本文介绍了在 nRF51822 上如何开始编写低功耗蓝牙(BLE)应用程序，包括对 BLE 的特性进行了概要的介绍，并且详细描述了构建一个定制服务的简单例程，这个简单的例程叫做 LED Button 服务。

1 简介

本文的目的是教你如何一步步创建自己的 BLE 应用程序，包括使用 nRF51822 芯片创建一个定制的服务。

1.1 最低要求

需要有嵌入式 C 语言编程经验，以便完全理解本应用手册。

1.1.1 需要的工具

需要一个 nRF51822 Evaluation Kit 的开发板，另外还需要下载和安装以下软件工具：

- S110 SoftDevice
- nRFgo Studio
- nRF51 SDK
- Keil MDK-ARM
- SEGGER's J-Link tools

如何把协议栈固件 S110 SoftDevice 烧录到 nRF51822 芯片中请参考:《nRF51822 Evaluation Kit User Guide》。

注意：当编写本文档时参考了最新版本的 nRF51 SDK 5.2.0 和最新版本的协议栈 S110 SoftDevice 6.0.0。

1.2 文档说明

下面的文档是重要的参考资料。

文档	描述
nRF51822 Evaluation Kit User Guide	使用 Evaluation Kit 开发板的介绍和配置，包括 Keil 和 SoftDevice 的配置。
nRF51 SDK documentation	这个文件在 SDK 安装的文件夹之下的子文件夹中，包含了 SDK 中所有功能 API 的文档。
S110 nRF51822 SoftDevice Specification	介绍了协议栈 S110 SoftDevice，包括资源的用法和高级的功能函数。
nRF51822 Product Specification	描述了 nRF51 的硬件、模块和电气特性。
nRF51 Series Reference Manual	介绍了 nRF51 芯片系列所有功能模块的描述和芯片所有的外围资源。
nAN-15: Creating Applications with the Keil C51 Compiler	这个应用手册包含使用 Keil µVision 的信息，它为 nRF24LE1 芯片而写，但是 3.3 节"Including files"和 3.4 节 ""Debug your project"同样适用于 nRF51822 芯片。
Bluetooth Core Specification, version 4.0 卷 1,3,4,6	这个文档由蓝牙技术联盟组织提供，包含了关于蓝牙服务和 profiles 的信息。

1.3 蓝牙技术资源

所有蓝牙技术联盟的服务、特性和描述都是根据蓝牙开发网站来定义，可以参考规范的不同部分找到 UUID 或者是数据格式的定义。

1.4 nRF51822 和 S110 SoftDevice

S110 SoftDevice 是 BLE 外围设备协议栈的解决方案，它集成了低功耗控制器、主机，并提供了一个完整和灵活的 API 用于在一个片上系统(SoC)构建一个低功耗蓝牙的解决方案，S110 SoftDevice 提供的是已经编译完成的 HEX 文件，在加载你的应用程序之前，你必须把它预先烧录到芯片中。

S110 SoftDevice 使用了一部分芯片的 flash 和 RAM，它们在你的代码中已被保护，因此你不能意外地写数据到该区域。S110 SoftDevice 也需要互斥地访问外围资源和寄存器。

如何烧录 S110 SoftDevice 到 nRF51822 中请参考：

《nRF51822 Evaluation Kit User Guide》

SoftDevice 使用了哪些资源，请参考：

《nRF51822 S110 SoftDeviceSpecification》

1.5 应用简述

LED Button 应用示例是为了让你学习如何在 nRF51822 上开发 BLE 应用，它是一个通过 BLE 的通知功能进行通信的 BLE 应用的简单演示。当它运行时，你可以通过集中器（见第 10 页 2.2.1 节“角色”中对集中器的定义部分）触发 nRF51822 上 LED 的输出，并且当在 nRF51822 上的按键被按下时集中器将会收到一个通知。

这个应用通过一个服务（见第 10 页 2.2.2 节“GATT 层”服务和特性的描述部分）被建立，这个服务包括 2 个特性：LED 特性和按键特性。LED 特性：通过没有回应的写远程操作 LED 的亮灭。按键特性：当按键被按下时，将会发送一个通知到集中器。

2 BLE 介绍

本章将介绍 BLE 协议不同的层，包括各个层的部件和它们的概念。

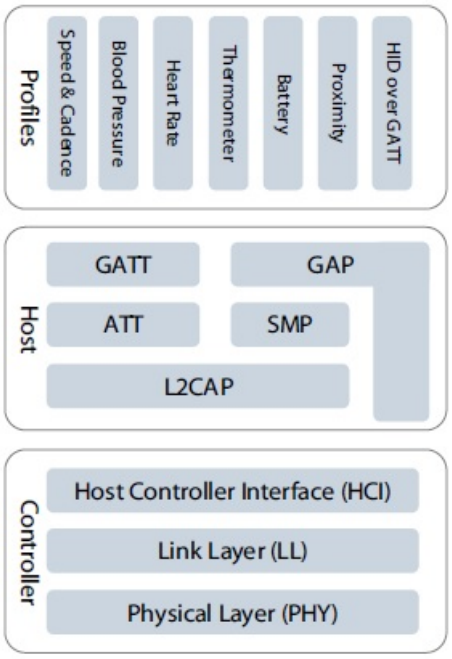


***Figure 1** Protocol stack components and layers*

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。

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 LED Button 应用例程中，使用 S110 SoftDevice 烧录到 nRF51822 作为外围设备，计算机或者手机作为集中器。

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，广播者角色只是广播信息而不接收信息。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，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。

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，每一个传输数据包最多可以携带 20 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。

2.2

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 LED Button 应用为例，外围设备（带有 LED 和按键）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。

2.2.2 GATT 层

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

在 LED Button 服务中，LED 和按键之间没有任何联系，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表示当前按键的状态，用另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，参见第 15 页第 2.2.5 节“空中操作和性质”。

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 S110 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 LED Button 应用例程，因为不关心它们的重用，所以把 LED 特性和按键特性放到了一个

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 LED BUTTON 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定制的服务。

对于 LED BUTTON 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 2 个定制的特性。

2.2.4 UUID

在第 10 页 2.2.2 节“GATT 层”中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。

2.2.4.1 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。

2.2.4.2 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，见第 29 页第 4.4.3 节“服务初始化”。

例如，在 LED BUTTON 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 LED BUTTON 示例中，0x1523 作为服务的 UUID，0x1524 作为 LED 特性的 UUID，0x1525 作为按键状态特性的 UUID。

2.2.5 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
- 指示

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。

2.2.5

2.2.5.1 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。

2.2.5.2 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。

2.2.5.3 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 LED BUTTON 示例中，控制 LED 的特性和当前按键状态的特性，都是 LED BUTTON 服务中的 2 个定制的特性。

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。

在按键特性中，当按键状态改变时客户端需要被通知到，但是应用层的确认没有必要，因此只需要通知的性质。

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 最小 BLE 应用简介

这个章节简单介绍了在 nRF51822 芯片上使用 S110 SoftDevice 协议栈构建一个最小的 BLE 应用的过程。

3.1 初始化介绍

有一些初始化函数通常在执行一个 BLE 应用之前调用，下面的表格中列出了这些初始化调用函数，在后面将对它们进行详细的介绍。

Initialization call	Ways to achieve this	In LED Button demo app
Enable	• Use SDK wrapper macro SOFTDEVICE_HANDLER_INIT from softdevice_handler.h • sd_softdevice_enable() in nrf_sdm.h	ble_stack_init() in main.c
Add event handler	• Function passed to SOFTDEVICE_HANDLER_INIT	ble_evt_dispatch() in main.c

Table 1 SoftDevice initialization calls

Initialization call	Ways to achieve this	In LED Button demo app
Set device name	• sd_ble_gap_device_name_set() in ble_gap.h	gap_params_init() in main.c
Set up advertising data	• ble_advdata_set() in ble_advdata.h • sd_ble_gap_adv_data_set in ble_gap.h	advertising_init() in main.c
Connection parameters	• ble_conn_params_init() in ble_conn_params.h • sd_ble_gap_ppcp_set() in ble_gap.h	conn_params_init() in main.c

Table 2 GAP initialization calls

Initialization call	Ways to achieve this	In LED Button demo app
Add one or more services	• sd_ble_gatts_service_add() in ble_gatts.h (Most often not done by application itself, only by services.)	ble_lbs_init() in ble_lbs.c.
Add one or more characteristics	• sd_ble_gatts_characteristic_add() in ble_gatts.h (Most often not done by application itself, only by services.)	ble_lbs_init() in ble_lbs.c.

Table 3 GATT initialization calls

Initialization call	Ways to achieve this	In LED Button demo app
Start advertising	• sd_ble_gap_adv_start() in ble_gap.h	advertising_start() in main.c

Table 4 App initialization calls

大部分采用数据结构的形式作为输入参数，这些数据结构包含一系列的配置和选项信息，阅读代码中的注释能更好地理解它们。

在广播开始之后，你就进入了 main 函数中的 for 循环。

3.2 协议栈 S110 SoftDevice

为了使用独特的射频特性，你必须使能 S110 SoftDevice 协议栈。见《S110 nRF51822 SoftDevice Specification》（S110 nRF51822 SoftDevice 协议栈说明书）中对硬件资源的详细需求。

3.3 广播

用于广播的数据结构如下：

ble_gap.h 中 ble_gap_conn_sec_mode_t

ble_advdata.h 中 ble_advdata_t

```
err_code = sd_ble_gap_device_name_set(&device_name_sec_mode, DEVICE_NAME
strlen(DEVICE_NAME));
err_code = sd_ble_gap_appearance_set(BLE_APPEARANCE_UNKNOWN);
err_code = ble_advdata_set(&advdata);
```

注意：传递给 sd_ble_gap_device_name_set()的安全模式仅适于设备本身的名字。

广播参数(ble_gap_adv_params_t）必须通过 sd_ble_gap_adv_start()来传递：

```
err_code = sd_ble_gap_adv_start(&m_adv_params);
```

3.4 连接参数

SDK 提供了一个名为 ble_conn_params 的模块用于管理连接参数更新，它通过 SoftDevice API

进行处理，包括请求的时间和第一次请求被拒绝再发送一个新的请求。
在初始化结构体 ble_conn_params_init_t 中，定义了更新过程的有关参数，例如，是否开始连接，什么开始写入一个特定的 CCCD，是否使用连接参数，发送更新请求的延时等等。
在初始化函数 ble_conn_params_init() 中，使用封装了初始化连接参数(ble_gap_conn_params_t）的结构体 ble_conn_params_init_t 作为输入参数进行连接参数初始化。

```
err_code = ble_conn_params_init(&cp_init);
```

ble_conn_params SDK 模块确保与主机（集中器）的连接参数相适应，如果不适应，外围设备将要求更改连接参数，超过设定的更新次数都没有更新成功后，它就会断开连接或者根据设置返回一个事件到应用层。

3.5 服务

服务可以通过 sd_ble_gatts_service_add()进行添加，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。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。变量 service_uuid 就是你想用于服务的 UUID。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。

```
err_code = sd_ble_gatts_service_add( BLE_GATTS_SVC_TYPE_PRIMARY,
&p_lbs->service_uuid,
&p_lbs->service_handle );
```

3.6 特性

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);
```

4 LED Button 应用实例

LED Button 应用示例是为了让你学习如何在 nRF51822 上开发 BLE 应用，它是一个简单的演示通过 BLE 的指示功能进行通信的 BLE 应用。当它运行时，你可以通过集中器触发 nRF51822 上 LED 的输出，并且当在 nRF51822 上的按键被按下时集中器将会收到一个通知。
这个应用通过一个服务被建立，这个服务包括 2 个特性：LED 特性和按键特性。LED 特性：通过没有回应的写远程操作 LED 的亮灭。按键特性：当按键被按下或释放时，将会发送一个通知。

4.1 代码简介

下面的章节将对这个应用是如何工作的进行一个简单的介绍，以帮助你理解和使用这些代码。

4.1.1 代码结构

这个示例代码分为 3 个文件：

- main.c
- ble_lbs.c

l ble_lbs.h

这个结构和 SDK 中其他的示例是一样的，在 main.c 中实现了应用的行为，分离的服务文件（ble_lbs.c，ble_lbs.h）实现了服务本身和它的行为。所有的输入输出的处理都是在应用层中进行的。

一个运行在 nRF51822 上的应用能够与以下几个部分进行交互：

l 硬件寄存器

寄存器在 nrf51.h 中定义，用到的常量在 nrf51_bitfields.h 中定义。在 LED BUTTON 应用中没有关于使用它们的示例。

l SDK 模块

SDK 模块包含了所有 BLE 开发的所有文件，从封装了硬件寄存器的基本头文件如 nrf_gpio.h，到提供给应用层的复杂的功能函数如 app_timer 或者 ble_conn_params。

l 协议栈 Softdevice 函数

用来配置或触发 Softdevice 中的行为，所有 Softdevice 函数的前缀都是以“sd_”开头。

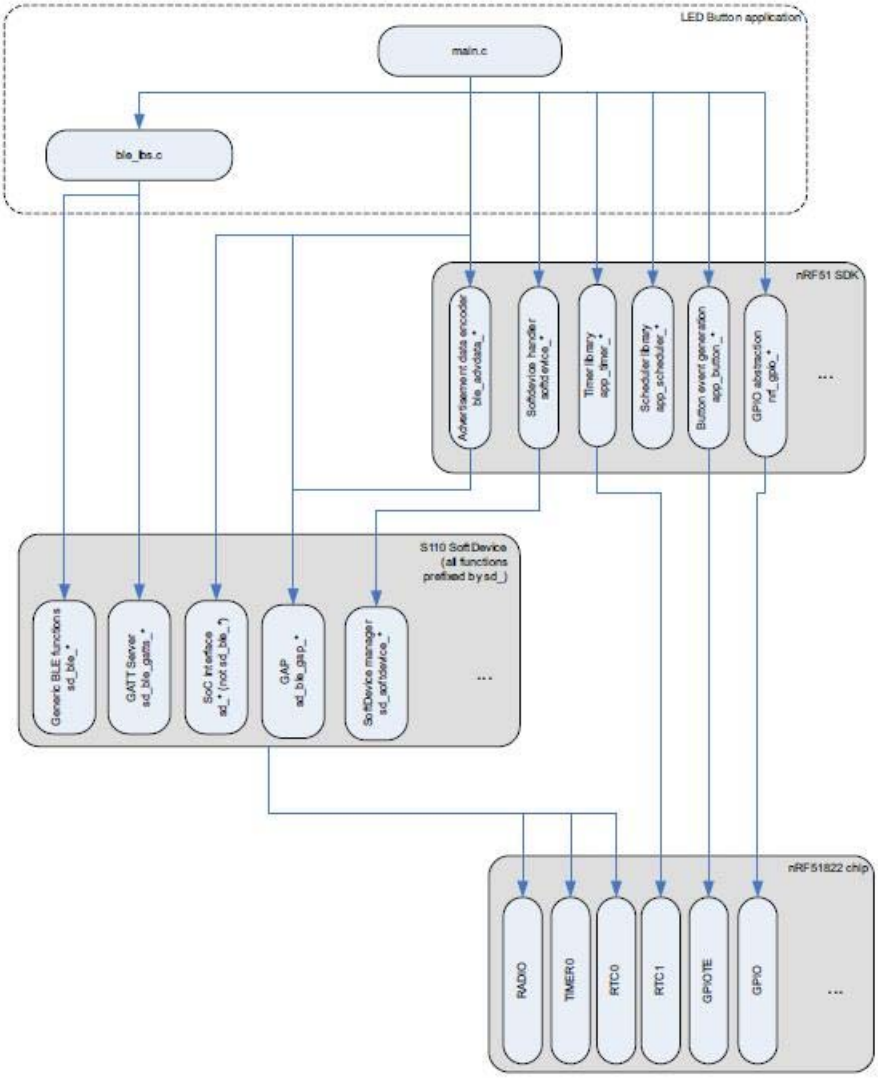


Figure 3 *LED Button application interfaces with the nRF51 SDK, S110 SoftDevice, and nRF51822 chip*

4.1.2 代码流程

在 nRF51822 上运行一个标准的 BLE 应用的基本流程是：初始化所有需要的部分，开始广播，进入省电模式，等待 BLE 事件发生。当接收到一个事件发生时，它将会被发送到 BLE 的服务和模块中，这个事件是但不仅限于以下一种事件：

- 当一个对等设备连接到 nRF51822 时
- 当一个对等设备写入到一个特性时
- 广播超时事件通知

这种流程让应用变得模块化，一个服务通常可以通过初始化加入一个应用之中，并且保证当事件发生时事件的句柄被调用。

4.1.3 在 KEIL 工程中进行导航和检测

建议在你开始查看示例代码之前先对它进行编译。编译之后，你可以在任何函数、变量、类型、定义上点击鼠标右键，通过选择 Go To Definition Of（跳转到定义体）或者 Go To Reference To（跳转到引用体）选项跳转到目标定义所在的地方。

Go To Definition Of 跳转到真正的实现方法（即源代码文件），Go To Reference To 跳转到它的头文件内的声明中，这意味着你不能跳转到 SoftDevice 中函数的定义体中，因为这些函数没有源码。但是，你可以跳转到他们的引用体，那里可以告诉你有关如何使用的方法说明。你可以使用这个有用的功能熟悉 API 函数和示例工程。

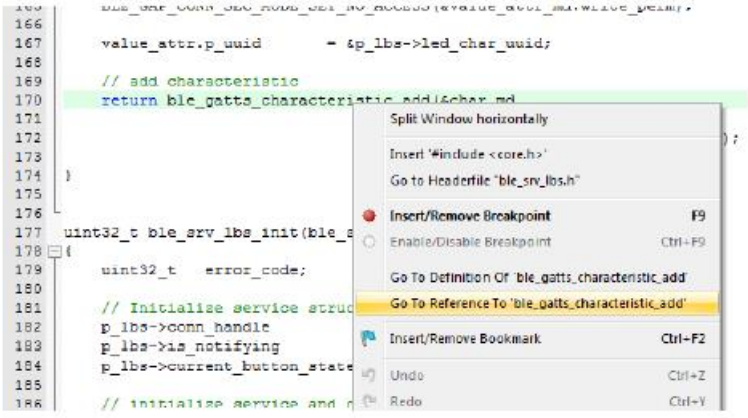


Figure 4 Finding definitions and references for methods and variables in Keil.

4.2 代码获取

这个应用笔记介绍了如何建立一个 LED Button 应用的步骤，另外，完整的示例代码在 GitHub 上的 Git 资源库中提供，你可以看到代码的历史版本以及它是如何被开发的。每一部分都有它自己的标签，在本文的末尾将告诉你如何查阅这些代码。

在 GitHub 上可以找到本工程的代码：https://github.com/NordicSemiconductor/nrf51-ble-app-lbs

4.3 创建

本应用例程需要使用 nRF51822 Evaluation Kit 开发板，当然，可以经过适当的修改也可以在 Development Kit.开发板上运行。

4.3.1 设置 Evaluation 开发板

因为 nRF51822 Evaluation Kit 开发板上已经有板载 SEGGER 芯片，因此你可以使用 USB 线就能直接开始工作。

4.3.2 设置应用

有很多模式样板代码需要用于开始创建一个应用和服务，所以第一步就是从 SDK 中复制代码：

1. 进入 Board\nrf6310\s110\ble_app_template 文件夹
2. 把这个文件夹复制到 Board\pca10001\s110\，并重新命名为：ble_app_lbs
3. 进入文件夹 ble_app_lbs 中的 arm 文件夹，把工程文件名从 ble_app_template 改为 ble_app_lbs。

4.3.3 设置服务

SDK 没有一个服务的模板，但是有一个现成的电池服务，这个服务是一个简单的首要服务，非常适合开始创建一个服务的模板。因此需要进行如下步骤：

1. 从 Source/ble/ble_services 复制 ble_bas.c 到 Board/pca10001/ble/ble_app_lbs/
2. 从 Include/ble/ble_services 复制 ble_bas.h 到 Board/pca10001/ble/ble_app_lbs/
3. 把 ble_bas.c 重命名为 ble_lbs.c，把 ble_bas.h 重命名为 ble_lbs.h
4. 在 Keil 左边的工程窗口中双击 Services 文件夹，选择添加 ble_lbs.c 文件。这样就把 ble_lbs.c 文件添加到了你的 keil 工程中。

因为这个是一个定制的服务，最好把它放在应用层文件夹中，以代替放在 SDK 的服务文件夹中。

4.4 实现服务

这个服务采用通用的方式实现，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。实现其他预定义服务的方法也是类似的。

4.4.1 API 设计

ble_lbs.h 头文件实现了各种数据结构、应用需要实现的事件句柄和以下 3 个 API 函数：

```
uint32_t ble_bas_init(ble_bas_t * p_bas, const ble_bas_init_t * p_bas_init);
void ble_bas_on_ble_evt(ble_bas_t * p_bas, ble_evt_t * p_ble_evt);
uint32_t ble_bas_battery_level_update(ble_bas_t * p_bas, uint8_t battery_level);
```

注意：本文档中代码片段的注释已经被去掉

在上面的代码中，ble_bas_t 用于引用这个服务实例，在后面还会用到。而 ble_bas_init_t 用于初始化参数，后面不会再用到。所有的 API 函数使用一个指向服务实例的指针作为第一个输入参数。

LED Button 服务的 API 函数也是同样的设计，步骤如下：

1. 在对应的头文件和源文件中，执行 Find and Replace All 功能，把所有出现 ble_bas 的地方替换成 ble_lbs；把所有出现 BLE_BAS 的地方替换成 BLE_LBS；把所有出现 p_bas 的地方替换成 p_lbs。
2. 在对应的头文件和源文件中，把 ble_lbs_battery_level_update()函数都移除。
3. 在 ble_lbs.c 文件中，使用#if 0 在第一个函数之前，使用#endif 在初始化函数 ble_lbs_init（）之上注释掉所有的函数，不要完全删除它们，因为后面建立函数还需要用到它们。
4. 在对应的头文件和源文件中，把 battery_level_update 函数都移除。
5. 在初始化函数 ble_lbs_init（）的末尾，删除 battery_level_char_add()函数，用 return NRF_SUCCESS 进行代替。
6. 考虑到服务要符合应用的需求，LED Button 服务需要知道什么时候按键状态改变了，才能把状态发送到集中器设备，因此，你需要增加一个函数当按键状态改变的时候应用程序能够调用它。

```
uint32_t ble_lbs_init(ble_lbs_t * p_lbs, const ble_lbs_init_t * p_lbs_init);
void ble_lbs_on_ble_evt(ble_lbs_t * p_lbs, ble_evt_t * p_ble_evt);
```

```
uint32_t ble_lbs_on_button_change(ble_lbs_t * p_lbs, uint8_t button_state);
```

这里有 2 个数据结构需要实现：ble_lbs_t 和 ble_lbs_init_t。

4.4.2 实现数据结构体

在第 25 页 4.4.1 节“API 设计”中，有一些用到的数据结构还没有定义：ble_lbs_t 和 ble_lbs_init_t.，我们可以基于电池服务的数据结构体实现类似的结构体，电池服务的数据结构体如下：

```
typedef struct
{
ble_bas_evt_handler_t        evt_handler;
bool                         support_notification;
ble_report_ref_t             * p_report_ref;
uint8_t                      initial_batt_level;
ble_cccd_security_mode_t     battery_level_char_attr_md;
ble_gap_conn_sec_mode_t      battery_level_report_read_perm;
} ble_bas_init_t;
```

```
typedef struct ble_bas_s
{
ble_bas_evt_handler_t         evt_handler;
uint16_t                      service_handle;
ble_gatts_char_handles_t      battery_level_handles;
uint16_t                      report_ref_handle;
uint8_t                       battery_level_last;
uint16_t                      conn_handle;
bool                          is_notification_supported;
} ble_bas_t;
```

以上的代码中的初始化结构体包含服务的事件句柄，一些选项参数，初始化值，安全模式，服务结构体包含服务的声明，如句柄，当前电量值，通知功能是否打开等。

电池服务使用一个通用的事件句柄让应用程序知道什么时候开始和停止读取电池电量。LED Button 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。

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);
```

然而，下面的参数参数还需要定义：

- 服务的句柄
- 特性的句柄

l 连接的句柄
l 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;
uint16_t                conn_handle;
} ble_lbs_t;
```

可以删除 ble_lbs.h 文件中有关电池服务事件的定义。

4.4.3 服务初始化

进入服务初始化函数 services_init()，函数 ble_lbs_init（）被调用。参数你不需要改变。

1. 删除 evt_handler, is_notification_supported 和 battery_level_last.
2. 在服务初始化结构体和服务结构体中，都要把 evt_handler 重命名为 led_write_handler。

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

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：

1. 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
#define LBS_UUID_BUTTON_CHAR    0x1524
```

在服务初始化中：

1. 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：

```
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。

1. 当为 LED Button 服务设置 UUID 时，使用这个 UUID 类型，它在 ble_lbs:init()中：

```
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，下面的章节将介绍如何添加特性。

4.4.3.1 实现按键特性

本服务将要实现 2 个特性，一个是控制 LED 的状态，一个是反馈按键的状态。这两个功能需要创建并增加 2 个特性到 ble_lbs.c 中实现，我们先从按键特性说起。

按键特性在有按键状态改变的时候有通知事件，同时也允许对等设备读取这个按键状态。这非常类似于在电池服务中电量特性的特征，因此你可以采用以下方式实现：

1. 找到 battery_level_char_add()函数并重命名为 button_char_add()。
   button_char_add()期望找到一个参数说明是否支持通知功能，在这个示例中我们希望只支持通知功能。
2. 删除检测参数 is_notification_supported 的 if 语句，但是保留 if 语句中的内容。
3. 确保 char_md 中对应字段设置为支持通知功能。
4. 删除所有与报告参考相关的代码和参数 p_report_ref（每次执行到 if 语句都检查 p_report_ref 是否设置）。

在电池服务的初始化中有一个标志设置了 CCCD 的安全模式，它存储在 ble_gap_conn_sec_mode_t 结构体中，这个结构体使用在头文件 ble_gap.h 中定义的宏 BLE_GAP_CONN_SEC_MODE 来设置，根据不同的安全等级定义了不同的宏，你可以根据属性的需要进行选择。

1. 使用宏 BLE_GAP_CONN_SEC_MODE_SET_OPEN 把 CCCD 设置成对任何连接和加密都是可读可写的模式。

1. 如果对于按键状态特性我们想让每一个连接都可读但不能写，可以使用 BLE_GAP_CONN_SEC_MODE_SET_NO_ACCESS 代替 BLE_GAP_CONN_SEC_MODE_SET_OPEN。

```
BLE_GAP_CONN_SEC_MODE_SET_OPEN(&cccd_md.read_perm);
BLE_GAP_CONN_SEC_MODE_SET_OPEN(&cccd_md.write_perm);
...
memset(&attr_md, 0, sizeof(attr_md));
BLE_GAP_CONN_SEC_MODE_SET_OPEN(&attr_md.read_perm);
BLE_GAP_CONN_SEC_MODE_SET_NO_ACCESS(&attr_md.write_perm);
```

1. 确保你不要删除字段 vloc 的设置，它决定了变量是使用协议栈的内存还是用户空间的内存。
2. 设置 UUID 的类型和 UUID 的值：

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

1. 初始化值不重要，你可以把 p_initial_value 设置为 NULL。
2. 确保把返回的特性句柄 button_char_handles 保存在正确的地方，最后调用的函数如下：

```
return ble_gatts_characteristic_add( p_lbs->service_handle,
&char_md,
&attr_char_value,
&p_lbs->button_char_handles );
```

删除#endif 以上的内容后，你就可以编译工程了，根据编译的警告删除没有用到的变量(initial_battery_level, encoded_report_ref, init_len, err_code)。

4.4.3.2 实现 LED 特性

LED 状态特性需要能够可读可写，但没有任何通知功能：

1. 复制按键特性函数，重命名为 led_char_add。
2. 删除与 cccd_md 有关的代码。
3. 增加写的性质代替通知性质（给这个特性使能写性质）。

```
char_md.char_props.write = 1;
```

1. 更改使用的 16 位 UUID 为 LBS_UUID_LED_CHAR：

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

保存返回的变量 led_char_handles（LED 特性的句柄），代替 button_char_handles 的位置，最后调用的函数如下：

```
return ble_gatts_characteristic_add( p_lbs->service_handle,
&char_md,
&attr_char_value,
&p_lbs->led_char_handles );
```

编译之后，你会发现有一些参数没有使用，删除它们。

4.4.3.3 增加特性

创建了增加特性的函数之后，你可以在服务初始化的末尾调用它们，如下面的例子：

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

因为任何错误都会导致函数提前退出，因此当你到达函数的末尾时可以认为初始化成功了。如果你想现在就进行测试，你可以跳到第 37 页第 4.5.1 节“为 evaluation kit 开发板修改模块”和第 40 页第 4.5.3 节“包含服务”，完成之后，最后的测试在第 47 页第 5 章“应用测试”中介

绍。测试时你会发现，你可以连接到这个设备并且可以发现所有的服务，但是其他的功能不能工作，因此还需要在这个服务中实现处理协议栈事件和按键处理。

4.4.4 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。

4.4.4.1 保存连接句柄

电池服务已经保存了连接句柄，但没有改变之前采用 Find and Replace 来查找时需要注意一些。

4.4.4.2 参数 CCCD 写的处理

现有的事件句柄监听 CCCD 写的操作并把它发送应用层的电池服务的事件句柄，但是，在本应用中，不需要这样。

原有的代码中实现了发送通知的方法，但是如果 CCCD 没有使能，sd_ble_gatts_hvx()（通知或者指示回调函数）将不允许发送通知，因此你不需要在应用层中进行检测，而是交给协议栈 SoftDevice 中去检测。

在 on_write()函数中，你可以删除与通知功能相关的代码。

4.4.4.3 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_lbs, p_evt_write->data[0]);
}
```

真正触发 LED 的操作在于应用层，这样的设计让服务很容易重用，虽然这只是针对于 LED 和按键的服务。

4.4.5 处理按键事件

你已经添加了一个回调 API 函数让服务知道按键何时被按下，但还没有全部实现，因此你需要从头文件中通过复制来添加，在处理按键按下的时候，你需要给对等设备发送一个通知以告知它新的按键状态。协议栈 SoftDevice API 函数 sd_ble_gatts_hvx 来完成这个事情，它需要连接句柄和结构体 ble_gatts_hvx_params_t 作为输入参数，它管理一个值被通知的整个过程。在结构体 ble_gatts_hvx_params_t 中，你需要设置为通知模式还是指示模式，用哪一个属性的句柄用来进行通知（本例中使用值的句柄），新的值以及值的长度。方法如下：

```
uint32_t ble_lbs_on_button_change(ble_lbs_t * p_lbs, uint8_t button_state)
{
ble_gatts_hvx_params_t params;
```

```
uint16_t len = sizeof(button_state);
memset(&params, 0, sizeof(params));
params.type = BLE_GATT_HVX_NOTIFICATION;
params.handle = p_lbs->button_char_handles.value_handle;
params.p_data = &button_state;
params.p_len = &len;
return   sd_ble_gatts_hvx(p_lbs->conn_handle, &params);
}
```

也可以使用 sd_ble_gatts_value_set()函数一次性设置特性的值，当通知的时候调用 sd_ble_gatts_hvx()不需要设置值和值的长度。但是，没必要使用 sd_ble_gatts_hvx()做任何事情，它不过类似于一个清洁工的角色。使用函数 sd_ble_gatts_value_set()更新一个可读(但不能通知)的值，因为这个函数不通过空中发送数据包。

按照本服务的实现方法，本例子可以应用在其他应用中。

4.5 应用层实现

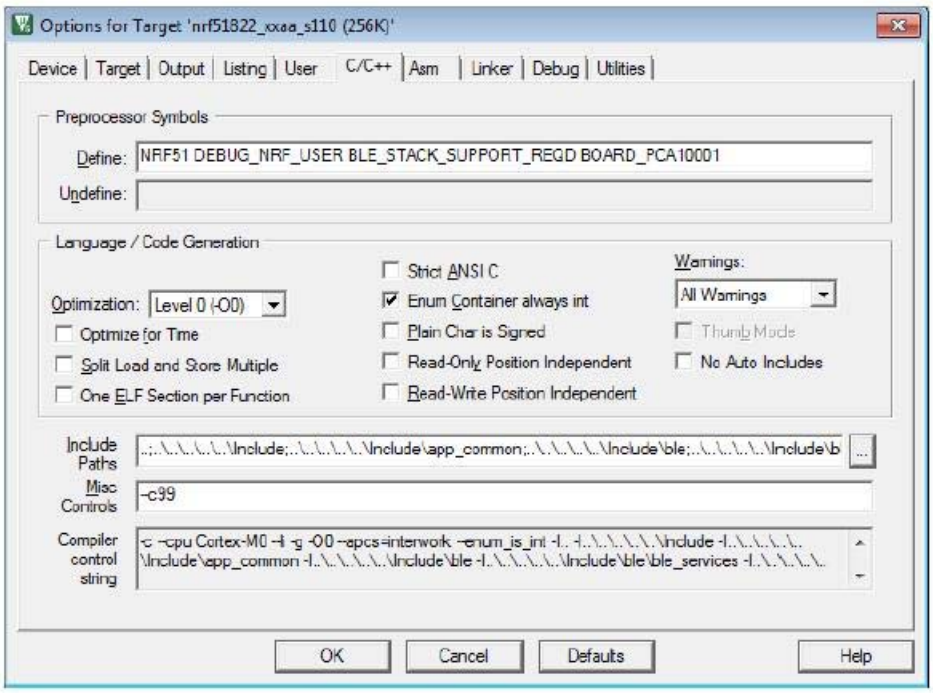
4.5.1 修改模板适应于 evaluation kit 开发板

需要使用模板进行一些修改以适应于 evaluation kit 开发板，以代替 Development Kit 开发板。

首先，你需要更改定义板子的宏定义：

1. 打开工程文件，进入到“Target options”标签和“C/C++”标签。
2. 把 BOARD_NRF6310 更改为 BOARD_PCA10001，如下图所示：

注意：如果你用 development kit 开发板代替 evaluation kit 开发板，跳到步骤 4 和步骤 5，并且把 ble_app template 拷贝到 Board\nrf6310\s110\ble_app_lbs 下，你便建立了一个工程。



你需要在 main.c 中删除一些引脚定义，因为在 evaluation kit 开发板上，没有一个单独的 LED 用于言断。

1. 在 main.c 中删除所有有关 ASSERT_LED_PIN_NO 的代码(宏定义，在 app_error_handler()中使用，在 leds_init()中设置为输出引脚)。
2. 你需要一个 LED 用于 LED Button 服务，因此你需要在 main.c 的顶部添加定义 LED 引

脚的宏定义。

```
#define LEDBUTTON_LED_PIN_NO        LED_0
```

1. 在 In leds_init()中配置引脚为作为 LED 引脚：

```
nrf_gpio_cfg_output(LEDBUTTON_LED_PIN_NO);
```

1. 在 SDK 4.1.0 以后，当错误发生时，默认的应用层错误处理是复位，但是对于开发，更有用的是使用提供的 debug 模块，你需要确保取消注释 debug 言断的处理，注释掉复位的处理：

```
void app_error_handler(uint32_t error_code, uint32_t line_num, const uint8_t *
p_file_name)
{
// [Comment removed from snippet for brevity]
ble_debug_assert_handler(error_code, line_num, p_file_name);
// On assert, the system can only recover with a reset.
//NVIC_SystemReset();
}
```

当你运行程序在调试模式下的时候，如果出现错误你可以停止 cpu，并能够发现文件中哪一行发生了错误。你可用 SoftDevice 中的 APP_ERROR_CHECK()宏来检测返回的错误代码得到有关错误的信息。如图 5 所示的例子.

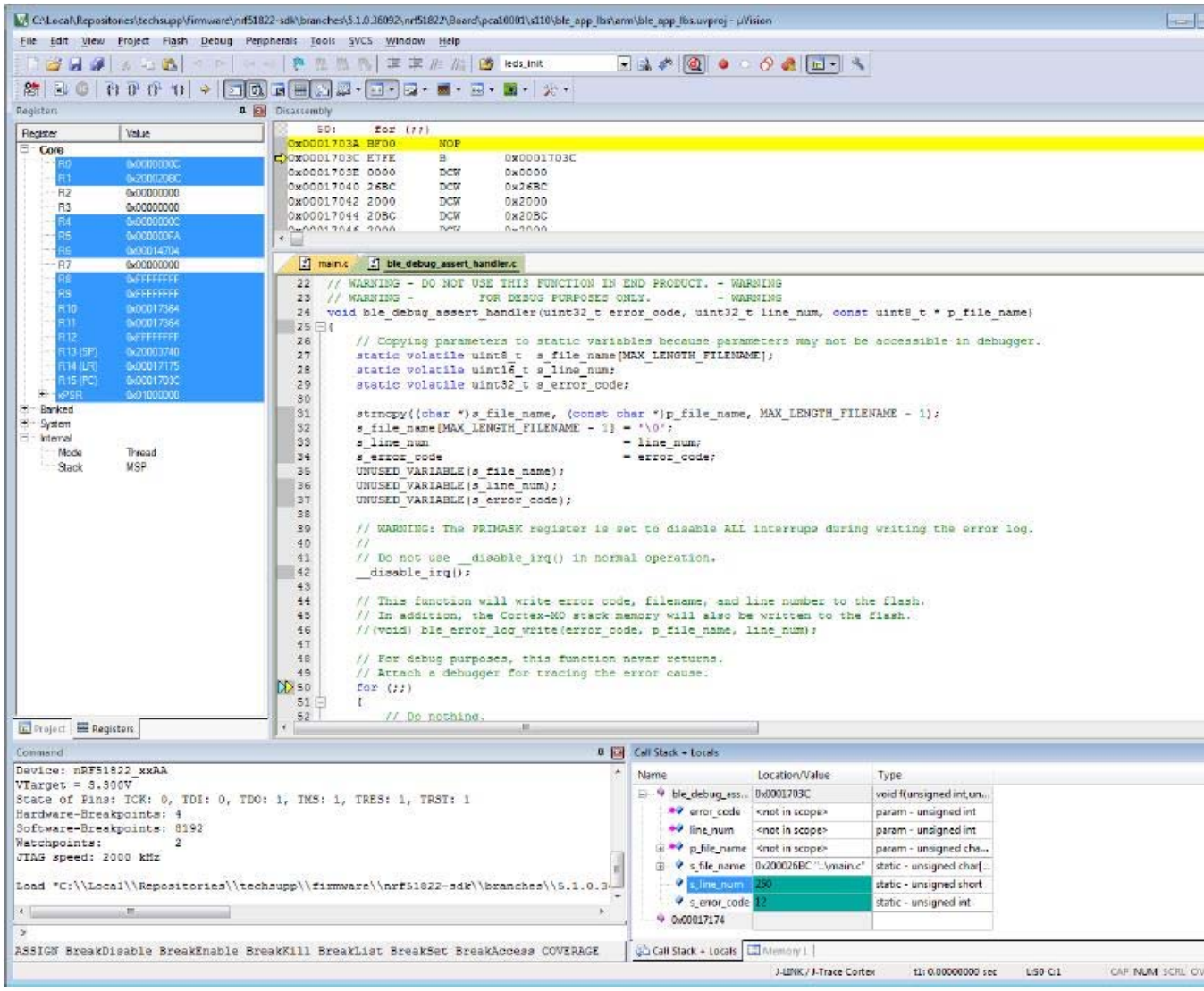


***Figure 5** Application is halted with the debugger in the assert handler, showing an error with error coa*

在实际的产品中，通常使用复位来恢复错误，但在开发中，日记更重要。

建议在工程中更改蓝牙设备的名字，可以通过改变 main.c 中的宏 DEVICE_NAME 来实现，例如可以改成“LedButtonDemo”。

4.5.2 使用调度

SDK 提供了调度模块，这个模块提供了把事件处理和中断处理转移到 main 函数中进行的机制，它保证了所有的中断处理都能快速被执行。

在开始使用的模板中，默认使能了调度功能。如果你不想使用它，你可以删除它的初始化和 main 循环中对它的调用（scheduler_init(), app_sched_execute()），设置 SDK 的模块初始化函数中相关的标识使用调度的参数为假（softdevice_handler, app_timer,app_button)。

关于调度的更多详情，请查阅 nRF51 SDK 文档。

4.5.3 包含服务

为了使用你创建的服务，你需要在模板中添加一些代码。

在 mian.c 文件中，必须出现调用 services_init()函数来初始化 LED Button 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_lbs.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。

1. 现在你可以初始化你的服务：

```
static void services_init(void)
{
uint32_t    err_code;
ble_lbs_    init_t init;
init.led_write_handler = led_write_handler;
err_code = ble_lbs_init(&m_lbs, &init);
APP_ERROR_CHECK(err_code);
}
```

在第 35 页 4.4.4.3 节“处理 LED 特性写”中，我们在服务结构体中设置了 led_write_handler，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。

1. 在 services_init()函数之上，添加回调函数 led_write_handler()，以实现声明。
2. 设置 LED 输出状态值，它是函数的一个输入参数：

```
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);
}
}
```

1. 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)

```
static void ble_evt_dispatch(ble_evt_t * p_ble_evt)
{
on_ble_evt(p_ble_evt);
ble_conn_params_on_ble_evt(p_ble_evt);
ble_lbs_on_ble_evt(&m_lbs, p_ble_evt);
}
```

4.5.4 测试

现在你可以进行测试了，见第 47 页第 5 章“应用测试”，使用 Master Control Panel，你可以写“1”到 LED 特性中来点亮 LED。

4.5.5 按键处理

为了完成本应用，你需要定义如何处理按键按下，你可以使用 SDK 提供的 app_button 模块，

这个模块将会提供当按键按下和释放时的一个回调函数。
在按键初始化 buttons_init()里，设置你需要的按键，在这个示例中使用了 evaluation kit 开发板上的 button 1。

1. 添加一个新的宏定义，只是为了让代码具有可读性：

```
#define   LEDBUTTON_BUTTON_PIN_NO      BUTTON_1
```

1. 在 buttons_init()里配置按键的引脚，添加按键配置数组如下：

```
static app_button_cfg_t buttons[] =
{
{WAKEUP_BUTTON_PIN, false, NRF_GPIO_PIN_PULLUP, NULL},
{LEDBUTTON_BUTTON_PIN_NO, false, NRF_GPIO_PIN_PULLUP, button_event_handler},
};
APP_BUTTON_INIT( buttons, sizeof(buttons) / sizeof(buttons[0]),
BUTTON_DETECTION_DELAY, true);
```

在 evaluation kit 开发板上的按键是低电平有效，所以第 2 个参数为 false，但它没有外部上拉电阻，因此你需要使用 NRF_GPIO_PIN_PULLUP 来使能内部上拉电阻。唤醒按键 wakeup_button 没有声明，所以它的回调函数设置为 NULL。完成之后，你就已经完成按键模块的初始化了。

1. 取消对函数 button_event_handler()的注释，app_button 模块将通过参数传递当前的按键状态，因此你可以通过服务 API 函数直接传递这个按键状态：

```
static void button_event_handler(uint8_t pin_no, uint8_t button action)
{
uint32_t err_code;
switch(pin_no)
{
case LEDBUTTON_BUTTON_PIN_NO:
err_code = ble_lbs_on_button_change(&m_lbs, button_action);
if (err_code != NRF_SUCCESS &&
err_code != BLE_ERROR_INVALID_CONN_HANDLE &&
err_code != NRF_ERROR_INVALID_STATE)
{
APP_ERROR_CHECK(err_code);
}
break;
default:
APP_ERROR_HANDLER(pin_no);
break;
}
}
```

在上面的代码中，我们忽略可能客户端 CCCD 还没有被写入，或者我们没有正确连接所带来的错误。

为了添加已经定义的函数，你需要确定按键模块已经被使能。默认情况下，应用模板建议连接事件和断开连接事件中来完成使能和禁止，这里我们也使用这种方式。取消对 app_button_enable() 和 app_button_disable()的注释：

```
switch (p_ble_evt->header.evt_id)
{
case BLE_GAP_EVT_CONNECTED:
…
/* … */
err_code = app_button_enable();
break;
case BLE_GAP_EVT_DISCONNECTED:
…
/* … */
err_code = app_button_disable();
if (err_code == NRF_SUCCESS)
{
advertising_start();
}
break;
```

现在你可以进行测试了，见第 47 页第 5 章“应用测试”的描述，然而为了让集中器在扫描的时候能够更容易区分不同的设备，有必要在广播数据包中加入本服务的 UUID。

4.5.6 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;
uint8_t    flags = BLE_GAP_ADV_FLAGS_LE_ONLY_LIMITED_DISC_MODE;

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

// Build and set advertising data
memset(&advdata, 0, sizeof(advdata));

advdata.name_type          = BLE_ADVDATA_FULL_NAME;
```

```
advdata.include_appearance    = true;
advdata.flags.size            = sizeof(flags);
advdata.flags.p_data          = &flags;

memset(&scanrsp, 0, sizeof(scanrsp));
scanrsp.uuids_complete.uuid_cnt   =   sizeof(adv_uuids) / sizeof(adv_uuids[0]);
scanrsp.uuids_complete.p_uuids   =   adv_uuids;

err_code = ble_advdata_set(&advdata, &scanrsp);
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 应用测试

需要一个 USB dongle 与开发板 evaluation kit，并配合 Master Control Panel 软件，以用于测试 BLE 应用。前期的准备工作在《*nRF51822 Evaluation Kit User Guide*》中的“Quick Start”章节有详细的介绍，打开 Master Control Panel 软件，你可以测 LED Button 应用如以下步骤：

1. 打开 Master Control Panel 软件

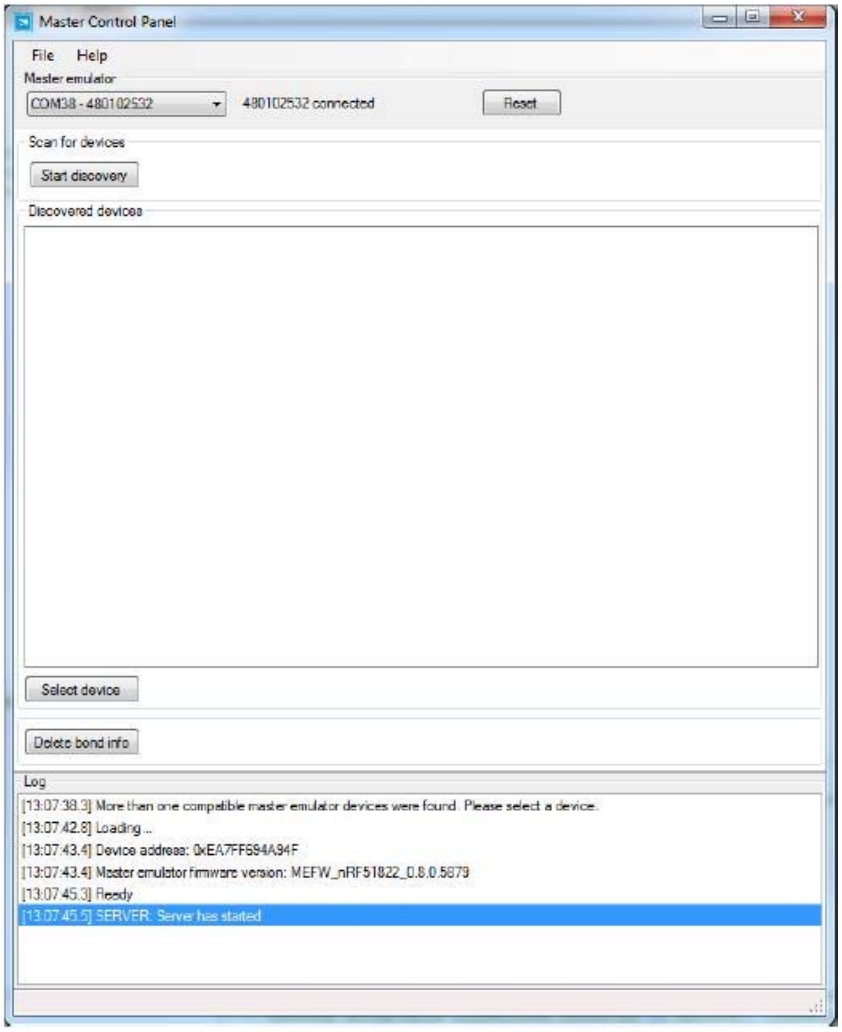


***Figure 6** Master Control Panel*

2. 当 Master Control Panel 已打开，点击“Start Discovery”，LED Button 设备很快就会出现在“Discovered devices”窗口中，如果没有出现，很可能是广播超时，按下 Evaluation Kit 开发板上的 button 0，或者重新复位芯片，重新开始广播。
3. 当设备出现时，选择它，然后点击“Select Device.”

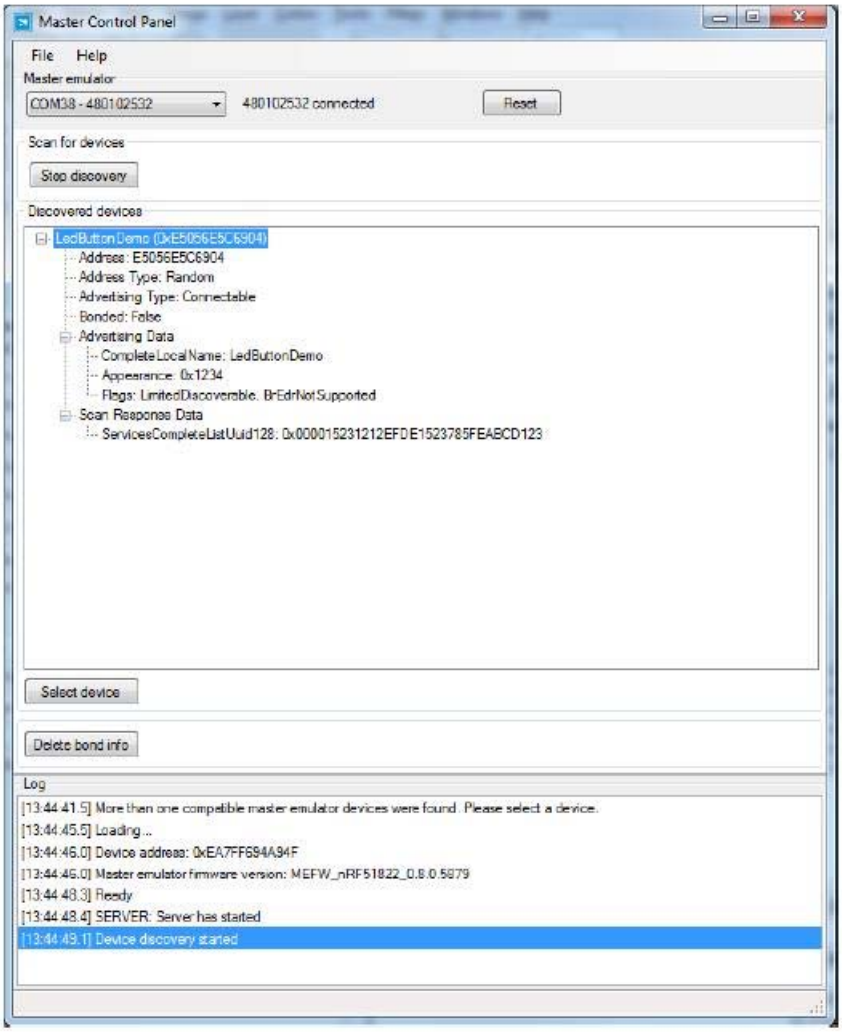


Figure 7 *Master Control Panel after the device has been discovered, showing the scan response data.*

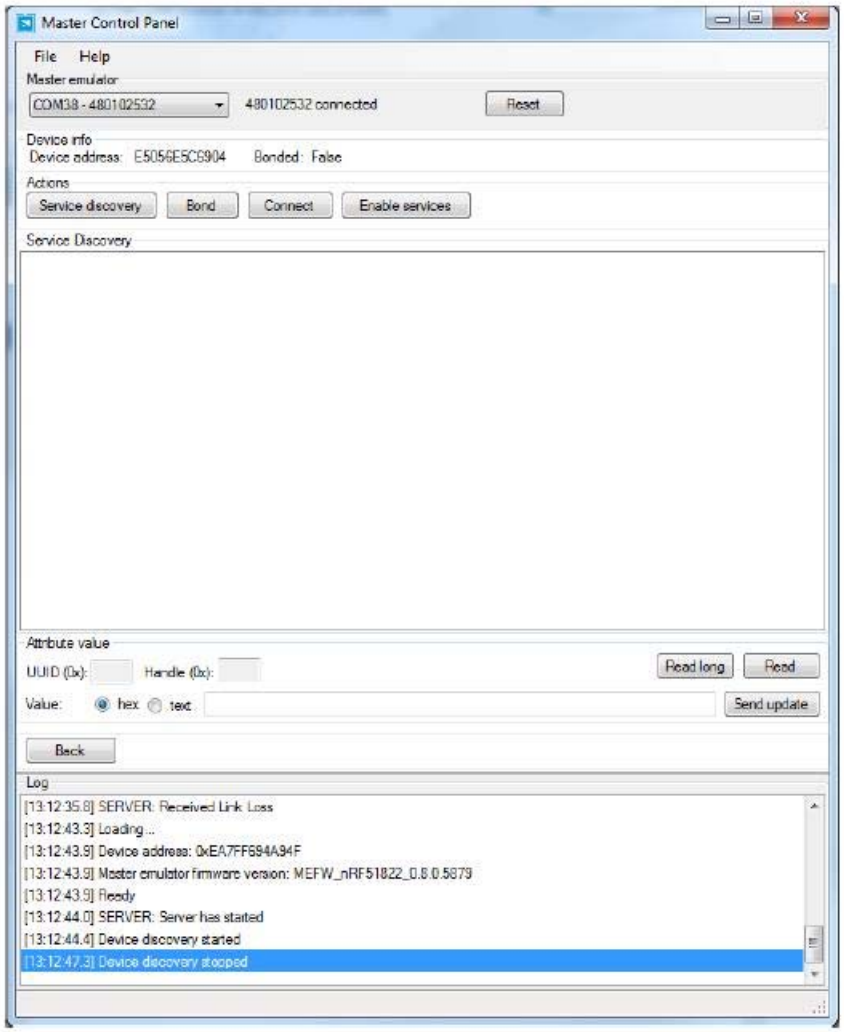


Figure 8 *Master Control Panel when the LED Button device has been selected.*

4. 点击“Service discovery”，这是第一次连接到这个设备，所以进行服务发现。

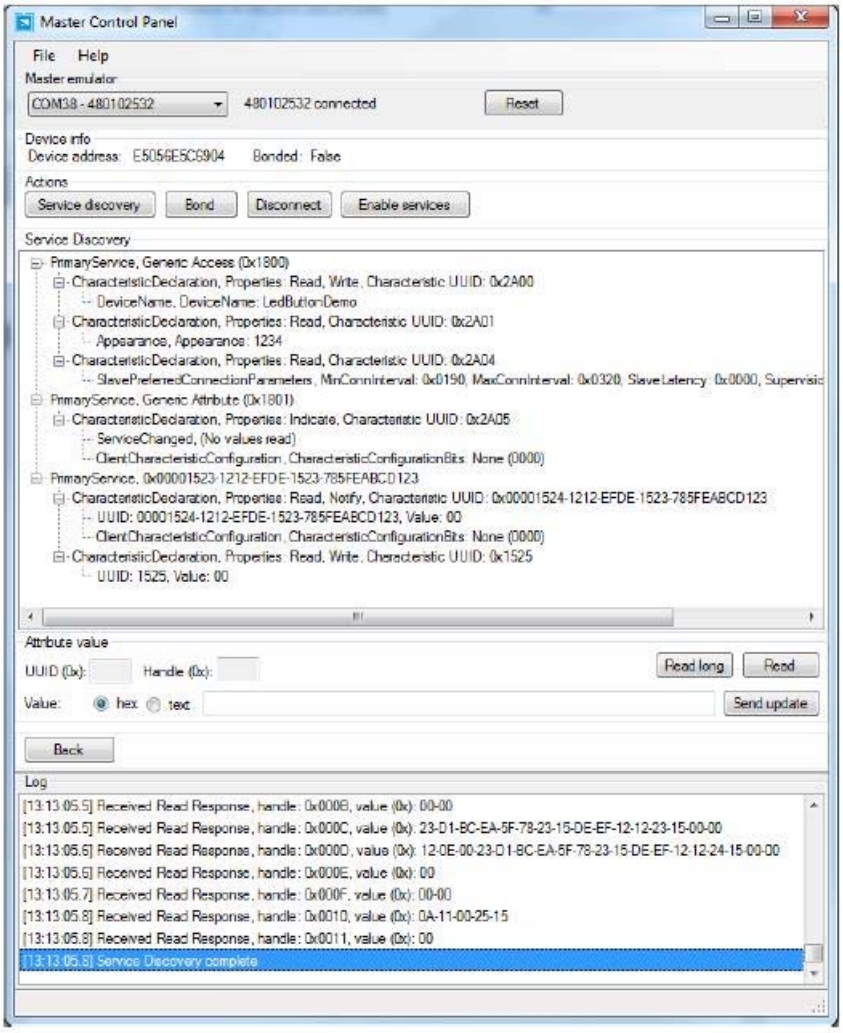


***Figure 9** Master Control Panel after service discovery*

你将会发现设备有 3 个服务 ，虽然我们只是添加了 1 个服务。在底部你可以找到 LED Button 的服务。另外 2 个是 GAP 服务(0x1800)，它包含 GAP 数据，包括之前设置的一些参数；一个是 GATT 服务(0x1801),它包含如果初始化服务后需要更改服务所用到的特性值。所有的 BLE 服务必须包含这些服务，协议栈 SoftDevice 自动添加了它们。

你现在可以打开通知功能了，看看按键按下时是否显示。

1. 点击“**Enable services**”以打开通知功能，按下 Evaluation Kit 开发板上的 Button 1。

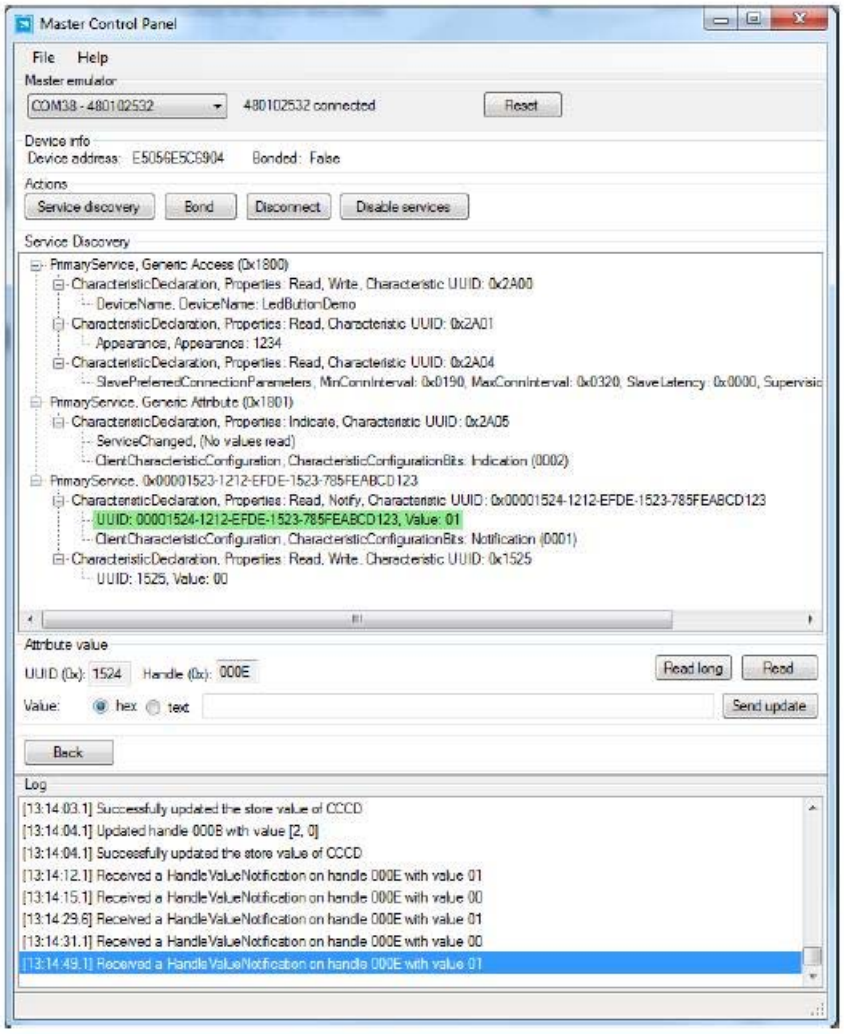


Figure 10 *Master Control Panel. Notifications have been turned on and a button has been pressed.*

你将会看到 CCCD 的通知位已经被设置为 1，并且当按键按下时特性的值就会更新。

1. 测试点亮 LED 灯：点击 LED 特性的值，在属性值的下面,设置值为 hex 格式，填入 01，点击“Send update”。这将通过空中发送一个写操作到设备上，设备将点亮 LED 灯

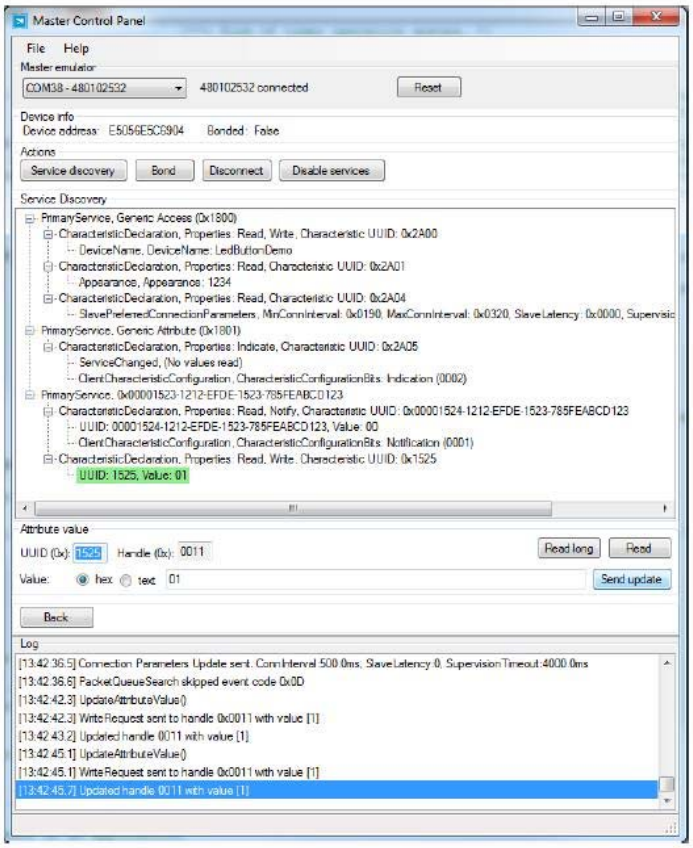


Figure 11 *Master Control Panel. The value of the LED characteristic has been updated to 01.*