17H26 的 GPIO 使用说明

本文是基于“v0110”版本的 SDK -H26SDKCommon_GATT_GPIO_TEST 的使用介绍，读者在看本文档时，可以直接打开对应名称的 SDK，加以实验。下面将从程序的起步和执行顺序讲起。

程序执行顺序 如下 main 中所示，在进入 while（1）之前，程序先后顺序运行 main 函数中的子函数，完成 17H26 的初始化。

```
int main (void) {
    cpu_wakeup_init();

    //clock_init
    write_reg8(0x66, 0x26); // 32M pll

    gpio_init();

   // reg_irq_mask = FLD_IRQ_ZB_RT_EN;
//    rf_drv_init(CRYSTAL_TYPE);
#if(DEBUG_FROM_FLASH)
    rf_drv_1M_init_flash();
#else
    rf_drv_1M_init_OTP();
#endif


#if(MODULE_ADC_ENABLE)
        adc_init();
#endif
    user_init ();

    watch_dog_en();
    while (1) {
        main_loop ();
        clr_watch_dog();
    }
}
```

主循环 main_loop();的作用是使得用户能够向其中添加特殊需求，本例中只讲如何添加按键点灯，以及从端向主端的数据传输部分（即长按，短按，连按按键的命令发送 ）功能，以及主端向从端的数据传输部分（ 接收来自 alert UUID 的数据指令）。

```
static inline void public_loop()
{
    tick_app_wakeup = buzzer_led_ui() ;
    blt_brx_sleep (tick_app_wakeup);
    if(blt_state!=BLT_LINK_STATE_ADV){
        blt_brx ();
    }
    else {
        // Must be on the final
        blt_send_adv (BLT_ENABLE_ADV_ALL);
        //blt_send_adv (BLT_ENABLE_ADV_38);
    }
}

void main_loop()
{
//  extern u8 start_ota_flag;
//  if(start_ota_flag==0)
//  {
//      user_ui_process();
//  }
    user_ui_process();
    if (blt_state == BLT_LINK_STATE_ADV)
    {
        blt_adv_interval = ((rand()% 10) + LOW_ADV_INTERVAL-1)*CLOCK_SYS_CLOCK_1MS;
    }
    else {
        if((device_status_tmp==CONNECTED_DEVICE_STATUS)&& clock_time_exceed(tick_connected_timer_tmp,1*1000*1000))
        {
            device_status_tmp=AFTER_CONNECTED_DEVICE_STATUS;
            blt_update_connPara_request(160,180,4,400);
        }
    }
/*****************************public area*****************/
    public_loop();
}
```

IO 口初始化和使用：

在 user_init()；中调用 下面的配置命令之前要先确定这一点：**PWM 相关的所有配置都用 write_reg16，而且对于部分 GPIO 而言，由于缺少一个优先级的设置，开启 pwm 功能时调用设置函数之后，要手动写寄存器优先级，不然 PWM 会错乱。普通 GPIO 的上下拉电阻用 analog_write()函数就行了。总之 IO 口相关的配置应该以《ST17H26 芯片手册 DS_ST17H2628283038-E11_Datasheet for LENZE ultra-low cost BLE SoC》为准!!**

1）、设置 GP18 为按键输入 GPIO 口，并使用内部下拉 100k 欧姆电阻的配置方式 。
2）、设置 GP7 为 LED 灯输出 GPIO 口，并使用下拉 100k 欧姆的配置方式

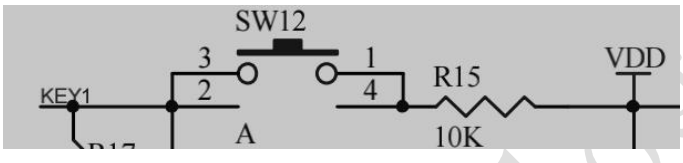
```
//  power_mode = Mode_Power_On;

    gpio_set_func(GPIO_GP18, AS_GPIO);
    gpio_set_output_en(GPIO_GP18,0);
    gpio_set_input_en(GPIO_GP18,1);
    analog_write (0x08, 0x0f);//G17  G18****100K
    gpio_set_func(GPIO_GP7, AS_GPIO);
    gpio_set_output_en(GPIO_GP7,1);
    gpio_set_input_en(GPIO_GP7,0);

    led_enter_mode(1);
    buzzer_enter_mode(1);

    /*set PWM Param*/
```

3)、设置 GP10 为输出 PWM 口，并使用无上下拉电阻的配置方式，并对对应的 pwm 频率加以设置。

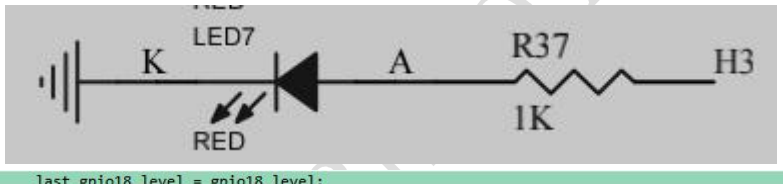
```
        u16 buzzer_freq = 727;      ///2.7K = 2M/740
//  }

    write_reg8(0x781,0x0f);//32M/(15+1) = 2M
    /* set buzzer pwm */
//  write_reg16(0x79a,727);//set pwm3 max cycle 2.7K = 2M/741
//  write_reg16(0x798,363);//set pwm3 duty_cycle = 50% = 370/1000
    write_reg16(0x79a,buzzer_freq);//set pwm3 max cycle 2.7K = 2M/741
    write_reg16(0x798,(buzzer_freq>>1));//set pwm3 duty_cycle = 50% = 370/1000

    gpio_set_output_en(GPIO_GP10,1);
    gpio_set_input_en(GPIO_GP10,0);
    gpio_set_func(GPIO_GP10,AS_PWM);
```

按键点灯：

GPIO18 接到原理图上的 KEY1，再上拉 10k 的电阻到电源端，下图程序中可以看到，在按键未按下时，gpio18 口为低电平，按键程序会跳转到 else 内的部分（下图中的第二个方框），做判断。在按键按下时，gpio18 口为高电平，按键程序会跳转到 if(button_press)（下图中的第一个方框）内的部分，设置 led 的闪灯模式为 3。



GPIO7 接到下图中 H3 的位置，当输出高电平时，灯亮，否则灭灯。




```
    last_gpio18_level = gpio18_level;

    u8 button_pressed = /*gpio17_level_inv || gpio18_level_inv || gpio22_level || gpio17_level ||*/
    if(button_pressed){
        if((!last_button_pressed)&&(0==btnClock)){
            btnClock=1;
            fisTime=clock_time();
            }
        if(!last_button_pressed && (!button_check_time || clock_time_exceed(last_button_release_tim
            buzzer_enter_mode(3);
            led_enter_mode(3);
            FFE1_value[0] = 0x01;
//          blt_push_notify(10,FFE1_value[0],1);
            last_button_pressed = 1;
            last_button_press_time = clock_tick;
            blt_disable_latency();
            //blt_push_notify(30, 0x01, 1);
            proximity_le_mode = 0;
            selfie_adv_mode_start_tick = clock_tick;
        }
         if(clock_time_exceed(last_button_press_time,900*1000)){  //last_button_press_time,1500ms
                FFE1_value[0] = 0x02;
//              blt_push_notify(10,FFE1_value[0],1);
         }
         if(!last_button_pressed &&(clock_time_exceed(last_button_release_time,65*1000))){
                FFE1_value[0] = 0x03;
                last_button_pressed = 1;
//              blt_push_notify(10,FFE1_value[0],1);
         }
    }
    else {
        if(last_button_pressed && clock_time_exceed(last_button_press_time, 80*1000)){
            last_button_pressed = 0;
            last_button_release_time = clock_tick;
        }
    }
    if((btnClock==1)&&clock_time_exceed(fisTime,1000*1000)){
            btnClock=2;
            blt_push_notify(10,FFE1_value[0],1);
            btnClock=0;
    }
```

长按，短按，连按按键的判断方法：

下图所示的程序是一个长短按的判定程序框架。FFE1_value 值 01 代表单击按键的情况，FFE1_value 值 02 代表长按按键的情况，FFE1_value 值 03 代表双击按键的情况。只要此次按键按下的时间和上次的时间间隔超过 500ms，我们就认为发生一次单击事件（实际的间隔用户可以自己调整）。当按键按下的时间超过 900ms 了，我们就认为 是长按事件。当此次按键按下的时间和上次放开按键的时间超过 65ms 了，我们就认为 是双按事件。需要注意的是，从按键按下到从从端向主端的发送按键数据，我们要等够一秒，才能确定实际是长按还是短按。最终通过 blt_push_notify(15，FFE0_value，1)；函数的对应 Handle =15 的 0xFFE1 UUID 发送键值（截图中的数值不对，以文字描述为准）。如果实际需要长按的时间超过 5s ，那么就需要对按键判定的时间节点(下图黄圈中的数)加以变通。

```
u8 button_pressed = /*gpio17_level_inv || gpio18_level_inv || gpio22_level || gpio17_level ||*/ gpio18_lev
if(button_pressed){
    if((!last_button_pressed)&&(0==btnClock)){
        btnClock=1;
        fisTime=clock_time();
        }
    if(!last_button_pressed && (!button_check_time || clock_time_exceed(last_button_release_time, 500*1000
        buzzer_enter_mode(3);
        led_enter_mode(3);
        FFE1_value[0] = 0x01;
//      blt_push_notify(10,FFE1_value[0],1);
        last_button_pressed = 1;
        last_button_press_time = clock_tick;
        blt_disable_latency();
        //blt_push_notify(30, 0x01, 1);
        proximity_le_mode = 0;
        selfie_adv_mode_start_tick = clock_tick;
    }
     if(clock_time_exceed(last_button_press_time,900*1000)){  //last_button_press_time,1500ms  last_button
          FFE1_value[0] = 0x02;
//        blt_push_notify(10,FFE1_value[0],1);
     }
     if(!last_button_pressed &&(clock_time_exceed(last_button_release_time,65*1000) ){
          FFE1_value[0] = 0x03;
          last_button_pressed = 1;
//        blt_push_notify(10,FFE1_value[0],1);
     }
}
else {
        if(last_button_pressed && clock_time_exceed(last_button_press_time, 80*1000)){
            last_button_pressed = 0;
            last_button_release_time = clock_tick;
        }
}
if((btnClock==1)&&clock_time_exceed(fisTime,1000*1000)){
        btnClock=2;
        blt_push_notify(10,FFE1_value[0],1);
        btnClock=0;
}
```

数传 ----设备接收来自 alert UUID 的报警指令 ，以及发送对应的按键信息

本节详细描述上节所讲的 blt_push_notify(15，FFE0_value，1)；

对应的 handle 值，是我们在 app_att.c 的下图所示部分所列出的第 N 个条目。如下图所示，我们今天使用的 UUID 是 FFE1 对应的 handle = 15；在本例中，FFE1 对应的 Property 为 Read 以及 Notify。主要是用在从端发送数据到主端的时候。

```
08 static const u16 FFE0_UUID = 0xffe0;
09 static const u16 FFE1_charUUID = 0xffe1;
10 static const u8  FFE1_prop = CHAR_PROP_READ | CHAR_PROP_NOTIFY;
11 //
12
```

```
const attribute_t my_Attributes[] =
{
//DFSADF
    {17,0,0,0,0}, //

/*****************************************************************************
                        GenericAttribute(Services)
*****************************************************************************/
    {4,2,2,(u8*)(&my_primaryServiceUUID),          (u8*)(&my_gattServiceUUID)},
    {0,2,1,(u8*)(&my_characterUUID),               (u8*)(&PROP_INDICATE)},
    {0,2,4,(u8*)(&my_serviceChangeUUID),           (u8*)(serviceChangeVal)},
    {0,2,2,(u8*)(&clientCharacterCfgUUID),         (u8*)(serviceChangeCCC)},
/*****************************************************************************
                       GenericAccess(Services)
*****************************************************************************/
    // gap, 5
    {5,2,2,(u8*)(&my_primaryServiceUUID),      (u8*)(&my_gapServiceUUID)},
    {0,2,1,(u8*)(&my_characterUUID),      (u8*)(&PROP_READ)},
    {0,2,sizeof (my_appearance), (u8*)(&my_appearanceUIID),  (u8*)(&my_appearance)},
    {0,2,1,(u8*)(&my_characterUUID),      (u8*)(&PROP_READ)},
    {0,2,sizeof (my_periConnParameters),(u8*)(&my_periConnParamUUID),   (u8*)(&my_periConnParameters)},
    /*****************************************************************************
            10  Immediate Alert  (Services)
    *****************************************************************************/
        {3,2,2,(u8*)(&my_primaryServiceUUID),   (u8*)(&immediateAlert_serviceUUID)},
        {0,2,1,(u8*)(&my_characterUUID),        (u8*)(&immediateAlertLevel_prop)},
        {0,2,1,(u8*)(&alertLevel_charUUID),         (u8*)(&immediateAlertLevel_value)},  // handle=7
    //  {0,2,2, (u8*)(&clientCharacterCfgUUID),    (u8*)(immediateAlertLevel_valueInCCC)},  //value
    /*****************************************************************************
            13   Private  (Services)
    *****************************************************************************/
        {4,2,2,(u8*)(&my_primaryServiceUUID),   (u8*)(&FFE0_UUID)},
        {0,2,1,(u8*)(&my_characterUUID),        (u8*)(&FFE1_prop)},
        {0,2,sizeof(FFE1_value),(u8*)(&FFE1_charUUID),      (u8*)(&FFE1_value)},
        {0,2,sizeof(FFE2_value),    (u8*)(&clientCharacterCfgUUID),    (u8*)(FFE2_value)},
/*****************************************************************************
                          OTA (Services)
*****************************************************************************/
```

handle = 15

上面我们看到了数据的单向通信方式，下面我们看看 UUID 是 alertLevel_charUUID 的 handle，数据的互通互有是怎么实现的呢？

```
static const u8 immediateAlertLevel_prop = CHAR_PROP_READ | CHAR_PROP_WRITE | CHAR_PROP_WRITE_WITHOUT_RSP;//
u8 immediateAlertLevel_value = 0;
u8 immediateAlertLevel_valueInCCC[2];
```

手机端通过 alert_level 对应 UUID，发送一串数据到从端。从端收到后，在下图中的系统内部回调函数中加以处理，完成数据传输的一步。

```c
/*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*/
int att_write_cb(void*p)
{

    rf_packet_att_write_t *att_req = p;

    u16 att_handle = att_req->handle | (att_req->handle1  << 8);

    u16 ret_handle = ATT_NO_HANDLED;
     if(12 == att_handle){  // IMMEDIATE Write callback
        u8 write_value = att_req->value[0];
        if(write_value == 2){   // || write_value == 2
            buzzer_enter_mode(2);
            led_enter_mode(4);
//              led_ui_buffer_MS[3].next_mode = 4;
            buzzer_ui_buffer_MS[2].offOn_Ms[0] = 180;
//              buzzer_ui_buffer_MS[3].next_mode = 2;
        }
        else if(write_value == 1){
            buzzer_enter_mode(2);
            led_enter_mode(2);
            buzzer_ui_buffer_MS[2].offOn_Ms[0] = 1000;
        }
        else {
            buzzer_enter_mode(0);
            led_enter_mode(0);
        }
    }



    ret_handle = att_handle;

//  return ATT_NO_HANDLED;///[illegible]
    return ATT_HANDLED;
}
```