Chapter 4: Bluetooth Low Energy (BLE)

Time: 3 Hours

At the end of this chapter you will understand the basics of BLE and how to create BLE projects on WICED devices.

4.1 BLE Introduction 2

4.2 Stack 2

4.3 Physical Layer (PHY) 3

4.4 Link Layer (LL) 3

4.5 Logical Link Control Adaptation Protocol (L2CAP) 4

4.6 Generic Access Profile (GAP) 4

4.7 Generic Attribute Profile (GATT) 4

4.8 Profiles, Services, Characteristics, and Attributes 5

4.8.1 Profiles 5

4.8.2 Services 5

4.8.3 Characteristics 6

4.8.4 Attributes 6

4.9 Attribute Protocol (ATT) 6

4.10 Security Manager (SM), Pairing and Bonding 7

4.11 CySmart 8

4.11.1 CySmart PC Application 8

4.11.2 CySmart Mobile Application 10

4.12 Using BLE in WICED Studio 11

4.12.1 Documentation 12

4.12.2 Bluetooth Stack Initialization and Callback 12

4.12.3 Advertisement Packet 20

4.12.4 GATT Database 21

4.12.5 Example Applications 21

4.12.6 WICED Bluetooth Designer 22

4.13 Advanced Topics 31

4.14 Exercise(s) 32

Exercise - 4.1 Create a BLE Advertiser 32

Exercise - 4.2 Connect using BLE 34

Exercise - 4.3 Save BLE Bonding Information 39

Exercise - 4.4 Add a Pairing Key 42

Exercise - 4.5 (Advanced) Implement a BLE Central Device ??? 44

Exercise - 4.6 (Advanced) Join a BLE Mesh Network ??? 44

4.15 Recommended Reading 45

# BLE Introduction

With the addition of Bluetooth Low Energy to the Bluetooth specification in 2010, it has become very popular in IoT devices such as smart watches, health monitors, beacons, etc. What these applications typically have in common is small batteries that are often not charged frequently. Therefore, low power is more critical than data transfer speed. Moreover, these types of devices don't require a constant connection. Rather, they can connect somewhat infrequently to send a burst of data.

The scenario described above is ideal for BLE. In fact, the way low power is achieved in BLE is not by lowering the power of the radio (i.e. the range), but rather by having the radio turned off most of the time. That is, BLE connections can stay active while only turning on the radio for a small percentage of each connection interval (e.g. a few hundred microseconds). The connection interval can be varied depending on the application from 7.5 ms to 4 seconds to trade off power and performance.

The MCU can also be put into sleep modes a large portion of the time to further reduce power.

BLE is also sometimes referred to as "Bluetooth Smart". The two terms are interchangeable. Devices that support both Classic Bluetooth and BLE (e.g. smartphones) are sometimes called "Smart Ready".

# Stack

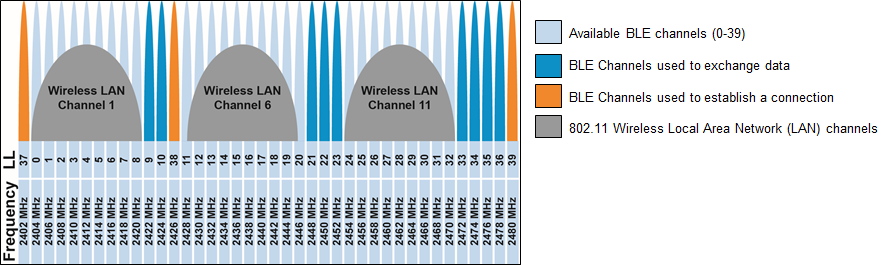
As with most complex systems, the BLE stack is broken into layers as shown below.



# Physical Layer (PHY)

BLE operates in the 2.4 GHz ISM band (2.400 – 2.480 GHz) using 40 channels with 2 MHz spacing between channels. 3 channels are used for advertising (i.e. establishing a connection) and 37 channels are used for data. Gaussian Frequency Shift Keying (GFSK) modulation is used.

Note that this is the same overall band as Bluetooth Classic and WiFi (802.11)! In order to work in the crowded 2.4 GHz ISM band, the 3 advertising channels (37, 38, and 39) are spread across the spectrum. For example, a region with 3 Wi-Fi access points operating on 3 different channels may look like this when superimposed on the BLE channels:



BLE uses adaptive frequency hopping (AFH) to avoid channels with poor signal strength or high error rates. In the example above, channels 0-8, 11-20, and 24-32 will likely be identified as channels that should be excluded from frequency hopping due to the interference from the Wi-Fi signals.

The maximum raw data transfer rate in BLE is 1 Mbps. In Bluetooth v5, the data rate can be doubled to 2 Mbps at the expense of range. Including overhead, the actual data transfer rate is ~300 Kbps in Bluetooth v4.1 and is ~800 Kbps in Bluetooth v4.2 and beyond due to the data length extension which allows larger payloads in each packet (27 bytes vs. 251 bytes). The max payload size can be different between transmit and receive to optimize application throughput.

# Link Layer (LL)

The link layer provides the methods for devices to find each other and connect. It also handles maintaining a reliable link once it has been established.

A device that is available will *Advertise* so that it can be discovered by nearby devices. The advertisement packet includes device information such as services supported and what type of connections, if any, the device will allow.

Devices that want to gather information or form connections will *Scan* for nearby devices that are advertising. Scan packets are sent out at irregular intervals to increase the chances of coinciding with a listening device's window. Once devices know about each other, the one that initiates the connection (i.e. the one that was scanning for devices) will be the *LL Master*, while the one that accepts the connection will be the *LL Slave*.

Once a connection is established, the link layer uses AES-128 encryption and 24-bit cyclic redundancy check (CRC) to guarantee a private and reliable connection. The link layer also implements AFH as described previously.

# Logical Link Control Adaptation Protocol (L2CAP)

The L2CAP layer is responsible for taking large packets of data from the upper layers and segmenting them into smaller packets for the link layer, and vice versa. The largest possible size for data packets being transmitted in BLE is called the Maximum Transmission Unit (MTU). It can be set in the range of 23 to 512 bytes.

# Generic Access Profile (GAP)

The GAP defines how devices discover each other, how they establish a connection, and how they interact with each other based on their roles. There are four GAP roles. The first two involve a connection, while the last two involve an exchange of data without a connection (i.e. advertise/scan only). They are:

|  |  |
| --- | --- |
| **GAP Role** | **Description** |
| Peripheral | A device that connects to a Central. Typically, this is an IoT device like a fitness monitor. |
| Central | A device that connects to a Peripheral. Most often, this is a smart phone or tablet. |
| Broadcaster | A device that only advertises. It may transmit useful data within the advertising packets. This may be an IoT device such as a beacon or a GPS tag. |
| Observer | A device that scans for devices and may use data from their advertising packets. |

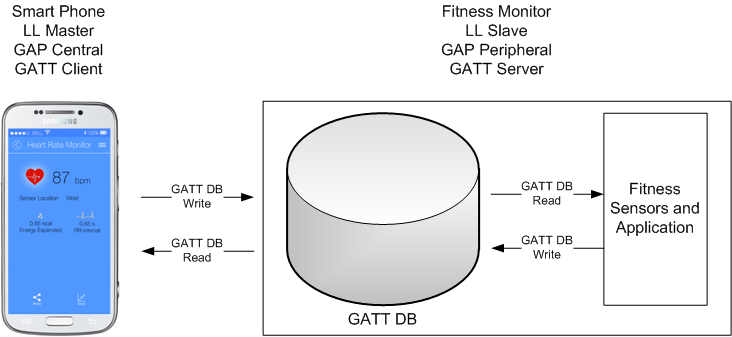
# Generic Attribute Profile (GATT)

Once a connection is made, the GATT determines how data is exchanged. There are 2 GATT roles:

|  |  |
| --- | --- |
| **GATT Role** | **Description** |
| Server | A device that contains data to be shared. Typically, this is an IoT device like a fitness monitor. |
| Client | A device that requests data from the server. Most often, this is a smart phone or tablet. |

Note: based on the above roles, the GATT server is typically a GAP peripheral while the GATT client is typically a GAP central.

Servers use a GATT Database to store data in a format defined by the Bluetooth spec. The database responds to read/write requests from server itself (e.g. when new data is available from a sensor) and from a connected client. Allowed transactions are defined when the database is setup in the server (e.g. which values the client can write/read vs. read). For example, a client may be allowed to write/read configuration settings on the server but may only be allowed to read the values of sensors.



# Profiles, Services, Characteristics, and Attributes

## Profiles

Once a GATT DB is available, how does each device know what data is stored and how it is represented? The answer is Profiles and Services (more on Services in a minute). A profile is a standard (or in some cases custom) agreement on what an application supports. It allows devices to understand what type of data is stored in the database of a device without having to do a complicated exchange of information each time a device connects. When a connection happens, devices only need to tell each other which profiles and services they support along with some basic configuration information about each service, and then they are ready to exchange data.

The Bluetooth SIG defines a set of Standard (a.k.a. Adopted) GATT Profiles. If two devices implement the same standard profile, they are guaranteed to be interoperable. Profiles specify required and optional services that must be included for an application to be compliant with the profile.

Non-standard (a.k.a. Custom) GATT Profiles are also supported by BLE and are often provided for proprietary technologies. For example, Cypress has a custom Profile for CapSense.

## Services

A profile is a collection of one or more services. A service is something that provides some related set of information. For example, the Blood Pressure Profile requires the Blood Pressure Service and Device Information Service. It may also contain optional services such as the Battery Service. All profiles require a Generic Access Service and a Generic Attribute Service.

Each service has a UUID – either one assigned by the SIG, or a custom one for custom services. In fact, advertisement packets may contain information about supported services for a device.

## Characteristics

A service is a collection of characteristics. The characteristics are different items that are all related to the service. For example, the Blood Pressure Service contains three characteristics: Blood Pressure Measurement, Intermediate Cuff Pressure, and Blood Pressure Feature. Each of these is related to blood pressure measurement but will contain different information.

Like profiles and services, characteristics also have UUIDs.

## Attributes

A characteristic is collection of attributes. An attribute specifies the format of the data and contains the data itself as a series of fields. For example, the Blood Pressure Measurement Characteristic contains an attribute structure with the fields Flags, Measurement Compound Value, etc. The exact fields included, and the units used to represent the data in each field are specified by the Flags field. In that way, by reading the flags, both devices know what data is in the GATT DB and how it is represented.

# Attribute Protocol (ATT)

The ATT defines the rules for BLE communication. It enables GATT clients to find and access attributes on GATT servers using six operations: Requests, Responses, Commands, Notifications, Indications, and Confirmations. Examples of each operation:

GATT Client reads data:

1. Client sends a Request for the data.
2. Server sends a Response with the data.

GATT Client writes a value (such as registering for notifications or indications):

1. Client sends a Command to the server.
2. Server receives the Command.

GATT Server sends a notification of new data (assumes client previously asked for notifications):

1. Server sends a Notification.
2. Client receives the Notification.

GATT Server sends an indication of new data (assumes client previously asked for indications):

1. Server sends an Indication.
2. Client receives the Indication and responds with a Confirmation.
3. Server receives the Confirmation.

# Security Manager (SM), Pairing and Bonding

BLE has two security modes, and several levels in each mode. They are:

|  |  |  |  |
| --- | --- | --- | --- |
| **Security** | **Level 1** | **Level 2** | **Level 3** |
| Mode 1 | No security | Unauthenticated  Encrypted | Authenticated  Encrypted |
| Mode 2 | Unauthenticated  Data Signed | Authenticated  Data Signed | N/A |

Authentication is the process of identifying a device and deciding whether a connection will be allowed. It can be done in one of several ways depending on the capabilities of the devices. The possible capabilities are:

1. No Input, No Output
2. Display Only
3. Display
4. Display: Yes/No
5. Keyboard Only

Need to understand these better and explain the possible options depending on the capabilities. What is display only vs. display yes/no vs display?

Need to understand/explain what data signed means. How is this different from encryption?

Need details on authentication and encryption schemes.

Once two BLE devices have established a connection (including authentication and key exchange if necessary), they are considered Paired. If the authentication information and keys are stored in memory by both devices, then the devices are Bonded. Devices that are bonded can connect in the future without going through the pairing process again.

The whole process looks like this:



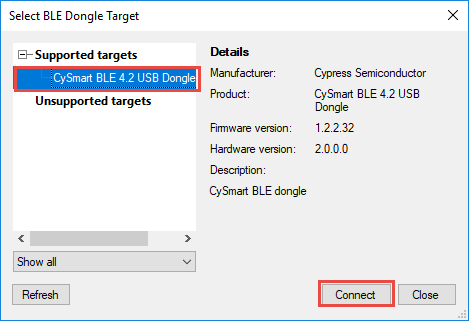
In Bluetooth v4.2, privacy 1.2 was introduced. This involves using a 48-bit resolvable private address (RPA) that can be changed frequently (every 1 second) to prevent tracking. Only peer devices that have the 128-bit identity resolving key (IRK) of a BLE device can connect to it.

# CySmart

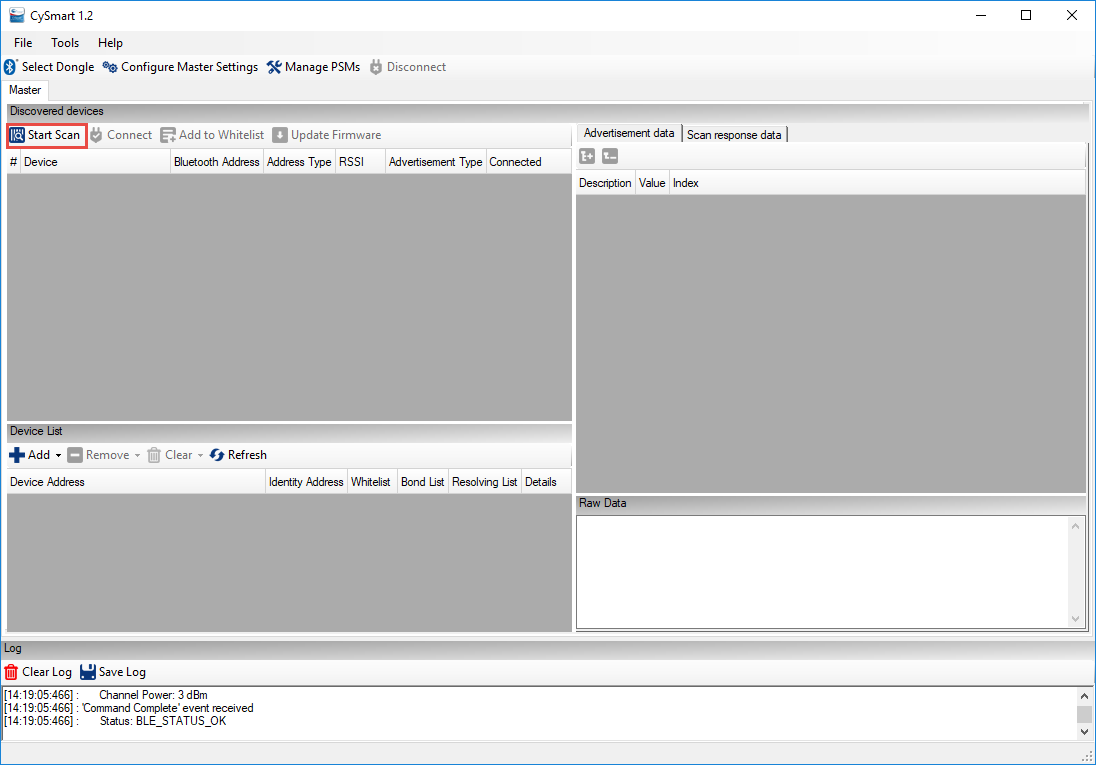
Cypress provides a PC and mobile device application (Android and iOS) called CySmart which can be used to scan, connect, and interact with services, characteristics, and attributes of BLE devices.

## CySmart PC Application

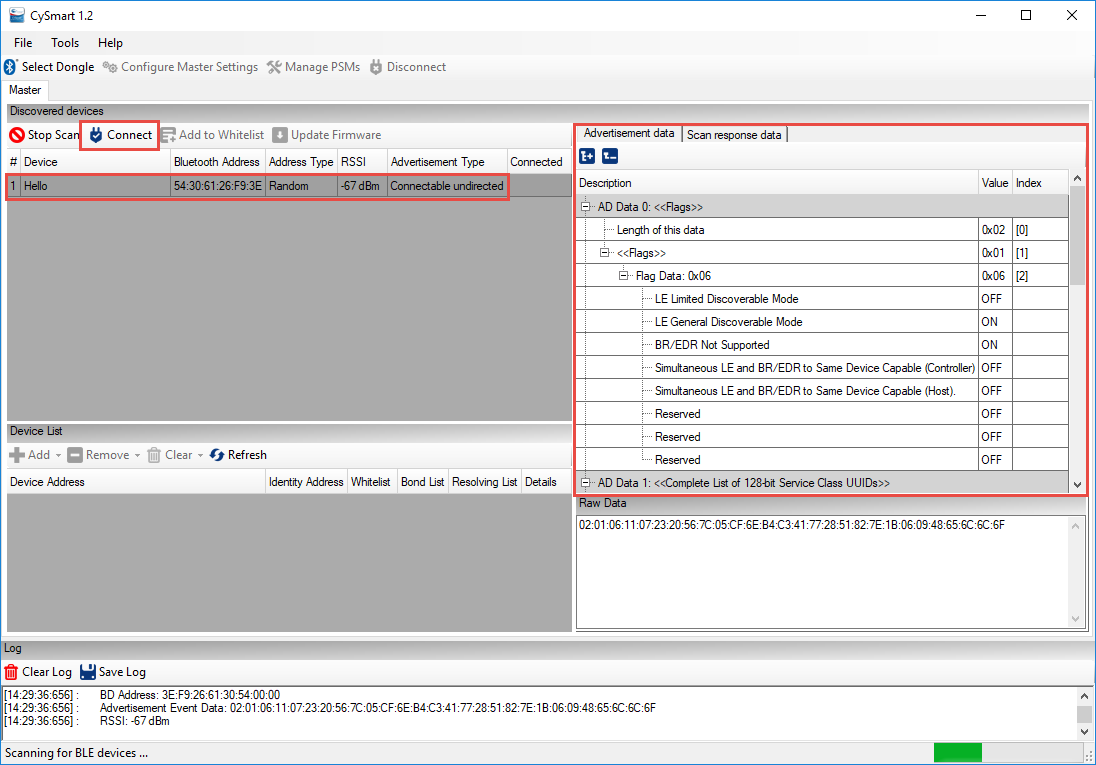
To use the CySmart PC Application, a CY5670 CySmart USB Dongle is required. When CySmart is started, it will search for supported targets and will display the results. Select the dongle that you want to use and click on "Connect".



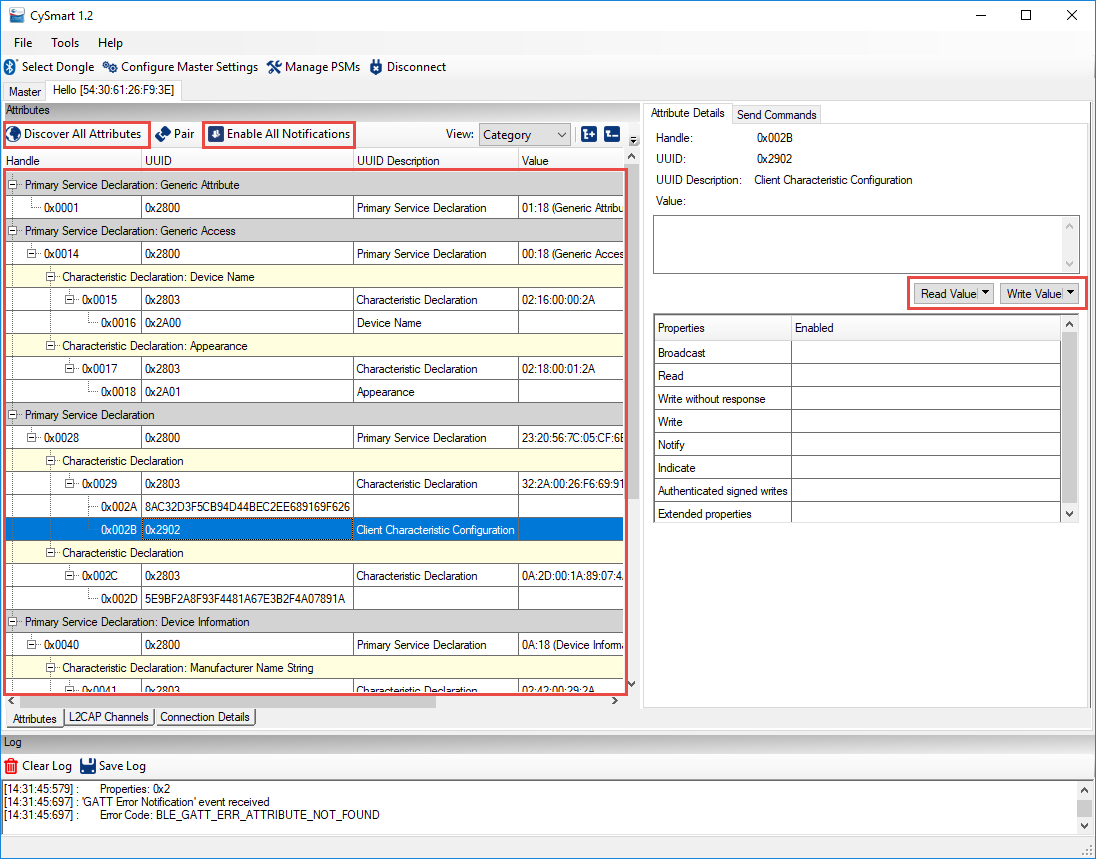
Once a dongle is selected, the main window will open as shown below. Click on "Start Scan" to search for advertising BLE devices.



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.



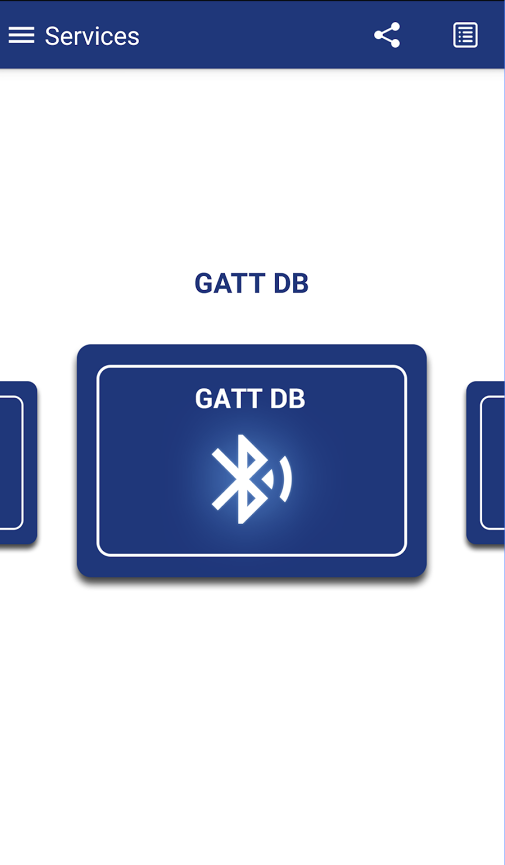
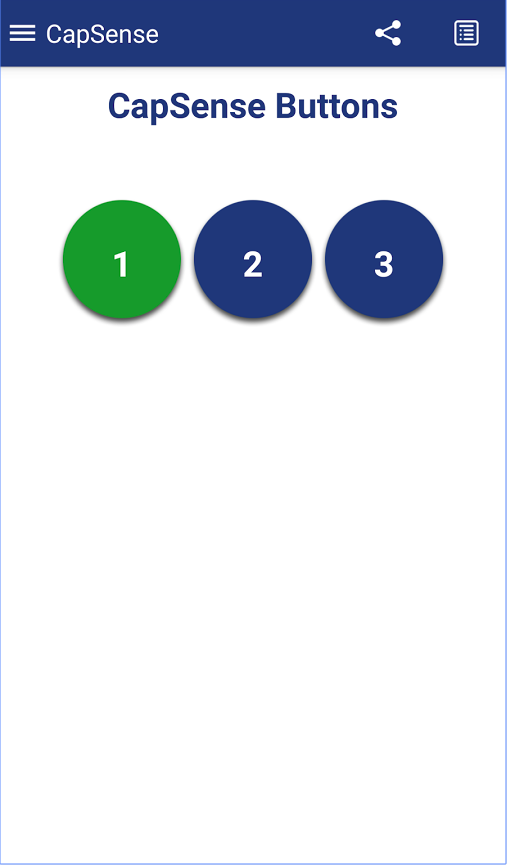
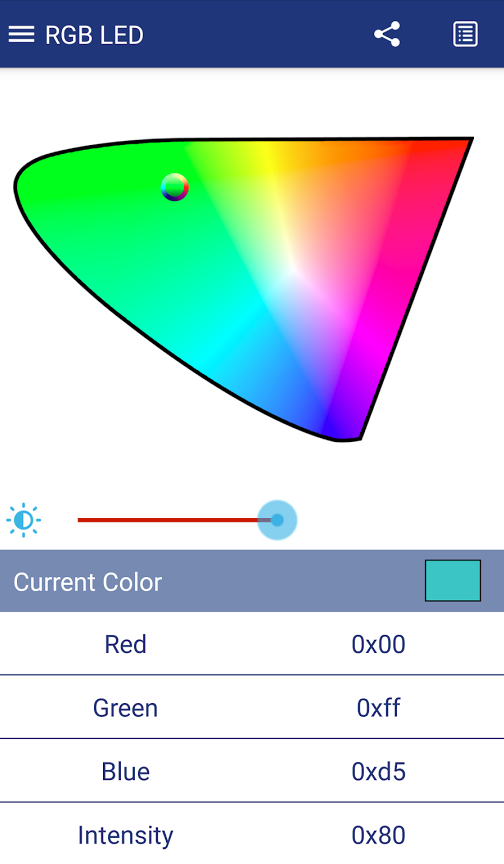
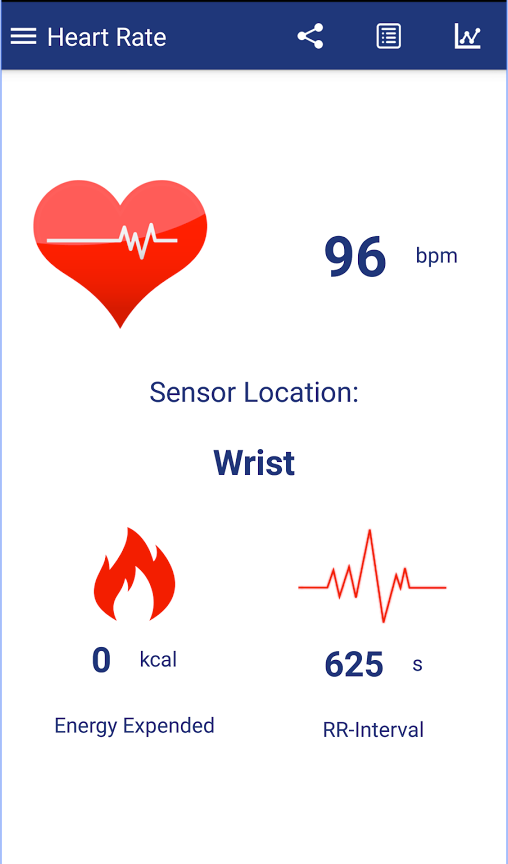
The complete User Guide for the CySmart PC application can be opened in the tool under *Help -> Help Topics*. It can also be found on the CySmart website at:

<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.

## 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.

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:

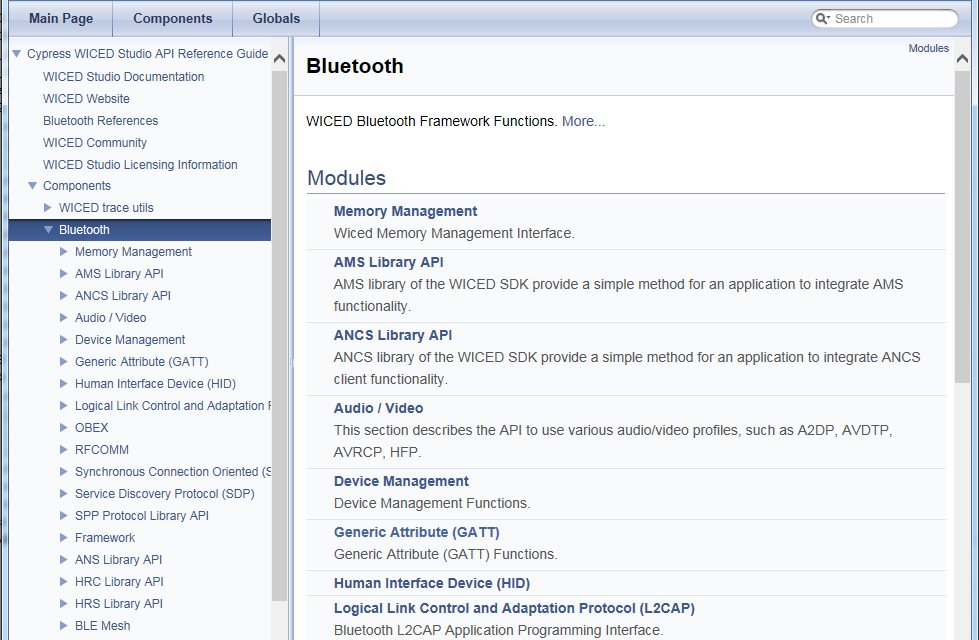
<http://www.cypress.com/documentation/software-and-drivers/cypresss-custom-ble-profiles-and-services>

# Using BLE in WICED Studio

Now that you have learned the basics of BLE, let's look at how to implement the firmware in WICED Studio. We will cover the overall architecture first and then show you some tools and techniques that can help simplify project creation.

## Documentation

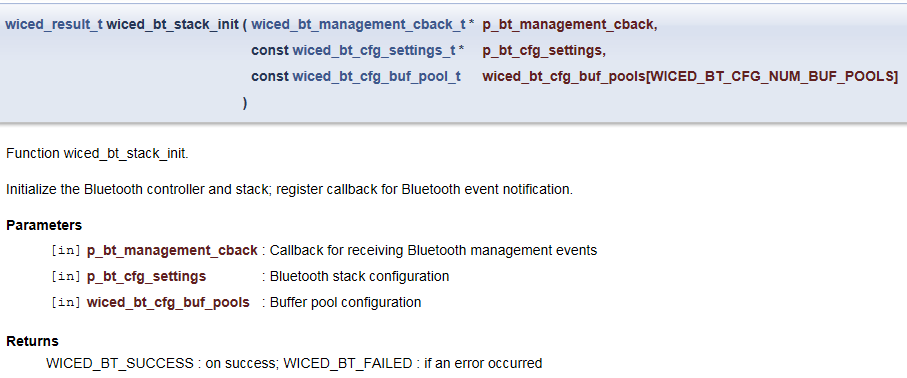
The relevant documentation for Bluetooth functions are in the WICED SDK documentation under Components🡪Bluetooth.



## Bluetooth Stack Initialization and Callback

As you have seen in prior exercises, Bluetooth applications start at *application\_start()* which does initialization that is required before starting the Bluetooth stack and then starts the stack using *wiced\_bt\_stack\_init()*.

Documentation for this function can be found in Components -> Bluetooth -> Framework

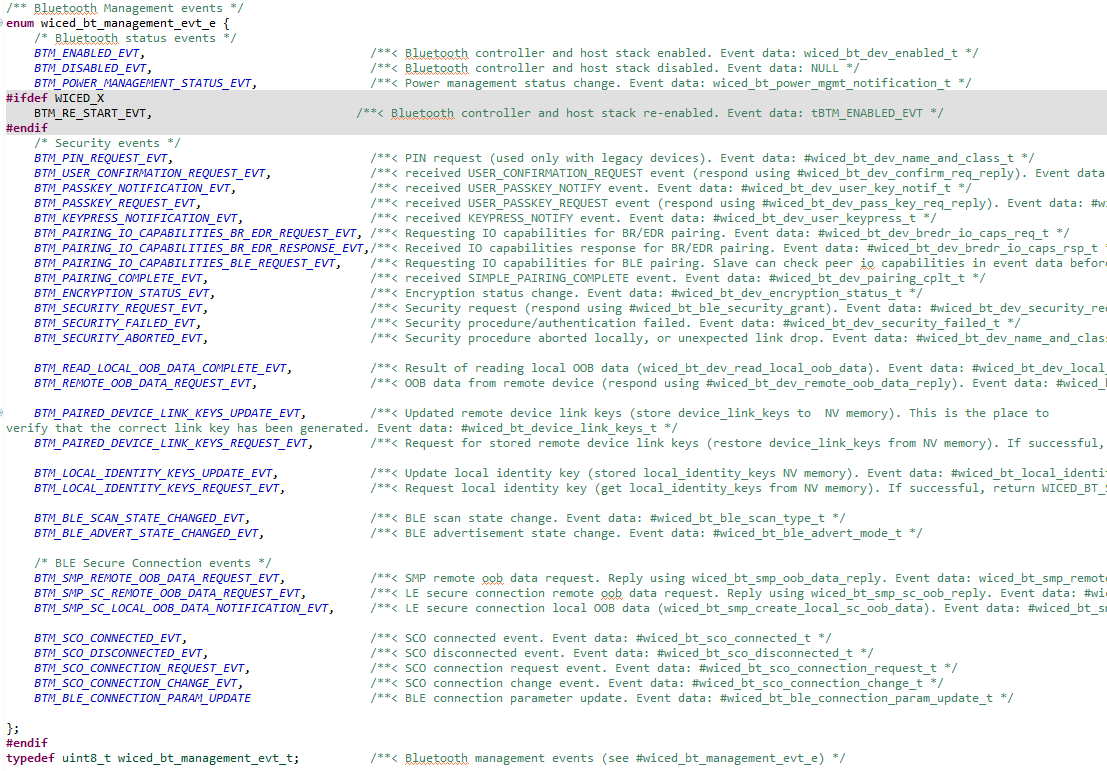


As you can see, there are three arguments. Each of these is discussed separately below.

### Callback Function

The first argument to *wiced\_bt\_stack\_init()* is a pointer to the callback function that is called by the Bluetooth stack for various events. This function typically handles the bulk of the application's functionality.

The list of callback events can be found in wiced\_bt\_dev.h and is shown here:



Applications will not typically implement all callback events. The events required will depend on the application.

All applications should have the BTM\_ENABLED\_EVT event implemented. This event occurs after the *wiced\_bt\_stack\_init()* function completes stack initialization and is the application's notification that it can continue with initialization that needs to be done after the stack has started and then start normal operation.

For a typical BLE application that connects using a secure (encrypted) link but does NOT store bonding information in NVRAM and does NOT require a passkey, the order of callback events will look like the following:

|  |  |  |
| --- | --- | --- |
| **Activity** | **Callback Event Name** | **Reason** |
| Powerup | BTM\_LOCAL\_IDENTITY\_KEYS\_REQUEST\_EVT | At initialization, the BLE stack looks to see if there are local encryption keys available. If bonding information is not stored in NVRAM then this state does not need to be implemented. |
| BTM\_ENABLED\_EVT | This occurs once the BLE stack has completed initialization. Typically, you will start up the rest of your application here. |
| BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT | This occurs when you enable advertisements. You will see a return value of 3 for fast advertisements. After a timeout, you may see this again with a return value of 4 for slow advertisements. Eventually the state changes to 0 (off) if there have been no connections, giving you a chance to save power. |
| Connect | BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT | Once the connection happens, the stack stops advertisements which will result in this event. You will see a return value of 0 which means advertisements have stopped. |
| Pair | BTM\_SECURITY\_REQUEST\_EVT | The occurs when the client requests a secure connection. When this event happens, you need to call wiced\_bt\_ble\_security\_grant() to allow a secure connection to be established. |
| BTM\_PAIRING\_IO\_CAPABILITIES\_BLE\_REQUEST\_EVT | This occurs when the client asks what type of capability your device has that will allow validation of the connection (e.g. screen, keyboard, etc.). You need to set the appropriate values when this event happens. |
| BTM\_ENCRYPTION\_STATUS\_EVT | This occurs when the secure link has been established. |
| BTM\_PAIRED\_DEVICE\_LINK\_KEYS\_UPDATE\_EVT | This event is used so that you can store the paired devices keys if you are storing bonding information. If not, then this state does not need to be implemented. |
| BTM\_PAIRING\_COMPLETE\_EVT | This event indicates that pairing has been completed successfully. |
| Disconnect | BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT | Upon a disconnect, the firmware will get a GATT event handler callback for the GATT\_CONNECTION\_STATUS\_EVENT (more on this later). At that time, it is the user's responsibility to determine if advertising should be re-started. If it is restarted, then you will get a BLE stack callback once advertisements have restarted with a return value of 3 (fast advertising) or 4 (slow advertising). |

If bonding information is stored to NVRAM, the event sequence will look like the following. The sequence is shown for three cases (each shaded differently):

1. First-time connection before bonding information is saved
2. Connection after bonding information has been saved for disconnect/re-connect without resetting the kit between connections.
3. Connection after bonding information has been saved for disconnect/reset/re-connect.

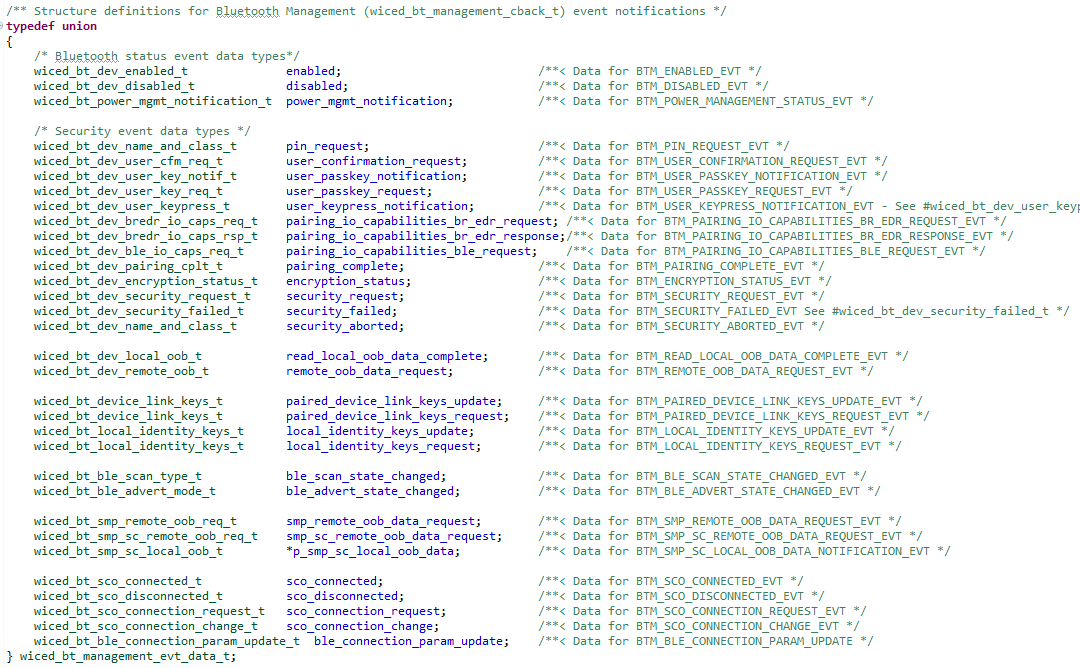
In the reconnect cases, you can see that the pairing sequence is greatly reduced since keys are already available.

|  |  |  |
| --- | --- | --- |
| **Activity** | **Callback Event Name** | **Reason** |
| 1st Powerup | BTM\_LOCAL\_IDENTITY\_KEYS\_REQUEST\_EVT | When this event occurs, the firmware needs to load the local keys from NVRAM. If keys have not been previously saved for the device, then this will not return valid keys. |
| BTM\_ENABLED\_EVT | This occurs once the BLE stack has completed initialization. Typically, you will start up the rest of your application here.  During this event, the firmware needs to load keys (which also includes the BD\_ADDR) for a previously bonded device from NVRAM and then call *wiced\_bt\_dev\_add\_device\_to\_address\_resolution\_db*() to allow connecting to an bonded device. If a device has not been previously bonded, this will return values of all 0. |
| BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT | This occurs when you enable advertisements. You will see a return value of 3 for fast advertisements. After a timeout, you may see this again with a return value of 4 for slow advertisements. Eventually the state changes to 0 (off) if there have been no connections, giving you a chance to save power. |
| BTM\_LOCAL\_IDENTITY\_KEYS\_UPDATE\_EVT | This event is called if reading of the keys from NVRAM failed (i.e. the return value from BTM\_LOCAL\_IDENTITY\_KEYS\_REQUEST\_EVT was not 0). During this event, the local device keys must be saved to NVRAM. |
| BTM\_LOCAL\_IDENTITY\_KEYS\_UPDATE\_EVT | Not sure why this is called twice…the keys are different for each time – public and random address maybe? |
| 1st Connect | BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT | Once the connection happens, the stack stops advertisements which will result in this event. You will see a return value of 0 which means advertisements have stopped. |
| 1st Pair | BTM\_SECURITY\_REQUEST\_EVT | The occurs when the client requests a secure connection. When this event happens, you need to call wiced\_bt\_ble\_security\_grant() to allow a secure connection to be established. |
| BTM\_PAIRING\_IO\_CAPABILITIES\_BLE\_REQUEST\_EVT | This occurs when the client asks what type of capability your device has that will allow validation of the connection (e.g. screen, keyboard, etc.). You need to set the appropriate values when this event happens. |
| BTM\_PASSKEY\_NOTIFICATION\_EVT | This event only occurs if the IO capabilities are set such that your device has the capability to display a value, such as BTM\_IO\_CAPABILITIES\_DISPLAY\_ONLY. In this event, the firmware should display the passkey so that it can be entered on the client to validate the connection. |
| BTM\_ENCRYPTION\_STATUS\_EVT | This occurs when the secure link has been established. Previously saved information such as paired device BD\_ADDR and notify settings is read. If no device has been previously bonded, this will return all 0's. |
| BTM\_PAIRED\_DEVICE\_LINK\_KEYS\_UPDATE\_EVT | During this event, the firmware needs to store the keys of the paired device (including the BD\_ADDR) into NVRAM so that they are available for the next time the devices connect. |
| BTM\_PAIRING\_COMPLETE\_EVT | This event indicates that pairing has been completed successfully.  Information about the paired device such as its BT\_ADDR should be saved in NVRAM at this point. You may also initialize other state information to be saved such as notify settings. |
| Disconnect | BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT | Upon a disconnect, the firmware will get a GATT event handler callback for the GATT\_CONNECTION\_STATUS\_EVENT (more on this later). At that time, it is the user's responsibility to determine if advertising should be re-started. If it is restarted, then you will get a BLE stack callback once advertisements have restarted with a return value of 3 (fast advertising) or 4 (slow advertising). |
| Re-Connect | BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT | Advertising off. |
| Re-Pair | BTM\_ENCRYPTION\_STATUS\_EVT | In this state, the firmware reads the state of the server from NVRAM. For example, the BD\_ADDR of the paired device and the saved state of any notify settings may be read.  Since the paired device BD\_ADDR and keys were already available, no other steps are needed to complete pairing. |
| Disconnect | BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT | Advertising on. |
| Reset | BTM\_LOCAL\_IDENTITY\_KEYS\_REQUEST\_EVT | Local keys are loaded from NVRAM. |
| BTM\_ENABLED\_EVT | Stack is enabled. Paired device keys (including the BD\_ADDR) are loaded from NVRAM and the device is added to the address resolution database. |
| BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT | Advertising on. |
| Re-Connect | BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT | Advertising off. |
| Re-Pair | BTM\_PAIRED\_DEVICE\_LINK\_KEYS\_REQUEST\_EVT | Since we are connecting to a known device (because it is in the address resolution database), this event is called by the stack so that the firmware can load the paired device's keys from NVRAM. |
| BTM\_ENCRYPTION\_STATUS\_EVT | In this state, the firmware reads the state of the server from NVRAM. For example, the BD\_ADDR of the paired device and the saved state of any notify settings may be read.  Since the paired device BD\_ADDR and keys were already available in NVRAM, no other steps are needed to complete pairing. |
| Disconnect | BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT | Advertising on. |

Should we talk in detail about the different types of connection, security, and authentication? For example, the pairing\_io\_capabilities event has lots of options for security and keys. Discuss these?

Need to discuss NVRAM functions such as how to read/write and what to read/write? Where does this belong?

In addition to the event itself, the callback function provides data that goes along with that event (if any). The type of the data provided depends on the event that caused the callback. The list of event data types can be found in wiced\_bt\_dev.h and is shown here:



Put examples of using the data for read/write nvram, etc.?

### Stack Configuration Settings

The second argument to *wiced\_bt\_stack\_init()* is a pointer to configuration settings for the Bluetooth stack. The configuration is usually kept in a separate source file called *wiced\_bt\_cfg.c*. This file is specified in the application's makefile. This file can (and should) be created by copying an existing file from another project or by using the *WICED Bluetooth Designer* (more on that later).

Add details on the stack configuration here? How much detail?

### Buffer Pool Settings

The third and final argument to *wiced\_bt\_stack\_init()* is an array of configuration settings for the buffer pools. Like the stack configuration, this is also usually specified in the wiced\_bt\_cfg.c file and will be created the same way (i.e. copying from another project or using *WICED Bluetooth Designer*).

Should have more detailed explanation of the buffer pools?

## Advertisement Packet

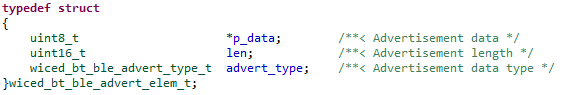
Lots of different information can be sent in an advertisement packet (e.g. device name, device appearance, primary service UUID etc.), but the total payload size is limited to a maximum of 31 bytes. The first element sent is the flags which is required. Each element (including the flags) uses the first byte to specify its payload length, the second byte to specify the type of data that follows, and the remaining bytes for the actual data.

For example, if you want to send flags, a 128-bit service UUID, and an 8-character device name, you would have:

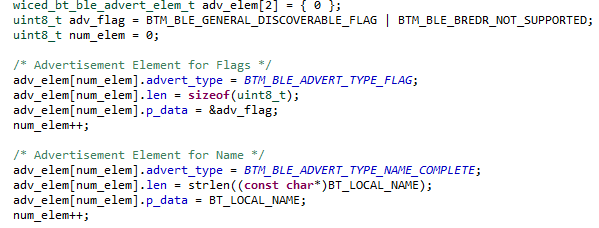
* Flags: 1-byte size + 1-byte Flag type + 1-byte Flag data = 3 bytes
* UUID: 1-byte size + 1-byte UUID type + 16-bytes for the 128-bit UUID data = 18 bytes
* Name: 1-byte size + 1-byte Name type + 8-byte Name data = 10 bytes
* Total = 3 + 18 + 10 = 31 bytes

You can send a shortened part of the name if necessary (i.e. *BTM\_BLE\_ADVERT\_TYPE\_NAME\_SHORT* instead of *BTM\_BLE\_ADVERT\_TYPE\_NAME\_COMPLETE*).

The advertisement data is set by creating an array of type *wiced\_ble\_advert\_elem\_t*. Each entry in the array is a structure that contains the type of the entry, the size of the entry, and a pointer to the data:



For example, a two-entry advertisement packet with Flags (for a BLE only device that is generally discoverable) and the name of the device would look like this:



Once the array is populated, the function *wiced\_bt\_ble\_set\_raw\_advertisement\_data* is called with the number of advertisement packet entries and a pointer to the array like this:



Finally, advertisements are started by calling *wiced\_bt\_start\_advertisements*:



## GATT Database

The GATT database is set up as a uint8\_t array containing all the Service and Characteristic information such as UUIDs and permissions.

Need explanation of the GATT callback function(s).

Need to add GATT details here. What functions are needed? Does the user need to validate writes are allowed before writing the database? Does the user need to send notifications/indications and look for indication confirmation – if not does the user need to re-send (timeout?)

Describe wiced\_bt\_gatt\_send\_notification() and how to use it. Describe sending indications too.

## Example Applications

The WICED Studio SDK contains a wealth of example snip and demo applications to illustrate how to use various BLE features. It is always recommended to start a new application based on one of the provided examples or by using *WICED Bluetooth Designer* (discussed in the next section) to simplify and speed up development.

Some of the example applications provided in the WICED Studio SDK are described below.

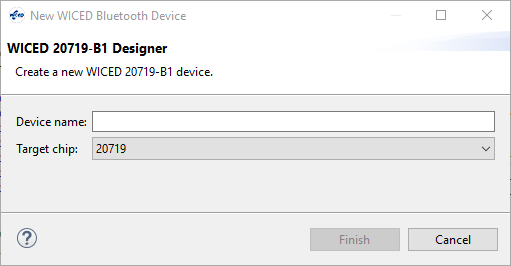
|  |  |  |  |
| --- | --- | --- | --- |
| **Application Name** | **Connection/Pairing**  **Method** | **Services** | **Description** |
| snip.ble.ams |  |  |  |
| snip.ble.anc |  |  |  |
| snip.ble.ancs |  |  |  |
| snip.ble.ans |  |  |  |
| snip.ble.bac |  |  |  |
| snip.ble.bas |  |  |  |
| snip.ble.eddystone |  |  |  |
| snip.ble.env\_sensing\_temp |  |  |  |
| snip.ble.gatt\_db |  |  |  |
| snip.ble.hrc |  |  |  |
| snip.ble.hrs |  |  |  |
| snip.ble.ibeacon |  | ` |  |
| snip.ble.multi\_beacon |  |  |  |
| snip.ble.mybeacon |  |  |  |
| demo.hello\_cleint |  |  |  |
| demo.hello\_sensor |  |  |  |

## WICED Bluetooth Designer

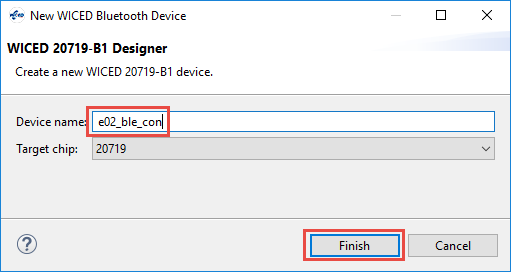
WICED Bluetooth Designer is a utility that helps set up a new Bluetooth application for Classic, BLE, or a combination of both. It creates C source files including a top-level file, a wiced\_bt\_cfg.c file, a makefile and a Make Target for the application. The top-level source file will contain initialization code, the Bluetooth callback function, and other necessary functionality based on the options chosen in the tool. Other C and header files may be created depending on the options. For example, it will create C and header files for the GATT database if the project has one.

A .wic file is created which contains the WICED Bluetooth Designer configuration. Open that file if you want to re-run the tool again but keep in mind that previous files will be over-written (after a backup copy is made) when you re-generate the code.

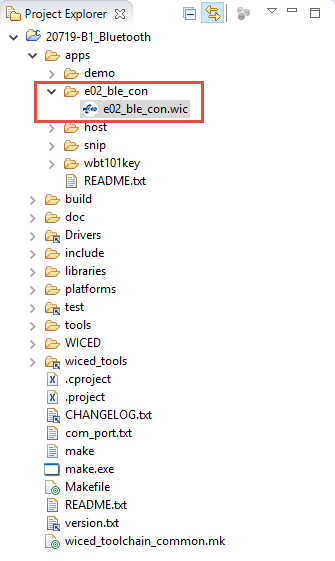
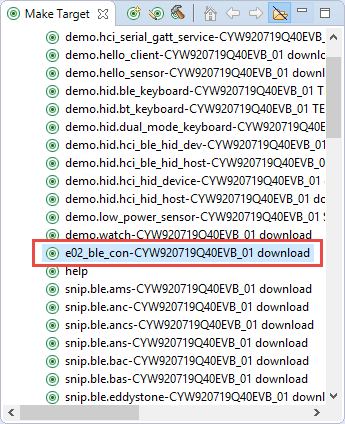
To run the tool, go to "*File -> New -> WICED Bluetooth Designer*". The first window to appear will look like this:



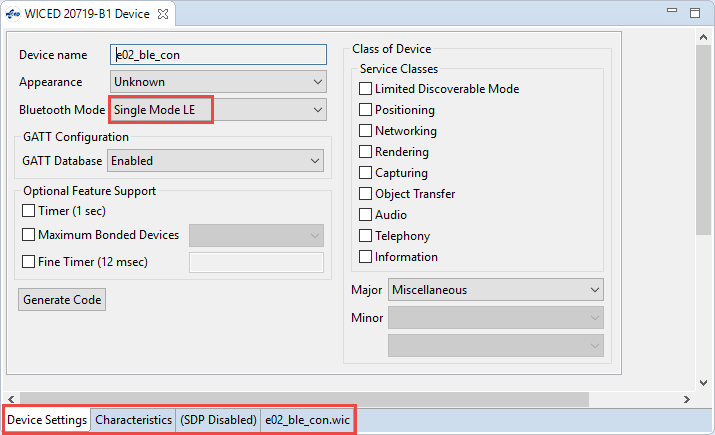
Enter a *Device name* and click "Finish". The name entered will be used for the name of the project as well as the name of the Bluetooth device. Either or both names can be changed later, but they both use this name to start out.



A folder with the name of the project is created under the *apps* folder containing the .wic file and a Make Target for the project is created. You will have to update the platform name in the Make Target if you are using a different platform such as a base board and shield combination. The project can be moved once the tool has generated the code. For example, it could be moved down two levels to a folder called *wbt101\ch04*. If you move the project, remember to edit the make target to reflect the new location.

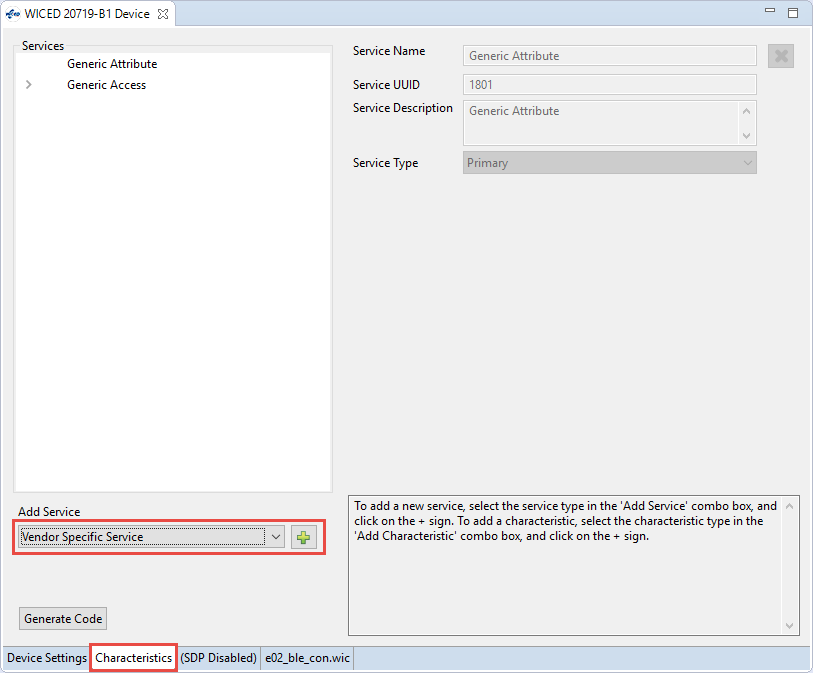
 

The center panel will show the .wic file as a configuration window with tabs along the bottom as shown here:

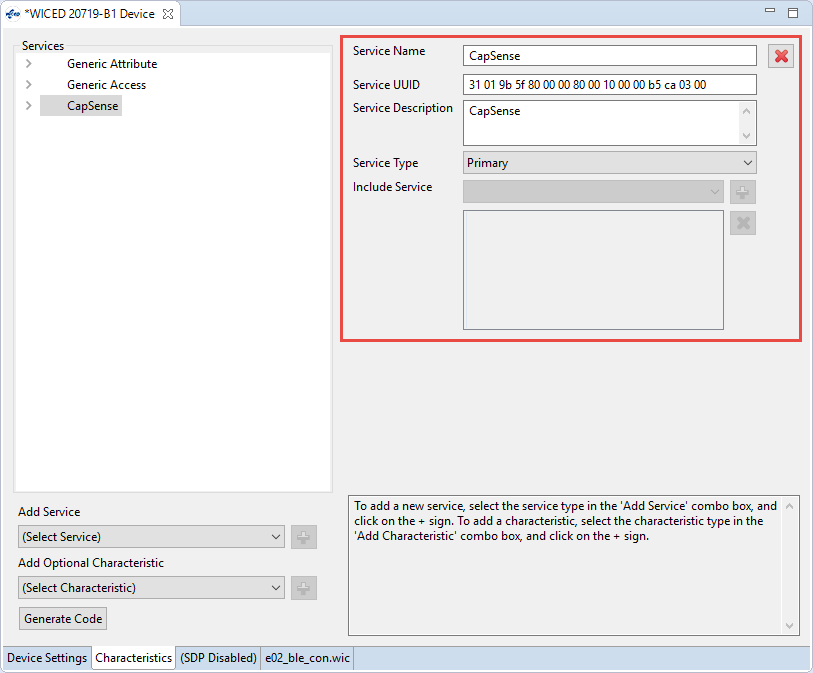


In this *Device Settings* tab you specify the device Appearance if desired, select the type of project (BLE only, Classic BR/EDR only, or dual mode), enable or disable the GATT database, enable or disable other features, etc.

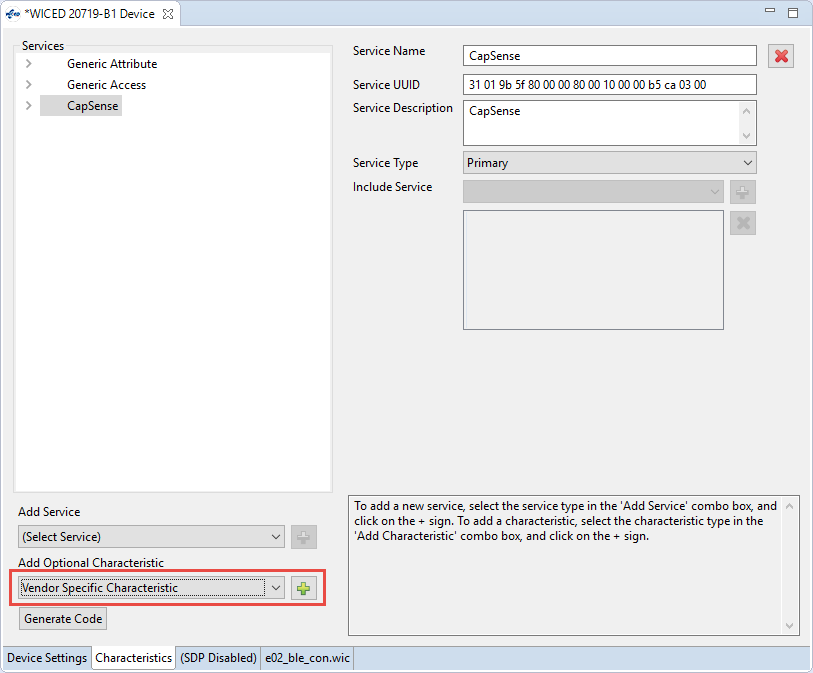
If the GATT database is enabled, the Characteristics tab will be available. On that tab, the two required services (Generic Attribute and Generic Access) will already be included. You can add and configure additional services and characteristics depending on the requirements of your project. To add a service, select the service type from the drop-down menu and click the "+" sign.



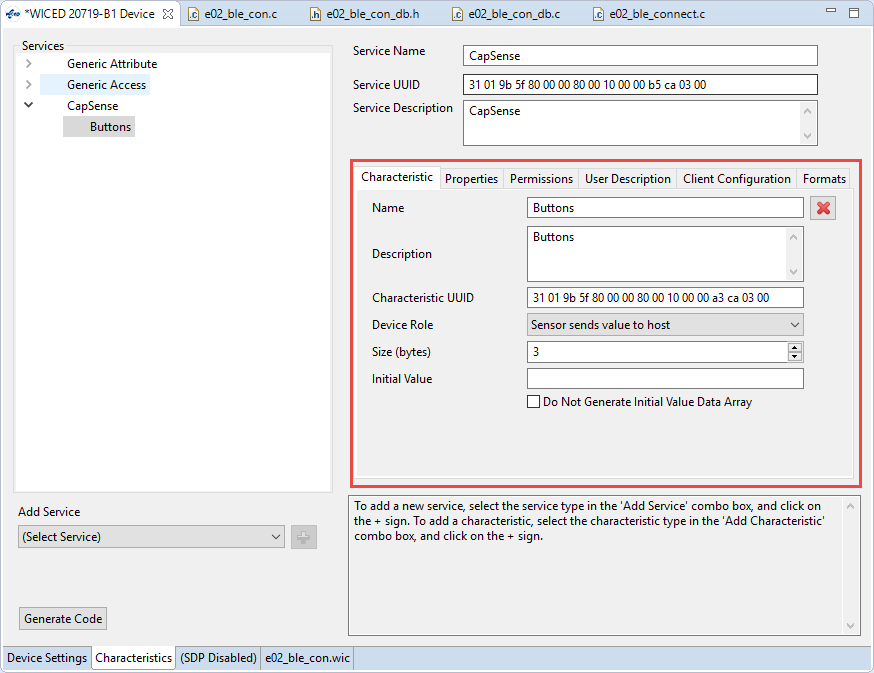
Once you click the "+" sign, you will be able to enter the Service Name, UUID, and Description. For example, the following shows the Cypress custom CapSense service.



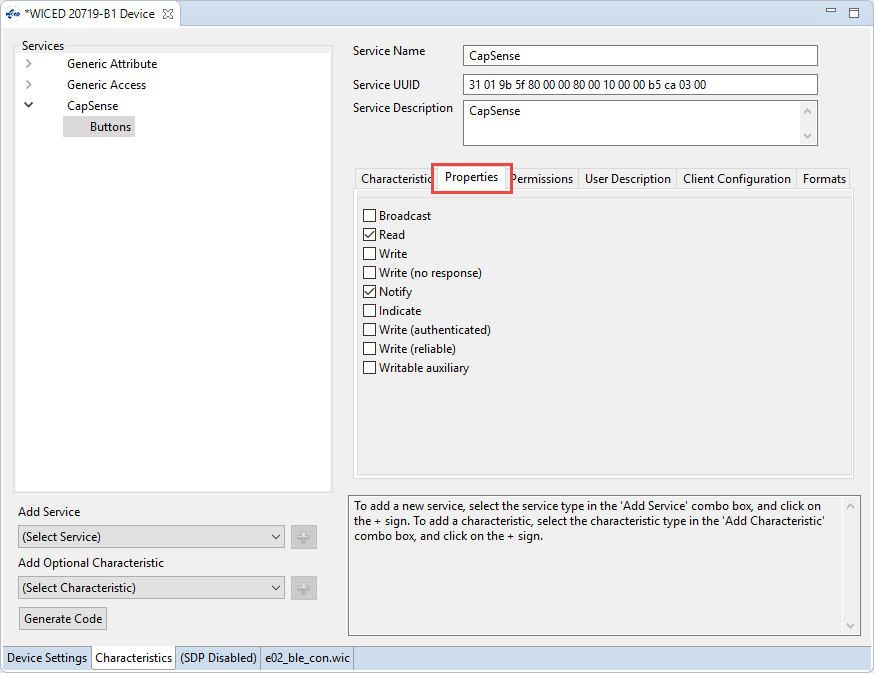
When the service is defined, you can add one or more Characteristics by selecting the Characteristic type from the drop down and clicking on the "+" sign.



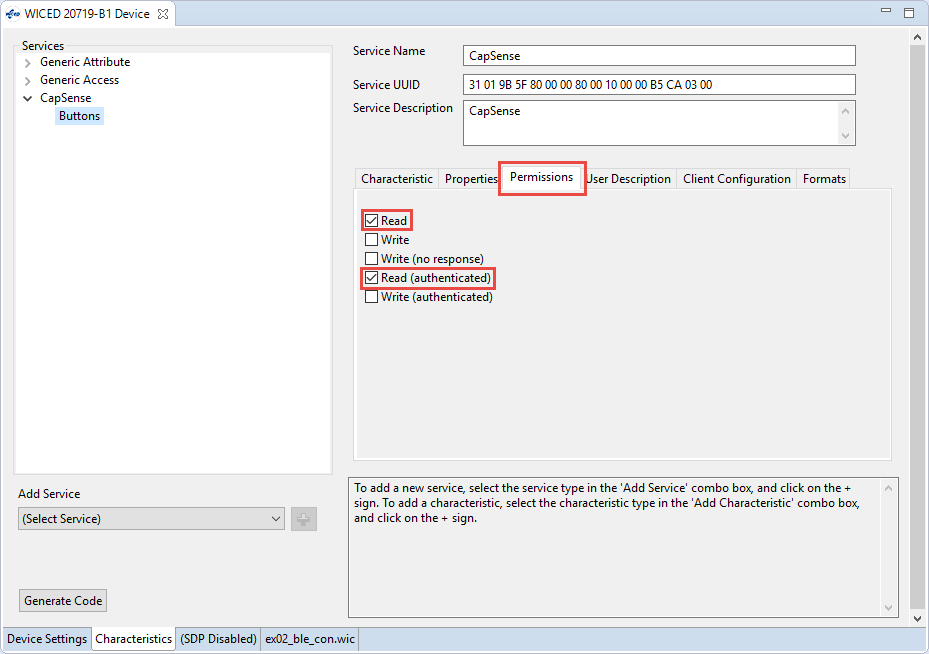
After the characteristic is added, you will have a place to enter the Characteristics properties. Note that there are multiple tabs used to enter all the Characteristic information. In this example, we have a Characteristic called "Buttons" with the UUID defined by Cypress for CapSense buttons. It is set to a size of 3 bytes which is required by the Cypress CapSense Button Characteristic.



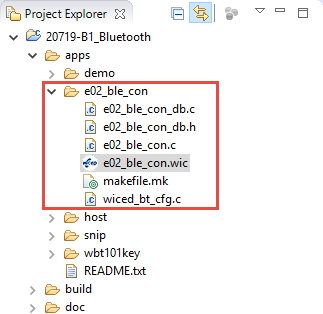
On the Properties tab, you can see that it is set so that the value can be read manually or notifications can be enabled.



On the Permissions tab, the default is "Read" which means the value can be read even prior to pairing (i.e. before the connection is encrypted). If you want to force an encrypted connection before allowing the value to be read, enable the settings for both "Read" and "Read (authenticated)".



Once everything is setup the way you want it, go back to the Device Settings tab and click "Generate Code". When that step is done, the project folder will look like the following. Note that there are separate C and header files for the GATT database structure called <project>\_db.c/.h.



At this point, you can move the project if desired and then edit the files to add your own application's functionality. The main project C file has sections marked with the text "TODO" which give you some hints as to where customization may be required.

As stated earlier, you can re-run the tool by opening the .wic file from the project. Note that if you moved the project, you will no longer be able to re-run the tool using that configuration file unless you move the project back under the apps folder first. When you re-generate code, all the existing generated files will be backed up first (with the extension .bak\_1) before creating the new ones but remember that any custom code you added will need to be manually added back into the new files.

# Advanced Topics

DTM – Direct Test Mode

HCI - Host Control Interface

OTA updates

Multi-role devices

Mesh

# Exercise(s)

* 1. Create a BLE Advertiser

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 |

1. Run the 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\ch04* 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>\_le01* 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 makefile, comment out the HCI\_TRACE\_OVER\_TRANSPORT and 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.
11. 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.

Questions:

1. What is the name of the Bluetooth Stack callback function?
2. How many bytes is the advertisement packet?
3. How long does the device advertise in the high duty cycle mode before switching to the low duty cycle mode? Where is this set (what file and what parameter)?
4. Why doesn’t low duty cycle advertising ever timeout?
   1. Connect using BLE

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 their values or be notified of changes.

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 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) |  |
| Pair 🡪 | BTM\_SECURITY\_REQUEST\_EVT 🡪 | Grant security |
| BTM\_PAIRING\_IO\_CAPABILITIES\_BLE\_REQUEST\_EVT🡪 | Capabilities are set |
| BTM\_ENCRYPTION\_STATUS\_EVT | Not used yet |
| BTM\_PAIRED\_DEVICE\_LINK\_KEYS\_UPDATE\_EVT | Not used yet |
| BTM\_PAIRING\_COMPLETE\_EVT | Not used yet |
| Read CapSense characteristic while touching buttons 🡪 | GATT\_ATTRIBUTE\_REQUEST\_EVT, GATTS\_REQ\_TYPE\_READ 🡪 | Returns button state |
| Read CapSense CCCD 🡪 | GATT\_ATTRIBUTE\_REQUEST\_EVT, GATTS\_REQ\_TYPE\_READ 🡪 | Returns button notification setting |
| Write 01:00 to CapSense CCCD 🡪 | GATT\_ATTRIBUTE\_REQUEST\_EVT, GATTS\_REQ\_TYPE\_WRITE 🡪 | Enables notifications |
| Touch buttons 🡪 |  | Send notifications |
| 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. |

1. Run the WICED Bluetooth Designer and set up a project called *ex02\_ble\_con.*
   1. Select *Unknown* for the *Appearance*.
   2. Enable the GATT database.
   3. 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. Note: the UUIDs are NOT the same – the 4th byte from the end are different.
      3. The size of the characteristic is 3 bytes.
      4. Set the initial value to *04 00 00*.
      5. Set the Properties to *Read* and *Notify*.
      6. Set the Permissions to *Read* and *Read (authenticated)*. This will mean that you need an authenticated connection (i.e. encrypted) before you will be able to read button values.
   5. Generate the code.
2. Move the project to the *wbt101\ch04* 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>\_le02* 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>\_le02* 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. Setup a global variable of type uint16\_t called connection\_id that will be used to save the connection ID. This will be used to send notifications when CapSense button values change. Initialize it to 0.
8. In the GATT connect callback function:
   1. On a connection:
      1. Save the connection ID to the variable connection\_id. That is:
         1. connection\_id = p\_conn\_status->conn\_id;
      2. Add a function call to enable pairing. If you don't do this, you won't be able to pair with the device which means you won't be able to enable notifications.
      3. Hint: Search for "TODO: Handle the connection" to find the right place to add the above changes.
   2. On a disconnection:
      1. Reset connection\_id to 0.
      2. Re-start advertisements using wiced\_bt\_start\_advertisements().
      3. Hint: Search for "TODO: Handle the disconnection"
9. 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.
10. 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.
    2. Before the main loop, initialize the I2C master.
    3. Do an initial I2C write to set the appropriate offset for the button data.
    4. In the main loop, 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).
       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. If the value has changed AND a connection is present (i.e. connection\_id != 0) AND the client has registered for notifications, send a notification with the new value.
       1. Hint: use the function wiced\_bt\_gatt\_send\_notification.
    7. Delay for 100ms.
11. 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.
12. In the application initialization (ex02\_ble\_con\_app\_init which is called during the event BTM\_ENABLED\_EVT) initialize and create the CapSense thread.
13. In the BTM\_PAIRING\_IO\_CAPABILITIES\_BLE\_REQUEST\_EVT event, change the following two settings:
    1. p\_event\_data->pairing\_io\_capabilities\_ble\_request.auth\_req = BTM\_LE\_AUTH\_REQ\_SC\_MITM\_BOND;
    2. p\_event\_data->pairing\_io\_capabilities\_ble\_request.init\_keys = BTM\_LE\_KEY\_PENC|BTM\_LE\_KEY\_PID;

These settings are used to determine the type of security used during pairing. The new settings specify to use a secure connection. (GJL: Is this accurate? What else can we add?)

1. In the makefile, comment out the HCI\_TRACE\_OVER\_TRANSPORT and 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.
2. Program the project to the board.
3. Open the mobile CySmart app. Open the app settings and make sure that "Delete bond on disconnect" is checked since we don't want to save bonding information (yet).
4. Connect to the device. Pairing should happen automatically.
5. Open the CapSense widget and observe the button display while touching the CapSense buttons on the kit.
6. Disconnect from the mobile CySmart app and start the PC CySmart app.
   1. Hint: you must have a CY5577 CySmart BLE USB dongle connected to your PC.
7. Start scanning and then connect to your device.
8. Click on "Discover all Attributes" and then on "Enable Notifications". Notice that you will get an authentication error. Click "OK" to close the error window.
9. Click on "Pair" and click "Yes" to add the device to the resolving list.
10. Click on "Enable All Notifications" again. Now when you touch a button you will see the characteristic value change.
    1. Hint: you can also turn on/off notifications individually by selecting the Client Characteristic Configuration Description (CCCD) attribute and writing a 1 (to enable) or a 0 (to disable) to the first byte.
11. Click on "Disable All Notifications", then read the CapSense button values manually in CySmart by clicking on the characteristic and then clicking the "Read Value" button.
12. Click "Disconnect".
13. From the Device List window (left side below the Discovered devices list), select any device listed and select "Clear -> All". This will remove any bonding information since we are not saving bonding information on the device (yet).

Questions:

1. What function is called when there is a Bluetooth 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?
5. How long does the device advertise in the high duty cycle mode before switching to the low duty cycle mode? How long does it advertise in the low duty cycle mode before stopping advertisements? Where are these specified?
   1. Save BLE Bonding Information

The prior exercise has been modified for you to save and restore bonding information to NVRAM. You will copy over the code, program it to your kit, experiment with it, and then answer questions about the stack events that occur.

By saving bonding information on both sides (i.e. the client and the server) future connections between the devices can be established more quickly with fewer steps. This is particularly useful for devices that require a pairing key (which will be added in the next exercise) since saving the bonding information means the key doesn't have to be entered every time the device connects.

Moreover, since the keys are saved on both devices, they don't need to be exchanged again. This means that after the first connection, there is no possibility of a MIM attack since the keys are not sent out over the air.

1. Copy ex03\_ble\_bond from the answer key.
   1. Create a new make target
   2. Update the device name in *wiced\_bt\_cfg.c* and *ex03\_ble\_bond.c* to *<inits>\_le03* where *<inits>* is your initials.
2. Open a UART terminal window to the PUART.
3. Build the project and program it to the board.
4. Open the CySmart PC application and connect to the dongle.
5. If there is anything listed in the "Device List" near the bottom of the screen, select any device from the list and choose "Clear > All". This will remove any stored bonding information from the PC so that it will not conflict with your new firmware. It is necessary to do this each time you re-program the kit so that the old information is not used.
6. Start scanning and then connect to your device.
7. Click on "Discover all Attributes".
8. Click on "Pair" and click "Yes" to add the device to the resolving list.
   1. Note down the Bluetooth Stack events that occur during pairing. This information is displayed in the UART.
9. Click on "Enable All Notifications". Touch the CapSense buttons and observe the characteristic value change.
10. Click "Disconnect". Do NOT remove the device from the Device List this time – we want bonding information retained.
11. Start scanning again and re-connect to your device.
12. Click on "Discover all Attributes" and "Pair".
    1. Once again note down the Bluetooth Stack events that occur during pairing. You will notice that fewer steps are required this time.
13. Note that notifications are enabled since they were enabled when you disconnected.
14. Disconnect again.
15. Power cycle the board. Once reconnected, either reset or re-open the UART terminal window.
16. Start scanning and then connect to your device for a third time.
17. Click on "Discover all Attributes" and "Pair" again.
    1. Note down the Bluetooth Stack events that occur this time during pairing. Compare to the previous two connections.
18. Disconnect from the device from the CySmart PC app. Clear the Device List so that the saved boding information won't interfere with the next exercise.
    1. Hint: You should clear the bonding information from CySmart anytime you are going to reprogram the kit since it will no longer have the bonding information on its side.

An overview of the changes from the prior exercise required to save/restore bonding information:

1. A structure called "hostinfo" is created which holds the BD\_ADDR of the bonded device and the value of the CapSense CCCD. The BD\_ADDR is used to determine when we have reconnected to the same device while the CCCD value is saved so that the state of notifications can be retained across connections for bonded devices.
2. Before initializing the GATT database, existing keys (if any) are loaded from NVRAM. If no keys are available this step will fail so it is necessary to look at the result of the NVRAM read. If the read was successful, then the keys are copied to the address resolution database.
   1. This makes the keys available with connecting to a bonded device.
3. In the Bluetooth stack event *BTM\_PAIRING\_COMPLETE\_EVT*, if bonding was successful write the information from the hostinfo structure into the NVRAM.
   1. This saves hostinfo upon initial pairing. This event is not called when bonded devices reconnect.
4. In the Bluetooth stack event *BTM\_ENCRYPTION\_STATUS\_EVT*, read bonding information from the NVRAM into the hostinfo structure.
   1. This reads hostinfo upon a subsequent connection when devices were previously bonded.
5. In the Bluetooth stack event *BTM\_PAIRED\_DEVICE\_LINK\_KEYS\_UPDATE\_EVT*, save the keys for the peer device to NVRAM.
6. In the Bluetooth stack event *BTM\_PAIRED\_DEVICE\_LINK\_KEYS\_REQUEST\_EVT*, read the keys for the peer device from NVRAM.
7. In the Bluetooth stack event *BTM\_LOCAL\_IDENTITY\_KEYS\_UPDATE\_EVT*, save the keys for the local device to NVRAM.
8. In the Bluetooth stack event *BTM\_LOCAL\_IDENTITY\_KEYS\_REQUEST\_EVT*, read the keys for the local device from NVRAM.
9. In the GATT connect callback:
   1. For a connection, save the BD\_ADDR of the remote device into the hostinfo structure. This will be written to NVRAM in the *BTM\_PAIRING\_COMPLETE\_EVT*.
   2. For a disconnection, clear out the BD\_ADDR from the hostinfo structure and reset the CCCD to 0.
10. In the GATT set value function, save the CapSense Button CCCD value to the hostinfo structure whenever it is updated and write the value into NVRAM.

Questions:

1. What stack events occur on first connection, subsequent connections, and subsequent connections after a power cycle? Why?

**1st Connection Subsequent Connection Subsequent Connection**

**(No Power Cycle) (Power Cycle)**

1. What items are stored in NVRAM?
2. Which event stores each piece of information?
3. Which event retrieves each piece of information?
   1. Add a Pairing Key

In this exercise, you will copy the project from the previous exercise and modify it to require a key to be entered to pair the device the first time. The key will be randomly generated by the device and will be displayed over the UART. The key will need to be entered in CySmart or in your Phone's Bluetooth connection settings before pairing will be allowed.

1. Copy ex03\_ble\_bond to ex04\_ble\_passkey. Rename the files and make the necessary updates.
   1. Hint: Change the name from *<inits>\_le03* to *<inits>\_le04* in the wiced\_bt\_cfg.c file and the ex04\_ble\_passkey.c file.
   2. Hint: Don’t forget to look for header file names in the two C files that contain ex03\_ble\_bond.
   3. Hint: Many function names and variable names start with "ex03\_ble\_bond". You can do a global search/replace to change these to "ex04\_ble\_passkey" if you want them to be consistent with the project name.
2. In the Bluetooth stack event BTM\_PAIRING\_IO\_CAPABILITIES\_BLE\_REQUEST\_EVT:
   1. Change the value for pairing\_io\_capabilities\_ble\_request.local\_iop\_cap from BTM\_IO\_CAPABILITIES\_NONE to BTM\_IO\_CAPABILITIES\_DISPLAY\_ONLY.
      1. This indicates that the device can display a key value.
3. Add a Bluetooth stack event called BTM\_PASSKEY\_NOTIFICATION\_EVT to send the value of the passkey to the UART.
   1. Hint: Make sure you print something around the value so that it is easy to find in the terminal window.
   2. Hint: The passkey must be 6 digits so print leading 0's if the value is less than 6 digits. (i.e. use %06d).
   3. Hint: The key is passed to the callback as:
      1. p\_event\_data->user\_passkey\_notification.passkey
4. Program the project to the board.
5. Open a UART terminal window.
6. Open the mobile CySmart app. Open the settings. Turn off "Delete bond on disconnect" and turn on "Initiate pairing after connection".
7. Attempt to Connect to the device. You will see a notification from the Bluetooth system asking for the passkey to be entered. Find the passkey on the UART terminal window and enter it into the device.
8. Once pairing and bonding completes, verify that the application still works.
9. Disconnect and reconnect. Observe that the key does not need to be entered to connect this time.
10. Check the setting "Delete bond on disconnect", then disconnect and reconnect. Observe that the key must be entered again to connect.
11. Now try the same thing using the PC version of CySmart. It will pop up a window when the passkey is needed.

Questions:

1. Other than BTM\_IO\_CAPABILITIES\_NONE and BTM\_IO\_CAPABILITIES\_DISPLAY\_ONLY, what other choices are available? What do they mean?
2. What additional stack callback event occurs compared to the previous exercise? At what point does it get called?
   1. (Advanced) Implement a BLE Central Device ???

??? Use Bluetooth designer or do from a template?

* 1. (Advanced) Join a BLE Mesh Network ???

??? Use Bluetooth designer or do from a template?

# Recommended Reading