青风带你玩蓝牙 nRF51822 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-nRF51822 开发板

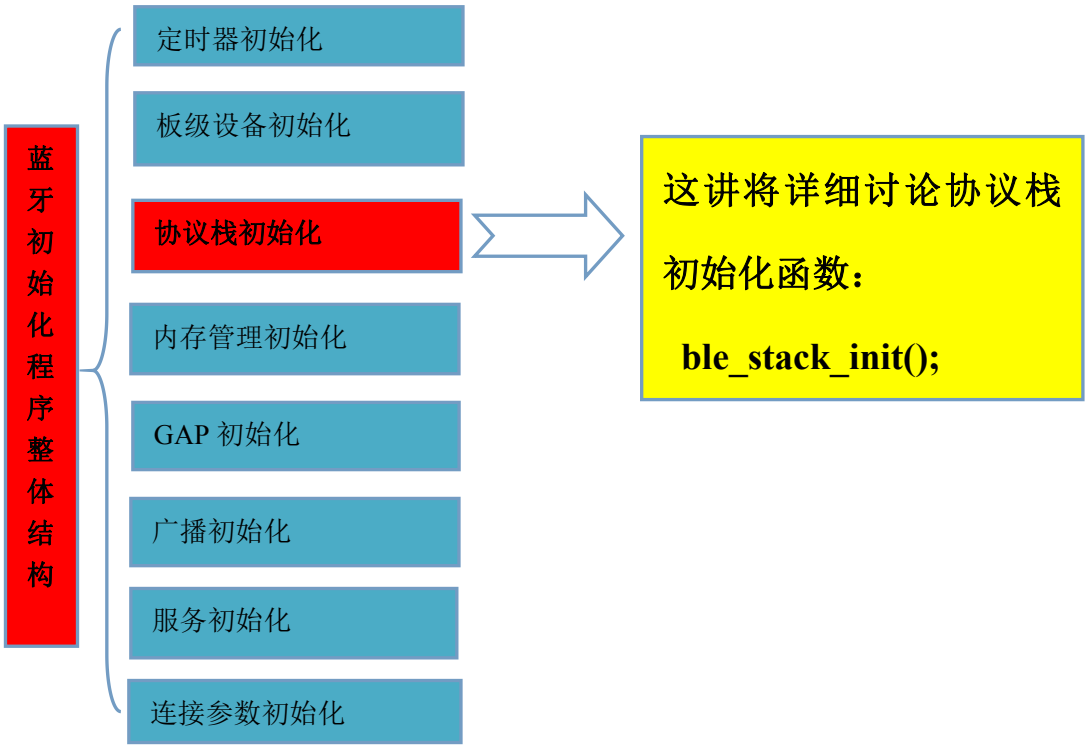
2.7 蓝牙协议初始化详解

对应蓝牙协议栈的初始化一直是大家关注的问题，Nordic 的协议栈的初始化及其调度机制将是本节的详细探讨内容。

并且通过分析基本原理，在匹配的 SDK10.0 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：s110。

```
/**@brief Function for application main entry.
 */
int main(void)
{
    uint32_t err_code;
    bool erase_bonds;

    // Initialize.
    timers_init();//定时器初始化
    buttons_leds_init(&erase_bonds);//按键和LED灯初始化

    ble_stack_init();//蓝牙协议栈初始化
    device_manager_init(erase_bonds);//设备管理初始化
    gap_params_init();//GAP参数初始化
    advertising_init();//广播初始化
    services_init();//服务初始化
    conn_params_init();//更新过程初始化

    // Start execution.
    application_timers_start();//定时器开始计时
    err_code = ble_advertising_start(BLE_ADV_MODE_FAST);//开始广播
    APP_ERROR_CHECK(err_code);

    // Enter main loop.
    for (;;)
    {
        power_manage();
    }
}
```

初始化部分



1：nRF51822 蓝牙协议栈初始化函数结构：

在一个 nrf51822 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)
{
    uint32_t err_code;

    nrf_clock_lf_cfg_t clock_lf_cfg = NRF_CLOCK_LFCLKSRC;

    // Initialize the SoftDevice handler module.初始化系统时钟与相关参数
    SOFTDEVICE_HANDLER_INIT(&clock_lf_cfg, NULL);

    //获取默认启动SoftDevice参数
    ble_enable_params_t ble_enable_params;
    err_code = softdevice_enable_get_default_config(CENTRAL_LINK_COUNT,
                                                    PERIPHERAL_LINK_COUNT,
                                                    &ble_enable_params);
    APP_ERROR_CHECK(err_code);

    //Check the ram settings against the used number of links检测内存设置使用的链接数
    CHECK_RAM_START_ADDR(CENTRAL_LINK_COUNT,PERIPHERAL_LINK_COUNT);

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

    // Register with the SoftDevice handler module for BLE events.使能回调派发函数，蓝牙事件
    err_code = softdevice_ble_evt_handler_set(ble_evt_dispatch);
    APP_ERROR_CHECK(err_code);

    // Register with the SoftDevice handler module for BLE events.系统事件
    err_code = softdevice_sys_evt_handler_set(sys_evt_dispatch);
    APP_ERROR_CHECK(err_code);
}
```

协议栈时钟设置

启动协议栈

回调派发设置

1：协议栈时钟初始化

2：初始化协议栈

3：派发回调函数的设置

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：协议栈系统时钟设置：

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. #define   NRF_CLOCK_LFCLKSRC
02.                     {.source          = NRF_CLOCK_LF_SRC_XTAL,
03.                      .rc_ctiv         = 0,                                \
04.                      .rc_temp_ctiv    = 0,                                \
05.                      .xtal_accuracy = NRF_CLOCK_LF_XTAL_ACCURACY_20_PPM}
```

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

```
06. #define NRF_CLOCK_LF_SRC_RC          (0)                    /**< 内部 RC 时钟 */
07. #define NRF_CLOCK_LF_SRC_XTAL        (1)                    /**< 外部晶振时钟. */
08. #define NRF_CLOCK_LF_SRC_SYNTH       (2)          /**< 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。

```
09. #define NRF_CLOCK_LF_XTAL_ACCURACY_250_PPM (0) /* Default */
10. #define NRF_CLOCK_LF_XTAL_ACCURACY_500_PPM (1)
11. #define NRF_CLOCK_LF_XTAL_ACCURACY_150_PPM (2)
12. #define NRF_CLOCK_LF_XTAL_ACCURACY_100_PPM (3)
13. #define NRF_CLOCK_LF_XTAL_ACCURACY_75_PPM  (4)
14. #define NRF_CLOCK_LF_XTAL_ACCURACY_50_PPM  (5)
15. #define NRF_CLOCK_LF_XTAL_ACCURACY_30_PPM  (6)
16. #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，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??

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 nRF51 下推荐配置 NRF_CLOCK_LF_SRC_RC 为：

rc_ctiv=16 and rc_temp_ctiv=2.

3：高速时钟合成低速时钟方式，这种方式下，协议栈使用 32M 高速晶振合成的低速时钟，由于低速时钟是在休眠模式和连接事件上使用的，因此这种状态下 32M 高速晶振必须一直运行，这样电流消耗是会比前面两种状态有提高的。

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 协议栈的使能：

第二个工作就是协议栈的初始化了，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：

```
1.      err_code = softdevice_enable_get_default_config(CENTRAL_LINK_COUNT,
2.                                                      PERIPHERAL_LINK_COUNT,
3.                                                      &ble_enable_params);
4.      APP_ERROR_CHECK(err_code);
5.
6.      //Check the ram settings against the used number of links 检测内存设置使用的链接数
7.      CHECK_RAM_START_ADDR(CENTRAL_LINK_COUNT,PERIPHERAL_LINK_COUNT);
8.
9.      // Enable BLE stack.使能协议栈
10.     err_code = softdevice_enable(&ble_enable_params);
11.     APP_ERROR_CHECK(err_code);
```

在这段代码中：softdevice_enable_get_default_config 配置函数定义了 CENTRAL_LINK_COUNT （主机设备数量）和 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 CENTRAL_LINK_COUNT 0

#define 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 CENTRAL_LINK_COUNT 1

#define 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 CENTRAL_LINK_COUNT 8

#define 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 CENTRAL_LINK_COUNT 2

#define 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）

然后根据这个设置 CHECK_RAM_START_ADDR 内配置制 RAM 使用的空间。同时在 softdevice_enable 函数中调用 sd_ble_enable 使能协议栈。

4 回调派发函数：

回调派发函数，分两个部分，一个是蓝牙事件派发，一个是系统事件派发。为什么会分两个派发函数？这和这个 nrf51822 协议栈的处理机制有关系，不同与 CC240 等蓝牙芯片带操作系统，nrf 系列蓝牙处理器采用派发方式。当蓝牙协议栈有事件需要处理的时候就会用到蓝牙事件派发。当系统有事件处理的时候就使用到系统事件派发。

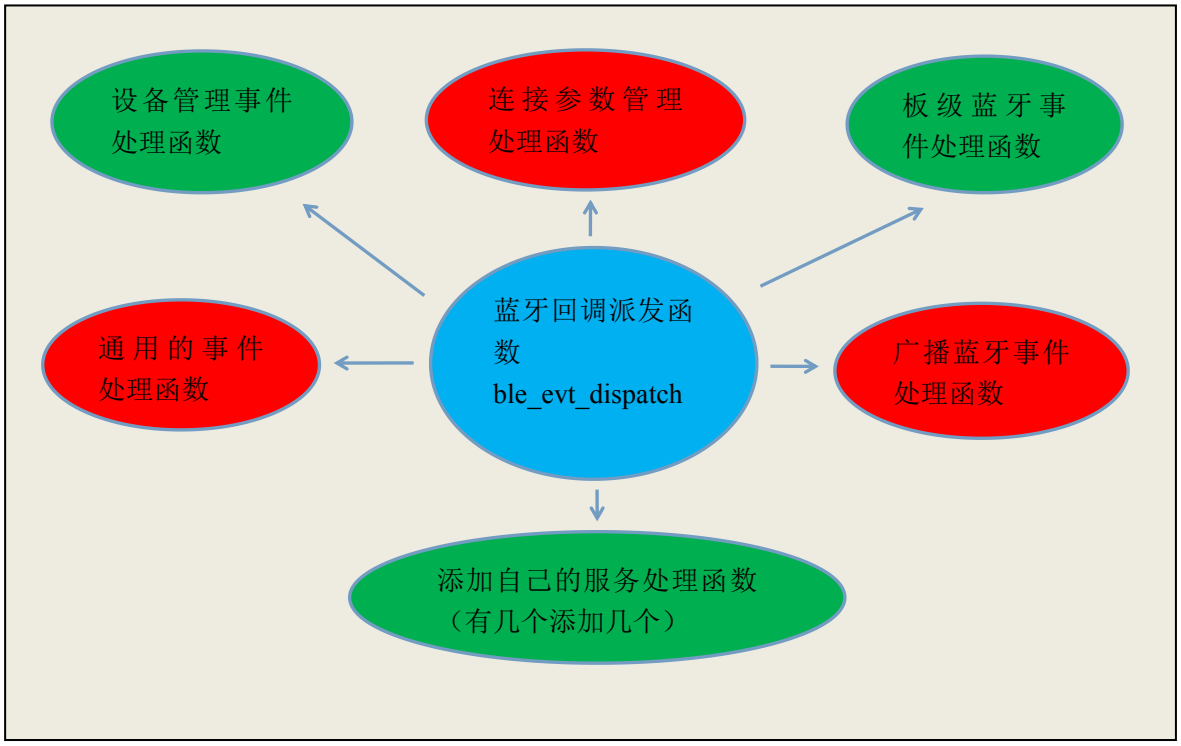
```
1.      // 注册回调派发函数，蓝牙事件
2.      err_code = softdevice_ble_evt_handler_set(ble_evt_dispatch);
3.      APP_ERROR_CHECK(err_code);
4.
5.      // 注册回调派发函数，系统事件
6.      err_code = softdevice_sys_evt_handler_set(sys_evt_dispatch);
7.      APP_ERROR_CHECK(err_code)
```

首先来看下 ble_evt_dispatch 派发函数的程序清单，下面进行了详细注释：

```
8.  static void  ble_evt_dispatch(ble_evt_t * p_ble_evt)
9.  {
```

```
10.     dm_ble_evt_handler(p_ble_evt);//设备管理事件处理函数
11.     ble_conn_params_on_ble_evt(p_ble_evt);//连接参数管理处理函数
12.     bsp_btn_ble_on_ble_evt(p_ble_evt);//板级蓝牙事件处理函数
13.     on_ble_evt(p_ble_evt);//通用的事件处理函数
14.     ble_advertising_on_ble_evt(p_ble_evt);//广播蓝牙事件处理函数
15.     /*YOUR_JOB add calls to _on_ble_evt functions from each service your application is using
16.     ble_xxs_on_ble_evt(&m_xxs, p_ble_evt);添加自己的服务处理函数
17.     ble_yys_on_ble_evt(&m_yys, p_ble_evt);
18.     */
19. }
```

这里要谈谈 NRF52 的蓝牙协议栈处理机制。在任何与 BLE 相关的事件被协议栈上抛上来给应用时，ble_evt_dispatch 蓝牙派发函数就会被调用，从而将事件抛给各个服务函数或处理派发模块，如上图所示。每种处理函数处理的事件，在事件结构体 ble_evt_t 中通过 id 来标注。不同的处理函数只需要处理自己要去做的事件（也就是感兴趣的事件，设计者在程序中编写的需要去处理的事件）。在一个蓝牙事件派发中，上图的红色部分是必须的。绿色部分是可选的，比如说设备管理事件处理函数，在需要操作内部 FLASH 的时候就需要设置。

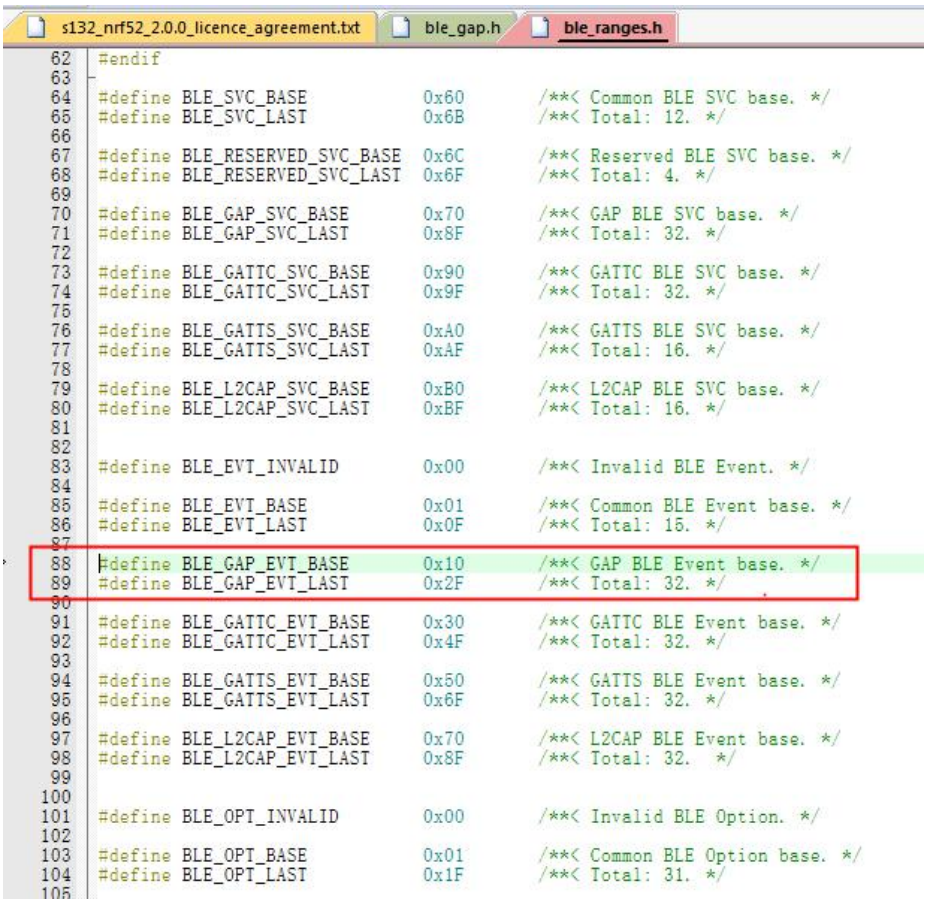


而对于 app 应用来说，就是添加自己的服务处理函数了，你有几个任务就添加几个，如何添加的参考《LED 读写任务》，《按键反馈任务》等详解教程。对应处理函数，我们找出一个出来说明一下，比如下面的连接参数管理处理函数：

```
17. void ble_conn_params_on_ble_evt(ble_evt_t * p_ble_evt)
18. {
19.     switch (p_ble_evt->header.evt_id)
20.     {
21.         case BLE_GAP_EVT_CONNECTED:
22.             on_connect(p_ble_evt);
```

```
23.                 break;
24.
25.             case BLE_GAP_EVT_DISCONNECTED:
26.                 on_disconnect(p_ble_evt);
27.                 break;
28.
29.             case BLE_GATTS_EVT_WRITE:
30.                 on_write(p_ble_evt);
31.                 break;
32.
33.             case BLE_GAP_EVT_CONN_PARAM_UPDATE:
34.                 on_conn_params_update(p_ble_evt);
35.                 break;
36.
37.             default:
38.                 // No implementation needed.
39.                 break;
40.         }
41. }
```

代码中加黑的代码就表示为 evt_id 传递的事件，那么这些事件是如何定义的了？协议栈抛给程序的是什么数据？下面就详细的讨论下，在官方 ble_ranges.h 文件中，**定义了一组事件 ID 的类**，为什么说是**类**？实际就是分群的意思。比如红框里的 BLE_GAP_EVT_BASE 为 GAP 蓝牙事件基础地址，到最后的 BLE_GAP_EVT_LAST 地址，一个是 32 个字节。这个字节区间是不定的，有的群为 16 个字节，有的为 8 个字节。



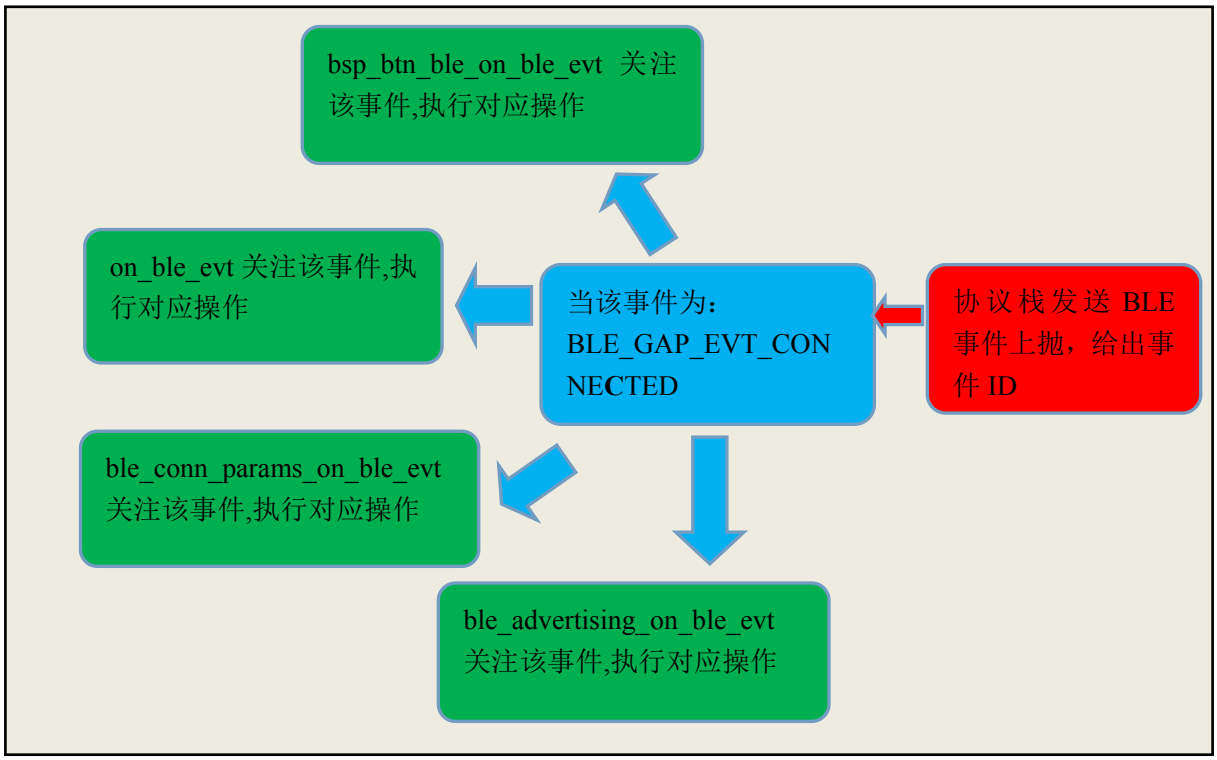
```
62  #endif
63
64  #define BLE_SVC_BASE            0x60        /**< Common BLE SVC base. */
65  #define BLE_SVC_LAST            0x6B        /**< Total: 12. */
66
67  #define BLE_RESERVED_SVC_BASE   0x6C        /**< Reserved BLE SVC base. */
68  #define BLE_RESERVED_SVC_LAST   0x6F        /**< Total: 4. */
69
70  #define BLE_GAP_SVC_BASE        0x70        /**< GAP BLE SVC base. */
71  #define BLE_GAP_SVC_LAST        0x8F        /**< Total: 32. */
72
73  #define BLE_GATTC_SVC_BASE      0x90        /**< GATTC BLE SVC base. */
74  #define BLE_GATTC_SVC_LAST      0x9F        /**< Total: 32. */
75
76  #define BLE_GATTS_SVC_BASE      0xA0        /**< GATTS BLE SVC base. */
77  #define BLE_GATTS_SVC_LAST      0xAF        /**< Total: 16. */
78
79  #define BLE_L2CAP_SVC_BASE      0xB0        /**< L2CAP BLE SVC base. */
80  #define BLE_L2CAP_SVC_LAST      0xBF        /**< Total: 16. */
81
82
83  #define BLE_EVT_INVALID         0x00        /**< Invalid BLE Event. */
84
85  #define BLE_EVT_BASE            0x01        /**< Common BLE Event base. */
86  #define BLE_EVT_LAST            0x0F        /**< Total: 15. */
87
88  #define BLE_GAP_EVT_BASE        0x10        /**< GAP BLE Event base. */
89  #define BLE_GAP_EVT_LAST        0x2F        /**< Total: 32. */
90
91  #define BLE_GATTC_EVT_BASE      0x30        /**< GATTC BLE Event base. */
92  #define BLE_GATTC_EVT_LAST      0x4F        /**< Total: 32. */
93
94  #define BLE_GATTS_EVT_BASE      0x50        /**< GATTS BLE Event base. */
95  #define BLE_GATTS_EVT_LAST      0x6F        /**< Total: 32. */
96
97  #define BLE_L2CAP_EVT_BASE      0x70        /**< L2CAP BLE Event base. */
98  #define BLE_L2CAP_EVT_LAST      0x8F        /**< Total: 32.  */
99
100
101 #define BLE_OPT_INVALID         0x00        /**< Invalid BLE Option. */
102
103 #define BLE_OPT_BASE            0x01        /**< Common BLE Option base. */
104 #define BLE_OPT_LAST            0x1F        /**< Total: 31. */
105
```

BLE_GAP_EVT 事件类中的内容采用结构体方式定义，该定义在文件 ble_gap.h 中给出，如下代码所示，结构体中第一了事件使用 BLE_GAP_EVT_BASE 作为起始 ID,后面依次分配，到最后的 BLE_GAP_EVT_LAST 为结束地址。大家观察结构体总过定义了 16 个时间 ID，那么一个时间 ID 是 2 个字节区间，正好 16 位，也就是说 ID 为 16 位形式。

```
42. enum BLE_GAP_EVTS
43. {
44.     BLE_GAP_EVT_CONNECTED    = BLE_GAP_EVT_BASE,
45.     BLE_GAP_EVT_DISCONNECTED,
46.     BLE_GAP_EVT_CONN_PARAM_UPDATE,
47.     BLE_GAP_EVT_SEC_PARAMS_REQUEST,
48.     BLE_GAP_EVT_SEC_INFO_REQUEST,
49.     BLE_GAP_EVT_PASSKEY_DISPLAY,
50.     BLE_GAP_EVT_KEY_PRESSED,
51.     BLE_GAP_EVT_AUTH_KEY_REQUEST,
52.     BLE_GAP_EVT_LESC_DHKEY_REQUEST,
53.     BLE_GAP_EVT_AUTH_STATUS,
54.     BLE_GAP_EVT_CONN_SEC_UPDATE,
55.     BLE_GAP_EVT_TIMEOUT,
56.     BLE_GAP_EVT_RSSI_CHANGED,
57.     BLE_GAP_EVT_ADV_REPORT,
58.     BLE_GAP_EVT_CONN_PARAM_UPDATE_REQUEST,
59.     BLE_GAP_EVT_SCAN_REQ_REPORT,
```

60. };

通过上面的讲解读者知道了事件如何定义，协议栈以何种形式给我们应用程序识别了。那么下面就讲讲应用程序需要做什么工作了。之前谈过了蓝牙协议栈处理机制的处理机制，事件 ID 抛过来，举个例子，如下图所示，如果这个事件是连接事件 BLE_GAP_EVT_CONNECTED 。那么在派发函数中的采用轮询的方式，看各个事件处理函数是否关注这个事件 ID，如果关注的话，也就是在程序中定义事件处理函数需要执行对应的操作。



比如 ble_conn_params_on_ble_evt 关注了 BLE_GAP_EVT_CONNECTED 事件，那么就执行连接操作，如下图所示：

```
264 }
265
266
267 void ble_conn_params_on_ble_evt(ble_evt_t * p_ble_evt)
268 {
269     switch (p_ble_evt->header.evt_id)
270     {
271         case BLE_GAP_EVT_CONNECTED:
272             on_connect(p_ble_evt);
273             break;
274
275         case BLE_GAP_EVT_DISCONNECTED:
276             on_disconnect(p_ble_evt);
277             break;
278
279         case BLE_GATTS_EVT_WRITE:
280             on_write(p_ble_evt);
281             break;
282
283         case BLE_GAP_EVT_CONN_PARAM_UPDATE:
284             on_conn_params_update(p_ble_evt);
285             break;
286
287         default:
288             // No implementation needed.
289             break;
290     }
291 }
292
```

执行连接操作

上面就讲完了蓝牙事件派发，下面再来说下系统事件派发，系统事件派发可以称为 soc 片上事件派发,专门针对芯片硬件处理事件。如下代码所示：

```
61. static void sys_evt_dispatch(uint32_t sys_evt)
62. {
63.     pstorage_sys_event_handler(sys_evt);//系统内存事件
64.     ble_advertising_on_sys_evt(sys_evt);//系统广播事件
65. }
```

其中 pstorage_sys_event_handler 将在后面《蓝牙内部 FLASH 存储》中详细说明。系统派发一样是通过 ID 还处理事件。通过协议栈函数 sd_evt_get 获取事件 ID，然后由处理事件函数进行相关的处理，如下图所示：

```
1116  void pstorage_sys_event_handler(uint32_t sys_evt)
1117  {
1118      if (m_state != STATE_IDLE && m_state != STATE_ERROR)
1119      {
1120          switch (sys_evt)
1121          {
1122              case NRF_EVT_FLASH_OPERATION_SUCCESS:
1123                  flash_operation_success_run();
1124                  break;
1125
1126              case NRF_EVT_FLASH_OPERATION_ERROR:
1127                  if (!(m_flags & MASK_FLASH_API_ERR_BUSY))
1128                  {
1129                      flash_operation_failure_run();
1130                  }
1131                  else
1132                  {
1133                      // As our last flash operation request was rejected by
1134                      // request by doing same code execution path as for fl
1135                      // event. This will promote code reuse in the implemen
1136                      flash_operation_success_run();
1137                  }
1138                  break;
1139
```

对应事件

执行的操作

在官方文件 nrf_soc.h 文件中定义了相关的事件:

```
258  enum NRF_RADIO_REQUEST_TYPE
259  {
260      NRF_RADIO_REQ_TYPE_EARLIEST,
261      NRF_RADIO_REQ_TYPE_NORMAL
262  };
263
264  /**@brief SoC Events. */
265  enum NRF_SOC_EVTS
266  {
267      NRF_EVT_HFCLKSTARTED,
268      NRF_EVT_POWER_FAILURE_WARNING,
269      NRF_EVT_FLASH_OPERATION_SUCCESS,
270      NRF_EVT_FLASH_OPERATION_ERROR,
271      NRF_EVT_RADIO_BLOCKED,
272      NRF_EVT_RADIO_CANCELED,
273      NRF_EVT_RADIO_SIGNAL_CALLBACK_INVALID_RETURN,
274      NRF_EVT_RADIO_SESSION_IDLE,
275      NRF_EVT_RADIO_SESSION_CLOSED,
276      NRF_EVT_NUMBER_OF_EVTS
277  };
278
279  /**@} */
280
```

5 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

```
66. #define   NRF_CLOCK_LFCLKSRC
67.                   {.source        = NRF_CLOCK_LF_SRC_RC,
68.                    .rc_ctiv       = 16,                                \
69.                    .rc_temp_ctiv  = 2,                                 \
70.                    .xtal_accuracy = NRF_CLOCK_LF_XTAL_ACCURACY_20_PPM}
```

编译后，下载运行。