Chapter 4A: The Essential BLE Peripheral Example

Time 2 Hours

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

4A.1 WICED BLE System Lifecycle

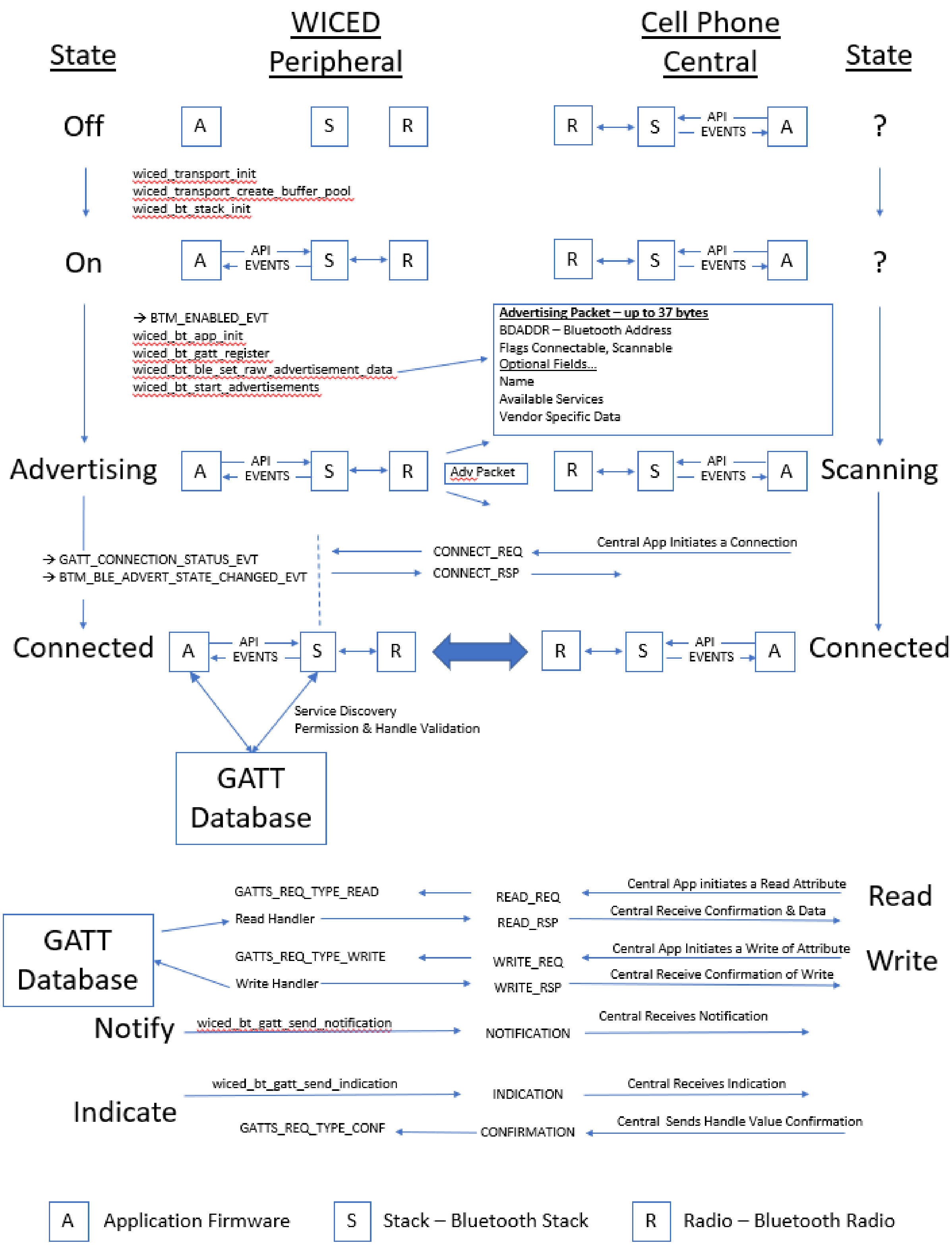
Basically, every book that I have ever read on Bluetooth or WiFi starts with the radio stack and works its way back (or up depending on your point of view) to the Application. You know the drill, 2.4 GHz Digital Spread Spectrum, Adaptive Frequency Hopping, blah blah blah. This approach surfaces a bewildering 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 BLE application that a cellphone 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
- Make a Connection
- Exchange Data (Read and Write)

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

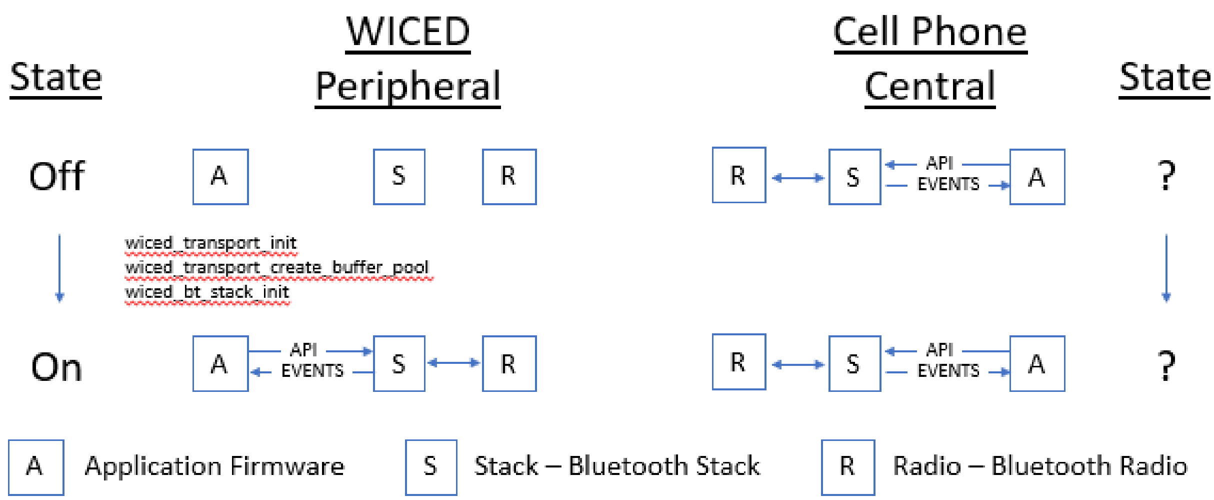


4A.1.1 Turning on the WICED Bluetooth Stack

In the beginning, you have a WICED device and a Cell Phone, and they are not connected, the WICED state is Off, so that's where we will start.

Like all great partnerships, every BLE 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.

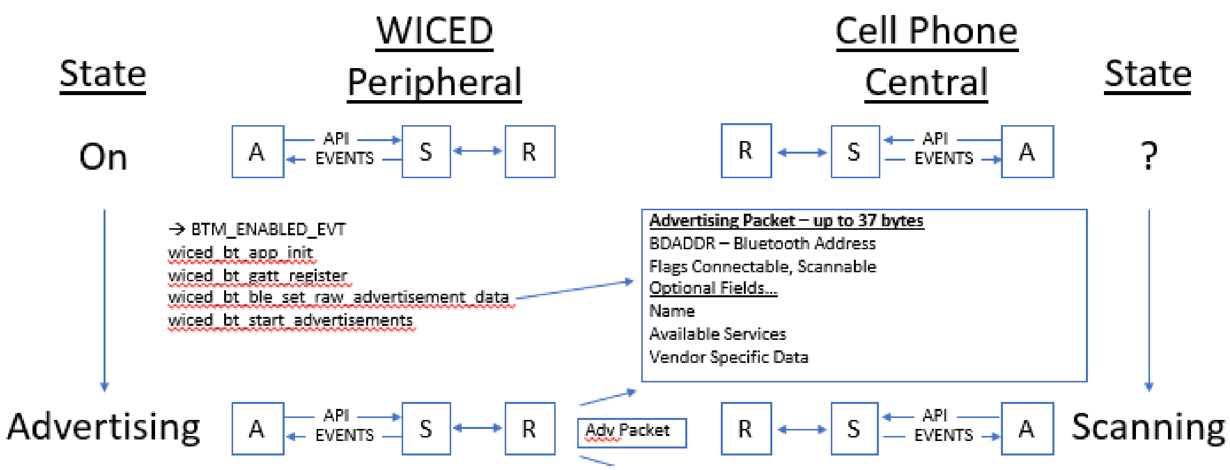
The first thing you do in your firmware is to turn on BLE. In WICED, that means that you initialize the Stack and provide it with a function that will be called when the Stack has events for you to process (this is often called the "callback" function for obvious reasons).



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 Address (BDADDR), 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.).

The Stack is responsible for broadcasting your advertising packets at a configurable interval into the open air. That means that all BLE 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 most important information sent in the advertising packet is called Flags. It tells the remote device how to make a connection by identifying the type of Bluetooth supported (BLE, 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.

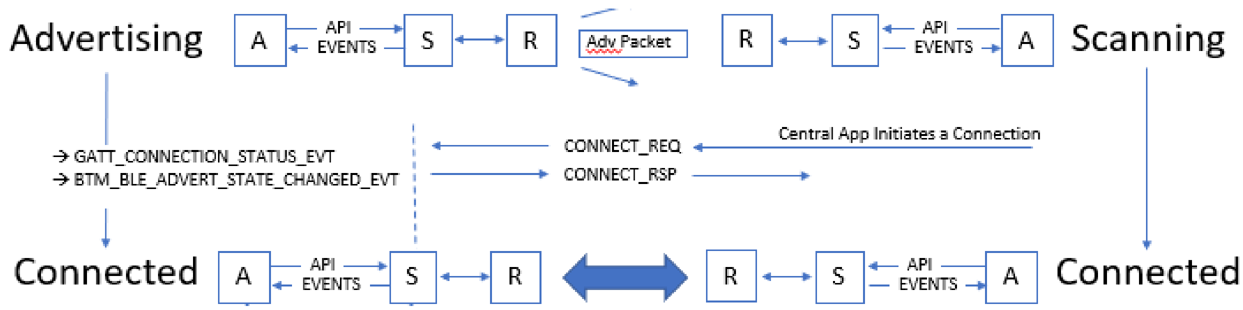
The format of the packet is quite simple. Each item you wish to advertise starts with a length byte, followed by the type (e.g. Flags or Name) and then the data, the size of which is determined by that length byte. The items are simple concatenated together, up to 31 bytes.

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 “connection_req” which will then go out the Central's radio and through the air to your WICED radio.

The WICED radio will feed the packet to the Stack which will respond AUTOMATICALLY back with a “connection_rsp” packet and stop advertising. You do not have to write code for the response to occur 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

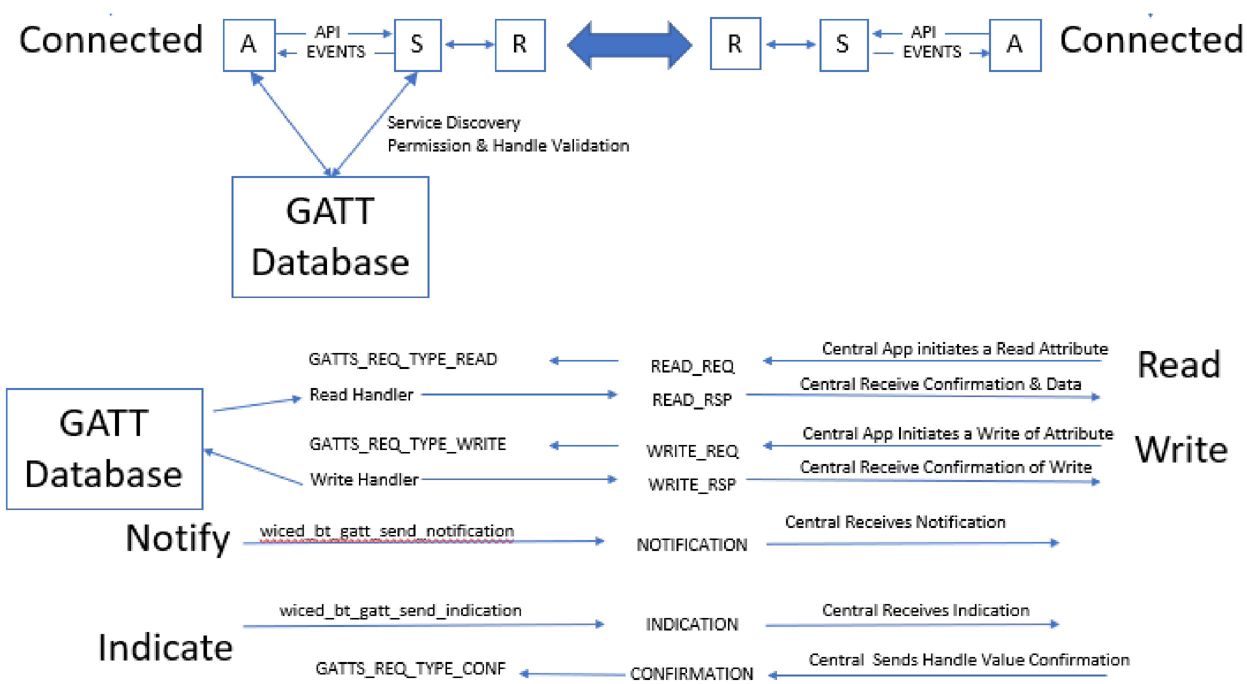
Now that you are connected you need to be able to exchange data. In the world of BLE this happens via the Attribute Protocol (ATT). The basic ATT protocol has 4 types of transactions: Read & Write which are initiated by the Central and Notify & Indicate which are initiated by the Peripheral.

ATT Protocol transactions are all keyed to a very simple database called the GATT database which typically resides on the Peripheral. Because the GATT Database is running on the Peripheral, that side is also commonly known as the GATT Server. Likewise, because the Central side is making requests of the database, it is commonly known as the GATT Client. 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 3.2. 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 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:

```
// BLE 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_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_32SOLICITATION_SRV_UUID = 0x1b, /* List of solicitated services - 32 bit UUIDs */
    BTM_BLE_ADVERT_TYPE_32SERVICE_DATA          = 0x1c, /* Service data - 32 bit UUID */
    BTM_BLE_ADVERT_TYPE_128SERVICE_DATA         = 0x1d, /* Service data - 128 bit UUID */
    BTM_BLE_ADVERT_TYPE_MANUFACTURER            = 0xFF  /* Manufacturer data */
};
```

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 structure 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:

```c
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;
```

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

```c
#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);
}
```

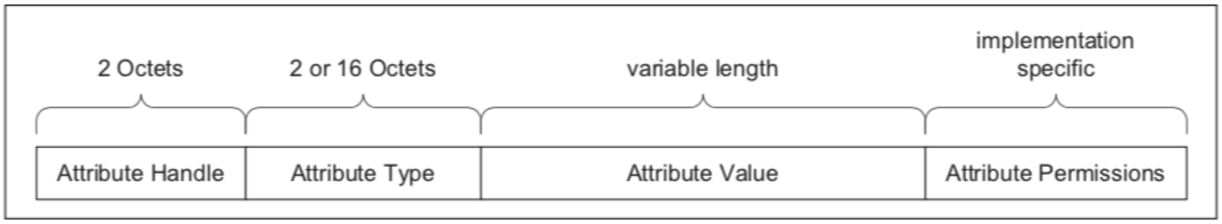
It turns out that the tool Bluetooth Designer helps you setup the Advertising Packet (including optionally adding the device name); more on this later.

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

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 or 16 bytes long
- 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 three levels: Profiles, Services and Characteristics. Note that Profiles, Services, and Characteristics are al just different types of Attributes.

A Profile is 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 BLE 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 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 UUID of the Service and the Attribute Permissions.

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

(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

(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»	Charac-teristic Properties	Character-istic Value Attribute Handle	Character-istic UUID	Read Only, No Authentication, No Authorization

(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 Properties are used by the Stack to enforce access to 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

(This figure is taken from the Bluetooth Specification)

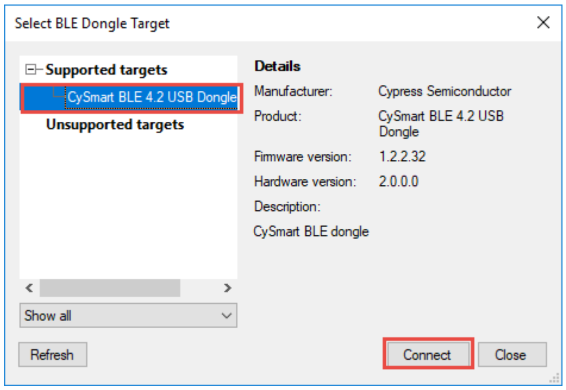
There are several other interesting Characteristic Attribute Types which will be discussed in the next chapter.

4A.4 CySmart

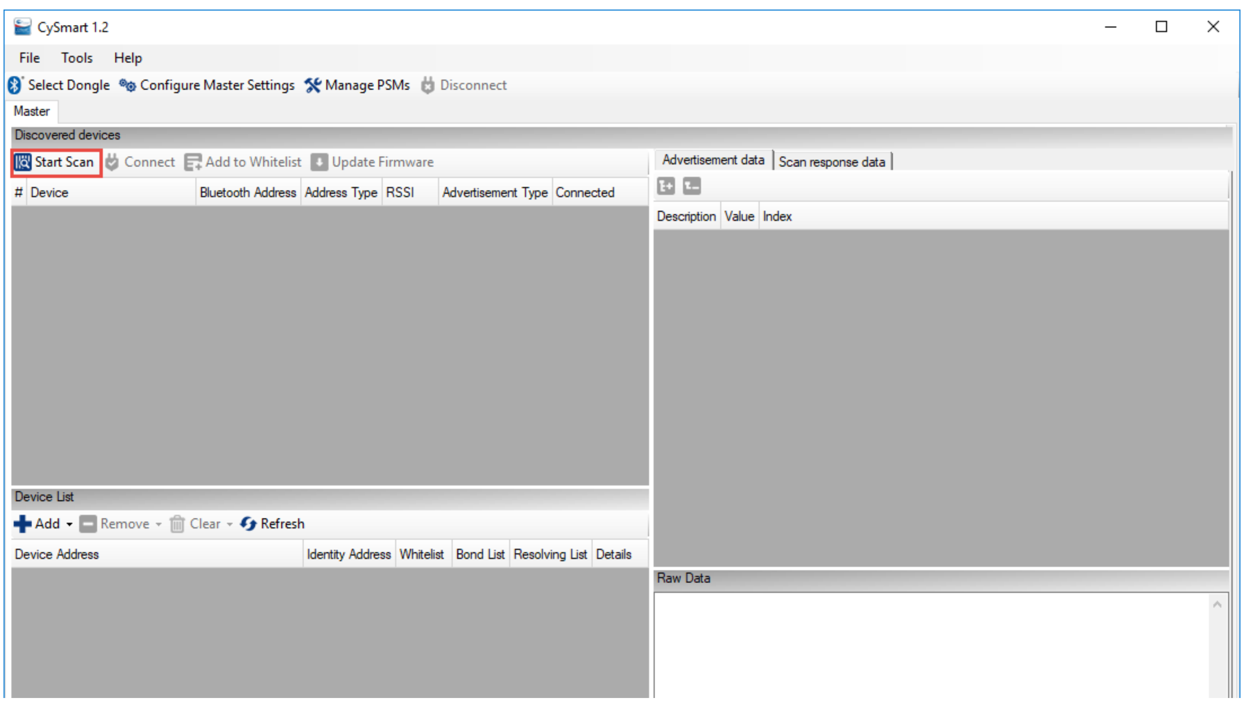
Cypress provides a PC and mobile device application (Android and iOS) called CySmart which can be used to scan, connect, and interact with services, characteristics, and attributes of BLE devices.

4A.4.1 CySmart PC Application

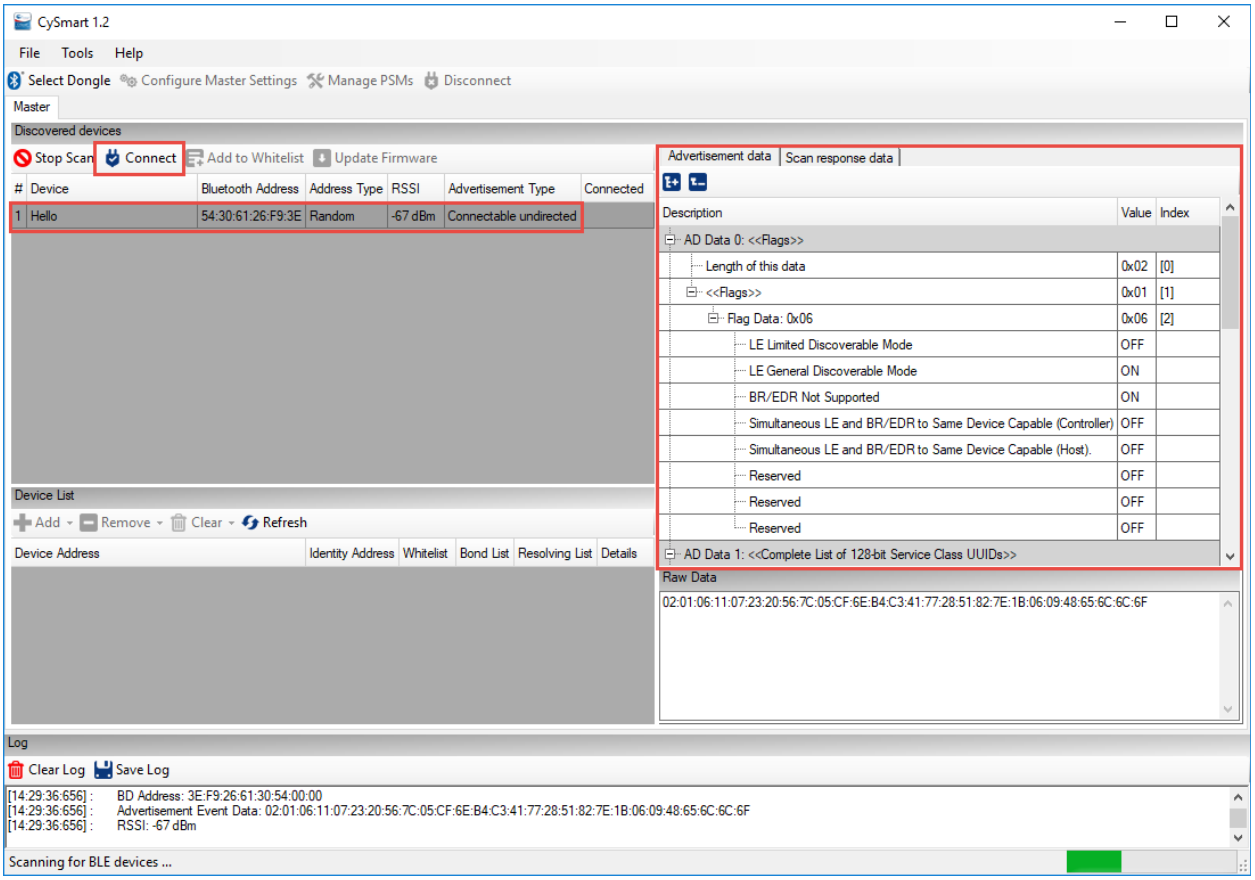
To use the CySmart PC Application, a CY5670 CySmart USB Dongle is required. When CySmart is started, it will search for supported targets and will display the results. Select the dongle that you want to use and click on "Connect".



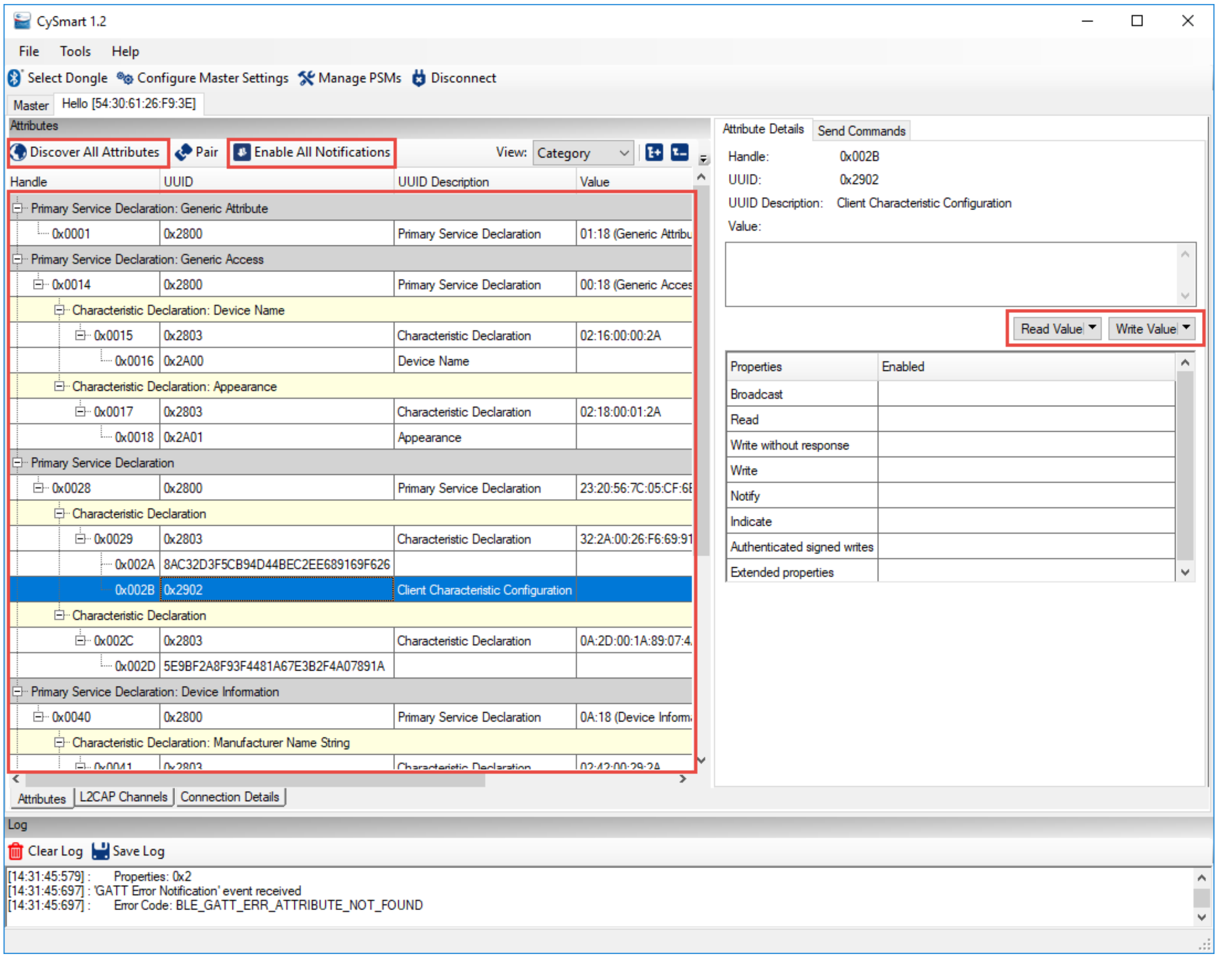
Once a dongle is selected, the main window will open as shown below. Click on "Start Scan" to search for advertising BLE devices.



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



Once 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.



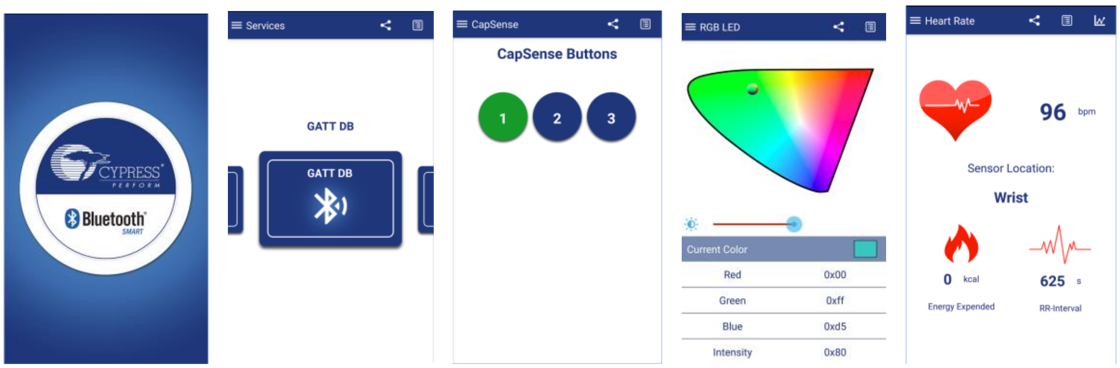
The complete User Guide for the CySmart PC application can be opened in the tool under *Help -> Help Topics*. It can also be found on the CySmart website at:

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.4.2 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 BLE device. It supports specialized screens for many of the BLE 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.



In the general settings, it is usually a good idea to check the box "Delete bond on disconnect" so that bonding information is not remembered for your device. This will prevent the phone from remembering a specific configuration for your kit which you may be changing as you create different projects for the same kit. Note that for the iPhone version, you must go to the phone's Bluetooth settings to delete the bonding information after you disconnect. You can also do that on Android after disconnecting if you forget to turn on the "Delete bond on disconnect" option.

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

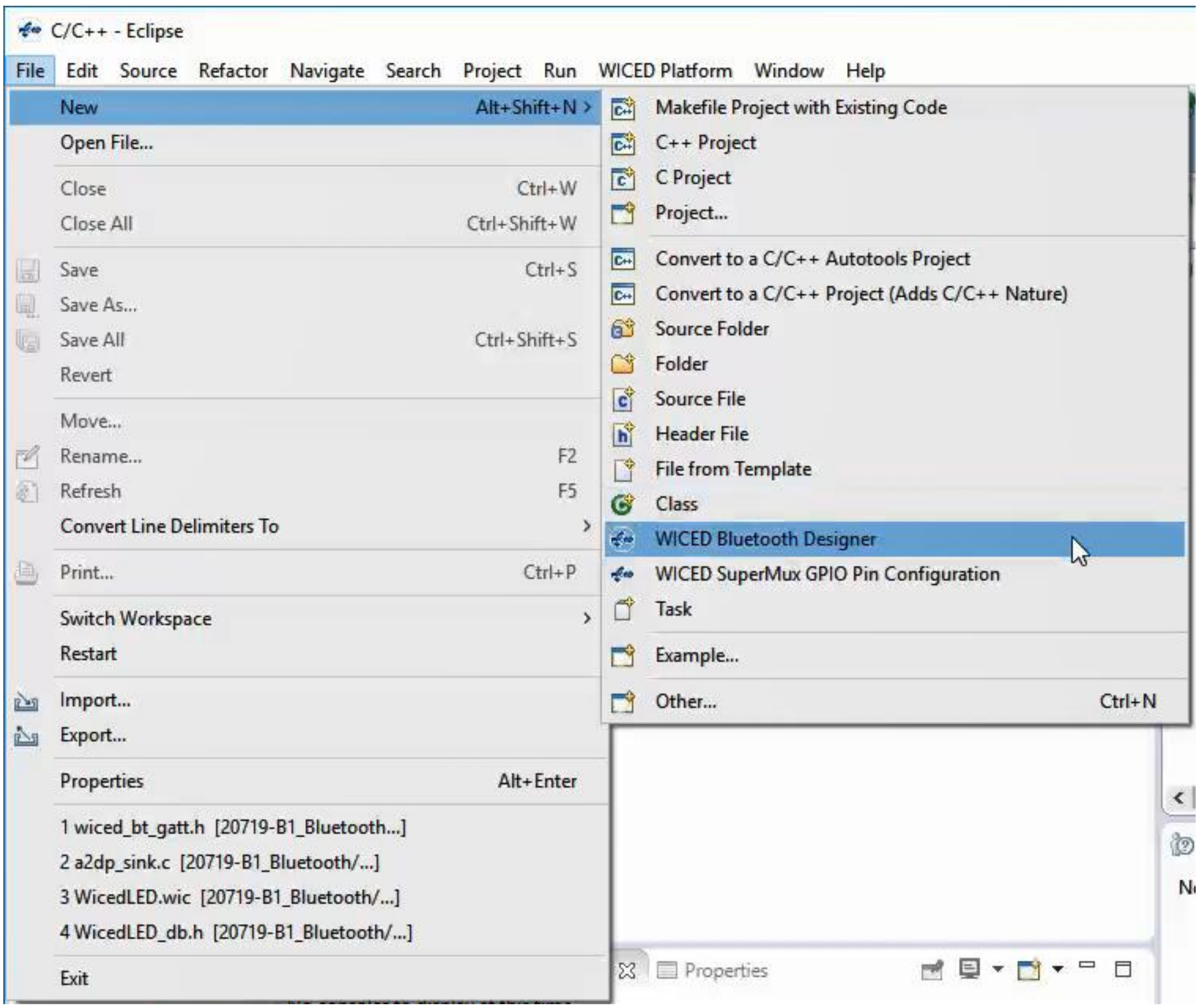
4A.5 WICED Bluetooth Designer

WICED Bluetooth Designer is a tool that will build a semi-customized template project for you for BLE or BR/EDR (aka Classic Bluetooth) or both. The tool copies in all the required files including the makefile, customizes them to your settings, and then creates a make target. The project is runnable with no changes (it doesn't do much, but it works).

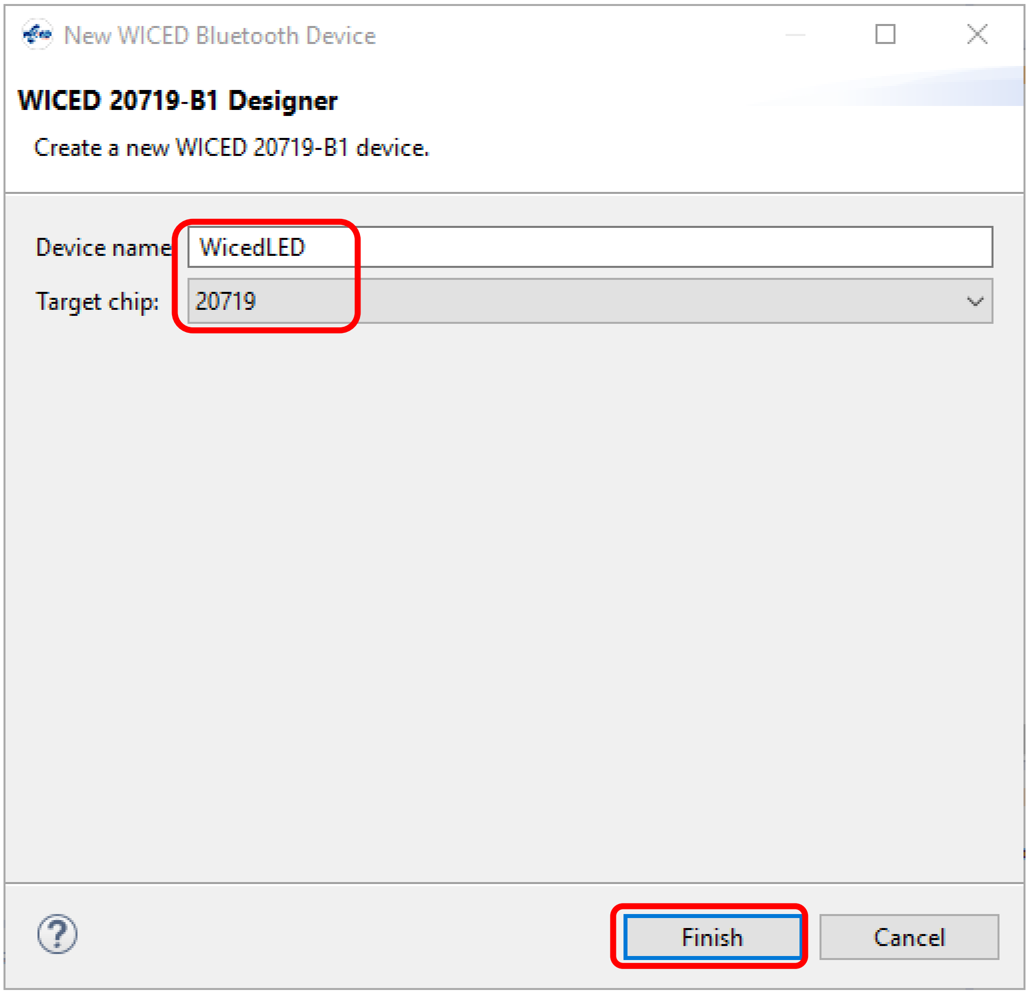
For this example, I am going to build a BLE project that has one custom service called the WicedLED Service with one writable characteristic called "LED". When the Central writes a 0 or 1 into that Characteristic, my application firmware will just write that value into the GPIO driving the LED.

4A.5.1 Running the Tool

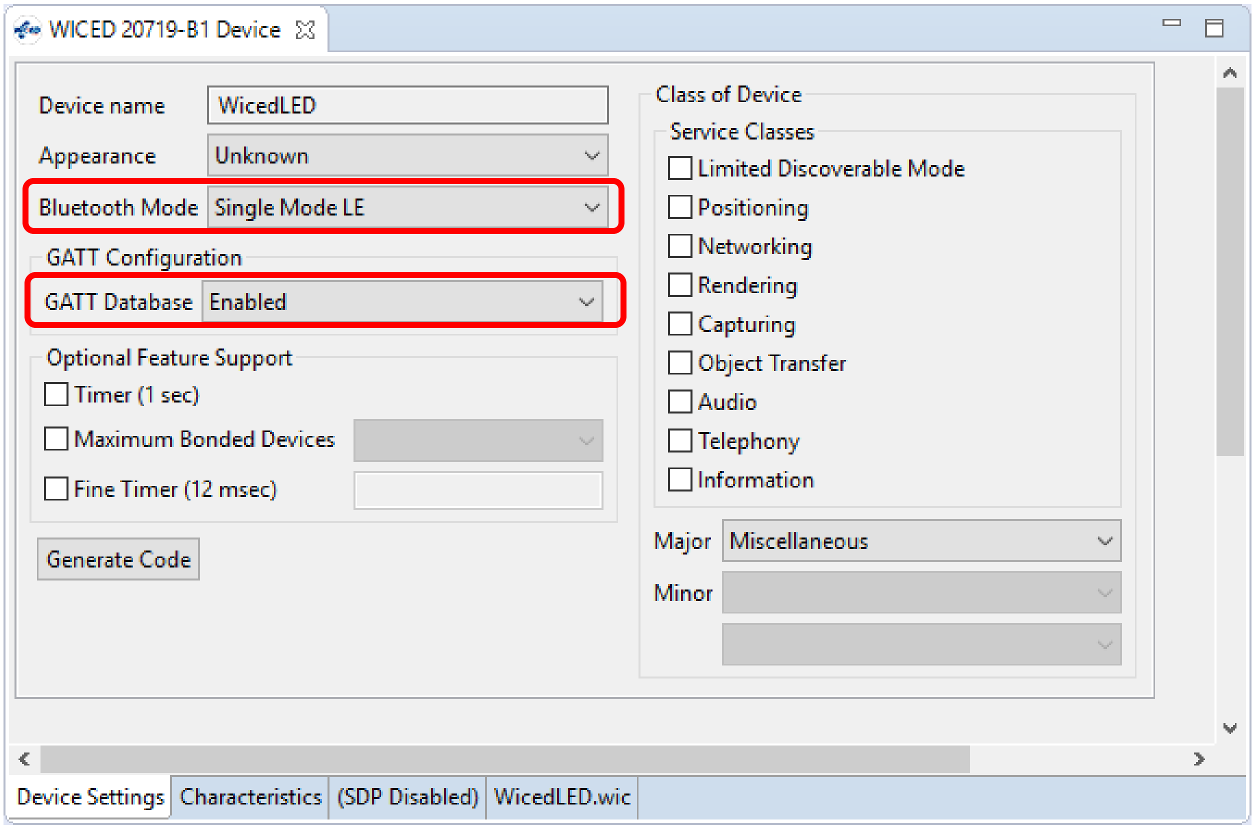
To run the tool, select the menu item *File->New->WICED Bluetooth Designer*.



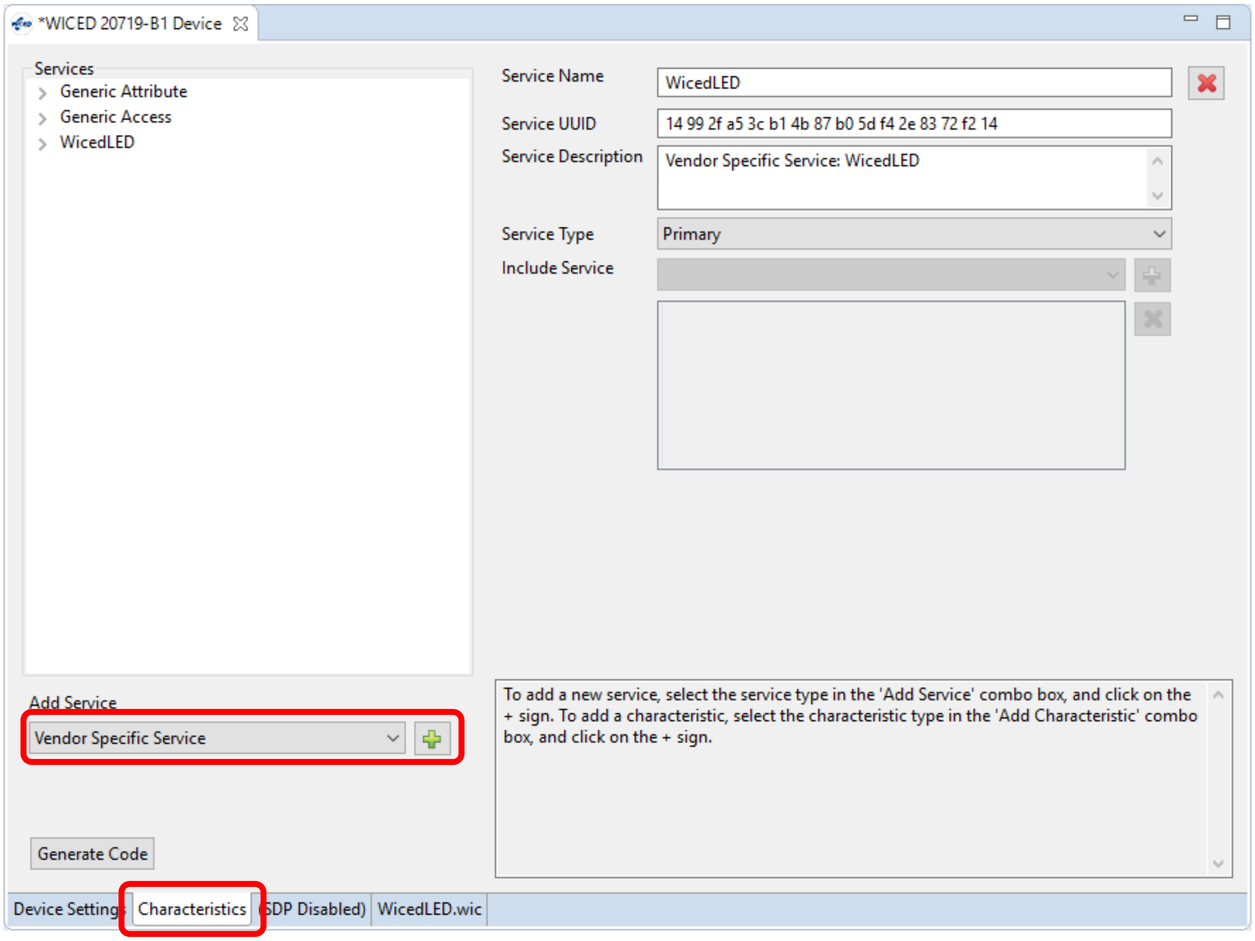
This will ask you to name your Project (also called the Device Name) and select a chip. In this case, I'll call the project *WicedLED* and I'll pick *20719* for the chip.



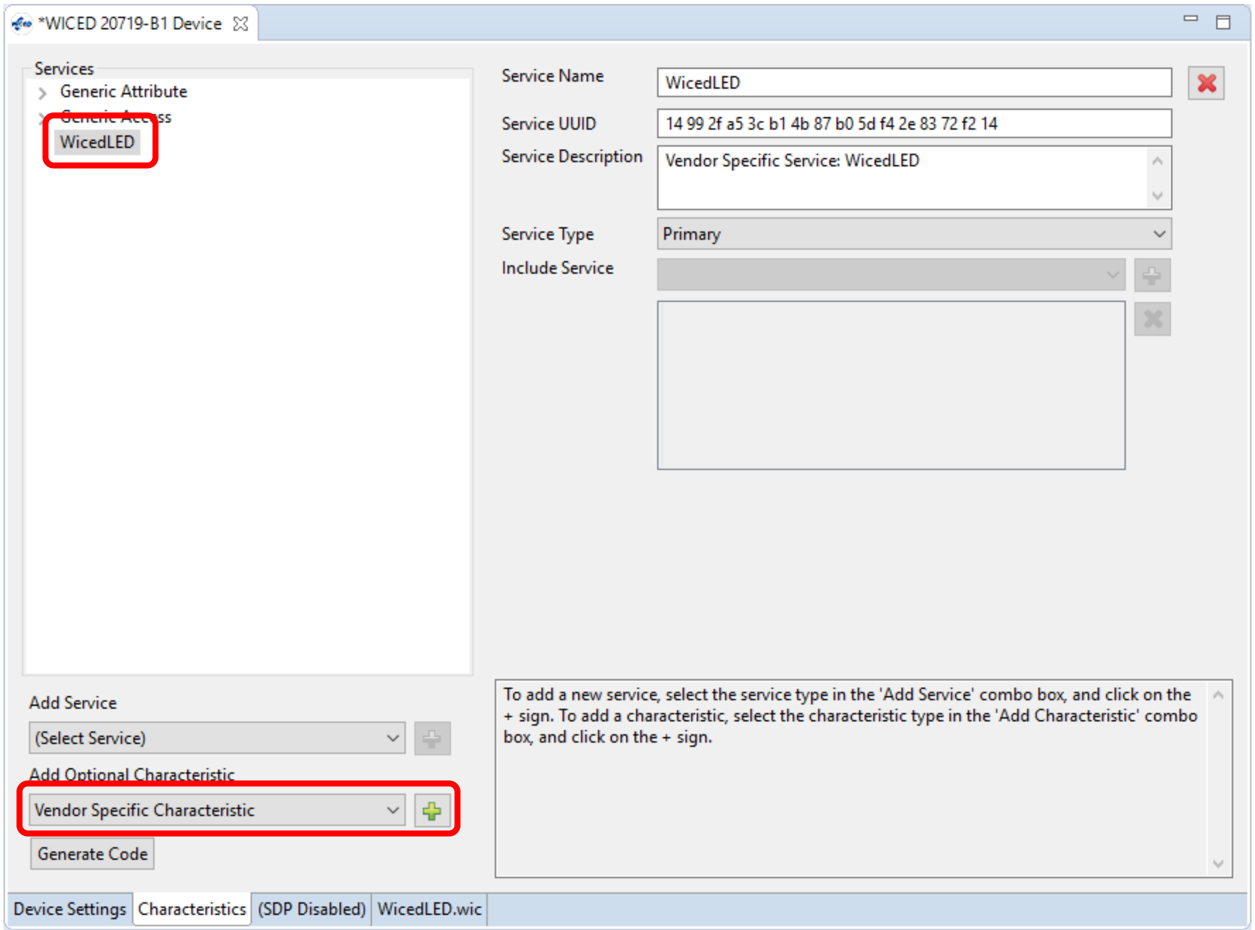
After you click Finish, you get a window allowing you to pick Dual Mode (aka BLE and classic), BR/EDR or Single Mode LE (aka BLE) along with some other options. I want the tool to help me build the GATT Database so I leave that enabled.



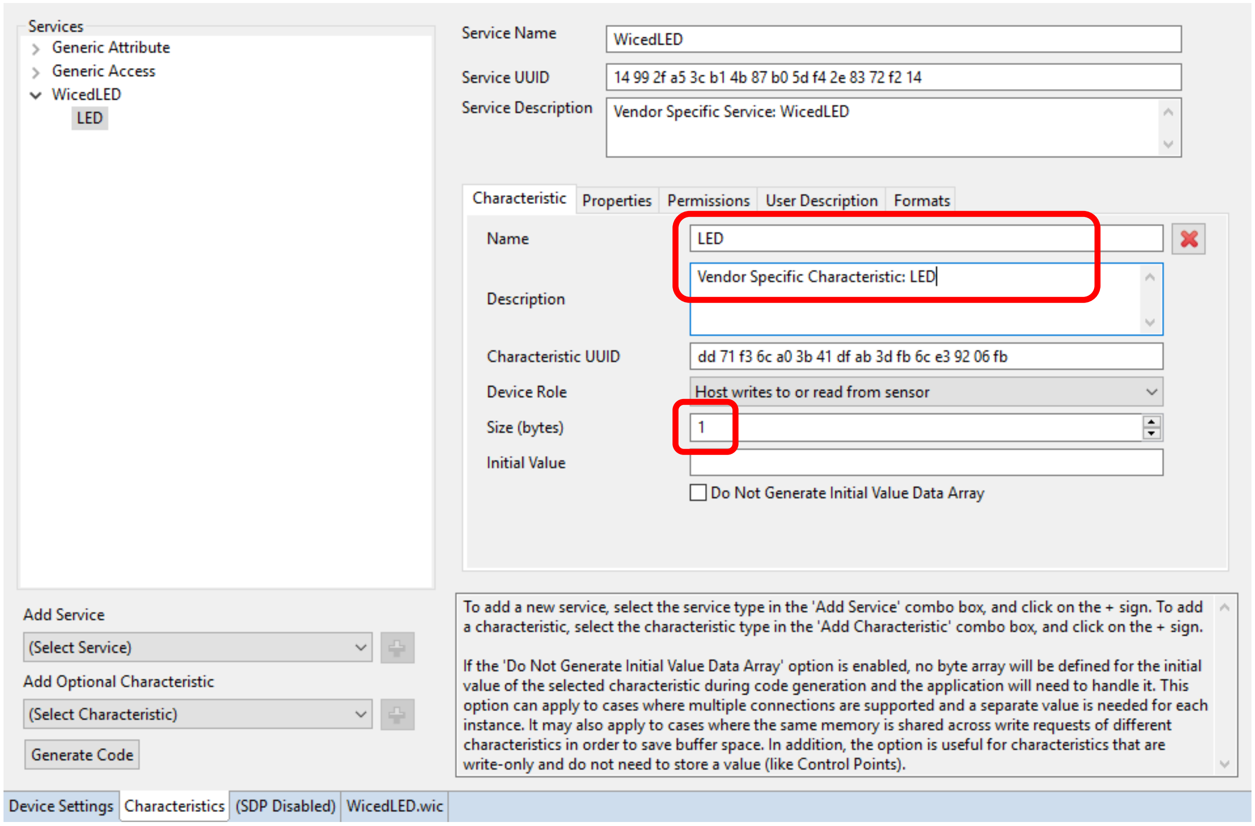
The next step is to setup a Service. To do this select the Characteristic tab. Then pick "Vendor Specific Service" and press the "+" button. After I do this I will see a new Service called "WicedLED" added to my Services. Notice that I could change the name in the "Service Name" box. I also let the tool choose a random UUID for this Service but I could specify my own UUID if I wanted.



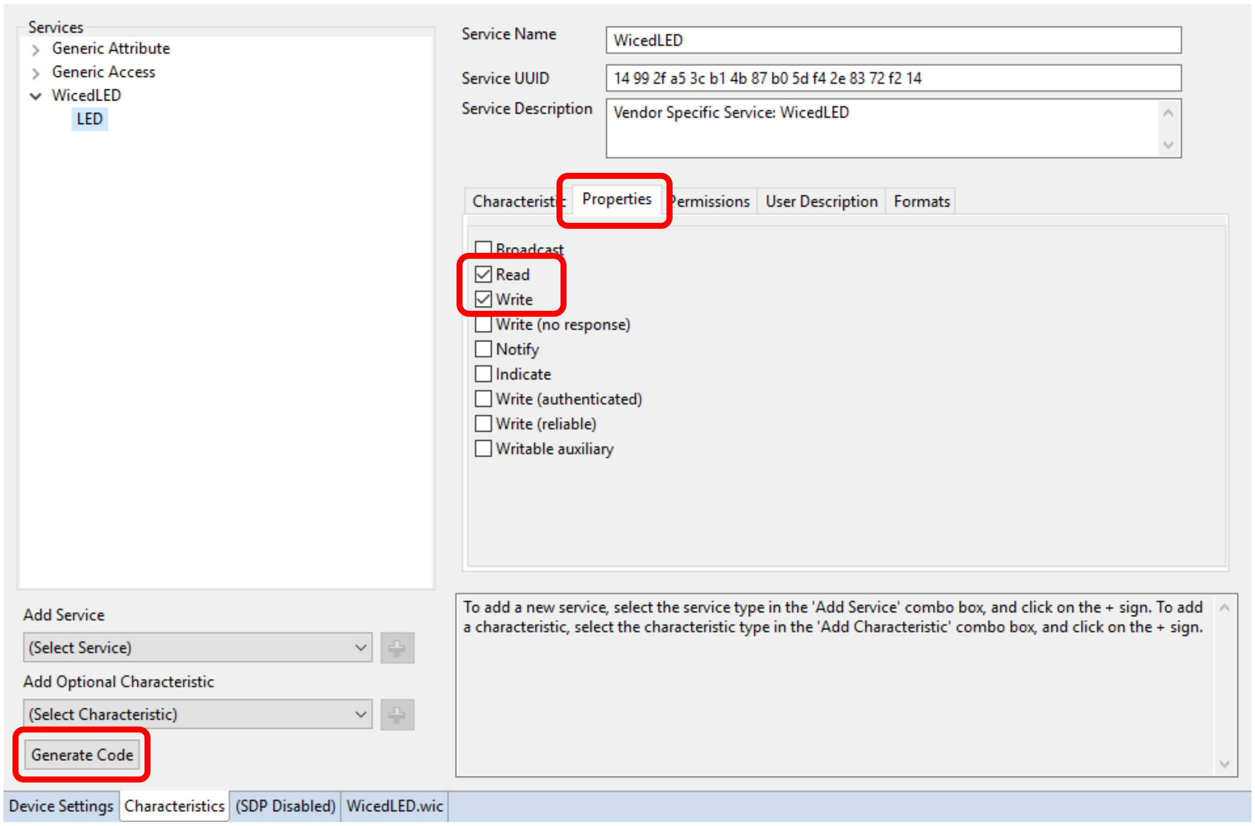
After the Service is configured I add one Characteristic by clicking on "WicedLED" in the Services window, then select "Vendor Specific Characteristic" and press the "+".



I then change the name of the Characteristic to "LED", specify that I want the Size to be 1 byte and leave the Initial Value blank which will result in a starting value of 0x00 (if you want a non-zero value in this field, you must put <u>exactly</u> 2 hex digits per byte with <u>exactly</u> 1 space between bytes for characteristics with more than 1 byte – make sure to check in the C source file for the proper initial value). Again, I'll keep the randomly assigned UUID for the Characteristic just like I did for the Service UUID.

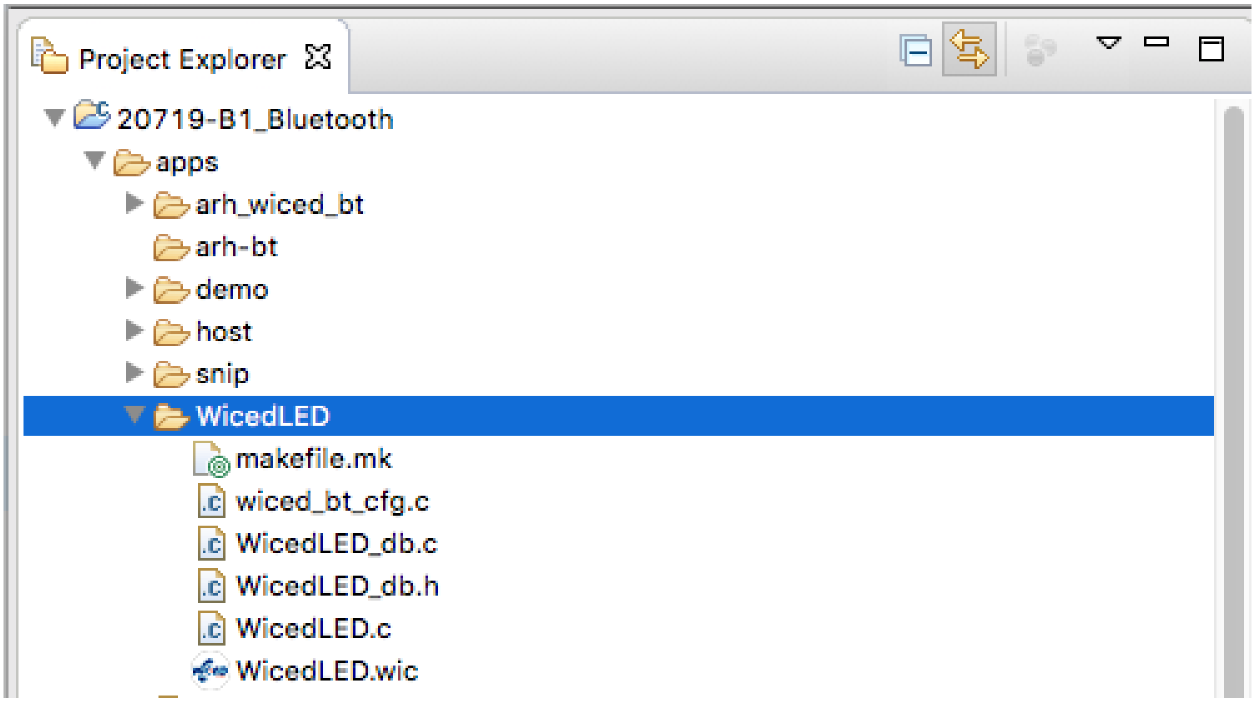


I want the client to be able to Read and Write this Characteristic, so click on the “Properties” tab and select “Read” and “Write”. When you make changes to the Properties, the tool makes the corresponding changes to the Permissions tab for you so you don't need to set them unless you need an unusual combination of Properties and Permissions.



After that press the “Generate Code” button. It is important to know that this will re-generate all the files after creating backup copies so any edits you have made to files will have to be re-done.

In a few seconds you will notice that you now have a new project in your apps tab in the Project Explorer.



4A.5.2 Editing the Firmware

To make this work I will make five changes to the generated project – four in WicedLED.c and one in the Make Target.

First, there are 3 header files that need to be included at the top of WicedLED.c. Namely:

```
#include "wiced_bt_stack.h"
#include "wiced_bt_app_common.h"
#include "wiced_hal_wdog.h"
```

Next, I want to use the PUART, so uncomment line 134 and comment out 137:

```
121 void application_start(void)
122 {
123     /* Initialize the transport configuration */
124     wiced_transport_init( &transport_cfg );
125
126     /* Initialize Transport Buffer Pool */
127     transport_pool = wiced_transport_create_buffer_pool ( TRANS_UART_BUFFER_SIZE, TRANS_UART_BUFFER_COUNT );
128
129 #if ((defined WICED_BT_TRACE_ENABLE) || (defined HCI_TRACE_OVER_TRANSPORT))
130     /* Set the Debug UART as WICED_ROUTE_DEBUG_NONE to get rid of prints */
131     //  wiced_set_debug_uart( WICED_ROUTE_DEBUG_NONE );
132
133     /* Set Debug UART as WICED_ROUTE_DEBUG_TO_PUART to see debug traces on Peripheral UART (PUART) */
134       wiced_set_debug_uart( WICED_ROUTE_DEBUG_TO_PUART );
135
136     /* Set the Debug UART as WICED_ROUTE_DEBUG_TO_WICED_UART to send debug strings over the WICED debug interf
137     //wiced_set_debug_uart( WICED_ROUTE_DEBUG_TO_WICED_UART );
138 #endif
139
```

Third, we don't want to pair to the device just yet so comment out the following line:

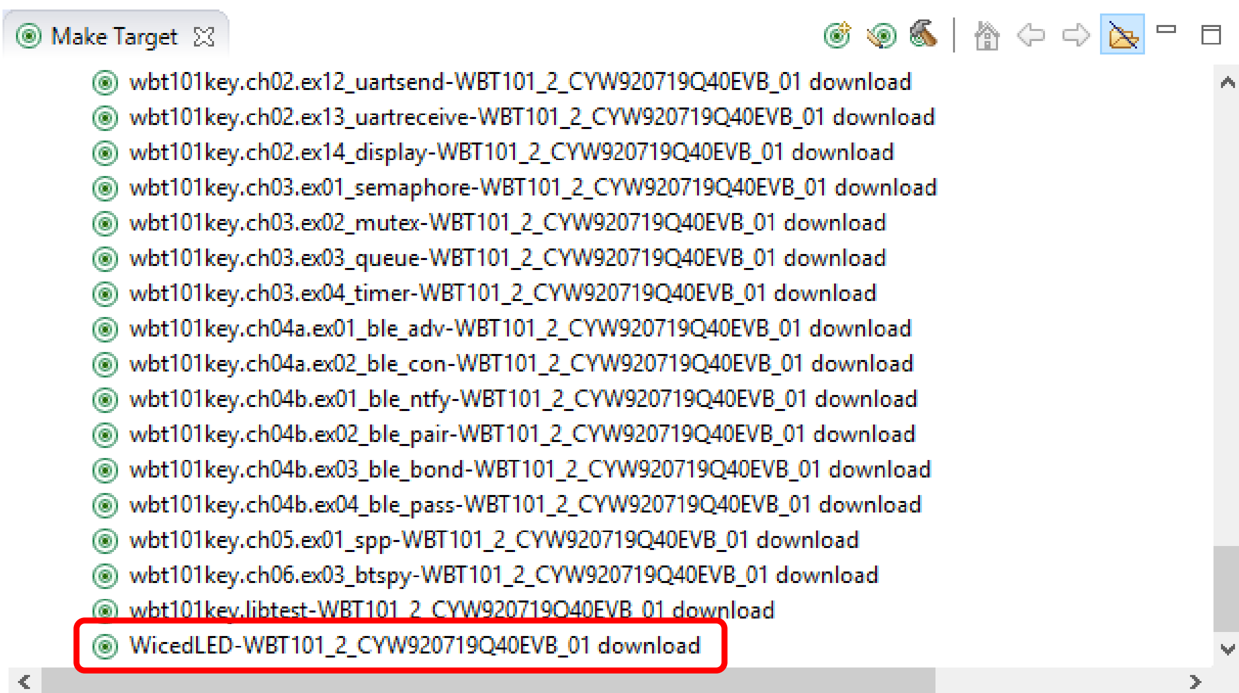
wiced_bt_set_pairable_mode(WICED_TRUE, 0);

Fourth, add two lines of code to write the LED and printout the result. We are going to use LED2 for this example. You will see this in a function called *wicedled_set_value* (do not confuse this with *wicedled_get_value*):

```
// TODO: 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_WICEDLED_LED_VALUE:
        /* Turn the LED on/off depending on the value written to the GATT database */
        WICED_BT_TRACE("Output = %d\n",wicedled_wicedled_led [0]);
        wiced_hal_gpio_set_pin_output(WICED_GPIO_PIN_LED_2, wicedled_wicedled_led [0]);
        break;
}
```

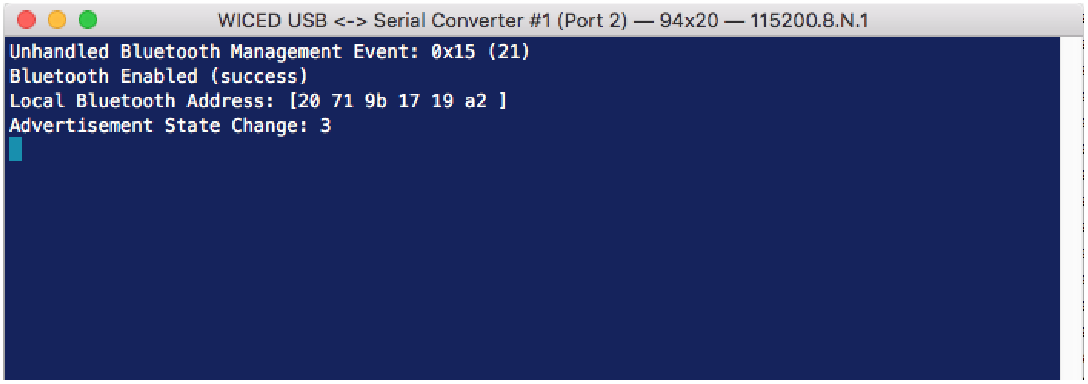
Notice how the GATT attribute (`wicedled_wicedled_led`) is updated for you by the stack when the write command is processed.

Finally, notice that Bluetooth Designer created a make target using the default platform. You are using a different platform (i.e. the kit + shield platform) so you need to edit the Make Target.



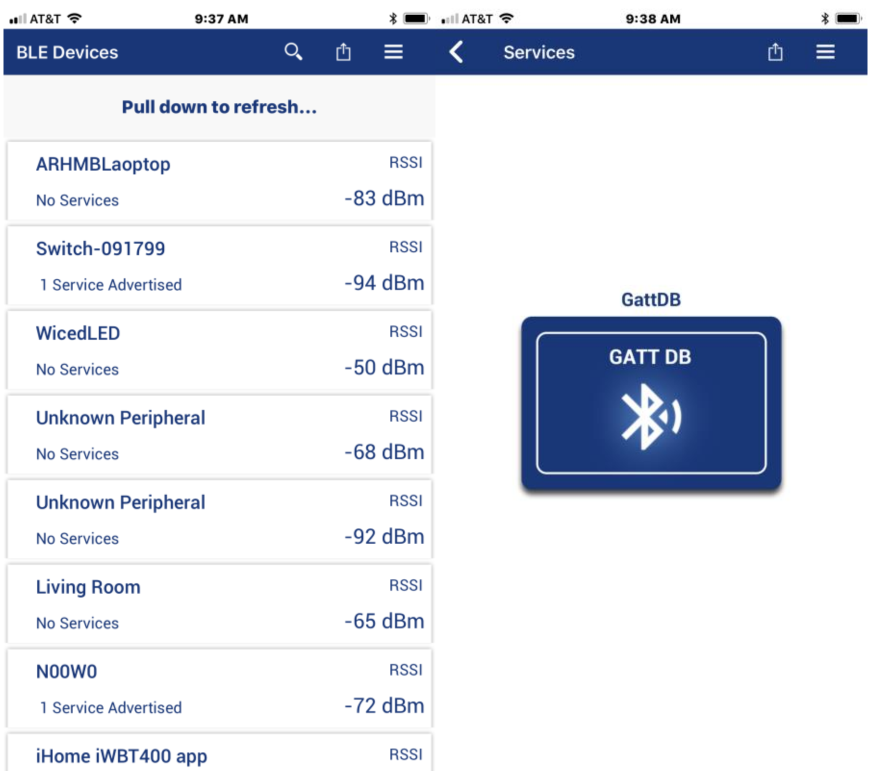
4A.5.3 Testing the Project

Start up a UART terminal and then run the make target. It will build and program the board. When the application firmware starts up you see some messages.

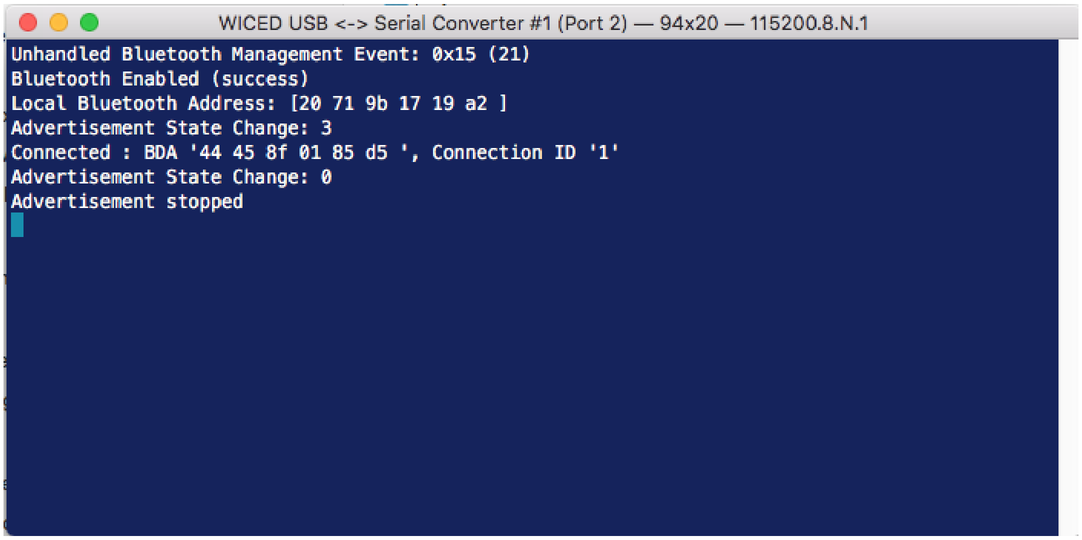


Run CySmart on your phone. When you see the “WicedLED” device, tap on it. CySmart will connect to the device and will show the GATT browser widget.

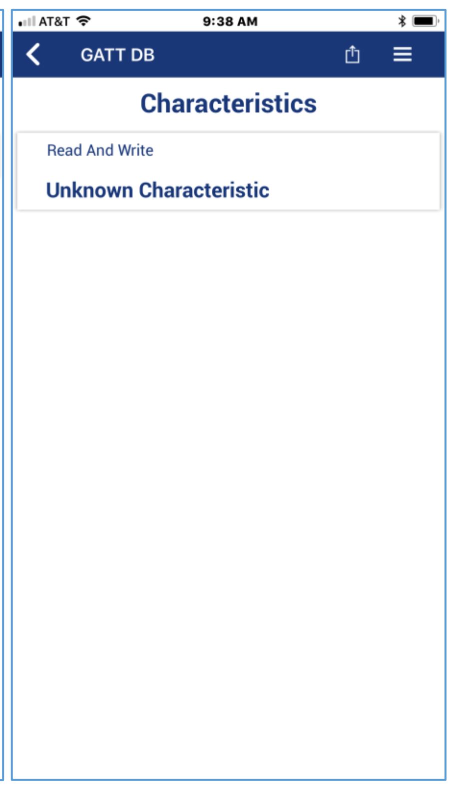
Note: If you are using the Android version of CySmart, before connecting go to the Settings and turn off the option "Initiate pairing after connection". If not, the connection will fail because we are not (yet) allowing pairing.



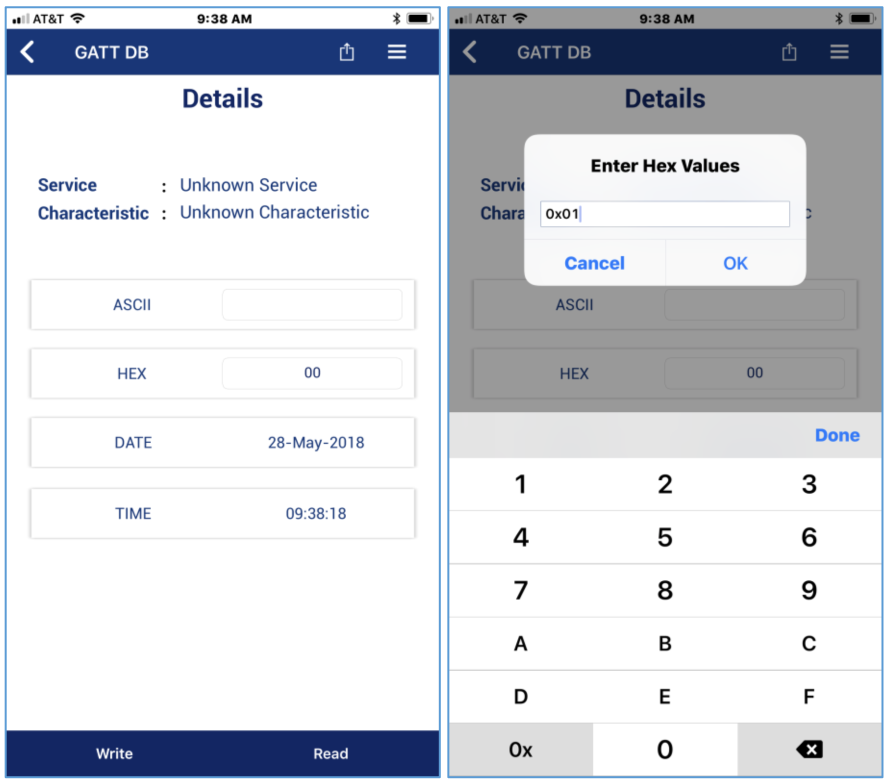
On the terminal window, you will see that there has been a connection and the advertising has stopped.



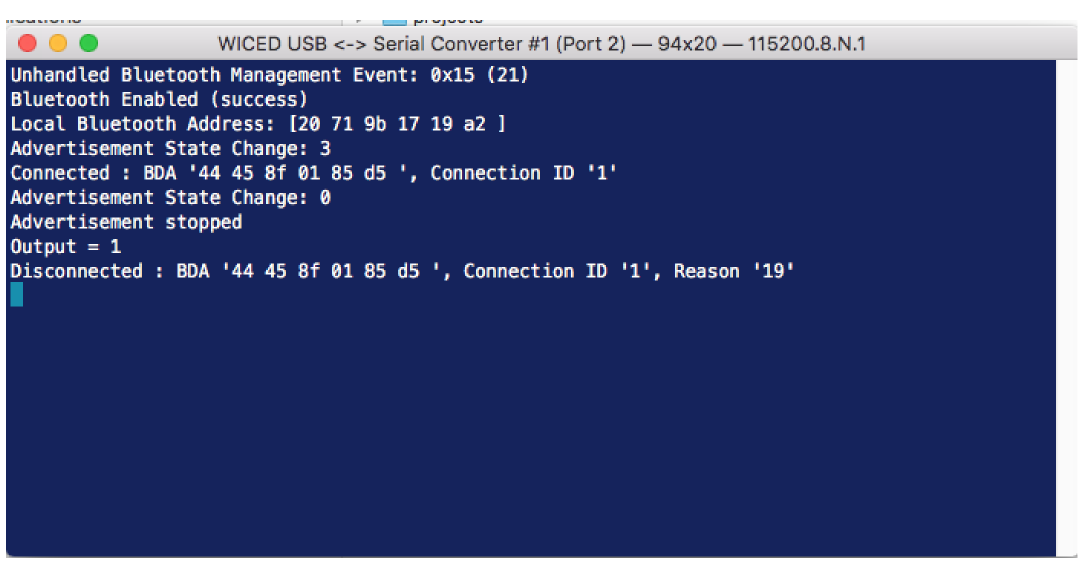
Back in CySmart, tap on the GATT DB widget to open the browser. You will see an Unknown Service (which I know is WicedLED). Tap on the Service and CySmart will tell you that there is an Unknown Characteristic (which I know is LED).



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.6 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. Bluetooth Designer will generate code to handle 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.6.1 Essential Bluetooth Management Events

Event	Description
BTM_ENABLED_EVT	When the Stack has everything going. The event data will tell if you 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.

WICED Bluetooth designer creates and registers a function called <appname>_management_callback to handle Management events.

4A.6.2 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.

WICED Bluetooth designer creates and registers a function called <appname>_event_handler to handle GATT events.

4A.6.3 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 WICED Bluetooth Designer with separate function calls.

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.

WICED Bluetooth designer creates a function called <appname>_connect_callback to handle these events. This function is called by the <appname>_event_handler function for connection events.

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.

WICED Bluetooth designer creates a function called <appname>_server_callback to handle these events. This function is called by the <appname>_event_handler function for attribute request events. In our application the wicedled_server_callback function calls wicedled_write_handler for GATTS_REQ_TYPE_WRITE events and that function calls wicedled_set_value, where we wrote the code to change the state of the LED (it does predictably the similar things for READ events).

4A.7 WICED Bluetooth Firmware Architecture

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

- Turn on the Stack
- Start Advertising
- Make a Connection
- Exchange Data (Read and Write)

The firmware created by WICED Bluetooth Designer mimics this flow.

4A.7.1 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 WICED API calls wiced_transport_init, wiced_transport_create_buffer_pools and wiced_bt_stack_init. One of the key arguments to wiced_bt_stack_init is a function pointer to the management callback.

WICED Bluetooth Designer creates a management callback function for you called <appname>_management_callback where <appname> is the name you gave to the project. 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 <appname>_management_callback function which then processes that event.

The <appname>_management_callback case for BTM_ENABLED_EVT event calls the function <appname>_app_init. It initializes the system including initialization of the GATT database and registering a callback function for GATT database events. The name of the GATT callback created by WICED Bluetooth Designer is <appname>_event_handler.

The <appname>_app_init function ends by calling the wiced_bt_start_advertising function.

4A.7.2 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 at the end of <appname>_app_init.

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 non-directed 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 <appname>_management_callback.

The <appname>_management_callback case for BTM_BLE_ADVERT_STATE_CHANGED_EVT looks at the event parameter to determine if it is a start or ending of advertising. In the Bluetooth Designer generated 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.7.3 Making a Connection

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 <appname>_event_handler function.

The <appname>_event_handler calls the function <appname>_connect_callback which uses the event parameter to determine if it is a connection or a disconnection. It then prints a message.

The Stack then stops the advertising and calls <appname>_mangement_callback with a management event BTM_BLE_ADVERT_STATE_CHANGED_EVT.

The <appname>_management_callback determines that it is a stop of advertising, and then calls <appname>_advertisement_stopped, which just prints out a message. You could add your own code here to, for instance, turn off an LED or restart advertisements.

4A.7.4 Exchange Data – Read (from the Central)

When the Central 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 <appname>_event_handler.

The <appname>_event_handler determines the event is GATT_ATTRIBUTE_REQUEST_EVT and calls the function <appname>_server_callback.

The <appname>_server_callback function looks at the event parameter and determines that it is a GATTS_REQ_TYPE_READ, then calls the function <appname>_read_handler.

The <appname>_read_handler calls the GATT Database API <appname>_get_value to find the current value of the Characteristic.

The <appname>_get_value 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, it returns a WICED_BT_GATT_SUCESS, which is then returned by <appname>_read_handler to <appname>_server_callback.

If something bad has happened in the <appname>_get_value function (like the requested Handle doesn't exist) it returns the appropriate error code i.e. WICED_BT_GATT_INVALID_HANDLE.

The <appname>_server_callback returns the status code generated by the <appname>_get_value function to the Stack. The Stack then either sends the error code, or it sends the data back to the Central.

To summarize, the function call hierarchy for a read is:

- <appname>_event_handler
    - <appname>_server_callback
        - <appname>_read_handler
            - <appname>_get_value

4A.7.5 Exchange Data – Write (from the Central)

When the Central 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 <appname>_event_handler.

The <appname>_event_handler determines the event is GATT_ATTRIBUTE_REQUEST_EVT and calls the function <appname>_server_callback.

The <appname>_ server_callback looks at the event parameter and determines that it is a GATTS_REQ_TYPE_WRITE, then calls the function <appname>_write_handler

The <appname>_write_handler calls the GATT Database API <appname>_set_value to update the current value of the Characteristic.

The <appname>_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, it returns a WICED_BT_GATT_SUCESS, which is then returned by the <appname>_read_handler to the <appname>_server_callback.

If something bad has happened in the <appname>_set_value function (like the requested Handle doesn't exist) it returns the appropriate error code i.e. WICED_BT_GATT_INVALID_HANDLE.

The <appname>_server_callback returns status code generated by the <appname>_set_value function to the Stack. The Stack then either send the error code, or a write response.

The function call hierarchy for a write is:

- <appname>_event_handler
    - <appname>_server_callback
        - <appname>_write_handler
            - <appname>_set_value

4A.8 WICED GATT Database Implementation

WICED Bluetooth Designer automatically creates a template GATT Database implementation to serve as a starting point. The database is split between <appname>_db.c, <appname>_db.h, and <appname>.c.

The implementation is generic and will work for most situations, however you can make changes to handle custom situations. When you start the Stack by calling wiced_bt_stack_init one of the parameters is a pointer to the GATT DB, meaning that the Stack will directly access your GATT DB for some purposes.

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

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

The WICED Implementation of the GATT Database is simple generic “C” (obviously) and is composed logically of four parts:

- An Array, named gatt_database, of uint8_t bytes that holds the Handles, Types and Permissions.
  - In <appname>_db.c
- An Array of Structs which holds Handles, a Maximum and Current Length and a Pointer to the actual Value.
  - In <appname>_db.h and <appname>.c
- The Values as arrays of unint8_t bytes.
  - In <appname>.c
- Functions that serve as the API
  - In <appname>.c

4A.8.1 gatt_database[]

The gatt_database is just an array of bytes with special meaning. To create the bytes representing an Attribute we have created 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 actual Attribute Handle, a 16-bit number. WICED Bluetooth Designer will automatically create Handles for you that will end up in the <appname>_db.h file. For example:

```
// ***** Primary Service 'Generic Attribute'
#define HDLS_GENERIC_ATTRIBUTE                  0x0001

// ***** Primary Service 'Generic Access'
#define HDLS_GENERIC_ACCESS                     0x0014
```

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

```
#define __UUID_LED                    0xbb, 0x4c, 0x2e, 0x3b, 0x64, 0xa5, 0x40, 0x78, 0x88, 0x5f, 0x57, 0x57, 0xb6, 0xb4, 0xfa, 0x3a
```

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 WICED Bluetooth Designer creates for the Attributes for Characteristics which will be in the form of #define HDLC_ for example:

```
// ----- Characteristic 'Appearance'
#define HDLC_GENERIC_ACCESS_APPEARANCE          0x0017
#define HDLC_GENERIC_ACCESS_APPEARANCE_VALUE    0x0018
```

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

The UUID is a 16 or 128 bit UUID in an array of bytes. WICED BT Designer will create #defines for the UUIDs in the file <appname>_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.

```
/*****************************************************************************
 *  GATT Database Defintions
 *****************************************************************************/
/*  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.8.2 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 value.

```
// 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;
```

WICED Bluetooth Designer will create this array for you automatically in <appname>.c.

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

/* GATT attribute lookup table                                */
/* (attributes externally referenced by GATT server database) */
gatt_db_lookup_table testwbt_gatt_db_ext_attr_tbl[] =
{
    /* { attribute handle,                   maxlen,                                     curlen,                                     attribute data } */
    {HDLC_GENERIC_ACCESS_DEVICE_NAME_VALUE, 7,                                          7,                                          testwbt_generic_access_device_name},
    {HDLC_GENERIC_ACCESS_APPEARANCE_VALUE,  2,                                          2,                                          testwbt_generic_access_appearance},
    {HDLC_LED_LED1_VALUE,                   sizeof(testwbt_led_led1),                   sizeof(testwbt_led_led1),                   testwbt_led_led1},
    {HDLD_LED_LED1_USER_DESCRIPTION,        sizeof(testwbt_led_led1_user_description), sizeof(testwbt_led_led1_user_description), testwbt_led_led1_user_description},
};
```

API functions <appname>_get_value and <appname>_set_value created by WICED Bluetooth Designer to help you search through this array to find the pointer to the value.

4A.8.3 uint8_t Arrays for the Values

WICED Bluetooth Designer will generate one array of uint8_t to hold the value of writable/readable Attributes. You will find these values in a section of the code in <appname>.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 wicedled_generic_access_device_name[] =
{'W','i','c','e','d','L','E','D'};
uint8_t wicedled_generic_access_appearance[]  = {0x00,0x00};
uint8_t wicedled_wicedled_led1[]              = {0x00};
```

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

4A.8.4 The Application Programming Interface

There are two functions which make up the interface to the GATT Database, <appname>_get_value and <appname>_set_value. Here are the function prototypes from the "WicedLED" application:

wiced_bt_gatt_status_t wicedled_get_value(uint16_t attr_handle, uint16_t conn_id, uint8_t *p_val, uint16_t max_len, uint16_t *p_len)

wiced_bt_gatt_status_t wicedled_set_value(uint16_t attr_handle, uint16_t conn_id, uint8_t *p_val, uint16_t len)

These functions have the following input parameters:

- uint16_t attribute_handle – Recall that all transactions in BLE are based on the handle. The Client writes data based on the handle and you respond to reads based on the handle.
- uint16_t conn_id – The device supports multiple connections, but BT designer does not so this parameter is unused.
- uint8_t *p_val – A pointer to the data. For a write, this is a pointer to the data that is copied into the database, for a read this is a pointer to a location where data that will be sent to the Client is copied from the database.
- (read) uint16_t max_len – When you get a read, you should not return more than max_len bytes. The generated code automatically does both the read and write correctly.
- (read) uint8_t *p_len – When a read occurs you need to tell the calling function how many bytes you are returning. For example, *p_len = 23; // returning 23 bytes.
- (write) uint16_t len – For a write, you will be told how many bytes got written to you.

Both the automatically generated functions loop through the GATT Database and look for an attribute handle that matches the input parameter. It then memcpy's 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.9 Exercises

Exercise - 4A.1 Create a BLE Advertiser

Introduction

In this exercise, you will create a project 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 1-byte of manufacturer specific data. Each time a button is pressed on the shield, 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

Project Creation

1. Run WICED Bluetooth Designer and set up a project called ex01_ble_adv.
    a. Select *Generic Tag* for the *Appearance*.
    b. Disable the GATT database.
    c. Generate the code.
2. Move the project to the *wbt101\ch04a* folder.
3. Change the Make Target to have the correct path to the project and change the platform name to include the shield/kit combo.
4. Find the location where the name is specified in *wiced_bt_cfg.c* and change it to *<inits>_adv* where *<inits>* is your initials. This is necessary so that you will be able to tell which device yours is from those that will be advertising.
    a. Hint: be sure to leave the trailing '\0'.

5. Open the main C file for the project and familiarize yourself with its structure.
6. Add includes for the following 3 header files:

```
#include "wiced_bt_stack.h"
#include "wiced_bt_app_common.h"
#include "wiced_hal_wdog.h"
```

    a. Hint: Don't forget to leave the trailing '\0' null termination at the end.
7. 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.
    a. Hint: Right click on the existing advertisement type and select *Open Declaration* to see all the available choices.
8. Add a global variable of type *uint8_t* called *manuf_data*. Initialize it to a value of 0.
9. Locate the function that sets up the advertisement data and add a new element to send the *manuf_data* value.
    a. Hint: The advertisement type for this element should be *BTM_BLE_ADVERT_TYPE_MANUFACTURER*.
    b. Hint: don't forget to increase the number of elements in the advertising data array.
10. Configure Button1 for a falling edge interrupt. Add a button interrupt callback that does the following:
    a. Clear the pin interrupt
    b. Increment *manuf_data*
    c. Update the advertisement packet data array
        i. Hint: you can just call the function that Bluetooth Designer created.
    d. Re-start advertisements
11. In the main C file change the debug UART to WICED_ROUTE_DEBUG_TO_PUART so that debug messages will show up on a terminal window. We will discuss using the HCI UART in the debugging chapter.

Testing

1. Program the project to the board and use the PC version of CySmart to examine the advertisement packets. Press the button and then stop / re-start the scan to see that the value has incremented.
    a. Hint: you must have a CY5577 CySmart BLE USB dongle connected to your PC to run CySmart.

Questions

1. How many bytes is the advertisement packet?

Exercise - 4A.2 Connect using BLE

Introduction

In this exercise, you will create a project that will have a custom CapSense Service containing a CapSense Button characteristic with data for 4 buttons. You will monitor the CapSense buttons on the shield board and update their states in the GATT database so that a client can read the values.

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, start CapSense thread.
	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 CapSense 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.

Project Creation

1. Run WICED Bluetooth Designer and set up a project called *ex02_ble_con.*
    a. Select *Unknown* for the *Appearance*.
    b. Enable the GATT database.
    c. Go to the Characteristics tab and add a Vendor Specific Service.
        i. Change the Service Name and Service Description to *CapSense*.
        ii. Change the UUID to (Hex): 31 01 9B 5F 80 00 00 80 00 10 00 00 B5 CA 03 00
    d. Add a Vendor Specific Characteristic to the CapSense Service
        i. Change the Name and Description to *Buttons*.
        ii. Change the UUID is (Hex): 31 01 9B 5F 80 00 00 80 00 10 00 00 A3 CA 03 00
            1. Hint: the UUIDs are NOT the same – the 4th byte from the end is different.
        iii. The size of the characteristic is 3 bytes.
        iv. Set the initial value to *04 00 00*.
            1. Hint: There must be exactly 1 space between the values (i.e. one space between 04 and the first 00, and one space between the first 00 and the second 00).
        v. Set the Properties to *Read*.
    e. Generate the code.
2. Move the project to the *wbt101\ch04a* folder.
3. Change the Make Target to have the correct path to the project and change the platform name to include the shield/kit combo.
4. Find the location where the name is specified in *wiced_bt_cfg.c* and change it to *<inits>_con* where *<inits>* is your initials. This is necessary so that you will be able to tell which device yours is from those that will be advertising.
    a. Hint: Don't forget to leave the trailing '\0' null termination at the end.
5. Find the location where the name is specified in the GATT database in ex02_ble_con.c and change it to *<inits>_con* where *<inits>* is your initials.
    a. Hint: Search for *device_name*.
    b. Hint: In this case, there is no trailing '\0'.
6. Open the main C file for the project and familiarize yourself with its structure.
7. Add includes for the following header files:

```
#include "wiced_bt_stack.h"
#include "wiced_bt_app_common.h"
#include "wiced_hal_wdog.h"
#include "wiced_rtos.h"
#include "wiced_hal_i2c.h"
```

8. Update the advertisement packet so that it sends the flags, name, and the UUID of the CapSense service.
    a. Hint: Figure out the length of the advertisement packet. If it is greater than 31 bytes it will not work. You may need to either change the device name or send a short name instead of the complete name in the advertisement packet.

b. Hint: The advertisement type for a complete service name is *BTM_BLE_ADVERT_TYPE_128SERVICE_DATA*.
c. Hint: There is a macro called "LEN_UUID_128" that you can use for the length.
d. Hint: You will have to set up a uint8_t array that has the UUID in it to use as the pointer to the data. You can use the macro in the GATT DB header file as the initialization to the array to set the value. For example:
    i. *uint8_t capsense_service_uuid[LEN_UUID_128] = { __UUID_CAPSENSE };*
e. Hint: don't forget to increase the number of elements in the advertising data array.

9. Write a thread function to read the CapSense button data from the shield every 100ms.
    a. Hint: you can use the thread from the peripherals chapter exercise on reading the CapSense buttons as a starting point. If you do that, everything is done except for saving the value to the GATT database.
    b. Before the main loop in the thread, initialize the I2C master.
    c. Do an initial I2C write to set the appropriate offset for the button data.
    d. In the main loop in the thread, perform an I2C read to get the latest button data.
    e. If the value has changed, save the button data to the correct location in the GATT database (the array name is ex02_ble_con_capsense_buttons and you need up update the third element in the array – i.e. index 2).
        i. Hint: The details of the CapSense Service and its Characteristics can be found at: http://www.cypress.com/documentation/software-and-drivers/cypresss-custom-ble-profiles-and-services in the file "CYPRESS CAPSENSE® SERVICE_001-97543.pdf". Among other things, this file explains why the Buttons Characteristic is 3 bytes and what each byte means.
    f. Delay for 100ms.
10. In the application initialization (ex02_ble_con_app_init, which is called during the event BTM_ENABLED_EVT) initialize and create the CapSense thread.
11. In the main C file, change the debug UART to WICED_ROUTE_DEBUG_TO_PUART so that debug messages will show up on a terminal window.
12. Find and comment out the call to wiced_bt_set_pariable_mode since we don't want to allow pairing yet. This will be covered in the next chapter.

Testing

1. Program the project to the board.
2. Open the mobile CySmart app.
    a. For Android, open the app settings and make sure that "Initiate pairing after connection" is not checked since we are not (yet) allowing pairing.
    b. For iPhone (YFS: do we need to do anything here?)
3. Connect to the device.
4. Open the GATT browser widget and then open the CapSense Service followed by the CapSense Button Characteristic.
5. Read the value while touching different buttons and observe that the value changes.

6. Hint: There is a CapSense widget in CySmart but it won't work because it depends on Notifications which we have not covered yet. That will be added to the project in the next chapter.
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 CapSense button values in CySmart by clicking on the characteristic and then clicking the "Read Value" button. Continue reading as you touch different buttons and verify that the values are correct.
    a. Hint: The Button Characteristic will be listed with its 128-bit UUID.
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?