Chapter 4B: More Advanced BLE Peripherals

Time 2 ¼ Hours

This chapter expands your basic knowledge of BLE Peripherals by introducing more Attribute Procedures, GATT Database Features, Security, WICED Configuration Files, Advertising Packet Features, Service Discovery, …

4B.1 Notify & Indicate 2

4B.2 Other Characteristic Descriptors 3

4B.3 Security 5

4B.3.1 Pairing 5

4B.3.2 Bonding 6

4B.3.3 Pairing & Bonding Process Summary 6

4B.3.4 Authentication, Authorization and the GATT DB 6

4B.3.5 Privacy 6

4B.4 WICED Configuration: wiced\_bt\_cfg.c 7

4B.5 WICED Configuration: Buffer Pools 8

4B.6 Advertising packet 9

4B.6.1 Using the Advertising Packet to Get Connected 9

4B.6.2 iBeacon 10

4B.6.3 Eddystone 10

4B.7 GATT Service Discovery 11

4B.8 WICED Bluetooth Designer 12

4B.8.1 Running the Tool 12

4B.8.2 Editing the Firmware 15

4B.8.3 Testing the Project 18

4B.9 WICED Bluetooth Firmware Architecture 20

4B.10 Low Power 26

4B.11 WICED Chips & the Architecture of HCI 26

4B.11.1 HCI 26

4B.11.2 BT Spy 28

4B.12 Exercises 29

Exercise - 4B.1 BLE Notifications 29

Exercise - 4B.2 BLE Pairing and Security 32

Exercise - 4B.3 Save BLE Pairing Information (i.e. Bonding) 35

Exercise - 4B.4 Add a Pairing Passkey 39

# Notify & Indicate

In the previous chapter, we talked about how the GATT Client can Read and Write the GATT Database running on the GATT Server. But, there are cases where you might want the Server to initiate communication. For example, if your Server is a CapSense Peripheral device, you might want to send the Client an update each time the CapSense values change. That leaves us with the obvious questions of how does the Server initiate communication to the Client, and when is it allowed to do so?

The answer to the first question is, the Server can notify the Client that one of the values in the GATT Database has changed by sending a Notification message. That message has the Handle of the Characteristic that has changed and a new value for that Characteristic. Notification messages are not responded to by the Client, and as such are not reliable. If you need a reliable message, you can instead send an Indication which the Client must respond to.

To send a Notification or Indication use the APIs:

* wiced\_bt\_send\_notification(conn\_id, handle, length, value)
* wiced\_bt\_send\_indication(conn\_id, handle, length, value)

By convention, the GATT Server will not send Notification or Indication messages unless they are turned on by the Client.

How do you turn on Notifications or Indications? In the last chapter, we talked about the GATT Attribute Database, specifically, the Characteristic. If you recall, a Characteristic is composed of a minimum of two Attributes:

* Characteristic Declaration
* Characteristic Value

However, information about the Characteristic can be extended by adding more Attributes, which go by the name of Characteristic Descriptors.

For the Client to tell the Server that it wants to have Indications or Notifications, four things need to happen.

First, the Server must add a new Characteristic Descriptor Attribute called the Client Characteristic Configuration Descriptor, often called the CCCD. This Attribute is simply a 16-bit mask field, where bit 0 represents the Notification flag, and bit 1 represents the Indication flag. In other words, the Client can Write a 1 to bit 0 of the CCCD to tell the Server that it wants Notifications.

To add the CCCD to your GATT DB use the following Macro:

* CHAR\_DESCRIPTOR\_UUID16\_WRITABLE (
  + <HANDLE>,
  + UUID\_DESCRIPTOR\_CLIENT\_CHARACTERISTIC\_CONFIGURATION,
  + LEGATTDB\_PERM\_READABLE | LEGATTDB\_PERM\_WRITE\_REQ | LEGATTDB\_PERM\_AUTH\_WRITABLE ),

The permissions above indicate that the CCCD value is readable whenever connected but will only be writable if the connection is authenticated (more on that later). To see the other possible choices, right click on one of them from inside WICED Studio and select "Open Declaration".

Second, you must change the Properties for the Characteristic to specify that the characteristic allows notifications. That is done by adding LEGATTDB\_CHAR\_PROP\_NOTIFY to the Characteristic's Properties. To see all the available choices, right-click on one of the existing Properties in WICED Studio and select "Open Declaration".

Third, in your GATT Attribute Write Callback you need to save the CCCD value that was written to you.

Finally, when a value that has Notify and/or Indicate enabled changes in your system, you must send out a new value using the appropriate API.

# Other Characteristic Descriptors

There are several other interesting Characteristic Descriptors that are defined by the Bluetooth SIG including:



A common Characteristic Descriptor to use is the Characteristic User Description which is just a text string that describes in human format the Characteristic Type. Many GATT Database Browsers (e.g. Light Blue) will display this information when you are looking at the GATT Database. To add the Characteristic User Description to your Characteristic just add:

* CHAR\_DESCRIPTOR\_UUID16 (
  + <Handle>,
  + UUID\_DESCRIPTOR\_CHARACTERISTIC\_USER\_DESCRIPTION,
  + LEGATTDB\_PERM\_READABLE ),

WICED Bluetooth has defines for the rest of the Descriptors which you can find in wiced\_bt\_uuid.h



# Security

To securely communicate between two devices, you want to: (1) Authenticate that both sides know who they are talking to; (2) ensure that all access to data is Authorized, (3) Encrypt all message that are transmitted; (4) verify the Integrity of those messages; and (5) ensure that the Identity of each side is hidden from eavesdroppers.

In BLE, this entire security framework is built around AES-128 symmetric key encryption. This type of encryption works by combining a Shared Secret code and the unencrypted data (typically called plain text) to create an encrypted message (typically called cypher text).

* CypherText = F(SharedSecret,PlainText)

There is a bunch of math that goes into AES-128, but for all practical purposes if the Shared Secret code is kept secret, you can assume that it is very unlikely that someone can read the original message.

If this scheme depends on a Shared Secret, the next question is how do two devices that have never been connected get a Shared Secret that no one else can see? In BLE, the process for achieving this state is called Pairing. A device that is Paired is said to be Authenticated.

## Pairing

Pairing is the process of arriving at the Shared Secret. The basic problem continues to be how do you send a Shared Secret over the air, unencrypted and still have your Shared Secret be Secret. The answer is that you use public key encryption. Both sides have a public/private key pair that is either embedded in the device, or calculated at startup. When you want to authenticate, both sides of the connection exchange public keys. Then both sides exchange encrypted random numbers that form the basis of the shared secret.

But how do you protect against Man-In-The-Middle? There are three possible methods.

Method 1 is called “Just works” In this mode you have no protection against MIM.

Method 2 is called “Out of Band”. Both sides of the connection need to be able to share the PIN via some other connection that is not Bluetooth such as NFC.

Method 3 is called “Numeric Comparison” (V2.PH.7.2.1). In this method, both sides display a 6-digit number that is calculated with a nasty cryptographic function based on random numbers and the public keys of the devices. The user observes both devices. If the key is the same on both, then the user confirms on both sides.

Method 4 is called “Passkey Entry” (V2.PH.7.2.3). For this method to work, at least one side needs to be able to enter a 6-digit numeric code. The other side must either be able to display a code that is randomly generated or else have the ability to enter the same code. In the latter case, the user chooses their own random code that is entered on both sides. Then, an exchange and comparison process starts with the Passkeys being divided up, encrypted, exchanged and compared with the other side.

## Bonding

The whole process of Pairing is a bit painful and time consuming. Certainly, you don’t want to have to repeat it every time two devices connect. This problem is solved by Bonding, which just saves all the relevant information into a non-volatile memory. The allows the next connection to launch without repeating the pairing process.

## Pairing & Bonding Process Summary



## Authentication, Authorization and the GATT DB

In Chapter 4A3.1 we talked about the Attributes and the GATT Database. Each Attribute has a permissions bit field that includes bits for Encryption, Authentication, and Authorization. The WICED Bluetooth Stack will guarantee that you will not be able to access and Attribute that is marked Encryption or Authentication unless the connection is Authenticated and/or Encrypted.

The Authorization flag is not enforced by the WICED Bluetooth Stack. Your Application is responsible for implementing the Authorization semantics. For example, you might now allow someone to turn off/on a switch without entering a password.

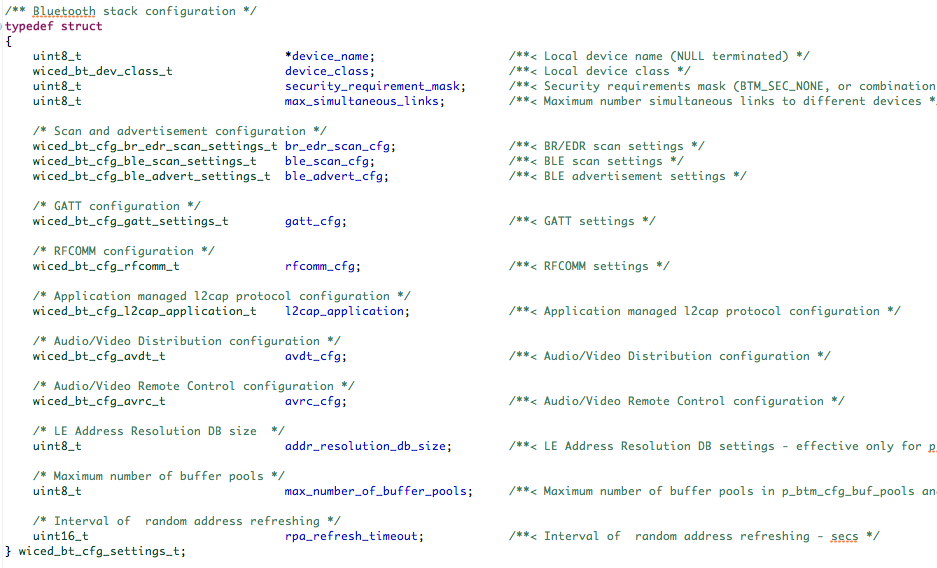
## Privacy

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.

# WICED Configuration: wiced\_bt\_cfg.c

When you initialize the BLE Stack one of the arguments you pass is a pointer to a structure of type wiced\_bt\_cfg\_settings\_t. This structure contains initialization information for both the BLE and Classic Bluetooth configuration. This structure is built for you by WICED Bluetooth Designer and typically resides in the file wiced\_bt\_cfg.c

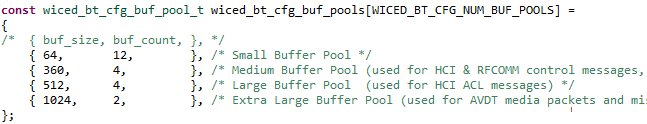
The structure definition is shown below. Note that many of the entries are themselves structures with multiple entries of their own.



# WICED Configuration: Buffer Pools

Rather than use the C typical memory allocation scheme, malloc, the WICED team has built a scheme optimized for Bluetooth. One of the arguments that you need to pass to the Stack initialization function is a pointer to the pools. This array is typically created for you by the WICED Bluetooth Designer.

There are four different size buffer pools. The configuration settings for them can be found in wiced\_bt\_cfg.c. The default settings are:



There is a file in the doc folder inside WICED Studio called WICED-Application-Buffer-Pools.pdf that contains some additional information on the use of buffer pools.

# NVRAM

There are many situations in a Bluetooth system where a non-volatile memory is required. One example of that is Bonding, where you are required to save the Link Keys for future use. WICED Bluetooth provides an abstraction called the “NVRAM”. The exact underlying implementation varies based on the device - i.e. 20719 uses 4K blocks of Flash, but the API and programming model remains the same.

To use the NVRAM, the WICED Bluetooth application developer is given access to block of non-volatile memory that is broken up into variable length non-Volatile Sections labeled with a number called the VSID. The VSID is an unsigned 16-bit integer. Each non-Volatile Section can be up to 255 bytes.

The API can be included in your project with #include “wiced\_hal\_nvram.h” which also #defines the first VSID to be WICED\_NVRAM\_VSID\_START and last VSID to be WICED\_NVRAM\_VSID\_END.

The write function for the NVRAM is:

uint16\_t wiced\_hal\_write\_nvram( uint16\_t vs\_id, uint16\_t data\_length, uint8\_t \*p\_data,

wiced\_result\_t \* p\_status);

The return value is the number of bytes written. You need to pass a pointer to a wiced\_result which will give you the success or failure of the write operation.

The read function for the NVRAM looks just like the write function:

uint16\_t wiced\_hal\_read\_nvram( uint16\_t vs\_id,uint16\_t data\_length, uint8\_t \* p\_data,

wiced\_result\_t \* p\_status);

The return value is the number of bytes read into your buffer, and p\_status tells you if the read succeeded.

You should be aware that the NVRAM has a wear leveling scheme built in that causes the reads and writes to take a variable amount of time. The wear leveling scheme also has a “defragmentation” scheme that runs during chip boot-up.

As the application developer, you are responsible for managing the application semantics of the VSIDs.

# Advertising packet

There are two main uses of the advertising packet:

* Identifying a Peripheral with some recognizable data so that a Central knows how to connect and talk to it.
* Sending out data (e.g. beacon data).

## Using the Advertising Packet to Get Connected

If you turn on the CySmart GATT browser, you will find that there are likely a bunch of unknown devices that are advertising around you. For instance, as I sit here right now I can see that there are several Bluetooth LE devices around me that I have no idea what they are.



When a Central wants to connect to a Peripheral, how does it know what Peripheral to talk to? There are two answers to that question.

First, it may advertise a service that the Central knows about (because it is defined by the Bluetooth SIG or is custom to your company). As we talked in the previous chapter you can customize the Advertising packet with information. In the picture above, you can see that some of the devices are advertising that they support 1 service. To do that they add a field of one of these types to the advertising packet along with the UUID of the Service:



The other scheme that is commonly used is to advertise “Manufacturer's Specific Data”. This data has two parts:

* A two-byte manufacturer code as specified by the Bluetooth SIG (e.g. Cypress = 0x0131).
* The actual data which is typically a Product ID that is unique for each product that the company makes.

The way that this works is that you would write a Central application that has a table of known Peripheral Product IDs that it knows how to talk to. Then the Peripherals would advertise their Manufacturer code and Product ID in the Manufacturers Data Field. When a Central sees something that it knows how to talk to, it can make the connection.

## iBeacon

iBeacon is an Advertising Packet format defined by Apple. The iBeacon information is embedded in the Manufacturer section of the advertising packet. It simply contains:

* Apple’s manufacturing ID
* Beacon type (2-bytes)
* Proximity UUID (16-bytes)
* Major number (2-bytes)
* Minor number (2-bytes)
* Measured Power (1-bytes)

Because the packet uses the Apple company ID you need to register with Apple to use iBeacon.

The measured power allows you to calibrate each iBeacon as you install it so that it can used for indoor location measurement.

## Eddystone

Eddystone is a Google protocol specification that defines a Bluetooth low energy (BLE) Advertising message format for proximity beacon messages. It describes several different frame types that may be used individually or in combinations to create beacons that can be used for a variety of applications.

There are currently four types of Eddystone Packets:

* UID – A unique beacon ID for use in mapping functions
* URL – An HTTP URL
* TLM – Telemetry information about the beacon such as battery voltage, device temperate, counts of packet broadcasts
* EID – Ephermal ID packets which broadcast a randomly changing number

The Advertising Packet has the following fields:

* Flags – Type 01
* 16-bit Service UUID – Type 03
* Service Data – Type 0x13

The Service Data contains the Eddystone packet type, then the actual data. The packet types are:

* UID – 0x00
* URL – 0x10
* TLM – 0x20
* EID – 0x30

In the project snip.ble.eddystone there is an example of creating this type of beacon.

You can find the detailed spec at https://github.com/google/eddystone

# GATT Service Discovery

We know that for a Central to read and write the GATT Database, it must know the handles of the characteristics. If the handles were not established a-priori (e.g. from the Product ID scheme described in Section 4B.6.1 ), then you need some mechanism to figure them out. That mechanism is called GATT Service Discovery.

Previously, we talked about Attribute protocols functions, Read, Write, Notify, Indicate. The Service Discovery procedure uses another Attribute function called “Read Group By Type”. The Group is just a range of Handles, and Type is the Attribute type. When a Central wants to discover all the Primary Services on a Peripheral, it will send a Read Group by Type request with the Handle Range set to 1 🡪 0xFFFF (all the possible Handles) and the Attribute Type set to <<Primary Service>>. The Peripheral will then respond with a list of the Primary Services, the UUIDs, the Handle start and end range for each Service.

Once the Central knows all the Service UUIDs and Handles, it can then iterate through each of the sub-ranges using the same Read Group by Type and look for Characteristics, Descriptors, etc.

On the Peripheral, the WICED Bluetooth Stack has a reference your GATT Database, and as such it responds to these requests automatically for you.

On the Central, you will need to implement the service discovery algorithm by calling the function wiced\_bt\_gatt\_send\_discover to execute the Read Group by Type request and then iterate through the responses to figure out the Handles, UUIDs etc.

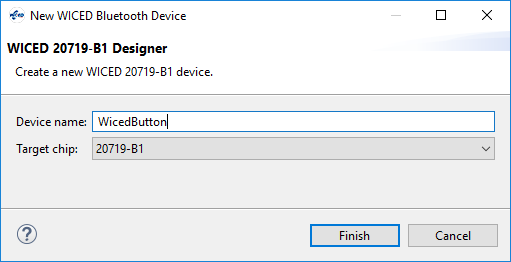
# WICED Bluetooth Designer

WICED Bluetooth Designer can be used to setup Characteristics for Notify and Indicate. It can also be used to create Characteristic User Descriptions.

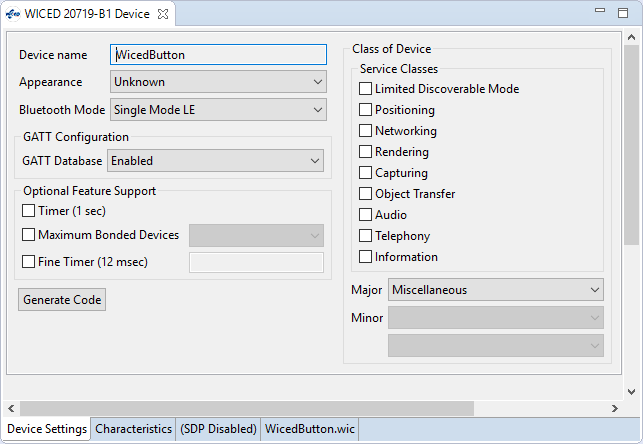
For this example, I'm going to build a BLE project that has a custom service called WicedButton Service containing one Characteristic called MB1. This MB1 Characteristic will hold a count of presses of mechanical button MB1 on the shield. The value will increment every time the button is pressed. It will be Readable by the Client and it will send notifications if the Client enables them.

## Running the Tool

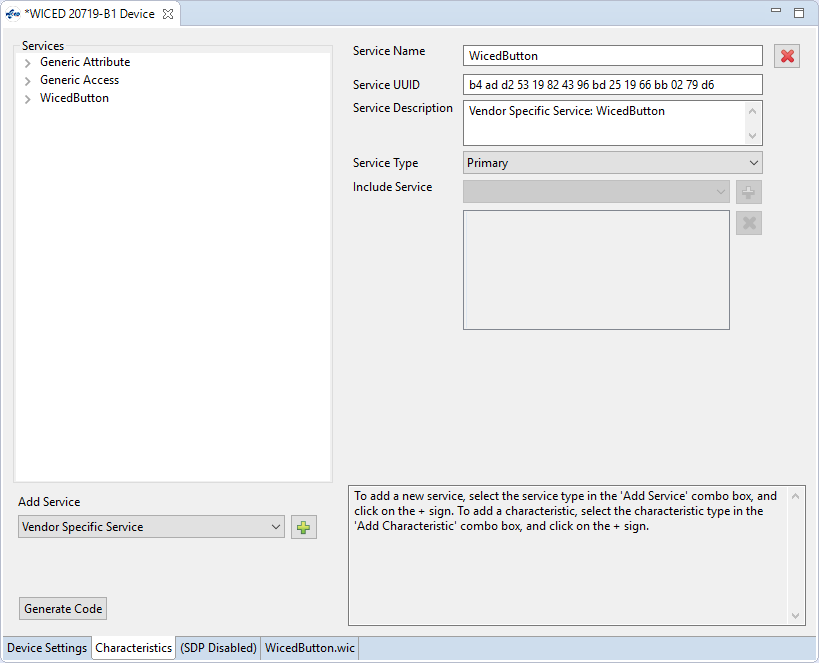
Start the tool as before from *File->New->WICED Bluetooth Designer* and enter WicedButton for the device name. Click on Finish to launch the configuration window.



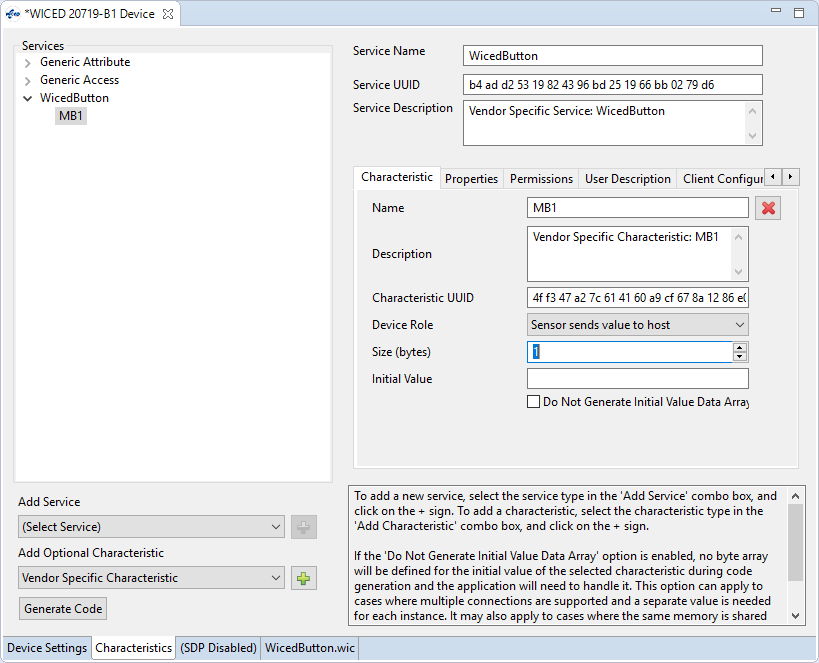
We will keep all the defaults on the Device Settings tab.



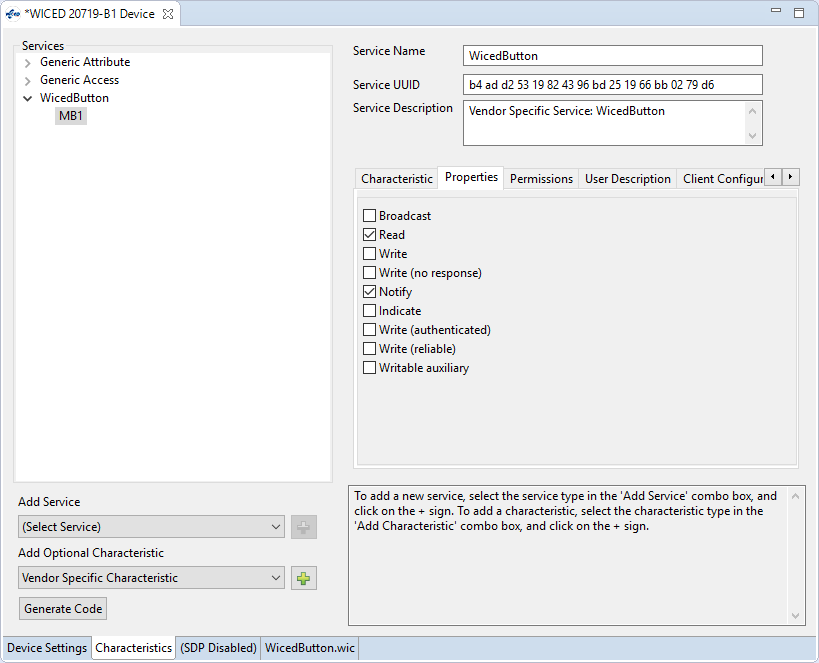
On the Characteristics tab, add a new Vendor Specific Service called WicedButton.



Next, add a Vendor Specific Characteristic and change its name and description to MB1. It has a size of 1.

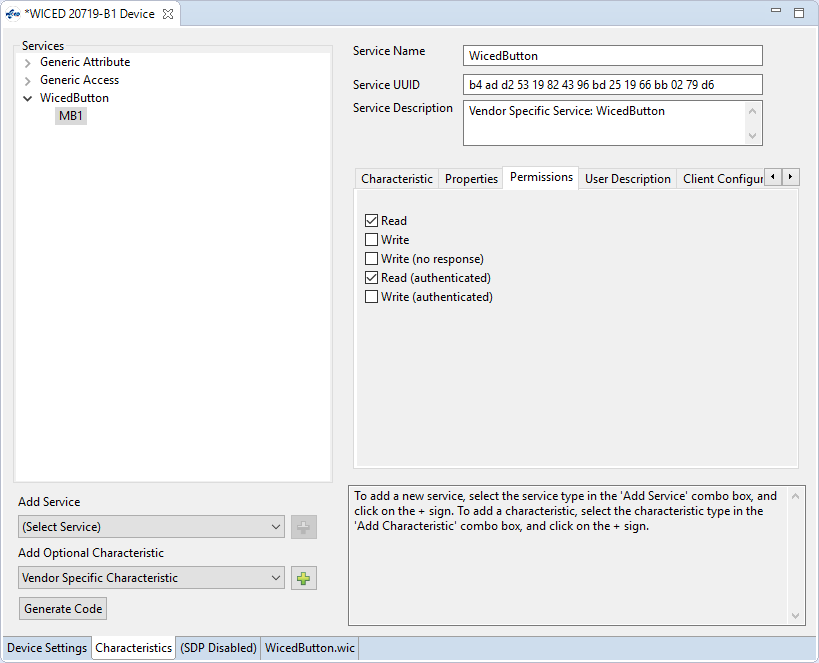


On the Properties tab, we want this Characteristic to have Read and Notify selected.



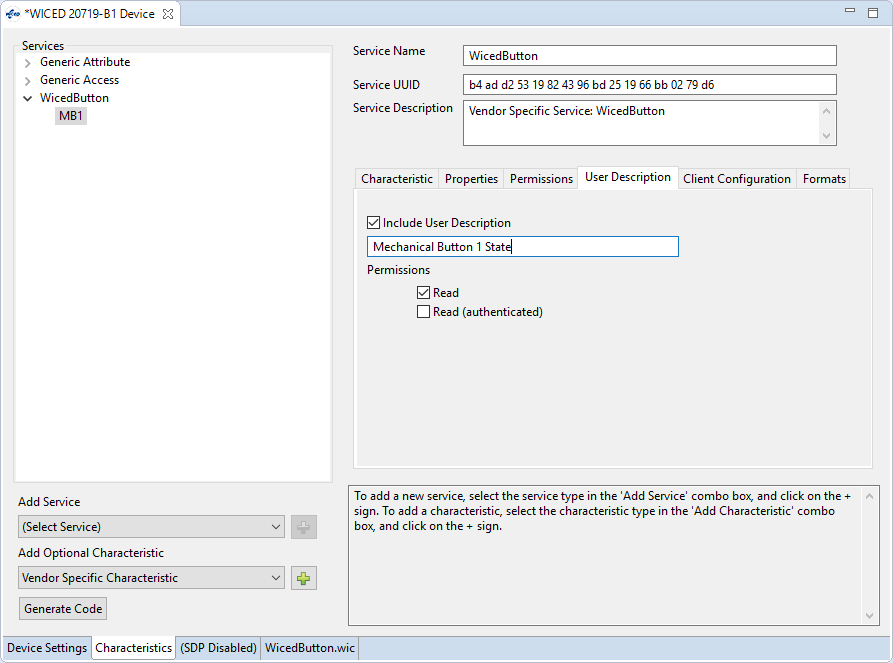
Now check the Permissions tab. It was set by the tool to Read based on our Properties selections. This means that we will be able to Read the Characteristic value without Pairing first. Let's also turn on Read (authenticated) so that Read will require an Authenticated (i.e. Paired) link.

(Note, you must also leave on Read although that does NOT mean that it will be Readable with a non-Authenticated link.)



Enabling/disabling Notifications requires a Paired connection by default – it can't be changed in WICED Bluetooth Designer, but you'll see how that can be done in the exercises.

Next, let's go to the User Description tab and include a User Description with the value of "Mechanical Button 1 State". We will allow this to be read without an Authenticated link.



Now, click the Generate Code button.

## Editing the Firmware

In WicedButton.c, we need to:

1. Add includes for 3 header files:

**#include** "wiced\_bt\_stack.h"

**#include** "wiced\_bt\_app\_common.h"

**#include** "wiced\_hal\_wdog.h"

1. Switch the debug messages to the PUART.



1. Declare a global variable called connection\_id. Upon a GATT connection (i.e. in wicedbutton\_connect\_callback), save the connection ID. Upong a GATT disconnection, reset the connection ID. The ID is needed to send a notification – you need to tell it which connected device to send the notification to. In our case we only allow one connection at a time but there are devices that allow multiple connections.

Global Variable:

uint16\_t connection\_id = 0;

GATT Connection:

/\* TODO: Handle the connection \*/

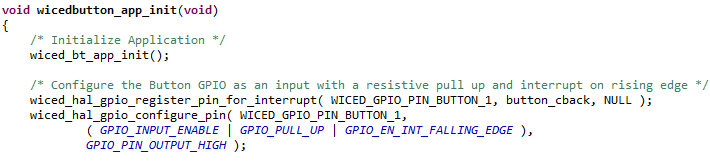
connection\_id = p\_conn\_status->conn\_id;

GATT Disconnection:

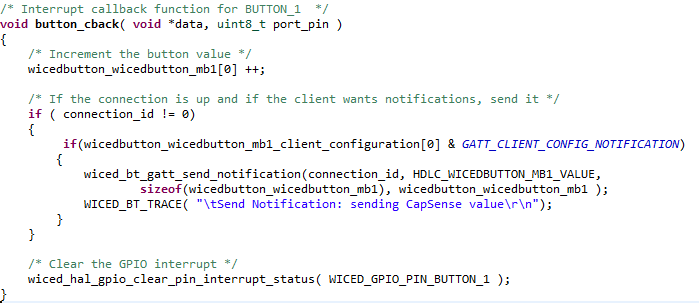
/\* TODO: Handle the disconnection \*/

connection\_id = 0;

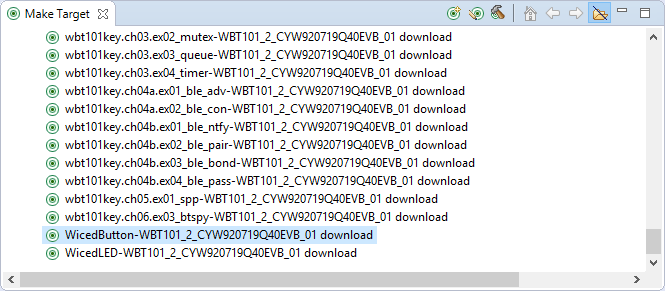
1. Configure MB1 as a falling edge interrupt.



1. In the ISR, we will increment the Characteristic value, and then send a notification if we have a connection and the notification is enabled.

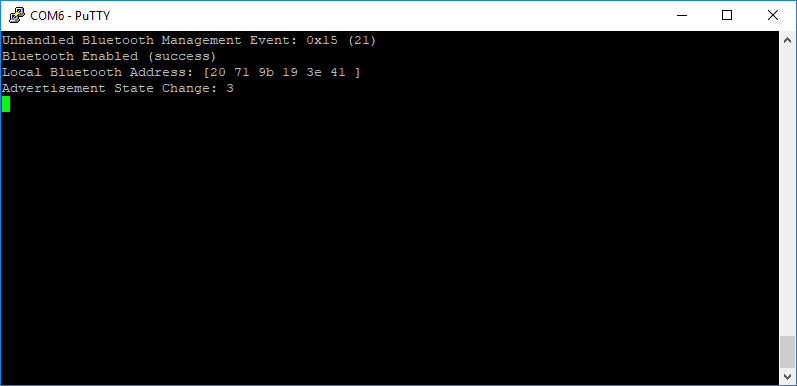


1. Update the Make Target to target for the kit/shield combination platform.



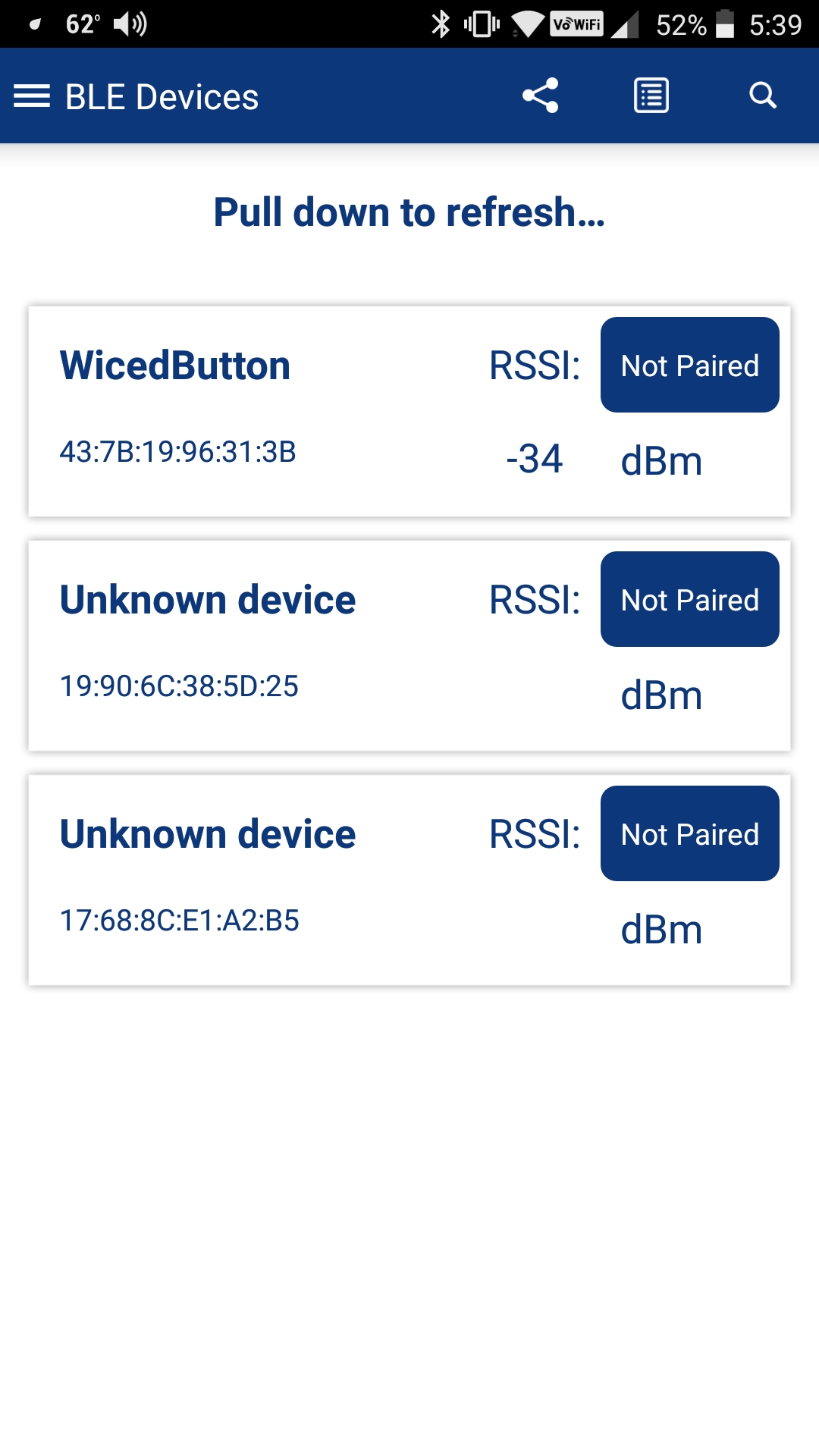
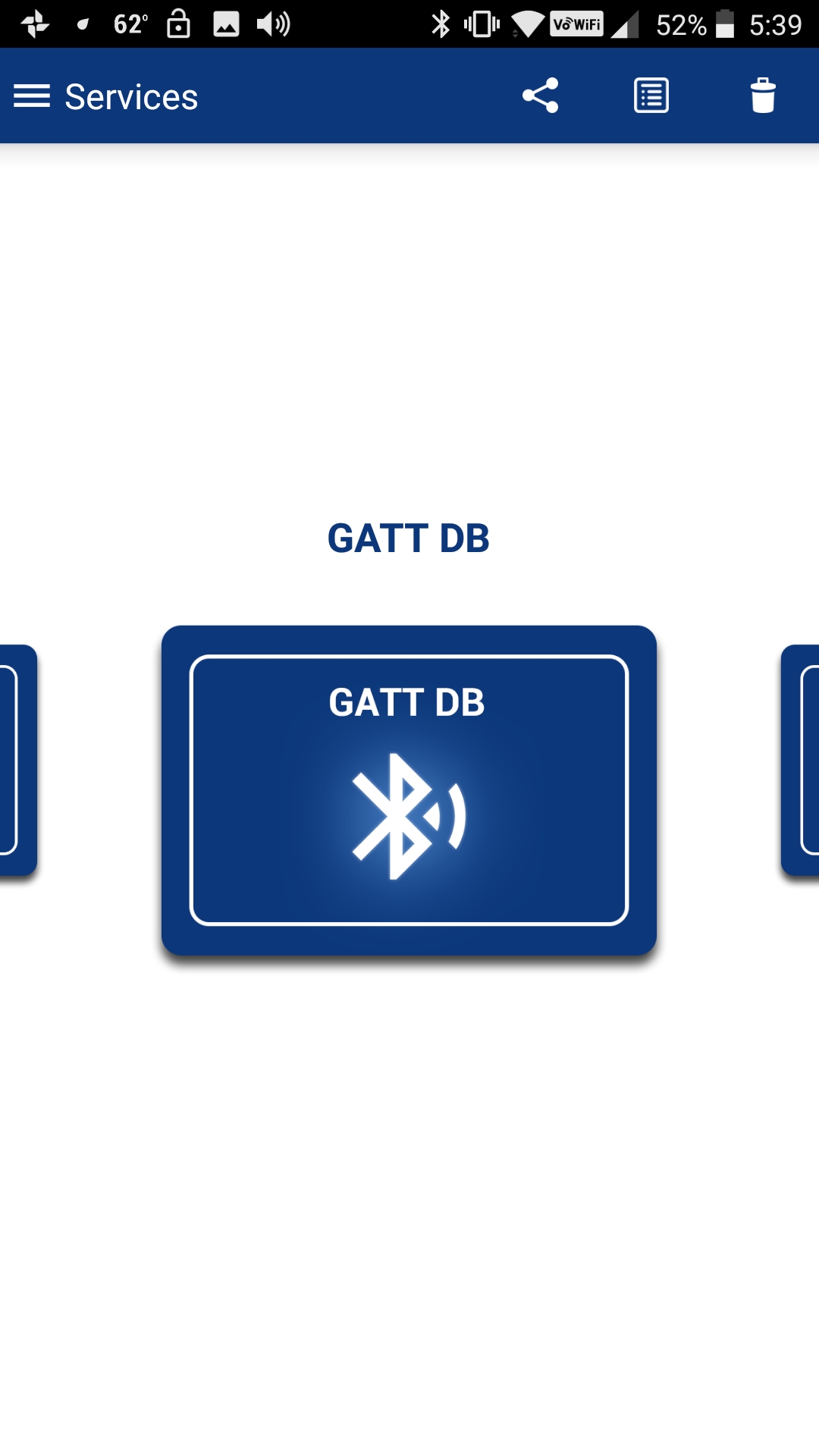
## Testing the Project

Start up a UART terminal and then run the Make Target to program the kit. When the firmware starts up you will see some messages.

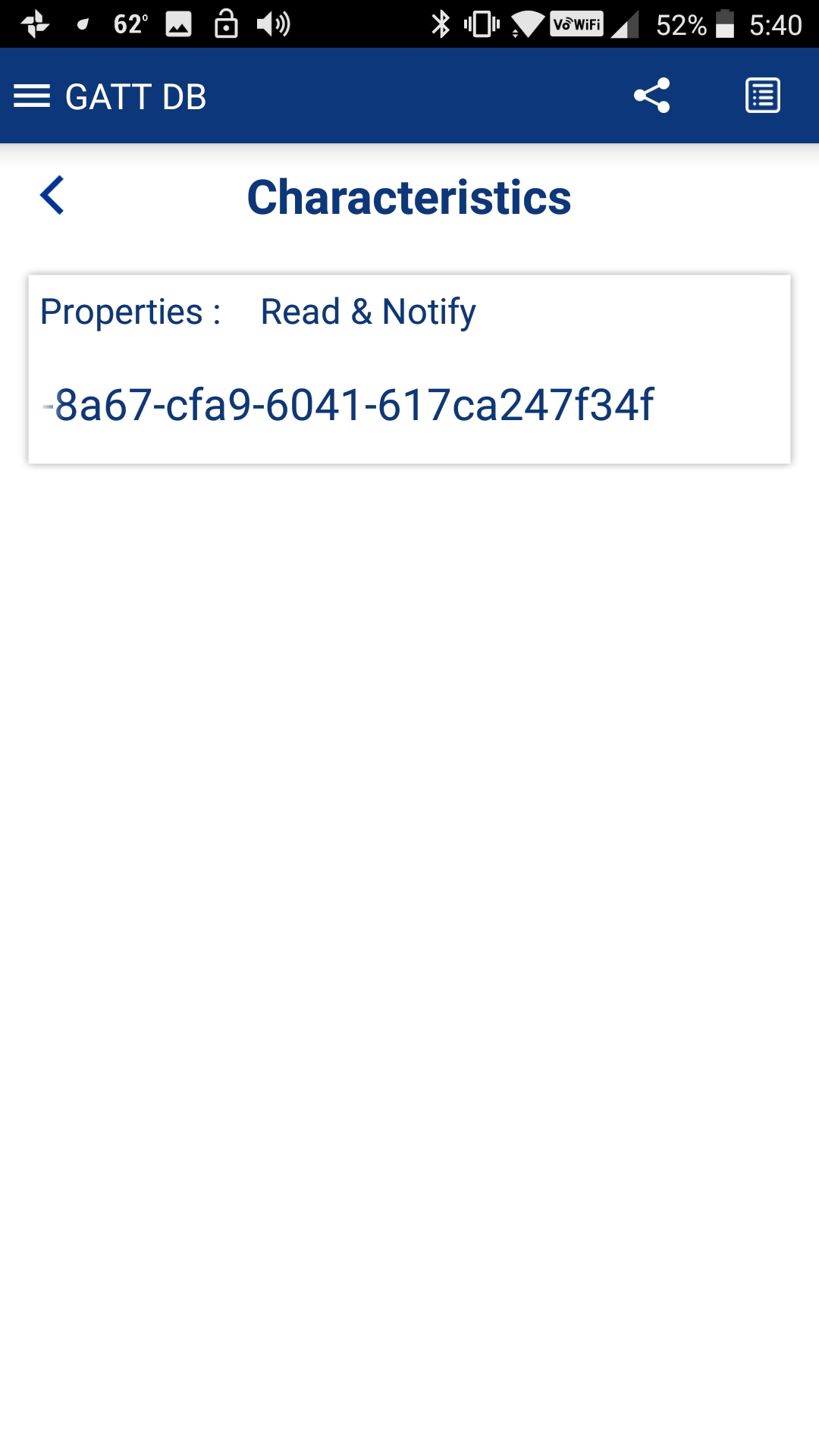
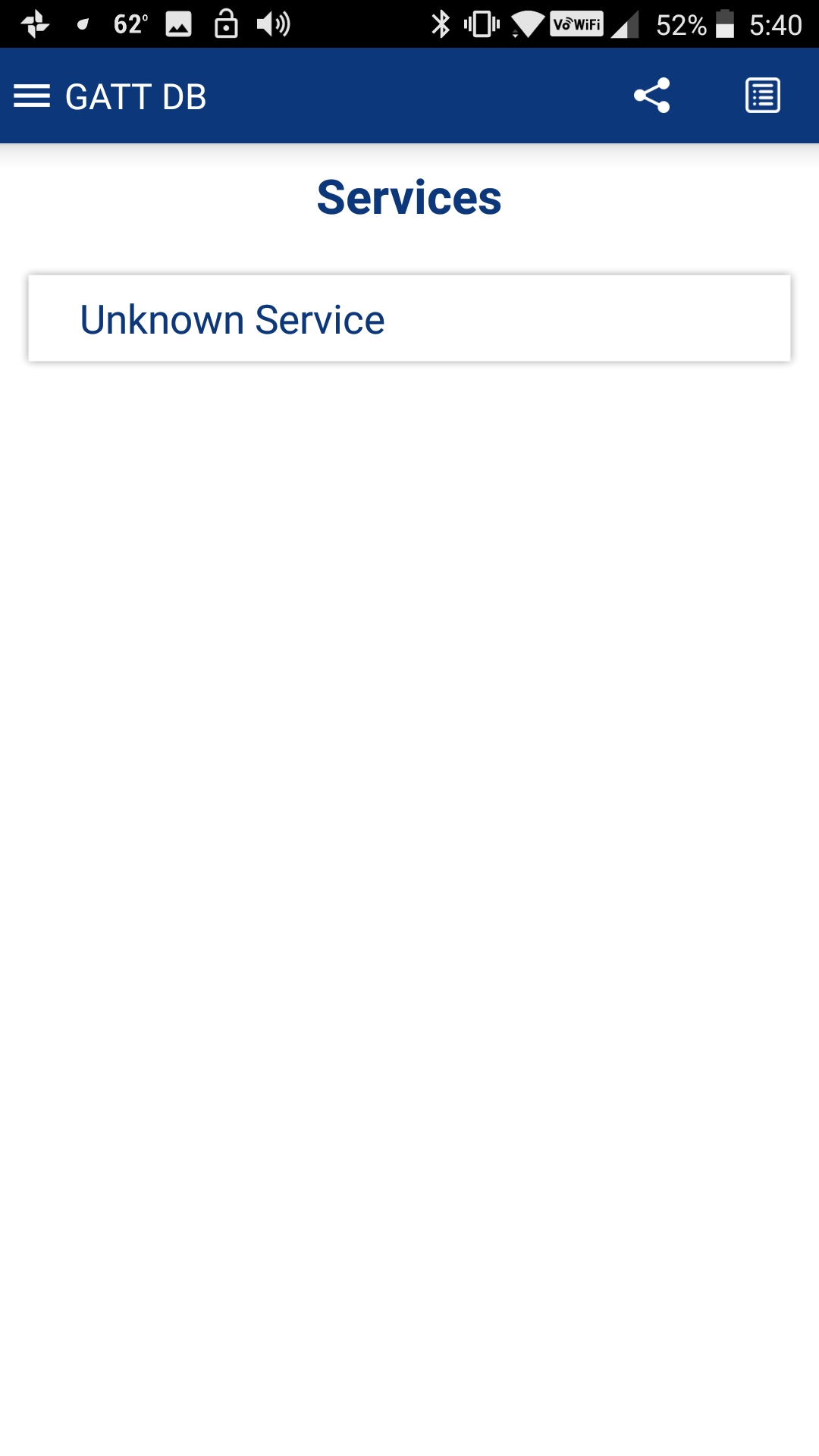


Run CySmart on your phone. When you see the “WicedLED” device, tap on it. CySmart will connect to the device and will show the GATT browser widget.

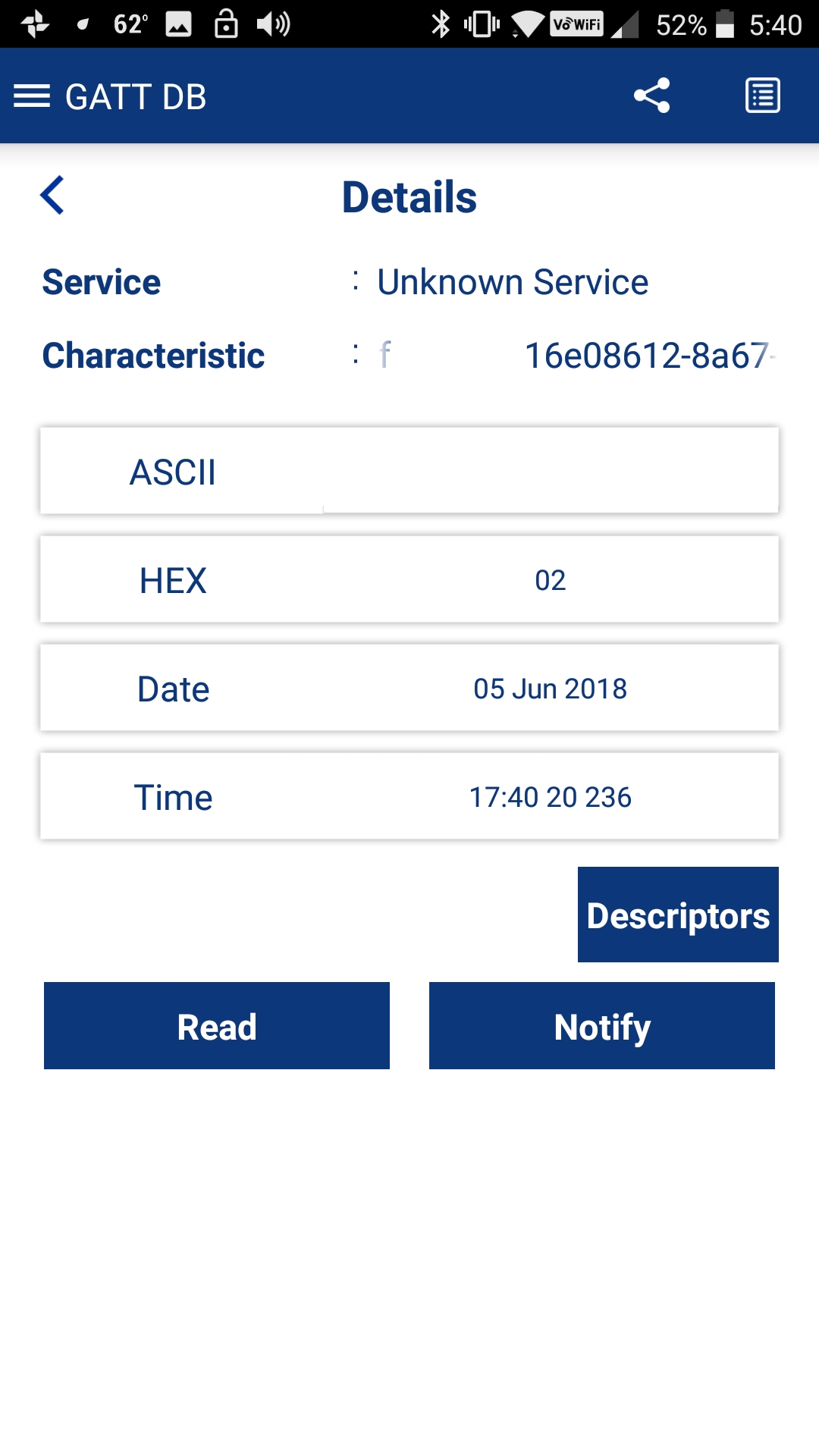
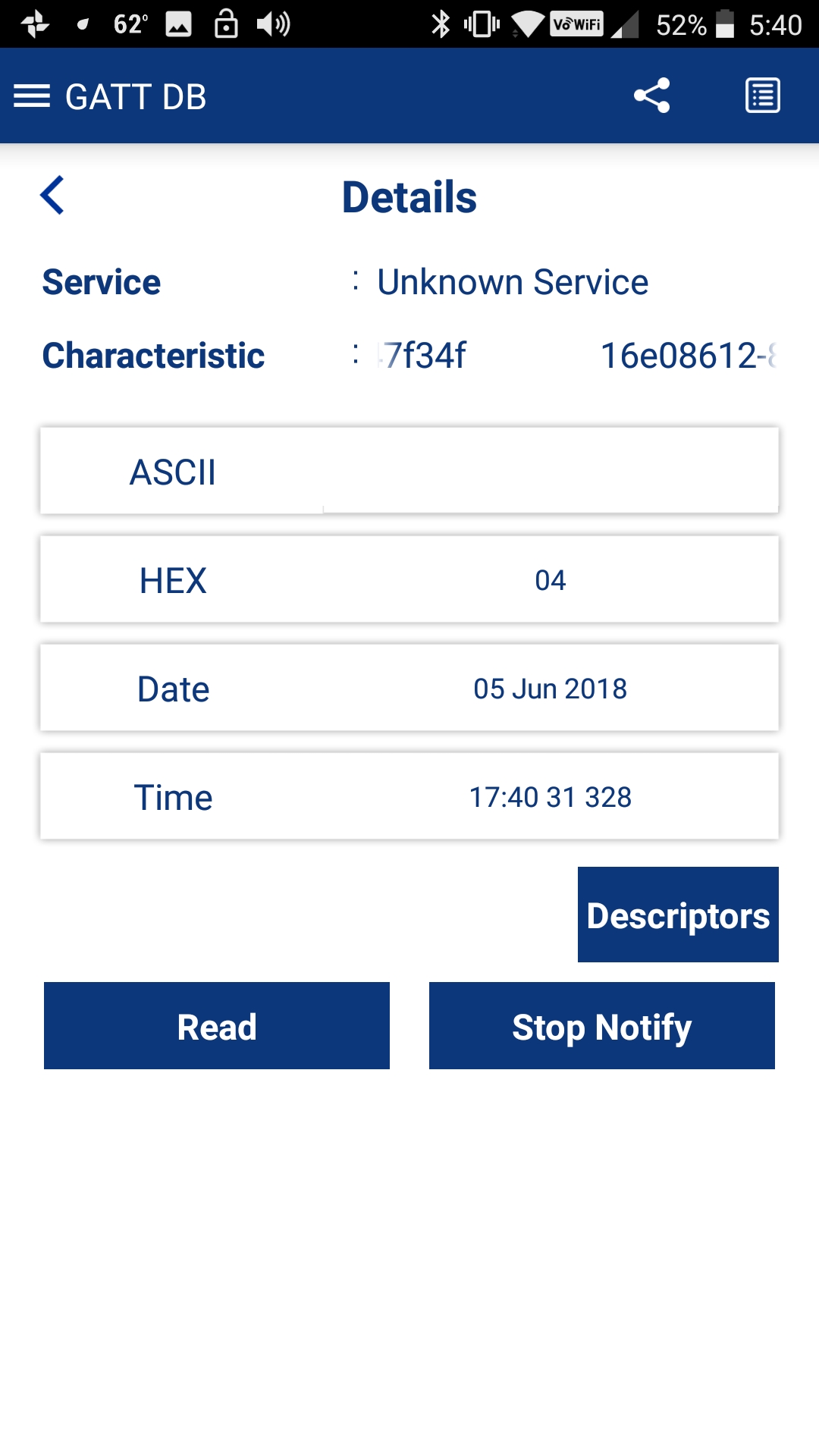
Note: If you are using the Android version of CySmart, before connecting go to the Settings. Turn ON the option "Initiate pairing after connection" and turn ON the option "Delete bond on disconnect". If not, the connection will fail the second time because we have not (yet) saved bonding information. If you are using an iPhone, you will want to go to the phone's Bluetooth settings and delete the device manually after disconnecting.

Tap on the GATT DB widget to open the browser. Then tap on the Unknown Service (which we know is WicedButton) and then on the Characteristic (which we know is MB1).

Tap the Read button to read the value. Press the button on the kit a few times and then Read again to see the incremented value. Then tap the Notify button to enable notifications. Now each time you press the button the value is shown automatically.

Finally, press back until CySmart disconnects.

# WICED Bluetooth Firmware Architecture

The firmware architecture is the same as was described in the previous chapter. The only difference is that there are additional Stack Management events and GATT Database events that occur.

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

|  |  |  |
| --- | --- | --- |
| **Activity** | **Callback Event Name (both Stack and GATT)** | **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 | GATT\_CONNECTION\_STATUS\_EVT | The callback needs to determine if the event is a connection or a disconnection. For a connection, the connection ID is saved, and pairing is enabled (if a secure link is required). |
| 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  (if secure link is required) | 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. |
| Read Values | GATT\_ATTRIBUTE\_REQUEST\_EVT 🡪 GATTS\_REQ\_TYPE\_READ | The firmware must get the value from the correct location in the GATT database. |
| Write Values | GATT\_ATTRIBUTE\_REQUEST\_EVT 🡪 GATTS\_REQ\_TYPE\_WRITE | The firmware must store the provided value in the correct location in the GATT database. |
| Notifications | N/A | Notifications must be sent whenever an attribute that has notifications set is updated by the firmware. Since the change comes from the local firmware, there is no stack or GATT event that initiates this process. |
| Disconnect | GATT\_CONNECTION\_STATUS\_EVT | For a disconnection, the connection ID is reset, all CCCD settings are cleared, and advertisements are restarted. |
| 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 | GATT\_CONNECTION\_STATUS\_EVT | The callback needs to determine if the event is a connection or a disconnection. For a connection, the connection ID is saved, and pairing is enabled (if a secure link is required). |
| 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. |
| Read Values | GATT\_ATTRIBUTE\_REQUEST\_EVT 🡪 GATTS\_REQ\_TYPE\_READ | The firmware must get the value from the correct location in the GATT database. |
| Write Values | GATT\_ATTRIBUTE\_REQUEST\_EVT 🡪 GATTS\_REQ\_TYPE\_WRITE | The firmware must store the provided value in the correct location in the GATT database. |
| Notifications | N/A | Notifications must be sent whenever an attribute that has notifications set is updated by the firmware. Since the change comes from the local firmware, there is no stack or GATT event that initiates this process. |
| 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 | GATT\_CONNECTION\_STATUS\_EVT | The callback needs to determine if the event is a connection or a disconnection. For a connection, the connection ID is saved, and pairing is enabled (if a secure link is required). |
| 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. |
| Read Values | GATT\_ATTRIBUTE\_REQUEST\_EVT 🡪 GATTS\_REQ\_TYPE\_READ | The firmware must get the value from the correct location in the GATT database. |
| Write Values | GATT\_ATTRIBUTE\_REQUEST\_EVT 🡪 GATTS\_REQ\_TYPE\_WRITE | The firmware must store the provided value in the correct location in the GATT database. |
| Notifications | N/A | Notifications must be sent whenever an attribute that has notifications set is updated by the firmware. Since the change comes from the local firmware, there is no stack or GATT event that initiates this process. |
| 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 | GATT\_CONNECTION\_STATUS\_EVT | The callback needs to determine if the event is a connection or a disconnection. For a connection, the connection ID is saved, and pairing is enabled (if a secure link is required). |
| 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. |
| Read Values | GATT\_ATTRIBUTE\_REQUEST\_EVT 🡪 GATTS\_REQ\_TYPE\_READ | The firmware must get the value from the correct location in the GATT database. |
| Write Values | GATT\_ATTRIBUTE\_REQUEST\_EVT 🡪 GATTS\_REQ\_TYPE\_WRITE | The firmware must store the provided value in the correct location in the GATT database. |
| Notifications | N/A | Notifications must be sent whenever an attribute that has notifications set is updated by the firmware. Since the change comes from the local firmware, there is no stack or GATT event that initiates this process. |
| Disconnect | BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT | Advertising on. |

# Low Power

TBD

# WICED Chips & the Architecture of HCI

In many complicated systems, hierarchy is used to manage the complexity. WICED Bluetooth is no different. The WICED Bluetooth Stack is called a Stack because it is a set of blocks that have well defined interfaces. Here is a simple picture of the software system that we have been using. You have been writing code in the block called “Application”. You have made API calls and gotten events from the “Attribute Protocol” and you implemented the “Generic Attribute Profile” by building the GATT Database. Moreover, you advertised using GAP and you Paired and Bonded by using the Security Manager.



## HCI

The next block to talk about is the “Host Controller Interface”.

For technical and cost reasons, when Bluetooth was originally created the Radio was a separate chip from the one that was running the Application. The Radio chip took the name of Controller because it was the Radio and Radio Controller, and the chip running the Application was called the Host because it was hosting the Application.

The interface between the Host and the Controller was typically UART or SPI. The data flying over that serial connection was formatted in Bluetooth SIG specific packets called “HCI Packets”.

By standardizing the HCI interface, it allowed big application processors (like those exiting in PCs and cellphones) to interface with Bluetooth. As time went by the Host and Controller have frequently merged into one chip (e.g. CYW20719), however the HCI interface persists even though both sides may be physically on the same chip. In this case, the HCI layer is essentially just a pass-through.



In some devices, the WICED Bluetooth Stack can be split into a “Host” and a “Controller” part. For example, the PSoC 6 and 4343W Combo Radio is a 2-chip solution that looks like this:



The HCI concept was extended by the WICED Software team to provide a means of communication between the application layer of two chips. They call this interface “WICED HCI”.



## BT Spy

BTSPY is a debugging tool provided by Cypress that can sniff the WICED data packets that are crossing the HCI interface. In a monolithic system, like the 20719, the WICED team created a Virtual HCI interface that can be reflected to a Serial UART. In a split setup, like the 4343W, the HCI interface can be “mirrored” to a one of the Serial Ports.

This tool will be talked about in detail in Chapter 6 – Debugging.

# Exercises

* 1. BLE Notifications

### Introduction

In this exercise, you will add notifications to the CapSense BLE project from the previous chapter.

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

|  |  |  |
| --- | --- | --- |
| **External Event** | **BLE Stack Event** | **Action** |
| Board reset 🡪 | BTM\_LOCAL\_IDENTITY\_KEYS\_REQUEST\_EVT 🡪 | Not used yet |
| BTM\_ENABLED\_EVT 🡪 | Initialize application, start CapSense thread. |
|  | BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT (BTM\_BLE\_ADVERT\_ UNDIRECTED \_HIGH) | 🡨 Start advertising |
| CySmart will now see advertising packets |  |  |
| Connect to device from CySmart 🡪 | GATT\_CONNECTION\_STATUS\_EVT 🡪 | Set the connection ID and enable pairing |
| BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT (BTM\_BLE\_ADVERT\_OFF) |  |
| Read CapSense characteristic while touching buttons 🡪 | GATT\_ATTRIBUTE\_REQUEST\_EVT, GATTS\_REQ\_TYPE\_READ 🡪 | Returns button state |
| 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. |

### Project Creation

1. Copy ch04a/ex02\_ble\_con to ch04b/ex01\_ble\_ntfy. Rename the files and make the necessary name updates.
   1. Hint: Don’t forget to update header file names in the two C files and don't forget to update the source file names in the makefile.
   2. Hint: Change the name from *<inits>\_con* to *<inits>\_ntfy* in the wiced\_bt\_cfg.c file and the ex01\_ble\_ntfy.c file.
   3. Hint: Many function names and variable names start with "ex02\_ble\_con". You can do a global search/replace to change these to "ex01\_ble\_ntfy" if you want them to be consistent with the project name.
   4. Hint: Remove the WICED Bluetooth Designer .wic file since it is no longer a starting point for the project.
2. In the GATT database header file, add a new handle for a Client Characteristic Configuration Descriptor (CCCD) for the CapSense Service, Buttons Characteristic.
   1. Hint: the format is: HDLD\_<service>\_<characteristic>\_CLIENT\_CONFIGURATION <value>.
   2. Hint: use the next free handle value.
3. In the GATT database C file, add the Client Characteristic Configuration Descriptor to the GATT database for the Button Characteristic.
   1. Hint: We are not adding in pairing yet so make sure the CCCD value has the Read and Write Permissions set. That is, don't include *LEGATTDB\_PERM\_AUTH\_WRITABLE* in the permissions.
4. In the GATT database C file, update the Properties for the Buttons Characteristic to enable Notifications.
5. In the main C file, add the CCCD initial value array and add it to the GATT attribute lookup table.
   1. Hint: The CCCD is an array of 2 uint8\_t values.
   2. Hint: Initialize the CCCD value to 0.
6. Declare 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.
7. In the GATT connect handler function:
   1. On a connection add code to:
      1. Save the connection ID to the variable connection\_id. That is:

connection\_id = p\_conn\_status->conn\_id;

* 1. On a disconnection add code to:
     1. Reset connection\_id to 0.
     2. Turn off the CCCD notifications.

1. In the CapSense Thread function, when a button value changes, check to see if there is a connection and if notifications are enabled. If both are true, send the notification.
   1. Hint: There is a bitmask defined called *GATT\_CLIENT\_CONFIG\_NOTIFICATION* which can be used to mask out the bit for notifications.
   2. Hint: the API to send the notification is *wiced\_bt\_gatt\_send\_notification*.
2. In the wiced\_bt\_cfg.c file, change the setting for rpa\_refresh\_timeout from WICED\_BT\_CFG\_DEFAULT\_RANDOM\_ADDRESS\_CHANGE\_TIMEOUT to WICED\_BT\_CFG\_DEFAULT\_RANDOM\_ADDRESS\_NEVER\_CHANGE.

### Testing

1. Create a Make Target and run it to program the project to the board.
2. Open the mobile CySmart app.
   1. Android: Open the app settings and make sure that "Initiate paring after connection" is not checked since we haven't enabled pairing (yet).
   2. iPhone: The iPhone version doesn’t have these settings so we will deal with it later.
3. Connect to the device.
4. Open the CapSense widget and observe the button display while touching the CapSense buttons on the kit. The widget uses notifications to update the button display.
5. Back out of the CapSense widget and open the GATT DB widget. Traverse down to the Button Characteristic and notice that there are now buttons for Read and Notify. Turn on Notify and then press the buttons to observe that changes are reported real-time.
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 All Notifications".
   1. Hint: you can also turn on/off notifications individually by selecting the Client Characteristic Configuration Description attribute and writing a 1 (to enable) or a 0 (to disable) to the LSB.
      1. Hint: Remember that BLE is little-endian so the left-most byte is the LSB.
9. Press the CapSense buttons and observe that the values update real-time due to the notifications.
10. Click on "Disable All Notifications"
11. Press the CapSense buttons again and observe that the values are no longer updated.
12. Click "Disconnect".
    1. BLE Pairing and Security

### Introduction

In this exercise, you will add Pairing to the previous project.

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

|  |  |  |
| --- | --- | --- |
| **External Event** | **BLE Stack Event** | **Action** |
| Board reset 🡪 | BTM\_LOCAL\_IDENTITY\_KEYS\_REQUEST\_EVT 🡪 | Not used yet |
| BTM\_ENABLED\_EVT 🡪 | Initialize application, start CapSense thread. |
|  | BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT (BTM\_BLE\_ADVERT\_ UNDIRECTED \_HIGH) | 🡨 Start advertising |
| CySmart will now see advertising packets |  |  |
| Connect to device from CySmart 🡪 | GATT\_CONNECTION\_STATUS\_EVT 🡪 | Set the connection ID and enable pairing |
| BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT (BTM\_BLE\_ADVERT\_OFF) |  |
| 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 |
| BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT (BTM\_BLE\_ADVERT\_UNDIRECTED\_HIGH) | Re-start advertising |
| Wait for timeout. 🡪 | BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT  (BTM\_BLE\_ADVERT\_ UNDIRECTED \_LOW) | Stack switches to lower advertising rate to save power |
| Wait for timeout. 🡪 | BTM\_BLE\_ADVERT\_STATE\_CHANGED\_EVT  (BTM\_BLE\_ADVERT\_OFF) | Stack stops advertising. |

### Project Creation

1. Copy ch04b/ex01\_ble\_ntfy to ch04b/ex02\_ble\_pair. Rename the files and make the necessary name updates.
   1. Hint: Don’t forget to update header file names in the two C files and don't forget to update the source file names in the makefile.
   2. Hint: Change the name from *<inits>\_ntfy* to *<inits>\_pair* in the wiced\_bt\_cfg.c file and the ex02\_ble\_pair.c file.
   3. Hint: Many function names and variable names start with "ex01\_ble\_ntfy". You can do a global search/replace to change these to "ex02\_ble\_pair" if you want them to be consistent with the project name.
2. Find the call to *wiced\_bt\_set\_pairable\_mode* mode that was commented out earlier and uncomment it.
3. 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;
4. These settings are used to determine the type of security used during pairing. The new settings specify to use a secure connection.
5. In the GATT database C file, update the Button Characteristic Permissions so that Reads require an authenticated link. Update the CCCD Permissions so that Writes require an authenticated link.
   1. Hint: You need both "LEGATTDB\_PERM\_READABLE" and " LEGATTDB\_PERM\_AUTH\_READABLE" to make a Characteristic readable only in an authenticated link. The same goes for " LEGATTDB\_PERM\_WRITE\_REQ" and LEGATTDB\_PERM\_AUTH\_WRITABLE". That is, you will ORing in new permissions but not removing any existing ones.

### Testing

1. Create a Make Target and run it to program the project to the board.
2. Open the mobile CySmart app.
   1. Android: Open the app settings and turn on the settings for "Initiate pairing after connection" and "Delete bond on disconnect".
      1. Hint: The bonding information should be deleted on disconnect because the firmware is not saving bonding information (yet).
   2. iPhone: The iPhone version of CySmart doesn’t have these settings.
3. Connect to the device. You should see a Pairing message once the connection is established.
4. Open the CapSense widget and observe the button display while touching the CapSense buttons on the kit.
5. Disconnect from the mobile CySmart app and start the PC CySmart app.
6. iPhone only: Go to the Bluetooth settings and remove the <inits>\_pair device from the list. This is necessary because the iPhone app does not have the setting to delete bonding information on disconnect.
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. Try reading the CapSense Button Characteristic Value manually. Notice that you again get an authentication error. Click "OK" to close the error window.
10. Click on "Pair" and click "No" when asked if you want to add the device to the resolving list.
11. Click on "Enable All Notifications" again. Now when you touch a button you will see the characteristic value change.
12. Click on "Disable All Nofitications" and then read the CapSense Button Characteristic Value manually. It should now work.
13. Click "Disconnect".

### Questions

1. How long does the device stay in high duty cycle advertising mode? How long does it stay in low duty cycle advertising mode? Where are these values set?
   1. Save BLE Pairing Information (i.e. Bonding)

### Introduction

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 passkey (which will be added in the next exercise) since saving the bonding information means the passkey 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.

The firmware has two "modes": *bonding mode* and *bonded mode*. After programming, the kit will start out in bonding mode. LED1 will blink to indicate that the kit is waiting to be Paired/Bonded. Once a Client connects to the kit and pairs with it, the Bonding information will be saved in non-volatile memory. At this point, the LED will stop flashing. The only Client that will be allowed to pair with the kit is the one that is bonded (the firmware only allows 1 bonded device at a time). If the Bonding information is removed from the Client, it will no longer be able to Pair/Bond with the kit without going through the Paring/Bonding process again.

To remove Bonding information from the kit and return bonding mode, press and release mechanical button MB1. This allows you to Pair/Bond from a Client that has "lost" the bonding information or to Pair/Bond with a new device.

### Project Creation

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>\_bond* where *<inits>* is your initials.

### Testing

1. Open a UART terminal window to the PUART.
2. Build the project and program it to the board.
3. Open the CySmart PC application and connect to the dongle.
4. If there is anything listed in the "Device List" near the bottom of the screen, click on any device from the list and choose "Clear > All". This will remove any stored bonding information from CySmart 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.
5. Start scanning and then connect to your device.
6. Click on "Discover all Attributes".
7. Click on "Pair" and click "No" when asked if you want 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.
8. Click on "Enable All Notifications". Touch the CapSense buttons and observe the characteristic value change.
9. Click "Disconnect". Do NOT remove the device from the Device List this time – we want bonding information retained.
10. Re-connect to your device.
11. Click on "Discover all Attributes" and "Pair". Again, say "No" to the question regarding the resolving list.
    1. Once again note down the Bluetooth Stack events that occur during pairing. You will notice that fewer steps are required this time.
12. Note that notifications are already enabled since they were enabled when you disconnected.
13. Disconnect again.
14. Reset or power cycle the board.
    1. Hint: If you power cycle the board, you will need to either reset or re-open the UART terminal window.
15. Start scanning and then connect to your device for a third time.
16. Click on "Discover all Attributes" and "Pair". Again, just say "No".
    1. Note down the Bluetooth Stack events that occur this time during pairing. Compare to the previous two connections.
17. Note that notifications are still enabled.
18. Disconnect again.
19. Clear the Device List and then reconnect. "Discover all Attributes" and then try to "Pair". Note that paring will not complete because CySmart no longer has the required keys to use.
    1. Hint: If you look in the UART window you will see a message about the security request being denied.
20. Click on Disconnect and close the Authentication failed message window.
21. Press MB1 on the kit and note that LED1 begins flashing. This indicates that the bonding information has been cleared.
22. Connect, Discover Attributes, and Pair again. This time it should work.
23. Note the steps that the firmware goes through this time.
24. Disconnect a final time and 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.

### Overview of Changes

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 and the variable called "bond\_mode" is set as FALSE. Otherwise, it stays TRUE, which means the device can accept new pairing requests.
3. In the BTM\_SECURITY\_REQUEST\_EVENT look to see if bond\_mode is TRUE. Security is only granted if the device is in bond\_mode.
4. In the Bluetooth stack event *BTM\_PAIRING\_COMPLETE\_EVT* if bonding was successful write the information from the hostinfo structure into the NVRAM and set bond\_mode to FALSE.
   1. This saves hostinfo upon initial pairing. This event is not called when bonded devices reconnect.
5. In the Bluetooth stack event *BTM\_ENCRYPTION\_STATUS\_EVT*, if the device is bonded (i.e. bond\_mode is FALSE), read bonding information from the NVRAM into the hostinfo structure.
   1. This reads hostinfo upon a subsequent connection when devices were previously bonded.
6. In the Bluetooth stack event BTM\_PAIRED\_DEVICE\_LINK\_KEYS\_UPDATE\_EVT, save the keys for the peer device to NVRAM.
7. In the Bluetooth stack event BTM\_PAIRED\_DEVICE\_LINK\_KEYS\_REQUEST\_EVT, read the keys for the peer device from NVRAM.
8. In the Bluetooth stack event BTM\_LOCAL\_IDENTITY\_KEYS\_UPDATE\_EVT, save the keys for the local device to NVRAM.
9. In the Bluetooth stack event BTM\_LOCAL\_IDENTITY\_KEYS\_REQUEST\_EVT, read the keys for the local device from NVRAM.
10. 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.
11. 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.
12. An interrupt is used on the GPIO connected to MB1. The ISR sets bond\_mode to TRUE, removes the bonded device from the list of bonded devices, removes the device from the address resolution database, and clears out the bonding information stored in NVRAM.
13. Finally, a Thread is used to control LED blinking based on the state of bond\_mode.

### Questions

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 Passkey

### Introduction

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

### Project Creation

1. Copy ex03\_ble\_bond to ex04\_ble\_pass. Rename the files and make the necessary updates.
   1. Hint: Change the name from *<inits>\_bond* to *<inits>\_pass* 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 and don't forget the source file names in the makefile.
   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\_pass" 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

### Testing

1. Create a Make Target and run it to program the project to the board.
2. Open a UART terminal window.
3. Open the mobile CySmart app.
   1. Android: Open the settings. Turn off "Delete bond on disconnect" and turn on "Initiate pairing after connection".
   2. iPhone: (YFS: Nothing needed here?)
4. 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.
5. Once Pairing and Bonding completes, verify that the application still works.
6. Disconnect and reconnect. Observe that the key does not need to be entered to Pair this time.
7. Delete bonding information:
   1. Android: Check the setting "Delete bond on disconnect", then disconnect.
   2. iPhone: Disconnect, then manually remove the bonding information from the phone's Bluetooth settings. (YFS: verify)
8. Press MB1 to put the kit into Bonding mode and then reconnect. Observe that the key must be entered again to connect.
9. Now try the same thing using the PC version of CySmart. It will pop up a window when the Passkey is needed.
   1. Hint: Remember to put the kit into Bonding mode first to remove the phone's Bonding information.

### 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?