Chapter 4A: The Essential BLE 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 2

4A.1.1 Turning on the WICED Bluetooth Stack 5

4A.1.2 Start Advertising 6

4A.1.3 Make a Connection 7

4A.1.4 Exchange Data 8

4A.2 Advertising Packets 9

4A.3 Attributes, the Generic Attribute Profile & GATT Database 10

4A.3.1 Attributes 10

4A.3.2 Profiles – Services - Characteristics 12

4A.3.3 Service Declaration in the GATT DB 12

4A.3.4 Characteristic Declaration in the GATT DB 13

4A.4 WICED Bluetooth Designer 15

4A.5 WICED Bluetooth Stack Events 28

4A.5.1 Essential Bluetooth Management Events 28

4A.5.2 Essential GATT Events 28

4A.5.3 Essential GATT Sub-Events 29

4A.6 WICED Bluetooth Firmware Architecture 30

4A.6.1 Turning on the Stack 30

4A.6.2 Start Advertising 30

4A.6.3 Making a Connection 31

4A.6.4 Exchange Data – Read (from the Central) 31

4A.6.5 Exchange Data – Write (from the Central) 32

4A.7 WICED GATT Database Implementation 34

4A.7.1 gatt\_database[] 34

4A.7.2 gatt\_db\_ext\_attr\_tbl 36

4A.7.3 uint8\_t Arrays for the Values 37

4A.7.4 The Application Programming Interface 37

4A.8 Exercises 39

Exercise - 4A.1 Create a BLE Advertiser 39

Exercise - 4A.2 Connect using BLE 42

# WICED BLE System Lifecycle

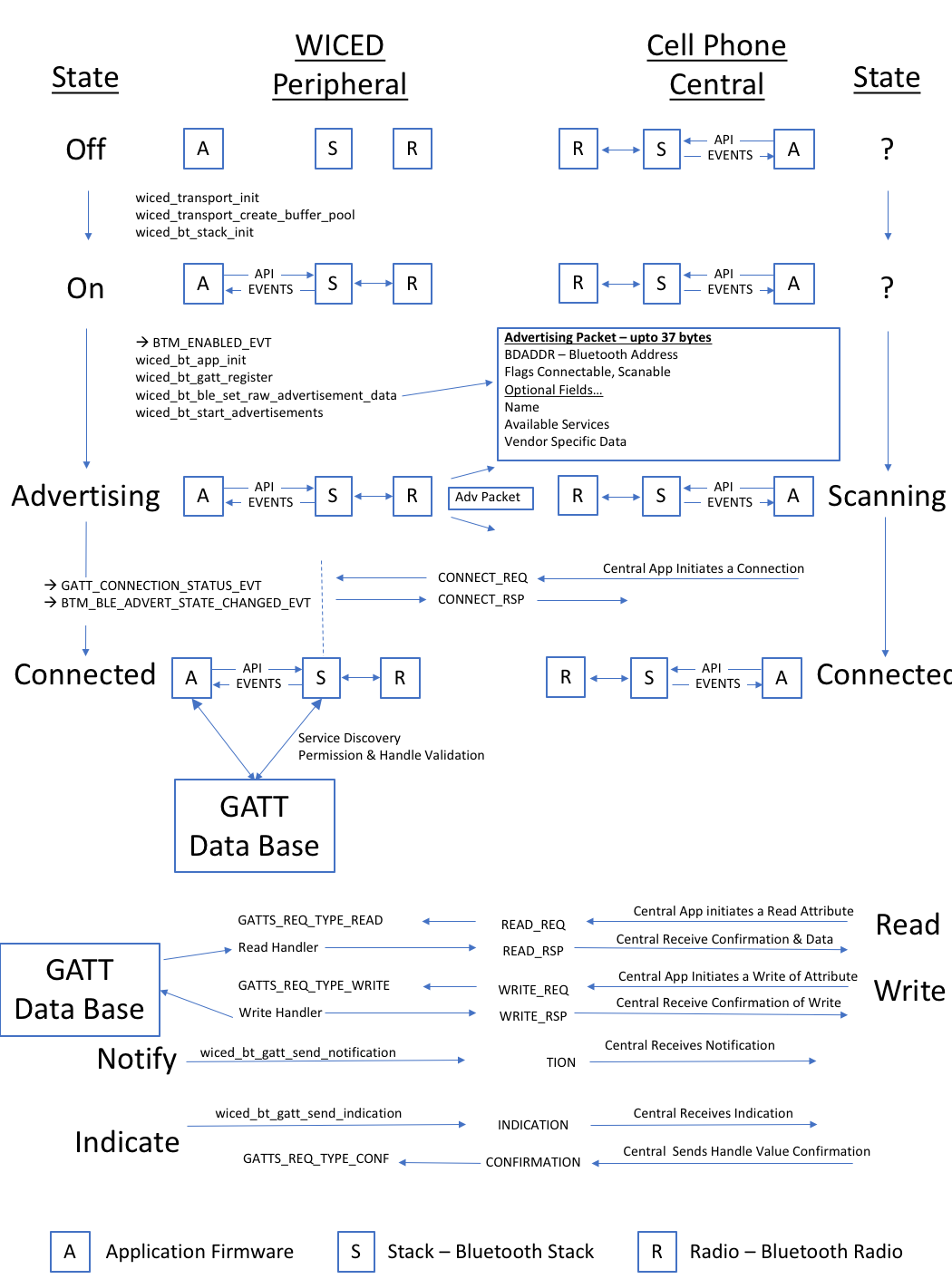
Basically, every book that I have ever read on Bluetooth or WiFi starts with the radio stack and works its way back (or up depending on your point of view) to the Application. You know the drill, 2.4 GHz Digital Spread Spectrum, Adaptive Frequency Hopping, blah blah blah. This approach surfaces a bewildering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

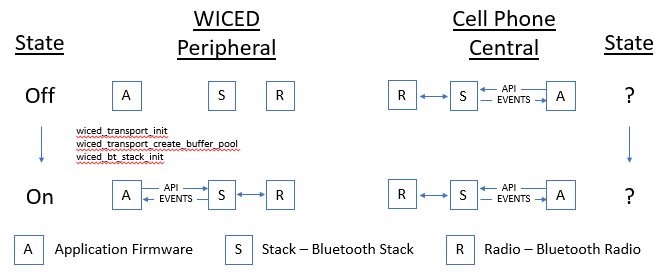


## Turning on the WICED Bluetooth Stack

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.

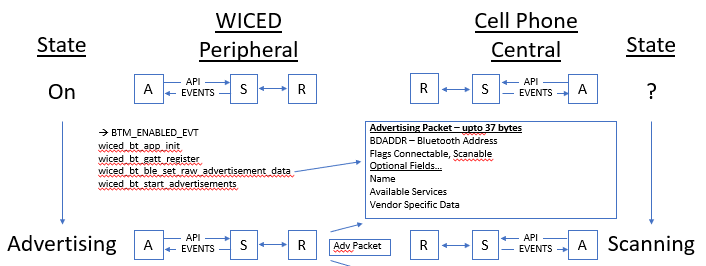
The first thing you do in your firmware is to turn on BLE. In WICED, that means that you initialize the Stack and provide it with a function that will be called when the Stack has events for you to process (this is often called the "callback" function for obvious reasons).



## 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 regular, 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. There are several things that can be done to enhance security, which I will address 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.

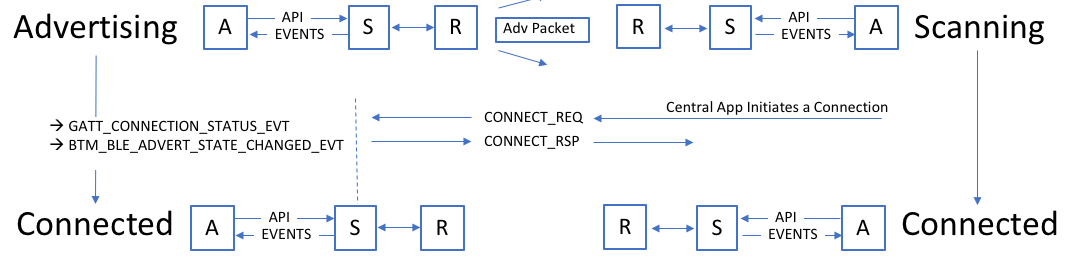
The format of the packet is quite simple. Each item you wish to advertise starts with a length byte, followed by the type (e.g. Flags or Name) and then the data, the size of which is determined by that length byte. The items are simple concatenated together, up to 31 bytes.

## 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).

In your Application firmware, you will now notice that you are connected and that you are no longer advertising. Now you can wait for more messages from the Central.



## Exchange Data

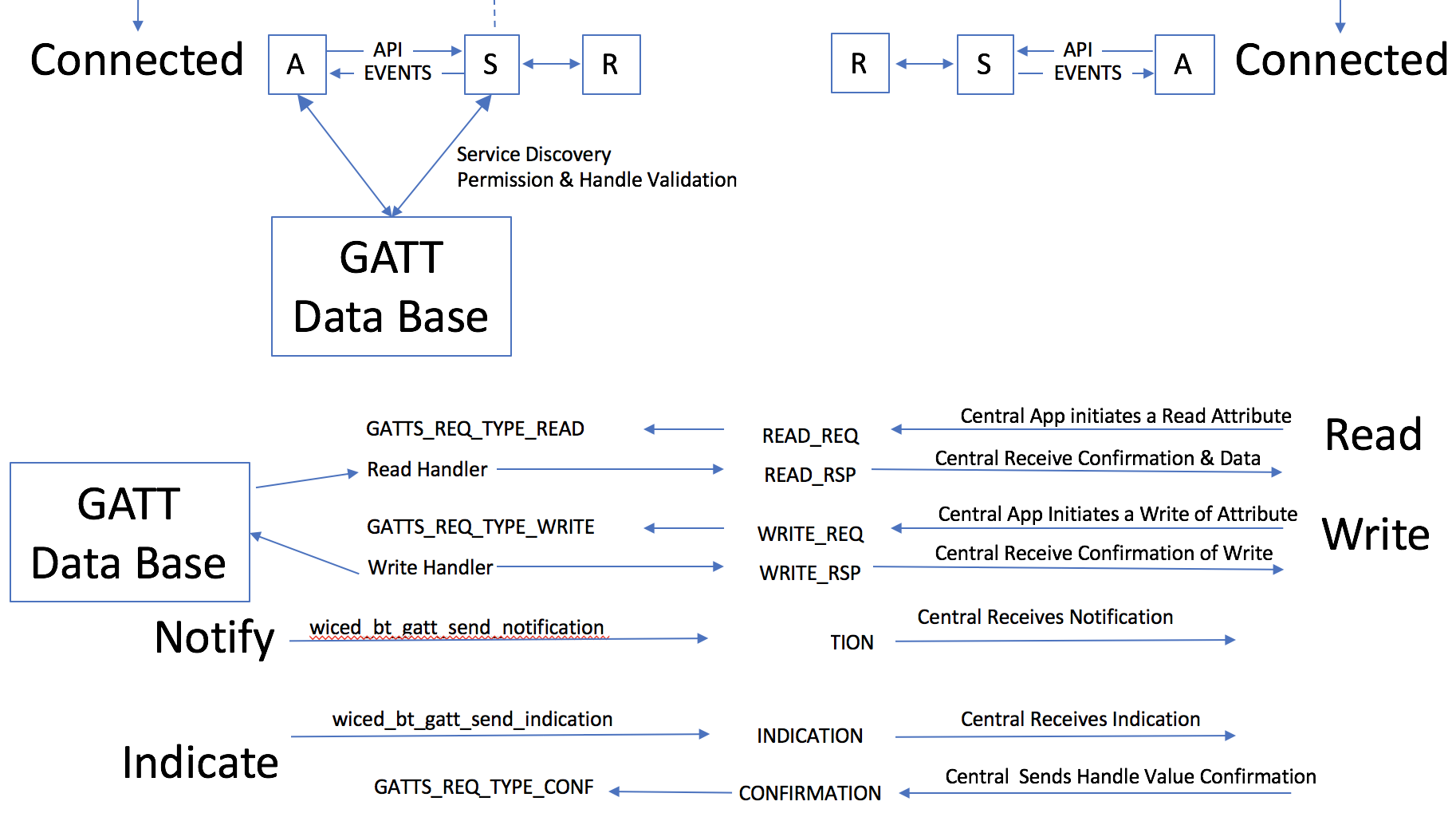
Now that you are connected you need to be able to exchange data. In the world of BLE this happens via the Attribute Protocol (ATT). The basic ATT protocol has 4 types of transactions: Read & Write which are initiated by the Central and Notify & Indicate which are initiated by the Peripheral.

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.



# Advertising Packets

The Advertising Packet is a string of 3-31 bytes that is broadcast at a regular, configurable interval. The packet is broken up into variable length fields that are in the form of

* 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 (i.e. is it connectable)

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

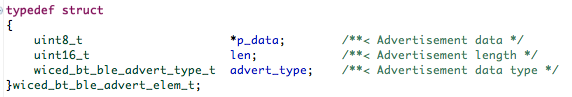


For example, if you had a device named “Kentucky” you could the name to the Advertising packet by adding the following bytes to your Advertising packet.

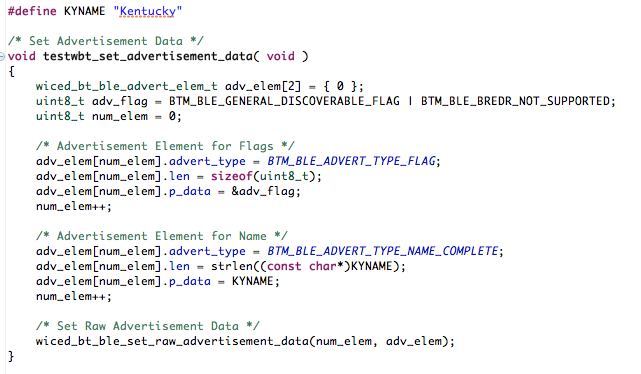
* 9 (The length is 9)
* BTM\_BLE\_ADVERT\_TYPE\_NAME\_COMPLETE
* ‘K’, ‘e’, ‘n’, ‘t’, ‘u’, ‘c’, ‘k’, ‘y’

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:



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



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 a number of interesting use cases which we will talk about in more detail in the next Chapter.

# Attributes, the Generic Attribute Profile & GATT Database

## Attributes

As described 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](https://www.bluetooth.com/specifications/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.

## 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.

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.

## 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.



(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.



(This figure is taken from the Bluetooth Specification)

## 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.



(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.



(This figure is taken from the Bluetooth Specification)

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

# WICED Bluetooth Designer

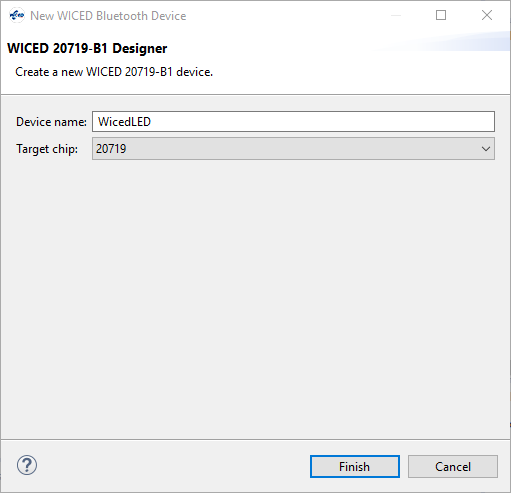
The WICED Bluetooth Designer is a tool that will build a semi-customized template project for you for BLE or BR/EDR (aka Classic Bluetooth) or both. The tool copies in all the required files including the makefile, customizes them to your settings, and then creates a make target. The project is runnable 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 “LED1”. When the Central writes a 0 or 1 into that Characteristic, my application firmware will just write that value into the GPIO driving LED1.

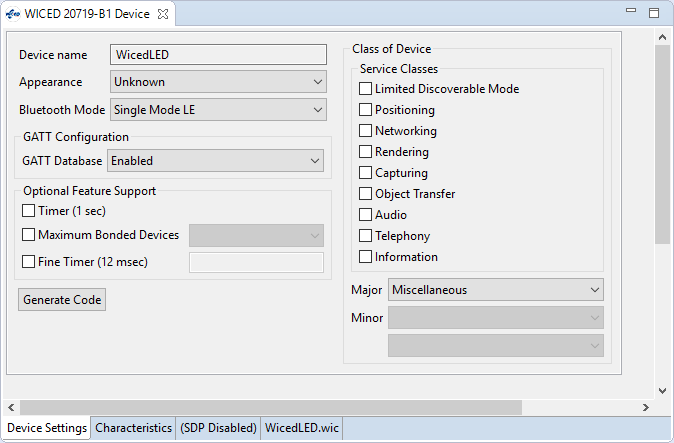
To run the tool, select the menu item *File->New->WICED Bluetooth Designer*.



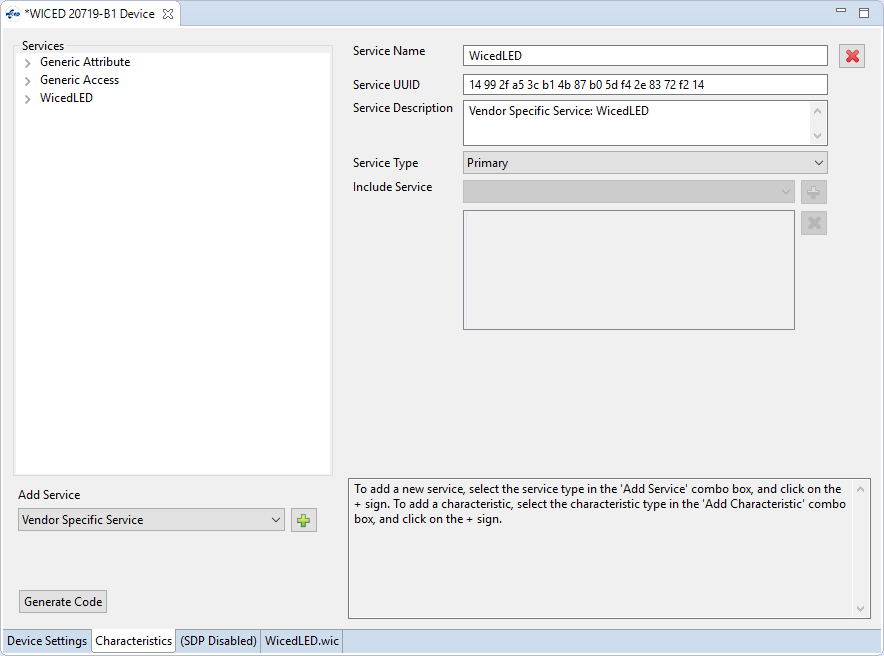
This will ask you to name your Project (also called the Device Name) and select a chip. In this case, I’ll call the project *WicedLED* and I'll pick *20719* for the chip.



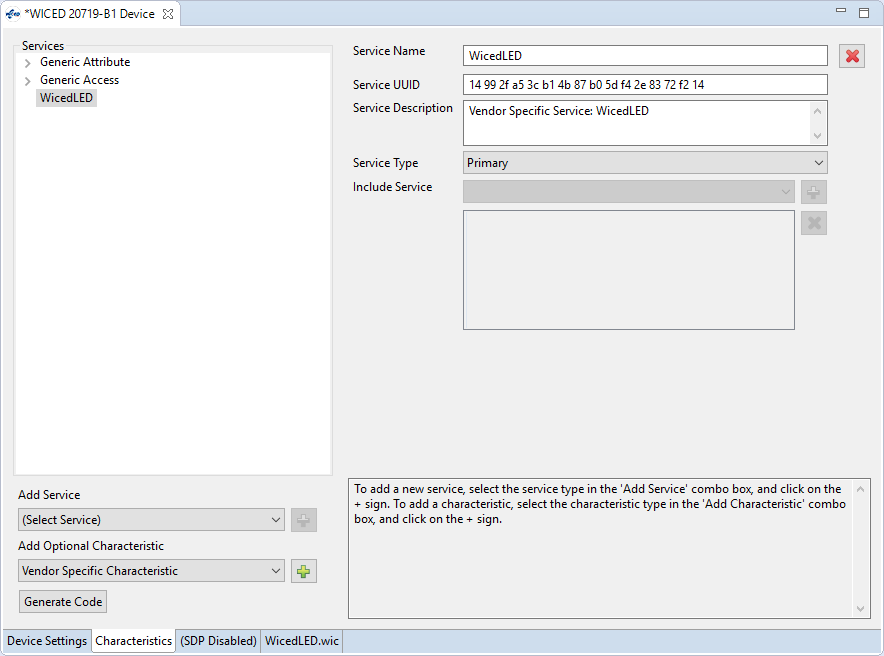
After you click Finish, you get a window allowing you to pick Dual Mode (aka BLE and classic), BR/EDR or Single Mode LE (aka BLE) along with some other options. I want the tool to help me build the GATT Database so I leave that enabled.



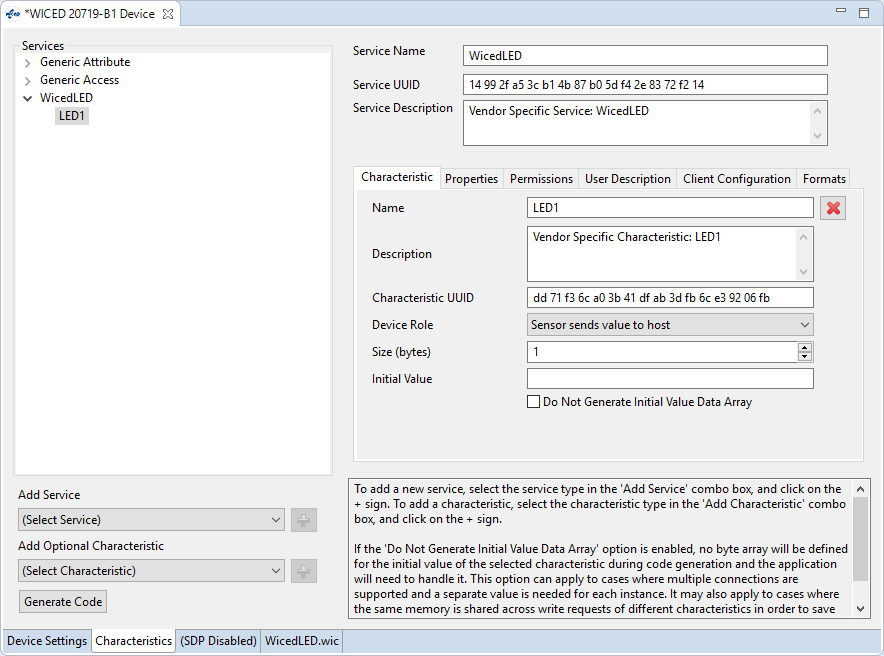
The next step is to setup a Service. To do this select the Characteristic tab. Then pick “Vendor Specific Service” and press the “+” button. After I do this I will see a new Service called “WicedLED” added to my Services. Notice that I could change the name in the “Service Name” box. I also let the tool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



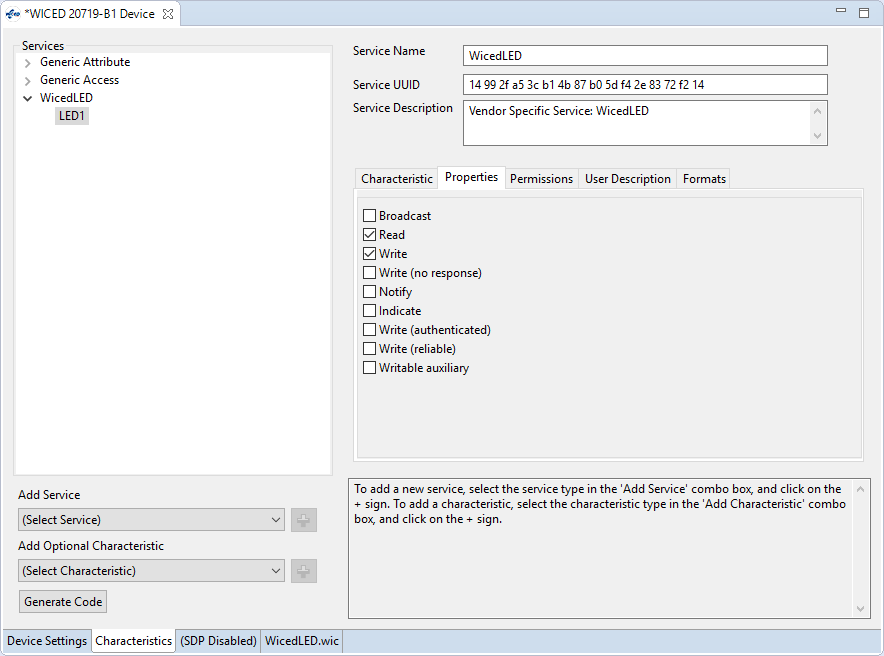
After the Service is configured I add one Characteristic by clicking on “WicedLED” in the Services window, then select “Vendor Specific Characteristic” and press the “+”.



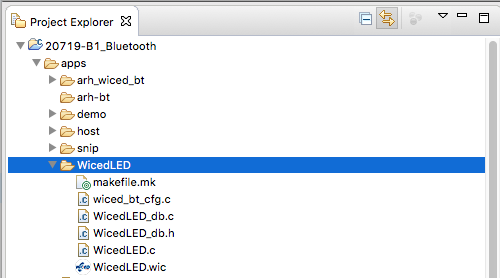
I then change the name of the Characteristic to "LED1", specify that I want the Size to be 1 byte and leave the Initial Value blank which will result in a starting value of 0x00 (if you want a non-zero value in this field, you must put exactly 2 hex digits per byte with exactly 1 space between bytes 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. There are two important things you should know about this: (1) it is a one-way operation; and (2) it re-generates all the files after creating backup copies so any edits you have made to files will have to be re-done. Be careful.

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

To make this work I will make two tiny changes to the generated project.

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



Then, in WicedLED.c add two lines of code to write the LED and printout the result. 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\_LED1\_VALUE:

/\* Turn the LED on/off depending on the value written to the GATT database \*/

WICED\_BT\_TRACE("Output = %d\n",wicedled\_wicedled\_led1[0]);

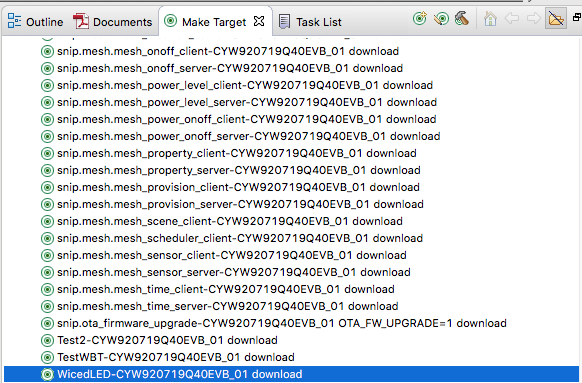
wiced\_hal\_gpio\_set\_pin\_output(WICED\_GPIO\_PIN\_LED\_1, wicedled\_wicedled\_led1[0]);

**break**;

}

Notice how the GATT attribute (wicedled\_wicedled\_led1) is updated for you by the stack when the write command is processed.

Notice that Bluetooth Designer created a make target for me. If you are using a different platform you need to edit the target.

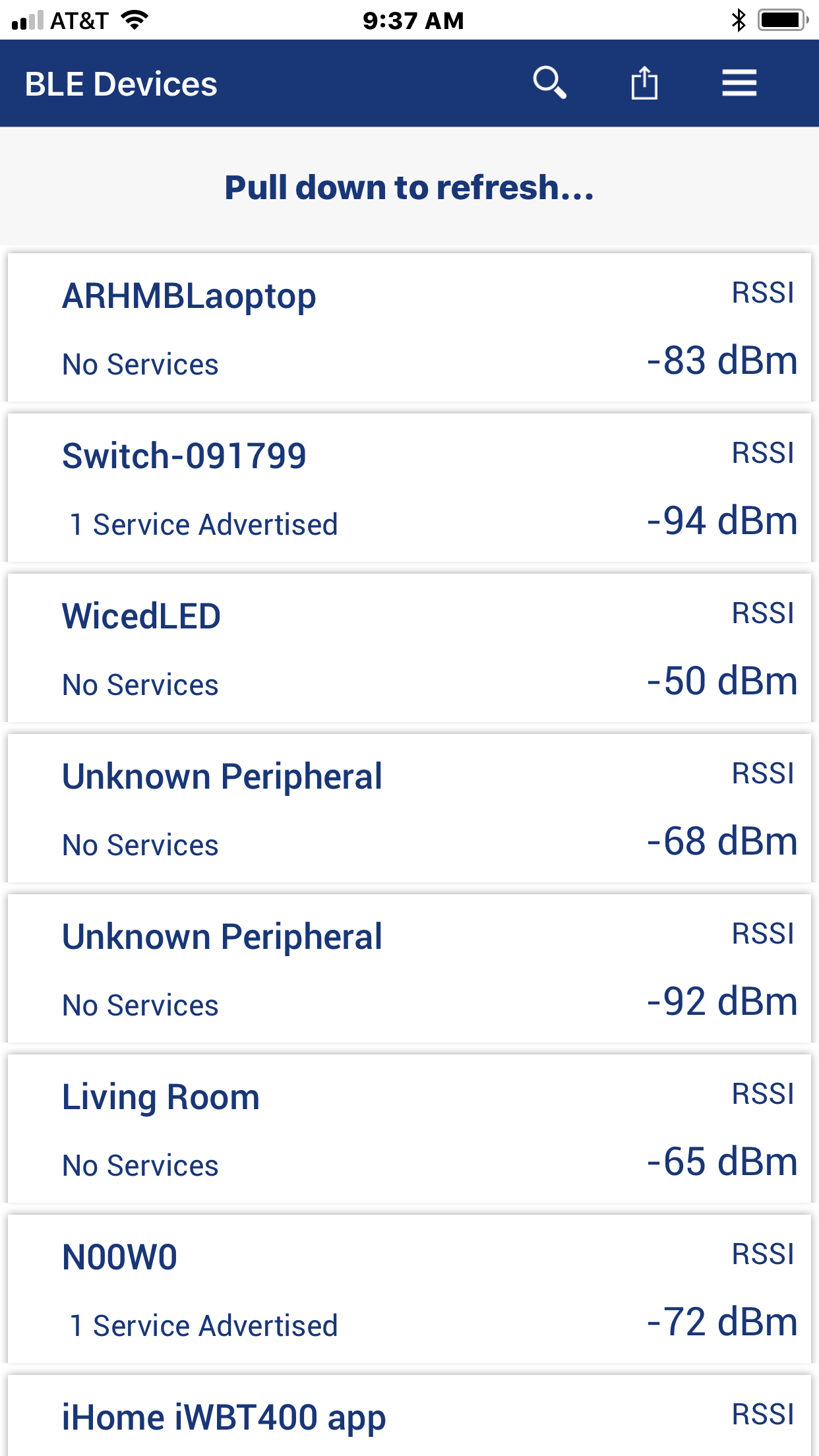


Now run the make target. It will build and program the board. When the application firmware starts up you see some messages.



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

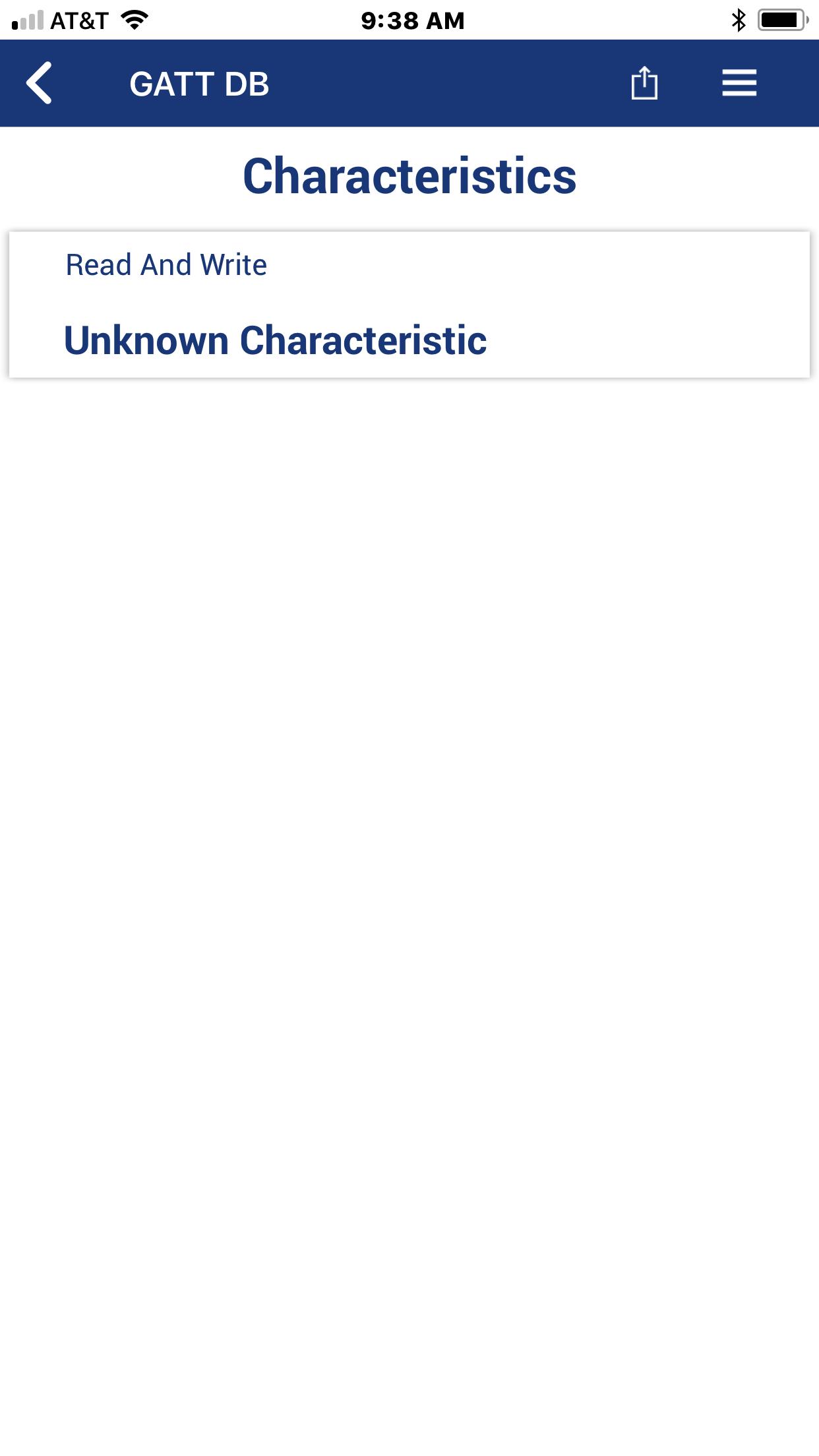
Note: If you are using the Android version of CySmart, before connecting go to the Settings and turn off the option "Initiate pairing after connection". If not, the connection will fail because we have not (yet) enabled pairing.



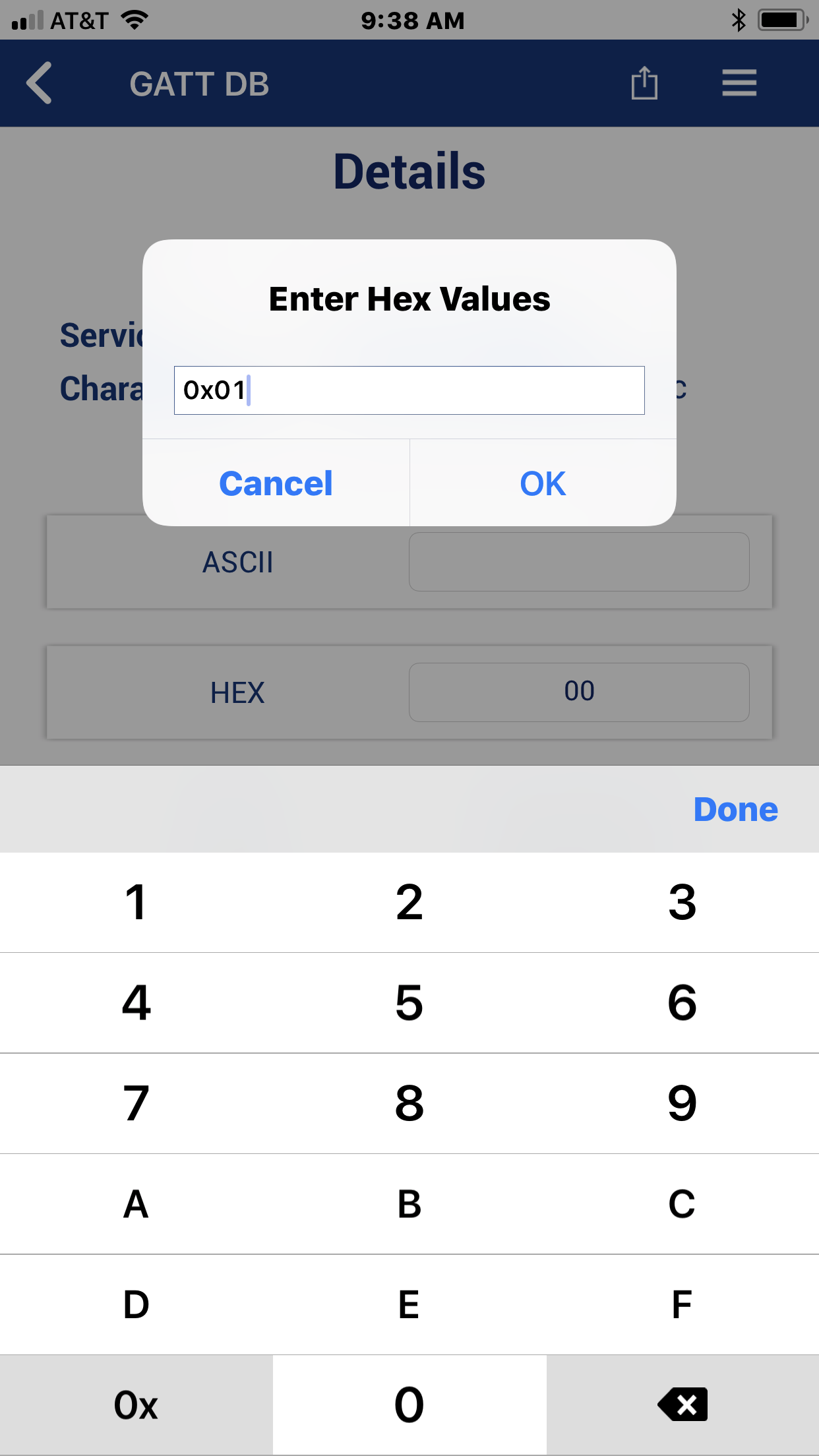
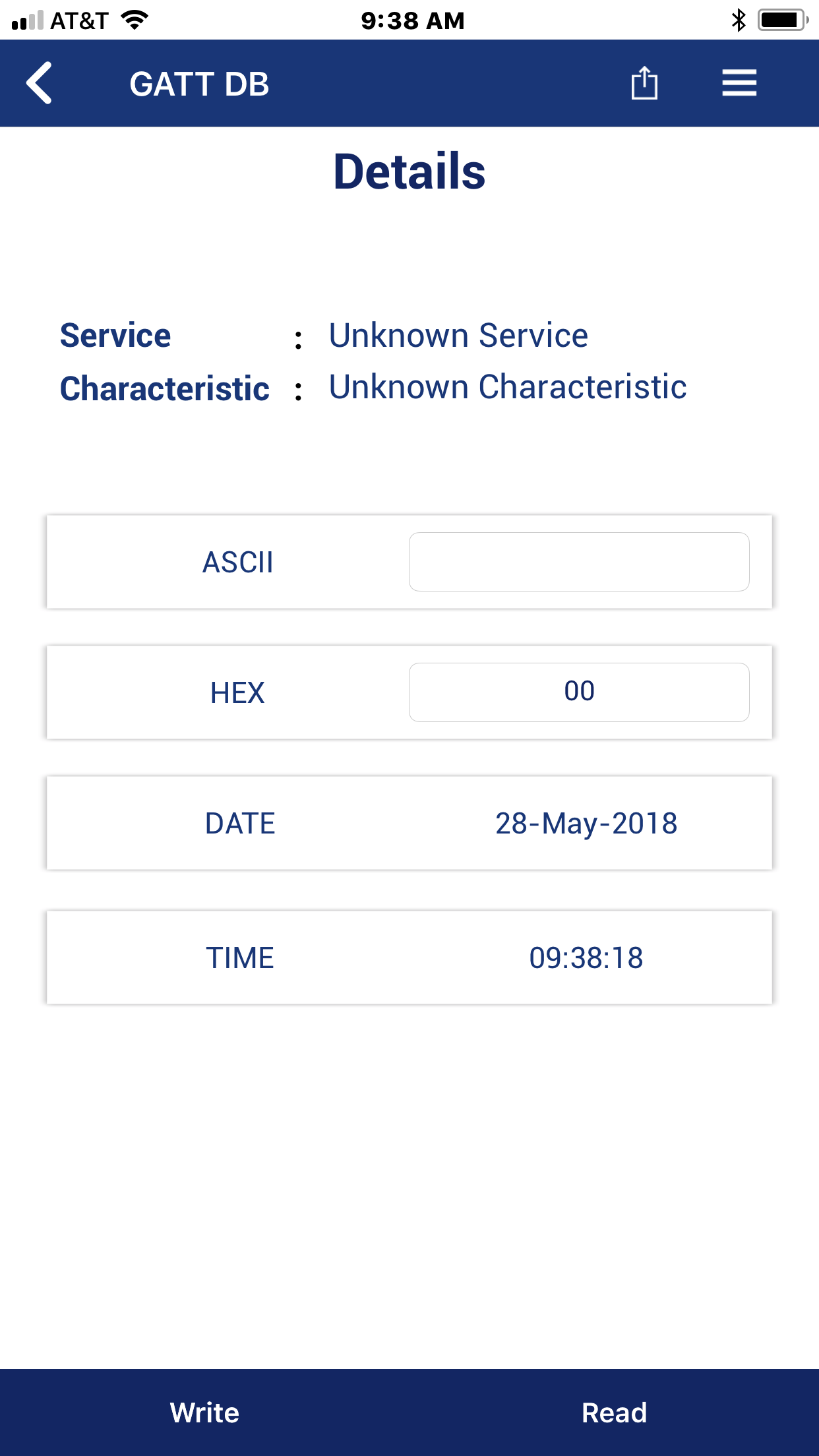
On the terminal window, you will see that there has been a connection and the advertising has stopped.



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 LED1).



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.

# 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:

## 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.

## 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.

## 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).

# 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.

## 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.

## 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 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.

## 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.

## 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

## 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

# 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:

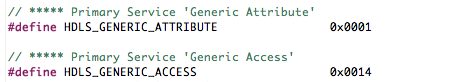
* An Array, named gatt\_database, of uint8\_t bytes that holds the Handles, Types and Permissions.
  + In <appname>\_db.c
* An Array of Structs which holds Handles, a Maximum and Current Length and a Pointer to the actual Value.
  + In <appname>\_db.h and <appname>.c
* The Values as arrays of unint8\_t bytes.
  + In <appname>.c
* Functions that serve as the API
  + In <appname>.c

## 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:



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:

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)

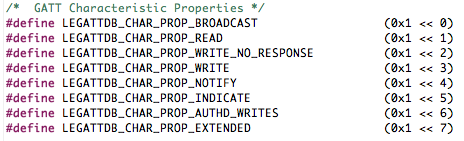
As before the handle parameter is just the 16-bit number that WICED Bluetooth Designer creates for the Attributes for Characteristics which will be in the form of #define HDLC\_ for example:



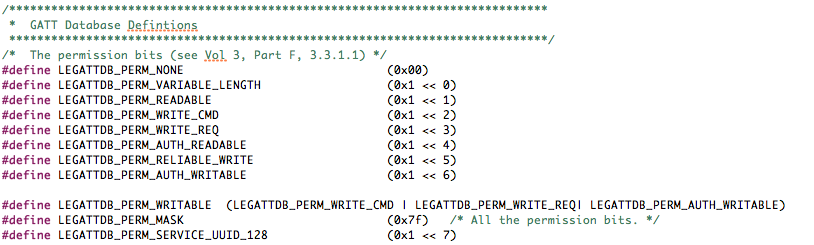
The handle\_value parameter is the Handle of the Attribute that will hold the Characteristic's Value.

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.

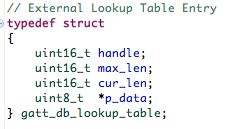


The Permission field is just a bit mask that sets the Permission of an Attribute (remember Permissions are on a per Attribute basis and Properties are on a per Characteristic basis). They are also defined in wiced\_bt\_gatt.h.

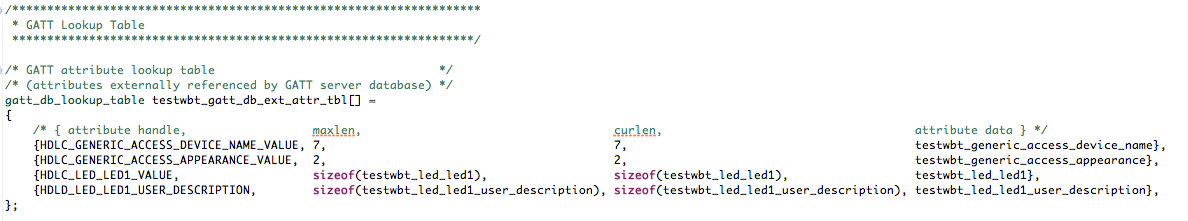


## gatt\_db\_ext\_attr\_tbl

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.



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



API functions will be created by WICED Bluetooth Designer to help you search through this array to find the pointer to the value.

## 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.

## 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.

# Exercises

* 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.
   1. Select *Generic Tag* for the *Appearance*.
   2. Disable the GATT database.
   3. 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. Open the main C file for the project and familiarize yourself with its structure.
5. 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.
   1. Hint: Don't forget to leave the trailing '\0' null termination at the end.
6. 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.
   1. Hint: Right click on the existing advertisement type and select *Open Declaration* to see all the available choices.
7. Add a global variable of type *uint8\_t* called *manuf\_data*. Initialize it to a value of 0.
8. Locate the function that sets up the advertisement data and add a new element to send the *manuf\_data* value.
   1. Hint: The advertisement type for this element should be *BTM\_BLE\_ADVERT\_TYPE\_MANUFACTURER*.
   2. Hint: don't forget to increase the number of elements in the advertising data array.
9. Configure Button1 for a falling edge interrupt. Add a button interrupt callback that does the following:
   1. Clear the pin interrupt
   2. Increment *manuf\_data*
   3. Update the advertisement packet data array
      1. Hint: you can just call the function that Bluetooth Designer created.
   4. Re-start advertisements
10. 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.
   1. Hint: you must have a CY5577 CySmart BLE USB dongle connected to your PC to run CySmart.

### Questions

1. What is the name of the Stack callback function?
2. How many bytes is the advertisement packet?

* 1. 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.*
   1. Select *Unknown* for the *Appearance*.
   2. Enable the GATT database.
   3. Go to the Characteristics tab and add a Vendor Specific Service.
      1. Change the Service Name and Service Description to *CapSense*.
      2. Change the UUID to (Hex): 31 01 9B 5F 80 00 00 80 00 10 00 00 B5 CA 03 00
   4. Add a Vendor Specific Characteristic to the CapSense Service
      1. Change the Name and Description to *Buttons*.
      2. 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.
      3. The size of the characteristic is 3 bytes.
      4. 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).
      5. Set the Properties to *Read*.
   5. 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.
   1. 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.
   1. Hint: Search for *device\_name*.
   2. 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. In the GATT connect callback function:
   1. On a disconnection:
      1. Re-start advertisements using wiced\_bt\_start\_advertisements().
      2. Hint: Search for wiced\_bt\_start\_advertisements to find the parameters required for the function.
      3. Hint: Search for "TODO: Handle the disconnection" to find where to re-start advertisements.
8. Update the advertisement packet so that it sends the flags, name, and the UUID of the CapSense service.
   1. 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.
   2. Hint: The advertisement type for a complete service name is *BTM\_BLE\_ADVERT\_TYPE\_128SERVICE\_DATA*.
   3. Hint: There is a macro called "LEN\_UUID\_128" that you can use for the length.
   4. 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:
      1. *uint8\_t capsense\_service\_uuid[LEN\_UUID\_128] = { \_\_UUID\_CAPSENSE };*
   5. 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.
   1. 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.
   2. Before the main loop in the thread, initialize the I2C master.
   3. Do an initial I2C write to set the appropriate offset for the button data.
   4. In the main loop in the thread, perform an I2C read to get the latest button data.
   5. 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).
      1. 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.
   6. Delay for 100ms.
10. Add includes for the RTOS header file (wiced\_rtos.h) and I2C header file (wiced\_hal\_i2c.h) at the top of the main C file.
11. In the application initialization (ex02\_ble\_con\_app\_init, which is called during the event BTM\_ENABLED\_EVT) initialize and create the CapSense thread.
12. 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.

### Testing

1. Program the project to the board.
2. Open the mobile CySmart app.
   1. For Android, open the app settings and make sure that "Initiate pairing after connection" is not checked since we are not (yet) allowing pairing.
   2. 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.
    1. 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?