

青风带你玩蓝牙 nRF51822 系列教程.....	2
-----作者: 青风.....	2
作者: 青风.....	3
出品论坛: www.qfv8.com	3
淘宝店: http://qfv5.taobao.com	3
QQ 技术群: 346518370.....	3
硬件平台: 青云 QY-nRF52832 开发板.....	3
3.1 蓝牙主机 1 拖 8 组网详解.....	3
1: nRF52832 蓝牙主机的主程序流程:	3
1.1 主机程序框架搭建:	3
1.2 主机蓝牙组网详解:	7
1.3 主机和从机通信通道搭建:	10
2 应用与调试.....	19
2.1 软件准备:	19
2.2 实验现象:	25

作者: 青风**出品论坛: www.qfv8.com****淘宝店: <http://qfv5.taobao.com>****QQ 技术群: 346518370****硬件平台: 青云 QY-nRF52832 开发板**

3.1 蓝牙主机 1 拖 8 组网详解

前面的讲义里讲解了部分主机蓝牙的基础知识, 么今天这一讲将来给大家分析下蓝牙主机如何和从机进行组网, 当然由于蓝牙协议栈的现在不使用 mesh 的话, 只能组网为微微网, 相关的基础知识, 讲在这一节将详细的进行讨论。

由于蓝牙 5.0 的微微网最大可以连接 20 个从机的, 不过因为板子的 led 有限你只能看到四个从机连接。本例通过主机和从机互相控制, 实现了一个主机和 8 个从机互动。当按键下从机按键后, 主机的灯会对应亮, 同时主机的按键也可以控制从机。如果你自己的开发工具上的按键, LED 灯同, 修改下程序中的硬件端口就可以。

1: nRF52832 蓝牙主机的主程序流程:

1.1 主机程序框架搭建:

首先我们看下 nrf52832 的主机程序如下所示, 下面来分析下主机流程:

```
int main(void)
{
    // Initialize.
    log_init();//打印初始化
    timer_init();//软件定时器初始化
    leds_init();//led 灯初始化
    buttons_init();//按键初始化
    power_management_init();//能量管理初始化
    ble_stack_init();//协议栈初始化
    gatt_init();//gatt 初始化
    db_discovery_init();//蓝牙数据 Database 发现初始化
```

```
lbs_c_init();//主机 led 灯服务初始化
ble_conn_state_init();//连接状态初始化

// Start execution.
NRF_LOG_INFO("Multilink example started.");
scan_start();//开始扫描

for (;;)
{
    idle_state_handle();
}
```

1: 首先是按键和 LED 初始化、定时器初始化, 这三个硬件初始化设置和从机初始化一样, 编写的时候对比从机的编写模式写, 区别不大。同时 power_management_init() 能量管理初始化函数, 由于编写结构相同, 因此和从机中的初始化一样, 没有任何变化。这里主要说明下按键初始化和按键中断:

```
static void buttons_init(void)
{
    ret_code_t err_code;

    // The array must be static because a pointer to it will be saved in the button handler module.
    static app_button_cfg_t buttons[] =
    {
        {LEDBUTTON_BUTTON, false, BUTTON_PULL, button_event_handler},//按键中断
    };
    err_code = app_button_init(buttons, ARRAY_SIZE(buttons), BUTTON_DETECTION_DELAY);
    APP_ERROR_CHECK(err_code);
}

static void button_event_handler(uint8_t pin_no, uint8_t button_action)
{
    ret_code_t err_code;

    switch (pin_no)
    {
        case LEDBUTTON_BUTTON:
            err_code = led_status_send_to_all(button_action);//主机按下后, 点亮所有从机
            {
                NRF_LOG_INFO("LBS write LED state %d", button_action);
            }
            break;
        default:
            APP_ERROR_HANDLER(pin_no);
    }
}
```

```

        break;
    }
}

```

2. ble_stack_init() 协议栈初始化函数对比从机点灯部分的《协议栈初始化详解》，基本结构没有变化，设备变化的地方有下面几个地方：

(1) 一个变化是 nrf_sdh_ble_default_cfg_set 函数中设置的从机和主机角色变化：

nrf_config.h 文件中从机点灯程序的设置如下：

```

11411 // <o> NRF_SDH_BLE_PERIPHERAL_LINK_COUNT - Maximum number of peripheral links.
11412 #ifndef NRF_SDH_BLE_PERIPHERAL_LINK_COUNT
11413 #define NRF_SDH_BLE_PERIPHERAL_LINK_COUNT 1
11414 #endif
11415
11416 // <o> NRF_SDH_BLE_CENTRAL_LINK_COUNT - Maximum number of central links.
11417 #ifndef NRF_SDH_BLE_CENTRAL_LINK_COUNT
11418 #define NRF_SDH_BLE_CENTRAL_LINK_COUNT 0
11419 #endif
11420 #endif
11421

```

主机 1 拖 8 点灯程序设置如下：

```

11414 // <o> NRF_SDH_BLE_PERIPHERAL_LINK_COUNT - Maximum number of peripheral links.
11415 #ifndef NRF_SDH_BLE_PERIPHERAL_LINK_COUNT
11416 #define NRF_SDH_BLE_PERIPHERAL_LINK_COUNT 0
11417 #endif
11418
11419 // <o> NRF_SDH_BLE_CENTRAL_LINK_COUNT - Maximum number of central links.
11420 #ifndef NRF_SDH_BLE_CENTRAL_LINK_COUNT
11421 #define NRF_SDH_BLE_CENTRAL_LINK_COUNT 8
11422 #endif
11423
11424 // <o> NRF_SDH_BLE_TOTAL_LINK_COUNT - Total link count.
11425 // <i> Maximum number of total concurrent connections using the default configuration.
11426
11427 #ifndef NRF_SDH_BLE_TOTAL_LINK_COUNT
11428 #define NRF_SDH_BLE_TOTAL_LINK_COUNT 8
11429 #endif
11430

```

表示该主机设备可使用 8 个中心主机设备进行连接的链路，也就是可以连接 8 路从机。

(2) 另外一个变化的是观察者里面的蓝牙处理函数 ble_evt_handler，一是因为主机和从机所发生的蓝牙事情是不同的。比如：BLE_GAP_EVT_ADV_REPORT 蓝牙 GAP 蓝牙广播报告事件，只有主机才会扫描报告。二是因为主机和从机的角色不多，再处理相同蓝牙事件的时候所做的处理是不相同的，例如发生 BLE_GAP_EVT_CONNECTED 事件时：

从机：

```

363 case BLE_GAP_EVT_CONNECTED:
364     NRF_LOG_INFO("Connected");
365     err_code = bsp_indication_set(BSP_INDICATE_CONNECTED); // 指示灯点亮
366     APP_ERROR_CHECK(err_code);
367     m_conn_handle = p_ble_evt->evt.gap_evt.conn_handle;
368     err_code = nrf_ble_qwr_conn_handle_assign(&m_qwr, m_conn_handle); // 分配连接句柄给队列写入模块
369     APP_ERROR_CHECK(err_code);
370     break;

```

主机：


```

261     switch (p_ble_evt->header.evt_id)
262     {
263         // Upon connection, check which peripheral has connected, initiate DB
264         // discovery, update LEDs status and resume scanning if necessary.
265         case BLE_GAP_EVT_CONNECTED:
266         {
267             NRF_LOG_INFO("Connection 0x%x established, starting DB discovery.",
268                           p_gap_evt->conn_handle);
269
270             APP_ERROR_CHECK_BOOL(p_gap_evt->conn_handle < NRF_SDH_BLE_CENTRAL_LINK_COUNT); //连接句柄如果没有分配完
271
272             err_code = ble_lbs_c_handles_assign(&m_lbs_c[p_gap_evt->conn_handle],
273                                                 p_gap_evt->conn_handle,
274                                                 NULL); //分频连接句柄
275
276             APP_ERROR_CHECK(err_code);
277
278             err_code = ble_db_discovery_start(&m_db_disc[p_gap_evt->conn_handle],
279                                              p_gap_evt->conn_handle); //开始发现对应句柄的蓝牙服务
280             if (err_code != NRF_ERROR_BUSY)
281             {
282                 APP_ERROR_CHECK(err_code);
283             }
284         }
285     }

```

由于主机需要连接连接 8 个从机，因此，每个从机都要分配一个操作句柄，同时需要判断是否达到最大的连接数量，通过一个扫描 LED 灯进行指示，如果到了则关闭扫描 LED 灯，代码如下图所示：

```

286     bsp_board_led_on(CENTRAL_CONNECTED_LED); //更新LED灯
287     if (ble_conn_state_central_conn_count() == NRF_SDH_BLE_CENTRAL_LINK_COUNT) //如果达到最大的连接数量
288     {
289         bsp_board_led_off(CENTRAL_SCANNING_LED); //关掉扫描LED
290     }
291     else
292     {
293         // Resume scanning.
294         bsp_board_led_on(CENTRAL_SCANNING_LED); //否则继续扫描
295         scan_start();
296     }
297     } break; // BLE_GAP_EVT_CONNECTED
298

```

(3) 当断开链接的时候，由于有多个链接，所以需要给你一个连接指示灯，判断是否连接设备为 0。同时端口后，主机开始从新扫描，代码如下：

```

301     case BLE_GAP_EVT_DISCONNECTED:
302     {
303         NRF_LOG_INFO("LBS central link 0x%x disconnected (reason: 0x%x)",
304                       p_gap_evt->conn_handle,
305                       p_gap_evt->params.disconnected.reason);
306
307         if (ble_conn_state_central_conn_count() == 0) //如果连接设备为0
308         {
309             err_code = app_button_disable();
310             APP_ERROR_CHECK(err_code);
311
312             // Turn off connection indication LED
313             bsp_board_led_off(CENTRAL_CONNECTED_LED); //关掉连接指示灯
314         }
315
316         // Start scanning
317         scan_start(); //开始扫描
318
319         // Turn on LED for indicating scanning
320         bsp_board_led_on(CENTRAL_SCANNING_LED);
321     }
322     } break;

```

在蓝牙状态处理回调函数中的其他地方，将在下面的章节陆续展开。

3. db_discovery_init(); // 蓝牙数据 Database 发现初始化

数据发现初始化，初始化设置几个标志位声明，这个部分在《主机串口篇》有详细讲解，发现过程和主机串口过程，只是发现的服务数目有增加。这里的区别将在下面章节具体论述。

4. gatt_init(); // gatt 初始化

GATT 初始化的主要是分配 GATT 事件句柄, 同时设置主机的 MTU 大小, 主机 MTU 大小需要和从机设置的 MTU 大小相同, 代码如下:

```
void gatt_init(void)
{
    ret_code_t err_code;
    //初始化 GATT, 分配 GATT 句柄
    err_code = nrf_ble_gatt_init(&m_gatt, gatt_evt_handler);
    APP_ERROR_CHECK(err_code);
    //设置主机的 MTU 大小
    err_code = nrf_ble_gatt_att_mtu_central_set(&m_gatt, NRF_SDH_BLE_GATT_MAX_MTU_SIZE);
    APP_ERROR_CHECK(err_code);
}
```

5. lbs_c_init();//客户端初始化, 也就是主机端

这个函数后面会具体展开讲述, 客户端取代之前的手机作为主机, 那么首先就需要配置这个客户端, 客户端初始化函数, 主要功能就是使能通知事件, 并且设置主机设备的触发事件。

5. scan_start();//开始扫描

主机扫描篇有详细论述, 这里不再累述。

6. ble_conn_state_init();//连接状态函数

这个函数比较简单, 就是初始化连接状态, 声明状态标志位后进行初始化清零。

上面的内容就是主机程序中主函数框架的基本构造, 下面的讲解我们都会进一步具有涉及到的如何组网的细节问题, 大家一定要深入理解。

1.2 主机蓝牙组网详解:

蓝牙的使用无非就是进行数据的通信, 从机通常都是作为服务端(一次数据交互中数据的提供者), 而主机通常都是作为客户端(一次数据交互中数据的使用者)。并不是固定的, 从机也可以作为客户端, 主机也可以作为服务端。从机和主机的概念是针对链路层来说的, 而客户端和服务端是针对 GATT 层来说的。这在主从连接的时候就必须弄清楚了。之前一讲的主机串口内容中, 我们基本弄清楚了: 主机扫描到发起连接, 然后主机发现服务, 建立主从通信通道的这一过程。那么区别与之前的内容, 这一讲的核心问题就是需要弄清楚下面两个问题:

1: 主机设备如何判断哪个从机是我们需要的?

2: 如何实现多个设备的连接?

1.2.1 主机如何判断哪个从机是需要连接的

当主机工程 `ble_app_multilink_central` 在主函数中初始化完成后就会调用 `scan_start()` 开始监听从机的 BLE 广播。每当监听到 BLE 广播时协议栈就会给上层一个广播事件，该事件由 `ble_evt_dispatch` 派发函数递交给 `on_ble_evt` 函数。

那么主机和从机设备的连接过程实际上是派发函数来处理，在协议栈初始化时初始化：`on_ble_evt(p_ble_evt);` // 蓝牙处理应用事件派发

首先来谈下 `on_ble_evt(p_ble_evt)`:

当我们在主函数里使用 `sd_ble_gap_scan_start` 函数启动主机扫描，如果发现了从机广播，则产生 `BLE_GAP_EVT_ADV_REPORT` 事件，也就是广播报告，那么派发函数就判断执行解析广播的操作，代码如下：

```

254 static void ble_evt_handler(ble_evt_t const * p_ble_evt, void * p_context)
255 {
256     ret_code_t err_code;
257
258     // For readability.
259     ble_gap_evt_t const * p_gap_evt = &p_ble_evt->evt.gap_evt;
260
261     switch (p_ble_evt->header.evt_id)
262     {
263         // Upon connection, check which peripheral has connected, initiate DB
264         // discovery, update LEDs status and resume scanning if necessary.
265         case BLE_GAP_EVT_CONNECTED:
266             // Upon disconnection, reset the connection handle of the peer which disconnected, update
267             // the LEDs status and start scanning again.
268         case BLE_GAP_EVT_DISCONNECTED:
269             // Upon BLE_GAP_EVT_ADV_REPORT:
270             on_adv_report(&p_gap_evt->params.adv_report); // 报告扫描设备
271             break;
272     }
273 }

```

当产生协议栈产生 `BLE_GAP_EVT_ADV_REPORT` 事件，那么就通过广播报告方式，调用 `on_adv_report` 函数，这个函数提取广播完整名字，这个方法在上一讲扫描器的应用中也有类似报告。

```

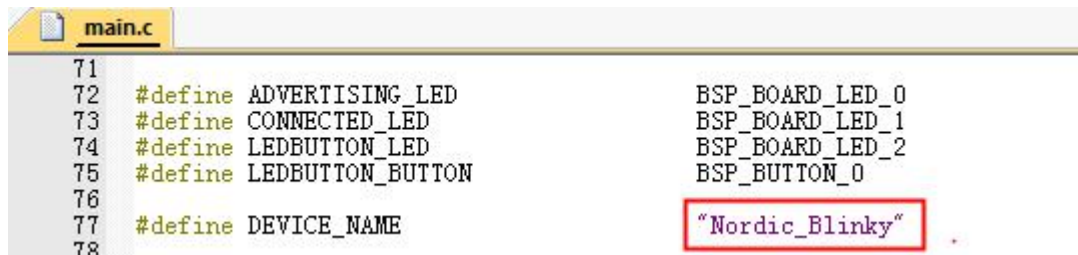
222 - */
223 static void on_adv_report(ble_gap_evt_adv_report_t const * p_adv_report) // 广播报告
224 {
225     ret_code_t err_code;
226
227     if (ble_advdata_name_find(p_adv_report->data.p_data,
228                             p_adv_report->data.len,
229                             m_target_periph_name)) // 发现指定名称的设备
230     {
231         // Name is a match, initiate connection. 对指定参数进行连接
232         err_code = sd_ble_gap_connect(&p_adv_report->peer_addr,
233                                     &m_scan_params,
234                                     &m_connection_param,
235                                     APP_BLE_CONN_CFG_TAG);
236         if (err_code != NRF_SUCCESS)
237         {
238             NRF_LOG_ERROR("Connection Request Failed, reason %d", err_code);
239         }
240     }
241     else
242     {
243         err_code = sd_ble_gap_scan_start(NULL, &m_scan_buffer); // 如果没发现继续扫描
244         APP_ERROR_CHECK(err_code);
245     }
246 }

```


通过 `ble_advdata_name_find` 函数发现指定广播名称的广播数据, 如果是的话, 则通过 `sd_ble_gap_connect` 函数对目标发起 `gap` 连接。

这样主机设备就可以判断哪个设备是我们需要连接的从机了, 所以这里, 在主程序和从机程序中, 广播名称都需要设置为: `Nordic_Blinky`

从机程序中:



```
71
72 #define ADVERTISING_LED      BSP_BOARD_LED_0
73 #define CONNECTED_LED      BSP_BOARD_LED_1
74 #define LEDBUTTON_LED      BSP_BOARD_LED_2
75 #define LEDBUTTON_BUTTON    BSP_BUTTON_0
76
77 #define DEVICE_NAME          "Nordic_Blinky"
78
```

主机程序中:

```
96 static char const m_target_periph_name[] = "Nordic_Blinky";
```

这种连接方法和**第一讲主机的蓝牙串口有区别的**, 第一讲的蓝牙串口是通过判断是否需要的从机 UUID 来启动连接。相同的都是调用 `sd_ble_gap_connect` 函数启动连接。

1.2.2 多个从机设备的连接

在上面的程序中, 当找到了名字为 `Nordic_Blinky` 设备后就关闭扫描侦听而发起连接, 也就是这个时候主机已经不能再监听别的 `ble` 设备的广播了, 自然也就不能再和别的设备发起连接了。那么这时候最多只能连接 1 个从设备。

但是为了组成微微网, 主机最多需要连接 8 个从机, 那么就必须能够继续启动扫描侦听, 并且周围还有别的广播设备, 并且其名字也是 `Nordic_Blinky`, 那么收到该广播时, 就能够在广播事件处理函数再次对这第二个设备发起连接。

所以主机例子中虽然在找到了一个设备名为 `Nordic_Blinky` 的设备后会关闭扫描并且发起连接, 但是当这个连接完成后在收到连接事件时会判断当前连接的设备是否已经大于最大可连接数, 如果没有到达最大连接数, 就会再次开启扫描, 这样就可以再次链接其他设备名为 `Nordic_Blinky` 的设备了。如下图所示。这个工作就在蓝牙处理函数 `ble_evt_handler` 发送连接事件 `BLE_GAP_EVT_CONNECTED` 进行处理, 因为上一个步骤中 `sd_ble_gap_connect` 函数启动连接的时候会触发该事件, 代码如下所示:

```

253  /*
254  static void ble_evt_handler(ble_evt_t const * p_ble_evt, void * p_context)
255  {
256      ret_code_t err_code;
257
258      // For readability.
259      ble_gap_evt_t const * p_gap_evt = &p_ble_evt->evt.gap_evt;
260
261      switch (p_ble_evt->header.evt_id)
262      {
263          // Upon connection, check which peripheral has connected, initiate DB
264          // discovery, update LEDs status and resume scanning if necessary.
265          case BLE_GAP_EVT_CONNECTED:
266          {
267              NRF_LOG_INFO("Connection 0x%x established, starting DB discovery.",
268                           p_gap_evt->conn_handle);
269
270              APP_ERROR_CHECK_BOOL(p_gap_evt->conn_handle < NRF_SDH_BLE_CENTRAL_LINK_COUNT); //连接句柄如果没有分配完
271
272              err_code = ble_lbs_c_handles_assign(&m_lbs_c[p_gap_evt->conn_handle],
273                                                  p_gap_evt->conn_handle,
274                                                  NULL); //分频连接句柄
275
276              APP_ERROR_CHECK(err_code);
277
278              err_code = ble_db_discovery_start(&m_db_disc[p_gap_evt->conn_handle],
279                                                p_gap_evt->conn_handle); //开始发现对应句柄的蓝牙服务
280
281              if (err_code != NRF_ERROR_BUSY)
282              {
283                  APP_ERROR_CHECK(err_code);
284              }
285
286              // Update LEDs status, and check if we should be looking for more
287              // peripherals to connect to.
288              bsp_board_led_on(CENTRAL_CONNECTED_LED); //更新LED灯
289              if (ble_conn_state_central_conn_count() == NRF_SDH_BLE_CENTRAL_LINK_COUNT) //如果达到最大的连接数量
290              {
291                  bsp_board_led_off(CENTRAL_SCANNING_LED); //关掉扫描LED
292              }
293              else
294              {
295                  // Resume scanning.
296                  bsp_board_led_on(CENTRAL_SCANNING_LED); //否则继续扫描
297                  scan_start();
298              }
299              break; // BLE_GAP_EVT_CONNECTED
300          }
301      }
302  }

```

一旦启动主机扫描,如果发现了从机广播则产生 BLE_GAP_EVT_ADV_REPORT 事件继续连接指定广播名称的从设备。如果没有发现,则会继续扫描。

那么上面分析就回答了前面提到的两个问题:

1: 主机设备如何判断哪个从机是我们需要的?

答: 通过设备的名称判断哪个从机是需要连接的

2: 如何实现多个设备的连接?

答: 通过蓝牙事件的回调函数, 连接事件的时候判断连接个数, 没有达到个数的继续开启主机扫描。

其他关于设备的设备连接方面的详细参考蓝牙串口的讲解。

1.3 主机和从机通信通道搭建:

1 拖 8 的实例实现的现象是 1 个主机可以连接 8 个从机, 从机按键然后发数据给主机, 主机对应亮/灭 led 灯。那么从机服务就是具有了 notify 功能的特征值, 通过通知主机, 主机接收数据后改变 led 等的状态。同时, 主机可以通过写的方式, 通过把主机按键状态变化值写入到从机, 来控制从机的 LED 灯变化。

那么主机和从机的通信通道就应该是空中属性的写和通知类型。下面首先讨论下从机通知的服务类型, 在前面一讲的主机蓝牙串口里也有, 但是这里的区别就是你要知

是哪个从机发过来的通知。下面我们就具体的来讨论一下这个问题。

首先要考虑的是从机发过来主机通知，那么主机要使能从机的 **notify** 功能，从机的通知数据才能被主机所介绍，关于如果设置从机通知在前面的从机例子《BLE 按键通知》里就详细的讲述了，这里不再累述。

1.3.1 主机设备通知使能及数据接收

本节主要谈下主机如何使能从机。在主机和从机发起连接后，首先主机要发现从机的服务，才能谈后面的数据传输。而整个发现过程需要通过 UUID 进行的。这个任务在主机代码中交给了数据发现函数来实现的，整个实现过程请参考《主机串口详解》教程，这里我们主要说明下 1 带 8 点灯的例子中主机的 UUID 注册与声明。首先在主函数中需要对主服务进行声明，调用函数 `lbs_c_init()` 函数对主机服务进行初始化，代码如下：

```

384 static void lbs_c_init(void)//不同链路主服务初始化
385 {
386     ret_code_t      err_code;
387     ble_lbs_c_init_t lbs_c_init_obj;
388
389     lbs_c_init_obj.evt_handler = lbs_c_evt_handler;
390
391     for (uint32_t i = 0; i < NRF_SDH_BLE_CENTRAL_LINK_COUNT; i++)
392     {
393         err_code = ble_lbs_c_init(&m_lbs_c[i], &lbs_c_init_obj);
394         APP_ERROR_CHECK(err_code);
395     }
396 }
397

```

通过 `ble_lbs_c_init` 函数实现主机初始化，这个函数是一个数据结构体类型，通过 `i` 的不同值，来分别代表不同的从机链路。这个函数被称为客户端处理事件初始化函数。该首先设置一个基础 UUID，这个基础 UUID 通过协议栈函数 `sd_ble_uuid_vs_add` 赋值给主设备。然后通过调用注册函数 `ble_db_discovery_evt_register` 通过 UUID 查处对应的服务。如果基础 UUID 相同，则会启动里面的回调函数 `db_discovery_evt_handler` 进行操作，服务发现过程我们就不展开，大家参考《主机串口详解》。代码如下：

```

244 uint32_t ble_lbs_c_init(ble_lbs_c_t * p_ble_lbs_c, ble_lbs_c_init_t * p_ble_lbs_c_init)
245 {
246     uint32_t      err_code;
247     ble_uuid_t     lbs_uuid;
248     ble_uuid128_t lbs_base_uuid = {LBS_UUID_BASE};
249
250     VERIFY_PARAM_NOT_NULL(p_ble_lbs_c);
251     VERIFY_PARAM_NOT_NULL(p_ble_lbs_c_init);
252     VERIFY_PARAM_NOT_NULL(p_ble_lbs_c_init->evt_handler);
253
254     p_ble_lbs_c->peer_lbs_db.button_cccd_handle = BLE_GATT_HANDLE_INVALID;
255     p_ble_lbs_c->peer_lbs_db.button_handle     = BLE_GATT_HANDLE_INVALID;
256     p_ble_lbs_c->peer_lbs_db.led_handle        = BLE_GATT_HANDLE_INVALID;
257     p_ble_lbs_c->conn_handle                   = BLE_CONN_HANDLE_INVALID;
258     p_ble_lbs_c->evt_handler                   = p_ble_lbs_c_init->evt_handler;
259
260     err_code = sd_ble_uuid_vs_add(&lbs_base_uuid, &p_ble_lbs_c->uuid_type);
261     if (err_code != NRF_SUCCESS)
262     {
263         return err_code;
264     }
265     VERIFY_SUCCESS(err_code);
266
267     lbs_uuid.type = p_ble_lbs_c->uuid_type;
268     lbs_uuid.uuid = LBS_UUID_SERVICE;
269
270     return ble_db_discovery_evt_register(&lbs_uuid);
271 }
272

```

添加基础UUID

注册主服务UUID

我们直接来到服务发现后处理部分,如果是我们完成了完整的服务发现过程,那么就会触发回调函数 `lbs_c_evt_handler`,该回调函数工作非常简单,首先是当发现主服务完成后 `BLE_LBS_C_EVT_DISCOVERY_COMPLETE` 事件后,启动主机使能从机的通知。代码如下:

```

177 static void lbs_c_evt_handler(ble_lbs_c_t * p_lbs_c, ble_lbs_c_evt_t * p_lbs_c_evt)
178 {
179     switch (p_lbs_c_evt->evt_type)
180     {
181     case BLE_LBS_C_EVT_DISCOVERY_COMPLETE://主服务发现完成
182     {
183         ret_code_t err_code;
184
185         NRF_LOG_INFO("LED Button service discovered on conn_handle 0x%x",
186                     p_lbs_c_evt->conn_handle);
187
188         err_code = app_button_enable();//主机按键初始化
189         APP_ERROR_CHECK(err_code);
190
191         // LED Button service discovered. Enable notification of Button.
192         err_code = ble_lbs_c_button_notif_enable(p_lbs_c);//主服务按键通知使能从机
193         APP_ERROR_CHECK(err_code);
194     } break; // BLE_LBS_C_EVT_DISCOVERY_COMPLETE
195
196     case BLE_LBS_C_EVT_BUTTON_NOTIFICATION://按键通知只能后
197     {
198         NRF_LOG_INFO("Link 0x%x, Button state changed on peer to 0x%x",
199                     p_lbs_c_evt->conn_handle,
200                     p_lbs_c_evt->params.button.button_state);
201
202         if (p_lbs_c_evt->params.button.button_state)
203         {
204             bsp_board_led_on(LED_BUTTON_LED);//通过按键状态来改变主机LED灯状态
205         }
206         else
207         {
208             bsp_board_led_off(LED_BUTTON_LED);
209         }
210     } break; // BLE_LBS_C_EVT_BUTTON_NOTIFICATION
211
212     default:
213         // No implementation needed.
214         break;
215     }
216 }

```

进入按键通知使能服务 `ble_lbs_c_button_notif_enable` 函数内部,使能通知实际上就是 CCCD 的写操作,实际上就是通过协议栈函数 `sd_ble_gattc_write` 来实现 CCCD 的写入,代码如下:

```

336 uint32_t ble_lbs_c_button_notif_enable(ble_lbs_c_t * p_ble_lbs_c)
337 {
338     VERIFY_PARAM_NOT_NULL(p_ble_lbs_c);
339
340     if (p_ble_lbs_c->conn_handle == BLE_CONN_HANDLE_INVALID)
341     {
342         return NRF_ERROR_INVALID_STATE;
343     }
344
345     return cccd_configure(p_ble_lbs_c->conn_handle,
346                         p_ble_lbs_c->peer_lbs_db.button_cccd_handle,
347                         true);
348 }
349

```

通过函数内部展开,发现函数其最终的目标是什么。其实就是通过主机写函数,写入 `cccd` 的值来实现 CCCD 使能:

```

310 static uint32_t cccd_configure(uint16_t conn_handle, uint16_t handle_cccd, bool enable)
311 {
312     NRF_LOG_DEBUG("Configuring CCCD. CCCD Handle = %d, Connection Handle = %d",
313         handle_cccd, conn_handle);
314
315     tx_message_t * p_msg;
316     uint16_t cccd_val = enable ? BLE_GATT_HVX_NOTIFICATION : 0; //是否是写CCCD
317
318     p_msg = &m_tx_buffer[m_tx_insert_index++];
319     m_tx_insert_index &= TX_BUFFER_MASK;
320
321     p_msg->req.write_req.gattc_params.handle = handle_cccd;
322     p_msg->req.write_req.gattc_params.len = WRITE_MESSAGE_LENGTH;
323     p_msg->req.write_req.gattc_params.p_value = p_msg->req.write_req.gattc_value; //要写的值
324     p_msg->req.write_req.gattc_params.offset = 0;
325     p_msg->req.write_req.gattc_params.write_op = BLE_GATT_OP_WRITE_REQ;
326     p_msg->req.write_req.gattc_value[0] = LSB_16(cccd_val);
327     p_msg->req.write_req.gattc_value[1] = MSB_16(cccd_val);
328     p_msg->conn_handle = conn_handle;
329     p_msg->type = WRITE_REQ;
330
331     tx_buffer_process();
332     return NRF_SUCCESS;
333 }

```

```

93 static void tx_buffer_process(void)
94 {
95     if (m_tx_index != m_tx_insert_index)
96     {
97         uint32_t err_code;
98
99         if (m_tx_buffer[m_tx_index].type == READ_REQ) //读应答
100         {
101             err_code = sd_ble_gattc_read(m_tx_buffer[m_tx_index].conn_handle,
102                 m_tx_buffer[m_tx_index].req.read_handle,
103                 0); //GATT读状态
104         }
105         else
106         {
107             err_code = sd_ble_gattc_write(m_tx_buffer[m_tx_index].conn_handle,
108                 &m_tx_buffer[m_tx_index].req.write_req.gattc_params); //GATT写状态
109         }
110         if (err_code == NRF_SUCCESS)
111         {
112             NRF_LOG_DEBUG("SD Read/Write API returns Success..");
113             m_tx_index++;
114             m_tx_index &= TX_BUFFER_MASK;
115         }
116         else
117         {
118             NRF_LOG_DEBUG("SD Read/Write API returns error. This message sending will be "
119                 "attempted again..");
120         }
121     }
122 }

```

当通知使能后, 主机就可以接受从机发来的通知信息, 本例有多个从机, 那么实现的时候, 这个任务交给派发函数 `ble_lbs_c_on_ble_evt` 来处理, 改派发函数在主函数 `main.c` 最开头进行了声明: 代码如下:

```

main.c*  ble_lbs_c.c  sdk_config.h  ble_lbs_c.h  pca10040.h  app_but
88 #define SLAVE_LATENCY 0
89 #define SUPERVISION_TIMEOUT MSEC_TO_UNITS(4000, UNIT_10_MS)
90
91
92 NRF_BLE_GATT_DEF(m_gatt);
93 BLE_LBS_C_ARRAY_DEF(m_lbs_c, NRF_SDH_BLE_CENTRAL_LINK_COUNT);
94 BLE_DB_DISCOVERY_ARRAY_DEF(m_db_disc, NRF_SDH_BLE_CENTRAL_LINK_COUNT);
95

```



```

88  */
89  #define BLE_LBS_C_ARRAY_DEF(_name, _cnt)
90  static ble_lbs_c_t _name[_cnt];
91  NRF_SDH_BLE_OBSERVERS(_name ## _obs,
92                          BLE_LBS_C_BLE_OBSERVER_PRIO,
93                          ble_lbs_c_on_ble_evt, &_name, _cnt)
94

```

派发函数 `ble_lbs_c_on_ble_evt` 可以称为主机点灯事件派发事件，该事件下处理主机点灯的空中属性操作。本例使用的通知和写都是使用该事件进行处理的，代码如下图所示。我们首先看这里面的接收从机通知 `BLE_GATTC_EVT_HVX` 事件后如何处理的。一旦从机使用 `hvx` 通知函数上传数据给主机，主机就会接收到数据，此时就会触发 `BLE_GATTC_EVT_HVX` 事件，该事件下执行 `on_hvx` 函数：

```

273 void ble_lbs_c_on_ble_evt(ble_evt_t const * p_ble_evt, void * p_context)
274 {
275     if ((p_context == NULL) || (p_ble_evt == NULL))
276     {
277         return;
278     }
279
280     ble_lbs_c_t * p_ble_lbs_c = (ble_lbs_c_t *)p_context;
281
282     switch (p_ble_evt->header.evt_id)
283     {
284     case BLE_GATTC_EVT_HVX:
285         on_hvx(p_ble_lbs_c, p_ble_evt); //接收从机通知
286         break;
287
288     case BLE_GATTC_EVT_WRITE_RSP:
289         on_write_rsp(p_ble_lbs_c, p_ble_evt); //写从机
290         break;
291
292     case BLE_GAP_EVT_DISCONNECTED:
293         on_disconnected(p_ble_lbs_c, p_ble_evt); //断开连接
294         break;
295
296     default:
297         break;
298     }
299 }

```

`on_hvx` 函数最终处理接收到的从机通知数据，内部如下图所示，触发数据类型为按键通知 `BLE_LBS_C_EVT_BUTTON_NOTIFICATION`，操作句柄为 `p_ble_lbs_c->conn_handle`，数据存 `params.button.button_state` 中：

```

152 static void on_hvx(ble_lbs_c_t * p_ble_lbs_c, ble_evt_t const * p_ble_evt)
153 {
154     // Check if the event is on the link for this instance
155     if (p_ble_lbs_c->conn_handle != p_ble_evt->evt.gattc_evt.conn_handle)
156     {
157         return;
158     }
159     // Check if this is a Button notification.
160     if (p_ble_evt->evt.gattc_evt.params.hvx.handle == p_ble_lbs_c->peer_lbs_db.button_handle)
161     {
162         if (p_ble_evt->evt.gattc_evt.params.hvx.len == 1)
163         {
164             ble_lbs_c_evt_t ble_lbs_c_evt;
165
166             ble_lbs_c_evt.evt_type = BLE_LBS_C_EVT_BUTTON_NOTIFICATION;
167             ble_lbs_c_evt.conn_handle = p_ble_lbs_c->conn_handle;
168             ble_lbs_c_evt.params.button.button_state = p_ble_evt->evt.gattc_evt.params.hvx.data[0];
169             p_ble_lbs_c->evt_handler(p_ble_lbs_c, &ble_lbs_c_evt);
170         }
171     }
172 }

```

一旦触发了蓝牙事件 BLE_LBS_C_EVT_BUTTON_NOTIFICATION, 在蓝牙事件回调函数 lbs_c_evt_handler 中, 就要执行如下图所示的操作, 也就是根据上传的数据来改变 LED 灯的亮灭:

```

177 static void lbs_c_evt_handler(ble_lbs_c_t * p_lbs_c, ble_lbs_c_evt_t * p_lbs_c_evt)
178 {
179     switch (p_lbs_c_evt->evt_type)
180     {
181         case BLE_LBS_C_EVT_DISCOVERY_COMPLETE://主服务发现完成
182         {
183             ret_code_t err_code;
184
185             NRF_LOG_INFO("LED Button service discovered on conn_handle 0x%x",
186                         p_lbs_c_evt->conn_handle);
187
188             err_code = app_button_enable();//主机按键初始化
189             APP_ERROR_CHECK(err_code);
190
191             // LED Button service discovered. Enable notification of Button.
192             err_code = ble_lbs_c_button_notif_enable(p_lbs_c);//主服务按键通知使能从机
193             APP_ERROR_CHECK(err_code);
194         } break; // BLE_LBS_C_EVT_DISCOVERY_COMPLETE
195
196         case BLE_LBS_C_EVT_BUTTON_NOTIFICATION://按键通知使能后
197         {
198             NRF_LOG_INFO("Link 0x%x, Button state changed on peer to 0x%x",
199                         p_lbs_c_evt->conn_handle,
200                         p_lbs_c_evt->params.button.button_state);
201
202             if (p_lbs_c_evt->params.button.button_state)
203             {
204                 bsp_board_led_on(LED_BUTTON_LED); //通过按键状态来改变主机LED灯状态
205             }
206             else
207             {
208                 bsp_board_led_off(LED_BUTTON_LED);
209             }
210         } break; // BLE_LBS_C_EVT_BUTTON_NOTIFICATION
211
212         default:
213             // No implementation needed.
214             break;
215     }
216 }
217

```

那么上面的工程就实现了从机数据通知给主机, 主机通过该数据来控制 LED 等的这个过程了。但是还有一个问题, 如果区分是哪个从机发来的数据了? 之前分配句柄的时候就是通过 i 不同来分配不同的句柄, 因此, 不同的 p_lbs_c_evt->conn_handle 决定了不同的从机链路, 我们可以通过句柄来区分到底是哪个从机的数据。

现在我们根据这个原理, 对源码进行小的修改。没有修改之前, 不管哪个从机按下按键 key1, 主机都只有 LED3 会点亮。现在我们修改成区分第一个从机链路和其他链路, 第 1 个链路从机按键 KEY1 按下后, 主机上 LED3 点亮, 其他链接的从机上按键 KEY1 按下后, 主机上 LED4 点亮。代码如下, 通过判断 p_lbs_c_evt->conn_handle 的值, 来决定你点亮哪个 LED 灯:

```

177 static void lbs_c_evt_handler(ble_lbs_c_t * p_lbs_c, ble_lbs_c_evt_t * p_lbs_c_evt)
178 {
179     switch (p_lbs_c_evt->evt_type)
180     {
181         case BLE_LBS_C_EVT_DISCOVERY_COMPLETE://主服务发现完成
182         {
183             ret_code_t err_code;
184
185             NRF_LOG_INFO("LED Button service discovered on conn_handle 0x%x",
186                         p_lbs_c_evt->conn_handle);
187
188             err_code = app_button_enable();//主机按键初始化
189             APP_ERROR_CHECK(err_code);
190
191             // LED Button service discovered. Enable notification of Button.
192             err_code = ble_lbs_c_button_notif_enable(p_lbs_c);//主服务按键通知使能从机
193             APP_ERROR_CHECK(err_code);
194         } break; // BLE_LBS_C_EVT_DISCOVERY_COMPLETE
195
196         case BLE_LBS_C_EVT_BUTTON_NOTIFICATION://按键通知使能后
197         {
198             NRF_LOG_INFO("Link 0x%x, Button state changed on peer to 0x%x",
199                         p_lbs_c_evt->conn_handle,
200                         p_lbs_c_evt->params.button.button_state);
201
202             if(p_lbs_c_evt->conn_handle==0x01)
203             {
204                 if (p_lbs_c_evt->params.button.button_state)
205                 {
206                     bsp_board_led_on(LED_BUTTON_LED); //通过按键状态来改变主机LED灯状态
207                 }
208                 else
209                 {
210                     bsp_board_led_off(LED_BUTTON_LED);
211                 }
212             }
213             else
214             {
215                 if (p_lbs_c_evt->params.button.button_state)
216                 {
217                     bsp_board_led_on(BSP_BOARD_LED_3); //通过按键状态来改变主机LED灯状态
218                 }
219                 else
220                 {
221                     bsp_board_led_off(BSP_BOARD_LED_3);
222                 }
223             }
224         }
225     }
226 }

```

链路 按键状态

这样就解决了从机---》主机这个方向上的数据传输问题，并且能够区分不同的从机数据。

1.3.1 主机设备数据写从机

再来讨论下主机---》从机的数据方向如何进行数据通信的，主机往从机写数据构成了该通道的数据流。本来实现一个主机按键按下，控制从机 LED 灯的演示，首先看主机按键设置，代码如下：

```

526 static void buttons_init(void)
527 {
528     ret_code_t err_code;
529
530     // The array must be static because a pointer to it will be saved in the button handler module.
531     static app_button_cfg_t buttons[] =
532     {
533         {LEDBUTTON_BUTTON, false, BUTTON_PULL, button_event_handler}, //按键中断
534     };
535
536     err_code = app_button_init(buttons, ARRAY_SIZE(buttons), BUTTON_DETECTION_DELAY);
537     APP_ERROR_CHECK(err_code);
538 }

```

设置了一个名称为 LEDBUTTON_BUTTON 按键在 NRF_GPIO_PIN_PULLUP 转态下，触发按键中断 button_event_handler，该按键中断函数如下所示：


```

496 static void button_event_handler(uint8_t pin_no, uint8_t button_action)
497 {
498     ret_code_t err_code;
499
500     switch (pin_no)
501     {
502     case LEDBUTTON_BUTTON:
503         err_code = led_status_send_to_all(button_action); //主机按下后, 点亮所有从机
504         {
505             NRF_LOG_INFO("LBS write LED state %d", button_action);
506         }
507         break;
508
509     default:
510         APP_ERROR_HANDLER(pin_no);
511         break;
512     }
513 }
514

```

当产生了 LEDBUTTON_BUTTON 按键按下事件后, 会调用主机写从机的 LED 灯函数, 该函数如下所示:

```

456 static ret_code_t led_status_send_to_all(uint8_t button_action) //按键状态发送
457 {
458     ret_code_t err_code;
459
460     for (uint32_t i = 0; i < NRF_SDH_BLE_CENTRAL_LINK_COUNT; i++)
461     {
462         err_code = ble_lbs_led_status_send(&m_lbs_c[i], button_action); //按键状态发送给主机, 主机写从机
463         if (err_code != NRF_SUCCESS &&
464             err_code != BLE_ERROR_INVALID_CONN_HANDLE &&
465             err_code != NRF_ERROR_INVALID_STATE)
466         {
467             return err_code;
468         }
469     }
470     return NRF_SUCCESS;
471 }

```

函数中调用了 ble_lbs_led_status_send 写状态函数, 把主机按键状态作为数据写入到从机, 通过 i 的值控制不同的从机。tx_buffer_process 这个函数就是前面通知使能中使用的主机 cccd 写函数, 这里同样可以作为写数据使用。

```

351 uint32_t ble_lbs_led_status_send(ble_lbs_c_t * p_ble_lbs_c, uint8_t status)
352 {
353     VERIFY_PARAM_NOT_NULL(p_ble_lbs_c);
354
355     if (p_ble_lbs_c->conn_handle == BLE_CONN_HANDLE_INVALID)
356     {
357         return NRF_ERROR_INVALID_STATE;
358     }
359
360     NRF_LOG_DEBUG("writing LED status 0x%x", status);
361
362     tx_message_t * p_msg;
363
364     p_msg = &m_tx_buffer[m_tx_insert_index++];
365     m_tx_insert_index &= TX_BUFFER_MASK;
366
367     p_msg->req.write_req.gattc_params.handle = p_ble_lbs_c->peer_lbs_db.led_handle;
368     p_msg->req.write_req.gattc_params.len = sizeof(status);
369     p_msg->req.write_req.gattc_params.p_value = p_msg->req.write_req.gattc_value;
370     p_msg->req.write_req.gattc_params.offset = 0;
371     p_msg->req.write_req.gattc_params.write_op = BLE_GATT_OP_WRITE_CMD;
372     p_msg->req.write_req.gattc_value[0] = status; //写的数据
373     p_msg->conn_handle = p_ble_lbs_c->conn_handle;
374     p_msg->type = WRITE_REQ;
375
376     tx_buffer_process(); //数据写
377     return NRF_SUCCESS;
378 }
379

```

从机接收到主机开发板写入的状态数据, 就控制 LED 灯翻转, 这个过程和手机作为主机控制从机的过程一样, 请参考从机教程《蓝牙 led 灯读写》。

这里同样，我们可以来进行一个小的修改，来区分是写哪个从机。原程序只写了一个按键中断，也就是按下这个中断后，所有的从机都会亮灯。我们下面写入两个按键中断，一个控制从机 0，一个控制从机 1，代码如下所示：

1. 增加一个按键中断所使用的按键，触发类型和中断函数用同一个：

```

526 static void buttons_init(void)
527 {
528     ret_code_t err_code;
529
530     // The array must be static because a pointer to it will be saved in the button handler module.
531     static app_button_cfg_t buttons[] =
532     {
533         {LEDBUTTON_BUTTON, false, BUTTON_PULL, button_event_handler}, //按键中断
534         {BSP_BUTTON_1, false, BUTTON_PULL, button_event_handler} //按键中断
535     };
536
537     err_code = app_button_init(buttons, ARRAY_SIZE(buttons), BUTTON_DETECTION_DELAY);
538     APP_ERROR_CHECK(err_code);
539 }

```

2. 中断中，不同按键，触发不同的数据写通道函数，如下图所示

```

496 static void button_event_handler(uint8_t pin_no, uint8_t button_action)
497 {
498     ret_code_t err_code;
499
500     switch (pin_no)
501     {
502         case LEDBUTTON_BUTTON:
503             err_code = led_status_send_to_all(button_action); //主机按下后，点亮所有从机
504             {
505                 NRF_LOG_INFO("LBS write LED state %d", button_action);
506             }
507             break;
508
509             //增加一个一对一的控制
510             case BSP_BUTTON_1:
511                 err_code = led_status_send_to_two(button_action); //主机按下后，点亮所有从机
512                 {
513                     NRF_LOG_INFO("LBS write LED2 state %d", button_action);
514                 }
515                 break;
516
517             default:
518                 APP_ERROR_HANDLER(pin_no);
519                 break;
520     }
521 }
522

```

3. 增加一写从机处理函数，通过 i 的值，分别定义 0 通道从机和 1 通道从机，代码如下所示：


```
456 static ret_code_t led_status_send_to_all(uint8_t button_action)//按键状态发送
457 {
458     ret_code_t err_code;
459     for (uint32_t i = 0; i < NRF_SDH_BLE_CENTRAL_LINK_COUNT; i++)
460     {
461         err_code = ble_lbs_led_status_send(&m_lbs_c[i], button_action);//按键状态发送给主机，主机写从机
462         if (err_code != NRF_SUCCESS &&
463             err_code != BLE_ERROR_INVALID_CONN_HANDLE &&
464             err_code != NRF_ERROR_INVALID_STATE)
465         {
466             return err_code;
467         }
468     }
469     return NRF_SUCCESS;
470 }
471
472
473
474 static ret_code_t led_status_send_to_two(uint8_t button_action)//按键状态发送
475 {
476     ret_code_t err_code;
477     // for (uint32_t i = 0; i < NRF_SDH_BLE_CENTRAL_LINK_COUNT; i++)
478     // {
479         err_code = ble_lbs_led_status_send(&m_lbs_c[1], button_action);//按键状态发送给主机，主机写从机
480         if (err_code != NRF_SUCCESS &&
481             err_code != BLE_ERROR_INVALID_CONN_HANDLE &&
482             err_code != NRF_ERROR_INVALID_STATE)
483         {
484             return err_code;
485         }
486     // }
487     return NRF_SUCCESS;
488 }
489 }
```

这时候，按下按键 1，从机 0 的 LED3 会被点亮。按下按键 2，从机 1 的 LED3 会别点亮。达到区分写入不同从机的过程。

讲了这么多，大家现在应该对整个主机 1 拖 8 组网的过程有了一个清楚的认识。读者如果理解了整个组网的过程，我们的讲座的目的也达到了。

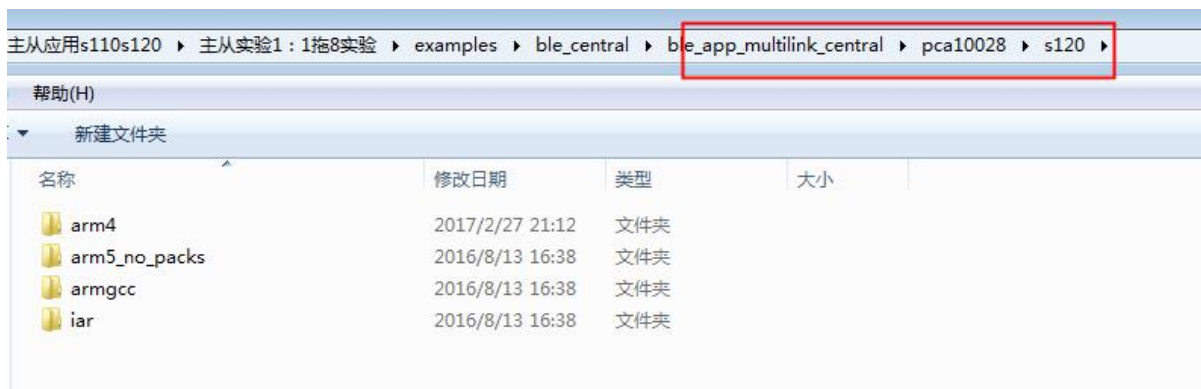
2 应用与调试

本实验使用多个开发板，开发板上有 4 个 LED 灯，我们用 4 个开发板为从机，一个开发板为主机。

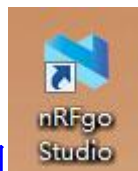
2.1 软件准备：

2.1.1 主机下载

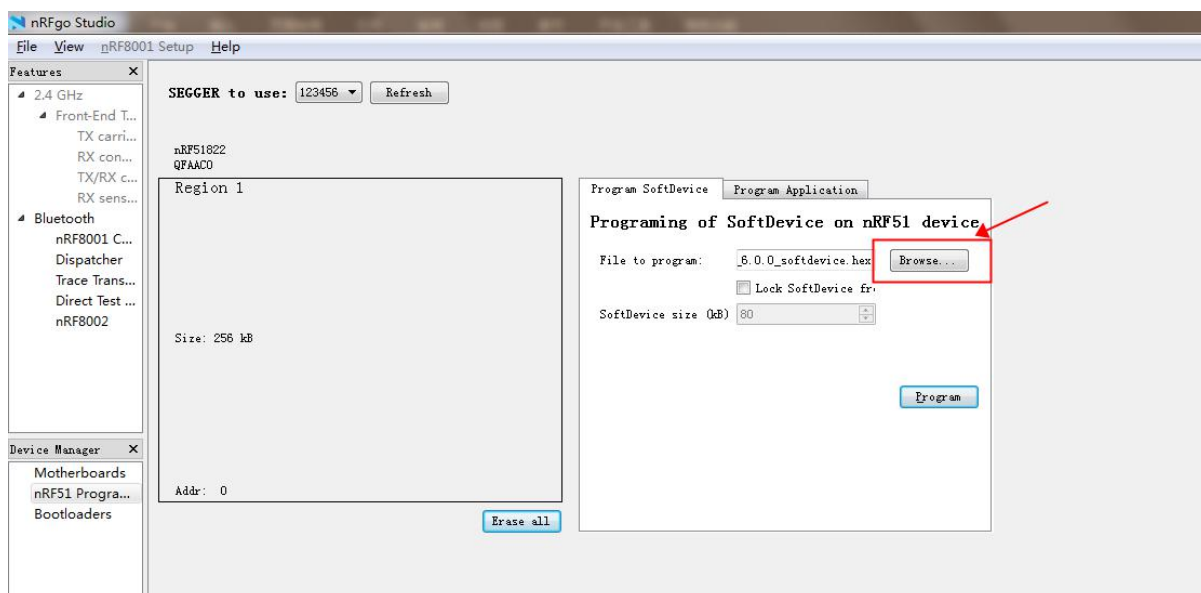
在代码文件中，打开工程主机程序，如下所示：



把上面提供的 KEIL 工程点击编译, 同时设置仿真器为 JLINK 仿真器, 详细设置仿真器过程请参考《青云 nRF51822 软件篇: 开发板环境与工程项目的建立》, 这个主机使用协议栈 S120.



1. 首先采用 下载协议栈, 打开 nRFGo Studio 软件, 同时把开发板 usb 连接电脑 PC 机, 如下图所示, 点击 program softdevice, 点击 browse 选择协议栈 S120 下载到其中一个开发板内, 做为主机:



S120 协议栈路径如下:

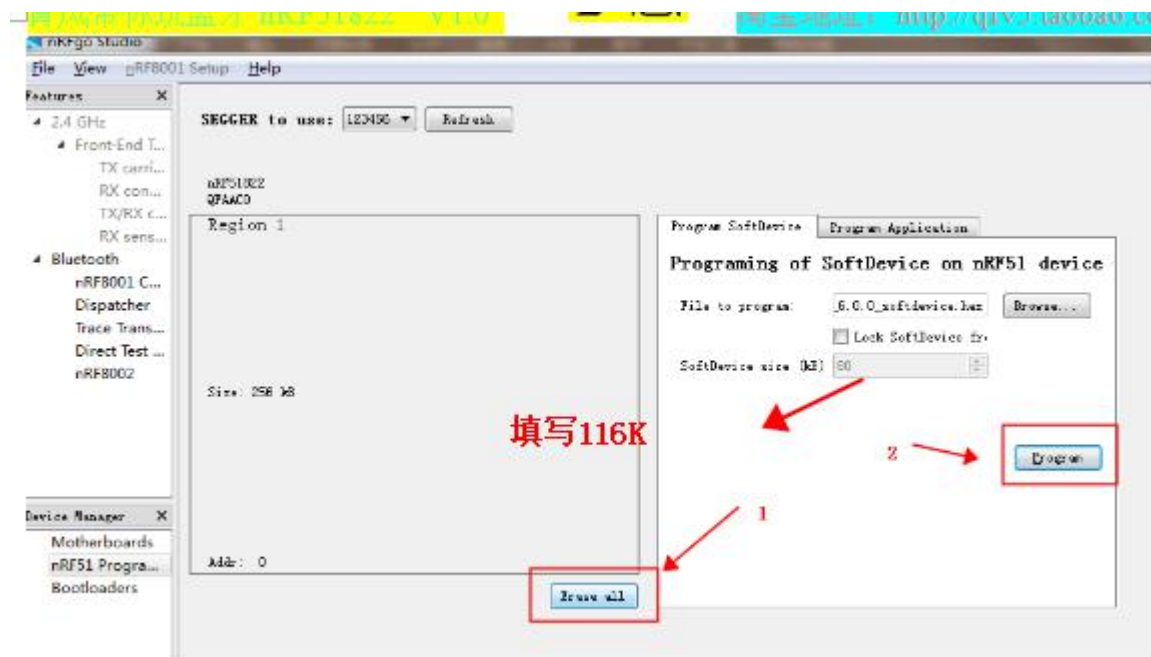
3) > 2016年开发产品 > 青云蓝牙DK开发板v2 > 5.青云测试代码 > 第5部分: 高级BLE主从应用s110s120 >

具(I) 帮助(H)

到库中 > 共享 > 新建文件夹

名称	修改日期	类型	大小
s120	2016/3/10 19:28	文件夹	
主从实验1: 1拖8实验	2016/3/10 20:16	文件夹	
主从实验2: 主从心电	2016/3/10 20:17	文件夹	
主从实验3: 主从串口	2016/3/10 20:18	文件夹	
主从实验3: 主从串口.rar	2016/3/15 10:47	360压缩 RAR 文件	7,574 KB

然后分两步, 首先整片擦除, 后下载协议栈, 如下图所示:



下载成功后会提示如下:

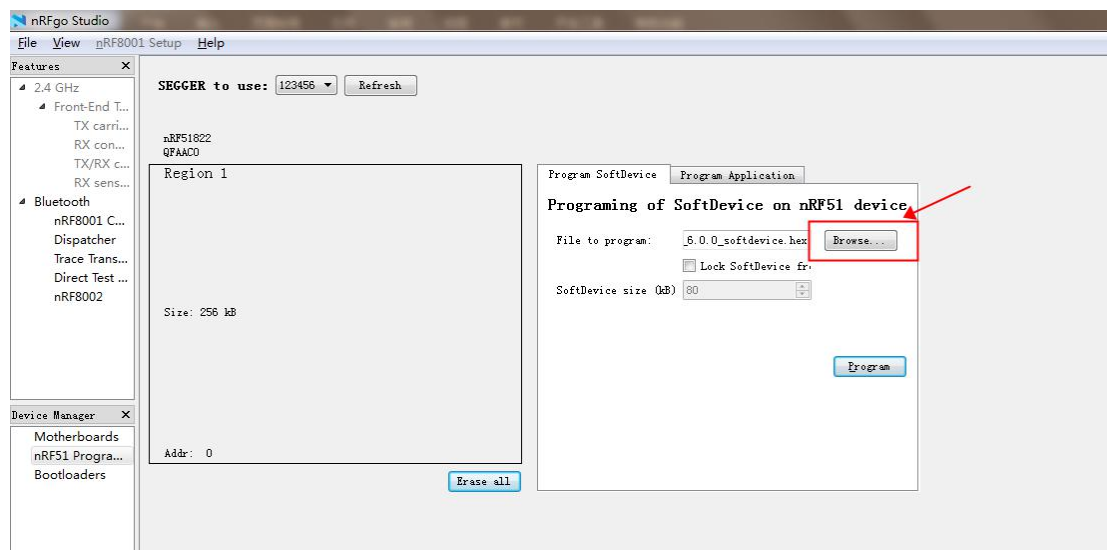
```
Log
(c) Nordic Semiconductor ASA 2008-2013
Erase succeeded
Programmed successfully softdevice F:/baiduyundownload/nRF Beacon/蓝牙协议栈/s110_nrf51822_6.0.0/s110_nrf51822_6.0.0_softdevice.hex
```

2. 协议栈下载完成后, 下载应用程序, 打开 MDK 的主机串口工程, 如本篇文章开头所示的工程路径, 打开后点击 MDK 的 load 按键:



2.1.2 从机下载:

打开 nRFgo Studio 软件, 同时把开发板 usb 连接电脑 PC 机, 如下图所示, 点击 program softdevice, 点击 browse 选择协议栈 S110 下载到其中另外四个开发板内, 做为从机:



S110 协议栈路径如下:

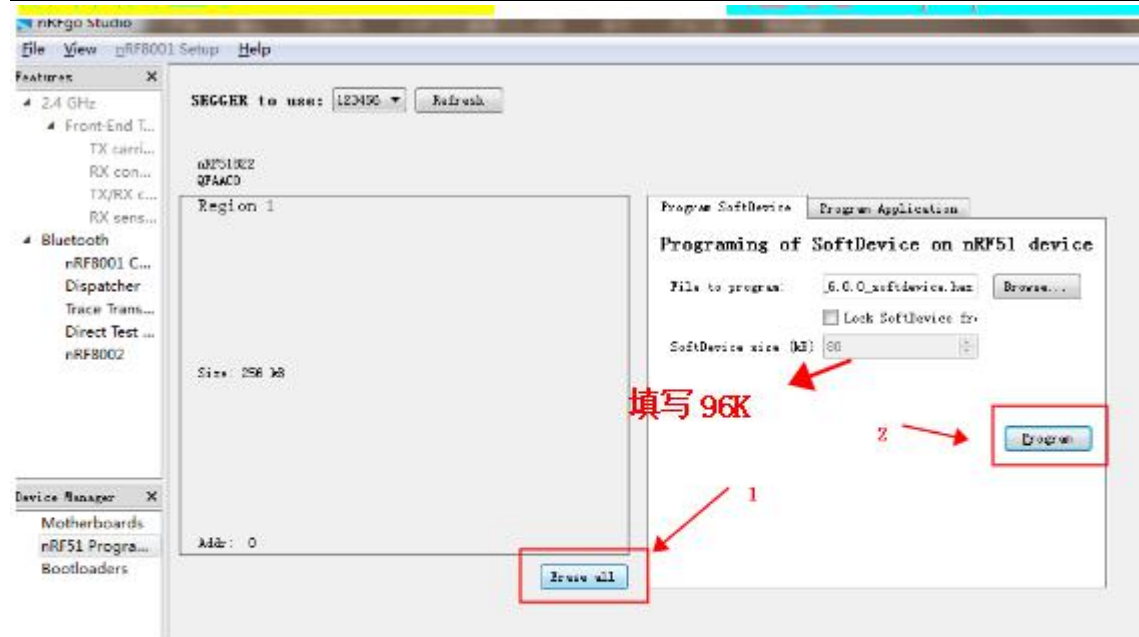
文件夹 (G:) > 2016年开发产品 > 青云蓝牙DK开发板v2 > 5.青云测试代码 > 第4部分: 高级BLE蓝牙实验从机 >

帮助(H)

新建文件夹

名称	修改日期	类型	大小
BLE实验1: 蓝牙样例	2016/1/8 20:45	文件夹	
BLE实验2: LED蓝牙任务读写	2016/3/8 15:16	文件夹	
BLE实验3: 按键通知	2016/3/14 21:05	文件夹	
BLE实验4: 蓝牙串口	2016/1/8 20:48	文件夹	
BLE实验5: 蓝牙遥控器	2016/3/16 21:29	文件夹	
BLE实验6: 蓝牙心电	2016/3/8 15:33	文件夹	
BLE实验7: FreeRTOS下运行蓝牙心电	2016/3/8 15:33	文件夹	
BLE实验8: RTX下运行蓝牙心电	2016/3/8 15:50	文件夹	
BLE实验9: 蓝牙温度传感	2016/1/8 21:30	文件夹	
BLE实验10: 蓝牙键盘	2016/3/8 15:33	文件夹	
BLE实验11: 蓝牙鼠标	2016/3/8 15:33	文件夹	
BLE实验12: 蓝牙血压计	2016/3/8 15:34	文件夹	
BLE实验13: 蓝牙ANC通知	2016/3/8 13:04	文件夹	
BLE实验14: 内部FLASH读写	2016/3/8 13:03	文件夹	
BLE实验15: 蓝牙接收数据FLASH存储	2016/4/20 20:54	文件夹	
BLE实验16: 蓝牙远程修改设备	2016/4/20 20:47	文件夹	
BLE实验17: 静态密码配对	2016/4/20 21:25	文件夹	
BLE实验18: 动态密码配对	2016/4/20 21:36	文件夹	
s110_nrf51_8.0.0	2016/3/8 13:02	文件夹	

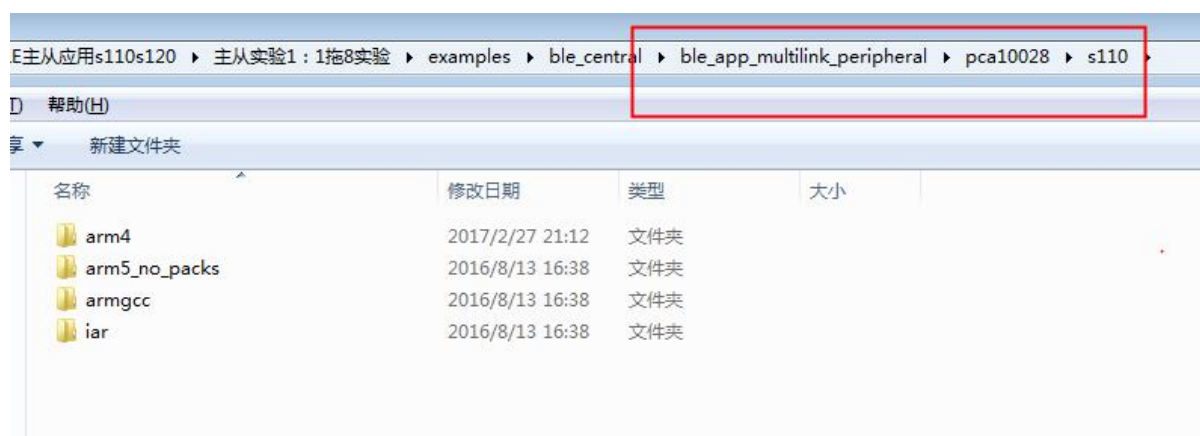
然后分两步, 首先整片擦除, 后下载协议栈, 如下图所示:



下载成功后会提示如下:



2. 协议栈下载完成后, 下载从机应用程序, 如下图所示的从机工程路径, 打开后点击 MDK 的 load 按钮:



下载完后作为从机。

2.2 实验现象:

开发分别通电,当主机连接上从机后,可以通过按下从机上的按键 0,来通知主机,主机对应的 LED 会点亮。