初学者教程 II: BLE Service

这篇教程的目的是了解 BLE、SDK 的基础知识，我们会尽量简化的建立并运行你的第一个 service。同时，这个教程也让你能用 BLE 的一些功能，同时也能让你探索更多的用法。我们不会学习 BLE 协议的细节 ,但是我们会快速的学习建立一个 SERVICE 所需要掌握的知识。

内容提要：

1、必要的基础知识

2、一些基础理论：

- The Generic Attribute Profile (GATT)
- Services
- Characteristics
- Universally Unique ID (UUID)

3、例程

硬件及软件准备

- nRF51 DK or nRF52 DK Kit
- nRF51 Dongle
- Keil V5.12 or later.
- Master Control Panel (MCP). 可以用电脑版或者手机版（iOS 手机用 lightblue）
- nRFgo Studio

- SDK V11.0.0
- For nRF51 DK use SoftDevice S130 V2.0.0 for nRF52 DK use S132 V2.0.0.

必备的基础知识：

这篇教程是“初学者教程 I：BLE 广播”的延续，学习这篇之前，最好先学习一下上一篇。

如上一篇教程一样，这篇教程也默认你会使用 KEIL、MCP 和 nRFgo Studio 及编译及下载程序到开发板及使用 MCP——不会用的用户请参考《视频教程》。

基础理论：

The Generic Attribute Profile (GATT)

在 The Bluetooth Core Specification 里是这样定义 GATT 的：

> *“The GATT Profile specifies the structure in which profile data is exchanged. This structure defines basic elements such as services and characteristics, used in a profile.”*

换句话说，GATT 协议是 BLE 中数据捆绑、推送、交换的一系列规则。可以参考 Bluetooth Core Specification v4.2（网盘中），Vol. 3, Part G 。这个可能不容易阅读，但是读完以后会有所收获。

Services

The Bluetooth Core Specification 是这样定义 Service 的：

> *"A service is a collection of data and associated behaviors to accomplish a particular function or feature. [...] A service definition may contain [...] mandatory characteristics and optional characteristics."*

换句话说，service 是信息的集合，比如说传感器的值。Bluetooth SIG 事先定义了一些特别的 service。比如说他们定义了 Heart Rate service。他们这样做是为了让开发者更容易做 APP 及固件来匹配标准的 Heart Rate service。但是，这并不是说你不可以用你自己的 service 架构的 heart rate 传感器。有时候人们会错误的认为他们只能准守 Bluetooth SIG 事先规定好的 service 来做他们的应用。用你自己的 service 来匹配你自己的应用也是完全没问题的。

Characteristics

The Bluetooth Core Specification 是这样定义 characteristic 的：

> *"A characteristic is a value used in a service along with properties and configuration information about how the value is accessed and information about how the value is displayed or represented."*

换句话说，characteristic 代表了实际的值和信息。安全系数、单位及其他的关于信息的元数据，也压缩在 characteristic 里。

就好比说有一个储藏室有很多文件柜，每个文件柜又有很多的抽屉。GATT profile 是这个储藏室，柜子是 service，装了各种信息的抽屉是 characteristic。有一些抽屉可能会上锁来限制读取他们的信息。

假如说有一个心率监控手表。这个手表至少要用到两个 service：

1. 一个 Heart rate service。它包含了三个 characteristic：
    - 一个强制的 Heart Rate Measurement characteristic 包含心率的值
    - 一个可选的 Body Sensor Location characteristic

- 一个有条件的 Heart Rate Control Point characteristic

2. 一个电池 service

- 一个强制的 Battery level characteristic.

为什么要这样来做？为什么不直接发送你需要的数据，而非要把他们捆绑到 characteristics 和 services 里呢？因为这样更具有灵活性、更有效率、更能交叉平台的兼容及更方便执行。当一个手机、安卓平板或者电脑发现一个广播心率 service 的设备时候，他们可以 100%的确定至少可以发现 heart rate measurement characteristic，这个 characteristic 是符合标准的。如果一个设备包含 2 个以上 service 的时候，你可以任意挑选一个 services 和 characteristics。通过这样绑定信息，设备可以快速发现可获取的信息并且进行必要的通讯，这样可以节约时间及功耗。

再比如说，储藏室坐落在一个小的办公室里并且有 2 个小柜子。第一个柜子给会计人员使用，抽屉里的文件都是商业的财务信息，并且是按日期分类的。抽屉上了锁，而且只有会计及高级管理层可以获取这些信息。第二个柜子是人力资源使用的，里面放了雇员的信息，并且是按字母排序的。公司的所有人员都知道储藏室的位置也知道是做什么用的，但是只有一些人员可以打开并且使用。这确保了效率、安全和秩序。

Universally Unique ID (UUID)

UUID 是一个缩写，你在 BLE 的领域里会经常看到。它是一个唯一的数字来鉴定 service 和 characteristics、descriptor 及特性。这些 ID 在空中传输，比如说，从机可以通知主机它提供的 service。为了节约传输时间及存储空间，在 nRF51 有 2 种 UUID：

第一种类型是短的 16-bit UUID。比如说，事先定义的 Heart rate service，UUID 是 0x180D 它附加的 characteristic： Heart Rate Measurement characteristic，UUID 是 0x2A37。

16-bit UUID 可以节约功耗及存储，但是他们只能提供相当有限的数字，所以你只能空中传输事先定义 Bluetooth SIG UUID。所以，你可能会需要第二种类型的 UUID 以便传输你自己定义的 UUID。

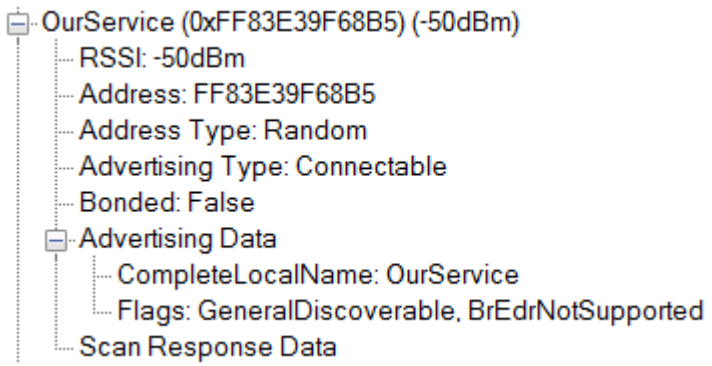
第二种类型是 128-bit UUID，有时会被认为是提供商的的特定 UUID。当你要定制自己的 services 和 characteristics 时，你需要用这种的 UUID。它的结构是像这样： 4A98xxxx-1CC4-E7C1-C757-F1267DD021E8，它叫 base UUID。4 个 x 代表你可以放入你自己的 16-bit ID 来定制你的 services 和 characteristics，可以像事先定义好的 UUID 那样用。这种方法你可以一次储存 base UUID 在 memory 里，不用它，也可以正常使用 16-bit ID。你可以用 nRFgo Studio 生成 UUID。

有趣的是，世界上没有数据库来保证全世界没有 2 个完全相同的 UUID。当你随机产生 2 个 128-bit UUID 时候，有 1/3,000,000,000,000,000,000,000,000,000,000,000,000,000,000 的几率，你会生成了 2 个完全相同的 ID。

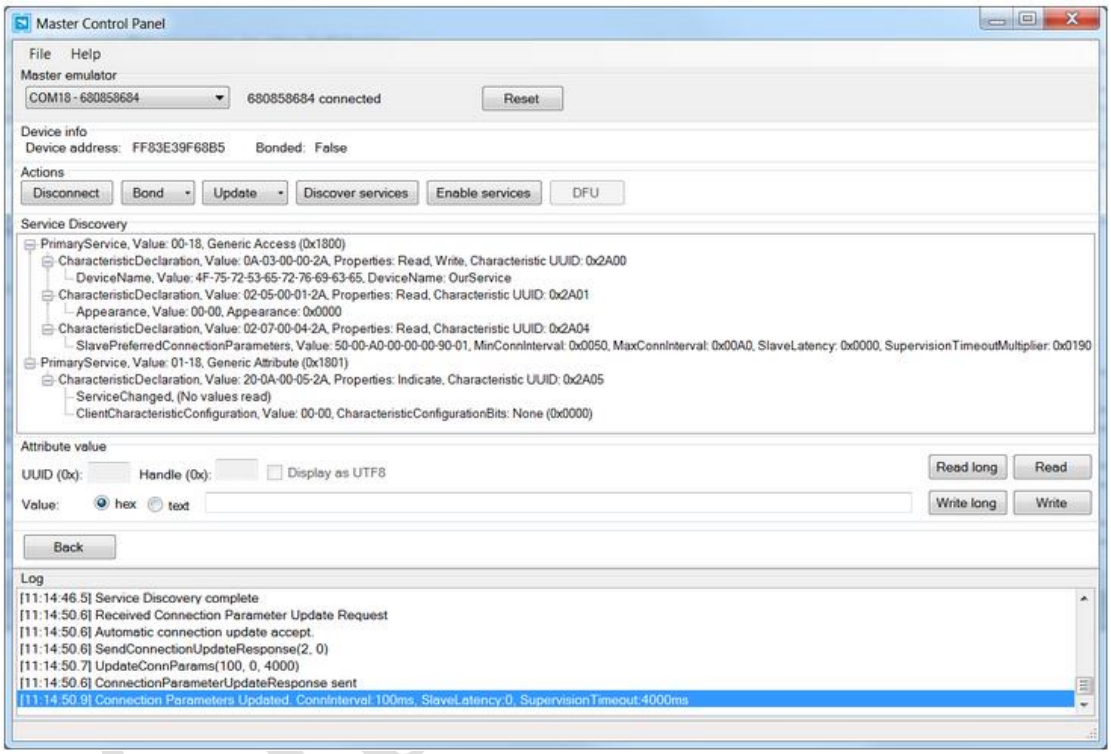
The example

准备工作

首先从网盘相同文件夹下下载好例程，这个例程是以 SDK 为模板的，但是去掉了与我们的教程无关的不必要的代码。把文件夹复制到 SDK_folder\examples\ble_peripheral 下，打开 nrf51-ble-tutorial-service.uvprojx 文件，如果你用 SEGGER Real-Time Terminal（RTT），可以看到程序的运行。这个例程编译起来应该不会有 errors，但是可能会有一些 warnings，可以不用管，点击 build 然后 load 进你的开发板。在 MCP 里看的话，会得到以下的信息：



如果点击“Select device”，然后点击“Discover services”，你可以看到：



这里你可以看到，我们只是下了程序到板子，就有 2 个强制的 services：

The Generic Access service. Service UUID 0x1800。三个强制的 characteristics:

1、 characteristics: Device name。UUID 0x2A00。

2、 characteristics: Appearance。UUID 0x2A01。

3、 characteristics: Peripheral Preferred Connection Parameters。UUID 0x2A04。

The Generic Attribute service。UUID 0x1801。一个可选 characteristic:

1、Characteristic: Service Changed. UUID 0x2A05.

The Generic Access Service 包含 device 的通用信息。你可以看到一个 characteristic 有 device name “OurService”。第二个 characteristic 有 appearance value，由于我们没有设置值，所以它只是显示 0x0000。第三个 characteristic 有不同的 parameter 用来建立连接。你可以看到此例中这些值来自 #defines 叫做 MIN_CONN_INTERVAL, MAX_CONN_INTERVAL, SLAVE_LATENCY, CONN_SUP_TIMEOUT。附录 1 有这些参数的简介，可以参考。

第二个service是Generic Attribute Service。简单的说，这个service可以用来通知service基本架构的主要改变。

Our first service

可以看到此例中含了 2 个 files: our_service.h 和 our_service.c。这里我先声明一些空函数和一小部分架构以便我们更容易开始。我还用了 nRFgo Studio 来创建和定义一个 128-bit base UUID，定义为 BLE_UUID_OUR_BASE_UUID。在工程中搜索 STEP 1-7，你可以找到我们需要增加代码的地方。

第一步：声明一个 service structure

首先我们需要一个地方来储存和我们 service 相关的数据和信息，我们需要用到 `ble_os_t` 架构。可以看到在 our_service.h 中，架构只能有一个 entry。 `service_handle` 是一个鉴定这个特殊 service 的 number，而且由 SoftDevice 指派值。在 main.c 中声明一个 `ble_os_t` 型的 `m_our_service` 的变量。所以我们可以把它给变量函数并且完全控制我们的 service。

第二步：初始化 service

在 main.c 中已经有一个函数叫做 `services_init()` 。在这个函数里，我们叫 `our_service_init()`。它做为一个参数带有一个指针到 `ble_os_t` 架构，所以确保你指针到我们的 `m_our_service` 变量：

```
our_service_init (&m_our_service);
```

第三步：加 UUID 到 BLE stack table

在 our_service.c 查一下 `our_service_init()` 的定义。可以看到这里几乎没有代码，所以我们要增加一些。我们必须先给我们的 service 建立一个 UUID。因为我们打算做一个定制的 service，所以我们要用定义好的 16-bit UUID 一起。加入以下代码：

```
    uint32_t   err_code;
    ble_uuid_t        service_uuid;
    ble_uuid128_t     base_uuid = BLE_UUID_OUR_BASE_UUID;
    service_uuid.uuid = BLE_UUID_OUR_SERVICE;
    err_code = sd_ble_uuid_vs_add(&base_uuid, &service_uuid.type);
    APP_ERROR_CHECK(err_code);
```

这些代码是来创建 2 个变量。一个有我们的 16-bit service UUID ,另一个是其他的 UUID。在第四行 BLE stack 中我们加入 our vendor specific UUID（因此有个 vs）到 UUID 的 table 中。最后在第五行我们做一个 quick error check。在所有的函数调回和返回一些 error code 后，这是一个很好的习惯。附录 2 解释了怎么 debug `APP_ERROR_CHECK()`。

第四步：增加我们的 service

现在我们准备好来初始化我们的 service 了。在上面的代码下再增加这些代码：

```
    err_code = sd_ble_gatts_service_add(BLE_GATTS_SRVC_TYPE_PRIMARY,
                                        &service_uuid,
                                        &p_our_service->service_handle);
    APP_ERROR_CHECK(err_code);
```

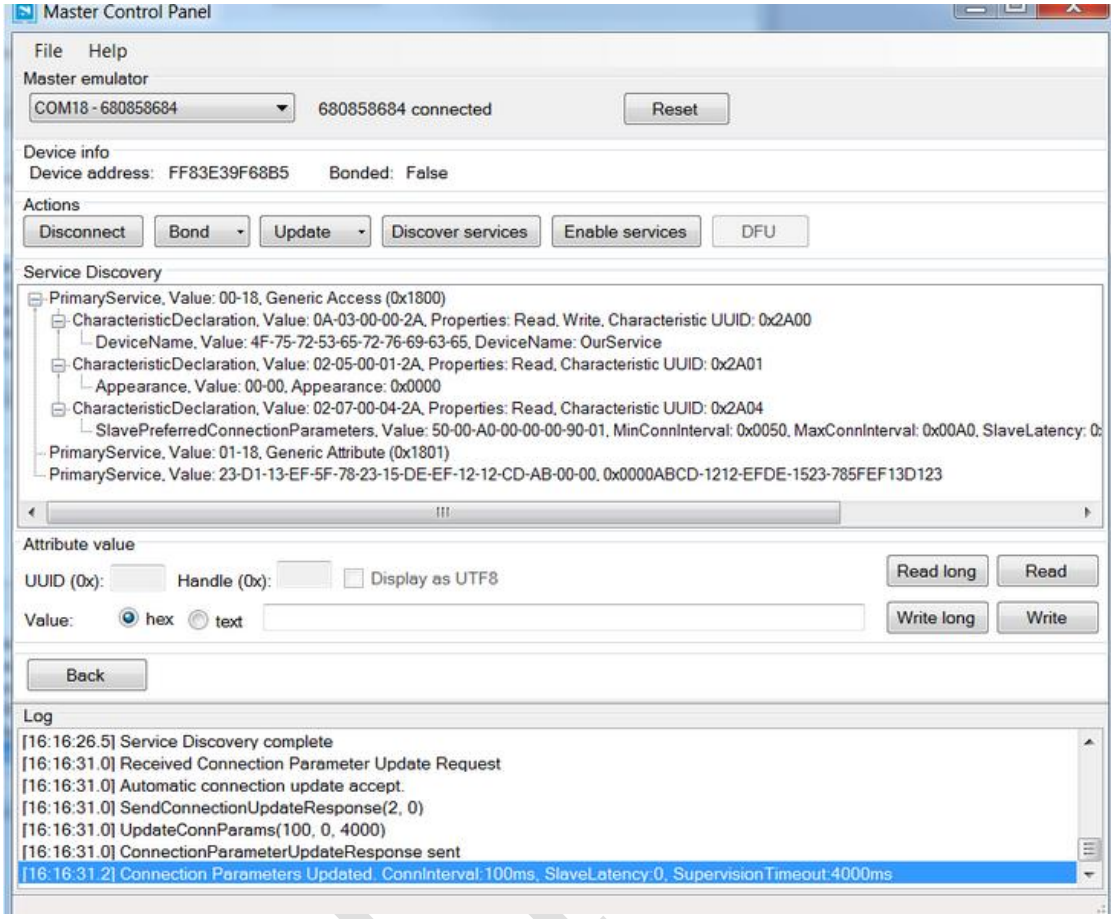
`sd_ble_gatts_service_add()` 函数有 3 个参数。第一个参数，我们指明了需要一个基本的

service。另外一个选择是用 BLE_GATTS_SRVC_TYPE_SECONDARY 来建立第二个 service。第二个 service 用的比较少，但是有些时候用来把 services 加到另一些 service 上。第二个参数指向我们创建的 service UUID。通过传输变量到 `sd_ble_gatts_service_add()`，我们的 service 可以被 BLE stack 识别。第三个传输给函数的变量指向 service_handle 数据储存的地方。`sd_ble_gatts_service_add()` 函数会创建一个 table 包含我们的 service，在 table 中，service_handle 是一个检索指向我们特别的 service。`our_service_init()` 现在应该是这样：

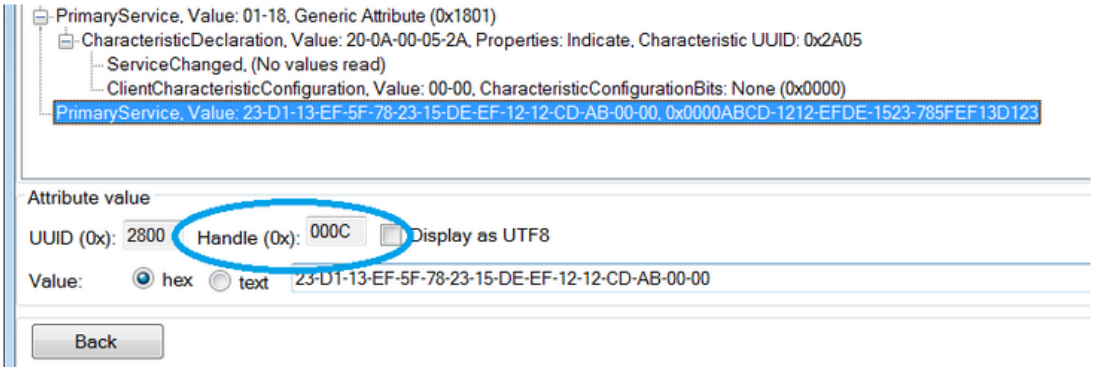
```
    void our_service_init(ble_os_t * p_our_service)
    {
        uint32_t   err_code; // Variable to hold return codes from library and softde
vice functions

        // OUR_JOB: Declare 16-bit service and 128-bit base UUIDs and add them to the
 BLE stack
        ble_uuid_t        service_uuid;
        ble_uuid128_t     base_uuid = BLE_UUID_OUR_BASE_UUID;
        service_uuid.uuid = BLE_UUID_OUR_SERVICE;
        err_code = sd_ble_uuid_vs_add(&base_uuid, &service_uuid.type);
        APP_ERROR_CHECK(err_code);

        // OUR_JOB: Add our service
        err_code = sd_ble_gatts_service_add(BLE_GATTS_SRVC_TYPE_PRIMARY,
                                            &service_uuid,
                                            &p_our_service->service_handle);
        APP_ERROR_CHECK(err_code);

        SEGGER_RTT_WriteString(0, "Exectuing our_service_init().\n");
        SEGGER_RTT_printf(0, "Service UUID: 0x%#04x\n", service_uuid.uuid);
        SEGGER_RTT_printf(0, "Service UUID type: 0x%#02x\n", service_uuid.type);
        SEGGER_RTT_printf(0, "Service handle: 0x%#04x\n", p_our_service->service_hand
le);
    }
```

编译，下载你的编码，打开 MCP。点击 connect，在点击 service discovery。然后你应该在底部看到我们的 service 带有定制的 UUID。你可以在 our_service.h 从#define 看到

base UUID，如果你仔细看的话会发现我们 16-bit service UUID：0000 **ABCD** -1212-EFDE-1523-785FEF13D123。



如果你打开 Segger RTT，你会看到应用相关的一些信息。**记得 uncomment 在 our_service_init()底部的四行** SEGGER_RTT。当程序开始，你可以看到在我们的 service 中用的值，service handle 设置到 0x000C。如果你在 MCP 中 highlight 我们的 service，你可以看到如下值：



Advertising

在上一篇教程，“Advertising”，我们讨论了不同的广播包，现在来广播我们的 base UUID。因为我们的 base UUID 是 16 bytes 长。广播包已经包含了一些数据，所以广播包中没有足够的空间了。所以我们需要把它放在 scan response 包中。

第五步：申明包含我们 service UUID 的变量

在 main.c 中，`advertising_init()` 里，申明一个包含我们 UUID 的变量：

```
ble_uuid_t m_adv_uuids[] = {BLE_UUID_OUR_SERVICE, BLE_UUID_TYPE_VENDOR_BEGIN};
```

BLE_UUID_OUR_SERVICE 是我们的 service UUID，BLE_UUID_TYPE_VENDOR_BEGIN 表明了一个 vendor specific base 的 UUID 的一部分。它更多的是指向初始化的 `our_service_init()` 里 UUIDtable 的 our base UUID 的索引。

第六步：申明及例举 scan response

申明及例举 scan response：

```
ble_advdata_t srdata;
memset(&srdata, 0, sizeof(srdata));
```

然后增加 UUID 到 scan response 包里：

```
srdata.uuids_complete.uuid_cnt = sizeof(m_adv_uuids) / sizeof(m_adv_uuids[0]);
srdata.uuids_complete.p_uuids = m_adv_uuids;
```

第七步：包含 scan response 的广播

最后初始化带 scan response 的广播：

```
err_code = ble_advertising_init(&advdata, &srdata, &options, on_adv_evt, NULL);
```

`advertising_init()` 应该是这样：

```
    static void advertising_init(void)
    {
        uint32_t      err_code;
        ble_advdata_t advdata;

        // Build advertising data struct to pass into ble_advertising_init().
        memset(&advdata, 0, sizeof(advdata));

        advdata.name_type               = BLE_ADVDATA_FULL_NAME;
        advdata.flags                   = BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE;

        ble_adv_modes_config_t options = {0};
        options.ble_adv_fast_enabled  = BLE_ADV_FAST_ENABLED;
        options.ble_adv_fast_interval = APP_ADV_INTERVAL;
        options.ble_adv_fast_timeout  = APP_ADV_TIMEOUT_IN_SECONDS;

        // OUR_JOB: Create a scan response packet and include the list of UUIDs
        ble_uuid_t m_adv_uuids[] = {BLE_UUID_OUR_SERVICE, BLE_UUID_TYPE_VENDOR_BEGI
N};

        ble_advdata_t srdata;
        memset(&srdata, 0, sizeof(srdata));
        srdata.uuids_complete.uuid_cnt = sizeof(m_adv_uuids) / sizeof(m_adv_uuids
[0]);
        srdata.uuids_complete.p_uuids = m_adv_uuids;

        err_code = ble_advertising_init(&advdata, &srdata, &options, on_adv_evt, NUL
L);
        APP_ERROR_CHECK(err_code);
    }
```

再一次编译、下载。我们的设备现在在 MCP 中应该是这样显示：

总结

现在，我们知道怎么创建你的第一个 service。如果你想要增加更多的 service，你只需简单的重复 `our_service_init()` 函数，及定义更多的 service UUID。

但是储存你所有数据的 characteristics 在哪呢？我们在下一篇中会说到。

附录 1

The connection parameters for a BLE connection is a set of parameters that determine when and how the Central and a Peripheral in a link transmits data. It is always the Central that actually sets the connection parameters used, but the Peripheral can send a so-called Connection Parameter Update Request, that the Central can then accept or reject.

There are basically three different parameters:

Connection interval: Determines how often the Central will ask for data from the Peripheral. When the Peripheral requests an update, it supplies a maximum and a minimum wanted interval. The connection interval must be between 7.5 ms and 4 s.

Slave latency: By setting a non-zero slave latency, the Peripheral can choose to not answer when the Central asks for data up to the slave latency number of times. However, if the Peripheral has data to send, it can choose to send data at any time. This enables a peripheral to stay sleeping for a longer time, if it doesn't have data to send, but still send data fast if needed. The text book example of such device is for example keyboard and mice, which want to be sleeping for as long as possible when there is no data to send, but still have low latency (and for the mouse: low connection interval) when needed.

Connection supervision timeout: This timeout determines the timeout from the last data exchange till a link is considered lost. A Central will not start trying to reconnect before the timeout has passed, so if you have a device which goes in and out of

range often, and you need to notice when that happens, it might make sense to have a short timeout.

Depending on which platform you're working with, there can be platform specific recommendations or requirements on these. For iOS, Apple maintains a "Bluetooth Accessory Design Guidelines" document, which among other things includes rules on these parameters.

附录 2

In the SDK examples, we check return codes with the macro APP_ERROR_CHECK.

This function calls the function `app_error_handler` in app_error.c. Here we find this code:

```
#ifndef DEBUG
    NVIC_SystemReset();
#else
```

This means that by default, the response to ANY return code other than NRF_SUCCESS will cause a system reset.

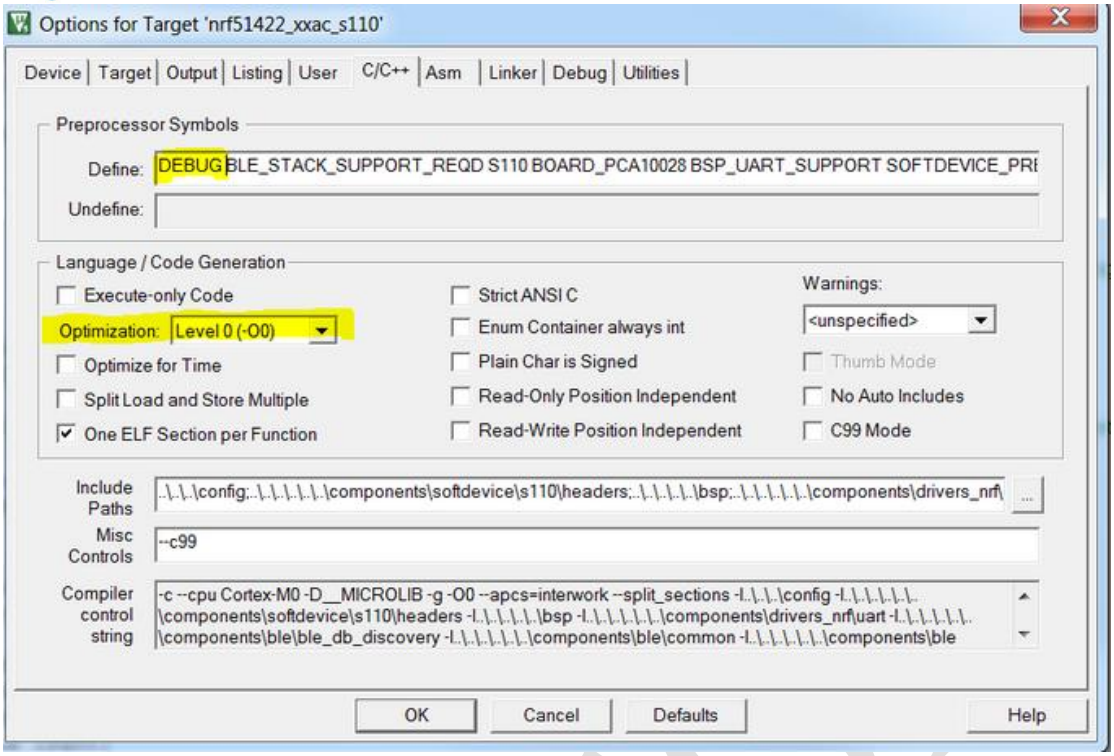
If we have defined DEBUG, this happens:

```
m_error_code = error_code;
m_line_num = line_num;
m_p_file_name = p_file_name;
UNUSED_VARIABLE(m_error_code);
UNUSED_VARIABLE(m_line_num);
UNUSED_VARIABLE(m_p_file_name);
__disable_irq();
while(loop);
```

We disable interrupts and stay in a loop forever. In this case we can either print out or read (with a debugger) the error code, line number and file name that returned the error code.
Note the warning in the documentation of app_error_handler:

@warning This handler is an example only and does not fit a final product. You need to analyze * how your product is supposed to react in case of error.

Finding the error with the Keil debugger:
1.Set optimization level 0 in target options -> C/C++ and define DEBUG in the preprocessor symbols.



2.Place a break point next to while(loop); (right click ->Insert breakpoint)

3.Run the program, and wait for the error to occur.

4.Read the information by hovering the mouse cursor over the value. Alternatively you can add the values to a watch by doing right click->add to watch X. To display the values in decimal format rather than in hexadecimal, right click the variable (in watch window) and click on "Hexadecimal Display".