Tutorial

Custom Bluetooth Service

For The DA1468x SoC

Abstract

This tutorial should be used as a reference guide to gain a deeper understanding of the 'Custom Profile Concept'. As such, it covers a broad range of topics including an introduction to Bluetooth Service structure and the usage of Dialog's APIs related to Attribute Protocol. Furthermore, it covers a number of sections containing in depth software analysis of a complete demonstration example.

Contents

Figures

Terms and Definitions

API	Application Programming Interface
ATT	Attribute Protocol
BD	Bluetooth
CCC	Client Characteristic Configuration
CUD	Characteristic User Description
GAP	Generic Access Profile
GATT	Generic Attribute Profile
H/W	Hardware
IP	Intellectual Property
LE	Low Energy
ms	Millisecond
PDU	Protocol Data Unit
SDK	Software Development Kit
UUID	Universally Unique Identifier

References

[1] UM-B-044, DA1468x Software Platform Reference, User Manual, Dialog Semiconductor.
[2] Naresh Gupta, “Inside BLEUTOOTH LOW ENERGY”, ARTECH HOUSE, 2013.
[3] Robin Heydon, “Bluetooth Low Energy - The Developer’s Handbook”, PRENTICE HALL, 2013

1 Introduction

1.1 Before You Start

Before you start you need to:

- Install the latest SmartSnippets Studio
- Download the latest SDK for the DA1468x platforms

These can be downloaded from the Dialog Semiconductor support portal.

Additionally, for this tutorial either a Pro or Basic Development kit is required.

The key goals of this tutorial are to provide:

- A basic understanding of Generic ATT profile
- A basic understanding of Dialog Bluetooth framework architecture
- A basic understanding of Bluetooth database creation process
- A complete sample project demonstrating the creation of a custom Bluetooth service

1.2 Attribute Protocol (ATT)

The attribute protocol provides mechanisms for discovering attributes of a remote device, as well as reading and writing attributes. The attribute protocol follows a client-server model. The server exposes a set of attributes to the client. The client can discover, read, and write those attributes. The server can also send notifications or indications to the client about any of the attributes. A device can implement a client, a server, or both client and server roles. At any given time, only one server can be active on a device.

1.2.1 Client-Server Architecture

Servers have data, this is known as the **peripheral** in the Generic Access Protocol (GAP).

Clients request data to/from servers, this is known as **central** in GAP.

Servers expose data using Attributes.

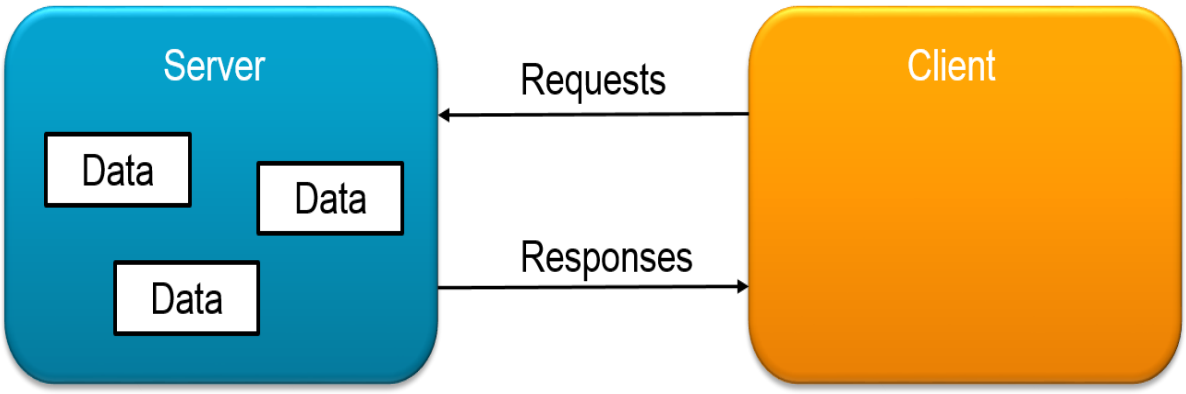

Figure 1: Client-Server Architecture

1.3 Attribute Definition

An attribute is something that represents data. It could be thought of as any data, at any given time, when the device is in any given state. ATT is designed to push or pull that data to or from a remote device. ATT also supports setting notifications and indications, so that the remote device can be alerted when the data changes. As well as containing the value of the data, an attribute has three properties associated with it:

1. Attribute Type
2. Attribute Handle
3. Access Permissions

1.3.1 Attribute Value

An attribute value is an octet array that contains the actual value of the attribute. The length of the attribute can be either fixed or variable:

- **Fixed length:** The length can be one, two or four octet.
- **Variable length:** The attribute can be a variable length string.

To simplify things, ATT does not allow multiple attribute values to be transmitted in a single PDU. A PDU contains only one attribute value and if the attribute value is too long to transmit in a single PDU, it can be split across multiple PDUs. There are some exceptions to this, for example, when a client requests multiple attributes to be read and the attributes have a fixed length, then the response can contain multiple attributes.

1.3.2 Attribute Type

The attribute type specifies what that particular attribute represents. This allows the client to understand the meaning of the attribute. The attribute type is identified by a Universally Unique Identifier (UUID). A UUID is a 128-bit value which is considered to be unique over space and time. The implementation could either use the set of predefined UUIDs or define its own UUIDs. In general, a shorter form of UUIDs is used. This shorter form is 16-bit. The 16-bit UUIDs are assigned by the Bluetooth SIG and published on the Bluetooth Assigned Numbers page on the Bluetooth SIG website.

1.3.3 Attribute Handle

All attributes on the server are assigned a unique non-zero attribute handle. This handle is used by the client in all operations with the server to identify the attribute. It is allowed to dynamically add or remove attributes on the server as long as the new attributes are not assigned a handle which has already been used by any other attribute in the past (even if that attribute has been deleted). This ensures that clients always get a unique attribute handle. Once an attribute has been assigned an attribute handle, it should not change over time. This ensures that clients can keep accessing that attribute with the same handle. The attributes on the server are ordered by the attribute handle. The attribute handle of 0x0000 is reserved and the attribute handle of 0xFFFF is known as the maximum attribute handle.

1.3.4 Attribute Permissions

Each attribute has an associated set of permissions which determines the level of access permitted for that particular attribute. The attribute permissions are used by a server to determine whether a client is allowed to read or write an attribute value, and whether Authentication or Authorization is required to access that particular attribute. Attribute permissions are a combination of the following three permissions:

1. **Access Permission:** This can be:
   - Readable
   - Writeable
   - Readable and Writeable
2. **Authentication Permission:** This is used by the server to determine if an authenticated physical link is required when a client attempts to access that attribute or when the server has to send a notification or indication to client. This can be set to either:
   - Authentication Required
   - No Authentication Required
3. **Authorization Permission:** This is used by the server to determine if client needs to be authorized before accessing an attribute value. This could be set to either:
   - Authorization Required
   - No Authorization Required

If client does not have sufficient permissions an error is returned.

1.4 BLE Service Structure

This section describes the internal structure of an ATT Profile. It contains a brief description of the attributes that make up a profile and then it explains the Dialog BLE framework architecture.

1.4.1 Service Declaration

A service is grouped using a **Service Declaration**. This is an attribute with an attribute type of either **Primary Service** or **Secondary Service**. All attributes that follow this Service Declaration and occur before the next Service Declaration are considered grouped with this service; they belong to this service. A Primary Service is one that encapsulates what the device does. A Secondary Service is one that helps the Primary Service to achieve its behavior. All Secondary Services are referenced from a Primary Service. The service declaration's value is a **Service UUID**. This is either a 16-bit Bluetooth UUID or a 128-bit custom UUID. Any service that a device does not understand can be safely ignored. To help with this, the Attribute Protocol allows the range of attribute handles of services to be discovered and only known services will be processed further.

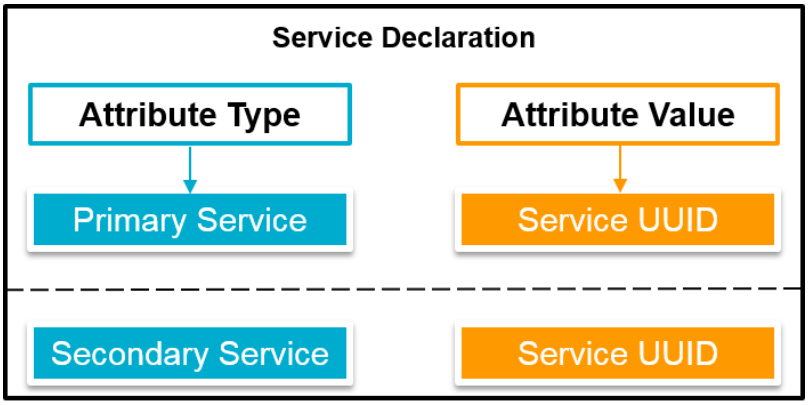

Figure 2: Primary and Secondary Service Declaration

1.4.2 Include Declaration

Secondary Services must be discovered separately. To do this, each service can have zero or more **Include** attributes. Include declarations always immediately follow the Service Declaration and come before any other attributes of the service. The Include definitions also encompass the handle range of the referenced service, along with the Service UUID of the included service. This allows very quick discovery of the referenced services, their grouped attributes and the type of the service. It does not state if this referenced service is a primary or a secondary service because this is not relevant.

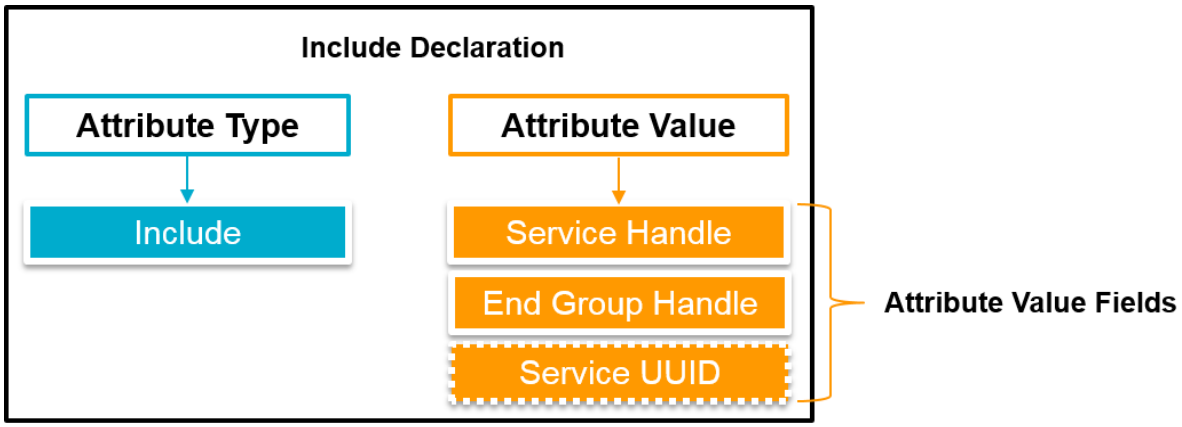

Figure 3: Including Service Declaration

Note*:* Given that four octets are used for handles in the Attribute Value fields, a full 128-bit Service UUID will not fit into the standard response packets used to find the included services. Therefore, when the included service has a 128-bit UUID, the Service UUID is not part of the value declaration. This means that an additional Attribute Protocol read request is required to find the type of the service included.

1.4.3 Characteristics

Grouping attributes together within a service, demonstrates how these attributes can be combined to provide a consistent interface to a block of behavior. The Bluetooth Low Energy architecture also makes it possible to group attributes to allow the state and behavior of a service to be exposed. More specifically, a characteristic exposes the type of data a value represents, whether a value can be read or written, how to configure the value to be indicated, notified, or broadcast, and what a value means. To do this, a characteristic is composed of three basic elements:

- Declaration
- Value
- Descriptor(s)

A **Declaration** is the start of a characteristic; it groups all the other attributes for this characteristic. The **Value** attribute contains the actual value for this characteristic. The **Descriptors** hold additional information or configuration for this characteristic.

1.4.3.1 Characteristic Declaration

To start a characteristic, a **Characteristic** attribute is used. This contains three fields, as shown in Figure 4:

- Properties
- Value handle
- Characteristic UUID

More specifically, the characteristic **Properties** determines if the characteristic Value attribute can be read, written, notified, indicated, broadcast, or authenticated in a signed write. The characteristic **Value Handle** field is the handle of the attribute that contains the value for the characteristic. The final field is the **Characteristic UUID** which holds the UUID that is used to identify the type of the characteristic value.

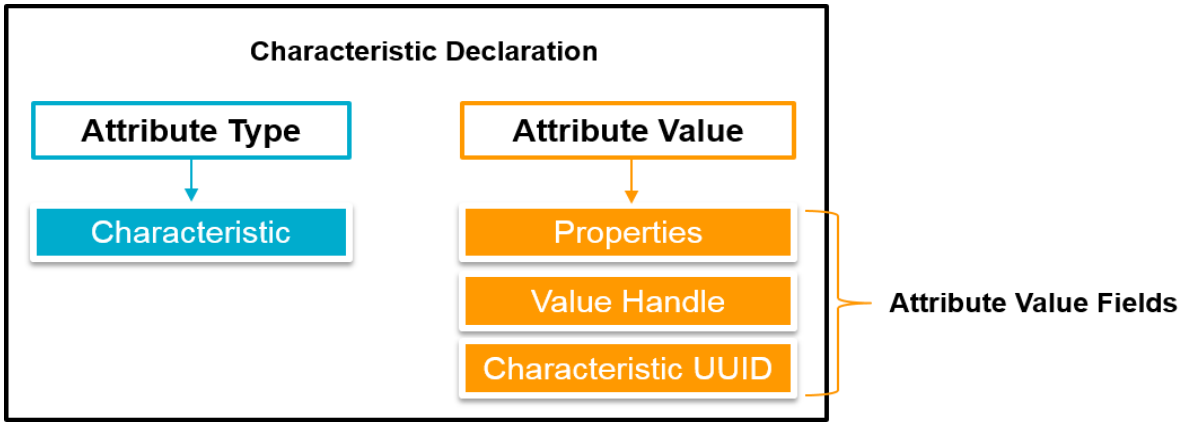

Figure 4: Characteristic Declaration

Note*:* **The Value Handle field allows a very quick search for the characteristic to be performed by a client. It returns only Characteristic Declarations. With this declaration, the attribute that holds the value is immediately available. If this field did not exist, the client would need to perform an additional search for attributes and effectively guess which attribute after the declaration was the value.**

1.4.3.2 Characteristic Value

The **Characteristic Value** is an attribute with a type that must match the characteristic declarations' **Characteristic UUID** field. Apart from that, it is an ordinary attribute. The biggest difference is that the types of actions that can be performed on this characteristic value attribute, are exposed in the characteristic declarations' **Properties** field and additionally might be in the **Characteristic Extended Properties** descriptor.

1.4.3.3 Characteristic Descriptors

There can be any number of descriptors on a characteristic. Most descriptors are optional, although they might be required depending on the Characteristic Declaration. Some descriptors might also be required by a service specification. The following descriptors can be included in a characteristic:

Characteristic Extended Properties

This is used to capture the additional extended properties, for example the ability to perform reliable writes on the value or to write the Characteristic User Description descriptor.

Characteristic User Description

Using this descriptor, a device can associate a text string with a characteristic.

Client Characteristic Configuration

If a characteristic is notifiable or indicatable, this descriptor must exist. It is a two-bit value, with one bit for notifications and the other for indications. Notifications and Indications are complementary procedures, so it's impossible to set both of these bits at the same time. How the value is notified or indicated is not defined in the core specifications; this is defined by the service specifications.

Server Characteristic Configuration

This is very similar to the Client Characteristic Configuration descriptor, except that is has only one bit which is used for broadcast. Setting this bit causes the device to broadcast data associated with the service in which this characteristic is grouped. The timing of this broadcast is determined by the service.

Characteristic Presentation Format

One of the goals for the Generic Attribute Profile is to enable generic clients. A generic client is defined as a device that can read the values of a characteristic and display them to the user without understanding what they mean. The Characteristic Presentation Format denotes if a characteristic can be used by a generic client.

Characteristic Aggregation Format

Some characteristic values are more complex than just a single value. To allow for such complex characteristic values, the Characteristic Aggregation Format descriptor allows multiple Characteristic Presentation Format descriptors to be referenced, so that individual fields of the value can be illustrated.

1.5 BLE Framework Architecture

The Dialog's BLE framework consists of the following building blocks:

- **BLE Service Framework** - provides implemented 'out-of-the-box' BLE services
- **Dialog BLE API** - a set of functions to initiate BLE operations or respond to BLE events
- **BLE Manager** - provides the interface to the BLE functionality of the chip
- **BLE Adapter** - provides the interface to the BLE stack and executes the BLE stack internal scheduler, BLE interrupts etc.
- **BLE Stack** - together with the BLE H/W IP, this implements all of the additional BLE stack layers up to GAP and GATT

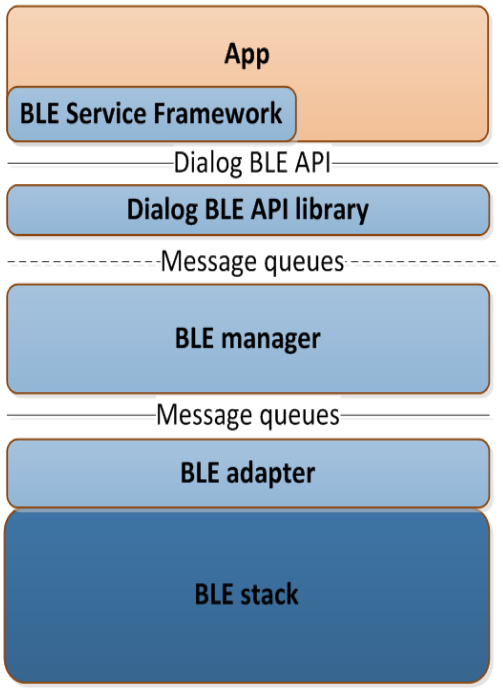

Figure 5: BLE Framework Architecture

A typical flow chart of a command execution is as follows:

1. In principle, an application should only have to interface either with the Dialog BLE API library or/and the BLE Service Framework.
2. Commands are sent to the BLE Manager over the command queue and Application tasks wait for the response on the response queue.
3. Once the command is received, the response message is sent on the response queue.
4. API call completes and application execution continues.
5. BLE events are received asynchronously from the BLE event queue.

Note*:* Some API calls don't send command messages but directly access BLE manager's device parameters structure (acquire, modify, release).

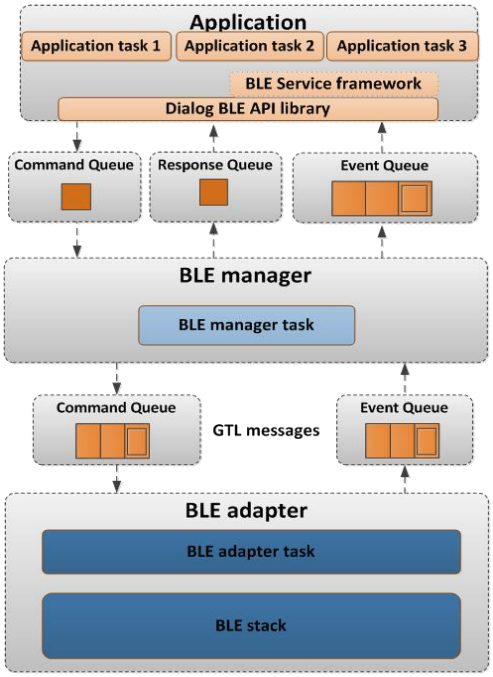

Figure 6: BLE Framework Flow Chart

1.6 Event Handling Levels

Several events can occur during the lifetime of the BLE application and these events need to be handled in a specific manner. In all Dialog's sample projects (found in the SDK) there are three levels of event handling:

1. Check whether Bluetooth events can be handled by well-defined Bluetooth Services.
2. If not, then the main application task should handle them.
3. If the application does not exhibit handlers for handling specific Bluetooth events, then ble_handle_event_default() should be invoked.

```
1st level
if (ble_service_handle_event(hdr)) {
        goto handled;
}

2nd level
switch (hdr->evt_code) {
case BLE_EVT_GAP_CONNECTED:
        handle_evt_gap_connected((ble_evt_gap_connected_t *) hdr);
        break;
case BLE_EVT_GAP_ADV_COMPLETED:
        handle_evt_gap_adv_completed((ble_evt_gap_adv_completed_t *) hdr);
        break;
case BLE_EVT_GAP_PAIR_REQ:
{
        ble_evt_gap_pair_req_t *evt = (ble_evt_gap_pair_req_t *) hdr;
        ble_gap_pair_reply(evt->conn_idx, true, evt->bond);
        break;
}
3rd level
default:
        ble_handle_event_default(hdr);
        break;
}
```

Figure 7: Levels of Event Handling

1.7 BLE Service Framework

Provides:

- The API to create new services.
- A pool of services to be used "out-of-the-box" in an end application.

Header files are in:

- sdk/ble_services/include

Usage:

- Call simple initialization functions
- Define callbacks for the various BLE service events

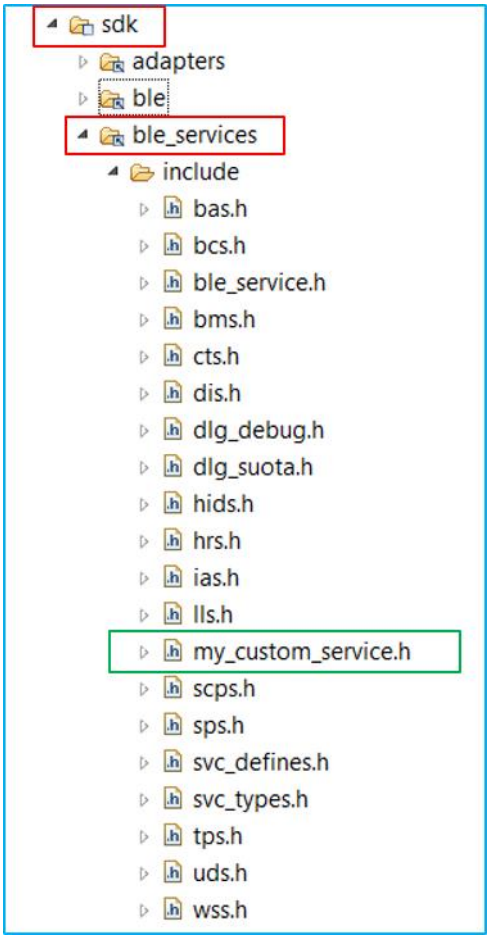

Figure 8: Out-of-the-Box Bluetooth Services in the Service Framework

2 Custom Database Creation Process

This section analyzes an application example which demonstrates creating a custom Bluetooth service. The example is based on the **ble_peripheral** sample code found in the SDK. It adds an additional custom 128-bit BLE service in the internal BLE Framework.

2.1 Working on a Custom BLE Service

This section goes through the steps required to create a custom 128-bit ATT Service. For demonstration purposes let's create a custom service which exhibits the following features:

- One Service Declaration of type Primary
- One Characteristic Declaration with write-read-notify access permissions
- Two Descriptors, one of type Client Characteristic Configuration and one of type Characteristic User Description

In general, it's good practice to separate the source code of Bluetooth service implementation from the application itself. With this approach, a Bluetooth service is portable to any application just by defining a few callback functions as well as invoking an initialization function in application context.

1. Declare callback functions. These callbacks will be called following specific BLE events to exchange data between the application task and the Bluetooth service itself. The number of callback functions depends on the number of characteristics as well as their associated permissions. For demonstration purposes let's define two callbacks: one triggered upon read requests and one triggered upon write requests. Please note that these callbacks will be triggered from application context.

```
/* Callback functions */
typedef struct {

    /* Handler for read requests – Triggered on application context */

    /* Handler for write requests – Triggered on application context */

} my_custom_service_cb_t;
```

2. Define a structure associated with the Bluetooth service. This structure should contain a variable of type ble_service_t, the previously defined callback functions, as well as all the attribute handles of Bluetooth service.

```
/* Service related variables */
typedef struct {
    ble_service_t svc;

    // Callback functions


    // Attribute handles of Bluetooth Service

} mc_service_t;
```

3. Define the total number of attribute handles of Bluetooth service:

```
/*
 * 0 --> Number of Included Services
 * 1 --> Number of Characteristic Declarations
 * 2 --> Number of Descriptors
 */
num_attr = ble_gatts_get_num_attr(0, 1, 2);
```

4. Declare all the attributes of type **Service** (either Primary or/and Secondary). Please note that all the custom Bluetooth services should exhibit a 128-bit UUID, while the Bluetooth SIG ones a 16-bit UUID.

```
/* Service declaration */
ble_uuid_from_string("00000000-1111-2222-2222-333333333333", &uuid);
ble_gatts_add_service(&uuid, GATT_SERVICE_PRIMARY, num_attr);
```

Note: Special care must be taken when defining a 128-bit UUID. If the UUID is not defined correctly, an assertion will be issued.

5. Declare all the attributes of type **Characteristic**:

```
/* Characteristic declaration */
ble_uuid_from_string("11111111-0000-0000-0000-111111111111", &uuid);
ble_gatts_add_characteristic(&uuid,
        GATT_PROP_READ | GATT_PROP_NOTIFY | GATT_PROP_WRITE,
         ATT_PERM_RW, 1, GATTS_FLAG_CHAR_READ_REQ, NULL,
                                        &mcs->mc_char_value_h);
```

Note: The GATTS_FLAG_CHAR_READ_REQ flag, is mandatory when enabling read permissions.

6. Declare all the attributes of type **Characteristic Descriptors**.

```
/* Descriptor declaration - Client Characteristic Configuration (CCC) */
ble_uuid_create16(UUID_GATT_CLIENT_CHAR_CONFIGURATION, &uuid);
ble_gatts_add_descriptor(&uuid, ATT_PERM_RW, 2, 0,
                                &mcs->mc_char_value_ccc_h);

/* Descriptor declaration - Characteristic User Description (CUD) */
ble_uuid_create16(UUID_GATT_CHAR_USER_DESCRIPTION, &uuid);
ble_gatts_add_descriptor(&uuid, ATT_PERM_READ,
        sizeof(char_user_descriptor_val), 0, &char_user_descriptor_h);
```

Note: For all the available GATT descriptor UUIDs see at sdk/ble/include/ble_uuid.h.

7. Register the newly created service in the ATT database. The first input parameter should be the first attribute handle of the service, whereas the last input parameter should be **zero**. In this

step, all the attribute handles should be registered, so that the BLE manager can update them automatically upon Bluetooth events.

```
/*
 * Register all the attribute handles so that they can be updated
 * by the BLE manager automatically.
 */
ble_gatts_register_service(&mcs->svc.start_h, &mcs->mc_char_value_h,
                           &mcs->mc_char_value_ccc_h, &char_user_descriptor_h , 0);
```

8. Calculate the last attribute handle of the service.

```
/* Calculate the last attribute handle of the BLE service */
mcs->svc.end_h = mcs->svc.start_h + num_attr;
```

9. When necessary, declare predefined attribute values.

```
/* Set default attribute values */
ble_gatts_set_value(mcs->mc_char_value_h, 1, variable_value);
ble_gatts_set_value(char_user_descriptor_h,  sizeof(char_user_descriptor_val),
                                                 char_user_descriptor_val);
```

10. Register the service in Dialog Bluetooth framework.

```
/* Register the BLE service in BLE framework */
ble_service_add(&mcs->svc);
```

11. Define handlers for specific Bluetooth events.

```
/* Declare handlers for specific BLE events */
mcs->svc.read_req  = handle_read_req;
mcs->svc.write_req = handle_write_req;
mcs->svc.cleanup  = cleanup;
```

12. Upon a write request, that is BLE_EVT_GATTS_WRITE_REQ, identify the attribute handle. In In case of invalid attribute handles, the appropriate error code should be sent back to the peer device.

```
/* Handler for write requests, that is BLE_EVT_GATTS_WRITE_REQ */
static void handle_write_req(ble_service_t *svc,
                             const ble_evt_gatts_write_req_t *evt)
{
    mc_service_t *mcs = (mc_service_t *) svc;
    att_error_t status = ATT_ERROR_WRITE_NOT_PERMITTED;
```

```
        /*
         * Identify for which attribute handle the write request has been sent to
         * and call the appropriate function.
         */
}
```

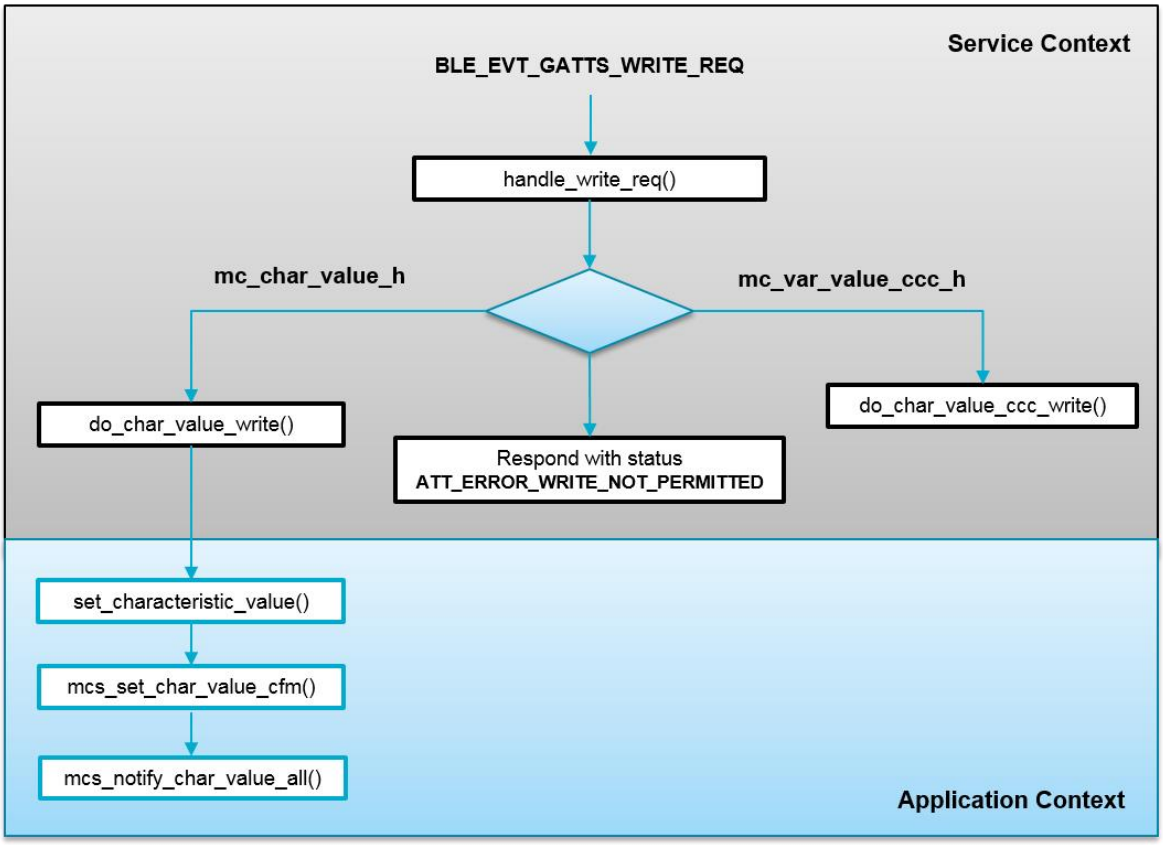

Figure 9: Write Request Event Flow Chart

13. Upon a read request, that is BLE_EVT_GATTS_READ_REQ, identify the attribute handle. In case of invalid attribute handles, the appropriate error code should be sent back.

```
/* Handler for read requests, that is BLE_EVT_GATTS_READ_REQ */
static void handle_read_req(ble_service_t *svc, const ble_evt_gatts_read_req_t *evt)
{
        mc_service_t *mcs = (mc_service_t *) svc;

        /*
         * Identify for which attribute the read request has been sent to
         * and call the appropriate function.
         */

}
```

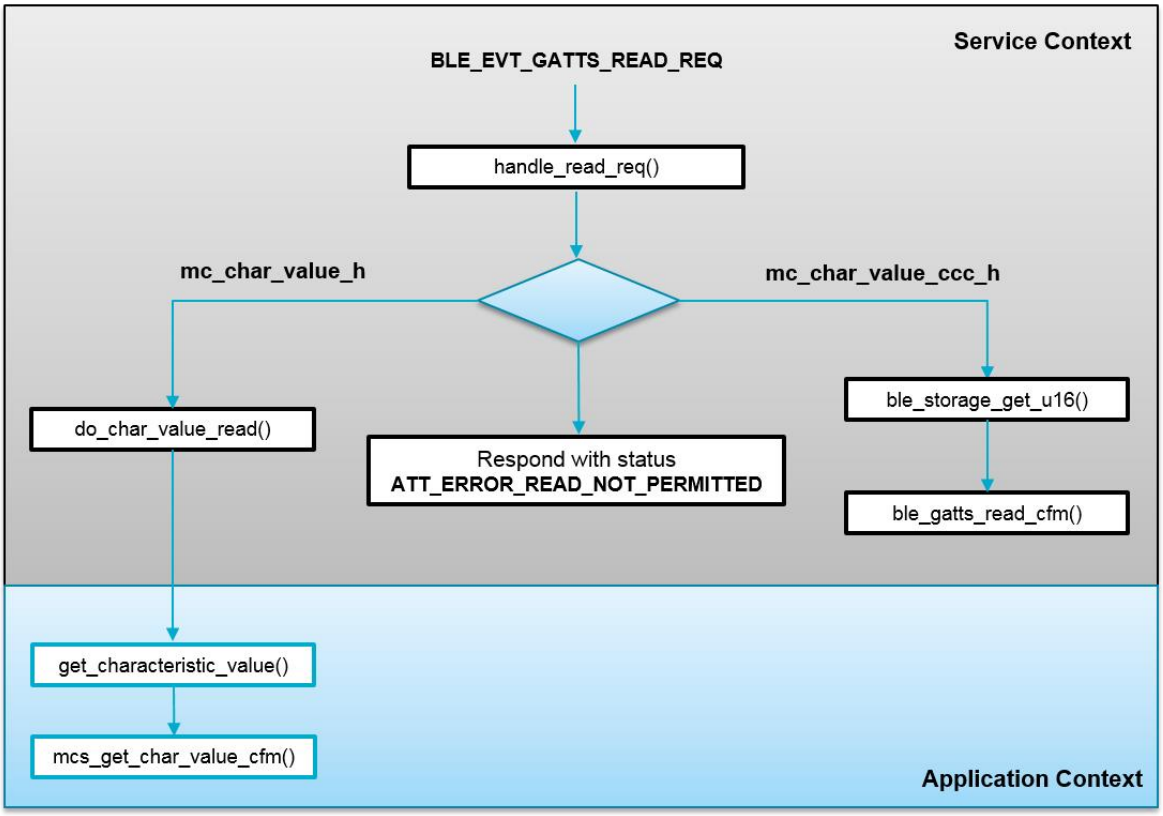

Figure 10: Read Request Event Flow Chart

14. In case of notifications/indications, how an attribute value is notified or indicated, is not defined in the core Bluetooth specifications. Thus, a custom process should be declared in order to signal all the connected peer devices about the updated characteristic value (given that notifications for that specific attribute are enabled).

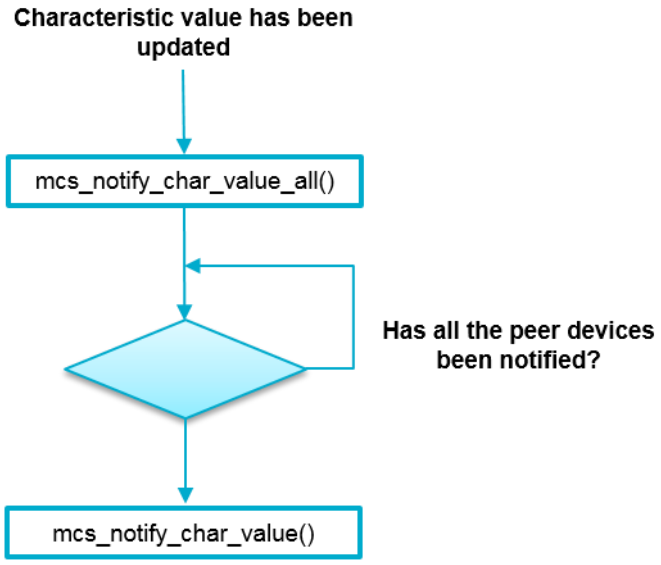

Figure 11: Characteristic Value Notification Flow Chart

3 Running The Demonstration Example

This section describes the steps required to prepare the Pro DevKit and other tools to successfully run the example code. A Pro DevKit as well as a smartphone are required for testing and verifying the code.

3.1 Verifying with a Scanner App

1. Establish a connection between the target device and your PC through the **USB2(DBG)** port of the motherboard. This port is used both for powering and communicating to the DA1468x SoC. For this tutorial a Pro DevKit is used.

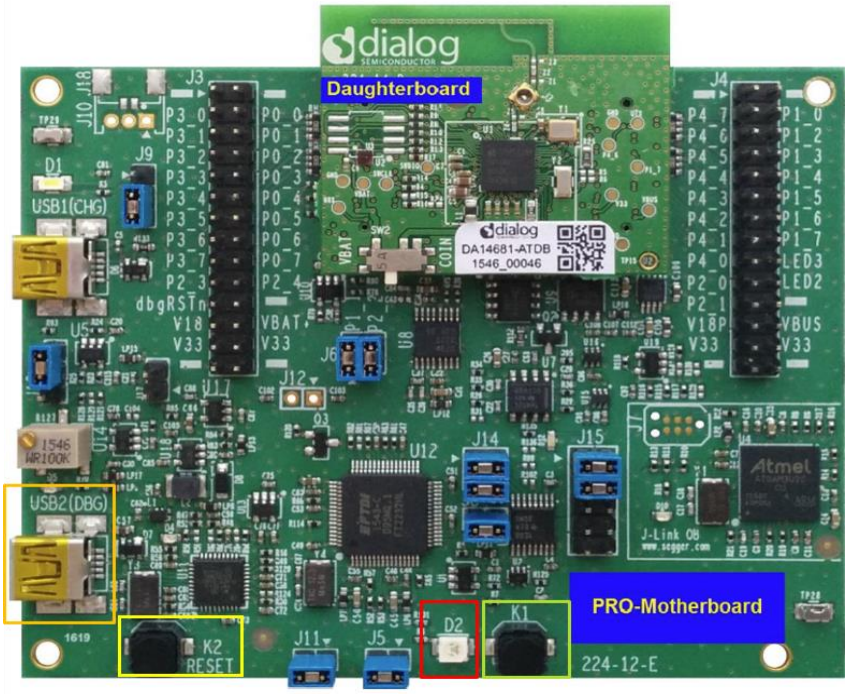

Figure 12: DA1468x Pro DevKit

2. Import and then make a copy of the **ble_peripheral** sample code found in the SDK of the DA1468x family of devices.

Note: It is essential to import the folder named scripts to perform various operations (including building, debugging, and downloading)

3. In the target application, add/modify all the required code blocks as illustrated in the Code Overview section.

Note: It is possible for the defined macros not to be taken into consideration instantly. Thus, resulting in errors during compile time. If this is the case, the easiest way to deal with the

issue is to: right-click on the application folder, select Index > Rebuild and then Index > Freshen All Files.

4. Build the project in either **Debug_QSPI** or **Release_QSPI** mode and burn the generated image to the chip (either via the serial or jtag port).
5. Press the **K2** button on Pro DevKit to start the chip executing its firmware.
6. When the project starts running, the DA1468x module should be visible by any Bluetooth scanner application. For this demonstration the **BLE Scanner** application was used.

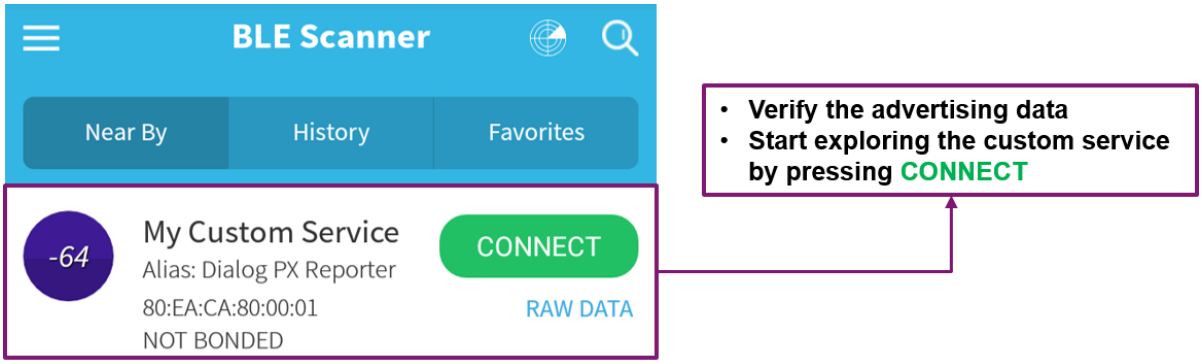

Figure 13: Verifying the Bluetooth Low Energy Device Output Using a Scanner App

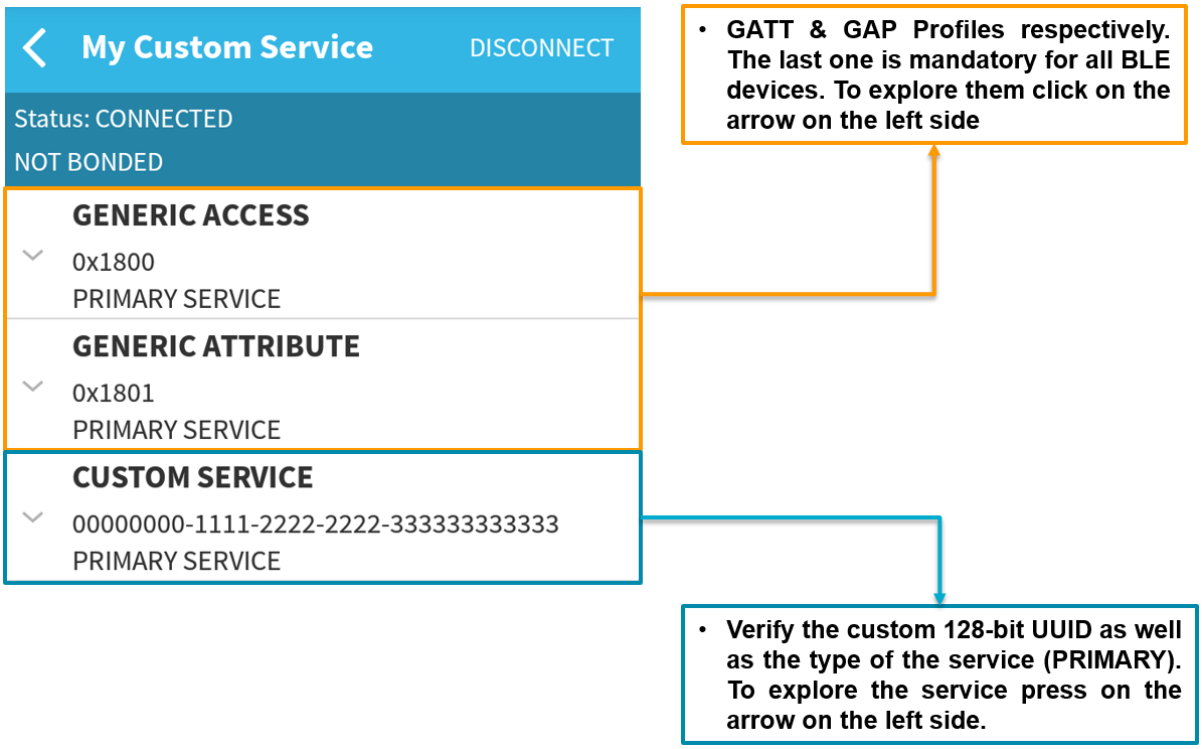

Figure 14: Exploring the Services after Connecting to a Remote Device

7. Verify the custom 128-bit UUID as well as the properties of the characteristic. Depending on the assigned properties, the scanner App will draw the corresponding symbols, for example **R** for reading, **W** for writing and **N** for enabling notifications.

   The descriptor with UUID equal to 0x2902 indicates whether notifications/indications are enabled or not. By default, notifications should be disabled. The descriptor with UUID equal to 0x2901 describes the role of the characteristic.

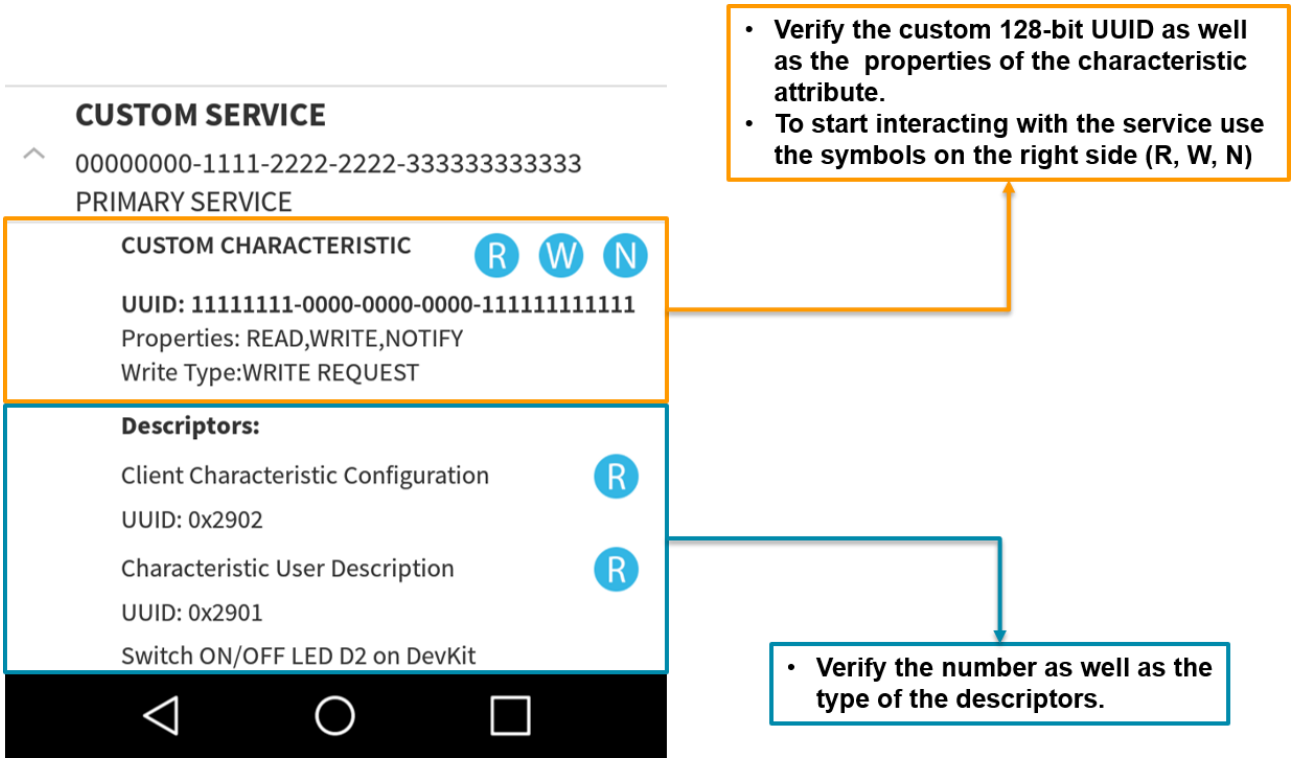

Figure 15: Verify the Service Characteristic

8. Verify the behavior of the characteristic This characteristic consists of 1-byte value. User can read as well as modify that value. If the attribute value is equal to 0x01, then the LED D2 on the Pro DevKit is turned on. For all other cases, the LED D2 is turned off.

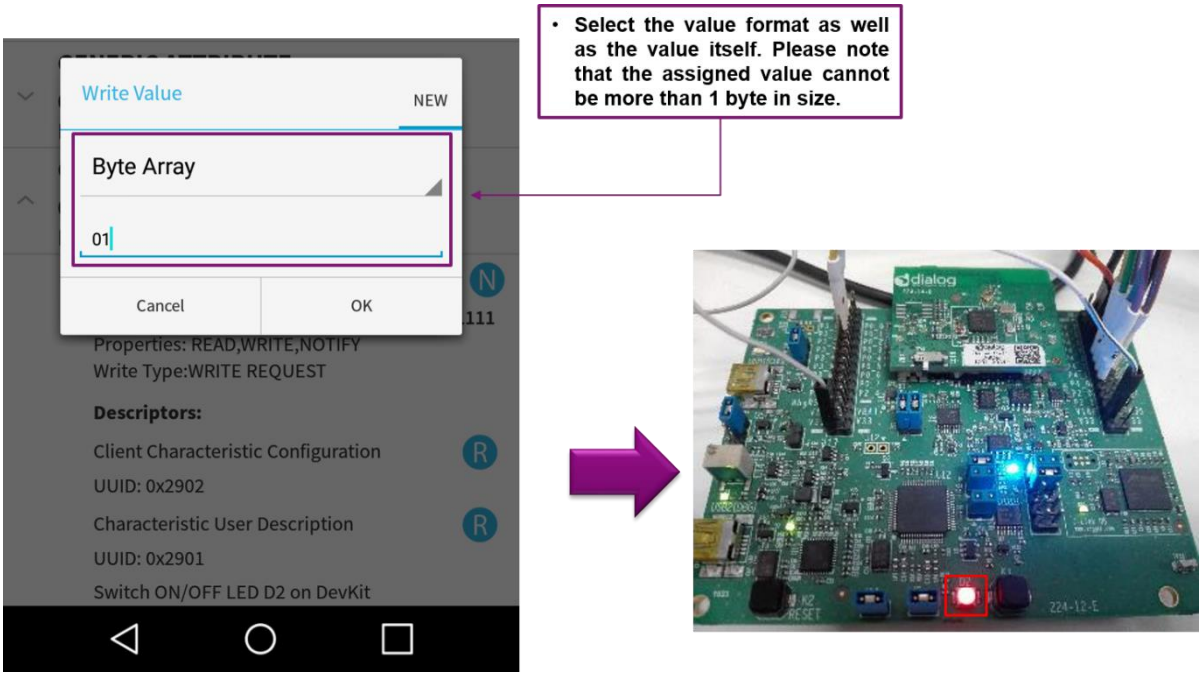

Figure 16: Verify the Characteristic Behavior (Write Operation)

4 Code Overview

This section provides the code blocks needed to successfully execute this tutorial.

4.1 Custom BLE Service Header File

Create a new header file, for instance my_custom_service.h, and add the following code:

```
#include <stdint.h>
#include <ble_service.h>

/* User-defined callback functions - Prototyping */
typedef void (* mcs_get_char_value_cb_t) (ble_service_t *svc, uint16_t conn_idx);

typedef void (* mcs_set_char_value_cb_t) (ble_service_t *svc,
                                          uint16_t conn_idx, const uint8_t *value);



/* User-defined callback functions */
typedef struct {

    /* Handler for read requests – Triggered on application context */
    mcs_get_char_value_cb_t get_characteristic_value;

    /* Handler for write requests – Triggered on application context */
    mcs_set_char_value_cb_t set_characteristic_value;

} my_custom_service_cb_t;
```

```
/*
 * \brief This function creates the custom BLE service and registers it in BLE framework
 *
 * \param [in] variable_value Default characteristic  value
 * \param [in] cb                   Application callback functions
 *
 * \return service handle
 *
 */
ble_service_t *mcs_init(const uint8_t *variable_value,
                        const my_custom_service_cb_t *cb);



/*
 * This function should be called by the application as a response to a read request
 *
 * \param[in] svc            service instance
 * \param[in] conn_idx    connection index
 * \param[in] status       ATT error
 * \param[in] value        attribute value
 */
void mcs_get_char_value_cfm(ble_service_t *svc, uint16_t conn_idx,
                                    att_error_t status, const uint8_t *value);


/*
 * This function should be called by the application as a response to a write request
 *
 * \param[in] svc           service instance
 * \param[in] conn_idx   connection index
 * \param[in] status       ATT error
 */
void mcs_set_char_value_cfm(ble_service_t *svc, uint16_t conn_idx,
                                                        att_error_t status);


/*
 * Notify all the connected peer devices that characteristic value has been
 * updated.
 *
 * \param[in] svc        service instance
 * \param[in] value     updated characteristic value
 */
void mcs_notify_char_value_all(ble_service_t *svc, const uint8_t *value);
```

4.2 Custom BLE Service Source File

Code snippet of custom BLE service implementation. Create a new source file, for instance my_custom_service.c, and add the following code:

```
#include <stdbool.h>
#include <stddef.h>
#include <string.h>
#include "osal.h"
#include "ble_att.h"
#include "ble_bufops.h"
#include "ble_common.h"
#include "ble_gatt.h"
#include "ble_gatts.h"
#include "ble_storage.h"
#include "ble_uuid.h"
#include "my_custom_service.h"


#define UUID_GATT_CLIENT_CHAR_CONFIGURATION (0x2902)

static const char char_user_descriptor_val[] = "Switch ON/OFF LED D2 on DevKit";


void mcs_notify_char_value(ble_service_t *svc, uint16_t conn_idx,
                                                const uint8_t *value);

/* Service related variables */
typedef struct {
      ble_service_t svc;

      // User-defined callback functions
      const my_custom_service_cb_t *cb;

      // Attribute handles of BLE service
      uint16_t mc_char_value_h;
      uint16_t mc_char_value_ccc_h;

} mc_service_t;


/* This function is called upon write requests to characteristic attribute value */
static att_error_t do_char_value_write(mc_service_t *mcs,
    uint16_t conn_idx, uint16_t offset, uint16_t length, const uint8_t *value)
{
      uint8_t ct;

      if (offset) {
            return ATT_ERROR_ATTRIBUTE_NOT_LONG;
      }


      /* Check if the length of the envoy data exceed the maximum permitted */
      if (length != 1) {
            return ATT_ERROR_INVALID_VALUE_LENGTH;
      }

      /*
        * Check whether the application has defined a callback function
```

```
         * for handling the event.
         */
        if (!mcs->cb || !mcs->cb->set_characteristic_value) {
                return ATT_ERROR_WRITE_NOT_PERMITTED;
        }

        ct = get_u8(value);

        /*
         * The application should get the data sent by the peer device.
         */
        mcs->cb->set_characteristic_value(&mcs->svc, conn_idx, &ct);

        return ATT_ERROR_OK;

}


/* This function is called upon write requests to CCC attribute value */
static att_error_t do_char_value_ccc_write(mc_service_t *mcs,
    uint16_t  conn_idx, uint16_t offset, uint16_t length, const uint8_t *value)
{
        uint16_t ccc;

        if (offset) {
                return ATT_ERROR_ATTRIBUTE_NOT_LONG;
        }

        if (length != sizeof(ccc)) {
                return ATT_ERROR_INVALID_VALUE_LENGTH;
        }

        ccc = get_u16(value);

        /* Store the envoy CCC value to the ble storage */
        ble_storage_put_u32(conn_idx, mcs->mc_char_value_ccc_h, ccc, true);

        return ATT_ERROR_OK;
}


/* This function is called upon read requests to characteristic attribute value */
static void do_char_value_read(mc_service_t *mcs,
                        const ble_evt_gatts_read_req_t *evt)
{
        /*
         * Check whether the application has defined a callback function
         * for handling the event.
         */
        if (!mcs->cb || !mcs->cb->get_characteristic_value) {
                ble_gatts_read_cfm(evt->conn_idx, evt->handle,
                        ATT_ERROR_READ_NOT_PERMITTED, 0, NULL);
                return;
        }
```

```
        /*
         * The application should provide the requested data to the peer device.
         */
        mcs->cb->get_characteristic_value(&mcs->svc, evt->conn_idx);

        // callback executed properly

}



/*
 * Notify all the connected peer devices that characteristic attribute
 * value has been updated.
 */
void mcs_notify_char_value_all(ble_service_t *svc, const uint8_t *value)
{
        uint8_t num_conn;
        uint16_t *conn_idx;

        ble_gap_get_connected(&num_conn, &conn_idx);

        while (num_conn--) {
                mcs_notify_char_value(svc, conn_idx[num_conn], value);
        }

        if (conn_idx) {
                OS_FREE(conn_idx);
        }
}



/* Notify the peer device that characteristic attribute value has been updated */
void mcs_notify_char_value(ble_service_t *svc, uint16_t conn_idx,
                                                        const uint8_t *value)
{
        mc_service_t *mcs = (mc_service_t *) svc;

        uint16_t ccc = 0x0000;
        uint8_t pdu[1];

        ble_storage_get_u16(conn_idx, mcs->mc_char_value_ccc_h, &ccc);


        /*
         * Check if the notifications are enabled from the peer device,
         * otherwise don't send anything.
         */
        if (!(ccc & GATT_CCC_NOTIFICATIONS)) {
                return;
        }

        pdu[0] = *((uint8_t *)value);
```

```
        ble_gatts_send_event(conn_idx, mcs->mc_char_value_h,
                             GATT_EVENT_NOTIFICATION, sizeof(pdu), pdu);
}


/*
 * This function should be called by the application as a response to read requests
 */
void mcs_get_char_value_cfm(ble_service_t *svc, uint16_t conn_idx,
                            att_error_t status, const uint8_t *value)
{
        mc_service_t *mcs = (mc_service_t *) svc;
        uint8_t pdu[1];

        pdu[0] = *value;

        /* This function should be used as a response for every read request */
        ble_gatts_read_cfm(conn_idx, mcs->mc_char_value_h, ATT_ERROR_OK,
                           sizeof(pdu), pdu);
}


/*
 * This function should be called by the application as a response to write requests
 */
void mcs_set_char_value_cfm(ble_service_t *svc, uint16_t conn_idx,
                                                        att_error_t status)
{
        mc_service_t *mcs = (mc_service_t *) svc;

        /* This function should be used as a response for every write request */
        ble_gatts_write_cfm(conn_idx, mcs->mc_char_value_h, status);
}


/* Handler for read requests, that is BLE_EVT_GATTS_READ_REQ */
static void handle_read_req(ble_service_t *svc,
                            const ble_evt_gatts_read_req_t *evt)
{
        mc_service_t *mcs = (mc_service_t *) svc;

        /*
         * Identify for which attribute handle the read request has been sent to
         * and call the appropriate function.
         */

        if (evt->handle == mcs->mc_char_value_h) {
                do_char_value_read(mcs, evt);
        } else if (evt->handle == mcs->mc_char_value_ccc_h) {
                uint16_t ccc = 0x0000;

                /* Extract the CCC value from the ble storage */
                ble_storage_get_u16(evt->conn_idx, mcs->mc_char_value_ccc_h, &ccc);

                // We're little-endian - OK to write directly from uint16_t
```

```
            ble_gatts_read_cfm(evt->conn_idx, evt->handle, ATT_ERROR_OK,
                                                        sizeof(ccc), &ccc);

        /*
         * Otherwise, read operations are not permitted
         */
        } else {
            ble_gatts_read_cfm(evt->conn_idx, evt->handle,
                    ATT_ERROR_READ_NOT_PERMITTED, 0, NULL);
        }

}


/* Handler for write requests, that is BLE_EVT_GATTS_WRITE_REQ */
static void handle_write_req(ble_service_t *svc,
                        const ble_evt_gatts_write_req_t *evt)
{
        mc_service_t *mcs = (mc_service_t *) svc;
        att_error_t status = ATT_ERROR_WRITE_NOT_PERMITTED;

        /*
         * Identify for which attribute handle the write request has been sent to
         * and call the corresponding function.
         */

        if (evt->handle == mcs->mc_char_value_h) {
            status = do_char_value_write(mcs, evt->conn_idx,
                        evt->offset, evt->length, evt->value);
            goto done;
        } else if (evt->handle == mcs->mc_char_value_ccc_h) {
            status = do_char_value_ccc_write(mcs, evt->conn_idx,
                        evt->offset, evt->length, evt->value);
            goto done;
        }

done:
            if (status == ((att_error_t) - 1)) {
            // Write handler executed properly, will be replied by cfm call
                return;
            }

            /*
             * Otherwise, write operations are not permitted
             */
            ble_gatts_write_cfm(evt->conn_idx, evt->handle, status);
}



/* Function to be called after a cleanup event */
static void cleanup(ble_service_t *svc)
{
        mc_service_t *mcs = (mc_service_t *) svc;
```

```
        ble_storage_remove_all(mcs->mc_char_value_ccc_h);

        OS_FREE(mcs);
}


/* Initialization function for My Custom Service (mcs).*/
ble_service_t *mcs_init(const uint8_t *variable_value ,
                                const my_custom_service_cb_t *cb)
{
        mc_service_t *mcs;

        uint16_t num_attr;
        att_uuid_t uuid;

        uint16_t char_user_descriptor_h;

        /* Allocate memory for the service handle */
        mcs = (mc_service_t *)OS_MALLOC(sizeof(*mcs));
        memset(mcs, 0, sizeof(*mcs));

         /* Declare handlers for specific BLE events */
        mcs->svc.read_req  = handle_read_req;
        mcs->svc.write_req = handle_write_req;
        mcs->svc.cleanup   = cleanup;
        mcs->cb = cb;

        /*
         * 0 --> Number of Included Services
         * 1 --> Number of Characteristic Declarations
         * 2 --> Number of Descriptors
         */
        num_attr = ble_gatts_get_num_attr(0, 1, 2);


        /* Service declaration */
        ble_uuid_from_string("00000000-1111-2222-2222-333333333333", &uuid);
        ble_gatts_add_service(&uuid, GATT_SERVICE_PRIMARY, num_attr);


        /* Characteristic declaration */
        ble_uuid_from_string("11111111-0000-0000-0000-111111111111", &uuid);
        ble_gatts_add_characteristic(&uuid, GATT_PROP_READ | GATT_PROP_NOTIFY |
           GATT_PROP_WRITE, ATT_PERM_RW, 1, GATTS_FLAG_CHAR_READ_REQ, NULL,
                                        &mcs->mc_char_value_h);


        /* Descriptor declaration - Client Characteristic Configuration (CCC) */
        ble_uuid_create16(UUID_GATT_CLIENT_CHAR_CONFIGURATION, &uuid);
        ble_gatts_add_descriptor(&uuid, ATT_PERM_RW, 2, 0,
                                &mcs->mc_char_value_ccc_h);

         /* Descriptor declaration – Characteristic User Description (CUD) */
        ble_uuid_create16(UUID_GATT_CHAR_USER_DESCRIPTION, &uuid);
        ble_gatts_add_descriptor(&uuid, ATT_PERM_READ,
```

```
        sizeof(char_user_descriptor_val), 0, &char_user_descriptor_h);


    /*
     * Register all the attribute handles so that they can
     * be updated by the BLE manager automatically.
     */
    ble_gatts_register_service(&mcs->svc.start_h, &mcs->mc_char_value_h,
            &mcs->mc_char_value_ccc_h, &char_user_descriptor_h ,0);

    /* Calculate the last attribute handle of the BLE service */
    mcs->svc.end_h = mcs->svc.start_h + num_attr;

    /* Set default attribute values */
    ble_gatts_set_value(mcs->mc_char_value_h, 1, variable_value);
    ble_gatts_set_value(char_user_descriptor_h,
        sizeof(char_user_descriptor_val), char_user_descriptor_val);

    /* Register the BLE service in BLE framework */
    ble_service_add(&mcs->svc);

     /* Return the service handle */
    return &mcs->svc;

}
```

4.3 Initializing the Custom BLE Service

In **ble_peripheral_task.c**, before ble_peripheral_task(), declare all the user-defined callback functions as well as variables for initializing the custom BLE service:

```
#if CFG_MY_CUSTOM_SERVICE
#include "my_custom_service.h"
#include "hw_gpio.h"


/* LED D2 status flag */
__RETAINED_RW volatile bool pin_status_flag = 0;

/* Characteristic value */
__RETAINED_RW uint8_t mcs_char_val = 0;

/* Handle of custom BLE service */
__RETAINED_RW ble_service_t *mcs = NULL;


/* Handler for read requests */
static void mcs_get_char_val_cb(ble_service_t *svc, uint16_t conn_idx)
{
    uint8_t var_value = mcs_char_val;

    /* Send the requested data to the peer device */
```

```
        mcs_get_char_value_cfm(svc, conn_idx, ATT_ERROR_OK, &var_value);
}


/* Handler for write requests */
static void mcs_set_char_val_cb(ble_service_t *svc, uint16_t conn_idx,
                                                    const uint8_t *value)
{
        mcs_char_val = *value;

    /*
     * Check the written value and if it is equal to 0x01 then turn on
     * LED D2 on DevKit.
     */
    if (mcs_char_val == 0x01) {
            hw_gpio_set_active(HW_GPIO_PORT_1, HW_GPIO_PIN_5);
            pin_status_flag = 1;  // Turn on LED D2
    } else {
            hw_gpio_set_inactive(HW_GPIO_PORT_1, HW_GPIO_PIN_5);
            pin_status_flag = 0;  // Turn off LED D2
    }

    /* Send an ACK to the peer device as a response to  the write request */
    mcs_set_char_value_cfm(svc, conn_idx, ATT_ERROR_OK);

    /*
     * Notify all the connected peer devices that characteristic
     * value has been changed.
     */
    mcs_notify_char_value_all(mcs, &mcs_char_val);
}

/* Declare callback functions for specific BLE events */
static const my_custom_service_cb_t mcs_callbacks = {
        .get_characteristic_value = mcs_get_char_val_cb,
        .set_characteristic_value = mcs_set_char_val_cb,
};
#endif
```

In **ble_peripheral_task.c**, inside ble_peripheral_task() and before starting advertising, register the custom BLE service in Dialog BLE framework:

```
#if CFG_MY_CUSTOM_SERVICE
        /* Initialize the custom BLE service */
        mcs = mcs_init(&mcs_char_val, &mcs_callbacks);
#endif
```

Optionally, in **ble_peripheral_task.c** source file change the advertising data:

```
/*
```

```
 * BLE peripheral advertising data
 */
static const uint8_t adv_data[] = {
      0x12, GAP_DATA_TYPE_LOCAL_NAME,
      'M', 'y', ' ', 'C', 'u', 's', 't', 'o', 'm', ' ', 'S', 'e', 'r', 'v', 'i', 'c', 'e'
};
```

4.4 Macro Definitions

In config/ble_peripheral_config.h, disable all the predefined services and add/enable the custom service as provided by this tutorial.

```
// enable debug service (see readme.txt for details)
#define CFG_DEBUG_SERVICE    (0)

#define CFG_BAS              (0)    // register BAS service
#define CFG_BAS_MULTIPLE     (0)    // add 2 instances of BAS service
#define CFG_CTS              (0)    // register CTS
#define CFG_DIS              (0)    // register DIS
#define CFG_DIS_FULL         (0)    // add all possible characteristics to DIS
#define CFG_SCPS             (0)    // register ScPS
#define CFG_USER_SERVICE     (0)    // register custom service (using 128-bit UUIDs)

#define CFG_MY_CUSTOM_SERVICE   (1)    // register my custom service
```

4.5 Hardware Initialization

In **main.c**, replace prvSetupHardware() with the following code to configure pins after a power-up/wake-up cycle. Please note that every time the system enters sleep, it loses all its pin configurations.

```
__RETAINED_RW extern volatile bool pin_status_flag;

/**
 * @brief Initialize the peripherals domain after power-up.
 *
 */
static void periph_init(void)
{
      hw_gpio_configure_pin(HW_GPIO_PORT_1, HW_GPIO_PIN_5,
       HW_GPIO_MODE_OUTPUT, HW_GPIO_FUNC_GPIO, pin_status_flag);

}

/**
 * \brief Initialize the peripherals domain after power-up.
 *
 */
static void prvSetupHardware( void )
```

```
{
#if mainCHECK_INTERRUPT_STACK == 1
	extern unsigned long _vStackTop[], _pvHeapStart[];
	unsigned long ulInterruptStackSize;
#endif

	/* Init hardware */
	pm_system_init(periph_init);

#if mainCHECK_INTERRUPT_STACK == 1
	/* The size of the stack used by main and interrupts is not defined in
	   the linker, but just uses whatever RAM is left.  Calculate the amount of
	   RAM available for the main/interrupt/system stack, and check it against
	   a reasonable number.  If this assert is hit then it is likely you don't
	   have enough stack to start the kernel, or to allow interrupts to nest.
	   Note - this is separate to the stacks that are used by tasks.  The stacks
	   that are used by tasks are automatically checked if
	   configCHECK_FOR_STACK_OVERFLOW is not 0 in FreeRTOSConfig.h –
	   but the stack used by interrupts is not.  Reducing the conifgTOTAL_HEAP_SIZE
	   setting will increase the stack available to main() and interrupts. */
	ulInterruptStackSize = ( ( unsigned long ) _vStackTop ) - ( ( unsigned long )
_pvHeapStart );
	OS_ASSERT( ulInterruptStackSize > 350UL );

	/* Fill the stack used by main() and interrupts to a known value, so its
	   use can be manually checked. */
	memcpy( ( void * ) _pvHeapStart, ucExpectedInterruptStackValues, sizeof(
ucExpectedInterruptStackValues ) );
#endif
}
```

Revision History

Revision	Date	Description
1.0	20-Dec-2017	First released version
2.0	18-Sep-2018	Updated application code, More descriptive steps

Status Definitions

Status	Definition
DRAFT	The content of this document is under review and subject to formal approval, which may result in modifications or additions.
APPROVED or unmarked	The content of this document has been approved for publication.

Disclaimer

Contacting Dialog Semiconductor

United Kingdom (Headquarters)
Dialog Semiconductor (UK) LTD
Phone: +44 1793 757700

Germany
Dialog Semiconductor GmbH
Phone: +49 7021 805-0

The Netherlands
Dialog Semiconductor B.V.
Phone: +31 73 640 8822

Email:
enquiry@diasemi.com

North America
Dialog Semiconductor Inc.
Phone: +1 408 845 8500

Japan
Dialog Semiconductor K. K.
Phone: +81 3 5769 5100

Taiwan
Dialog Semiconductor Taiwan
Phone: +886 281 786 222

Web site:
www.dialog-semiconductor.com

Hong Kong
Dialog Semiconductor Hong Kong
Phone: +852 2607 4271

Korea
Dialog Semiconductor Korea
Phone: +82 2 3469 8200

China (Shenzhen)
Dialog Semiconductor China
Phone: +86 755 2981 3669

China (Shanghai)
Dialog Semiconductor China
Phone: +86 21 5424 9058