len);
            if(!sendDataResp)
                {
                    return errorCodeUnpackSendDataResp;
                }
                if(sendDataResp->base_response->err_code)
                {
                    epb_unpack_send_data_response_free(sendDataResp);
                    return sendDataResp->base_response->err_code;
                }
                epb_unpack_send_data_response_free(sendDataResp);
        }
```

```
                break;
            case ECI_resp_init:
                {
                #if defined EAM_md5AndAesEnrypt
                    uint32_t length = len- fix_head_len;//加密后数据长度
                    uint8_t *p = malloc (length);
                    if(!p){printf("\r\nNot enough memory!"); if(data)free(data);data = NULL; return 0;}
                    AES_Init(session_key);
                    //解密数据
                    AES_Decrypt(p,data+fix_head_len,len- fix_head_len,session_key);
                    uint8_t temp;
                    temp = p[length - 1];//算出填充长度
                    len = len - temp;//取加密前数据长度
                    memcpy(data + fix_head_len, p ,length -temp);//把明文放回
                    if(p){free(p);p = NULL;}
                #endif
                InitResponse *initResp = epb_unpack_init_response(data+fix_head_len, len-fix_head_len);
                if(!initResp) {return errorCodeUnpackInitResp; }
                    if(initResp->base_response)
                    {
                        if(initResp->base_response->err_code == 0)
                        {
//检查 challenge 返回值
                            if(initResp->has_challeange_answer)
                            {
                                if(crc32(0,challeange,CHALLENAGE_LENGTH) ==
initResp->challeange_answer)
                                {
                                    mpbledemo2Sta.init_state = true;
                                }
                            }
                            mpbledemo2Sta.wechats_switch_state = true;
                        }
                        else
                        {
                            epb_unpack_init_response_free(initResp);
                            return initResp->base_response->err_code;
                        }
                    }
                epb_unpack_init_response_free(initResp);
            }
                break;
            case ECI_push_recvData:
                {//收到数据
```

```
                #if defined EAM_md5AndAesEnrypt
                    uint32_t length = len- fix_head_len;//加密后数据长度
                    uint8_t *p = malloc (length);
                    if(!p){printf("\r\nNot enough memory!"); if(data)free(data);data = NULL; return 0;}
                    AES_Init(session_key);
                    //解密数据
                    AES_Decrypt(p,data+fix_head_len,len- fix_head_len,session_key);
                    uint8_t temp;
                    temp = p[length - 1];//算出填充长度
                    len = len - temp;//取加密前数据长度
                    memcpy(data + fix_head_len, p ,length -temp);//把明文放回
                    if(p){free(p);p = NULL;}
                #endif
                RecvDataPush *recvDatPush;
                recvDatPush = epb_unpack_recv_data_push(data+fix_head_len, len-fix_head_len);
                if(!recvDatPush) {return errorCodeUnpackRecvDataPush; }
        //把收到的数据包包体中的数据段指向 BlueDemoHead 类型
                BlueDemoHead *bledemohead = (BlueDemoHead*)recvDatPush->data.data;
        //通过数据内容中的命令号判断是开灯还是关灯，并执行相应操作
                if(ntohs(bledemohead->m_cmdid ) == openLightPush)
                {
                    mpbledemo2Sta.light_state = true;
                    light_on(MPBLEDEMO2_LIGHT);
                }
                else if(ntohs(bledemohead->m_cmdid )   == closeLightPush)
                {
                        mpbledemo2Sta.light_state = false;
                        light_off(MPBLEDEMO2_LIGHT);
                }
                epb_unpack_recv_data_push_free(recvDatPush);
            }
                    break;
        //切换界面 push 通知
                case ECI_push_switchView:
                    {
                #if defined EAM_md5AndAesEnrypt
                    uint32_t length = len- fix_head_len;//加密后数据长度
                    uint8_t *p = malloc (length);
                    if(!p){printf("\r\nNot enough memory!"); if(data)free(data);data = NULL; return 0;}
                    AES_Init(session_key);
                    //解密数据
                    AES_Decrypt(p,data+fix_head_len,len- fix_head_len,session_key);
                    uint8_t temp;
                    temp = p[length - 1];//算出填充长度
```

```
                    len = len - temp;//取加密前数据长度
                    memcpy(data + fix_head_len, p ,length -temp);//把明文放回
                    if(p){free(p);p = NULL;}
                #endif
                    SwitchViewPush *swichViewPush;
                    swichViewPush = epb_unpack_switch_view_push(data+fix_head_len,len-fix_head_len);
                    if(!swichViewPush)
                    {
                        return errorCodeUnpackSwitchViewPush;
                    }
                    epb_unpack_switch_view_push_free(swichViewPush);
                }
                    break;
//IOS 平台微信进入后台通知。
                case ECI_push_switchBackgroud:
                    {
                #if defined EAM_md5AndAesEnrypt
                    uint32_t length = len- fix_head_len;//加密后数据长度
                    uint8_t *p = malloc (length);
                    if(!p){printf("\r\nNot enough memory!"); if(data)free(data);data = NULL; return 0;}
                    AES_Init(session_key);
                    //解密数据
                    AES_Decrypt(p,data+fix_head_len,len- fix_head_len,session_key);
                    uint8_t temp;
                    temp = p[length - 1];//算出填充长度
                    len = len - temp;//取加密前数据长度
                    memcpy(data + fix_head_len, p ,length -temp);//把明文放回
                    if(p){free(p);p = NULL;}
                #endif
                    SwitchBackgroudPush *switchBackgroundPush =
epb_unpack_switch_backgroud_push(data+fix_head_len,len-fix_head_len);
                    if(! switchBackgroundPush)
                    {
                        return errorCodeUnpackSwitchBackgroundPush;
                    }
                    epb_unpack_switch_backgroud_push_free(switchBackgroundPush);
                }
                    break;
                case ECI_err_decode:
                    break;
                default:
                    break;
            }
            return 0;
```

```
}
```

4.4.16. 设备注册结构定义

这是根据设备需要自定义的一个结构体，实际使用时可以根据需要添加、删减。

```
data_handler mpbledemo2_data_handler = {
        .m_product_type = PRODUCT_TYPE_MPBLEDEMO2, //设备类型信息
        .m_data_produce_func = &mpbledemo2_data_produce_func, //设备生产数据函数接口
        .m_data_free_func = &mpbledemo2_data_free_func, //设备释放数据接口
        .m_data_consume_func = &mpbledemo2_data_consume_func, //设备处理数据函数接口
        .m_data_error_func = &mpbledemo2_data_error_func, //设备错误处理函数接口
        .m_data_on_ble_evt_func = &ble_mpbledemo2_on_ble_evt, //设备处理 ble 事件函数接口
        .m_data_init_func = &mpbledemo2_init, //设备初始化函数接口
        .m_data_main_process_func = &mpbledemo2_main_process, //设备主流程函数接口
        .m_data_button_handler_func = &mpbledemo2_button_handler, //设备按键处理函数接口
        .m_data_produce_args = &m_info, //设备数据
        .next = NULL //设备存储链表指针
};
```

4.5. ble_wechat_util

这里定义了微信服务与注册设备、转序等有关的函数以及结构体等。

4.5.1. 设备存储链表节点

节点里定义了设备函数接口与参数等内容。

```
typedef struct data_handler{
        int m_product_type;
        data_produce_func           m_data_produce_func;
        data_free_func                      m_data_free_func;
        data_consume_func           m_data_consume_func;
        data_error_func             m_data_error_func;
        data_on_ble_evt_func        m_data_on_ble_evt_func;
        data_init_func                      m_data_init_func;
        data_main_process_func      m_data_main_process_func;
        data_button_handler_func    m_data_button_handler_func;
        void *m_data_produce_args;
        struct data_handler *next;
} data_handler;
```

4.5.2. 注册设备宏

通过设备名称把设备接口节点存储到链表。

```
#define REGISTER(NAME)    \
        do {                                    \
```

```
                data_handler *tmp = &NAME##_data_handler; \
                tmp->next = first_handler.next; \
                first_handler.next = tmp;    \
        } while(0)
```

4.5.3. 参数设置宏

把相应的命令、参数初始化到设备参数结构体中，如 mpbledemo2_info。

```
#define ARGS_ITEM_SET(ARGS_TYPE, ARGS_POINTER, ITEM_NAME, ITEM_VALUE)        \
        do {   \
                ARGS_TYPE *tmp = (ARGS_TYPE *)ARGS_POINTER;    \
                tmp->ITEM_NAME = ITEM_VALUE;   \
        } while(0)
```

4.5.4. 固定头

该固定头是设备与微信通信格式的固定头。

```
typedef struct
{
        unsigned char bMagicNumber;
        unsigned char bVer;
        unsigned short nLength;
        unsigned short nCmdId;
        unsigned short nSeq;
} BpFixHead;
```

4.5.5. 通过类型查找设备

通过定义的设备类型在链表中查找注册进来的设备并返回节点指针。

```
data_handler* get_handler_by_type(int type)
{
        data_handler* handler = &first_handler;
        while(handler->next != NULL) {
                handler = handler->next;
                if (handler->m_product_type == type) {
                        return handler;
                }
        }
        return NULL;
}
```

4.5.6. 转序函数

这里定义了转序函数，此函数用来转序数据中的 short 或者 long 型数据。
将一个无符号短整型或者长整型的主机数值转换为网络字节顺序，或者反过来。

```
unsigned short htons(unsigned short val)
{
        unsigned short tmp = (val & 0xFF00) >> 8;
        val = (val << 8) & 0xFF00;
        val |= tmp;

        return val;
}
```

将一个无符号短整形数从网络字节顺序转换为主机字节顺序。

```
unsigned short ntohs(unsigned short val)
{
        unsigned short tmp = (val & 0x00FF) << 8;
        val = (val >> 8) & 0x00FF;
        val |= tmp;
        return val;
}
#define BigLittleSwap16(A)    ((((uint16_t)(A) & 0xff00) >> 8) | \
                                    (((uint16_t)(A) & 0x00ff) << 8))
#define BigLittleSwap32(A)    ((((uint32_t)(A) & 0xff000000) >> 24) | \
                                    (((uint32_t)(A) & 0x00ff0000) >> 8) | \
                                    (((uint32_t)(A) & 0x0000ff00) << 8) | \
                                    (((uint32_t)(A) & 0x000000ff) << 24))
int checkCPUendian()
{
    union{
            unsigned long i;
            uint8_t s[4];
    }c;
    c.i = 0x12345678;
    return (0x12 == c.s[0]);
}
unsigned long t_htonl(unsigned long h)
{
    return checkCPUendian() ? h : BigLittleSwap32(h);
}
unsigned long t_ntohl(unsigned long n)
{
    return checkCPUendian() ? n : BigLittleSwap32(n);
}
unsigned short t_htons(unsigned short h)
{
    return checkCPUendian() ? h : BigLittleSwap16(h);
}
```

```
unsigned short t_ntohs(unsigned short n)
{
        return checkCPUendian() ? n : BigLittleSwap16(n);
}
```

4.6. epb.c 与 epb_MmBp.c

这里主要是与微信服务 protobuf（C 版本）有关的内容，不过多介绍，可以直接使用。

4.7. Md5.c

这里主要是计算设备类型加设备 ID 字符摘要的内容，md5 可以实现算好烧录到设备中来也可以在设备第一次上电的时候在本地算好，当然这时仍需要烧录设备 ID。之后的微信版本（具体版本请等通知）会有直接通过 mac 地址认证的方式来认证，但是这种方法会有一定的安全隐患。

关于 md5 摘要这里也不做具体说明。

4.8. Aes.c

这里是与 AES 加解密相关的函数，packs7 填充。注意解密时进行还原并算出原始数据长度后再进行解包。

4.9. Crc32.c

这里是与 CRC32 相关的函数，在此不做介绍。

5. 添加设备与设备数据流向

5.1. 添加设备

5.1.1. 定义设备

首先需要定义一个设备，这里定义了一个叫 mpbledemo2 的设备，与 Mpbledemo2 设备相关的函数与定义全部在 mpbledemo2.h 与 mpbledemo2.c 中定义。如 mpbledemo2 需要的硬件资源、命令、协议版本等等。在 mpbledemo2 中我们把设备需要的硬件资源与系统需要的资源分开了，如设备的灯与按键是在设备的头文件中定义的；系统需要的广播 led 灯、系统复位按键、以及串口是在 comsource 的头文件中定义的。这样会更便于管理。

然后根据 demo 的开发要求定义相应的数据结构。如数据头、参数结构、错误号等。

之后就是设备需要的功能函数，如 data_consume_func、data_produce_func、data_free_func 等等。

5.1.2. 把设备添加到服务与系统中

添加到系统中的步骤。

- 在主函数初始化的时候需要把设备注册到设备列表里面；
- 初始化指向存储设备接口节点的指针；

- 通过该指针调用设备的初始化函数，初始化设备所需要的资源；

5.1.3. 把设备添加到服务中

服务与设备有关联的地方主要有三个。

- 设备发送数据需要使用的 ble_wechat_indicate_data（服务）
- 设备接收数据需要使用的 data_consume_func（设备）
- 有错误时需要处理错误的 data_error_func（设备）

5.2. 设备数据流向

在这里简单的举个接收与发送例子。

5.2.1. 数据接收

微信 app 向设备 push 数据是通过 write 特征字，有 write 事件后。ble_wechat_on_ble_evt 函数把 write 事件交给微信服务的私有函数 on_write。on_write 函数接受完数据之后调用注册设备的 data_consume_func 函数。然后设备的 data_consume_func 对数据进行解包与处理。

5.2.2. 发送数据

发送数据时设备直接调用微信服务的 ble_wechat_indicate_data 函数，ble_wechat_indicate_data 函数调用 ble_wechat_indicate_data_chunk 函数进行分帧发送（如果需要分帧发送的话）。发送完一帧之后会有一个确认事件 BLE_GATTS_EVT_HVC，当收到该事件后会再次调用 ble_wechat_indicate_data_chunk 函数发送剩余帧，知道数据发送完毕。