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

3.1 Bluetooth® Stack and the device ROM 3
3.2 Bluetooth® LE system lifecycle 3
3.2.1 Turn on the Stack 5
3.2.2 Start advertising 5
3.2.3 Make a connection 8
3.2.4 Exchange data 9
3.3 Attributes, the Generic Attribute Profile & GATT Database 10
3.3.1 Attributes 10
3.3.2 Profiles, Services, Characteristics 10
3.3.3 Service Declaration 11
3.3.4 Characteristic Declaration 12
3.4 Creating a simple Bluetooth® LE peripheral 13
3.4.1 ModusToolbox™ Bluetooth® configurator 13
3.4.2 Editing the firmware 18
3.4.3 Testing the application 21
3.5 Stack Events 24
3.5.1 Essential Bluetooth® management events 24
3.5.2 Essential GATT Events 24
3.6 Firmware architecture 25
3.6.1 Turn on the Stack 25
3.6.2 Start Advertising 26
3.6.3 Process GATT connection events 26
3.6.4 Processing Client Read Events from the Stack 26
3.6.5 Processing Client Write Events from the Stack 27
3.7 GATT database implementation 29
3.7.1 gatt_database[] array 29
3.7.2 gatt_db_ext_attr_tbl 31
3.7.3 uint8_t arrays for the values 31
3.7.4 The application programming interface 32
3.8 Stack Configuration 33
3.9 Buffer Pools 35
3.10 Scan Response Packets 36
3.11 AIROC™ Bluetooth® Connect 37
3.12 Differences for btstack_v3 devices 38
3.13 Exercises 40
Exercise 1: Simple Bluetooth® LE Peripheral 40
Exercise 2: Implement a connection status LED 41
Exercise 3: Scan Response Packet 43

Document conventions

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

3.1 Bluetooth® Stack and the device ROM

The Stack in AIROC™ Bluetooth® SDK devices is contained mostly in ROM. This saves lots of space in flash for the user application. However, it means that the Stack functionality is hard coded in a particular device. In BTSDK devices, the Stack is one of two flavors called btstack_v1 and btstack_v3. Don't worry – both support the latest features and bug fixes by using small updates called "patches" that are applied during the boot sequence. However, the Stacks have a different Bluetooth® configuration and slightly different APIs, meaning the firmware you write to interact with the Stack will depend on the version that is in the device you are using.

The CYW20835 AIROC™ device used in this class has btstack_v1 in its ROM so that's what we will focus on, but at the end of the chapter we'll point out areas where the Stack and configuration are different so you will be able to write applications for btstack_v3 devices as well.

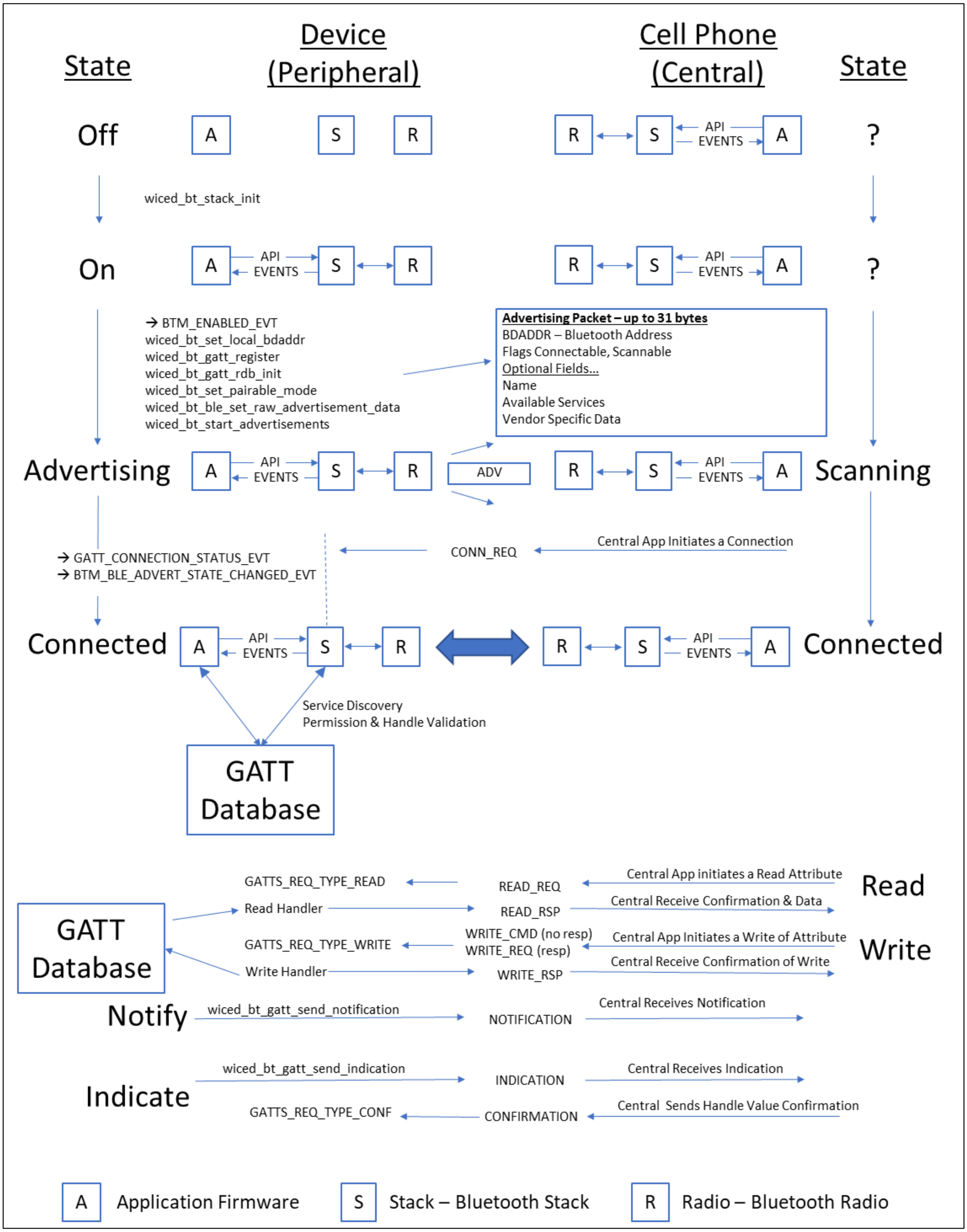
3.2 Bluetooth® LE system lifecycle

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:



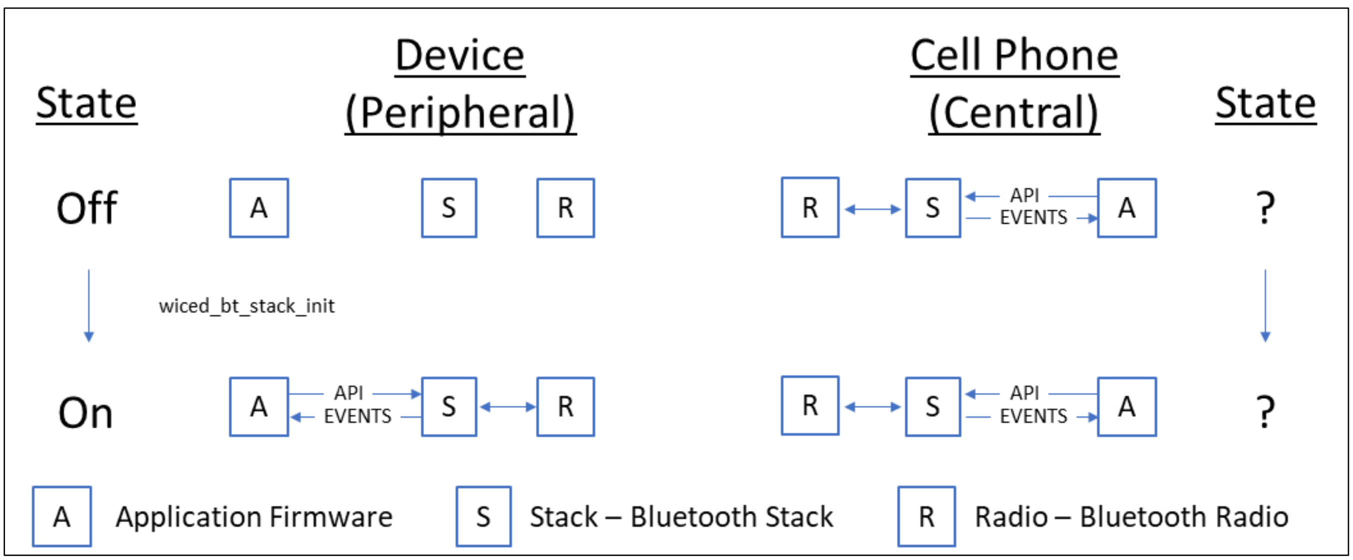
3.2.1 Turn on the Stack

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).



3.2.2 Start advertising

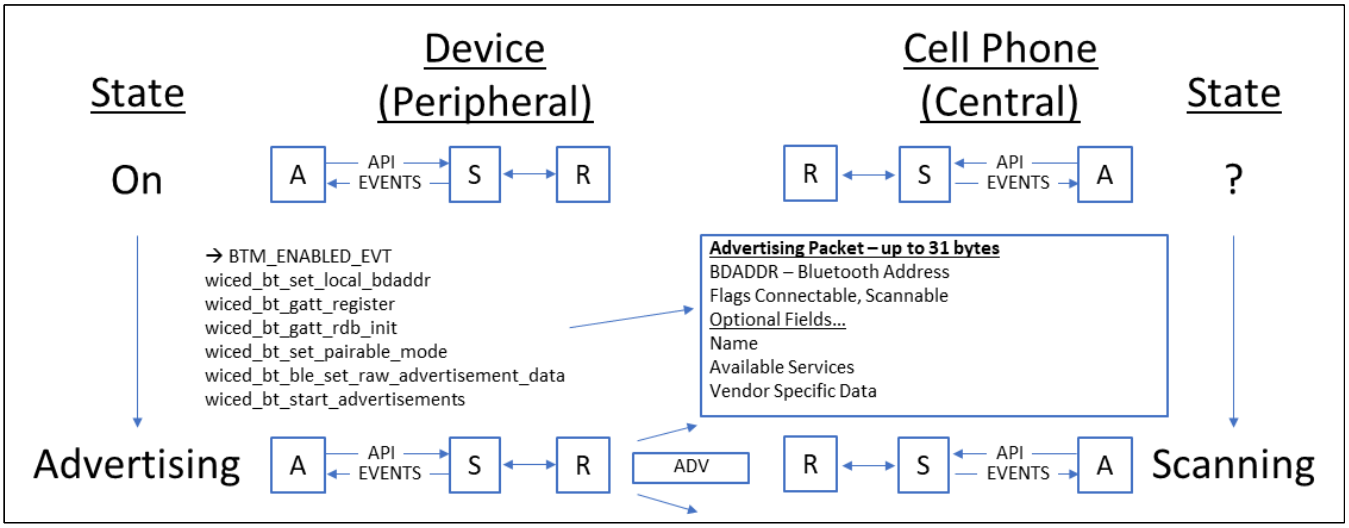
For a Central to know of your existence you need to send out Advertising packets. The Advertising Packet will contain your Bluetooth® Device Address (BDA), some flags that include information about your connection availability status, and one or more optional fields for other information, like your device 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.



3.2.2.1 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 solicitated services - 16 bit UUIDs */
    BTM_BLE_ADVERT_TYPE_128SOLICITATION_SRV_UUID    = 0x15,             /**< List of solicitated 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 solicitated 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.

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.

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);
}
```

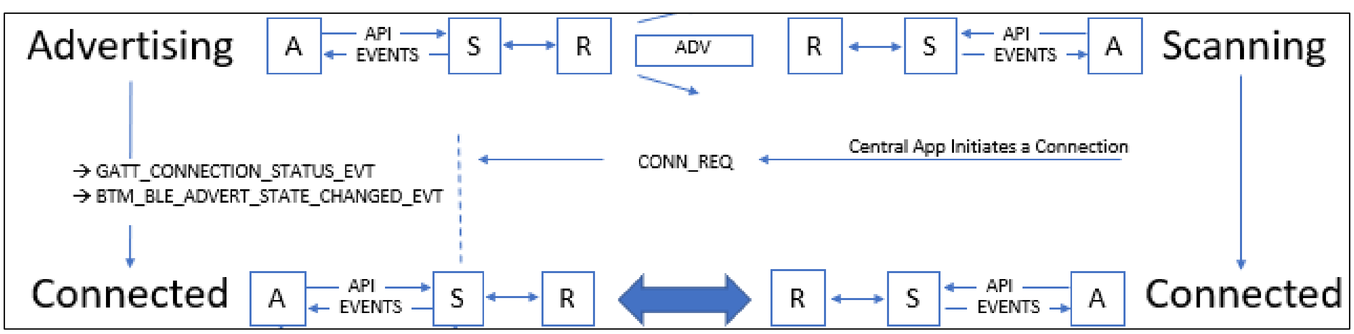
There is a scan response packet that can hold an additional 31 bytes which will be discussed later. The scan response packet array can also be set up by the ModusToolbox™ Bluetooth® Configurator.

3.2.3 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.



3.2.4 Exchange data

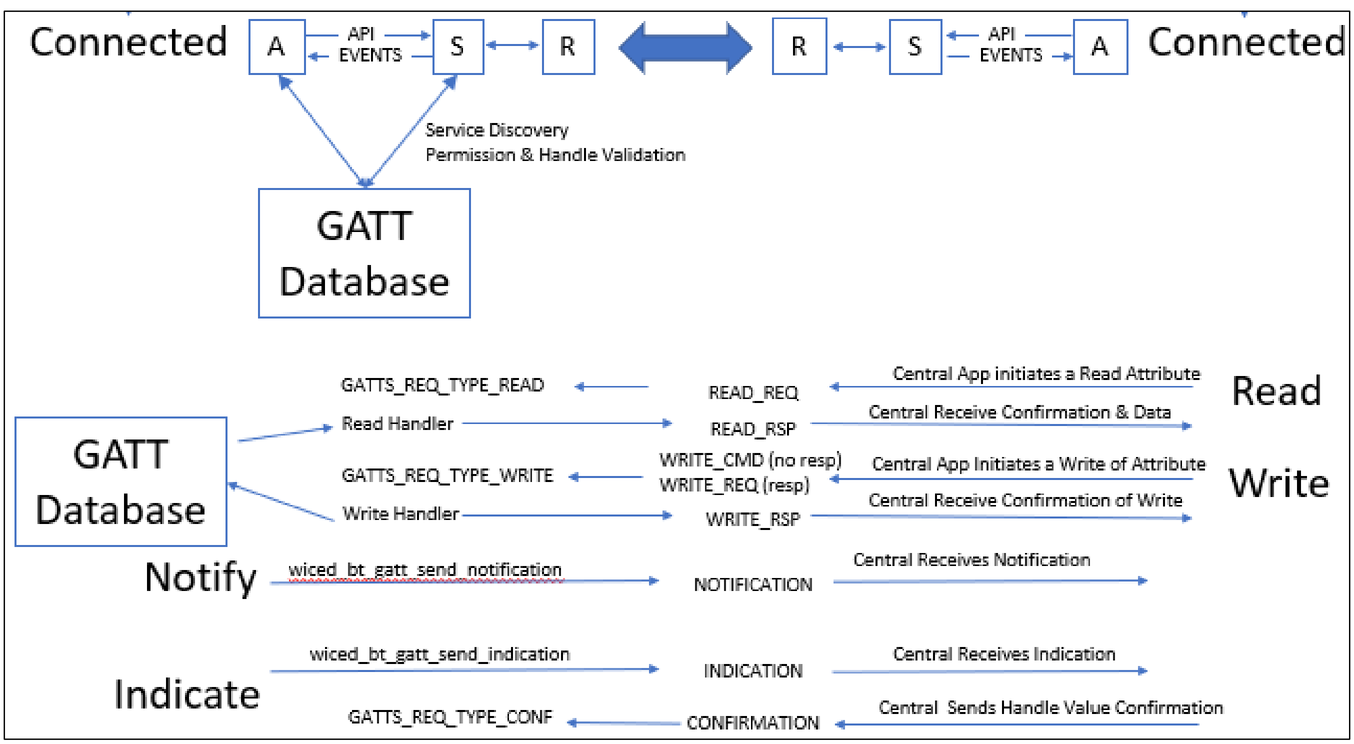
Now that you are connected you need to be able to exchange data. In the world of Bluetooth® LE this happens via the Attribute Protocol (ATT). The basic ATT protocol has 4 types of transactions: Read & Write which are initiated by the Client and Notify & Indicate which are initiated by the Server.

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

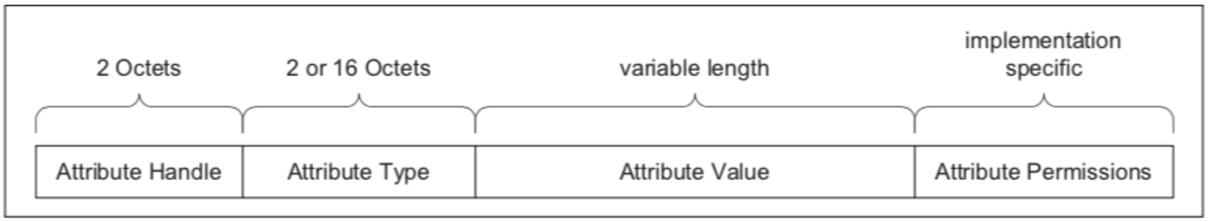
I'll talk in more detail about the GATT database implementation later. With all of that, here is the final section of the big picture:



3.3 Attributes, the Generic Attribute Profile & GATT Database

3.3.1 Attributes

A GATT Database is a 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.

3.3.2 Profiles, Services, Characteristics

The GATT Database is "flat" – it's just a bunch 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.

3.3.3 Service Declaration

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)

3.3.4 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»	Charac-teristic Properties	Character-istic Value Attribute Handle	Character-istic 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.

3.4 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 "MySvc" 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.

3.4.1 ModusToolbox™ Bluetooth® configurator

You will use this in Exercise 1:

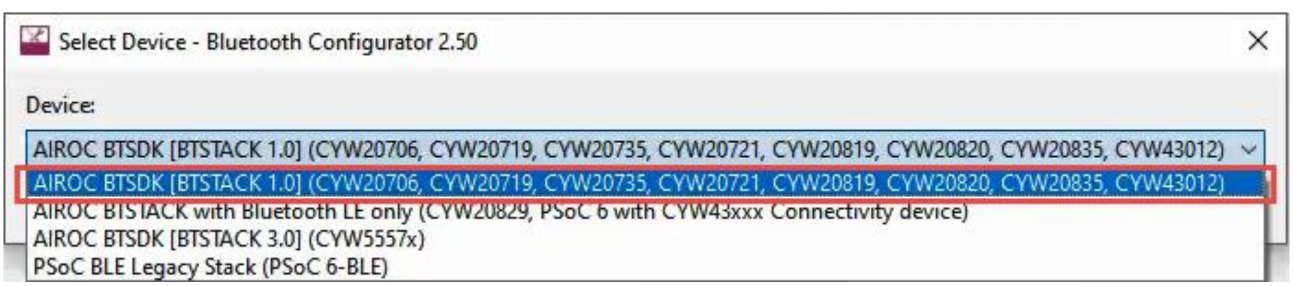
The Bluetooth® Configurator is a ModusToolbox™ tool that will set up a customized GATT database for your application. The Bluetooth Configurator generates two files that you will be using: *cycfg_gatt_db.c* and *cycfg_gatt_db.h*. It also generates a timestamp file called *cycfg_bt.timestamp*.

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)

If you use the Eclipse IDE for ModusToolbox™, select the application and then click the link for the Bluetooth® Configurator in the Quick Panel. The configuration associated with the application will be opened if it exists. If a configuration does not exist for the application, it will be created using the correct version for the device used in your application.

For the CLI, the configuration associated with the application will be opened if it exists. If a configuration does not exist, you must use the tool to create one using **File > New** and then select the correct version from the drop-down list. The devices associated with each version are listed so you don't need to remember which one goes with each device. For the CYW20835 device used in this class, it is BTSTACK 1.0.



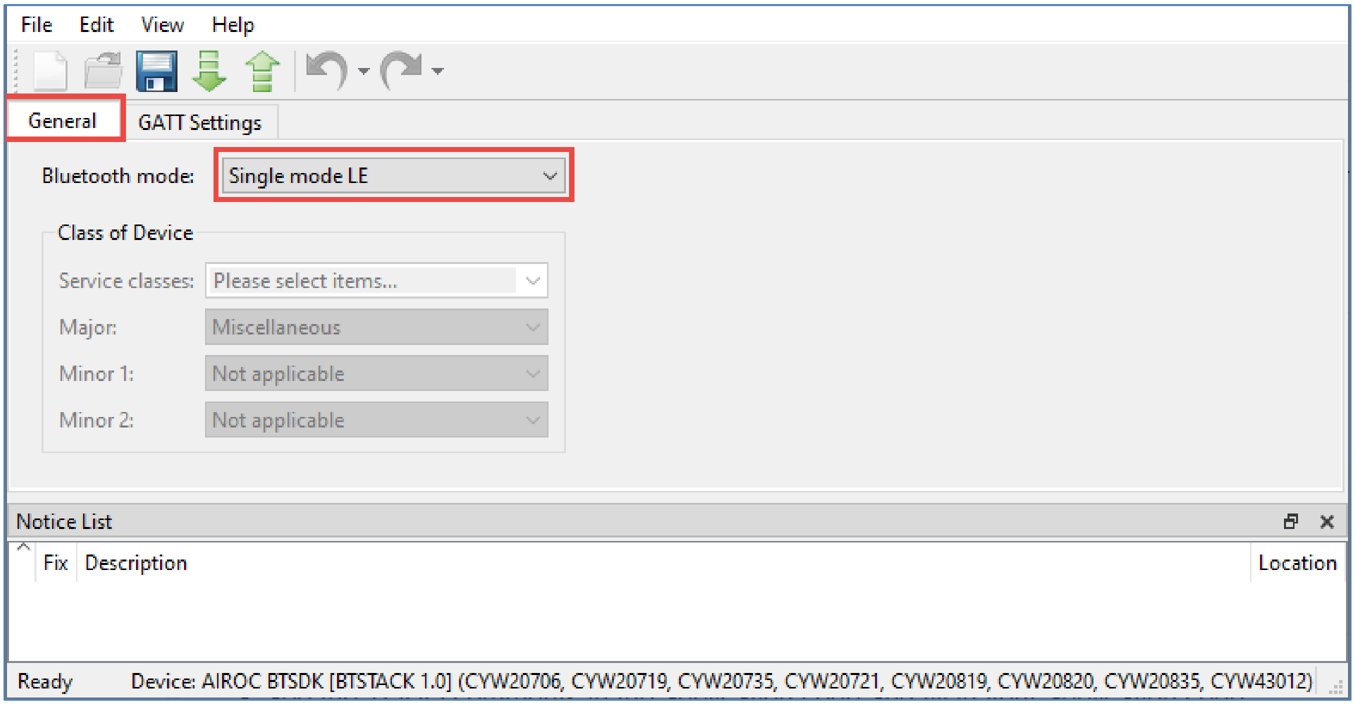
For the last two options to launch the tool, you must either create a new configuration (**File > New**) or open an existing one (**File > Open**).

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

3.4.1.1 General tab

As you can see below, the **General** tab allows you to select the required Bluetooth® mode. In our case, we just want a **Single mode LE device**, meaning that this application will only support Bluetooth® Low Energy. Other options allow you to select **Single Mode BR/EDR** (i.e. Bluetooth® Classic) or **Dual Mode** (i.e. both Classic and LE).

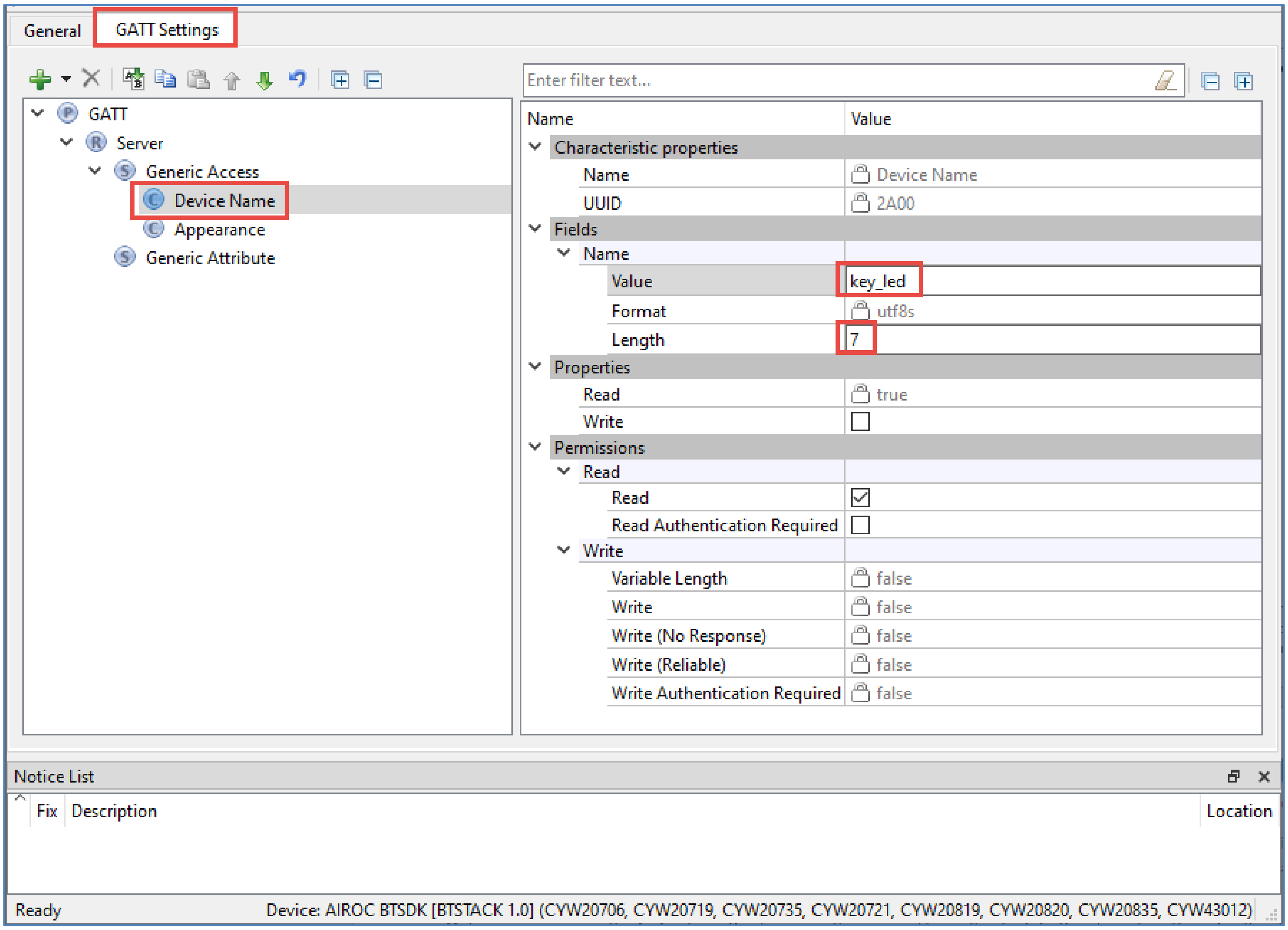
The **Class of Device** settings only apply to BR/EDR and Dual mode operation.



3.4.1.2 GATT Settings tab

In the **GATT Settings** tab, you set up the GATT database that your device needs. Note that the configurator already has the required Generic Access and Generic Attribute Services defined.

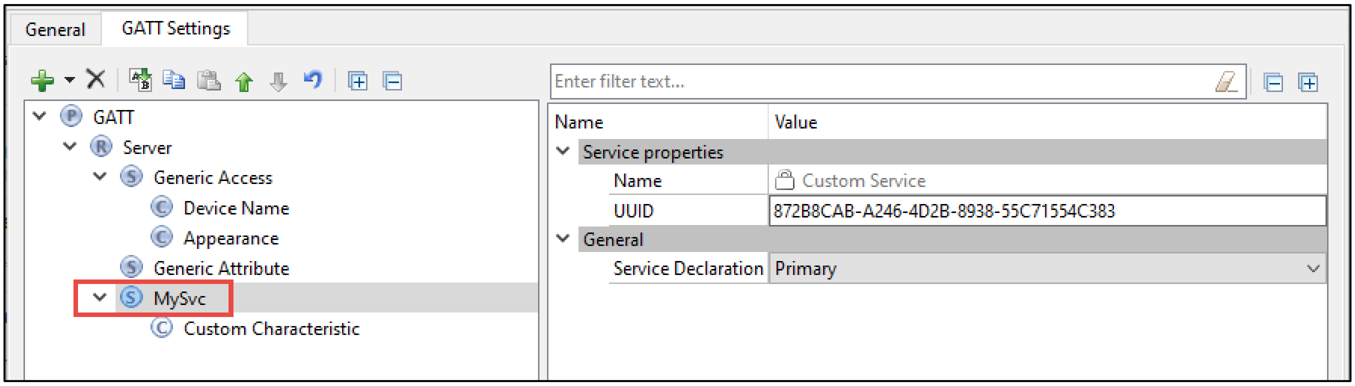
The first step is to give the device a name in the Generic Access Service. We will call this device "key_led". You must also enter the correct length for the name – in this case, it is 7.



Note: *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.*

The next step is to set up a new Service. 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 now appears in the GATT database tree.
2. Right-click on the Custom Service and select **Rename**. Call the service "MySvc".
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.



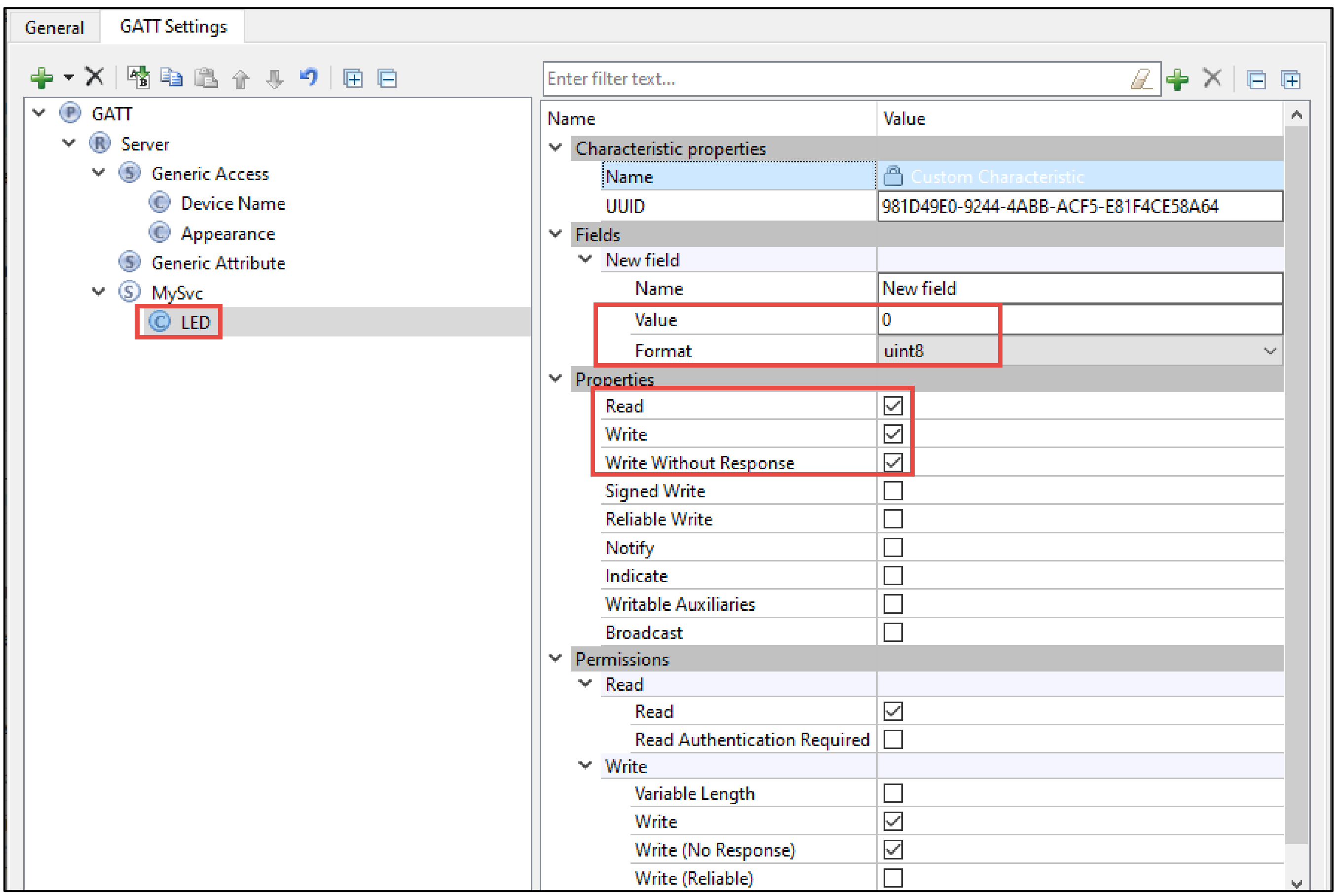
The Service includes a Characteristic, which we are going to use to control the LED. To do this you:

1. Right-click on **Custom Characteristic** under MySvc and **Rename** it 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.



3.4.1.3 Generated code

Once you have set up 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 2 files for the GATT database and a timestamp file. They are:

File Name	Purpose
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.

3.4.2 Editing the firmware

You will use this in Exercise 1:

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. The template includes a little bit of set-up code for the `BTM_ENABLED_EVT` and some helper functions:

- `app_bt_management_callback` is the callback function for Bluetooth® Stack events. The `BTM_ENABLED_EVT` prints the Bluetooth® Device Address (BDA) and starts advertising for a connection. You will add additional functionality to this function during the exercises.
- `app_gatt_callback` is the callback function for GATT events such as connect/disconnect and attribute read/write requests.
- `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.
- `app_set_advertisement_data` creates the advertising packet that includes the device name.

There is also a set of utility functions provided in *app_bt_utils.c* and *app_bt_utils.h*. These functions convert return codes to human readable strings for various Bluetooth® events and return values.

To modify the template firmware for our example application, the changes required are:

1. Start by opening *app.c* and verify that the code from the Bluetooth® configurator is included:

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

2. The `application_start` function initializes the Stack by registering its callback and providing the Stack settings and buffer pools. We will talk about these in more detail later but for now just review the function call.

3. The `BTM_ENABLED_EVT` case in `app_bt_management_callback` reads and reports the 6-byte Bluetooth® Device Address (BDA) in the terminal when the Stack gets enabled.

Note: The `WICED_BT_TRACE` function has a special format of %B which is used to print a Bluetooth® Device Address.

4. The Bluetooth® Device Address must be unique to avoid collisions with other devices.

   By default, the address format is defined in a file in the SDK. It can be found at:

   mtb_shared/wiced_btsdk/dev-kit/baselib/<device>/<version>/COMPONENT_<device>/platforms/<device>_.btp*

   In this file, there are lots of device specific settings. The one that controls the address is:

   ```
   DLConfigBD_ADDRBase = "20835B1*****"
   ```

   The asterisk characters mean that a value will be chosen for those digits during build. Therefore, the 6-byte address generated for your device will start with 20835B1 with 5 digits after that. By default, the 5 digits are based on the MAC address of your computer. That is, the address generated should always be the same for your computer but will be different for other computers.

   There are 2 cases where this may cause a problem: (1) if you are programming more than one kit from a single computer and want them to operate at the same time - such as a peripheral and a central; or (2) when using a virtual machine, a MAC address may not be found in which case the 5 digits will all be set to 0.

   Due to the above potential issues, we will change a setting to get random values for those 5 digits so that there aren't any collisions between students. This means that you will get a different Bluetooth address each time you rebuild an application.

   To set that up, open the *makefile* that is in your application and find the line that says:

   ```
   BT_DEVICE_ADDRESS?=default
   ```

   and change it to:

   ```
   BT_DEVICE_ADDRESS?=random
   ```

   Note that in this case "random" only means use random values for the 5 digits with an asterisk in configuration file. The resulting address is still public device address for your device. Don't confuse this with a truly random device address. We will discuss Bluetooth LE address types in more detail in the privacy section in the next chapter.

5. Back in *app.c*, 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 and initialize the GATT database*/
   wiced_bt_gatt_register( app_gatt_callback );
   wiced_bt_gatt_db_init( gatt_database, gatt_database_len );
   ```

6. Next, we 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 );
```

7. The `BTM_ENABLED_EVT` finishes by calling the helper function to set up the advertisement packet and then starts advertising. These two function calls are already provided in the template.

8. Next, we need to specify what the firmware does during 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 provided in the template – you just need to add a new case). This event will occur whenever the Central reads the LED characteristic. The code uses the GATT database value, not the state of the pin itself, and so non-zero implies "on" and zero means "off".

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

Note: *Finding the value of the Characteristic and passing it to the Stack so that it can be transmitted to the Client is handled by the code that is already provided – the code you added just prints a message to the UART so that you will know when the Client reads the state of the LED.*

Note: *The case name (HDLC_MYSVC_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_mysvc_led) is defined in cycfg_gatt_db.c. You will see these in more detail later.*

9. In `app_gatt_set_value`, the template function automatically takes the value that the Client sent to the Stack and updates the `app_mysvc_led` array in the GATT database with a call to `memcpy`.

```
// 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;
```

   However, we do need to add a case to physically turn the LED ON or OFF based on the value that was written by the Client. We will also print a message to the UART whenever the Characteristic value is written. Again, the switch statement is in the template – just add the new case. We are going to use LED2 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 Ex 01: Add code for any action required when this attribute is written
// For example, you may need to write the value into NVRAM if it needs to be
persistent
switch ( attr_handle )
{
 case HDLC_MYSVC_LED_VALUE:
      wiced_hal_gpio_set_pin_output(LED2, !(app_mysvc_led[0]) );
      WICED_BT_TRACE( "Turn the LED %s\r\n", app_mysvc_led[0] ? "ON" : "OFF" );
      break;

}
```

3.4.3 Testing the application

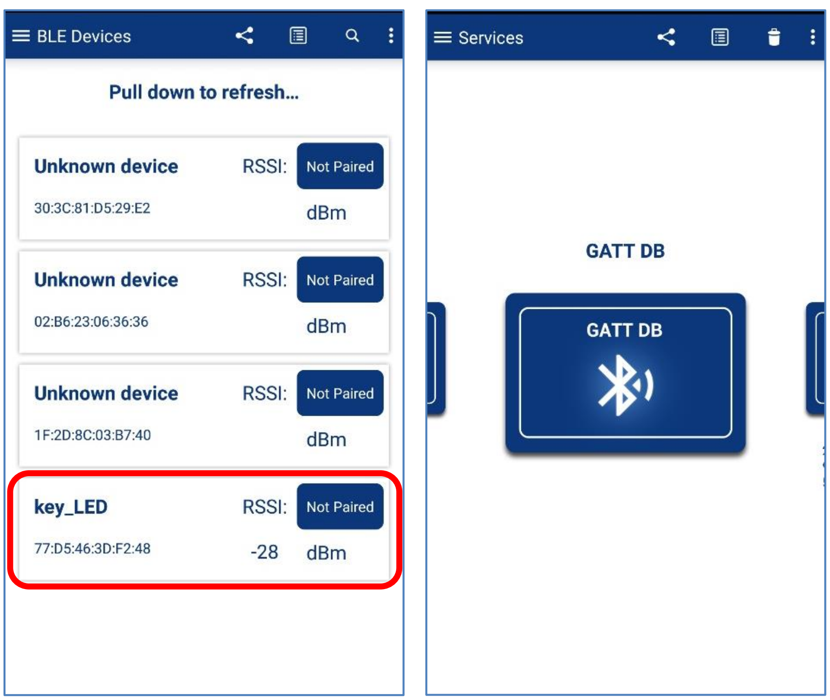
You will use this in Exercise 1:

Start up a UART serial terminal emulator connected to the PUART with a baud rate of 115200, then build and program your kit. When the application firmware starts up you see some messages indicating that the application is running, several Bluetooth® Stack events have occurred and the device is advertising in the undirected high duty cycle mode. The Bluetooth® device address assigned to your device is also printed.

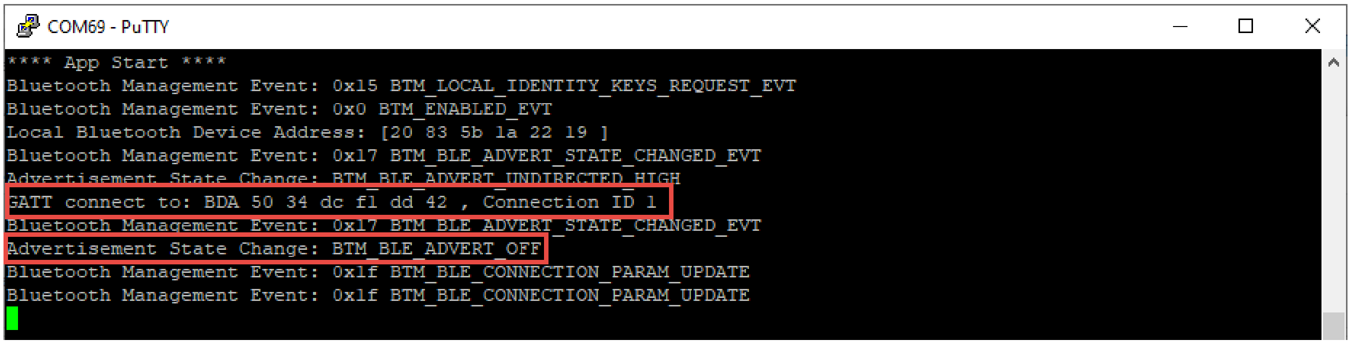
```
COM69 - PuTTY
**** App Start ****
Bluetooth Management Event: 0x15 BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT
Bluetooth Management Event: 0x0 BTM_ENABLED_EVT
Local Bluetooth Device Address: [20 83 5b 1a 22 19 ]
Bluetooth Management Event: 0x17 BTM_BLE_ADVERT_STATE_CHANGED_EVT
Advertisement State Change: BTM_BLE_ADVERT_UNDIRECTED_HIGH
```

Note: *If you need a refresher on using a serial terminal emulator, see ModusToolbox™ Level 1 Getting Started class, Tools chapter, Serial Terminal Emulator section.*

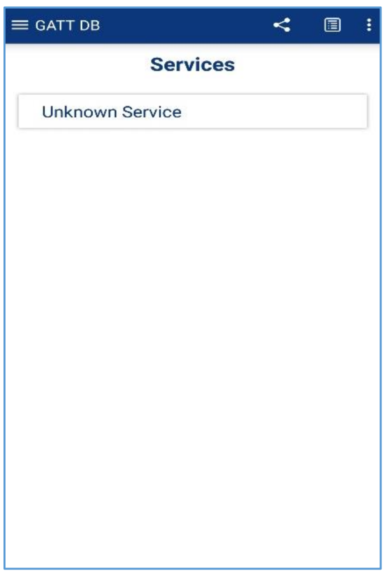
Run AIROC™ Connect on your phone. When you see the "<inits>_LED" device, tap on it. AIROC™ Connect will connect to the device and will show the GATT browser widget.

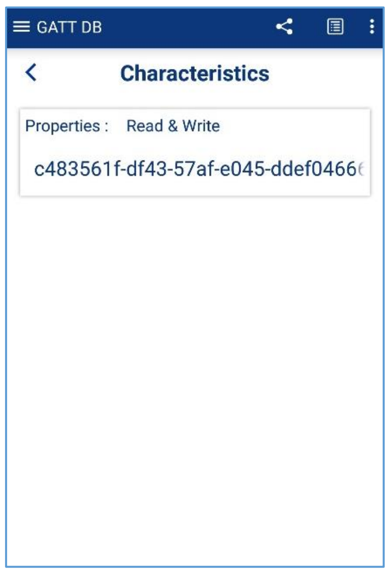


On the terminal window, you will see that there has been a connection and the advertising has stopped.



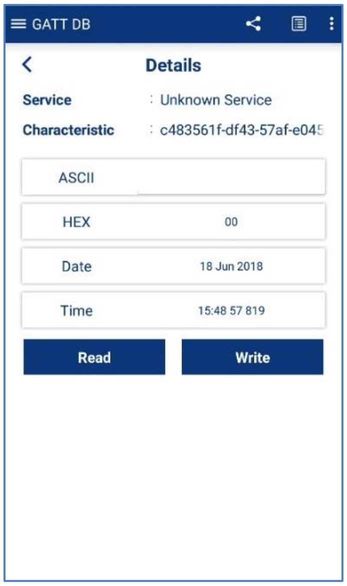
Back in AIROC™ Connect, tap on the GATT DB widget to open the browser. You will see an Unknown Service (which I know is "MySvc"). Tap on the Service and the app will tell you that there is a Characteristic (which I know is LED).



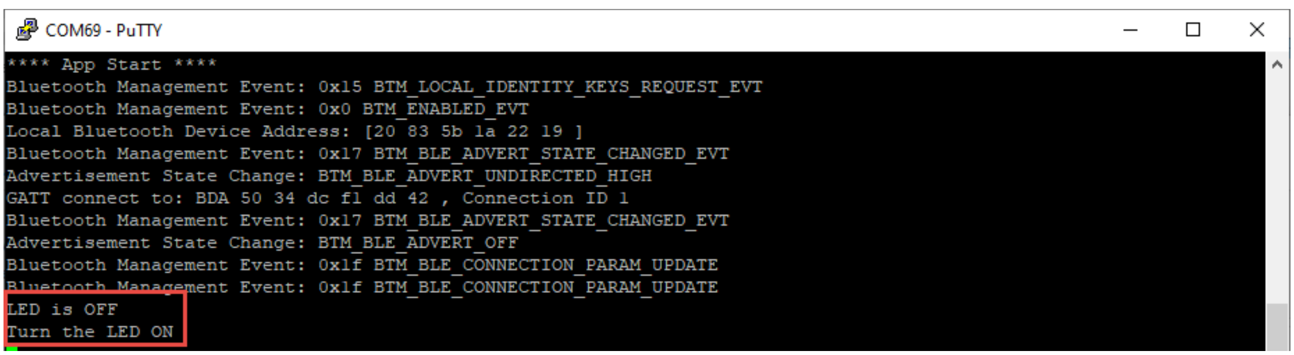


Note: *In the iOS version of AIROC™ Connect, the Characteristic UUID will not be shown – it will just say "Unknown Characteristic".*

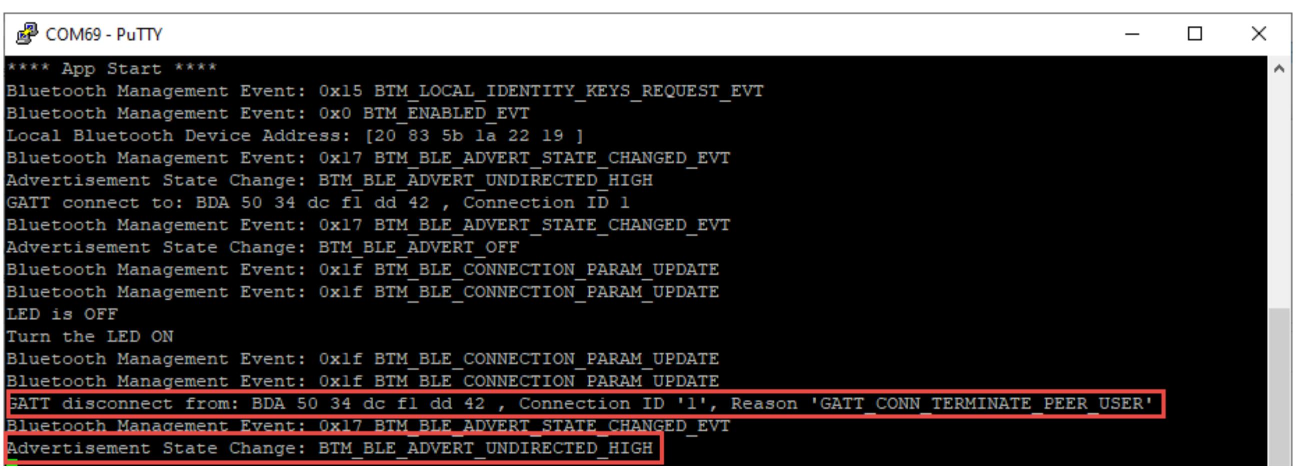
Tap on the Service to see details about it. First, tap the Read button and you will see that the current value is 0. Now you can Write 1's or 0's into the Characteristic and you will find that the LED turns on and off accordingly.



In the UART window, you will see messages when you read and write the LED Characteristic.



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



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

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.

Management 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.

These events are handled in the Stack callback function `app_bt_management_callback`.

3.5.2 Essential GATT Events

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

GATT 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`.

These events are handled in the GATT event callback function `app_gatt_callback`.

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. The only two that concern us at the moment are:

Attribute Request Event	Description
`GATTS_REQ_TYPE_READ`	When the Central reads the value of a characteristic. The firmware must find the value in the GATT database and provide it to the Stack.
`GATTS_REQ_TYPE_WRITE`	When the Central writes the value of a characteristic. The firmware must take the value from the Stack and put it in GATT database.

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

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

3.6.1 Turn on the Stack

When a WICED device turns on, the chip boots, starts the RTOS and then jumps to a function called `application_start` which is where your application firmware starts. At that point in the proceedings, your application firmware is responsible for turning on the Stack. 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® Stack 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 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.

In our example, the `app_bt_management_callback` case for `BTM_ENABLED_EVT` event does the following:

1. Prints out the device's Bluetooth® address.
2. Calls the functions `wiced_bt_gatt_register` and `wiced_bt_gatt_db_init` to register a callback function for GATT database events and initialize the GATT database.
3. Calls `wiced_bt_set_pairable_mode` to set the pairable mode to false. That means the device will allow connections but not pairing.
4. Calls the functions `app_set_advertisement_data` and `wiced_bt_start_advertising` to set the advertisement packet and finally start advertising.

3.6.2 Start Advertising

The Stack is triggered to start advertising by the last step of the `BTM_ENABLED_EVT` event 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 2nd and 3rd 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 ON or OFF 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 the GATT event `GATT_CONNECTION_STATUS_EVT` which is processed by the `app_bt_gatt_event_callback` function. That event uses the `connected` parameter to determine if it is a connection or a disconnection event and prints a useful message.

On a connection, the Stack automatically 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 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 */
```

When I looked at this table for the first time I thought to myself that Victor must have a sense of humor after all, given error code `WICED_BT_GATT_ERR_UNLIKELY`.

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`

3.6.5 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.

Note: *There is another event code called `GATTS_REQ_TYPE_PREP_WRITE` which is used when writing large amounts of data with a non-zero offset. This event code will not be implemented for our simple example since our characteristic is only a single byte.*

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_SUCESS, 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

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* which are found in the application's *GeneratedSource* directory.

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 four parts. The first three are in *cycfg_gatt_db.c* while the last is implemented in the application code (in *app.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.

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                          0x0001
/* Service Generic Attribute */
#define HDLS_GATT                         0x0006

/* Service MYSVC */
#define HDLS_MYSVC                        0x0007
```

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_MYSVC      0x83u, 0xC3u, 0x54u, 0x15u, 0xC7u, 0x55u, 0x38u, 0x89u,
0x2Bu, 0x4Du, 0x46u, 0xA2u, 0xABu, 0x8Cu, 0x2Bu, 0x87u
```

In addition, there are a bunch of predefined standard 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_MYSVC_LED                    0x0008
#define HDLC_MYSVC_LED_VALUE              0x0009
```

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 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 GATTDB_PERM_WRITABLE                 (GATTDB_PERM_WRITE_CMD |
                                             GATTDB_PERM_WRITE_REQ|
                                             GATTDB_PERM_AUTH_WRITABLE)
#define GATTDB_PERM_MASK                     (0x7f)
#define GATTDB_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_MYSVC_LED_VALUE,       1,      1,      app_mysvc_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_mysvc_led[]  =      {0x00u, };
```

From the above you can see that when you want to access the value of the LED Characteristic for the "MySvc" Service, you will find the data in `app_mysvc_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.

3.7.4 The application programming interface

There are two functions in the template which make up the interface to the GATT Database, `app_gatt_get_value` and `app_gatt_set_value`. Here are the function prototypes:

```
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 Attribute handle. This place is marked with `//TODO:` in the two functions.

These functions return a `wiced_bt_gatt_status_t` result which tells the Stack what to do next. Assuming things works these functions return `WICED_BT_GATT_SUCCESS`. In the case of a write, that tells the Stack to send a WRITE Response to the Client indicating success.

3.8 Stack Configuration

When you initialize the Bluetooth LE Stack one of the arguments you pass is a pointer to a structure of type `wiced_bt_cfg_settings_t`. This structure contains initialization information for both the Bluetooth LE and Classic Bluetooth configuration. This structure is provided by the BSP or the starter template application and typically resides in the file *app_bt_cfg.c*.

The structure definition is shown below. Many of the entries are themselves structures with multiple entries of their own so there is quite a bit of configuration information in this structure.

```
/** Bluetooth stack configuration */
typedef struct
{
    uint8_t                          *device_name;               /**< Local device name (NULL terminated) */
    wiced_bt_dev_class_t             device_class;               /**< Local device class */
    uint8_t                          security_requirement_mask;  /**< Security requirements mask (BTM_SEC_NONE, or combination
    uint8_t                          max_simultaneous_links;     /**< Maximum number simultaneous links to different devices *

    /* Scan and advertisement configuration */
    wiced_bt_cfg_br_edr_scan_settings_t br_edr_scan_cfg;         /**< BR/EDR scan settings */
    wiced_bt_cfg_ble_scan_settings_t    ble_scan_cfg;            /**< BLE scan settings */
    wiced_bt_cfg_ble_advert_settings_t  ble_advert_cfg;          /**< BLE advertisement settings */

    /* GATT configuration */
    wiced_bt_cfg_gatt_settings_t     gatt_cfg;                   /**< GATT settings */

    /* RFCOMM configuration */
    wiced_bt_cfg_rfcomm_t            rfcomm_cfg;                 /**< RFCOMM settings */

    /* Application managed l2cap protocol configuration */
    wiced_bt_cfg_l2cap_application_t l2cap_application;          /**< Application managed l2cap protocol configuration */

    /* Audio/Video Distribution configuration */
    wiced_bt_cfg_avdt_t              avdt_cfg;                   /**< Audio/Video Distribution configuration */

    /* Audio/Video Remote Control configuration */
    wiced_bt_cfg_avrc_t              avrc_cfg;                   /**< Audio/Video Remote Control configuration */

    /* LE Address Resolution DB size  */
    uint8_t                          addr_resolution_db_size;    /**< LE Address Resolution DB settings - effective only for p

    /* Maximum number of buffer pools */
    uint8_t                          max_number_of_buffer_pools; /**< Maximum number of buffer pools in p_btm_cfg_buf_pools an

    /* Interval of  random address refreshing */
    uint16_t                         rpa_refresh_timeout;        /**< Interval of  random address refreshing - secs */
} wiced_bt_cfg_settings_t;
```

You may at some point need to modify all of the Stack settings for different applications, but two of the sub-structures that you will frequently change are `ble_advert_cfg` (advertising interval and timeout values) and `gatt_cfg` (GATT settings such as appearance, number of simultaneous client and server connections allowed, maximum attribute length and maximum MTU sizes). You may also change the value of `rpa_refresh_timeout` when dealing with privacy.

Here are examples of what these structures look like in the *app_bt_cfg.c* file from a typical application.

```
.ble_advert_cfg =                                            /* BLE advertisement settings */
{
    .channel_map                      = BTM_BLE_ADVERT_CHNL_37 |
                                        BTM_BLE_ADVERT_CHNL_38 |
                                        BTM_BLE_ADVERT_CHNL_39,

    .high_duty_min_interval           = WICED_BT_CFG_DEFAULT_HIGH_DUTY_ADV_MIN_INTERVAL,
    .high_duty_max_interval           = WICED_BT_CFG_DEFAULT_HIGH_DUTY_ADV_MAX_INTERVAL,
    .high_duty_duration               = 30,

    .low_duty_min_interval            = 1024,
    .low_duty_max_interval            = 1024,
    .low_duty_duration                = 60,

    .high_duty_directed_min_interval = WICED_BT_CFG_DEFAULT_HIGH_DUTY_DIRECTED_ADV_MIN_INTERVAL,
    .high_duty_directed_max_interval = WICED_BT_CFG_DEFAULT_HIGH_DUTY_DIRECTED_ADV_MAX_INTERVAL,

    .low_duty_directed_min_interval  = WICED_BT_CFG_DEFAULT_LOW_DUTY_DIRECTED_ADV_MIN_INTERVAL,
    .low_duty_directed_max_interval  = WICED_BT_CFG_DEFAULT_LOW_DUTY_DIRECTED_ADV_MAX_INTERVAL,
    .low_duty_directed_duration      = 30,

    .high_duty_nonconn_min_interval  = WICED_BT_CFG_DEFAULT_HIGH_DUTY_NONCONN_ADV_MIN_INTERVAL,
    .high_duty_nonconn_max_interval  = WICED_BT_CFG_DEFAULT_HIGH_DUTY_NONCONN_ADV_MAX_INTERVAL,
    .high_duty_nonconn_duration      = 30,

    .low_duty_nonconn_min_interval   = WICED_BT_CFG_DEFAULT_LOW_DUTY_NONCONN_ADV_MIN_INTERVAL,
    .low_duty_nonconn_max_interval   = WICED_BT_CFG_DEFAULT_LOW_DUTY_NONCONN_ADV_MAX_INTERVAL,
    .low_duty_nonconn_duration       = 0
},
```

```
.gatt_cfg =                                                  /* GATT configuration */
{
    .appearance                       = APPEARANCE_GENERIC_TAG,
    .client_max_links                 = 0,
    .server_max_links                 = 1,
    .max_attr_len                     = 512,
    #if !defined(CYW20706A2)
    .max_mtu_size                     = 517
    #endif
},
```

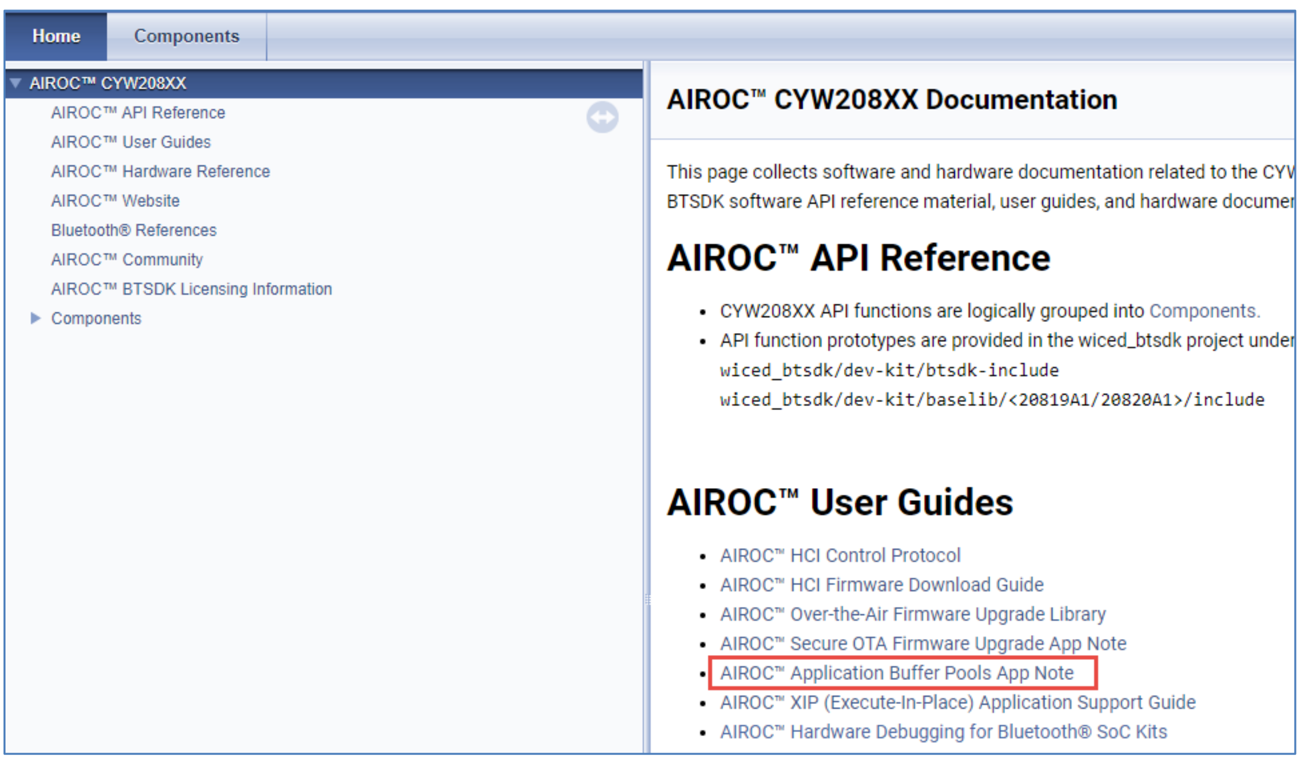
```
/* Interval of  random address refreshing */
.rpa_refresh_timeout                  = WICED_BT_CFG_DEFAULT_RANDOM_ADDRESS_CHANGE_TIMEOUT,
```

3.9 Buffer Pools

Rather than use the C typical memory allocation scheme, malloc, the btstack_v1 solution has a scheme optimized for Bluetooth®. One of the arguments that you need to pass to the Stack initialization function is a pointer to the pools. This array is also defined in *app_bt_cfg.c* and there are four different size buffer pools. The default settings are:

```
/* TODO set the application buffer pool as per your use case */
/*****************************************************************************
 * wiced_bt_stack buffer pool configuration
 *
 * Configure buffer pools used by the stack
 *
 * Pools must be ordered in increasing buf_size.
 * If a pool runs out of buffers, the next  pool will be used
 *****************************************************************************/
const wiced_bt_cfg_buf_pool_t wiced_bt_cfg_buf_pools[WICED_BT_CFG_NUM_BUF_POOLS] =
{
/*  { buf_size, buf_count } */
    { 64,       12  },      /* Small Buffer Pool */
    { 360,      6   },      /* Medium Buffer Pool (used for HCI & RFCOMM control messages, min recommended
    { 1056,     6   },      /* Large Buffer Pool  (used for HCI ACL messages) */
    { 1056,     0   },      /* Extra Large Buffer Pool - Used for avdt media packets and miscellaneous (if
};
```

There is a user guide on the BTSDK documentation page that contains additional information on the use of buffer pools. For example, the large buffer pool should be set to at least as large as the MTU value plus 12.



You can read the amount of free memory in the device at initialization and after starting the stack by using the function `wiced_memory_get_free_bytes`.

3.10 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 by creating a helper function similar to the `app_set_advertisement_data` function used to set up the advertising packet. Remember that the scan response packet doesn't need the flags field and the maximum length is 31 bytes.

Once the scan response packet is set up, you just call `wiced_bt_ble_set_raw_scan_response_data` to pass that information to the Stack instead of the `wiced_bt_ble_set_raw_advertisement_data` that was used for advertising packets.

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.

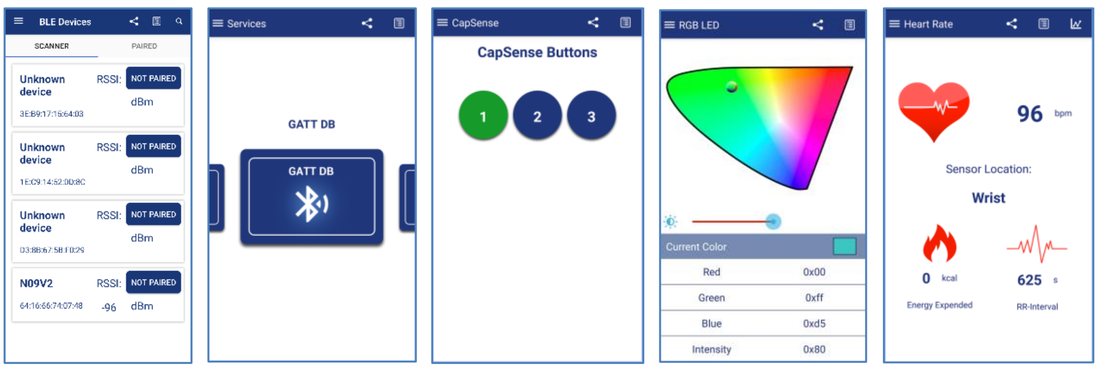
3.11 AIROC™ Bluetooth® Connect

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

3.12 Differences for btstack_v3 devices

As mentioned at the beginning of this chapter, some devices contain btstack_v3 in their ROM. While most things are the same, there are a few notable differences when using btstack_v3 instead of btstack_v1. This section briefly describes the differences.

3.12.1.1 Bluetooth® Configurator for btstack_v3 devices

The Bluetooth® Configurator for devices whose ROM contains btstack_v3 has additional tabs for **General LE**, **GAP Settings**, and **L2CAP Settings**. These tabs allow you to use the configurator to generate additional code rather than modifying it and including it in the application manually. This includes things like number of allowed client and remote connections, advertisement settings, advertisement, and scan response packets, etc.

The additional configurator tabs look the same as those in the Type1 Bluetooth® class, so you can review that material for additional details.

Once the configuration is saved for a btstack_v3 device, the following files will be generated. Each of the three header files must be included in the application to gain access to the required definitions and structures.

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. These replace the app_bt_cfg.c and app_bt_cfg.h files that are modified and included manually in btstack_v1 applications.
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.

In a btstack_v3 application, the files *app_bt_cfg.c* and *app_bt_cfg.h* will not be present. This is because the equivalent information is contained in the generated files *cycfg_bt_settings.c*, *cycfg_bt_settings.h*, *cycfg_gap.c*, and *cycfg_gap.h*.

3.12.1.2 Firmware for btstack_v3 devices

Stack initialization and memory management

The `wiced_bt_stack_init` function only takes 2 arguments instead of three. The second argument will point to the Stack configuration structure in the generated source file *cycfg_bt_settings.c* instead of the structure manually included in the application file *app_bt_cfg.c*. For example:

```
wiced_bt_stack_init (app_bt_management_callback, &wiced_bt_cfg_settings);
```

The third argument is missing for btstack_v3 because memory management is done differently in btstack_v3. First, before calling `wiced_bt_stack_init`, you must call `wiced_bt_create_heap` to initialize a dynamic memory area. For example:

```
wiced_bt_heap_t *p_default_heap = NULL;
```

```
p_default_heap = wiced_bt_create_heap("default_heap", NULL, (1024 * 6), NULL,
WICED_TRUE);
```

Second, the GATT event callback has two extra events – one to request memory and one to free up memory required for GATT events:

Event	Description
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.

The code required to allocate and free memory for the events listed above is the same as the Type1 Bluetooth® class, so you can review that material for additional details.

GATT read and write events

In btstack_v3 applications the `GATT_ATTRIBUTE_REQUEST_EVT` request sub-events are handled differently. There are more sub-events that split out different types of Attribute request events. For details on the different events and the read and write handlers, see the Type1 Bluetooth® class.

Advertising and Scan Response Packets

For btstack_v3 devices is that there will not normally be a helper function to set up advertising packets or scan response packets. That's because those packets are set up by the Bluetooth® Configurator and are placed in the generated files *cycfg_gap.c*, and *cycfg_gap.h*. So, you only have to call the appropriate function to set the raw advertisement or scan response data and point to the generated structures. A define is provided in the header file for the packet data size as well. For example:

```
wiced_bt_ble_set_raw_advertisement_data(CY_BT_ADV_PACKET_DATA_SIZE,
                                        cy_bt_adv_packet_data);
```

3.13 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 "MySvc" 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. Create a new ModusToolbox™ application for the CYW920835M2EVB-01 BSP.

On the application template page, use the **Browse** button to use the template application from the class files under *Templates/**ch03_ex01_ble***.

☐ 2. Follow the instructions in section 3.4.1 to set up the Bluetooth® configuration.

☐ 3. Open *app.c* and follow the instructions in section 3.4.2 to complete the code.

Note: *Look for the string "TODO" in main.c to find the locations that need changes. There are comments for both exercise 1 and exercise 2.*

☐ 4. Build the application and program your kit.

☐ 5. Follow the instructions in section 3.4.3 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: *We will used LED1 since LED2 is already used by the application.*

Application Creation

☐ 1. Create a new ModusToolbox™ application for the CYW920835M2EVB-01 BSP.

On the application template page, use the **Browse** button to start from the completed application for exercise 1. If you did not complete exercise 1, the solution can be found in *Projects/key_ch03_ex01_ble*. Name the new application ***ch03_ex02_status***.

☐ 2. Launch the Bluetooth® Configurator and set the device name to **<inits>_status**.

☐ 3. Save changes and close the configurator.

☐ 4. Open *app.c*.

☐ 5. Add the following include lines:
#include "wiced_hal_pwm.h"
#include "wiced_hal_aclk.h"

☐ 6. In the `BTM_ENABLED_EVT` case, configure the PWM to connect to LED1 but then disable it:

```
/* Configure the PWM and then disable it. */
wiced_hal_pwm_start( PWM0, PMU_CLK, LED_TOGGLE, PWM_INIT, 0 );
wiced_hal_pwm_disable(PWM0);
```

Note: *Routing of the PWM to the LED1 pin and enabling the ACLK will be done once advertising starts so we don't want to do it here.*

Note: *You may see some items underlined in red before you build – these are things that are in the new includes that you added.*

☐ 7. Declare a global `uint16_t` variable called `connection_id` to keep track of the connection ID. Initialize it to 0.

☐ 8. Set/clear the connection ID variable in the `app_gatt_callback` when a connection is made/lost.

For a connection: `connection_id = p_conn->conn_id;`
For a disconnection: `connection_id = 0;`

☐ 9. Update the LED's state whenever the advertisement state changes.

In the `BTM_BLE_ADVERT_STATE_CHANGED_EVT`, do the following:

1. If `p_event_data->ble_advert_state_changed` is `BTM_BLE_ADVERT_OFF` , advertising is not running.
   i. If f the connection ID is 0, the device is not connected so the LED should be off. Disable the PWM and ACLK, set the GPIO function for LED1 to WICED_GPIO, and configure the pin as an output with the output low (remember the LED is active low).
   ii. If f the connection ID is not 0, the device is connected so the LED should be on. Disable the PWM and ACLK, set the GPIO function for LED1 to WICED_GPIO, and configure the pin as an output with the output low (remember the LED is active low).
2. If `p_event_data->ble_advert_state_changed` is not `BTM_BLE_ADVERT_OFF`, some sort of advertising is running so the LED should be blinking. Set the GPIO function to WICED_PWM0, enable ACLK, and enable the PWM.

Note: *The function to set the pin as a GPIO or a PWM is `wiced_hal_gpio_select_function`.*

Note: *The function to configure the pin is `wiced_hal_gpio_configure_pin`.*

Note: *The functions to disable and enable the ACLK and PWM are `wiced_hal_aclk_disable`, `wiced_hal_aclk_enable`, `wiced_hal_pwm_disable`, and `wiced_hal_pwm_enable`.*

Note: *See the AIROC™ BTSDK documentation for additional information about the functions.*

☐ 10. If you are impatient, you can speed up testing by changing the values of `low_duty_duration` to `15` in *app_bt_cfg.c. This* will make the advertising timeout faster so you will be able to test the advertising off state without having to wait as long.

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 the kit 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 MySvc Service.

Application Creation

☐ 1. Create a new ModusToolbox™ application for the CYW920835M2EVB-01 BSP.

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. Launch the Bluetooth® Configurator and set the device name to **<inits>_response**.

☐ 3. Save changes and close the configurator.

☐ 4. Open *app_bt_cfg.c* and change the values for `high_duty_duration` and `low_duty_duration` in the `ble_advert_cfg` section to `0`.

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

☐ 5. Open *app.c*.

☐ 6. Make a copy the `app_set_advertisement_data` function and change the name of the new function to app_set_scan_response_data.

Note: *Don't forget to create a function prototype near the beginning of app.c.*

☐ 7. Update the function so that it has a single field with the MySvc UUID.

Note: *The field types can be found in* `wiced_bt_ble_advert_type_t` *which is in wiced_bt_ble.h. In this case, the field type is* `BTM_BLE_ADVERT_TYPE_128SRV_COMPLETE`.

Note: *There is a define called __UUID_SERVICE_MYSVC in cycfg_gatt_db.h (in GeneratedSource) that you can use to create an array containing the Service UUID.*

Note: *Remember that a scan response packet doesn't have the flags field so you should only have one field (i.e. one element).*

☐ 8. At the end of the function, change the function call that sets the raw advertisement data to instead call `wiced_bt_ble_set_raw_scan_response_data` to set up the scan response packet.

☐ 9. In the `BTM_ENABLED_EVT`, call your new function before starting advertisements.

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