Chapter 4: More WICED BLE Peripherals

Time 2 Hours

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

3.1 Notify & Indicate 2

3.2 Other Characteristic Descriptors 3

3.1 Security 4

3.1.1 Pairing 5

3.1.2 Bonding 5

3.1.3 Pairing & Bonding Process Summary 6

3.1.4 Authentication, Authorization and the GATT DB 6

3.1.5 Privacy 6

3.2 WICED Configuration: Wiced\_bt\_cfg.c 6

3.3 WICED Configuration: Buffer Pools 6

3.4 Advertising packet 6

3.4.1 Using the Advertising Packet to Get Connected 7

3.4.2 iBeacon 8

3.4.3 Eddystone 8

3.5 GATT Service Discovery 8

3.6 WICED Bluetooth Events 9

3.6.1 More Bluetooth Management Events 9

3.6.2 More GATT Events 9

3.7 WICED Chips & the Architecture of HCI 9

3.7.1 HCI 9

3.7.2 BT Spy 11

3.8 Projects 11

3.9 WICED Bluetooth 201 12

3.9.1 Low Power 12

3.9.2 L2CAP 12

3.9.3 Other Profiles 12

3.9.4 Whitelist 12

3.9.5 Scan Response 12

3.9.6 Central 12

3.9.7 GATT MTU 12

3.9.8 Mesh 12

3.9.9 Non-GATT DB Based Attribute Protocols 12

3.9.10 Privacy 12

3.9.11 OTA Bootloading 12

3.9.12 Multirole devices 12

3.9.13 Direct Test Mode 12

3.10 Terms 12

# 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. For instance, 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 question 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 Central 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. The Notification messages are not responded to by the Central, and as such are not reliable. If you need a reliable message, you can send an Indication which the Central 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

Then in your GATT Attribute Write Callback you will need to save the value that was written to you. If a one is written to the CCCD from then on, when a value changes in your system, you will be able to send out a new value.

# 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 Authenticate that both sides know who they are talking to, ensure that all access to data is Authorized, Encrypt all message that are transmitted, verify the Integrity of those messages and 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 function, 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 1 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 are exchanged becomes secure. If you don’t over hear 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 e.g. 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 that the number on the display.

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. The 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 over and over again. 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.

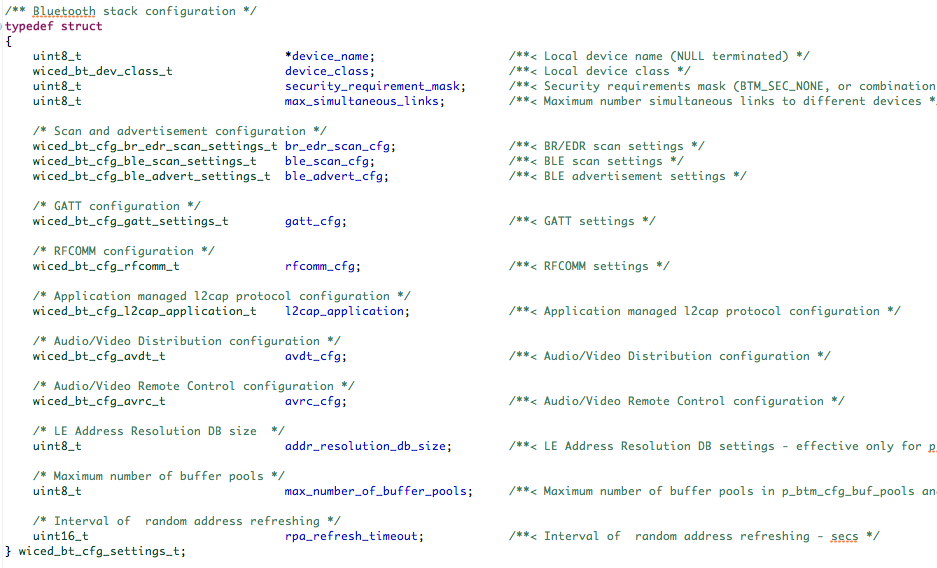
## Privacy

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:



# 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 (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 questions.

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 “Manufacturers 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 unique product ID

The way that this works is that you would write a Central application that would have a table of known Peripheral Product IDs that it knew how to talk to. Then the Peripherals would advertise their custom Product ID and Company code in the Manufacturers Data Field. When a Central saw something that it knew how to talk to it would make the connection.

## iBeacon

iBeacon is an Advertising Packet format defined

## 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 using in mapping functions
* URL – An HTTP URL
* TLM – Telemetry information about the beacon, i.e. battery voltage, device temperate, counts of packet broadcasts
* EID – Ephermal ID packets 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 contain the Eddystone packet type, then the actual data. The packet types are

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

In the 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 3.4.1 ), then you will 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 procedures 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.

As the Central will then know 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, Descriptor 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 this service discovery algorithm by calling the function wiced\_bt\_gatt\_send\_discover to execute the Read Group by Type request. Then iterating through the responses to figure out the Handles, UUID etc.

# WICED Bluetooth Events

## 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 PIN |
| BTM\_PASSKEY\_REQUEST\_EVT |  |
| BTM\_KEYPRESS\_NOTIFICATION\_EVT |  |
| BTM\_PAIRING\_IO\_CAPABILITIES\_BLE\_REQUEST\_EVT | Requesting IO capabilities for BLE pairing |
| BTM\_SECURITY\_REQUEST\_EVT |  |
| BTM\_SECURITY\_FAILED\_EVT |  |
| BTM\_SECURITY\_ABORTED\_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\_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 built, the Radio was a separate chip from the chip that was running your 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 a UART, SPI or SDIO. The data flying over that serial connection was formatted in a Bluetooth SIG specific packets called “HCI Packets”. The WICED Software team extended this packet format and is now called “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. 20719, however the WICED HCI interface persists.



The WICED Bluetooth Stack can be split into a “Host” and a “Controller” part for example the PSoC 6 and 4343W Combo Radio looks like this:



## BT Spy

The BTSPY is a debugging tool provided by Cypress that can sniff the WICED data packets that are crossing the HCI interface. In a monolithic setup, like the 20719, WICED 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

# Projects

Add notify to the CapSense project

Add Paring to the CapSense project

Add Bonding to the CapSense project

Make an Eddystone project and use android to do something

Make an iBeacon project

# 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