

## Chapter 3: Basic Bluetooth® LE Peripheral

After completing this chapter, you will have all the required knowledge to create the most basic Bluetooth® Low Energy peripheral with a custom service containing a characteristic that can be read and written from a central.

### Table of contents

<b>3.1</b>	<b>Bluetooth® LE system lifecycle.....</b>	<b>3</b>
3.1.1	Turn on the stack .....	5
3.1.2	Start advertising.....	5
3.1.3	Make a connection .....	8
3.1.4	Exchange data .....	9
<b>3.2</b>	<b>Attributes, the Generic Attribute Profile &amp; GATT Database .....</b>	<b>10</b>
3.2.1	Attributes.....	10
3.2.2	Profiles, Services, Characteristics.....	10
3.2.3	Service Declaration .....	11
3.2.4	Service Declaration in the GATT DB.....	11
3.2.5	Characteristic Declaration .....	12
<b>3.3</b>	<b>Creating a simple Bluetooth® LE peripheral.....</b>	<b>13</b>
3.3.1	Bluetooth® configurator .....	13
3.3.2	Editing the firmware .....	20
3.3.3	Testing the application .....	22
<b>3.4</b>	<b>Bluetooth® libraries and settings .....</b>	<b>25</b>
3.4.1	Libraries.....	25
3.4.2	Makefile settings .....	25
<b>3.5</b>	<b>Stack Events .....</b>	<b>26</b>
3.5.1	Essential Bluetooth® management events .....	26
3.5.2	Essential GATT Events.....	26
<b>3.6</b>	<b>Firmware architecture .....</b>	<b>27</b>
3.6.1	Turn on the Stack .....	27
3.6.2	Start Advertising.....	28
3.6.3	Process GATT connection events .....	28
3.6.4	Process GATT attribute requests.....	28
3.6.5	Perform Memory Management.....	31
3.6.6	Function list.....	31
<b>3.7</b>	<b>GATT database implementation .....</b>	<b>32</b>
3.7.1	gatt_database[] array .....	32
3.7.2	gatt_db_ext_attr_tbl.....	34
3.7.3	uint8_t Arrays for the Values.....	34
<b>3.8</b>	<b>Scan Response Packets .....</b>	<b>35</b>
<b>3.9</b>	<b>AIROC™ Bluetooth® Connect .....</b>	<b>36</b>
<b>3.10</b>	<b>Exercises .....</b>	<b>37</b>
	Exercise 1: Simple Bluetooth® LE Peripheral.....	37
	Exercise 2: Implement a connection status LED.....	37
	Exercise 3: Scan Response Packet .....	39

## Document conventions

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

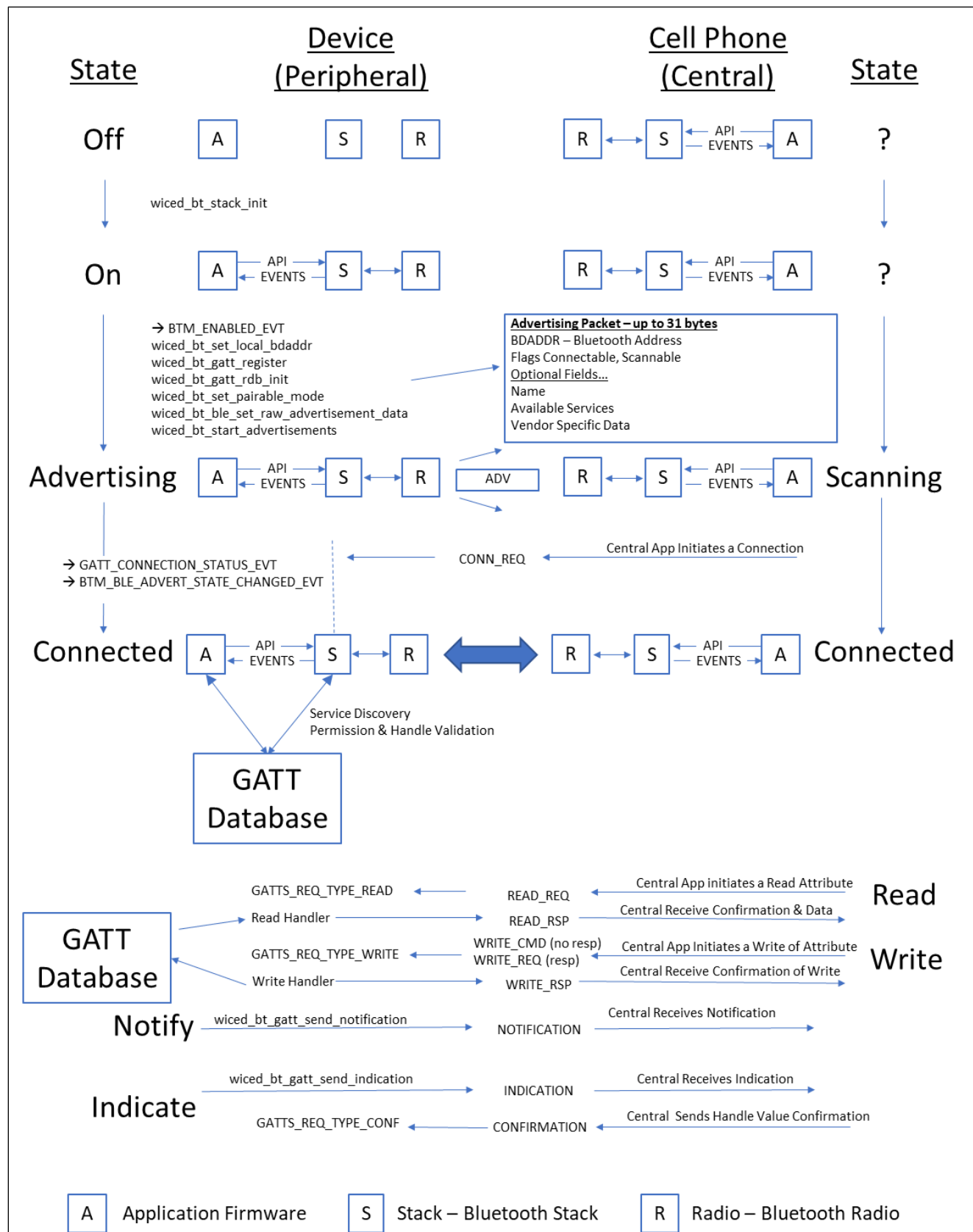
## 3.1 Bluetooth® LE system lifecycle

All Bluetooth® systems work the same basic way. You write Application [A] Firmware which calls Bluetooth® APIs in the Stack [S]. The Stack then talks to the Radio [R] hardware which in turn, sends and receives data. When something happens in the Radio, the Stack will also initiate actions in your Application firmware by creating Events (e.g. when it receives a message from the other side.). Your Application is responsible for processing these events and doing the right thing. This basic architecture is also true of Apps running on a cellphone (in iOS or Android) but we will not explore that in more detail in this course other than to run existing Apps on those devices.

There are 4 steps your application firmware needs to handle:

1. Turn on the Bluetooth® Stack (from now on referred to as "the Stack")
2. Start Advertising as connectable
3. Process GATT connection events from the Stack
4. Process GATT attribute requests from the Stack such as read and write and do memory management

Here is the overall picture, we will discuss this in pieces as we go:



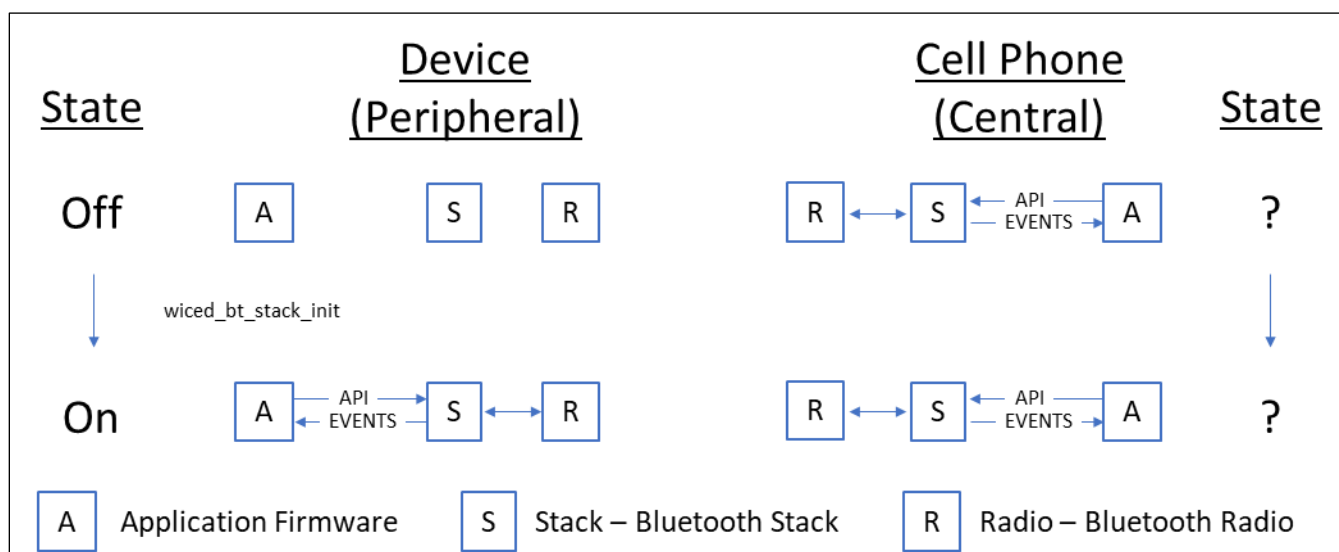
### 3.1.1 Turn on the stack

In the beginning, you have a Bluetooth® device and a Cell Phone and they are not connected. The Stack state is Off. That's where we will start.

Like all great partnerships, every Bluetooth® LE connection has two sides, one side called the **Peripheral** and one side called the **Central**. In the picture below, you can see that the Peripheral starts Off, there is no connection from the Peripheral to the Central (which is in an unknown state). In fact, at this point the Central doesn't know anything about the Peripheral and vice versa.

From a practical standpoint, the Peripheral should be the device that requires the lowest power – often it will be a small battery powered device like a health tracker or a watch. The reason is that the Central needs to Scan for devices (which is power consuming) while the Peripheral only needs to Advertise for short periods of time. Note that the GATT database is often associated with the Peripheral, but that is not required and sometimes it is the other way around.

The first thing you do in your firmware is to turn on Bluetooth® LE. That means that you initialize the Stack and provide it with a function that will be called when the Stack has events for you to process (this is often called the "callback" function for obvious reasons).



### 3.1.2 Start advertising

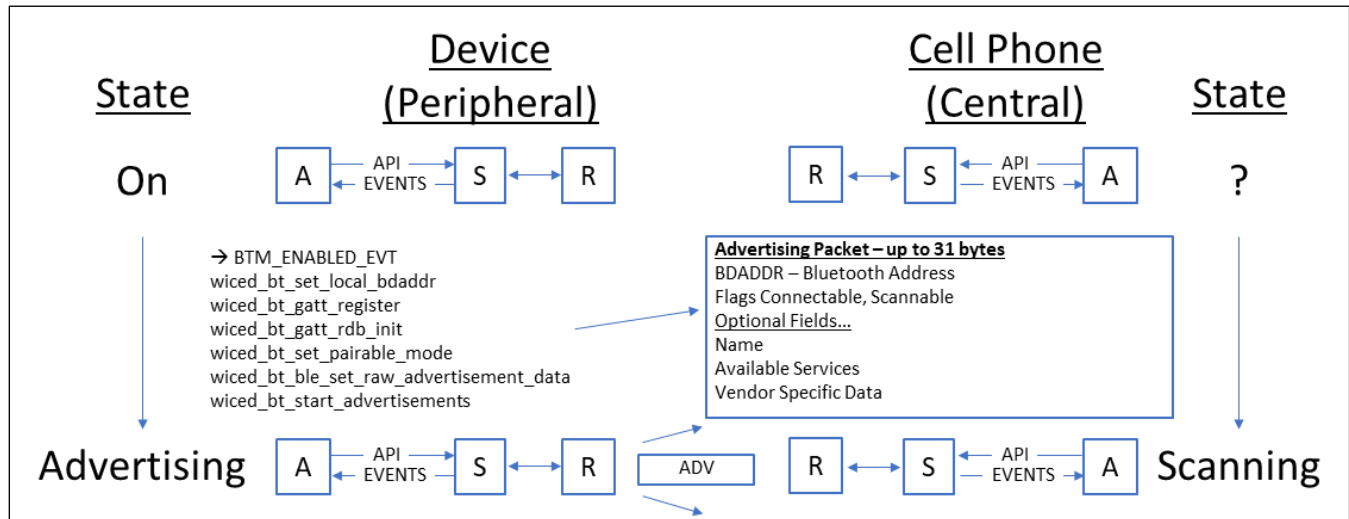
For a Central to know of your existence you need to send out Advertising packets. The Advertising Packet will contain your Bluetooth® Device Address (BDA), some flags that include information about your connection availability status, and one or more optional fields for other information, like your device name or what Services you provide (e.g. Heart Rate, Temperature, etc.). An advertising packet can contain at most 31 bytes.

There are four primary types of Bluetooth® Advertising Packets:

- BTM\_BLE\_EVT\_CONNECTABLE\_ADVERTISEMENT
- BTM\_BLE\_EVT\_CONNECTABLE\_DIRECTED\_ADVERTISEMENT
- BTM\_BLE\_EVT\_SCANNABLE\_ADVERTISEMENT
- BTM\_BLE\_EVT\_NON\_CONNECTABLE\_ADVERTISEMENT

When a Scannable Advertising Packet is scanned, the peripheral sends a Scan Response Packet (BTM\_BLE\_EVT\_SCAN\_RSP), which contains up to another 31 bytes of information.

The Stack is responsible for broadcasting your advertising packets at a configurable interval into the open air. That means that all Bluetooth® LE Centrals that are scanning and in range may hear your advertising packet and process it. Obviously, this is not a secure way of exchanging information, so be careful what you put in the advertising packet. We will discuss ways of improving security later.



### 3.1.2.1 Advertising packets

The Advertising Packet 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:

```
enum wiced_bt_ble_advert_type_e {
    BTM_BLE_ADVERT_TYPE_FLAG                = 0x01,          /**< Advertisement flags */
    BTM_BLE_ADVERT_TYPE_16SRV_PARTIAL        = 0x02,          /**< List of supported services - 16 bit UUIDs (partial) */
    BTM_BLE_ADVERT_TYPE_16SRV_COMPLETE       = 0x03,          /**< List of supported services - 16 bit UUIDs (complete) */
    BTM_BLE_ADVERT_TYPE_32SRV_PARTIAL        = 0x04,          /**< List of supported services - 32 bit UUIDs (partial) */
    BTM_BLE_ADVERT_TYPE_32SRV_COMPLETE       = 0x05,          /**< List of supported services - 32 bit UUIDs (complete) */
    BTM_BLE_ADVERT_TYPE_128SRV_PARTIAL       = 0x06,          /**< List of supported services - 128 bit UUIDs (partial) */
    BTM_BLE_ADVERT_TYPE_128SRV_COMPLETE      = 0x07,          /**< List of supported services - 128 bit UUIDs (complete) */
    BTM_BLE_ADVERT_TYPE_NAME_SHORT           = 0x08,          /**< Short name */
    BTM_BLE_ADVERT_TYPE_NAME_COMPLETE        = 0x09,          /**< Complete name */
    BTM_BLE_ADVERT_TYPE_TX_POWER             = 0x0A,          /**< TX Power level */
    BTM_BLE_ADVERT_TYPE_DEV_CLASS            = 0x0D,          /**< Device Class */
    BTM_BLE_ADVERT_TYPE_SIMPLE_PAIRING_HASH_C = 0x0E,          /**< Simple Pairing Hash C */
    BTM_BLE_ADVERT_TYPE_SIMPLE_PAIRING_RAND_C = 0x0F,          /**< Simple Pairing Randomizer R */
    BTM_BLE_ADVERT_TYPE_SM_TK                = 0x10,          /**< Security manager TK value */
    BTM_BLE_ADVERT_TYPE_SM_OOB_FLAG          = 0x11,          /**< Security manager Out-of-Band data */
    BTM_BLE_ADVERT_TYPE_INTERVAL_RANGE       = 0x12,          /**< Slave connection interval range */
    BTM_BLE_ADVERT_TYPE_SOLICITATION_SRV_UUID = 0x14,          /**< List of solicited services - 16 bit UUIDs */
    BTM_BLE_ADVERT_TYPE_128SOLICITATION_SRV_UUID = 0x15,          /**< List of solicited services - 128 bit UUIDs */
    BTM_BLE_ADVERT_TYPE_SERVICE_DATA         = 0x16,          /**< Service data - 16 bit UUID */
    BTM_BLE_ADVERT_TYPE_PUBLIC_TARGET        = 0x17,          /**< Public target address */
    BTM_BLE_ADVERT_TYPE_RANDOM_TARGET        = 0x18,          /**< Random target address */
    BTM_BLE_ADVERT_TYPE_APPEARANCE           = 0x19,          /**< Appearance */
    BTM_BLE_ADVERT_TYPE_ADVERT_INTERVAL      = 0x1A,          /**< Advertising interval */
    BTM_BLE_ADVERT_TYPE_LE_BD_ADDR           = 0x1B,          /**< LE device bluetooth address */
    BTM_BLE_ADVERT_TYPE_LE_ROLE              = 0x1C,          /**< LE role */
    BTM_BLE_ADVERT_TYPE_256SIMPLE_PAIRING_HASH = 0x1D,          /**< Simple Pairing Hash C-256 */
    BTM_BLE_ADVERT_TYPE_256SIMPLE_PAIRING_RAND = 0x1E,          /**< Simple Pairing Randomizer R-256 */
    BTM_BLE_ADVERT_TYPE_32SOLICITATION_SRV_UUID = 0x1F,          /**< List of solicited services - 32 bit UUIDs */
    BTM_BLE_ADVERT_TYPE_32SERVICE_DATA      = 0x20,          /**< Service data - 32 bit UUID */
    BTM_BLE_ADVERT_TYPE_128SERVICE_DATA     = 0x21,          /**< Service data - 128 bit UUID */
    BTM_BLE_ADVERT_TYPE_CONN_CONFIRM_VAL     = 0x22,          /**< LE Secure Connections Confirmation Value */
    BTM_BLE_ADVERT_TYPE_CONN_RAND_VAL        = 0x23,          /**< LE Secure Connections Random Value */
    BTM_BLE_ADVERT_TYPE_URI                  = 0x24,          /**< URI */
    BTM_BLE_ADVERT_TYPE_INDOOR_POS            = 0x25,          /**< Indoor Positioning */
    BTM_BLE_ADVERT_TYPE_TRANS_DISCOVER_DATA  = 0x26,          /**< Transport Discovery Data */
    BTM_BLE_ADVERT_TYPE_SUPPORTED_FEATURES    = 0x27,          /**< LE Supported Features */
    BTM_BLE_ADVERT_TYPE_UPDATE_CH_MAP_IND     = 0x28,          /**< Channel Map Update Indication */
    BTM_BLE_ADVERT_TYPE_PB_ADV               = 0x29,          /**< PB-ADV */
    BTM_BLE_ADVERT_TYPE_MESH_MSG             = 0x2A,          /**< Mesh Message */
    BTM_BLE_ADVERT_TYPE_MESH_BEACON          = 0x2B,          /**< Mesh Beacon */
    BTM_BLE_ADVERT_TYPE_3D_INFO_DATA          = 0x3D,          /**< 3D Information Data */
    BTM_BLE_ADVERT_TYPE_MANUFACTURER         = 0xFF,          /**< 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 10 bytes to your Advertising packet:

- Length: 9 (the length is 1 for the field type plus 8 for the data)
- Type: BTM\_BLE\_ADVERT\_TYPE\_NAME\_COMPLETE
- Data: '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. Each entry in the array of `wiced_bt_ble_advert_elem_t` structures contains data for one advertising field.

**Note:** *The advertisement data array and number of elements in the array are both generated by the Bluetooth® configurator when you set up an advertising packet, so you will typically just use those values instead of creating an advertisement packet manually. You will see how to do this when we create our first example.*

The 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.

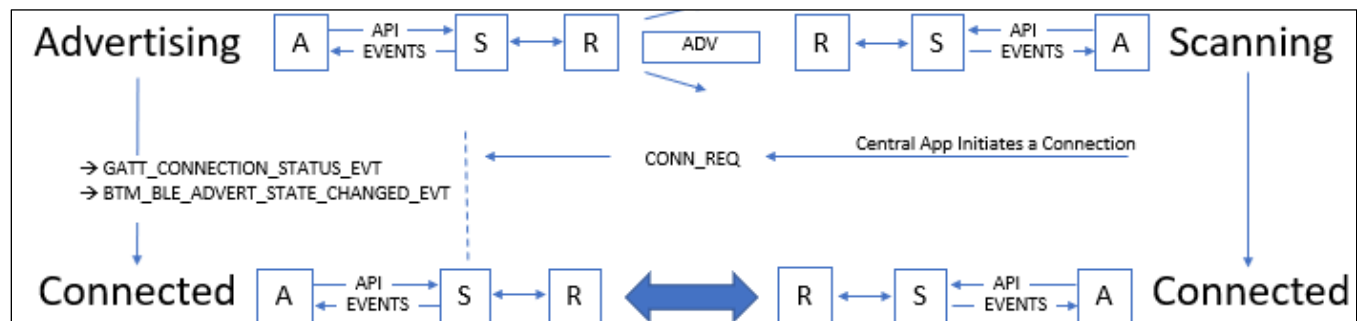
There is a scan response packet that can hold an additional 31 bytes which will be discussed later. The scan response packet array can also be setup by the Bluetooth® configurator.

### 3.1.3 Make a connection

Once a Central device processes your advertising packet it can choose what to do next such as initiating a connection. When the Central App initiates a connection, it will call a function which will trigger its Stack to generate a Bluetooth® Packet called a "conn\_req" which will then go out the Central's radio and through the air to your radio.

The Peripheral's radio will feed the packet to the Stack and it will automatically stop advertising. You do not have to write code to respond to the connection request, but the Stack will generate two callbacks to your firmware (more on that later).

You are now connected and can start exchanging messages with the central.





### 3.1.4 Exchange data

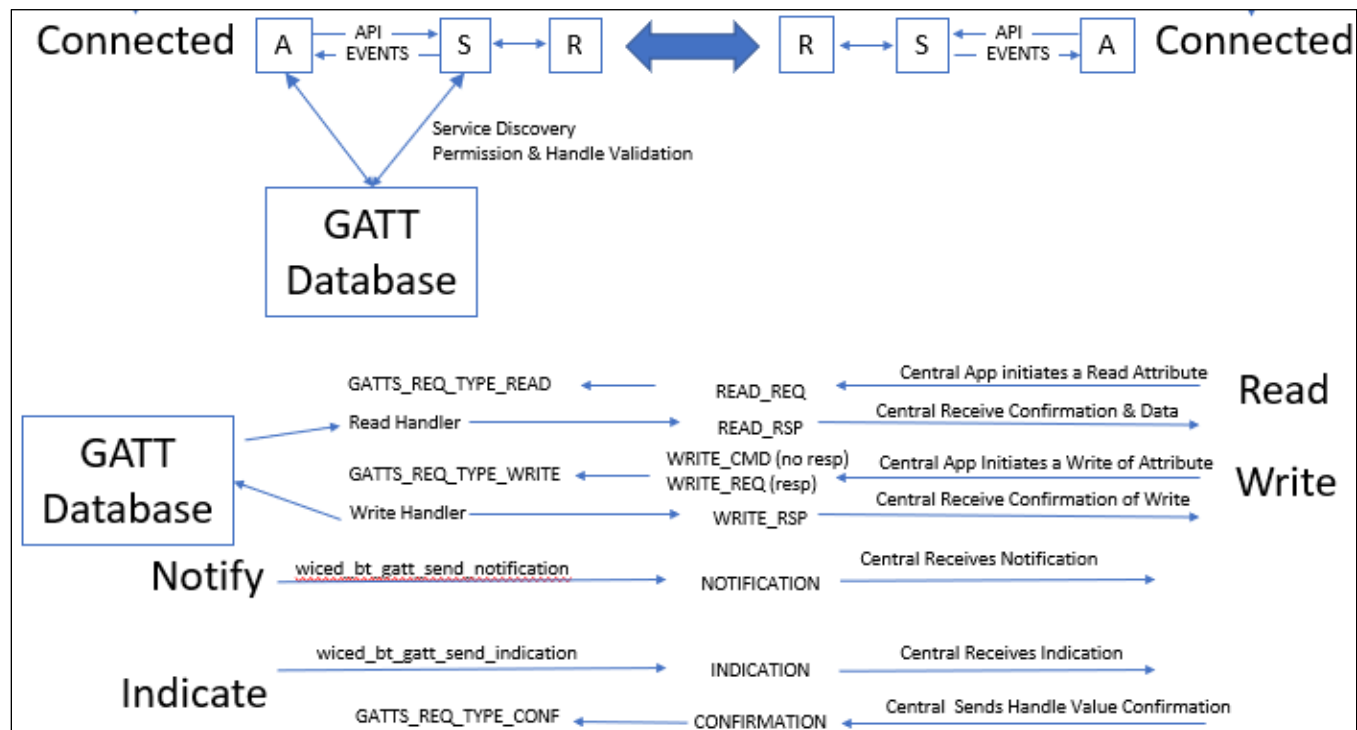
Now that you are connected you need to be able to exchange data. In the world of Bluetooth® LE this happens via the Attribute Protocol (ATT). The basic ATT protocol has 4 types of transactions: Read & Write which are initiated by the Client and Notify & Indicate which are initiated by the Server.

ATT Protocol transactions are all keyed to a very simple database called the GATT database which typically (but not always) resides on the Peripheral. The side that maintains the GATT database is commonly known as the GATT Server or just Server. Likewise, the side that makes requests of the database is commonly known as the GATT Client or just Client. The Client is typically (but not always) the Central. So, in the most common case, the Peripheral is the Server and the Central is the Client. This relationship may be confusing so be careful.

You can think of the GATT Database as a simple table. The columns in the table are:

- Handle – 16-bit numeric primary key for the row
- Type – A Bluetooth® SIG specified number (called a UUID) that describes the Data
- Data – An array of 1-x bytes
- Permission Flags

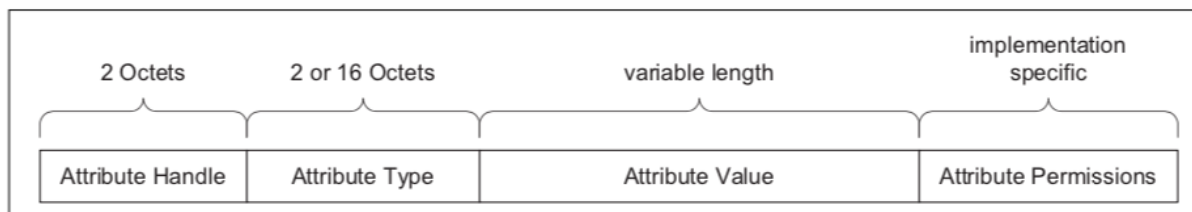
I'll talk in more detail about the GATT database implementation later. With all of that, here is the final section of the big picture:



## 3.2 Attributes, the Generic Attribute Profile & GATT Database

### 3.2.1 Attributes

As mentioned earlier, the GATT Database is just a table with up to 65535 rows. Each row in the table represents one Attribute and contains a Handle, a Type, a Value and Permissions.



(This figure is taken from the Bluetooth® Specification)

The Handle is a 16-bit unique number to represent that row in the database. These numbers are assigned by you, the firmware developer, and have no meaning outside of your application. You can think of the Handle as the database primary key.

The Type of each row in the database is identified with a Universally Unique Identifier (UUID). The UUID scheme has two interesting features:

- Attribute UUIDs are 2 octets or 16 octets long. You can purchase a 2-octet UUID from the SIG for around \$5K
- Some UUIDs are defined by the Bluetooth® SIG and have specific meanings and some can be defined by your application firmware to have a custom meaning.

In the Bluetooth® spec they frequently refer to UUIDs by a name surrounded by « ». To figure out the actual hex value for that name you need to look at the [assigned numbers](#) table on the Bluetooth® SIG website. Also, most of the common UUIDs are inserted for you into the right place by the ModusToolbox™ tools (more on this later).

The Permissions for Attributes tell the Stack what it can and cannot do in response to requests from the Central/Client. The Permissions are just a bit field specifying Read, Write, Encryption, Authentication, and Authorization. The Central/Client can't read the permission directly, meaning if there is a permission problem the Peripheral/Server just responds with a rejection message. The Bluetooth® configurator helps you get the Permissions set correctly when you make the database, and the Stack takes care of enforcing them.

### 3.2.2 Profiles, Services, Characteristics

The GATT Database is "flat" – it's just a bunch of rows with one Attribute per row. This creates a problem because a totally flat organization is painful to use, so the Bluetooth® SIG created a semantic hierarchy. The hierarchy has two levels: Services and Characteristics. Note that Services and Characteristics are just different types of Attributes.

In addition to Services and Characteristics, there are also Profiles which are a previously agreed to, or Bluetooth® SIG specified related set of data and functions that a device can perform. If two devices implement the same Profile, they are guaranteed to interoperate. A Profile contains one or more Services.

A Service is just a group of logically related Characteristics, and a Characteristic is just a value (represented as an Attribute) with zero, one or more additional Attributes to hold meta data (e.g. units). These meta-data Attributes are typically called Characteristic Descriptors.

For instance, a Battery Service could have one Characteristic - the battery level (0-100 %) - or you might make a more complicated Service, for instance a CapSense Service with a bunch of CapSense widgets represented as Characteristics.

There are two Services that are required for every Bluetooth® LE device. These are the Generic Attribute Service and the Generic Access Service. Other Services will also be included depending on what the device does.

Each of the different Attribute Types (i.e. Service, Characteristic, etc.) uses the Attribute Value field to mean different things.

### 3.2.3 Service Declaration

#### 3.2.4 Service Declaration in the GATT DB

To declare a Service, you need to put one Attribute in the GATT Database. That row just has a Handle, a type of 0x2800 (which means this GATT Attribute is a declaration of a Primary Service), the Attribute Value (which in this case is just the UUID of the Service) and the Attribute Permission.

Attribute Handle	Attribute Type	Attribute Value	Attribute Permission
0xNNNN	0x2800 – UUID for «Primary Service» OR 0x2801 for «Secondary Service»	16-bit Bluetooth UUID or 128-bit UUID for Service	Read Only, No Authentication, No Authorization

GATT Row for a Service (This figure is taken from the Bluetooth® Specification)

For the Bluetooth® defined Services, you are obligated to implement the required Characteristics that go with that Service. You are also allowed implement custom Services that can contain whatever Characteristics you want. The Characteristics that belong to a Service must be in the GATT database after the declaration for the Service that they belong to and before the next Service declaration.

You can also include all the Characteristics from another Service into a new Service by declaring an Include Service.

Attribute Handle	Attribute Type	Attribute Value			Attribute Permission
0xNNNN	0x2802 – UUID for «Include»	Included Service Attribute Handle	End Group Handle	Service UUID	Read Only, No Authentication, No Authorization

GATT Row for an Included Service (This figure is taken from the Bluetooth® Specification)

### 3.2.5 Characteristic Declaration

To declare a Characteristic, you are required to create a minimum of two Attributes: the Characteristic Declaration (0x2803) and the Characteristic Value. The Characteristic Declaration creates the property in the GATT database, specifies the UUID and configures the Properties for the Characteristic (which controls permissions for the characteristic as you will see in a minute). This Attribute does not contain the actual value of the characteristic, just the handle of the Attribute (called the Characteristic Value Attribute Handle) that holds the value.

Attribute Handle	Attribute Types	Attribute Value			Attribute Permissions
0xNNNN	0x2803—UUID for «Characteristic»	Characteristic Properties	Characteristic Value Attribute Handle	Characteristic UUID	Read Only, No Authentication, No Authorization

GATT Row for a Characteristic Declaration (This figure is taken from the Bluetooth® Specification)

Each Characteristic has a set of Properties that define what the Central/Client can do with the Characteristic. These Characteristic Properties are used by the Stack to enforce access to the Characteristic by the Client (e.g. Read/Write) and they can be read by the Client to know what they can do. The Properties include:

- Broadcast – The Characteristic may be in an Advertising broadcast
- Read – The Client/Central can read the Characteristic
- Write Without Response – The Client/Central can write to the Characteristic (and that transaction does not require a response by the Server/Peripheral)
- Write – The Client/Central can write to the Characteristic and it requires a response from the Peripheral/Server
- Notify – The Client can request Notifications from the Server of Characteristic values changes with no response required by the Client/Central. The Stack will send notifications from the GATT server when a database characteristic changes.
- Indicate – The Client can ask for Indications from the Server of Characteristic value changes and requires a response by the Client/Central. The Stack sends indications from the GATT server when a database characteristic changes and waits for the client to send the response.
- Authenticated Signed Writes – The client can perform digitally signed writes
- Extended Properties – Indicates the existence of more Properties (mostly unused)

When you configure the Characteristic Properties, you must ensure that they are consistent with the Attribute Permissions of the Characteristic Value.

The Characteristic Value Attribute holds the value of the Characteristic in addition to the UUID. It is typically the next row in the database after the Characteristic Declaration Attribute.

Attribute Handle	Attribute Type	Attribute Value	Attribute Permissions
0xNNNN	0xuuuu – 16-bit Bluetooth UUID or 128-bit UUID for Characteristic UUID	Characteristic Value	Higher layer profile or implementation specific

GATT Row for a Characteristic Value (This figure is taken from the Bluetooth® Specification)

There are several other interesting Characteristic Attribute Types which will be discussed in a later chapter.

### 3.3 Creating a simple Bluetooth® LE peripheral

Now that you understand the basic system lifecycle and GATT data format, we will create a simple Bluetooth® LE peripheral application from a template. The application we create will have one custom service called "PSoC" with one characteristic called "LED" that can be read and written. When the Client writes a value into the Characteristic, the application firmware will just write that value into the GPIO driving the LED. That will allow you to turn the LED off (by writing 0) or on (by writing anything other than 0) from a BLE Central.

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

#### 3.3.1 Bluetooth® configurator

You will do this in **Error! Reference source not found.**

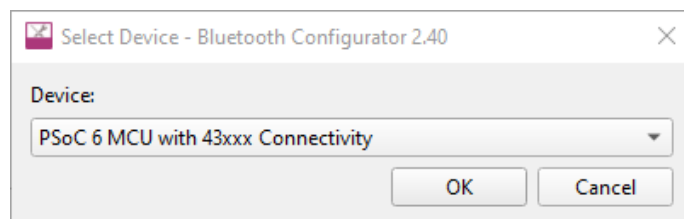
The Bluetooth® configurator is a ModusToolbox™ tool that will setup device configuration, advertising packets, scan response packets, and a customized GATT database for your application.

You can launch the tool in a few different ways:

- If you are using the Eclipse IDE for ModusToolbox™, click the link in the Quick Panel
- If you are using the CLI, use the command `make config_bt` from the application root directory
- Run the `bt-configurator` tool from the Windows Start menu
- Run from the ModusToolbox™ tools installation directory (`<Install Directory>/ModusToolbox/tools_<version>/bt-configurator/bt-configurator.exe`)

For the Eclipse IDE, if a configuration exists, it will be opened. If a configuration does not exist it will be created using the correct device.

For the CLI, if a configuration exists, it will be opened. If a configuration does not exist, you must use the tool to create one. You will need to select the correct device (PSoC™ 6 MCU with 43xxx Connectivity).

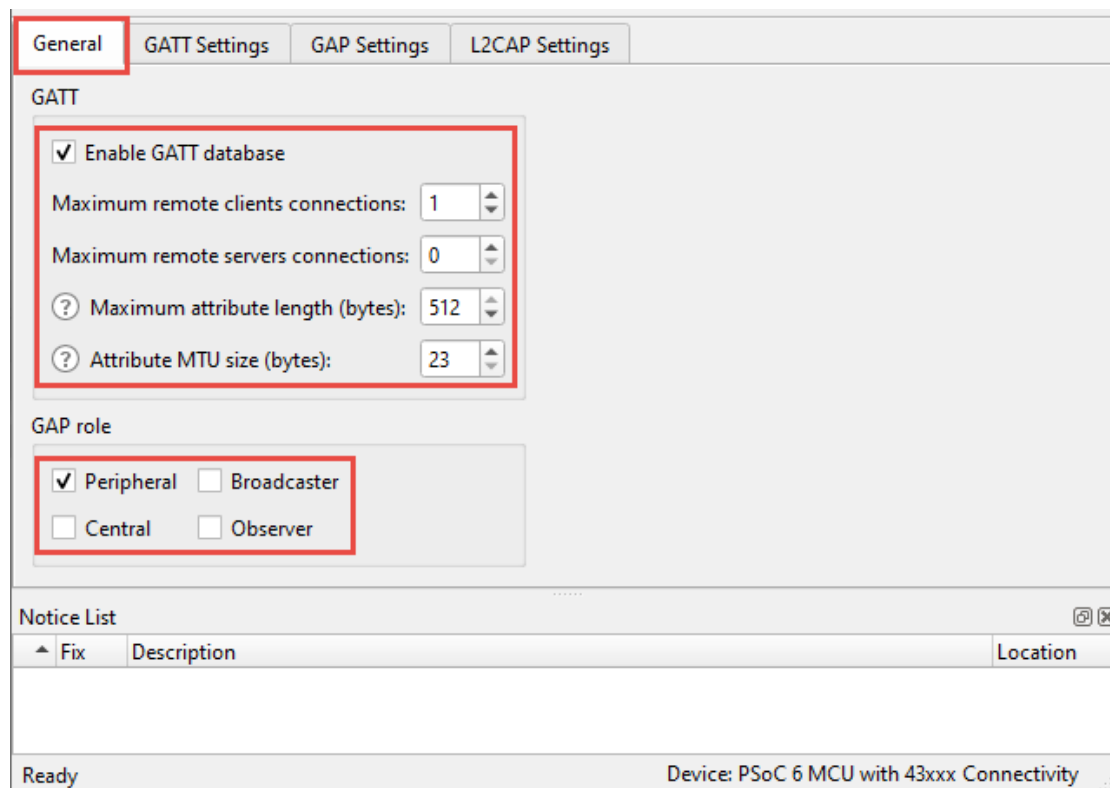


For the last two options, you must use the tool to either create a new configuration or open an existing one.

Once you have created or opened a configuration, the tool will have tabs for General settings, GATT settings, GAP settings, and L2CAP settings. We will go through these tabs one at a time.

### 3.3.1.1 General tab

As you can see below, the **General** tab allows you to enable a GATT database, select the maximum number of client or server connections, set length restrictions on the attribute length and maximum transmission unit (MTU), and setup the different roles the device will support. In our case, all of the defaults will work - since we are creating a peripheral that will allow at most 1 client to connect at a time and will have a GATT database.



The screenshot shows the 'General' tab of the Bluetooth configuration interface. The 'GATT' section is highlighted with a red box, showing the following settings:

- ☒ Enable GATT database
- Maximum remote clients connections: 1
- Maximum remote servers connections: 0
- Maximum attribute length (bytes): 512
- Attribute MTU size (bytes): 23

The 'GAP role' section is also highlighted with a red box, showing the following settings:

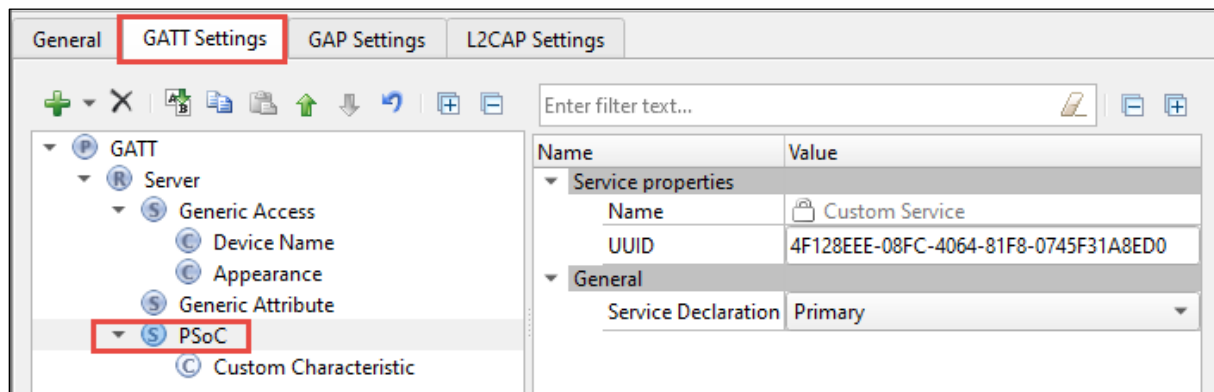
- ☒ Peripheral ☐ Broadcaster
- ☐ Central ☐ Observer

The 'Notice List' section at the bottom shows a table with columns: Fix, Description, and Location. The status bar at the bottom indicates 'Ready' and 'Device: PSoC 6 MCU with 43xxx Connectivity'.

### 3.3.1.2 GATT Settings tab

In the **GATT Settings** tab, we want to set up the GATT database that our device needs. Note that the configurator already has the required Generic Access and Generic Attribute Services defined so our first step is to create our custom Service called "PSoC". To do this:

1. Right-click on **Server** and choose **Add Service > Custom Service** (it is near the bottom of the list). A Custom Service entry appears in the GATT database.
2. Right-click on the custom service and select **Rename**. Call the service "PSoC".
3. 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.



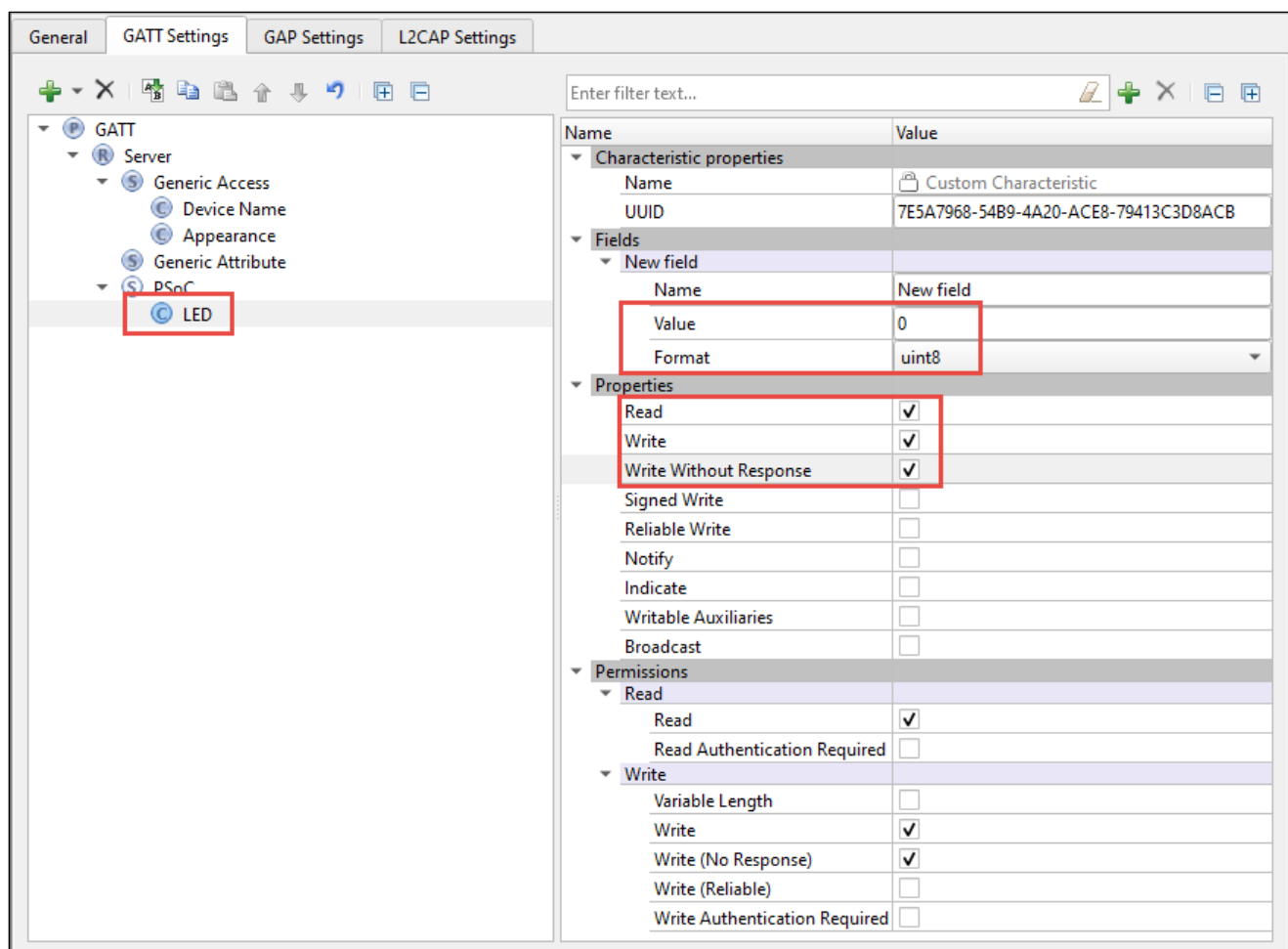
The Service includes a Characteristic, which we are going to use to control the LED. To do this you:

1. Right-click on **Custom Characteristic** under the "PSoC" Service and **Rename** to "LED".
2. Change the format from `utf8s` (which requires a length) to `uint8` (which has a length of 1 by definition).
3. Change the value of the LED characteristic to 0, which we will take to mean "OFF". This will be the initial value.
4. We want the client to be able to Read and Write this Characteristic, so under **Properties**, enable **Read**, **Write** and **Write Without Response**.

*Note: 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.*

*Note: Since we selected both **Write** and **Write Without Response**, a client will be able to use either a `WRITE_REQ` or a `WRITE_CMD` write a new value to the server.*

5. Again, keep the randomly assigned UUID for the Characteristic just like you did for the Service UUID.



The screenshot shows the ModusToolbox GATT Settings window. On the left, the 'LED' characteristic is selected under the 'PSoC' service. The right pane displays the configuration for this characteristic:

Name	Value
<b>Characteristic properties</b>	
Name	Custom Characteristic
UUID	7E5A7968-54B9-4A20-ACE8-79413C3D8ACB
<b>Fields</b>	
New field	
Name	New field
Value	0
Format	uint8
<b>Properties</b>	
Read	<input checked="" type="checkbox"/>
Write	<input checked="" type="checkbox"/>
Write Without Response	<input checked="" type="checkbox"/>
Signed Write	<input type="checkbox"/>
Reliable Write	<input type="checkbox"/>
Notify	<input type="checkbox"/>
Indicate	<input type="checkbox"/>
Writable Auxiliaries	<input type="checkbox"/>
Broadcast	<input type="checkbox"/>
<b>Permissions</b>	
Read	
Read	<input checked="" type="checkbox"/>
Read Authentication Required	<input type="checkbox"/>
Write	
Variable Length	<input type="checkbox"/>
Write	<input checked="" type="checkbox"/>
Write (No Response)	<input checked="" type="checkbox"/>
Write (Reliable)	<input type="checkbox"/>
Write Authentication Required	<input type="checkbox"/>



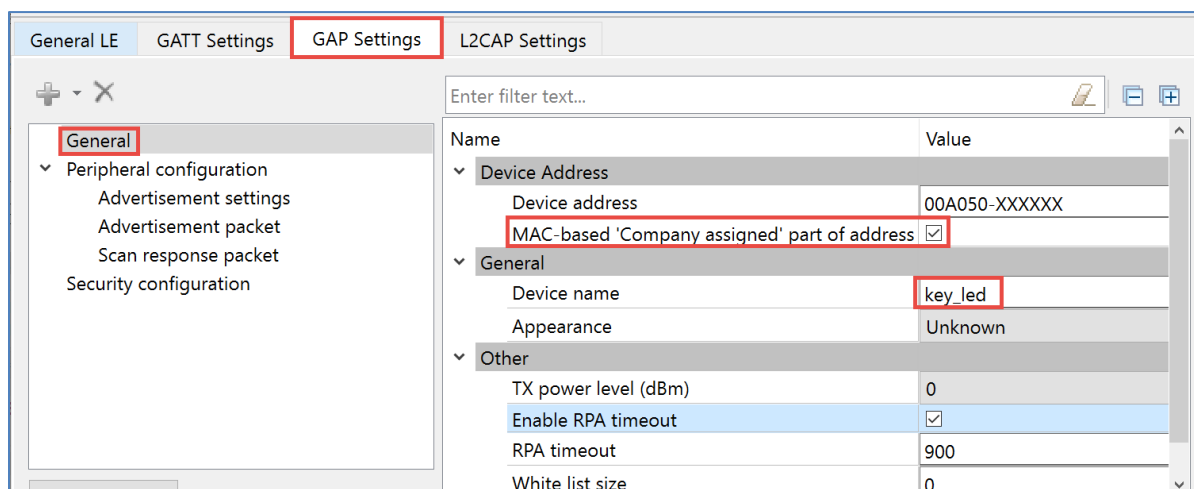
### 3.3.1.3 GAP Settings tab

Next up is the **GAP Settings** tab. This tab has several sub-sections depending on what was chosen on the **General** tab. The first sub-section (unfortunately also called General) allows you to configure overall device settings such as the device name, device address, appearance, etc.

First, we will enter a **Device name**. 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.

Next click the box **MAC-based 'Company assigned' part of address**. This tells the configurator to generate values for the last 3 bytes of the device's Bluetooth® device address (BDA) – indicated by X's - by using the computer's MAC address as a seed for a random number generator. This means that any device created on the same computer will have the same BDA, but devices created on different computers should have different BDAs. This is useful for a class setting so that each student will have a different BDA, but it is not generally used in a production environment.

All other settings will be left with their default values for now.

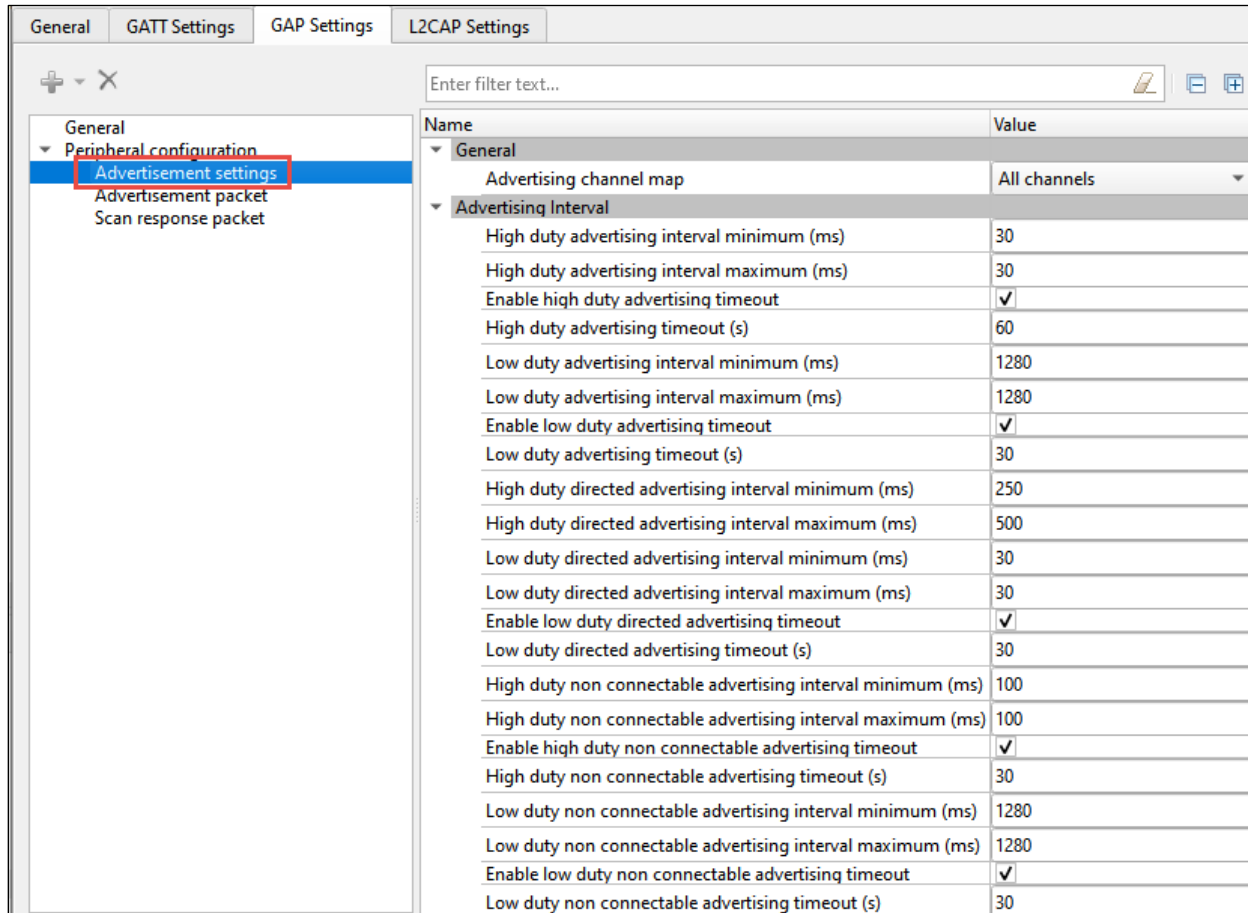


Name	Value
Device Address	
Device address	00A050-XXXXXX
MAC-based 'Company assigned' part of address	<input checked="" type="checkbox"/>
General	
Device name	key_led
Appearance	Unknown
Other	
TX power level (dBm)	0
Enable RPA timeout	<input checked="" type="checkbox"/>
RPA timeout	900
White list size	0

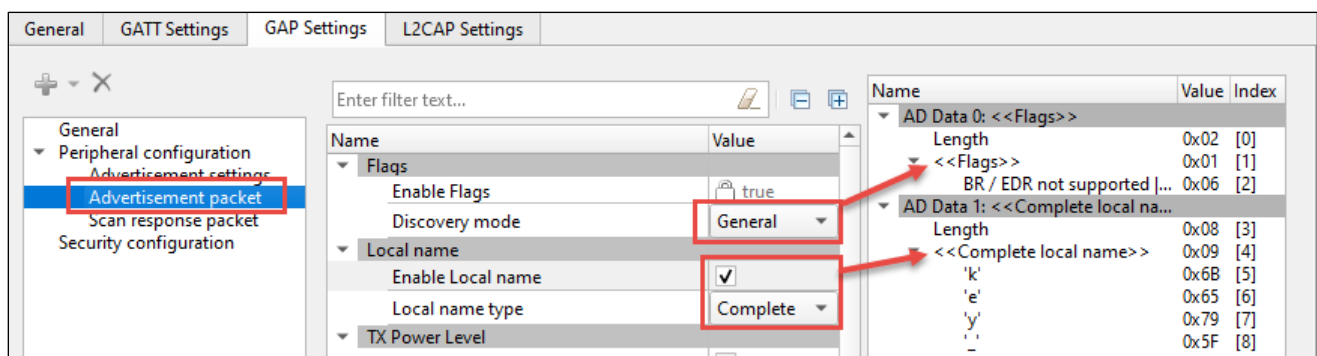
**Note:** The "Enable RPA timeout" selection will be enabled in all of our exercises because some iOS devices require that RPA is enabled for peripherals to function properly. RPA means that the device will advertise a random address called a "resolvable private address" instead of the device address specified in the configurator. The device address that is specified will be used as the device's public identity address. This will be covered in the chapter on security and privacy.

The next section is the **Peripheral configuration**. It has sections to set the advertisement settings, advertisement packet, and scan response packet.

The **Advertisement settings** section allows you to specify intervals and timeouts for each type of advertising. For this example, we will use the default advertisement setting values.



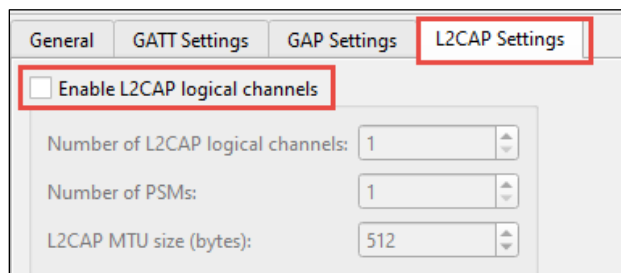
The **Advertisement packet** and **Scan response packet** sections allow you to setup advertisement and scan response data. The middle panel is where you make your selections and the right panel shows what the resulting packet contents will be. For our example, we will set the flags to allow general discovery (which is the default) and we will include the complete local device name in the advertising packet. This will allow you to easily identify your device when you want to connect to it. Notice that the advertising packet has 3 bytes for the flags (length, type, and value) and 9 bytes for the local name (length, type, and 7 bytes for the name).



We will not include a scan response packet yet.

### 3.3.1.4 L2CAP Settings tab

The L2CAP Settings tab allows you to enable/disable L2CAP channels and to configure their settings. We will leave the L2CAP channels disabled for now.



### 3.3.1.5 Generated code

Once you have setup everything the way you want, click the **Save** button to save the file *design.cybt* in the application root directory.

*Note: 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.*

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.

The configurator generates 3 sets of files and a timestamp. They are:

File Name	Purpose
cycfg_bt_settings.c cycfg_bt_settings.h	Data from the General tab such as client max links, advertisement settings from the GAP Settings tab, and data from the L2CAP Settings tab.
cycfg_gap.c cycfg_gap.h	Data from the GAP Settings tab including device address, device name, and advertising packet data.
cycfg_gatt_db.c cycfg_gatt_db.h	Data from the GATT Settings tab including Service and Characteristic declarations, properties, permissions, and handle assignments.
cycfg_bt.timestamp	Timestamp file used to determine if the generated files are up-to-date with the configuration.

Even though you won't modify these files by hand, you will need to use values from them in the firmware.

### 3.3.2 Editing the firmware

You will do this in **Error! Reference source not found.**

We'll take a more detailed look at the firmware later, but for now we will start with a template for the exercises that provides most of the code. To modify the template firmware for our example application, the changes required are:

1. Start by opening *main.c* and including the code from the Bluetooth® configurator:

```
/* Include header files from BT configurator */
#include "cycfg_bt_settings.h"
#include "cycfg_gap.h"
#include "cycfg_gatt_db.h"
```

2. Configure the LED pin:

```
/* Initialize LED Pin */
cyhal_gpio_init(CYBSP_USER_LED, CYHAL_GPIO_DIR_OUTPUT,
CYHAL_GPIO_DRIVE_STRONG, CYBSP_LED_STATE_OFF);
```

3. Initialize the Bluetooth® platform specific configuration from the BSP and initialize the stack:

```
/* Configure platform specific settings for the BT device */
cybt_platform_config_init(&cybsp_bt_platform_cfg);

/* Initialize stack and register the callback function */
wiced_bt_stack_init (app_bt_management_callback, &wiced_bt_cfg_settings);
```

**Note:** The *cybsp\_bt\_platform\_cfg* structure sets up the UART interface between the PSoC™ and the connectivity device, the low power interface (which you will see in the low power chapter) and it provides the size of a pool of memory used by the Bluetooth® task. If you want to see this structure, it is defined in the BSP. By default, the file can be found in the application at:  
<application\_root>/libs/TARGET\_<bsp\_name>/bluetooth/cybsp\_bt\_config.c.

4. In the *app\_bt\_management\_callback* function, find the *BTM\_ENABLED\_EVT* and:

- a. Set the Bluetooth® device address to the value generated by the configurator and print it out

```
/* Set the local BDA from the value in the configurator and print it */
wiced_bt_set_local_bdaddr((uint8_t *)cy_bt_device_address, BLE_ADDR_PUBLIC);
wiced_bt_dev_read_local_addr( bda );
printf( "Local Bluetooth® Device Address: " );
print_bd_address(bda);
```

- b. Register and initialize the GATT database

```
/* Register GATT callback and initialize database*/
wiced_bt_gatt_register( app_bt_gatt_event_callback );
wiced_bt_gatt_db_init( gatt_database, gatt_database_len, NULL );
```

- c. Disable device pairing (we will add pairing in a later chapter)

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

- d. Set advertisement data to the values generated by the configurator and start advertising

```
/* Set advertisement packet and begin advertising */  
wiced_bt_ble_set_raw_advertisement_data(CY_BT_ADV_PACKET_DATA_SIZE,  
                                         cy_bt_adv_packet_data);  
wiced_bt_start_advertisements( BTM_BLE_ADVERT_UNDIRECTED_HIGH, 0, NULL );
```

5. In `app_bt_write_handler`, add a case to the existing switch statement for the LED characteristic's handle to send the newly received value to the GPIO that controls the LED (the switch is already in the template – you just need to add a new case).

```
// Add action when specified handle is written  
case HDLC_PSOC_LED_VALUE:  
    cyhal_gpio_write(CYBSP_USER_LED, app_psoc_led[0] == 0 );  
    printf( "Turn the LED %s\r\n", app_psoc_led[0] ? "ON" : "OFF" );  
    break;
```

**Note:** The code provided already puts the new value into the GATT database – what you are adding is to handle the LED state and print a message to the UART.

**Note:** The case name (`HDLC_PSOC_LED_VALUE`) is the handle for the LED characteristic's value attribute which is defined in `cycfg_gatt_db.h`. The name of the array that holds the value (`app_psoc_led`) is defined in `cycfg_gatt_db.c`.

6. In `app_bt_gatt_req_read_handler`, `app_bt_gatt_req_read_by_type_handler` and `app_bt_gatt_req_read_multi_handler`, add a case to the existing switch statement to print the state of the LED to the UART (the switch is already in the template – you just need to add a new case). This event will occur whenever the Central reads the LED characteristic.

```
// Add action when specified handle is read  
case HDLC_PSOC_LED_VALUE:  
    printf( "LED is %s\r\n", app_psoc_led[0] ? "ON" : "OFF" );  
    break;
```

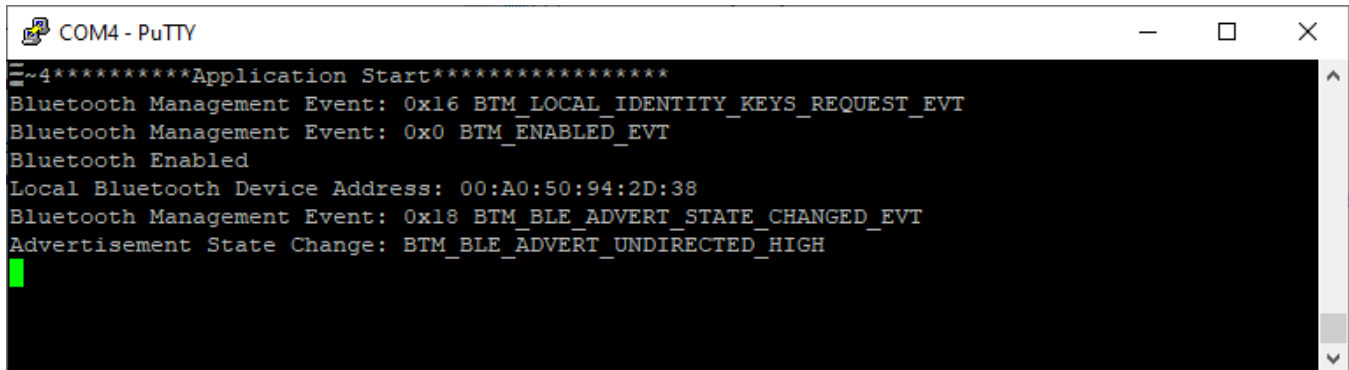
**Note:** The code uses the GATT database value, not the state of the LED itself (which is active low), and so non-zero implies “on” and zero means “off”.

**Note:** The read is already handled by the code that is already provided – what you are adding is just to print a message to the UART so that it will indicate when the Client reads the value.

### 3.3.3 Testing the application

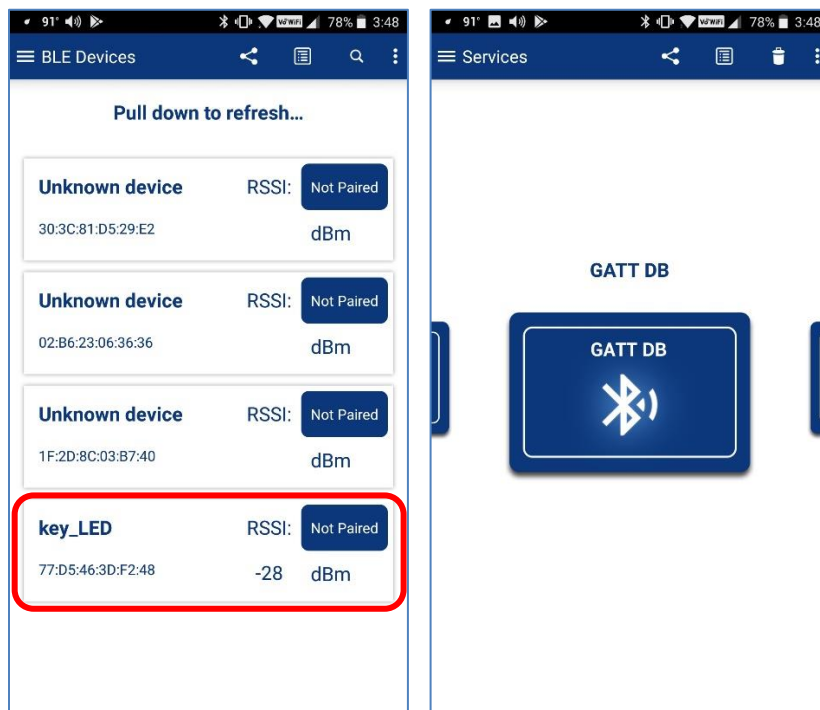
You will do this in **Error! Reference source not found.**

Start up a UART serial terminal emulator with a baud rate of 115200, then build and program your kit. When the application firmware starts up you see some messages. If you need a refresher on using a serial terminal emulator, see ModusToolbox™ Level 1 Getting Started class, Tools chapter, Serial Terminal Emulator section.

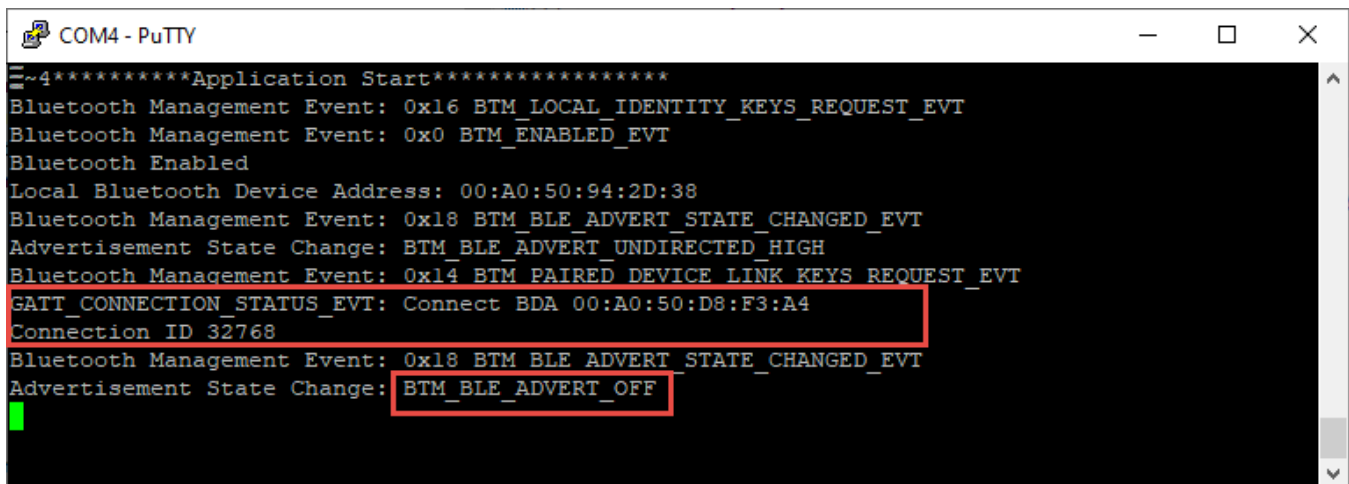


```
COM4 - PuTTY
~4*****Application Start*****
Bluetooth Management Event: 0x16 BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT
Bluetooth Management Event: 0x0 BTM_ENABLED_EVT
Bluetooth Enabled
Local Bluetooth Device Address: 00:A0:50:94:2D:38
Bluetooth Management Event: 0x18 BTM_BLE_ADVERT_STATE_CHANGED_EVT
Advertisement State Change: BTM_BLE_ADVERT_UNDIRECTED_HIGH
█
```

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

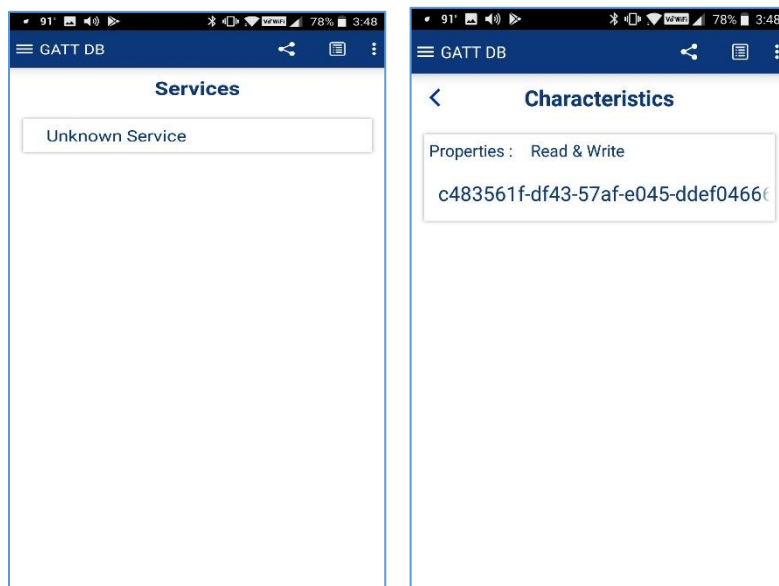


On the terminal window, you will see that there has been a connection and the advertising has stopped.



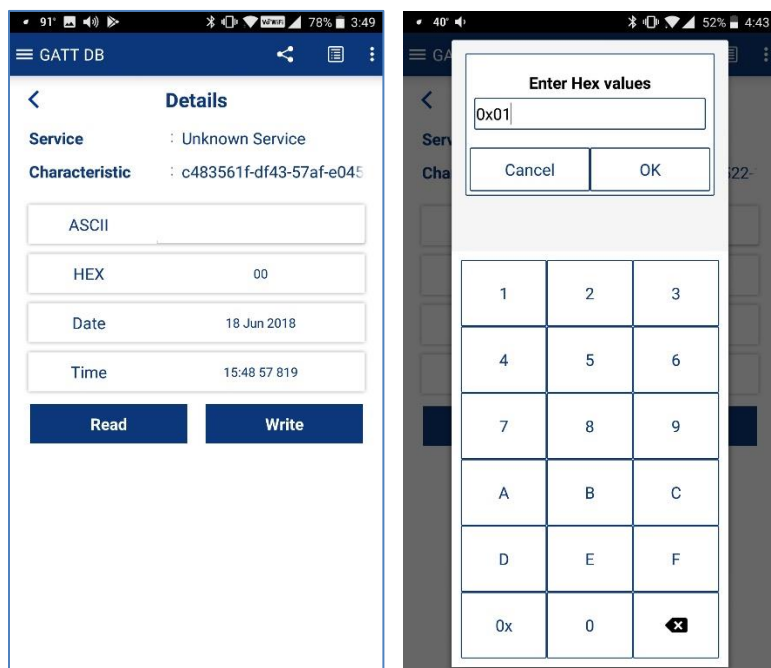
```
~4*****Application Start*****~4
Bluetooth Management Event: 0x16 BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT
Bluetooth Management Event: 0x0 BTM_ENABLED_EVT
Bluetooth Enabled
Local Bluetooth Device Address: 00:A0:50:94:2D:38
Bluetooth Management Event: 0x18 BTM_BLE_ADVERT_STATE_CHANGED_EVT
Advertisement State Change: BTM_BLE_ADVERT_UNDIRECTED_HIGH
Bluetooth Management Event: 0x14 BTM_PAIRING_DEVICE_LINK_KEYS_REQUEST_EVT
GATT_CONNECTION_STATUS_EVT: Connect BDA 00:A0:50:D8:F3:A4
Connection ID 32768
Bluetooth Management Event: 0x18 BTM_BLE_ADVERT_STATE_CHANGED_EVT
Advertisement State Change: BTM_BLE_ADVERT_OFF
```

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



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

Tap on the Service to see details about it. First, tap the Read button and you will see that the current value is 0. Now you can Write 1s or 0's into the Characteristic and you will find that the LED turns on and off accordingly.



Finally press back until AIROC™ Connect disconnects. When that happens, you will see the disconnect message in the terminal window along with advertising restarting.

```

COM4 - PuTTY
*****Application Start*****
Bluetooth Management Event: 0x16 BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT
Bluetooth Management Event: 0x0 BTM_ENABLED_EVT
Bluetooth Enabled
Local Bluetooth Device Address: 00:A0:50:94:2D:38
Bluetooth Management Event: 0x18 BTM_BLE_ADVERT_STATE_CHANGED_EVT
Advertisement State Change: BTM_BLE_ADVERT_UNDIRECTED_HIGH
Bluetooth Management Event: 0x18 BTM_BLE_ADVERT_STATE_CHANGED_EVT
Advertisement State Change: BTM_BLE_ADVERT_UNDIRECTED_LOW
Bluetooth Management Event: 0x14 BTM_PAIRING_DEVICE_LINK_KEYS_REQUEST_EVT
GATT_CONNECTION_STATUS_EVT: Connect BDA 00:A0:50:D8:F3:A4
Connection ID 32768
Bluetooth Management Event: 0x18 BTM_BLE_ADVERT_STATE_CHANGED_EVT
Advertisement State Change: BTM_BLE_ADVERT_OFF
LED is OFF
Turn the LED ON
LED is ON
Turn the LED OFF
LED is OFF
Disconnected : BDA 00:A0:50:D8:F3:A4
Connection ID '32768', Reason 'GATT CONN TERMINATE PEER USER'
Bluetooth Management Event: 0x18 BTM_BLE_ADVERT_STATE_CHANGED_EVT
Advertisement State Change: BTM_BLE_ADVERT_UNDIRECTED_HIGH

```

In the next section we will walk through the code in more detail.



## 3.4 Bluetooth® libraries and settings

### 3.4.1 Libraries

The host stack running on the PSoC™ 6 MCU requires an RTOS. By default, FreeRTOS is used. There is one top level library that must be included by the application:

- *btstack-integration* – Bluetooth® controller firmware. It contains COMPONENT directories for several different Infineon Bluetooth® families based on the device architecture. In our case, we will be using `COMPONENT_HCI-UART` since the PSoC™ 6 communicates with the CYW43012 using the HCI protocol over a UART interface. The component setting is in the BSP's *Makefile* so we don't need to set it to get the correct files.

Several additional libraries are included indirectly by *btstack-integration*:

- *btstack* – Bluetooth® host stack implementation.
- *freertos* – standard FreeRTOS library.
- *abstraction-rtos* - used to abstract RTOS functions from the specific RTOS chosen. Allows other libraries to use generic RTOS functions that get mapped to the appropriate underlying RTOS functions.
- *clib-support* – support library that provides hooks to make C library functions such as malloc and free thread-safe.

### 3.4.2 Makefile settings

Several COMPONENT settings are required in the application *Makefile* to include the appropriate library code. Specifically:

```
COMPONENTS=FREERTOS RTOS_AWARE WICED_BLE
```

- `FREERTOS` and `RTOS_AWARE` are required to pull in the correct code from the *abstraction-rtos* library.
- `WICED_BLE` is required to pull in the correct code from the *btstack* library.

## 3.5 Stack Events

Before we get to the details of the firmware, we should discuss Bluetooth® Stack events for a minute. All interactions between the Stack and the application are done using callbacks that are generated by Stack events so it is important concept to understand.

The Stack generates Events based on what is happening in the Bluetooth® world. After an event is created, the Stack will call the callback function which you registered when you turned on the Stack. Your callback firmware must look at the event code and the event parameter and take the appropriate action.

There are two classes of events: Management, and GATT. Each of these has its own callback function.

### 3.5.1 Essential Bluetooth® management events

The Stack will generate management events for lots of different things but for now we will ignore all of them except two. More of them will be covered in later chapters when we cover things like encryption, pairing, and bonding.

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.

### 3.5.2 Essential GATT Events

The Stack will also generate lots of GATT events, but the ones we need to know about are:

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.
GATT_GET_RESPONSE_BUFFER_EVT	This event occurs when the Stack needs a block of memory allocated to store a response value.
GATT_APP_BUFFER_TRANSMITTED_EVT	This event occurs when the Stack is done with a block of memory so that the buffer can be freed up.

#### 3.5.2.1 Essential GATT\_ATTRIBUTE\_REQUEST\_EVT sub-events

In addition to the GATT events described above, there are sub-events associated with the attribute request event. These include various read and write events.

## 3.6 Firmware architecture

At the very beginning of this chapter we discussed the four steps to make a basic Bluetooth® LE Peripheral:

- Turn on the Stack
- Start Advertising
- Process GATT connection Events from the Stack
- Process GATT attribute requests from the Stack such as read and write and perform memory management

The Bluetooth® template provided for this class mimics this flow.

### 3.6.1 Turn on the Stack

The main function in a Bluetooth® application typically starts by doing high level initialization of the PSoC™ MCU (`cybsp_init`), and the CYW43012 device (`cybt_platform_config_init`) along with any other initial setup such as enabling interrupts, setting up a debug UART, initializing pins, etc.

At that point in the proceedings, your application firmware is responsible for turning on the Stack and making a connection to the WICED radio. This is done with the API call `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.

The rest of main may setup other RTOS tasks if necessary (the tasks needed for the Stack are started by the Stack initialization) and then it launches the RTOS scheduler. At this point, main is done.

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 the stack it is ready to go, it generates the `BTM_ENABLED_EVT` event and calls the `app_bt_management_callback` function which then processes that event. For this event, you will typically do any remaining application specific initialization, set the Bluetooth® device address, initialize the GATT database, setup pairing, setup the advertisement packet, and start advertisements.

### 3.6.2 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 doesn't do anything except to print out the new advertising state but you could add your own code here to, for instance, turn an LED off or on to indicate the connection status.

### 3.6.3 Process GATT connection events

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.

The Stack then generates a GATT event called `GATT_CONNECTION_STATUS_EVT` which is processed by the `app_bt_gatt_event_callback` function and in turn calls the `app_bt_connect_event_handler`.

The connect event handler uses the event parameter to determine if it is a connection or a disconnection. It then prints a useful message.

On a connection, the stack stops the advertising which results in another `BTM_BLE_ADVERT_STATE_CHANGED_EVT` management event, this time because advertising stopped instead of started.

### 3.6.4 Process GATT attribute requests

GATT attribute requests consist of write and read events. There are different types of read and write requests that may be received. Here we will focus on the basic `GATT_REQ_READ`, `GATT_REQ_WRITE`, and `GATT_CMD_WRITE`, but the other cases can be seen in the template.

### 3.6.4.1 Process write events

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` which calls `app_bt_gatt_event_callback`, which determines the event is `GATT_ATTRIBUTE_REQUEST_EVT`. It then in turn calls `app_bt_server_event_handler` which looks at the event data and determines that the event is a write and calls the function `app_bt_write_handler` to update the current value of the Characteristic.

The `app_bt_write_handler` function looks through that GATT Database to find the Attribute that matches the Handle requested. It then copies the value from the Stack generated request into the GATT Database. Finally, the set value function returns a code to indicate what happened - 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. If your callback function returns `WICED_BT_GATT_SUCCESS`, the Stack sends a Write response of `0x1E` to the client. If your callback returns something other than `WICED_BT_GATT_SUCCESS`, the Stack sends an error response with the error code that you chose.

There is a switch statement in `app_bt_write_handler` that can be used to perform required actions based on the Characteristic being written. For example, if it is a Characteristic to control an LED, the LED is turned on or off based on the value. The same handler function is used for both `GATT_REQ_WRITE`, and `GATT_CMD_WRITE`.

To summarize, function call hierarchy for a write is:

1. Stack calls `app_bt_gatt_event_callback` with `GATT_ATTRIBUTE_REQUEST_EVT`
2. `app_bt_gatt_event_callback` calls `app_bt_server_event_handler`
3. `app_bt_server_event_handler` detects that the request is a write
4. `app_bt_server_event_handler` calls `app_bt_write_handler`
5. An appropriate return code is generated and passed back to the Stack

### 3.6.4.2 Process read events

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_bt_gatt_event_callback`, which then calls `app_bt_server_event_handler`. The code for this event looks at the event parameter and determines that it is one of the different read types. It then calls the appropriate function (`app_bt_gatt_req_read_handler`, `app_bt_gatt_req_read_by_type_handler`, `app_bt_gatt_req_read_multi_handler`). Each of these functions use `app_bt_find_by_handle` to find the location of the Characteristic value in the GATT database.

Once the correct location is found, the data is copied out of the GATT Database into the location requested by the Stack.

Finally, a return code is generated to indicate what happened - either `WICED_BT_GATT_SUCCESS`, 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 */
}
```

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_bt_gatt_event_callback` with `GATT_ATTRIBUTE_REQUEST_EVT`
2. `app_bt_gatt_event_callback` calls `app_bt_server_event_handler`
3. `app_bt_server_event_handler` detects that the request is a read
4. `app_bt_server_event_handler` calls the appropriate read handler
5. The read handler calls `app_bt_find_by_handle` to find the location in the GATT database
6. The data is copied to the location where the Stack expects to find it
7. An appropriate return code is generated and passed back to the Stack

### 3.6.5 Perform Memory Management

The last two events that the Stack sends to the GATT event callback are for memory management. The Stack will inform the application when it needs a memory buffer to store data so that it can pass back and forth to/from the application. The Stack does this with a GATT callback of type `GATT_GET_RESPONSE_BUFFER_EVT`. For this event, the application calls `malloc` to allocate a buffer of the size requested by the stack and sets up a pointer to a function that will be used later to free that buffer.

Once the memory buffer is no longer needed, the Stack generates a callback of type `GATT_APP_BUFFER_TRENASMITTED_EVT`. For this event, the application frees the specified buffer by calling the function that was set up in the prior event.

### 3.6.6 Function list

In addition to the functions described above, the template has several other functions. Some of these are in `main.c` while others are in `app_bt_utils.c`. The full list of functions is:

File	Function	Purpose
<code>main.c</code>	<code>main</code>	Main entry point. Initialize PSoC™ MCU and 43012, initialize stack, start RTOS scheduler.
<code>main.c</code>	<code>app_bt_management_callback</code>	Callback function for Bluetooth® stack management events. The <code>BTM_ENABLED_EVT</code> is called as soon as the stack is initialized so all other functionality is usually setup and launched from that event.
<code>main.c</code>	<code>app_bt_gatt_event_callback</code>	Callback function for GATT events.
<code>main.c</code>	<code>app_bt_connect_event_handler</code>	The GATT callback uses this function for connect/disconnect events.
<code>main.c</code>	<code>app_bt_server_event_handler</code>	The GATT callback uses this function when a GATT attribute request event occurs. This includes read, write, etc.
<code>main.c</code>	<code>app_bt_write_handler</code>	The GATT server event handler calls this function for GATT writes.
<code>main.c</code>	<code>app_bt_gatt_req_read_handler</code> <code>app_bt_gatt_req_read_by_type_handler</code> <code>app_bt_gatt_req_read_multi_handler</code>	The GATT server event handler calls these functions for GATT reads.
<code>main.c</code>	<code>app_bt_find_by_handle</code>	The GATT read handler functions call this function to identify the location of an attribute in the GATT database from its handle.
<code>main.c</code>	<code>app_bt_alloc_buffer</code> <code>app_bt_free_buffer</code>	These functions are called to allocate and free up memory that the Stack needs.
<code>app_bt_utils.c</code>	<code>print_bd_address</code>	This function prints a Bluetooth® Device Address in an easy to read format.
<code>app_bt_utils.c</code>	<code>get_bt_event_name</code> <code>get_bt_advert_mode_name</code> <code>get_bt_gatt_disconne_reason_name</code> <code>get_bt_gatt_status_name</code>	These functions convert return codes to their equivalent string to allow for more informative messages.

## 3.7 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. At the very least it is worth understanding the handles in *cycfg\_gatt\_db.h* since you will often need to use the handle names in the application code.

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 Bluetooth® Events.

The implementation of the GATT Database is simple generic "C" (obviously) and is composed logically of three parts.

- 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.

### 3.7.1 gatt\_database[] array

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".

#### 3.7.1.1 Services

Services are created using 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 0x01
```



```
/* Service Generic Attribute */
#define HDLS_GATT 0x06

/* Service BT101 */
#define HDLS_PSOC 0x07
```

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_PSOC 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*.

### 3.7.1.2 Characteristics

Characteristics are created using 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_PSOC_LED 0x08
#define HDLC_PSOC_LED_VALUE 0x09
```

**Note:** The *\_VALUE* parameter is the Handle of the Attribute that will hold the Characteristic's Value so that's the one you will most often need to use in the application.

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*:

```
#define GATTDB_CHAR_PROP_BROADCAST (0x1 << 0)
#define GATTDB_CHAR_PROP_READ (0x1 << 1)
#define GATTDB_CHAR_PROP_WRITE_NO_RESPONSE (0x1 << 2)
#define GATTDB_CHAR_PROP_WRITE (0x1 << 3)
#define GATTDB_CHAR_PROP_NOTIFY (0x1 << 4)
#define GATTDB_CHAR_PROP_INDICATE (0x1 << 5)
#define GATTDB_CHAR_PROP_AUTHD_WRITES (0x1 << 6)
#define GATTDB_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*.

```
#define GATTDB_PERM_NONE (0x00)
#define GATTDB_PERM_VARIABLE_LENGTH (0x1 << 0)
#define GATTDB_PERM_READABLE (0x1 << 1)
#define GATTDB_PERM_WRITE_CMD (0x1 << 2)
#define GATTDB_PERM_WRITE_REQ (0x1 << 3)
#define GATTDB_PERM_AUTH_READABLE (0x1 << 4)
#define GATTDB_PERM_RELIABLE_WRITE (0x1 << 5)
#define GATTDB_PERM_AUTH_WRITABLE (0x1 << 6)
```

```
#define GATTDDB_PERM_WRITABLE (GATTDDB_PERM_WRITE_CMD | GATTDDB_PERM_WRITE_REQ |  
GATTDDB_PERM_AUTH_WRITABLE)  
#define GATTDDB_PERM_MASK (0x7f)  
#define GATTDDB_PERM_SERVICE_UUID_128 (0x1 << 7)
```

### 3.7.2 gatt\_db\_ext\_attr\_tbl

The `gatt_database` array just defines the structure – it 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, an 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, 7, 7, app_gap_device_name },  
    { HDLC_GAP_APPEARANCE_VALUE, 2, 2, app_gap_appearance },  
    { HDLC_PSOC_LED_VALUE, 1, 1, app_psoc_led },  
};
```

### 3.7.3 uint8\_t Arrays for the Values

The 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.

```
/* *****  
 * 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_psoc_led[] = {0x00u, };
```

From the above you can see that when you want to access the value of the LED Characteristic for the "PSoC" Service, you will find the data in `app_psoc_led[0]`.

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

## 3.8 Scan Response Packets

Once a Central finds a Peripheral based on the advertising packet and wants to know more about it, the Central can look for scan response data. For a peripheral, the scan response packet looks just like an advertising packet except that the Flags field is not required. Like the advertising packet, the scan response packet is limited to 31 bytes.

You set up the scan response packet contents in the Bluetooth® configurator the same way as you do for the advertising packet. You then call the function `wiced_bt_ble_set_raw_scan_response_data` to pass that information to the Stack. That function takes the same arguments as `wiced_bt_ble_set_raw_advertisement_data` – that is, the number of advertising elements in the array, and a pointer to the array.

When you start advertising with an advertising type other than `_NONCONN_` then the Central will be able to read your scan response data. For example, `_DISCOVERABLE_` will allow the scan response to be read but will not allow connections and `_UNDIRECTED_` will allow the scan response to be read and will allow connections.

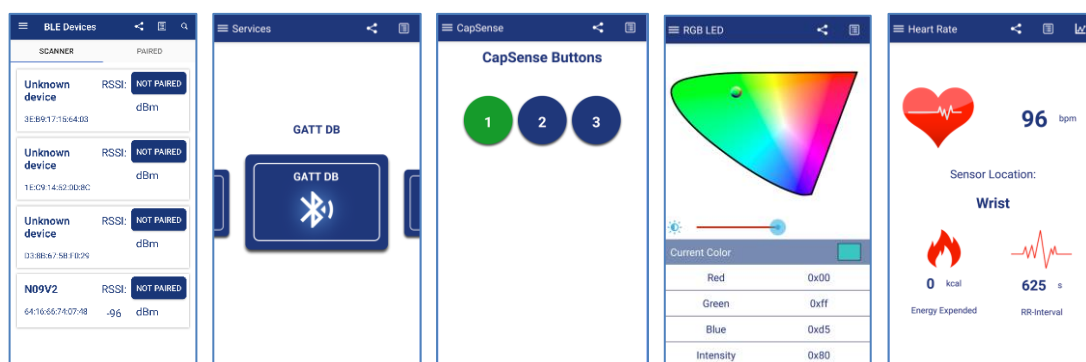
## 3.9 AIROC™ Bluetooth® Connect

Infineon provides a mobile device application for Android and iOS called AIROC™ Bluetooth® Connect (or just AIROC™ Connect) which can be used to scan, connect, and interact with services, characteristics, and attributes of Bluetooth® LE devices.

There are other utilities available for iOS and Android (such as Lightblue) which will also work. Feel free to use one of those if you are more comfortable with it.

The app is available on the Google Play store and the Apple App store. It can connect and interact with any connectable Bluetooth® LE device. It supports specialized screens for many of the Bluetooth® LE adopted services and a few Infineon custom services such as CapSense and RGB LED control. In addition, there is a GATT database browser that can be used to read and write attributes for all services even if they are not supported with specialized screens.

The images below are from the Android version of the app. The iOS version is similar but not identical.



**Note:** When you are scanning for devices, if the list is too long you can enter part of your device's name as a filter by tapping on the magnifying glass icon at the top of the screen.

Documentation of the Infineon custom profiles supported by the tool can be found in the Downloads section on the following website:

<https://www.infineon.com/cms/en/design-support/tools/utilities/wireless-connectivity>

## 3.10 Exercises

### Exercise 1: Simple Bluetooth® LE Peripheral

For this exercise, you will recreate the simple Bluetooth® LE peripheral described earlier in this chapter and test it using AIROC™ Connect. As a reminder, it will have a Service called "PSoC" with a Characteristic called "LED" that you can read/write from the client. Writing the value will cause the LED to turn on or off.



1. Use project creator to create a new ModusToolbox™ application for the BSP you are using.

On the application template page, use the **Browse** button to specify the template application from the class files under *Templates/ch03\_ex01\_ble*.



2. Follow the instructions in section 3.3.1 to setup the Bluetooth® configuration.



3. Open *main.c* and follow the instructions in section 3.3.2 to complete the code.

*Note:* Look for the string "TODO" in *main.c* to find the locations that need changes.



4. Build the application and program your kit.



5. Follow the instructions in section 3.3.3 to test your application.

### Exercise 2: Implement a connection status LED

In this exercise, you will enhance the previous exercise to add a connection status LED. It will be:

- Off – when the device is not advertising
- Blinking – when the device is advertising
- On – when there is a connection

*Note:* The connections status LED uses *CYBSP\_USER\_LED2*. If you are using the *CY8CPROTO-062-4343W* kit, there is no *CYBSP\_USER\_LED2* present. If you want to do this exercise on that kit, you can use *P10\_0* instead of *CYBSP\_USER\_LED2* and then connect an external LED between *VTARG* and *P10\_0* on the board (they are next to each other on header J1).

#### Application Creation



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

On the application template page, use the **Browse** button to specify the completed application for exercise 1. If you did not complete exercise 1, the solution can be found in *Projects/key\_ch03\_ex01\_ble*. Name the new application *ch03\_ex02\_status*.



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



3. Observe the values for **High duty advertising timeout (s)** and **Low duty advertising timeout (s)**.

*Note: You can reduce the timeout values if you don't want to have to wait as long to see advertising stop. For example, values of 20 for both timeouts means it will take 40 seconds for advertising to stop.*

- ☐ 4. Save changes and close the configurator.
- ☐ 5. Open *main.c*.
- ☐ 6. Declare a HAL PWM object to drive the LED.
- ☐ 7. In the main function, setup the PWM to connect to CYBSP\_USER\_LED2.

Select a reasonable frequency (e.g. 2 Hz) and set the duty cycle to 100% so that LED will begin in the off state (the LED is active LOW).

Don't forget to start the PWM.

*Note: The CY8CPROTO-062-4343W kit does not have a pin called CYBSP\_USER\_LED2 so if you are using that kit, use P10\_0 instead of CYBSP\_USER\_LED2 and connect an external LED between VTARG and P10\_0 on the board.*

- ☐ 8. Declare a global `uint16_t` variable to keep track of the connection ID. Initialize it to 0.
- ☐ 9. Set/clear the connection ID variable in the `app_bt_connect_event_handler` when a connection is made/lost.

For a connection: `connection_id = p_conn_status->conn_id;`

For a disconnection: `connection_id = 0;`

- ☐ 10. Update the LED's state whenever the advertisement state changes.

In the `BTM_BLE_ADVERT_STATE_CHANGED_EVT`, change the PWM duty cycle as follows:

If `p_event_data->ble_advert_state_changed` is `BTM_BLE_ADVERT_OFF`, advertising is not running. Set the LED on or off based on `connection_id`. Note that the LED off is achieved with a duty cycle of 100% while on is a duty cycle of 0%.

If `p_event_data->ble_advert_state_changed` is not `BTM_BLE_ADVERT_OFF`, some sort of advertising is running so set the PWM to toggle at a 50% duty cycle.

## Testing

- ☐ 1. Program the application to your kit.
- ☐ 2. Use AIROC™ Connect to connect to the kit. Observe the three states: (1) not advertising; (2) advertising; and (3) connected.

*Note: You will need to wait for the high duty and low duty cycle timeouts to expire before you will see the "not advertising state". Once advertising times out, you will need to reset it to start advertising again.*

## Exercise 3: Scan Response Packet

In this exercise, you will add a scan response packet to include the UUID of the "PSoC" Service.

### Application Creation



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

On the application template page, use the **Browse** button to start from the completed application for exercise 2. If you did not complete exercise 2, the solution can be found in *Projects/key\_ch03\_ex02\_status*. Name the new application *ch03\_ex03\_response*.



2. Open the Bluetooth® configurator, go to the **GAP Settings** and set the device name to **<init>\_response**.



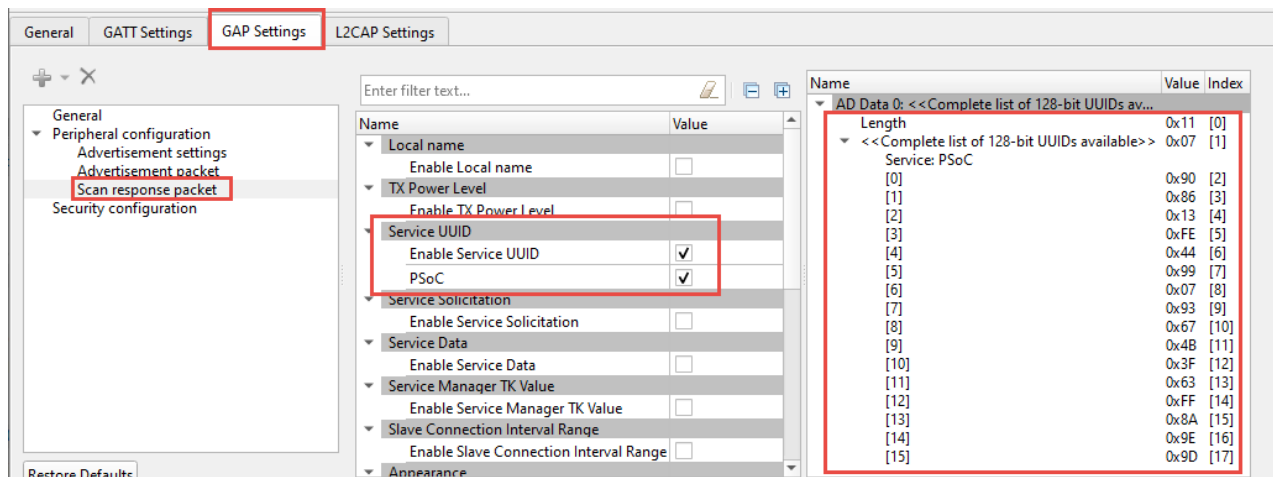
3. In the **Advertisement settings**, uncheck the box for **Enable High duty advertising timeout**.

*Note: This will cause the device to continue advertising until a connection is made. This allows you more time to look at the advertising and scan response packets without it timing out.*



4. In the **Scan response packet** section, check the boxes for **Enable Service UUID** and then **PSoC**.

*Note: You can see what the scan response packet will contain in the panel to the right in the configurator.*



5. Save changes and close the configurator.



6. Open *main.c*.



7. In the `BTM_ENABLED_EVT`, setup the scan response packet before starting advertisements. The function looks just like the function used to setup the advertisement packet.

```
wiced_bt_ble_set_raw_scan_response_data(CY_BT_SCAN_RESP_PACKET_DATA_SIZE,
                                         cy_bt_scan_resp_packet_data);
```

*Note: The name of the scan response packet and its contents be found in the file *GeneratedSource/cycfg\_gap.c*.*

## Testing

- ☐ 1. Program the application to your kit.
- ☐ 2. Open LightBlue on your phone.

*Note: AIROC™ Connect does not show scan response data so we will use LightBlue for this exercise.*

- ☐ 3. Start scanning and find your device in the list.
- ☐ 4. If you are using the Android version of LightBlue, tap on your device name to see the advertising and scan response packets.

*Note: The Android version of LightBlue will show the complete raw advertising and scan response data when you tap on the device name as the "Adv. packet". The scan response data can be seen after the advertising packet data. Note that the value in the packet is little endian so the value shown on the app will be the opposite of the service UUID.*

- ☐ 5. If you are using the iOS version of LightBlue, connect to your device and then click the link to show advertisement data.

*Note: The iOS version of LightBlue requires that you connect to the device to see the advertising packet. Once connected, click the "Show" link next to "ADVERTISEMENT DATA" to see the decoded values. You will see a line for the Service UUID since it is included in the scan response packet.*



#### **Trademarks**

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

**Published by**  
**Infineon Technologies AG**  
**81726 Munich, Germany**

**© 2023 Infineon Technologies AG.**  
**All Rights Reserved.**

#### **IMPORTANT NOTICE**

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

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

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

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

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

#### **WARNINGS**

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

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