RTL8762D Proximity Smart Application Design Spec

V1.0

2020/06/28

修订历史

日期	版本	修改	作者
2020/06/28	V0.1	初稿，由 RTL8762C 文档继承	Ken_mei

目　录

修订历史 2
目　录 3
图目录 5
表目录 6
1. 概述 7
1.1. 器件清单 7
1.2. 系统需求 8
1.3. 术语定义 8
2. 硬件设计 9
2.1. 电路设计 9
2.1.1. Master 9
2.1.2. Slave 9
2.2. 引脚定义 9
2.2.1. Master 9
2.2.2. Slave 9
3. 防丢器简单操作说明 10
4. 防丢器的软件概述 11
5. 防丢器 IO 初始化和处理 12
5.1. IO 初始化 12
5.2. IO 处理 13
5.2.1. 按键的处理 13
5.2.2. IO 逻辑的实现 14
6. 按键状态变换说明 16
6.1. Slave 长按键 16
6.2. Master 长按键 18
6.3. Slave 短按键 20
6.4. Master 短按键 21
7. 广播 23
8. GATT 相关 Service 和 Characteristic 24
8.1. Immediately Alert Service 24
8.2. Link Loss Alert Service 27
8.3. Tx Power Service 33
8.4. Battery Service 35

8.5. Device Information Service 37
8.6. Key Notification Service 46
9. Central Application 51
9.1. Immediately Alert Client 51
9.2. Link Loss Alert Client 51
9.3. Tx Power Client 52
9.4. Battery Client 54
9.5. Device Information Client 56
9.6. Key Notification Client 59
10. 服务的初始化和注册回调函数 61
11. DLPS 62
11.1. DLPS 概述 62
11.2. DLPS 使能和配置 62
11.3. DLPS 的条件和唤醒源 64
12 参考文献 65

图目录

图 1.1 防丢器使用场景.....7
图 4.1 防丢器应用系统框图.....11
图 5.1 IO 的初始化流程 12
图 5.2 蓝牙状态迁移.....14

表目录

表 8.1 包含的 Service 以及 UUID 列表.....24
表 8.2 Immediate Alert Service Characteristic 列表.....24
表 8.3 Alert Level Characteristic Value Format.....24
表 8.4 Alert Level Enumerations.....25
表 8.5 Alert Level Enumerations.....26
表 8.6 Link Loss Service Characteristic 列表.....27
表 8.7 Alert Level Characteristic Value Format.....27
表 8.8 Alert Level Enumerations.....27
表 8.9 Tx Power Service Characteristic 列表.....33
表 8.10 Tx Power Level Characteristic Value Format.....33
表 8.11 Battery Service Characteristic 列表.....35
表 8.12 Battery Level Characteristic Value Format.....35
表 8.13 Device Information Service Characteristic 列表.....38
表 8.14 Device Information Characteristic Value Format.....38
表 8.15 System ID Characteristic Value Format.....38
表 8.16 IEEE 11073-20601 Regulatory Certification Data List Characteristic Value Format.....39
表 8.17 PnP ID Characteristic Value Format.....39
表 8.18 Vendor ID Enumerations.....39
表 8.19 Key Notification Service Characteristic 列表.....47
表 8.20 Set Alert Time Characteristic Value Format.....47
表 8.21 Key Value Characteristic Value Format.....47

1. 概述

1.1. 关于防丢器(Proximity)

防丢器的设计主要是针对手机、钱包、钥匙、行李等贵重物品的防丢设计方案，以及对丢失物品的寻找。移动终端和防丢器处于连接状态之下，手机和防丢器均可发起寻找功能，即实现声光报警。当防丢器和移动终端由于距离较远时也会，移动终端和防丢器也会同时发出提醒报警。

图 1.1 防丢器使用场景

1.2. 关于智能防丢器(Proximity Smart)

智能防丢器是普通防丢器的一个超集。除普通防丢器功能外， 智能防丢器是主从一体角色， 也就是说智能防丢器被另外一个智能防丢器或者手机等连接的同时， 还可以作为主机连接的智能防丢， 通俗的讲， 多个智能防丢器连在一起， 就成了一组防丢链。

1.3. 器件清单

1. Bee Evaluation Board 母版
2. RTL8762D_QFN40/QFN48 子板

3. 蜂鸣器

1.4. 系统需求

PC 端需要下载和安装的工具：

1. Keil MDK-ARM 5.12
2. SEGGER’s J-Link tools
3. RTL8762D SDK
4. RTL8762x Flash programming algorithm

1.5. 术语定义

1. master：连接的发起方；
2. slave：连接的接受方。
3. 防丢器：基于 RTL8762D 开发的防丢器应用；
4. DPLS：Deep Low Power State，超低功耗睡眠状态。

2. 硬件设计

2.1. 电路设计

2.1.1. Master

Master 电路部分直接用 RTL8762D EVB 进行模拟，具体请参照 RTL8762D EVB SCH。其中按键用 EVB 上 KEY3 模拟，LED 用 EVB 上 LED1 模拟。

2.1.2. Slave

Slave 电路部分直接用 RTL8762D EVB 进行模拟，具体请参照 RTL8762D EVB SCH。其中按键用 EVB 上 KEY2 模拟，LED 用 EVB 上 LED0 模拟，BEEP 用 EVB 上 LED2 代替。

2.2. 引脚定义

2.2.1. Master

RTL8762D 子板：

```
LED     P0_2
BEEP    P3_4
KEY     P2_3
```

2.2.2. Slave

RTL8762D 子板：

```
LED     P0_1
BEEP    P3_4
KEY     P2_4
```

3. 防丢器简单操作说明

1. 初始状态防丢器在 idle 状态，短按 Master 和 Slave 按键可以看到 LED 闪烁一下，表示可以正常工作。
2. Slave 长按按键直到 LED 开始 1s 闪烁，此时 slave 开始广播。
3. Master 长按按键直到 LED 开始 1s 闪烁，此时 master 开始扫描广播。
4. 当 master 与 slave 连接成功 LED 改为 3.7s 闪烁。
5. Master 短按按键 slave 的 LED 闪烁和蜂鸣器鸣叫，此时 slave 短按按键可停止 LED 和蜂鸣器。
6. Slave 短按按键 master 的 LED 闪烁和蜂鸣器鸣叫，此时 master 短按按键可停止 LED 和蜂鸣器。
7. 当 master 和 slave 任意一方断开连接，LED 闪烁和蜂鸣器鸣叫。
8. 同时作为 master 和 slave 复杂行为，详见第 6 章。

4. 防丢器的软件概述

防丢器应用主要与 IO Driver 和 BT 的交互，完成特定的功能，其主要功能的架构如图 4.1 所示。

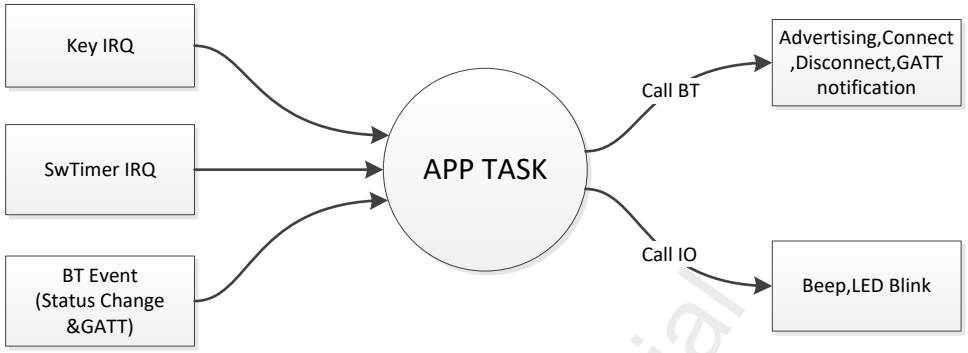

图 4.1 防丢器应用系统框图

系统初始化之后，App task 任务开始运行，该任务主要完成创建应用层消息队列，向 Upper Stack 注册应用层的回调函数，以及对应用层的外围借口进行初始化，然后不断地查询消息队列以完成对消息的处理。Upper stack，IO ISR ，Timer ISR 发送消息到应用层的消息队列，然后 App task 从消息队列中取出消息进行处理。

```
while (true)
    {
        if (os_msg_recv(evt_queue_handle, &event, 0xFFFFFFFF) == true)
        {
            if (event == EVENT_IO_TO_APP)
            {
                T_IO_MSG io_msg;
                if (os_msg_recv(io_queue_handle, &io_msg, 0) == true)
                {
                    app_handle_io_msg(io_msg);
                }
            }
            else
            {
                gap handle msg(event);
            }
        }
    }
```

5. 防丢器 IO 初始化和处理

防丢器定义了 3 个 IO， 分别为 LED ，KEY， BEEP。在 board.h 中定义如下：

```
#define LED_M     P0_2
#define LED_S     P0_1
#define BEEP      P3_4
#define KEY_S     P2_4
```

5.1. IO 初始化

IO 的初始化流程如下：

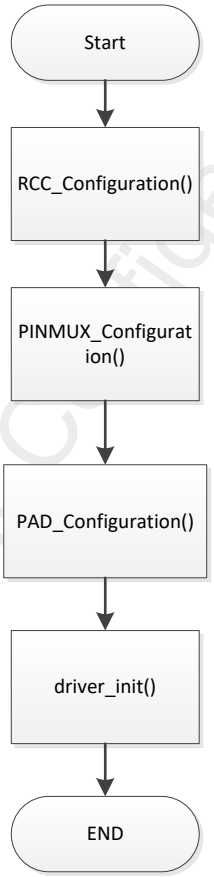

图 5.1 IO 的初始化流程

在 driver_init()中，注册并使能了 KEY 的中断，为防止 OS 在初始化中接受到中断导致 OS 逻辑异常， 建议 driver_init()放在 apptask 的中调用。

```
void app_main_task(void *p_param)
{
    uint8_t event;

    os_msg_queue_create(&io_queue_handle,      MAX_NUMBER_OF_IO_MESSAGE,
```

```
sizeof(T_IO_MSG));
    os_msg_queue_create(&evt_queue_handle,  MAX_NUMBER_OF_EVENT_MESSAGE,
sizeof(uint8_t));

    gap_start_bt_stack(evt_queue_handle,                io_queue_handle,
MAX_NUMBER_OF_GAP_MESSAGE);

    driver_init();
    ……
    ……
    ……
}
```

5.2. IO 处理

5.2.1. 按键的处理

防丢器中的按键区分长按和短按，长按和短按的实现如下：

按键的状态定义如下：

```
typedef enum
{
    MSG_KEYS_SHORT_PRESS     = 1,
    MSG_KEYS_SHORT_RELEASE   = 2,
    MSG_KEYS_LONG_PRESS      = 3,
    MSG_KEYS_LONG_RELEASE    = 4,
    MSG_KEYM_SHORT_PRESS     = 5,
    MSG_KEYM_SHORT_RELEASE   = 6,
    MSG_KEYM_LONG_PRESS      = 7,
    MSG_KEYM_LONG_RELEASE    = 8,
} T_MSG_TYPE_PXP_GPIO;
```

其中 KEYS，表示 slave 的 key，KEYM 表示 master 的 key。

没有按键按下时，gKeySStatus / gKeyMStatus 为 keyIdle。

当有按键按下的时候，首先判断电平是否发生变化，以区分是 master 还是 slave，并进入相应处理函数。设置按键状态 gKeySStatus / gKeyMStatus 为 keyShortPress，翻转中断触发极性，同时启动长按键定时器。

如果是长按键，定时器到期后，sw timer 的处理 handle 就将按键状态设置为 keyLongPress，同时发出长按的 message 给 apptask。

如果是短按键，当按键 release 后，中断触发，先翻转中断触发极性（等待后续的 key press）;然后判断按键状态，如果是 keyShortPress，则停止长按键定时器，同时发出短按的 message 给 apptask； 最后设置按键状态为 keyIdle。

Master 和 slave 使用同一套代码，在 APP task 收到长按的 message 后，做蓝牙相关的状态切换逻辑。收到短按的 message 后， 做 io 相关的逻辑。

防丢器的蓝牙状态如下：

```
    typedef enum _PxpState
    {
```

```
    PxpStateIdle     = 0,
    PxpStateAdv      = 1,
    PxpStateScan     = 2,
    PxpStateAdvScan   = 3,
    PxpStateLinkM    = 4,
    PxpStateLinkS    = 5,
    PxpStateLinkMADV  = 6,
    PxpStateLinkSScan = 7,
    PxpStateLinkMS    = 8,
} PxpState;
```

防丢器的 IO 状态如下：

```
typedef enum _IoState
{
    IoStateIdle    = 0,
    IoStateAdvScanBlink = 1,
    IoStateLinkBlink = 2,
    IoStateImmAlert = 3,
    IoStateLlsAlert = 4
} IoState;
```

下面是状态的迁移图,状态迁移图也非常直观的表述了防丢器长按的蓝牙状态切换逻辑：

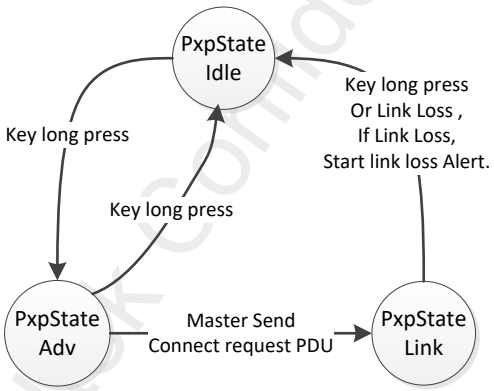

图 5.2 蓝牙状态迁移

防丢器按键短按逻辑：

A) 如果防丢器在 idle 状态， LED 闪烁一下， 表示设备能够正常工作；

B) 如果防丢器的 IO 在 Alert 状态， 关闭 LED 和 BEEP；

C) 如果防丢器在 link 状态， 且 IO 不在 Alert 状态，则发送 alert 的 notification 到 master。

5.2.2. IO 逻辑的实现

IO 逻辑实现是 Master 靠启动 void StartPxpMIO(uint32_t lowPeroid, uint32_t HighPeroid, uint8_t mode, uint32_t cnt)这个函数实现的，slave 靠启动 void StartPxpSIO(uint32_t lowPeroid, uint32_t HighPeroid, uint8_t mode, uint32_t cnt)这个函数实现的。这个函数是控制的防丢器的 LED 和 BEEP，它有四个参数。第一个参数 lowPeroid 是表示 IO 翻转时低电平的持续时间，

第二参数 HighPeroid 表示 IO 翻转时高电平的持续时间，第三个参数 mode 表示 LED 还是 BEEP 还是两个一起启动，第一个参数 cnt 表示 IO 翻转的次数。

通过启动 xTimerPxpIO 的软定时器， 当到达软定时器的中断后，根据当前 IO 的情况设置下一次定时器的延时时间， 并根据 cnt 值判断定时器是否需要重启继续 IO 的翻转状态。

6. 按键状态变换说明

6.1. Slave 长按键

长按键发生后向 apptask 发送消息，apptask 利用 Pxp_HandleButtonEvent 函数进行处理。如果消息是 slave 长按键，根据当前状态处理。

1. 初始状态是 PxpStateIdle，开始广播 le_adv_start();,PxpState 改为 PxpStateAdv。
2. 在状态 PxpStateAdv 下长按键，停止广播 le_adv_stop();
3. 当处于 scan 时，即 PxpStateScan 状态，按下 slave 长按键，则开始广播 le_adv_start();
4. 当处于 scan 和 adv 时，即 PxpStateAdvScan 状态，按下 slave 长按键，则停止广播 le_adv_stop();
5. 当作为 master 已经建立连接时，即 PxpStateLinkM 状态，按下 slave 长按键，开始广播 le_adv_start();
6. 当作为 slave 已经建立连接时，即 PxpStateLinkS 状态，按下 slave 长按键，则断开连接 le_disconnect();
7. 当作为 master 已经建立连接，同时还在广播时，即 PxpStateLinkMADV 状态，按下 slave 长按键，则停止广播 le_adv_stop();
8. 当作为 slave 已经建立连接，同时还在扫描时，即 PxpStateLinkSScan 状态，按下 slave 长按键，则断开连接 le_disconnect();
9. 当同时作为 slave 和 master 已经建立连接时，即 PxpStateLinkMS 状态，按下 slave 长按键，则断开连接 le_disconnect();

```
else if (keytype == MSG_KEYS_LONG_PRESS)
    {
        switch (gPxpState)
        {
        case PxpStateIdle:
            if (gPowerFlg == false)
            {
                gPowerFlg = true;
                le_adv_start();
            }
            else
            {
                APP_PRINT_ERROR0("ERROR POWER STATUS");//error status
            }
            break;
        case PxpStateAdv:
            if (gPowerFlg == true)
            {
                gPowerFlg = false;
```

```
            le_adv_stop();
        }
        else
        {
            APP_PRINT_ERROR0("ERROR POWER STATUS");//error status
        }
        break;
    case PxpStateScan:
        le_adv_start();
        break;
    case PxpStateAdvScan:
        le_adv_stop();
        break;
    case PxpStateLinkM:
        le_adv_start();
        break;
    case PxpStateLinkS://link as slave
        if (gPowerFlg == true)
        {
            gPowerFlg = false;
            for (i = 0; i < APP_MAX_LINKS; i++)
            {
                if (PXPLINK[i] == SlaveLink)
                {
                    le_disconnect(i);
                }
                break;
            }
        }
        else
        {
            APP_PRINT_ERROR0("ERROR POWER STATUS");//error status
        }
        break;
    case PxpStateLinkMADV:
        le_adv_stop();
        break;
    case PxpStateLinkSScan:
    case PxpStateLinkMS:
        for (i = 0; i < APP_MAX_LINKS; i++)
        {
            if (PXPLINK[i] == SlaveLink)
            {
                le_disconnect(i);
            }
            break;
        }
        break;
    default:
        break;
    }
};
```

6.2. Master 长按键

长按键发生后向 apptask 发送消息，apptask 利用 Pxp_HandleButtonEvent 函数进行处理。如果消息是 master 长按键，根据当前状态处理。

1. 初始状态是 PxpStateIdle，开始扫描 le_adv_start();,PxpState 改为 PxpStateScan;
2. 在状态 PxpStateScan 下长按键，停止扫描 le_scan_stop();
3. 当处于 adv 时，即 PxpStateAdv 状态，按下 master 长按键，则开始扫描 le_scan_start();
4. 当处于 scan 和 adv 时，即 PxpStateAdvScan 状态，按下 master 长按键，则停止扫描 le_scan_stop();
5. 当作为 slave 已经建立连接时，即 PxpStateLinkS 状态，按下 master 长按键，开始扫描 le_scan_start();
6. 当作为 master 已经建立连接时，即 PxpStateLinkM 状态，按下 master 长按键，则断开连接 le_disconnect();
7. 当作为 slave 已经建立连接，同时还在扫描时，即 PxpStateLinkSScan 状态，按下 master 长按键，则停止扫描 le_scan_stop();
8. 当作为 master 已经建立连接，同时还在广播时，即 PxpStateLinkMADV 状态，按下 master 长按键，则断开连接 le_disconnect();
9. 当同时作为 slave 和 master 已经建立连接时，即 PxpStateLinkMS 状态，按下 master 长按键，则断开连接 le_disconnect();

```
else if (keytype == MSG_KEYM_LONG_PRESS)
    {
        switch (gPxpState)
        {
        case PxpStateIdle:
            if (gPowerFlg == false)
            {
                gPowerFlg = true;
                link_mgr_clear_device_list();
                le_scan_start();
            }
            else
            {
                APP_PRINT_ERROR0("ERROR POWER STATUS");//error status
            }
            break;
        case PxpStateAdv:
        case PxpStateLinkS:
            le_scan_start();
            break;
        case PxpStateScan:
            if (gPowerFlg == true)
            {
                gPowerFlg = false;
                le_scan_stop();
```

```
            }
            else
            {
                APP_PRINT_ERROR0("ERROR POWER STATUS");//error status
            }
            break;
        case PxpStateAdvScan:
            le_scan_stop();
            break;
        case PxpStateLinkSScan:
            le_scan_stop();
            break;
        case PxpStateLinkM://link as master
            if (gPowerFlg == true)
            {
                gPowerFlg = false;
                for (i = 0; i < APP_MAX_LINKS; i++)
                {
                    if (PXPLINK[i] == MasterLink)
                    {
                        le_disconnect(i);
                    }
                    break;
                }
            }
            else
            {
                APP_PRINT_ERROR0("ERROR POWER STATUS");//error status
            }
            break;
        case PxpStateLinkMADV:
            for (i = 0; i < APP_MAX_LINKS; i++)
            {
                if (PXPLINK[i] == MasterLink)
                {
                    le_disconnect(i);
                }
                break;
            }
            break;
        case PxpStateLinkMS:
            for (i = 0; i < APP_MAX_LINKS; i++)
            {
                if (PXPLINK[i] == MasterLink)
                {
                    le disconnect(i);
                }
                break;
            }
            break;
        }
    }
```

6.3. Slave 短按键

短按键发生后向 apptask 发送消息，apptask 利用 Pxp_HandleButtonEvent 函数进行处理。如果消息是 slave 短按键，根据当前状态处理。

1. 初始状态 PxpStateIdle，slave 短按键按下 LED 闪烁一下 StartPxpSIO；
2. 处于广播状态 PxpStateAdv 下，当 gSIoState 为断线报警 IoStateLlsAlert 时，停止报警改为长闪 LED。
3. 当作为 slave 或 master 建立连接时，如果 gSIoState 为立即报警 IoStateImmAlert 时，停止报警改为长闪 LED。如果不是则利用 Key Notification Service，将按键消息通知出去。

```
    if (keytype == MSG_KEYS_SHORT_PRESS) //salve short press
    {
        switch (gPxpState)
        {
        case PxpStateIdle:
            if (gSIoState == IoStateIdle)
            {
                StartPxpSIO(ALERT_LOW_PERIOD, ALERT_HIGH_PERIOD, LED_BLINK,
                            1); /*low period 0.9s, high period 0.1s,  led blink,
1times(cnt)*/
            }
            break;
        case PxpStateAdv:
            if (gSIoState == IoStateLlsAlert) // close beep
            {
                StopPxpSIO();
                gSIoState = IoStateAdvScanBlink;
                StartPxpSIO(ALERT_LOW_PERIOD    *    4,    ALERT_HIGH_PERIOD,
LED_BLINK,
                            0xffffffff);
            }
            break;
        case PxpStateLinkS: //link as slave
        case PxpStateLinkSScan: //link as slave
        case PxpStateLinkMS: //link as slave
            if (gSIoState == IoStateImmAlert)
            {
                StopPxpSIO();
                gSIoState = IoStateLinkBlink;
                StartPxpSIO(ALERT LOW PERIOD    *    4,    ALERT HIGH PERIOD,
LED BLINK,
                            0xffffffff);
            }
            else
            {
                value_to_send = 1;
                for (i = 0; i < APP_MAX_LINKS; i++)
                {
                    if (PXPLINK[i] == SlaveLink)
```

```
                    server_send_data(i, kns_srv_id, KNS_KEY_VALUE_INDEX, \
                                     &value_to_send,          sizeof(uint8_t),
GATT_PDU_TYPE_NOTIFICATION);
                    break;
                }

            }
            break;
        default:
            break;
        }
    }
```

6.4. Master 短按键

短按键发生后向 apptask 发送消息，apptask 利用 Pxp_HandleButtonEvent 函数进行处理。如果消息是 master 短按键，根据当前状态处理。

1. 初始状态 PxpStateIdle，master 短按键按下 LED 闪烁一下 StartPxpMIO；

2. 处于扫描状态 PxpStateAdv 下，当 gSIoState 为断线报警 IoStateLlsAlert 时，停止报警改为长闪 LED。

3. 当作为 slave 或 master 建立连接时，如果 gSIoState 为立即报警 IoStateImmAlert 时，停止报警改为长闪 LED。如果不是则利用 Key Notification Service，将按键消息通知出去。

```
else if (keytype == MSG_KEYM_SHORT_PRESS)
    {
        switch (gPxpState)
        {
        case PxpStateIdle:
            if (gMIoState == IoStateIdle)
            {
                StartPxpMIO(ALERT_LOW_PERIOD, ALERT_HIGH_PERIOD, LED_BLINK,
                            1); /*low period 0.9s, high period 0.1s,  led blink,
1times(cnt)*/
            }
            break;
        case PxpStateScan:
            if (gMIoState == IoStateLlsAlert) // close beep
            {
                StopPxpMIO();
                gMIoState = IoStateAdvScanBlink;
                StartPxpMIO(ALERT_LOW_PERIOD    *    2,    ALERT_HIGH_PERIOD,
LED_BLINK,
                            0xffffffff);
            }
            break;
        case PxpStateLinkM:
        case PxpStateLinkMADV:
        case PxpStateLinkMS:
            if (gMIoState == IoStateImmAlert)
            {
                StopPxpMIO();
```

```
            gMIoState = IoStateLinkBlink;
            StartPxpMIO(ALERT_LOW_PERIOD    *    4,    ALERT_HIGH_PERIOD,
LED_BLINK,
                        0xffffffff);
         }
         else
         {
            uint8_t alertVal = 2;
            for (i = 0; i < APP_MAX_LINKS; i++)
            {
               if (PXPLINK[i] == MasterLink)
               {
                  ias_client_write_char(i,   sizeof(uint8_t),   &alertVal,
GATT_WRITE_TYPE_CMD);
               }
               break;
            }
         }
         break;
      default:
         break;
      }
   }
```

7. 广播

防丢器的广播内容如下，手机或者其他蓝牙主机设备进行搜索的时候，会搜索到设备名称为“PXP_SMART”的设备，广播内容包括 GAP_ADTYPE_FLAGS,UUID 为 IAS Service 的 UUID，Local Name 为“PXP_SMART”,如进行 active scan 的话，可以扫描到 scan response data 为 KEYRING 的 Appearance 的字段。

```
/** @brief  GAP - scan response data (max size = 31 bytes) */
static const uint8_t scan_rsp_data[] =
{
    0x03,
    GAP_ADTYPE_APPEARANCE,
    LO_WORD(GAP_GATT_APPEARANCE_UNKNOWN),
    HI_WORD(GAP_GATT_APPEARANCE_UNKNOWN),
};

/** @brief  GAP - Advertisement data (max size = 31 bytes, best kept short
to conserve power) */
static const uint8_t adv_data[] =
{
    0x02,
    GAP_ADTYPE_FLAGS,
    GAP_ADTYPE_FLAGS_LIMITED | GAP_ADTYPE_FLAGS_BREDR_NOT_SUPPORTED,

    0x03,
    GAP ADTYPE 16BIT COMPLETE,
    0x02,//LO WORD(GATT UUID IMMEDIATE ALERT SERVICE),
    0x18,//HI WORD(GATT UUID IMMEDIATE ALERT SERVICE),

    0x0A,
    GAP_ADTYPE_LOCAL_NAME_COMPLETE,
    'P', 'X', 'P', '_', 'S', 'M', 'A', 'R', 'T',
};
```

8. GATT 相关 Service 和 Characteristic

防丢器应用中包含如下几个 Service：

1. Immediate Alert Service：操作防丢器立即开始报警；
2. Link Loss Service: 发生断线时，防丢器报警；
3. Tx Power Service: 指示发射功率；
4. Battery Service: 汇报设备的电池电量，提醒更换电池，电量过低时不可以进行 OTA；
5. Device Information Service：显示设备基本信息；
6. Key Notification Service: 设置报警次数，发送按键报警到 Master。

各服务名称及 UUID 如表 8.1 所示。

表 8.1 包含的 Service 以及 UUID 列表

Service Name	Service UUID
Immediate Alert Service	0x1802
Link Loss Service	0x1803
Tx Power Service	0x1804
Battery Service	0x180F
Device Information Service	0x180A
Key Notification Service	0x0000FFD0-BB29-456D-989D-C44D07F6F6A6

8.1. Immediately Alert Service

Immediately Alert Service 是标准的 GATT Service，所有的功能和定义在 ias.c&ias.h 里实现。

表 8.2 Immediate Alert Service Characteristic 列表

Characteristic Name	Requirement	Characteristic UUID	Properties	Description
Alert Level	M	0x2A06	Write Without Response	See Alert Level

Alert Level 描述了设备的报警级别，数据格式为 8 位无符号整型，初始值为 0。报警级别分为 3 级，分别对应 0--不报警，1--温和报警，2--严厉报警，如表 8.4 所示。

表 8.3 Alert Level Characteristic Value Format

Names	Field Requirement	Format	Mininum Value	Maxinum Value	Additional Information
Alert Level	Mandatory	uint8	0	2	See Enumeration

表 8.4 Alert Level Enumerations

Key	0	1	2	3-255
Value	No Alert	Mild Alert	High Level	Reserved

Immediate Alert Service 仅定义一个 characteristic -- Alert Level，该 characteristic 是一个 control point，使对测端可以通过写(write no response)这个 characteristic 来触发本地设备报警，而且通过 Level 的值设定报警的级别。

当本地设备报警被触发后，以下的方式可以停止报警:

- 报警时间到达，目前程序中设定的是 10 次；
- 用户关闭报警；
- 新的 Alert Level 被写入；
- 链路断开，开启新的报警（之前的报警可以理解为停止）。

IAS 的 GATT Attribute Table 如下：

```
const T_ATTRIB_APPL ias_attr_tbl[] =
{
    /*----------------- Immediate Alert Service -------------------*/
    {
        (ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_LE),  /* wFlags     */
        {                                       /* bTypeValue */
            LO_WORD(GATT_UUID_PRIMARY_SERVICE),
            HI_WORD(GATT_UUID_PRIMARY_SERVICE),
            LO_WORD(GATT_UUID_IMMEDIATE_ALERT_SERVICE),    /* service UUID */
            HI_WORD(GATT_UUID_IMMEDIATE_ALERT_SERVICE)
        },
        UUID_16BIT_SIZE,                          /* bValueLen     */
        NULL,                                    /* pValueContext */
        GATT_PERM_READ                            /* wPermissions  */
    },

    /* Alert Level Characteristic */
    {
        ATTRIB_FLAG_VALUE_INCL,                     /* wFlags */
        {                                       /* bTypeValue */
            LO_WORD(GATT_UUID_CHARACTERISTIC),
            HI_WORD(GATT_UUID_CHARACTERISTIC),
            GATT_CHAR_PROP_WRITE_NO_RSP,                    /*  characteristic
properties */
        },
        1,                                      /* bValueLen */
        NULL,
        GATT_PERM_READ                            /* wPermissions */
    },

    /* Alert Level Characteristic value  */
    {
```

```
        ATTRIB_FLAG_VALUE_APPL,                  /* wFlags */
        {                                       /* bTypeValue */
            LO_WORD(GATT_UUID_CHAR_ALERT_LEVEL),
            HI_WORD(GATT_UUID_CHAR_ALERT_LEVEL)
        },
        0,                                      /* variable size */
        NULL,
        GATT_PERM_WRITE  |  GATT_PERM_READ                                  /*
wPermissions */
    }
};
```

表 8.5 Alert Level Enumerations

Names	Sub Content	Vlaue	Description
Alert Level	msg_type	SERVICE_CALLBACK_TYPE_WRITE_CHAR_VALUE	Write Characteristic Event
	msg_data	write_alert_level	Characteristic Write Value

当对测端通过 IAS 服务向防丢器发送报警数据时，App task 调用 IAS 的写回调函数，并在 AppHandleGATTCallback 进行处理。

```
else if (service_id == ias_srv_id)
{
  T_IAS_CALLBACK_DATA *p_ias_cb_data = (T_IAS_CALLBACK_DATA *)p_data;
  APP_PRINT_INFO1("p_ias_cb_data->msg_type is %x", p_ias_cb_data->msg_type);
  if(p_ias_cb_data->msg_type == SERVICE_CALLBACK_TYPE_WRITE_CHAR_VALUE)
  {
   g_pxp_immediate_alert_level= p_ias_cb_data->msg_data.write_alert_level;
   os_timer_stop(&xTimerAlert);
   if (g_pxp_immediate_alert_level)
   {
     if (g_pxp_immediate_alert_level == 2)
     {
       StopPxpSIO();
       gSIoState = IoStateImmAlert;
       StartPxpSIO(ALERT_LOW_PERIOD, ALERT_HIGH_PERIOD, LED_BLINK | BEEP_ALERT,
                           gTimeParaValue);
     }
     else if (g_pxp_immediate_alert_level == 1)
     {
        StopPxpSIO();
        gSIoState = IoStateImmAlert;
        StartPxpSIO(ALERT_LOW_PERIOD, ALERT_HIGH_PERIOD, LED_BLINK,
                           gTimeParaValue);
     }
   }
  }
}
```

在防丢器中，用户可以发送三种不同等级的 Immediate Alert，即对 IAS Service 写入的属性值：

1. 0：不报警；
2. 1：LED 闪烁报警；
3. 2：LED 闪烁报警和蜂鸣器报警；

8.2. Link Loss Alert Service

Link Loss Alert Service 是标准的 GATT Service，所有的功能和定义在 lls.c&lls.h 里实现。

表 8.6 Link Loss Service Characteristic 列表

Characteristic Name	Requirement	Characteristic UUID	Properties	Description
Alert Level	M	0x2A06	Read/Write	See Alert Level

Alert Level 描述了设备的报警级别，数据格式为 8 位无符号整型，初始值为 0。报警级别分为 3 级，分别对应 0--不报警，1--温和报警，2--严厉报警，如表 8.8 所示。

表 8.7 Alert Level Characteristic Value Format

Names	Field Requirement	Format	Mininum Value	Maxinum Value	Additional Information
Alert Level	Mandatory	uint8	0	2	See Enumeration

表 8.8 Alert Level Enumerations

Key	0	1	2	3-255
Value	No Alert	Mild Alert	High Level	Reserved

Link Loss Alert Service 仅定义一个 characteristic -- Alert Level，该 characteristic 是读写的，使对测端可以通过写(write response)这个 characteristic 来设置设备的报警的级别，报警级别设置好后，当发生 link loss 后，程序会根据设备的报警级别，启动相应的报警状态，包括 LED 闪烁，蜂鸣器报警等。

LLS 的 GATT Attribute Table 如下：

```
/**< @brief profile/service definition.  */
const T_ATTRIB_APPL lls_attr_tbl[] =
{
    /*----------------- Link Loss Service ------------------*/
    {
        (ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_LE),  /* wFlags     */
        {                                       /* bTypeValue */
            LO_WORD(GATT_UUID_PRIMARY_SERVICE),
            HI_WORD(GATT_UUID_PRIMARY_SERVICE),
            LO_WORD(GATT_UUID_LINK_LOSS_SERVICE),    /* service UUID */
            HI_WORD(GATT_UUID_LINK_LOSS_SERVICE)
```

```
        },
        UUID_16BIT_SIZE,                          /* bValueLen     */
        NULL,                                    /* pValueContext */
        GATT_PERM_READ                            /* wPermissions  */
    },

    /* Alert Level Characteristic */
    {
        ATTRIB_FLAG_VALUE_INCL,                    /* wFlags */
        {                                        /* bTypeValue */
            LO_WORD(GATT_UUID_CHARACTERISTIC),
            HI_WORD(GATT_UUID_CHARACTERISTIC),
            GATT_CHAR_PROP_READ  |  GATT_CHAR_PROP_WRITE,  /*  characteristic
properties */
        },
        1,                                       /* bValueLen */
        NULL,
        GATT_PERM_READ                            /* wPermissions */
    },

    /* Alert Level Characteristic value  */
    {
        ATTRIB_FLAG_VALUE_APPL,                    /* wFlags */
        {                                        /* bTypeValue */
            LO_WORD(GATT_UUID_CHAR_ALERT_LEVEL),
            HI_WORD(GATT_UUID_CHAR_ALERT_LEVEL)
        },
        0,                                       /* variable size */
        NULL,
        GATT_PERM_READ | GATT_PERM_WRITE          /* wPermissions */
    }
};
```

当对测端通过 LLS 服务向防丢器发送报警级别时，App task 调用 LLS 的写回调函数，并在 AppHandleGATTCallback 进行处理。

当对测端通过 LLS 服务向防丢器读取报警级别时，App task 调用 LLS 的读回调函数，并在 AppHandleGATTCallback 进行处理。

```
else if (service_id == lls_srv_id)
{
  T_LLS_CALLBACK_DATA *p_lls_cb_data = (T_LLS_CALLBACK_DATA *)p_data;
  switch (p_lls_cb_data->msg_type)
  {
    case SERVICE CALLBACK TYPE WRITE CHAR VALUE:
     g pxp linkloss alert level = p lls cb data->msg data.write alert level;
          break;
    case SERVICE CALLBACK TYPE READ CHAR VALUE:
      lls set parameter(LLS PARAM LINK LOSS ALERT LEVEL,1,
                                    &g pxp linkloss alert level);
          break;
     default:
          break;
  }
}
```

当发生 Link Loss（断线时），状态转换相应的函数，就会判断链路丢失的原因，并根据之前设定的报警级别，启动相应的报警，并重新启动广播。

```
case GAP_CONN_STATE_DISCONNECTED:
       {
          memset(&app_link_table[conn_id], 0, sizeof(T_APP_LINK));
          if ((disc_cause != (HCI_ERR | HCI_ERR_REMOTE_USER_TERMINATE))
              && (disc_cause != (HCI_ERR | HCI_ERR_LOCAL_HOST_TERMINATE)))
          {
              APP_PRINT_ERROR2("app_handle_conn_state_evt: connection lost,
conn_id %d, cause 0x%x", conn_id,
                               disc_cause);
              if (gPxpState == PxpStateLinkM)
              {
                  gPxpState = PxpStateIdle;
                  PXPLINK[conn_id] = NoLink;
                  le_scan_start();
                  gMIoState = IoStateLlsAlert;

                  StartPxpMIO(ALERT_LOW_PERIOD, ALERT_HIGH_PERIOD, LED_BLINK
| BEEP_ALERT,
                              0xffffffff); /*low period 0.9s, high period 0.1s,
led blink,  10times(cnt)*/
              }
              else if (gPxpState == PxpStateLinkMADV)
              {
                  gPxpState = PxpStateAdv;
                  PXPLINK[conn_id] = NoLink;
                  le_scan_start();
                  gMIoState = IoStateLlsAlert;
                  StartPxpMIO(ALERT_LOW_PERIOD, ALERT_HIGH_PERIOD, LED_BLINK
| BEEP_ALERT,
                              0xffffffff); /*low period 0.9s, high period 0.1s,
led blink,  10times(cnt)*/
```

```
            }
            else if (gPxpState == PxpStateLinkS)
            {
                gPxpState = PxpStateIdle;
                PXPLINK[conn_id] = NoLink;
                le_adv_start();
                gSIoState = IoStateLlsAlert;
                if (g_pxp_linkloss_alert_level == 2)
                {
                    StartPxpSIO(ALERT_LOW_PERIOD,        ALERT_HIGH_PERIOD,
LED_BLINK | BEEP_ALERT,
                                0xffffffff); /*low period 0.9s, high period
0.1s,  led blink,  10times(cnt)*/
                }
                if (g_pxp_linkloss_alert_level == 1)
                {
                    StartPxpSIO(ALERT_LOW_PERIOD,        ALERT_HIGH_PERIOD,
LED_BLINK,
                                0xffffffff); /*low period 0.9s, high period
0.1s,  led blink,  10times(cnt)*/
                }
            }
            else if (gPxpState == PxpStateLinkSScan)
            {
                gPxpState = PxpStateScan;
                PXPLINK[conn_id] = NoLink;
                le_adv_start();
                gSIoState = IoStateLlsAlert;
                if (g_pxp_linkloss_alert_level == 2)
                {
                    StartPxpSIO(ALERT_LOW_PERIOD,        ALERT_HIGH_PERIOD,
LED_BLINK | BEEP_ALERT,
                                0xffffffff); /*low period 0.9s, high period
0.1s,  led blink,  10times(cnt)*/
```

```
                }
                if (g_pxp_linkloss_alert_level == 1)
                {
                    StartPxpSIO(ALERT_LOW_PERIOD,        ALERT_HIGH_PERIOD,
LED_BLINK,
                                0xffffffff); /*low period 0.9s, high period
0.1s, led blink, 10times(cnt)*/
                }
            }
            else if (gPxpState == PxpStateLinkMS)
            {
                if (PXPLINK[conn_id] == MasterLink)
                {
                    PXPLINK[conn_id] = NoLink;
                    gPxpState = PxpStateLinkS;
                    le_scan_start();
                    gMIoState = IoStateLlsAlert;
                    StartPxpMIO(ALERT_LOW_PERIOD,        ALERT_HIGH_PERIOD,
LED_BLINK | BEEP_ALERT,
                                0xffffffff); /*low period 0.9s, high period
0.1s, led blink, 10times(cnt)*/
                }
                else if (PXPLINK[conn_id] == SlaveLink)
                {
                    PXPLINK[conn_id] = NoLink;
                    gPxpState = PxpStateLinkM; le_adv_start();
                    gSIoState = IoStateLlsAlert;
                    if (g_pxp_linkloss_alert_level == 2)
                    {
                        StartPxpSIO(ALERT_LOW_PERIOD,    ALERT_HIGH_PERIOD,
LED_BLINK | BEEP_ALERT,
                                    0xffffffff);  /*low  period  0.9s,  high
period 0.1s, led blink, 10times(cnt)*/
                    }
```

```
                    if (g_pxp_linkloss_alert_level == 1)
                    {
                        StartPxpSIO(ALERT_LOW_PERIOD,     ALERT_HIGH_PERIOD,
LED_BLINK,
                                    0xffffffff);  /*low  period  0.9s,  high
period 0.1s,  led blink,  10times(cnt)*/
                    }
                }//else if (PXPLINK[conn_id] == SlaveLink)
            }//else if (gPxpState == PxpStateLinkMS)
        }
        else
        {
            APP_PRINT_INFO0("TERMINATE LINK");
            if (gPxpState == PxpStateLinkM)
            {
                gPxpState = PxpStateIdle;
                PXPLINK[conn_id] = NoLink;
                StopPxpMIO();
            }
            else if (gPxpState == PxpStateLinkMADV)
            {
                gPxpState = PxpStateAdv;
                PXPLINK[conn_id] = NoLink;
                StopPxpMIO();
            }
            else if (gPxpState == PxpStateLinkS)
            {
                gPxpState = PxpStateIdle;
                PXPLINK[conn_id] = NoLink;
                StopPxpSIO();
            }
            else if (gPxpState == PxpStateLinkSScan)
            {
                gPxpState = PxpStateScan;
```

```
                PXPLINK[conn_id] = NoLink;
                StopPxpSIO();
            }
            else if (gPxpState == PxpStateLinkMS)
            {
                if (PXPLINK[conn_id] == MasterLink)
                {
                    PXPLINK[conn_id] = NoLink;
                    gPxpState = PxpStateLinkS;
                    StopPxpMIO();
                }
                else if (PXPLINK[conn_id] == SlaveLink)
                {
                    PXPLINK[conn_id] = NoLink;
                    gPxpState = PxpStateLinkM;
                    StopPxpSIO();
                }
            }
        }
    }       break;
```

8.3. Tx Power Service

Tx Power Service 是标准的 GATT Service，所有的功能和定义在 tps.c&tps.h 里实现。

Tx Power Service 只包含一个 Tx Power Service Characteristic，为只读的，如表 8.9 所示。

表 8.9 Tx Power Service Characteristic 列表

Characteristic Name	Requirement	Characteristic UUID	Properties	Description
Tx Power Level	M	0x2A07	Read	See Tx Power Level

Tx Power Level 表示当前的发射功率，单位 dBm，范围从-100dBm 至+20dBm，分辨率为 1dBm。数据格式为有符号 8 位整型，初始值为 0，如表 8.10 所示。

表 8.10 Tx Power Level Characteristic Value Format

Names	Field Requirement	Format	Minimum Value	Maximum Value	Additional Information
Tx Power Level	Mandatory	sint8	-100	20	none

目前在我们的程序中， 默认的 Tx 发射功率是 0dBm。

TPS 的 GATT Attribute Table 如下：

```
const T_ATTRIB_APPL tps_attr_tbl[] =
{
    /*----------------- TX Power Service -------------------*/
    {
        (ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_LE),  /* wFlags     */
        {                                       /* bTypeValue */
            LO_WORD(GATT_UUID_PRIMARY_SERVICE),
            HI_WORD(GATT_UUID_PRIMARY_SERVICE),
            LO_WORD(GATT_UUID_TX_POWER_SERVICE),    /* service UUID */
            HI_WORD(GATT_UUID_TX_POWER_SERVICE)
        },
        UUID_16BIT_SIZE,                          /* bValueLen     */
        NULL,                                    /* pValueContext */
        GATT_PERM_READ                            /* wPermissions  */
    },
    {
        ATTRIB_FLAG_VALUE_INCL,                     /* wFlags */
        {                                       /* bTypeValue */
            LO_WORD(GATT_UUID_CHARACTERISTIC),
            HI_WORD(GATT_UUID_CHARACTERISTIC),
            GATT_CHAR_PROP_READ,          /* characteristic properties */
        },
        1,                                      /* bValueLen */
        NULL,
        GATT_PERM_READ                            /* wPermissions */
    },
    {
        ATTRIB_FLAG_VALUE_APPL,                     /* wFlags */
        {                                       /* bTypeValue */
            LO_WORD(GATT_UUID_CHAR_TX_LEVEL),
            HI_WORD(GATT_UUID_CHAR_TX_LEVEL)
        },
        0,                                      /* variable size */
        NULL,
        GATT_PERM_READ                           /* wPermissions */
    }
};
```

当对测端通过 TPS 服务向防丢器读取 power level 时，App task 调用 TPS 的读回调函数，并在 AppHandleGATTCallback 进行处理。

```
    else if (service_id == tps_srv_id)
    {
        T_TPS_CALLBACK_DATA *p_tps_cb_data = (T_TPS_CALLBACK_DATA *)p_data;
        if(p_tps_cb_data->msg_type == SERVICE_CALLBACK_TYPE_READ_CHAR_VALUE)
        {
            if(p tps cb data->msg data.read value index==
TPS_READ_TX_POWER_VALUE)
```

```
            {
                uint8_t tps_value = 0;
                tps_set_parameter(TPS_PARAM_TX_POWER, 1, &tps_value);
            }
        }
    }
```

8.4. Battery Service

Battery Service 是标准的 GATT Service，所有的功能和定义在 bas.c&bas.h 里实现。

Battery Service 包含一个 Battery Level 的 Characteristic，如表 8.11 所示。

表 8.11 Battery Service Characteristic 列表

Characteristic Name	Requirement	Characteristic UUID	Properties	Description
Battery Level	M	0x2A19	Read/Notify	See Battery Level

Battery Level 表示当前电量水平，范围从 0%～100%，数据格式为无符号 8 位整型，如表 8.12 所示。

表 8.12 Battery Level Characteristic Value Format

Names	Field Requirement	Format	Minimum Value	Maximum Value	Additional Information
Battery Level	Mandatory	uint8	0	100	Enumerations Key: 101-255 — Vlaue: Reserved

BAS 的 GATT Attribute Table 如下：

```
static const T_ATTRIB_APPL bas_attr_tbl[] =
{
    /*----------------- Battery Service -------------------*/
    /* <<Primary Service>> */
    {
        (ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_LE),   /* flags     */
        {                                            /* type value */
            LO WORD(GATT UUID PRIMARY SERVICE),
            HI WORD(GATT UUID PRIMARY SERVICE),
            LO WORD(GATT UUID BATTERY),            /* service UUID */
            HI WORD(GATT UUID BATTERY)
        },
        UUID 16BIT SIZE,                          /* bValueLen     */
        NULL,                                    /* p_value_context */
        GATT_PERM_READ                           /* permissions  */
    },

    /* <<Characteristic>> */
```

```
    {
        ATTRIB_FLAG_VALUE_INCL,                  /* flags */
        {                                        /* type_value */
            LO_WORD(GATT_UUID_CHARACTERISTIC),
            HI_WORD(GATT_UUID_CHARACTERISTIC),
#if BAS_BATTERY_LEVEL_NOTIFY_SUPPORT
            (GATT_CHAR_PROP_READ |                          /* characteristic
properties */
             GATT_CHAR_PROP_NOTIFY)
#else
            GATT_CHAR_PROP_READ
#endif
  /* characteristic UUID not needed here, is UUID of next attrib. */
        },
        1,                                       /* bValueLen */
        NULL,
        GATT_PERM_READ                           /* permissions */
    },
    /* Battery Level value */
    {
        ATTRIB_FLAG_VALUE_APPL,                   /* flags */
        {                                        /* type_value */
            LO_WORD(GATT_UUID_CHAR_BAS_LEVEL),
            HI_WORD(GATT_UUID_CHAR_BAS_LEVEL)
        },
        0,                                    /* bValueLen */
        NULL,
        GATT_PERM_READ                           /* permissions */
    }
#if BAS_BATTERY_LEVEL_NOTIFY_SUPPORT
    ,
    /* client characteristic configuration */
    {
        ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_CCCD_APPL,   /* flags */
        {                                                 /* type_value */
            LO_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
            HI_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
    /* NOTE: this value has an instantiation for each client, a write to */
    /* this attribute does not modify this default value:             */
            LO_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT),
                                     /* client char. config. bit field */
            HI_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT)
        },
        2,                                    /* bValueLen */
        NULL,
        (GATT_PERM_READ | GATT_PERM_WRITE)        /* permissions */
    }
#endif
};
```

由于 BAS 的 notify 是可选的属性，所以在 attribute table 中用宏定义进行编译选择使能。

当对测端通过 BAS 服务向防丢器读取电量值时，App task 调用 BAS 的读回调函数，并在 AppHandleGATTCallback 进行处理。

如果 attribute table 中注册了电量的 notify 属性， 当对测端通过 BAS 服务向防丢器写使

能电池电量的通知功能，App task 调用 BAS 的写使能回调函数，并在 AppHandleGATTCallback 进行处理，写使能打开后， APP 程序可以启动一个定时器， 定期上报电池的电量。

```
    else if (service_id == bas_srv_id)
    {
        T_BAS_CALLBACK_DATA *p_bas_cb_data = (T_BAS_CALLBACK_DATA *)p_data;
        switch (p_bas_cb_data->msg_type)
        {
        case SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION:
            {
              switch(p_bas_cb_data->msg_data.notification_indification_index)
                {
                case BAS_NOTIFY_BATTERY_LEVEL_ENABLE:
                    {
                        APP_PRINT_INFO0("BAS_NOTIFY_BATTERY_LEVEL_ENABLE");
                    }
                    break;

                case BAS_NOTIFY_BATTERY_LEVEL_DISABLE:
                    {
                        APP_PRINT_INFO0("BAS_NOTIFY_BATTERY_LEVEL_DISABLE");
                    }
                    break;
                default:
                    break;
                }
            }
            break;

        case SERVICE_CALLBACK_TYPE_READ_CHAR_VALUE:
            {
              if(p_bas_cb_data->msg_data.read_value_index==
BAS_READ_BATTERY_LEVEL)
                {
                    uint8_t battery_level = 90;
                    APP_PRINT_INFO1("BAS_READ_BATTERY_LEVEL:
battery_level %d", battery_level);
                    bas_set_parameter(BAS_PARAM_BATTERY_LEVEL,              1,
&battery_level);
                }
            }
            break;
        default:
            break;
        }
    }
```

8.5. Device Information Service

Device Infomation Service 是标准的 GATT Service，所有的功能和定义在 dis.c&dis.h 里实现。

Device Infomation Service 定义了多个只读的 characteristic，而且都是可选的。客户可以通过宏定义进行选择自己所需的定义字段。

表 8.13 Device Information Service Characteristic 列表

Characteristic Name	Requirement	Characteristic UUID	Properties
Manufacturer Name String	O	0x2A29	Read
Model Number String	O	0x2A24	Read
Serial Number String	O	0x2A25	Read
Hardware Revision String	O	0x2A27	Read
Firmware Revision String	O	0x2A26	Read
Software Revision String	O	0x2A28	Read
System ID	O	0x2A23	Read
Regulatory Certification Data List	O	0x2A2A	Read
PnP ID	O	0x2A50	Read

Device Information Service 中包含一部分显示设备名称和固件版本等基本信息的 Characteristic，如表 8.14 所示。

表 8.14 Device Information Characteristic Value Format

Names	Field Requirement	Format	Minimum Value	Maximum Value	Additional Information
Manufacturer Name	Mandatory	utf8s	N/A	N/A	none
Model Number	Mandatory	utf8s	N/A	N/A	none
Serial Number	Mandatory	utf8s	N/A	N/A	none
Hardware Revision	Mandatory	utf8s	N/A	N/A	none
Firmware Revision	Mandatory	utf8s	N/A	N/A	none
Software Revision	Mandatory	utf8s	N/A	N/A	none

(1) System ID Characteristic

System ID 由两个字段组成，分别为 40bit 制造商定义的 ID 和 24bit 组织唯一标识符 (OUI)，如表 8.15 所示。

表 8.15 System ID Characteristic Value Format

Names	Field Requirement	Format	Minimum Value	Maximum Value	Additional Information
Manufacturer Identifier	Mandatory	uint40	0	1099511627775	none

Organization Unique Identifier	Mandatory	uint24	0	16777215	none

(2) IEEE 11073-20601 Regulatory Certification Data List Characteristic

IEEE 11073-20601 Regulatory Certification Data List 列举了设备依附的各种各样的管理或服从认证的项目，如表 8.16 所示。

表 8.16 IEEE 11073-20601 Regulatory Certification Data List Characteristic Value Format

Names	Field Requirement	Format	Minimum Value	Maximum Value	Additional Information
Data	Mandatory	reg-cert-data-list[2]	N/A	N/A	none

(3) PnP ID Characteristic

PnP ID 是一组用于创建唯一设备 ID 的数值，包括了 Vendor ID Source、Vendor ID、Product ID、Product Version，这些数值分别用来辨别具有给定的类型/模板/版本的所有设备，如表 8.17 所示。

表 8.17 PnP ID Characteristic Value Format

Names	Field Requirement	Format	Minimum Value	Maximum Value	Additional Information
Vendor ID Source	Mandatory	uint8	1	2	See Enumerations
Vendor ID	Mandatory	uint16	N/A	N/A	None
Product ID	Mandatory	uint16	N/A	N/A	None
Product Version	Mandatory	uint16	N/A	N/A	None

表 8.18 Vendor ID Enumerations

Key	1	2	3-255	0
Value	Bluetooth SIG assigned Company Identifier value from the Assigned	USB Implementer's Forum assigned Vendor ID Value	Reserved for future use	Reserved for future use

DIS 的 GATT Attribute Table 如下：

```
static const T_ATTRIB_APPL dis_attr_tbl[] =
{
```

```
    /*----------------- Device Information Service ------------------*/
    /* <<Primary Service>> */
    {
        (ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_LE), /* flags     */
        {                                   /* type_value */
            LO_WORD(GATT_UUID_PRIMARY_SERVICE),
            HI_WORD(GATT_UUID_PRIMARY_SERVICE),
            LO_WORD(GATT_UUID_DEVICE_INFORMATION_SERVICE),   /* service UUID
*/
            HI_WORD(GATT_UUID_DEVICE_INFORMATION_SERVICE)
        },
        UUID_16BIT_SIZE,                          /* bValueLen     */
        NULL,                                  /* p_value_context */
        GATT_PERM_READ                          /* permissions  */
    }

#if DIS_CHAR_MANUFACTURER_NAME_SUPPORT
    ,
    /* <<Characteristic>> */
    {
        ATTRIB_FLAG_VALUE_INCL,                     /* flags */
        {                                       /* type_value */
            LO_WORD(GATT_UUID_CHARACTERISTIC),
            HI_WORD(GATT_UUID_CHARACTERISTIC),
            GATT_CHAR_PROP_READ                               /* characteristic
properties */
            /* characteristic UUID not needed here, is UUID of next attrib.
*/
        },
        1,                                      /* bValueLen */
        NULL,
        GATT_PERM_READ                            /* permissions */
    },
    /* Manufacturer Name String characteristic value */
    {
        ATTRIB_FLAG_VALUE_APPL,                     /* flags */
        {                                       /* type_value */
            LO_WORD(GATT_UUID_CHAR_MANUFACTURER_NAME),
            HI_WORD(GATT_UUID_CHAR_MANUFACTURER_NAME)
        },
        0,                                      /* variable size */
        (void *)NULL,
        GATT_PERM_READ                            /* permissions */
    }
#endif

#if DIS CHAR MODEL NUMBER SUPPORT
    ,
    /* <<Characteristic>> */
    {
        ATTRIB FLAG VALUE INCL,                     /* flags */
        {                                       /* type_value */
            LO_WORD(GATT_UUID_CHARACTERISTIC),
            HI_WORD(GATT_UUID_CHARACTERISTIC),
            GATT_CHAR_PROP_READ                               /* characteristic
properties */
```

```
            /* characteristic UUID not needed here, is UUID of next attrib.
*/
        },
        1,                                  /* bValueLen */
        NULL,
        GATT_PERM_READ                        /* permissions */
    },
    /* Model Number characteristic value */
    {
        ATTRIB_FLAG_VALUE_APPL,                 /* flags */
        {                                   /* type_value */
            LO_WORD(GATT_UUID_CHAR_MODEL_NUMBER),
            HI_WORD(GATT_UUID_CHAR_MODEL_NUMBER)
        },
        0,                                  /* variable size */
        (void *)NULL,
        GATT_PERM_READ                        /* permissions */
    }
#endif

#if DIS_CHAR_SERIAL_NUMBER_SUPPORT
    ,
    /* <<Characteristic>> */
    {
        ATTRIB_FLAG_VALUE_INCL,                 /* flags */
        {                                   /* type_value */
            LO_WORD(GATT_UUID_CHARACTERISTIC),
            HI_WORD(GATT_UUID_CHARACTERISTIC),
            GATT_CHAR_PROP_READ                 /* characteristic properties
*/
            /* characteristic UUID not needed here, is UUID of next attrib.
*/
        },
        1,                                  /* bValueLen */
        NULL,
        GATT_PERM_READ                        /* permissions */
    },
    /* Serial Number String String characteristic value */
    {
        ATTRIB_FLAG_VALUE_APPL,                 /* flags */
        {                                   /* type_value */
            LO_WORD(GATT_UUID_CHAR_SERIAL_NUMBER),
            HI_WORD(GATT_UUID_CHAR_SERIAL_NUMBER)
        },
        0,                                  /* variable size */
        (void *)NULL,
        GATT PERM READ                        /* permissions */
    }
#endif

#if DIS CHAR HARDWARE REVISION SUPPORT
    ,
    /* <<Characteristic>> */
    {
        ATTRIB_FLAG_VALUE_INCL,                 /* flags */
        {                                   /* type_value */
```

```
            LO_WORD(GATT_UUID_CHARACTERISTIC),
            HI_WORD(GATT_UUID_CHARACTERISTIC),
            GATT_CHAR_PROP_READ                  /* characteristic properties
*/
            /* characteristic UUID not needed here, is UUID of next attrib.
*/
        },
        1,                                    /* bValueLen */
        NULL,
        GATT_PERM_READ                          /* permissions */
    },
    /* Manufacturer Name String characteristic value */
    {
        ATTRIB_FLAG_VALUE_APPL,                  /* flags */
        {                                     /* type_value */
            LO_WORD(GATT_UUID_CHAR_HARDWARE_REVISION),
            HI_WORD(GATT_UUID_CHAR_HARDWARE_REVISION)
        },
        0,                                    /* variable size */
        (void *)NULL,
        GATT_PERM_READ                          /* permissions */
    }
#endif

#if DIS_CHAR_FIRMWARE_REVISION_SUPPORT
    ,
    /* <<Characteristic>> */
    {
        ATTRIB_FLAG_VALUE_INCL,                  /* flags */
        {                                     /* type_value */
            LO_WORD(GATT_UUID_CHARACTERISTIC),
            HI_WORD(GATT_UUID_CHARACTERISTIC),
            GATT_CHAR_PROP_READ                  /* characteristic properties
*/
            /* characteristic UUID not needed here, is UUID of next attrib.
*/
        },
        1,                                    /* bValueLen */
        NULL,
        GATT_PERM_READ                          /* permissions */
    },
    /* Firmware revision String characteristic value */
    {
        ATTRIB_FLAG_VALUE_APPL,                  /* flags */
        {                                     /* type value */
            LO WORD(GATT UUID CHAR FIRMWARE REVISION),
            HI WORD(GATT UUID CHAR FIRMWARE REVISION)
        },
        0,                                    /* variable size */
        (void *)NULL,
        GATT PERM READ                          /* permissions */
    }
#endif

#if DIS_CHAR_SOFTWARE_REVISION_SUPPORT
    ,
```

```
    /* <<Characteristic>> */
    {
        ATTRIB_FLAG_VALUE_INCL,                 /* flags */
        {                                       /* type_value */
            LO_WORD(GATT_UUID_CHARACTERISTIC),
            HI_WORD(GATT_UUID_CHARACTERISTIC),
            GATT_CHAR_PROP_READ                 /* characteristic properties
*/
            /* characteristic UUID not needed here, is UUID of next attrib.
*/
        },
        1,                                      /* bValueLen */
        NULL,
        GATT_PERM_READ                          /* permissions */
    },
    /* Manufacturer Name String characteristic value */
    {
        ATTRIB_FLAG_VALUE_APPL,                 /* flags */
        {                                       /* type_value */
            LO_WORD(GATT_UUID_CHAR_SOFTWARE_REVISION),
            HI_WORD(GATT_UUID_CHAR_SOFTWARE_REVISION)
        },
        0,                                      /* variable size */
        (void *)NULL,
        GATT_PERM_READ                          /* permissions */
    }
#endif

#if DIS_CHAR_SYSTEM_ID_SUPPORT
    ,
    /* <<Characteristic>> */
    {
        ATTRIB_FLAG_VALUE_INCL,                 /* flags */
        {                                       /* type_value */
            LO_WORD(GATT_UUID_CHARACTERISTIC),
            HI_WORD(GATT_UUID_CHARACTERISTIC),
            GATT_CHAR_PROP_READ                             /* characteristic
properties */
            /* characteristic UUID not needed here, is UUID of next attrib.
*/
        },
        1,                                      /* bValueLen */
        NULL,
        GATT_PERM_READ                          /* permissions */
    },
    /* System ID String characteristic value */
    {
        ATTRIB FLAG VALUE APPL,                 /* flags */
        {                                       /* type value */
            LO WORD(GATT UUID CHAR SYSTEM ID),
            HI WORD(GATT UUID CHAR SYSTEM ID)
        },
        0,                                      /* variable size */
        (void *)NULL,
        GATT_PERM_READ                          /* permissions */
    }
```

```
#endif

#if DIS_CHAR_IEEE_CERTIF_DATA_LIST_SUPPORT
    ,

    /* <<Characteristic>> */
    {
        ATTRIB_FLAG_VALUE_INCL,                 /* flags */
        {                                       /* type_value */
            LO_WORD(GATT_UUID_CHARACTERISTIC),
            HI_WORD(GATT_UUID_CHARACTERISTIC),
            GATT_CHAR_PROP_READ                 /* characteristic properties
*/
            /* characteristic UUID not needed here, is UUID of next attrib.
*/
        },
        1,                                      /* bValueLen */
        NULL,
        GATT_PERM_READ                          /* permissions */
    },
    /* Manufacturer Name String characteristic value */
    {
        ATTRIB_FLAG_VALUE_APPL,                 /* flags */
        {                                       /* type_value */
            LO_WORD(GATT_UUID_CHAR_IEEE_CERTIF_DATA_LIST),
            HI_WORD(GATT_UUID_CHAR_IEEE_CERTIF_DATA_LIST)
        },
        0,                                      /* variable size */
        (void *)NULL,
        GATT_PERM_READ                          /* permissions */
    }
#endif

#if DIS_CHAR_PNP_ID_SUPPORT
    ,
    /* <<Characteristic>> */
    {
        ATTRIB_FLAG_VALUE_INCL,                 /* flags */
        {                                       /* type_value */
            LO_WORD(GATT_UUID_CHARACTERISTIC),
            HI_WORD(GATT_UUID_CHARACTERISTIC),
            GATT_CHAR_PROP_READ                 /* characteristic properties
*/
            /* characteristic UUID not needed here, is UUID of next attrib.
*/
        },
        1,                                      /* bValueLen */
        NULL,
        GATT PERM READ                          /* permissions */
    },
    /* Manufacturer Name String characteristic value */
    {
        ATTRIB_FLAG_VALUE_APPL,                 /* flags */
        {                                       /* type_value */
            LO_WORD(GATT_UUID_CHAR_PNP_ID),
            HI_WORD(GATT_UUID_CHAR_PNP_ID)
```

```
        },
        0,                                      /* variable size */
        (void *)NULL,
        GATT_PERM_READ                            /* permissions */
    }
#endif
};
```

当对测端通过 DIS 服务向防丢器读取设备信息时，App task 调用 DIS 的读回调函数，并在 AppHandleGATTCallback 进行处理。

```
else if (service_id == dis_srv_id)
    {
        T_DIS_CALLBACK_DATA *p_dis_cb_data = (T_DIS_CALLBACK_DATA *)p_data;
        switch (p_dis_cb_data->msg_type)
        {
        case SERVICE_CALLBACK_TYPE_READ_CHAR_VALUE:
            {
                if        (p_dis_cb_data->msg_data.read_value_index        ==
DIS_READ_MANU_NAME_INDEX)
                {
                    const uint8_t DISManufacturerName[] = "Realtek BT";
                    dis_set_parameter(DIS_PARAM_MANUFACTURER_NAME,
                                      sizeof(DISManufacturerName),
                                      (void *)DISManufacturerName);

                }
                else    if    (p_dis_cb_data->msg_data.read_value_index    ==
DIS_READ_MODEL_NUM_INDEX)
                {
                    const uint8_t DISModelNumber[] = "Model Nbr 0.9";
                    dis_set_parameter(DIS_PARAM_MODEL_NUMBER,
                                      sizeof(DISModelNumber),
                                      (void *)DISModelNumber);
                }
                else    if    (p_dis_cb_data->msg_data.read_value_index    ==
DIS_READ_SERIAL_NUM_INDEX)
                {
                    const uint8_t DISSerialNumber[] = "RTKBeeSerialNum";
                    dis_set_parameter(DIS_PARAM_SERIAL_NUMBER,
                                      sizeof(DISSerialNumber),
                                      (void *)DISSerialNumber);

                }
                else    if    (p_dis_cb_data->msg_data.read_value_index    ==
DIS READ HARDWARE REV INDEX)
                {
                    const uint8 t DISHardwareRev[] = "RTKBeeHardwareRev";
                    dis set parameter(DIS PARAM HARDWARE REVISION,
                                      sizeof(DISHardwareRev),
                                      (void *)DISHardwareRev);
                }
                else    if    (p_dis_cb_data->msg_data.read_value_index    ==
DIS_READ_FIRMWARE_REV_INDEX)
                {
                    const uint8_t DISFirmwareRev[] = "RTKBeeFirmwareRev";
```

```
                dis_set_parameter(DIS_PARAM_FIRMWARE_REVISION,
                                  sizeof(DISFirmwareRev),
                                  (void *)DISFirmwareRev);
            }
            else    if    (p_dis_cb_data->msg_data.read_value_index    ==
DIS_READ_SOFTWARE_REV_INDEX)
            {
                const uint8_t DISSoftwareRev[] = "RTKBeeSoftwareRev";
                dis_set_parameter(DIS_PARAM_SOFTWARE_REVISION,
                                  sizeof(DISSoftwareRev),
                                  (void *)DISSoftwareRev);
            }
            else    if    (p_dis_cb_data->msg_data.read_value_index    ==
DIS_READ_SYSTEM_ID_INDEX)
            {
                const uint8_t DISSystemID[DIS_SYSTEM_ID_LENGTH] = {0, 1, 2,
0, 0, 3, 4, 5};
                dis_set_parameter(DIS_PARAM_SYSTEM_ID,
                                  sizeof(DISSystemID),
                                  (void *)DISSystemID);

            }
            else    if    (p_dis_cb_data->msg_data.read_value_index    ==
DIS_READ_IEEE_CERT_STR_INDEX)
            {
                const uint8_t DISIEEEDataList[] = "RTKBeeIEEEDatalist";
                dis_set_parameter(DIS_PARAM_IEEE_DATA_LIST,
                                  sizeof(DISIEEEDataList),
                                  (void *)DISIEEEDataList);
            }
            else    if    (p_dis_cb_data->msg_data.read_value_index    ==
DIS_READ_PNP_ID_INDEX)
            {
                uint8_t DISPnpID[DIS_PNP_ID_LENGTH] = {0};
                dis_set_parameter(DIS_PARAM_PNP_ID,
                                  sizeof(DISPnpID),
                                  DISPnpID);
            }


        }
        break;
    default:
        break;
    }
}
```

8.6. Key Notification Service

Key Notification Service 是自定义的 GATT Service，是防丢器私有的，所有的功能和定义在 kns.c&kns.h 里实现。

Key Notification Service 定义了两个 characteristic，一个为可读写的， 用来配置 link loss 和 immediately alert 的报警次数，如表 8.19 所示。

另外一个为通知属性， 用来向 master 发送 alert 消息，如表 8.19 所示。

表 8.19 Key Notification Service Characteristic 列表

Characteristic Name	Requirement	Characteristic UUID	Properties	Description
Set Alert time	M	0x0000FFD1-BB29-456D-989D-C44D07F6F6A6	Read/Write	See Set Alert Level
Key Value	M	0x0000FFD2-BB29-456D-989D-C44D07F6F6A6	Notify	See Key Value

Set Alert Time Characteristic

Set Alert Time 为自定义的设置时间的 Characteristic。数据格式为 32 位无符号整型，初始值为 30，单位为秒，最小值为 0，最大值为 0xffffffff，如表 8.20 所示。

表 8.20 Set Alert Time Characteristic Value Format

Names	Field Requirement	Format	Minimum Value	Maximum Value	Additional Information
Set Alert Time	Mandatory	uint32	0	0xffffffff	none

Key Value Characteristic

Key Value 表示按键信息，数据格式为 8 位无符号整型。在连线的情况下按下发送 value 1 到 master 端 ，如表 8.21 所示。

表 8.21 Key Value Characteristic Value Format

Names	Field Requirement	Format	Value	Additional Information
Key Value	Mandatory	uint8	0	None

KNS 的 GATT Attribute Table 如下：

```
static const T_ATTRIB_APPL kns_attr_tbl[] =
{
    /*----------------- simple key Service -------------------*/
    /* <<Primary Service>>, .. */
    {
        (ATTRIB_FLAG_VOID | ATTRIB_FLAG_LE),    /* wFlags     */
        {   /* bTypeValue */
            LO_WORD(GATT_UUID_PRIMARY_SERVICE),
            HI_WORD(GATT_UUID_PRIMARY_SERVICE),
        },
        UUID_128BIT_SIZE,                       /* bValueLen     */
```

```
       (void *)GATT_UUID128_KNS_SERVICE,          /* pValueContext */
       GATT_PERM_READ                           /* wPermissions  */
   },

   /* Set para Characteristic */
   {
       ATTRIB_FLAG_VALUE_INCL,                   /* wFlags */
       {   /* bTypeValue */
           LO_WORD(GATT_UUID_CHARACTERISTIC),
           HI_WORD(GATT_UUID_CHARACTERISTIC),
           GATT_CHAR_PROP_READ  |  GATT_CHAR_PROP_WRITE,                    /*
characteristic properties */
       },
       1,                                     /* bValueLen */
       NULL,
       GATT_PERM_READ                           /* wPermissions */
   },

   /* Set para Characteristic value  */
   {
       ATTRIB_FLAG_VALUE_APPL | ATTRIB_FLAG_UUID_128BIT,                    /*
wFlags */
       {   /* bTypeValue */
           GATT_UUID128_CHAR_PARAM
       },
       0,                                     /* variable size */
       NULL,
       GATT_PERM_READ  |  GATT_PERM_WRITE                                   /*
wPermissions */
   },

   /* Key  <<Characteristic>>, .. */
   {
       ATTRIB_FLAG_VALUE_INCL,                   /* wFlags */
       {   /* bTypeValue */
           LO_WORD(GATT_UUID_CHARACTERISTIC),
           HI_WORD(GATT_UUID_CHARACTERISTIC),
           (             /* characteristic properties */
              GATT_CHAR_PROP_NOTIFY)
           /* characteristic UUID not needed here, is UUID of next attrib.
*/
       },
       1,                                     /* bValueLen */
       NULL,
       GATT PERM READ                           /* wPermissions */
   },
   /* simple key value */
   {
       ATTRIB FLAG VALUE APPL | ATTRIB FLAG UUID 128BIT,                    /*
wFlags */
       {   /* bTypeValue */
           GATT_UUID128_CHAR_KEY
       },
       0,                                     /* bValueLen */
       NULL,
       GATT_PERM_READ                           /* wPermissions */
```

```
    },
    /* client characteristic configuration */
    {
        ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_CCCD_APPL,                    /*
wFlags */
        {   /* bTypeValue */
            LO_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
            HI_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
            /* NOTE: this value has an instantiation for each client, a write
to */
            /*   this   attribute   does   not   modify   this   default   value:
*/
            LO_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT), /* client char. config.
bit field */
            HI_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT)
        },
        2,                                      /* bValueLen */
        NULL,
        (GATT_PERM_READ | GATT_PERM_WRITE)         /* wPermissions */
    }
};
```

当对测端通过 KNS 服务向防丢器设置或者读写参数是（link loss and immediately alert 报警的次数），App task 调用 KNS 的读回调函数，并在 AppHandleGATTCallback 进行处理。

当对测端通过 KNS 服务向防丢器写使能按键报警的通知功能，App task 调用 KNS 的写使能回调函数，并在 AppHandleGATTCallback 进行处理，写使能打开后， 在连线状态时，用户短按按键， 防丢器向 master 发送报警的 notification。

```
    else  if (service_id == kns_srv_id)
    {
        T_KNS_CALLBACK_DATA    *p_kns_cb_data    =    (T_KNS_CALLBACK_DATA
*)p_data;
        switch (p_kns_cb_data->msg_type)
        {
        case SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION:
            {
                switch
(p_kns_cb_data->msg_data.notification_indification_index)
                {
                case KNS_NOTIFY_ENABLE:
                    {
                        APP_PRINT_INFO0("KNS_NOTIFY_ENABLE");
                    }
                    break;

                case KNS_NOTIFY_DISABLE:
                    {
                        APP_PRINT_INFO0("KNS_NOTIFY_DISABLE");
                    }
                    break;
                default:
                    break;
                }
            }
```

```
            break;

        case SERVICE_CALLBACK_TYPE_READ_CHAR_VALUE:
            {
                if (p_kns_cb_data->msg_data.read_index == KNS_READ_PARA)
                {
                    APP_PRINT_INFO1("KNS_READ_PARA,gTimeParaValue  is  %x",
gTimeParaValue);
                    kns_set_parameter(KNS_PARAM_VALUE,                    4,
&gTimeParaValue);
                }
            }
            break;
        case SERVICE_CALLBACK_TYPE_WRITE_CHAR_VALUE:
            {
                gTimeParaValue = p_kns_cb_data->msg_data.write_value;
            }
            break;

        default:
            break;
        }
    }
```

9. Central Application

9.1. Immediately Alert Client

Immediately Alert Client Callbacks 函数注册。

```
const T_FUN_CLIENT_CBS ias_client_cbs =
{
    ias_client_discover_state_cb,       //!<  Discovery  State  callback
function pointer
    ias_client_discover_result_cb,     //!<  Discovery  result  callback
function pointer
    NULL,                             //!< Read response callback function
pointer
    ias_client_write_result_cb,      //!< Write result callback function
pointer
    NULL,                           //!< Notify Indicate callback function
pointer
    ias_client_disconnect_cb           //!< Link disconnection callback
function pointer
};
```

client_write_char 函数发送报警值。

```
bool  ias_client_write_char(uint8_t  conn_id,  uint16_t  length,  uint8_t
*p_value, T_GATT_WRITE_TYPE type)
```

9.2. Link Loss Alert Client

Link Loss Alert Client Callbacks 函数注册。

```
const T_FUN_CLIENT_CBS lls_client_cbs =
{
    lls_client_discover_state_cb,       //!<  Discovery  State  callback
function pointer
    lls_client_discover_result_cb,     //!<  Discovery  result  callback
function pointer
    lls_client_read_result_cb,      //!< Read response callback function
pointer
    lls_client_write_result_cb,      //!< Write result callback function
pointer
    NULL,//kns_client_notif_ind_result_cb,      //!<   Notify   Indicate
callback function pointer
    lls_client_disconnect_cb           //!< Link disconnection callback
function pointer
};
```

lls_client_read_by_handle 获取 server Alert Level，得到 response 后，

调用 lls_client_read_result_cb 回调函数。

cb_data.cb_content.read_result.data.v1_read.p_value= p_value; 读取 Alert Level 值。

```
static void lls_client_read_result_cb(uint8_t conn_id,  uint16_t cause,
                                      uint16_t  handle,  uint16_t  value_size,
uint8_t *p_value)
{
    T_LLS_CLIENT_CB_DATA cb_data;
    uint16_t *hdl_cache;
    hdl_cache = lls_table[conn_id].hdl_cache;

    cb_data.cb_type = LLS_CLIENT_CB_TYPE_READ_RESULT;

    APP_PRINT_INFO2("lls_client_read_result_cb:   handle   0x%x,   cause
0x%x", handle, cause);
    cb_data.cb_content.read_result.cause = cause;

    if (handle == hdl_cache[HDL_LLS_PARA])
    {
        cb_data.cb_content.read_result.type = LLS_READ_PARA;
        if (cause == GAP_SUCCESS)
        {
            cb_data.cb_content.read_result.data.v1_read.p_value
                                                              = p_value;
            cb_data.cb_content.read_result.data.v1_read.value_size      =
value_size;
        }
        else
        {
            cb_data.cb_content.read_result.data.v1_read.value_size = 0;
        }
    }
    else
    {
        return;
    }
    /* Inform application the read result. */
    if (lls_client_cb)
    {
        (*lls_client_cb)(lls_client, conn_id, &cb_data);
    }

    return;
}
```

9.3. Tx Power Client

Tx Power Client Callbacks 函数注册。

```
const T_FUN_CLIENT_CBS tps_client_cbs =
{
    tps_client_discover_state_cb,       //!<  Discovery  State  callback
function pointer
    tps_client_discover_result_cb,     //!<  Discovery  result  callback
function pointer
```

```
    tps_client_read_result_cb,      //!< Read response callback function
pointer
    NULL,     //!< Write result callback function pointer
    NULL,       //!< Notify Indicate callback function pointer
    tps_client_disc_cb                  //!< Link disconnection callback
function pointer
};
```

利用 tps_read_power_level(uint8_t conn_id)函数读取 server 发射功率，获得 response 后，调用 tps_client_read_result_cb 回调函数。

cb_data.cb_content.read_result.data.txpower_level = *p_value; response 的发射功率赋值给 txpower_level。

```
 static void tps_client_read_result_cb(uint8_t conn_id,  uint16_t cause,
                                uint16_t  handle,  uint16_t  value_size,
uint8_t *p_value)
{
    T_TPS_CLIENT_CB_DATA cb_data;
    uint16_t *hdl_cache;
    hdl_cache = tps_table[conn_id].hdl_cache;
    cb_data.cb_type = TPS_CLIENT_CB_TYPE_READ_RESULT;

    PROFILE_PRINT_INFO2("tps_client_read_result_cb:  handle  0x%x,  cause
0x%x", handle, cause);
    cb_data.cb_content.read_result.cause = cause;

    if (handle == hdl_cache[HDL_TPS_PARA])
    {
        cb_data.cb_content.read_result.type = TPS_READ_PARA;
        if (cause == GAP_SUCCESS)
        {
            if (value_size != 1)
            {
                PROFILE_PRINT_ERROR1("tps_client_read_result_cb:   invalid
battery value len %d", value_size);
                return;
            }
            cb_data.cb_content.read_result.data.txpower_level = *p_value;
        }
    }
    else
    {
        return;
    }

    if (tps client cb)
    {
        (*tps client cb)(tps client, conn id, &cb data);
    }
    return;
}
```

9.4. Battery Client

Battery Client Callbacks 函数注册。

```
    const T_FUN_CLIENT_CBS bas_client_cbs =
{
    bas_client_discover_state_cb,       //!<  Discovery  State  callback
function pointer
    bas_client_discover_result_cb,     //!<  Discovery  result  callback
function pointer
    bas_client_read_result_cb,      //!< Read response callback function
pointer
    bas_client_write_result_cb,     //!< Write result callback function
pointer
    bas_client_notify_ind_cb,            //!<  Notify  Indicate  callback
function pointer
    bas_client_disc_cb                  //!< Link disconnection callback
function pointer
};
```

bas_client_read_result_cb，为 read request responsed 回调函数。bas_read_battery_level(); 读取 server 电量，server response 后调用 bas_client_read_result_cb 回调函数。cb_data.cb_content.read_result.data.battery_level = *p_value; 获取当前电池电量。(*bas_client_cb)(bas_client, conn_id, &cb_data);调用注册的回调函数。

```
static void bas_client_read_result_cb(uint8_t conn_id,  uint16_t cause,
                uint16_t handle, uint16_t value_size,uint8_t *p_value)
{
    T_BAS_CLIENT_CB_DATA cb_data;
    uint16_t *hdl_cache;
    hdl_cache = bas_table[conn_id].hdl_cache;
    cb_data.cb_type = BAS_CLIENT_CB_TYPE_READ_RESULT;

    PROFILE_PRINT_INFO2("bas_client_read_result_cb:  handle  0x%x,  cause
0x%x", handle, cause);
    cb_data.cb_content.read_result.cause = cause;

    if (handle == hdl_cache[HDL_BAS_BATTERY_LEVEL_CCCD])
    {
        cb_data.cb_content.read_result.type = BAS_READ_NOTIFY;
        if (cause == GAP_SUCCESS)
        {
            uint16_t ccc_bit;
            if (value_size != 2)
            {
                PROFILE_PRINT_ERROR1("bas_client_read_result_cb:   invalid
cccd len %d", value_size);
                return;
            }
            LE ARRAY TO UINT16(ccc bit, p value);

            if (ccc bit & GATT CLIENT CHAR CONFIG NOTIFY)
            {
                cb_data.cb_content.read_result.data.notify = true;
```

```
            }
            else
            {
                cb_data.cb_content.read_result.data.notify = false;
            }
        }
    }
    else if (handle == hdl_cache[HDL_BAS_BATTERY_LEVEL])
    {
        cb_data.cb_content.read_result.type = BAS_READ_BATTERY_LEVEL;
        if (cause == GAP_SUCCESS)
        {
            if (value_size != 1)
            {
                PROFILE_PRINT_ERROR1("bas_client_read_result_cb:   invalid
battery value len %d", value_size);
                return;
            }
            cb_data.cb_content.read_result.data.battery_level = *p_value;
        }
    }
    else
    {
        return;
    }

    if (bas_client_cb)
    {
        (*bas_client_cb)(bas_client, conn_id, &cb_data);
    }
    return;
}
```

Server 通知电量时，调用 bas_client_notify_ind_cb Notify 回调函数，

cb_data.cb_content.notify_data.battery_level = *p_value;获得 notify 电量值。

bas_set_notify(conn_id, true)控制 notification flag

```
static  T_APP_RESULT  bas_client_notify_ind_cb(uint8_t  conn_id,  bool
notify, uint16_t handle,
                                    uint16_t    value_size,    uint8_t
*p_value)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    T_BAS_CLIENT_CB_DATA cb_data;
    uint16_t *hdl_cache;

    hdl_cache = bas_table[conn_id].hdl_cache;
    cb_data.cb_type = BAS_CLIENT_CB_TYPE_NOTIF_IND_RESULT;

    if (handle == hdl_cache[HDL_BAS_BATTERY_LEVEL])
    {
        cb_data.cb_content.notify_data.battery_level = *p_value;
    }
    else
    {
        return APP_RESULT_SUCCESS;
```

```
    }

    if (bas_client_cb)
    {
        app_result = (*bas_client_cb)(bas_client, conn_id, &cb_data);
    }

    return app_result;
}
```

9.5. Device Information Client

Device Information Client Callbacks 函数注册。

```
* @brief Ias BLE Client Callbacks.
*/
const T_FUN_CLIENT_CBS dis_client_cbs =
{
    dis_client_discover_state_cb,       //!<  Discovery  State  callback
function pointer
    dis_client_discover_result_cb,     //!<  Discovery  result  callback
function pointer
    dis_client_read_result_cb,      //!< Read response callback function
pointer
    NULL,//lls_client_write_result_cb,       //!< Write result callback
function pointer
    NULL,//kns_client_notif_ind_result_cb,      //!<   Notify   Indicate
callback function pointer
    dis_client_disconnect_cb           //!< Link disconnection callback
function pointer
};
```

bas_client_read_result_cb，为 read request responsed 回调函数，读回设备信息。

```
static void dis_client_read_result_cb(uint8_t conn_id,  uint16_t cause,
                                uint16_t  handle,  uint16_t  value_size,
uint8_t *p_value)
{
    T_DIS_CLIENT_CB_DATA cb_data;
    uint16_t *hdl_cache;
    hdl_cache = dis_table[conn_id].hdl_cache;

    cb_data.cb_type = DIS_CLIENT_CB_TYPE_READ_RESULT;

    APP_PRINT_INFO2("dis_client_read_result_cb:   handle   0x%x,   cause
0x%x", handle, cause);
    cb_data.cb_content.read_result.cause = cause;

    if (handle == hdl_cache[HDL_DIS_SYSTEM_ID])
    {
        cb_data.cb_content.read_result.type = DIS_READ_SYSTEM_ID;
        if (cause == GAP_SUCCESS)
        {
            cb_data.cb_content.read_result.data.v1_read.p_value         =
```

```
p_value;
            cb_data.cb_content.read_result.data.v1_read.value_size      =
value_size;
        }
        else
        {
            cb_data.cb_content.read_result.data.v1_read.value_size = 0;
        }
    }
    else if (handle == hdl_cache[HDL_DIS_MODEL_NUMBER])
    {
        cb_data.cb_content.read_result.type = DIS_READ_MODEL_NUMBER;
        if (cause == GAP_SUCCESS)
        {
            cb_data.cb_content.read_result.data.v1_read.p_value         =
p_value;
            cb_data.cb_content.read_result.data.v1_read.value_size      =
value_size;
        }
        else
        {
            cb_data.cb_content.read_result.data.v1_read.value_size = 0;
        }
    }
    else if (handle == hdl_cache[HDL_DIS_SERIAL_NUMBER])
    {
        cb_data.cb_content.read_result.type = DIS_READ_SERIAL_NUMBER;
        if (cause == GAP_SUCCESS)
        {
            cb_data.cb_content.read_result.data.v1_read.p_value         =
p_value;
            cb_data.cb_content.read_result.data.v1_read.value_size      =
value_size;
        }
        else
        {
            cb_data.cb_content.read_result.data.v1_read.value_size = 0;
        }
    }
    else if (handle == hdl_cache[HDL_DIS_FIRMWARE_REVISION])
    {
        cb_data.cb_content.read_result.type                             =
DIS_READ_FIRMWARE_REVISION;
        if (cause == GAP_SUCCESS)
        {
            cb data.cb content.read result.data.v1 read.p value         =
p value;
            cb data.cb content.read result.data.v1 read.value size      =
value size;
        }
        else
        {
            cb_data.cb_content.read_result.data.v1_read.value_size = 0;
        }
    }
    else if (handle == hdl_cache[HDL_DIS_HARDWARE_REVISION])
```

```
    {
        cb_data.cb_content.read_result.type                                 =
DIS_READ_HARDWARE_REVISION;
        if (cause == GAP_SUCCESS)
        {
            cb_data.cb_content.read_result.data.v1_read.p_value             =
p_value;
            cb_data.cb_content.read_result.data.v1_read.value_size          =
value_size;
        }
        else
        {
            cb_data.cb_content.read_result.data.v1_read.value_size = 0;
        }
    }
    else if (handle == hdl_cache[HDL_DIS_SOFTWARE_REVISION])
    {
        cb_data.cb_content.read_result.type                                 =
DIS_READ_SOFTWARE_REVISION;
        if (cause == GAP_SUCCESS)
        {
            cb_data.cb_content.read_result.data.v1_read.p_value             =
p_value;
            cb_data.cb_content.read_result.data.v1_read.value_size          =
value_size;
        }
        else
        {
            cb_data.cb_content.read_result.data.v1_read.value_size = 0;
        }
    }
    else if (handle == hdl_cache[HDL_DIS_MANUFACTURER_NAME])
    {
        cb_data.cb_content.read_result.type                                 =
DIS_READ_MANUFACTURER_NAME;
        if (cause == GAP_SUCCESS)
        {
            cb_data.cb_content.read_result.data.v1_read.p_value             =
p_value;
            cb_data.cb_content.read_result.data.v1_read.value_size          =
value_size;
        }
        else
        {
            cb_data.cb_content.read_result.data.v1_read.value_size = 0;
        }
    }
    else if (handle == hdl_cache[HDL_DIS_IEEE_CERTIF_DATA_LIST])
    {
        cb_data.cb_content.read_result.type                                 =
DIS_READ_IEEE_CERTIF_DATA_LIST;
        if (cause == GAP_SUCCESS)
        {
            cb_data.cb_content.read_result.data.v1_read.p_value             =
p_value;
            cb_data.cb_content.read_result.data.v1_read.value_size          =
```

```
value_size;
        }
        else
        {
            cb_data.cb_content.read_result.data.v1_read.value_size = 0;
        }
    }
    else if (handle == hdl_cache[HDL_DIS_PNP_ID])
    {
        cb_data.cb_content.read_result.type = DIS_READ_PNP_ID;
        if (cause == GAP_SUCCESS)
        {
            cb_data.cb_content.read_result.data.v1_read.p_value         =
p_value;
            cb_data.cb_content.read_result.data.v1_read.value_size      =
value_size;
        }
        else
        {
            cb_data.cb_content.read_result.data.v1_read.value_size = 0;
        }
    }
    else
    {
        return;
    }
    /* Inform application the read result. */
    if (dis_client_cb)
    {
        (*dis_client_cb)(dis_client, conn_id, &cb_data);
    }

    return;
}}
```

9.6. Key Notification Client

Key Notification Client Callbacks 函数注册。

```
bool kns_client_start_discovery(uint8_t conn_id)
{
    PROFILE_PRINT_INFO0("kns_client_start_discovery");
    if (conn_id >= kns_link_num)
    {
        PROFILE_PRINT_ERROR1("kns_client_start_discovery: failed invalid
conn_id %d", conn_id);
        return false;
    }
    /* First clear handle cache. */
    memset(&kns_table[conn_id], 0, sizeof(T_KNS_LINK));
    kns_table[conn_id].disc_state = DISC_KNS_START;
    if (client_by_uuid128_srv_discovery(conn_id, kns_client,
                                        (uint8_t   *)GATT_UUID128_KNS_SERVICE)
```

```
== GAP_CAUSE_SUCCESS)
    {
        return true;
    }
    return false;
}
```

kns_client_set_v3_notify 控制 notification flag, notify 为 0 disable notification, 为 1 enable notification

```
bool kns_client_set_v3_notify(uint8_t conn_id, bool notify)
```

Server 通知按键时，调用 kns_client_notif_ind_result_cb 回调函数。

cb_data.cb_content.notif_ind_data.data.p_value = p_value 获取按键值

```
static T_APP_RESULT kns_client_notif_ind_result_cb(uint8_t conn_id, bool
notify,
                                                   uint16_t handle,
                                                   uint16_t value_size, uint8_t
*p_value)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    T_KNS_CLIENT_CB_DATA cb_data;
    uint16_t *hdl_cache;
    hdl_cache = kns_table[conn_id].hdl_cache;

    cb_data.cb_type = KNS_CLIENT_CB_TYPE_NOTIF_IND_RESULT;

    if (handle == hdl_cache[HDL_KNS_NOTIFY_KEY])
    {
        cb_data.cb_content.notif_ind_data.type = KNS_KEY_NOTIFY;
        cb_data.cb_content.notif_ind_data.data.value_size = value_size;
        cb_data.cb_content.notif_ind_data.data.p_value = p_value;
    }
    else
    {
        return app_result;
    }
    /* Inform application the notif/ind result. */
    if (kns_client_cb)
    {
        app_result = (*kns_client_cb)(kns_client, conn_id, &cb_data);
    }

    return app_result;
}
```

10. 服务的初始化和注册回调函数

防丢器一共有 6 个 service 和 7 个 client， 在 main 函数中通过调用 app_le_profile_init() 进行注册和初始化。服务的注册和初始化如下：

```
void app_le_profile_init(void)
{
    server_init(6);
    ias_srv_id  = ias_add_service(app_profile_callback);
    lls_srv_id  = lls_add_service(app_profile_callback);
    tps_srv_id  = tps_add_service(app_profile_callback);
    kns_srv_id  = kns_add_service(app_profile_callback);
    bas_srv_id  = bas_add_service(app_profile_callback);
    dis_srv_id  = dis_add_service(app_profile_callback);
    server_register_app_cb(app_profile_callback);

    client_init(7);
    gaps_client_id= gaps_add_client(app_client_callback, APP_MAX_LINKS);
    ias_client_id  = ias_add_client(app_client_callback, APP_MAX_LINKS);
    lls_client_id  = lls_add_client(app_client_callback, APP_MAX_LINKS);
    tps_client_id  = tps_add_client(app_client_callback, APP_MAX_LINKS);
    bas_client_id = bas_add_client(app_client_callback, APP_MAX_LINKS);
    dis_client_id = dis_add_client(app_client_callback, APP_MAX_LINKS);
    kns_client_id  = kns_add_client(app_client_callback, APP_MAX_LINKS);
    client register general client cb(app client callback);
}
```

11. DLPS

11.1. DLPS 概述

RTL8762D 支持 DLPS（Deep Lower Power State，即深度睡眠状态）模式[1]，当系统大多数时间处于空闲状态时，进入该模式可以大大减少功耗。该模式下 Power、Clock、CPU、Peripheral、RAM 都可以关闭/掉电以降低系统的功耗，进入 DLPS 模式之前需要保存必要的数据以恢复系统。当有事件需要处理时，系统就会退出 DLPS 模式，CPU、Peripheral、Clock、RAM 重新上电并且恢复到进入 DLPS 之前的状态，然后响应唤醒事件。

11.2. DLPS 使能和配置

打开 DLPS 的配置有以下步骤：

(1). 打开 board.h 中的相关宏：

#define DLPS_EN 1

对程序中所使用的相关模块，还需要打开各模块的宏。

#define USE_USER_DEFINE_DLPS_EXIT_CB 1

#define USE_USER_DEFINE_DLPS_ENTER_CB 1

#define USE_GPIO_DLPS 1

(2). 在 main.c 中的 pwr_mgr_init 函数中注册 DLPS 相关 CallBack function：

a). DLPS_IORegUserDlpsEnterCb: 在 callback function 中执行进入 DLPS 时所需的配置；

b). DLPS_IORegUserDlpsExitCb: 在 callback function 中执行退出 DLPS 是所需的配置；

c). dlps_check_cb_reg: 通过配置注册 DLPS_PxpCheck 后，通过 allowedPxpEnterDlps 的值设置防丢器是否可以进入 DLPS。

```
void PxpEnterDlpsSet(void)
{
    Pad_Config(KEY_S,       PAD_SW_MODE,       PAD_IS_PWRON,       PAD_PULL_UP,
PAD_OUT_DISABLE, PAD_OUT_LOW);
    Pad_Config(LED_S,       PAD_SW_MODE,       PAD_IS_PWRON,       PAD_PULL_DOWN,
PAD_OUT_DISABLE, PAD_OUT_LOW);
    Pad_Config(KEY_M,       PAD_SW_MODE,       PAD_IS_PWRON,       PAD_PULL_UP,
PAD_OUT_DISABLE, PAD_OUT_LOW);
    Pad Config(LED M,       PAD SW MODE,       PAD IS PWRON,       PAD PULL DOWN,
PAD OUT DISABLE, PAD OUT LOW);
    Pad Config(BEEP,       PAD SW MODE,       PAD IS PWRON,       PAD PULL DOWN,
PAD OUT DISABLE, PAD OUT LOW);
    System WakeUpDebounceTime(0x8);
    if (keySstatus)
    {
        System_WakeUpPinEnable(KEY_S,                       PAD_WAKEUP_POL_LOW,
```

```
PAD_WK_DEBOUNCE_ENABLE);
    }
    else
    {
        System_WakeUpPinEnable(KEY_S,                    PAD_WAKEUP_POL_HIGH,
PAD_WK_DEBOUNCE_ENABLE);
    }

    if (keyMstatus)
    {
        System_WakeUpPinEnable(KEY_M,                     PAD_WAKEUP_POL_LOW,
PAD_WK_DEBOUNCE_ENABLE);
    }
    else
    {
        System_WakeUpPinEnable(KEY_M,                    PAD_WAKEUP_POL_HIGH,
PAD_WK_DEBOUNCE_ENABLE);
    } }

void PxpExitDlpsInit(void)
{
    Pad_Config(LED_S,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,
PAD_OUT_ENABLE, PAD_OUT_LOW);
    Pad_Config(LED_M,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,
PAD_OUT_ENABLE, PAD_OUT_LOW);
    Pad_Config(BEEP,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,
PAD_OUT_ENABLE, PAD_OUT_LOW);
    Pad_Config(KEY_S,     PAD_PINMUX_MODE,     PAD_IS_PWRON,     PAD_PULL_UP,
PAD_OUT_DISABLE, PAD_OUT_LOW);
    Pad_Config(KEY_M,     PAD_PINMUX_MODE,     PAD_IS_PWRON,     PAD_PULL_UP,
PAD_OUT_DISABLE, PAD_OUT_LOW);
}

bool DLPS_PxpCheck(void)
{
    return allowedPxpEnterDlps;
}

void pwr_mgr_init(void)
{
#if DLPS_EN
    if (false == dlps_check_cb_reg(DLPS_PxpCheck))
    {
        DBG_DIRECT("Error: dlps_check_cb_reg(DLPS_RcuCheck) failed!\n");
    }
    DLPS IORegUserDlpsEnterCb(PxpEnterDlpsSet);
    DLPS IORegUserDlpsExitCb(PxpExitDlpsInit);
    DLPS IORegister();
    lps mode set(LPM DLPS MODE);
#endif
}
```

11.3. DLPS 的条件和唤醒源

防丢器应用中，广播状态和连接状态均可进入 DLPS，但需要符合相关广播参数及连接参数的设置。

(1). 广播状态：主要广播参数符合条件，并且开启 DLPS 功能，则系统可以直接进入 DLPS。在需要广播时，系统自动退出 DLPS，发送广播包，随后再次进入 DLPS。

(2). 连接状态：当防丢器与对测端建立连接之后，防丢器端会请求连接参数更新，所请求参数符合进入 DLPS 条件。当参数更新成功，并且开启 DLPS 时，便可以进入 DLPS。一般情况，为保证防丢器能够正确进入 DLPS， 需要请求更改连接参数，ChangeConnectionParameter(400, 0, 2000); //interval = 400*1.25ms

```
void ChangeConnectionParameter(uint16_t interval, uint16_t latency, uint16_t timeout)
{
    le_update_conn_param(0, interval, interval, latency, timeout / 10, interval * 2 - 2,
                         interval * 2 - 2);
}
```

可以通过蓝牙事件、RTC 以及 Wakeup Pin 将防丢器从 DLPS 中唤醒。

防丢器的按键需要选择都有 Wakeup 功能的 Pin，否则按键可能没有效果。另外用户在处理按键唤醒事件时，应在唤醒并进入 GPIO 中断后，通过 allowedPxpEnterDlps 暂时禁止系统进入 DLPS，待按键消息处理完成之后，在允许进入，保证按键事件不被影响。

12 参考文献

[1] RTL8762D Proximity Application Design Spec.pdf

[2] IEEE Std 11073-20601 ™- 2008 Heath Information – Personal Health Device Communication – Application Profile – Optimized Exchange Protocol – version 1.0 or later.

[3] Profile Interface Design.pdf