Chapter 4B: More WICED BLE Peripherals

Time 2 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, …

4B.1 Notify & Indicate 2

4B.2 Other Characteristic Descriptors 3

4B.3 Security 4

4B.3.1 Pairing 5

4B.3.2 Bonding 5

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 6

4B.5 WICED Configuration: Buffer Pools 7

4B.6 Advertising packet 7

4B.6.1 Using the Advertising Packet to Get Connected 7

4B.6.2 iBeacon 8

4B.6.3 Eddystone 9

4B.7 GATT Service Discovery 9

4B.8 WICED Bluetooth Stack Events 10

4B.8.1 More Bluetooth Management Events 10

4B.8.2 More GATT Events 11

4B.9 WICED Chips & the Architecture of HCI 11

4B.9.1 HCI 11

4B.9.2 BT Spy 12

4B.10 WICED Bluetooth 201 14

4B.10.1 Low Power 14

4B.10.2 L2CAP 14

4B.10.3 Other Profiles 14

4B.10.4 Whitelist 14

4B.10.5 Scan Response 14

4B.10.6 Central 14

4B.10.7 GATT MTU 14

4B.10.8 Mesh 14

4B.10.9 Non-GATT DB Based Attribute Protocols 14

4B.10.10 Privacy 14

4B.10.11 OTA Bootloading 14

4B.10.12 Multirole devices 14

4B.10.13 Direct Test Mode 14

4B.11 Exercises 15

Exercise - 4B.1 Projects 15

# 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, two things need to happen. First, the Server must add a new Attribute to the Characteristic Descriptors 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
  + UID\_DESCRIPTOR\_CLIENT\_CHARACTERISTIC\_CONFIGURATION
  + LEGATTDB\_PERM\_READABLE | LEGATTDB\_PERM\_WRITE\_REQ | LEGATTDB\_PERM\_AUTH\_WRITABLE | LEGATTDB\_PERM\_AUTH\_READABLE

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

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 exchanging 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 encrypt the Shared Secret using a 6-digit PIN Code. Which gives the eavesdropper a one in a million change at guessing your PIN.

But where do you get the PIN Code? The answer to that question is that there are three ways, depending on what I/O capability you have on the devices:

Method 1 is called “Just works”. Both sides assume a default PIN code of 000000. Obviously, this isn’t secure at the start (because it can be eavesdropped) but after the Shared Secret is exchanged it becomes secure. If you don’t overhear the Shared Secret exchange, then you don’t have the Shared Secret.

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 “Passkey Entry”. For this method to work one side needs to be able to display a random 6-digit number and the other side needs to be able to either enter the number or confirm the number on the display is the expected value.

The Pairing process starts with Device 1 sending a pairing request. That request includes its I/O capabilities. The input possibilities are none, yes/no, and keyboard. The output capabilities are none and display. Device 2 responds with the same information.

Now one side of the connection can display the PIN, and the other side can type it in. Then the encryption and key exchange process can happen.

## 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.

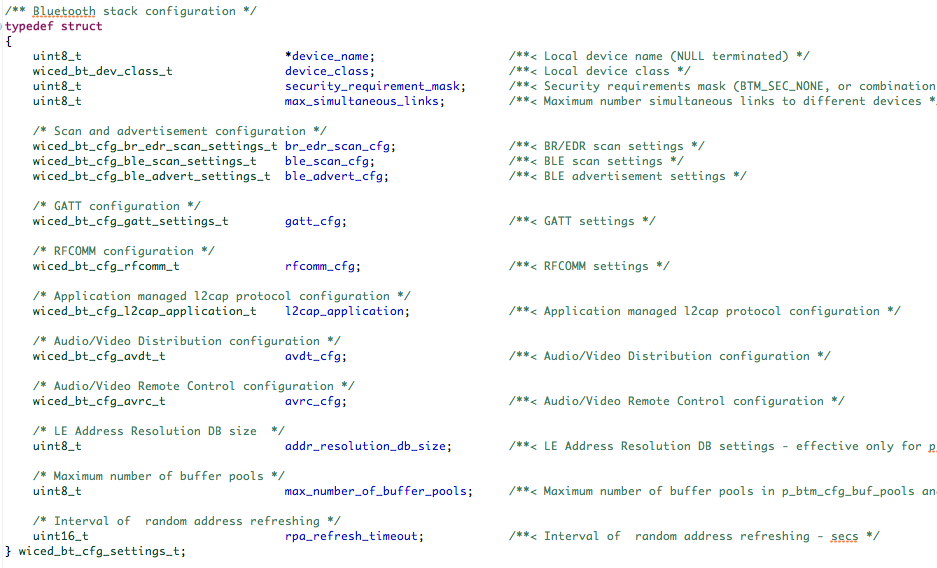
## Privacy

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.

# 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 :



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.

## 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 theniterate through the responses to figure out the Handles, UUIDs etc.

# WICED Bluetooth Stack Events

In the previous chapter I showed you the basic Bluetooth Stack events required to make a connection and exchange data. In order to add in additional functionality described in this chapter such as pairing, bonding, and passkey exchange, there are more events that will come into play as listed below.

## More Bluetooth Management Events

|  |  |
| --- | --- |
| **Event** | **Description** |
| BTM\_LOCAL\_IDENTITY\_KEYS\_REQUEST\_EVT | Restore local identity keys from NVRAM |
| BTM\_LOCAL\_IDENTITY\_KEYS\_UPDATE\_EVT | Save newly created local identity keys to NVRAM |
| BTM\_SECURITY\_REQUEST\_EVT | Request to initiate pairing |
| BTM\_USER\_CONFIRMATION\_REQUEST\_EVT | You should request that the user confirm the PIN |
| BTM\_PASSKEY\_NOTIFICATION\_EVT | You have received a PIN |
| BTM\_PASSKEY\_REQUEST\_EVT |  |
| BTM\_KEYPRESS\_NOTIFICATION\_EVT |  |
| BTM\_PAIRING\_IO\_CAPABILITIES\_BLE\_REQUEST\_EVT | This event occurs during pairing when your device must respond with the Input/Output capabilities of your Peripheral |
| BTM\_SECURITY\_REQUEST\_EVT |  |
| BTM\_SECURITY\_FAILED\_EVT |  |
| BTM\_SECURITY\_ABORTED\_EVT |  |
| BTM\_PAIRED\_DEVICE\_LINK\_KEYS\_REQUEST\_EVT | Read the local keys from NVRAM |
| BTM\_ENCRYPTION\_STATUS\_EVT |  |
| BTM\_PAIRING\_COMPLETE\_EVT | The pairing is complete, you are off and running. Save the keys to NVRAM |
| BTM\_PAIRED\_DEVICE\_LINK\_KEYS\_UPDATE\_EVT | You need to save the updated link keys to NVRAM |
| BTM\_USER\_CONFIRMATION\_REQUEST\_EVT |  |

## More GATT Events

None

# 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, SPI or SDIO. The data flying over that serial connection was formatted in Bluetooth SIG specific packets called “HCI Packets”. The WICED Software team extended this packet format and called it “WICED HCI”.

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 WICED HCI interface persists even though both sides may be running on the same chip.



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:



## 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

# WICED Bluetooth 201

## Low Power

## L2CAP

## Other Profiles

### AMS – Apple Media Service

### ANS – Alert Notification Service

### BAS – Battery Service

#### Example Server Project BAS

#### Example Client Project is BAC

### HRS – Heart Rate Service

#### Example Server HRS

#### Example Client HRC

### ANCS – Apple Notification Center Service

### HID – Human Interface Device

## Whitelist

## Scan Response

## Central

## GATT MTU

## Mesh

## Non-GATT DB Based Attribute Protocols

## Privacy

## OTA Bootloading

## Multirole devices

## Direct Test Mode

# 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.
   2. In the GATT database C file, update the Properties for the Buttons Characteristic to enable Notify.
4. In the main C file, add the CCCD 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.
5. 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.
6. 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.

### 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: YFS: do we need anything here?
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 the 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 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 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. In the GATT connect handler function, on a connection add code to enable pairing.
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. (Is this accurate? What else can we add? Will it already be covered in the chapter?)
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.

### 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: YFS: do we need anything here? I think you just need to clear it manually from the Bluetooth settings at the disconnect step later, right?
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. (YFS: is this correct?).
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 "Yes" 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.
13. Click "Disconnect".
14. From the Device List window (left side below the Discovered devices list), click on any device listed and select "Clear -> All". This will remove any bonding information since we are not saving bonding information on the device (yet).
    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 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.

### 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, 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.
5. Start scanning and then connect to your device.
6. Click on "Discover all Attributes".
7. 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.
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. Start scanning again and re-connect to your device.
11. 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.
12. Note that notifications are enabled since they were enabled when you disconnected.
13. Disconnect again.
14. Power cycle the board. Once reconnected, 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.
    1. Note down the Bluetooth Stack events that occur this time during pairing. Compare to the previous two connections.
17. Disconnect from the 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.

### 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.
   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 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 displayed over the UART. The passkey will need to be entered in CySmart or in your Phone's Bluetooth connection settings before pairing 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 connect 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. Reset the kit and reconnect. Observe that the key must be entered again to connect.
   1. Hint: Resetting the kit is needed because if you don't it will expect the phone to already have the Bonding information and it will reject the connection because it thinks your phone is an imposter. (GJL: why does resetting the kit work? It seems like it should not. Is this a bug in the FW? Is something not restored correctly either on powerup or on connection to the same BD\_ADDR?)(Actually on my Android phone I don't have to reset the board for it to ask for the code again so that behaves differently than the PC app).
9. 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. Eddystone

TBD…

* 1. iBeacon

TBD…