使用恩智浦 BLE 协议栈创建自定义的 Profile-服务器端(Server)

原文：https://community.nxp.com/docs/DOC-341317

Bluetooth®LowEnergy（或 BLE,低功耗蓝牙）是一种无线技术，允许在拥有数据的设备（服务器端 Server）和请求数据的设备（客户端 Client）之间进行信息交换。服务器通常是电池供电的小型设备，与传感器或执行器相连接，以收集数据或执行某些操作，而客户端通常是在系统中使用该信息或显示信息给用户的设备（最常见的客户端设备是智能手机）。

在创建自定义 BLE Profile 时，我们需要同时考虑它在服务器端和在客户端上的实现。服务器端要包含一个数据库，里面的数据可以被访问和修改；客户端要通过驱动程序去访问这些数据，并处理返回值。

这篇文章解释了如何使用恩智浦 BLE 协议栈实现一个自定义的 profile 的服务器端。我们在 MKW40Z160 上，通过一个电位器的例子来说明。

通用属性配置文件（Generic Attribute Profile）

在开始之前，我们需要熟悉 BLE 的通信方式。Generic Attribute Profile（GATT）协议负责在 BLE 连接之上交换 profile 数据和用户数据。所有标准的 BLE profile 均基于 GATT，并且必须遵循它的规范才能运行。
GATT 定义了两个通信中的角色：服务器和客户端。

- **GATT 服务器负责**存储数据，并接受来自 GATT 客户端的请求、命令和确认。
- **GATT 客户端**负责访问远端 GATT 服务器上的数据，通过读取、写入、通告或指示等操作。

图 1 GATT 客户端 - 服务器

GATT 使用属性（attribute）来提供数据，属性又被组织起来以描述 GATT 服务器中的信息。组织起来的属性可以是配置文件（Profile）、服务(Service)、特征(Characteristic)和描述符(Descriptor)。Profile 是高级定义，用于确定整个应用程序的行为（例如，是心率监视器还是温度传感器）。Profile 由一个或多个有独立功能的服务组成（例如，心率监测器需要一个心率传感器和一个电池测量单元）。服务由一个或多个特征组成，这些特征可以是测量值、控制点或其他数据（例如，心率传感器可能有一个心率特征和一个传感器位置特征）。最后，描述符定义了如何访问特征。

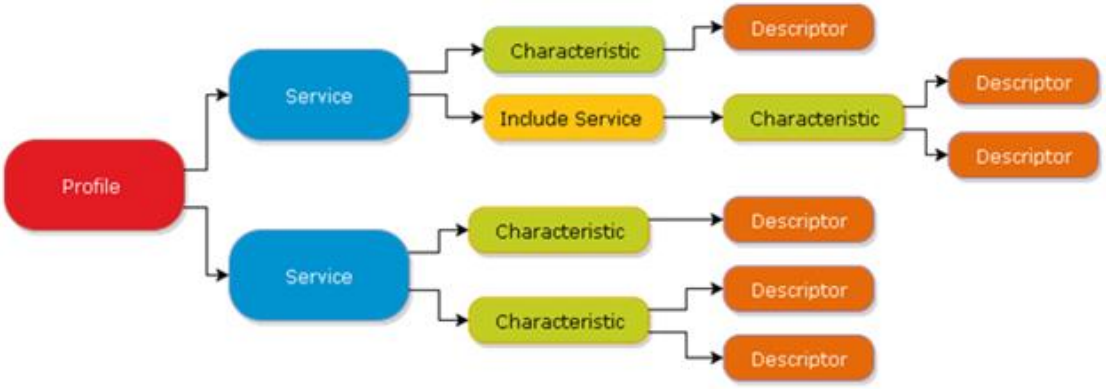

图 2 GATT 数据库结构

向 GATT 数据库中添加一个新服务（Service）

服务器端的 GATT 数据库仅包含描述服务、特征和描述符的属性。没有 profile 是因为它被隐含在了其中。每个 profile 由一组特定的服务构成。在恩智浦 Connectivity 软件中，可以用宏方便的定义数据库中的每个属性。

GATT 数据库中的每个服务和特征都有一个通用唯一标识符（UUID）。这些 UUID 由蓝牙标准化组织分配给被采纳的服务和特征。当使用自定义 profile 时，必须分配私有的 UUID。在恩智浦 Connectivity 软件中，自定义的 UUID 在文件 gatt_uuid128.h 中。必须使用宏 UUID128（name，bytes）来定义每个新的 UUID。其中 name 是一个标识符，可帮助我们稍后在代码中引用这个 UUID。bytes 变量是一个 16 个字节（128 位）的序列，它是自定义 UUID 的值。下面是电位计服务和与之相关的电位计相对值特性的定义。

```
/* Potentiometer Service */
UUID128(uuid_service_potentiometer, 0xE0, 0x1C, 0x4B, 0x5E, 0x1E, 0xEB,
0xA1, 0x5C, 0xEE, 0xF4, 0x5E, 0xBA, 0x04, 0x56, 0xFF, 0x02)

/* Potentiometer Characteristic */
UUID128(uuid_characteristic_potentiometer_relative_value, 0xE0, 0x1C,
0x4B, 0x5E, 0x1E, 0xEB, 0xA1, 0x5C, 0xEE, 0xF4, 0x5E, 0xBA, 0x04, 0x57, 0xFF,
0x02)
```

有了合适的 UUID 后，需要将新服务添加到 GATT 数据库中。它定义在文件 gatt_db.h 中。通过几个简单的宏指令将每个属性按正确的顺序组织起来。下面在 gatt_db 文件中的代码展示了如何实现电位计服务。

```
PRIMARY_SERVICE_UUID128(service_potentiometer,
uuid_service_potentiometer)
    CHARACTERISTIC_UUID128(char_potentiometer_relative_value,
uuid_characteristic_potentiometer_relative_value,
(gGattCharPropRead_c | gGattCharPropNotify_c))
```

```
        VALUE_UUID128(value_potentiometer_relative_value,
uuid_characteristic_potentiometer_relative_value,
(gPermissionFlagReadable_c ), 1, 0x00)
        CCCD(cccd_potentiometer)
        DESCRIPTOR(cpfd_potentiometer,
gBleSig_CharPresFormatDescriptor_d, (gPermissionFlagReadable_c),
7, gCpfdUnsigned8BitInteger, 0x00,
                   0xAD/*Unit precentage UUID in Little Endian (Lower
byte)*/,
                    0x27/*Unit precentage UUID in Little Endian
(Higher byte)*/,
                    0x01, 0x00, 0x00)
```

PRIMARY_SERVICE_UUID128（service_name，service_uuid）在 GATT 数据库中定义了一个使用 128 位自定义 UUID 的新服务。它需要两个参数：service_name 是该服务的名称，在后面的例子中会用到。service_uuid 是先前在 gatt_uuid128.h 中定义的服务的 UUID 的标识符。

CHARACTERISTIC_UUID128（characteristic_name，characteristic_uuid，flags）在上面的服务中定义了一个使用 128 位自定义 UUID 的新特性。它需要三个参数：characteristic_name 是代码中特征的名称，characteristic_uuid 是先前在 gatt_uuid128.h 中定义的特征的 UUID 的标识符。最后，flags 是所有特性属性（读取，写入，通告等）的合并。

VALUE_UUID128（value_name，characteristic_uuid，permission_flags，number_of_bytes，initial_values ...）定义上面的特征的值。value_name 是稍后在代码中用于读取或修改特征值的标识符。characteristic_uuid 是先前定义的特征的 UUID 标识符。permission_flags 决定了这个值的访问方式（只读，只写或二者均可）。number_of_bytes 确定值的字节数，后面跟着每个字节的初始值。

CCCD（cccd_name）为之前的特征定义一个新的描述符(Client Characteristic Configuration Descriptor)。cccd_name 是稍后将在代码中使用的 CCCD 的名称。这是一个可选项，取决于前面特征中的标志位。

DESCRIPTOR（descriptor_name，descriptor_format，permissions，size，descriptor_bytes ...）为之前定义的特征定义一个描述符。descriptor_name 定义该描述符的名称。descriptor_format 定义描述符的类型。permissions 确定了如何访问描述符。最后是描述符的大小和内容。

所有在 GATT 数据库中用到的宏，都在 BLEADG 里面的第 7 章“创建 GATT 数据库”中做了详细的描述（BLEADG 包含在恩智浦 Connectivity 软件文档中）。

为新服务制作驱动程序

当新服务在 gatt_db.h 中被定义后，就需要驱动程序来支持该服务并正确响应客户端的请求。为此，需要为每个新添加服务创建两个新文件：（service name）_service.c 和（service name）_interface.h。

service.c 文件包含与处理服务数据相关的的所有函数， interface.h 文件包含可以被应用程序引用的服务的定义。我们建议参考已有的.c 和.h 文件。

interface.h 头文件应当包含以下内容。

1. 服务配置的结构体，包括一个 16 位的 Service Handle 变量和 Service 中每个特征值所对应的变量。

```
/*! Potentiometer Service - Configuration */
typedef struct psConfig_tag
{
    uint16_t    serviceHandle;                  /*!<Service handle */
    uint8_t     potentiometerValue;             /*!<Input report field */
} psConfig_t;
```

2. 启动服务和停止服务的函数声明。这是初始化/取消初始化服务所需要的。

```
bleResult_t Ps_Start(psConfig_t *pServiceConfig);
bleResult_t Ps_Stop(psConfig_t *pServiceConfig);
```

3. 订阅和取消订阅的函数声明。这是订阅/取消订阅一个特定的客户端所需要的。

```
bleResult_t Ps_Subscribe(deviceId_t clientDeviceId);
bleResult_t Ps_Unsubscribe();
```

4. 根据您的应用程序所需，制作一系列用来读取，写入，更新某个或某些特征的函数。

```
bleResult_t Ps_RecordPotentiometerMeasurement (uint16_t
serviceHandle, uint8_t newPotentiometerValue);
```

service.c 文件应包括以下内容。

1. 一个 deviceId_t 变量，用于存储客户端的 ID。

```
/*! Potentiometer Service - Subscribed Client*/
static deviceId_t mPs_SubscribedClientId;
```

2. Start，Stop，Subscribe 和 Unsubscribe 的函数实现。Start 函数可以包含一些代码，用于为服务的特征值设置初始值。

```
bleResult_t Ps_Start (psConfig_t *pServiceConfig)
{
    /* Clear subscibed clien ID (if any) */
    mPs_SubscribedClientId = gInvalidDeviceId_c;

    /* Set the initial value defined in pServiceConfig to the characteristic values */
    return Ps_RecordPotentiometerMeasurement
(pServiceConfig->serviceHandle,
                                                pServiceConfig->potentiomete
rValue);
}

bleResult_t Ps_Stop (psConfig_t *pServiceConfig)
{
  /* Unsubscribe current client */
    return Ps_Unsubscribe();
}

bleResult_t Ps_Subscribe(deviceId_t deviceId)
{
   /* Subscribe a new client to this service */
    mPs_SubscribedClientId = deviceId;

    return gBleSuccess_c;
}

bleResult_t Ps_Unsubscribe()
{
   /* Clear current subscribed client ID */
    mPs_SubscribedClientId = gInvalidDeviceId_c;
    return gBleSuccess_c;
}
```

3. 服务相关函数的实现。用于写入、读取或通告服务里的特征的值。我们的例子只实现了两个：一个用于更新 GATT 数据库中的特征值的 public 函数，一个用于向客户端发布最近更新数据的通知（notification）的 local 函数。

```
bleResult_t Ps_RecordPotentiometerMeasurement (uint16_t
serviceHandle, uint8_t newPotentiometerValue)
{
    uint16_t  handle;
    bleResult_t result;

    /* Get handle of Potentiometer characteristic */
    result = GattDb_FindCharValueHandleInService(serviceHandle,
        gBleUuidType128_c,
(bleUuid_t*)&potentiometerCharacteristicUuid128, &handle);

    if (result != gBleSuccess_c)
        return result;

    /* Update characteristic value */
    result = GattDb_WriteAttribute(handle, sizeof(uint8_t),
(uint8_t*)&newPotentiometerValue);

    if (result != gBleSuccess_c)
        return result;

    Ps_SendPotentiometerMeasurementNotification(handle);

    return gBleSuccess_c;
}
```

上面的函数中，我们首先拿到想要修改的那个特征值的 handle。Handle 为应用程序提供一个索引(index)以方便访问数据库中的每个 attribute。通过调用 GattDb_FindCharValueHandleInService 函数，并提供电位计相对值的 UUID，就可以返回想要的 handle。有了 handle 之后，就可以用 GattDb_WriteAttribute 函数将新的值写入 GATT 数据库，从而让其被客户端 Client 访问到。最后调用我们的第二个函数来发出通知。

```
static void Ps_SendPotentiometerMeasurementNotification
(
  uint16_t handle
)
{
    uint16_t  hCccd;
    bool_t isNotificationActive;
```

```
    /* Get handle of CCCD */
    if (GattDb_FindCccdHandleForCharValueHandle(handle, &hCccd) !=
gBleSuccess_c)
        return;

    if (gBleSuccess_c == Gap_CheckNotificationStatus
        (mPs_SubscribedClientId, hCccd, &isNotificationActive) &&
        TRUE == isNotificationActive)
    {
        GattServer_SendNotification(mPs_SubscribedClientId, handle);
    }
}
```

SendPotentiometerMeasurementNotification 函数向客户端发出一个通知（notification）。在函数中，首先取回我们在 GATT 数据库中定义的 CCCD 的 handle 值。然后，它检查客户端 client 是否已经在 CCCD 中设置了通知的请求。如果有，则发送通知给客户端，以便客户端执行对特征值的读取。

在 BLEADG 文档中的第 6 章和第 7 章中，有对所有访问 GATT 数据库的函数和使用 GATT 服务器函数的详细说明。第 8 章包含有关如何创建自定义 profile 的说明。BLEADG 是恩智浦 Connectivity 软件文档的一部分。

将新服务集成到现有 BLE 项目中

到目前为止，我们已经在数据库中创建了一个新服务，并且已经定义了处理它的函数。现在必须把它集成到一个项目中，以便通过恩智浦 Connectivity 协议栈进行管理。

恩智浦 Connectivity 软件项目的文件夹目录结构分为五个不同的模块。App 包括所有应用程序文件。Bluetooth 包含与 BLE 通信相关的文件。Framework 包含给协议栈提供的处理内存，低功耗等相关的辅助软件。KSDK 包含 Kinetis SDK 的底层驱动程序（ADC, GPIO...）。RTOS 包含与操作系统相关的文件。

图 3 文件夹目录结构

服务的相关文件必须被添加到项目中的 Bluetooth 目录下，具体在 profiles 子目录下。我们为 service.c 文件创建一个新文件夹，并且在 interface 子文件夹下添加了 interface.h 文件。

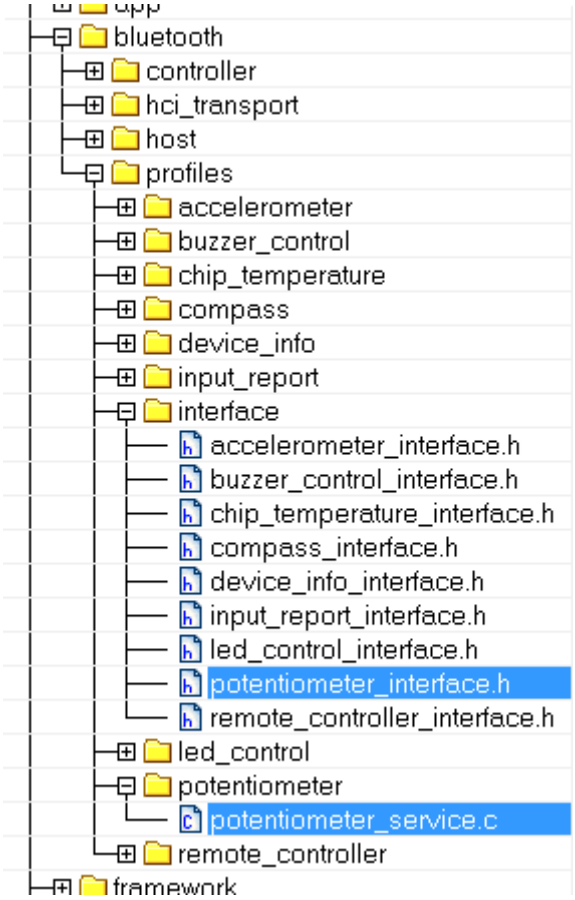

图 4 包含的 service 文件

当文件在项目中加好后，需要在协议栈中初始化服务。文件 app.c 是恩智浦 BLE 协议栈的主要应用程序文件。它调用所有 BLE 的初始化和应用程序回调。service_interface.h 文件必须要添加到此文件中。

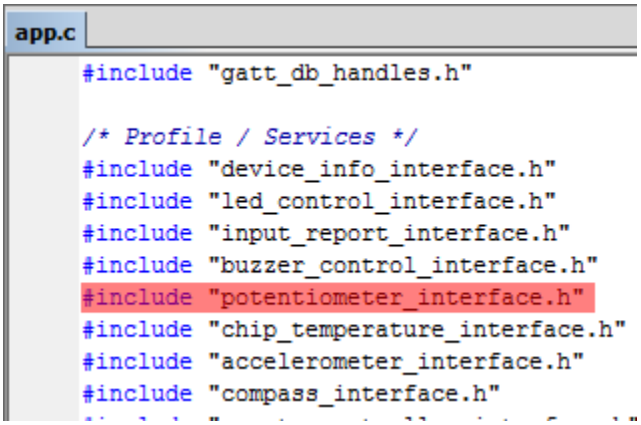

图 5 交互头文件内容

然后在局部变量的定义中，需要为新服务添加一个新的服务配置变量。这个变量的类型和在服务接口文件中定义的变量类型相同。这个变量的初始化需要用到服务名称（在 gattdb.h 中定义的 service name），所有特征值的初始值也要一并初始化。

```
/* Service Data*/
static disConfig_t disServiceConfig = {service_device_info};
static lcsConfig_t lcsServiceConfig = {service_led_control, 0};
static irsConfig_t irsServiceConfig = {service_input_report, 0};
static bcsConfig_t bcsServiceConfig = {service_buzzer_control, FALSE};
static psConfig_t psServiceConfig = {service_potentiometer, 0};
static ctsConfig_t ctsServiceConfig = {service_chip_temperature, 2500};
```

图 6 服务配置结构

现在必须初始化该服务。它在 BleApp_Config 函数内通过调用最近添加的服务的 Start 函数执行。

```
static void BleApp_Config()
{
    /* Read public address from controller */
    Gap_ReadPublicDeviceAddress();

    /* Register for callbacks*/
    App_RegisterGattServerCallback(BleApp_GattServerCallback);

    .
    .
    .

    mAdvState.advOn = FALSE;
    /* Start services */
    Lcs_Start(&lcsServiceConfig);
    Dis_Start(&disServiceConfig);
    Irs_Start(&irsServiceConfig);
    Bcs_Start(&bcsServiceConfig);
    Ps_Start(&psServiceConfig);
```

最后，必须将 subscribe 和 unsubscribe 函数添加到正确的 host 回调中。在 BleApp_ConnectionCallback 函数中，必须在 gConnEvtConnected_c（设备已连接）这个 case 分支下面调用 subscribe 函数，并且必须在 gConnEvtDisconnected_c（设备已断开连接）这个 case 分支下面调用 unsubscribe 函数。

```
static void BleApp_ConnectionCallback (deviceId_t peerDeviceId,
gapConnectionEvent_t* pConnectionEvent)
{
    switch (pConnectionEvent->eventType)
```

```
    {
        case gConnEvtConnected_c:
        {

            .
            .
            .

                /* Subscribe client*/
                mPeerDeviceId = peerDeviceId;
                Lcs_Subscribe(peerDeviceId);
                Irs_Subscribe(peerDeviceId);
                Bcs_Subscribe(peerDeviceId);
                Cts_Subscribe(peerDeviceId);
                Ps_Subscribe(peerDeviceId);
                Acs_Subscribe(peerDeviceId);
                Cps_Subscribe(peerDeviceId);
                Rcs_Subscribe(peerDeviceId);

            .
            .
            .

        case gConnEvtDisconnected_c:
        {
        /* UI */
           Led1Off();

           /* Unsubscribe client */
           mPeerDeviceId = gInvalidDeviceId_c;
           Lcs_Unsubscribe();
           Irs_Unsubscribe();
           Bcs_Unsubscribe();
           Cts_Unsubscribe();
           Ps_Unsubscribe();
           Acs_Unsubscribe();
           Cps_Unsubscribe();
           Rcs_Unsubscribe();
```

在此之后，客户端的应用程序就可以访问服务了。

处理通知和写请求

当初始化新服务后，客户端就可以访问 GATT 数据库的属性(attribute)，通过发出命令（读取，写入，通知...）等方式。然而，当客户端写入某个属性或将 CCCD 设置为开始通知时，服务器端的程序必须知道这些请求以在必要时处理它们。

处理通知

当特性已被配置为要发通知时，客户端希望每隔一段时间都会从其中接收到消息，具体取决于预先配置的参数。为了表明这一点，客户端为特征写入特定的 CCCD，指示通知必须开始/停止发送。

发生这种情况时，BleApp_GattServerCallback 将在主程序中被执行。应用程序要观察 CCCD 以及 gEvtCharacteristicCccdWritten_c 事件。发生事件表明 CCCD 已经被写入。需要用条件语句逐条判断是哪个 CCCD 被改了，并作出相应的动作。

```
static void BleApp_GattServerCallback (deviceId_t deviceId,
gattServerEvent_t* pServerEvent)
{
    switch (pServerEvent->eventType)
    {
      case gEvtCharacteristicCccdWritten_c:
        {
            /*
     Attribute CCCD write handler: Create a case for your registered attribute and
     execute callback action accordingly
     */
            switch(pServerEvent->eventData.charCccdWrittenEvent.handl
e)
            {
            case cccd_input_report:{
              //Determine if the timer must be started or stopped
              if
(pServerEvent->eventData.charCccdWrittenEvent.newCccd){
                // CCCD set, start timer
                TMR_StartTimer(tsiTimerId, gTmrIntervalTimer_c,
gTsiUpdateTime_c ,BleApp_TsiSensorTimer, NULL);
#if gAllowUartDebug
                Serial_Print(debugUartId, "Input Report notifications enabled
\n\r", gNoBlock_d);
```

```
#endif
                }
                else{
                    // CCCD cleared, stop timer
                    TMR_StopTimer(tsiTimerId);
#if gAllowUartDebug
                    Serial_Print(debugUartId, "Input Report notifications disabled
\n\r", gNoBlock_d);
#endif
                }
            }
                break;

            case cccd_potentiometer:{
                //Determine if the timer must be started or stopped
                if
(pServerEvent->eventData.charCccdWrittenEvent.newCccd){
                    // CCCD set, start timer
                    TMR_StartTimer(potTimerId, gTmrIntervalTimer_c,
gPotentiometerUpdateTime_c ,BleApp_PotentiometerTimer, NULL);
#if gAllowUartDebug
                    Serial_Print(debugUartId, "Potentiometer notifications enabled
\n\r", gNoBlock_d);
#endif
                }
                else{
                    // CCCD cleared, stop timer
                    TMR_StopTimer(potTimerId);
#if gAllowUartDebug
                    Serial_Print(debugUartId, "Potentiometer notifications disabled
\n\r", gNoBlock_d);
#endif
                }
            }
                break;
```

在这个例子中，当 gEvtCharacteristicCccdWritten_c 触发时，一个 switch-case 选择器用来确定被写入的 CCCD。通过读取 pServerEvent 结构体中的 eventData.charCccdWrittenEvent.handle 变量来完成。得到的 handle 必须将与 gatt_db.h 中定义的 CCCD 的名称进行比较，以获得每个可通知的特征。

```
PRIMARY_SERVICE_UUID128(service_potentiometer, uuid_service_potentiometer)
    CHARACTERISTIC_UUID128(char_potentiometer_relative_value, uuid_characteristic_potentiometer_relati
        VALUE_UUID128(value_potentiometer_relative_value, uuid_characteristic_potentiometer_relative_v
        CCCD(cccd_potentiometer)
        DESCRIPTOR(cpfd_potentiometer, gBleSig_CharPresFormatDescriptor_d, (gPermissionFlagReadable_c)
                0xAD/*Unit precentage UUID in Little Endian (Lower byte)*/,
                0x27/*Unit precentage UUID in Little Endian (Higher byte)*/,
                0x01, 0x00, 0x00)
```

图 7 CCCD 名称

一旦检测到正确的 CCCD， 程序必须确定它是否已设置或清除。这是通过读取 pServerEvent 结构中的 eventData.charCccdWrittenEvent.newCccd 并执行相应操作来完成的。在示例代码中，一个计时器被启动或停止。当计时器达到预设值，就会发送一个新的通知，通过先前在服务文件中定义的 Ps_RecordPotentiometerMeasurement 函数（请参阅为新服务制作驱动程序章节）。

处理写请求

写请求的回调不会像通知那样自动生成。它们必须在应用程序初始化期间被注册。需要注意的是，启用此功能后，写入的值不会自动存储在 GATT 数据库中。开发人员必须设计代码才能完成此操作，有时还要执行其他应用程序的相关操作。这个过程中，会用到 GattServer_RegisterHandlesForWriteNotifications 函数，以及所有要生成回调的特性的 handle。

```
* Configure writtable attributes that require a callback action */
    uint16_t notifiableHandleArray[] = {value_led_control,
value_buzzer, value_accelerometer_scale, value_controller_command,
value_controller_configuration};
    uint8_t notifiableHandleCount = sizeof(notifiableHandleArray)/2;
    bleResult_t initializationResult =
GattServer_RegisterHandlesForWriteNotifications(notifiableHandleCount,
(uint16_t*)&notifiableHandleArray[0]);
```

在这个例子中，我们创建了一个包含所有可写特征的数组。注册回调的函数需要这些参数：被注册的特征的 handle 的总数，以及一个指向拥有这些句柄的数组的指针。

当客户端连接后，每次写入一个已配置的特性时，BleApp_GattServerCallback 函数中的 gEvtAttributeWritten_c 代码段将会被执行。必须读取变量 pServerEvent-> eventData.attributeWrittenEvent.handle 以确定写入特征的句柄并相应地执行操作。

根据用户应用程序的设计， GATT 数据库中的值会被更新。为此，必须执行函数 GattDb_WriteAttribute。建议在 service.c 文件中创建一个更新数据库中属性的函数。

```
case gEvtAttributeWritten_c:
        {
```

```
            /*
            Attribute write handler: Create a case for your
registered attribute and
            execute callback action accordingly
            */
            switch(pServerEvent->eventData.attributeWrittenEvent.ha
ndle){
              case value_led_control:{
                bleResult_t result;

                //Get written value
                uint8_t* pAttWrittenValue =
pServerEvent->eventData.attributeWrittenEvent.aValue;

                //Create a new instance of the LED configurator
structure
                lcsConfig_t lcs_LedConfigurator = {
                  .serviceHandle = service_led_control,
                  .ledControl.ledNumber =
(uint8_t)*pAttWrittenValue,
                  .ledControl.ledCommand =
(uint8_t)*(pAttWrittenValue + sizeof(uint8_t)),
                };

                //Call LED update function
                result = Lcs_SetNewLedValue(&lcs_LedConfigurator);

                //Send response to client
                BleApp_SendAttWriteResponse(&deviceId, pServerEvent,
&result);

              }
              break;
```

执行完所有必需的操作后，服务器必须向客户端发送一个响应(response)。为此，服务器会调用函数 GattServer_SendAttributeWrittenStatus，参数包括句柄和给客户端反馈的错误代码（OK 或错误状态）。

```
static void BleApp_SendAttWriteResponse (deviceId_t* pDeviceId,
gattServerEvent_t* pGattServerEvent, bleResult_t* pResult){
  attErrorCode_t attErrorCode;

  // Determine response to send (OK or Error)
```

```
  if(*pResult == gBleSuccess_c)
    attErrorCode = gAttErrCodeNoError_c;
  else{
    attErrorCode = (attErrorCode_t)(*pResult & 0x00FF);
  }
  // Send response to client
  GattServer_SendAttributeWrittenStatus(*pDeviceId,
pGattServerEvent->eventData.attributeWrittenEvent.handle,
attErrorCode);
}
```

有关如何处理可写特征的更多信息，请参阅 BLEADG 第 5 章（包含在 NXP Connectivity 软件文档中）。

参考

Bluetooth? Low Energy Application Developer’s Guide (BLEADG)- 包含在恩智浦连接软件文档中

FRDM-KW40Z Demo Application (XLI 这个后面还有个链接要加上)