Chapter 4: Descriptors, Notifications and Indications

In this chapter you will learn about Bluetooth™ LE characteristic descriptors including how to set up the descriptor for notifications and indications. You will learn how to us the Bluetooth™ configurator to enable notifications and indications, how to implement them in the client firmware and how to test them using the AIROC™ Connect app.

Table of contents

Document conventions

Convention	Usage	Example
`Courier New`	Displays code and text commands	`CY_ISR_PROTO(MyISR);` `make build`
Italics	Displays file names and paths	*sourcefile.hex*
[**bracketed, bold**]	Displays keyboard commands in procedures	[**Enter**] or [**Ctrl**] [**C**]
Menu > Selection	Represents menu paths	**File > New Project > Clone**
Bold	Displays GUI commands, menu paths and selections, and icon names in procedures	Click the **Debugger** icon, and then click **Next**.

4.1 Characteristic Descriptors and the CCCD

In the previous chapter, you learned that Bluetooth™ LE Characteristics have at least two entries in the GATT database: the Characteristic declaration and the Characteristic Value declaration. In addition, there may be other entries that contain other data about the Characteristic. These other entries are called Characteristic Descriptors and are defined by the Bluetooth™ SIG. A few of them are:

Name	Uniform Type Identifier	Assigned Number	Specification
Characteristic Aggregate Format	org.bluetooth.descriptor.gatt.characteristic_aggregate_format	0x2905	GSS
Characteristic Extended Properties	org.bluetooth.descriptor.gatt.characteristic_extended_properties	0x2900	GSS
Characteristic Presentation Format	org.bluetooth.descriptor.gatt.characteristic_presentation_format	0x2904	GSS
Characteristic User Description	org.bluetooth.descriptor.gatt.characteristic_user_description	0x2901	GSS
Client Characteristic Configuration	org.bluetooth.descriptor.gatt.client_characteristic_configuration	0x2902	GSS
Environmental Sensing Configuration	org.bluetooth.descriptor.es_configuration	0x290B	GSS
Environmental Sensing Measurement	org.bluetooth.descriptor.es_measurement	0x290C	GSS
Environmental Sensing Trigger Setting	org.bluetooth.descriptor.es_trigger_setting	0x290D	GSS
External Report Reference	org.bluetooth.descriptor.external_report_reference	0x2907	GSS
Number of Digitals	org.bluetooth.descriptor.number_of_digitals	0x2909	GSS
Report Reference	org.bluetooth.descriptor.report_reference	0x2908	GSS
Server Characteristic Configuration	org.bluetooth.descriptor.gatt.server_characteristic_configuration	0x2903	GSS
Time Trigger Setting	org.bluetooth.descriptor.time_trigger_setting	0x290E	GSS
Valid Range	org.bluetooth.descriptor.valid_range	0x2906	GSS
Value Trigger Setting	org.bluetooth.descriptor.value_trigger_setting	0x290A	GSS

A commonly used Characteristic Descriptor is the Characteristic User Description which is just a text string that describes, in human-readable format, the Characteristic Type. Many GATT Database Browsers (e.g. Light Blue and AIROC™ Connect) will display this information when you are looking at the GATT Database. As you will see later, you can use the Bluetooth™ Configurator to easily add and setup Characteristic Descriptors.

4.1.1 Client Characteristic Configuration Descriptor (CCCD)

One of the descriptors listed above is the Client Characteristic Configuration Descriptor. It is used to specify that a Characteristic should support Notifications or Indications. You will see the details of how this works in the next section.

4.2 Notifications and Indications

In the previous chapter, you saw how the GATT Client can Read and Write the GATT Database running on the GATT Server. However, there are cases where you might want the Server to initiate communication. For example, if your Server is a Peripheral device, you might want to send the Client an update each time the state of a button changes. That leaves us with the obvious questions of how does the Server initiate communication to the Client, and how does the Server know if the Client wants those messages?

The answer to the first question is, the Server can notify the Client that one of the values in the GATT Database has changed by sending a Notification or Indication message. That message has the Handle of the Characteristic that has changed and a new value for that Characteristic. Notification messages are not responded to by the Client, and as such are not reliable. If you need a reliable message, you can instead send an Indication which the Client must respond to.

To answer the second question, the GATT Server will not send Notification or Indication messages unless they are turned on by the Client. There are four parts to this:

First, the Server must have a Client Characteristic Configuration Descriptor (CCCD) for the Characteristic in question. The CCCD is simply a 16-bit mask field, where bit 0 represents the Notification flag, and bit 1 represents the Indication flag. In other words, the Client can write a 1 to bit 0 of the CCCD to tell the Server that it wants Notifications. This is easily done using the Bluetooth™ Configurator.

Second, you must change the Properties for the Characteristic to specify that the characteristic allows notifications or indications. This is also easily done in the Bluetooth™ Configurator.

Third, you need to save the CCCD value when it is written to you. This is already handled by the template code for the GATT write handler whenever the CCCD is written by the Client. That is, the Client writes to the CCCD Attribute just like any other Attribute.

Finally, when a value that has Notify and/or Indicate enabled changes in your system, you must send out a new value using the appropriate API function:

```
wiced_bt_gatt_server_send_notification(conn_id, handle, length, value, context)
wiced_bt_gatt_server_send_indication(conn_id, handle, length, value, context)
```

4.3 Creating an Application With Notifications

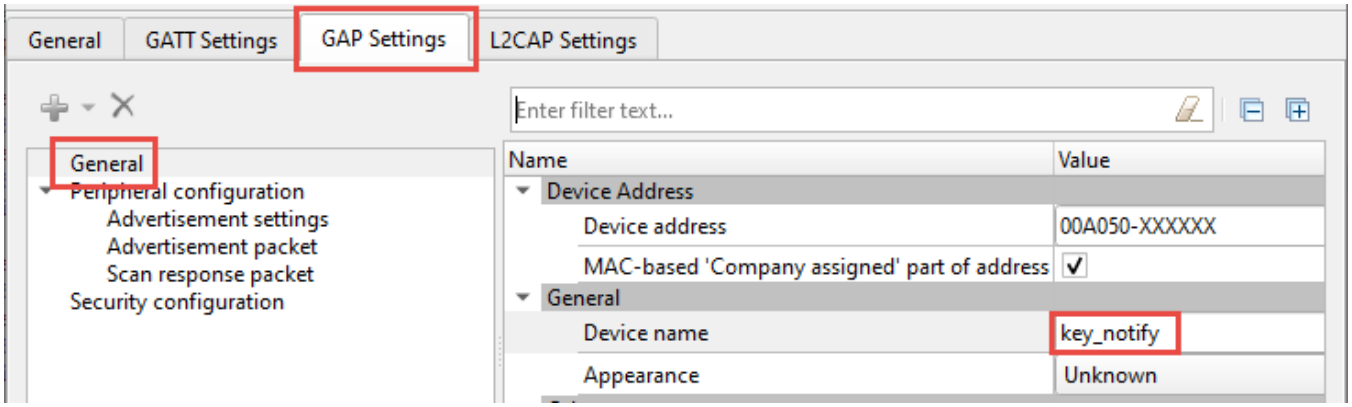
In this section, we will start with exercise 2 from the previous chapter and will add a new Characteristic that keeps track of how many times the user button on the kit has been pressed. We will enable Notifications on that Characteristic and update the firmware to send a Notification whenever the button is pressed, but only if the Client has enabled them.

You will get to try this yourself in the first exercise.

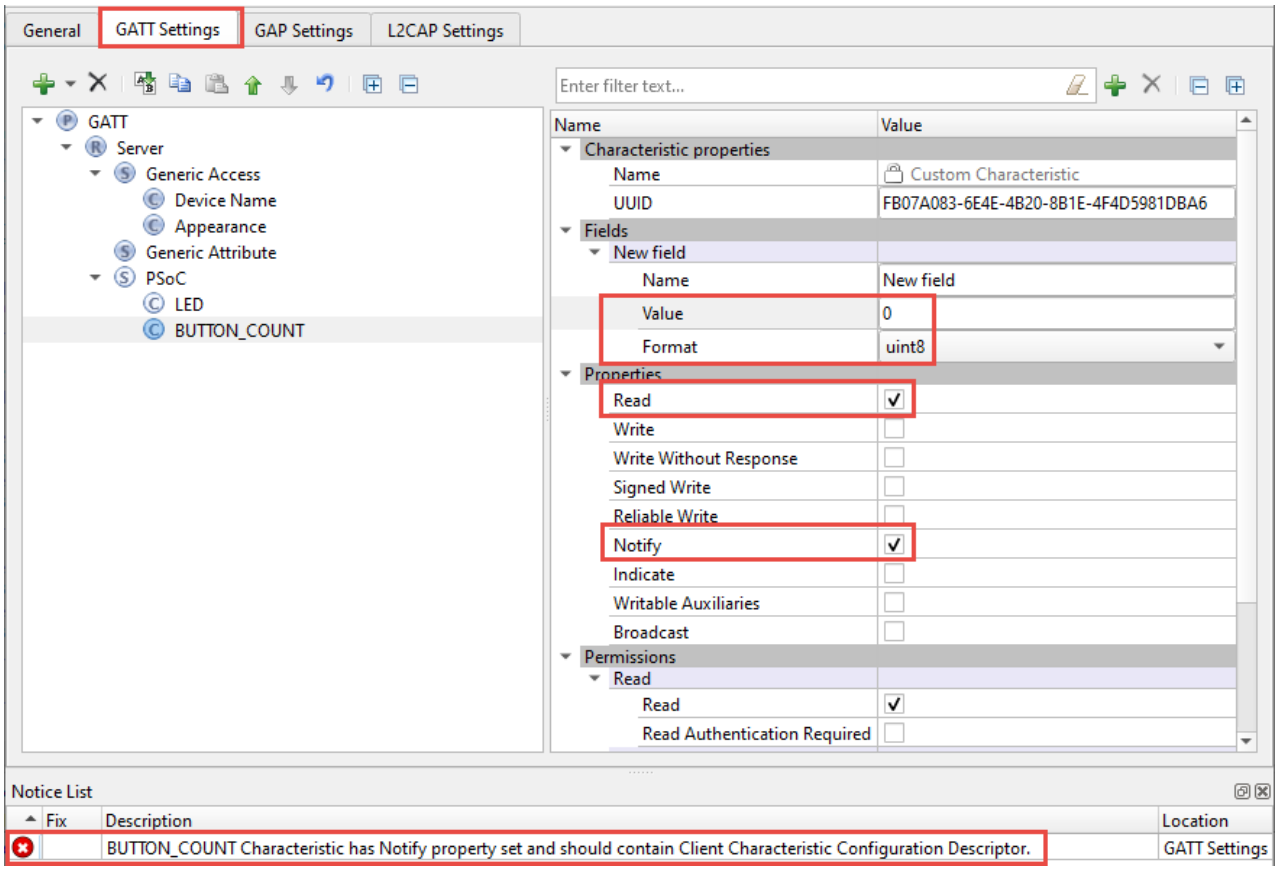
4.3.1 Configurator

You will do this in Exercise 1:

Once the application is created using the prior completed exercise as the template, the next step is to run the Bluetooth™ Configurator. Open the **GAP Settings** tab, and change the device name to <inits>_notify.

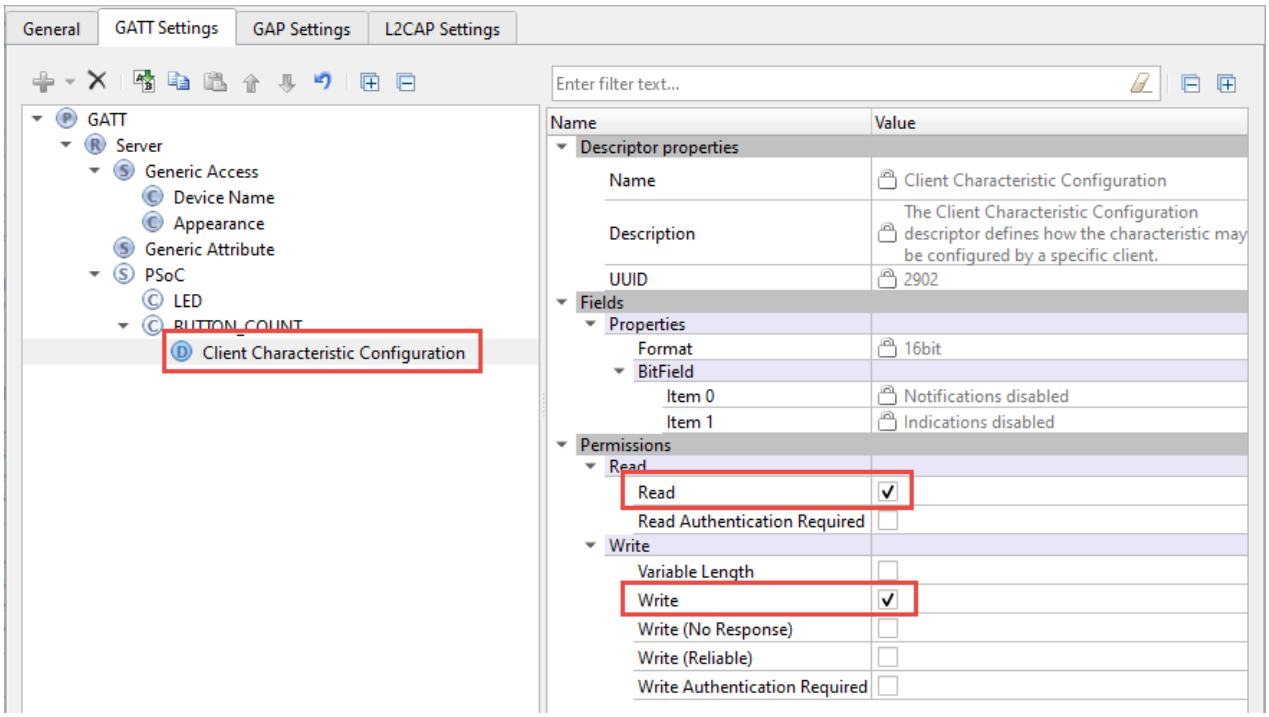


Next go to the GATT Settings tab, right click on the "PSoC" Service and add a new custom Characteristic. Rename the new Characteristic to BUTTON_COUNT. Set the format to `unit8` and set the initial value to 0. In this case, it doesn't make sense to write to this characteristic since it is counting mechanical button presses so we will set it to read but we will also enable notifications.



Notice the error message that says you need a Client Characteristic Configuration Descriptor. We will add that next.

Right click on the BUTTON_COUNT Characteristic and select **Add Descriptor > Client Characteristic Configuration**. We will leave the permissions as they are – this will allow the Client to read and write the value CCCD.



That's all we need in the configurator. Next, save and close.

4.3.2 Firmware

You will do this in Exercise 1:

The changes required in the firmware are straightforward to add the notification functionality.

First, we will configure the pin with the user button called `CYBSP_USER_BUTTON` to have a falling edge interrupt. This can be done in main.

```
/* Initialize the user button and configure a falling edge interrupt */
cyhal_gpio_init(CYBSP_USER_BTN, CYHAL_GPIO_DIR_INPUT, CYHAL_GPIO_DRIVE_PULLUP,
                CYBSP_BTN_OFF);
cyhal_gpio_register_callback(CYBSP_USER_BTN, &app_button_isr_data);
cyhal_gpio_enable_event(CYBSP_USER_BTN, CYHAL_GPIO_IRQ_FALL, 7, true);
```

In the button ISR, we will just increment the value of the BUTTON_COUNTER Characteristic and unlock a separate task that is used to handle notifications. This is done because Bluetooth™ stack functions should never be called from inside an interrupt service routine.

Let me say that again because it is important – you should never call a Bluetooth™ stack function from inside an interrupt service routine.

```
static void app_button_isr(void *handler_arg, cyhal_gpio_event_t event)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Increment button counter */
    app_psoc_button_count[0]++;

    /* Notify the counter task that the button was pressed */
    vTaskNotifyGiveFromISR( NotifyTaskHandle, &xHigherPriorityTaskWoken );

    /* If xHigherPriorityTaskWoken is now set to pdTRUE then a context
    Switch should be performed to ensure the interrupt returns
    directly to the highest priority task. The macro used for this
    purpose is dependent on the port in use and may be called
    portEND_SWITCHING_ISR(). */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Note: *The name of the BUTTON_COUNTER Characteristic value can be found in GeneratedSource/cycfg_gatt_db.c.*

To complete the interrupt, don't forget the structure to configure the interrupt callback and its arguments. This must be done after the *app_button_isr* function is declared but before it is used in *main*.

```
/* Structure for GPIO interrupt */
cyhal_gpio_callback_data_t app_button_isr_data =
{
    .callback     = app_button_isr,
    .callback_arg = NULL

};
```

We are going to need a task to send the notifications so we will setup a task handle as a global variable.

```
/* Global variable for notify task handle */
TaskHandle_t NotifyTaskHandle = NULL;
```

Then, we will create the task in main before starting the FreeRTOS scheduler.

```
/* Start task to handle Counter notifications */
xTaskCreate (notify_task, "NotifyTask", TASK_STACK_SIZE, NULL, TASK_PRIORITY,
             &NotifyTaskHandle);
```

Finally, we need to actually define the task. Whenever the button interrupt unlocks the task it will print a message and then send the notification.

```
static void notify_task(void * arg)
{
    /* Notification values received from ISR */
    uint32_t ulNotificationValue;
    while(true)
    {
        /* Wait for the button ISR */
        ulNotificationValue = ulTaskNotifyTake(pdFALSE, portMAX_DELAY);

        /* If button was pressed increment value and check to see if a
        * BLE notification should be sent. If this value is not 1, then
        * it was not a button press (most likely a timeout) that caused
        * the event so we don't want to send a BLE notification. */
        if (ulNotificationValue == 1)
        {
            if( connection_id ) /* Check if we have an active connection */
            {
                /* Check to see if the client has asked for notifications */
                if( app_psoc_button_count_client_char_config[0] &
                    GATT_CLIENT_CONFIG_NOTIFICATION )
                {
                     printf( "Notify button press count: (%d)\n",
                             app_psoc_button_count[0] );
                     wiced_bt_gatt_server_send_notification( connection_id,
                              HDLC_PSOC_BUTTON_COUNT_VALUE,
                              app_psoc_button_count_len,
                              app_psoc_button_count,
                              NULL);
                }
            }
        }
    }

}
```

Note: *The name of the Characteristic Value handle can be found in GeneratedSource/cycfg_gatt_db.h. Make sure you use the Characteristic Value, not the Characteristic Declaration.*

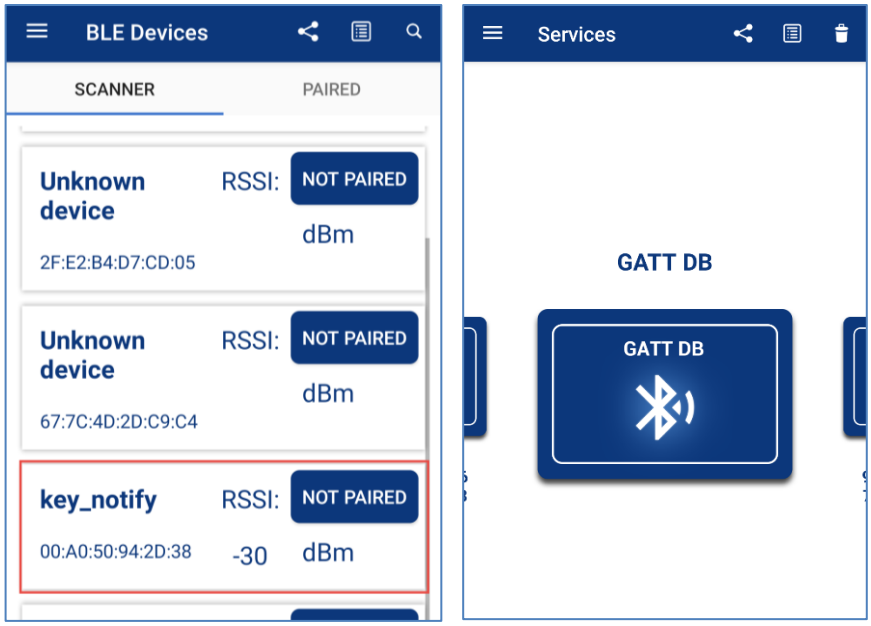
Note: *The value GATT_CLIENT_CONFIG_NOTIFICATION is an enumeration that can be found in btstack/<version>/wiced_include/wiced_bt_gatt.h.*

That's all we need! The stack takes care of actually sending out the notification for us whenever `wiced_bt_gatt_server_send_notification` is called. Once it is sent, you will get a callback event `GATT_HANDLE_VALUE_NOTIF`.

4.3.3 Testing the Application

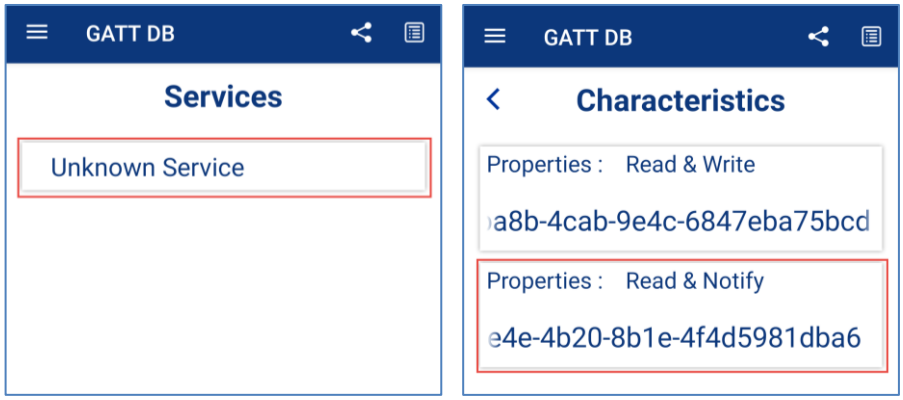
You will do this in Exercise 1:

Start up a UART terminal connected to the kit's port so and then program the kit. Run AIROC™ Connect on your phone. When you see the "<inits>_notify" device, tap on it. The app will connect to the device and will show the GATT browser widget.

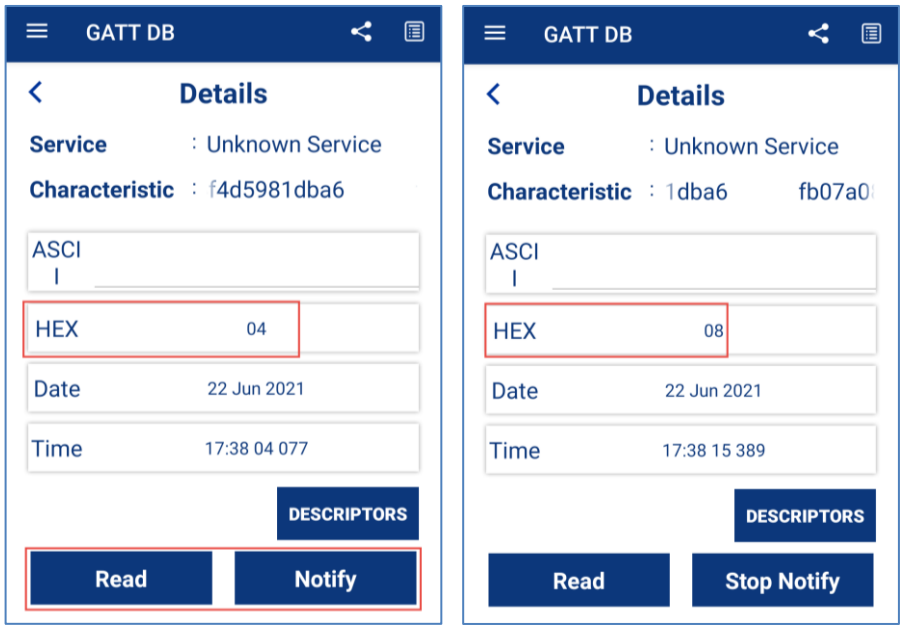


Note: If there are too many devices listed, you can use the search button to find your device quickly.

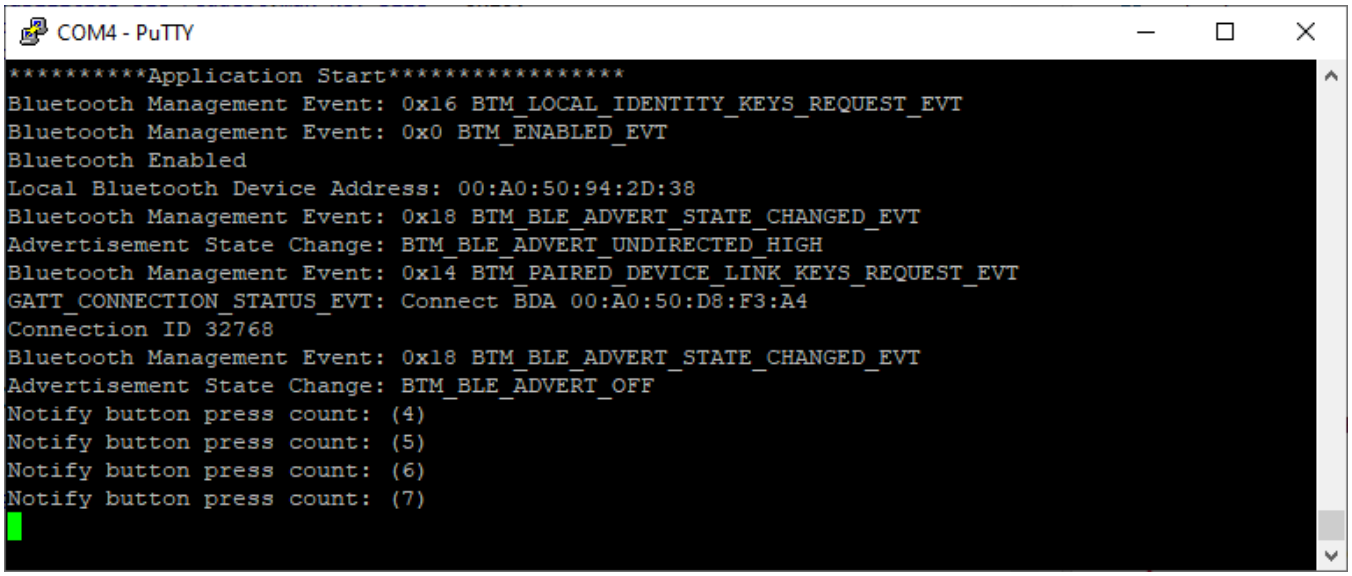
Tap on the GATT DB widget to open the browser. Then tap on the Unknown Service (which we know is "PSoC") and then on the Characteristic with Read and Notify Properties (which we know is BUTTON_COUNT).



Tap the Read button to read the value. Press the button on the kit a few times and then Read again to see the incremented value. Then tap the Notify button to enable notifications. Press the button a few more times and notice how each time you press the button the value is updated automatically.



While using AIROC™ Connect you will see messages like this in the UART terminal window:



4.4 Indication Responses

Indications behave just like Notifications except that the Client is required to send a response back to the Server when it has received the notification.

The changes to the GATT database are pretty obvious – just enable Indicate instead of Notify in the Characteristic's Properties. The CCCD is exactly the same. In the firmware, use the function `wiced_bt_gatt_server_send_indication` to send the indication. In this case, you will check the CCCD

for `GATT_CLIENT_CONFIG_INDICATION` to decide if the Client wants an indication to be sent. When the stack sends the indication, you will get the callback event `GATT_HANDLE_VALUE_IND`.

Once a response is received from the Client, the stack will generate a `GATT_HANDLE_VALUE_CONF` callback event. You can use this event to validate that the indication was received by the Client.

4.5 Exercises

Exercise 1: Notifications

For this exercise, you will recreate the application described earlier in this chapter and will test it with AIROC™ Connect. As a reminder, it will have a Characteristic called BUTTON_COUNT that counts button presses and will send notifications if the Client has asked for them.

1. Create a new ModusToolbox™ application for the BSP you are using.

   On the application template page, use the **Browse** button to start from the completed application for exercise 2 from the previous chapter. If you did not complete that exercise, the solution can be found in *Projects/key_ch03_ex02_status*.
   Name the new application *ch04_ex01_notify*.

2. Follow the instructions in section 4.3.1 to modify the Bluetooth™ configuration to add a new Characteristic that supports notifications.

   Note: If you previously reduced the advertising timeouts you may want to increase them again so that you have more time to connect to the device.

3. Open *main.c* and follow the instructions in section 4.3.2 to complete the necessary changes in the code.
4. Build the application and program your kit.
5. Follow the instructions in section 4.3.3 to test your application.

Exercise 2: Indications

In this exercise, you will start from the previous exercise and will change the Notifications to Indications. You will print a message to the UART when an Indication response is received from the Client.

1. Create a new ModusToolbox™ application for the BSP you are using.

   On the application template page, use the **Browse** button to start from the completed application for exercise 1. If you did not complete exercise 1, the solution can be found in *Projects/key_ch04_ex01_notify*. Name the new application *ch04_ex02_indicate*.

2. Open the Bluetooth® configurator and set the device name to **<inits>_indicate**.

3. Change the properties on the BUTTON_COUNT Characteristic to enable Indications instead of Notifications.

4. Save changes and close the configurator.

5. Open *main.c*.

6. Modify *notify_task* to send indications instead of notifications

Note: *There are 2 changes required – you must change the if statement that checks the value of the CCCD to look for indications instead of notifications and you must call a different API function to send an Indication.*

Note: *Optional: You may want to change the name of the task and the* `printf` *message to be more consistent with Indications instead of Notifications.*

7. Print a message when a confirmation is received from the Client.

   An empty case for `GATT_HANDLE_VALUE_CONF` is already included in the GATT Attribute Request event handler function. Just add the message to the existing empty case.

8. Build the application and program your kit.

9. Test the application the same way you tested the prior exercise.

   Notice the message in the UART when an Indication response is received from the Client.

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Published by

Infineon Technologies AG

81726 Munich, Germany

IMPORTANT NOTICE

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics (“Beschaffenheitsgarantie”) .

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer’s compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer’s products and any use of the product of Infineon Technologies in customer’s applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer’s technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (**www.infineon.com**).

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies’ products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.