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

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.

4B.2   Other Characteristic Descriptors

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

Name	Uniform Type Identifier	Assigned Number	Specification
Characteristic Aggregate Format	org.bluetooth.descriptor.gatt.characteristic_aggregate_format	0×2905	GSS
Characteristic Extended Properties	org.bluetooth.descriptor.gatt.characteristic_extended_properties	0×2900	GSS
Characteristic Presentation Format	org.bluetooth.descriptor.gatt.characteristic_presentation_format	0×2904	GSS
Characteristic User Description	org.bluetooth.descriptor.gatt.characteristic_user_description	0×2901	GSS
Client Characteristic Configuration	org.bluetooth.descriptor.gatt.client_characteristic_configuration	0×2902	GSS
Environmental Sensing Configuration	org.bluetooth.descriptor.es_configuration	0×290B	GSS
Environmental Sensing Measurement	org.bluetooth.descriptor.es_measurement	0×290C	GSS
Environmental Sensing Trigger Setting	org.bluetooth.descriptor.es_trigger_setting	0×290D	GSS
External Report Reference	org.bluetooth.descriptor.external_report_reference	0×2907	GSS
Number of Digitals	org.bluetooth.descriptor.number_of_digitals	0×2909	GSS
Report Reference	org.bluetooth.descriptor.report_reference	0×2908	GSS
Server Characteristic Configuration	org.bluetooth.descriptor.gatt.server_characteristic_configuration	0×2903	GSS
Time Trigger Setting	org.bluetooth.descriptor.time_trigger_setting	0×290E	GSS
Valid Range	org.bluetooth.descriptor.valid_range	0×2906	GSS
Value Trigger Setting	org.bluetooth.descriptor.value_trigger_setting	0×290A	GSS

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

```
 89  enum ble_uuid_characteristic_descriptor
 90  {
 91      UUID_DESCRIPTOR_CHARACTERISTIC_EXTENDED_PROPERTIES    = 0x2900,
 92      UUID_DESCRIPTOR_CHARACTERISTIC_USER_DESCRIPTION       = 0x2901,
 93      UUID_DESCRIPTOR_CLIENT_CHARACTERISTIC_CONFIGURATION   = 0x2902,
 94      UUID_DESCRIPTOR_SERVER_CHARACTERISTIC_CONFIGURATION   = 0x2903,
 95      UUID_DESCRIPTOR_CHARACTERISTIC_PRESENTATION_FORMAT    = 0x2904,
 96      UUID_DESCRIPTOR_CHARACTERISTIC_AGGREGATE_FORMAT       = 0x2905,
 97      UUID_DESCRIPTOR_VALID_RANGE                           = 0x2906,
 98      UUID_DESCRIPTOR_EXTERNAL_REPORT_REFERENCE             = 0x2907,
 99      UUID_DESCRIPTOR_REPORT_REFERENCE                      = 0x2908,
100      UUID_DESCRIPTOR_NUMBER_OF_DIGITALS                    = 0x2909,
101      UUID_DESCRIPTOR_VALUE_TRIGGER_SETTING                 = 0x290A,
102      UUID_DESCRIPTOR_ENVIRONMENT_SENSING_CONFIGURATION     = 0x290B,
103      UUID_DESCRIPTOR_ENVIRONMENT_SENSING_MEASUREMENT       = 0x290C,
104      UUID_DESCRIPTOR_ENVIRONMENT_SENSING_TRIGGER_SETTING   = 0x290D,
105      UUID_DESCRIPTOR_TIME_TRIGGER_SETTING                  = 0x290E,
106  };
```

4B.3   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.

4B.3.1 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.

4B.3.2 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.

4B.3.3 Pairing & Bonding Process Summary
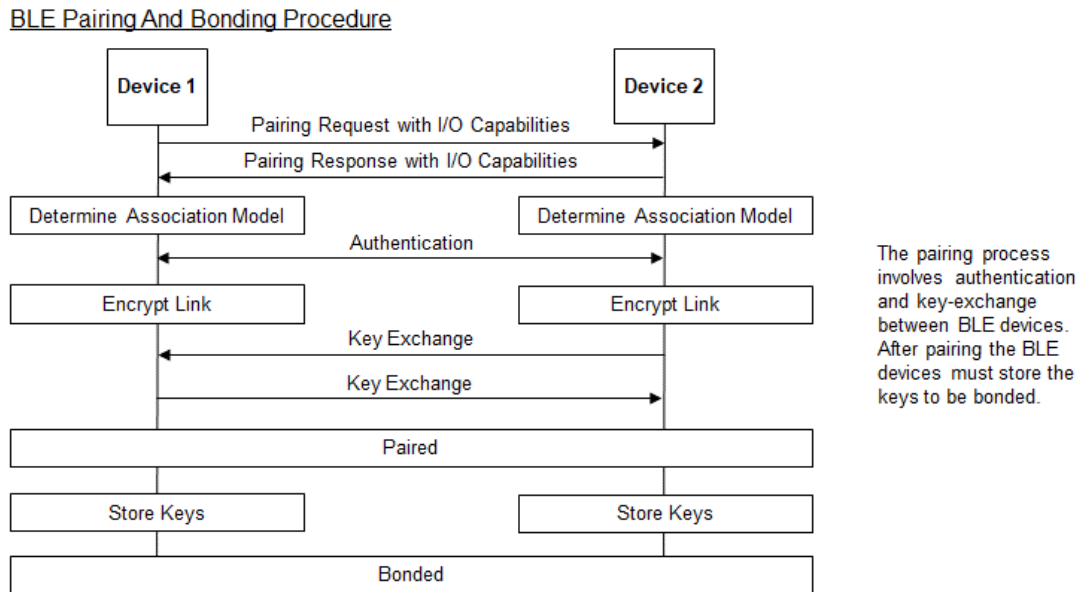
4B.3.4 Authentication, Authorization and the GATT DB

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.

4B.3.5 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.

4B.4   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.

```
/** Bluetooth stack configuration */
typedef struct
{
    uint8_t                         *device_name;               /**< Local device name (NULL terminated) */
    wiced_bt_dev_class_t            device_class;               /**< Local device class */
    uint8_t                         security_requirement_mask;  /**< Security requirements mask (BTM_SEC_NONE, or combination
    uint8_t                         max_simultaneous_links;     /**< Maximum number simultaneous links to different devices *

    /* Scan and advertisement configuration */
    wiced_bt_cfg_br_edr_scan_settings_t br_edr_scan_cfg;        /**< BR/EDR scan settings */
    wiced_bt_cfg_ble_scan_settings_t    ble_scan_cfg;           /**< BLE scan settings */
    wiced_bt_cfg_ble_advert_settings_t  ble_advert_cfg;         /**< BLE advertisement settings */

    /* GATT configuration */
    wiced_bt_cfg_gatt_settings_t        gatt_cfg;               /**< GATT settings */

    /* RFCOMM configuration */
    wiced_bt_cfg_rfcomm_t               rfcomm_cfg;             /**< RFCOMM settings */

    /* Application managed l2cap protocol configuration */
    wiced_bt_cfg_l2cap_application_t    l2cap_application;      /**< Application managed l2cap protocol configuration */

    /* Audio/Video Distribution configuration */
    wiced_bt_cfg_avdt_t                 avdt_cfg;               /**< Audio/Video Distribution configuration */

    /* Audio/Video Remote Control configuration */
    wiced_bt_cfg_avrc_t                 avrc_cfg;               /**< Audio/Video Remote Control configuration */

    /* LE Address Resolution DB size  */
    uint8_t                             addr_resolution_db_size; /**< LE Address Resolution DB settings - effective only for p

    /* Maximum number of buffer pools */
    uint8_t                             max_number_of_buffer_pools; /**< Maximum number of buffer pools in p_btm_cfg_buf_pools an

    /* Interval of  random address refreshing */
    uint16_t                            rpa_refresh_timeout;    /**< Interval of  random address refreshing - secs */
} wiced_bt_cfg_settings_t;
```

4B.5   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:

```
const wiced_bt_cfg_buf_pool_t wiced_bt_cfg_buf_pools[WICED_BT_CFG_NUM_BUF_POOLS] =
{
/*   { buf_size, buf_count, }, */
    { 64,       12,         }, /* Small Buffer Pool */
    { 360,      4,          }, /* Medium Buffer Pool (used for HCI & RFCOMM control messages,
    { 512,      4,          }, /* Large Buffer Pool  (used for HCI ACL messages) */
    { 1024,     2,          }, /* Extra Large Buffer Pool (used for AVDT media packets and mi
};
```

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.
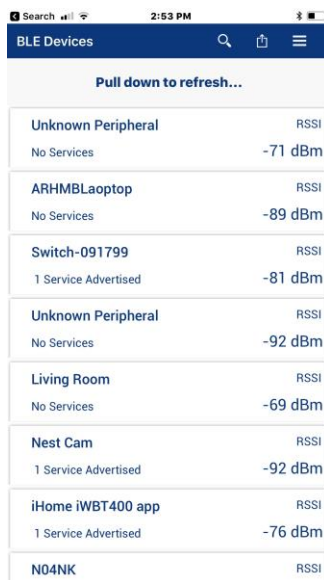
4B.6 Advertising packet

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).

4B.6.1 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:

```
BTM_BLE_ADVERT_TYPE_16SRV_PARTIAL     = 0x02,    /**< List of supported services - 16 bit UUIDs (partial)
BTM_BLE_ADVERT_TYPE_16SRV_COMPLETE    = 0x03,    /**< List of supported services - 16 bit UUIDs (complete)
BTM_BLE_ADVERT_TYPE_32SRV_PARTIAL     = 0x04,    /**< List of supported services - 32 bit UUIDs (partial)
BTM_BLE_ADVERT_TYPE_32SRV_COMPLETE    = 0x05,    /**< List of supported services - 32 bit UUIDs (complete)
BTM_BLE_ADVERT_TYPE_128SRV_PARTIAL    = 0x06,    /**< List of supported services - 128 bit UUIDs (partial)
BTM_BLE_ADVERT_TYPE_128SRV_COMPLETE   = 0x07,    /**< List of supported services - 128 bit UUIDs (complete
```

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.

4B.6.2 iBeacon

iBeacon is an Advertising Packet format defined by Apple.

4B.6.3 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

4B.7   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.

4B.8   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.

4B.8.1 Running the Tool

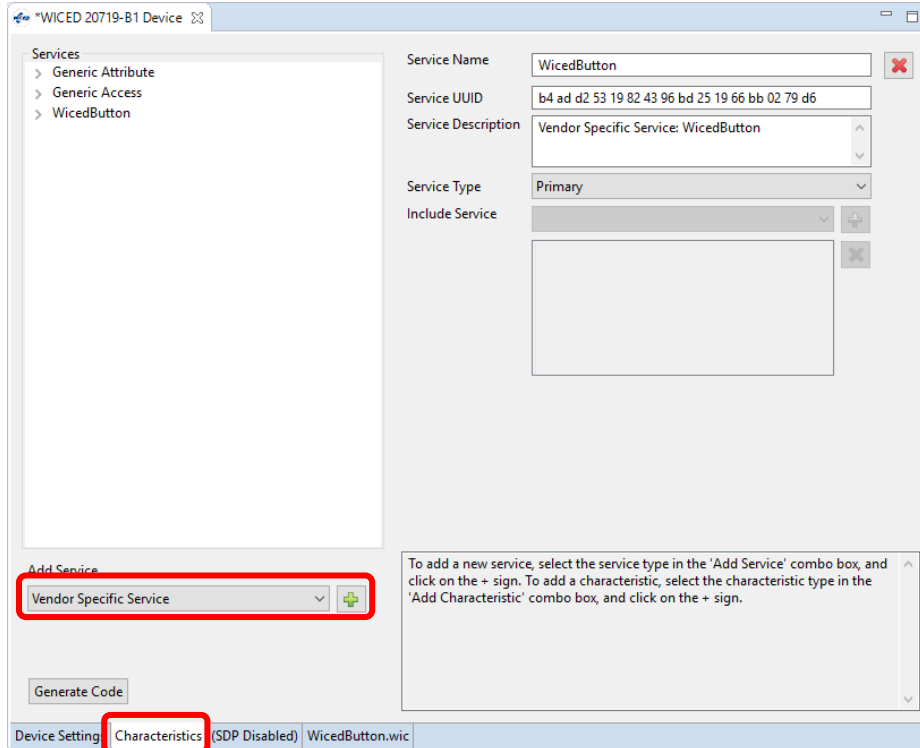Start the tool as before from *File->New->WICED Bluetooth Designer* and enter WicedButton for the device name. Click on Finish to launch the configuration window.
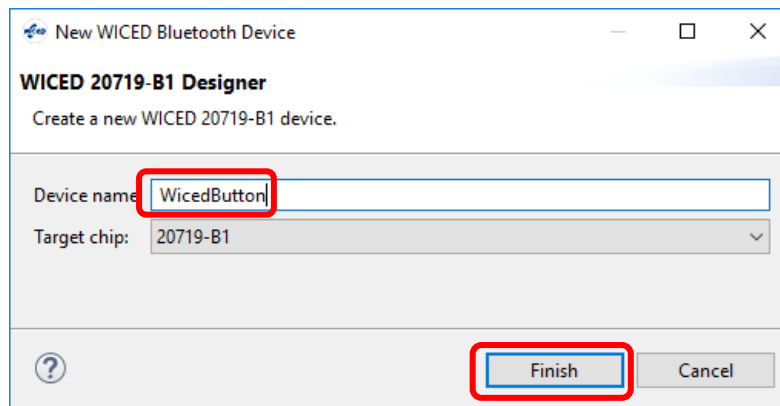
We will keep all the defaults on the Device Settings tab.
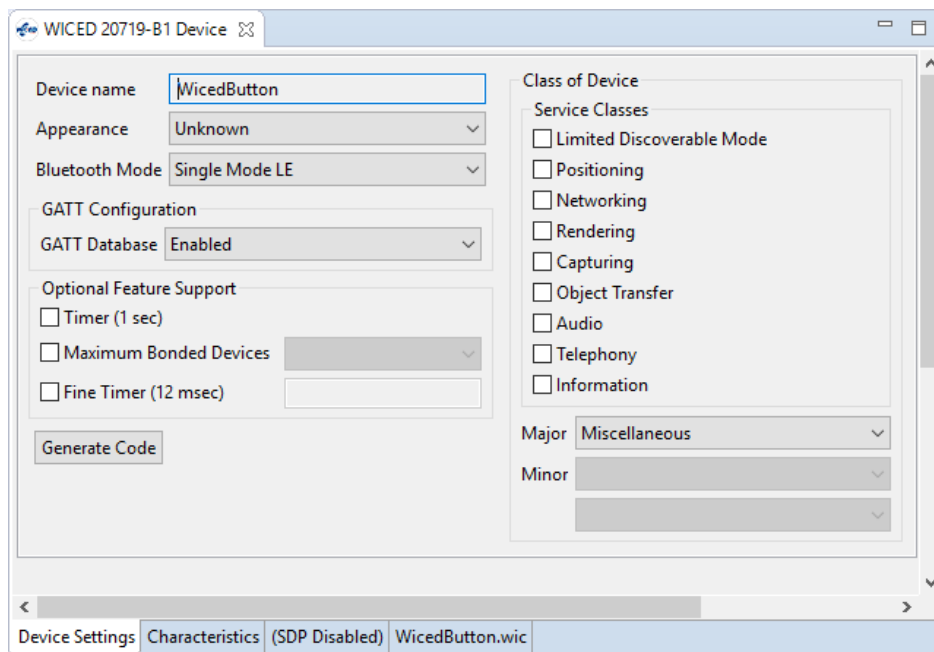
On the Characteristics tab, add a new Vendor Specific Service called WicedButton.

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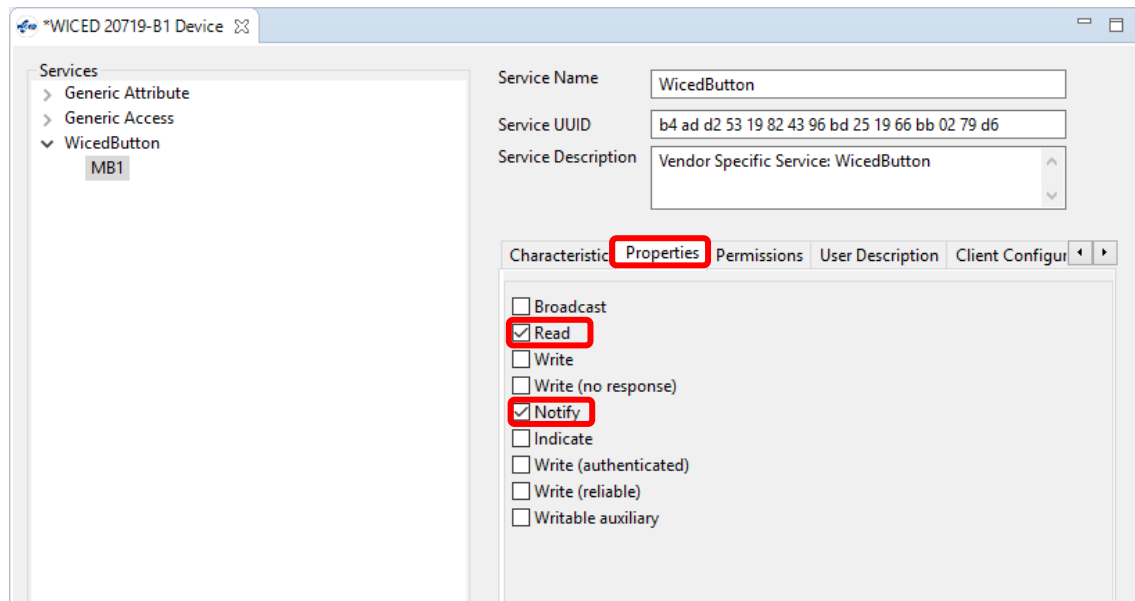
Now check the Permissions tab. It was set by the tool to Read based on our Properties selections. This means that we will be able to Read the Characteristic value without Pairing first. Let's also turn on Read (authenticated) so that Read will require an Authenticated (i.e. Paired) link.

(Note, you must also leave on Read although that does NOT mean that it will be Readable with a non-Authenticated link.)
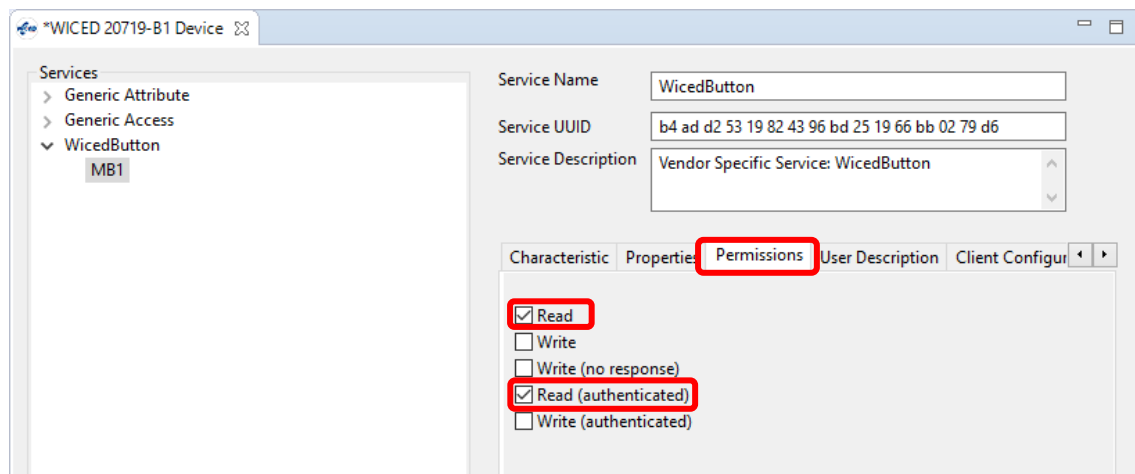
Enabling/disabling Notifications requires a Paired connection by default – it can't be changed in WICED Bluetooth Designer, but you'll see how that can be done in the exercises.

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.
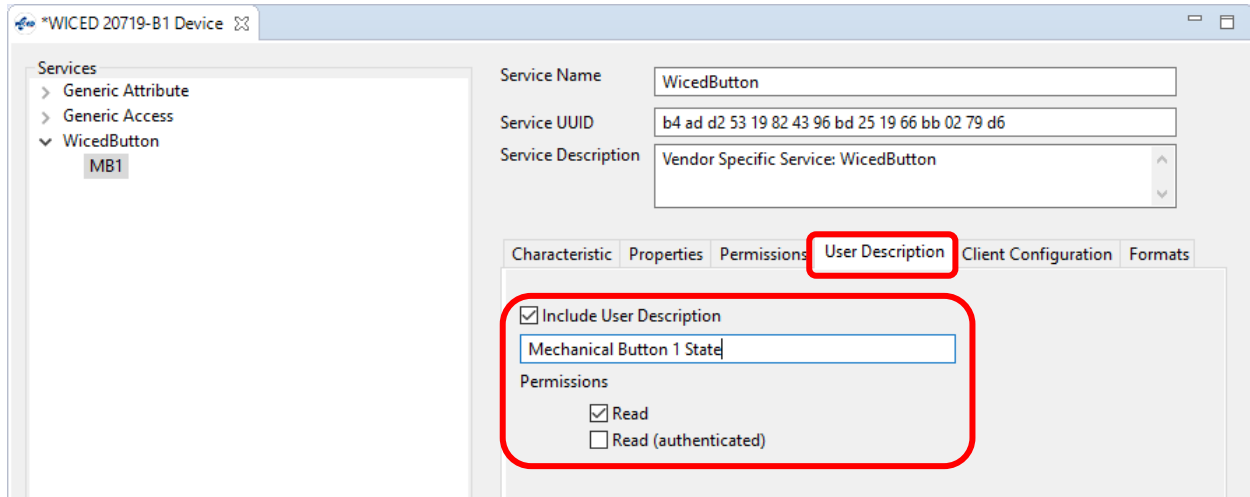
Now, click the Generate Code button.

4B.8.2 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"
```

2. Switch the debug messages to the PUART.

```
void application_start(void)
{
    /* Initialize the transport configuration */
    wiced_transport_init( &transport_cfg );

    /* Initialize Transport Buffer Pool */
    transport_pool = wiced_transport_create_buffer_pool ( TRANS_UART_BUFFER_SIZE, TRANS_UART_BUFFER_COUNT );

#if ((defined WICED_BT_TRACE_ENABLE) || (defined HCI_TRACE_OVER_TRANSPORT))
    /* Set the Debug UART as WICED_ROUTE_DEBUG_NONE to get rid of prints */
    //  wiced_set_debug_uart( WICED_ROUTE_DEBUG_NONE );

    /* Set Debug UART as WICED_ROUTE_DEBUG_TO_PUART to see debug traces on Peripheral UART (PUART) */
    wiced_set_debug_uart( WICED_ROUTE_DEBUG_TO_PUART );

    /* Set the Debug UART as WICED_ROUTE_DEBUG_TO_WICED_UART to send debug strings over the WICED debug interf
    //wiced_set_debug_uart( WICED_ROUTE_DEBUG_TO_WICED_UART );
#endif

    /* Initialize Bluetooth Controller and Host Stack */
    wiced_bt_stack_init(wicedled_management_callback, &wiced_bt_cfg_settings, wiced_bt_cfg_buf_pools);
}
```

3. 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;

4. Configure MB1 as a falling edge interrupt.

```
void wicedbutton_app_init(void)
{
    /* Initialize Application */
    wiced_bt_app_init();

    /* Configure the Button GPIO as an input with a resistive pull up and interrupt on rising edge */
    wiced_hal_gpio_register_pin_for_interrupt( WICED_GPIO_PIN_BUTTON_1, button_cback, NULL );
    wiced_hal_gpio_configure_pin( WICED_GPIO_PIN_BUTTON_1,
            ( GPIO_INPUT_ENABLE | GPIO_PULL_UP | GPIO_EN_INT_FALLING_EDGE ),
            GPIO_PIN_OUTPUT_HIGH );
```

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

```c
/* Interrupt callback function for BUTTON_1  */
void button_cback( void *data, uint8_t port_pin )
{
    /* Increment the button value */
    wicedbutton_wicedbutton_mb1[0] ++;

    /* If the connection is up and if the client wants notifications, send it */
    if ( connection_id != 0)
    {
        if(wicedbutton_wicedbutton_mb1_client_configuration[0] & GATT_CLIENT_CONFIG_NOTIFICATION)
        {
            wiced_bt_gatt_send_notification(connection_id, HDLC_WICEDBUTTON_MB1_VALUE,
                    sizeof(wicedbutton_wicedbutton_mb1), wicedbutton_wicedbutton_mb1 );
            WICED_BT_TRACE( "\tSend Notification: sending CapSense value\r\n");
        }
    }

    /* Clear the GPIO interrupt */
    wiced_hal_gpio_clear_pin_interrupt_status( WICED_GPIO_PIN_BUTTON_1 );
}
```
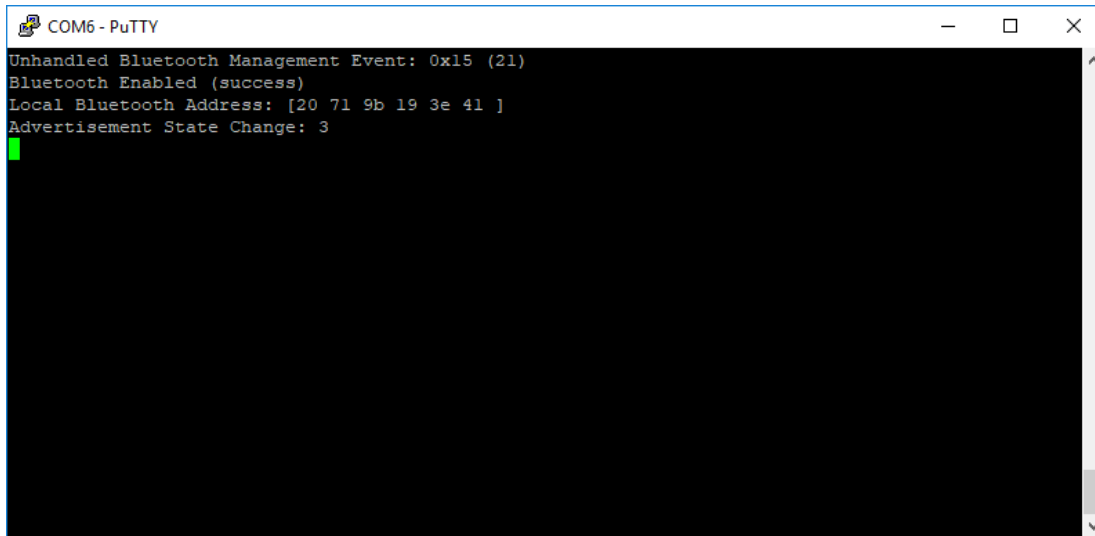
6. Update the Make Target to target for the kit/shield combination platform.

```
Make Target

    wbt101key.ch03.ex02_mutex-WBT101_2_CYW920719Q40EVB_01 download
    wbt101key.ch03.ex03_queue-WBT101_2_CYW920719Q40EVB_01 download
    wbt101key.ch03.ex04_timer-WBT101_2_CYW920719Q40EVB_01 download
    wbt101key.ch04a.ex01_ble_adv-WBT101_2_CYW920719Q40EVB_01 download
    wbt101key.ch04a.ex02_ble_con-WBT101_2_CYW920719Q40EVB_01 download
    wbt101key.ch04b.ex01_ble_ntfy-WBT101_2_CYW920719Q40EVB_01 download
    wbt101key.ch04b.ex02_ble_pair-WBT101_2_CYW920719Q40EVB_01 download
    wbt101key.ch04b.ex03_ble_bond-WBT101_2_CYW920719Q40EVB_01 download
    wbt101key.ch04b.ex04_ble_pass-WBT101_2_CYW920719Q40EVB_01 download
    wbt101key.ch05.ex01_spp-WBT101_2_CYW920719Q40EVB_01 download
    wbt101key.ch06.ex03_btspy-WBT101_2_CYW920719Q40EVB_01 download
    WicedButton-WBT101_2_CYW920719Q40EVB_01 download
    WicedLED-WBT101_2_CYW920719Q40EVB_01 download
```

4B.8.3 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.
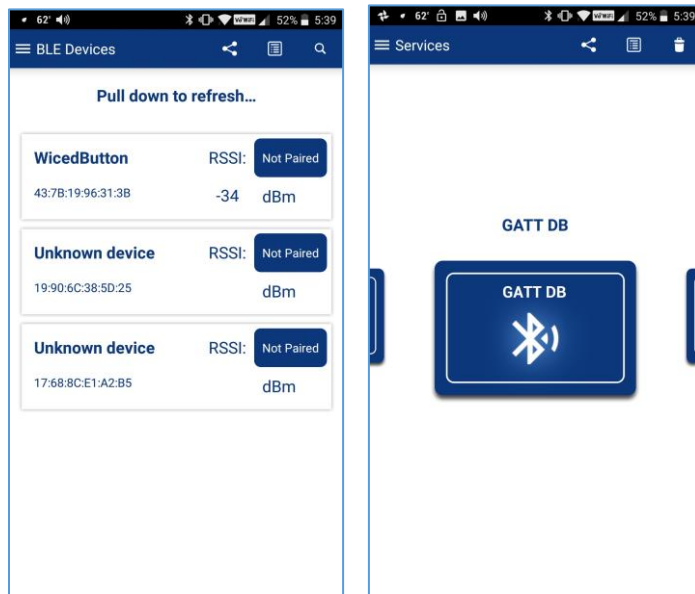
Tap on the GATT DB widget to open the browser. Then tap on the Unknown Service (which we know is WicedButton) and then on the Characteristic (which we know is MB1).
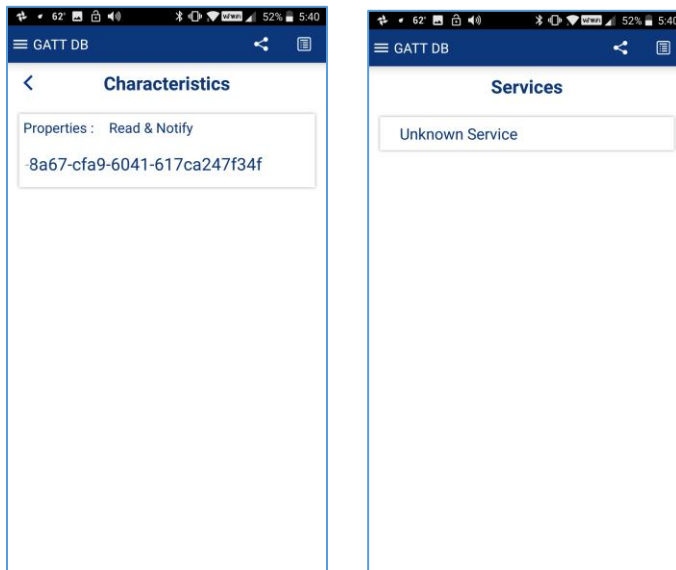
Tap the Read button to read the value. Press the button on the kit a few times and then Read again to see the incremented value. Then tap the Notify button to enable notifications. Now each time you press the button the value is shown automatically.
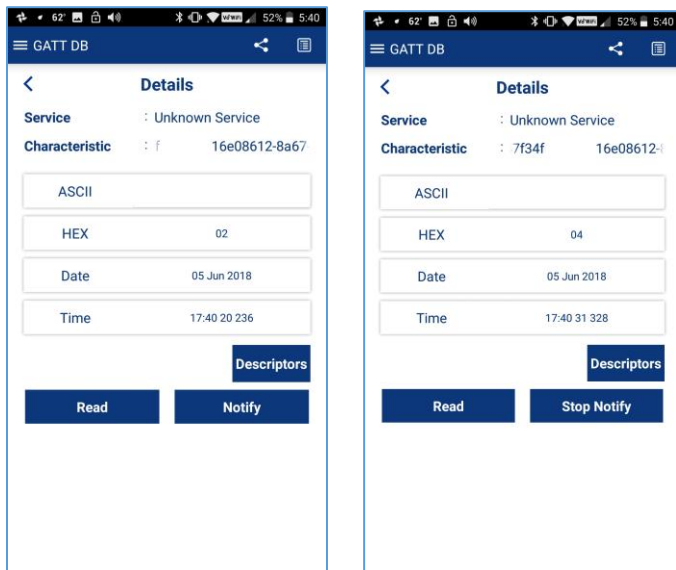
Finally, press back until CySmart disconnects.

4B.9 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?

Activity	Callback Event Name	Reason
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. |

Activity	Callback Event Name	Reason
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,

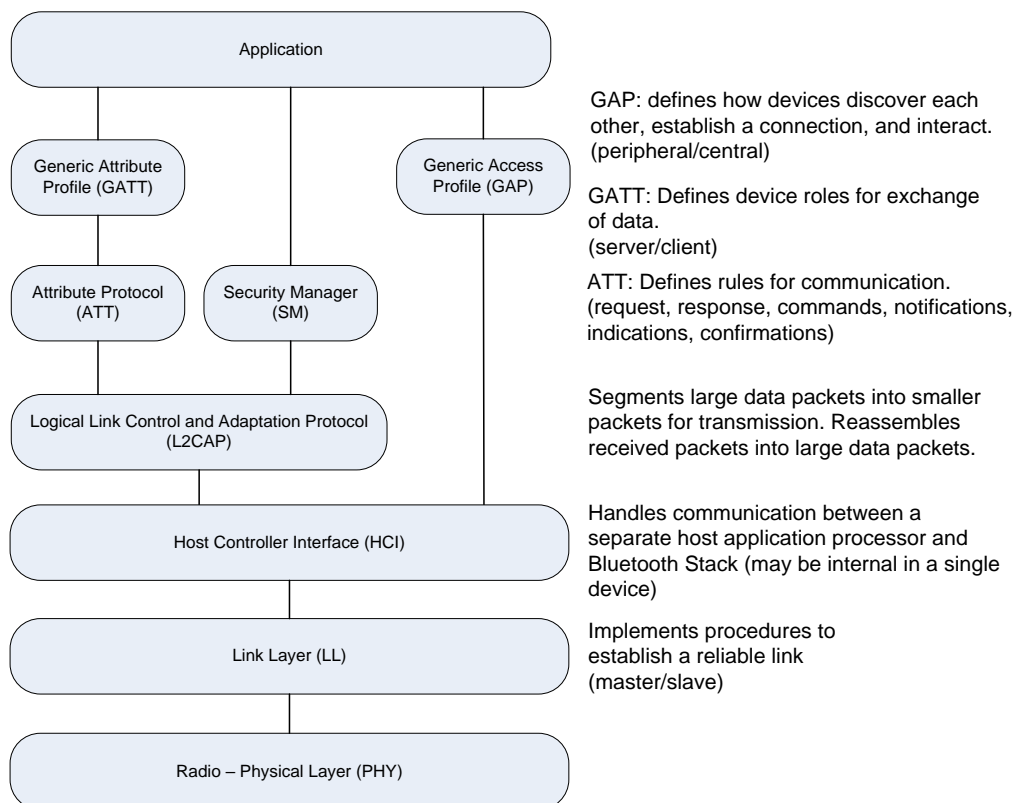| Activity	Callback Event Name	Reason
		there is no stack or GATT event that initiates this process.
Disconnect	BTM_BLE_ADVERT_STATE_CHANGED_EVT	Advertising on.
Reset	BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT	Local keys are loaded from NVRAM.
	BTM_ENABLED_EVT	Stack is enabled. Paired device keys (including 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.

4B.10   Low Power

TBD

4B.11   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.

Application	
Generic Attribute Profile (GATT) / **Generic Access Profile (GAP)**	GAP: defines how devices discover each other, establish a connection, and interact. (peripheral/central)
	GATT: Defines device roles for exchange of data. (server/client)
Attribute Protocol (ATT) / **Security Manager (SM)**	ATT: Defines rules for communication. (request, response, commands, notifications, indications, confirmations)
Logical Link Control and Adaptation Protocol (L2CAP)	Segments large data packets into smaller packets for transmission. Reassembles received packets into large data packets.
Host Controller Interface (HCI)	Handles communication between a separate host application processor and Bluetooth Stack (may be internal in a single device)
Link Layer (LL)	Implements procedures to establish a reliable link (master/slave)
Radio – Physical Layer (PHY)	

4B.11.1 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 <u>Controller</u> because it was the Radio and Radio Controller, and the chip running the Application was called the <u>Host</u> 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 passthrough.

20719	Application
	Host Stack
	HCI
	Controller Stack
	Radio
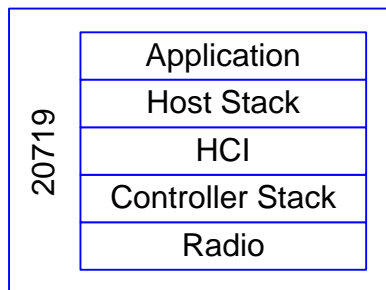
In some devices, the WICED Bluetooth Stack can be split into a "Host" and a "Controller" part. For example, the PSoC 6 and 4343W Combo Radio is a 2-chip solution that looks like this:

| PSoC 6 | Application |
| | Host Stack |

HCI - UART

| 4343W | Controller Stack |
| | Radio |
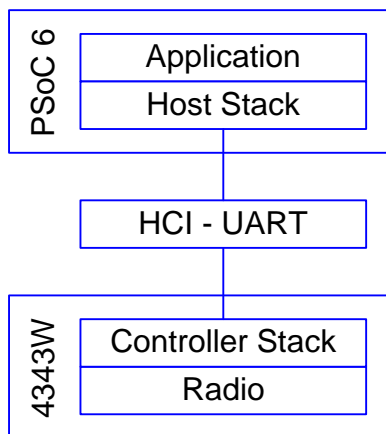
The HCI concept was extended by the WICED Software team to provide a means of communication between the application layer of two chips. They call this interface "WICED HCI".

20719	Application
	Host Stack
	HCI
	Controller Stack
	Radio

WICED HCI - UART

| Host Processor | System Application |
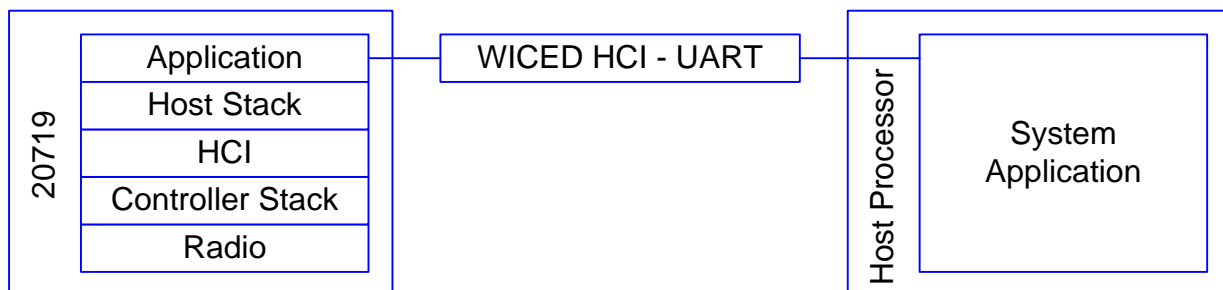
4B.11.2 BT Spy

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.

4B.12　WICED Bluetooth 201

4B.12.1　MTU

4B.12.2　L2CAP

4B.12.3　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

4B.12.4　Whitelist

4B.12.5　Scan Response

4B.12.6　Central

4B.12.7　Mesh

4B.12.8　Non-GATT DB Based Attribute Protocols

4B.12.9　Privacy

4B.12.10　OTA Bootloading

4B.12.11　Multirole devices

4B.12.12　Direct Test Mode

4B.13 Exercises

Exercise - 4B.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.
   a. 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.

b. Hint: Change the name from *<inits>_con* to *<inits>_ntfy* in the wiced_bt_cfg.c file and the ex01_ble_ntfy.c file.

c. 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.

d. 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.

a. Hint: the format is: HDLD_<service>_<characteristic>_CLIENT_CONFIGURATION <value>.

b. 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.

a. 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.

a. Hint: The CCCD is an array of 2 uint8_t values.

b. 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:

a. On a connection add code to:

i. Save the connection ID to the variable connection_id. That is:

connection_id = p_conn_status->conn_id;

b. On a disconnection add code to:

i. Reset connection_id to 0.

ii. Turn off the CCCD notifications.

8. 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.

a. Hint: There is a bitmask defined called *GATT_CLIENT_CONFIG_NOTIFICATION* which can be used to mask out the bit for notifications.

b. Hint: the API to send the notification is *wiced_bt_gatt_send_notification*.

9. 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.
   a. Android: Open the app settings and make sure that "Initiate paring after connection" is not checked since we haven't enabled pairing (yet).
   b. 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 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.
   a. 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".
   a. 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.
      i. 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".

Exercise - 4B.2 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.

a. 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.

b. Hint: Change the name from *<inits>_ntfy* to *<inits>_pair* in the wiced_bt_cfg.c file and the ex02_ble_pair.c file.

c. 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:

   a. p_event_data->pairing_io_capabilities_ble_request.auth_req = BTM_LE_AUTH_REQ_SC_MITM_BOND;

   b. 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.

   a. 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.

   a. Android: Open the app settings and turn on the settings for "Initiate pairing after connection" and "Delete bond on disconnect".

      i. Hint: The bonding information should be deleted on disconnect because the firmware is not saving bonding information (yet).

   c. 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 "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?

Exercise - 4B.3 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.
   a. Create a new make target
   b. 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.
   a. 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 <u>NOT</u> 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.
    a. 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.
    a. 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".
    a. 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.
    a. 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.
    a. 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.
   a. 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.
   a. 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:
    a. 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.
    b. 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?

Exercise - 4B.4 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.
   a. Hint: Change the name from *<inits>_bond* to *<inits>_pass* in the wiced_bt_cfg.c file and the ex04_ble_passkey.c file.
   b. 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.
   c. 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:
   a. Change the value for pairing_io_capabilities_ble_request.local_iop_cap from BTM_IO_CAPABILITIES_NONE to BTM_IO_CAPABILITIES_DISPLAY_ONLY.
      i. 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.
   a. Hint: Make sure you print something around the value so that it is easy to find in the terminal window.
   b. Hint: The Passkey must be 6 digits so print leading 0's if the value is less than 6 digits. (i.e. use %06d).
   c. Hint: The key is passed to the callback as:
      i. 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.
   a. Android: Open the settings. Turn off "Delete bond on disconnect" and turn on "Initiate pairing after connection".
   b. 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:
    a. Android: Check the setting "Delete bond on disconnect", then disconnect.
    b. 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.
    a. 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?