Chapter 4B: More Advanced BLE Peripherals

Time 4 ½ 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	0x2905	GSS
Characteristic Extended Properties	org.bluetooth.descriptor.gatt.characteristic_extended_properties	0x2900	GSS
Characteristic Presentation Format	org.bluetooth.descriptor.gatt.characteristic_presentation_format	0x2904	GSS
Characteristic User Description	org.bluetooth.descriptor.gatt.characteristic_user_description	0x2901	GSS
Client Characteristic Configuration	org.bluetooth.descriptor.gatt.client_characteristic_configuration	0x2902	GSS
Environmental Sensing Configuration	org.bluetooth.descriptor.es_configuration	0x290B	GSS
Environmental Sensing Measurement	org.bluetooth.descriptor.es_measurement	0x290C	GSS
Environmental Sensing Trigger Setting	org.bluetooth.descriptor.es_trigger_setting	0x290D	GSS
External Report Reference	org.bluetooth.descriptor.external_report_reference	0x2907	GSS
Number of Digitals	org.bluetooth.descriptor.number_of_digitals	0x2909	GSS
Report Reference	org.bluetooth.descriptor.report_reference	0x2908	GSS
Server Characteristic Configuration	org.bluetooth.descriptor.gatt.server_characteristic_configuration	0x2903	GSS
Time Trigger Setting	org.bluetooth.descriptor.time_trigger_setting	0x290E	GSS
Valid Range	org.bluetooth.descriptor.valid_range	0x2906	GSS
Value Trigger Setting	org.bluetooth.descriptor.value_trigger_setting	0x290A	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 arriving at the Shared Secret. The basic problem continues to be how do you send a Shared Secret over the air, unencrypted and still have your Shared Secret be Secret. The answer is that you use public key encryption. Both sides have a public/private key pair that is either embedded in the device, or calculated at startup. When you want to authenticate, both sides of the connection exchange public keys. Then both sides exchange encrypted random numbers that form the basis of the shared secret.

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

Method 1 is called "Just works". In this mode you have no protection against MITM.

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

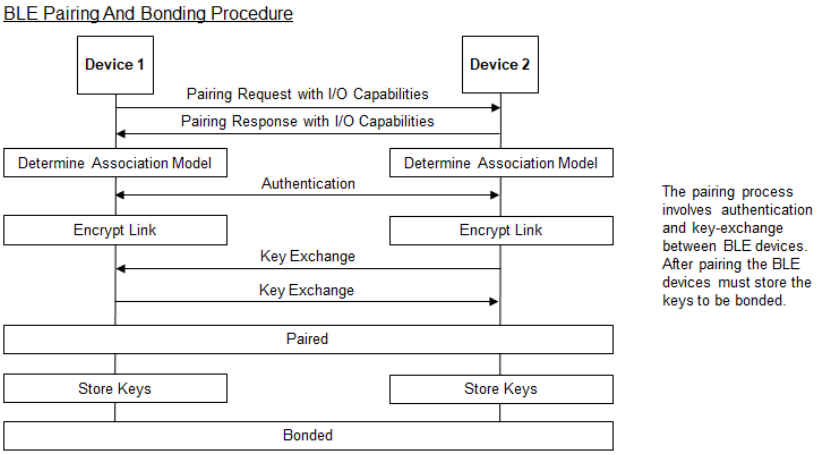
Method 3 is called "Numeric Comparison" (V2.PH.7.2.1). In this method, both sides display a 6-digit number that is calculated with a nasty cryptographic function based on random numbers and the public keys of the devices. The user observes both devices. If the key is the same on both, then the user confirms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

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

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

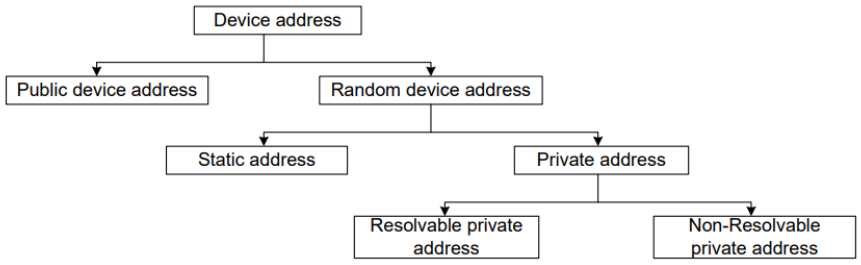
In Chapter 4A3.1 we talked about the Attributes and the GATT Database. Each Attribute has a permissions bit field that includes bits for Encryption, Authentication, and Authorization. The WICED Bluetooth Stack will guarantee that you will not be able to access an Attribute that is marked Encryption or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

BLE devices are identified using a 48-bit device address. This device address is part of all the packets sent by the device in the advertising channels. A third device which listens on all three advertising channels can easily track the activities of a device by using its device address. Privacy is a feature that reduces the ability to track a BLE device by using a private address that is generated and changed at regular intervals.

There are a few different types of address types possible for BLE devices; these are shown in the following table:



The device address can be a Public Device Address or a Random Device Address. The Public Device Addresses are comprised of a 24-bit company ID (an Organizationally Unique Identifier or OUI based on an IEEE standard) and a 24-bit company-assigned number (unique for each device); these addresses do not change over time.

There are two types of Random Addresses: Static Address and Private Address. The Static Address is a 48-bit randomly generated address with the two most significant bits set to 1. Static Addresses change only when the device is power-cycled. A device using a Public Device Address or Static Address can be easily discovered and connected to by a peer device. Private Addresses change at regular intervals to ensure that the BLE device cannot be tracked. A Non-Resolvable Private Address changes on every reconnection. These cannot be resolved by the peer device and must be shared with the peer device during the preceding connection. Resolvable Private Addresses (RPA) can be changed at regular intervals, can be resolved and are used by Privacy-enabled devices.

Every Privacy-enabled BLE device has a unique address called the Identity Address and an Identity Resolving Key (IRK). The Identity Address is the Public Address or Static Address of the BLE device. The IRK is used by the BLE device to generate its RPA and is used by peer devices to resolve the RPA of the BLE device. Both the Identity Address and the IRK are exchanged during the third stage of the pairing process. Privacy-enabled BLE devices maintain a list that consists of the peer device's Identity Address, the local IRK used by the BLE device to generate its RPA, and the peer device's IRK used to resolve the peer device's RPA. This is called the Resolving List. Only peer devices that have the 128-bit identity resolving key (IRK) of a BLE device can connect to it.

A Privacy-enabled BLE device periodically changes its RPA to avoid tracking. The BLE Stack configures the Link Layer with a value called RPA Timeout that specifies the time after which the Link Layer must generate a new RPA. In WICED Studio, this value is set in wiced_bt_cfg.c and is called rpa_refresh_timeout. If the rpa_refresh_timeout is set to 0 (i.e. *WICED_BT_CFG_DEFAULT_RANDOM_ADDRESS_NEVER_CHANGE*), privacy is disabled, and a public device address will be used.

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.

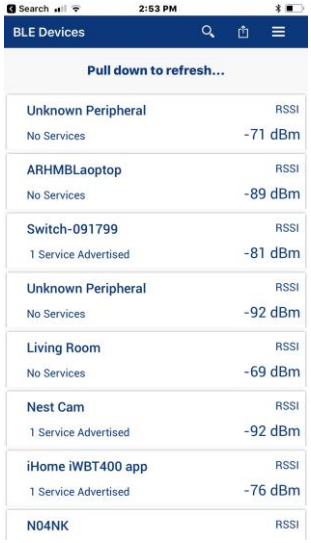
4B.6 Advertising packet

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 quite a few 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. The iBeacon information is embedded in the Manufacturer section of the advertising packet. It simply contains:

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

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

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

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.

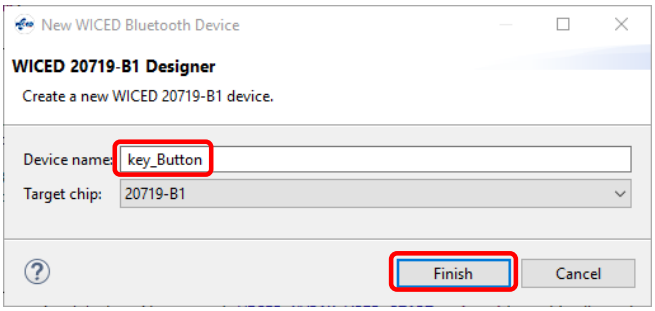
4B.8 WICED Bluetooth Designer

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

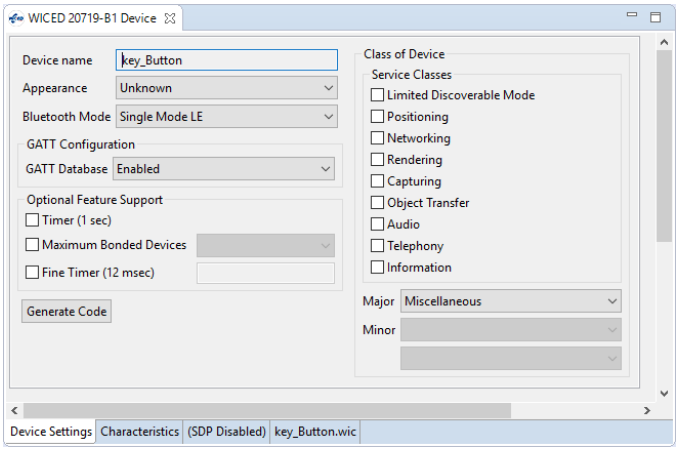
For this example, I'm going to build a BLE project that has a custom Service called WicedButton containing one Characteristic called MB1. This MB1 Characteristic will count the number of times mechanical button MB1 on the shield has been pressed. It will be Readable by the Client and it will send Notifications if the Client enables them.

4B.8.1 Running the Tool

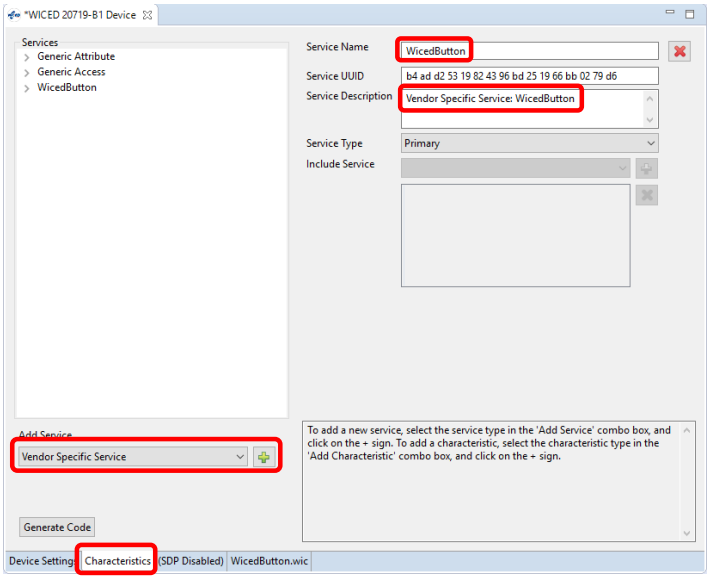
Start the tool from *File->New->WICED Bluetooth Designer*. I'll use a Device name of key_Button. **When you do this yourself, use a unique name such as *<inits>_Button* where *<inits>* is your initials**. Otherwise you will have trouble finding your specific device among all the ones that are advertising. Click on Finish to launch the configuration window.



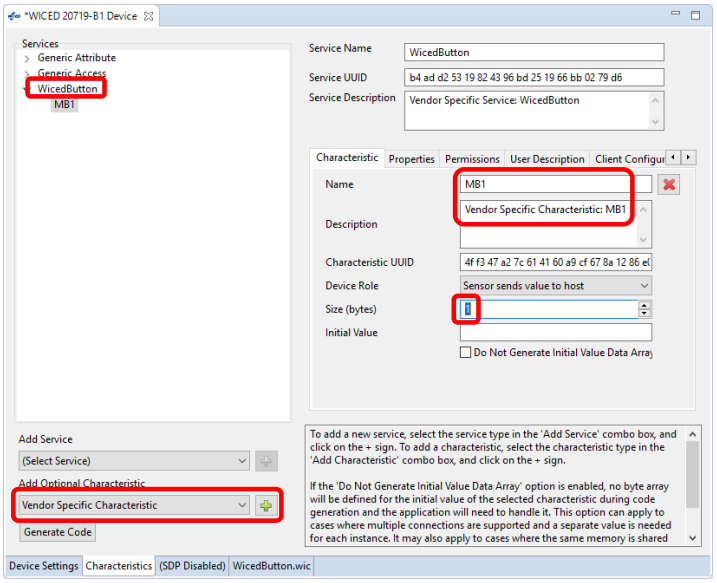
We will keep all the defaults on the Device Settings tab.



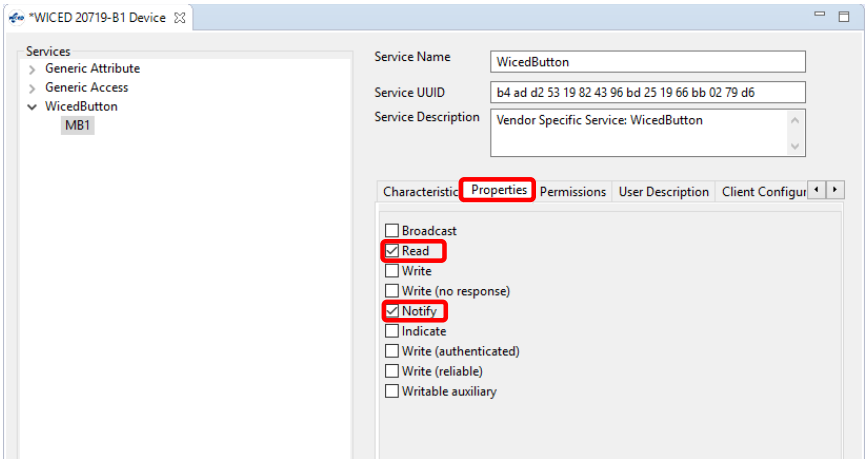
On the Characteristics tab, add a new Vendor Specific Service and change its name to WicedButton.



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

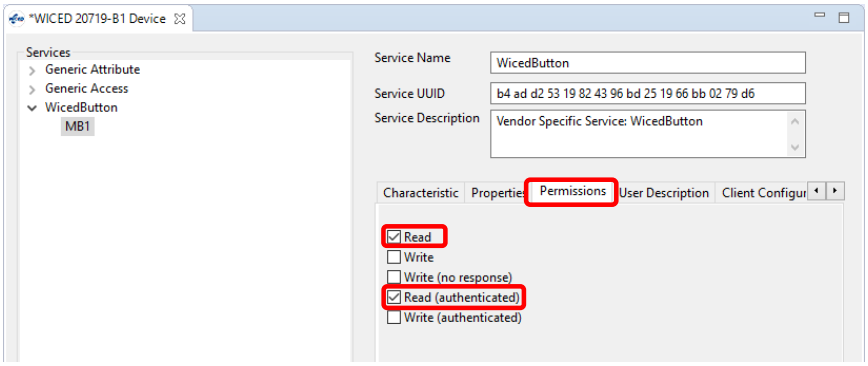


On the Properties tab, we want this Characteristic to have Read and Notify selected.



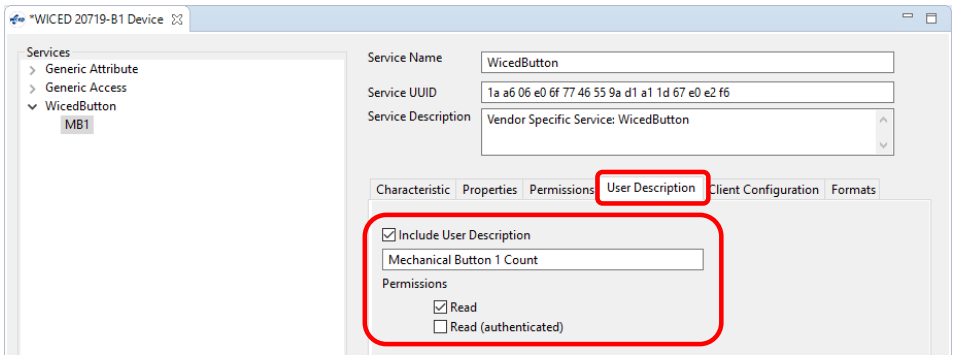
Now check the Permissions tab. It was set by the tool to Read based on our Properties selections. This means that we will be able to Read the Characteristic value without Pairing first. Let's also turn on Read (authenticated) so that Read will require an Authenticated (i.e. Paired) link.

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



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 Count". We will allow this to be read without an Authenticated link.



Now, click the Generate Code button.

4B.8.2 Editing the Firmware

In <inits>_Button.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 interfa
    //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 <inits>_button_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 key_button_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. Create the button callback function. In the callback we will increment the Characteristic value, and then send a notification if we have a connection and the notification is enabled.

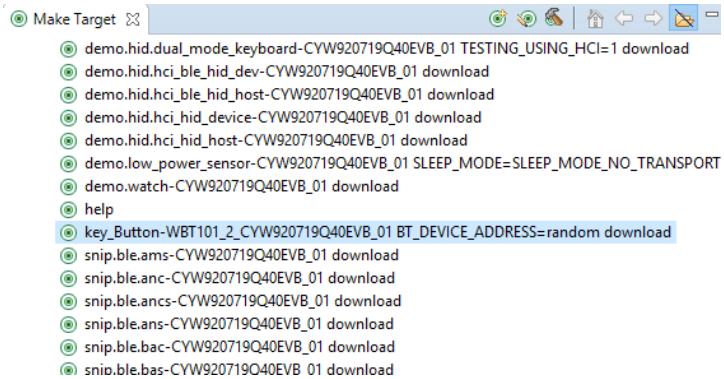
```
/* Interrupt callback function for BUTTON_1  */
void button_cback( void *data, uint8_t port_pin )
{
    /* Increment the button value */
    key_button_wicedbutton_mb1 [0] ++;

    /* If the connection is up and if the client wants notifications, send it */
    if ( connection_id != 0)
    {
         if(key_button_wicedbutton_mb1_client_configuration[0] & GATT_CLIENT_CONFIG_NOTIFICATION)
        {
            wiced_bt_gatt_send_notification(connection_id, HDLC_WICEDBUTTON_MB1_VALUE,
                    sizeof(key_button_wicedbutton_mb1), key_button_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 );
}
```

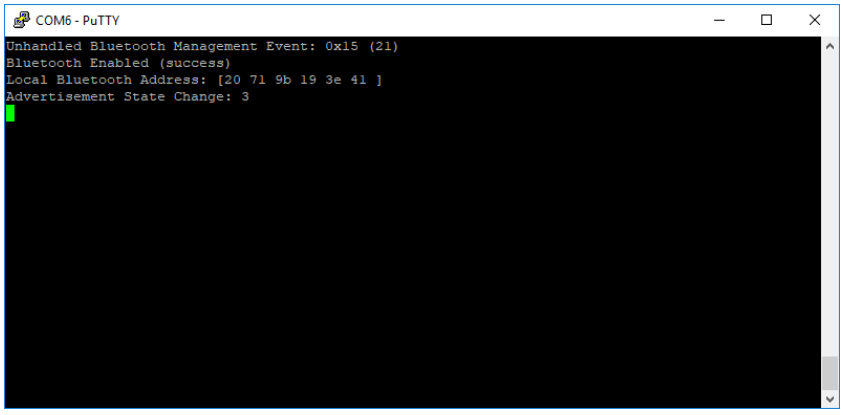
6. In wiced_bt_cfg.c, change the rpa_refresh_timeout to *WICED_BT_CFG_DEFAULT_RANDOM_ADDRESS_NEVER_CHANGE* so that privacy is disabled.

7. Update the Make Target to target for the kit/shield combination platform and add the option BT_DEVICE_ADDRESS=random.

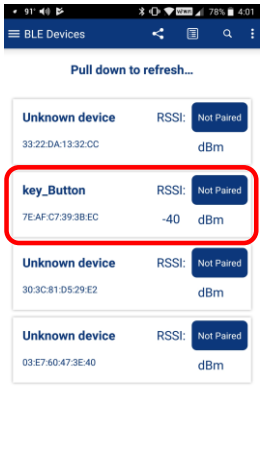


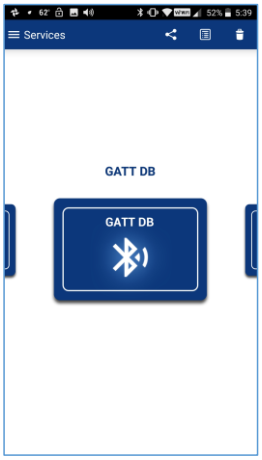
4B.8.3 Testing the Project

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.

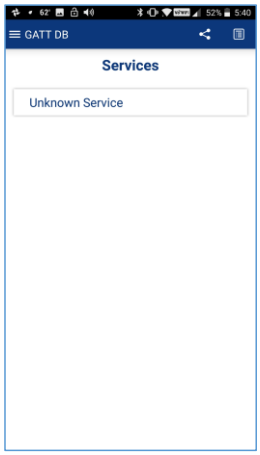


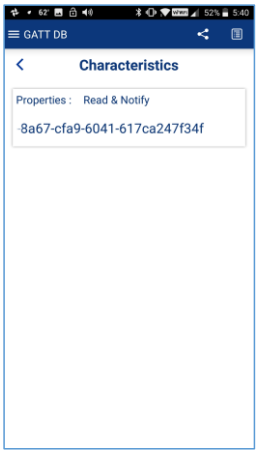
Run CySmart on your phone. When you see the "<inits>_Button" device, tap on it. CySmart will connect to the device and will show the GATT browser widget.



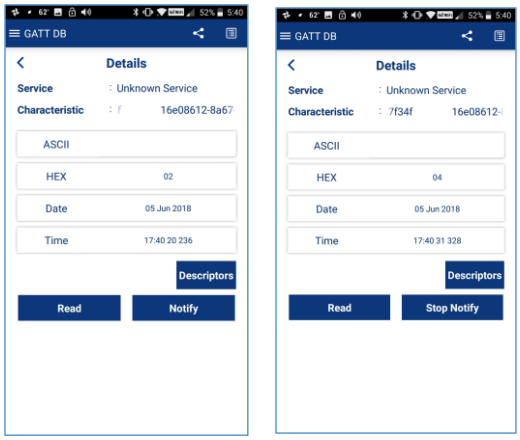


Tap on the GATT DB widget to open the browser. Then tap on the Unknown Service (which we know is WicedButton) and then on the Characteristic (which we know is MB1).





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.



When you are done, press back until CySmart disconnects. Then go to your phone's Bluetooth settings and remove the device from the list of paired devices. If you don't do this, you will have trouble connecting again once you re-program your kit since it will no longer match the information stored on the phone.

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 use privacy, 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 the privacy keys are available. If privacy is not enabled, 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 privacy keys from NVRAM. If keys have not been previously saved for the device, then this state must return a value other than WICED_BT_SUCESS such as WICED_BT_ERROR. The non-success return value causes the stack to generate new privacy 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 privacy keys from NVRAM failed (i.e. the return value from BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT was not 0). During this event, the privacy keys must be saved to NVRAM.
	BTM_LOCAL_IDENTITY_KEYS_UPDATE_EVT	This is called twice to update both the IRK and the ER in two steps.

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.
Read Values	GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_READ	The firmware must get the value from the correct location in the GATT database.

Activity	Callback Event Name	Reason
Write Values	GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_WRITE	The firmware must store the provided value in the correct location in the GATT database.
Notifications	N/A	Notifications must be sent whenever an attribute that has notifications set is updated by the firmware. Since the change comes from the local firmware, there is no stack or GATT event that initiates this process.
Disconnect	BTM_BLE_ADVERT_STATE_CHANGED_EVT	Upon a disconnect, the firmware will get a GATT event handler callback for the GATT_CONNECTION_STATUS_EVENT (more on this later). At that time, it is the user's responsibility to determine if advertising should be re-started. If it is restarted, then you will get a BLE stack callback once advertisements have restarted with a return value of 3 (fast advertising) or 4 (slow advertising).
Re-Connect	GATT_CONNECTION_STATUS_EVT	The callback needs to determine if the event is a connection or a disconnection. For a connection, the connection ID is saved, and pairing is enabled (if a secure link is required).
	BTM_BLE_ADVERT_STATE_CHANGED_EVT	Advertising off.
Re-Pair	BTM_ENCRYPTION_STATUS_EVT	In this state, the firmware reads the state of the server from NVRAM. For example, the BD_ADDR of the paired device and the saved state of any notify settings may be read. Since the paired device BD_ADDR and keys were already available, no other steps are needed to complete pairing.
Read Values	GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_READ	The firmware must get the value from the correct location in the GATT database.
Write Values	GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_WRITE	The firmware must store the provided value in the correct location in the GATT database.
Notifications	N/A	Notifications must be sent whenever an attribute that has notifications set is updated by the firmware. Since the change comes from the local firmware, there is no stack or GATT event that initiates this process.
Disconnect	BTM_BLE_ADVERT_STATE_CHANGED_EVT	Advertising on.
Reset	BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT	Local keys are loaded from NVRAM.
	BTM_ENABLED_EVT	Stack is enabled. Paired device keys (including the BD_ADDR) are loaded from

Activity	Callback Event Name	Reason
		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. If keys are not available, this state must return WICED_BT_ERROR. That return value causes the stack to generate keys and then it will call the corresponding update event so that the new keys can be saved in 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.

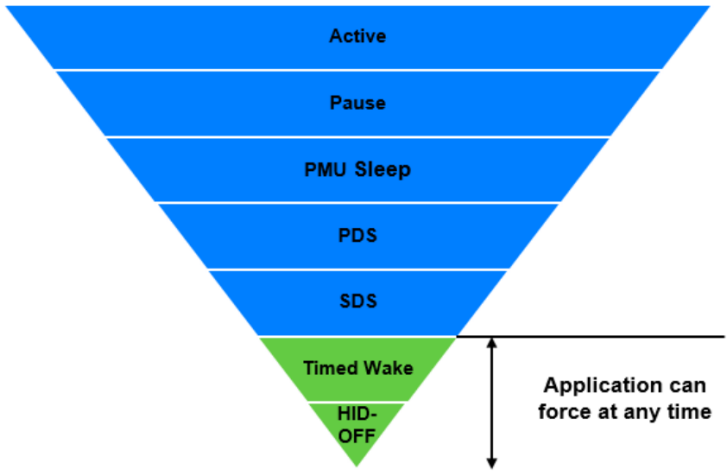
4B.10 Low Power

4B.10.1 Power Modes

WICED Bluetooth devices support different power modes. However, it is important to note that not all the devices support every mode. The following table shows the different WICED Power Modes:

Mode	Description
Active	Active mode is the normal operating mode in which all peripherals are available, and the CPU is active.
Pause	In this mode, the CPU is in Wait for Interrupt (WFI) and the HCLK, which is the high frequency clock derived from the main crystal oscillator, is running at a lower clock speed. Other clocks are active, and the state of the entire chip is retained. Pause mode is chosen when the other lower power modes are not possible.
PMU Sleep	In this mode, the CPU is in WFI and the HCLK is not running. The PMU determines if other clocks can be turned off and does so accordingly. The state of the entire chip is retained, the internal LDOs run at a lower voltage (voltage is managed by the PMU), and SRAM is retained.
Power Down Sleep (PDS)	This mode is an extension of the PMU Sleep wherein most of the peripherals such as UART and SPI are turned OFF. The entire memory is retained, and on wakeup the execution resumes from where it was paused.
Shut Down Sleep (SDS)	Everything is turned OFF except LHL GPIOs, RTC, and LPO. The device can come out of this mode either due to Bluetooth activity or an LHL interrupt. This mode makes use of micro-Bluetooth Core Scheduler (µBCS), which is a compressed scheduler different from the regular BCS. Before going into this mode, the application can store some bytes of data into the Always-On RAM (AON). When the device comes out of this mode, the data from AON is restored. After waking from SDS, the application will start from the beginning (warmboot) and must restore its state based on information stored in AON. In the SDS mode, a single Bluetooth task with no data activity, such as an ACL connection, BLE connection, or BLE advertisement can be performed. If there is data activity during these tasks, the system will undergo full boot and normal BCS will be called.
Timed-Wake	The device can enter this mode asynchronously, that is, the application can force the device into this mode at any time without asking the permission from other blocks. LHL, RTC, and LPO are the only active blocks. A timer that runs off the LPO is used to wake the device after a pre-determined fixed time.
HID-OFF	This mode is similar to Timed-Wake, but in HID-OFF mode even the LPO and RTC are turned OFF. So, the only wakeup source is a LHL interrupt.

The following diagram shows the hierarchy of these power modes. Note that the CYW20719 supports SDS but not for HID-OFF or Timed Wake.



The Power Management Unit (PMU) core manages power and clock resources for the entire chip, including Clock/Reset management and power management. The CYW20719 has an advanced PMU, which automatically controls the power switches of all the resources. The following table shows the operational power modes of the SoC peripherals:

S. No	Power Domain	Peripherals	Operational Power Modes
1	VDDC	PWM	If ACLK is used, then the hardware block can operate until the PMU enters Sleep. If LHL clock is used, then the hardware block can operate until the PMU enters SDS.
2	VDDCG	I2C, SPI, PUART, WDT, ARM GPIO, Dual Input 32-bit Timer	The hardware blocks can operate until the PMU enters Sleep.
3	VBAT/LHL	LHL GPIO, Analog PMU, RTC	The hardware blocks can operate until the PMU enters SDS.
4	VBAT/LHL	Aux ADC	The Aux ADC can operate until the PMU enters Sleep.

4B.10.2 WICED Low-Power code

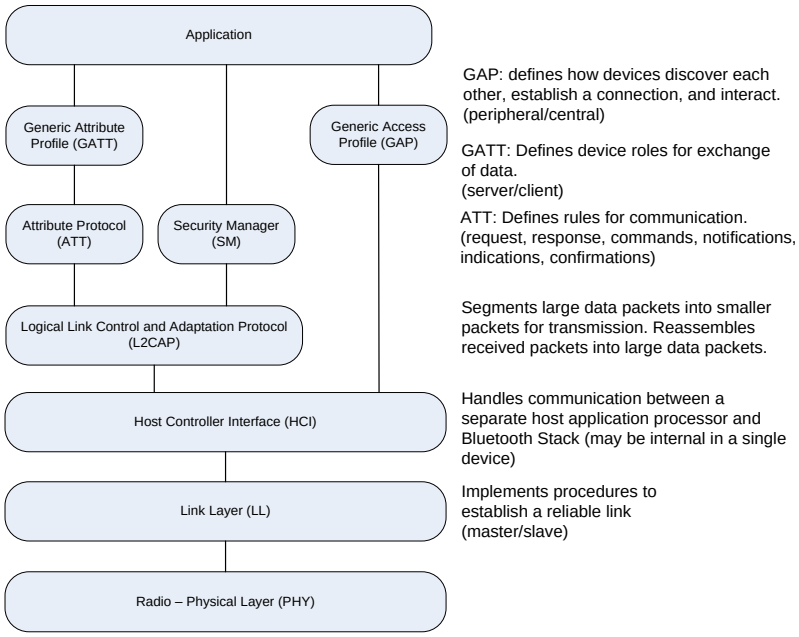
The header file *wiced_sleep.h* contains the API related to low power operation of CYW20719. That header file must be included in the source code to call the sleep API functions.

- wiced_sleep_configure(): Use this function to enable the low power operation of the device. The parameter to be passed to this function contains a callback that will be called by the PMU to poll for sleep permission. In the callback, the application must return one of these values based on the requirements:
- `WICED_SLEEP_NOT_ALLOWED` – The application can return this value if it does not want the device to enter Sleep mode.
- `WICED_SLEEP_ALLOWED_WITHOUT_SHUTDOWN` -The application can return this value if low power is allowed, but the device should not enter SDS. This means that the lowest power mode that the device can enter is PDS. This value should be passed if data exchange over Bluetooth is expected and entering SDS will be irrelevant.
- `WICED_SLEEP_ALLOWED_WITH_SHUTDOWN` – When this value is returned, the device can enter any of the low power modes including SDS.

See the low_power_sensor example included in WICED. The example demonstrates how low power can be configured in CYW20719.

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.



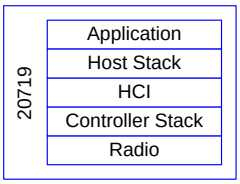
4B.11.1 HCI

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

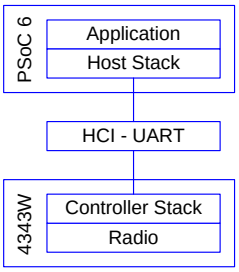
For technical and cost reasons, when Bluetooth was originally created the Radio was a separate chip from the one that was running the Application. The Radio chip took the name of Controller because it was the Radio and Radio Controller, and the chip running the Application was called the Host because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

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

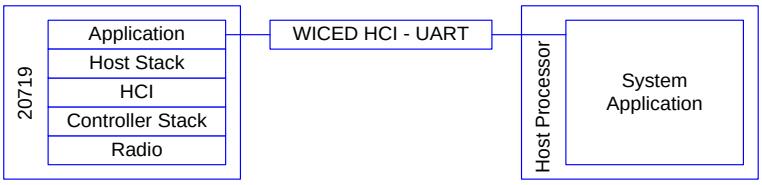
By standardizing the HCI interface, it allowed big application processors (like those exiting in PCs and cellphones) to interface with Bluetooth. As time went by the Host and Controller have frequently merged into one chip (e.g. CYW20719), however the HCI interface persists even though both sides may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



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



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



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 Exercises

Exercise - 4B.1 Simple BLE Project with Notifications using WICED BT Designer

Follow the instructions in section 4B.8 to use WICED BT Designer to create a project with a Service called WicedButton and a Characteristic called MB1 that will keep track of how many times the button has been pressed and will send a notification if it is enabled by the client.

Hint: Remember to use your initials in the project name (i.e. device name) so that you can find it in the list of devices that will be advertising.

Hint: Remember to add the option BT_DEVICE_ADDRESS=random to the make target so that your device's address will not conflict with another kit in the class.

Exercise - 4B.2 BLE Notifications for CapSense

Introduction

In this exercise, you will add notifications manually to the CapSense BLE project from the previous chapter. This will allow you to become more familiar with the GATT database structure in the firmware and will also allow you to re-use the custom code created for handling the CapSense button Service.

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/ex03_ble_con to ch04b/ex02_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. Update the maxlen and curlen in the GATT database lookup table (gatt_db_lookup_table).
   c. Hint: Many function names and variable names start with "ex03_ble_con". You can do a global search/replace to change these to "ex02_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
   a. Hint: The CCCD is an array of 2 uint8_t values.
   b. Hint: Initialize the CCCD value to 0.
6. Add the CCCD handle and array name to the GATT attribute lookup table.
7. 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.
8. 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.
9. 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*.
10. 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 to disable privacy.

Testing

1. Create a Make Target and run it to program the project to the board. Make sure you include the option for BT_DEVICE_ADDRESS=random.
2. Open the mobile CySmart app.
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.
7. Start scanning. When you see your device show up, stop scanning and then connect to your device.
8. Notice that the address that appears in the scan results is the "Public Address". This is because we have disabled privacy.
9. 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.
10. Press the CapSense buttons and observe that the values update real-time due to the notifications.
11. Click on "Disable All Notifications"
12. Press the CapSense buttons again and observe that the values are no longer updated.
13. Click "Disconnect".

Exercise - 4B.3 BLE Pairing and Security

Introduction

In this exercise, you will add Pairing and Security (Encryption) 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/ex02_ble_ntfy to ch04b/ex03_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 ex03_ble_pair.c file.
    c. Hint: Many function names and variable names start with "ex02_ble_ntfy". You can do a global search/replace to change these to "ex03_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. The authorization request and init_keys each have many options which can be explored in wiced_bt_dev.h. The values we selected determine that the link must use an LE secure connection with MITM and bonding, and that the encryption information and identity keys of the peer device are distributed.
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. Be sure to include the option for BT_DEVICE_ADDRESS=random.
2. Open the mobile CySmart app.
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.
6. Go to the phone's Bluetooth settings and remove the <inits>_pair device from the paired devices list. This is necessary so that when you re-program the kit the phone won't have stale bonding information stored which could prevent you from connecting.
7. Start the PC CySmart app.
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 since we haven't yet enabled privacy.
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.4 Save BLE Pairing Information (i.e. Bonding) and Enable Privacy

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 MITM 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 Templates/ch04b/ex04_ble_bond from the electronic class material folder.
    a. Create a new make target. Don't forget the BT_DEVICE_ADDRESS option.
    b. Update the device name in *wiced_bt_cfg.c* and *ex04_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. Click ‘Configure Master Settings’ and, under ‘Privacy 1.2’, change the Address Generation Interval to match the *rpa_refresh_timeout* in *wiced_bt_cfg*.
5. 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.

6. Start scanning. Once you see your device in the list stop scanning. Note that your device shows up with a Random Bluetooth address now since privacy is enabled.
7. Connect to your device.
8. Click on "Discover all Attributes".
9. Click on "Pair" and click "Yes" when asked if you want to add the device to the resolving list so that the privacy keys will be remembered by CySmart.
    a. Note down the Bluetooth Stack events that occur during pairing. This information is displayed in the UART.
10. Click on "Enable All Notifications". Touch the CapSense buttons and observe the characteristic value changes.
11. Click "Disconnect". Do NOT remove the device from the Device List this time – we want bonding information retained.
12. Start a new scan and stop when your device appears in the list.
13. Notice how the Address is now listed as a Public Identity Address rather than Random in the table of discovered devices. Look at the Resolving List; both the Random Device Address and the Public Identity Address are listed. If you click on 'View …', some Details concerning the device appear. Multiple things, including the Identity Resolving Key, are listed. The IRK is used to map the Private Random Address to the Public Identity Address.
14. Re-connect to your device.
15. Click on "Discover all Attributes" and "Pair".
    a. Once again note down the Bluetooth Stack events that occur during pairing. You will notice that fewer steps are required this time.
16. Touch the buttons and observe that notifications are already enabled since they were enabled when you disconnected. This information was retained in NVRAM.
17. Disconnect again.
18. 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.
19. Start scanning, stop when your device appears, and then connect to your device for a third time.
20. Click on "Discover all Attributes" and "Pair".
    a. Note down the Bluetooth Stack events that occur this time during pairing. Compare to the previous two connections.
21. Note that notifications are still enabled.
22. Disconnect again.
23. Clear the Device List.
24. Start scanning and stop when your device appears. Notice that it again has a Random address type.
25. Connect to your device, "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.
26. Click on Disconnect and close the Authentication failed message window.

27. Press MB1 on the kit and note that LED1 begins flashing. This indicates that the bonding information has been cleared from the device and it will now allow a new connection.
28. Connect, Discover Attributes, and Pair again. This time it should work.
29. Note the steps that the firmware goes through this time.
30. 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 wiced_bt_cfg.c, rpa_refresh_timeout is set to WICED_BT_CFG_DEFAULT_RANDOM_ADDRESS_CHANGE_TIMEOUT, enabled Link Layer Privacy.
7. In the Bluetooth stack event BTM_PAIRED_DEVICE_LINK_KEYS_UPDATE_EVT, save the keys for the peer device to NVRAM.
8. In the Bluetooth stack event BTM_PAIRED_DEVICE_LINK_KEYS_REQUEST_EVT, read the keys for the peer device from NVRAM.
9. In the Bluetooth stack event BTM_LOCAL_IDENTITY_KEYS_UPDATE_EVT, save the keys for the local device to NVRAM.
10. In the Bluetooth stack event BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT, read the keys for the local device from NVRAM.
11. 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.
12. 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.
13. 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.
14. A Thread is used to control LED blinking based on the state of bond_mode.
15. Finally, privacy is enabled in wiced_bt_cfg.c by updating the rpa_refresh_timeout to *WICED_BT_CFG_DEFAULT_RANDOM_ADDRESS_CHANGE_TIMEOUT*.

Questions

1. What items are stored in NVRAM?

2. Which event stores each piece of information?

3. Which event retrieves each piece of information?

4. In what event is the privacy info read from NVRAM?

5. Which event is called if privacy information is not retrieved after new keys have been generated by the stack?

Exercise - 4B.5 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 ex04_ble_bond to ex05_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 ex05_ble_pass.c file.
    b. Hint: Don't forget to look for header file names in the two C files that contain ex04_ble_bond and don't forget the source file names in the makefile.
    c. Hint: Many function names and variable names start with "ex04_ble_bond". You can do a global search/replace to change these to "ex05_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. As usual, don't forget the BT_DEVICE_ADDRESS=random option.
2. Open a UART terminal window.
3. Open the mobile CySmart app.
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. Disconnect, then manually remove the bonding information from the phone's Bluetooth settings.

8. Press MB1 to put the kit into Bonding mode and then reconnect. Observe that the key must be entered again to connect.
9. Disconnect again and remove the bonding information from the phone's Bluetooth settings.
10. 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 from the kit. This is necessary since we only allow bonding information from one device to be stored in our firmware. The next exercise will fix that.

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?

Exercise - 4B.6 (Advanced) Add Multiple Bonding Capability

Introduction

In this exercise, you will copy the multiple bonding project from the templates folder and use it to bond to up to 4 different devices at one time

Project Creation

1. Copy Templates/ch04b/ex06_ble_multi from the folder provided in the electronic class material, create the Make Target, and make the necessary updates.
    a. Hint: Make sure you change "key" to your initials in both the ex06_ble_multi.c file and wiced_bt_cfg.c file to so that you will be able to find your device.

Testing

1. Download the project onto the kit.
2. Open a UART terminal window.
3. The device starts out in bonding mode (the red LED should be flashing).
4. Open the mobile CySmart app.
5. Discover all attributes in the GATT database, and attempt to Pair with the device.
6. Once Pairing completes, verify that the application still works. The device will now be in “normal mode”.
7. Disconnect from the device. Keep the bonding information on the phone.
8. To put the device back in bonding mode, press MB1. The red LED will begin flashing again.
    a. If you already have the max number of devices bonded and press MB1, it will remove the oldest bonded device before going back into bonding mode.
9. Connect to the device using the PC version of CySmart and Pair with the device.
10. Verify that the application still works.
11. Press MB2. You should see a list of the bonded devices on the terminal window.
12. Disconnect from the PC CySmart app, connect again from the phone, and verify that the application still works. It should connect and pair without requiring the passkey.
13. Disconnect from the device, re-connect from the PC, Pair, and verify that the application still works. Again, a passkey should not be required to Pair with the device.
14. Disconnect from the device.
15. Clear the bonding information from the phone and CySmart on the PC.
16. Note: you may not be able to bond to multiple computers running CySmart, but you can connect to a PC and a phone or multiple phones.

Overview of Changes

1. A #define for BOND_MAX which is the maximum number of devices that can be bonded at a time (default is set to 4).
2. NVSRAM is organized so that we have:
    a. 1 VSID for the local keys (i.e. privacy keys).
    b. 1 VSID to keep track of how many bonded devices we have and the next one to be over-written.
    c. VSIDs to hold host information (i.e. BD_ADDR and CCCD values). There is one VSID for each bonded device so this is BOND_MAX VSIDs.
    d. VSIDs to hold encryption keys for each bonded host. There is one VSID for each bonded device so this is BOND_MAX VSIDs.
3. Add an interrupt callback function that prints bonding information when MB2 is pressed.
4. Update interrupt callback function for MB1 so that it just toggles whether we are in bonding mode or not. Upon entering bonding mode, if the max number of devices is already bonded, it will remove the oldest information from the bonded device list, address resolution database, and NVRAM (hostinfo and paired device keys).
5. Update BTM_PAIRING_COMPLETE_EVT so that it stores the newly bonded device's host information into the correct VSID slot in NVRAM. That is, it needs to store the information in the first free location. This case also increments the number of bonded devices and increments the next free slot location since a new device has just completed bonding.
6. Update BTM_ENCRYPTION_STATUS_EVT so that it searches for the BD_ADDR of the device that was just paired. If it is found, then the device was previously bonded, so its host information can be read from NVRAM.
7. Update BTM_PAIRED_DEVICE_LINK_KEYS_UPDATE_EVT so that it stores the newly bonded device's encryption key information into the correct VSID slot in NVSRAM. That is, it needs to store the information in the first free location.
8. Update BTM_PAIRED_DEVICE_LINK_KEYS_REQUEST_EVT so that it searches through all bonded devices to determine if the device that is currently trying to pair already has bonding information. If the information is found, it is loaded from NVRAM. If the information is not found, this case returns WICED_BT_ERROR which causes the stack to generate new keys and then call BTM_PAIRED_DEVICE_LINK_KEYS_UPDATE_EVT.
9. Update the GATT set_value function so that it stores any changes to the CCCD value to the proper NVRAM location. That is, the value must be stored in the location that is assigned to the currently connected host.

Exercise - 4B.7 (Advanced) BLE Low Power

TODO: project needs to be updated to build off of ex04_ble_bond

Introduction

In this exercise, you will copy the low power project from the templates folder and measure the power consumptions in different power modes.

Project Creation

1. Copy Templates/ch04b/ ex07_low_power from the folder provided in the electronic class material, create the Make Target, and make the necessary updates.
    a. Hint: Make sure you change "key" to your initials in the wiced_app_cfg.c so that you will be able to find your device.

Testing

1. Connect an ammeter across jumper J15 to allow current measurement.
2. Download the project onto the kit.
3. Open a UART terminal window.
4. Open the PC CySmart app.
5. Click 'Configure Master Settings', then 'Connection Parameters', enter the following values, and hit OK.
    a. Connection Interval Minimum: 100ms
    b. Connection Interval Maximum: 100ms
    c. Supervision Timeout: 5120ms
6. Attempt to Connect to the device. You will see a notification asking to confirm the connection parameters. Select 'Yes'.
7. Discover all attributes in the GATT database, and attempt to Pair with the device. You will see a notification for numeric comparison (to prevent MITM attacks). Check the UART terminal window to confirm the codes are the same.
8. Once Pairing completes, verify that the application still works. The device will now be in SDS mode.
9. Enable all notifications and take note of the changing values of the motion sensor; it appears as a Primary Service Declaration (typically the third in the GATT database) with a long series of hexadecimal values.
10. Once the device is done sending the series of notifications it will go back to sleep. To erase bond information and start undirected advertisement, press MB1.
11. Compare the power consumption values between when the device is sending notifications and when it is sleeping.

Commented [GL1]: Need a section that explains the files and architecture of this FW. Need a section with questions.

Commented [GL2]: There are a bunch of compile warnings.

Commented [GL3]: What does "still" mean here? I didn't verify anything before now.

Commented [GL4]: How do I know that?

Commented [GL5]: How do I know what mode it is in at a given time? The UART doesn't really give much in the way of useful info.

Commented [GL6]: Why do we do this? What do all of the S's printed mean?

Project Description

Questions