FR801xH 如何构建系统

Bluetooth Low Energy SOC

www.freqchip.com

Contents

1 概述 3
2 项目创建 3
2.1 项目目录树结构 3
2.2 项目配置 4
2.2.1 Device 页面 5
2.2.2 Output 页面 5
2.2.3 User 页面 6
2.3 项目头文件路径和编译选项 6
2.4 链接设置 8
2.5 调试设置 9
3 用户入口函数 9
4 程序运行流程 11
4.1 概述 11
4.2 回调函数 12
4.3 Weak 函数入口 13
4.4 软件定时器 14
4.5 任务 15
4.6 硬件中断服务程序 17
5 错误处理 18
5.1 概述 18
5.2 错误码 19
5.3 可恢复错误处理 19
5.4 不可恢复错误 19
6 版本历史 21

1 概述

本文档旨在指导用户了解 801xH 软件 SDK 基本开发框架。一个 801xH 的项目可以看做是多个不同组件的集合。

801xH SDK 包含以下组件：

- BLE 5.0 协议栈和常见 Profile
- BLE SIG Mesh 协议栈
- 多个中间件组件
- 非抢占式操作系统
- 保持链接睡眠和关机睡眠调用接口
- 多种外设驱动
- 调试函数和错误处理
- 系统常用辅助函数

801xH SDK 的工程在选择某个组件时，需要包含该组件的头文件到应用程序，调用组件的接口函数。各组件在 SDK 软件包中的路径结构如下：

BLE 5.0 协议栈组件，components\ble\include\gap 和 components\ble\include\gatt

BLE SIG Mesh 协议栈组件，components\ble\include\mesh

常见 Profile 组件，components\ble\include\profile

中间件组件，components\modules

非抢占式操作系统，components\modules\os

外设驱动，components\driver

系统常用辅助函数和睡眠函数，components\modules\sys

以上组件中，BLE 5.0 协议栈组件，BLE SIG Mesh 协议栈组件和非抢占式操作系统的源代码存在于 lib 库和 rom code 中，使用这些组件时只需要引用组件的头文件即可，不需要引用源代码。

2 项目创建

2.1 项目目录树结构

一个示例项目的目录树结构可能如下：

```
- myProject/
        - components/ - ble/ - profiles/ - ble_simple_profile/ - src1.c
                          - library/ - fr8010h_stack.lib
                                     - syscall.txt
                          - driver/ - sr2c.c
                          - modules/ - platform/ - source/ - core_cm3_isr.c
                                                            - app_boot_vectors.s
                                     - patch/ - patch.c
                                     - button/ - src5.c
        - examples/ - none_evm/ -ble_simple_peripheral/ - code/ - proj_main.c
                                                                 - ble_simple_peripheral.c
                                                                 - user_task.c
                                                          - keil/ - ble_5_0.sct
                                                                 - ble_simple_peripheral.uvprojx
```

该示例项目 myProject 包含以下组成部分：

1. 可选的 component 目录中包含了项目所需的组件，

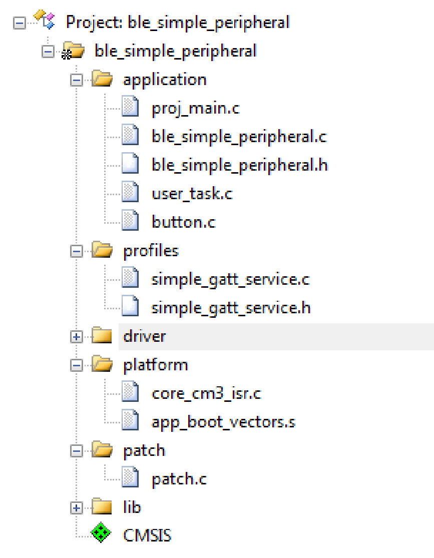
其中必须选取如下组件：

library 目录，该目录包含 ble5.0 协议栈的 rom code 函数地址表和协议栈底层 lib 库。

中间件 modules 目录中 platform 组件和 patch 组件，

2. Example/none_evm/ ble_simple_peripheral/code 目录，包含项目的源代码。
3. Example/none_evm/ ble_simple_peripheral/keil 目录，包含 keil 工程的链接脚本和工程启动文件。

接下来，需要创建 Keil 工程项目，项目创建后依次选中各组件的源代码加入到工程中，一切顺利的话，keil 项目目录树应如下显示



Keil 工程项目的示例目录结构

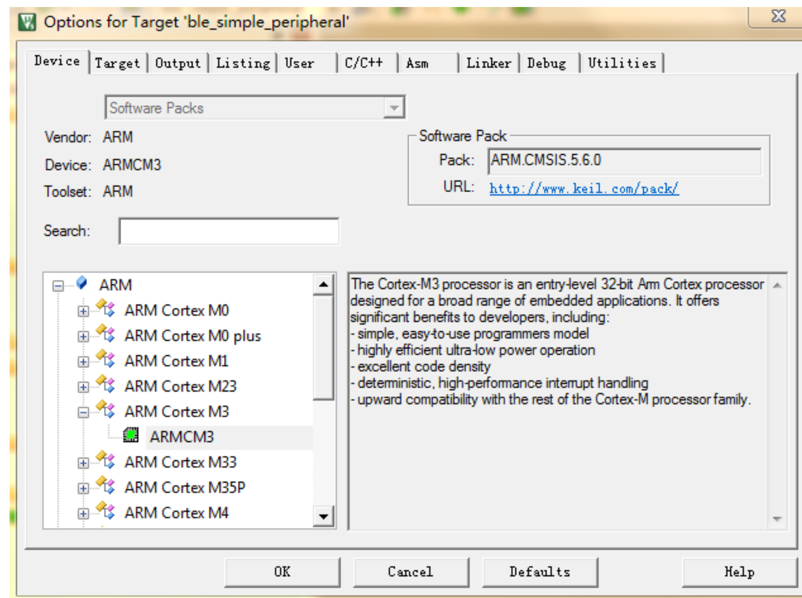
Keil 项目和所需的源文件和 lib 库都已选择完毕了，接下来几小节会介绍如何对项目进行必要的配置工作。

2.2 项目配置

依次点击 keil 工作界面最上面一栏菜单”Project”->”Options for Target”开始进行项目配置，

2.2.1 Device 页面

在 device 这一页面，选择项目针对的 CPU。Fr801xH 采用 Cotex-M3 的 CPU 内核，这里需要选择 ARMCM3 的 CPU 核心，如下图所示

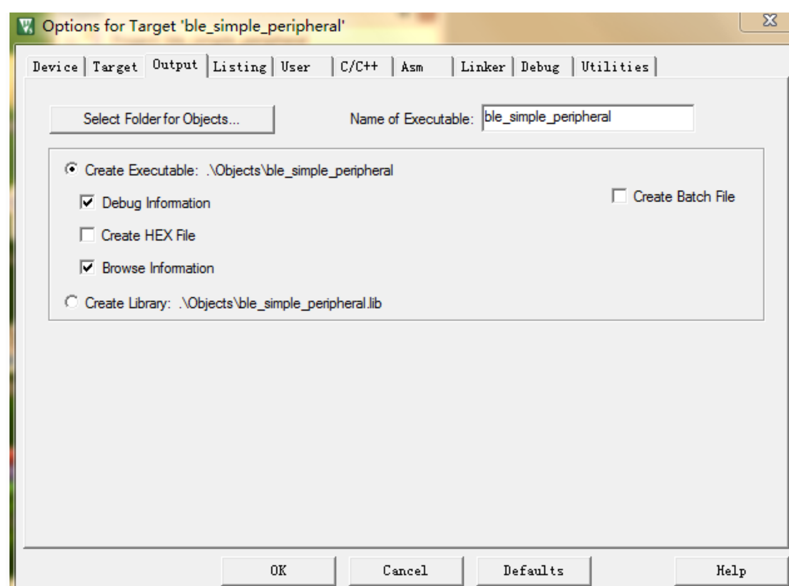


Keil 工程项目设备选择

注解
如果 Device 页面下拉框内找不到 ARM Cortex M3 的设备，则需要安装 keil 工具针对 Cotex-M3 核支持的软件包： https://www.keil.com/dd2/arm/armcm3/

2.2.2 Output 页面

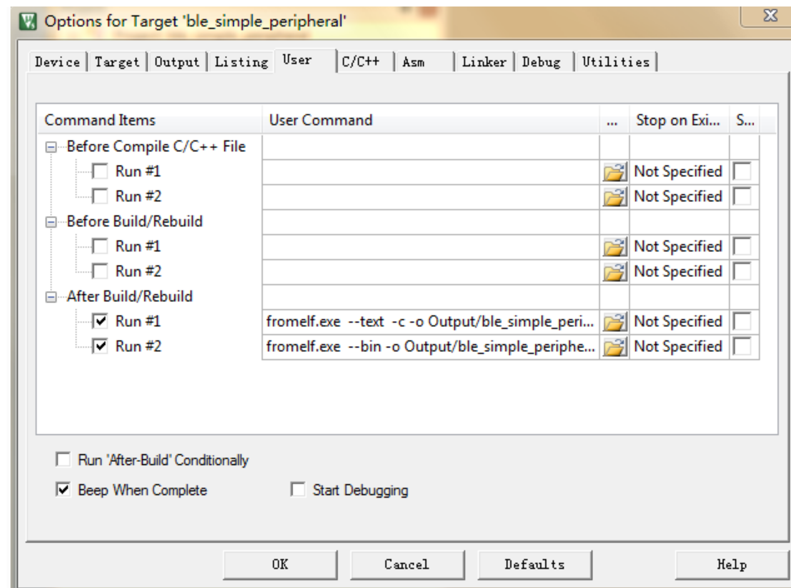
在 Output 页面，设置项目输出文件的名字和输出文件的类型，建议按如下设置



Keil 工程项目输出选择

2.2.3 User 页面

在 User 页面，设置链接完成后的动作，这里需要执行 2 个 window 脚本命令，将链接完的文件生成可执行的 bin 文件和产生项目的反汇编代码。



Keil 工程项目 User 页面配置

#1 运行命令语句产生反汇编文件，如下：

fromelf.exe --text -c -o Output/ble_simple_peripheral.txt Objects/ble_simple_peripheral.axf

#2 运行命令语句产生可执行 bin 文件，如下：

fromelf.exe --bin -o Output/ble_simple_peripheral.bin Objects/ble_simple_peripheral.axf

输出的反汇编和可执行文件的名字和路径通过更改语句中的路径实现。

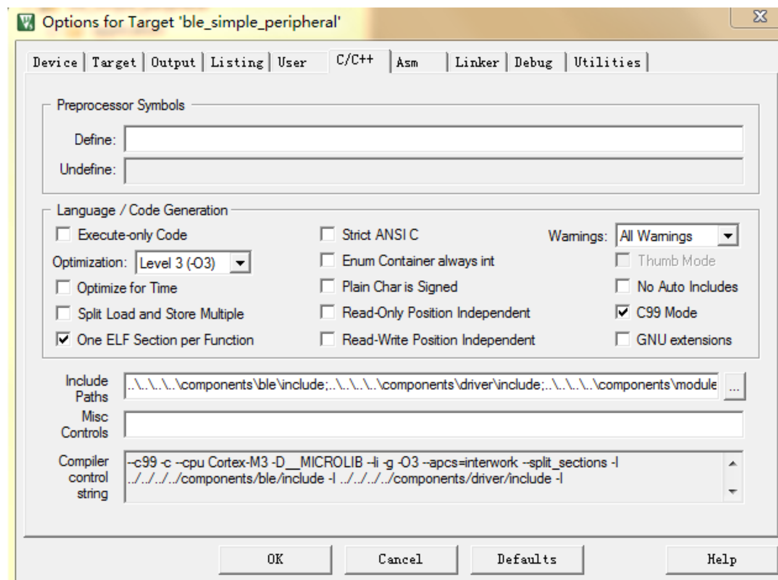
例如 fromelf.exe --bin -o ../bin/ble5.0.bin Objects/ble_simple_peripheral.axf

使用该命令后，会在\examples\none_evm\ble_simple_peripheral\BIN\目录产生可执行文件 ble5.0.bin

2.3 项目头文件路径和编译选项

应用工程所需用到的组件均通过引用组件的头文件来调用。这里介绍如何设置组件头文件的相对路径，依次点击 keil 工作界面最上面一栏菜单"Project"->"Options for Target"->"C/C++"进入 C 编译界面。

一般建议按下图设置编译选项



Keil 工程项目编译配置

编译配置分为三个部分：

- 预定义宏
- 编译选项
- 头文件路径

预定义宏

在 Preprocessor Symbols 栏，Define 后面可以填写多个宏定义，这些宏定义的影响范围是整个项目所有文件。

例如 MARCO1,MARCO2 表示工程定义了宏变量 MARCO1，MARCO2 这两个宏变量。

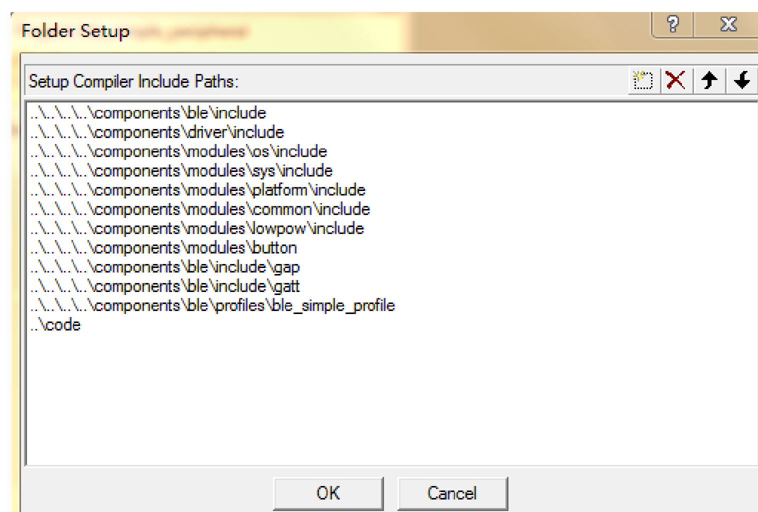
编译选项

在 Language/Code Generation 栏，建议按上图选择进行设置。常用的选项解释如下。

- Optimization，选择编译优化等级，一般来讲，优化等级越高，生成的代码空间越小。
- Optimize for Time，勾选表示编译以执行速度优先，编译的代码量会稍大。
- One ELF Section per Function，勾选该选项表示以单个函数做为优化单元，对冗余函数进行优化，能极大的节省生成代码的空间，未被调用的函数不会被放置到最后生成的 bin 文件和反汇编文件中。
- C99 Mode，勾选该选项表示支持最新的 C 语言标准 C99。

头文件路径

在 Include Paths 栏点击右边的(...)按钮进入项目头文件路径设置。应如下图所示



Keil 工程项目头文件引用路径设置举例

上图的示例中，示例工程一共引用了以下组件的头文件路径，分别为

1. BLE 5.0 协议栈组件，

    ..\..\..\..\components\ble\include\gap

    ..\..\..\..\components\ble\include\gatt

    ..\..\..\..\components\ble\include

2. Profile 组件，

    ..\..\..\..\components\ble\profiles\ble_simple_profile

3. 非抢占式操作系统，

    ..\..\..\..\components\modules\os\include

4. 外设驱动，

    ..\..\..\..\components\driver\include

5. 系统常用辅助函数和睡眠函数，

    ..\..\..\..\components\modules\sys\include

6. 中间件组件，

    必须包含的 platform 组件，

    ..\..\..\..\components\modules\platform\include

    ..\..\..\..\components\modules\common\include

    ..\..\..\..\components\modules\lowpow\include

    Button 组件，

    ..\..\..\..\components\modules\button

7. 项目自身代码组件，

    ..\code

上述的组件头文件路径均为相对路径。相对路径的编写规则如下，

- ✧ 以当前工程启动文件 ble_simple_peripheral.uvprojx（举例）为起点，符号..\表示上一级目录，符号.\表示下一级目录，均可连续使用多次
- ✧ 例如有如下目录结构

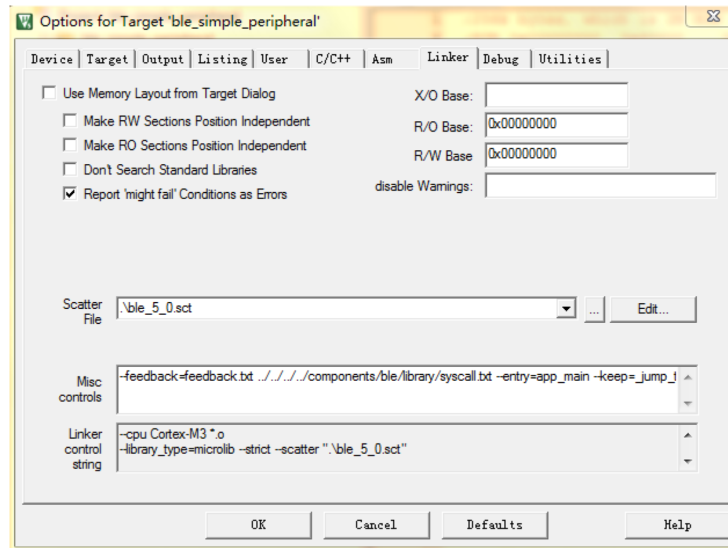
```
- Fr801xH SDK/
        - components/ - ble/ - profiles/ - ble_simple_profile/ - src1.c
        - examples/ - none_evm/ -ble_simple_peripheral/ - keil/ - ble_simple_peripheral.uvprojx
```

相对路径..\..\..\..\components\ble\profiles\ble_simple_profile 表示 ble_simple_profile 组件的头文件引用路径。

2.4 链接设置

应用工程编译成功后会进行链接操作对各个函数和变量赋予执行地址和分配大小。这里介绍如何设置链接脚本文件，依次点击 keil 工作界面最上面一栏菜单"Project"->"Options for Target"->"Linker"进入 C 编译界面。

一般建议按下图设置编译选项



Keil 工程项目链接设置

- ✧ Scatter File 栏点击后面...按钮，选择默认的 ble_5_0.sct 链接脚本。每一个示例工程均有提供链接脚本，并且基本相同，做应用工程时可以从 examples\none_evm 下任一示例工程 keil 目录下拷贝一份链接脚本到应用工程文件夹中，然后选中该链接脚本即可。
- ✧ Misc controls 栏，这里默认填如下字符串：

--feedback=feedback.txt ../../../../components/ble/library/syscall.txt --entry=app_main --keep=_jump_table --datacompressor=off

辅助控制字符串解释如下：

../../../../components/ble/library/syscall.txt，链接时会使用 rom code 里的函数地址进行链接。

注解
关于链接脚本.sct 文件的编写，可以参考文档《Fr801xH 如何编写链接脚本》

2.5 调试设置

Fr801xHSDK 的 keil 工程支持 J-link 在线调试功能，该功能的设置介绍，参考《Fr801xH 快速入门》章节：烧录到设备，小节：J-Link 工具在线烧录。

恭喜，您已完成 **Fr801xH 基于 Keil 开发工具的项目创建学习！**

接下来，将介绍如何开始编写应用程序。

3 用户入口函数

Fr801xH 软件开发框架下，应用程序开始于 3 个入口函数，入口函数本质是协议栈 lib 库内部定义的 weak 属性的函数，在 Fr801xH 程序启动时内部会依次进行调用。

应用层重新定义这些 weak 属性的函数，在编译时，编译器会将应用层定义的这 3 个同名，同类型的函数进行编译，而不会编译协议栈 lib 库内部定义的这 3 个 weak 属性的函数，那么 Fr801xH 程序启动时，内部代码就会依次运行应用层定义的这 3 个函数，此时，应用层的代码得到了执行。

使用 Fr801xH SDK 进行项目开发时，必须要定义了如下 3 个入口函数如下，入口函数一般需要定义在项目的 proj_main.c 里面。在

exampl\no_evm 文件夹下的所有示例工程内，均可在 proj_main.c 内看到入口函数的定义。

```
void user_custom_parameters(void)
void user_entry_before_ble_init(void)
void user_entry_after_ble_init(void)
```

下面分别介绍这 3 个入口函数。

- void user_custom_parameters(void)

该函数用于设置一些系统的重要参数，这些参数均保存于全局变量__jump_talbe 内。常见的设置项如下

```
#include "jump_table.h"
void user_custom_parameters(void)
{
    memcpy(__jump_table.addr.addr,"\x0F\x09\x07\x09\x17\x20",6); //设置设备的本地静态 mac 地址
    __jump_table.image_size = 0x24000;    //设置项目 bin 文件的最大 size。
    __jump_table.firmware_version = 0x00010000;    //设置项目的固件版本号。
    __jump_table.system_clk = SYSTEM_SYS_CLK_12M;    //设置 CPU 运行的频率
    __jump_table.lp_clk_calib_cnt = 50;    //设置内部 rtc 时钟  校准耗时时间。
}
```

该函数在 bootloader 执行完毕，初始化程序开始运行之前被调用。

- void user_entry_before_ble_init(void)

该入口函数一般用于配置外围设备，可以初始化外设驱动，但不可调用 ble5.0 协议栈和操作系统组件函数，常见的设置如下

```
void user_entry_before_ble_init(void)
{
    pmu_set_sys_power_mode(PMU_SYS_POW_BUCK); //设置芯片供电选择
    pmu_enable_irq(PMU_ISR_BIT_LVD);    //使能 pmu 的 LVD 中断
    NVIC_EnableIRQ(PMU_IRQn);    //使能 PMU 中断
    system_set_port_mux(GPIO_PORT_A, GPIO_BIT_2, PORTA2_FUNC_UART1_RXD);    //配置 PA2 做为 UART1 的 RX 脚
    system_set_port_mux(GPIO_PORT_A, GPIO_BIT_3, PORTA3_FUNC_UART1_TXD);    //配置 PA3 做为 UART1 的 TX 脚
    uart_init(UART1, BAUD_RATE_115200);    //初始化 UART1 做为 log 口输出
    ool_write(PMU_REG_ADKEY_ALDO_CTRL, ool_read(PMU_REG_ADKEY_ALDO_CTRL) & (~(1<<3)));    //设置 ALDO 电压
    system_set_pclk(SYSTEM_SYS_CLK_12M); //切换 CPU 运行频率到 12M
}
```

该函数在 lib 库初始化 ble5.0 协议栈之前被调用。

- void user_entry_after_ble_init(void)

该函数是可以调用所有组件函数，包括 ble5.0 协议栈和操作系统组件函数，常见的函数调用如下

```
#include "gap_api.h"
#include "gatt_api.h"
#include "simple_gatt_service.h"
// Gap 事件的接收处理回调函数
void app_gap_evt_cb(gap_event_t *p_event)
{
    switch(p_event->type)
    {
        case GAP_EVT_ALL_SVC_ADDED:   //在所有 profile 创建完毕后，协议栈会上传该消息。
            sp_start_adv();     // 调用 simple profile 组件函数开始广播
        break;
    }
}
void user_entry_after_ble_init (void)
{
    uint8_t local_name[] = "Simple Peripheral";
    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 事件的接收处理回调函数 app_gap_evt_cb。
    gap_bond_manager_init(0x32000,0x33000,8,true); //初始化 ble 协议栈的绑定管理功能
    sp_gatt_add_service();   //调用 simple profile 组件 service 创建函数，
}
```

以上是一个简单的创建 service 的 profile，并开始广播的示例代码。其中用到了回调函数的概念。回调函数的介绍将在下一节：程序运行流程中进行详细介绍。

注解
关于芯片上电后程序如何跳转到入口程序，可以参考文档《Fr801xH 应用程序启动流程》

4 程序运行流程

4.1 概述

在 Fr801xH SDK 框架下开发应用程序，开发者需要知道程序的运行流程，来更好的进行应用程序的开发。上一节介绍到芯片上电后底层代码启动，然后会依次调用应用层重定义的 3 个入口函数，分别是 void user_custom_parameters(void)，void user_entry_before_ble_init(void)，void user_entry_after_ble_init(void)。第 1 个入口函数不能调用任何组件函数，只能设置系统参数，第 2 个入口函数只能调用外设驱动组件函数，第 3 个入口函数最后运行，在第 3 个入口函数内，能调用所有组件的函数。

用户在调用各组件的函数时，会遇到程序流程的跳转，程序跳转指程序在一定条件下会跳转到特定的函数运行，比如： BLE 协议栈通知应用程序某个事件，软件定时器定时时间到，等。

Fr801xH SDK 的各组件函数，一共存在以下 5 种形式的程序流程跳转。

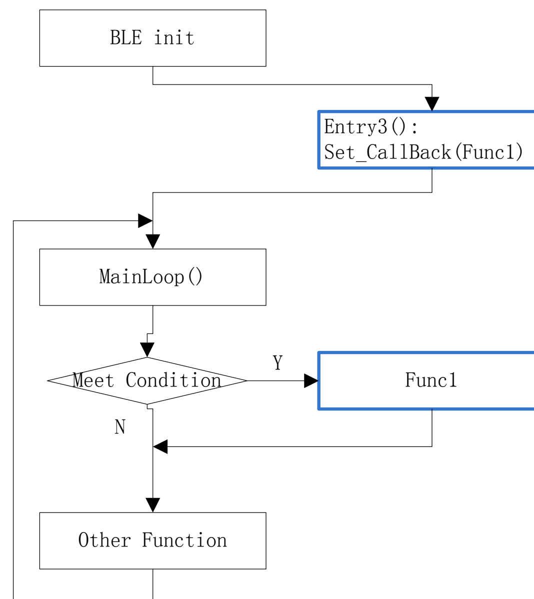
- 回调函数

- Weak 函数入口
- 软件定时器
- 任务
- 硬件中断服务程序

4.2 回调函数

回调函数指，应用程序将自定义的某个函数指针做为参数传递给 Fr801x H SDK 的回调设置函数，在特定的事件或条件发生时，由 Fr801x H SDK 底层调用该指针来执行应用程序自定一的某个函数，这个应用层自定义的函数就是回调函数。

回调函数的执行时刻示意图如下



回调函数设置与执行示意图

图中蓝色方框是应用层程序。

Fr801X H SDK 中 BLE 5.0 协议栈组件存在 2 个回调设置函数。下面是 GAP 事件的应用代码示例

```
#include "gap_api.h"
void user_entry_after_ble_init(void)
{
        gap_set_cb_func(app_gap_evt_cb);
        gatt_add_service(&simple_profile_svc);
        …
}
```

Gap_set_cb_func 将应用层定义的特定函数 app_gap_evt_cb 的函数指针做为参数传递给 Gap 事件回调设置函数，底层在接收到 Gap 特定事件时就会执行该特定函数 app_gap_evt_cb，达到通知应用层消息的目的。

设置完 Gap 特定事件回调函数后，示例代码继续执行创建 profile 的代码。然后等待协议栈底层执行 profile 创建过程，所有 profile 创建完毕后，底层协议栈会执行指定的应用层回调函数 app_gap_evt_cb。

回调函数的定义示例代码如下

```
#include "gap_api.h"
void app_gap_evt_cb(gap_event_t *p_event)
{
    switch(p_event->type)
    {
        case GAP_EVT_ALL_SVC_ADDED:
            sp_start_adv();
        break;
     }
}
```

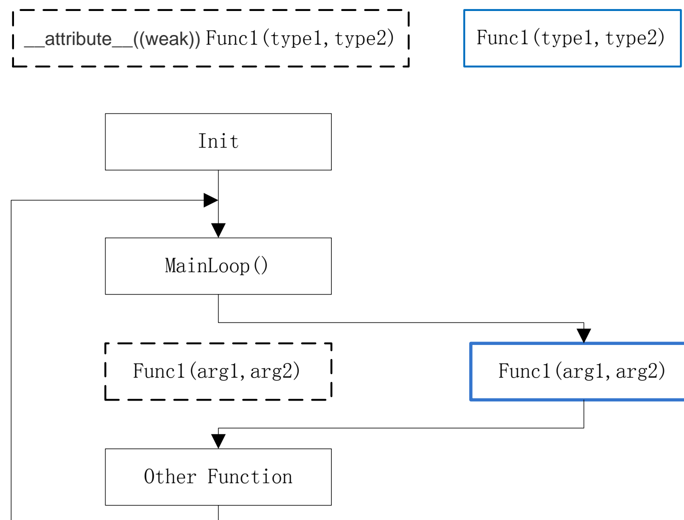
协议栈底层代码在所有 profile 创建完毕后会执行该回调函数，然后应用层在回调函数内在 profile 创建完毕事件处理分支之后继续执行其他的代码。这里示例代码是开启了 BLE 的广播。

Fr801X H SDK 中 BLE 5.0 协议组件中包含 Gatt 回调设置函数，原理与 Gap 回调设置函数相同，使用方式可以参见《Fr8010xH BLE 协议栈组件使用说明》

4.3 Weak 函数入口

Weak 函数指，组件已经定义了同类型，同名的函数，但是该函数使用__attribute__((weak))修饰符，并且在组件内部已经存在调用关系，如果应用层重新定义相同类型，相同名字的函数，则在编译器编译的时候，会选择编译应用层定义的同名函数，而不会编译协议栈定义的 weak 函数，那么在组件内部程序调用到原有的 weak 函数的地方时，程序会执行应用层定义的同类型，同名字的函数。

Weak 函数的调用示意图如下



Weak 函数设置与执行示意图

应用层定义了与组件内部相同名字，相同类型的函数体 Func1，在底层调用到 Func1 的函数时，就会直接执行应用层定的函数体 Func1，而不执行组件内部定义的函数体 Func1(用虚线标识)。

在 BLE 协议栈组件中，低功耗睡眠时会使用到 2 个 Weak 函数，分别是

- __attribute__((section("ram_code"))) __attribute__((weak)) void user_entry_before_sleep_imp(void)
- __attribute__((section("ram_code"))) __attribute__((weak)) void user_entry_after_sleep_imp(void)

第一个函数在协议栈进入睡眠之前会调用，第二个函数在协议栈睡眠唤醒之后会调用。调用的代码入下

```
case RWIP_DEEP_SLEEP:
{
        ool_write(PMU_REG_SYSTEM_STATUS, PMU_SYS_WK_MAGIC);
        user_entry_before_sleep_imp();
        low_power_save();       //    进入睡眠
        low_power_restore();    //    睡眠唤醒恢复
        user_entry_after_sleep_imp();
        rf_init(&rwip_rf);
        ool_write(PMU_REG_SYSTEM_STATUS, PMU_SYS_PO_MAGIC);
}
break:
```

如果应用层需要开启睡眠的功能，在应用层需要重新定义这两个函数，让 BLE 协议栈组件程序在睡眠前后能够执行应用层的代码。

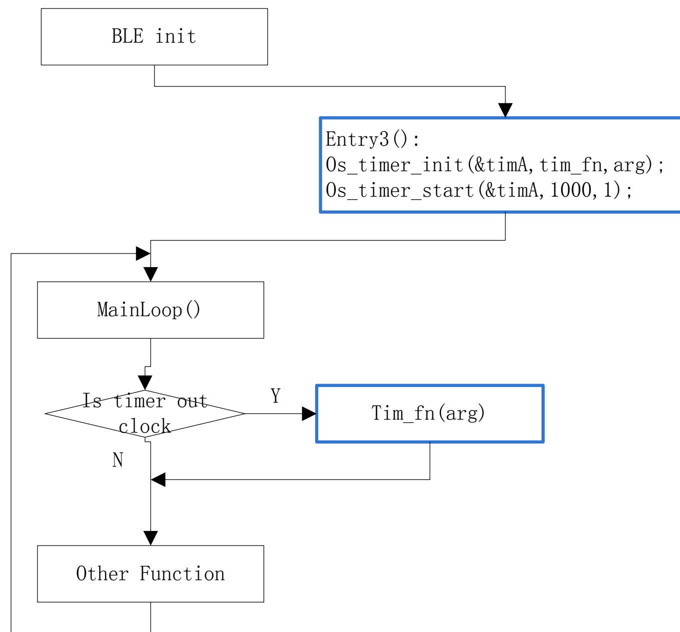
Exampe\no_evm 下的各示例程序的 proj_main.c 内均有这两个 weak 函数重定义的使用示例。
一般在 proj_main 重定义的函数代码如下

```
__attribute__((section("ram_code"))) void user_entry_before_sleep_imp(void)
{
}
__attribute__((section("ram_code"))) void user_entry_after_sleep_imp(void)
{
    system_set_port_mux(GPIO_PORT_A, GPIO_BIT_2, PORTA2_FUNC_UART1_RXD);
    system_set_port_mux(GPIO_PORT_A, GPIO_BIT_3, PORTA3_FUNC_UART1_TXD);
    uart_init(UART1, BAUD_RATE_115200);
    NVIC_EnableIRQ(UART1_IRQn);
    NVIC_EnableIRQ(PMU_IRQn);
}
```

在 void user_entry_after_sleep_imp(void)函数内重新初始化外设，因为睡眠时会关闭 CPU 和外设的电源，外设的寄存器值都会消失，所以在睡眠唤醒的函数内，需要重新初始化必要的外设，比如上面重新初始化了打印 log 的串口 UART1 到 PA2&PA3。

4.4 软件定时器

软件定时器指调用非抢占式操作系统组件的定时器函数来定义的定时器，从定时器启动开始时刻开始，到定时的时间到，系统会执行软件定时器指定的回调函数。
软件定时器在程序运行时的调用关系示意图如下



软件定时器函数设置与执行示意图

下面给出一个使用周期性 1 秒钟执行一次的软件定时器的代码示例，如下，

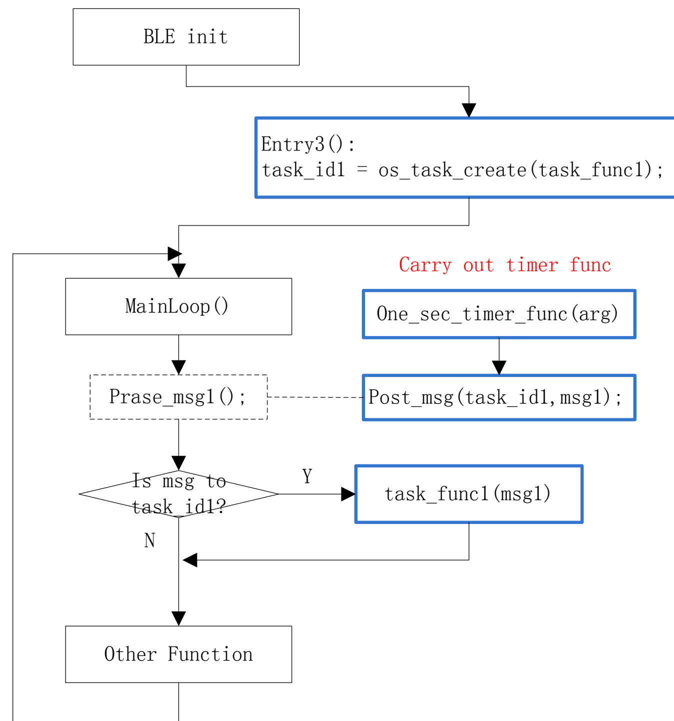
```
#include "os_timer.h"
os_timer_t timA;
void timA_fn(void *arg)
{
        co_printf("1s log\r\n");
}
void user_entry_after_ble_init (void)
{
      os_timer_init()&timA,timA_fn,NULL);
      os_timer_start(&timA,1000,1);
        ...
}
```

在定时器启动的时刻起，每隔 1 秒钟，系统会执行软件定时器 timA 定义的执行函数 void timA_fn(void *arg)。

4.5 任务

任务指的是应用层调用非抢占式操作系统组件的任务函数创建的一个能接收消息并能处理消息的回调函数，在指向该任务的消息被抛送之后很短的时间内，该任务回调函数会被系统执行，这样消息得到处理。消息属于操作系统组件的内容，可以在第 3 个入口函数之后的任意应用层执行代码内抛送，包括任务自身也能给自己抛送消息。

任务在系统中执行的流程示意图如下：



任务函数设置与执行示意图

上图的示意图中，应用程序在执行 1s 钟一次的软件定时器函数时，向早就创建好的任务 task1，抛送了一个消息 msg1。底层系统的调度主循环分析抛送出来的 msg1，发现该消息是指向任务 ID 为 task_id1 的，则会执行 task_id1 对应的任务函数 task_func1()。

下面给出一个使用软件定时器抛送消息给某个任务的代码示例，如下，

```
uint16_t taskA_id;
os_task_t taskA;
os_timer_t timA;
void taskA_fn(void *arg)
{
        Co_printf(“1s timer\r\n”);
}
void timA_fn(void *arg)
{
        os_event_t evt;
        os_msg_post(taskA_id,&evt);
}
void user_entry_after_ble_init (void)
{
        os_timer_init()&timA,timA_fn,NULL);
        os_timer_start(&timA,1000,1);
        taskA_id = os_task_craete(taskA_fn);
        …
}
```

在第 3 个入口函数处定义了一个 1s 钟执行一次的软件定时器 timA，然后定一个了任务 taskA。在软件定时器执行函数内向 taskA 抛送一个消

息 evt，那么 taskA 的执行函数 taskA_fn 会很快被到执行。上面例子中，taskA_fn 因为 1s 钟被抛送一次消息，所以会 1s 钟被系统执行一次。

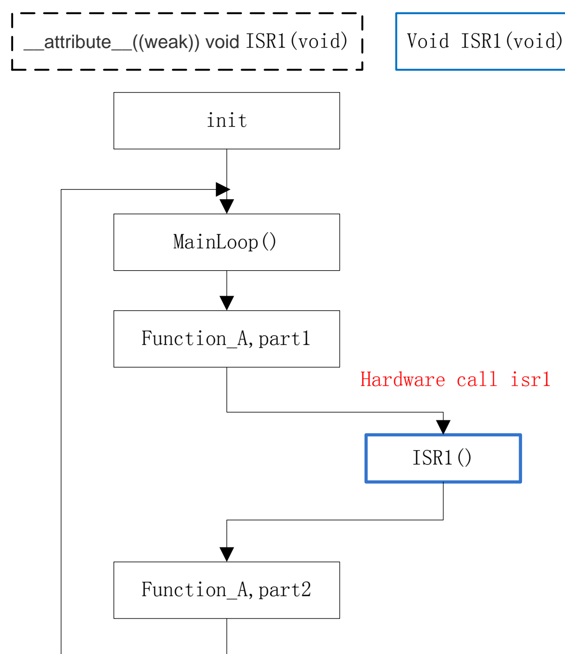
4.6 硬件中断服务程序

硬件中断服务程序指 801xH 芯片的硬件在需要通知软件执行时，会调用的函数。在硬件需要执行中断服务程序时，CPU 会中断正在运行的程序，保存当前程序执行的现场执行环境，然后跳转到硬件中断服务程序执行，执行完毕后，再恢复之前被保存的程序执行环境，重新跳转回原来程序运行。

Fr801xH 在 BLE 协议栈组件 和 外设驱动组件时，会涉及到硬件中断服务程序。其中 BLE 协议栈组件的硬件中断服务程序均在 lib 库内被执行，不需要应用层参与。外设驱动组件的硬件中断程序，需要用户定义。

一般来讲，Fr801X H SDK 内外设驱动组件的中断服务程序均有定义成一个 weak 函数。应用层需要重新定义一次同类型，同名字的函数，保证硬件调用中断服务程序时，能执行应用层定义的函数。

中断执行服务程序的流程示意图如下



中断服务函数设置与执行示意图

示意图中，应用层重定义了中断服务程序 (interrupt service routine)， void ISR1(void)。系统执行到 Function_A 时，硬件需要调用中断服务程序 ISR1， 则直接中断执行 Function_A，然后调用了应用层定义的中断服务程序 ISR1()之后，在重新执行 FunctionA 的剩余代码。

外设驱动组件中，分为数字模块的中断服务程序，和 pmu 模块的中断服务程序。前者有独立的中断向量表号码，后者公用一个中断向量表号码。中断向量表是系统内部定义的一个 32 长度 word 类型的数组，该数组定义在 app_boot_vectors.s 文件中，除数组 0 外，数组的每个序号的值都是中断服务程序的执行函数的地址值，用函数指针的值表示。除数组 0 外，每个序号代表一个固定的中断类型。

硬件在需要通知软件执行中断服务程序时，会查找中断类型对应的数组值，即中断服务函数的指针值，然后执行该函数。

下面给出一个，使用数字模块 UART(通用异步收发传输器)的中断服务程序设置与执行的代码示例。

```
__attribute__((section("ram_code"))) void uart0_isr_ram(void)
{
        ...
}
void user_entry_after_ble_init (void)
{
        system_set_port_mux(GPIO_PORT_D, GPIO_BIT_6, PORTD6_FUNC_UART0_RXD);
        system_set_port_mux(GPIO_PORT_D, GPIO_BIT_7, PORTD7_FUNC_UART0_TXD);
        uart_init(UART0, BAUD_RATE_115200);
        NVIC_EnableIRQ(UART0_IRQn);
        ...
}
```

向量中断表中的设置如下

```
import uart0_isr_ram
...
DCD     rwble_isr_patch         ; 0
DCD     timer0_isr_ram          ; 1
DCD     timer1_isr_ram          ; 2
DCD     uart0_isr_ram           ; 3
DCD     uart1_isr_ram           ; 4
```

系统在接收到来自于 PA2 脚发送来的 uart 的数据时，会进入 uart0 对应的中断服务程序 void uart0_isr_ram(void)。

另外，中断服务程序在软件设置了禁止中断的代码后，即使硬件通知软件执行，也不会立即执行中断服务程序，而要等到使能中断的代码运行后，才会执行。全局的中断函数禁止使能函数如下：

- void GLOBAL_INT_START(void)
- void GLOBAL_INT_STOP(void)

这两个函数定义在外设组件的头文件"driver_plf.h"中。

注解
关于外设驱动组件涉及到的详细中断服务函数，可以参考文档《Fr801xH 外设驱动组件介绍》

5 错误处理

5.1 概述

在应用程序开发中，及时发现并处理在运行时期的错误，对于保证应用程序的健壮性非常重要。常见的运行时的错误有如下几种：

- 可恢复的错误：

  通过函数的返回值(错误码)标识的错误

- 不可恢复的错误：

  断言失败（使用 assert 宏）造成的错误

CPU 异常，访问非法地址、非法指令等

系统级检查：看门狗超时、堆栈溢出等

本章将介绍 Fr801xH 中针对可恢复错误的处理机制，并提供不可恢复错误的产生原因，方便查找问题。

5.2 错误码

Fr801xH SDK 中调用 BLE 协议栈组件的 GAP 和 GATT 函数时，会返回 16bit 整形的错误码，这些错误码如果开启 BLE 协议栈组件的 lib 库 log 的话，能直观的看到执行某项具体操作后执行的错误码结果，log 采用"(hl code):0x2A"形式将错误码打印出来。错误码 0 表示成功。其他的错误码在 components\ble\inclue\ ble_hl_error.h 文件中已经定义好。

而其他组件通常返回值就是错误码。

完整的错误代码列表，请见《Fr8010xH 错误码参考》中查看。

5.3 可恢复错误处理

可恢复错误指程序返回错误码，但是不会中断程序执行，程序可以做一些恢复措施，继续运行。

1. 尝试恢复

示例：

```
uint16_t task_id;
task_id = os_task_create(func1);
if(task_id == TASK_ID_FAIL)
{
        os_task_delete(task_id_delete);  //删除不需要的 task，释放 task_id
        task_id   =   os_task_create(func1);
}
```

5.4 不可恢复错误

- 断言失败

系统出现一般断言失败错误时，会打印出错的代码所在的文件，和在文件中的行数，方便查找，打印完毕后，系统会一直执行死循环做为提醒，因为断言失败是不允许发生的，所以系统不会恢复。

示例：

```
[21:45:12.631]收←◆freqchip
[21:45:15.689]收←◆Software Version: 1.7.4
Hardware Version: 1.0
Compiler Date: Feb 19 2020
Compiler Time: 21:09:59

Firmware version is 1.0
Build date: Feb 17 2020 17:47:41
BLE Peripheral
Local BDADDR: 0xBDADD0F08010
..\modules\os\os_timer.c 142
```

找到 modules\os_timer.c 文件中的第 142 行代码为 TIMER_ASSERT(0); 该错误表示在未设置软件定时执行函数的情况下，启动了定时器，并

且运行时间到，系统找不到可执行的定时器函数，导致错误。

● CPU 异常

CPU 异常指的是硬件错误，系统会进入硬件错误的中断服务程序，Fr801X H 定义的硬件错误中断服务程序在中间件 modules 目录中 platform 组件的文件"core_cm3_isr.c"中。CPU 异常是严重的硬件错误，系统不会恢复，需用重启。

示例：

```
Crash, dump regs:
PC     = 0x200060A4
LR     = 0x000085B3
```

在该中断服务程序中，会打印进中断之前的 PC（program counter）程序计数指针和程序返回地址的值。通过查找反汇编文件，可以找到出错之前程序运行的语句。

以上面示例为例进行错误查找，

步骤 1，找到 PC 指针指向地址是 0x200060A4

步骤 2，在工程目录下找到 ble_5_0.txt，该文件是程序的反汇编文件，找到 0x200060A4 对应的汇编语句

```
i.user_proj_main
user_proj_main
    0x20006098:  a004      ..    ADR   r0,{pc}+0x14 ; 0x200060ac
    0x2000609a:  f7fcfe07  ....  BL    __0printf$8 ; 0x20002cac
    0x2000609e:  f04f4000  O..@  MOV   r0,#0x80000000
    0x200060a2:  6801      .h    LDR   r1,[r0,#0]
    0x200060a4:  a006      ..    ADR   r0,{pc}+0x1c ; 0x200060c0
    0x200060a6:  f7fcfe01  ....  BL    __0printf$8 ; 0x20002cac
    0x200060aa:  e7fe      ..    B     0x200060aa ; user_proj_main + 18
```

图 2 PC 指针对应的汇编语句

步骤 3，找到汇编对应的 c 语言代码是

```
56  void user_proj_main(void)
57  {
58      printf("user proj_main\r\n");
59    printf("crash:%x\r\n",*(uint32_t *)(0x80000000));
60      while(1);
```

0x80000000 地址取值出错了。

● 看门狗超时

应用层在使用了外设组件中的开门狗函数之后，如果程序因为某种原因，死循环，或者等硬件外设总线过长，导致开门狗超时时，会进入看门狗超时硬件中断服务程序。硬件看门狗超时时是不可恢复错误，需要重启运行。

示例：

```
wdt_rest:1

PC     = 0x2000609E
LR     = 0x000085B3
R0     = 0x00000010
R1     = 0x50058000
R2     = 0x00000060
R3     = 0x00008595
鹏reqchip2,0,0,144a8
```

在看门狗中断服务程序中，会打印进中断之前的 PC（program counter）程序计数指针和程序返回地址的值。通过查找反汇编文件，可以找到出错之前程序运行的语句。

以上面示例为例进行错误查找，

步骤 1，找到 PC 指针指向地址是 0x2000609E

步骤 2，在工程目录下找到 ble_5_0.txt，该文件是程序的反汇编文件，找到 0x2000609E 对应的汇编语句

```
user_proj_main
    0x20006098:    a001        ..      ADR      r0,{pc}+8 ; 0x200060a0
    0x2000609a:    f7fcfe07    ....    BL       __0printf$8 ; 0x20002cac
    0x2000609e:    e7fe        ..      B        0x2000609e ; user_proj_main + 6
$d
    0x200060a0:    72657375    user    DCD    1919251317
    0x200060a4:    6f727020    pro     DCD    1869770784
    0x200060a8:    616d5f6a    j_ma    DCD    1634557802
    0x200060ac:    0a0d6e69    in..    DCD    168652393
    0x200060b0:    00000000    ....    DCD    0
```

图 4 PC 指针对应的汇编语句

步骤 3，找到汇编对应的 c 语言代码是

```
56  void user_proj_main(void)
57  {
58      printf("user proj_main\r\n");
59    //printf("crash:%x\r\n",*(uint32_t *)(0x80000000));
60      while(1);
```

汇编语言在 0x20000609e 反复跳转。就是对应 C 语言这里的 while(1)

- 堆内存分配失败

如果出现分配某个内存过大时，系统突然重启，表示内存分配失败，底层软件不允许内存分配情况出现，重启了系统，在重启之前，不会打印任何信息，这区别于以上 3 中不可恢复的错误。

一旦出现系统突然重启的情况时，需要在程序运行时调用剩余堆栈大小查询函数 uint16_t os_get_free_heap_size(void)，确认可分配内存的最大值。

该函数定于在非抢占式操作系统组件的头文件”os_mem.h”中。

恭喜，您已完成 **Fr801xH** 项目软件程序调用组件时的流程控制！

接下来，将介绍分开介绍各个组件如何调用。

6 版本历史

Version	Date	Author	Description
0.1	2020-03-11	Dong youcai	Draft