Chapter 4: The Essential BLE Example

Time 2 Hours

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

3.1 WICED BLE System Lifecycle 2

3.1.1 Turning on the WICED Bluetooth Stack 4

3.1.2 Start Advertising 4

3.1.3 Make a Connection 5

3.1.4 Exchange Data 6

3.2 Attributes, the Generic Attribute Profile & GATT Database 7

3.2.1 Attributes 7

3.2.2 Profiles – Services - Characteristics 7

3.2.3 Service Declaration in the GATT DB 8

3.2.4 Characteristic Declaration in the GATT DB 8

3.2.5 Minimum Services 10

3.3 WICED Bluetooth Designer 10

3.4 WICED Bluetooth Firmware Architecture 10

3.4.1 Turning on the WICED Bluetooth Stack 10

3.4.2 Start Advertising 10

3.4.3 Making a Connection 11

3.4.4 Exchange Data – Read (from the Central) 11

3.4.5 Exchange Data – Write (from the Central) 12

3.5 WICED GATT Database Implementation 12

3.5.1 gatt\_database[] 13

3.5.2 gatt\_db\_ext\_attr\_tbl 15

3.5.3 uint8\_t Arrays for the Values 15

3.5.4 The Application Programming Interface 16

3.6 WICED Bluetooth Stack Events 17

3.6.1 Essential Bluetooth Management Events 17

3.6.2 Essential GATT Events 17

3.6.3 Essential GATT Server Events 17

Things I want to fix

* A better definition of profile
* A better description of the required services with some explanation of WTF they are

# WICED BLE System Lifecycle

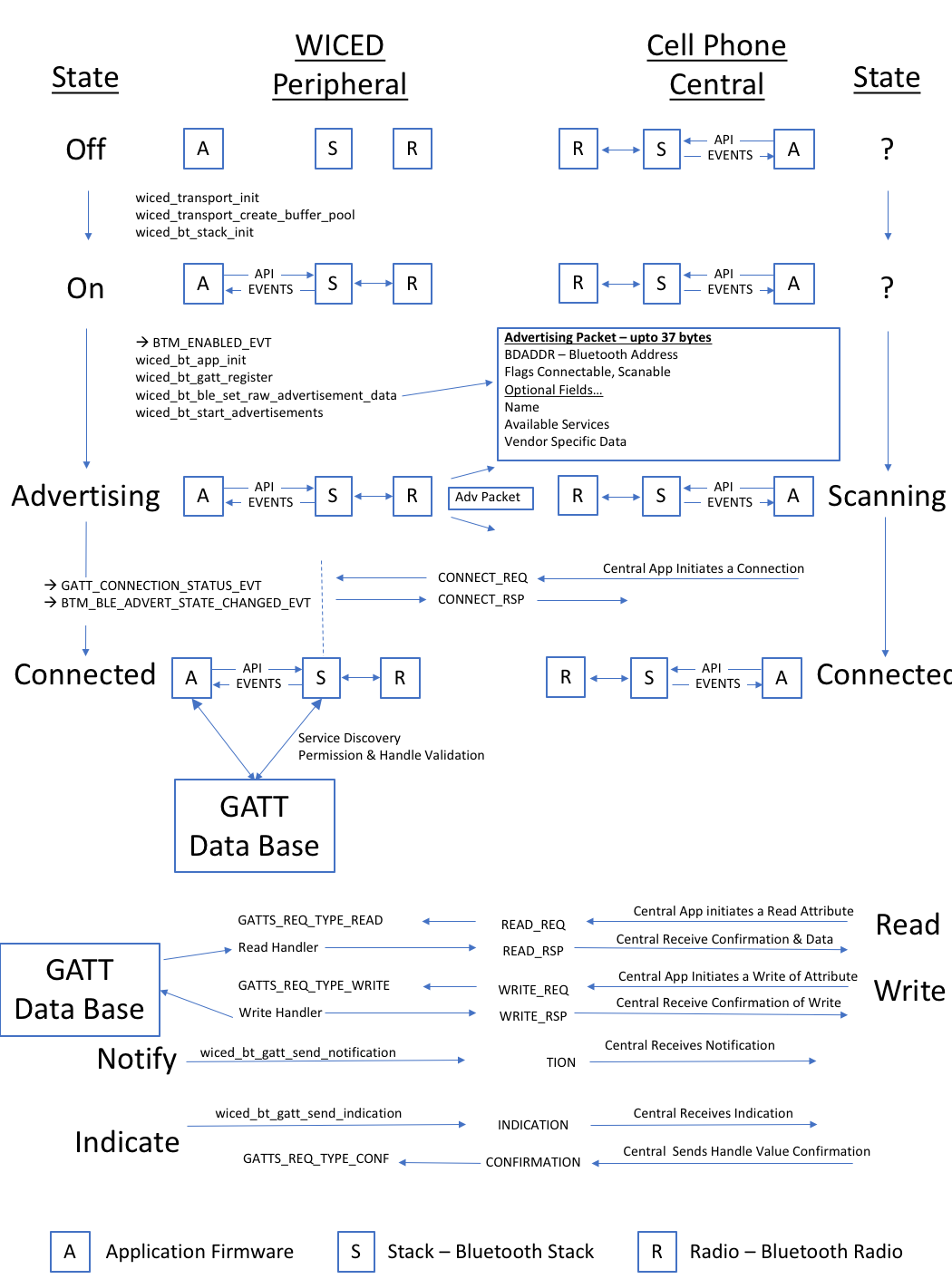
Basically, every book that I have ever read on Bluetooth or WiFi starts with the radio stack and works its way back (or up depending on your point of view) to the Application. You know the drill, 2.4 GHz Digital Spread Spectrum, Adaptive Frequency Hopping blah blah blah. This approach surfaces a bewildering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

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 Apps on those devices.

There are 4 steps

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

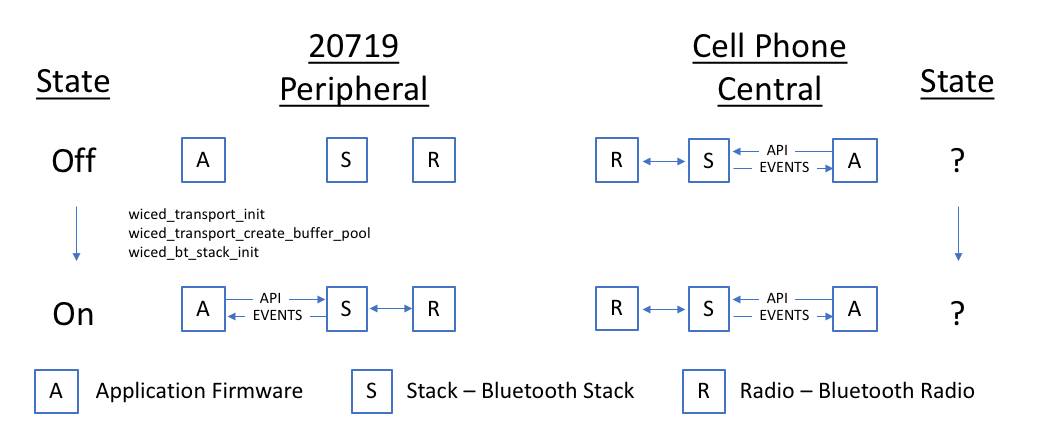
Here is the picture:



## 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.

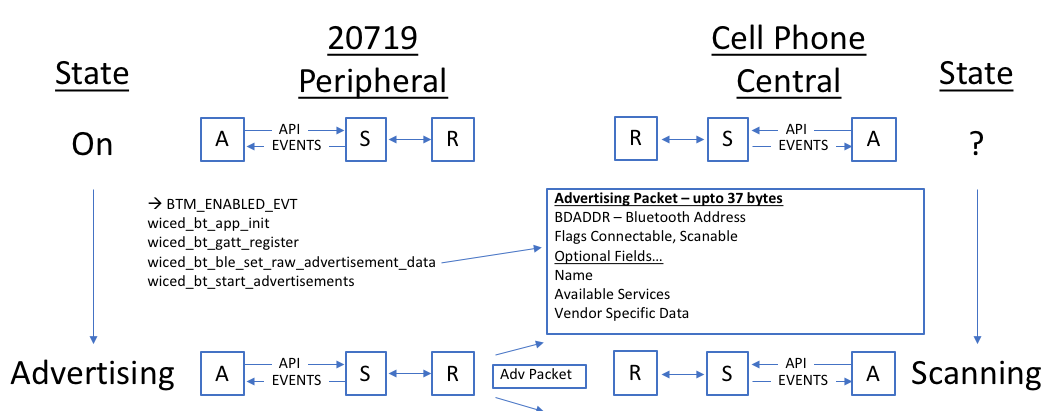
Like all great partnerships, every BLE connection has two sides, one side called the **Peripheral** and one side called the **Central**. In the picture below you can see that the Peripheral starts Off, there is no connection from the Peripheral to the Central (which is in an unknown state). In fact,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



## 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.

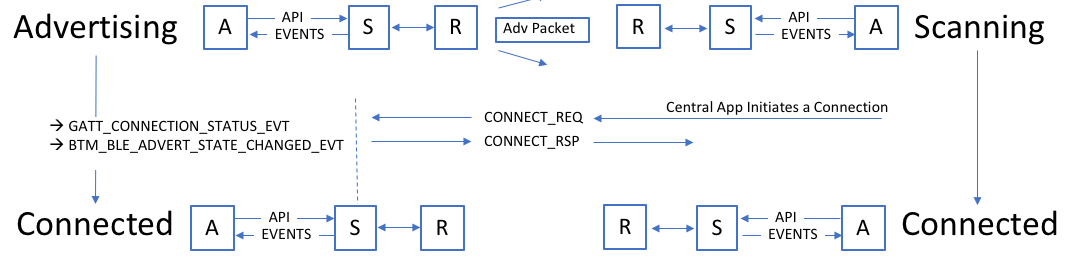


## Make a Connection

Once a Central device processes your advertising packet it can choose what to do next including initiate a connection. When the Central App initiates a connection, it will call an API which will trigger the Bluetooth Stack which will generate a Bluetooth Packet called a “connection\_req” which will go out the Central radio and through the air to your WICED radio.

The WICED radio will feed the packet to the WICED Bluetooth Stack which will respond AUTOMATICALLY back with a “connection\_rsp” packet.

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

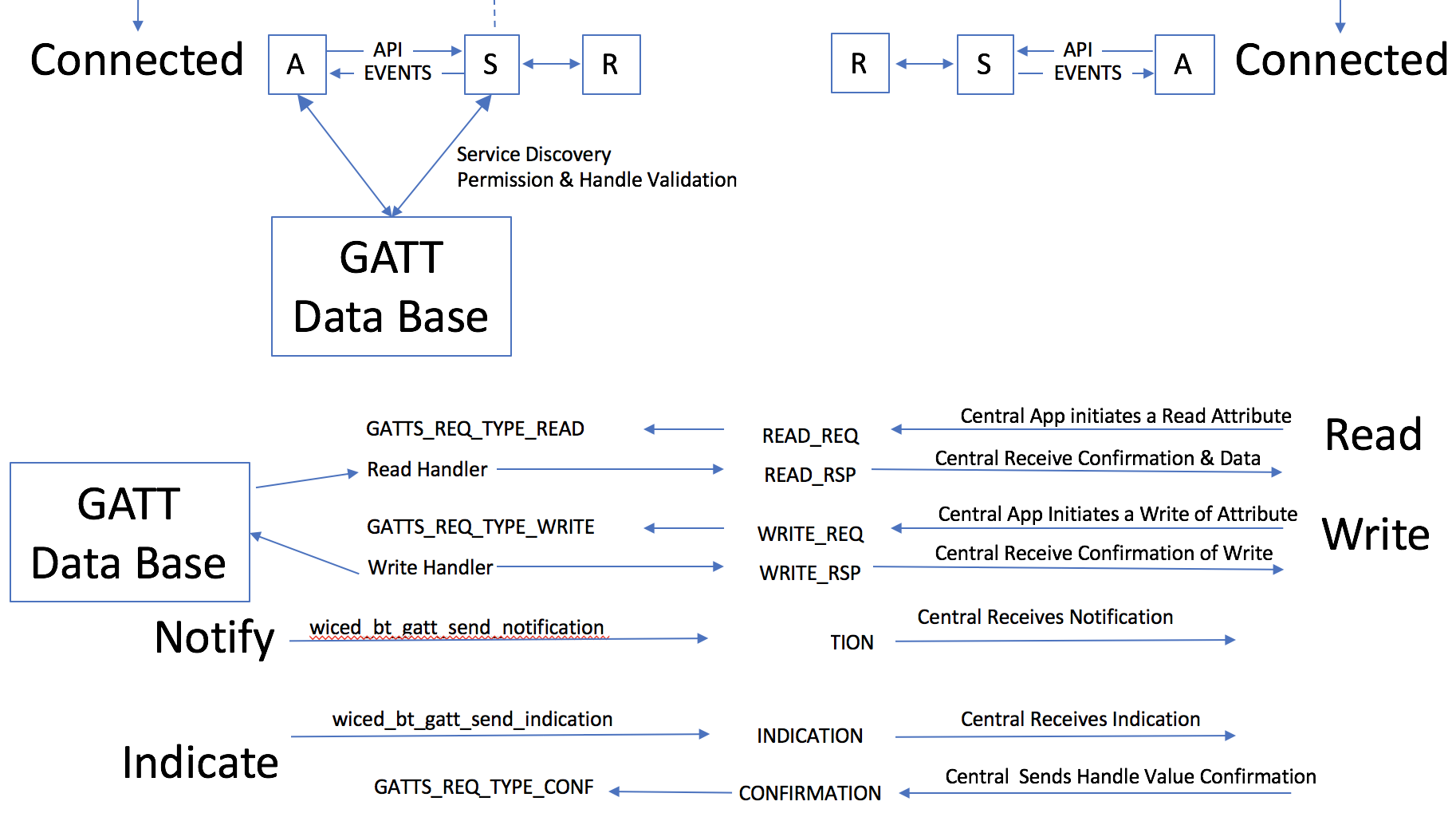
Now that you are connected you need to be able to exchange data. In the world of BLE this happens via the Attribute Protocol (ATT). The basic ATT protocol has 4 types of transactions, Read & Write which are initiated by the Central and Notify & Indicate which are initiated by the Peripheral.

ATT Protocol transactions are all keyed to very simple database called the GATT database which is running on the Peripheral. Because the GATT Database is running on the Peripheral, that side is also known commonly known as the GATT Server. And 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 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.



# 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.



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 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 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 at 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 Bluetooth stack what it can and cannot do in response to requests from the Central/Client. The Permissions are just a bit field with 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. And once again WICED helps you get the permission set correctly when you make the database. The Stack also takes care of enforcing the Permissions

## Profiles – Services - Characteristics

The GATT Database is “flat”, just a bunch rows with one Attribute per row. The problem is a totally flat organization is a painful, 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 possibly 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 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 Sensors states represented as Characteristics.

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, the UUID of the Service and the Attribute Permissions.



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.

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



## 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 permission (aka Properties). 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.



Each Characteristic also has a set of Properties that define what the Central/Client can do with the Characteristic. These Properties are used by the WICED Bluetooth 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 that it 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 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
* Indicate – The Client can ask for Indications of Characteristic value changes by the Server and requires a response by the Client/Central
* Authenticated Signed Writes – The client it can perform digitally signed writes
* Extended Properties – Indicates the existence of more Properties (mostly unused)

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

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.



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

## Minimum Services

To function, every BLE device is required to have a minimum of two services.

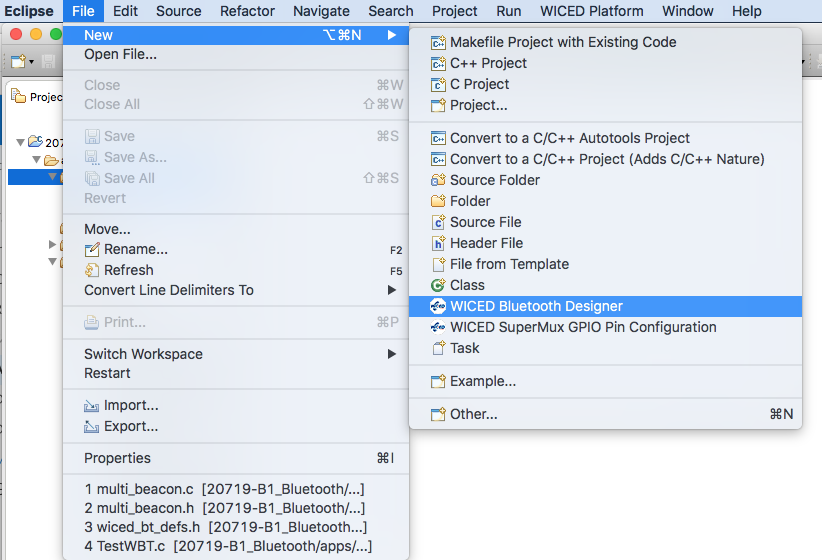
* Generic Attribute Service
* Generic Access Service

# WICED Bluetooth Designer

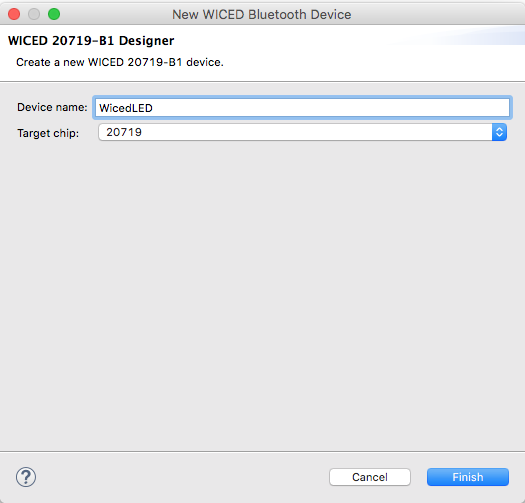
The WICED Bluetooth Designer is a tool that will build a semi-customized template project for you for BLE or BR/EDR or both. The tool copies in all the required files including the makefile, customizes them to your setting and then sets up a make target. The project is runnable with no changes (it doesn’t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

For this example project I am going to build a BLE project that has One 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.

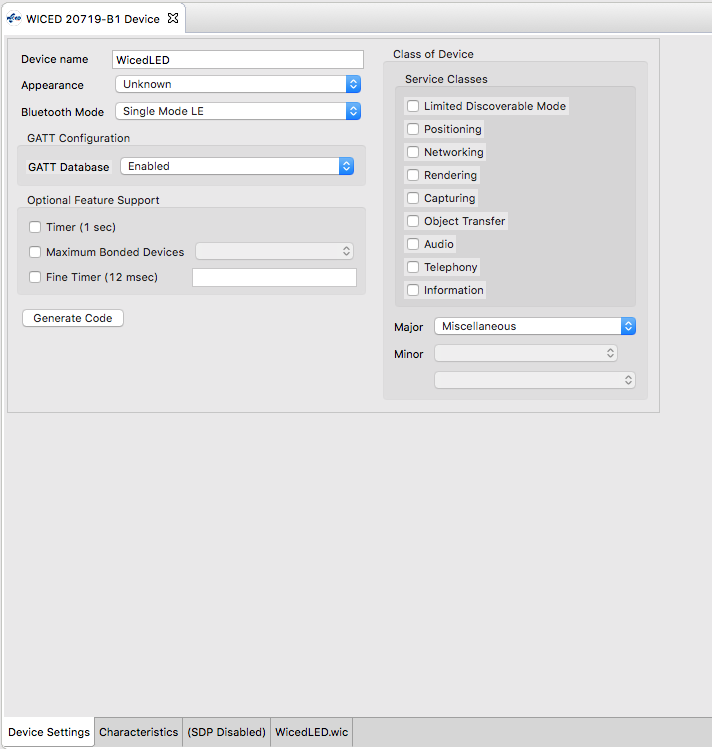
To run the tool, go to the File->New->WICED Bluetooth Designer menu



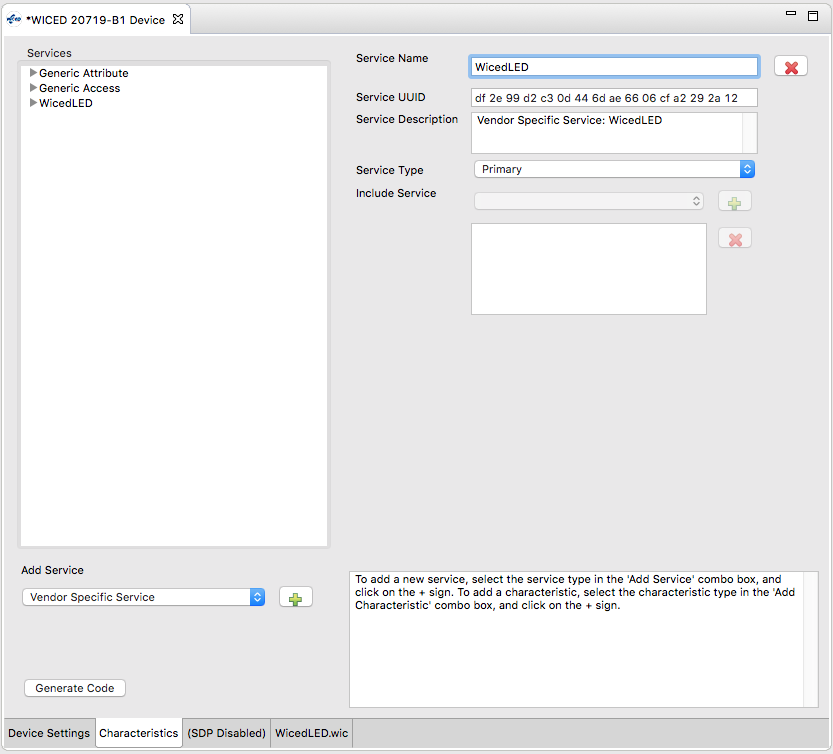
This will ask you to name your Project (also called the Device Name) and select a chip. In this case, I’ll pick 20719.



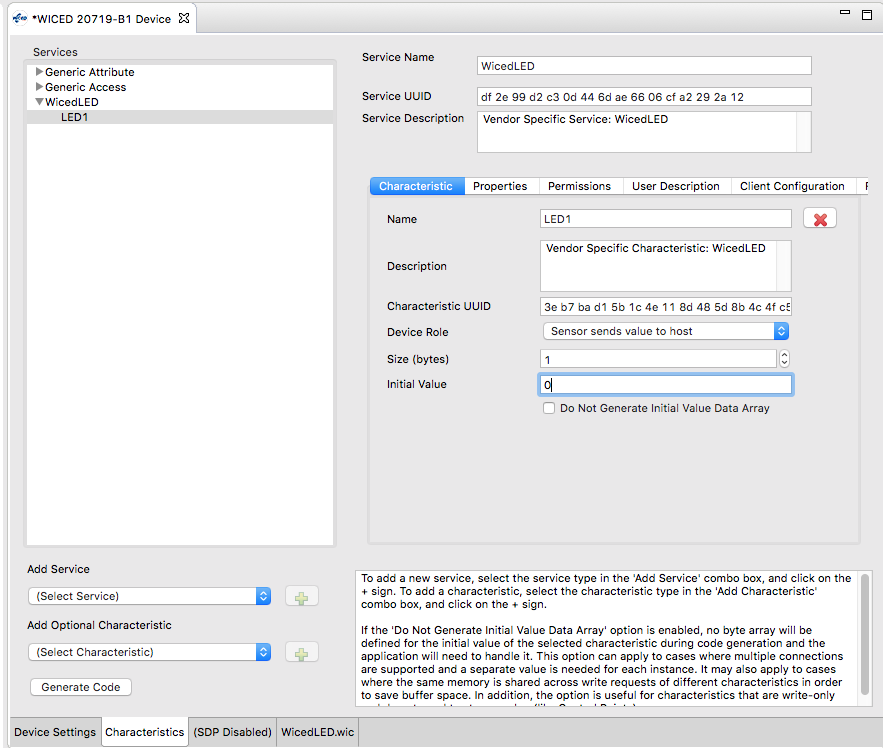
You then can pick Dual Mode, BR/EDR or Single Mode LE (aka BLE). I want the tool to help me build the GATT Database so I leave that enabled.



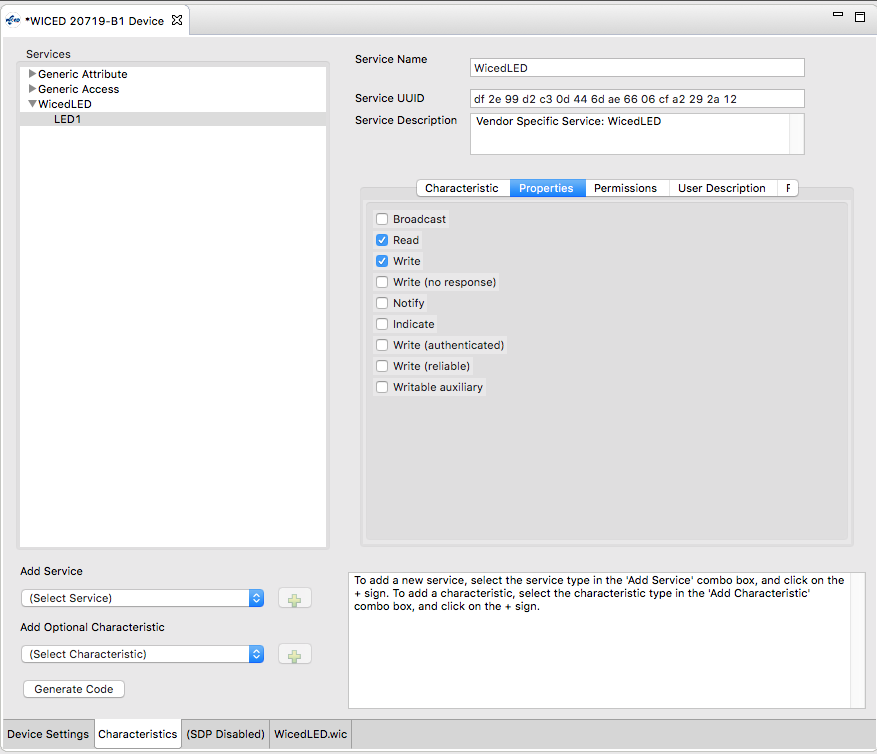
The next step is to setup a Service. To do this select the Characteristic tab. Then pick “Vendor Specific Service” and press the “+” button. After I do this I will see a new Service called “WicedLED” added to my Services. Notice that I could change the name in the “Service Name” box. I also let the tool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



After the Service is configured I add one characteristic by clicking on the “WicedLED” in the Services window, then select “Vendor Specific Characteristic” and press the “+”. I then specify that I want the “Size” to be 1 and the initial value to be 0

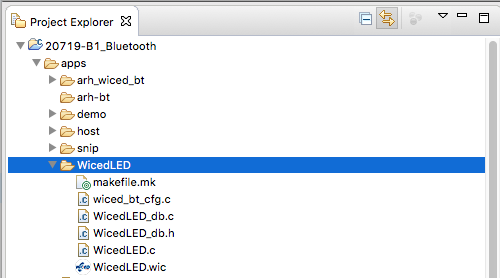


I want this Characteristic to be read or write, so click on the “Properties” tab and pick out “Read” and “Write”



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 blows away whatever files you already have. 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 function I will make three tiny changes to the generated project.

I want the state of the LED to be 0 or 1, not the character ‘0’. Open WicedLED.c and go to the place where the LED Characteristic is initialized.



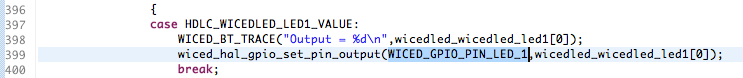
And change it to 0 instead of ‘0’



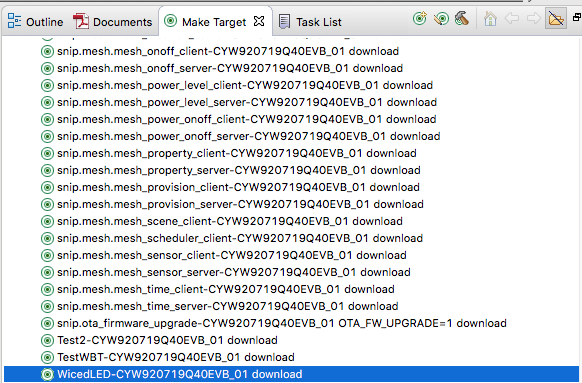
Next I want to use the PUART, so uncomment line 134 and comment out 137



Finally, 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”.



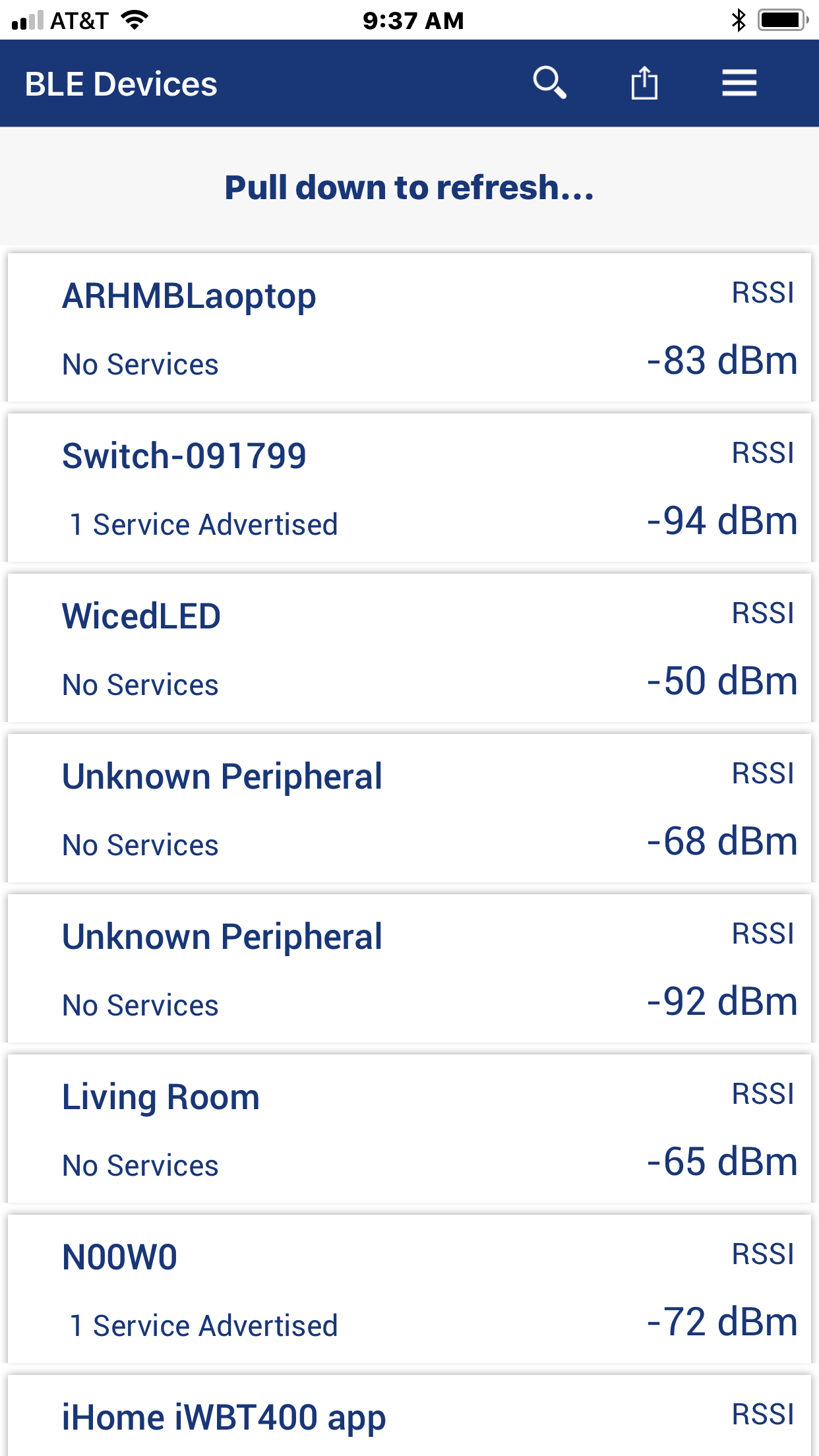
Notice that the Bluetooth Designer created a make target for me. If you are using a different platform you need to edit the target.



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



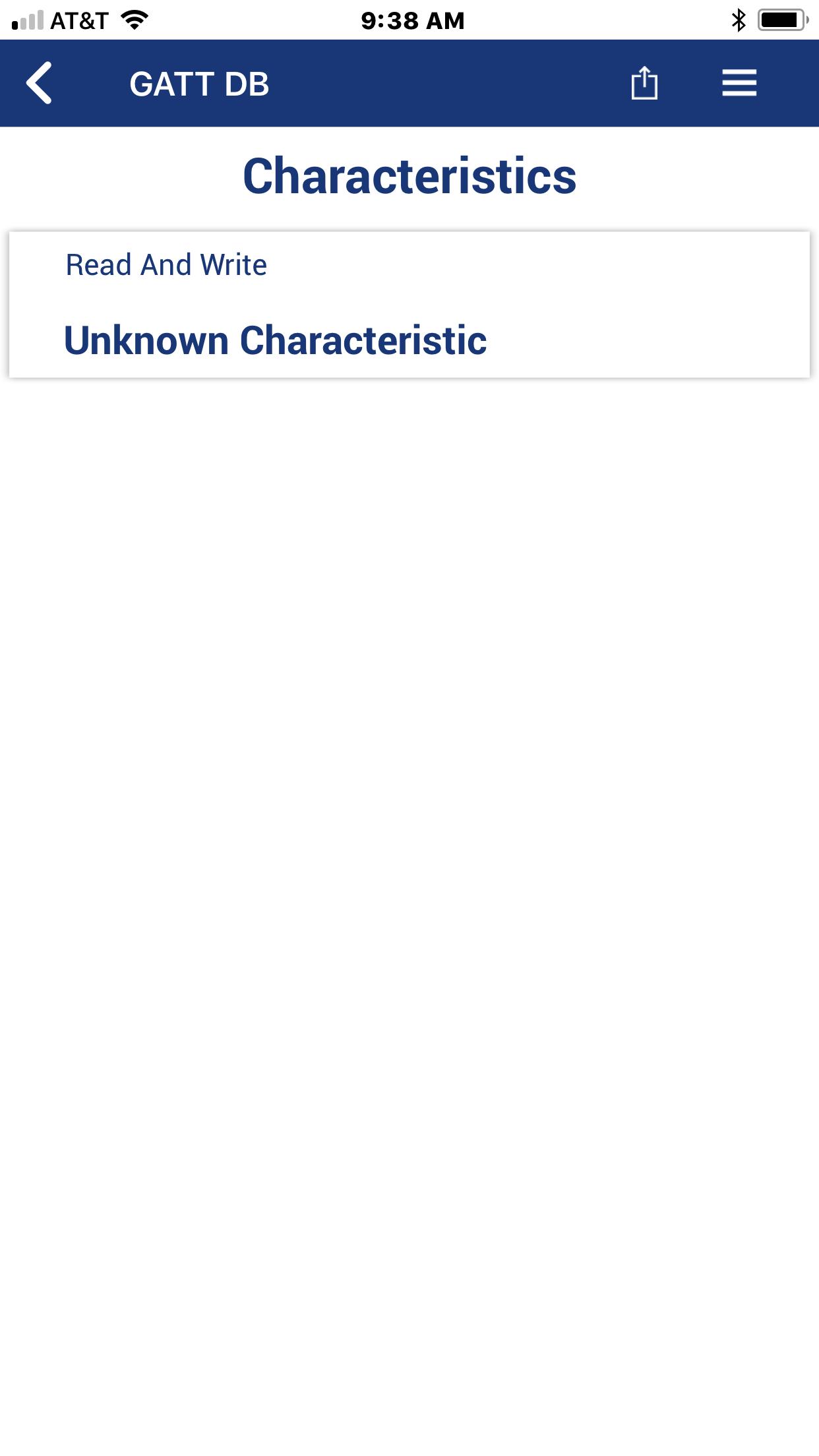
Run CySmart. When you see the “WicedLED” device. Click on it. Then CySmart will start the GATT browser.

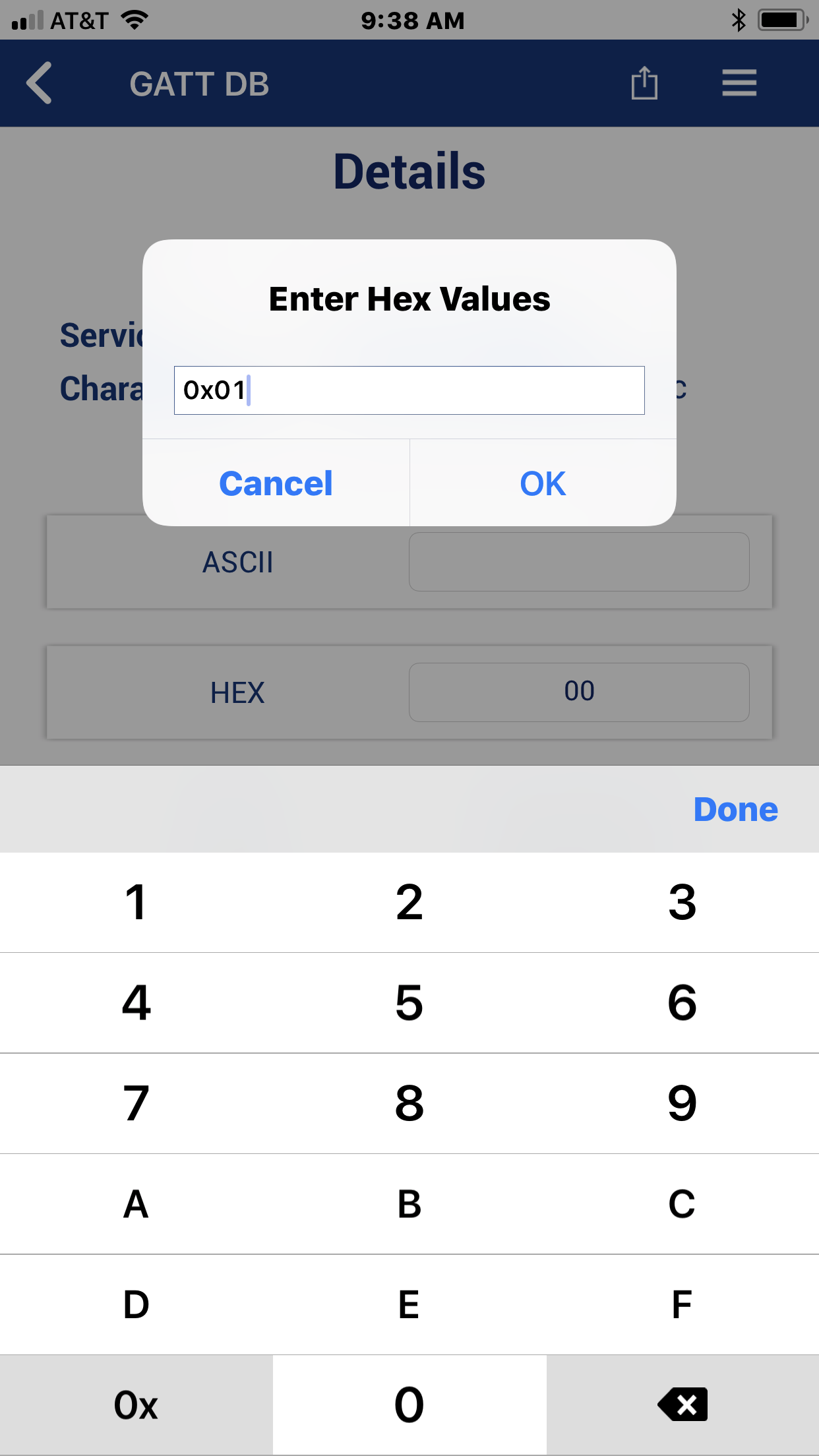
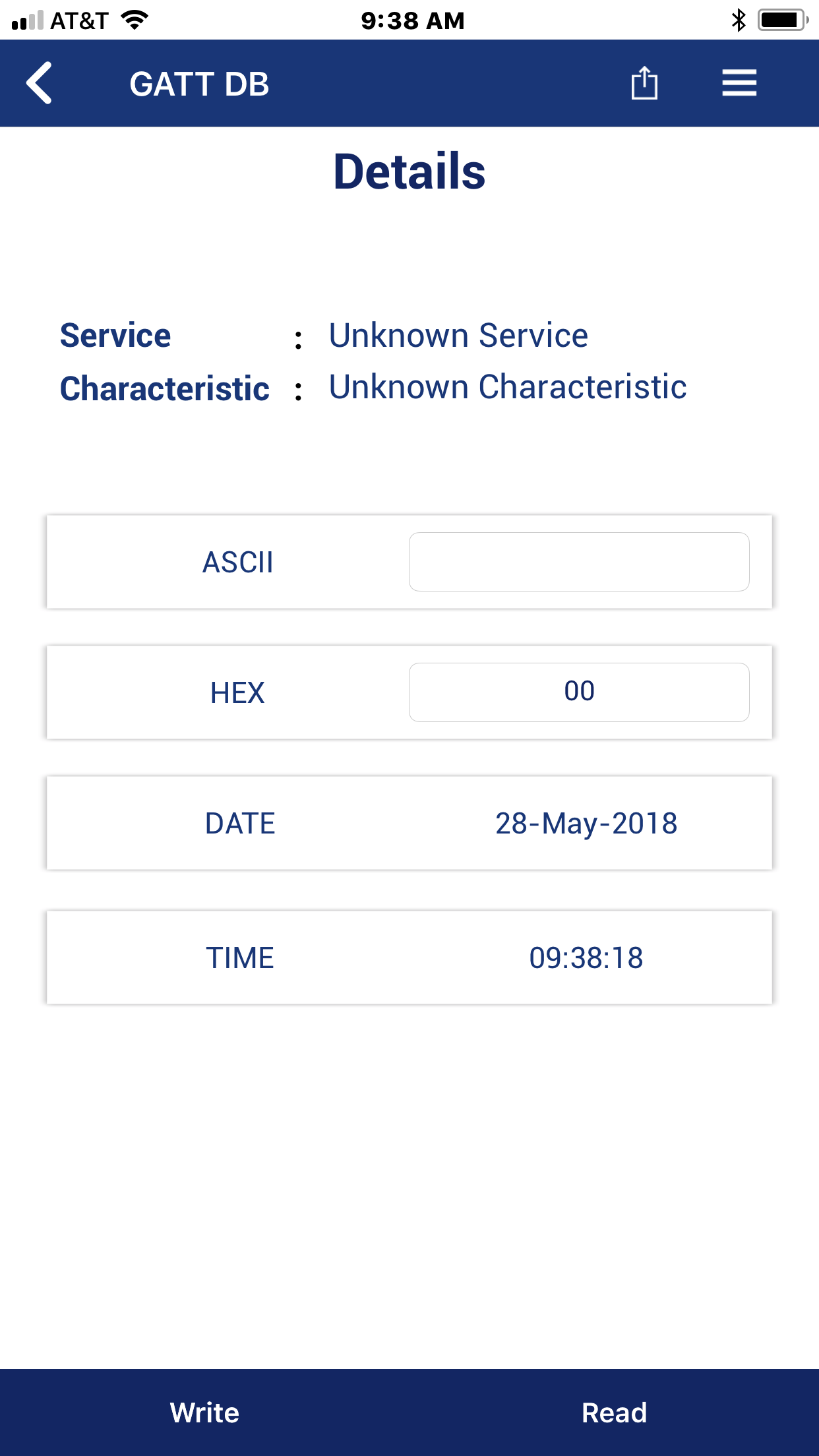
And you will see that there has been a connection and the advertising has stopped.



Back in CySmart I see an Unknown Service (which I actually know is WicedLED). Click the Service and CySmart will tell you that there is an Unknown Characteristic (which I actually know is LED1).

First, press 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.



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



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

# 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 WICED Bluetooth Stack
* Start Advertising
* Make a Connection
* Exchange Data (Read and Write)

The firmware created by WICED Bluetooth Designer mimics this flow

## Turning on the WICED Bluetooth 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 WICED Bluetooth 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. It is your job to write the management callback function which processes events

The WICED Bluetooth Stack generates the event BTM\_ENABLED\_EVT which is processed by the function \_management\_callback.

The function \_app\_init is called by the \_management\_callback function. It initializes the system and ends by calling the wiced\_bt\_start\_advertising function

## Start Advertising

The Bluetooth Stack is triggered to start advertising by the last step of the Off 🡪 On process call to wiced\_bt\_start\_advertising.

The Bluetooth Stack generates the BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT management event and calls the \_management\_callback.

The \_management\_callback 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 with a Central that is actively Scanning hearing your advertising packet and deciding to connect. It then sends you a connection request.

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

The WICED Bluetooth generates a GATT event called GATT\_CONNECTION\_STATUS\_EVT which is processed by the \_event\_handler function.

The \_event\_handler calls the function \_connect\_callback which uses the event parameter to determine if it is a connection or a disconnection. It then prints a message

The Bluetooth Stack then stops the advertising and calls the \_mangement\_callback with a management event BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT

The \_management\_callback looks determines that it is a stop of advertising, and then calls \_advertisement\_stopped, which just prints out a message.

## 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.

The Stack generates a GATT\_ATTRIBUTE\_REQUEST\_EVT and calls testwbt\_event\_handler.

The testwbt\_event\_handler determines the event is GATT\_ATTRIBUTE\_REQUEST\_EVT and calls the function \_server\_callback.

The \_server\_callback looks at the event parameter and determines that it is a GATTS\_REQ\_TYPE\_READ, it then calls the function \_read\_handler

The \_read\_handler calls the Gatt Database API get\_value to find the current value of the Characteristic.

The \_get\_value function looks through that GATT Database to find the Attribute that matches the Handle requested.

The \_get\_value function copies the value bytes out of the GATT Database into the location requested by the stack.

The \_get\_value returns a WICED\_BT\_GATT\_SUCESS, which is then returned by the \_read\_handler to the \_server\_callback.

If something bad has happened in the \_get\_value function (like the requested Handle doesn’t exist) it returns the appropriate error code i.e. WICED\_BT\_GATT\_INVALID\_HANDLE

The \_server\_callback returns status code generated by the \_get\_value function to the Bluetooth Stack

The Bluetooth Stack then either send the error code, or it send the data back to the Central.

## 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 WICED Bluetooth Stack generates the GATT event GATT\_ATTRIBUTE\_REQUEST\_EVT and calls the function \_server\_callback.

The \_server\_callback looks at the event parameter and determines that it is a GATTS\_REQ\_TYPE\_WRITE, it then calls the function \_write\_handler

The \_write\_handler calls the Gatt Database API set\_value to update the current value of the Characteristic.

The \_set\_value function looks through that GATT Database to find the Attribute that matches the Handle requested.

The \_set\_value function copies the value bytes from the Stack generated request into the GATT Database.

The \_set\_value returns a WICED\_BT\_GATT\_SUCESS, which is then returned by the \_read\_handler to the \_server\_callback.

If something bad has happened in the \_set\_value function (like the requested Handle doesn’t exist) it returns the appropriate error code i.e. WICED\_BT\_GATT\_INVALID\_HANDLE

The \_server\_callback returns status code generated by the \_set\_value function to the Bluetooth Stack

The Bluetooth Stack then either send the error code, or a write response.

# WICED GATT Database Implementation

The WICED Bluetooth Designer automatically creates a template GATT Database implementation to serve as a starting point. The implementation is generic and will work for most situations however, you can make changes to handle custom situations. When you start the WICED BT Stack by calling wiced\_bt\_stack\_init one of the parameters is a pointer to the GATT DB, meaning that the WICED BT Stack will directly access your GATT DB for some purposes.

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

And 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

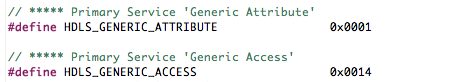
* An Array, named gatt\_database, of uint8\_t bytes that holds the handles, Types and Permissions.
* An Array of Structs which holds Handles, A Maximum and Current Length and a Pointer to the actual Value
* The Actual values as arrays of unint8\_t bytes
* Functions that serve as the API

## 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. The WICED Bluetooth Designer will automatically create Handles for you that will end up in the \_db.h file. For example:



The Service parameter is the UUID of the service, just an array of bytes. The 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)

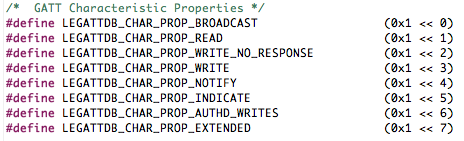
As before the handle parameter is just the 16-bit number. And as before the WICED Bluetooth Designer creates handles for the Attributes for Characteristics which will be in the form of #define HDLC\_ for example:



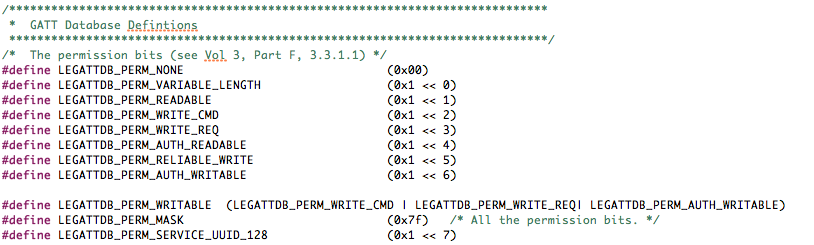
The handle\_value parameter is the Attribute handle of the value Characteristic Attribute.

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

Properties is a bit mask which sets the properties (i.e. read write etc.) The bit mask is defined in wiced\_bt\_gatt.h

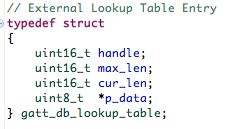


The Permission are just a bit mask that sets the Permission of an Attribute (remember Permission are on a per Attribute basis and Properties are on a per Characteristic basis). They are also defined in wiced\_bt\_gatt.h

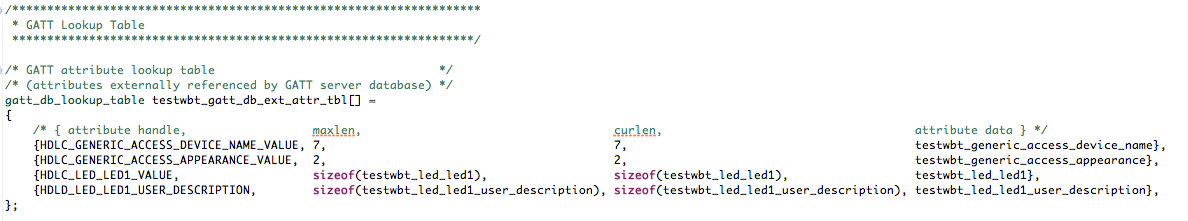


## gatt\_db\_ext\_attr\_tbl

The gatt\_database array does not contain the actual value of Attributes. To find the actual values there is an array of structures of type gatt\_db\_lookup\_table. Each structure contains a handle, a max and actual length and a pointer to the actual value.



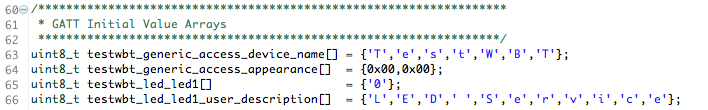
The WICED Bluetooth Designer will create this array for you automatically.



The API functions will help you search through this array to find a pointer to the actual value.

## uint8\_t Arrays for the Values

The 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 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, the LED Characteristic and the LED Characteristic Name.



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 your GATT Database, \_get\_value and \_set\_value (the function names are both prepended with the name of your app as defined in the WICED Bluetooth Designer). Here are the function prototypes from my “testwbt” application:

wiced\_bt\_gatt\_status\_t testwbt\_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 testwbt\_set\_value( uint16\_t attr\_handle, uint16\_t conn\_id, uint8\_t \*p\_val, uint16\_t len )

These two 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 the BT designer does not so this parameter is unused.
* uint8\_t \*p\_val – A pointer to the data. For a write, you should copy this data into your value, for a read you should copy your data into the location pointed to by this pointer
* (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... e.g \*p\_len = 23; // returning 23 bytes
* (write) uint16\_t len – For 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 value, or writing into the buffer for the WICED BT Stack to send back to the Client.

Both functions have a switch where you might put in custom code to do something based on which 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 BT 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.

# WICED Bluetooth Stack Events

The WICED Bluetooth Stack generates Events based on what is happening the Bluetooth world. After the event is created, the Stack will call the appropriate 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 three classes of events, Management, GATT, and GATT Server. The Bluetooth Designer will generate code to handle a bunch 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 tell you BTM\_BLE\_ADVERT\_OFF or one of the many different levels of active advertising |

## Essential GATT Events

|  |  |
| --- | --- |
| Event | Description |
| GATT\_CONNECTION\_STATUS\_EVT | When a connection is made, or disconnected. 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 |

## Essential GATT Server Events

|  |  |
| --- | --- |
| 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. |