**FR801xH 如何构建多从机工程**

Bluetooth Low Energy SOC

作者：董有才

www.freqchip.com



**目录**

[1. 综述 3](#_Toc35701034)

[2. 引用必要组件 3](#_Toc35701035)

[3. 入口函数 4](#_Toc35701036)

[4. 按键响应处理 5](#_Toc35701037)

[5. GAP事件回调函数 6](#_Toc35701038)

[6. Profile数据收发 7](#_Toc35701039)

[7. 工程小结 9](#_Toc35701040)

# 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

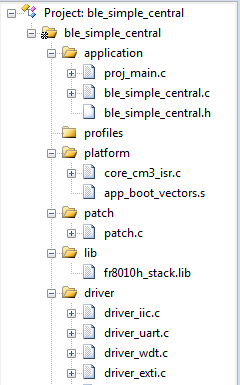
* 引用必要组件
* 入口函数初始化
* 按键响应处理
* GAP事件回调函数
* 链接加密绑定操作
* Profile数据接收与发送

下面几节将分别介绍每一步骤的详细过程。

# 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。

# 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。

# 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。

# 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。

# 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。

# 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。