

基于 FR801xH 的多连接组网

一、多连接实现

1.1 综述

本节内容旨在指导用户基于 FR801xH SDK 软件开发框架快速开发一个多连接的工程，我们将基于 SDK 中的例程 ble_multi_role 实现多连接，ble_multi_role 本身是一个主从一体的例程，作为从机被其他主机（例如手机）搜索连接的同时，也可以搜索并连接其他从机设备。

1.2 修改链接数

```
void user_init_static_memory(void)
{
    initial_static_memory(22, 1, 20, 20, 27, 20, 27, 254, 0x800); //1 条广播链路 20 条连接 20
    个链路层接收缓冲, 链路层最大接收长度 27 20 路发送缓冲 发送长度 27 广播长度 254 堆
    栈深度 0x800
}
```

在 proj_main.c 中添加以上函数，来修改链接数，SDK 默认链接数为 6，这里修改成 20。（函数中的参数配置，目前没有开放，请按照我们推荐参数修改，不建议随意修改。）

1.3 修改 app_gap_evt_cb

app_gap_evt_cb 是 gap event 回调函数，

GAP_EVT_ADV_REPORT: 解析 scan 到的广播, 判断广播内容

```

case GAP_EVT_ADV_REPORT:
{
    uint8_t scan_name[] = "Simple Multi Role";
    if (p_event->param.adv_rpt->data[0] == 0x12
        && p_event->param.adv_rpt->data[1] == GAP_ADVTYPE_LOCAL_NAME_COMPLETE
        && memcmp(&(p_event->param.adv_rpt->data[2]), scan_name, 0x11) == 0)
    {
        #if defined (MASTER)
            if((p_event->param.adv_rpt->src_addr.addr.addr[5] ==
                0xff)&&(p_event->param.adv_rpt->src_addr.addr.addr[4] != 0xff)){//Connect dev_A ==
                if(connect_state == 0){
                    gap_stop_scan();
                    gap_start_conn(&(p_event->param.adv_rpt->src_addr.addr),
                        p_event->param.adv_rpt->src_addr.addr_type,
                        80, 80, 0, 500);//作为 master 开始连接 slaver_A
                    connect_state = 1;
                    co_printf("gap_start_conn\r\n");
                }
            }
            #elif defined (SLAVER_A)
                mac_addr_t local_addr;
                gap_address_get(&local_addr);
                if((p_event->param.adv_rpt->src_addr.addr.addr[4] ==
                    local_addr.addr[4])&&(p_event->param.adv_rpt->src_addr.addr.addr[5] != 0xff)){//Connect dev_B ==
                    if(connect_state == 0){
                        gap_stop_scan();
                        gap_start_conn(&(p_event->param.adv_rpt->src_addr.addr),
                            p_event->param.adv_rpt->src_addr.addr_type,
                            80, 80, 0, 500);//作为 slaver_A 开始连接 slaver_B
                        connect_state = 1;
                        co_printf("gap_start_conn\r\n");
                    }
                }
            }
        }
    }
}
break;

```

GAP_EVT_MASTER_CONNECT: 主机连接成功的回调 event, 缓存 conidx 以及 ID

```

case GAP_EVT_MASTER_CONNECT:
{
    extern uint8_t client_id;
    gatt_discovery_all_peer_svc(client_id,p_event->param.master_encrypt_conidx);
    for(uint8_t i=0;i<12;i++){
        if(multirol.slave_id[i].dev_id == 0){
            multirol.slave_id[i].dev_id =
                (p_event->param.master_connect.peer_addr.addr[4] << 8)
                |p_event->param.master_connect.peer_addr.addr[5];
            multirol.slave_id[i].link_idx = p_event->param.master_connect.conidx ;
            co_printf("link[%d] devID = %x\r\n",multirol.slave_id[i].link_idx,
                multirol.slave_id[i].dev_id);
            break;
        }
    }
    multirol.slave_num++;
    co_printf("multirol.slave_num = %d\r\n",multirol.slave_num);
}
break;

```

GAP_EVT_SLAVE_CONNECT: 从机连接成功的回调 event, 缓存 conidx 以及 ID

```

case GAP_EVT_SLAVE_CONNECT:
{
co_printf("gatt_get_mtu%d  %d\r\n",p_event->param.slave_connect.conidx,gatt_get_mtu(p_event->pa
ram.slave_connect.conidx));
co_printf("con_interval:%d,con_latency  :%d\r\n",p_event->param.slave_connect.con_interval,p_event->
param.slave_connect.con_latency);
multirol.master_num ++;
multirol.master_id.dev_id = (p_event->param.slave_connect.peer_addr.addr[4] << 8)
|p_event->param.slave_connect.peer_addr.addr[5];
multirol.master_id.link_idx = p_event->param.slave_connect.conidx;
os_timer_start(&os_timer_mtu,200,false);
LED_ON;
}
break;

```

GAP_EVT_DISCONNECT: 连接断开的回调 event, 判断是否需要广播或者开始扫描

```

case GAP_EVT_DISCONNECT:
{
uint8_t i=0;
for( i=0;i<12;i++){
if((multirol.slave_id[i].link_idx ==
p_event->param.disconnect.conidx)&&(multirol.slave_id[i].dev_id !=0)){
multirol.slave_id[i].dev_id = 0;
multirol.slave_num--;
connect_state = 0;
mr_central_start_scan();//建立连接后 开始 scan
break;
}
}
if(i == 12){//slave disconnect
co_printf("Start advertising\r\n");
gap_start_advertising(0);
LED_OFF;
}
}
break;

```

1.4 profile 数据收发

数据接收:

```
static uint16_t mr_central_msg_handler(gatt_msg_t *p_msg)//从机向主机发送数据
{
    switch(p_msg->msg_evt){
        case GATTC_MSG_NTF_REQ:
            {
                if(p_msg->att_idx == 0)
                {
                    #if defined(MASTER)
                    uart_write(UART0, p_msg->param.msg.p_msg_data, p_msg->param.msg.msg_len);
                    #elif defined(SLAVER_A)

                    gatt_client_write_to_master(multirol.master_id.link_idx,p_msg->param.msg.p_msg_data,p_msg->param.msg.msg_len);
                    #endif
                }
            }
            break;
    }
}

static uint16_t sp_gatt_msg_handler(gatt_msg_t *p_msg)
{
    switch(p_msg->msg_evt)
    {
        case GATTC_MSG_READ_REQ:
            sp_gatt_read_cb((uint8_t *)p_msg->param.msg.p_msg_data),
            &(p_msg->param.msg.msg_len), p_msg->att_idx);
            break;

        case GATTC_MSG_WRITE_REQ:
            sp_gatt_write_cb((uint8_t*)(p_msg->param.msg.p_msg_data),
            (p_msg->param.msg.msg_len), p_msg->att_idx);
            break;

        default:
            break;
    }
    return p_msg->param.msg.msg_len;
}
```

数据发送

```
void gatt_client_write_to_master(uint8_t conidx, uint8_t *data, uint8_t len)//从机向主机发送数据
{
    gatt_ntf_t ntf_att;
    ntf_att.att_idx = SP_IDX_CHAR4_VALUE;
    ntf_att.conidx = conidx;
    ntf_att.svc_id = sp_svc_id;
    ntf_att.data_len = MIN(len,gatt_get_mtu(conidx) - 3);
    ntf_att.p_data = data;
    gatt_notification(ntf_att);
}

void gatt_client_write_to_slaver(uint8_t conidx,uint8_t *buffer,uint16_t len)//主机向从机发送数据
{
    gatt_client_write_t write;
    write.conidx = conidx; //GATT 事件所处的链接的链接号
    write.client_id = client_id; //profile 由底层分配的 profile id 号
    write.att_idx = 1; //RX //本操作针对 UUID 数组序号 1 对应的属性
    write.p_data = buffer; //进行写操作要发送的数据 buffer
    write.data_len = MIN(len,gatt_get_mtu(conidx) - 3); //写操作的数据长度

    gatt_client_write_cmd(write);
}
```

1.5 UART0 初始化

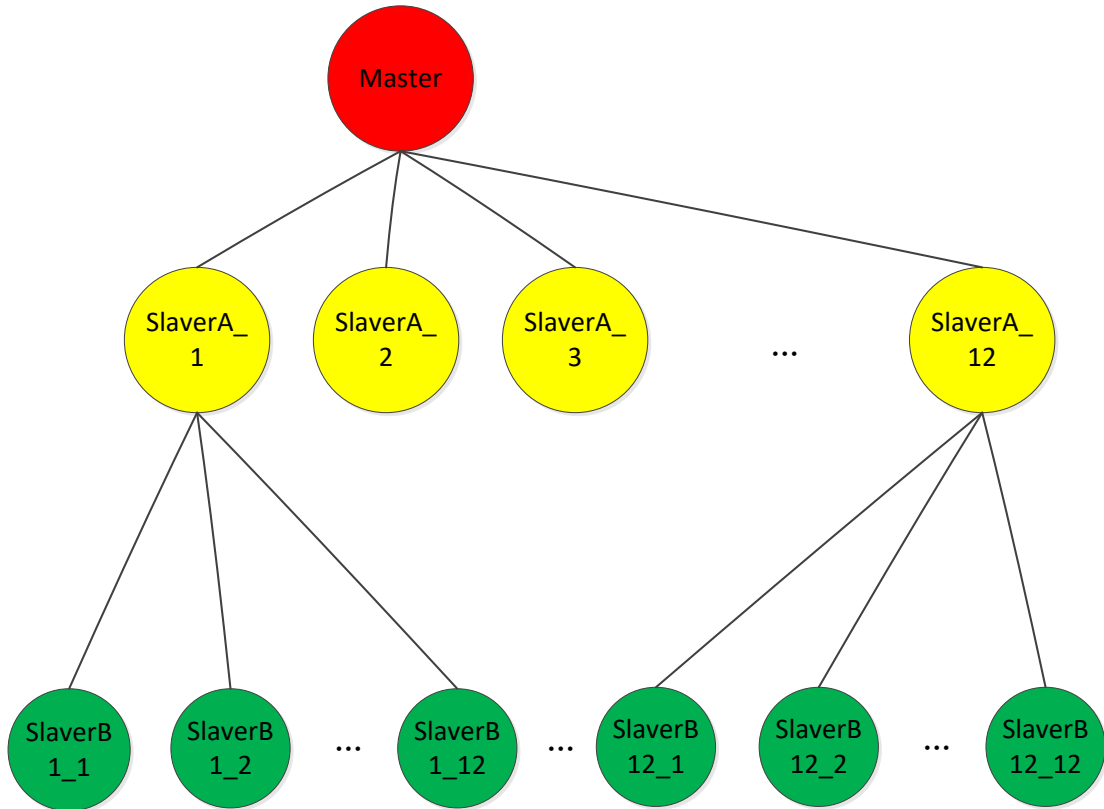
```

system_set_port_pull(GPIO_PD4, true);
system_set_port_mux(GPIO_PORT_D, GPIO_BIT_4, PORTD4_FUNC_UART0_RXD);
system_set_port_mux(GPIO_PORT_D, GPIO_BIT_5, PORTD5_FUNC_UART0_TXD);
uart_init(UART0, BAUD_RATE_115200);

NVIC_EnableIRQ(UART0_IRQn);
    
```

二、FR8016H 多连接拓扑结构

实现多连接后，我们开始实现多连接组网



多连接组网采用一拖 12 从节点的连接方式连接，其中 12 个从节点 (slaverA)，每个节点又拖 12 个节点的方式连接。实现 157 个 FR8016H 节点连接。

三、节点功耗测试数据

3.1 slaverA 节点在保持连接情况下的功耗数据

slaverA 连接 12 个 slaverB	slaverA 电流/mA
40ms(interval:32)	3.12mA
slaverA 连接 12 个 slaverB,同时连接一个 Master(40ms)	3.65mA

3.2 slaverB 节点在不同连接间隔的情况下对应的功耗数据

slaverB 连接间隔	slaverB 电流/uA
1s(interval:32,latency:24)	47.5
500ms(interval:40,latency:10)	71
200ms(interval:32,latency:5)	141
100ms(interval:40,latency:2)	254
40ms(interval:32,latency:0)	580

四、通信速率测试

	连接间隔(ms)	电流	通信速率 (byte/s)
M			
SA	40	3.4mA	
SB	40	580uA	2k
M			
SA	40	3.4mA	
SB	100	254uA	2k
M			
SA	100	1.13mA	
SB	100	254uA	2k

五、测试效果图

