

艾克姆科技

nRF52840 开发指南-下册（主机）

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

艾克姆科技飞字团队

[2019.7.3]

官方店铺: <https://acmemcu.taobao.com>

官方论坛: <http://930ebbs.com>

版权所有：艾克姆科技，如有引用，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 开发板为硬件平台，基于 SDK16.0 的库，针对 BLE 主机开发，从基本概念和新建 BLE 主机工程模板开始，一步步讲解蓝牙 BLE 主机程序框架和流程：扫描、建立连接、MTU 交换、连接参数管理、PHY 更新、服务发现、使能通知、接收通知以及读写从机特征值等]

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
2019.9.20	A	强光手电	彭震	初建(基于 SDK15.3 编写)
2020.1.2	B	强光手电	彭震	修改为 SDK16.0 版本

第一章：新建 BLE 主机工程模板

1. 学习目的

1. 了解 BLE 主机工程的建立过程和配置项目。
2. 了解 BLE 主机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）。

❖ 注：本章对应源码是“实验 1-1：BLE 主机工程模板”。

2. 新建 BLE 工程模板

2.1. 规划工程目录

和 BLE 从机新建工程一样，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、external、modules：从 SDK 中拷贝的库文件。

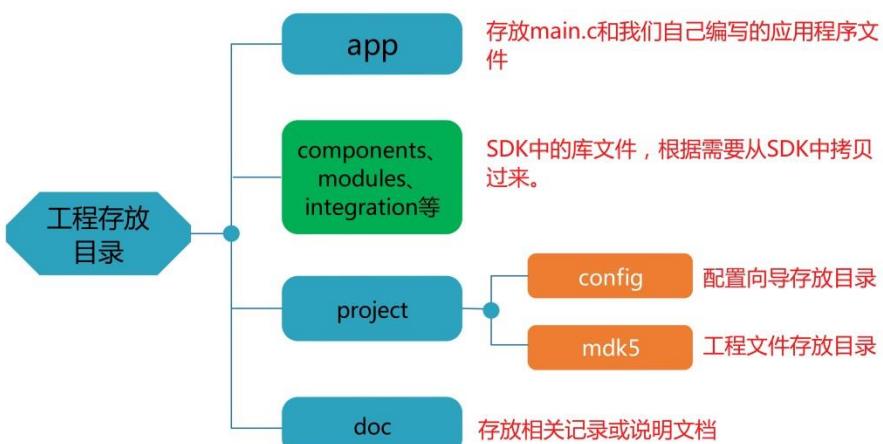


图 1-1：工程存放目录

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_template_c 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 SDK.16.0，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。

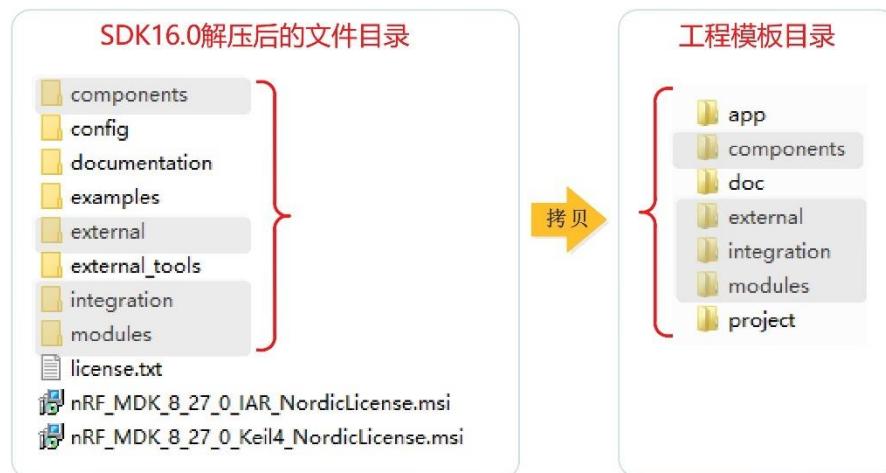


图 1-2：拷贝库文件

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_template_c（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，template 表示模板，c 表示主机（Central：中心设备），工程存放到 D 盘 ble_app_template_c 文件夹。

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

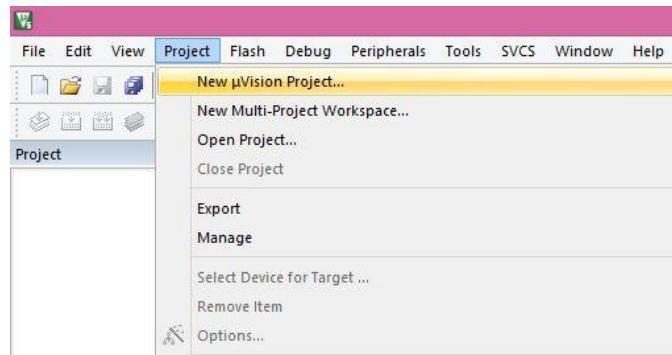


图 1-3：新建工程窗口

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。

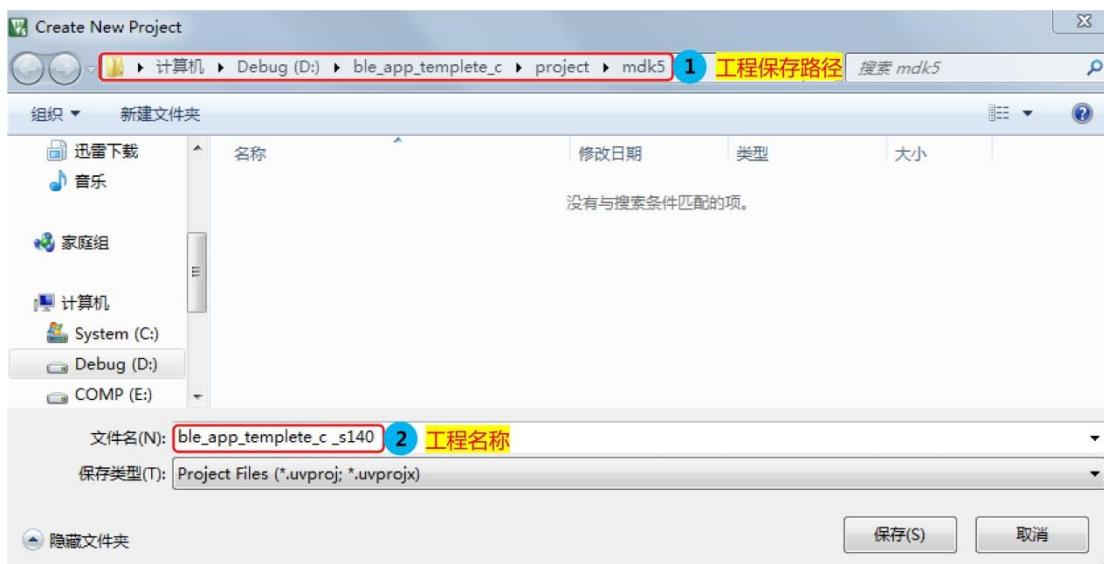


图 1-4：设置工程路径和工程名

保存后，工程名称是：ble_app_template_c，工程保存路径是：“…\ble_app_template\project\mdk5\ble_app_template_c”。

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。

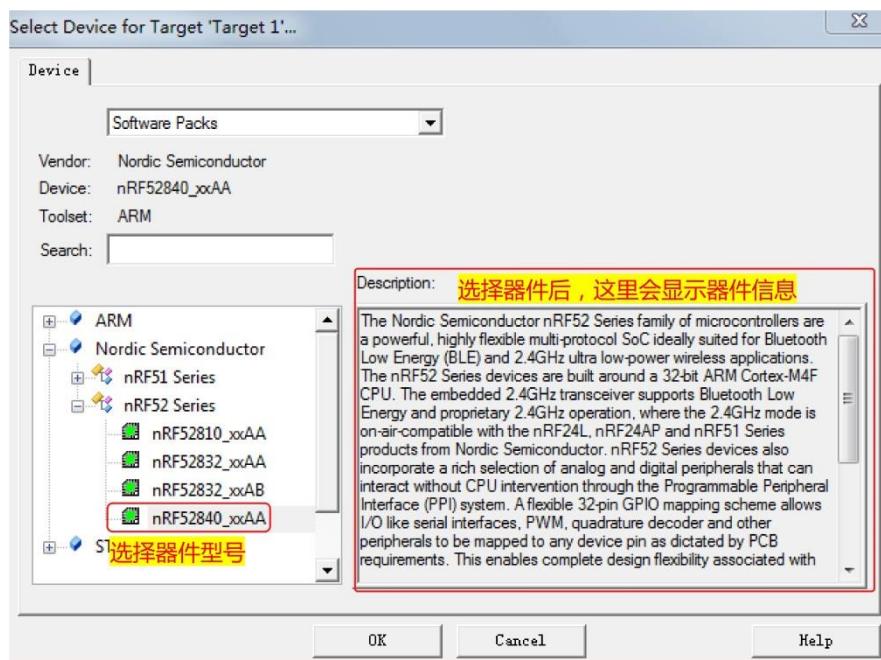


图 1-5：选择器件

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。

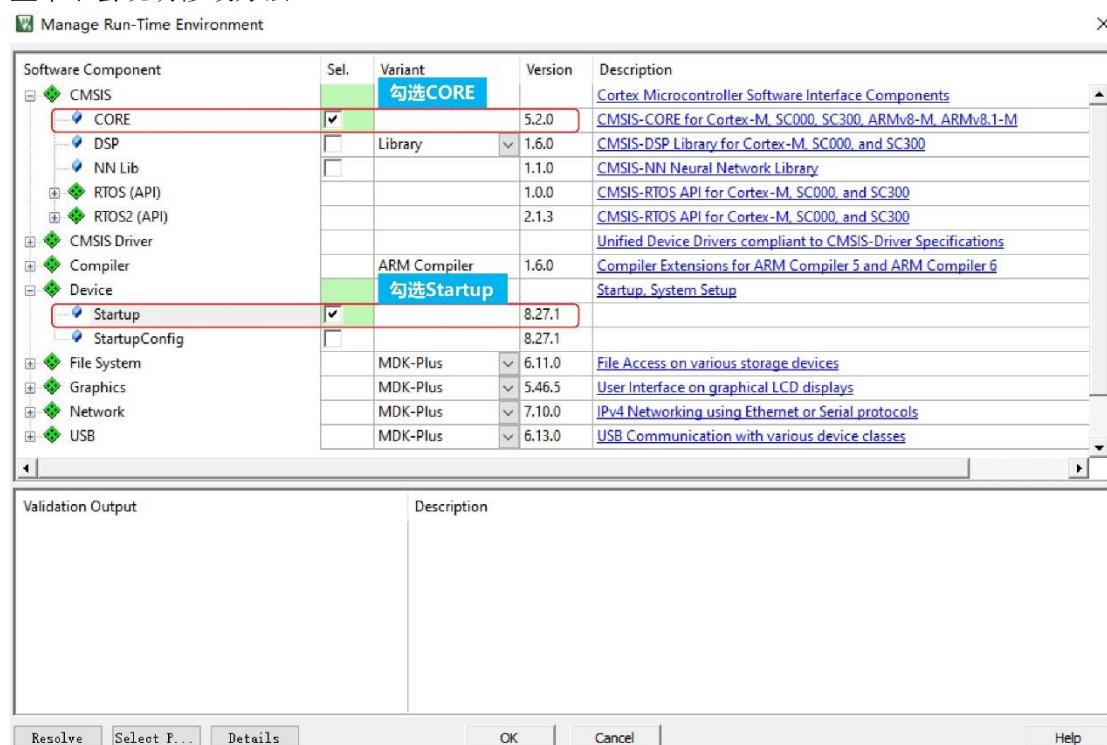


图 1-6: 配置 RTE

❖ **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。

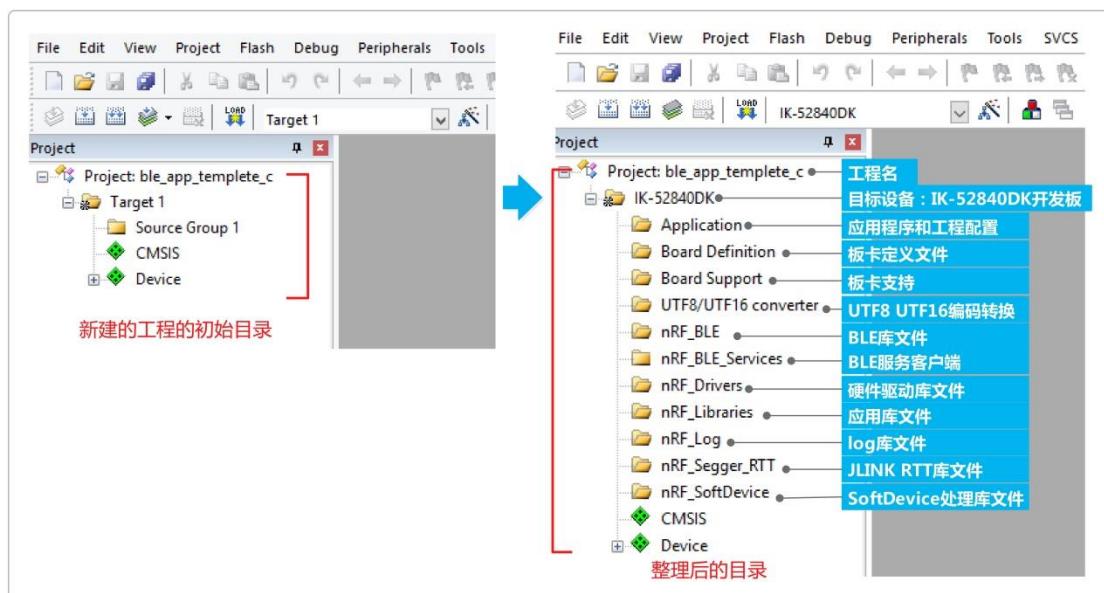


图 1-7：整理工程目录

6. 管理软件包

点击下图中的图标，打开软件包管理窗口。



图 1-8：打开软件包管理窗口

打开的软件包管理窗口如下图所示，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。

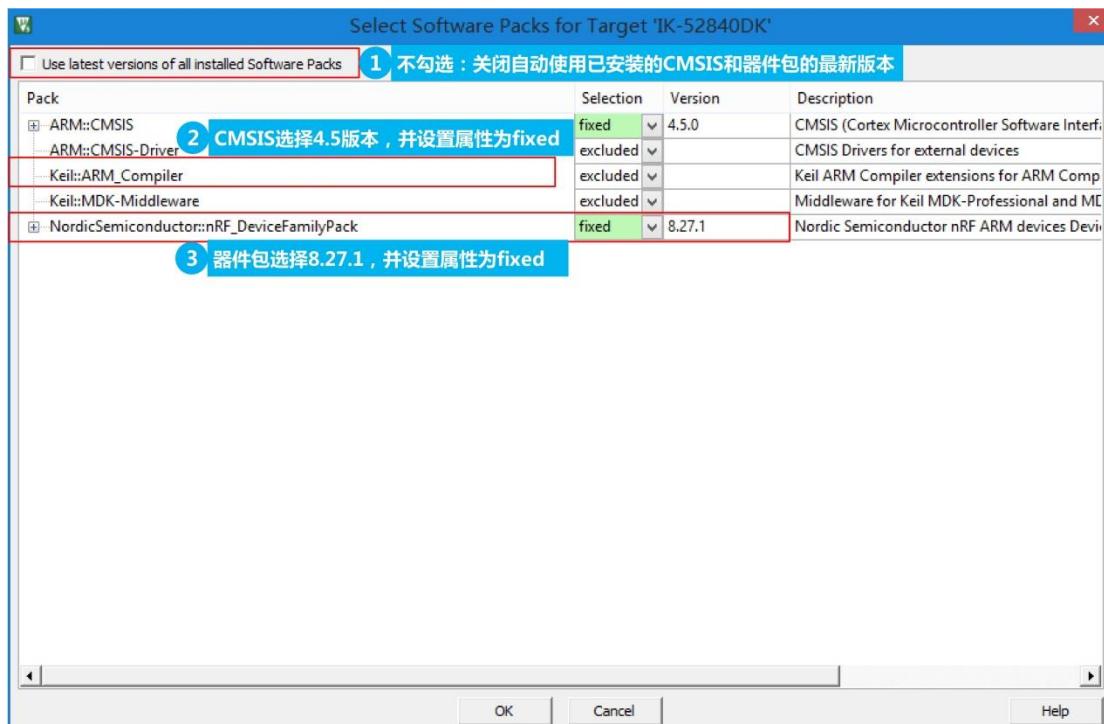


图 1-9：修改 StartUp 和 CORE 的版本

- 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。这里我们不勾选，即关闭自动使用使用最新版本。

- 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 CMSIS 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，并设置版本为 fixed（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）。

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 1-1: “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 1-2: “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 1-3: “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、配对管理等等，“nRF_BLE”组中加入的文件如下表所示。

表 1-4: “nRF_BLE”组中加入的文件

文件名	路径
ble_advdata.c	..\\components\\ble\\common

ble_db_discovery.c	..\..\components\ble\ble_db_discovery
nrf_ble_gatt.c	..\..\components\ble\ble\nrf_ble_gatt
ble_srv_common.c	..\..\components\ble\ble\common
nrf_ble_gq.c	..\..\components\ble\ble\nrf_ble_gq
nrf_ble_scan.c	..\..\components\ble\ble\nrf_ble_scan

5. nRF_BLE_Services 组

预留，后续章节连接具体设备时使用。

6. nRF_Drivers 组

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

表 1-5: “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
nrfx_atomic.c	..\..\modules\nrfx\soc

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 1-6: “nRF_Libraries”组中加入的文件

文件名	路径
app_button.c	..\..\components\libraries\button
app_error.c	..\..\components\libraries\util
app_error_handler_keil.c	..\..\components\libraries\util
app_error_weak.c	..\..\components\libraries\util
app_scheduler.c	..\..\components\libraries\scheduler
app_timer2.c	..\..\components\libraries\timer
app_fifo.c	..\..\components\libraries\fifo
app_uart_fifo.c	..\..\components\libraries\uart
app_util_platform.c	..\..\components\libraries\util
drv_rtc.c	..\..\components\libraries\timer
hardfault_implementation.c	..\..\components\libraries\hardfault
nrf_assert.c	..\..\components\libraries\util
nrf_atfifo.c	..\..\components\libraries\atomic_fifo

nrf_atomic.c	..\..\components\libraries\atomic
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_queue.c	..\..\components\libraries\queue
nrf_ringbuf.c	..\..\components\libraries\ringbuf
nrf_section_iter.c	..\..\components\libraries\experimental_section_vars
nrf_sortlist.c	..\..\components\libraries\sortlist
nrf_strerror.c	..\..\components\libraries\strerror
retarget.c	..\..\components\libraries\uart

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 1-7: “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_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，他们的作用是实现 JLINK-RTT Viewer 作为 Log 输出终端，打印 Log。

表 1-8: “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，他们用于使能或禁止 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，并将他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 1-9: “nRF_SoftDevice” 组中加入的文件

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

2.5. 新建 main.c 文件并添加到工程

- 执行“File→New”新建文件，如下图。

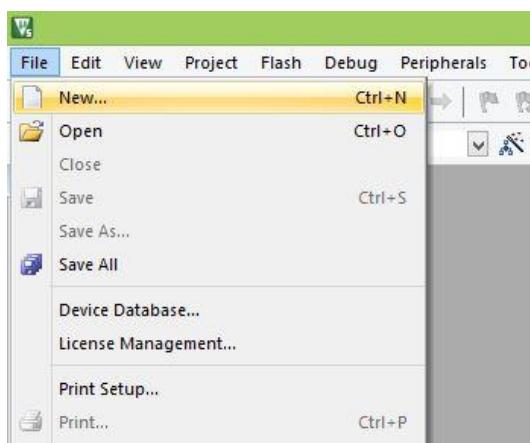


图 1-10: 新建文件

- 点击保存按钮“Save”保存文件，文件名称为“main.c”，保存到工程存放目录的 app 文件里面。
- 将“main.c”文件添加到工程的“Application”组。

2.6. 添加“sdk_config.h”文件

- 从 SDK16.0 的 ble_app_uart_c 程序中拷贝“sdk_config.h”文件到本例的“.. \project\config”目录下
- 将“sdk_config.h”文件添加到工程的“Application”组，如下图所示。

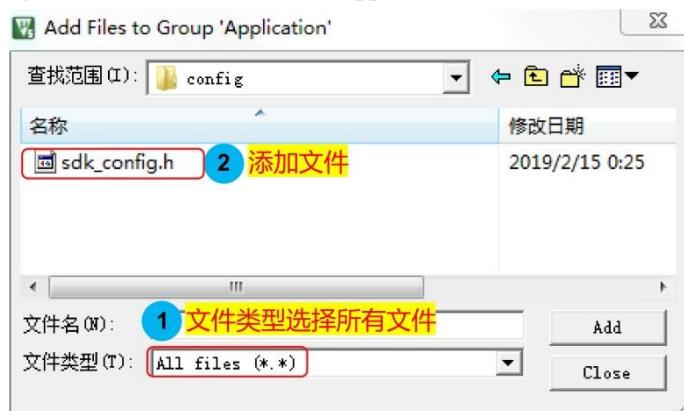


图 1-11: 加入“sdk_config.h”文件

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

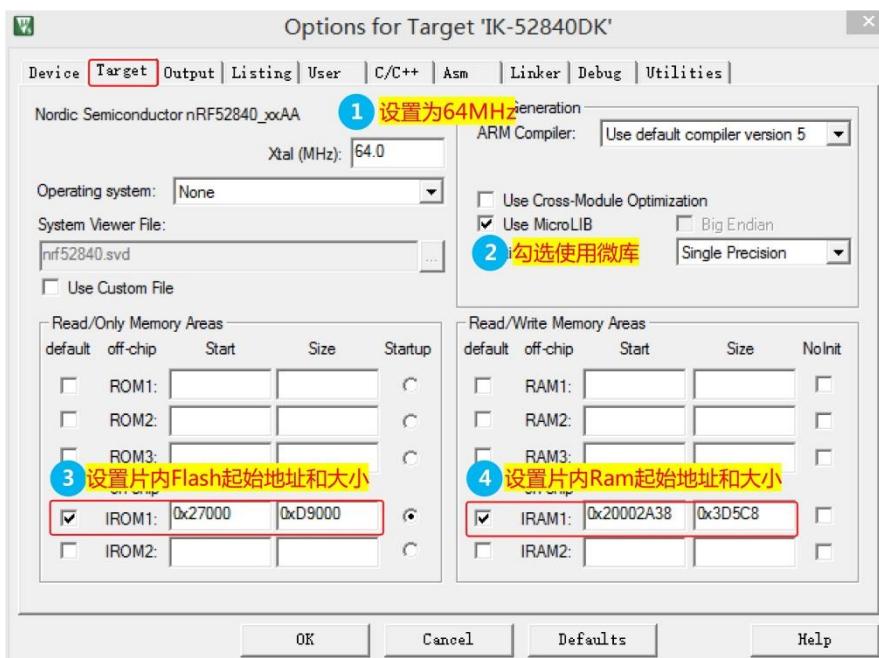


图 1-12：配置“Target”选项卡

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)
0x20002A38	0x3D5C8

3.2. 配置“Output”选项卡

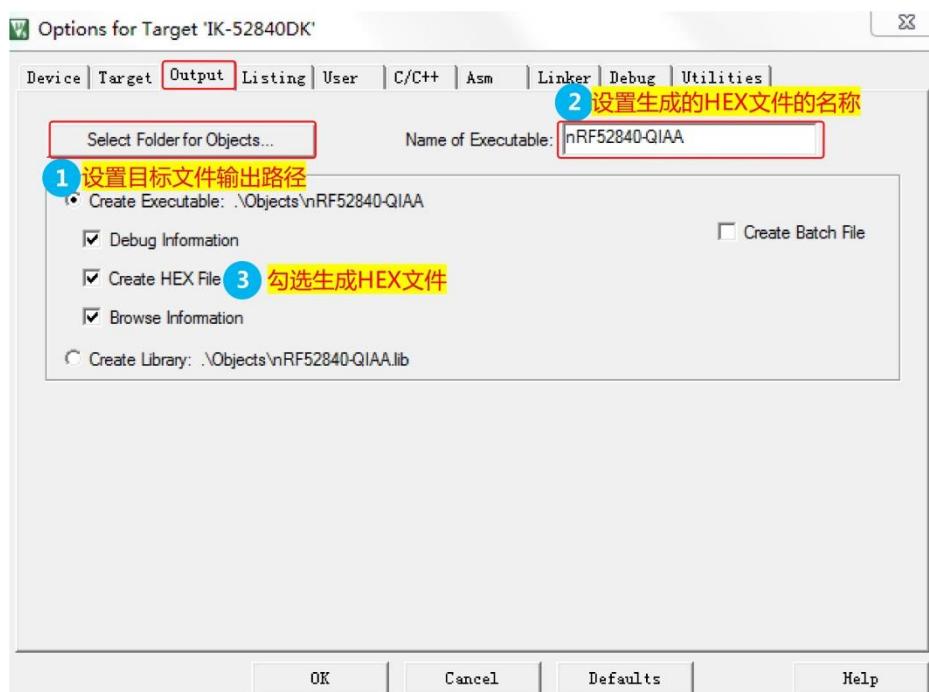


图 1-13: 配置“Output”选项卡

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

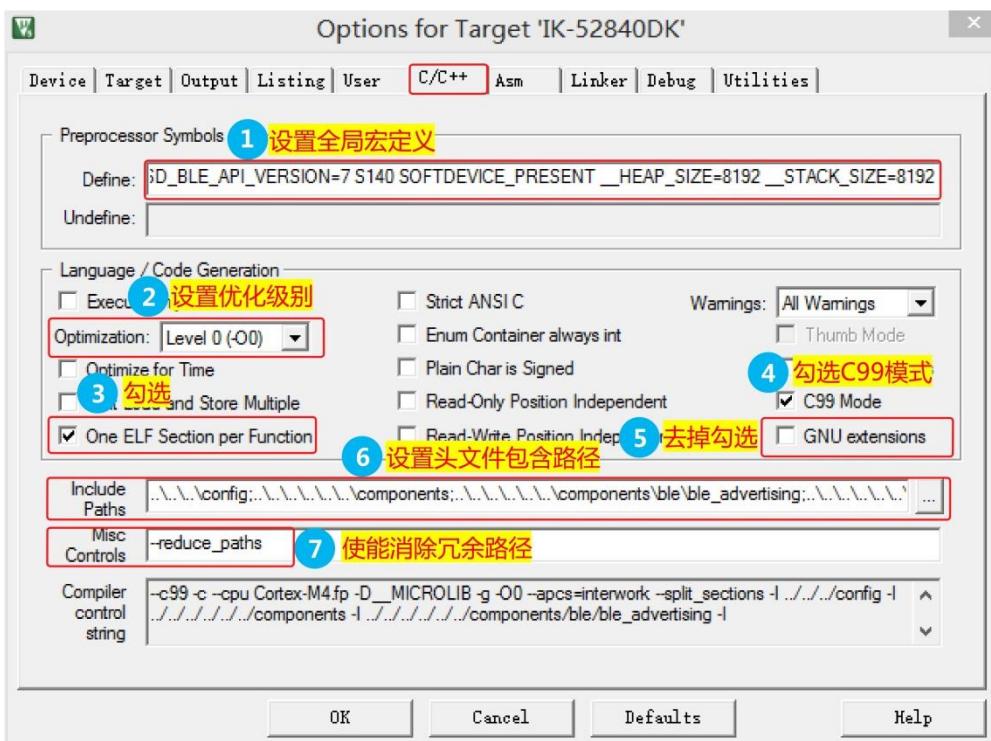


图 1-14：配置“C/C++”选项卡

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：

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 Section per Function”。

One ELF Section 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：

BLE 工程需要包含的头文件路径很多，这里不一一列出，读者可以参考本节对应的例子中的配置。

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。

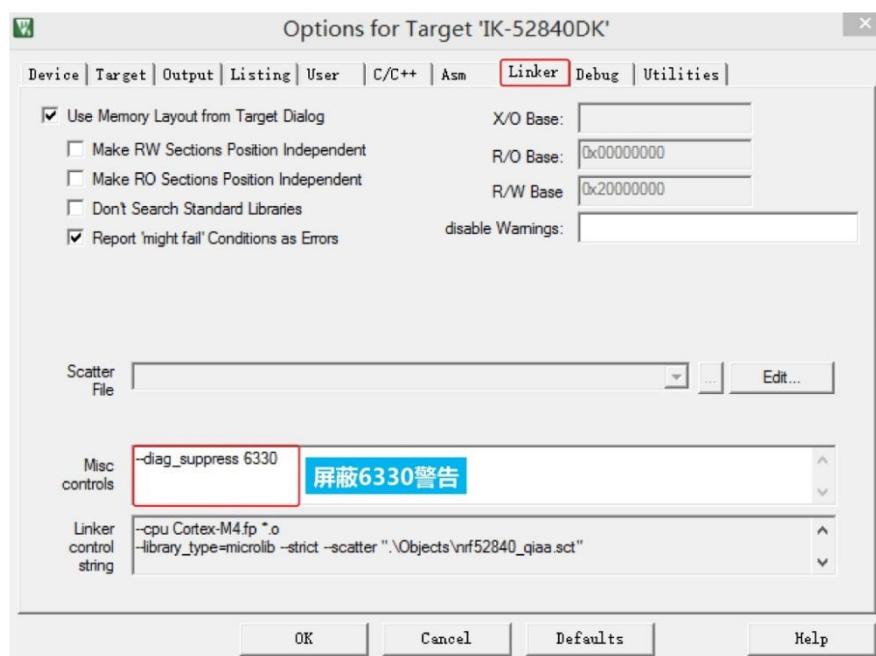


图 1-15: 配置 “Linker” 选项卡

3.5. 配置“Debug”选项卡

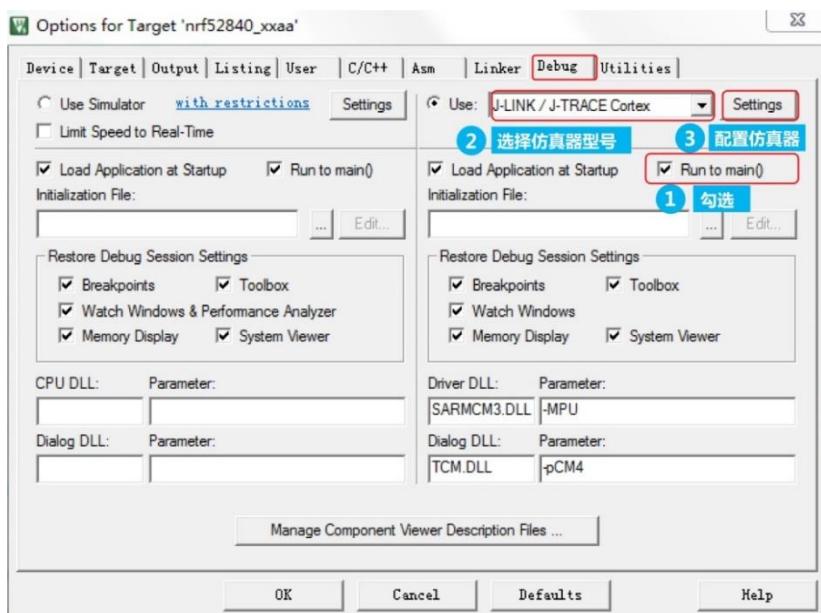


图 1-16: 配置“Debug”选项卡

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。

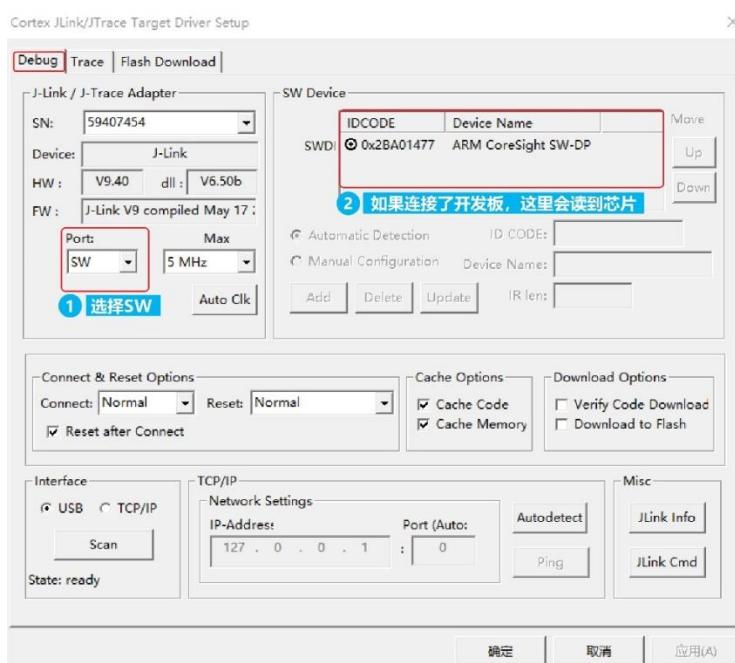


图 1-17: 配置调试接口

切换到“Flash Download”选项卡，加载编程算法。

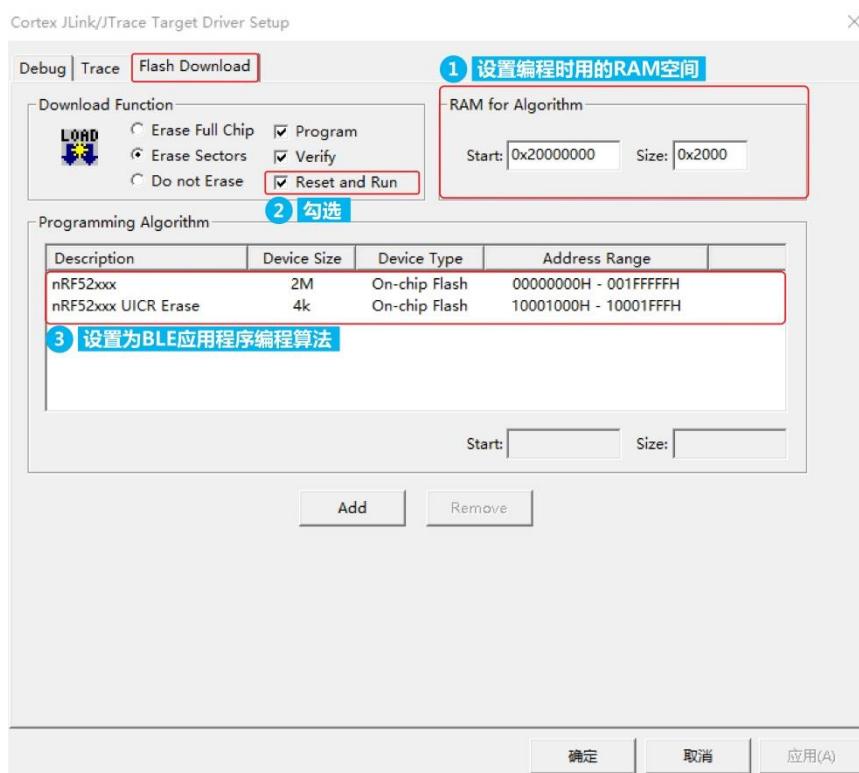


图 1-18：配置下载选项

“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。
- **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。
- **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。

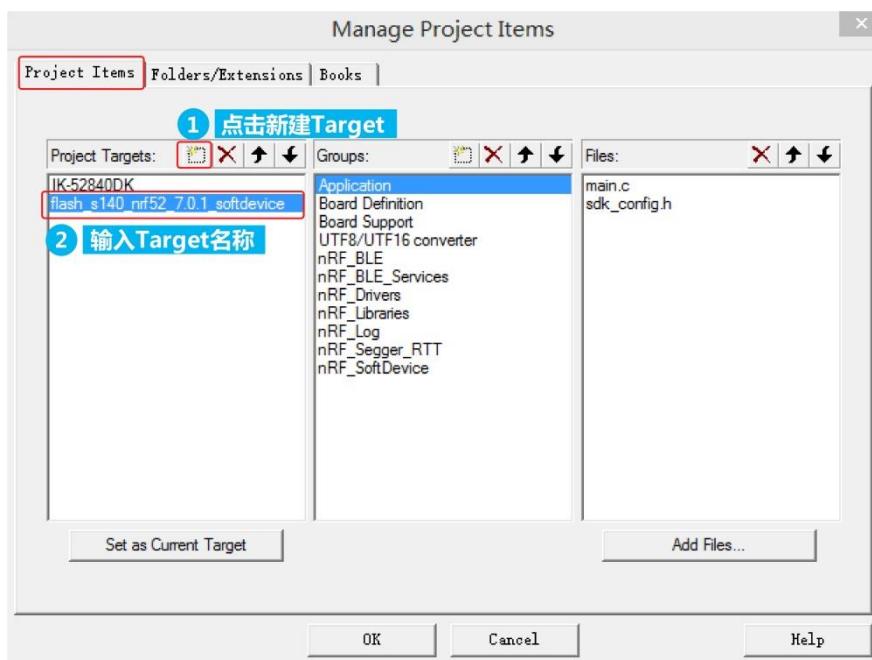


图 1-19：新建下载协议栈的“Target”

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。

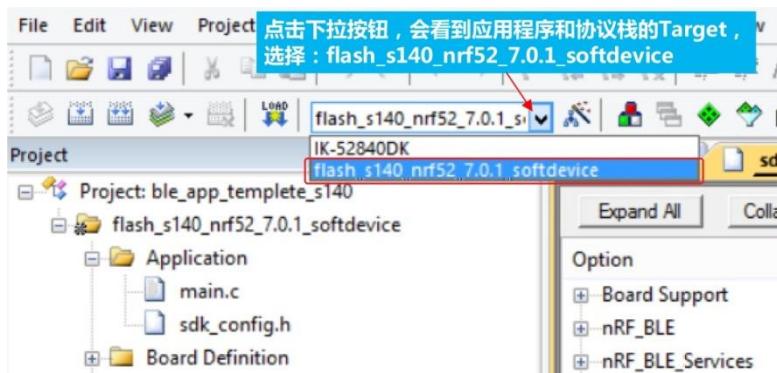


图 1-20：切换到协议栈的“Target”

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。

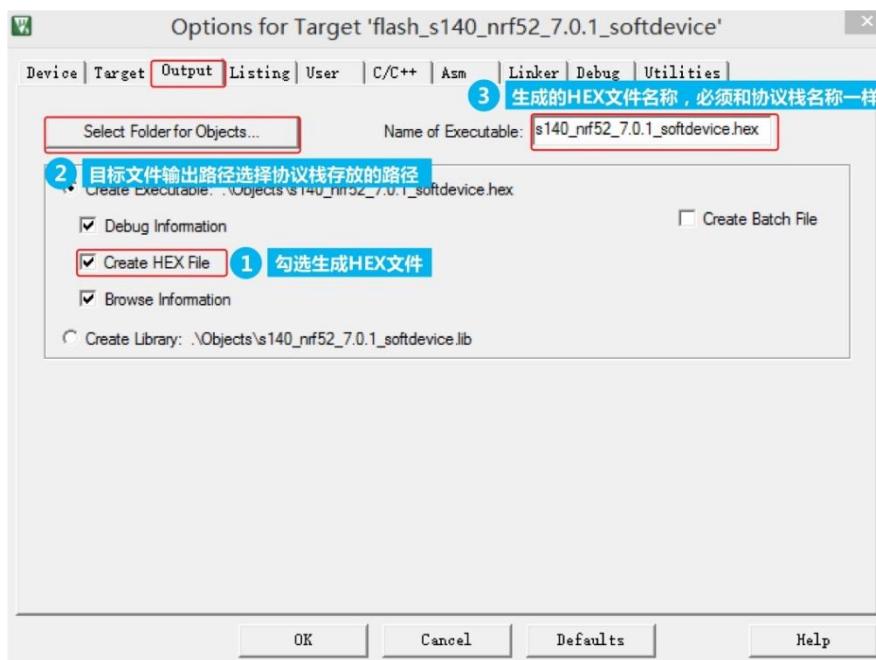


图 1-21：配置“Output”选项卡

配置下面几个项目：

- 勾选生成 HEX 文件的选项。
- 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。
- 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”。

4.3. 配置“Debug”选项卡

协议栈 Target 的“Debug”选项卡配置除了编程算法之外和应用程序的配置一样。协议栈 Target 编程算法按照下图所示配置。

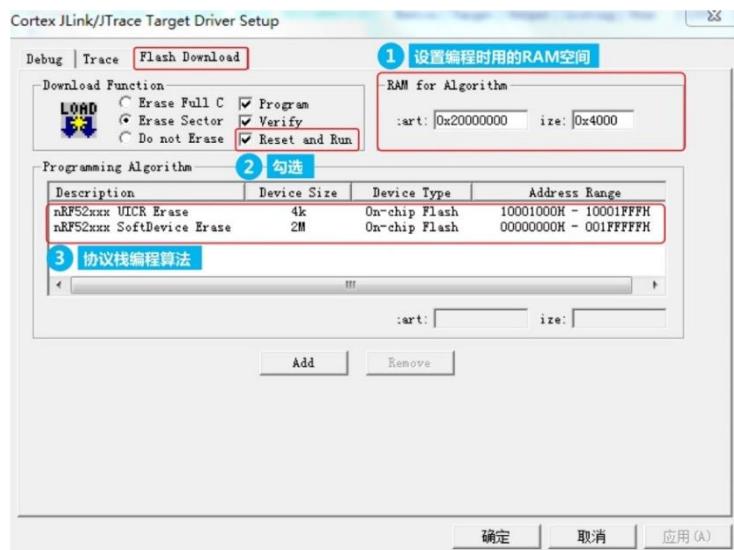


图 1-22：配置下载选项

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。

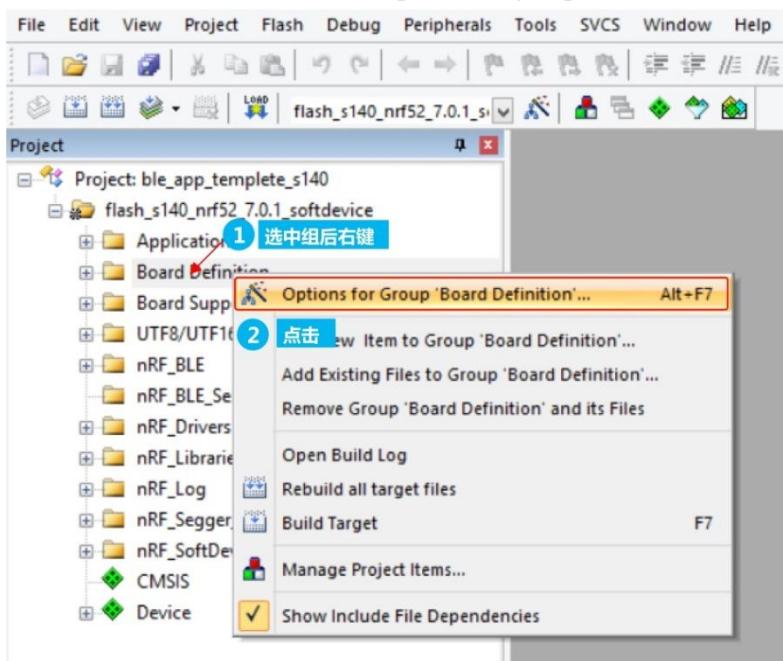


图 1-23: 选中组

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

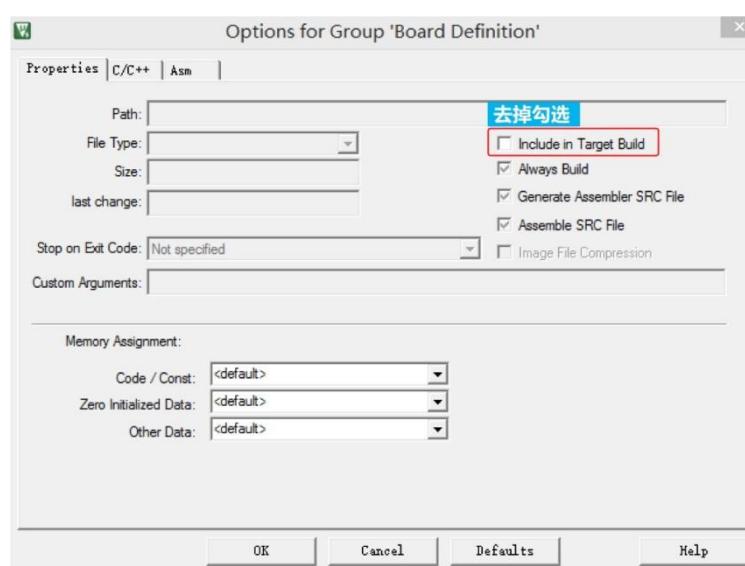


图 1-24: 去掉勾选

- 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

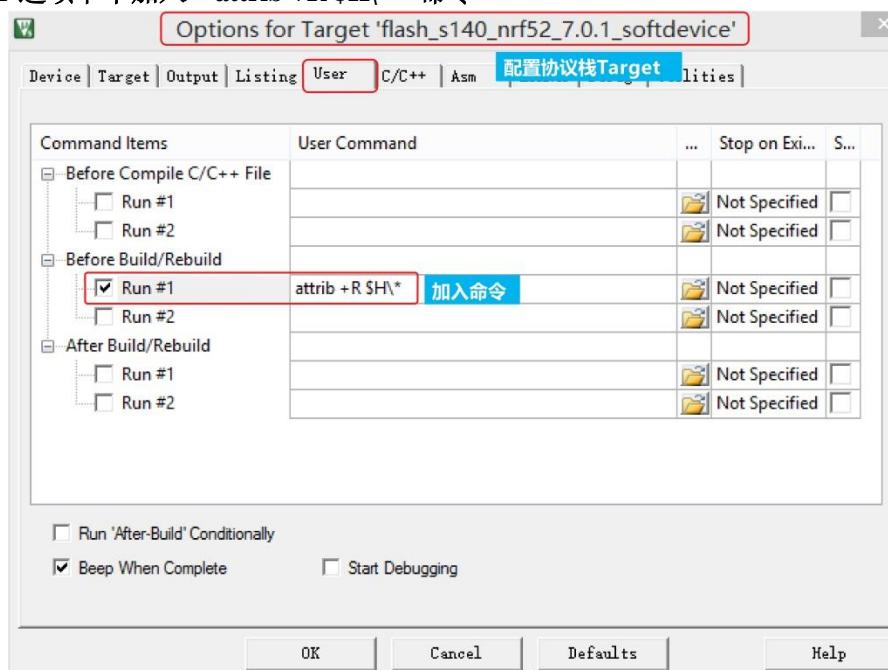


图 1-25：“User” 中加入命令

至此，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→Erase”即可下载协议栈或全片擦除芯片。

第二章：扫描

1. 学习目的

- 掌握主动扫描和被动扫描的区别以及什么情况下使用主动扫描和被动扫描。
- 掌握扫描程序的流程以及如何获取扫描信息。
- 掌握扫描信息的解析，包括设备地址、RSSI、设备名称以及 UUID 等。

2. 扫描的基本概念

从机通过广播向外发布自身信息，告知自己的存在，等待主机来发现和连接自己，主机启动后进入扫描态，通过扫描发现周边的设备(从机)，获取从机的信息，这些信息包含地址、广播数据、RSSI 等，主机可以据此决定是否发起连接。

2.1. 主动扫描和被动扫描

2.1.1. 主动扫描和被动扫描区别

扫描分为主动扫描和被动扫描。理解主动扫描和被动扫描之前，需要先了解什么是扫描请求和扫描响应。

- 扫描请求：由链路层处于扫描态的设备发送，链路层处于广播态的设备接收。
- 扫描响应：由链路层处于广播态的设备发送，链路层处于扫描态的设备接收。

❖ 注意：扫描请求 SCAN_REQ 和扫描响应 SCAN_RSP 都属于广播包，而不属于连接包。

■ 区别

- 被动扫描仅仅接收广播包，不会发送扫描请求。
- 主动扫描接收广播包后会发送扫描请求给处于广播态的设备，并通过处于广播态的设备返回的扫描响应获取额外的数据。由此我们可以看出广播可发送的最大数据长度为 62 个字节（31 个字节广播数据加上 31 个字节扫描响应）。

■ 什么情况下使用被动扫描

如果我们的应用中，能确定从机不需要发送“扫描响应”，即从机的广播数据不超过 31 个字节，只需发送广播包即可，这时候，主机可以使用被动扫描。

■ 什么情况下使用主动扫描

如果我们的应用中，不能确定从机是否会发送“扫描响应”，这时，主机为了具备获取扫描响应数据的能力，应使用主动扫描。

2.1.2. 被动扫描流程

被动扫描的流程如下图所示，从图中可以看到，主机启动被动扫描后，主机只接收了广播包，主机和从机之间没有扫描请求和扫描响应。

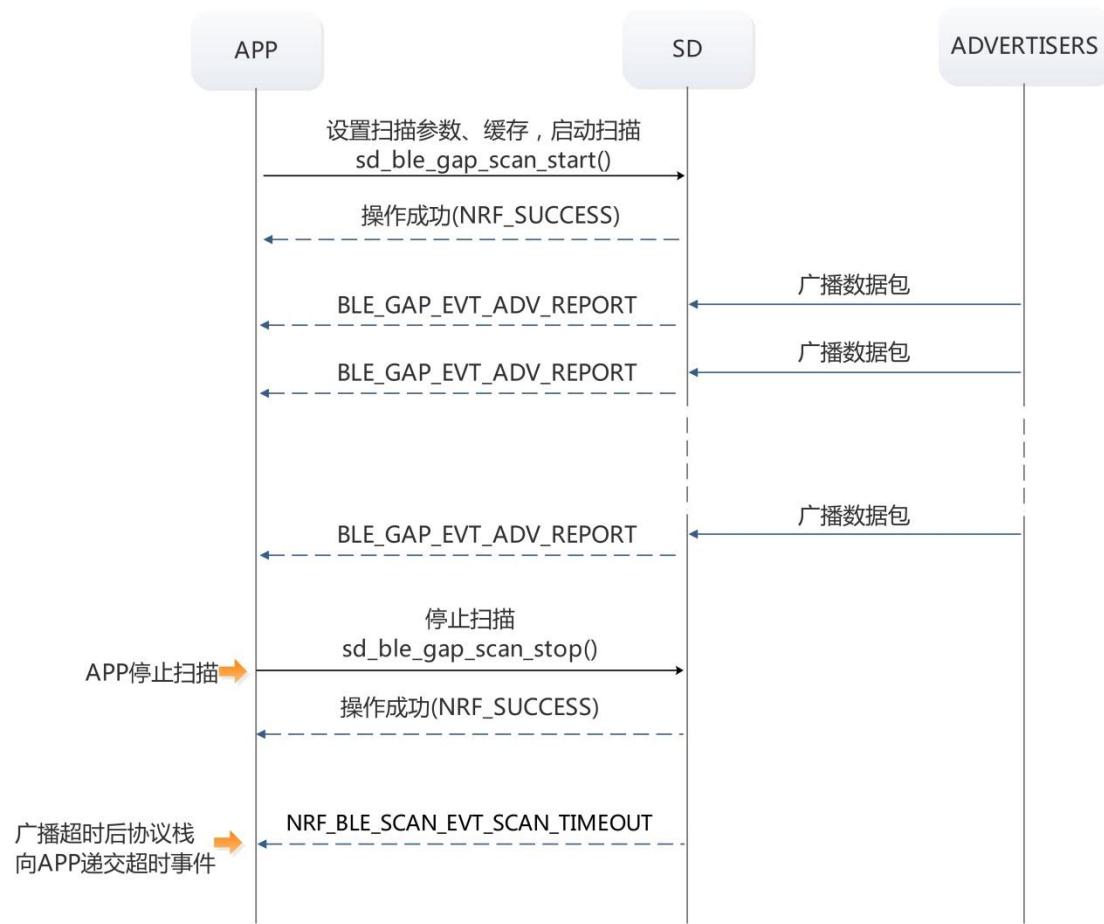


图 2-1：被动扫描流程

2.1.3. 主动扫描流程

主动扫描的流程如下图所示，从图中可以看到，主机启动主动扫描后，主机接收了广播包后，会向从机发送扫描请求，以获取更多的数据，从机接收到扫描请求后，会返回扫描响应。

相对于被动扫描，主动扫描时主机和从机之间多了一次数据的交互(扫描请求和扫描响应)，如下图红字体部分所示。

- ❖ 注意：扫描请求和扫描响应是由协议栈来完成的，不需要应用程序来参与，应用程序只参与了扫描的配置和启动以及扫描到设备后对扫描信息的处理(接收协议栈递交的事件)。

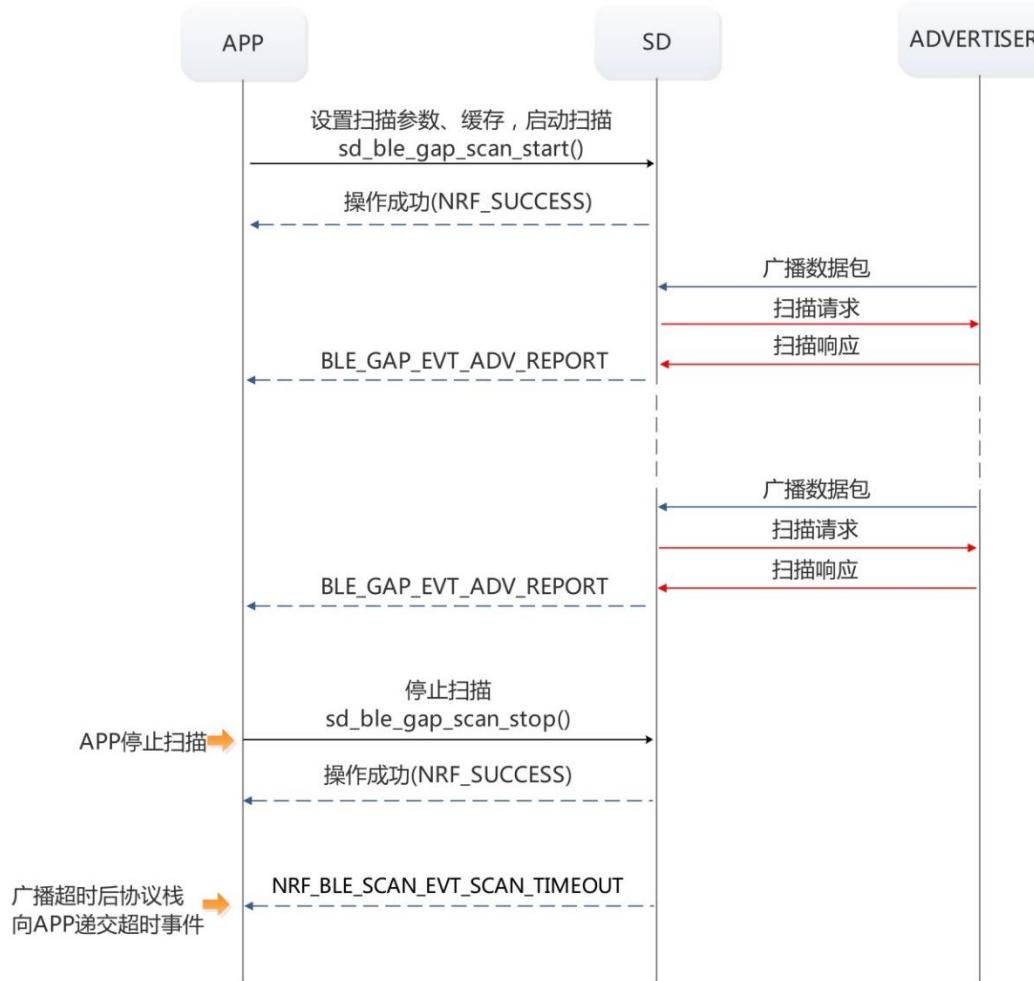
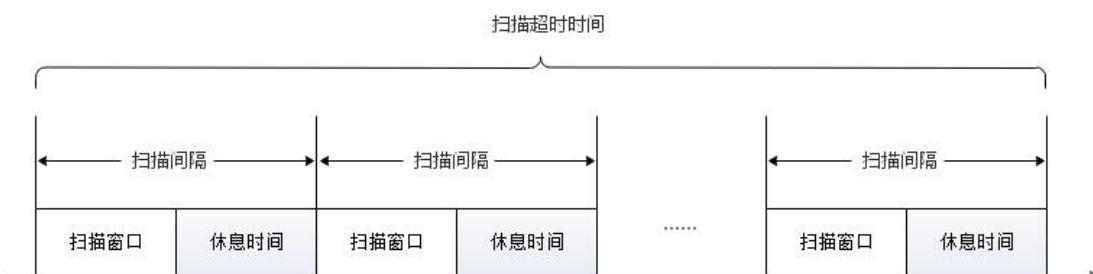


图 2-2：主动扫描流程

2.2. 扫描的参数

扫描中有 3 个重要的时间参数需要注意：

- 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。
- 扫描超时时间(scan timeout): 扫描持续的时间，可以配置为固定的时长或者持续扫描(不超时)。



■ 扫描间隔定义：两个连续的扫描窗口的起始时间的时间差

图 2-3：扫描窗口和扫描间隔

由上图中可以清楚地看到，这3个时间参数决定了扫描的时效和扫描对能量的消耗，扫描窗口密度越大，扫描到设备的速度越快，但是消耗的能量也就越大，反之，扫描到设备的速度变慢，但是能量消耗也会降低。

3. 程序编写

本例的程序主流程图如下图所示，注意流程图里面的“必须项”和“非必须项”。

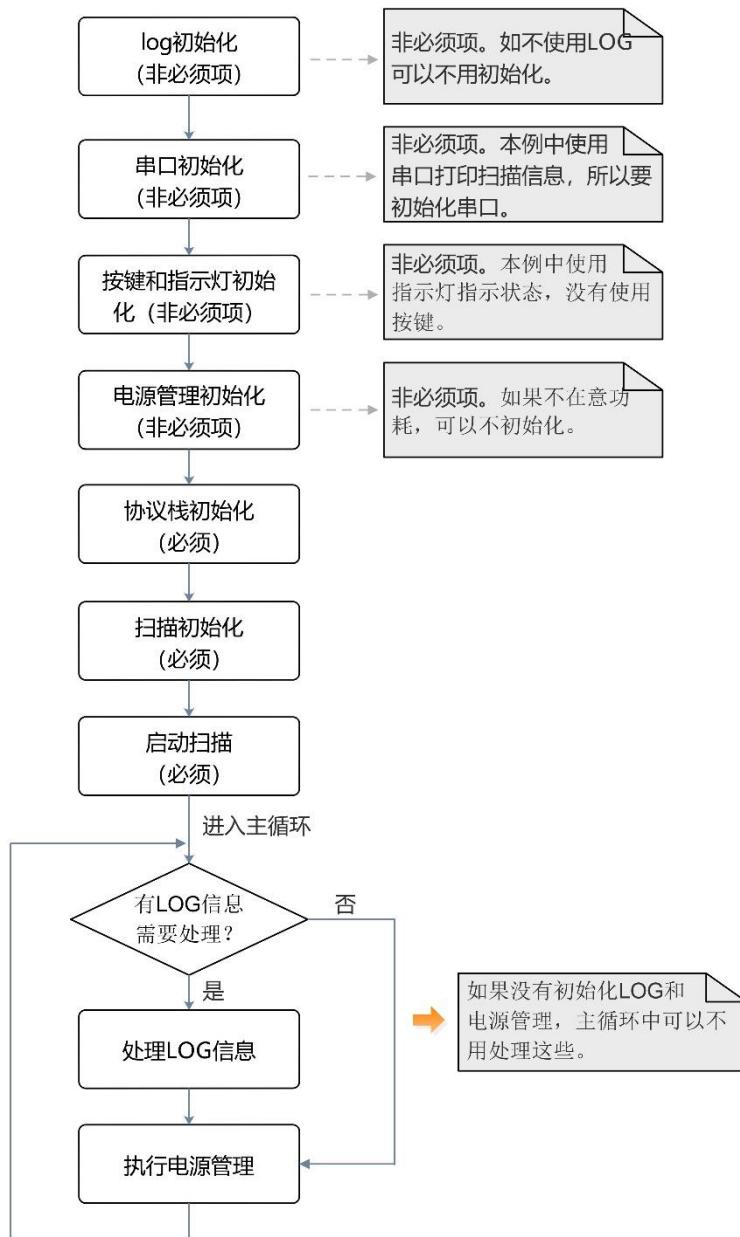


图 2-4：主程序流程

由上图可以看到，程序中LOG、串口、按键和指示灯、电源管理以及主循环中的LOG和电源管理的处理，这些都是由实际的情况决定是否使用的，协议栈初始化以及扫描的初始化和启动扫描是必须要执行的。

本章我们重点讲解的是扫描，下面列举了本例使用这些可选项的原因，对于这些可选的

项目的详细描述，可参考《开发指南下册-从机开发》，这里不再赘述。

- LOG 初始化：虽然不是必须项，但是通常程序中都会使用 LOG，调试的时候可以打开 LOG 打印调试信息，调试完成后，可以在“ `sdk_config.h`”文件中关闭 LOG。注意，本例中使用了串口打印扫描获取的广播数据，因此，使用 LOG 的时候，LOG 的输出终端要配置为 JLINK-RTT，而不要配置为 UART。
- 串口初始化：本例中通过串口打印扫描获取的广播数据，因此需要初始化串口。
- 按键和指示灯初始化：本例使用了指示灯指示 BLE 工作状态，没有用到按键，因此程序中只需初始化指示灯即可。
- 电源管理初始化：对于一个应用来说，在保证性能的情况下，功耗越低越好，因此本例中使用电源管理，让设备在空闲的情况下进入低功耗模式，从而节省能量。
- 主循环中 LOG 和电源管理的处理：使用了 LOG 和电源管理，则需要在主循环中执行处理代码。

3.1. 定义扫描器实例

应用程序中使用宏定义 `NRF_BLE_SCAN_DEF` 实例化一个扫描器，下面的代码实例化一个名称为“`m_scan`”的扫描器。

代码清单： 定义一个名称为 `m_scan` 扫描器实例

```
1. //定义名称为 m_scan 的扫描器实例
2. NRF_BLE_SCAN_DEF(m_scan);
```

`NRF_BLE_SCAN_DEF` 是一个带参数的宏，输入参数“`_name`”即为实例化的扫描器的名称，该宏定义展开后代码如下。

代码清单： `NRF_BLE_SCAN_DEF` 宏

```
1. #define NRF_BLE_SCAN_DEF(_name) \
2.     static nrf_ble_scan_t _name; \
3.     NRF_SDH_BLE_OBSERVER(_name ## _ble_obs, \
4.                           NRF_BLE_SCAN_OBSERVER_PRIO, \
5.                           nrf_ble_scan_on_ble_evt, &_name); \
```

- 1) 行 2：定义了 `static` 类型扫描器结构体变量，为扫描器结构体分配了内存，该语句执行后，扫描器结构体变量常驻内存，即实例化了一个名为“`_name`”的扫描器，协议栈和应用程序均可通过该名称访问扫描器实例。
- 2) 行 3~行 5：注册了名称为“`_name_ble_obs`”的 BLE 扫描事件监视者，注册成功后，当协议栈有事件产生时，会告知 BLE 扫描事件监视者“`_name_obs`”，“`_name_obs`”接收到协议栈的事件后，进入事件处理函数 `nrf_ble_scan_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。

由 `NRF_BLE_SCAN_DEF` 宏的代码可以看到，扫描器实例化时主要的工作是：为扫描器分配了内存和注册 BLE 扫描事件监视者。分配的内存即是 `nrf_ble_scan_t` 结构体占用的内

存，该结构体包含了扫描参数以及存储扫描数据的缓存，扫描器每扫描到一个从机设备时，获取的设备的广播数据会保存到 `scan_buffer_data` 数组，应用程序获取的广播数据也是从该数组中读取的，这就是实际存储扫描数据的地方，`nrf_ble_scan_t` 结构体声明如下。

代码清单：扫描器结构体 `nrf_ble_scan_t`

```

1. typedef struct
2. {
3.     #if (NRF_BLE_SCAN_FILTER_ENABLE == 1)
4.         nrf_ble_scan_filters_t scan_filters;           //过滤器数据
5.     #endif
6.     //如果设置为 true，则在过滤器匹配或者从白名单识别到设备后，自动连接设备
7.     bool                      connect_if_match;
8.     //连接参数
9.     ble_gap_conn_params_t      conn_params;
10.    //该变量用来跟踪当过滤器匹配或白名单匹配触发连接时使用的连接设置
11.    uint8_t                   conn_cfg_tag;
12.    //扫描参数
13.    ble_gap_scan_params_t     scan_params;
14.    //扫描事件句柄，如果不需要再应用程序中处理扫描事件，可以设置为 NULL
15.    nrf_ble_scan_evt_handler_t evt_handler;
16.    //SoftDevice 存储广播报告的缓存区
17.    uint8_t                   scan_buffer_data[NRF_BLE_SCAN_BUFFER];
18.    //存储指向 SoftDevice 存储广播报告的缓存区的指针
19.    ble_data_t                scan_buffer;
20. } nrf_ble_scan_t;

```

BLE 扫描事件监视者是扫描器从协议栈获取事件的通道，协议栈将扫描相关的事件通过扫描器的事件处理函数 `nrf_ble_scan_on_ble_evt()` 提交给扫描器，扫描器在该事件函数中即可进行相应处理。

本例中，我们的目的是获取扫描数据，因此我们只需关心广播报告事件（`BLE_GAP_EVT_ADV_REPORT`），该事件表示协议栈扫描到了一个从机设备，并将扫描数据储存到了扫描器的 `scan_buffer_data` 数组。

另外，扫描器接收到广播报告事件后，会根据过滤器等相关配置来处理此次事件，之后会根据应用程序在调用 `nrf_ble_scan_init()` 函数初始化扫描时是否注册了事件句柄来决定是否将该事件通知应用程序，因此，应用程序如果需要扫描器事件，则在初始化扫描时需要注册事件句柄。

3.2. 扫描初始化

扫描初始化部分要做的工作主要包括 3 个方面：配置扫描参数；编写事件回调函数（如应用程序需要关注扫描器事件）；调用 `nrf_ble_scan_init()` 函数初始化扫描。

初始化的流程如下图所示，首先定义扫描初始化结构体变量，之后对其进行配置，扫描初始化结构体变量中包含了扫描参数，这里我们可以选择是否配置该结构体中的扫描参数，

如果配置了，那么调用 `nrf_ble_scan_init()` 函数初始化扫描时会使用该结构体中的扫描参数配置扫描器，如果初始化结构体中没有配置扫描参数，`nrf_ble_scan_init()` 函数中会使用“ `sdk_config.h`”文件中的扫描参数配置扫描器，这时，如果我们需要修改扫描参数，扫描参数在“ `sdk_config.h`”文件中修改即可。

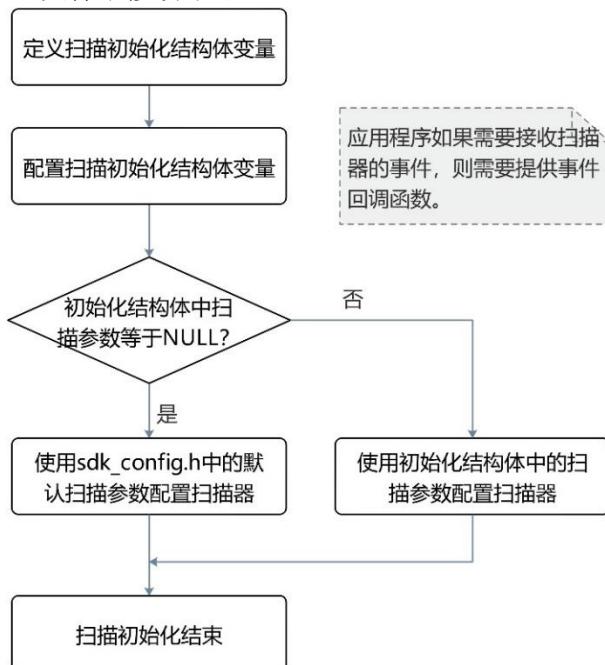


图 2-5：扫描初始化流程

3.2.1. 定义扫描初始化结构体变量

扫描初始化结构体包含了扫描初始化所需要的必要参数，该结构体声明如下，对于本例来说，我们只实现扫描，因此 `connect_if_match` 需要设置为 `false`，也就是扫描到设备后不主动发起连接，并且连接参数不用设置。至于扫描参数“`p_scan_param`”，如果这里设置了，扫描初始化函数 `nrf_ble_scan_init()` 中会使用设置的扫描参数配置扫描器，否则使用“ `sdk_config.h`”文件中的扫描参数配置扫描器。

代码清单：扫描初始化结构体

```

1. typedef struct
2. {
3.     //扫描参数：可以初始化为 NULL，如果为 NULL，则从静态配置(sdk_config.h)加载扫描参数
4.     ble_gap_scan_params_t const * p_scan_param;
5.     //如果设置为 true，则模块会在过滤器匹配或从白名单中成功识别设备后自动连接
6.     bool connect_if_match;
7.     //连接参数：可以初始化为 NULL，如果为 NULL，则从静态配置加载连接参数
8.     ble_gap_conn_params_t const * p_conn_param;
9.     //该变量用于跟踪当过滤器或白名单匹配后，连接所使用的连接设置
10.    uint8_t conn_cfg_tag;
11. } nrf_ble_scan_init_t;

```

扫描初始化结构体中的 `ble_gap_scan_params_t` 结构体指针 `p_scan_param` 用来指向扫描

参数（可以设置为 NULL，即使用 sdk_config.h 文件中的扫描参数配置扫描器）。

`ble_gap_scan_params_t` 是扫描参数结构体，是扫描相关的非常重要的一个结构体，其声明如下。他包含了扫描窗口、扫描间隔、过滤策略以及对扩展广播包扫描的支持等信息。

其中扩展广播包扫描是 5.0 BLE 的一个特性，这部分的内容会在后续章节中单独讲解，这里只需要知道他是 5.0BLE 的特性就可以了。

代码清单：扫描参数结构体

```

1. typedef struct
2. {
3.     //=1: 扫描器接收扩展广播包
4.     //=0: 扫描器不接收次级信道上的广播包，也无法接收长广播 PDU
5.     uint8_t      extended          : 1;
6.     //仅用于扩展广播包，当前 SoftDevice 不支持该特性
7.     uint8_t      report_incomplete_evt : 1;
8.     //=1: 执行主动扫描，发送扫描请求，否则执行被动扫描。和 sd_ble_gap_connect()一起使用时，  

9.     //该参数忽略
10.    uint8_t      active           : 1;
11.    //扫描过滤策略
12.    uint8_t      filter_policy     : 2;
13.    //PHY 设置，如果设置成 BLE_GAP_PHY_AUTO，扫描的 PHY 会默认设置为 BLE_GAP_PHY_1MBPS
14.    uint8_t      scan_phys;
15.    //扫描间隔: 单位 625us
16.    uint16_t     interval;
17.    //扫描窗口: 单位 625us，如果扫描器包含 BLE_GAP_PHY_1MBPS 和 BLE_GAP_PHY_CODED，扫描间
18.    //隔应大于等于 2 个扫描窗口
19.    uint16_t     window;
20.    //扫描超时时间，单位: 10ms
21.    uint16_t     timeout;
22.    //首要和次级广播信道掩码，扫描器至少要使用一个首要广播信道 (37 38 39 信道)
23.    ble_gap_ch_mask_t   channel_mask;
24. } ble_gap_scan_params_t;
```

下面的代码定义了本例中使用的名为 `init_scan` 的扫描初始化结构体变量，定义后先使用 `memset()` 函数清零，之后再进行相关配置。

代码清单：定义名为 init_scan 的扫描初始化结构体变量

```

1. //定义扫描初始化结构体变量
2. nrf_ble_scan_init_t init_scan;
3. //先清零，再配置
4. memset(&init_scan, 0, sizeof(init_scan));
```

3.2.2. 扫描参数的配置

正如前文所述，扫描参数可以使用初始化结构体来配置，也可以将初始化结构体中的扫描参数结构体指针设置为“NULL”，从而使用 sdk_config.h 文件中的配置，这两种方式效果一样，但是在操作上有些差别。

■ 不使用初始化结构体中的扫描参数

将 p_scan_param 指向“NULL”，扫描初始化函数中会使用“sdk_config.h”文件中的配置，如下图所示，如果我们需要修改扫描参数，在这里修改即可。

因为定义扫描初始化结构体变量时对该变量进行了清零操作，因此，不配置 p_scan_param 等于 p_scan_param 指向 NULL。

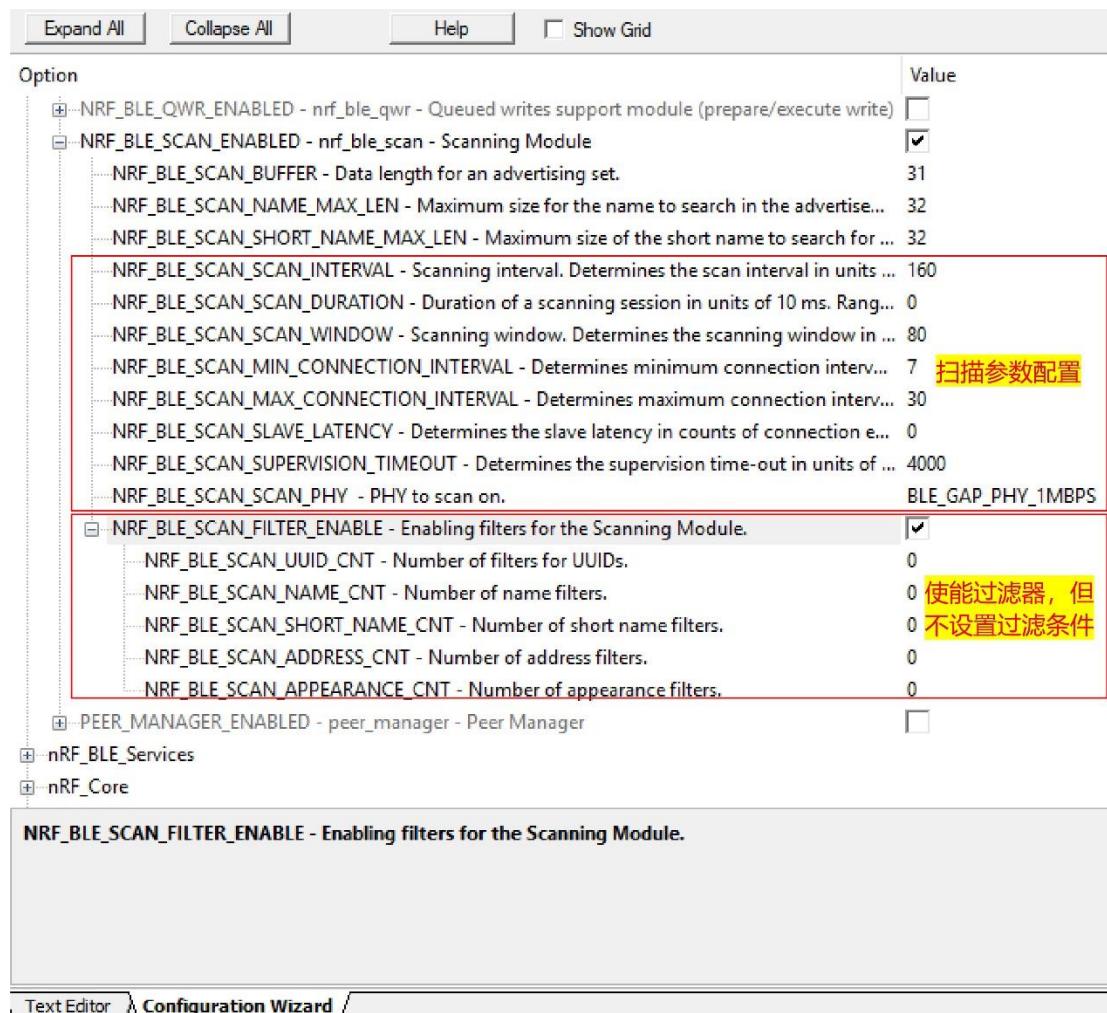


图 2-6：配置“sdk_config.h”文件

■ 使用初始化结构体中的扫描参数

p_scan_param 是结构体指针变量，因此我们需要先定义一个 ble_gap_scan_params_t 结构体变量并初始化相关扫描参数，之后将该变量的地址赋值给“p_scan_param”。本例中，我们定义了“m_scan_param”变量并根据需求表中的扫描参数对其进行初始化，代码如下。

代码清单：定义扫描参数结构体变量 m_scan_param 并初始化

```
1. static ble_gap_scan_params_t const m_scan_param =
```

```

2. {
3.     .active          = 0x01, // 使用主动扫描
4.     // 扫描间隔设置为: 160*0.625ms = 100ms
5.     .interval        = NRF_BLE_SCAN_SCAN_INTERVAL,
6.     // 扫描窗口设置为: 80*0.625ms = 50ms
7.     .window          = NRF_BLE_SCAN_SCAN_WINDOW,
8.     // 接收除了不是发给本机的定向广播之外的所有广播数据
9.     .filter_policy   = BLE_GAP_SCAN_FP_ACCEPT_ALL,
10.    .timeout         = 0, // 扫描超时时间设置为 0, 即扫描不超时
11.    .scan_phys       = BLE_GAP_PHY_1MBPS, // 1 Mbps PHY
12. };

```

3.2.3. 事件处理

从扫描流程图中可以看到，主机扫描到一个从机设备后，协议栈会提交“BLE_GAP_EVT_ADV_REPORT”事件，该事件会被扫描器实例的事件处理函数 nrf_ble_scan_on_ble_evt()接收，即扫描器程序模块接收了该事件。

扫描器程序模块接收了“BLE_GAP_EVT_ADV_REPORT”事件后，会根据过滤器/白名单的匹配结果向应用程序提交下列事件。

- **NRF_BLE_SCAN_EVT_FILTER_MATCH:** 匹配过滤器或在多过滤器模式下匹配所有过滤器。
- **NRF_BLE_SCAN_EVT_WHITELIST_ADV_REPORT:** 当发现白名单中的设备时，通知主应用程序。
- **NRF_BLE_SCAN_EVT_NOT_FOUND:** 扫描数据不匹配过滤器。

本例中，不过滤扫描数据，即不设置过滤条件，因此每扫描到一个设备，都没有相应的过滤条件与之对应，这样对过滤器来说处理结果都是“不匹配的”，所以，提交给应用程序的事件是：NRF_BLE_SCAN_EVT_NOT_FOUND。应用程序如果需要关注该事件，在调用 nrf_ble_scan_init() 执行扫描初始化得时候注册事件回调函数，并在事件函数里面处理该事件即可，代码如下。

代码清单：应用程序扫描事件处理函数

```

1. static void scan_evt_handler(scan_evt_t const * p_scan_evt)
2. {
3.     ret_code_t err_code;
4.     // 翻转 LED 指示灯 D4 的状态，指示扫描到设备
5.     nrf_gpio_pin_toggle(LED_4);
6.     // 判断事件类型
7.     switch(p_scan_evt->scan_evt_id)
8.     {
9.         // 因为不过滤扫描信息，即不设置过滤条件，所以只需关注 NRF_BLE_SCAN_EVT_NOT_FOUND
10.        // 事件

```

```

11.     case NRF_BLE_SCAN_EVT_NOT_FOUND:
12.     {
13.         //这里可以获取扫描数据
14.     } break;
15.     //扫描超时事件
16.     case NRF_BLE_SCAN_EVT_SCAN_TIMEOUT:
17.     {
18.         NRF_LOG_INFO("Scan timed out.");
19.         //重启扫描
20.         scan_start();
21.     } break;
22.
23.     default:
24.     break;
25. }
26. }
```

3.2.4. 初始化

扫描初始化的库函数是 nrf_ble_scan_init(), 函数原型如下:

表 2-1: nrf_ble_scan_init()函数

函数原型	ret_code_t nrf_ble_scan_init (nrf_ble_scan_t *const p_scan_ctx, nrf_ble_scan_init_t const *const p_init, nrf_ble_scan_evt_handler_t evt_handler)
函数功能	初始化扫描模块。
参 数	[out] p_scan_ctx: 指向扫描模块实例的指针, 该结构必须由应用程序提供, 它由此函数初始化, 之后用于标识此特定模块实例。 [in] p_init: 可以初始化为 NULL, 如果为 NULL, 则从静态配置加载初始化模块所需的参数。如果模块要自动建立连接, 则必须使用相关数据对其进行初始化。 [in] evt_handler: 扫描事件句柄, 如果主应用程序中没有实现任何处理, 则可以初始化为 NULL。
返回值	NRF_SUCCESS: 初始化成功。 NRF_ERROR_NULL: p_scan_ctx 指向了 NULL。

观察 nrf_ble_scan_init()函数的参数, 可以看到前文中我们定义的扫描器实例、扫描初始化结构体和扫描事件处理函数也是调用 nrf_ble_scan_init()函数初始化扫描时需要用到的参

数，从时效上看，他们的区别如下。

- 扫描器实例：实例化后，在程序的整个生命周期中存在。
- 扫描初始化结构体：仅在扫描初始化函数中有效，初始化过程中会将扫描初始化结构体中的参数拷贝到扫描器实例。
- 扫描事件处理函数：如果应用程序提供了事件处理函数，事件处理函数在程序的整个生命周期中有效。

本例中，我们使用初始化结构体中的扫描参数初始化扫描，因此应用程序需要定义一个扫描参数结构体变量并初始化需要使用到的参数变量，代码清单如下。

代码清单：扫描参数

```

1. static ble_gap_scan_params_t const m_scan_param =
2. {
3.     .active      = 0x01, // 使用主动扫描
4.     .interval    = NRF_BLE_SCAN_SCAN_INTERVAL, // 扫描间隔设置为: 160*0.625ms = 100ms
5.     .window      = NRF_BLE_SCAN_SCAN_WINDOW, // 扫描窗口设置为: 80*0.625ms = 50ms
6.     // 接收除了不是发给本机的定向广播之外的所有广播数据
7.     .filter_policy = BLE_GAP_SCAN_FP_ACCEPT_ALL,
8.     .timeout      = 0, // 扫描超时时间设置为 0, 即扫描不超时
9.     .scan_phys    = BLE_GAP_PHY_1MBPS, // 1 Mbps PHY
10. };

```

扫描初始化函数代码如下，因为应用程序需要获取扫描数据，因此，初始化时应用程序提供了事件处理函数 scan_evt_handler，并将其作为函数 nrf_ble_scan_init()的参数传递给该函数。

代码清单：扫描初始化

```

1. static void scan_init(void)
2. {
3.     ret_code_t         err_code;
4.     // 定义扫描初始化结构体变量
5.     nrf_ble_scan_init_t init_scan;
6.     // 先清零，再配置
7.     memset(&init_scan, 0, sizeof(init_scan));
8.     // 自动连接设置为 false
9.     init_scan.connect_if_match = false;
10.    // 使用初始化结构体中的扫描参数配置扫描器，这里 p_scan_param 指向定义的扫描参数
11.    init_scan.p_scan_param      = &m_scan_param;
12.    // conn_cfg_tag 设置为 1
13.    init_scan.conn_cfg_tag     = APP_BLE_CONN_CFG_TAG;
14.    // 初始化扫描器
15.    err_code = nrf_ble_scan_init(&m_scan, &init_scan, scan_evt_handler);
16.    APP_ERROR_CHECK(err_code);

```

17. }

3.3. 启动和停止扫描

扫描初始化完成后并不会自行启动，需要应用程序去启动扫描，启动扫描的函数是 `nrf_ble_scan_start()`，其原型如下表所示。

表 2-2: `nrf_ble_scan_start()` 函数

格式说明符	描述	示例
函数原型	<code>ret_code_t nrf_ble_scan_start((n nrf_ble_scan_t const *const p_scan_ctx,)</code>	
函数功能	根据扫描初始化时配置的参数启动扫描。	
参数	[in] <code>p_scan_ctx</code> : 指向扫描器实例。	
返回值	<code>NRF_SUCCESS</code> : 扫描启动成功，否则返回错误代码。 <code>NRF_ERROR_NULL</code> : 输入参数 <code>p_scan_ctx</code> 指向 NULL。	

启动扫描器后，设置指示灯用来指示当前正在执行扫描，代码如下。

代码清单：启动扫描

```

1. static void scan_start(void)
2. {
3.     ret_code_t ret;
4.     //启动扫描器 m_scan
5.     ret = nrf_ble_scan_start(&m_scan);
6.     APP_ERROR_CHECK(ret);
7.     //设置指示灯状态为：正在执行扫描（D1 闪烁）
8.     ret = bsp_indication_set(BSP_INDICATE_SCANNING);
9.     APP_ERROR_CHECK(ret);
10. }
```

扫描的停止有下面的三种情况：

- 应用程序主动停止：应用程序调用 `nrf_ble_scan_stop()` 函数停止扫描，该函数没有参数和返回值，应用程序直接调用即可。
- 扫描超时后停止：扫描持续的时间达到设置的超时时间后，扫描会停止并产生扫描超时事件“`NRF_BLE_SCAN_EVT_SCAN_TIMEOUT`”，应用程序通过该事件可得知扫描已经停止。
- 只有一条链路的情况下，主机发起连接时会在发起连接的函数中停止扫描。

3.4. 扫描数据的处理

扫描的目的是为了获取设备的信息，即从机的广播数据以及 RSSI。本例中，主机扫描到从机设备后，协议栈向扫描器提交了 `BLE_GAP_EVT_ADV_REPORT` 事件，扫描器处理

后向应用程序提交 NRF_BLE_SCAN_EVT_NOT_FOUND 事件，应用程序由此知道已经扫描到一个广播设备，那么，应用程序如何获取扫描数据的？

再看扫描初始化时应用程序提供的扫描事件处理函数 scan_evt_handler()，该函数的参数为“scan_evt_t const * p_scan_evt”，scan_evt_t 是扫描事件结构体，其声明如下，他不仅包含了事件类型，同时也包含了该事件类型对应的参数。如本例中的事件类型是 NRF_BLE_SCAN_EVT_NOT_FOUND，他对应的事件参数 p_not_found。

代码清单：扫描事件结构体 scan_evt_t

```

1. typedef struct
2. {
3.     nrf_ble_scan_evt_t scan_evt_id; //事件类型
4.     union
5.     {
6.         nrf_ble_scan_evt_filter_match_t filter_match; //过滤器匹配
7.         ble_gap_evt_scan_req_report_t req_report; //扫描请求报告参数
8.         ble_gap_evt_timeout_t timeout; //超时事件参数
9.         //广告报告白名单的事件参数
10.        ble_gap_evt_adv_report_t const * p_whitelist_adv_report;
11.        ble_gap_evt_adv_report_t const * p_not_found; //过滤器不匹配事件参数
12.        nrf_ble_scan_evt_connected_t connected; //连接事件参数
13.        nrf_ble_scan_evt_connecting_err_t connecting_err; //连接时错误事件
14.    } params;
15.    ble_gap_scan_params_t const * p_scan_params; //扫描参数
16. } scan_evt_t;
```

scan_evt_t 中的 params 是 union 类型，因此每个事件产生时只有该事件的参数有效。ble_gap_evt_adv_report_t 的声明如下，可以看到他包含了所有的扫描信息，应用程序通过该结构体即可获取扫描数据。

代码清单：广播报告结构体 ble_gap_evt_adv_report_t

```

1. typedef struct
2. {
3.     //广播报告类型
4.     ble_gap_adv_report_type_t type;
5.     //对端设备地址，习惯上称为 MAC 地址
6.     ble_gap_addr_t peer_addr;
7.     //定向广播设备地址
8.     ble_gap_addr_t direct_addr;
9.     //指示接收主 primary 广播包的 PHY
10.    uint8_t primary_phy;
11.    //指示接收 secondary 广播包的 PHY
12.    uint8_t secondary_phy;
13.    //发射功率等级
```

```

14. int8_t          tx_power;
15. //RSSI, 单位 dBm
16. int8_t          rssi;
17. //接收到最后一个广播包的信道索引(0-39)
18. uint8_t         ch_index;
19. uint8_t         set_id;
20. uint16_t        data_id:12;
21. //广播或扫描响应数据
22. ble_data_t      data;
23. ble_gap_aux_pointer_t aux_pointer;
24. } ble_gap_evt_adv_report_t;

```

3.4.1. 获取从机设备地址和 RSSI 实验

◆ 注：本节对应源码是“实验 2-1：扫描获取从机设备地址和 RSSI”。

应用程序接收到 NRF_BLE_SCAN_EVT_NOT_FOUND 事件后，该事件对应的参数“p_not_found”有效，因此我们可以从“p_not_found”中获取从机设备地址和 RSSI。具体流程为：事件处理函数 scan_evt_handler()中的 NRF_BLE_SCAN_EVT_NOT_FOUND 事件处理代码中定义一个 ble_gap_evt_adv_report_t 结构体指针并指向“p_not_found”，之后通过该指针读取数据并打印出数据。

代码清单：获取从机设备地址和 RSSI

```

1. static void scan_evt_handler(scan_evt_t const * p_scan_evt)
2. {
3.     ret_code_t err_code;
4.     //翻转 LED 指示灯 D4 的状态, 指示扫描到设备
5.     nrf_gpio_pin_toggle(LED_4);
6.     //判断事件类型
7.     switch(p_scan_evt->scan_evt_id)
8.     {
9.         //因为不过滤扫描信息, 即不设置过滤条件, 所以只需关注 NRF_BLE_SCAN_EVT_NOT_FOUND
10.        //事件
11.        case NRF_BLE_SCAN_EVT_NOT_FOUND:
12.        {
13.            //定义广播报告结构体指针 p_adv 并指向扫描事件结构体中的 p_not_found
14.            ble_gap_evt_adv_report_t const * p_adv =
15.                p_scan_evt->params.p_not_found;
16.            //串口打印 MAC 地址
17.            printf("MAC ADDRESS: %02X%02X%02X%02X%02X%02X",
18.                   p_adv->peer_addr.addr[0],
19.                   p_adv->peer_addr.addr[1],
20.                   p_adv->peer_addr.addr[2],
21.                   p_adv->peer_addr.addr[3],

```

```
22.             p_adv->peer_addr.addr[4],  
23.             p_adv->peer_addr.addr[5]  
24.         );  
25.         //打印 RSSI  
26.         printf("RSSI: %d\r\n", p_adv->rssi);  
27.     } break;  
28.     //扫描超时事件  
29.     case NRF_BLE_SCAN_EVT_SCAN_TIMEOUT:  
30.     {  
31.         NRF_LOG_INFO("Scan timed out.");  
32.         //重启扫描  
33.         scan_start();  
34.     } break;  
35.     default:  
36.         break;  
37.     }  
38. }
```

■ 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, 并使用 USB 数据线连接开发板和计算机。

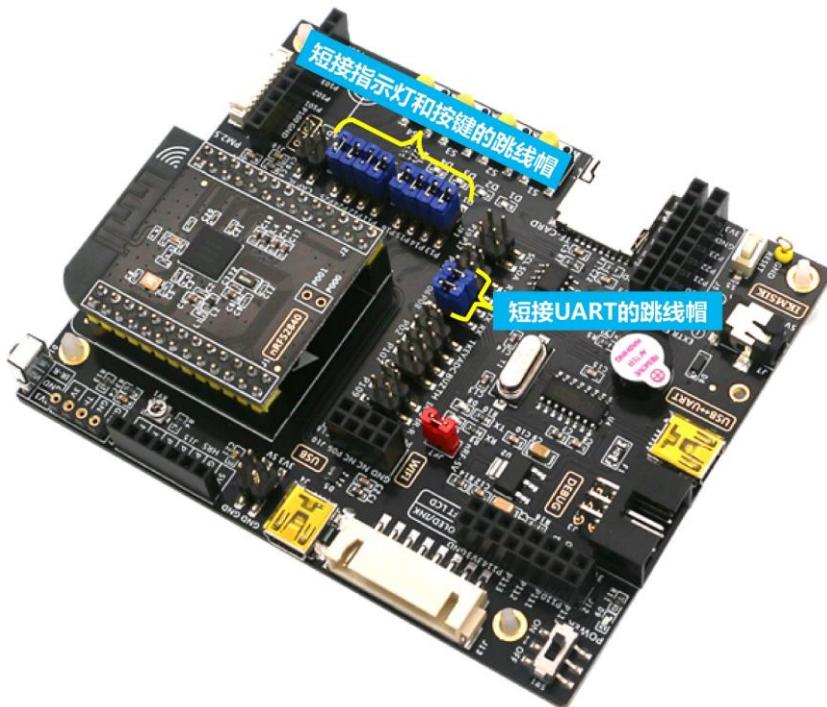


图 2-7：开发板跳线帽短接

■ 测试步骤

1. 解压“…\5: 开发指南（下册-主机）配套实验源码\”目录下的压缩文件“实验 2-1: 扫描获取从机设备地址和 RSSI”，将解压后得到的文件夹“ble_app_scan”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_scan\project\mdk5”目录下的工程“ble_app_scan.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，计算机上的串口调试助手会显示扫描到的广播设备的设备地址和 RSSI，如下图所示。

MAC ADDR: B5C499B055FE	RSSI: -43
MAC ADDR: B5C499B055FE	RSSI: -42
MAC ADDR: B5C499B055FE	RSSI: -58
MAC ADDR: B5C499B055FE	RSSI: -49
MAC ADDR: B5C499B055FE	RSSI: -59
MAC ADDR: B5C499B055FE	RSSI: -49
MAC ADDR: B5C499B055FE	RSSI: -41
MAC ADDR: B5C499B055FE	RSSI: -40
MAC ADDR: B5C499B055FE	RSSI: -44
MAC ADDR: B5C499B055FE	RSSI: -42
MAC ADDR: B5C499B055FE	RSSI: -50
MAC ADDR: B5C499B055FE	RSSI: -49
MAC ADDR: B5C499B055FE	RSSI: -41
MAC ADDR: B5C499B055FE	RSSI: -40
MAC ADDR: B5C499B055FE	RSSI: -44
MAC ADDR: B5C499B055FE	RSSI: -42

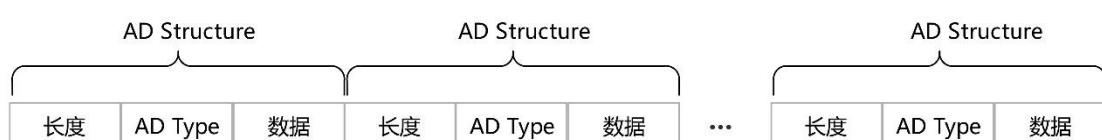
图 2-8：扫描获取的广播设备的设备地址和 RSSI

3.4.2. 获取从机设备名称实验

❖ 注：本节对应源码是“实验 2-2：扫描获取从机设备的设备名称”。

虽然设备名称、MAC 地址和 RSSI 都是通过 ble_gap_evt_adv_report_t 取数据，但是他们的获取方法是有所差异的。观察 ble_gap_evt_adv_report_t 结构体，我们会发现 MAC 地址、RSSI 直接获取就可以了，但是设备名称、UUID 等数据是存放在同一个数组里面的，数据按照广播报文数据的格式存放（即按照 AD Structure 的格式存放），因此，程序中需要对数组中的数据进行解析才能获取设备名称。

下图是广播报文数据的格式，由此可以看到解析数据时，根据长度逐个解析出 AD Structure，根据 AD Structure 中的 AD type 解析出当前 AD Structure 是不是设备名称。需要注意的是设备名称有两种类型：完整的设备名称和裁剪的设备名称，因此判断 AD type 时需要兼顾这两种类型。



Length=AD Type+数据的字节数

图 2-9：广播报文结构

设备名称解析函数代码清单如下：

代码清单：解析设备名称

```

1. static uint32_t find_adv_name(const ble_gap_evt_adv_report_t *p_adv_report)
2. {
3.     p_data = p_adv_report->data.p_data;
4.     //检索广播数据
5.     while (index < p_adv_report->data.len)
6.     {
7.         uint8_t field_length = p_data[index];
8.         uint8_t field_type   = p_data[index + 1];
9.         //如果 ADV_TYPE 是完整的设备名称或者裁剪的设备名称，打印出设备名称
10.        if ((field_type == BLE_GAP_AD_TYPE_COMPLETE_LOCAL_NAME) || (field_type ==
11.            BLE_GAP_AD_TYPE_SHORT_LOCAL_NAME))
12.        {
13.            printf(" name:");
14.            //串口打印设备名称
15.            for(i=0;i<field_length-1;i++)app_uart_put(p_data[index+2+i]);
16.            return NRF_SUCCESS;
17.        }
18.        index += field_length + 1;
19.    }
20.    return NRF_ERROR_NOT_FOUND;
21. }
```

事件处理函数 scan_evt_handler()中的 NRF_BLE_SCAN_EVT_NOT_FOUND 事件处理代码里面加入设备名称解析函数 find_adv_name()，这样每扫描到一个广播设备，如果广播包中存在设备名称，即可解析出设备名称并通过串口打印出设备名称。

代码清单：事件处理函数 scan_evt_handler()中加入设备名称解析函数

```

1. case NRF_BLE_SCAN_EVT_NOT_FOUND:
2. {
3.     //定义广播报告结构体指针 p_adv 并指向扫描事件结构体中的 p_not_found
4.     ble_gap_evt_adv_report_t const * p_adv =
5.             p_scan_evt->params.p_not_found;
6.     //查找广播数据中的设备名称，找到后通过串口打印出设备名称
7.     if(find_adv_name(p_adv) == NRF_SUCCESS)
8.     {
9.         printf("\r\n");
10.    }
11. } break;
```

■ 硬件连接

同“实验 2-1”。

■ 测试步骤

- 解压“…\5: 开发指南（下册-主机）配套实验源码\”目录下的压缩文件“实验 2-2: 扫描获取从机设备的设备名称”，将解压后得到的文件夹“ble_app_scan”拷贝到合适的目录，如“D\ nRF52840”。
- 启动 MDK5.27。
- 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_scan\project\mdk5”目录下的工程“ble_app_scan.uvproj”。
- 切换到协议栈 target，下载协议栈。
- 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。
- 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。
- 程序运行后，计算机上的串口调试助手会显示扫描到的广播设备的设备地址、RSSI 以及设备名称，如下图所示。

```
name:Nordic_UART
```

图 2-10：扫描获取的广播设备的设备名称

3.4.3. 获取 UUID 实验

◆ 注：本节对应源码是“实验 2-3：扫描获取从机设备的 UUID”。

UUID 和设备名称一样，也需要解析，解析时注意 UUID 的 AD Type 有多种，本例中解析以下 4 种 UUID 的 AD Type。

- BLE_GAP_AD_TYPE_16BIT_SERVICE_UUID_COMPLETE: 完整的 16 位 UUID 列表。
- BLE_GAP_AD_TYPE_16BIT_SERVICE_UUID_MORE_AVAILABLE: 部分 16 位 UUID 列表。
- BLE_GAP_AD_TYPE_128BIT_SERVICE_UUID_COMPLETE: 完整的 128 位 UUID 列表。
- BLE_GAP_AD_TYPE_128BIT_SERVICE_UUID_MORE_AVAILABLE: 部分 128 位

UUID 列表。

由此，我们编写的 UUID 解析函数代码清单如下：

代码清单：解析 UUID

```
1. static uint32_t find_adv_uuid(const ble_gap_evt_adv_report_t *p_adv_report)
2. {
3.     uint8_t i;
4.     uint32_t index = 0;
5.     uint32_t res = NRF_ERROR_NOT_FOUND;
6.     uint8_t * p_data;
7.
8.     p_data = p_adv_report->data.p_data;
9.     //检索广播数据
10.    while (index < p_adv_report->data.len)
11.    {
12.        uint8_t field_length = p_data[index];
13.        uint8_t field_type = p_data[index + 1];
14.        //如果 ADV_TYPE 是完整的 16 位 UUID 列表或者部分 16 位 UUID 列表，打印出 UUID
15.        if ((field_type == BLE_GAP_AD_TYPE_16BIT_SERVICE_UUID_COMPLETE) || (field_
16.            type == BLE_GAP_AD_TYPE_16BIT_SERVICE_UUID_MORE_AVAILABLE))
17.        {
18.            printf("16bit uuid:");
19.            //串口打印设备名称
20.            for(i=0;i<field_length-1;i++)printf("%02X",p_data[index+2+i]);
21.            res = NRF_SUCCESS;
22.        }
23.        //如果 ADV_TYPE 是完整的 128 位 UUID 列表或者部分 128 位 UUID 列表，打印出 UUID
24.        else if ((field_type == BLE_GAP_AD_TYPE_128BIT_SERVICE_UUID_COMPLETE) || (
25.            field_type == BLE_GAP_AD_TYPE_128BIT_SERVICE_UUID_MORE_AVAILABLE))
26.        {
27.            printf("128 complete uuid:");
28.            //串口打印设备名称
29.            for(i=0;i<field_length-1;i++)printf("%02X",p_data[index+2+i]);
30.            res = NRF_SUCCESS;
31.        }
32.        index += field_length + 1;
33.    }
34.    return res;
35. }
```

事件处理函数 scan_evt_handler()中的 NRF_BLE_SCAN_EVT_NOT_FOUND 事件处理代码里面加入设备 UUID 解析函数 find_adv_uuid()，这样每扫描到一个广播设备，如果广播包中存在 uuid，该函数即可解析出 uuid 并通过串口打印出 uuid。

代码清单：事件处理函数 scan_evt_handler() 中加入设备名称解析函数

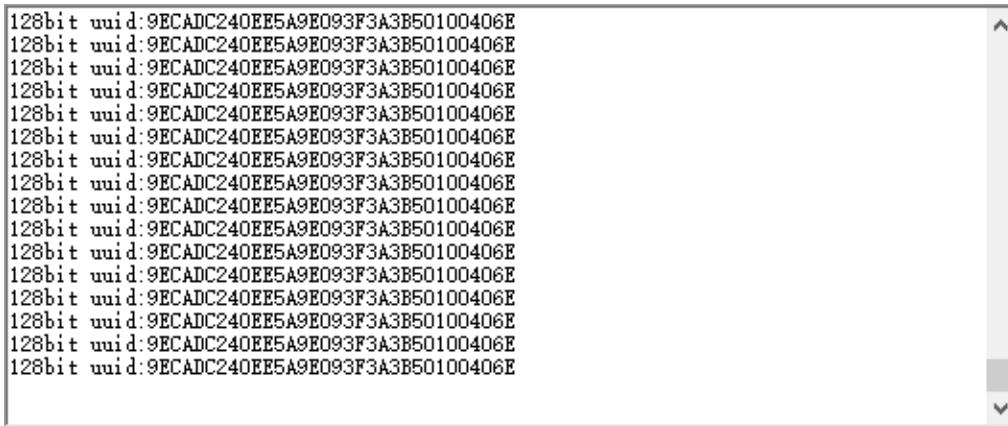
```
1. case NRF_BLE_SCAN_EVT_NOT_FOUND:
2. {
3.     // 定义广播报告结构体指针 p_adv 并指向扫描事件结构体中的 p_not_found
4.     ble_gap_evt_adv_report_t const * p_adv =
5.         p_scan_evt->params.p_not_found;
6.     // 广播数据中查找 UUID，并通过串口打印找到的 UUID
7.     if(find_adv_uuid(p_adv) == NRF_SUCCESS)
8.     {
9.         printf("\r\n");
10.    }
11. } break;
```

■ 硬件连接

同“实验 2-2”。

■ 测试步骤

1. 解压“…\5：开发指南（下册-主机）配套实验源码”目录下的压缩文件“实验 2-3：扫描获取从机设备的 UUID”，将解压后得到的文件夹“ble_app_scan”拷贝到合适的目录，如“D\pRF52840”。
2. 启动 MDK5.27。
3. 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_scan\project\mdk5”目录下的工程“ble_app_scan.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，计算机上的串口调试助手会显示扫描到的广播设备的设备地址、RSSI 以及设备名称，如下图所示。



```
128bit uid:9ECADC240EE5A9E093F3A3B50100406E
```

图 2-11：扫描获取的广播设备的 UUID

3.4.4. 其他数据

对于其他的广播数据如服务数据、发射功率等级等等，解析的方法和设备名称、UUID 类似，参考他们的解析函数很容易编写相应的解析代码，只要熟悉他们的 AD Structure 即可。

对于广播数据的 AD Structure 以及 AD type 的类型，读者可参考《开发指南下册-从机》种对广播的描述章节，这里不再赘述。

第三章：扫描信息过滤

1. 学习目的

- 掌握过滤的原理和作用。
- 掌握如何配置过滤器以及实现组合过滤器。

2. 过滤原理

主机执行扫描时，扫描到的从机设备并不一定都是主机需要的，这时候，主机就需要做过滤处理，过滤掉无用的设备的数据，只关心需要的设备的数据。

考虑一下这样的应用场景：如果我们在某个应用中需要主机只连接指定的 MAC 地址的从机设备或者只连接指定的设备名称的从机设备，主机应该怎么实现这样的功能？这时候，主机就可以使用过滤器来过滤掉非指定 MAC 地址或设备名称的从机设备。

那么，这里说的过滤器指的是什么？所谓的过滤器是 Nordic 提供一个程序模块，该程序模块实现了对 MAC 地址、设备名称、UUID 和外观的过滤。过滤器的原理示意图如下图所示，应用程序使用过滤器时需要先添加过滤器（可添加的过滤器类型有 MAC 地址、设备名称、UUID 或外观过滤器），之后使能过滤器。过滤器启动后，当扫描器接收到 BLE_GAP_EVT_ADV_REPORT 事件时过滤器开始起作用，当扫描数据和已使能的过滤器匹配时扫描器会执行后续操作（发起连接）并产生匹配事件通知应用程序，当扫描数据和已使能的过滤器不匹配时，扫描器会“放弃”该数据，值得注意的是，此时过滤器也会产生一个“不匹配”事件（NRF_BLE_SCAN_EVT_NOT_FOUND 事件）通知应用程序。

对于过滤器来说，还有一种特殊的情况，即应用中使用了白名单。白名单的优先级高于过滤器，也就是如果应用中使用了白名单，当主机扫描到的设备地址是白名单中的设备地址时，过滤器会直接“放行”并执行下一步操作，如发起连接。

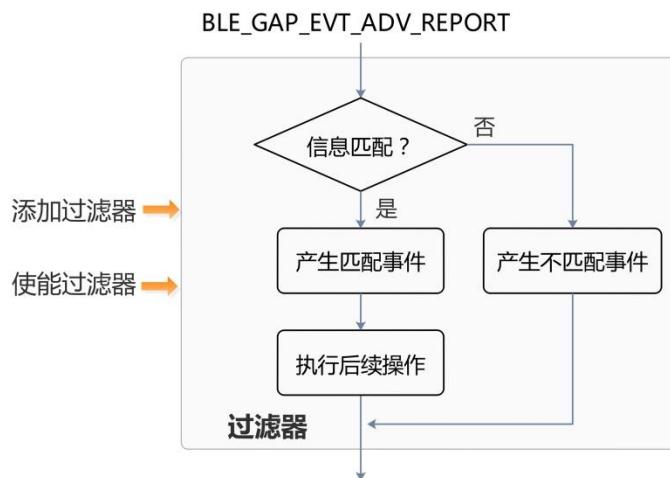


图 3-1：过滤器原理示意图

由过滤器原理示意图可以看到，对于过滤器来说，重点是过滤器类型以及过滤器的匹配

和事件。

■ 过滤器类型

过滤器类型如下表所示，共有 6 种，其中组合过滤器是前 5 种过滤器的任意组合。

表 3-1：过滤器类型

序号	过滤类型	描述
1	NRF_BLE_SCAN_NAME_FILTER	设备名称过滤器。
2	NRF_BLE_SCAN_ADDR_FILTER	设备地址过滤器。
3	NRF_BLE_SCAN_UUID_FILTER	UUID 过滤器。
4	NRF_BLE_SCAN_APPEARANCE_FILTER	外观过滤器。
5	NRF_BLE_SCAN_SHORT_NAME_FILTER	裁剪的设备名称过滤器。
6	NRF_BLE_SCAN_ALL_FILTER	组合过滤器。

■ 过滤器的匹配和事件

过滤器是否匹配决定了扫描器是否向该设备发起连接，而事件则是用来通知应用程序的，当然，应用程序可以选择是否关注该事件。

过滤器可产生的事件有：NRF_BLE_SCAN_EVT_FILTER_MATCH、NRF_BLE_SCAN_EVT_NOT_FOUND 和 NRF_BLE_SCAN_EVT_WHITELIST_ADV_REPORT，接下来我们看一下这些事件是在什么情况下产生的。

1. 使用非组合的过滤数据时：若扫描数据匹配过滤器则产生 NRF_BLE_SCAN_EVT_FILTER_MATCH 事件，否则产生 NRF_BLE_SCAN_EVT_NOT_FOUND 事件。

示例：添加了一个设备地址过滤器，当扫描到的设备的地址匹配设备地址过滤器时产生 NRF_BLE_SCAN_EVT_FILTER_MATCH 事件，不匹配时产生 NRF_BLE_SCAN_EVT_NOT_FOUND 事件。

2. 使用组合过滤数据时：若扫描数据匹配所有过滤器则产生 NRF_BLE_SCAN_EVT_FILTER_MATCH 事件，否则产生 NRF_BLE_SCAN_EVT_NOT_FOUND 事件。这里要注意一下，组合过滤器也可以只包含一种类型的过滤器，这时候若扫描数据匹配该过滤器就等于匹配所有的过滤器，同样会产生匹配事件。

示例 1：添加设备地址和设备名称的组合过滤器，当扫描到的设备的地址和设备名称均匹配过滤器时产生 NRF_BLE_SCAN_EVT_FILTER_MATCH 事件，否则产生 NRF_BLE_SCAN_EVT_NOT_FOUND 事件。

3. 若扫描到的设备的设备地址存在于白名单中，过滤器不会执行过滤，而是直接进行下一步操作如发起连接，同时过滤器会产生 NRF_BLE_SCAN_EVT_WHITELIST_ADV_REPORT 事件通知应用程序。

3. 程序编写

过滤器的使用比较简单，应用程序只需根据自身需求添加过滤器，并使能过滤器即可，若应用程序需要关注过滤器事件，只需在扫描初始化时注册扫描事件处理函数即可接收过滤器事件。

3.1. 添加过滤器

应用程序添加过滤器的函数是 `nrf_ble_scan_filter_set()`，函数原型如下表所示。通过该函数可将任一类型的过滤器添加到扫描器，一般地，添加过滤器会在扫描初始化函数中执行。

表 3-2: `nrf_ble_scan_filter_set()` 函数

格式说明符	描述	示例
函数原型	<pre>ret_code_t nrf_ble_scan_filter_set (nrf_ble_scan_t *const p_scan_ctx, nrf_ble_scan_filter_type_t type, void const * p_data)</pre>	
函数功能	该函数添加类型为 <code>nrf_ble_scan_filter_type_t</code> 的新过滤器，若给定类型的过滤器数量不超过 <code>NRF_BLE_SCAN_UUID_CNT</code> 、 <code>NRF_BLE_SCAN_NAME_CNT</code> 、 <code>NRF_BLE_SCAN_ADDRESS_CNT</code> 或 <code>NRF_BLE_SCAN_APPEARANCE_CNT</code> 的值，则添加过滤器。	
参数	<p>[in,out] <code>p_scan_ctx</code>: 指向扫描模块实例。</p> <p>[in] <code>type</code>: 过滤器类型。</p> <p>[in] <code>p_data</code>: 要添加的过滤器数据。</p>	
返回值	<p><code>NRF_SUCCESS</code>: 添加过滤器成功。</p> <p><code>NRF_ERROR_NULL</code>: 将 NULL 指针作为输入参数。</p> <p><code>NRF_ERROR_DATA_SIZE</code>: 设备名称过滤器长度过长。最大长度对应于 <code>NRF_BLE_SCAN_NAME_MAX_LEN</code>。</p> <p><code>NRF_ERROR_NO_MEMORY</code>: 添加的过滤器数量超过可用过滤器的数量。</p> <p><code>NRF_ERROR_INVALID_PARAM</code>: 过滤器类型错误，可用的过滤器类型：<code>nrf_ble_scan_filter_type_t</code>。</p> <p><code>BLE_ERROR_GAP_INVALID_BLE_ADDR</code>: BLE 地址类型无效。</p>	

■ 示例：向扫描器中添加一个设备地址过滤器

添加设备地址过滤器时，先要定义过滤器数据，也就是我们的目标设备地址（不希望被过滤的设备的设备地址）。

代码清单：定义设备地址

```

1. //设备地址过滤器数据，该地址不会被过滤器滤掉
2. static ble_gap_addr_t const m_target_periph_addr =
3. {
4.     //地址类型：随机静态地址
5.     .addr_type = BLE_GAP_ADDR_TYPE_RANDOM_STATIC,
6.     .addr      = {0x17, 0xB7, 0xD5, 0x59, 0x2A, 0xC9}

```

```
7. };
```

之后，调用 nrf_ble_scan_filter_set()函数添加到扫描器即可。

代码清单：向扫描器中添加一个设备地址过滤器

```
1. //添加一个设备地址过滤器
2. err_code = nrf_ble_scan_filter_set(&m_scan,
3.                                     SCAN_ADDR_FILTER,
4.                                     m_target_periph_addr.addr);
5. APP_ERROR_CHECK(err_code);
```

3.2. 使能过滤器

过滤器通过函数 nrf_ble_scan_filters_enable()使能，使能的过滤器类型必须是已经添加的，否则函数会返回错误，过滤器使能函数原型如下表所示。

表 3-3: nrf_ble_scan_filters_enable()函数

函数原型	<pre>ret_code_t nrf_ble_scan_filters_enable (nrf_ble_scan_t *const p_scan_ctx, uint8_t mode, bool match_all)</pre>
函数功能	<p>该函数用于使能过滤器。 过滤器可以相互组合。例如，我们可以启用一个过滤器或多个过滤器，如 (NRF_BLE_SCAN_NAME_FILTER NRF_BLE_SCAN_UUID_FILTER)启用 UUID 和设备名称过滤器。</p>
参 数	<p>[in] <code>p_scan_ctx</code>: 指向扫描模块实例。 [in] <code>mode</code>: 过滤器类型。 [in] <code>match_all</code> : 如果此标志置位，则所有已使能的过滤器类型必须匹配，<code>NRF_BLE_SCAN_EVT_FILTER_MATCH</code> 事件才能提交给主应用程序，否则，只要有一个过滤器类型匹配即会触发事件。</p>
返回值	<p><code>NRF_SUCCESS</code>: 过滤器使能成功。 <code>NRF_ERROR_INVALID_PARAM</code>: 过滤器类型错误，可用的过滤器类型：<code>nrf_ble_scan_filter_type_t</code>。 <code>NRF_ERROR_NULL</code>: 将 <code>NULL</code> 指针作为输入参数。</p>

■ 示例：使能设备地址过滤器

代码清单：使能设备地址过滤器

```
1. err_code = nrf_ble_scan_filters_enable(&m_scan,
```

```

2.                                     //过滤器类型为设备地址过滤器
3.                                     NRF_BLE_SCAN_ADDR_FILTER,
4.                                     false);
5. APP_ERROR_CHECK(err_code);

```

3.3. 过滤设备地址实验

❖ 注：本实验在“实验 2-1：扫描获取从机设备地址和 RSSI”的基础上修改，对应的实验源码是：“实验 3-1：过滤设备地址”。

3.3.1. 实验内容

向扫描器添加一个设备地址过滤器，过滤数据为随机静态地址{0x11, 0x22, 0x33, 0x44, 0x55, 0xCC}。应用程序在扫描初始化时注册事件处理函数 scan_evt_handler()接收事件，当扫描器扫描到地址匹配的从机设备后向应用程序提交 NRF_BLE_SCAN_EVT_FILTER_MATCH 事件，此时串口打印“MAC Address filter match”，当从机地址不匹配时，提交 NRF_BLE_SCAN_EVT_NOT_FOUND 事件，串口打印“not match”。

3.3.2. 程序编写

首先定义过滤器数据，本实验过滤数据为随机静态地址{0x11, 0x22, 0x33, 0x44, 0x55, 0xCC}。

代码清单：定义设备地址

```

1. //设备地址过滤器数据，该地址不会被过滤器滤掉
2. static ble_gap_addr_t const m_target_periph_addr =
3. {
4.     //地址类型：随机静态地址
5.     .addr_type = BLE_GAP_ADDR_TYPE_RANDOM_STATIC,
6.     .addr      = { 0x11, 0x22, 0x33, 0x44, 0x55, 0xCC}
7. };

```

接着，在扫描初始化函数中调用 nrf_ble_scan_filter_set()函数添加地址过滤器并使能过滤器。

代码清单：扫描初始化函数中添加一个设备地址过滤器

```

1. static void scan_init(void)
2. {
3.     ret_code_t          err_code;
4.     //定义扫描初始化结构体变量
5.     nrf_ble_scan_init_t init_scan;
6.     //先清零，再配置
7.     memset(&init_scan, 0, sizeof(init_scan));
8.     //自动连接设置为 false
9.     init_scan.connect_if_match = false;
10.    //使用初始化结构体中的扫描参数配置扫描器，这里 p_scan_param 指向定义的扫描参数

```

```
11.     init_scan.p_scan_param      = &m_scan_param;
12.     //conn_cfg_tag 设置为 1
13.     init_scan.conn_cfg_tag      = APP_BLE_CONN_CFG_TAG;
14.     //初始化扫描器
15.     err_code = nrf_ble_scan_init(&m_scan, &init_scan, scan_evt_handler);
16.     APP_ERROR_CHECK(err_code);
17.
18.     //向扫描器添加一个设备地址过滤器
19.     err_code = nrf_ble_scan_filter_set(&m_scan,
20.                                         SCAN_ADDR_FILTER,
21.                                         m_target_periph_addr.addr);
22.     APP_ERROR_CHECK(err_code);
23.     //使能设备地址过滤器
24.     err_code = nrf_ble_scan_filters_enable(&m_scan,
25.                                         NRF_BLE_SCAN_ADDR_FILTER,
26.                                         false);
27.     //检查函数返回值
28.     APP_ERROR_CHECK(err_code);
29. }
```

扫描事件处理函数中处理 NRF_BLE_SCAN_EVT_FILTER_MATCH 和 NRF_BLE_SCAN_EVT_NOT_FOUND 事件，在这两个事件里分别通过串口打印各自的提示信息。

代码清单：扫描事件处理函数中

```
1. static void scan_evt_handler(scan_evt_t const * p_scan_evt)
2. {
3.     //翻转 LED 指示灯 D4 的状态，指示扫描到设备
4.     nrf_gpio_pin_toggle(LED_4);
5.     //判断事件类型
6.     switch(p_scan_evt->scan_evt_id)
7.     {
8.         //扫描数据匹配地址过滤器
9.         case NRF_BLE_SCAN_EVT_FILTER_MATCH:
10.            {
11.                //串口打印匹配提示信息
12.                printf("MAC Address filter match\r\n");
13.            } break;
14.         //扫描数据不匹配地址过滤器
15.         case NRF_BLE_SCAN_EVT_NOT_FOUND:
16.            {
17.                //串口打印不匹配提示信息
18.                printf("not match\r\n");
19.            } break;
20.     } //扫描超时事件
```

```

21.     case NRF_BLE_SCAN_EVT_SCAN_TIMEOUT:
22.     {
23.         NRF_LOG_INFO("Scan timed out.");
24.         //重启扫描
25.         scan_start();
26.     } break;
27.     default:
28.     break;
29. }
30. }
```

“sdk_config.h”文件中使能过滤器并设置过滤器的数量，本实验添加了1个设备地址过滤器，因此NRF_BLE_SCAN_ADDRESS_CNT要设置为1。

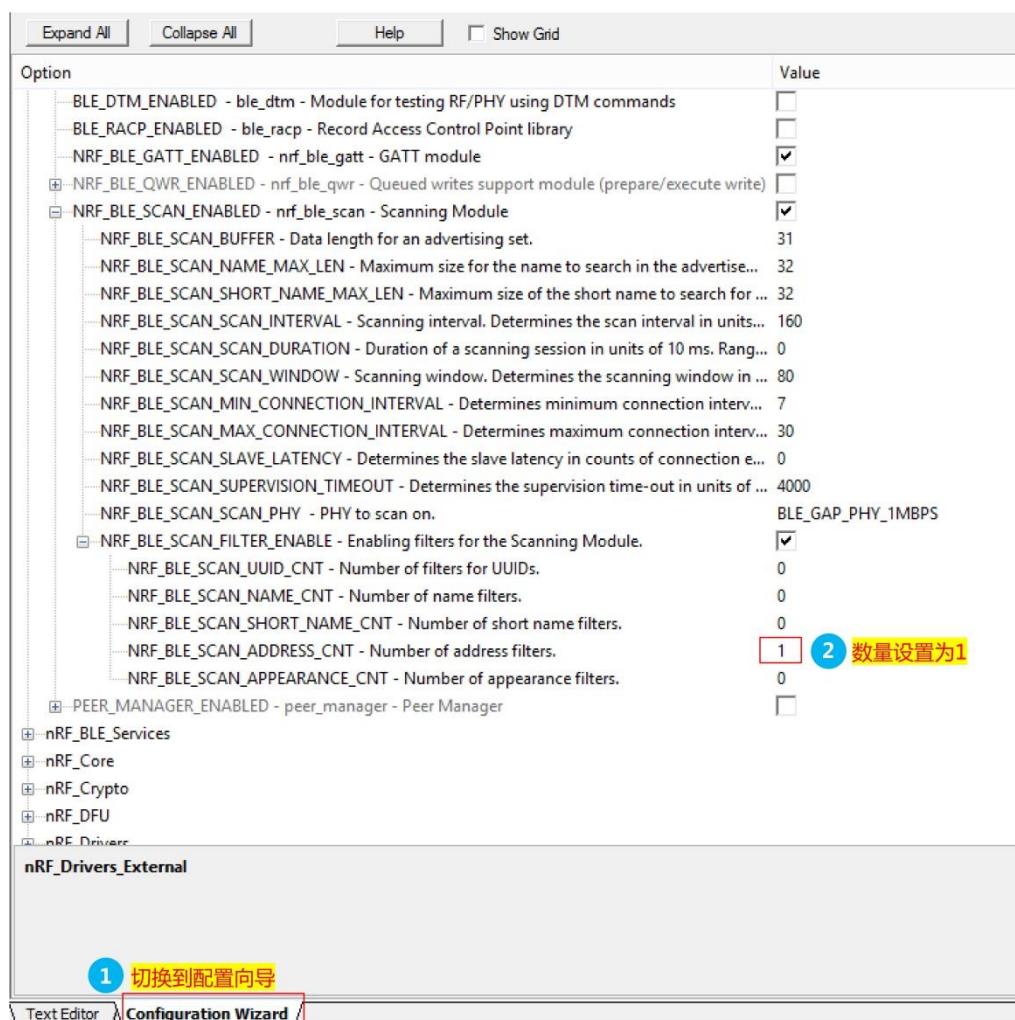


图 3-2：设置地址过滤器数量

3.3.3. 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，并使用USB数据线连接开发板和计算机。

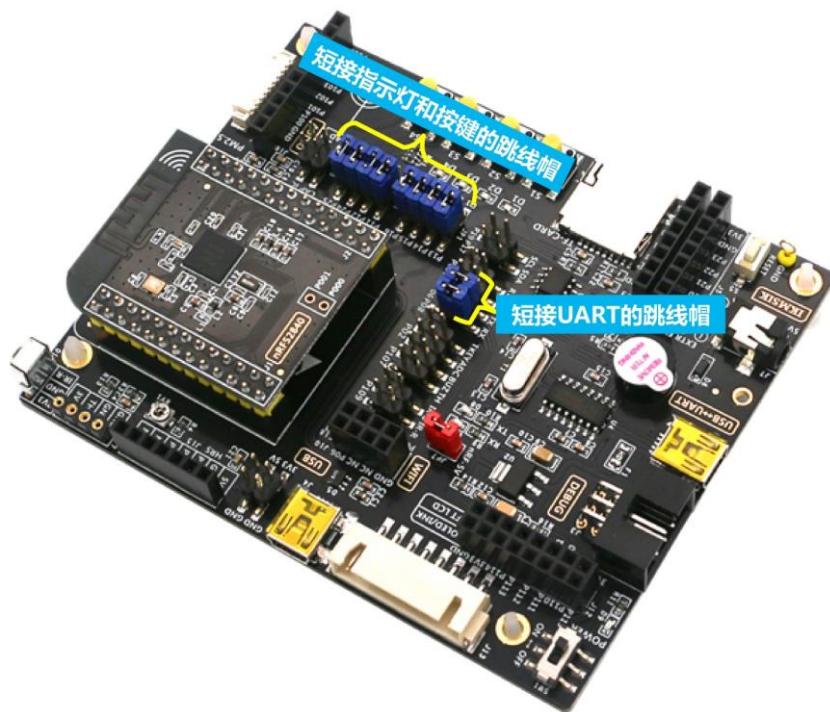


图 3-9：开发板跳线帽短接测试步骤

3.3.4. 测试步骤

1. 解压“…\5：开发指南（下册-主机）配套实验源码\”目录下的压缩文件“实验 3-1：过滤设备地址”，将解压后得到的文件夹“ble_app_scan”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_scan\project\mdk5” 目录下的工程“ble_app_scan.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，主机开发板上的指示灯 D1 闪烁指示正在扫描。
8. 将另外一块开发板烧写《开发指南下册-从机》的“实验 4-1：读设备地址、写随机静态设备地址”，该实验的设备地址是：随机静态地址{0x11, 0x22, 0x33, 0x44, 0x55, 0xCC}，和过滤器匹配，这时串口调试助手会显示“MAC Address filter match”。
9. 修改“实验 4-1”中的设备地址，重新编译后烧写到作为从机的开发板，这时，因为地址和过滤器不匹配，串口调试助手会显示“not match”。

3.4. 过滤设备名称实验

❖ 注：本实验在“实验 3-1：过滤设备地址”的基础上修改，本实验对应的实验源码是：

“实验 3-2：过滤设备名称”。

3.4.1. 实验内容

向扫描器添加一个设备名称过滤器，过滤数据为“BLE_Template”。应用程序在扫描初始化时注册事件处理函数 scan_evt_handler() 接收事件，当扫描器扫描到地址匹配的从机设备后向应用程序提交 NRF_BLE_SCAN_EVT_FILTER_MATCH 事件，此时串口打印“device name filter match”。

3.4.2. 程序编写

和地址过滤器一样，首先定义过滤器数据，本实验过滤数据为“BLE_Template”。

代码清单：定义设备名称

```
1. //设备名称过滤器数据，具有该设备名称的从机不会被过滤器滤掉
2. static char const m_target_periph_name[] = "BLE_Template";
```

接着，在扫描初始化函数中调用 nrf_ble_scan_filter_set() 函数添加设备名称过滤器并使能过滤器。

代码清单：扫描初始化函数中添加一个设备名称过滤器

```
1. //向扫描器添加一个设备名称过滤器
2. err_code = nrf_ble_scan_filter_set(&m_scan,
3.                                     SCAN_NAME_FILTER,
4.                                     m_target_periph_name);
5. APP_ERROR_CHECK(err_code);
6. //使能设备名称过滤器
7. err_code = nrf_ble_scan_filters_enable(&m_scan,
8.                                         NRF_BLE_SCAN_NAME_FILTER,
9.                                         false);
10. //检查函数返回值
11. APP_ERROR_CHECK(err_code);
```

扫描事件处理函数中处理的方式和地址过滤器一样，也是处理 NRF_BLE_SCAN_EVT_FILTER_MATCH 和 NRF_BLE_SCAN_EVT_NOT_FOUND 事件，读者参考地址过滤器实验中的描述和实验源码即可，本节后续的实验不再描述事件的处理。

“sdk_config.h”文件中使能过滤器并设置过滤器的数量，本实验添加了 1 个设备名称过滤器，因此 NRF_BLE_SCAN_NAME_CNT 要设置为 1。

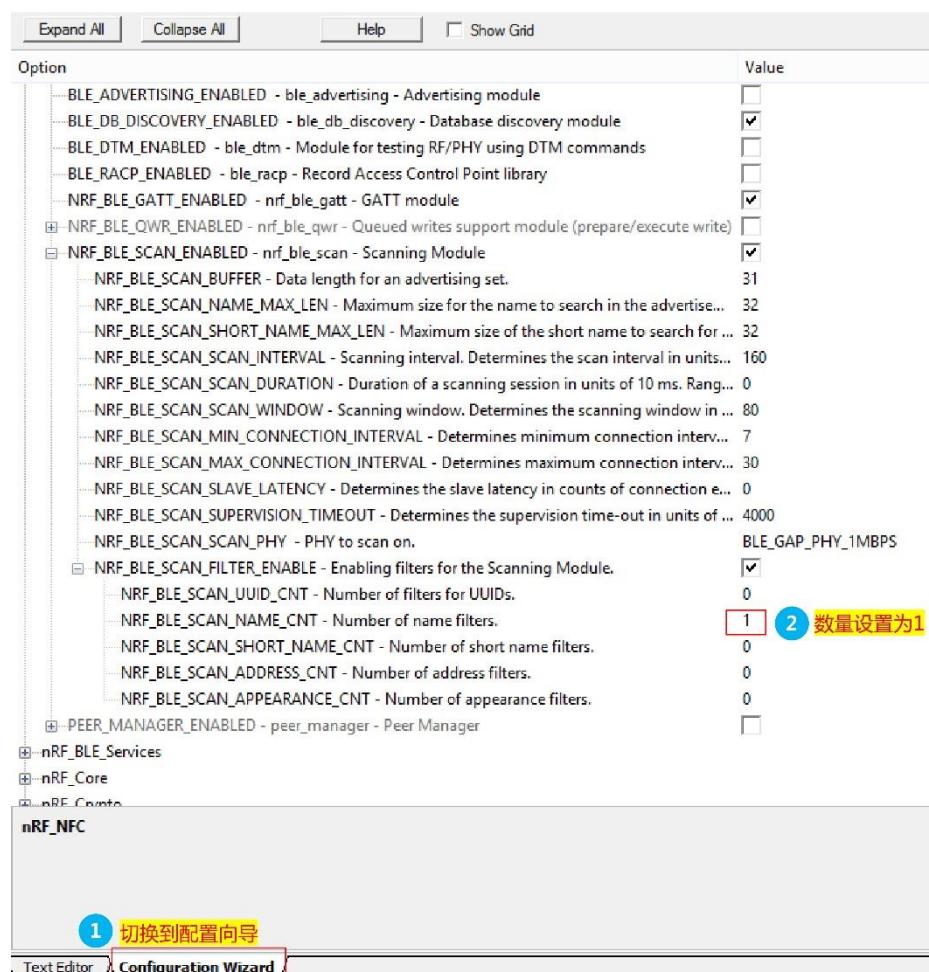


图 3-4：设置名称过滤器数量

3.4.3. 硬件连接

同“实验 3-1”。

3.4.4. 测试步骤

- 解压“…\5：开发指南（下册-主机）配套实验源码\”目录下的压缩文件“实验 3-2：过滤设备名称”，将解压后得到的文件夹“ble_app_scan”拷贝到合适的目录，如“D\ nRF52840”。
- 启动 MDK5.27。
- 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_scan\project\mdk5”目录下的工程“ble_app_scan.uvproj”。
- 切换到协议栈 target，下载协议栈。
- 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。
- 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，主机开发板上的指示灯 D1 闪烁指示正在扫描。
8. 将另外一块开发板烧写《开发指南下册-从机》的“实验 4-3：广播中加入设备名称”，该实验的设备默认的设备名称是：全称“BLE_Template”，和过滤器匹配，这时串口调试助手会显示“device name filter match”。
9. 修改“实验 4-3”中的设备名称，如改为“BLE_Temp”，重新编译后烧写到作为从机的开发板，这时，可以观察到主机的指示灯 D4 闪烁表示主机扫描到从机设备，但是因为设备名称和过滤器不匹配，串口不会打印信息。

3.5. 过滤 UUID 实验

- ❖ 注：本实验在“实验 2-2：扫描获取从机设备地址和 RSSI”的基础上修改，对应的实验源码是：“实验 3-3：过滤 UUID”。
- ❖ 程序中使用了心率服务的 UUID，因此 main.c 文件中要引用头文件“ble_srv_common.h”。

3.5.1. 实验内容

向扫描器添加如下 2 个 UUID 过滤器：

- 1 个过滤数据为标准 UUID（心率服务的 UUID：0x180D）。
- 1 个过滤数据为自定义 UUID：基数是{0x40, 0xE3, 0x4A, 0x1D, 0xC2, 0x5F, 0xB0, 0x9C, 0xB7, 0x47, 0xE6, 0x43, 0x00, 0x00, 0x53, 0x86}，16 位 UUID 是 0x000A。

应用程序在扫描初始化时注册事件处理函数 scan_evt_handler()接收事件，当扫描器扫描到 UUID 匹配的从机设备后向应用程序提交 NRF_BLE_SCAN_EVT_FILTER_MATCH 事件，此时串口打印“uuid filter match”，同时为了方便区分设备，串口打印匹配提示信息时也会打印设备的 MAC 地址。

3.5.2. 程序编写

本例需要定义两种类型的 UUID 作为过滤数据：一个标准的心率服务 UUID 和一个自定义的 UUID（开发指南上册-从机的实验 7-3：串口透传长包传输-判断 notify 所用的 UUID）。注意，自定义的 UUID 是需要定义基数的。

代码清单：定义 UUID 过滤器数据

```

1. //UUID 过滤器数据：标准 UUID
2. ble_uuid_t uuid_ble =
3. {
4.     .uuid = BLE_UUID_HEART_RATE_SERVICE,//心率服务 UUID
5.     .type = BLE_UUID_TYPE_BLE,//类型为标准 UUID
6. };
7.
8. //自定义 UUID 基数，添加自定义 UUID 时需要写入 UUID 基数
9. #define MY_BASE_UUID  {{0x40, 0xE3, 0x4A, 0x1D, 0xC2, 0x5F, 0xB0, 0x9C, 0xB7,
10.                      0x47, 0xE6, 0x43, 0x00, 0x00, 0x53, 0x86}}
11. //UUID 过滤器数据：自定义 UUID
12. static ble_uuid_t const uuid_custom =
13. {
```

```

14.     .uuid = 0x000A, //UUID 数值, 会替换 UUID 基数中的第 12 和第 13 个字节
15.     .type = BLE_UUID_TYPE_VENDOR_BEGIN//类型为自定义 UUID
16. };

```

接着, 在扫描初始化函数中调用 nrf_ble_scan_filter_set()函数添加 UUID 过滤器并使能过滤器, 这里要注意以下两点:

- 1) 本例使用了 2 个 UUID 过滤器, 因此, 要执行 2 次添加 UUID 过滤器的操作, 对于设备地址过滤器、设备名称过滤器以及下一节中讲解的外观过滤器也是一样, 如果有多个过滤器时, 需要执行多次添加过滤器操作。
- 2) UUID 过滤器数据为自定义的 UUID 时, 需要向协议栈写入自定义的 UUID 基数。

代码清单: 扫描初始化函数中写入自定义 UUID 基数和添加 UUID 过滤器

```

1. static void scan_init(void)
2. {
3.     ret_code_t          err_code;
4.     //定义扫描初始化结构体变量
5.     nrf_ble_scan_init_t init_scan;
6.
7.     //写入自定义 UUID 基数
8.     uint8_t              uuid_type;
9.     ble_uuid128_t        uarts_base_uuid = MY_BASE_UUID;
10.    err_code = sd_ble_uuid_vs_add(&uarts_base_uuid, &uuid_type);
11.    APP_ERROR_CHECK(err_code);
12.
13.    //先清零扫描初始化结构体, 再配置
14.    memset(&init_scan, 0, sizeof(init_scan));
15.    //自动连接设置为 false
16.    init_scan.connect_if_match = false;
17.    //使用初始化结构体中的扫描参数配置扫描器, 这里 p_scan_param 指向定义的扫描参数
18.    init_scan.p_scan_param      = &m_scan_param;
19.    //conn_cfg_tag 设置为 1
20.    init_scan.conn_cfg_tag     = APP_BLE_CONN_CFG_TAG;
21.    //初始化扫描器
22.    err_code = nrf_ble_scan_init(&m_scan, &init_scan, scan_evt_handler);
23.    APP_ERROR_CHECK(err_code);
24.
25.    //向扫描器添加第 1 个 UUID 过滤器, 过滤数据为标准 UUID
26.    err_code = nrf_ble_scan_filter_set(&m_scan,
27.                                       SCAN_UUID_FILTER,
28.                                       &uuid_ble);
29.    APP_ERROR_CHECK(err_code);
30.    //向扫描器添加第 2 个 UUID 过滤器, 过滤数据为自定义 UUID
31.    err_code = nrf_ble_scan_filter_set(&m_scan,

```

```

32.                               SCAN_UUID_FILTER,
33.                               &uuid_custom);
34. APP_ERROR_CHECK(err_code);
35.
36. //使能 UUID 过滤器
37. err_code = nrf_ble_scan_filters_enable(&m_scan,
38.                                         NRF_BLE_SCAN_UUID_FILTER,
39.                                         false);
40. //检查函数返回值
41. APP_ERROR_CHECK(err_code);
42. }

```

“sdk_config.h”文件中使能过滤器并设置过滤器的数量，本实验添加了2个UUID过滤器，因此NRF_BLE_SCAN_UUID_CNT要设置为2。

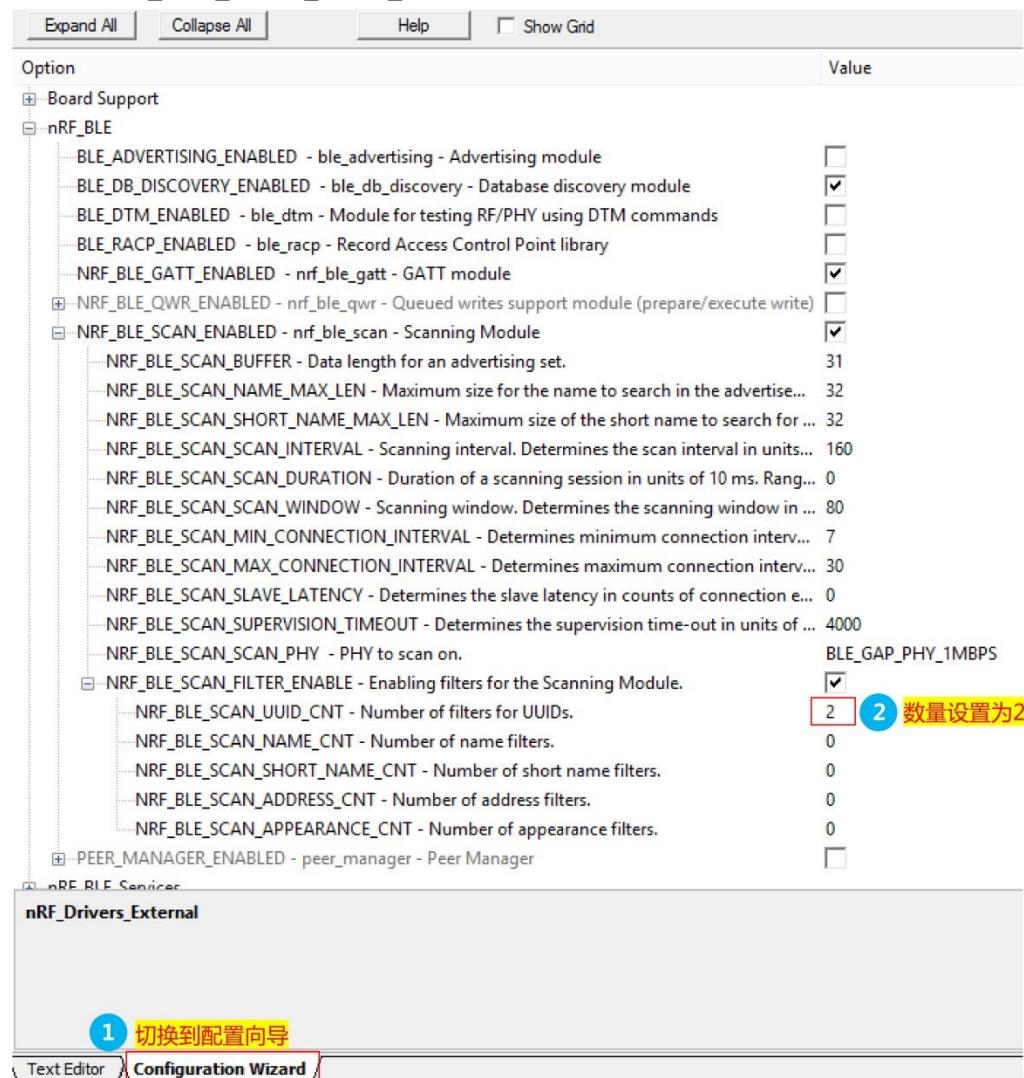


图 3-5：设置 UUID 过滤器数量

3.5.3. 硬件连接

同“实验 3-1”。

3.5.4. 测试步骤

1. 解压“…\5: 开发指南（下册-主机）配套实验源码\”目录下的压缩文件“实验 3-3: 过滤 UUID”，将解压后得到的文件夹“ble_app_scan”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_scan\project\mdk5”目录下的工程“ble_app_scan.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，主机开发板上的指示灯 D1 闪烁指示正在扫描。
8. 将另外两块开发板分别烧写《开发指南下册-从机》的“实验 6-1: 实现心率 Profile”，和“实验 7-3: 串口透传长包传输-判断 notify”。该实验的 UUID 和过滤器匹配，这时串口调试助手会显示“uuid filter match”。

```

MAC ADDR:3A2CA22281C1 uuid filter match
MAC ADDR:3A2CA22281C1 uuid filter match
MAC ADDR:1F7F0207FDEF uuid filter match

```

图 3-6: UUID 匹配信息

3.6. 过滤外观实验

◆ 注：本实验在“实验 2-2: 扫描获取从机设备地址和 RSSI”的基础上修改，对应的实验源码是：“实验 3-4: 过滤外观”。

3.6.1. 实验内容

向扫描器添加如下 1 个外观过滤器（心率腕带，编号 833）。应用程序在扫描初始化时注册事件处理函数 scan_evt_handler()接收事件，当扫描器扫描到 UUID 匹配的从机设备后向应用程序提交 NRF_BLE_SCAN_EVT_FILTER_MATCH 事件，此时串口打印“appearance filter match”。

3.6.2. 程序编写

首先定义外观过滤器数据，本实验过滤数据为“心率腕带”，对应编码：833。

代码清单： 定义外观过滤器数据

```
1. //外观过滤器数据（心率腕带），具有该外观的从机不会被过滤器滤掉
2. static uint16_t const m_target_periph_appearance = BLE_APPEARANCE_HEART_RATE_SENSOR_HEART_RATE_BELT;
```

接着，在扫描初始化函数中调用 nrf_ble_scan_filter_set() 函数添加外观过滤器并使能过滤器。

代码清单： 扫描初始化函数中添加一个外观过滤器

```
1. static void scan_init(void)
2. {
3.     ret_code_t         err_code;
4.     //定义扫描初始化结构体变量
5.     nrf_ble_scan_init_t init_scan;
6.     //先清零，再配置
7.     memset(&init_scan, 0, sizeof(init_scan));
8.     //自动连接设置为 false
9.     init_scan.connect_if_match = false;
10.    //使用初始化结构体中的扫描参数配置扫描器，这里 p_scan_param 指向定义的扫描参数
11.    init_scan.p_scan_param      = &m_scan_param;
12.    //conn_cfg_tag 设置为 1
13.    init_scan.conn_cfg_tag     = APP_BLE_CONN_CFG_TAG;
14.    //初始化扫描器
15.    err_code = nrf_ble_scan_init(&m_scan, &init_scan, scan_evt_handler);
16.    APP_ERROR_CHECK(err_code);
17.
18.    //向扫描器添加一个外观过滤器
19.    err_code = nrf_ble_scan_filter_set(&m_scan,
20.                                         SCAN_APPEARANCE_FILTER,
21.                                         &m_target_periph_appearance);
22.    APP_ERROR_CHECK(err_code);
23.    //使能外观过滤器
24.    err_code = nrf_ble_scan_filters_enable(&m_scan,
25.                                         NRF_BLE_SCAN_APPEARANCE_FILTER,
26.                                         false);
27.    //检查函数返回值
28.    APP_ERROR_CHECK(err_code);
29. }
```

“`sdk_config.h`”文件中使能过滤器并设置过滤器的数量，本实验添加了1个外观过滤器，因此`NRF_BLE_SCAN_APPEARANCE_CNT`要设置为1。

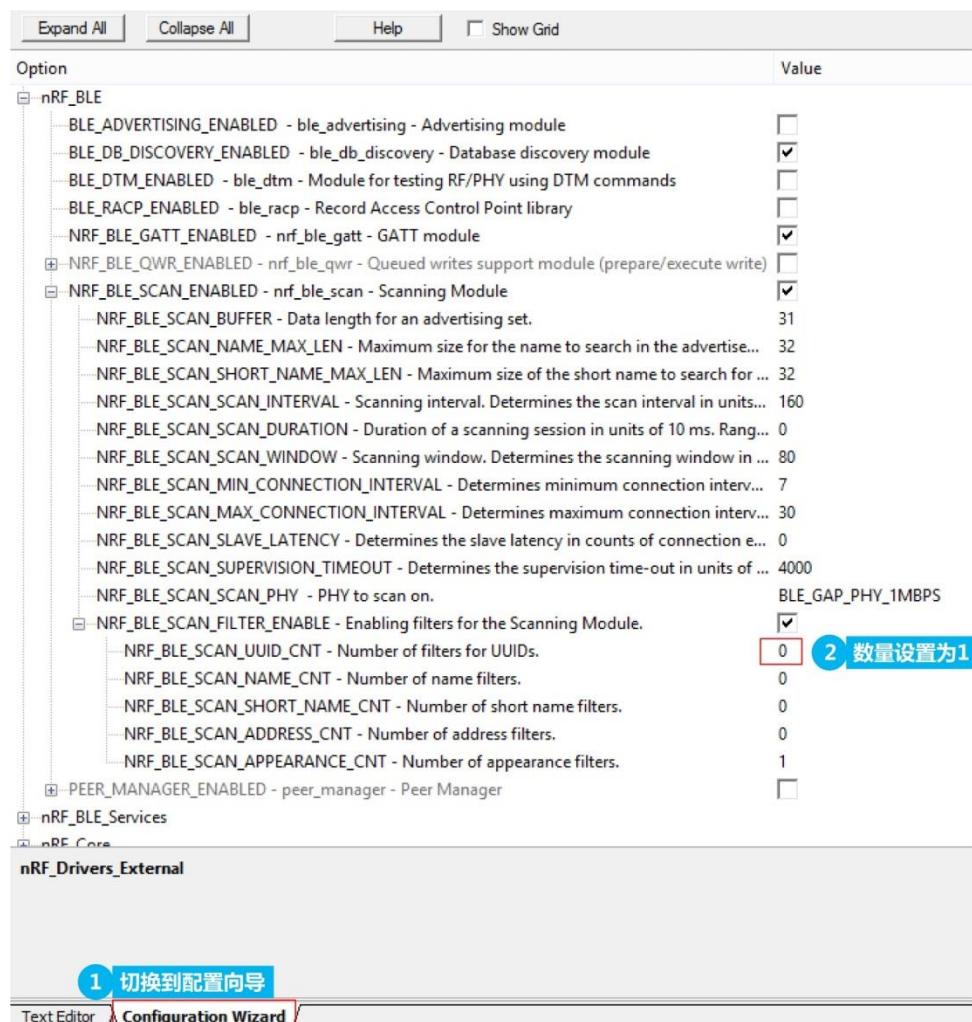


图 3-7：设置外观过滤器数量

3.6.3. 硬件连接

同“实验3-1”。

3.6.4. 测试步骤

- 解压“…\5：开发指南（下册-主机）配套实验源码\”目录下的压缩文件“实验3-4：过滤外观”，将解压后得到的文件夹“ble_app_scan”拷贝到合适的目录，如“D\NRF52840”。
- 启动MDK5.27。
- 在MDK5中执行“Project→Open Project”打开“…\ble_app_scan\project\mdk5”目录下的工程“ble_app_scan.uvproj”。
- 切换到协议栈target，下载协议栈。
- 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。
- 点击下载按钮下载程序。

7. 程序运行后，主机开发板上的指示灯 D1 闪烁指示正在扫描。
8. 将作为从机的开发板烧写《开发指南下册-从机》的“实验 6-1：实现心率 Profile”。该实验的外观是：心率腕带，和过滤器匹配，因此会观察到串口调试助手会显示“appearance filter match”。

第四章：发起连接

1. 学习目的

- 掌握连接的流程及事件。
- 掌握主机建立连接部分代码的编写。

2. 建立连接流程

设备依靠广播只能传输少量的数据，同时广播也是一个不安全的操作，因为广播者不知道他发送的数据有没有被扫描设备接收到，因此，为了实现更加复杂和安全可靠的数据传输，设备之间需要建立连接，建立连接的流程如下图所示。

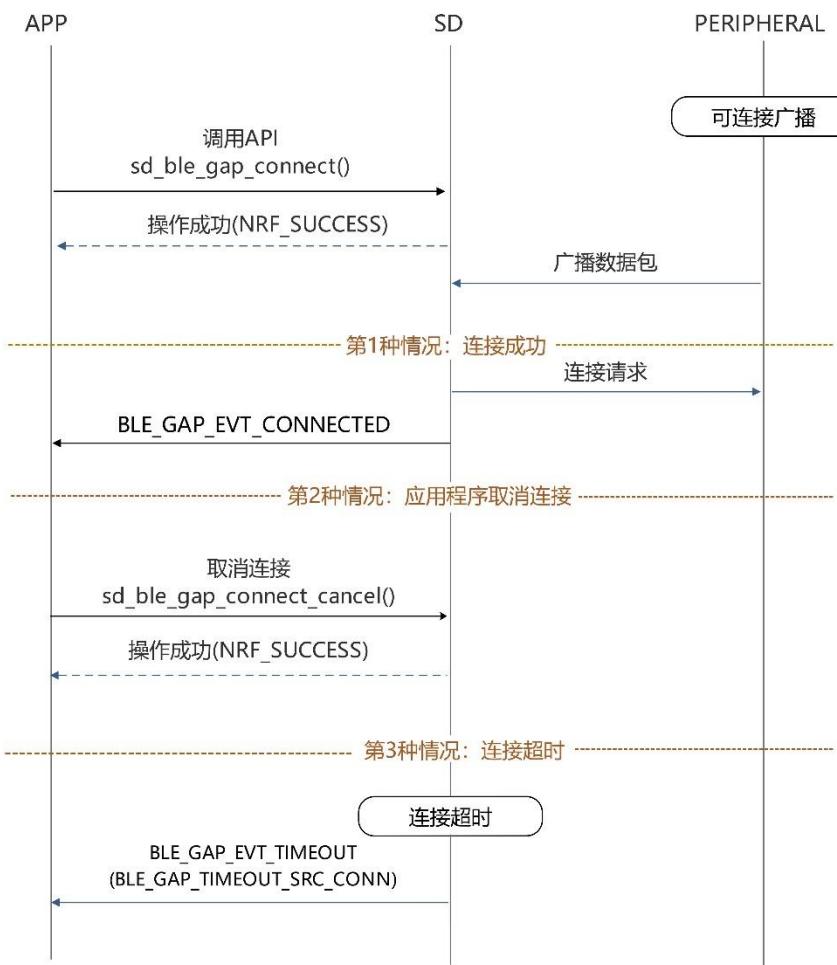


图 4-1：建立连接流程

外围设备发送可连接广播，中心设备通过扫描获取广播设备信息、通过过滤器甄别设备，若该设备通过过滤器（即该设备是自己要去连接的设备），中心设备调用 API 函数 `sd_ble_gap_connect()`发起连接，`sd_ble_gap_connect()`函数调用后，后续流程是下面 3 种情况之一。

- 第1种情况：协议栈发送连接请求，连接请求中包含了一些必要的参数，外围设备接收连接请求并作为从机进入连接，中心设备作为主机进入连接，连接建立成功，协议栈会向应用程序提交“BLE_GAP_EVT_CONNECTED”事件。一旦连接建立，主从设备之间不再使用广播信道进行通讯，而使用数据信道通信。
- 第2种情况：建立连接是需要一定的时间的，主机发现从机没有响应的时候，可以主动取消连接请求（调用sd_ble_gap_connect_cancel()函数取消）。
- 第3种情况：从机无响应，连接超时，协议栈向应用程序提交“BLE_GAP_EVT_TIMEOUT(BLE_GAP_TIMEOUT_SRC_CONN)”事件。

3. 程序编写

- ❖ 注1：本实验在“实验3-3：过滤UUID”的基础上修改，连接的从机为《开发指南下册-从机》中的“实验7-3：串口透传长包传输-判断notify”，本实验对应的实验源码是：“实验4-1：建立连接”。
- ❖ 注2：因为本章讲的是连接的建立，下一章才会讲到MTU交换，因此本章的实验中没有加入MTU交换相关的实现代码，所以当连接建立后，会周期性的断开。

程序中对连接的处理一方面是扫描初始化时配置“过滤器匹配后自动连接”，另一方面注册BLE事件监视者，关注连接相关的事件。同时，因为要连接的从机是自定义UUID：基数是{0x40, 0xE3, 0x4A, 0x1D, 0xC2, 0x5F, 0xB0, 0x9C, 0xB7, 0x47, 0xE6, 0x43, 0x00, 0x00, 0x53, 0x86}，16位UUID是0x0000A，因此程序中删除过滤数据为标准UUID的过滤器并在“sdk_config.h”文件中将NRF_BLE_SCAN_UUID_CNT设置为1。

1. 扫描初始化时配置“过滤器匹配后自动连接”

扫描初始化函数scan_init()中将扫描初始化结构体中的“connect_if_match”设置为“true”，代码清单如下。这样，当扫描的设备匹配过滤器后，扫描器会自动调用sd_ble_gap_connect()函数发起连接。

代码清单：设置扫描匹配后自动连接

```
1. //自动连接设置为 false  
2. init_scan.connect_if_match = true;
```

2. 注册BLE事件监视者，关注连接相关的事件

连接发起后，应用程序需要知道执行的结果，从而进行相应的处理，如连接建立成功后设置指示灯、执行服务发现等。因此，应用程序需要注册BLE事件监视者接收BLE事件中连接相关的事件（3个事件）并处理。

- BLE_GAP_EVT_CONNECTED：连接建立事件，设置指示灯和执行服务发现（详见服务发现章节）。
- BLE_GAP_EVT_DISCONNECTED：连接断开事件，重启扫描（设置指示灯）。
- BLE_GAP_EVT_TIMEOUT(BLE_GAP_TIMEOUT_SRC_CONN)：建立连接超时事件。

1) 协议栈初始化函数 ble_stack_init()中注册 BLE 事件监视者，代码如下。

代码清单：BLE 协议栈初始化函数中注册 BLE 事件监视者

```
1. //注册 BLE 事件回调函数
2. NRF_SDH_BLE_OBSERVER(m_ble_observer, APP_BLE_OBSERVER_PRIO, ble_evt_handler,
3. NULL);
```

2) 事件处理函数 ble_evt_handler()中加入对连接相关的 3 个事件的处理，代码清单如下。

代码清单：BLE 事件处理函数

```
1. static void ble_evt_handler(ble_evt_t const * p_ble_evt, void * p_context)
2. {
3.     ret_code_t err_code;
4.     ble_gap_evt_t const * p_gap_evt = &p_ble_evt->evt.gap_evt;
5.     //判断事件类型
6.     switch (p_ble_evt->header.evt_id)
7.     {
8.         case BLE_GAP_EVT_CONNECTED://连接建立事件
9.             //设置指示灯
10.            err_code = bsp_indication_set(BSP_INDICATE_CONNECTED);
11.            APP_ERROR_CHECK(err_code);
12.            break;
13.
14.         case BLE_GAP_EVT_DISCONNECTED://连接断开事件
15.             //重启扫描
16.             scan_start();
17.             //LOG 输出断开连接原因
18.             NRF_LOG_INFO("Disconnected. conn_handle: 0x%u, reason: 0x%u",
19.                         p_gap_evt->conn_handle,
20.                         p_gap_evt->params.disconnected.reason);
21.             break;
22.
23.         case BLE_GAP_EVT_TIMEOUT://连接超时事件
24.             if (p_gap_evt->params.timeout.src == BLE_GAP_TIMEOUT_SRC_CONN)
25.             {
26.                 //LOG 输出提示信息
27.                 NRF_LOG_INFO("Connection Request timed out.");
28.             }
29.             break;
30.
31.         default:
32.             break;
33.     }
34. }
```

4. 硬件连接

同“实验 3-1”。

5. 测试步骤

- 解压“…\5：开发指南（下册-主机）配套实验源码\”目录下的压缩文件“实验 4-1：建立连接”，将解压后得到的文件夹“ble_app_scan”拷贝到合适的目录，如“D\ nRF52840”。
- 启动 MDK5.27。
- 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_scan\project\mdk5”目录下的工程“ble_app_scan.uvproj”。
- 切换到协议栈 target，下载协议栈。
- 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。
- 点击下载按钮下载程序。
- 程序运行后，主机开发板上的指示灯 D1 闪烁指示正在扫描。
- 将作为从机的开发板烧写《开发指南下册-从机》的“实验 7-3：串口透传长包传输-判断 notify”。程序运行后，从机开发板上的指示灯 D1 闪烁指示正在广播。
- 这时可以观察到，主从机的指示灯 D1 均由闪烁变为常亮，指示连接建立成功。
- 使用 Dongle 抓包，可以很清楚地看到连接的建立，如下图所示。

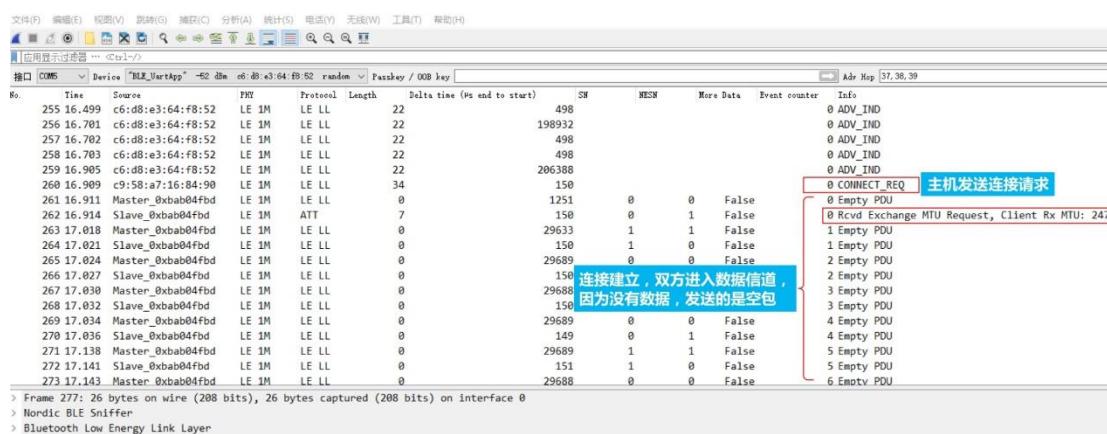


图 4-2：Dongle 捕获的连接请求包

- 说明：本章的代码中还没有处理 MTU 交换以及连接参数更新，因此，当连接建立后，会周期性的断开。

第五章：MTU 交换

1. 学习目的

- 掌握 MTU 交换的原理及执行流程。
- 掌握主机中 MTU 交换部分代码的编写。

2. MTU 交换原理

BLE 设备支持的 MTU（Maximum Transmission Unit：最大传输单元）可能各不相同，设备之间也不知道对方能支持的最大值，为了能让连接的设备之间以双方都支持的最大值传输数据，连接建立后，设备之间会进行一次 MTU 交换。注意：MTU 不是协商值，它是每个设备可以独立指定的信息参数。

1. 交换 MTU 请求

设备之间进行 MTU 交换时，只有客户端可以发起交换 MTU 请求。客户端使用交换 MTU 请求通知服务器客户端的最大接收 MTU 大小，并请求服务器以其最大接收 MTU 大小进行响应，双方由此知道对方支持的 MTU 大小的最大值，MTU 交换请求格式如下表所示。

表 5-1：交换 MTU 请求格式

参数	大小（字节数）	描述
属性操作码	1	交换 MTU 请求。
客户端 Rx MTU	2	客户端接收 MTU 大小。

交换 MTU 请求仅在客户端连接期间发送一次。

2. 交换 MTU 响应

客户端发送交换 MTU 响应以回复收到的交换 MTU 请求，其格式如下表所示。

表 5-2：交换 MTU 响应数据格式

参数	大小（字节数）	描述
属性操作码	1	交换 MTU 响应。
服务器 Rx MTU	2	服务器接收 MTU 大小。

服务器和客户端应将 ATT_MTU 设置为客户端 Rx MTU 和服务器 Rx MTU 的最小值，大小相同，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，应在服务器中应用此 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，应在客户端中应用此 ATT_MTU 值。

❖ 注：蓝牙内核协议 5.0 中明确规定：只有客户端可以发起交换 MTU 请求，但是 Nordic 的 SDK 中：无论是客户端还是服务器，GATT 库都会在建立连接时自动发出交换 MTU

请求，这一点是和蓝牙内核协议不符合的。

3. MTU 交换执行流程

下图是 MTU 交换的流程，该流程中以客户端最大 Rx MTU=80，服务器最大 Rx MTU=75 为例描述交换的流程和结果。

- 连接建立后，客户端发起交换 MTU 请求通知服务器自己最大接收 MTU 的大小是 80 个字节。
- 服务器接收到交换 MTU 请求后，协议栈向应用程序提交“BLE_GATTC_EVT_EXCHANGE_MTU_RSP(ATT_MTU=80)”事件，服务器由此知道双方支持的最大 MTU 值为 75。
- 服务器端应用程序调用协议栈 API 函数 sd_ble_gatts_exchange_mtu_reply()通知协议栈回复交换 MTU 请求，操作成功后协议栈发出交换 MTU 响应(服务器 Rx MTU=75)回复交换 MTU 请求，客户端接收到响应后，由此知道双方支持的最大 MTU 值为 75。
- 客户端和服务器将 MTU 值设置为 75（双方都支持的最大 MTU 值）。

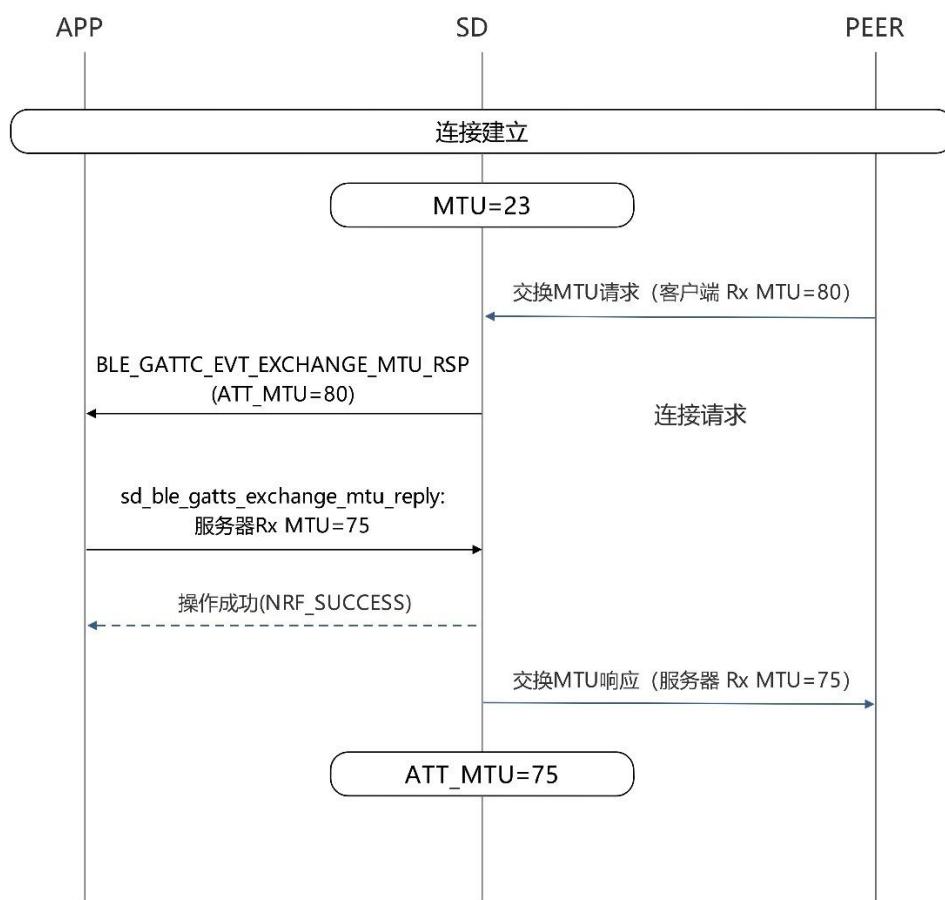


图 5-1：MTU 交换流程

4. 程序编写

❖ 注 1：本实验在“实验 4-1：建立连接”的基础上修改，连接的从机为《开发指南下册-

从机》中的“实验 7-3：串口透传长包传输-判断 notify”，本实验对应的实验源码是：“实验 5-1：加入 MTU 交换功能”。

MTU 交换属于属性协议，MTU 交换是由程序中的 GATT 模块处理的，因此程序中需要加入 GATT 部分的代码，程序编写步骤如下：

1. 引用 GATT 头文件

“main.c”文件中需要使用 GATT 库，因此需要引用 GATT 头文件。

代码清单：引用 GATT 头文件

1. //GGATT 需要引用的头文件

2. #include "nrf_ble_gatt.h"

2. 实例化一个 GATT 模块

GATT 使用时需要先实例化，GATT 使用宏“NRF_BLE_GATT_DEF”实例化，实例化后会为 GATT 实例分配静态内存和注册事件回调函数，本例中我们实例化一个名称为 m_gatt 的 GATT 实例，代码清单如下。

代码清单：定义名称为 m_gatt 的 GATT 模块实例

1. NRF_BLE_GATT_DEF(m_gatt);

NRF_BLE_GATT_DEF 是一个带参数的宏，输入参数“_name”即为实例化的 GATT 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

```
1. #define NRF_BLE_GATT_DEF(_name) \
2. static nrf_ble_gatt_t _name; \
3. NRF_SDH_BLE_OBSERVER(_name ## _obs, \
4.                      NRF_BLE_GATT_BLE_OBSERVER_PRIO, \
5.                      nrf_ble_gatt_on_ble_evt, &_name)
```

- 1) 行 2：定义了 static 类型 gatt 结构体变量，为 gatt 结构体分配了内存，该语句执行后，gatt 结构体变量常驻内存，即实例化了一个名为“_name”的 GATT 实例，协议栈和应用程序均可通过该名称访问该 GATT 实例。
- 2) 行 3~行 5：注册了名称为“_name_obs”的 BLE 事件监视者，注册成功后，当协议栈有事件产生时，会通过“nrf_ble_gatt_on_ble_evt()事件回调函数”将 BLE 事件提交给 GATT 模块，即“nrf_ble_gatt_on_ble_evt()事件回调函数”是 GATT 模块真正接收和处理 BLE 事件的地方。该事件回调函数中处理 MY 交换，若交换成功，则向主应用程序提交事件“NRF_BLE_GATT_EVT_ATT_MTU_UPDATED”通知主应用程序“MTU 已更新”。

3. 初始化 GATT

实例化 GATT 后，还需要初始化 GATT 程序模块。初始化函数中调用 nrf_ble_gatt_init() 函数初始化 GATT 程序模块，同时注册事件处理函数 gatt_evt_handler() 用于接收 GATT 模块提交的事件。之后调用 nrf_ble_gatt_att_mtu_central_set() 函数设置中心设备 ATT_MTU 大小

(客户端 Rx MTU 的最大值)。

代码清单：初始化 GATT 程序模块，设置设备支持的 MTU 大小

```

1. void gatt_init(void)
2. {
3.     ret_code_t err_code;
4.     //初始化 GATT 程序模块
5.     err_code = nrf_ble_gatt_init(&m_gatt, gatt_evt_handler);
6.     APP_ERROR_CHECK(err_code);
7.     //设置 ATT MTU 的大小,这里设置的值为 247
8.     err_code = nrf_ble_gatt_att_mtu_central_set(&m_gatt, NRF_SDH_BLE_GATT_MAX_MTU
9.                                                 _SIZE);
10.    APP_ERROR_CHECK(err_code);
11. }
```

设置中心设备 ATT_MTU 大小用到了 nrf_ble_gatt_att_mtu_central_set()函数，该函数原型如下表所示。

表 5-3: nrf_ble_gatt_att_mtu_central_set()函数

格式说明符	描述	示例
函数原型	ret_code_t nrf_ble_gatt_att_mtu_central_set (nrf_ble_gatt_t * uint16_t)	p_gatt, desired_mtu
函数功能	设置中心设备 ATT_MTU 大小。	
参数	[in,out] p_gatt: 指向 GATT 结构体。 [in] desired_mtu: 请求 ATT_MTU 大小。	
返回值	NRF_SUCCESS: 操作成功。 NRF_ERROR_NULL: 输入参数 p_gatt 指向 NULL。 NRF_ERROR_INVALID_PARAM: 如果 desired_mtu 的大小大于 NRF_SDH_BLE_GATT_MAX_MTU_SIZE 或小于 BLE_GATT_ATT_MTU_DEFAULT。	

4. 主应用程序 GATT 事件处理函数

主应用程序初始化 GATT 时注册了事件回调函数 gatt_evt_handler()用于接收 GATT 模块提交的“NRF_BLE_GATT_EVT_ATT_MTU_UPDATED”(MTU 已更新)事件。在该事件函数中，我们用一个变量 m_ble_uarts_max_data_len 记录实际可发送的最大用户数据长度，该数据长度=MTU-操作码长度(1字节)-句柄长度(2字节)。记录该长度的用途是：当主机向从机发送数据时，可以根据该变量的长度判断发送的数据有没有超过一次发送数据的最大长度。

代码清单：主应用程序 GATT 事件处理函数

```

1. void gatt_evt_handler(nrf_ble_gatt_t * p_gatt, nrf_ble_gatt_evt_t const * p_evt)
2. {
3.     //如果是 MTU 交换事件
4.     if (p_evt->evt_id == NRF_BLE_GATT_EVT_ATT_MTU_UPDATED)
5.     {
6.         NRF_LOG_INFO("ATT MTU exchange completed.");
7.         //设置串口透传服务的有效数据长度 (MTU-opcode-handle)
8.         m_ble_uarts_max_data_len = p_evt->params.att_mtu_effective - OPCODE_LENGTH
9.                             - HANDLE_LENGTH;
10.        NRF_LOG_INFO("Ble UARTS max data length set to 0x%X(%d)", m_ble_uarts_max_
11.                      data_len, m_ble_uarts_max_data_len);
12.    }
13. }

```

5. 设置设备支持的最大 MTU 大小

最后，在“sdk_config.h”文件中配置设备支持的最大 MTU 大小，即设置NRF_SDH_BLE_GATT_MAX_MTU_SIZE 的值，本例中我们设置为 247，即主机支持的最大 MTU 长度为 247。

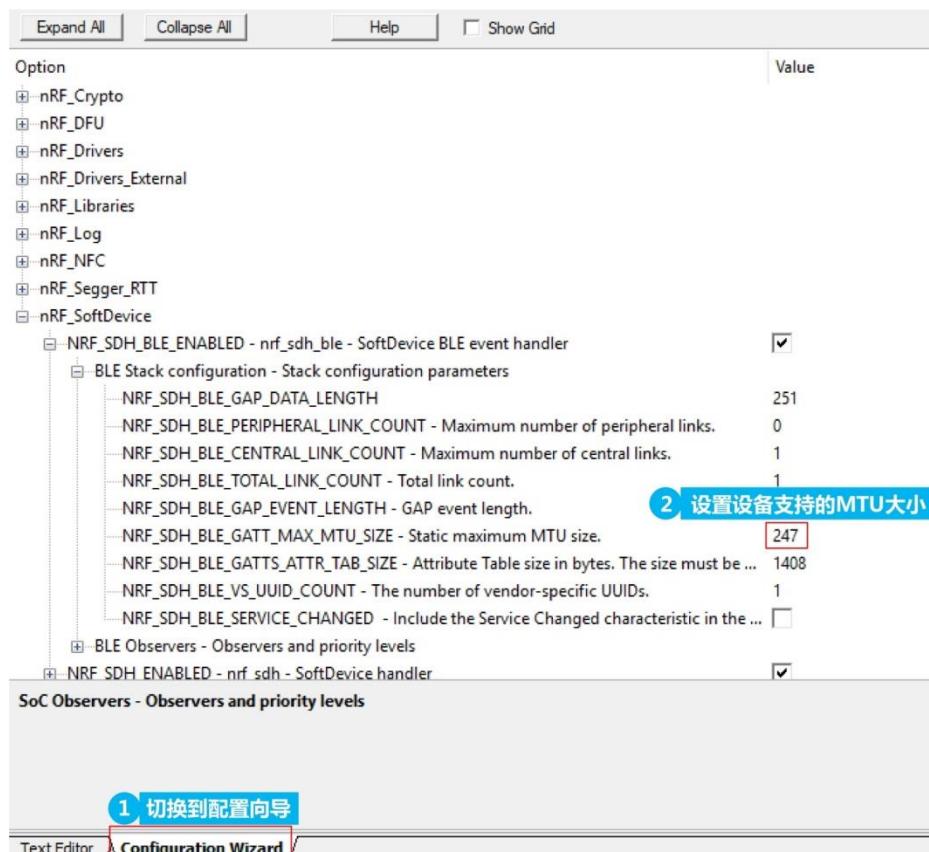


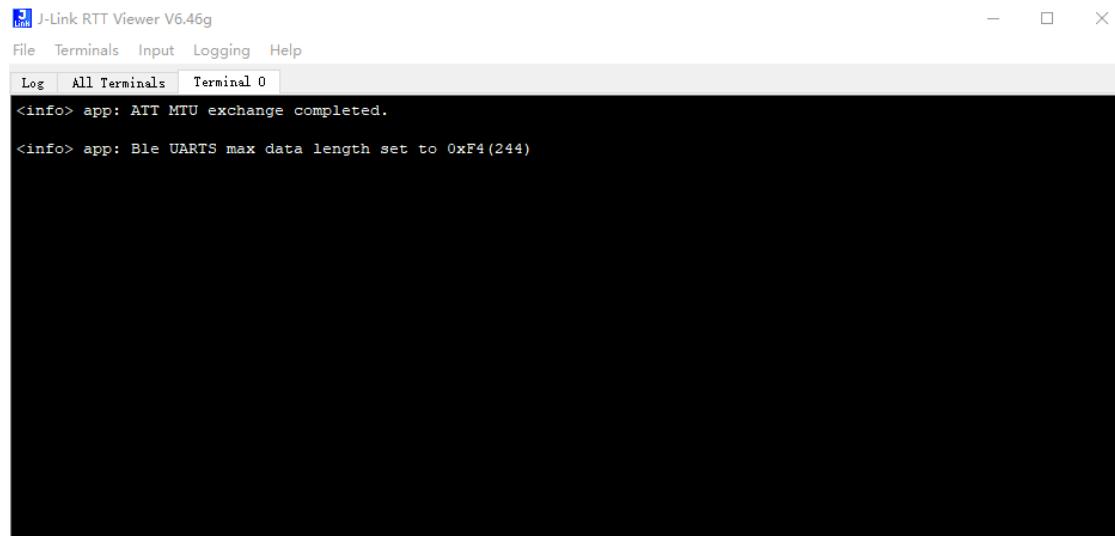
图 5-2：设置设备支持的最大 MTU 大小

5. 硬件连接

同“实验 3-1”。

6. 测试步骤

1. 解压“…\5: 开发指南（下册-主机）配套实验源码\”目录下的压缩文件“实验 5-1: 加入 MTU 交换功能”，将解压后得到的文件夹“ble_app_scan”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_scan\project\mdk5”目录下的工程“ble_app_scan.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。
7. 程序运行后，主机开发板上的指示灯 D1 闪烁指示正在扫描。
8. 将作为从机的开发板烧写《开发指南下册-从机》的“实验 7-3: 串口透传长包传输-判断 notify”。程序运行后，从机开发板上的指示灯 D1 闪烁指示正在广播。
9. 这时可以观察到，主从机的指示灯 D1 均由闪烁变为常亮，指示连接建立成功。
10. JLINK-RTT 中可以观察到 MTU 交换完成、有效数据长度设置为 244(=MTU 长度-1-2)，如下图所示。



```
J-Link RTT Viewer V6.46g
File Terminals Input Logging Help
Log All Terminals Terminal 0
<info> app: ATT MTU exchange completed.
<info> app: Ble UARTS max data length set to 0xF4(244)
```

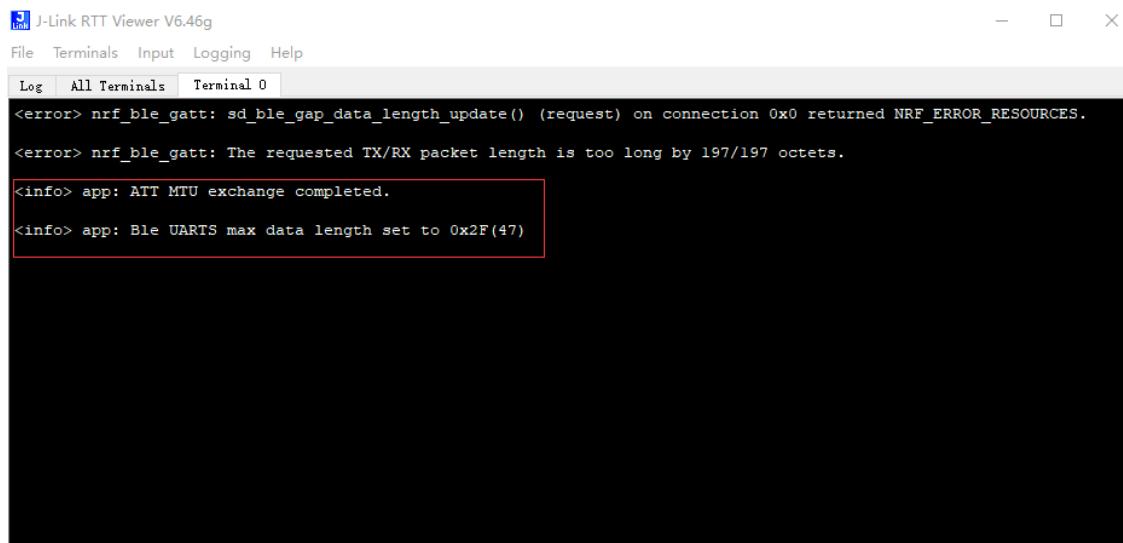
图 5-3: LOG 输出 MTU 交换信息

11. 如果想进一步观察 MTU 交换过程，可以使用 Dongle 抓包。如下图所示，加入 MTU 交换功能后，主从机在建立连接后进行了 MTU 交换，因为客户端和服务器同时支持的 Rx MTU 是 247，因此交换完成后，客户端和服务器的 Rx MTU 都会设置为 247。

No.	Time	Source	PMT	Protocol	Length	Delta time (μs end to start)	SN	NESN	More Data	Event Info
723	10.864	c9:58:a7:16:84:90	LE 1M	LE LL	12		150			0 SCAN_REQ
724	10.864	c9:2a:59:d5:b7:17	LE 1M	LE LL	24		150			0 SCAN_RSP
725	10.865	c9:2a:59:d5:b7:17	LE 1M	LE LL	22		301			0 ADV_IND
726	10.866	c9:2a:59:d5:b7:17	LE 1M	LE LL	22	48722				0 ADV_IND
727	10.867	c9:2a:59:d5:b7:17	LE 1M	LE LL	22		498			0 ADV_IND
728	10.868	c9:2a:59:d5:b7:17	LE 1M	LE LL	22		498			0 ADV_IND
729	10.869	c9:58:a7:16:84:90	LE 1M	LE LL	34		149			0 CONNECT_REQ
730	10.870	Master_0x38306d9f	LE 1M	ATT	7	1249	0	0	False	0 Sent Exchange MTU Request, Client Rx MTU: 247
731	10.871	Slave_0x38306d9f	LE 1M	ATT	7	150	0	1	False	0 Rcvd Exchange MTU Request, Client Rx MTU: 247
732	10.872	Master_0x38306d9f	LE 1M	LE LL	9	29576	1	1	True	1 Control Opcode: LL_LENGTH_REQ
733	10.873	Slave_0x38306d9f	LE 1M	LE LL	9	150	1	0	True	1 Control Opcode: LL_LENGTH_REQ
734	10.976	Master_0x38306d9f	LE 1M	LE LL	9	29545	0	0	True	2 Control Opcode: LL_LENGTH_RSP
735	10.978	Slave_0x38306d9f	LE 1M	ATT	7	150	0	1	False	2 Rcvd Exchange MTU Response, Server Rx MTU: 247
736	10.979	Master_0x38306d9f	LE 1M	ATT	7	29561	1	1	True	3 Sent Exchange MTU Response, Server Rx MTU: 247
737	10.980	Slave_0x38306d9f	LE 1M	LE LL	9	151	1	0	False	3 Control Opcode: LL_LENGTH_RSP
738	10.981	Master_0x38306d9f	LE 1M	ATT	27	29560	0	0	False	4 Sent Find By Type Value Request, GATT Primary Se
739	10.982	Slave_0x38306d9f	LE 1M	LE LL	0	150	0	1	False	4 Empty PDU
740	10.983	Master_0x38306d9f	LE 1M	LE LL	0	29473	1	1	False	5 Empty PDU
741	10.984	Slave_0x38306d9f	LE 1M	ATT	9	150	1	0	False	5 Rcvd Find_By_Type_Value_Response

图 5-4: Dongle 捕获的 MTU 交换包

12. 在“sdk_config.h”文件中修改 MTU 大小为 50，即设置 NRF_SDH_BLE_GATT_MAX_MTU_SIZE 的值为 50，重新编译后下载到开发板运行，这时在 JLINK-RTT 中可以观察到 MTU 交换完成后有效数据长度设置为 47 (=MTU 长度-1-2)，如下图所示。



```
J-Link RTT Viewer V6.46g
File Terminals Input Logging Help
Log All Terminals Terminal 0
<error> nrf_ble_gatt: sd_ble_gap_data_length_update() (request) on connection 0x0 returned NRF_ERROR_RESOURCES.
<error> nrf_ble_gatt: The requested TX/RX packet length is too long by 197/197 octets.
<info> app: ATT MTU exchange completed.
<info> app: Ble UARTS max data length set to 0x2F(47)
```

图 5-5: LOG 输出 MTU 交换信息

说明: 本章的代码中还没有处理连接参数更新，因此，当连接建立后，会周期性的断开。

第六章：PHY 更新

1. 学习目的

- 掌握 PHY 更新的原理及执行流程。
- 掌握主机中 PHY 更新部分代码的编写。

2. PHY 概述

PHY 更新是蓝牙 5.0 新增的内容，在 5.0 之前，BLE 的符号速率只有一种：LE 1M。到了 5.0，符号速率增加了 2 种，即 5.0 共有如下表所示的三种符号速率：

表 6-1: 5.0 PHY、调制方式和编码方案

PHY	调制方式	编码方式		数据速率
		接入包头	载荷	
LE 1M	1 Msym/s 调制	Uncoded	Uncoded	1 Mb/s
LE 2M	2 Msym/s 调制	Uncoded	Uncoded	2 Mb/s
LE Coded	1 Msym/s 调制	S=8	S=8	125 kb/s
			S=2	500 kb/s

对上表中的 3 种 PHY 归纳如下：

- LE 1M：蓝牙 4.x 中使用的 PHY，未编码，每个数据位使用 1 个符号表示。
- LE 2M：蓝牙 5.0 新增的 PHY，未编码，每个数据位使用 1 个符号表示。在不改变数据包类型的情况下，将原来的包传输所需要的时间缩短为原来的一半，由此提高了数据吞吐量。LE 2M 也就是我们常说的 5.0 的高速模式使用的 PHY。
- LE Coded：蓝牙 5.0 新增的 PHY。仅支持 1M/s 符号速率，使用前向纠错编码（FEC）。可以使用两种编码方案：S = 2，2 个符号表示 1 比特，因此支持 500kb / s 的比特率；S = 8，8 个符号表示 1 比特，因此支持 125kb 的比特率/秒。LE Coded 使用 FEC 提高了对错误的容忍能力，因此提高了接收机的接收灵敏度，他在不改变发射功率的情况下实现了更远的传播距离，可以认为他是通过“牺牲”数据速率换取了传输距离的扩展。LE Coded 也就是我们常说的 5.0 的远距模式使用的 PHY。

需要注意的是，蓝牙内核协议 5.0 规定：LE 1M 是 BLE 设备强制支持的，而 LE 2M 和 LE Coded 是可选的，因此，一个 BLE 设备宣传支持 5.0 BLE，并不表示他一定支持 LE 2M 和 LE Coded。比如手机，虽然支持 5.0，但是大多数手机都是不支持 LE Coded 的。

◆ 说明：nRF52840 支持“表 6-1”种描述的三种 PHY。

3. PHY 更新流程

PHY 更新规程用于更改发送 PHY 或接收 PHY，或同时更改发送和接收 PHY。PHY 更新是在建立连接后进行的，设备进入连接状态后，主设备可以发起 PHY 更新，从设备也可以发起 PHY 更新。

为了方便阅读本节描述的 PHY 更新流程，我们先要了解一下链路层 PHY 相关的 PDU 格式。

1. PHY 请求和响应

主机和从机均可发送 PHY 请求，即 LL_PHY_REQ。若主机发送 PHY 请求，从机需要发送响应，即 LL_PHY_RSP，主机接收到响应后发送 PHY 更新指示，即 LL_PHY_UPDATE_IND。若从机发送 PHY 请求，则主机发送 PHY 更新指示，而不会发送响应，使用时要注意到这一区别。LL_PHY_REQ 和 LL_PHY_RSP 格式如下表所示。

表 6-2: LL_PHY_REQ 和 LL_PHY_RSP 格式

CtrData	
TX_PHYS (1 字节)	RX_PHYS (1 字节)

LL_PHY_REQ 和 LL_PHY_RSP 由 2 个字段组成：

- TX_PHYS: 应设置为 PHY 请求或响应发送方首选的发送 PHY。
- RX_PHYS: 应设置为 PHY 请求或响应发送方首选的接收 PHY。

每个字段均有下表所示的 8 位组成，每个字段至少要有一个位设置为 1。每个字段可以设置多个首选 PHY，如可以同时设置位 0 和位 1 为“1”，即发送方首选 PHY 为：LE_1M 和 LE_2M。

表 6-3: PHY 字段位的定义

位	描述
0	发送方首选 LE 1M PHY (可能还有其他 PHY)。
1	发送方首选 LE 2M PHY (可能还有其他 PHY)。
2	发送方首选 LE Coded PHY(可能还有其他 PHY)。
3~7	保留为将来使用

2. PHY 更新指示

PHY 更新指示，即 LL_PHY_UPDATE_IND，其数据格式如下表所示。

表 6-4: LL_PHY_UPDATE_IND 格式

CtrData		
M_TO_S_PHY (1 字节)	S_TO_M_PHY (1 字节)	Instant (1 字节)

LL_PHY_UPDATE_IND 由 3 个字段组成：

- M_TO_S_PHY: 指示将用于从主设备发送到从设备的数据包的 PHY。
- S_TO_M_PHY: 指示将用于从从设备发送到主设备的数据包的 PHY。
- Instant: 瞬时，用于指示设备在经过多少个连接事件后使用 M_TO_S_PHY 和

S_TO_M_PHY 指示的 PHY。主机允许的 Instant 最小值应不小于 6 个连接事件。(因为 BLE 没有时钟，因此 LL_PHY_UPDATE_IND 发出后，双方为了保持同步修改 PHY，只能依靠对连接事件的计数来确定何时修改 PHY，Instant 指示了对连接间隔计数的数值)。

M_TO_S_PHY 和 S_TO_M_PHY 字段均有下表所示的 8 个位组成，如果 M_TO_S_PHY 和 S_TO_M_PHY 字段均为零，毫无疑问，Instant 字段将失去意义，此时，Instant 字段保留供将来使用。

表 6-5: M_TO_S_PHY 和 S_TO_M_PHY 字段位的定义

位	描述
0	发送方首选 LE 1M PHY。
1	发送方首选 LE 2M PHY。
2	发送方首选 LE Coded PHY。
3~7	保留为将来使用

了解了 PHY 的 PDU，接下来我们来分析一下 Nordic SDK 对 PHY 更新的处理流程。根据 PHY 更新发起的角色、设备首选的 PHY 及最终的执行结果，主机的 PHY 更新有以下 6 种情况。

1. 第 1 种情况：从机发起了 PHY 更新规程，执行后 PHY 不改变。

当从机发起 PHY 更新规程时，它会发送 LL_PHY_REQ PDU (从机首选 PHY 为: tx_phys=2Mbit, rx_phys=2Mbit)。主机协议栈接收到请求后向应用程序提交 BLE_GAP_EVT_PHY_UPDATE_REQUEST 事件，应用程序在该事件中调用协议栈 API 函数 sd_ble_gap_phy_update() (主机首选 PHY 为: tx_phys=1Mbit, rx_phys=1Mbit) 向协议栈发出“PHY 更新指示”的指令，之后协议栈发出 PHY 更新指示，指示应设置的 TX PHY 和 RX PHY 以及在多少个连接事件后设置。因为主机只支持 tx_phys=1Mbit, rx_phys=1Mbit，因此，规程执行完成后，PHY 没有改变，TX PHY 和 RX PHY 仍为 1Mbit。

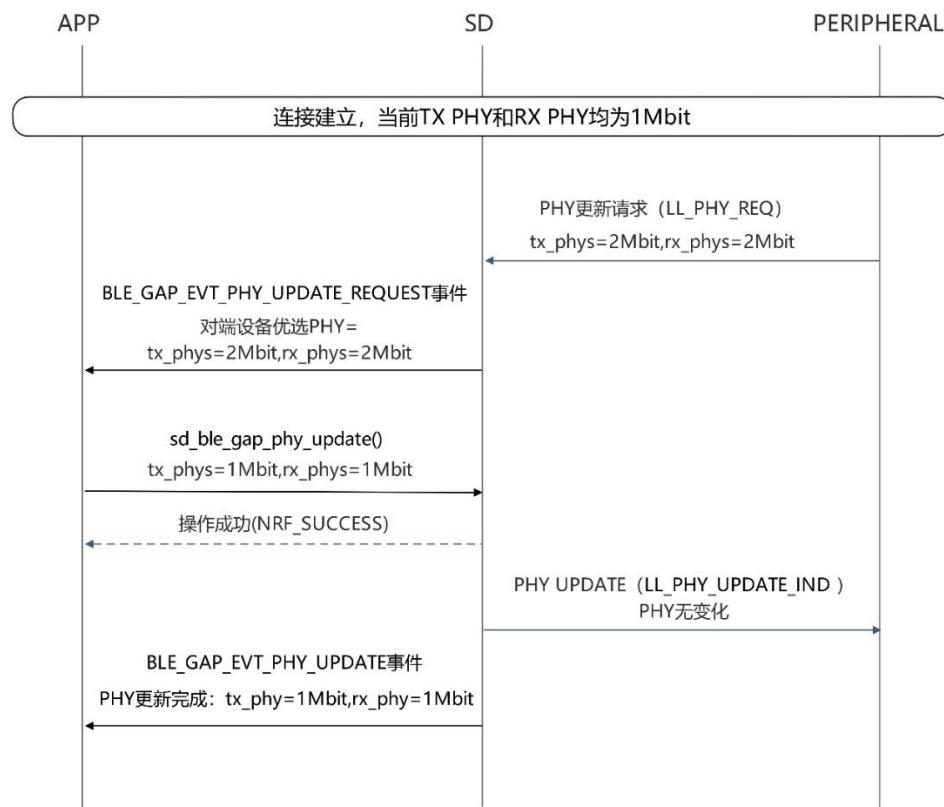


图 6-1：从机发起 PHY 更新规程，执行完成后 PHY 不改变

2. 第 2 种情况：从机发起 PHY 更新规程，从机和主机首选 PHY 相同。

从机发起 PHY 更新规程，从机首选的 PHY 是：tx_phys=2Mbit, rx_phys=2Mbit.。主机首选的 PHY 也是：tx_phys=2Mbit, rx_phys=2Mbit.。执行后，M_TO_S_PHY=2Mbit, S_TO_M_PHY=2Mbit。

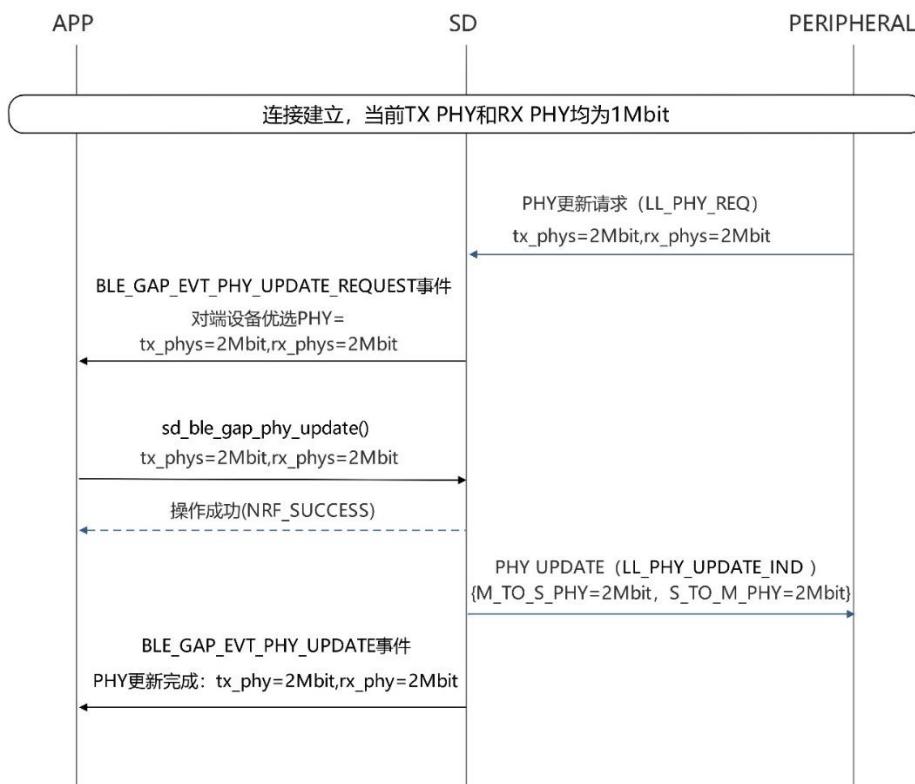


图 6-2: 从机发起 PHY 更新规程, 执行完成后 PHY 为 2Mbit

3. 第 3 种情况: 主机发起 PHY 更新规程, 从机不支持 PHY 更新

主机发起 PHY 更新规程, 主机首选 PHY: tx_phys=ALL PHY, rx_phys= ALL PHY (ALL PHY 表示 3 种 PHY: LE 1M, LE 2M, LE Coded 主机都可以支持)。主机发送请求后, 从机返回 “LL_UNKNOWN_RSP”, 这种情况, 表示从机不支持 PHY 更新。



图 6-3: 主机发起 PHY 更新规程, 对端设备不支持

4. 第 4 种情况: 主机发起 PHY 更新规程, 主从机均支持 ALL PHY。

主从机首选 PHY 均为 ALL PHY 时, 执行后 M_TO_S_PHY=2Mbit, S_TO_M_PHY=2Mbit, 可以看到, 3 种 PHY 都支持的情况下, PHY 更新的结果是选择 2Mbit。

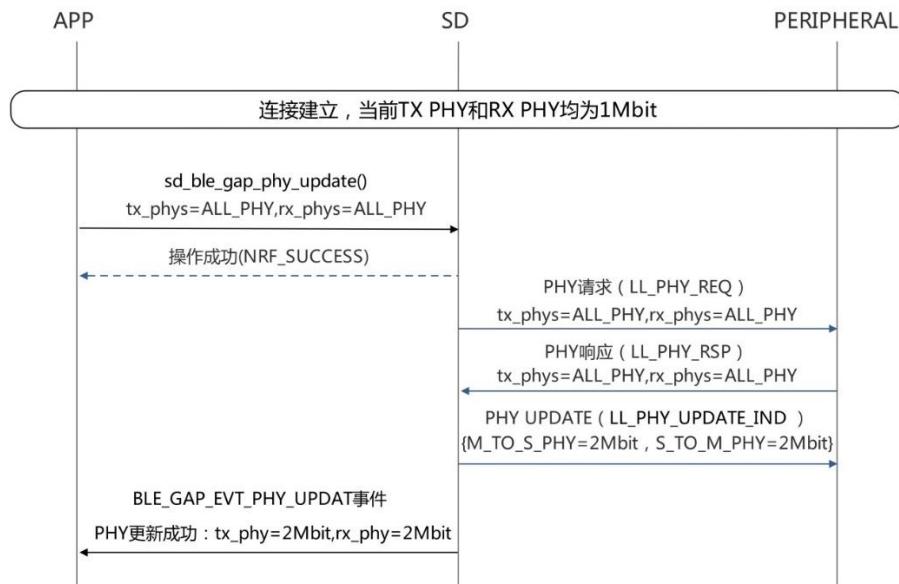


图 6-4: 主机发起 PHY 更新规程, 执行完成后 PHY 为 2Mbit

5. 第 5 种情况: 从机发起, `tx_phys` 和 `rx_phys` 不对称。

从机发起 PHY 更新规程, 从机首选 PHY 不对称, 如下图所示, 执行后 `M_TO_S_PHY`
=1Mbit, `S_TO_M_PHY`=2Mbit。

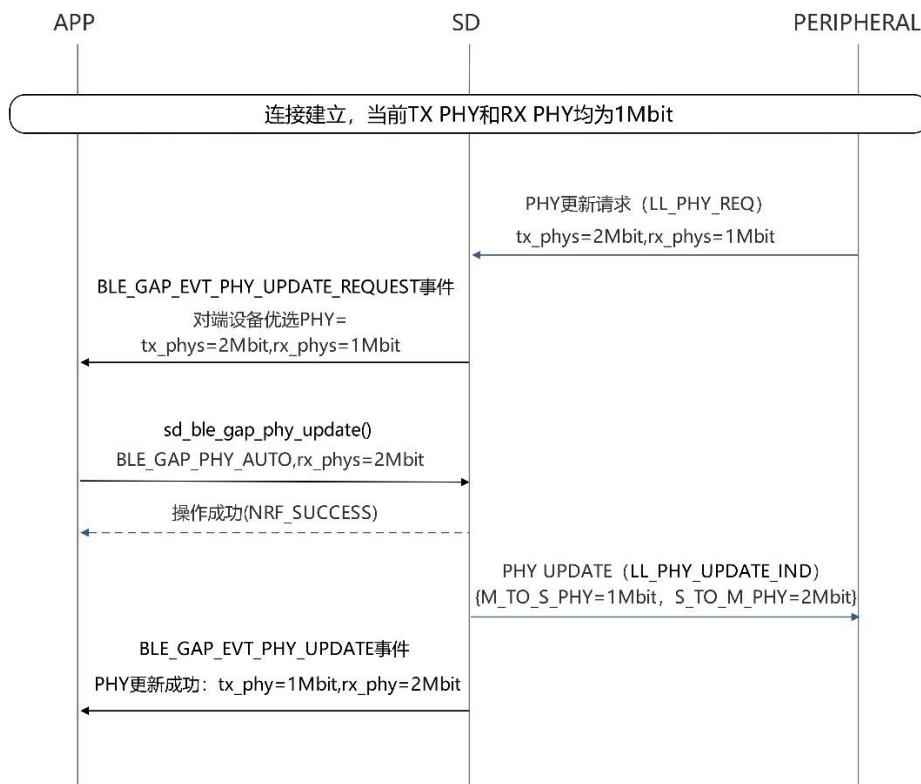


图 6-5: 从机发起, `tx_phys` 和 `rx_phys` 不对称

6. 第 6 种情况: 主机发起 PHY 更新规程, 从机响应时: `tx_phys` 和 `rx_phys` 等于 0。执行结果: 无效的 LMP 参数。

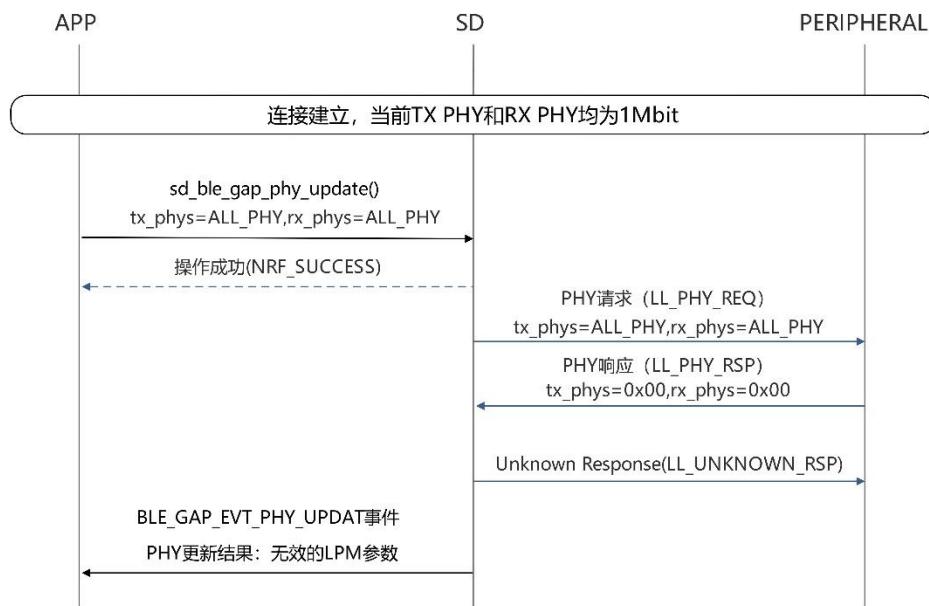


图 6-6: LMP 参数无效

4. 程序编写

◆ 注：本实验在“实验 5-1：加入 MTU 交换功能”的基础上修改，连接的从机为《开发指南下册-从机》中的“实验 7-3：串口透传长包传输-判断 notify”，本实验对应的实验源码是：“实验 6-1：加入 PHY 更新”。

对于 PHY 更新，发起或者响应使用的 API 函数都是 `sd_ble_gap_phy_update()`，该函数原型如下表所示。

表 6-6: `sd_ble_gap_phy_update()` 函数

格式说明符	描述	示例
函数原型	<pre>uint32_t sd_ble_gap_phy_update (uint16_t conn_handle, ble_gap_phys_t const * p_gap_phys)</pre>	
函数功能	<p>该函数用于发起或者响应 PHY 更新规程，执行成功后，总是产生 <code>BLE_GAP_EVT_PHY_UPDATE</code> 事件。</p> <ul style="list-style-type: none"> 如果该函数用于发起 PHY 更新规程，并且 <code>tx_phys</code> 和 <code>rx_phys</code> 中提供的唯一选项等于各个方向上当前活动的 PHY，链路层将不会发起 PHY 更新规程。 如果 <code>tx_phys</code> 和 <code>rx_phys</code> 设置为 <code>BLE_GAP_PHY_AUTO</code>，协议栈将根据对端设备的 PHY 首选项和本地链路配置选择 PHY。在这种情况下，PHY 更新过程将产生一个 phy 组合，该组合符合使用 <code>sd_ble_cfg_set()</code> 配置的时间约束和当前链路层数据长度。 当作为中心设备时，SoftDevice 将在每个方向上选择最快的公共 PHY 	

	<p>Y。</p> <ul style="list-style-type: none"> 如果对端设备不支持 PHY 更新过程，则生成的 BLE_GAP_EVT_PHY_UPDATE 事件的状态将设置为 BLE_HCI_UNSUPPORTED_REMOTE_FEATURE。 如果 PHY 更新规程由于程序冲突而被对等方拒绝，则状态将为 LE_HCI_STATUS_CODE_LMP_ERROR_TRANSACTION_COLLISION 或 BLE_HCI_DIFFERENT_TRANSACTION_COLLISION。如果对端设备使用无效参数响应 PHY 更新过程，则状态将为 BLE_HCI_STATUS_CODE_INVALID_LMP_PARAMETERS。如果对端设备由于其他原因拒绝了 PHY 规程，则状态将包含对等体指定的原因。
参数	<p>[in,out] <code>conn_handle</code>: 连接句柄，指示请求 PHY 更新的连接。 [in] <code>p_gap_phys</code>: 指向 PHY 结构体。</p>
返回值	<p>NRF_SUCCESS: 成功请求 PHY 更新。 NRF_ERROR_INVALID_ADDR: 提供了无效的指针。 BLE_ERROR_INVALID_CONN_HANDLE: 连接句柄无效。 NRF_ERROR_INVALID_PARAM: 提供的参数无效。 NRF_ERROR_INVALID_STATE: 没有已建立的连接。 NRF_ERROR_RESOURCES: 为此连接配置的连接事件长度不足以发送 PHY 更新包。 NRF_ERROR_BUSY: 该规程已在进行中或当前不允许，处理挂起事件并等待挂起的过程完成并重试。</p>

主机可以发起 PHY 更新，也可以不主动发起，仅响应从机的 PHY 请求，编程的时候根据自身需求决定使用哪一种方式或者同时具备这两种。

1. 定义主机首选 PHY

程序中，我们首先要定义主机的首选 PHY，首选 PHY 使用结构体 `ble_gap_phys_t` 定义，其声明如下。

代码清单：首选 PHY 结构体声明

```
1. typedef struct
2. {
3.     uint8_t tx_phys;      //首选发送 PHYs
4.     uint8_t rx_phys;      //首选接收 PHYs
5. } ble_gap_phys_t;
```

首选发送 PHYs 和首选接收 PHYs 可以设置为：BLE_GAP_PHY_1MBPS、BLE_GAP_PHY_2MBPS、BLE_GAP_PHY_CODED 或他们的组合如“BLE_GAP_PHY_1MBPS | BLE_GAP_PHY_2MBPS”，也可以设置为 BLE_GAP_PHY_AUTO，由协议栈选择 PHY。本例中我们定义的主机首选 PHY 如下。

代码清单：主机首选 PHY

```

1. ble_gap_phys_t const phys =
2. {
3.     .rx_phys = BLE_GAP_PHY_2MBPS, //首选发送 PHYs 为 2Mbit
4.     .tx_phys = BLE_GAP_PHY_2MBPS, //首选接收 PHYs 为 2Mbit
5. };

```

2. 主机不主动发起，仅响应从机的 PHY 请求时的处理

对于主机来说，协议栈接到从机的 PHY 请求后，会产生“BLE_GAP_EVT_PHY_UPDATE_REQUEST”事件，因此，我们只需在 BLE 事件处理函数 ble_evt_handler()中加入对该事件的处理即可，即在该事件中调用 sd_ble_gap_phy_update()函数响应从机 PHY 请求。

代码清单：PHY 请求事件

```

1. case BLE_GAP_EVT_PHY_UPDATE_REQUEST:
2. {
3.     NRF_LOG_DEBUG("PHY update request.");
4.     //响应从机 PHY 请求
5.     err_code = sd_ble_gap_phy_update(p_ble_evt->evt.gap_evt.conn_handle, &phys);
6.     //检查函数返回值
7.     APP_ERROR_CHECK(err_code);
8. } break;

```

3. 主机发起 PHY 请求

PHY 更新规程必须在连接建立后才能发起，因此，本例中在连接建立事件“BLE_GAP_EVT_CONNECTED”中调用 sd_ble_gap_phy_update()函数发送 PHY 请求，代码清单如下。

代码清单：主机发起 PHY 请求

```

1. case BLE_GAP_EVT_CONNECTED:// 连接建立事件
2.     //设置指示灯
3.     err_code = bsp_indication_set(BSP_INDICATE_CONNECTED);
4.     APP_ERROR_CHECK(err_code);
5.     //发送 PHY 请求
6.     err_code = sd_ble_gap_phy_update(p_gap_evt->conn_handle, &phys);
7.     APP_ERROR_CHECK(err_code);
8. break;

```

5. 硬件连接

同“实验 3-1”。

6. 测试步骤

- 解压“…\5：开发指南（下册-主机）配套实验源码\”目录下的压缩文件“实验 6-1：加入 PHY 更新”，将解压后得到的文件夹“ble_app_scan”拷贝到合适的目录，如“D\ nRF52840”。
- 启动 MDK5.27。
- 在 MDK5 中执行“Project→Open Project”打开“…\ble_app_scan\project\mdk5”目录下的工程“ble_app_scan.uvproj”。
- 切换到协议栈 target，下载协议栈。
- 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。
- 点击下载按钮下载程序。
- 程序运行后，主机开发板上的指示灯 D1 闪烁指示正在扫描。
- 将作为从机的开发板烧写《开发指南下册-从机》的“实验 7-3：串口透传长包传输-判断 notify”。程序运行后，从机开发板上的指示灯 D1 闪烁指示正在广播。
- 这时可以观察到，主从机的指示灯 D1 均由闪烁变为常亮，指示连接建立成功。
- 使用 Dongle 抓包观察 PHY 更新过程，如下图所示，连接建立后主机发起 PHY 更新请求，从机响应 PHY 请求，之后主机发送 PHY 更新指示，主从机按照 PHY 更新指示中的“M_TO_S_PHY”和“S_TO_M_PHY”字段设置 PHY。

No.	Time	Source	PHY	Protocol	Le	Delta time (μs)	NESN	More Data	Event	Info
138	9.181	Slave_0xb9a1e715	LE 1M	ATT	7	151	0	1	False	0 Rcvd Exchange MTU Request, Client Rx MTU: 247
139	9.182	Master_0xb9a1e715	LE 1M	LE LL	9	29576	1	1	True	1 Control Opcode: LL_LENGTH_REQ
140	9.183	Slave_0xb9a1e715	LE 1M	ATT	7	150	1	0	False	1 Rcvd Exchange MTU Response, Server Rx MTU: 247
141	9.288	Master_0xb9a1e715	LE 1M	ATT	7	29561	0	0	False	2 Sent Exchange MTU Response, Server Rx MTU: 247
142	9.290	Slave_0xb9a1e715	LE 1M	LE LL	0	151	0	1	False	2 Empty PDU
143	9.291	Master_0xb9a1e715	LE 1M	LE LL	0	29632	1	1	False	3 Empty PDU
144	9.292	Slave_0xb9a1e715	LE 1M	LE LL	9	151	1	0	False	3 Control Opcode: LL_LENGTH_RSP
145	9.292	Master_0xb9a1e715	LE 1M	LE LL	3	29615	0	0	False	4 Control Opcode: LL_PHY_REQ
146	9.293	Slave_0xb9a1e715	LE 1M	LE LL	0	151	0	1	False	4 Empty PDU
147	9.294	Master_0xb9a1e715	LE 1M	LE LL	0	29663	1	1	False	5 Empty PDU
148	9.295	Slave_0xb9a1e715	LE 1M	LE LL	0	150	1	0	False	5 Empty PDU
149	9.397	Master_0xb9a1e715	LE 1M	LE LL	0	29689	0	0	False	6 Empty PDU
150	9.400	Slave_0xb9a1e715	LE 1M	LE LL	3	150	0	1	False	6 Control Opcode: LL_PHY_RSP
151	9.404	Master_0xb9a1e715	LE 1M	LE LL	5	29665	1	1	False	7 Control Opcode: LL_PHY_UPDATE_IND
152	9.412	Slave_0xb9a1e715	LE 1M	LE LL	0	150	1	0	False	7 Empty PDU
153	9.413	Master_0xb9a1e715	LE 1M	LE LL	0	29649	0	0	False	8 Empty PDU
154	9.414	Slave_0xb9a1e715	LE 1M	LE LL	0	150	0	1	False	8 Empty PDU
155	9.415	Master_0xb9a1e715	LE 1M	LE LL	0	29689	1	1	False	9 Empty PDU
156	9.416	Slave_0xb9a1e715	LE 1M	LE LL	0	150	1	0	False	9 Empty PDU
157	9.518	Master_0xb9a1e715	LE 1M	LE LL	0	29688	0	0	False	10 Empty PDU
158	9.519	Slave_0xb9a1e715	LE 1M	LE LL	0	150	0	1	False	10 Empty PDU
159	9.520	Master_0xb9a1e715	LE 1M	LE LL	0	29690	1	1	False	11 Empty PDU
160	9.521	Slave_0xb9a1e715	LE 1M	LE LL	0	150	1	0	False	11 Empty PDU
161	9.521	Master_0xb9a1e715	LE 1M	LE LL	0	29688	0	0	False	12 Empty PDU
162	9.522	Slave_0xb9a1e715	LE 1M	LE LL	0	150	0	1	False	12 Empty PDU
163	9.624	Master_0xb9a1e715	LE 2M	LE LL	0	29685	1	1	False	13 Empty PDU
164	9.627	Slave_0xb9a1e715	LE 2M	LE LL	0	150	1	0	False	13 Empty PDU
165	9.629	Master_0xb9a1e715	LE 2M	LE LL	0	29761	0	0	False	14 Empty PDU

图 6-7: PHY 更新过程

- 修改“phys. tx_phys”为 BLE_GAP_PHY_1MBPS，此时，tx 和 rx 的 PHY 不对称，编译后烧写到主机开发板运行。使用 Dongle 抓包观察 PHY 更新过程，如下图所示，可以看到 PHY 更新完成后：主机发送、从机接收的 PHY 设置为 1Mbit，主机接收、从机发

送的 PHY 设置为 2Mbit。

No.	Time	Source	PHY	Protocol	Le	Delta time (μ s)	NESN	More Data	Event	Info
46..	307.1..	C9:58:a7:16:84:90	LE 1M	LE LL	...	166	0	0	False	0 CONNECT_REQ
46..	307.1..	Master_0x1033c99d	LE 1M	ATT	7	1252	0	0	False	0 Sent Exchange MTU Request, Client Rx MTU: 247
46..	307.1..	Slave_0x1033c99d	LE 1M	ATT	7	151	0	1	False	0 Rcvd Exchange MTU Request, Client Rx MTU: 247
46..	307.1..	Master_0x1033c99d	LE 1M	LE LL	9	29576	1	1	True	1 Control Opcode: LL_LENGTH_REQ
46..	307.1..	Slave_0x1033c99d	LE 1M	ATT	7	151	1	0	False	1 Rcvd Exchange MTU Response, Server Rx MTU: 247
46..	307.2..	Master_0x1033c99d	LE 1M	ATT	7	29560	0	0	False	2 Sent Exchange MTU Response, Server Rx MTU: 247
46..	307.2..	Slave_0x1033c99d	LE 1M	LE LL	0	150	0	1	False	2 Empty PDU
46..	307.2..	Master_0x1033c99d	LE 1M	LE LL	0	29633	1	1	False	3 Empty PDU
46..	307.2..	Slave_0x1033c99d	LE 1M	LE LL	9	150	1	0	False	3 Control Opcode: LL_LENGTH_RSP
46..	307.2..	Master_0x1033c99d	LE 1M	LE LL	3	29617	0	0	False	4 Control Opcode: LL_PHY_REQ
47..	307.2..	Slave_0x1033c99d	LE 1M	LE LL	0	150	0	1	False	4 Empty PDU
47..	307.2..	Master_0x1033c99d	LE 1M	LE LL	0	29664	1	1	False	5 Empty PDU
47..	307.2..	Slave_0x1033c99d	LE 1M	LE LL	0	151	1	0	False	5 Empty PDU
47..	307.2..	Master_0x1033c99d	LE 1M	LE LL	0	29688	0	0	False	6 Empty PDU
47..	307.3..	Slave_0x1033c99d	LE 1M	LE LL	3	150	0	1	False	6 Control Opcode: LL_PHY_RSP
47..	307.3..	Master_0x1033c99d	LE 1M	LE LL	5	29664	1	1	False	7 Control Opcode: LL_PHY_UPDATE_IND
47..	307.3..	Slave_0x1033c99d	LE 1M	LE LL	0	151	1	0	False	7 Empty PDU
47..	307.3..	Master_0x1033c99d	LE 1M	LE LL	0	29648	0	0	False	8 Empty PDU
47..	307.3..	Slave_0x1033c99d	LE 1M	LE LL	0	150	0	1	False	8 Empty PDU
47..	307.3..	Master_0x1033c99d	LE 1M	LE LL	0	29689	1	1	False	9 Empty PDU
47..	307.3..	Slave_0x1033c99d	LE 1M	LE LL	0	150	1	0	False	9 Empty PDU
47..	307.4..	Master_0x1033c99d	LE 1M	LE LL	0	29689	0	0	False	10 Empty PDU
47..	307.4..	Slave_0x1033c99d	LE 1M	LE LL	0	150	0	1	False	10 Empty PDU
47..	307.4..	Master_0x1033c99d	LE 1M	LE LL	0	29689	1	1	False	11 Empty PDU
47..	307.4..	Slave_0x1033c99d	LE 1M	LE LL	0	150	1	0	False	11 Empty PDU
47..	307.4..	Master_0x1033c99d	LE 1M	LE LL	0	29688	0	0	False	12 Em. ④ 主机发送、从机接收的PHY设置为1Mbit
47..	307.4..	Slave_0x1033c99d	LE 1M	LE LL	0	150	0	1	False	12 Empty PDU
47..	307.5..	Master_0x1033c99d	LE 1M	LE LL	0	29690	1	1	False	13 Empty PDU
47..	307.5..	Slave_0x1033c99d	LE 2M	LE LL	0	146	1	0	False	13 Empty PDU
47..	307.5..	Master_0x1033c99d	LE 1M	LE LL	0	29728	0	0	False	14 Empty PDU

图 6-8: PHY 更新过程

说明: 本章的代码中还没有处理连接参数更新, 因此, 当连接建立后, 会周期性的断开。

第七章：连接参数更新

1. 学习目的

- 掌握连接参数更新的流程。
- 掌握连接参数更新部分代码的编写。

2. 连接参数更新流程

设备建立连接时，主机会向从机发送连接请求，连接请求中包含了连接参数，这个连接参数即是主、从机使用的初始连接参数。

当连接持续一段时间后，连接参数可能不再适合当前的服务使用，如设备之间不需要频繁交换数据时，设备希望减小连接间隔，从而降低功耗，这时，就要进行连接参数更新。

连接参数更新一般在建立连接后延迟一段时间进行，这是因为连接刚建立时，主机会执行服务发现等需要频繁交换数据包的操作，这时，若更新连接参数，会增加这些操作耗用的时间。另外，初始连接间隔一般较小，若此时更新连接参数，增加了连接间隔，同样会导致服务发现的过程耗时增加。

连接参数更新有 2 种方式，一种是“强制”方式的，一种是“协商”方式的。

1. “强制”方式的连接参数更新

主机发送连接参数更新指示“LL_CONNECTION_UPDATE_IND”，该指示中包含了新的连接参数以及瞬时（主、从机在计数多少个连接事件后开始使用新的连接参数，瞬时的值不能小于 6 个连接事件），从机只有两个选择：要么接受并使用该连接参数，要么断开连接。

“强制”方式的连接参数更新流程如下图所示，这种方式只有主机可以发起，没有“协商”，从机只能接受或者断开连接。

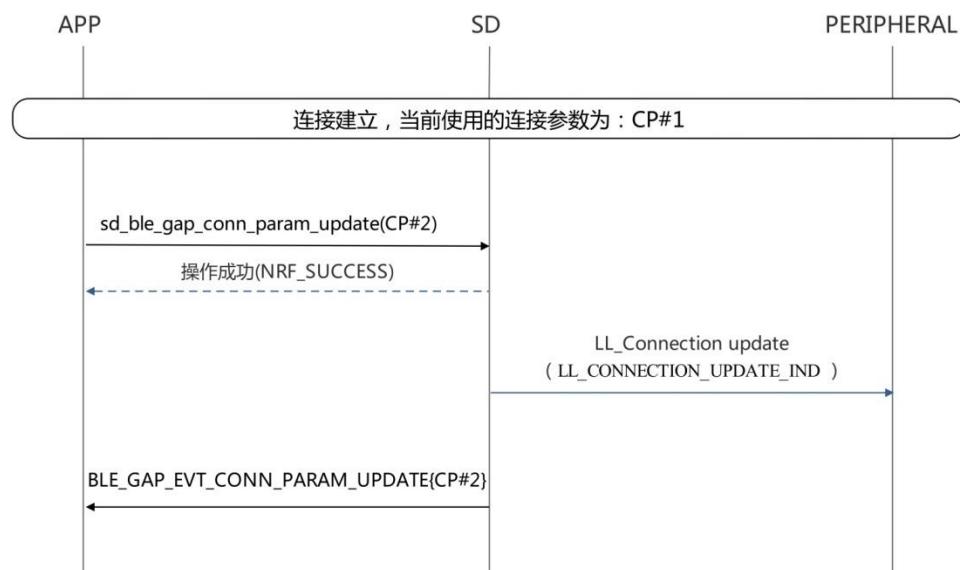


图 7-1：连接参数更新流程

LL_CONNECTION_UPDATE_IND 数据包格式如下表所示。

表 7-1: LL_CONNECTION_UPDATE_IND 格式

WinSize (1 字节)	WinOffset (2 字节)	Interval (2 字节)	Latency (2 字节)	Timeout (2 字节)	Instant (2 字节)
-------------------	---------------------	--------------------	-------------------	-------------------	-------------------

组成“LL_CONNECTION_UPDATE_IND”的6个字段意义如下：

- WinSize: 传输窗口大小。
- WinOffset: 传输窗口偏移量。
- Interval: 连接间隔。之前我们说到连接间隔时有最大值和最小值，而该命令是“强制”式的，所以只有一个确定的数值。连接间隔时长计算公式：Interval * 1.25 ms。
- Latency: 从机延迟。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。
- Timeout: 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。
- Instant: 瞬时。用于指示设备在经过多少个连接事件后开始使用新的连接参数，这和PHY更新种的瞬时意义是一样的。瞬时在连接参数更新中还有一个重要的作用，他解决了无线系统中一个重要的问题：报文重传，如果在更新命令后，出现报文重传，只要在瞬时之前重传成功，就不会有问题。

2. “协商”方式的连接参数更新

“协商”式的连接参数更新只能由从机发起，连接建立后，若从机希望修改连接参数，从机可以向主机发起连接参数更新请求，请求中包含了从机期望的连接参数。主机接收到从机的请求后，可以接受或拒绝从机的请求。

由此我们也可以看到，连接参数更新是由主机决定的，从机只能请求，而没有决定权。

看下图的流程，主从机当前使用的连接参数记为“CP#1”，从机发起连接参数更新请求（新的连接参数记为CP#2），主机接收到请求后，产生“BLE_GAP_EVT_CONN_PARAM_UPDATE_REQUEST”事件，主应用程序在该事件中接受或拒绝从机的更新请求。接受或拒绝请求都是通过调用sd_ble_gap_conn_param_update()函数实现的。

- 主机接受从机的连接参数更新请求：sd_ble_gap_conn_param_update(CP#2)。
- 主机拒绝从机的连接参数更新请求：sd_ble_gap_conn_param_update(NULL)。

Nordic SDK 的主机程序使用的都是“协商”式的连接参数更新，本例中我们同样实现的是这种方式。

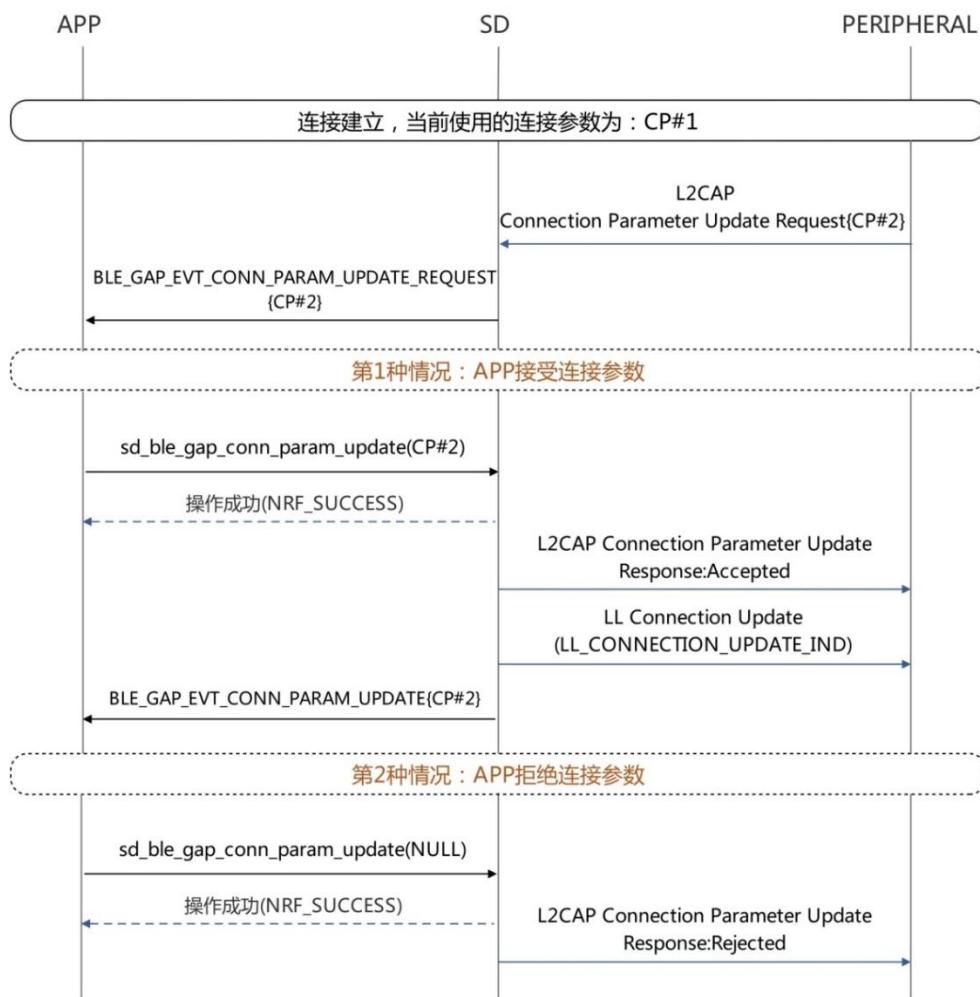


图 7-2: 连接参数更新流程

连接参数更新请求包格式如下图所示。

表 7-2: 连接参数更新请求包格式

Code=0x12	Identifier	Length	Interval Min	Interval Max	Slave Latency	Timeout Multiplier
1 字节	1 字节	1 字节	2 字节	2 字节	2 字节	2 字节

组成“LL_CONNECTION_UPDATE_IND”的 6 个字段意义如下：

- Interval Min: 连接参数最小值, 单位 1.25ms, 取值范围: 6 ~3200, Interval Min 应不大于 Interval Max。
- Interval Max: 连接参数最大值, 单位 1.25ms, 取值范围: 6 ~3200, Interval Max 应不小于 Interval Min。
- Slave Latency: 从机延迟, 取值范围: 0 到 ((connSupervisionTimeout / (connIntervalMax * 2)) -1), 并且 Slave Latency 字段应小于 500。
- Timeout Multiplier: 定义监督超时时间, 计算公式: 监督超时= Timeout Multiplier*10 ms, Timeout Multiplier 字段的值应在 10 到 3200 之间

连接参数响应包格式如下图所示。

表 7-3: 连接参数响应包格式

Code=0x13	Identifier	Length	Result
1 字节	1 字节	1 字节	2 字节

Result 字段指示对连接参数更新请求的响应，Result 值为 0x0000 表示主机已接受连接参数，而 0x0001 表示主机已拒绝连接参数

表 7-4: Result 字段定义

位	描述
0x0000	连接参数已接受。
0x0001	连接参数已拒绝。
其他值	保留为将来使用

3. 程序编写

❖ 注：本实验在“实验 6-1：加入 PHY 更新”的基础上修改，连接的从机为《开发指南下册-从机》中的“实验 7-3：串口透传长包传输-判断 notify”，本实验对应的实验源码是：“实验 7-1：加入连接参数更新”。

本例中，我们实现的是“协商”式的连接参数更新，主机端只需在“BLE_GAP_EVT_CONN_PARAM_UPDATE_REQUEST”事件中调用 sd_ble_gap_conn_param_update()函数应答从机的请求即可。sd_ble_gap_conn_param_update()函数原型如下表所示。

表 7-5: sd_ble_gap_conn_param_update()函数

格式说明符	描述	示例
函数原型	uint32_t sd_ble_gap_conn_param_update (uint16_t conn_handle, ble_gap_conn_params_t const * p_conn_params)	
函数功能	该函数用于更新连接参数。 • 中心设备角色中，这将启动链路层连接参数更新规程。外围设备角色中，这将发送相应的 L2CAP 请求并等待中心设备执行该规程。在这两种情况下，无论成功与否，都会通过 LE_GAP_EVT_CONN_PARAM_UPDATE 事件通知应用程序执行结果 • 该函数既可用作回复 BLE_GAP_EVT_CONN_PARAM_UPDATE_REQUEST 请求的中心，也可用作启动未请求的过程。	
参数	[in,out] conn_handle : 连接句柄。 [in] p_conn_params : 指向期望的连接参数，如果外围设备调用该函数时将其设置为 NULL，则将使用 GAP 服务的 PPCP 特征中的参数。如果中心设备设置为 NULL，表示拒绝外围设备的更新请求。	

返回值	<p>NRF_SUCCESS: 连接参数更新请求启动成功。</p> <p>NRF_ERROR_INVALID_ADDR: 提供了无效的指针。</p> <p>NRF_ERROR_INVALID_PARAM: 提供的参数无效。</p> <p>NRF_ERROR_INVALID_STATE: 没有已建立的连接。</p> <p>NRF_ERROR_RESOURCES: 为此连接配置的连接事件长度不足以发送 PHY 更新包。</p> <p>NRF_ERROR_BUSY: 该规程已在进行中，处理挂起事件并等待挂起的规程完成并重试。</p> <p>BLE_ERROR_INVALID_CONN_HANDLE: 连接句柄无效。</p> <p>NRF_ERROR_NO_MEM: 内存不足。</p>
-----	---

程序中，在 ble_evt_handler()函数中的“BLE_GAP_EVT_CONN_PARAM_UPDATE_REQUEST”事件下加入响应从机请求的代码，主机可以接受，也可以拒绝，实现的代码分别如下。

代码清单：主机接受从机的连接参数更新请求

```

1. case BLE_GAP_EVT_CONN_PARAM_UPDATE_REQUEST:
2. //接受连接参数请求
3. err_code = sd_ble_gap_conn_param_update(p_gap_evt->conn_handle,
4.                                         &p_gap_evt->params.conn_param_update_req
5.                                         .conn_params);
6. APP_ERROR_CHECK(err_code);
7. break;

```

代码清单：主机拒绝从机的连接参数更新请求

```

8. case BLE_GAP_EVT_CONN_PARAM_UPDATE_REQUEST:
9. //拒绝连接参数请求
10. err_code = sd_ble_gap_conn_param_update(p_gap_evt->conn_handle, NULL);
11. APP_ERROR_CHECK(err_code);
12. break;

```

4. 硬件连接

同“实验 3-1”。

5. 测试步骤

- 解压“…\5：开发指南（下册-主机）配套实验源码\”目录下的压缩文件“实验 7-1：加入连接参数更新”，将解压后得到的文件夹“ble_app_scan”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_scan\project\mdk5” 目录下的工程 “ble_app_scan.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。
 7. 程序运行后，主机开发板上的指示灯 D1 闪烁指示正在扫描。
 8. 将作为从机的开发板烧写《开发指南下册-从机》的“实验 7-3：串口透传长包传输-判断 notify”。程序运行后，从机开发板上的指示灯 D1 闪烁指示正在广播。
 9. 这时可以观察到，主从机的指示灯 D1 均由闪烁变为常亮，指示连接建立成功。
 10. 使用 Dongle 抓包观察连接参数更新过程，如下图所示，连接建立后从机约在 5 秒后发起连接参数更新请求(下图中的“Connection Parameter Request”), 主机接收到请求后，发送响应包接受连接参数更新请求(下图中的“Connection Parameter Response(Accepted)”), 之后，链路层发送连接参数更新指示 “LL_CONNECTION_UPDATE_IND” 确定连接参数以及瞬时。

No.	Time	Source	PHY	Protocol	Len	Delta time (μs)	NESN	More Data	Event or Info	Passkey / OOB key	Adv H
7374	251.955	Master_0x19e06535	LE 2M	LE LL	0	29760	1	1	False	163 Empty PDU	
7375	251.956	Slave_0x19e06535	LE 2M	LE LL	0	149	1	0	False	163 Empty PDU	
7376	251.956	Master_0x19e06535	LE 2M	LE LL	0	29762	0	0	False	164 Empty PDU	
7377	251.957	Slave_0x19e06535	LE 2M	LE LL	0	149	0	1	False	164 Empty PDU	
7378	251.958	Master_0x19e06535	LE 2M	LE LL	0	29762	1	1	False	165 Empty PDU	
7379	251.959	Slave_0x19e06535	LE 2M	LE LL	0	150	1	0	False	165 Empty PDU	
7380	251.960	Master_0x19e06535	LE 2M	LE LL	0	29760	0	0	False	166 Empty PDU	
7381	251.961	Slave_0x19e06535	LE 2M	LE LL	0	149	0	1	False	166 Empty PDU	
7382	251.963	Master_0x19e06535	LE 2M	LE LL	0	29761	1	1	False	167 Empty PDU	
7383	251.965	Slave_0x19e06535	LE 2M	L2CAP	16	150	1	0	False	167 Connection Parameter Update Request	
7384	251.966	Master_0x19e06535	LE 2M	L2CAP	10	29697	0	0	False	168 Connection Parameter Update Response (Accepted)	
7385	251.967	Slave_0x19e06535	LE 2M	LE LL	0	150	0	1	False	168 Empty PDU	
7386	252.069	Master_0x19e06535	LE 2M	LE LL	12	29720	1	1	False	169 Control Opcode: LL_CONNECTION_UPDATE_REQ	
7387	252.070	Slave_0x19e06535	LE 2M	LE LL	0	150	1	0	False	169 Empty PDU	

图 7-3 连接参数更新

说明：至此，例子代码已经实现了建立连接、MTU 交换、PHY 更新和连接参数更新，连接建立后，不会再周期性断开。但是因为没有实现服务发现和数据传输，因此，传输的都是空包。

第八章：实现串口透传主机

1. 学习目的

- 了解客户端发现对端服务器的服务、特征和描述符所用的属性协议以及发现的流程。
- 掌握程序中发现模块初始化、启动发现以及事件处理部分代码的编写（服务发现相关的协议流程实际都由 DB（数据库）发现模块完成了，应用程序只需初始化 DB 发现模块、启动服务发现和接收并处理服务发现的相关事件即可）。
- 掌握串口透传客户端代码的编写、数据的接收以及发送（串口透传主机的参考意义非常大，对于其他的主机程序，程序结构和流程都是类似的）。

2. 发现服务

客户端和设备建立连接后，客户端需要去发现设备公开的首要服务，从而使用服务，也就是客户端必须在成功发现对端服务器的服务、特征和描述符之后才能接收或发送数据。客户端有 3 种发现服务的方式：发现所有首要服务、按服务 UUID 发现首要服务和查找包含服务，其中查找包含服务使用较少，本章不做详解。

2.1. 发现所有首要服务

客户端若希望发现对端设备的所有公开的首要服务，可以使用属性协议的“按组类型读取请求”，请求的句柄范围设置为“0x0001~0xFFFF”，即起始句柄为“0x0001”，结束句柄为“0xFFFF”，属性类型设置为首要服务（UUID：0x2800，Primary Service）。之后，服务器会将发现的一个或多个首要服务回复给客户端。

“按组类型读取请求”和“按组类型读取响应”数据格式如下表所示。

表 8-1：按组类型读取请求格式

参数	大小（字节数）	描述
属性操作码（Attribute Opcode）	1	0x10 = 按组类型读取请求。
起始句柄（Starting Handle）	2	第一个请求句柄值。
结束句柄（Ending Handle）	2	最后一个请求句柄值。
属性类型（Attribute Group Type）	2 或 16	2 字节或 16 字节 UUID。

表 8-2：按组类型读取响应格式

参数	大小（字节数）	描述
属性操作码（Attribute Opcode）	1	0x11 = 按组类型读取响应。
长度（Length）	2	每个属性数据的大小。
属性数据列表（Attribute Data List）	4~(ATT_MTU-2)	属性数据列表。

发现所有首要服务流程如下图所示，APP 调用 API 函数 sd_ble_gattc_primary_services_discover()通知协议栈启动服务发现规程（函数参数 p_srcv_uuid 指向 NULL 表示发现所有首

要服务),之后协议栈使用“按组类型读取请求”发现服务器公开的首要服务,随后,服务器将发现的首要服务回复给客户端(按组类型读取响应),并向应用程序提交“BLE_GATT C_EVT_PRIM_SRVC_DISC_RSP”事件。

如果没有属性需要返回,则将发送带有“Attribute Not Found”错误代码的错误响应。

如果最后一个服务的最后一个句柄不是0xFFFF,客户端将继续发送“按组类型读取请求”,请求的起始句柄紧跟着上一条响应中最后一个服务的最后一个句柄。

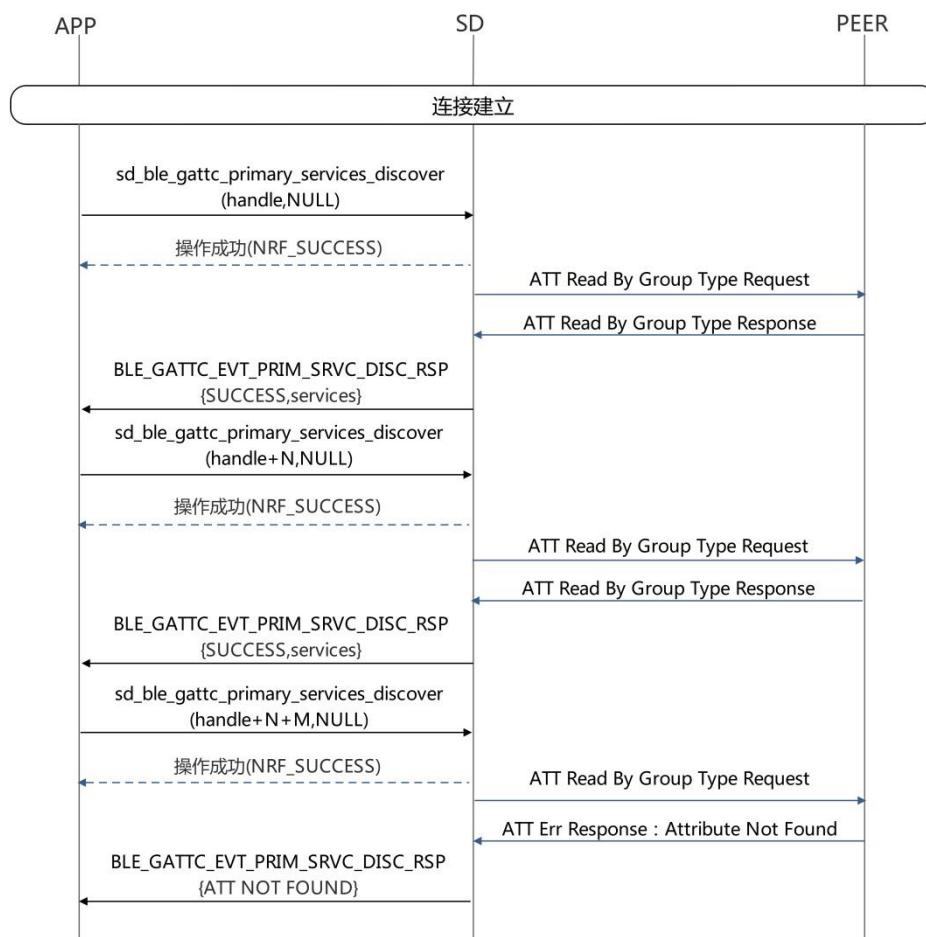


图 8-1: 发现服务流程-发现所有首要服务

2.2. 按服务 UUID 发现服务

如果客户端不想发现所有的服务,而是只想发现某个特定的服务,客户端可以使用“按类型值读取请求”,将句柄范围设置为“0x0001~0xFFFF”,即起始句柄为“0x0001”,结束句柄为“0xFFFF”,属性类型设置为首要服务(UUID: 0x2800, Primary Service),属性值设置为要查找的服务的UUID。之后,服务器会响应一个或多个句柄信息的列表。

按类型值读取请求和响应的数据格式如下表所示。

表 8-3: 按类型值读取请求格式

参数	大小(字节数)	描述
属性操作码 (Attribute Opcode)	1	0x08 = 按类型值读取请求。
起始句柄 (Starting Handle)	2	第一个请求句柄值。

结束句柄 (Ending Handle)	2	最后一个请求句柄值。
属性类型 (Attribute Type)	2	要查找的 2 字节 UUID。

表 8-4: 按类型值读取响应格式

参数	大小 (字节数)	描述
属性操作码 (Attribute Opcode)	1	0x09 = 按类型值读取响应。
句柄列表 (Length)	4 ~(ATT_MTU-1)	一个或多个句柄信息的列表。
属性数据列表 (Attribute Data List)	2 ~(ATT_MTU-2)	属性数据列表。

按服务 UUID 发现服务流程如下图所示，APP 调用 API 函数 `sd_ble_gattc_primary_services_discover()` 通知协议栈启动服务发现规程（函数参数 `p_srvc_uuid` 指向需要发现的服务的 UUID，表示使用按类型值读取请求），之后协议栈向服务器发送“按类型值读取请求”，随后，服务器将查找的一个或多个句柄信息的列表返回给客户端，并向应用程序提交“BLE_GATT_EVT_PRIM_SRVC_DISC_RSP”事件。

如果没有属性需要返回，则将发送带有“Attribute Not Found”错误代码的错误响应。

如果最后一个服务的最后一个句柄不是 0xFFFF，客户端将继续发送“按类型值读取请求”，请求的起始句柄紧跟着上一条响应中最后一个服务的最后一个句柄。

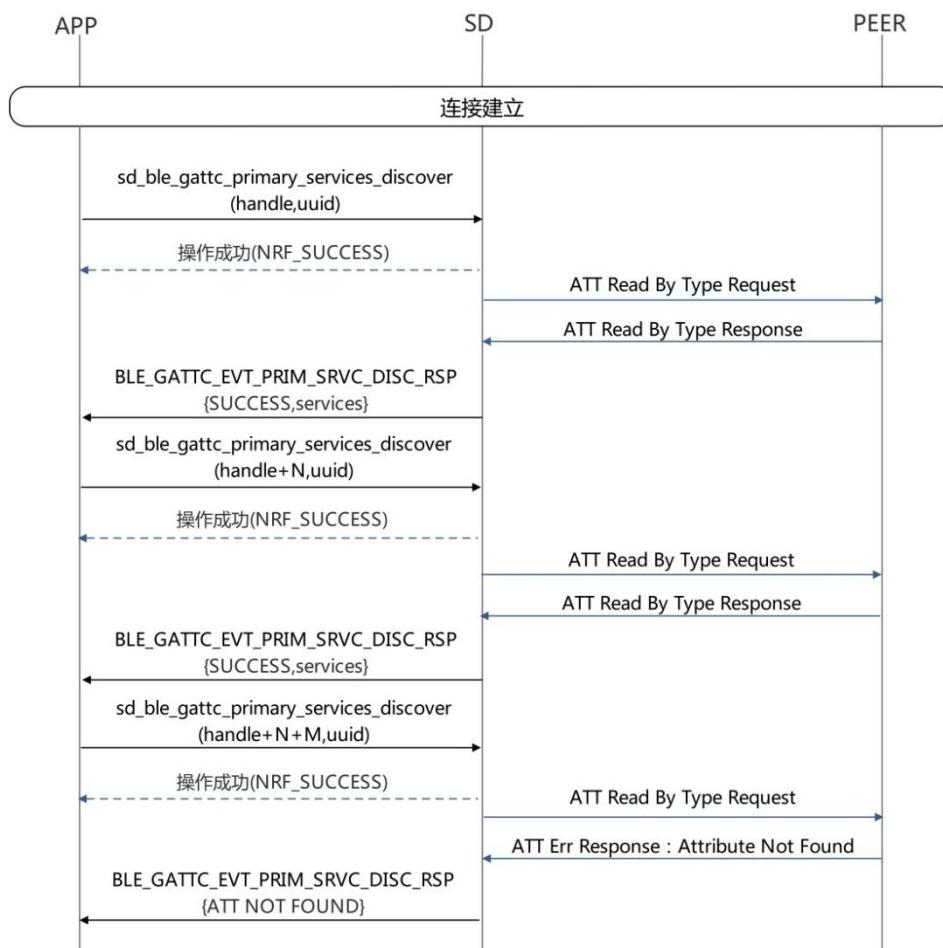


图 8-2: 发现服务流程-按服务 UUID 发现首要服务

3. 发现特征

服务发现完成之后，接着就可以发现服务的特征，发现特征同样使用属性协议的“按类型值读取请求”操作完成。客户端将属性类型设置为特征(UUID: 0x2803, GATT Characteristic Declaration)，将句柄范围设置为该服务的句柄范围，即可发现该服务的所有特征。

发现特征的流程如下图所示，APP 调用 API 函数 `sd_ble_gattc_characteristics_discover()` 通知协议栈启动特征发现规程（函数参数 `p_handle_range` 指向起始句柄和结束句柄），之后协议栈向服务器发送“按类型值读取请求”，随后，服务器的响应中包含每一个特征的声明和句柄，并向应用程序提交“BLE_GATT_C_EVT_CHAR_DISC_RSP”事件。如果没有属性需要返回，则将发送带有“Attribute Not Found”错误代码的错误响应。

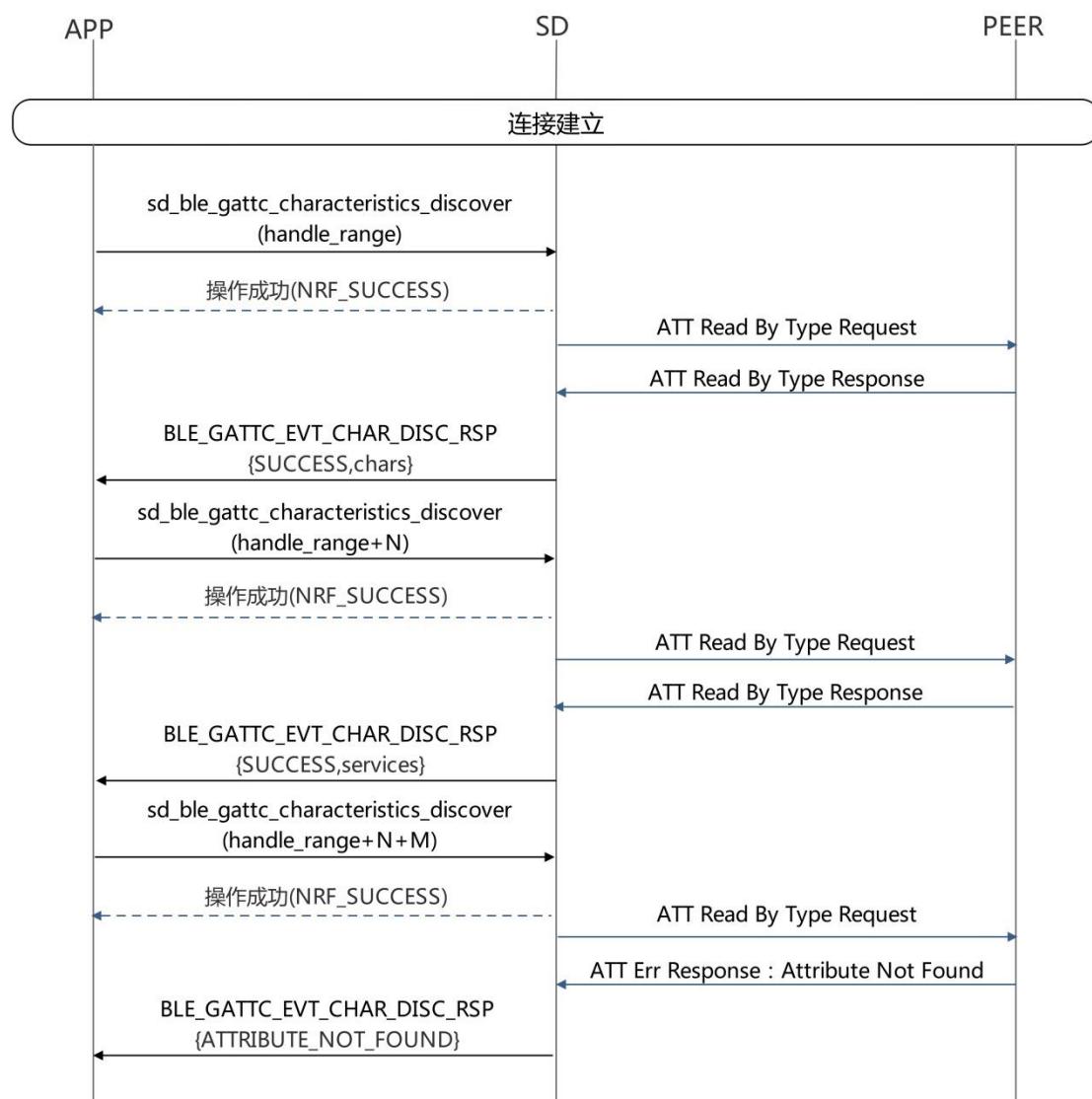


图 8-3：发现特征流程

4. 发现描述符

特征发现完成之后，接着就可以发现特征的描述符，发现描述符时使用属性协议的“查

找信息请求”操作完成。服务器通过查找信息响应返回特征的所有描述符的句柄和类型。

查找信息响应必须是完整的句柄-UUID 对，并且句柄-UUID 对不能在响应数据包之间拆分，也就是一个响应包中包含的必定是完整的句柄-UUID 对，而不能只包含句柄或者只包含 UUID。同时，句柄-UUID 对应按属性句柄的升序返回。

按类型值读取请求和响应的数据格式如下表所示。

表 8-5: 查找信息请求格式

参数	大小 (字节数)	描述
属性操作码 (Attribute Opcode)	1	0x04 = 查找信息请求。
起始句柄 (Starting Handle)	2	第一个请求句柄值。
结束句柄 (Ending Handle)	2	最后一个请求句柄值。

表 8-6: 查找信息响应格式

参数	大小 (字节数)	描述
属性操作码 (Attribute Opcode)	1	0x09 = 查找信息响应。
格式 (Format)	1	信息数据的格式。
信息数据 (Information Data)	4 ~(ATT_MTU-2)	信息数据，其格式由“格式”字段确定。

Format 参数指示了句柄-UUID 对中的 UUID 的类型：标准的 16 位 UUID 或自定义的 128 位 UUID，Format 参数定义如下表所示。

表 8-7: 查找信息响应格式

名称	格式	描述
句柄和 16 位 UUID	0x01	1 个或多个句柄-UUID 对 (16 位 UUID)。
句柄和 128 位 UUID	0x02	1 个或多个句柄-UUID 对 (128 位 UUID)。

发现描述符的流程如下图所示，APP 调用 API 函数 `sd_ble_gattc_descriptors_discover()` 通知协议栈启动描述符发现规程（函数参数 `p_handle_range` 指向起始句柄和结束句柄），之后协议栈向服务器发送“按类型值读取请求”，随后，服务器的响应中包含句柄-UUID 对。如果没有属性需要返回，则将发送带有“Attribute Not Found”错误代码的错误响应。

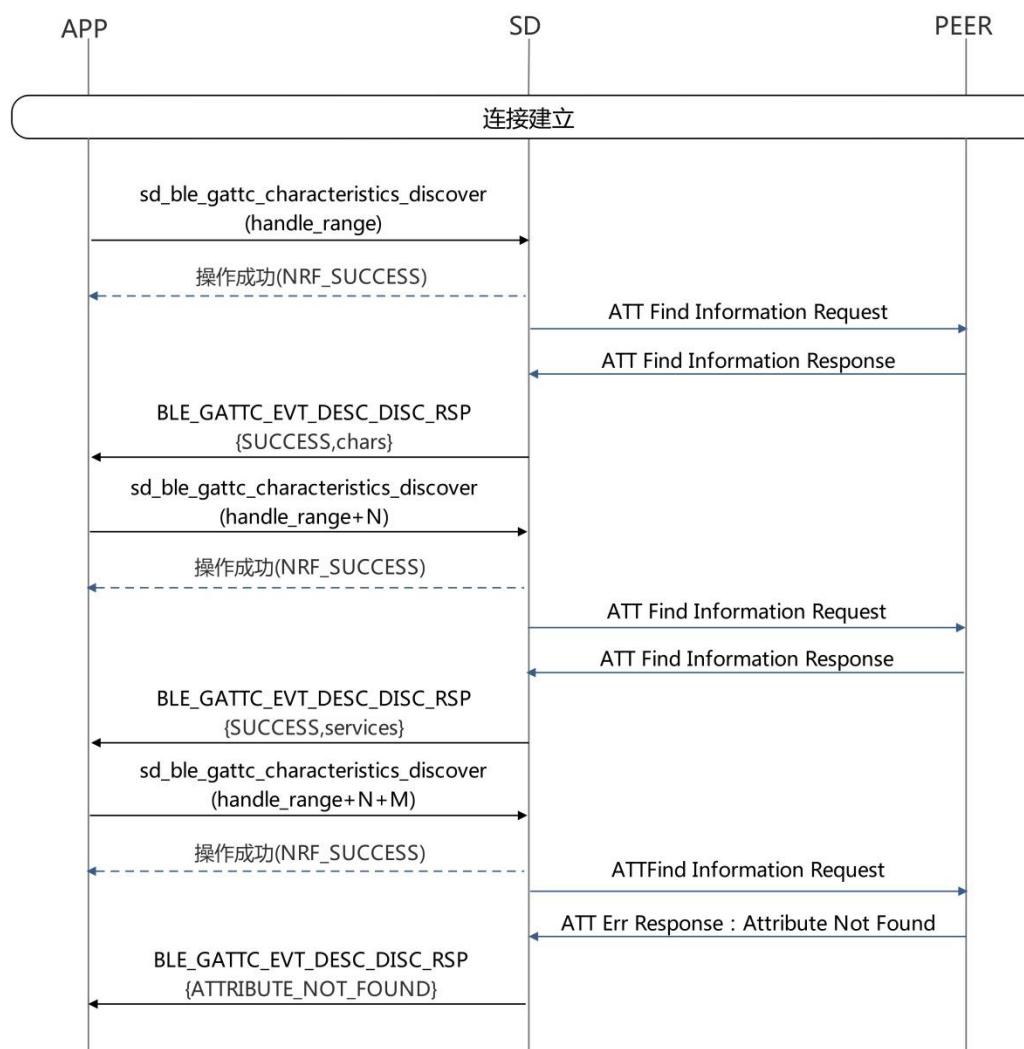


图 8-4：发现描述符流程

5. 程序编写

❖ 注：本实验在“实验 7-1：加入连接参数更新”的基础上修改，连接的从机为《开发指南下册-从机》中的“实验 7-3：串口透传长包传输-判断 notify”，本实验对应的实验源码是：“实验 8-1：串口透传主机”。

Nordic 的 SDK 中已经提供了串口透传主机的 demo，他在 SDK 中的路径是：“nRF5_SDK_16.0.0_98a08e2\examples\ble_central\ble_app_uart_c”。

本例中，会使用 SDK 中串口透传客户端的代码，为了区别于 SDK 中的串口透传 demo 以及描述客户端文件的编写，我们新建串口透传客户端的文件：在工程的 app 文件夹下面新建一个 ble_uarts_c 文件夹，之后新建如下表所示的文件保存到该文件夹，同时将“ble_uarts_c.c”文件加入到工程的 nRF_BLE_Services 组。

表 8-8：串口透传客户端文件

项目	文件名
串口透传客户端文件	ble_uarts_c.c

1. 需要引用的头文件

因为在“main.c”文件中需要引用串口透传客户端文件中的函数和定义以及数据库发现模块，所以“main.c”文件需要引用下面的头文件。

代码清单：需要引用的头文件

1. #include "ble_uarts_c.h"
2. #include "ble_db_discovery.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。

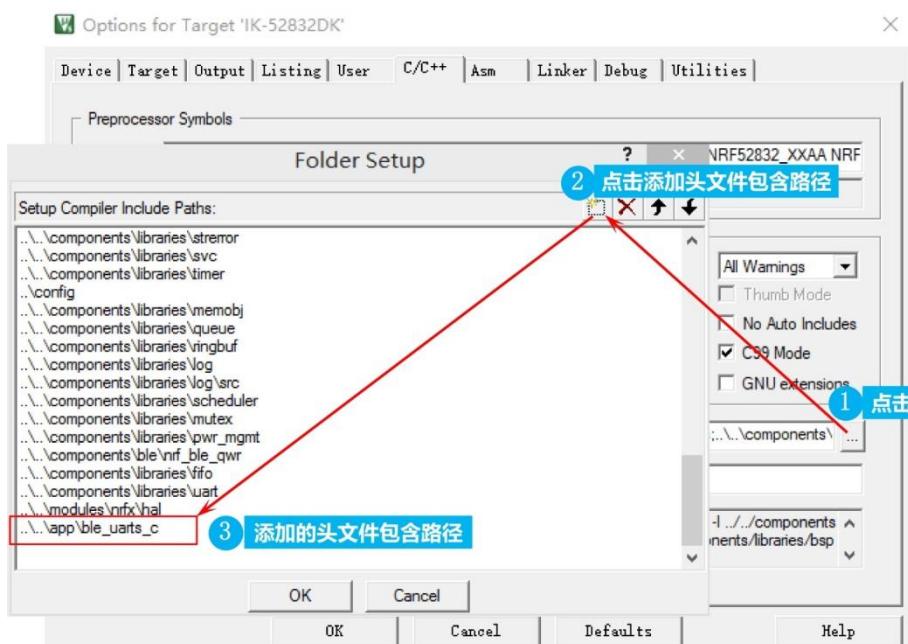


图 8-5：添加头文件包含路径

本例需要添加的头文件路径如下表：

表 8-9：头文件包含路径

序号	路径
1	...\\app\\ble_uarts_c

5.1. 程序编写流程

本例中的重点是使用 DB 发现模块发现对端服务器上的服务、特征和描述符。DB 发现模块是 Nordic 提供的库，用于客户端发现服务器的服务、特征和描述符。应用程序可以使用 DB 发现模块公开的 API 和类型在对端服务器上执行服务和特征的发现，并且 DB 发现模块还可用在多个远程设备中发现所需的服务。

DB 发现模块可以发现的每个服务的最大特征数由“db_disc_config.h”中定义的服务结构中的特征数决定，如果对端设备具有超过支持的特征数量，则将发现第一个找到的特征，并且将忽略任何其他特征。DB 发现模块仅在对端服务器搜索以下描述符：客户端特征配置

描述符、特征扩展性质描述符和特征用户描述描述符。

下图是程序中服务发现和数据收发的流程，应用程序首先需要初始化 DB 发现模块，接着初始化串口透传客户端。初始化串口透传客户端时会将串口透传的服务 UUID 注册给 DB 发现模块，DB 发现模块由此知道应用程序在服务器上发现哪个服务。接着，当连接建立后客户端启动发现规程并等待发现完成事件，发现完成后为服务和特征分配句柄并使能服务器的 notify，之后即可在事件中接收服务器的数据、调用发送函数将数据发送给客户端。

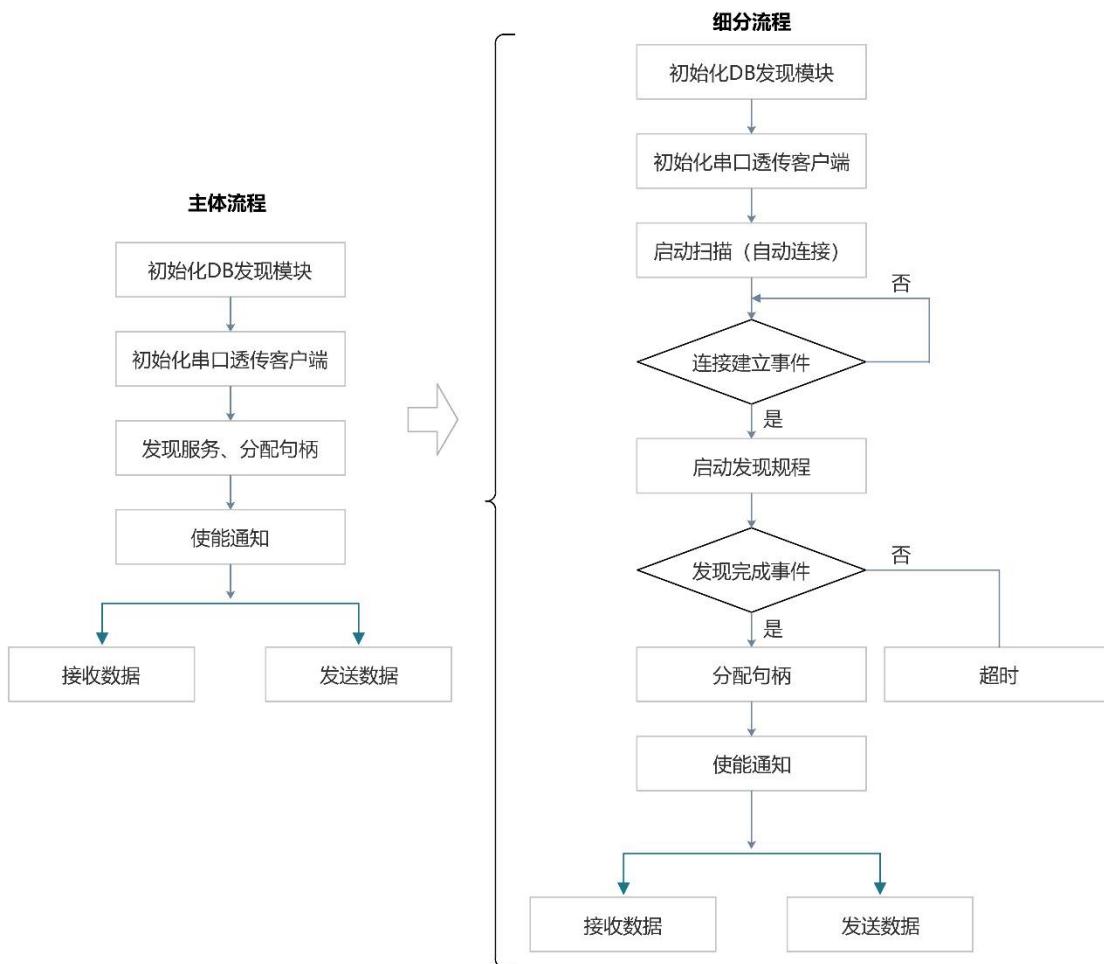


图 8-6：程序编写流程

5.2. 初始化 DB 发现模块

主机程序中，所有的发现操作都是由 DB 发现模块完成的，换句话说就是 DB 发现模块完成了 BLE 的服务发现相关的协议，应用程序只需初始化、启动和关注事件即可。

初始化 DB 发现模块包含定义 DB 发现模块实例、调用 `ble_db_discovery_init()` 函数初始化 DB 发现模块两个部分。

5.2.1. 定义 DB 发现模块实例

程序中使用 DB 发现模块时，需要定义 DB 发现模块实例，用来记录发现的状态和接收协议栈事件（将 DB 发现模块中的事件处理函数“`ble_db_discovery_on_ble_evt`”注册给协议栈）。

本例中定义了名称为 m_db_disc 的数据库发现模块实例，代码清单如下。

代码清单：定义名称为 m_db_disc 的数据库发现模块实例

```
1. //定义名称为 m_db_disc 的 DB 发现模块实例
2. BLE_DB_DISCOVERY_DEF(m_db_disc);
```

5.2.2. 初始化 DB 发现模块

DB 发现模块使用 ble_db_discovery_init() 函数初始化，函数原型如下。

表 8-10: ble_db_discovery_init() 函数

函数原型	<code>uint32_t ble_db_discovery_init (ble_db_discovery_evt_handler_t evt_handler)</code>
函数功能	该函数用于初始化 DB 发现模块。
参数	[in] <code>evt_handler</code> : 当与已注册的服务发现有关的任何事件发生时，DB 发现模块将调用该事件处理程序。
返回值	<code>NRF_SUCCESS</code> : 初始化成功。 <code>NRF_ERROR_NULL</code> : 参数 <code>evt_handler</code> 为 NULL。

初始化时会注册一个事件处理函数，该函数用于处理来自 DB 发现模块的事件。事件处理函数中会根据发现的 UUID 将事件转发到各自的服务，对于本例来说只有串口透传服务，因此只需将事件转发给串口透传服务，事件处理函数代码清单如下。

代码清单：DB 发现模块事件处理函数

```
1. //DB 发现模块事件处理函数，这里将事件转发给各个服务，本例只有串口透传服务，因此只转发给串
2. //口透传
3. static void db_disc_handler(ble_db_discovery_evt_t * p_evt)
4. {
5.     //事件转发给串口透传服务
6.     ble_uarts_c_on_db_disc_evt(&m_ble_uarts_c, p_evt);
7.     //如果有其他服务，这里添加转发给服务的函数
8. }
```

DB 发现模块事件处理函数中将事件转发给串口透传服务事件处理函数 “ble_uarts_c_on_db_disc_evt”，该函数中会判断事件类型以及串口透传客户端初始化时注册给 DB 发现模块的 UUID，之后获取 DB 发现模块提供的串口透传特征句柄，最终调用串口透传客户端初始化时注册的事件处理函数执行下一步处理（分配句柄和使能 Notify），代码清单如下。

代码清单：串口透传 DB 发现模块事件处理函数

```
1. void ble_uarts_c_on_db_disc_evt(ble_uarts_c_t * p_ble_uarts_c, ble_db_discovery_
    evt_t * p_evt)
```

```
2. {
3.     ble_uarts_c_evt_t uarts_c_evt;
4.     memset(&uarts_c_evt,0,sizeof(ble_uarts_c_evt_t));
5.
6.     ble_gatt_db_char_t * p_chars = p_evt->params.discovered_db.characteristics;
7.
8.     //判断 UUID 是不是串口透传的 UUID, 如果是的话, 表示串口透传服务发现完成
9.     if ( (p_evt->evt_type == BLE_DB_DISCOVERY_COMPLETE)
10.         && (p_evt->params.discovered_db.srv_uuid.uuid == BLE_UUID_UARTS_SERVICE)
11.         && (p_evt->params.discovered_db.srv_uuid.type == p_ble_uarts_c->uuid_type))
12.    {
13.        //获取特征句柄, 调用串口透传客户端事件处理函数时作为参数传递给该函数, 由其分配句柄
14.        for (uint32_t i = 0; i < p_evt->params.discovered_db.char_count; i++)
15.        {
16.            switch (p_chars[i].characteristic.uuid.uuid)
17.            {
18.                case BLE_UUID_UARTS_RX_CHARACTERISTIC:
19.                    uarts_c_evt.handles.uarts_rx_handle = p_chars[i].characteristic
20.                                         .handle_value;
21.                    break;
22.                case BLE_UUID_UARTS_TX_CHARACTERISTIC:
23.                    uarts_c_evt.handles.uarts_tx_handle = p_chars[i].characteristic
24.                                         .handle_value;
25.                    uarts_c_evt.handles.uarts_tx_cccd_handle = p_chars[i].cccd_hand
26.                                         le;
27.                    break;
28.                default:
29.                    break;
30.            }
31.        }
32.        //调用串口透传客户端事件处理函数 (该函数是初始化串口透传客户端时注册的)
33.        if (p_ble_uarts_c->evt_handler != NULL)
34.        {
35.            uarts_c_evt.conn_handle = p_evt->conn_handle;//保存连接句柄
36.            //设置事件类型
37.            uarts_c_evt.evt_type = BLE_UARTS_C_EVT_DISCOVERY_COMPLETE;
38.            //调用串口透传客户端事件处理函数
39.            p_ble_uarts_c->evt_handler(p_ble_uarts_c, &uarts_c_evt);
40.        }
41.    }
42. }
```

5.2.3. 启动服务发现

启动服务发现是在连接建立后执行的，一般地，应用程序接收到“BLE_GAP_EVT_CONNECTED”事件后调用 ble_db_discovery_start()函数启动服务发现，该函数原型如下。

表 8-11: ble_db_discovery_start()函数

函数原型	<code>uint32_t ble_db_discovery_start((ble_db_discovery_t * uint16_t))</code>
函数功能	该函数用于启动在服务器上发现 GATT 数据库。
参数	[out] <code>p_db_discovery</code> : 指向 DB 发现结构体。 [in] <code>conn_handle</code> : 将要启动服务发现的连接的句柄。
返回值	NRF_SUCCESS: 初始化成功。 NRF_ERROR_NULL: 参数指针指向 NULL。 NRF_ERROR_INVALID_STATE: 调用该函数之前没有调用“ble_db_discovery_init()”函数初始化 DB 发现模块，或者没有调用“ble_db_discovery_evt_register()”函数。 NRF_ERROR_BUSY: 如果已经使用 DB 发现结构体“ble_db_discovery_t”进行了服务发现，返回该错误代码。这时可以使用不同的 DB 发现结构体执行发现，或等待数据库发现事件，然后重试。

启动服务发现的代码清单如下，他是在 BLE 事件处理函数 ble_evt_handler()的连接建立事件“BLE_GAP_EVT_CONNECTED”下执行的。

代码清单：启动服务发现

```

1. case BLE_GAP_EVT_CONNECTED://建立连接事件
2. //其他功能代码
3.
4. //启动服务发现，串口透传客户端会等待发现完成事件
5. err_code = ble_db_discovery_start(&m_db_disc, p_ble_evt->evt.gap_evt.
6.                                     conn_handle);
7. APP_ERROR_CHECK(err_code);
8. break;

```

5.3. 初始化串口透传客户端

初始串口透传客户端包含定义串口透传客户端实例和初始化串口透传客户端两个部分。

5.3.1. 定义串口透传客户端实例

应用程序中使用宏定义 BLE_UARTS_C_DEF 实例化一个串口透传客户端，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：定义串口透传客户端实例

```
1. //定义串口透传客户端实例
2. BLE_UARTS_C_DEF(m_ble_uarts_c);
```

BLE_UARTS_C_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：定义串口透传客户端实例

```
1. #define BLE_UARTS_C_DEF(_name) \
2. static ble_uarts_c_t _name; \
3. NRF_SDH_BLE_OBSERVER(_name ## _obs, \
4.                      BLE_UARTS_C_BLE_OBSERVER_PRIO, \
5.                      ble_uarts_c_on_ble_evt, &_name)
```

可以看到BLE_UARTS_C_DEF主要完成2件事情：定义了static类型串口透传客户端结构体变量和注册BLE事件监视者。

1. 定义static类型串口透传客户端结构体变量

该变量定义后，即为串口透传客户端结构体分配了内存：因为变量为static类型，因此该语句执行后，串口透传客户端结构体变量常驻内存，即实例化了一个名为“_name”的串口透传客户端，协议栈和应用程序均可通过该名称访问串口透传客户端实例。

本例中，串口透传客户端结构体包含4个变量，之所以定义这4个变量的原因如下。

- 1) **uuid_type**: UUID类型，初始化串口透传客户端时需要指明服务的UUID类型（标准的还是自定义的），所以，需要包含UUID类型。
- 2) **conn_handle**: 连接句柄，指示该客户端的连接。
- 3) **handles**: 客户端读/写服务器的特征值和接收Notify都需要与之连接的对端设备的特征句柄，因此需要定义该变量。
- 4) **evt_handler**: 串口透传客户端事件句柄，当产生与串口透传相关的事件时调用。如串口透传主机接收到从机发送过来的数据时，调用该事件处理函数处理接收的数据。
- 5) **error_handler**: 串口透传客户端错误事件句柄，当产生错误时调用。
- 6) *** p_gatt_queue**: 指向BGQ(BLE GATT Queue)实例，BGQ是SDK16.0新增加的功能，他的作用是当SoftDevice无法立即处理GATT规程时，缓存GATT规程，等待SoftDevice空闲时处理，大多数GATT客户端库都使用了BGQ。

串口透传客户端结构体的代码清单如下：

代码清单：串口透传客户端结构体

```
1. //串口透传客户端结构体
2. struct ble_uarts_c_s
3. {
4.     uint8_t          uuid_type; //UUID类型
```

```

5.     uint16_t          conn_handle; //连接句柄
6.     ble_uarts_c_handles_t    handles;      //与之连接需要交互数据的对端设备的句柄
7.     //串口透传事件句柄, 当产生与串口透传相关的事件时调用
8.     ble_uarts_c_evt_handler_t evt_handler;
9.     ble_srv_error_handler_t   error_handler; //串口透传错误事件句柄, 当产生错误时调用
10.    nrf_ble_gq_t           * p_gatt_queue; //指向 GATT Queue 实例
11. };

```

2. 注册 BLE 事件监视者

串口透传客户端需要关注 BLE 协议栈的事件，因此需要注册 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，BLE 事件监视者接收到协议栈的事件后，进入事件处理函数 `ble_uarts_c_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。

这里要注意：BLE 事件监视者注册的事件处理函数 `ble_uarts_c_on_ble_evt()` 是实际接收协议栈事件的地方，而串口透传客户端结构体中的事件处理函数是在 `ble_uarts_c_on_ble_evt()` 中调用的。

■ 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。本例中，我们关注 BLE 协议栈提交的事件中的 2 个事件。

- **BLE_GATTC_EVT_HVX 事件：**通知或指示事件，因为需要接收服务器的通知，因此需要关注该事件。
 - **BLE_GAP_EVT_DISCONNECTED 事件：**连接断开事件，连接断开后，需要将透传客户端实例中的连接句柄设置为无效，并且通知应用程序启动扫描，因此需要关注该事件。
- 事件处理函数代码清单如下。

代码清单：串口透传 BLE 事件监视者事件处理函数

```

1. void ble_uarts_c_on_ble_evt(ble_evt_t const * p_ble_evt, void * p_context)
2. {
3.     ble_uarts_c_t * p_ble_uarts_c = (ble_uarts_c_t *)p_context;
4.     //检查参数是否正确
5.     if ((p_ble_uarts_c == NULL) || (p_ble_evt == NULL))
6.     {
7.         return;
8.     }
9.     //检查连接句柄是否有效
10.    if ( (p_ble_uarts_c->conn_handle != BLE_CONN_HANDLE_INVALID)
11.        &&(p_ble_uarts_c->conn_handle != p_ble_evt->evt.gap_evt.conn_handle)
12.        )
13.    {
14.        return;
15.    }
16.    //判断事件类型

```

```

17.     switch (p_ble_evt->header.evt_id)
18.     {
19.         case BLE_GATTC_EVT_HVX://通知或指示事件
20.             on_hvx(p_ble_uarts_c, p_ble_evt);
21.             break;
22.         case BLE_GAP_EVT_DISCONNECTED://连接断开事件
23.             if (p_ble_evt->evt.gap_evt.conn_handle == p_ble_uarts_c->conn_handle
24.                 && p_ble_uarts_c->evt_handler != NULL)
25.             {
26.                 ble_uarts_c_evt_t uarts_c_evt;
27.                 //设置事件类型
28.                 uarts_c_evt.evt_type = BLE_UARTS_C_EVT_DISCONNECTED;
29.                 //串口透传实例中的连接句柄设置为无效
30.                 p_ble_uarts_c->conn_handle = BLE_CONN_HANDLE_INVALID;
31.                 //调用串口透传客户端事件处理函数(该函数是初始化串口透传客户端时注册的)
32.                 p_ble_uarts_c->evt_handler(p_ble_uarts_c, &uarts_c_evt);
33.             }
34.             break;
35.         default:
36.             break;
37.     }
38. }
```

事件处理函数中的on_hvx()函数是用来处理BLE_GATTC_EVT_HVX事件的,通俗地说,该函数是处理从机发过来的数据的。

on_hvx()函数处理从协议栈接收到的Handle Value Notification(句柄值通知),并检查他是否是来自对端设备的串口透传TX特性的通知,如果是,此函数将调用串口透传客户端事件处理函数(该函数是初始化串口透传客户端时注册的),由应用程序处理接收的数据。

代码清单: on_hvx()函数

```

1. static void on_hvx(ble_uarts_c_t * p_ble_uarts_c, ble_evt_t const * p_ble_evt)
2. {
3.     //是串口透传 TX 特性的通知
4.     if ((p_ble_uarts_c->handles.uarts_tx_handle != BLE_GATT_HANDLE_INVALID)
5.         && (p_ble_evt->evt.gattc_evt.params.hvx.handle == p_ble_uarts_c->handles.
6.               uarts_tx_handle)
7.         && (p_ble_uarts_c->evt_handler != NULL))
8.     {
9.         ble_uarts_c_evt_t ble_uarts_c_evt;
10.        //设置事件类型
11.        ble_uarts_c_evt.evt_type = BLE_UARTS_C_EVT_UARTS_TX_EVT;
12.        //p_data 指向数据存放的地址
13.        ble_uarts_c_evt.p_data = (uint8_t *)p_ble_evt->evt.gattc_evt.params.hvx.
14.                               data;
```

```

15.     //获取数据长度
16.     ble_uarts_c_evt.data_len = p_ble_evt->evt.gattc_evt.params.hvx.len;
17.     //调用串口透传客户端事件处理函数（该函数是初始化串口透传客户端时注册的）
18.     p_ble_uarts_c->evt_handler(p_ble_uarts_c, &ble_uarts_c_evt);
19.     NRF_LOG_DEBUG("Client sending data.");
20. }
21. }
```

5.3.2. 初始化串口透传客户端

初始化串口透传客户端的作用是初始化串口透传实例中的串口透传客户端结构体，将结构体中的连接句柄、特征值句柄初始化为无效，并保存事件回调函数，之后将串口透传服务的UUID注册给DB发现模块。

程序中，为了更清晰和模块化编程，首先定义一个串口透传客户端初始化结构体，并设置串口透传客户端的事件回调函数，之后调用ble_uarts_c_init()函数初始化串口透传客户端，代码如下。

代码清单：初始化串口透传客户端

```

1. static void uarts_c_init(void)
2. {
3.     ret_code_t      err_code;
4.     ble_uarts_c_init_t init;
5.     //串口透传客户端事件处理函数
6.     init.evt_handler    = ble_uarts_c_evt_handler;
7.     init.error_handler = uarts_error_handler;
8.     //GATT Queue 赋值，指向 m_ble_gatt_queue
9.     init.p_gatt_queue  = &m_ble_gatt_queue;
10.    //初始化串口透传客户端
11.    err_code = ble_uarts_c_init(&m_ble_uarts_c, &init);
12.    APP_ERROR_CHECK(err_code);
13. }
```

ble_uarts_c_init()函数代码清单如下：

代码清单：初始化串口透传客户端：ble_uarts_c_init()函数

```

1. uint32_t ble_uarts_c_init(ble_uarts_c_t * p_ble_uarts_c, ble_uarts_c_init_t * p_
   ble_uarts_c_init)
2. {
3.     uint32_t      err_code;
4.     ble_uuid_t    uart_uuid;
5.     ble_uuid128_t uarts_base_uuid = UARTS_BASE_UUID;
6.     //检查函数参数是否合法
7.     VERIFY_PARAM_NOT_NULL(p_ble_uarts_c);
```

```

8.     VERIFY_PARAM_NOT_NULL(p_ble_uarts_c_init);
9.     VERIFY_PARAM_NOT_NULL(p_ble_uarts_c_init->p_gatt_queue);
10.    //将自定义的 UUID 基数写入到协议栈
11.    err_code = sd_ble_uuid_vs_add(&uart_base_uuid, &p_ble_uarts_c->uuid_type);
12.    VERIFY_SUCCESS(err_code);
13.
14.    uart_uuid.type = p_ble_uarts_c->uuid_type;
15.    uart_uuid.uuid = BLE_UUID_UARTS_SERVICE;
16.    //初始化串口透传客户端实例: 将连接句柄、特征值句柄初始化为无效, 保存事件回调函数,
17.    //GATT Queue 指向定义的 GATT Queue 实例
18.    p_ble_uarts_c->conn_handle          = BLE_CONN_HANDLE_INVALID;
19.    p_ble_uarts_c->evt_handler         = p_ble_uarts_c_init->evt_handler;
20.    p_ble_uarts_c->error_handler       = p_ble_uarts_c->error_handler;
21.    p_ble_uarts_c->handles.uarts_tx_handle = BLE_GATT_HANDLE_INVALID;
22.    p_ble_uarts_c->handles.uarts_rx_handle = BLE_GATT_HANDLE_INVALID;
23.    p_ble_uarts_c->p_gatt_queue        = p_ble_uarts_c_init->p_gatt_queue;
24.    //将串口透传服务的 UUID 注册给 DB 发现模块
25.    return ble_db_discovery_evt_register(&uart_uuid);
26. }

```

■ 串口透传客户端事件处理函数 ble_uarts_c_evt_handler

串口透传客户端初始化时注册的事件回调函数也就是“5.3.1”节中 ble_uarts_c_on_ble_evt()函数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串口透传客户端“ble_uarts_c 程序模块”定义的事件。

“ble_uarts_c 程序模块”定义了如下 3 个事件，用来通知应用程序执行相应动作。

代码清单：“ble_uarts_c 程序模块”定义的事件

```

1. typedef enum
2. {
3.     BLE_UARTS_C_EVT_DISCOVERY_COMPLETE, //服务发现完成
4.     BLE_UARTS_C_EVT_NUS_TX_EVT,        //指示主机接收到从机的数据
5.     BLE_UARTS_C_EVT_DISCONNECTED      //连接断开
6. } ble_uarts_c_evt_type_t;

```

应用程序在串口透传客户端事件处理函数“ble_uarts_c_evt_handler”中处理这些事件，各个事件的处理如下。

- **BLE_UARTS_C_EVT_DISCOVERY_COMPLETE:** 服务发现完成事件，指示 DB 发现模块已经成功发现对端设备的服务，应用程序接收到该事件后即可分配特征值句柄和使能从机的 Notify。
- **BLE_UARTS_C_EVT_NUS_TX_EVT:** 指示主机接收到从机的通知或指示，应用程序接收到该事件后即可读取接收的数据。本例中，将读取的数据通过串口输出。
- **BLE_UARTS_C_EVT_DISCONNECTED:** 指示客户端连接已经断开，这时主机可以重

启扫描。

对应的串口透传客户端事件处理函数代码清单如下，另外，再次强调一下：BLE 的数据接收是没有专门的函数去接收数据的（BLE 发送有专用的函数），接收是通过事件来完成的，本例中，BLE 接收到数据后，透传客户端会向应用程序发送“BLE_UARTS_C_EVT_NUS_RX_EVT”事件，应用程序接收到该事件后，即可读取接收的数据。

代码清单：串口透传客户端事件处理函数

```

1. static void ble_uarts_c_evt_handler(ble_uarts_c_t * p_ble_uarts_c, ble_uarts_c_e
2.                                     vt_t const * p_ble_uarts_evt)
3. {
4.     ret_code_t err_code;
5.
6.     switch (p_ble_uarts_evt->evt_type)
7.     {
8.         case BLE_UARTS_C_EVT_DISCOVERY_COMPLETE://服务发现完成事件
9.             NRF_LOG_INFO("Discovery complete.");
10.            //分配特征句柄
11.            err_code = ble_uarts_c_handles_assign(p_ble_uarts_c, p_ble_uarts_evt->
12.                                                    conn_handle, &p_ble_uarts_evt->handles);
13.            APP_ERROR_CHECK(err_code);
14.            //使能从机的 Notify (通知)
15.            err_code = ble_uarts_c_tx_notif_enable(p_ble_uarts_c);
16.            APP_ERROR_CHECK(err_code);
17.            NRF_LOG_INFO("Connected to device with Nordic UART Service.");
18.            break;
19.        case BLE_UARTS_C_EVT_NUS_TX_EVT://数据接收事件，串口打印出接收的数据
20.            uart_chars_print(p_ble_uarts_evt->p_data, p_ble_uarts_evt->data_len);
21.            break;
22.        case BLE_UARTS_C_EVT_DISCONNECTED://连接断开，重启扫描
23.            NRF_LOG_INFO("Disconnected.");
24.            scan_start();
25.            break;
26.    }
27. }
```

■ 向 DB 发现模块注册服务 UUID

应用程序通常只会关心某些特定的服务，如本例中应用程序只会关心从对端设备中发现的服务是不是串口透传服务，因此，应用程序在初始化串口透传客户端时，会使用 ble_db_discovery_evt_register() 函数将自己关心的服务的 UUID 注册给 DB 发现模块，通知 DB 发现模块哪些服务是自己关心的。ble_db_discovery_evt_register() 函数原型如下表所示。

表 8-12: ble_db_discovery_evt_register() 函数

函数原型	uint32_t ble_db_discovery_evt_register
------	--

	<pre>(const ble_uuid_t *const p_uuid)</pre>
函数功能	<p>该函数用于向 DB 发现模块注册服务 UUID。应用程序可以使用此功能来通知他有兴趣在服务器上发现哪个服务。</p> <p>注意：该模块可以发现的服务总数是 BLE_DB_DISCOVERY_MAX_SRV。这实际上意味着最大可能的注册数量等于 BLE_DB_DISCOVERY_MAX_SRV。</p>
参数	[out] <code>p_uuid</code> : 指向要在服务器上发现的服务的 UUID 的指针。
返回值	<p><code>NRF_SUCCESS</code>: 操作成功。</p> <p><code>NRF_ERROR_NULL</code>: 参数指针指向 NULL。</p> <p><code>NRF_ERROR_INVALID_STATE</code>: 调用该函数之前没有调用“<code>ble_db_discovery_init()</code>”函数初始化 DB 发现模块。</p> <p><code>NRF_ERROR_NO_MEM</code>: 注册的数量超过了 DB 模块允许注册的最大值。</p>

5.4. 发送数据

主机将数据发送给从机是通过写操作（Write）完成的，即主机通过“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。本例中，主机通过写串口透传 TX 特征值将数据发送给从机。

在 SDK16.0 之前，GATT 中描述的所有写入规程都是直接通过 API 函数 `sd_ble_gattc_write()` 实现的，包括特征值或描述符写入，有或没有响应写，签名或无签名写，长特征值/描述符或可靠写入。

SDK16.0 新增加了 BGQ，客户端的 GATT 规程不会直接使用 `sd_ble_gattc_write()` 函数实现，而是通过 BGQ 请求来完成的。GATT 规程通过 BGQ 请求写入到 BGQ，如果此时 SoftDevice 是空闲的，则立即处理此请求，否则，将请求保留在队列中，等待处理。当然，BGQ 程序模块中最终调用的还是 `sd_ble_gattc_write()` 函数。

GATT 请求通过函数 `nrf_ble_gq_item_add()` 加入到 BGQ 实例，该函数原型如下表所示。

表 8-13: `nrf_ble_gq_item_add()` 函数

函数原型	<pre>ret_code_t nrf_ble_gq_item_add (const nrf_ble_gq_t *const p_gatt_queue, const nrf_ble_gq_req_t *const p_req, uint16_t conn_handle)</pre>
函数功能	<p>该函数用于向 BGQ 实例添加 GATT 请求。</p> <p>此函数将 GATT 请求添加到 BGQ 实例，并为保存在请求描述符中的数据分配必要的内存。如果 SoftDevice 是空闲的，则立即处理此请求，否则，将请求保留在队列中，等待处理。</p>

参数	[in] <code>p_gatt_queue</code> : 指向 BGQ 实例。 [in] <code>p_req</code> : 指向请求。 [in] <code>conn_handle</code> : 连接句柄。
返回值	<code>NRF_SUCCESS</code> : 请求加入队列成功。 <code>NRF_ERROR_NULL</code> : 参数指针指向 NULL。 <code>NRF_ERROR_NO_MEM</code> : 队列空间不足。 <code>NRF_ERROR_INVALID_PARAM</code> : 连接句柄或 <code>p_req</code> 无效。

调用 `nrf_ble_gq_item_add()` 函数写入请求前，需要初始化写入参数，指定写操作命令、待写的特征或描述符的句柄、写入数据的长度以及写入的数据等。本例中，我们将写串口透传 TX 特征值封装成函数 `ble_uarts_c_string_send()`，供应用程序调用，代码清单如下。

代码清单：BLE 发送数据

```

1.  uint32_t ble_uarts_c_string_send(ble_uarts_c_t * p_ble_uarts_c, uint8_t * p_string, uint16_t len
2.    gth)
3.  {
4.      VERIFY_PARAM_NOT_NULL(p_ble_uarts_c);
5.      // 定义 BGQ 请求结构体变量
6.      nrf_ble_gq_req_t write_req;
7.      // 初始化参数前，先清零 BGQ 请求结构体变量
8.      memset(&write_req, 0, sizeof(nrf_ble_gq_req_t));
9.      // 检查数据长度是否正确
10.     if (length > BLE_UARTS_MAX_DATA_LEN)
11.     {
12.         NRF_LOG_WARNING("Content too long.");
13.         return NRF_ERROR_INVALID_PARAM;
14.     }
15.     // 检查连接句柄是否有效
16.     if (p_ble_uarts_c->conn_handle == BLE_CONN_HANDLE_INVALID)
17.     {
18.         NRF_LOG_WARNING("Connection handle invalid.");
19.         return NRF_ERROR_INVALID_STATE;
20.     }
21.     // 初始化写入参数
22.     write_req.type          = NRF_BLE_GQ_REQ_GATTC_WRITE; // BGQ 写请求
23.     write_req.error_handler.cb = gatt_error_handler; // GATTC 和 BGQ 处理事件句柄
24.     write_req.error_handler.p_ctx = p_ble_uarts_c;
25.     // 属性句柄设置为串口透传 rx 特征句柄
26.     write_req.params.gattc_write.handle = p_ble_uarts_c->handles.uarts_rx_handle;
27.     write_req.params.gattc_write.len    = length; // 写入的数据长度
28.     write_req.params.gattc_write.offset = 0; // 偏移量设置为 0
29.     write_req.params.gattc_write.p_value = p_string; // 指向写入的数据

```

```

29.     write_req.params.gattc_write.write_op = BLE_GATT_OP_WRITE_CMD;//写入命令，无响应
30.     write_req.params.gattc_write.flags   = BLE_GATT_EXEC_WRITE_FLAG_PREPARED_WRITE;//执行准备好的写
31.     //执行写操作
32.     return nrf_ble_gq_item_add(p_ble_uarts_c->p_gatt_queue, &write_req, p_ble_uarts_c->conn_handle);
33. }

```

主应用程序中，当串口接收完一包数据后，调用函数 `ble_uarts_c_string_send()` 将数据发送给从机，代码清单如下。

注意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从机。

代码清单：主应用程序中发送串口接收的数据给从机

```

1. case APP_UART_DATA_READY:
2.     UNUSED_VARIABLE(app_uart_get(&data_array[index])); //获取串口接收的数据
3.     index++; //数据长度加1
4.     //串口数据接收完成
5.     if ((data_array[index - 1] == '\n') || (index >= (m_ble_uarts_max_data_len)))
6.     {
7.         NRF_LOG_DEBUG("Ready to send data over BLE UARTS");
8.         NRF_LOG_HEXDUMP_DEBUG(data_array, index);
9.         //发送数据给从机
10.        do
11.        {
12.            ret_val = ble_uarts_c_string_send(&m_ble_uarts_c, data_array, index);
13.            if ( (ret_val != NRF_ERROR_INVALID_STATE) && (ret_val != NRF_ERROR_RESOURCES) )
14.            {
15.                APP_ERROR_CHECK(ret_val);
16.            }
17.        } while (ret_val == NRF_ERROR_RESOURCES); //如发送队列空间不足，一直执行发送操作
18.
19.
20.        index = 0; //数据长度清零
21.    }
22. break;

```

5.5. 使能 Notify 和接收数据

从机中串口透传服务的 TX 特征的性质是 Notify（通知），因此需要写描述符 CCCD 的值为“0x01”，即使能通知后才能接收从机发送的 Notify。使能 Notify 是在串口透传客户端事件处理函数的“BLE_UARTS_C_EVT_DISCOVERY_COMPLETE”事件下执行的，即服务发现完成后使能 Notify。

写 CCCD 和写特征值一样，同样是使用 BGQ 请求完成的，写 CCCD 的代码清单如下。和写特征值代码对比可以发现，他们的操作方式是一样的，只是设置的句柄和写入的数据不一样。

代码清单：使能 Notify（写 CCCD）

```

1. static uint32_t cccd_configure(ble_uarts_c_t * p_ble_uarts_c, bool notification_enable)
2. {
3.     //定义 BGQ 请求结构体变量
4.     nrf_ble_gq_req_t cccd_req;
5.     //取得写入的数据（使能/关闭 Notify）
6.     uint8_t         cccd[BLE_CCCD_VALUE_LEN];
7.     uint16_t        cccd_val = notification_enable ? BLE_GATT_HVX_NOTIFICATION : 0;
8.     //初始化参数前，先清零 BGQ 请求结构体变量
9.     memset(&cccd_req, 0, sizeof(nrf_ble_gq_req_t));
10.    cccd[0] = LSB_16(cccd_val);
11.    cccd[1] = MSB_16(cccd_val);
12.    //初始化写入参数
13.    cccd_req.type           = NRF_BLE_GQ_REQ_GATTC_WRITE; //BGQ 写请求
14.    cccd_req.error_handler.cb = gatt_error_handler; //GATTC 和 BGQ 处理事件句柄
15.    cccd_req.error_handler.p_ctx = p_ble_uarts_c;
16.    //属性句柄设置为客户端特征配置描述符(CCCD)的句柄
17.    cccd_req.params.gattc_write.handle = p_ble_uarts_c->handles.uarts_tx_cccd_handle;
18.    cccd_req.params.gattc_write.len   = BLE_CCCD_VALUE_LEN;//写入的数据长度
19.    cccd_req.params.gattc_write.offset = 0;//偏移量设置为 0
20.    cccd_req.params.gattc_write.p_value = cccd;//指向写入的数据
21.    cccd_req.params.gattc_write.write_op = BLE_GATT_OP_WRITE_REQ;//写入请求，有响应
22.    cccd_req.params.gattc_write.flags = BLE_GATT_EXEC_WRITE_FLAG_PREPARED_WRITE;//执行准备好的
写
23.    //执行写操作
24.    return nrf_ble_gq_item_add(p_ble_uarts_c->p_gatt_queue, &cccd_req, p_ble_uarts_c->conn_handle
);
25. }
```

事件中接收数据使能 Notify 之后，从机即可向主机发送数据（Notify）。正如前文所述，BLE 发送有专用的 API 函数，但是接收数据是没有专用的 API 的，接收数据是通过接收到协议栈提交的事件完成的。

通过“5.3.1 节”中注册的 BLE 事件观察者，串口透传客户端可以获取 BLE_GATTC_EVT_HVX 事件（通知或指示事件），之后串口透传客户端处理该事件并向应用程序发送“BLE_UARTS_C_EVT_NUS_TX_EVT”事件通知应用程序接收数据（调用串口透传客户端事件处理函数），主应用程序中接收数据后，通过串口输出数据。

6. 硬件连接

同“实验 3-1”。

7. 测试步骤

1. 解压“…\5：开发指南（下册-主机）配套实验源码\”目录下的压缩文件“实验 8-1：串口透传主机”，将解压后得到的文件夹“ble_app_uarts_c”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_c\project\mdk5”目录下的工程“ble_app_uarts_c.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。
7. 程序运行后，主机开发板上的指示灯 D1 闪烁指示正在扫描。
8. 将作为从机的开发板烧写《开发指南下册-从机》的“实验 7-3：串口透传长包传输-判断 notify”。程序运行后，从机开发板上的指示灯 D1 闪烁指示正在广播。
9. 主机和从机分别使用 USB 线连接到电脑，电脑上打开串口调试助手，分别选择主机和从机对应的串口号，设置波特率为 115200bps，之后点击串口调试助手上的“打开串口”按钮。
10. 主机和从机上电运行后，可以观察到，主从机的指示灯 D1 均由闪烁变为常亮，指示连接建立成功。
11. 在和主机连接的串口调试助手发送栏中输入发送的数据并勾选“发送新行”，之后点击发送按钮发送数据，观察和从机连接的串口调试助手，应能收到主机发送的数据。
12. 在和从机连接的串口调试助手发送栏中输入发送的数据并勾选“发送新行”，之后点击发送按钮发送数据，观察和主机连接的串口调试助手，应能收到主机发送的数据。

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。