ST17H26 ble_0109_sdk 开发说明 V1.02

Ui.c 用户文件模板

由于在<Lenze 17H26 BLE SDK User Guide_v1.01>中已经明确地说明了一个标准 SDK 的软件架构和基本文件，本节中主要介绍用户怎样在现有的 SDK 之上，合理安排程序，进行自己的开发。

```
#include "Ui.h"  //包含用户自定义的宏定义以及数据类型描述的头文件
#include "../../proj_lib/ble_l2cap/ble_ll_ota.h"   //包含 MCU 运行所需的 BLE 协议栈
#include "../../proj/drivers/flash.h" //包含用户 MCU 相关的 flash 外设驱动程序

static inline void send_databuf_tmp();

////////////////////////////////cfg address  /////////////

#if (1)  //设备连接后或广播状态下开启低功耗模式，此时观测不到变量的实时变化
#define     SUSPEND_STATE           SUSPEND_CONN | SUSPEND_ADV
#else  // 设备是全速运行而不是省电模式，此时可以通过 TDebug 工具来观测到变量的实时变化
#define     SUSPEND_STATE           0
#endif

#if(OTA_ENABLE)  //开启空中更新固件
#define TEST_OTA_1 0
#endif

#define  TEST_SUSPEND_TIME_ENABLE 0

#define ADV_LED_PORT          GPIO_GP7

/****设备广播包信息和广播响应包信息和设备初始的 mac 地址，详情见<Lenze 17H26 BLE SDK
User Guide_v1.01>****/
u8  tbl_mac [] = {0x20, 0x17, 0x06, 0x29, 0x01, 0x09};
u8  tbl_adv [42] =
    {   0x00, 37,
        0, 0, 0, 0, 0, 0,      //mac address
        0x02, 0x01, 0x06,                          // BLE limited discoverable
mode and BR/EDR not supported
        0x03, 0x02, 0x12, 0x18,        // incomplete list of service class UUIDs
(0x1812, 0x180F)
        0x03,0x19,GAP_APPEARE_ROLE&0x0ff,(GAP_APPEARE_ROLE>>8)&0xff,

        19, 0x09,
        'D','E','M','O','_','H','I','D','_','0','1','0','9',' ',' ',' ',' ',' ',//
must be is 18 byte
        0, 0, 0         //reserve 3 bytes for CRC
    };
```

```
u8  tbl_rsp [] =
        {0x04, 6,                                   //type len
          0, 0, 0, 0, 0, 0,     //mac address
         0, 0, 0                                    //reserve 3 bytes for CRC
        };
/////////////////////////变量初始化区/////////////////////////////////////////

extern u16 blt_conn_inst;
extern u8 os_check;
/****************************************** user define***************/


u16 button_value_tmp=0;
u16 button_last_value_tmp=0;

u16 normal_key_tmp=0;
u16 last_normal_key_tmp=0;

u16 consumer_key_tmp=0;
u16 last_consumer_key_tmp=0;

u16 joystic_key_tmp=0;
u16 last_joystic_key_tmp=0;
u16 joystic_buf_tmp=0;

u8 last_mouse_key_tmp=0;
u8 mouse_buf_tmp[4]={0,0,0,0};

u8 flag_has_new_event_tmp=0;

u8 device_status_tmp=0;

u8 flag_start_adv_tmp=0;

u32 tick_adv_timer_tmp=0;
u32 tick_led_timer_tmp=0;
u32 tick_app_wakeup = 0;
u32 tick_connected_timer_tmp=0;

u8  connected_idle_time_count_tmp=0;
u8  adv_start_count_tmp=0;
//u8 flag_power_key_press_tmp=0;
//u8 flag_change_mode_key_press_tmp=0;
u32 tick_power_key_press_tmp=0;

u8 cur_led_state_tmp = 0;
```

```
static u8 battLevel_tmp = 100;
u32 tick_batt_timer_tmp=0;
u8 need_low_battery_alarm_tmp=0;

u32 tick_hardware_scan_tmp=0;


#define  CFG_DEVEICE_NAME_LEN 32-1
//////////////////////////变量初始化区域尾 ////////////////////////////

////////////配合 OTP 生成的动态设备名，并及时更新到当前广播包中//////////////////
static inline void  device_name_extern_config_init()
{
  #if 1
    int i = 0;
    u8  dev_name_len = 0;
    u32 dev_name_uburing_addr = CFG_DEVEICE_NAME_ADDR;//指向当前的设备名地址存放区
域  u8 *reslut_device;
    // get device name data.
    for(i=0; i<32; i++){
//查询当前的设备名地址存放区域内部不为空值时，获取设备名字及其长度
        if((*(u8*)(CFG_DEVEICE_NAME_ADDR+CFG_DEVEICE_NAME_LEN-i) != 0)  &&
(*(u8*)(CFG_DEVEICE_NAME_ADDR+CFG_DEVEICE_NAME_LEN-i) != 0xff) ){
            if(dev_name_len == 0){
                dev_name_uburing_addr =
CFG_DEVEICE_NAME_ADDR+CFG_DEVEICE_NAME_LEN-i;
            }
            dev_name_len ++;
        }else{
            if(dev_name_len){
                break;
            }
        }
    }
//取得当前设备名存放地址
    reslut_device =(u8*)(dev_name_uburing_addr-dev_name_len+1);

    if(dev_name_len != 0)// no device name in device name under buring
    {
        memset((u8*)(tbl_adv+21),0x20,18);
//将设备名及时更新到当前广播包对应的偏移位置中
        memcpy((u8*)(tbl_adv+21) , reslut_device, dev_name_len);
            }

}
```

```
// 设备普通属性 GATT 的初始化
static inline void gatt_init_tmp()
{
//初始化广播包，响应包，以及设备地址
    blt_init (tbl_mac, tbl_adv, tbl_rsp); //get mac addr
//初始化设备支持的服务
    shutter_att_init ();
//为底层代码声明的省电标志符 blt_suspend_mask 赋予一个值（0：关闭省电；1：开启省电）
    blt_suspend_mask = SUSPEND_STATE;

}

static inline void rf_set_power_level_tmp()
{
    /*
以 4 个 16 进制数据为一组，每组中的 4 个数据依次对应赋值到负责控制 RF 强度的寄存器
0xa2,0x04,0xa7 和 0x8d 中。不同组合的数字所对应的功率等级在其后注释中，为 7dBm~-37dBm
不等。
    0x25, 0x7c, 0x67, 0x67,         // 7 dBm
    0x0a, 0x7c, 0x67, 0x67,         // 5 dBm
    0x06, 0x74, 0x43, 0x61,         // -0.6
    0x06, 0x64, 0xc2, 0x61,         // -4.3
    0x06, 0x64, 0xc1, 0x61,         // -9.5
    0x05, 0x7c, 0x67, 0x67,         // -13.6
    0x03, 0x7c, 0x67, 0x67,         // -18.8
    0x02, 0x7c, 0x67, 0x67,         // -23.3
    0x01, 0x7c, 0x67, 0x67,         // -27.5
    0x00, 0x7c, 0x67, 0x67,         // -30
    0x00, 0x64, 0x43, 0x61,         // -37
    0x00, 0x64, 0xcb, 0x61,         // -max power down PA & PAD
*/  //采用第三组数据，对应 rf 功率为-0.6dBm
    analog_write (0xa2, 0x06);
    analog_write (0x04, 0x74);
    analog_write (0xa7, 0x43);
    analog_write (0x8d, 0x61);
}
//无线更新的实现方法
void OTA_init_tmp()
{
#if(OTA_ENABLE)
    u8 buf[4] = {0};
    flash_read_page(OTA_FLASH_ADDR_START, 4, buf); //检测无线下载区的地址内是否有值
    u32 tmp = buf[0] | (buf[1]<<8) |(buf[2]<<16) | (buf[3]<<24);
    if(tmp != ONES_32){
        flash_erase_block(OTA_FLASH_ADDR_START);
        sleep_us(1*1000*1000); // because flash_erase_block may exit premature,
```

```
because of flash_wait_done
    }
#endif
}
//为要使用的 GPIO 口进行初始化设定
void test_init_tmp()
{
    gpio_write(GPIO_GP4,0);  //设定 GPIO4 输出低电平
    gpio_set_output_en(GPIO_GP4,1);//设定 GPIO4 是输出引脚
    gpio_set_output_en(GPIO_GP5,1);//设定 GPIO5 是输出引脚
    gpio_set_output_en(GPIO_GP7,1);//设定 GPIO7 是输出引脚
    gpio_set_input_en(GPIO_GP10,1);//设定 GPIO10 是输入引脚
    gpio_set_output_en(GPIO_GP10,0);//设定 GPIO4 不是输出引脚

}
//调用上面定义好的函数，完成设备初始化功能
void user_init()
{
#if(DEBUG_FROM_FLASH)
    OTA_init_tmp();
    set_tp_flash();
    set_freq_offset_flash();
    set_mac_flash(tbl_mac);
#else
    set_tp_OTP();
    set_freq_offset_OTP();
    set_mac_OTP(tbl_mac);
#endif
    device_name_extern_config_init();
    rf_set_power_level_tmp();
    gatt_init_tmp();
//
    tick_led_timer_tmp=clock_time();
    tick_connected_timer_tmp=tick_led_timer_tmp;
    tick_hardware_scan_tmp=clock_time();
    test_init_tmp();
}
////////////////////// enable attribute table write/read call back functions ///////
/*Use l2cap_att_read_write_cb_flag to decide whether sdk should call
att_read_cb/att_write_cb
//使用 L2cap att_read/write_cb ,当设备收到无线读/写进程之后，即可根据发送的请求,响应
反馈
int att_read_cb(void*p ){
    rf_packet_att_read_t *req= (rf_packet_att_read_t*)p;
     return ATT_NO_HANDLED;
}
```

```
    ////////////////////// enable different event call back functions /////////
     /*Decide whether event cb should be called by SDK,should be defined in
user_init*/
/*Master write my_Attributes,  sdk firstly call att_write_cb. So user shall decide
here
 * to allow or disallow the write operation and return different value
 * User should not modify the name of function and shall not delete it even if user
don't
 * need this function
 *
 * Return: ATT_NO_HANDLED means user has not processed the write operation,
 *             SDK should also write the value automatically
 *            ATT_HANDLED means user has processed the write operation,
 *            SDK should not re-write the value*/
//att_write_cb()是用户不能删除的函数，用于响应设备发送过来的写请求，会有 2 种返回值类型，
分别代表用户处理//了写操作（SDK 应该自动地向对应 handle 写入值），或用户未处理写操作（SDK
不用向对应 handle 写入值）
int att_write_cb(void*p)
{
    rf_packet_att_write_t *src = (rf_packet_att_write_t*)p;
    u8 *value;
    value=&src->value;


    #if(OTA_ENABLE)
     if(src->handle == OTA_CMD_OUT_DP_H){
        u8 result=otaWrite(src);

        return ATT_HANDLED;
    }
#endif
     return ATT_NO_HANDLED;
}
/*
 other opcode
 * */
void att_response_cb( u8 *p){
    rf_packet_l2cap_req_t * req = (rf_packet_l2cap_req_t *)p;
    return;
}


 /*Following functions are used to define different events, details please refers
to blt_ll.h, user
  * can use flag ll_event_cb_flag to decide whether call this event*/
```

```
#if (LOW_COST_EVENT_CB_MODE)

  /*Decide whether event cb should be called by SDK,should be defined in user_init*/
  //ll_event_cb_flag = BLT_EV_FLAG_CONNECT  | BLT_EV_FLAG_TERMINATE |
BLT_EV_FLAG_BOND_START;

/*task_connection_established
 * This event is returned once module receives a connection request packet
 * and establishs a connection successfully
 * ex: start send connection parameter update request after a time from
 * the connection event; notify application connection establishment state*/
 //task_connection_established 函数是 SDK 在设备间成功建立连接后自动调用的，用户可在其
中加入自定义程序
void task_connection_established(rf_packet_connect_t* p){
    //adv_start_tick = last_update_paramter_time = clock_time();// in bond state
better
    //adv_time_cnt = 0;
    tick_connected_timer_tmp=clock_time();
    tick_batt_timer_tmp=tick_connected_timer_tmp;
    connected_idle_time_count_tmp=0;
    battLevel_tmp=100;
    gpio_write(ADV_LED_PORT,1);
    device_status_tmp=CONNECTED_DEVICE_STATUS;
}

/*task_connection_terminated
 * This event is returned once connection is terminated
 * ex:notify application connection terminated; reset connection para*/
 //task_connection_terminated 函数是 SDK 在设备间断开连接后自动调用的，用户可在其中加
入自定义程序
void task_connection_terminated(rf_packet_connect_t* p){
    device_status_tmp=POWER_ON_DEVICE_STATUS;
    gpio_write(GPIO_GP4,1);
    blt_smp_paring_req_recvd = 1;
    hid_setting_flag(0);
}

/*task_bond_finished
 * This event is returned once encryption process is finished
 * ex: HID key can be used once connection is encrypted*/
 //task_bond_finished 函数是 SDK 在设备间连接使用 HID 以及 SMP 协议后自动调用的，用户可
在其中加入自定义程序
void task_bond_finished(rf_packet_connect_t* p){
    if(!blt_smp_paring_req_recvd){// reconnection
        if(os_check == 2){
            hid_setting_flag(1);
```

```
        }
    }
}
#endif

u32 button_value = 0;
u32 button_value_bcup = 0;          //back up of  button
//样例程序，获取按键状态信息
static inline u8 button_get_status(u32 pin)
{
    u8 value=0;//no button press
//设置当前 GPIO_pin 为无上下拉电阻的浮动状态
    gpio_setup_up_down_resistor(pin,PM_PIN_UP_DOWN_FLOAT);
//设置当前 GPIO_pin 为高电平输出
    gpio_write(pin,1);
    sleep_us(20);
//读取当前 GPIO_pin 的引脚状态，若为低电平，设置按键按下后的值为 1.
    if(!gpio_read(pin))
    {
        value=1;
    }
//设置当前 GPIO_pin 为 下拉 100K 欧姆电阻的浮动状态
    gpio_setup_up_down_resistor(pin,PM_PIN_PULLDOWN_100K);
    gpio_write(pin,0);
    sleep_us(20);
//读取当前 GPIO_pin 的引脚状态，若为高电平，设置按键按下后的值为 2.
    if(gpio_read(pin))
    {
        value=2;
    }
    return value; //返回按键状态信息
}

u16  senddata =0;
u8 sendflag =0;
static inline u16  button_get_value()
{
    u16 status,value;
    value=0;
    status=button_get_status(GPIO_GP10);//GPIO_GP10
    value|=(status==1)?0x01:((status==2)?0x02:0x00);
    return value;
}
//样例程序，根据获取的按键状态信息，发送不同的 HID 键值
static inline void hw_scan(){
    button_value = button_get_value();
```

```
    if(button_value == button_value_bcup){
        return;
    }
    button_value_bcup = button_value;
    if(button_value == 0){
        flag_has_new_event_tmp|=SEND_C_DATA;
        reportConsumerControlIn[0]=0x0;
    }
    else{
        if(button_value&0x01){
            flag_has_new_event_tmp|=SEND_C_DATA;
            reportConsumerControlIn[0]=0xea;
        }
        else if(button_value&0x02){
            flag_has_new_event_tmp|=SEND_C_DATA;
            reportConsumerControlIn[0]=0xe9;
        }
    }


}
//样例程序，获取按键状态信息，并决定不同的输出信号
static inline void user_ui_process()
{
#if(TEST_SUSPEND_TIME_ENABLE==0)
//  test_key_tmp();
    hw_scan();

#endif
}
//样例程序，此函数调用了用来处理 ble slave 与 ble master 每一次通信时收发包的程序，还包含了功率管理的功能。一般情况下不用改变
static inline void public_loop()
{
    blt_brx_sleep (tick_app_wakeup);//功率管理
   if(blt_state!=BLT_LINK_STATE_ADV)//连接状态
    {
//处理 ble slave 与 ble master 每一次通信时更新参数和收发包的程序
        blt_brx ();
        if(blt_conn_inst > 20 && os_check < 2)
        {
            os_check = 2;
            vr_autoSetMode();
            hid_setting_flag(1);    //  android set ccc at default
        }
   }
```

```
    else     //广播状态
    {
        // Must be on the final
        blt_send_adv(BLT_ENABLE_ADV_ALL);
    }
}
//主循环中调用的函数，调用了上面所述的外设信息处理程序，以及广播和更新参数部分的程序
void main_loop()
{
    if(clock_time_exceed(tick_hardware_scan_tmp,20*1000))
    {
        tick_hardware_scan_tmp=clock_time();
        extern u8 start_ota_flag;
        if(start_ota_flag==0)
        {
            user_ui_process();
        }


    {
    if (blt_state == BLT_LINK_STATE_ADV)
    {
        blt_wakeup_src = 0;
        flag_has_new_event_tmp=0;
        blt_adv_interval= 20 * CLOCK_SYS_CLOCK_1MS;
        #if(TEST_OTA_1)
            if(clock_time_exceed(tick_led_timer_tmp,1000*1000))
        #else
            if(clock_time_exceed(tick_led_timer_tmp,200*1000))
        #endif
        {
            tick_led_timer_tmp=clock_time();
            cur_led_state_tmp = !cur_led_state_tmp;
            gpio_write(ADV_LED_PORT,cur_led_state_tmp);
        }
    }
    else
    {
        send_databuf_tmp();

        if(connected_idle_time_count_tmp==0)
        {
//          blt_retry=1;
        }
    }
```

```
}
/*****************************public area******************/
  public_loop();
}
//样例程序，根据设备的当前状态，发送连接更新参数或者 HID 键值到对应的服务 UUID handle 中
static inline void send_databuf_tmp()
{

#if 1
    /******************test with dongle  used for production*************/

    if(get_hid_ccc_flag()==0)
    {
        return;
    }
    else
    {
    //设备当前处于连接状态，那么发送连接更新参数
        if((device_status_tmp==CONNECTED_DEVICE_STATUS)&&( blt_fifo_num()<3))
        {
            device_status_tmp=AFTER_CONNECTED_DEVICE_STATUS;
            blt_update_connPara_request(8,16,4,600);
        }
    }
#endif

#if(KEYBOARD_REPORT_SUPPORT)
    //若设备当前开启了键盘报告支持，且有新数据发送的需要那么发送连接更新参数
    if((flag_has_new_event_tmp&SEND_K_DATA)&&( blt_fifo_num()<3))
    {
        flag_has_new_event_tmp&=~SEND_K_DATA;


        blt_push_notify_data(HID_NORMAL_KB_REPORT_INPUT_DP_H,reportKeyIn,8);
    }
#endif

#if(CONSUME_REPORT_SUPPORT)
    //若设备当前开启了消费型报告支持，且有满足新数据发送的条件那么发送连接更新参数

    if((flag_has_new_event_tmp&SEND_C_DATA)&&( blt_fifo_num()<3))
    {
        flag_has_new_event_tmp&=~SEND_C_DATA;
```

```
    blt_push_notify_data(HID_CONSUME_KB_REPORT_INPUT_DP_H,reportConsumerControl
In,2);
    }

#endif

#if(MOUSE_REPORT_SUPPORT)
    //若设备当前开启了鼠标报告支持，且有满足新数据发送的条件那么发送连接更新参数
    if((flag_has_new_event_tmp&SEND_M_DATA)&&( blt_fifo_num()<3))
    {
        flag_has_new_event_tmp&=~SEND_M_DATA;

        blt_push_notify_data(HID_MOUSE_REPORT_INPUT_DP_H,reportMouseIn,4);
    }
#endif

#if(JOYSTIC_REPORT_SUPPORT)
    //若设备当前开启了游戏手柄类报告支持，且有满足新数据发送的条件那么发送连接更新参数
    if((flag_has_new_event_tmp&SEND_J_DATA)&&( blt_fifo_num()<3))
    {

        flag_has_new_event_tmp&=~SEND_J_DATA;

        blt_push_notify_data(HID_JOYSTIC_REPORT_INPUT_DP_H,reportJoyStickIn,9);
    }
#endif


    #endif
}
```