Chapter 4A: The Essential Bluetooth LE Peripheral Example

After completing chapter 4A you will have all the required knowledge to create the most basic WICED Bluetooth Low Energy Peripheral.

4A.1 WICED Bluetooth LE System Lifecycle

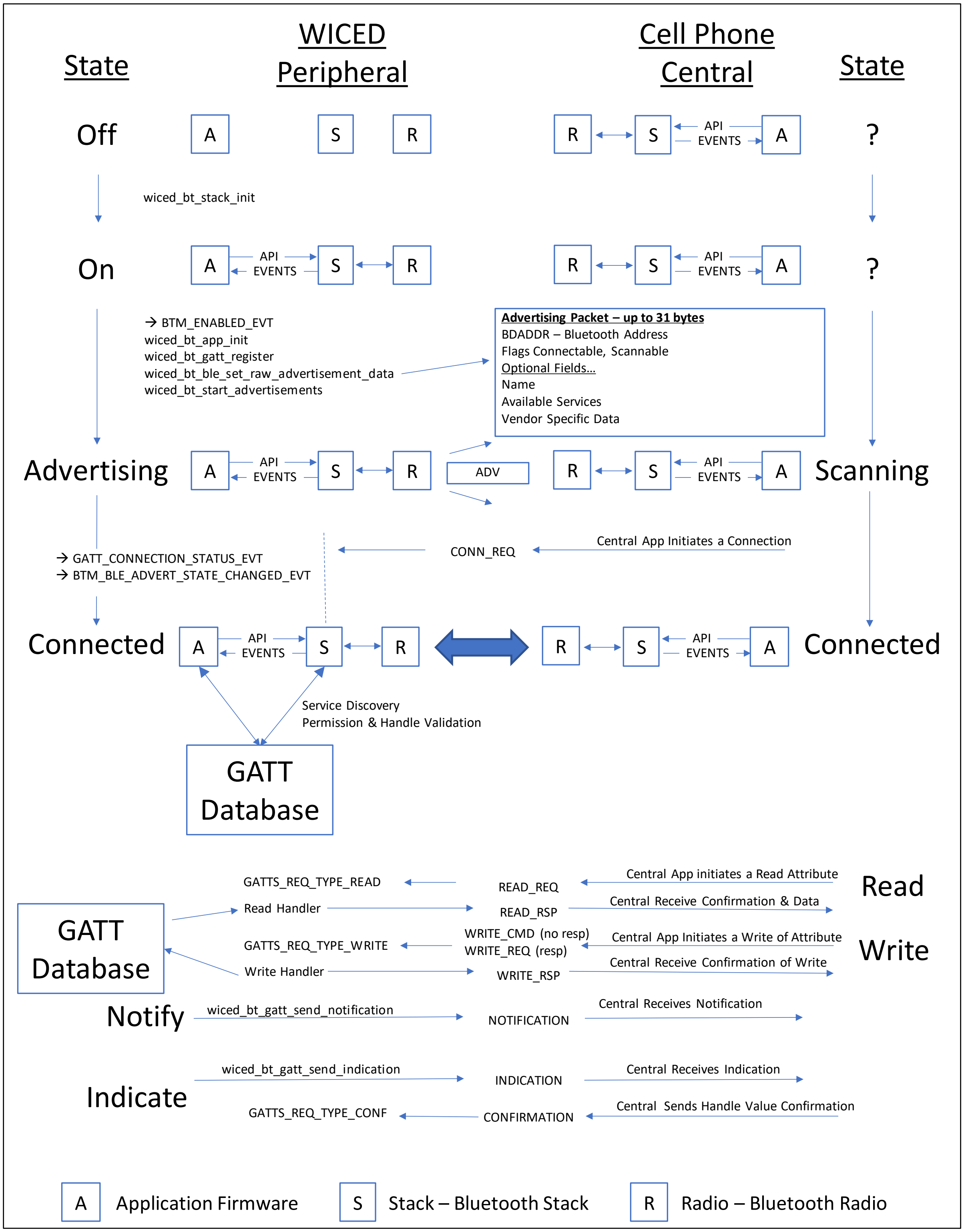
Basically, every book that I have ever read on Bluetooth or Wi-Fi starts with the radio stack and works its way back (or up depending on your point of view) to the Application. You know the drill, 2.4 GHz Digital Spread Spectrum, Adaptive Frequency Hopping, blah blah blah. This approach introduces a bewildering number of technical issues which have almost nothing to do with building your first system. That approach is cool and everything, and it has stuff which eventually you will need to know, but that is not what we are going to do here. In this chapter I am going to give you the absolute minimum that you need to know to write your first WICED Bluetooth LE application that a cell phone App can connect with. Before you launch into this chapter please install CySmart (for Android or Apple iOS) from the appropriate App store and also install the PC version of CySmart on your laptop.

All these wireless 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:

- Turn on the WICED Bluetooth Stack (from now on referred to as "the Stack")
- Start Advertising as connectable
- Process connection events from the Stack
- Process read/write events from the Stack

Here is the overall picture which I will describe in pieces as we go:



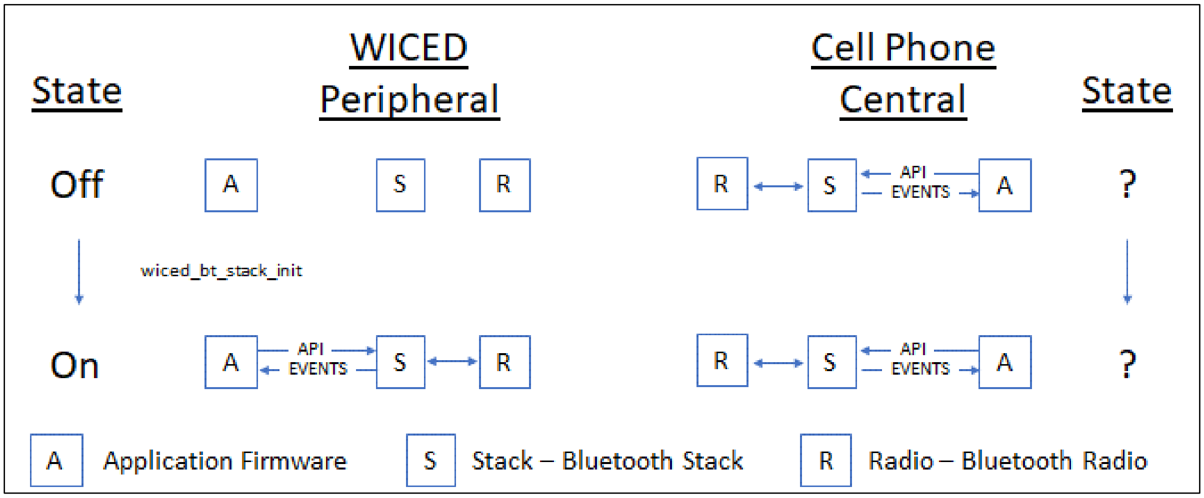
4A.1.1 Turning on the WICED Bluetooth Stack

In the beginning, you have a Bluetooth SoC device and a Cell Phone, and they are not connected, the Stack state is Off, so 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 beacon, a watch, etc. 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).



4A.1.2 Start Advertising

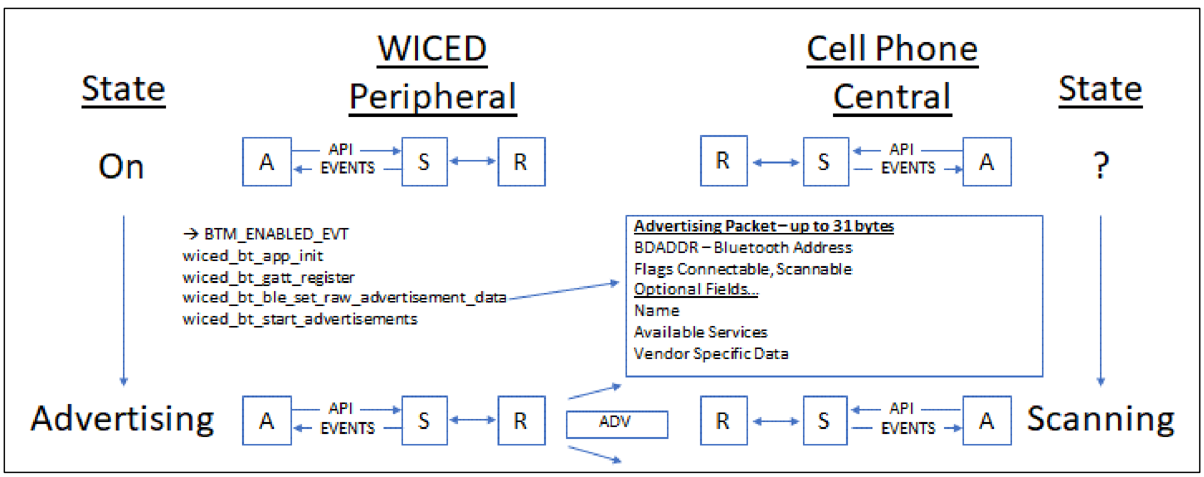
For a Central to know of your existence you need to send out Advertising packets. The Advertising Packet will contain your Bluetooth Device Address (BDA), some flags that include information about your connection availability status, and one or more optional fields for other information, like your device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

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 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. I will discuss ways of improving security later.



The first item in the advertising packet is called Flags. It tells the remote device how to make a connection by identifying the type of Bluetooth supported (Bluetooth LE, Classic, BR/EDR) and the way connections are allowed. The packet can also carry extra information, such as the device name, address, role and so on, but it has a maximum size of 31 bytes.

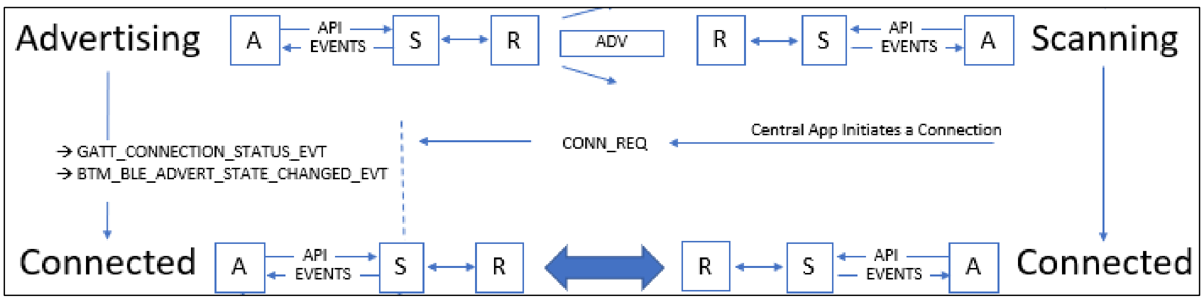
The format of the packet is quite simple. Each item you wish to advertise starts with a length byte, followed by the type (e.g. Flags or Name) and then the data, the size of which is determined by that length byte. The items are simple concatenated together, up to 31 bytes.

4A.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 an API 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 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.



4A.1.4 Exchange Data

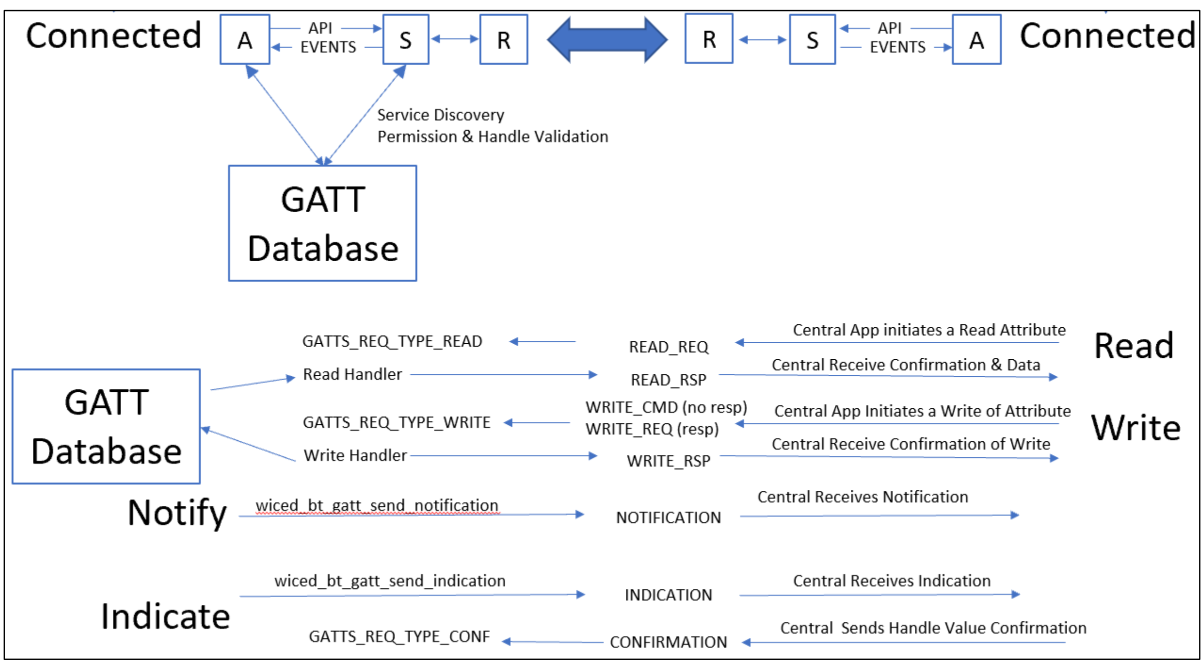
Now that you are connected you need to be able to exchange data. In the world of Bluetooth LE this happens via the Attribute Protocol (ATT). The basic ATT protocol has 4 types of transactions: Read & Write which are initiated by the Client and Notify & Indicate which are initiated by the Server.

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. This leads to the obvious confusion that the Peripheral is the Server and the Central is the Client, 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 in section 4A.7. With all of that, here is the final section of the big picture.



4A.2 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:

```
/** Advertisement data types */
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 bytes to your Advertising packet:

- `9` (the length is 1 for the field type plus 8 for the data)
- `BTM_BLE_ADVERT_TYPE_NAME_COMPLETE`
- `'K', 'e', 'n', 't', 'u', 'c', 'k', 'y'`

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

The `wiced_bt_ble_advert_elem_t` structure is defined as:

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

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

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

```
#define KYNAME "Kentucky"

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

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

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

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

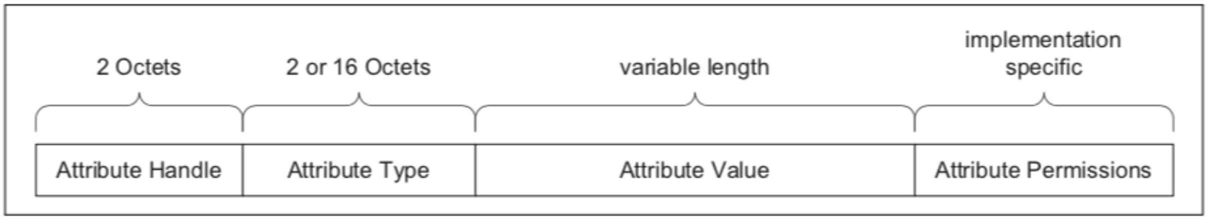
The Advertising packet enables several interesting use cases which we will talk about in more detail in the next chapter.

There is also a scan response packet that can hold an additional 31 bytes which will be discussed two chapters after this.

4A.3 Attributes, the Generic Attribute Profile & GATT Database

4A.3.1 Attributes

As mentioned earlier, the 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 WICED 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. WICED helps you get the permission set correctly when you make the database, and the Stack takes care of enforcing the Permissions.

4A.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 spec'd 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.

4A.3.3 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 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 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)

4A.3.4 Characteristic Declaration in the GATT DB

To declare a Characteristic, you are required to create a minimum two Attributes: the Characteristic Declaration (0x2803) and the Characteristic Value. The Characteristic Declaration creates the property in the GATT database, sets up 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) 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 sends 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 the next chapter.

4A.4 Simple Bluetooth LE Example

For this example, I am going to build a Bluetooth LE application that has one custom service called the "BT101" Service with one writable characteristic called "LED". When the Client writes a 0 or 1 (strictly any non-zero value) into that Characteristic, my application firmware will just write that value into the GPIO driving the LED. You will get to try this yourself in Exercise - 4A.1.

4A.4.1 Bluetooth Configurator

The Bluetooth Configurator is a tool that will build a semi-customized GATT database and device configuration for Bluetooth Low Energy applications. 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*.

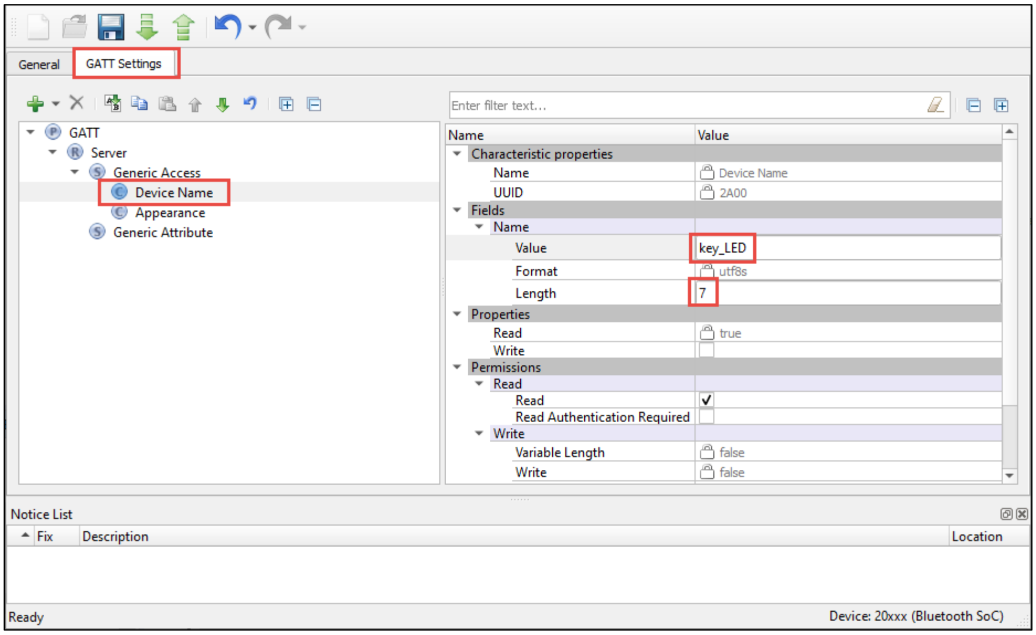
To run the tool, click on **Bluetooth Configurator (new configuration)** in the Quick Panel.

Note Once you have a configuration, the link should just say **Bluetooth Configurator** and clicking the link will open the existing configuration for the application.

Remember, if you are working from the command line or using VS Code, `make config_bt` from the application's directory will run it for you.

This will populate default General and GATT Settings. We will leave the General setting for Bluetooth mode as "Single mode LE", so switch to the GATT Settings tab.

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



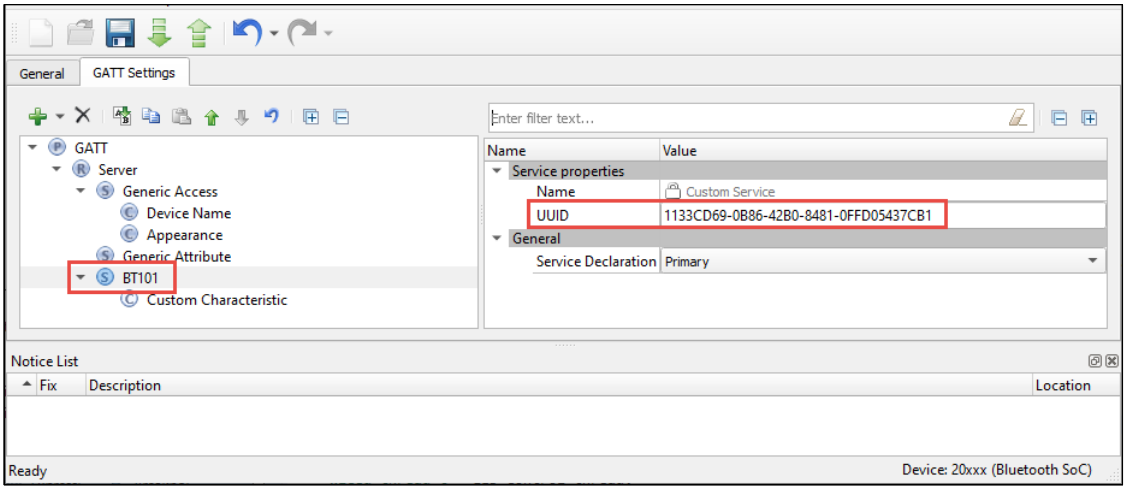
It is important that the name you choose is unique or you will not be able to identify your device when making connections from your cell phone. In this case, I've called the device *key_LED*. When you do this yourself, use a unique device name, such as *<inits>_LED* where *<inits>* is your initials.

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

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

1. Select **Server** in the GATT database.
2. Right-click and choose **Add Service**, then select **Custom Service** (it is near the bottom of the list). A Custom Service entry appears in the GATT database.
3. Right-click on the custom service and select **Rename**. Call the service “BT101”.

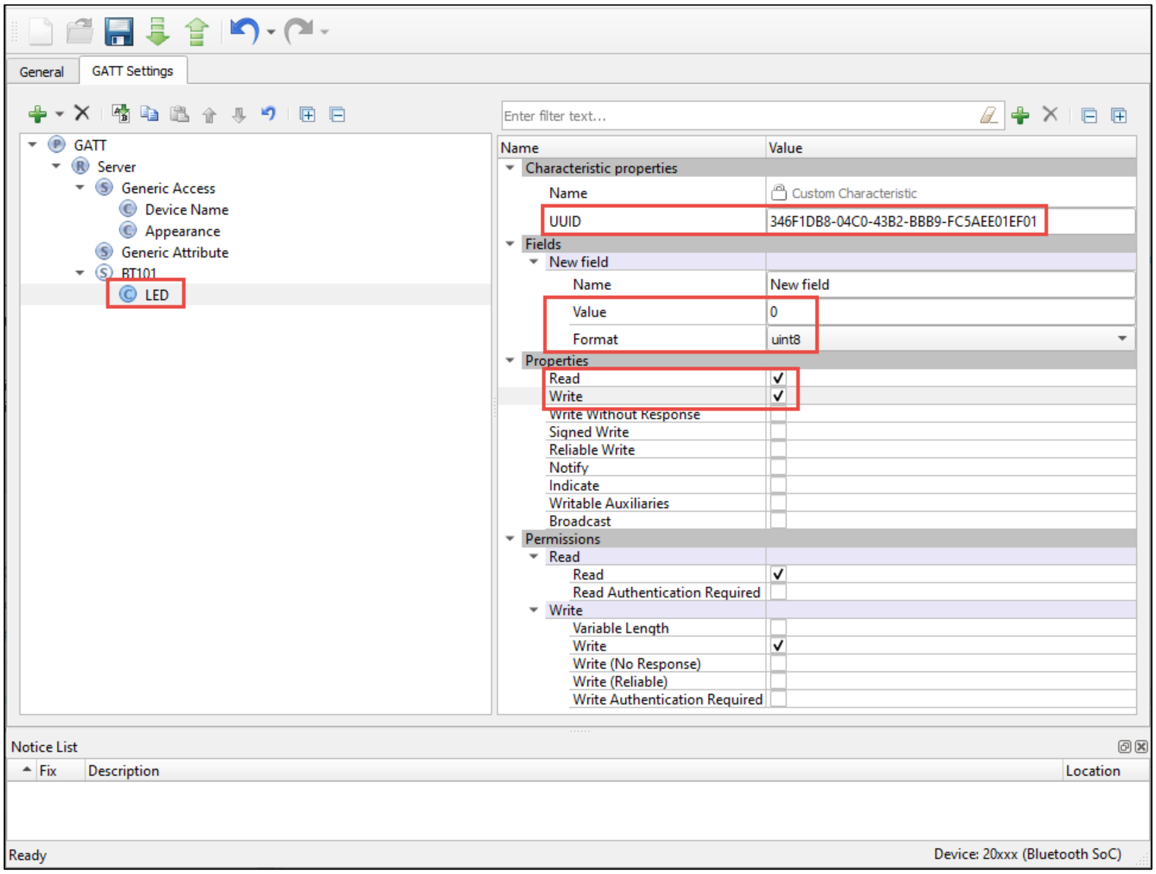
4. The tool will choose a random UUID for this Service, but you could specify your own UUID if desired. For this exercise, just keep the random UUID.



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

1. Right-click on **Custom Characteristic** under the BT101 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** and **Write**. Note that the tool makes the corresponding changes to the **Permissions** section for you, so you don't need to set them unless you need an unusual combination of Properties and Permissions.

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



6. Click the **Save** button to save the file *design.cybt*. Note that 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.
7. 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.

4A.4.2 Generated Code

The configurator creates two files - *cycfg_gatt_db.c* and *cycfg_gatt_db.h* in the *GeneratedSource* directory as well as a timestamp file. Among other things, the header file includes the following defines for our BT101 Service and its LED.

```
/* Service BT101 */
#define HDLS_BT101                                    0x07
/* Chara cteristic LED */
#define HDLC_BT101_LED                                0x08
#define HDLC_BT101_LED_VALUE                          0x09
```

The C source contains the GATT database structure and some global variables like the device name and led characteristic value (initialized to 0), which you will use later.

```
const uint8_t gatt_database[] =
{
    /* Primary Service: Generic Access */
    PRIMARY_SERVICE_UUID16 (HDLS_GAP, __UUID_SERVICE_GENERIC_ACCESS),
        /* Characteristic: Device Name */
        CHARACTERISTIC_UUID16 (HDLC_GAP_DEVICE_NAME, HDLC_GAP_DEVICE_NAME_VALUE,
        __UUID_CHARACTERISTIC_DEVICE_NAME, LEGATTDB_CHAR_PROP_READ,
        LEGATTDB_PERM_READABLE),
        /* Characteristic: Appearance */
        CHARACTERISTIC_UUID16 (HDLC_GAP_APPEARANCE, HDLC_GAP_APPEARANCE_VALUE,
        __UUID_CHARACTERISTIC_APPEARANCE, LEGATTDB_CHAR_PROP_READ,
        LEGATTDB_PERM_READABLE),

    /* Primary Service: Generic Attribute */
    PRIMARY_SERVICE_UUID16 (HDLS_GATT, __UUID_SERVICE_GENERIC_ATTRIBUTE),

    /* Primary Service: BT101 */
    PRIMARY_SERVICE_UUID128 (HDLS_BT101, __UUID_SERVICE_BT101),
        /* Characteristic: LED */
        CHARACTERISTIC_UUID128_WRITABLE (HDLC_BT101_LED, HDLC_BT101_LED_VALUE,
        __UUID_CHARACTERISTIC_BT101_LED, LEGATTDB_CHAR_PROP_READ |
        LEGATTDB_CHAR_PROP_WRITE, LEGATTDB_PERM_READABLE |
LEGATTDB_PERM_WRITE_REQ),
};

/* Length of the GATT database */
const uint16_t gatt_database_len = sizeof(gatt_database);

/*********************************************************************************
***
 * GATT Initial Value Arrays

*********************************************************************************
*/

uint8_t app_gap_device_name[] = {'k', 'e', 'y', '_', 'L', 'E', 'D', };
uint8_t app_gap_appearance[]  = {0x00, 0x00, };
uint8_t app_bt101_led[]       = {0x00, };
```

4A.4.3 Editing the Firmware

The template for the exercises in this chapter includes a little bit of setup code for the `BTM_ENABLED_EVT` and some very helpful functions, as follows.

- `app_bt_management_callback` is the callback function that you edited in chapter 2. The `BTM_ENABLED_EVT` code now prints the Bluetooth Device Address (BDA), sets up the GATT database, and starts advertising for a connection.
- `app_gatt_callback` handles GATT events such as connect/disconnect and attribute read/write requests.
- `app_set_advertisement_data` creates the advertising packet that includes the device name you will see in the CySmart app.
- `app_gatt_get_value` searches the GATT database for the requested characteristic and extracts the value. We use this function to read the state of the LED.
- `app_gatt_set_value` searches the GATT database for the requested characteristic and updates the value. We use this function to write the state of the LED into the database and, later, notify the central device.

Follow these instructions to configure the device behavior. Note that the template sets up the PUART for debugging traces so you can use `WICED_BT_TRACE` to better understand how the Stack is behaving.

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

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

2. Template code for the `BTM_ENABLED_EVT` case in `app_bt_management_callback` reads and reports the 6-byte Bluetooth Device Address (BDA) in the terminal when the Stack gets enabled. Note that this 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/20819A1/<version>/platforms/208XX_OCF.btp

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

```
DLConfigBD_ADDRBase = "20819A1*****"
```

   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 20819A1 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; 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.

3. 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 );
```

4. Next, I don't want to allow pairing to the device just yet so configure the pairing mode with the parameters set to `WICED_FALSE`:

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

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

The following edits control what the firmware does during GATT read and write requests.

5. Add the following case in `app_gatt_get_value` 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. Note that the code uses the GATT database value, not the state of the pin itself, and so non-zero implies “on” and zero means “off”. The read itself is handled by the code that is already provided – what you are adding is just to print a message to the UART so that you will know when the client read the value.

```
// TODO Ex 01: Add code for any action required when this attribute is read
switch ( attr_handle )
{
        case HDLC_BT101_LED_VALUE:
                WICED_BT_TRACE( "LED is %s\r\n", app_bt101_led[0] ? "ON" : "OFF"
                );
                break;
}
```

6. In `app_gatt_set_value`, notice how the template function automatically updates the GATT database with a call to `memcpy`. There is no need to write to the `app_bt101_led` array.

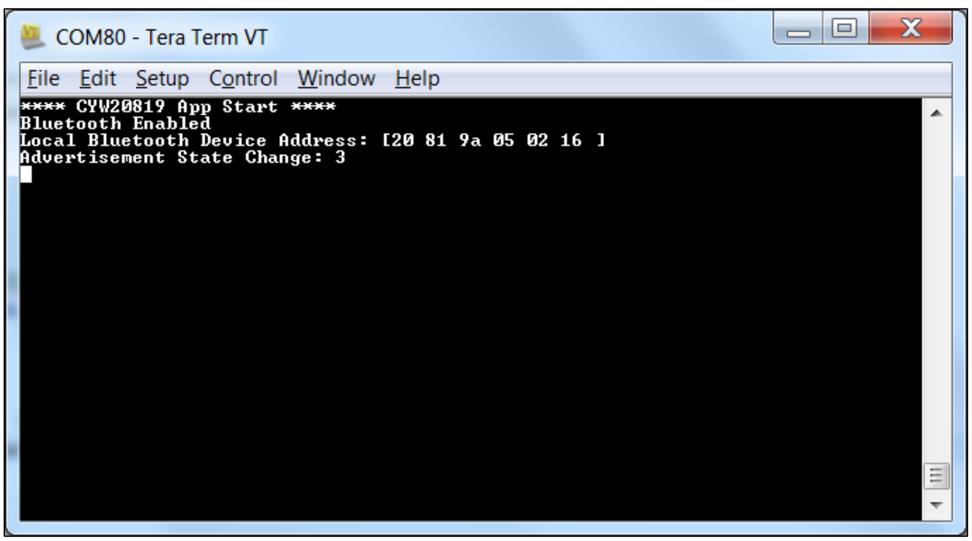
```
// Value fits within the supplied buffer; copy over the value
app_gatt_db_ext_attr_tbl[i].cur_len = len;
memcpy(app_gatt_db_ext_attr_tbl[i].p_data, p_val, len);
res = WICED_BT_GATT_SUCCESS;
```

7. Add the following case in `app_gatt_set_value` to update the LED and printout the result. Again, the switch statement is in the template – just add the new case. This event will occur whenever the Central writes the LED characteristic. We are going to use 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.

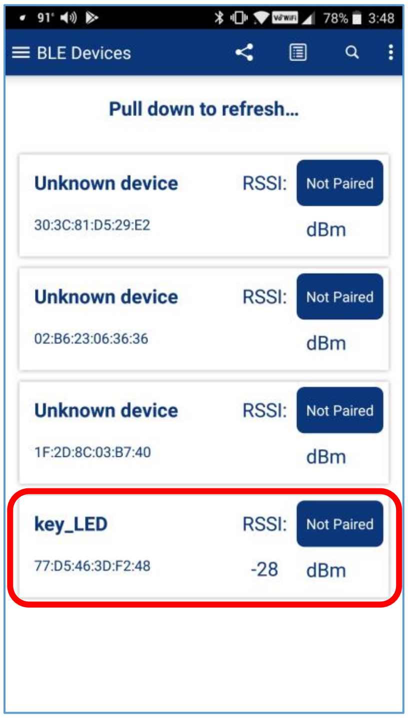
```
// TODO Ex01: Add code for any action required when this attribute is written
// For example, you may need to write the value into NVRAM if it needs to be
persistent
switch ( attr_handle )
{
        case HDLC_BT101_LED_VALUE:
                wiced_hal_gpio_set_pin_output(LED2, app_bt101_led[0] == 0 );
                WICED_BT_TRACE( "Turn the LED %s\r\n", app_bt101_led[0] ? "ON" :
                "OFF" );
                break;
}
```

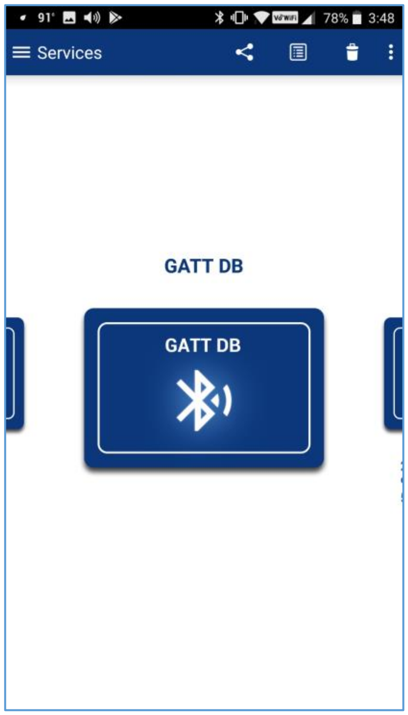
4A.4.4 Testing the Application

Start up a UART terminal (115200, 8, 1, N), then build and program your kit. When the application firmware starts up you see some messages.

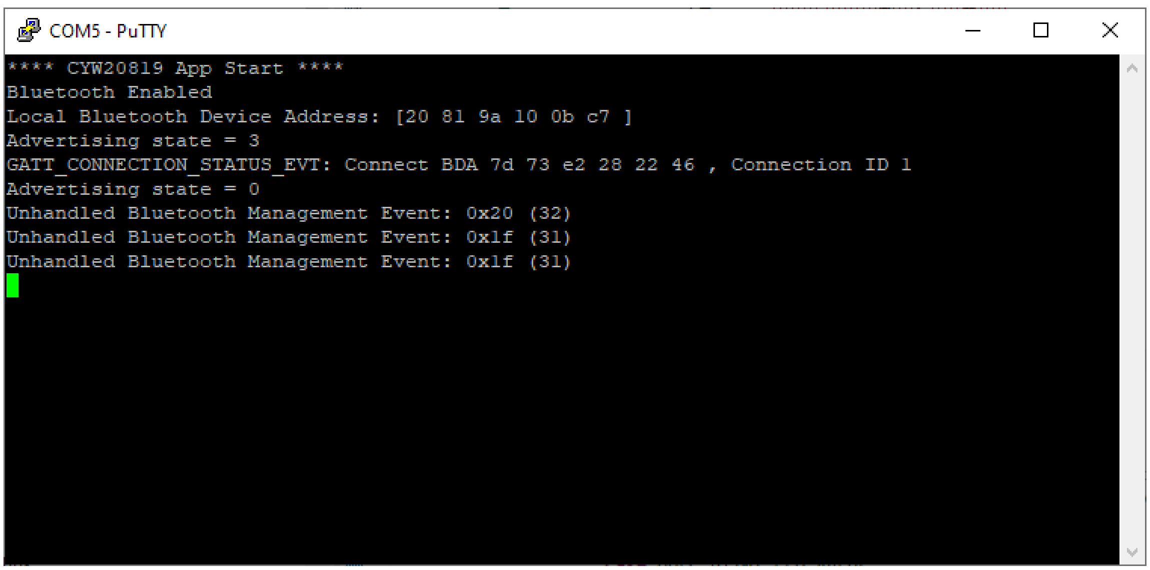


Run CySmart on your phone (more details on CySmart later on). When you see the "<inits>_LED" device, tap on it. CySmart will connect to the device and will show the GATT browser widget.



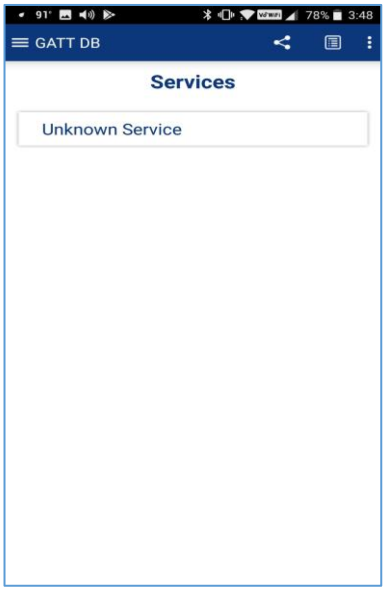


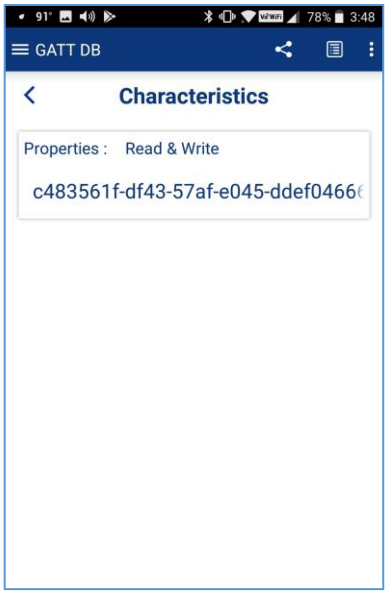
On the terminal window, you will see that there has been a connection and the advertising has stopped.



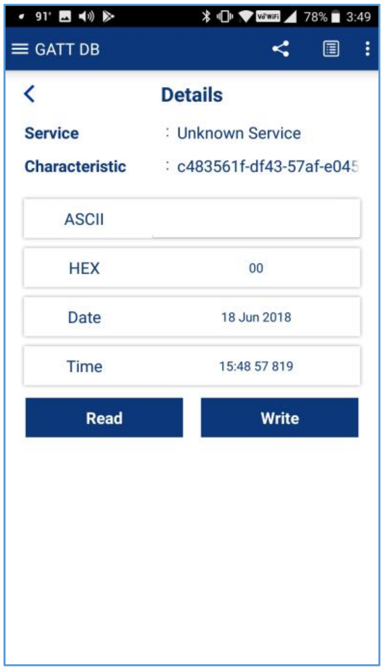
Back in CySmart, tap on the GATT DB widget to open the browser. You will see an Unknown Service (which I know is BT101). Tap on the Service and CySmart will tell you that there is a Characteristic with the UUID shown (which I know is LED).

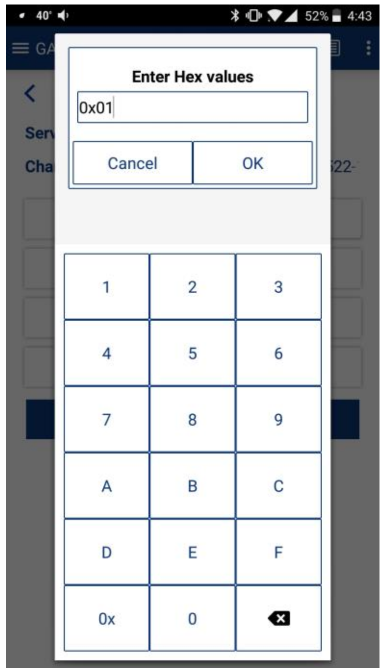
Note: In the iOS version of CySmart, the Characteristic UUID will not be shown – it will just say "Unknown Characteristic".



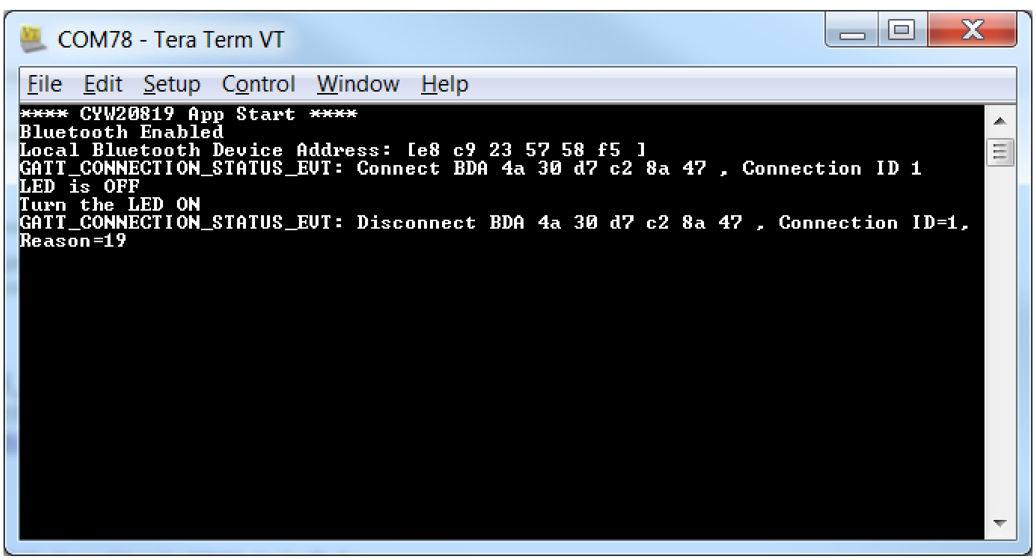


Tap on the Service to see details about it. First, tap the Read button and you will see that the current value is 0. Now you can Write 1s or 0's into the Characteristic and you will find that the LED turns on and off accordingly.





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



In the next several sections we will walk you through the code.

4A.5 WICED Bluetooth Stack Events

The Stack generates Events based on what is happening in the Bluetooth world. After an event is created, the Stack will call the callback 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. The Stack will generate more events than are needed for the first simple example, and I will deal with them in the next chapter.

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

4A.5.1 Essential Bluetooth Management Events

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

The ModusToolbox starter template code (*app.c*) for this class provides and registers a function called `app_bt_management_callback` (or similar) to handle Management events.

4A.5.2 Essential GATT Events

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

The ModusToolbox starter template code (*app.c*) for this class provides and registers a function called `app_gatt_callback` (or similar) to handle GATT events.

4A.5.3 Essential GATT Sub-Events

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

GATT_CONNECTION_STATUS_EVT

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

Event	Description
`connected == WICED_TRUE`	A GATT connection has been established.
`connected != WICED_TRUE`	A GATT connection has been broken.

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

GATT_ATTRIBUTE_REQUEST_EVT

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

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

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

4A.6 WICED Bluetooth Firmware Architecture

At the very beginning of this chapter I told you that there are four steps to make a basic WICED Bluetooth LE Peripheral:

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

The Bluetooth template provided for this class mimics this flow.

4A.6.1 Turning 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 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.

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

When you start the Stack, it generates the `BTM_ENABLED_EVT` event and calls the `app_bt_management_callback` function which then processes that event.

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

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

4A.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 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 does not do anything when advertising is started, but you could, for instance, turn on an LED to indicate the advertising state.

4A.6.3 Processing Connection Events from the Stack

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

The Stack responds to the Central with a connection accepted message.

The Stack then generates a GATT event called `GATT_CONNECTION_STATUS_EVT` which is processed by the `app_gatt_callback` function.

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

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

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

4A.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`

4A.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`

4A.7 WICED GATT Database Implementation

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

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

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

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

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

The WICED 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.

4A.7.1 gatt_database[]

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

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

- `PRIMARY_SERVICE_UUID16(handle, service)`
- `PRIMARY_SERVICE_UUID128(handle, service)`
- `SECONDARY_SERVICE_UUID16(handle, service)`
- `SECONDARY_SERVICE_UUID128(handle, service)`
- `INCLUDE_SERVICE_UUID16(handle, service_handle, end_group_handle, service)`
- `INCLUDE_SERVICE_UUID128(handle, service_handle, end_group_handle)`

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

```
/* Service Generic Access */
#define HDLS_GAP                                      0x0001u

/* Service Generic Attribute */
#define HDLS_GATT                                     0x0006u

/* Service BT101 */
#define HDLS_BT101                                    0x0007u
```

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

```
#define __UUID_SERVICE_BT101 0xD5u, 0x8Eu, 0x79u, 0x8Bu, 0x2Cu, 0xDEu, 0x11u,
0x89u, 0x45u, 0x47u, 0x5Au, 0x31u, 0x6Au, 0xA3u, 0xFAu, 0x34u
```

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

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

- `CHARACTERISTIC_UUID16(handle, handle_value, uuid, properties, permission)`
- `CHARACTERISTIC_UUID128(handle, handle_value, uuid, properties, permission)`
- `CHARACTERISTIC_UUID16_WRITABLE(handle, handle_value, uuid, properties, permission)`
- `CHARACTERISTIC_UUID128_WRITABLE(handle, handle_value, uuid, properties, permission)`

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

```
/* Characteristic LED */
#define HDLC_BT101_LED                          0x0008u
#define HDLC_BT101_LED_VALUE                    0x0009u
```

The `_VALUE` parameter is the Handle of the Attribute that will hold the Characteristic's Value.

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

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

```
/*  GATT Characteristic Properties */
#define LEGATTDB_CHAR_PROP_BROADCAST                    (0x1 << 0)
#define LEGATTDB_CHAR_PROP_READ                         (0x1 << 1)
#define LEGATTDB_CHAR_PROP_WRITE_NO_RESPONSE            (0x1 << 2)
#define LEGATTDB_CHAR_PROP_WRITE                        (0x1 << 3)
#define LEGATTDB_CHAR_PROP_NOTIFY                       (0x1 << 4)
#define LEGATTDB_CHAR_PROP_INDICATE                     (0x1 << 5)
#define LEGATTDB_CHAR_PROP_AUTHD_WRITES                 (0x1 << 6)
#define LEGATTDB_CHAR_PROP_EXTENDED                     (0x1 << 7)
```

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

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

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

4A.7.2 gatt_db_ext_attr_tbl

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

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

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

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

gatt_db_lookup_table_t app_gatt_db_ext_attr_tbl[] =
{
    /* { attribute handle,  maxlen, curlen, attribute data } */
    { HDLC_GAP_DEVICE_NAME_VALUE, 8,      8,      app_gap_device_name },
    { HDLC_GAP_APPEARANCE_VALUE, 2,      2,      app_gap_appearance },
    { HDLC_BT101_LED_VALUE, 1,      1,      app_bt101_led },
};
```

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

4A.7.3 uint8_t Arrays for the Values

Bluetooth Configurator will generate arrays of `uint8_t` to hold the values of writable/readable Attributes. You will find these values in a section of the code in *cycfg_gatt_db.c* marked with a comment "GATT Initial Value Arrays". In the example below, you can see there is a Characteristic with the name of the device, a Characteristic with the GAP appearance, and the LED Characteristic.

```
/*************************************************************************************
 * 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_bt101_led[]    = {0x00u, };
```

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

4A.7.4 The Application Programming Interface

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

```
wiced_bt_gatt_status_t app_gatt_get_value( wiced_bt_gatt_attribute_request_t *p_attr );

wiced_bt_gatt_status_t app_gatt_set_value( wiced_bt_gatt_attribute_request_t *p_attr );
```

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

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

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

You are supposed to return a `wiced_bt_gatt_status_t` which will tell the Stack what to do next. Assuming things works this function will return `WICED_BT_GATT_SUCCESS`. In the case of a Write this will tell the Stack to send a WRITE Response indicating success to the Client.

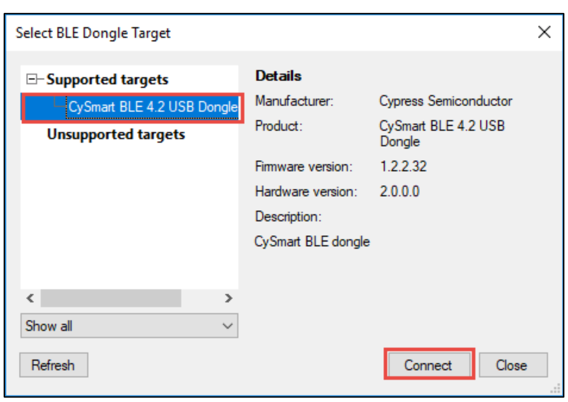
4A.8 CySmart

Cypress provides a PC and mobile device application (Android and iOS) called CySmart which can be used to scan, connect, and interact with services, characteristics, and attributes of Bluetooth LE devices.

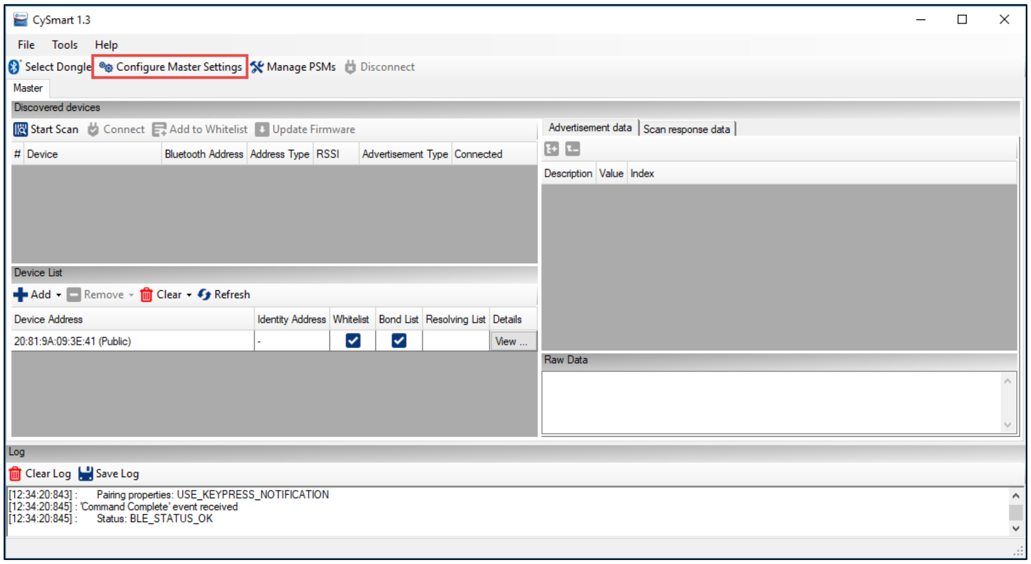
There are other utilities available for iOS and Android (such as Lightblue Explorer) which will also work. Feel free to use one of those if you are more comfortable with it.

4A.8.1 CySmart PC Application

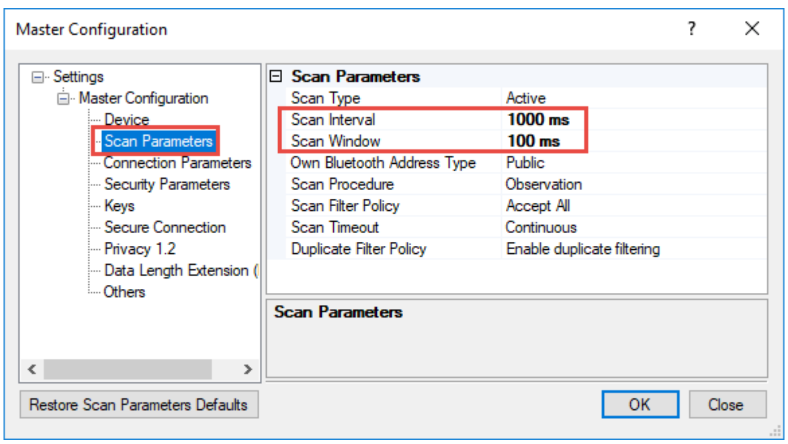
To use the CySmart PC Application, a CY5677 CySmart USB Dongle is required. When CySmart is started, it will search for supported targets and will display the results. Select the dongle that you want to use and click on **Connect**.



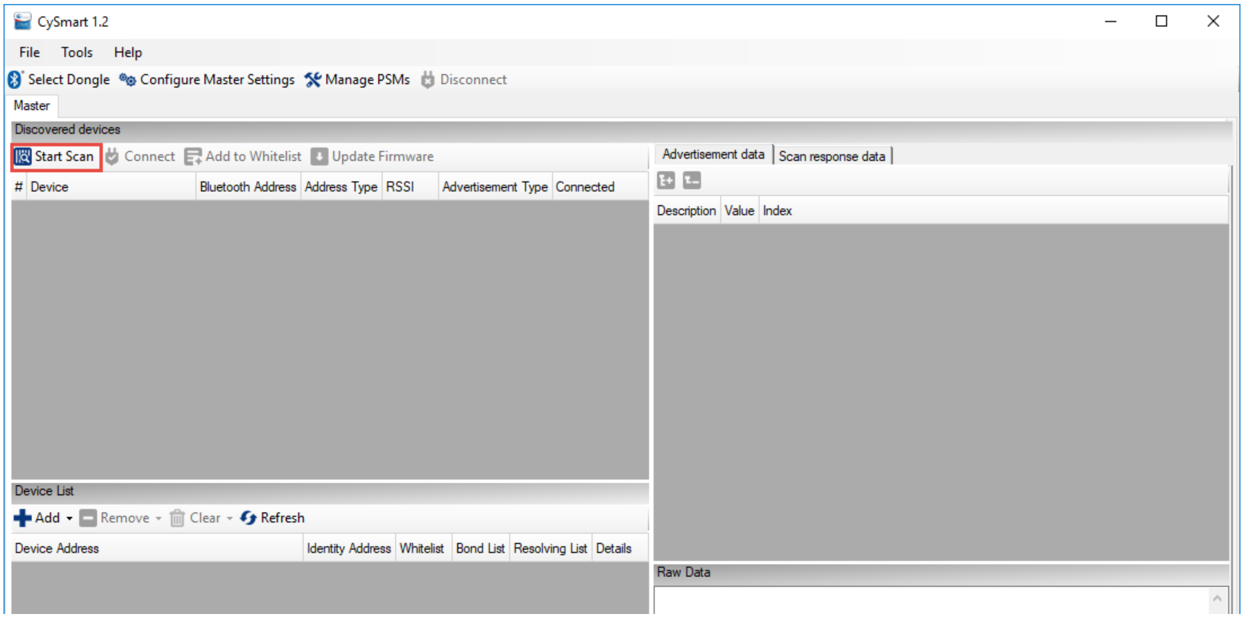
Once a dongle is selected, the main window will open as shown below.



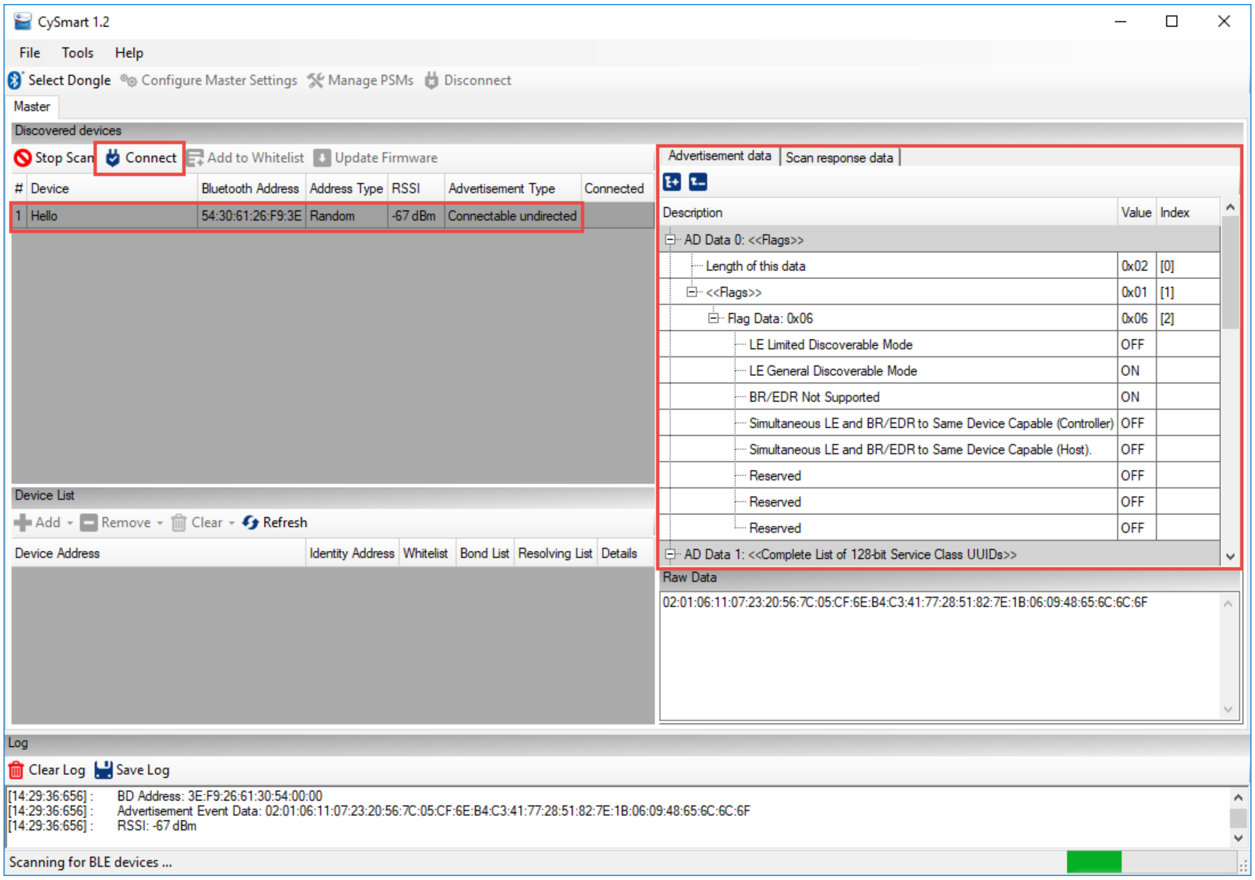
<u>Before starting a scan, it is a good idea to configure the master settings so that scanning is done less frequently. This is especially important in a class environment where there may be many devices advertising at the same time</u>. The tool may act strangely if it is trying to scan too fast. It is recommended to set the Scan Interval to 1000 ms and the Scan Window to 100 ms. Note that these settings are NOT saved when you close CySmart so you will need to set them each time you restart it.



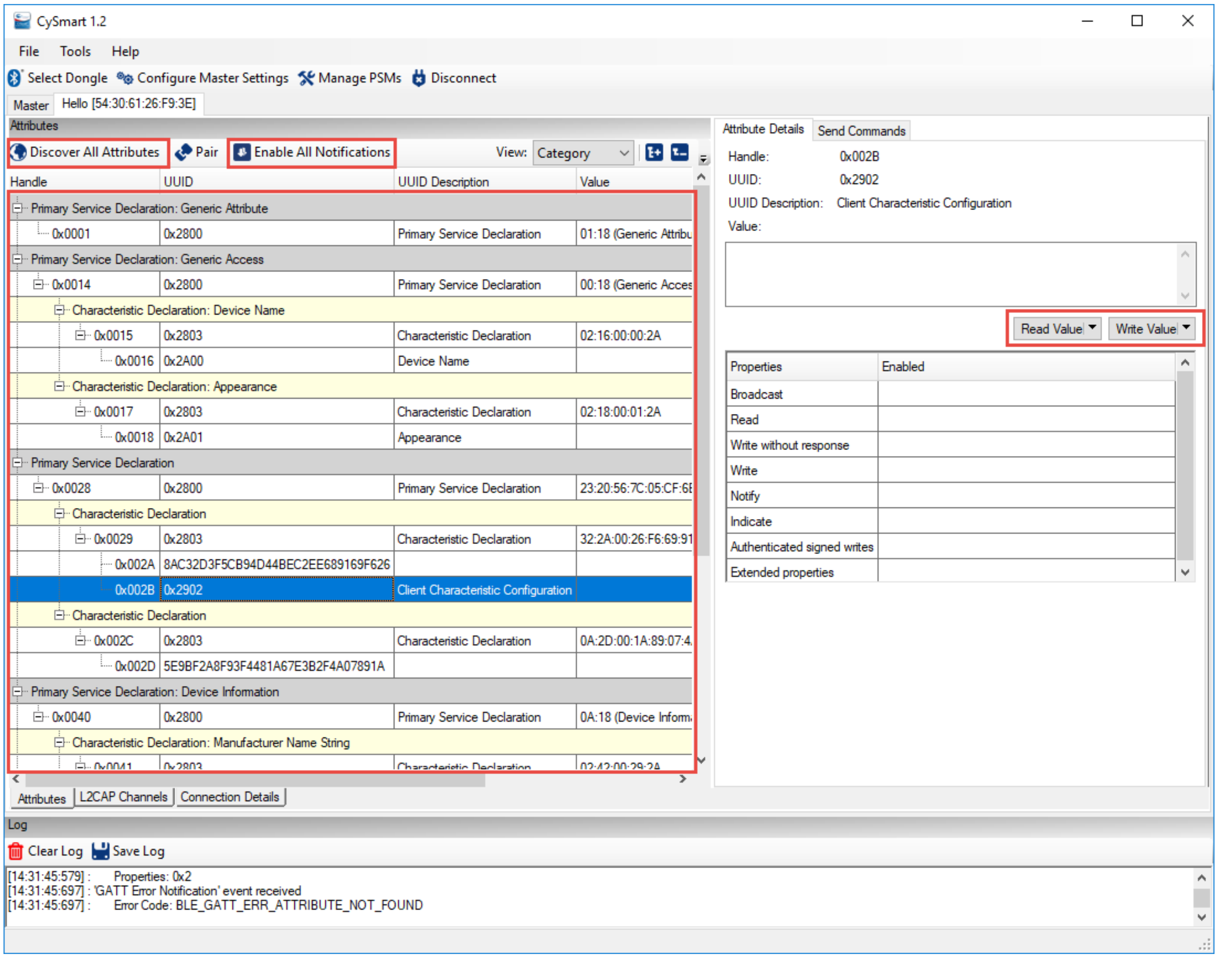
Once you click **OK** to close the Master Configuration widow, click on **Start Scan** from the main window to search for advertising Bluetooth LE devices.



Once the device that you want to connect to appears, click on **Stop Scan** and then click on the device you are interested in. You can then see its Advertisement data and Scan response data in the right-hand window. Click **Connect** to connect to the device.



When the device is connected, click on **Pair** and then **Discover All Attributes**. Once that is complete, you will see a representation of all Services, Characteristics, and Attributes from the GATT database. You can read and write values by clicking on an attribute and using the buttons in the right-hand window. Click **Enable All Notifications** if you want to see real-time value updates in the left-hand window for characteristics that have notification capability.



The complete user guide for the CySmart PC application can be opened in the tool under **Help > Help Topics**. It can also be found on the CySmart website at:

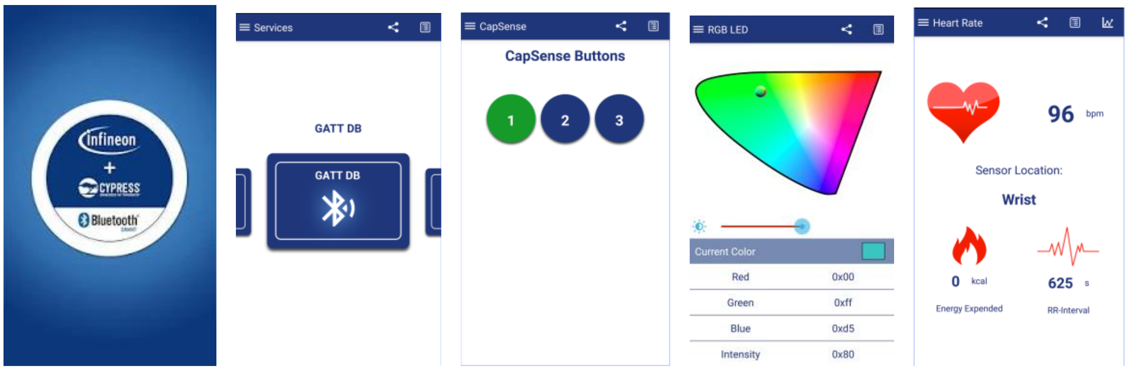
http://www.cypress.com/documentation/software-and-drivers/cysmart-bluetooth-le-test-and-debug-tool

Scroll down to the Related Files section of the page to find the User Guide.

4A.8.2 CySmart Mobile Application

The CySmart mobile application is available on the Google Play store and the Apple App store. The app 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 Cypress 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.



Complete documentation and source code can be found on the CySmart Mobile App website at:

http://www.cypress.com/documentation/software-and-drivers/cysmart-mobile-app

Documentation of the Cypress custom profiles supported by the tool can be found at:

http://www.cypress.com/documentation/software-and-drivers/cypresss-custom-ble-profiles-and-services

4A.9 Exercises

Exercise - 4A.1 Create a Bluetooth LE Application with a BT101 Service

☐

1. Use the ch04a template to create an application called **ch04a_ex01_ble**.

   Follow the instructions in section 4A.4 to use the Bluetooth Configurator to set up a Service called “BT101” with a Characteristic called “LED” that allows an LED on the kit to be controlled from your phone using CySmart.

   Hint The template *app.c* file has comments marked with `TODO` for locations that need changes for exercises 1 and 2.

Exercise - 4A.2 Implement a connection status LED

Introduction

In this exercise, you will implement a connection status LED that is:

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

Hint Use LED1 for the connection status since LED2 is used by other applications for the LED Characteristic.

Application Creation

1. Use the ch04a template to create an application called **ch04a_ex02_status**.
2. Launch the Bluetooth Configurator.
    a. Set the device name to <inits>_status. Make sure you hit enter or click in a field outside the Value. If not, the new value may not be saved and the length may not be updated.
    b. Save your changes and close the configurator. We are only using GATT to handle connections in this exercise and so there is no need to add a service.
3. Open the *Makefile* and set `BT_DEVICE_ADDRESS?=random`.

    Hint The template *app.c* file has comments marked with `TODO` for locations that need changes for exercises 1 and 2.
4. Open *app.c* and add `#include "cycfg_gatt_db.h"`

    Hint If you don't see *cycfg_gatt_db.h* in the *GeneratedSource* directory, right click on the directory and select **Refresh**.
5. Add `#include "wiced_hal_pwm.h"`
6. Add `#include "wiced_hal_aclk.h"`
7. In the `BTM_ENABLED_EVT` case, register/initialize the GATT database and disallow pairing.

```
/* Register the GATT callback and initialize the GATT database */
wiced_bt_gatt_register( app_gatt_callback );
wiced_bt_gatt_db_init( gatt_database, gatt_database_len );
/* Disable Pairing */
wiced_bt_set_pairable_mode( WICED_FALSE, WICED_FALSE );
```

8. In the `BTM_ENABLED_EVT` case:
    a. Configure the PWM to connect to LED1.
    b. Start the ACLK and the PWM.

c. Set the compare value to match the maximum value so that the LED is always off.

```
/* Start the PWM in the LED always off state */
wiced_hal_gpio_configure_pin(LED1,
GPIO_OUTPUT_ENABLE, GPIO_PIN_OUTPUT_LOW);
wiced_hal_gpio_select_function(LED1, WICED_PWM0);
wiced_hal_aclk_enable( PWM_FREQUENCY, ACLK1, ACLK_FREQ_24_MHZ );
wiced_hal_pwm_start( PWM0, PMU_CLK, PWM_ALWAYS_OFF, PWM_INIT, 0 );
```

d. **Hint** Use the pre-defined macros for `PWM_ALWAYS_ON`, `PWM_ALWAYS_OFF`, and `PWM_TOGGLE` from the template to make the PWM code easier to write.

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

f. **Hint** We could have used the device configurator to connect the PWM to LED1 but we decided to do it in the code for this case. Either way is OK.

□ 9. Declare a global `uint16_t` variable to keep track of the connection ID (call it `connection_id`) and initialize to `0`.

Hint This will be needed so that when advertisements stops, you will know if the LED should be turned ON (connected) or OFF (not connected).

□ 10. Set/clear the connection ID variable at the appropriate places.

Hint Look in the GATT connect callback function.

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

□ 11. Turn the LED ON or OFF when advertising stops based on the connection ID.

Hint In the `BTM_BLE_ADVERT_STATE_CHANGED_EVT` case, create a switch statement for `p_event_data->ble_advert_state_changed` that handles the following cases; `BTM_BLE_ADVERT_OFF` (sets the LED on or off based on `connection_id`), `BTM_BLE_ADVERT_UNDIRECTED_HIGH` and `BTM_BLE_ADVERT_UNDIRECTED_LOW` (both set the PWM to toggle).

Hint Use `wiced_hal_pwm_change_values` and make use of the macros `PWM_ALWAYS_OFF`, `PWM_ALWAYS_ON`, and `PWM_TOGGLE` that are provided in the template.

□ 12. **Note**: If you are impatient, like us, you can speed up your testing by changing the values of `high_duty_duration` and `low_duty_duration` to `15` in *app_bt_cfg.c*, which will make the Stack change advertising states much faster.

Testing

□ 1. Program the application to your kit.

2. Use the PC version of CySmart to connect to the kit. Observe the state of LED1 in three states: (1) when not advertising; (2) when advertising; and (3) when a connection is active.

    a. **Hint** You must have a CY5677 CySmart Bluetooth LE USB dongle connected to your PC to run CySmart.

    b. **Hint** Don't forget to update the scan interval and window to 1000 and 100 ms respectively.

    c. **Hint** You will have to wait for the advertising timeout while not connected to see the first case.

Exercise - 4A.3 Create a Bluetooth LE Advertiser

Introduction

In this exercise, you will create an application that will send out advertisement packets but will not allow any connections. This is common for devices like beacons or locator tags. The advertisement packet will include the flags, complete name, appearance and three bytes of manufacturer specific data. Each time a button is pressed on the kit, the value of the manufacturer data will be incremented, and advertisements will be re-started.

Below is a table showing the events that occur during this exercise. Arrows indicate the cause/effect of the Stack events.

External Event	BLE Stack Event	Action
Board reset >	`BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT >`	Not used yet
	`BTM_ENABLED_EVT >`	Initialize application, start the button interrupt
	`BTM_BLE_ADVERT_STATE_CHANGED_EVT (BTM_BLE_ADVERT_NONCONN_HIGH)`	< Start advertising
Scan for devices in CySmart PC application. Look at advertising data.		
Press MB1.	`BTM_BLE_ADVERT_STATE_CHANGED_EVT (BTM_BLE_ADVERT_NONCONN_HIGH)`	< Update information in the advertising packet and restart advertising
Re-start scan in CySmart. Look at new advertising data.		
Wait for timeout >	`BTM_BLE_ADVERT_STATE_CHANGED_EVT (BTM_BLE_ADVERT_NONCONN_LOW)`	Stack switches to lower advertising rate to save power

Application Creation

1. Use the ch04a template to create an application called **ch04a_ex03_adv**.
2. Launch the Bluetooth Configurator.
    a. Set the device name to <inits>_adv. Make sure you hit enter or click in a field outside the Value. If not, the new value may not be saved, and the length may not be updated.
    b. In the "Appearance" characteristic, choose the "Generic Tag" type.
    c. Save your changes and close the configurator.
3. Open the *Makefile* and set `BT_DEVICE_ADDRESS?=random`.

   Hint The template *app.c* file does NOT have `TODO` comments for the locations to change for the remaining exercises. You should have become familiar enough with the firmware by now to find the right places to change.
4. Open *app.c* and add `#include "cycfg_gatt_db.h"`
5. Locate the line in the main C file that starts advertisements.

   Change the advertisement type to `BTM_BLE_ADVERT_NONCONN_HIGH` because we don't want the device to be connectable.

   Hint Right click on the existing advertisement type and select **Open Declaration** to see all the available choices. You may have to build the application and/or do an **Index > Rebuild** before this will work.
6. Locate the function that sets up the advertisement data and add a new element to send Cypress' unique manufacturer ID and a count value.
    a. **Hint** Create a global `uint8_t` array of size three. Set the first two values equal to `0x31` and `0x01`. The third value will hold the count value.
        i. The Cypress manufacturer ID assigned by the Bluetooth SIG is `0x0131`. The value is little endian in the advertising packet which is why the first two bytes are `0x31` and `0x01`.
    b. **Hint** The advertisement type for this element should be `BTM_BLE_ADVERT_TYPE_MANUFACTURER`.
    c. **Hint** Don't forget to increase the number of elements in the advertising data array.
7. Configure `USER_BUTTON1` for a falling edge interrupt in the `BTM_ENABLED_EVT`. Add a button interrupt callback to do the following:
    a. Clear the pin interrupt.
    b. Increment the third byte of the array holding the manufacturer's data (i.e. the count value).

c. Update the advertisement packet data array.

Hint You can just call the function that sets up the advertising packet again.

Testing

1. Program the application to the board and use the PC version of CySmart to examine the advertisement packets.
2. Start scanning and the stop once you see your device listed.
3. Then click on your device to see its advertisement data.
4. Press the button, re-start/stop the scan, and look at your device's scan response to see that the value has incremented.

   Hint Don't forget to update the scan interval and window to 1000 and 100 ms respectively.

Questions

1. How many bytes is the advertisement packet?

Exercise - 4A.4 Connect using Bluetooth LE

Introduction

In this exercise, you will create an application that will have a custom Service called "BT101" containing two Characteristics:

- A Button characteristic with the state of the button on the kit
- An LED Characteristic to control an LED.

You will monitor the button on the kit board and update its state in a GATT Characteristic so that a client can read the value. The LED Characteristic will behave like the LED in exercise 01 – you will be able to Read and Write the LED state from a client to control the LED on the board.

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

External Event	BLE Stack Event	Action
Board reset >	`BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT >`	Not used yet
	`BTM_ENABLED_EVT >`	Initialize application
	`BTM_BLE_ADVERT_STATE_CHANGED_EVT (BTM_BLE_ADVERT_ UNDIRECTED _HIGH)`	< Start advertising
CySmart will now see advertising packets		
Connect to device from CySmart >	`GATT_CONNECTION_STATUS_EVT >`	Set the connection ID and enable pairing
	`BTM_BLE_ADVERT_STATE_CHANGED_EVT (BTM_BLE_ADVERT_OFF)`	
Read Button characteristic while touching buttons >	`GATT_ATTRIBUTE_REQUEST_EVT, GATTS_REQ_TYPE_READ >`	Returns button state
Disconnect >	`GATT_CONNECTION_STATUS_EVT >`	Clear the connection ID and re-start advertising
	`BTM_BLE_ADVERT_STATE_CHANGED_EVT (BTM_BLE_ADVERT_UNDIRECTED_HIGH)`	
Wait for timeout. >	`BTM_BLE_ADVERT_STATE_CHANGED_EVT (BTM_BLE_ADVERT_ UNDIRECTED _LOW)`	Stack switches to lower advertising rate to save power
Wait for timeout. >	`BTM_BLE_ADVERT_STATE_CHANGED_EVT (BTM_BLE_ADVERT_OFF)`	Stack stops advertising.

Application Creation

1. Instead of starting from the ch04a template, use the **ch04a_ex04_con** template to create the application.

   This template combines the functionality of exercises 4A.1 (LED Characteristic) and 4A.2 (advertising/connected indicator LED). You will add the Button Characteristic and button interrupt functionality.

2. Open the Bluetooth Configurator.

   a. Set the device name to <inits>_con. Make sure you hit enter or click in a field outside the Value. If not, the new value may not be saved.

   b. Add a second Characteristic to the BT101 Service by right clicking on the BT101 Service and selecting **Add Characteristic > Custom Characteristic**.

   c. Rename it to "Button".

   d. Configure Button to be a `uint8` that is initially 0 and only enable it for read since you can't "write" to the button's state.

3. Save your changes and close the configurator.

4. Open the *Makefile* and set `BT_DEVICE_ADDRESS?=random`.

5. Configure the button pin for an interrupt on both edges and create an interrupt callback. In the callback, save the current state of the button to the appropriate GATT array.

   a. **Hint** On the CYW920819EVB-02 kit, invert the value before storing it in the array since the button is active low and we want the button Characteristic value to be high when the button is pressed. This is not necessary on the CYBT-213043-MESH kit because the button is active high.

   b. **Hint** You can find the name of the array in the GATT Initial Value Arrays section of the GATT database code.

Testing

1. Program the application to the board.
2. Open the mobile CySmart app.
3. Connect to the device.
4. Open the GATT browser widget and then open the BT101 Service followed by the Button Characteristic.
5. Read the value while both pressing and not pressing the button to see the values.
6. Switch to the LED characteristic and verify that it still works to turn the LED ON/OFF.
7. Disconnect from the mobile CySmart app and start the PC CySmart app.
8. Start scanning and then connect to your device.

9. Click on **Discover all Attributes**.
10. Read the button value in CySmart by clicking on the Characteristic and then clicking the **Read Value** button.

    Continue reading as you press and release the button and verify that the value is correct.
11. Click **Disconnect**.

Questions

1. What function is called when there is a Stack event? Where is it registered?
2. What function is called when there is a GATT database event? Where is it registered?
3. Which GATT events are implemented? What other GATT events exist? (**Hint** Right click and select **Open Declaration** on one of the implemented events)
4. In the GATT `GATT_ATTRIBUTE_REQUEST_EVT`, what request types are implemented? What other request types exist?