目录

青风带你玩蓝牙 nRF52832 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

青风 5.0 蓝牙教程：蓝牙自定义与动态广播

对应如果实现自定义广播，是不少读者提出的一个问题，本节将针对如果实现自定义广播进行讲解。当然，在讲解这一节之前，请大家首先阅读《广播初始化与广播切换》和《蓝牙 5.0GAP 与 GATT 详解》这两篇内容，以上内容是一个基础概念。

1：nRF52xx 蓝牙 BLE 广播内容参数

很多读者希望在广播里最大化的广播自己的参数内容。广播更多的参数，本节我们的目标就是实现在广播中自定义内容参数。首先我们要明确广播包到底有多大的空间？

在蓝牙 4.x 的协议中，广播包的大小为 31 个字节，如果主机才有主动扫描，还有一个 31 字节的大小的广播回包，也就是说如果是蓝牙 4.x 模式，最大可要实现 62 个字节大小的空间。

蓝牙 5.0 把广播信道抽象为两类，一种叫主广播信道(primary advertisement channels)，另一种叫次广播信道，或者第二广播信道（Secondary Advertising Packets）。

所谓的主广播类似于蓝牙 4.x 的广播，只工作在 37，38，39 三个信道，最大广播字节为 31 字节。而次广播允许蓝牙在除开 37,38,39 三个通道之外的其他 37 个信道上发送长度介于 0-255 字节的数据。次广播信道(0-36 channel)广播 255 字节数据。本讲主要讲述主广播的配置，关于次广播后面会专门写一篇文章。

在 ble_advertising.h 文件中，提供了广播的初始化参数结构体如下所示：

```
typedef struct
{
    ble_advdata_t                 advdata;          /*广播包数据*/
    ble_advdata_t                 srdata;           /*广播回应包数据*/
    ble_adv_modes_config_t        config;           /*选择使用哪种广播模式和时段*/
    ble_adv_evt_handler_t         evt_handler;      /*将在广播事件上调用的事件处理程序。*/
    ble_adv_error_handler_t error_handler;     /*错误处理程序，它将把内部错误导入主应用程序。*/
} ble_advertising_init_t;
```

如果你需要自定义广播内容，那么就就需要在广播包数据 advdata 或者广播回包数据 srdata 中添加内容，关于广播包和广播回包的概念已经在广播初始化那篇教程中讲述了，这里就不累述了，

下面我们主要是讨论广播包和广播回包中可以定义的内容有哪些？

首先可以观察到 advdata 和 srdata 都是结构体 ble_advdata_t 类型，该结构体内定义了广播包或者广播回包内可以定义的内容，结构体如下所示：

```
01. typedef struct
02. {
03.     ble_advdata_name_type_t     name_type;          /*设备名称的类型*/
04.     uint8_t                     short_name_len;     /*短设备名称的长度(如果指定了短类型) */
05.     bool                        include_appearance;     /*确定是否包括外观 */
06.     uint8_t                     flags;                  /*广播数据标志字段 */
07.     int8_t                      *p_tx_power_level;      /*TX 电平发送功率等级. */
08.     ble_advdata_uuid_list_t     uuids_more_available;   /*部分服务 UUID 列表. */
09.     ble_advdata_uuid_list_t     uuids_complete;     /*服务 UUID 的完全列表 */
10.     ble_advdata_uuid_list_t     uuids_solicited;    /*请求服务的 UUID 列表 */
11.     ble_advdata_conn_int_t *    p_slave_conn_int;   /*从机连接间隔的范围*/
12.     ble_advdata_manuf_data_t    * p_manuf_specific_data;    /*制造商特定的数据 */
13.     ble_advdata_service_data_t  * p_service_data_array;     /*服务数据结构数组 */
14.     uint8_t                     service_data_count;     /*服务数据结构的数量。 */
15.     bool                        include_ble_device_addr;    /*确定是否包含 LE 蓝牙设备地址 */
16.     ble_advdata_le_role_t       le_role;
17.     /*LE 角色域. 这个角色区域仅仅用于 NFC.对应 BLE 广播，设置为 NULL. */
18.     ble_advdata_tk_value_t      * p_tk_value;
19.     /*安全管理 TK 值的区域. 这个角色区域仅仅用于 NFC.对应 BLE 广播，设置为 NULL.*/
20.     uint8_t                     *p_sec_mgr_oob_flags;
21.    /*安全管理器带外标志字段. 这个角色区域仅仅用于 NFC.对应 BLE 广播，设置为 NULL.*/
22.     ble_gap_lesc_oob_data_t     *p_lesc_data;
23.     *LE OOB 数据的安全连接. 这个角色区域仅仅用于 NFC.对应 BLE 广播，设置 NULL.*/
24. } ble_advdata_t;
```

其中，

①广播包内如果需要修改如下内容的：

name_type:设备名称的类型

short_name_len:短设备名称的长度(如果指定了短类型)

include_appearance:确定是否包括外观

参考教程《蓝牙 5.0GAP 与 GATT 详解》这一章中讲解了。

②广播包内如果需要修改如下内容的：

flags 广播数据标志字段，

参考教程《广播初始化与广播切换》这一章中讲解了。

③广播包内如果需要修改如下内容的：

p_tx_power_level：TX 电平发送功率等级

参考教程《信号发射功率的设置》这一章中讲解了。

这几个部分之前已经描述过，下面想讨论下其他几个广播内容参数的实现：

2：自定义广播的实现

2.1 广播包中包含 UUID 的值

UUID 的种类分为两种：一种是 SIG 定义的公共服务 UUID，所有的公共服务共用一个 128bit 的基础 uuid，不同的服务采用一个 16bit uuid 进行定义。另外一种就是私有服务的 UUID,这是一个自定义的 128bit UUID。关于 UUID 的具体区别和用法，请参看《UUID 的总结详细》这章的内容，这里我们主要讨论如果在广播中广播 UUID 的值。注意，广播包里的 UUID 不影响服务特征值中 UUID 的值，仅仅是让广播把 UUID 的值广播给扫描者，方便观察。在 ble_gap.h 文件中列成了 UUID 的类型：

ble_gap.h

```
/**@defgroup BLE_GAP_EVT_ADV_SET_TERMINATED_REASON GAP Advertising Set Terminated reasons
 * @{ */
#define BLE_GAP_EVT_ADV_SET_TERMINATED_REASON_TIMEOUT             0x01  /**< Timeout value reached. */
#define BLE_GAP_EVT_ADV_SET_TERMINATED_REASON_LIMIT_REACHED       0x02  /**< @ref ble_gap_adv_params_t::max_adv_evts
/**@} */

/**@defgroup BLE_GAP_AD_TYPE_DEFINITIONS GAP Advertising and Scan Response Data format
 * @note Found at https://www.bluetooth.org/Technical/AssignedNumbers/generic_access_profile.htm
 * @{ */
#define BLE_GAP_AD_TYPE_FLAGS                                0x01 /**< Flags for discoverability. */
#define BLE_GAP_AD_TYPE_16BIT_SERVICE_UUID_MORE_AVAILABLE    0x02 /**< Partial list of 16 bit service UUIDs. */
#define BLE_GAP_AD_TYPE_16BIT_SERVICE_UUID_COMPLETE          0x03 /**< Complete list of 16 bit service UUIDs. */
#define BLE_GAP_AD_TYPE_32BIT_SERVICE_UUID_MORE_AVAILABLE    0x04 /**< Partial list of 32 bit service UUIDs. */
#define BLE_GAP_AD_TYPE_32BIT_SERVICE_UUID_COMPLETE          0x05 /**< Complete list of 32 bit service UUIDs. */
#define BLE_GAP_AD_TYPE_128BIT_SERVICE_UUID_MORE_AVAILABLE   0x06 /**< Partial list of 128 bit service UUIDs. */
#define BLE_GAP_AD_TYPE_128BIT_SERVICE_UUID_COMPLETE         0x07 /**< Complete list of 128 bit service UUIDs. */
#define BLE_GAP_AD_TYPE_SHORT_LOCAL_NAME                     0x08 /**< Short local device name. */
#define BLE_GAP_AD_TYPE_COMPLETE_LOCAL_NAME                  0x09 /**< Complete local device name. */
#define BLE_GAP_AD_TYPE_TX_POWER_LEVEL                       0x0A /**< Transmit power level. */
#define BLE_GAP_AD_TYPE_CLASS_OF_DEVICE                      0x0D /**< Class of device. */
#define BLE_GAP_AD_TYPE_SIMPLE_PAIRING_HASH_C                0x0E /**< Simple Pairing Hash C. */
#define BLE_GAP_AD_TYPE_SIMPLE_PAIRING_RANDOMIZER_R          0x0F /**< Simple Pairing Randomizer R. */
#define BLE_GAP_AD_TYPE_SECURITY_MANAGER_TK_VALUE            0x10 /**< Security Manager TK Value. */
#define BLE_GAP_AD_TYPE_SECURITY_MANAGER_OOB_FLAGS           0x11 /**< Security Manager Out Of Band Flags. */
#define BLE_GAP_AD_TYPE_SLAVE_CONNECTION_INTERVAL_RANGE      0x12 /**< Slave Connection Interval Range. */
#define BLE_GAP_AD_TYPE_SOLICITED_SERVICE_UUIDS_16BIT        0x14 /**< List of 16-bit Service Solicitation UUIDs. */
#define BLE_GAP_AD_TYPE_SOLICITED_SERVICE_UUIDS_128BIT       0x15 /**< List of 128-bit Service Solicitation UUIDs. */
#define BLE_GAP_AD_TYPE_SERVICE_DATA                         0x16 /**< Service Data - 16-bit UUID. */
#define BLE_GAP_AD_TYPE_PUBLIC_TARGET_ADDRESS                0x17 /**< Public Target Address. */
#define BLE_GAP_AD_TYPE_RANDOM_TARGET_ADDRESS                0x18 /**< Random Target Address. */
#define BLE_GAP_AD_TYPE_APPEARANCE                           0x19 /**< Appearance. */
#define BLE_GAP_AD_TYPE_ADVERTISING_INTERVAL                 0x1A /**< Advertising Interval. */
#define BLE_GAP_AD_TYPE_LE_BLUETOOTH_DEVICE_ADDRESS          0x1B /**< LE Bluetooth Device Address. */
#define BLE_GAP_AD_TYPE_LE_ROLE                              0x1C /**< LE Role. */
#define BLE_GAP_AD_TYPE_SIMPLE_PAIRING_HASH_C256             0x1D /**< Simple Pairing Hash C-256. */
#define BLE_GAP_AD_TYPE_SIMPLE_PAIRING_RANDOMIZER_R256       0x1E /**< Simple Pairing Randomizer R-256. */
#define BLE_GAP_AD_TYPE_SERVICE_DATA_32BIT_UUID              0x20 /**< Service Data - 32-bit UUID. */
#define BLE_GAP_AD_TYPE_SERVICE_DATA_128BIT_UUID             0x21 /**< Service Data - 128-bit UUID. */
#define BLE_GAP_AD_TYPE_LESC_CONFIRMATION_VALUE              0x22 /**< LE Secure Connections Confirmation Value */
#define BLE_GAP_AD_TYPE_LESC_RANDOM_VALUE                    0x23 /**< LE Secure Connections Random Value */
#define BLE_GAP_AD_TYPE_URI                                  0x24 /**< URI */
#define BLE_GAP_AD_TYPE_3D_INFORMATION_DATA                  0x3D /**< 3D Information Data. */
#define BLE_GAP_AD_TYPE_MANUFACTURER_SPECIFIC_DATA           0xFF /**< Manufacturer Specific Data. */
/**@} */
```

那么首先，在广播参数里列出了三类 UUID 列表的情况，下面来一一进行介绍：

```
01.     ble_advdata_uuid_list_t     uuids_more_available;    /*部分服务 UUID 列表. */
02.     ble_advdata_uuid_list_t     uuids_complete;      /*服务 UUID 的完全列表 */
03.     ble_advdata_uuid_list_t     uuids_solicited;     /*请求服务的 UUID 列表 */
```

uuids_more_available：英文翻译为更多可能的 UUID，官方的解释为只显示部分 UUID 列表在广播中，实际工程中还有更多 UUID。我们可以认为广播中显示部分 UUID 列表。

uuids_complete：英文翻译为完全体的 UUID，也就是说在广播中显示工程中的全部的 UUID。

uuids_solicited：称为请求服务的 UUID 列表。一个从机设备可以发送服务请求数据类型广播去邀请主机设备进行连接，该主机设备包含一个或者多个这个服务器请求数据广播所指定的服务。

① 首先谈谈如何在广播中广播 uuids_complete，广播中，UUID 专门有一个结构体进行标识，如下所示：

```
01. typedef struct
02. {
03.     uint16_t        uuid_cnt;       /*UUID 的数目. */
```

```
    ble_uuid_t *            p_uuids;        /*指向 UUID 数组条目的指针 */
} ble_advdata_uuid_list_t;
```

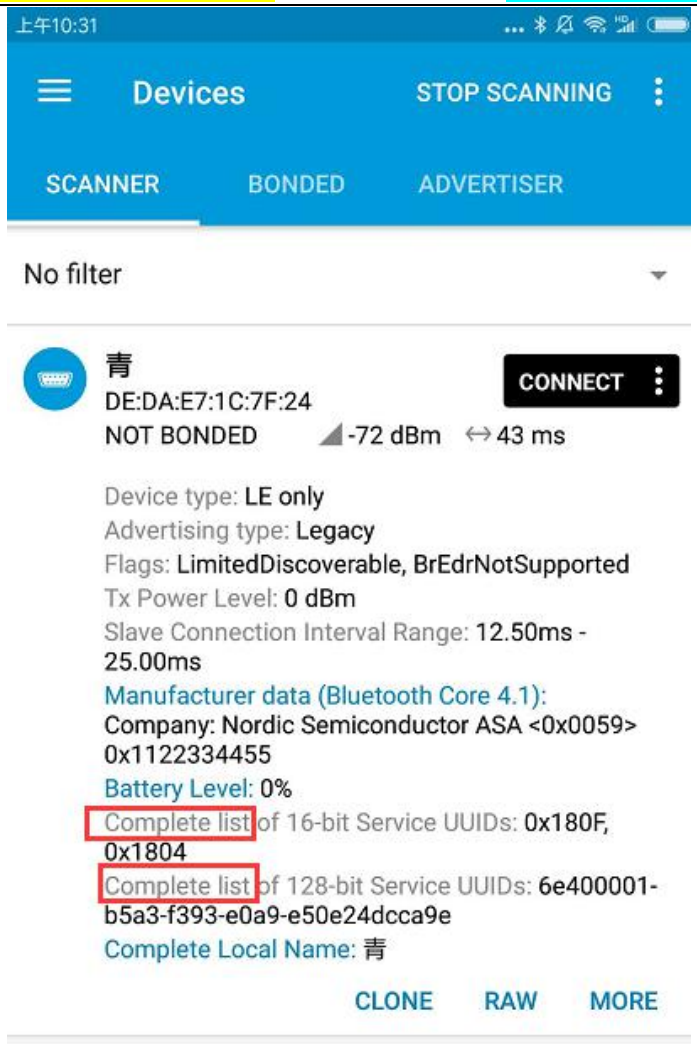
这两个参数，一个表示 UUID 的数目，一个表示指向需要广播的 UUID 列表的指针。所有我们如果需要在广播中广播 UUID，需要专门建立一个 UUID 的结构体，让这个指针指向这个结构体。比如，在主函数 main.c 文件中，声明如下 m_adv_uuids 结构体：

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

这个结构体内包含了 3 个 UUID 的列表；一个蓝牙串口，一个电池服务，一个 TX 服务。同时标注服务 UUID 的类型，其中串口服务为私有服务，电池服务和 TX 服务为 SIG 定义的公共服务。两种服务类型 UUID 长度是不同的。我们在广播初始化下添加如下代码：

```
static void advertising_init(void)
{
    uint32_t                    err_code;
    ble_advertising_init_t init;

    .....
    .....
    /**********************下面是广播回包****************/
    //定义完全 UUID 包含（一个 128bit  的 uuid 和两个服务 16bit uuID）到广播包中
    init.srdata.uuids_complete.uuid_cnt = sizeof(m_adv_uuids) / sizeof(m_adv_uuids[0]);
    init.srdata.uuids_complete.p_uuids  = m_adv_uuids;

    .....
    .....
    err_code = ble_advertising_init(&m_advertising, &init);
    APP_ERROR_CHECK(err_code);

    ble_advertising_conn_cfg_tag_set(&m_advertising, APP_BLE_CONN_CFG_TAG);
}
```

这里由于要显示 128bit 的 uuid，长度比较长，因此把 UUID 的参数放入广播包回包中显示。编译后，通过手机 app nrf connect 扫描后显示如下图所示，图中显示 Complete list of，表示是完整的 UUID 列表：



②再谈谈 uuids_more_available 类型，该类型只是显示部分 UUID 列表在广播中，实际工程中还有更多 UUID。那么设置中 UUID 的数目是可以控制的，通过设置 uuid_cnt 的数目控制广播中显示的 UUID 数目，不管广播包还是广播回包，只提供 31 字节空间，因此需要注意可使用的空间。具体代码设置如下所示：

```
20. static void advertising_init(void)
21. {
22.     uint32_t                 err_code;
23.     ble_advertising_init_t init;
24.
25.     ....................................
26.     ....................................
27.     /**********************下面是广播回包*****************/
28.     //定义完全 UUID 包含（一个 128bit 的 uuid 和两个服务 16bit uuID）到广播包中
29.     //控制广播中广播的 UUID 的数目
30.     init.srdata.uuids_more_available.uuid_cnt = sizeof(m_adv_uuids) / sizeof(m_adv_uuids[0])-1;
31.     //广播的 UUID 结构体
32.     init.srdata.uuids_more_available.p_uuids= m_adv_uuids;
33.
34.     ....................................
```

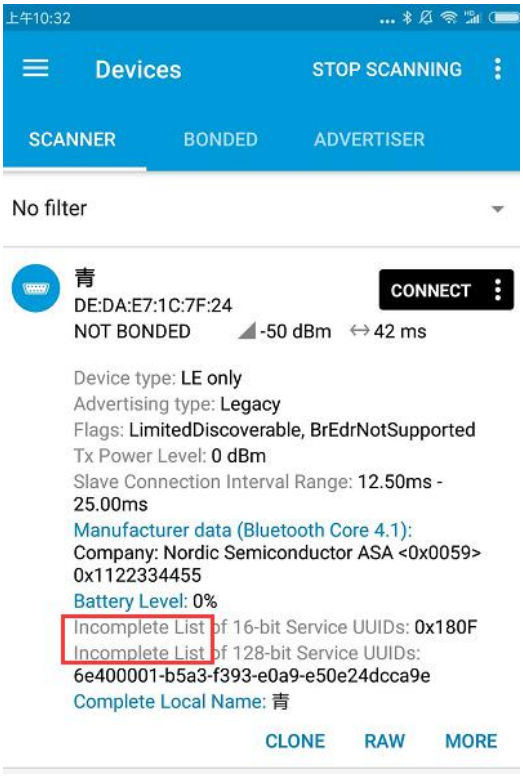
```
35.     ......................................
36.       err_code = ble_advertising_init(&m_advertising, &init);
37.       APP_ERROR_CHECK(err_code);
38.
39.       ble_advertising_conn_cfg_tag_set(&m_advertising, APP_BLE_CONN_CFG_TAG);
40. }
```

uuid_cnt 参数的大小决定了需要在广播中广播出去的 UUID，而应用中有更多的 UUID 没有显示的。这个应用的可能有以下的原因：

*广播由于空间有限，由于 UUID 数量很多，不可能让广播中就行显示所有的 UUID 的值。

*客户不希望广播阶段就透露了该应用的所有 UUID 值，因为广播是可以被任何主机扫描所能观察到的，如果需要保密，连接下则可以添加密钥配对。当用户需要保护部分服务不被知道，可以采用这种方式。

把工程中广播修改入上代码后，编译通过。再下载到开发板中，通过手机 app nrf connect 扫描后显示如下图所示，图中显示 Incomplete list of，表示不完全的 UUID 列表，显示了一个 16bit uuid 和一个 128bit uuid：

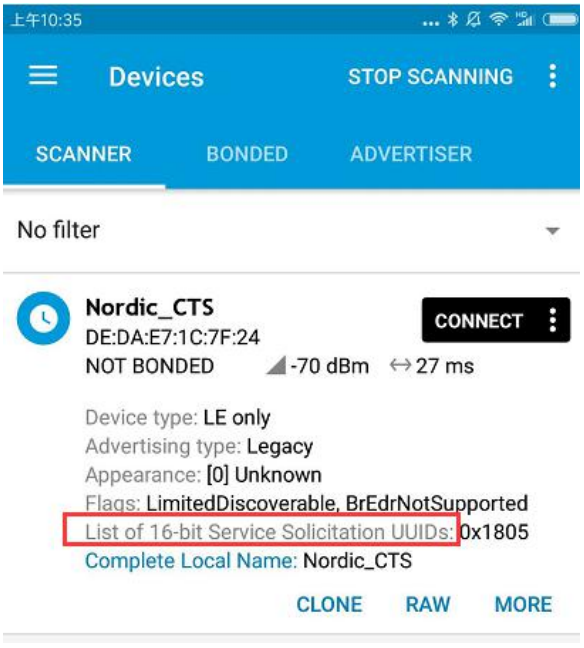


③一个典型的请求服务的 UUID 列表的例子就是 ANCS 广播。这个例子上一个 GATT 外设端，同时需要一个有这个 ANCS 的 GATT 服务的主机端。通过在广播中广播 ANCS 请求服务的的 UUID，去告诉扫描端（ios 设备）它正在“寻找”一个具有 ANCS 服务的主机设备。对于当前时间服务 CTS 的客户机也是如此。ble_app_cts_c 使用所请求的服务是因为它需要一个具有当前时间服务服务器的 gatt 服务器的中心设备，在 ble_app_cts_c 工程中，广播初始化的代码具体如下所示：

```
01. static ble_uuid_t m_adv_uuids[] = {{BLE_UUID_CURRENT_TIME_SERVICE,
    BLE_UUID_TYPE_BLE}};
02.
03. static void advertising_init()
```

```
04. {
05.     ret_code_t                  err_code;
06.     ble_advertising_init_t init;
07.
08.     memset(&init, 0, sizeof(init));
09.
10.     init.advdata.name_type          = BLE_ADVDATA_FULL_NAME;
11.     init.advdata.include_appearance         = true;
12.     init.advdata.flags      = BLE_GAP_ADV_FLAGS_LE_ONLY_LIMITED_DISC_MODE;
13.     init.advdata.uuids_solicited.uuid_cnt = sizeof(m_adv_uuids) / sizeof(m_adv_uuids[0]);
14.     init.advdata.uuids_solicited.p_uuids    = m_adv_uuids;
15.
41.       .....
42.     .....
16.
17.     init.evt_handler = on_adv_evt;
18.
19.     err_code = ble_advertising_init(&m_advertising, &init);
20.     APP_ERROR_CHECK(err_code);
21.
22.     ble_advertising_conn_cfg_tag_set(&m_advertising, APP_BLE_CONN_CFG_TAG);
23. }
```

把工程编译后，下载到开发板中，通过手机 app nrf connect 扫描后显示如下图所示：



2.2 广播包中包含从机的连接间隔参数

从机与主机直接的连接间隔，是由从机提出与主机进行协商，然后再由主机决定的参数，在

《GAP 初始化》和《连接参数初始化》这两章教程里已经相信的探讨过连接参数的概念，这里面我们来实现如何在广播中把连接间隔广播出去。在广播数据结构体中给了一个结构体参数 ble_advdata_conn_int_t，指定了广播中可以广播的两个连接参数的值,如下所示:

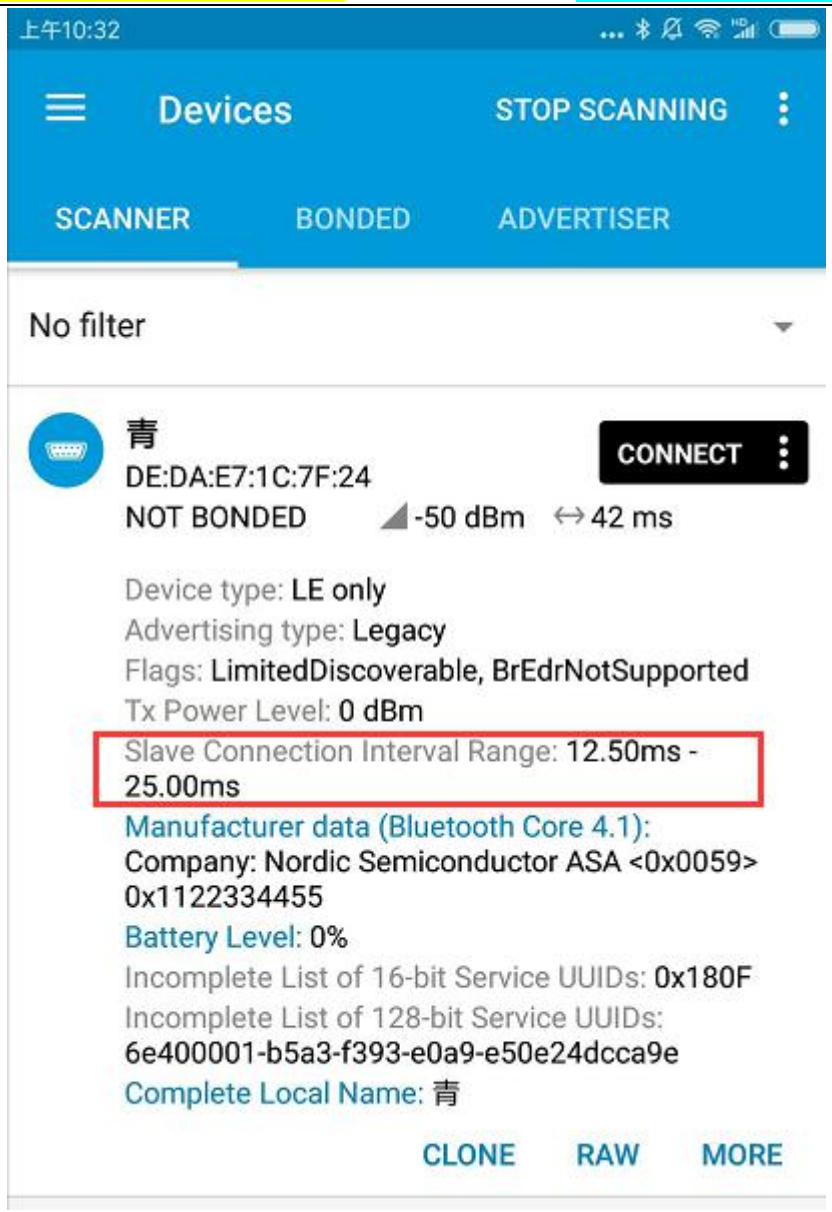
```
typedef struct
{
    uint16_t                min_conn_interval;
    /*最小的连接间隔值，以 1.25 ms 为一个单位，范围在 6 在 3200 个单位 (也就是 7.5 ms 到 4 s). */
    uint16_t                max_conn_interval;
/*最大的连接间隔值 l， 以 1.25 ms 为一个单位，范围在 6 to 3200(也就是 7.5 ms 到 4 s). 当参数值为 0xFFFF 表示没有最大的特定值 */
} ble_advdata_conn_int_t;
```

使用这个结构体，可以在广播初始化中定义需要广播的连接间隔的参数，具体代码如下所示：

```
 static void advertising_init()
{
    ret_code_t          err_code;
    ble_advertising_init_t init;

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

    .....
    .....

 //从设备连接间隔范围最小值：10*1.25ms = 12.5ms
  conn_range.min_conn_interval = 10;
  //从设备连接间隔范围最大值：20*1.25ms = 25ms
  conn_range.max_conn_interval = 20;
  //广播数据中包含从设备连接间隔范围
    init.advdata.p_slave_conn_int = &conn_range;
    .....
    .....

  err_code = ble_advertising_init(&m_advertising, &init);
    APP_ERROR_CHECK(err_code);

    ble_advertising_conn_cfg_tag_set(&m_advertising, APP_BLE_CONN_CFG_TAG);
}
```

配置结构体中的连接间隔时间后，通过 ble_advertising_init 函数配置到广播中。

修改代码后，编译后成功后下载到开发板中，通过手机 app nrf connect 扫描后显示如下图所示：



2.3 广播包中包含制造商的自定义参数

制造商或者厂家的自定义参数 manuf_specific_data 是根据制造商的需求，在广播中加入制造商自定义的数据，该数据格式可以更加制造商的要求自定义，最典型的一个应用就是 ibeacon 协议，ibeacon 协议就是苹果公司自定义广播数据格式，关于 ibeacon 的应用详细参考《ibeacon 原理与应用详解》这章内容。下面我们来自定义一组广播数据。首先，在 ble_advdata.h 文件中，定义了结构体 ble_advdata_manuf_data_t 表示公司厂家的数据与 ID 代码，如下所示：

```
01. typedef struct
02. {
03.     uint16_t        company_identifier;          /*公司的 ID 代码. */
04.     uint8_array_t   data;                        /*制造者自定义的数据. */
05. } ble_advdata_manuf_data_t;
```

这个结构体就两个参数，一个参数为公司的 ID 代码，一个为制造商自定义的数据。

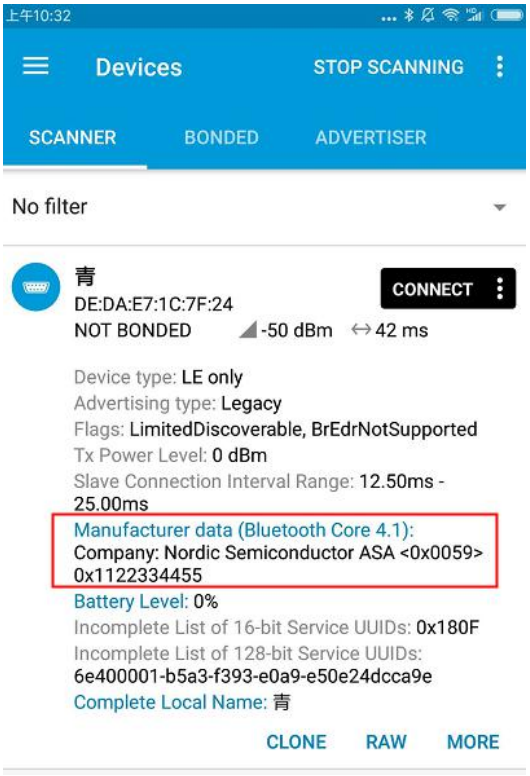
◎公司的 ID 号每个公司都有独立申请的值，一般 company_identifier 都是厂家在 SIG 申请定义的唯一 ID 号。不同公司的 ID 号可以具体在 SIG 网站查询。我们在例子中才有 Nordic 的制造商 ID 进行演示。Nordic 的制造商的 ID 号为：0x0059

◎制造商自定义的数据：这个参数可以自由的设置，只有广播包的空间足够，就可以任意的布置，下面例子中，我们定义 5 个字节的自定义数据 0x11,0x22,0x33,0x44,0x55，这些数据内容和数据长度都可以自由修改。

具体的声明代码如下所示：

```
uint8_t     my_adv_manuf_data[5] = {0x11,0x22,0x33,0x44,0x55};
//0x0059 是 Nordic 的制造商 ID
 manuf_specific_data.company_identifier = 0x0059;
 //指向制造商自定义的数据
manuf_specific_data.data.p_data = my_adv_manuf_data;
 //制造商自定义的数据大小
 manuf_specific_data.data.size = sizeof(my_adv_manuf_data)
```

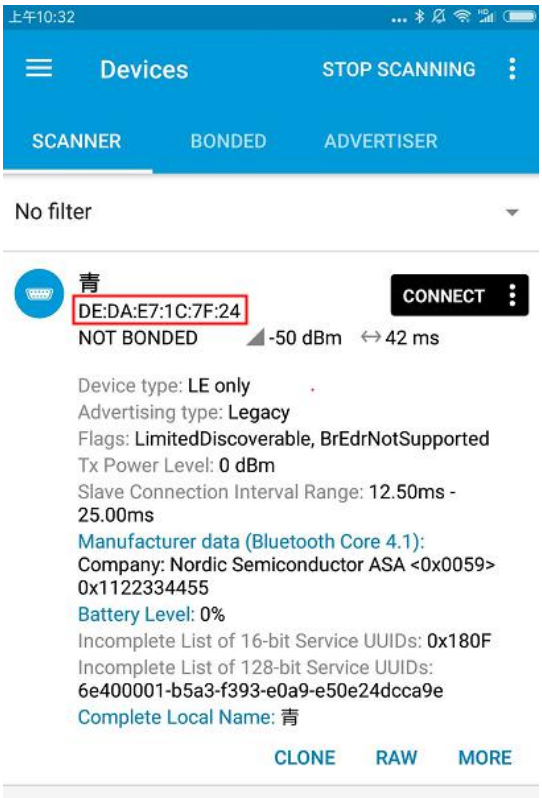
配置广播初始化中的结厂家自定义参数构体后，再通过 ble_advertising_init 函数配置到广播中。在广播初始化中修改代码后，编译成功后下载到开发板中，通过手机 app nrf connect 扫描后显示如下图所示，显示了自定义的广播参数：



2.4 广播包中包含蓝牙设备地址

蓝牙设备地址也称为 MAC 地址，MAC 地址是广播包中自带的，因此不需要在广播初始化中进行配置。Mac 地址实际上不属于自定义的广播内容,可以见广播包抓包分析的那篇内容。如果需要

修改，则需要调用专用的 API 函数接口 sd_ble_gap_address_get 函数进行修改，具体内容请参看教程《蓝牙 MAC 地址》章节。



3：动态广播的切换

当有用户需要在广播中广播某个应用的数据，比如需要在广播包中观察设备的电量状态等参数值，这时候我们就需要考虑如何实现在广播包中包含服务数据。广播包中包含服务数据并不能替代蓝牙服务的建立，它只是把蓝牙服务中的数据传输到广播包中，已广播包的形式发送出去。

3.1 广播包中包含服务数据

在这篇文章最开头，在 ble_advdata_t 结构体中分别声明了两个参数：

```
ble_advdata_service_data_t    * p_service_data_array;          /*服务数据结构数组  */
uint8_t                                service_data_count;          /*服务数据结构的数量。  */
```

其中专门针对蓝牙服务提供了一个结构体定义 ble_advdata_service_data_t，该结构体定义了广播中可以广播的蓝牙服务参数，具体代码如下所示：

```
01. typedef struct
02. {
03.     uint16_t        service_uuid;                    /*服务的 UUID. */
04.     uint8_array_t   data;                            /*服务的数据*/
```

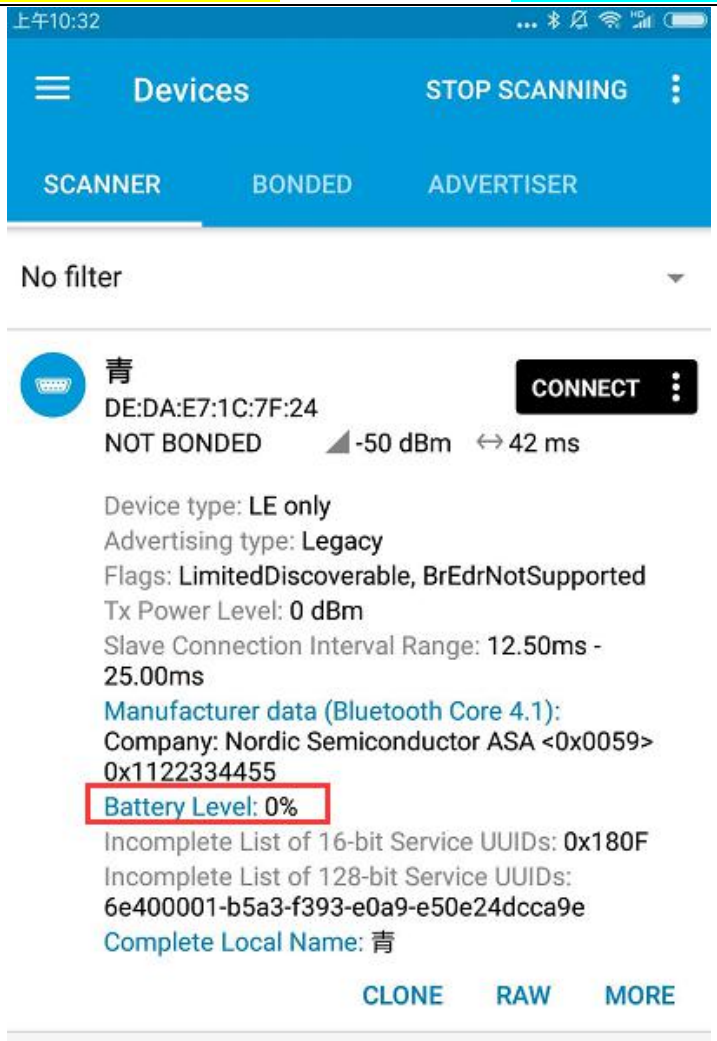
```
05. } ble_advdata_service_data_t;
```

这个结构体内包含两个参数，一个参数表示该服务的 UUID 号，一个表示要传递的服务参数值，比如下面实验的例子，我们来传递一个电池电量的参数，那么电池服务的 UUID 就是一个 SIG 的 16bit 公共服务 UUID，手机设备可以识别为电池 BATTERY_SERVICE。电池电量的参数为我们在 adc 中采集的电压量，转换的电量百分比的值，关于电池服务的这个参数，参考《电池服务》这一章。我们把采集到的电池电量的百分比的值分配给 data.p_data 参数中去。代码如下所示：

```
01.   ble_advdata_service_data_t   service_data;
02.  //电池电量服务 UUID
03.    service_data.service_uuid = BLE_UUID_BATTERY_SERVICE;
04.   service_data.data.size = sizeof(percentage_batt_lvl);
05.    service_data.data.p_data = &percentage_batt_lvl;
06.      //广播数据中加入服务数据
07.    init.advdata.p_service_data_array = &service_data;
08.   //因为只加入一个服务数据，所以服务数据数量设置为 1
09.    init.advdata.service_data_count = 1;
```

同时，需要在广播初始化中，定义下需要加入的服务的数量，由于演示中只演示了电池服务一个服务，因此，该值可以直接设为 1。

配置广播初始化中的服务参数构体后，再通过 ble_advertising_init 函数配置到广播中。在广播初始化中修改代码后，编译成功后下载到开发板中，通过手机 app nrf connect 扫描后显示如下图所示，显示了服务的广播参数：



我们发现，可以在广播中，可以看到电池电量的参数值。但是，这个值一直为 0，这是什么原因了？仔细分析，我们最初在广播初始化中，导入的电池电量初始化为 0，而广播初始化函数只是在 main 主函数中初始化了一次，也就是说这个参数值只是最开头定义的初始化值 0。当我们通过 adc 采集到电池电量后，电池电量的参数随之更新。但是广播初始化函数却没有在任何的回调函数中出现，也就不能动态的更新电池电量的值了。

3.2 服务数据的更新

如果我们需要更新广播中服务参数的值，相当于要实现一个动态广播的功能。动态广播的基本思路就是需要在对应的任何的回调函数中来修改广播的参数值。

回调函数的选择可选的场景很多，比如如下几种情况：

◎当 adc 采集了电压完成后，立即去更新广播参数值。这时候可以在 adc 采集参数后，触发中进行广播参数的更新。

◎当某个外部中断，比如按键按下后，立即去触发广播参数的更新。这个时候可以在按键触发的中断回调中进行参数更新。

◎当每过一段时间，设备去读取服务的参数值，来定时的触发广播参数的更新。这时候可以采用软件定时器或者硬件定时器的中进行更新。

当然出现回调函数触发广播更新的情况不仅仅只有上面几种情况，对应每种状态，我们都可

以在对应的情况下进行处理。

再来谈谈广播更新的问题。广播更新之前，需要首先停止广播，然后再重新配置广播参数，再开启广播这样一个过程。这三个步骤都有对应的 API 可以直接调用。下面我们演示在按键中触发广播更新的代码，具体代码如下所示：

```
void bsp_event_handler(bsp_event_t event)//板级处理事件
{
    uint32_t err_code;
    switch (event)
    {
        .....
        .....

        case BSP_EVENT_KEY_3:
                //停止当前广播
                 sd_ble_gap_adv_stop(m_advertising.adv_handle);
                //重新广播初始化，带入更新后的服务参数值到广播包中
                 advertising_init();
                  //重新开始广播
                 advertising_start();
        default:
            break;
    }
}
```

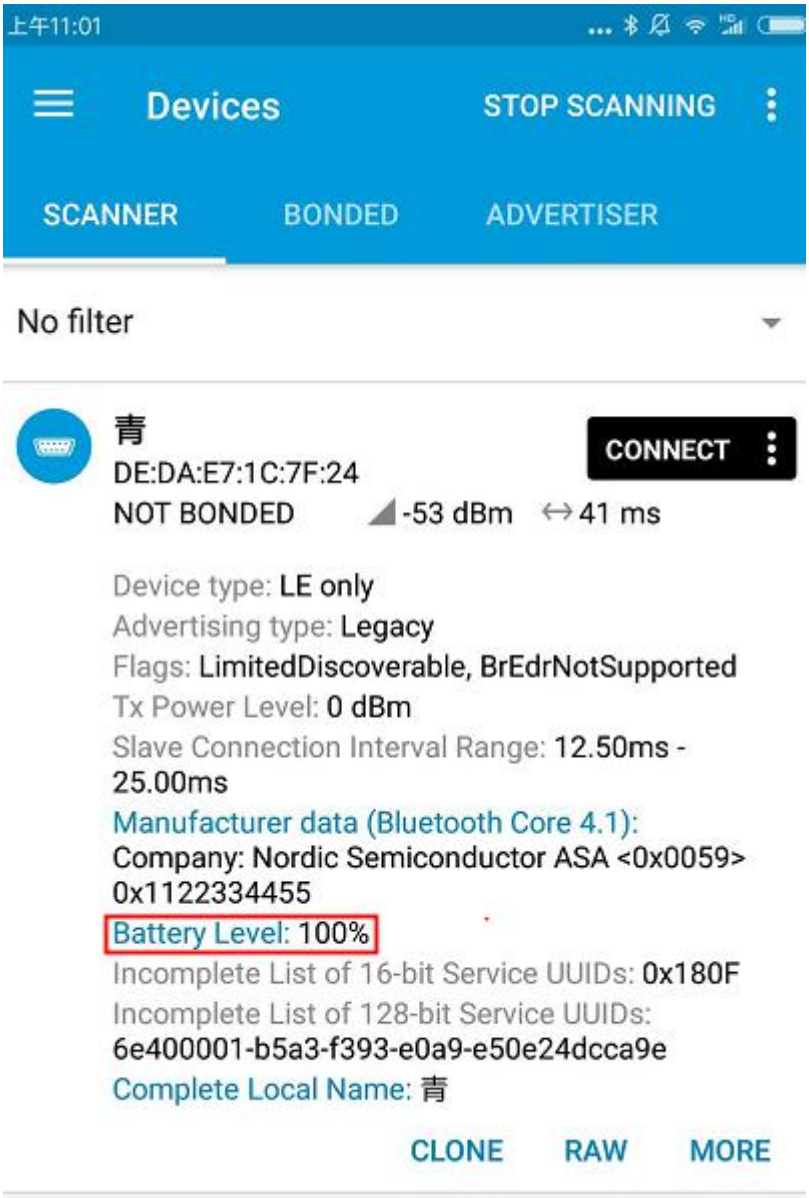
在按键中断回调函数中，配置 BSP_EVENT_KEY_3 触发事件。关于协议栈下按键的触发应用配置请参看《协议栈下按键的使用》这章的教程，这里就不累述了，我们只来谈谈在 BSP_EVENT_KEY_3 触发事件下，调入如下三个 API 函数：

①停止当前广播 sd_ble_gap_adv_stop(m_advertising.adv_handle);

②重新广播初始化，带入更新后的服务参数值到广播包中 advertising_init();

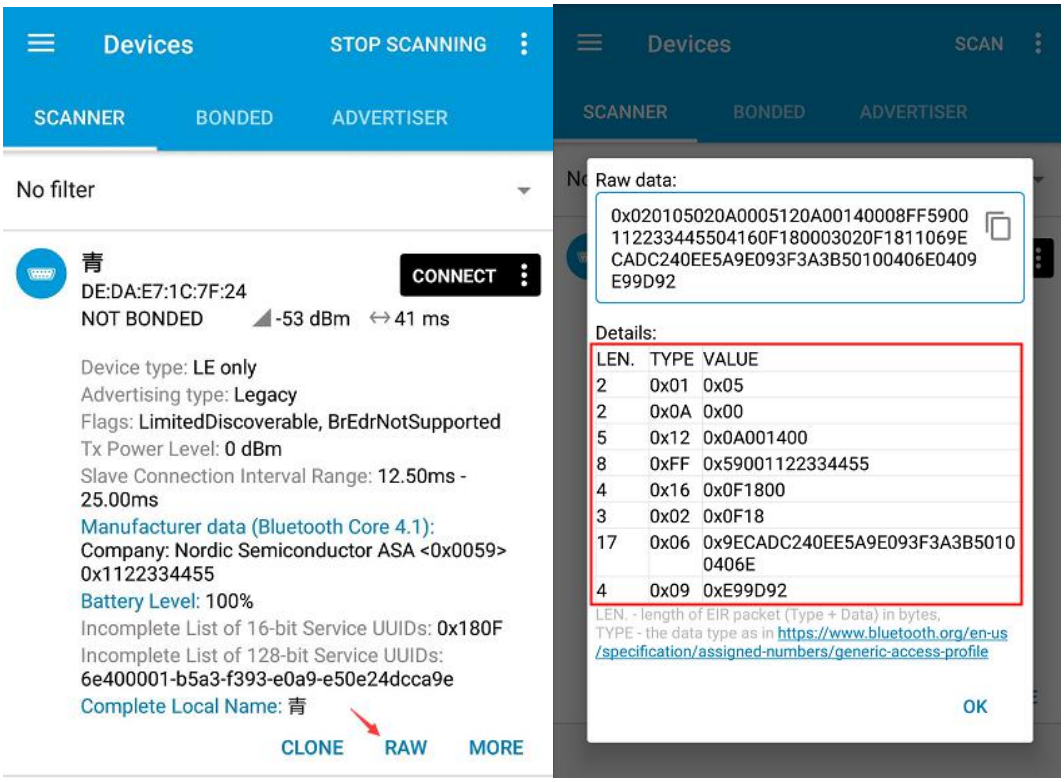
③重新开始广播 advertising_start();

这时候，如果你按下了开发板上的按键 4 之后，可以发现广播包会发生动态的更新，更新为最新的电池采集的电量百分比的值，如下图所示，显示当前电量 100%。



4：本章总结

本章主要讲述主广播信道下，如果配置广播内的内容。主广播包含了广播包和广播回包，广播包和广播回应包都是 31 字节，也就是说，在你设置自己的广播内容的时候，需要统计下，最后总的广播内容是否超出空间了，如果超出最大的空间，是会出现广播出错的状态的。对应广播的大小可以通过抓包器或者手机 app 进行观察，抓包器的方式有专门一讲抓包分析了。App 可以通过 nrf connect app，点开广播包下的 RAW 按钮，可以弹出广播数据的 16 进制类别，如下图所示：



比如第一个设备类型，LEN 为 0x02 一个字节，TYPE 为 0x01 一个字节，VALUE 为 0x05 一个字节，一共三个字节，依次类推。这点是开发者在自定义广播的时候需要特别注意的问题。