青风带你玩蓝牙 nRF52832 系列教程

----------作者：青风

出品论坛： www.qfv8.com 青风电子社区

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 蓝牙主机串口详解

前面的讲义里设计了很多蓝牙的基础知识，不过都是针对蓝牙从机设置的，那么今天这一讲将来给大家分析下蓝牙主机，相关的基础知识，这一节将详细的进行讨论。

并且通过蓝牙主机串口，来分析蓝牙主机的基本原理，以及编程方面的相关问题进行具体讨论。

1：nRF52832 蓝牙主机的主程序流程：

1.1 主机程序流程分析：

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)
{
    log_init(); //打印初始化
    timer_init();//软件定时器初始化
    uart_init();//串口初始化
    buttons_leds_init();//按键和 LED 初始化
    db_discovery_init();//蓝牙数据 Database 发现初始化
    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
    nus_c_init();//Nordic UART Service Client module 初始化
    //打印提示
    printf("BLE UART central example started.\r\n");
    NRF_LOG_INFO("BLE UART central example started.");

    scan_start();//开始扫描
```

```
    APPL_LOG("Scan started\r\n");

    for (;;)
    {
      idle_state_handle();//待机状态
    }
}
```

1. 首先是定时器初始化，按键和 LED 初始化，串口初始化，这三个硬件初始化设置和从机初始化一样，编写的时候没有任何变化。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。

2. ble_stack_init()协议栈初始化函数对比从机部分的《协议栈初始化详解》，基本结构没有变化，设备变化的地方有两个：

（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 文件中从机串口程序的设置如下：

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

主机串口程序设置如下：

```
11408
11409 // <o> NRF_SDH_BLE_PERIPHERAL_LINK_COUNT - Maximum number of peripheral links.
11410 #ifndef NRF_SDH_BLE_PERIPHERAL_LINK_COUNT
11411 #define NRF_SDH_BLE_PERIPHERAL_LINK_COUNT 0
11412 #endif
11413
11414 // <o> NRF_SDH_BLE_CENTRAL_LINK_COUNT - Maximum number of central links.
11415 #ifndef NRF_SDH_BLE_CENTRAL_LINK_COUNT
11416 #define NRF_SDH_BLE_CENTRAL_LINK_COUNT 1
11417 #endif
11418
```

（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：

```
393         case BLE_GAP_EVT_CONNECTED:
394             NRF_LOG_INFO("Connected to target");
395             err_code = ble_nus_c_handles_assign(&m_ble_nus_c, p_ble_evt->evt.gap_evt.conn_handle, NULL);//分配处理句柄
396             APP_ERROR_CHECK(err_code);
397 
398             err_code = bsp_indication_set(BSP_INDICATE_CONNECTED);//指示灯亮
399             APP_ERROR_CHECK(err_code);
400 
401             // 开始发现服务，NUS客户端等待发现结果
402             err_code = ble_db_discovery_start(&m_db_disc, p_ble_evt->evt.gap_evt.conn_handle);
403             APP_ERROR_CHECK(err_code);
404             break;
```

这种连接事件发生的时候，从机和主机在响应这一个蓝牙事件的时候处理的过程是不同的，主机需要发现服务。而从机只需要等待被连接。其他蓝牙事件下的处理也是有所区别的，这里可以使用官方默认配置下的事件响应，有需要变动的时候我们再来涉及如何修改。

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。

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，代码具体如下：

```
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. nus_c_init();//串口客户端初始化，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。

6.scan_start();//开始扫描

```
static void scan_start(void)//主机开始扫描
{
    ret_code_t ret;

    ret = sd_ble_gap_scan_start(&m_scan_params, &m_scan_buffer);
    APP_ERROR_CHECK(ret);

    ret = bsp_indication_set(BSP_INDICATE_SCANNING);
```

```
    APP_ERROR_CHECK(ret);
}
```

开始扫描函数里出现的两个函数： 第一个函数 sd_ble_gap_scan_start (&m_scan_params, &m_scan_buffer)是一个协议栈封装函数，也就是使用 &m_scan_params 定义的参数开始进行扫描：

```
static const ble_gap_scan_params_t m_scan_params =
  {
    .active       = SCAN_ACTIVE,//主动扫描
    .interval     = SCAN_INTERVAL,//扫描间隔
    .window       = SCAN_WINDOW,//扫描窗口
    .timeout      = SCAN_TIMEOUT//扫描超时
    .scan_phys          = BLE_GAP_PHY_1MBPS,//扫描的物理层
    .filter_policy      = BLE_GAP_SCAN_FP_ACCEPT_ALL,//扫描模式
  };
```

关于扫描参数的详细内容，我们后面会专门出一讲《主机扫描》进行讲解。

第二个函数 bsp_indication_set(BSP_INDICATE_SCANNING)设置扫描时候需要使用的板级设备，比如 LED 的闪烁表示在扫描。

上面的主机程序中主函数流程这几个重要函数说明，下面的讲解我们都会进一步具有涉及到的地方，大家一定要深入理解。

1.2 主机蓝牙串口过程分析：

我们要弄清楚两个问题：

1：主机设备和从机设备如何连接？

2：连接后主机的数据流的走向？

1.2.1 主机设备和从机设备连接分析

主机和从机设备的连接过程实际上是交个了两个派发函数来处理。这两个派发函数在 main 主函数最开头进行了声明，分别是：

```
ble_gatt_db.h | main.c* | ble_nus_c.c* | ble_db_discovery.h | ble_db_discovery.c

83  #define ECHOBACK_BLE_UART_DATA  1                                   /**< Echo the UART data that is received over t
84
85
86  BLE_NUS_C_DEF(m_ble_nus_c);                                         /**< BLE NUS service client instance. */
87  NRF_BLE_GATT_DEF(m_gatt);                                           /**< GATT module instance. */
88  BLE_DB_DISCOVERY_DEF(m_db_disc);                                    /**< DB discovery module instance. */
89
```

串口主机派发回调函数

数据发现处理回调函数

BLE_NUS_C_DEF(m_ble_nus_c); //蓝牙主机处理派发函数

BLE_DB_DISCOVERY_DEF(m_db_disc); //蓝牙数据发现事件派发函数

下面针对整发现与连接过程进行描述：

首先来分析下前面谈的协议栈初始化下，观察函数下的 ble_evt_handler 蓝牙事件处理派发函数：

```
467 /**@brief Function for initializing the BLE stack.
468  *
469  * @details Initializes the SoftDevice and the BLE event interrupt.
470  */
471 static void ble_stack_init(void)
472 {
473     ret_code_t err_code;
474
475     err_code = nrf_sdh_enable_request();
476     APP_ERROR_CHECK(err_code);
477
478     // Configure the BLE stack using the default settings.
479     // Fetch the start address of the application RAM.
480     uint32_t ram_start = 0;
481     err_code = nrf_sdh_ble_default_cfg_set(APP_BLE_CONN_CFG_TAG, &ram_start);
482     APP_ERROR_CHECK(err_code);
483
484     // Enable BLE stack.
485     err_code = nrf_sdh_ble_enable(&ram_start);
486     APP_ERROR_CHECK(err_code);
487
488     // Register a handler for BLE events.
489     NRF_SDH_BLE_OBSERVER(m_ble_observer, APP_BLE_OBSERVER_PRIO, ble_evt_handler, NULL);
490 }
491
```

当我们在主函数里使用 scan_start()函数-->sd_ble_gap_scan_start 函数启动主机扫描后，如果主机发现了从机广播，则会产生 BLE_GAP_EVT_ADV_REPORT 事件，也就是广播报告，那么蓝牙事件派发函数就判断执行解析广播的操作，**这个过程不需要连接，直接扫描过程就可以实现**。代码如下：

```
381 */
382 static void ble_evt_handler(ble_evt_t const * p_ble_evt, void * p_context)
383 {
384     ret_code_t            err_code;
385     ble_gap_evt_t const * p_gap_evt = &p_ble_evt->evt.gap_evt;
386
387     switch (p_ble_evt->header.evt_id)
388     {
389         case BLE_GAP_EVT_ADV_REPORT:
390             on_adv_report(&p_gap_evt->params.adv_report);
391             break; // BLE_GAP_EVT_ADV_REPORT
392
393         case BLE_GAP_EVT_CONNECTED:
394             NRF_LOG_INFO("Connected to target");
395             err_code = ble_nus_c_handles_assign(&m_ble_nus_c, p_ble_evt->evt.gap_evt.conn_handle, NULL);//分配处理句柄
396             APP_ERROR_CHECK(err_code);
397
398             err_code = bsp_indication_set(BSP_INDICATE_CONNECTED);//指示灯亮
399             APP_ERROR_CHECK(err_code);
400
401             // 开始发现服务，NUS客户端等待发现结果
402             err_code = ble_db_discovery_start(&m_db_disc, p_ble_evt->evt.gap_evt.conn_handle);
403             APP_ERROR_CHECK(err_code);
404             break;
405
```

发生连接事件

该事件下会触发 on_adv_report 广播报告函数，该函数就是用于报告 MAC 地址，并且通过比对 UUID 发起连接。函数的整个解析过程如下：

当产生 BLE_GAP_EVT_ADV_REPORT 事件，那么就通过 ble_advdata_uuid_find 函数判断是否是需要的 UUID，和之前保存在&m_nus_uuid 指针内的 UUID 进行比较，如果是需要的 UUID，就用过 sd_ble_gap_connect 函数启动连接。如果连接成功，则主机停止扫描，指示灯会点亮，sd_ble_gap_connect 函数的连接实际上是通过 MAC 地址连进行连接，也就是函数内第一个参&p_adv_report->peer_addr 报告里的 MAC 地址。同时通过 LOG 打印输出 MAC 地址。

如果连接失败，则需要通过 sd_ble_gap_scan_start 函数重新开始扫描。代码如下所示：

```
343  static void on_adv_report(ble_gap_evt_adv_report_t const * p_adv_report)
344  {
345      ret_code_t err_code;
346
347      if (ble_advdata_uuid_find(p_adv_report->data.p_data, p_adv_report->data.len, &m_nus_uuid))
348      {
349          err_code = sd_ble_gap_connect(&p_adv_report->peer_addr,
350                                        &m_scan_params,
351                                        &m_connection_param,
352                                        APP_BLE_CONN_CFG_TAG);
353
354          if (err_code == NRF_SUCCESS)
355          {
356              // scan is automatically stopped by the connect
357              err_code = bsp_indication_set(BSP_INDICATE_IDLE);
358              APP_ERROR_CHECK(err_code);
359              NRF_LOG_INFO("Connecting to target %02x%02x%02x%02x%02x%02x",
360                           p_adv_report->peer_addr.addr[0],
361                           p_adv_report->peer_addr.addr[1],
362                           p_adv_report->peer_addr.addr[2],
363                           p_adv_report->peer_addr.addr[3],
364                           p_adv_report->peer_addr.addr[4],
365                           p_adv_report->peer_addr.addr[5]
366                           );
367          }
368      }
369      else
370      {
371          err_code = sd_ble_gap_scan_start(NULL, &m_scan_buffer);
372          APP_ERROR_CHECK(err_code);
373      }
374  }
375
```

发现广播数据

发起蓝牙连接

报告从机的MAC地址

当连接上了蓝牙后，就会产生 BLE_GAP_EVT_CONNECTED 事件，那么在 on_ble_evt 函数中就会启动 GATT 的基础数据发现 ble_db_discovery_start，代码如下：

```
382  static void ble_evt_handler(ble_evt_t const * p_ble_evt, void * p_context)
383  {
384      ret_code_t           err_code;
385      ble_gap_evt_t const * p_gap_evt = &p_ble_evt->evt.gap_evt;
386
387      switch (p_ble_evt->header.evt_id)
388      {
389          case BLE_GAP_EVT_ADV_REPORT:
390              on_adv_report(&p_gap_evt->params.adv_report);
391              break; // BLE_GAP_EVT_ADV_REPORT
392
393          case BLE_GAP_EVT_CONNECTED:
394              NRF_LOG_INFO("Connected to target");
395              err_code = ble_nus_c_handles_assign(&m_ble_nus_c, p_ble_evt->evt.gap_evt.conn_handle, NULL);//分配处理句柄
396              APP_ERROR_CHECK(err_code);
397
398              err_code = bsp_indication_set(BSP_INDICATE_CONNECTED);//指示灯亮
399              APP_ERROR_CHECK(err_code);
400
401              // 开始发现服务，NUS客户端等待发现结果
402              err_code = ble_db_discovery_start(&m_db_disc, p_ble_evt->evt.gap_evt.conn_handle);
403              APP_ERROR_CHECK(err_code);
404              break;
```

连接成功就开始发现服务

一旦在函数中通过 ble_db_discovery_start 启动了 GATT 基础数据的发现后，协议栈会触发发现主服务等事件，触发事件后要执行一些对应操作。该函数最后一句 discovery_start 函数则会启动主服务发现，代码如下：

```
910 uint32_t ble_db_discovery_start(ble_db_discovery_t * const p_db_discovery, uint16_t conn_handle)
911 {
912     VERIFY_PARAM_NOT_NULL(p_db_discovery);
913     VERIFY_MODULE_INITIALIZED();
914
915     if (m_num_of_handlers_reg == 0)
916     {
917         // No user modules were registered. There are no services to discover.
918         return NRF_ERROR_INVALID_STATE;
919     }
920
921     if (p_db_discovery->discovery_in_progress)
922     {
923         return NRF_ERROR_BUSY;
924     }
925
926     return discovery_start(p_db_discovery, conn_handle);
927 }
```

```
870 static uint32_t discovery_start(ble_db_discovery_t * const p_db_discovery, uint16_t conn_handle)
871 {
872     uint32_t err_code;
873     ble_gatt_db_srv_t * p_srv_being_discovered;
874
875     memset(p_db_discovery, 0x00, sizeof(ble_db_discovery_t));
876
877     p_db_discovery->conn_handle = conn_handle;
878
879     m_pending_usr_evt_index    = 0;
880
881     p_db_discovery->discoveries_count = 0;
882     p_db_discovery->curr_srv_ind      = 0;
883     p_db_discovery->curr_char_ind     = 0;
884
885     p_srv_being_discovered = &(p_db_discovery->services[p_db_discovery->curr_srv_ind]);
886     p_srv_being_discovered->srv_uuid = m_registered_handlers[p_db_discovery->curr_srv_ind];
887
888     NRF_LOG_DEBUG("Starting discovery of service with UUID 0x%x on connection handle 0x%x.",
889                   p_srv_being_discovered->srv_uuid.uuid, conn_handle);
890
891     err_code = sd_ble_gattc_primary_services_discover(conn_handle,
892                                                       SRV_DISC_START_HANDLE,
893                                                       &(p_srv_being_discovered->srv_uuid));
894     if (err_code != NRF_ERROR_BUSY)
895     {
896         VERIFY_SUCCESS(err_code);
897         p_db_discovery->discovery_in_progress = true;
898         p_db_discovery->discovery_pending     = false;
899     }
900     else
901     {
902         p_db_discovery->discovery_in_progress = true;
903         p_db_discovery->discovery_pending     = true;
904     }
905
906     return NRF_SUCCESS;
907 }
```

一旦启动可主服务发现后，这些对应的发现处理过程就交给了派发函数中的 ble_db_discovery_on_ble_evt 数据发现事件派发了。数据发现派发由 BLE_DB_DISCOVERY_DEF 进行声明：

```
85
86 BLE_NUS_C_DEF(m_ble_nus_c);                                    /**< BLE NUS service client instance. */
87 NRF_BLE_GATT_DEF(m_gatt);                                      /**< GATT module instance. */
88 BLE_DB_DISCOVERY_DEF(m_db_disc);                               /**< DB discovery module instance. */
```

发现派发函数

```
87  #define BLE_DB_DISCOVERY_DEF(_name)
88  static ble_db_discovery_t _name = {.discovery_in_progress = 0,
89                                     .discovery_pending = 0,
90                                     .conn_handle = BLE_CONN_HANDLE_INVALID};
91  NRF_SDH_BLE_OBSERVER(_name ## _obs,
92                       BLE_DB_DISC_BLE_OBSERVER_PRIO,
93                       ble_db_discovery_on_ble_evt, &_name)
94
```

下面看看 ble_db_discovery_on_ble_evt 函数内部是如何工作的：

```
910
911  void ble_db_discovery_on_ble_evt(ble_db_discovery_t * const p_db_discovery,
912                                   const ble_evt_t * const    p_ble_evt)
913  {
914      VERIFY_PARAM_NOT_NULL_VOID(p_db_discovery);
915      VERIFY_PARAM_NOT_NULL_VOID(p_ble_evt);
916      VERIFY_MODULE_INITIALIZED_VOID();
917
918      switch (p_ble_evt->header.evt_id)
919      {
920          case BLE_GATTC_EVT_PRIM_SRVC_DISC_RSP://主服务发现报告
921              on_primary_srv_discovery_rsp(p_db_discovery, &(p_ble_evt->evt.gattc_evt));
922              break;
923
924          case BLE_GATTC_EVT_CHAR_DISC_RSP://找到主服务特征值后触发特征值报告
925              on_characteristic_discovery_rsp(p_db_discovery, &(p_ble_evt->evt.gattc_evt));
926              break;
927
928          case BLE_GATTC_EVT_DESC_DISC_RSP://触发描述符查找
929              on_descriptor_discovery_rsp(p_db_discovery, &(p_ble_evt->evt.gattc_evt));
930              break;
931
932          case BLE_GAP_EVT_DISCONNECTED:
933              on_disconnected(p_db_discovery, &(p_ble_evt->evt.gap_evt));
934              break;
935
936          default:
937              break;
938      }
939  }
940
```

数据发现事件首先需要做的是发现主服务，也就是当触发 BLE_GAP_EVT_CONNECTED 连接事件后，启动 ble_db_discovery_start，会产生一个 BLE_GATTC_EVT_PRIM_SRVC_DISC_RSP 事件，也就是主服务发现事件，这个事件下通过 on_primary_srv_discovery_rsp 找主服务的信息参数：UUID 和特征值，如下代码所示：

```
*/
static void on_primary_srv_discovery_rsp(ble_db_discovery_t * const    p_db_discovery,
                                         const ble_gattc_evt_t * const p_ble_gattc_evt)
{
    ble_gatt_db_srv_t                    * p_srv_being_discovered;
    p_srv_being_discovered = &(p_db_discovery->services[p_db_discovery->curr_srv_ind]);

    if (p_ble_gattc_evt->conn_handle != p_db_discovery->conn_handle)
    {
        return;
    }
    if (p_ble_gattc_evt->gatt_status == BLE_GATT_STATUS_SUCCESS)
    {
        uint32_t err_code;
        const ble_gattc_evt_prim_srvc_disc_rsp_t * p_prim_srvc_disc_rsp_evt;

        DB_LOG("Found service UUID 0x%x\r\n", p_srv_being_discovered->srv_uuid.uuid);

        p_prim_srvc_disc_rsp_evt = &(p_ble_gattc_evt->params.prim_srvc_disc_rsp);

        p_srv_being_discovered->srv_uuid     = p_prim_srvc_disc_rsp_evt->services[0].uuid;//uuid报告
        p_srv_being_discovered->handle_range = p_prim_srvc_disc_rsp_evt->services[0].handle_range;

        err_code = characteristics_discover(p_db_discovery,
                                            p_ble_gattc_evt->conn_handle);//特征值发现

        if (err_code != NRF_SUCCESS)
        {
            p_db_discovery->discovery_in_progress = false;

            // Error with discovering the service.
            // Indicate the error to the registered user application.
            discovery_error_evt_trigger(p_db_discovery,
                                        err_code,
                                        p_ble_gattc_evt->conn_handle);

            m_pending_user_evts[0].evt.evt_type    = BLE_DB_DISCOVERY_AVAILABLE;
            m_pending_user_evts[0].evt.conn_handle = p_ble_gattc_evt->conn_handle;
            //m_evt_handler(&m_pending_user_evts[0].evt);
        }
    }
```

如果发现主服务特征值参数完成，则触发 BLE_GATTC_EVT_CHAR_DISC_RSP 事件，这个事件下，发现从机设备的特征参数，使用 on_characteristic_Discovery _rsp 函数，这个函数需要有几个地方说明一下：

1：特征值数量肯定不是只有一个，一个服务里定义了多个特征值，在代码里设置了 BLE_GATT_DB_MAX_CHARS 这个宏定义，定义特征值最大个数。那么在发现服务处理里，必须要找到全部的特征值才会触发下面的事件。当我们找到的特征值等于设置 BLE_GATT_DB_MAX_CHARS，就 perform_desc_discov = true 来表示找到了全部特征值。

```
#ifndef BLE_GATT_DB_MAX_CHARS
#define BLE_GATT_DB_MAX_CHARS 6        /**< The maximum number of characteristics present in a service record. */
#endif // BLE_GATT_DB_MAX_CHARS
```

2:在找到全部特征值后，会启动查找描述符的查找：

err_code = descriptors_discover(p_db_discovery, &raise_discov_complete);描述符和特征值一样，是有多个的，如果一次查找完了，则在函数里 raise_discov_complete 标志位为 1，直接在这个函数里结束整个查找过程。如果没有查找完了？这里就好触发 BLE_GATTC_EVT_DESC_DISC_RSP 事件，进入专门的描述符查找函数。

代码如下：

```
563     * @param[in] p_ble_gattc_evt   Pointer to the GATT Client event.
564     */
565    static void on_characteristic_discovery_rsp(ble_db_discovery_t * const    p_db_discovery,
566                                                const ble_gattc_evt_t * const  p_ble_gattc_evt)
567    {
568        uint32_t                err_code;
569        ble_gatt_db_srv_t     * p_srv_being_discovered;
570        bool                    perform_desc_discov = false;
571
572        p_srv_being_discovered = &(p_db_discovery->services[p_db_discovery->curr_srv_ind]);
573
574        if (p_ble_gattc_evt->gatt_status == BLE_GATT_STATUS_SUCCESS)
575        {
576            const ble_gattc_evt_char_disc_rsp_t * p_char_disc_rsp_evt;
577
578            p_char_disc_rsp_evt = &(p_ble_gattc_evt->params.char_disc_rsp);
579
580            // Find out the number of characteristics that were previously discovered (in earlier
581            // characteristic discovery responses, if any).
582            uint8_t num_chars_prev_disc = p_srv_being_discovered->char_count;
583
584            // Find out the number of characteristics that are currently discovered (in the
585            // characteristic discovery response being handled).
586            uint8_t num_chars_curr_disc = p_char_disc_rsp_evt->count;
587
588            // Check if the total number of discovered characteristics are supported by this module.
589            if ((num_chars_prev_disc + num_chars_curr_disc) <= BLE_GATT_DB_MAX_CHARS)
590            {
594            else
595            {
601            uint32_t i;
602            uint32_t j;
603
604            for (i = num_chars_prev_disc, j = 0; i < p_srv_being_discovered->char_count; i++, j++)
605            {
612            ble_gattc_char_t * p_last_known_char;
613
614            p_last_known_char = &(p_srv_being_discovered->charateristics[i - 1].characteristic);
615
616            // If no more characteristic discovery is required, or if the maximum number of supported
617            // characteristic per service has been reached, descriptor discovery will be performed.
618            if (
619                !is_char_discovery_reqd(p_db_discovery, p_last_known_char) ||
620                (p_srv_being_discovered->char_count == BLE_GATT_DB_MAX_CHARS)//特征值的个数
621               )
622            {
623                perform_desc_discov = true;//表示找到了全部特征值
624            }
625            else
626            {
642        }
```

```
642        }
643        else
644        {
645            // The previous characteristic discovery resulted in no characteristics.
646            // descriptor discovery should be performed.
647            perform_desc_discov = true;
648        }
649
650        if (perform_desc_discov)
651        {
652            bool raise_discov_complete;
653
654            p_db_discovery->curr_char_ind = 0;
655
656            err_code = descriptors_discover(p_db_discovery, &raise_discov_complete);//全部找到后开始找描述符
657
658            if (err_code != NRF_SUCCESS)
659            {
660                p_db_discovery->discovery_in_progress = false;
661
662                discovery_error_evt_trigger(p_db_discovery, err_code);
663
664                return;
665            }
666            if (raise_discov_complete)
667            {
668                // No more characteristics and descriptors need to be discovered. Discovery is complete.
669                // Send a discovery complete event to the user application.
670                DB_LOG("[DB]: Discovery of service with UUID 0x%x completed with success for Connection"
671                       " handle %d\r\n", p_srv_being_discovered->srv_uuid.uuid,
672                       p_db_discovery->conn_handle);
673
674                discovery_complete_evt_trigger(p_db_discovery, true);
675
676                on_srv_disc_completion(p_db_discovery);
677            }
678        }
679    }
```

找到全部特征值标志位

如果发现了全部的特征值，开始查描述符，描述符没有全部查完，就需要重新查找，则触发 BLE_GATTC_EVT_DESC_DISC_RSP 事件，这个事件下，发现特征值的属性（也就是描述符），使用 on_descriptor_discovery_rsp 函数：

```
890
891         case BLE_GATTC_EVT_CHAR_DISC_RSP://找到主服务特征值后触发特征值报告
892             on_characteristic_discovery_rsp(p_db_discovery, &(p_ble_evt->evt.gattc_evt));
893             break;
894
895         case BLE_GATTC_EVT_DESC_DISC_RSP://触发描述符查找
896             on_descriptor_discovery_rsp(p_db_discovery, &(p_ble_evt->evt.gattc_evt));
897             break;
898
899         default:
900             break;
901     }
902 }
903
```

进入到描述符发现函数 on_descriptor_discovery_rsp 中，当描述符发现完成后，raise_discov_complete 会被置位，这时会在函数中调用发现完成事件出触发函数 discovery_complete_evt_trigger。

```
729 static void on_descriptor_discovery_rsp(ble_db_discovery_t * const    p_db_discovery,
730                                          const ble_gattc_evt_t * const p_ble_gattc_evt)
731 {
732     const ble_gattc_evt_desc_disc_rsp_t * p_desc_disc_rsp_evt;
733     ble_gatt_db_srv_t                   * p_srv_being_discovered;
734
735     if (p_ble_gattc_evt->conn_handle != p_db_discovery->conn_handle)
736     {
740     p_srv_being_discovered = &(p_db_discovery->services[p_db_discovery->curr_srv_ind]);
741
742     p_desc_disc_rsp_evt = &(p_ble_gattc_evt->params.desc_disc_rsp);
743
744     ble_gatt_db_char_t * p_char_being_discovered =
745         &(p_srv_being_discovered->charateristics[p_db_discovery->curr_char_ind]);
746
747     if (p_ble_gattc_evt->gatt_status == BLE_GATT_STATUS_SUCCESS)
748     {
788     bool raise_discov_complete = false;
789
790     if ((p_db_discovery->curr_char_ind + 1) == p_srv_being_discovered->char_count)
791     {
792         // No more characteristics and descriptors need to be discovered. Discovery is complete.
793         // Send a discovery complete event to the user application.
794
795         raise_discov_complete = true;          没有新的特征值和描述符需要发现，则发现完成
796     }
797     else
798     {
823     if (raise_discov_complete)
824     {
825         NRF_LOG_DEBUG("Discovery of service with UUID 0x%x completed with success"
826                       " on connection handle 0x%x.",
827                       p_srv_being_discovered->srv_uuid.uuid,
828                       p_ble_gattc_evt->conn_handle);          启动发现完成触发函数
829
830         discovery_complete_evt_trigger(p_db_discovery, true, p_ble_gattc_evt->conn_handle);
831         on_srv_disc_completion(p_db_discovery, p_ble_gattc_evt->conn_handle);
832     }
```

发现完成事件出触发函数 discovery_complete_evt_trigger 中，is_srv_found 被设置为 ture，那么则会在函数中触发 BLE_DB_DISCOVERY_COMPLETE 事件，代码如下图所示：

```
191 static void discovery_complete_evt_trigger(ble_db_discovery_t * p_db_discovery,
192                                             bool                is_srv_found,
193                                             uint16_t            conn_handle)
194 {
195     ble_db_discovery_evt_handler_t   p_evt_handler;
196     ble_gatt_db_srv_t              * p_srv_being_discovered;
197
198     p_srv_being_discovered = &(p_db_discovery->services[p_db_discovery->curr_srv_ind]);
199
200     p_evt_handler = registered_handler_get(&(p_srv_being_discovered->srv_uuid));
201
202     if (p_evt_handler != NULL)
203     {
204         if (m_pending_usr_evt_index < DB_DISCOVERY_MAX_USERS)
205         {
206             // Insert an event into the pending event list.
207             m_pending_user_evts[m_pending_usr_evt_index].evt.conn_handle = conn_handle;
208             m_pending_user_evts[m_pending_usr_evt_index].evt.params.discovered_db =
209                 *p_srv_being_discovered;
210
211             if (is_srv_found)
212             {
213                 m_pending_user_evts[m_pending_usr_evt_index].evt.evt_type =
214                     BLE_DB_DISCOVERY_COMPLETE;
215             }
216             else
217             {
218                 m_pending_user_evts[m_pending_usr_evt_index].evt.evt_type =
219                     BLE_DB_DISCOVERY_SRV_NOT_FOUND;
220             }
221
222             m_pending_user_evts[m_pending_usr_evt_index].evt_handler = p_evt_handler;
223             m_pending_usr_evt_index++;
224
225             if (m_pending_usr_evt_index == m_num_of_handlers_reg)
226             {
227                 // All registered modules have pending events. Send all pending events to the use
228                 // modules.
229                 pending_user_evts_send();
230             }
231             else
232             {
233                 // Too many events pending. Do nothing. (Ideally this should not happen.)
234             }
235         }
236     }
237 }
```

在主函数中，调用数据发现初始化的时候，设置了数据发现中断函数 db_disc_handler，该函数内会调用一个数据中断处理函数 ble_nus_c_on_db_disc_evt。

```
626 /** @brief Function for initializing the Database Discovery Module. */
627 static void db_discovery_init(void)
628 {
629     ret_code_t err_code = ble_db_discovery_init(db_disc_handler);
630     APP_ERROR_CHECK(err_code);
631 }
```

数据发现处理中断

```
167 static void db_disc_handler(ble_db_discovery_evt_t * p_evt)
168 {
169     ble_nus_c_on_db_disc_evt(&m_ble_nus_c, p_evt);
170 }
```

数据发现中断处理

数据发现中断处理函数 ble_nus_c_on_db_disc_evt 的主要功能就是当数据发现标志 BLE_DB_DISCOVERY_COMPLETE 完成后，会触发 BLE_NUS_C_EVT_DISCOVERY_COMPLETE 串口处理事件，代码如下图所示。

ble_gatt_db.h | main.c* | ble_nus_c.c*

```
53  NRF_LOG_MODULE_REGISTER();
54
55  void ble_nus_c_on_db_disc_evt(ble_nus_c_t * p_ble_nus_c, ble_db_discovery_evt_t * p_evt)
56  {
57      ble_nus_c_evt_t nus_c_evt;
58      memset(&nus_c_evt,0,sizeof(ble_nus_c_evt_t));
59
60      ble_gatt_db_char_t * p_chars = p_evt->params.discovered_db.charateristics;
61
62      // Check if the NUS was discovered.
63      if (    (p_evt->evt_type == BLE_DB_DISCOVERY_COMPLETE)
64          &&  (p_evt->params.discovered_db.srv_uuid.uuid == BLE_UUID_NUS_SERVICE)
65          &&  (p_evt->params.discovered_db.srv_uuid.type == p_ble_nus_c->uuid_type))
66      {
67          for (uint32_t i = 0; i < p_evt->params.discovered_db.char_count; i++)
68          {
69              switch (p_chars[i].characteristic.uuid.uuid)
70              {
71                  case BLE_UUID_NUS_RX_CHARACTERISTIC:
72                      nus_c_evt.handles.nus_rx_handle = p_chars[i].characteristic.handle_value;
73                      break;
74
75                  case BLE_UUID_NUS_TX_CHARACTERISTIC:
76                      nus_c_evt.handles.nus_tx_handle = p_chars[i].characteristic.handle_value;
77                      nus_c_evt.handles.nus_tx_cccd_handle = p_chars[i].cccd_handle;
78                      break;
79
80                  default:
81                      break;
82              }
83          }
84          if (p_ble_nus_c->evt_handler != NULL)
85          {
86              nus_c_evt.conn_handle = p_evt->conn_handle;
87              nus_c_evt.evt_type    = BLE_NUS_C_EVT_DISCOVERY_COMPLETE;
88              p_ble_nus_c->evt_handler(p_ble_nus_c, &nus_c_evt);
89          }
90      }
91  }
```

那么整个过程就实现了连接后的基础数据发现，并且触发了蓝牙串口发现完成事件。

总结一下 db 发现的整个过程：

连接发现服务后会产生 case BLE_GATTC_EVT_PRIM_SRVC_DISC_RSP（主服务发现事件）

-->on_primary_srv_discovery_rsp(p_db_discovery, &(p_ble_evt->evt.gattc_evt));

-->err_code = characteristics_discover(p_db_discovery);发现主服务的特征值后触发

-->case BLE_GATTC_EVT_CHAR_DISC_RSP:（特征值发现事件）

-->on_characteristic_discovery_rsp(p_db_discovery, &(p_ble_evt->evt.gattc_evt));发现特征值

-->判断是否特征值已经全部发现或者特征值超过了预定义的个数

--> err_code = characteristics_discover(p_db_discovery);(开始扫描描述符)触发

-->case BLE_GATTC_EVT_DESC_DISC_RSP:（描述符发现事件）

--> on_descriptor_discovery_rsp(p_db_discovery, &(p_ble_evt->evt.gattc_evt));(继续发现描述符，并且判断特征值已经全部发现或者特征值超过了预定义的个数)

-->判断是否发现完毕，触发 BLE_DB_DISCOVERY_COMPLETE 事件

-->触发串口服务发现完成事件标志 BLE_NUS_C_EVT_DISCOVERY_COMPLETE

因此。ble_db_discovery_on_ble_evt 数据发现事件派发函数就负责实现上面的这些操作。

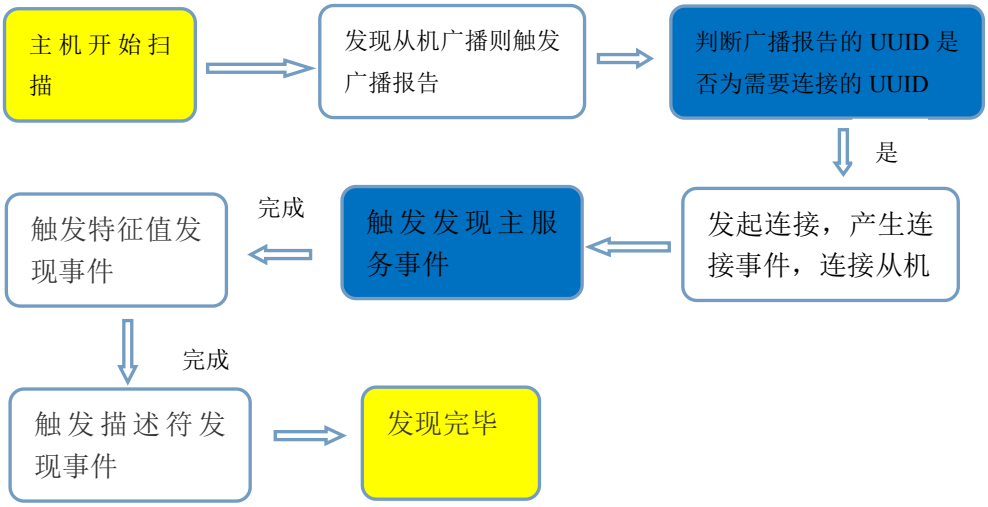
那么整个主机和从机设备的连接过程可以归纳为：

1：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，开始解析 UUID,如果是需要对应的 UUID，则产生连接对应的 MAC 地址的硬件。

2：连接后触发产生 BLE_GAP_EVT_CONNECTED 事件，启动 GATT 的基础数据发现 ble_db_discovery_start。

3：发现过程全程交给 ble_db_discovery_on_ble_evt 派发实现。

归纳如下图所示：



1.3 主机蓝牙串口数据流分析：

1.3.1 从机到主机的数据流向：

从机发送数据过来，主机接收数据，然后主机通过串口发送到 PC 用串口调制助手显示。那么整个数据流在程序中如何实现的？

首先弄清楚，从机发送到主机的接收数据，这个蓝牙过程的属性应该是通知类型。其实整个过程都交给由派发函数 ble_nus_c_on_ble_evt(&m_ble_nus_c,p_ble_evt)；进行处理了。下面我们就通过分析代码来详细解析下过程：

首先，在 main 主函数里，初始化函数 nus_c_int();这个函数初始化主机，我们看里面调用的 ble_nus_c_init（）函数：

```
485 }
486 /**@brief Function for initializing the NUS Client串口客户端初始化.
487  */
488 static void nus_c_init(void)
489 {
490     uint32_t         err_code;
491     ble_nus_c_init_t nus_c_init_t;
492
493     nus_c_init_t.evt_handler = ble_nus_c_evt_handler;//客户端处理事件
494
495     err_code = ble_nus_c_init(&m_ble_nus_c, &nus_c_init_t);//客户端初始化
496     APP_ERROR_CHECK(err_code);
497 }
498
```

&nus_c_init_t 结构体使用了前面定义的客户端处理事件 nus_c_init_t.evt_handler = ble_nus_c_evt_handler,这个触发事件就在 nus_c_int()初始化函数里进行了触发，如下代码所示：

```
98  }
99
100 uint32_t ble_nus_c_init(ble_nus_c_t * p_ble_nus_c, ble_nus_c_init_t * p_ble_nus_c_init)//客户端初始化
101 {
102     uint32_t      err_code;
103     ble_uuid_t    uart_uuid;
104     ble_uuid128_t nus_base_uuid = NUS_BASE_UUID;//基础UUID，和从机UUID一致
105
106     if ((p_ble_nus_c == NULL) || (p_ble_nus_c_init == NULL))
107     {
108         return NRF_ERROR_NULL;
109     }
110
111     err_code = sd_ble_uuid_vs_add(&nus_base_uuid, &p_ble_nus_c->uuid_type);//UUID的添加
112     if (err_code != NRF_SUCCESS)
113     {
114         return err_code;
115     }
116
117     uart_uuid.type = p_ble_nus_c->uuid_type;//服务类型
118     uart_uuid.uuid = BLE_UUID_NUS_SERVICE;//主服务UUID类型
119
120     // save the pointer to the ble_uart_c_t struct locally
121     mp_ble_nus_c = p_ble_nus_c;
122
123     p_ble_nus_c->conn_handle   = BLE_CONN_HANDLE_INVALID;//无效的连接句柄
124     p_ble_nus_c->evt_handler   = p_ble_nus_c_init->evt_handler;//触发串口客户端操作
125     p_ble_nus_c->nus_rx_handle = BLE_GATT_HANDLE_INVALID;//无效的属性处理
126     p_ble_nus_c->nus_tx_handle = BLE_GATT_HANDLE_INVALID;
127
128     return ble_db_discovery_evt_register(&uart_uuid, db_discover_evt_handler);//应用程序可以使用这个函数来通知
129
130 }
131
```

一旦触发了 ble_nus_c_evt_handler，我们进入到客户端处理事件函数内部，这个函数内部是对应各种事件后触发的对应操作。如下图所示，当发现前面一节的蓝牙串口主机已经发现完了从机，就触发了 BLE_NUS_C_EVT_DISCOVERY_COMPLETE 事件，那么这种该事件情况下会通过函数 ble_nus_c_tx_notif_enable (p_ble_nus_c)使能通知，代码如下：

```c
static void ble_nus_c_evt_handler(ble_nus_c_t * p_ble_nus_c, ble_nus_c_evt_t const * p_ble_nus_evt)
{
    ret_code_t err_code;

    switch (p_ble_nus_evt->evt_type)
    {
        case BLE_NUS_C_EVT_DISCOVERY_COMPLETE://串口主机发现完成事件
            NRF_LOG_INFO("Discovery complete.");
            //分配主机蓝牙操作句柄的空间
            err_code = ble_nus_c_handles_assign(p_ble_nus_c, p_ble_nus_evt->conn_handle, &p_ble_nus_evt->handles);
            APP_ERROR_CHECK(err_code);
            //使能TX通知
            err_code = ble_nus_c_tx_notif_enable(p_ble_nus_c);
            APP_ERROR_CHECK(err_code);
            NRF_LOG_INFO("Connected to device with Nordic UART Service.");
            break;

        case BLE_NUS_C_EVT_NUS_TX_EVT://主机串口TX事件
            //调用脑打印输出
            ble_nus_chars_received_uart_print(p_ble_nus_evt->p_data, p_ble_nus_evt->data_len);
            break;

        case BLE_NUS_C_EVT_DISCONNECTED:
            NRF_LOG_INFO("Disconnected.");
            scan_start();
            break;
    }
}
/**@snippet [Handling events from the ble_nus_c module] */
```

1.如何蓝牙主机发现事件完成

使能从机上传主机的通知

2.从机数据上传触发TX事件

主机通过串口在电脑上打印

蓝牙发现完成事件并不能让我们直接进入图上第二个事件 BLE_NUS_C_EVT_NUS_TX_EV 主机发现串口 TX 事件中去接收从机数据的。而是需要通过下面的步骤实现进入：在主机发现完成事件下，触发 ble_nus_c_tx_notif_enable (p_ble_nus_c)使能通知函数,这个函数里就有一个 CCCD 描述符的配置函数：cccd_configure(p_ble_nus_c->conn_handle,p_ble_nus_c->nus_rx_cccd_handle, true);这个函数最后一个参数为 ture。

```c
uint32_t ble_nus_c_tx_notif_enable(ble_nus_c_t * p_ble_nus_c)
{
    VERIFY_PARAM_NOT_NULL(p_ble_nus_c);

    if ( (p_ble_nus_c->conn_handle == BLE_CONN_HANDLE_INVALID)
       ||(p_ble_nus_c->handles.nus_tx_cccd_handle == BLE_GATT_HANDLE_INVALID)
       )
    {
        return NRF_ERROR_INVALID_STATE;
    }
    return cccd_configure(p_ble_nus_c->conn_handle,p_ble_nus_c->handles.nus_tx_cccd_handle, true);
}
```

进入这个函数，也就是最后一个参数 enable 为 1，那么就产生了 BLE_GATT_HVX_NOTIFICATION 事件，同时通过 sd_ble_gattc_write(conn_handle, &write_params）函数把里面配置的 GATTC 写参数发送到从机，从而使能从机，这个过程和点击手机 nrf connect APP 上的使能从机通知功能按键一个作用。代码如下：

```c
/**@brief Function for creating a message for writing to the CCCD.
 */
static uint32_t cccd_configure(uint16_t conn_handle, uint16_t cccd_handle, bool enable)//CCCD配置
{
    uint8_t buf[BLE_CCCD_VALUE_LEN];

    buf[0] = enable ? BLE_GATT_HVX_NOTIFICATION : 0;
    buf[1] = 0;

    const ble_gattc_write_params_t write_params = {
        .write_op = BLE_GATT_OP_WRITE_REQ,
        .flags    = BLE_GATT_EXEC_WRITE_FLAG_PREPARED_WRITE,
        .handle   = cccd_handle,
        .offset   = 0,
        .len      = sizeof(buf),
        .p_value  = buf
    };

    return sd_ble_gattc_write(conn_handle, &write_params);
}
```

从机接到通知使能的数据包后，会给一个回包，这个发回的数据包依靠协议栈回调函数进行处理，就是给派发函数 ble_nus_c_on_ble_evt(&m_ble_nus_c,p_ble_evt);这个函数声明不同与老版本，在 main.c 函数最开头第 86 行，以观察者方式调用：

```
86  BLE_NUS_C_DEF(m_ble_nus_c);                主机服务派发
87  NRF_BLE_GATT_DEF(m_gatt);
88  BLE_DB_DISCOVERY_DEF(m_db_disc);
```

```
83  #define BLE_NUS_C_DEF(_name)
84  static ble_nus_c_t _name;
85  NRF_SDH_BLE_OBSERVER(_name ## _obs,
86                       BLE_NUS_C_BLE_OBSERVER_PRIO,
87                       ble_nus_c_on_ble_evt, &_name)
```

ble_nus_c_on_ble_evt(&m_ble_nus_c,p_ble_evt);串口主机连接事件回调处理，以接收从协议栈 SOFTDEVICE 上发来的事件，解析事件 evt_id，这个通知使能返回的数据包产生一个 BLE_GATTC_EVT_HVX://GATT 通知事件，这个事件下就调用函数 on_hvx(p_ble_nus_c, p_ble_evt)，代码如下：

```
131
132 void ble_nus_c_on_ble_evt(ble_nus_c_t * p_ble_nus_c, const ble_evt_t * p_ble_evt)//函数来处理从SoftDevice BLE事件,
133 {
134     if ((p_ble_nus_c == NULL) || (p_ble_evt == NULL))
135     {
136         return;
137     }
138
139     if ( (p_ble_nus_c->conn_handle != BLE_CONN_HANDLE_INVALID)
140        &&(p_ble_nus_c->conn_handle != p_ble_evt->evt.gap_evt.conn_handle)
141        )
142     {
143         return;
144     }
145
146     switch (p_ble_evt->header.evt_id)//解析发过来的事件ID
147     {
148         case BLE_GATTC_EVT_HVX://GATT通知事件
149             on_hvx(p_ble_nus_c, p_ble_evt);//设置触发RX操作事件,接收蓝牙数据
150             break;
151
152         case BLE_GAP_EVT_DISCONNECTED://断开事件
153             if (p_ble_evt->evt.gap_evt.conn_handle == p_ble_nus_c->conn_handle
154                     && p_ble_nus_c->evt_handler != NULL)
155             {
156                 ble_nus_c_evt_t nus_c_evt;
157
158                 nus_c_evt.evt_type = BLE_NUS_C_EVT_DISCONNECTED://客户断开事件
159
160                 p_ble_nus_c->conn_handle = BLE_CONN_HANDLE_INVALID;
161                 p_ble_nus_c->evt_handler(p_ble_nus_c, &nus_c_evt);
162             }
163             break;
164     }
165 }
166
```

进入 on_hvx(p_ble_nus_c, p_ble_evt)函数，看看内部进行什么操作，代码如下。这个函数内部就做了三个操作：1:触发 BLE_NUS_C_EVT_NUS_TX_EVT 事件 2:保存主机蓝牙接收到的数据。3 保存接收长度。实际上这个数据和数据长度正是串口从机代码中的上传函数 sd_ble_gatts_hvx(p_nus->conn_handle, &hvx_params) 发过来的。

```c
static void on_hvx(ble_nus_c_t * p_ble_nus_c, ble_evt_t const * p_ble_evt)
{
    // HVX can only occur from client sending.
    if (   (p_ble_nus_c->handles.nus_tx_handle != BLE_GATT_HANDLE_INVALID)
        && (p_ble_evt->evt.gattc_evt.params.hvx.handle == p_ble_nus_c->handles.nus_tx_handle)
        && (p_ble_nus_c->evt_handler != NULL))
    {
        ble_nus_c_evt_t ble_nus_c_evt;                                   触发主机接收从机数据

        ble_nus_c_evt.evt_type = BLE_NUS_C_EVT_NUS_TX_EVT;//触发TX操作，接收从机上传数据
        ble_nus_c_evt.p_data   = (uint8_t *)p_ble_evt->evt.gattc_evt.params.hvx.data;//数据
        ble_nus_c_evt.data_len = p_ble_evt->evt.gattc_evt.params.hvx.len;//数据长度

        p_ble_nus_c->evt_handler(p_ble_nus_c, &ble_nus_c_evt);
        NRF_LOG_DEBUG("Client sending data.");
    }
}
```

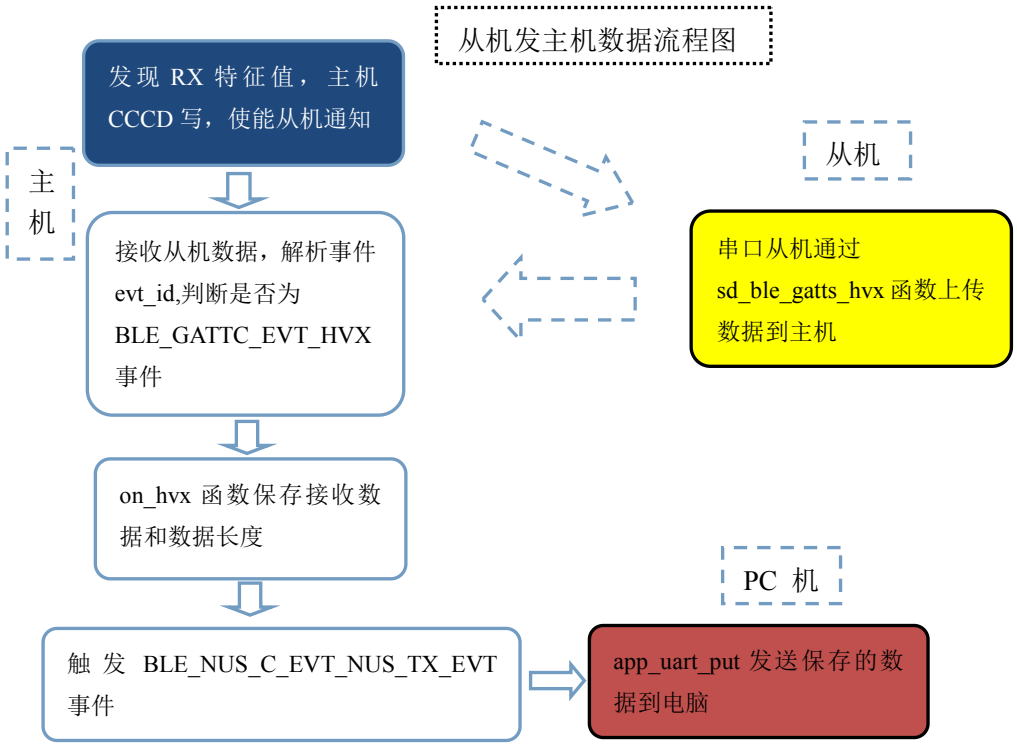
这时才触发第二个 BLE_NUS_C_EVT_NUS_TX_EVT 事件，在主机事件处理函数中，会把上面保存的数据，通过 ble_nus_chars_received_uart_print 把从机发过来，主机存储的数据发送到电脑上。

```c
static void ble_nus_c_evt_handler(ble_nus_c_t * p_ble_nus_c, ble_nus_c_evt_t const * p_ble_nus_evt)
{
    ret_code_t err_code;

    switch (p_ble_nus_evt->evt_type)
    {
        case BLE_NUS_C_EVT_DISCOVERY_COMPLETE://串口主机发现完成事件      1.如何蓝牙主机发现事件完成
            NRF_LOG_INFO("Discovery complete.");
            //分配主机蓝牙操作句柄的空间
            err_code = ble_nus_c_handles_assign(p_ble_nus_c, p_ble_nus_evt->conn_handle, &p_ble_nus_evt->handles);
            APP_ERROR_CHECK(err_code);
            //使能TX通知
            err_code = ble_nus_c_tx_notif_enable(p_ble_nus_c);          使能从机上传主机的通知
            APP_ERROR_CHECK(err_code);
            NRF_LOG_INFO("Connected to device with Nordic UART Service.");
            break;

        case BLE_NUS_C_EVT_NUS_TX_EVT://主机串口TX事件                  2.从机数据上传触发TX事件
            //向电脑打印输出
            ble_nus_chars_received_uart_print(p_ble_nus_evt->p_data, p_ble_nus_evt->data_len);
            break;

                                                                        主机通过串口在电脑上打印
        case BLE_NUS_C_EVT_DISCONNECTED:
            NRF_LOG_INFO("Disconnected.");
            scan_start();
            break;
    }
}
/**@snippet [Handling events from the ble_nus_c module] */
```

ble_nus_chars_received_uart_print 函数内部实际上就是调用了 app_uart_put 发送到电脑上的，如果需要验证主机发送给从机的数据是否正确，则可以设置 ECHOBACK_BLE_UART_DATA，使得数据从新发回从机。

```
178  static void ble_nus_chars_received_uart_print(uint8_t * p_data, uint16_t data_len)
179  {
180      ret_code_t ret_val;
181
182      NRF_LOG_DEBUG("Receiving data.");
183      NRF_LOG_HEXDUMP_DEBUG(p_data, data_len);
184
185      for (uint32_t i = 0; i < data_len; i++)
186      {
187          do
188          {
189              ret_val = app_uart_put(p_data[i]);          1 发生到电脑上
190              if ((ret_val != NRF_SUCCESS) && (ret_val != NRF_ERROR_BUSY))
191              {
192                  NRF_LOG_ERROR("app_uart_put failed for index 0x%04x.", i);
193                  APP_ERROR_CHECK(ret_val);
194              }
195          } while (ret_val == NRF_ERROR_BUSY);
196      }
197      if (p_data[data_len-1] == '\r')
198      {
199          while (app_uart_put('\n') == NRF_ERROR_BUSY);
200      }
201      if (ECHOBACK_BLE_UART_DATA)                  2．如果要验证是否发送和接收一致，可以定义数据返回
202      {
203          // Send data back to peripheral.
204          do                                                把数据又重新发回从机
205          {
206              ret_val = ble_nus_c_string_send(&m_ble_nus_c, p_data, data_len);
207              if ((ret_val != NRF_SUCCESS) && (ret_val != NRF_ERROR_BUSY))
208              {
209                  NRF_LOG_ERROR("Failed sending NUS message. Error 0x%x. ", ret_val);
210                  APP_ERROR_CHECK(ret_val);
211              }
212          } while (ret_val == NRF_ERROR_BUSY);
213      }
214  }
```

那么上面从从机发数据，主机接收，接收后串口方式发 PC 机的整个数据流过程就讲清楚了，我们归纳如下图所示，注意一定要是在第一节讲的的发现服务完成后，也就是触发了 BLE_NUS_C_EVT_DISCOVERY_COMPLETE **事件：**



1.3.2 主机发送到从机的数据流向：

主机发送到从机，首先通过 PC 机上串口调制助手发数据到主机，主机通过蓝牙传输给从机。那么这整个过程的蓝牙操作属性应该是写。下面分析下这个数据方向怎么实现的。

首先的 PC 机通过串口发主机，这个很简单，就是使用的串口外设串口事件处理 uart_event_handle，代码如下：

```
458
459 /**@brief Function for initializing the UART.
460  */
461 static void uart_init(void)
462 {
463     uint32_t err_code;
464
465     const app_uart_comm_params_t comm_params =
466       {
467         .rx_pin_no    = RX_PIN_NUMBER,
468         .tx_pin_no    = TX_PIN_NUMBER,
469         .rts_pin_no   = RTS_PIN_NUMBER,
470         .cts_pin_no   = CTS_PIN_NUMBER,
471         .flow_control = APP_UART_FLOW_CONTROL_ENABLED,
472         .use_parity   = false,
473         .baud_rate    = UART_BAUDRATE_BAUDRATE_Baud38400
474       };
475
476     APP_UART_FIFO_INIT(&comm_params,
477                        UART_RX_BUF_SIZE,
478                        UART_TX_BUF_SIZE,
479                        uart_event_handle,
480                        APP_IRQ_PRIORITY_LOW,
481                        err_code);
482
483     APP_ERROR_CHECK(err_code);
484 }
```

串口处理事件里，通过 app_uart_get 接收 PC 机发来的数据。如果有数据发送过来，那么直接启动 ble_nus_c_string_send(&m_ble_nus_c, data_array, index)函数，把数据通过蓝牙发送出去：

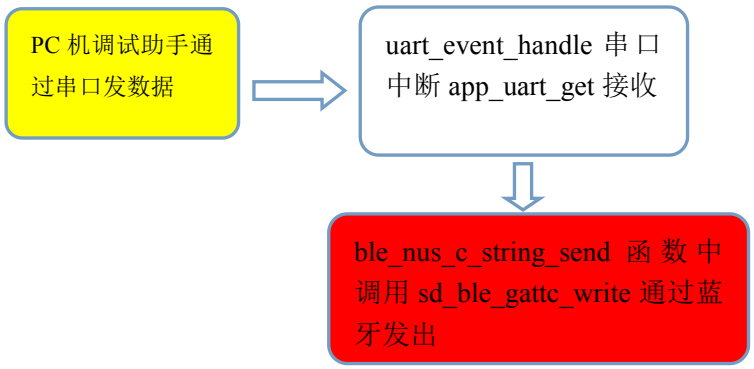
```c
void uart_event_handle(app_uart_evt_t * p_event)
{
    static uint8_t data_array[BLE_NUS_MAX_DATA_LEN];
    static uint16_t index = 0;
    uint32_t ret_val;

    switch (p_event->evt_type)
    {
        /**@snippet [Handling data from UART] */
        case APP_UART_DATA_READY:
            UNUSED_VARIABLE(app_uart_get(&data_array[index]));
            index++;

            if ((data_array[index - 1] == '\n') || (index >= (m_ble_nus_max_data_len)))
            {
                NRF_LOG_DEBUG("Ready to send data over BLE NUS");
                NRF_LOG_HEXDUMP_DEBUG(data_array, index);

                do
                {
                    ret_val = ble_nus_c_string_send(&m_ble_nus_c, data_array, index);
                    if ( (ret_val != NRF_ERROR_INVALID_STATE) && (ret_val != NRF_ERROR_BUSY) )
                    {
                        APP_ERROR_CHECK(ret_val);
                    }
                } while (ret_val == NRF_ERROR_BUSY);

                index = 0;
            }
            break;

        /**@snippet [Handling data from UART] */
        case APP_UART_COMMUNICATION_ERROR:
            NRF_LOG_ERROR("Communication error occurred while handling UART.");
            APP_ERROR_HANDLER(p_event->data.error_communication);
            break;

        case APP_UART_FIFO_ERROR:
            NRF_LOG_ERROR("Error occurred in FIFO module used by UART.");
            APP_ERROR_HANDLER(p_event->data.error_code);
            break;

        default:
            break;
    }
}
```

蓝牙发送出去

ble_nus_c_string_send(&m_ble_nus_c, data_array, index)蓝牙数据发送函数，对比前面的 CCCD 描述符配置函数 cccd_configure(p_ble_nus_c->conn_handle, p_ble_nus_c->nus_rx_cccd_handle, true)，大家看看是不是很相似。其也是通过 sd_ble_gattc_write(conn_handle, &write_params）函数把里面配置的 GATTC 写参数发送到从机，前面的是把通知使能事件作为 BUF 写出去，这里把真正的数据发送出去：

```c
//串口发送

uint32_t ble_nus_c_string_send(ble_nus_c_t * p_ble_nus_c, uint8_t * p_string, uint16_t length)
{
    if (p_ble_nus_c == NULL)
    {
        return NRF_ERROR_NULL;
    }

    if (length > BLE_NUS_MAX_DATA_LEN)
    {
        return NRF_ERROR_INVALID_PARAM;
    }
    if ( p_ble_nus_c->conn_handle == BLE_CONN_HANDLE_INVALID)
    {
        return NRF_ERROR_INVALID_STATE;
    }

    const ble_gattc_write_params_t write_params = {
        .write_op = BLE_GATT_OP_WRITE_CMD,
        .flags    = BLE_GATT_EXEC_WRITE_FLAG_PREPARED_WRITE,
        .handle   = p_ble_nus_c->nus_tx_handle,
        .offset   = 0,
        .len      = length,
        .p_value  = p_string
    };

    return sd_ble_gattc_write(p_ble_nus_c->conn_handle, &write_params);//写出去
}
```

那么这里通过 PC 机调制助手把数据发给主机，主机发送给从机的数据流向就清楚了。



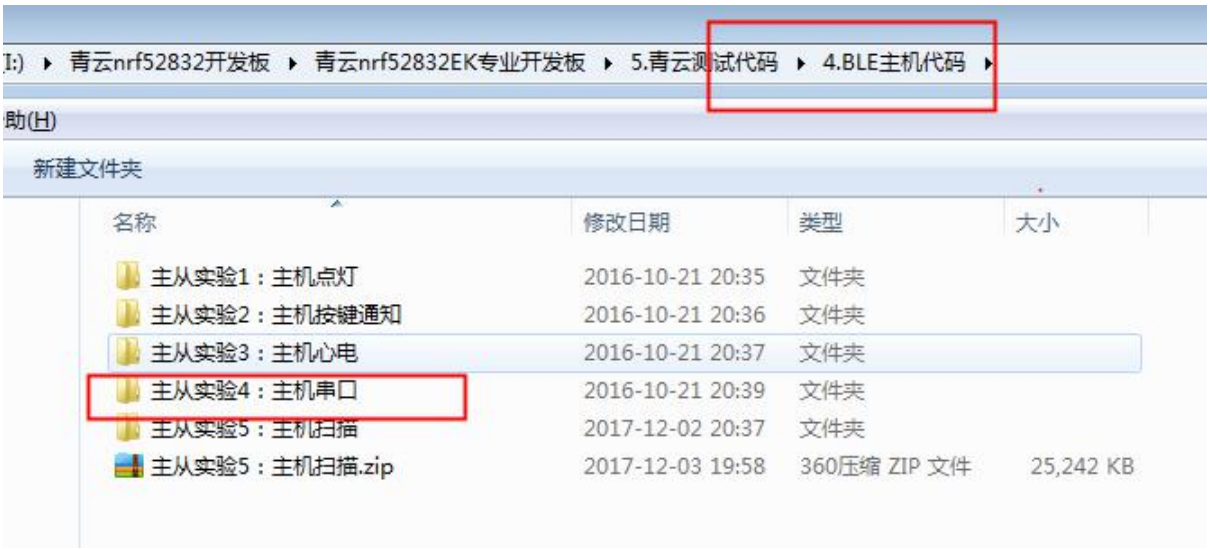
讲了这么多，大家现在应该对整个主机的设备过程有了一个清楚的认识。读者如果理解了整个数据交换过程，我们这讲的目的也达到了。

2 应用与调试

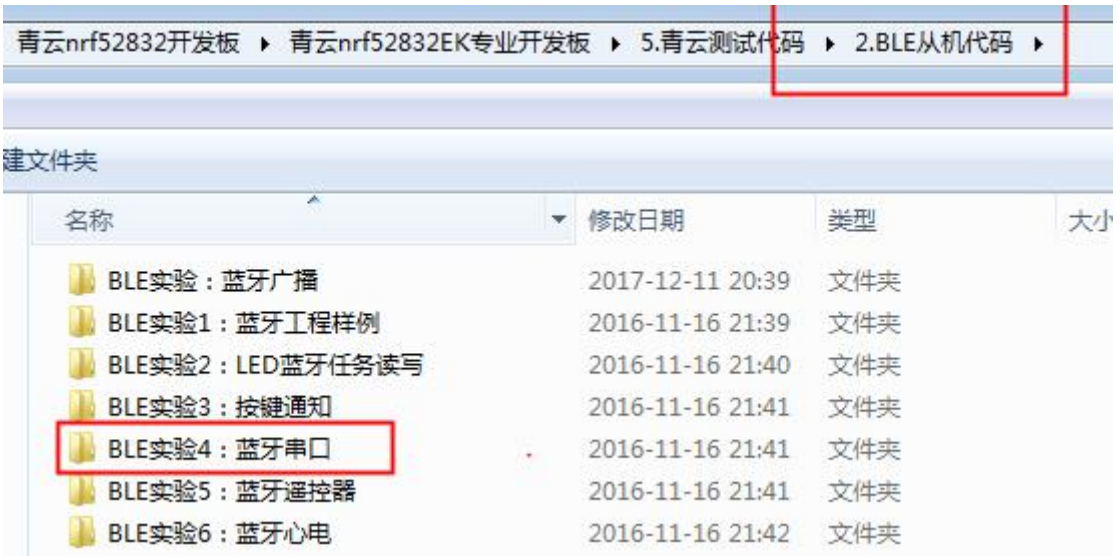
本实验使用两个开发板，不需要使用手机完成，一个开发板为主机，一个开发板为从机。

2.1 软件准备：

在代码文件中，打开工程主从串口实验, 如下所示：主机：



从机：



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 nRF52832 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 S132.

1. 首先采用 nrfgo 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 sofrdevice,点击 browse 选择协议栈 S132 下载到其中一个开发板内，做为主机：

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：

从机下载：

1. 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 sofrdevice，点击 browse 选择协议栈 S132 下载到其中另外一个开发板内，做为从机

2. 协议栈下载完成后，下载从机应用程序，如本篇文章开头所示的从机工程路径，打开后点击 MDK 的 load 按键：

下载完后作为从机。

2.2 实验现象：

把开发板串口接好，打开两个调制助手，设置如下图所示，波特率为 115200，开流控，这时两边就可以互传数据了：