青风带你玩蓝牙 nRF52832 系列教程

----------作者：青风

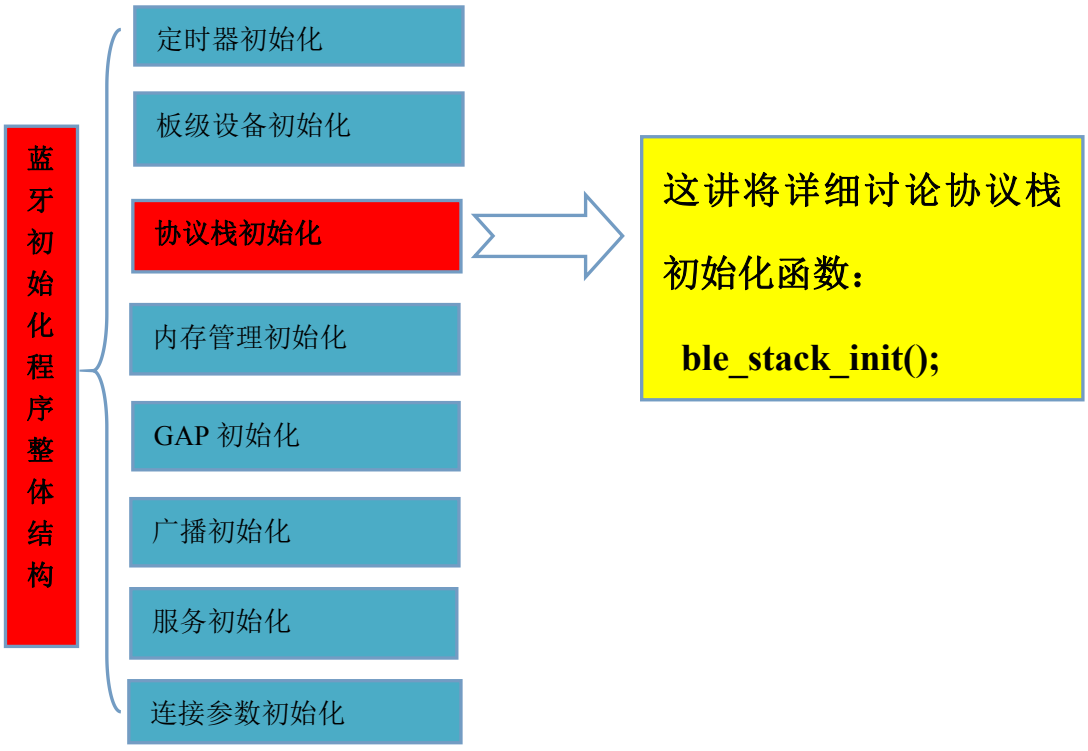
出品论坛：　www.qfv8.com　青风电子社区

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

2.7 蓝牙协议初始化详解

对应蓝牙协议栈的初始化一直是大家关注的问题，Nordic 的协议栈没有开源，但是留下了大部分的配置接口，在我们设计蓝牙的时候是至关重要的，学好蓝牙初始化配置是我们后面的蓝牙设计基础。

本章将通过分析蓝牙协议的一些基本原理，在匹配的 SDK15.0 的蓝牙样例的例子基础上就行分析与讲解，使用的协议栈为：s132。



1：nRF52832 蓝牙协议栈初始化函数结构：

在一个 nrf52832 的工程下，初始化蓝牙协议栈函数为 ble_stack_init()，其基本结构如下图所示，实际上分为三个部分：

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

    err_code = nrf_sdh_enable_request();//协议栈回复使能应答，主要是设置系统时钟。
    APP_ERROR_CHECK(err_code);                                                          1

    // Configure the BLE stack using the default settings.使用默认设置配置协议栈
    // Fetch the start address of the application RAM.获取应用程序RAM的起始地址
    uint32_t ram_start = 0;
    //协议栈默认配置设置  (默认配置标号--连接配置，ram起始地址)
    err_code = nrf_sdh_ble_default_cfg_set(APP_BLE_CONN_CFG_TAG, &ram_start);
    APP_ERROR_CHECK(err_code);                                                          2

    // Enable BLE stack.使能协议栈
    err_code = nrf_sdh_ble_enable(&ram_start);
    APP_ERROR_CHECK(err_code);

    // Register a handler for BLE events.注册蓝牙处理事件
    NRF_SDH_BLE_OBSERVER(m_ble_observer, APP_BLE_OBSERVER_PRIO, ble_evt_handler, NULL);  3
}
```

1：协议栈回复使能应答，主要工作就是协议栈时钟初始化配置

2：初始化协议栈，设置协议栈相关处理函数，使能协议栈

3：注册蓝牙处理调度事件

这个协议栈函数就是完成上面三个工作，那么上面的三个工作是做什么？有什么作用？下面就来具体讨论：

2：协议栈回复使能应答：

首先我们来解析下协议栈回复使能应答函数，这个函数是用于观察者观察初始化协议栈是否开始使能，使能是否成功。同时配置一下协议栈使用的低速时钟。代码具体如下：

```
01. ret_code_t nrf_sdh_enable_request(void)
02. {
03.     ret_code_t ret_code;
04.     if (m_nrf_sdh_enabled)
05.     {
06.         return NRF_ERROR_INVALID_STATE;
07.     }
08.     m_nrf_sdh_continue = true;
09.
10.     // Notify observers about SoftDevice enable request.通知观察者，协议栈使能应答
11.     if (sdh_request_observer_notify(NRF_SDH_EVT_ENABLE_REQUEST) ==
    NRF_ERROR_BUSY)
12.     {
13.         // Enable process was stopped.  使能过程被停止，否则跳出循环
14.         return NRF_SUCCESS;
15.     }
```

```
16.
17.       // Notify observers about starting SoftDevice enable process.通知观察者，协议栈使能开始
18.       sdh_state_observer_notify(NRF_SDH_EVT_STATE_ENABLE_PREPARE);
19.       //首先配置时钟源
20.       nrf_clock_lf_cfg_t const clock_lf_cfg =
21.       {
22.             .source        = NRF_SDH_CLOCK_LF_SRC,
23.             .rc_ctiv       = NRF_SDH_CLOCK_LF_RC_CTIV,
24.             .rc_temp_ctiv = NRF_SDH_CLOCK_LF_RC_TEMP_CTIV,
25.             .accuracy      = NRF_SDH_CLOCK_LF_ACCURACY
26.       };
27.        //进入临界区域的宏。
28.       CRITICAL_REGION_ENTER();
29.     #ifdef ANT_LICENSE_KEY       //如果定义的是 ANT
30.       ret_code = sd_softdevice_enable(&clock_lf_cfg, app_error_fault_handler,
    ANT_LICENSE_KEY);
31. #else
32.        //否则，BLE 下使能之前配置的时钟参数
33.       ret_code = sd_softdevice_enable(&clock_lf_cfg, app_error_fault_handler);
34. #endif
35.
36.       m_nrf_sdh_enabled = (ret_code == NRF_SUCCESS);//使能成功
37.        //退出临界区
38.       CRITICAL_REGION_EXIT();
39.
40.       if (ret_code != NRF_SUCCESS)
41.       {
42.             return ret_code;
43.       }
44.       m_nrf_sdh_continue    = false;
45.       m_nrf_sdh_suspended = false;
46.
47. #ifndef S140
48.       // Set the interrupt priority after enabling the SoftDevice, since
49.       // sd_softdevice_enable() sets the SoftDevice interrupt priority.
50.       swi_interrupt_priority_workaround();
51. #endif
52.
53.       //  使能协议栈中断
54.       // Interrupt priority has already been set by the stack.
55.       softdevices_evt_irq_enable();
56.
57.       //通知观察者，协议栈使能已经完成
58.       sdh_state_observer_notify(NRF_SDH_EVT_STATE_ENABLED);
```

```
59.         return NRF_SUCCESS;
60. }
```

所谓的观察者，就是开发者或者用户，代码中设置了交互方式，比如使用 LED 灯，串口等方式对设备状态进行观察，并且给观察者分配了回调函数。具体的内容我们先来分析下代码。

对代码进行下说明：

第 04--08 行：判断 m_nrf_sdh_enabled 协议栈使能标志位是否使能了，如果使能就返回错误状态表示重复使能，设备重启。如何没有使能，m_nrf_sdh_continue 继续使能标志位被置 1。

第 10--15 行：通知观察者，协议栈使能应答，如果回复忙，则使能过程被停止，否则跳出循环。

第 18 行：通知观察者，协议栈使能开始。

第 20--34 行：配置协议栈时钟。

第 36 行： 配置完后，使能标志位 m_nrf_sdh_enabled 为设置为 1，表示使能成功。

第 54 行： 使能协议栈中断嵌套。

第 58 行：通知观察者，协议栈使能已经完成

上面几个部分就为该段程序所做的主要工作，初始化协议栈时钟，并且通过使能标志位置 1 的方式表示协议栈时钟使能完成，同时完成协议栈中断嵌套使能。最后把协议栈状态通知给观察者。

2.1 协议栈时钟设置

协议栈下需要设置设置产生方式，在函数中设置定义 NRF_CLOCK_LFCLKSRC 为一个结构体，结构体下有设置 4 个参数，如下所示：

```
01.     nrf_clock_lf_cfg_t const clock_lf_cfg =
02.     {
03.         .source        = NRF_SDH_CLOCK_LF_SRC,
04.         .rc_ctiv       = NRF_SDH_CLOCK_LF_RC_CTIV,
05.         .rc_temp_ctiv = NRF_SDH_CLOCK_LF_RC_TEMP_CTIV,
06.         .accuracy      = NRF_SDH_CLOCK_LF_ACCURACY
07.     };
08.
```

第一个参数.source 为设置时钟源，协议栈需要一个低频的时钟源，时钟源有 3 个选择：

a.内部 RC 时钟

b.外部晶振时钟

c.合成的时钟

在 sdk_config.h 文件中，定义了三种协议栈时钟源，分别为：

```
09. // <0=> NRF_CLOCK_LF_SRC_RC            /**< 内部 RC 时钟 */
10. // <1=> NRF_CLOCK_LF_SRC_XTAL          /**< 外部晶振时钟. */
11. // <2=> NRF_CLOCK_LF_SRC_SYNTH       /**< 从高速时钟合成的低速时钟 */
```

三个参数的使用是有区别的，

①：首先谈下外部晶振时钟，要使用外部晶振时钟，在硬件上，必须外接 32.768KHz

低速晶振，这种状态下对电流的消耗是最低的。那么外部晶振在选择的时候，需要考虑不同的精确度，而这体现在第四个参数.xtal_accuracy 参数上，这个参数设置在外部时钟中 NRF_CLOCK_LFCLKSRC_XTAL_ACCURACY_x_PPM,这个 x 是指外部晶体的精度。

```
12. #define NRF_CLOCK_LF_XTAL_ACCURACY_250_PPM (0) /* Default */
13. #define NRF_CLOCK_LF_XTAL_ACCURACY_500_PPM (1)
14. #define NRF_CLOCK_LF_XTAL_ACCURACY_150_PPM (2)
15. #define NRF_CLOCK_LF_XTAL_ACCURACY_100_PPM (3)
16. #define NRF_CLOCK_LF_XTAL_ACCURACY_75_PPM  (4)
17. #define NRF_CLOCK_LF_XTAL_ACCURACY_50_PPM  (5)
18. #define NRF_CLOCK_LF_XTAL_ACCURACY_30_PPM  (6)
19. #define NRF_CLOCK_LF_XTAL_ACCURACY_20_PPM  (7)
```

ppm 是百万分一的意思。在晶振里面，由于晶振的振荡频率随着温度的变化会发生很小的漂移，称之为温漂，有温漂的定义为：温度变化一摄氏度的时候，震荡频率相对于标称值的变化量。如果漂移了百万分之一，称之为 1ppm。

ppm 是一个相对变化量，1ppm 指百万分之一，也就是相对标称频率的变化量。时钟源有两个重要指标，一个是稳定度，一个是准确度。准确度是指与标称值的偏差，稳定度是指随外部因素变化而产生的变化量。

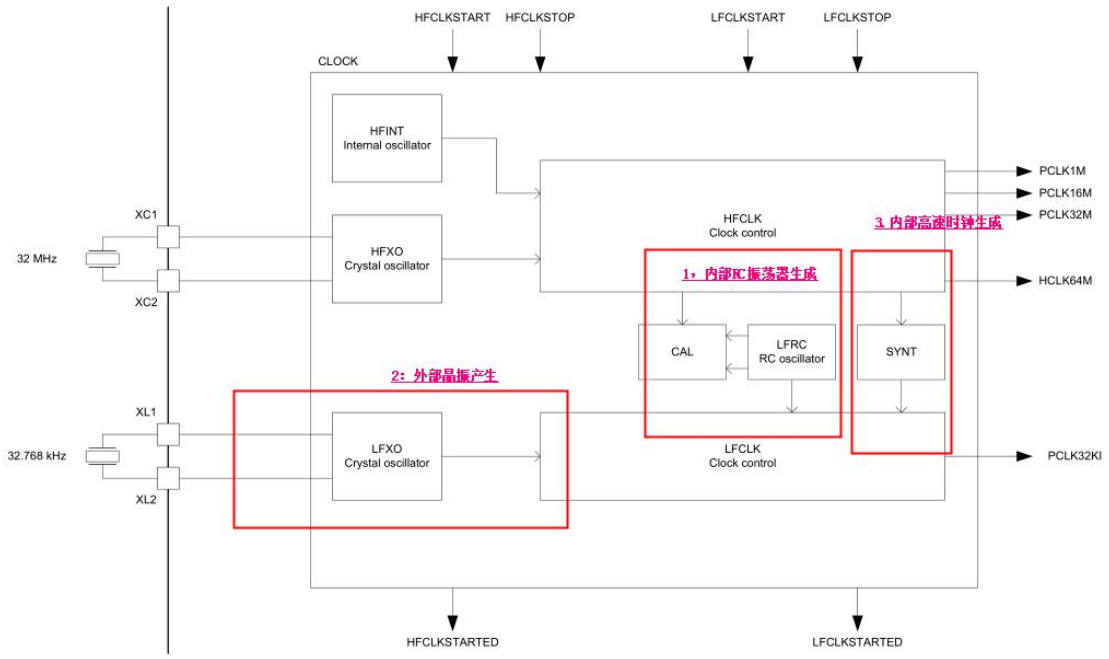
当选择外部晶振的时候，其他另外两个参数 .rc_ctiv 和.rc_temp_ctiv 必须为 0 了。

②：内部 RC 时钟振荡器生成，如果需要使用内部 RC 时钟时，进行校准的时候芯片的 32MHZ 高速时钟必须运行，在 4s 间隔下将增加 6 到 7ua 的的平均电流消耗。同时 RC 功能也消耗一点的电流，因此相比于使用外部晶振，将增加 8 到 10ua 的电流消耗。

多消耗一定电流，节省一定成本，你会选择吗??

③：内部高速时钟合成，这种方式和 RC 时钟振荡器生成类似，需要高速时钟作为生成基础。这种方式所耗费的电流是最大的，功耗是最高的。SYNT 的功耗是要高于 RC 内部振荡器的。

这三种情况，在时钟树结构中有详细的表示，如下图所示。我们要注意一点，这里的时钟仅仅是提供给 RTC0 的时钟，而协议栈是以 RTC0 为同步时钟运行的（和操作系统 ucos 类似），所以说这里设置的时钟称为协议栈时钟。



芯片整体运行的外设等功能使用的是高速主时钟，也就是图上的 PLCK 时钟。有些朋友编写外设程序能正常运行，但是写 BLE 程序的时候却发现没有广播，就是因为协议栈时钟没有而仅仅设置了高速时钟，在 BLE 代码中，默认使用的外部低速晶振作为协议栈时钟。如下配置

```
#ifndef NRF_SDH_CLOCK_LF_SRC
#define     NRF_SDH_CLOCK_LF_SRC    1
#endif
```

第二和第三两个参数 .rc_ctiv 和.rc_temp_ctiv 实际上就是针对 NRF_CLOCK_LF_SRC_RC 模式的，下面讨论下：

.rc_ctiv ：在 1/4 秒单位下的校准时间间隔，为了避免过度的时间漂移，在一个刻度时间间隔下，最大的温度变化允许为 0.5 度

.rc_temp_ctiv：温度变化下的校准间隔。

在 nRF52 下推荐配置 NRF_CLOCK_LF_SRC_RC 为：

rc_ctiv=16 and rc_temp_ctiv=2.

第四个参数.xtal_accuracy ，前面已经谈过，是针对外部晶振时钟的时钟精度配置的，这里不再累述

上面就总结了下协议栈下，需要使用的低速时钟的来源，读者可以根据自己的实际需求进行选择。

3 协议栈默认配置设置 ：

第二个工作**协议栈默认配置设置 ， 也称为**协议栈的初始化了，其实上协议栈的初始化相当的简单，由于整个 nrf52 的协议栈没有开源，而是流出了操作接口。初始化过程实际上是为了相应的蓝牙协议栈事件分配 RAM 空间。代码如下：

```
//协议栈默认配置设置 （默认配置标号--连接配置，ram 起始地址）
err_code = nrf_sdh_ble_default_cfg_set(APP_BLE_CONN_CFG_TAG, &ram_start);
APP_ERROR_CHECK(err_code);
```

这函数第一个形参 APP_BLE_CONN_CFG_TAG 表示连接参数配置标号。

第二个形参 &ram_start 表示协议栈 RAM 起始地址。

这个函数内部，会首先对蓝牙 5.0 协议栈进行一次配置，具体配置我们展开来讨论：

3.1 配置链接数目和角色

第一步配置的是链接数目和角色。这里分为两个部分，一个部分是配置链接数目的设置，一个部分是链接角色的配置，具体代码如下：

```
#if (NRF_SDH_BLE_TOTAL_LINK_COUNT != 0)
    // Configure the connection count.配置链接数目
    memset(&ble_cfg, 0, sizeof(ble_cfg));
    ble_cfg.conn_cfg.conn_cfg_tag = conn_cfg_tag;//设置标号
    ble_cfg.conn_cfg.params.gap_conn_cfg.conn_count =
                                        NRF_SDH_BLE_TOTAL_LINK_COUNT;//总的链接数量
    ble_cfg.conn_cfg.params.gap_conn_cfg.event_length =
                                       NRF_SDH_BLE_GAP_EVENT_LENGTH;//GAP 事件长度
    ret_code = sd_ble_cfg_set(BLE_CONN_CFG_GAP, &ble_cfg, *p_ram_start);//添加协议栈配置
    if (ret_code != NRF_SUCCESS)
    {
      NRF_LOG_ERROR
      ("sd_ble_cfg_set() returned %s when attempting to set BLE_CONN_CFG_GAP.",
                          nrf_strerror_get(ret_code));
    }

// 配置链接角色
    memset(&ble_cfg, 0, sizeof(ble_cfg));
    //从机角色数目
    ble_cfg.gap_cfg.role_count_cfg.periph_role_count=
                                       NRF_SDH_BLE_PERIPHERAL_LINK_COUNT;
#ifndef S112
    //主机角色数目
     ble_cfg.gap_cfg.role_count_cfg.central_role_count=
                                          NRF_SDH_BLE_CENTRAL_LINK_COUNT;
    ble_cfg.gap_cfg.role_count_cfg.central_sec_count=
                                           MIN(NRF_SDH_BLE_CENTRAL_LINK_COUNT,
                                     BLE_GAP_ROLE_COUNT_CENTRAL_SEC_DEFAULT);
#endif //继续添加协议栈配置
    ret_code = sd_ble_cfg_set(BLE_GAP_CFG_ROLE_COUNT, &ble_cfg, *p_ram_start);
                                  if (ret_code != NRF_SUCCESS)
    {
```

```
33.             NRF_LOG_ERROR("sd_ble_cfg_set() returned %s when attempting to set
34.                                     BLE_GAP_CFG_ROLE_COUNT.",
35.             nrf_strerror_get(ret_code));
36.         }
```

首先谈下配置链接数目参数，这个配置分为三个部分：

（1）ble_cfg.conn_cfg.conn_cfg_tag：设置连接标号，可以作为配置连接的标志，可以在创建连接的时候，使得 sd_ble_gap_adv_start() 和 sd_ble_gap_connect()函数调用该配置。

（2）ble_cfg.conn_cfg.params.gap_conn_cfg.conn_count：总的链接数量，这个总的连接数目等于程序中设置的从机和主机的数目和。从机和主机的数目下面配置链接角色的时候谈。

（3） ble_cfg.conn_cfg.params.gap_conn_cfg.event_length：这个称为 GAP 事件长度,gap 事件长度是在蓝牙底层发送数据包时给的处理时间。如果数据 MTU 大的话，可以把数据处理时间设置长点。

然后通过 sd_ble_cfg_set 函数把配置参数添加到协议栈配置中去，如本添加不成功，这打印输出提示，如果成功了，就开始下面的角色配置。

链接角色配置的基本定义：

在这段代码中 role_count_cfg 配置函数定义了 NRF_SDH_BLE_CENTRAL_LINK_COUNT（主机设备数量）和 NRF_SDH_BLE_PERIPHERAL_LINK_COUNT（从机设备数量），这个数量值的是本设备可以提供的服务链接数，所以这两个参数在不同情况下的定义是有区别的：

1：做为从机：比如蓝牙样例，蓝牙串口从机等例子，提供的是从机设备，因此定义：

```
#define   NRF_SDH_BLE_CENTRAL_LINK_COUNT              0
#define   NRF_SDH_BLE_PERIPHERAL_LINK_COUNT           1
```

（1 个可使用外部从机设备进行连接的链路）

2：作为主机：

当作为主机，比如蓝牙串口主机中，设置为：

```
#define   NRF_SDH_BLE_CENTRAL_LINK_COUNT              1
#define   NRF_SDH_BLE_PERIPHERAL_LINK_COUNT           0
```

（1 个可使用中心主机设备进行连接的链路）

或者 1 拖 8 主机实验中，设置为：

```
#define   NRF_SDH_BLE_CENTRAL_LINK_COUNT              8
#define   NRF_SDH_BLE_PERIPHERAL_LINK_COUNT           0
```

（8 个可使用中心主机设备进行连接的链路，也就是连接 8 路从机）

3：做为主从一体机：

```
#define   NRF_SDH_BLE_CENTRAL_LINK_COUNT              2
#define   NRF_SDH_BLE_PERIPHERAL_LINK_COUNT           1
```

（2 个可使用中心主机设备进行连接的链路，也就是做为主机连接 2 路从机），（1 个可使用外部从机设备进行连接的链路，作为从机可以接 1 个主机）

设置完成后，通过 sd_ble_cfg_set 函数把配置角色参数添加到协议栈配置中去。

3.2 配置 MTU 协商值

第二个配置的参数为 MTU 协商的值，最大传输单元（Maximum Transmission Unit，MTU）是指一种通信协议的某一层上面所能通过的最大数据包大小（以字节为单位）。因为协议数据单元的包头和包尾的长度是固定的，MTU 越大，则一个协议数据单元的承载的有效数据就越长，通信效率也越高。MTU 越大，传送相同的用户数据所需的数据包个数也越低。MTU 也不是越大越好，因为 MTU 越大，传送一个数据包的延迟也越大；并且 MTU 越大，数据包中 bit 位发生错误的概率也越大。因此在蓝牙 5.0 协议下是限定了 MTU 的最大值的。早期蓝牙的 4.0 保证一个最新 23 字节的传输包，ATT 的默认 MTU 为 23 个 bytes， 除去 ATT 的 opcode 一个字节以及 ATT 的 handle 2 个字节之后，剩下的 20 个字节便是留给 GATT 的了。这样的空间实在是太小。在蓝牙 5.0 里，对 MTU 的值进行了扩展：

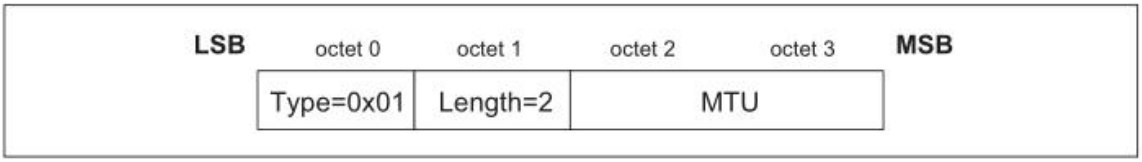


Figure 5.2: MTU Option Format

在 MTU 分配的操作帧，字节 2 和字节 3 可以分配为设置 MTU 的大小，理论上最大可以设置 672 个字节（见 5.0 核心协议栈 p1780 页)。

为了兼容早期蓝牙 4.x 的设置，蓝牙 5.0 里提出 MTU 协商功能，也就是改变传输的 ATT 的 MTU 就行了，大家经过友好的协商，得到双方都能够接受的 MTU 的值。从机代码中，可以设置这个参数的大小，代码如下：

```
#if (NRF_SDH_BLE_GATT_MAX_MTU_SIZE != 23)//如果 MTU 协商不是等于 23 字节
    memset(&ble_cfg, 0x00, sizeof(ble_cfg));
    ble_cfg.conn_cfg.conn_cfg_tag    = conn_cfg_tag;//设置连接编号
    ble_cfg.conn_cfg.params.gatt_conn_cfg.att_mtu = NRF_SDH_BLE_GATT_MAX_MTU_SIZE;//设置 MTU 的字节长度
    //协议栈配置的添加
    ret_code = sd_ble_cfg_set(BLE_CONN_CFG_GATT, &ble_cfg, *p_ram_start);
    if (ret_code != NRF_SUCCESS)
    {
        NRF_LOG_ERROR
       ("sd_ble_cfg_set() returned %s when attempting to set BLE_CONN_CFG_GATT.",
                            nrf_strerror_get(ret_code));
    }
#endif    // NRF_SDH_BLE_GATT_MAX_MTU_SIZE != 23
```

代码中 NRF_SDH_BLE_GATT_MAX_MTU_SIZE 的值就是你需要设置的 MTU 的大小，一般默认是 23 字节。在蓝牙串口历程里设置为了 247 字节。

3.3 设置定制的 UUID 数目

这里设置 UUID 的数目，这个数目指的是私有任务的 UUID 数目，也就是自定义的 128bit 的 UUID 数目。比如你定制了蓝牙串口和蓝牙 LED 灯两个主任务，每个主任务分配 128bit UUID，这时 NRF_SDH_BLE_VS_UUID_COUNT 就定义为 2，这里一定注意是私有任务主任务 UUID 的数目，SIG 定义的公有任务不计入其中。

```
50.     memset(&ble_cfg, 0, sizeof(ble_cfg));
51.     ble_cfg.common_cfg.vs_uuid_cfg.vs_uuid_count = NRF_SDH_BLE_VS_UUID_COUNT;
52.
53.     ret_code = sd_ble_cfg_set(BLE_COMMON_CFG_VS_UUID, &ble_cfg, *p_ram_start);
54.     if (ret_code != NRF_SUCCESS)
55.     {
56.         NRF_LOG_ERROR("sd_ble_cfg_set() returned %s when attempting to set
57.                                 BLE_COMMON_CFG_VS_UUID.", nrf_strerror_get(ret_code));
58.     }
```

如果增加了 UUID 服务后，ram 使用的空间要增加，RAM 的空间也要修改,一个服务大概是 0x10 大小，你 ram 设置里增大对应参数值。

3.4 GATTS 的属性表大小

GATT 定义了以下通用的 Attribute Type 有个大小空间，关于 GATT Profile 定义的详细讲解，我们会单独出一篇文章讲解，这里主要先说下这个 Attribute Type 的 空间大小设置。再配置列表中使用 NRF_SDH_BLE_GATTS_ATTR_TAB_SIZE 表示 GATT 定义了以下通用的 Attribute Type 空间大小。在 GATT 中，由 Primary Service、Secondary Service 和 Characteristic 构成了属于 ATT protocol 中定义的“group of attributes”。而 NRF_SDH_BLE_GATTS_ATTR_TAB_SIZE 就是为了这些设置的数据分配一个空间。

```
59.     memset(&ble_cfg, 0x00, sizeof(ble_cfg));
60.     ble_cfg.gatts_cfg.attr_tab_size.attr_tab_size = NRF_SDH_BLE_GATTS_ATTR_TAB_SIZE;
61.
62.     ret_code = sd_ble_cfg_set(BLE_GATTS_CFG_ATTR_TAB_SIZE, &ble_cfg, *p_ram_start);
63.     if (ret_code != NRF_SUCCESS)
64.     {
65.         NRF_LOG_ERROR("sd_ble_cfg_set() returned %s when attempting to set
66.                                 BLE_GATTS_CFG_ATTR_TAB_SIZE.", nrf_strerror_get(ret_code));
67.     }
```

3.5 使能服务变化特征值

这里的 GATT Service 是 GATT Profile 为自己定义的一个 Service，它的 UUID 为 0x1801。它

存在的意义在于，它包含了一个 Service Changed Characteristic。一旦 server 端的 GATT 属性分布有变（增加了或者移除了一些信息），server 就可以通过这个 Characteristic 的 indication 来通知 client（前提是，client 配置过 server 的《Client Characteristic Configuration》，允许 server 进行 indicate）。Service Changed Characteristic 的格式如下：

Attribute Handle	Attribute Type	Attribute Value			Attribute Permission
0xNNNN	0x2803 – UUID for «Charac-teristic»	Characteristic Properties = 0x20	0xMMMM = Handle of Characteristic Value	0x2A05 – UUID for «Service Changed»	No Authentication, No Authorization

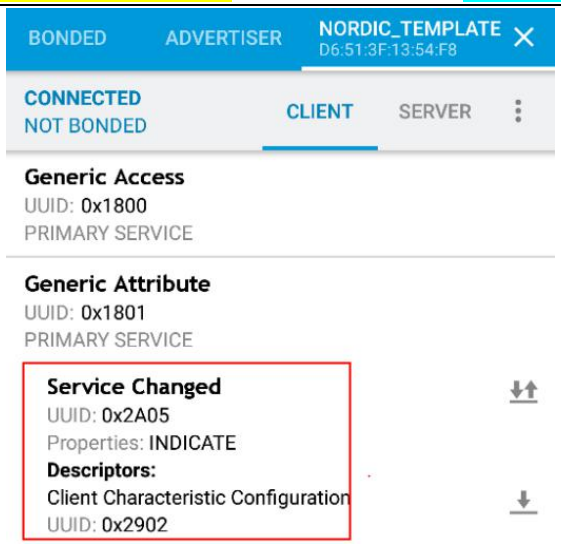
Table 7.2: Service Changed Characteristic declaration

Service Changed Characteristic 具有固定的 properties——0x20，这意味着它只能通过 indication 来告知自身的变化。它的 Value Handle 包含了发生变化的 Attribute 的起止 handle，client 可以对这个范围内的所有 Attribute 重新进行查询。

Attribute Handle	Attribute Type	Attribute Value		Attribute Permission
0xM-MMM	0x2A05 – UUID for «Service Changed»	0xuuuu – Start of Affected Attribute Handle Range	0xuuuu – End of Affected Attribute Handle Range	No Authentication, No Authorization, Not Readable, Not Writable

```
68.     memset(&ble_cfg, 0x00, sizeof(ble_cfg));
69.     ble_cfg.gatts_cfg.service_changed.service_changed = NRF_SDH_BLE_SERVICE_CHANGED;
70.
71.     ret_code = sd_ble_cfg_set(BLE_GATTS_CFG_SERVICE_CHANGED, &ble_cfg, *p_ram_start);
72.     if (ret_code != NRF_SUCCESS)
73.     {
74.         NRF_LOG_ERROR("sd_ble_cfg_set() returned %s when attempting to set
75.                                 BLE_GATTS_CFG_SERVICE_CHANGED.",nrf_strerror_get(ret_code));
76.     }
```

下图是蓝牙样例里配置 Service Changed Characteristic 的值在客户端的显示：



4 使能协议栈：

配置完协议栈的参数后，使用 nrf_sdh_ble_enable 函数使能协议栈，这个函数内部调用了协议栈的 sd 函数：

sd_ble_enable(p_app_ram_start)

这个函数功能是调用初始化了 BLE 栈。在初始化协议栈时，除了 sd_ble_cfg_set 函数之外没有其他的相关函数可以在这个之前调用。

在具有相同主要版本号的软设备之间，特定配置的内存需求不会增加。当应用程序没有对协议栈进行自定义配置*p_app_ram_base 的值时，说明该应用没有在 sd_ble_enable 之前调用 sd_ble_cfg_set。所以在代码中，调用使能初始化函数的时候需要先调用协议栈配置函数 sd_ble_cfg_set。

程序在运行时，集成电路的 RAM 被分为两个区域：软设备 RAM 区域位于 0x20000000 和 APP_RAM_BASE-1 之间，应用程序的 RAM 区域位于 APP_RAM_BASE 和调用堆栈的开始之间。初始化代码如下：

```
1.          // Enable BLE stack.使能协议栈
2.         err_code = nrf_sdh_ble_enable(&ram_start);
3.         APP_ERROR_CHECK(err_code);
```

5 注册蓝牙处理事件：

注册蓝牙处理事件是以提供一个观察者的形式，来实现蓝牙处理派发回调函数。SDK15 的回调派发函数和之前的 SDK 版本有较大区别，我们会专门写一篇文章进行介绍，这里先谈下协议栈处理回调函数。

```
4.   NRF_SDH_BLE_OBSERVER(m_ble_observer, APP_BLE_OBSERVER_PRIO, ble_evt_handler, NULL);
```

NRF_SDH_BLE_OBSERVER 函数内定义了四个形参，这四个形参解释如下：

参数 _name 观察者的名字

参数 _prio 观察者处理事件的优先级。 数字越小，优先级越高。

参数 _handler 蓝牙事件处理

参数 _context 事件处理程序的参数.

观察者主要是指 RTT 的打印观察。这里谈下蓝牙相关的蓝牙处理事件 ble_evt_handler，这个函数可以点击反键观察其源函数：

```
1.  static void ble_evt_handler(ble_evt_t const * p_ble_evt, void * p_context)
2.  {
3.      ret_code_t err_code = NRF_SUCCESS;
4.
5.      switch (p_ble_evt->header.evt_id)
6.      {
7.          case BLE_GAP_EVT_DISCONNECTED://GAP 断开事件
8.              NRF_LOG_INFO("Disconnected.");//打印断开
9.              // LED indication will be changed when advertising starts.
10.             break;
11.
12.         case BLE_GAP_EVT_CONNECTED://GAP 连接事件
13.             NRF_LOG_INFO("Connected.");//打印连接
14.             err_code = bsp_indication_set(BSP_INDICATE_CONNECTED);//点亮连接指示灯
15.             APP_ERROR_CHECK(err_code);
16.             m_conn_handle = p_ble_evt->evt.gap_evt.conn_handle;
17.             err_code = nrf_ble_qwr_conn_handle_assign(&m_qwr, m_conn_handle);//将连接句柄分配给队
    列写模块的给定实例的函数
18.             APP_ERROR_CHECK(err_code);
19.             break;
20.
21.         case BLE_GAP_EVT_PHY_UPDATE_REQUEST://PHY 是升级应答
22.         {
23.             NRF_LOG_DEBUG("PHY update request.");
24.             ble_gap_phys_t const phys =
25.             {
26.                 .rx_phys = BLE_GAP_PHY_AUTO,
27.                 .tx_phys = BLE_GAP_PHY_AUTO,
28.             };
29.             err_code = sd_ble_gap_phy_update(p_ble_evt->evt.gap_evt.conn_handle, &phys);
30.             APP_ERROR_CHECK(err_code);
31.         } break;
32.
33.         case BLE_GATTC_EVT_TIMEOUT://客户端超时
34.             // Disconnect on GATT Client timeout event.
35.             NRF_LOG_DEBUG("GATT Client Timeout.");
36.             err_code = sd_ble_gap_disconnect(p_ble_evt->evt.gattc_evt.conn_handle,
37.                                              BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
38.             APP_ERROR_CHECK(err_code);
39.             break;
40.
41.         case BLE_GATTS_EVT_TIMEOUT://服务端超时
```

```
42.                 // Disconnect on GATT Server timeout event.
43.                 NRF_LOG_DEBUG("GATT Server Timeout.");
44.                 err_code = sd_ble_gap_disconnect(p_ble_evt->evt.gatts_evt.conn_handle,
45.
    BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
46.                 APP_ERROR_CHECK(err_code);
47.                 break;
48.
49.             default:
50.                 // No implementation needed.
51.                 break;
52.         }
53. }
```

蓝牙处理事件函数中主要是根据协议栈触发的蓝牙事件进行对应的动作,主要是进行了以下几个蓝牙事件：

BLE_GAP_EVT_DISCONNECTED//蓝牙端口事件

BLE_GAP_EVT_CONNECTED://GAP 连接事件

BLE_GAP_EVT_PHY_UPDATE_REQUEST://PHY 是升级应答

BLE_GATTS_EVT_TIMEOUT://服务端超时

BLE_GATTC_EVT_TIMEOUT://客户端超时

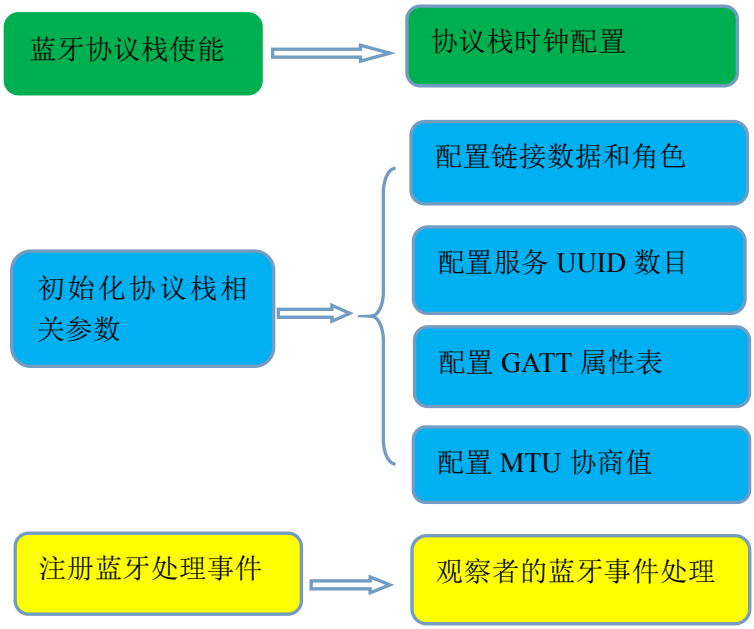
关于事件触发的内容专门写一篇文章进行介绍，这里主要是关系下这些事件下执行什么操作。

BLE_GATTS_EVT_TIMEOUT 服务端超时和 BLE_GATTC_EVT_TIMEOUT 客户端超时这两个操作表示主机和从机发生 GATT 事件超时，这个时候，从机会主动调用 sd_ble_gap_disconnect 函数断开蓝牙连接。

BLE_GAP_EVT_PHY_UPDATE_REQUEST 是蓝牙 5.0 的关于 PHY 的参数升级应答处理。

BLE_GAP_EVT_DISCONNECTED 表示指示蓝牙如果连接断开，从机会判断断开后通过 RTT 打印输出断开，通知观察者。

BLE_GAP_EVT_CONNECTED 表示蓝牙发生连接后，从机会判断连接，并且通过 RTT 打印输出连接和通过 LED 长亮这两种方式通知观察者。



蓝牙协议栈初始化函数的主要工作

6 理论应用：协议栈采用内部 RC

里面讲完了，下面实践一下，我们改下协议栈时钟，采用内部时钟 RC，设置如下

```
#ifndef NRF_SDH_CLOCK_LF_SRC
#define NRF_SDH_CLOCK_LF_SRC 0
#endif

#ifndef NRF_SDH_CLOCK_LF_RC_CTIV
#define NRF_SDH_CLOCK_LF_RC_CTIV 16
#endif

#ifndef NRF_SDH_CLOCK_LF_RC_TEMP_CTIV
#define NRF_SDH_CLOCK_LF_RC_TEMP_CTIV 2
#endif
```

编译后，下载运行，不带低速晶振 32.768Khz 的时候也可以正常运行。