艾克姆科技

nRF52840 开发指南-下册（从机）

[基于 Nordic 蓝牙低功耗/802.15.4 Soc-nRF52840]

艾克姆科技飞宇团队

[2018.12.20]

官方店铺：*https://acmemcu.taobao.com*
官方论坛：*http://930ebbs.com*

版权所有：艾克姆科技，引用请注明出处

本文档技术支持负责人：强光手电

[本文档以艾克姆科技 IK-52840DK 开发套件为硬件平台，针对 BLE 开发，从基本概念和新建 BLE 工程模板开始，一步步讲解蓝牙 BLE 程序框架及功能的实现]

修订历史记录

Revision Records

日期 Date	版本 Version	编制 Written By	审核 Checked By	说明 Explanation
2018.12.20	A	强光手电	彭震	初建(基于 SDK15.2 编写)
2020.1.2	B	强光手电	彭震	修改为 SDK16.0 版本

第一章：蓝牙 BLE 技术简介

1. 学习目的

1. 本章的目的是为了让读者对低功耗蓝牙技术有个大致的了解，尤其对于初次接触低功耗蓝牙的读者，应当先对 BLE 有一个总体的了解，熟悉 BLE 的一些常用概念，这些基本的概念对于后续的编程很重要。
2. 通过本章，读者应了解什么是单模和双模、BLE 和蓝牙 4.x 的区别以及蓝牙 5.0 的三大特点等概念。

2. 蓝牙技术的起源

- 蓝牙的创始人是瑞典爱立信公司。
- 爱立信早在 1994 年就已进行研发。
- 1997 年，爱立信与其他设备生产商联系，并激发了他们对该项技术的浓厚兴趣。
- 1998 年 2 月，5 个跨国大公司，包括爱立信、诺基亚、IBM、东芝及 Intel 组成了一个特殊兴趣小组（SIG），他们共同的目标是建立一个全球性的小范围无线通信技术，即现在的蓝牙。
- 蓝牙技术是以 10 世纪的一位丹麦国王命名的。

“蓝牙(Bluetooth)”一词来源于 10 世纪的丹麦国王 Harald Blåand（英文姓名为 Harold Bluetooth）。这位国王将四分五裂的局面统一起来的行为，与这种传输技术将各种设备无线连接起来，有相似的地方。为了纪念他，SIG 将自己的无线技术取名“蓝牙”。

2.1. 什么是蓝牙低功耗(BLE)

BLE 是蓝牙低功耗的简称（Bluetooh Low Energy）。蓝牙低功耗(BLE)技术是低成本、短距离、可互操作的鲁棒性无线技术，工作在免许可的 2.4GHz ISM 射频频段。它从一开始就设计为超低功耗(ULP)无线技术。它利用许多智能手段最大限度地降低功耗。

- 低成本，低功耗。
- 快速启动，瞬间连接。最快 3ms 低延迟。
- 传输距离的提高。
- 高安全性。使用 AES-128 加密算法进行数据报加密认证。

蓝牙低功耗在短距离无线通信领域占据着越来越重要的地位，在很多应用场合均可见到蓝牙低功耗的身影：

- 医疗和健康设备，如血糖仪、数字血压计、血气计、心率监视器、数字体温计等。
- 运动休闲，如心率监测仪、体温计、计步器、节拍器、测高仪等。
- 鼠标、键盘、遥控器。

- Beacons。
- 智能家居，如灯光控制、温度控制、湿度控制、安全锁控制、门窗控制的低功耗、高安全性解决方案。

2.2. 蓝牙版本及 4.0 和 BLE 的关系

蓝牙技术联盟 ： (Bluetooth Special Interest Group, SIG)，是蓝牙核心规格版本及蓝牙技术的管理者，本身不负责具体的蓝牙产品的设计和生产等。蓝牙技术的发展经历了下面几个主要的版本：

1. 蓝牙 1.0：基本码率，Basic Rate，BR。
2. 蓝牙 2.0：增强码率，Enhanced Data Rata，EDR。
3. 蓝牙 3.0：引入全新的交替射频技术(AMP: Alternate MAC/PHY)，允许蓝牙协议栈针对任一任务动态地选择正确射频，通过瞬间使用消费者设备中已存在的辅助无线电提供更快的吞吐量。
4. 蓝牙 4.0 是第一个综合性规范，其加入了全新的蓝牙低功耗技术，即蓝牙 4.0 集三种规格于一体，最重要的特点就是省电，可以使一粒纽扣电池连续工作数年。

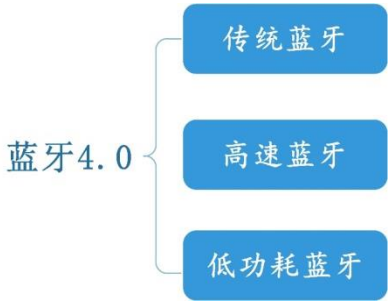


图 1-1：蓝牙 4.0 技术规范

蓝牙 4.0 包括三个子规范，即传统蓝牙技术、高速蓝牙和新的蓝牙低功耗技术。BLE 是蓝牙 4.0 的一部分。最常见的两种蓝牙核心规格：BR/EDR 和 BLE。

5. 蓝牙 4.2 提升了 BLE 数据传输速度，支持长包传输，单个数据包最大可传输 255 个字节。同时改善了隐私保护程度。
6. 蓝牙 5.0 相对于蓝牙 4.2，具有 2 倍的数据速率、4 倍的覆盖范围和 8 倍的广播能力。
   - 高速模式：支持 2Mbps 数据速率，实现了数据吞吐量翻倍。
   - 远距模式：通过 500kbps 和 125kbps 速率的全新编码物理层 (PHY) 实现远距离覆盖。
   - 广播扩展：蓝牙 BLE 有 3 个广播信道，之前的蓝牙 BLE 版本中广播载荷通过全部三个广播通道进行发送，蓝牙 5.0 中这三个通道仅用于发送指针，显示发送载荷的时间和地点，广播载荷仅在一个数据通道上传输一次。

2.3. 蓝牙的三种标志和设备类型

1. 蓝牙的三种标志(Logo)

为了更好地区分蓝牙的版本，SIG 发布了三种标志用于人们分辨这三个版本之间的区别和兼容性，

图 1-2：蓝牙的三种标志

- Bluetooth Smart Ready：适用于任何双模蓝牙 4.0 的电子产品，如苹果公司 iPhone 4S 及以后的产品和安装了安卓 4.3 以上的系统，并且蓝牙芯片支持 4.0 的手机。
- Bluetooth Smart：单模设备，如智能手环。
- 标准 Bluetooth：传统蓝牙设备，如用于传输语音的蓝牙耳机。

2. 设备类型

低功耗蓝牙分为单模（Bluetooth Smart）和双模（Bluetooth Smart Ready）两种设备：

- 单模设备：只支持蓝牙低功耗 (BLE)，如用 nRF51822 开发的蓝牙设备即为单模设备，单模设备对低功耗的要求很高。
- 双模设备：既支持传统蓝牙也支持蓝牙低功耗，这两种技术使用同一个射频前端和天线，双模设备一般都有足够的供电能力，对低功耗的要求不高。

图 1-3：典型的单模和双模设备

- 兼容性

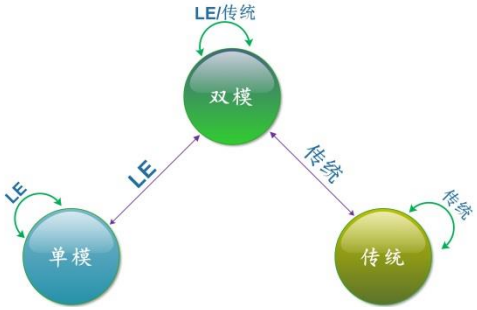


图 1-4：单模和双模设备的兼容性

双模设备之间：可以通过 LE 通信，也可以通过传统蓝牙通信；双模设备和单模设备通过 LE 通信；双模设备和传统蓝牙通过传统蓝牙通信。

- 单模设备之间：通过 LE 通信。
- 传统蓝牙：通过传统蓝牙通信。
- 单模设备和传统蓝牙：不能通信。

2.4. BLE 的体系结构

蓝牙低功耗包含三个部分：控制器、主机和应用程序。

图 1-5：BLE 体系结构

1. 物理层(Physical Layer)

- 调制方式：GFSK，高斯频移键控。
- 工作在 2.4GHz ISM 频段。频率范围：2.400-2.4835 GHz
- ISM 频段：(Industrial Scientific Medical Band)主要是开放给工业、科学和医用 3 个主要机构使用的频段。ISM 频段属于无许可（Free License）频段。2.4G ISM 频段是唯一一个在所有国家都无需授权的频段，只需要遵守一定的发射功率即可。我们熟知的无线局域网、蓝牙、ZigBee 等无线网络，均可工作在 2.4GHz ISM 频段上。
- 2.4G 被划分为 40 个 RF 信道(f=2402+k*2 MHz, k=0, … ,39)，信道间隔 2MHz，其中：
  广播信道：3 个，固定。
  数据信道：37 个，自适应跳频。
- 无线速率：1Mbps、2Mbps。

2. 链路层(LL)

链路层负责广播、扫描、建立和维护连接，以及确保数据包按照正确的方式组织、正确

地校验值和加密序列等。

1) 链路层状态机

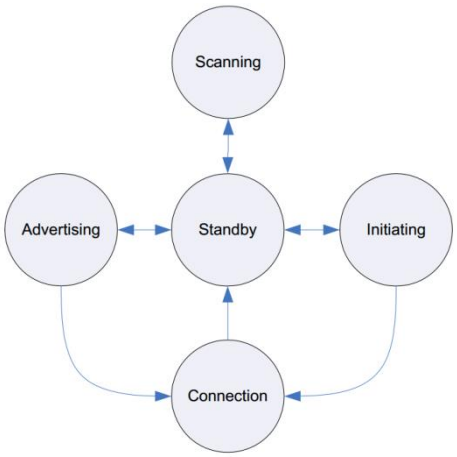


图 1-6：链路层状态机

链路层设备有 5 种状态：

- 就绪态：上电后，链路层进入并保持就绪态，直到接收到主机的命令。状态机的中心状态，处于其它状　态下都可以进入到此状态。
- 广播态：发送广播报文和扫描响应，
- 扫描态：侦听广播设备。
- 发起态：发起连接。
- 连接态：唯一一个用到数据信道的状态，两个设备只有在连接态中才能互相传送数据。

链路层可能同时存在多个链路，同时拥有多个独立的状态机，但是链路层永远不能同时成为主从机。

链路层状态机有助于我们清晰相关概念，如设备上电后，他不可能直接进入到连接态，必须由广播态或发起态才能进入到连接态。

✧ 需要特别注意的是：这里说的是链路层的状态机，而不是整个系统的。

2) 链路层信道映射

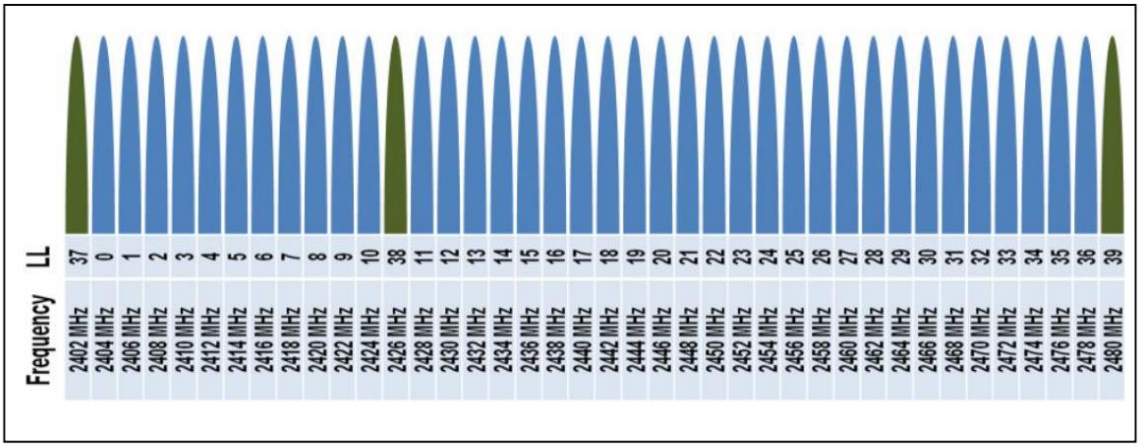


图 1-7： 链路层信道映射

广播信道：37 、38、39，对应的中心频率是 2402MHz，2426MHz，2480MHz。广播信道之间至少相差 24MHz。每次广播，都会在 3 个信道上将广播数据发送一次，这能有效地避免干扰，即使一个信道存在干扰，另外的信道也可以很好地工作，而三个信道同时被干扰的情况极少。

✧ 为什么广播信道是 3 个，而不是更多？

广播信道越多，各个信道同时受到干扰的几率越小，抗干扰性越强。但是广播信道越多，发射数据占用的时间就越长，功耗也就越高。所以，在综合考虑抗干扰性和功耗的情况下，SIG 将广播设定为 3 个。

3. 主机控制器接口(HCI)

主机控制器接口 HCI(Host controller interface)在 Host 和 Controller 之间提供一个标准化的接口，该层可以由应用程序接口 API 实现或者使用硬件接口 UART、SPI 或 USB 来控制。

控制器通过 HCI 发送数据和事件给主机，主机通过 HCI 发送命令和数据给控制器。

4. 逻辑链路控制和适配协议(L2CAP)

- L2CAP 层向上层提供数据封装服务，从而使逻辑上允许端到端的数据通信。
- L2CAP 提供可复用，分割和重组的面向连接或者无连接的数据服务。L2CAP 数据包可以达到 64K。L2CAP 允许每个 L2CAP 通道流控和重传。

5. 安全管理层(SM)

安全管理层 SM(Security Manager)提供配对和秘钥的分发。

6. 属性协议(ATT)

属性协议(ATT)规定了怎样去访问对端设备的数据，数据存储在属性服务器的“属性”里，供属性客户端进行读写操作。

7. 通用属性配置文件(GATT)

通用属性配置文件(GATT)建立在属性协议的基础上，GATT 通过使用 ATT 协议层定义了如何发现与使用服务、特征和描述符的标准方法。

■ 客户端与服务端架构：GATT 定义了两个角色：服务器和客户端。

- 服务器：客户端提供数据服务，就是数据中心。
- 客户端：从服务器读写应用数据，就是访问数据者。

✧ 特别注意：下面三个概念是完全独立的，他们针对是不同的层。如主机既可以是 GATT 客户端，也可以是 GATT 服务器。而从机也一样，既可以是 GATT 客户端，也可以是 GATT 服务器。

- 服务器：客户端提供数据服务，就是数据中心。客户端：从服务器读写应用数据，就是访问数据者。
- 主机、从机：针对链接层。

- 外围设备、中央设备：GAP 定义的角色。

8. 通用访问配置文件(GAP)

GAP 层负责处理设备的接入方式和过程，包括设备发现，链路建立，链路终止以及实现绑定。

9. 应用层

应用层定义了三种种类型：特征（characteristic）、服务（service）和配置文件（profile）。

- 特征：采用已知格式、以通用唯一识别码（UUID）作为表示的一个小块数据，由于特征要求能够重复使用，因而设计时没有涉及行为，特征被定义为计算机的可读格式，当计算机遇到一个从未接触过的特征时，计算机可以去更新这个特征的读取规则，从而正确读取特征，并向用户展示读取的特征。
- 服务：服务是人类可读的一组特征及其相关的行为规范，只定义了位于服务器上的相关特性和行为，而不定义客户端的行为。服务有两种类型，首要服务和次要服务，一个服务是首要服务还是次要服务取决于服务的定义，首要服务表征一个给定的设备主要做些什么。正是通过这些服务，用户才了解到该设备是做什么的。次要服务是那些协助主要业务或其他次要服务的服务。
- 配置文件：是用例或应用的最终体现。配置文件是描述一个或多个设备的说明，每一个设备提供一个或多个服务，配置文件描述了如何发现和链接设备，从而为每台设备确定了拓展结构，配置文件还描述的客户端的行为，用于发现服务和服务特性。

2.5. BLE 广播、扫描和连接事件

2.5.1. 广播事件

- 通用广播：最常用的广播方式，可以被扫描，接收到连接请求时可以作为从设备进入一个连接。
- 定向广播：针对于快速建立连接的需求，定向广播会占满整个广播信道，数据净荷只包含广播者和发起者地址，发起者收到发给自己的定向广播后，会立即发送连接请求。
- 不可连接广播：广播数据，而不进入连接态。
- 可发现广播：不可连接，但可以响应扫描。

BLE 广播间隔：是指两次广播事件之间的最小时间间隔，一般取值范围在 20ms-10.24S 之间，链路层会在每次广播时间期间产生一个随机广播延时时间（0ms-10ms）。

2.5.2. 扫描事件

每次扫描，设备打开接收器去监听广播设备，这称为一个扫描事件，扫描事件有两个时间参数扫描窗口和扫描间隔：

- 扫描窗口(scan window)：一次扫描进行的时间宽度。

- 扫描间隔(scan interval)：两个连续的扫描窗口的起始时间之间的时间差，包括扫描休息的时间和扫描进行的时间。

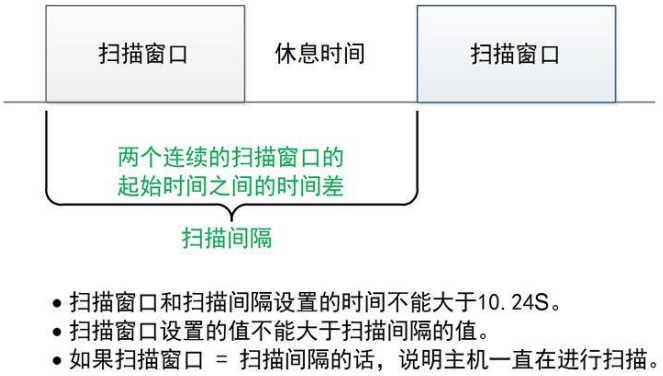


图 1-8：扫描间隔和扫描窗口

2.5.3. 连接事件

- 一个连接事件是指主设备和从设备之间相互发送数据包的过程。
- 所有的数据交换都是通过连接事件来完成。
- 每个事件发生在某个数据通道（0~36）。
- 一个连接中，主从设备依靠连接事件交换数据。
- 设备连接后，无论有无数据收发，连接事件都在按照设置的连接参数周而复始的进行着，直到一方停止响应。
- 主机与从机可在单次连接事件进行多次数据传输。

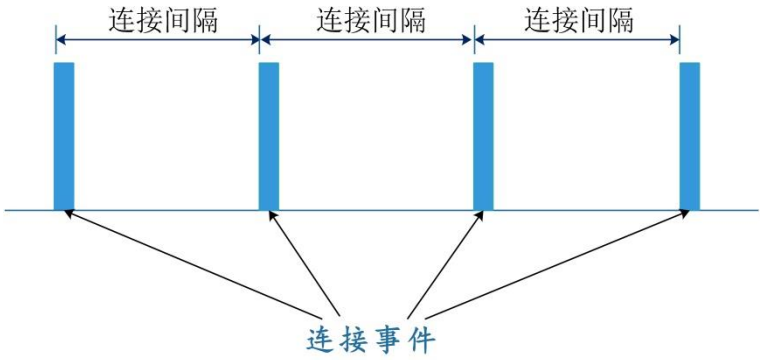


图 1-9：连接事件

1) 连接参数

- 连接间隔：必须是 1.25ms 的倍数，范围是从最小值 6（7.5ms）到最大值 3200（4.0s）。间隔参数包括两个 16 位的值，第一个为最小连接间隔，第二个为最大连接间隔。
- 从机延迟：这个参数描述了从机跳过连接事件的次数。这使外围设备具有一定的灵活性，如果它不具有任何数据传送，它可以选择跳过连接事件，并保持睡眠，从而提供了一些积蓄力量。这一决定取决于外围设备。
- 监督超时：这是两个成功的连接事件之间间隔的最大值。如果超过这个时间还未出现成

功的连接事件，那么设备将会考虑失去连接，返回一个未连接状态。这个参数值使用10ms 的步进(10ms 的倍数)。监督超时时间从最小 10（100ms）到最大 3200（32.0s）。同时超时时间必须大于有效连接事件。

2) 连接参数和功耗、传输速度的关系

不同的应用也许需要不同的连接间隔，一个长时间的连接间隔将会节约更多的能量，因为设备可以在两个连接事件之间睡眠更长的时间。但是他会导致数据发送不及时，如果有数据要发送那么他只能够在下一次连接事件到来时才能被发送。

表 1-1：连接参数和功耗、传输速度的关系

短连接间隔	高功耗，高数据吞吐量，发送等待时间短。
长连接间隔	低功耗，低数据吞吐量，发送等待时间长。
低或者 0 潜伏值	从机在没有数据发送的情况下高功耗，从机可以快速的收到主机的数据。
高潜伏值	从机在没有数据发送的情况下可以低功耗；从机无法及时收到主机的数据，但主机能及时收到从机的数据。

2.6. Profile、Service 、Characteristic 和 UUID

1. profile（配置文件）

profile 是 Service 的集合，它是预定义的，并不是实际存在于设备中，所以在外围设备的代码中，我们看不到 profile 的实现代码。Profile 分为标准的和自定义的，标准的 profile 是由 SIG 定义和发布的，如心率 profile、防丢 profile，自定义的 profile 是由开发者自己定义，如 SDK 中的蓝牙串口透传的 profile。标准的 profile 可以在 SIG 的网站上查到，网址如下：

https://www.bluetooth.com/specifications/gatt

一个 profile 文件可以包含一个或者多个服务，如心率 profile 包含了 Heart Rate Service（心率服务）和 Device Information Service（设备信息服务），从 SIG 网站上下载心率 profile，打开后可以看到心率传感器服务需求表格，由表中可以看到心率 profile 规定必须包含 Heart Rate Service 和 Device Information Service。

Service	Heart Rate Sensor
Heart Rate Service	M
Device Information Service	M

M：强制要求的。

2. Service（服务）

服务是一组特征和它们公开的行为的集合，一个服务可以包含一个或多个特性。Service 也分为标准的和自定义的，标准的 Service 是由 SIG 定义和发布的，如心率 Service、防丢 Service，自定义的 Service 是由开发者自己定义，如 SDK 中的蓝牙串口透传的 Service。标准的 Service 可以在 SIG 的网站上查到（网址同 profile）。

3. Characteristic（特征）

特征是具有特定意义的数值，如心率、温度值等等。BLE 主从机之间的数据传输实际传输的就是特征值。

4. UUID

UUID(Universally Unique Identifier)是一个 128 位的数字，用来标志属性的类型。Service 和 Characteristic 都是一种属性，都需要一个唯一的 UUID 来标识。

既然 UUID 是 128 位的数字，那么为什么有 16 位的 UUID？

因为 128 位的 UUID 相当长，设备间为了识别数据的类型需要发送长达 16 字节的数据。为了提高传输效率，蓝牙技术联盟(SIG)定义了一个“UUID 基数”，结合一个较短的 16 位数使用。二者仍然遵循通用唯一识别码的分配规则，只不过在设备间传输常用的 UUID 时，只发送较短的 16 位版本，接收方收到后补上蓝牙 UUID 基数即可。

第二章：新建 BLE 工程模板

1. 学习目的

蓝牙 BLE 工程模板是学习 NORDIC 蓝牙 BLE 程序开发的切入点，BLE 工程模板完成了 Nordic BLE 程序开发的基本框架，功能类似于一个 BLE 的“最小系统”。通过本章的新建 BLE 工程模板，我们需要了解和掌握以下内容：

1. 了解 BLE 工程的建立过程和配置项目。
2. 了解对于一个基本的 BLE 工程，需要加入哪些文件以及这些文件在工程中的分组。
3. 掌握利用 MDK 新建协议栈“Target”的方法，建立协议栈“Target”后，即可使用 MDK 下载协议栈 HEX 文件。（该方式不仅可以下载协议栈 HEX 文件，如需要使用 MDK 下载其他 HEX 文件均可参考此方法）。

2. 新建 BLE 工程模板

2.1. 规划工程目录

和裸机新建工程一样，在建立工程之前，我们需要先考虑一下工程文件的组织，也就是工程文件在计算机中的存放目录。清晰的工程目录既方便我们管理工程中的各个文件，也方便日后的维护和移植，我们可以根据自己的习惯和喜好来建立自己的工程目录，但是也不要太随意，文件目录应该一目了然，目录中各个文件夹的名字要能准确地指示里面的内容。

下面是我们建立工程时使用的工程目录，供大家参考，其中：

- App 文件夹：用于存放 main.c 文件和我们自己编写的应用程序文件。
- Project 文件夹：
  - Config 文件夹：用于存放工程配置向导（sdk_config.h）。
  - mdk5 文件夹：用于存放工程文件。
- doc 文件夹：用于存放说明之类的文档。
- components、integration、modules 等：从 SDK 中拷贝的库文件。

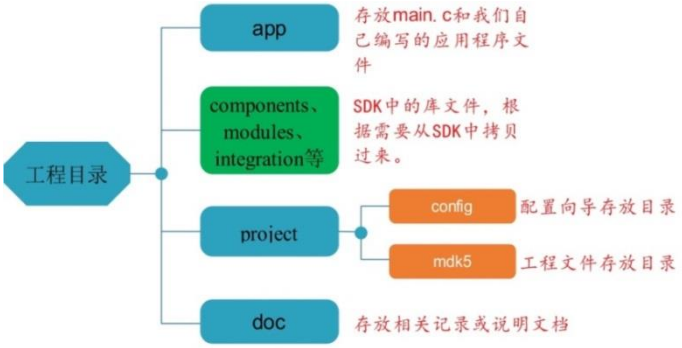


图 2-1：工程存放目录

2.2. 建立工程存储目录、拷贝库文件

按照上文中描述的工程目录新建用于存放工程各个模块的文件夹。先在 D 盘新建一个名字为 ble_app_templete 的文件夹，然后在这个文件夹下面新建 3 个名字分别为 project、doc 和 app 的文件夹，之后在 project 文件夹里面再新建 config 和 mdk5 两个文件夹，其中 config 文件夹用于存放工程配置向导（sdk_config.h），mdk5 文件夹用于存放工程文件。

之后，解压 SDK16.0，并将需要的库文件（包含 components、integration、modules 和 external 文件）拷贝到 BLE 工程模板目录下，如下图所示。

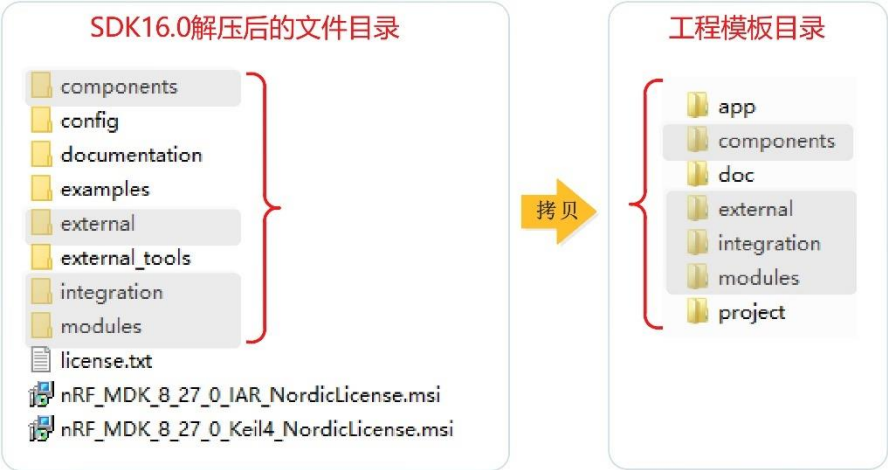


图 2-2：拷贝库文件

库文件可以根据需要去拷贝，也可以全部拷贝过来，全部拷贝占用的空间会大一些，但是用起来方便，以后添加功能时不需要再次去拷贝，故建议直接拷贝 SDK 中的 components、integration、moudles 和 external 文件夹到工程目录下。

2.3. 新建工程

建立工程存放文件夹和拷贝库文件之后，我们就可以开始建立工程了。

工程名取为：ble_app_templete_s140（这样取名是为了符合 Nordic 的 SDK 命名规则，其中 ble 表示这是一个 ble 的工程，app 表示应用程序，templete 表示模板，s140 表示使用的协议栈是 nRF52840 的协议栈 s140），工程存放到 D 盘 ble_app_templete 文件夹。

1. 启动 MDK，点击【Project】，在弹出的下拉菜单中选择【New uVision Project】。

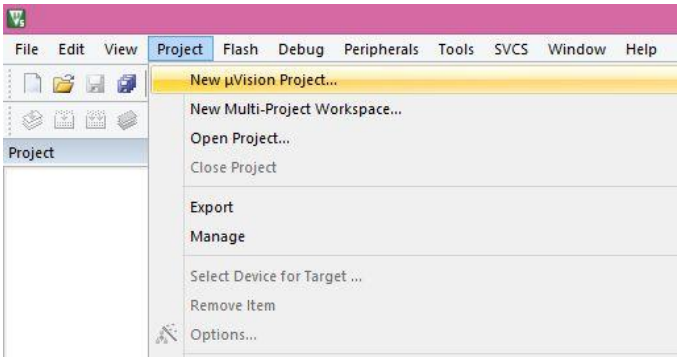


图 2-3：新建工程窗口

2. 设置工程名和工程保存路径，设置完成后点击【保存】。

✧ 工程路径和工程名设置注意事项：工程路径和工程名不能包含汉字字符(虽然有些计算机使用汉字字符没有问题，但是还是建议不要使用汉字字符，因为 MDK 对汉字字符的支持比较差)，同时路径不要过深，否则打开工程或仿真时可能会出现问题。

图 2-4：设置工程路径和工程名

保存后，工程名称是：ble_app_templete_s140，工程保存路径是："…\ble_app_templete\project\mdk5\ble_app_templete_s140"。

3. 保存工程后，会弹出器件选择窗口，选择好器件后点击【确定】。

开发板上使用的 nRF52840 型号是：nRF52840-QIAA，所以，在下图的器件列表中需要选择这个型号。选中器件后，右边的文本框中会显示该器件的描述信息。

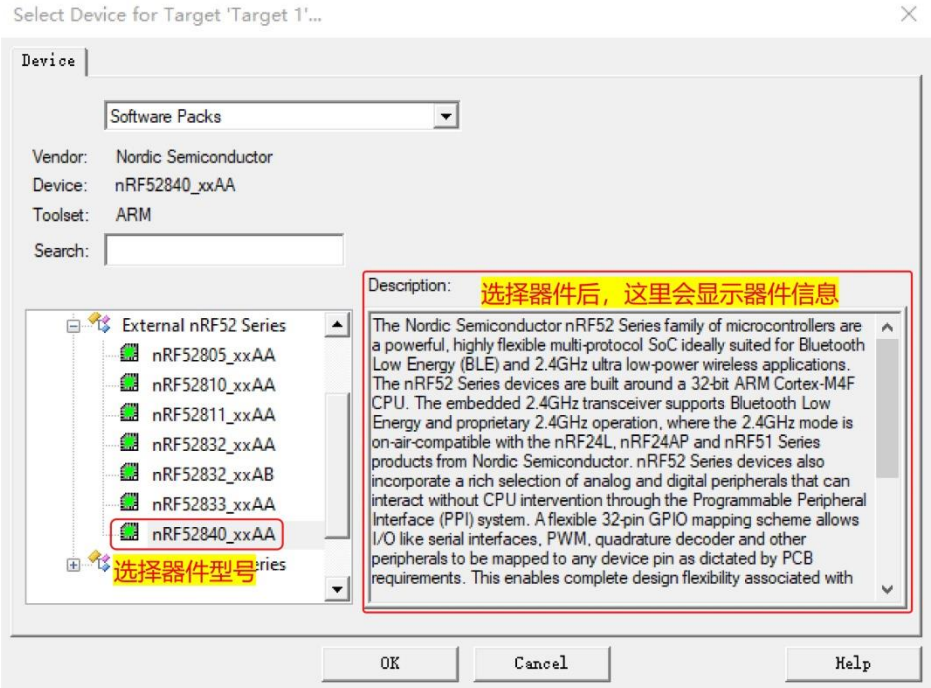


图 2-5：选择器件

4. 配置 RTE(Run-Time Environment)，选择完成后点击【OK】。

勾选两个**必选项**：CMSIS 中的 CORE 和 DEVICE 中的 StartUp。注意他们的版本号，SDK 版本不一样，对应的 CMSIS 和 StartUp 版本号可能会不一样，本文使用的 SDK 版本是 SDK16.0，SDK16.0 对应的 CORE 版本是 4.5.0，StartUp 版本是 8.27.1。

下图中，我们可以看到 CORE 的版本只能选择 5.2.0，StartUp 版本只能选择 8.27.1，这是因为安装 MDK5.27 时会自动安装 5.2.0 的 CORE，而 MDK 新建工程时只会显示已安装的最新的 CORE 和 StartUp 版本，所以，这里 CORE 的版本选择 5.2.0，StartUp 版本选择 8.27.1。工程建立好了之后，我们可以在工程配置中修改，以保持和 SDK16.0 一致，后面在工程配置章节会说明修改方法。

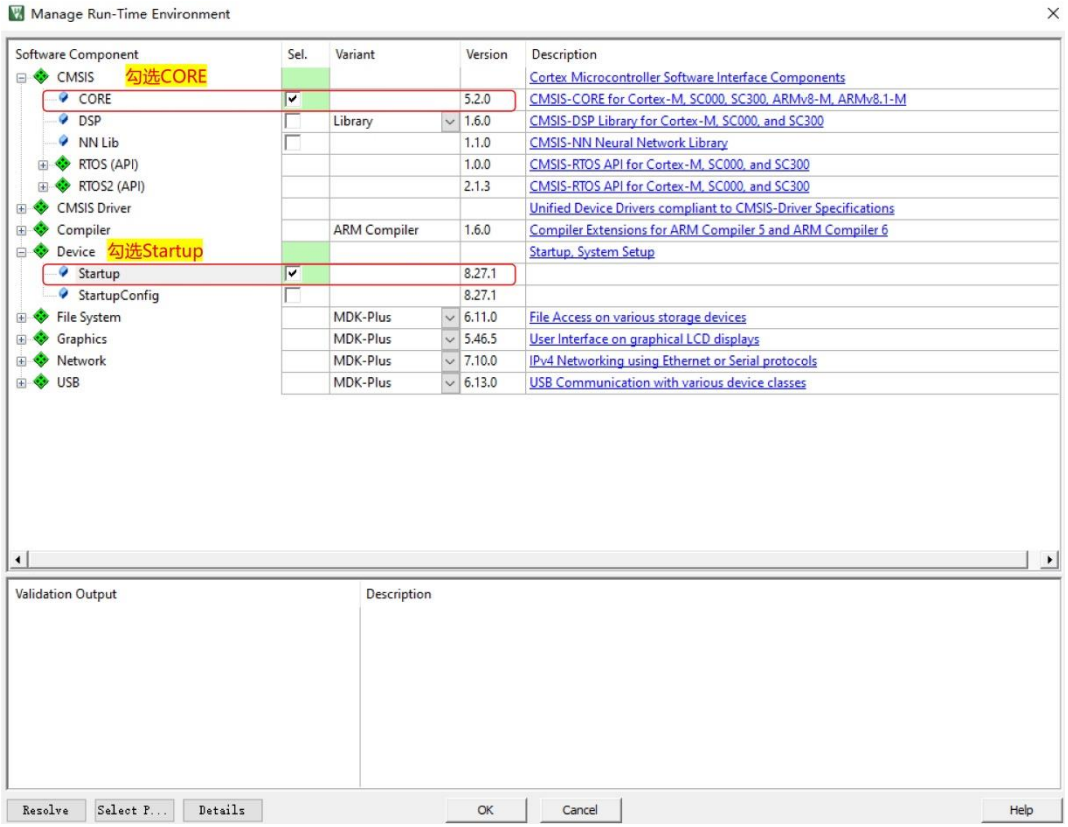


图 2-6：配置 RTE

✧ **注意事项：**再次说明一下，新建工程配置 RTE 时，对于 CORE 和 StartUp，MDK 会自动选择最新的版本，在这个步骤我们直接选择最新版本，后面配置工程时再根据需要修改为自己需求的版本即可。

5. 管理 MDK 工程目录。

主要是添加组(也就是在 MDK 中添加文件夹)、修改组名称和软件包（pack），目的是为了目录清晰，方便添加文件和管理文件。这里我们使用的工程目录是参考 SDK 里面的 BLE 工程的，这么做的好处是和 SDK 保持一致，以方便我们阅读 SDK 里面的参考代码。

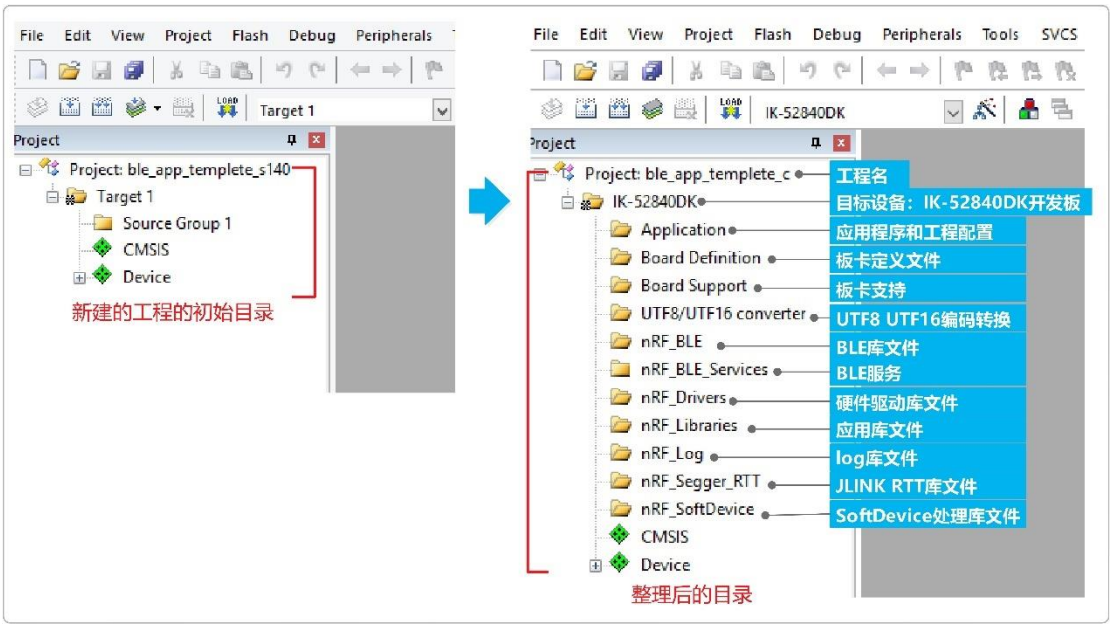


图 2-7：整理工程目录

1) 整理目录时会用到下面几个操作：

- 修改组名称：先选中需要修改名称的组，然后单击即可修改名称。**注意，不是双击。**
- 添加组：选中 Target(即整理后的目录中的 IK-52840DK)，右键选中“Add Group”，即可添加一个组。
- 调整组的顺序（如果组中添加了文件，也可以在这里调整文件的顺序）。

  点击工程项目管理图标

图 2-8：打开工程项目管理窗口

按照下图所示根据自己的习惯调整组的顺序，如果组中添加了文件，也可以调整组中文件的顺序。

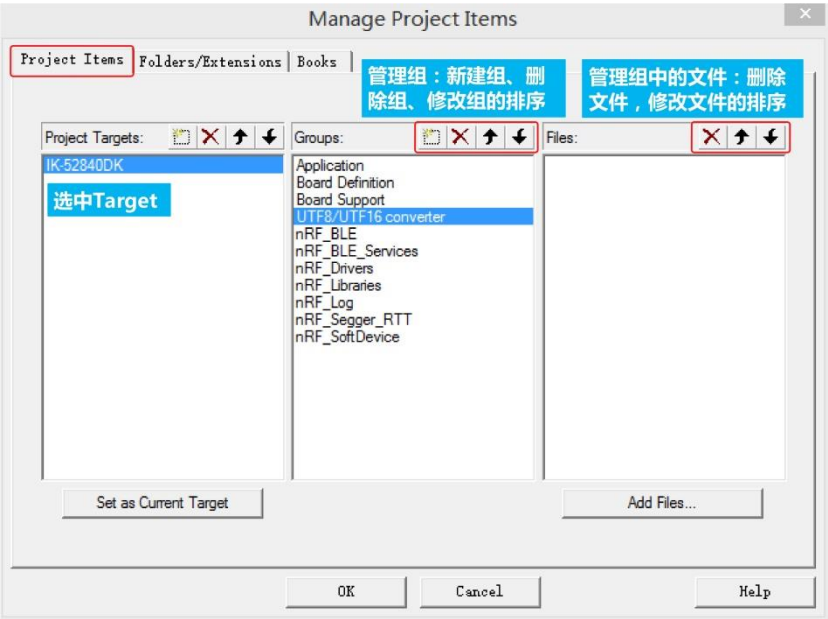


图 2-9：调整组的顺序

2) 管理软件包：这里不需要过多操作，我们可以在这修改 StartUp 的版本和选择是否自动使用安装的最新的软件包。

- 设置是否自动使用安装的最新的软件包：

勾选“Use latest versions of all installed Software Packs”即可打开自动使用安装的最新的软件包，不勾选即可以使用指定的版本。

- 修改 StartUp 和 CORE 的版本：

如我们的计算机中安装了 Nordic nRF5x 几个版本的器件包，就会显示多个版本的器件包，因为 SDK16.0 使用的 CMSISI 版本是 4.5，器件包版本是 8.27.1，所以我们要选择 CMSIS 的版本是 4.5，器件包的版本是 8.27.1，同时去掉自动选择最新版本的勾选，如下图所示。

点击下图所示的图标，打开软件包管理窗口。

图 2-10：打开软件包管理窗口

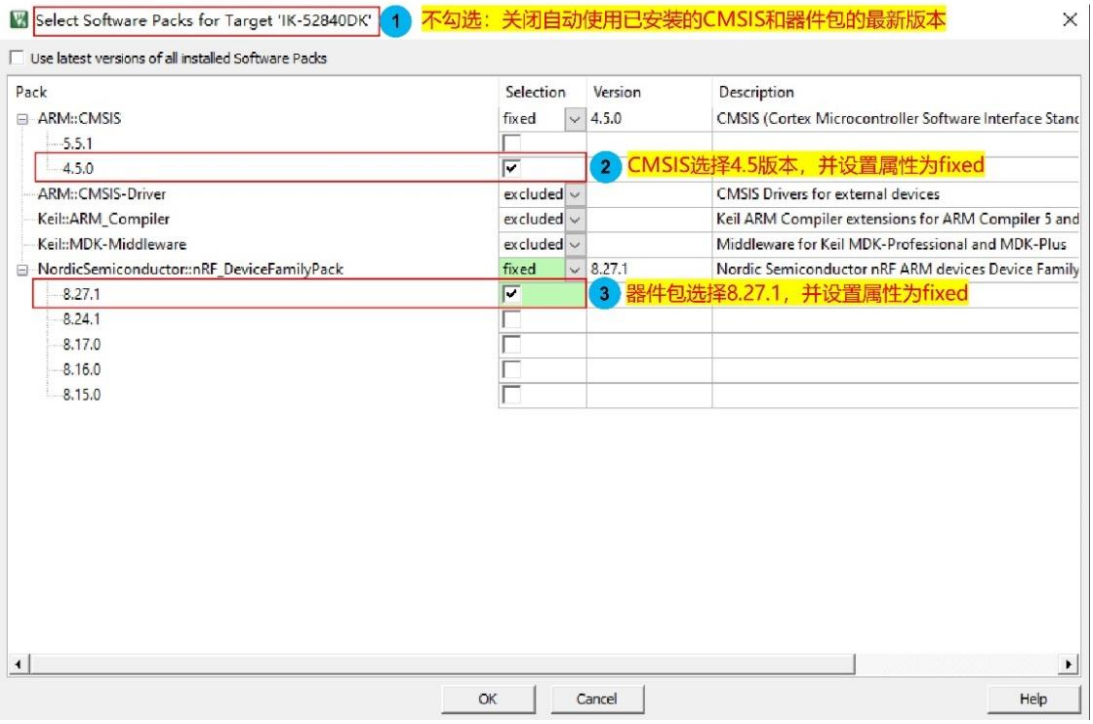


图 2-11：修改 StartUp 和 CORE 的版本

这里设置下面 3 项，这么设置是为了让我们建立的工程使用确定的软件版本，这样就可以避免因为自动引用最新版本引起问题的风险。

- 关闭自动使用最新的软件包功能。
- CMSIS 版本设置为 4.5.0，并设置版本 fixed。
- pack 版本设置为 8.27.1，并设置版本 fixed。

2.4. 添加需要的库文件

1. Board Definition 组

“Board Definition”组中需要加入板卡定义文件“boards.c”，该文件针对于具体板卡，

主要要用来定义板卡的指示灯、按键、IO 输出电压（仅 nRF52840 具有此功能）以及常用的操作函数。对于 IK-52840DK 开发板（兼容 Nordic 的 PCA10056），设计了 4 个指示灯和按键，在“boards.c”文件中，我们可以看到这些按键和指示灯的初始化函数以及一些基本功能如点亮、熄灭、翻转的操作函数。

表 2-1：“Board Definition”组中加入的文件

文件名	路径
boards.c	..\..\components\boards

2. Board Support 组

“Board Support”组加入的是板卡支持包文件，即 BSP 文件，主要用于实现指示灯和按键的驱动，如下表所示。

表 2-2：“Board Support”组中加入的文件

文件名	路径
bsp.c	..\..\components\libraries\bsp

3. UTF8/UTF16 converter 组

“UTF8/UTF16 converter”组只需加入一个实现 UTF8/UTF16 编码转换的文件，如下表所示。

表 2-3：“UTF8/UTF16 converter”组中加入的文件

文件名	路径
utf.c	..\..\external\utf_converter

4. nRF_BLE 组

“nRF_BLE”组加入的是 BLE 相关的库文件，如广播、连接参数协、配对管理等等，“nRF_BLE”组中加入的文件如下表所示。

表 2-4：“nRF_BLE”组中加入的文件

文件名	路径
ble_advdata.c	..\..\components\ble\common
ble_conn_params.c	..\..\components\ble\common
ble_conn_state.c	..\..\components\ble\common
ble_srv_common.c	..\..\components\ble\common
ble_advertising.c	..\..\components\ble\ble_advertising
nrf_ble_gatt.c	..\..\components\ble\nrf_ble_gatt
nrf_ble_qwr.c	..\..\components\ble\nrf_ble_qwr

5. nRF_BLE_Services 组

预留，后续章节添加 BLE 服务文件。

6. nRF_Drivers 组

“nRF_Drivers”组加入的是各种外设的驱动库文件，如时钟、GPIOTE、UART 等等，“nRF_Drivers”组中加入的文件如下表所示。

表 2-5：“nRF_Drivers”组中加入的文件

文件名	路径
nrf_drv_clock.c	..\..\integration\nrfx\legacy
nrf_drv_uart.c	..\..\integration\nrfx\legacy
nrfx_clock.c	..\..\modules\nrfx\drivers\src
nrfx_gpiote.c	..\..\modules\nrfx\drivers\src
nrfx_prs.c	..\..\modules\nrfx\drivers\src\prs
nrfx_uart.c	..\..\modules\nrfx\drivers\src
nrfx_uarte.c	..\..\modules\nrfx\drivers\src
nrf_atomic.c	..\..\components\libraries\atomic

7. nRF_Libraries 组

“nRF_Libraries”组加入的是各种应用的库文件，如 APP 定时器、CRC 校验、软件 FIFO、简易文件系统以及错误处理等等，“nRF_Libraries”组中加入的文件如下表所示。

表 2-6：“nRF_Libraries”组中加入的文件

文件名	路径
app_button.c	..\..\components\libraries\button
app_scheduler.c	..\..\components\libraries\scheduler
app_timer2.c	..\..\components\libraries\timer
drv_rtc.c	..\..\components\libraries\timer
app_uart_fifo.c	..\..\components\libraries\uart
retarget.c	..\..\components\libraries\uart
hardfault_implementation.c	..\..\components\libraries\hardfault
nrf_atfifo.c	..\..\components\libraries\atomic_fifo
nrf_atflags.c	..\..\components\libraries\atomic_flags
nrf_balloc.c	..\..\components\libraries\balloc
nrf_fprintf.c	..\..\external\fprintf
nrf_fprintf_format.c	..\..\external\fprintf

nrf_memobj.c	..\..\components\libraries\memobj
nrf_pwr_mgmt.c	..\..\components\libraries\pwr_mgmt
nrf_ringbuf.c	..\..\components\libraries\ringbuf
nrf_section_iter.c	..\..\components\libraries\experimental_section_vars
app_error.c	..\..\components\libraries\util
app_error_handler_keil.c	..\..\components\libraries\util
app_error_weak.c	..\..\components\libraries\util
app_util_platform.c	..\..\components\libraries\util
nrf_assert.c	..\..\components\libraries\util
nrf_strerror.c	..\..\components\libraries\strerror
nrf_sortlist.c	..\..\components\libraries\sortlist

8. nRF_Log 组

“nRF_Log”组中加入的文件如下表所示，Log 程序模块为程序提供日志打印功能。

表 2-7：“nRF_Log”组中加入的文件

文件名	路径
nrf_log_backend_rtt.c	..\..\components\libraries\log\src
nrf_log_backend_serial.c	..\..\components\libraries\log\src
nrf_log_backend_uart.c	..\..\components\libraries\log\src
nrf_log_default_backends.c	..\..\components\libraries\log\src
nrf_log_frontend.c	..\..\components\libraries\log\src
nrf_log_str_formatter.c	..\..\components\libraries\log\src

9. nRF_Segger_RTT 组

“nRF_Segger_RTT”组中加入的文件如下表所示，它们的作用是实现 JLINK-RTT Viewer 作为 Log 输出终端，打印 Log。

表 2-8：“nRF_Segger_RTT”组中加入的文件

文件名	路径
SEGGER_RTT.c	..\..\external\segger_rtt
SEGGER_RTT_Syscalls_KEIL.c	..\..\external\segger_rtt
SEGGER_RTT_printf.c	..\..\external\segger_rtt

10. nRF_SoftDevice

“nRF_SoftDevice”组中加入的文件如下表所示，它们用于使能或禁止 SoftDevice 以及

向应用程序发布 SoftDevice 事件，功能如下：

- 使能 SoftDevice 及其事件中断。
- 禁止 SoftDevice。
- 从 SoftDevice 接收堆栈事件，并将它们转发给应用程序已注册的事件监查者。
- 发送 SoftDevice 状态事件给应用程序和驱动。
- 发送可由事件监查者接受或拒绝的 SoftDevice 状态请求

表 2-9：“nRF_SoftDevice”组中加入的文件

文件名	路径
nrf_sdh.c	..\..\components\softdevice\common
nrf_sdh_ble.c	..\..\components\softdevice\common
nrf_sdh_soc.c	..\..\components\softdevice\common

2.5. 添加工程配置向导文件

工程配置向导文件名称是“sdk_config.h”，我们可以自己编写配置向导文件，不过更方便的方法是直接从 SDK 中拷贝配置向导文件，然后根据自己工程的需求进行修改。

SDK 中各个例子的工程都含有一个配置向导文件“sdk_config.h”，路径是“…\pca10056\s140\config”，我们可以拷贝和我们自己的应用比较接近的例子的配置向导文件来进行修改。

具体操作为拷贝“sdk_config.h”文件到工程的“…\project\config”目录下，然后将它添加到工程的“Application”组，添加的时候要注意文件类型选择“All files”，如下图所示。

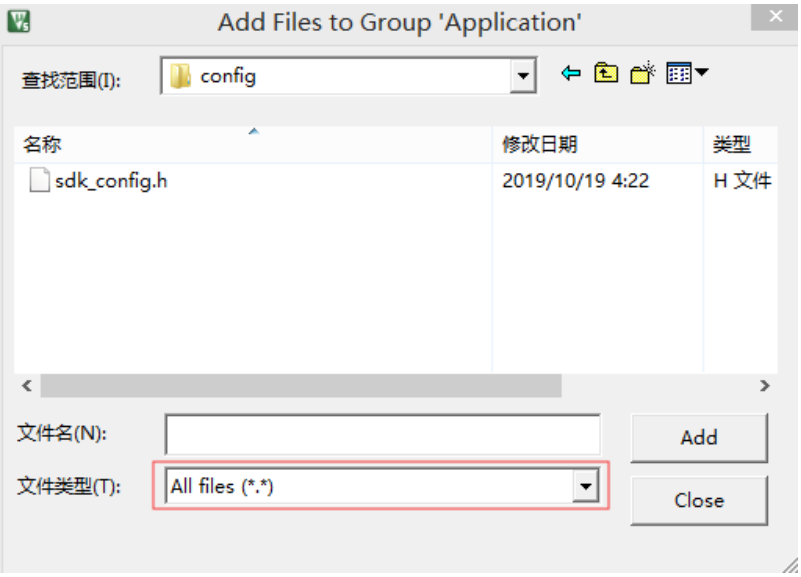


图 2-12：添加“sdk_config.h”文件

2.6. 新建 main.c 文件并添加到工程。

1. 执行“File→New”新建文件，如下图：

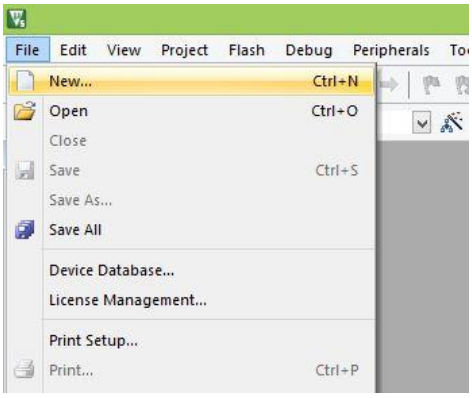


图 2-13：新建文件

2. 点击保存按钮“Save”将文件保存为“main.c”。

通常，工程都会包含一个“main.c”文件，用于存放 C 程序的入口函数（main()函数），所以，我们需要先新建一个“main.c”文件并添加到工程。

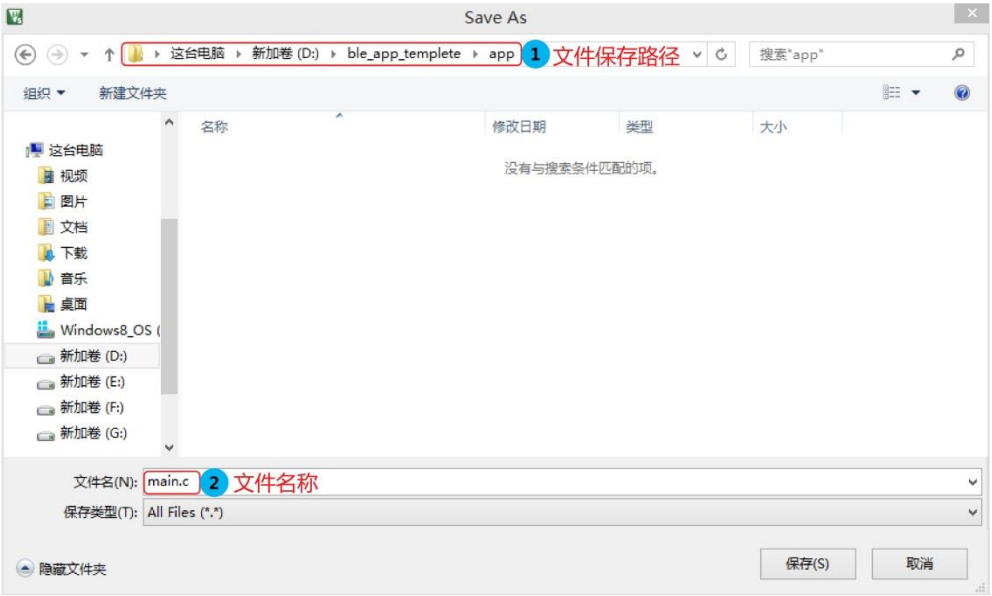


图 2-14：保存文件

3. 将“main.c”文件添加到工程。

“main.c”文件创建后并不会自动加入到工程，需要手动添加。下面是将“main.c”文件添加到 Application 组的方法，可以通过三种方式来添加。

- **方法 1：**双击组名 Application 打开添加文件窗口，导航到“main.c”文件的存放路径，添加即可。
- **方法 2：**选中 Application 组，右键选择“Add Existing Files To…”打开添加文件窗口，导航到“main.c”文件的存放路径，添加文件。
- **方法 3：**通过工程项目管理添加。点击工具栏中的图标，打开工程项目管理窗口，添加文件。如下图所示：

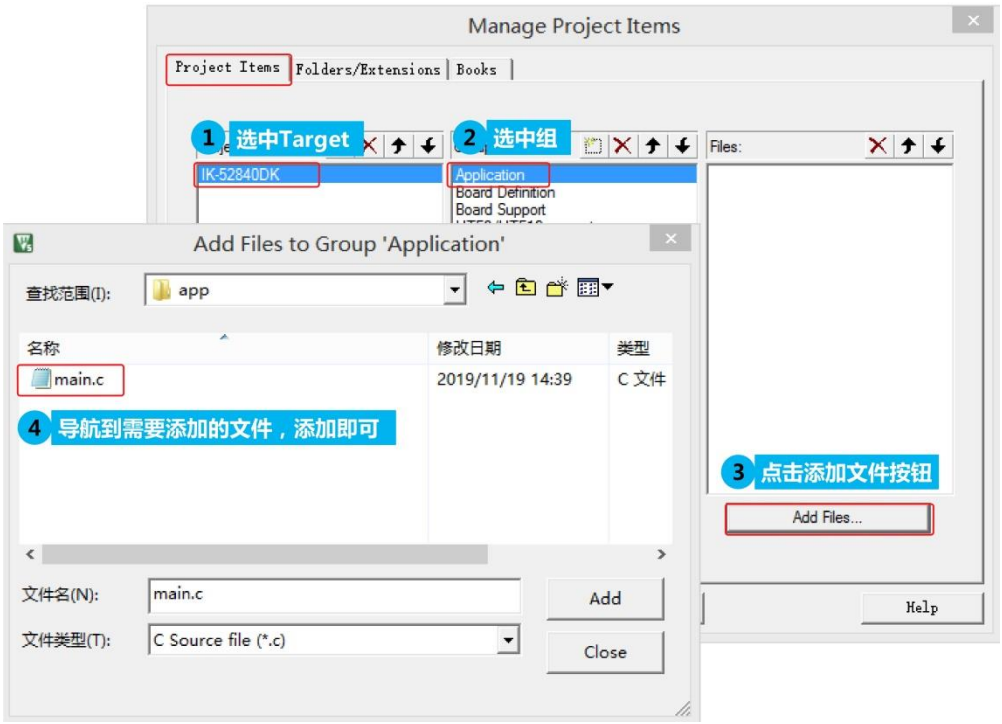


图 2-15：通过工程项目管理添加文件

3. 配置工程

文件添加完成后，接下来需要对工程进行配置，对于 BLE 工程来说，主要配置的项目包括：“Target”、“Output”、“C/C++”、“Linker” 和 “Debug”。

3.1. 配置 “Target” 选项卡

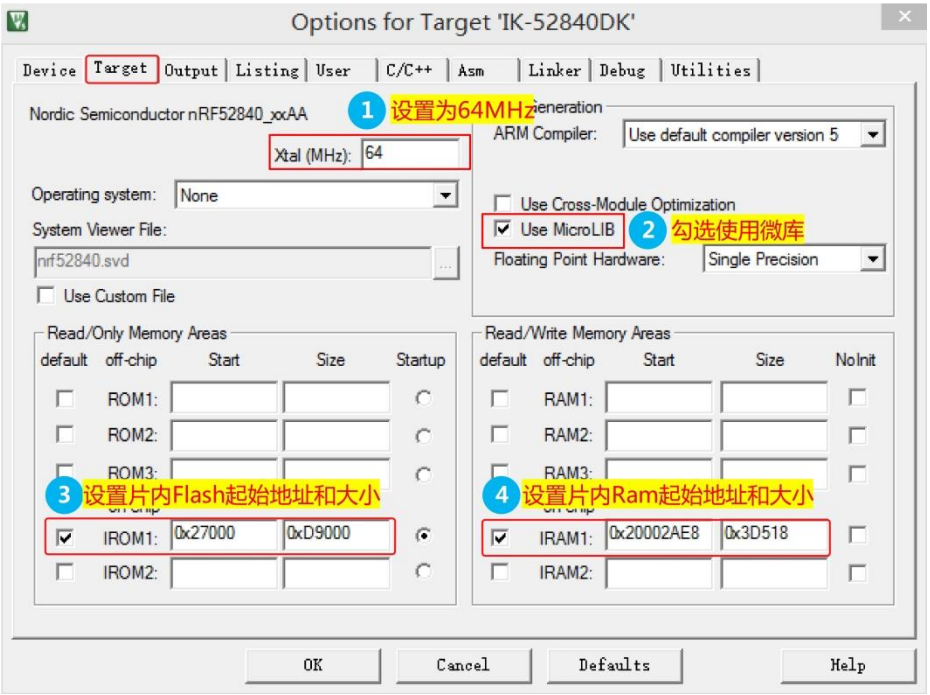


图 2-16：配置 “Target” 选项卡

主要配置下面几个项目：

- 晶振频率：晶振频率 Xtal 是用于软件仿真的，设置或不设置对硬件烧写和仿真都没有影响。这里设置为 64MHz。
- MicroLib 库：勾选使用 MicroLib 库。

  Microlib 是缺省 C 库的备选库。为进一步改进基于 ARM 处理器的应用代码密度，RealView MDK 采用了新型 Microlib C 库（用于 C 的 ISO 标准运行时库的一个子集），并将其代码镜像降低最小以满足微控制器应用的需求。Microlib C 库进行了高度优化以使代码变得很小，可将运行时库代码大大降低。
- 内存设置

对于 BLE 工程，片内 Flash 和 RAM 都需要保留一部分给协议栈使用，BLE 工程模板内存设置如下：

片内 ROM 设置：

起始地址(16 进制)	大小(16 进制)
0x27000	0xD9000

片内 RAM 设置：

起始地址(16 进制)	大小(16 进制)
0x20002AE8	0x3D518

3.2. 配置“Output”选项卡

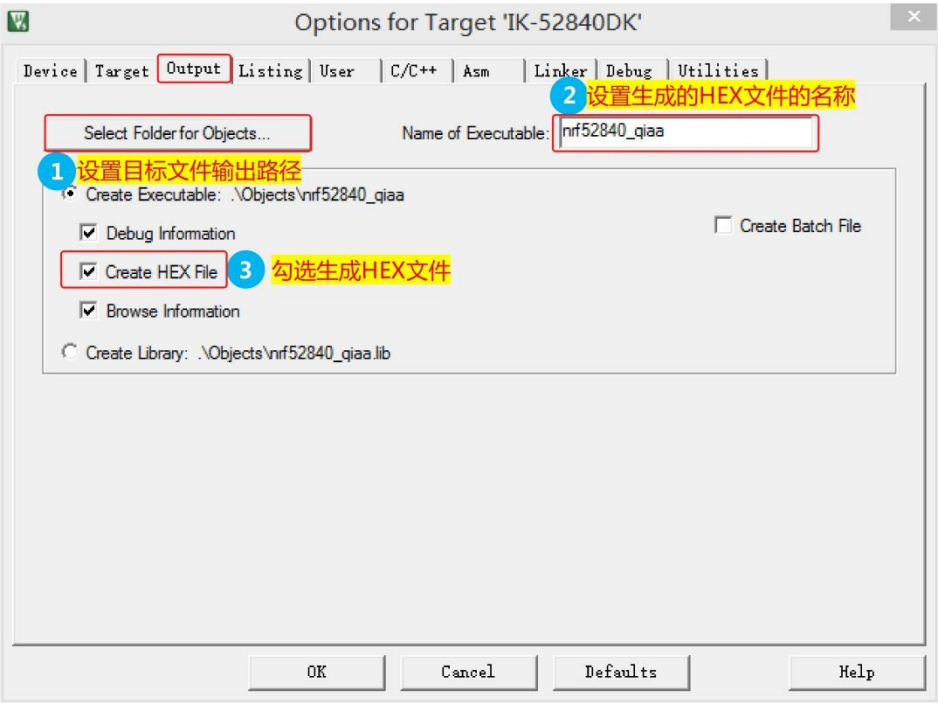


图 2-17：配置“Output”选项卡

主要配置下面几个项目：

- 指定目标文件输出路径：若无必要，输出路径不用修改，使用默认的即可。
- 设置生成 HEX 文件。

  需要勾选“Create HEX File”，使能生成 HEX 文件，并设置一下输出的 HEX 文件的名

称，设置之后，工程编译成功即可生成 HEX 文件，生成的 HEX 文件位于指定的“目标文件输出路径”的目录下。

- HEX 文件名称：输入生成的 HEX 文件的名称。

3.3. 配置“C/C++”选项卡

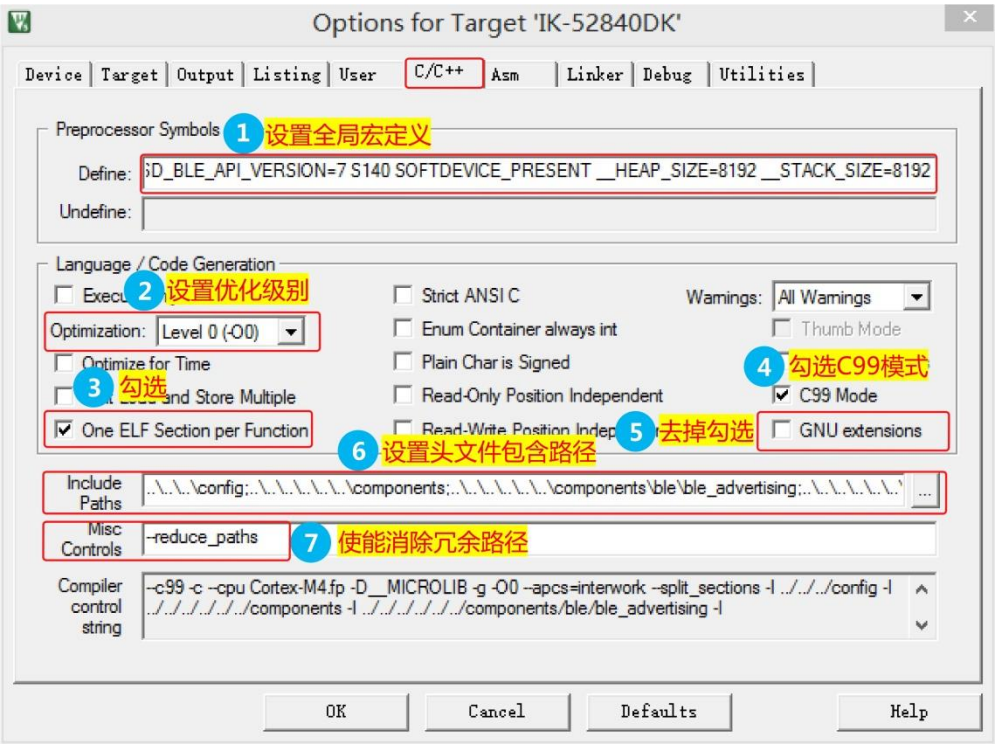


图 2-18：配置“**C/C++**”选项卡

主要配置下面几个项目：

1. 全局宏定义

全局宏定义是被整个工程使用的宏定义，如果有多个宏定义，用空格隔开。注意，这里定义的宏定义，对于整个工程有效。本例中需要加入下面几个宏定义：

- **APP_TIMER_V2：**使用 V2 版本 APP 定时器。
- **APP_TIMER_V2_RTC1_ENABLED：**V2 版本 APP 定时器需要使用 RTC1 产生时基。
- **BOARD_PCA10056：**因为芯片是 nRF52840，所以要定义 BOARD_PCA10056，定义之后，工程会包含“pca10056.h”头文件，该头文件中定义了开发板外设如 led、按键、串口等使用的引脚。当然，我们也可以不包含这个头文件，自己来定义引脚。
- **CONFIG_GPIO_AS_PINRESET：**使能 P0.18 的复位功能，即加入该全局宏定义后，P0.18 不能作为普通 IO 使用，只能作为复位引脚。
- **NRF52840_XXAA：**指定芯片型号是 nRF52 系列中的 nRF52840_XXAA。
- **NRF_SD_BLE_API_VERSION=7：**工程用的协议栈的版本，SDK16.0 使用的协议栈版本是 7.0.1，所以这个宏定义的值设置为 7。
- **SOFTDEVICE_PRESENT：**该工程使用协议栈。
- **S140：**该工程使用的协议栈是 S140（nRF52840 对应的 BLE 协议栈是 S140）。
- **FLOAT_ABI_HARD：**浮点运算编译选项。

- **__STACK_SIZE=8192**：栈大小。
- **__HEAP_SIZE=8192**：堆大小。

2. 设置优化级别“Optimization”。

0 表示不优化，设置越大，优化级别越高。一般仿真调试的时候，优化级别设置为 Level 0（最低），调试完成后，设置为 Level 3（最高），以减小编译后的代码大小。

3. 勾选“One ELF Secion per Function”。

One ELF Secion per Function 的机制是将每一个函数作为一个优化的单元，而并非整个文件作为参与优化的单元。该机制具有的这种优化功能特别重要，尤其是在对于生成的二进制文件大小有严格要求的场合。

One ELF Section per Function 对于一个大工程的优化效果尤其突出，对于小工程优化效果不是很明显。想象一下这样的一个应用场合：在 nRF52840 程序开发过程中，我们会使用 SDK 中的组件库“components”，我们加入组件库中的一个文件到工程并不表示我们会使用这个文件中所有的函数，这样，最后生成的二进制文件中就有可能包含众多的冗余函数，造成了存储空间的浪费，通过使用 One ELF Section per Function，即可在最后生成的二进制文件中将冗余函数排除掉，从而节省存储空间。

4. 启用“C99 Mode”

在组件库中，很多地方变量的声明放在了可执行语句的后面(C99 之前不允许这么做)，如果要使用组件库，就需要勾选这个选项，否则编译的时候会出现很多错误。

5. 设置头文件包含路径：

设置头文件包含路径在前文已有描述，此处不再赘述。

6. 设置消除冗余路径

Misc Controls 栏中填入“--reduce_paths”关键字，设置 MDK 自动消除冗余路径。

3.4. 配置“Linker”选项卡

“Linker”选项卡中需要配置的是忽略警告设置，BLE 工程编译的时候会产生 6330 的警告，这里我们在“Misc controls”栏加入“--diag_suppress 6330”忽略掉 6330 警告，如下图所示。

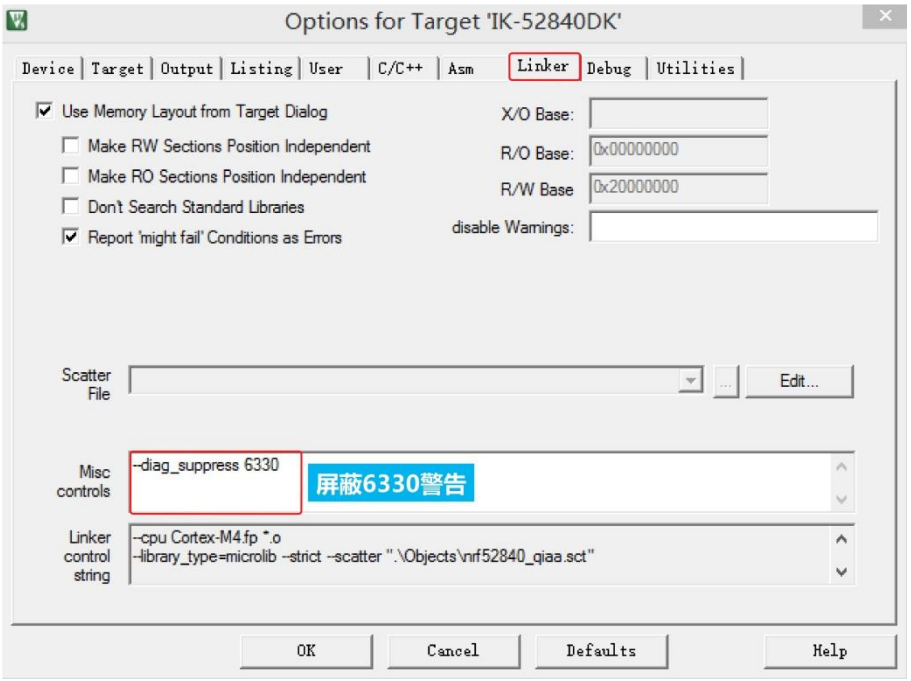


图 2-19：配置“Linker”选项卡

3.5. 配置“Debug”选项卡

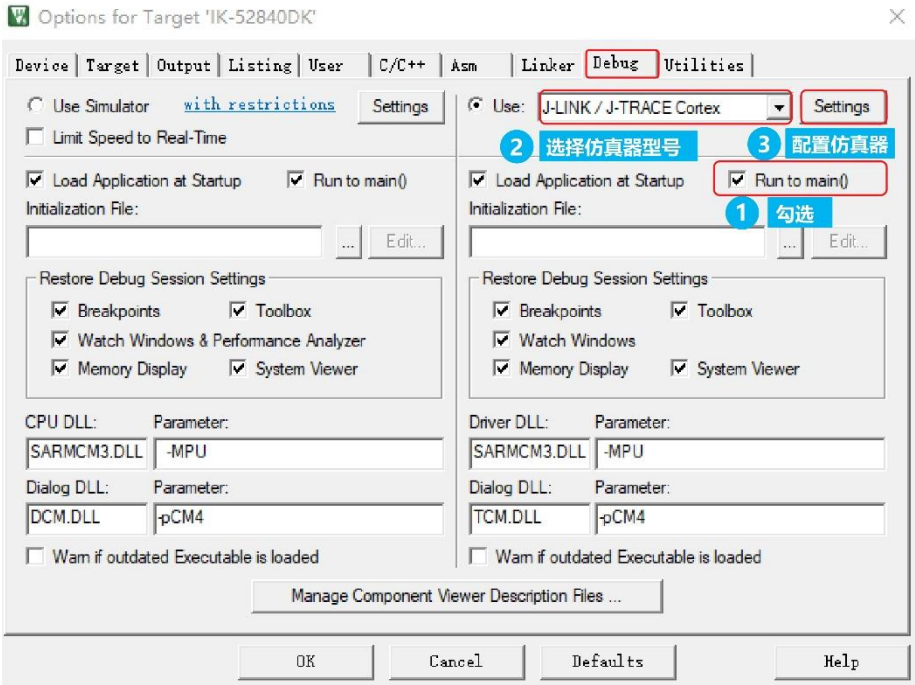


图 2-20：配置“Debug”选项卡

主要配置下面几个项目：

- Run to main()：设置仿真时，程序是否自动运行到 main()函数，勾选后，仿真时，程序会自动运行到 main()函数。
- 仿真器设置：本教程使用的仿真器是 JLINK，所以这里选择“J-LINK/J-TRACE Cortex”，之后，点击“Setting”按钮进入设置界面。(注意：设置时需要将 JLINK 仿真器连接到

计算机)。

仿真器设置界面如下，设置时，先切换到“Debug”选项卡，将调试接口设置为“SW”。如下图所示。

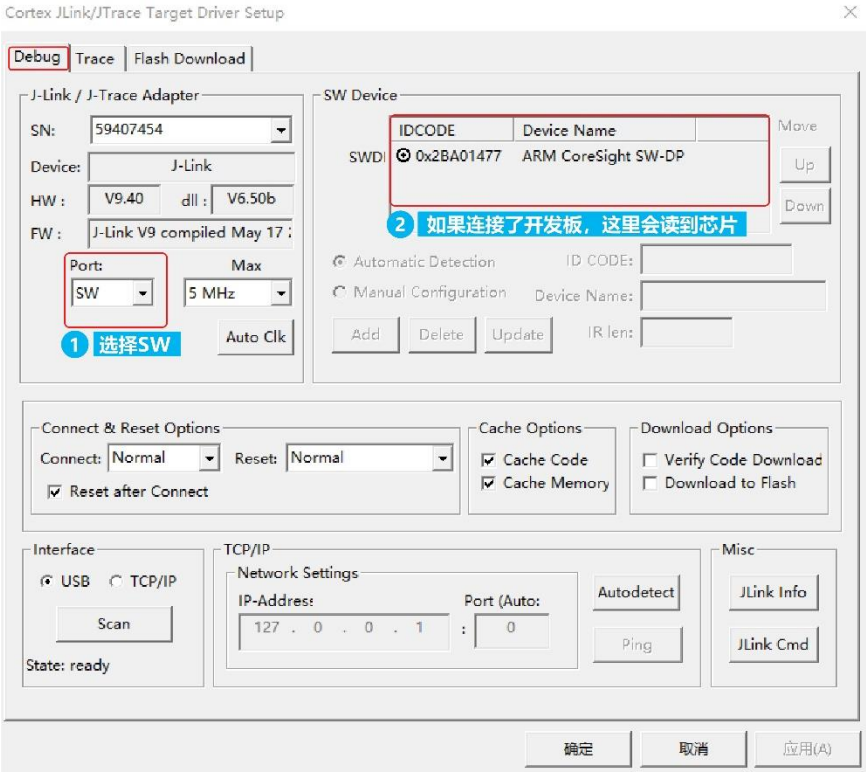


图 2-21：配置调试接口

切换到“Flash Download”选项卡，加载编程算法。

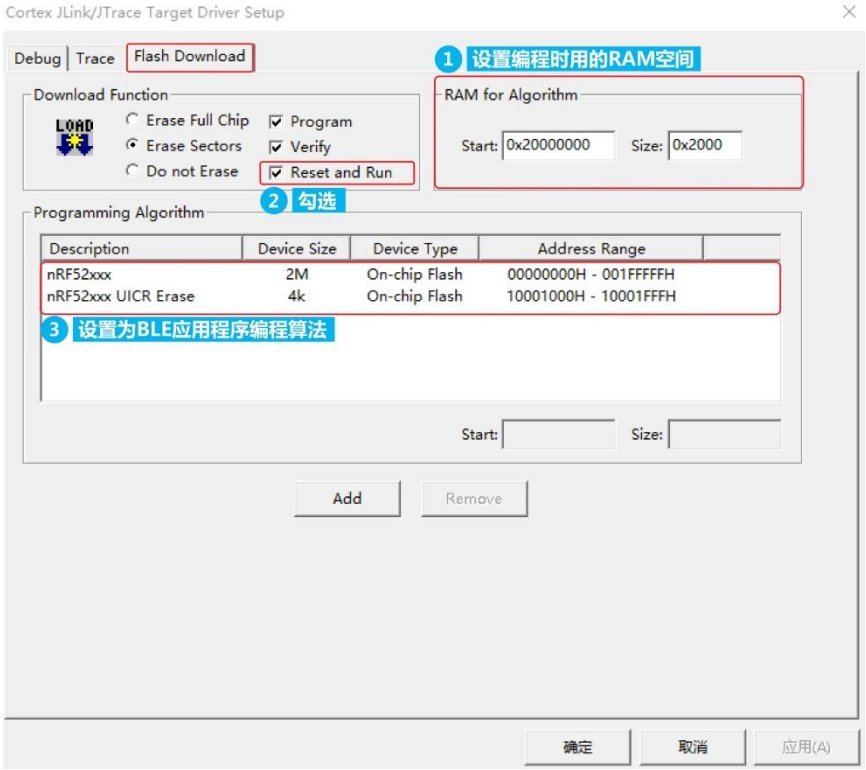


图 2-22：配置下载选项

“Flash Download”选项卡主要设置项目如下：

- Reset and Run：下载完成后自动复位并运行程序，这一项很重要。有时候我们发现程序下载后需要断电重启后才能运行，原因有可能就是漏勾选了这一项。
- RAM for Algorithm：指定用于烧写程序的 RAM 区域，一般使用默认设置即可。
- program algorithm：Flash 编程算法，一般建立工程选择芯片后会自动添加，如果没有，点击“ADD”按钮按照上图添加即可。

4. 协议栈“Target”

这一步是为了使用 MDK 来下载协议栈的 HEX 文件，在 MDK 中，源码编译后可以直接下载，但是 MDK 不能直接下载 HEX 文件，而协议栈是已经编译好的 HEX 文件，这里，我们通过新建一个协议栈的“Target”来实现在 MDK 中直接下载协议栈的 HEX 文件。

4.1. 新建协议栈“Target”

点击 MDK 工具栏中的图标，打开工程项目管理窗口，新建协议栈“Target”，如下图所示。

新建的“Target”名称使用协议栈的名称“flash_s140_nrf52_7.0.1_softdevice”，这样看起来比较方便，容易区别于应用程序。

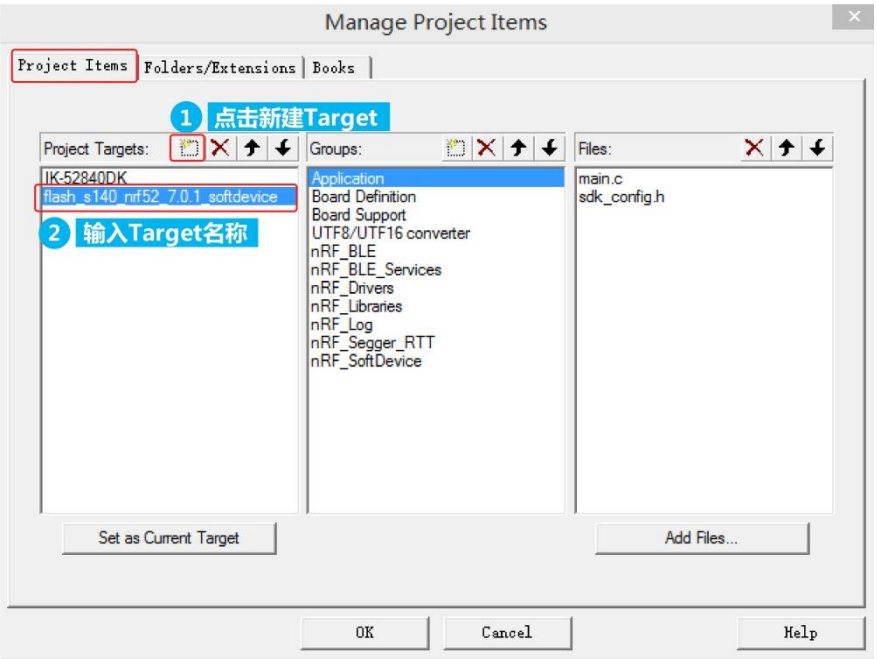


图 2-23：新建下载协议栈的“Target”

新建下载协议栈的“Target”后，同样需要对它进行配置，这时点击 MDK 的“Select Target”栏的下拉按钮，会看到 2 个 Target，分别对应应用程序和协议栈，这里要选择协议栈的 Target 进行配置，如下图所示。

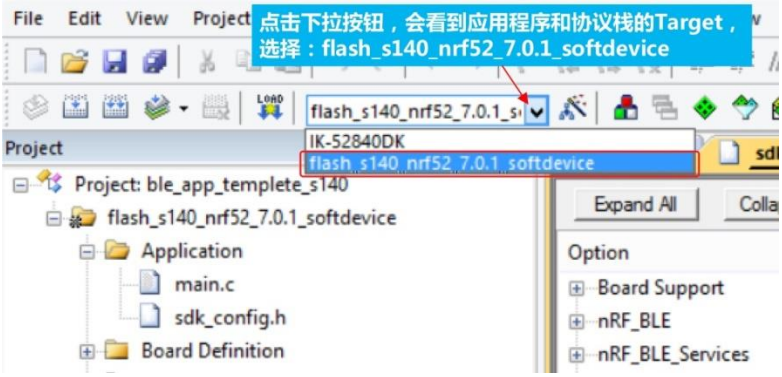


图 2-24：切换到协议栈的“Target”

选择协议栈的 Target 后，接下来需要对其进行配置，需要配置的项目是：“Output”、和“Debug”。

4.2. 配置“Output”选项卡

点击魔术棒，打开配置窗口，之后切换到“Output”选项卡，如下图所示。

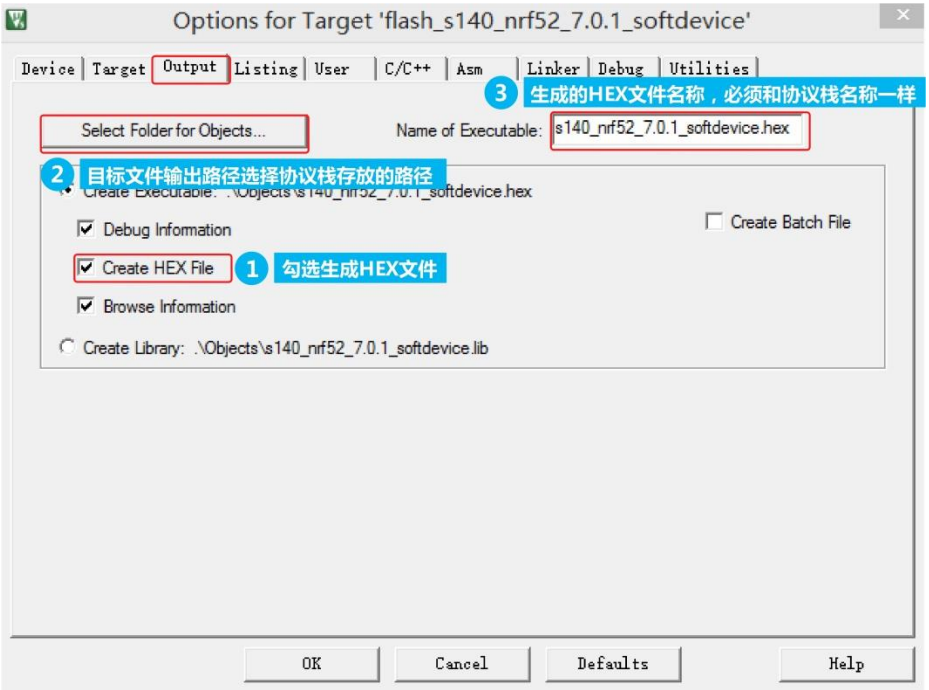


图 2-25：配置“Output”选项卡

配置下面几个项目：

- 勾选生成 HEX 文件的选项。
- 填入生成的 HEX 文件的名称，这里必须和协议栈名称一样，即必须输入“s140_nrf52_7.0.1_softdevice.hex”。
- 指定目标文件输出路径：该路径必须是协议栈 HEX 文件的存放路径，nRF52840 的 BLE 协议栈 HEX 文件在 SDK 中的路径是“..\components\softdevice\s140\hex”，所以这里必须设置指向这个路径。

4.3. 配置“Debug”选项卡

点击魔术棒，打开配置窗口，之后切换到“Debug”选项卡，按照如下图所示配置。

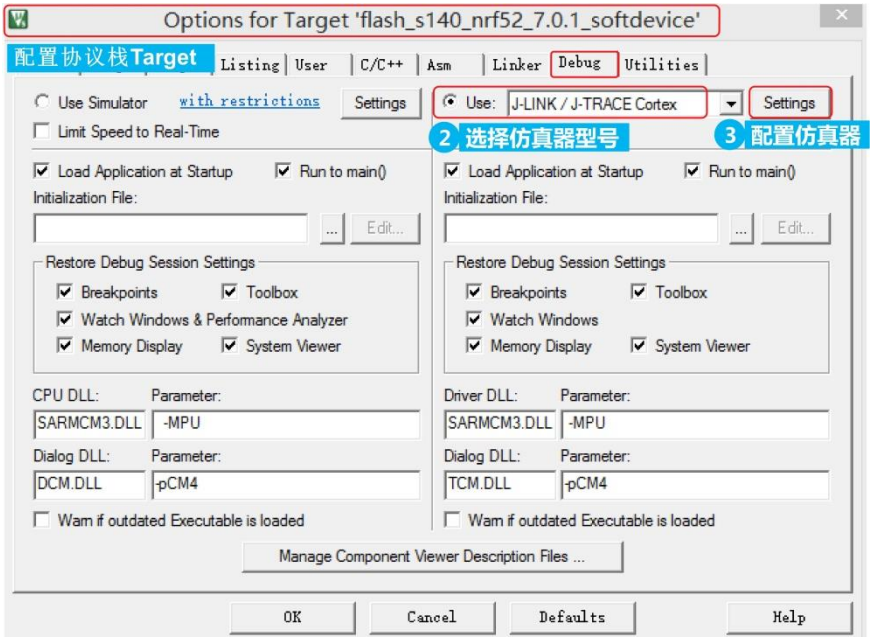


图 2-26：配置“Debug”选项卡

对于协议栈的 Target 来说，只需设置仿真器即可，因为本教程使用的仿真器是 JLINK，所以这里选择“J-LINK/J-TRACE Cortex”，之后，点击“Setting”按钮进入设置界面。(注意：设置时需要将 JLINK 仿真器连接到计算机)。

仿真器设置界面如下，设置时，先切换到“Debug”选项卡，将调试接口设置为“SW”。如下图所示。

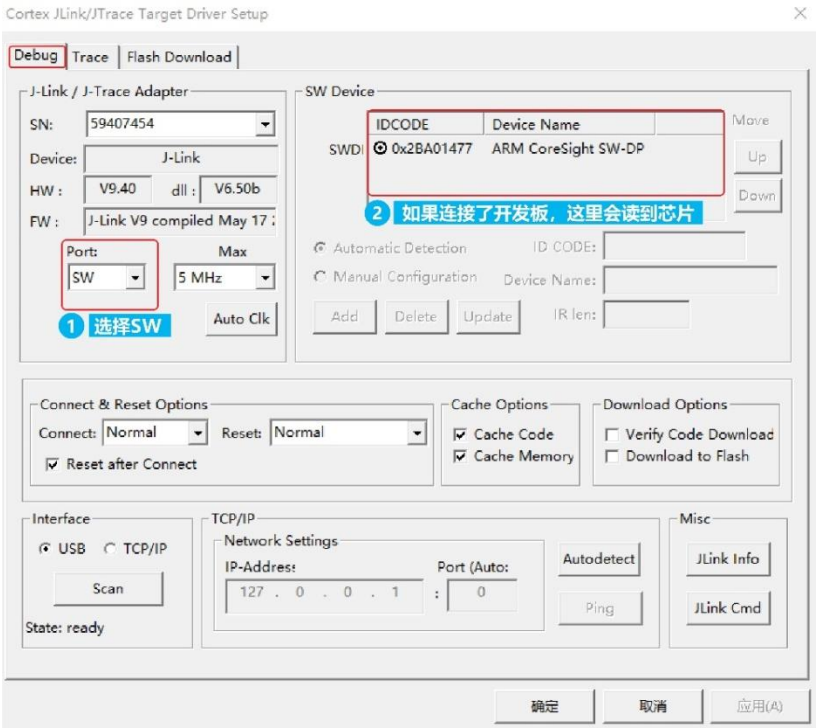


图 2-27：配置调试接口

再切换到“Flash Download”选项卡，加载编程算法，注意这里需要加载协议栈的编程算法，如下图所示。

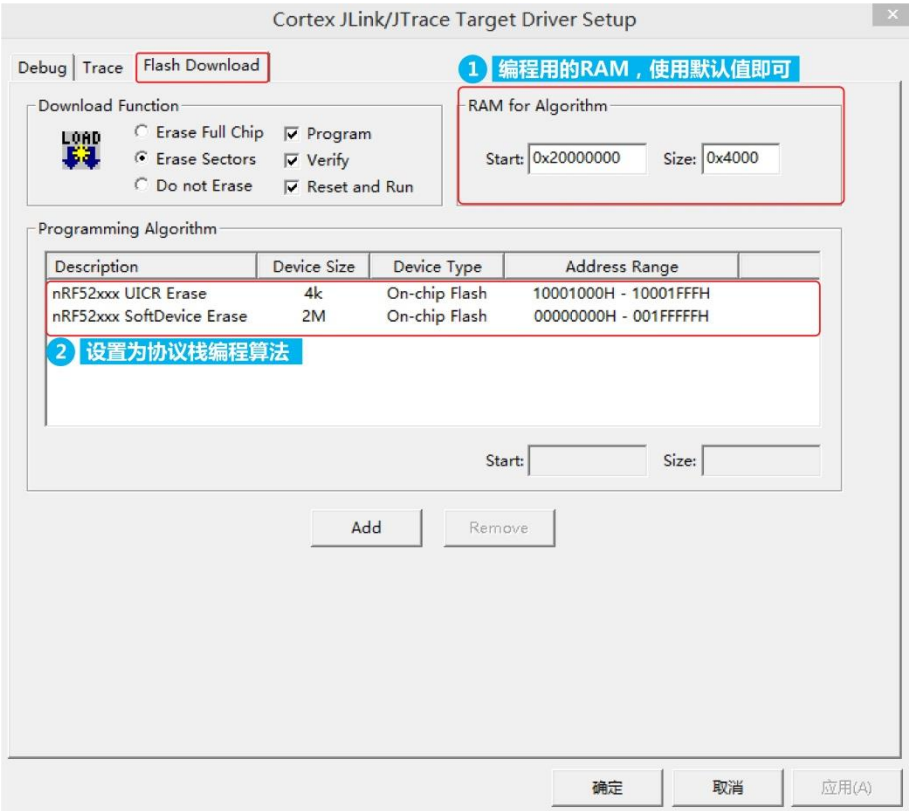


图 2-28：配置下载选项

“Flash Download”选项卡主要设置项目如下：

- RAM for Algorithm：指定用于烧写程序的 RAM 区域，一般使用默认设置即可。
- program algorithm：Flash 编程算法，按照上图所示设置。

4.4. 防止协议栈丢失

协议栈是预编译的，是以 HEX 文件提供的，协议栈没有源码，不需要编译。我们新建协议栈 Target 为了在 MDK 中能下载协议栈。因为新建的协议栈 Target，目标文件输出路径和生成的 HEX 文件和协议栈 HEX 文件完全一样，一旦编译成功就会覆盖协议栈的 HEX 文件，同时，MDK 在编译的时候，会先删除掉之前的 HEX 文件，所以，为了防止协议栈 HEX 文件丢失，可以人为制造错误，让协议栈 Target 无法编译，同时加入命令，编译的时候不删除协议栈 HEX。

1. 移除协议栈 Target 编译列表中的文件

- 选中组后右键，在弹出的菜单中点击“Options for group…”，如下图所示。

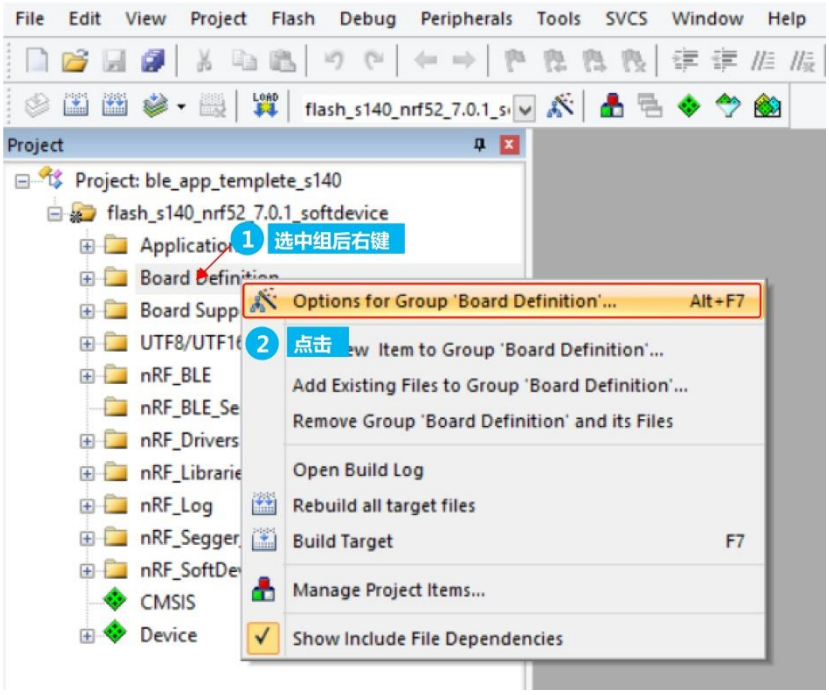


图 1-23：选中组

- 去掉“Include in Target Build”的勾选，如下图所示。

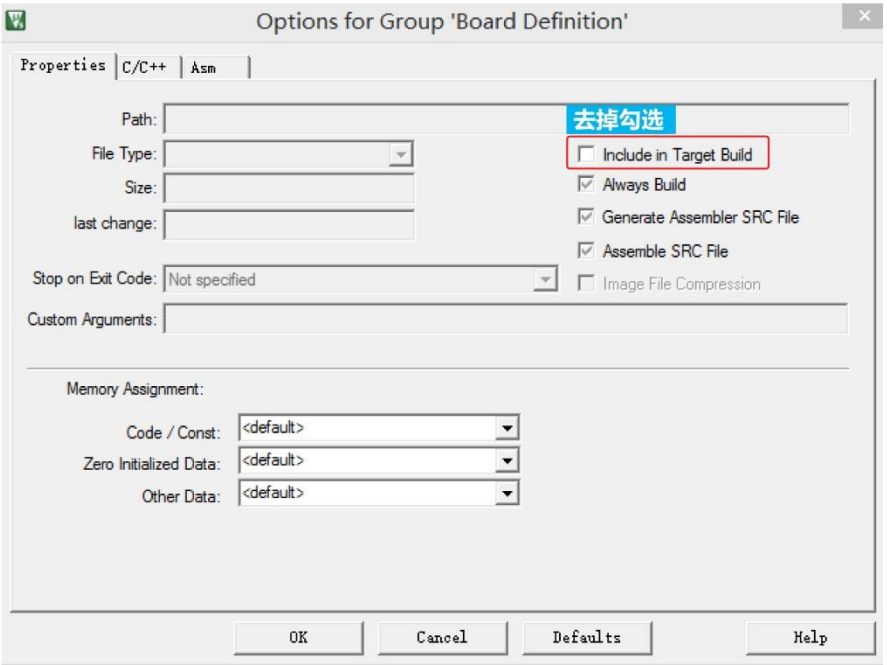


图 1-24：去掉勾选

- 重复上述操作，将其它组、CMSIS 和 Drvice 均从编译列表中移除。

2. User 选项卡中加入“attrib +R $H*”命令

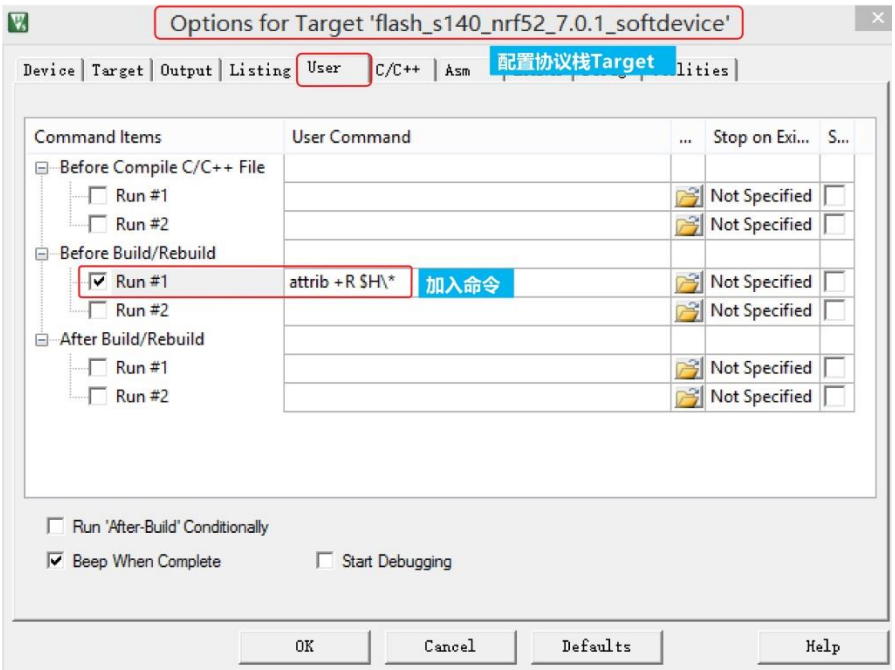


图 1-25：“User”中加入命令

到此，协议栈 Target 建立完成，我们用 MDK 下载协议栈或者进行全片擦除的时候，只需要切换到协议栈 Target，点击下载按钮或者执行“Flash→Earse”即可下载协议栈或全片擦除芯片。

第三章：实现 BLE 基本功能

1. 学习目的

1. 本章的目的是为了让读者掌握构建一个外围设备的最小 BLE 系统的框架以及需要使用的程序模块。
2. 读者在阅读本章时，应将注意力放在框架和模块上的整体功能上，自上而下去学习 BLE 开发，也就是我们先要搞懂 BLE 开发的程序结构和需要使用的模块以及模块的功能，之后再去深入学习各个模块。
3. 各个小节中，会列出读者关心的疑问，这些疑问会在后续章节中使用单章来讲解。如定时器模块，在本章我们需要知道为什么要初始化？它有哪些作用？如何初始化？至于定时器模块使用的细节会在后续章节单独讲解。

2. 实现的功能

本章实现的是一个 BLE 外围设备（外围设备和中心设备连接上之后就是我们常说的从机，而中心设备和外围设备连接上之后就是我们常说的主机）的“最小系统”，实现的功能如下表所示。

表 3-1：BLE 工程模板实现的参数和功能

项目	名称	内容
广播参数	广播间隔	187.5 ms。
	广播超时时间	无限广播，不超时。
	广播模式	快速广播。
广播数据	设备名称	BLE_Template。
	外观	包含。
	Flags	一般可发现模式，不支持 BR/EDR。
首选连接参数	最小连接间隔	100ms。
	最大连接间隔	200ms。
	从机延迟	0.
	监督超时	4 秒。
连接参数协商	首次协商延时	5 秒。
	每次协商之间的间隔	30 秒。

	最次协商最大尝试次数	3 次。
包含的服务	GAP 服务	强制包含。
	GATT 服务	强制包含。
指示灯	D1 闪烁	指示正在广播。
	D1 常亮	指示已经和中心设备建立连接。

3. BLE 程序结构

一个 BLE 程序通常需要至少包含 4 个必要的部分：系统初始化、启动、空闲管理和事件处理。如下图所示：

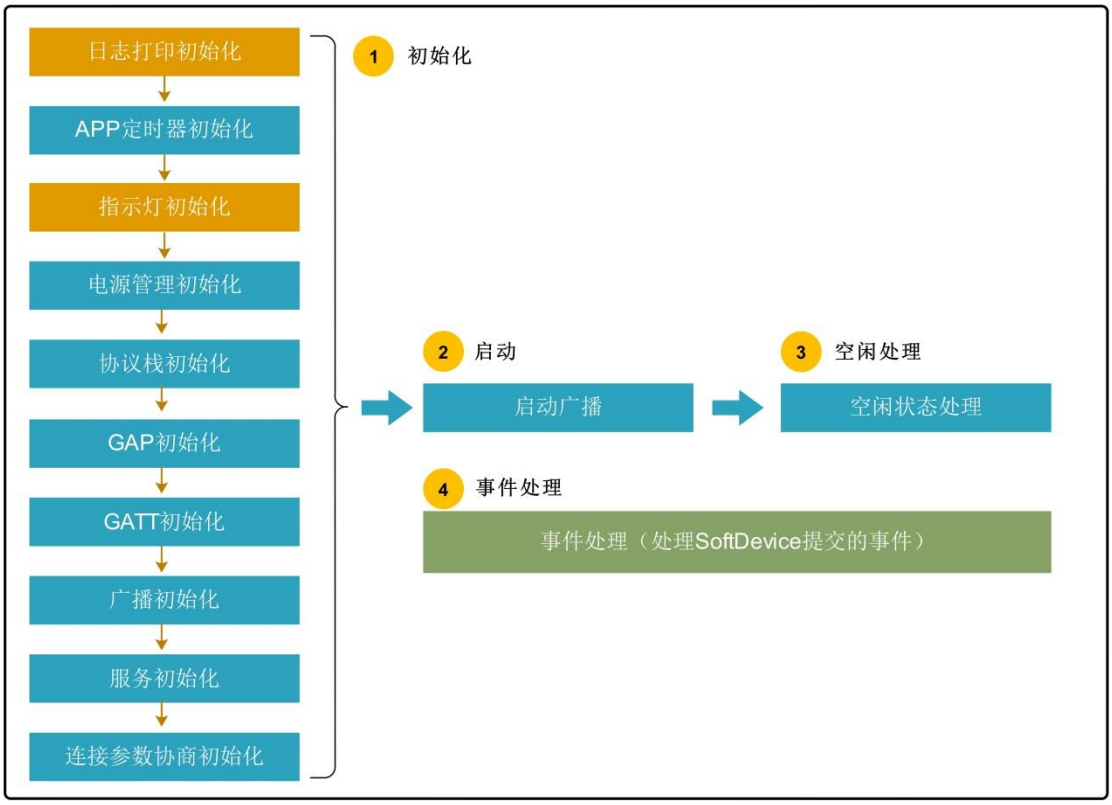


图 3-1：BLE 程序结构

1. 系统初始化

初始化部分用于将系统配置为预设状态，等待启动操作启动系统。初始化包含了十个部分，其中日志打印和指示灯不是必须的。

- 日志打印：用于输出调试信息，日志打印不是 BLE 程序运行必须的，但是它是一个工程“不可或缺”的部件。
- APP 定时器初始化：初始化 APP 定时器，用于实现各种定时任务。尤其要注意 APP 定时器是基于 RTC1 的软件定时器，并不是 BLE 运行的“时钟”。但是因为连接参数协商程序模块使用了 APP 定时器，所以即使应用程序没有用到 APP 定时器，也要执行 APP

定时器初始化。

- 板卡外设初始化：板卡上电后，需要将板卡上的外设初始化到自己想要的状态。板卡外设初始化可以根据具体的需求来操作，比如本例中我们只用到了 LED 指示 BLE 工作状态，那么只需要初始化 LED 就可以了。
- BLE 相关初始化：包含广播、首选连接参数、GAP 层、服务等初始化配置，通过配置将具体的配置参数传递给 SoftDevice，当我们启动协议栈后，整个 BLE 部分的程序就会按照我们设定的参数运行。

2. 启动

对于外围设备来说，启动的是广播，启动之后，系统开始按照配置的广播间隔对外广播数据，等待中心设备连接。对于中心设备来说，启动的是扫描，按照配置的扫描参数扫描周边的设备。

3. 空闲管理

空闲管理的作用主要是为了实现低功耗，BLE 程序运行时，CPU 并不是总是在运行的，空闲管理就是让 CPU 在没有指令执行时进入到低功耗模式等待事件唤醒，从而降低功耗，延长电池使用时间。空闲管理同时还处理挂起的日志打印信息，如果程序中使用了日志打印功能。

4. 事件处理

应用程序通过注册“事件监视者”监视 BLE 协议栈事件，当协议栈有事件产生时，会告知事件监视者，应该程序通过事件监视者的事件处理函数获取和处理事件，如外围设备和中心设备连接后，SoftDevice 会提交“连接事件”，应用程序即可在事件处理函数中接收该事件并执行相应动作。

4. 编写代码

4.1. 日志打印

4.1.1. Log 功能作用

日志打印（Log）模块在嵌入式软件开发过程中极其重要，尤其是在程序调试阶段，更是分析程序流程和程序 Bug 的重要手段。

嵌入式软件中，日志的打印一般会使用自己封装的 Log 程序模块，而不会直接使用 printf 函数输出信息，这么做的原因是：

1. 可以在 Log 中封装数据流的接口，根据需求将信息打印到各种终端。如有 UART 接口的设备，可以通过 UART 打印，有显示屏的设备，可以通过显示屏显示，使用 JLINK 仿真器时通过 JLINK RTT 打印等等。
2. 方便增加打印内容，如在不同类型的日志信息前加上不同的前缀、对于错误信息加上代码的行号以方便定位错误我想在打印。

Log 程序模块一般会包含输出终端选择、日志开关和日志级别设置的功能，这些功能让应用程序很方便地根据自身的需求去使用 Log。

1. 输出终端选择

使用 Log 时，需要定义 Log 的输出终端，对于 nRF52840 的 SDK，可选择的输出终端有 2 种：UART 和 JLINK-RTT，调试时可以根据实际情况选择使用哪种输出终端。输出终端是在“sdk_config.h”文件中定义，如下图所示。

图 3-1：“sdk_config.h”文件中选择输出终端

✧ 注：JLINK RTT（ Real Time Terminal）是 SEGGER 公司推出的可以在嵌入式应用中与用户进行交互的实时终端。嵌入式软件开发过程中，为了提高调试效率，我们一般会通过串口打印出调试信息，但是这会占用设备的串口，这对于串口资源紧张的单片机来说非常不方便，如 nRF51822 和 nRF52832，它们都是只有一个串口，如果串口用于其他用途，那么就没办法再使用串口输出调试信息。这时，我们可以用 JLINK 的 RTT（ Real Time Terminal）来代替物理串口输出调试信息。使用 RTT 时要注意：J-Link 驱动的版本不能低于 V4.90。

2. 日志开关

我们知道嵌入式软件一般会包含很多程序模块，试想一下这样的场景：如果我们调试的时候，某个程序模块运行时出现了问题，为了更好地定位问题，我们需要其它程序模块不打印，只打印这个模块的 Log，或者当我们调试完成后正式发布软件时，需要将 Log 功能全部关闭。这时候，我们希望 Log 具有“开关”的功能，可以根据需要打开/关闭各个程序模块中的 Log。

Log 正是通过加入日志开关解决这种需求的，一般 Log 会有一个总日志开关和功能模块日志开关。总日志开关用于打开/关闭日志打印功能，关闭后，程序中所有 Log 全部关闭，功能模块日志开关用于打开/关闭某个程序模块。因此，使用 Log 时需要打开总日志开关，然后根据需要打开各个程序模块的开关。

nRF52840 的 SDK 中，日志开关是在“sdk_config.h”文件中，如下图所示，我们可以看到一个总日志开关“NRF_LOG_ENABLE”和各个程序模块的日志开关“nrf_log module configuration”，通过这 2 个开关，Log 的打开/关闭以及程序模块中的 Log 的打开/关闭。

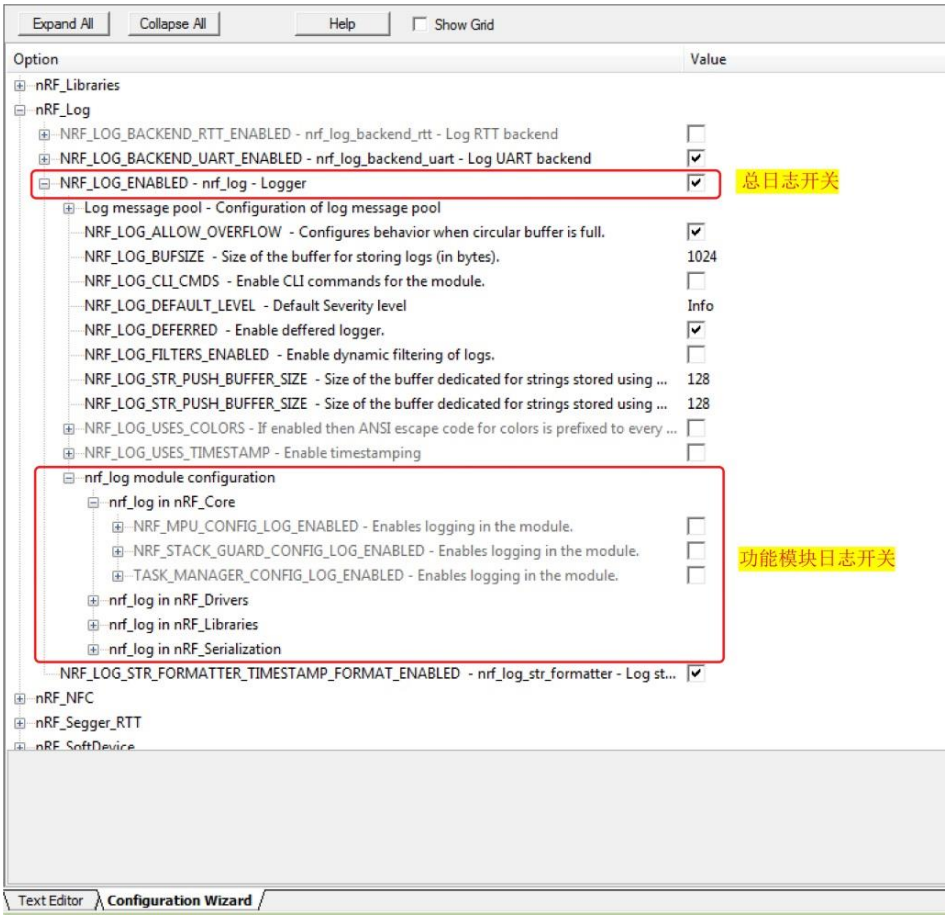


图 3-2：日志开关

3. 日志级别

日志级别对应于日志的重要性程度，我们还是通过场景来说明。当我们调试程序时，如果程序运行出现错误，这时候最重要的是解决错误问题，所以我们希望只打印错误信息，而不打印其它信息，这样更方便我们去定位和处理错误。同样，如果程序中出现警告，我们要做的是跟踪警告信息，这时我们希望 Log 只打印警告信息供我们分析。如果程序正常运行，我们希望观察到一些运行和调试信息，这时候，我们希望 Log 打印的是这些常规信息。

nRF52840 的 SDK 中，日志级别分为 5 级，日志级别同样是在“sdk_config.h”文件中设置的，如下图所示。

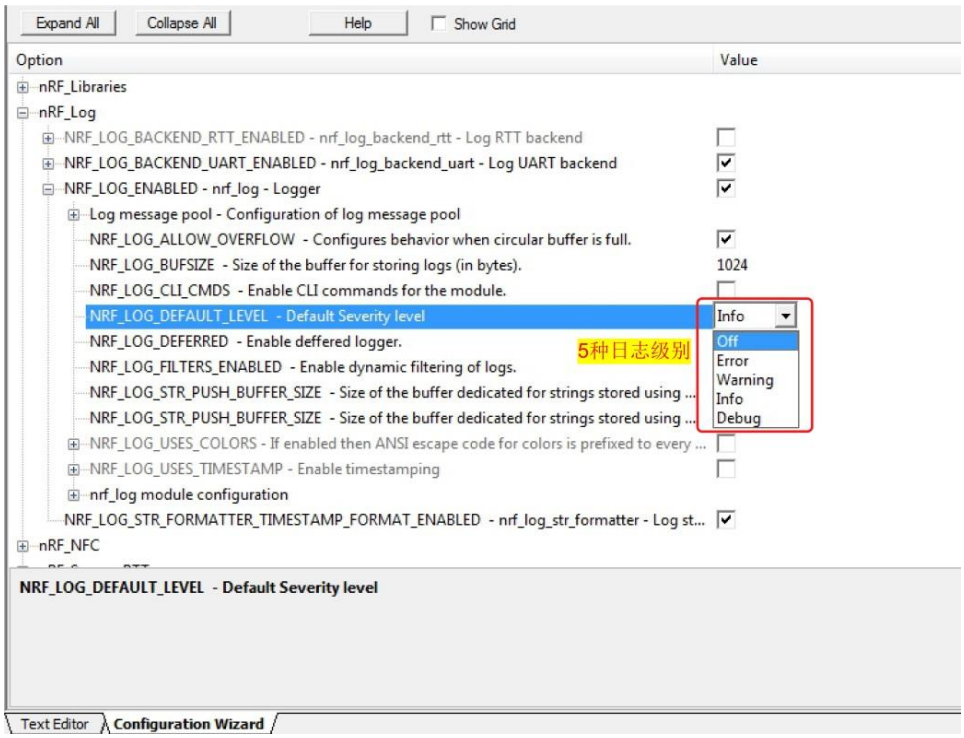


图 3-3：日志级别

上图中的 5 个日志级别作用如下：

1) Off ：关闭日志输出。
2) Error：日志只输出错误信息，该信息对应的 Log 输出函数是“NRF_LOG_ERROR”。
3) Warning：日志只输出警告信息，该信息对应的 Log 输出函数是“NRF_LOG_WARNING”。
4) Info：日志只输出基本信息，该信息对应的 Log 输出函数是“NRF_LOG_INFO”。
5) Debug：日志只输出调试信息，该信息对应的 Log 输出函数是“NRF_LOG_DEBUG”。

4.1.2. 加入 Log

使用 Log 时，首先要引用 Log 的头文件（BLE 工程模板所需的文件和头文件路径包含在第二章中已经完成，本章不再说明这一部分内容），然后初始化 Log 程序模块，并在主循环中加入 Log 信息的输出处理，之后就可以通过 Log 输出函数打印信息。

Log 需要引用 3 个头文件，因为我们需要在“main.c”文件中使用了 Log 程序模块，所以“main.c”文件需要引用下面的头文件。

代码清单： 应用 Log 头文件

```
//日志打印（log）需要引用的头文件
#include "nrf_log.h"
#include "nrf_log_ctrl.h"
#include "nrf_log_default_backends.h"
```

初始化 Log 程序模块时，会根据“sdk_config.h”文件中的配置对 Log 进行初始化。

代码清单： 初始化 Log

```
static void log_init(void)
{
```

```
    //初始化 log 程序模块
    ret_code_t err_code = NRF_LOG_INIT(NULL);
    APP_ERROR_CHECK(err_code);
    //设置 log 输出终端（根据 sdk_config.h 中的配置设置输出终端为 UART 或者 RTT）
    NRF_LOG_DEFAULT_BACKENDS_INIT();
}
```

主循环中加入 Log 信息处理，这样，当 CPU 空闲时即会逐个处理 Log 缓存队列中待输出的日志信息。

代码清单： Log 信息处理

```
if (NRF_LOG_PROCESS() == false)
{
    //其它功能代码
}
```

4.1.3. Log 配置（UART 作为输出终端）

打开“sdk_config.h”文件，切换到“Configuration Wizard”，主要配置项目如下图箭头所指，其他的使用默认的配置即可。

✧ 注：本节对应的试验源码是：“实验 3-1：使用 UART 打印 LOG”。

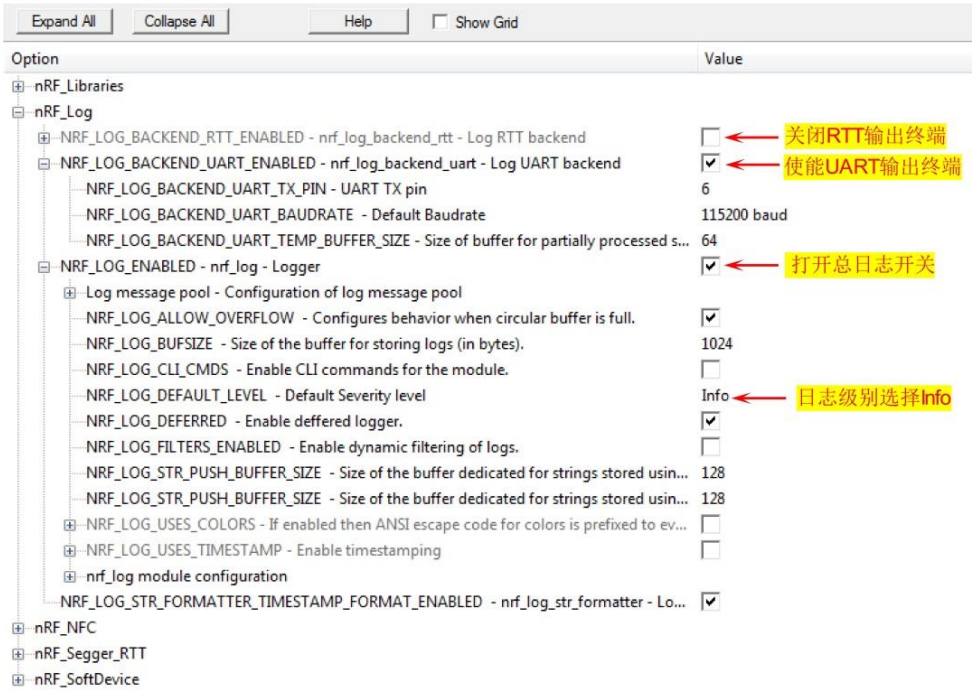


图 3-4：配置 UART 作为 LOG 输出终端

修改好配置后，编译工程并下载到开发板运行，计算机上打开串口调试助手，按动开发板上的复位按键，可以观察到，程序每次复位启动时会输出“BLE Template example:Log use uart as output terminal”。

4.1.4. Log 配置（RTT 作为输出终端）

打开“sdk_config.h”文件，切换到“Configuration Wizard”，主要配置项目如下图箭头所指，其他的使用默认的配置即可。

✧ 注：本节对应的试验源码是：“实验 3-2：使用 JLINK-RTT 打印 LOG”。

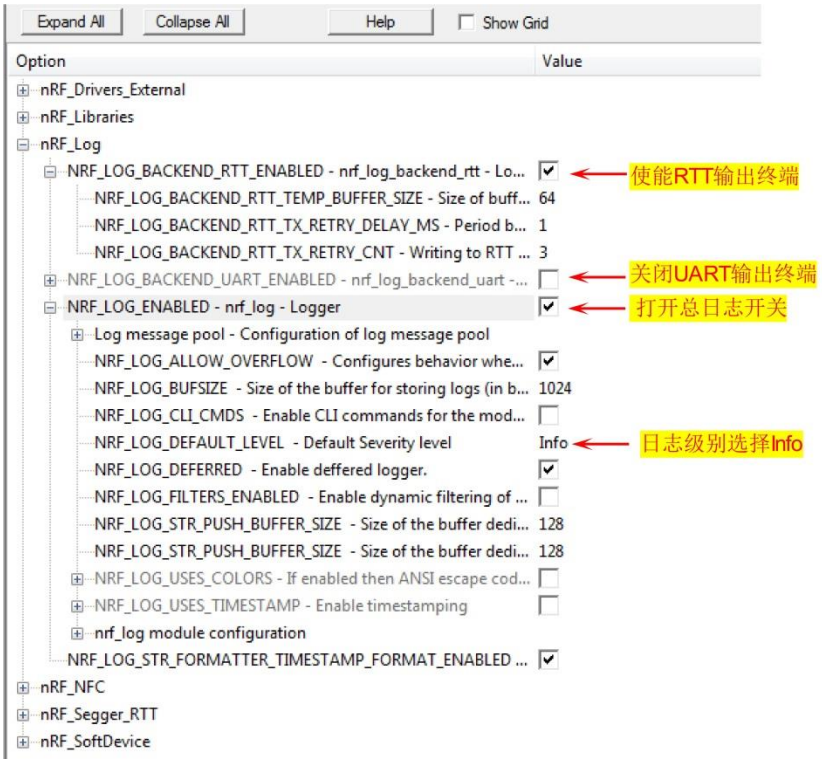


图 3-5：配置 JLINK-RTT 作为 LOG 输出终端

修改好配置后，编译工程并下载到开发板运行。之后还需要配置 JLINK-RTT，配置好之后，才能接收 LOG 打印的信息，步骤如下。

1. 启动 JLINK-RTT Viewer

安装了 JLINK 驱动后，可以从开始菜单或者桌面快捷方式启动 JLINK-RTT Viewer，如下图所示。

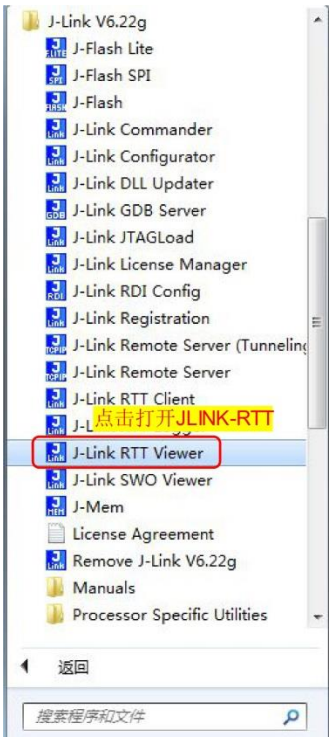


图 3-6：打开 JLINK-RTT Viewer

2. 配置 JLINK-RTT Viewer

JLINK-RTT Viewer 启动后，会弹出配置窗口，按照下图所示配置。

- 连接 JLINK：选择 USB。
- 目标设备：因为开发板使用的芯片是 nRF52840-QIAA，所以选择“nRF52840-xxAA”。
- 调试接口：因为 nRF52840 只支持 SWD，所以必须选择 SWD。

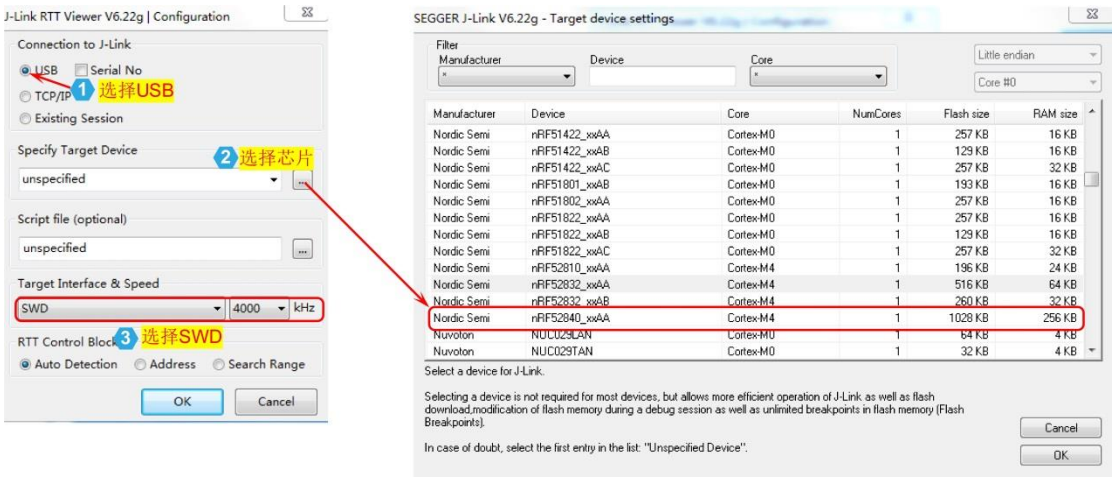


图 3-7：配置 JLINK-RTT Viewer

3. 信息显示

配置完成后，点击“确定”按钮即进入 JLINK-RTT Viewer 界面，如下图所示：

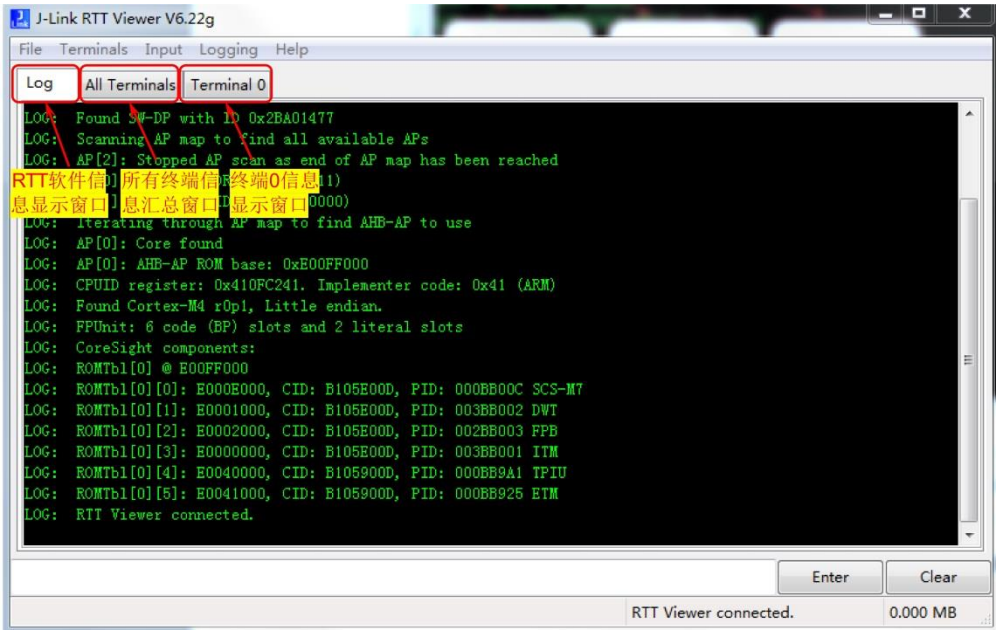


图 3-8：JLINK-RTT Viewer 界面

开发板下载了“实验 3-2”的程序后，每次启动运行时会输出“BLE Template example:Log use JLINK-RTT as output terminal”，这时我们在“Terminal0” 窗口可以观察到该信息，同时，因为“ALL Terminals” 窗口是所有“Terminals”信息的汇总，所以在“ALL Terminals”窗口也可以看到该信息。

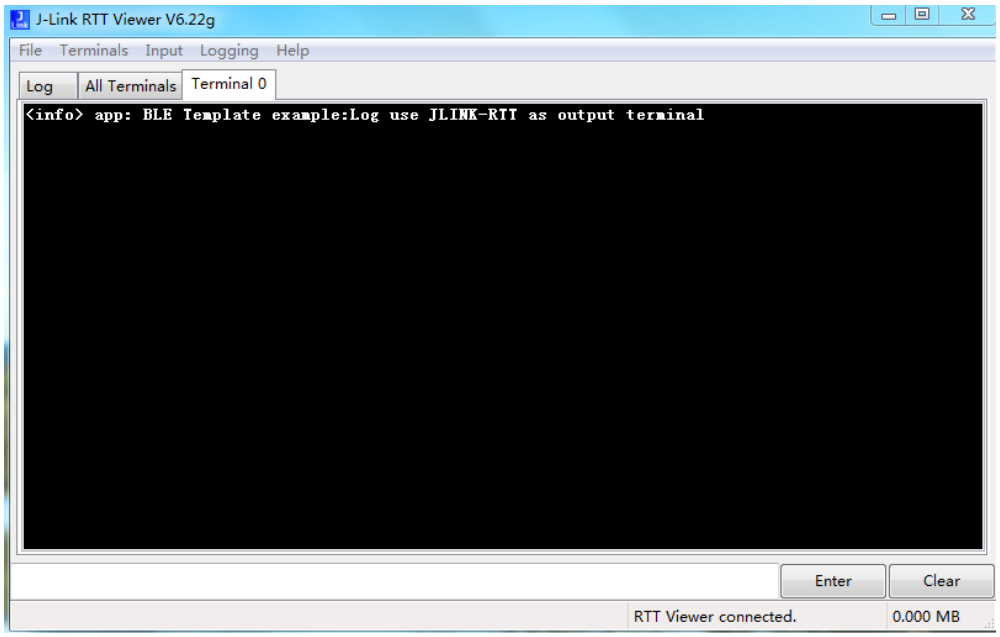


图 3-9：日志显示

4. JLINK-RTT Viewer 常用操作

■ 清除终端显示信息：在需要清空显示信息的终端名字上点击右键，然后在弹出的下拉菜单中点击“Clear terminal”，即可清空该终端显示的信息。

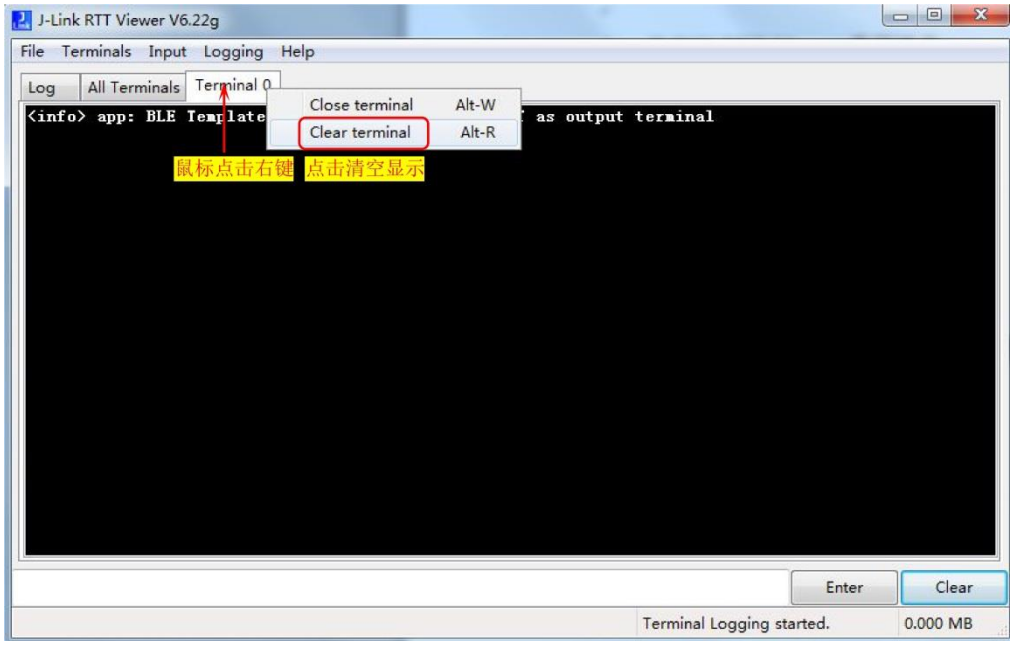


图 3-10：清空终端显示的信息

■ 保存日志

有时候，我们希望将 JLINK-RTT Viewer 的日志信息保存到计算机，以便于分析日志信息，这时候可以使用 JLINK-RTT Viewer 的日志保存功能，步骤如下：

- 执行“Logging→Start Terminal Logging”，如下图所示。

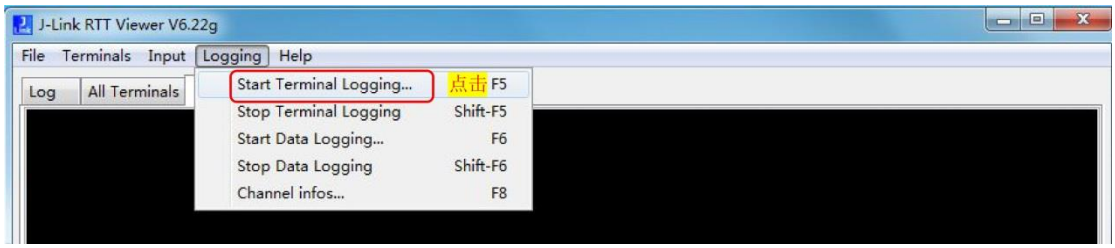


图 3-11：保存日志

- 在弹出的窗口中设置好保存路径和文件名称后，点击“保存”按钮，之后，日志信息即会被保存到该文件，如下图所示。

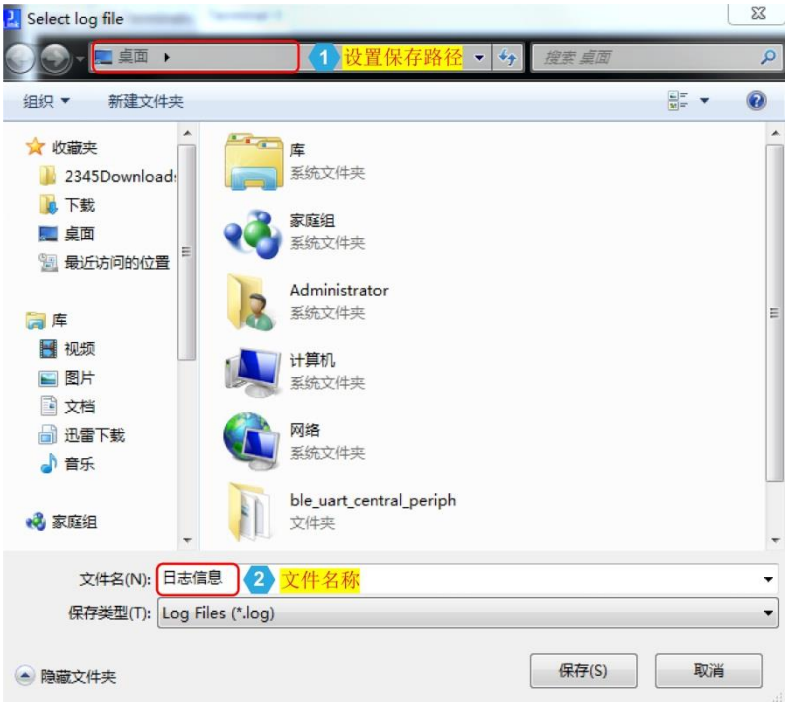


图 3-12：保存日志

- 查到保存的日志文件时，可以用记事本打开保存的日志文件，如下图所示。

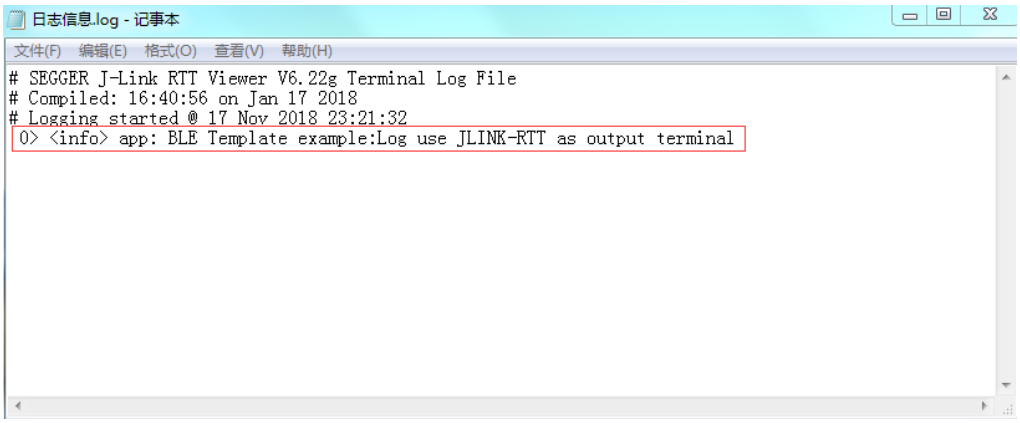


图 3-13：打开保存的日志

4.2. 初始化 APP 定时器

4.2.1. APP 定时器作用

APP 定时器基于实时计数器 RTC1 的软件定时器，APP 定时器允许用户同时创建多个定时任务，APP 定时器在 RTC1 中断处理程序中检查超时并执行超时处理程序的调用，在软中断（SWI0）处理定时列表，RTC1 中断和 SWI0 均使用低中断优先级 APP_LOW。

在 BLE 的程序中，APP 定时器的用途主要有以下几种：

1. 按键消抖以及实现长按和短按。
2. 定时驱动 LED 指示灯，和 BLE 事件配合实现 BLE 的状态指示（如指示灯 D1 闪烁指示正在广播，常亮指示已经连接）。
3. 连接参数更新，连接成功后，开启 APP 定时器，超时后进行连接参数更新。
4. 用户创建定时任务，实现各种定时任务，如创建一个超时时间为 1 秒的 APP 定时器作

为实时时钟的时基。

因为 BLE 中的连接参数更新使用了 APP 定时器，所以，即使 BLE 程序中不用按键、指示灯，也不创建用户定时任务，APP 定时器也需要初始化。

4.2.2. 加入 APP 定时器初始化代码

使用 APP 定时器时，需要引用头文件“app_timer.h”，然后，初始化 APP 定时器模块，之后即可创建用户定时任务，实现定时。BLE 工程模板中我们没有创建用户定时任务，所以只需初始化 APP 定时器即可。

因为我们需要在“main.c”文件中使用了 APP 定时器程序模块，所以“main.c”文件需要引用下面的头文件。

```
//APP 定时器需要引用的头文件
#include "app_timer.h"
```

APP 定时器初始化函数代码如下，如果我们创建用户定时任务，也可以放到这个函数里面，这样方便管理代码。

代码清单：初始化 APP 定时器模块

```
//初始化 APP 定时器模块
static void timers_init(void)
{
    //初始化 APP 定时器模块
    ret_code_t err_code = app_timer_init();
      //检查返回值
    APP_ERROR_CHECK(err_code);

    //加入创建用户定时任务的代码，创建用户定时任务。
}
```

4.2.3. 本节留下的问题

1. 用户如果需要定时，如何使用该定时器？

   如何使用 APP 定时器创建定时任务实现定时，参阅《第 5 章：创建定时任务》。

2. 按键是如何使用 APP 定时器实现消抖以及短按和长按检测的？

   按键的消抖以及短按和长按，参阅《第 8 章：按键应用》。

4.3. 指示灯初始化

4.3.1. 指示灯作用

指示灯并不是必须的，需要针对具体的硬件板卡进行操作。本节描述的是针对开发板的硬件来说的，开发板上设计了 4 个 LED 指示灯，如下表所示。

表 3-2：LED 和按键引脚分配

LED	引脚	说明
指示灯 D1	P0.13	独立 IO
指示灯 D2	P0.14	独立 IO
指示灯 D3	P0.15	独立 IO
指示灯 D4	P0.16	独立 IO

✧ 注：独立 IO 表示开发板没有其他的电路使用这个 IO。

BLE 工程模板中使用了 LED 指示灯 D1 指示 BLE 工作状态，所以我们需要初始化指示灯，即设置驱动 LED 的引脚为输出，并使其输出高电平，即设置 LED 初始状态为熄灭。

初始化和驱动指示灯有两种方式：

- 使用 GPIO 函数直接初始化和驱动。
- 使用 BSP 函数初始化和驱动。Nordic 的 SDK 中提供了板级支持包 BSP(Board support package)，他实现了板卡硬件外设和协议栈之间的交互(主要是按键和指示灯)，如按下按键唤醒系统、用指示灯指示广播、扫描、连接等状态。

对于使用何种方式驱动指示灯，一般地如果在程序中只用到点亮和熄灭，可以使用 GPIO 函数直接驱动或者使用 BSP 函数，如果需要用到指示灯闪烁，使用 BSP 函数会更方便，因为 BSP 初始化指示灯时会创建 APP 定时器，可以通过 APP 定时器驱动指示灯以一定间隔闪烁。BLE 工作模板中通过指示灯 D1 闪烁指示 BLE 广播，所以这里我们使用 BSP 函数初始化 LED。

4.3.2. 编写指示灯初始化代码

初始化 LED 时，调用库函数 bsp_init()，函数参数使用 BSP_INIT_LEDS 即可初始化开发板上的 4 个 LED 指示灯，代码如下所示。

代码清单：初始化和指示灯

```
static void leds_init(void)
{
    ret_code_t err_code;
    //初始化 BSP 指示灯
    err_code = bsp_init(BSP_INIT_LEDS, NULL);
    //检查函数返回的错误代码
    APP_ERROR_CHECK(err_code);
}
```

4.3.3. 本节留下的问题

1. 使用 BSP 函数如何初始化和使用按键？

   参阅《第 8 章：按键应用》。

4.4. 初始化和运行电源管理

SoftDevice 实现了一个简单易用的 SoftDevice Power API，用于优化电源管理。启用 SoftDevice 时，应用程序必须使用此 API 以确保实现正确的功能。

电源管理运行后，在使用 API 等待应用程序事件时，只要 SoftDevice 不使用 CPU，CPU 就会进入 IDLE 状态，此时，SoftDevice 直接处理的中断不会唤醒应用程序。应用程序中断将按预期唤醒应用程序。另外，当系统进入 System Off 模式时，使用电源管理模块的 API 可以确保在断电之前停止 SoftDevice 服务。

使用电源管理功能时，首先要初始化电源管理模块，之后在主循环中运行电源管理，这样，当 CPU 空闲时会执行电源管理 API，进入低功耗模式等待事件唤醒，从而实现低功耗。

1. 初始化电源管理模块

电源管理模块通过调用 nrf_pwr_mgmt_init()函数初始化，该函数原型如下表所示。

表 3-3：nrf_pwr_mgmt_init()函数

函数原型	ret_code_t nrf_pwr_mgmt_init (void)
函数功能	初始化电源管理模块。 根据配置，此功能在系统控制块（SCB）中设置 SEVONPEND，SoftDevice 的中断优先级高于 SVC，此操作不安全
参　　数	无。
返回值	NRF_SUCCESS：启动广播成功。 NRF_ERROR_INVALID_STATE：广播程序模块没有初始化。

BLE 工程模板中，电源管理初始化相关的代码封装在 power_management_init()函数中，该函数中会调用 nrf_pwr_mgmt_init()函数完成电源管理初始化，如下所示。

代码清单：电源管理初始化

```
static void power_management_init(void)
{
    ret_code_t err_code;
    //初始化电源管理
    err_code = nrf_pwr_mgmt_init();
    //检查函数返回的错误代码
    APP_ERROR_CHECK(err_code);
}
```

2. 运行电源管理

运行电源管理是通过调用 nrf_pwr_mgmt_run()函数执行的，该函数必须放到主循环里面执行，以确保 CPU 空闲时即可运行电源管理。

表 3-4：nrf_pwr_mgmt_run()函数

函数原型	void nrf_pwr_mgmt_run(void)

函数功能	运行电源管理，该函数需要放到主循环里面执行。
参　　数	无。
返回值	无。

BLE 工程模板主循环中的 idle_state_handle()函数封装了 LOG 处理和运行电源管理的执行函数，该函数检查是否有挂起的日志操作，如没有，则睡眠直到下一个事件发生后唤醒系统。

代码清单：运行电源管理

```
//空闲状态处理函数。如果没有挂起的日志操作，则睡眠直到下一个事件发生后唤醒系统
static void idle_state_handle(void)
{
    //处理挂起的 log
    if (NRF_LOG_PROCESS() == false)
    {
        //运行电源管理，该函数需要放到主循环里面执行
        nrf_pwr_mgmt_run();
    }
}
```

4.5. BLE 协议栈初始化

4.5.1. 初始化步骤

使用 BLE 功能之前，需要先初始化 BLE 协议栈，BLE 协议栈初始化包括使能 SoftDevice、配置 BLE 协议栈参数、使能 BLE 协议栈和注册 BLE 协议栈事件回调函数。

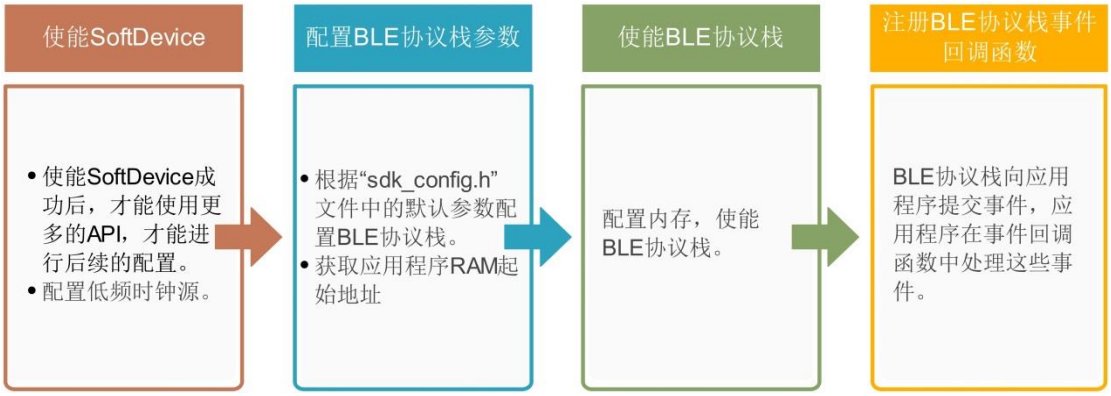


图 3-14：BLE 协议栈初始化流程

BLE 工程模板中，BLE 协议栈初始化代码封装在“ble_stack_init()”函数中，代码清单如下：

代码清单：初始化 BLE 协议栈

```
static void ble_stack_init(void)
```

```
{
    ret_code_t err_code;
    //请求使能 SoftDevice，该函数会根据 sdk_config.h 文件中低频时钟的设置来配置低频时钟
    err_code = nrf_sdh_enable_request();
    APP_ERROR_CHECK(err_code);

    //定义保存应用程序 RAM 起始地址的变量
    uint32_t ram_start = 0;
    //使用 sdk_config.h 文件的默认参数配置协议栈，获取应用程序 RAM 起始地址，
    //保存到变量 ram_start
    err_code = nrf_sdh_ble_default_cfg_set(APP_BLE_CONN_CFG_TAG, &ram_start);
    APP_ERROR_CHECK(err_code);

    //使能 BLE 协议栈
    err_code = nrf_sdh_ble_enable(&ram_start);
    APP_ERROR_CHECK(err_code);

    //注册 BLE 事件回调函数
    NRF_SDH_BLE_OBSERVER(m_ble_observer, APP_BLE_OBSERVER_PRIO, ble_evt_handler,
                         NULL);
}
```

1. 使能 SoftDevice

从上面的代码中，我们可以看到，初始化 BLE 协议栈时，首先必须使能 SoftDevice，使能 SoftDevice 时用的库函数是 `nrf_sdh_enable_request()`，该函数在使能 SoftDevice 时同时根据 sdk_config.h 文件中低频时钟的设置来配置低频时钟（配置方法见 2.5.2 节），函数原型如下。

表 3-5：`nrf_sdh_enable_request()`函数

函数原型	ret_code_t　nrf_sdh_enable_request (void)
函数功能	请求使能 SoftDevice。 　　该函数向所有使用 NRF_SDH_REQUEST_OBSERVER 宏注册的所有观察者发布 NRF_SDH_EVT_ENABLE_REQUEST 请求，观察者可能会确认/也可能不确认请求，如果所有的观察者都确认，则 SoftDevice 使能成功，否则，进程将被停止，而未确认的观察者如果准备好后会负责通过调用“nrf_sdh_request_continue”函数重启进程。
参　　数	无。
返回值	NRF_SUCCESS：初始化成功。 NRF_ERROR_INVALID_STATE：协议栈已经使能。

2. 配置 BLE 协议栈的参数

SoftDevice 使能后，接下来需要配置 BLE 协议栈的参数，这是通过库函数 nrf_sdh_ble_default_cfg_set()完成的，该函数一方面使用 sdk_config.h 文件中设置的默认参数配置协议栈，另一方面获取 MDK 中设置的应用程序 RAM 起始地址，并通过函数的参数 ram_start 返回应用程序 RAM 起始地址，后面使能 BLE 协议栈时会用到该地址。

表 3-6：nrf_sdh_ble_default_cfg_set()函数

函数原型	ret_code_t nrf_sdh_ble_default_cfg_set(uint8_t conn_cfg_tag, uint32_t * p_ram_start)
函数功能	设置默认的 BLE 协议栈配置。 。
参　　数	[in] conn_cfg_tag：配置标记。 [in] p_ram_star：应用程序 RAM 起始地址。
返回值	NRF_SUCCESS：初始化成功。 NRF_ERROR_INVALID_STATE：协议栈已经使能。

3. 使能 BLE 协议栈

BLE 协议栈参数配置完成后，即可通过 nrf_sdh_ble_enable()库函数使能 BLE 协议栈和分配内容，该函数会对 RAM 起始地址和大小进行检查，并通过 log 打印出检查结果。

表 3-7：nrf_sdh_ble_enable()函数

函数原型	ret_code_t nrf_sdh_ble_enable(uint32_t * p_app_ram_start)
函数功能	配置和使能 BLE 协议栈。
参　　数	[in] p_app_ram_start：应用程序 RAM 起始地址。
返回值	NRF_SUCCESS：BLE 协议栈初始化成功。 NRF_ERROR_INVALID_STATE：BLE 协议栈已经初始化，不能再次进行初始化。 NRF_ERROR_INVALID_ADDR：提供无效的或未对齐的指针。 NRF_ERROR_NO_MEM：内存不足。

4. 注册事件监视者

程序运行过程中，应用程序需要实时获取自己感兴趣的协议栈的事件以进行相应的处理，也就是应用程序需要“监视”协议栈的事件，所以在使能 BLE 协议栈后，需要注册事件监视者，当协议栈有事件产生时，会告知事件监视者，事件监视者则会采取相应的行动。以蓝牙连接断开事件为例，其执行流程如下：

- 应用程序注册 BLE 协议栈事件监视者。

- 协议栈检测到蓝牙连接断开（BLE_GAP_EVT_DISCONNECTED 事件）。
- 协议栈告知事件监视者蓝牙连接断开。
- 事件监视者处理蓝牙断开事件（事件处理函数中处理蓝牙断开事件）。

事件监视者使用宏 NRF_SDH_BLE_OBSERVER 注册，该宏描述如下，共有 4 个参数，其中“_prio”是固定的数值，应用程序注册事件监视者时该参数的值是 3，应用程序不能修改该数值。

表 3-8：NRF_SDH_BLE_OBSERVER 宏

宏	#define NRF_SDH_BLE_OBSERVER (_name, _prio, _handler, _context)
功能描述	该宏用于注册事件监视者，程序模块如果想接收事件通知，就必须使用该宏注册事件处理程序。
参　　数	[in] _name：事件监视者名称。 [in] _prio：事件监视者事件处理程序的优先级，数字越小，优先级越高。 [in] _handler：BLE 事件句柄。 [in] _context：事件处理函数参数。

注册事件监视者时应用程序需要提供事件处理函数，用于接收和处理 BLE 事件，BLE 事件处理函数编写格式如下，函数名称必须和注册的名称必须一样。

代码清单：BLE 事件处理函数格式

```
static void ble_evt_handler(ble_evt_t const * p_ble_evt, void * p_context)
{
    ret_code_t err_code = NRF_SUCCESS;
    //判断 BLE 事件类型，根据事件类型执行相应操作
    switch (p_ble_evt->header.evt_id)
    {
        //断开连接事件
        case BLE_GAP_EVT_DISCONNECTED:
            //打印提示信息
             NRF_LOG_INFO("Disconnected.");
            break;

        //添加处理其它 BLE 事件的代码

```

```
        default:
            break;
    }
}
```

对于 BLE 工程模板，只需要处理连接事件、连接断开事件、PHY 更新事件和超时事件这几个基本的 BLE 事件即可，代码清单如下。

代码清单：BLE 事件处理函数格式

```
static void ble_evt_handler(ble_evt_t const * p_ble_evt, void * p_context)
{
    ret_code_t err_code = NRF_SUCCESS;
    //判断 BLE 事件类型，根据事件类型执行相应操作
    switch (p_ble_evt->header.evt_id)
    {
        //断开连接事件
        case BLE_GAP_EVT_DISCONNECTED:
            //打印提示信息
            NRF_LOG_INFO("Disconnected.");
            break;

        //连接事件
        case BLE_GAP_EVT_CONNECTED:
            NRF_LOG_INFO("Connected.");
            //设置指示灯状态为连接状态，即指示灯 D1 常亮
            err_code = bsp_indication_set(BSP_INDICATE_CONNECTED);
            APP_ERROR_CHECK(err_code);
            //保存连接句柄
            m_conn_handle = p_ble_evt->evt.gap_evt.conn_handle;
            //将连接句柄分配给排队写入实例，分配后排队写入实例和该连接关联，这样，当有
            //多个连接的时候，通过关联不同的排队写入实例，很方便单独处理各个连接
            err_code = nrf_ble_qwr_conn_handle_assign(&m_qwr, m_conn_handle);
            APP_ERROR_CHECK(err_code);
            break;

        //PHY 更新事件
        case BLE_GAP_EVT_PHY_UPDATE_REQUEST:
        {
            NRF_LOG_DEBUG("PHY update request.");
            ble_gap_phys_t const phys =
            {
                .rx_phys = BLE_GAP_PHY_AUTO,
                .tx_phys = BLE_GAP_PHY_AUTO,
```

```
            };
            //响应 PHY 更新规程
            err_code = sd_ble_gap_phy_update(p_ble_evt->evt.gap_evt.conn_handl
                       e, &phys);
            APP_ERROR_CHECK(err_code);
        } break;

        //GATT 客户端超时事件
        case BLE_GATTC_EVT_TIMEOUT:
            NRF_LOG_DEBUG("GATT Client Timeout.");
            //断开当前连接
            err_code = sd_ble_gap_disconnect(p_ble_evt->evt.gattc_evt.conn_han
                       dle,
            BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
            APP_ERROR_CHECK(err_code);
            break;

        //GATT 服务器超时事件
        case BLE_GATTS_EVT_TIMEOUT:
            NRF_LOG_DEBUG("GATT Server Timeout.");
            //断开当前连接
            err_code = sd_ble_gap_disconnect(p_ble_evt->evt.gatts_evt.conn_han
                       dle,
            BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
            APP_ERROR_CHECK(err_code);
            break;

        default:
            break;
    }
}
```

在 BLE 协议栈初始化的过程中，和开发者关系比较大的有 2 个方面：配置低频时钟源和分配内存，低频时钟源涉及到硬件电路的设计，假如我们的硬件电路没有设计外部 32.768KHz 晶体的话，就必须要配置使用内部 RC 作为低频时钟源，因此我们要掌握配置低频时钟源的方法。分配内存涉及到协议栈和应用程序对片内 RAM 的占用，BLE 程序中片内 RAM 是分为两部分的，一部分给 SoftDevice 使用，另外一部分给应用程序使用，为了合理使用内存，我们需要在使能 BLE 协议栈时调整内存的分配，接下来我们着重看一下这两部分是如何操作的。

4.5.2. 配置低频时钟

SoftDevice 开放了低频时钟的配置接口，允许上层应用程序配置低频时钟源（LFCLK 时钟源），应用程序也必须为 SoftDevice 配置低频时钟源，可配置的低频时钟源有 2 种：

1) 片内 32.768KHz RC 振荡源。
2) 片外 32.768KHz 晶体。

它们的区别如下：

- 使用内部 RC 振荡源：缺点是精度稍差，优点是不会增加硬件成本，同时不会占用 PCB 空间，设备可以设计的更小巧。
- 使用外部 32.768KHz 晶振：优点是精度高，缺点是会增加设备成本和占用 PCB 空间。

低频时钟是在 nrf_sdh_enable_request()函数中配置的，进入该函数，可以看到配置的代码如下，定义了一个时钟配置结构体 clock_lf_cfg 并对其变量指定初始化，初始化时使用的宏是在“sdk_config.h”文件中定义的，之后调用 API 函数 sd_softdevice_enable()使能 SoftDevice 时，使用该结构体作为 sd_softdevice_enable()函数的参数，由该 API 函数完成低频时钟配置。

代码清单：nrf_sdh_enable_request()函数中低频时钟配置代码

```
//低频时钟配置
nrf_clock_lf_cfg_t const clock_lf_cfg =
{
    .source       = NRF_SDH_CLOCK_LF_SRC,//时钟源
    .rc_ctiv      = NRF_SDH_CLOCK_LF_RC_CTIV,//SoftDevice 校准定时器间隔
    //SoftDevice 在恒温下校准定时器的间隔，即温度不变的情况下校准定时器的间隔
    .rc_temp_ctiv = NRF_SDH_CLOCK_LF_RC_TEMP_CTIV,
    .accuracy     = NRF_SDH_CLOCK_LF_ACCURACY//外部时钟的精度
};
```

上述代码中的 4 个用于初始化宏时钟配置结构体成员变量的宏的定义在“sdk_config.h”文件中，通过修改这 4 个宏的值即可设置低频时钟源，默认配置使用的是外部 32.768KHz 晶体，这 4 个宏的作用如下：

1) **NRF_SDH_CLOCK_LF_SRC：**低频时钟源，可设置为外部晶体或内部 RC
   - NRF_CLOCK_LF_SRC_RC：使用内部 RC 作为低频时钟源。
   - NRF_CLOCK_LF_SRC_XTAL：使用外部 32.768KHz 晶体作为低频时钟源。
2) **NRF_SDH_CLOCK_LF_RC_CTIV：**校准间隔，低频时钟源设置为使用内部 RC 时推荐值为 16，低频时钟源设置为使用外部晶体时该值为 0。
3) **NRF_SDH_CLOCK_LF_RC_TEMP_CTIV：**SoftDevice 在恒温下校准定时器的间隔，即温度不变的情况下校准定时器的间隔，低频时钟源设置为使用内部 RC 时推荐值为 2，低频时钟源设置为使用外部晶体时该值为 0。
4) **NRF_SDH_CLOCK_LF_ACCURACY：**外部时钟精度，可设置的值如下：

   NRF_CLOCK_LF_ACCURACY_250_PPM

   NRF_CLOCK_LF_ACCURACY_500_PPM

   NRF_CLOCK_LF_ACCURACY_150_PPM

   NRF_CLOCK_LF_ACCURACY_100_PPM

NRF_CLOCK_LF_ACCURACY_75_PPM

NRF_CLOCK_LF_ACCURACY_50_PPM

NRF_CLOCK_LF_ACCURACY_30_PPM

NRF_CLOCK_LF_ACCURACY_20_PPM

NRF_CLOCK_LF_ACCURACY_10_PPM

NRF_CLOCK_LF_ACCURACY_5_PPM

NRF_CLOCK_LF_ACCURACY_2_PPM

NRF_CLOCK_LF_ACCURACY_1_PPM

低频时钟源设置为使用内部 RC 时设置为 NRF_CLOCK_LF_ACCURACY_500_PPM，低频时钟源设置为使用外部晶体时，根据外部晶体的实际参数从上述可设置值中选择对应的值。

1. 使用外部 32.768KHz 晶体作为低频时钟源时“sdk_config.h”文件中的配置如下：

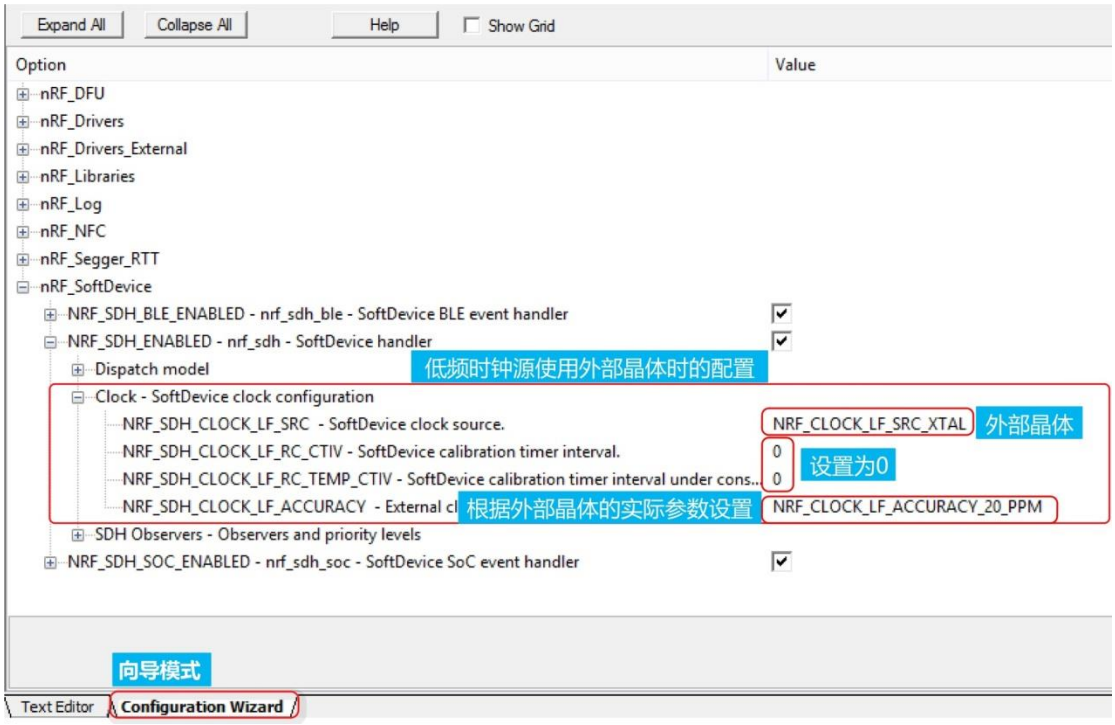


图 3-15：使用外部晶体作为低频时钟源

2. 使用内部 RC 作为低频时钟源时“sdk_config.h”文件中的配置如下：

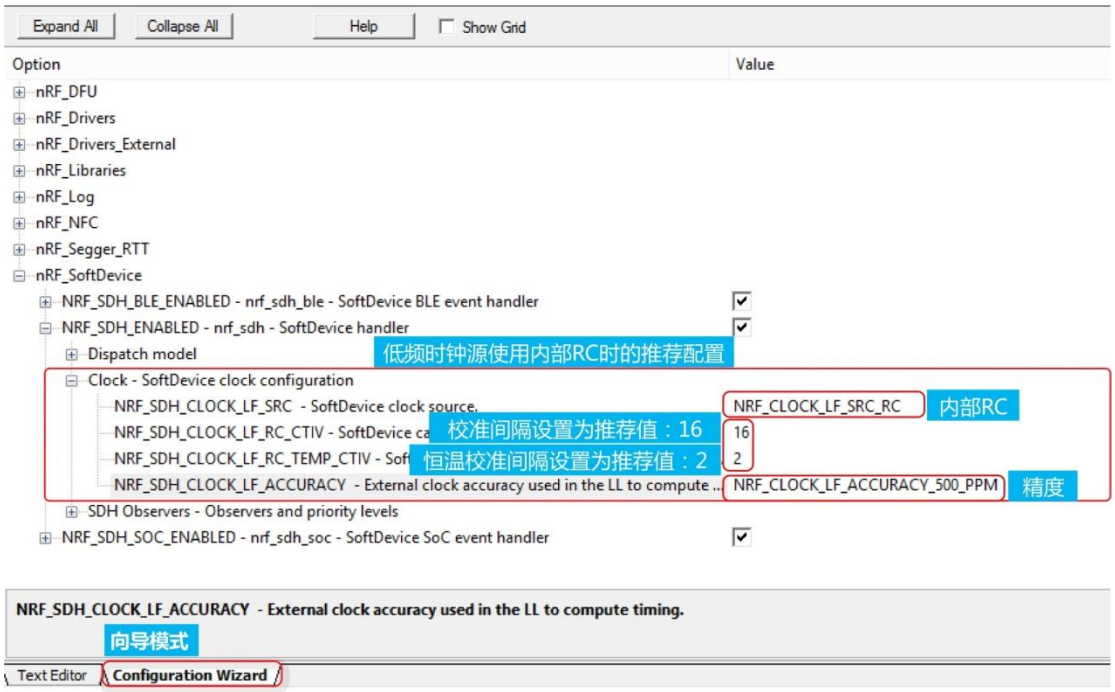


图 3-16：使用内部 RC 作为低频时钟源

4.5.3. BLE 协议栈使能和内存配置

BLE 程序中，RAM 是分为两部分的，一部分保留给 SoftDevice 使用，另外一部分给应用程序使用。RAM 分配是在 MDK 的工程配置中进行的，BLE 工程模板中 RAM 分配如下图所示，通过设置 RAM 起始地址和大小确定分配给应用程序的 RAM 空间以及保留给 SoftDevice 的 RAM 空间。我们知道 nRF52840 的片内 RAM 起始地址是 0x20000000，下图中的设置即将 0x20000000~0x2000225F 这一 RAM 空间分配给了 SoftDevice，从 0x20002260 开始的大小为 0x3DDA0 的 RAM 空间分配给了应用程序。

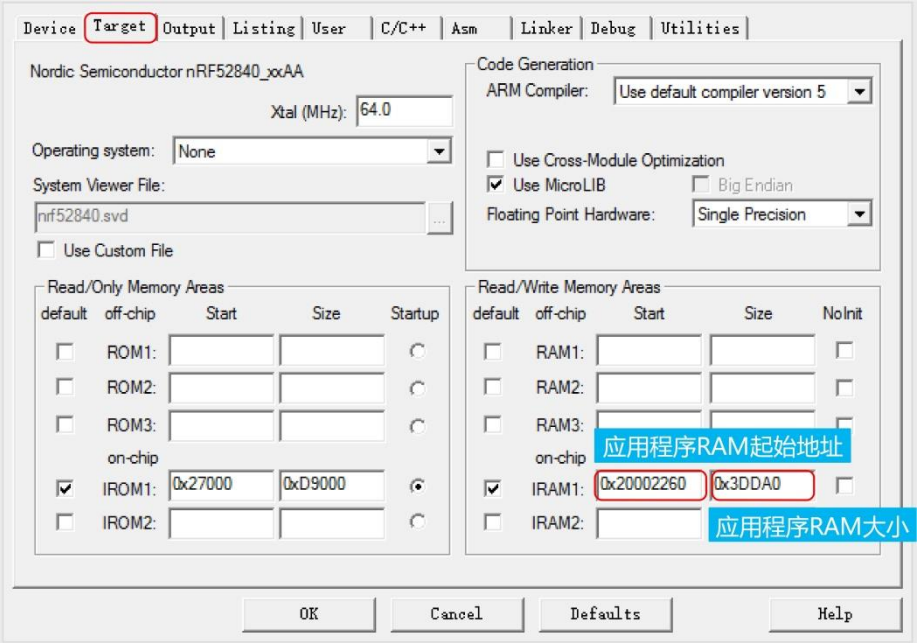


图 3-17：BLE 工程模板 RAM 分配

BLE 工程模板相当于 BLE 的“最小系统”，所以它保留给 SoftDevice 的 RAM 空间是最小的，也就是 BLE 工程模板保留给 SoftDevice 的 RAM 空间是 SoftDevice 对 RAM 的最小需求。当我们在 BLE 工程模板中加入其他功能的时候，如加入服务、配对绑定、自定义 UUID 等等，或者是一个功能比较多的 BLE 程序中去掉某些功能的时候，协议栈和应用程序对 RAM 的占用通常会改变，这时候，可能会出现下面三种情况：

1. RAM 起始地址设置过小，程序无法运行。
2. RAM 起始地址设置过大，保留给协议栈的 RAM 空间足够，程序正常运行，我们希望调整内存分配，以节省 RAM，从而能为应用程序分配更多的 RAM。
3. RAM 起始地址设置过大，保留给协议栈的 RAM 空间足够，程序正常运行，我们不关心 RAM 是否浪费。

分配 RAM 时的三种情况对应的流程如下图所示。

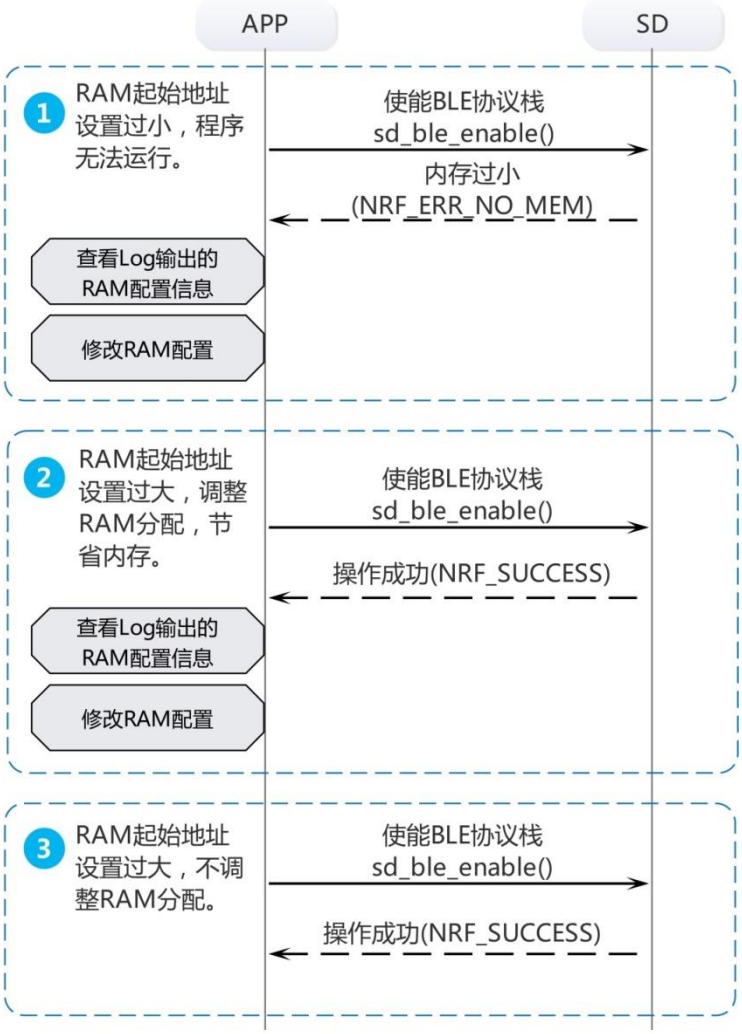


图 3-18：分配 RAM 时的流程

由上文可以看到，对于 RAM 分配，我们需要的是通过 Log 获取系统输出的 RAM 分配信息，根据该信息去修改 MDK 中 RAM 的配置。为了获取该提示信息，我们需要做到以下两点。

1. 为 SoftDevice 保留的 RAM 空间能让 SoftDevice 运行，即 RAM 的起始地址设置的不小

于 BLE 工程模板中的设置，这样即使程序不能正常运行，但是可以运行到通过 Log 输出 RAM 分配提示信息。

2. 开启 Log，从而获取系统输出的 RAM 提示信息（如果使用 Log 参考 2.1 节）。

接下来，我们通过实例来练习一下分配 RAM 时的三种情况的处理方法，

1. 第一种情况：内存分配过小，程序无法正常运行。

为了演示 RAM 分配过小的情况，人为将分配给协议栈的 RAM 改小，如下图所示，同时，开启 Log（输出终端选择 JLINK-RTT）。

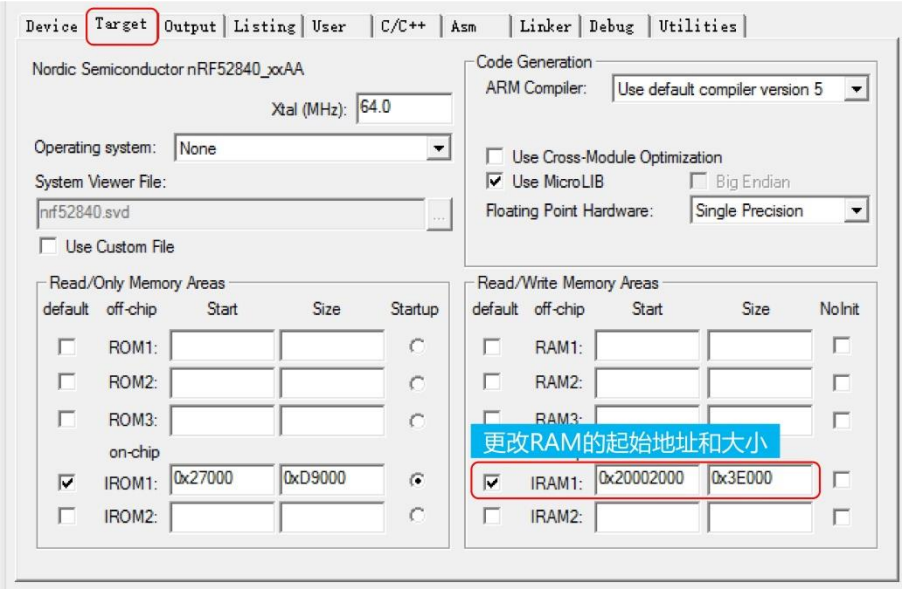


图 3-19：修改 RAM 配置

全局宏定义中加入“DEBUG”，如下图所示，加入“DEBUG”后，程序运行出错后不会复位，这样方便我们观察 LOG 信息。

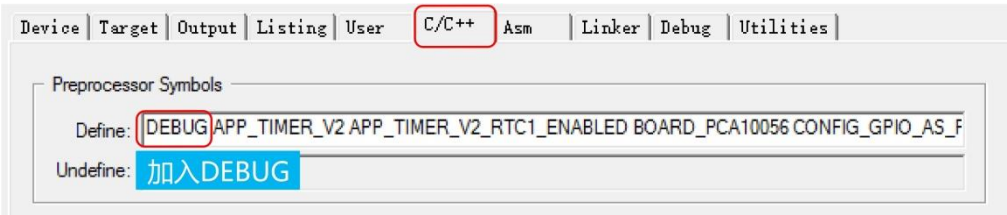


图 3-20：全局宏定义中加入“DEBUG”

编译程序，编译成功后将程序下载到开发板运行，计算机上打开 RTT Viewer，这时可以在 RTT 上看到如下图所示的信息。从中我们可以看到程序在调用 sd_ble_enable()函数时返回错误代码：内存不足，同时程序通过 Log 输出了 RAM 应该设置的正确的数值。

这里，我们也可以看出：因为分配 RAM 时为 SoftDevice 保留了运行必须的 RAM 空间，所以该程序虽然无法正常运行，但是可以运行到通过 Log 输出 RAM 的调整信息，因此，当我们在开发过程中遇到内存不足的问题时，可以通过 LOG 来获取正确的 RAM 配置。

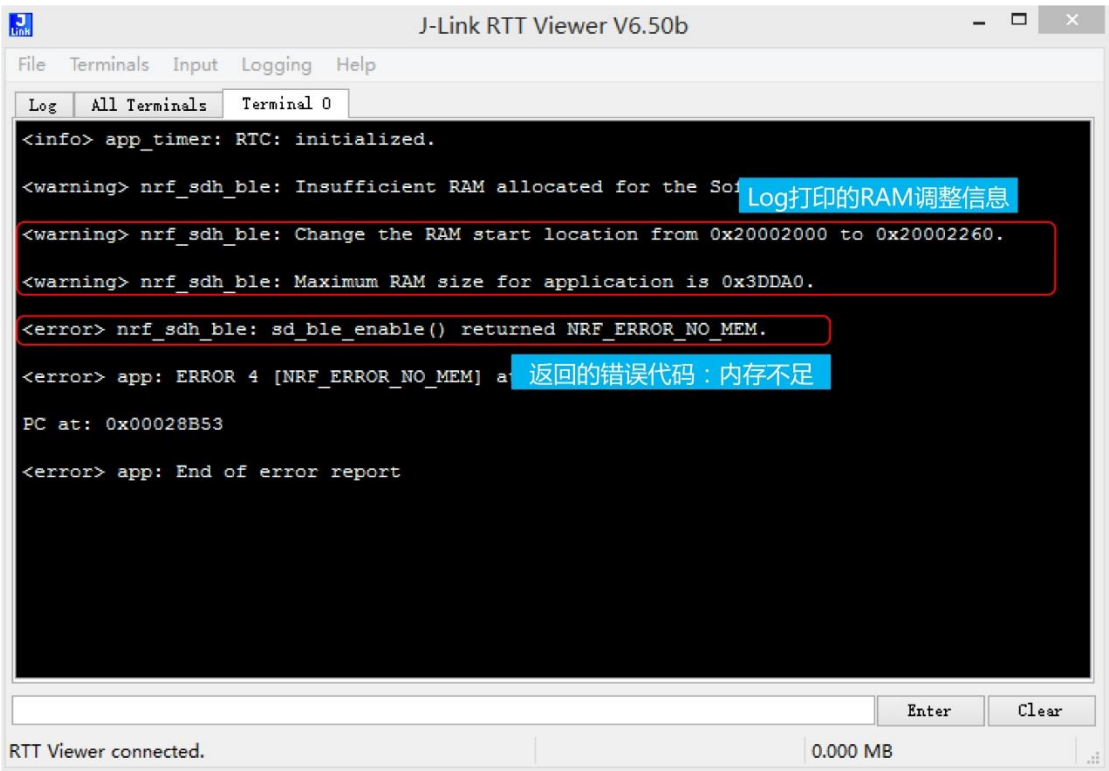


图 3-19：RAM 分配过小时的信息

有了内存调整信息后，我们在 MDK 工程配置中按照 Log 中的提示调整 RAM 分配，将 RAM 起始地址设置为 0x20002260，RAM 大小设置为 0x3DDA0，然后重新编译下载，这时程序会正常运行。

2. 第二种情况：内存分配过大，希望调整 RAM，节省内存。

这种情况下程序是正常运行的，调整内存按照第一种方法同样的方式处理，即根据 Log 输出信息调整 RAM 分配，以达到 RAM 分配最优化。

3. 第三种情况：内存分配过大，用户不关心浪费的那一点内存。

这种情况下若程序正常运行，用户无需理会。

4.6. 配置 GAP 参数

GAP 是蓝牙里面相对比较复杂的概念，本节我们需要掌握的知识点是：GAP 作用、GAP 角色、GAP 安全模式和 GAP 定义的服务（GAP 服务）。

4.6.1. GAP 作用

GAP（Generic Access Profile），通用访问配置文件。GAP 定义了设备如何彼此发现、建立连接以及如何实现绑定，同时描述了设备如何成为广播者和观察者，并且实现无需连接的传输。同时，GAP 定义了如何用不同类型的地址来实现隐私性和可解析性。GAP 和 BLE 体系结构中低层的关系如下图所示。

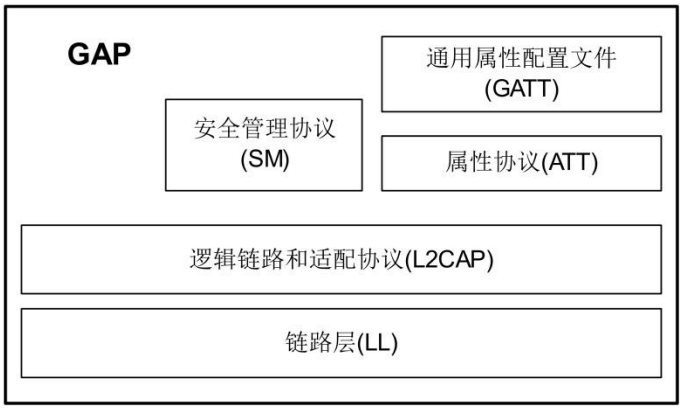


图 3-20：GAP 和 BLE 体系结构中低层的关系

GAP 的作用就是定义以下 4 个方面：

- GAP 角色。
- 可发现性模式和规程。
- 连接模式和规程。
- 安全模式和规程。

从学习 GAP 开始，我们会接触到 2 个非常重要的概念：模式（Mode）和规程（Procedure）。它们用来描述设备的行为，的定义和区别如下：

- 模式（Mode）：模式描述是设备的工作状态，当一个设备被配置为按照某种方式操作一段较长的时间时，称为模式。如广播模式，表示设备正处于广播状态，一般会持续很长时间。
- 规程（Procedure）：规程描述的是在有限的时间内进行特定的操作，如连接参数更新规程，它是在较短的时间内执行了连接参数更新的操作。

4.6.2. GAP 角色

BLE BLE 为设备在物理传输定义了 4 种 GAP 角色，一个设备可以支持多个 GAP 角色，如既可以是广播者，也可以同时是外围设备：

- 广播者：广播发送者，不是可连接的设备。
- 观察者：扫描广播，不能够启动连接。
- 外围设备：广播发送者，是可连接的设备，连接后成为从设备。
- 中心设备：扫描广播启动连接，连接后成为主设备。

要理解 GAP 为什么分为 4 种角色，就要知道蓝牙标准制定时的考虑。我们知道 BLE 主打低功耗、所以 BLE 体系结构中，为了最大可能优化设备，节省功耗，所有的层都采用了非对称的设计，对于物理层的无线电装置，可以是这 3 种形式。

- 芯片只有发射机：只能发射无线信号，不能接收无线信号，硬件成本低。
- 芯片只有接收机：只能接收无线信号，不能发射无线信号，硬件成本低。
- 芯片同时具有接收机和发射机：既可以接收无线信号，也可以发射无线信号，硬件成本较高。

这样，当某个应用只需要在设备之间单向传输数据时，其中一个设备可以作为广播者，采用只有发射机的芯片，另外一个设备可以作为观察者，采用只有接收机的芯片，这就构成了非对称系统，如下图所示。它们实现了单向的传输，并且用了尽量少的硬件资源，所以，功耗会更低，同时理论上造价也会更低。

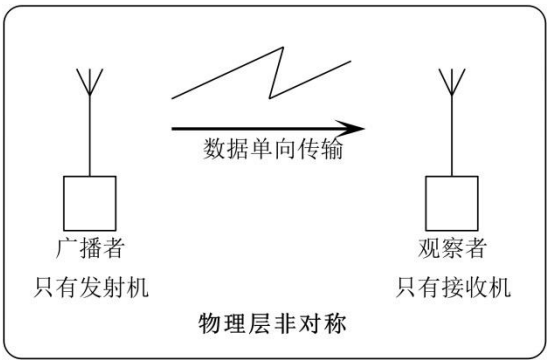


图 3-21：物理层非对称

当然，我们也可以用同时具有接收机和发射机的设备实现上面的功能（实际产品中，用这种方式的更多），但是因为只需实现单向传输，广播者多了用不到的接收机，观察者多了用不上的发射机，这会增加电流的消耗和硬件成本。

由此可以看到，蓝牙标准的制定考虑得足够多，从功耗、成本以及各种应用场景都做了全面的考虑。

但是，需要注意的是，芯片的价格不仅和硬件成本相关、还和芯片的出货量关系非常大，虽然蓝牙标准考虑了只有发射机和只有接收机的情况，并且为它们“量身定制”了广播者和观察者这样的角色，不过，实际应用中，绝大多数用户会采用同时具备接收机和发射机的方案，这使得芯片厂商更乐于推出具有同时具有收发器的芯片，如 Nordic 和 TI 的蓝牙 BLE 芯片方案，都是同时具备接收机和发射机的。所以，同时具备无线收发器的芯片的出货量远远大于只有发射机或只有接收机的芯片，从而具备无线收发器的芯片的市场价格会低于只有发射机或只有接收机的芯片，甚至，在市面上根本见不到只有发射机或只有接收机的芯片。

蓝牙核心规格中定义了在 BLE 物理传输上操作时每个 GAP 角色的物理层和链路层功能需要遵从的要求。如下表所示。

表 3-9：GAP 角色对物理层的要求

	GAP 角色			
	广播者	**观察者**	**外围设备**	**中心设备**
物理层功能				
发射机	M	O	M	M
接收机	O	M	M	M
链路层功能				
就绪态	M	M	M	M
广播态	M	E	M	E
扫描态	E	M	E	M

发起态		E	E	E	M
连接态	从机	E	E	M	E
	主机	E	E	E	M
广播事件类型					
可连接可扫描非定向事件		E	E	M	E
可连接非定向事件		E	E	O	E
可连接定向事件		E	E	O	E
不可连接不可扫描非定向事件		M	E	O	E
不可连接不可扫描定向事件		O	E	O	E
可扫描非定向事件		O	E	O	E
可扫描定向事件		O	E	O	E
扫描类型					
被动扫描		E	M	E	O
主动扫描		E	O	E	C1
链路层控制规程					
连接更新规程		E	E	M	M
信道图更新规程		E	E	M	M
加密规程		E	E	O	O
主机发起交换规程		E	E	M	M
从机发起交换规程		E	E	C2	C2
连接参数更新规程		E	E	O	O
版本交换规程		E	E	M	M
终止规程		E	E	M	M

M：强制要求的。 O：可选择的。 E：不包含的。

C1：如果支持被动扫描，那么主动扫描是可选择的，否则，强制支持主动扫描。

C2：如果支持连接参数更新，强制支持，否则是可选的。

4.6.3. GAP 安全模式

GAP 定义了如下表所示的两类安全模式和 4 个安全规程，它们被服务用于描述所需的安全级别，每个服务都可以根据自己对安全性的需求使用它们来进行描述。

表 3-10：GAP 安全模式和等级

安全模式和规程	广播者	观察者	外围设备	中心设备
安全模式 1	E	E	O	O
安全模式 2	E	E	O	O
认证规程	E	E	O	O
授权规程	E	E	O	O
连接数据签名规程	E	E	O	O

认证签名数据规程	E	E	O	O

安全模式是针对服务的，从上表中我们可以看到对于广播者和观察者，它们仅是一个对外广播数据，一个扫描、获取广播数据，它们没有服务，所以不需要安全模式。对于外围设备和中心设备，安全模式也是可选的，不是必须的，它取决于服务对安全性的需求。

安全模式 1 包含 4 个等级，安全模式 2 包含 2 个等级，如下表所示，安全模式 1 通过不同级别的加密保证安全，安全模式 2 通过不同级别的数字签名保证安全。

表 3-11：GAP 安全模式和等级

安全模式	等级	描述
安全模式 1	通过不同级别的加密保证安全。	
	等级 1	无安全性（无认证和加密）。
	等级 2	带加密的未认证配对。
	等级 3	带加密的认证配对。
	等级 4	带加密的 LE 安全连接配对。
安全模式 2	通过不同级别的数字签名保证安全	
	等级 1	带数据签名的未认证配对。
	等级 2	带数据签名的认证配对。

4.6.4. GAP 服务

GAP 层定义了 GAP 服务，它的作用是用来确定设备的信息，GAP 服务包含了 5 个特征：设备名称、外观特征、外围设备首选连接参数、中心设备地址解析和可解析私有地址。4 类 GAP 角色对 GAP 服务的需求如下表所示，正如前文所述，广播者和观察者没有服务，当然不能包含 GAP 服务，外围设备和中心设备对 GAP 服务是强制包含的。

表 3-12：GAP 服务需求

	广播者	观察者	外围设备	中心设备
GAP 服务	E	E	M	M

对于每一个 GAP 角色，当其包含 GAP 服务时，对 GAP 的特征的需求也是不一样的，如下表所示，对于设备名称和外观特征，外围设备和中心设备是强制包含的，对于外围设备首选连接参数，外围设备是可选择是否包含的，而中心设备是不能包含的，对于中心设备地址解析和可解析私有地址是有条件包含的。

表 3-13：GAP 角色对 GAP 服务的特征的需求

特征	广播者	观察者	外围设备	中心设备
设备名称	E	E	M	M
外观特征	E	E	M	M
外围设备首选连接参数	E	E	O	E
中心设备地址解析	E	E	C3	C2
可解析私有地址	E	E	C3	C3

C2：如果支持链路层隐私，强制包含该特征，否则不包含。

C3：如果支持链路层隐私，该特征可选择包含，否则不包含。

1. Device Name 特征

Device Name（设备名称）特征应包含 UTF-8 字符串的设备名称，一个设备只允许有一个设备名称，长度范围是 0~248 字节。如果设备是可发现的，设备名称特征应不需认证或授权即可读，如果设备是不可发现的，设备名称特征在没有认证或授权时应不可读。

表 3-14：设备名称特征

属性类型	属性值	属性许可
0x2A00（UUID）	设备名称	当设备可发现时，设备名称特征不需认证或授权即可读。 设备名称特征是否可写是可选的，写的认证和授权可以由上层的规范或具体的实现来定义。

GAP 设备名称特征操作需要掌握两个 API 函数：sd_ble_gap_device_name_set ()和 sd_ble_gap_device_name_get ()，其中 sd_ble_gap_device_name_set ()用于设置 GAP 设备名称，sd_ble_gap_device_name_get ()用于读取 GAP 设备名称，函数原型如下。

表 3-15：sd_ble_gap_device_name_set ()函数

函数原型	uint32_t sd_ble_gap_device_name_set (ble_gap_conn_sec_mode_t const * p_write_perm, uint8_t const * p_dev_name, uint16_t len)
函数功能	设置 GAP 设备名称。 注意：如果设备名称是存储在 Flash 中的，不能修改，否则函数会返回“NRF_ERROR_FORBIDDEN”。
参数	[in] p_write_perm：设备名称特征的写许可。 [in] p_dev_name：指向 UTF-8 编码，非空终止的字符串。 [in] len：指向的 UTF-8 编码、非空终止的字符串的长度，不能超过 BLE_GAP_DEVNAME_MAX_LEN 定义的长度。
返回值	• NRF_SUCCESS：GAP 设备名称和许可设置成功。 • NRF_ERROR_INVALID_ADDR：提供了无效的指针。 • NRF_ERROR_INVALID_PARAM：提供无效的参数。 • NRF_ERROR_DATA_SIZE：提供了无效的数据长度。 • NRF_ERROR_FORBIDDEN：设备名称不是可写的。

GAP 设备名称设置函数中涉及到了非空终止的字符串的概念和计算设备名称长度的方法，这是我们操作时需要注意的。

- 非空终止的字符串（NULL-terminated string）：C 语言中以'\0'作为字符串的结束标志，'\0'

在 ASCII 称为 NUL，非空终止的字符串即不包含'\0'的字符串。

- 计算设备名称字符串长度：计算长度时，需要获取的是不包含结束符'\0'的字符串长度，所以使用 strlen()函数来计算字符串长度，Strlen()计算字符串的长度时，返回的长度不包括结束符'\0'。

■ GAP 设备名称设置示例：首先要定义设备名称字符串，可以使用宏定义，也可以使用数组，同时要设置 GAP 的安全模式，这里设置的是无安全性，然后调用 API 函数 sd_ble_gap_device_name_set ()设置设备名称。

代码清单：设置 GAP 设备名称示例

```
#define DEVICE_NAME        "BLE_Template"            //设备名称字符串

ble_gap_conn_sec_mode_t sec_mode;
//设置 GAP 的安全模式
BLE_GAP_CONN_SEC_MODE_SET_OPEN(&sec_mode);
//设置 GAP 设备名称
err_code = sd_ble_gap_device_name_set(&sec_mode,
                                      (const uint8_t *)DEVICE_NAME,
                                      strlen(DEVICE_NAME));
//检查函数返回的错误代码
APP_ERROR_CHECK(err_code);
```

表 3-16：sd_ble_gap_device_name_get ()函数

函数原型	uint32_t sd_ble_gap_device_name_get(uint8_t * p_dev_name, uint16_t * p_len)
函数功能	读取 GAP 设备名称。 注意：如果获取的设备名称比提供的缓存大，p_len 返回的是实际获取的设备名称的长度，而缓存中只能根据自身大小保存部分设备名称信息，应用程序可以根据这些信息来调整缓存的大小。
参　　数	[out] p_dev_name：指向用于存放获取的设备名称字符串的缓存，获取的设备名称是 UTF-8 编码、非空终止的字符串。 [in,out] len：输入时提供用来保存设备名称的缓存（p_dev_name 指向的缓存）的大小，输出时返回的是实际获取的设备名称的长度。
返回值	NRF_SUCCESS：获取 GAP 设备名称成功。 NRF_ERROR_INVALID_ADDR：提供了无效的指针。 NRF_ERROR_DATA_SIZE：提供了无效的数据长度。

- GAP 设备名称读取示例：首先需要定义一个数组用来保存读取的设备名称，之后调用 API 函数 sd_ble_gap_device_name_get()读取，读取时需要指定读取的长度。这里需要注意数组长度（name_buf）和指定读取的长度（name_len）的关系。调用 sd_ble_gap_device_name_get()读取时，name_len 向函数传递了应用程序提供的用来保存设备名称的缓存的长度，读取后，函数 name_len 返回的是实际读取的设备名称的长度，这时候，如果读取的实际长度超过了 name_len，多出的部分不会保存到到 name_buf 数组。

代码清单：读取 GAP 设备名称示例

```
//保存读取的 GAP 设备名称，注意数组的长度
uint8_t name_buf[15];
//指定读取的长度
uint16_t name_len = 15;
//读取 GAP 设备名称
err_code = sd_ble_gap_device_name_get(name_buf,&name_len);
//读取成功，串口打印出设备名称
if(err_code == NRF_SUCCESS)
{
    printf("%s\r\n",name_buf);
}
```

2. Appearance 特征

外观是一个 16 位的数值，外观由 SIG 定义，用来列举设备的外观样式，一个设备只允许有一个外观特征。

外观指示设备是普通手机，鼠标、键盘等等，通常被用来在用户界面的设备旁显示设备的小图标，通过图标向用户展示设备。

表 3-17：外观特征

属性类型	属性值	属性许可
0x2A01（UUID）	外观	无需认证或授权即可读取。 外观称特征是否可写是可选的，写的认证和授权可以由上层的规范或具体的实现来定义。

如何获取外观的编码？可以在 SIG 网站上查到，网址如下：https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.characteristic.gap.appearance.xml 。

也可以在 BLE 工程的“ble_types.h”头文件中查到，如下图所示。

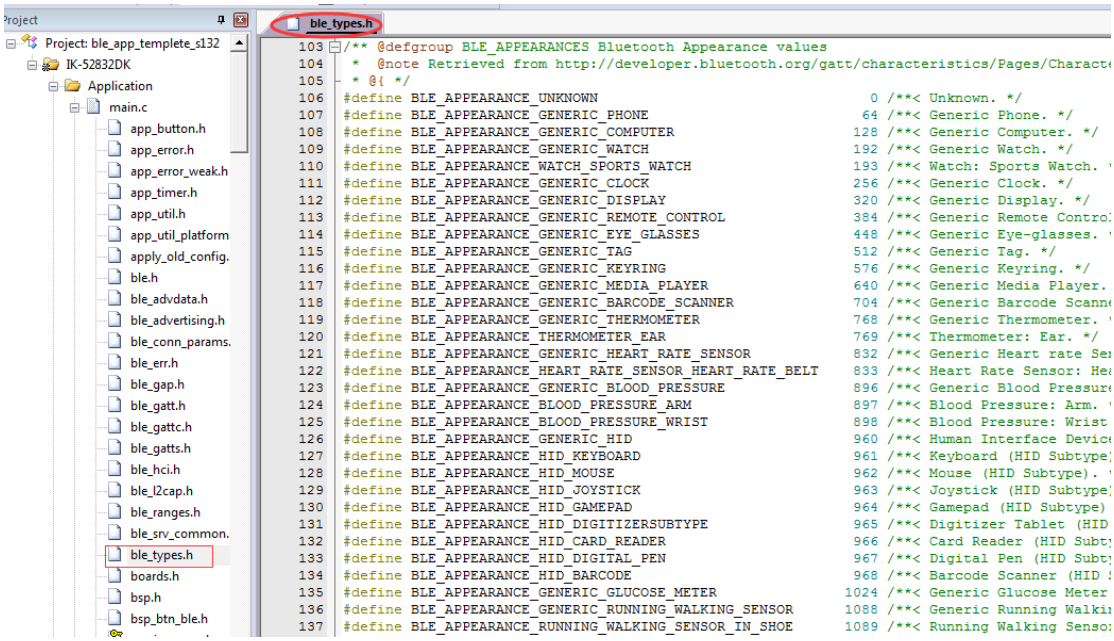


图 3-22：外观编码

GAP 外观特征操作需要掌握两个 API 函数：sd_ble_gap_appearance_set ()和 sd_ble_gap_appearance_get ()，其中 sd_ble_gap_appearance_set ()用于设置 GAP 外观特征，sd_ble_gap_appearance_get ()用于读取 GAP 外观特征，函数原型如下。

表 3-18：sd_ble_gap_appearance_set ()函数

函数原型	uint32_t sd_ble_gap_appearance_set(uint16_t appearance)
函数功能	设置 GAP 设备名称。
参　　数	[in] appearance：外观（16 位），设置的数值参考 SIG 的编码文件。
返回值	NRF_SUCCESS：GAP 设备名称和许可设置成功。 NRF_ERROR_INVALID_PARAM：提供了无效的参数。

- GAP 外观特征设置示例：设置 GAP 外观特征为心率传感器：心率带，对于本章的 BLE 工程模板，因为没有对应的外观特征，所以不需要设置，使用默认值 0（未知设备）即可。

代码清单：设置 GAP 外观特征示例

```
//设置 GAP 外观特征为心率传感器：心率带
err_code = sd_ble_gap_appearance_set(BLE_APPEARANCE_HEART_RATE_SENSOR_HEART_RATE
            _BELT);
//检查函数返回的错误代码
APP_ERROR_CHECK(err_code);
```

表 3-19：sd_ble_gap_appearance_get ()函数

函数原型	uint32_t sd_ble_gap_appearance_set(uint16_t appearance)
函数功能	读取 GAP 外观特征。
参　　数	[in] appearance：指向外观（16 位）的变量。
返回值	NRF_SUCCESS：读取外观特征成功。 NRF_ERROR_INVALID_ADDR：提供了无效的指针。

■ 读取 GAP 外观特征示例：首先定义一个无符号 16 位整形变量 my_appearance，作为 API 函数 sd_ble_gap_appearance_get()的输入参数，用于保存读取的外观，之后调用该 API 函数读取外观特征，读取成功后，外观特征保存到 my_appearance 中。

代码清单：读取 GAP 外观特征示例

```
//用于保存读取的外观特征，因为外观是 16 位的，所以定义 16 位的变量
uint16_t my_appearance;
//读取外观特征
err_code = sd_ble_gap_appearance_get(&my_appearance);
//读取成功后，打印外观编码
if(err_code == NRF_SUCCESS)
{
    NRF_LOG_INFO("%d\r\n",my_appearance);
}
```

3. PPCP 特征

外围设备首选连接参数（The Peripheral Preferred Connection Parameters (PPCP)）特征包含了外围设备倾向的连接参数，一个设备只允许有一个外围设备首选连接参数。外围设备首选连接参数必须可读，而且必须不可写。

表 3-20：外围设备首选连接参数特征

属性类型	属性值	属性许可
0x2A04（UUID）	外围设备首选连接参数	可读不可写。 认证和授权可以由上层的规范或具体的实现来定义。

为什么不使用固定的连接参数，而需要外围设备提供首选连接参数？因为外围设备和中心设备建立连接的时候，对于主机来说，并不一定能适应外围设备的连接参数，这时候，主机就可以以此为参考来调整连接参数，而不用去猜测这些参数。

外围设备首选连接包含 4 项目，每个项目 2 个字节共 8 个字节，其数据格式如下表所示。

表 3-21：外围设备首选连接参数数据格式

名称	大小（字节数）	描述
最小连接间隔	2	最小连接间隔，单位：1.25 ms。

		最小连接间隔时间 =最小连接间隔 * 1.25 ms。最小连接间隔可设置的范围 0x0006~0x0C80，对应的时间范围是（7.5ms 到 4.0s）。0xFFFF 表示没有特定的最小值，超出范围之外的值保留供将来使用。
最大连接间隔	2	最大连接间隔，单位：1.25 ms。 最大连接间隔时间 =最大连接间隔 * 1.25 ms。最大连接间隔可设置的范围 0x0006 ~ 0x0C80，对应的时间范围是（7.5ms 到 4.0s），必须等于大于最小连接间隔。0xFFFF 表示没有特定的最小值，超出范围之外的值保留供将来使用。
从机延迟	2	从机延迟定义了从机跳过连接事件的次数。这使外围设备具有一定的灵活性，如果它不具有任何数据传送，它可以选择跳过连接事件，并保持睡眠，从而提供了一些积蓄力量。这一决定取决于外围设备。从机延迟可设置的范围：0x0000~ 0x01F3。超出范围之外的值保留供将来使用。
监督超时	2	监督超时定义了连接监督超时时间，这是两个成功的连接事件之间间隔的最大值。如果超过这个时间还未出现成功的连接事件，那么设备将会考虑失去连接，返回一个未连接状态。 单位：10ms。 从机延迟可设置的范围：0x000A ~ 0x0C80。超出范围之外的值保留供将来使用。 监督超时时间= N * 10 ms，所以对应的时间范围是（100 ms 到 32s）。

我们可以看到连接间隔有最小连接间隔和最大连接间隔，是一个范围而不是一个固定的数值，连接间隔为什么不用固定的值而用范围？

这是因为从设备连接间隔范围代表外围设备倾向的连接间隔，连接参数完全由主设备控制，范围的目的是为了能让主机去选择一个主机认为合适的，如果把最小值和最大值设置成相同的，那么主机就会只用你设定的这个值。如果是个范围的话，通常主机会选择接近最大值的值或者直接用最大值的值。

从机可以在建立连接之后要求主机再更新连接参数，只要连接建立了，随时可以再次更新连接参数。

外围设备首选连接参数操作需要掌握两个 API 函数：sd_ble_gap_ppcp_set()和 sd_ble_gap_ppcp_get()，其中 sd_ble_gap_ppcp_set()用于设置外围设备首选连接参数，sd_ble_gap_ppcp_get()用于读取外围设备首选连接参数，函数原型如下。

1. sd_ble_gap_ppcp_set()函数

表 3-22：sd_ble_gap_ppcp_set()函数

函数原型	uint32_t sd_ble_gap_ppcp_set (ble_gap_conn_params_t const * p_conn_params

	)
函数功能	设置外围设备首选连接参数。
参　　数	[out] p_conn_params：指向保存连接参数的结构体变量。
返回值	NRF_SUCCESS：设置外围设备首选连接参数成功。 NRF_ERROR_INVALID_ADDR：提供了无效的指针。 NRF_ERROR_INVALID_PARAM：提供了无效的参数。

■ 外围设备首选连接参数示例：首先需要定义一个连接参数结构体变量 gap_conn_params，并初始化其成员变量，即设置外围设备首选连接参数的数值，之后调用 API 函数 sd_ble_gap_ppcp_set()，以 gap_conn_params 为输入参数设置外围设备首选连接参数。

代码清单：设置外围设备首选连接参数示例

```
1.  //定义连接参数结构体变量
2.  ble_gap_conn_params_t   ;
3.
4.   //设置首选连接参数，设置前先清零 gap_conn_params 变量
5.   memset(&gap_conn_params, 0, sizeof(gap_conn_params));
6.
7.   gap_conn_params.min_conn_interval = MIN_CONN_INTERVAL;//最小连接间隔
8.   gap_conn_params.max_conn_interval = MAX_CONN_INTERVAL;//最小连接间隔
9.   gap_conn_params.slave_latency     = SLAVE_LATENCY;     //从机延迟
10.  gap_conn_params.conn_sup_timeout  = CONN_SUP_TIMEOUT; //监督超时
11.  //调用协议栈 API sd_ble_gap_ppcp_set 配置 GAP 参数
12.  err_code = sd_ble_gap_ppcp_set(&gap_conn_params);
13.  //检查函数返回的错误代码
14.  APP_ERROR_CHECK(err_code);
```

2. sd_ble_gap_ppcp_get()函数

表 3-23：sd_ble_gap_ppcp_get()函数

函数原型	uint32_t sd_ble_gap_ppcp_get(ble_gap_conn_params_t *　　p_conn_params)
函数功能	设置外围设备首选连接参数。 注意：如果获取的设备名称比提供的缓存大，p_len 返回的是实际获取的设备名称的长度，而缓存中只能根据自身大小保存部分设备名称信息，应用程序可以根据这些信息来调整缓存的大小。
参　　数	[out] p_conn_params：指向用来保存读取的外围设备首选连接参数的结构体变量。

返回值	NRF_SUCCESS：获取外围设备首选连接参数成功。 NRF_ERROR_INVALID_ADDR：提供了无效的指针。

- 外围设备首选连接参数读取示例：首先需要定义一个 ble_gap_conn_params_t 结构体变量 my_gap_conn_params 用来保存读取的外围设备首选连接参数，之后调用 API 函数 sd_ble_gap_ppcp_get()读取，读取成功后，读取的外围设备首选连接参数会保存到 my_gap_conn_params。

代码清单：读取外围设备首选连接参数示例

```
//读取外围设备首选连接参数
//定义用来保存读取的外围设备首选连接参数的变量
ble_gap_conn_params_t   my_gap_conn_params;
//调用协议栈 API sd_ble_gap_ppcp_set 配置 GAP 参数
  err_code = sd_ble_gap_ppcp_get(&my_gap_conn_params);
//读取成功后，打印出读取的外围设备首选连接参数
if(err_code == NRF_SUCCESS)
{
    NRF_LOG_INFO("min_conn_interval = %d",my_gap_conn_params.min_conn_interval);
    NRF_LOG_INFO("max_conn_interval = %d",my_gap_conn_params.max_conn_interval);
    NRF_LOG_INFO("slave_latency = %d",my_gap_conn_params.slave_latency);
    NRF_LOG_INFO("conn_sup_timeout = %d",my_gap_conn_params.conn_sup_timeout);

}
```

3. Central Address Resolution 特征

蓝牙 4.2 为了提高蓝牙的安全性，增加了随即地址，Central Address Resolution（中心设备地址解析）用来定义设备是否支持地址解析隐私，一个设备只允许有一个中心设备地址解析特征。

表 3-24：中心设备地址解析特征

属性类型	属性值	属性许可
0x2AA6（UUID）	中心设备地址解析支持	无需认证或授权即可读取。 不可写。

中心设备地址解析长度为 1 个字节，如下表所示：

表 3-25：中心设备地址解析特征

值	描述
0	该设备不支持地址解析。
1	该设备支持地址解析。
2 – 255	保留供将来使用。

4. Resolvable Private Address Only 特征

设备应通过读取Resolvable Private Address Only特性来检查对端设备是否仅在绑定后使用可解析私有地址（RPA）。Resolvable Private Address Only 特性定义设备是否仅将 Resolvable Private Addresses（RPA）用作本地地址，如下表所示。

表 3-26：中心设备地址解析特征

属性类型	属性值	属性许可
0x2AC9（UUID）	Resolvable Private Address Only	无需认证或授权即可读取。 不可写。

表 3-27：中心设备地址解析特征

值	描述
0	绑定后，只有可解析私有地址用作本地地址。
1 – 255	保留供将来使用。

4.6.5. 代码编写

BLE 工程模板中，我们将 GAP 相关的初始化封装在 gap_params_init()函数中，方便调用。GAP 初始化中我们要做的是设置 GAP 安全模式和 GAP 服务，主要包括设置设备名称、外观特性和首选连接参数。

1. GAP 安全模式：这里设置的是 BLE_GAP_CONN_SEC_MODE_SET_OPEN(&sec_mode)，即无安全性，主机可以修改设备名称特征。
2. GAP 服务：BLE 工程模板实现的是外围设备，对于 GAP 服务必须包含设备名称特征和外观特征，其他 3 个特征是可选或有条件包含的。
   - Device Name 特征：设置为“BLE_Template”。
   - Appearance 特征：BLE 工程模板没有实现一个具体的应用，所以 BLE 工程模板对应的外观特征是“0”，即未知设备。外观特征初始值就是“0”，GAP 初始化函数中没有设置外观特征。
   - PPCP（外围设备首选连接参数）：虽然这个特征不是外围设备强制包含的，但是外围设备中一般都会包含，以便中心设备能第一时间知道外围设备倾向的连接参数。
   - Central Address Resolution 特征：对于外围设备是有条件选择包含的，但是协议栈是直接写死的，没有提供该特征的配置接口，也就是协议栈直接将该特征包含到了 GAP 服务，应用程序没有选择权。
   - PRAO（Resolvable Private Address Only）特征：该特征对于外围设备是有条件选择包含的，BLE 工程模板中没有包含该特征。

GAP 初始化函数 gap_params_init()代码清单如下：

代码清单：GAP 初始化

```
static void gap_params_init(void)
{
```

```
    ret_code_t              err_code;
    //定义连接参数结构体变量
    ble_gap_conn_params_t   gap_conn_params;
    ble_gap_conn_sec_mode_t sec_mode;
    //设置 GAP 的安全模式，即设置设备名称特征的写权限，这里设置的是安全模式 1 等级 1，
    //即无安全性
    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&sec_mode);
    //设置 GAP 设备名称
    err_code = sd_ble_gap_device_name_set(&sec_mode,
                                          (const uint8_t *)DEVICE_NAME,
                                          strlen(DEVICE_NAME));
    //检查函数返回的错误代码
    APP_ERROR_CHECK(err_code);

    //如果需要设置外观特征，在这里使用如下的代码设置
    /*
    err_code = sd_ble_gap_appearance_set(BLE_APPEARANCE_);
    APP_ERROR_CHECK(err_code); */

    //设置首选连接参数，设置前先清零 gap_conn_params
    memset(&gap_conn_params, 0, sizeof(gap_conn_params));

    gap_conn_params.min_conn_interval = MIN_CONN_INTERVAL;//最小连接间隔
    gap_conn_params.max_conn_interval = MAX_CONN_INTERVAL;//最小连接间隔
    gap_conn_params.slave_latency     = SLAVE_LATENCY;    //从机延迟
    gap_conn_params.conn_sup_timeout  = CONN_SUP_TIMEOUT; //监督超时
    //调用协议栈 API sd_ble_gap_ppcp_set 配置 GAP 参数
    err_code = sd_ble_gap_ppcp_set(&gap_conn_params);
    APP_ERROR_CHECK(err_code);
}
```

4.7. GATT 初始化

GATT 程序模块的作用是用于协商和跟踪 GATT 连接参数和更新数据长度，在 BLE 程序中，初始化 GATT 程序模块只需调用库函数 nrf_ble_gatt_init ()即可完成初始化，函数原型如下。

表 3-28：nrf_ble_gatt_init()函数

函数原型	ret_code_t nrf_ble_gatt_init(nrf_ble_gatt_t * p_gatt, nrf_ble_gatt_evt_handler_t evt_handler)
函数功能	初始化 GATT 模块。

参　　数	[in] evt_handler：事件句柄。 [out] p_gatt：指向 GATT 结构体变量。
返回值	NRF_SUCCESS：操作成功。 NRF_ERROR_NULL：指针 p_gatt 为 NULL。

初始化 GATT 程序模块的程序清单如下，首先定义一个 GATT 模块实例 m_gatt，然后调用库函数 nrf_ble_gatt_init()执行初始化，初始化成功后，默认的数据长度等参数会写到到 m_gatt。

代码清单：初始化 GATT 程序模块

```
NRF_BLE_GATT_DEF(m_gatt);                //定义 GATT 模块实例

static void gatt_init(void)
{
    //初始化 GATT 程序模块
    ret_code_t err_code = nrf_ble_gatt_init(&m_gatt, NULL);
    //检查函数返回的错误代码
    APP_ERROR_CHECK(err_code);
}
```

4.8. 广播初始化

广播者对外广播的目的是为了像周边的设备显示自己的存在和“我是谁”，所以广播包中需要按照规定的格式写入一些数据。

为什么要按照规定的格式来写入数据？

因为不按照规定的格式来写入，广播报文就毫无规则可言，那么对端设备就无法理解广播中数据的意义，从而无法解析广播报文，那么 BLE 也就失去了它的一大优势“互操作性”。设备每次广播时，会在 3 个广播信道上发送相同的报文。这些报文被称为一个广播事件。除了定向报文以外，其他广播事件均可以选择“20ms ~ 10.28s”不等的间隔。

广播初始化中，需要完成的是配置广播数据、广播模式和广播间隔，同时应用程序要提供广播事件处理函数用于接收广播事件。广播初始化通过调用库函数 ble_advertising_init()完成，该函数原型如下。

表 3-29：ble_advertising_init()函数

函数原型	uint32_t ble_advertising_init (ble_advertising_t *const p_advertising, ble_advertising_init_t const *const p_init)
函数功能	广播初始化函数。该函数对所需的广播数据进行编码并将其传递给协议栈。

参　　数	[out] p_advertising：广播模块实例。必须由应用程序提供，广播模块实例在该函数中初始化，之后作为一个特定的模块实例使用。 [in] p_init：指向广播初始化结构体。
返回值	NRF_SUCCESS：广播初始化成功。 NRF_ERROR_INVALID_PARAM：p_init 指向的广播初始化结构体中包含了无效的广播配置数据。

4.8.1. 配置广播数据和模式

1. 广播数据

广播初始化中需要配置广播包中包含哪些数据，一般情况下，建议广播包中至少包含：设备名称、Flags、外观和首要服务的 UUID。对于 BLE 工程模板，因为仅有两个必须的服务（GAP 服务和 GATT 服务），没有其他的服务，所以这里我们在广播数据中只加入设备名称、外观和 Flasg。

2. 广播模式和间隔

连接前，外围设备需要先广播，向中央设备通告自己的存在，配置广播时，我们要确定在我们的应用中需要使用哪一种广播模式，SDK 中定义了下面几种可连接的广播模式：

表 3-30：广播模式对应的行为

广播模式	广播行为
BLE_ADV_MODE_IDLE	没有可连接的广播在进行中。
BLE_ADV_MODE_DIRECTED_HIGH_DUTY	定向广播（高占空比）。断开连接后会立即尝试和最近连接过的对端设备进行重连接。这种模式对于无缝恢复意外断开连接的对端设备非常有用。但是因为他会堵塞广播通道，影响其他设备的广播，所以持续时间不能过长。
BLE_ADV_MODE_DIRECTED	定向广播（低占空比）。
BLE_ADV_MODE_FAST	快速地广播一小段时间。
BLE_ADV_MODE_SLOW	慢速广播，类似于快速广播。默认情况下相对于快速广播使用更长的广播间隔和广播持续时间。因为广播间隔较长，所以连接此类设备速度会慢一些。不过，连接间隔和持续时间是由用户定义的。

不同的广播模式的广播间隔和持续时间不一样，广播间隔和广播持续时间的影响如下：

- 广播间隔：单位 0.625ms，广播快，容易被中心设备发现，慢则省电。
- 广播持续时间：保持广播的时间，为了省电，可以广播一段时间之后进入休眠。

3. 广播事件处理函数

应用程序在广播初始化的时候需要注册广播事件处理函数，用于接收广播事件，广播事件处理函数的格式如下，函数的名称必须和注册的名称一样。

代码清单：广播事件处理函数格式

```
static void on_adv_evt(ble_adv_evt_t ble_adv_evt)
{
    ret_code_t err_code;
    //判断广播事件类型
    switch (ble_adv_evt)
    {
        //快速广播启动事件：快速广播启动后会产生该事件
        case BLE_ADV_EVT_FAST:
            //添加功能代码
        break;

        //其他广播事件

        //广播 IDLE 事件：广播停止后会产生该事件
        case BLE_ADV_EVT_IDLE:
            //添加功能代码
        break;

        default:
        break;
    }
}
```

4. 写入连接设置标记

广播初始化，还需要调用 ble_advertising_conn_cfg_tag_set()函数写入连接设置标志，它的作用是如果广播连接后，用来跟踪使用什么连接设置。这个功能是为了将来考虑的，在当前 SoftDevice 版本中（S140 V6.1 版本），只能写 1。ble_advertising_conn_cfg_tag_set()函数原型如下。

表 3-31：ble_advertising_conn_cfg_tag_set()函数

函数原型	void ble_advertising_conn_cfg_tag_set (ble_advertising_t *const p_advertising, uint8_t ble_cfg_tag)
函数功能	用于更改后续连接的连接设置标记。
参　　数	[in] p_advertising：广播模块实例。

	[in] ble_cfg_tag：指向 GATT 结构体变量。
返回值	无。

4.8.2. 编写代码

首先定义一个广播初始化结构体，之后，对结构体中需要配置的成员变量赋值，然后以此初始化结构体做为函数输入参数，调用库函数 ble_advertising_init()完成初始化。

代码清单：广播初始化

```
static void advertising_init(void)
{
    ret_code_t             err_code;
    //定义广播初始化配置结构体变量
    ble_advertising_init_t init;
    //配置之前先清零
    memset(&init, 0, sizeof(init));
    //设备名称类型：全称
    init.advdata.name_type               = BLE_ADVDATA_FULL_NAME;
    //是否包含外观：包含
    init.advdata.include_appearance      = true;
    //Flag:一般可发现模式，不支持 BR/EDR
    init.advdata.flags                   = BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DIS
                                           C_MODE;
    //设置广播模式为快速广播
    init.config.ble_adv_fast_enabled  = true;
    //设置广播间隔和广播持续时间
    init.config.ble_adv_fast_interval = APP_ADV_INTERVAL;
    init.config.ble_adv_fast_timeout  = APP_ADV_DURATION;
    //广播事件回调函数
    init.evt_handler = on_adv_evt;
    //初始化广播
    err_code = ble_advertising_init(&m_advertising, &init);
    APP_ERROR_CHECK(err_code);
    //设置广播配置标记。APP_BLE_CONN_CFG_TAG 是用于跟踪广播配置的标记，这是为未来预留的
    //一个参数，在将来的 SoftDevice 版本中，可以使用 sd_ble_gap_adv_set_configure()配
    //置新的广播配置，当前 SoftDevice 版本（S140 V7.0.1 版本）支持的最大广播集数量为 1，
    //因此 APP_BLE_CONN_CFG_TAG 只能写 1。
    ble_advertising_conn_cfg_tag_set(&m_advertising, APP_BLE_CONN_CFG_TAG);
}
```

因为 BLE 工程模板中，我们使用的广播模式是快速广播，所以广播事件处理函数中，只需处理 BLE_ADV_EVT_FAST 事件和 BLE_ADV_EVT_IDLE 事件，处理如下：

- BLE_ADV_EVT_FAST 事件：当快速广播成功启动后产生该事件，该事件中设置广播指示灯为正在广播，指示当前状态为广播状态（D1 指示灯闪烁）。
- BLE_ADV_EVT_IDLE 事件：当广播超时后产生该事件，该事件中设置广播指示灯为正在广播停止（D1 指示灯熄灭）。

代码清单：广播事件处理函数

```
//广播事件处理函数
static void on_adv_evt(ble_adv_evt_t ble_adv_evt)
{
    ret_code_t err_code;
    //判断广播事件类型
    switch (ble_adv_evt)
    {
        //快速广播启动事件：快速广播启动后会产生该事件
        case BLE_ADV_EVT_FAST:
            NRF_LOG_INFO("Fast advertising.");
            //设置广播指示灯为正在广播（D1 指示灯闪烁）
            err_code = bsp_indication_set(BSP_INDICATE_ADVERTISING);
            APP_ERROR_CHECK(err_code);
            break;
        //广播 IDLE 事件：广播超时后会产生该事件
        case BLE_ADV_EVT_IDLE:
            //设置广播指示灯为广播停止（D1 指示灯熄灭）
            err_code = bsp_indication_set(BSP_INDICATE_IDLE);
            APP_ERROR_CHECK(err_code);
            break;

        default:
            break;
    }
}
```

4.8.3. 本节留下的问题

1. BLE 工程模板实现的是无限广播，如果需要实现有限广播该如何操作？
2. 如何在广播中包含其它广播数据，如 UUID、自定义厂商数据等等？
   如何在广播中包含其它广播数据，参阅《第 4 章：全面掌握广播》。

4.9. 服务初始化

4.9.1. 初始化排队写入模块

服务初始化部分需要做的是初始化排队写入模块以及添加服务和，对于 BLE 工程模板，因为没有除了 GAP 服务和 GATT 服务之外的其他服务，所以只需初始化排队写入模块即可。

■ **什么是排队写入？为什么需要排队写入？**

写入 GATT 特性可以在单个写入命令或一系列命令中完成，显然，将写操作分割为几个命令使得可以写比单个包长的属性值，这样的一系列写命令，称为排队写入，排队写入由多个准备写入命令和一个执行写命令组成。

排队写入模块使用 SoftDevice 的通用属性配置文件（GATT）服务器接口提供排队写入的实现，将此模块添加到 GATT 服务器，以实现支持对某些或全部特性的排队写入，这样，GATT 客户端就可以使用一系列命令写入这些特性。

排队写入模块通过调用 nrf_ble_qwr_init()函数执行初始化，初始化时，应用程序需要提供排队写入实例和初始化结构体，nrf_ble_qwr_init()函数原型如下表所示。

表 3-32：nrf_ble_qwr_init ()函数

函数原型	ret_code_t nrf_ble_qwr_init(nrf_ble_qwr_t * p_qwr, nrf_ble_qwr_init_t const * p_qwr_init)
函数功能	初始化排队写入模块。对于一个给定的排队写入实例，初始化函数只能执行一次。
参　　数	[out] p_qwr：排队写入实例，必须由应用程序提供，排队写入实例由该函数初始化，之后被用来识别特定的排队写实例。 [in] p_qwr_init：初始化结构体。
返回值	NRF_SUCCESS：队列写入模块初始化成功。 NRF_ERROR_NULL：提供的指针为 NULL。 NRF_ERROR_INVALID_STATE：排队写入模块已经被初始化。

BLE 工程模板中，服务初始化相关的代码封装在 services_init()函数中，我们添加其它服务的时候，也应该在该函数中添加，这样程序比较清晰，管理起来也比较方便。

代码清单：服务初始化

```
//服务初始化，包含初始化排队写入模块和初始化应用程序使用的服务
static void services_init(void)
{
    ret_code_t          err_code;
    //定义排队写入初始化结构体变量
    nrf_ble_qwr_init_t qwr_init = {0};

    //排队写入事件处理函数
    qwr_init.error_handler = nrf_qwr_error_handler;
    //初始化排队写入模块
    err_code = nrf_ble_qwr_init(&m_qwr, &qwr_init);
    //检查函数返回值
```

```
    APP_ERROR_CHECK(err_code);

    //加入添加服务的代码，因为 BLE 工程模板只有两个强制包含的服务：GAP 服务和 GATT 服务，
    //所以这里不需要加入添加服务的代码
}
```

4.9.2. 本节留下的问题

1. BLE 工程模板中包含了 GAP 和 GATT 两个必须包含的服务，那么如何添加其它的服务和特征？

   如何添加标准的服务，参阅《第 6 章：实现一个标准的 Profile》的第 4.6 节和第 4.7 节。

   如何添加自定义的服务，参阅《第 7 章：实现一个自定义的 Profile》的第 3.6 节。

4.10. 连接参数协商初始化

4.10.1. 连接参数协商

连接参数协商模块的作用是用于启动和执行连接参数协商规程。

为什么要更新连接参数？

很多应用场景中，从机和主机刚建立连接时，会使用“快速”的连接参数以达到迅速交换信息的目的，但是，从机和主机之间并不是一直需要快速的交换信息，因为这样会加快从机电池能量的消耗，这时候，从机就希望能降低连接事件的频度(连接间隔)，从而降低功耗，延长电池工作时间。

从机可以通过向主机发送“连接参数更新请求”来请求修改连接参数，当从机向主机发送请求后，如果主机觉得合适，会同意这个请求并返回结果代码为接受的“连接参数更新响应”，如果主机觉得不合适，则会返回结果代码为拒绝的“连接参数更新响应”。从机在请求被拒绝后可以再次发送新的请求。由此我们可以看出，连接参数的更新能不能得以实现是由主机来决定的，所以，为了减少主机拒绝从机连接参数更新请求的可能性，在请求里面，连接间隔是一个范围而不是一个固定值，从而让主机有一个可选择的范围，增加主机接受请求的可能性。

对于主机来说，在连接建立时，主机通过连接请求数据包发送连接参数。当连接活跃了一段时间，连接参数也许不再适用于当前使用的服务。出于提高效率的目的，连接参数需要进行更新。这时候，主机向从机发送连接更新请求(注意：不是连接参数更新请求，不要和从机发送的连接参数更新请求混淆了)，即 LL_CONNECTION_UPDATE_REQ，当中携带了新的参数。这些参数不必进行协商，从机只有两种选择：要么接受和使用它们，要么断开链路。

SDK 中是通过连接参数协商程序模块来处理连接参数协商的，对于应用程序来说，需要提供的是连接参数协商的配置参数，同时提供 2 个事件回调函数接收连接参数协商模块的反馈信息，之后调用库函数 ble_conn_params_init ()初始化连接参数协商模块，初始化成功之后，连接参数的协商由连接参数协商程序模块自动完成。ble_conn_params_init ()函数原型如下：

表 3-33：ble_conn_params_init ()函数

函数原型	ret_code_t ble_conn_params_init (const ble_conn_params_init_t * p_init)
函数功能	初始化连接参数协商模块。 注意如果连接参数协商需要在通知/指示或任一特征被对端设备使能后触发，那么该函数必须在初始化服务之后调用。
参　　数	[in] p_init：包含初始化连接参数协商模块所需的所以选项和数据。
返回值	NRF_SUCCESS：初始化成功。 NRF_ERROR_INVALID_ADDR：提供的连接参数指针无效。 NRF_ERROR_INVALID_PARAM：提供的连接参数无效。 NRF_ERROR_NULL：p_init 指向 NULL。 NRF_ERROR_INTERNAL：发生了未知的错误。

4.10.2. 连接参数协商模块的参数和事件

SDK 中提供了连接参数协商配置结构体 ble_conn_params_init_t，该结构体中封装了所有连接参数协商所需的配置和数据，对于初始化连接参数协商模块来说，最重要的就是设置结构体中成员变量的值，同时开发者也要理解这些值的功能和作用，ble_conn_params_init_t 结构体定义如下：

代码清单：连接参数模块初始化结构体

```
//连接参数模块初始化结构体，包含初始化连接参数协商模块所需的所有选项和数据
typedef struct
{
  //指向应用程序中设置的连接参数，如果 p_conn_params 在初始化函数 conn_params_init()中
  //设置为 NULL，那么连接参数将从主机获取
  ble_gap_conn_params_t *        p_conn_params;
  //连接或启动通知到首次发起连接参数更新请求之间的时间（单位是定时器时钟滴答数量）
  uint32_t                       first_conn_params_update_delay;
  //每次调用 sd_ble_gap_conn_param_update()函数发起连接参数更新请求的之间的间隔时间
  //（单位是定时器时钟滴答数量），4.0BLE 规格中建议值是 30 秒
  uint32_t                       next_conn_params_update_delay;
  //放弃连接参数协商前尝试连接参数协商的最大次数
  uint8_t                        max_conn_params_update_count;
  //设置连接参数更新从通知使能或连接事件开始计时。
  //如果从通知使能开始计时，设置为对应 CCCD(客户端特性配置描述符)句柄，
  //如果时从连接事件开始计时，设置为 BLE_GATT_HANDLE_INVALID，
  uint16_t                       start_on_notify_cccd_handle;
  //设置为 true 时，如果连接参数更新失败，则自动断开连接。设置为 false 则不会断开连接
  bool                           disconnect_on_fail;
  //事件处理函数句柄，处理连接参数协商事件
  ble_conn_params_evt_handler_t evt_handler;
```

```
22.   //错误处理函数句柄，发生错误时调用
23.   ble_srv_error_handler_t       error_handler;
24. } ble_conn_params_init_t;
```

配置结构体 ble_conn_params_init_t 中包含了 2 个事件句柄，其中 evt_handler 用来处理连接参数协商事件，error_handler 用来处理错误事件，应用程序需要提供这 2 个事件回调函数，在初始化时注册。

1. 连接参数协商事件

SDK 中为连接参数协商事件定义了如下两个事件类型：协商成功事件和协商失败事件。

代码清单：连接参数协商模块事件类型

```
1.  typedef enum
2.  {
3.      BLE_CONN_PARAMS_EVT_FAILED,                 //协商失败
4.      BLE_CONN_PARAMS_EVT_SUCCEEDED               //协商成功
5.  } ble_conn_params_evt_type_t;
```

连接参数协商事件处理函数的格式如下，注意函数名称必须和注册的函数名称一样。在连接参数协商事件处理函数中，可以判断事件类型，根据事件类型执行相应的操作，如协商失败后断开连接，协商成功后点亮一个 LED 指示协商成功等等。

代码清单：连接参数协商事件处理函数

```
1.  //连接参数协商模块事件处理函数
2.  static void on_conn_params_evt(ble_conn_params_evt_t * p_evt)
3.  {
4.      ret_code_t err_code;
5.      //判断事件类型，根据事件类型执行动作
6.      //连接参数协商失败
7.      if (p_evt->evt_type == BLE_CONN_PARAMS_EVT_FAILED)
8.      {
9.          //添加处理协商失败的功能代码;
10.     }
11.     //连接参数协商成功
12.     if (p_evt->evt_type == BLE_CONN_PARAMS_EVT_SUCCEEDED)
13.     {
14.         //添加处理协商成功的功能代码;
15.     }
16. }
```

2. 错误事件

错误事件处理函数的格式如下，注意函数名称必须和注册的函数名称一样。错误事件用来接收连接参数协商模块发生错误时提交的错误代码，开发时可以根据错误代码分析错误的原因。

代码清单：错误事件处理函数

```
//连接参数协商模块错误处理事件，参数 nrf_error 包含了错误代码，通过 nrf_error 可以分析
//错误信息
static void conn_params_error_handler(uint32_t nrf_error)
{
    //添加处理错误代码 nrf_error 的功能代码
    APP_ERROR_HANDLER(nrf_error);
}
```

4.10.3. 编写代码

知道了如何配置连接参数协商模块的参数以及编写连接参数协商模块的事件处理函数，我们就可以编写出连接参数协商部分的初始化代码。首先定义一个连接参数初始化结构体，之后，对结构体中各个成员变量赋值，最终以此初始化结构体做为函数输入参数，调用库中的函数 ble_conn_params_init()完成初始化。

代码清单：连接参数协商模块初始化

```
static void conn_params_init(void)
{
    ret_code_t             err_code;
    //定义连接参数协商模块初始化结构体
    ble_conn_params_init_t cp_init;
    //配置之前先清零
    memset(&cp_init, 0, sizeof(cp_init));
    //设置为 NULL，从主机获取连接参数
    cp_init.p_conn_params                  = NULL;
    //连接或启动通知到首次发起连接参数更新请求之间的时间设置为 5 秒
    cp_init.first_conn_params_update_delay = FIRST_CONN_PARAMS_UPDATE_DELAY;
    //每次调用 sd_ble_gap_conn_param_update()函数发起连接参数更新请求的之间的
    //间隔时间设置为：30 秒
    cp_init.next_conn_params_update_delay  = NEXT_CONN_PARAMS_UPDATE_DELAY;
    //放弃连接参数协商前尝试连接参数协商的最大次数设置为：3 次
    cp_init.max_conn_params_update_count   = MAX_CONN_PARAMS_UPDATE_COUNT;
    //连接参数更新从连接事件开始计时
    cp_init.start_on_notify_cccd_handle    = BLE_GATT_HANDLE_INVALID;
    //连接参数更新失败不断开连接
    cp_init.disconnect_on_fail             = false;
    //注册连接参数更新事件句柄
    cp_init.evt_handler                    = on_conn_params_evt;
    //注册连接参数更新错误事件句柄
    cp_init.error_handler                  = conn_params_error_handler;
    //调用库函数（以连接参数更新初始化结构体为输入参数）初始化连接参数协商模块
```

```
    err_code = ble_conn_params_init(&cp_init);
    //检查函数返回的错误代码
    APP_ERROR_CHECK(err_code);
}
```

之后，编写两个事件的处理函数，在 BLE 工程模板中，对于连接参数协商事件处理函数，我们只处理协商失败事件，当协商失败后，断开当前连接（注：若协商失败后断开连接，更方便的操作方法是将初始化配置结构体中的 disconnect_on_fail 设置为 true）。对于错误事件处理函数，协商处理模块提交的错误代码直接交给错误处理模块处理。

代码清单：连接参数协商事件处理函数

```
static void on_conn_params_evt(ble_conn_params_evt_t * p_evt)
{
    ret_code_t err_code;
    //判断事件类型，根据事件类型执行动作
    //连接参数协商失败
    if (p_evt->evt_type == BLE_CONN_PARAMS_EVT_FAILED)
    {
        err_code = sd_ble_gap_disconnect(m_conn_handle, BLE_HCI_CONN_INTERVAL_UN
                   ACCEPTABLE);
        APP_ERROR_CHECK(err_code);
    }
}
```

代码清单：错误事件处理函数

```
//连接参数协商模块错误处理事件，参数 nrf_error 包含了错误代码，通过 nrf_error 可以分析
//错误信息
static void conn_params_error_handler(uint32_t nrf_error)
{
    //添加处理错误代码 nrf_error 的功能代码
    APP_ERROR_HANDLER(nrf_error);
}
```

4.11. 启动广播

广播参数配置完成之后，调用库函数 ble_advertising_start()即可启动广播，该函数的输入参数 advertising_mode 必须和广播初始化中设置的广播模式一样，如广播初始化中设置的广播模式是 BLE_ADV_MODE_FAST，那么输入参数 advertising_mode 也必须是 BLE_ADV_MODE_FAST。

表 3-34：ble_advertising_start ()函数

函数原型	uint32_t ble_advertising_start (ble_advertising_t *const p_advertising, ble_adv_mode_t advertising_mode)
函数功能	使用广播初始化中设置的广播模式启动广播。
参　　数	[in] p_advertising：广播模块实例。 [in] advertising_mode：广播模式。
返回值	NRF_SUCCESS：启动广播成功。 NRF_ERROR_INVALID_STATE：广播程序模块没有初始化。

■ 启动快速广播示例：

ret_code_t err_code = ble_advertising_start(&m_advertising, BLE_ADV_MODE_FAST);

■ 启动慢速广播示例：

ret_code_t err_code = ble_advertising_start(&m_advertising, BLE_ADV_MODE_SLOW);

到这里，BLE 工程模板所需的代码全部完成，接下来我们将 BLE 工程模板下载到开发板运行。

4.12. 试验步骤

1. 解压“…\4：开发指南（下册）配套实验源码\”目录下的压缩文件“试验 3-3：BLE 工程模板（外围设备）”，将解压后得到的文件夹“ble_app_templete”拷贝合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_templete\project\mdk5”目录下的工程“ble_app_templete_s140.uvproj”。
4. 切换到 Target“flash_s140_nrf52_7.0.1_softdevice”，点击下载协议栈。
5. 切换到 Target“IK-52840DK”，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
6. 点击下载按钮下载程序 。如果需要对程序进行仿真，点击 Debug 按钮 即可将程序下载到 nRF52840 进行仿真。
7. 程序运行后，指示灯 D1 闪烁表示正在广播。
8. 手机上打开 nRF Connect，点击“SCAN”按钮启动扫描，这时会扫描到名为“BLE_Template”的设备，如下图所示。

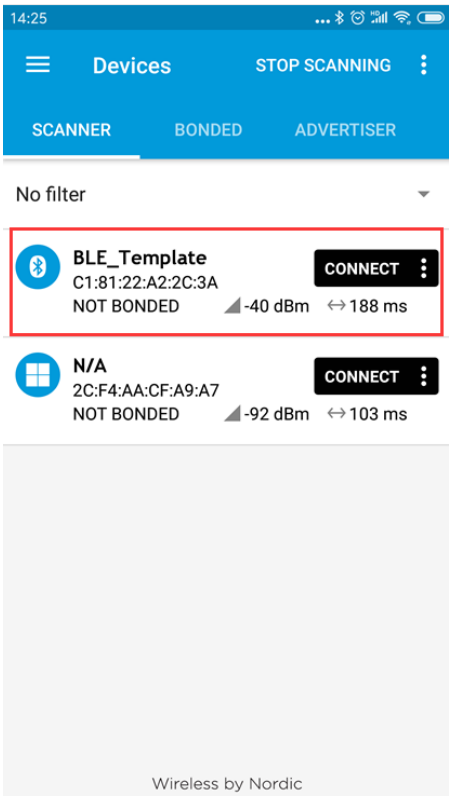


图 3-23：nRF Connect 扫描设备

9. 点击设备名称，展开广播信息。

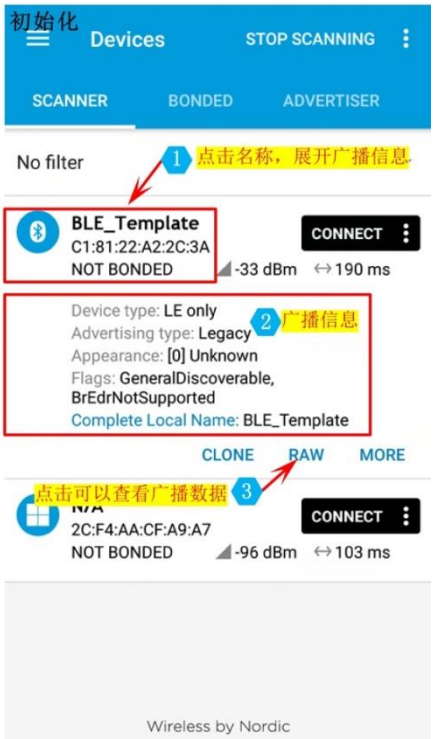


图 3-24：nRF Connect 上查看广播信息

在广播信息中我们可以看到如下信息，这些信息也正是我们在广播初始化中配置的广播信息。

- Device Type：LE only。仅 LE，
- Flags：GeneralDiscoverable，BrEdrNotSupported。一般可发现模式，不支持 BR/EDR。
- Appearance：[0]Unknown。外观的编码为 0，即未知设备。
- Complete Local Name：BLE_Templete。本地设备名称。

如果我们需要查看具体的，点击“RAW”打开数据窗口，如下图所示。

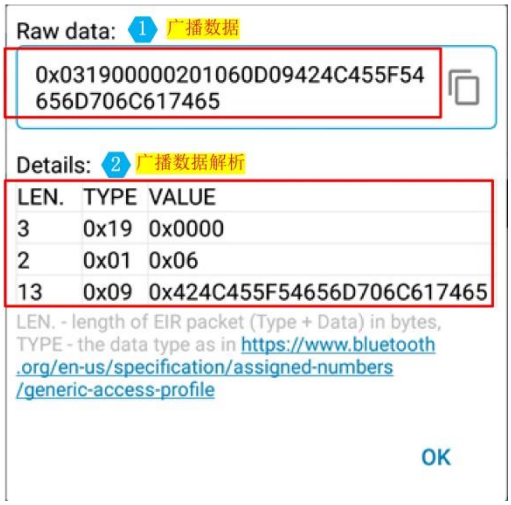


图 3-25：广播数据和解析

我们将广播数据取出，整理如下表所示，从中可以很方便地看到广播各个字段的含义以及广播数据长度。其中类型是由 SIG 定义，可以在 SIG 网站上查到，网址如下：https://www.bluetooth.com/specifications/assigned-numbers/generic-access-profile

表 3-35：广播数据分析

数据长度	类型	数值（16 进制）	总长度（字节数）
3	0x19（外观，1 个字节）	0000：未知设备（2 个字节）。	3+1 =4
2	0x01（flags，1 个字节）	06：LE 一般可发现模式，不支持 BR/EDR。（1 个字节）	2+1=3
13	0x09（本地设备名称，全称，1 个字节）	42(B) 4C(L) 45(E) 5F(_) 54(T) 65(e) 6D(m) 70(p) 6C(l) 61(e) 74(t) 65(e) （12 个字节）	13+1=14
广播数据总长度：4+3+14=21（字节）。			

10. 点击“CONNECT”按钮，连接设备，连接成功后，可以看到开发板上的指示灯 D1 常亮，指示已经建立连接，nRF Connect 上可以看到 BLE 工程模板中只包含了 2 个必须包含的服务：GAP 服务和 GATT 服务。

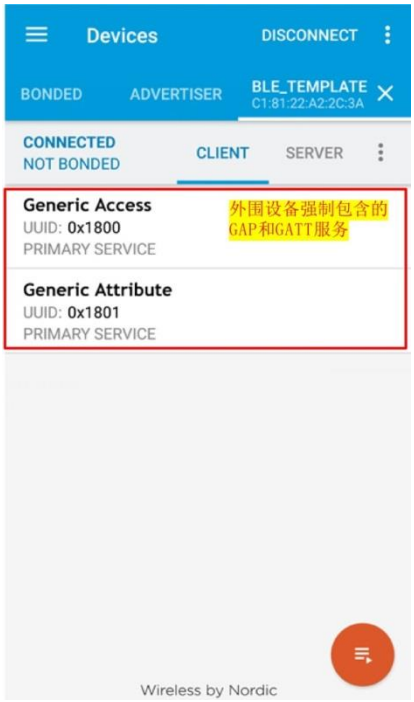


图 3-26：BLE 工程模板包含的服务

11. 点击服务名称即可展开服务，展开后可以看到服务中包含了哪些特征以及这些特征的属性。下图是展开后的 GAP 服务，GAP 服务包含了 4 个特征，特征名称右边的向上的箭头是“写”按钮，点击后可以写从机的特征值，向下的箭头是“读”按钮，点击后可以读出从机的特征值。下图中 GAP 的 4 个特征已经全部读出，可以看到读出的数值和我们在程序中配置的数值是一致的。

图 3-27：GAP 服务包含的特征

12. 断开连接：点击 nRF Connect 上的“DISCONNECT”按钮即可断开连接，断开后，系统重新进入广播模式，这时可以看到开发板上的指示灯 D1 闪烁，指示正在广播。

第四章：全面掌握广播

1. 学习目的

1. 掌握广播的基本概念和 SDK 实现广播的流程。
2. 了解广播的报文格式。
3. 掌握广播报文中可以包含哪些数据，这些数据的意义以及在程序里面如何将这些数据加入到广播报文中。

2. 基本概念和流程

广播就是设备将自身愿意展示的信息按照一定的间隔以“扫描者”可理解的方式向周边发射。

广播有如下四种类型，广播报文的报头中有 4 个位专门用于指示广播报文的类型（参见第 3 节：广播报文格式）。

- 通用广播：最常用的广播方式，可以被扫描，接收到连接请求时可以作为从设备进入一个连接。
- 定向广播：针对于快速建立连接的需求，定向广播会占满整个广播信道，定向广播的数据净荷只包含广播者和发起者地址，发起者收到发给自己的定向广播后，会立即发送连接请求。定向广播最长时间不能超过 1.28 秒，并且不能被主动扫描。
- 不可连接广播：广播数据，而不进入连接态，也不响应扫描，这是唯一一个允许硬件设备只有发射机的广播类型，因为它不需要接收任何数据。
- 可发现广播：不可连接，但可以响应扫描。

Nordic 的 SDK 里面实现的广播流程如下图所示，包含广播的配置、启动和停止。

- 广播的配置：应用程序首先要根据自己的需求配置广播的间隔、广播的模式以及广播中包含哪些数据，这称为广播初始化，广播初始化在配置好各项参数之后，最终调用 API 函数 sd_ble_gap_adv_set_configure()将配置传递给 SoftDevice 完成配置。配置成功后，SoftDevice 会返回“NRF_SUCCESS”。
- 广播的启动：应用程序初始化广播成功后，广播并没有启动，应用程序需要根据初始化中设置的模式启动广播，应用程序启动广播最终是调用 API 函数 sd_ble_gap_adv_start()通知 SoftDevice 启动广播，SoftDevice 启动广播成功后会返回“NRF_SUCCESS”。
- 广播的停止：广播的停止有 2 种方式，一是应用程序主动停止广播，即应用程序通过调用 API 函数 sd_ble_gap_adv_stop()通知 SoftDevice 停止广播，SoftDevice 成功停止广播后会返回“NRF_SUCCESS”；二是超时停止广播，即广播设置了超时时间，在设定的超时时间内没有和中心设备建立连接，广播即会超时停止，这时 SoftDevice 会向应用程序提交“BLE_GAP_EVT_ADV_SET_TERMINATED”事件。

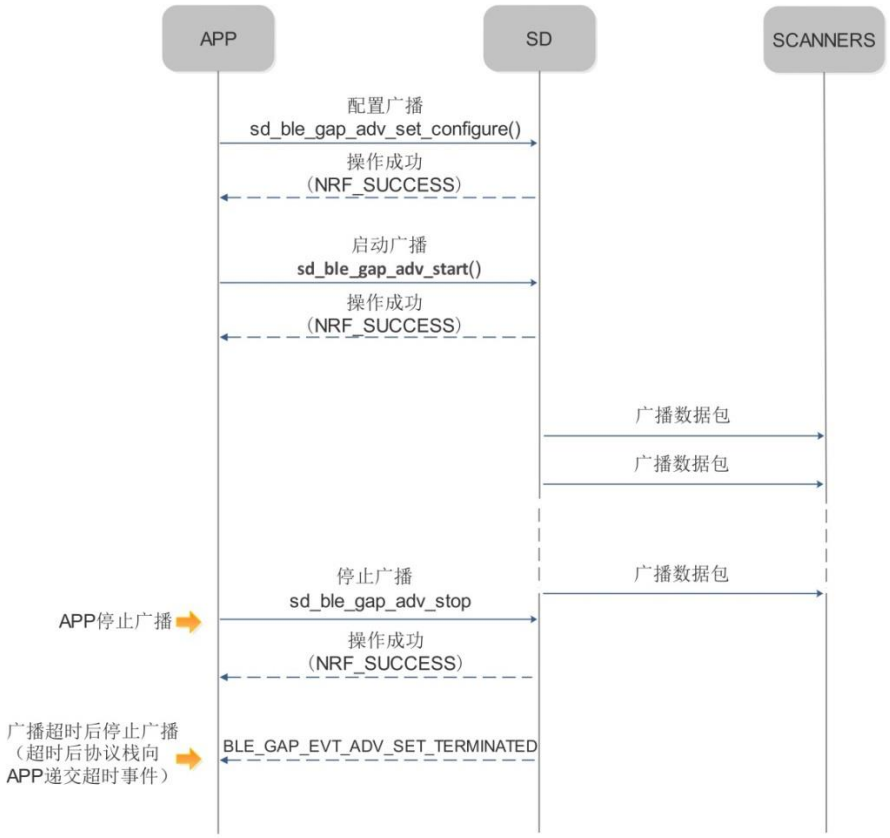


图 4-1：SDK 实现广播的流程

其中：

- APP：Application，应用程序。
- SD：SoftDevice。
- SCANNERS：扫描设备。

广播数据需要被“扫描者”理解，因此广播数据就必须遵循一定的格式，蓝牙核心规范中定义了广播的报文格式，设备广播时，广播数据按照规定的报文格式传输，设备扫描时，按照规定的格式解析数据。因此，要深入了解广播，就需要了解广播报文的数据格式。

接下来，我们看一下蓝牙核心规范中定义的广播报文的数据格式。

3. 广播数据包报文结构

BLE 报文结构如下，它由下图所示的各个域组成。因为有的域的长度超过了一个字节，所以在传输的过程中就涉及到多字节域中哪个字节先传输的问题，BLE 报文传输时的字节序和比特序如下：

- 字节序：大多数多字节域是从低字节开始传输的。注意，并不是所有的多字节域都是从低字节开始传输的。
- 比特序：各个字节传输时，每个字节都是从低位开始。

图 4-2：BLE 报文结构

3.1. 前导

前导是一个 8 比特的交替序列。他不是 01010101 就是 10101010，取决于接入地址的第一个比特。

- 若接入地址的第一个比特为 0：01010101
- 若接入地址的第一个比特为 1：10101010

接收机可以根据前导的无线信号强度来配置自动增益控制。

3.2. 接入地址

接入地址有两种类型：广播接入地址和数据接入地址。

1. 广播接入地址：固定为 0x8E89BED6，在广播、扫描、发起连接时使用。
2. 数据接入地址：随机值，不同的连接有不同的值。在连接建立之后的两个设备间使用。对于数据信道，数据接入地址是一个随机值，但需要满足下面几点要求：
   1) 数据接入地址不能超过 6 个连续的“0”或“1”。
   2) 数据接入地址的值不能与广播接入地址相同。
   3) 数据接入地址的 4 个字节的值必须互补相同。
   4) 数据接入地址不能有超 24 次的比特翻转(比特 0 到 1 或 1 到 0，称为 1 次比特翻转)。
   5) 数据接入地址的最后 6 个比特需要至少两次的比特翻转。
   6) 符合上面条件的有效随机数据接入地址大概有 231 个。

3.3. 报头

报头的内容取决于该报文是广播报文还是数据报文。广播报文的报头如下图所示：

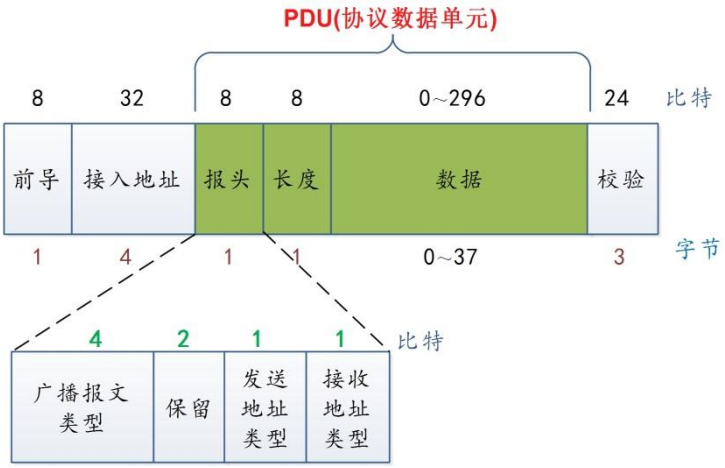


图 4-3：广播报文报头

广播报文的报头包含 4bit 广播报文类型、2bit 保留位、1bit 发送地址类型和 1bit 接收地址类型。

1. 广播报文类型

核心规范中定义了 7 种类型，如下图所示，通过该字段即可分辨出广播报文类型，如该字段是 ADV_IND，表示类型为通用广播。

PDU Type $b_3b_2b_1b_0$	Packet Name
0000	ADV_IND
0001	ADV_DIRECT_IND
0010	ADV_NONCONN_IND
0011	SCAN_REQ
0100	SCAN_RSP
0101	CONNECT_REQ
0110	ADV_SCAN_IND
0111-1111	Reserved

Core_v4.2 P2583的表2.1

广播报文类型共7种	
ADV_IND	通用广播指示
ADV_DIRECT_IND	定向连接指示
ADV_NONCONN_IND	不可连接指示
SCAN_REQ	主动扫描请求
SCAN_RSP	主动扫描响应
CONNECT_REQ	连接请求
ADV_SCAN_IND	可扫描指示
Reserved	保留

图 4-4：广播报文类型

每种广播报文类型都具有不同的数据格式及行为。*Core_v4.2 的 2584 页的 2.3.1 节详细的描述了各个广播报文类型，大家可以阅读此章节进一步了解。*

2. 发送地址类型和接收地址类型

发送地址类型和接收地址类型指示了设备使用公共地址(Public Address)还是随机地址(Random Address)。公共地址和随机地址的长度一样，都包含 6 个字节共 48 位。BLE 设备至少要拥有这两种地址类型中的一种，当然也可以同时拥有这两种地址类型。关于公共地址和随机地址的详细描述请阅读本章的第 6 节《设备地址》。

3.4. 长度

- 广播报文：长度域包含 6 个比特，有效值的范围是 6~37。
- 数据报文：长度域包含 5 个比特，有效值的范围是 0~31。

广播报文和和数据报文的长度域有所不同，主要原因是：广播报文除了最多 31 个字节的数据之外，还必须要包含 6 个字节的广播设备地址。6+31=37，所以需要 6 比特的长度域。

3.5. 数据(AdvData)

广播和扫描响应的数据格式如下图所示，由有效数据部分和无效数据部分组成。

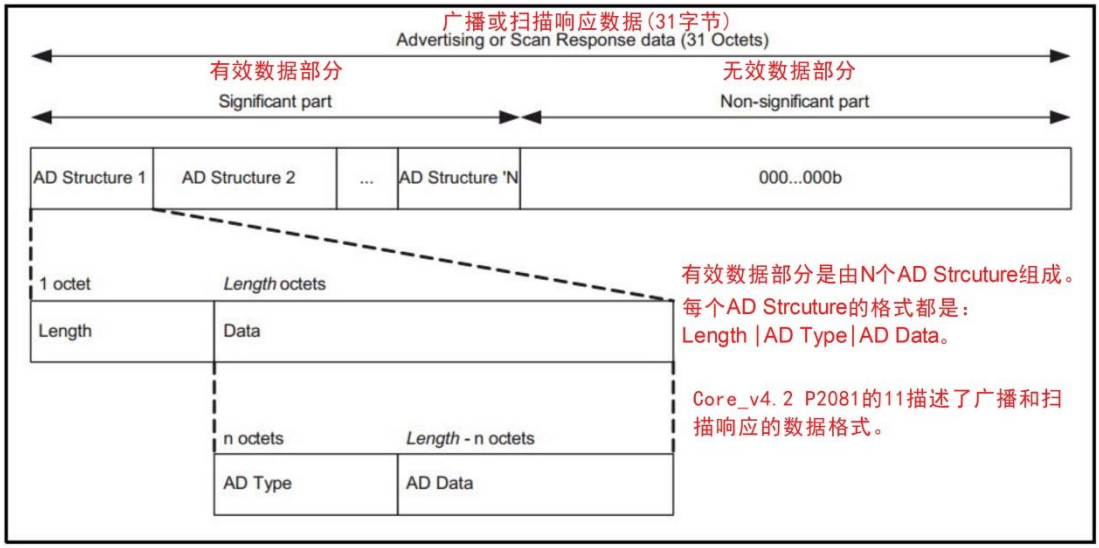


图 4-5：广播和扫描响应的数据格式

- 有效数据部分：包含 N 个 AD Structure，每个 AD Structure 由 Length，AD Type 和 AD Data 组成。其中：
  - Length：AD Type 和 AD Data 的长度。
  - AD Type：指示 AD Data 数据的含义。
  - AD Data：数据。

我们应该记住 AD Structure 的结构，这对我们理解广播数据以及扫描后的数据解析非常重要。主机扫描到广播后，如何知道广播中包含的数据的意义？它就是逐个解析广播包中的 AD Structure 来获取对应的数据的。

- 问题来了，我们怎么知道有哪些 AD Type？他们又表示什么意义？

  我们可以通过下面 2 种方式查看 AD Type 和他们表示的意义。
  - 从 SIG 官网查询，但是需要是会员才可以查询。

  https://www.bluetooth.org/Technical/AssignedNumbers/generic_access_profile.htm
  - 查看 Nordic 的 SDK 中的定义，AD type 的定义在程序的“ble_gap.h”头文件中。定义如下：

```
#define BLE_GAP_AD_TYPE_FLAGS                        0x01 /**< Flags for discoverability. */
#define BLE_GAP_AD_TYPE_16BIT_SERVICE_UUID_MORE_AVAILABLE   0x02 /**< Partial list of 16 bit
service UUIDs. */
#define BLE_GAP_AD_TYPE_16BIT_SERVICE_UUID_COMPLETE        0x03 /**< Complete list of 16 bit
service UUIDs. */
#define BLE_GAP_AD_TYPE_32BIT_SERVICE_UUID_MORE_AVAILABLE   0x04 /**< Partial list of 32 bit
service UUIDs. */
#define BLE_GAP_AD_TYPE_32BIT_SERVICE_UUID_COMPLETE        0x05 /**< Complete list of 32 bit
service UUIDs. */
#define BLE_GAP_AD_TYPE_128BIT_SERVICE_UUID_MORE_AVAILABLE  0x06 /**< Partial list of 128 bit
```

```
service UUIDs. */
#define BLE_GAP_AD_TYPE_128BIT_SERVICE_UUID_COMPLETE       0x07 /**< Complete list of 128 bit
service UUIDs. */
#define BLE_GAP_AD_TYPE_SHORT_LOCAL_NAME         0x08 /**< Short local device name. */
#define BLE_GAP_AD_TYPE_COMPLETE_LOCAL_NAME      0x09 /**< Complete local device name. */
#define BLE_GAP_AD_TYPE_TX_POWER_LEVEL           0x0A /**< Transmit power level. */
#define BLE_GAP_AD_TYPE_CLASS_OF_DEVICE          0x0D /**< Class of device. */
#define BLE_GAP_AD_TYPE_SIMPLE_PAIRING_HASH_C    0x0E /**< Simple Pairing Hash C. */
#define BLE_GAP_AD_TYPE_SIMPLE_PAIRING_RANDOMIZER_R   0x0F /**< Simple Pairing Randomizer R. */
#define BLE_GAP_AD_TYPE_SECURITY_MANAGER_TK_VALUE      0x10 /**< Security Manager TK Value. */
#define BLE_GAP_AD_TYPE_SECURITY_MANAGER_OOB_FLAGS          0x11 /**< Security Manager Out Of
Band Flags. */
#define BLE_GAP_AD_TYPE_SLAVE_CONNECTION_INTERVAL_RANGE     0x12 /**< Slave Connection Interval
Range. */
#define BLE_GAP_AD_TYPE_SOLICITED_SERVICE_UUIDS_16BIT      0x14 /**< List of 16-bit Service
Solicitation UUIDs. */
#define BLE_GAP_AD_TYPE_SOLICITED_SERVICE_UUIDS_128BIT     0x15 /**< List of 128-bit Service
Solicitation UUIDs. */
#define BLE_GAP_AD_TYPE_SERVICE_DATA                0x16 /**< Service Data - 16-bit UUID. */
#define BLE_GAP_AD_TYPE_PUBLIC_TARGET_ADDRESS       0x17 /**< Public Target Address. */
#define BLE_GAP_AD_TYPE_RANDOM_TARGET_ADDRESS       0x18 /**< Random Target Address. */
#define BLE_GAP_AD_TYPE_APPEARANCE                  0x19 /**< Appearance. */
#define BLE_GAP_AD_TYPE_ADVERTISING_INTERVAL        0x1A /**< Advertising Interval. */
#define BLE_GAP_AD_TYPE_LE_BLUETOOTH_DEVICE_ADDRESS 0x1B /**< LE Bluetooth Device Address. */
#define BLE_GAP_AD_TYPE_LE_ROLE                        0x1C /**< LE Role. */
#define BLE_GAP_AD_TYPE_SIMPLE_PAIRING_HASH_C256       0x1D /**< Simple Pairing Hash C-256. */
#define BLE_GAP_AD_TYPE_SIMPLE_PAIRING_RANDOMIZER_R256  0x1E /**< Simple Pairing Randomizer
R-256. */
#define BLE_GAP_AD_TYPE_SERVICE_DATA_32BIT_UUID    0x20 /**< Service Data - 32-bit UUID. */
#define BLE_GAP_AD_TYPE_SERVICE_DATA_128BIT_UUID   0x21 /**< Service Data - 128-bit UUID. */
#define BLE_GAP_AD_TYPE_URI                        0x24 /**< URI */
#define BLE_GAP_AD_TYPE_3D_INFORMATION_DATA        0x3D /**< 3D Information Data. */
#define BLE_GAP_AD_TYPE_MANUFACTURER_SPECIFIC_DATA  0xFF /**< Manufacturer Specific Data. */
```

3.6. 校验

BLE 采用的是 24 位 CRC 校验。CRC 对报头、长度和数据进行计算。24 位 CRC 的生成多项式如下：

$$CRC = x^{24} + x^{10} + x^{9} + x^{6} + x^{4} + x^{3} + x^{1} + x^{0}$$

理解了广播报文的格式后，我们还需要知道广播报文中可以包含哪些数据。接下来，我们看一下 SDK 中定义了哪些数据可以包含到广播报文中。

4. 广播可以包含的数据

SDK 里面定义了广播数据结构体 ble_advdata_t，该结构体描述了广播可以包含的数据，ble_advdata_t 结构体声明如下。

代码清单： 广播数据结构体声明

```
typedef struct {
    ble_advdata_name_type_t name_type; //设备名称类型
    uint8_t short_name_len;  //裁剪的设备名称的长度
    bool include_appearance; //是否包含外观
    uint8_t flags;           //标志
    int8_t * p_tx_power_level; //发射功率等级，指的是传输该广播包时的发射功率
    ble_advdata_uuid_list_t    uuids_more_available; //服务 UUID 部分列表
    ble_advdata_uuid_list_t    uuids_complete;       //服务 UUID 完整列表
    ble_advdata_uuid_list_t    uuids_solicited;      //服务请求 UUID 列表
    ble_advdata_conn_int_t     * p_slave_conn_int;   //从机连接间隔范围
    ble_advdata_manuf_data_t   * p_manuf_specific_data; //制造商自定义数据
    ble_advdata_service_data_t * p_service_data_array;  //服务数据
    uint8_t service_data_count;            //服务数据的长度(字节数)
    bool include_ble_device_addr;          //是否包含设备地址
    ble_advdata_le_role_t le_role;         //仅用于 NFC，BLE 广播设置为 NULL
    ble_advdata_tk_value_t * p_tk_value;   //仅用于 NFC，BLE 广播设置为 NULL
    uint8_t * p_sec_mgr_oob_flags;         //仅用于 NFC，BLE 广播设置为 NULL
    ble_gap_lesc_oob_data_t * p_lesc_data; //仅用于 NFC，BLE 广播设置为 NULL
} ble_advdata_t;
```

我们将广播数据结构体 ble_advdata_t 整理如下表，这样阅读更方便。下表中的数据类型都是可以加入到广播报文中的，一般地，建议在广播数据中至少包含设备名称、Flags、外观和首要服务的 UUID 列表。

表 4-1：广播数据结构体中定义的数据类型

数据类型	变量名称	描述
设备名称	name_type	名称类型（裁剪的设备名称或全称）。
	short_name_len	裁剪的设备名称长度，使用全称时忽略。
外观	include_appearance	外观包含标志。=true：广播数据中包含外观，否则不包含。
Flags	flags	标志。
发射功率等级	p_tx_power_level	指向发射功率等级，默认为 NULL，即广播数据中不包含发射功率等级。
服务数据	p_service_data_array	指向服务数据，默认为 NULL，即广播数据中

		不包含服务数据。
	service_data_count	加入广播的服务数据的数量。
从设备连接间隔	p_slave_conn_int	指向从设备连接间隔，默认为 NULL，即广播数据中不包含从设备连接间隔。
服务的 UUID	uuids_more_available	广播中包含完整的服务 UUID 列表。
	uuids_complete	广播中包含部分的服务 UUID 列表。
服务请求 UUID 列表	uuids_solicited	服务请求 UUID 列表。
厂商自定义数据	p_manuf_specific_data	指向厂商自定义数据，默认为 NULL，即广播数据中不包含厂商自定义数据。
包含设备地址	include_ble_device_addr	包含设备地址标志。=true：广播数据中包含设备地址，否则不包含。 我们知道，广播报文是 31 个字节（数据）+6 个字节（设备地址），6 个字节的设备地址是必须包含的，这里设置的是 31 个字节数据中是否再次包含设备地址。

到这里，我们对广播及其报文有了总体了解，接下来，我们详细看一下各个数据类型的意义以及在程序里面如何将它们加入到广播报文中。

5. 设备地址

BLE 设备地址可以使用公共地址（Public Device Address）或随机地址（Random Device Address）两种地址类型，一个 BLE 至少使用一种地址类型，当然也可以同时具备两种地址类型。公共地址和随机地址的长度一样，都是 48 位(6 个字节)的。BLE 设备地址类型的关系如下图所示。

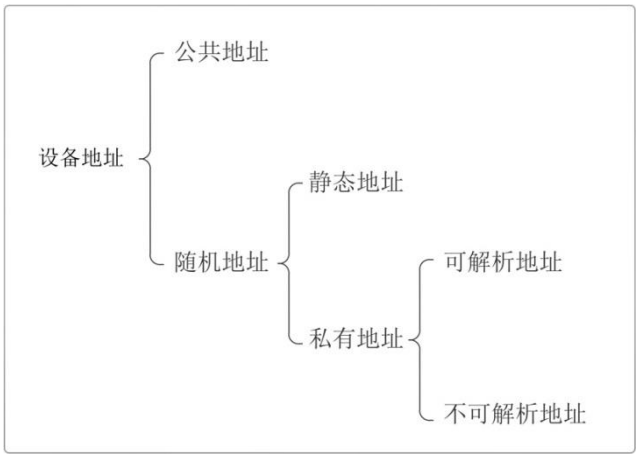


图 4-6：设备地址类型关系

从上图中设备地址的关系我们可以看到，设备地址一共有如下 4 种类型：

- 公共地址：从 IEEE 申请（购买），IEEE 保证地址分发的唯一性。
- 随机静态地址（Static Device Address）：自己定义，上电初始化完成后不能修改。
- 随机不可解析私有地址（Non-resolvable private address）：定时更新地址，蓝牙核心规范建议 15 分钟更新一次。
- 随机可解析私有地址（RPA：Resolvable Private Address）：通信双方使用共享的身份解析密钥（IRK：Identity Resolving Key），生成和解析可解析私有地址。只有一台设备拥有另一台广播设备的 IRK 时，才能跟踪该广播设备的活动。

5.1. 公共地址

公共地址由两部分组成，如下图所示。公共地址由制造商从 IEEE 申请（购买），由 IEEE 注册机构为该制造商分配的机构唯一标识符 OUI(Organizationally Unique Identifier)。这个地址是独一无二，不能修改的。

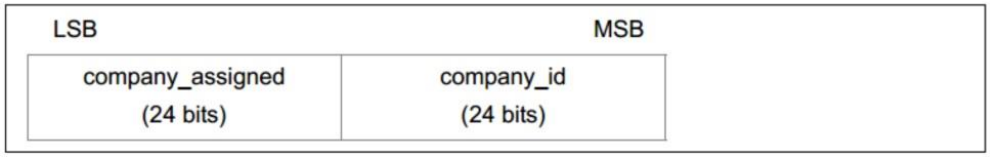


图 4-7：公共地址格式

公共地址能明确的指示出设备，同时具有唯一性，但是安全度不够，试想当我们携带使用公共地址的 BLE 设备时，不法分子可以通过技术手段跟踪该唯一的公共地址，即可跟踪到该 BLE 设备的使用者。

为了加强隐私保护，蓝牙核心规范中提供了另外一种地址：随机地址，随机地址是随机产生的，不是固定分配的，随机地址又分为多种类型，以适应不同的应用场景对隐私的需求。

5.2. 随机地址

5.2.1. 随机静态地址

1. 随机静态地址的定义

随机静态地址是随机生成的 48 位地址，随机静态地址必须符合以下要求：

- 静态地址的最高 2 位有效位必须是 1。
- 静态地址最高 2 位有效位之外的其余部分不能全为 0。
- 静态地址最高 2 位有效位之外的其余部分不能全为 1。
- 一个上电周期内不变。

静态地址的格式如下图，共 48 位，最高 2 位是固定的，必须是 1。

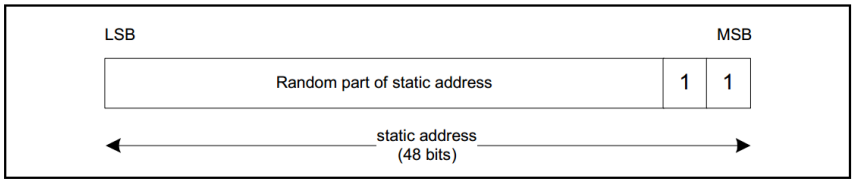


图 4-8：静态地址格式

2. 代码中使用随机静态地址的方法

程序中使用随机静态地址时，可以每次上电时重新设置随机地址，但是初始化完成后，不能修改地址。设置随机静态地址和读取随机静态地址的流程如下图所示。

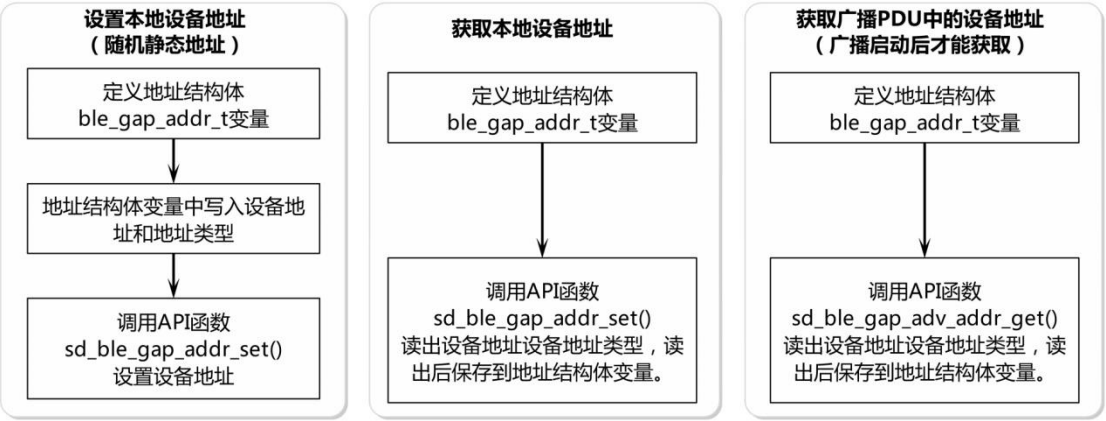


图 4-9：设置和读取静态地址流程

从上图中我们可以看到，读写随机静态地址需要用到地址结构体 ble_gap_addr_t 和两个 API 函数。其中地址结构体 ble_gap_addr_t 的声明如下。读和写的时候，都需要定义地址结构体，读的时候用来保存读取的设备地址和设备地址类型，写的时候用来将待写入的地址传递给 API 函数 sd_ble_gap_addr_set()。

代码清单：地址结构体 ble_gap_addr_t 声明

```
typedef struct
{
  //该位由 SoftDevice 设置，用来指示可解析私有地址是否已经被解析（当对端设备使用隐私时）
  //作为 API 函数的输入参数时该位忽略
  uint8_t addr_id_peer : 1;
  uint8_t addr_type    : 7;        //地址类型
  uint8_t addr[BLE_GAP_ADDR_LEN]; //48 位设备地址存放数组，LSB 格式.
} ble_gap_addr_t;
```

读写设备地址使用的两个 API 函数 sd_ble_gap_addr_set()和 sd_ble_gap_addr_get()原型如下。

表 4-2：sd_ble_gap_addr_set()函数

函数原型	uint32_t sd_ble_gap_addr_set(ble_gap_addr_t const * p_addr)
函数功能	设置本地设备地址。 地址类型必须是：BLE_GAP_ADDR_TYPE_PUBLIC 或者 BLE_GAP_ADDR_TYPE_RANDOM_STATIC。 注意：正在执行广播、扫描或创建连接时不能改变设备地址。设备地址在绑定时会分发给对端设备，如果地址改变了，存储在对端设备中的地址将无效，并且对端设备将失去使用旧地址重新连接的能力。
参数	[in] p_addr：指向保存设备地址的地址结构体变量。

返回值	• NRF_SUCCESS：设备地址设置成功。 • NRF_ERROR_INVALID_ADDR：提供了无效的指针。 • BLE_ERROR_GAP_INVALID_BLE_ADDR：提供无效的设备地址。 • NRF_ERROR_BUSY：协议栈忙，处理挂起的事件后重试。 • NRF_ERROR_INVALID_STATE：设置处于不能修改地址的状态（正在执行广播、扫描或创建连接）。

表 4-3：sd_ble_gap_addr_get()函数

函数原型	uint32_t sd_ble_gap_addr_get(ble_gap_addr_t const * p_addr)
函数功能	获取本地蓝牙设备地址。 注意：无论隐私设置如何，该函数始终返回设备地址。也就是说地址类型总是返回 BLE_GAP_ADDR_TYPE_PUBLIC 或 BLE_GAP_ADDR_TYPE_RANDOM_STATIC。
参数	[out] p_addr：指向保存获取的设备地址的地址结构体变量。
返回值	• NRF_SUCCESS：获取设备地址成功。 • NRF_ERROR_INVALID_ADDR：提供了无效的或指向 NULL 的指针。

表 4-4：sd_ble_gap_adv_addr_get()函数

函数原型	uint32_t sd_ble_gap_adv_addr_get(uint8_t adv_handle, ble_gap_addr_t * p_addr)
函数功能	获取广播者使用的蓝牙设备地址。 注意此函数将返回广播 PDU 中的蓝牙地址。当使用不可解析私有地址时，SoftDevice 会按照设置的间隔周期性地生成一个新的私有地址，因此，调用此 API 返回的地址可能不是广播 PDU 中使用的最新地址。
参数	[in] adv_handle：广播句柄。 [out] p_addr：指向保存获取的设备地址的地址结构体变量。
返回值	• NRF_SUCCESS：获取设备地址成功。 • NRF_ERROR_INVALID_ADDR：提供了无效的或指向 NULL 的指针。 • BLE_ERROR_INVALID_ADV_HANDLE：查找不到提供的广播句柄。 • NRF_ERROR_INVALID_STATE：该广播集当前没有在广播。

■ **写设备地址示例**

定义地址结构体 my_addr，然后设置需要写入的设备地址，并将地址类型设置为随机静态地址 BLE_GAP_ADDR_TYPE_RANDOM_STATIC。注意静态地址最高 2 个位必须都为 1。下面的代码中设置的地址用二进制表示如下，最高 2 个位都是 1（红色字体）：

11001100 01010101 01000100 00110011 00100010 00010001

代码清单：写设备地址示例代码

```
1.  //定义地址结构体变量 my_addr
2.  static ble_gap_addr_t  my_addr;
3.  //初始化地址和地址类型
4.  my_addr.addr[0] = 0x11;
5.  my_addr.addr[1] = 0x22;
6.  my_addr.addr[2] = 0x33;
7.  my_addr.addr[3] = 0x44;
8.  my_addr.addr[4] = 0x55;
9.  //注意地址最高两个位必须为 1，其它所有的位不能同时为 0，也不能同时为 1
10. my_addr.addr[5] = 0xcc;
11. my_addr.addr_type = BLE_GAP_ADDR_TYPE_RANDOM_STATIC;//地址类型设置为随机静态地址
12. err_code = sd_ble_gap_addr_set(&my_addr); //写入设备地址
```

■ **读设备地址示例**

定义地址结构体 my_addr，用来保存读取的设备地址和类型，然后调用 API 函数 sd_ble_gap_addr_get()读取设备地址，读取成功后，设置地址会保存到 my_addr。

代码清单：读设备地址示例代码

```
1.  //定义地址结构体变量 my_addr,用来保存读取的设备地址和类型
2.  static ble_gap_addr_t  my_addr;
3.  //读取设备地址，读取成功后，设备地址保存到 my_addr
4.  err_code = sd_ble_gap_addr_get(&my_addr);
```

■ **读取广播 PDU 中的设备地址示例**

在广播启动后读取，定义地址结构体 my_addr，用来保存读取的设备地址和类型，然后调用 API 函数 sd_ble_gap_adv_addr_get()读取设备地址，读取成功后，设备地址会保存到 my_addr。

代码清单：读广播 PDU 中的设备地址示例代码

```
1.  //定义地址结构体变量 my_addr
2.  static ble_gap_addr_t  my_addr;
3.  //获取广播 PDU 中的设备地址
4.  err_code = sd_ble_gap_adv_addr_get(m_advertising.adv_handle,&my_addr);
```

3. 读取芯片出厂时设置的地址

如果我们做过 SDK 里面的 BLE 的实验，可以看到 Nordic 的 SDK 提供的 BLE 例子里面都没有设置设备地址，那么这些例子使用的什么类型的地址，地址从哪儿来的？

nRF52840 芯片在出厂时已经固化了 48 位地址和地址类型（地址类型为随机静态地址），应用程序可以从下面两个寄存器读出地址类型和地址。

- DEVICEADDRTYPE 寄存器（只读寄存器）：地址类型，位 0=1，即随机静态地址。

Bit number				31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Id				A
Reset 0xFFFFFFFF				**1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1**
Id	**RW**	**Field**	**Value Id**	**Value**	**Description**
A	R	DEVICEADDRTYPE			Device address type
			Public	0	Public address
			Random	1	Random address

图 4-10：地址类型寄存器

- DEVICEADDR[n]寄存器（只读寄存器）：包含 DEVICEADDR[0]和 DEVICEADDR[1]两个寄存器，保存的是 48 位设备地址。DEVICEADDR[0]的 0~31 位和 DEVICEADDR[1]的 0~15 位组成 48 位地址。

Bit number				31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Id				A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A
Reset 0xFFFFFFFF				**1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1**
Id	**RW**	**Field**	**Value Id**	**Value**	**Description**
A	R	DEVICEADDR			48 bit device address DEVICEADDR[0] contains the least significant bits of the device address. DEVICEADDR[1] contains the most significant bits of the device address. Only bits [15:0] of DEVICEADDR[1] are used.

图 4-11：地址寄存器

SDK 中的 BLE 例子使用的设备地址是初始化时协议栈从 DEVICEADDR[n]寄存器中读出来的，DEVICEADDR[n]寄存器中存储的是随机静态地址，所以 BLE 例子中使用的都是随机静态地址。

由此可见，如果应用程序不设置设备地址的话，设备就会使用协议栈默认从芯片读取的设备地址，地址类型为随机静态地址。

5.2.2. 不可解析私有地址

1. 不可解析私有地址的定义

设备生成不可解析私有地址时必须符合以下要求：

- 地址的最高 2 位有效位必须是 0。
- 地址最高 2 位有效位之外的其余部分不能全为 0。
- 地址最高 2 位有效位之外的其余部分不能全为 1。
- 不可解析私有地址不能和公共地址一样。

不可解析私有地址的格式如下图，共 48 位，最高 2 位是固定的，必须是 0。

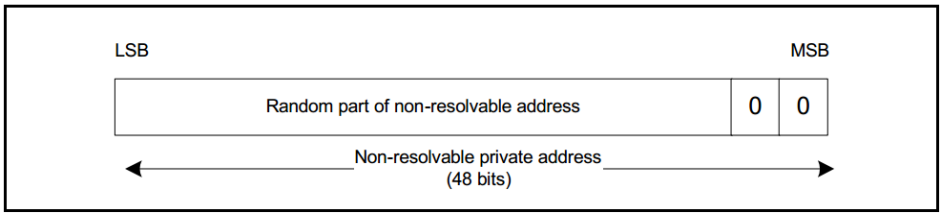


图 4-12：不可解析私有地址格式

不可解析私有地址相当于周期性改变的随机静态地址，不可解析私有地址一直在变化，并且该地址是个随机数，没有提供任何可解析的信息，因此，很难通过跟踪地址来跟踪设备，所以具有很高的安全性。但是因为地址一直变化，又没有可解析的信息，这就导致受信任的设备也没法分辨该地址的真实身份。

由此可见，不可解析私有地址在隐私上“敌我”不分，不管是谁，统统让你无法分辨我的真实身份。因此不可解析私有地址在实际应用中使用的不多。

2. 代码中使用不可解析私有地址的方法

程序中使用不可解析私有地址时，只需要配置地址模式、地址类型和地址循环周期即可，协议栈会自动生成随机的不可解析私有地址。设置不可解析私有地址的流程如下图所示，注意，设置私有地址时使用的结构体是私有地址结构体 ble_gap_privacy_params_t，使用的 API 函数是 sd_ble_gap_privacy_set()，这和设置随机静态地址是不一样的。

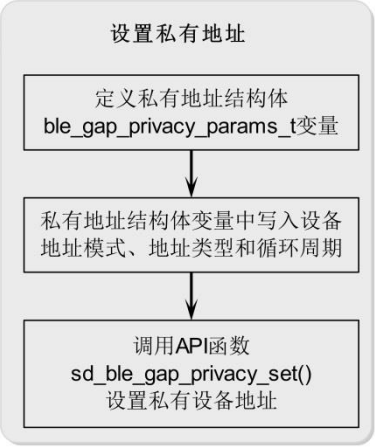


图 4-13：设置私有地址流程

从上图中我们可以看到，写私有地址需要用到私有地址结构体 ble_gap_privacy_params_t 和 API 函数 sd_ble_gap_privacy_set()。其中私有地址结构体 ble_gap_privacy_params_t 的声明如下。

代码清单：私有地址结构体 ble_gap_privacy_params_t 声明

```
typedef struct
{
  //私有地址模式，默认设置的是：BLE_GAP_PRIVACY_MODE_OFF
  uint8_t        privacy_mode;
  //私有地址类型，只能是 BLE_GAP_ADDR_TYPE_RANDOM_PRIVATE_RESOLVABLE 或
  //BLE_GAP_ADDR_TYPE_RANDOM_PRIVATE_NON_RESOLVABLE
  uint8_t        private_addr_type;
  //私有地址循环周期，单位是秒。
  uint16_t       private_addr_cycle_s;
  //指向身份解析密钥
  ble_gap_irk_t *p_device_irk;
} ble_gap_privacy_params_t;
```

私有地址使用 API 函数 sd_ble_gap_privacy_set()设置，该函数原型如下。

表 4-5：sd_ble_gap_privacy_set()函数

函数原型	uint32_t sd_ble_gap_privacy_set(ble_gap_privacy_params_t const * p_privacy_params)

函数功能	设置私有地址。 注意：正在执行广播、扫描或创建连接时不能改变私有地址配置。
参　　数	[in] p_privacy_params：指向私有地址配置结构体变量。
返回值	NRF_SUCCESS：设备地址设置成功。 NRF_ERROR_BUSY：协议栈忙，处理挂起的事件并重试。 BLE_ERROR_GAP_INVALID_BLE_ADDR：提供无效的设备地址。 NRF_ERROR_INVALID_ADDR：指向隐私配置的指针为 NULL 或无效。 又或者，privacy 参数中的 p_device_irk 指针是无效指针。 NRF_ERROR_INVALID_PARAM：提供的参数超出了范围。 NRF_ERROR_INVALID_STATE：设置处于不能修改地址的状态（正在执行广播、扫描或创建连接）。。

■ **私有地址设置示例**

定义私有地址结构体 my_addr，并初始化其成员变量如下。

- 模式设置为 BLE_GAP_PRIVACY_MODE_DEVICE_PRIVACY，
- 类型设置为 BLE_GAP_ADDR_TYPE_RANDOM_PRIVATE_NON_RESOLVABLE
- 循环周期：根据自己的需求设置，蓝牙核心规范推荐值是 15 分钟。
- IRK：设置为 NULL。

然后调用 API 函数 sd_ble_gap_privacy_set()设置地址，设置成功后，协议栈会自动成功不可解析私有地址，并且地址每 15 秒改变一次。

代码清单：设置私有地址示例代码

```
//定义私有地址结构体变量 my_addr
static ble_gap_privacy_params_t  my_addr = {0};
//初始化地址模式、地址类型和循环周期，私有地址由协议栈自动生成
my_addr.privacy_mode = BLE_GAP_PRIVACY_MODE_DEVICE_PRIVACY;
my_addr.private_addr_type = BLE_GAP_ADDR_TYPE_RANDOM_PRIVATE_NON_RESOLVABLE;
//地址循环周期设置为 15 秒。蓝牙核心规范推荐值是 15 分钟。
my_addr.private_addr_cycle_s = 15;
my_addr.p_device_irk = NULL;

//写入私有地址
err_code = sd_ble_gap_privacy_set(&my_addr);
```

5.2.3. 可解析私有地址

设备生成可解析私有地址时必须符合以下要求：

- 地址的最高 2 位有效位必须是 0 和 1。
- Prand 中地址最高 2 位有效位之外的其余部分不能全为 0。

- Prand 中地址最高 2 位有效位之外的其余部分不能全为 1。
- 可解析私有地址不能和公共地址一样。

不可解析私有地址的格式如下图，共 48 位，最高 2 位是固定的，必须是 0。

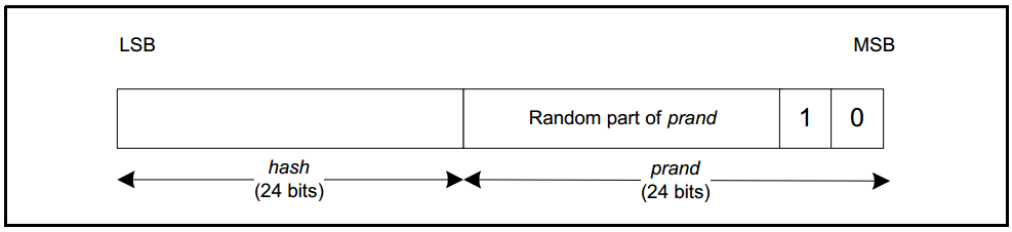


图 4-14：可解析私有地址格式

代码中使用可解析私有地址的方法和不可解析私有地址类似，参考不可解析私有地址的章节即可。对于可解析私有地址，在绑定教程之后再深入学习更合适。

5.3. 随机静态地址读写实验

本实验在“实验 3-3：BLE 工程模板（外围设备）”的基础上修改。增加了设置随机静态地址和读出设备地址的代码。

✧ 注：本节对应的实验源码是：“实验 4-1：读设备地址、写随机静态设备地址”。

5.3.1. 代码编写

本例中使用函数 sd_ble_gap_addr_set()设置随机静态地址为“CC 55 44 33 22 11”，设置成功后，再使用函数 sd_ble_gap_addr_get()读出地址并打印出读出的设备地址和类型。

代码清单： 设置随机静态地址和读出设备地址

```
//定义地址结构体变量 my_addr
static ble_gap_addr_t  my_addr;
/*--------------------------设置设备地址:随机静态地址----------------------*/
//初始化地址和地址类型
my_addr.addr[0] = 0x11;
my_addr.addr[1] = 0x22;
my_addr.addr[2] = 0x33;
my_addr.addr[3] = 0x44;
my_addr.addr[4] = 0x55;
//注意地址最高两个位必须为 1，其它所有的位不能同时为 0，也不能同时为 1
my_addr.addr[5] = 0xcc;
//地址类型设置为随机静态地址
my_addr.addr_type = BLE_GAP_ADDR_TYPE_RANDOM_STATIC;
//写入地址
err_code = sd_ble_gap_addr_set(&my_addr);//
if(err_code != NRF_SUCCESS)
{
```

```
    //打印设置地址失败
    NRF_LOG_INFO("Set Address Failed!");
}
/*--------------------------------END--------------------------------*/

/*-----------------------读出设备地址和地址类型-----------------------*/
err_code = sd_ble_gap_addr_get(&my_addr);
//地址读取成功，串口打印出地址
if(err_code == NRF_SUCCESS)
{
    //打印地址类型
    NRF_LOG_INFO("Address Type: %02X\r\n",my_addr.addr_type);
    //打印地址
    NRF_LOG_INFO("Address: %02X:%02X:%02X:%02X:%02X:%02X\r\n",
                                    my_addr.addr[0], my_addr.addr[1],
                                    my_addr.addr[2], my_addr.addr[3],
                                    my_addr.addr[4], my_addr.addr[5]);
}
/*--------------------------------END--------------------------------*/
```

5.3.2. 实验步骤

1. 解压“…\4：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 4-1：读设备地址、写随机静态设备地址”，将解压后得到的文件夹“ble_app_templete”拷贝合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_templete\project\mdk5”目录下的工程“ble_app_templete_s140.uvproj”。
4. 切换到 Target“flash_s140_nrf52_7.0.1_softdevice”，点击下载协议栈。
5. 切换到 Target“IK-52840DK”，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
6. 程序运行后，指示灯 D1 闪烁表示正在广播。
7. 手机上打开 nRF Connect，点击“SCAN”按钮启动扫描，这时会扫描到名为“BLE_Template”的设备，其设备地址为“CC 55 44 33 22 11”，和我们设置的地址一样，如下图所示。

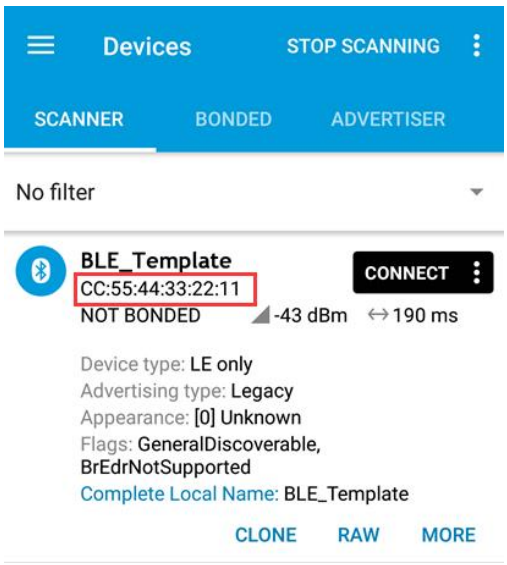


图 4-15：查看设备地址

同时，程序启动时也会通过串口打印出读取的设备地址和类型，如下图所示。

图 4-16：串口打印设备地址和类型

5.4. 设置私有地址实验

本实验在“实验 3-3：BLE 工程模板（外围设备）”的基础上修改。增加了设置私有地址的代码。

✧ 注：本节对应的实验源码是：“实验 4-2：写私有设备地址”。

5.4.1. 代码编写

本例中使用函数 sd_ble_gap_privacy_set()设置私有地址，为了方便观察地址的变化，地址循环周期设置为 15 秒（蓝牙核心规范推荐值是 15 分钟）。代码中的 my_addr.private_addr_type 是私有地址的类型，使用不可解析私有地址和可解析私有地址时设置的值如下。

- BLE_GAP_ADDR_TYPE_RANDOM_PRIVATE_NON_RESOLVABLE：不可解析私有地址。
- BLE_GAP_ADDR_TYPE_RANDOM_PRIVATE_RESOLVABLE：可解析私有地址。

代码清单： 设置随机静态地址和读出设备地址

```
//定义私有地址结构体变量 my_addr
static ble_gap_privacy_params_t  my_addr = {0};
//初始化地址模式、类型和循环周期，私有地址不需要应用程序提供，协议栈会自动生成私有地址
my_addr.privacy_mode = BLE_GAP_PRIVACY_MODE_DEVICE_PRIVACY;
//地址类型为不可解析私有地址，如果是可解析私有地址，这里要设置为：
//BLE_GAP_ADDR_TYPE_RANDOM_PRIVATE_RESOLVABLE
my_addr.private_addr_type = BLE_GAP_ADDR_TYPE_RANDOM_PRIVATE_NON_RESOLVABLE;
//为了方便观察地址变化，这里地址循环周期设置为 15 秒，蓝牙核心规范推荐值是 15 分钟。
my_addr.private_addr_cycle_s = 15;
my_addr.p_device_irk = NULL;

//写入私有地址
err_code = sd_ble_gap_privacy_set(&my_addr);
if(err_code != NRF_SUCCESS)
{
    //打印设置地址失败
    NRF_LOG_INFO("Set Address Failed!");
}
```

5.4.2. 实验步骤

1. 解压“…\4：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 4-2：写私有地址”，将解压后得到的文件夹“ble_app_templete”拷贝合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_templete\project\mdk5”目录下的工程“ble_app_templete_s140.uvproj”。
4. 切换到 Target“flash_s140_nrf52_7.0.1_softdevice”，点击下载协议栈。
5. 切换到 Target“IK-52840DK”，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
6. 程序运行后，指示灯 D1 闪烁表示正在广播。
7. 手机上打开 nRF Connect，点击“SCAN”按钮启动扫描，这时会扫描到名为“BLE_Template”的设备，我们可以观察到每隔 15 秒，设备地址变化一次，如下图所示。因为写入的是不可解析私有地址，所以地址的最高 2 位都是 0。

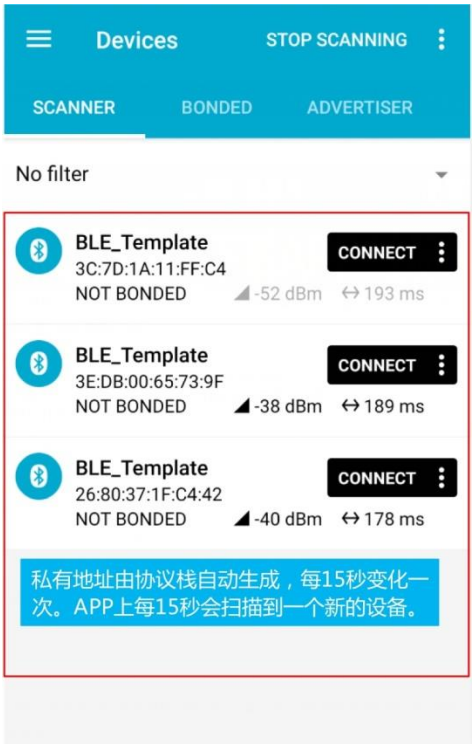


图 4-17：不可解析私有地址

6. 本地设备名称

本地设备名称(Local Name)有如下 2 种类型。

- 完整的本地名称(Complete local name)：不能截断。
- 裁剪的本地名称(Shortened local name)：可以被截断，截取的内容必须是从名称的起始位置开始连续的字符。

■ **设备名称示例：**

完整的本地设备名称是：BT_Device_Name，裁剪的本地设备名称可以是 BT_Device，也可以是 BT_Dev 等，但不能是 T_DEV（因为 T_DEV 不是从名称的起始位置开始的）。

■ **应用场景：**

- 完整的本地名称：设备愿意对外展示全部的设备名称，并且设备名称不长，广播包可以容纳。
- 裁剪的本地名称：设备名称很长，一个广播包无法容纳，或者我们不想在广播的时候对外展示全部的设备名称，这时可以使用裁剪的设备名称，当连接建立后，主机可以通过读 GAP 设备名称特征来获取完整的设备名称。

虽然裁剪的设备名称用的少，但是很有用。如设备名称：IKMSIK-Uart-V2.01-20171121，可以使用裁剪的设备名称，长度设置为 11，那么广播的设备名称是：IKMSIK-Uart，当需要全部名称的时候，可以连接之后查询设备名称特征值，获取全部设备名称。

6.1. 使用本地名称的步骤

程序中使用本地名称的步骤如下图所示，首先在初始化 GAP 的函数里面设置 GAP 服务的设备名称特征（使用 API 函数 sd_ble_gap_device_name_set()设置），之后在初始化广播时

设置名称类型，也就是广播包中包含设备名称的方式，可设置的类型如下：

- 不包含设备名称：名称类型设置为 BLE_ADVDATA_NO_NAME。
- 包含全称：名称类型设置为 BLE_ADVDATA_FULL_NAME。
- 包含裁剪的设备名称：名称类型设置为 BLE_ADVDATA_SHORT_NAME，同时还要设置裁剪的长度。

设置好名称类型之后，广播初始化时就知道了如何将 GAP 服务的设备名称特征字符串“拷贝”到广播包中，如名称类型设置为全称，那么广播初始化时会将 GAP 服务的设备名称特征字符串全部“拷贝”过来。

广播包中包含设备名称时，还需要注意广播包的剩余空间是否能够容纳设备名称，如果无法容纳，设备名称也会被截断。这也是很多开发者遇到过的问题：明明设置的是全称，为什么扫描显示的只是部分的设备名称？原因就是因为广播包的剩余空间无法容纳全部的设备名称，设备名称被截断了。

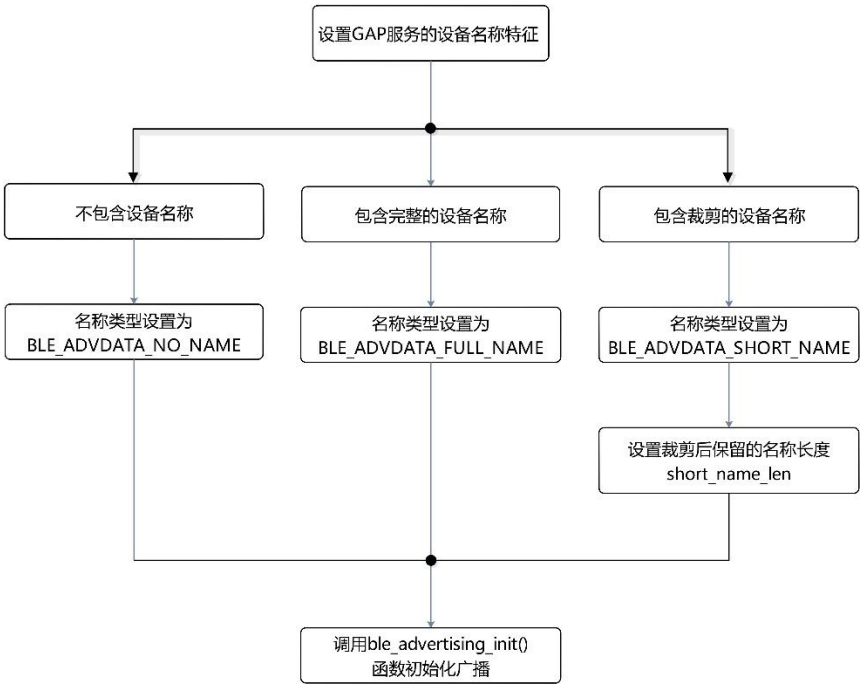


图 4-19：配置设备名称流程

6.2. 广播中加入设备名称实验

本实验在“实验 3-3：BLE 工程模板（外围设备）”的基础上修改。

BLE 工程模板里面使用的设备名称是全称，本实验在广播初始化函数里面增加了不使用设备名称和使用裁剪的设备名称的代码。

✧ 注：本节对应的实验源码是：“实验 4-3-1：广播中加入设备名称”。

6.2.1. 代码编写

本例在广播初始化函数 advertising_init()中加入了广播中不包含设备名称和包含裁剪的设备名称的代码（工程模板默认使用的是全称），代码清单如下：

代码清单： 配置设备名称

```
//设备名称类型：不包含设备名称
//init.advdata.name_type                = BLE_ADVDATA_NO_NAME;

//设备名称类型：全称
init.advdata.name_type               = BLE_ADVDATA_FULL_NAME;

//设备名称类型：使用裁剪的设备名称,使用裁剪的设备名称时需要设置长度
//init.advdata.name_type                = BLE_ADVDATA_SHORT_NAME;
//init.advdata.short_name_len           = 6;
```

6.2.2. 实验步骤

1. 解压“…\4：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 4-3：广播中加入设备名称”，将解压后得到的文件夹“ble_app_templete”拷贝合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_templete\project\mdk5”目录下的工程“ble_app_templete_s140.uvproj”。
4. 切换到 Target“flash_s140_nrf52_7.0.1_softdevice”，点击下载协议栈。
5. 切换到 Target“IK-52840DK”，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
6. 程序运行后，指示灯 D1 闪烁表示正在广播。
7. 手机上打开 nRF Connect，点击“SCAN”按钮启动扫描，因为代码中设置的名称类型是全称，而且广播包的剩余空间也足够容纳设备名称，所以扫描时可以看到全部的设备名称，如下图所示。

图 4-20：广播包中包含全称

8. 程序中屏蔽“使用全称的代码”，取消屏蔽“不包含设备名称的代码”，编译后下载运行。这时候使用 nRF Connect 扫描，可以看到，扫描到的广播是没有设备名称的，如下图所示。

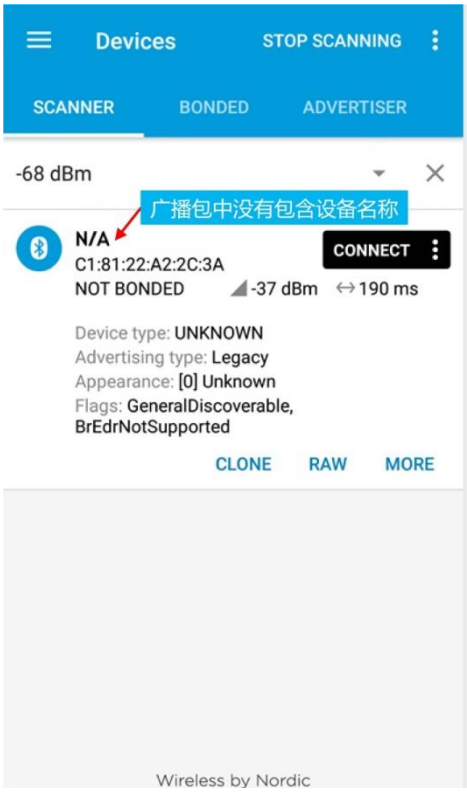


图 4-21：广播包中不包含设备名称

9. 程序中屏蔽“不包含设备名称的代码”，取消屏蔽“使用裁剪的设备名称代码”，编译后下载运行。这时候使用 nRF Connect 扫描，可以看到，扫描到的设备名称和我们在代码中设置的裁剪长度一致（6 个字节），如下图所示。

图 4-22：广播包中包含裁剪的设备名称

点击“CONNECT”按钮连接设备，连接成功后读取 GAP 服务的设备名称特征，读出后，可以看到裁剪的设备名称是从从名称的起始位置开始连续的 6 个字符（程序中设置的裁剪长度是 6），如下图所示。

图 4-23：读出 GAP 服务的设备名称特征

6.3. 广播中使用中文设备名称实验

本实验在“实验 4-3-1：广播中加入设备名称”的基础上修改。

本实验使用中文 “艾克姆”做为设备名称，设备名称类型为全称。

✧ 注：本节对应的实验源码是：“实验 4-3-2：广播中使用中文设备名称”。

6.3.1. 代码编写

使用中文设备名称时，首先要得到中文转 URL 的编码（可以使用在线中文转 URL 编码工具获取），之后定义一个保存编码的数组，在 GAP 初始化函数中调用 sd_ble_gap_device_name_set()函数设置设备名称时，将该数组为参数传递给函数写入到协议栈。最后在广播初始化函数 advertising_init()中设置设备名称类型为全称即可。

1. 得到“艾克姆”的 URL 编码

   “艾克姆”转为 URL 后为“%E8%89%BE%E5%85%8B%E5%A7%86”。

2. 定义设备名称数组

   设备名称数组定义如下，注意赋值时去掉 URL 编码中的“%”。

   const char device_name[9] = {0xE8,0x89,0xBE, 0xE5,0x85,0x8B,0xE5,0xA7,0x86};

3. GAP 初始化函数中设置广播名称，代码清单如下：

代码清单： GAP 初始化函数中配置设备名称

```
//设置 GAP 设备名称
err_code = sd_ble_gap_device_name_set(&sec_mode,
                                      (const uint8_t *)device_name,
                                     sizeof(device_name));
//检查函数返回的错误代码
APP_ERROR_CHECK(err_code);
```

4. 广播初始化函数 advertising_init()中设置设备名称类型为全称。

6.3.2. 实验步骤

1. 解压“…\4：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 4-3-2：广播中使用中文设备名称”，将解压后得到的文件夹“ble_app_templete”拷贝合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_templete\project\mdk5”目录下的工程“ble_app_templete_s140.uvproj”。
4. 切换到 Target“flash_s140_nrf52_7.0.1_softdevice”，点击下载协议栈。
5. 切换到 Target“IK-52840DK”，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件

“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。

6. 程序运行后，指示灯 D1 闪烁表示正在广播。
7. 手机上打开 nRF Connect，点击“SCAN”按钮启动扫描，这时会扫描到设备名称为“艾克姆”的设备，如下图所示。

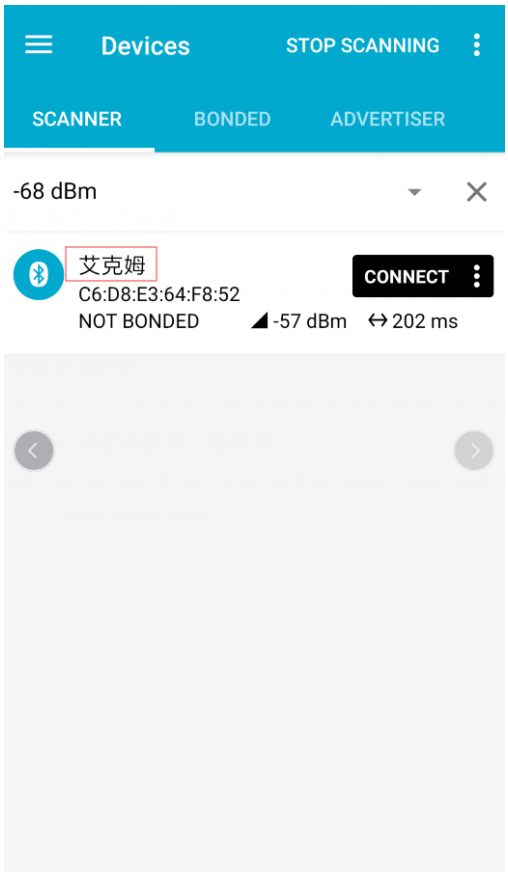


图 4-24：使用中文设备名称

7. 外观

7.1. 加入外观的步骤

外观是 GAP 服务的一个特征，外观的值是在 GAP 初始化函数里面设置的，广播初始化时只能选择广播包中是否包含外观，而不能设置外观的值。所以，配置外观的步骤是首先在 GAP 初始化函数中根据实际应用设置外观特征的值，然后在广播初始化函数中设置包含或不包含外观，如下图所示。

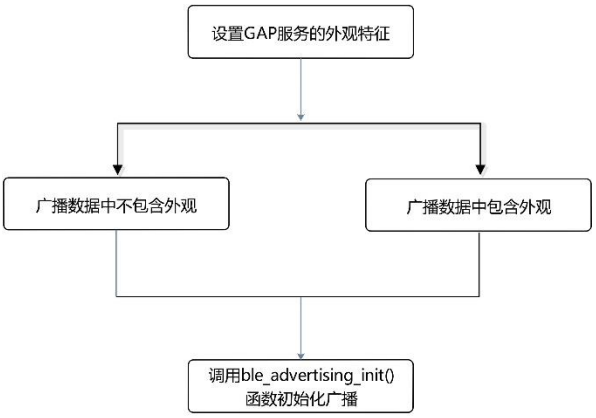


图 4-25：配置外观流程

1. GAP 初始化时设置外观特征

外观特征使用 API 函数 sd_ble_gap_appearance_set()设置，外观是 GAP 服务的一个特征，因此外观设置是在 GAP 初始化函数 gap_params_init()中执行的，下面是将外观设置为鼠标的示例代码。

代码清单： 设置外观特征

```
//设置外观特征。外观特征的值要根据实际应用去查询外观编码，这里设置的是：鼠标
err_code = sd_ble_gap_appearance_set(BLE_APPEARANCE_HID_MOUSE);
APP_ERROR_CHECK(err_code);
```

■ **如何确定外观的值？**

这是要根据具体应用去查询外观编码获取的（可以在 SIG 网站上查询或者在“ble_types.h”头文件中查询），如应用实现的是心率带，查询外观编码可以知道心率带的外观编码是 833，程序中对应的宏定义是：BLE_APPEARANCE_HEART_RATE_SENSOR_HEART_RATE_BELT。

2. 广播初始化时设置是否包含外观

广播初始化函数 advertising_init()中，通过设置广播初始化结构体中的 include_appearance 确定广播包中是否包含外观，include_appearance 的值为“true”包含外观，否则不包含。

```
//是否包含外观：包含
init.advdata.include_appearance      = true;
```

7.2. 广播中加入外观实验

本实验在“实验 3-3：BLE 工程模板（外围设备）”的基础上修改。广播初始化函数里面增加了包含外观的代码，。

✧ 注：本节对应的实验源码是：“实验 4-4：广播中加入外观”。

7.2.1. 代码编写

本例设置外观特征为鼠标，并设置广播包中包含外观，代码清单如下：

代码清单： 配置外观特征（代码中的 16~18 行）

```
1.  static void gap_params_init(void)
2.  {
3.      ret_code_t              err_code;
4.      //定义连接参数结构体变量
5.      ble_gap_conn_params_t   gap_conn_params;
6.      ble_gap_conn_sec_mode_t sec_mode;
7.      //设置 GAP 的安全模式
8.      BLE_GAP_CONN_SEC_MODE_SET_OPEN(&sec_mode);
9.      //设置 GAP 设备名称
10.     err_code = sd_ble_gap_device_name_set(&sec_mode,
11.                                           (const uint8_t *)DEVICE_NAME,
12.                                           strlen(DEVICE_NAME));
13.     //检查函数返回的错误代码
14.     APP_ERROR_CHECK(err_code);
15.
16.     //设置外观特征。外观特征的值要根据实际应用去查询外观编码，这里设置的是：鼠标
17.     err_code = sd_ble_gap_appearance_set(BLE_APPEARANCE_HID_MOUSE);
18.     APP_ERROR_CHECK(err_code);
19.
20.     //设置首选连接参数，设置前先清零 gap_conn_params
21.     memset(&gap_conn_params, 0, sizeof(gap_conn_params));
22.     gap_conn_params.min_conn_interval = MIN_CONN_INTERVAL;//最小连接间隔
23.     gap_conn_params.max_conn_interval = MAX_CONN_INTERVAL;//最小连接间隔
24.     gap_conn_params.slave_latency     = SLAVE_LATENCY;     //从机延迟
25.     gap_conn_params.conn_sup_timeout  = CONN_SUP_TIMEOUT; //监督超时
26.     //调用协议栈 API sd_ble_gap_ppcp_set 配置 GAP 参数
27.     err_code = sd_ble_gap_ppcp_set(&gap_conn_params);
28.     APP_ERROR_CHECK(err_code);
29. }
```

代码清单： 配置广播包中包含外观（代码中的第 11 行）

```
1.  static void advertising_init(void)
2.  {
3.      ret_code_t              err_code;
4.      //定义广播初始化配置结构体变量
5.      ble_advertising_init_t init;
6.      //配置之前先清零
7.      memset(&init, 0, sizeof(init));
```

```
    //设备名称类型：全称
    init.advdata.name_type               = BLE_ADVDATA_FULL_NAME;
    //是否包含外观：包含
    init.advdata.include_appearance      = true;
    //Flag:一般可发现模式，不支持 BR/EDR
    init.advdata.flags      = BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE;
    //设置广播模式为快速广播
    init.config.ble_adv_fast_enabled  = true;
    //设置广播间隔和广播持续时间
    init.config.ble_adv_fast_interval = APP_ADV_INTERVAL;
    init.config.ble_adv_fast_timeout  = APP_ADV_DURATION;
    //广播事件回调函数
    init.evt_handler = on_adv_evt;
    //初始化广播
    err_code = ble_advertising_init(&m_advertising, &init);
    APP_ERROR_CHECK(err_code);
    //设置广播配置标记。APP_BLE_CONN_CFG_TAG 是用于跟踪广播配置的标记，这是为未来预留的
    //一个参数，在将来的 SoftDevice 版本中，可以使用 sd_ble_gap_adv_set_configure()配
    //置新的广播配置，当前 SoftDevice 版本（S140 V7.0.1 版本）支持的最大广播集数量为 1，
    //因此 APP_BLE_CONN_CFG_TAG 只能写 1。
    ble_advertising_conn_cfg_tag_set(&m_advertising, APP_BLE_CONN_CFG_TAG);
}
```

7.2.2. 实验步骤

1. 解压“…\4：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 4-4：广播中加入外观”，将解压后得到的文件夹“ble_app_templete”拷贝合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_templete\project\mdk5”目录下的工程“ble_app_templete_s140.uvproj”。
4. 切换到 Target“flash_s140_nrf52_7.0.1_softdevice”，点击下载协议栈。
5. 切换到 Target“IK-52840DK”，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
6. 程序运行后，指示灯 D1 闪烁表示正在广播。
7. 手机上打开 nRF Connect，点击“SCAN”按钮启动扫描，可以看到广播包中包含了代码中设置的外观，如下图所示。

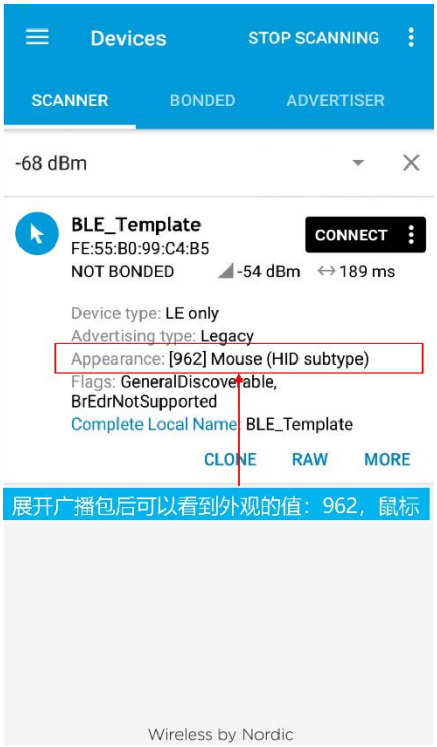


图 4-26：广播包中包含外观

8. Flags

Flags 是位字段的序列，当其中任何一个位不为零且广播可连接时广播包中应包含 flags，否则，Flags 可以被忽略。Flags 只能包含在广播包中，扫描响应包中不能包含 Flags。Flags 的作用是在广播包中加入如下标志：

- 有限可发现模式。
- 一般可发现模式。
- 不支持 BR/EDR。
- 设备同时支持 LE 和 BR/EDR（控制器）。
- 设备同时支持 LE 和 BR/EDR（主机）。

有了这些标志，当设备被扫描到后，对端设备即可根据这些标志执行相应动作，如 Flags 指示了设备不支持 BR/EDR，对端设备扫描到该设备后，即可知道该设置仅支持 LE，不支持传统蓝牙。Flags 的格式如下表所示。

表 4-6：Flags 格式

数据类型	位	描述
Flags	0	有限可发现模式。
	1	一般可发现模式。
	2	不支持 BR/EDR。
	3	设备同时支持 LE 和 BR/EDR（控制器）。

	4	设备同时支持 LE 和 BR/EDR（主机）。
	5~7	保留

1. 有限可发现模式和一般可发现模式的区别

有限可发现模式有时间的限制，一般维持的时间是 30 秒，而一般可发现模式没有时间的限制，有限可发现模式广播的间隔比一般可发现模式小。

从时间的限制上，我们可以看出有限可发现模式对连接的迫切性和目的性比一般可发现模式高，一个处于有限可发现模式的设备正在广播，那么他一定是刚被用户操作过并且极希望被连接。

2. 不支持 BD/EDR

不支持 BD/EDR 表示设备是单模设备，仅支持蓝牙低功耗，不支持传统蓝牙。

9. 发射功率等级

9.1. 发射功率等级的定义

发射功率等级（TX Power Level），指的是传输该广播包时使用的发射功率值。长度 1 个字节，单位是 dBm。发射功率等级定义如下表所示。

表 4-7：发射功率等级格式

数据类型	描述
发射功率等级（TX Power Level）	大小：1 字节。 范围： -127 到 +127 dBm。

发射功率等级可以用来计算路径损耗，计算公式如下：

pathloss = Tx Power Level － RSSI。

发射功率等级的一个应用场景如下图所示，设备 A 的发射功率等级为 0dBm，接收时读取的 RSSI 是-40 dBm，链路损耗是-40 dBm。设备 B 的发射功率等级为+4dBm，接收时读取的 RSSI 是-60 dBm，，链路损耗是-64 dBm。应用程序中，通过链路损耗即可判断出哪个设备离自己近一些。

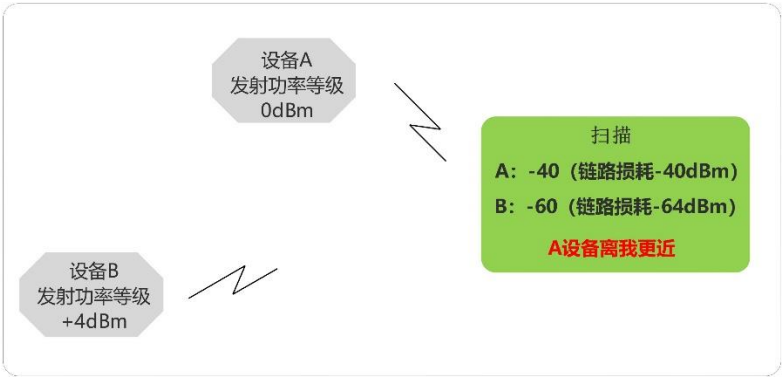


图 4-27：发射功率等级应用示意图

✧ **注意：**这里的发射功率等级指的是广播报文中包含发射功率等级，并不是设置芯片射频电路的发射功率。当然，我们可以将发射功率等级的数值设置为芯片实际配置的发射功率，这样，对端设备接收到广播报文后，通过报文中的发射功率等级即可知道该设备配置的发射功率是多少。

9.2. 广播中加入发射功率等级实验

本实验在“实验 3-3：BLE 工程模板（外围设备）”的基础上修改。广播初始化函数advertising_init()里面增加了配置发射功率等级的代码，。

✧ 注：本节对应的实验源码是：“实验 4-5：广播中加入发射功率等级”。

9.2.1. 代码编写

加入发射功率等级时，因为广播初始化结构体中发射功率等级的变量是指针，所以要定义一个int8类型的变量，并设置发射等级，然后将该变量的地址赋值给指针`p_tx_power_level`，代码清单如下：

代码清单： 配置外观特征

```
//定义保存发射功率等级的变量
int8_t tx_power_level = 0;
//发送功率等级
init.advdata.p_tx_power_level = &tx_power_level;
```

9.2.2. 实验步骤

1. 解压“…\4：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 4-5：广播中加入发射功率等级”，将解压后得到的文件夹“ble_app_templete”拷贝合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_templete\project\mdk5”目录下的工程“ble_app_templete_s140.uvproj”。
4. 切换到 Target“flash_s140_nrf52_7.0.1_softdevice”，点击下载协议栈。
5. 切换到 Target“IK-52840DK”，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
6. 程序运行后，指示灯 D1 闪烁表示正在广播。
7. 手机上打开 nRF Connect，点击“SCAN”按钮启动扫描，可以看到广播包中包含了发射功率等级，如下图所示。

图 4-28：广播包中包含发射功率等级

10. 服务的 UUID

10.1. UUID

广播数据中，一般会包含一个服务 UUID 列表，用以展示自己支持的服务，但是 GAP 和 GATT 服务的 UUID 不能加到广播包中。广播包中可以根据自身的情况包含一部分服务的 UUID 或者包含所有服务的 UUID。部分 UUID 列表和完整的 UUID 列表同时只能用一种，不能在广播数据中同时包含这两种。广播包中包含完整的服务 UUID 很容易理解，那么一般什么情况下包含部分 UUID？下列两种应用场景下一般会使用部分 UUID 列表。

- 广播数据无法容纳全部服务的 UUID。
- 设备不想在广播阶段对外展示其所支持的全部的服务。

蓝牙核心规范中定义的服务 UUID 的格式如下表所示。

表 4-8：服务 UUID 的格式

数据类型	描述
部分 16 位服务 UUID	还有更多的 16 位服务 UUID。
完整的 16 位服务 UUID	全部的 16 位服务 UUID。
部分 32 位服务 UUID	还有更多的 32 位服务 UUID。

完整的 32 位服务 UUID	全部的 32 位服务 UUID。
部分 128 位服务 UUID	还有更多的 128 位服务 UUID。
完整的 128 位服务 UUID	全部的 128 位服务 UUID。

■ **UUID 是什么？**

UUID(通用唯一识别码：Universally Unique Identifier)是一个 128 位的数字，用来标志属性的类型。服务也是一种属性，当然也需要用 UUID 来标志。

1. 既然 UUID 是 128 位的数字，那么为什么有 16 位的 UUID？

因为 128 位的 UUID 相当长，设备间为了识别数据的类型需要发送长达 16 个字节的数据。为了提高传输效率，蓝牙技术联盟(SIG)定义了一个“UUID 基数”，结合一个较短的 16 位数使用。二者仍然遵循通用唯一识别码的分配规则，只不过在设备间传输常用的 UUID 时，只发送较短的 16 位版本，接收方收到后补上蓝牙 UUID 基数即可。

2. UUID 分为标准的 UUID 和厂商自定义的 UUID。

- 标准的 UUID：由 SIG 发布，采用 UUID 基数+16 位 UUID 的形式，如心率服务的 UUID 是 0x180D，使用的 UUID 基数是：00000000 - 0000 - 1000 - 8000 – 00805F9B34FB。
- 厂商自定义的 UUID：同样采用 UUID 基数+16 位 UUID 的形式，由厂商定义，如 BLE 串口服务的 UUID 是 0x0001，使用的 UUID 基数是：6E400001 – B5A3 – F393 – E0A9 – E50E24DCCA9E。

下图中是标准 UUID 传输的示意图，180D 是心率服务的 UUID，因为是标准的 UUID，所以收发双方都知道 UUID 基数是多少，因此传输的时候只会传输 16 位的 UUID，接收方会自己补上 UUID 基数。

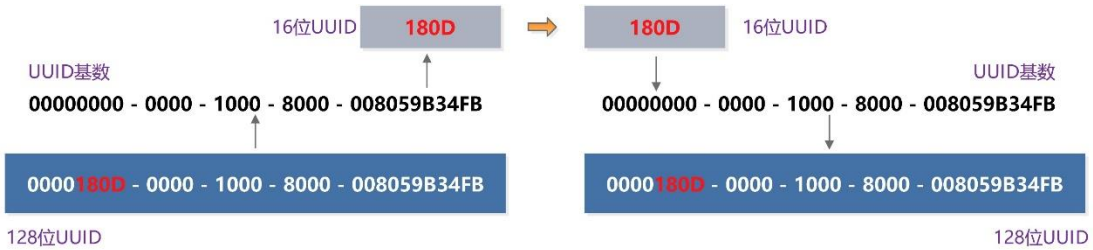


图 4-29：标准 UUID 传输示意图

再次强调一下：完整的 UUID 是 128 位的，我们常说的 16 位 UUID 本质上还是 128 位的，只不过传输的时候为了提高效率，只用了 16 位 UUID 的形式。

为了方便管理，增加 UUID 的可读性，蓝牙低功耗使用的那部分 UUID 被分为下列几组：

- 0x1800 ~ 0x26FF：用作服务类通用唯一识别码。
- 0x2700 ~ 0x27FF：用于标识计量单位。
- 0x2800 ~ 0x28FF：用于区分属性类型。
- 0x2900 ~ 0x29FF：用作特征描述。
- 0x2A00 ~ 0x7FFF：用于区分特征类型。

10.2. 加入服务 UUID 的步骤

广播中加入服务 UUID 的步骤如下图所示，首先定义 UUID 数组并初始化，然后配置广初始化结构体中如何包含 UUID 数组（是包含全部的服务 UUID 还是部分的服务 UUID），之后调用广播初始化函数完成广播初始化，广播初始化成功后，服务的 UUID 列表即会加入到广播数据包中。

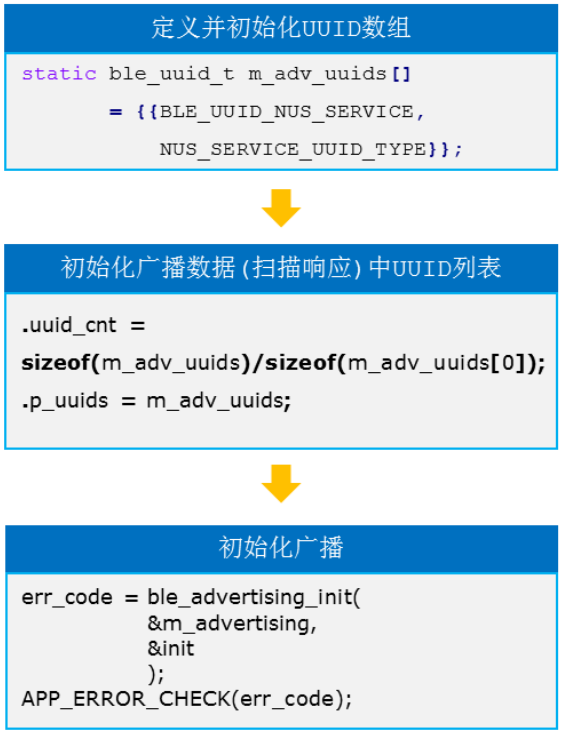


图 4-30：广播中加入服务 UUID 的步骤

UUID 列表可以放在广播数据里面，也可以放在扫描响应数据里面，注意：广播包长度限制，即广播包的剩余空间是否能容纳的了 UUID 列表。

1. 定义并初始化 UUID 数组

UUID 结构体声明如下，在代码中，对于 16 位 UUID 和 128 位 UUID 均使用该结构体定义（对于厂商自定义的 128 位 UUID，使用该结构体定义 16 个字节中的第 12 和 13 字节，初次之外，程序中还要定义 UUID 基数）。

代码清单： UUID 结构体

```
typedef struct
{
  uint16_t    uuid; //16 位 UUID，即 128 位 UUID 中的第 12 和 13 字节
  uint8_t     type; //UUID 类型
} ble_uuid_t;
```

蓝牙核心规范中定义了 2 种类型的 UUID：标准 UUID 和厂商自定义 UUID，程序中为

了操作方便，增加了一种类型：无效的 UUID，用来标志某个 UUID 是否合法。

```
#define BLE_UUID_TYPE_UNKNOWN      0x00 //无效的 UUID
#define BLE_UUID_TYPE_BLE          0x01 //标准 UUID
#define BLE_UUID_TYPE_VENDOR_BEGIN 0x02 //厂商自定义 UUID
```

有了 UUID 结构体声明和 UUID 类型，就可以很方便地定义 UUID 列表（UUID 结构体数组），如下所示。

心率例程中服务 UUID 列表

```
static ble_uuid_t m_adv_uuids[] =
      {{BLE_UUID_HEART_RATE_SERVICE, BLE_UUID_TYPE_BLE},
      {BLE_UUID_BATTERY_SERVICE, BLE_UUID_TYPE_BLE},
      {BLE_UUID_DEVICE_INFORMATION_SERVICE, BLE_UUID_TYPE_BLE}};
```

BLE 串口透传例程中服务 UUID 列表

```
static ble_uuid_t m_adv_uuids[] =
     {{BLE_UUID_NUS_SERVICE, NUS_SERVICE_UUID_TYPE}};
```

2. 初始化广播数据中的 UUID 列表

定义并初始化了服务 UUID 数组之后，服务 UUID 数组并没有被包含到广播数据中，我们还需要配置广播数据中如何包含 UUID 数组。

广播数据中通过 ble_advdata_uuid_list_t 结构体来引用服务 UUID 数组，该结构体声明如下，其中 p_uuids 指向服务 UUID 数组入口，uuid_cnt 是引用服务 UUID 的数量，这两个变量赋值后，广播初始化函数就会知道从何处取得服务 UUID，取多少个服务 UUID。

代码清单： ble_advdata_uuid_list_t 结构体

```
1. typedef struct
2. {
3.     uint16_t        uuid_cnt;          //UUID 数量
4.     ble_uuid_t *    p_uuids;           //指向服务 UUID 数组入口
5. } ble_advdata_uuid_list_t;
```

广播数据结构体 ble_advdata_t 中定义了两个 ble_advdata_uuid_list_t 结构体变量 uuids_more_available 和 uuids_complete，用来实现部分 UUID 列表和完整 UUID 列表。

- uuids_more_available：广播中包含部分 UUID 列表时使用。
- uuids_complete：广播中包含完整的 UUID 列表时使用。

广播中使用完整的 UUID 列表和使用部分的 UUID 列表时的代码如下。

代码清单： 广播中使用完整的 UUID 列表示例

```
//广播数据中包含全部的服务 UUID
 init.advdata.uuids_complete.uuid_cnt = sizeof(m_adv_uuids) / sizeof(m_adv_uuids[0]);
 init.advdata.uuids_complete.p_uuids  = m_adv_uuids;
```

代码清单： 广播中使用部分的 UUID 列表示例

```
 //广播数据中包含部分的服务 UUID
init.advdata.uuids_more_available.uuid_cnt = (sizeof(m_adv_uuids) / sizeof(m_adv_uuids[0]))-1;
init.advdata.uuids_more_available.p_uuids  = m_adv_uuids;
```

10.3. 广播中加入 UUID 列表实验

本实验在“实验 3-3：BLE 工程模板（外围设备）”的基础上修改。广播初始化函数里面增加了使用部分的 UUID 列表和完整的 UUID 列表的代码。

程序里面定义的 UUID 数组包含心率服务和电池服务，这是为了演示如何在广播中加入 UUID 列表，实际程序中并没有对应的服务。

✧ 注：本节对应的实验源码是：“实验 4-6：广播中加入 UUID 列表”。

10.3.1. 代码编写

定义的 UUID 数组，该数组包含心率服务和电池服务，代码清单如下：

代码清单： 定义服务 UUID 数组

```
//定义服务 UUID 数组，这里定义的 UUID 数组包含心率服务和电池服务，是为了演示如何在广播中加入 UUID 列表，程序中并没有对应的服务
static ble_uuid_t m_adv_uuids[] =
{
    {BLE_UUID_HEART_RATE_SERVICE,           BLE_UUID_TYPE_BLE},
    {BLE_UUID_BATTERY_SERVICE,              BLE_UUID_TYPE_BLE},
};
```

广播初始化中根据实际需求（设备愿意对外展示全部服务 UUID 还是仅展示服务 UUID），配置广播中包含部分 UUID 或者全部 UUID。

代码清单： 广播初始化中配置 UUID 列表（15 行到 22 行）

```
1.  static void advertising_init(void)
2.  {
3.      ret_code_t             err_code;
```

```
    //定义广播初始化配置结构体变量
    ble_advertising_init_t init;
    //配置之前先清零
    memset(&init, 0, sizeof(init));
    //设备名称类型：全称
    init.advdata.name_type               = BLE_ADVDATA_FULL_NAME;
    //是否包含外观：包含
    init.advdata.include_appearance      = true;
    //Flag:一般可发现模式，不支持 BR/EDR
    init.advdata.flags                   = BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DIS
                                           C_MODE;
    //广播数据中包含全部的服务 UUID
    init.advdata.uuids_complete.uuid_cnt = sizeof(m_adv_uuids) / sizeof(m_adv_uu
                                           ids[0]);
    init.advdata.uuids_complete.p_uuids  = m_adv_uuids;
    //广播数据中包含部分的服务 UUID
    //init.advdata.uuids_more_available.uuid_cnt = (sizeof(m_adv_uuids) / sizeof
                                                   (m_adv_uuids[0]))-1;
    //init.advdata.uuids_more_available.p_uuids  = m_adv_uuids;

    //设置广播模式为快速广播
    init.config.ble_adv_fast_enabled  = true;
    //设置广播间隔和广播持续时间
    init.config.ble_adv_fast_interval = APP_ADV_INTERVAL;
    init.config.ble_adv_fast_timeout  = APP_ADV_DURATION;
    //广播事件回调函数
    init.evt_handler = on_adv_evt;
    //初始化广播
    err_code = ble_advertising_init(&m_advertising, &init);
    APP_ERROR_CHECK(err_code);
    //设置广播配置标记。APP_BLE_CONN_CFG_TAG 是用于跟踪广播配置的标记，这是为未来预留的
    //一个参数，在将来的 SoftDevice 版本中，可以使用 sd_ble_gap_adv_set_configure()配
    //置新的广播配置，当前 SoftDevice 版本（S140 V7.0.1 版本）支持的最大广播集数量为 1，
    //因此 APP_BLE_CONN_CFG_TAG 只能写 1。
    ble_advertising_conn_cfg_tag_set(&m_advertising, APP_BLE_CONN_CFG_TAG);
}
```

10.3.2. 实验步骤

1. 解压“…\4：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 4-6：广播中加入 UUID 列表”，将解压后得到的文件夹“ble_app_templete”拷贝合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。

3. 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_templete\project\mdk5”目录下的工程“ble_app_templete_s140.uvproj”。
4. 切换到 Target“flash_s140_nrf52_7.0.1_softdevice”，点击下载协议栈。
5. 切换到 Target“IK-52840DK”，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
6. 程序运行后，指示灯 D1 闪烁表示正在广播。
7. 手机上打开 nRF Connect，点击“SCAN”按钮启动扫描，因为代码中设置的广播中包含完整的 UUID 列表，而且广播包的剩余空间也足够容纳，所以扫描时可以看到全部服务的 UUID，如下图所示。

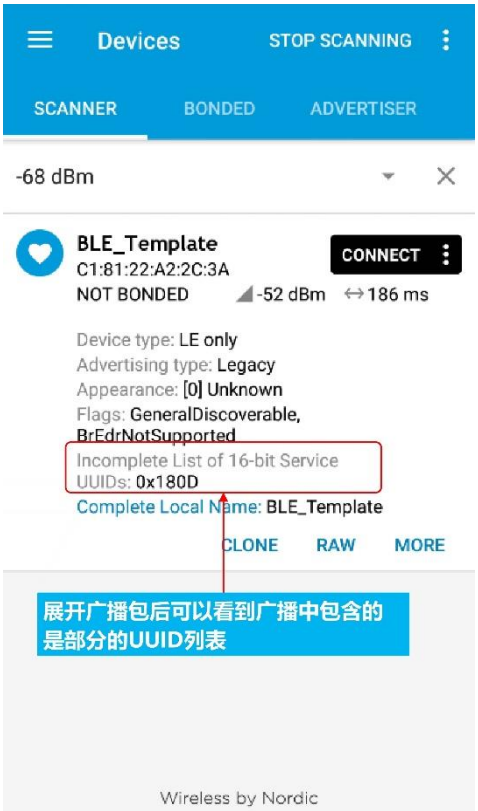


图 4-31：广播包中包含全称

8. 程序中屏蔽“加入完整 UUID 列表”的代码，取消屏蔽“加入部分 UUID 列表” 的代码，编译后下载运行。这时候使用 nRF Connect 扫描，可以看到，广播数据中包含的是部分 UUID，如下图所示。

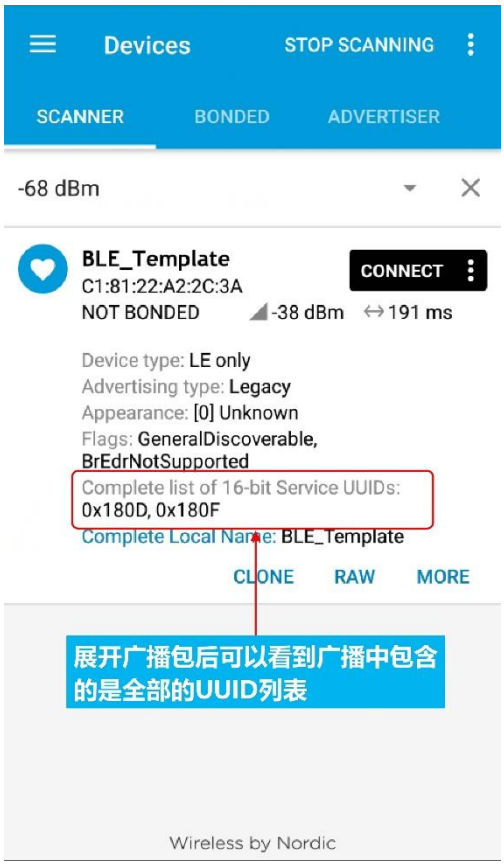


图 4-32：广播包中包含部分 UUID 列表

11. 服务数据

11.1. 服务数据的定义

服务数据用来在广播中发送某个服务的数据，服务数据可以不需要连接就能解析出数据。如发送的是电池电量服务数据，手机端 APP 扫描到广播数据后，可以显示出电池电量。

服务数据由 UUID 和数据两部分组成。广播数据中可以加入多个服务数据，如可以同时加入电池电量服务数据和温度服务数据等。

11.2. 加入服务 UUID 的步骤

广播中加入服务数据的步骤如下图所示，首先定义服务数据结构体变量并初始化，然后配置广播初始化结构体中包含服务数据，即设置数据服务的 UUID 以及指针指向服务数据，之后调用广播初始化函数完成广播初始化，广播初始化成功后，服务数据即会加入到广播数据包中。

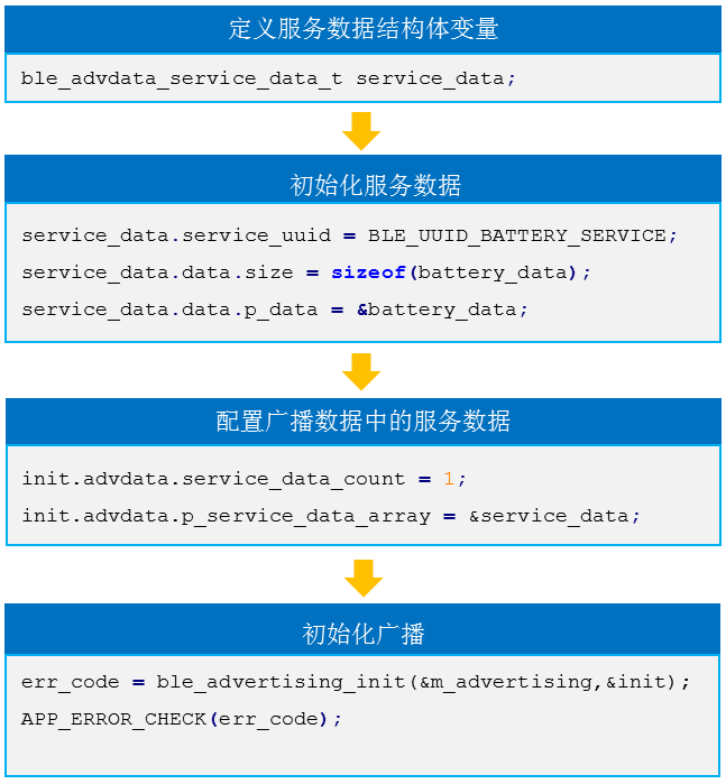


图 4-33：广播中加入服务数据的步骤

1. 定义服务数据结构体变量

服务数据结构体声明如下，该结构体有 2 个成员变量，用于描述服务的 UUID 和附加的服务数据。

代码清单： 服务数据结构体声明

```
typedef struct
{
    uint16_t                    service_uuid;    //服务的 UUID
    uint8_array_t               data;            //附加的服务数据
} ble_advdata_service_data_t;
```

代码清单：定义服务数据结构体变量

```
//定义服务数据结构体
ble_advdata_service_data_t service_data;
```

2. 初始化服务数据

初始化服务数据也就是给服务数据结构体成员变量赋值，下面的代码是初始化服务数据结构体变量为电池电量服务数据，电池电量服务是标准服务，其 UUID 是 0x180F。

代码中 p_data 是一个指针，不能直接将电池电量赋值给该指针，需要定义一个保存电池电量的变量 battery_data，将该变量的地址赋值给指针 p_data。

代码清单： 初始化服务数据

```
//电池电量服务 UUID 0x180F
service_data.service_uuid = BLE_UUID_BATTERY_SERVICE;
service_data.data.size = sizeof(battery_data);
service_data.data.p_data = &battery_data;
```

3. 配置广播数据中服务数据

初始化服务数据之后，服务数据并没有被包含到广播数据中，我们还需要通过配置将服务数据加入到广播数据中。

广播中加入一个服务数据的代码如下所示，即设置 p_service_data_array 指向已经初始化的服务数据，service_data_count 为加入的服务数据的数量。

代码清单： 广播中加入 1 个服务数据

```
//广播数据中加入服务数据
init.advdata.p_service_data_array = &service_data;
init.advdata.service_data_count = 1;
```

11.3. 加入服务数据实验

本实验在“实验 3-3：BLE 工程模板（外围设备）”的基础上修改。广播初始化函数里面增加了加入电池电量服务数据。

本例的重点是为了说明如何在广播中加入服务数据，程序里面使用的电池电量是模拟电池电量的数据，并不是实际采样得到的数据。

✧ 注：本节对应的实验源码是：“实验 4-7：广播中加入电池电量服务数据”。

11.3.1. 代码编写

因为我们要在广播中加入电池电量服务数据，所以只需定义一个服务结构体变量 service_data，并将其初始为电量服务数据，之后配置广播中包含该广播数据即可。

代码清单： 广播中加入服务数据（4 行到 7 行和 19 行到 26 行）

```
1.  static void advertising_init(void)
2.  {
3.      ret_code_t             err_code;
4.      //电池电量
5.      uint8_t battery_data = 85;
6.      //定义服务数据结构体
7.      ble_advdata_service_data_t service_data;
8.
9.      //定义广播初始化配置结构体变量
```

```
10.    ble_advertising_init_t init;
11.    //配置之前先清零
12.    memset(&init, 0, sizeof(init));
13.    //设备名称类型：全称
14.    init.advdata.name_type               = BLE_ADVDATA_FULL_NAME;
15.    //是否包含外观：包含
16.    init.advdata.include_appearance      = true;
17.    //Flag:一般可发现模式，不支持 BR/EDR
18.    init.advdata.flags      = BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE;
19.    //电池电量服务 UUID 0x180F
20.    service_data.service_uuid = BLE_UUID_BATTERY_SERVICE;
21.    service_data.data.size = sizeof(battery_data);
22.    service_data.data.p_data = &battery_data;
23.    //广播数据中加入服务数据
24.    init.advdata.p_service_data_array = &service_data;
25.    //因为只加入一个服务数据，所以服务数据数量设置为 1
26.    init.advdata.service_data_count = 1;
27.    //设置广播模式为快速广播
28.    init.config.ble_adv_fast_enabled  = true;
29.    //设置广播间隔和广播持续时间
30.    init.config.ble_adv_fast_interval = APP_ADV_INTERVAL;
31.    init.config.ble_adv_fast_timeout  = APP_ADV_DURATION;
32.    //广播事件回调函数
33.    init.evt_handler = on_adv_evt;
34.    //初始化广播
35.    err_code = ble_advertising_init(&m_advertising, &init);
36.    APP_ERROR_CHECK(err_code);
37.    //设置广播配置标记。APP_BLE_CONN_CFG_TAG 是用于跟踪广播配置的标记，这是为未来预留的
38.    //一个参数，在将来的 SoftDevice 版本中，可以使用 sd_ble_gap_adv_set_configure()配
39.    //置新的广播配置，当前 SoftDevice 版本（S140 V7.0.1 版本）支持的最大广播集数量为 1，
40.    //因此 APP_BLE_CONN_CFG_TAG 只能写 1。
41.    ble_advertising_conn_cfg_tag_set(&m_advertising, APP_BLE_CONN_CFG_TAG);
42. }
```

11.3.2. 实验步骤

1. 解压“…\4：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 4-7：广播中加入电池电量服务数据”，将解压后得到的文件夹“ble_app_templete”拷贝合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_templete\project\mdk5”目录下的工程“ble_app_templete_s140.uvproj”。
4. 切换到 Target“flash_s140_nrf52_7.0.1_softdevice”，点击下载协议栈。

5. 切换到 Target“IK-52840DK”，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
6. 程序运行后，指示灯 D1 闪烁表示正在广播。
7. 手机上打开 nRF Connect，点击“SCAN”按钮启动扫描，因为代码中设置的广播中包含电池电量服务数据，而且广播包的剩余空间也足够容纳，所以扫描时可以看到电池电量服务数据，如下图所示。

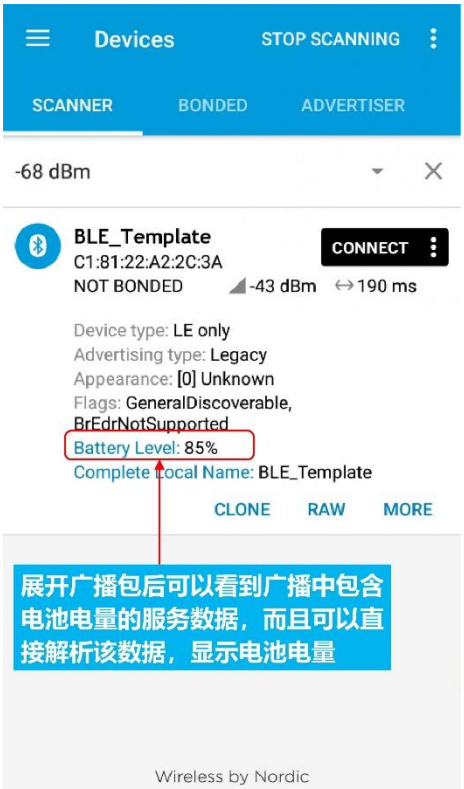


图 4-34：广播包中包含电池电量服务数据

12. 从设备连接间隔范围

12.1. 从设备连接间隔范围定义

从设备连接间隔范围代表外围设备倾向的连接间隔，注意：连接参数是完全由主设备控制的，从设备只能表达自己的倾向和申请协商连接参数更新。广播中加入从设备连接间隔范围，目的是向主机告知自己倾向的连接间隔，给主机一个参考值。从设备连接间隔范围格式如下图所示，包含 4 个字节数据：1 个 16 位的最小连接间隔和 1 个 16 位的最大连接间隔。

最小连接间隔（2字节）	最大连接间隔（2字节）

- 可设置值的范围：0x0006~0x0C80。
- 单位：1.25ms。
- 如果值设置为0xFFFF，表示从机没有明确间隔范围。

图 4-35：从设备连接间隔范围格式

12.2. 加入从设备连接间隔范围的步骤

广播中加入从设备连接间隔范围的步骤如下图所示，首先定义从设备连接间隔范围结构体变量并初始化，之后调用广播初始化函数完成广播初始化，广播初始化成功后，从设备连接间隔范围即会加入到广播数据包中。

图 4-36：加入从设备连接间隔范围的步骤

1. 定义从设备连接间隔范围结构体变量

从设备连接间隔范围结构体 ble_advdata_conn_int_t 声明如下，该结构体有 2 个 16 位的成员变量，用于描述最小连接间隔和最大连接间隔。

代码清单： 从设备连接间隔范围结构体声明

```
typedef struct
{
```

```
    //最小连接间隔，单位 1.25ms，可设置的范围：6~3200（即 7.5ms~4s）
    uint16_t            min_conn_interval;
    //最大连接间隔，单位 1.25ms，可设置的范围：6~3200（即 7.5ms~4s）
    uint16_t            max_conn_interval;
} ble_advdata_conn_int_t;
```

代码清单： 定义从设备连接间隔范围结构体变量

```
//定义从设备连接间隔范围结构体变量
ble_advdata_conn_int_t conn_range;
```

2. 初始化从设备连接间隔范围

初始化从设备连接间隔范围也就是给从设备连接间隔范围结构体成员变量赋值，即设置从设备连接间隔范围的最大值和最小值。

代码清单： 初始化服务数据

```
//从设备连接间隔范围最小值：30*1.25ms = 37.5ms
conn_range.min_conn_interval = 30;
//从设备连接间隔范围最大值：100*1.25ms = 125ms
conn_range.max_conn_interval = 100;
```

3. 配置广播数据中加入从设备连接间隔范围

初始化从设备连接间隔范围之后，从设备连接间隔范围并没有被包含到广播数据中，我们还需要通过配置将从设备连接间隔范围加入到广播数据中。

对于从设备连接间隔范围，设置 p_slave_conn_int 指向已经初始化的从设备连接间隔范围结构体变量 conn_range 即可。

代码清单： 广播中加入 1 个服务数据

```
//广播数据中包含从设备连接间隔范围
init.advdata.p_slave_conn_int = &conn_range;
```

12.3. 广播中加入从设备连接间隔范围实验

本实验在“实验 3-3：BLE 工程模板（外围设备）”的基础上修改。

本例在广播中加从设备连接间隔范围，最小间隔设置为 30，即 30*1.25ms = 37.5ms，最小间隔设置为 100，即 100*1.25ms = 125ms。

✧ 注：本节对应的实验源码是："实验 4-8：广播中加入从设备连接间隔范围"。

12.3.1. 代码编写

按照前文描述的流程，首先定义从设备连接间隔范围结构体变量（代码的第 4 和第 5 行），然后初始化连接间隔的最大值和最小值（代码的第 17~20 行），本例中设置的最小值是 30，最大值是 100，之后设置广播中包含从设备连接间隔范围（代码的第 21~22 行），这样在广播初始化成功后，从设备连接间隔范围即会被加入到广播数据中。

代码清单： 广播中加入从设备连接间隔范围（4 行到 5 行和 17 行到 22 行）

```
1.  static void advertising_init(void)
2.  {
3.    ret_code_t             err_code;
4.    //定义从设备连接间隔范围结构体变量
5.    ble_advdata_conn_int_t conn_range;
6.
7.    //定义广播初始化配置结构体变量
8.    ble_advertising_init_t init;
9.    //配置之前先清零
10.   memset(&init, 0, sizeof(init));
11.   //设备名称类型：全称
12.   init.advdata.name_type               = BLE_ADVDATA_FULL_NAME;
13.   //是否包含外观：包含
14.   init.advdata.include_appearance      = true;
15.   //Flag:一般可发现模式，不支持 BR/EDR
16.   init.advdata.flags     = BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE;
17.   //从设备连接间隔范围最小值：30*1.25ms = 37.5ms
18.   conn_range.min_conn_interval = 30;
19.   //从设备连接间隔范围最大值：100*1.25ms = 125ms
20.   conn_range.max_conn_interval = 100;
21.   //广播数据中包含从设备连接间隔范围
22.   init.advdata.p_slave_conn_int = &conn_range;
23.   //设置广播模式为快速广播
24.   init.config.ble_adv_fast_enabled  = true;
25.   //设置广播间隔和广播持续时间
26.   init.config.ble_adv_fast_interval = APP_ADV_INTERVAL;
27.   init.config.ble_adv_fast_timeout  = APP_ADV_DURATION;
28.   //广播事件回调函数
29.   init.evt_handler = on_adv_evt;
30.   //初始化广播
31.   err_code = ble_advertising_init(&m_advertising, &init);
32.   APP_ERROR_CHECK(err_code);
33.   //设置广播配置标记。APP_BLE_CONN_CFG_TAG 是用于跟踪广播配置的标记，这是为未来预留的
34.  //一个参数，在将来的 SoftDevice 版本中，可以使用 sd_ble_gap_adv_set_configure()配
```

```
35.  //置新的广播配置，当前 SoftDevice 版本（S140 V7.0.1 版本）支持的最大广播集数量为 1，
36.  //因此 APP_BLE_CONN_CFG_TAG 只能写 1。
37.  ble_advertising_conn_cfg_tag_set(&m_advertising, APP_BLE_CONN_CFG_TAG);
38. }
```

12.3.2. 实验步骤

1. 解压“…\4：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 4-8：广播中加入从设备连接间隔范围”，将解压后得到的文件夹“ble_app_templete”拷贝合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_templete\project\mdk5”目录下的工程“ble_app_templete_s140.uvproj”。
4. 切换到 Target“flash_s140_nrf52_7.0.1_softdevice”，点击下载协议栈。
5. 切换到 Target“IK-52840DK”，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
6. 程序运行后，指示灯 D1 闪烁表示正在广播。
7. 手机上打开 nRF Connect，点击“SCAN”按钮启动扫描，因为代码中设置的广播中包含从设备连接间隔范围，而且广播包的剩余空间也足够容纳，所以扫描时可以看到从设备连接间隔范围，如下图所示。

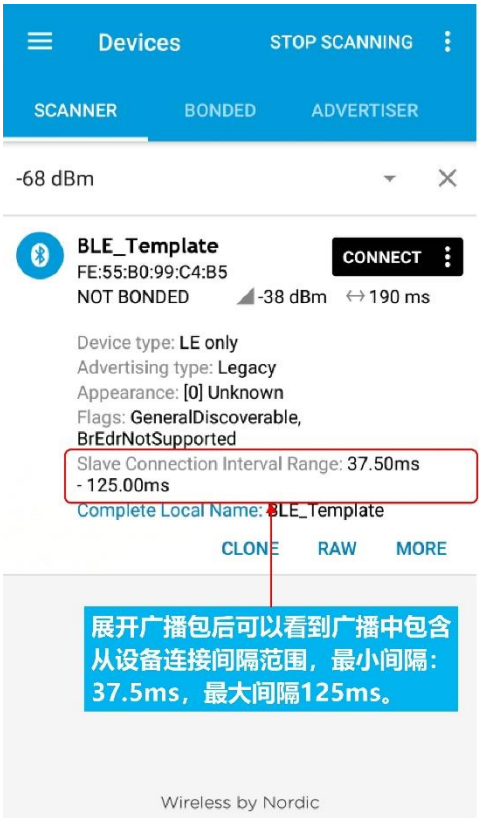


图 4-37：广播数据中包含从设备连接间隔范围

13. 厂商自定义数据

13.1. 厂商自定义数据的定义

厂商自定义数据（Manufacturer Specific Data），是用来在广播数据中加入厂商自己定义的数据。这无疑增加了开发的灵活性，厂商可以根据自己的需求组织数据格式，并将数据加入到广播数据中，如 iBeacon 使用的就是厂商自定义数据。

厂商自定义数据的格式如下表所示，广播数据中加入厂商自定义数据时要特别注意数据长度不能超过广播包长度的限制，也就是要确保广播包剩余空间足够容纳需要加入的数据。

表 4-9：厂商自定义数据格式

数据类型	描述	
厂商自定义数据	大小：≥2 字节。	
	厂商 ID	自定义数据
	2 字节。	N 字节，不能超过广播包大小的限制。

13.2. 加入厂商自定义数据的步骤

广播中加入厂商自定义数据的步骤如下图所示，首先定义厂商自定义数据结构体变量并初始化，然后配置广播初始化结构体中包含厂商自定义数据，之后调用广播初始化函数完成广播初始化，广播初始化成功后，厂商自定义数据即会加入到广播数据包中。

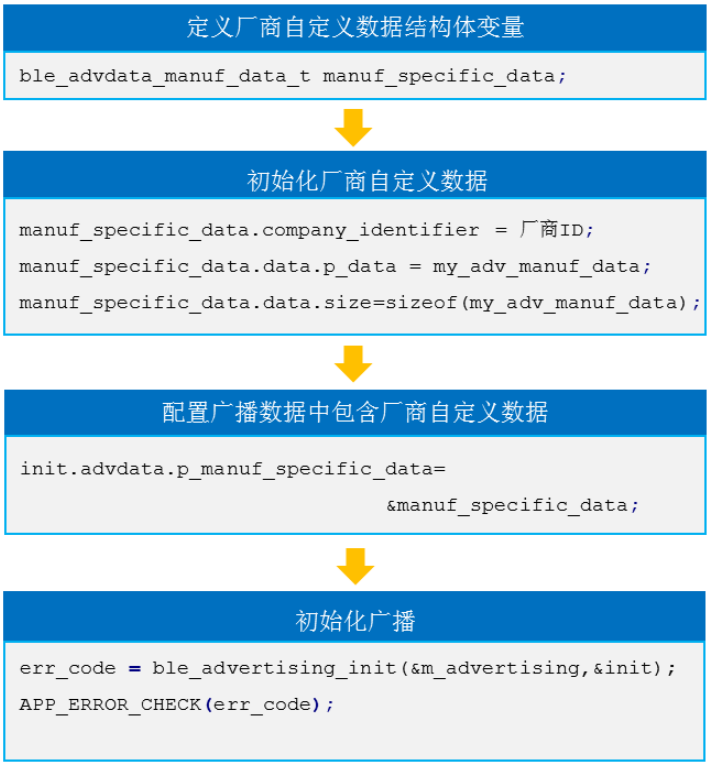


图 4-38：广播中加入厂商自定义数据的步骤

1. 定义厂商自定义数据结构体变量

厂商自定义数据结构体 ble_advdata_manuf_data_t 声明如下，该结构体有 2 个成员变量，用于描述厂商 ID 和附加的自定义数据。

代码清单： 厂商自定义数据结构体声明

```
typedef struct
{
    uint16_t            company_identifier;     //厂商 ID
    uint8_array_t       data;                   //附件的自定义数据
} ble_advdata_manuf_data_t;
```

代码清单： 定义厂商自定义数据结构体变量

```
//定义个一个制造商自定义数据的结构体变量，配置广播数据时将该变量的地址赋值给
//广播数据结构体中的 p_manuf_specific_data
ble_advdata_manuf_data_t manuf_specific_data;
```

2. 初始化厂商自定义数据

初始化厂商自定义数据时需要设置厂商 ID（下面代码里面设置的 0x0059 是 Nordic 的 ID），指针 p_data 指向自定义的数据，并将数据大小赋值给 size。

代码清单： 初始化厂商自定义数据

```
//制造商 ID，0x0059 是 Nordic 的 ID
manuf_specific_data.company_identifier = 0x0059;
//指向制造商自定义的数据
manuf_specific_data.data.p_data = my_adv_manuf_data;
//制造商自定义的数据大小(字节数)
manuf_specific_data.data.size   = sizeof(my_adv_manuf_data);
```

3. 配置广播数据中加入厂商自定义数据

初始化厂商自定义数据之后，厂商自定义数据并没有被包含到广播数据中，我们还需要通过配置将厂商自定义数据加入到广播数据中。

对于厂商自定义数据，设置 p_manuf_specific_data 指向已经初始化的厂商自定义数据结构体变量 manuf_specific_data 即可。

代码清单： 广播中加入 1 个服务数据

```
//广播中加入制造商自定义数据
init.advdata.p_manuf_specific_data    = &manuf_specific_data;
```

13.3. 广播中加入厂商自定义数据实验

本实验在“实验 3-3：BLE 工程模板（外围设备）”的基础上修改。

本例在广播中加入的厂商自定义数据为：

- 厂商 ID：0x0059。
- 自定义数据：0x01,0x02,0x03,0x04,0x05 共 5 个字节自定义数据。

✧ 注：本节对应的实验源码是：“实验 4-9：广播中加入厂商自定义数据”。

13.3.1. 代码编写

按照前文描述的流程，首先定义厂商自定义数据结构体变量（代码的第 9~105 行），然后初始化厂商自定义数据，设置厂商 ID 为 0x0059，自定义数据指针指向保存自定义的数组，计算自定义数据长度（代码的第 12~17 行），之后设置广播中包含厂商自定义数据（代码的第 29~30 行），这样在广播初始化成功后，厂商自定义数据即会被加入到广播数据中。

代码清单： 广播中加入厂商自定义数据

```
1.  static void advertising_init(void)
2.  {
3.      ret_code_t              err_code;
4.      //定义一个数组，存放自定义的数据，因为 ble_advdata_manuf_data_t 中
5.      //是通过指针来引用数据的，所以必须要定义存放自定义数据的变量，将
6.      //变量的地址赋值给 ble_advdata_manuf_data_t
7.      uint8_t       my_adv_manuf_data[5] = {0x01,0x02,0x03,0x04,0x05};
8.      //定义个一个制造商自定义数据的结构体变量，配置广播数据时将该变量的地址赋值给
9.      //广播数据结构体中的 p_manuf_specific_data
10.     ble_advdata_manuf_data_t manuf_specific_data;
11.
12.     //制造商 ID，0x0059 是 Nordic 的 ID
13.     manuf_specific_data.company_identifier = 0x0059;
14.     //指向制造商自定义的数据
15.     manuf_specific_data.data.p_data = my_adv_manuf_data;
16.     //制造商自定义的数据大小(字节数)
17.     manuf_specific_data.data.size    = sizeof(my_adv_manuf_data);
18.
19.     //定义广播初始化配置结构体变量
20.     ble_advertising_init_t init;
21.     //配置之前先清零
22.     memset(&init, 0, sizeof(init));
23.     //设备名称类型：全称
24.     init.advdata.name_type               = BLE_ADVDATA_FULL_NAME;
25.     //是否包含外观：包含
26.     init.advdata.include_appearance      = true;
27.     //Flag:一般可发现模式，不支持 BR/EDR
```

```
    init.advdata.flags      = BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE;
    //广播中加入制造商自定义数据
    init.advdata.p_manuf_specific_data    = &manuf_specific_data;

    //设置广播模式为快速广播
    init.config.ble_adv_fast_enabled  = true;
    //设置广播间隔和广播持续时间
    init.config.ble_adv_fast_interval = APP_ADV_INTERVAL;
    init.config.ble_adv_fast_timeout  = APP_ADV_DURATION;
    //广播事件回调函数
    init.evt_handler = on_adv_evt;
    //初始化广播
    err_code = ble_advertising_init(&m_advertising, &init);
    APP_ERROR_CHECK(err_code);
    //设置广播配置标记。APP_BLE_CONN_CFG_TAG 是用于跟踪广播配置的标记，这是为未来预留的
    //一个参数，在将来的 SoftDevice 版本中，可以使用 sd_ble_gap_adv_set_configure()配
    //置新的广播配置，当前 SoftDevice 版本（S140 V7.0.1 版本）支持的最大广播集数量为 1，
    //因此 APP_BLE_CONN_CFG_TAG 只能写 1。
    ble_advertising_conn_cfg_tag_set(&m_advertising, APP_BLE_CONN_CFG_TAG);
}
```

13.3.2. 实验步骤

1. 解压“…\4：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 4-9：广播中加入厂商自定义数据”，将解压后得到的文件夹“ble_app_templete”拷贝合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_templete\project\mdk5”目录下的工程“ble_app_templete_s140.uvproj”。
4. 切换到 Target“flash_s140_nrf52_7.0.1_softdevice”，点击下载协议栈。
5. 切换到 Target“IK-52840DK”，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
6. 程序运行后，指示灯 D1 闪烁表示正在广播。
7. 手机上打开 nRF Connect，点击“SCAN”按钮启动扫描，扫描到设备后，展开广播数据即可看到厂商自定义数据，如下图所示。

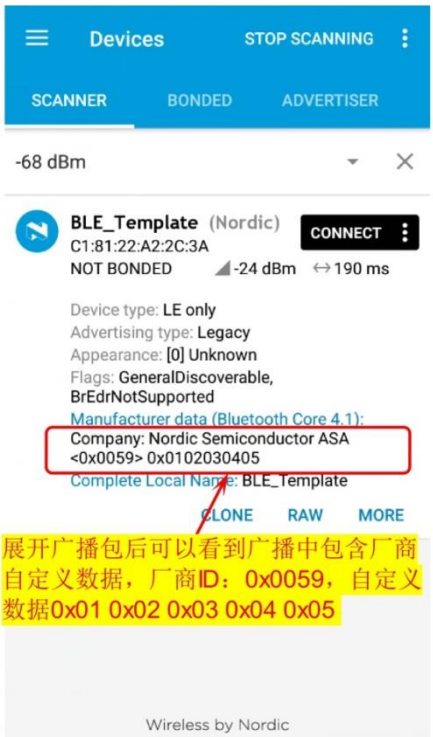


图 4-39：广播数据中包含厂商自定义数据

第五章：创建定时任务

1. 学习目的

3. 掌握如何使用 APP 定时器创建定时任务。
4. 掌握 APP 定时器的两种类型：周期定时器和单次定时器的应用和区别。

2. APP 定时器

Nordic 的 SDK 中创建定时任务是通过 APP 定时器来实现的。

APP 定时器是基于实时计数器 RTC1、通过链表实现的软件定时器，APP 定时器具有软件定时器的一般特征，应用程序可以配置、创建、启动和停止定时器，应用程序也可以同时创建多个定时任务，APP 定时器基本原理框图如下图所示。APP 定时器可以创建两种类型的定时器：单次定时器和周期定时器，它们的区别如下。

- 单次定时器：定时器超时后不会自动启动，需要应用程序去启动。
- 周期定时器：定时器超时后会自动启动，重新开始计时。

APP 定时器启动后，当配置的定时时间到达，即产生超时事件，因此，应用程序需要提供超时事件回调函数接收事件，并在该函数中执行应用程序需要执行的功能代码。

APP 定时器在 RTC1 中断处理程序中执行用户超时处理程序的超时和调用检查，同时使用软中断（SWI0）处理定时列表，SWI0 使用 APP_LOW 中断优先级。

应用程序启动/停止 APP 定时器时，启动/停止操作仅加入队列，并触发软件中断，实际的定时器启动/停止操作由 SWI0 中断处理程序执行。由于 SWI0 中断在 APP_LOW 中断优先级下运行，因此，如果调用定时器函数的应用程序代码在 APP_LOW 或 APP_HIGH 中运行，则在应用程序处理程序返回之前不会执行定时器启动/停止操作。

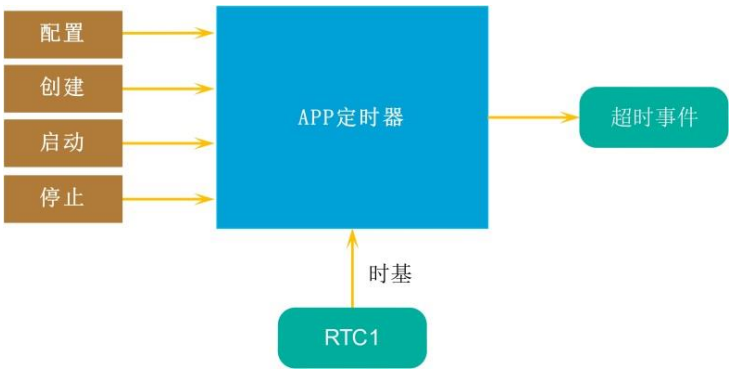


图 5-1：APP 定时器原理框图

■ **APP 定时器的用途主要有以下几种：**

1) APP 按键程序模块使用 APP 定时器消抖以及实现长按和短按。
2) 定时控制 LED 指示灯的亮灭，和 BLE 事件配合实现 BLE 的状态指示。
3) BLE 应用中用于连接参数更新，连接成功后，开启 APP 定时器，超时后进行连接参数

更新。

4) 用户创建定时任务，实现各种定时任务，如创建一个超时时间为 1 秒的 APP 定时器作为实时时钟的时基。

■ **APP 定时器相对于硬件定时器优缺点如下：**

- 缺点：不适合于短时间定时，相对于硬件定时器会运行更多的代码，即占用的 CPU 运行时间更多。
- 优点：可创建的定时器数量多，能实现更长时间的定时，使用更灵活、方便。更重要的是省电，尤其在 BLE 应用中，省电效果更明显，这也是为什么 BLE 应用中绝大多数需要定时的场合会使用 APP 定时器。（BLE 应用使用 APP 定时器更省电的原因：BLE 应用中总是会启动 RTC1，而 APP 定时器是以 RTC1 为时基，因此，使用 APP 定时器增加的耗电量仅仅是软件运行带来的额外电量消耗）。

3. 使用 APP 定时器

APP 定时器应用的流程如下图所示，应用程序需要定义 APP 定时器和超时事件，同时提供超时事件回调函数。在创建 APP 定时器时，应用程序可以根据需求创建单次或周期定时器，创建成功之后，应用程序即可执行定时器启动/停止操作。

需要特别注意的是对于裸机程序，应用程序需要去初始化低频时钟 LFCLK，对于 BLE 应用，应用程序无需初始化低频时钟，因为 SoftDevice 会初始化低频时钟。

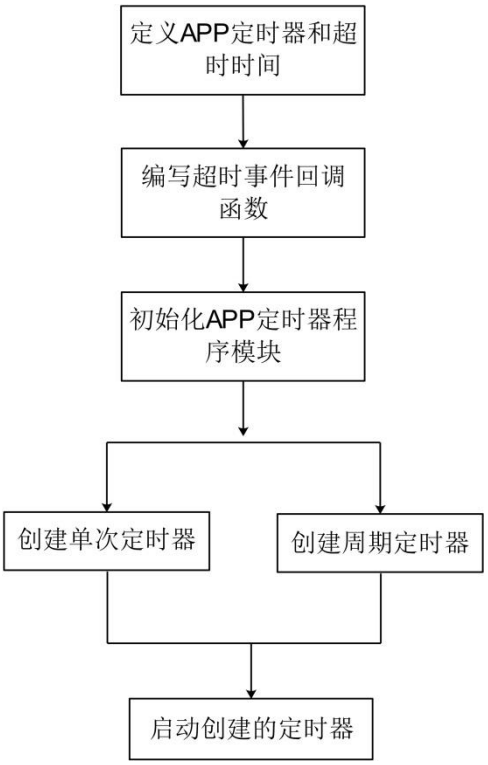


图 5-2：APP 定时器的应用步骤

1. 定义 APP 定时器和超时时间

APP 定时器使用宏 APP_TIMER_DEF 定义具体的 APP 定时器，该宏创建 APP 定时器标识符并为其分配静态内存，之后通过 APP 定时器标识符即可操作该定时器。使用

APP_TIMER_DEF 宏定义 APP 定时器的格式如下，其中定时器标识符也称为定时器 ID，定时器标识符命名不要过于随意，否则，当创建的 APP 定时器比较多的时候，查看起来不方便。

```
APP_TIMER_DEF(定时器标识符);
```

■ 示例：定义一个标志符为 m_beep_timer_id 的 APP 定时器

```
APP_TIMER_DEF(m_beep_timer_id);
```

超时时间即是 APP 定时器的定时时间。APP 定时器的时基是 RTC1，对于 RTC 来说，它只能分辨出 tick（时钟节拍），但是 tick 和时间的对应关系并不明显(需要计算)，如果每次修改定时时间都要计算一下 tick 数，那就太麻烦了，我们希望的是设置定时时间时直接输入时间值，如定时 100ms 时直接设置参数为 100，这样无疑会方便很多。SDK 考虑到了这一点，SDK 专门提供了一个宏 APP_TIMER_TICKS 用来完成 APP 定时器的定时时间和 tick 之间的转换，使用该宏定义超时时间的格式如下：

```
APP_TIMER_TICKS(超时时间，单位 ms)
```

程序中使用时，会通过宏对 APP_TIMER_TICKS()再次定义，这相当于给定义的超时时间取一个更容易记住和理解的“别名”，如程序中定义指示灯翻转的间隔为 500ms 的代码如下，显然，LED_TOGGLE_INTERVAL 比 APP_TIMER_TICKS(500)更容易理解。

```
1. #define LED_TOGGLE_INTERVAL     APP_TIMER_TICKS(500)
```

2. 编写超时事件回调函数

APP 定时器超时后即会回调创建定时器时注册的超时事件回调函数，该回调函数即是应用程序实现功能的地方，在这个函数里面，我们可以编写功能代码，实现所需的功能。

超时事件回调函数的格式如下，其中函数名称必须和创建 APP 定时器时注册的超时事件回调函数一样。

代码清单：APP 定时器超时事件回调函数格式

```
1. static void 函数名称(void * p_context)
2. {
3.     //防止编译器警告，同时清晰地表明 p_context 未使用，而不是应用程序忽略了。如果应用
4.     //程序需要使用 p_context，则不需要这行代码
5.     UNUSED_PARAMETER(p_context);
6.     //加入应用程序需要实现的功能代码
7. }
```

3. 初始化 APP 定时器程序模块

APP 定时器程序模块初始化的库函数是 app_timer_init()函数，该函数会初始化实时时钟 RTC1 并保存 RTC1 当前计数值，但是该函数不会初始化低频时钟 LFCLK，因此，裸机程序需要自己去初始化低频时钟（BLE 应用由 SoftDevice 初始化），app_timer_init()函数原型如下表所示。

表 5-1：app_timer_init()函数

函数原型	ret_code_t app_timer_init(void)
函数功能	初始化 APP 定时器程序模块。
参　　数	无。
返回值	初始化成功后返回 NRF_SUCCESS。

4. 创建 APP 定时器

APP 定时器通过调用 app_timer_create()函数创建，函数原型如下表所示。该函数有 3 个输入参数，其中“p_timer_id”是步骤 1 中定义的 APP 定时器的标识符，“timeout_handler”是步骤 2 中应用程序提供的超时事件回调函数，“mode”是创建的 APP 定时器模式，可创建的模式有如下两种。

- APP_TIMER_MODE_REPEATED：创建周期定时器。
- APP_TIMER_MODE_SINGLE_SHOT：创建单次定时器。

表 5-2：app_timer_create()函数

函数原型	ret_code_t app_timer_create(app_timer_id_t const * p_timer_id, app_timer_mode_t mode, app_timer_timeout_handler_t timeout_handler)
函数功能	创建 APP 定时器实例。 ● APP 定时器创建没有通过临界区进行保护，因此，不能同时从几个中断级别调用它。 ● 应用程序可以在已创建的 APP 定时器实例上再次调用该函数重新初始化 APP 定时器，但 APP 定时器必须先停止运行。 ● FreeRTOS 和 RTX 操作系统中不允许在先前创建的 APP 定时器实例上调用该函数重新初始化 APP 定时器。
参　　数	[in] p_config：指向 APP 定时器标识。 [in] mode：APP 定时器模式。 [in] timeout_handler：超时事件句柄，APP 定时器超时后执行。
返回值	NRF_SUCCESS：创建 APP 定时器成功。 NRF_ERROR_INVALID_PARAM：函数参数无效。 NRF_ERROR_INVALID_STATE：APP 定时器程序模块没有初始化或者创建的 APP 定时器已经在运行中。

5. 启动/停止 APP 定时器

APP 定时器创建成功后并不会启动，需要应用程序去启动。启动和停止一个 APP 定时

器的函数分别是 app_timer_start()和 app_timer_stop()，函数原型如下表所示。

表 5-19：app_timer_start()函数

函数原型	ret_code_t app_timer_start(app_timer_id_t timer_id, uint32_t timeout_ticks, void * p_context)
函数功能	启动指定的 APP 定时器。 ● 最小的超时 tick 数不能小于 5。 ● 对于多个活动的 APP 定时器，若它们发生超时的时间接近（在 1 到 3 个 tick 范围内），那么发生的超时将具有最大 3 个 tick 的正抖动。 ● 如果一个 APP 定时器已经启动运行，调用该函数去启动该 APP 定时器，启动操作被忽略。
参　　数	[in] timer_id：APP 定时器标识。 [in] timeout_ticks：RTC1 的 tick(时钟滴答)数，不能小于 5 ticks。 [in] p_context：传递给超时回调函数。
返回值	NRF_SUCCESS：APP 定时器启动成功。 NRF_ERROR_INVALID_PARAM：函数参数无效。 NRF_ERROR_INVALID_STATE：APP 定时器程序模块没有初始化或者启动的 APP 定时器没有创建。 NRF_ERROR_NO_MEM：APP 定时器操作队列满。

表 5-3：app_timer_stop()函数

函数原型	ret_code_t app_timer_stop(app_timer_id_t timer_id)
函数功能	停止指定的 APP 定时器。
参　　数	[in] timer_id：APP 定时器标识。
返回值	NRF_SUCCESS：APP 定时器停止成功。 NRF_ERROR_INVALID_PARAM：函数参数无效。 NRF_ERROR_INVALID_STATE：APP 定时器程序模块没有初始化或者启动的 APP 定时器没有创建。 NRF_ERROR_NO_MEM：APP 定时器操作队列满。

■ 示例：启动一个 APP 定时器，超时时间：LED_TOGGLE_INTERVAL

代码清单：启动 APP 定时器 m_led_toggle_timer_id

```
//启动 APP 定时器 m_led_toggle_timer_id
err_code = app_timer_start(m_led_toggle_timer_id, LED_TOGGLE_INTERVAL, NULL);
```

```
//检查错误代码
APP_ERROR_CHECK(err_code);
```

■ 示例：停止一个 APP 定时器

代码清单：停止 APP 定时器 m_led_toggle_timer_id

```
//停止 APP 定时器 m_led_toggle_timer_id
err_code = app_timer_stop(m_led_toggle_timer_id);
//检查错误代码
APP_ERROR_CHECK(err_code);
```

4. 裸机创建定时任务实验

本实验在开发指南上册“实验 20-1：RTC TICK 事件实现定时”的基础上修改。本例中创建 2 个 APP 定时器。

- LED 定时器：标识符 `m_led_toggle_timer_id`，周期定时器，超时时间 100ms，超时事件中翻转指示灯 D1 的状态。按下 S1 按键启动 LED 定时器，按下 S2 按键停止 LED 定时器。
- 蜂鸣器定时器：标识符 `m_beep_timer_id`，单次定时器，超时时间 500ms。按下 S3 按键后鸣响蜂鸣器，并启动蜂鸣器定时器，超时事件中关闭蜂鸣器鸣响。

✧ 注：本节对应的实验源码是：“实验 5-1：裸机使用 APP 定时器创建定时任务”。

4.1.1. 添加需要的文件

RTC 试验中已经加入了低频时钟所需的文件，因此，这里只需加入 APP 定时器需要的文件即可，另外，因为 App Timer v2 不再使用“nrfx_rtc.c”作为 RTC 的驱动文件，所以本例中可以删除“nrfx_rtc.c”文件，需要加入的文件如下表所示。

表 5-4： APP 定时器需要加入的文件

文件名	SDK 中的目录	描述
app_timer2.c	..\components\libraries\timer	APP 定时器驱动文件。从 SDK16.0 开始App Timer v2 已退出实验阶段并作为正式发布版取代之前的 App Timer。
drv_rtc.c	..\components\libraries\timer	App Timer v2 使用的RTC驱动文件
nrf_sortlist.c	..\components\libraries\sortlist	排序列表库文件。
nrf_atfifo.c	..\components\libraries\atomic_fifo	atomic FIFO。

4.1.2. 头文件引用和路径设置

1. 需要引用的头文件

因为在“main.c”文件中使用了 APP 定时器程序模块，所以“main.c”文件需要引用下面的头文件。

代码清单：引用的头文件

```
#include "nrf_drv_clock.h"
#include "app_timer.h"
```

2. 需要添加的头文件包含路径

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

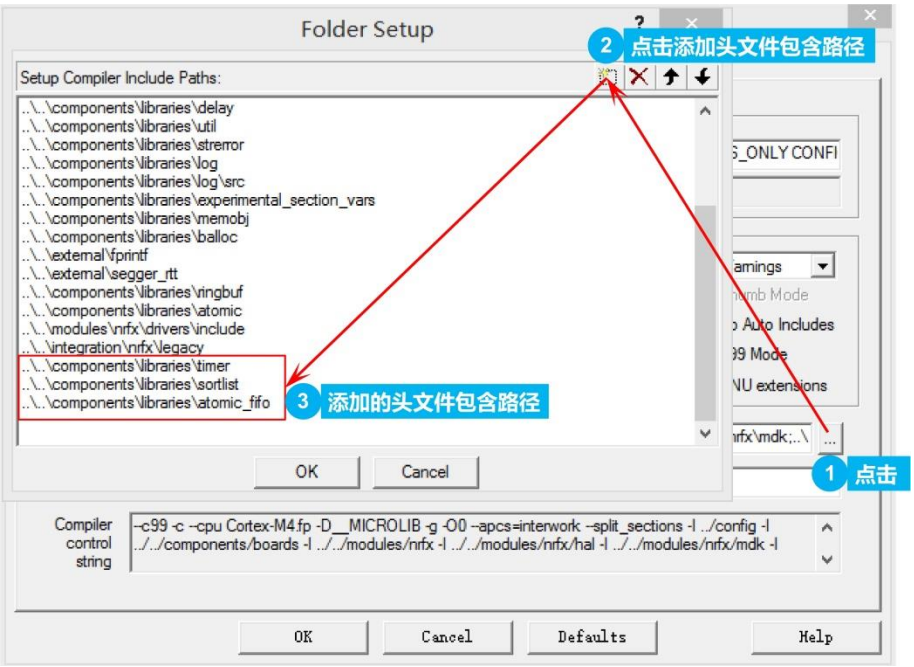


图 5-3：添加头文件包含路径

APP 定时器需要添加的头文件路径如下表：

表 5-5： 头文件包含路径

序号	路径
1	..\..\components\libraries\timer
2	..\..\components\libraries\sortlist
3	..\..\components\libraries\atomic_fifo

3. 全局宏定义中加入“APP_TIMER_V2 APP_TIMER_V2_RTC1_ENABLED”

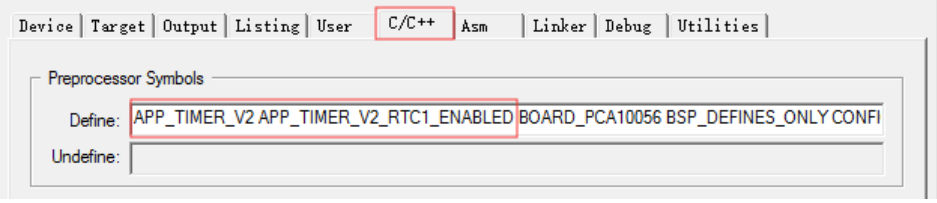


图 5-4：全局宏定义中加入 APP 定时器的宏

4.1.3. 工程配置

打开“sdk_config.h”文件，加入低频时钟 LFCLK 和 RTC 所需的配置，如下图所示，编辑内容时切换到“Text Editor”进行编辑（编辑的内容参考实验的源码，因此代码很长，此处不贴出代码，参见“sdk_config.h”文件的（259~286）行），编辑完成后，切换到“Configuration Wizard”，即可观察到配置的具体项目。

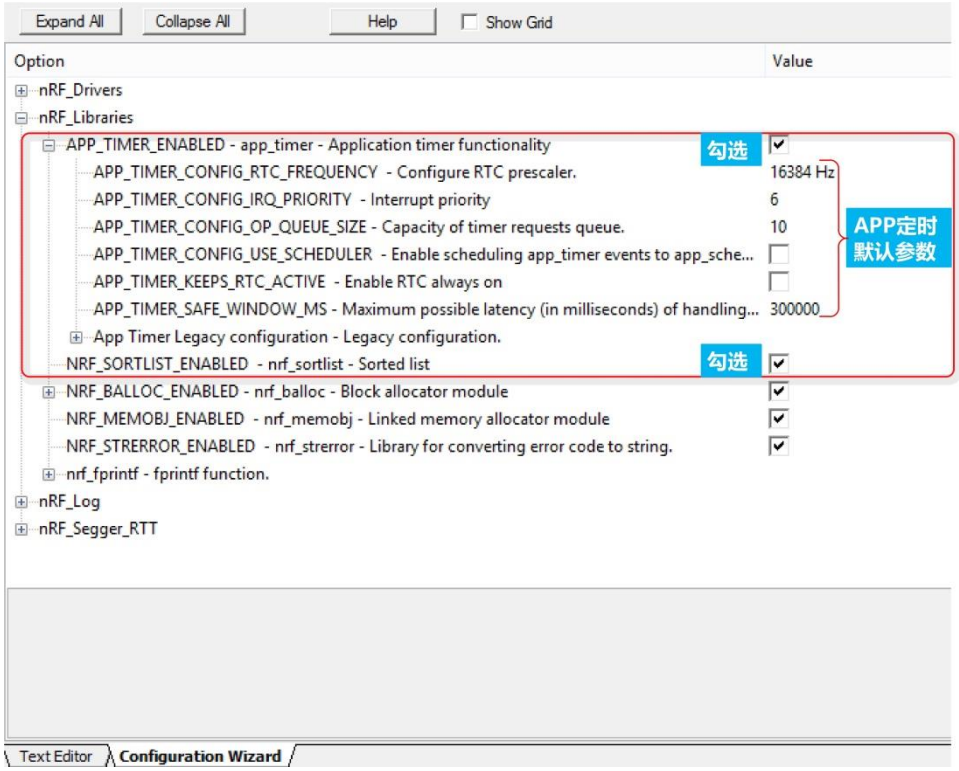


图 5-5：工程配置

4.1.4. 代码编写

使用 APP 定时器时，首先定义 APP 定时器和超时时间，本例定义 2 个 APP 定时器 m_led_toggle_timer_id 和 m_beep_timer_id，代码清单如下。

代码清单：定义 APP 定时器和超时时间

```
1.  //定义 APP 定时器 m_led_toggle_timer_id（LED 定时器），用于驱动指示灯 D1，每 100ms 翻转一
2.  //次 D1 的状态
3.  APP_TIMER_DEF(m_led_toggle_timer_id);
4.
5.  //定义 APP 定时器 m_beep_timer_id（蜂鸣器定时器），用于驱动蜂鸣器鸣响，鸣响时间 500ms
6.  APP_TIMER_DEF(m_beep_timer_id);
7.
8.  //LED 定时器超时时间：100ms
9.  #define LED_TOGGLE_INTERVAL          APP_TIMER_TICKS(100)
10. //蜂鸣器定时器超时时间：500ms
11. #define BEEP_INTERVAL                APP_TIMER_TICKS(500)
```

接下来编写 2 个 APP 定时器的超时事件回调函数，函数名称分别命名为 led_toggle_timeout_handler 和 beep_timeout_handler。led_toggle_timeout_handler()函数中加入翻转 LED 指示灯 D1 的代码，即 LED 定时器每次超时后翻转一次 D1 状态，beep_timeout_handler()函数中，加入关闭蜂鸣器鸣响的代码，即蜂鸣器定时器启动时驱动蜂鸣器鸣响，蜂鸣器定时器超时后关闭鸣响，从而达到驱动蜂鸣器鸣响指定时长的目的。

代码清单：编写 2 个 APP 定时器的超时事件回调函数

```
//LED 定时器超时事件回调函数
static void led_toggle_timeout_handler(void * p_context)
{
    //防止编译器警告，同时清晰地的表明 p_context 未使用，而不是应用程序忽略了。如果应用
    //程序需要使用 p_context，则不需要这行代码
    UNUSED_PARAMETER(p_context);
    //翻转指示灯 D1 的状态
    nrf_gpio_pin_toggle(LED_1);
}

//蜂鸣器定时器超时事件回调函数
static void beep_timeout_handler(void * p_context)
{
    //防止编译器警告，同时清晰地的表明 p_context 未使用，而不是应用程序忽略了。如果应用
    //程序需要使用 p_context，则不需要这行代码
    UNUSED_PARAMETER(p_context);
    //蜂鸣器定时器超时后，停止蜂鸣器鸣响
    nrf_gpio_pin_clear(BEEP_P12);
}
```

编写完 APP 定时器超时事件回调函数之后，即可创建 APP 定时器，当然，在创建之前还需要调用 app_timer_init()初始化 APP 定时器程序模块。本例中创建了 2 个 APP 定时器，其中 LED 定时器是周期定时器，调用 app_timer_create()函数创建时模式设置为 APP_TIMER_MODE_REPEATED，蜂鸣器定时器是单次定时器，调用 app_timer_create()函数创建时模式设置为 APP_TIMER_MODE_SINGLE_SHOT。

代码清单：初始化 APP 定时器程序模块和创建 APP 定时器

```
//初始化 APP 定时器程序模块和创建 APP 定时器
static void timers_init(void)
{
    ret_code_t err_code;
    //初始化 APP 定时器程序模块
    err_code = app_timer_init();
    //检查错误代码
    APP_ERROR_CHECK(err_code);

```

```
    //创建 APP 定时器：LED 定时器，周期定时器，启动后周期运行，用于驱动指示灯闪烁
    err_code = app_timer_create(&m_led_toggle_timer_id,
                                APP_TIMER_MODE_REPEATED,
                                led_toggle_timeout_handler);
    //检查错误代码
    APP_ERROR_CHECK(err_code);
    //创建 APP 定时器：蜂鸣器定时器，单次定时器，启动后只运行一次，用于驱动蜂鸣器鸣响
    err_code = app_timer_create(&m_beep_timer_id,
                                APP_TIMER_MODE_SINGLE_SHOT,
                                beep_timeout_handler);
     //检查错误代码
     APP_ERROR_CHECK(err_code);
}
```

对于裸机程序中使用 APP 定时器，需要应用程序初始化低频时钟 LFCLK，因为 APP 定时器需要使用 RTC1，而 RTC1 的时钟源是低频时钟。

代码清单：初始化低频时钟

```
//配置低频时钟，时钟源设置的是外部 32.768KHz 晶体
static void lfclk_config(void)
{
    //初始化时钟模块，设置低频时钟源
    ret_code_t err_code = nrf_drv_clock_init();
    APP_ERROR_CHECK(err_code);
    //请求低频时钟，输入参数为 NULL 表示低频时钟启动后不产生事件
    nrf_drv_clock_lfclk_request(NULL);
}
```

APP 定时器创建完成后，即可根据需求启动或停止定时器，本例中按下 S1 按键启动 LED 定时器，按下 S2 按键停止 LED 定时器。按下 S3 按键后鸣响蜂鸣器，并启动蜂鸣器定时器，超时事件中关闭蜂鸣器鸣响，代码清单如下：

代码清单：主函数

```
/*****************************************************************************
* 描   述 : main 函数
* 入   参 : 无
* 返回值 : int 类型
*****************************************************************************/
int main(void)
{
   ret_code_t err_code;

   //初始化开发板上的 4 个 LED 和 4 个按键
   bsp_board_init(BSP_INIT_LEDS | BSP_INIT_BUTTONS);
```

```
12.   nrf_gpio_cfg_output(BEEP_P12);
13.   nrf_gpio_pin_clear(BEEP_P12);
14.   //初始化 APP 定时器程序模块和创建 APP 定时器
15.   timers_init();
16.
17.   //初始化低频时钟 LFCLK
18.   lfclk_config();
19.
20.   while (true)
21.   {
22.      //检测按键 S1 是否按下,S1 按键按下后启动 LED 定时器
23.      if(nrf_gpio_pin_read(BUTTON_1) == 0)
24.      {
25.         //软件消抖:本例为了演示 APP 定时器的应用，故按键检测使用了简单的软件消抖，而没
26.         //有使用 APP 按键程序模块处理抖动
27.         nrf_delay_ms(20);
28.         //按键 S1 有效
29.         if(nrf_gpio_pin_read(BUTTON_1) == 0)
30.         {
31.            //启动 LED 定时器
32.             err_code = app_timer_start(m_led_toggle_timer_id, LED_TOGGLE_INTE
33.                                        RVAL, NULL);
34.             APP_ERROR_CHECK(err_code);
35.          }
36.       }
37.       //检测按键 S2 是否按下
38.       if(nrf_gpio_pin_read(BUTTON_2) == 0)
39.       {
40.          //软件消抖
41.          nrf_delay_ms(20);
42.          //按键 S2 有效
43.          if(nrf_gpio_pin_read(BUTTON_2) == 0)
44.          {
45.             //停止 LED 定时器
46.             err_code = app_timer_stop(m_led_toggle_timer_id);
47.             APP_ERROR_CHECK(err_code);
48.          }
49.        }
50.        //检测按键 S3 是否按下
51.        if(nrf_gpio_pin_read(BUTTON_3) == 0)
52.        {
53.           //软件消抖
54.           nrf_delay_ms(20);
55.           //如果按键 S3 有效
```

```
            if(nrf_gpio_pin_read(BUTTON_3) == 0)
            {
              //鸣响蜂鸣器
              nrf_gpio_pin_set(BEEP_P12);
              //启动蜂鸣器定时器，超时后关闭蜂鸣器鸣响
              err_code = app_timer_start(m_beep_timer_id, BEEP_INTERVAL, NULL);
              APP_ERROR_CHECK(err_code);
            }
        }
    }
}
```

4.1.5. 硬件连接

本章的 2 个实验需要使用 P0.17~P0.20 驱动 4 个 LED 指示灯，P0.13~P0.16 检测 4 个按键， P0.06 和 P0.08 作为串口通讯引脚，后按照下图所示短接跳线帽。

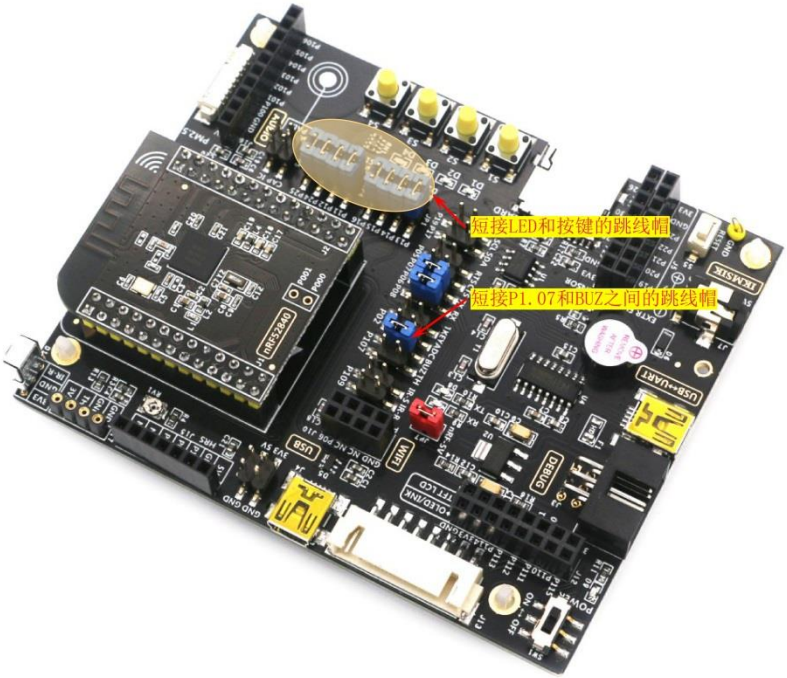


图 5-6：开发板跳线帽短接

4.1.6. 实验步骤

1. 解压“…\3：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 5-1：裸机使用 APP 定时器创建定时任务”，将解压后得到的文件夹“app_timer”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\app_timer\project\mdk5”目录下的工程“app_timer.uvproj”。

4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，按一次 S1 按键，启动 LED 定时器，这时会观察到 LED 指示灯 D1 以 100ms 的间隔闪烁，按一次 S2 按键，停止 LED 定时器，这时会观察到 D1 停止闪烁。按一次 S3 按键，蜂鸣器开始鸣响，同时启动蜂鸣器定时器，可以观察到蜂鸣器鸣响 500ms 后停止鸣响。

5. BLE 应用创建定时任务实验

本实验在开发指南下册“实验 3-3：BLE 工程模板（外围设备）”的基础上修改。本例中创建一个超时时间为 200ms 的周期性 APP 定时器，定时器超时事件中翻转 LED 指示灯 D4 的状态，即每 200ms 驱动 D4 状态翻转一次。

✧ 注：本节对应的实验源码是：“实验 5-2：BLE 应用中使用 APP 定时器创建定时任务”。

5.1.1. 代码编写

BLE 应用中使用 APP 定时器和裸机下使用的流程一样，区别是 BLE 应用中不需要加入初始化低频时钟的代码，因为 SoftDevice 会初始化低频时钟，也不需要初始化 APP 定时器程序模块，因为即使最简单的 BLE 应用中也会使用 APP 定时器更新连接参数，所以，BLE 应用中都会初始化 APP 定时器程序模块。

下面是 BLE 应用中使用 APP 定时器需要加入的代码，读者可以和裸机程序对比阅读，看一下他们的差别。

首先定义 APP 定时器和超时时间，本例定义 1 个 APP 定时器 m_led_toggle_timer_id，并定义该定时器超时时间为 200ms，代码清单如下。

代码清单：定义 APP 定时器和超时时间

```
//定义 APP 定时器 m_led_toggle_timer_id（LED 定时器），用于驱动指示灯 D4，每 200ms 翻转一
//次 D4 的状态
APP_TIMER_DEF(m_led_toggle_timer_id);
//LED 定时器超时时间：200ms
#define LED_TOGGLE_INTERVAL          APP_TIMER_TICKS(200)
```

接下来编写 APP 定时器的超时事件回调函数，函数名称命名为 led_toggle_timeout_handler。led_toggle_timeout_handler()函数中加入翻转 LED 指示灯 D1 的代码，即 LED 定时器每次超时后翻转一次 D1 状态。

代码清单：编写 APP 定时器的超时事件回调函数

```
//LED 定时器超时事件回调函数
```

```
static void led_toggle_timeout_handler(void * p_context)
{
    //防止编译器警告，同时清晰地的表明 p_context 未使用，而不是应用程序忽略了。如果应用
    //程序需要使用 p_context，则不需要这行代码
    UNUSED_PARAMETER(p_context);
    //翻转指示灯 D4 的状态
    nrf_gpio_pin_toggle(LED_4);
}
```

timers_init()函数中加入创建 APP 定时器的代码，因为 BLE 应用中已经初始化了 APP 定时器程序模块，所以只需创建 APP 定时器即可。

代码清单：创建 APP 定时器

```
//初始化 APP 定时器模块、创建 APP 定时器
static void timers_init(void)
{
    //初始化 APP 定时器模块
    ret_code_t err_code = app_timer_init();
    //检查返回值
    APP_ERROR_CHECK(err_code);
    //创建 APP 定时器：LED 定时器，周期定时器，启动后周期运行，用于驱动指示灯闪烁
    err_code = app_timer_create(&m_led_toggle_timer_id,
                                APP_TIMER_MODE_REPEATED,
                                led_toggle_timeout_handler);
    APP_ERROR_CHECK(err_code);
}
```

APP 定时器创建完成后，即可根据需求启动或停止定时器。本例中是实现驱动 LED 指示灯 D4 以 200ms 的间隔闪烁，因此，创建 APP 定时器后，启动定时器即可，代码清单如下。这里注意一下，对于 BLE 应用，我们习惯于把所有需要在广播启动之前启动的 APP 定时器全部放到 application_timers_start()函数里面，以方便管理。

代码清单：启动 APP 定时器

```
//启动已经创建的 APP 定时器
static void application_timers_start(void)
{
    ret_code_t err_code;
    //启动 LED 定时器，定时器超时间隔为 LED_TOGGLE_INTERVAL，即 200ms
    err_code = app_timer_start(m_led_toggle_timer_id, LED_TOGGLE_INTERVAL, NULL);
    //检查函数返回的错误代码
    APP_ERROR_CHECK(err_code);
}
```

5.1.2. 硬件连接

同“实验 5-1：裸机使用 APP 定时器创建定时任务”。

5.1.3. 实验步骤

1. 解压“…\3：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 5-2：BLE 应用中使用 APP 定时器创建定时任务”，将解压后得到的文件夹“ble_app_templete”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_templete\project\mdk5”目录下的工程“ble_app_templete_s140.uvproj”。
4. 切换到 Target“flash_s140_nrf52_7.0.1_softdevice”，点击下载协议栈。
5. 切换到 Target“IK-52840DK”，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
6. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
7. 程序运行后，开发板上的指示灯 D4 以 200ms 的间隔闪烁。

第六章：实现一个标准的 Profile

1. 学习目的

1. 本章根据 SIG 发布的心率配置文件编写程序实现一个最简化的心率 Profile，为了便于理解如何根据 SIG 的文档实现标准的 Profile 以及如何加入服务、特征等，相对于 Nordic SDK16.0 中的心率 demo，去掉了一些附加的部分：
   - 没有实现配对绑定。
   - 没有加入电池服务。
   - 没有通过软件模拟动态变化的心率数据，仅发送固定的数据。
2. 读者在阅读本章时，对于 profile、服务和特征，应将注意力放在以下几个方面。
   - 如何根据 SIG 的文档实现标准的 Profile。
   - 加入服务和特征的一般方法。
   - 服务结构体、服务初始化结构体的声明和注册服务的 BLE 事件监视者。
3. 掌握 BLE 的数据传输。
   - 从机主动发起数据传输的方式。
   - 主机主动发起数据传输的方式。
   - 如何发送数据。

2. 配置文件、服务和特征

2.1. Profile(配置文件)

配置文件是一种规范，配置文件定义为可能的应用，并规定了蓝牙设备与其它蓝牙设备通信的一般行为。配置文件构建于蓝牙标准基础上，以更明确界定蓝牙模块传输的数据类型。设备的应用决定了它需要支持的配置文件种类。

如心率配置文件中，定义了两种角色、规定了心率传感器必须包含的服务、建议的广播参数以及绑定和非绑定下建立连接的过程等等。

■ **标准配置文件和自定义配置文件**

为了方便用户使用，体现 BLE 的互操作性，SIG 采纳了很多常用 Profile 并在官网上发布，这些 Profile 称为标准 Profile，如心率 Profile，防丢器 Profile、健康体温计 Profile 等。但是在实际应用中，应用场景多种多样，厂商对功能的设计需求也各不相同，仅仅依靠标准 Profile 是无法满足这些需求的，因此，蓝牙核心规范允许厂商定义适用于自己产品需求的 Profile，这些 Profile 称为自定义 Profile，如串口透传 Profile、蓝牙灯控 Profile 等等。

因此，配置文件分两种类型：标准配置文件和自定义配置文件，如下图所示。

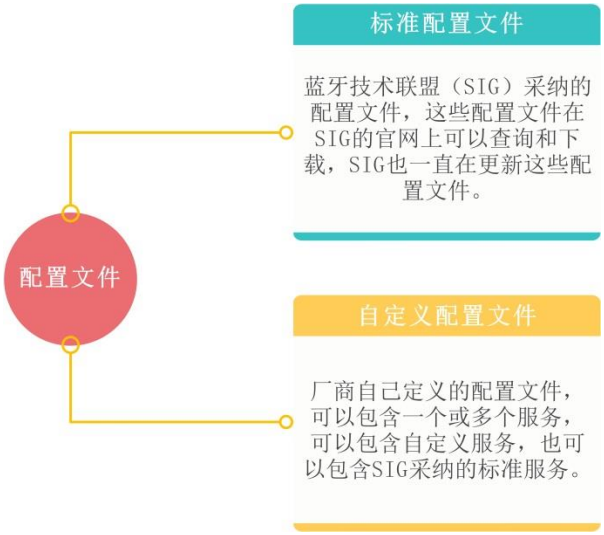


图 6-1：标准 Profile 和自定义 Profile

标准 Profile 的规格书在 SIG 的官网可以查询和下载（SIG 已采纳的配置文件：https://www.bluetooth.com/specifications/gatt）。下表列出的是 SIG 采纳的一部分配置文件。

表 6-1：部分 SIG 采纳的配置文件

配置文件规格	版本	状态	已采纳日期
ANP 警报通知配置文件	1.0	使用中	2011 年 9 月 13 日
BLP 血压配置文件	1.0	使用中	2011 年 10 月 25 日
GLP 血糖配置文件	1.0	使用中	2012 年 4 月 10 日
CPP 自行车功率配置文件	1.1	使用中	2016 年 5 月 3 日
CSCP 自行车速度及节奏配置文件	1.0	使用中	2012 年 8 月 21 日
FMP Find Me 配置文件	1.0	使用中	2011 年 6 月 21 日
HRP 心率配置文件	1.0	使用中	2011 年 7 月 12 日

2.2. Service(服务)

服务是一组特征和通过它们所公开的行为的集合，一个服务可以包含多个特征。如心率服务包含 Heart Rate measurement、Body Sensor Location 和 Heart Rate Control Point 这些特征。服务也和 Profile 一样，分为标准服务和自定义服务，标准服务是由 SIG 采纳发布的，而自定义服务是厂商自己定义的，下表中列出了一部分 SIG 发布的标准服务，同样，这些标准服务的规格书也可以在 SIG 的官网查询和下载的，下载的网址和 Profile 一样，下表中列出了一部分 SIG 采纳的服务。

表 6-2：部分 SIG 发布的服务

配置文件规格	版本	状态	已采纳日期
ANS 警报通知服务	1.0	使用中	2011 年 9 月 13 日

BAS 电池服务	1.0	使用中	2011 年 12 月 27 日
BLP 血压服务	1.0	使用中	2011 年 10 月 25 日
CTS 当前时间服务	1.0	使用中	2014 年 10 月 7 日
DIS 设备信息服务	1. 1	使用中	2011 年 11 月 29 日
HIDS HID 服务	1.0	使用中	2011 年 12 月 27 日
HRS 心率服务	1.0	使用中	2011 年 7 月 12 日

服务存在于从机中，每个服务代表从机的一个能力，Profile 规格书中规定了需要包含哪些服务，哪些服务是必须包含的，哪些服务是可选择性包含的。下图中是一个心率的配置文件定义的心率传感器需要包含的服务，由图中可以看出，心率传感器必须包含心率服务和设备信息服务。

图 6-2：心率配置文件

2.3. Characteristic(特征) 和描述符

1. 特征

特征包含三个部分：声明、数值和描述符，其中声明和数值是必不可少的，而描述符可以是一个或多个。

特征的数值是我们最终要操作的数据，如心率传感器的心率测量特征，它的特征值就是心率值和 RR-Interval 值，心率传感器通过将此特征值发送给主机，从而使能主机可以获取心率传感器的心率值和 RR-Interval 值，当然，主机也可以主动读取心率测量特征来获取这心率传感器的心率值和 RR-Interval 值。

通过无线传输数据本身是一件简单的事情，但是 BLE 为了达到低功耗、互操作性以及适应各种场景的目的，给这些数值设定了一些规范，从而让事情变得复杂很多。换句话说就是要让数据传输变得简单、可靠、通用，那么事先定的规则就多，规则多了，理解起来就会

困难很多。

服务的规格书中会定义服务需要包含哪些特征，下图是心率服务规格书中定义的心率服务包含的特征，其中心率测量特征是强制包含的，身体测点位置特征是可选择包含的，心率 Control Point 是条件包含的（如果支持能量消耗统计，必须包含，否则，不能包含该特征）。

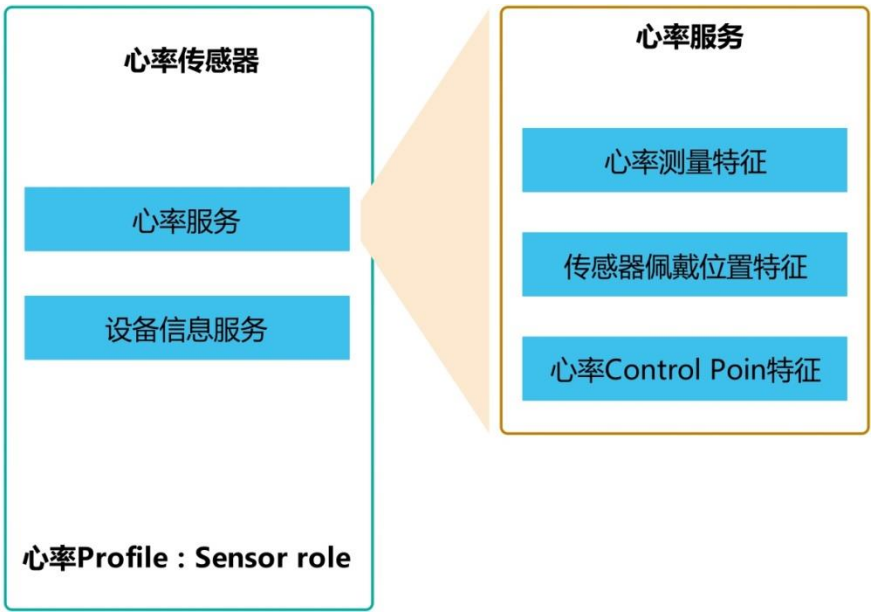


图 6-3：心率服务的特征

对于特征来说，要建立一个特征，首先要进行声明，如下图所示：

图 6-4：特征声明

■ **特征性质**

特征性质是一个 8 位字段，确定了特征数值属性对一系列操作的支持情况，各个位的描述如下表所示：

表 6-3：特征性质

性质	值	描述
广播（Broadcast）	0x01	广播位，如果置位，必须要有服务器特征配置描述符（CCCD）。
读（Read）	0x02	置位表示允许读。
无响应写（Write Without Response）	0x04	无需响应写入。
写（Write）	0x08	置位表示允许写。
读（Notify）	0x10	通知位，如果置位，必须要有客户端特征配置描述符(CCCD)。
指示（Indicate）	0x20	指示位，如果置位，必须要有客户端特征配置描

		述符(CCCD)。
签名认证写入（Authenticated Signed Writes）	0x40	签名认证写入。
扩展特征（Extended Properties）	0x80	扩展性质位。

■ **特征数值**

特征数值是一个属性，特征性质字段给出了特征数值属性的访问权限，如读/写等等，通过特征性质字段的描述，确定了特征数值属性可以执行的操作类型。

2. 描述符

特征的描述符大多数描述符是可选的，一个特征可以包含 0 到多个描述符，需要注意的是如果使用了通知(notify)或指示(indicate)，必须要有客户端特征配置描述符(Client Characteristic Configuration Descriptor，CCCD)，CCCD 的 UUID 是 0x2902。

3. 需求描述

心率 Profile 给出了明确的规定：如果一个设备声称符合心率 Profile，应以服务指定的方式支持此服务中所有指示的强制功能（过程强制），这也同样适用于指示的可选的或有条件具备的功能。因此，我们需要阅读 SIG 发布的相关文档，根据文档的描述列出需求，之后才能编写代码实现需求。对于实现心率 Profile，至少需要阅读下表中的三个文档才能列出需求。

表 6-4：部分 SIG 发布的服务 Heart Rate Sensor

文档名称	版本	描述
Heart Rate Profile(HRP)	V10	心率配置文件。
Heart Rate Service(HRS)	V10	心率服务。
Device Information Service(DIS)	V11r00	设备信息服务。

这些文档从 SIG 的网站上可以下载到，下载的网址见《2.1 节：Profile（配置文件）》。打开网页后，可以看到 SIG 发布的 profile 和服务文档，如下图所示，红框中是心率相关的配置文件和服务。

CTS	Current Time Service		1.1	Active	07 Oct 2014
DIS	Device Information Service	设备信息服务	1.1	Active	29 Nov 2011
ESP	Environmental Sensing Profile		1.0	Active	18 Nov 2014
ESS	Environmental Sensing Service		1.0	Active	18 Nov 2014
FMP	Find Me Profile		1.0	Active	21 Jun 2011
FTMP	Fitness Machine Profile		1.0	Active	14 Feb 2017
FTMS	Fitness Machine Service		1.0	Active	14 Feb 2017
GSS	GATT Specification Supplement		1.0	Active	12 Jul 2017
GLP	Glucose Profile		1.0	Active	10 Apr 2012
GLS	Glucose Service		1.0	Active	10 Apr 2012
HIDS	HID Service		1.0	Active	27 Dec 2011
HOGP	HID over GATT Profile		1.0	Active	27 Dec 2011
HPS	HTTP Proxy Service		1.0	Active	06 Oct 2015
HRP	Heart Rate Profile	心率配置文件	1.0	Active	12 Jul 2011
HRS	Heart Rate Service	心率服务	1.0	Active	12 Jul 2011
HTP	Health Thermometer Profile		1.0	Active	24 May 2011
HTS	Health Thermometer Service		1.0	Active	24 May 2011
IAS	Immediate Alert Service		1.0	Active	21 Jun 2011

图 6-5：SIG 网站上心率相关的配置文件和服务

3.1. 角色

心率 Profile 定义了如下图所示的两类角色：心率传感器和集中器（Collector），心率传感器是测量心率和其它信息的设备，集中器是接收心率传感器心率测量值和其它信息的设备。

- 心率传感器应是 GATT 服务器。
- 集中器应是 GATT 客户端。

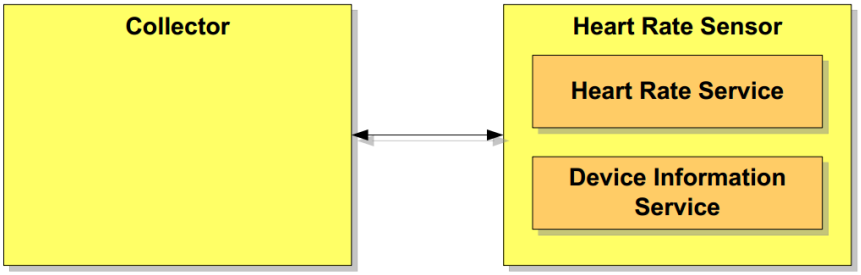


图 6-6：心率 Profile 定义的两类角色

本章我们要实现的是心率传感器的角色，心率传感器实例化一个并且只能实例化一个心

率服务和实例化一个设备信息服务。

心率 Profile 具有传输依赖性，心率 Profile 应在 LE 上传输。

3.2. 需求表和依据

本小节我们先列出需求表，然后按照表中的序号逐条解释表中列出的各个项目的依据，这样更容易理解。

表 6-5：心率传感器（从机）需求表

序号	项目	名称	内容
1	广播数据	设备名称	HeartSensor。
		服务 UUID	0x180D（心率服务 UUID），0x180A（设备信息服务 UUID）。广播包中包含全部服务的 UUID。
		外观	心率腕带，编码 833。
		Flags	一般可发现模式，不支持 BR/EDR。
2	广播参数	广播间隔	187.5 ms。
		广播超时时间	无限广播，不超时。
		广播模式	快速广播。
3	首选连接参数	最小连接间隔	400ms。
		最大连接间隔	650ms。
		从机延迟	0。
		监督超时	4 秒。
4	连接参数协商	首次协商延时	5 秒。
		每次协商之间的间隔	30 秒。
		最次协商最大尝试次数	3 次。
5	GAP 服务	设备名称	心率传感器可以支持设备名称特性的写入属性，以便允许收集器将设备名称写入心率传感器。
6	包含的服务	心率服务	见《心率服务包含的特征》。
		设备信息服务	见《设备信息服务包含的特征》。
7	指示灯	D1 闪烁	指示正在广播。

		D1 常亮	指示已经和中心设备建立连接。
8	APP 定时器	心率测量特征值通知发送定时器。	周期性定时器，超时时间 1 秒。
		传感器接触状态更新定时器	周期性定时器，超时时间 5 秒。

表 6-6：心率服务包含的特征

服务	特征名称	特征性质	安全许可
心率服务	心率测量	Notify（通知）	无
	心率测量客户端特征描述符（CCCD）	读，写	无
	传感器身体测点位置	读	无

表 6-7：设备信息服务包含的特征

服务	特征名称	特征性质	安全许可
设备信息服务	Manufacturer Name String	读	无

3.2.1. 广播数据

一般地，建议在广播数据中至少包含设备名称、Flags、外观和首要服务的 UUID 列表。心率 Profile 规格书的 3.1 节也对此提出了如下需求（注意关键字“宜”表明以下两条需求不是强制性的需求）。

1. 心率传感器的广播数据中宜包含心率服务的 UUID，这样，Collector 在发起连接前就可以识别到该设备是心率传感器，这会增强用户使用的体验。
2. 为了增强用户使用的体验，广播数据或扫描响应数据中宜包含设备名称。

因此，我们在广播包中包含心率服务的 UUID、设备信息服务的 UUID（虽然心率 Profile 规格书中没有提到建议包含，但是按照一般的做法，设备信息服务是首要服务，本例中也包含此 UUID），同时，在广播包中包含设备名称。

外观可以让主机扫描到心率传感器后在列表上显示一个小图标，更加直观，本例中，广播包中包含外观，并设置外观为心率腕带，查询外观编码可以知道心率带的外观编码是 833。

我们使用的芯片 nRF52840 是单模的，仅支持 LE，同时心率 Profile 规格书的《2.5 节：传输依赖性》规定：心率 Profile 应在 LE 上传输。所以，我们应在 Flags 中表明心率传感器不支持 BR/EDR。

心率 Profile 规格书的《5 节：连接建立规程》规定：心率传感器宜使用 GAP 一般可发现模式，本例中我们使用一般可发现模式。

由此可知，Flags 设置为：一般可发现模式，不支持 BR/EDR。

3.2.2. 广播参数

心率 Profile 规格书的《第 5 节：连接建立规程》给出了广播参数的推荐值，如下面表

所示。该推荐值将广播分为两个阶段，前 30 秒广播间隔小，主机容易扫描到心率传感器，并能快速建立连接，但是消耗的能量较大。30 秒之后广播间隔加大，以节省能量消耗，此时主机扫描心率传感器的速度以及建立连接的速度会变慢。

广播分为两个阶段的原因是：一般认为，当一个设备启动时，是最有可能被连接的，所以，在前 30 秒内快速广播，以达到快速连接的目的。30 秒后，被连接的可能性变小，这时候，降低广播速度以节省能量。

表 6-8：推荐的广播参数

广播历时	参数	值
前 30 秒（快速连接）	广播间隔	（20~30）ms
30 秒后（节省能量）	广播间隔	（1~2.5）s

本例中，我们不采用心率 Profile 规格书中推荐的广播参数，使用如下的广播参数。

- 广播间隔：187.5 ms。
- 广播历时：无限广播，不超时。
- 广播模式：快速广播，这是 Nordic 的 SDK 里面定义的模式，是一个相对的概念，并不是蓝牙的标准定义的。

3.2.3. 首选连接参数和连接参数协商

首选连接参数是从机倾向的连接参数，用户可以根据自己的需求设置首选连接参数，本例中设置如下：

- 最小连接间隔：400ms。
- 最大连接间隔：650ms。
- 从机延迟：0，即不延迟。
- 监督超时：4 秒。

从机和主机刚建立连接时，会使用“快速”的连接参数（主机在连接请求中提供的连接参数）以达到迅速交换信息的目的。但是，从机和主机之间并不是一直需要快速的交换信息，因为这样会加快从机电池能量的消耗，因此，从机希望能采用自己倾向的连接参数。这时，从机就需要和主机协商连接参数。

连接参数协商一般会在连接建立成功后延时一段时间进行，因为在连接建立的过程中交互的数据较多，延时一段时间可以避开“数据高峰”，在相对“空闲”阶段完成协商，以免影响连接建立的速度。除了延时，应用程序还需要配置连接参数协商的时间间隔和最大尝试次数，本例中使用的协商参数如下。

- 首次协商延时：5 秒。
- 每次协商之间的间隔：30 秒。
- 最次协商最大尝试次数：3 次。

3.2.4. GAP 服务

心率 Profile 规格书的《第 3 节：心率传感器角色需求》给出了对 GAP 服务设备名称的

额外需求：心率传感器可以支持设备名称特征的写入属性，以便允许 Collector 将设备名称写入心率传感器，即主机可以修改 GAP 的设备名称。

本例中设置 GAP 的设备名称特征可写，允许主机修改。

3.2.5. 包含的服务

心率 Profile 规格书的《第 3 节：心率传感器角色需求》描述了心率传感器需要包含的服务，如下表所示，心率传感器强制包含心率服务和设备信息服务。

表 6-9：心率传感器需要包含的服务

服务	心率传感器	UUID
Heart Rate Service（心率服务）	M	0x180D
Device Information Service（设备信息服务）	M	0x180A

M：强制要求的。

1. 心率服务

心率服务规格书《第 3 节：服务的特征》中定义了心率服务包含的特征，如下表所示。

表 6-10：心率服务的特征

特征名称	需求	强制属性	可选属性	安全许可
心率测量特征	M	Notify	—	无
心率测量特征 CCCD	M	读，写	—	无
身体测点位置特征	O	读	—	无
心率 Control Point 特征	C.1	写	—	无

M：强制的。 O：可选的

C.1：如果支持能量消耗特性，强制包含，否则，不包含。

注意：

- 安全许可“无”表示此服务不强加任何安全要求，并不表示此服务不能使用安全许可。
- 未列为强制或可选的属性将被排除。

从上表中可以看到，对于心率服务，心率测量特征是必须要包含的，因为心率测量特征的性质是 Notify，所以心率测量特征必须要有 CCCD，用来使能/关闭 Notify。身体测点位置特征是可选择包含的，心率 Control Point 特征是有条件包含的，如果心率传感器支持能量统计，就必须包含该特征，主机可以通过写该特征值复位能量消耗统计。

本例实现的是一个最小需求的心率传感器，没有能量统计特性，所以不包含心率 Control Point 特征，本例的心率传感器包含了一个强制的心率测量特征和一个可选的身体测点位置特征。

本例不考虑心率服务包含的特征的安全许可（特征安全会在后面的配对、绑定章节单独讲解）。

2. 设备信息服务

设备信息服务规格书《第 3 节：服务的特征》中定义了设备信息服务包含的特征，如下表所示。

表 6-11：设备信息服务包含的特征

特征名称	需求	强制属性	可选属性	安全许可
Manufacturer Name String	O	读	—	无
Model Number String	O	读	—	无
Serial Number String	O	读	—	无
Hardware Revision String	O	读	—	无
Firmware Revision String	O	读	—	无
Software Revision String	O	读	—	无
System ID	O	读	—	无
IEEE 11073-20601 Regulatory Certification Data List	O	读	—	无
PnP ID	O	读	—	无

O：可选的

注意：

- 安全许可“无”表示此服务不强加任何安全要求，并不表示此服务不能使用安全许可。
- 未列为强制或可选的属性将被排除。

从上表中可以看到，设备信息服务包含的特征没有强制包含的，都是可选的，本例中设备信息服务只包含一个可选特征：制造商名称字符串特征。

本例不考虑设备信息服务包含的特征的安全许可（特征安全会在后面的配对、绑定章节单独讲解）。

3.2.6. 数据传输

BLE 的数据传输是双向的，蓝牙核心规范中定义了两类数据传输的规程：服务器发起规程和客户端发起规程，通俗地来说就是从机主动向主机发起数据传输和主机向从机发起数据传输。

1. 从机主动发起

从机如果想主动发起数据传输，只能通过两种方式：通知（Notify）和指示（Indicate），从机可以通过这两种方式主动将数据发送给主机，它们的区别如下：

- 通知（Notify）：从机发送通知后，不会关心主机有没有接收到，通知属于不可靠消息。通知需要使用客户端配置描述符（CCCD）配置（CCCD 使能/关闭通知）。
- 指示（Indicate）：从机发送指示后，必须得到主机的应答才能发送下一条指示，指示属于可靠消息。指示同样需要使用客户端配置描述符（CCCD）配置（CCCD 使能/关闭指示）。

2. 主机主动发起

主机通过从机发送的通知或指示可以获取从机的数据，但是数据发送的主动权在从机，主机是“被动接收”，主机无法决定从机什么时候发送数据，同时主机也无法将数据发送给从机。

主机如果想主动发起数据传输，也有两种方式：读和写，通过读和写，主机可以读/写特征值和特征描述符。

- 读（Read）：主机通过“读”可以读取从机的特征值和特征描述符，从而获取从机的数据。
- 写（Write）：主机通过“写”可以写入从机的特征值和特征描述符，从而将数据发送给从机。

综上所述，从机通过通知和指示可以将数据发送给主机，实现从机到主机的数据传输，主机通过读和写既可以读取从机的数据，也可以将数据发送给从机。

到这里，我们就可以描述出心率传感器的具体数据传输，如下图所示。

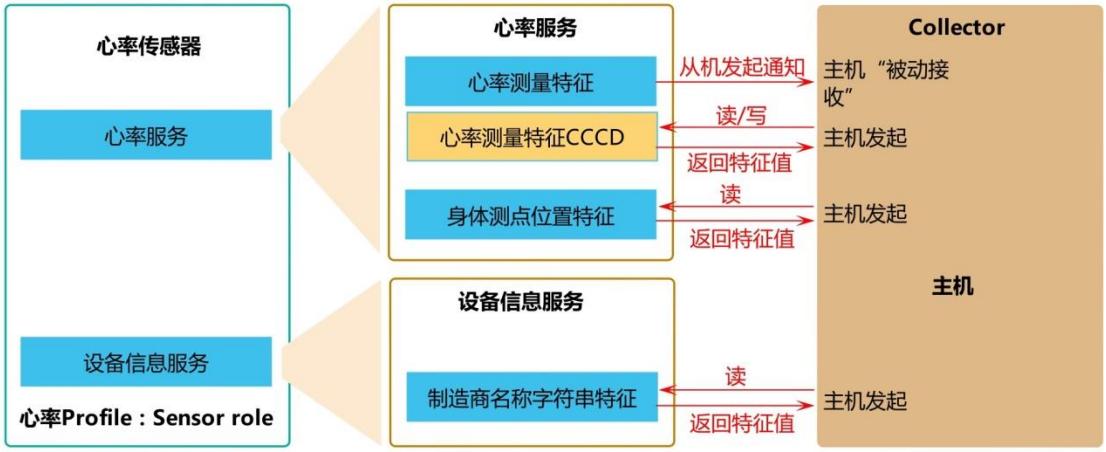


图 6-7：心率传感器数据传输

■ **心率服务的特征**

- 心率测量特征：特征性质是通知，本例中创建一个超时时间 1 秒的周期性 APP 定时器，定时发送心率数据。
- 心率测量特征 CCCD：特征性质是读写，主机可通过写 CCCD 使能/关闭心率测量特征的通知，主机可以通过“读”读取 CCCD 的值。
- 身体测点位置特征：特征性质是读，主机可以通过“读”读取身体测点位置特征值，从而获取身体测点位置（本例中定义的身体测点位置是手腕）。

■ **设备信息服务的特征**

- 制造商名称字符串特征：特征性质设置为“读”，即主机只能通过“读”读取该特征值，获取制造商名称字符串，从机是无法主动发送该特征值的。

清楚了数据传输后，还有一个重要的问题需要明白，那就是特征的数据格式，只有知道了数据格式，我们才能组织数据去发送。

1. 心率服务心率测量特征数据格式

心率服务规格书《第 3.1 节：心率测量》描述了心率特征的数据格式，如下表所示，心率测量特征包含 Flag、心率、能量消耗字段和 RR-Interval 四个字段。

表 6-12：心率测量特征数据格式

Flag 字段	**心率字段**	**能量消耗字段**	**RR-Interval 字段**
UINT8	UINT8 或 UINT16	UINT16	UINT16，可以包含多个
必须包含	必须包含	选择包含	选择包含

1) Flags 字段

心率测量特征中必须包含 Flag 字段，Flag 字段中保留位（位 5~位 7）必须设置为 0，Flag 字段中各个位段的作用如下面所示。

表 6-13：Flags 字段

位段	描述
位 0	心率数据格式位。=0：UINT8；=1：UINT16。
位 1~位 2	传感器接触状态位。该位段指示是否支持传感器接触功能以及是否支持皮肤接触检测功能。 ● 如果服务支持传感器接触功能，位 2 应设置为 1，否则，位 2 设置为 0。 ● 如果支持传感器接触功能，当设备检测到没有接触到皮肤或者接触不良，位 1 应设置为 0，否则，设置为 1。
位 3	能量消耗状态位。指示能量消耗字段是否存在于心率测量特征中。如果存在，该位应设置为 1，否则设置为 0。
位 4	RR-Interval 位。指示 RR-Interval 值是否存在于心率测量特征中。如果一个或者多个 RR-Interval 值存在，该位应设置为 1，否则设置为 0。
位 5~位 7	保留供将来使用，必须设置为 0。

2) 心率测量值字段

心率测量特征中必须包含心率测量值字段，心率测量值指的是心率，即 BPM（Beats Per Minute），心率是指心脏在一分钟跳动的次数，BPM 是指次/分。

心率测量值字段格式可以是 UINT8 或 UINT16（Flag 字段的位 0 指示了心率格式是 UINT8 还是 UINT16）。虽然大多数人类应用的心率测量值不会超过 255 bpm，但特殊应用（例如动物）可能需要支持更高的 bpm 值。如果心率测量值小于或等于 255 bpm，则宜使用 UINT8 格式以节省电量消耗，如果心率测量值超过 255 bpm，则应使用 UINT16 格式。

3) 能量消耗字段

能量消耗字段表示自上次重置以来消耗的累积能量（单位：千焦耳）。如果服务器支持能量消耗计算，则能量消耗字段可以包括在心率测量特征中。如果使用能量消耗，则通常以设定的测试间隔测量 10 次才会在心率测量特征中包含一次能量消耗。

因为能量消耗是 UINT16 格式，可以表示的最高值是 65535 千焦耳。如果能量消耗达到 65535 千焦耳的最大值（0xFFFF），该字段值应保持为 0xFFFF，以便客户端可以知道需要重置能量消耗字段。

4) RR-Interval 字段

如果设备支持 RR-Interval 测量，RR-Interval 字段可以包括在心率测量特征中，如果 RR-Interval 值存在于心率测量特征中，服务器应设置 Flag 字段中的位 4 为 1 并且应在心率测量特征中包含一个或多个 RR-Interval 值，否则，心率测量特征中应不包含 RR-Interval 值并且 Flag 字段中的位 4 应设置为 0。

本例中，设置心率的数据格式为 UINT16、支持传感器接触检测、不支持能量检测、心

率特征中包含 3 个 RR-Interval，因此 Flag 字段的内容如下：

表 6-14：Flag 字段

位 0	位 1	位 2	位 3	位 4	位 5~位 7
1	X	1	0	1	000
心率为 UINT16 格式	接触状态	支持皮肤接触检测	不支持能量消耗	包含 RR-Interval	保留位

表 6-15：心率测量特征数据格式

Flag	心率	第 1 个 RR-Interval	第 2 个 RR-Interval	第 3 个 RR-Interval
1	2 个字节	2 个字节	2 个字节	2 个字节

2. 心率服务身体测点位置特征

本例中身体测点位置特征值使用单个字节表示，传感器测点位置编码如下。注意身体测点位置特征值是静态的，在一个连接中不能改变。

代码清单：传感器测点位置编码

```
//传感器测点位置编码
#define BLE_HRS_BODY_SENSOR_LOCATION_OTHER      0 //其他位置
#define BLE_HRS_BODY_SENSOR_LOCATION_CHEST      1 //胸部
#define BLE_HRS_BODY_SENSOR_LOCATION_WRIST      2 //手腕
#define BLE_HRS_BODY_SENSOR_LOCATION_FINGER     3 //手指
#define BLE_HRS_BODY_SENSOR_LOCATION_HAND       4 //手
#define BLE_HRS_BODY_SENSOR_LOCATION_EAR_LOBE   5 //耳垂
#define BLE_HRS_BODY_SENSOR_LOCATION_FOOT       6 //脚
```

3. 制造商名称字符串特征数据格式

制造商名称字符串特征值是一个代表设备制造商名称的字符串，本例中设置字符串为“IKMSIK”。

3.2.7. 皮肤接触检测状态更新

皮肤接触检测状态即 Flag 字段的位 1 的值，本例中创建一个超时时间为 5 秒的周期性 APP 定时器，定时改变皮肤接触检测状态，从而模拟传感器接触。

4.代码实现

本例在开发指南（下册）配套实验“实验 3-3：BLE 工程模板（外围设备）”的基础上修改。

✧ 注：本节对应的实验源码是：“实验 6-1：实现心率 Profile”。

4.1. 新建文件和路径设置

Nordic 的 SDK 中已经提供了心率传感器的 demo，在 SDK 中的路径是：

“nRF5_SDK_16.0.0_98a08e2\examples\ble_peripheral\ble_app_hrs”，心率 demo 包含了心率和设备信息两个服务，它们在 SDK 中的路径分别是：

- 心率服务：nRF5_SDK_16.0.0_98a08e2\components\ble\ble_services\ble_hrs。
- 设备信息服务：nRF5_SDK_16.0.0_98a08e2\components\ble\ble_services\ble_dis。

本例中，会使用 SDK 中心率服务和设备信息服务的代码，为了区别于心率 demo，我们新建心率服务和设备信息服务的文件。

1. 在工程的 app 文件夹下面新建 ble_hrs 和 ble_dis 两个文件夹，分别存放心率服务和设备信息服务文件。
2. 新建如下表所示的文件，分别保存到 ble_hrs 和 ble_dis 文件夹，并将“my_ble_hrs.c”和“my_ble_dis.c”添加到工程中。

表 6-16：需要新建的文件

项目	文件名
心率服务文件	my_ble_hrs.c
	my_ble_hrs.h
设备信息服务文件	my_ble_dis.c
	my_ble_dis.h

1. 需要引用的头文件

因为在“main.c”文件中需要使用心率和设备信息服务文件，所以“main.c”文件需要引用下面的头文件。

```
#include "my_ble_hrs.h"
#include "my_ble_dis.h"
```

2. 需要添加的头文件包含路径

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

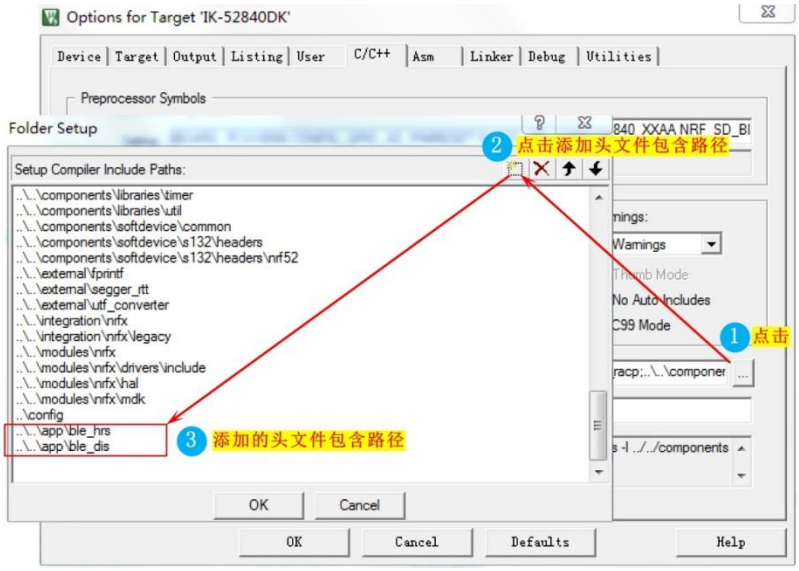


图 6-8：添加头文件包含路径

本例需要添加的头文件路径如下表：

表 6-17： 头文件包含路径

序号	路径
1	..\..\app\ble_hrs
2	..\..\app\ble_dis

4.2. 工程配置

因为我们新建的心率和设备信息服务文件中都加入了宏定义开关执行条件编译，所以需要在“sdk_config.h”文件中加入心率服务和设备信息的宏定义，并勾选心率服务和设备信息服务，如下图所示。

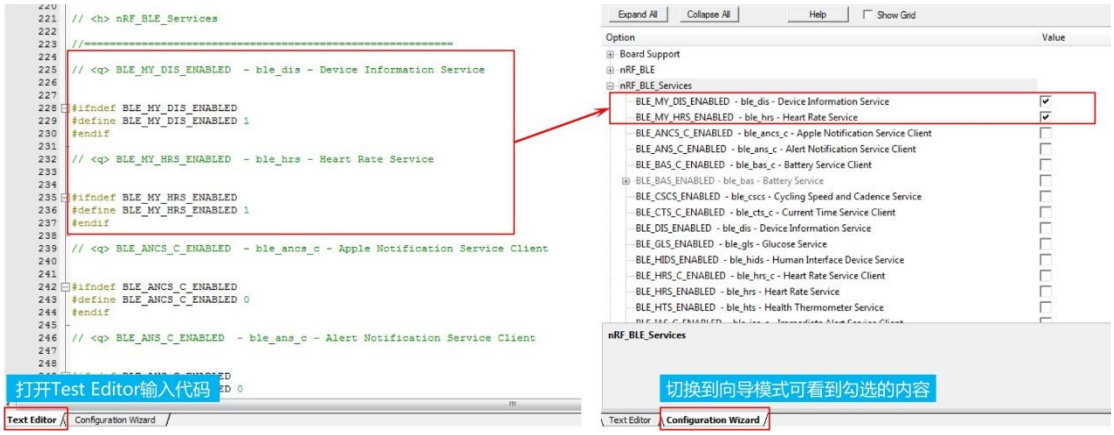


图 6-9：工程配置

4.3. 广播数据和广播参数

广播中包含哪些数据以及广播模式是在广播初始化函数 advertising_init()中完成的，根据前文《需求描述》中对广播数据和广播模式的描述，程序中我们需要加入下面几项。

1. 包含设备名称的全称

注意设备名称是 GAP 服务的一个特征，广播中只能设置是否包含设备名称，是包含全称还是裁剪的设备名称，广播初始化时是无法修改设备名称的。

广播中包含设备名称全称代码清单如下。

代码清单：包含设备名称全称

```
//设备名称类型：全称
init.advdata.name_type               = BLE_ADVDATA_FULL_NAME;
```

2. 包含外观特征

外观特征和设备名称一样，也是 GAP 服务的一个特征，广播中只能设置是否包含外观特征。《需求描述》中定义的外观特征是心率腕带，编码 833。广播中包含设备外观特征代码清单如下。

代码清单：包含外观特征

```
//是否包含外观：包含
init.advdata.include_appearance      = true;
```

3. 包含心率服务的 UUID（0x180D）和设备信息服务的 UUID（0x180A）。

若要在广播中包含服务 UUID，应用程序首先需要定义并初始化 UUID 列表，之后在广播初始化函数中根据需求设置包含全部的 UUID 或是部分 UUID。

代码清单：定义并初始化服务 UUID 列表

```
static ble_uuid_t m_adv_uuids[] =
{
    {BLE_UUID_HEART_RATE_SERVICE,           BLE_UUID_TYPE_BLE},//心率服务 UUID
    {BLE_UUID_DEVICE_INFORMATION_SERVICE,  BLE_UUID_TYPE_BLE} //设备信息服务 UUID
};
```

根据《需求描述》的要求，设置广播中包含全部首要服务的 UUID。

代码清单：广播中包含服务的 UUID

```
//广播中包含服务的 UUID
init.advdata.uuids_complete.uuid_cnt = sizeof(m_adv_uuids) / sizeof(m_adv_uuids[
                                       0]);
init.advdata.uuids_complete.p_uuids  = m_adv_uuids;
```

4. 设置 flags

广播初始化函数中设置 Flags 为一般可发现模式，不支持 BR/EDR。

代码清单：设置 flags

```
//Flag:一般可发现模式，不支持 BR/EDR
init.advdata.flags           = BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE;
```

5. 设置广播参数。

根据《需求描述》中广播参数的描述，定义广播间隔是广播持续时间，代码清单如下。

代码清单：广播间隔和广播超时时间

```
#define APP_ADV_INTERVAL   300  // 广播间隔 (187.5 ms)，单位 0.625 ms
#define APP_ADV_DURATION   0    // 广播持续时间，单位：10ms。设置为 0 表示不超时
```

之后，在广播初始化函数中设置广播参数：广播间隔、广播超时时间和广播模式。

代码清单：设置广播参数

```
//设置广播模式为快速广播
init.config.ble_adv_fast_enabled  = true;
//设置广播间隔和广播持续时间
init.config.ble_adv_fast_interval = APP_ADV_INTERVAL;
init.config.ble_adv_fast_timeout  = APP_ADV_DURATION;
```

以上即是本例广播设置的内容，完整广播初始化函数代码清单如下。

代码清单：广播初始化函数

```
//广播初始化
static void advertising_init(void)
{
    ret_code_t             err_code;
    //定义广播初始化配置结构体变量
    ble_advertising_init_t init;
    //配置之前先清零
    memset(&init, 0, sizeof(init));
    //设备名称类型：全程
    init.advdata.name_type               = BLE_ADVDATA_FULL_NAME;
    //是否包含外观：包含
    init.advdata.include_appearance      = true;
    //Flag:一般可发现模式，不支持 BR/EDR
    init.advdata.flags          = BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE;
    init.advdata.uuids_complete.uuid_cnt = sizeof(m_adv_uuids) / sizeof(m_adv_uuids[0]);
    init.advdata.uuids_complete.p_uuids  = m_adv_uuids;
    //设置广播模式为快速广播
    init.config.ble_adv_fast_enabled  = true;
    //设置广播间隔和广播持续时间
    init.config.ble_adv_fast_interval = APP_ADV_INTERVAL;
    init.config.ble_adv_fast_timeout  = APP_ADV_DURATION;
    //广播事件回调函数
    init.evt_handler = on_adv_evt;
    //初始化广播
    err_code = ble_advertising_init(&m_advertising, &init);
    APP_ERROR_CHECK(err_code);
    //设置广播配置标记。APP_BLE_CONN_CFG_TAG 是用于跟踪广播配置的标记，这是为未来预留的
    //一个参数，在将来的 SoftDevice 版本中，可以使用 sd_ble_gap_adv_set_configure()配
    //置新的广播配置，当前 SoftDevice 版本（S140 V7.0.1 版本）支持的最大广播集数量为 1，
    //因此 APP_BLE_CONN_CFG_TAG 只能写 1。
    ble_advertising_conn_cfg_tag_set(&m_advertising, APP_BLE_CONN_CFG_TAG);
}
```

4.4. 首选连接参数和连接参数协商

连接参数协商的时间和次数定义如下。

- 首次协商延时：5 秒。
- 每次协商之间的间隔：30 秒。
- 最次协商最大尝试次数：3 次。

代码清单：连接参数协商时间和次数

```
// 定义首次调用 sd_ble_gap_conn_param_update()函数更新连接参数延迟时间（5 秒）
#define FIRST_CONN_PARAMS_UPDATE_DELAY  APP_TIMER_TICKS(5000)
// 定义每次调用 sd_ble_gap_conn_param_update()函数更新连接参数的间隔时间（30 秒）
#define NEXT_CONN_PARAMS_UPDATE_DELAY   APP_TIMER_TICKS(30000)
// 定义放弃连接参数协商前尝试连接参数协商的最大次数（3 次）
#define MAX_CONN_PARAMS_UPDATE_COUNT    3
```

连接参数数值和连接参数协商时间和次数的定义会在连接参数协商初始化函数 conn_params_init()中使用，协商初始化函数使用这些参数初始化配置连接参数协商模块，配置成功后，连接参数协商会按照设置的参数自动进行，协商程序模块会将协商结果以事件的方式通知应用程序，应用程序在连接参数协商模块事件处理函数 on_conn_params_evt()中处理协商事件。

4.5. GAP 服务

设备名称、外观和首选连接参数是 GAP 服务的特征，它们是在 GAP 初始化函数 gap_params_init()函数中设置的。

根据《需求描述》中对设备名称的描述，修改设备名称字符串为“HeartSensor”。

代码清单：定义设备名称字符串

```
#define DEVICE_NAME    "HeartSensor"     // 设备名称字符串
```

GAP 初始化函数中设置 GAP 安全模式为无安全性，并写入设备名称，代码清单如下。

代码清单：定义设备名称字符串

```
ble_gap_conn_sec_mode_t sec_mode;
//设置 GAP 的安全模式
BLE_GAP_CONN_SEC_MODE_SET_OPEN(&sec_mode);
//设置 GAP 设备名称
err_code = sd_ble_gap_device_name_set(&sec_mode,
                                      (const uint8_t *)DEVICE_NAME,
                                      strlen(DEVICE_NAME));
```

根据《需求描述》中对外观特征的描述，设置为外观特征为行旅腕带（BLE_APPEARANCE_HEART_RATE_SENSOR_HEART_RATE_BELT）。

代码清单：设置外观特征

```
//设置外观特征:心率腕带
err_code = sd_ble_gap_appearance_set(BLE_APPEARANCE_HEART_RATE_SENSOR_HEART_RA
                                     TE_BELT);
APP_ERROR_CHECK(err_code);
```

根据《需求描述》中对首选连接参数的描述，修改连接参数的数值，如下所示。

- 最小连接间隔：400ms。
- 最大连接间隔：650ms。
- 从机延迟：0。

- 监督超时：4 秒。

代码清单：定义连接参数数值

```
1. #define MIN_CONN_INTERVAL   MSEC_TO_UNITS(400, UNIT_1_25_MS) // 最小连接间隔 (0.4 秒)
2. #define MAX_CONN_INTERVAL   MSEC_TO_UNITS(650, UNIT_1_25_MS) // 最大连接间隔 (0.65 秒)
3. #define SLAVE_LATENCY       0                                // 从机延迟
4. #define CONN_SUP_TIMEOUT    MSEC_TO_UNITS(4000, UNIT_10_MS)  // 监督超时(4 秒)
```

之后，在 GAP 初始化函数中设置首选连接参数，代码清单如下。

代码清单：配置连接参数

```
1. //设置首选连接参数，设置前先清零 gap_conn_params
2. memset(&gap_conn_params, 0, sizeof(gap_conn_params));
3.
4. gap_conn_params.min_conn_interval = MIN_CONN_INTERVAL;//最小连接间隔
5. gap_conn_params.max_conn_interval = MAX_CONN_INTERVAL;//最小连接间隔
6. gap_conn_params.slave_latency     = SLAVE_LATENCY;    //从机延迟
7. gap_conn_params.conn_sup_timeout  = CONN_SUP_TIMEOUT; //监督超时
8. //调用协议栈 API sd_ble_gap_ppcp_set 配置 GAP 参数
9. err_code = sd_ble_gap_ppcp_set(&gap_conn_params);
10.APP_ERROR_CHECK(err_code);
```

4.6. 添加心率服务

下图是加入心率服务的一般流程，首先要根据心率服务规格书和软件的需求声明心率服务结构体和心率服务初始化结构体，初始化结构体仅在心率服务初始化时使用，而心率服务结构体在实例化后会常驻内存，在程序生命期间一直有效。之后，定义心率服务实例，该步骤实例化心率服务，同时注册心率服务的 BLE 事件监视者，接收协议栈事件。

✧ 注：该流程不仅仅适用于添加心率服务，对于添加其它服务具有通用性，即加入其它服务时也可以参照这个流程（对于一些不需主动发送数据的服务，添加的流程可能会省略掉其中的一些步骤，如设备信息服务，会省略掉声明服务结构体和定义服务实例，省略的原因参见添加设备信息服务章节）。

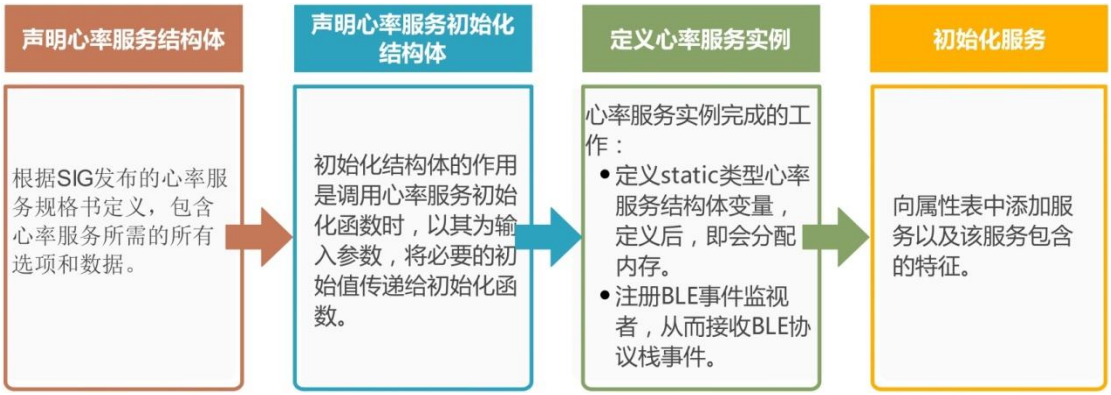


图 6-10：添加心率服务的流程

4.6.1. 声明心率服务结构体

心率服务结构体要包含心率服务所需的所有选项和数据，也就是要包含心率服务规格书中定义的强制包含的、可选择包含的以及条件包含的特征和相关数据，以适应不同的应用场合，心率服务结构体的声明如下。

代码清单：心率服务结构体声明

```
//心率服务结构体，包含心率服务所需的所有选项和数据
struct ble_hrs_s
{
    //心率服务事件回调函数，预留给用户使用，本例中没有使用，初始化时设置为 NULL
    ble_hrs_evt_handler_t        evt_handler;
    //能量消耗支持标志，true=支持能量消耗统计
    bool                         is_expended_energy_supported;
    //传感器接触检测支持标志，true=支持传感器接触检测
    bool                         is_sensor_contact_supported;
    //心率服务句柄（由协议栈提供）
    uint16_t                     service_handle;
    //心率测量特征句柄
    ble_gatts_char_handles_t     hrm_handles;
    //传感器身体测点位置特征句柄
    ble_gatts_char_handles_t     bsl_handles;
    //Heart Rate Control Point 特征句柄
    ble_gatts_char_handles_t     hrcp_handles;
    //连接句柄（由协议栈提供，如未建立连接，该值为 BLE_CONN_HANDLE_INVALID）
    uint16_t                     conn_handle;
    //传感器接触标志，ture=检测到传感器接触
    bool                         is_sensor_contact_detected;
    //自上次心率测试传输后的一组 rr_interval
    uint16_t                     rr_interval[BLE_HRS_MAX_BUFFERED_RR_INTERVALS];
    //自上次心率测试传输后的 rr_interval 数量
    uint16_t                     rr_interval_count;
    //当前一次可发送的最大心率测量数据长度，根据当前 ATT MTU 值调整
```

```
    uint8_t                     max_hrm_len;
};
```

4.6.1.1. 声明心率服务初始化结构体

服务初始化结构体一方面用来初始化服务结构体中需要预先初始化的参数，如设置心率服中的传感器佩戴位置、是否支持传感器接触检测等，另外一方面用来设置服务包含的特征的安全需求，这些参数都是在将服务加入到协议栈属性表之前要确定的。

代码清单：心率服务初始化结构体声明

```
//心率服务初始化结构体，包含初始化心率服务所需的所有选项和数据
typedef struct
{
    //心率服务事件回调函数，预留给用户使用，本例中没有使用，初始化时设置为 NULL
    ble_hrs_evt_handler_t       evt_handler;
    //传感器接触检测支持标志
    bool                        is_sensor_contact_supported;
    //身体测点位置
    uint8_t *                   p_body_sensor_location;
    //写 CCCD（客户端特征描述符）时的安全需求
    security_req_t              hrm_cccd_wr_sec;
    //读 BSL（传感器位置）特征值时的安全需求
    security_req_t              bsl_rd_sec;
} ble_hrs_init_t;
```

4.6.1.2. 定义心率服务实例

应用程序中使用宏定义 BLE_HRS_DEF 实例化一个心率服务，下面的代码实例化一个名称为“m_hrs”的心率服务实例。

代码清单：定义心率服务实例

```
//定义心率服务实例
BLE_HRS_DEF(m_hrs);
```

BLE_HRS_DEF 是一个带参数的宏，输入参数“_name”即为实例化的心率服务的名称，该宏定义展开后代码如下。

代码清单：心率服务实例宏定义

```
#define BLE_HRS_DEF(_name)                                          \
static ble_hrs_t _name;                                             \
NRF_SDH_BLE_OBSERVER(_name ## _obs,                                 \
                     BLE_HRS_BLE_OBSERVER_PRIO,                     \
                     ble_hrs_on_ble_evt, &_name)
```

1. 行 2：定义了 static 类型心率服务结构体变量，为心率服务结构体分配了内存，该语句执行后，心率服务结构体变量常驻内存，即实例化了一个名为“_name”的心率服务，协议栈和应用程序均可通过该名称访问心率服务实例。
2. 行 3~行 5：注册了名称为“_name_obs”的心率服务 BLE 事件监视者，注册成功后，当协议栈有事件产生时，会告知 BLE 事件监视者“_name_obs”，“_name_obs”接收到协议栈的事件后，最终的目的是将事件提交给心率程序模块，使得心率程序模块能够处理这些事件，因此，注册 BLE 事件监视者时需要提供一个事件回调函数（本例提供的事件回调函数是 ble_hrs_on_ble_evt()，当然，读者在编写代码时可以根据自己的习惯和喜好来命名该回调函数），当“_name_obs”接收到协议栈的事件后，即会执行该回调函数，回调函数中即可根据事件的类型进行相应的处理。

■ 事件回调函数格式

注册 BLE 事件监视者时提供的事件回调函数并不是随意写的，它必须遵循一定的格式，否则，当 BLE 事件监视者接收到协议栈的事件后，无法正确的执行回调函数，事件回调函数的格式如下。

代码清单：事件回调函数的格式

```
1. void 函数名称(ble_evt_t const * p_ble_evt, void * p_context)
2. {
3.     //定义一个心率服务结构体指针并指向心率服务结构体
4.     ble_hrs_t * p_hrs = (ble_hrs_t *) p_context;
5.     //根据事件 ID 判断接收的事件类型
6.     switch (p_ble_evt->header.evt_id)
7.     {
8.         //是自己需要处理的事件 1
9.         case 事件 1:
10.            //事件处理代码
11.            break;
12.        //是自己需要处理的事件 2
13.        case 事件 2:
14.            //事件处理代码
15.            break;
16.
17.        //是自己需要处理的事件 n
18.        case 事件 n:
19.            //事件处理代码
20.            break;
21.        default:
22.            break;
23.    }
24.}
```

1. 行 1：事件回调函数返回值类型为“void”，即无返回值。事件回调函数有如下两个形参。

- p_ble_evt：ble_evt_t 结构体指针，该参数用于传递从 BLE 协议栈接收到的事件，通过事件头中的事件 ID 即可分辨事件的类型。事件的类型在 BLE 各个程序模块中定义，如“ble_gap.h”头文件中定义了 GAP 的 BLE 事件类型，“ble_gatt.h”头文件中定义了 GATT 的 BLE 事件类型。

代码清单：BLE 事件结构体 ble_evt_t

```
1. //通用 BLE 事件类型，封装了模块的事件报告
2. typedef struct
3. {
4.    ble_evt_hdr_t header;            //事件头，用来标志事件的类型
5.    //事件联合体，因为联合体共用内存，所以协议栈每次提交的事件只会是一个类型
6.    union
7.    {
8.       ble_common_evt_t  common_evt; //通用事件，事件 ID 前缀 BLE_EVT_
9.       ble_gap_evt_t     gap_evt;    //GAP 事件，事件 ID 前缀 BLE_GAP_EVT_
10.      ble_gattc_evt_t   gattc_evt;  //GATT 客户端事件，事件 ID 前缀 BLE_GATTC_EVT
11.      ble_gatts_evt_t   gatts_evt;  //GATT 服务器事件，事件 ID 前缀 BLE_GATTS_EVT
12.      ble_l2cap_evt_t   l2cap_evt;  //L2CAP 事件，事件 ID 前缀 BLE_L2CAP_EVT
13.   } evt;
14.} ble_evt_t;
```

- p_context：void 类型指针，这里形参使用 void 类型指针是为了可以接收不同类型的数据。

2. 行 4：对于心率服务，p_context 传递的是心率服务结构体数据类型，因此，这里定义了一个心率服务结构体指针变量 p_hrs，并将 p_context 进行强制类型转换后赋值给 p_hrs，之后，函数中即可通过 p_hrs 访问心率服务结构体的各个成员变量。
3. 行 6~行 23：根据 BLE 事件结构体 ble_evt_t 中的事件 ID 判断事件类型，从而处理自己“感兴趣”的事件。

本例中，我们关注协议栈提交的事件中的 3 个事件，关注连接事件和连接断开事件的原因是发送心率特征值时需要通过心率服务连接句柄判断连接是否有效，心率服务连接句柄需要通过这两个事件来更新。关注写事件的原因是为了接收主机写 CCCD 的事件，从而开启或关闭心率测量特征的通知。

- 连接事件：将连接句柄保存到心率服务实例 p_hrs 的“conn_handle”。
- 连接断开事件：设置心率服务实例 p_hrs 中的连接句柄“conn_handle”为无效（BLE_CONN_HANDLE_INVALID）。
- 写事件：接收主机写 CCCD 的事件，从而使能或关闭心率测量特征的通知。

心率服务 BLE 事件监视者的事件回调函数实现的代码如下。

代码清单：心率服务 BLE 事件监视者的事件回调函数

```
1. //心率服务 BLE 事件监视者的事件回调函数
```

```
void ble_hrs_on_ble_evt(ble_evt_t const * p_ble_evt, void * p_context)
{
  //定义一个心率结构体指针并指向心率结构体
  ble_hrs_t * p_hrs = (ble_hrs_t *) p_context;
  //判断事件类型
  switch (p_ble_evt->header.evt_id)
  {
    case BLE_GAP_EVT_CONNECTED://连接事件
      //将连接句柄保存到心率服务实例 p_hrs 的“conn_handle”
      on_connect(p_hrs, p_ble_evt);
      break;
    case BLE_GAP_EVT_DISCONNECTED://连接断开事件
      //设置心率服务实例 p_hrs 中的连接句柄“conn_handle”为 BLE_CONN_HANDLE_INVALID
      on_disconnect(p_hrs, p_ble_evt);
      break;
    case BLE_GATTS_EVT_WRITE://写事件
      //使能或关闭心率测量特征的通知
      on_write(p_hrs, p_ble_evt);
      break;
    default:
      break;
  }
}
```

代码中的 on_connect()、on_disconnect()和 on_write()是实现具体功能的代码，这些函数功能比较简单，容易理解，读者可以自行查看本章的例子中的代码。

4.6.1.3. 初始化心率服务

一般地，初始化服务的代码会集中放在“main.c”文件的 services_init()函数中，应用中有多少个服务，就需要在这个函数中加入多少个对应的服务初始化代码。这样做的好处是便于管理，使得程序的结构更加清晰。对于心率服务，在 services_init()函数中加入的代码如下。

代码清单：初始化心率服务

```
//传感器身体测点位置：手腕，因为心率服务初始化结构体中使用的是指针，因此要先定义一个变
//量，之后将该 变量的地址赋值给心率服务初始化结构体中的 p_body_sensor_location
body_sensor_location = BLE_HRS_BODY_SENSOR_LOCATION_WRIST;
//清零心率初始化结构体
memset(&hrs_init, 0, sizeof(hrs_init));
//心率服务事件回调函数设置为 NULL
hrs_init.evt_handler                  = NULL;
//支持传感器接触检测
hrs_init.is_sensor_contact_supported = true;
//设置传感器佩戴位置
hrs_init.p_body_sensor_location       = &body_sensor_location;
```

```
//设置心率服务的安全等级：无安全性
hrs_init.hrm_cccd_wr_sec = SEC_OPEN;
hrs_init.bsl_rd_sec      = SEC_OPEN;
//初始化心率服务
err_code = ble_hrs_init(&m_hrs, &hrs_init);
APP_ERROR_CHECK(err_code);
```

从上面的代码可以看到，心率服务初始化代码里面通过心率服务初始化结构体设置了需要事先设置的参数，之后调用 ble_hrs_init()函数完成心率服务的初始化。ble_hrs_init()函数代码清单如下。

代码清单：ble_hrs_init()函数

```
//初始化心率服务
 uint32_t ble_hrs_init(ble_hrs_t * p_hrs, const ble_hrs_init_t * p_hrs_init)
{
    uint32_t              err_code;
    //定义 UUID 结构体变量
    ble_uuid_t            ble_uuid;
    //定义特征参数结构体变量
    ble_add_char_params_t add_char_params;
    uint8_t               encoded_initial_hrm[MAX_HRM_LEN];

    //初始化心率服务结构体
    //拷贝心率服务初始化结构体中的心率服务事件句柄
    p_hrs->evt_handler                 = p_hrs_init->evt_handler;
    //拷贝心率服务初始化结构体中的心率服务事件句柄
    p_hrs->is_sensor_contact_supported = p_hrs_init->is_sensor_contact_support
                                         ed;
    //连接句柄初始值设置为：无效的连接句柄
    p_hrs->conn_handle                 = BLE_CONN_HANDLE_INVALID;
    //传感器接触设置为 false，即未接触
    p_hrs->is_sensor_contact_detected  = false;
    //rr_interval 数量初始值设置为 0
    p_hrs->rr_interval_count           = 0;
    //设置当前一次可发送的最大心率测量数据长度
    p_hrs->max_hrm_len                 = MAX_HRM_LEN;

    //心率服务的 UUID
    BLE_UUID_BLE_ASSIGN(ble_uuid, BLE_UUID_HEART_RATE_SERVICE);
    //添加服务到属性表
    err_code = sd_ble_gatts_service_add(BLE_GATTS_SRVC_TYPE_PRIMARY,
                                        &ble_uuid,
                                         &p_hrs->service_handle);
    //检查返回的错误代码
```

```
33.     if (err_code != NRF_SUCCESS)
34.     {
35.         return err_code;
36.     }
37.
38.     //加入心率测量特征
39.     memset(&add_char_params, 0, sizeof(add_char_params));
40.     //心率测量特征的 UUID
41.     add_char_params.uuid              = BLE_UUID_HEART_RATE_MEASUREMENT_CHAR;
42.     //设置心率测量特征值的最大长度
43.     add_char_params.max_len           = MAX_HRM_LEN;
44.     //设置心率测量特征值的初始长度
45.     add_char_params.init_len          = 0;
46.     add_char_params.p_init_value      = encoded_initial_hrm;
47.     //设置心率测量的特征值长度为可变长度
48.     add_char_params.is_var_len        = true;
49.     //设置传感器位置特征的性质：支持通知
50.     add_char_params.char_props.notify = 1;
51.     add_char_params.cccd_write_access = p_hrs_init->hrm_cccd_wr_sec;
52.     //为心率服务添加心率测量特征
53.     err_code = characteristic_add(p_hrs->service_handle, &add_char_params,
54.                                   &(p_hrs->hrm_handles));
55.     //检查返回的错误代码
56.     if (err_code != NRF_SUCCESS)
57.     {
58.         return err_code;
59.     }
60.     //如果心率服务包含传感器位置特征，为心率服务添加传感器位置特征
61.     if (p_hrs_init->p_body_sensor_location != NULL)
62.     {
63.         //初始化参数之前，先清零 add_char_params
64.         memset(&add_char_params, 0, sizeof(add_char_params));
65.         //设置传感器位置特征的 UUID
66.         add_char_params.uuid              = BLE_UUID_BODY_SENSOR_LOCATION_CHAR;
67.         //设置传感器位置特征值的最大长度
68.         add_char_params.max_len           = sizeof(uint8_t);
69.         //设置传感器位置特征值的初始长度
70.         add_char_params.init_len          = sizeof(uint8_t);
71.         //设置传感器位置特征的初始值
72.         add_char_params.p_init_value      = p_hrs_init->p_body_sensor_location;
73.         //设置传感器位置特征的性质：支持读取
74.         add_char_params.char_props.read = 1;
75.         //设置读传感器位置特征值的安全需求
76.         add_char_params.read_access       = p_hrs_init->bsl_rd_sec;
```

```
77.         //为心率服务添加传感器位置特征
78.         err_code = characteristic_add(p_hrs->service_handle, &add_char_params,
79.                                       &(p_hrs->bsl_handles));
80.         //检查返回的错误代码
81.         if (err_code != NRF_SUCCESS)
82.         {
83.             return err_code;
84.         }
85.     }
86.     return NRF_SUCCESS;
87.}
```

ble_hrs_init()函数有下面几个方面需要我们重点掌握，注意，这几个方面对于其它服务是通用的。

1. 服务添加到属性表

服务通过协议栈 API 函数 sd_ble_gatts_service_add()加入到属性表，该函数原型如下。

表 6-18：sd_ble_gatts_service_add()函数

函数原型	uint32_t sd_ble_gatts_service_add(uint8_t type, ble_uuid_t const * p_uuid, uint16_t * p_handle)
函数功能	向属性表中添加服务声明。 注意：次要服务只与引用他们的实体的上下文相关，因此，禁止向属性表中添加未被其他服务引用的次要服务声明。
参　　数	[in] type：首要服务或次要服务。 [in] p_uuid：指向服务 UUID。 [out] p_handle：指向存储分配的 16 位句柄的变量。
返回值	NRF_SUCCESS：添加服务成功。 NRF_ERROR_INVALID_ADDR：提供的指针无效。 NRF_ERROR_INVALID_PARAM：提供的参数无效。 NRF_ERROR_FORBIDDEN：提供了禁用的数据，某些 UUID 是保留给协议栈使用的，应用程序不能使用。 NRF_ERROR_NO_MEM：内存不足，无法完成操作。

调用 sd_ble_gatts_service_add()时需要提供服务的类型和 UUID 这两个参数，见代码清单的行 26~行 31。对于心率服务来说，这两个参数都是确定的：心率服务是首要服务，它的 UUID 是 0x180D。另外对于 UUID，sd_ble_gatts_service_add()只接受 ble_uuid_t 格式，所以需要通过宏 BLE_UUID_BLE_ASSIGN 将心率的 UUID 转换为 ble_uuid_t 格式（代码清单的第 26 行）。

服务加入成功后，协议栈会给服务分配 16 位的句柄，该句柄会通过函数的输出参数保存到心率服务实例“m_hrs”中。

2. 添加特征

添加特征需要在服务添加成功后进行，添加特征时首先定义特征参数结构体变量，之后，根据《需求描述》中对特征的描述设置特征参数结构体成员变量的值，最后调好 characteristic_add()函数完成特征添加。

特征参数结构体 ble_add_char_params_t 包含了 characteristic_add()函数所需的参数，其声明如下。

代码清单：ble_add_char_params_t 结构体声明

```
1.  typedef struct
2.  {
3.      uint16_t                    uuid; //特征 UUID(16 位 UUID)
4.      //UUID 类型，等于 0 表示使用的是标准 UUID，否则，该值在加入 UUID 基数时由
5.      //sd_ble_uuid_vs_add 函数返回
6.      uint8_t                     uuid_type;
7.      uint16_t                    max_len;        //特征值的最大长度
8.      uint16_t                    init_len;       //特征值的初始长度
9.      uint8_t *                   p_init_value;   //特征的初始编码值
10.     bool                        is_var_len;     //指示特征值长度是不是可变长度
11.     ble_gatt_char_props_t       char_props;     //特征性质
12.     ble_gatt_char_ext_props_t char_ext_props; //特征扩展性质
13.     bool                        is_defered_read;  //指示是否支持延迟读操作
14.     bool                        is_defered_write; //指示是否支持延迟写操作
15.     security_req_t              read_access;      //读特征值的安全需求
16.     security_req_t              write_access;     //写特征值的安全需求
17.     security_req_t              cccd_write_access;//写特征的 CCCD 的安全需求
18.     //指示特征存储在应用程序区域还是存储在协议栈
19.     bool                        is_value_user;
20.     ble_add_char_user_desc_t  *p_user_descr;     //如果需要，指向用户描符
21.     ble_gatts_char_pf_t     *p_presentation_format; //如果需要，指向特征表示格式
22. } ble_add_char_params_t;
```

characteristic_add()函数原型如下。

表 6-19：characteristic_add()函数

函数原型	uint32_t characteristic_add (uint16_t service_handle, ble_add_char_params_t * p_char_props, ble_gatts_char_handles_t * p_char_handle)
函数功能	向给定的服务中加入特征。

参　　数	[in] service_handle：服务的句柄。 [in] p_char_props：添加特征所需的信息。 [out] p_char_handle：添加的特征的句柄。
返回值	NRF_SUCCESS：添加特征成功，返回 NRF_SUCCESS，否则返回错误代码。

characteristic_add()函数共有 3 个参数，其中 service_handle 是服务的句柄，服务的句柄是添加服务成功后协议栈分配的，p_char_props 是特征参数结构体，它提供了添加特征时所需的特征参数，p_char_handle 是输出参数，特征添加成功后，协议栈会为特征分配句柄，该句柄会通过函数的输出参数 p_char_handle 保存到心率服务实例“m_hrs”中。

由此可以得出，添加特征时最重要的就是初始化特征参数结构体，为了更直观，我们将添加特征时需要配置的项目总结如下。

- 特征的 UUID。
- 特征的性质。
- 特征的描述符。
- 特征值的长度。
- 特征值的初始值。
- 特征的存储位置。
- 安全性。

接下来，我们看一下，心率服务添加心率测量特征和传感器身体测点位置特征时是如何初始化的。

■ **添加心率测量特征**

根据《需求描述》中对心率测量特征的描述，重点配置项目的配置如下：

- 特征的 UUID：心率测量特征 UUID 是 0x2A37。
- 特征的性质：通知。
- 特征的描述符：因为特征性质是通知，所以必须要有 CCCD。
- 特征值的长度：可变长度。特征值的长度可以设置为可变长度和固定长度，对于可变长度，需要设置最大长度 max_len、初始长度 init_len 和可变长度标志 is_var_len（设置为 true）。对于固定长度，将最大长度 max_len 和初始长度 init_len 设置为一样，可变长度标志 is_var_len 设置为 false 即可。
- 特征值的初始值：initial_hrm 数组，数组长度 9（1 个字节 flag + 2 个字节心率+3 个 RR-Interval 共 6 个字节）。
- 特征的存储位置：存储于应用程序区域。
- 安全性：无安全性。

对应的代码清单如下：

代码清单：添加心率测量特征

```
1.  //配置参数之前先清零 add_char_params
```

```
memset(&add_char_params, 0, sizeof(add_char_params));
//心率测量特征的 UUID
add_char_params.uuid              = BLE_UUID_HEART_RATE_MEASUREMENT_CHAR;
//设置心率测量特征值的最大长度
add_char_params.max_len           = MAX_HRM_LEN;
//设置心率测量特征值的初始长度
add_char_params.init_len          = 0;
add_char_params.p_init_value      = initial_hrm;
//设置心率测量的特征值长度为可变长度
add_char_params.is_var_len        = true;
//设置传感器位置特征的性质：支持通知
add_char_params.char_props.notify = 1;
add_char_params.cccd_write_access = p_hrs_init->hrm_cccd_wr_sec;
//为心率服务添加心率测量特征
err_code = characteristic_add(p_hrs->service_handle, &add_char_params,
                              &(p_hrs->hrm_handles));
//检查返回的错误代码
if (err_code != NRF_SUCCESS)
{
   return err_code;
}
```

■ **添加身体测点特征**

根据《需求描述》中对传感器身体测点特征的描述，重点配置项目的配置如下：

- 特征的 UUID：传感器身体测点特征的 UUID 是 0x2A38。
- 特征的性质：读。
- 特征的描述符：无。
- 特征值的长度设置：固定长度，一个字节。
- 特征值的初始值：传感器身体测点为手腕。
- 特征的存储位置：存储于应用程序区域。
- 安全性：无安全性。

对应的代码清单如下：

代码清单：添加传感器身体测点特征

```
//初始化参数之前，先清零 add_char_params
memset(&add_char_params, 0, sizeof(add_char_params));
//设置传感器位置特征的 UUID
add_char_params.uuid              = BLE_UUID_BODY_SENSOR_LOCATION_CHAR;
//设置传感器位置特征值的最大长度
add_char_params.max_len           = sizeof(uint8_t);
//设置传感器位置特征值的初始长度
add_char_params.init_len          = sizeof(uint8_t);
//设置传感器位置特征的初始值
```

```
add_char_params.p_init_value    = p_hrs_init->p_body_sensor_location;
//设置传感器位置特征的性质：支持读取
add_char_params.char_props.read = 1;
//设置读传感器位置特征值的安全需求
add_char_params.read_access     = p_hrs_init->bsl_rd_sec;
//为心率服务添加传感器位置特征
err_code = characteristic_add(p_hrs->service_handle, &add_char_params,
                              &(p_hrs->bsl_handles));
//检查返回的错误代码
if (err_code != NRF_SUCCESS)
{
    return err_code;
}
```

4.7. 添加设备信息服务

从《需求描述》中对心率服务和设备信息服务的描述可以看到，心率服务包含心率测量和传感器身体测点特征，它既需要主动发送心率数据给主机，又可以接受主机的“读”，所以，心率服务需要关心有没有和主机建立连接、通知有没有使能等问题。而设备信息服务只包含了一个特征：制造商名称字符串特征，该特征是只读的，而且它的值初始化后就不会改变。主机读取制造商名称字符串特征值时，必定和心率传感器建立了连接并且发现服务和特征完成，因此它不用关心是否建立连接之类的问题，也就是说，它不需要去接收协议栈的事件，所以，设备信息服务不需要注册事件监视者。

加入设备信息服务的流程如下图所示，注意和加入心率服务的区别。

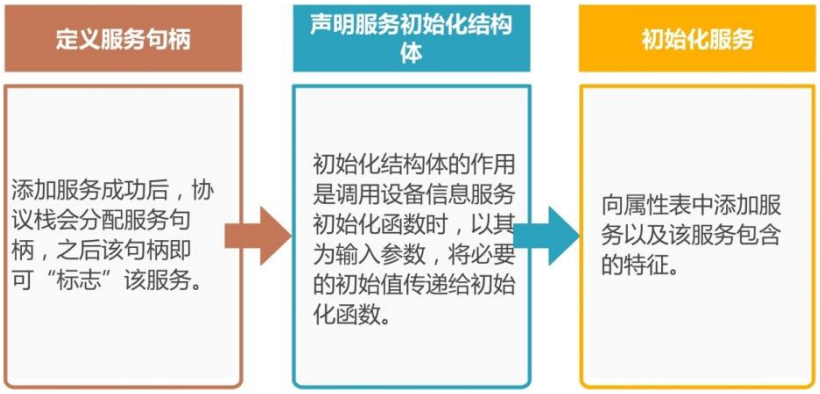


图 6-11：添加设备信息服务的流程

4.7.1. 定义服务句柄变量

设备信息服务只包含了一个只读的特征，对于这种简单的服务，不需要专门定义一个结构体来标志服务，只需要定义一个服务句柄变量来标志服务即可。

代码清单：定义服务句柄变量

```
//定义 uint16 类型变量，用来保存服务句柄。添加服务成功后，协议栈会分配句柄保存到该变量，
//之后该句柄即可用来标志设备信息服务
```

```
static uint16_t                 service_handle;
```

4.7.2. 声明设备信息服务初始化结构体

设备信息服务初始化结构体包含初始化设备信息服务所需的所有可能的特征，即包含了设备信息服务规格书中描述的所有可以包含的特征，对于本例，只用到了制造商名称字符串特征。

代码清单：设备信息服务初始化结构体

```
//设备信息服务初始化结构体，包含初始化设备信息服务所需的所有可能的特征
typedef struct
{
    ble_srv_utf8_str_t       manufact_name_str;          //制造商名称字符串
    ble_srv_utf8_str_t       model_num_str;              //型号字符串
    ble_srv_utf8_str_t       serial_num_str;             //序列号字符串
    ble_srv_utf8_str_t       hw_rev_str;                 //硬件版本字符串
    ble_srv_utf8_str_t       fw_rev_str;                 //固件版本字符串
    ble_srv_utf8_str_t       sw_rev_str;                 //软件版本字符串
    ble_dis_sys_id_t *       p_sys_id;                   //System ID
    //IEEE 11073-20601 监管认证列表
    ble_dis_reg_cert_data_list_t * p_reg_cert_data_list;
    ble_dis_pnp_id_t *             p_pnp_id;          //PnP ID
    security_req_t                 dis_char_rd_sec;   //读 DIS 特征值时的安全需求
} ble_dis_init_t;
```

4.7.3. 初始化设备信息服务

初始化设备信息服务和初始化心率服务类似，首先在 services_init()函数中加入设备信息服务初始化代码。

代码清单：初始化设备信息服务

```
//定义设备信息服务初始化结构体
ble_dis_init_t     dis_init;
//清零设备信息初始化结构体
memset(&dis_init, 0, sizeof(dis_init));
ble_srv_ascii_to_utf8(&dis_init.manufact_name_str, (char *)MANUFACTURER_NAME);
//设置设备信息服务的安全等级：无安全性
dis_init.dis_char_rd_sec = SEC_OPEN;
//初始化设备信息服务
err_code = ble_dis_init(&dis_init);
APP_ERROR_CHECK(err_code);
```

其中，MANUFACTURER_NAME 是我们定义的制造商名称字符串，该字符串根据公司信息修改。

代码清单：定义设备名称字符串

```
1.  #define MANUFACTURER_NAME        "IKMSIK"            // 制造商名称字符串
```

services_init()函数中设置好了需要事先设置的参数，之后调用 ble_dis_init ()函数完成设备信息服务的初始化。services_init()函数代码清单如下。

代码清单：设备信息服务初始化函数

```
1.  //初始化设备信息服务
2.  uint32_t ble_dis_init(ble_dis_init_t const * p_dis_init)
3.  {
4.      uint32_t   err_code;
5.      //定义 UUID 结构体变量
6.      ble_uuid_t ble_uuid;
7.
8.      //设备信息服务的 UUID
9.      BLE_UUID_BLE_ASSIGN(ble_uuid, BLE_UUID_DEVICE_INFORMATION_SERVICE);
10.     //添加服务到属性表
11.     err_code = sd_ble_gatts_service_add(BLE_GATTS_SRVC_TYPE_PRIMARY, &ble_uuid,
12.                                         &service_handle);
13.     //检查返回的错误代码
14.     if (err_code != NRF_SUCCESS)
15.     {
16.         return err_code;
17.     }
18.
19.     //如果制造商名称字符串长度大于 0，表示字符串有效
20.     if (p_dis_init->manufact_name_str.length > 0)
21.     {
22.         //添加制造商名称字符串特征
23.         err_code = char_add(BLE_UUID_MANUFACTURER_NAME_STRING_CHAR,
24.                             p_dis_init->manufact_name_str.p_str,
25.                             p_dis_init->manufact_name_str.length,
26.                             p_dis_init->dis_char_rd_sec,
27.                             &manufact_name_handles);
28.         //检查返回的错误代码
29.         if (err_code != NRF_SUCCESS)
30.         {
31.             return err_code;
32.         }
33.     }
34.     return NRF_SUCCESS;
35. }
```

1. 服务添加到属性表

服务通过协议栈 API 函数 sd_ble_gatts_service_add()加入到属性表，服务加入成功后，协议栈会给服务分配 16 位的句柄，该句柄会通过函数的输出参数保存到之前定义的“service_handle”变量中，之后，设备信息服务句柄即可用来标志设备信息服务。

2. 添加特征

添加特征的过程和心率服务一样，区别是各个参数的配置。根据《需求描述》中对制造商名称特征的描述，重点配置项目的配置如下：

- 特征的 UUID：制造商名称特征 UUID 是 0x2A29。
- 特征的性质：读。
- 特征的描述符：无。
- 特征值的长度：固定长度。定义的制造商名称字符串的长度，本例中定义的字符串是“IKMSIK”，长度为 6。
- 特征值的初始值：指向定义的制造商名称字符串。
- 特征的存储位置：存储于应用程序区域。
- 安全性：无安全性。

对应的代码清单如下：

代码清单：定义连接参数数值

```
//定义特征参数结构体变量
ble_add_char_params_t add_char_params;
//检查指向制造商名称字符串的指针是否有效
APP_ERROR_CHECK_BOOL(p_char_value != NULL);
//检查制造商名称字符是否大于 0
APP_ERROR_CHECK_BOOL(char_len > 0);
//初始化参数之前，先清零 add_char_params
memset(&add_char_params, 0, sizeof(add_char_params));
//制造商名称字符串特征的 UUID
add_char_params.uuid            = uuid;
//设置制造商名称字符串特征值的最大长度
 add_char_params.max_len         = char_len;
//设置制造商名称字符串特征值的初始长度
 add_char_params.init_len        = char_len;
//设置制造商名称字符串特征的初始值
add_char_params.p_init_value    = p_char_value;
//设置制造商名称字符串特征的性质：支持读取
add_char_params.char_props.read = 1;
add_char_params.read_access     = rd_sec;
//向设备信息服务中添加制造商名称字符串特征
return characteristic_add(service_handle, &add_char_params, p_handles);
```

5. 发送数据和传感器接触状态更新

5.1. 发送心率测量特征通知

蓝牙的数据发送是必须要掌握的内容，注意蓝牙只有发送函数，没有接收函数，蓝牙数据的接收都是在协议栈完成的，协议栈接收到数据后，会向应用程序或各个程序模块提交事件，应用程序或其它程序模块根据事件结构体中提供的信息获取接收的数据。

根据《需求描述》中对数据传输部分的描述，我们需要创建一个超时事件为 1 秒的周期性 APP 定时器，定时发送心率测量特征值通知（创建 APP 定时器参见第五章，此处不再赘述）。

心率测量定时器超时事件回调函数中加入下面的代码，用于发送心率测量特征通知。

代码清单：心率测量定时器超时事件回调函数

```
//心率测量定时器事件回调函数
static void heart_rate_timeout_handler(void * p_context)
{
    ret_code_t      err_code;
    uint16_t        heart_rate;

    UNUSED_PARAMETER(p_context);
    //心率值写死为：270bpm
    heart_rate = 270;
    //发送心率测量值
    err_code = ble_hrs_heart_rate_measurement_send(&m_hrs, heart_rate);
    if ((err_code != NRF_SUCCESS) &&
        (err_code != NRF_ERROR_INVALID_STATE) &&
        (err_code != NRF_ERROR_RESOURCES) &&
        (err_code != NRF_ERROR_BUSY) &&
        (err_code != BLE_ERROR_GATTS_SYS_ATTR_MISSING)
       )
    {
        APP_ERROR_HANDLER(err_code);
     }
}
```

从上面的代码可以看到，设置了心率值后，调用 ble_hrs_heart_rate_measurement_send() 函数完成通知的发送，该函数代码如下。

代码清单：ble_hrs_heart_rate_measurement_send()函数

```
//发送心率测量特征通知，即心率数据
//注意：这里发送的心率数据并不是真实的、从心率传感器采样的数据，而是直接发送固定的数据
uint32_t ble_hrs_heart_rate_measurement_send(ble_hrs_t * p_hrs, uint16_t
                                             heart_rate)
```

```
5.  {
6.      static uint8_t rr_val = 0;
7.      uint32_t err_code;
8.      //flag 初始值设置为 0x15，即设置心率的数据格式为 UINT16、支持传感器接触检测、
9.      //不支持能量检测、心率特征中包含 RR-Interval
10.     uint8_t flags = 0x15;
11.
12.     //如果连接有效，发送心率测量特征通知
13.     if (p_hrs->conn_handle != BLE_CONN_HANDLE_INVALID)
14.     {
15.         uint8_t                hrm_data[1+2+6];
16.         uint16_t               hvx_len;
17.         //定义 GATT HVx 参数结构体
18.         ble_gatts_hvx_params_t hvx_params;
19.         //写入心率测量特征的数值
20.         if (p_hrs->is_sensor_contact_detected)
21.         {
22.             //更新传感器接触检测状态
23.             flags |= HRM_FLAG_MASK_SENSOR_CONTACT_DETECTED;
24.         }
25.         hrm_data[0] = flags;
26.         hrm_data[1] = heart_rate;
27.         hrm_data[2] = heart_rate>>8;
28.         hrm_data[3] = rr_val+10;
29.         hrm_data[4] = 0x01;
30.         hrm_data[5] = rr_val+5;
31.         hrm_data[6] = 0x01;
32.         hrm_data[7] = rr_val;
33.         hrm_data[8] = 0x01;
34.         //为了方便观察数据，每次发送后，数据加 1
35.         rr_val++;
36.         //特征值长度为 9
37.         hvx_len = 1+2+6;
38.
39.         //设置之前先清零
40.         memset(&hvx_params, 0, sizeof(hvx_params));
41.         //心率测量特征值句柄
42.         hvx_params.handle = p_hrs->hrm_handles.value_handle;
43.         //类型为通知
44.         hvx_params.type   = BLE_GATT_HVX_NOTIFICATION;
45.         hvx_params.offset = 0;
46.         //发送的数据长度
47.         hvx_params.p_len  = &hvx_len;
48.         //发送的数据
```

```
        hvx_params.p_data = hrm_data;
        //发送心率测量特征通知
        err_code = sd_ble_gatts_hvx(p_hrs->conn_handle, &hvx_params);
    }
    else//返回连接状态无效的错误代码
    {
        err_code = NRF_ERROR_INVALID_STATE;
    }
    return err_code;
}
```

ble_hrs_heart_rate_measurement_send()函数中通过心率服务的连接句柄判断连接是否建立，只有连接句柄有效，该函数才会执行后续的发送操作。

函数中定义了 GATT HVx 参数结构体变量 hvx_params，hvx_params 包含了发送所需要的参数，函数中在判断连接句柄有效后，接着设置 hvx_params 各个成员变量的值，最终调用协议栈 API 函数 sd_ble_gatts_hvx()将数据发送出去，sd_ble_gatts_hvx()函数原型如下。

表 6-20：sd_ble_gatts_hvx()函数

函数原型	uint32_t sd_ble_gatts_hvx (uint16_t conn_handle, ble_gatts_hvx_params_t const * p_hvx_params)
函数功能	通知或指示一个特征值。 该 API 函数会检查特征的 CCCD，以验证通知或指示已被客户端使能。它还能够在发出 PDU 之前更新属性值，以便应用程序可以通过调用单个 API 原子性地执行数据更新和服务器启动的事务。 产生的事件： BLE_GATTS_EVT_HVN_TX_COMPLETE：通知发送完成。 BLE_GATTS_EVT_HVC：接收到对端设备返回的确认信息，这个事件是针对指示的。
参　　数	[in] conn_handle：连接句柄。 [in,out] p_hvx_params：指向 HVx 参数结构体。如果 ble_gatts_hvx_params_t 的 p_data 包含非空指针，在发送通知或指示之前，属性值将使用其指向的内容进行更新。如果属性值已经更新，ble_gatts_hvx_params_t 的 p_len 会被 SoftDevice 更新，以包含写入的实际字节数，否则将设置为 0。
返回值	NRF_SUCCESS：成功将通知或指示加入队列以待传输，并可选择更新属性值。 BLE_ERROR_INVALID_CONN_HANDLE：连接句柄无效。 NRF_ERROR_INVALID_STATE：出现了以下的一种或多种情况。

	• 连接状态无效。 • 通知和/或指示没有使能。 • 正在进行 ATT_MTU 交换。 NRF_ERROR_INVALID_ADDR：提供了无效的指针。 NRF_ERROR_INVALID_PARAM：提供了无效的参数。 BLE_ERROR_INVALID_ATTR_HANDLE：提供了无效的属性句柄。只有应用程序直接添加的属性才可用于通知和指示。 BLE_ERROR_GATTS_INVALID_ATTR_TYPE：提供了无效的属性类型，只能通知和指示特征值。 NRF_ERROR_NOT_FOUND：找不到属性。 NRF_ERROR_FORBIDDEN：连接的当前安全级别低于与此特性相关的 CCCD 的写权限所需的安全级别。 NRF_ERROR_DATA_SIZE：提供了的数据大小无效。 NRF_ERROR_BUSY：BLE_GATT_HVX_INDICATION 规程已经在执行中。等待 BLE_GATTS_EVT_HVC 事件并重试。 BLE_ERROR_GATTS_SYS_ATTR_MISSING：系统属性缺失。使用 sd_ble_gatts_sys_attr_set()函数将它们设置为已知值。 NRF_ERROR_RESOURCES：队列中排队的通知过多，等待 BLE_GATTS_EVT_HVN_TX_COMPLETE 事件并重试。 NRF_ERROR_TIMEOUT：已经有 GATT 规程超时，如不重新建立连接，则无法执行新的 GATT 规程。

上面发送数据的代码看起来很多，但是大多数的代码都是准备需要的发送数据，实际发送数据的流程并不复杂，发送数据主要包含以下 4 个步骤。

1. **判断是否具备发送条件：**这里至少要判断服务的连接句柄是否有效，只有连接句柄有效，才可以发送数据，否则发送必然出错。若服务连接句柄有效，即可作为 sd_ble_gatts_hvx()函数的第 1 个参数传递给函数。
2. **定义 ble_gatts_hvx_params_t 类型变量并初始化：**ble_gatts_hvx_params_t 结构体的声明如下，包含 5 个成员变量，发送数据之前，必须初始化这些变量，之后以此为 sd_ble_gatts_hvx()函数的第 2 个参数传递给函数。

代码清单：ble_gatts_hvx_params_t 结构体声明

```
//GATT HVx 参数结构体
typedef struct
{
  uint16_t          handle;       //特征值句柄
  uint8_t           type;         //类型：通知或指示
  uint16_t          offset;       //属性值内的偏移量
  uint16_t         *p_len;        //待写入的数据字节数，返回写入的数据字节数
```

```
8.   uint8_t const    *p_data;        //实际的数据内容,如为 NULL 则使用当前的属性值
9. } ble_gatts_hvx_params_t;
```

- Handle：特征值句柄，这是添加特征成功后，由协议栈分配的。因此我们只需将特征值句柄赋值给它即可。
- Type：类型，用来指出发送的是通知还是指示，发送通知时设置为 BLE_GATT_HVX_NOTIFICATION，发送指示时设置为 BLE_GATT_HVX_INDICATION。
- *p_len：作为输入参数的时候，是指待写入到发送队列的数据字节数，作为输出参数时指的是已经成功写入的字节数。
  - p_len 是指针变量，所以不能直接将发送长度数值赋值给它，需要先定义一个变量，将该变量的值设置为发送的数据长度，之后该变量的地址赋值给 p_len。
  - 如果特征值长度在添加特征时设置的是可变长度，那么“*p_len”不超过最大长度即可。
  - 如果特征值长度在添加特征时设置的是固定长度，那么“*p_len”的值必须和设置的固定长度的值一样。
- *p_data：指向实际发送的数据。
  - p_data 是指针变量，赋值时只能接收地址，所以需要定义一个数组，之后数组的地址赋值给 p_len。
  - p_len 是 uint8_t 类型的，所以存储发送数据的数组需要定义为 uint8_t 类型或者转换为 uint8_t 类型。
- Offset：属性值内的偏移量。Offset 的作用如下图所示，Offset 可以设置发送数据的起始位置，这让发送数据更加灵活。

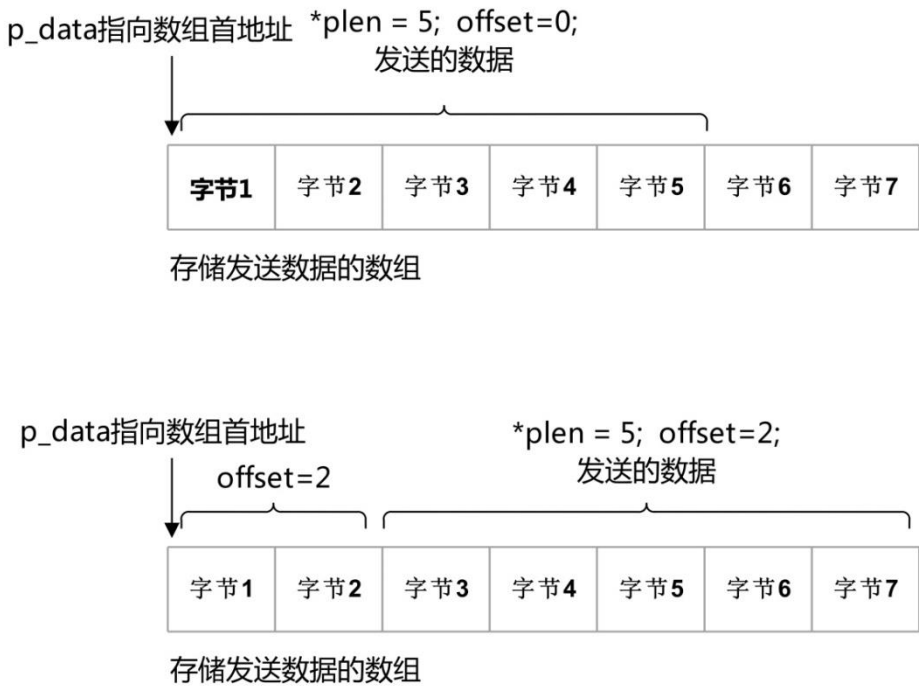


图 6-12：Offset 的作用

3. **以服务连接句柄和作为函数的参数调用 sd_ble_gatts_hvx()函数发送数据。**

4. **检查 sd_ble_gatts_hvx()函数的返回值，判断是否成功并执行相应的处理。**

5.2. 传感器接触状态更新

根据《需求描述》中对数据传输部分的描述，创建一个超时事件为 5 秒的周期性 APP 定时器，定时取反传感器接触状态，也就是取反心率服务实例 m_hrs 的 is_sensor_contact_detected 变量的值。

代码清单：传感器接触状态更新定时器事件回调函数

```
//传感器接触状态更新定时器事件回调函数
static void sensor_contact_update_timeout_handler(void * p_context)
{
    static bool sensor_contact_detected = false;
    UNUSED_PARAMETER(p_context);
    //取反传感器接触状态
    sensor_contact_detected = !sensor_contact_detected;
    //设置传感器接触状态
    ble_hrs_sensor_contact_detected_update(&m_hrs, sensor_contact_detected);
}
```

代码清单：更新传感器接触检测位

```
//更新传感器接触检测位
void ble_hrs_sensor_contact_detected_update(ble_hrs_t * p_hrs, bool is_sensor_contact_detected)
{
    p_hrs->is_sensor_contact_detected = is_sensor_contact_detected;
}
```

6. 下载验证

1. 解压“…\4：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 6-1：实现心率 Profile”，将解压后得到的文件夹“ble_app_hrs”拷贝合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_hrs\project\mdk5”目录下的工程“ble_app_hrs.uvproj”。
4. 切换到 Target“flash_s140_nrf52_7.0.1_softdevice”，点击下载协议栈。
5. 切换到 Target“IK-52840DK”，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qfaa.hex”位于工程目录下的“Objects”文件夹中。
6. 程序运行后，指示灯 D1 闪烁表示正在广播。
7. 手机上打开 nRF Connect，点击“SCAN”按钮启动扫描，扫描到设备“HeartSensor”后点击展开，如下图所示。

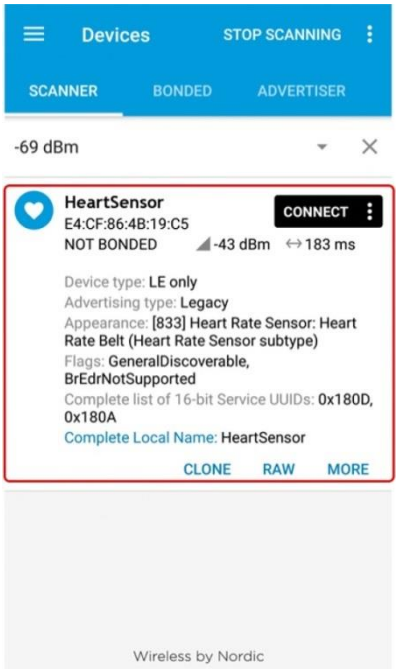


图 6-13：广播数据

从上图中可以看到，扫描到的广播数据和我们设置的是一致的。

- 广播中包含了设备名称“HeartSensor”。
- 广播中包含了外观：心率腕带，编码 833。
- Flags：一般可发现模式，不支持 BR/EDR。
- 广播中包含了完整的 UUID 列表：0x180D（心率服务 UUID），0x180A（设备信息服务 UUID）。

8. 点击“CONNECT”按钮连接设备，连接成功后，可以看到心率服务和设备信息服务，如下图所示。

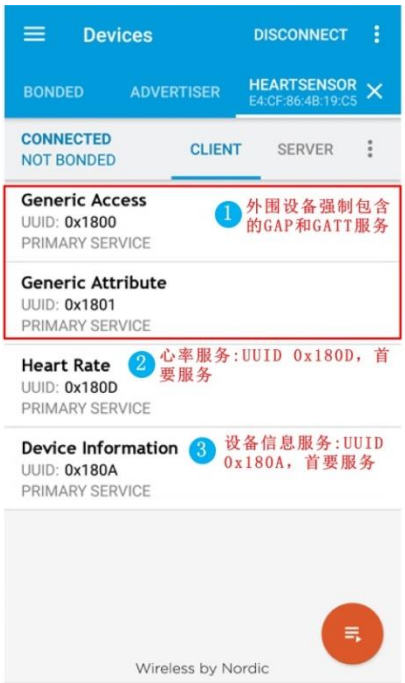


图 6-14：连接成功

9. 点击心率服务“Gereric Access”展开服务，如下图所示，读取设备名称、外观和首选连接参数特征，可以看到读出的数值和代码里面设置的是一致的。

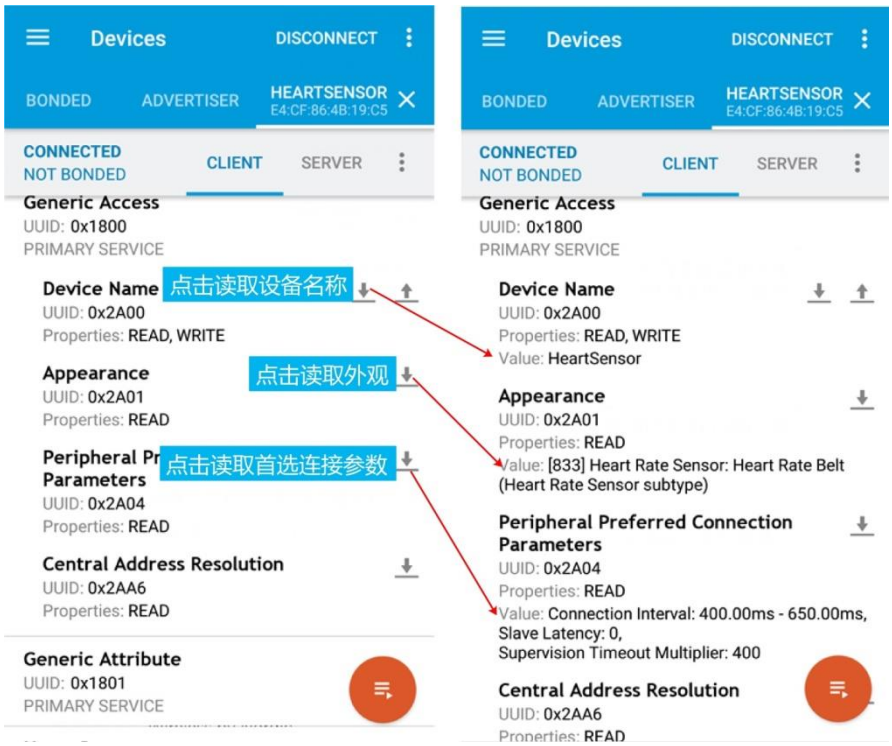


图 6-15：GAP 服务

10. 点击心率服务“Heart Rate”展开服务，如下图所示。使能通知后，即可看到心率传感器发送的心率数据，心率数据每秒更新一次，传感器接触状态每 5 秒改变一次（Contact is NOT Detected 或 Contact is Detected）。

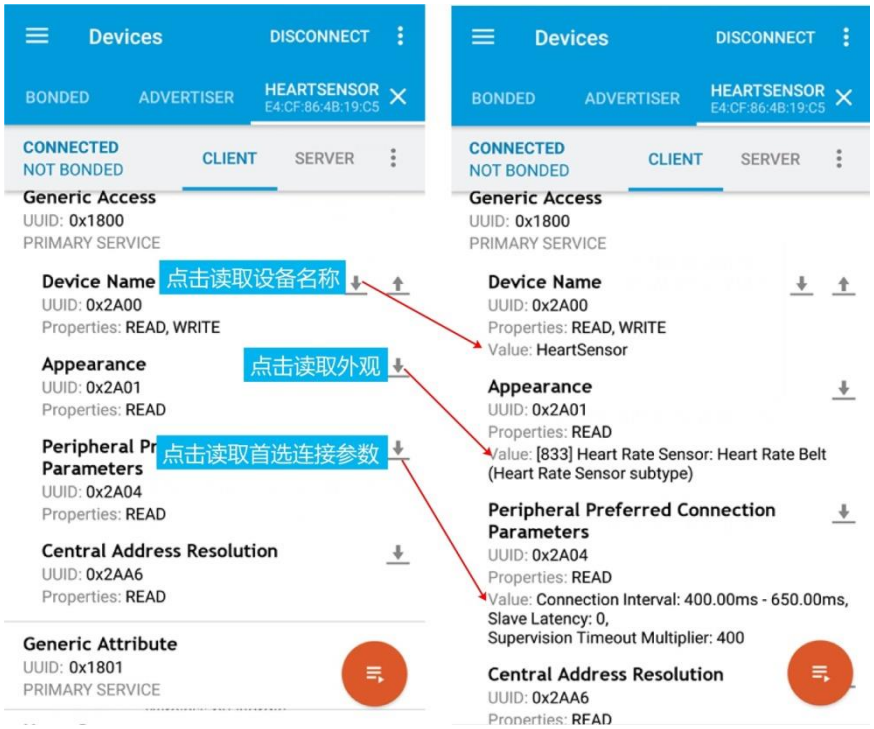


图 6-16：心率服务

上图中看到的心率测量特征值是经过 nRF Connect 解析过的，如果我们希望看到原始数据，按照下图所示操作。

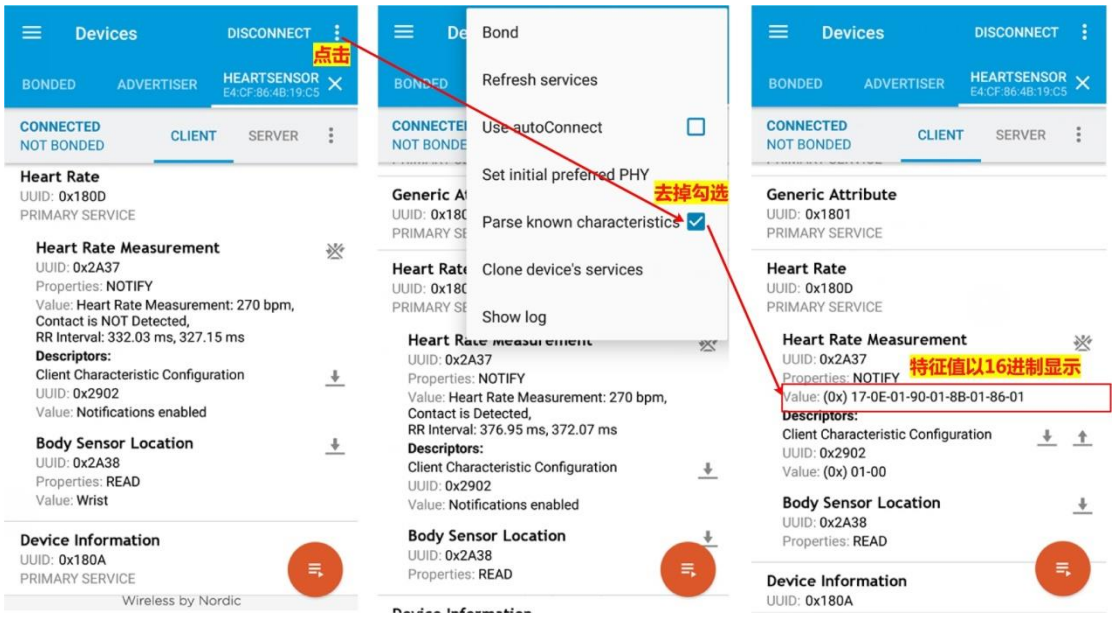


图 6-17：心率特征值以 16 进制显示

11. 点击设备信息服务“Device Information”展开服务，读取制造商名称特质，可以看到读出的特征值和代码里面设置的是一致的。

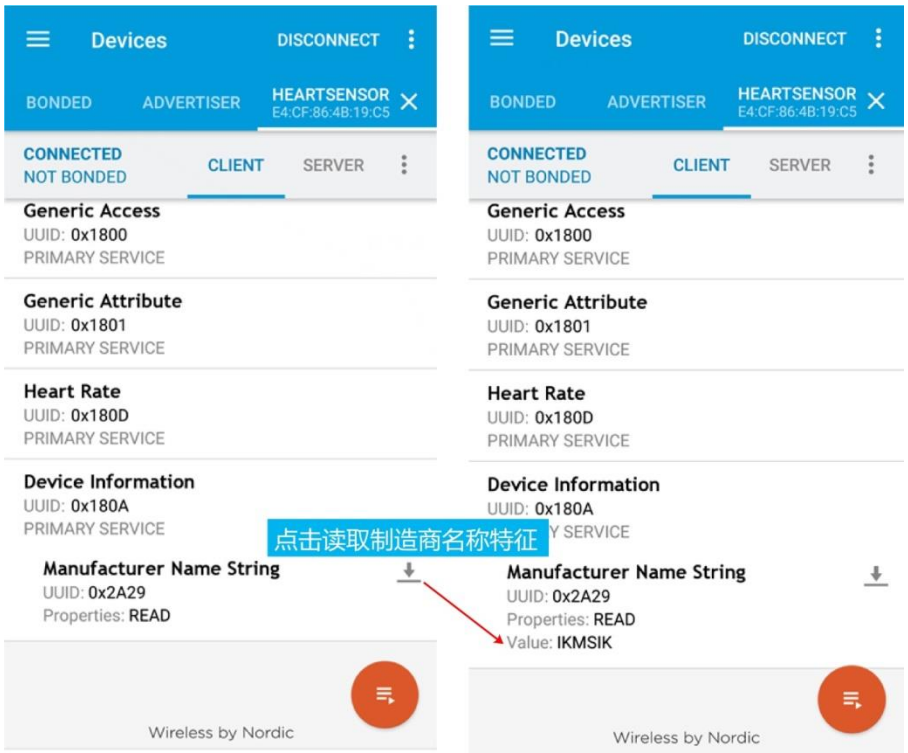


图 6-18：设备信息服务

第七章：实现一个自定义 profile

1. 学习目的

本章实现自定义 Profile：串口透传的基本功能（传输最大长度 20 个字节），阅读过程中，要注意和《第六章：实现一个标准的 Profile》对比，同时应重点关注以下问题。

- 如何生成和使用自定义 UUID。
- 广播中如何增加扫描响应？为什么要增加扫描响应？
- 自定义服务的初始化和标准服务有哪些不同的地方？（自定义服务需要向 Softdevice 写入 128 位的 UUID 基数）。
- 结合第六章的心率服务实例和本章串口透传服务的实例，加深对服务实例的理解。
- 掌握如何实现长包传输（单次传输最大长度 247 个字节，有效载荷 244 个字节）。

2. 需求描述

2.1. 功能描述

蓝牙 BLE 串口透传实现的功能如下图所示，串口透传 Profile 包含一个串口透传服务，串口透传服务包含 TX（特征性质：Notify）和 RX（特征性质：Write）两个特征，分别用于向主机发送数据和接收主机发过来的数据。

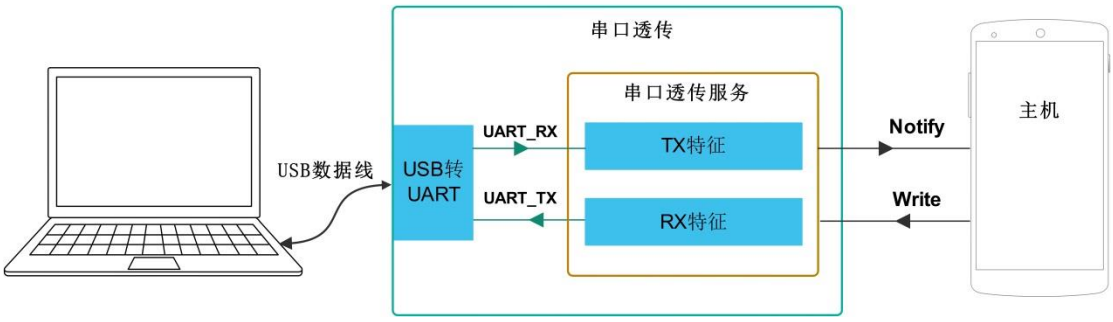


图 7-1：串口透传示意图

数据传输方向以透传设备为中心，数据“出”透传设备，称为 TX，数据“入”透传设备，称为 RX。

- 串口接收的数据发送给主机：透传设备接收到计算机通过串口发送过来的数据后，数据写入属性表中 TX 的特征值，并将 TX 特征值通知给主机，从而实现串口接收的数据发送给主机。
- 接收主机发送的数据通过串口输出：主机通过写 RX 特征值将数据传送给透传设备，透传设备在写事件中获取数据，之后将接收的数据通过串口输出给计算机。

2.2. 生成自定义 UUID

自定义 UUID 由制造商自己定义，这就不可避免地存在 UUID 重复的可能，对于自定义 UUID，没有任何机制可以保证 UUID 的唯一性。

Bluetooth SIG 推荐使用下面链接的在线工具生成自定义 UUID，生成的时候使用版本 4，此站点生成的版本 4 UUID 是使用安全随机数生成器生成的，这使得生成的 UUID 出现重复的可能变得极小。

网址：https://www.uuidgenerator.net/

打开网址后，即可看到生成的自定义 UUID，如下图所示，如果要批量生成自定义 UUID，在“Bulk Version 4 UUID Generation”下面的“How Many?”中输入要生成的 UUID 的数量，然后点击“Generate”按钮即可。

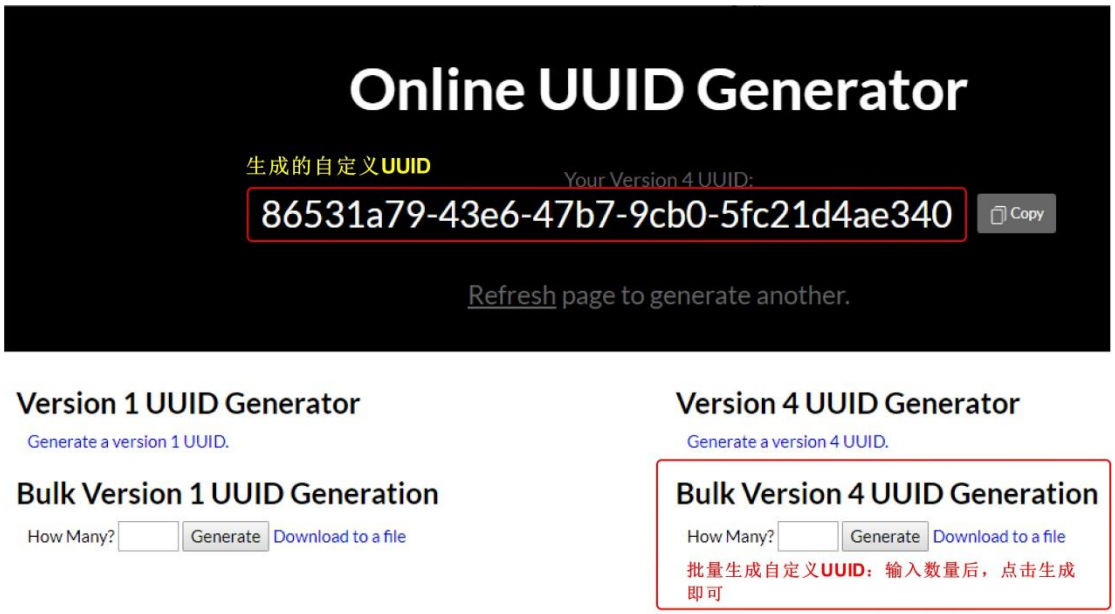


图 7-2：生成自定义 UUID

这里，我们生成的自定义 UUID 是：86531A79-43E6-47B7-9CB0-5FC21D4AE340，接下来我们看一下如何使用生成的自定义 UUID。

对于 UUID，我们知道 UUID 采用的是 128 位 UUID 基数+16 位 UUID 的形式，使用时 16 位 UUID 会替换 128 位 UUID 基数数组中的第 12 和第 13 个字节。标准的 UUID，它的 128 位基数和 16 位 UUID 都是由 Bluetooth SIG 定义好的，我们是不能改变的，而自定义 UUID，128 位基数和 16 位 UUID 都需要我们自己定义的。UUID 详细的描述参见《第四章的第 10 节》。

如下图所示，我们生成自定义 UUID，是为了用它作为 128 位的 UUID 基数，因此，其中的“1A79”是用不上的，因为，我们会使用 16 位的 UUID 替代它。这里，我们会把“1A79”修改为“0000”，并在程序中定义一个数组保存 128 位的 UUID 基数（注意生成的 UUID 和程序中 UUID 基数数组中的字节序）。

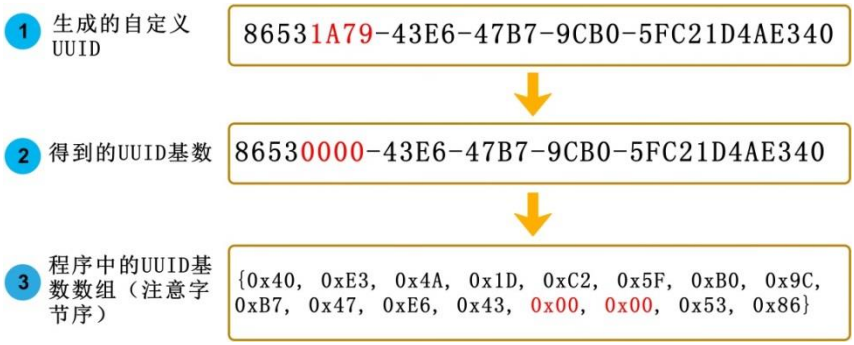


图 7-3：使用生成自定义 UUID

有了自定义的 UUID 基数，我们还需要自己定义 16 位的 UUID，本例中包含了一个服务和两个特征，所以我们要定义三个 16 位的 UUID。

表 7-1：定义服务和特征的 UUID

序号	16 位 UUID	对应的 128 位 UUID
串口透传服务	000A	86530A00-43E6-47B7-9CB0-5FC21D4AE340
TX 特征	000B	86530B00-43E6-47B7-9CB0-5FC21D4AE340
RX 特征	000C	86530C00-43E6-47B7-9CB0-5FC21D4AE340

2.3. 需求表

根据串口透传的功能，我们列出需求表。

表 7-2：串口透传（从机）需求表

序号	项目	名称	内容
1	广播数据	设备名称	BLE_UartApp。
		服务 UUID	扫描响应中包含串口透传服务 UUID。
		外观	不包含外观。
		Flags	一般可发现模式，不支持 BR/EDR。
2	广播参数	广播间隔	100 ms。
		广播超时时间	无限广播，不超时。
		广播模式	快速广播。
3	首选连接参数	最小连接间隔	200ms。
		最大连接间隔	650ms。
		从机延迟	0。
		监督超时	4 秒。

4	连接参数协商	首次协商延时	5 秒。
		每次协商之间的间隔	30 秒。
		最次协商最大尝试次数	3 次。
5	GAP 服务	设备名称	Ble_UartApp。
6	包含的服务	串口透传服务	见《表 7-3：串口透传服务包含的特征》。
7	指示灯	D1 闪烁	指示正在广播。
		D1 常亮	指示已经和中心设备建立连接。

表 7-3：串口透传服务包含的特征

服务	特征名称	特征性质	安全许可
串口透传服务	TX 特征	Notify（通知）	无
	串口透传客户端特征描述符（CCCD）	读，写	无
	RX 特征	写	无

3. 代码实现

本例在开发指南（下册）配套实验“实验 3-3：BLE 工程模板（外围设备）”的基础上修改。

✧ 注：本节对应的实验源码是：“实验 7-1：实现串口透传 Profile”。

3.1. 新建文件和路径设置

本例中会用到 nRF52840 的串口收发数据，所以工程中需要增加 UART 相关的文件和配置，UART 部分可以参考开发指南上册《第 10 章：串口数据收发》，此处不再赘述，这里我们只描述实现串口透传 Profile 相关的文件和配置。

1. 需要新建的文件

1) 在工程的 app 文件夹下面新建 ble_uarts 文件夹，存放串口透传服务文件。
2) 新建如下表所示的文件，保存到 ble_uarts 文件夹，并将“ble_app_uarts.c”添加到工程中。

表 7-4：需要新建的文件

项目	文件名
串口透传服务文件	my_ble_uarts.c
	my_ble_uarts.h

2. 需要引用的头文件

因为在“main.c”文件中需要使用串口透传服务文件，所以“main.c”文件需要引用下面的头文件。

#include " my_ble_uarts.h"

3. 需要添加的头文件包含路径

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

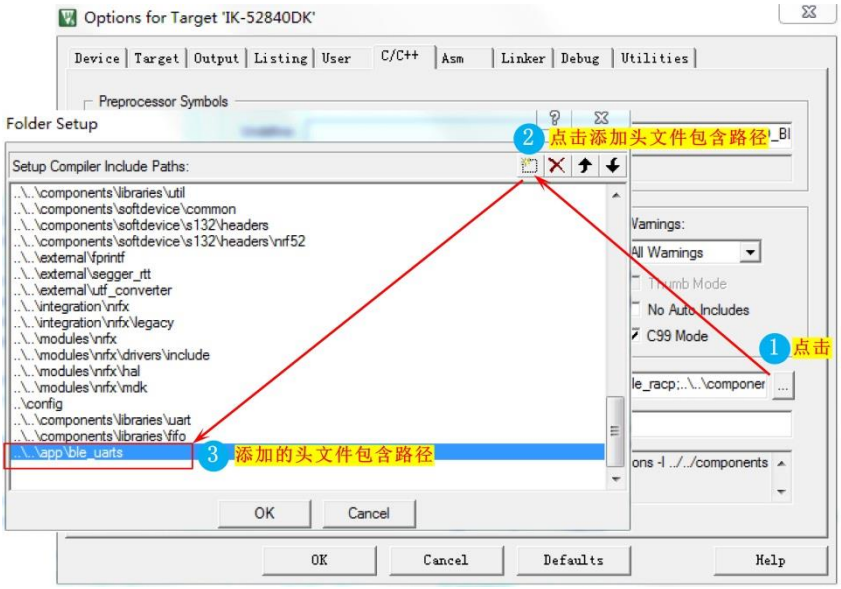


图 7-4：添加头文件包含路径

本例需要添加的头文件路径如下表：

表 7-5： 头文件包含路径

序号	路径
1	..\..\app\ble_uarts

3.2. 工程配置

因为我们新建的串口透传服务文件中加入了宏定义开关执行条件编译，所以需要在“sdk_config.h”文件中加入串口透传的宏定义，并勾选串口透传服务，如下图所示。

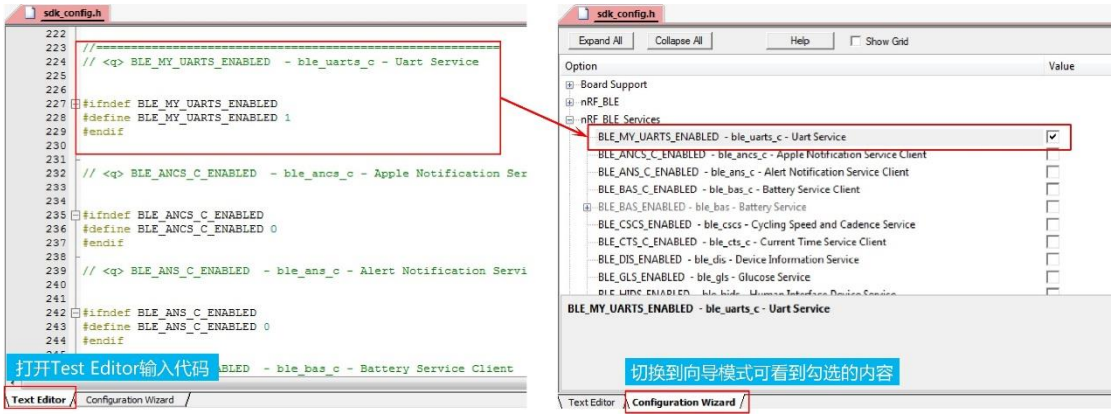


图 7-5：工程配置

3.3. 广播数据和广播参数

广播中包含哪些数据以及广播模式是在广播初始化函数 advertising_init()中完成的，根据前文《需求描述》中对广播数据和广播模式的描述，程序中我们需要加入下面几项。

1. 包含设备名称的全称

广播中包含设备名称全称代码清单如下。

代码清单：包含设备名称全称

```
//设备名称类型：全称
init.advdata.name_type              = BLE_ADVDATA_FULL_NAME;
```

2. 包含外观特征设置

因为串口透传没有对应的外观特征编码因此本例广播数据中不包含外观特征，代码清单如下。

代码清单：不包含外观特征

```
//是否包含外观：包含
init.advdata.include_appearance      = false;
```

3. 设置 flags

广播初始化函数中设置 Flags 为一般可发现模式，不支持 BR/EDR。

代码清单：设置 flags

```
//Flag:一般可发现模式，不支持 BR/EDR
init.advdata.flags           = BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE;
```

4. 包含串口透传服务的 UUID（0x000A）

广播中包含自定义服务的 UUID 和包含标准服务的 UUID 是有所区别的，它们的区别在于数据长度不一样，因此包含自定义服务的 UUID 时要特别注意广播包的剩余空间是否能够容纳的下自定义服务的 UUID。

- 标准服务的 UUID：加入广播数据中的长度为 16 位，2 个字节。
- 自定义服务的 UUID：加入广播数据中的长度为 128 位，16 个字节。

下图描述中广播数据中加入标准和自定义服务 UUID 的情况，可以看到对于自定义 UUID，最多只能包含一个。

加入标准服务UUID的长度

LEN	AD Type	第1个UUID	第2个UUID	.....	第n个UUID
1个字节	1个字节	2个字节	2个字节	.....	2个字节

加入标准服务UUID的长度 = 2 + 2*UUID数量

- 加入1个标准服务UUID的长度 = 2 + 2*1 =4个字节
- 加入2个标准服务UUID的长度 = 2 + 2*2 =6个字节

加入自定义服务UUID的长度

LEN	AD Type	UUID
1个字节	1个字节	16个字节

加入自定义服务UUID的长度 = 2 + 16 = 18个字节

- 广播包中最多只能包含一个自定义服务的UUID

图 7-6：广播中加入服务的 UUID 长度

广播中一般不会仅仅包含服务的 UUID，还会包含 flags、设备名称等信息，如果广播包中的剩余空间不足以容纳服务的 UUID，而我们的应用中又必须要将服务的 UUID 加入到广播包，这时候可以将服务的 UUID 放到扫描响应包里面。

本例中，我们就是将自定义的串口透传服务的 UUID 加入到了扫描响应包，因为广播包无法容纳，请看下面的计算。

- 包含 flags 的字节数：长度 + AD Type + flags = 3 个字节。
- 包含设备名称：长度 + AD Type + 设备名称 = 13 个字节。

Flags 和设备名称一共占用了 16 个字节，广播包最大长度是 31 个字节，所以剩余空间是 15 个字节，而包含 1 个自定义服务 UUID 的长度是 18 个字节，因此，广播包无法容纳。

理解了广播中包含标准服务和自定义服务 UUID 的区别后，广播中包含服务 UUID 就很容易实现了，首先定义并初始化 UUID 列表，之后在广播初始化函数中配置扫描响应包含串口透传服务的 UUID 即可。

代码清单：定义并初始化服务 UUID 列表

```
//定义串口透传服务 UUID 列表
static ble_uuid_t m_adv_uuids[]          =
{
    {BLE_UUID_UARTS_SERVICE, UARTS_SERVICE_UUID_TYPE}
};
```

根据《需求描述》的要求，我们需要包含的是全部首要服务的 UUID，本例中即包含串口透传服务的 UUID。

代码清单：扫描响应中包含服务的 UUID

```
//UUID 放到扫描响应里面
init.srdata.uuids_complete.uuid_cnt = sizeof(m_adv_uuids) / sizeof(m_adv_uuids[0
                                      ]);
init.srdata.uuids_complete.p_uuids  = m_adv_uuids;
```

对于广播中包含自定义服务 UUID，除了注意数据长度之外，还需要注意广播初始化中会用到自定义的 UUID 的基数，而自定义 UUID 的基数在是服务初始化时写入到 Softdevice 的，因此服务初始化必须在广播初始化之前执行。

5. 设置广播参数

根据《需求描述》中广播参数的描述，定义广播间隔为 100ms（100/0.625=160），广播持续时间为 0（0=广播不超时），代码清单如下。

代码清单：广播间隔和广播超时时间

```
1.  #define APP_ADV_INTERVAL    160  // 广播间隔 (100 ms)，单位 0.625 ms
2. #define APP_ADV_DURATION    0    // 广播持续时间，单位：10ms。设置为 0 表示不超时
```

之后，在广播初始化函数中设置广播参数：广播间隔、广播超时时间和广播模式。

代码清单：设置广播参数

```
1.  //设置广播模式为快速广播
2. init.config.ble_adv_fast_enabled  = true;
3. //设置广播间隔和广播持续时间
4. init.config.ble_adv_fast_interval = APP_ADV_INTERVAL;
5. init.config.ble_adv_fast_timeout  = APP_ADV_DURATION;
```

3.4. 首选连接参数和连接参数协商

连接参数协商协商的时间和次数定义如下。

- 首次协商延时：5 秒。
- 每次协商之间的间隔：30 秒。
- 协商最大尝试次数：3 次。

代码清单：连接参数协商时间和次数

```
1.  #define FIRST_CONN_PARAMS_UPDATE_DELAY  APP_TIMER_TICKS(5000)  // 定义首次调用
    sd_ble_gap_conn_param_update()函数更新连接参数延迟时间（5 秒）
2. #define NEXT_CONN_PARAMS_UPDATE_DELAY   APP_TIMER_TICKS(30000) // 定义每次调用
   sd_ble_gap_conn_param_update()函数更新连接参数的间隔时间（30 秒）
3. #define MAX_CONN_PARAMS_UPDATE_COUNT    3                      // 定义放弃连接参数
   协商前尝试连接参数协商的最大次数（3 次）
```

连接参数数值和连接参数协商时间和次数的定义会在连接参数协商初始化函数 conn_params_init()中使用，协商初始化函数使用这些参数初始化配置连接参数协商模块，配置成功后，连接参数协商会按照设置的参数自动进行，协商程序模块会将协商结果以事件的方式通知应用程序，应用程序在连接参数协商模块事件处理函数 on_conn_params_evt()中处理协商事件。

3.5. GAP 服务

设备名称、外观和首选连接参数是 GAP 服务的特征，它们是在 GAP 初始化函数 gap_params_init()函数中设置的。

根据《需求描述》中对设备名称的描述，修改设备名称字符串为“BLE_UartApp”。

代码清单：定义设备名称字符串

```
1. #define DEVICE_NAME     " BLE_UartApp"      // 设备名称字符串
```

GAP 初始化函数中设置 GAP 安全模式为无安全性，并写入设备名称，代码清单如下。

代码清单：定义设备名称字符串

```
1. ble_gap_conn_sec_mode_t sec_mode;
2.  //设置 GAP 的安全模式
3.  BLE_GAP_CONN_SEC_MODE_SET_OPEN(&sec_mode);
4.  //设置 GAP 设备名称
5.  err_code = sd_ble_gap_device_name_set(&sec_mode,
6.                                        (const uint8_t *)DEVICE_NAME,
7.                                        strlen(DEVICE_NAME));
```

根据《需求描述》中对首选连接参数的描述，修改连接参数的数值，如下所示。

- 最小连接间隔：200ms。
- 最大连接间隔：650ms。
- 从机延迟：0。
- 监督超时：4 秒。

代码清单：定义连接参数数值

```
1. #define MIN_CONN_INTERVAL   MSEC_TO_UNITS(200, UNIT_1_25_MS) // 最小连接间
   隔 (0.2 秒)
2. #define MAX_CONN_INTERVAL   MSEC_TO_UNITS(650, UNIT_1_25_MS) // 最大连接间
   隔 (0.65 秒)
3. #define SLAVE_LATENCY       0                                // 从机延迟
4. #define CONN_SUP_TIMEOUT    MSEC_TO_UNITS(4000, UNIT_10_MS)  // 监督超时(4 秒)
```

之后，在 GAP 初始化函数中设置首选连接参数，代码清单如下。

代码清单：设置首选连接参数

```
1.  //设置首选连接参数，设置前先清零 gap_conn_params
2. memset(&gap_conn_params, 0, sizeof(gap_conn_params));
3.
4. gap_conn_params.min_conn_interval = MIN_CONN_INTERVAL;//最小连接间隔
5. gap_conn_params.max_conn_interval = MAX_CONN_INTERVAL;//最小连接间隔
6. gap_conn_params.slave_latency     = SLAVE_LATENCY;     //从机延迟
7. gap_conn_params.conn_sup_timeout  = CONN_SUP_TIMEOUT; //监督超时
8. //调用协议栈 API sd_ble_gap_ppcp_set 配置 GAP 参数
9. err_code = sd_ble_gap_ppcp_set(&gap_conn_params);
10.APP_ERROR_CHECK(err_code);
```

3.6. 添加串口透传服务

下图是加入串口透传服务的一般流程，和加入心率服务的流程基本一样，所不同的是心率服务是标准服务，所以心率服务结构体需要参照 Bluetooth SIG 采纳的《心率服务规格书》来声明。而串口透传服务是自定义服务，没有对应的规格书可以参照，它需要我们根据具体的需求来声明。

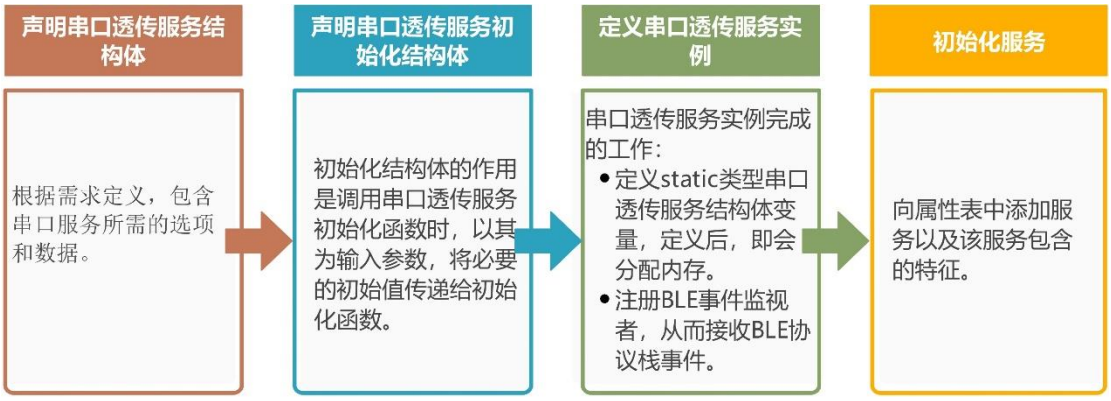


图 7-7：添加串口透传服务的流程

3.6.1. 声明串口透传服务结构体

声明串口透传服务结构体之前，我们不妨先考虑一下，串口透传服务需要包含哪些必要的元素。

1) 串口透传服务需要一个服务句柄变量，添加服务时用来保存协议栈返回的服务句柄。
2) 串口透传服务包含两个特征（TX 特征和 RX 特征），我们需要操作 TX 和 RX 特征，所以需要 TX 特征和 RX 特征句柄变量，添加特征时用来保存协议栈返回的特征句柄。如果不需要操作特征，则可以忽略特征句柄。
3) 添加服务时需要指明服务的 UUID 类型（标准的还是自定义的），所以，需要包含 UUID 的类型。
4) 串口透传服务需要接收主机发送过来的数据，因此我们需要一个函数指针变量，当串口透传服务的 BLE 事件监视者接收到协议栈事件后，若是“写 RX 特征值事件”，则通过函数指针变量执行回调。
5) 其他：我们还可以根据自己的需求在串口透传服务结构体里面加入其他的一些变量，如定义一个用来标志“Notify”是否使能的变量等等。

综上所述，本例中串口透传服务结构体的声明如下。

代码清单：串口透传服务结构体声明

```
//串口透传服务结构体，包含服务所需要的信息
struct ble_uarts_s
{
    uint8_t                    uuid_type;        //UUID 类型
    uint16_t                   service_handle;   //串口透传服务句柄（由协议栈提供）
    ble_gatts_char_handles_t   tx_handles;       //TX 特征句柄
    ble_gatts_char_handles_t   rx_handles;       //RX 特征句柄
```

```
    ble_uarts_data_handler_t   data_handler;    //处理接收数据的事件句柄
};
```

3.6.2. 声明串口透传服务初始化结构体

服务初始化结构体用来在添加服务之前初始化一些必要的参数，对于串口透传服务，它的结构体里面的 5 个变量只有事件句柄是应用程序提供的，需要初始化，因此服务初始化结构体声明如下。

代码清单：串口透传服务初始化结构体声明

```
//串口服务初始化结构体，
typedef struct
{
    ble_uarts_data_handler_t data_handler; //处理接收数据的事件句柄
} ble_uarts_init_t;
```

其中 ble_uarts_data_handler_t 是我们定义的函数指针，该函数指针指向的函数带有 1 个 ble_uarts_evt_t 类型参数，用来传递串口事件类型。

代码清单：定义函数指针类型 ble_uarts_data_handler_t

```
//定义函数指针类型 ble_uarts_data_handler_t
typedef void (* ble_uarts_data_handler_t) (ble_uarts_evt_t * p_evt);
```

本例中定义了如下两种串口透传事件 BLE_UARTS_EVT_RX_DATA 和 BLE_UARTS_EVT_TX_RDY，分别用来指示接收到新的数据和数据发送完成。

代码清单：定义串口透传事件类型

```
//定义串口透传事件类型
typedef enum
{
    BLE_UARTS_EVT_RX_DATA,      //接收到新的数据
    BLE_UARTS_EVT_TX_RDY,       //准备就绪，可以发送新数据
} ble_uarts_evt_type_t;
```

3.6.3. 定义串口透传服务实例

应用程序中使用宏定义 BLE_HRS_DEF 实例化一个串口透传服务，下面的代码实例化一个名称为“m_uarts”的串口透传服务实例。

代码清单：定义名称为 m_uarts 的串口透传服务实例

```
//定义串口透传服务实例
BLE_UARTS_DEF(m_uarts);
```

BLE_UARTS_DEF 是一个带参数的宏，输入参数“_name”即为实例化的心率服务的名称，该宏定义展开后代码如下。

代码清单：定义串口透传服务实例

```
1.  #define BLE_UARTS_DEF(_name)                                    \
2.      static ble_uarts_t _name;                                   \
3.      NRF_SDH_BLE_OBSERVER(_name ## _obs,                         \
4.                           BLE_NUS_BLE_OBSERVER_PRIO,             \
5.                           ble_uarts_on_ble_evt,                  \
6.                           &_name)
```

3. 行 2：定义了 static 类型串口透传服务结构体变量，为串口透传服务结构体分配了内存，该语句执行后，串口透传服务结构体变量常驻内存，即实例化了一个名为“_name”的串口透传服务，协议栈和应用程序均可通过该名称访问串口透传服务实例。
4. 行 3~行 5：注册了名称为“_name_obs”的串口透传服务 BLE 事件监视者，注册成功后，当协议栈有事件产生时，会告知 BLE 事件监视者“_name_obs”，“_name_obs”接收到协议栈的事件后，进入事件处理函数 ble_uarts_on_ble_evt()，在该函数中根据事件的类型调用相应的函数完成处理。

■ **编写事件回调函数**

注册 BLE 事件监视者时提供的事件回调函数具有规定的格式，这是我们编写回调函数时需要遵循的，这一点我们在《第六章：实现一个标准的 Profile》中已经提到过。

本例中，我们关注协议栈提交的事件中的 2 个事件：BLE_GATTS_EVT_WRITE 事件和 BLE_GATTS_EVT_HVN_TX_COMPLETE 事件，代码清单如下。

- BLE_GATTS_EVT_WRITE 事件：写事件，接收主机发送的数据。
- BLE_GATTS_EVT_HVN_TX_COMPLETE：TX 就绪事件。

代码清单：串口透传服务 BLE 事件监视者的事件回调函数

```
1.  //串口透传服务 BLE 事件监视者的事件回调函数
2.  void ble_uarts_on_ble_evt(ble_evt_t const * p_ble_evt, void * p_context)
3.  {
4.      //检查参数是否有效
5.      if ((p_context == NULL) || (p_ble_evt == NULL))
6.      {
7.          return;
8.      }
9.      //定义一个串口透传结构体指针并指向串口透传结构体
10.     ble_uarts_t * p_uarts = (ble_uarts_t *)p_context;
11.     //判断事件类型
12.     switch (p_ble_evt->header.evt_id)
13.     {
14.         //写事件
15.         case BLE_GATTS_EVT_WRITE:
```

```
            //接收主机发送的数据（主机是通过“写 TX 特征值”将数据发送给串口透传设备的）
            on_write(p_uarts, p_ble_evt);
            break;
         //TX 就绪事件
         case BLE_GATTS_EVT_HVN_TX_COMPLETE:
            on_hvx_tx_complete(p_uarts, p_ble_evt);
            break;
         default:
            break;
      }
}
```

代码中的 on_write()和 on_hvx_tx_complete ()函数是实现具体功能的代码，其中 on_write()用于处理主机“写 TX 特征值”，也就是处理主机向从机发送的透传数据，其代码清单如下。该函数中对事件进行了判断，如果是“写 TX 特征值”事件，则执行串口透传服务初始化时注册的事件回调函数。

代码清单：on_write()函数

```
static void on_write(ble_uarts_t * p_uarts, ble_evt_t const * p_ble_evt)
{
   //定义一个串口透传事件结构体变量，用于执行回调时传递参数
   ble_uarts_evt_t                 evt;
   //定义 write 事件结构体指针并指向 GATT 事件中的 wirite
   ble_gatts_evt_write_t const * p_evt_write = &p_ble_evt->evt.gatts_evt.params.
                                                write;
   //清零 evt
   memset(&evt, 0, sizeof(ble_uarts_evt_t));
   //指向串口透传实例
   evt.p_uarts       = p_uarts;
   //设置连接句柄
   evt.conn_handle = p_ble_evt->evt.gatts_evt.conn_handle;
   //写 RX 特征值
   if ((p_evt_write->handle == p_uarts->rx_handles.value_handle) &&
            (p_uarts->data_handler != NULL))
   {
      //设置事件类型
      evt.type                  = BLE_UARTS_EVT_RX_DATA;
      //设置数据长度
      evt.params.rx_data.p_data = p_evt_write->data;
      //设置数据长度
      evt.params.rx_data.length = p_evt_write->len;
      //执行回调
      p_uarts->data_handler(&evt);
   }
```

```
    else
    {
        //该事件和串口透传服务无关，忽略
    }
}
```

on_hvx_tx_complete()函数用于处理通知发送完成事件，如果我们需要关心通知是否发送完成则处理这个事件，否则不用处理该事件。

代码清单：on_hvx_tx_complete ()函数

```
static void on_hvx_tx_complete(ble_uarts_t * p_uarts, ble_evt_t const * p_ble_evt)
{
    //定义一个串口透传事件结构体变量 evt，用于执行回调时传递参数
    ble_uarts_evt_t              evt;
    //清零 evt
    memset(&evt, 0, sizeof(ble_uarts_evt_t));
    //设置事件类型
    evt.type        = BLE_UARTS_EVT_TX_RDY;
    //指向串口透传服务实例
    evt.p_uarts     = p_uarts;
    //设置连接句柄
    evt.conn_handle = p_ble_evt->evt.gatts_evt.conn_handle;
    //指向 link context
    evt.p_link_ctx  = p_client;
    //执行回调
    p_uarts->data_handler(&evt);
}
```

on_write()和 on_hvx_tx_complete ()函数最终都会执行串口透传服务初始化时注册的串口透传事件回调函数（p_uarts->data_handler(&evt);执行了回调），该函数中，我们可以读取主机发送过来的数据以及判断通知是否发送完成。

3.6.4. 初始化串口透传服务

services_init()函数中加入串口透传服务初始化的代码，首先定义一个串口透传服务初始化结构体，并设置串口透传的事件回调函数，之后调用 ble_uarts_init()函数初始化串口透传服务，代码如下。

代码清单：初始化串口透传服务

```
//清零串口透传服务初始化结构体
memset(&uarts_init, 0, sizeof(uarts_init));
//设置串口透传事件回调函数
uarts_init.data_handler = uarts_data_handler;
```

```
//初始化串口透传服务
err_code = ble_uarts_init(&m_uarts, &uarts_init);
APP_ERROR_CHECK(err_code);
```

■ **ble_uarts_init()函数**

ble_uarts_init()函数代码清单如下，可以看到自定义服务的初始化和标准服务的初始化流程是类似的，主要区别是自定义服务的初始化需要将自定义 UUID 基数写入到 Softdevice。

代码清单：ble_uarts_init()函数

```
uint32_t ble_uarts_init(ble_uarts_t * p_uarts, ble_uarts_init_t const * p_uarts_init)
{
    ret_code_t            err_code;
    //定义 16 位 UUID 结构体变量
    ble_uuid_t            ble_uuid;
    //定义 128 位 UUID 结构体变量，并初始化为串口透传服务 UUID 基数
    ble_uuid128_t         nus_base_uuid = UARTS_BASE_UUID;
    //定义特征参数结构体变量
    ble_add_char_params_t add_char_params;
    //检查指针是否为 NULL
    VERIFY_PARAM_NOT_NULL(p_uarts);
    VERIFY_PARAM_NOT_NULL(p_uarts_init);

    //拷贝串口透传服务初始化结构体中的事件句柄
    p_uarts->data_handler = p_uarts_init->data_handler;

    //将自定义 UUID 基数添加到 SoftDevice
    err_code = sd_ble_uuid_vs_add(&nus_base_uuid, &p_uarts->uuid_type);
    VERIFY_SUCCESS(err_code);
    //UUID 类型和数值赋值给 ble_uuid 变量
    ble_uuid.type = p_uarts->uuid_type;
    ble_uuid.uuid = BLE_UUID_UARTS_SERVICE;

    //添加串口透传服务
    err_code = sd_ble_gatts_service_add(BLE_GATTS_SRVC_TYPE_PRIMARY,
                                        &ble_uuid,
                                        &p_uarts->service_handle);
    VERIFY_SUCCESS(err_code);
    /*---------------------以下代码添加 RX 特征--------------------*/
    //配置参数之前先清零 add_char_params
    memset(&add_char_params, 0, sizeof(add_char_params));
    //RX 特征的 UUID
    add_char_params.uuid                    = BLE_UUID_UARTS_RX_CHARACTERISTIC;
    //RX 特征的 UUID 类型
```

```
35.   add_char_params.uuid_type                = p_uarts->uuid_type;
36.   //设置 RX 特征值的最大长度
37.   add_char_params.max_len                   = BLE_UARTS_MAX_RX_CHAR_LEN;
38.   //设置 RX 特征值的初始长度
39.   add_char_params.init_len                  = sizeof(uint8_t);
40.   //设置 RX 的特征值长度为可变长度
41.   add_char_params.is_var_len                = true;
42.   //设置 RX 特征的性质支持写
43.   add_char_params.char_props.write          = 1;
44.   //设置 RX 特征的性质支持无响应写
45.   add_char_params.char_props.write_wo_resp = 1;
46.   //设置读/写 RX 特征值的安全需求：无安全性
47.   add_char_params.read_access   = SEC_OPEN;
48.   add_char_params.write_access = SEC_OPEN;
49.   //为串口透传服务添加 RX 特征
50.   err_code = characteristic_add(p_uarts->service_handle, &add_char_params,
51.                                 &p_uarts->rx_handles);
52.   //检查返回的错误代码
53.   if (err_code != NRF_SUCCESS)
54.   {
55.      return err_code;
56.   }
57.   /*---------------------添加 RX 特征-END------------------------*/
58.
59.   /*---------------------以下代码添加 TX 特征--------------------*/
60.   //配置参数之前先清零 add_char_params
61.   memset(&add_char_params, 0, sizeof(add_char_params));
62.   //TX 特征的 UUID
63.   add_char_params.uuid              = BLE_UUID_UARTS_TX_CHARACTERISTIC;
64.   //TX 特征的 UUID 类型
65.   add_char_params.uuid_type         = p_uarts->uuid_type;
66.   //设置 TX 特征值的最大长度
67.   add_char_params.max_len           = BLE_UARTS_MAX_TX_CHAR_LEN;
68.   //设置 TX 特征值的初始长度
69.   add_char_params.init_len          = sizeof(uint8_t);
70.   //设置 TX 的特征值长度为可变长度
71.   add_char_params.is_var_len        = true;
72.   //设置 TX 特征的性质：支持通知
73.   add_char_params.char_props.notify = 1;
74.   //设置读/写 RX 特征值的安全需求：无安全性
75.   add_char_params.read_access       = SEC_OPEN;
76.   add_char_params.write_access      = SEC_OPEN;
77.   add_char_params.cccd_write_access = SEC_OPEN;
78.   //为串口透传服务添加 TX 特征
```

```
79.    return characteristic_add(p_uarts->service_handle, &add_char_params,
80.                              &p_uarts->tx_handles);
81.    /*---------------------添加 TX 特征-END------------------------*/
82.}
```

■ **串口透传事件回调函数 uarts_data_handler**

串口服务初始化时注册的事件回调函数，也就是“3.6.3”节中 on_write()和 on_hvx_tx_complete ()函数是最终执行回调的函数，因此该函数一旦被调用即表示产生了 BLE_UARTS_EVT_RX_DATA（接收到新的数据）或者 BLE_UARTS_EVT_TX_RDY（准备就绪，可以发送新数据）事件。由此可知，该事件回调函数中，我们根据产生的事件类型，即可判断出产生的事件，从而执行相应的动作。

- 接收 BLE 主机发过来的数据：若产生的事件类型是 BLE_UARTS_EVT_RX_DATA，表示接收到 BLE 主机发送过来的数据，我们可以取出数据进行处理，本例中取出数据后通过串口打印出数据。
- 判断 BLE 发送是否就绪：若产生的事件类型是 BLE_UARTS_EVT_TX_RDY，表示发送已经准备就绪，可以发送新数据。

本例中，串口服务初始化事件回调函数代码如下：

代码清单：ble_uarts_init()函数

```
1.  static void uarts_data_handler(ble_uarts_evt_t * p_evt)
2.  {
3.     //判断事件类型:接收到新数据事件
4.     if (p_evt->type == BLE_UARTS_EVT_RX_DATA)
5.     {
6.        uint32_t err_code;
7.        //串口打印出接收的数据
8.        for (uint32_t i = 0; i < p_evt->params.rx_data.length; i++)
9.        {
10.          do
11.          {
12.             err_code = app_uart_put(p_evt->params.rx_data.p_data[i]);
13.             if ((err_code != NRF_SUCCESS) && (err_code != NRF_ERROR_BUSY))
14.             {
15.                NRF_LOG_ERROR("Failed receiving NUS message. Error 0x%x. ",
16.                err_code);
17.                APP_ERROR_CHECK(err_code);
18.             }
19.          } while (err_code == NRF_ERROR_BUSY);
20.       }
21.       if (p_evt->params.rx_data.p_data[p_evt->params.rx_data.length - 1] ==
22.       '\r')
23.       {
24.          while (app_uart_put('\n') == NRF_ERROR_BUSY);
```

```
25.         }
26.     }
27.     //判断事件类型:发送就绪事件，该事件在后面的实验会用到，当前我们在该事件中翻转指示灯 D4
28.     //的状态，指示该事件的产生
29.     if (p_evt->type == BLE_UARTS_EVT_TX_RDY)
30.     {
31.         nrf_gpio_pin_toggle(LED_4);
32.     }
33. }
```

串口透出服务初始化函数中还向 SoftDevice 中添加了自定义 UUID，对于自定义服务来说，我们必须要掌握如何将自定义 UUID 基数添加到 SoftDevice，下面我们通过一个单独的章节来描述如何添加自定义 UUID。

3.6.5. 自定义 UUID 基数添加到 SoftDevice

自定义 UUID 基数添加到 SoftDevice 需要 3 个步骤完成，如下图所示。

图 7-8：添加自定义 UUID 的步骤

■ **编写代码**

自定义 UUID 基数通过 sd_ble_uuid_vs_add()函数写入到 SoftDevice，sd_ble_uuid_vs_add()函数原型如下。

表 7-6：`sd_ble_uuid_vs_add()`函数

函数原型	uint32_t sd_ble_uuid_vs_add (ble_uuid128_t const * p_vs_uuid, uint8_t * p_uuid_type)
函数功能	添加厂商自定义 UUID 基数到 SoftDevice。 注意：提供的 UUID 的第 12 和 13 个字节的值会被忽略，因为它们总是被 ble_uuid_t 中的 16 位 uuid 字段替换。

	如果添加的 UUID 在 BLE 堆栈的内部表中已存在，则将在 p_uuid_type 中返回相应的索引以及错误代码。
参　　数	[in] p_vs_uuid：指向 128 位小端模式的自定义 UUID。注意自定义 UUID 数组中的第 12 和第 13 个字节会被忽略。 [out] p_uuid_type：指向用来保存 SoftDevice 返回 UUID 类型的 8 位变量，向协议栈写入 UUID 时，协议栈会返回对应的 UUID 类型。
返回值	NRF_SUCCESS：加入 UUID 成功。 NRF_ERROR_INVALID_ADDR：p_vs_uuid 或 p_uuid_type 为 NULL 或不存在。 NRF_ERROR_NO_MEM：内存不足，没有足够的空间存储自定义 UUID。

添加自定义 UUID 的代码如下，本例中首先定义一个 ble_uuid128_t 类型变量 nus_base_uuid，将自定义的 128 位 UUID 赋值给该变量，之后调用 sd_ble_uuid_vs_add()函数（nus_base_uuid 变量为输入参数）写入 UUID，写入成功后 SoftDevice 返回 UUID 类型保存到串口透传服务实例的成员变量 uuid_type 中。

代码清单：写入自定义 UUID

```
//定义串口透传服务 128 位 UUID 基数
#define UARTS_BASE_UUID      {{0x40, 0xE3, 0x4A, 0x1D, 0xC2, 0x5F, 0xB0, 0x9C, 0xB7, 0x47, 0xE6, 0x43, 0x00, 0x00, 0x53, 0x86}}

//定义 128 位 UUID 结构体变量，并初始化为串口透传服务 UUID 基数
ble_uuid128_t         nus_base_uuid = UARTS_BASE_UUID;

//将自定义 UUID 基数添加到 SoftDevice
err_code = sd_ble_uuid_vs_add(&nus_base_uuid, &p_uarts->uuid_type);
VERIFY_SUCCESS(err_code);
```

■ "sdk_config.h"文件中设置自定义 UUID 数量

打开"sdk_config.h"文件，切换到“Configuration”，根据程序中实际添加的自定义 UUID 设置 NRF_SDH_BLE_VS_UUID_COUNT 的数值，本例中我们只添加了一个自定义 UUID，因此设置为 1。

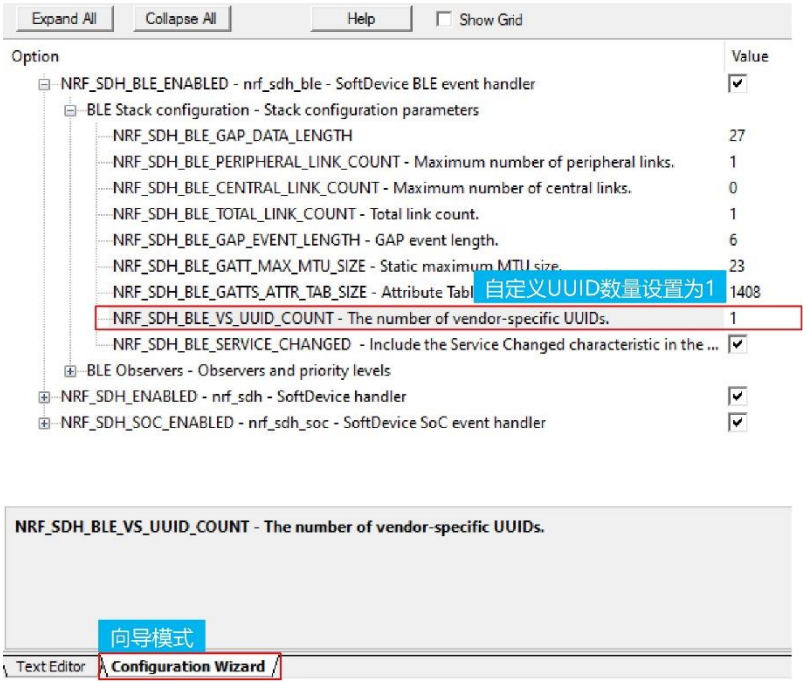


图 7-9：设置自定义 UUID 数量

3.7. 发送数据

本例中，当串口接收到一包数据后通过 Notify 将数据发送给 BLE 主机，串口事件处理函数中并不是每接收一个字节数据就执行一次发送的，串口接收数据时，会将接收的数据先保存到缓存数组，当接收的数据长度达到设定的最大值或者接收到换行符后，则认为一包数据接收完成，这时候才会将接收的数据发送给主机。

代码清单：串口事件处理函数

```
void uart_event_handle(app_uart_evt_t * p_event)
{
    static uint8_t data_array[BLE_UARTS_MAX_DATA_LEN];
    static uint8_t index = 0;
    uint32_t       err_code;
    //判断事件类型
    switch (p_event->evt_type)
    {
        case APP_UART_DATA_READY://串口接收事件
            UNUSED_VARIABLE(app_uart_get(&data_array[index]));
            index++;
            //接收串口数据，当接收的数据长度达到 m_ble_uarts_max_data_len 或者接收到换行
            //符后认为一包数据接收完成
            if ((data_array[index - 1] == '\n') ||
                (data_array[index - 1] == '\r') ||
                (index >= m_ble_uarts_max_data_len))
            {
                if (index > 1)
```

```
19.                 {
20.                     NRF_LOG_DEBUG("Ready to send data over BLE NUS");
21.                     NRF_LOG_HEXDUMP_DEBUG(data_array, index);
22.                     //串口接收的数据使用 notify 发送给 BLE 主机
23.                     do
24.                     {
25.                       uint16_t length = (uint16_t)index;
26.                       err_code = ble_uarts_data_send(&m_uarts, data_array, &length
27.                                                      , m_conn_handle);
28.                       if ((err_code != NRF_ERROR_INVALID_STATE) &&
29.                           (err_code != NRF_ERROR_RESOURCES) &&
30.                           (err_code != NRF_ERROR_NOT_FOUND))
31.                       {
32.                          APP_ERROR_CHECK(err_code);
33.                       }
34.                     } while (err_code == NRF_ERROR_RESOURCES);
35.                 }
36.                 index = 0;
37.             }
38.             break;
39.         //通讯错误事件，进入错误处理
40.         case APP_UART_COMMUNICATION_ERROR:
41.             APP_ERROR_HANDLER(p_event->data.error_communication);
42.             break;
43.         //FIFO 错误事件，进入错误处理
44.         case APP_UART_FIFO_ERROR:
45.             APP_ERROR_HANDLER(p_event->data.error_code);
46.             break;
47.         default:
48.            break;
49.     }
50.}
```

发送 Notify 时，需要注意下面几点：

- Notify 只有在连接建立之后才可以发送，所以在发送通知时必须检查连接是否建立，如果在未建立连接的情况下发送通知，则会返回错误代码 NRF_ERROR_NOT_FOUND，ble_uarts_data_send()函数中检测了连接状态。
- Notify 只有使能了通知后才能发送，所以比较好的做法是先判断通知是否使能，然后再发送数据。

3.8. 下载验证

1. 解压“…\4：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 7-1：实现串口透传 Profile”，将解压后得到的文件夹“ble_app_uarts”拷贝合适的目录，如

“D\nRF52840”。

2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_uarts\project\mdk5”目录下的工程“ble_app_uarts.uvproj”。
4. 切换到 Target“s132_nrf52_6.1.0_softdevice”，点击下载协议栈。
5. 切换到 Target“IK-52840DK”，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
6. 程序运行后，指示灯 D1 闪烁表示正在广播。
7. 手机上打开 nRF Connect，点击“SCAN”按钮启动扫描，连接成功后指示灯 D1 常亮，扫描到设备“BLE_UartApp”后连接设备并使能通知，如下图所示。

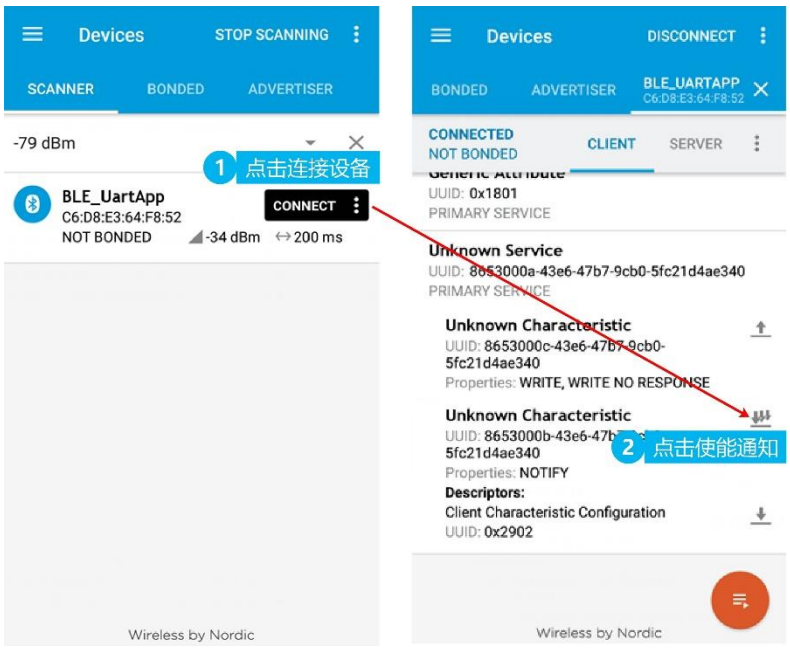


图 7-10：连接设备并使能通知

8. nRF Connect 发送数据给开发板：按照下图所示，填入发送的数据后点击“SEND”按钮发送，发送后在计算机上的串口调试助手中可以看到发送的数据，注意这里发送的是 16 进制数据，因此串口调试助手要选择“HEX 显示”。

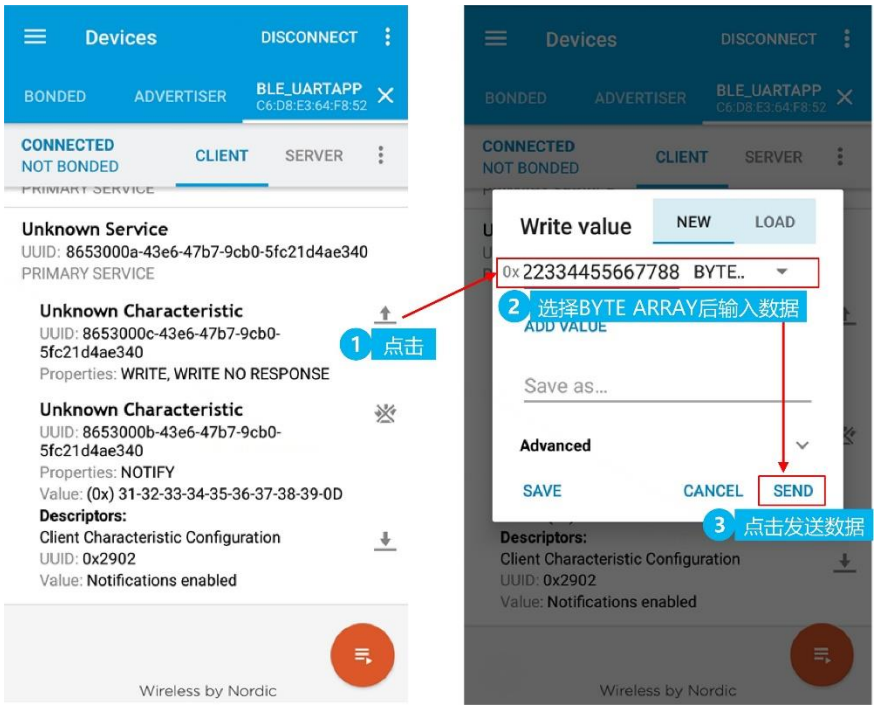


图 7-11：nRF Connect 发送数据给开发板

9. 开发板发送数据给 nRF Connect：串口调试助手发送栏填入发送的数据并勾选“发送新行”后点击发送按钮发送数据，nRF Connect 上会显示接收到的数据，如下图所示，接收到的数据也是以 16 进制显示的，这里我们发送的是字符“123456789”，显示的 16 进制如下，可以看到数据后面多了一个“0D”，这是因为发送数据时勾选了“发送新行”后多发了一个字节的数据。

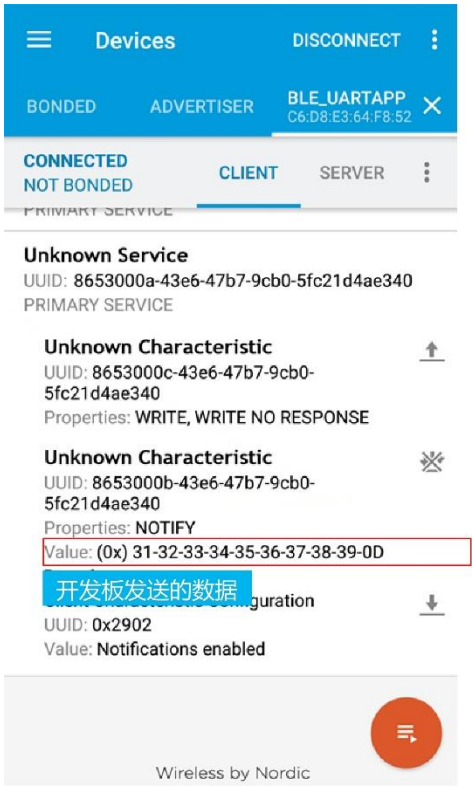


图 7-12：开发板发送数据给 nRF Connect

4. 实现长包传输

本例在实验“实验 7-1：实现串口透传 Profile”的基础上修改，对应的实验源码是：“实验 7-2：串口透传长包传输”。

4.1. MTU 交换

蓝牙 4.2 之前，BLE 的 MTU（Maximum Transmission Unit，最大传输单元）为 23 个字节，这 23 个字节中包含了 1 个字节的操作码（op code）和 2 个字节的属性句柄（attribute handle），因此一次最多传输 20 个字节。从蓝牙 4.2 开始，BLE 支持长包传输，MTU 扩展到了 247 个字节，除去 op code 和 attribute handle，一次可传输的最大长度为 244 个字节。

有这两种 MTU 存在，那么客户端和服务器是如何知道使用什么样的 MTU 的？

客户端和服务器是通过协商来交换各自支持的 MTU 的长度的，这个过程称为 MTU 交换（Exchange MTU），通过 MTU 交换，客户端和服务器具备了自适应 MTU 的功能，但是注册每个建立的连接只能在连接建立时交换一次，而不能实时通过 MTU 交换协商数据传输长度，MTU 交换的步骤如下：

1. **客户端发送 MTU 交换请求：**连接建立时，客户端使用 MTU 交换请求（Exchange MTU Request）通知服务器自己的最大接收 MTU 大小，并请求服务器以其最大接收 MTU 大小进行响应。MTU 交换请求命令格式如下表所示。

表 7-7：MTU 交换请求格式

参数	大小（字节数）	描述
Attribute Opcode	1	0x02 = MTU 交换请求。
Client Rx MTU	2	客户端接收 MTU 的大小。

客户端 Rx MTU 的值应大于或等于默认的 ATT_MTU，MTU 交换请求仅在客户端连接期间发送一次。客户端 Rx MTU 参数应设置为客户端可以接收的属性协议 PDU 的最大大小。

2. **服务器发送 MTU 交换响应：**服务器接收到 MTU 交换请求后，发送 MTU 交换响应以回复收到的 MTU 交换请求。MTU 交换响应命令格式如下表所示。

表 7-8：MTU 交换响应格式

参数	大小（字节数）	描述
Attribute Opcode	1	0x03 = MTU 交换响应。
Server Rx MTU	2	服务器接收 MTU 的大小。

服务器 Rx MTU 的值应大于或等于默认的 ATT_MTU，服务器 Rx MTU 参数应设置为服务器可以接收的属性协议 PDU 的最大大小。

指向 MTU 交换之后，客户端和服务器应将 MTU 设置为交换的最小值，并且客户端和服务器的 MTU 应设置为相同的值，以确保客户端可以正确检测长属性读取的最终数据包。

服务器在发送此响应之后且在发送任何其他属性协议 PDU 之前，应使用此 ATT_MTU 值，客户端在收到此响应之后且在发送任何其他属性协议 PDU 之前，应使用此 ATT_MTU 值。

如果客户端 MTU 交换请求中的 Rx MTU 或服务器 MTU 交换响应中的 Rx MTU 错误地提供了小于默认 ATT_MTU 的数值，则不应更改 ATT_MTU，而应将 ATT_MTU 应为默认 ATT_MTU。

4.2. 代码实现

应用程序中实现 MTU 交换的步骤如下图所示。

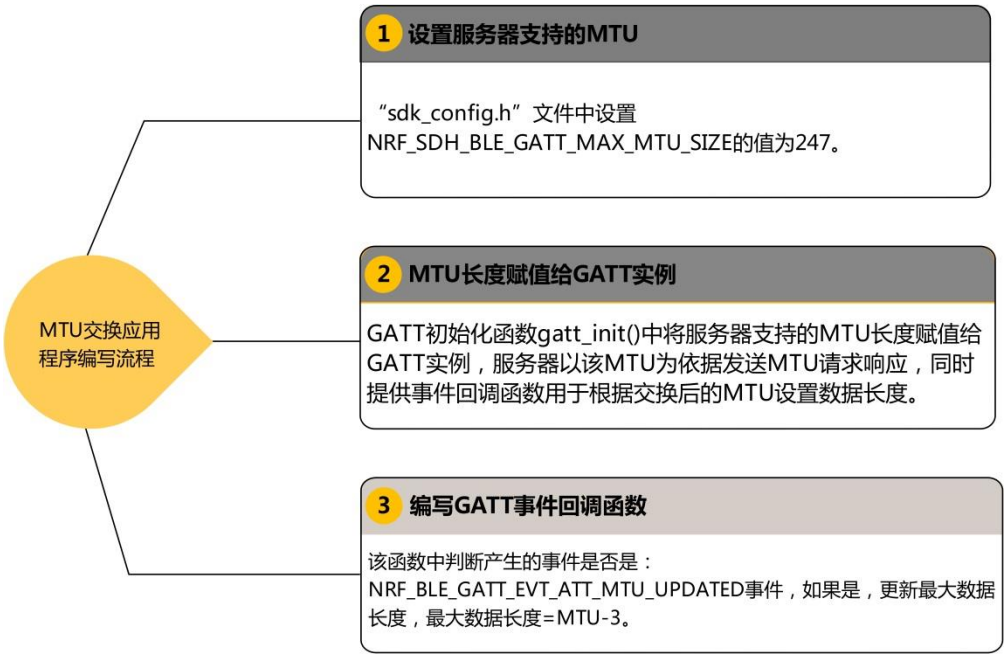


图 7-13：MTU 交换应用程序编写流程

1. 设置服务器支持的 MTU

设置 MTU 时，首先要确定所用的芯片是否支持蓝牙 4.2 BLE 或 4.2 以上版本，如 nRF52832、nRF52810、nRF52840 都支持，而 nRF51822 不支持。

在芯片硬件支持的情况下，打开“sdk_config.h”文件，将 NRF_SDH_BLE_GATT_MAX_MTU_SIZE 的值设置为 247，如下图所示。

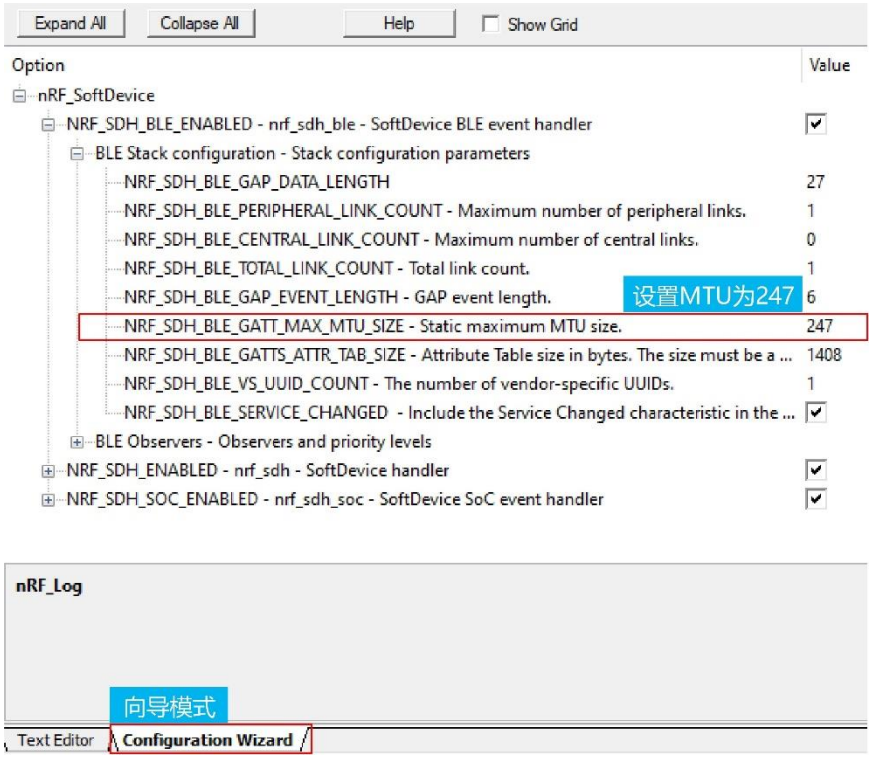


图 7-14：设置服务器支持的 MTU

2. **MTU 长度赋值给 GATT 实例**

GATT 初始化函数 gatt_init()中调用 nrf_ble_gatt_att_mtu_periph_set()函数将服务器支持的 MTU 长度赋值给 GATT 实例，代码清单如下。

代码清单：初始化 GATT 程序模块

```
static void gatt_init(void)
{
  //初始化 GATT 程序模块
  ret_code_t err_code = nrf_ble_gatt_init(&m_gatt, NULL);
  //检查函数返回的错误代码
  APP_ERROR_CHECK(err_code);
  //设置 ATT MTU 的大小,这里设置的值为 247
  err_code = nrf_ble_gatt_att_mtu_periph_set(&m_gatt, NRF_SDH_BLE_GATT_MAX_MTU_S
                                              IZE);
  APP_ERROR_CHECK(err_code);
}
```

nrf_ble_gatt_att_mtu_periph_set()函数原型如下表所示：

表 7-9：nrf_ble_gatt_att_mtu_periph_set()函数

函数原型	ret_code_t nrf_ble_gatt_att_mtu_periph_set (nrf_ble_gatt_t * p_gatt, uint16_t desired_mtu

	)
函数功能	用于作为外围设备时设置下一个连接的 ATT_MTU 大小。
参　　数	[in] p_gatt：指向 GATT 结构体。 [out] desired_mtu：请求的 ATT_MTU 大小。
返回值	NRF_SUCCESS：操作成功。 NRF_ERROR_NULL：p_gatt 为 NULL。 NRF_ERROR_INVALID_PARAM：如果 desired_mtu 的值大于 NRF_SDH_BLE_GATT_MAX_MTU_SIZE 或小于 BLE_GATT_ATT_MTU_DEFAULT。

3. **编写 GATT 事件回调函数**

当 MTU 交换完成我们需要更新数据传输的长度，这是在 GATT 事件回调函数中完成的，当 MTU 交换完成后，会产生 NRF_BLE_GATT_EVT_ATT_MTU_UPDATED 事件，此时即可更新数据长度，代码清单如下。

代码清单：GATT 事件回调函数

```
void gatt_evt_handler(nrf_ble_gatt_t * p_gatt, nrf_ble_gatt_evt_t const * p_evt)
{
    //如果是 MTU 交换事件
    if ((m_conn_handle == p_evt->conn_handle) && (p_evt->evt_id == NRF_BLE_GATT_E
        VT_ATT_MTU_UPDATED))
    {
        //设置串口透传服务的有效数据长度（MTU-opcode-handle）
        m_ble_uarts_max_data_len = p_evt->params.att_mtu_effective - OPCODE_LENGTH
                                   - HANDLE_LENGTH;
        NRF_LOG_INFO("Data len is set to 0x%X(%d)", m_ble_uarts_max_data_len,
        m_ble_uarts_max_data_len);
    }
    NRF_LOG_DEBUG("ATT MTU exchange completed. central 0x%x peripheral 0x%x",
                  p_gatt->att_mtu_desired_central,
                  p_gatt->att_mtu_desired_periph);
}
```

4.3. 下载验证

1. 解压“…\4：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 7-2：串口透传长包传输”，将解压后得到的文件夹“ble_app_uarts”拷贝合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_uarts\project\mdk5”目录下的工程“ble_app_uarts.uvproj”。
4. 切换到 Target“s132_nrf52_6.1.0_softdevice”，点击下载协议栈。

5. 切换到 Target“IK-52840DK”，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
6. 程序运行后，指示灯 D1 闪烁表示正在广播。
7. 手机上打开 nRF Connect，点击“SCAN”按钮启动扫描，扫描到设备“BLE_UartApp”后连接设备并使能通知，连接成功后指示灯 D1 常亮，如下图所示。

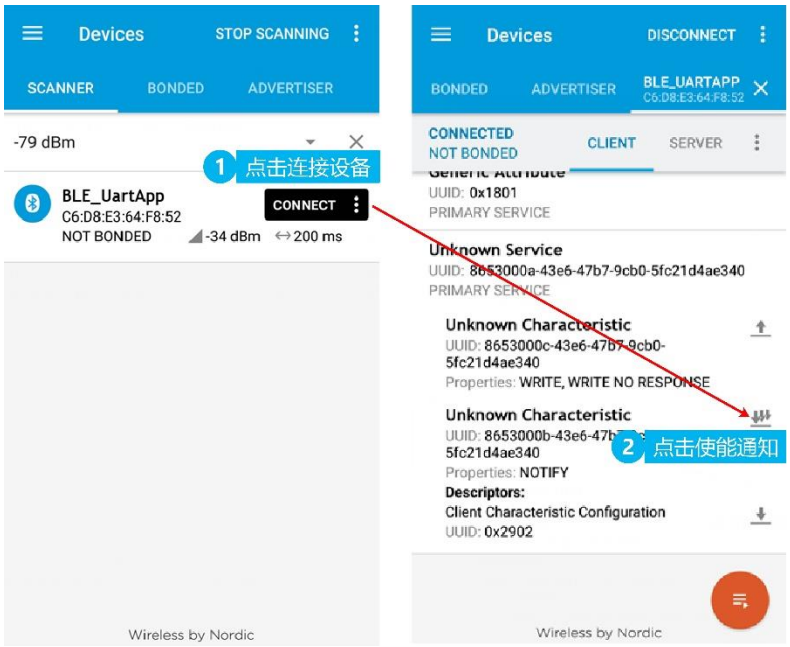


图 7-15：连接设备并使能通知

8. nRF Connect 发送数据给开发板：按照下图所示，填入 244 个字节的数据后点击“SEND”按钮发送，发送后在计算机上的串口调试助手中可以看到发送的数据，注意这里发送的是 16 进制数据，因此串口调试助手要选择“HEX 显示”。

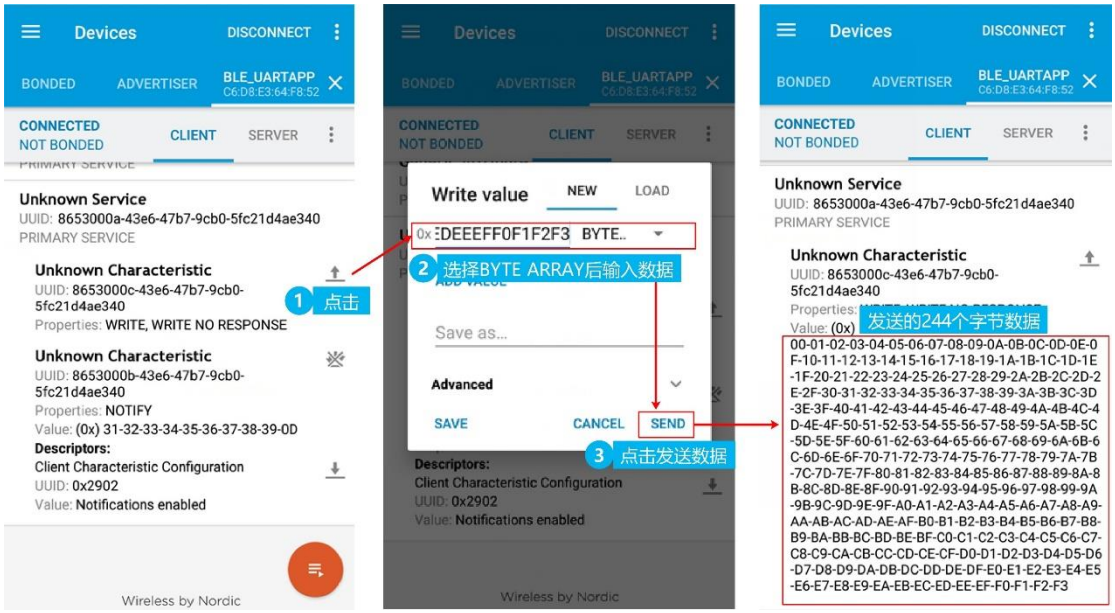


图 7-16：nRF Connect 发送数据给开发板

9. 开发板发送数据给 nRF Connect：串口调试助手发送栏填入 244 个字节的数据并勾选“发送新行”后点击发送按钮发送数据，nRF Connect 上会显示接收到的数据，如下图所示。

Devices
DISCONNECT
BONDED
ADVERTISER
BLE_UARTAPP
C6:D8:E3:64:F8:52
CONNECTED
NOT BONDED
CLIENT
SERVER
Unknown Characteristic
UUID: 8653000b-43e6-47b7-9cb0-5fc21d4ae340
Properties: NOTIFY
开发板发送的数据
Value: (0x) 30-31-32-33-34-35-36-37-38-39-30-31-32-33-34-35-36-37-38-39-30-31-32-33-34-35-36-37-38-39-

第八章：按键处理

1. 学习目的

1. 掌握如何使用 APP 按键实现按键的短按、长按以及 APP 按键的特点。
2. 掌握如何使用 BSP 按键实现按键的短按、长按和释放。

2. APP 按键原理

按键是嵌入式系统中经常使用到的输入设备，用户可以通过按键向系统发送一些操作的“指令”。

按键分为矩阵式按键和独立式按键，矩阵式按键通常将按键排列成矩阵形式，通过行/列扫描读取按键状态，适用于按键较多的场合。独立式按键使用一个 I/O 连接一个按键，电路配置灵活，软件结构相对简单，但是在按键数量较多时会占用大量的 I/O，所以独立式按键一般适用于按键较少的场合。

本章描述的按键处理针对的是独立式按键，Nordic 提供了针对按键处理的库，称为 APP 按键。APP 按键实现了按键消抖、单次键入的判定，支持多个按键检测，同时具有低功耗的特点。

2.1. 按键抖动和单次键入的判定

开发板上使用的按键为轻触按键，它属于机械弹性开关，对于机械弹性开关按键来说，当机械触点闭合、断开时，由于机械触点的弹性作用，按键开关并不会在闭合时马上稳定地接通，在断开时也不会立即断开，因而在闭合及断开的瞬间均伴随有一连串的抖动，如下图所示，因此，我们在检测按键时，需要进行按键消抖。

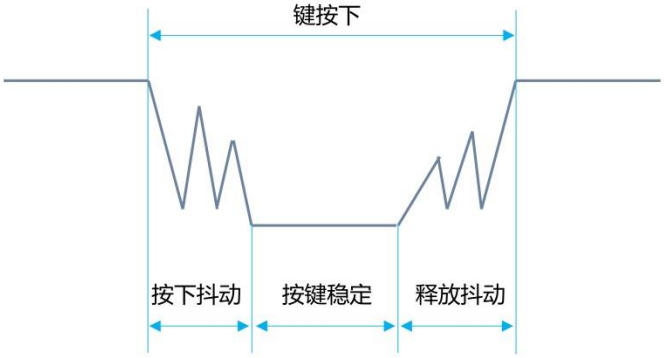


图 8-1：按键抖动

简单的按键消抖方式可以通过软件延时来实现，当检测到按键按下时，延时（5~10）ms 左右再次检测该按键的状态，如果按键的状态还是按下的状态，则认为检测到一个有效的按键操作，即完成一个单次键入的判定。这种方式需要软件延时等待，程序编写简单，但是占用 CPU 运行时间，适用在简单的程序中使用。

APP 按键通过 I/O 中断（GPIOTE 输入）和 APP 定时器实现按键处理，支持自定义按

键有效状态，即可以定义常态为高电平、按下为低电平，也可以定义常态为低电平、按下为高电平，这样对于高电平有效和低电平有效的硬件按键电路均适用，并且 APP 按键还可以检测按键的按下和释放。由此可见，对于复杂一些的应用或者对执行效率要求较高的应用，APP 按键更适合。

接下来，我们看一下 APP 按键实现的原理，下图是一个典型的单个按键检测的示意图，为了描述方便，假设按键电路是低电平有效，即常态为高电平，按键按下后为低电平，APP 定时器延时时间为 50ms。

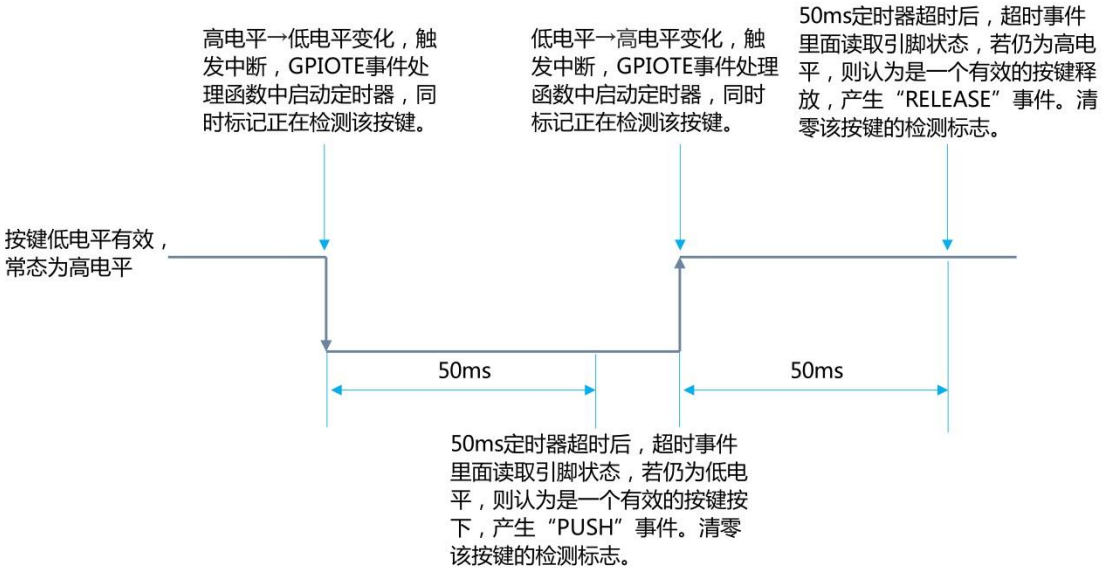


图 8-2：单个按键检测示意图

由上图我们可以看到，按键检测分为两个阶段：按下检测和释放检测。

- 按下检测：当按键状态由常态进入按下状态后，因为按键连接的 I/O 配置为任意电平变化触发事件中断的 GPIOTE 输入（可以配置为高精度或低精度），因此，会产生中断。中断处理函数中会记录下该引脚为检测状态，之后启动 APP 定时器计时，50ms 后 APP 定时器超时，在超时事件里面再次查询该引脚当前状态，若仍为低电平，则判定这是一个有效的按键按下。
- 释放检测：释放检测和按下检测的过程一样，区别是释放检测阶段按键连接的引脚电平和按下检测不同。

以上是按键检测的典型流程，实际使用时按键不可避免会有抖动之类的干扰，APPA 按键处理干扰的方式如下。

■ 持续时间小于设定的延时（50ms）

如下图所示，APP 按键模块检测到按键按下后启动 APP 定时器，如果在定时器没有超时之前再次检查到按键动作，这时候会停止 APP 定时器，并且清零该按键的检测标志，忽略此次按键检测。

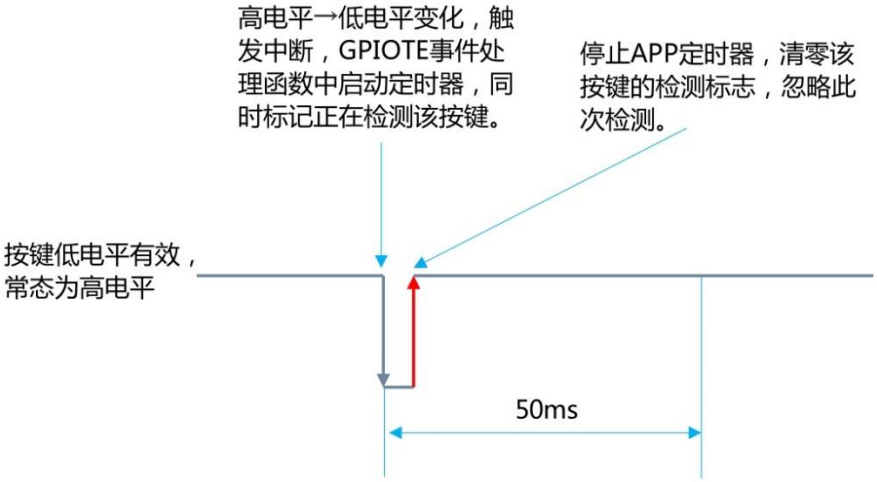


图 8-3：干扰小于设定的消抖时间

■ 50ms 内多次干扰

如果之后，再次检测到按键按下，APP 按键重新开始一个按键检测流程，即设置标志并启动 APP 定时器计时，50ms 超时后在 APP 定时器事件处理函数里面处理该按键。

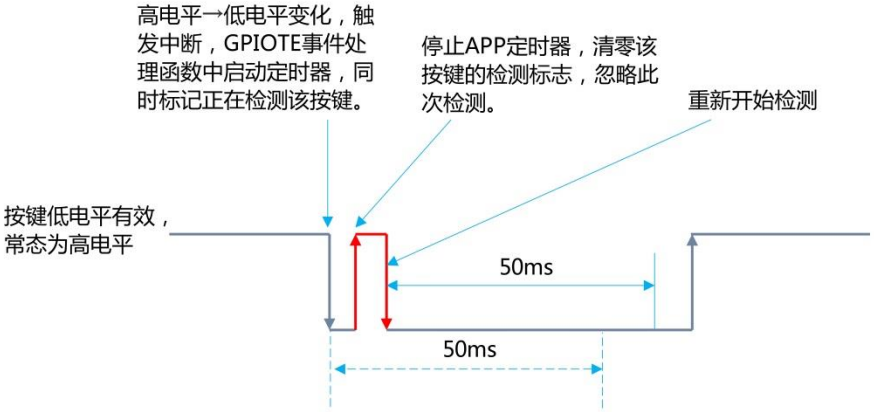


图 8-4：多次干扰的情况

2.2. 低功耗

大多数应用场合，我们对功耗的要求都比较高，应用中我们希望外设消耗的能量越低越好，APP 按键考虑了功耗问题，在低功耗方面进行了优化。

1) 通过 I/O 中断（GPIOTE 输入）检测按键状态，因此 CPU 不用去轮询按键连接的 I/O 的状态，从而让 CPU 拥有更多的休息时间。
2) I/O 中断由 GPIOTE 输入产生，我们知道 GPIOTE 可以配置为高精度和低精度，高精度使用独立的 GPIOTE 通道和高频时钟，功耗相对较大，低精度使用 GPIOTE PORT 和低频时钟，低精度不能用来跟踪高速引脚的变化，但是对于按键来说足够，低精度模式下系统进入低功耗的时候可以关闭高频时钟，因此低精度功耗更低。另外低频时钟在系统进入低功耗（不是 System Off 模式，System Off 模式下所有时钟是关闭的）后是一直开启的，所以 APP 按键使用 GPIOTE PORT 时增加的功耗很小。
3) 按键消抖和单次键入的判定是通过 APP 定时器来实现的，APP 定时器本身是低功耗的，同时对于 BLE 应用，它本身就使用了 APP 定时器，也就是无论我们有没有使用 APP

按键，在 BLE 应用中 APP 定时器都会初始化，因此，这一部分增加的功耗也很小。

2.3. APP 按键的优缺点

1. 优点

- 低功耗（使用 GPIOTE PORT），因此适用于对功耗要求高的设备。
- 中断检测，节省 CPU 运行时间。
- 具备按键消抖功能，消抖时长可设置。
- 支持多个按键检测。

2. 缺点

- APP 按键支持多个按键检测，但是因为消抖定时器只有一个，因此多个按键之间是有“干扰”的，当 APP 按键检测到一个按键按下后，启动 APP 定时器开始计时，这时候如果另外一个按键按下，APP 按键会重新开始计时，这就导致无法实现按键的并发检测。
- 仅支持独立按键，不支持矩阵按键。
- APP 按键本身没有实现长按和短按，长按和短按需要在 APP 按键的基础上封装实现。

3. 总结

APP 按键尤其适用于按键数量较少、不需要并发检测、对功耗要求高的应用。

3. APP 按键应用流程

APP 按键应用的流程如下图所示，需要特别注意的是对于裸机程序，应用程序需要去初始化低频时钟 LFCLK 和 APP 定时器，对于 BLE 应用，应用程序无需初始化这些程序模块，因为 SoftDevice 会初始化低频时钟。

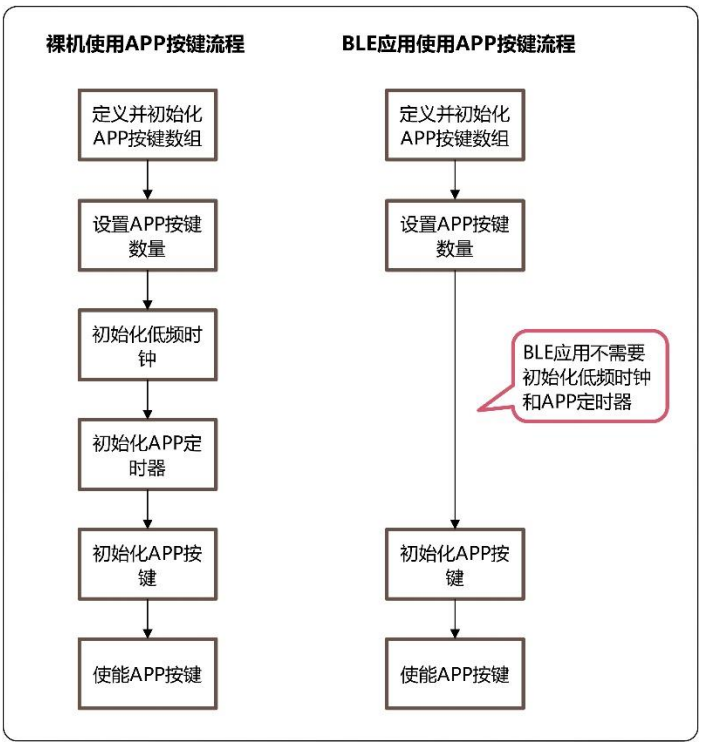


图 8-5：APP 按键应用流程

1. 定义并初始化按键数组

使用 APP 按键之前，需要先定义和初始化 APP 按键，定义 APP 按键表明了程序中使用的 APP 按键，初始化则设置了这些 APP 按键的属性。APP 按键使用结构体 app_button_cfg_t 定义，app_button_cfg_t 描述了一个 APP 按键的属性，如下表所示。

表 8-1：APP 按键属性

按键属性	描述
引脚	按键连接的引脚编号。
按键状态	可设置为高电平有效或低电平有效。
高精度	“高精度”时按键使用 GPIOTE 通道。该项仅在“sdk_config.h”中使能了“BUTTON_HIGH_ACCURACY_ENABLED”后才可以使用，通常，考虑到功耗的关系，APP 按键不会使用高精度，程序中该项被宏定义开关屏蔽，即 APP 按键默认使用“低精度”。
上拉/下拉电阻	可设置为上拉、下拉或无上/下拉。
事件回调函数	按键有效时调用的事件处理函数。

app_button_cfg_t 的原型如下：

代码清单：APP 按键配置结构体

```
1.  typedef struct
2.  {
3.      uint8_t              pin_no;            /*按键使用的引脚 */
4.      /*按键的有效状态：APP_BUTTON_ACTIVE_HIGH 或 APP_BUTTON_ACTIVE_LOW. */
5.      uint8_t              active_state;
6.  #if defined(BUTTON_HIGH_ACCURACY_ENABLED) && (BUTTON_HIGH_ACCURACY_ENABLED == 1)
7.       bool                 hi_accuracy;       /*=1：高精度*/
8.  #endif
9.      nrf_gpio_pin_pull_t  pull_cfg;          /*引脚的上拉/下拉配置 */
10.     app_button_handler_t button_handler;    /*按下按键时要调用的处理程序. */
11. } app_button_cfg_t;
```

APP 按键驱动库要求：APP 按键不能直接定义成 app_button_cfg_t 变量，而应定义成 app_button_cfg_t 结构体数组变量，这个数组称为 APP 按键数组，数组大小对应 APP 按键的数量。且 app 按键数组必须定义为 static 类型，因为按键处理模块需要保存指向该数组的指针。

一般地，在定义 APP 按键数组时会对其进行初始化，APP 按键数组定义和初始化的格式如下，可以看到，这里初始化时设置了 4 个按键属性，而 app_button_cfg_t 是有 5 个成员变量的，原因是 APP 按键极少使用“高精度”，app_button_cfg_t 中的“高精度”被宏定义开关屏蔽了。

代码清单：定义并初始化 APP 按键数组

```
1.  static const app_button_cfg_t APP 按键数组名称[APP 按键数量] =
```

```
{
    {引脚，有效状态（如 false），上/下拉(如 BUTTON_PULL)，事件处理函数},
    {引脚，false，BUTTON_PULL，事件处理函数}
};
```

APP 按键程序模块检测到有效按键后即会调用按键处理函数，按键处理函数中可以获取按键引脚和按键动作，从而可以判断出是哪个 APP 按键有效，以及该按键的动作是按下还是释放。

APP 按键处理函数的格式如下，该函数由应用程序提供，在这个函数里面，我们可以编写功能代码，针对按键的动作实现所需的功能。

代码清单：APP 按键事件处理函数格式

```
void 函数名称(uint8_t pin_no, uint8_t button_action)
{
    //是 BUTTON_1 按键的事件
    if(pin_no == BUTTON_1)
    {
        switch (button_action)
        {
            case APP_BUTTON_PUSH: //按键按下事件
                /* 添加功能代码 */
                break;
            case APP_BUTTON_RELEASE:  //按键释放事件
                /* 添加功能代码 */
                break;
            default:
                break;
        }
    }
    else if(pin_no == BUTTON_n) //是 BUTTON_n 按键的事件
    {

    }
}
```

2. 设置 APP 按键数量

APP 按键数组的大小是我们定义的 APP 按键数量，但是设置 APP 按键数量不仅如此，正如前文所述，APP 按键通常使用“低精度”即 GPIOTE PORT，因此还需要在在“sdk_config.h”中设置“GPIOTE_CONFIG_NUM_OF_LOW_POWER_EVENTS”的值，该值需设置和 APP 按键数量一致。

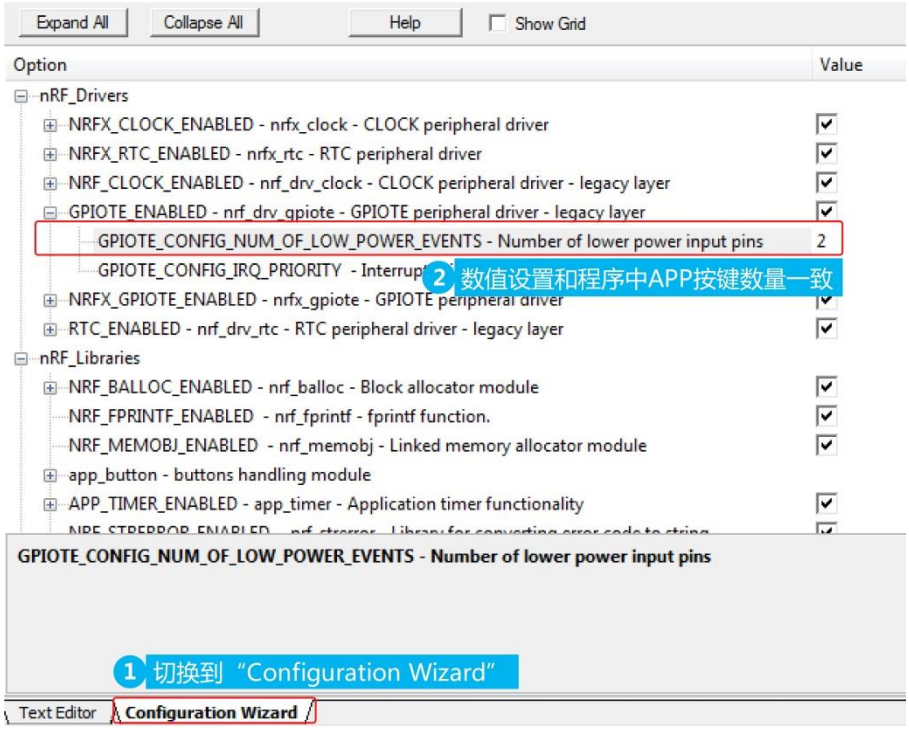


图 8-6："sdk_config.h"中设置 GPIOTE 低功耗事件数量

3. 初始化 APP 定时器程序模块

初始化低频时钟和 APP 定时器参见[第 5 章：创建定时任务]。

4. 初始化 APP 按键

APP 按键初始化的库函数是 app_button_init()函数，该函数根据 APP 按键数组中配置的按键属性初始化 APP 按键，app_button_init()函数原型如下表所示。

表 8-2：`app_button_init`()函数

函数原型	uint32_t app_button_init (app_button_cfg_t const * p_buttons, uint8_t button_count, uint32_t detection_delay)
函数功能	初始化 APP 按键。该函数将指定的引脚初始化为按键，并将 Button Handler 模块配置为 GPIOTE 用户（但不会使能按键检测，之后需要调用 app_button_enable()函数使能按键检测）。
参　　数	[in] p_buttons：指向 APP 按键数组，注意按键数组必须是 static 类型。 [in] button_count：初始化的按键数量。 [in] detection_delay：GPIOTE 事件到判定按键有效之间的时间。
返回值	初始化成功返回 NRF_SUCCESS，否则返回错误代码。

下面的代码是 APP 按键初始化的示例，可以看到初始化时需要提供 APP 按键数组和按

键数量以及有效电平持续时间，通常，会用 ARRAY_SIZE 宏计算数组 my_buttons 中的按键数量，而不是直接提供数值，以防止按键数量设置错误。

有效电平持续时间 BUTTON_DETECTION_DELAY 是用来消除按键抖动的判定时间，一般设置为 50ms。

代码清单：APP 按键初始化示例

```
ret_code_t err_code;
err_code = app_button_init(my_buttons, ARRAY_SIZE(my_buttons),
                           BUTTON_DETECTION_DELAY);
```

5. 使能 APP 按键

APP 按键初始化后并不会使能，需要调用 `app_button_enable()`函数使能，该函数会使能所有已初始化的 APP 按键，函数原型如下表所示。

表 8-3：`app_button_enable()`函数

函数原型	uint32_t app_button_enable (void)
函数功能	使能按键检测。
参　　数	无。
返回值	NRF_SUCCESS：使能按键检测成功。

代码清单：使能 APP 按键示例

```
ret_code_t err_code;
//使能 APP 按键
err_code = app_button_enable();
```

APP 按键使能后，可以通过 app_button_disable()函数禁止，该函数会禁止所有已初始化的 APP 按键，函数原型如下表所示。

表 8-4：app_button_disable()函数

函数原型	uint32_t app_button_disable (void)
函数功能	禁止按键检测。
参　　数	无。
返回值	成功禁止按键检测返回 NRF_SUCCESS，否则返回错误代码。

如果应用程序中需要查询某个 APP 按键是否按下，可以使用 app_button_is_pushed()函数，函数原型如下。

表 8-5：app_button_is_pushed()函数

函数原型	bool app_button_is_pushed (uint8_t　button_id)
函数功能	检测指定的按键当前是否按下。
参　　数	[in]button_id：需要检测的按键索引（提供给 app_button_init 的

	app_button_cfg_t 中）。
返回值	按键状态。

4. 裸机使用 APP 按键实验

本实验在开发指南上册“实验 20-1：RTC TICK 事件实现定时”的基础上修改。本例中初始化 2 个 APP 按键，他们的功能如下表所示。

表 8-6：2 个 APP 按键功能

按键	功能
按键 S1	按下后翻转指示灯 D1 的状态，释放后翻转指示灯 D2 的状态。
按键 S2	按下后翻转指示灯 D3 的状态，释放后翻转指示灯 D4 的状态。

✧ 注：本节对应的实验源码是：“实验 8-1：裸机使用 APP 按键（2 个按键）”。

4.1.1. 添加需要的文件

RTC 实验中已经加入了低频时钟所需的文件，因此，这里只需加入 APP 定时器需要的文件即可，如下表所示。

表 8-7： APP 定时器需要加入的文件

文件名	SDK 中的目录	描述
app_timer.c	..\components\libraries\timer	APP 定时器驱动文件。
app_button.c	..\components\libraries\button	APP 按键驱动文件。
nrfx_gpiote.c	..\modules\nrfx\drivers\src	GPIOTE 驱动文件。

4.1.2. 头文件引用和路径设置

1. 需要引用的头文件

因为在“main.c”文件中使用了 APP 按键和 APP 定时器程序模块，所以“main.c”文件需要引用下面的头文件。

代码清单：APP 按键需要引用的头文件

```
#include "app_timer.h"
#include "app_button.h"
```

2. 需要添加的头文件包含路径

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

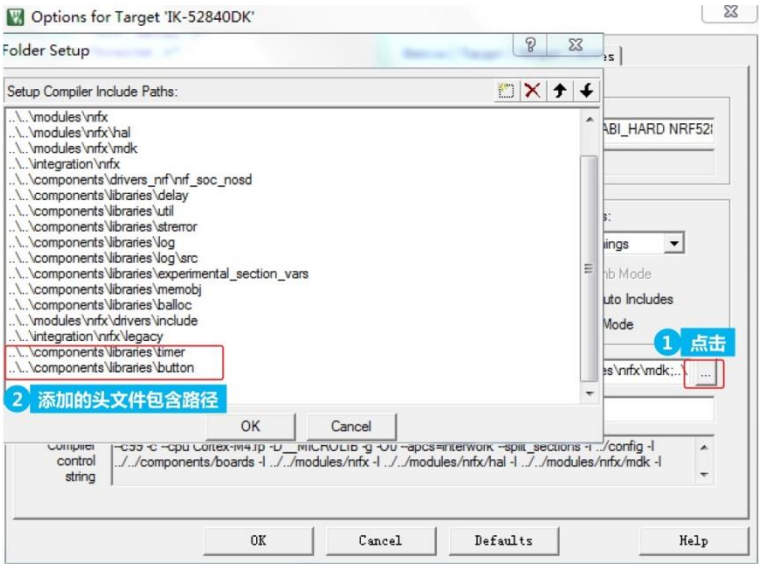


图 8-7：添加头文件包含路径

APP 定时器需要添加的头文件路径如下表：

表 8-8： 头文件包含路径

序号	路径
1	..\..\modules\nrfx\drivers\include
2	..\..\integration\nrfx\legacy
3	..\..\components\libraries\timer
4	..\..\components\libraries\button

4.1.3. 工程配置

打开“sdk_config.h”文件，加入 APP 按键需要的配置，如下图所示，编辑内容时切换到“Text Editor”进行编辑（编辑的内容参考实验的源码，因此代码很长，此处不贴出代码）本例需要添加如下 3 个程序模块的配置。

- GPIOTE：（293~398）行：本例中使用了 2 个 APP 按键，因此“GPIOTE_CONFIG_NUM_OF_LOW_POWER_EVENTS”的值应设置为 2。
- APP 按键：（540~558）行。
- APP 定时器：（559~652）行。

编辑完成后，切换到“Configuration Wizard”，即可观察到配置的具体项目：

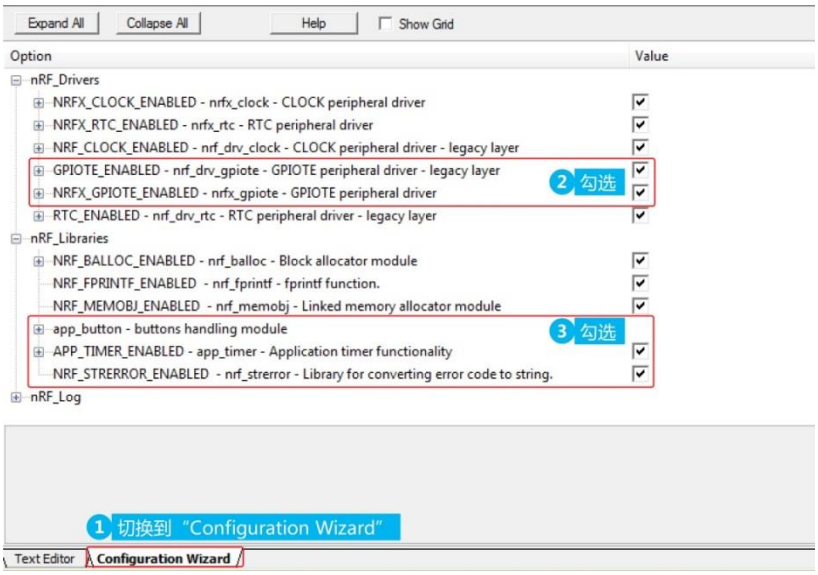


图 8-8：工程配置

4.1.4. 代码编写

使用 APP 按键时，首先定义和初始化按键数组，本例定义 2 个 APP 按键，按键属性均为低电平有效、引脚开启上拉电阻，并且 2 个 APP 按键使用同一个事件处理函数 my_button_event_handler，代码清单如下。

代码清单：定义和初始化按键数组

```
//app 按键数组必须定义为 static 类型，因为按键处理模块需要保存指向该数组的指针
static const app_button_cfg_t my_buttons[MY_BUTTONS_NUMBER] =
{
    {BUTTON_1, false, BUTTON_PULL, my_button_event_handler},
    {BUTTON_2, false, BUTTON_PULL, my_button_event_handler}
};
```

按键事件处理函数中对判断按键的引脚和动作，按键 S1（BUTTON_1）按下后翻转指示灯 D1 的状态，释放后翻转指示灯 D2 的状态；按键 S2（BUTTON_2）按下后翻转指示灯 D3 的状态，释放后翻转指示灯 D4 的状态。

代码清单：APP 按键事件处理函数

```
void my_button_event_handler(uint8_t pin_no, uint8_t button_action)
{
   if(pin_no == BUTTON_1) //是 BUTTON_1 按键
   {
      switch (button_action)//判断按键动作
      {
         case APP_BUTTON_PUSH: //按键按下事件
            nrf_gpio_pin_toggle(LED_1);//翻转指示灯 D1
            break;
         case APP_BUTTON_RELEASE:  //按键释放事件
```

```
            nrf_gpio_pin_toggle(LED_2);//翻转指示灯 D2
            break;
        default:
            break;
    }
  }
  else if(pin_no == BUTTON_2)  //是 BUTTON_2 按键
  {
     switch (button_action) //判断按键动作
     {
        case APP_BUTTON_PUSH: //按键按下事件
            nrf_gpio_pin_toggle(LED_3);//翻转指示灯 D3
            break;
        case APP_BUTTON_RELEASE: //按键释放事件
            nrf_gpio_pin_toggle(LED_4);//翻转指示灯 D4
            break;
        default:
            break;
    }
  }
}
```

APP 按键初始化和使能代码清单如下，因为 APP 按键需要使用 APP 定时器和低频时钟源，因此在 APP 按键初始化和使能之前需要先初始化低频时钟和 APP 定时器：

代码清单：初始化和使能 APP 按键

```
ret_code_t err_code;

//初始化低频时钟
lfclk_config();
//初始化 APP 定时器
err_code = app_timer_init();
APP_ERROR_CHECK(err_code);
//初始化 APP 按键
err_code = app_button_init(my_buttons, ARRAY_SIZE(my_buttons),
                           BUTTON_DETECTION_DELAY);
//检查函数返回的错误代码
APP_ERROR_CHECK(err_code);
//使能 APP 按键
err_code = app_button_enable();
//检查函数返回的错误代码
APP_ERROR_CHECK(err_code);
```

4.1.5. 硬件连接

本实验需要使用 P0.13~P0.16 驱动 LED 指示灯 D1~D4，P0.11、P0.12、P0.24 和 P0.25 检测 4 个按键，因此需要短接 LED 和按键的跳线帽，如下图所示。

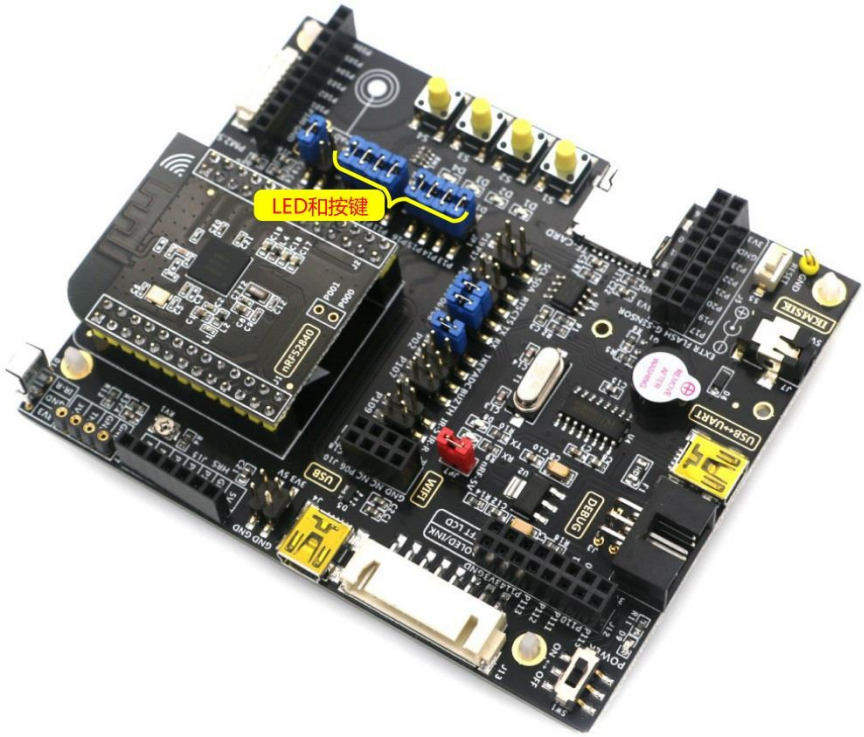


图 8-9：开发板跳线帽短接

4.1.6. 实验步骤

1. 解压“…\3：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 8-1：裸机使用 APP 按键（2 个按键）”，将解压后得到的文件夹“app_button”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\app_button\project\mdk5”目录下的工程“app_button.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，按下 S1 按键，D1 状态翻转，松开 S1 按键 D2 状态翻转，按下 S2 按键，D3 状态翻转，松开 S2 按键 D4 状态翻转。

■ **思考题：本例中使用了 2 个 APP 按键，如需要使用 4 个 APP 按键，如何修改？**

修改重点提示：APP 按键数组定义为 4 个按键，“sdk_config.h” 中设置“GPIOTE_CONFI

G_NUM_OF_LOW_POWER_EVENTS”的值为 4，实验源码见：实验 8-2：裸机使用 APP 按键（4 个按键）。

5. BLE 应用使用 APP 按键实验

本实验在“实验 7-3：串口透传长包传输-判断 notify”的基础上修改。本例中初始化 4 个 APP 按键，他们的功能如下表所示。

表 8-9：4 个 APP 按键功能

按键	功能
按键 S1	按下后点亮指示灯 D3。
按键 S2	按下后熄灭指示灯 D3。
按键 S3	按下后点亮指示灯 D4。
按键 S4	按下后熄灭指示灯 D4。

✧ 注：本节对应的实验源码是：“实验 8-3：BLE 串口透传中使用 APP 按键（4 个按键）”。

本例中，对于低频时钟、APP 定时器以及 APP 按键所需添加的文件、头文件包含路径以及 sdk_congfig.h 文件的配置（APP 定时器数量需要根据使用的 APP 按键数量配置），“实验 7-3”中已经都做好了，因此不需要再进行相关操作。

BLE 应用中使用 APP 按键需要注意以下两点：

- sdk_congfig.h 文件需要修改程序中初始化的 APP 按键数量设置“GPIOTE_CONFIG_NUM_OF_LOW_POWER_EVENTS”的值，本例中使用了 4 个 APP 按键，因此该宏的值设置为 4。
- BLE 应用中除了 APP 按键外，还经常使用 BLE 按键（APP 按键和 BLE 事件关联），BLE 按键是在 APP 按键的基础上封装的，因此如果某个按键配置为了 BLE 按键，就不能再次将其配置为 APP 按键。

5.1.1. 代码编写

BLE 应用中使用 APP 按键相对于裸机更简单一些，因为不需要初始化低频时钟和 APP 定时器，本例中我们将 APP 按键数组定义及初始化、APP 按键初始化和 APP 按键使能封装到 buttons_init()函数中，以方便使用，代码清单如下。

代码清单：初始化按键

```
static void buttons_init(void)
{
    ret_code_t err_code;

    //app 按键数组必须定义为 static 类型，因为按键处理模块需要保存指向该数组的指针
    static app_button_cfg_t buttons[] =
    {
        {BUTTON_1, APP_BUTTON_ACTIVE_LOW, BUTTON_PULL, button_event_handler},
        {BUTTON_2, APP_BUTTON_ACTIVE_LOW, BUTTON_PULL, button_event_handler},
```

```
10.       {BUTTON_3, APP_BUTTON_ACTIVE_LOW, BUTTON_PULL, button_event_handler},
11.       {BUTTON_4, APP_BUTTON_ACTIVE_LOW, BUTTON_PULL, button_event_handler},
12.     };
13.     //初始化 APP 按键
14.     err_code = app_button_init(buttons, ARRAY_SIZE(buttons),
15.                                BUTTON_DETECTION_DELAY);
16.     APP_ERROR_CHECK(err_code);
17.     //使能 APP 按键
18.     err_code = app_button_enable();
19.     APP_ERROR_CHECK(err_code);
20. }
```

和裸机程序一样，按键事件处理函数中通过“pin_no”获取按键的引脚、“button_action”获取按键的动作，以此判断当前有效的按键及按键的动作，从而执行相应的功能。

本例中，按键 S1（BUTTON_1）和 S2（BUTTON_2）按下后分别点亮和熄灭指示灯 D3，按键 S3（BUTTON_3）和 S4（BUTTON_4）按下后分别点亮和熄灭指示灯 D4，程序清单如下。

代码清单：APP 按键事件处理函数

```
1.  void button_event_handler(uint8_t pin_no, uint8_t button_action)
2.  {
3.     switch (pin_no)//判断键值
4.     {
5.        case BUTTON_1://是 BUTTON_1 按键
6.           if(button_action == APP_BUTTON_PUSH)//按键按下事件
7.           {
8.              nrf_gpio_pin_clear(LED_3);//点亮指示灯 D3
9.           }
10.          break;
11.       case BUTTON_2://是 BUTTON_2 按键
12.          if(button_action == APP_BUTTON_PUSH)//按键按下事件
13.          {
14.             nrf_gpio_pin_set(LED_3);//熄灭指示灯 D3
15.          }
16.          break;
17.       case BUTTON_3://是 BUTTON_3 按键
18.          if(button_action == APP_BUTTON_PUSH)//按键按下事件
19.          {
20.             nrf_gpio_pin_clear(LED_4);//点亮指示灯 D4
21.          }
22.          break;
23.      case BUTTON_4://是 BUTTON_4 按键
24.          if(button_action == APP_BUTTON_PUSH)//按键按下事件
25.          {
```

```
            nrf_gpio_pin_set(LED_4);//熄灭指示灯 D4
        }
        break;
    default:
        return;
  }
}
```

最后，在 main()函数中调用 buttons_init()函数初始化配置和启动 4 个 APP 按键即可。

5.1.2. 硬件连接

同“实验 8-1：裸机使用 APP 按键（2 个按键）”。

5.1.3. 实验步骤

1. 解压“…\3：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 8-3：BLE 串口透传中使用 APP 按键（4 个按键）”，将解压后得到的文件夹“ble_app_uarts”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 Keil MDK5 中执行“Project→Open Project”打开“…\ble_app_uarts\project\mdk5”目录下的工程“ble_app_uarts.uvproj”。
4. 切换到协议栈 target，下载协议栈。
5. 切换到应用程序 target，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
6. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
7. 程序运行后，指示灯 D1 闪烁表示程序运行正常，开发板正在广播，本例增加 APP 按键后，串口透传的功能和原先一样，仍可透传数据。
8. 分别按下 S1~S4 按键，可以观察到 D3 和 D4 点亮、熄灭。

6. BSP 实现长按检测

6.1. BSP 按键原理

APP 按键本身只能实现按键的按下和释放检测功能，它是不能实现长按检测的，按键长按检测需要用 BSP 实现。

BSP(Board Support Package) 是根据具体硬件编写的板级支持包，它是根据实际的硬件“定制”的，SDK 中的 BSP 是根据开发板硬件“定制”的，它主要实现的是 LED 指示灯驱动和按键检测。

对于按键检测部分，BSP 是在 APP 按键的基础上再次封装，底层使用的仍是 APP 按键。BSP 按键在 APP 按键实现的按下和释放检测的基础上增加了一个 APP 定时器用作长按计时，

从而实现长按检测。

BSP 按键使用的重点是要理解按键和动作事件之间的关联，BSP 预定义了一个 APP 按键数组 app_buttons，该按键数组包含了 8 个按键（Nordic 的 demo 板 PCA10056 支持 4 个按键，艾克姆开发板支持 5 个按键：4 个轻触按键加一个触摸按键），为了让按键和事件关联，BSP 同时根据按键的数量定义同样多的事件数组 m_events_list，按键数组和事件数组根据数组下标对应，这样就实现了按键和事件的关联。

那么动作又是如何关联的？BSP 预定义了 22 个事件，事件数组中每个元素是一个结构体变量，它包含 3 个成员“push_event”、“long_push_event”和“release_event”，这 3 个成员分别表示了短按、长按和释放动作对应的事件，每个动作都可以设置为 22 个事件中的一种，这样设置之后，按键和动作事件就完全关联起来了。当 BSP 检测到该按键动作后，会将设置的事件递交给 BSP 初始化时注册的 BSP 事件处理函数，应用程序在该函数中即可根据事件类型执行相应的功能，下图描述了按键和动作事件的关联，为了描述方便，下图以一个按键（S1 按键）为例。

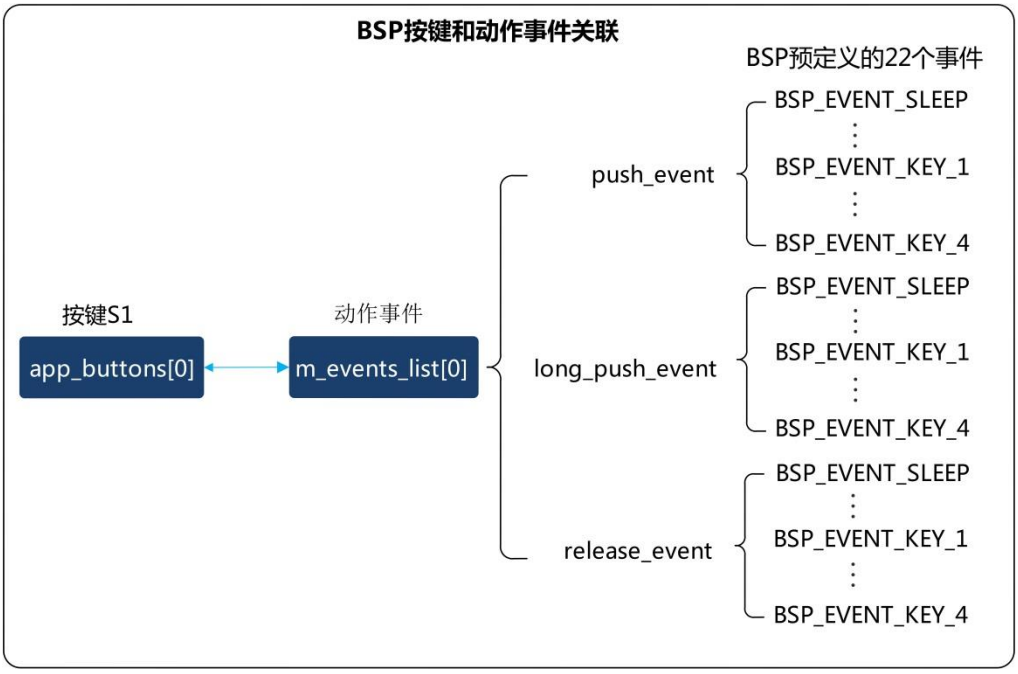


图 8-10：按键和动作事件的关联

从上图中，我们可以看到按键和事件数组是通过数组下标关联的，一个按键可以配置 3 个动作事件。所谓事件则是一个实现的功能代号（这一点一定要分清），事件和按键是关联的关系，而不是固定的关系，如我们需要的功能是长按按键 S1 后系统进入睡眠，看看下面 2 种关联事件的设置，您会很容易理解这一点

- 可以设置按下动作（long_push_event）关联事件 BSP_EVENT_SLEEP，在 BSP 事件处理函数中接收到 BSP_EVENT_SLEEP 后执行系统睡眠操作，这样在编程时看起来比较清晰。
- 可以设置按下动作（long_push_event）关联到除 BSP_EVENT_NOTHING 外的其它事件，如关联到事件 BSP_EVENT_KEY_4，只要我们在 BSP 事件处理函数中接收到 BSP_EVENT_KEY_4 后执行系统睡眠操作，同样可以实现一样的功能。但是我们肯定

不会这么做，因为这样太容易让人误解了（按键 S1 长按，事件类型是 BSP_EVENT_KEY_4，这显然是自己难为自己的做法）。由此我们也会想到：如果预定义的事件里面没有我需要的事件怎么办？这当然可以自己修改或者增加事件。

一个按键可以配置 3 个动作的事件，我们可以根据自己的实际需求进行配置，不需要产生事件的动作配置为“BSP_EVENT_NOTHING”。如应用中只需要使用长按，那么只配置长按动作的事件，短按和释放动作的事件配置为“BSP_EVENT_NOTHING”即可。

6.2. BSP 按键应用流程

BSP 按键应用的流程如下图所示。

图 8-11：BSP 按键应用的流程

1. 定义按键数组和事件数组

BSP 预定的按键数组包含 8 个按键，每个按键的定义通过宏定义开启，默认开启了 4 个按键 BSP_BUTTON_0~ BSP_BUTTON_3，用户可以根据板卡修改按键数组。

代码清单：BSP 预定义的按键

```
static const app_button_cfg_t app_buttons[BUTTONS_NUMBER] =
{
    #ifdef BSP_BUTTON_0
    {BSP_BUTTON_0, false, BUTTON_PULL, bsp_button_event_handler},
    #endif // BUTTON_0
    #ifdef BSP_BUTTON_1
    {BSP_BUTTON_1, false, BUTTON_PULL, bsp_button_event_handler},
    #endif // BUTTON_1
    #ifdef BSP_BUTTON_2
    {BSP_BUTTON_2, false, BUTTON_PULL, bsp_button_event_handler},
    #endif // BUTTON_2
    #ifdef BSP_BUTTON_3
```

```
    {BSP_BUTTON_3, false, BUTTON_PULL, bsp_button_event_handler},
    #endif // BUTTON_3
    #ifdef BSP_BUTTON_4
    {BSP_BUTTON_4, false, BUTTON_PULL, bsp_button_event_handler},
    #endif // BUTTON_4
    #ifdef BSP_BUTTON_5
    {BSP_BUTTON_5, false, BUTTON_PULL, bsp_button_event_handler},
    #endif // BUTTON_5
    #ifdef BSP_BUTTON_6
    {BSP_BUTTON_6, false, BUTTON_PULL, bsp_button_event_handler},
    #endif // BUTTON_6
    #ifdef BSP_BUTTON_7
    {BSP_BUTTON_7, false, BUTTON_PULL, bsp_button_event_handler},
    #endif // BUTTON_7
};
```

BSP 预定义一些可能用到的事件，如下表所示，用户可以根据自己的需求修改或增加事件。

表 8-10：预定义事件

事件	描述
BSP_EVENT_NOTHING	将此事件分配给一个动作意味着该动作不产生事件。
BSP_EVENT_DEFAULT	默认事件。
BSP_EVENT_CLEAR_BONDING_DATA	清空绑定信息。
BSP_EVENT_CLEAR_ALERT	清除 ALERT。
BSP_EVENT_DISCONNECT	断开 BLE 连接。
BSP_EVENT_ADVERTISING_START	开始广播。
BSP_EVENT_ADVERTISING_STOP	停止广播。
BSP_EVENT_WHITELIST_OFF	移除广播白名单。
BSP_EVENT_BOND	绑定。
BSP_EVENT_RESET	设备复位。
BSP_EVENT_SLEEP	设备进入睡眠模式。
BSP_EVENT_WAKEUP	设备从睡眠模式唤醒。
BSP_EVENT_SYSOFF	设备进入 System Off 模式（无唤醒）。
BSP_EVENT_DFU	设备进入 DFU 模式。
BSP_EVENT_KEY_0	按键 S1 PUSH 动作默认事件（仅当此按键存在时）。
BSP_EVENT_KEY_1	按键 S2 PUSH 动作默认事件（仅当此按键存在时）。
BSP_EVENT_KEY_2	按键 S3 PUSH 动作默认事件（仅当此按键存在时）。
BSP_EVENT_KEY_3	按键 S4 PUSH 动作默认事件（仅当此按键存在时）。
BSP_EVENT_KEY_4	按键 S5 PUSH 动作默认事件（仅当此按键存在时）。

BSP_EVENT_KEY_5	按键 S6 PUSH 动作默认事件（仅当此按键存在时）。
BSP_EVENT_KEY_6	按键 S7 PUSH 动作默认事件（仅当此按键存在时）。
BSP_EVENT_KEY_7	按键 S8 PUSH 动作默认事件（仅当此按键存在时）。

2. BSP 初始化

BSP 初始化的库函数是 bsp_init()，默认情况下，该函数将 4 个按键的“push_event”分配“BSP_EVENT_KEY_0~BSP_EVENT_KEY_3”事件，“long_push_event”和“release_event”分配 BSP_EVENT_NOTHING 事件，app_button_init()函数原型如下表所示。

表 8-11：bsp_init()函数

函数原型	uint32_t bsp_init (uint32_t type, bsp_event_callback_t callback)
函数功能	初始化 BSP，该函数初始化 BSP 以允许状态指示和按键检测，初始化时会给按键分配默认事件。调用该函数之前，必须初始化 APP 定时器和 GPIOTE 模块。
参　　数	[in] type：使用的外设类型。使用按键时参数为 BSP_INIT_BUTTONS，使用指示灯时参数为 BSP_INIT_LED，同时初始化按键和指示灯时参数为 BSP_INIT_LED | BSP_INIT_BUTTONS。 [in] callback：BSP 事件处理函数。
返回值	NRF_SUCCESS：BSP 初始化成功。 NRF_ERROR_INVALID_STATE：未初始化 APP 定时器。 NRF_ERROR_NO_MEM：APP 定时器数量超过了设定的最大值。 NRF_ERROR_INVALID_PARAM：没有空闲的、可以使用的 GPIOTE。 NRF_ERROR_INVALID_STATE ：按键或 GPIOTE 未初始化。

BSP 按键初始化时应用程序需要提供 BSP 事件处理函数用于接收 BSP 事件，BSP 事件处理函数根据 BSP 事件类型执行相应功能，尤其要注意的是 BSP 事件处理函数只能分辨事件类型，无法分辨事件是由哪个按键产生的，因此在为按键动作分配事件时要注意区分。如应用的需求是按键 1 的长按产生 BSP_EVENT_SLEEP 事件，那么程序中只将按键 1 的长按动作分配 BSP_EVENT_SLEEP 事件，这样只有按键 1 长按才会产生该事件。如应用的需求是按键 1 的长按和按键 2 的短按均产生 BSP_EVENT_SLEEP 事件，那么程序中将按键 1 的长按动作和按键 2 的短按动作都分配 BSP_EVENT_SLEEP 事件，这样当按键 1 长按或按键 2 短按时都会产生该事件。由此我们可以看到，BSP 事件只关心事件的处理方式使得按键处理更加方便、灵活。

BSP 事件处理函数编写时需要遵循一定的格式，如下所示。

代码清单：BSP 事件处理函数格式

```
1. void bsp_event_handler(bsp_event_t event)
```

```
{
    uint32_t err_code;
    switch (event)//判断事件类型
    {
        case BSP 事件 1:
            /* 添加功能代码 */
            break;
        case BSP 事件 2:
            /* 添加功能代码 */
            break;
        case BSP 事件 n:
            /* 添加功能代码 */
            break;
        default:
            break;
    }
}
```

3. 按键动作分配事件

很显然，按键的动作默认分配的事件不一定能满足我们的需求，因此，BSP 初始化后我们可以根据实际需求使用 bsp_event_to_button_action_assign()函数为按键动作分配事件。

表 8-12：bsp_event_to_button_action_assign()函数

函数原型	uint32_t bsp_event_to_button_action_assign(uint32_t button, bsp_button_action_t action, bsp_event_t event)
函数功能	为按键分配特定的事件。该函数允许为按键重新分配标准的事件，为了解除按键分配的事件，BSP 事件中专门提供了 BSP_EVENT_NOTHING 事件，将他分配给按键后，按键将不会再产生事件。
参　　数	[in] button：分配事件的按键 ID。 [in] action：指定按键分配事件的动作。 [in] event：分配给按键的事件。
返回值	NRF_SUCCESS：事件分配成功。 NRF_ERROR_INVALID_PARAM：指定的按键 ID 或按键动作是无效的(未定义的)。

- 示例 1：按键数组中的 0 号按键的长按动作分配 BSP_EVENT_SLEEP 事件，短按和释放动作分配 BSP_EVENT_KEY_0 事件。可以看到下面的代码中并没有为短按动作分配事件，这是因为 BSP 初始化时默认为短按动作分配了 BSP_EVENT_KEY_0 事件。

代码清单：按键动作分配事件示例

```
ret_code_t err_code;
// 按键数组中的 0 号按键长按动作分配 BSP_EVENT_SLEEP 事件
err_code = bsp_event_to_button_action_assign(0, BSP_BUTTON_ACTION_LONG_PUSH, BSP
_EVENT_KEY_0);
// 按键数组中的 0 号按键释放动作分配 BSP_EVENT_KEY_0 事件
err_code = bsp_event_to_button_action_assign(0, BSP_BUTTON_ACTION_RELEASE, BSP
_EVENT_KEY_0);
APP_ERROR_CHECK(err_code);
```

- 示例 2：按键数组中的 0 号按键的长按动作分配 BSP_EVENT_SLEEP 事件，短按和释放动作不分配事件，这时只有长按动作起作用。可以看到下面的代码中并没有为释放动作分配 BSP_EVENT_NOTHING 事件，这是因为 BSP 初始化时默认为释放动作分配了 BSP_EVENT_NOTHING 事件。

代码清单：按键动作分配事件示例

```
ret_code_t err_code;
// 按键数组中的 0 号按键长按动作分配 BSP_EVENT_SLEEP 事件
err_code = bsp_event_to_button_action_assign(0, BSP_BUTTON_ACTION_LONG_PUSH, BSP
_EVENT_KEY_0);
// 按键数组中的 0 号按键释放动作分配 BSP_EVENT_NOTHING 事件，即释放动作不产生事件
err_code = bsp_event_to_button_action_assign(0, BSP_BUTTON_ACTION_PUSH, BSP
_EVENT_NOTHING);
APP_ERROR_CHECK(err_code);
```

7. 裸机使用 BSP 按键实验

本实验在“实验 8-1：裸机使用 APP 按键（2 个按键）”的基础上修改。本例中使用 2 个 BSP 按键，他们的功能如下表所示。

表 8-13：BSP 按键功能规划

按键	动作	事件	功能
S1	短按	BSP_EVENT_LED1_ON	点亮指示灯 D1。
	长按	BSP_EVENT_LED2_ON	点亮指示灯 D2。
	释放	BSP_EVENT_LED3_ON	点亮指示灯 D3。
S2	长按	BSP_EVENT_LED_OFF	熄灭指示灯 D1、D2、D3。

✧ 注：本节对应的实验源码是：“实验 8-4：裸机使用 BSP 按键（2 个按键）”。

7.1.1. 添加需要的文件

BSP 按键是在 APP 按键的基础上实现的，在 APP 按键的基础上只需添加“bsp.c”文件即可，如下表所示。

表 8-14： APP 定时器需要加入的文件

文件名	SDK 中的目录	描述
bsp.c	..\..\components\libraries\bsp	BSP 器驱动文件。

7.1.2. 头文件引用和路径设置

1. 需要引用的头文件

因为在“main.c”文件中使用了 BSP 按键，所以“main.c”文件需要引用下面的头文件。

代码清单：BSP 需要引用的头文件

```
1. #include "bsp.h"
```

2. 需要添加的头文件包含路径

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

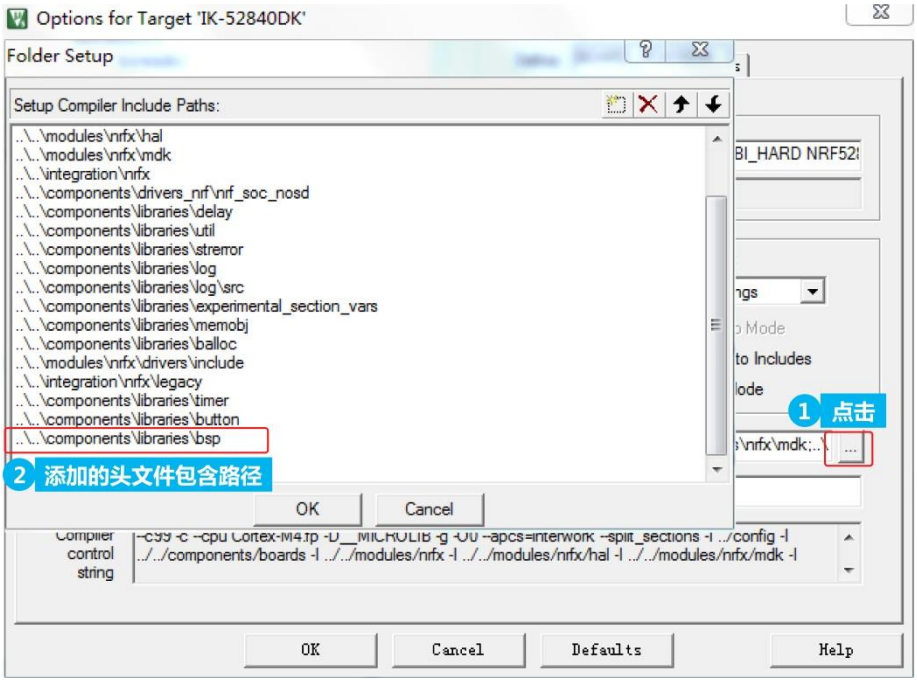


图 8-12：添加头文件包含路径

APP 定时器需要添加的头文件路径如下表：

表 8-15： 头文件包含路径

序号	路径
1	..\..\components\libraries\bsp

7.1.3. 工程配置

BSP 配置注意以下两个方面：

- BSP 在“sdk_config.h”中没有需要配置的项目，但 BSP 是以 APP 按键为基础的，因此要注意设置 APP 按键数量。
- 检查全局宏定义中是否包含了“BSP_DEFINES_ONLY”，如果包含了，需要删除该宏定义，因为“BSP_DEFINES_ONLY”宏会屏蔽掉 BSP 对按键动作的定义。

7.1.4. 代码编写

BSP 按键看似复杂，实际使用起来是比较简单的，看下面的实现代码，简单的几步即可实现按键的短按、长按、释放或他们的组合。

使用 BSP 按键时，首先定义按键数组和动作事件，BSP 定义的按键数组中默认开启了 4 个按键，本例使用了其中的 2 个按键（对应开发板的按键 S1 和 S2），因此，按键数组可以不用修改。

本例实现的功能是通过按键控制指示灯的亮灭，BSP 预定义的事件中没有描述点/关的事件，为了直观，我们加入点/关灯的事件，代码清单如下。

代码清单：增加事件

```
typedef enum
{
    BSP_EVENT_NOTHING = 0,  //将此事件分配给一个动作意味着该动作不产生事件
    BSP_EVENT_DEFAULT,      //默认事件
    BSP_EVENT_CLEAR_BONDING_DATA,  //清空绑定信息
    BSP_EVENT_CLEAR_ALERT,          //清除 ALERT
    BSP_EVENT_DISCONNECT,           //断开 BLE 连接
    BSP_EVENT_ADVERTISING_START,   //开始广播
    BSP_EVENT_ADVERTISING_STOP,    //停止广播
    BSP_EVENT_WHITELIST_OFF,       //移除广播白名单
    BSP_EVENT_BOND,     //绑定
    BSP_EVENT_RESET,    //设备复位
    BSP_EVENT_SLEEP,    //设备进入睡眠模式
    BSP_EVENT_WAKEUP,   //设备从睡眠模式唤醒
    BSP_EVENT_SYSOFF,   //设备进入 System Off 模式（无唤醒）
    BSP_EVENT_DFU,      //设备进入 DFU 模式
    BSP_EVENT_KEY_0,    //按键 1 PUSH 动作默认事件（仅当此按键存在时）
    BSP_EVENT_KEY_1,    //按键 2 PUSH 动作默认事件（仅当此按键存在时）
    BSP_EVENT_KEY_2,    //按键 3 PUSH 动作默认事件（仅当此按键存在时）
    BSP_EVENT_KEY_3,    //按键 4 PUSH 动作默认事件（仅当此按键存在时）
    BSP_EVENT_KEY_4,    //按键 5 PUSH 动作默认事件（仅当此按键存在时）
    BSP_EVENT_KEY_5,    //按键 6 PUSH 动作默认事件（仅当此按键存在时）
    BSP_EVENT_KEY_6,    //按键 7 PUSH 动作默认事件（仅当此按键存在时）
    BSP_EVENT_KEY_7,    //按键 S1 PUSH 动作默认事件（仅当此按键存在时）
    BSP_EVENT_KEY_LAST = BSP_EVENT_KEY_7,
    /* 以下为本例增加的事件 */
    BSP_EVENT_LED1_ON,  //点亮指示灯 D2
    BSP_EVENT_LED2_ON,  //点亮指示灯 D2
    BSP_EVENT_LED3_ON,  //点亮指示灯 D2
    BSP_EVENT_LED_OFF,  //点亮指示灯 D2
} bsp_event_t;
```

一般地，初始化 BSP 和按键动作分配事件会封装到一个函数里面执行，这样代码会更加清晰。

按键动作分配事件时只需按照“BSP 按键功能规划”表进行分配即可，代码清单如下。

代码清单：为按键动作分配事件

```
static void buttons_init(void)
{
    ret_code_t err_code;
    //初始化 BSP 按键
    err_code = bsp_init(BSP_INIT_BUTTONS, bsp_event_handler);
    APP_ERROR_CHECK(err_code);
    //按键数组中的 0 号按键：短按动作分配 BSP_EVENT_LEDD1_ON 事件
    err_code = bsp_event_to_button_action_assign(0, BSP_BUTTON_ACTION_PUSH,
                                                 BSP_EVENT_LED1_ON);
    APP_ERROR_CHECK(err_code);
    //按键数组中的 0 号按键：长按动作分配 BSP_EVENT_LEDD2_ON 事件
    err_code = bsp_event_to_button_action_assign(0, BSP_BUTTON_ACTION_LONG_PUSH,
                                                BSP_EVENT_LED2_ON);
    APP_ERROR_CHECK(err_code);
    //按键数组中的 0 号按键：释放动作分配 BSP_EVENT_LEDD3_ON 事件
    err_code = bsp_event_to_button_action_assign(0, BSP_BUTTON_ACTION_RELEASE,
                                                 BSP_EVENT_LED3_ON);
    APP_ERROR_CHECK(err_code);

    //按键数组中的 1 号按键：短按动作分配 BSP_EVENT_LED_OFF 事件
    err_code = bsp_event_to_button_action_assign(1, BSP_BUTTON_ACTION_LONG_PUSH,
                                                BSP_EVENT_LED_OFF);
    APP_ERROR_CHECK(err_code);
    //按键数组中的 1 号按键：释放动作分配 BSP_EVENT_NOTHING 事件，即释放动作不产生事件
    err_code = bsp_event_to_button_action_assign(1, BSP_BUTTON_ACTION_PUSH,
                                               BSP_EVENT_NOTHING);
    APP_ERROR_CHECK(err_code);
}
```

如果需要修改长按动作的时间，修改宏定义 BSP_LONG_PUSH_TIMEOUT_MS（单位 ms，默认值 1000ms）的值即可。

最后，编写 BSP 事件处理函数 bsp_event_handler()用于接收 BSP 事件，事件处理函数中判断事件类型，并根据事件类型执行点/关灯的功能。

代码清单：BSP 事件处理函数

```
void bsp_event_handler(bsp_event_t event)
{
    switch (event)//判断事件类型
```

```
    {
        case BSP_EVENT_LED1_ON://点亮 D1
            nrf_gpio_pin_clear(LED_1);
            break;
        case BSP_EVENT_LED2_ON://点亮 D2
            nrf_gpio_pin_clear(LED_2);
            break;
        case BSP_EVENT_LED3_ON://点亮 D3
            nrf_gpio_pin_clear(LED_3);
            break;
        case BSP_EVENT_LED_OFF://熄灭 D1 D2 D3
            nrf_gpio_pin_set(LED_1);
            nrf_gpio_pin_set(LED_2);
            nrf_gpio_pin_set(LED_3);
            break;
      default:
            break;
    }
}
```

7.1.5. 硬件连接

同“实验 8-1：裸机使用 APP 按键（2 个按键）”。

7.1.6. 实验步骤

1. 解压“…\3：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 8-4：裸机使用 BSP 按键（2 个按键）”，将解压后得到的文件夹“app_button”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\app_button\project\mdk5”目录下的工程“app_button.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，长按下 S1 按键，可以观察到：
   - S1 按键刚按下时的短按动作产生 BSP_EVENT_LED1_ON 事件：D1 指示灯点亮。
   - 约 1 秒后 S1 按键长按动作产生 BSP_EVENT_LED2_ON 事件：D2 指示灯点亮。
   - 松开 S1 按键时释放动作产生 BSP_EVENT_LED3_ON 事件：D3 指示灯点亮。
   - 长按 S2 按键约 1 秒后 S2 按键长按动作产生 BSP_EVENT_LED_OFF 事件：D1、D2 和 D3 指示灯熄灭。

8. BLE 应用使用 BSP 按键实验

本实验在“实验 8-3：BLE 串口透传中使用 APP 按键（4 个按键）”的基础上修改。本例中使用 4 个 BSP 按键，由于 BLE 串口透传的例子中 D1 指示灯用于指示广播和连接，本实验使用按键控制 D3 和 D4，具体功能如下表所示。

表 8-16：BSP 按键功能规划

按键	动作	事件	功能
S1	短按	BSP_EVENT_LED3_ON	点亮指示灯 D3。
S2	短按	BSP_EVENT_LED3_OFF	熄灭指示灯 D3。
S3	长按	BSP_EVENT_LED4_ON	点亮指示灯 D4。
S4	长按	BSP_EVENT_LED4_OFF	熄灭指示灯 D4。

✧ 注：本节对应的实验源码是：“实验 8-5：BLE 串口透传中使用 BSP 按键（4 个按键）”。

8.1.1. 代码编写

BLE 应用中使用 BSP 按键和裸机中的使用方法是一样的，这里只贴出按键动作分配事件的代码（这在 BSP 按键中是重点）。

先整理出“BSP 按键功能规划”表，再对照该表编写事件分配部分的代码，这样效率更高，也不容易出错。本实验根据“BSP 按键功能规划”表编写的事件分配代码如下，这里要注意 BSP 初始化时默认分配的事件。

代码清单：为按键动作分配事件

```
static void buttons_init(void)
{
    ret_code_t err_code;
    //初始化 BSP 指示灯
    err_code = bsp_init(BSP_INIT_LEDS | BSP_INIT_BUTTONS, bsp_event_handler);
    APP_ERROR_CHECK(err_code);
    //按键数组中的 0 号按键：短按动作分配 BSP_EVENT_LEDD1_ON 事件
    err_code = bsp_event_to_button_action_assign(0, APP_BUTTON_PUSH,
                                                 BSP_EVENT_LED3_ON);
    APP_ERROR_CHECK(err_code);
    //按键数组中的 0 号按键：长按动作分配 BSP_EVENT_LEDD2_ON 事件
    err_code = bsp_event_to_button_action_assign(1, APP_BUTTON_PUSH,
                                                BSP_EVENT_LED3_OFF);
    APP_ERROR_CHECK(err_code);
    //按键数组中的 0 号按键：释放动作分配 BSP_EVENT_LEDD3_ON 事件
    err_code = bsp_event_to_button_action_assign(2, BSP_BUTTON_ACTION_LONG_PUSH,
                                                 BSP_EVENT_LED4_ON);
    APP_ERROR_CHECK(err_code);
    //按键数组中的 1 号按键：释放动作分配 BSP_EVENT_NOTHING 事件，即释放动作不产生事件
```

```
    err_code = bsp_event_to_button_action_assign(2, BSP_BUTTON_ACTION_PUSH,
                                                 BSP_EVENT_NOTHING);
    APP_ERROR_CHECK(err_code);

    //按键数组中的 1 号按键：短按动作分配 BSP_EVENT_LED_OFF 事件
    err_code = bsp_event_to_button_action_assign(3, BSP_BUTTON_ACTION_LONG_PUSH,
                                                  BSP_EVENT_LED4_OFF);
    APP_ERROR_CHECK(err_code);
    //按键数组中的 1 号按键：释放动作分配 BSP_EVENT_NOTHING 事件，即释放动作不产生事件
    err_code = bsp_event_to_button_action_assign(3, BSP_BUTTON_ACTION_PUSH,
                                                 BSP_EVENT_NOTHING);
    APP_ERROR_CHECK(err_code);
}
```

8.1.2. 硬件连接

同“实验 8-1：裸机使用 APP 按键（2 个按键）”。

8.1.3. 实验步骤

1. 解压“…\3：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 8-5：BLE 串口透传中使用 BSP 按键（4 个按键）”，将解压后得到的文件夹“ble_app_uarts”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 Keil MDK5 中执行“Project→Open Project”打开“…\ble_app_uarts\project\mdk5”目录下的工程“ble_app_uarts.uvproj”。
4. 切换到协议栈 target，下载协议栈。
5. 切换到应用程序 target，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
6. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
9. 程序运行后，指示灯 D1 闪烁表示程序运行正常，开发板正在广播，本例增加 BSP 按键后，串口透传的功能和原先一样，仍可透传数据。
7. 短按 S1 按键 D3 点亮，短按 S2 按键 D3 熄灭，长按 S3 按键 D4 点亮、长按 S4 按键 D4 熄灭。

第九章：Flash 存储 FS

1. 学习目的

1. 了解可以通过哪几种方式操作片内 Flash 以及它们的区别。
2. 掌握 FS（Flash Storage）在裸机程序和使用 SoftDevice 的程序中的应用。

2. FS 概述

对于操作片内 Flash，可用的方式有下表所示的 3 种：通过芯片外设 NVMC 操作 Flash、通过 FS 程序模块和 FDS 程序模块，其中，NVMC 的方式只能在裸机下使用，FS 和 FDS 可以在裸机和使能了 SoftDevice 的应用中使用。

表 9-1：操作片内 Flash 的 3 种方法

NVMC（非易失性存储器控制器）	片内外设，NVMC 提供了写或擦除 Flash 区域和 UICR（用户信息配置寄存器）的功能，但是它仅在裸机下可以使用，在使能了 SoftDevice 的应用中不能使用。
FS（Flash Storage）	用于读、写、擦除片内 Flash 的程序模块，可用于裸机或者使用 SoftDevice 的应用。FS 是一个低层的软件库，它为 Flash 操作提供了原始的接口，自身不具备管理功能。
FDS（Flash Data Storage）	FDS 实现的是一个简单的文件系统，它极大地减少了数据损坏的风险，同时降低了开发人员和芯片 FLASH 的直接交互，FDS 提供了相应操作的 API 函数，开发人员通过这些 API 即可实现 FLASH 数据存储而不用关心太多的细节问题。FDS 内部使用的是 FS，FDS 可用于裸机或者使用 SoftDevice 的应用。

FS 程序模块定义了一个异步接口，用于通过读、写和（页）擦除操作访问片内 Flash。FS 通过对已注册事件处理程序的回调，将操作结果通知给应用程序。FS 接口的功能可以由具有不同特征的不同后端实现，当前实现的后端有两种：nrf_fstorage_nvmc 和 nrf_fstorage_sd ，这两种后端分别针对裸机和使能了 SoftDevice 的应用，也就是裸机程序后端必须使用“nrf_fstorage_nvmc”，使能了 SoftDevice 后端必须使用“nrf_fstorage_sd”，如下表所示。

表 9-2：FS 接口后端对应的使用场合

名称	外设	特点
nrf_fstorage_nvmc	片内 Flash	使用 NVMC 外设，仅能在不使用 SoftDevice 或者使用了 SoftDevice，但 SoftDevice 未使能的情况下使用。
nrf_fstorage_sd	片内 Flash	使用 SoftDevice flash API。可以在 SoftDevice 存在的任何时候使用，无论 SoftDevice 状态如何（使能/禁用）。SoftDevice 不存在时不能使用。

FS 作为低层的软件库，有它自身的优缺点，而它的一些优点同时也是它的缺点，下面

列出了 FS 的优缺点以及应用的建议。

■ **FS 优点：**

- 简单明了，可以直接在指定的空闲 Flash 空间上存储数据。
- 数据是“透明存储”的，数据的读写很直观，开发者容易理解。
- 裸机和使能了 SoftDevice 的情况下均可使用。

■ **FS 缺点：**

- 用户需要自己指定数据在 Flash 中的存储位置，因此需要用户自己确定 Flash 的空闲空间。
- 数据是“透明存储”的，没有以记录的方式存储，也没有更新和检索功能，不便于管理。
- FS 没有对 Flash 擦写做平衡管理，而 nRF52840 的页面最大只能擦写 10000 次，因此 FS 不能适应擦写频繁的应用。

■ **应用建议：**

使用 FS 时应对 Flash 的擦写次数进行评估，以保证产品的生命周期内擦写次数不超过芯片支持的最大值。建议在存储的数据条目较少、数据变动频率很低的场合下使用 FS。

3. FS 应用流程

FS 使用流程如下图所示，首先定义 FS 实例，然后初始化 FS，之后即可调用 API 实现读、写和擦除操作。

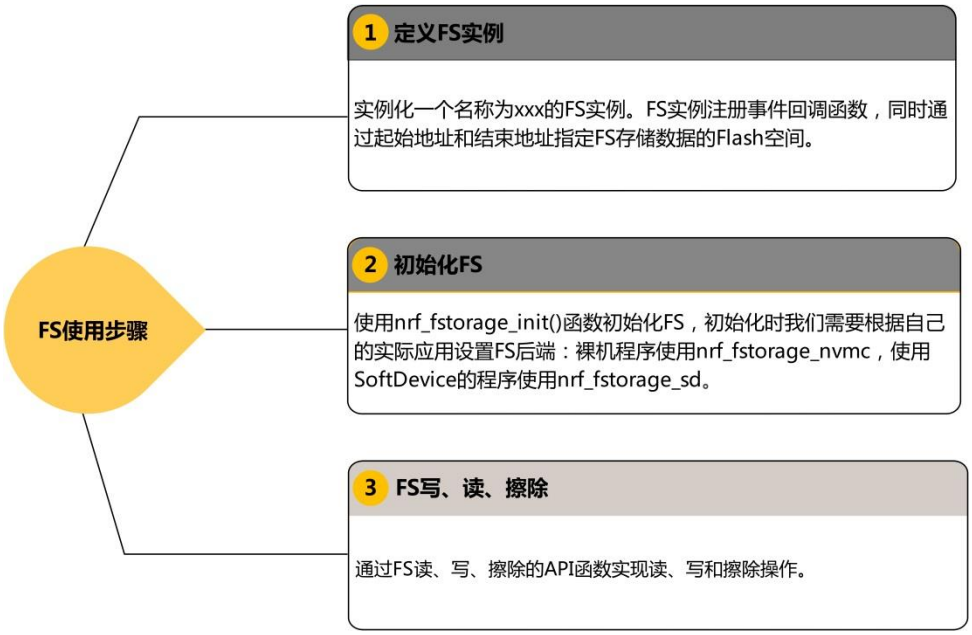


图 9-1：FS 使用流程

3.1. 定义 FS 实例

使用 FS 前，需要先定义 FS 实例，FS 使用宏 NRF_FSTORAGE_DEF 实例化。定义 FS 实例时需要提供以下四项。

- 实例名称：用以标志该实例。
- 事件回调函数：用于将 FS 操作结果通知给应用程序。

- 起始地址：和结束地址一起确定 FS 存储数据的 Flash 空间。
- 结束地址：和起始地址一起确定 FS 存储数据的 Flash 空间。

定义 FS 实例的代码清单如下：

代码清单：定义 FS 实例

```
NRF_FSTORAGE_DEF(nrf_fstorage_t 实例名称) =
{
    .evt_handler = 事件回调函数,
    .start_addr  = 起始地址,
    .end_addr    = 结束地址,
};
```

FS 事件处理函数由应用程序提供，FS 在操作后会执行回调将事件传递给事件处理函数，这样应用程序就可以及时获取 FS 执行的结果，从而可以进行相应的处理。FS 提供的事件类型如下：

代码清单：FS 定义的事件类型

```
typedef enum
{
    NRF_FSTORAGE_EVT_READ_RESULT,   //读完成事件，该事件虽然定义了，但是 FS 没有用到
    NRF_FSTORAGE_EVT_WRITE_RESULT,  //写完成事件
    NRF_FSTORAGE_EVT_ERASE_RESULT   //擦除完成事件
} nrf_fstorage_evt_id_t;
```

FS 事件处理函数中，我们可以判断事件类型执行相应处理，这里需要注意的是：NRF_FSTORAGE_EVT_READ_RESULT 事件虽然定义了，但是 FS 没有用到，也就是即使执行了 FS 读操作，该事件也不会产生。FS 事件处理函数的编写需要遵循一定的格式，如下所示。

代码清单：FS 事件处理函数

```
static void fstorage_evt_handler(nrf_fstorage_evt_t * p_evt)
{
    //FS 操作产生错误
    if (p_evt->result != NRF_SUCCESS)
    {
        //这里添加对该事件的处理代码
    }
    //判断事件类型，执行相应处理
    switch (p_evt->id)
    {
        //FS 写完成事件
        case NRF_FSTORAGE_EVT_WRITE_RESULT:
        {
```

```
            //这里添加对该事件的处理代码
        } break;
        //擦除完成事件
        case NRF_FSTORAGE_EVT_ERASE_RESULT:
        {
            //这里添加对该事件的处理代码
        } break;
        default:
          break;
    }
}
```

3.2. 初始化

FS 初始化的库函数是 nrf_fstorage_init()函数，该函数原型如下。

表 9-3：nrf_fstorage_init()函数

函数原型	ret_code_t nrf_fstorage_init (nrf_fstorage_t * p_fs, nrf_fstorage_api_t * p_api, void * p_param)
函数功能	初始化 FS。
参　　数	[in] p_fs：待初始化的 FS 实例。 [in] p_api：指向实现 Flash 存储的 API 集，使用 SoftDevice 和不使用 SoftDevice 时使用的 API 是不一样的，这里要指明使用哪个 API 集。 [in] p_param：一个可选的参数，用于传递给特定于实现的 API 调用。
返回值	NRF_SUCCESS：初始化成功。 NRF_ERROR_NULL：如果 p_fs 中的 p_fs 或 p_api 字段为 NULL。 NRF_ERROR_INTERNAL：发生了其他的错误。

调用 nrf_fstorage_init()函数初始化 FS 时，主要是设置 FS 的后端，即根据自己的应用情况（是否使用 SoftDevice）设置 API 实现。

- 裸机程序：FS 后端使用 nrf_fstorage_nvmc。
- 使用 SoftDevice 的程序：FS 后端使用 nrf_fstorage_sd。

代码清单：FS 初始化

```
ret_code_t rc;
//定义一个 FS 的 API 实现结构体 p_fs_api
nrf_fstorage_api_t * p_fs_api;
p_fs_api = &nrf_fstorage_nvmc; //裸机程序 FS 后端使用 nrf_fstorage_nvmc
```

```
//p_fs_api = &nrf_fstorage_sd; //使用 SoftDevice 的程序 FS 后端使用 nrf_fstorage_sd
//初始化 FS
rc = nrf_fstorage_init(
                &FS 实例,     //定义的 FS 实例
                p_fs_api,     //后端名称
                NULL          //可选参数，不使用时设置为 NULL
);
//用错误处理模块检查函数返回值，这里我们也可以自己来判断
APP_ERROR_CHECK(rc);
```

3.3. FS 写

FS 使用 nrf_fstorage_write()函数将数据写入到 Flash，该函数原型如下。

表 9-4：nrf_fstorage_write()函数

函数原型	ret_code_t nrf_fstorage_write (nrf_fstorage_t const * p_fs, uint32_t dest, void const * p_src, uint32_t len, void * p_param)
函数功能	该函数用于向 Flash 写入数据。将 len 字节数据从 p_src（源地址）写入 dest（目的地址）。在使用 SoftDevice 实现时，如果数据长度超过 NRF_FSTORAGE_SD_MAX_WRITE_SIZE 大小字节，则通过多次调用 sd_flash_write()来写入数据，完成后只发送一个事件。 ✧ 注意：要写入 Flash 的数据必须保存在内存中，直到操作终止并接收到事件为止。
参　　数	[in] p_fs：FS 实例。 [in] dest：Flash 中写入数据的地址。 [in] p_src：指向要写入的数据。 [in] len：数据长度（字节数）。 [in] p_param：一个可选的参数，用于传递给特定于实现的 API 调用。
返回值	NRF_SUCCESS：写入操作被接受。 NRF_ERROR_NULL：如果 p_fs 或 p_src 为 NULL。 NRF_ERROR_INVALID_STATE：FS 模块没有初始化。 NRF_ERROR_INVALID_LENGTH：写入的字节数 len 为 0 或不是编程单元的倍数或是无效的。 NRF_ERROR_INVALID_ADDR：如果写入的地址 dest 超过了 p_fs 中定义的 Flash 边界或者地址是未对齐的。

	NRF_ERROR_NO_MEM：如果没有可用的内存来接受操作。当使用 SoftDevice 实现时，该错误表明内部操作队列已满，无法接受该操作。

使用 nrf_fstorage_write()函数的示例如下，FS 写时需要指定写入的 Flash 起始地址、存储写入数据的缓存和写入的字节数。因为 FS 接口是异步的，因此待写入的数据必须先保存在内存中，然后执行写操作。

代码清单：FS 写

```
ret_code_t rc = nrf_fstorage_write(
    &FS 实例,
    Flash 地址,
    &存储写入数据的缓存,
    写入的数据长度（字节数）,
    NULL //可选参数，不使用时设置为 NULL
);
//用错误处理模块检查函数返回值，这里我们也可以自己来判断
APP_ERROR_CHECK(rc);
```

FS 写时需要注意 2 个方面：

- 写入数据的 Flash 页面必须是擦除过的页面。因为 Flash 特性是通过写只能将各个位由“1”写为“0”，而不能由“0”写为“1”，因此要通过擦除操作将所有的位恢复为“1”之后再执行写入操作。
- 数据写入到 Flash 是需要花一定时间的，并且在写数据到 Flash 时 CPU 是挂起的，这在 BLE 程序中尤其要注意，因为 CPU 挂起时间过长可能会导致 BLE 连接断开等问题，因此，在大批量数据写入到 Flash 时，我们需要根据实际情况进行处理，例如可以将数据分片，间隔性写入，从而保证 CPU 有足够的时间处理 BLE 部分的任务。

3.4. FS 读

FS 使用 nrf_fstorage_read()函数从 Flash 读取数据，该函数原型如下。

表 9-5：nrf_fstorage_read()函数

函数原型	ret_code_t nrf_fstorage_read (nrf_fstorage_t const * p_fs, uint32_t addr, void * p_dest, uint32_t len)
函数功能	该函数用于从 Flash 读出 len 字节数据，并将读取的数据拷贝到 p_dest。
参　　数	[in] p_fs：FS 实例。 [in] addr：Flash 中读出数据的地址。

	[in] p_dest：存储读出的数据的缓存。 [in] len：读出的数据长度（字节数）。
返回值	NRF_SUCCESS：操作成功。 NRF_ERROR_NULL：如果 p_fs 或 p_dest 为 NULL。 NRF_ERROR_INVALID_STATE：FS 模块没有初始化。 NRF_ERROR_INVALID_LENGTH：读出的字节数 len 为 0 或无效。 NRF_ERROR_INVALID_ADDR：如果读出的地址 dest 超过了 p_fs 中定义的 Flash 边界或者地址是未对齐的。

使用 nrf_fstorage_read()函数的示例如下，FS 读时需要指定读出数据的 Flash 起始地址、存储读出数据的缓存和读出的字节数。

代码清单：FS 读

```
ret_code_t rc = nrf_fstorage_read(
    &FS 实例,
    Flash 地址,
    &存储读取的数据的缓存,
    读取的数据长度（字节数）
);
//用错误处理模块检查函数返回值，这里我们也可以自己来判断
APP_ERROR_CHECK(rc);
```

3.5. FS 擦除

FS 使用 nrf_fstorage_erase()函数擦除指定的 Flash 页面，该函数原型如下。

表 9-6：nrf_fstorage_erase()函数

函数原型	ret_code_t nrf_fstorage_erase (nrf_fstorage_t const * p_fs, uint32_t page_addr, uint32_t len, void * p_param)
函数功能	该函数用于擦除 Flash 页面，此功能从地址 page_addr 的页面开始删除 len 页面。 必须在页面边界上启动擦除操作。
参　　数	[in] p_fs：FS 实例。 [in] page_addr：擦除的 Flash 页面地址。 [in] len：擦除的页数。 [in] p_param：传递给事件处理程序的用户定义参数（可能为 NULL）。
返回值	NRF_SUCCESS：操作成功。

	NRF_ERROR_NULL：如果 p_fs 为 NULL。 NRF_ERROR_INVALID_STATE：FS 模块没有初始化。 NRF_ERROR_INVALID_LENGTH：如果 len 为 0。 NRF_ERROR_INVALID_ADDR：如果地址 page_addr 超过了 p_fs 中定义的 Flash 边界或者地址是未对齐的。 NRF_ERROR_NO_MEM：如果没有可用的内存来接受操作。当使用 SoftDevice 实现时，该错误表明内部操作队列已满，无法接受该操作。

使用 `nrf_fstorage_erase()`函数擦除 Flash 页面的示例如下，擦除时需要指定待擦除页面的起始地址以及需要擦除的页数，如果页数大于 1，擦除操作会擦除从“Flash 地址”所在页面开始的连续页面。

代码清单：FS 擦除页面

```
ret_code_t rc = nrf_fstorage_erase(
    &FS 实例,
    Flash 地址,
    擦除的页数,
    NULL  //可选参数，不使用时设置为 NULL
);
//用错误处理模块检查函数返回值，这里我们也可以自己来判断
APP_ERROR_CHECK(rc);
```

4. 裸机 FS 读/写 Flash 实验

本实验在开发指南（上册）配套实验源码“实验 10-1：串口数据收发”的基础上修改。程序中检测 S1 和 S2 按键的状态，当 S1 按键按下后，擦除 Flash 的页 62（起始地址 0x3E000），当 S2 按键按下后，向页 62 起始地址开始连续写入 256 个字节数据，之后读出数据并通过串口输出数据。

✧ 注：本节对应的实验源码是：“实验 9-1：FS 读写-裸机”。

4.1. 添加需要的文件

裸机程序使用 FS 时需要加入的文件如下表所示。

表 9-7： 裸机程序使用 FS 需要加入的文件

文件名	SDK 中的目录	描述
nrf_nvmc.c	…\modules\nrfx\hal	NVMC 驱动文件(裸机程序需要添加)。
nrf_fstorage.c	…\components\libraries\fstorage	FS 驱动文件(裸机程序和使用 Softdevice 的程序均需要添加)。

nrf_fstorage_nvmc.c	…\components\libraries\fstorage	FS 后端 nrf_fstorage_nvmc(裸机程序需要添加)。

4.2. 头文件引用和路径设置

1. 需要引用的头文件

因为在“main.c”文件中使用了 FS 的驱动程序，所以“main.c”文件需要引用下面的头文件。

```
#include "nrf_fstorage.h"
#include "nrf_fstorage_nvmc.h"
```

2. 需要添加的头文件包含路径

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

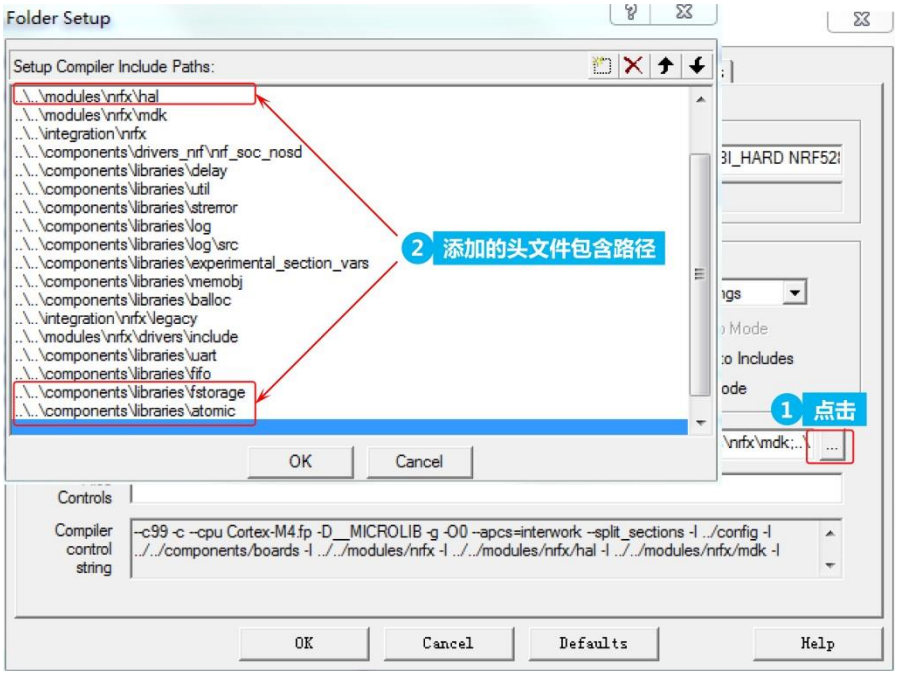


图 9-2：添加头文件包含路径

本例需要添加的头文件路径如下表：

表 9-8： 头文件包含路径

序号	路径
1	..\..\components\libraries\fstorage
2	..\..\modules\nrfx\hal
3	..\..\components\libraries\atomic

4.3. 工程配置

打开“sdk_config.h”文件，加入 FS 需要的配置，如下图所示，编辑内容时切换到“Text Editor”进行编辑（编辑的内容参考实验的源码，因此代码很长，此处不贴出代码，参见

“sdk_config.h”文件的（589~644）行），编辑完成后，切换到“Configuration Wizard”，即可观察到配置的具体项目：

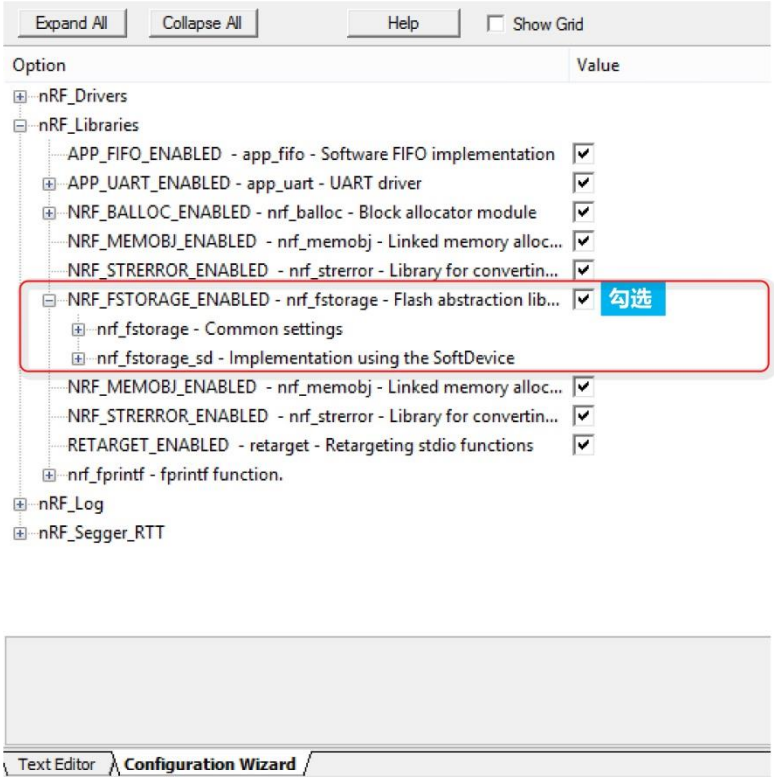


图 9-3：工程配置

4.4. 代码编写

1. 定义 FS 实例

使用 FS 时，首先要定义 FS 实例，这里我们定义一个名称为 my_fs 的 FS 实例，注册事件回调函数 `fstorage_evt_handler`。因为 FS 使用 Flash 空间为页 62，因此设置起始地址为 0x3e000，结束地址为 0x3ffff，程序清单如下。

代码清单：定义名称为 my_fs 的实例

```
//定义名称为 my_fs 的 FS 实例：定义回调函数和 FS 空间
NRF_FSTORAGE_DEF(nrf_fstorage_t my_fs) =
{
    //FS 事件回调函数
    .evt_handler = fstorage_evt_handler,

    //定义 FS 的空间（起始地址和结束地址），我们必须自己设置起始地址和结束地址
    //在调用 nrf_fstorage_init()函数初始化之前，我们也可以使用
    //nrf5_flash_end_addr_get() 函数获取可用于写入数据的 Flash 的最后一页上的结束地址
    .start_addr = 0x3e000,
    .end_addr   = 0x3ffff,
};
```

事件回调函数由应用程序提供，本例中在事件处理函数中判断了“NRF_FSTORAGE_EVT_WRITE_RESULT”和“NRF_FSTORAGE_EVT_ERASE_RESULT”事件，并打印出了事件信息。

代码清单：FS 事件处理函数

```
//FS 事件处理函数
static void fstorage_evt_handler(nrf_fstorage_evt_t * p_evt)
{
    //FS 操作错误
    if (p_evt->result != NRF_SUCCESS)
    {
      printf("--> Event received: ERROR while executing an fstorage operation.");
       return;
    }
    //判断事件类型，执行相应处理，这里只是打印出事件信息
    switch (p_evt->id)
    {
       //FS 写完成事件
       case NRF_FSTORAGE_EVT_WRITE_RESULT:
       {
          printf("--> Event received: wrote %d bytes at address 0x%x.",
                          p_evt->len, p_evt->addr);
       } break;
       //擦除完成事件
       case NRF_FSTORAGE_EVT_ERASE_RESULT:
       {
          printf("--> Event received: erased %d page from address 0x%x.",
                          p_evt->len, p_evt->addr);
       } break;

       default:
           break;
     }
}
```

2. 初始化 FS

使用 FS 时，首先要定义 FS 实例，这里我们定义一个名称为 my_fs 的 FS 实例，注册事件回调函数 `fstorage_evt_handler`。因为 FS 使用 Flash 空间为页 62，因此设置起始地址为 0x3e000，结束地址为 0x3ffff，程序清单如下。

代码清单：定义名称为 my_fs 的实例

```
//设置 API 实现，因为没有使用 SoftDevice，所以 API 后端需要使用 nrf_fstorage_nvmc
nrf_fstorage_api_t * p_fs_api;
```

```
p_fs_api = &nrf_fstorage_nvmc;
//初始化 FS
rc = nrf_fstorage_init(&my_fs, p_fs_api, NULL);
APP_ERROR_CHECK(rc);
```

3. FS 擦除操作

程序运行时检测到按键 S1 按下，执行擦除操作，擦除页 62，因为只需要擦除一次页面，所以 nrf_fstorage_erase()函数中的页数为 1，如果需要擦除页 62 和页 63，页数改为 2 即可，擦除页 62 的代码清单如下。

代码清单：FS 擦除页 62

```
//页号
#define PAGE_NUM         62
//页起始地址
#define FLASH_START_ADDR 4*1024*PAGE_NUM

//S1 按键按下：擦除页面 62（页面 62 的起始地址是：0x3e000）
if(nrf_gpio_pin_read(BUTTON_1) == 0)
{
   nrf_delay_ms(20);
   if(nrf_gpio_pin_read(BUTTON_1) == 0)
   {
      nrf_gpio_pin_clear(LED_1);//点亮 LED 指示灯 D1
      while(nrf_gpio_pin_read(BUTTON_1) == 0);//等待按键释放
      nrf_gpio_pin_set(LED_1);//熄灭 LED 指示灯 D1
      //擦除页面 62
      rc = nrf_fstorage_erase(&my_fs, FLASH_START_ADDR, 1,NULL);
      APP_ERROR_CHECK(rc);
    }
}
```

4. FS 写操作和读操作

程序运行时检测到按键 S2 按下，从页 62 的起始地址 0x3e000 依次写入 256 个字节数据，写完之后，读出数据并通过串口打印出数据，代码清单如下。

代码清单：FS 写和读

```
static uint8_t m_write_data[256];  //写缓存数组
static uint8_t m_read_data[256];  //读缓存数组

//初始化写入到 Flash 的数据
for(i=0;i<256;i++)m_write_data[i] = i+1;

//检测按键 S2 是否按下
```

```
if(nrf_gpio_pin_read(BUTTON_2) == 0)
{
   nrf_delay_ms(20);
   if(nrf_gpio_pin_read(BUTTON_2) == 0)
   {
      //点亮 LED 指示灯 D2
      nrf_gpio_pin_clear(LED_2);
      while(nrf_gpio_pin_read(BUTTON_2) == 0);//等待按键释放
      //熄灭 LED 指示灯 D2
      nrf_gpio_pin_set(LED_2);

      //将 256 个字节数据从起始地址 0x3e000 依次写入到 Flash
      rc = nrf_fstorage_write(&my_fs, FLASH_START_ADDR, &m_write_data, 256, NULL);
      APP_ERROR_CHECK(rc);
      //等待写入操作完成
      wait_for_flash_ready(&my_fs);
      //从 Flash 起始地址 0x3e000 连续读出 256 个字节数据
      rc = nrf_fstorage_read(&my_fs, FLASH_START_ADDR, &m_read_data,256);
      APP_ERROR_CHECK(rc);
      //串口打印出读出的数据
      for(i=0;i<256;i++)PutHexString(m_read_data[i]);
    }
}
```

4.5. 硬件连接

本实验需要使用 P0.13~P0.16 驱动 LED 指示灯 D1~D4，P0.06 和 P0.08 作为串口通讯引脚，按照下图所示短接跳线帽。

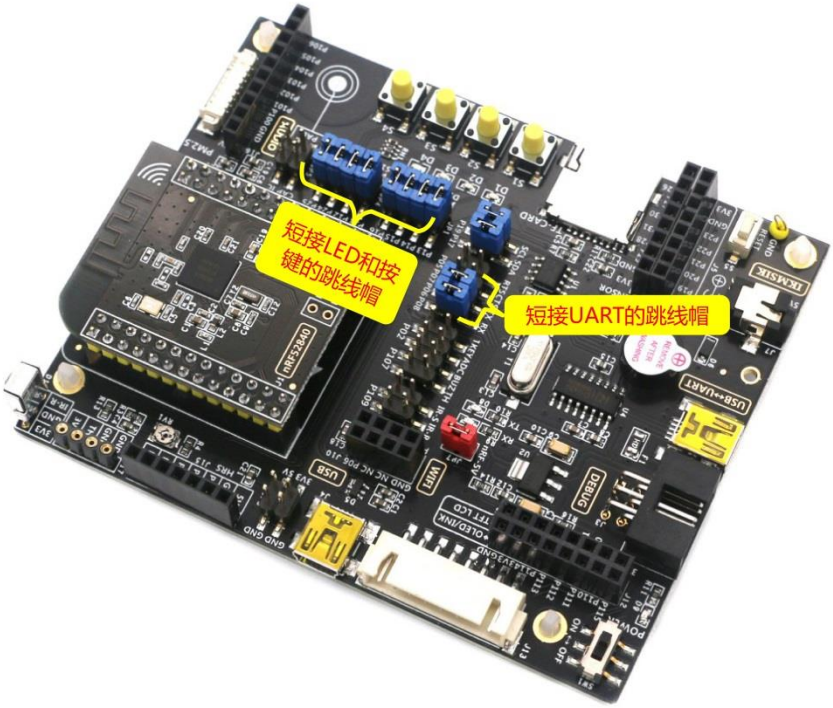


图 9-4：开发板跳线帽短接

4.6. 实验步骤

1. 解压“…\4：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 9-1：FS 读写-裸机”，将解压后得到的文件夹“flash_fstorage”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 Keil MDK5.27。
3. 在 Keil MDK5 中执行“Project→Open Project”打开“…\flash_fstorage\project\mdk5”目录下的工程“flash_fstorage.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，电脑上打开串口调试助手，按下 S1 按键擦除 nRF52840 的 Flash 页面 62，擦除执行后，串口会输出事件信息，串口调试助手中会观察到如下图所示的信息。

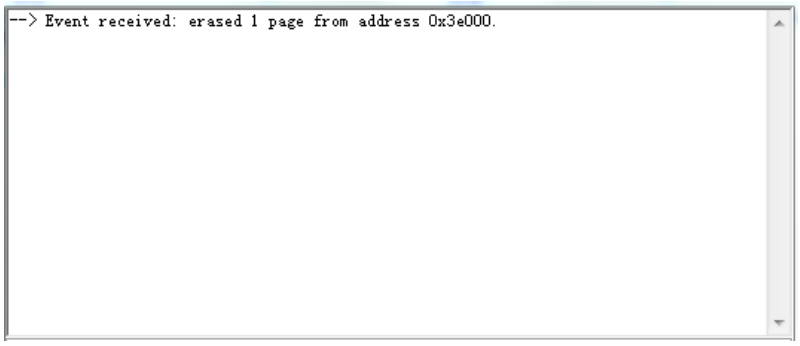


图 9-5：擦除事件信息

7. 按下 S2 按键向 Flash 页面 62 的起始地址开始连续写入 256 个字节数据，写完后读出数据通过串口输出，串口调试助手中会观察到事件信息和读出的数据，如下图所示。

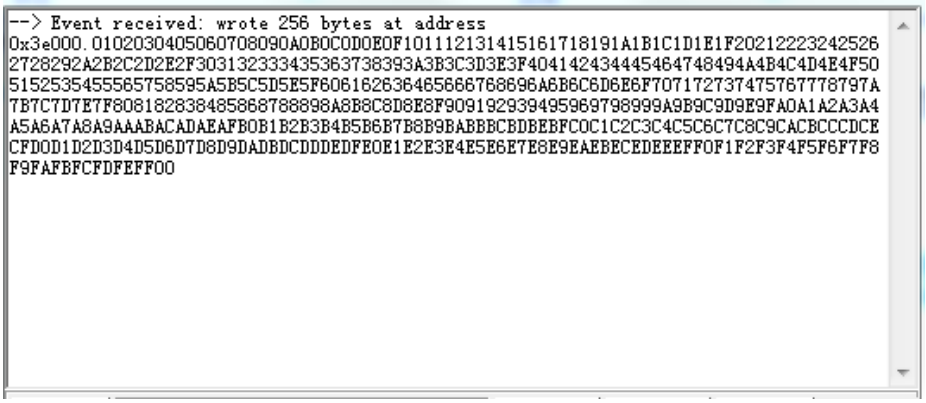


图 9-6：写事件信息和读取的数据

5. BLE 应用 FS 读/写 Flash 实验

本实验在开发指南（下册）配套实验源码“实验 7-3：串口透传长包传输-判断 notify”的基础上修改。当开发板和主机连接后，若接收到主机发送的数据，根据第一个字节是不是 0x00 执行不同的动作。

- 第一个字节不是“0x00”：将接收的主机发送的数据写入到 Flash 的页 62（起始地址 0x3E000）。
- 第一个字节为“0x00”：从 Flash 地址 0x3E000 开始连续读出 256 个字节数据，并通过串口输出。

✧ 注：本节对应的实验源码是：“实验 9-2：BLE 串口透传中使用 FS”。

5.1. 添加需要的文件

BLE 程序使用 FS 时需要加入的文件如下表所示。

表 9-9： BLE 程序使用 FS 时需要加入的文件

文件名	SDK 中的目录	描述
nrf_fstorage.c	…\components\libraries\fstorage	FS 驱动文件(裸机程序和使用 Softdevice 的程序均需要添加)。
nrf_fstorage_sd.c	…\components\libraries\fstorage	FS 后端 nrf_fstorage_sd (使用 Softdevice 的程序需要添加)。

5.2. 需要引用的头文件

本例中，对于 FS 需要加入的文件、头文件包含路径以及 sdk_congfig.h 文件的配置，“实验 7-3”中已经都做好了，因此不需要在进行操作，我们只需在“main.c”文件中引用 FS 的头文件即可。

代码清单：引用 FS 的头文件

```
//FS 需要引用的头文件
#include "nrf_fstorage.h"
#include "nrf_fstorage_sd.h"
```

5.3. 程序流程

BLE 应用中 FS 的应用流程和裸机程序是一样的，但是在程序的流程上是不一样的，主要原因是 FS 操作用时较长，并且在执行过程中会挂起 CPU，因此为了不影响 BLE 的流程，我们将擦除操作和写 Flash 操作分开，以降低连续占用 CPU 的时间。

FS 实例的定义以及初始化，参见前文的描述即可，本节我们主要描述 BLE 应用下的程序处理流程。

当主机通过 BLE 向开发板发送数据后，程序进入串口透传事件处理函数，在该函数中对接收数据的第一个字节进行判断，如果是 0x00：置位 FS 读标志，如果不是 0x00：擦除 Flash 页 62，置位 FS 写标志，流程如下图所示。

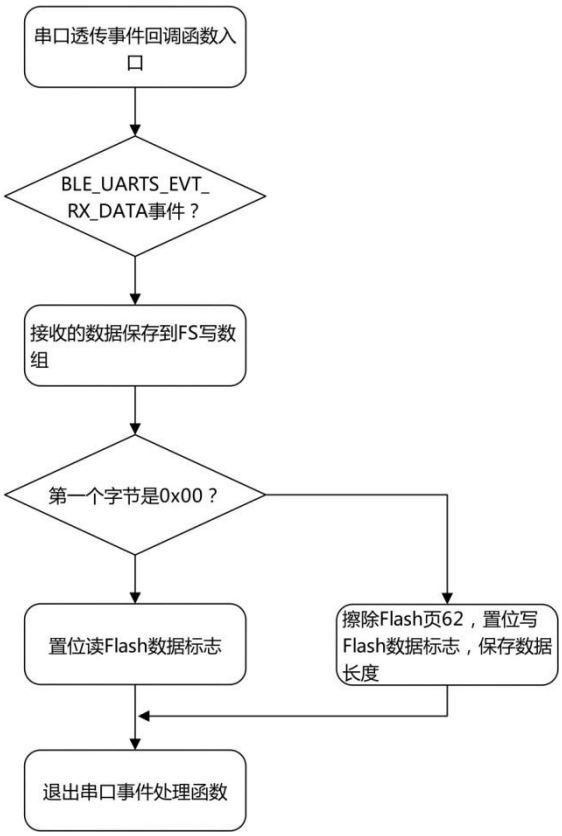


图 9-7：串口透传事件处理函数中的处理流程

主循环中判断 FS 的读和写标志，如果 FS 写标志置位：将 BLE 接收的数据写入到 Flash，如果 FS 读标志置位：读出 256 个字节数据并通过串口输出。

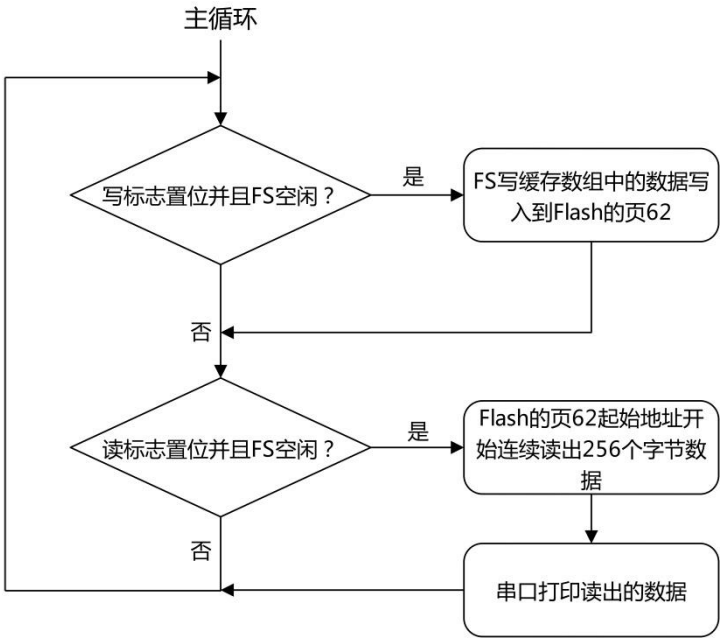


图 9-8：主循环中的处理流程

为了确保 FS 操作是在 FS 空闲时执行，我们使用一个 BOOL 变量记录 FS 的状态，在操作 FS 前（擦除和写），置位该变量，当 FS 完成操作后会进入 FS 事件处理函数中，在该函数中判断事件（擦除完成和写完成事件）并清零 FS 忙标志。

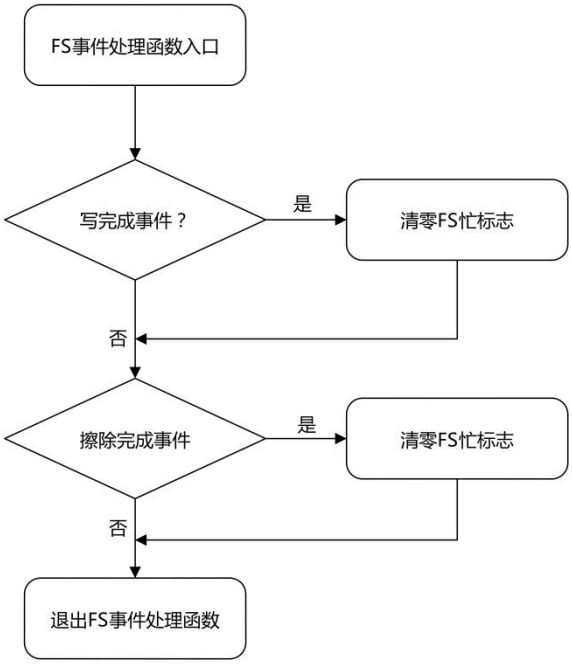


图 9-9：FS 事件处理函数的处理流程

5.4. 代码编写

按照上节所述的程序主体流程，编写代码。

本例中为了操作方便，我们声明一个结构体 my_fs_info_t，将 FS 操作相关的信息和所需的资源都放到这个结构体，程序清单如下。

代码清单：FS 操作结构体

```
typedef struct
{
    uint8_t write_data[256];//FS 写缓存数组
    uint8_t read_data[256]; //FS 读缓存数组
    uint32_t write_len;     //写入的数据长度
    bool need_write;        //写标志
    bool need_read;         //读标志
    bool busy;              //FS 忙标志
}my_fs_info_t;
//定义 FS 操作结构体变量
static my_fs_info_t my_fs_info;
```

串口透传事件回调函数中接收数据，注意 FS 擦除和写是分开执行的，这里完成写数据接收后，仅执行 FS 擦除，FS 写放到主循环执行。

代码清单：串口透传事件回调函数

```
static void uarts_data_handler(ble_uarts_evt_t * p_evt)
{
    ret_code_t rc;
    //判断事件类型:接收到新数据事件
    if (p_evt->type == BLE_UARTS_EVT_RX_DATA)
    {
        //接收的数据保存到 FS 写缓存
        for (uint32_t i = 0; i < p_evt->params.rx_data.length; i++)
        {
            my_fs_info.write_data[i] = p_evt->params.rx_data.p_data[i];
        }
        //如果第一个字节是 0x00，读 Flash 标志置位
        if(my_fs_info.write_data[0] == 0x00)
        {
            my_fs_info.need_read = true;
        }
        else //若第一个字节不是 0x00，擦除 Flash 的页 62，同时置位写 Flash 标志
        {
            //擦除页面 62
            if(my_fs_info.busy == false)
            {
                //设置标志
                my_fs_info.need_write = true;
                my_fs_info.busy = true;
                rc = nrf_fstorage_erase(&my_fs, FLASH_START_ADDR, 1,NULL);
                APP_ERROR_CHECK(rc);
                //保存待写入 Flash 的数据长度
                my_fs_info.write_len = p_evt->params.rx_data.length;
```

```
            }
        }
    }
    //判断事件类型:发送就绪事件，该事件在后面的实验会用到，当前我们在该事件中翻转指
    //示灯 D4 的状态，指示该事件的产生
    if (p_evt->type == BLE_UARTS_EVT_TX_RDY)
    {
        nrf_gpio_pin_toggle(LED_4);
    }
}
```

主循环中判断 FS 读和 FS 写标志位，在 FS 空闲的情况下根据标志执行 FS 读或 FS 写操作。

代码清单：主循环

```
while(true)
{
    if((my_fs_info.need_write == true) && (my_fs_info.busy == false))
    {
        my_fs_info.busy = true;
        my_fs_info.need_write = false;
        //将 BLE 接收的数据从起始地址 0x3e000 依次写入到 Flash
        rc = nrf_fstorage_write(&my_fs, FLASH_START_ADDR, &my_fs_info.write_data,
                                my_fs_info.write_len, NULL);
        if(rc != NRF_SUCCESS)//FS 写不成功
        {
            NRF_LOG_INFO("0x%x",rc);//打印错误代码
        }
    }
    //如果 FS 读标志置位，读出数据并通过串口打印
    if((my_fs_info.need_read == true)&&(my_fs_info.busy == false))
    {
        my_fs_info.need_read = false;
        //从 Flash 起始地址 0x3e000 连续读出 256 个字节数据
        rc = nrf_fstorage_read(&my_fs, FLASH_START_ADDR, &my_fs_info.read_data,
                                256);
        //FS 读不成功
        if(rc != NRF_SUCCESS)
        {
            NRF_LOG_INFO("0x%x",rc);//打印错误代码
        }
        else
        {
            //串口输出读取的数据
```

```
30.            for(uint16_t i=0;i<256;i++)app_uart_put(my_fs_info.read_data[i]);
31.         }
32.      }
33.      //处理挂起的 LOG 和运行电源管理
34.      idle_state_handle();
35. }
```

FS 事件处理函数中，若产生了擦除完成事件或写完成事件表示对应的操作已经完成，FS 空闲，因此在这两个事件里面清零 FS 忙标志。

代码清单：FS 事件处理函数

```
1.  //FS 事件处理函数
2.  static void fstorage_evt_handler(nrf_fstorage_evt_t * p_evt)
3.  {
4.     //FS 操作错误
5.     if (p_evt->result != NRF_SUCCESS)
6.     {
7.        NRF_LOG_INFO("--> Event received: ERROR while executing an fstorage
8.                     operation.");
9.        return;
10.    }
11.    //判断事件类型，执行相应处理，这里只是打印出事件信息
12.    switch (p_evt->id)
13.    {
14.       //FS 写完成事件
15.       case NRF_FSTORAGE_EVT_WRITE_RESULT:
16.       {
17.          my_fs_info.busy = false;
18.       } break;
19.       //擦除完成事件
20.       case NRF_FSTORAGE_EVT_ERASE_RESULT:
21.       {
22.          my_fs_info.busy = false;
23.       } break;
24.       default:
25.         break;
26.    }
27. }
```

5.5. 硬件连接

同“实验 9-1：FS 读写-裸机”。

5.6. 实验步骤

1. 解压“…\4：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 9-2：BLE 串口透传中使用 FS”，将解压后得到的文件夹“ble_app_uarts”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 Keil MDK5.27。
3. 在 Keil MDK5 中执行“Project→Open Project”打开“…\ble_app_uarts\project\mdk5”目录下的工程“ble_app_uarts.uvproj”。
4. 切换到协议栈 target，下载协议栈。
5. 切换到应用程序 target，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
6. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
7. 程序运行后，手机上打开 nRF CONNECT 并连接开发板，连接成功后，按照下图所示发送数据，开发板接收到数据后即会将数据写入到 Flash，注意发送的数据长度必须为 4 的倍数。

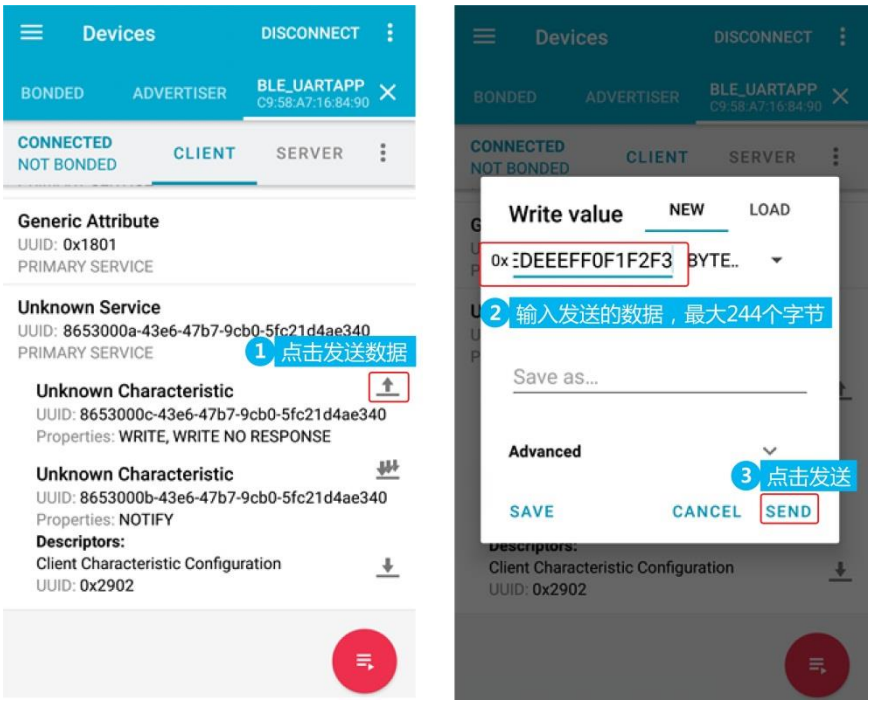


图 9-10：nRF CONNECT 发送写入到 Flash 的数据

8. 按照下图所示，发送数据“00”，开发板接收到该数据后即会读出 Flash 中存储的数据并通过串口输出。

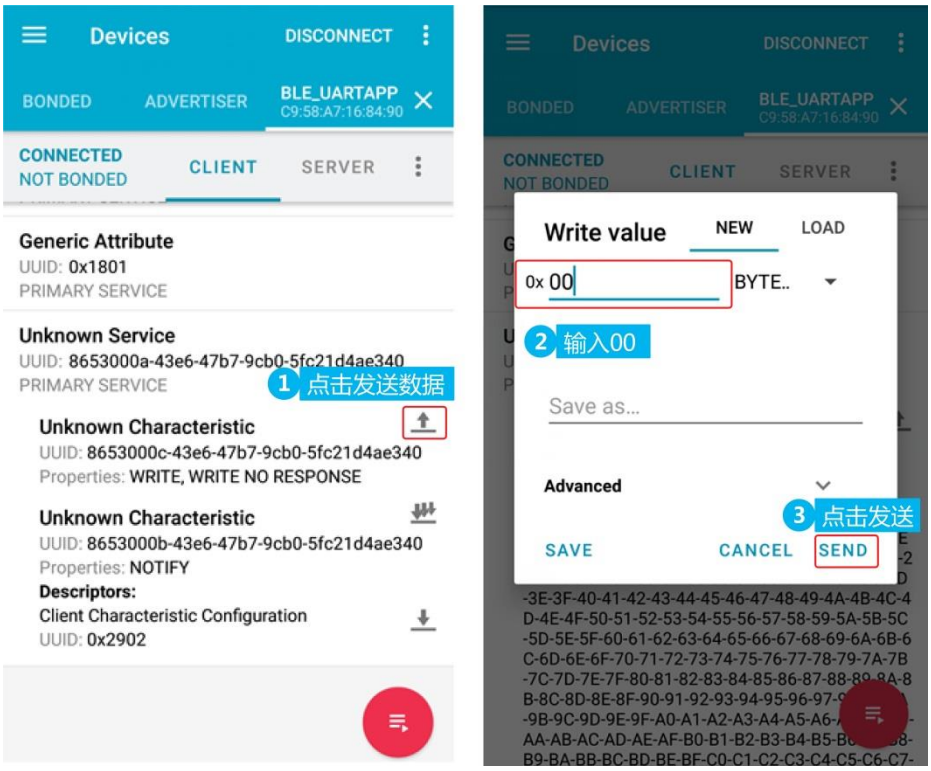


图 9-11：nRF CONNECT 发送“00”读 Flash 数据

9. 串口调试助手中可以观察到读出的数据正是 BLE 接收后写入到 Flash 的数据，因为写入 Flash 的数据是 244 个字节，读出的数据是 256 个字节，因此可以看到读出的最后 12 个字节都是“FF”。

图 9-12：串口调试助手中查看读出的数据

第十章：Flash 存储 FDS

1. 学习目的

1. 了解 FDS 的操作方式以及存储格式。
2. 掌握 FDS（Flash Data Storage）在裸机程序和使用 SoftDevice 的程序中的应用。

✧ 相对于 Flash Storage 来说，FDS 对数据的组织性和安全性更强，并且在 SDK 的 BLE 例子中使用 FDS 来存储配对相关的信息，因此，FDS 是我们学习 BLE 必须要掌握的内容，本节详细解释 FDS 程序模块的特点和应用流程，并通过实例说明如何在我们的程序中加入和使用 FDS 存储数据。

2. FDS 概述

FDS（Flash Data Storage）实现的是一个简易的文件系统，他极大地减少了数据损坏的风险，同时降低了开发人员和芯片 Flash 的直接交互，FDS 提供了相应操作的 API 函数，开发人员通过这些 API 即可实现 Flash 数据存储而不用关心太多的细节问题，开发人员可以在开发过程中把 FDS 当做一个“黑盒子”来使用。

FDS 通过文件来组织数据，文件可以包含一条或多条记录(Record)，记录中包含的是我们实际存储的数据，对记录可进行写、删除、更新、恢复等操作。FDS 程序模块的特性如下：

- 通过持续验证最小化数据损坏的风险：如系统掉电时正在进行写操作，这可能会导致数据写入不全，即写入的数据是无效数据，验证能让 FDS 辩别哪些是无效数据并能确保 FDS 提供给用户的数据是正确的。
- 可选的 CRC 校验：当打开一条记录时，CRC 校验可确保数据写入后有没有发生畸变。
- 减少 Flash 操作(更新和删除时)：FDS 不会删除 Flash 的整个页面，而是存储新数据副本，之后通过一个单字操作废除原先的旧数据，这种方式可有效地减少对 Flash 的操作。
- 基本的损耗平衡：Flash 的使用寿命是有限的，Flash 具有最大可擦写次数，一旦对 FLASH 的某个块的擦写超过这个次数，这个块就会损坏。所以，对 Flash 的擦写应该均匀的分布在块上，将块进行充分利用，减少擦写次数，提高 Flash 的使用寿命。FDS 考虑到了这一点，他提供了一个基本的损耗平衡策略，通过顺序写入和碎片收集提供偶数级的 Flash 使用。
- 无需拷贝即可获取数据，这使得访问数据时的影响与数据的长度无关。
- FDS 存储时使用可变的数据长度，最小化地使用 Flash。如我们只需存储一个字时，只需将数据长度设置为 1 即可，如果存储数据长度固定为某个值，那么当存储的数据长度小于这个值时，必然会造成 Flash 的浪费。
- 数据储存上具备充分的灵活性：记录可以由多个数据块组成，这些被储存在 Flash 中的数据块可以不连续分布在内存中，并且没有对存储数据的限制，也就是说存储的数据可

以是一些特殊字符。

FDS 在底层是通过 FS（Flash Storage）来向 Flash 中写入数据的，FDS 支持同步读和异步写。

3. FDS 功能

FDS API 公开了操作文件和记录的功能，文件由一条或多条包含实际数据的记录组成，每条记录都由一个 key 标志并通过文件 ID 关联到文件。文件实际就是一组记录的集合，文件 ID 和记录 key 都不要求必须是唯一的，并且文件可以包含具有相同 key 的多个记录，可以通过文件 ID 和记录 key 的任意组合访问记录。

例如，一个应用中使用了下面 2 个文件。

- File 1 包含 2 条记录：0x1111="Phone1", 0x2222="data: 12345"
- File 2 包含 3 条记录：0x1111="Tablet1", 0x2222="data: abcdef", 0x2222="data: 67890"

我们可以遍历文件 1 中的所有记录、所有 key 为 0x2222 的记录、文件 2 中 key 为 0x1111 的记录。

3.1. 记录的操作

1. 新建记录

将新记录写入 Flash 时，必须提供记录 key，文件 ID 和要存储的数据。我们还可以保留内存并使用生成的预订令牌以便稍后写入记录或再次取消预订，而不是立即写入记录。

写记录成功后，write 函数返回一个可用于访问记录的记录描述符，在访问记录之前，必须等待表示写入操作成功完成的事件。

2. 操作记录

要读取、更新或删除记录的内容，必须通过其描述符访问该记录。当我们首次向 Flash 写一条记录的时候，FDS 会创建并返回一个该记录的描述符，创建记录后，即可使用下面的记录查找函数来获取记录描述符，这些函数允许我们根据记录键和文件 ID 搜索记录。

- fds_record_find。
- fds_record_find_by_key。
- fds_record_find_in_file。

FDS 并没有要求记录 key 或文件 ID 是唯一的，因此，在查找记录的时候可能会找到多个匹配的记录。记录查找函数每次返回一个匹配的记录并保存当前查找的进度，返回的 Token 记录了最新匹配的记录的位置，通过 Token，在接下来的查找中就可以不必从头开始，直接从 Token 记录的位置开始即可。

3. 读记录

我们可以直接从 Flash 中读取记录的内容（存储的数据，如果需要，还可以读取元数据）。这意味着应用程序决定数据是复制，存储在 RAM 中还是就地使用。

要访问记录内容，需要打开记录以检索指向记录数据和元数据存储在 Flash 中的位置的指针。记录使用 fds_record_open()函数打开，fds_record_open()函数确保在访问记录时不会修改记录或将记录移动到 Flash 中的其他位置。切记，记录访问操作完成后，一定要关闭记录以释放对记录的锁定。

4. 更新记录

当我们更新记录时，FDS 并不是将新数据写到原有的记录中去覆盖旧数据，而是创建一个新记录，然后将原有的记录作废。更新记录时，记录更新函数会返回一个新的记录的描述符。

需要注意的是：因为 FDS 更新记录的方式是作废旧记录，创建新记录，所以，如果操作频繁，每次更新操作都会新建一条记录，占用一些 Flash 空间，时间长了就会很快占用较大的 Flash 空间，这时，就需要我们通过碎片整理函数来释放 Flash。

5. 删除记录

记录删除时并不是真正的删除记录数据和擦除 Flash 空间，而是通过设置标志将需要删除的记录作废。当一条记录被删除后，就不能再对他进行打开、读、定位等相关的操作。

记录删除后，记录所使用的 Flash 空间也不会立即释放，如果要释放作废记录使用的 Flash 空间，必须通过碎片收集函数来实现。

6. 碎片收集

FDS 依赖于碎片收集来回收已经失效的记录所使用的闪存空间，而不是立即删除记录，因此，如果在碎片收集过程中出现断电，FDS 可确保不会丢失数据。

碎片收集会需要进行很多的 Flash 操作，这就意味着碎片收集会消耗掉更多的电源能量，而在很多的应用中是使用纽扣电池供电的，电源能量是很重要的，因此，FDS 的策略是不自动进行碎片收集，什么时候进行碎片收集由应用程序决定，这样，就可以避免不必要的碎片收集操作，从而节省能量。

理想状态下，我们应该在 Flash 空间即将耗尽、BLE 活动不频繁的时候进行碎片收集，但是实际操作的时候，我们只能以需求为主。当 Flash 空间耗尽，这时候进行写操作会返回错误代码 FDS_ERR_NO_SPACE_IN_FLASH，此时，我们需要进行碎片收集并等待碎片收集完成，之后再调用写函数进行写操作。

7. FDS 对 Key 和文件 ID 的限制

记录 key 的范围是 0x0001 - 0xBFFF。0x0000 保留给系统使用，0xC000~ 0xFFFF 保留给配对管理器使用，应用程序只有在不使用配对管理器的时候才能使用这部分记录 key。

文件 ID 的范围是 0x0000 - 0xBFFF。0xFFFF 保留给系统使用，0xC000~0xFFFE 保留给配对管理模块使用，应用程序只有在不使用配对管理器的时候才能使用这部分文件 ID。

3.2. 记录的配置项目

使用记录的时候，记录相关的一些参数是需要我们去配置的，这些参数以宏定义的方式存放在 sdk_config.h 文件中，各个参数已经设置好了默认值，一般情况下，直接使用默认值即可，如果需要修改这些配置参数，修改 sdk_config.h 文件中对应的宏定义的值即可。

- FDS_OP_QUEUE_SIZE：FDS 程序模块内部使用的队列大小，如果 FDS 用户较多或者应用程序中希望一次排队更多的操作而不等待先前的操作完成，可加大队列的大小。一般情况下，如果应用程序中经常接收到错误代码 FDS_ERR_NO_SPACE_IN_QUEUES，则需要增加队列的大小。
- FDS_VIRTUAL_PAGE_SIZE：虚拟页面的大小。默认情况下虚拟页面和物理页面大小一样(NRF52832 的 Flash 一个页面大小是：1024 个字，4K 字节)，当然，我们可以增加

虚拟页面大小，以便能够存储大于物理页面的数据。

- FDS_VIRTUAL_PAGES：FDS 使用的虚拟页的数量。FDS 使用的 Flash 的大小取决与这个值和 FDS_VIRTUAL_PAGE_SIZE 的值。
- FDS_MAX_USERS：FDS 最大用户数量。所谓的 FDS 用户也就是我们通过 fds_register 函数注册的用户，一个用户对应一个回调函数，不同的程序模块使用 FDS 时需要注册各自的回调函数，FDS_MAX_USERS 这个值决定了应用程序中有多少个程序模块可以使用 FDS，如果应用程序返回错误代码 FDS_ERR_USER_LIMIT_REACHED，则需要增大 FDS_MAX_USERS 的值来允许更多的程序模块使用 FDS。
- FDS_CRC_ENABLED：可选的配置项，如果使能，FDS 打开记录读取数据时会对记录进行校验。另外，可以启用检查以确保在将数据写入记录和实际写入数据这个时间段内，数据没有改变。启用 FDS_CRC_ENABLED 后，内存占用会略有增加。
- FDS_CRC_CHECK_ON_WRITE：可选的配置项，如果使能，FDS 写操作时会进行校验。使能该配置时 FDS_CRC_CHECK_ON_READ 必须使能。

另外，FDS 还可以设置下面的编译标志：

- FDS_THREADS：如果置位，启用代码中的一些临界区。启用 FDS_THREADS 会增加代码大小。另外，因为临界区会关闭中断，因此在启用此标志之前，请确保您了解关闭中断的含义并确保在应用程序中使用适当的编程模型。

3.3. 存储格式

FDS 以记录的方式将数据存储到 Flash，这些记录被分组到文件中。在大多数用例中，我们无需详细了解 FDS 如何将数据存储在 Flash 中。本节描述了 FDS 使用的数据格式的相关信息，但如果您对详细信息不感兴趣，则可以跳过此部分。

1. 存储格式

记录由记录头（记录元数据）和实际内容组成，记录头用来标志记录，记录内容是实际存储的数据，它们按照写入的顺序连续存储在 Flash 中。

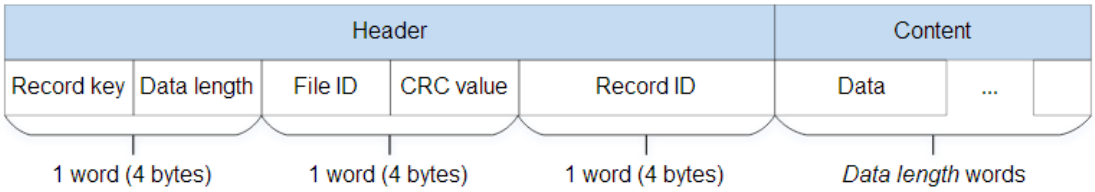


图 10-1：记录的存储格式

记录头由 3 个字（12 个字节）组成，各个部分的描述如下表所示：

表 10-1：记录头说明

域	大小(位)	描述
Record key	16	key 用来查找记录，注意 FDS_RECORD_KEY_DIRTY (0x0000) 是系统保留用来标志无效记录的，所以 key 的值不能设置的和它他相同。

Data length	16	记录中存储的数据长度。
File ID	16	记录关联的文件的 ID。数值 FDS_FILE_ID_INVALID (0xFFFF)被系统用来识别未正确写入的记录，应用程序不能使用该值作为文件 ID。
CRC value	16	整条记录的 CRC 校验值(CRC 校验可以通过设置宏定义 FDS_CRC_ENABLED 的值来启用或关闭)。
Record ID	32	记录的唯一 ID。

FDS 将记录头写入 Flash 时，首先写入记录 key 和数据长度，接着写入记录 ID，最后写入文件 ID 和 CRC 校验值，并完成成功的写入操作。检索记录的时候，FDS 会忽略掉所有记录头中第二个字没有写入的记录，因为缺少 key 意味着这条记录没有写入完整。

2. 最大存储数据长度

记录的最大长度取决于以下三个值：

- 虚拟 Flash 页面的大小(sdk_config.h 中定义的 FDS_VIRTUAL_PAGE_SIZE)。
- 页面标签(page tag)的大小(2 个字)。
- 记录头的大小(3 个字)。

sdk_config.h 中默认的设置是虚拟 Flash 的页面的大小和物理页面大小一样(对于 nRF52840 来说是 1024 个字，即 4096 个字节)。因此，对于默认值来说去掉页面标签和记录头之后，最大数据长度是 1019 个字。

✧ 如果要存储更大的数据，可以修改虚拟页面的大小，或者使用 Flash Storage 程序模块来存储数据。

3. 页面标签（Page tag）

FDS 使用的每个虚拟页面都标有一个页面标记，系统使用该标记来存储有关该页面的信息。两个字长的页面标签包含了有关页面用途（数据存储或垃圾收集）以及页面上安装的文件系统版本的信息。

FDS 使用下表所示的页面标签。

表 10-2：页面标签

字 0	字 1	描述
0xDEADC0DE	0xF11E01FF	碎片收集期间用于交换的页面。
0xDEADC0DE	0xF11E01FE	用于存储数据的页面。

✧ 页面标签在 FDS 首次初始化时写入，仅在碎片收集时更新。

4. FDS 应用流程

FDS 使用流程如下图所示，首先注册 FDS 事件句柄，然后初始化 FDS，之后即可调用 API 执行记录读、写、更新、删除和碎片整理等操作。在这里面尤其要注意的是：FDS 的初始化也是异步的，必须要先注册事件句柄之后才能初始化，初始化执行后要根据事件句柄中

接收的初始化完成事件判断初始化是否成功，只有初始化成功后，才能执行 FDS 相关操作。

图 10-2：FDS 使用流程

4.1. 注册 FDS 事件句柄

FDS 使用 fds_register()函数注册事件句柄，该函数原型如下。

表 10-3：fds_register()函数

函数原型	ret_code_t fds_register(fds_cb_t cb)
函数功能	注册 FDS 事件句柄。可注册的事件句柄的最大数量由“sdk_config.h”文件中的宏定义 FDS_MAX_USERS 确定，如需要修改可注册的事件句柄的最大数量，修改这个值即可。
参　　数	[in]cb：事件处理函数。
返回值	NRF_SUCCESS：事件句柄注册成功。 FDS_ERR_USER_LIMIT_REACHED：已注册的事件句柄达到最大数量。

FDS 事件处理函数由应用程序提供，FDS 在操作后会执行回调将事件传递给事件处理函数，这样应用程序就可以及时获取 FDS 执行的结果，从而可以进行相应的处理。FDS 提供了如下的 6 种事件类型：

代码清单：FDS 定义的事件类型

```
typedef enum
{
    FDS_EVT_INIT,        //FDS 初始化事件
    FDS_EVT_WRITE,       //FDS 写记录事件
    FDS_EVT_UPDATE,      //FDS 记录更新事件
    FDS_EVT_DEL_RECORD,  //FDS 记录删除事件
```

```
7.      FDS_EVT_DEL_FILE,    //FDS 文件删除事件
8.      FDS_EVT_GC           //FDS 碎片整理事件
9.  } fds_evt_id_t;
```

FDS 事件处理函数中，我们可以根据实际应用情况判断事件类型并执行相应处理，如程序里面只执行了 FDS 初始化和记录的读和写，那么只需判断 FDS 的初始化事件、读记录和写记录事件即可。FDS 事件处理函数的编写需要遵循一定的格式，如下所示。

代码清单：FDS 事件处理函数格式

```
1.  static void fds_evt_handler(fds_evt_t const * p_evt)
2.  {
3.     //判断事件类型
4.     switch (p_evt->id)
5.     {
6.        case FDS_EVT_INIT://FDS 初始化事件
7.        if (p_evt->result == NRF_SUCCESS)  //初始化成功
8.        {
9.           //添加功能代码
10.       }
11.       break;
12.       case FDS_EVT_WRITE:  //FDS 写记录事件
13.       {
14.          if (p_evt->result == NRF_SUCCESS) //写记录成功
15.          {
16.             //添加功能代码
17.          }
18.       } break;
19.       case FDS_EVT_UPDATE: //FDS 更新记录事件
20.       {
21.          //记录更新成功
22.          if (p_evt->result == NRF_SUCCESS)
23.          {
24.             //添加功能代码
25.          }
26.       } break;
27.       case FDS_EVT_DEL_RECORD: //FDS 记录删除事件
28.       {
29.          if (p_evt->result == NRF_SUCCESS) //记录删除成功
30.          {
31.             //添加功能代码
32.          }
33.       }break;
34.      case FDS_EVT_DEL_FILE: //FDS 删除文件事件
35.       {
```

```
        if (p_evt->result == NRF_SUCCESS) //文件删除成功
        {
            //添加功能代码
        }
    }break;
    case FDS_EVT_GC: //FDS 碎片整理事件
    {
        if (p_evt->result == NRF_SUCCESS) //碎片整理成功
        {
            //添加功能代码
        }
    }break;
    default:
      break;
  }
}
```

4.2. 初始化 FDS

FDS 初始化的库函数是 fds_init()函数，该函数原型如下。

表 10-4：fds_init()函数

函数原型	ret_code_t fds_init(void)
函数功能	初始化 FDS，该函数初始化 FS 模块并安装文件系统（除非已安装）。该函数是异步的，它通过事件报告完成情况，因此，在调用该函数进行初始化之前，需要先调用 fds_register 函数注册事件处理函数，这样，在事件处理函数中就可以接收到操作完成事件。
参　　数	无。
返回值	NRF_SUCCESS：初始化操作已经成功加入执行队列。 FDS_ERR_NO_PAGES：Flash 没有足够的剩余空间安装文件系统。

FDS 初始化的流程如下图所示，FDS 执行初始化后，要等待 FDS 事件处理函数接收到初始化事件，并判断初始化成功后，才可以继续执行 FDS 的其它事务。

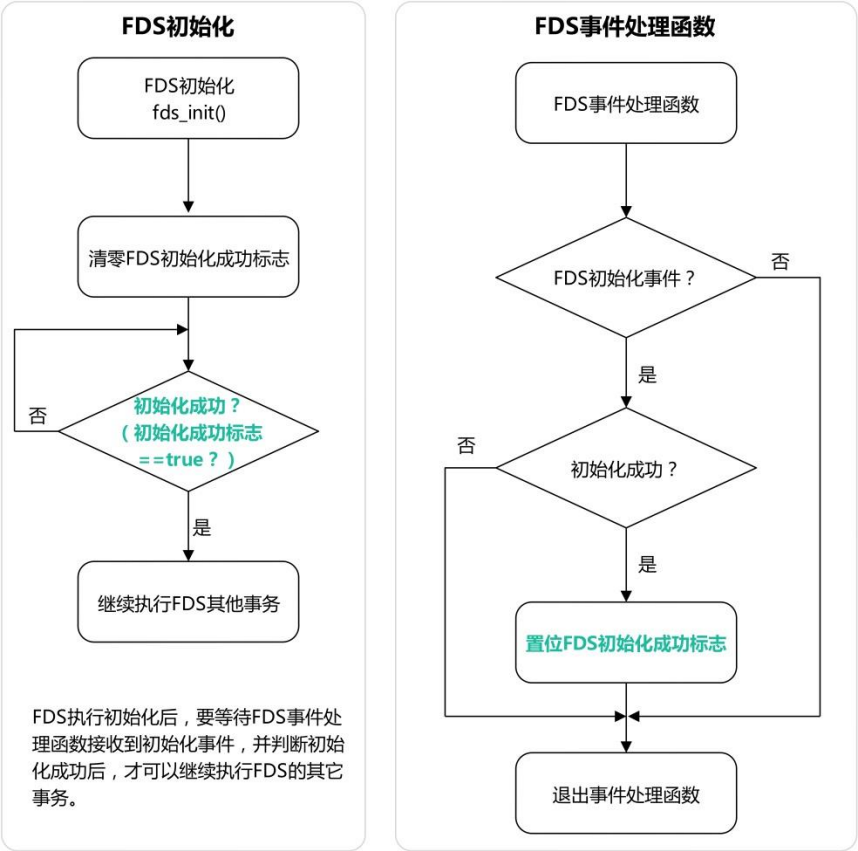


图 10-3：FDS 初始化流程

FDS 初始化的示例代码如下：

代码清单：FDS 初始化示例

```
ret_code_t rc = fds_init();//初始化 FDS
APP_ERROR_CHECK(rc);//用错误处理模块检查函数返回值
//FDS 初始化是异步的，因此要等待 FDS 初始化完成
wait_for_fds_init();
```

4.3. 查找记录

FDS 提供了如下 3 个函数用于查找记录：

- fds_record_find：根据指定的文件和记录 key 查找记录，即在指定的文件中查找指定记录 key 的记录。
- fds_record_find_by_key：根据给定的记录 key 查找记录，该函数会在所有文件中查找。
- fds_record_find_in_file：查找指定文件中的所有记录。

3 个记录查找函数均会在搜索到第一个符合的记录后返回，查找到的记录信息保存到“p_desc”指向的记录描述符中。另外，3 个函数均有“p_token”参数用于保存当前操作搜索的位置以便继续查找时不用从头开始搜索，3 个记录查找函数原型如下：

表 10-5：fds_record_find()函数

函数原型	ret_code_t fds_record_find

	(uint16_t file_id, uint16_t record_key, fds_record_desc_t * p_desc, fds_find_token_t * p_token)
函数功能	根据给定的记录 key 在指定的文件中查找对应的记录。该函数查找给定记录 key 的文件中的第一条记录。如果要在文件中搜索具有相同 key 的下一条记录，再次调用该函数并提供相同的 fds_find_token_t 结构体，以便从找到的最后一条记录的位置继续搜索。
参　　数	[in] file_id：文件 ID。 [in] record_key：记录 key。 [out] p_desc：查找到的记录的描述符。 [out] p_token：包含相关操作进度的 Token。
返回值	NRF_SUCCESS：查找到符合的记录。 FDS_ERR_NOT_INITIALIZED：模块未初始化。 FDS_ERR_NULL_ARG：p_desc 或 p_token 指向 NULL。 FDS_ERR_NOT_FOUND：没有查找到符合的记录。

表 10-6：fds_record_find_by_key()函数

函数原型	ret_code_t fds_record_find_by_key (uint16_t record_key, fds_record_desc_t * p_desc, fds_find_token_t * p_token)
函数功能	根据给定的记录 key 查找记录。该函数查找具有给定记录 key 的第一条记录，与记录所属文件无关。如果要继续搜索具有相同 key 的下一条记录，再次调用该函数并提供相同的 fds_find_token_t 结构体，以便从找到的最后一条记录的位置继续搜索。
参　　数	[in] record_key：记录 key。 [out] p_desc：查找到的记录的描述符。 [out] p_token：包含相关操作进度的 Token。
返回值	NRF_SUCCESS：查找到符合的记录。 FDS_ERR_NOT_INITIALIZED：模块未初始化。 FDS_ERR_NULL_ARG：p_desc 或 p_token 指向 NULL。 FDS_ERR_NOT_FOUND：没有查找到符合的记录。

表 10-7：fds_record_find_in_file()函数

函数原型	ret_code_t fds_record_find_in_file (uint16_t file_id, fds_record_desc_t * p_desc, fds_find_token_t * p_token)
函数功能	搜索文件中的任一记录。该函数查找指定文件中的第一条记录，与记录 key 无关。如果要继续搜索同一文件中的下一条记录，再次调用该函数并提供相同的 fds_find_token_t 结构体，以便从找到的最后一条记录的位置继续搜索。
参　　数	[in] file_id：文件 ID。 [out] p_desc：查找到的记录的描述符。 [out] p_token：包含相关操作进度的 Token。
返回值	NRF_SUCCESS：查找到符合的记录。 FDS_ERR_NOT_INITIALIZED：模块未初始化。 FDS_ERR_NULL_ARG：p_desc 或 p_token 指向 NULL。 FDS_ERR_NOT_FOUND：没有查找到符合的记录。

记录查找函数中使用了 fds_record_desc_t 和 fds_find_token_t 两个结构体，其中 fds_record_desc_t 称为记录描述符结构体，它用来关联一个具体的记录，应用程序只有使用记录描述符才能访问记录，记录查找即是通过文件或记录 key 等条件来获取记录描述符的过程。记录描述符结构体的声明如下：

代码清单：记录描述符结构体声明

```
1. typedef struct
2. {
3.     uint32_t         record_id;         //记录唯一 ID
4.     uint32_t const * p_record;          //Flash 中记录的最后已知位置
5.     uint16_t         gc_run_count;      //该记录被碎片收集过的次数
6.     bool             record_is_open;    //记录是否打开
7. } fds_record_desc_t;
```

fds_find_token_t 用于保存记录查找的进度，这样我们在继续查找时可以从 fds_find_token_t 保存的查找进度开始查找，而不必从头开始。但是如果我们执行全新的记录查找，务必将 fds_find_token_t 结构体变量清零。fds_find_token_t 结构体声明如下：

代码清单：记录查找令牌结构体声明

```
1. typedef struct
```

```
{
    uint32_t const * p_addr;//记录存储的地址
    uint16_t         page;  //记录所在的页面
} fds_find_token_t;
```

使用 fds_record_find()函数查找记录的示例代码如下，查找到符合的记录后，记录描述符保存到“desc”，应用程序即可使用“desc”访问该记录。

代码清单：FDS 查找记录示例

```
//定义文件 ID 和该文件包含的记录的 KEY
#define FILE_ID                    (0x1000)//文件 ID
#define RECORD_KEY                 (0x1001)//记录 KEY

//清零记录查找令牌
memset(&tok, 0x00, sizeof(fds_find_token_t));
//在 DEVICE_VERSION_FILE 文件中查找记录 HW_VERSION_KEY
ret_code_t rc = fds_record_find(FILE_ID, RECORD_KEY, &desc, &tok);
//判断查找结果，进行相应处理
if (rc == NRF_SUCCESS)
{
   //添加功能代码
}
else
{
   //添加功能代码
}
```

4.4. 读取记录

读取一条记录时必须有该记录的描述符，记录的描述符可以通过记录查找函数获取。

读取记录内容之前，先要打开记录，记录打开后会阻止该记录所在虚拟页面上的碎片收集操作，从而确保读取过程中记录不会被修改，读取完成后关闭记录以便碎片收集可以操作该记录。打开和关闭记录的函数一般成对出现，记录打开后，当读取完成后，务必用关闭函数关闭记录。读取记录内容的流程如下图所示：

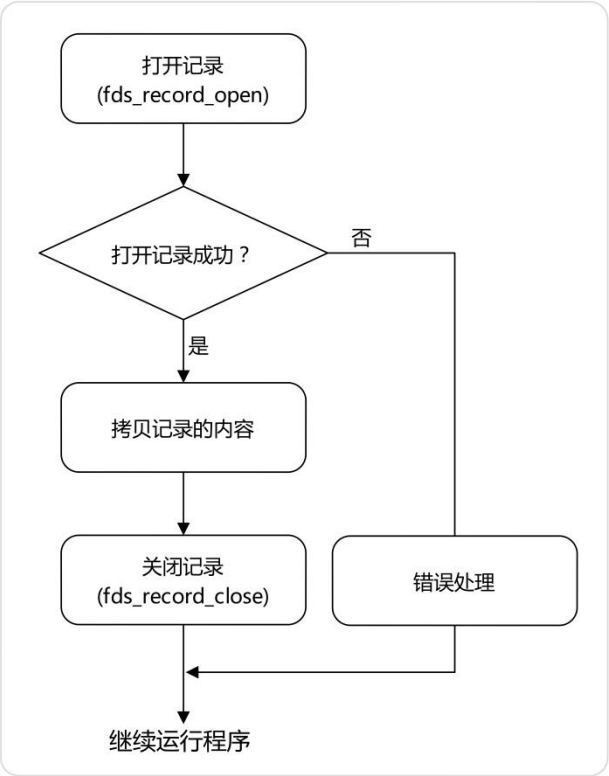


图 10-4：读取记录内容步骤

FDS 打开记录的库函数是 fds_record_open()，该函数原型如下。

表 10-8：fds_record_open()函数

函数原型	ret_code_t fds_record_open (fds_record_desc_t * p_desc, fds_flash_record_t * p_flash_record,)
函数功能	该函数用于打开记录以读取数据。 ● fds_record_open 函数打开存储在 Flash 中的记录，以便应用程序读取记录。该函数执行后，会初始化和记录关联的 fds_flash_record_t 结构体，并返回一个指向该结构体的指针，通过这个指针即可访问记录数据及其关联的元数据。 ● fds_record_open 函数执行后，会阻止该记录所在虚拟页面上的碎片收集操作，因此，只要记录保持打开，fds_flash_record_t 中字段指向的内存内容就保证不会被修改，这保证了记录数据在读取过程中的完整性。 ● 切记，读完记录后，需要调用 fds_record_close 将其关闭，这样，碎片收集可以回收存储记录的虚拟页面上的空间。注意，必须为 fds_record_close 函数提供与此函数相同的描述符。
参　　数	[in] p_desc：待打开的记录的描述符。

	[out] p_flash_record：存储在 Flash 中的记录。
返回值	NRF_SUCCESS：记录打开成功。 FDS_ERR_NULL_ARG：p_desc 或 p_flash_record 指针为 NULL。 FDS_ERR_NOT_FOUND：记录不存在，这可能是记录被删除了或者 Flash 中没有写入该记录。 FDS_ERR_CRC_CHECK_FAILED：记录的 CRC 校验错误。

FDS 关闭记录的库函数是 fds_record_close()，该函数原型如下。

表 10-9：fds_record_close()函数

函数原型	ret_code_t fds_record_close(fds_record_desc_t * p_desc)
函数功能	该函数用于关闭记录。关闭记录后允许在该记录所在页面上进行碎片收集操作(如果该页面上没有其他记录被打开)。注意关闭记录时使用的记录描述符要和打开记录时的描述符一样。另外，关闭记录并不会作废该记录的描述符，我们仍然可以使用该描述符去操作对应的记录。
参　　数	[in] p_desc：待关闭的记录的描述符。
返回值	NRF_SUCCESS：记录关闭成功。 FDS_ERR_NULL_ARG：p_desc 指针为 NULL。 FDS_ERR_NO_OPEN_RECORDS：记录没有使用 fds_record_open()打开。 FDS_ERR_NOT_FOUND：记录不存在。

4.5. 写记录

FDS 写记录的库函数是 fds_record_write()函数，该函数原型如下。

表 10-10：fds_record_write()函数

函数原型	ret_code_t fds_record_write (fds_record_desc_t * p_desc, fds_record_t const * p_record,)
函数功能	该函数用于向 Flash 中写入一条记录。 ● 除了记录 key 必须与 FDS_RECORD_KEY_DIRTY 不同且文件 ID 必须与 FDS_FILE_ID_INVALID 不同外，写入时对文件 ID 和记录 key 没有限制。特别是，没有关于文件 ID 或记录 key 的唯一性的限制，具有相同文件 ID 的所有记录都分组到一个文件中。如果不存在具有指定 ID 的文件，则会创建该文件。 在文件中可以有多个具有相同记录键的记录。 ● 记录数据可以包含多个块，数据必须与 4 字节边界对齐，并且由于它不在内部缓冲，因此必须将其保留在内存中，直到收到写入操作

	的回调为止。 ● 数据长度不得超过 FDS_VIRTUAL_PAGE_SIZE 字减去 14 个字节。 ● 写入函数是异步的，通过发送到已注册事件处理函数的事件报告操作完成。
参　　数	[out] p_desc：写入的记录返回的描述符。这是一个输出参数，调用记录写函数操作成功后会返回记录描述符，p_desc 即指向这条记录描述符，之后，程序可以通过这条记录描述符对记录进行相应的操作。 [in] p_record：待写入的记录。
返回值	NRF_SUCCESS：写操作成功加入队列。 FDS_ERR_NOT_INITIALIZED：FDS 模块没有初始化。 FDS_ERR_NULL_ARG：p_record 指向 NULL。 FDS_ERR_INVALID_ARG：文件 ID 或记录 key 无效。 FDS_ERR_UNALIGNED_ADDR：记录数据没有 4 字节对齐。 FDS_ERR_RECORD_TOO_LARGE：记录数据长度超过最大长度。 FDS_ERR_NO_SPACE_IN_QUEUES：操作队列满，写操作无法加入队列或者记录数据块的数量超过了可缓存的数量。 FDS_ERR_NO_SPACE_IN_FLASH：Flash 没有足够的空间存放该记录。

FDS 写记录的流程如下图所示，FDS 写记录是异步的，执行写记录后，要等待 FDS 事件处理函数接收到写记录事件，才可以继续执行 FDS 的其它事务，并且需要在写记录事件中判断写记录是否成功，从而执行相应的处理。

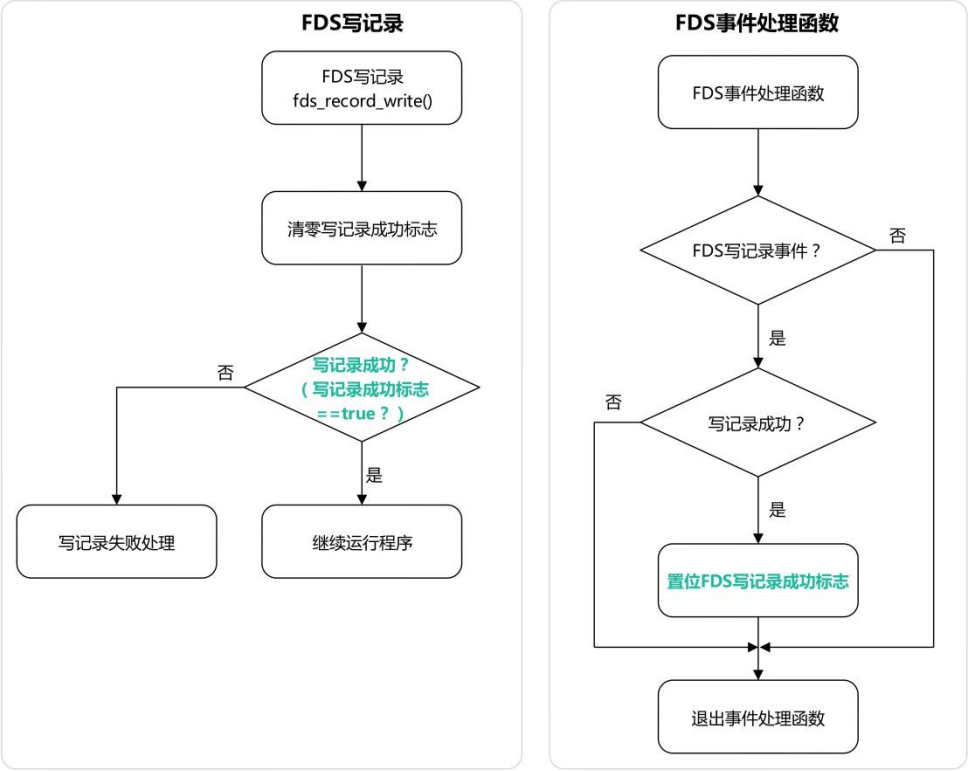


图 10-5：写记录流程

写记录的示例代码如下，尤其注意待写入的数据在 Flash 中的存储必须是 4 字节对齐的，即 m_version_record 中的 p_data 指向的地址必须是 4 字节对齐的。

代码清单：FDS 写记录示例

```
static bool volatile m_fds_writed;//定义写记录成功标志

m_fds_writed = false;//写记录前清零写记录成功标志
rc = fds_record_write(&desc, &m_version_record);//写入记录
APP_ERROR_CHECK(rc);//用错误处理模块检查函数返回值
wait_for_fds_write();//等待写记录完成
```

4.6. 记录更新

FDS 更新记录的库函数是 fds_record_update()函数，该函数原型如下。更新记录并不会将原记录彻底从 Flash 中删除，更新记录时会将更新的内容以新的记录写入 Flash，原先的记录作废。这样做的好处是可以不用擦除 Flash，从而可以节省时间和降低 Flash 页面擦写的频度。

表 10-11：fds_record_update()函数

函数原型	ret_code_t fds_record_update (fds_record_desc_t * p_desc, fds_record_t const * p_record,)
函数功能	该函数用于更新一条指定的记录，更新记录首先将新记录（p_record）写入 Flash，然后删除旧记录（由 p_desc 标识）。 ● 除了记录 key 必须与 FDS_RECORD_KEY_DIRTY 不同、文件 ID 必须与 FDS_FILE_ID_INVALID 不同外，对文件 ID 和记录 key 没有限制。特别是，没有关于文件 ID 或记录 key 的唯一性的限制。具有相同文件 ID 的所有记录都分组到一个文件中，如果不存在具有指定 ID 的文件，则会创建该文件。在文件中可以有多个具有相同记录 key 的记录。 ● 记录数据可以包含多个块，数据必须与 4 字节边界对齐，并且由于它不在内部缓冲，因此必须将其保留在内存中，直到收到操作的回调为止。 ● 数据长度不得超过 FDS_VIRTUAL_PAGE_SIZE 字减去 14 个字节。 ● 该函数是异步的，它通过发送到已注册事件处理函数的事件报告更新操作完成。
参　　数	[in,out] p_desc：待更新的记录的描述符，函数以 NRF_SUCCESS 返回时，此参数包含新写入记录的描述符。 [in] p_record：待写入的更新记录。

返回值	NRF_SUCCESS：记录更新操作成功加入队列。 FDS_ERR_NOT_INITIALIZED：FDS 模块没有初始化。 FDS_ERR_INVALID_ARG：文件 ID 或记录 key 无效。 FDS_ERR_UNALIGNED_ADDR：记录数据没有 4 字节对齐。 FDS_ERR_RECORD_TOO_LARGE：记录数据长度超过最大长度。 FDS_ERR_NO_SPACE_IN_QUEUES：操作队列满，写操作无法加入队列或者记录数据块的数量超过了可缓存的数量。 FDS_ERR_NO_SPACE_IN_FLASH：Flash 没有足够的空间存放更新的记录。

FDS 更新记录是异步的，因此 FDS 更新记录后，要等待 FDS 事件处理函数接收到更新记录事件后，才可以继续执行 FDS 的其它事务，并且需要在更新记录事件中判断更新记录是否成功，从而执行相应的处理，其流程和写记录类似。

4.7. 删除记录

FDS 删除记录的库函数是 fds_record_delete()函数，该函数原型如下。删除记录并不会将原记录彻底从 Flash 中删除，而是将记录作废。记录删除后，记录所占用的 Flash 空间可以通过碎片收集回收。

表 10-12：fds_record_delete()函数

函数原型	ret_code_t fds_record_delete(fds_record_desc_t * p_desc)
函数功能	该函数用于删除一条记录。 ● 记录一旦删除后，就不能通过 fds_record_find、fds_record_find_by_key 或 fds_record_find_in_file 函数来查找定位，另外，也不能使用 fds_record_open 函数打开记录。记录删除后，记录所使用的 Flash 空间也不会立即释放，如果要释放作废记录使用的 Flash 空间，必须通过碎片收集函数来实现。 ● 记录删除函数是异步操作的，它是通过发送一个事件给已注册的事件处理函数来报告操作完成的。
参　　数	[in] p_desc：待删除的记录的描述符。
返回值	NRF_SUCCESS：初始化操作已经成功加入执行队列。 FDS_ERR_NOT_INITIALIZED：模块未初始化。 FDS_ERR_NULL_ARG：p_desc 指向 NULL。 FDS_ERR_NO_SPACE_IN_QUEUES：队列已满，操作无法加入队列。

FDS 删除记录是异步的，因此 FDS 删除记录后，要等待 FDS 事件处理函数接收到删除记录事件，才可以继续执行 FDS 的其它事务，并且需要在删除记录事件中判断删除记录是

否成功，从而执行相应的处理，其流程和写记录类似。

4.8. 碎片收集

FDS 碎片收集的库函数是 fds_gc()函数，该函数原型如下。碎片收集运行时会占用较长的时间，程序中应尽量减少碎片收集的次数。

表 10-13：fds_gc()函数

函数原型	ret_code_t fds_gc(void)
函数功能	该函数用于运行碎片收集。 ● 碎片收集回收已被删除的记录占用的 Flash 空间，或者由于例如断电而无法完全写入的闪存空间。该函数是异步的。它是通过发送一个事件给已注册的事件处理函数来报告操作完成的。
参　　数	无。
返回值	NRF_SUCCESS：初始化操作已经成功加入执行队列。 FDS_ERR_NOT_INITIALIZED：模块未初始化。 FDS_ERR_NO_SPACE_IN_QUEUES：队列已满，操作无法加入队列。

FDS 碎片收集是异步的，因此碎片收集后，要等待 FDS 事件处理函数接收到碎片收集事件，才可以继续执行 FDS 的其它事务，并且需要在碎片收集事件中判断碎片收集是否成功，从而执行相应的处理，其流程和写记录类似。碎片收集的示例代码如下：

代码清单：FDS 碎片收集示例

```
static bool volatile m_fds_gc; //定义碎片收集完成标志

m_fds_gc = false;//碎片收集完成标志清零
ret_code_t rc = fds_gc();//FDS 碎片回收
APP_ERROR_CHECK(rc);//用错误处理模块检查函数返回值
wait_for_fds_gc();//等待碎片收集完成
```

4.9. 信息统计

FDS 提供了一个用于统计信息的函数 fds_stat()，应用程序可以通过该函数获取 FDS 的文件系统统计信息，函数原型如下。

表 10-14：fds_stat()函数

函数原型	ret_code_t fds_stat(fds_stat_t *　p_stat)
函数功能	该函数用于检索文件系统统计信息的函数。 此函数检索文件系统统计信息，例如打开的记录数，垃圾回收可以回收的空间等。
参　　数	[out] p_stat：文件系统统计信息。

返回值	NRF_SUCCESS：如果统计信息已成功返回。 FDS_ERR_NOT_INITIALIZED：模块未初始化。 FDS_ERR_NULL_ARG：p_stat 指针为 NULL。

获取 FDS 文件系统统计信息的示例代码如下，首先定义一个 FDS 文件系统统计信息结构体变量“stat”用于保存读取的信息，之后调用 fds_stat()函数获取统计信息，获取成功后，统计信息保存到变量“stat”。

代码清单：获取 FDS 文件系统统计信息示例

```
1. //定义一个 FDS 文件系统统计信息结构体，用于保存获取的统计信息
2. fds_stat_t stat = {0};
3. //获取 FDS 文件系统统计信息
4. ret_code_t rc = fds_stat(&stat);
5. //用错误处理模块检查函数返回值
6. APP_ERROR_CHECK(rc);
7. //打印可用页面数量
8. printf("Pages available:%d\r\n", stat.pages_available);
9. //打印处于打开状态的记录的数量
10.printf("open  record   :%d\r\n", stat.open_records);
11.//打印记录的数量
12.printf("Valid records  :%d\r\n", stat.valid_records);
13.//打印“无效”的记录数量
14.printf("Dirty records  :%d\r\n", stat.dirty_records);
```

5.初学 FDS 时的疑问

如果您第一次接触FDS，您有可能会对使用FDS时如何唯一地表示一条记录感到疑惑，本章旨在消除这个疑惑，如果您理解了这一点，您会发现 FDS 用起来很方便。

■ **疑问**

FDS 的文件 ID 和记录的 KEY 可以一样，同一个文件里面的记录 KEY 也可以一样，虽然记录 ID 对于记录来说是唯一的，但是记录 ID 的值在记录更新后会变化，那么，我们在程序中怎么确定一条记录。

■ **解答**

没错，FDS 确实是这样的。但是，这仅仅表示 FDS 本身没有对文件 ID 和记录的 KEY 的唯一性做要求，并不表示我们在实际应用中可以随意定义文件 ID 和记录的 KEY。

记住，我们在使用 FDS 时正是通过文件 ID 和记录 Key 来确定一条记录的，这表示我们使用 FDS 时需要规划一下文件 ID 和记录 Key，以确保它们没有重复的，下表是规划文件和记录的一个例子，可以看到，这样规划后，程序中就可以通过文件 ID 和记录 key 唯一确定一条记录了，我们可以使用记录查找函数，以文件 ID 和记录 key 为参数获取对应记录的描述符，从而操作这条记录。

表 10-15：规划文件和记录

文件 ID	记录 Key	描述
0x1000	0x1001	该记录由文件 ID（0x1000）和记录 key（0x1001）唯一标志。
	0x 1002	该记录由文件 ID（0x1000）和记录 key（0x1002）唯一标志。
0x2000	0x 2001	该记录由文件 ID（0x2000）和记录 key（0x2001）唯一标志。
	0x 2002	该记录由文件 ID（0x2000）和记录 key（0x2002）唯一标志。
	0x 2003	该记录由文件 ID（0x2000）和记录 key（0x2003）唯一标志。

6.裸机 FDS 实验

本实验在开发指南（上册）配套实验源码“实验 10-1：串口数据收发”的基础上修改。程序中使用如下表所示的 2 条记录，这 2 条记录均归属于同一个文件。

图 10-16：文件和记录

文件 ID	记录名称	记录 Key	描述
0x1000（宏定义 DEVICE_FILE）	m_version_record	1001	存储硬件/软件版本信息。
	m_fw_record	1002	存储固件的大小。

程序中通过 4 个按键分别演示了记录的写、更新、读取、删除、获取统计信息和碎片收集功能。

- S1 按键：查找 DEVICE_FILE 文件中是否已经写入了记录 m_version_record 和 m_fw_record，如果已经写入，更新记录，否则，写入记录。
- S2 按键：读取记录 m_version_record 内容，并通过串口输出读取的记录内容。
- S3 按键：删除记录 m_version_record 和 m_fw_record。
- S4 按键：获取 FDS 文件系统统计信息并通过串口打印出文件系统统计信息，之后执行碎片收集。

✧ 注：本节对应的实验源码是：“实验 10-1：FDS 数据存储-裸机”。

6.1. 添加需要的文件

裸机程序使用 FDS 需要加入的文件如下表所示。

表 10-17： 裸机程序使用 FDS 需要加入的文件

文件名	SDK 中的目录	描述
nrf_nvmc.c	..\modules\nrfx\hal	NVMC 驱动文件(裸机程序需要添加)。
nrf_fstorage_nvmc.c	..\components\libraries\fstorage	FS 后端 nrf_fstorage_nvmc(裸机程序需要添加)。
crc16.c	..\components\libraries\crc16	CRC16 校验库文件。
nrf_fstorage.c	..\components\libraries\fstorage	FS 驱动文件(裸机程序和使用 Softdevice 的程序均需要添加)。

nrf_atfifo.c	..\components\libraries\atomic_fifo	atomic FIFO。
fds.c	...\components\libraries\fds	FDS 库文件。

6.2. 头文件引用和路径设置

1. 需要引用的头文件

因为在“main.c”文件中使用了 FDS 的驱动程序，所以“main.c”文件需要引用下面的头文件。

#include "fds.h"

另外，本例中使用了字符串的操作，因此需要引用 C 库头文件“string.h”。

#include <string.h>

2. 需要添加的头文件包含路径

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

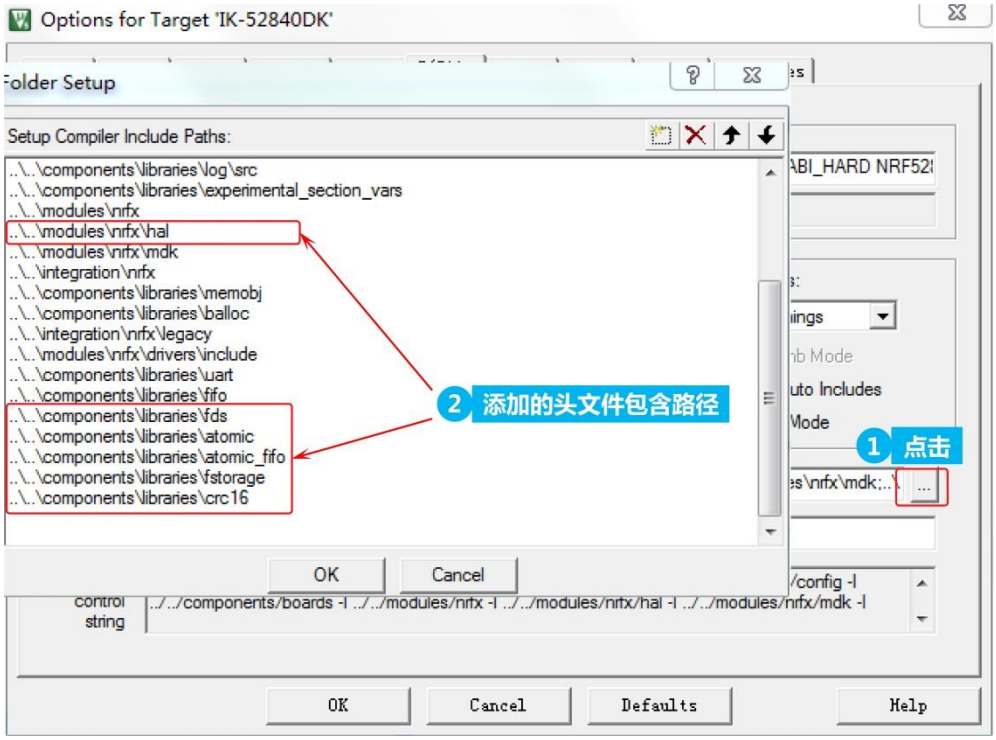


图 10-6：添加头文件包含路径

本例需要添加的头文件路径如下表：

表 10-18： 头文件包含路径

序号	路径
1	..\..\modules\nrfx\hal
2	..\..\components\libraries\crc16
3	..\..\components\libraries\fstorage
4	..\..\components\libraries\atomic_fifo
5	..\..\components\libraries\fds
6	..\..\components\libraries\atomic

6.3. 工程配置

打开“sdk_config.h”文件，加入 FDS 需要的配置，如下图所示，编辑内容时切换到“Text Editor”进行编辑（编辑的内容参考实验的源码，因此代码很长，此处不贴出代码）本例需要添加如下 3 个程序模块的配置。

- CRC：（588~593）行。
- FDS：（595~710）行。
- FS：（712~767）行。

编辑完成后，切换到“Configuration Wizard”，即可观察到配置的具体项目：

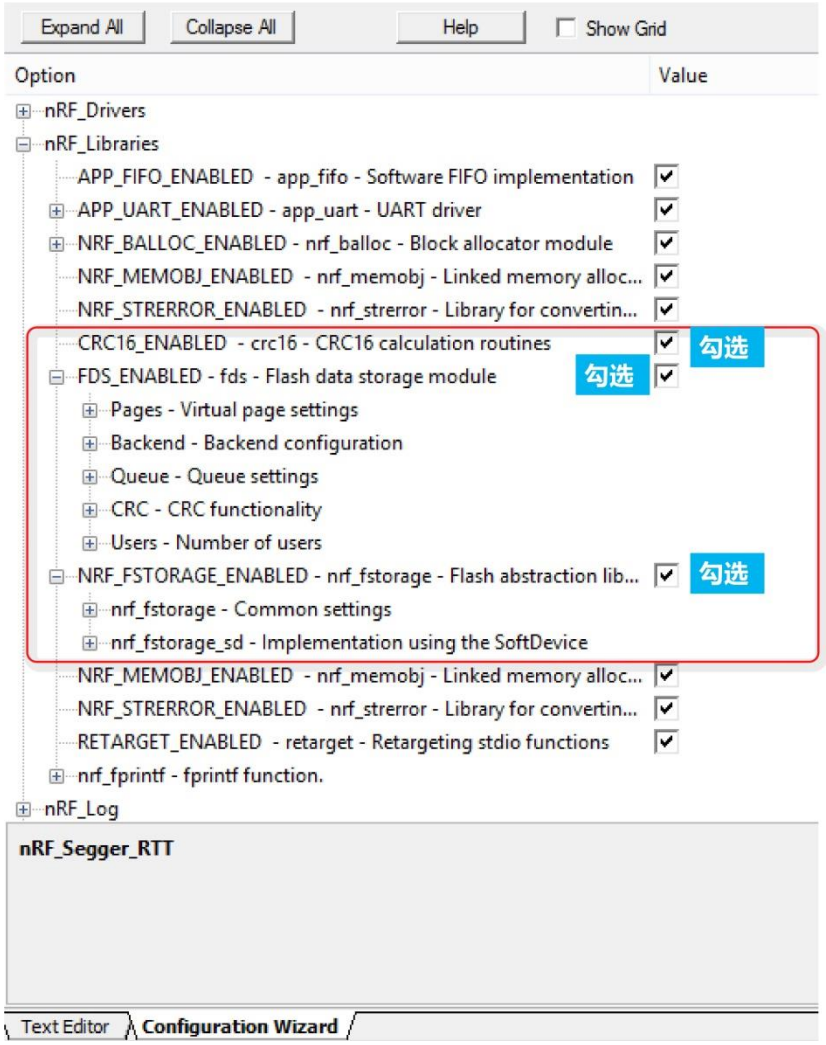


图 10-7：工程配置

6.4. 代码编写

1. 注册 FDS 事件句柄

根据 FDS 的应用流程，首先需要注册 FDS 事件句柄，即编写 FDS 事件处理函数，之后将该事件处理函数注册给 FDS 程序模块，FDS 事件处理函数程序清单如下，本例中我们关注了记录的初始化、写、更新、删除和碎片收集事件，读者在开发的时候，需要更新实际情况关注对应的事件。

代码清单：FDS 事件处理函数

```
1. static void fds_evt_handler(fds_evt_t const * p_evt)
2. {
3.     //串口打印出 FDS 事件，方便实验时观察 FDS 的操作结果
4.     printf("Event: %s received (%s)\r\n",
5.                    fds_evt_str[p_evt->id],
6.                    fds_err_str[p_evt->result]);
7.     //判断事件类型
8.     switch (p_evt->id)
9.     {
10.        case FDS_EVT_INIT://FDS 初始化事件
11.            if (p_evt->result == NRF_SUCCESS)  //初始化成功
12.            {
13.                m_fds_initialized = true;
14.            }
15.            break;
16.        case FDS_EVT_WRITE:  //FDS 写记录事件
17.        {
18.            //写记录成功
19.            if (p_evt->result == NRF_SUCCESS)
20.            {
21.                m_fds_writed = true;
22.                printf("Record ID:\t0x%04x\r\n",  p_evt->write.record_id);
23.                printf("File ID:\t0x%04x\r\n",    p_evt->write.file_id);
24.                printf("Record key:\t0x%04x\r\n", p_evt->write.record_key);
25.            }
26.        } break;
27.        case FDS_EVT_UPDATE:  //FDS 更新记录事件
28.        {
29.            if (p_evt->result == NRF_SUCCESS)  //更新记录成功
30.            {
31.                m_fds_updated = true;
32.                printf("Record ID:\t0x%04x\r\n",  p_evt->write.record_id);
33.                printf("File ID:\t0x%04x\r\n",    p_evt->write.file_id);
34.                printf("Record key:\t0x%04x\r\n", p_evt->write.record_key);
35.            }
36.        } break;
37.        //FDS 删除记录事件
38.        case FDS_EVT_DEL_RECORD:
39.        {
40.            //删除记录成功打印出记录 ID、文件 ID 和记录 key
41.            if (p_evt->result == NRF_SUCCESS)
42.            {
43.                printf("Record ID:\t0x%04x\r\n",  p_evt->del.record_id);
44.                printf("File ID:\t0x%04x\r\n",    p_evt->del.file_id);
```

```
            printf("Record key:\t0x%04x\r\n", p_evt->del.record_key);
         }
      } break;
      case FDS_EVT_GC:  //FDS 碎片整理事件
      {
         if (p_evt->result == NRF_SUCCESS) //碎片整理成功
         {
            printf("gc completed");
            m_fds_gc = true;
         }
      } break;
      default:
        break;
   }
}
```

编写好 FDS 事件处理函数后，调用 fds_register()注册即可，代码清单如下。

代码清单：注册 FDS 事件句柄

```
//注册 FDS 事件回调函数接收 FDS 事件
(void)fds_register(fds_evt_handler);
```

2. 初始化 FDS

FDS 注册事件句柄后，接下来要做的是初始化，代码清单如下。

代码清单：初始化 FDS

```
rc = fds_init();//初始化 FDS
APP_ERROR_CHECK(rc);//用错误处理模块检查函数返回值
//FDS 初始化是异步的，因此要等待 FDS 初始化完成
wait_for_fds_init();
```

FDS 初始化是异步的，初始化完成后会通过 FDS 事件处理函数将初始化结果通知应用程序。本例中我们使用变量 m_fds_initialized 标志 FDS 初始化是否成功，应用程序中执行 FDS 初始化后，等待 m_fds_initialized 置位以判断 FDS 是否成功初始化（FDS 事件处理函数中接收到 FDS 初始化完成事件后读取初始化结果，如初始化成功置位 m_fds_initialized）。

代码清单：等待 FDS 初始化完成

```
//等待 FDS 初始化完成
static void wait_for_fds_init(void)
{
    while (!m_fds_initialized)
    {
        power_manage();  //等待过程中进入低功耗
    }
```

```
}
```

3. 定义和初始化记录

本例中使用 fds_record_t 声明两个记录 m_version_record 和 m_fw_record，它们的 file_id 均初始化为 DEVICE_FILE，即它们均归属于文件 DEVICE_FILE。

代码清单：定义并初始化记录 m_version_record 和 m_fw_record

```
//包含硬件/软件版本信息的记录
static fds_record_t const m_version_record =
{
    .file_id           = DEVICE_FILE,
    .key               = DEVICE_VERSION_KEY,
    .data.p_data       = &device_version,
    //记录的长度必须以 4 字节（字）为单位
    .data.length_words = (sizeof(device_version) + 3) / sizeof(uint32_t),
};
//包含硬件版本信息的记录
static fds_record_t const m_fw_record =
{
    .file_id           = DEVICE_FILE,
    .key               = FW_INFORMATION_KEY,
    .data.p_data       = &fw_information,
    //记录的长度必须以 4 字节（字）为单位
    .data.length_words = (sizeof(fw_information) + 3) / sizeof(uint32_t),
};
```

初始化记录时，要特别注意的是记录数据必须是 4 字节对齐的，否则 FDS 会出错。本例中定义的记录数据如下。

代码清单：定义并初始化记录 m_version_record 和 m_fw_record

```
//记录 m_version_record 的内容，也可以定义为数组，但是注意一定要 4 字节对齐
typedef struct
{
        char     hw_version[12];
        char     sw_version[12];
}__attribute__((aligned(4)))device_version_t;
//记录 m_fw_record 的内容，也可以定义为数组，但是注意一定要 4 字节对齐
typedef struct
{
        uint32_t    size;
}__attribute__((aligned(4)))fw_information_t;

//初始化硬件和软件版本
static device_version_t device_version =
```

```
{
    .hw_version = "HW-V1.11",
    .sw_version = "SW-V2.22"
};

//初始化固件大小
static fw_information_t fw_information =
{
    .size = 11111,
};
```

4. 按键 S1 按下：写记录和更新记录

写记录和更新记录前，我们先定义记录描述符和记录查找令牌，记录描述符用来标志操作的记录，记录查找令牌用来查找记录，代码清单如下。

代码清单：定义记录描述符和记录查找令牌

```
//定义并初始化记录描述符结构体变量
fds_record_desc_t desc = {0};
//定义并初始化记录查找令牌结构体变量
fds_find_token_t  tok  = {0};
```

主循环中检测到按键 S1 按下后，等待按键释放，之后执行查找记录 m_version_record 和 m_fw_record，如果查找到记录已经存在于 Flash，更新记录，否则，写入新记录。

这里我们是为了演示写记录和更新记录，因此没有修改记录的值，更新记录的时候还是使用原来的记录值更新的。写记录和更新记录代码清单如下（省略了按键检测的代码）。

代码清单：写记录和更新记录

```
//清零 tok，从头开始查找
memset(&tok, 0x00, sizeof(fds_find_token_t));
//在 DEVICE_FILE 文件中查找记录 DEVICE_VERSION_KEY
rc = fds_record_find(DEVICE_FILE, DEVICE_VERSION_KEY, &desc, &tok);
//查找到 DEVICE_VERSION_KEY 记录后，更新记录
if (rc == NRF_SUCCESS)
{
    printf("Record ID %d\r\n", desc.record_id);
    m_fds_updated = false;
    //更新记录 DEVICE_VERSION_KEY
    rc = fds_record_update(&desc, &m_version_record);
    APP_ERROR_CHECK(rc);
    wait_for_fds_update();
}
else//记录不存在，写入记录
{
    printf("Record is not exist,write record...\r\n");
```

```
    m_fds_writed = false;
    rc = fds_record_write(&desc, &m_version_record);
    APP_ERROR_CHECK(rc);
    wait_for_fds_write();
}
//在 DEVICE_FILE 文件中查找记录 m_fw_record
rc = fds_record_find(DEVICE_FILE, FW_INFORMATION_KEY, &desc, &tok);
//查找到 m_fw_record 记录后，更新记录
if (rc == NRF_SUCCESS)
{
    printf("Record ID %d\r\n", desc.record_id);
    m_fds_updated = false;
    //更新记录 m_fw_record
    rc = fds_record_update(&desc, &m_fw_record);
    APP_ERROR_CHECK(rc);
    wait_for_fds_update();
}
else//记录不存在，写入记录
{
   printf("Record is not exist,write record...\r\n");
   m_fds_writed = false;
   rc = fds_record_write(&desc, &m_fw_record);
   APP_ERROR_CHECK(rc);
   wait_for_fds_write();
}
```

5. 按键 S2 按下：读取记录 m_version_record 的内容

主循环中检测到按键 S2 按下后，等待按键释放，之后读取记录 m_version_record 的内容。

读取记录时首先有获取记录描述符，可以通过文件 ID 和记录 KEY 查找记录获取记录描述符。之后打开记录，拷贝记录数据，读取完成后关闭记录，代码清单如下（省略了按键检测的代码）。

代码清单：读取记录

```
//清零 tok，从头开始查找
memset(&tok, 0x00, sizeof(fds_find_token_t));
//在 DEVICE_FILE 文件中查找记录 m_version_record
rc = fds_record_find(DEVICE_FILE, DEVICE_VERSION_KEY, &desc, &tok);
//查找到记录后，更新记录
if (rc == NRF_SUCCESS)
{
    //保存读取的记录数据
   fds_flash_record_t temp = {0};
```

```
10.   //打开记录读取记录内容
11.   rc = fds_record_open(&desc, &temp);
12.   APP_ERROR_CHECK(rc);
13.   static device_version_t temp_version;
14.   //拷贝记录内容
15.   memcpy(&temp_version, temp.p_data, sizeof(device_version_t));
16.   //串口打印记录内容
17.   printf("HW Version is: %s  ", temp_version.hw_version);
18.   printf("SW Version is: %s", temp_version.sw_version);
19.   //读取后，关闭记录
20.  rc = fds_record_close(&desc);
21.  APP_ERROR_CHECK(rc);
22.}
23.else {  //串口打印记录不存在
24.  printf("Record is not exist!\r\n");
25.}
```

6. 按键 S3 按下：删除记录 m_version_record 和 m_fw_record

主循环中检测到按键 S3 按下后，等待按键释放，之后分别查找记录 m_version_record 和 m_fw_record 并删除记录，代码清单如下（省略了按键检测的代码）。

代码清单：删除记录

```
1.  //清零 tok，从头开始查找
2.  memset(&tok, 0x00, sizeof(fds_find_token_t));
3.  //在 DEVICE_FILE 文件中查找记录 m_version_record
4.  rc = fds_record_find(DEVICE_FILE, DEVICE_VERSION_KEY, &desc, &tok);
5.  //删除记录 m_version_record
6.  if (rc == NRF_SUCCESS)fds_record_delete(&desc);
7.  else printf("Record is not exist!\r\n");
8.  //在 DEVICE_FILE 文件中查找记录 m_fw_record
9.  rc = fds_record_find(DEVICE_FILE, FW_INFORMATION_KEY, &desc, &tok);
10. //删除记录 m_fw_record
11. if (rc == NRF_SUCCESS)fds_record_delete(&desc);
12. else printf("Record is not exist!\r\n");
```

7. 按键 S4 按下：获取 FDS 统计信息和碎片收集

主循环中检测到按键 S4 按下后，等待按键释放，之后获取 FDS 文件系统统计信息并通过串口打印出文件系统统计信息，接着执行碎片收集，代码清单如下（省略了按键检测的代码）。

代码清单：获取 FDS 统计信息和碎片收集

```
1.  //定义一个 FDS 文件系统统计信息结构体，用于保存获取的统计信息
2.  fds_stat_t stat = {0};
```

```
//获取 FDS 文件系统统计信息
rc = fds_stat(&stat);
APP_ERROR_CHECK(rc);
//打印可用页面数量
printf("Pages available:%d\r\n", stat.pages_available);
//打印处于打开状态的记录的数量
printf("open  record   :%d\r\n", stat.open_records);
//打印记录的数量
printf("Valid records  :%d\r\n", stat.valid_records);
//打印“无效”的记录数量
printf("Dirty records  :%d\r\n", stat.dirty_records);
m_fds_gc = false;//碎片收集完成标志清零
rc = fds_gc();//FDS 碎片回收
APP_ERROR_CHECK(rc);//用错误处理模块检查函数返回值
wait_for_fds_gc();//等待碎片回收完成
```

6.5. 硬件连接

本实验需要使用 P0.13~P0.16 驱动 LED 指示灯 D1~D4，P0.06 和 P0.08 作为串口通讯引脚，按照下图所示短接跳线帽。

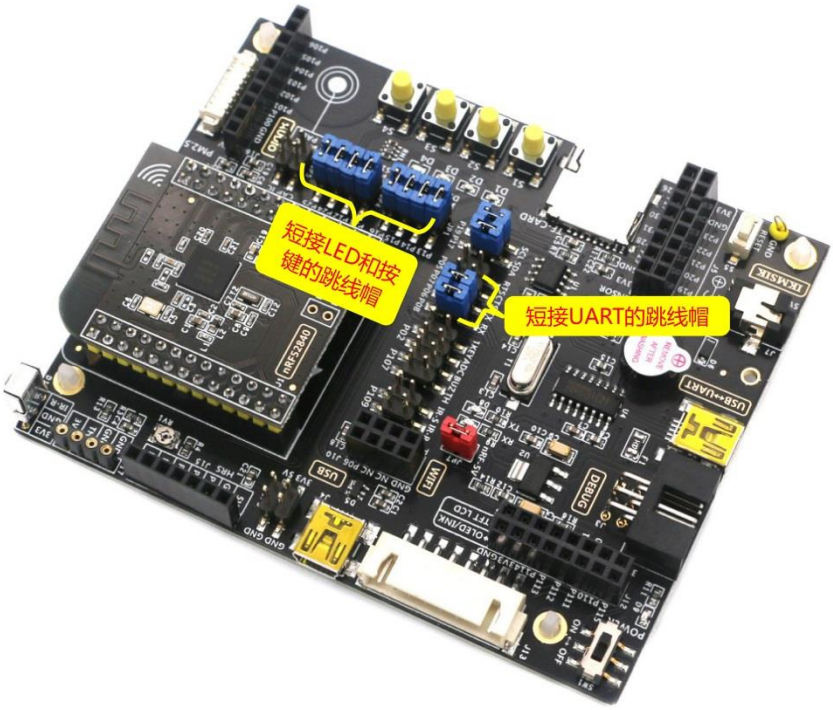


图 10-8：开发板跳线帽短接

6.6. 实验步骤

1. 为了更好地演示实验，首先全片擦除芯片，以确保 Flash 没有已存储的记录。
2. 解压“…\4：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 10-1：FDS 数据存储-裸机”，将解压后得到的文件夹“flash_fds”拷贝到合适的目录，如

“D\nRF52840”。

3. 启动 Keil MDK5.27。
4. 在 Keil MDK5 中执行“Project→Open Project”打开“…\flash_fds\project\mdk5”目录下的工程“flash_fds.uvproj”。
5. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
6. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
7. 电脑上打开串口调试助手，程序运行后，可以在串口调试助手中观察到“Event: FDS_EVT_INIT received (NRF_SUCCESS)”，表示 FDS 初始化成功。

图 10-9：FDS 初始化成功

8. 按下 S1 按键后松开，因为 Flash 中没有写入记录 m_version_record 和 m_fw_record，因此会写入新记录 m_version_record 和 m_fw_record，串口调试助手中可以观察到写入的记录的信息。

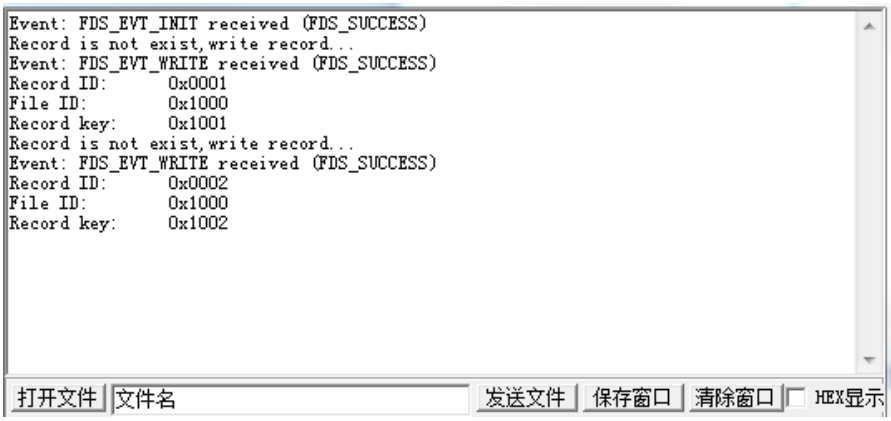


图 10-10：写入记录 m_version_record 和 m_fw_record

9. 再次按下 S1 按键后松开，这时因为 Flash 中已经写入了记录 m_version_record 和 m_fw_record，因此程序会更新这两条记录，串口调试助手中可以观察到更新的记录的信息。

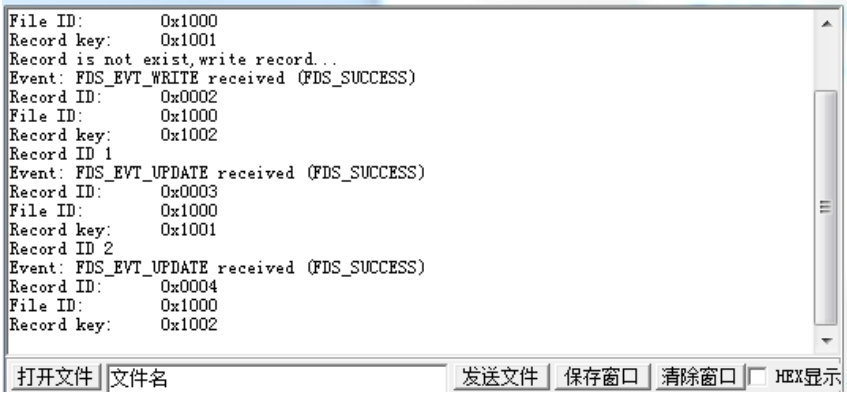


图 10-11：更新记录 m_version_record 和 m_fw_record

10. 按下 S2 按键后松开，程序会读出记录 m_version_record 的数据，串口调试助手中可以观察到读取的记录的数据，如下图所示。

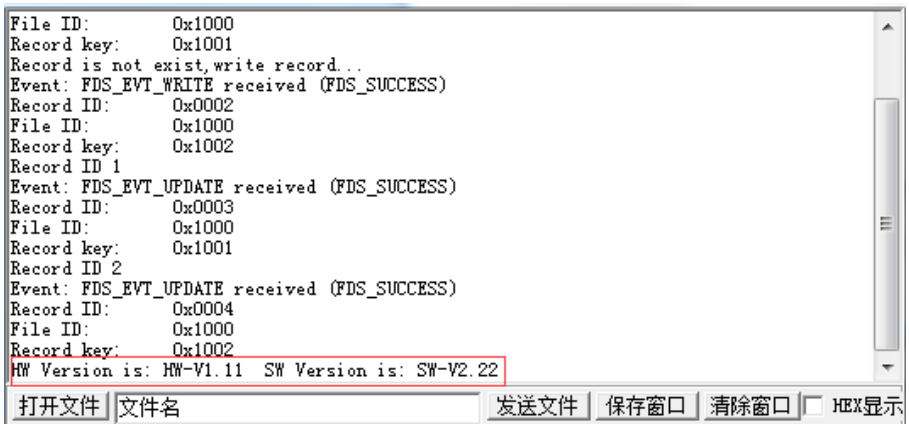


图 10-12：写入记录 m_version_record 和 m_fw_record

11. 按下 S4 按键后松开，串口调试助手中可以观察到 FDS 统计信息，如下图所示，因为我们写入了 2 条记录，更新了 2 条记录，所以可以看到有效记录为 2 条，作废的记录为 2 条。之后程序执行碎片收集，串口调试助手中可以观察“gc completed”表示碎片收集完成。

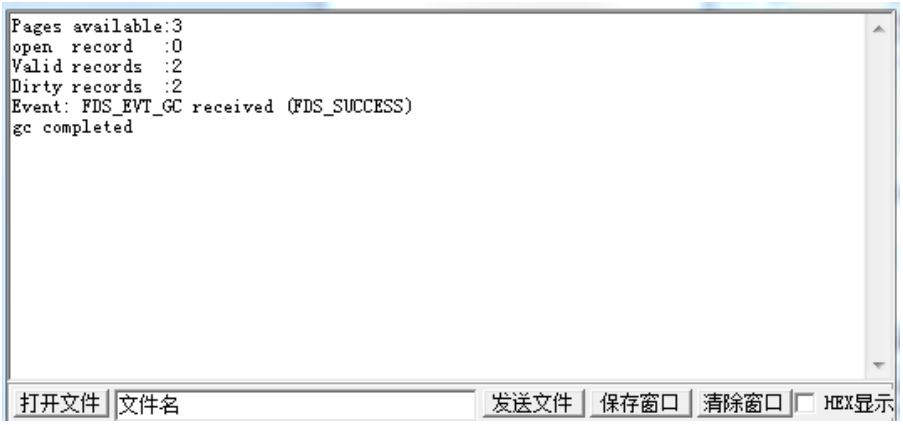


图 10-13：FDS 统计信息和碎片收集

12. 按下 S3 按键后删除记录，然后再按下 S4 按键查看统计信息，这时因为记录已经删除，可以观察到统计信息中有效记录为 0 条。

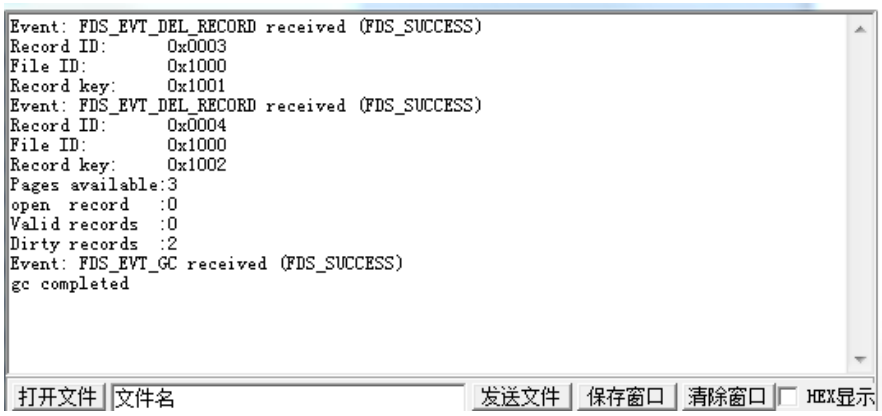


图 10-14：删除记录

7. BLE 应用 FDS 实验

本实验在开发指南（下册）配套实验源码“实验 7-3：串口透传长包传输-判断 notify”的基础上修改。程序中使用如下表所示的 2 条记录，这 2 条记录均归属于同一个文件。

图 10-19：文件和记录

文件 ID	记录名称	记录 Key	描述
0x1000（宏定义 DEVICE_FILE）	m_version_record	1001	存储硬件/软件版本信息。
	m_desp_record	1002	存储设备的描述信息。

程序运行后，首先写入 m_version_record 和 m_desp_record 两条记录，当开发板和主机连接后，若接收到主机发送的命令，根据第一个字节区分命令执行不同的动作。

- 第一个字节是字符“1”：将字符“1”后的数据作为新内容更新记录 m_version_record，要求发送的数据必须是 17 个字节（字符“1”+8 字节硬件版本+8 字节软件版本）。
- 第一个字节是字符“2”：将字符“2”后的数据作为新内容更新记录 m_desp_record，发送的长度范围（2~244）字节。
- 第一个字节是字符“3”：读取记录 m_version_record 和 m_desp_record 的数据并通过 JLINK-RTT 输出。
- 第一个字节是字符“4”：执行碎片收集。

✧ 注：本节对应的实验源码是：“实验 10-2：BLE 串口透传中使用 FDS”。

7.1. 添加需要的文件

BLE 程序使用 FDS 时需要加入的文件如下表所示。

表 10-20： 裸机程序使用 FDS 需要加入的文件

文件名	SDK 中的目录	描述
nrf_fstorage_sd.c	..\components\libraries\fstorage	FS 后端 nrf_fstorage_sd(使用 Soft device 的程序需要添加)。
crc16.c	..\components\libraries\crc16	CRC16 校验库文件。

nrf_fstorage.c	..\components\libraries\fstorage	FS 驱动文件(裸机程序和使用 Softdevice 的程序均需要添加)。
nrf_atfifo.c	..\components\libraries\atomic_fifo	atomic FIFO。
fds.c	...\components\libraries\fds	FDS 库文件。

7.2. 头文件引用

本例中，对于 FDS 需要加入的文件、头文件包含路径以及 sdk_congfig.h 文件的配置，“实验 7-3”中已经都做好了，因此不需要再进行操作，我们只需在“main.c”文件中引用 FDS 的头文件即可。

代码清单：引用 FDS 头文件

```
//引用 FDS 头文件
#include "fds.h"
```

7.3. 程序流程

FDS 注册事件句柄、初始化以及记录的操作，参见前文的描述即可，本节我们主要描述 BLE 应用下的程序处理流程。

当主机通过 BLE 向开发板发送数据后，程序进入串口透传事件处理函数，在该函数中对接收数据的第一个字节进行判断，根据第一个字节区分命令执行相应的功能，流程如下图所示。

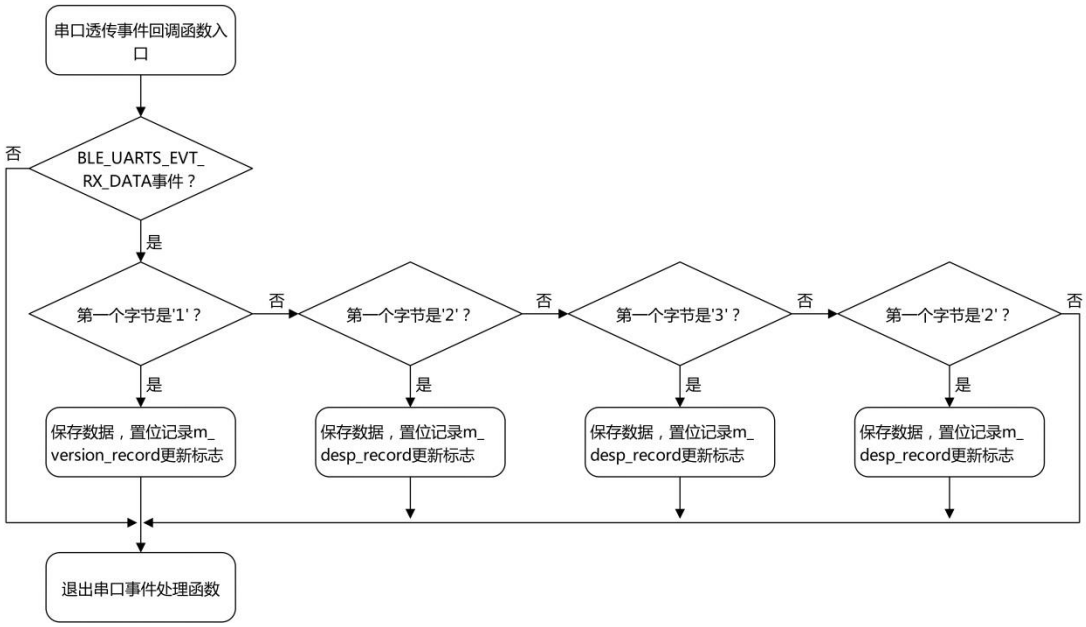


图 10-15：串口透传事件处理函数中的处理流程

主循环中判断各个标志，如标志位置位，则清零标志位后执行功能。

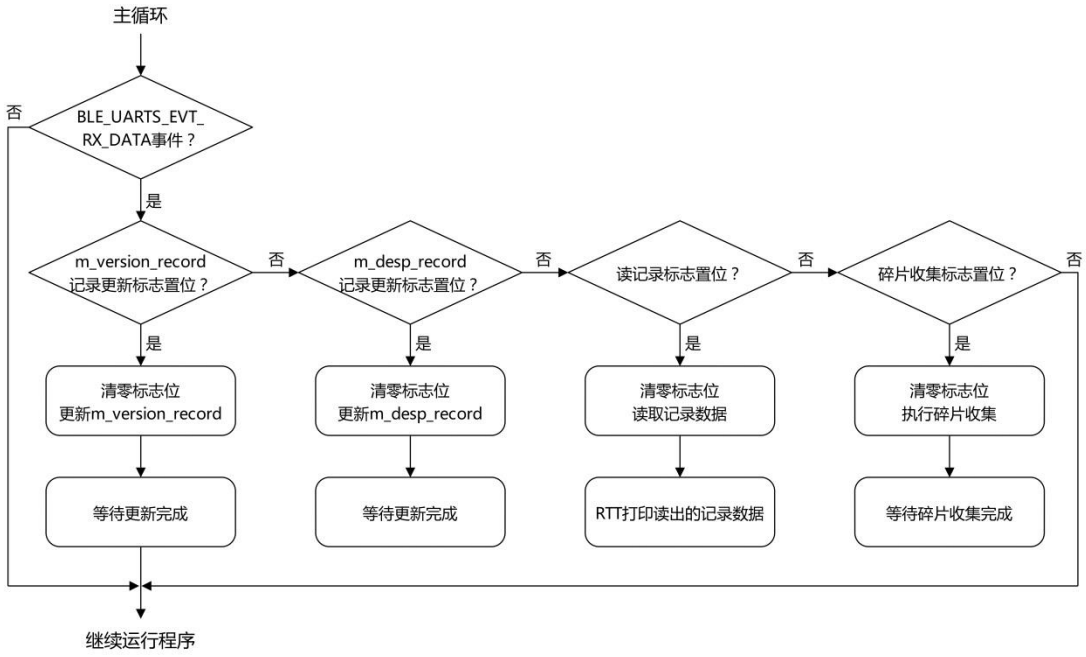


图 10-16：主循环中的处理流程

由于 FDS 的初始化、写、更新等操作都是异步的，因此我们在执行这些操作后需要等待操作完成才能进行 FDS 的其他操作。程序中我们是通过定义的 FDS 忙标志（my_fds_info.busy）来判断 FDS 是否已经执行完操作的，具体步骤是在执行 FDS 异步操作之前，将忙标志置位（FDS 忙），之后执行 FDS 操作等待 FDS 事件，FDS 事件处理函数中接收到对应的事件后，若 FDS 操作成功则清零忙标志，从而保证 FDS 执行异步操作时不会再执行其他 FDS 操作。

7.4. 代码编写

按照上节所述的程序主体流程，编写代码。

本例中为了操作方便，我们声明一个结构体 my_fds_info_t，将 FDS 相关的标志都放到这个结构体，程序清单如下。

代码清单：FDS 异步操作标志的结构体

```
typedef struct
{
    bool version_record_update;    //version_record 记录更新标志
    bool desp_record_update;       //desp_record 记录更新标志
    bool read;        //读记录标志
    bool gc;          //碎片收集标志
    bool busy;        //FDS 忙标志
}my_fds_info_t;
```

串口透传事件回调函数中的处理代码如下，这里根据接收的命令执行操作同时置位相应的标志。

代码清单：串口透传事件回调函数

```
//串口透传事件回调函数，串口透出服务初始化时注册
static void uarts_data_handler(ble_uarts_evt_t * p_evt)
{
    uint32_t i;
    //判断事件类型:接收到新数据事件
    if (p_evt->type == BLE_UARTS_EVT_RX_DATA)
    {
        //如果第一个字节是字符'1'，更新记录 m_version_record
        if(p_evt->params.rx_data.p_data[0] == '1')
        {
            if(p_evt->params.rx_data.length == 17)
            {
                my_fds_info.version_record_update = true;
                for (i = 0; i < 8; i++)
                {
                    device_version.hw_version[i] = p_evt->params.rx_data.p_data[i+1];
                    device_version.sw_version[i] = p_evt->params.rx_data.p_data[8+i+1];
                }
            }
        }
        //如果第一个字节是字符'2'，更新记录 m_fw_record
        else if(p_evt->params.rx_data.p_data[0] == '2')
        {
            my_fds_info.desp_record_update = true;
            memset(deivce_desp.text,0,244);
            for (i = 0; i < p_evt->params.rx_data.length; i++)
                deivce_desp.text[i] = p_evt->params.rx_data.p_data[i+1];
        }
        //如果第一个字节是字符'3'，读取记录 m_version_record 和 m_fw_record 的数据并通
        //过 LOG 输出
        else if(p_evt->params.rx_data.p_data[0] == '3')
        {
            my_fds_info.read = true;
        }
        //如果第一个字节是字符'4'，执行碎片收集
        else if(p_evt->params.rx_data.p_data[0] == '4')
        {
            my_fds_info.gc = true;
        }
        else
        {
        }
    }
    //判断事件类型:发送就绪事件，该事件在后面的实验会用到，当前我们在该事件中翻转指示
```

```
    //灯 D4 的状态，指示该事件的产生
    if (p_evt->type == BLE_UARTS_EVT_TX_RDY)
    {
        nrf_gpio_pin_toggle(LED_4);
    }
}
```

主循环中判断 FDS 各个标志，如果标志置位并且 FDS 空闲的情况下根据标志执行 FDS 操作。

代码清单：主循环

```
while(true)
{
    if(my_fds_info.read == true)//读取记录
    {
        my_fds_info.read = false;
        read_version_record();  //读取记录数据并通过 JLINK-RTT 打印
        read_desp_record();
    }
    //更新记录 m_version_record
    if((my_fds_info.version_record_update == true)&& (my_fds_info.busy == false))
    {
        memset(&tok, 0x00, sizeof(fds_find_token_t)); //清零 tok，从头查找
        my_fds_info.busy = true;
        //在 DEVICE_FILE 文件中查找记录 m_version_record
        rc = fds_record_find(DEVICE_FILE, DEVICE_VERSION_KEY, &desc, &tok);
        if (rc == NRF_SUCCESS)
        {
            my_fds_info.version_record_update = false;
            //更新记录 m_version_record
            rc = fds_record_update(&desc, &m_version_record);
            APP_ERROR_CHECK(rc);
            wait_for_fds_ready();
        }
    }
    //更新记录 m_fw_record
    if((my_fds_info.desp_record_update == true) && (my_fds_info.busy == false))
    {
        memset(&tok, 0x00, sizeof(fds_find_token_t)); //清零 tok，从头查找
        //在 DEVICE_FILE 文件中查找记录 m_fw_record
        rc = fds_record_find(DEVICE_FILE, DEVICE_DESP_KEY, &desc, &tok);
        if (rc == NRF_SUCCESS)
        {
            my_fds_info.busy = true;
```

```
34.          my_fds_info.desp_record_update = false;
35.          //更新记录 m_fw_record
36.          rc = fds_record_update(&desc, &m_desp_record);
37.          APP_ERROR_CHECK(rc);
38.          wait_for_fds_ready();
39.        }
40.     }
41.     //碎片收集
42.     if((my_fds_info.gc == true) && (my_fds_info.busy == false))
43.     {
44.        my_fds_info.gc = false;
45.        my_fds_info.busy = true;
46.        rc = fds_gc();//FDS 碎片回收
47.        APP_ERROR_CHECK(rc);//用错误处理模块检查函数返回值
48.        wait_for_fds_ready();
49.        NRF_LOG_INFO("FDS gc completed");
50.     }
51.     //处理挂起的 LOG 和运行电源管理
52.     idle_state_handle();
53.}
```

FDS 事件处理函数中，接收 FDS 操作的各个事件，接收到事件后查询操作是否成功，若成功则清零 FDS 忙标志，从而通知应用程序 FDS 空闲。

代码清单：FS 事件处理函数

```
1. static void fds_evt_handler(fds_evt_t const * p_evt)
2. {
3.    switch (p_evt->id) //判断事件类型
4.    {
5.       case FDS_EVT_INIT://FDS 初始化事件
6.          if (p_evt->result == NRF_SUCCESS)  //初始化成功
7.          {
8.             my_fds_info.busy = false;  //清零 FDS 忙标志
9.          }
10.         break;
11.      case FDS_EVT_WRITE:  //FDS 写记录事件
12.      {
13.         if (p_evt->result == NRF_SUCCESS) //写记录成功
14.         {
15.            my_fds_info.busy = false;  //清零 FDS 忙标志
16.         }
17.      } break;
18.      case FDS_EVT_UPDATE:  //FDS 更新记录事件
19.      {
```

```
        if (p_evt->result == NRF_SUCCESS) //写记录成功
        {
           my_fds_info.busy = false;//清零 FDS 忙标志
        }
     } break;
     case FDS_EVT_GC:  //FDS 碎片整理事件
     {
        if (p_evt->result == NRF_SUCCESS) //碎片整理成功
        {
            my_fds_info.busy = false;  //清零 FDS 忙标志
        }
     } break;
     default:
       break;
  }
}
```

7.5. 硬件连接

同“实验 10-1：FDS 数据存储-裸机”。

7.6. 实验步骤

1. 为了更好地演示实验，首先全片擦除芯片，以确保 Flash 没有已存储的记录。
2. 解压“…\4：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 10-2：BLE 串口透传中使用 FDS”，将解压后得到的文件夹“ble_app_uarts”拷贝到合适的目录，如“D\nRF52840”。
3. 启动 Keil MDK5.27。
4. 在 Keil MDK5 中执行“Project→Open Project”打开“…\ble_app_uarts\project\mdk5”目录下的工程“ble_app_uarts.uvproj”。
5. 切换到协议栈 target，下载协议栈。
6. 切换到应用程序 target，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
7. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
8. 读取记录：手机上打开 nRF CONNECT 并连接开发板，连接成功后，按照下图所示发送数据，开发板接收到命令后会读取记录的内容并通过 JLINK-RTT 输出。

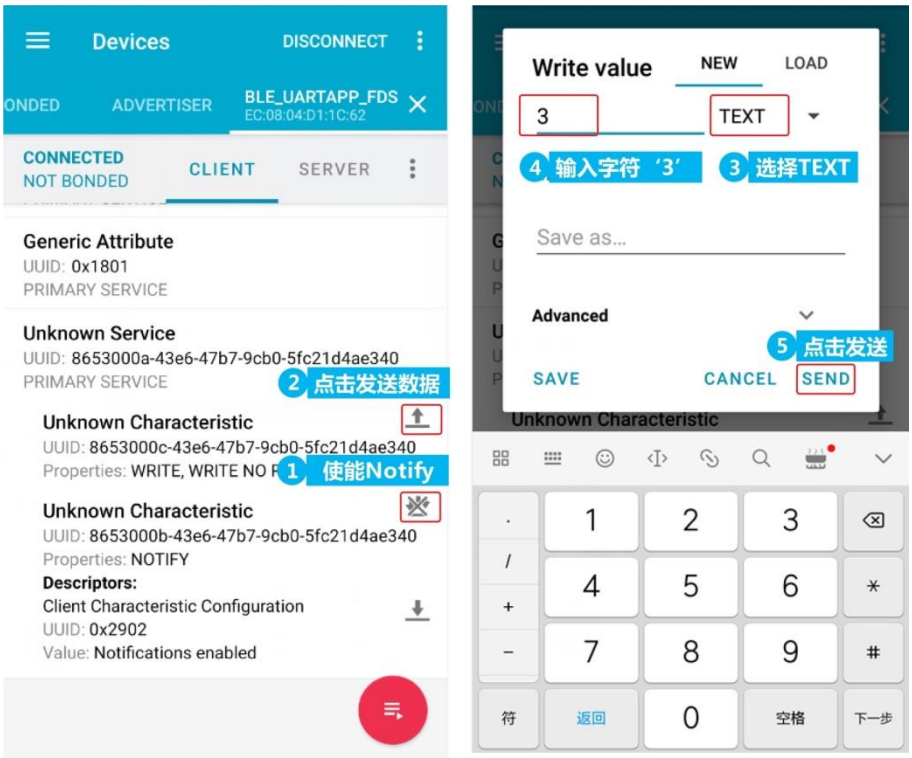


图 10-17：读取 FDS 记录

JLINK-RTT 上可观察到读取的记录数据，如下图所示。

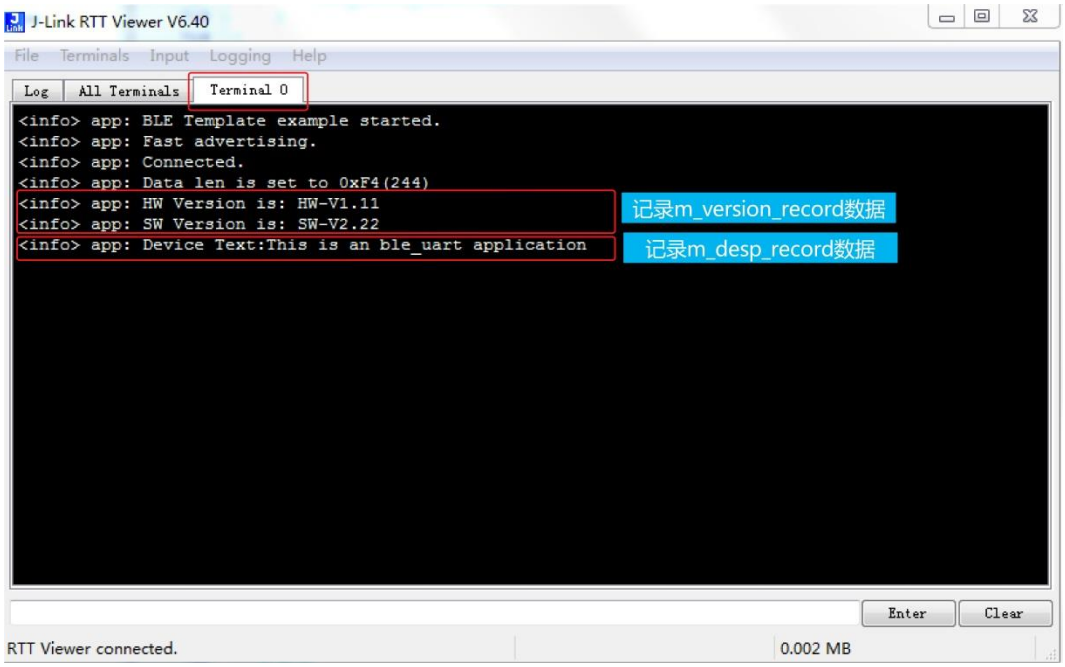


图 10-18：读取的记录数据

9. 更新记录：按照下图所示，分别发送更新记录 m_version_record 和 m_desp_record 的命令。

- 更新记录 m_version_record：发送“1HW-V2.00SW-V3.00”，注意数据长度必须是 17 个字符，第一个字符“1”表示更新记录 m_version_record。
- 更新记录 m_desp_record：发送“2update record”，第一个字符“2”表示更新记录

m_desp_record，数据长度范围：2~244 字节。

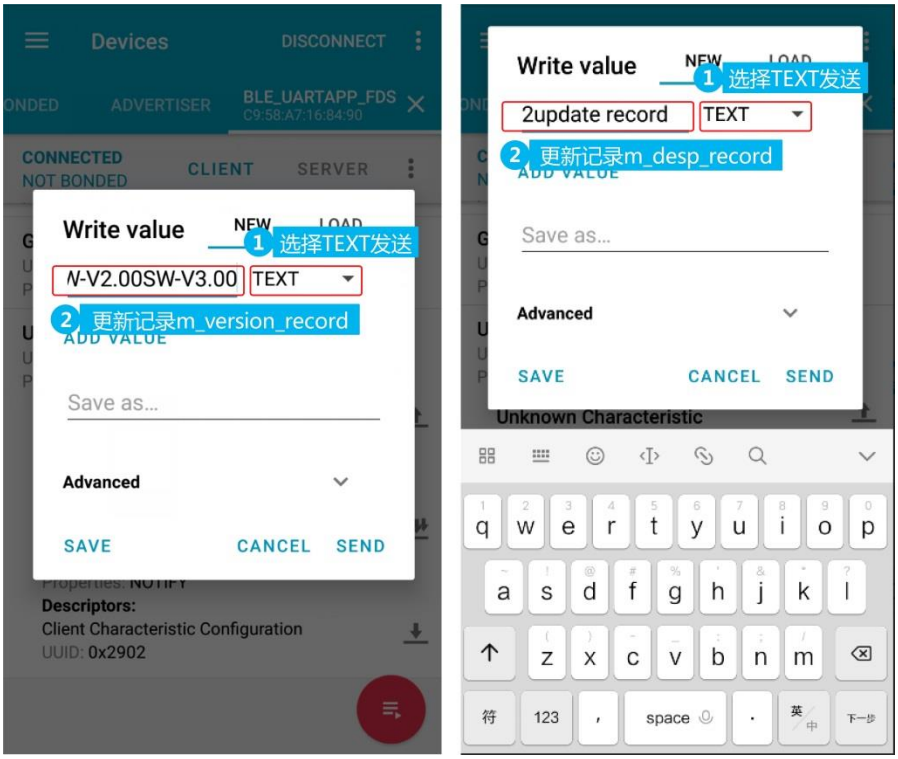


图 10-19：更新记录

更新记录后，再发送命令读取记录，可以看到记录数据已经更新，如下图所示。

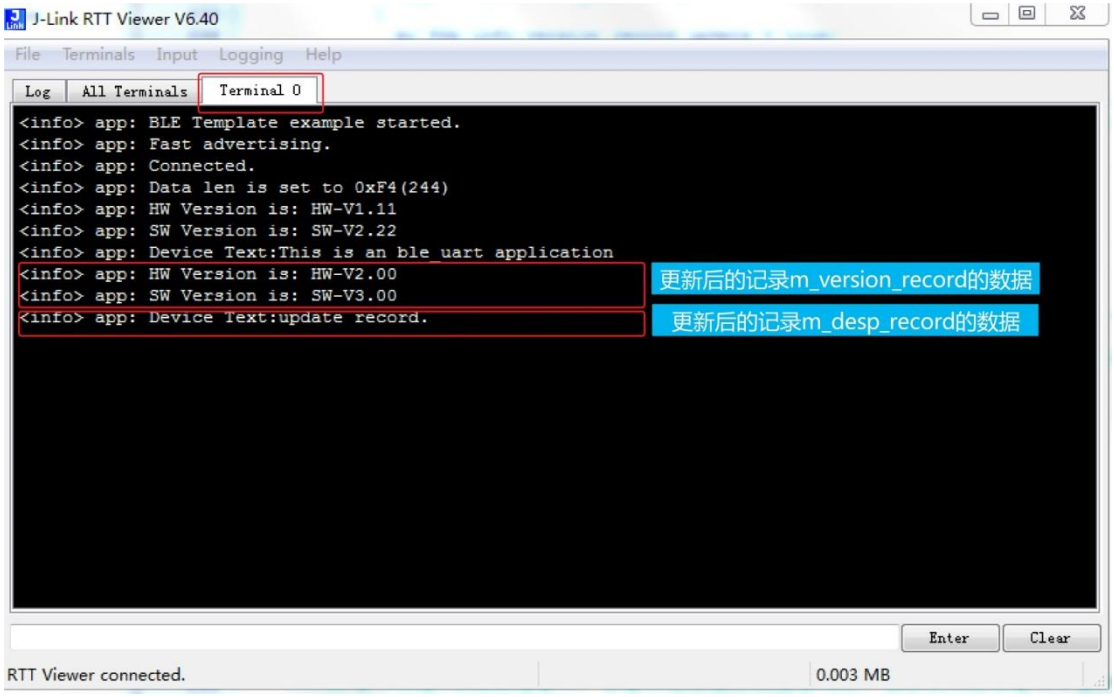


图 10-20：读取的记录数据

10. 碎片收集：nRF CONNECT 上发送字符“4”，执行碎片收集。

第十一章：固件更新（DFU）

1. 学习目的

1. 掌握 DFU 操作环境的搭建以及如何生成 key。
2. 掌握如何使用 mergehex 软件合并 hex 文件。
3. 掌握如何在自己的应用程序中加入 DFU 服务，让应用程序具备 DFU 功能。

- **Secure DFU 学习建议：**Secure DFU 比较复杂，尤其是它的操作环境，需要安装很多软件，进行很多步骤才能搭建完成。如果您是首次接触 Secure DFU，实验过程中一定要仔细，有耐心，首先一定要按照本文档中的步骤将 Secure DFU 跑通，对 Secure DFU 有个直观的了解，然后再进一步去学习。
- **应用程序加入 DFU 功能建议：**DFU 主要依赖 Bootloader，对于应用程序来说，只需加入 DFU 服务让应用程序具备进入 Bootloader 的能力即可，因此，建议应用程序开发完成后再加入 DFU 功能。

2. 概述

2.1. DFU 功能的重要性

产品设计的过程中，往往存在一个让人头疼的问题：当我们开发的产品交付使用后，如果出现 BUG 或者需要增加功能，我们不得不再次打开设备连接编程器，对设备进行编程，更换程序以修补 BUG，虽然这样也能解决问题，但是会导致人力、物力成本的增加，同是会让用户对产品的体验度变差，更甚者，会让用户对产品失去信心。

有没有一种更好的方式来解决这个问题？

当然有 – 让我们的设备具备 DFU 功能！

设备具备 DFU 功能后，这些问题不复存在，当我们发现产品有 BUG 需要修改时、产品的功能需要完善时、产品的程序需要优化时……我们都可以在不对设备进行任何物理操作的情况下 Updata 固件，让产品以最完美的姿态呈现给用户。

2.2. Nordic Secure DFU

Nordic 提供了完整的 DFU 方案，支持对 SD、Bootloader 或 APP 升级，经过多个版本的改进，当前的版本称为 Secure DFU，即升级时通过一个 key 认证，只有具备相同的 key 才能对设备进行升级，从而保证只有我们自己可以对自己的设备进行升级，别人因不知道 key，是无法恶意操作我们的设备进行升级的。

Secure DFU 已经发展成一个很成熟的 DFU 方案，在我们的工程中，我们可以直接使用 Secure DFU，甚至我们不用过于关心 Bootloader 实现的细节而只需在应用程序中增加 DFU 服务的代码即可让工程具备 DFU 的功能。

DFU 的核心是 Bootloader，DFU 接收新固件、更新应用程序都由 Bootloader 完成。程

序启动后会首先执行 Bootloader，Bootloader 会查询相关标志，根据标志决定去执行应用程序还是停留在 Bootloader。

Nordic 的 Bootloader 工程在 SDK 中的路径是："…\nRF5_SDK_16.0.0_98a08e2\examples\dfu\bootloader_secure"。nRF52840 的固件更新可以通过：BLE、串口和 USB 三种方式实现，每种方式都有 Release 和 Debug 两个版本，如下图所示，它们的区别是：

- Release：升级时会对发送的文件的版本进行验证。
- Debug：不会进行版本验证，方便调试。

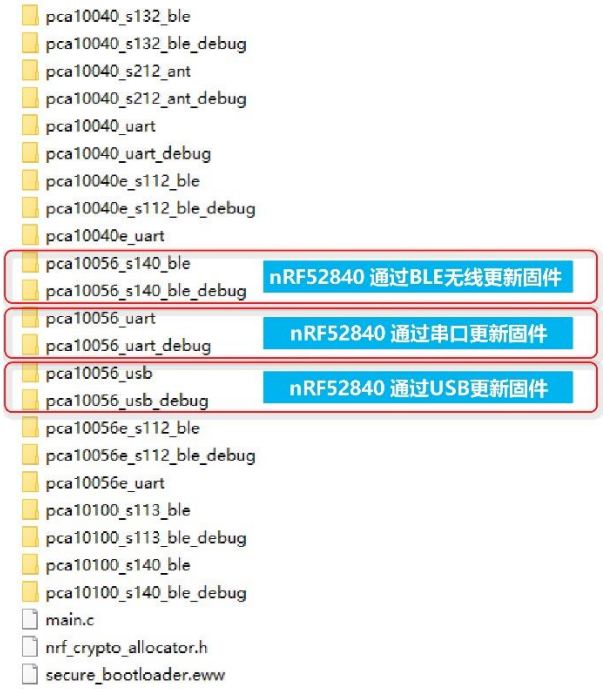


图 11-1：Bootloader 版本

✧ 注：本章中，我们使用的 Bootloader 是 Release 版本。

2.3. 相关术语、缩略词

1) DFU：全称为 Device Firmware Update，设备固件升级。
2) MBR：全称为 Master Boot Record，主引导记录。
3) OTA：Over the Air，空中下载。
4) Bootloader：引导加载程序。
5) SD：Softdevice，Nordic 提供的 BLE 协议栈。
6) BLE：Bluetooth Low Energy，蓝牙低功耗。一种低功耗、低成本、短距离、可互操作的鲁棒性无线技术，工作在免许可的 2.4GHz ISM 射频频段。
7) APP：应用程序。注意在这里 APP 不是指的安卓手机上的 APP，而是 nRF52840 固件程序中相对 Bootloader 而言的，是用户实现自己功能的程序。

3. 需要的工具软件

Secure DFU 除了需要开发指南上册《第三章：搭建开发环境》章节安装的软件之外，

还需要用到下表所示的软件工具。

表 11-1：需要的工具软件

软件工具	描述
gcc-arm-none-eabi-7-2018-q2-update-win32	编译 micro-ecc。32 位和 64 位机均安装该版本。
MinGW	Windows 平台下实现 Makefile。
micro-ecc 源码“micro-ecc-master.zip”	Nordic 的 DFU 中采用了 micro-ecc 实现 ECDSA 算法，需要用到 micro-ecc 源码。
python-2.7.14：32 位机安装 **python-2.7.14.amd64.msi：64 位机安装**	pc-nrfutil 需要用到 python-2.7 的环境。
pc-nrfutil.zip	Nordic 发布的 PC 端的工具，支持 DFU 和加密功能。

■ **所用软件工具下载链接：**

1. gcc-arm-none-eabi 下载链接：https://launchpad.net/gcc-arm-embedded/+announcement/15015
2. MinGW 下载链接：https://sourceforge.net/projects/mingw/files/latest/download?source=files
3. micro-ecc 下载链接：https://github.com/kmackay/micro-ecc
4. Python 下载链接：https://www.python.org/downloads/

✧ 注：资料包里已经下载好了 DFU 所需的软件工具。读者无需自己下载，直接安装即可。

4. 操作步骤(详细版)

4.1. 安装 gcc-arm-none-eabi

1. 双击“gcc-arm-none-eabi-7-2018-q2-update-win32”，弹出 gcc-arm 的安装向导，单击[OK]。

图 11-2：启动安装

2. 点击【下一步】，如下图所示。

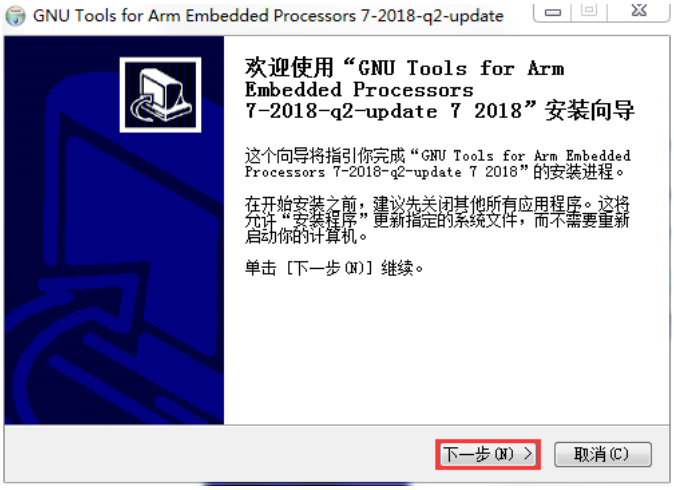


图 11-3：开始安装

3. 点击【我接受】，如下图所示。

图 11-4：接受安装协议

4. 选择安装路径，如下图所示。

此处，可以根据自己的需要选择安装路径，本文档设置的安装路径是默认安装路径，即安装在 C 盘。

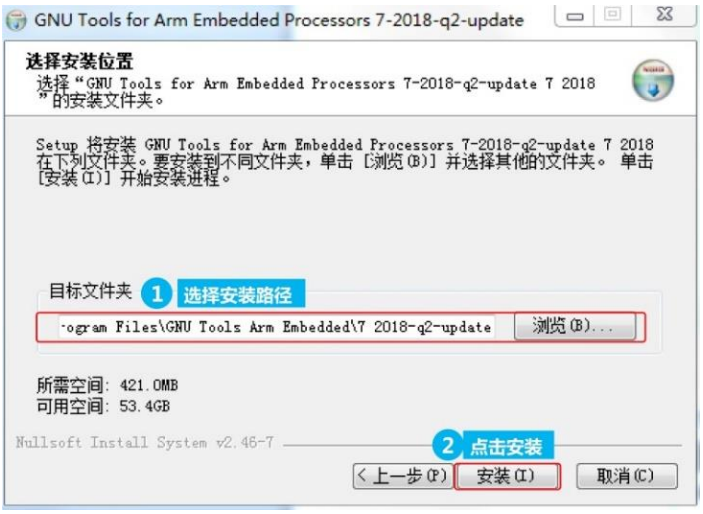


图 11-5：选择安装路径

5. gcc-arm 开始安装，等待 gcc-arm 安装完成，如下图所示。

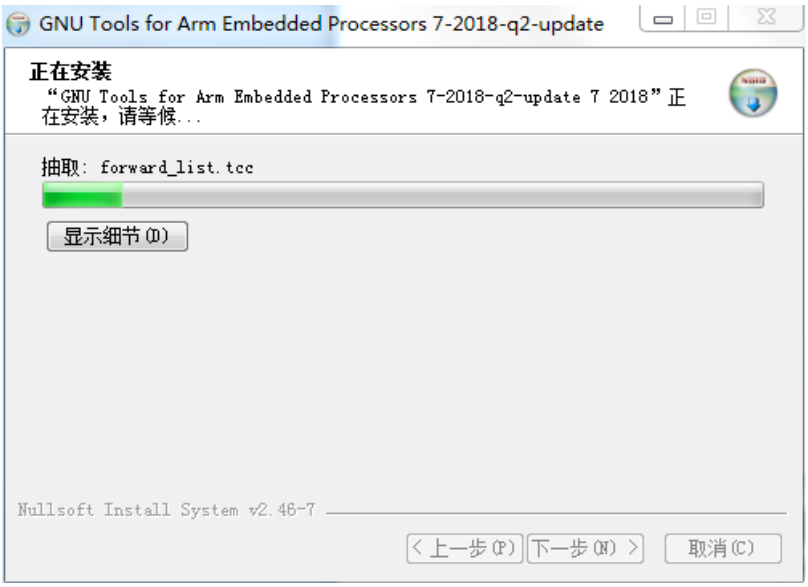


图 11-6：等待安装完成

6. 点击【完成】完成安装，如下图所示。

图 11-7：完成 gcc-arm 安装

4.2. 安装 MinGW

4.2.1. 安装

1. 双击 “mingw-get-setup.exe”，弹出 MinGW 的安装向导，单击【Install】。

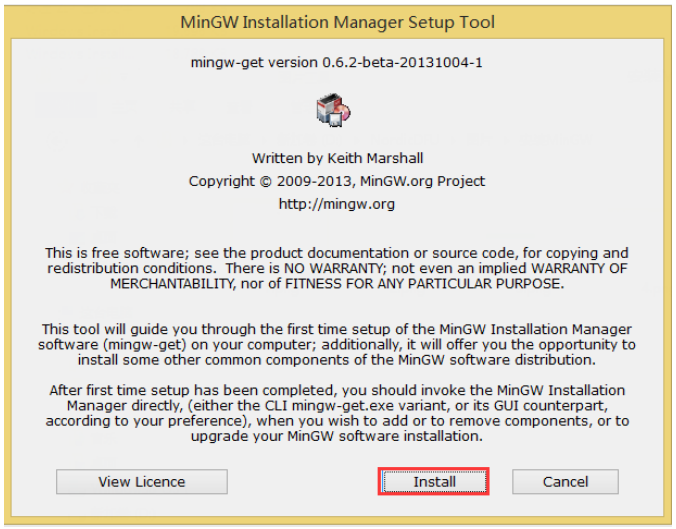


图 11-8：安装 MinGW

2. 选择安装路径即相关配置(这里我们使用默认的安装配置即可)，点击【Continue】，如下图所示。

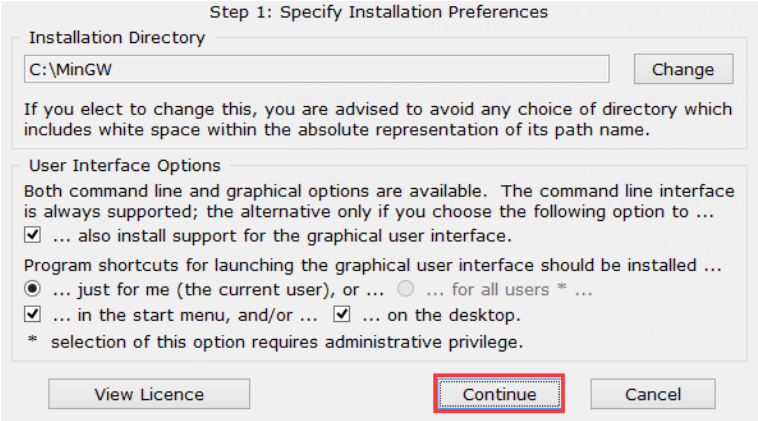


图 11-9：设置安装配置

3. 等待安装完成。

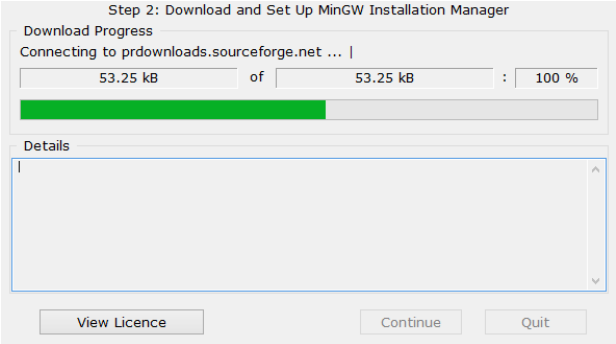


图 11-10：等待安装完成

4. 选择下图所示的 3 项，弹出的对话框中点击【Mark for installation】

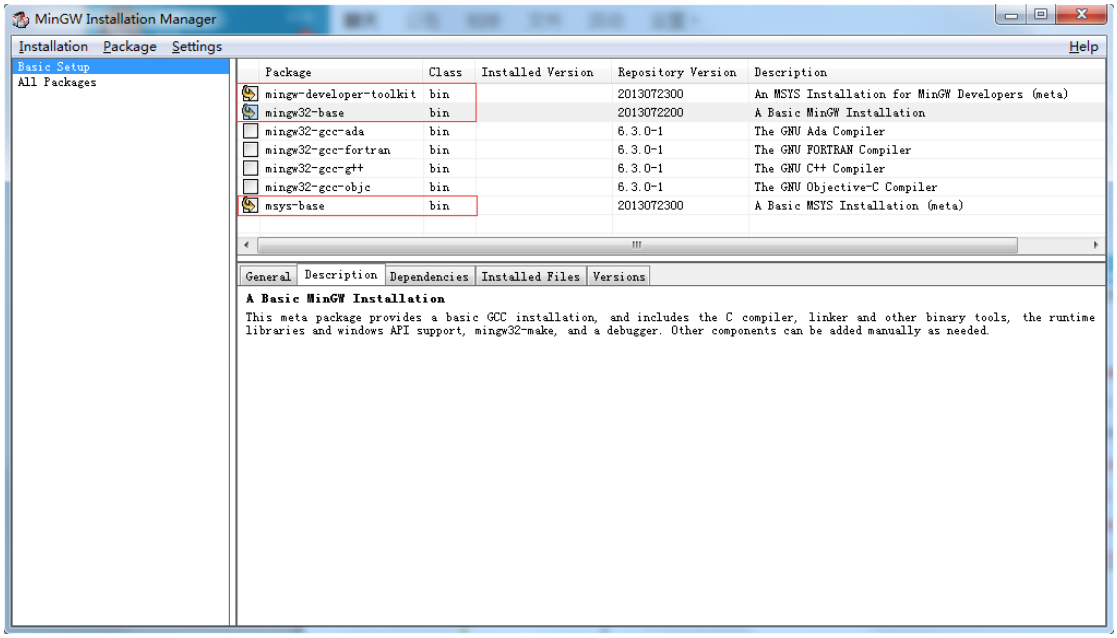


图 11-11：选择安装的组件

5. 点击【installation】，弹出的下拉列表中点击【apply change】，如下图所示。

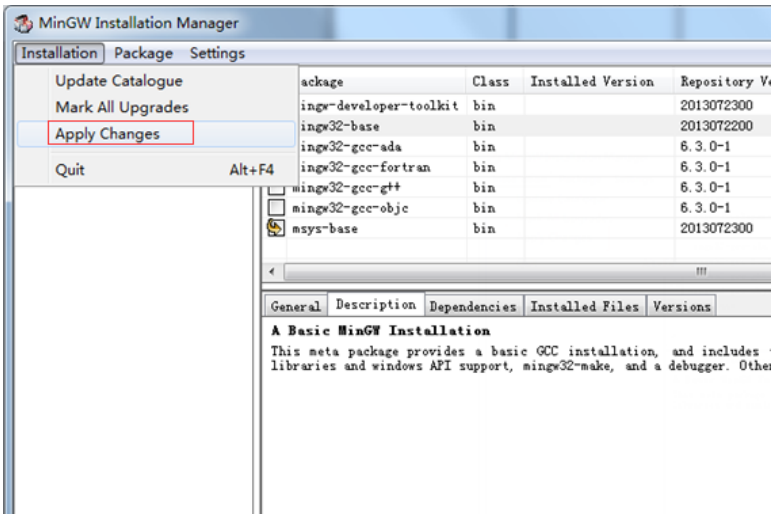


图 11-12：安装组件

6. 等待组件安装完成，完成后如下图所示。

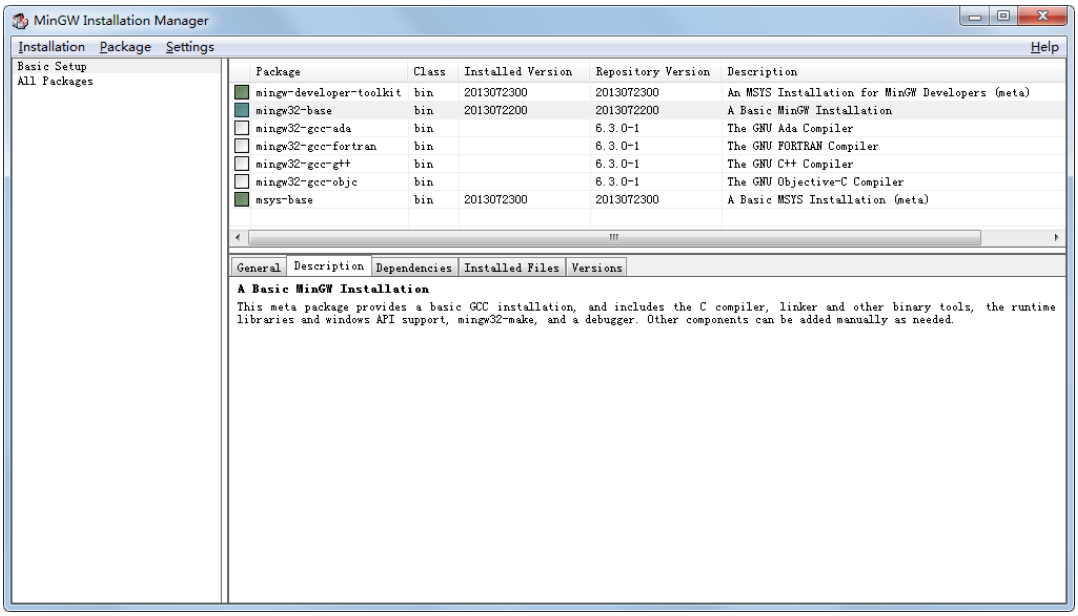


图 11-13：组件安装完成

4.2.2. 配置环境变量

MinGW 安装完成后，需要在系统环境变量的“path”中加入 MinGW 的路径，加入的步骤如下：

1. 对着“我的电脑”右键选择“属性”。
2. 点击“高级系统设置”。
3. 点击“环境变量(N) …”，打开环境变量设置串口，找到系统环境变量中的“path”，然后点击【编辑】，如下图所示。

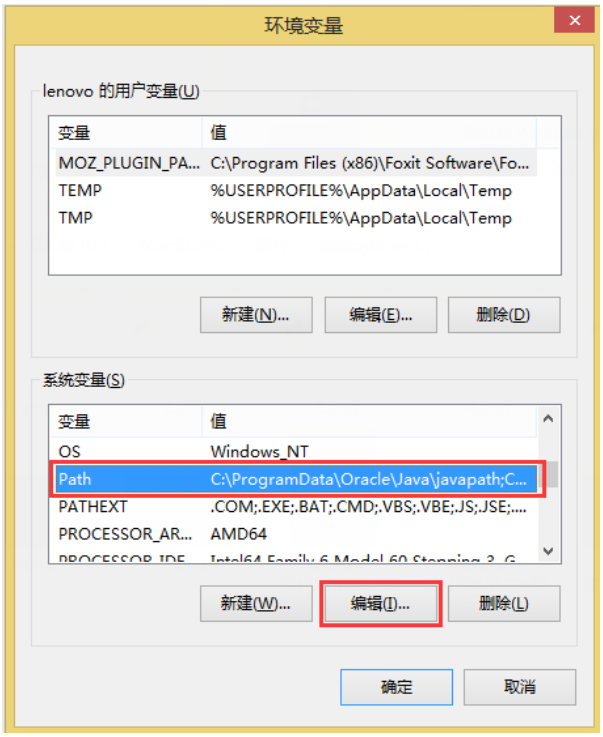


图 11-14：配置环境变量

4. 加入“C:\MinGW\bin”路径，路径之间用英文输入法下的“;”隔开，如下图所示。

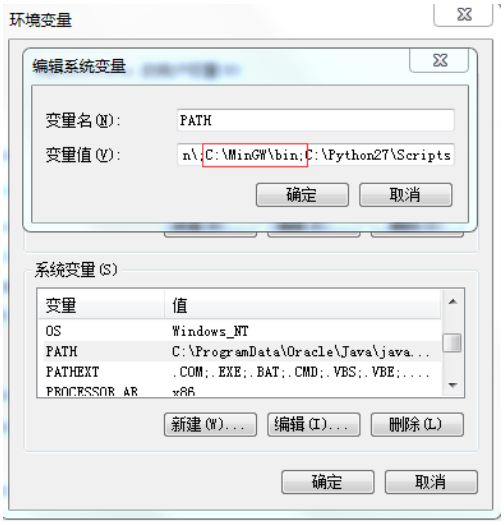


图 11-15：加入 MinGW 路径

4.2.3. 验证安装是否成功。

1. 同时按下“窗口键 ＋R”，打开运行窗口，输入“cmd” 打开命令窗口。

图 11-16：窗口键 ＋ R 键

2. 命令窗口中输入“gcc -v”，注意**空格**和 **V 要小写**。查询 GCC 版本号，出现如下信息，表示安装成功，如下图所示。

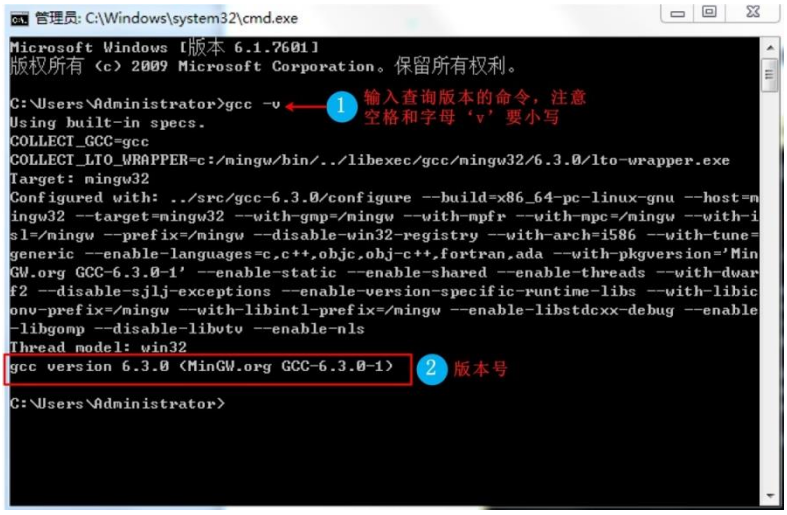


图 11-17：验证安装

4.3. 工程中加入 micro-ecc 源码

解压“micro-ecc-master.zip”到“nRF5_SDK_16.0.0_98a08e2\external\micro-ecc\micro-ecc”目录下，并将文件夹名称改为“micro-ecc”，如下图所示。

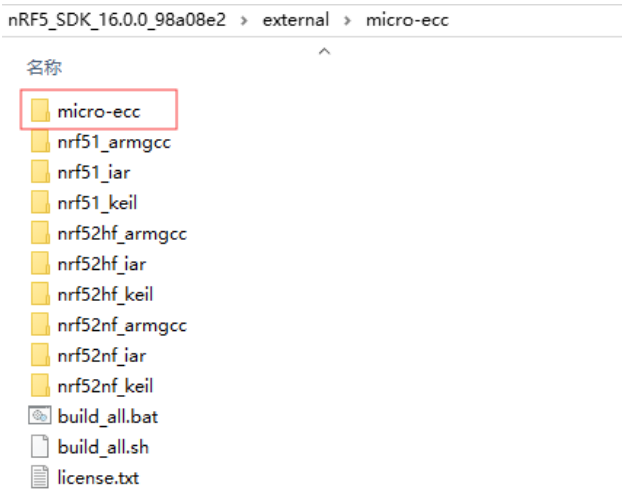


图 11-18：工程中加入 micro-ecc 源码

4.4. 生成 micro_ecc_lib_nrf52.lib

4.4.1. 生成步骤

1. 启动 msys.bat

   msys.bat 位于“C:\MinGW\msys\1.0”，如下图所示，双击打开 msys.bat。

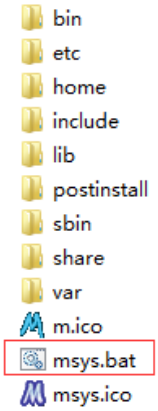


图 11-19：打开 msys.bat

2. 切换到“…\nRF5_SDK_16.0.0_98a08e2\external\micro-ecc\nrf52hf_keil\armgcc”目录，本例中切换到“D:\DFU\nRF5_SDK_16.0.0_98a08e2\external\micro-ecc\nrf52hf_keil\armgcc”目录。

   切换目录时用 cd 命令，可以直接拖拽 armgcc 文件夹到命令窗口，然后修改，这样就可以避免输入路径的麻烦。

图 11-20：切换目录

3. 输入 make，编译生成 micro_ecc_lib_nrf52.lib，如下图所示。

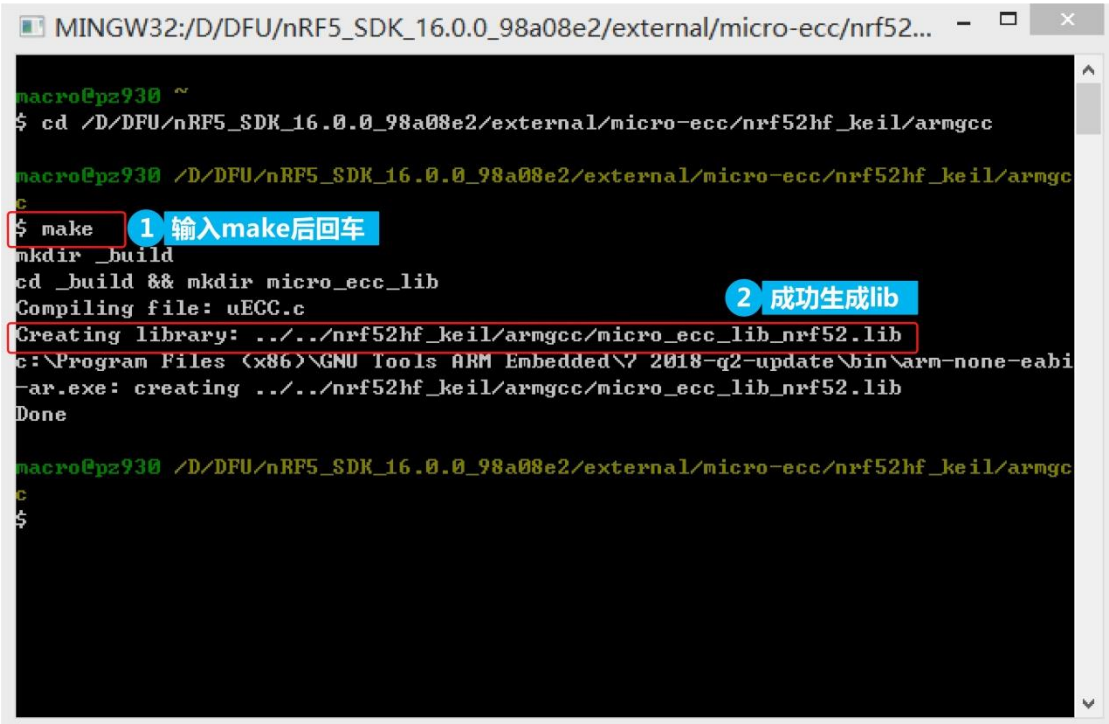


图 11-21：生成 micro_ecc_lib_nrf52.lib

4. 编译成功后，生成的 micro_ecc_lib_nrf52.lib 文件位于 armgcc 文件夹，如下图所示。

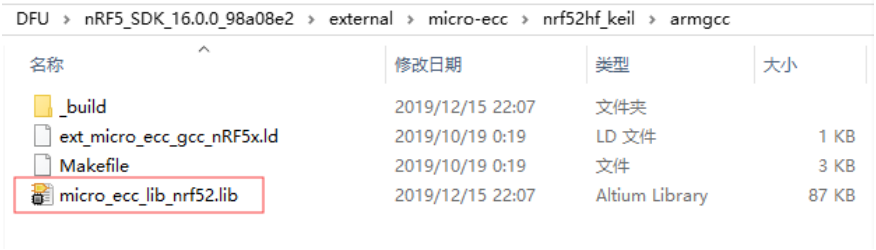


图 11-22：micro_ecc_lib_nrf51.lib 保存的路径

4.4.2. 该步骤问题总结

1. Make 的时候提示“找不到文件”，如下图所示。

```
MINGW32:/D/DFU/nRF5_SDK_15.2.0_9412b96/external/micro-ecc/nrf52hf_keil/armgcc

Administrator@WIN-20161227EQV ~
$ cd /D/DFU/nRF5_SDK_15.2.0_9412b96/external/micro-ecc/nrf52hf_keil/armgcc

Administrator@WIN-20161227EQV /D/DFU/nRF5_SDK_15.2.0_9412b96/external/micro-ecc/
nrf52hf_keil/armgcc                    make时提示找不到文件
$ make
make: C:/Program Files (x86)/GNU Tools ARM Embedded/6 2017-q2-update/bin/arm-non
e-eabi-gcc: Command not found
Cannot find: 'C:/Program Files (x86)/GNU Tools ARM Embedded/6 2017-q2-update/bin
/arm-none-eabi-gcc'.
Please set values in: "/D/DFU/nRF5_SDK_15.2.0_9412b96/components/toolchain/gcc/M
akefile.windows"
according to the actual configuration of your system.
../../../../components/toolchain/gcc/Makefile.common:129: *** Cannot continue.
Stop.

Administrator@WIN-20161227EQV /D/DFU/nRF5_SDK_15.2.0_9412b96/external/micro-ecc/
nrf52hf_keil/armgcc
$
```

图 11-23：make 时找不到文件

- **原因**：makefile.windows 中 gcc-arm-none-eabi 路径配置的不对。
- **解决办法**：用记事本打开“...\nRF5_SDK_16.0.0_98a08e2\components\toolchain\gcc”目录下的 makefile.windows 文件，如下图所示。

DFU › nRF5_SDK_16.0.0_98a08e2 › components › toolchain › gcc

名称	修改日期	类型
dump.mk	2019/10/18 23:34	MK 文件
gcc_nrf51_common.ld	2019/10/18 23:34	LD 文件
Makefile.common	2019/10/18 23:34	COMMON 文件
Makefile.posix	2019/10/18 23:34	POSIX 文件
Makefile.windows	2019/10/18 23:34	WINDOWS 文件

图 11-24：makefile.windows 文件所在目录

检查版本号是不是“7.3.1”，检查路径是不是和 gcc-arm-none-eabi 实际安装的路径一致。注意：有的计算机默认安装的路径是 C:\Program Files\GNU Tools ARM Embedded\6 2017-q2-update，而该文件中的默认设置的路径是 C:/Program Files (x86)/ GNU Tools ARM Embedded\7 2018-q2-update/bin/，要修改成和实际安装路径一致才能执行 make。

```
Makefile.windows - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H) 1 必须和实际安装路径一致
GNU_INSTALL_ROOT := C:/Program Files (x86)/GNU Tools ARM Embedded/7 2018-q2-update/bin/
GNU_VERSION := 7.3.1  2 版本号
GNU_PREFIX := arm-none-eabi
```

图 11-25：检查版本号和路径

到这里，我们已经得到了 micro_ecc_lib_nrf52.lib 文件，接下来，需要使用 nrfutil 软

件生成 key。

4.5. 安装 python-2.7

4.5.1. 安装

1. 双击 “python-2.7.14.amd64.msi”，弹出 python 的安装向导，单击【Install】。

✧ **注意：**python-2.7 有 32 位 64 位版本的，安装时根据自己的电脑选择对应的安装文件。

- python-2.7.14.msi：32 位机安装。
- python-2.7.14.amd64.msi：64 位机安装。

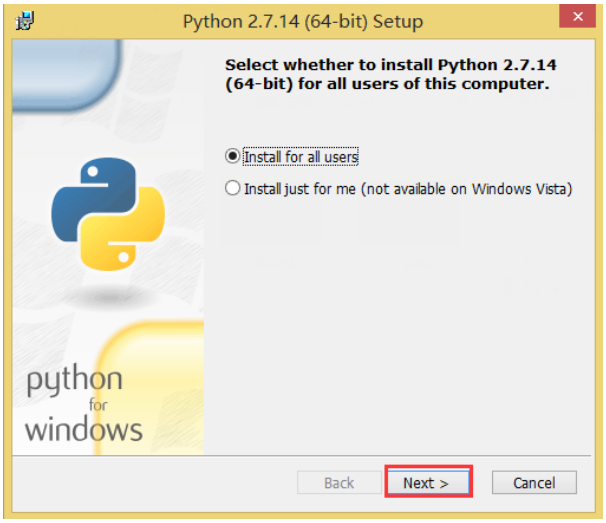


图 11-26：python 安装

2. 选择 python 安装的路径(安装路径一般都选择安装在 C 盘)，点击【next】，如下图所示。

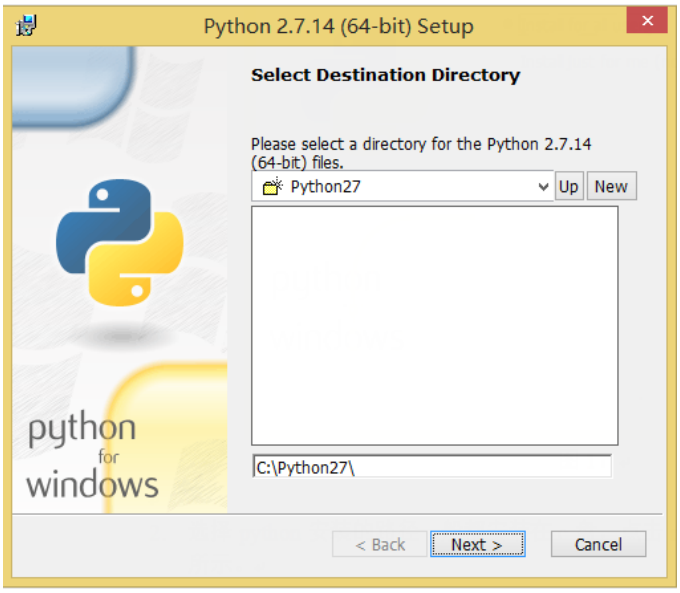


图 11-27：选择 python 安装的路径

3. 选择需要安装的文件，这里选择默认的，即全部安装，点击【next】，如下图所示。

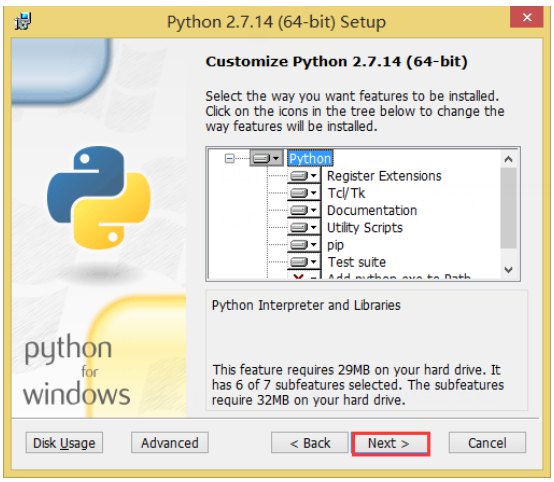


图 11-28：选择需要安装的文件

4. 等待 python 安装完成，如下图所示。

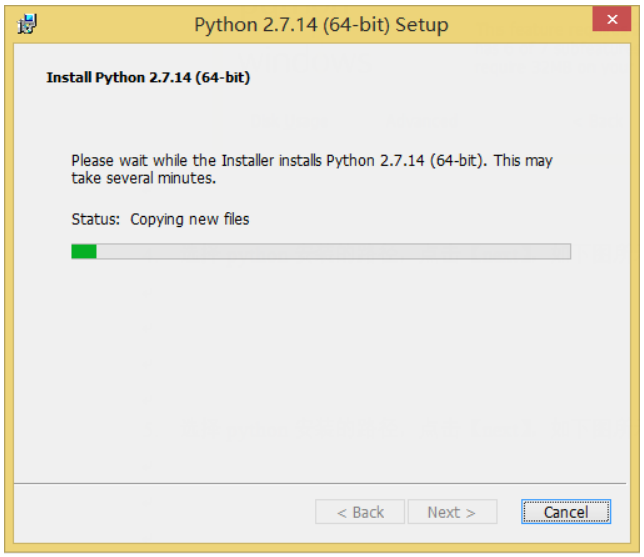


图 11-29：等待 python 安装完成

5. 点击【Finish】完成安装，如下图所示。

图 11-30：完成安装

4.5.2. 配置环境变量。

python 安装完成后，需要在系统环境变量的“path”中加入 python 的路径，加入的步骤如下：

1. 对着“我的电脑”右键选择“属性”。
2. 点击“高级系统设置”。
3. 点击“环境变量(N) …”，打开环境变量设置串口，找到系统环境变量中的“path”，然后点击【编辑】，如下图所示。

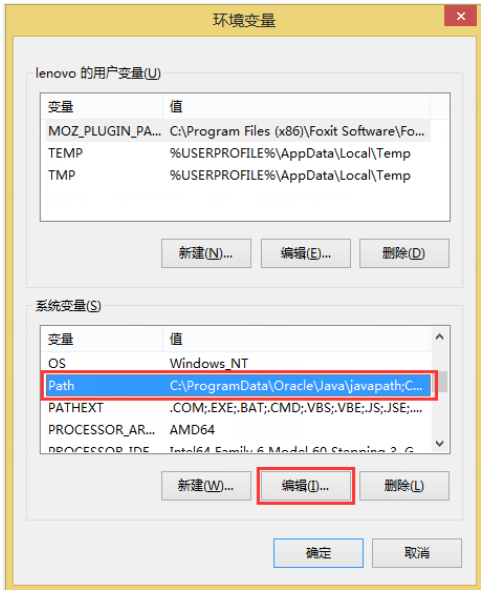


图 11-31：配置环境变量

4. 加入“C:\Python27”和“C:\Python27\Scripts”两个路径，路径之间用英文输入法下的“;”隔开，如下图所示。

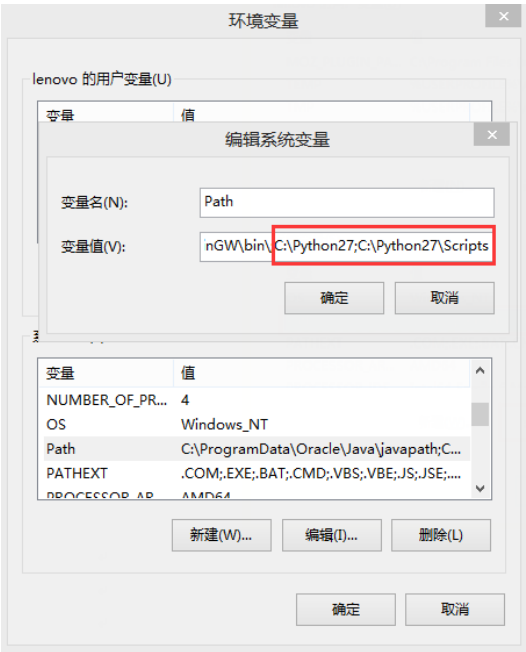


图 11-32：配置 Python 路径

4.5.3. 验证安装是否成功。

1. 同时按下“窗口键 +R”，打开运行窗口，输入“cmd” 打开命令窗口。
2. 输入“python -V”，注意**空格**和 **V 要大写**。查询 python 版本号，如能正确查询到版本号，表示安装成功，如下图所示。

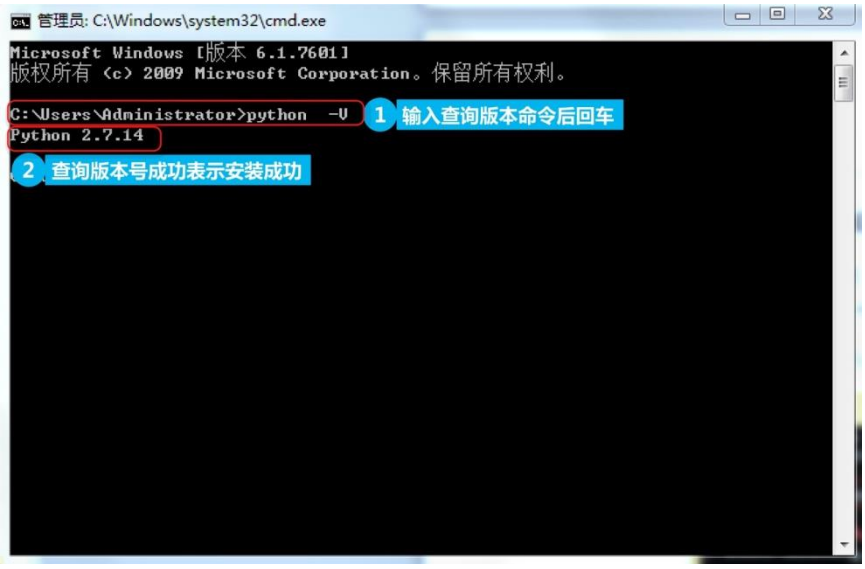


图 11-33：验证安装

4.5.4. 该步骤问题总结

1. Python 安装完成后，查询 python 版本号时出现如下图所示信息。

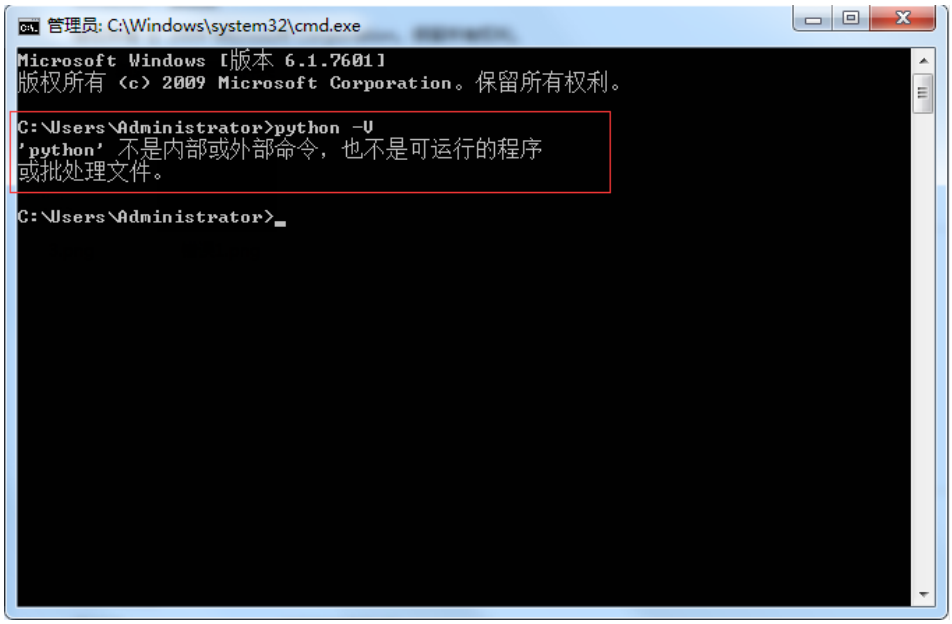


图 11-34：查询 python 版本号出错

- **原因：**系统环境变量路径配置的不对，没有将 Python 的 2 个路径都加入到系统环境变量的 path 中。
- **解决办法：**系统环境变量的 path 中加入 2 个路径：

“C:\Python27”和“C:\Python27\Scripts”，并且路径之间用英文输入法时的“;”隔开。

4.6. 安装 pc-nrfutil 和生成 key

4.6.1. 安装 pc-nrfutil

1. D 盘根目录下新建一个名为 DFU 的文件夹，解压“pc-nrfutil.zip”到该文件夹，打开“…\pc-nrfutil”目录，鼠标放在文件空白的地方，shift+右键执行“在此处打开命令窗口(W)”。
2. 命令窗口中输入“python setup.py install”，回车。开始通过网络安装，当出现如下图所示信息时表示安装完成。

   注意：安装时需要保持联网，安装会占用一些时间，请耐心等待安装完成。

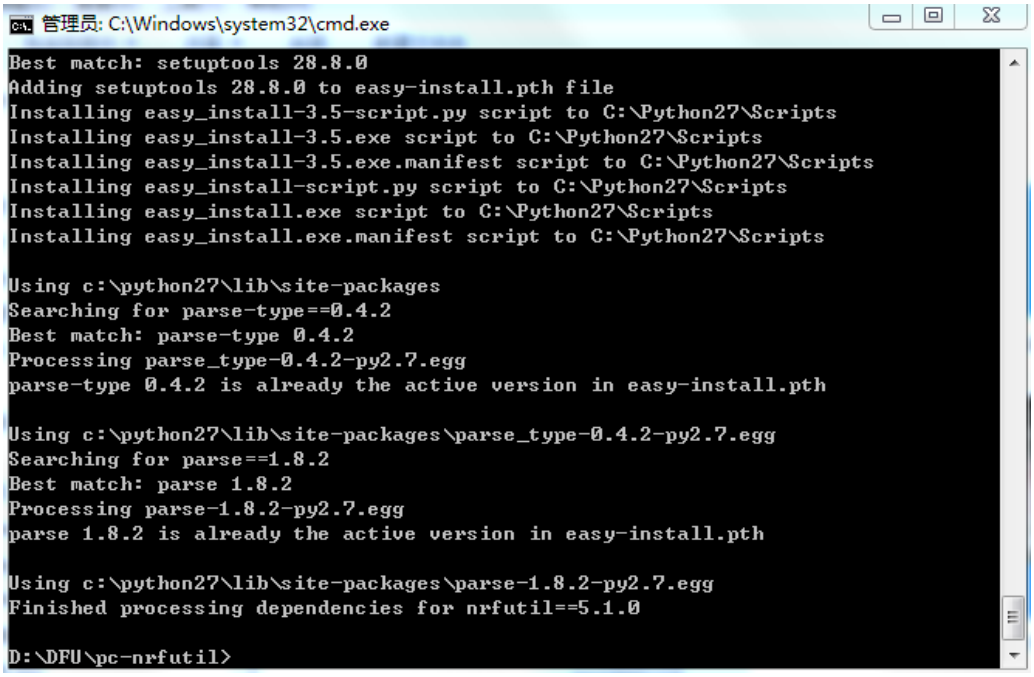


图 11-35：安装完成

4.6.2. 生成 private key 和 public key

1. 生成 private.key。鼠标放在 D:\DFU 文件夹内空白的地方，shift+右键执行“在此处打开命令窗口(W)”，命令窗口中输入“nrfutil keys generate D:\DFU\private.pem”，回车。

✧ **注意：**红字部分是我们用来存放生成的 private.key 的路径，这个路径可以自己设置，但是必须是有效的路径。本例中执行此命令后，生成的 private.pem 存放在“D:\DFU”目录下。

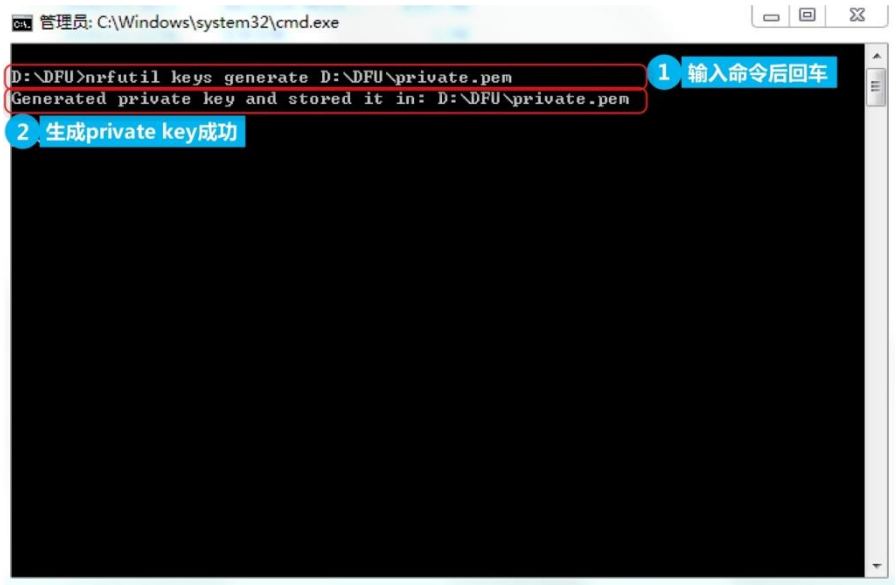


图 11-36：生成 private key

2. 使用 private.pem 生成 public key。命令窗口中输入“nrfutil keys display --key pk --format code D:\DFU\private.pem --out_file D:\DFU\dfu_public_key.c”，回车。

✧ **注意：**红字部分是 private.pem 的存放路径和生成的 dfu_public_key.c 的存放路径，private.pem 是我们上一步中生成的，dfu_public_key.c 的存放路径可以自己设置，但是必须是有效的路径。本例中执行此命令后，生成的 dfu_public_key.c 存放在“D:\DFU”目录下。

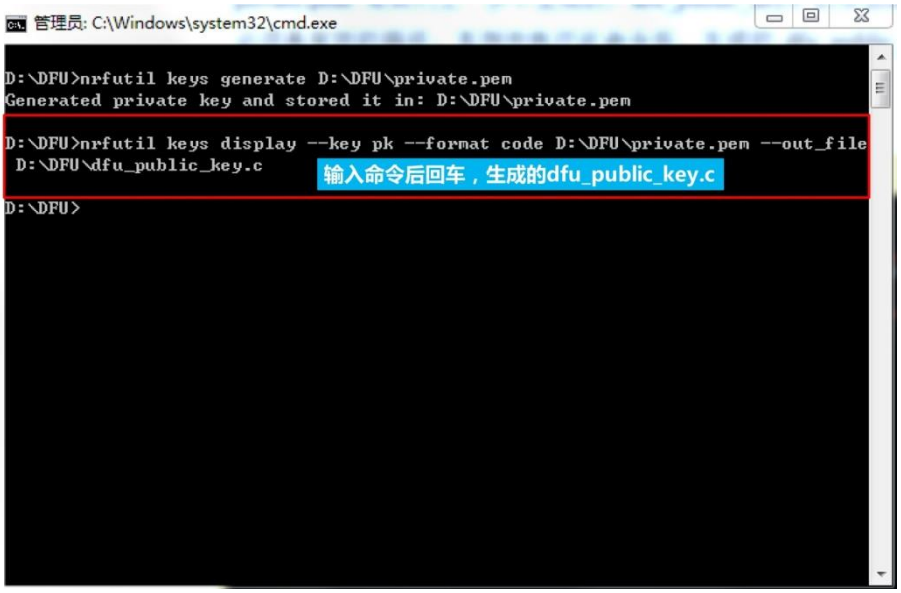


图 11-37：生成 dfu_public_key.c

3. 使用生成的“dfu_public_key.c”替换“…\nRF5_SDK_16.0.0_98a08e2\examples\dfu”目录下的“dfu_public_key.c”如下图所示。

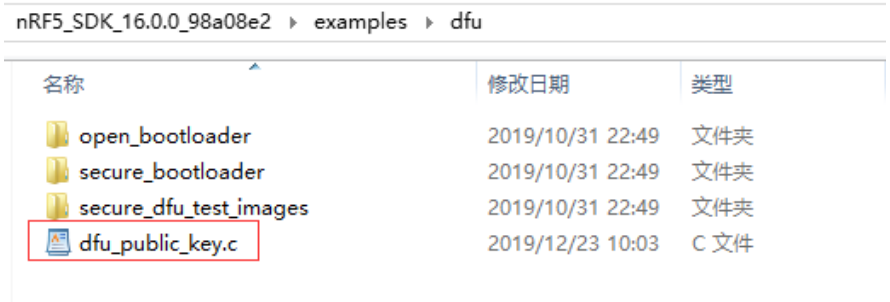


图 11-38：替换 dfu_public_key.c

4.7. 编译和烧写 Bootloader

4.7.1. 编译 Bootloader

用 MDK5 打开“…\nRF5_SDK_16.0.0_98a08e2\examples\dfu\secure_bootloader\pca10056_ble\arm5_no_packs”目录下的工程“secure_bootloader_ble_s140_pca10056.uvprojx”，点击编译按钮编译工程。

如果我们前面的步骤操作都正确的话，这里看以看到 Bootloader 编译成功，即无错误无警告，如下图所示。

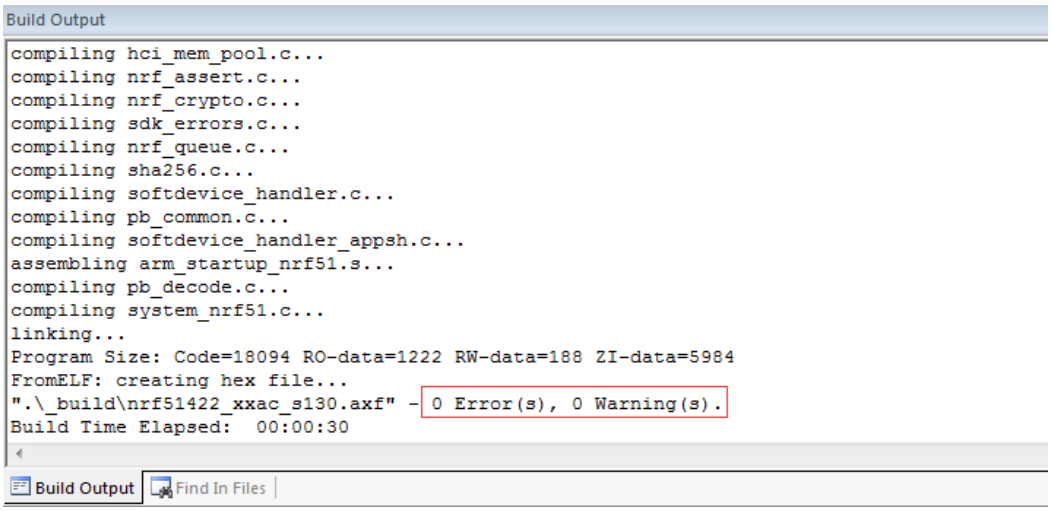


图 11-39：编译 Bootloader

4.7.2. 烧写 Bootloader

1. 切换到 Target“s132_nrf52_6.1.0_softdevice”，设置编程算法后，点击 下载协议栈。
2. 切换到 Target“nrf52840_xxaa_s140”，点击 下载应用程序。

烧写成功后：开发板上的 D1 和 D2 指示灯点亮表示程序正常运行。手机上打开“nRF Connect”，点击“SCAN”即可扫描到名为“DfuTarg”的设备，如下图所示。

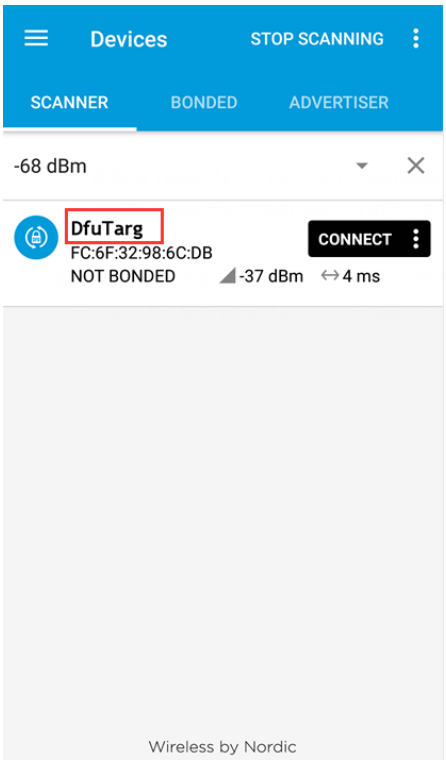


图 11-40：手机扫描到“DfuTarg”

4.8. 升级文件打包

这里以 SDK 中的 “ble_app_buttonless_dfu”为例说明如何打包升级文件(仅升级应用程序)。

buttonless_dfu 是 Nordic 提供的一个通过发送命令即可让设备进入 Bootloader 的示例应用程序(Nordic 早期的 DFU，需要通过启动时按下按键让设备进入 Bootloader，buttonless_dfu 是指通过蓝牙发送命令让设备进入 Bootloader，无需使用按键)。

1. 编译 ble_app_buttonless_dfu：MDK 打开“…\nRF5_SDK_16.0.0_98a08e2\examples\ble_peripheral\ble_app_buttonless_dfu\pca10056\s140\arm5_no_packs”目录下的“ble_app_buttonless_dfu_pca10056_s140.uvprojx”工程，编译工程，编程成功后生成“nrf52840_xxaa.hex”文件，该文件路径为“…\nRF5_SDK_16.0.0_98a08e2\examples\ble_peripheral\ble_app_buttonless_dfu\pca10056\s140\arm5_no_packs_build”。
2. 新建一个文件夹，如“D:\dfufile”，将 ble_app_buttonless_dfu 编译后生成的 hex 文件“nrf52840_xxaa.hex”和“private.pem”文件拷贝到该文件夹里面。
3. 鼠标放在 dfufile 文件夹里面空白的地方，shift+右键执行“在此处打开命令窗口(W)”。
4. 命令窗口中输入：nrfutil pkg generate --hw-version 52 --application-version 1 --application nrf52840_xxaa.hex --sd-req 0xCA --key-file private.pem dfufile.zip，输入完成后回车，如下图所示。

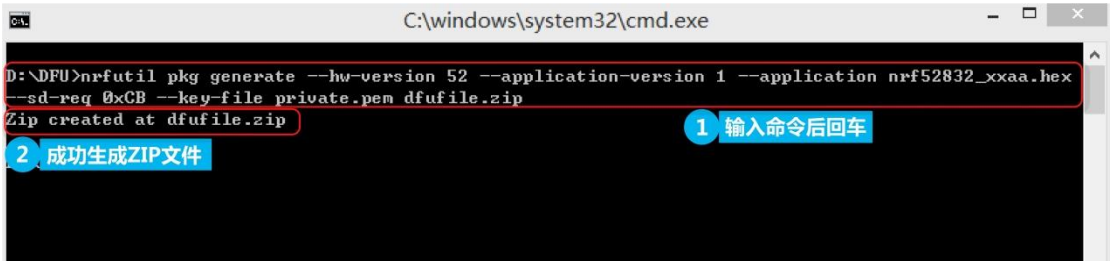


图 11-41：生成 ZIP 文件

这一步输入命令打包时需要注意版本，Nordic 定义如下的版本规则，如果版本号错误，Bootloader 不会接受该升级文件。

1. Hardware version：硬件版本，如果是 nRF52840，设置为 52。
2. SoftDevice Firmware ID ：协议栈(SoftDevice)ID，应输入使用的协议栈版本对应的 ID，本例使用的协议栈是 flash_s140_nrf52_7.0.1_softdevice，所以协议栈 ID 是 0xCA。其它版本的协议栈对应的 ID 如下：

表 11-21：SoftDevice ID

SoftDevice	FWID
s112_nrf52_6.0.0	0xA7
s112_nrf52_6.1.0	0xB0
s130_nrf51_1.0.0	0x67
s130_nrf51_2.0.0	0x80
s132_nrf52_2.0.0	0x81
s130_nrf51_2.0.1	0x87
s132_nrf52_2.0.1	0x88
s132_nrf52_3.0.0	0x8C
s132_nrf52_3.1.0	0x91
s132_nrf52_4.0.0	0x95
s132_nrf52_4.0.2	0x98
s132_nrf52_4.0.3	0x99
s132_nrf52_4.0.4	0x9E
s132_nrf52_4.0.5	0x9F
s132_nrf52_5.0.0	0x9D
s132_nrf52_5.1.0	0xA5
s132_nrf52_6.0.0	0xA8

s132_nrf52_6.1.0	0xAF
s140_nrf52_6.0.0	0xA9
s140_nrf52_6.1.0	0xAE
s140_nrf52_7.0.1	0xCA
s132_nrf52_7.0.1	0xCB

4.9. 执行 DFU

4.9.1. 首次无线 DFU

1. 将打包的“dfufile.zip”文件发送到手机。
2. 手机上打开“nRF Connect”，点击“SCAN”启动扫描，这时会扫描到名为“DfuTarg”的设备，点击“CONNECT”连接设备，连接成功后，会看到“nRF Connect”上的“DFU”图标，如下图所示。

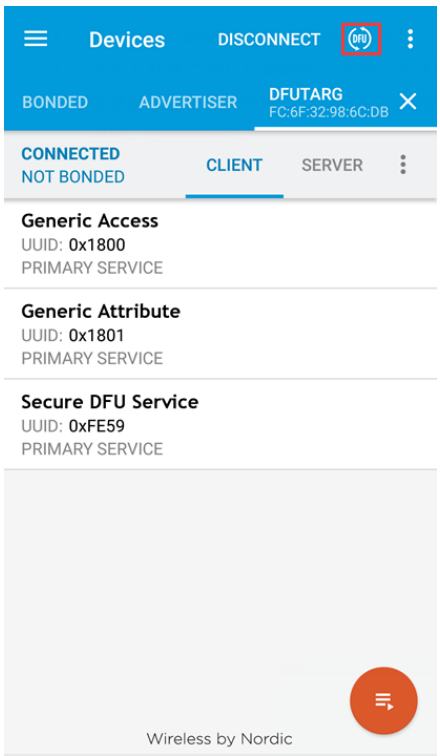


图 11-42：连接成功

3. 点击“DFU”图标，弹出选择文件的对话框，这里选择 ZIP 文件，点击【OK】后，导航到“dfufile.zip”文件的路径，打开“dfufile.zip”文件。

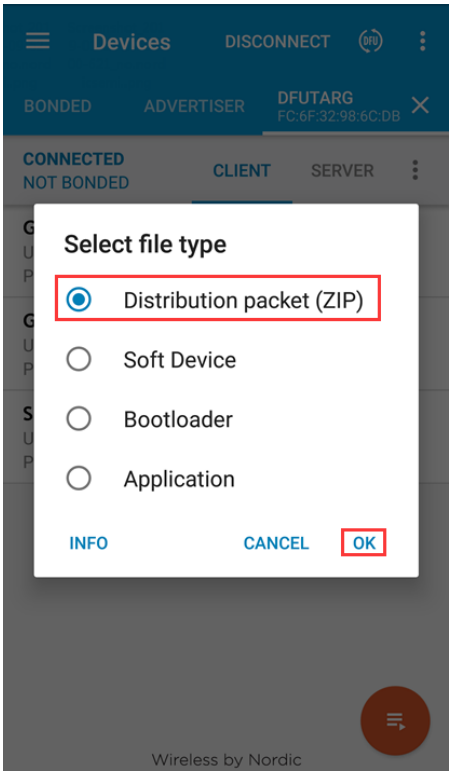


图 11-43：选择升级文件

4. 选择“dfufile.zip”文件后，升级开始，如下图所示，APP 中会显示升级的进度。

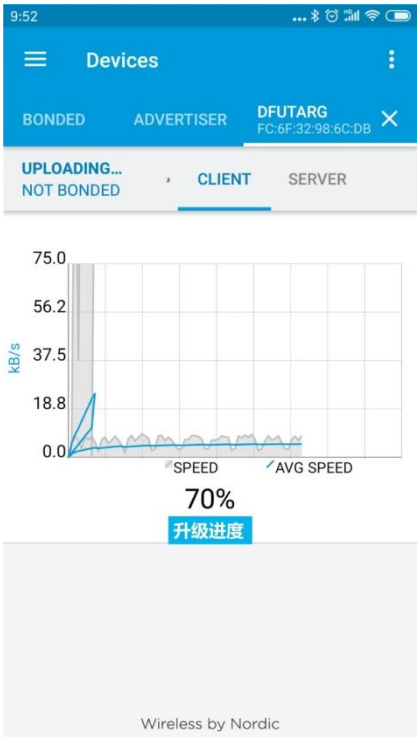


图 11-44：DFU

5. 升级完成后，程序会跳转到升级的应用程序运行，这时用“nRF Connect”扫描，会扫描到名为“Nordic_Buttonless”的设备，说明升级成功。

4.9.2. 再次无线 DFU

经过上面的 DFU 步骤，我们已经将应用程序发送到设备，并让设备跳转执行应用程序，这时候，如果我们要再次对设备进行无线更新固件，按照下面的步骤进行。

1. 点击手机“nRF Connect”上的“SCAN”启动扫描，这时会扫描到名为“Nordic_Buttonless”的设备，点击“CONNECT”连接设备，如下图所示。

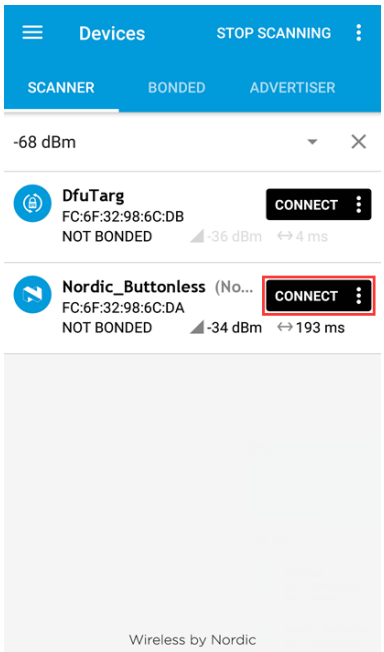


图 11-45：扫描并连接设备

2. 连接成功后，点击“Secure DFU Service”展开服务，之后点击右侧的双箭头使能指示，然后点击向上的小箭头发送升级命令，如下图所示。

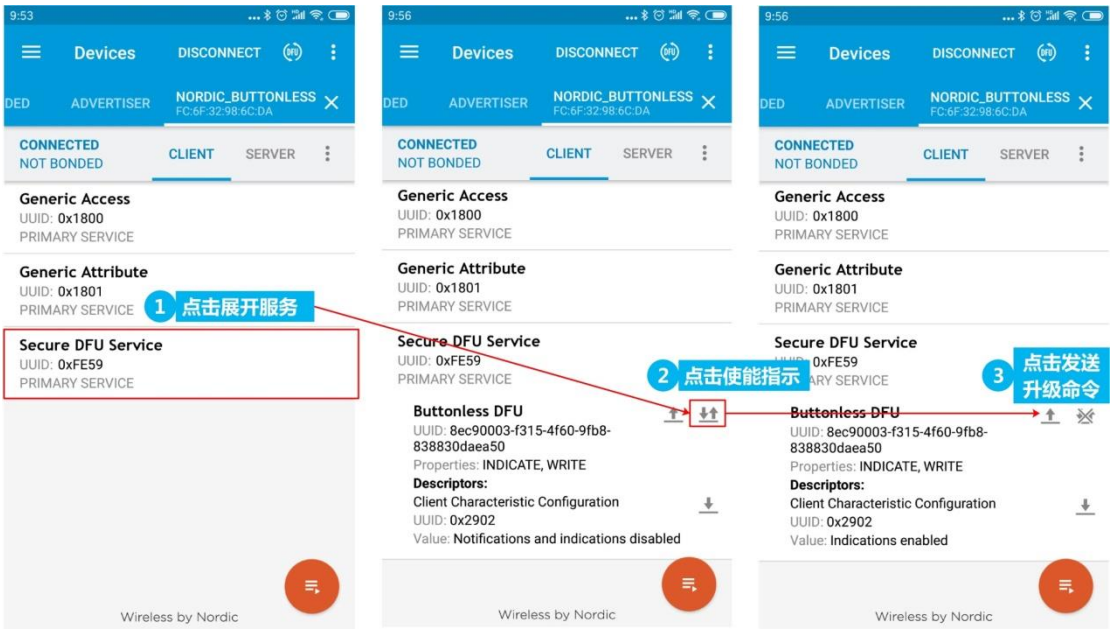


图 11-46：点击写特征值按钮

3. 弹出的窗口中点击“SEND”按钮发送升级命令

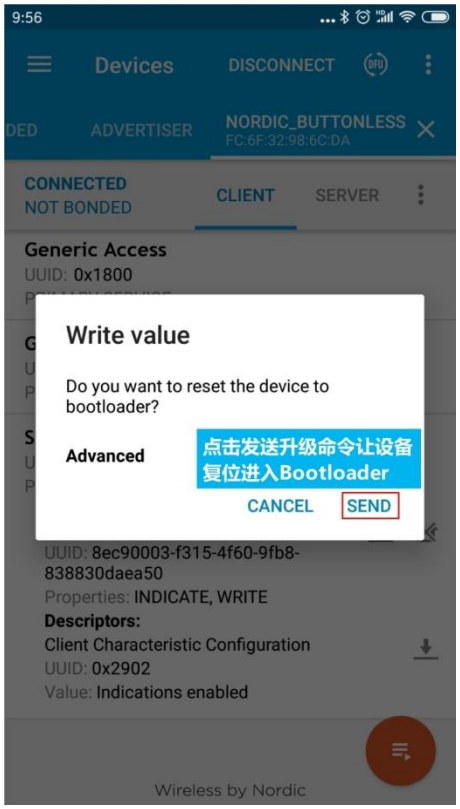


图 11-47：发送命令

发送升级命令后，设备会复位，复位后进入 Bootloader，和之前升级一样："nRF Connect"中点击"SCAN"启动扫描，这时会扫描到名为"DfuTarg"的设备，点击"CONNECT"连接设备，连接成功后，会看到"nRF Connect"上的"DFU"，点击"DFU"按钮，发送新的打包的 ZIP 文件即可更新设备的应用程序，更新完成后，设备同样会跳转到新的应用程序执行。

5. 代码的烧写和合并

5.1. 生成 settings 和代码的分开烧写

由前面的步骤我们发现，我们需要先烧写好协议栈和 Bootloader，然后通过 DFU 写入应用程序，这时候可以正常运行应用程序。如果我们直接烧写协议栈、Bootloader 和应用程序，程序是无法正常运行的。

这是因为设备启动后，Bootloader 会检查 settings 页面的 CRC，如果 settings 页面不存在，程序会继续停留在 Bootloader，而不会跳转去执行应用程序。

所以，要使设备能实现正常跳转，就需要烧写 settings 文件。

■ **生成 settings 文件步骤：**

1. 新建一个文件夹，名字随便取，如"dfufile"。
2. 将应用程序编译后生成的 hex 文件拷贝到 dfufile 文件夹，并改名为 app.hex。
3. 鼠标放在 dfufile 文件夹里面空白的地方，shift+右键执行"在此处打开命令窗口(W)"。
4. 命令窗口中输入：nrfutil settings generate --family NRF52840 --application

nrf52840_xxaa.hex --application-version 1 --bootloader-version 1 --bl-settings-version 1 settings.hex。输入完成后回车，如下图所示。

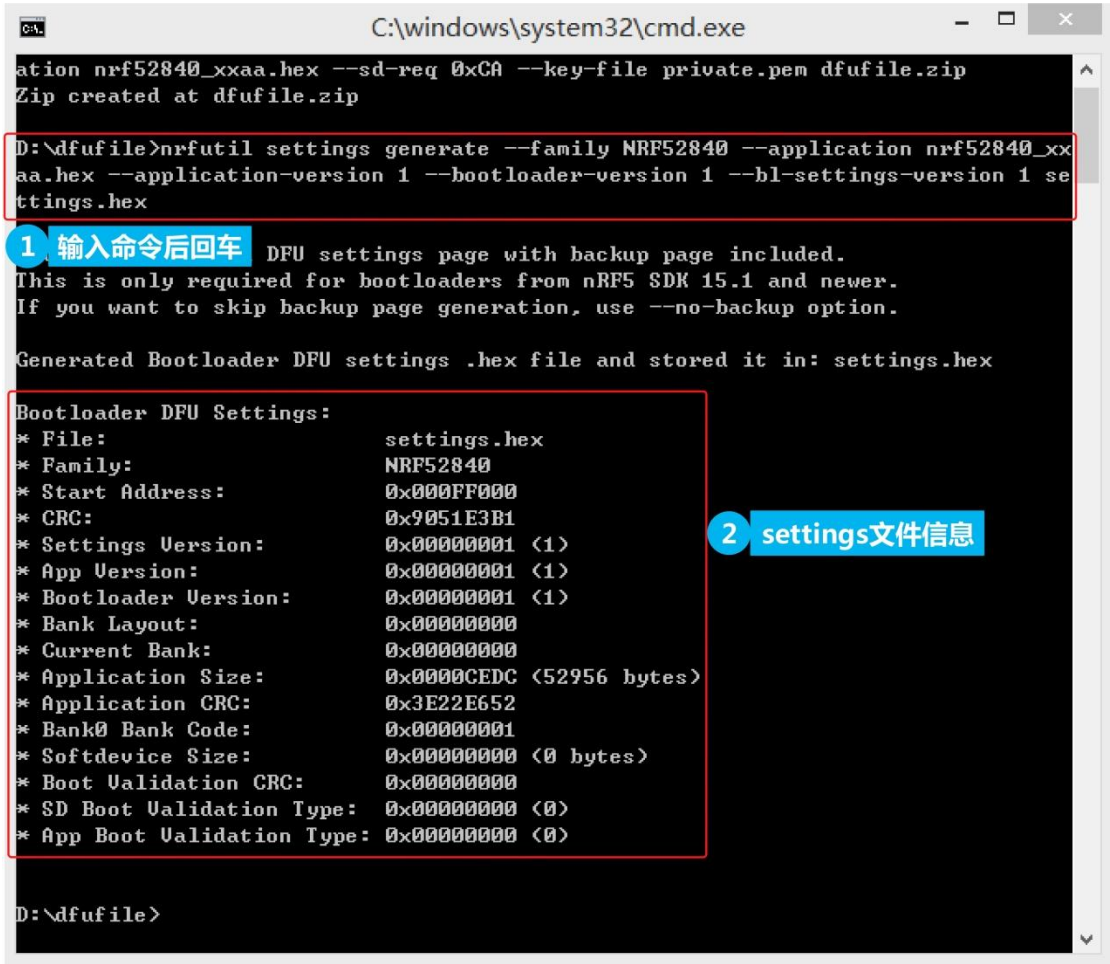


图 11-48：生成 settings 文件

分开烧写的步骤：

有了 settings 文件，就可以分开烧写，烧写后程序会运行应用程序而不是停留在 Bootloader。这里我们使用 J-Flash 烧写，烧写分为 4 步完成：

1. 烧写协议栈：“flash_s140_nrf52_7.0.1_softdevice.hex”。
2. 烧写 settings 文件：“settings.hex”。
3. 烧写 Bootloader：“nrf52840_xxaa_s140.hex” (Bootloader 编译后生成的 HEX 文件)。
4. 烧写应用程序:“nrf52840_xxaa.hex”(ble_app_buttonless_dfu 编译后生成的 HEX 文件)。

烧写完成后，按下快捷键 F9 运行程序，程序运行后，开发板上的指示灯 D1 闪烁，指示设备正在广播，这时用 nRF Connect 可以扫描名为“Nordic_Buttonless” 的设备，说明程序运行的是应用程序。

5.2. 使用 mergehex 合并烧写

“5.1 节”中的分开烧写的方法是验证加入“Settings”文件后，程序烧写完成可以直接运行应用程序，量产的时候，如果也用分开的方式烧录，会浪费很多时间，我们希望一次就可以烧写完成。这时，可以使用 mergehex 来合并 HEX 文件，将待烧写的多个 HEX 文件合

并成一个 HEX 文件。

Mergehex.exe 文件包含在 nRF5x Command Line Tools 里面，当我们安装了 nRF5x Command Line Tools 后，即可使用 Mergehex。

表 11-3：Mergehex 命令表

命令缩写	命令	描述
-h	--help	显示帮助。
-v	--version	查看 Mergehex 的版本。
-q	--quiet	退出。
-m	--merge <hex.file> <hex.file> [<hex.file>]	合并 HEX 文件，必须和—output 一起使用。
-o	--output <hex.file>	合并生成的 HEX 文件，必须和—merge 命令一起使用。

Mergehex 合并 HEX 文件的命令格式如下：

```
mergehex -m file1.hex file2.hex file3.hex -o output_file.hex
```

注意 Mergehex 一次最多支持合并 3 个 hex 文件，而要使程序启动后会跳转到 APP 执行，需要合并 4 个文件，所以我们要进行 2 次合并。

1. 合并：sd+ bootloader+app，操作步骤如下：

1) 新建一个文件夹，名字随便取，如“dfufile”。
2) 将协议栈(s140_nrf52_7.0.1_softdevice.hex)、Bootloader（nrf52840_xxaa_s140.hex）、应用程序(nrf52840_xxaa.hex)和 settings.hex 文件拷贝到 dfufile 文件夹。
3) 鼠标放在 dfufile 文件夹里面空白的地方，shift+右键执行“在此处打开命令窗口(W)”。
4) 命令窗口中输入：mergehex -m s140_nrf52_7.0.1_softdevice.hex nrf52840_xxaa_s140.hex nrf52840_xxaa.hex -o out3t1.hex。输入完成后回车，如下图所示。

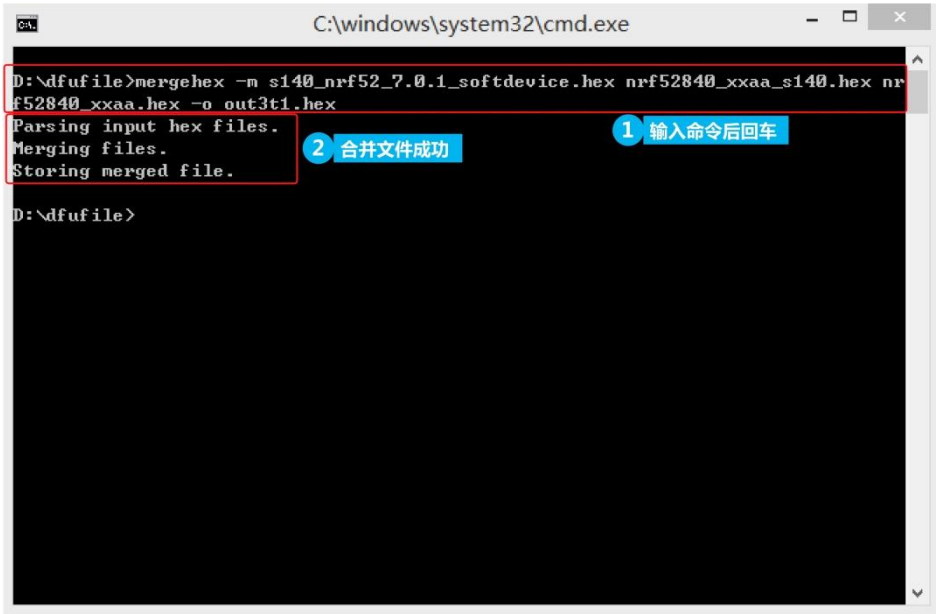


图 11-49：合并 sd+ bootloader+app

2. 合并：out3t1.hex + settings.hex 操作步骤如下：

输入命令：mergehex -m out3t1.hex settings.hex -o out4t1.hex，输入完成后回车，如下图所示，合并后生成 out4t1.hex，这样就将需要烧写的多个 HEX 文件合并成了一个，烧写时只需要烧写这一个 HEX 文件就可以了。

使用 J-Flash 将 out4t1.hex 下载到开发板运行，可以观察到开发板启动后会跳转到应用程序去执行。

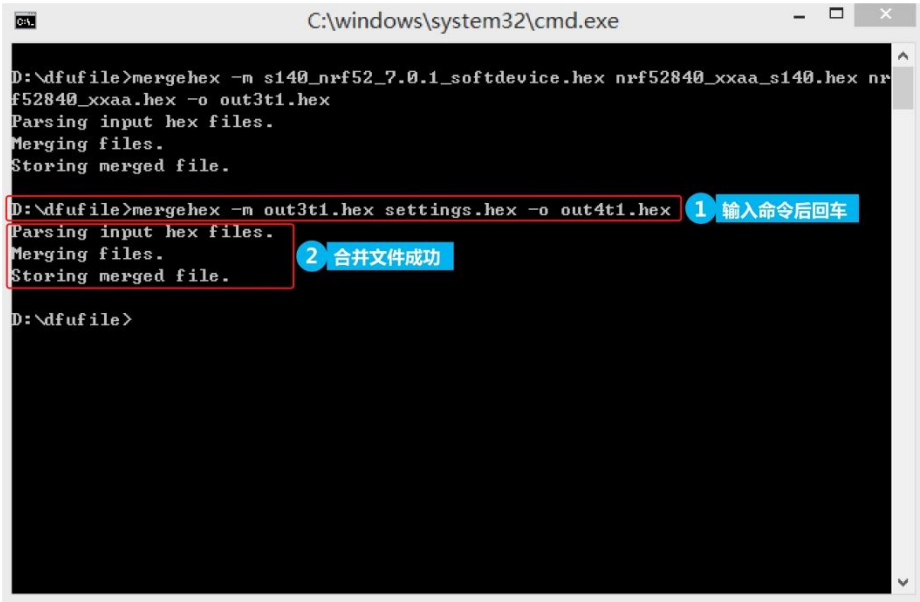


图 11-50：合并 out3t1+settings

6. BLE 串口透传中加入 DFU 服务

无线升级的主体功能都是由 Bootloader 完成，对于应用程序来说，只需要加入 DFU 服务，让应用程序可以接收手机 APP 发送的升级命令，并能在接收命令后跳转到 Bootloader

即可，接下来的升级流程交给 Bootloader 完成。

所以，如果我们开发的应用程序要具备升级功能，只需要在应用程序中加入 DFU 服务即可。本节以开发指南下册从机实验中的串口透传例程“实验 7-3：串口透传长包传输-判断 notify”说明如何在我们自己的应用程序中加入 DFU 服务。

应用程序中加入 DFU 服务需要参考 SDK 中的“ble_app_buttonless_dfu”例程，也就是将该例程中 DFU 相关的部分移植到我们的应用程序即可。

✧ 注：本节对应的实验源码是：“实验 11-1：BLE 串口透传加入 DFU”。

6.1. 需要加入的文件和引用的头文件

本节以开发指南下册从机实验中的串口透传例程“实验 7-3：串口透传长包传输-判断 notify”为基础，说明在该实验的基础上增加 DFU 功能时需要增加的文件和引用的头文件。

1. DFU 需要添加的文件

新建两个名称分别为“nRF_DFU”和“nRF_SVC”的组，然后加入下面所列的文件。

表 11-4： DFU 需要添加的文件

文件名	路径	加入的组
ble_dfu.c	..\components\ble\ble_services\ble_dfu	nRF_DFU
ble_dfu_bonded.c	..\components\ble\ble_services\ble_dfu	nRF_DFU
ble_dfu_unbonded.c	..\components\ble\ble_services\ble_dfu	nRF_DFU
nrf_dfu_svci.c	..\components\libraries\bootloader\dfu	nRF_SVC

2. DFU 需要包含头文件路径

✧ 注意：以下的头文件包含路径是在“实验 7-3：串口透传长包传输-判断 notify”的基础上需要加入的，若在其他工程中加入 DFU，请注意，需要包含的头文件路径可能和下表中的不一样。

表 11-5： DFU 需要包含头文件路径

文件名	路径
1	..\..\components\libraries\bootloader
2	..\..\components\libraries\bootloader\dfu
3	..\..\components\ble\ble_services\ble_dfu

3. 需要引用的头文件

使用 DFU 时，main.c 文件需要引用下面的头文件。

代码清单： DFU 引用的头文件

```
#include "nrf_power.h"
#include "nrf_bootloader_info.h"
```

```
#include "ble_dfu.h"
#include "nrf_dfu_ble_svci_bond_sharing.h"
#include "nrf_svci_async_function.h"
#include "nrf_svci_async_handler.h"
```

4. 需要增加的全局宏定义

全局宏定义中加入“BL_SETTINGS_ACCESS_ONLY NRF_DFU_SVCI_ENABLED NRF_DFU_TRANSPORT_BLE=1”，加入的步骤如下：

1) 点击魔术棒或者执行 Project→Option for Target，打开工程配置窗口。
2) 切换到“C/C++”选项卡。
3) Define 中加入 DFU 需要的全局宏定义，如下图所示。

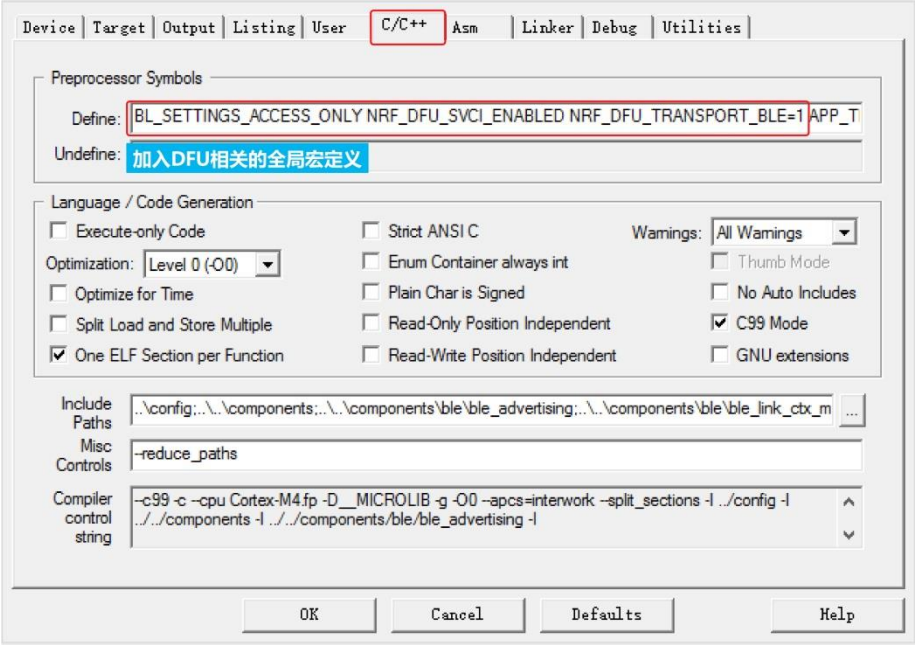


图 11-51：全局宏定义加入 DFU 需要的宏定义

6.2. 工程配置

工程配置中需要使能 DFU 程序模块和修改自定义 UUID 数量，打开“sdk_config.h”文件，切换到“Configuration Wizard”，如下图所示。

1. 使能 DFU 程序模块

勾选“BLE_DFU_ENABLED”。

2. 修改 NRF_SDH_BLE_VS_UUID_COUNT

BLE 串口透传使用的是自定义 UUID，它包含了一个 128 位的 UUID 基数，DFU 使用的也是自定义 UUID，它同样包含了一个自定义 UUID 基数，所以，需要修改自定义 UUID 的数量为 2，否则程序是无法运行的。

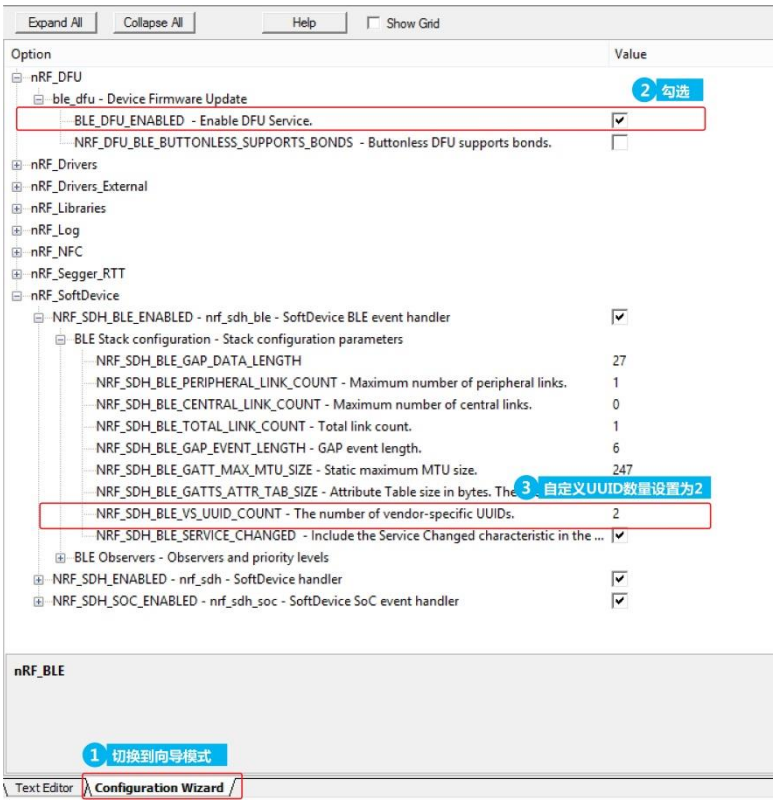


图 11-52：使能 DFU 程序模块和修改自定义 UUID 数量

3. 修改 RAM 配置

修改了自定义 UUID（NRF_SDH_BLE_VS_UUID_COUNT）的数量后，还要根据自定义 UUID 的数量修改内存，每个自定义 UUID 占用的 RAM 大小为 0x10，所以这里增加了一个自定义 UUID，需要按照如下修改内存配置。

- 修改前：起始地址 0x20002AE8，大小 0x3D518
- 修改后：起始地址 0x20002AF8，大小 0x3D508

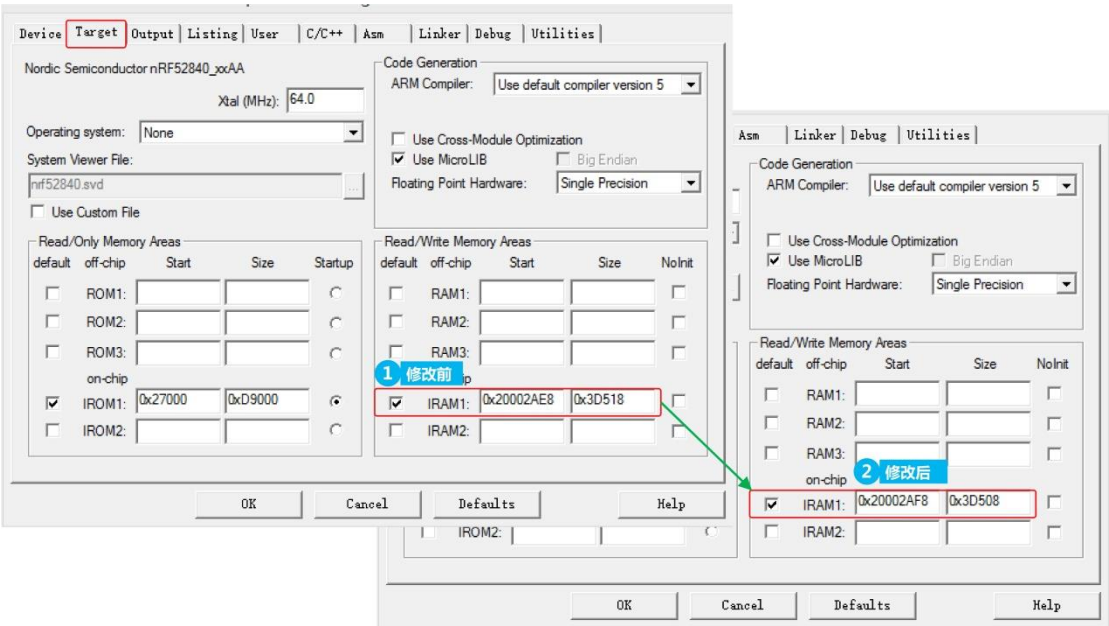


图 11-53：修改 RAM 配置

6.3. 使能 logs

为了方便调试，我们可以打开 logs 输出功能，这样方便观察程序输出的信息，这里我们使能 RTT Viewer 输出 logs，打开“sdk_config.h”文件，配置如下图所示。

✧ **注意：**logs 功能并不是必须的，如果不需要输出调试信息，这一步可以省略。

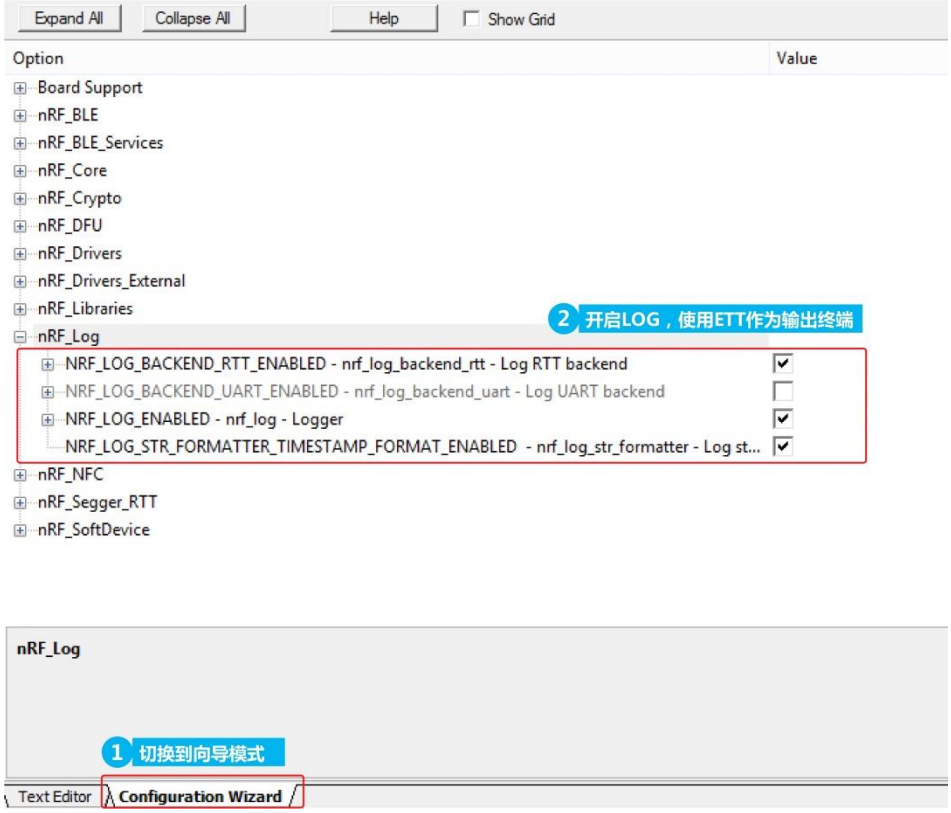


图 11-54：使能 logs

6.4. 需要增加的代码

1. 加入 DFU 服务

服务初始化函数 services_init()中加入 DFU 服务初始化的代码，代码清单如下。

代码清单： services_init()函数中加入 DFU 服务初始化

```
static void services_init(void)
{
    //定义 DFU 服务初始化结构体
    ble_dfu_buttonless_init_t dfus_init = {0};

    //设置 DFU 事件回调函数
    dfus_init.evt_handler = ble_dfu_evt_handler;
    //初始化 DFU 服务
    err_code = ble_dfu_buttonless_init(&dfus_init);
    //检查函数返回的错误代码
    APP_ERROR_CHECK(err_code);
}
```

2. DFU 事件回调函数

DFU 服务初始化时注册了 DFU 事件回调函数 ble_dfu_evt_handler()，所以，应用程序需要提供该回调函数用于接收 DFU 事件。

代码清单：DFU 事件回调函数

```
//DFU 事件处理函数。如果需要在 DFU 事件中执行操作，可以在相应的事件里面加入处理代码
static void ble_dfu_evt_handler(ble_dfu_buttonless_evt_type_t event)
{
    switch (event)
    {
        //该事件指示设备正在准备进入 bootloader
        case BLE_DFU_EVT_BOOTLOADER_ENTER_PREPARE:
        {
            NRF_LOG_INFO("Device is preparing to enter bootloader mode.");
            //阻止设备在断开连接时广播
            ble_adv_modes_config_t config;
            advertising_config_get(&config);
            //连接断开后设备不自动进行广播
            config.ble_adv_on_disconnect_disabled = true;
            //修改广播配置
            ble_advertising_modes_config_set(&m_advertising, &config);

            //断开当前已经连接的所有其他绑定设备。在设备固件更新成功（或中止）后，
            //需要在启动时接收服务更改指示
            uint32_t conn_count = ble_conn_state_for_each_connected(disconnect,
                                  NULL);
            NRF_LOG_INFO("Disconnected %d links.", conn_count);
            break;
        }
        //该事件指示函数返回后设备即进入 bootloader
        case BLE_DFU_EVT_BOOTLOADER_ENTER:
            //如果应用程序有数据需要保存到 Flash，通过 app_shutdown_handler 返回 flase
            //以延迟复位，从而保证数据正确写入到 Flash
            NRF_LOG_INFO("Device will enter bootloader mode.");
            break;
        //该事件指示进入 bootloader 失败
        case BLE_DFU_EVT_BOOTLOADER_ENTER_FAILED:
            //进入 bootloader 失败，应用程序需要采取纠正措施来处理问题，如使用
            //APP_ERROR_CHECK 复位设备
            NRF_LOG_ERROR("Request to enter bootloader mode failed asynchroneously.");
            break;
        //该事件指示发送响应失败
        case BLE_DFU_EVT_RESPONSE_SEND_ERROR:
            NRF_LOG_ERROR("Request to send a response to client failed.");
```

```
            //发送响应失败，应用程序需要采取纠正措施来处理问题，如使用
            //APP_ERROR_CHECK 复位设备
            APP_ERROR_CHECK(false);
            break;
        default:
            NRF_LOG_ERROR("Unknown event from ble_dfu_buttonless.");
            break;
    }
}
```

3. 断开连接和防止设备进入广播状态

DFU 事件回调函数中若接收到 BLE_DFU_EVT_BOOTLOADER_ENTER_PREPARE 事件，表示设备正在准备复位进入 bootloader，这时设备应断开当前连接。在 BLE 的程序中，连接断开后，设备会自动进入广播状态，因此，需要重新配置广播参数，关闭设备断开连接后自动进入广播状态的功能。

代码清单：设置广播模式配置结构体 p_config 中的广播模式、间隔和超时时间，以便修改广播配置。

```
static void advertising_config_get(ble_adv_modes_config_t * p_config)
{
    memset(p_config, 0, sizeof(ble_adv_modes_config_t));

    p_config->ble_adv_fast_enabled  = true;
    p_config->ble_adv_fast_interval = APP_ADV_INTERVAL;
    p_config->ble_adv_fast_timeout  = APP_ADV_DURATION;
}
```

代码清单：断开当前连接

```
//断开当前连接，设备准备进入 bootloader 之前，需要先断开连接
static void disconnect(uint16_t conn_handle, void * p_context)
{
    UNUSED_PARAMETER(p_context);
    //断开当前连接
    ret_code_t err_code = sd_ble_gap_disconnect(conn_handle, BLE_HCI_REMOTE_USER
                                _TERMINATED_CONNECTION);
    if (err_code != NRF_SUCCESS)
    {
        NRF_LOG_WARNING("Failed to disconnect connection. Connection handle:
                            %d Error: %d", conn_handle, err_code);
    }
    else
    {
```

```
        NRF_LOG_DEBUG("Disconnected connection handle %d", conn_handle);
    }
}
```

4. 注册 SoftDevice 状态监视者

SoftDevice 状态监视者用于 SoftDevice 状态改变或者即将改变时接收 SoftDevice 事件，以便应用程序能根据 SoftDevice 状态做出相应处理。

代码清单：注册 SoftDevice 状态监视者

```
//注册SoftDevice状态监视者,用于SoftDevice状态改变或者即将改变时接收SoftDevice事件
NRF_SDH_STATE_OBSERVER(m_buttonless_dfu_state_obs, 0) =
{
    .handler = buttonless_dfu_sdh_state_observer,
};
```

代码清单： SoftDevice 状态监视者事件处理函数

```
static void buttonless_dfu_sdh_state_observer(nrf_sdh_state_evt_t state, void *
p_context)
{
    if (state == NRF_SDH_EVT_STATE_DISABLED)
    {
        //表明Softdevice在复位之前已经禁用，告之bootloader启动时应跳过CRC
        nrf_power_gpregret2_set(BOOTLOADER_DFU_SKIP_CRC);

        //进入system off模式
        nrf_pwr_mgmt_shutdown(NRF_PWR_MGMT_SHUTDOWN_GOTO_SYSOFF);
    }
}

```

5. 注册应用程序关闭事件处理程序

代码清单：注册应用程序关闭事件处理程序

```
//注册优先级为0的应用程序关闭事件处理程序
NRF_PWR_MGMT_HANDLER_REGISTER(app_shutdown_handler, 0);
```

代码清单：app_shutdown_handler 事件回调函数

```
//关机准备处理程序。在关闭过程中，将以 1 秒的间隔调用此函数，直到函数返回 true。当函数返回 true 时，表示应用程序已准备好复位为 DFU 模式
static bool app_shutdown_handler(nrf_pwr_mgmt_evt_t event)
{
    switch (event)
    {
        case NRF_PWR_MGMT_EVT_PREPARE_DFU:
            NRF_LOG_INFO("Power management wants to reset to DFU mode.");
            // YOUR_JOB: Get ready to reset into DFU mode
            //
            // If you aren't finished with any ongoing tasks, return "false" to
            // signal to the system that reset is impossible at this stage.
            //
            // Here is an example using a variable to delay resetting the device.
            //
            // if (!m_ready_for_reset)
            // {
            //      return false;
            // }
            // else
            //{
            //
            //    // Device ready to enter
            //    uint32_t err_code;
            //    err_code = sd_softdevice_disable();
            //    APP_ERROR_CHECK(err_code);
            //    err_code = app_timer_stop_all();
            //    APP_ERROR_CHECK(err_code);
            //}
            break;

        default:
            // YOUR_JOB: Implement any of the other events available from the power management module:
            //      -NRF_PWR_MGMT_EVT_PREPARE_SYSOFF
            //      -NRF_PWR_MGMT_EVT_PREPARE_WAKEUP
            //      -NRF_PWR_MGMT_EVT_PREPARE_RESET
            return true;
    }

    NRF_LOG_INFO("Power management allowed to reset to DFU mode.");
    return true;
```

```
}
```

6. 处理异步 SVCI 接口

在 main()函数中加入如下所示的代码，用于处理异步 SVCI 接口和广播设备名称。

✧ **注意：**这段代码加上之后，开发板中必须要烧写 Bootloader 和 settings 才能运行该串口透传应用程序。因此，建议在调试应用程序的时候，先屏蔽这段代码，这样就可以像普通的 BLE 一样来调试该应用程序，调试完成后再启用这段代码。

代码清单：处理异步 SVCI 接口

```
ret_code_t err_code;
//使能中断之前，初始化异步 SVCI 接口到 Bootloader
err_code = ble_dfu_buttonless_async_svci_init();
```

6.5. 该步骤问题总结

■ **问题 1：应用程序返回错误代码 0x0004**

- 原因：错误代码 0x0004 是 NRF_ERROR_NO_MEM，这是因为 BLE 串口透传使用的是自定义 UUID，它包含了一个 128 位的 UUID 基数，DFU 使用的也是自定义 UUID，它同样包含了一个自定义 UUID 基数，即应用程序中自定义 UUID 基数是 2 个，而“sdk_config.h”文件中修改了 NRF_SDH_BLE_VS_UUID_COUNT 的值为 2 后，没有修改内存配置，从而导致内存分配不够。
- 解决办法：按照 6.2 节第 3 个条目处理即可。

第十二章：实现时钟日历

1. 学习目的

1. 了解 C 标准库中与时间相关的一些类型和函数。
2. 掌握 nRF52840 裸机程序和 BLE 程序下实现时钟日历的方法以及它们的区别。

2. 时间的处理

对于时钟日历，我们首先想到的是如何表示时间，也就是如何表示“年、月、日、时、分、秒”信息，因此我们需要了解一下 C 标准库的头文件<time.h>，<time.h>中定义了一些用于处理日期和时间的类型和函数，本章中我们需要用到的是 tm 结构体和时间的格式化，<time.h>中还有其他的时间类型和相关函数，感兴趣的读者可以自行了解一下。

- tm 结构体：用来记录“年、月、日、时、分、秒”信息。
- 时间的格式化：将时间格式化为我们需要的格式。方便串口打印和 OLED 显示。

1. tm 结构体

tm 结构体是用于记录日期时间的结构体，它存储了“年、月、日、时、分、秒”以及星期和夏令时标志等信息，最小单位是秒，tm 结构体原型如下。

代码清单：tm 结构体

```
struct tm {
    int tm_sec;         // 秒，范围从:[0~59]
    int tm_min;         // 分，范围:[0~59]
    int tm_hour;        // 小时，范围:[0~23]
    int tm_mday;        // 一月中的第几天，范围:[1~31]
    int tm_mon;         // 月份，范围:[0~11]
    int tm_year;        // 自 1900 起的年数
    int tm_wday;        // 一周中的第几天，范围:[0~6]
    int tm_yday;        // 一年中的第几天，范围:[0~365]
    int tm_isdst;       // 夏令时,0：不实行夏令时，正：不实行夏令时，负：情况不明
};
```

2. 时间格式化

tm 结构体在记录和存储时间上很方便，但很多时候处理起来不方便，如通过串口打印时间，这时候我们希望得到的是按照我们指定的格式用字符串表示的时间。C 标准库考虑到了这一点，<time.h>中专门提供了一个时间格式化函数 Strftime()，该函数可以将 tm 结构体表示的时间按照指定的格式化规则，格式化为字符串，strftime()函数原型如下表所示。

表 12-1：strftime()函数

函数原型	size_t strftime(char * str size_t maxsize const char * format const struct tm * timeptr)
函数功能	时间格式化函数，该函数按照 format 指定的格式化规则格式化 tm 结构体 timeptr 表示的时间，并将格式化结果存储到 str 中。
参　　数	[in] str：指向目标数组的指针，用来复制产生的 C 字符串。 [in] maxsize：复制到 str 的最大字符数。 [in]format：这是 C 字符串，包含了普通字符和特殊格式说明符的任何组合，这些格式说明符由函数替换为表示 tm 中所指定时间的相对应值。格式说明符详见“*表 12-2：格式说明符*”。 [in] timeptr：指向 tm 结构的指针。
返回值	如果产生的 C 字符串小于 size 个字符（包括空结束字符），返回复制到 str 中的字符总数（不包括空结束字符），否则返回零。

表 12-2：格式说明符

格式说明符	描述	示例
%a	星期几的缩写	Sun
%A	星期几的全称	Sunday
%b	月份名称的缩写	Mar
%B	月份名称的全称	March
%c	日期和时间表示法	Sun Aug 19 02:56:02 2012
%d	一月中的第几天（01-31）	19
%H	24 小时格式的小时（00-23）	14
%I	12 小时格式的小时（01-12）	05
%j	一年中的第几天（001-366）	231
%m	十进制数表示的月份（01-12）	08
%M	分（00-59）	55
%p	AM 或 PM 名称	PM
%S	秒（00-61）	02
%U	一年中的第几周，以第一个星期日作为第一周的第一天（00-53）	33
%w	十进制数表示的星期几，星期日表示为 0（0-6）	4
%W	一年中的第几周，以第一个星期一作为第一周的第一天（00-53）	34
%x	日期表示法	08/19/12
%X	时间表示法	02:50:06

%y	年份，最后两个数字（00-99）	01
%Y	年份	2012
%Z	时区的名称或缩写	CDT
%%	一个%符号	%

■ **时间格式化示例**

下面是一个使用 strftime()函数格式化 tm 结构体的示例，代码中定义一个 tm 结构体 t 并初始化时间，之后调用 strftime()函数按照"%Y-%m-%d-%H:%M:%S"格式规则格式化 t，即将时间格式化为“年-月-日-时:分:秒”，格式化结果通过串口输出。

代码清单：时间格式化示例

```
struct tm t;//定义一个 tm 结构体变量 t
static char cal_string[80];//定义用来保存格式化结果的数组
//初始化 t
t.tm_year = 2019 - 1900;
t.tm_mon = 5;
t.tm_mday = 22;
t.tm_hour = 16;
t.tm_min = 26;
t.tm_sec = 30;
//格式化字符串
strftime(cal_string, 80, "%Y-%m-%d-%H:%M:%S", &t);
//打印结果
printf("time:\t%s\r\n", nrf_cal_get_time_string());
```

程序执行后，串口输出 tm 结构体 t 格式化后的字符串：“time: 2019-06-22-16:26:30 ”。

3. RTC 实现时钟日历(OLED 显示)

本实验在开发指南（上册）配套实验源码“实验 10-1：串口数据收发”的基础上修改。本例中通过 SPI 驱动 OLED，SPI 相关的内容读者可参见《开发指南上册》中 SPI 章节。由于本例中需要用到 C 标准库中的时间日期处理相关的函数，因此需要引用头文件“time.h”。

✧ 注：本节对应的实验源码是：“实验 12-1：RTC 实现时钟日历(OLED 显示)”。

3.1. 实验内容

定义一个 tm 结构体的变量 t，用于存储时间，通过 RTC0 的比较事件产生 1 秒的计时，在每个事件中断里面更新 t 的时间，同时根据更新的时间内容（秒更新、时分更新和日期更新）设置时间更新标志，主循环里面通过检查时间更新标志来修改 OLED 的显示。

系统启动后，通过串口来同步时间，注意这里的同步是手动同步，不是自动同步。

3.2. 软件设计

RTC 实现的时钟日历原理框图如下，主要包含下面四个方面的内容：

- 秒计时：由 RTC0 的比较事件提供。
- 时间的记录：定义一个 tm 结构体的变量 t，记录时间，另外再定义一个时间更新标志，用于表示时间的改变和改变的类型，本例中定义了三种类型：秒更新、时分更新和日期更新，这么做的目的是为了更新 oled 显示内容的时候只更新改变了的部分，而不用更新整个显示界面。
- 时间同步：通过串口手工同步。
- oled 显示的更新：主循环中，根据时间更新标志更新显示。

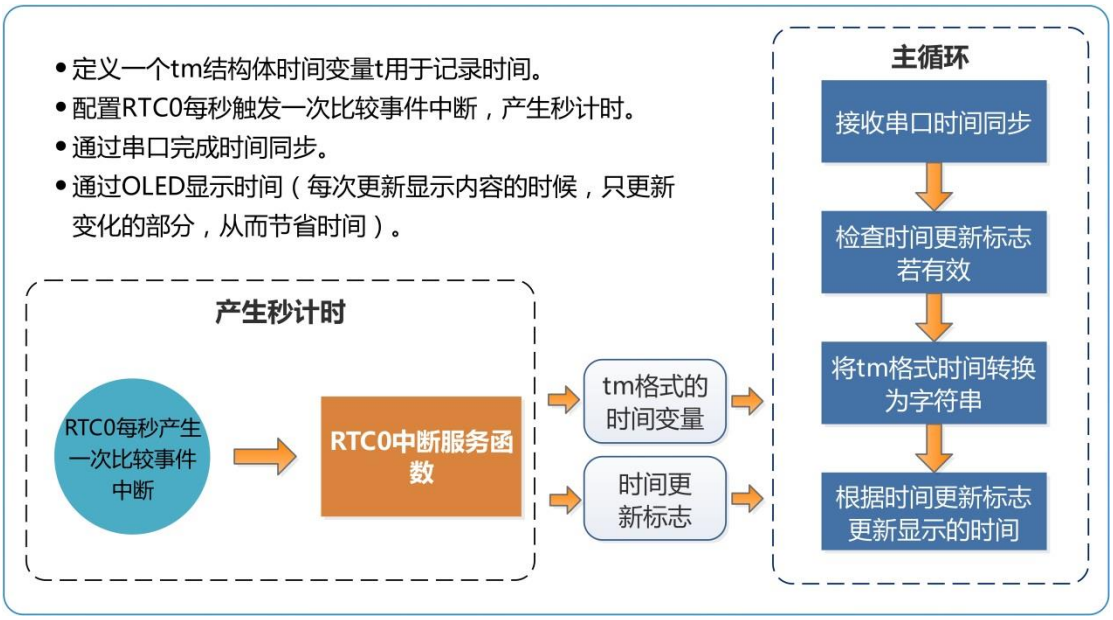


图 12-1：时钟日历原理框图

1. 实现秒计时

秒计时部分主要包含的部分：配置 RTC0 的通道 0 每秒产生一次比较匹配事件，在事件中断服务函数中更新时间，RTC0 和低频时钟的配置代码如下。

代码清单：配置低频时钟，配置 RTC0 每秒产生一次比较事件中断

```
void nrf_cal_init(void)
{
    //使用外部 32.768 kHz 晶体振荡器作为低频时钟源
    NRF_CLOCK->LFCLKSRC = CLOCK_LFCLKSRC_SRC_Xtal << CLOCK_LFCLKSRC_SRC_Pos;
    //LFCLK 启动完成事件清零
    NRF_CLOCK->EVENTS_LFCLKSTARTED = 0;
    //启动 LFCLK
    NRF_CLOCK->TASKS_LFCLKSTART = 1;
    //等待 LFCLK 启动完成
    while(NRF_CLOCK->EVENTS_LFCLKSTARTED == 0);

```

```
    //配置 RTC 每秒产生一次比较匹配事件
    CAL_RTC->PRESCALER = 4095;
    CAL_RTC->EVTENSET = RTC_EVTENSET_COMPARE0_Msk;
    CAL_RTC->INTENSET = RTC_INTENSET_COMPARE0_Msk;
    CAL_RTC->CC[0] = 8;
    CAL_RTC->TASKS_START = 1;
    NVIC_SetPriority(CAL_RTC_IRQn, CAL_RTC_IRQ_Priority);
    NVIC_EnableIRQ(CAL_RTC_IRQn);
}
```

RTC0 事件中断服务函数中通过 CAL_updata()函数更新时间和设置时间更新标志 TimeUpdataFlag。

代码清单：RTC0 中断服务函数

```
void CAL_RTC_IRQHandler(void)
{
    //如果是 RTC0 通道 0 的比较匹配事件
    if(CAL_RTC->EVENTS_COMPARE[0])
    {
        //清零事件
        CAL_RTC->EVENTS_COMPARE[0] = 0;
        //清零计数器
        CAL_RTC->TASKS_CLEAR = 1;
        //更新时间
        CAL_updata();
    }
}
```

2. 更新 oled 时间显示

主循环中通过查询时间更新标志 TimeUpdataFlag 来判断是否需要更新时间显示，同时根据 TimeUpdataFlag 决定需要更新哪一部分的显示。程序中定义了 3 种类型，如下所示：

代码清单：时间显示更新类型

```
#define UPDATA_SEC    1    //更新秒
#define UPDATA_HM     2    //更新时、分
#define UPDATA_DATE   3    //更新年、月、日
```

这么做的目的是为了减少程序的工作量，如只是秒数有变化的时候，我们只需要修改 oled 上秒的显示，而不用刷新整个屏幕。但是要注意的是：在每次进位的时候，前面的显示内容都要刷新，否则 oled 显示在进位的时候会漏刷新一次。如时、分发生变化的时候，除了刷新时、分，秒显示也要刷新，否则当秒计时满 1 分钟的时候，只会刷新时分显示。

主循环中查询时间更新标志的代码如下：

代码清单：更新 OLED 显示

```
//检查时间更新标志
if(TimeUpdataFlag)
{
    oled_updata(TimeUpdataFlag);
    TimeUpdataFlag = 0;  //更新 OLED 显示后清零时间更新标志
    //如需要串口也输出时间信息，将宏定义 UART_LOG 设置为 1
    #if UART_LOG
    printf("current time:\t%s\r\n", nrf_cal_get_time_string());
    #endif
}
```

如果时间更新标志非零，调用 oled_updata()函数更新 OLED 显示，oled_updata()函数代码清单如下：

代码清单：更新 OLED 显示

```
//根据时间更新标志更新 oled 显示，这么做的好处是不用每次都更新整个屏幕
void oled_updata(uint8_t flag)
{
    uint8_t i;
    char *p;
    //获取格式化的时间字符串
    p = nrf_cal_get_time_string();
    switch(flag)
    {
        //更新秒显示：只需更新秒显示即可
        case UPDATA_SEC:
            //仅更新秒显示
            LCD_timestr(96,6,(uint8_t*)(p+17));
            LCD_timestr(104,6,(uint8_t*)(p+18));
            break;
        //更新时和分显示：更新时分和秒显示
        case UPDATA_HM:
            //更新秒显示
            LCD_timestr(96,6,(uint8_t*)(p+17));
            LCD_timestr(104,6,(uint8_t*)(p+18));
            //更新时和分显示
            LCD_P24x48Str(0,0,*(p+11));
            LCD_P24x48Str(24,0,*(p+12));
            LCD_P24x48Str(48,0,':');
            LCD_P24x48Str(72,0,*(p+14));
            LCD_P24x48Str(96,0,*(p+15));
            break;
```

```
            //更新年月日显示：更新年月日和时分秒显示
        case UPDATA_DATE:
            //更新秒显示
            LCD_timestr(96,6,(uint8_t*)(p+17));
            LCD_timestr(104,6,(uint8_t*)(p+18));
            //更新时和分显示
            LCD_P24x48Str(0,0,*(p+11));
            LCD_P24x48Str(24,0,*(p+12));
            LCD_P24x48Str(48,0,':');
            LCD_P24x48Str(72,0,*(p+14));
            LCD_P24x48Str(96,0,*(p+15));
            //更新年、月、日显示
            for(i=0;i<10;i++)
            {
                LCD_timestr(0+8*i,6,(uint8_t*)(p+i));
            }
            break;
        default:
            break;
        }
}
```

3. 串口时间同步

当串口接收到字符“s”后，进入时间同步的处理流程，当接收完年、月、日、时、分、秒数据后，刷新整个 oled 屏幕，显示同步的时间。

代码清单：串口接收同步时间信息

```
if(app_uart_get(&cr) == NRF_SUCCESS)
{
    switch(cr)
    {
        //接收到 s 字符后，进入同步时间处理流程
        case 's':
            year = (uint32_t)uart_get_parameter("Enter year", 1970, 2100);
            month = (uint32_t)uart_get_parameter("Enter month", 0, 11);
            day = (uint32_t)uart_get_parameter("Enter day", 1, 31);
            hour = (uint32_t)uart_get_parameter("Enter hour", 0, 23);
            minute = (uint32_t)uart_get_parameter("Enter minute", 0, 59);
            second = (uint32_t)uart_get_parameter("Enter second", 0, 59);
            nrf_cal_set_time(year, month-1, day, hour, minute, second);
            //时间同步数据接收完成后，刷新整个 oled 显示
            oled_refresh_fullscreen();
            break;
        default:
```

```
            break;
    }
}
```

3.3. 硬件连接

本实验需要使用的 nRF52840 的引脚如下表所示。

表 12-3：格式说明符

部件	引脚	描述
OLED	P1.10	输出：OLED-CS，OLED 片选。
	P1.11	输出：OLED-DC，OLED 命令/数据选择 H=命令 L=数据。
	P1.12	输出：OLED-RST，OLED 复位。
	P1.13	输出：OLED-MOSI，SPI 主出从入。
	P1.14	输出：OLED-CLK，SPI 时钟。
UART	P0.06	串口发送 TXD。
	P0.08	串口接收 RXD。
	P0.05	串口 RTS：发送请求，硬件流控开启时有效，默认不使用。
	P0.07	串口 CTS：发送允许，硬件流控开启时有效，默认不使用。

OLED 安装和跳线帽短接如下图，虽然本例中没有用到按键和 LED，但是按键和 LED 是常用外设，一般都会短接它们的跳线帽。

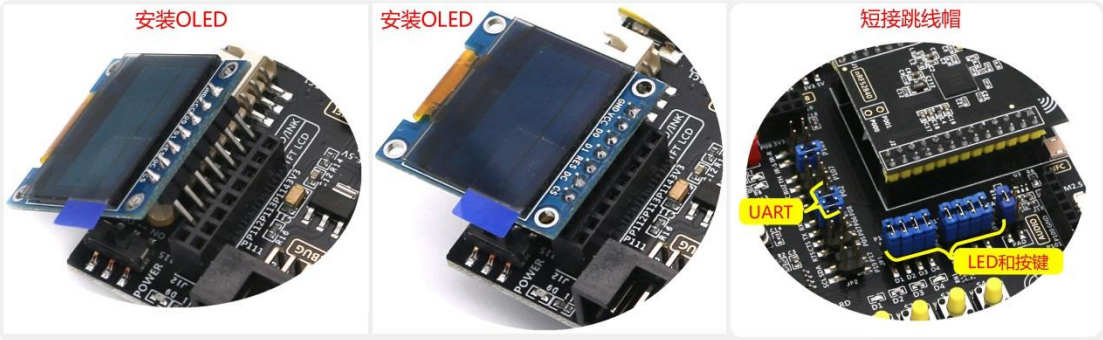


图 12-2：OLED 安装和跳线帽短接

3.4. 实验步骤

1. 解压“…\3：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 12-1：RTC 实现时钟日历(OLED 显示)”，将解压后得到的文件夹“Calendar_oled”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\Calendar_oled\project\mdk5”目录下的工程“Calendar_oled.uvproj”。
4. 点击编译按钮编译工程 。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目

录下的“Objects”文件夹中。

5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，OLED 会显示默认设置的时间并且每秒更新一次时间，如下图所示。
7. 通过 UART 同步时间：计算机上打开串口调试助手，波特率设置为 115200bps，勾选“发送新行”，之后选择开发板对应的串口号，打开串口，如下图所示。
   - 输入字符“s”，点击发送，开发板会返回“Enter year”。
   - 输入年份，示例中输入的是“2017”，点击发送，开发板会返回“Enter month”。
   - 输入月份，示例中输入的是“9”，点击发送，开发板会返回“Enter day”。
   - 输入日，示例中输入的是“15”，点击发送，开发板会返回“Enter hour”。
   - 输入时，示例中输入的是“15”，点击发送，开发板会返回“Enter minute”。
   - 输入分，示例中输入的是“16”，点击发送，开发板会返回“Enter second”。
   - 输入秒，示例中输入的是“15”，点击发送，完成手动时间同步。

图 12-3：UART 同步时间

8. 观察 OLED 显示的时间

时间同步完成后，观察开发板上的 OLED，OLED 会从同步的时间开始显示，并且每秒刷新一次显示。

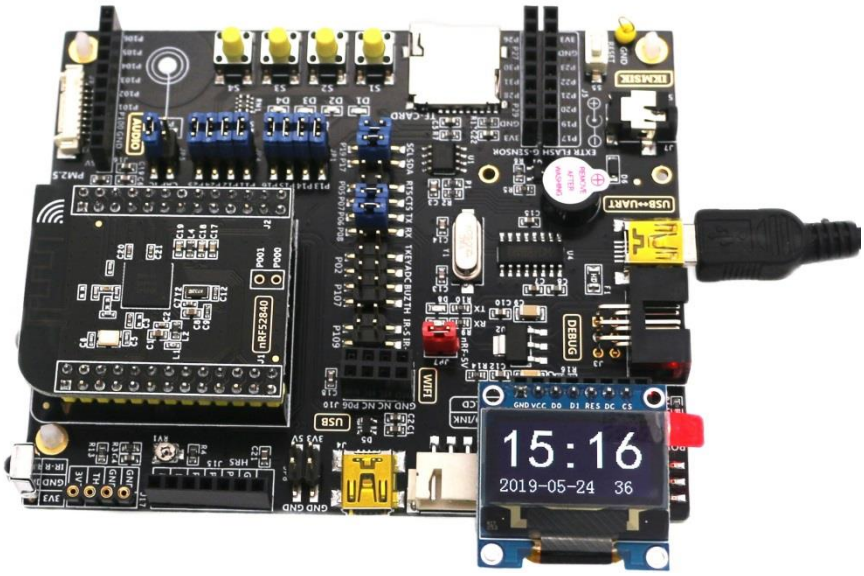


图 12-4：OLED 显示日期和时间

9. 如果希望UART也实时打印出时间信息，将“nrf_calendar.h”文件中的宏定义UART_LOG的值改为“1”即可。

4. BLE 下实现时钟日历(OLED 显示)

本实验在开发指南（下册）配套实验源码“实验 7-3：串口透传长包传输-判断 notify”的基础上修改，实验中会使用 APP 定时器，关于 APP 定时器的应用读者可参见《开发指南下册》中的“第五章：创建定时任务”。由于本例中需要用到 C 标准库中的时间日期处理相关的函数，因此需要引用头文件“time.h”。

✧ 注：本节对应的实验源码是：“实验 12-2：蓝牙 BLE 下实现时钟日历(OLED 显示，使用外部 32.768KHz 晶体)”。

4.1. 实验内容

创建一个周期性的 APP 定时器，用于产生 1 秒的计时，定义一个 tm 结构体的时间变量 t，用于记录时间。APP 定时器启动后，每秒产生一次超时事件，在其事件回调函数中更新更新 t 的时间，，同时根据更新的时间内容（秒更新、时分更新和日期更新）设置时间更新标志，主循环里面通过检查时间更新标志来修改 OLED 的显示。

系统启动后，通过手机 APP“nRF CONNECT”手动同步时间。

本例和裸机下实现的 RTC 时钟的区别主要是：通过应用定时器产生秒计时(裸机下是通过 RTC0)，通过手机 APP“nRF CONNECT”同步时间(裸机下是通过串口同步时间)。

4.2. 软件设计

BLE 下实现的时钟日历原理框图如下，和裸机程序下实现一样，也包含四个方面的内容，只是实现的方式有所差别。

- 秒计时：由周期性 APP 定时器提供。
- 时间的记录：定义一个 tm 结构体的变量 t，记录时间，另外再定义一个时间更新标志，用于表示时间的改变和改变的类型，本例中定义了三种类型：秒更新、时分更新和日期更新，这么做的目的是为了更新 oled 显示内容的时候只更新改变了的部分，而不用更新整个显示界面。
- 时间同步：通过手机 APP“nRF CONNECT”手工同步。
- oled 显示的更新：主循环中，根据时间更新标志更新显示。
- 将SPI的中断优先级改为3，否则由于SPI中断优先级过低会导致OLED无法正常显示。

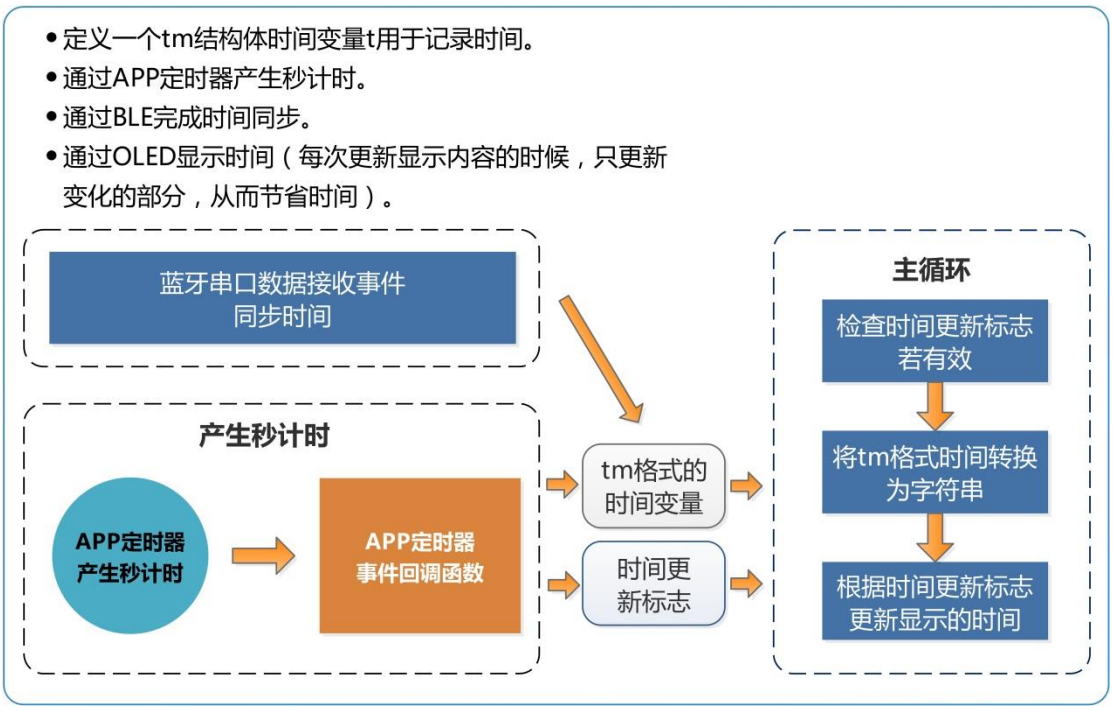


图 12-5：BLE 时钟日历原理框图

1. 实现秒计时

■ 创建 APP 定时器

创建一个超时时间为 1 秒的周期性 APP 定时器，用作秒计时。创建时注册事件句柄 second_updata_handler()，当定时器超时后，调用 second_updata_handler()，该事件处理函数中更新时间。

BLE 程序中在协议栈初始化时已经配置了低频时钟，因此我们不需要去初始化低频时钟，同时注意 APP 定时器不会启动，需要在后面启动该 APP 定时器。

代码清单：创建一个超时时间为 1 秒的周期性 APP 定时器

```
//秒计时应用定时器 ID
APP_TIMER_DEF(m_second_timer_id);
//秒计时应用定时器间隔 1 秒
#define SECOND_INTERVAL          APP_TIMER_TICKS(1000)

static void timers_init(void)
{
    //初始化 APP 定时器模块
    ret_code_t err_code = app_timer_init();
    //检查返回值
    APP_ERROR_CHECK(err_code);
    //创建一个应用定时器，产生秒计时。模式：周期性应用定时器，注册应用定时器事件句柄
    err_code = app_timer_create(&m_second_timer_id,
                                APP_TIMER_MODE_REPEATED,
                                second_updata_handler);
    APP_ERROR_CHECK(err_code);
```

```
}
```

■ APP 定时器事件处理函数中更新时间

事件处理函数 second_updata_handler()中调用 CAL_updata()函数更新时间，代码清单如下：

代码清单：配置低频时钟，配置 RTC0 每秒产生一次比较事件中断

```
//秒计时 应用定时器事件回调函数
static void second_updata_handler(void * p_context)
{
    UNUSED_PARAMETER(p_context);
    //更新 tm 格式时间变量和时间更新标志
    CAL_updata();
}
```

■ 启动 APP 定时器

APP 定时器一般在广播开始前启动，秒计时应用定时器启动后，即开始产生秒计时，也就是提供了一个以秒为单位的计时基准。

代码清单：启动 APP 定时器

```
//启动秒计时应用定时器。启动后，即开始产生秒计时，也就是提供了一个以秒为单位的计时基准
static void application_timers_start(void)
//启动 APP 定时器
ret_code_t err_code = app_timer_start(m_second_timer_id, SECOND_INTERVAL, NULL);
APP_ERROR_CHECK(err_code);
```

2. 更新 oled 时间显示

同“RTC 实现时钟日历(OLED 显示)”实验。

3. BLE 时间同步

当开发板和“nRF CONNECT”连接，由“nRF CONNECT”发送时间同步信息同步开发板的时间，发送的格式为“#”加上年、月、日、时、分、秒，如需要将时间同步为：2019 年 5 月 23 日 15 点 35 分 28 秒，发送“#20190523153528”。

串口透传事件回调函数中接收到时间同步信息后，如时间格式正确，刷新整个 oled 屏幕，显示同步的时间。

代码清单：串口透传事件处理函数中接收 BLE 时间同步信息

```
uint32_t year, month, day, hour, minute, second;
//判断事件类型:接收到新数据事件
if (p_evt->type == BLE_UARTS_EVT_RX_DATA)
{
    //时间同步数据格式#20170902101030
```

```
    //以‘#’开头，并且数据长度等于 15 表示是：时间同步数据
    //注意：串口透传的功能仍然可以使用，只有当‘#’开头并且数据长度为 15 个字节才会被认为是
    //是时间同步数据
    if((p_evt->params.rx_data.length==15) && (p_evt->params.rx_data.p_data[0] == '#'))
    {
       year = nus_get_parameter(&p_evt->params.rx_data.p_data[1],4);
       month = nus_get_parameter(&p_evt->params.rx_data.p_data[5],2);
       day = nus_get_parameter(&p_evt->params.rx_data.p_data[7],2);
       hour = nus_get_parameter(&p_evt->params.rx_data.p_data[9],2);
       minute = nus_get_parameter(&p_evt->params.rx_data.p_data[11],2);
       second = nus_get_parameter(&p_evt->params.rx_data.p_data[13],2);
       if(year>1970 && year<2100) //检查数据是否合法
         if(month>0 && month<13)
           if(day>=1 && day<=31)
              if(hour<=23)
                 if(minute<=59)
                   if(second<=59)
                   {
                      //更新 tm 格式的时间变量
                      nrf_cal_set_time(year, month-1, day, hour, minute, second);
                      //时间同步数据接收完成后，刷新整个 oled 显示
                      oled_refresh_fullscreen();
                    }
    }
    else  //不是‘#’开头的数据仍作为串口透传数据
    {
       for (uint32_t i = 0; i < p_evt->params.rx_data.length; i++)
       {
          while (app_uart_put(p_evt->params.rx_data.p_data[i]) != NRF_SUCCESS);
       }
       while (app_uart_put('\r') != NRF_SUCCESS);
       while (app_uart_put('\n') != NRF_SUCCESS);
    }
}
```

4. 修改 SPI 的中断优先级

打开“sdk_config.h”文件，切换到“Configuration Wizard”，将 SPI 中断优先级改为 3，如下图所示。

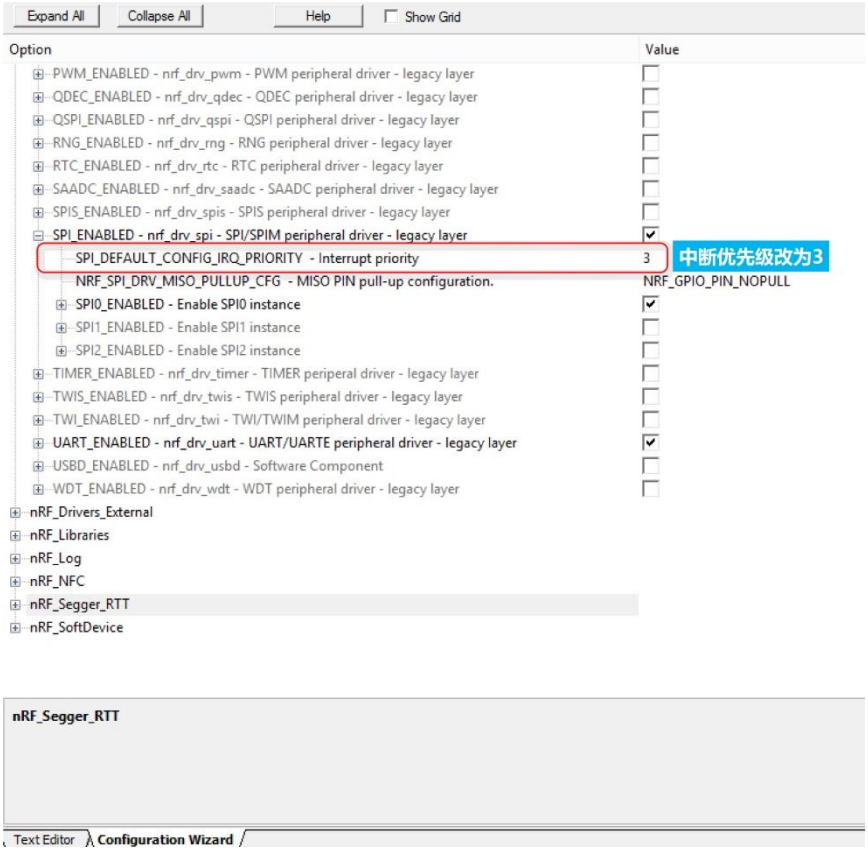


图 12-6：修改 SPI 中断优先级

4.3. 硬件连接

同“实验 12-1：RTC 实现时钟日历(OLED 显示)”。

4.4. 实验步骤

1. 解压“…\3：开发指南（下册）配套实验源码\”目录下的压缩文件“实验 12-2：蓝牙 BLE 下实现时钟日历(OLED 显示，使用外部 32.768KHz 晶体)”，将解压后得到的文件夹“ble_app_uarts_calendar”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_uarts_calendar\project\mdk5”目录下的工程“ble_app_uarts_calendar.uvproj”。
4. 切换到协议栈 target，下载协议栈。
5. 切换到应用程序 target，点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
6. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
7. 程序运行后，OLED 会显示默认设置的时间并且每秒更新一次时间，如下图所示。
8. 通过 BLE 同步时间：手机上打开“nRF CONNECT”，点击 SCAN 按钮扫描设备，扫描到“BLE_UartApp_Cal”后，连接设备，连接成功后展开服务列表，点击向上的小箭头

发送时间同步信息，如下图所示。

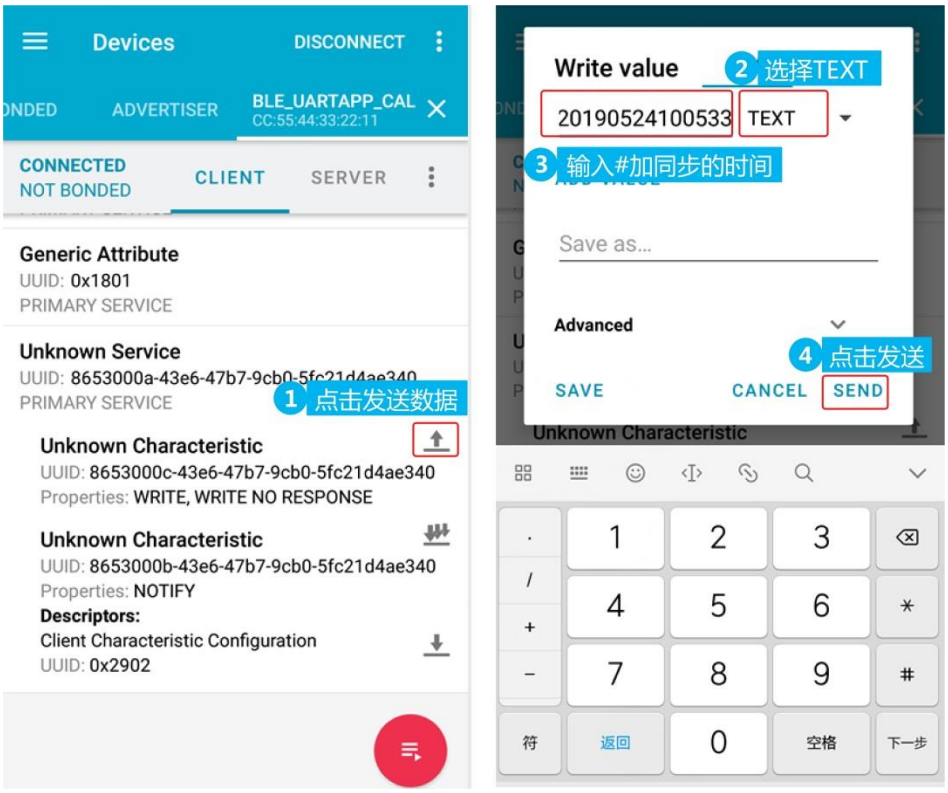


图 12-7：BLE 同步时间

时间同步数据格式是‘#’加上时间数据，如#20190523153528，表示 2019 年 5 月 23 日 15 点 35 分 28 秒。

9. 观察 OLED 显示的时间

时间同步完成后，观察开发板上的 OLED，OLED 会从同步的时间开始显示，并且每秒刷新一次显示。

■ **思考题 1：BLE 实现时钟日历如何使用内部 RC 作为低频时钟源？**

提示：根据开发指南下册的 4.5.2 节：配置低频时钟修改低频时钟源，实验源码：

- 实验 12-3：蓝牙 BLE 下实现时钟日历(OLED 显示，使用内部 RC)

■ **思考题 2：时钟日历程序中，不使用 SPI 驱动 OLED，而是使用 GPIO 模拟 SPI 驱动 OLED，应该怎么实现？**

提示：只需将 OLED 驱动程序中的写 OLED 函数改为模拟 SPI 驱动即可，实验源码：

- 实验 12-4：蓝牙 BLE 下实现时钟日历(模拟 SPI 驱动 OLED)。

第十三章：MESH：运行第一个 MESH 例子

写在前面话：

BLE Mesh 涉及到了很多新的概念，从现在开始我们会逐步接触到这些概念，艾克姆科技 Mesh 的教程是从开发者角度编写的，教程从运行第一个 MESH 例子开始，逐步讲解开发环境的搭建、代码的实现、Mesh 规格在代码中的体现。同时，Mesh 使用了 SES 集成开发环境，很多开发者熟悉的是 keil、IAR 集成开发环境，因为也需要花时间来熟悉 SES。

总之，学习 MESH 是需要花费一定时间和精力的，“枯燥”的学习现在开始，“Are you ready？Go！”。

1. 学习目的

1. 逐步理解 BLE Mesh 涉及的基本概念，对于红字标注的概念一定要做到深刻理解。
2. 了解 Nordic 发布的用于 Mesh 的 SDK。BLE Mesh 会使用新的 SDK，Nordic 发布的用于 Mesh 的 SDK 名称是：nrf5SDKforMesh，当前版本是 V4.0，下文简称 Mesh SDK。
3. 了解 MESH 实验的流程。

2. 实验概述

蓝牙从 4.0 版本开始，加入了全新的蓝牙低功耗技术（BLE），BLE 最重要的特点就是省电，可以使一粒纽扣电池连续工作数年，同时由于 BLE 优异的互操作性、安全性等诸多特性，让 BLE 技术迅速得以大范围应用，各种低功耗设备如 beacon、手环、键鼠、智能遥控器、智能锁层出不穷，可以说“BLE”已无处不在。

自从 BLE 发布以来，广大用户一直关心一个问题：BLE 支持 Mesh 组网吗？终于，蓝牙技术联盟（SIG）不负众望，于 2017 年 7 月发布了 Mesh 的首个版本。各个芯片厂商也开始发布 Mesh 的 SDK，Nordic Semiconductor 作为 ULP 无线技术全球领先的方案厂商，在 Mesh 规格发布的当天即推出首个 Mesh SDK 版本，以帮助开发者快速构建基于蓝牙 Mesh 的应用，截止当前时间，Mesh SDK 已更新到 4.0 版本。

■ **什么是 Mesh**

BLE Mesh 实现的是无线设备之间的多对多（many-to-many m:m）的互通性。Mesh 能让我们创建基于成千上万个设备的大型网络，网络中的设备可以互相安全可靠地通信，同时支持数据中继，从而可以辐射更大的物理区域。蓝牙 MESH 支持一个网络中最多 32767 个设备，最大网络直径为 126 跳。由此可见，BLE Mesh 无疑为大型设备网络的理想之选，可以预见，BLE Mesh 必将在楼宇自动化，无线传感器网络，资产跟踪等应用中大显身手。

■ **第一个实验实现的功能**

对于无线实验来说，最经典的就是点灯实验，本章我们要运行的第一个实验同样是灯光控制实验，即 Mesh SDK 里面的“light_switch”实验。

在进行“light_switch”实验之前，我们需要先了解一些基本的概念，以方便我们执行实验步骤（重要概念）。

- Device（设备）：能够被配置到 Mesh 上的实体，也就是 Mesh 中的节点在配置前称为 Device。
- Node（节点）：经过 Provisioner 配置的 Device，Node 可以在网状网络中发送或接收消息。
- Provisioner（预置器）：能够将设备添加到网状网络的节点。

因此，对于 MESH 网络来说，一个设备必须经过 Provisioner 的配置才能够入网，该设备在没入网之前称为 Device，配置成功加入网络后称为网络中的 Node，设备成为 Node 后即可收/发消息了。

接下来，我们完成“light_switch”实验，使用手机作为 Provisioner，直观地感受一下配置过程。

■ **软硬件需求**

表 13-1：第一个 MESH 实验软硬件需求

序号	名称	描述
1	nRF.Mesh-V2.1.4.apk	安卓手机端 APP。IOS 系统自行到苹果商店下载安装。
2	IK-52840DK 开发板	至少 2 块。
3	J-LINK V9 仿真器（含转接板和转接线）	1 个。
4	USB 数据线	若干。

3. 实验步骤

本实验中使用手机作为 Provisioner 配置设备入网，实验示意图如下，这里我们直接使用 Mesh SDK 里面已经编译好的“light switch”固件。

将两块开发板分别烧写“light switch Server”固件和“light switch Client”固件，之后通过手机分别对它们进行配置，配置成功后，按下 Client 上的 S1 按键，Server 上的指示灯 D1 点亮，按下 Client 上的 S2 按键，Server 上的指示灯 D1 熄灭。

可以看到，这里面有 3 个主要的步骤。

1. 烧写固件：固件烧写后，开发板就是一个“能够被配置到 MESH 上的实体”，这时它是“Device”。
2. 配置：Provisioner 将烧写好固件的开发板配置为网络中的节点，配置完成后开发板成为“Node”。
3. 收/发消息：按下 Client 上的按键控制 Server 上指示灯的亮灭。

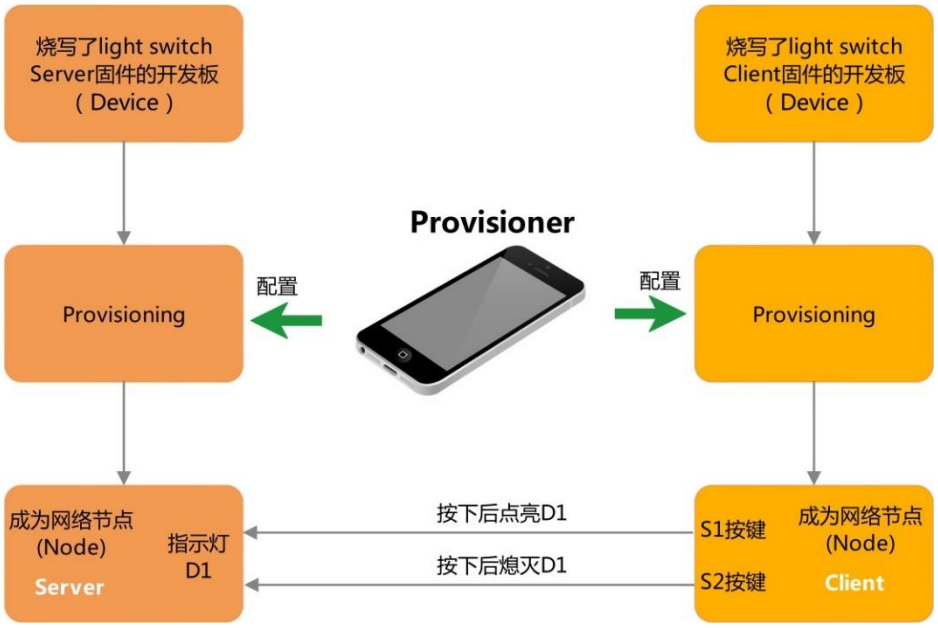


图 13-1：手机作为 Provisioner 实验示意图

3.1. 烧写和配置 Sever

Mesh SDK 中提供了已经合并好协议栈的 Light switch 例子的固件，在 Mesh SDK 中的路径：nrf5_SDK_for_Mesh_v4.0.0_src\bin\merged\ospace\examples\light_switch。

1. 烧写作为 Sever 的 IK-52840DK 开发板

- 打开 J-Flash，将作为 Sever 的 IK-52840DK 开发板烧写固件“light_switch_server_nrf52840_xxAA_s140_7.0.1_merged_sd.hex”。
- 烧写成功后，开发板的 4 个 LED 指示灯同时闪烁 2 次，指示程序正常运行。

2. 配置 Sever

1) 手机上打开 APP：“nRF Mesh”，点击“ADD NODE”启动扫描，扫描到“nRF5x Mesh Light”后，点击“nRF5x Mesh Light”，如下图所示。

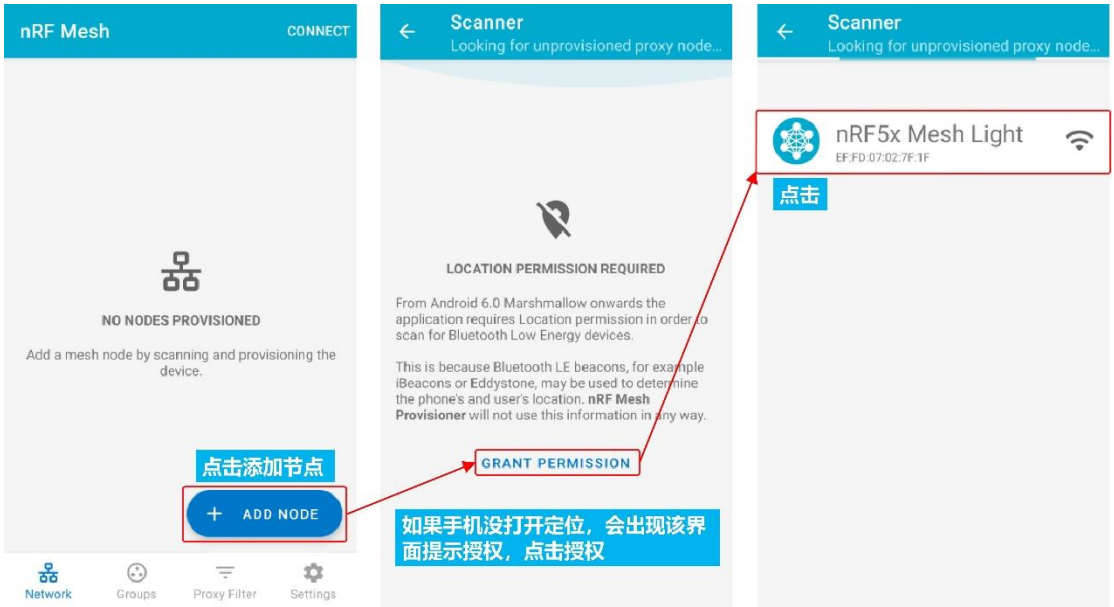


图 13-2：扫描设备

2) 点击[IDENTIFY]，之后会观察到开发板上指示灯 D3 和 D4 快速闪烁一段时间。点击[PROVISION]启动配置流程，配置过程需要一段时间，在这个过程中会观察到开发板的 4 个指示灯 D1~D4 同时闪烁 3 次表示配置成功，之后弹出配置完成窗口，点击[OK]完成配置。

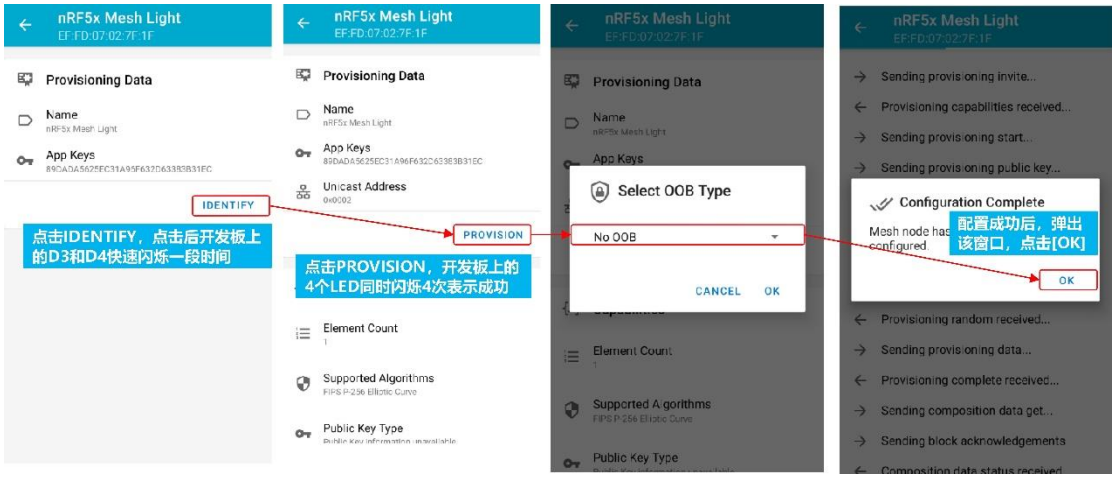


图 13-3：配置 Server

3) 点击配置按钮，之后点击“Generic On Off Server”。

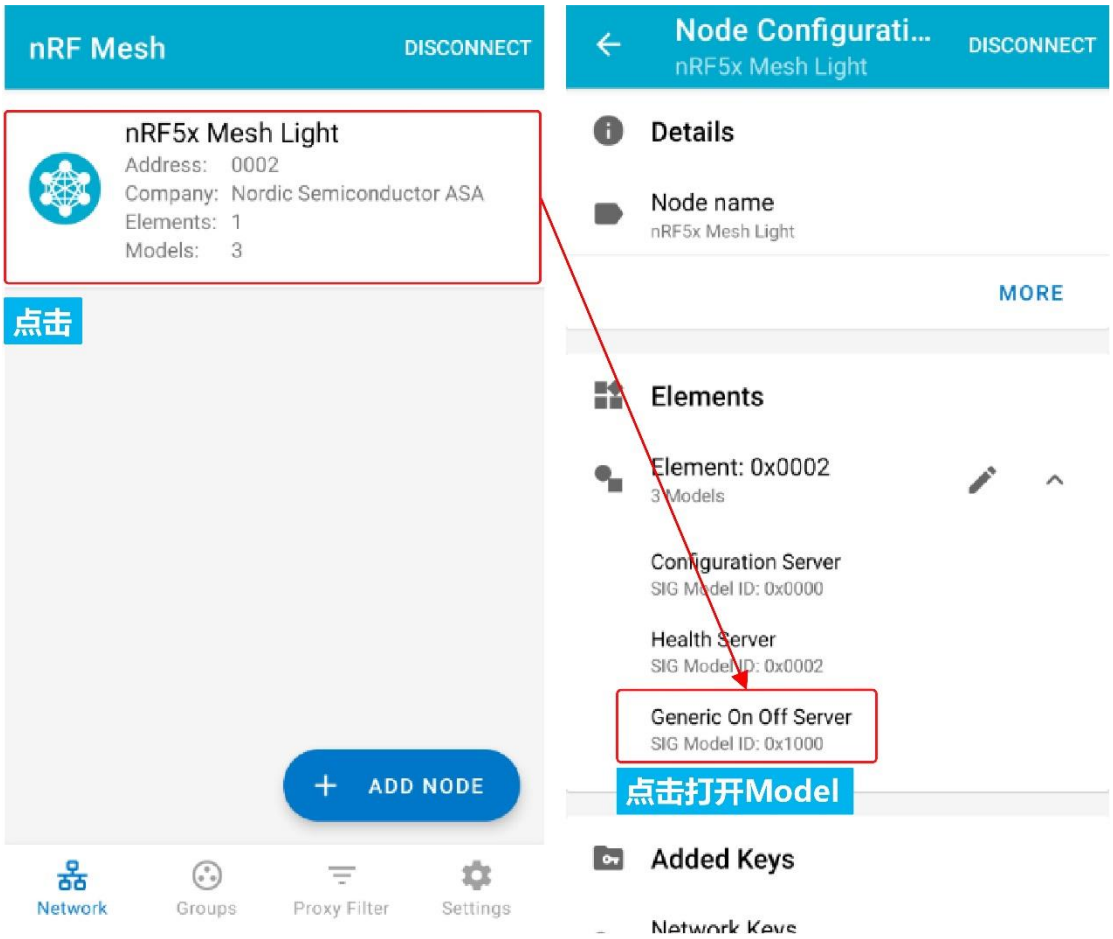


图 13-4：打开 Sever Model

4) 点击[BIND KEY]，绑定 NetKey，如下图所示。

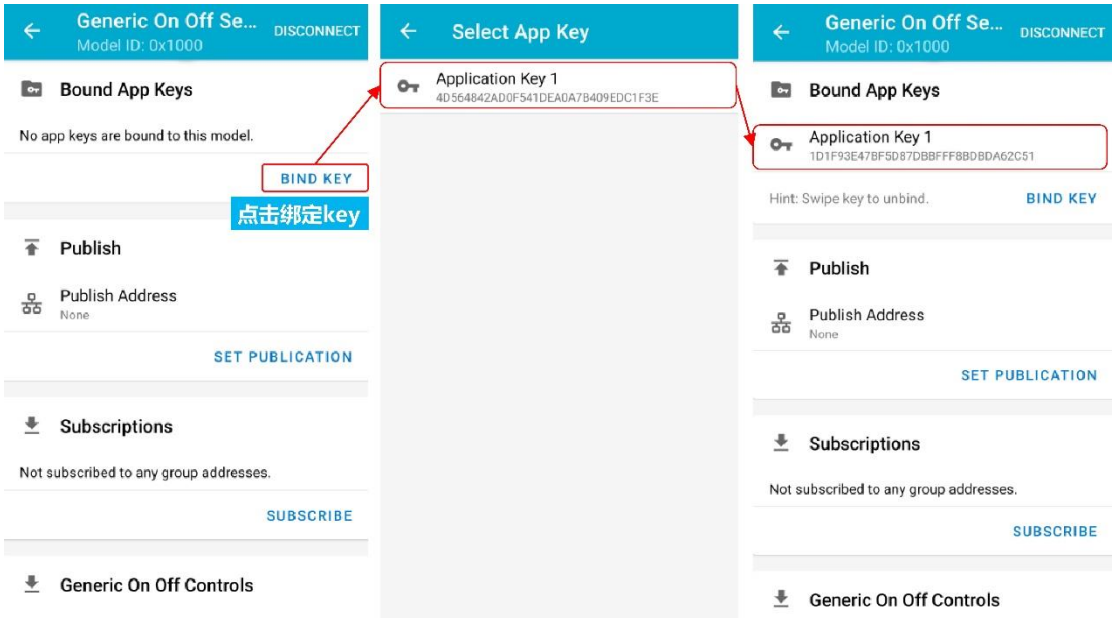


图 13-5：绑定 NetKey

5) 绑定 NetKey 完成后，可以通过 APP 控制开发板上的指示灯 D1 的亮灭和读取指示灯状态，如下图所示。

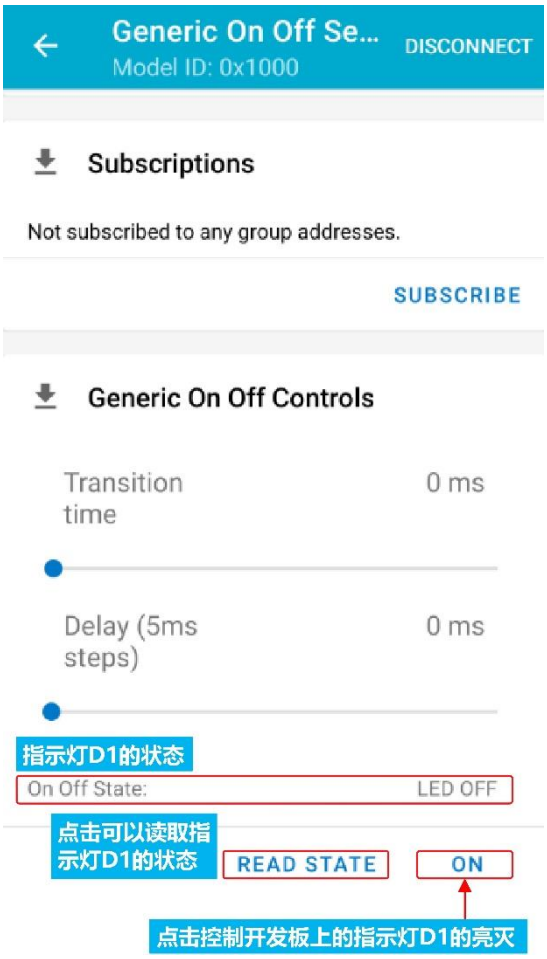


图 13-6：APP 控制 D1 和读取 D1 状态

3.2. 烧写和配置 Client

1. 烧写作为 Client 的 IK-52840DK 开发板

1) 打开 J-Flash，将作为 Client 的 IK-52840DK 开发板烧写“light_switch_client_nrf52840_xxAA_s140_7.0.1_merged_sd.hex”固件。

2) 烧写成功后，开发板的 4 个 LED 指示灯同时闪烁 2 次，指示程序正常运行。

2. 配置 Sever

1) 手机上打开 APP：nRF Mesh，点击“+”启动扫描，扫描到“nRF5x Mesh Switch”后，点击“nRF5x Mesh Light”，如下图所示。

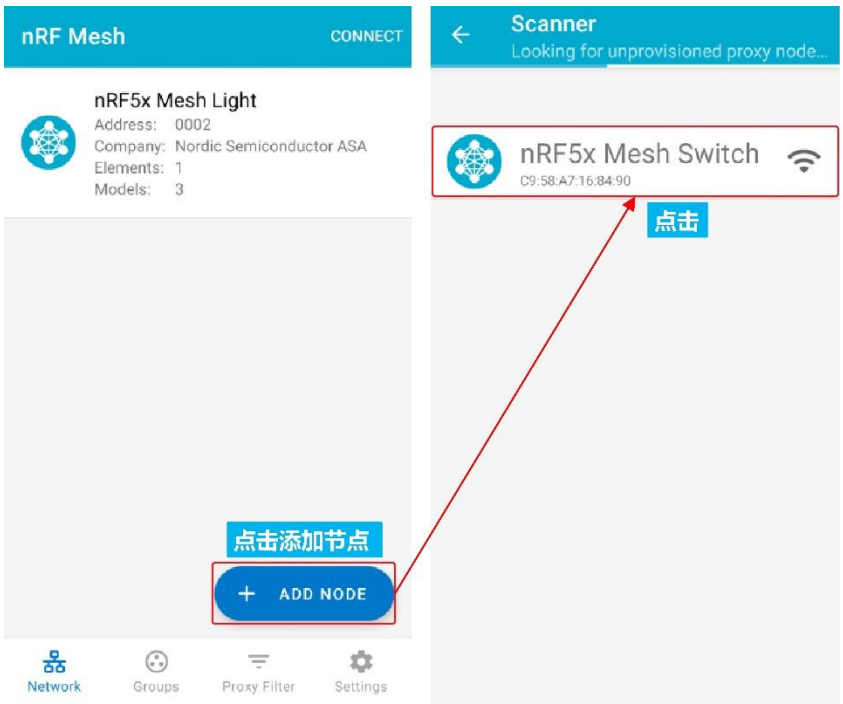


图 13-7：扫描设备

2) 点击[IDENTIFY]，之后会观察到开发板上指示灯 D3 和 D4 快速闪烁一段时间。点击[PROVISION]启动配置流程，配置过程需要一段时间，此过程中会观察到开发板的 4 个指示灯同时闪烁 3 次表示配置成功，之后弹出配置完成窗口，点击[OK]完成配置。

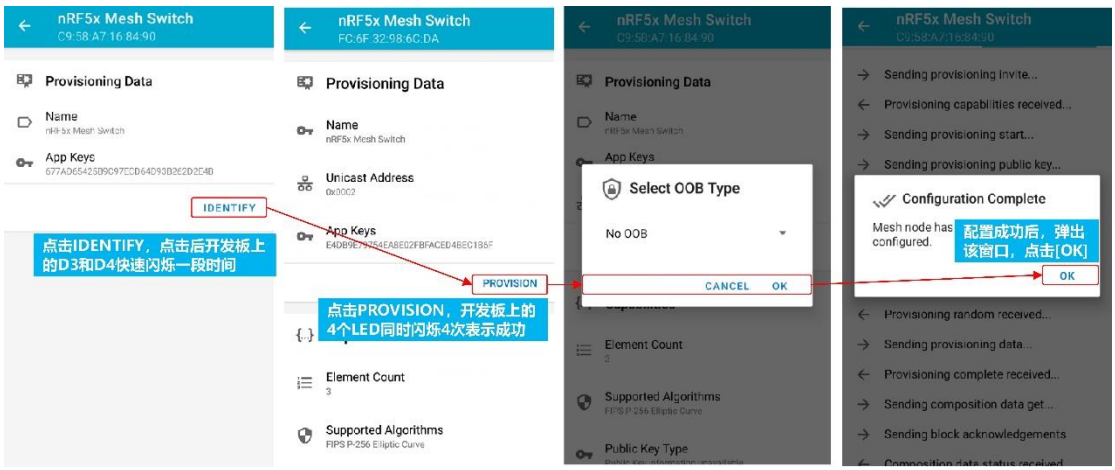


图 13-8：配置 Client

3) 点击配置按钮，之后点击展开“Element:0x0004”列表，如下图所示。

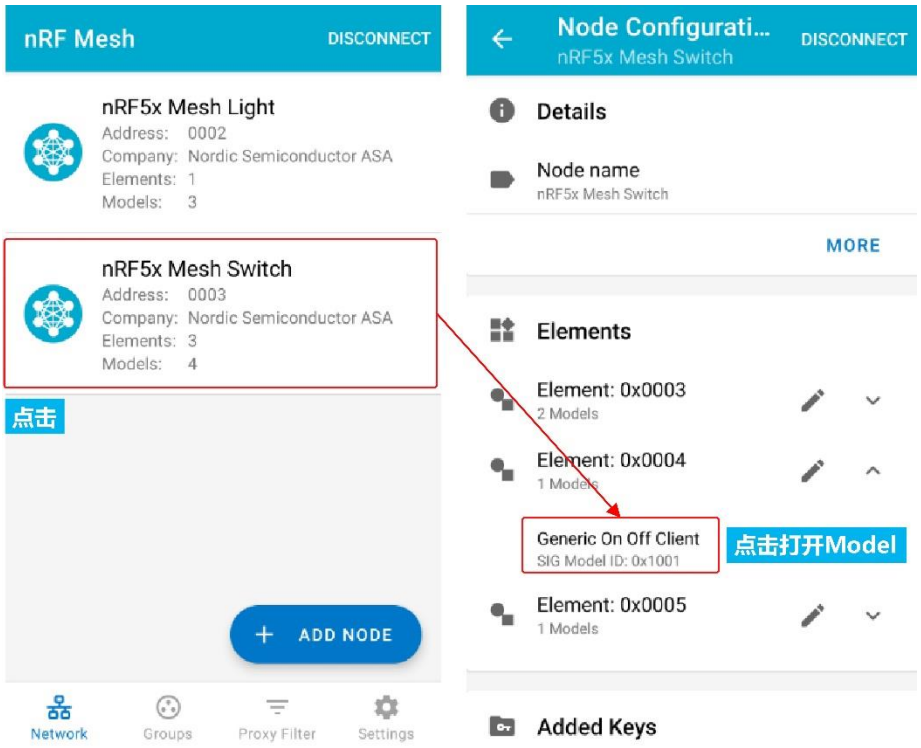


图 13-9：展开“Element:0x0004”

4) 点击[BIND KEY]，绑定 NetKey，如下图所示。

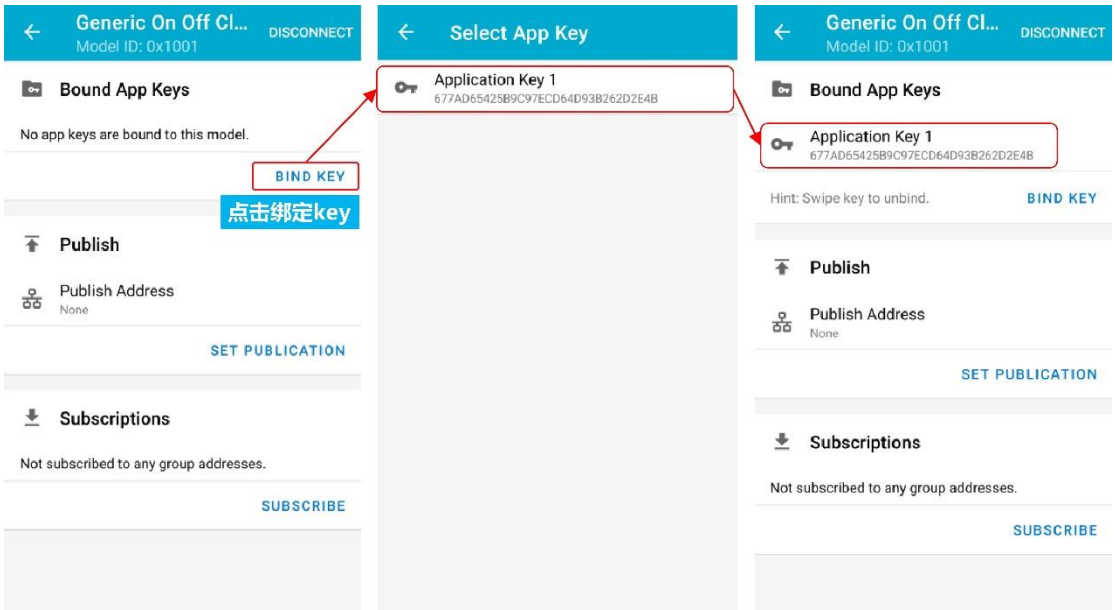


图 13-10：绑定 NetKey

5) 点击“nRF5x Mesh Light”，查看 Sever 的单播地址，如下图所示。

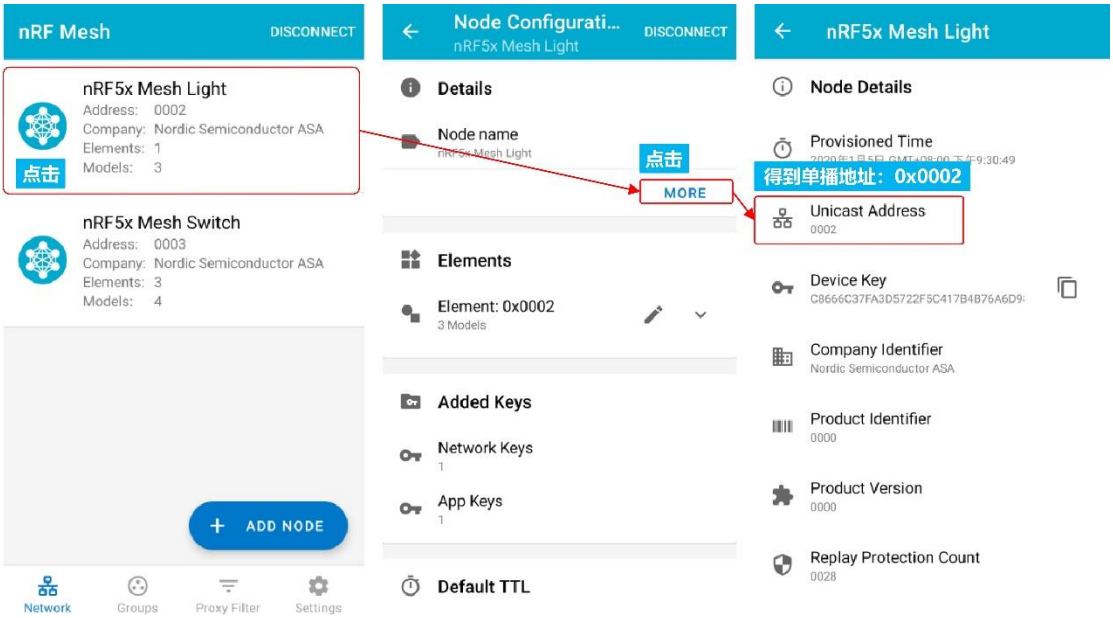


图 13-11：获取 Sever 的单播地址

6) 设置 Client 的 Publish 地址，注意地址输入后一定要点击“APPLY”，如下图所示。

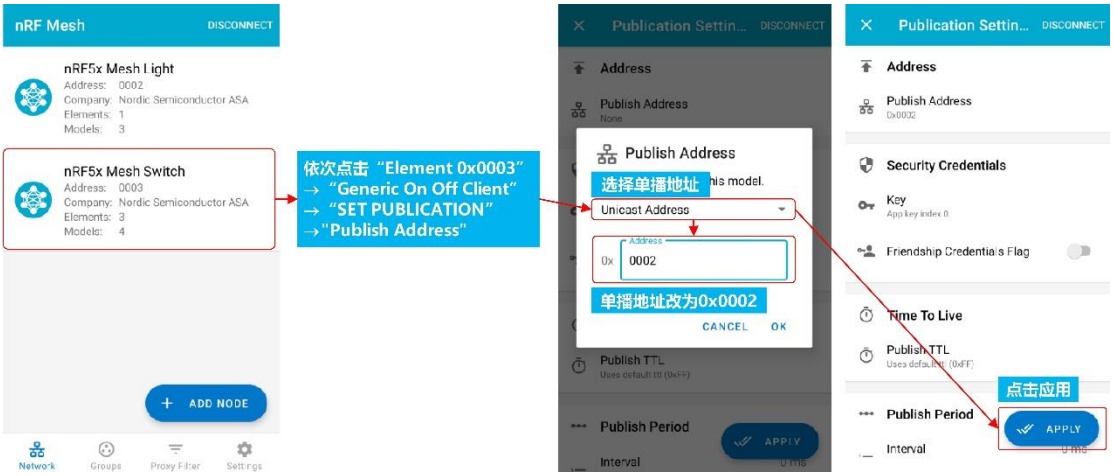


图 13-12：设置 Client 的 Publish 地址

3.3. 测试

经过前面的步骤，两块开发板均已配置入网，通过 Client 上的按键 S1 和 S2 即可发送消息控制 Server 上的指示灯 D1 的亮灭。

- 按下 Client 上的 S1 按键，Server 上的指示灯 D1 点亮，
- 按下 Client 上的 S2 按键，Server 上的指示灯 D1 熄灭

我们还可以继续将一块开发板烧写 Sever 的固件，烧写后按照前文描述的方式配置，配置成功入网后，可以通过 Client 上的按键 S3 和 S4 发送消息控制 Server 上的指示灯 D1 的亮灭。当然，我们可以烧写更多的 Server 并配置入网，

第十四章：搭建 Mesh 开发环境

1. 需要的工具软件

表 14-1：需要的工具软件

序号	软件工具	描述
1	SEGGER Embedded Studio（SES）	SEGGER 推出的集成开发环境。
2	nRF Mesh.apk	安卓手机端 APP。
3	nRF5_SDK_16.0.0_98a08e2.zip	nRF5 软件开发包。
4	nrf5_SDK_for_Mesh_v4.0.0_src.zip	Mesh 软件开发包。

1.1. SEGGER Embedded Studio

SEGGER Embedded Studio，简称 SES，是 SEGGER 公司推出的一款嵌入式跨平台集成开发环境，支持 Windows、Linux 和 MAC 系统。

做单片机、嵌入式开发的朋友有可能对 SEGGER 公司不了解，但是相信大多都用过或听说过“J-LINK 仿真器”，没错，SES 正是设计 J-Link 仿真器的公司 SEGGER 推出的嵌入式集成开发环境。更为重要的是，SES 针对非商业用途是免费的，如果使用 SES 开发 Nordic 的芯片，可以直接申请免费的授权许可。

- 开发板配套资料包里面已经下载好 Window 系统下的 SES 安装文件，安装文件在开发板资料包的“教程盘(B 盘)\ 7 - Mesh\1：所需工具软件”目录下。
- 如果我们使用的是 MAC 或者 Linux 系统，可以到 SEGGER 的网站上下载对应的版本。

下载链接：https://www.segger.com/downloads/embedded-studio/。打开后界面如下，根据自己电脑系统选择对应的版本下载即可。

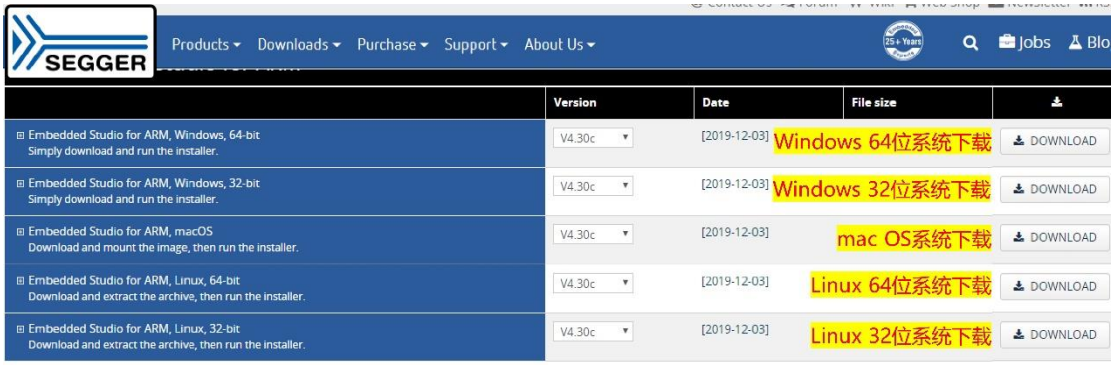


图:14-1：SES 下载页面

✧ **说明**：开发 Nordic 系列的芯片是可以免费使用 SES 的（Nordic 已经为开发者购买了 SES 使用版权），也就是说使用 SES 开发 Nordic 的芯片无需担心版权的问题。

1.2. 手机端 APP

Mesh 手机端用的调试 APP 是“nRF Mesh”，安卓系统将资料包里面的安装文件发送到手机安装即可。安装文件在开发板资料包的“教程盘(B 盘)\7 - Mesh\2：手机端 APP”目录下。

2. 安装 SES

2.1. 安装

1. 双击 SES 安装文件，弹出 SES 安装向导，单击【Next】。

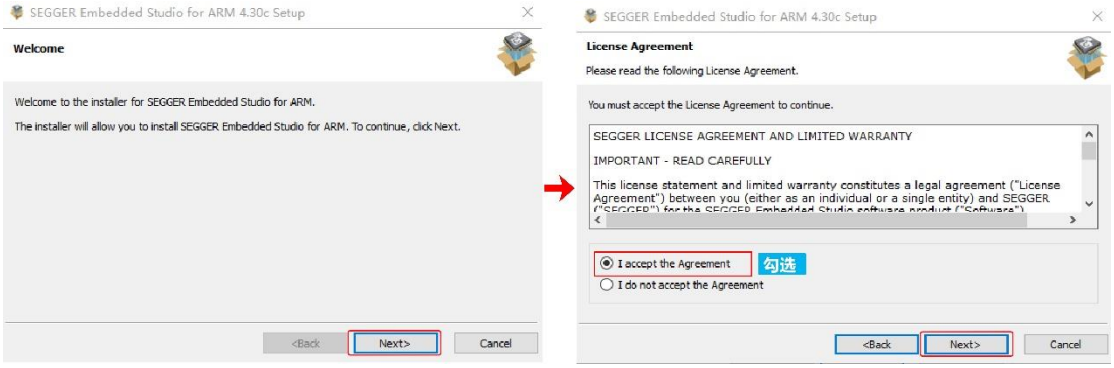


图 14-2：SES 安装

2. 设置安装路径后，单击【Next】，建议使用默认的安装路径。

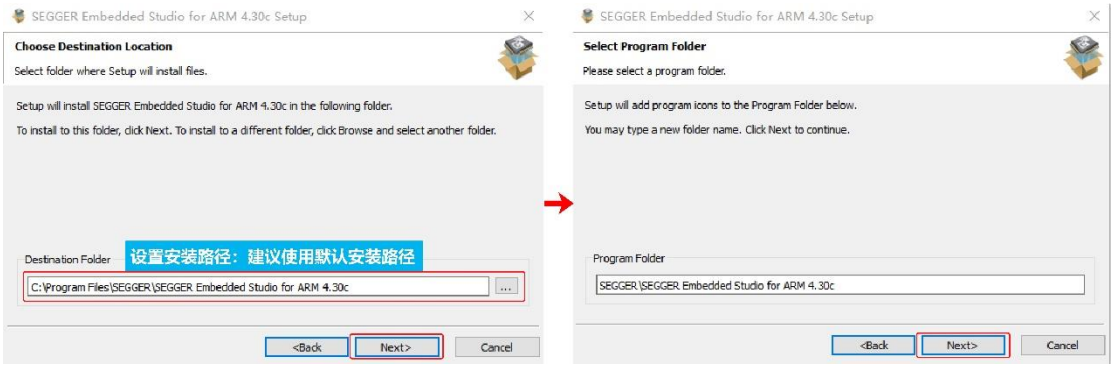


图 14-3：设置安装路径

3. 设置文件关联和安装的附加组件后，单击【Next】。

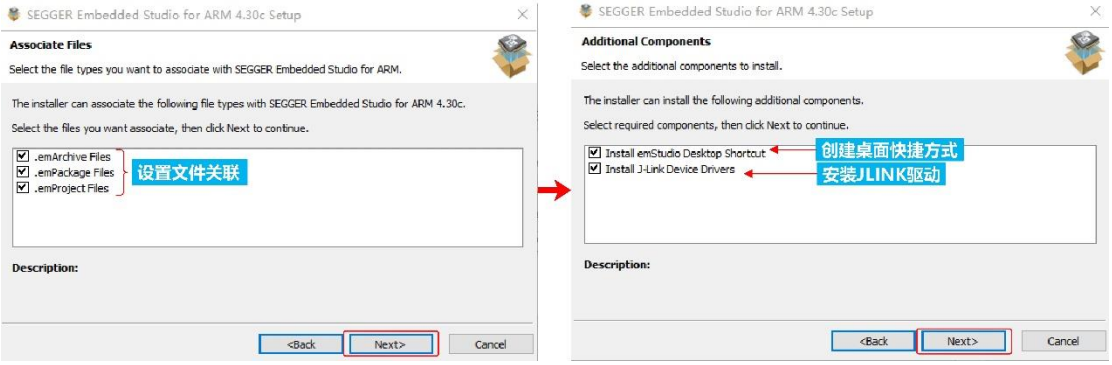


图 14-4：设置文件关联和安装的附加组件

4. 等待 SES 安装进度完成后，单击【Finish】完成安装。

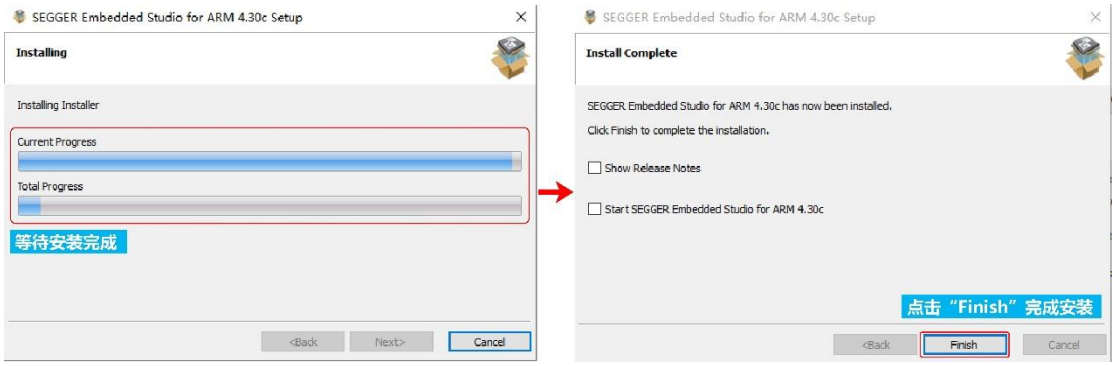
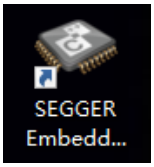


图 14-5：完成安装

2.2. 激活 SES

SES 安装完成后，我们还需要激活才能正常使用，这里我们激活一个免费的许可，步骤如下。

1. 双击 SES 桌面快捷方式图标，启动 SES。



2. 如果弹出了许可申请窗口，忽略这一步。如下图所示，选中工程后右键，在弹出的菜单中点击[Build]，因为我们还没有激活 SES，因此执行编译就会弹出。

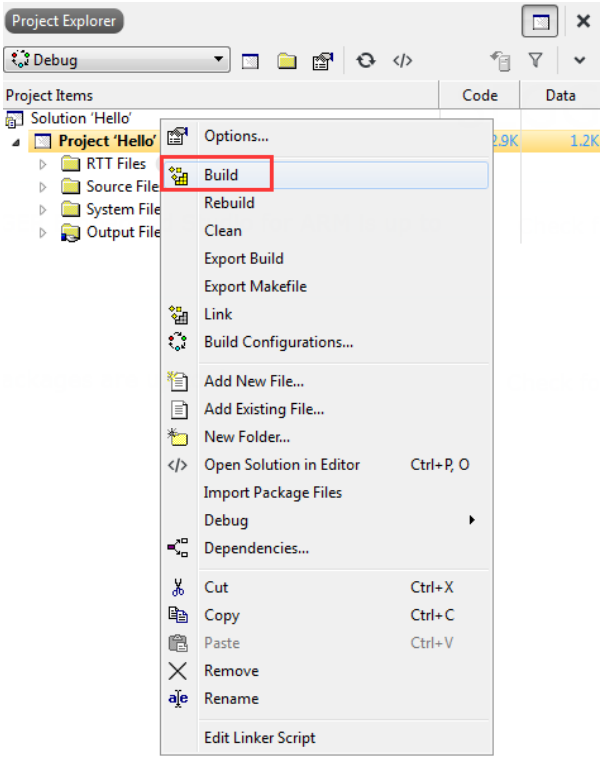


图 14-6：打开许可申请窗口

3. 这时会弹出如下界面，提示我们没有检测到许可，点击“Activate Your Free License”申请许可。

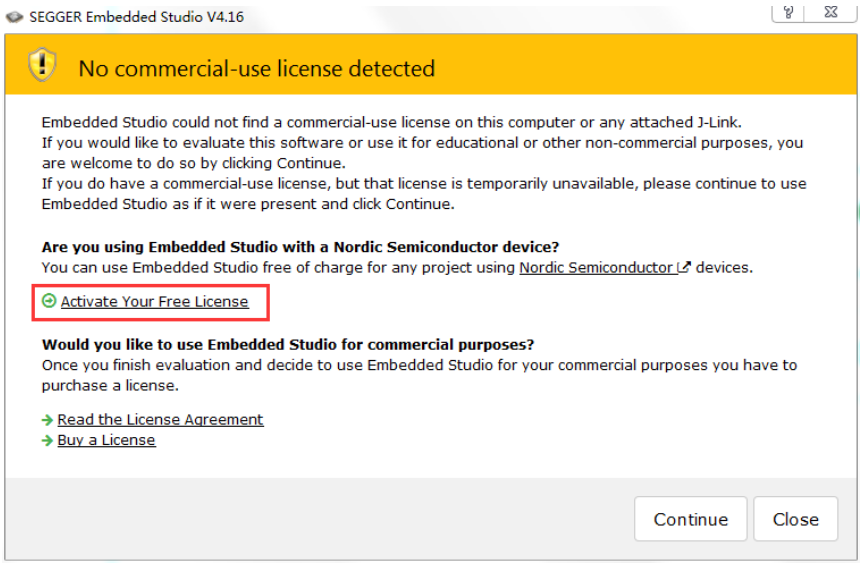


图 14-7：打开许可申请窗口

4. 填写好相关信息后，点击【Request License】，序列号会发到申请的邮箱里面，注意：序列号发送可能会有延迟，如果申请后邮箱没有接收到邮件，稍等一段时间后就会收到邮件。

图 14-8：申请序列号

5. 打开邮箱拷贝序列号，如下图所示，注意需要拷贝的内容。

图 14-9：拷贝序列号

6. “Tools→License Manager”输入序列号，完成激活。

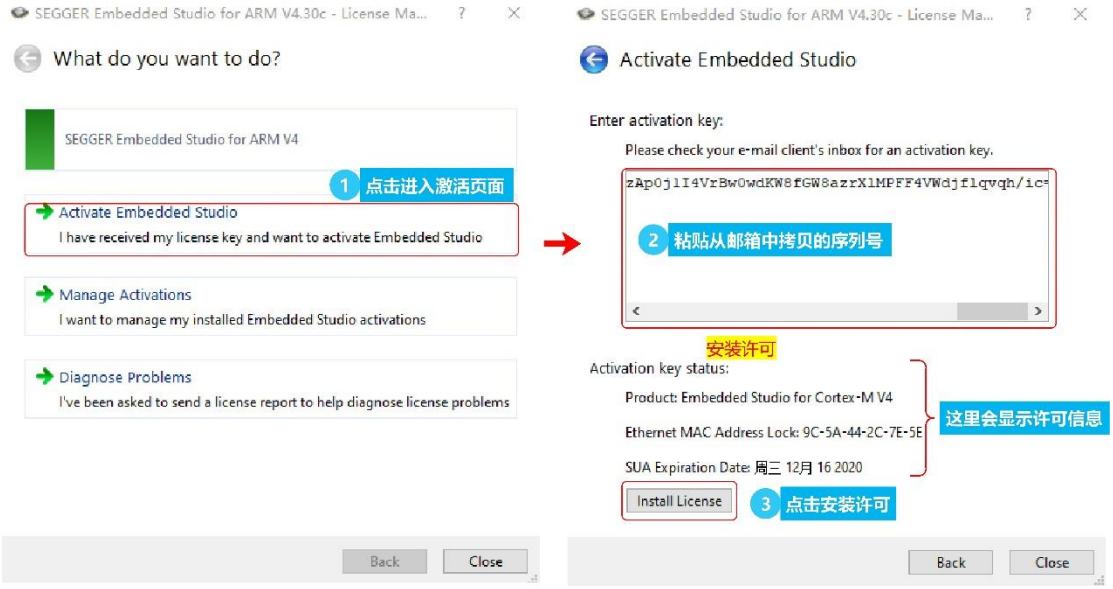


图 14-10：激活 SES

✧ 注意：如果申请序列号后关闭了窗口，可以到“Tools→License Manager”中激活 SES。

2.3. 安装 Pack

执行“Tools→Package Manager”，打开 Pack 管理器，找到“Nordic pack”的安装项，按照下图所示安装 nRF 的 Pack。

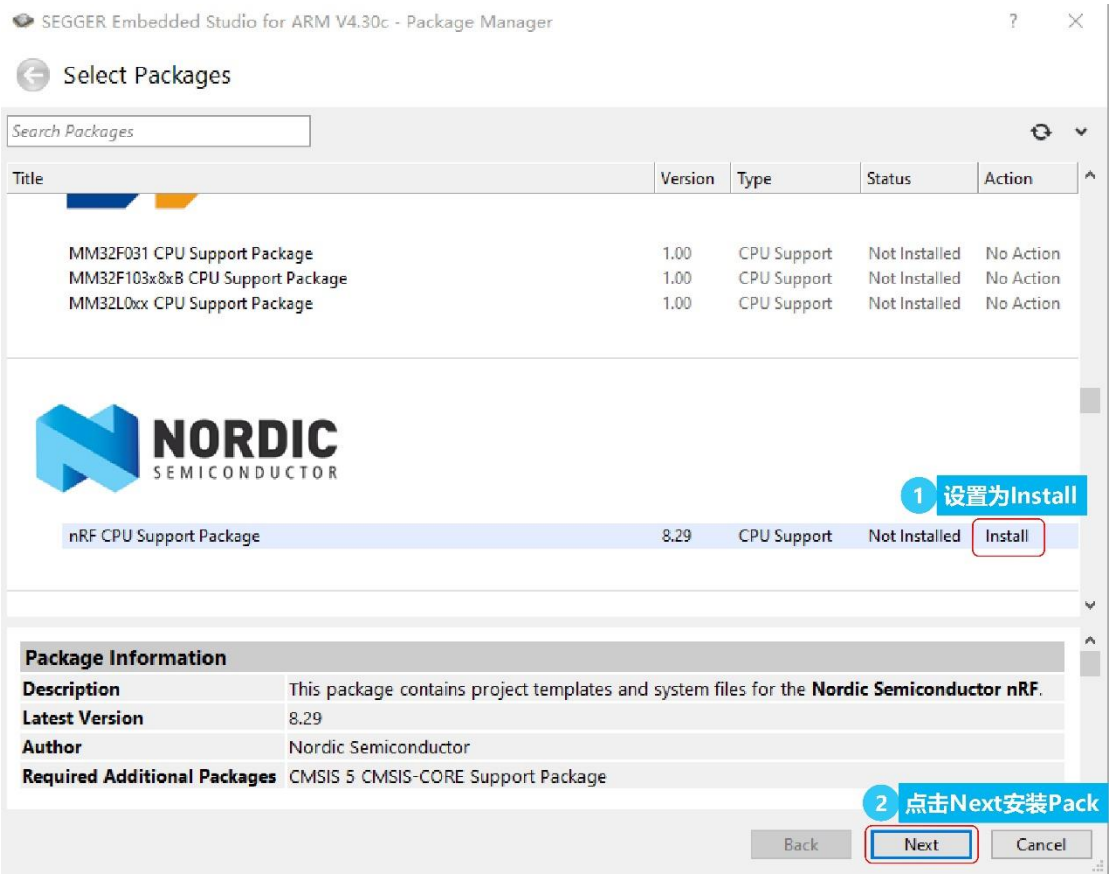


图 14-11：安装 Pack

Pack 安装需要保持联网，安装会占用一些时间，请耐心等待安装完成，安装完成后如下图所示，点击“Finish”完成 pack 安装。

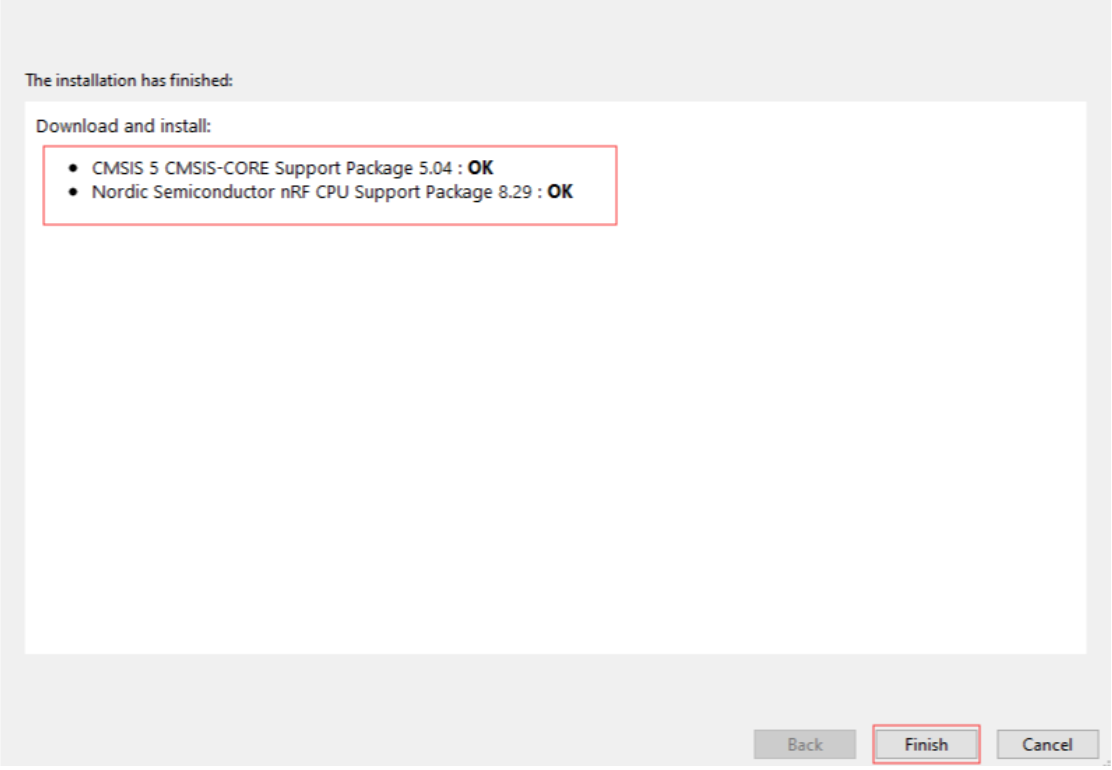


图 14-12： Pack 安装完成

3. 编译 MESH 工程

本节以 MESH SDK 中的“light_switch”的“server”为例说明如何编译 MESH 的工程。MESH 的工程会用到 SDK 中的库文件，因此需要注意 SDK 的版本和放置的路径。

- SDK 版本：MESH SDK-V4.0.0 对应的 SDK 版本是 SDK16.0。
- 将 MESH SDK（V4.0.0）和 SDK16.0 解压到同级目录下，如下图所示。

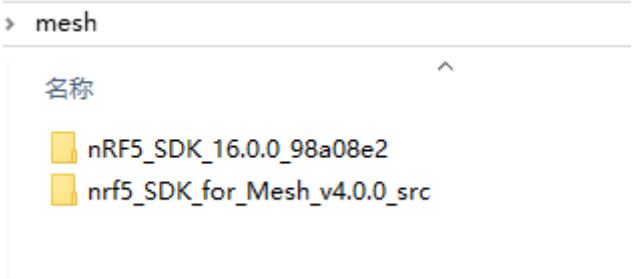


图 14-13：MESH SDK 和 SDK16.0 解压到同级目录

1. 打开工程

打开“..\nrf5_SDK_for_Mesh_v4.0.0_src\examples\light_switch\server”目录，双击“light_switch_server_nrf52840_xxAA_s140_7_0_1.emProject”打开工程。

2. 编译工程

点击编译按钮编译工程，如下图所示。

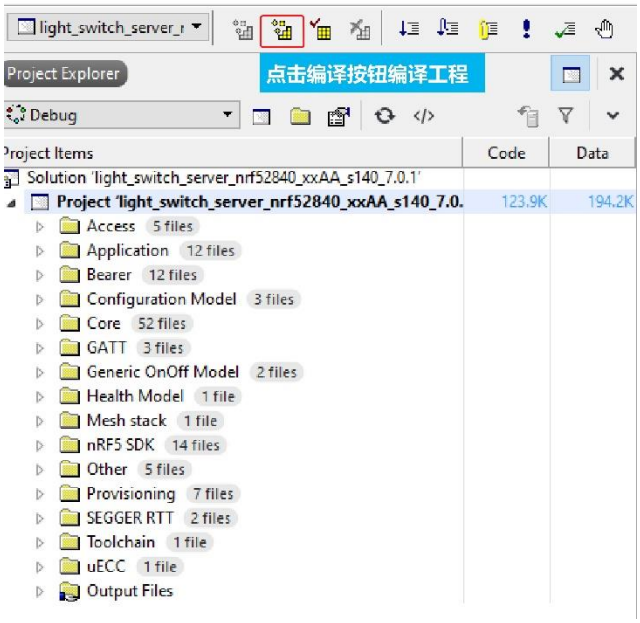


图 14-14：编译工程

编译成功后，信息输出窗口会显示工程编译后的 Flash 和 RAM 大小，如下图所示。

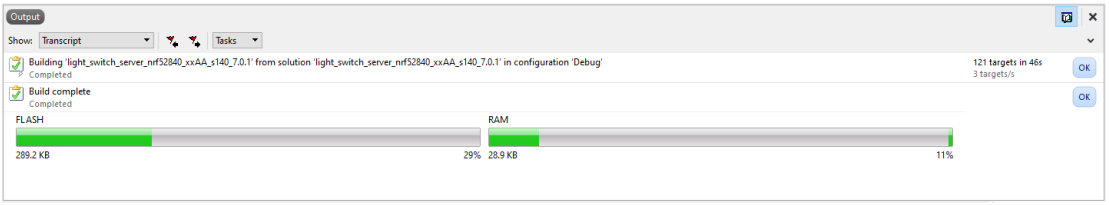


图 14-15：编译成功

参考文献

1. Cortex-M4 Technical Reference Manual。
2. nRF5_SDK_16.0.0_98a08e2，Nordic Semiconductor。
3. nRF5_SDK_16.0.0_offline_doc，Nordic Semiconductor。
4. nRF52840_PS_v1.1，Nordic Semiconductor。
5. Bluetooth Core Specification 4.2，SIG。
6. Bluetooth Core Specification Addendum 6，SIG。
7. Bluetooth Core Specification 5.0，SIG。
8. Bluetooth Core Specification Addendum 7，SIG。
9. Heart Rate Profile(HRP)V10，SIG。
10. Heart Rate Service(HRS) V10，SIG。
11. Device Information Service(DIS) V11r00，SIG。
12. Mesh Profile Specification 1.0.1，SIG。
13. Mesh Model Specification 1.0.1，SIG。
14. Mesh Device Properties 1.2，SIG。