FR801xH 如何构建多从机工程

Bluetooth Low Energy SOC

www.freqchip.com

Contents

1 综述 3
2 引用必要组件 3
3 入口函数 4
4 按键响应处理 5
5 GAP 事件回调函数 6
6 Profile 数据收发 7
7 工程小结 9

1 综述

本文档旨在指导用户基于 801xH SDK 软件开发框架快速开发一个多从机的工程。在阅读本文档内容之前，推荐事先阅读《Fr8010x H 如何构建系统》文档，了解在 801xH SDK 软件开发框架下，应用工程如何建立和配置，应用程序入口函数，应用程序流程跳转，以及错误和处理措施。

构建一个多从机的系统分为以下几步

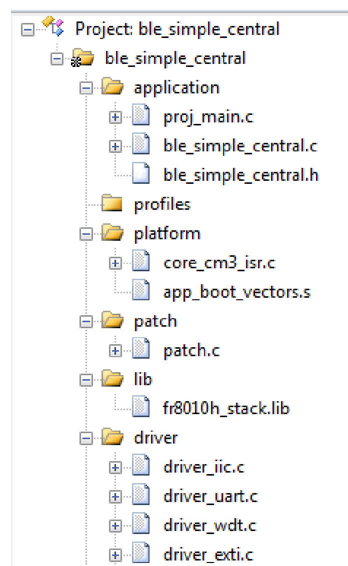
- 引用必要组件
- 入口函数初始化
- 按键响应处理
- GAP 事件回调函数
- 链接加密绑定操作
- Profile 数据接收与发送

下面几节将分别介绍每一步骤的详细过程。

2 引用必要组件

构建一个多从机系统会使用到 BLE 5.0 协议栈组件，外设驱动组件和非抢占式操作系统组件 三部分。

根据文档《Fr8010x H 如何构建系统》第二章介绍的步骤，创建一个 keil 的应用工程，然后将以上 3 部分组件的源代码或 lib 库文件引用到工程中。正常情况下，项目的目录树结构如下图所示



多从机工程项目的示例目录结构

该示例项目中包含了以下组成部分：

- library 目录下协议栈底层 lib 库。
- 中间件 modules 目录中 platform 和 patch 文件夹下 c 文件，
- Drivers 目录下，所有的外设驱动 c 文件
- 用户应用层程序组件。

在用户应用层组件中，一般使用 Proj_main.c 文件实现入口函数，ble_simple_central.c 文件实现 gap 事件回调函数处理，profile 创建 以及 profile 事件回调函数。

按照文档《Fr8010x H 如何构建系统》第二章介绍的步骤设置好项目的编译与链接选项后，项目工程就建立完毕。

下面通过一个示例工程来介绍如何通过调用协议栈组件和外设驱动组件实现多从机的系统。

该示例工程的具体功能是，通过按键 PD6，PD7 按下后，在按键中断服务程序内，主动连接两个不同 mac 地址的对端设备，连接上对端设备后，对链接进行绑定或加密操作，加密完成后，扫描对端服务集合，扫描动作结束后，在对对端服务的某个 UUID 属性进行读写操作。

3 入口函数

在第 3 个入口函数内，能调用所有的组件函数，包括协议栈组件和外设驱动组件，在本示例工程中，第 3 个入口函数内主要做如下初始化动作：初始化按键，设置本地设备名字，设置 GAP 事件回调函数，配置绑定管理功能，初始化绑定过程参数，创建 client profile。

示例代码

```
void user_entry_after_ble_init(void)
{
    os_timer_init(&button_anti_shake_timer, button_anti_shake_timeout_handler, NULL);      //初始化按键防抖软件定时器
    pmu_set_pin_pull(GPIO_PORT_D, BIT(6)|BIT(7), true);      //配置 PD6 和 PD7 脚默认为内部拉高，处于高电平
    pmu_port_wakeup_func_set(GPIO_PD6|GPIO_PD7);      //设置 PD6，PD7 做为按键，管脚拉低进中断服务程序。
                                                      //该按键中断函数为 pmu 模块功能，不担心 sleep 会禁止中断。
    uint8_t local_name[] = "Simple Central";
    gap_set_dev_name(local_name, sizeof(local_name));      //设置本地设备的名字
    gap_set_cb_func(app_gap_evt_cb);      //设置 GAP 事件回调函数
    gap_bond_manager_init(0x32000,0x33000,8,true);      //使能协议栈的绑定管理功能，设置 flash 地址 0x32000 存储绑
                    //定信息，flash 地址 0x33000 存储对端服务信息，一共支持 8 个不同 mac 地址的设备绑定信息存储。
    gap_security_param_t param =
    {
        .mitm = false,      //不需要中间认证环节
        .ble_secure_conn = false,      //不启用 security_connection 加密。
        .io_cap = GAP_IO_CAP_NO_INPUT_NO_OUTPUT,      //本机设备没有 IO 输入输出能力。
        .pair_init_mode = GAP_PAIRING_MODE_WAIT_FOR_REQ,
        .bond_auth = true,      //需要绑定
        .password = 0,
    };
    gap_security_param_init(&param);      //设置绑定过程为双方均无需 pin 码认证。
    gatt_client_t client;
    client.p_att_tb = client_att_tb;      //client profile 感兴趣的 UUID 数组
    client.att_nb = 2;      //UUID 数组的元素个数只有 2 个。
    client.gatt_msg_handler = simple_central_msg_handler;      //设置 client profile 的 gatt 事件回调处理函数
    client_id = gatt_add_client(&client);      //创建 client profile 并返回分配的 profile id。
}
```

上面初始化代码中，注意事项如下：

1. 配置绑定管理的 flash 地址时，要注意该地址必须是 flash 页面的起始地址，即需要时 0x1000 的整数倍，如果有 OTA 功能的话，两个 flash 起始地址都要设置在 OTA 第二个分区地址之后，以免被 OTA 的备份程序覆盖。
2. 设置了 GAP 事件的回调处理函数 void app_gap_evt_cb(gap_event_t *p_event)，底层产生 GAP 事件时，会直接执行该回调函数，应用层需要在该回调函数内部对不同的事件分支做处理，保证程序能够继续执行。
3. 创建 client profile 时需要用到一个 UUID 的数组 client_att_tb 变量。定义在 ble_simple_central.c 内部。该变量存放 Client profile 需要进行操作属性 UUID 列表，示例代码如下。

```
const gatt_uuid_t client_att_tb[] =
{
    [0]   ={ UUID_SIZE_2, UUID16_ARR(0xFFF1)},
    [1]   ={ UUID_SIZE_2, UUID16_ARR(0xFFF2)},
};
```

需要对 UUID 分别为 0xFFF1 和 0xFFF2 的属性进行操作。

4 按键响应处理

在第 3 个入口程序对按键 PD6、PD7 初始化之后，如果 PD6 和 PD7 被按下(变低电平)，则会进入 pmu 外部中断服务程序，我们在该中断服务程序内读取按键，然后对不同的按键做不同的处理。

示例

```
static os_timer_t button_anti_shake_timer;                //按键防抖软件定时器
static uint32_t curr_button_before_anti_shake = 0;        //记录防抖定时器启动之前，按下的键值
static void button_anti_shake_timeout_handler(void *param) {        //防抖软件定时器时间到执行函数
    uint32_t pmu_int_pin_setting = ool_read32(PMU_REG_PORTA_TRIG_MASK);
    uint32_t gpio_value = ool_read32(PMU_REG_GPIOA_V);
    gpio_value &= pmu_int_pin_setting;
    gpio_value ^= pmu_int_pin_setting;                    //获取此时的按键按下键值
    if(gpio_value == curr_button_before_anti_shake) {     //如果当前的按键值与 10ms 防抖时间之前记录的按键值相同则执行
            If(gpio_value == GPIO_PD6)
            gap_start_conn(mac_addr1,0,12, 12, 0, 300);         //主动连接地址为 mac_addr1 的对端设备
        else if(gpio_value == GPIO_PD7)
            gap_start_conn(mac_addr2,0,12, 12, 0, 300);         //主动连接地址为 mac_addr2 的对端设备
    }
}
__attribute__((section("ram_code"))) void pmu_gpio_isr_ram(void) {   //pmu 外部中断服务程序，PD6/PD7 电平变化时进入。
    uint32_t pmu_int_pin_setting = ool_read32(PMU_REG_PORTA_TRIG_MASK);
    uint32_t gpio_value = ool_read32(PMU_REG_GPIOA_V);
    ool_write32(PMU_REG_PORTA_LAST, gpio_value);
    uint32_t tmp = gpio_value & pmu_int_pin_setting;
    curr_button_before_anti_shake = tmp^pmu_int_pin_setting;        //记录第一次的按键按下键值
    os_timer_start(&button_anti_shake_timer, 10, false);           //启动 10ms 的防抖软件定时器。
}
```

按键响应处理加入了一个 10ms 的防抖软件定时器，在防抖定时器 10ms 时间到时，再次读取按键值，两次按键值一致就开始处理按键的响应事件。按下 PD6，主动连接设备 1，按下 PD7，主动连接设备 2.

发起主动连接的动作后，连接动作结束以及链接建立时，底层会上传响应的 GAP 事件，应用层应该在 GAP 事件回调函数内对这两个分支做进一步的处理。下一节介绍 GAP 事件回调处理。

另外，主动连接设备的动作必须等待上一次连接动作结束才能进行，如果同时进行两个连接动作，后一个会报错，错误码为 0x9B(LL_ERR_ACL_CON_EXISTS）。 连接动作结束的标志是，底层上传 GAP 事件: 主动连接动作结束的。

5 GAP 事件回调函数

在前面第 3 步中，入口函数初始化时，设置了 GAP 事件回调函数 void app_gap_evt_cb(gap_event_t *p_event)，该函数处理底层上传的 GAP 事件，在主动发起连接的动作结束和链接建立时，该 GAP 事件回调函数被底层代码执行，应用层可以在该 GAP 回调函数内在不同事件分支处理时，调用下一步的组件函数。

示例代码

```
void app_gap_evt_cb(gap_event_t *p_event)        //GAP 事件回调函数，p_event 是上传的事件内容指针。
{
    switch(p_event->type)                        //上传的事件的类型。
    {
        case GAP_EVT_MASTER_CONNECT:             //做为主机的链接建立完成事件
        {                                        // p_event->param.master_connect.conidx 是链接号，链接建立的链接号由底层分配,
                                                 //从 0 到 19。多链接情况下，分配的链接号都是不一样的。
            if (gap_security_get_bond_status())      //返回当前链接的对端设备是否是绑定过的设备
                gap_security_enc_req(p_event->param.master_connect.conidx);      //是绑定的设备，直接发起加密操作。
            else
                gap_security_pairing_req(p_event->param.master_connect.conidx);  //不是绑定的设备，发起绑定操作。
        }
        break;
        case GAP_EVT_DISCONNECT:        //链接断开事件。上传断开链接的链接号和断开链接的错误码
            co_printf("Link[%d] disconnect,reason:0x%02X\r\n",p_event->param.disconnect.conidx,p_event->param.disconnect.reason);
        break;
        case GAP_EVT_CONN_END:          //主动连接结束事件。上传参数返回连接动作的错误码。
            co_printf("conn_end,reason:0x%02x\r\n",p_event->param.conn_end_reason);
            break;
        case GAP_SEC_EVT_MASTER_ENCRYPT:  //链接加密动作结束，
            extern uint8_t client_id;              //client_id 是第 3 章初始化时创建 profile，底层分配的 profile  id 号。
            gatt_discovery_all_peer_svc(client_id,p_event->param.master_encrypt_conidx);         //对对端所有服务集合扫描
                      //如果只需要扫描对端设备的某一个服务集合，也可使用下面的方式，只扫面对端一个组 UUID 的服务。
            //uint8_t group_uuid[] = {0xb7, 0x5c, 0x49, 0xd2, 0x04, 0xa3, 0x40, 0x71, 0xa0, 0xb5, 0x35, 0x85, 0x3e, 0xb0, 0x83, 0x07};
            //gatt_discovery_peer_svc(client_id,event->param.master_encrypt_conidx,16,group_uuid);
        break;
    }
```

GAP 事件回调函数，底层在执行时，会输入包含事件内容的指针，事件内容包含事件的类型，由 p_event->type，GAP 支持的事件类型定义在”gap_api.h”结构体 gap_event_type_t 内。

按键响应函数里发起的主动连接动作，如果成功连接，底层会依次上传两个 GAP 事件，分别是 GAP_EVT_CONN_END 和

GAP_EVT_MASTER_CONNECT。 前者表示主动连接的动作结束了，用户可以按 PD7，进行第二个主动连接的动作。 后者表示主动连接产生的链接已经建立，并上传分配的链接号，对端的链接参数以及 mac 地址等信息。

多链接情况下，不同的链接由一个底层分配的链接号来代表，赋值从 0 到 19 之间，Fr801x H 的芯片最多支持 20 个链接。链接一旦建立，后续在该链接上进行的所有操作，均需要输入链接号，包括链接参数更新，MTU 交换，；链接对端服务扫描，对链接对端设备服务属性的操作，断开链接，等等操作。对这些操作的 API 调用时均需要链接号的参数。

所以应用层，应该记录这个底层分配并上传的链接号和对端设备的 mac 地址信息，保证后续能对不同的链接做不同的操作。

在链接建立事件 GAP_EVT_MASTER_CONNECT 之后，应用层调用了进行加密或绑定操作的函数，在加密或绑定动作完成时，如果正常的话，底层会上传 GAP 事件 GAP_SEC_EVT_MASTER_ENCRYPT，并附带加密完成的链接号，应用层在链接加密完成后，调用了扫描对端服务集合的函数。扫描服务的动作结束的标志是，client profile 的 GATT 事件回调函数，收到底层上传的 operation 为 GATT_OP_PEER_SVC_REGISTERED 的操作完成事件，将在下一节进行介绍。

6 Profile 数据收发

做为主机时，需要建立 client 的 profile 来对属性进行各种 GATT 的操作，profile 的创建过程在第 3 章入口函数初始化时有介绍，创建时指定了 GATT 的事件回调函数为 uint16_t simple_central_msg_handler(gatt_msg_t *p_msg)。

接收到对端发送的 notification/indication 数据，对端发送的 read response 数据时，底层都会上传对应事件到 GATT 事件回调函数。应用层根据回调函数上传的事件内容可以获取接收到的数据。

GATT 事件回调函数示例

```
uint16_t simple_central_msg_handler(gatt_msg_t *p_msg){      //GATT 事件回调函数，上传指向事件内容的指针
    switch(p_msg->msg_evt){                 //判断事件的类型，GATT 事件类型定义在"gatt_api.h"结构体 gatt_msg_evt_t 内
        case GATTC_MSG_NTF_REQ:{            //收到对端设备发送的 notification 数据
            if(p_msg->att_idx == 0){        //ntf 数据针对 UUID 数组序号 0 对应的属性
                show_reg(p_msg->param.msg.p_msg_data,p_msg->param.msg.msg_len,1);  //打印收到的 ntf 数据
            }
        }
        break;
        case GATTC_MSG_READ_IND:{           //收到对端设备发送的 read 操作的 response 数据
            if(p_msg->att_idx == 0){        //read response 数据针对 UUID 数组序号 0 对应的属性
                show_reg(p_msg->param.msg.p_msg_data,p_msg->param.msg.msg_len,1);  //打印收到的 read response 数据
            }
        }
        break;
  …接下页。
```

```
        case GATTC_MSG_CMP_EVT:{                        //GATT 某个动作完成。
            if(p_msg->param.op.operation == GATT_OP_PEER_SVC_REGISTERED){       //如果结束的动作为扫描对端服务集，则继续。
                uint16_t att_handles[2];
                memcpy(att_handles,p_msg->param.op.arg,4);      //上传感兴趣的 UUID 扫描到的 handler 号。
                show_reg((uint8_t *)att_handles,4,1);       //打印 UUID 对应的 handler 号，如果为 0 表示未扫描到。

                gatt_client_enable_ntf_t ntf_enable;        //下面使能对端 notification 功能
                ntf_enable.conidx = p_msg->conn_idx;        //GATT 事件所处的链接的链接号。
                ntf_enable.client_id = client_id;           //profile 由底层分配的 profile id 号
                ntf_enable.att_idx = 0; //TX                //本操作针对 UUID 数组序号 0  对应的属性
                gatt_client_enable_ntf(ntf_enable);         //发起使能对端 UUID 数组序号 0 对应属性的 Nottingham 功能。
            }
        }
        break;
}
```

在第 5 节介绍到，链接加密完成后，应用层发起了对该链接上对端设备的服务集合进行扫描的动作，该动作结束时，底层会上传扫描结束的事件到 Client profile 指定的 GATT 事件回调函数。

在扫描对端服务集合动作结束的处理分支下，底层上传的事件内容包含了扫描的结果，即感兴趣的 UUID 对应在对端服务集合中的 handler 号，应用层可以打印每个 UUID 对应的 handler 号，如果 handler 号为 0，表示这个 UUID 在对端服务集合中没有扫描到，后续对该 UUID 的任何 gatt 操作都要禁止，比如，使能 ntf，读，写操作等。

一般来讲，如果扫描到的 handler 号非 0，且在从机的服务集内，该 UUID 对应的属性包含 notification 的权限时，主机要立即使能从机该属性的 notification 功能。从机在接收到主机发来的 ntf_enable 消息后，才能对主机进行 notification 的操作。

底层在执行 GATT 事件回调函数时，都会上传该事件发生链接的链接号，由变量 p_msg->conn_idx 表示，应用层可以根据该链接号，对多链接情况下，不同的链接执行不同的操作。

做为主机，接收到从机的数据有两个途径，一是接收到对端发送的 notification/Indiacation 数据，二是主机对从机服务中的某个属性进行读操作，从机对读操作进行数据回复。 针对这两种情况，如果协议栈底层收到数据，都会上传 GATT 事件通知应用层。两种情况对应的 GATT 事件在上述示例代码中都有体现。

下面介绍主机如何对多个从机进行写和读操作。示例代码

```
        case GATTC_MSG_CMP_EVT:{                        //GATT 的某个动作完成。
            if(p_msg->param.op.operation == GATT_OP_PEER_SVC_REGISTERED){       //如果结束的动作为扫描对端服务集，则继续。
                gatt_client_write_t write;
                write.conidx = p_msg->conn_idx;             //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 = "\x1\x2\x3\x4\x5\x6\x7";     //进行写操作要发送的数据 buffer
                write.data_len = 7;                         //写操作的数据长度
                gatt_client_write_cmd(write);               //开始执行写操作
                gatt_client_read_t read;
                read.conidx = p_msg->conn_idx;              //GATT 事件所处的链接的链接号。
                read.client_id = client_id;                 //profile 由底层分配的 profile id 号
                read.att_idx = 0; //TX                      //本操作针对 UUID 数组序号 0 对应的属性
                gatt_client_read(read);                     //开始执行写操作
            }
        }
```

主机对从机的进行 GATT 的读和写操作，需要在扫描完对端的服务集合之后，并且要确保 UUID 对应的 handler 号非 0，才能对 UUID 对应的属性进行读和写的操作。

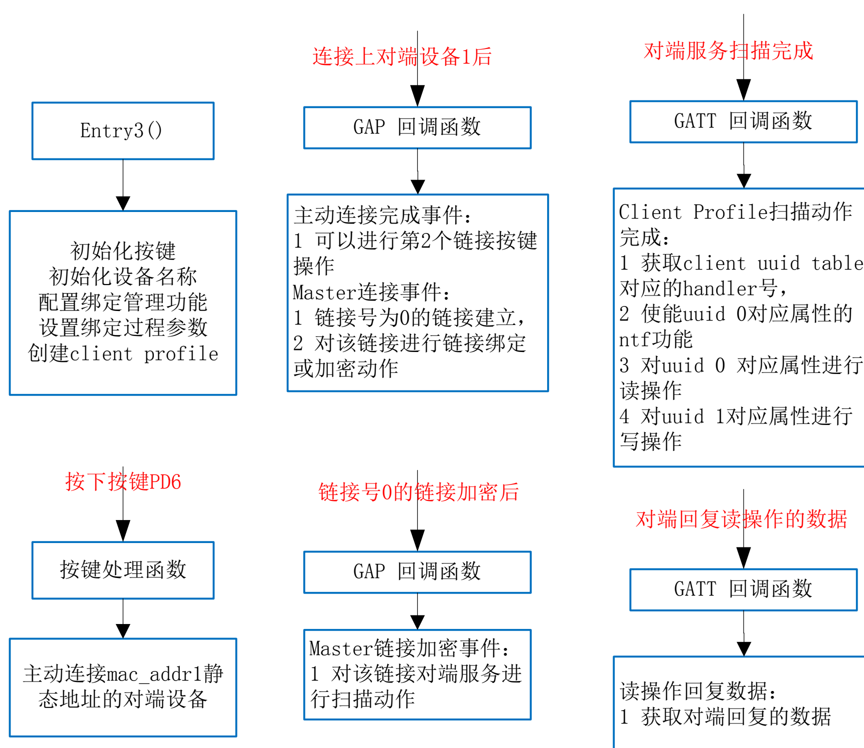
主机的进行 GATT 读写操作时，都需要输入链接号参数，表示对哪个链接的对端设备进行属性的读写。应用层根据链接号就能确保多链接下，对不同链接做不同的操作。

其中 GATT 读写操作，在完成时，底层都会上传 GATT 动作完成事件，应用层判断 p_msg->param.op.operation 来决定具体哪个事件执行完成，所有可能的 operation 定义在"gatt_api.h"内的*@defgroup GATT_OPERATION_NAME*。读操作完成后，如果对端从机回复读的数据，则底层会上传从机回复的 read response 数据的事件。

至此一个做为主机的链接从建立，绑定加密，扫描从机服务集合到针对 profile 定义的某个 UUID 属性的使能 ntf，读写操作都介绍完毕。下节小结一下整个项目的程序流程。

7 工程小结

示例工程的程序运行流程示例图如下



多从机工程项目软件执行流程图

应用程序在第 3 个入口函数初始化必要的组件后，等待用户按下 PD6，主动连接某个设备 1，底层执行 initiation 的动作，应用层等待底层连接成功和连接动作结束的事件上传。

如果链接建立成功，协议栈底层依次上传 GAP 事件：主动连接动作完成事件和 Master 链接建立事件。应用层在第一个事件之后，应设立标志位，允许用户按下 PD7，进行第二个主动连接的动作。在第二个链接事件之后，应用层调用绑定或加密函数，对刚建立的链接进行加密操作。

如果加密过程成功，协议栈底层会继续上传 GAP 事件：Master 链接加密事件，应用层在该事件后，执行对链接对端设备服务集合扫描的操作。

如果扫描结束，扫描操作结束的标志是 Client Profile 的 GATT 事件回调函数接收到 operation 为 GATT_OP_PEER_SVC_REGISTERED 的操

作完成事件。应用层此时需要确认扫描到的 handler 是否正确，正确的情况下，要使能对端含 ntf 权限的属性。

链接建立之后的程序流程基本结束。如果后续应用层需要对某个链接做读写操作，需要输入链接号操作，并调用"gatt_api.h"定义的相应函数进行。

构建多从机的工程时需要注意两点：

1. 链接建立时，底层就会上传该链接的链接号，后续的加密成功事件，扫描服务集合完成事件，读写操作完成事件均会上传链接号。
2. 主机在对不同的链接进行绑定加密操作，扫描服务集合，使能 ntf 功能，以及对某个链接的属性进行读写操作，均需要输入链接号参数。

以上两点，是做多从机工程的要点。

该示例工程涉及到的入口函数，回调函数，中断服务函数，软件定时器的程序流程的跳转，在文档《Fr801X H 如何构建系统》中均有详细的介绍。用户可以浏览该文档获得进一步理解。