RTL8762D Proximity Application Design Spec

V1.1

2021/03/30

Revision History

Date	Version	Comments	Author	Reviewer
2021/03/30	V0.1	First draft, inherited from rtl8762c document, and added CPU clock setting feature	Ken_mei	
2021/03/30	V1.0			

Contents

Figure List

Table List

1. Overview

Proximity application is mainly designed to provide anti-lost and recovery function for mobile phones, wallets, keys, and luggage and other valuable things. If mobile terminals keep connected with proximity device, both of them can search for each other by sound and light alarm. If far away from each other, both will also alarm.

Figure 1.1 Proximity application usage scenario

1.1. Device list

1. Bee Evaluation Board — Mother board
2. RTL8762D_QFN40/QFN48/ QFN56 — Daughter board
3. Buzzer

1.2. System requirement

Tools that need to be downloaded and installed on PC:

1. Keil MDK-ARM 5.12
2. SEGGER's J-Link tools
3. RTL8762D SDK

4. RTL8762x Flash programming algorithm

Tools that need to be installed on mobile device:

1. RtkPxp

1.3. Terminology Definition

1. SUT：System under test, Android or IOS mobile devices with proximity application

2. Proximity device：Proximity application based on development of RTL8762D

3. DPLS：Deep Low Power State

2. Hardware design

2.1. Circuit design

Simulate circuit with RTL8762D EVB, More information please refer to RTL8762D EVB SCH. And keyboard is simulated with KEY2 in EVB.

LED is simulated with LED0 and BEEP is simulated with LED1 on EVB.

2.2. Pin definition

RTL8762DK Daughter board：
LED   P0_1
BEEP   P0_2
KEY   P2_4

3. Software Structure Overview

Proximity application mainly interacts with IO Driver and BT to accomplish specific function. The software structure is shown in Figure 3.1
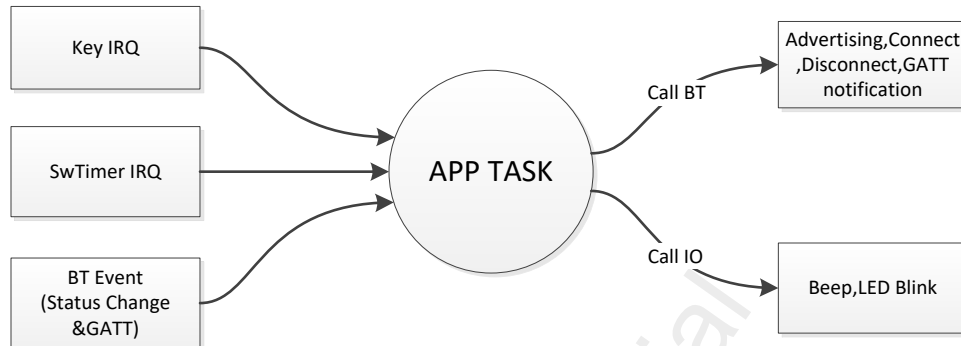
Figure 3.1 Proximity software structure

After system initialization, App task starts running to create message queue for APP layer, register APP layer callback function in Upper Stack and initialize interface for APP layer. Then App task will keep polling message queue to handle message. Upper stack, IO ISR and Timer ISR send message to message queue of APP layer, APP task then pops message from the queue to handle.

```
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);
            }
        }
    }
```

4. IO Initialization and Handling

There are 3 IOs (LED, KEY, BEEP) defined in Proximity application. Definition in file board.h is shown below:

```
#define LED      P0_1      // LED0 ON EVB
#define BEEP     P0_2      // LED1 ON EVB
#define KEY      P2_4
```

4.1. IO Initialization

Process of IO Initialization:
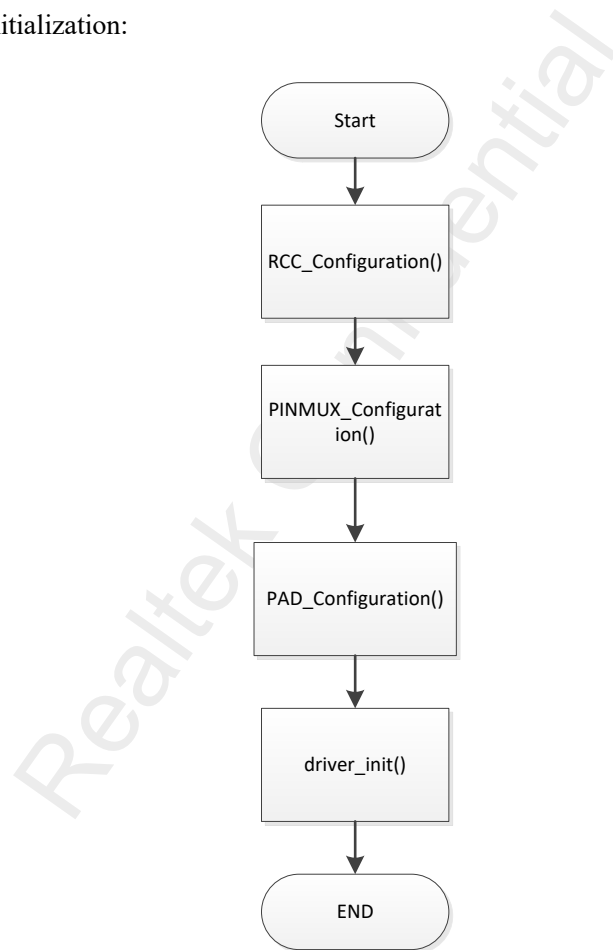
Figure 4.1 IO Initialization Process

In driver_init(), KEY interrupt is registered and enabled. To prevent OS initialization from being interrupted, which will cause logical exception, it is recommended to execute driver_init() in APP task.

```
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();
    ......
    ......
    ......
}
```

4.2. IO Handling

4.2.1.Key Press handling

Key press in Proximity application includes long press and short press, which are implemented below

Key Status definition:

```
typedef enum  KeyStatus
{
      keyIdle = 0,
      keyShortPress,
      keyLongPress,
} KeyStatus;
```

If there is no key pressed, key status is keyIdle, namely in releasing status. When key is pressed, set key status to keyPress to trigger interrupt polarity reversal and invoke os_timer_start(&xTimerLongPress). When timer xTimerLongPress expired, sw timer handler will set key status to keyLongPress and send long press message to APP task. After key is released, interrupt is triggered to reverse polarity (waiting for following key press) and then identify key status. If current status is keyShortPress, os_timer_start (&xTimerLongPress) will be invoked and short press message will be sent to APP task. Finally, restore key status to keyIdle.

After receiving long press message, APP task will switch to Bluetooth related state. After receiving short press message, APP task switch to IO related state.

Bluetooth state of Proximity application:

```
PxpStateIdle = 0,
PxpStateAdv  = 1,
PxpStateLink = 2,
```

IO state of Proximity application:

```
IoStateIdle     = 0,
IoStateAdvBlink = 1,
IoStateImmAlert = 2,
IoStateLlsAlert = 3,
```

The following is state transition mechanism showing status transition in key long press
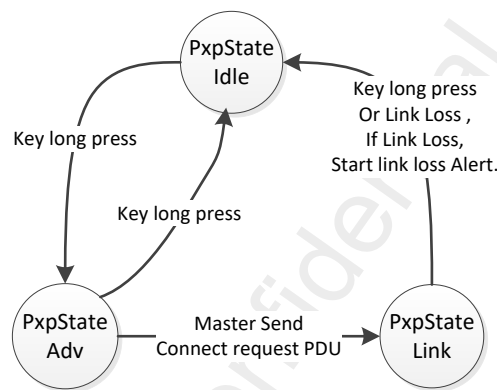
Figure 4.2 Bluetooth State Change Machine

Proximity short press process:

A) When proximity is in idle state, LED flickers once indicating Proximity device works properly

B) When IO is in Alert state, turn off LED and BEEP

C) When proximity is in link state and IO isn't in Alert status, send alert notification to master device

4.2.2. IO Process implementation

IO process is implemented by invoking function void StartPxpIO (uint32_t lowPeroid, uint32_t HighPeroid, uint8_t mode, uint32_t cnt), which controls LED and BEEP in Proximity application. There are 4 input parameters: lowPeriod, HighPeriod, mode and cnt.

- lowPeriod: The duration of low level when IO reverses polarity;

- HighPeriod: The duration of high level when IO reverses polarity;

- mode: indicates the use of LED or BEEP or both;
- cnt: times of IO polarity invertion.

StartPxpIO is a soft timer implemented by starting xTimerPxpIO. When soft timer interrupt is generated, ISR will set the delay time of next timer according to IO and determine whether to restart timer so as to keep the IO reversal state based on cnt value.

5. Advertising

With Proximity device advertising, mobile phones or other BT host devices can discover the device named REALTEK_PXP. Advertising data include GAP_ADTYPE_FLAGS, UUID (IAS Service), Local Name (REALTEK_PXP). When active scan is used, scan response data with Appearance field of KEYRING cans be scanned.

```
/** @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 GENERIC KEYRING),
    HI WORD(GAP GATT APPEARANCE GENERIC KEYRING),
};

/** @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,
    LO WORD(GATT UUID IMMEDIATE ALERT SERVICE),//0x02
    HI WORD(GATT UUID IMMEDIATE ALERT SERVICE),//0x18

    0x09,
    GAP ADTYPE LOCAL NAME COMPLETE,
    'R', 'E', 'A', 'L', ' ', 'P', 'X', 'P'
};
```

6. GATT Service and Characteristic

Services of Proximity application:

1. Immediate Alert Service: Operate Proximity device to make it alarm immediately;

2. Link Loss Service: Start alarming once connection is broken;

3. Tx Power Service: Indicate transmitting power;

4. Battery Service: Report battery level, remind to change battery and prohibit OTA when battery level is low;

5. Device Information Service: Display device information;

6. Key Notification Service: Set alarm times and send key alarm notification to Master device;

Services and related UUIDs are shown in Table 6.1

Table 6.1 Services and related UUIDs

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

6.1. Immediately Alert Service

Immediately Alert Service is standard GATT service, whose functions are defined in file ias.c and ias.h.

Table 6.2 Immediate Alert Service Characteristic list

Characteristic Name	Requirement	Characteristic UUID	Properties	Description

Alert Level	M	0x2A06	Write Without Response	See Alert Level

Alert Level defines alert level of device whose data format is 8-bit unsigned integer with initial value 0. There are 3 alarm levels, 0 represents no alert, 1 represents medium alert, 2 represent highly alert, shown in Table 6.4

Table 6.3 Alert Level Characteristic Value Format

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

Table 6.4 Alert Level Enumerations

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

Immediate Alert Service defines only one characteristic, Alert Level. Alert Level is a control point that enables SUT to trigger local device alarm by writing no response to Alert Level. Alarm intensity can be adjusted by modifying Alert level value.

When local device alarms, the following methods can stop alarming:

- Reach alarm times, default set is 10
- User turn off alarm
- New Alert Level is written
- Link is disconnected and new alarm will be started (previous alarm can be considered as stopped).

GATT Attribute Table of IAS is shown below:

```
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 */
        }
};
```

Table 6.5 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

When SUT send alert data to Proximity device through IAS service, APP task invokes IAS Write Callback function and handles alert data in function AppHandleGATTCallback.

```
else if (service id == ias srv id)
    {
        T IAS CALLBACK DATA *p ias cb data = (T IAS CALLBACK DATA
*)p data;
        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;
            if (g pxp immediate alert level == 1)
            {
                gIoState = IoStateImmAlert;
                StartPxpIO(ALERT LOW PERIOD, ALERT HIGH PERIOD,
LED BLINK, 10);
            }
            if (g pxp immediate alert level == 2)
            {
                gIoState = IoStateImmAlert;
                StartPxpIO(ALERT LOW PERIOD, ALERT HIGH PERIOD,
LED BLINK | BEEP ALERT, 10);
            }
            else
            {
                StopPxpIO();
            }
        }
    }
```

In Proximity application, user can send 3 different Immediate Alert levels, which are attribute values to be written to IAS Service:

- 0: No alarm

- 1: LED flickering alarm

- 2: LED flickering and Buzzer beeping alarm

6.2. Link Loss Alert Service

Link Loss Alert Service is standard GATT Service. Its functions are defined in files lls.c and lls.h.

<p align="center">Table 6.6 Link Loss Service Characteristic list</p>

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

Alert Level defines alert level of device whose data format is 8-bit unsigned integer with initial value 0. There are 3 levels, 0 represents no alert, 1 represents medium alert, 2 represent highly alert, shown in Table 6.8

<p align="center">Table 6.7 Alert Level Characteristic Value Format</p>

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

<p align="center">Table 6.8 Alert Level Enumerations</p>

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

Link Loss Alert Service defines only one characteristic, Alert Level. Alert Level is a readable and writable characteristic that enables SUT to configure local device alert level by writing response to Alert Level. Device alarms at configured level when link loss occurs.

GATT Attribute Table of LLS is shown below:

```
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 */
}
};
```

When SUT send alert level data to Proximity device through LLS service, APP task invokes LLS Write Callback function and handles the data in function AppHandleGATTCallback.

When SUT read alert level data from Proximity device through LLS service, APP task invokes LLS Read Callback function and handles the data in function AppHandleGATTCallback.

```
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;
        }
    }
```

When link loss occurs, function related to state transition will determine the cause of link loss, start alarming based on alert level and restart advertising.

```
case GAP_CONN_STATE_DISCONNECTED:

    {

        if ((disc_cause != (HCI_ERR | HCI_ERR_REMOTE_USER_TERMINATE))

            && (disc_cause != (HCI_ERR | HCI_ERR_LOCAL_HOST_TERMINATE)))

        {

            APP_PRINT_ERROR1("app_handle_conn_state_evt: connection lost

cause 0x%x", disc_cause);

            if (gPowerFlg == true)

            {

                le_adv_start();


                if (g_pxp_linkloss_alert_level == 1)

                {

                    gIoState = IoStateLlsAlert;

                    StartPxpIO(ALERT_LOW_PERIOD, ALERT_HIGH_PERIOD,

LED_BLINK, gTimeParaValue);

                }

                if (g_pxp_linkloss_alert_level == 2)

                {

                    gIoState = IoStateLlsAlert;

                    StartPxpIO(ALERT_LOW_PERIOD, ALERT_HIGH_PERIOD,

LED_BLINK | BEEP_ALERT, gTimeParaValue);

                }

                else

                {
```

```
                //nothing to do
              }
          }
      }
      gPxpState = PxpStateIdle;
   }
   break;
```

6.3. Tx Power Service

Tx Power Service is a standard GATT service whose functions are defined in files tps.c and tps.h. It contains only one read-only characteristic, Tx Power Service, which is shown in Table 6.9

Table 6.9 Tx Power Service Characteristic List

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

Tx Power Level indicates current transmitting power with unit dBm, which ranges from -100dBm to 200dBm with resolution of 1dBm. The data format is signed 8-bit integer with initial value 0 as shown in Table 6.10.

Table 6.10 Tx Power Level Characteristic Value Format

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

The default value of Tx Power Level is 0dBm.

GATT Attribute Table of TPS is shown below:

```
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  */
    },

    /* Alert Level Characteristic */
    {
        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 */
    },

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

When SUT reads alert level data from Proximity device through TPS service, APP task invokes TPS Read Callback function and handles the data in function AppHandleGATTCallback.

```
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);
```

```
            }
        }
    }
```

6.4. Battery Service

Battery Service is a standard GATT service whose functions are defined in files bas.c and bas.h. It contains only one characteristic, Battery Level, as shown in Table 6.11

<center>Table 6.11 Battery Service Characteristic List</center>

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

Battery Level indicates current battery level ranging from 0 to 100 percent. Its data format is unsigned 8-bit integer, as shown in Table 6.12.

<center>Table 6.12 Battery Level Characteristic Value Format</center>

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

GATT Attribute Table of BAS is shown below:

```
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
};
```

Notify attribute of BAS is optional, and can be compiled and enabled through macro definition in attribute table.

When SUT read battery level data from Proximity device through BAS service, APP task invokes BAS Read Callback function and handles the data in function AppHandleGATTCallback.

If notify attribute is registered in attribute table and SUT writes battery level enable notification to Proximity application through BAS service, APP task will invoke BAS write enable Callback function and handle the notification in function AppHandleGATTCallback. When write is enabled, APP can start a timer to report battery level of battery regularly.

```
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;
    }
}
```

6.5. Device Information Service

Device Information Service is a standard GATT service whose functions are defined in files dis.c and dis.h. There are multiple optional read-only characteristic and user can select necessary field by macro definition.

<p align="center">Table 6.13 Device Information Service Characteristic List</p>

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 contains characteristic of displaying basic information, e.g. device name and firmware version, as shown in Table 6.14

<p align="center">Table 6.14 Device Information Characteristic Value Format</p>

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 is made up of 2 fields, including 40-bit vendor-defined ID and 24-bit OUI (Organized Unique Identity), as shown in Table 6.15

Table 6.15 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 lists different certifications which device needs to manage or comply with, as shown in Table 6.16

Table 6.16 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 is a group of ID value used to create unique device ID. It includes Vendor ID Source, Vendor ID, Product ID and Product Version. These values are used to identify devices with specific type/mode/version, as shown in Table 4.14

Table 6.17 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

Table 6.18 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

GATT Attribute Table of DIS is shown below:

```
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
};
```

When SUT reads device information from Proximity device through DIS service, APP task
invokes DIS Read Callback function and handles the information in function
AppHandleGATTCallback.

```
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;
    }
}
```

6.6. Key Notification Service

Key Notification Service is a customized GATT Service private for Proximity application. Its functions are defined in files kns.c and kns.h.

There are 2 characteristics defined in Key Notification Service: Set Alert Time can be read and written to configure alarm times after link loss occurs; Key Value is used to send alert notification to master device.

Table 6.19 Key Notification Service Characteristic List

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 is a characteristic to set alert time. Its data format is 32-bit unsigned integer with initial value 30 in second, and value range from 0 to 0xFFFFFFFF, shown in Table 6.20

Table 6.20 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 indicates key information. Its data format is 8-bit unsigned integer. When connected, it sends value 1 to master device, shown in Table 6.21

Table 6.21 Key Value Characteristic Value Format

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

GATT Attribute Table of KNS is shown below:

```
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 */
    }
};
```

When SUT sets or reads/writes parameter (link loss alarm times) to Proximity device through

KNS service, APP task invokes KNS Read Callback function and handles the read/write operation

in function AppHandleGATTCallback.

When SUT writes to enable notification of key alarm to Proximity device through KNS service,

APP task invokes KNS Write Callback function and handles in function

AppHandleGATTCallback. When write enabled and link connected, user short press key to make

Proximity device to send alarm notification to master device.

```
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_INFO0("KNS_READ_PARA");
                    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;
        }
    }
```

7. Initialize and register callback function for service

There are 6 services in total for Proximity application, and main() function invokes app_le_profile_init() to initialize and register them. The process is shown as below:

```c
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);
}
```

8. DLPS

8.1. DLPS Overview

RTL8762D supports DLPS (Deep Lower Power State) mode. When the system is idle for most of the time, entering DLPS mode can greatly reduce power consumption. In this mode, Power, Clock, CPU, Peripheral and RAM can be turned off to reduce power consumption, but vital data need to be saved before entering DLPS mode. With events to be handled, system will quit DLPS mode and CPU, Peripheral, Clock and RAM will be turned on to revert to the state before entering DLPS and then respond to wakeup events.

8.2. Enable and configure DLPS

The following steps are required for configuring DLPS:

    (1). Define macro to enable DLPS in board.h:

```
     #define DLPS_EN    1
```

Define other related macros:

```
    #define USE_USER_DEFINE_DLPS_EXIT_CB      1
    #define USE_USER_DEFINE_DLPS_ENTER_CB     1
    #define USE_GPIO_DLPS                      1
```

    (2). Register DLPS Callback functions in pwr_mgr_init function of main.c:

a). DLPS_IORegUserDlpsEnterCb: carry out configuration required to enter DLPS

b). DLPS_IORegUserDlpsExitCb: carry out configuration required to exit DLPS

c). dlps_check_cb_reg: register DLPS_PxpCheck by carrying out the configuration and
    determine whether Proximity application can enter DLPS based on value of
    allowedPxpEnterDlps

```
void PxpEnterDlpsSet(void)
{
    Pad Config(KEY, PAD SW MODE, PAD IS PWRON, PAD PULL UP,
PAD OUT DISABLE, PAD OUT LOW);
    Pad Config(LED, PAD SW MODE, PAD IS PWRON, PAD PULL NONE,
PAD OUT ENABLE, PAD OUT LOW);
    Pad Config(BEEP, PAD SW MODE, PAD IS PWRON, PAD PULL NONE,
PAD_OUT_ENABLE, PAD_OUT_LOW);
```

```
    System_WakeUpPinEnable(KEY, 1, 0);
}

void PxpExitDlpsInit(void)
{
    Pad_Config(LED, 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, 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
}
```

8.3. DLPS Conditions and Wakeup

In Proximity application, both advertising state and connected state can enter DLPS, but configuration need conform to related parameter requirements.

(1). advertising state: When main advertising parameters meet conditions and DLPS function is enabled, system will enter DLPS mode. When advertising is required, system will exit DLPS automatically to send advertising packet, then return to DLPS mode again.

(2). Connected state: When Proximity device connects with SUT, Proximity device will request to update connection parameters which meet DLPS conditions. If parameters are updated successfully and DLPS function is enabled, system will enter DLPS mode. Generally, to ensure that Proximity device can correctly enter DLPS mode, invoke ChangeConnectionParameter(400, 0, 2000) to modify parameters; //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);

    }
```

Bluetooth events, RTC and Wakeup Pin can wake up Proximity application from DLPS.

Key of Proximity device should select Pin with Wakeup function; otherwise key press may have no effect. When system is handling key press wakeup event and has entered GPOI interrupt, user should temporarily prohibit device to enter DLPS mode by invoking allowedPxpEnterDlps. Only when key press event has finished handling can DLPS be entered so as to prevent key press event handler from being interrupted.

9. CPU CLOCK

The frequency of RTL8762D series can run up to 90MHz. Customers can change the CPU clock rate based on the enumeration below, weighing the power consumption of the device and CPU efficiency.

```
{
    SYSTEM_20MHZ = 20000000,
    SYSTEM_40MHZ = 40000000,
    SYSTEM_80MHZ = 80000000,
    SYSTEM_90MHZ = 90000000,
    SYSTEM_100MHZ = 100000000     // Not recommended
} T_SYSTEM_CLOCK;
```

CPU clock rate is usually set in main function：

set_system_clock(SYSTEM_90MHZ);

Users can switch CPU clock rate at any time when program is running and hardware modules are idle. It is required to finish existing hardware operations before switching CPU clock rate, such as UART, timers, etc.

10. Reference

[1] RTL8762C PXP Design Spec.pdf

[2] IEEE Std 11073-20601 ™- 2008 Hea;th Information – Personal Health Device

    Communication – Application Profile – Optimized Exchange Protocol – version 1.0 or later.

[3] Profile Interface Design.pdf