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

2.8 蓝牙防丢器详解

蓝牙（Smart Bluetooth）防丢器，是采用蓝牙 4.x 技术专门为智能手机设计的防丢器。其工作原理主要是通过距离变化来判断物品是否还控制在你的安全范围。主要适用于手机、钱包、钥匙、行李等贵重物品的防丢，也可用于防止儿童或宠物的走失

蓝牙技术本身是为了连接（比如蓝牙耳机，蓝牙键盘等）而并非防丢，所谓防丢，是根据蓝牙信号的强弱来判断距离.本讲将结合实例详细讲解 nrf52832 蓝牙放弃器的设置思路。

2.8.1 蓝牙防丢器原理分析：

当前主流的智能防丢器主要有两种，一种是设备防丢，一种是人防丢;用于设备防丢的防丢器形式比较单一，多位防丢贴片或防丢挂件，而用于人防丢的防丢器形式就比较多了，挂件、手表、项链、鞋等等，形式五花八门。

那么在 nrf51822 中实现的原理是什么了？简单来说，我实现通过距离变化来判断物品是否还控制在你的安全范围内的具有核心功能的防丢器，同时具有双向报警功能，电池电量显示功能的设计来实现。下图为市面上常见的防丢器构造：

本例中蓝牙防丢器的服务建立将是重点分析对象，本例中建立了 5 个服务：

1.TX Power Service - 发射功率服务

该服务可以调节蓝牙的发射功率，从而可以影响设备和手机蓝牙直接的检测及其通信距离。设备端需要通过主机控制接口 HCI 来获得发射功率参数，并以 read 属性提供给 master。这个服务可以设置发送功率大小，直接作用于 APP 主机分析出的 rssi 的大小。

2. Immediate Alert Service - 即时报警服务

该服务可以使用触发报警服务，服务为 write 属性，供 master 主机写告警级别。主机 APP 发送立即报警参数后，从机设备端会收到 write 的回调，其根据告警级别进行相应告警。这个功能相当于防丢器中的寻物功能，可以通过手机找到设备。

3.Link Loss Service - 连接丢失服务

当连接丢失之后（可能是电池没电或者离开 APP 太远）都会导致连接丢失事件，丢失后会以通知的方式发送到手机 APP，手机根据该事件作出响应。ble_app_proximity 的程序中使用通知的方式显示出来。write/read 属性，供 master 设置链路断开情况下默认的告警级别。设备的 RSSI 通过手机的接收端的接口来获得，并不需要设备端提供 service。这个功能相当于防丢器中的防丢功能，当物体链接丢失后会进行报警。

4.Immediate Alert Service client - locator role of the Find Me profile 定位器服务

该服务可以通过定位，通过设备定位，通知手机 APP 设备，手机进行报警。

5.Battery Service - 电池服务

通过 AD 采样电池电压，发送到手机 APP，在主机 APP 上显示电量。

2.8.2 蓝牙防丢器程序解析：

蓝牙防丢器的主程序代码解析如下，关于 nrf5x 系列处理器的主函数流程分析，在前面的篇章里已经详细讲过，这里就不累述了，我们主要是来谈下不同的地方。

```
01. int main(void)
02. {
03.     uint32_t err_code;
04.     bool erase_bonds;
05.     // Initialize.
06.     err_code = NRF_LOG_INIT();
07.     APP_ERROR_CHECK(err_code);
08.     timers_init();//初始化定时器
09.     buttons_leds_init(&erase_bonds);//设备初始化
10.     ble_stack_init();//初始化协议栈
11.     adc_configure();//ADC 配置 用于采集电池电量
12.     peer_manager_init(erase_bonds);//匹配管理初始化，也就是之前的设备管理初始化
13.     if (erase_bonds == true)
14.     {
15.         NRF_LOG_DEBUG("Bonds erased!\r\n");
16.     }
17.     gap_params_init();//GAP 初始化
18.     advertising_init();//广播初始化
19.     db_discovery_init();//数据发现初始化
20.     services_init();//服务初始化
21.     conn_params_init();//链接参数更新初始化
22.     // Start execution.
23.     advertising_start();//开始广播
24.     // Enter main loop.
25.     for (;;)
26.     {
27.         power_manage();
28.     }
29. }
```

蓝牙防丢器的目标就是为了建立上面谈到的 5 个服务。所以首先应该变动的就是服务初始化中的子服务初始化和协议栈初始化中的派发函数，现在先来谈谈这两个地方如何设置的，首先是服务初始化中：建立 5 个服务初始化函数

```
30. static void services_init(void)
31. {
32.     tps_init();//发射功率服务
33.     ias_init();//立即报警服务
34.     lls_init();//链路丢失服务
35.     bas_init();//电池服务
```

```
    ias_client_init();//立即报警服务客户端
}
```

1.发射功率服务，发送功率的设置和应用在专题**《青风带你学蓝牙：蓝牙发射功率设置》**详细谈过，设置从机的发射功率，这里就不展开了。

```
static void tps_init(void)
{
    uint32_t           err_code;
    ble_tps_init_t tps_init_obj;

    memset(&tps_init_obj, 0, sizeof(tps_init_obj));
    tps_init_obj.initial_tx_power_level = TX_POWER_LEVEL;

BLE_GAP_CONN_SEC_MODE_SET_ENC_NO_MITM(&tps_init_obj.tps_attr_md.read_perm);
    BLE_GAP_CONN_SEC_MODE_SET_NO_ACCESS(&tps_init_obj.tps_attr_md.write_perm);

    err_code = ble_tps_init(&m_tps, &tps_init_obj);//设置发送功率的
    APP_ERROR_CHECK(err_code);
}
```

2.电池服务，电池服务的设置和应用在专题**《青风带你学蓝牙：蓝牙电池服务的应用》**详细谈过，如果采用定时器进行电池电量采集，这里同时配置了 adc_configure()进行电池采集，这里也不展开了。

```
static void bas_init(void)
{
    uint32_t           err_code;
    ble_bas_init_t bas_init_obj;

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

    bas_init_obj.evt_handler                = on_bas_evt;
    bas_init_obj.support_notification = true;
    bas_init_obj.p_report_ref               = NULL;
    bas_init_obj.initial_batt_level     = 100;

    .....................................
    .....................................
    err_code = ble_bas_init(&m_bas, &bas_init_obj);
    APP_ERROR_CHECK(err_code);
}
```

2.8.2.1 立即报警的服务（从机报警）：

即时报警服务用于设置报警水平来产生一个立即报警，可以用于设置报警等级来进行寻物，函数如下所示：使用 ble_ias_init 函数初始化理解报警，其中函数第二个形参 ias_init_obj 中的事件参数 evt_handler 作为触发回调事件。

```
68. static void ias_init(void)
69. {
70.     uint32_t           err_code;
71.     ble_ias_init_t ias_init_obj;
72.
73.     memset(&ias_init_obj, 0, sizeof(ias_init_obj));
74.     ias_init_obj.evt_handler = on_ias_evt;
75.
76.     err_code = ble_ias_init(&m_ias, &ias_init_obj);//立即报警服务
77.     APP_ERROR_CHECK(err_code);
78. }
```

进入到源函数 ble_ias_init 的源函数中，把第二个形参的事件赋值给第一个形参的事件 evt,上面的初始化函数中设置的第一个形参为全局变量&m_ias 作为指示 my 立即报警服务。Ias 服务初始化函数在以前做任务初始化的工程中已经详细和大家描述过，作为一个 SIG 的公共任务，初始化过程如下代码所示

```
79. uint32_t ble_ias_init(ble_ias_t * p_ias, const ble_ias_init_t * p_ias_init)
80. {
81.     uint32_t    err_code;
82.     ble_uuid_t ble_uuid;
83.
84.     // Initialize service structure
85.     if (p_ias_init->evt_handler == NULL)
86.     {
87.         return NRF_ERROR_INVALID_PARAM;
88.     }
89.     p_ias->evt_handler = p_ias_init->evt_handler;
90.
91.     // 添加服务，设置服务 UUID，UUID 类型
92.     BLE_UUID_BLE_ASSIGN(ble_uuid, BLE_UUID_IMMEDIATE_ALERT_SERVICE);
93.     err_code = sd_ble_gatts_service_add(BLE_GATTS_SRVC_TYPE_PRIMARY,
94.                                         &ble_uuid,
95.                                         &p_ias->service_handle);
96.     if (err_code != NRF_SUCCESS)
97.     {
98.         return err_code;
99.     }
```

```
100.
101.       // 添加报警特征值
102.       return alert_level_char_add(p_ias);
103. }
```

alert_level_char_add 报警特征值的添加需要严格遵循 SIG 提供的 IMMEDIATE ALERT SERVICE 服务手册，具体资料见我们资料文件夹，在手册中规定了服务类型，如下表所示，为无返回的写 特征，同时在特征值属性中必须设置报警水平。

	Broadcast	Read	Write without Response	Write	Notify	Indicate	Signed Write	Reliable Write	Writable Auxiliaries
Alert Level	X	X	M	X	X	X	X	X	X

Table 3.2: Characteristic properties

关于报警水平的表示，手册表述如下，也就是分为三个水平。1：没有报警等级。2：中等报警等级。3：高报警等级

When the Alert Level characteristic is written the device shall start alerting to the written alert level.

If the written alert level is “No Alert," no alerting shall be done on this device.

If the written alert level is “Mild Alert," the device shall alert.

If the written alert level is “High Alert," the device shall alert in the strongest possible way.

所以在特征属性中，需要设置初始化的报警水平，代码具体如下所示：

```
104. static uint32_t alert_level_char_add(ble_ias_t * p_ias)
105. {
106.       .....................................
107.
108.       char_md.char_props.write_wo_resp = 1;//写无回应
109.       char_md.p_char_user_desc             = NULL;
110.      char_md.p_char_pf                    = NULL;
111.      char_md.p_user_desc_md               = NULL;
112.      char_md.p_cccd_md                    = NULL;
113.      char_md.p_sccd_md                    = NULL;
114.
115.    ..............................
116.    ..............................
117.      memset(&attr_char_value, 0, sizeof(attr_char_value));
118.      initial_alert_level = INITIAL_ALERT_LEVEL;//无报警
119.
120.       attr_char_value.p_uuid       = &ble_uuid;
```

```
121.      attr_char_value.p_attr_md = &attr_md;
122.      attr_char_value.init_len    = sizeof (uint8_t);
123.      attr_char_value.init_offs = 0;
124.      attr_char_value.max_len     = sizeof (uint8_t);
125.      attr_char_value.p_value     = &initial_alert_level;//特征值属性设置的报警水平值
126.
127.      return sd_ble_gatts_characteristic_add(p_ias->service_handle,
128.                                             &char_md,
129.                                             &attr_char_value,
130.                                             &p_ias->alert_level_handles);
131. }
```

接着谈下 on_ias_evt 触发事件，触发回调事件是指当 ias 立即报警的派发函数中，出现了对应事件，通过关心这个事件反过来触发回调操作。关于派发函数的内容详见参考《青风带你学蓝牙：协议栈初始化及派发机制》这篇文章。这里我们首先找到关心 ias 的派发函数，代码如下，

```
132. void ble_ias_on_ble_evt(ble_ias_t * p_ias, ble_evt_t * p_ble_evt)
133. {
134.      switch (p_ble_evt->header.evt_id)
135.      {
136.          case BLE_GAP_EVT_CONNECTED:
137.              on_connect(p_ias, p_ble_evt);
138.              break;
139.
140.          case BLE_GATTS_EVT_WRITE:
141.              on_write(p_ias, p_ble_evt);
142.              break;
143.
144.          default:
145.              // No implementation needed.
146.              break;
147.      }
148. }
```

回调派发里关心的事件 evt_id 比较简单，只关心了连接事件 BLE_GAP_EVT_CONNECTED 和写事件 BLE_GATTS_EVT_WRITE，其中连接事件下执行操作 on_connect(p_ias, p_ble_evt)，这个函数里没有触发 evt_type，只在写事件 BLE_GATTS_EVT_WRITE 触发了 evt_type。

```
149. static void on_write(ble_ias_t * p_ias, ble_evt_t * p_ble_evt)
150. {
151.      ble_gatts_evt_write_t * p_evt_write = &p_ble_evt->evt.gatts_evt.params.write;
152.
153.      if ((p_evt_write->handle == p_ias->alert_level_handles.value_handle) && (p_evt_write->len ==
     1))
154.      {
155.          // 报警水平写，触发应用事件回调
```

```
156.            ble_ias_evt_t evt;
157.            evt.evt_type = BLE_IAS_EVT_ALERT_LEVEL_UPDATED;
158.            evt.params.alert_level = p_evt_write->data[0];
159.
160.            p_ias->evt_handler(p_ias, &evt);
161.        }
162. }
```

这时候终于可以回答前面的 on_ias_evt 触发回调操作什么时候触发了，当这个操作关心的事件类型一旦发生，我们就执行某个对应操作。那么这个回调函数里关心的什么事件？也就是上面派发中的写事件触发的 BLE_IAS_EVT_ALERT_LEVEL_UPDATED，很简单，只关心这个事件，其他的都没管，代码如下：

```
163. static void on_ias_evt(ble_ias_t * p_ias, ble_ias_evt_t * p_evt)
164. {
165.     switch (p_evt->evt_type)
166.     {
167.         case BLE_IAS_EVT_ALERT_LEVEL_UPDATED:
168.             alert_signal(p_evt->params.alert_level);//更新报警水平
169.             break;//BLE_IAS_EVT_ALERT_LEVEL_UPDATED
170.
171.         default:
172.             // No implementation needed.
173.             break;
174.     }
175. }
```

当触发了事件后，我们就执行 **alert_signal 函数，这个函数做什么用的？设置报警水平的下的操作。根据在** on_write 写函数中 evt.params.alert_level = p_evt_write->data[0]，也就是写入的报警水平的值来设置触发对应的报警操作。

```
176. static void alert_signal(uint8_t alert_level)
177. {
178.     uint32_t err_code;
179.     switch (alert_level)
180.     { //如果写入的是没有报警，执行下面操作
181.         case BLE_CHAR_ALERT_LEVEL_NO_ALERT:
182.             err_code = bsp_indication_set(BSP_INDICATE_ALERT_OFF);
183.             APP_ERROR_CHECK(err_code);
184.             break;//BLE_CHAR_ALERT_LEVEL_NO_ALERT
185.       //如果写入的是中等报警，执行下面操作
186.         case BLE_CHAR_ALERT_LEVEL_MILD_ALERT:
187.             err_code = bsp_indication_set(BSP_INDICATE_ALERT_0);
188.             APP_ERROR_CHECK(err_code);
189.             break;//BLE_CHAR_ALERT_LEVEL_MILD_ALERT
190.       //如果写入的是强等级报警，执行下面操作
191.         case BLE_CHAR_ALERT_LEVEL_HIGH_ALERT:
```

```
192.                 err_code = bsp_indication_set(BSP_INDICATE_ALERT_3);
193.                 APP_ERROR_CHECK(err_code);
194.                 break;//BLE_CHAR_ALERT_LEVEL_HIGH_ALERT
195.
196.             default:
197.                 // No implementation needed.
198.                 break;
199.     }
200. }
```

按照配置设置指标所需的设备状态，所谓的设备状态，开发板上主要是就是用 LED 进行指示：

```
201. uint32_t bsp_indication_set(bsp_indication_t indicate)
202. {
203.     uint32_t err_code = NRF_SUCCESS;
204.
205. #if   LEDS_NUMBER > 0 && !(defined BSP_SIMPLE)
206.
207.     if (m_indication_type & BSP_INIT_LED)
208.     {
209.         err_code = bsp_led_indication(indicate);
210.     }
211.
212. #endif // LEDS_NUMBER > 0 && !(defined BSP_SIMPLE)
213.     return err_code;
214. }
```

那么我们详细解读下设备指示函数，设备指示函数在 bps.c 这个文件中，定义了蓝牙开发板上的相关状态，包括广播，断开等等状态下，设备的指示，主要是用的 LED 指示。这里面我们先不说其他的，只关心下 alert_signal 函数中设置不同报警等级后执行的操作，代码如下所示：

```
215. static uint32_t bsp_led_indication(bsp_indication_t indicate)
216. {
217.     uint32_t err_code     = NRF_SUCCESS;
218.     uint32_t next_delay = 0;
219.
220.     switch (indicate)
221.     {
222.       . . . . . .
223.       . . . . . .
224.         case BSP_INDICATE_ALERT_0:
225.         case BSP_INDICATE_ALERT_1:
226.         case BSP_INDICATE_ALERT_2:
```

```
227.            case BSP_INDICATE_ALERT_3:
228.            case BSP_INDICATE_ALERT_OFF:
229.                err_code      = app_timer_stop(m_alert_timer_id);
230.                next_delay = (uint32_t)BSP_INDICATE_ALERT_OFF - (uint32_t)indicate;
231.
232.                // a little trick to find out that if it did not fall through ALERT_OFF
233.                if (next_delay && (err_code == NRF_SUCCESS))
234.                {
235.                    m_alert_mask = ALERT_LED_MASK;
236.                    if (next_delay > 1)
237.                    {
238.                err_code = app_timer_start(m_alert_timer_id, BSP_MS_TO_TICK(
239.                                        (next_delay * ALERT_INTERVAL)), NULL);
240.                    }
241.                    LEDS_ON(m_alert_mask);
242.                }
243.                else
244.                {
245.                    LEDS_OFF(m_alert_mask);
246.                    m_alert_mask = 0;
247.                }
248.                break;
```

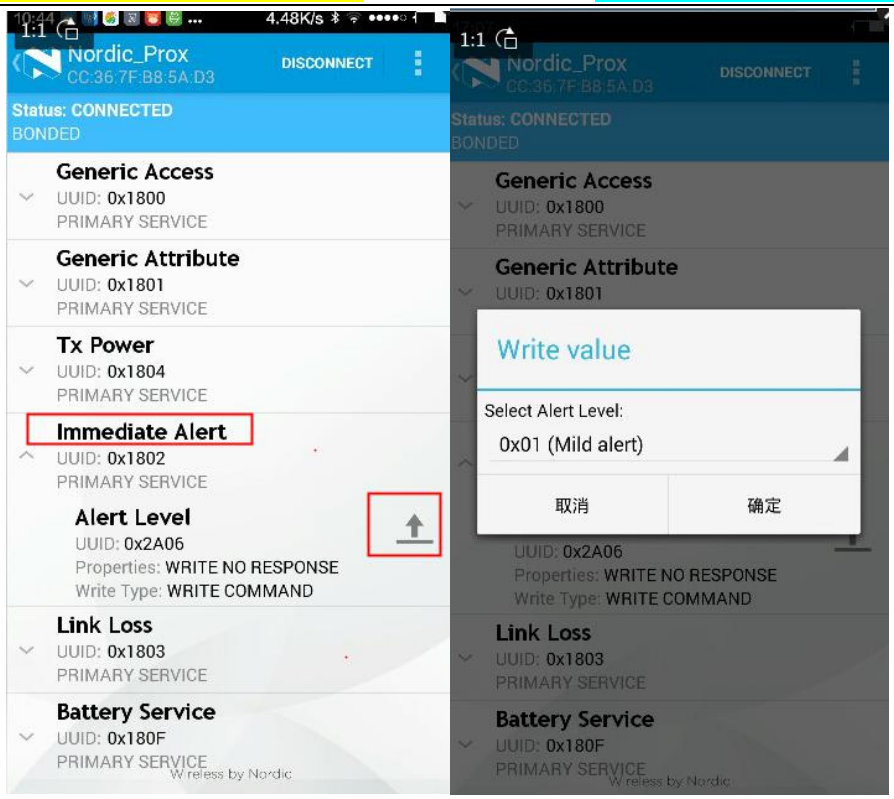
详细说下，首先如果前面设置无报警，那么执行 BSP_INDICATE_ALERT_OFF，那么 next_delay 为 0，所以执行 LEDS_OFF(m_alert_mask)操作，也就是关掉报警灯 m_alert_mask，这个时候报警的 LED 灯是熄灭的。

如果前面设置为中等报警，那么执行 BSP_INDICATE_ALERT_0，程序中 case BSP_INDICATE_ALERT_0 这个位置没有任何程序，也没有 breack，也就是不会跳出执行，那么继续向下运行，那么执行到 next_delay 时，next_delay 为 5，更加结构体的排序决定：

```
BSP_INDICATE_FATAL_ERROR,          /**< See \ref BSP_INDICATE_FATAL_ERROR. */
BSP_INDICATE_ALERT_0,              /**< See \ref BSP_INDICATE_ALERT_0. */
BSP_INDICATE_ALERT_1,              /**< See \ref BSP_INDICATE_ALERT_1. */
BSP_INDICATE_ALERT_2,              /**< See \ref BSP_INDICATE_ALERT_2. */
BSP_INDICATE_ALERT_3,              /**< See \ref BSP_INDICATE_ALERT_3. */
BSP_INDICATE_ALERT_OFF,            /**< See \ref BSP_INDICATE_ALERT_OFF. */
BSP_INDICATE_USER_STATE_OFF,       /**< See \ref BSP_INDICATE_USER_STATE_OFF. */
```

这时候就可以开一个定时器延迟，延迟时间为 next_delay * ALERT_INTERVAL，也就是 5*200ms，有 1s 的时间延迟，也就是说，设置中断报警的时候，会有 1s 的频率闪灯。

如果前面设置为高等级报警，那么执行 BSP_INDICATE_ALERT_3，程序中的 BSP_INDICATE_ALERT_3 也下没有做任何操作，也是要往下面执行下去，那么执行到 next_delay 时，next_delay 为 1，也就是说开一个定时器延迟，有 200ms 的时间延迟，会产生 200ms 的频率闪灯，由于这个频率下，人眼是无法识别的，所以出现常亮的现象。实验中采用 MCP APP 测试如下如下。连接 APP 后，在立即报警服务中写入报警等级：



当输入中级报警，LED3 灯会有 1s 的频率闪灯报警。

当输入高等级报警，LED3 灯会有 200ms 的频率闪灯报警，人眼观察为常亮。当然如果有蜂鸣器，也可以设置蜂鸣器之类的发出噪声来寻物。

2.8.2.2 链接丢失服务：

链接丢失服务用于判断物体是否超出范围，用于寻物时使用，超出范围，链接丢失，此时，设备或者手机主机都会报警。函数如下所示：

链接丢失服务使用 ble_lls_init 函数初始化链接丢失服务，其中函数第二个形参 lls_init_obj 中的事件参数 evt_handler 作为触发回调事件，initial_alert_level 设置丢失后报警水平。

```
249. static void lls_init(void)
250. {
251.     uint32_t           err_code;
252.     ble_lls_init_t lls_init_obj;
253.
254.     // Initialize Link Loss Service
255.     memset(&lls_init_obj, 0, sizeof(lls_init_obj));
256.
257.     lls_init_obj.evt_handler               = on_lls_evt;
258.     lls_init_obj.error_handler             = service_error_handler;
259.     lls_init_obj.initial_alert_level = INITIAL_LLS_ALERT_LEVEL;
260.
261.   BLE_GAP_CONN_SEC_MODE_SET_ENC_NO_MITM(&lls_init_obj.lls_attr_md.read_perm);
```

```
262.     BLE_GAP_CONN_SEC_MODE_SET_ENC_NO_MITM(&lls_init_obj.lls_attr_md.write_perm);
263.
264.     err_code = ble_lls_init(&m_lls, &lls_init_obj);
265.     APP_ERROR_CHECK(err_code);
266. }
```

进入到源函数 ble_lls_init 的源函数中，把第二个形参的事件赋值给第一个形参的事件 evt,上面的初始化函数中设置的第一个形参为全局变量&m_lls 作为指示 my 链接丢失服务。那么本服务初始化函数和上一个立即报警服务类似，作为一个 SIG 的公共任务，初始化过程如下代码所示

```
267.  uint32_t ble_lls_init(ble_lls_t * p_lls, const ble_lls_init_t * p_lls_init)
268. {
269.     uint32_t     err_code;
270.     ble_uuid_t ble_uuid;
271.
272.     // Initialize service structure
273.     if (p_lls_init->evt_handler == NULL)
274.     {
275.         return NRF_ERROR_INVALID_PARAM;
276.     }
277.
278.     p_lls->evt_handler     = p_lls_init->evt_handler;
279.     p_lls->error_handler = p_lls_init->error_handler;
280.
281.     // 添加服务 UUID
282.     BLE_UUID_BLE_ASSIGN(ble_uuid, BLE_UUID_LINK_LOSS_SERVICE);
283.
284.     err_code = sd_ble_gatts_service_add(BLE_GATTS_SRVC_TYPE_PRIMARY,
285.                                         &ble_uuid,
286.                                         &p_lls->service_handle);
287.
288.     if (err_code != NRF_SUCCESS)
289.     {
290.         return err_code;
291.     }
292.
293.     // 报警水平特征值
294.       return alert_level_char_add(p_lls, p_lls_init);
295. }
```

报警特征值的添加需要严格遵循 SIG 提供的 Link-Loss-Service 服务手册，具体资料见我们资料文件夹，在手册中规定了服务类型，如下表所示，可写特征，可读特征，同时在特征值属性中设置报警水平指针。

	Broadcast	Read	Write without Response	Write	Notify	Indicate	Signed Write	Reliable Write	Writable Auxiliaries
Alert Level	X	M	X	M	X	X	X	X	X

```
296. static uint32_t alert_level_char_add(ble_lls_t * p_lls, const ble_lls_init_t * p_lls_init)
297. {
298.
299.     char_md.char_props.read   = 1;
300.     char_md.char_props.write = 1;
301.     char_md.p_char_user_desc = NULL;
302.     char_md.p_char_pf        = NULL;
303.     char_md.p_user_desc_md   = NULL;
304.     char_md.p_cccd_md        = NULL;
305.     char_md.p_sccd_md        = NULL;
306.     ..............................
307.     ..................................
308.     memset(&attr_char_value, 0, sizeof(attr_char_value));
309.     initial_alert_level = p_lls_init->initial_alert_level;
310.
311.    attr_char_value.p_uuid    = &ble_uuid;
312.     attr_char_value.p_attr_md = &attr_md;
313.     attr_char_value.init_len  = sizeof (uint8_t);
314.     attr_char_value.init_offs = 0;
315.     attr_char_value.max_len   = sizeof (uint8_t);
316.     attr_char_value.p_value   = &initial_alert_level;
317.
318.     return sd_ble_gatts_characteristic_add(p_lls->service_handle,
319.                                            &char_md,
320.                                            &attr_char_value,
321.                                            &p_lls->alert_level_handles);
322. }
```

同样我们需要关心下派发函做的工作，派发函数里就关心了连接，断开，认证三个事件。 其中连接事件 BLE_GAP_EVT_CONNECTED，认证 BLE_GAP_EVT_AUTH_STATUS 这两个事件是顺序执行的，在配对防丢器时，依次操作的。代码如下

这两个事件触发了 BLE_LLS_EVT_LINK_LOSS_ALERT ，同时设置了报警水平为 BLE_CHAR_ALERT_LEVEL_NO_ALERT 也就是说，刚连接的时候，我们会设置报警为无报警。那么当防丢器超出链接范围，发送丢失了？这时参数 BLE_GAP_EVT_DISCONNECTED 事件，这个事件触发什么操作了？下面来看下代码：

```
323. void ble_lls_on_ble_evt(ble_lls_t * p_lls, ble_evt_t * p_ble_evt)//链接丢失派发函数
```

```
324. {
325.     switch (p_ble_evt->header.evt_id)
326.     {
327.         case BLE_GAP_EVT_CONNECTED:
328.             on_connect(p_lls, p_ble_evt);//连接事件
329.             break;
330.
331.         case BLE_GAP_EVT_DISCONNECTED:
332.             on_disconnect(p_lls, p_ble_evt);//断开事件
333.             break;
334.
335.         case BLE_GAP_EVT_AUTH_STATUS:
336.             on_auth_status(p_lls, p_ble_evt);//认证事件
337.             break;
338.
339.         default:
340.             // No implementation needed.
341.             break;
342.     }
343.}
344.
345. static void on_connect(ble_lls_t * p_lls, ble_evt_t * p_ble_evt)//连接事件
346. {
347.     // Link reconnected, notify application with a no_alert event
348.     ble_lls_evt_t evt;
349.
350.     p_lls->conn_handle       = p_ble_evt->evt.gap_evt.conn_handle;
351.
352.     evt.evt_type                 = BLE_LLS_EVT_LINK_LOSS_ALERT;
353.     evt.params.alert_level = BLE_CHAR_ALERT_LEVEL_NO_ALERT;
354.     p_lls->evt_handler(p_lls, &evt);
355. }
356.
357. static void on_auth_status(ble_lls_t * p_lls, ble_evt_t * p_ble_evt)//认证事件
358. {
359.     if (p_ble_evt->evt.gap_evt.params.auth_status.auth_status ==
    BLE_GAP_SEC_STATUS_SUCCESS)
360.     {
361.         ble_lls_evt_t evt;
362.
363.         evt.evt_type                 = BLE_LLS_EVT_LINK_LOSS_ALERT;
364.         evt.params.alert_level = BLE_CHAR_ALERT_LEVEL_NO_ALERT;
365.
366.         p_lls->evt_handler(p_lls, &evt);
```

```
367.        }
368. }
```

这个就是断开事件了，这里启动了判断，是否是 BLE_HCI_CONNECTION_TIMEOUT 超时，如果是的，则启动 ble_lls_alert_level_get(p_lls, &evt.params.alert_level)函数，获取你 APP 设置的报警等级，当然，APP 这个必须是要设置有报警等级，如果还是无报警，那么防丢器即使丢失也是不会报警的。

```
369. static void on_disconnect(ble_lls_t * p_lls, ble_evt_t * p_ble_evt)//链接  丢失中的断开服务
370. {
371.        uint8_t reason = p_ble_evt->evt.gap_evt.params.disconnected.reason;
372.
373.        if (reason == BLE_HCI_CONNECTION_TIMEOUT)
374.        {
375.            // Link loss detected, notify application
376.            uint32_t          err_code;
377.            ble_lls_evt_t evt;
378.
379.            evt.evt_type = BLE_LLS_EVT_LINK_LOSS_ALERT;//触发链接丢失服务
380.            err_code = ble_lls_alert_level_get(p_lls, &evt.params.alert_level);//获取 APP 发来的报警
    水平
381.            if (err_code == NRF_SUCCESS)
382.            {
383.                p_lls->evt_handler(p_lls, &evt);
384.            }
385.            else
386.            {
387.                if (p_lls->error_handler != NULL)
388.                {
389.                    p_lls->error_handler(err_code);
390.                }
391.            }
392.        }
393. }
```

那么在 on_lls_evt 回调事件中，当发生丢失事件后，我们就更新报警水平，更新报警水平函数上节讲的立即报警中的更新报警水平使用的是同一个函数：

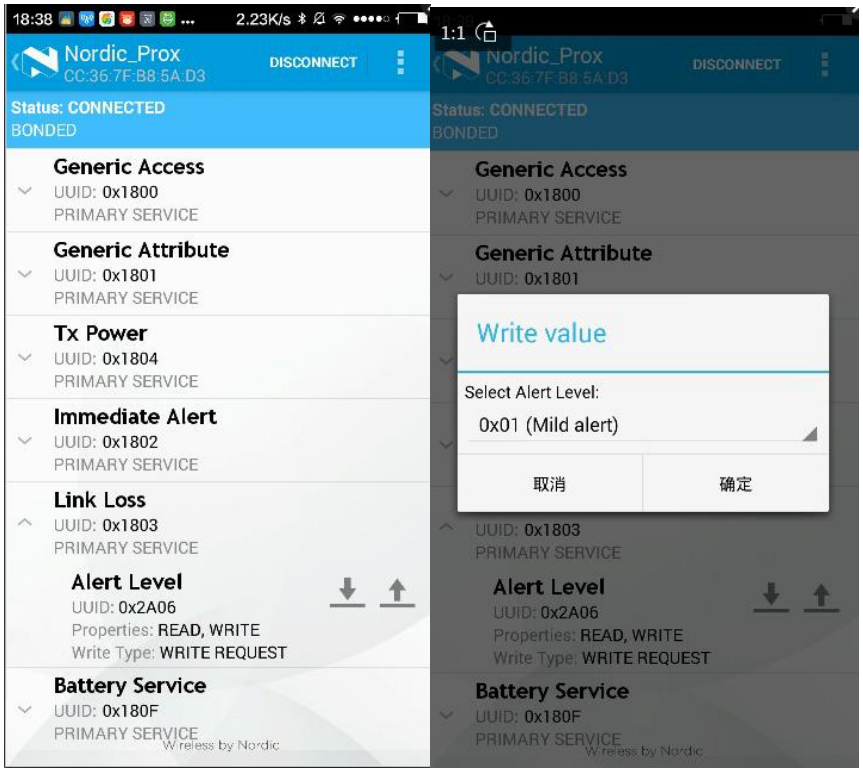
```
394. static void on_lls_evt(ble_lls_t * p_lls, ble_lls_evt_t * p_evt)
395. {
396.        switch (p_evt->evt_type)
397.        {
398.            case BLE_LLS_EVT_LINK_LOSS_ALERT://丢失事件
399.                alert_signal(p_evt->params.alert_level);//更新报警水平
400.                break;//BLE_LLS_EVT_LINK_LOSS_ALERT
401.
```

```
402.            default:
403.                // No implementation needed.
404.                break;
405.        }
406.}
```

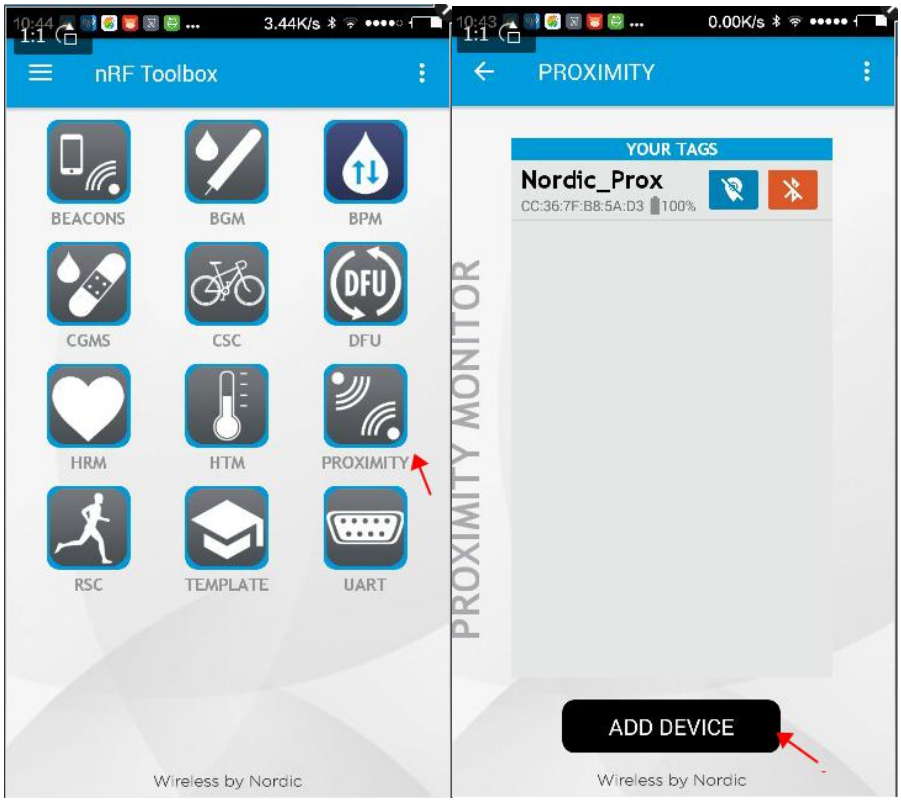
这时候，当我们手机 APP 设置为立即丢失服务为中等报警的时候，当防丢器丢失后，手机会收到丢失提醒，同时防丢器开始报警，报警采用的 1s 的 led 灯闪烁。

当我们手机 APP 设置为立即丢失服务为中等报警的时候，当防丢器丢失后，手机会收到丢失提醒，同时防丢器开始报警，报警采用的 200ms 的 led 灯闪烁，肉眼观察为常亮。

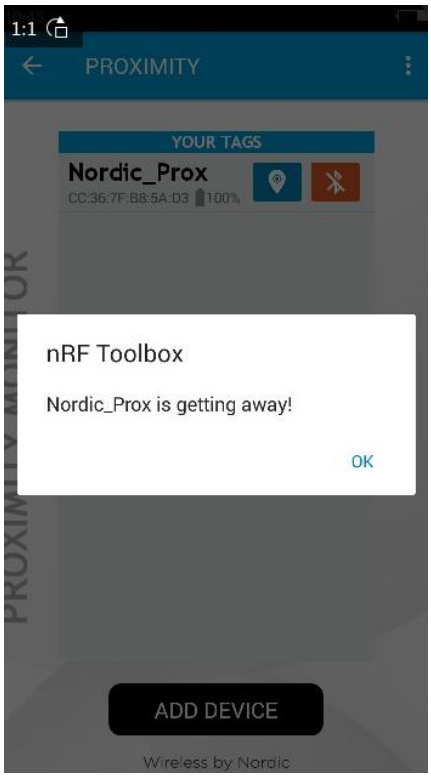
实验中采用 MCP APP 测试如下如下，连接服务，点击 link loss 写入箭头，写入中等服务 0x01,如下图所示：



断开 mcp 的连接，然后打开 APP nrf TOOL,点击 PROXIMITY 服务，添加防丢器广播，如下图所示：



把手机拿开范围外，如果超出范围，手机 APP 报警提示，同时，放丢器开发板会通过 LED3 灯进行 1s 的闪烁报警，当然可以换成蜂鸣器之类的设备：



2.8.2.3 双向报警之主机报警：

要实现双向报警的主机报警，也就是需要采用立即报警客户端服务了，这个服务就是主设备可以连接多个从设备，也就是防丢器，同时可以使主设备进行报警。那么首先就是初始化立即报警客户端服务，并且触发回调事件 on_ias_c_evt，代码如下：

```
407. static void ias_client_init(void)
408. {
409.     uint32_t               err_code;
410.     ble_ias_c_init_t ias_c_init_obj;
411.
412.     memset(&ias_c_init_obj, 0, sizeof(ias_c_init_obj));
413.
414.     m_is_high_alert_signalled = false;
415.
416.     ias_c_init_obj.evt_handler     = on_ias_c_evt;//回调函数
417.     ias_c_init_obj.error_handler =    service_error_handler;
418.
419.     err_code = ble_ias_c_init(&m_ias_c, &ias_c_init_obj);//初始化立即报警客户端服务
420.     APP_ERROR_CHECK(err_code);
421. }
```

立即报警客户端服务的服务建立初始化函数如下，这个服务直接分配 UUID,通知启动设备返现注册函数 ble_db_discovery_evt_register，具体代码如下：

```
422. uint32_t ble_ias_c_init(ble_ias_c_t * p_ias_c, ble_ias_c_init_t const * p_ias_c_init)
423. {
424.     VERIFY_PARAM_NOT_NULL(p_ias_c);
425.     VERIFY_PARAM_NOT_NULL(p_ias_c_init->evt_handler);
426.     VERIFY_PARAM_NOT_NULL(p_ias_c_init);
427.   //初始化服务结构体
428.     p_ias_c->evt_handler     = p_ias_c_init->evt_handler;
429.     p_ias_c->error_handler   = p_ias_c_init->error_handler;
430.     p_ias_c->conn_handle   = BLE_CONN_HANDLE_INVALID;
431.     p_ias_c->alert_level_char.handle_value = BLE_GATT_HANDLE_INVALID;
432.    //配置 UUID
433.     BLE_UUID_BLE_ASSIGN(p_ias_c->alert_level_char.uuid,
    BLE_UUID_ALERT_LEVEL_CHAR);
434.     BLE_UUID_BLE_ASSIGN(p_ias_c->service_uuid,
    BLE_UUID_IMMEDIATE_ALERT_SERVICE);
435.       //启动设备发现注册
436.     return ble_db_discovery_evt_register(&p_ias_c->service_uuid);
437. }
```

这时候就初始化了立即报警客户端服务了，那么在主函数中启动设备发现函数

db_discovery_init，代码如下:

```
438. static void db_discovery_init(void)
439. {
440.     uint32_t err_code = ble_db_discovery_init(db_disc_handler);
441.
442.     APP_ERROR_CHECK(err_code);
443. }

444. static void db_disc_handler(ble_db_discovery_evt_t * p_evt)
445. {
446.     ble_ias_c_on_db_disc_evt(&m_ias_c, p_evt);
447. }
```

最终会启动设备发现服务，我们直接说其中的重点，也就是当报警水平特征值为有效的时候，会启动 evt_type 为 BLE_IAS_C_EVT_DISCOVERY_COMPLETE，也就是发现客户端事件。

```
448. void ble_ias_c_on_db_disc_evt(ble_ias_c_t * p_ias_c, const ble_db_discovery_evt_t * p_evt)
449. {
450. .....
451. .....
452.
453.     if (evt.alert_level.handle_value != BLE_GATT_HANDLE_INVALID)
454.     {
455.         evt.evt_type = BLE_IAS_C_EVT_DISCOVERY_COMPLETE;
456.     }
457.
458.     p_ias_c->evt_handler(p_ias_c, &evt);
459. }
```

这时表示发现立即报警客户端被完全发现，那么就可以启动报警的指针 m_is_ias_present 为真，并且设置函数 ble_ias_c_handles_assign 可以配置多个客户端。

```
460. static void on_ias_c_evt(ble_ias_c_t * p_ias_c, ble_ias_c_evt_t * p_evt)
461. {
462.     uint32_t err_code;
463.
464.     switch (p_evt->evt_type)
465.     {
466.         case BLE_IAS_C_EVT_DISCOVERY_COMPLETE:
467.             // IAS is found on peer. The Find Me Locator functionality of this app will work.
468.             err_code = ble_ias_c_handles_assign(&m_ias_c,
469.                                                 p_evt->conn_handle,
470.                                                 p_evt->alert_level.handle_value);
471.             APP_ERROR_CHECK(err_code);
472.
473.             m_is_ias_present = true;//报警指针
```

```
474.                 break;//BLE_IAS_C_EVT_DISCOVERY_COMPLETE
475.             ..........................................
476.             ..........................................
477.         }
478. }
```

主机报警如何启动了？比如我们手机丢了，我们可以通过按下防丢器上的按键，使得手机报警发声，来找到主机。因此，在按键中断中设置按键事件，当按键 0 被按下后，同时前面的发现设备后产生的报警指针为真，m_is_high_alert_signalled 也为真的时候，向手机的发送一个高报警等级。如果你再按一下，再发生一个无报警等级。

这现象是，如果手机在连接范围内。你按下按键 1 后，手机首先报警，然后再按下按键 1 后，手机不报警。当然需要注意，按键按的是短按，长按会产生模拟链路丢失的现象，具体大家详细看代码，这里不展开。

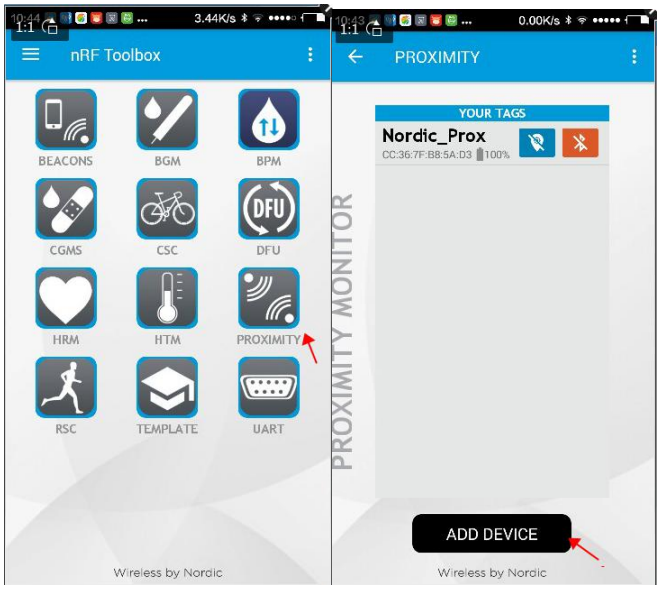
```
479. static void bsp_event_handler(bsp_event_t event)
480. {
481.     uint32_t err_code;
482.
483.     switch (event)
484.     {
485.         ..........................................
486.         ..........................................
487.         case BSP_EVENT_KEY_0://设置按键 1 触发事件
488.         {
489.             if (m_is_ias_present)//如果发现客户端
490.             {
491.                 if (!m_is_high_alert_signalled)
492.                 {
493.                 err_code = ble_ias_c_send_alert_level(&m_ias_c,
    BLE_CHAR_ALERT_LEVEL_HIGH_ALERT);//发送报警等级
494.                 }
495.                 else
496.                 {
497.                  err_code = ble_ias_c_send_alert_level(&m_ias_c,
    BLE_CHAR_ALERT_LEVEL_NO_ALERT);//发送报警等级
498.                 }
499.
500.                 if (err_code == NRF_SUCCESS)
501.                 {
502.                     m_is_high_alert_signalled = !m_is_high_alert_signalled;
503.                 }
504.                 else if (
505.                     (err_code != BLE_ERROR_NO_TX_PACKETS)
506.                     &&
507.                     (err_code != BLE_ERROR_GATTS_SYS_ATTR_MISSING)
```

```
508.                            &&
509.                            (err_code != NRF_ERROR_NOT_FOUND)
510.                                 )
511.                        {
512.                            APP_ERROR_HANDLER(err_code);
513.                        }
514.                }
515.            }break;//BSP_EVENT_KEY_0
516.
517.            case BSP_EVENT_KEY_1: // STOP_ALERTING_BUTTON_ID   设置按键 2 为停止报
    警
518.                err_code = bsp_indication_set(BSP_INDICATE_ALERT_OFF);
519.                APP_ERROR_CHECK(err_code);
520.                break;//BSP_EVENT_KEY_1
521.
522.            default:
523.                break;
524.        }
525.}
```

ble_ias_c_send_alert_level 函数就是写一个 GATT 的特征值给手机，手机 APP 根据特征值来设置报警声音了。 上面的按键 1 是用于解除绑定，以便连接另一个设备

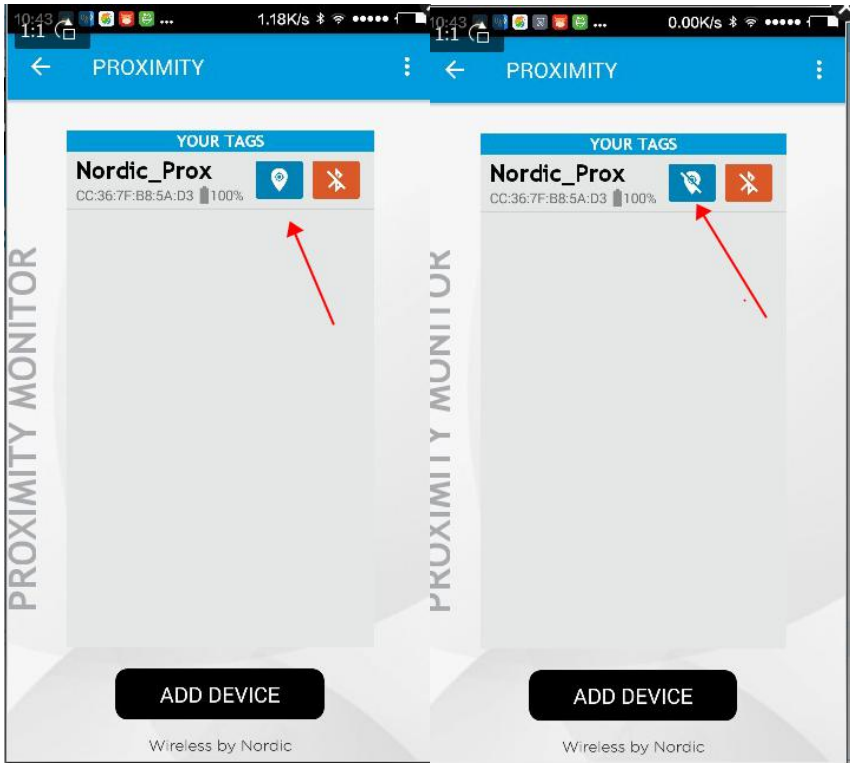
2.8.3 蓝牙防丢器调试：

首先开发板下载协议栈，然后下载应用程序。再打开防丢器 APP,如下图 1 所示，如果开发板程序下载成功，则点击防丢器广播，并且进行连接，如下图 2：

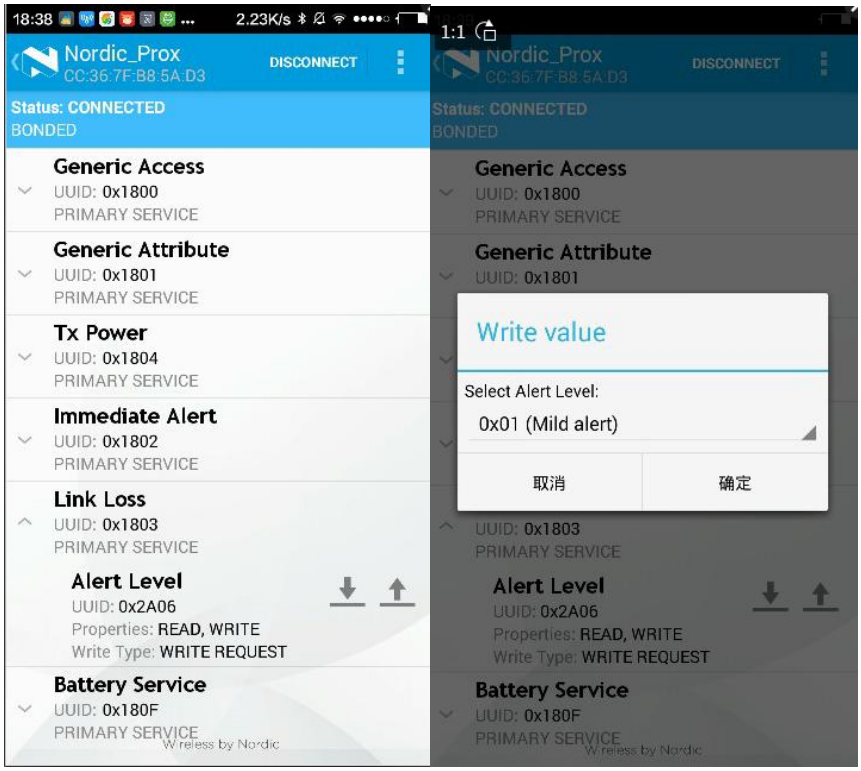


首先是寻物功能：点击箭头指向的寻物功能，点击后，事件上就是向防丢器写了一个高报警

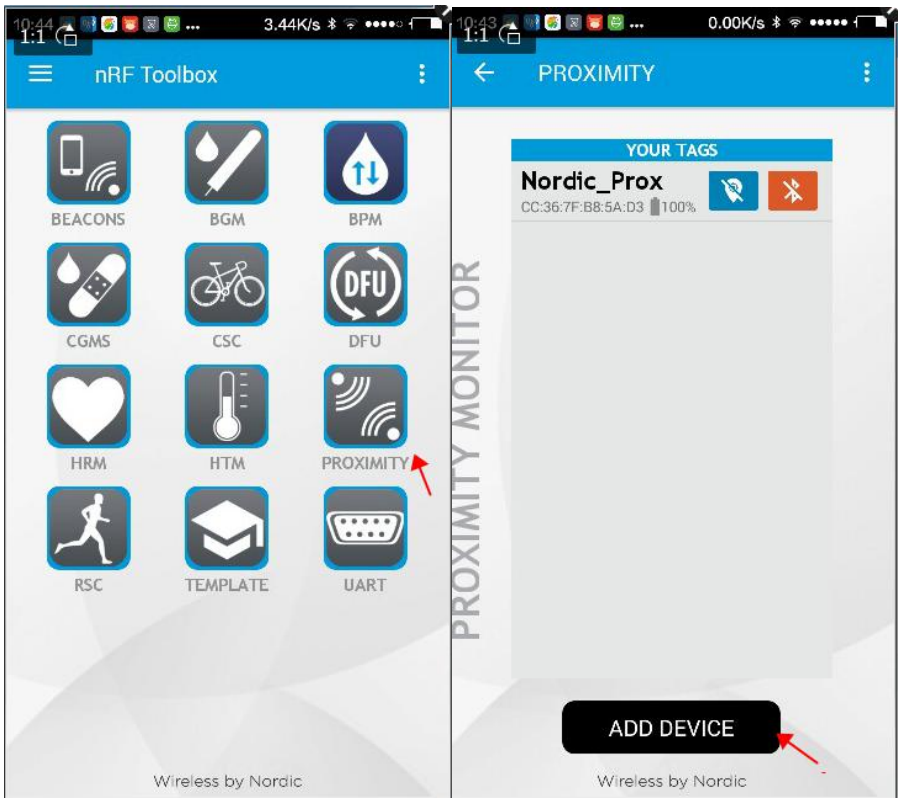
等级，LED3 灯会常亮报警：

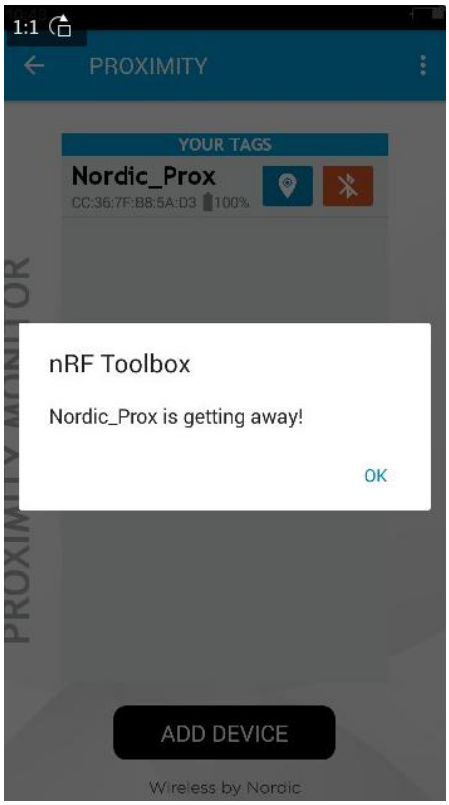


然后是防丢功能，防丢功能需要先设置丢失后的报警等级，如果不设置，默认无报警，当丢失链接后，开发板是不会报警的，只有手机会报警提示。打开 MCP app，连接 PROX 服务，如图 1 所示，找到 link loss 服务，点击写入箭头，我们设置为中等报警，如下图 2 所示，写入成功后，断开 APP 的连接：



再打开防丢器 APP，点击防丢器广播，并且进行连接，如下图所示，那么把开发进行远离手机设备，当超出链接范围，手机会出现如图 3 的丢失报警，同时开发板的 LED3 会进行报警闪烁：





最后是防丢器找手机：

按下按键 1，手机首先鸣叫报警，然后再按下按键 1 后，手机不报警。

2.8.4 本章总结：

本章主要分析了防丢器中的三个主要的服务功能，这三个服务建立了，也就对应了双向报警，防丢这三个功能。我们开发板上只有 LED 设备，当然大家可以加入蜂鸣器设备等发出噪音进行报警，也就是在报警等级更新函数中设置。本例是一个很经典的蓝牙 4.0 的应用，希望大家认真阅读代码进行研究。