

# telink-mesh灯控实现

## mesh 实现方式

### 消息转发

APP依次与每个单灯建立蓝牙连接，然后设置mesh信息。（APP每次只能连接一个节点）

APP+Light方式：APP发出的所有控制消息都是直接发送到APP直连的节点上，然后直连节点根据消息的目的地址决定是否处理该消息或是转发该消息。

Max Bridge Connect Number默认是8， 同一个消息最多会被重复发送8次，以确保目标结点收到该消息。

Max Relay Number 默认是3， 也就是说消息离开直连接点后最多被非直连接点转发三次，对于较远的节点按照如下方式传递消息。

APP -> light1(直连接点) -> light2(1st relay) -> light3(2nd relay) -> light4(3rd relay) -> light5

### 节点分组

group概念就是该节点订阅哪个group地址的消息，每个节点最多可以设置八个分组。

当收到的消息目的地址是group地址时，与此同时该节点包含该分组，则处理消息。组的概念只是相当于给节点多加了几个地址，当对组里的节点控制时，与单节点控制并没有什么区别，只是消息的目的地址为组地址，而且处理该消息的节点可能有多（包含该分组的节点都会处理消息）。

地址用两个字节表示，最高位为1时，表示group地址。

```
group1地址 -> 0x80 01
group2地址 -> 0x80 02
group3地址 -> 0x80 03
```

APP直连节点时，直接对该节点设置分组情况，add group或者 delete group。

### PWM控制

灯相关的控制都是通过 (R, G, B, Lum) 四个变量实现的，表示灯的颜色及亮度。无论是颜色还是亮度都是通过调整相应IO口的PWM波占空比来实现的。PWM波由max\_ticks、cmp\_ticks两个量控制，max\_ticks表示脉冲周期的持续时间，cmp\_ticks表示周期内高电平保持时间。在telink灯控示例中，max\_ticks保持恒定，PWM波的调整都是通过改变cmp\_ticks的值来实现的。

可以简单的这样理解，对于灯的R通道来说，其颜色值R与亮度值Lum得乘积表示cmp\_ticks（即占空比），颜色越深或者亮度越大占空比越高。

```
/* 设置PWM占空比 */
void pwm_set_lum (int id, u16 y, int pol)
{
    u32 lum = ((u32)y * PWM_MAX_TICK) / (255*256);

    pwm_set_cmp (id, pol ? PWM_MAX_TICK - lum : lum);
}
```

```

}

/* 根据某通道的颜色值及亮度值，计算出占空比 */
u32 get_pwm_cmp(u8 val, u8 lum){
    if(lum >= ARRAY_SIZE(rgb_lumen_map) - 1){
        lum = ARRAY_SIZE(rgb_lumen_map) - 1;
    }
    u16 val_lumen_map = rgb_lumen_map[lum];

    #if LIGHT_ADJUST_STEP_EN
        light_step_correct_mod(&val_lumen_map, lum);
    #endif
    u32 val_temp = val;
    return (val_temp * val_lumen_map) / 255;
}

/* 灯设备R通道设置 */
void light_adjust_R(u8 val, u8 lum){
    pwm_set_lum (PWMID_R, get_pwm_cmp(val, lum), 0);
}

/* 节点灯设备控制 */
void light_adjust_RGB_hw(u8 val_R, u8 val_G, u8 val_B, u8 lum){
    light_adjust_R(val_R, lum);
    light_adjust_G(val_G, lum);
    light_adjust_B(val_B, lum);
}

```

上面灯的控制是立刻生效的，telink还提供了渐变模式，即灯颜色或亮度的变化渐变到目标状态。

渐变控制的生效是在定时器函数 `light_onoff_step_timer()` 里面实现的，使用一个全局变量 `light_step` 表示渐变参数，需要改变状态时只需要修改该全局变量的值，定时器会自动扫描该量，然后去生效状态改变。

## 闹钟(定时)任务

### telink-8269中的时间表示

telink-8269中没有rtc硬件模块，系统时间重启后归零。

上电后的时间记录在一个全局变量 `rtc` 中，在主循环中不断调用 `rtc_run()` 函数来根据系统晶振维持时间，时间变量的结构如下：

```

typedef struct{
    u16 year;
    u8 month;
    u8 day;
    u8 hour;
    u8 minute;
    u8 second;
    u8 week;
    u32 tick_last; //if add element, must after tick_last
}rtc_t;

```

```
rtc_t rtc = {
    .year = 1970,
    .month = 1,
    .day = 1,
    .hour = 0,
    .minute = 0,
    .second = 0,
    .week = 1,
};
```

## 定时轮询

telink方案中定时（闹钟）任务最多可以设16个，在一个长度为16的数组全局变量中保存，定时任务的结构如下：

```
typedef struct{ // max 10BYTES
    union {
        u8 event;
        u8 valid_flag;
    }par0;
    u8 index;
    struct {
        u8 cmd : 4;
        u8 type : 3;
        u8 enable : 1;
    }par1;
    u8 month;
    union {
        u8 day;
        u8 week;    // BIT(n)
    }par2;
    u8 hour;
    u8 minute;
    u8 second;
    u8 scene_id;
    //u8 alarm total for notify
}alarm_ev_t;
```

在时间保持函数 `rtc_run()` 中，会每秒调用一次 `rtc_increase_and_check_event()` 函数，该函数会调用 `alarm_event_check()`，而该函数将当前时间rtc与闹钟任务中的时间做对比，到达定时时间时就会激发相应的操作。

目前的定时任务，只有三种动作：开、关、情景模式生效。

## 控制消息结构

```
typedef struct{
    u8 sno[3];
    u8 src[2];
    u8 dst[2];
    u8 val[23]; // op[1~3], params[0~10], mac-app[5], ttl[1], mac-net[4]
    // get status req: params[0]=tick mac-app[2-3]=src-mac1...
    // get status rsp: mac-app[0]=ttc mac-app[1]=hop-count
}rf_packet_att_value_t;
```

member	note
sno[3]	每条消息对应一个特定的sno
src[2]	原地址
dst[2]	目的地址
op[3]	操作码（目前为3个字节），op[0]操作类型，op[1]op[2] vendor_id
param[0~10]	参数

比如All\_on消息表示打开所有灯，命令消息表示为 11 11 11 11 00 ff ff d0 11 02 01 01 00，消息各字段意义如下：

member	bytes	note
sno[3]	0x11 0x11 0x11	每个消息唯一
src	0x11 0x00	源地址
dst	0xff 0xff	目的地址
op[3]	0xd0 0x11 0x02	0xd0表示开关操作，0x11 0x02表示公司ID 0x211
param	01 01 00	0x01表示开，0x01 0x00表示延时1ms

当节点收到目的地址是自身地址或者是自身订阅的组地址的消息时，会调用 rf\_link\_data\_callback (u8 \*p) 函数，在此函数内用一系列 if-else 结构根据 op[0] 处理消息，由于 op[0] 的最高两位固定为11，所以用其低六位来比较 op = op[0]&0x3f; 。

```
void rf_link_data_callback (u8 *p)
{
    if(op == LGT_CMD_LIGHT_ONOFF){
        /* 开关类消息处理 */
    }
    else if (op == LGT_CMD_LIGHT_CONFIG_GRP){
        /* 组控制 */
    }

    else if (op == LGT_CMD_CONFIG_DEV_ADDR){
```

```

        /* 设备地址设置 */
    }
    else if(op == LGT_CMD_LIGHT_SET){
        /* 设置灯亮度 */
    }
    else if(op == LGT_CMD_LIGHT_RC_SET_RGB){
        /* 设置灯颜色（根据颜色表） */
    }
    else if (op == LGT_CMD_SET_RGB_VALUE){
        /* 设置灯颜色（根据RGB值） */
    }
    else if (op == LGT_CMD_KICK_OUT){
        /* 节点踢出mesh */
    }
    else if (op == LGT_CMD_SET_TIME){
        /* 设置设备时间 */
    }
    else if (op == LGT_CMD_ALARM){
        /* 闹钟设置 */
    }
    else if (op == LGT_CMD_SET_SCENE){
        /* 设置情景模式 */
    }
    else if (op == LGT_CMD_LOAD_SCENE){
        /* 载入情景模式 */
    }
    else if (op == LGT_CMD_NOTIFY_MESH){
        /* mesh通知消息 */
    }
    else if (op == LGT_CMD_MESH_OTA_DATA){
        /* OTA升级 */
    }
}

```

## 开关消息

开：RGB=(255, 255, 255), Lum=100 关：RGB=(0, 0, 0), Lum=0

```

if(op == LGT_CMD_LIGHT_ONOFF){
    if(params[0] == LIGHT_ON_PARAM){
        light_onoff(1);
    }else if(params[0] == LIGHT_OFF_PARAM){
        light_onoff(0);
    }
}
}

```

## 组控制消息

组控制消息分两类，新建组和删除组，这里引用的库函数实现，源码不可见。

```

else if (op == LGT_CMD_LIGHT_CONFIG_GRP){

```

```

u16 val = (params[1] | (params[2] << 8));
if(params[0] == LIGHT_DEL_GRP_PARAM){
    extern u8 rf_link_del_group(u16 group);
    if(rf_link_del_group(val)){
        cfg_led_event(LED_EVENT_FLASH_1HZ_4S);
    }
}else if(params[0] == LIGHT_ADD_GRP_PARAM){
    extern u8 rf_link_add_group(u16 group);
    if(rf_link_add_group(val)){
        cfg_led_event(LED_EVENT_FLASH_1HZ_4S);
    }
}
}
}

```

## 地址配置

节点的地址默认使用本节点mac地址的最后一个字节表示，也可以手动配置。

```

else if (op == LGT_CMD_CONFIG_DEV_ADDR){
    u16 val = (params[0] | (params[1] << 8));
    extern u8 rf_link_add_dev_addr(u16 deviceaddress);
    if(!dev_addr_with_mac_flag(params) || dev_addr_with_mac_match(params)){
        if(rf_link_add_dev_addr(val)){
            mesh_pair_proc_get_mac_flag();
        }
    }
}
}

```

## 设置灯亮度

当param[0]为0xFE或0xFF时分别表示音乐模式开始、关闭，即灯的亮度随音量变化，没有看到实现代码。当param[0]在0~100之间时，表示设置灯的亮度为param[0]。

```

else if(op == LGT_CMD_LIGHT_SET){
    if(music_time){
        last_music_tick = clock_time();
    }
    if(light_off){
        return;
    }
    if(params[0] == 0xFE){
        // start music
        led_lum_tmp = led_lum;
        music_time = 1;
    }else if(params[0] == 0xFF){
        // stop music
        led_lum = led_lum_tmp;
        music_time = 0;
    }else if(params[0] > 100 || is_lum_invalid(params[0]) || led_lum == params[0]){
        return;
    }else{

```

```

        led_lum = params[0];
    }
    light_adjust_RGB(led_val[0], led_val[1], led_val[2], led_lum);
    if(!music_time){
        lum_changed_time = clock_time();
        device_status_update();
    }
}

```

## 设置颜色（根据颜色表）

源码中定义了一张颜色表，APP可直接在颜色表中选取颜色来设置。

```

else if(op == LGT_CMD_LIGHT_RC_SET_RGB){
    if(light_off || params[0] > RGB_MAP_MAX){
        return;
    }
    table_map_idx = params[0];
    led_val[0] = rgb_map[table_map_idx][0];
    led_val[1] = rgb_map[table_map_idx][1];
    led_val[2] = rgb_map[table_map_idx][2];

    light_adjust_RGB(led_val[0], led_val[1], led_val[2], led_lum);

    lum_changed_time = clock_time();
}

```

## 设置颜色（根据RGB值）

根据RGB值设置颜色，当灯处于关闭状态时，设置无效。

```

else if (op == LGT_CMD_SET_RGB_VALUE)
{
    if(light_off){
        return;
    }
    if(params[0] == 1){                //R
        led_val[0] = params[1];
        light_adjust_R (led_val[0], led_lum);
    }else if(params[0] == 2){          //G
        led_val[1] = params[1];
        light_adjust_G (led_val[1], led_lum);
    }else if(params[0] == 3){          //B
        led_val[2] = params[1];
        light_adjust_B (led_val[2], led_lum);
    }else if(params[0] == 4){          //RGB
        led_val[0] = params[1];
        led_val[1] = params[2];
        led_val[2] = params[3];
        light_adjust_RGB(led_val[0], led_val[1], led_val[2], led_lum);
    }else if(params[0] == 5){          //CT

```

```

        //temporary processing as brightness
        if(light_off || params[1] > 100 || led_lum == params[1]){
            return;
        }
        led_lum = params[1];
        light_adjust_RGB(led_val[0], led_val[1], led_val[2], led_lum);
    }

    lum_changed_time = clock_time();
}

```

## 踢出网络

节点踢出网络

```

else if (op == LGT_CMD_KICK_OUT)
{
    irq_disable();
    void kick_out(u8 par);
    kick_out(params[0]);
    light_sw_reboot();
}

```

## 设置时间

设置时间是将消息传过来的时间直接赋值给全局变量 rtc

另外这里用到了 `cfg_led_event()` 还没有清楚什么意思。

```

else if (op == LGT_CMD_SET_TIME)
{
    rtc_t rtc_old, rtc_new;
    memcpy(&rtc_old, &rtc, sizeof(rtc_t));
    memcpy(&rtc_new.year, params, 7);
    if(0 == rtc_set_time((rtc_t *)params)){
        //ok
        check_event_after_set_time(&rtc_new, &rtc_old);
        cfg_led_event(LED_EVENT_FLASH_1HZ_3T);
    }else{
        //invalid params
        cfg_led_event(LED_EVENT_FLASH_4HZ_3T);
    }
}
}

```

## 闹钟

继续调用函数 `alarm_ev_callback()`，在 `alarm_ev_callback()` 函数中对闹钟设置。



```

else if (op == LGT_CMD_ALARM)
{
    if(0 == alarm_ev_callback((u8*)params)){
        cfg_led_event(LED_EVENT_FLASH_1HZ_3T);
    }else{
        cfg_led_event(LED_EVENT_FLASH_4HZ_3T);
    }
}
}

```

alarm\_ev\_callback函数如下所示

```

int alarm_ev_callback(const u8 *ev){
    alarm_ev_t* p_ev = (alarm_ev_t*)ev;
    int ret = 0;
    switch(p_ev->par0.event){
        case ALARM_EV_ADD:
            if(0 == p_ev->index){
                p_ev->index = get_next_schedule_idx();
            }

            if(alarm_par_check(p_ev) == -1){
                return -1;
            }

            ret = alarm_add(p_ev);
            break;

        case ALARM_EV_DEL:
            alarm_del(p_ev, NULL);
            break;

        case ALARM_EV_CHANGE:
            if(alarm_par_check(p_ev) == -1){
                return -1;
            }

            alarm_del(p_ev, NULL);
            ret = alarm_add(p_ev);
            break;

        case ALARM_EV_ENABLE:
        case ALARM_EV_DISABLE:
            {
                alarm_ev_t ev_rsp;
                if(ALARM_ENABLE_DISABLE_OK == alarm_del(p_ev, &ev_rsp)){
                    ev_rsp.par1.enable = ALARM_EV_ENABLE == p_ev->par0.event ? 1 : 0;
                    ret = alarm_add(&ev_rsp);
                }else{
                    ret = -1;
                }
            }
            break;
    }
}

```

```

        default :
            break;
    }

    alarm_event_check();
    return ret;
}

```

## 设置情景模式

情景模式的设置内容是RGB颜色和亮度，包括情景模式添加、删除和装载。泰凌微方案中情景模式是保存在flash固定区域的，最多支持16个情景模式，设置情景模式及将相应的模式保存到flash中，需要某个情景模式时，根据id去读取相应位置的flash来获取情景模式的值。

```

else if (op == LGT_CMD_SET_SCENE)
{
    if(params[0] == SCENE_ADD){
        // add scene: params valid & no repetition
        scene_t *pData = (scene_t*)(params+1);
        if(pData->id && pData->lum <= 100
            && pData->rgb[0] <= 0xFF
            && pData->rgb[1] <= 0xFF
            && pData->rgb[2] <= 0xFF){
            if(scene_add(pData)){
                cfg_led_event(LED_EVENT_FLASH_1HZ_3T);
            }
        }

        }else if(params[0] == SCENE_DEL){
            // del scene
            if(scene_del(params[1])){
                cfg_led_event(LED_EVENT_FLASH_1HZ_3T);
            }
        }
    }
else if (op == LGT_CMD_LOAD_SCENE)
{
    scene_load(params[0]);
}

```

## OTA升级

该部分还没细看

```

else if (op == LGT_CMD_MESH_OTA_DATA)
{
    u16 idx = params[0] + (params[1] << 8);
    if(is_not_master_sending_ota_st()){ // no update firmware for itself
        if(CMD_START_MESH_OTA == idx){

            //mesh_ota_master_start_firmware_from_20000();

```

```

        mesh_ota_master_start_firmware_from_own();
        //cfg_led_event(LED_EVENT_FLASH_1HZ_4S);
    }else if(CMD_STOP_MESH_OTA == idx){
        if(is_mesh_ota_slave_running()){
            // reboot to initial flash: should be delay to relay command.
            mesh_ota_slave_reboot_delay(); // reboot after 320ms
        }
    }else{
        if(mesh_ota_slave_save_data(params)){
            static u16 mesh_ota_error_cnt;
            mesh_ota_error_cnt++;
        }
    }
}
}
}
}

```