

# Chapter 5c: BLE Hosted Mode

After completing this chapter, you will understand how to create a simple BLE client to talk to a smart phone using a PSoC 6 as a host for a connected CYW43012 device.

<b>5C.1 INTRODUCTION .....</b>	<b>2</b>
<b>5C.2 BLUETOOTH STACK EVENTS .....</b>	<b>3</b>
5c.2.1 ESSENTIAL BLUETOOTH MANAGEMENT EVENTS .....	3
5c.2.2 ESSENTIAL GATT EVENTS .....	3
5c.2.3 ESSENTIAL GATT SUB-EVENTS .....	3
<b>5C.3 BLUETOOTH FIRMWARE ARCHITECTURE .....</b>	<b>4</b>
5c.3.1 APPLICATION STRUCTURE .....	4
5c.3.2 MAKEFILE .....	5
5c.3.3 HEADER FILES .....	5
5c.3.4 TURNING ON THE STACK .....	5
5c.3.5 START THE FREERTOS SCHEDULER .....	6
5c.3.6 START ADVERTISING .....	6
5c.3.7 PROCESSING CONNECTION EVENTS FROM THE STACK .....	8
5c.3.8 PROCESSING CLIENT READ EVENTS FROM THE STACK .....	9
5c.3.9 PROCESSING CLIENT WRITE EVENTS FROM THE STACK .....	10
<b>5C.4 DEMO WALKTHROUGH.....</b>	<b>11</b>
5c.4.1 APPLICATION CREATION .....	11
5c.4.2 BLUETOOTH CONFIGURATOR .....	12
5c.4.3 EDITING THE FIRMWARE .....	16
5c.4.4 TESTING THE APPLICATION .....	18
<b>5C.5 EXERCISES (PART 1) .....</b>	<b>21</b>
5c.5.1 EXERCISE 1: BASIC BLE PERIPHERAL .....	21
<b>5C.6 GATT DATABASE IMPLEMENTATION .....</b>	<b>21</b>
5c.6.1 GATT_DATABASE[] .....	21
5c.6.2 GATT_DB_EXT_ATTR_TBL .....	23
5c.6.3 UINT8_T ARRAYS FOR THE VALUES .....	24
5c.6.4 APPLICATION PROGRAMMING INTERFACE .....	24
<b>5C.7 NOTIFICATIONS .....</b>	<b>25</b>
<b>5C.8 NOTIFICATION DEMO WALKTHROUGH .....</b>	<b>26</b>
5c.8.1 RUNNING THE BLUETOOTH CONFIGURATOR .....	27
5c.8.2 EDITING THE FIRMWARE .....	31
5c.8.3 TESTING THE APPLICATION .....	33
<b>5C.9 EXERCISES (PART 2) .....</b>	<b>36</b>
5c.9.1 EXERCISE 2: NOTIFICATION .....	36
<b>5C.10 SECURITY .....</b>	<b>36</b>
5c.10.1 PAIRING .....	37
5c.10.2 BONDING .....	39
5c.10.3 PAIRING & BONDING PROCESS SUMMARY .....	39
5c.10.4 AUTHENTICATION, AUTHORIZATION AND THE GATT DB .....	39
5c.10.5 SECURITY IN THE BLUETOOTH CONFIGURATOR .....	40
5c.10.6 LINK LAYER PRIVACY .....	41
<b>5C.11 FIRMWARE ARCHITECTURE FOR SECURITY .....</b>	<b>42</b>
<b>5C.12 EXERCISES (PART 3) .....</b>	<b>47</b>
5c.12.1 EXERCISE 3: PARING .....	47
5c.12.2 EXERCISE 4: BONDING .....	50
5c.12.3 EXERCISE 5: PASSKEY AND NUMERIC COMPARISON .....	55

## 5c.1 Introduction

As you have already seen, the PSoC 6 plus 43xxx device provides a powerful IoT solution. In addition to the MCU functionality and Wi-Fi functionality demonstrated so far, this solution is also capable of performing BLE operations.

As was described in the BLE Basics chapter, the upper part of the BLE stack runs on the PSoC 6 while the lower part of the stack runs on the CYW4343W or CYW43012. Communication between the two parts of the stack is done using HCI which uses a 4 wire UART (Tx, Rx, CTS, RTS) at a baud rate of 115,200.

There are also 2 additional pins used for low power modes: one for the host to wake the device and one for the device to wake the host. This will be discussed in the low power chapter.

Recall that there are 4 basic steps to make a basic BLE Peripheral:

- Turn on the Stack
- Start Advertising
- Process Connection Events from the Stack
- Process Read/Write Events from the Stack

The Bluetooth firmware you write will follow this flow:

1. Initialize the stack and provide it with a Bluetooth management event callback function and a stack configuration structure.
2. In the Bluetooth management callback function:
  - a. Initialize any other required resources when the BTM\_ENABLED\_EVT occurs (i.e. when the stack has started up). For example:
    - i. Initialize the GATT database and provide a GATT event callback function
    - ii. Start BLE advertising
  - b. Respond to all other required Bluetooth management events. For example:
    - i. Respond to paring requests
    - ii. Respond to security requests
3. In the GATT event callback, respond to all required GATT events. For example:
  - a. Respond to connection/disconnection events
  - b. Respond to read and write events

## 5c.2 Bluetooth Stack Events

A central part of the firmware architecture is the two callback functions that you register during initialization: one for Bluetooth management events, and one for GATT events. The Stack generates events based on what is happening in the Bluetooth world. After an event is created, the Stack will call the callback the appropriate function which you registered. Your callback firmware must look at the event code and the event parameter and take the appropriate action.

Let's look at some of the events that these two callback functions need to respond to. Note that there are more events than those listed here - these are just the essential events that just about every BLE device will need to handle.

For the purposes of a simple example, you need to understand these events:

### 5c.2.1 Essential Bluetooth Management Events

Event	Description
BTM_ENABLED_EVT	When the Stack has everything going. The event data will tell you if it happened with WICED_SUCCESS or !WICED_SUCCESS.
BTM_BLE_ADVERT_STATE_CHANGED_EVT	When Advertising is either stopped or started by the Stack. The event parameter will tell you BTM_BLE_ADVERT_OFF or one of the many different levels of active advertising.

The ModusToolbox starter template for this class provides and registers a function called `app_bt_management_callback` to handle Management events.

### 5c.2.2 Essential GATT Events

Event	Description
GATT_CONNECTION_STATUS_EVT	When a connection is made or broken. The event parameter tells you WICED_TRUE if connected.
GATT_ATTRIBUTE_REQUEST_EVT	When a GATT Read or Write occurs. The event parameter tells you GATTS_REQ_TYPE_READ or GATTS_REQ_TYPE_WRITE.

The ModusToolbox starter template for this class provides and registers a function called `app_gatt_callback` to handle GATT events.

### 5c.2.3 Essential GATT Sub-Events

In addition to the GATT events described above, there are sub-events associated with each of the main events which are handled in the template.

#### GATT\_CONNECTION\_STATUS\_EVT

For this example, there are two sub-events for a Connection Status Event that we care about. Namely:

Event	Description
<code>connected == WICED_TRUE</code>	A GATT connection has been established.
<code>connected != WICED_TRUE</code>	A GATT connection has been broken.

The `app_gatt_callback` function contains some basic code to handle connect/disconnect events and you can add your own functionality as needed.

## GATT\_ATTRIBUTE\_REQUEST\_EVT

For this example, there are two sub-events for an Attribute Request Event that we care about. Namely:

Event	Description
GATTS_REQ_TYPE_READ	A GATT Attribute Read has occurred. The event parameter tells you the request handle and where to save the data.
GATTS_REQ_TYPE_WRITE	A GATT Attribute Write has occurred. The event parameter tells you the handle, a pointer to the data and the length of the data.

The `app_gatt_callback` function contains some basic code to handle attribute read/write events and you can add your own functionality as needed. In our application the `app_gatt_callback` function calls `app_gatt_set_value` for `GATTS_REQ_TYPE_WRITE` events and that function contains the code we wrote to change the state of the LED (it does predictably similar things for READ events).

## 5c.3 Bluetooth Firmware Architecture

Now that you understand the role of the callback functions and some of the essential events, let's look at the firmware in a bit more detail.

### 5c.3.1 Application Structure

The application structure will be the same as other PSoC 6 applications with a few additions for Bluetooth. Namely:

#### Bluetooth and FreeRTOS Libraries

The Bluetooth stack uses FreeRTOS to manage various tasks, so you must include FreeRTOS even if you are not using it for other purposes in your firmware. The required libraries are:

- `bluetooth-freertos` (this also includes the `btstack` library)
- `freertos`
- `abstraction-rtos`

In addition to the libraries, you will need a file to configure the FreeRTOS settings. This file is called `FreeRTOSConfig.h`.

#### Bluetooth Configuration Settings

There is a file typically called `app_bt_cfg.c` that contains a structure of type `cybt_platform_config_t` which sets up the communication interface between the host and device. This includes both the UART and low power wake up pin configuration. This structure is passed to `cybt_platform_config_init` which is called during application initialization.

The stack configuration settings are also typically located in `app_bt_cfg.c` in a structure of type `wiced_bt_cfg_settings_t`. You will edit the stack configuration, for example, to optimize the scanning and advertising parameters. This structure is passed to `wiced_bt_stack_init` which is called during application initialization.

Your application code will include `app_bt_cfg.h` to get access to these configuration structures.

## Bluetooth Configurator Files

The Bluetooth Configurator is used to setup the GATT database for a BLE application. This tool will be shown in detail later, but the input file will have the extension .cybt. The output files from the configurator are placed in the GeneratedSource folder. Your application code will include a GeneratedSource/cycfg\_gatt\_db.h reference.

## Utility Functions

The files util\_functions.c and util\_functions.h are used to convert return codes from the Stack to text strings to simplify debugging.

### 5c.3.2 Makefile

The Makefile needs to include COMPONENTS for FREERTOS and WICED\_BLE so that the appropriate files from the libraries are included:

```
COMPONENTS= FREERTOS WICED_BLE
```

### 5c.3.3 Header Files

To use the BLE and FreeRTOS API functions, a few header files must be included in your code. Some of the includes you will typically use are:

```
#include <FreeRTOS.h>
#include <task.h>
#include "wiced_bt_stack.h"
```

Other header files may be required depending on your application's functionality such as wiced\_timer.h for timers and wiced\_memory.h for memory management utilities. The WICED header files can be found in the wiced\_include folder in the btstack library.

### 5c.3.4 Turning on the Stack

When your application firmware starts executing, it is responsible for turning on the Stack to make a connection to the WICED radio by calling wiced\_bt\_stack\_init.

One of the key arguments to wiced\_bt\_stack\_init is a function pointer to the management callback. The template uses the name app\_bt\_management\_callback for the Bluetooth management callback.

In app\_bt\_management\_callback it is your job to fill in what the firmware does to processes various events. This is implemented as a switch statement in the callback function where the cases are the Stack events. Some of the necessary actions are provided automatically and others will need to be written by you.

When you start the Stack, it generates the BTM\_ENABLED\_EVT event and calls the app\_bt\_management\_callback function which then processes that event.

The `app_bt_management_callback` case for `BTM_ENABLED_EVT` event calls the functions `wiced_bt_gatt_register` and `wiced_bt_gatt_db_init`, which registers a callback function for GATT database events and initializes the GATT database.

The `BTM_ENABLED_EVT` ends by calling the `wiced_bt_start_advertising` function.

### 5c.3.5 Start the FreeRTOS scheduler

Typically, the last step that is done in main is to start the FreeRTOS scheduler with the function `vTaskStartScheduler()`.

### 5c.3.6 Start Advertising

The Stack is triggered to start advertising by the last step of the Off → On process with the call to `wiced_bt_start_advertising`.

The function `wiced_bt_start_advertising` takes 3 arguments. The first is the advertisement type and has 9 possible values:

```
BTM_BLE_ADVERT_OFF,           /**< Stop advertising */
BTM_BLE_ADVERT_DIRECTED_HIGH, /**< Directed advertisement (high duty cycle) */
BTM_BLE_ADVERT_DIRECTED_LOW,  /**< Directed advertisement (low duty cycle) */
BTM_BLE_ADVERT_UNDIRECTED_HIGH, /**< Undirected advertisement (high duty cycle) */
BTM_BLE_ADVERT_UNDIRECTED_LOW, /**< Undirected advertisement (low duty cycle) */
BTM_BLE_ADVERT_NONCONN_HIGH,  /**< Non-connectable advertisement (high duty cycle) */
BTM_BLE_ADVERT_NONCONN_LOW,   /**< Non-connectable advertisement (low duty cycle) */
BTM_BLE_ADVERT_DISCOVERABLE_HIGH, /**< discoverable advertisement (high duty cycle) */
BTM_BLE_ADVERT_DISCOVERABLE_LOW /**< discoverable advertisement (low duty cycle) */
```

For undirected advertising (which is what we will use in our examples) the 2<sup>nd</sup> and 3<sup>rd</sup> arguments can be set to 0 and NULL respectively.

The Stack then generates the `BTM_BLE_ADVERT_STATE_CHANGED_EVT` management event and calls the `app_bt_management_callback`.

The `app_bt_management_callback` case for `BTM_BLE_ADVERT_STATE_CHANGED_EVT` looks at the event parameter to determine if it is a start or end of advertising. In the template code it does not do anything when advertising is started, but you could, for instance, turn on an LED to indicate the advertising state.

### Advertising Packets

The Advertising Packet itself is a string of 3-31 bytes that is broadcast at a configurable interval. The interval chosen has a big influence on power consumption and connection establishment time. The packet is broken up into variable length fields. Each field has the form:

- Length in bytes (not including the Length byte)
- Type
- Optional Data

The minimum packet requires the <<Flags>> field which is a set of flags that defines how the device behaves (e.g. is it connectable?).

Here is a list of the other field Types that you can add:

```
/** Advertisement data types */
enum wiced_bt_ble_advert_type_e {
    BTM_BLE_ADVERT_TYPE_FLAG                = 0x01,
    BTM_BLE_ADVERT_TYPE_16SRV_PARTIAL        = 0x02,
    BTM_BLE_ADVERT_TYPE_16SRV_COMPLETE       = 0x03,
    BTM_BLE_ADVERT_TYPE_32SRV_PARTIAL        = 0x04,
    BTM_BLE_ADVERT_TYPE_32SRV_COMPLETE       = 0x05,
    BTM_BLE_ADVERT_TYPE_128SRV_PARTIAL       = 0x06,
    BTM_BLE_ADVERT_TYPE_128SRV_COMPLETE      = 0x07,
    BTM_BLE_ADVERT_TYPE_NAME_SHORT           = 0x08,
    BTM_BLE_ADVERT_TYPE_NAME_COMPLETE        = 0x09,
    BTM_BLE_ADVERT_TYPE_TX_POWER             = 0x0A,
    BTM_BLE_ADVERT_TYPE_DEV_CLASS            = 0x0D,
    BTM_BLE_ADVERT_TYPE_SIMPLE_PAIRING_HASH_C = 0x0E,
    BTM_BLE_ADVERT_TYPE_SIMPLE_PAIRING_RAND_C = 0x0F,
    BTM_BLE_ADVERT_TYPE_SM_TK                = 0x10,
    BTM_BLE_ADVERT_TYPE_SM_OOB_FLAG          = 0x11,
    BTM_BLE_ADVERT_TYPE_INTERVAL_RANGE       = 0x12,
    BTM_BLE_ADVERT_TYPE_SOLICITATION_SRV_UUID = 0x14,
    BTM_BLE_ADVERT_TYPE_128SOLICITATION_SRV_UUID = 0x15,
    BTM_BLE_ADVERT_TYPE_SERVICE_DATA         = 0x16,
    BTM_BLE_ADVERT_TYPE_PUBLIC_TARGET        = 0x17,
    BTM_BLE_ADVERT_TYPE_RANDOM_TARGET        = 0x18,
    BTM_BLE_ADVERT_TYPE_APPEARANCE           = 0x19,
    BTM_BLE_ADVERT_TYPE_ADVERT_INTERVAL     = 0x1A,
    BTM_BLE_ADVERT_TYPE_LE_BD_ADDR           = 0x1B,
    BTM_BLE_ADVERT_TYPE_LE_ROLE              = 0x1C,
    BTM_BLE_ADVERT_TYPE_256SIMPLE_PAIRING_HASH = 0x1D,
    BTM_BLE_ADVERT_TYPE_256SIMPLE_PAIRING_RAND = 0x1E,
    BTM_BLE_ADVERT_TYPE_32SOLICITATION_SRV_UUID = 0x1F,
    BTM_BLE_ADVERT_TYPE_32SERVICE_DATA      = 0x20,
    BTM_BLE_ADVERT_TYPE_128SERVICE_DATA     = 0x21,
    BTM_BLE_ADVERT_TYPE_CONN_CONFIRM_VAL     = 0x22,
    BTM_BLE_ADVERT_TYPE_CONN_RAND_VAL        = 0x23,
    BTM_BLE_ADVERT_TYPE_URI                  = 0x24,
    BTM_BLE_ADVERT_TYPE_INDOOR_POS           = 0x25,
    BTM_BLE_ADVERT_TYPE_TRANS_DISCOVER_DATA  = 0x26,
    BTM_BLE_ADVERT_TYPE_SUPPORTED_FEATURES   = 0x27,
    BTM_BLE_ADVERT_TYPE_UPDATE_CH_MAP_IND    = 0x28,
    BTM_BLE_ADVERT_TYPE_PB_ADV               = 0x29,
    BTM_BLE_ADVERT_TYPE_MESH_MSG             = 0x2A,
    BTM_BLE_ADVERT_TYPE_MESH_BEACON          = 0x2B,
    BTM_BLE_ADVERT_TYPE_3D_INFO_DATA         = 0x3D,
    BTM_BLE_ADVERT_TYPE_MANUFACTURER         = 0xFF,
    /**< Advertisement flags */
    /**< List of supported services - 16 bit UUIDs (partial) */
    /**< List of supported services - 16 bit UUIDs (complete) */
    /**< List of supported services - 32 bit UUIDs (partial) */
    /**< List of supported services - 32 bit UUIDs (complete) */
    /**< List of supported services - 128 bit UUIDs (partial) */
    /**< List of supported services - 128 bit UUIDs (complete) */
    /**< Short name */
    /**< Complete name */
    /**< TX Power level */
    /**< Device Class */
    /**< Simple Pairing Hash C */
    /**< Simple Pairing Randomizer R */
    /**< Security manager TK value */
    /**< Security manager Out-of-Band data */
    /**< Slave connection interval range */
    /**< List of solicited services - 16 bit UUIDs */
    /**< List of solicited services - 128 bit UUIDs */
    /**< Service data - 16 bit UUID */
    /**< Public target address */
    /**< Random target address */
    /**< Appearance */
    /**< Advertising interval */
    /**< LE device bluetooth address */
    /**< LE role */
    /**< Simple Pairing Hash C-256 */
    /**< Simple Pairing Randomizer R-256 */
    /**< List of solicited services - 32 bit UUIDs */
    /**< Service data - 32 bit UUID */
    /**< Service data - 128 bit UUID */
    /**< LE Secure Connections Confirmation Value */
    /**< LE Secure Connections Random Value */
    /**< URI */
    /**< Indoor Positioning */
    /**< Transport Discovery Data */
    /**< LE Supported Features */
    /**< Channel Map Update Indication */
    /**< PB-ADV */
    /**< Mesh Message */
    /**< Mesh Beacon */
    /**< 3D Information Data */
    /**< Manufacturer data */
};

typedef uint8_t wiced_bt_ble_advert_type_t; /**< BLE advertisement data type (see #wiced_bt_ble_advert_type_e) */
```

For example, if you had a device named "Kentucky" you could add the name to the Advertising packet by adding the following bytes to your Advertising packet:

- 9 (the length is 1 for the field type plus 8 for the data)
- BTM\_BLE\_ADVERT\_TYPE\_NAME\_COMPLETE
- 'K', 'e', 'n', 't', 'u', 'c', 'k', 'y'

The Bluetooth API function `wiced_bt_ble_set_raw_advertisement_data()` will allow you to configure the data in the packet. You pass it an array of structures of type `wiced_bt_ble_advert_elem_t` and the number of elements in the array.

The `wiced_bt_ble_advert_elem_t` structure is defined as:

```
typedef struct
{
    uint8_t          *p_data;          /**< Advertisement data */
    uint16_t         len;              /**< Advertisement length */
    wiced_bt_ble_advert_type_t advert_type; /**< Advertisement data type */
}wiced_bt_ble_advert_elem_t;
```



One important note: the "len" parameter is the length of just the data. It does NOT include the 1-byte for the advertising field type.

To implement the earlier example of adding "Kentucky" to the Advertising Packet as the Device name, I could do this:

```
#define KYNAME "Kentucky"

/* Set Advertisement Data */
void testwbt_set_advertisement_data( void )
{
    wiced_bt_ble_advert_elem_t adv_elem[2] = { 0 };
    uint8_t adv_flag = BTM_BLE_GENERAL_DISCOVERABLE_FLAG | BTM_BLE_BREDR_NOT_SUPPORTED;
    uint8_t num_elem = 0;

    /* Advertisement Element for Flags */
    adv_elem[num_elem].advert_type = BTM_BLE_ADVERT_TYPE_FLAG;
    adv_elem[num_elem].len = sizeof(uint8_t);
    adv_elem[num_elem].p_data = &adv_flag;
    num_elem++;

    /* Advertisement Element for Name */
    adv_elem[num_elem].advert_type = BTM_BLE_ADVERT_TYPE_NAME_COMPLETE;
    adv_elem[num_elem].len = strlen((const char*)KYNAME);
    adv_elem[num_elem].p_data = KYNAME;
    num_elem++;

    /* Set Raw Advertisement Data */
    wiced_bt_ble_set_raw_advertisement_data(num_elem, adv_elem);
}
```

The Advertising packet enables several interesting use cases which we will talk about in more detail in the next chapter.

There is also a scan response packet that can hold an additional 31 bytes which will not be covered in this course but is explained in the full WICED Bluetooth 101 class.

### 5c.3.7 Processing Connection Events from the Stack

The getting connected process starts when a Central that is actively Scanning hears your advertising packet and decides to connect. It then sends you a connection request.

The Stack responds to the Central with a connection accepted message, and then the Stack generates a GATT event called `GATT_CONNECTION_STATUS_EVT` which is processed by the `app_gatt_callback` function.

The code for the `GATT_CONNECTION_STATUS_EVT` event uses the event parameter to determine if it is a connection or a disconnection. It then prints a message.

On a connection, the Stack then stops the advertising and calls `app_bt_management_callback` with a management event `BTM_BLE_ADVERT_STATE_CHANGED_EVT`.

The `app_bt_management_callback` determines that it is a stop of advertising and just prints out a message. You could add your own code here to, for instance, turn off an LED or restart advertisements.



### 5c.3.8 Processing Client Read Events from the Stack

When the Client wants to read the value of a Characteristic, it sends a read request with the Handle of the Attribute that holds the value of the Characteristic. We will talk about how handles are exchanged between the devices later.

The Stack generates a `GATT_ATTRIBUTE_REQUEST_EVT` and calls `app_gatt_callback`, which determines the event is `GATT_ATTRIBUTE_REQUEST_EVT`. The code for this event looks at the event parameter and determines that it is a `GATTS_REQ_TYPE_READ`, then calls the function `app_gatt_get_value` to find the current value of the Characteristic.

That function looks through that GATT Database to find the Attribute that matches the Handle requested. It then copies the value's bytes out of the GATT Database into the location requested by the Stack.

Finally, the get value function returns a code to indicate what happened - either `WICED_BT_GATT_SUCESS`, or if something bad has happened (like the requested Handle doesn't exist) it returns the appropriate error code such as `WICED_BT_GATT_INVALID_HANDLE`. The list of the return codes is taken from the `wiced_bt_gatt_status_e` enumeration. This enumeration includes (partial list):

```
enum wiced_bt_gatt_status_e
{
    WICED_BT_GATT_SUCCESS                = 0x00,  /**< Success */
    WICED_BT_GATT_INVALID_HANDLE         = 0x01,  /**< Invalid Handle */
    WICED_BT_GATT_READ_NOT_PERMIT        = 0x02,  /**< Read Not Permitted */
    WICED_BT_GATT_WRITE_NOT_PERMIT       = 0x03,  /**< Write Not permitted */
    WICED_BT_GATT_INVALID_PDU            = 0x04,  /**< Invalid PDU */
    WICED_BT_GATT_INSUF_AUTHENTICATION   = 0x05,  /**< Insufficient Authentication */
    WICED_BT_GATT_REQ_NOT_SUPPORTED      = 0x06,  /**< Request Not Supported */
    WICED_BT_GATT_INVALID_OFFSET         = 0x07,  /**< Invalid Offset */
    WICED_BT_GATT_INSUF_AUTHORIZATION    = 0x08,  /**< Insufficient Authorization */
    WICED_BT_GATT_PREPARE_Q_FULL         = 0x09,  /**< Prepare Queue Full */
    WICED_BT_GATT_NOT_FOUND              = 0x0a,  /**< Not Found */
    WICED_BT_GATT_NOT_LONG               = 0x0b,  /**< Not Long Size */
    WICED_BT_GATT_INSUF_KEY_SIZE          = 0x0c,  /**< Insufficient Key Size */
    WICED_BT_GATT_INVALID_ATTR_LEN       = 0x0d,  /**< Invalid Attribute Length */
    WICED_BT_GATT_ERR_UNLIKELY           = 0x0e,  /**< Error Unlikely */
    WICED_BT_GATT_INSUF_ENCRYPTION        = 0x0f,  /**< Insufficient Encryption */
    WICED_BT_GATT_UNSUPPORT_GRP_TYPE     = 0x10,  /**< Unsupported Group Type */
    WICED_BT_GATT_INSUF_RESOURCE         = 0x11,  /**< Insufficient Resource */
}
```

The status code generated by the get value function is returned up through the function call hierarchy and eventually back to the Stack, which in turn sends it to the Client.

To summarize, the course of events for a read is:

1. Stack calls `app_gatt_callback` with `GATT_ATTRIBUTE_REQUEST_EVT`
2. `app_gatt_callback` detects the `GATTS_REQ_TYPE_READ` request type
3. `app_gatt_callback` calls `app_gatt_get_value`

### 5c.3.9 Processing Client Write Events from the Stack

When the Client wants to write a value to a Characteristic, it sends a write request with the Handle of the Attribute of the Characteristic along with the data.

The Stack generates the GATT event `GATT_ATTRIBUTE_REQUEST_EVT` and calls the function `app_gatt_callback`, which determines the event is `GATT_ATTRIBUTE_REQUEST_EVT`. The code for this event looks at the event parameter and determines that it is a `GATTS_REQ_TYPE_WRITE`, then calls the function `app_gatt_set_value` to update the current value of the Characteristic.

The `app_gatt_set_value` function looks through that GATT Database to find the Attribute that matches the Handle requested. It then copies the value bytes from the Stack generated request into the GATT Database. Finally, the set value function returns a code to indicate what happened just like the Read - either `WICED_BT_GATT_SUCCESS`, or the appropriate error code. The list of the return codes is again taken from the `wiced_bt_gatt_status_e` enumeration.

The status code generated by the set value function is returned up through the function call hierarchy and eventually back to the Stack. One difference here is that if your callback function returns `WICED_BT_GATT_SUCCESS`, the Stack sends a Write response of 0x1E. If your callback returns something other than `WICED_BT_GATT_SUCCESS`, the stack sends an error response with the error code that you chose.

To summarize, function call hierarchy for a write is:

1. Stack calls `app_gatt_callback` with `GATT_ATTRIBUTE_REQUEST_EVT`
2. `app_gatt_callback` detects the `GATTS_REQ_TYPE_WRITE` request type
3. `app_gatt_callback` calls `app_gatt_set_value`

## 5c.4 Demo Walkthrough

There are lots of examples available that show BLE functionality. For example:

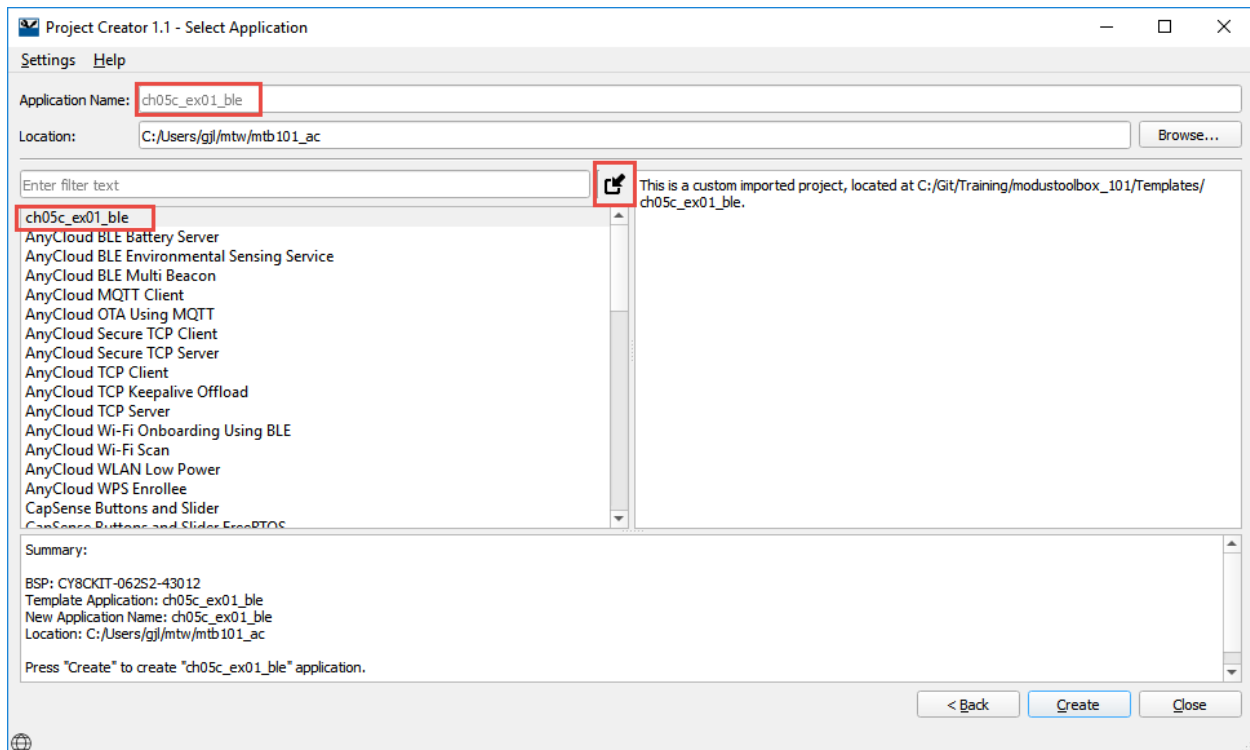
- AnyCloud BLE Multi Beacon
- AnyCloud BLE Find Me Profile
- AnyCloud BLE Battery Server
- AnyCloud BLE Environmental Sensing Service

You will have a chance to experiment with some of these in the exercises, but as a starting point, we will create a brand-new minimal BLE peripheral from an almost empty template application.

### 5c.4.1 Application Creation

For this example, I am going to build a BLE application called "ch05c\_ex01\_ble" with one custom service called the "Modus101" Service and one writable Characteristic called "LED". When the Client writes a 0 or 1 (strictly any non-zero value) into that Characteristic, my application firmware will just write that value into the GPIO driving the LED.

First, I'll create a new project for the CY8CKIT-062S2-43012 kit and I'll use the **Import...** button to import from the **ch05c\_ex01\_ble** template.



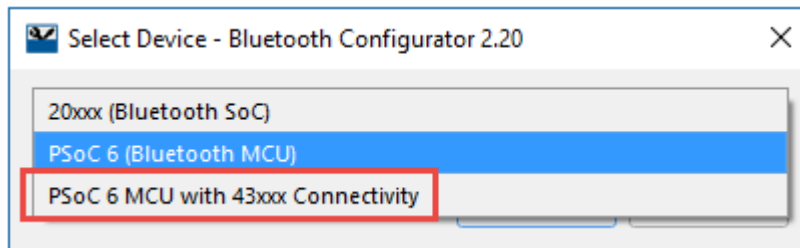
## 5c.4.2 Bluetooth Configurator

The GATT configuration defines the attributes - services, characteristics, and descriptors - of your BLE device. It can be tricky to get everything just right, so there is a tool called the Bluetooth Configurator to simplify the process.

If your project already has a .cybt file, you can open the Bluetooth Configurator by double clicking on it from inside Eclipse, or you can launch it from the Quick Panel. If you don't already have a .cybt file, you can create a new one by running the Bluetooth Configurator from the application's folder from the command line. The command is:

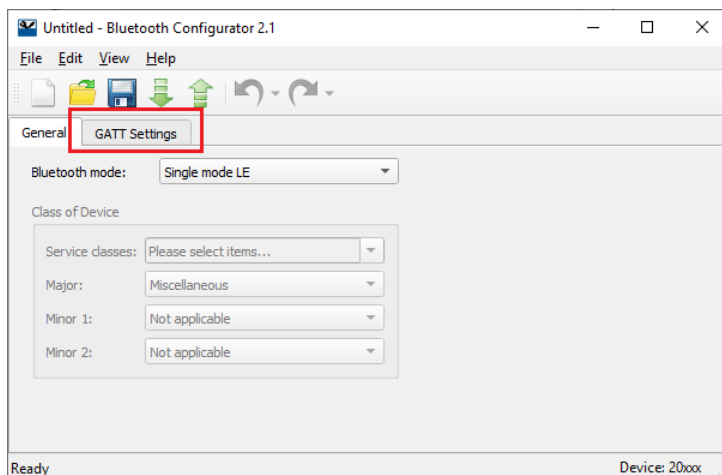
```
make config_bt
```

When the tool opens, use **File > New** and select "PSoC 6 MCU with 43xxx Connectivity" as the device family. If you don't see that listed, choose "20xxx".

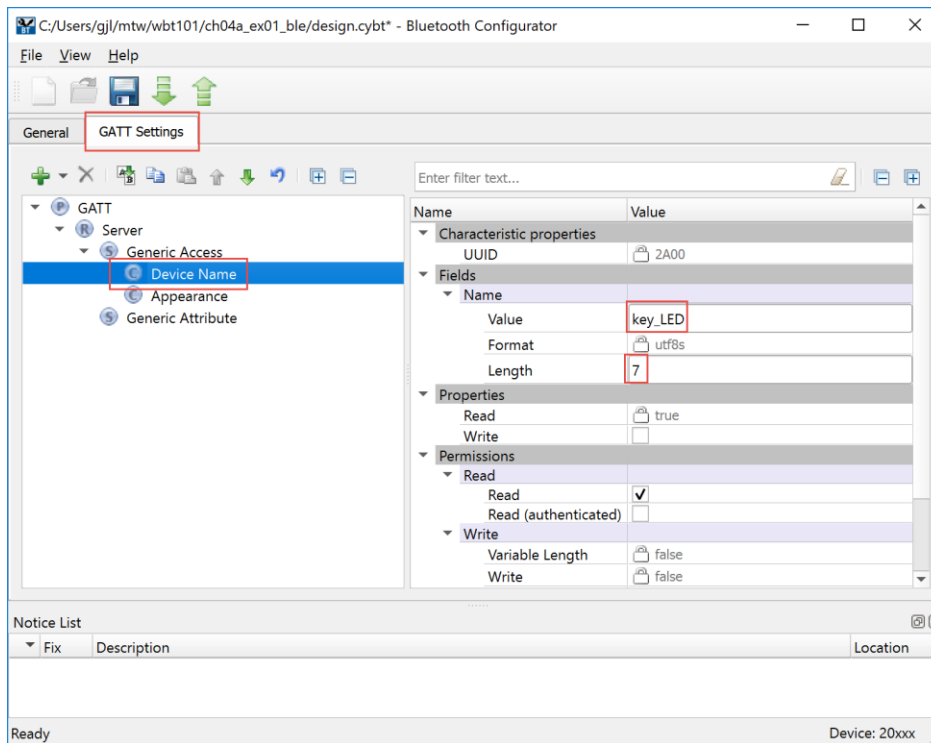


After setting up your configuration (more on that in a minute), save the file to the application's folder (unfortunately the save location does NOT default to the folder that you ran it from). The name doesn't matter as long as the extension is .cybt (I usually call it design.cybt while the code examples usually use cycfg\_bt.cybt). An application can only have one file with the extension .cybt.

Once the configurator is open, you will see default General and GATT Settings. We will leave the General settings alone, so switch to the GATT Settings tab.



You need to give your device a name and this is done by clicking on the *Device Name* field and typing into the *Value* text box. The name is just a string (format “utf8s” per the BLE spec). You must press **Enter** to get the tool to calculate the length for you.

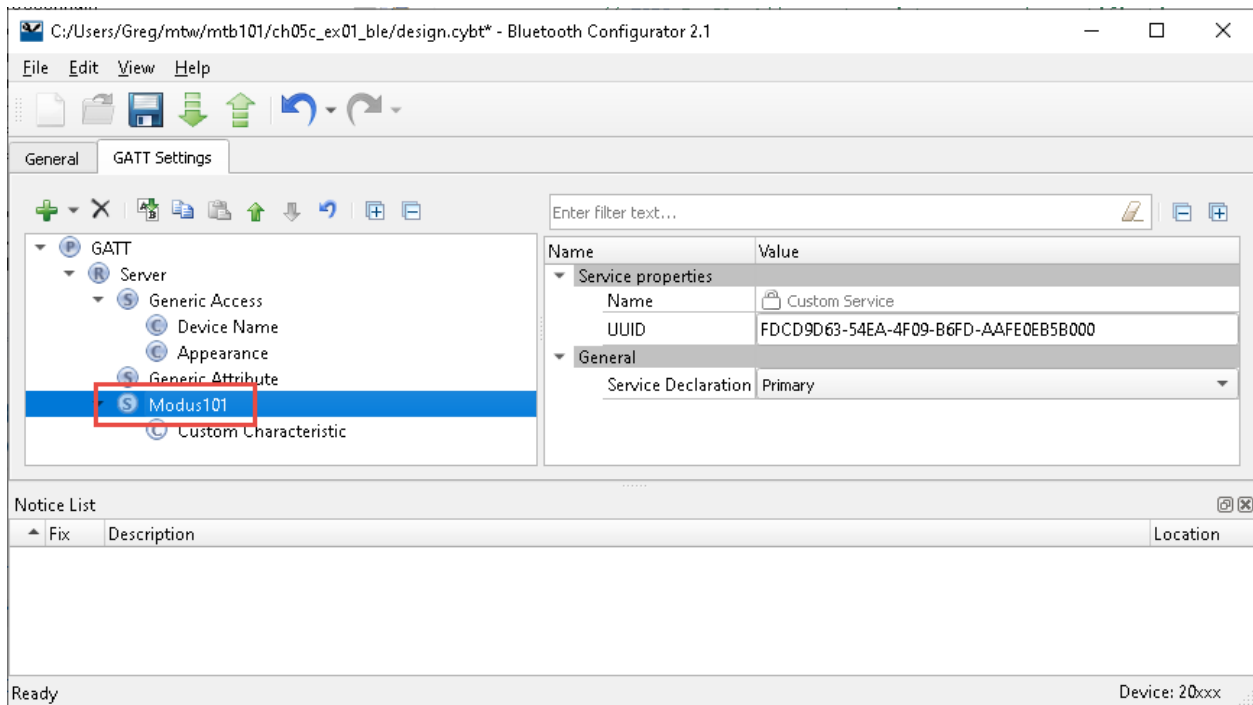


It is important that the name you choose is unique or you will not be able to identify your device when making connections from your cell phone. In this case, I've called the device *key\_LED*. When you do this yourself, use a unique device name such as <inits> LED where <inits> is your initials.

Make sure you press the "enter" key after typing in the name. This will calculate the string length and will put it in the Length field.

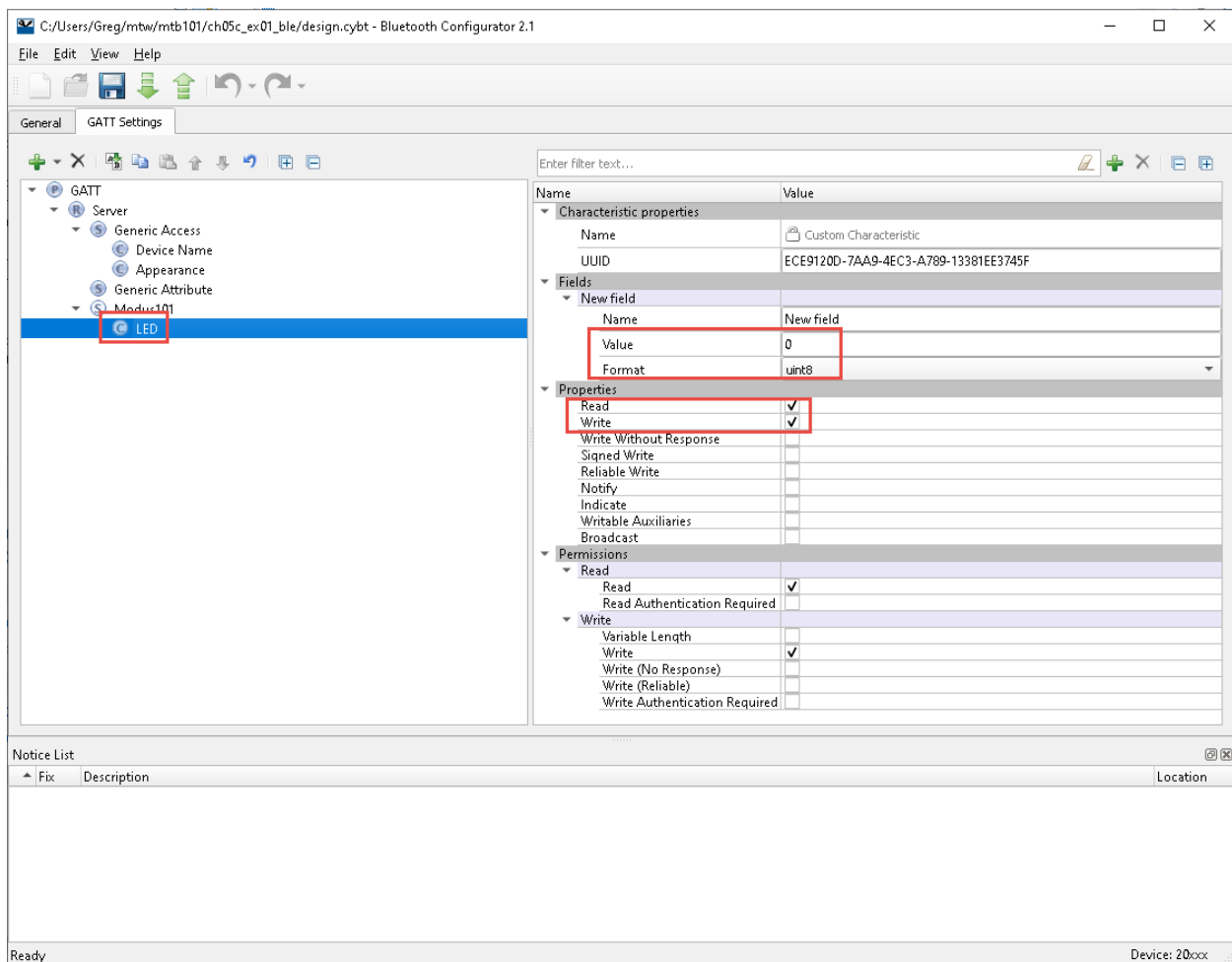
The next step is to set up a Service. To do this:

1. Select *Server* in the GATT database.
2. Right-click and choose *Add Service*, then select *Custom Service* (it is near the bottom of the list). A *Custom Service* entry appears in the GATT database.
3. Right-click on the custom service and select *Rename*. Call the service “Modus101”.
4. The tool will choose a random UUID for this Service, but you could specify your own UUID if desired. For this exercise, just keep the random UUID.



The Service includes a Characteristic, which we are going to use to control the LED. To do this you:

1. Right-click on *Custom Characteristic* under the *Modus101* Service and *Rename* to “LED”.
2. Under Fields, optionally provide a name. This name is not used and can be left as-is.
3. Change the format from utf8s (which requires a length) to uint8 (which has a length of 1 by definition).
4. Set the value of the LED characteristic to 0, which we will take to mean “OFF”. This will be the initial value.
5. We want the client to be able to Read and Write this Characteristic, so under *Properties*, enable *Read* and *Write*. Note that the tool makes the corresponding changes to the *Permissions* section for you, so you don't need to set them unless you need an unusual combination of Properties and Permissions.
6. Again, keep the randomly assigned UUID for the Characteristic just like you did for the Service UUID.



7. Click the **Save** button to save the file design.cybt. Again, the name of the file doesn't matter as long as the extension is cybt so you may see different names used in other applications.



8. Saving will create a GeneratedSource directory with the code generated based on your selections. You should not modify the generated code by hand – any changes should be done by re-running the Bluetooth Configurator.

### 5c.4.3 Editing the Firmware

The template includes a little bit of setup code for the BTM\_ENABLED\_EVT and some very helpful functions, as follows.

- app\_bt\_management\_callback() is the callback function that you edited in chapter 2. The BTM\_ENABLED\_EVT code sets and prints the Bluetooth Device Address (BDA), sets up the GATT database, and starts advertising for a connection.
- app\_gatt\_callback() handles GATT events such as connect/disconnect and attribute read/write requests.
- app\_set\_advertisement\_data() creates the advertising packet that includes the device name you will see in the CySmart app.
- app\_gatt\_get\_value() searches the GATT database for the requested characteristic and extracts the value. We use this function to read the state of the LED.
- app\_gatt\_set\_value() searches the GATT database for the requested characteristic and updates the value. We use this function to write the state of the LED into the database and, later, notify the central device.

1. Start by opening main.c and adding the include for the generated database, as follows:

```
#include "cycfg_gatt_db.h"
```

2. Template code for the **BTM\_ENABLED\_EVT** case in app\_bt\_management\_callback() sets the 6-byte Bluetooth Device Address and then prints it to the terminal when the stack is enabled. The address must be unique to avoid collisions with other devices. However, it does mean that your device's BDA will change every time the kit is reprogrammed or reset. In the future, the library will be updated so that the default BDA for each kit is unique.

If you want to have a fixed BDA for your kit, you can remove the random number generation and instead set it to any value you wish, but make sure it doesn't collide with any other student's address. I would suggest using the ASCII values for your initials for 3 or 4 of the bytes.

3. In the **BTM\_ENABLED\_EVT** case, add the following lines to set up the GATT database according to your selections in the Configurator:

```
/* Register GATT callback */  
wiced_bt_gatt_register( app_gatt_callback );  
  
/* Initialize the GATT database*/  
wiced_bt_gatt_db_init( gatt_database, gatt_database_len, NULL );
```

- Next, I don't want to allow pairing to the device just yet so configure the pairing mode with the parameters set to WICED\_FALSE:

```
/* Disable pairing */
wiced_bt_set_pairable_mode( WICED_FALSE, WICED_FALSE );
```

The above will allow you to connect to your device and open the GATT database.

The following edits enable the device to respond to GATT read and write requests.

- Add the following case in `app_gatt_get_value()` to print the state of the LED to the UART (the switch statement is already in the template – you just need to add a new case). This event will occur whenever the Central reads the LED characteristic. Note that the code uses the GATT database value, not the state of the pin itself, and so non-zero implies “on” and zero means “off”. The name of the value array is `app_modus101_led`. It can be found in the GeneratedSource file `cycfg_gatt_db.c` which we will talk about it a minute.

```
// TODO Ex 01: Add code for any action required when this attribute is read
switch ( attr_handle )
{
    case HDLC_MODUS101_LED_VALUE:
        printf("LED is %s\n", app_modus101_led[0] ? "ON":"OFF");
        break;
}
```

- In `app_gatt_set_value()`, notice how the template function automatically updates the GATT database with a call to `memcpy()`. There is no need to write to the `app_modus101_led` array.

```
// Value fits within the supplied buffer; copy over the value
app_gatt_db_ext_attr_tbl[i].cur_len = len;
memcpy(app_gatt_db_ext_attr_tbl[i].p_data, p_val, len);
res = WICED_BT_GATT_SUCCESS;
```

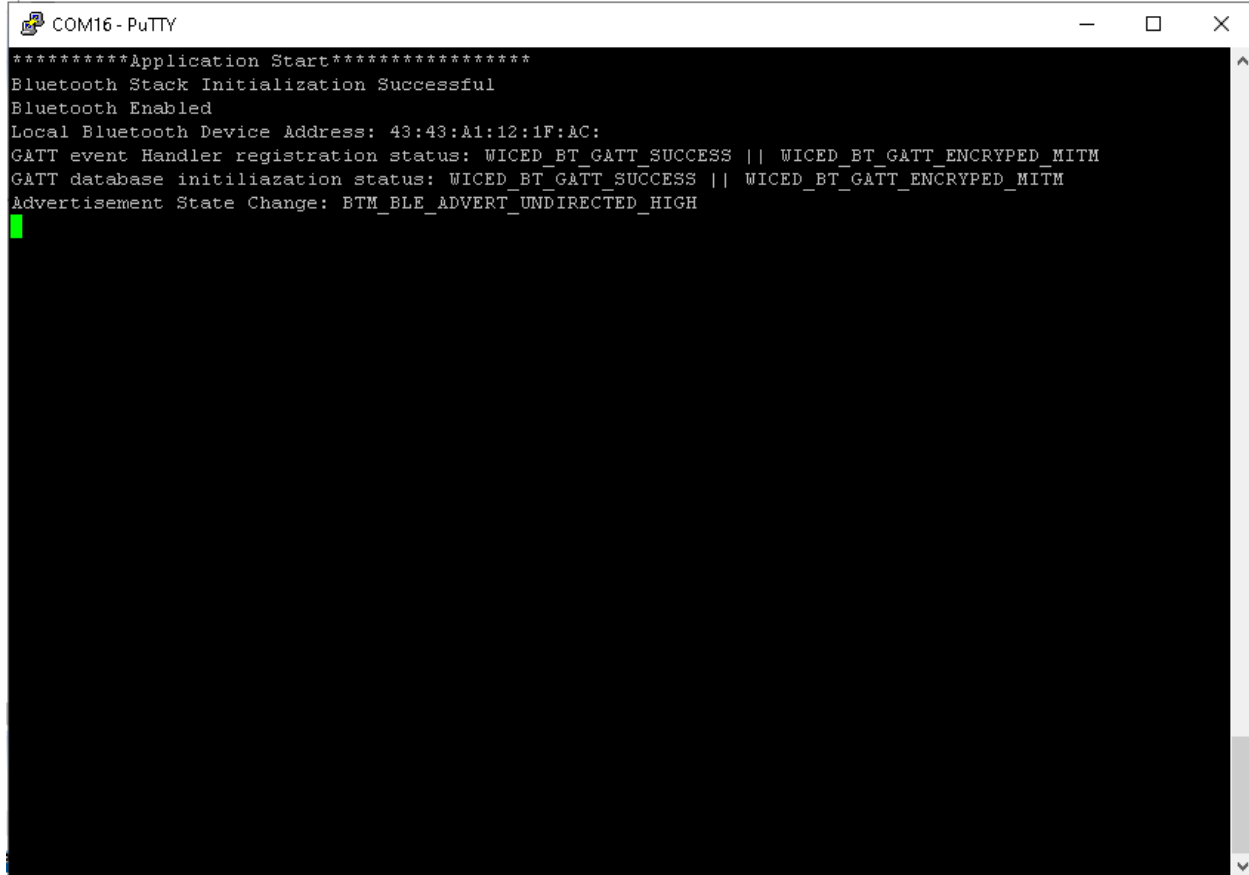
- Add the following case in `app_gatt_set_value()` to update the LED and printout the result. Again, the switch statement is in the template; just add the new case. This event will occur whenever the Central writes the LED characteristic. We are going to use `LED_2` for this example. Note that the LEDs on the kit are active low so the pin is set to the NOT of the value.

```
// TODO Ex01: Add code for any action required when this attribute is written
// For example, you may need to write the value into EERPOM if it needs to be
persistent
switch ( attr_handle )
{
    case HDLC_MODUS101_LED_VALUE:
        cyhal_gpio_write( CYBSP_USER_LED, app_modus101_led[0] == 0 );
        printf("Turn the LED %s\n", app_modus101_led[0] ? "ON":"OFF");
        break;
}
```

- Build and program the kit.

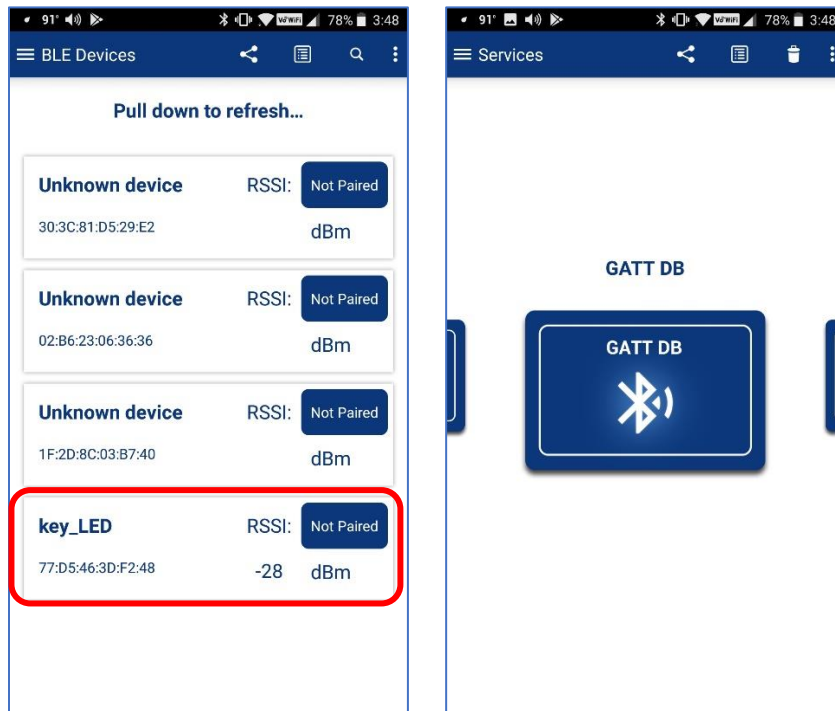
#### 5c.4.4 Testing the Application

Start up a UART terminal (115200, 8, 1, N). Make sure you enable the setting to automatically generate a CR for every LF (in Putty this is under Terminal > Implicit CR in every LF). Then build and program your kit. When the application firmware starts up you see some messages.



```
*****Application Start*****
Bluetooth Stack Initialization Successful
Bluetooth Enabled
Local Bluetooth Device Address: 43:43:A1:12:1F:AC:
GATT event Handler registration status: WICED_BT_GATT_SUCCESS || WICED_BT_GATT_ENCRYPED_MITM
GATT database initialization status: WICED_BT_GATT_SUCCESS || WICED_BT_GATT_ENCRYPED_MITM
Advertisement State Change: BTM_BLE_ADVERT_UNDIRECTED_HIGH
```

Run CySmart on your phone. When you see the "<inits>\_LED" device, tap on it. CySmart will connect to the device and will show the GATT browser widget.



On the terminal window, you will see that there has been a connection and the advertising has stopped.

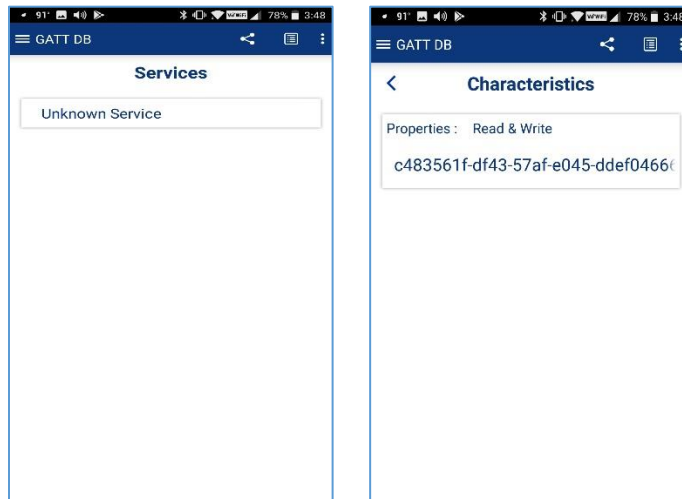
```

COM16 - PuTTY
*****Application Start*****
Bluetooth Stack Initialization Successful
Bluetooth Enabled
Local Bluetooth Device Address: 43:43:A1:12:1F:AC:
GATT event Handler registration status: WICED_BT_GATT_SUCCESS || WICED_BT_GATT_ENCRYPED_MITM
GATT database initialization status: WICED_BT_GATT_SUCCESS || WICED_BT_GATT_ENCRYPED_MITM
Advertisement State Change: BTM_BLE_ADVERT_UNDIRECTED_HIGH
Advertisement State Change: BTM_BLE_ADVERT_UNDIRECTED_LOW
GATT_CONNECTION_STATUS_EVT: Connect BDA 5B:D3:BE:49:D6:79:Connection ID 32768
Advertisement State Change: BTM_BLE_ADVERT_OFF
Unhandled Bluetooth Management Event: 0x20 BTM_BLE_CONNECTION_PARAM_UPDATE

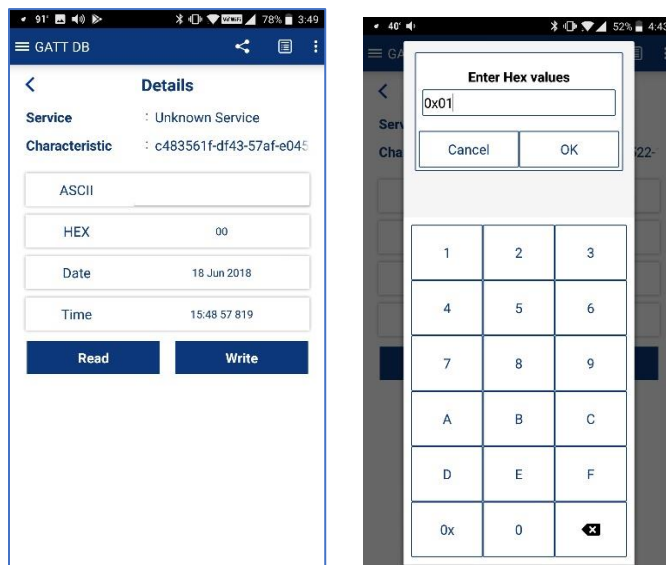
```

Back in CySmart, tap on the GATT DB widget to open the browser. You will see an Unknown Service (which I know is Modus101). Tap on the Service and CySmart will tell you that there is a Characteristic with the UUID shown (which I know is LED).

**Note:** In the iOS version of CySmart, the Characteristic UUID will not be shown – it will just say "Unknown Characteristic".



Tap on the Characteristic to see details about it. First, tap the Read button and you will see that the current value is 0. Now you can Write Hex values of 0x01 or 0x00 into the Characteristic and you will find that the LED turns on and off accordingly.



Finally press back until CySmart disconnects. When that happens, you will see the disconnect message in the terminal window and advertising will restart.

## 5c.5 Exercises (Part 1)

### 5c.5.1 Exercise 1: Basic BLE Peripheral



Follow the steps in section [5c.4 Demo Walkthrough](#) to create a basic BLE peripheral that allows you to control the LED on the kit.

## 5c.6 GATT Database Implementation

The Bluetooth Configurator automatically creates a GATT Database implementation to serve as a starting point. The database is split between `cycfg_gatt_db.c` and `cycfg_gatt_db.h`.

Even though the Bluetooth Configurator will create all of this for you, some understanding of how it is constructed is worthwhile knowing. The implementation is generic and will work for most situations, however you can make changes to handle custom situations.

When the stack has started (i.e. in the `BTM_ENABLED_EVT` callback), you need to provide a GATT callback function by calling `wiced_bt_gatt_register` and initialize the GATT database by calling `wiced_bt_gatt_db_init`. The latter takes a pointer to the GATT DB definition and its length. This allows the stack to directly access your GATT DB for some purposes.

The GATT DB is used by both the Stack and by your application firmware. The Stack will directly access the Handles, UUIDs and Permissions of the Attributes to process some of the Bluetooth Events. Mainly the Stack will verify that a Handle exists and that the Client has Permission to access it before it gives your application a callback.

Your application firmware will use the GATT DB to read and write data in response to WICED BT Events.

The WICED BTSTACK implementation of the GATT Database is simple generic "C" (obviously) and is composed logically of four parts. The first three are in `cycfg_gatt_db.c` while the last is implemented in the application code (in `main.c` in the template).

- An Array, named `gatt_database`, of `uint8_t` bytes that holds the Handles, Types and Permissions.
- An Array of Structs, named `app_gatt_db_ext_attr_tbl`, which holds Handles, a Maximum and Current Length and a Pointer to the actual Value.
- The Values as arrays of `uint8_t` bytes.
- Functions that serve as the API

### 5c.6.1 `gatt_database[]`

The `gatt_database` is just an array of bytes with special meaning.

To create the bytes representing an Attribute there is a set of C-preprocessor macros that "do the right thing". To create Services, use the macros:

- `PRIMARY_SERVICE_UUID16(handle, service)`
- `PRIMARY_SERVICE_UUID128(handle, service)`
- `SECONDARY_SERVICE_UUID16(handle, service)`

- SECONDARY\_SERVICE\_UUID128(handle, service)
- INCLUDE\_SERVICE\_UUID16(handle, service\_handle, end\_group\_handle, service)
- INCLUDE\_SERVICE\_UUID128(handle, service\_handle, end\_group\_handle)

The handle parameter is just the Service Handle, which is a 16-bit number. The Bluetooth Configurator will automatically create Handles for you that will end up in the `cycfg_gatt_db.h` file. For example:

```
/* Service Generic Access */
#define HDLS_GAP 0x0001u
/* Service Generic Attribute */
#define HDLS_GATT 0x0006u
/* Service Modus101 */
#define HDLS_MODUS101 0x0007u
```

The Service parameter is the UUID of the service, just an array of bytes. The Bluetooth Configurator will create them for you in `cycfg_gatt_db.h`. For example:

```
#define __UUID_SERVICE_MODUS101 0xD5u, 0x8Eu, 0x79u, 0x8Bu, 0x2Cu, 0xDEu, 0x11u,
0x89u, 0x45u, 0x47u, 0x5Au, 0x31u, 0x6Au, 0xA3u, 0xFAu, 0x34u
```

In addition, there are a bunch of predefined UUIDs in `wiced_bt_uuid.h`.

To create Characteristics, use the following C-preprocessor macros which are defined in `wiced_bt_gatt.h`:

- CHARACTERISTIC\_UUID16(handle, handle\_value, uuid, properties, permission)
- CHARACTERISTIC\_UUID128(handle, handle\_value, uuid, properties, permission)
- CHARACTERISTIC\_UUID16\_WRITABLE(handle, handle\_value, uuid, properties, permission)
- CHARACTERISTIC\_UUID128\_WRITABLE(handle, handle\_value, uuid, properties, permission)

As before, the handle parameter is just the 16-bit number that the Bluetooth Configurator creates for the Characteristics which will be in the form of `#define HDLC_` for example:

```
/* Characteristic LED */
#define HDLC_MODUS101_LED 0x0008u
#define HDLC_MODUS101_LED_VALUE 0x0009u
```

The `_VALUE` parameter is the Handle of the Attribute that will hold the Characteristic's Value. That is, a Characteristic has (at least) two attributes: one to declare the Characteristic and one to hold its value. When you want to read/write the Characteristic, you have to use the handle for the Attribute containing the value, not the declaration.

The UUIDs are 16-bits or 128-bits in an array of bytes. The Bluetooth Configurator will create `#defines` for the UUIDs in the file `cycfg_gatt_db.h`.

Properties is a bit mask which sets the properties (i.e. Read, Write etc.) The bit mask is defined in `wiced_bt_gatt.h`:

```
/* GATT Characteristic Properties */
#define LEGATTDB_CHAR_PROP_BROADCAST (0x1 << 0)
#define LEGATTDB_CHAR_PROP_READ (0x1 << 1)
#define LEGATTDB_CHAR_PROP_WRITE_NO_RESPONSE (0x1 << 2)
#define LEGATTDB_CHAR_PROP_WRITE (0x1 << 3)
```



```
#define LEGATTDB_CHAR_PROP_NOTIFY (0x1 << 4)
#define LEGATTDB_CHAR_PROP_INDICATE (0x1 << 5)
#define LEGATTDB_CHAR_PROP_AUTHD_WRITES (0x1 << 6)
#define LEGATTDB_CHAR_PROP_EXTENDED (0x1 << 7)
```

The Permission field is just a bit mask that sets the Permission of an Attribute (remember Permissions are on a per Attribute basis and Properties are on a per Characteristic basis). They are also defined in `wiced_bt_gatt.h`.

```
/* The permission bits (see Vol 3, Part F, 3.3.1.1) */
#define LEGATTDB_PERM_NONE (0x00)
#define LEGATTDB_PERM_VARIABLE_LENGTH (0x1 << 0)
#define LEGATTDB_PERM_READABLE (0x1 << 1)
#define LEGATTDB_PERM_WRITE_CMD (0x1 << 2)
#define LEGATTDB_PERM_WRITE_REQ (0x1 << 3)
#define LEGATTDB_PERM_AUTH_READABLE (0x1 << 4)
#define LEGATTDB_PERM_RELIABLE_WRITE (0x1 << 5)
#define LEGATTDB_PERM_AUTH_WRITABLE (0x1 << 6)

#define LEGATTDB_PERM_WRITABLE (LEGATTDB_PERM_WRITE_CMD | LEGATTDB_PERM_WRITE_REQ |
LEGATTDB_PERM_AUTH_WRITABLE)
#define LEGATTDB_PERM_MASK (0x7f) /* All the
permission bits. */
#define LEGATTDB_PERM_SERVICE_UUID_128 (0x1 << 7)
```

## 5c.6.2 gatt\_db\_ext\_attr\_tbl

The `gatt_database` array does not contain the actual values of Attributes. To find the values there is an array of structures of type `gatt_db_lookup_table`. Each structure contains a handle, a max length, actual length and a pointer to the array where the value is stored.

```
// External Lookup Table Entry
typedef struct
{
    uint16_t handle;
    uint16_t max_len;
    uint16_t cur_len;
    uint8_t *p_data;
} gatt_db_lookup_table;
```

The Bluetooth Configurator will create this array for you automatically in `cycfg_gatt_db.c`:

```
/* *****
 * GATT Lookup Table
 * ***** */

gatt_db_lookup_table_t app_gatt_db_ext_attr_tbl[] =
{
    /* { attribute handle, maxlen, curlen, attribute data } */
    { HDLC_GAP_DEVICE_NAME_VALUE, 8, 8, app_gap_device_name },
    { HDLC_GAP_APPEARANCE_VALUE, 2, 2, app_gap_appearance },
    { HDLC_MODUS101_LED_VALUE, 1, 1, app_modus101_led },
};
```

The functions `app_gett_get_value` and `app_gatt_set_value` help you search through this array to find the pointer to the value.

### 5c.6.3 uint8\_t Arrays for the Values

Bluetooth Configurator will generate arrays of uint8\_t to hold the values of writable/readable Attributes. You will find these values in a section of the code in cycfg\_gatt\_db.c marked with a comment "GATT Initial Value Arrays". In the example below, you can see there is a Characteristic with the name of the device, a Characteristic with the GAP appearance, and the LED Characteristic. These are the array names that you will use in your firmware to access a GATT database value. In the simple peripheral example, we used app\_modus101\_led[0] when we needed to know the value for the LED characteristic.

```
/* *****  
 * GATT Initial Value Arrays  
 * ***** */  
  
uint8_t app_gap_device_name[] = {'k', 'e', 'y', '_', 'L', 'E', 'D', '\0', };  
uint8_t app_gap_appearance[] = {0x00u, 0x00u, };  
uint8_t app_modus101_led[] = {0x00u, };
```

One thing that you should be aware of is the endianness. Bluetooth uses little endian, which is the same as ARM processors.

### 5c.6.4 Application Programming Interface

There are two functions in our template which make up the interface to the GATT Database: app\_gatt\_get\_value and app\_gatt\_set\_value. Here are the function prototypes from the template code:

```
wiced_bt_gatt_status_t app_gatt_get_value( wiced_bt_gatt_attribute_request_t *p_attr );  
wiced_bt_gatt_status_t app_gatt_set_value( wiced_bt_gatt_attribute_request_t *p_attr );
```

These functions receive a pointer to the GATT attribute request structure. That structure contains, among other things, the attribute handle, a pointer to the value to be read/written, the length of the value to be written for writes, and a pointer to the length of the value received for reads.

Both functions loop through the GATT Database and look for an attribute handle that matches the input parameter. Then they memcpy the data into the right place, either saving it in the database, or writing into the buffer for the Stack to send back to the Client.

Both functions have a switch where you might put in custom code to do something based on the handle. This place is marked with //TODO: in the two functions.

You are supposed to return a wiced\_bt\_gatt\_status\_t which will tell the Stack what to do next. Assuming things work this function will return WICED\_BT\_GATT\_SUCCESS. In the case of a Write this will tell the Stack to send a WRITE Response indicating success to the Client.

## 5c.7 Notifications

In the previous example, I showed you how the GATT Client can Read and Write the GATT Database running on the GATT Server. But, 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 a button value changes. That leaves us with the obvious questions of how does the Server initiate communication to the Client, and when is it allowed to do so?

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 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 send a Notification or Indication:

- `wiced_bt_gatt_send_notification (conn_id, handle, length, value)`
- `wiced_bt_gatt_send_indication (conn_id, handle, length, value)`

By convention, the GATT Server will not send Notification or Indication messages unless they are turned on by the Client.

How do you turn on Notifications or Indications? In the last chapter, we talked about the GATT Attribute Database, specifically, the Characteristic. As stated previously, a Characteristic is composed of a minimum of two Attributes:

- Characteristic Declaration
- Characteristic Value

However, information about the Characteristic can be extended by adding more Attributes, which go by the name of Characteristic Descriptors.

For the Client to tell the Server that it wants to have Indications or Notifications, four things need to happen.

First, the Server must add a new Characteristic Descriptor Attribute called the Client Characteristic Configuration Descriptor, often called the CCCD. This Attribute 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.

To add the CCCD to your GATT DB use the following macro (note that Bluetooth Configurator generates this code for you in `cycfg_gatt_db.c`):

```
CHAR_DESCRIPTOR_UUID16_WRITABLE (
    <HANDLE>,
    UUID_DESCRIPTOR_CLIENT_CHARACTERISTIC_CONFIGURATION,
    LEGATTDDB_PERM_READABLE | LEGATTDDB_PERM_WRITE_REQ | LEGATTDDB_PERM_AUTH_WRITABLE ),
```

The permissions above indicate that the CCCD value is readable whenever connected but will only be writable if the connection is authenticated (more on that later). To see the other possible choices, right click on one of them from inside Eclipse IDE and select "Open Declaration".

Second, you must change the Properties for the Characteristic to specify that the characteristic allows notifications. That is done by adding LEGATTDB\_CHAR\_PROP\_NOTIFY to the Characteristic's Properties. To see all the available choices, right-click on one of the existing Properties in the Eclipse IDE and select "Open Declaration".

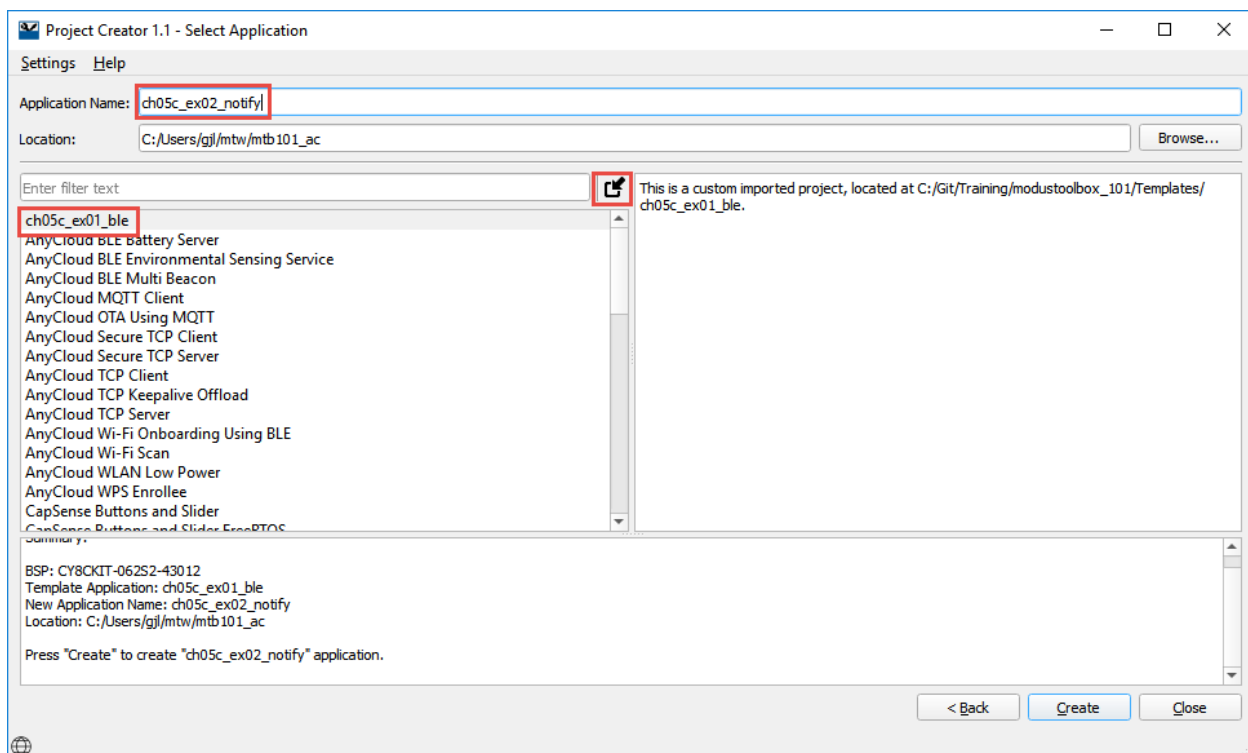
Third, in your GATT Attribute Write Callback you need to save the CCCD value that was written to you (note that this is done automatically in `app_gatt_set_value()` because the CCCD is writable).

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.

## 5c.8 Notification Demo Walkthrough

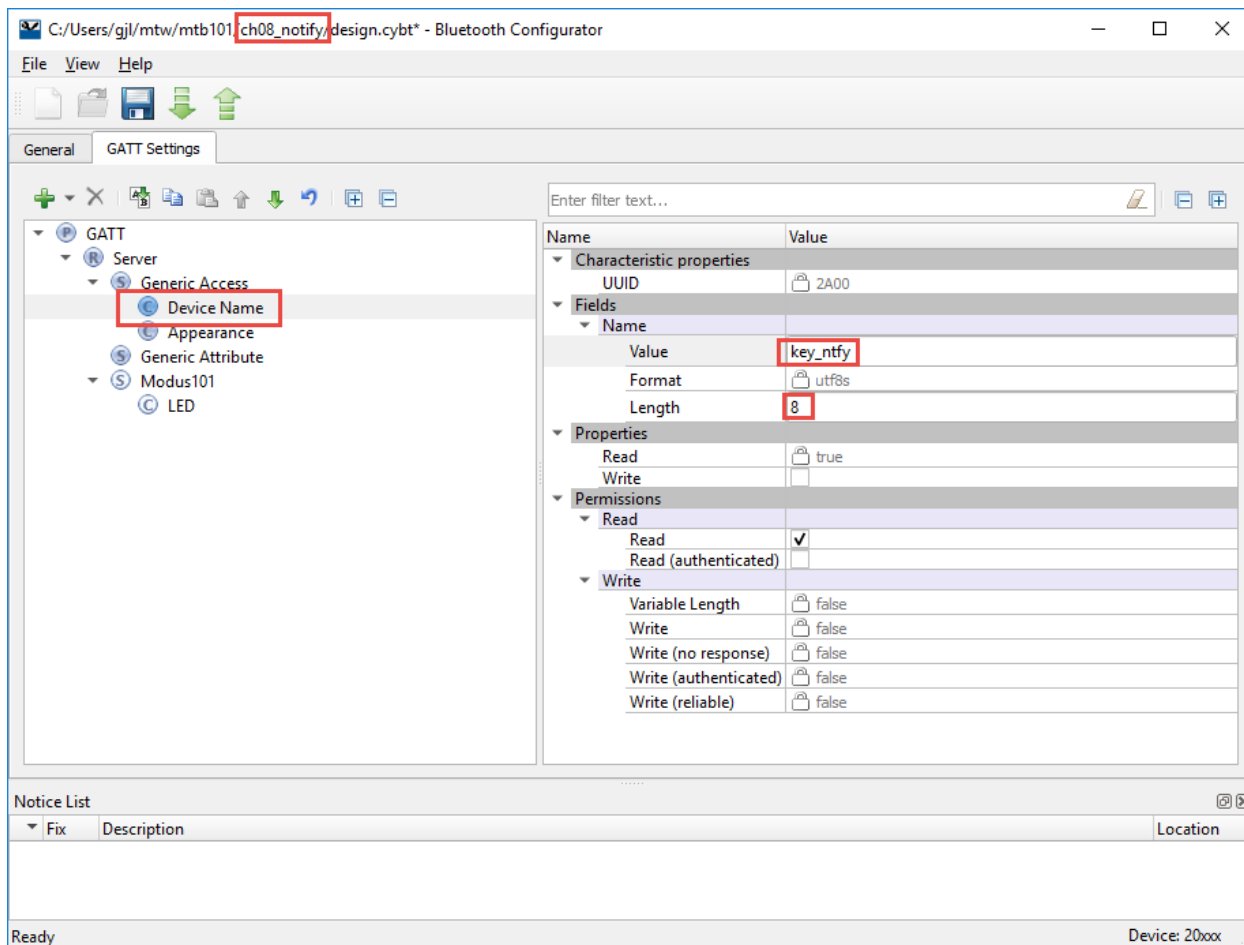
Next, we will add notifications to the previous exercise. We will add a new Characteristic called "Counter" that will count how many times the user button has been pressed since reset. That Characteristic will have Read and Notify properties set so that you can read the value or register to be notified any time the value changes.

The first thing I will do is import the previously completed exercise (not the template) into a new one using the Project Creator Import function and I'll call the new application **ch05c\_ex02\_notify**.

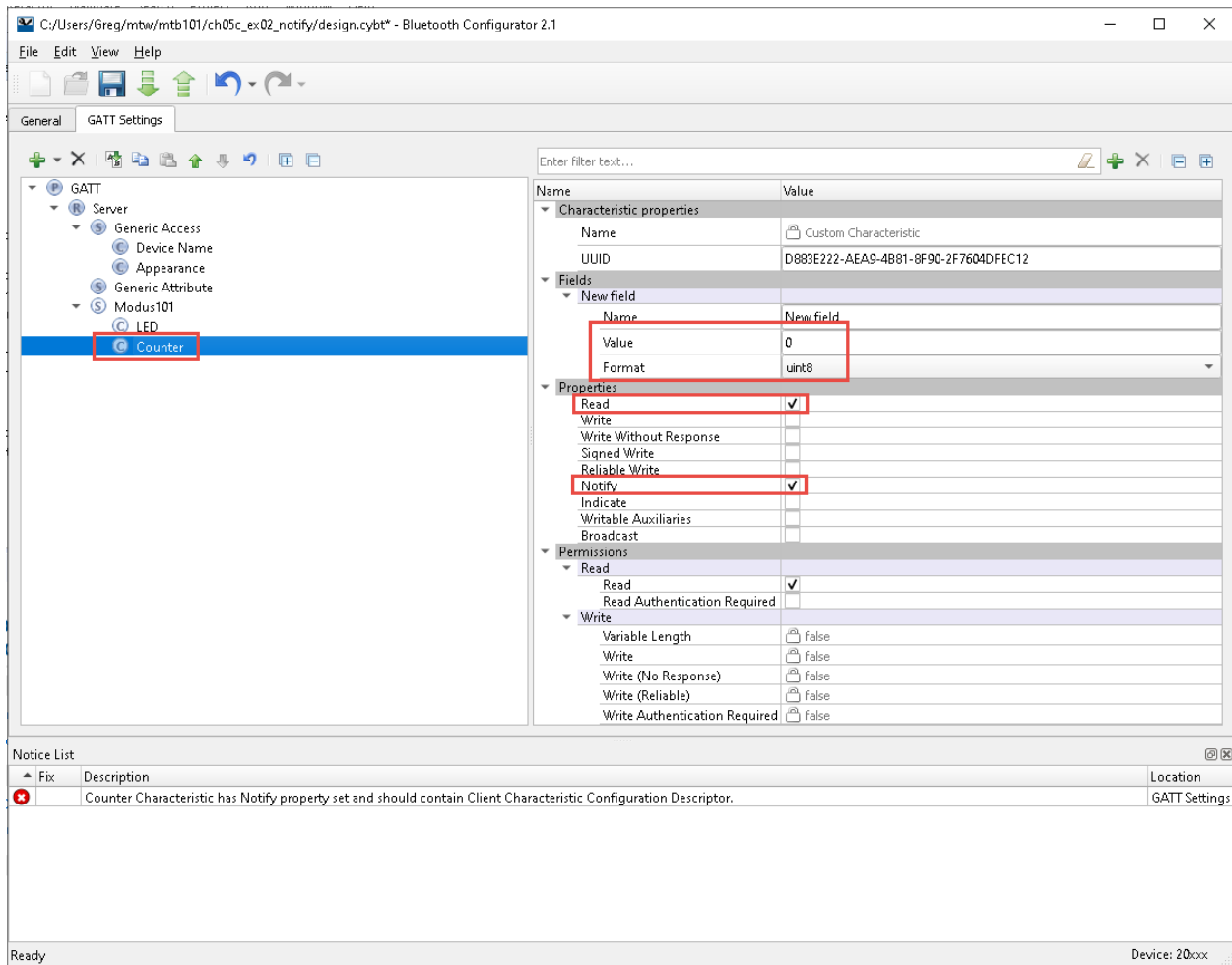


## 5c.8.1 Running the Bluetooth Configurator

Start the Bluetooth Configurator by clicking on the link in the Quick Panel or by double clicking on design.cybt in the new project. Make sure you open the one from the ch05c\_ex02\_notify project. In the Generic Access service change the Device Name. I'll use a device name of "key\_ntfy". **When you do this yourself, use a unique name such as <inits>\_ntfy where <inits> is your initials.** Otherwise you will have trouble finding your specific device among all the ones that are advertising.



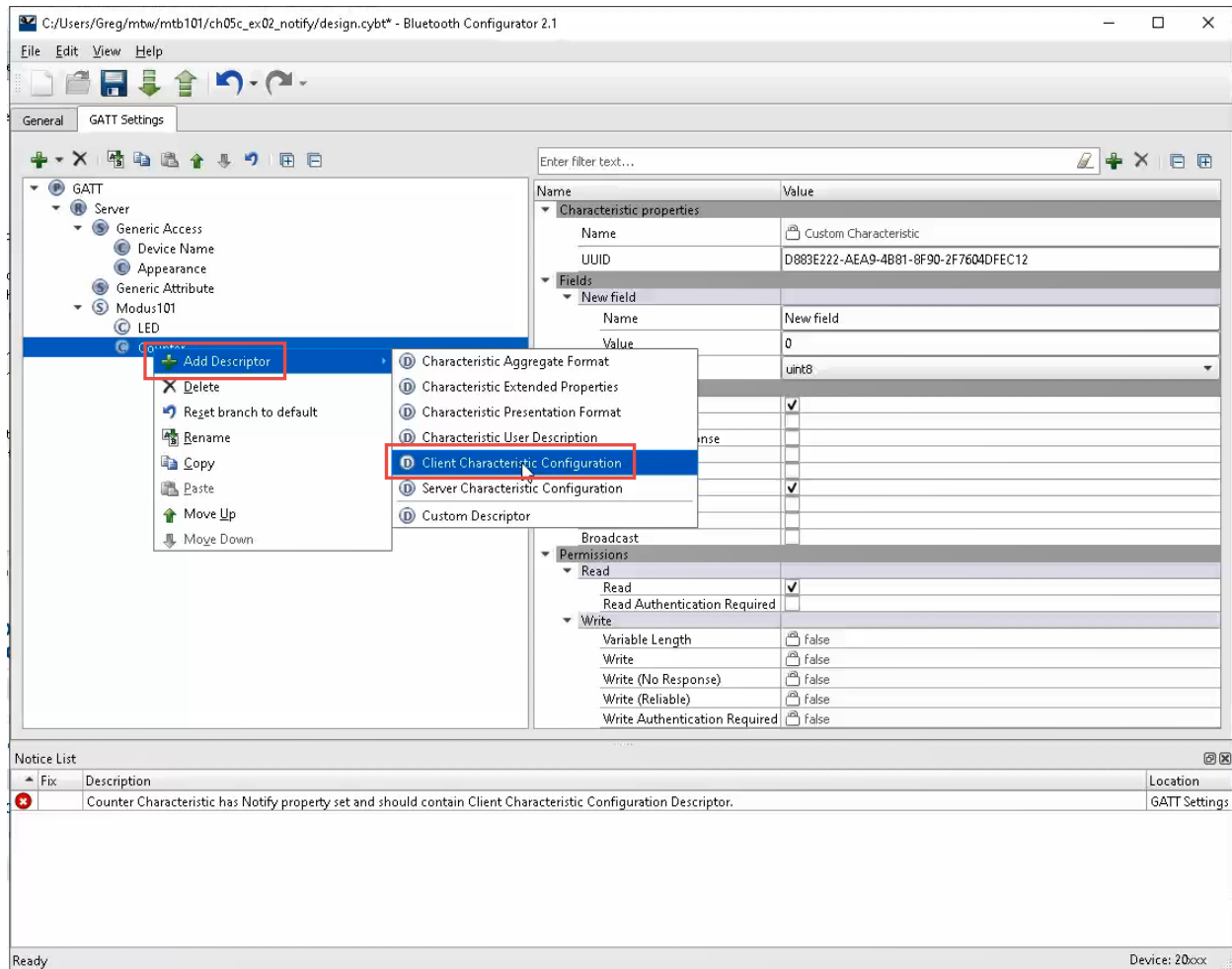
Add a new Custom Characteristic to the "Modus101" Service and rename it to "Counter", give it the type uint8 and initial value of 0.



Under Properties enable Read. Now look in the Permission section. It was set by the tool to Read based on our Properties selections. This means that we will be able to Read the Characteristic value without Pairing first. In real-world applications you would most likely also turn on Read (authenticated) so that Read will require an Authenticated (i.e. Paired) link but we shall handle pairing later.

Back under Properties, enable Notify so that the peripheral will be able to tell us when Counter value changes. Note that enabling notifications generates an error because you have not yet made notifications possible. The message tells you to add a CCCD (Client Characteristic Configuration Descriptor), which we will do next.

Add a CCCD by right-clicking on Counter, then **Add Descriptor**, and choose *Client Characteristic Configuration*.

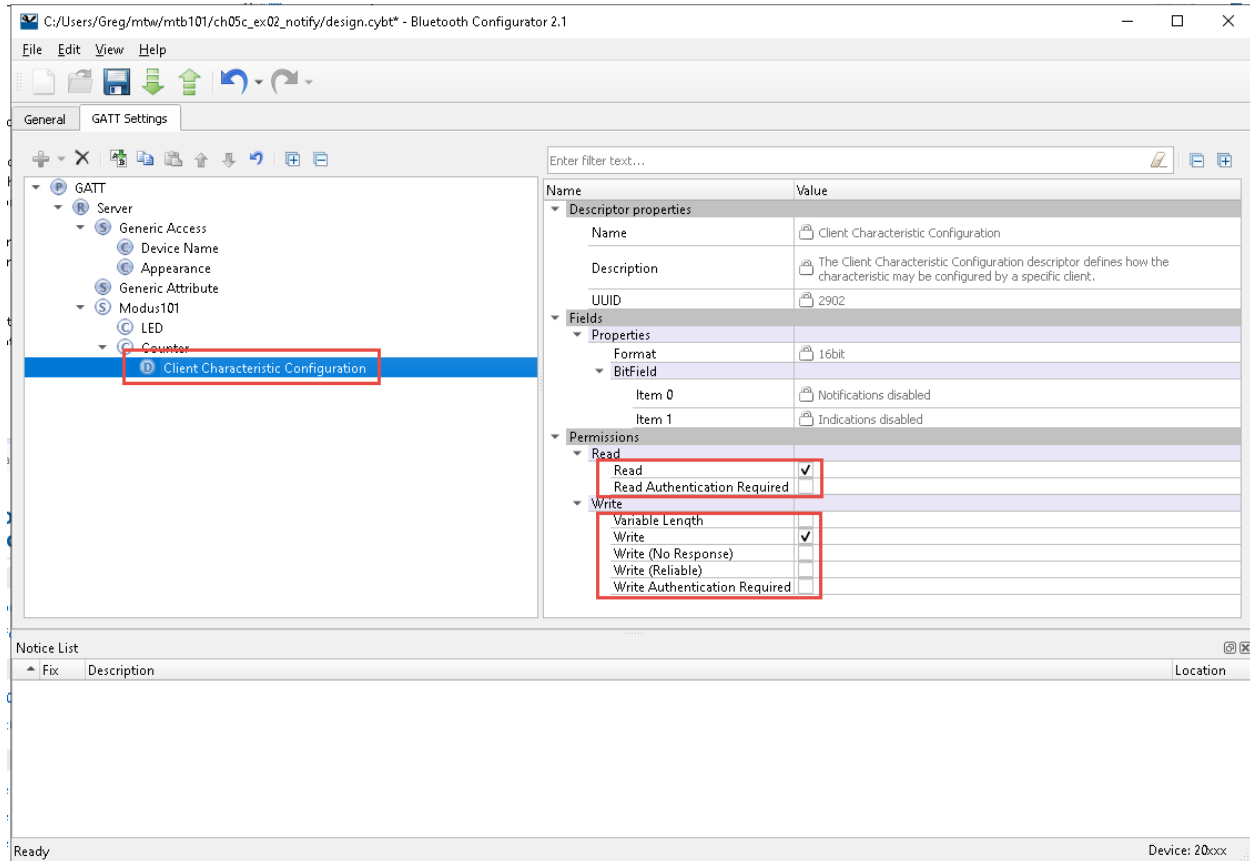


The new characteristic sets Notifications and Indications to disabled by default (and you cannot change that from the tool. If you want to enable them automatically, it is better practice to enable them during pairing).



Make sure Write permission is set on the CCCD so that you will be able to set it from CySmart. Note that the error message has gone away.

Make sure the Write (authenticated) permission check box is not set because we are not (yet) requiring the devices to pair before enabling notifications.



Finally, save your edits and close the Bluetooth Configurator.

## 5c.8.2 Editing the Firmware

In main.c, we need to make the following changes:

1. Declare a global variable called `connection_id`. Upon a GATT connection (i.e. in `app_gatt_callback`), save the connection ID. Upon a GATT disconnection, reset the connection ID. The ID is needed to send a notification. You need to tell it which connected device to send the notification to. In our case we only allow one connection at a time, but there are devices that allow multiple connections.

```
Global Variable:
uint16_t connection_id = 0;

GATT Connection:
/* Handle the connection */
connection_id = p_conn->conn_id;
GATT Disconnection:
/* Handle the disconnection */
connection_id = 0;
```

2. Declare a global variable called `CounterTaskHandle`. This will be the handle for a task we will create that will send notifications when the button is pressed. The task will be unlocked by the button ISR.

```
TaskHandle_t CounterTaskHandle = NULL;
```

3. Configure `BUTTON_1` as a falling edge interrupt under the `BTM_ENABLED_EVT`.

```
/* Configure CYBSP_USER_BUTTON for a falling edge interrupt */
cyhal_gpio_init(CYBSP_USER_BUTTON, CYHAL_GPIO_DIR_INPUT,
CYHAL_GPIO_DRIVE_PULLUP, CYBSP_BTN_OFF);
cyhal_gpio_register_callback(CYBSP_USER_BUTTON, button_cback, NULL);
cyhal_gpio_enable_event(CYBSP_USER_BUTTON, CYHAL_GPIO_IRQ_FALL, 3, true);
```

4. Create a function (and a declaration) for the button callback. In the callback we will just increment the Button Characteristic value and unlock the counter task.  
Note that the array `app_modus101_counter` was created by the Bluetooth Configurator. It holds the value for our counter characteristic. The name that the configurator uses is of the form: `app_<service_name>_<characteristic_name>`. The button callback function will look like this:

```
void button_cback(void *handler_arg, cyhal_gpio_irq_event_t event)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Increment button counter */
    app_modus101_counter[0]++;

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

```

5. Create a function (and a declaration) to send a notification. The function will wait to be unlocked by the button ISR and will send a notification if we have a connection and the notification is enabled. The function will look like this:

```

/* Counter task to send a notification */
static void counter_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_modus101_counter_client_char_config[0] &
                    GATT_CLIENT_CONFIG_NOTIFICATION )
                {
                    printf( "Notifying counter change (%d)\n",
                        app_modus101_counter[0] );
                    wiced_bt_gatt_send_notification(
                        connection_id,
                        HDLC_MODUS101_COUNTER_VALUE,
                        app_modus101_counter_len,
                        app_modus101_counter );
                }
            }
        }
        else
        {
            /* The call to ulTaskNotifyTake() timed out. */
        }
    }
}

```

6. Start the counter task in main before starting the scheduler.

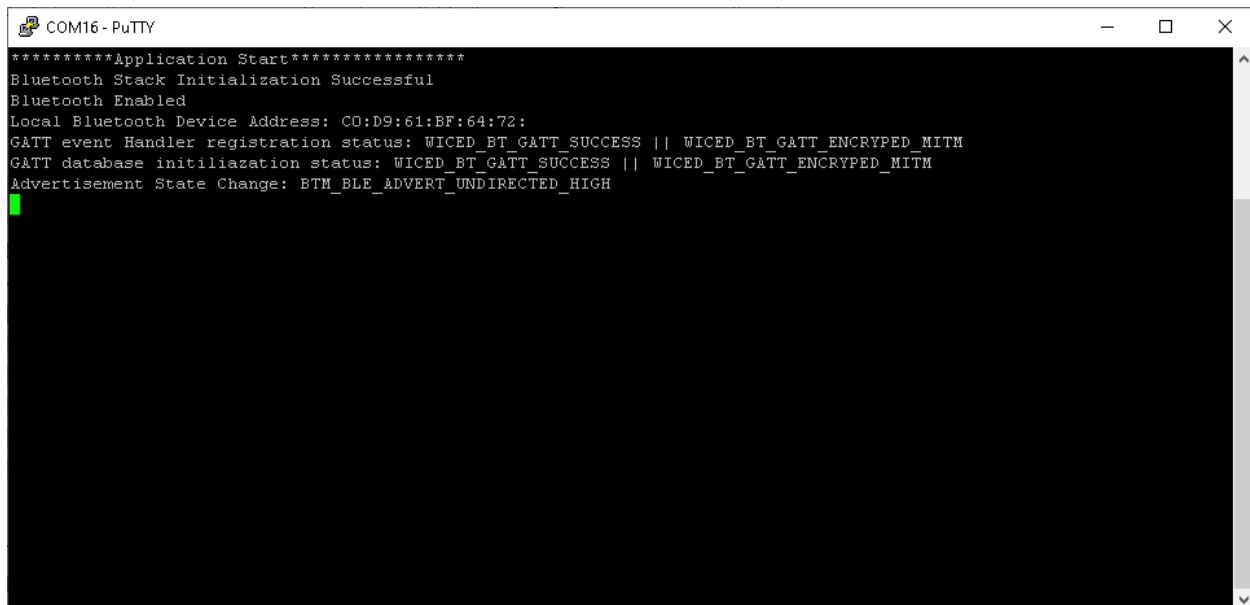
```
/* Start task to handle Counter notifications */
xTaskCreate (counter_task,
"CounterTask",
TASK_STACK_SIZE,
NULL,
TASK_PRIORITY,
&CounterTaskHandle);
```

7. Add a debug message to in app\_gatt\_set\_value() so you know when notifications get enabled/disabled. Note that the switch statement is already there but you need to add a new case.

```
switch( attr_handle )
{
    case HDLD_MODUS101_COUNTER_CLIENT_CHAR_CONFIG:
        printf("Setting notify (0x%02x, 0x%02x)\n", p_val[0], p_val[1]);
        break;
```

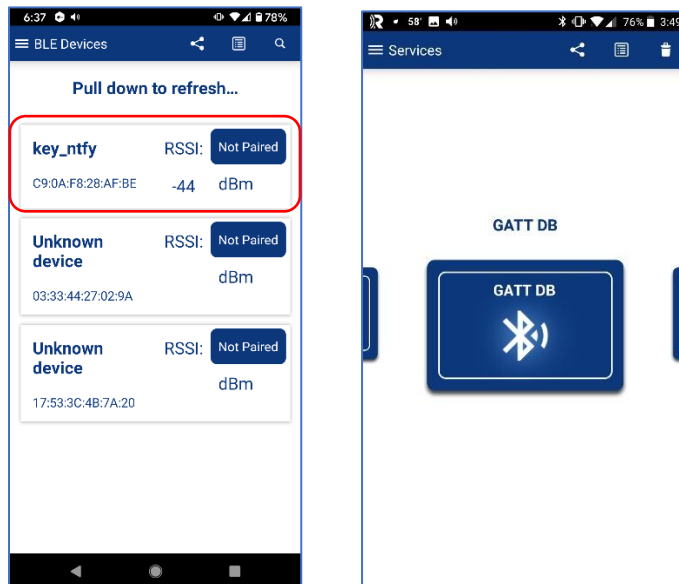
### 5c.8.3 Testing the Application

Start up a UART terminal to the Peripheral UART port with a baud of 115200 and then program the kit. When the firmware starts up you will see some messages.

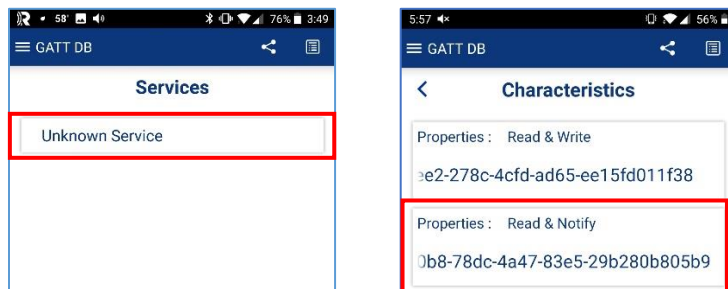


```
COM16 - PuTTY
*****Application Start*****
Bluetooth Stack Initialization Successful
Bluetooth Enabled
Local Bluetooth Device Address: CO:D9:61:BF:64:72:
GATT event Handler registration status: WICED_BT_GATT_SUCCESS || WICED_BT_GATT_ENCRYPED_MITM
GATT database initialization status: WICED_BT_GATT_SUCCESS || WICED_BT_GATT_ENCRYPED_MITM
Advertisement State Change: BTM_BLE_ADVERT_UNDIRECTED_HIGH
```

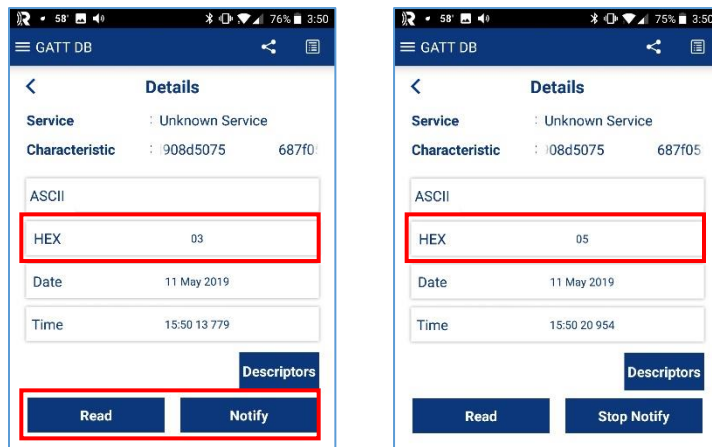
Run CySmart on your phone. When you see the "<init>\_ntfy" device, tap on it. CySmart will connect to the device and will show the GATT browser widget.



Tap on the GATT DB widget to open the browser. Then tap on the Unknown Service (which we know is Modus101) and then on the Characteristic that has Read and Notify Properties (which we know is Counter).



Tap the **Read** button to read the value. Press the user button on the kit a few times and then **Read** again to see the incremented value. Then tap the **Notify** button to enable notifications. Now each time you press the button the value is shown automatically.



You will see messages like this in the terminal emulator:

```
COM16 - PuTTY
*****Application Start*****
Bluetooth Stack Initialization Successful
Bluetooth Enabled
Local Bluetooth Device Address: C9:0A:F8:28:AF:BE:
GATT event Handler registration status: WICED_BT_GATT_SUCCESS || WICED_BT_GATT_ENCRYPED_MITM
GATT database initialization status: WICED_BT_GATT_SUCCESS || WICED_BT_GATT_ENCRYPED_MITM
Advertisement State Change: BTM_BLE_ADVERT_UNDIRECTED_HIGH
GATT_CONNECTION_STATUS_EVT: Connect BDA 6D:04:EF:F2:03:67:Connection ID 32768
Advertisement State Change: BTM_BLE_ADVERT_OFF
Unhandled Bluetooth Management Event: 0x20 BTM_BLE_CONNECTION_PARAM_UPDATE
Setting notify (0x01, 0x00)
Notifying counter change (3)
Notifying counter change (4)
Notifying counter change (5)
Notifying counter change (6)
Notifying counter change (7)
Notifying counter change (8)
Notifying counter change (9)
Notifying counter change (10)
Notifying counter change (11)
Notifying counter change (12)
Notifying counter change (13)
Setting notify (0x00, 0x00)
```

## 5c.9 Exercises (Part 2)

### 5c.9.1 Exercise 2: Notification



Follow the steps in section [5c.8 Notification Demo Walkthrough](#) to add a button press counter Characteristic that has Read and Notify Properties.

## 5c.10 Security

To securely communicate between two devices, you want to: (1) Authenticate that both sides know who they are talking to; (2) ensure that all access to data is Authorized, (3) Encrypt all message that are transmitted; (4) verify the Integrity of those messages; and (5) ensure that the Identity of each side is hidden from eavesdroppers.

In BLE, this entire security framework is built around AES-128 symmetric key encryption. This type of encryption works by combining a Shared Secret code and the unencrypted data (typically called plain text) to create an encrypted message (typically called cypher text).

- $CypherText = F(SharedSecret, PlainText)$

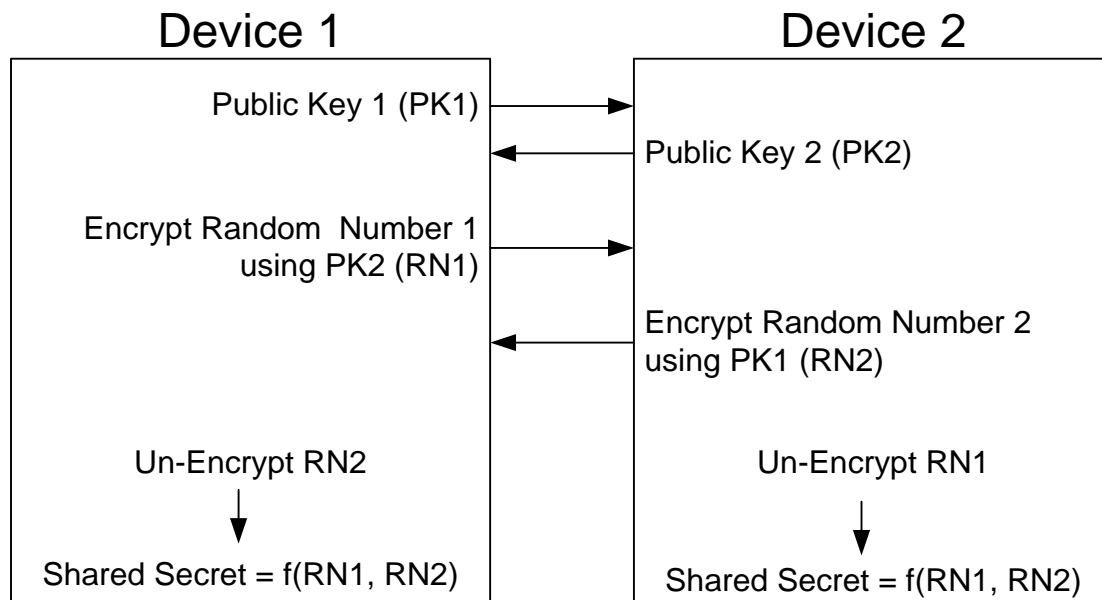
There is a bunch of math that goes into AES-128, but for all practical purposes if the Shared Secret code is kept secret, you can assume that it is very unlikely that someone can read the original message.

If this scheme depends on a Shared Secret, the next question is how do two devices that have never been connected get a Shared Secret that no one else can see? In BLE, the process for achieving this state is called Pairing. A device that is Paired is said to be Authenticated.



### 5c.10.1 Pairing

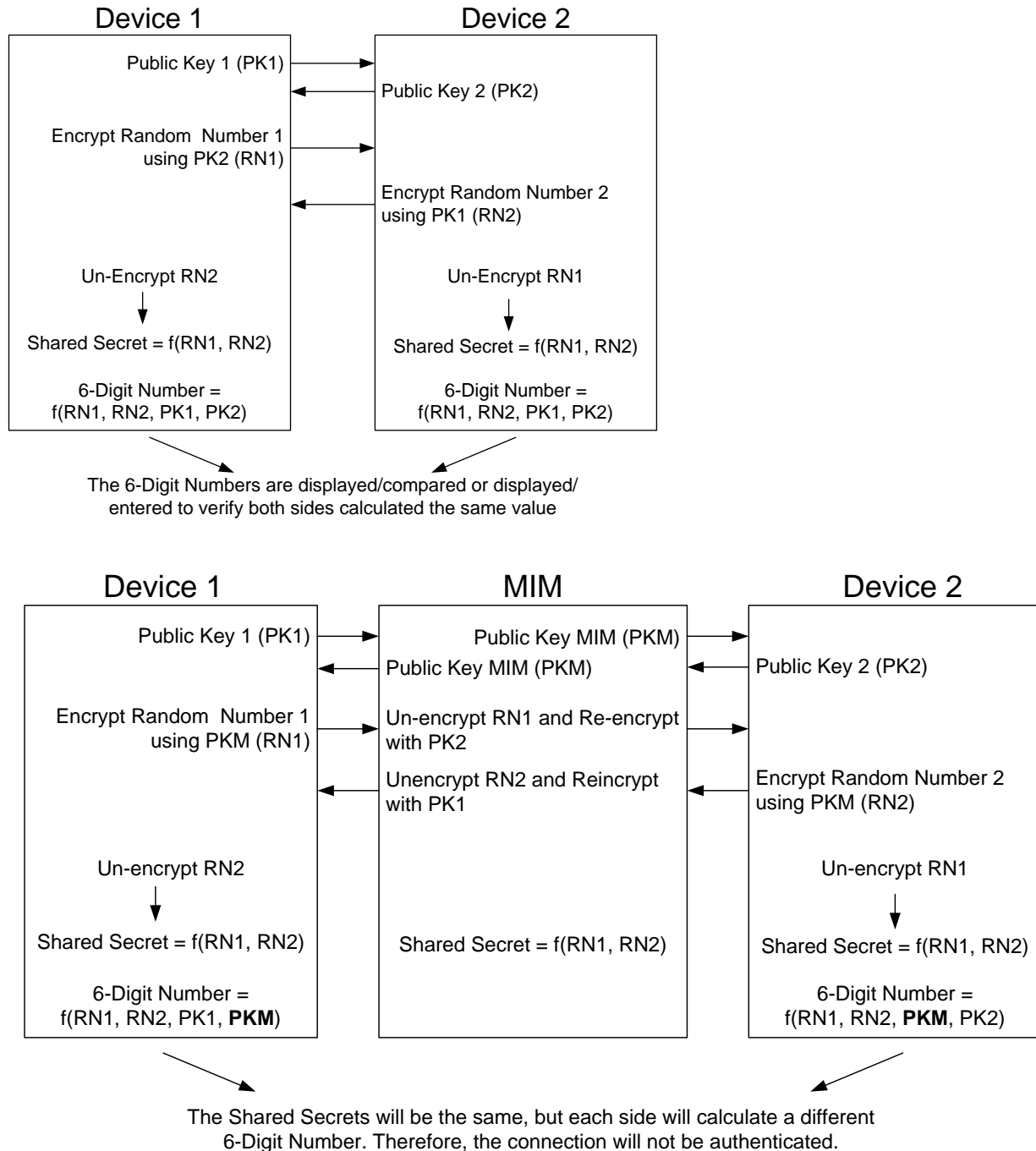
Pairing is the process of arriving at the Shared Secret. The basic problem continues to be how do you send a Shared Secret over the air, unencrypted and still have your Shared Secret be Secret. The answer is that you use public key encryption. Both sides have a public/private key pair that is either embedded in the device or calculated at startup. When you want to authenticate, both sides of the connection exchange public keys. Then both sides exchange encrypted random numbers that form the basis of the shared secret.



But how do you protect against Man-In-The-Middle (MIM)? There are four possible methods.

- Method 1 is called "Just works". In this mode you have no protection against MIM.
- Method 2 is called "Out of Band". Both sides of the connection need to be able to share the PIN via some other connection that is not Bluetooth such as NFC.
- Method 3 is called "Numeric Comparison" (V2.PH.7.2.1). In this method, both sides display a 6-digit number that is calculated with a nasty cryptographic function based on the random numbers used to generate the shared key and the public keys of each side. The user observes both devices. If the number is the same on both, then the user confirms on one or both sides. If there is a MITM, then the random numbers on both sides will be different so the 6-digit codes would not match.
- Method 4 is called "Passkey Entry" (V2.PH.7.2.3). For this method to work, at least one side needs to be able to enter a 6-digit Passkey. The other side must be able to display the Passkey. One device displays the Passkey and the user is required to enter the Passkey on the other device. Then an exchange and comparison process happen with the Passkeys being divided up, encrypted, exchanged and compared.

Pictorially, the process with no MIM and with MIM is shown below. Note that if there is a man in the middle, the two sides will calculate different numbers because the number is a function of the public keys used to encrypt the random numbers. If both sides used the same two public keys, then there can't be a man in the middle.

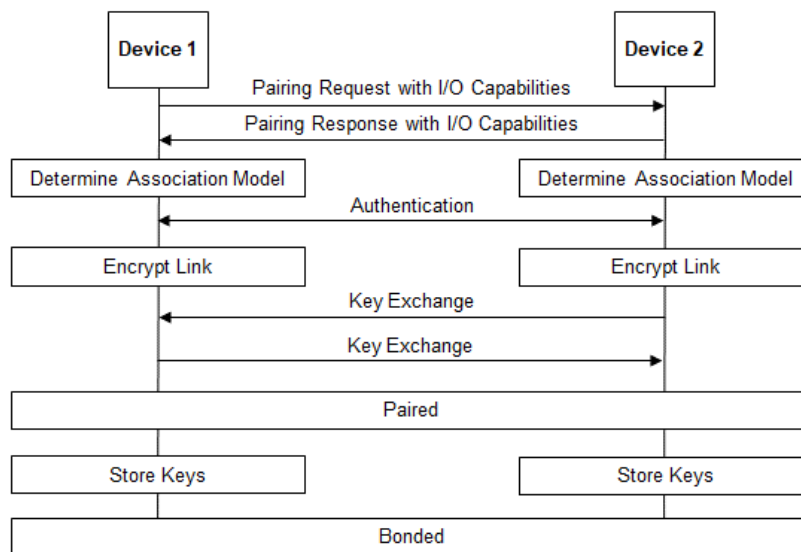


## 5c.10.2 Bonding

The whole process of Pairing is a bit painful and time consuming. It is also the most vulnerable part of establishing security, so it is beneficial to do it only once. Certainly, you don't want to have to repeat it every time two devices connect. This problem is solved by Bonding, which just saves all the relevant information into a non-volatile memory. This allows the next connection to launch without repeating the pairing process.

## 5c.10.3 Pairing & Bonding Process Summary

### BLE Pairing And Bonding Procedure



The pairing process involves authentication and key-exchange between BLE devices. After pairing the BLE devices must store the keys to be bonded.

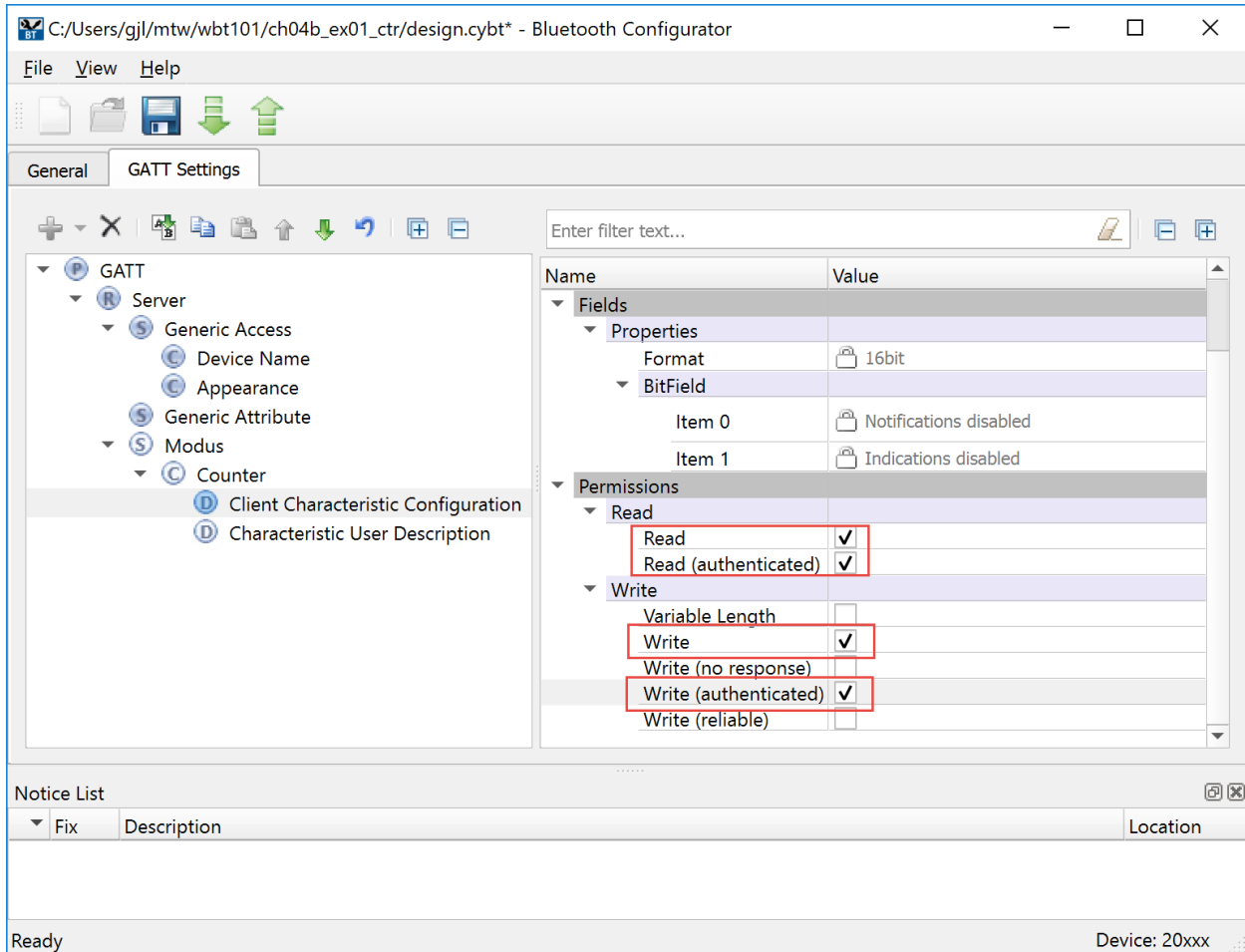
## 5c.10.4 Authentication, Authorization and the GATT DB

In Chapter 4A3.1, we talked about the Attributes and the GATT Database. Each Attribute has a permissions bit field that includes bits for Encryption, Authentication, and Authorization. The Bluetooth Stack will guarantee that you will not be able to access an Attribute that is marked Encryption or Authentication unless the connection is Authenticated and/or Encrypted.

The Authorization flag is not enforced by the Bluetooth Stack. Your Application is responsible for implementing the Authorization semantics. For example, you might not allow someone to turn off/on a switch without entering a password.

## 5c.10.5 Security in the Bluetooth Configurator

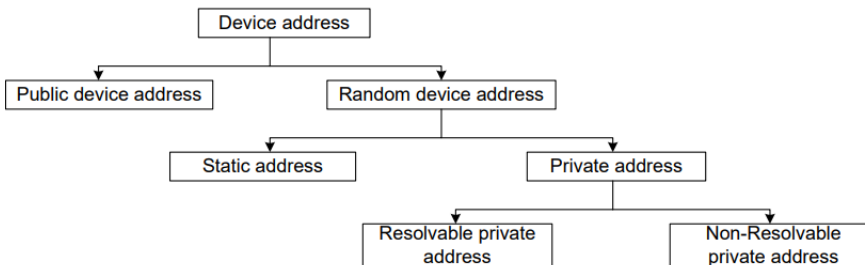
In order to enable security (i.e. to require pairing before allowing the client read or write of a characteristic) you just click the "Read (authenticated)" or "Write (authenticated)" button in the permissions for that characteristic. Note that this is a bitmask setting, so you must still keep "Read" and/or "Write" selected when you enable authentication. If not, reads/writes to that characteristic will fail. Enabling security works the same way for Descriptors such as the CCCD. That is, you can require authentication before allowing reads/writes of the CCCD (thereby preventing the client from turning Notifications on/off without pairing first) from the previous example like this:



## 5c.10.6 Link Layer Privacy

BLE devices are identified using a 48-bit device address. This device address is part of all the packets sent by the device in the advertising channels. A third device which listens on all three advertising channels can easily track the activities of a device by using its device address. Link Layer Privacy is a feature that reduces the ability to track a BLE device by using a private address that is generated and changed at regular intervals. Note that this is different than security (i.e. encrypting of messages).

There are a few different types of address types possible for BLE devices:



The device address can be a Public Device Address or a Random Device Address. The Public Device Addresses are comprised of a 24-bit company ID (an Organizationally Unique Identifier or OUI based on an IEEE standard) and a 24-bit company-assigned number (unique for each device); these addresses do not change over time.

There are two types of Random Addresses: Static Address and Private Address. The Static Address is a 48-bit randomly generated address with the two most significant bits set to 1. Static Addresses are generated on first power up or during manufacturing. A device using a Public Device Address or Static Address can be easily discovered and connected to by a peer device. Private Addresses change at some interval to ensure that the BLE device cannot be tracked. A Non-Resolvable Private Address cannot be resolved by any device so the peer cannot identify who it is connecting to. Resolvable Private Addresses (RPA) can be resolved and are used by Privacy-enabled devices.

Every Privacy-enabled BLE device has a unique address called the Identity Address and an Identity Resolving Key (IRK). The Identity Address is the Public Address or Static Address of the BLE device. The IRK is used by the BLE device to generate its RPA and is used by peer devices to resolve the RPA of the BLE device. Both the Identity Address and the IRK are exchanged during the third stage of the pairing process. Privacy-enabled BLE devices maintain a list that consists of the peer device's Identity Address, the local IRK used by the BLE device to generate its RPA, and the peer device's IRK used to resolve the peer device's RPA. This is called the Resolving List. Only peer devices that have the 128-bit identity resolving key (IRK) of a BLE device can determine the device's address.

A Privacy-enabled BLE device periodically changes its RPA to avoid tracking. The BLE Stack configures the Link Layer with a value called RPA Timeout that specifies the time after which the Link Layer must generate a new RPA. In ModusToolbox, this value is set in `app_bt_cfg.c` and is called `rpa_refresh_timeout`. If the `rpa_refresh_timeout` is set to 0 (i.e. `WICED_BT_CFG_DEFAULT_RANDOM_ADDRESS_NEVER_CHANGE`), privacy is disabled, and a public device address will be used.

Apart from this, Bluetooth 5.0 introduced more options in the form of privacy modes. There are two modes: device privacy mode and network privacy mode. A device in device privacy mode is only concerned about the privacy of the device itself and will accept advertising physical channel PDU's (Advertising, Scanning and Initiating packets) from peer devices that contain their identity address as well as ones that contain a private address, even if the peer device has distributed its IRK in the past. In network privacy mode, a device will only accept advertising packets from peer devices that contain a private address. By default, network privacy mode is used when private addresses are resolved and generated by the Controller. The Host can specify the privacy mode to be used with each peer identity on the resolving list. The table below shows the logical representation of the resolving list entries. Depending on the privacy mode entry in the resolving list, the device will behave differently with each peer device.

Device	Local IRK	Peer IRK	Peer Identity Address	Identity Address Type	Privacy Mode
1	Local IRK	Peer 1 IRK	Peer 1 Identity Address	Static/Public	Network/Device
2	Local IRK	Peer 2 IRK	Peer 2 Identity Address	Static/Public	Network/Device
3	Local IRK	Peer 3 IRK	Peer 3 Identity Address	Static/Public	Network/Device

## 5c.11 Firmware Architecture for Security

The firmware architecture is the same as was described in the previous chapter. The only difference is that there are additional Stack Management events and GATT Database events that occur.

For a typical BLE application that connects using a Paired link but does NOT use privacy, does NOT store bonding information in EEPROM and does NOT require a passkey, the order of callback events will look like this:

Activity	Callback Event Name (both Stack and GATT)	Reason
Powerup	BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT	At initialization, the BLE stack looks to see if the privacy keys are available. If privacy is not enabled, then this state does not need to be implemented as long as you return a default value of WICED_BT_SUCCESS.
	BTM_ENABLED_EVT	This occurs once the BLE stack has completed initialization. Typically, you will start up the rest of your application here.
	BTM_BLE_ADVERT_STATE_CHANGED_EVT	This occurs when you enable advertisements. You will see a return value of 3 for fast advertisements. After a timeout, you may see this again with a return value of 4 for slow advertisements. Eventually the state changes to 0 (off) if there have been no connections, giving you a chance to save power.

Activity	Callback Event Name (both Stack and GATT)	Reason
Connect	GATT_CONNECTION_STATUS_EVT	The callback needs to determine if the event is a connection or a disconnection. For a connection, the connection ID is saved, and pairing is enabled (if a secure link is required).
	BTM_BLE_ADVERT_STATE_CHANGED_EVT	Once the connection happens, the stack stops advertisements which will result in this event. You will see a return value of 0 which means advertisements have stopped.
Pair (if secure link is required)	BTM_SECURITY_REQUEST_EVT	The occurs when the client requests a secure connection. When this event happens, you need to call <code>wiced_bt_ble_security_grant()</code> to allow a secure connection to be established.
	BTM_PAIRING_IO_CAPABILITIES_BLE_REQUEST_EVT	This occurs when the client asks what type of capability your device has that will allow validation of the connection (e.g. screen, keyboard, etc.). You need to set the appropriate values when this event happens.
	BTM_ENCRYPTION_STATUS_EVT	This occurs when the secure link has been established.
	BTM_PAIRING_DEVICE_LINK_KEYS_UPDATE_EVT	This event is used so that you can store the paired devices keys if you are storing bonding information. If not, then this state does not need to be implemented.
	BTM_PAIRING_COMPLETE_EVT	This event indicates that pairing has been completed successfully.
Read Values	GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_READ	The firmware must get the value from the correct location in the GATT database.
Write Values	GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_WRITE	The firmware must store the provided value in the correct location in the GATT database.
Notifications	N/A	Notifications must be sent whenever an attribute that has notifications set is updated by the firmware. Since the change comes from the local firmware, there is no stack or GATT event that initiates this process.
Disconnect	GATT_CONNECTION_STATUS_EVT	For a disconnection, the connection ID is reset, all CCCD settings are cleared, and advertisements are restarted.
	BTM_BLE_ADVERT_STATE_CHANGED_EVT	Upon a disconnect, the firmware will get a GATT event handler callback for the <code>GATT_CONNECTION_STATUS_EVENT</code> (more on this later). At that time, it is the user's responsibility to determine if advertising should be re-started. If it is restarted, then you will get a BLE stack callback once advertisements have restarted with a return value of 3 (fast advertising) or 4 (slow advertising).

If bonding information is stored to EEPROM, the event sequence will look like the following. The sequence is shown for three cases (each shaded differently):

1. First-time connection before bonding information is saved
2. Connection after bonding information has been saved for disconnect/re-connect without resetting the kit between connections.
3. Connection after bonding information has been saved for disconnect/reset/re-connect.

In the reconnect cases, you can see that the pairing sequence is greatly reduced since keys are already available.

Activity	Callback Event Name	Reason
1 <sup>st</sup> Powerup	BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT	When this event occurs, the firmware needs to load the privacy keys from EEPROM. If keys have not been previously saved, then this state must return a value other than WICED_BT_SUCESS such as WICED_BT_ERROR. The non-success return value causes the stack to generate new privacy keys.
	BTM_ENABLED_EVT	This occurs once the BLE stack has completed initialization. Typically, you will start up the rest of your application here.  During this event, the firmware needs to load keys (which also includes the BD_ADDR) for a previously bonded device from EEPROM and then call <i>wiced_bt_dev_add_device_to_address_resolution_db()</i> to allow connecting to an bonded device. If a device has not been previously bonded, this will return values of all 0.
	BTM_BLE_ADVERT_STATE_CHANGED_EVT	This occurs when you enable advertisements. You will see a return value of 3 for fast advertisements. After a timeout, you may see this again with a return value of 4 for slow advertisements. Eventually the state changes to 0 (off) if there have been no connections, giving you a chance to save power.
	BTM_LOCAL_IDENTITY_KEYS_UPDATE_EVT	This event is called if reading of the privacy keys from EEPROM failed (i.e. the BDA returned is all 0's meaning no address has been stored). During this event, the privacy keys must be saved to EEPROM.
	BTM_LOCAL_IDENTITY_KEYS_UPDATE_EVT	This is called twice to update both the IRK and the ER in two steps.
1 <sup>st</sup> Connect	GATT_CONNECTION_STATUS_EVT	The callback needs to determine if the event is a connection or a disconnection. For a connection, the connection ID is saved, and pairing is enabled (if a secure link is required).
	BTM_BLE_ADVERT_STATE_CHANGED_EVT	Once the connection happens, the stack stops advertisements which will result in this event. You will see a return value of 0 which means advertisements have stopped.



Activity	Callback Event Name	Reason
1 <sup>st</sup> Pair	BTM_SECURITY_REQUEST_EVT	The occurs when the client requests a secure connection. When this event happens, you need to call <code>wiced_bt_ble_security_grant()</code> to allow a secure connection to be established.
	BTM_PAIRING_IO_CAPABILITIES_BLE_REQUEST_EVT	This occurs when the client asks what type of capability your device has that will allow validation of the connection (e.g. screen, keyboard, etc.). You need to set the appropriate values when this event happens.
	BTM_PASSKEY_NOTIFICATION_EVT	This event only occurs if the IO capabilities are set such that your device has the capability to display a value, such as <code>BTM_IO_CAPABILITIES_DISPLAY_ONLY</code> . In this event, the firmware should display the passkey so that it can be entered on the client to validate the connection.
	BTM_USER_CONFIRMATION_REQUEST_EVT	This event only occurs if the IO capabilities are set such that your device has the capability to display a value and accept Yes/No input, such as <code>BTM_IO_CAPABILITIES_DISPLAY_AND_YES_NO_INPUT</code> . In this event, the firmware should display the passkey so that it can be compared with the value displayed on the Client. This state should also provide confirmation to the Stack (either with or without user input first).
	BTM_ENCRYPTION_STATUS_EVT	This occurs when the secure link has been established. Previously saved information such as paired device <code>BD_ADDR</code> and notify settings is read. If no device has been previously bonded, this will return all 0's.
	BTM_PAIRING_DEVICE_LINK_KEYS_UPDATE_EVT	During this event, the firmware needs to store the keys of the paired device (including the <code>BD_ADDR</code> ) into EEPROM so that they are available for the next time the devices connect.
	BTM_PAIRING_COMPLETE_EVT	This event indicates that pairing has been completed successfully.  Information about the paired device such as its <code>BT_ADDR</code> should be saved in EEPROM at this point. You may also initialize other state information to be saved such as notify settings.
Read Values	GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_READ	The firmware must get the value from the correct location in the GATT database.
Write Values	GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_WRITE	The firmware must store the provided value in the correct location in the GATT database.
Notifications	N/A	Notifications must be sent whenever an attribute that has notifications set is updated by the firmware. Since the change comes from the local firmware, there is no stack or GATT event that initiates this process.
Disconnect	BTM_BLE_ADVERT_STATE_CHANGED_EVT	Upon a disconnect, the firmware will get a GATT event handler callback for the <code>GATT_CONNECTION_STATUS_EVENT</code> (more on this later). At that time, it is the user's responsibility to determine if advertising should be re-started. If it is restarted, then you will get a BLE stack callback once advertisements have restarted with a return value of 3 (fast advertising) or 4 (slow advertising).

Activity	Callback Event Name	Reason
Re-Connect	GATT_CONNECTION_STATUS_EVT	The callback needs to determine if the event is a connection or a disconnection. For a connection, the connection ID is saved, and pairing is enabled (if a secure link is required).
	BTM_BLE_ADVERT_STATE_CHANGED_EVT	Advertising off.
Re-Pair	BTM_ENCRYPTION_STATUS_EVT	In this state, the firmware reads the state of the server from EEPROM. For example, the BD_ADDR of the paired device and the saved state of any notify settings may be read. Since the paired device BD_ADDR and keys were already available, no other steps are needed to complete pairing.
Read Values	GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_READ	The firmware must get the value from the correct location in the GATT database.
Write Values	GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_WRITE	The firmware must store the provided value in the correct location in the GATT database.
Notifications	N/A	Notifications must be sent whenever an attribute that has notifications set is updated by the firmware. Since the change comes from the local firmware, there is no stack or GATT event that initiates this process.
Disconnect	BTM_BLE_ADVERT_STATE_CHANGED_EVT	Advertising on.
Reset	BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT	Local keys are loaded from EEPROM.
	BTM_ENABLED_EVT	Stack is enabled. Paired device keys (including the BD_ADDR) are loaded from EEPROM and the device is added to the address resolution database.
	BTM_BLE_ADVERT_STATE_CHANGED_EVT	Advertising on.
Re-Connect	GATT_CONNECTION_STATUS_EVT	The callback needs to determine if the event is a connection or a disconnection. For a connection, the connection ID is saved, and pairing is enabled (if a secure link is required).
	BTM_BLE_ADVERT_STATE_CHANGED_EVT	Advertising off.
Re-Pair	BTM_PAIRING_DEVICE_LINK_KEYS_REQUEST_EVT	Since we are connecting to a known device (because it is in the address resolution database), this event is called by the stack so that the firmware can load the paired device's keys from EEPROM. If keys are not available, this state must return WICED_BT_ERROR. That return value causes the stack to generate keys and then it will call the corresponding update event so that the new keys can be saved in EEPROM.
	BTM_ENCRYPTION_STATUS_EVT	In this state, the firmware reads the state of the server from non-volatile memory. For example, the BD_ADDR of the paired device and the saved state of any notify settings may be read. Since the paired device BD_ADDR and keys were already available in EEPROM, no other steps are needed to complete pairing.
Read Values	GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_READ	The firmware must get the value from the correct location in the GATT database.
Write Values	GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_WRITE	The firmware must store the provided value in the correct location in the GATT database.
Notifications	N/A	Notifications must be sent whenever an attribute that has notifications set is updated by the firmware. Since the change comes from the local firmware, there is no stack or GATT event that initiates this process.
Disconnect	BTM_BLE_ADVERT_STATE_CHANGED_EVT	Advertising on.

## 5c.12 Exercises (Part 3)

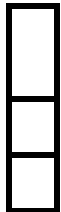
### 5c.12.1 Exercise 3: Paring

Add pairing capability to the Notification exercise. Change the LED, Counter, and CCCD Attribute permissions so that they require a paired (authenticated) connection before they can be read/written.

Below is a table showing the events that occur during this exercise. Arrows indicate the cause/effect of the stack events. New events introduced in this exercise are highlighted.

External Event	BLE Stack Event	Action
Board reset →	BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT →	Not used yet
	BTM_ENABLED_EVT →	Initialize application
	BTM_BLE_ADVERT_STATE_CHANGED_EVT (BTM_BLE_ADVERT_UNDIRECTED_HIGH)	← Start advertising
CySmart will see advertising packets		
Connect to device from CySmart →	GATT_CONNECTION_STATUS_EVT →	Set the connection ID and enable pairing
	BTM_BLE_ADVERT_STATE_CHANGED_EVT (BTM_BLE_ADVERT_OFF)	
Pair →	BTM_SECURITY_REQUEST_EVT →	Grant security
	BTM_PAIRING_IO_CAPABILITIES_BLE_REQUEST_EVT →	Capabilities are set
	BTM_ENCRYPTION_STATUS_EVT	Not used yet
	BTM_PAIRING_DEVICE_LINK_KEYS_UPDATE_EVT	Not used yet
	BTM_PAIRING_COMPLETE_EVT	Not used yet
Read Button characteristic while pressing button →	GATT_ATTRIBUTE_REQUEST_EVT, GATTS_REQ_TYPE_READ →	Returns button state
Read Button CCCD →	GATT_ATTRIBUTE_REQUEST_EVT, GATTS_REQ_TYPE_READ →	Returns button notification setting
Write 01:00 to Button CCCD →	GATT_ATTRIBUTE_REQUEST_EVT, GATTS_REQ_TYPE_WRITE →	Enables notifications
Press button →		Send notifications
Disconnect →	GATT_CONNECTION_STATUS_EVT →	Clear the connection ID
	BTM_BLE_ADVERT_STATE_CHANGED_EVT (BTM_BLE_ADVERT_UNDIRECTED_HIGH)	Re-start advertising
Wait for timeout →	BTM_BLE_ADVERT_STATE_CHANGED_EVT (BTM_BLE_ADVERT_UNDIRECTED_LOW)	Stack switches to lower advertising rate to save power
Wait for timeout →	BTM_BLE_ADVERT_STATE_CHANGED_EVT (BTM_BLE_ADVERT_OFF)	Stack stops advertising

### Application Creation



1. Import the previously completed notification exercise (not the template) into a new one using the Project Creator Import function. Call the new application **ch05c\_ex03\_pair**.
2. Open the Bluetooth Configurator.
  - a. Change the Device Name to <init>\_pair.



- b. In the Counter characteristic set the "Read Authentication Required" permission, which will make the peripheral reject read requests unless the devices are paired.

Hint: You MUST leave "Read" checked also. It will not work with just "Read Authentication Required" checked.



- c. Update the Client Characteristic Configuration descriptor to require authenticated read and write.

This will cause the application to require pairing to view or change the notification settings.

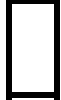
Hint: You MUST leave "Read" and "Write" checked also.



- d. Save the edits and close the configurator.



3. In main.c, look for the call to `wiced_bt_set_pairable_mode()` mode and set the first argument to `WICED_TRUE` to allow pairing.



4. Leave the second argument as `WICED_FALSE` - when it is set to true, this argument indicates that ONLY previously paired devices are allowed to connect.



5. In the `BTM_PAIRING_IO_CAPABILITIES_BLE_REQUEST_EVT` management case tell the central that you require MITM protection, but the device has no IO capabilities.

```
p_event_data->pairing_io_capabilities_ble_request.auth_req =  
BTM_LE_AUTH_REQ_SC_MITM_BOND;  
p_event_data->pairing_io_capabilities_ble_request.init_keys =  
BTM_LE_KEY_PENC|BTM_LE_KEY_PID;  
p_event_data->pairing_io_capabilities_ble_request.local_io_cap =  
BTM_IO_CAPABILITIES_NONE;
```



6. In the `BTM_SECURITY_REQUEST_EVT` management case grant the authorization to the central by using the following code:

```
wiced_bt_ble_security_grant( p_event_data->security_request.bd_addr,  
WICED_BT_SUCCESS );
```

## Testing


1. Build and program the application to the board.
2. Open the mobile CySmart app. Watch the UART messages during the next steps to see which BT events occur.
3. Connect to the device.
4. Open the GATT browser, navigate to the Counter characteristic (the one with Read and Notify Properties), press Descriptors and then the Client Characteristic Configuration.
5. If requested, accept the invitation to pair the devices.
6. Return to the Counter Characteristic, enable Notifications and observe the Counter value as you press the button on the kit.
7. Disable Notifications and observe that the Counter value does not update as you press the button on the kit.
8. Read the Counter value manually to verify that it is still incrementing.
9. Disconnect from the mobile CySmart app.
10. Go to the phone's Bluetooth settings and remove the <init>\_pair device from the paired devices list.

This is necessary so that when you re-program the kit the phone won't have stale bonding information stored which could prevent you from re-connecting. In the next exercise we'll store bonding information on the BLE device so that you will be able to leave the devices paired if you desire.

## 5c.12.2 Exercise 4: Bonding

### Introduction

The prior exercise has been modified for you to save and restore bonding information to emulated EEPROM. You will create the completed application from a template, program it to your kit, experiment with it, and then answer questions about the stack events that occur.

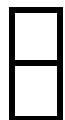
By saving Bonding information on both sides (i.e. the client and the server) future connections between the devices can be established more quickly with fewer steps. This is particularly useful for devices that require a pairing passkey (which will be added in the next exercise) since saving the bonding information means the passkey doesn't have to be entered every time the device connects.

Moreover, since the keys are saved on both devices, they don't need to be exchanged again. This means that after the first connection, there is no possibility of a MITM attack since the keys are not sent out over the air.

The firmware has two "modes": *bonding mode* and *bonded mode*. After programming, the kit will start out in bonding mode. LED1 will blink at 1 Hz to indicate that the kit is waiting to be Paired/Bonded. Once a Client connects to the kit and pairs with it, the Bonding information will be saved in non-volatile memory. The LED will be ON since the kit is connected. The only Client that will be allowed to pair with the kit is the one that is bonded (the firmware only allows 1 bonded device at a time for now). If the Bonding information is removed from the Client, it will no longer be able to Pair/Bond with the kit without going through the Pairing/Bonding process again.

When you disconnect, LED1 will blink at 5 Hz to indicate that it is bonded. To remove Bonding information from the kit and return bonding mode, press 'e' in the UART terminal window. This will erase the stored bonding information and put the kit back into Bonding mode. LED1 will now go back to flashing at 1 Hz. When you reconnect, the bonding process must be done again to connect. This allows you to Pair/Bond from a Client that has "lost" the bonding information or to Pair/Bond with a new device without having to reprogram the kit.

### Application Creation



1. Create a new application called **ch05c\_ex04\_bond** using the template provided.
2. Open the Bluetooth Configurator.
  - a. Change the Device Name to <init>\_bond.
  - b. Save the edits and close the configurator.



3. Open `app_bt_cfg.c` and verify the `rpa_refresh_timeout` is set to:
  - `WICED_BT_CFG_DEFAULT_RANDOM_ADDRESS_CHANGE_TIMEOUT`This enables privacy with an address change frequency of 900 seconds.



4. The remaining code for this exercise has already been implemented.

## Testing

☐  
☐  
☐  
☐

1. Open a UART terminal window to the PUART.
2. Build the application and program it to the board.
3. Open the CySmart mobile application.
4. Start scanning and locate your device.

Your device shows up with a Random Bluetooth address now since privacy is enabled. (Note that on iOS you can't see the Bluetooth device address).

☐  
☐  
☐  
☐

5. Connect to your device, open the GATT browser, click on the Service, and then on the Counter Characteristic.
6. If requested, accept the invitation to pair the devices.
7. Note down the Bluetooth Stack events that occur during pairing. This information is displayed in the UART.
8. Disconnect from the device. Do NOT remove the device from the phone's list of paired devices this time.

Hint: You will notice that the LED is blinking at 5 Hz. The firmware was written to do this when it is not connected but has bonding information stored.

☐  
☐  
☐  
☐  
☐

9. Re-scan and find your device in the list.
10. Re-connect to your device and view the Counter Characteristic.
11. Once again note down the Bluetooth Stack events that occur during pairing. You will notice that fewer steps are required this time.
12. Disconnect again.
13. Reset or power cycle the board.

Hint: If you power cycle the board, you will need to either reset or re-open the UART terminal window.

☐  
☐  
☐  
☐  
☐

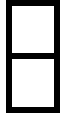
14. Start a scan, find your device in the list, connect to your device for a third time and then view the Counter Characteristic.
15. Note down the Bluetooth Stack events that occur this time during pairing. Compare to the previous two connections.
16. Disconnect again.
17. Remove the device from the list of bonded devices in the Phone's Bluetooth settings.
18. Start a scan and find your device.



19. Connect to your device and try to read the Counter Characteristic.

Note that pairing will not complete because CySmart no longer has the required keys to use. You will not be able to read the Counter value because it requires an authenticated connection.

- Hint: If you look in the UART window you will see a message about the security request being denied.



20. Disconnect from the device.

21. Press "e" in the UART window to erase bonding information and reset the kit.

This forces it to restart advertising (it would restart advertising automatically if you waited long enough for the disallowed pairing operation to timeout).

Note that LED1 begins blinking at 1 Hz. This indicates that the bonding information has been cleared from the device and it will now allow a new connection.



22. Scan, Connect, and attempt to read the Counter Characteristic again. This time it should work.

23. Note the steps that the firmware goes through this time.

24. Disconnect a final time and remove the device from the phone's paired Bluetooth devices so that the saved bonding information won't interfere with any future tests.

Hint: You should clear the bonding information anytime you are going to reprogram the kit or otherwise clear bonding information since the BLE device will no longer have the bonding information on its side.

## Overview of Changes

- There are a lot of messages printed in this example for learning purposes. In a real application, most if not all of these messages would be removed.
- The LED characteristic and functionality were removed so that the LED can indicate connection status. A PWM is added that will operate the LED as follows:

Advertising?	Connected?	Bonded?	LED
No	No	N/A	OFF
No	Yes	N/A	ON
Yes	No	No	Blinking at 1 Hz
Yes	No	Yes	Blinking at 5 Hz
Yes	Yes	N/A	N/A - this case doesn't occur

- A structure called "hostinfo" is created which holds the BD\_ADDR of the bonded device and the value of the Button CCCD. The BD\_ADDR is used to determine when we have reconnected to the same device while the CCCD value is saved so that the state of notifications can be retained across connections for bonded devices.
- Before initializing the GATT database, existing keys (if any) are loaded from EEPROM. If no keys are available this step will fail so it is necessary to look at the result of the EEPROM read. If the read was successful, then the keys are copied to the address resolution database and the



variable called "bonded" is set as TRUE. Otherwise, it stays FALSE, which means the device can accept new pairing requests.

- In the BTM\_SECURITY\_REQUEST\_EVENT look to see if bonded is FALSE. Security is only granted if the device is not bonded.
- In the Bluetooth stack event *BTM\_PAIRING\_COMPLETE\_EVT* if bonding was successful write the information from the hostinfo structure into the EEPROM and set bonded to TRUE.
  - This saves hostinfo upon initial pairing. This event is not called when bonded devices reconnect.
- In the Bluetooth stack event *BTM\_ENCRYPTION\_STATUS\_EVT*, if the device is bonded (i.e. bonded is TRUE), read bonding information from the EEPROM into the hostinfo structure.
  - This reads hostinfo upon a subsequent connection when devices were previously bonded.
- In the Bluetooth stack event *BTM\_PAIRING\_DEVICE\_LINK\_KEYS\_UPDATE\_EVT*, save the keys for the peer device to EEPROM.
- In the Bluetooth stack event *BTM\_PAIRING\_DEVICE\_LINK\_KEYS\_REQUEST\_EVT*, read the keys for the peer device from EEPROM.
- In the Bluetooth stack event *BTM\_LOCAL\_IDENTITY\_KEYS\_UPDATE\_EVT*, save the keys for the local device to EEPROM.
- In the Bluetooth stack event *BTM\_LOCAL\_IDENTITY\_KEYS\_REQUEST\_EVT*, read the keys for the local device from EEPROM.
- In the GATT connect callback:
  - For a connection, save the BD\_ADDR of the remote device into the hostinfo structure. This will be written to EEPROM in the *BTM\_PAIRING\_COMPLETE\_EVT*.
  - For a disconnection, clear out the BD\_ADDR from the hostinfo structure and reset the CCCD to 0.
- In the GATT set value function, save the Button CCCD value to the hostinfo structure whenever it is updated and write the value into EEPROM.
- The UART is configured to accept input with a receive callback. Instead of using retarget-io, the UART is used directly from the HAL. The rx\_cback function looks for the key "e". If it has been sent, it sets bonded to FALSE, removes the bonded device from the list of bonded devices, removes the device from the address resolution database, and clears out the bonding information stored in EEPROM.
- Finally, privacy is enabled in *wiced\_bt\_cfg.c* by updating the *rpa\_refresh\_timeout* to *WICED\_BT\_CFG\_DEFAULT\_RANDOM\_ADDRESS\_CHANGE\_TIMEOUT*.

## Questions

☐

1. What items are stored in EEPROM?

☐

2. Which event stores each piece of information?

☐

3. Which event retrieves each piece of information?

☐

4. In what event is the privacy info read from EEPROM?

☐

5. Which event is called if privacy information is not retrieved after new keys have been generated by the stack?

## 5c.12.3 Exercise 5: Passkey and Numeric Comparison

### Introduction

The prior exercise has been modified for you to support both numeric comparison and passkey notification before allowing bonding (the method chosen will depend on the central device's capabilities). You will create the completed application from a template, program it to your kit, and experiment with it.

### Application Creation

- ☐ 1. Create a new application called **ch05c\_ex05\_verify** using the template provided.
- ☐ 2. Open the Bluetooth Configurator.
  - a. Change the Device Name to <init>\_verify.
  - b. Save the edits and close the configurator.
- ☐ 3. The remaining code for this exercise has already been implemented.

### Testing

- ☐ 1. Open a UART terminal window to the PUART.
- ☐ 2. Build the application and program it to the board.
- ☐ 3. Open the CySmart mobile application.
- ☐ 4. Start scanning and locate your device.  
 Your device shows up with a Random Bluetooth address now since privacy is enabled.  
 (Note that on iOS you can't see the Bluetooth device address).
- ☐ 5. Connect to your device, open the GATT browser, click on the Service, and then on the Counter Characteristic.
- ☐ 6. Follow the prompts on the phone and the UART terminal to complete pairing.  
 Note: By default, if Numeric Comparison is used the application is configured to only require the user to validate that the numbers match on the phone side. If you want to test the application with validation required on both sides of the connection, change the value of the macro `USE_2SIDE_NUMERIC_VERIFICATION` in `main.c` to `true`. In that case, you **MUST** press 'y' on the kit UART terminal **BEFORE** validating the request on the phone. This is a bug in the current stack implementation.
- ☐ 7. Note down the Bluetooth Stack events that occur during pairing. This information is displayed in the UART.
- ☐ 8. Disconnect from the device. Do **NOT** remove the device from the phone's list of paired devices this time.
- ☐ 9. Re-scan and find your device in the list.
- ☐ 10. Re-connect to your device and view the Counter Characteristic.



11. Once again note down the Bluetooth Stack events that occur during pairing.

You will notice that fewer steps are required this time and you are not required to enter a passkey or perform numeric comparison.



12. Disconnect again.



13. Remove the device from the list of bonded devices in the Phone's Bluetooth settings.

## Overview of Changes

- Two additional states are added to the Bluetooth management callback:
  - `BTM_USER_CONFIRMATION_REQUEST_EVT` - this event prints out the number for numeric comparison and sends a confirmation using `wiced_bt_dev_confirm_req_reply`. That is, the user agrees to the comparison only on the phone. The macro `USE_2SIDE_NUMERIC_VERIFICATION` can be set to `true` if you want to require validation on both sides of the connection.
  - `BTM_PASSKEY_NOTIFICATION_EVT` - this event prints out the passkey that the user must enter on the central to allow the connection and sends a confirmation using `wiced_bt_dev_confirm_req_reply`. Bonding proceeds when the user enters the correct passkey on the phone.
- The local IO capabilities in the `BTM_PAIRING_IO_CAPABILITIES_BLE_REQUEST_EVT` are changed to `BTM_IO_CAPABILITIES_DISPLAY_AND_YES_NO_INPUT`

## 5c.12.4 Exercise 6: Code Examples

In this exercise, you will look through and try out some of the existing AnyCloud BLE examples.



1. Look at the AnyCloud BLE examples listed in Project Creator. Go to GitHub to see the README.md files for each example to understand what they do.



2. Create and run one or more of the code examples.