青风带你玩蓝牙 nRF51822 系列教程

----------作者：青风

出品论坛：　www.qfv8.com　青风电子社区

作者： 青风
出品论坛: www.qfv8.com
淘宝店：http://qfv5.taobao.com
QQ 技术群：346518370
硬件平台：青云 QY-nRF52832 开发板

2.4 蓝牙 BLE 之 LED 灯任务的读写

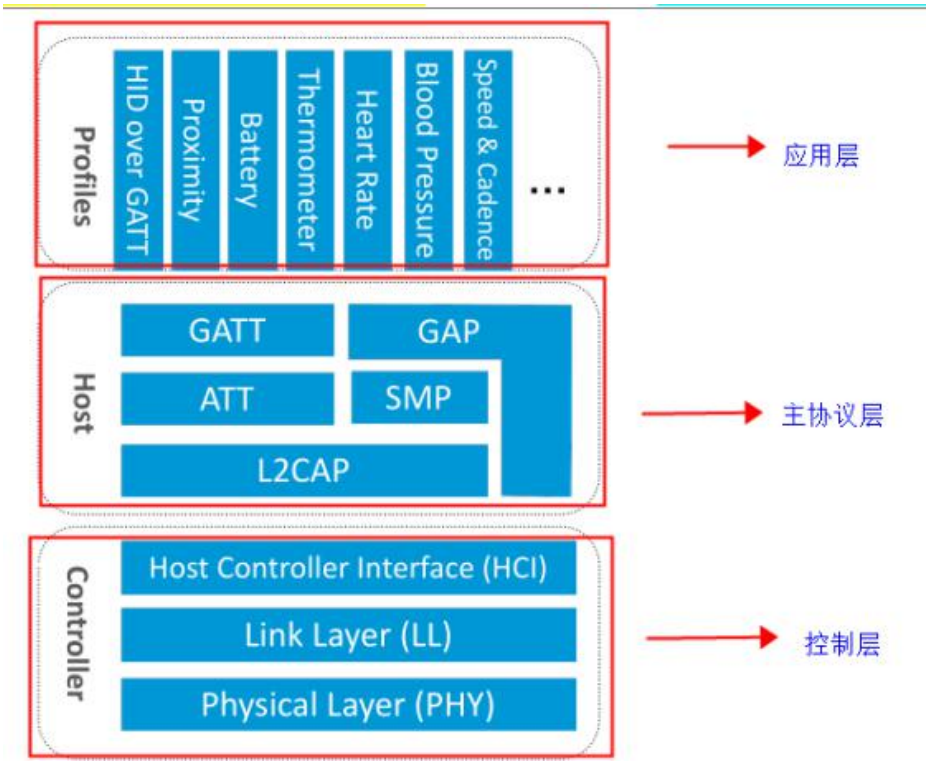
今天 BLE LED 这个例子应用示例是为了让你学习如何在 nRF52832 上开发 BLE 应用，它是一个通过 BLE 的空中读和写控制属性功能进行通信的 BLE 应用的简单演示。当程序运行时，你可以通过主机控制 nRF52832 开发板上 LED 的输出点亮。

这个应用实际上是通过一个服务"GATT 层"服务被建立，那么这个服务应该包括 1 个特性：LED 的特性。如果有多个任务就可以设置多个特性。我们设置 LED 特性这里应该为：通过没有回应的写远程操作 LED 的亮灭，不发通知给集中器（主机）。

下面首先来介绍下几个基本的蓝牙概念：

1 nRF52832 蓝牙协议的分层：

蓝牙协议栈结构分为三层，其结构如下图 1 所示，分为应用层，主协议栈层，控制层，下面来详细进行介绍，同时结合本实例提出一些蓝牙基本概念。



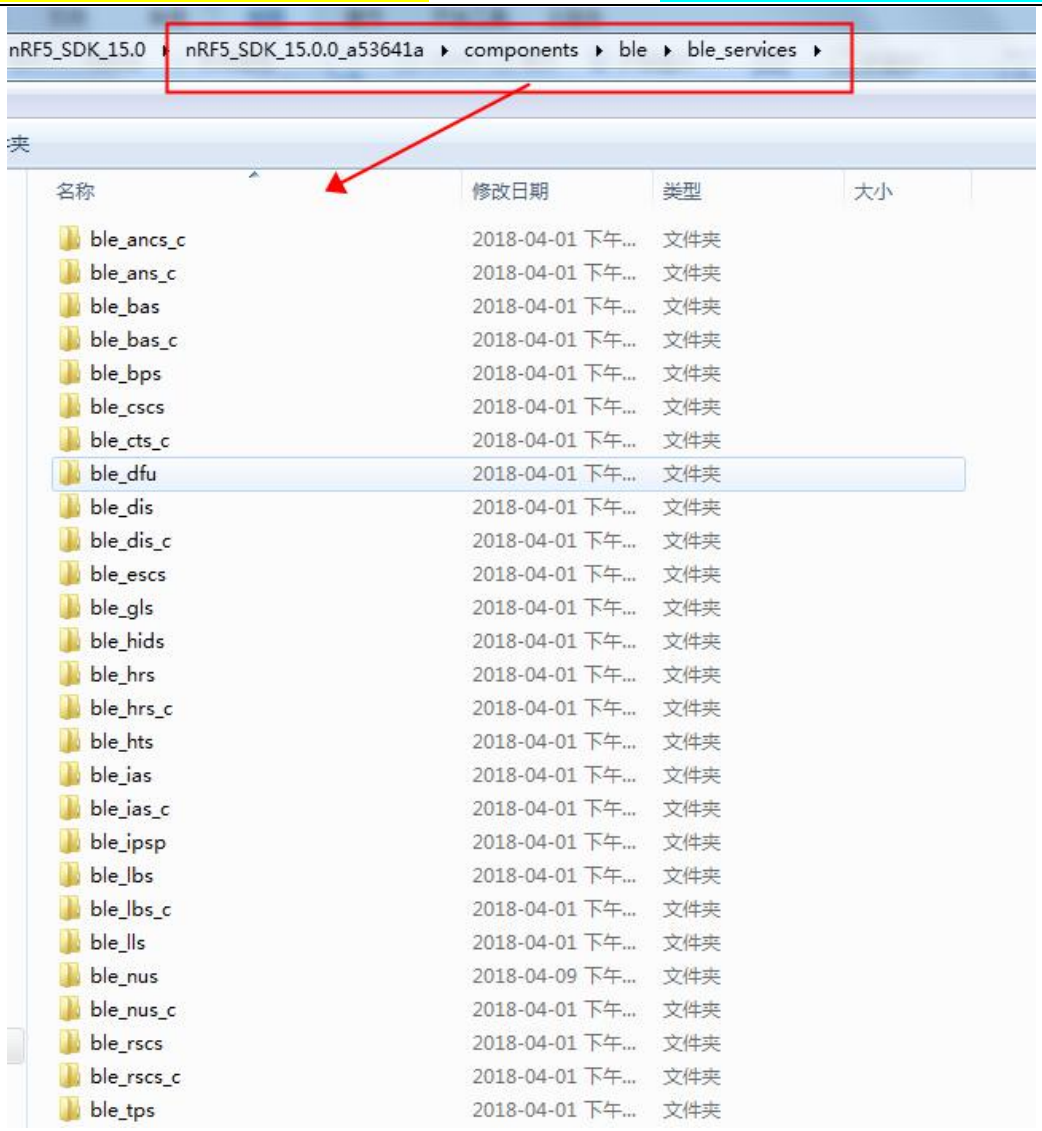
1.1 首先是应用层：

应用层 Profiles：也就是说你建立的蓝牙应用任务，蓝牙任务实际上分为两类：标准蓝牙任务规范 profile（公有任务），非标准蓝牙任务规范 profile（私有有任务）

标准蓝牙任务规范 profile：指的是从蓝牙特别兴趣小组 SIG 的官方网站上可以看到已经发布的 GATT 规范列表，包括警告通知（alert notificantion),血压测量（blood pressure),心率（heart rate),电池（battery)等等，如上图所示。它们都是针对具体的低功耗蓝牙的应用实例来设计的。目前蓝牙技术联盟还在不断的制定新的规范，并且将陆续发布。详细内容大家可以参看蓝牙特别兴趣小组 SIG 的官方网站进行了解。在 nrf52 的 SDK 开发软件包里包含了一个文件夹，定义了这些标准应用规范，如下图 2 所示。

非标准蓝牙任务规范 profile，也称为私有任务。是供应商自定义的任务，在蓝牙 SIG 小组内未定义的任务规范，比如本例所谈的蓝牙 LED 灯任务。

蓝牙服务建立这章内容，主要就来讲解两种任务规范如何建立的，如何实现一个蓝牙应用。



1.2 控制器的控制层

如上图 1 所示，实际内部分为三部分：

1：主机控制器（HCI），也称为设备管理器

设备管理器是基带中的一个功能模块，控制一般的蓝牙设备行为。它负责所有与数据无关的蓝牙系统操作，例如询问附件蓝牙设备是否存在，连接蓝牙设备，或者让本地的蓝牙设备可以被其他设备发现或者连接。

为了执行相应的功能，设备管理器要求通过基带的资源控制器访问传输媒介。同时，设备管理器还通过 HCI 命令提供本地设备行为的控制功能，例如管理设备的本地名称。存储链路密钥等。

2：链路层（LL)：

链路层主要负责链路管理，链路控制。包括负责创建，维护和释放逻辑链路已经更新设备之间物理链路的相关参数。

3：物理层：

物理层模块负责从物理信道传输和接收信息数据包。在基带和物理层之间，一条控制路允许基带模块控制物理层的时隙和频率载波。同时，物理层模块向物理信道和基带发送和接收符合格式要求的数据流。

2 主机主协议层详细说明：

本实验主要就是编写主机层，因此下面需要来详细说明：

2.1 通用访问规范（Generic Access Profile，GAP）

GAP 是应用层能够直接访问 BLE 协议栈的最底层，它包括管理广播和连接事件的有关参数。GAP 模块代表了所有蓝牙设备的共用基础功能，如传输，协议或者应用规范所使用的模式和访问过程。GAP 的服务包括设备发现，连接方式，安全，认证，关联模型和服务发现等。我们会在专门写一篇文章**《GAP 详解》**里进行介绍。

注意:GAP 的更多详细介绍见《Bluetooth Core Specification》（蓝牙核心规范）的第 3 卷 C 部分。

2.1.1 角色

为了创建和维持一个 BLE 连接，引入了"角色"这一概念。一个 BLE 设备不是集中器角色就是外围设备角色，这是根据是谁发起这个连接来确定的。集中器设备总是连接的发起者，而外围设备总是被连接者。集中器和外围设备的关系就像链路层中的主机和从机的概念。

在 BLE LED 应用例程中，使用 S132 SoftDevice 烧录到 nRF52832 作为外围设备，计算机或者手机作为集中器（主机）。

除了集中器角色和外围设备角色，蓝牙核心规范还定义了观察者角色和广播者角色，观察者角色监听空中的事件（抓包器），广播者角色只是广播信息而不接收信息（ibeacon 类似应用）。观察者角色和广播者角色都只广播而并不建立连接。它们在我们的这个应用中并不适用。

注意：在一个连接的另一端的设备被称为对等设备，不管它是集中器还是外围设备。

2.1.2 广播

集中器能够与外围设备建立连接，外围设备必须处于广播状态，它每经过一个时间间隔发送一次广播数据包，这个时间间隔称为广播间隔，它的范围是 20ms 到 10.24s。广播间隔影响建立连接的时间。

集中器发送一个连接请求来发起连接之前，必须接收到一个广播数据包，外围设备发送一个广播数据包之后一小段时间内只监听连接请求。

一个广播数据包最多能携带 31 字节的数据，它通常包含用户可读的名字、关于设备发送数据包的有关信息、用于表示此设备是否可被发现的标志等类似的标志。

当集中器接收到广播数据包后，它可能发送请求更多数据包的请求，称为扫描回应，如果它被设置成主动扫描，外围设备将会发送一个扫描回应做为对集中器请求的回应，扫描回应最多可以携带 31 字节的数据(《广播初始化详解》中详细介绍)。

广播，包括扫描请求和扫描回应，出现在远离 WLAN 使用的 2.4G 频段之外的 3 个频率上，以防止被 WiFi 干扰。

2.1.3 扫描

扫描是集中器监听广播数据包和发送扫描请求的过程，它有 2 个定时参数需要特别注意：扫描窗口和扫描间隔。

对于每一个扫描间隔，集中器扫描的时间等于一个扫描窗口，这就意味着如果扫描窗口等于扫描间隔，那么集中器将处于连续扫描之中。扫描窗口和扫描间隔之比为扫描占空比（《主机扫描》详细讲解）。

2.1.4 发起

如果集中器想建立一个连接，当扫描监听到广播数据包后它将采用相同的过程：当要发起连接时，集中器接收到一个广播数据包之后将会发送一个连接请求。

2.1.5 连接

集中器和外围设备第一次交换数据定义为连接状态。在一个连接状态中，集中器将会在一个特定定义的间隔从外围设备请求数据，这个间隔称为连接间隔，它由集中器决定并应用于连接，但是外围设备可以发送连接参数更新请求给集中器。根据蓝牙核心规范，连接间隔必须在 7.5ms 到 4s 之间。

如果外围设备在一个时间帧内没有回应集中器的数据包，称为连接监管超时，连接被认为丢失。

可以通过在每一个连接间隔中传输多个数据包以获得更高的数据吞吐量，蓝牙 5.0 里，每一个传输数据包最多可以携带 247 个字节的应用数据。但是如果电流消耗是重点，同时外围设备也没有数据要发送，它可以选择忽略一定数量的连接间隔，这个忽略连接间隔的数目称为从机延时(slave latency)。

在一个连接中，除了广播信道，设备间在频带的所有信道中进行通信。当然对于应用层，这是完全透明的。

2.2 通用属性配置文件（Generic Attribute profile，GATT）

GATT 层是传输真正数据所在的层。包括了一个数据传输和存储框架以及其基本操作。GTTA 定义了两类角色：

服务器（server）和客户端（client）,GATT 角色无需和 GAP 角色绑定，但是可能由更高层的规范进行指定。下面来详细介绍在任务中的定义：

2.2.1 角色

除了 GAP 定义了角色之外，BLE 还定义了另外 2 种角色：GATT 服务器和 GATT 客户端，它们完全独立于 GAP 的角色。提供数据的设备称为 GATT 服务器，访问 GATT 服务器而获得数据的设备称为 GATT 客户端。

以 BLE LED 应用为例，外围设备（带有 LED）作为服务器，集中器作为客户端。

注意：一个设备可以同时作为服务器和客户端。这里采用开发板作为服务器，手机作为客户端

2.2.2 GATT 的规范层次

一个 GATT 服务器通过一个称为属性表的表格组织数据，这些数据就是用于真正发送的数据。

2.2.2.1 属性

一个属性包含句柄、UUID、值，句柄是属性在 GATT 表中的索引，在一个设备中每一个属性的句柄都是唯一的。UUID 包含属性表中数据类型的信息，它是理解属性表中的值的每一个字节的意义的关键信息。在一个 GATT 表中可能有许多属性，这些属性能可能有相同的 UUID。

2.2.2.2 特性

一个特性至少包含 2 个属性：一个属性用于声明，一个属性用于存放特性的值。

所有通过 GATT 服务传输的数据必须映射成一系列的特性，可以把特性中的这些数据看成是一个个捆绑起来的数据，每个特性就是一个自我包容而独立的数据点。例如，如果几块数据总是一起变化，那么我们可以把它们集中在一个特性里。

以 BLE LED 应用为例，外围设备（带有 LED）作为服务器，集中器作为客户端。

在 BLE LED 服务中，如果多任务，任务和任务之间没有任何联系，而且它们可以各自独立地改变， 因此，可以让它们成为独立的特性，所以我们用一个特性用来表示当前 LED 的状态。

2.2.2.3 描述符

任何在特性中的属性不是定义为属性值就是为描述符。描述符是一个额外的属性以提供更多特性的信息，它提供一个人类可识别的特性描述的实例。

然而，有一个特别的描述符值得特别地提起：客户端特性配置描述符(Client Characteristic Configuration Descriptor，CCCD)，这个描述符是给任何支持通知

或指示功能的特性额外增加的。在 CCCD 中写入"1"使能通知功能，写入"2"使能指示功能，写入"0"同时禁止通知和指示功能。

在 S132 SoftDevice 协议栈中，对任何使能了通知功能或是指示功能的特性，协议栈将自动加入这个类型的描述符。

2.2.2.4 服务

一个服务包含一个或多个特性，这些特性是逻辑上相关的集合体。

GATT 服务一般包含几块具有相关的功能，比如特定传感器的读取和设置，人机接口的输入输出。组织具有相关的特性到服务中既实用又有效，因为它使得逻辑上和用户数据上的边界变得更加清晰，同时它也有助于不同应用程序间代码的重用。GATT 基于蓝牙技术联盟(SIG)官方而设计，SIG 建议根据它们的规范设计自己的 profile。

对于 BLE LED 应用例程，把 LED 特性放到服务中。

2.2.2.5 profile（数据配置文件）

一个 profile 文件可以包含一个或者多个服务，一个 profile 文件包含需要的服务的信息或者为对等设备如何交互的配置文件的选项信息。设备的 GAP 和 GATT 的角色都可能在数据的交换过程中改变，因此，这个**文件应该包含广播的种类、所使用的连接间隔、所需的安全等级等信息。**

需要注意的是一个 profile 中的属性表不能包含另一个属性表。

在 BLE LED 示例中的 profile 不是一个标准描述的 profile。

2.2.3 标准的定制服务和特性

蓝牙技术联盟(SIG)已经定义一些 profile、服务、特性和根据协议栈的 GATT 层定义的属性。但是，协议栈中只实现了一部分应用的 BLE 服务，那就意味着，只要协议栈支持 GATT，就可能为一个应用建立一个它需要的 profile 和服务。 既然在一个应用中可以支持 profile 和服务，那么就可以在这个应用中建立一个个人定制的服务。

对于 BLE LED 这个示例来说，蓝牙技术联盟没有包含这个应用，因此它建立了一个定制的服务，包括 1 个定制的特性。

注:讲到这里大家应该清楚我们的任务就是要建立私有的 profile 了。

2.3 如何定制私有 profile 服务：

只要协议栈支持 GATT，就可能为一个应用建立一个它需要的 profile 和服务。因

此该 profile 必须符合 GATT 的规范定，这讲主要就是讲解如果建立私有服务规范。

2.3.1 私有服务的 UUID

在“GATT 层”中规范定义的所有属性都有一个 UUID 值，UUID 是全球唯一的 128 位的号码，它用来识别不同的特性。

2.3.1.1 蓝牙技术联盟 UUID

蓝牙核心规范制定了两种不同的 UUID，一种是基本的 UUID，一种是代替基本 UUID 的 16 位 UUID。

所有的蓝牙技术联盟定义 UUID 共用了一个基本的 UUID：

0x0000xxxx-0000-1000-8000-00805F9B34FB

为了进一步简化基本 UUID，每一个蓝牙技术联盟定义的属性有一个唯一的 16 位 UUID，以代替上面的基本 UUID 的‘x’部分。例如，心率测量特性使用 0X2A37 作为它的 16 位 UUID，因此它完整的 128 位 UUID 为：

0x0000**2A37**-0000-1000-8000-00805F9B34FB

虽然蓝牙技术联盟使用相同的基本 UUID，但是 16 位的 UUID 足够唯一地识别蓝牙技术联盟所定义的各种属性。

蓝牙技术联盟所用的基本 UUID 不能用于任何定制的属性、服务和特性。对于定制的属性，必须使用另外完整的 128 位 UUID。

2.3.1.2 供应商特定的 UUID

SoftDevice 根据蓝牙技术联盟定义 UUID 类似的方式定义 UUID：先增加一个特定的基本 UUID，再定义一个 16 位的 UUID（类似于一个别名），再加载在基本 UUID 之上。这种采用为所有的定制属性定义一个共用的基本 UUID 的方式使得应用变为更加简单，至少在同一服务中更是如此。

使用软件 nRFgo Studio 非常容易产生一个新的基本 UUID，具体方法参考教程**《UUID 的设置与总结》**。

例如，在 BLE LED 示例中，采用 0x0000xxxx-1212-EFDE-1523-785FEABCD123 作为基本 UUID。

蓝牙核心规范没有任何规则或是建议如何对加入基本 UUID 的 16 位 UUID 进行分配，因此你可以按照你的意图来任意分配。

例如，在 BLE LED 示例中，0x1523 作为服务的 UUID，0x1524 作为 LED 特性的 UUID。

2.3.2 空中操作和性质

大部分的空中操作事件都是采用句柄来进行的，因为句柄能够唯一识别各个属性。如何使用特性依据它的性质，特性的性质包括：

- 写
- 没有回应的写
- 读
- 通知：客户端发给请求给服务器，不需要服务器回复一个响应
- 指示：客户端发给请求给服务器，需要服务器回复一个响应

更多的性质在蓝牙规范中有明确的定义，但以上性质更为常用。

2.3.2.1 写和没有回应的写

写和没有回应的写允许 GATT 客户端写入一个值到 GATT 服务器的一个特性中。它们之间不同的地方在于没有回应的写事件没有任何应用层上的确认或回应。

2.3.2.2 读

读性质表明一个 GATT 客户端可以读取在 GATT 服务器中特性的值。

2.3.2.3 通知和指示

通知和指示性质允许 GATT 服务器在其某个特性改变的时候对 GATT 客户端进行提醒，通知和指示之间不同之处在于指示有应用层上的确认，而通知没有。

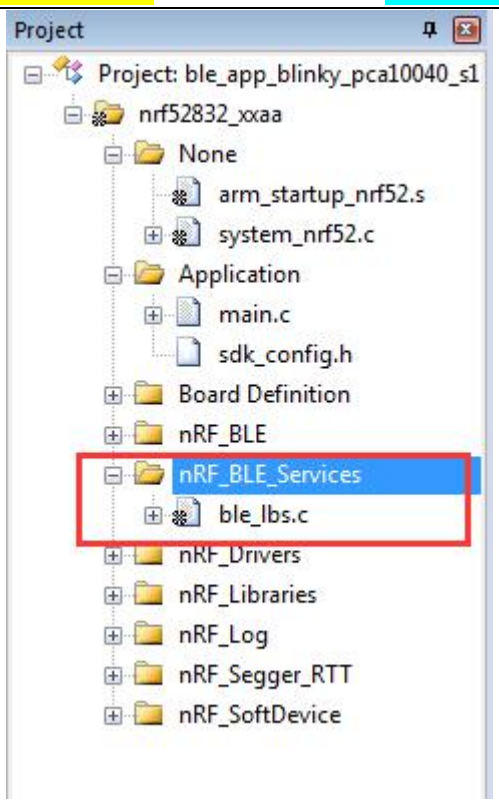
在 BLE LED 示例中，控制 LED 的特性，都是 LED 服务中的定制的特性。在 LED 特性中，集中器需要能够设置它的值和能够读取它的值。因为应用层级别的确认没有必要，因此你可以使用没有回应的写和读的性质。

注意：GATT 和它的下一层 ATT 协议在《蓝牙核心规范》第 3 卷，第 F 和 G 部分中有详细的描述。

3 工程框架的搭建

3.1 工程文件的添加

蓝牙点灯的工程项目的框架可以以蓝牙样例为基础，添加蓝牙服务函数，也就是说，我们只需要编写服务函数就可以了。服务函数包含一个.c 和.h 文件，比如本例中 ble_lbs.c 文件为蓝牙任务的驱动函数文件，ble_lbs.h 文件为对应的驱动头文件，工程目录如下所示，下面我们就来讨论下这个文件如何编写的。



在 ble_lbs.c 文件中，首先我们需要封装一个 ble_lbs_init 服务初始化函数，这个服务初始化函数里定义我们的主任务和子服务。下面就来一一展开。

3.2 主服务的添加

任何服务的添加都可以通过协议栈函数 sd_ble_gatts_service_add()进来实现。这个工作最好不要在应用层代码中建立服务，而是在一个单独的文件中建立服务，比如我们要建立一个 LED 的私有任务，单独开辟建立 ble_lbs.c 文件，在 ble_lbs_init 服务初始化函数中调用 sd_ble_gatts_service_add()来添加服务，函数如下所示。

```
err_code = sd_ble_gatts_service_add(BLE_GATTS_SRVC_TYPE_PRIMARY,
                                    &ble_uuid, &p_lbs->service_handle);
```

三个形参介解释如下：

一个服务不是主服务就是次服务，但是在通常实际的应用中大部分使用主服务。该函数第一个参数 BLE_GATTS_SRVC_TYPE_PRIMARY 声明为主服务添加。

变量 ble_uuid 就是你想用于服务的 UUID。在函数前用结构体类型进行了赋值。如下所示，指向其基础 UUID:

```
ble_uuid.type = p_lbs->uuid_type;
ble_uuid.uuid = LBS_UUID_SERVICE;
```

第三个变量 service_handle 是一个输出变量，当创建一个服务的时候将会返回一个唯一的句柄值，这个句柄可以在以后用于识别不同的服务， &p_lbs->Service_handle 也就是指向你定义的这个点灯的服务。

4 私有服务的实现

如果我们使用的服务不是蓝牙兴趣小组 SIG 定义的服务，那么就应该是一个私有服务。这个服务采用通用的方式实现定义，因此可以很容易重用于其他应用。它能够使应用程序通过初始化就能使用这个服务、处理事件、提供输入输出的实现。实现其他自定义的私有服务的方法也是类似的了。

4.1 服务数据结构体设置

用到的数据结构还没有定义：ble_lbs_t 和 ble_lbs_init_t.，我们数据结构体如下：

1.LED 服务不依赖于任何启动或停止，所以只使用一个函数作为回调函数，当 LED 特性被写入时被调用该处理，如下代码所示。这个句柄是初始化中唯一有效的参数，也是初始化结构体中唯一的成员。

```
typedef struct
{
        ble_lbs_led_write_handler_t    led_write_handler;
} ble_lbs_init_t;
```

在这个结构体中，函数类型的定义如下(在头文件中必须在 ble_lbs_init_t 定义之前添加，代替之前已经存在的事件句柄定义):

```
typedef void (*ble_lbs_led_write_handler_t) (ble_lbs_t * p_lbs, uint8_t new_state);
```

这个句柄会在主函数中调用，通过处理函数的状态 new_state 参数值来判断是否点灯。

2.第二个部分，下面的参数还需要定义：

- 服务的句柄
- LED 灯特性的句柄
- 按键特性的句柄
- UUID 类型

l LED 写的回调函数

服务结构体定义如下：

```
typedef struct ble_lbs_s
{
uint16_t                        service_handle;
ble_gatts_char_handles_t        led_char_handles;
ble_gatts_char_handles_t        button_char_handles;
uint8_t                         uuid_type;
ble_lbs_led_write_handler_t     led_write_handler;
} ble_lbs_t;
```

4.2 服务初始化

4.2.1 服务初始化 ble_lbs_init 函数的编写

进入服务初始化函数 ble_lbs_init（）中。这个函数主要就是来完成上面定义的结构体 ble_lbs_t 的调用。

●首先设置：在服务初始化结构体和服务结构体中，led_write_handler 作为 led 写入处理句柄，作为回调函数。如下代码所示，把服务初始化结构体中你的写入 handler 操作赋值给服务结构体写入 handler。

```
p_lbs->led_write_handler = p_lbs_init->led_write_handler;
```

●第二步，UUID 需要重新设置，因为本服务中将要使用一个定制（私有）的 UUID，以代替蓝牙技术联盟所定义的 UUID。

首先，先定义一个基本 UUID，一种方式是采用 nRFgo Studio 来生成：

1.打开 nRFgo Studio

2.在 nRF8001 Setup 菜单中，选择 Edit 128-bit UUIDs 选项，点击 Add new。

这就产生了一个随机的 UUID，可以用于你的定制服务中。

新产生的基本 UUID 必须以数组的形式包含在源代码中，但是只需要在一个地方用到：为了可读性，在头文件 ble_lbs.h 中以宏定义的方式添加，连同用于服务和特性的 16 位 UUID 也一起定义：

```
#define LBS_UUID_BASE { 0x23, 0xD1, 0xBC, 0xEA, 0x5F, 0x78,
0x23, 0x15, 0xDE, 0xEF,0x12, 0x12, 0x00, 0x00, 0x00, 0x00 }
#define LBS_UUID_SERVICE        0x1523
#define LBS_UUID_LED_CHAR       0x1525
```

在服务初始化中：添加基本 UUID 到协议栈列表中，就设置了服务使用这个基本 UUID。在 ble_lbs_init()中只添加一次基础服务 UUID：

```
ble_uuid128_t base_uuid = LBS_UUID_BASE;
err_code = sd_ble_uuid_vs_add(&base_uuid, &p_lbs->uuid_type);
if (err_code != NRF_SUCCESS)
{
return err_code;
}
```

以上代码段为加入一个定制的基本 UUID 到协议栈中，并且保存了返回的 UUID 类型。

当该任务服务设置主服务 UUID 时，在 ble_lbs_init()主服务声明函数中继续添加 gatt 主服务 UUID 为 LBS_UUID_SERVICE：

```
ble_uuid.type = p_lbs->uuid_type;
ble_uuid.uuid = LBS_UUID_SERVICE;
err_code =
sd_ble_gatts_service_add( BLE_GATTS_SRVC_TYPE_PRIMARY,
&ble_uuid,
&p_lbs->service_handle);
if (err_code != NRF_SUCCESS)
{
return err_code;
}
```

以上代码只是添加了一个空的服务，所以还必须添加特性到服务中，下面的将介绍如何添加特性。

4.2.2 控制 LED 特征属性的建立

LED 状态特性需要能够可读可写，但没有任何通知功能：

1.命名 Led 特性名为 led_char_add。

2.设置 GATT 特征值特性,增加写的性质（给这个特性使能写性质）和读属性。

```
    char_md.char_props.write = 1;//无回复的写
    char_md.char_props.read  = 1;//读
    char_md.p_char_user_desc  = NULL;
    char_md.p_char_pf         = NULL;
    char_md.p_user_desc_md    = NULL;
    char_md.p_cccd_md         = NULL;
    char_md.p_sccd_md         = NULL;
```

3. 使用的子特征值 16 位 UUID 为 LBS_UUID_LED_CHAR：

```
ble_uuid.type = p_lbs->uuid_type;
ble_uuid.uuid = LBS_UUID_LED_CHAR;
```

4. 设置 GATT 属性，GATT 表中可能有许多属性，下面结合代码说明下有哪些属性：

```
    memset(&attr_md, 0, sizeof(attr_md));

BLE_GAP_CONN_SEC_MODE_SET_OPEN(&attr_md.read_perm);
BLE_GAP_CONN_SEC_MODE_SET_OPEN(&attr_md.write_perm);
    attr_md.vloc      = BLE_GATTS_VLOC_STACK;
    attr_md.rd_auth = 0;
    attr_md.wr_auth = 0;
    attr_md.vlen      = 0;

    memset(&attr_char_value, 0, sizeof(attr_char_value));

    attr_char_value.p_uuid      = &ble_uuid;
    attr_char_value.p_attr_md = &attr_md;
```

```
    attr_char_value.init_len  = sizeof(uint8_t);
    attr_char_value.init_offs = 0;
    attr_char_value.max_len   = sizeof(uint8_t);
    attr_char_value.p_value   = NULL;
```

p_uuid:　　属性 UUID 的指针

p_attr_md：属性元数据结构指针

init_len：　属性字节长度

init_offs：　属性字节的偏移量

max_len：　最大的长度

p_value ：　指向属性数据的指针

5. 保存返回的变量 led_char_handles（LED 特性的句柄），特性可以通过 sd_ble_gatts_characteristic_add()函数进行添加，它有 4 个参数。为了代码清晰，这个函数应该只能出现在服务文件中，而不能出现在应用层中。

第 1 个参数是特性要加入的服务的句柄，第 2 个参数是特性的结构体，它是一个全局变量，它包含了特性可能用到的性质（读，写，通知等）。第 3 个参数是值属性的描述，它包含了它的 UUID，长度和初始值。第 4 个参数是返回的特性和描述符的唯一句柄，这个句柄可以在以后用于识别不同的特性。例如，在写事件中用于识别哪一个特性被写入。

```
err_code = sd_ble_gatts_characteristic_add( p_lbs->service_handle,
                                            &char_md,
                                            &attr_char_value,
                                            &p_lbs->led_char_handles);
```

创建了增加特性的函数之后，你可以在服务初始化 ble_lbs_init()的末尾调用它们，如下所示：

```
// Add characteristics
err_code = led_char_add(p_lbs, p_lbs_init);
if (err_code != NRF_SUCCESS)
{
return err_code;
```

```
}
return   NRF_SUCCESS;
```

上面就把主服务和子服务特征值和特征值属性添加完成了。

4.3 处理协议栈事件

当协议栈需要通知应用程序一些有关它的事情的时候，协议栈事件就发生了，例如当写入特性或是描述符时。对应于本应用，你需要写入 LED 特性，为了让通知功能更好地工作，你需要保存连接句柄，通过这个句柄，你可以在连接事件和断开事件中实现某些操作。

作为 API 的一部分，你可以定义一个函数 ble_lbs_on_ble_evt 用来处理协议栈事件，可以使用简单的 switch-case 语句通过返回事件头部的 id 号来区分不同的事件，并进行不同的处理。

```
void ble_lbs_on_ble_evt(ble_lbs_t * p_lbs, ble_evt_t * p_ble_evt)
{
    switch (p_ble_evt->header.evt_id)
    {
        case BLE_GATTS_EVT_WRITE:
            on_write(p_lbs, p_ble_evt);
            break;
        default:
            break;
    }
}
```

4.4 处理 LED 特性写

当 LED 特性被写入的时候，你添加到数据结构的函数指针将会通知到应用层，你可以在 on_write()函数中实现这样的功能。

当接收到一个写事件时，验证这个写事件是发生在对应的特性上是一个基本的任务，包括验证数据的长度，回调函数是否已设置。如果所有这些都是正确的，则回调函数将会调用，并且把已经写入的值作为输入参数。因此，on_write()函数的内容将会是这样：

```
ble_gatts_evt_write_t * p_evt_write =
&p_ble_evt->evt.gatts_evt.params.write;

if ((p_evt_write->handle == p_lbs->led_char_handles.value_handle)
&&

(p_evt_write->len == 1) &&

(p_lbs->led_write_handler != NULL))

{

 p_lbs->led_write_handler(p_ble_evt->evt.gap_evt.conn_handle,
p_lbs, p_evt_write->data[0]);//写入的数据

}
```

p_evt_write->data[0]数据真正触发 LED 的操作在于应用层，这样的设计让服务很容易使用，可以针对任何私有服务。

4.5 应用层实现

4.5.1 包含服务

在 main.c 文件中，必须出现调用 services_init()函数来初始化 LED 服务：

1.在 main.c 中包含 ble_lbs.h 头文件：

```
#include  "ble_lbs.h"
```

2. 如果没有添加源文件，则添加源文件到工程中：在工程窗口的左边在 Services 文件上点击右键，单击 Add file，选择 ble_led.c 文件。

3. 在 main.c 中添加服务的数据结构作为全局静态变量：

```
static ble_lbs_t      m_lbs;
```

注意：事件通过在 main.c 中使用静态变量的方式被保存，m_lbs 作为指针指向的变量经常会出现，指向它的指针为 p_lbs。

现在你可以在服务初始化函数中初始化你的定义服务了：

```
static void services_init(void)

{
```

```
    ret_code_t           err_code;
    ble_lbs_init_t       init       = {0};
    nrf_ble_qwr_init_t qwr_init = {0};

    // 初始化队列空间
    qwr_init.error_handler = nrf_qwr_error_handler;

    err_code = nrf_ble_qwr_init(&m_qwr, &qwr_init);
    APP_ERROR_CHECK(err_code);

    // 初始化服务
    init.led_write_handler = led_write_handler;

    err_code = ble_lbs_init(&m_lbs, &init);
    APP_ERROR_CHECK(err_code);
}
```

处理 LED 特性写中，我们在服务结构体中设置了 led_write_handler 回调，当 LED 特性被写入的时候将会调用。这个回调函数通过上面的初始化结构体被设置，但这个回调函数还没有实现。

在 services_init()函数之上，添加回调函数 led_write_handler()，以实现声明。设置 LED 输出状态值，它是函数的一个输入参数，其中第二个形参 led_state 对应 on_write()函数中 p_evt_write->data[0]的数据：

```
Static void led_write_handler(ble_lbs_t * p_lbs, uint8_t led_state)
{
if (led_state)
{
nrf_gpio_pin_set(LEDBUTTON_LED_PIN_NO);
}
```

```
else
{
nrf_gpio_pin_clear(LEDBUTTON_LED_PIN_NO);
}
}
```

最后，添加服务事件的处理函数到应用层事件调度函数(回调函数)，在 main 函数最开头，添加 BLE_LBS_DEF(m_lbs)回调处理函数，该函数内部调用了协议栈处理函数 ble_lbs_on_ble_evt。该函数前面已经谈过，这个蓝牙处理回调函数是整个操作的关键。

下面来简单描述下整个点灯的触发过程：

当服务建立好后，从机和主机相连，这里以手机为主机。

手机 app 点击写从机事件-----》

从机协议栈产生 BLE_GATTS_EVT_WRITE 事件-----》

ble_lbs_on_ble_evt 函数中对应该事件触发 on_write 写操作-----》

on_write 中就写入 p_evt_write->data[0]数据-----》

led_write_handler 回调中根据写入参数控制灯的状态。

4.5.2 加入本服务的 UUID 到广播数据包中

在广播数据包中包含服务 UUID，可以使集中器利用这个信息决定是否进行连接。在第 8 页第 2.1.2 节"广播"中所提到，一个广播数据包最多可携带 31 字节，如果需要更多的数据需要传输，可以使用扫描回应发送。你需要增加一个定制的 16 位的 UUID 到扫描回应数据包中，因为广播数据包已经没有可用的空间了。

广播数据的设置在 main.c 的 advertising_init()中，设置广播数据结构体，并调用 ble_advdata_set()来设置，它使用 2 个相同的数据类型的参数，一个是广播数据包，一个是扫描回应数据包。你必须添加一个数据结构作为扫描回应的参数。服务 UUID 设置为 LBS_UUID_SERVICE，类型使用结构体 ble_lbs_t 中的 uuid_type，广播数据包的初始化如下：

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

// YOUR_JOB: Use UUIDs for service(s) used in your application.
ble_uuid_t adv_uuids[] = {{LBS_UUID_SERVICE,m_lbs.uuid_type}};


.....................
.....................

memset(&scanrsp, 0, sizeof(scanrsp));
scanrsp.uuids_complete.uuid_cnt  =  sizeof(adv_uuids) /
sizeof(adv_uuids[0]);
scanrsp.uuids_complete.p_uuids  =  adv_uuids;
//添加到扫描回应包中：
 err_code = ble_advdata_encode(&srdata,
                               m_adv_data.scan_rsp_data.p_data
                               , &m_adv_data.scan_rsp_data.len);
 APP_ERROR_CHECK(err_code);
.............................
.............................
}
```

因为 m_lbs 结构体的 uuid_type 在这里被使用，所以确保它在 services_init()中已经被设置，并保证它在 advertising_init()之前被调用,在 main 中：

```
int main(void)
{
…
```

```
services_init();
advertising_init();
...
```

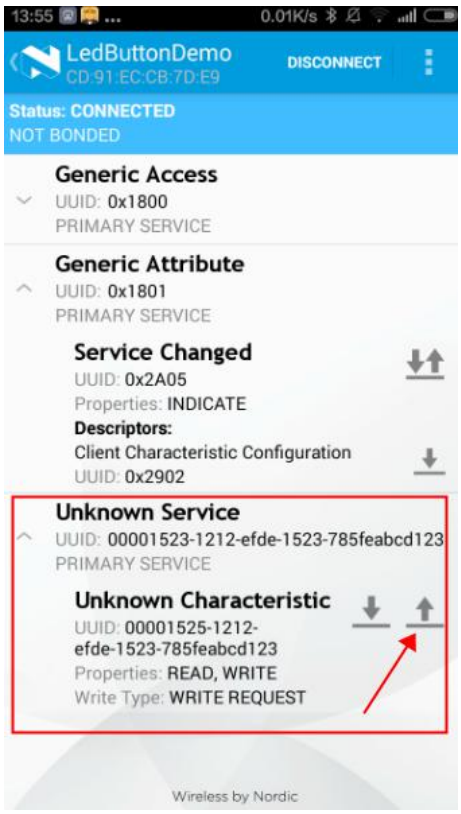
到此，这个应用已经建立完成。

5.下载验证：

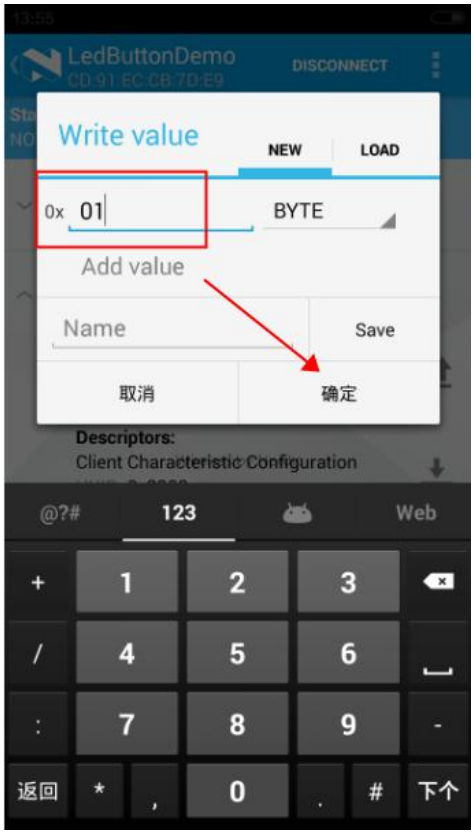
编写下载请参看《2.BLE 实验第二节：蓝牙 LED 任务读写使用说明》

载完成应用代码后，按下开发板复位按键运行程序。然后打开 app 如下图所示，发送名字为 Nordic_Blinky 的工程名字：

点击连接 connect，连接成功如下图所示：



显示私有服务，可以点击展开，查 UUID，特性：可读可写。可以点击写特征值，写入 0x01：



写入 0x01 后，反馈给 QY-nrf51822 开发板，LED 灯被点亮，如果在写入 0x00，LED 灯被熄灭。