Chapter 4B: More Advanced BLE Peripherals

This chapter expands your basic knowledge of BLE Peripherals by introducing more Attribute Procedures, GATT Database Features, Security, WICED Configuration Files, HCI, etc.

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 Peripheral device, you might want to send the Client an update each time a button value changes. 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_gatt_send_notification (conn_id, handle, length, value)
- wiced_bt_gatt_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 (note that Bluetooth Configurator generates this code for you in cycfg_gatt_db.c):

- 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 ModusToolbox IDE 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 ModusToolbox IDE and select "Open Declaration".

Third, in your GATT Attribute Write Callback you need to save the CCCD value that was written to you (note that this is done automatically in app_gatt_set_value() because the CCCD is writable).

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 commonly used Characteristic Descriptor 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 and CySmart) will display this information when you are looking at the GATT Database. To add the Characteristic User Description to your Characteristic just add this macro (again, note that Bluetooth Configurator generates this code for you in cycfg_gatt_db.c):

- 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 (which is in wiced_btsdk/dev-kit/baselib/20819A1/include).

```
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 Bluetooth Configurator

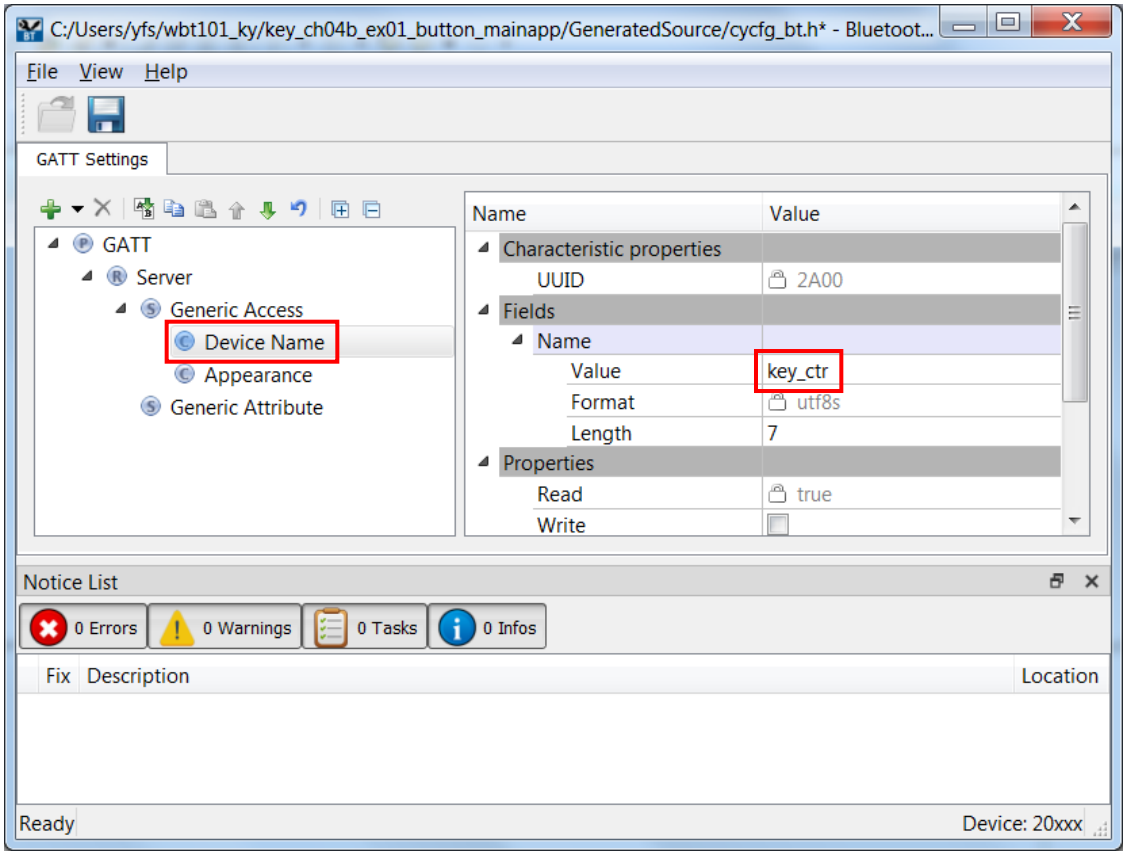
The Bluetooth Configurator can be used to setup Characteristics for Notify and Indicate. It can also be used to create Characteristic User Descriptions.

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

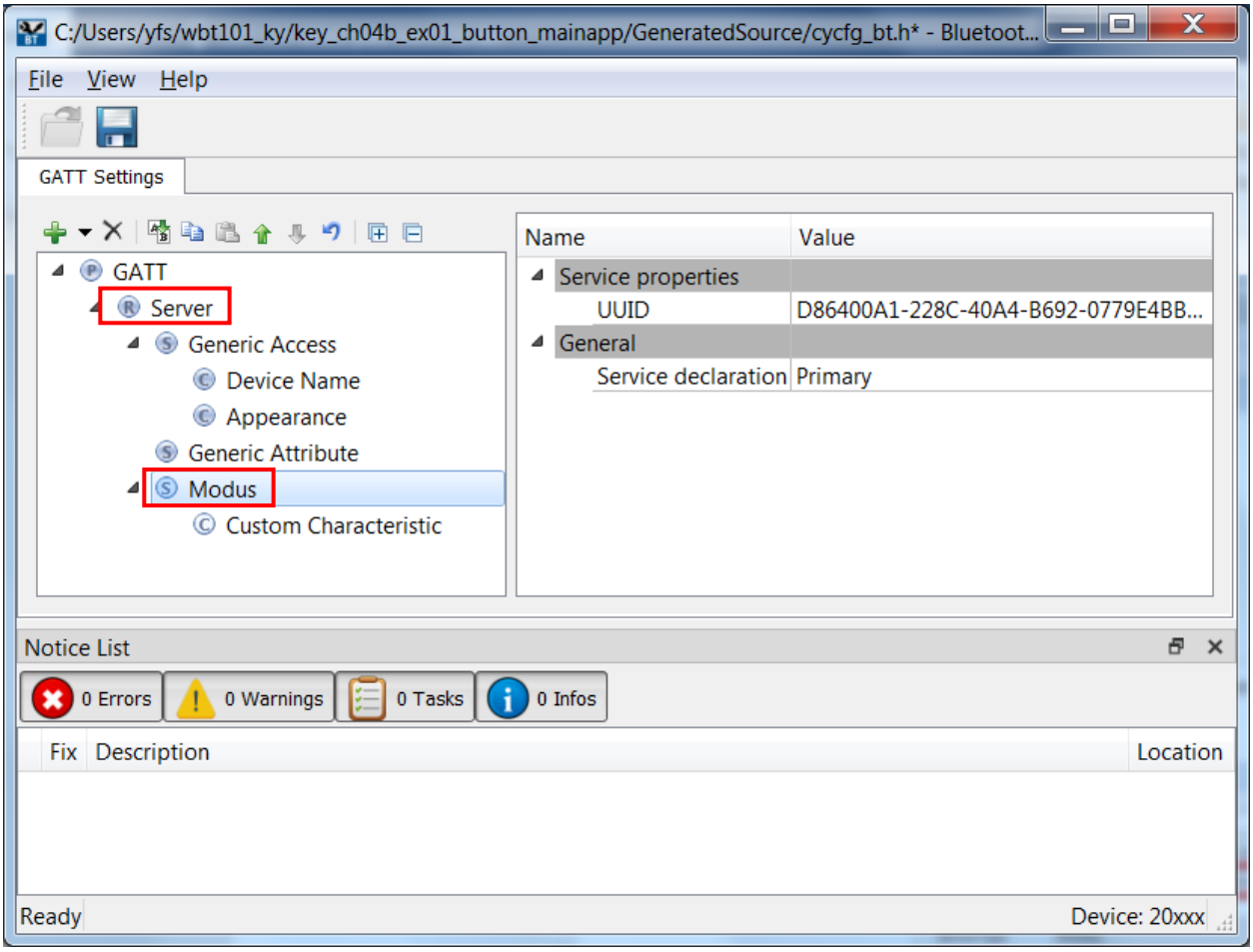
You will try this yourself in Exercise 4B.1

4B.3.1 Running the Tool

Start the Bluetooth Configurator by clicking on the link in the Quick Panel. In the Generic Access service set the Device Name. I'll use a device name of “key_ctr”. **When you do this yourself, use a unique name such as *<inits>_ctr* where *<inits>* is your initials**. Otherwise you will have trouble finding your specific device among all the ones that are advertising.



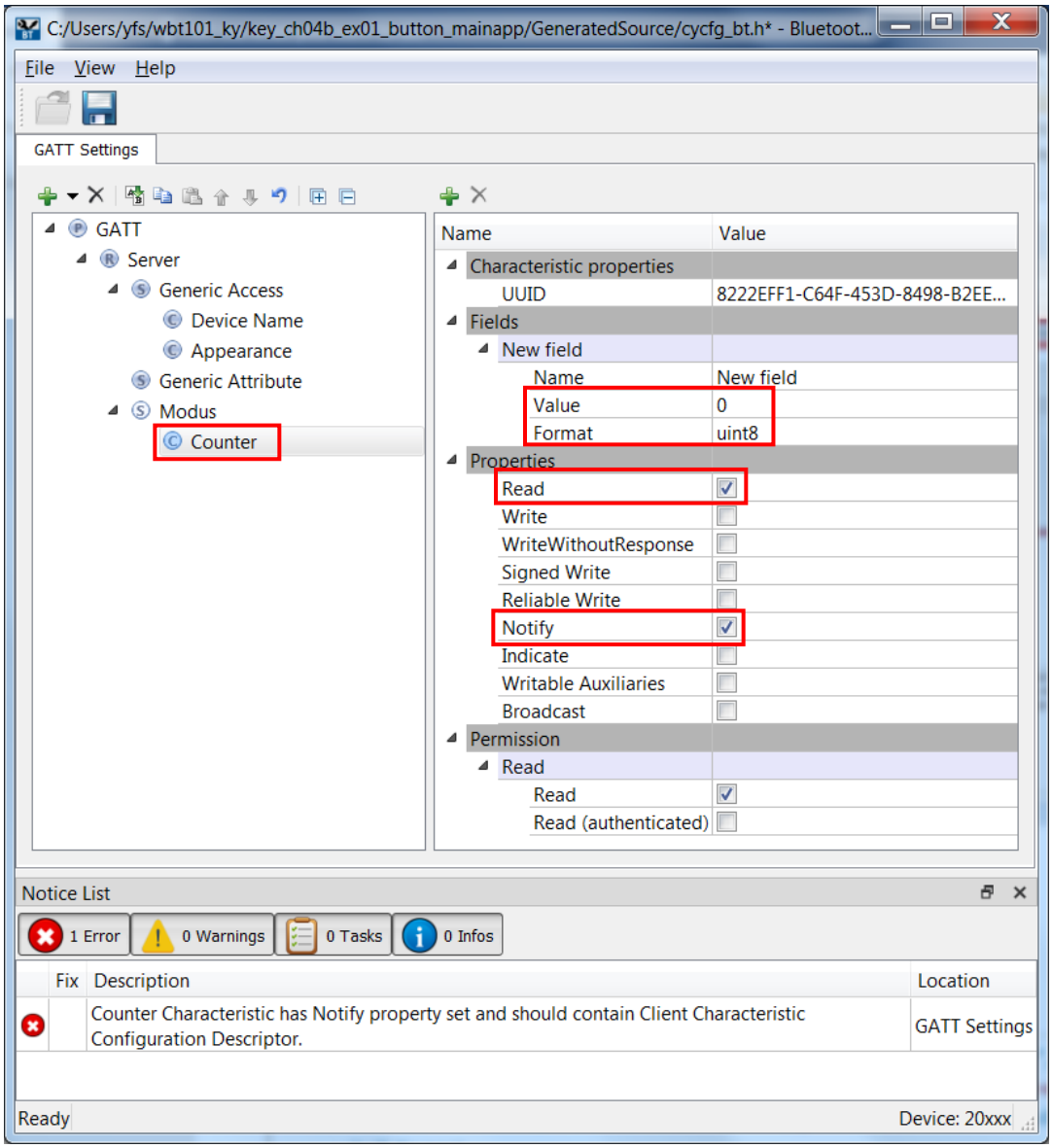
Right click on Server to add a new Custom Service and change its name to Modus (right-click on the new service name and select Rename).



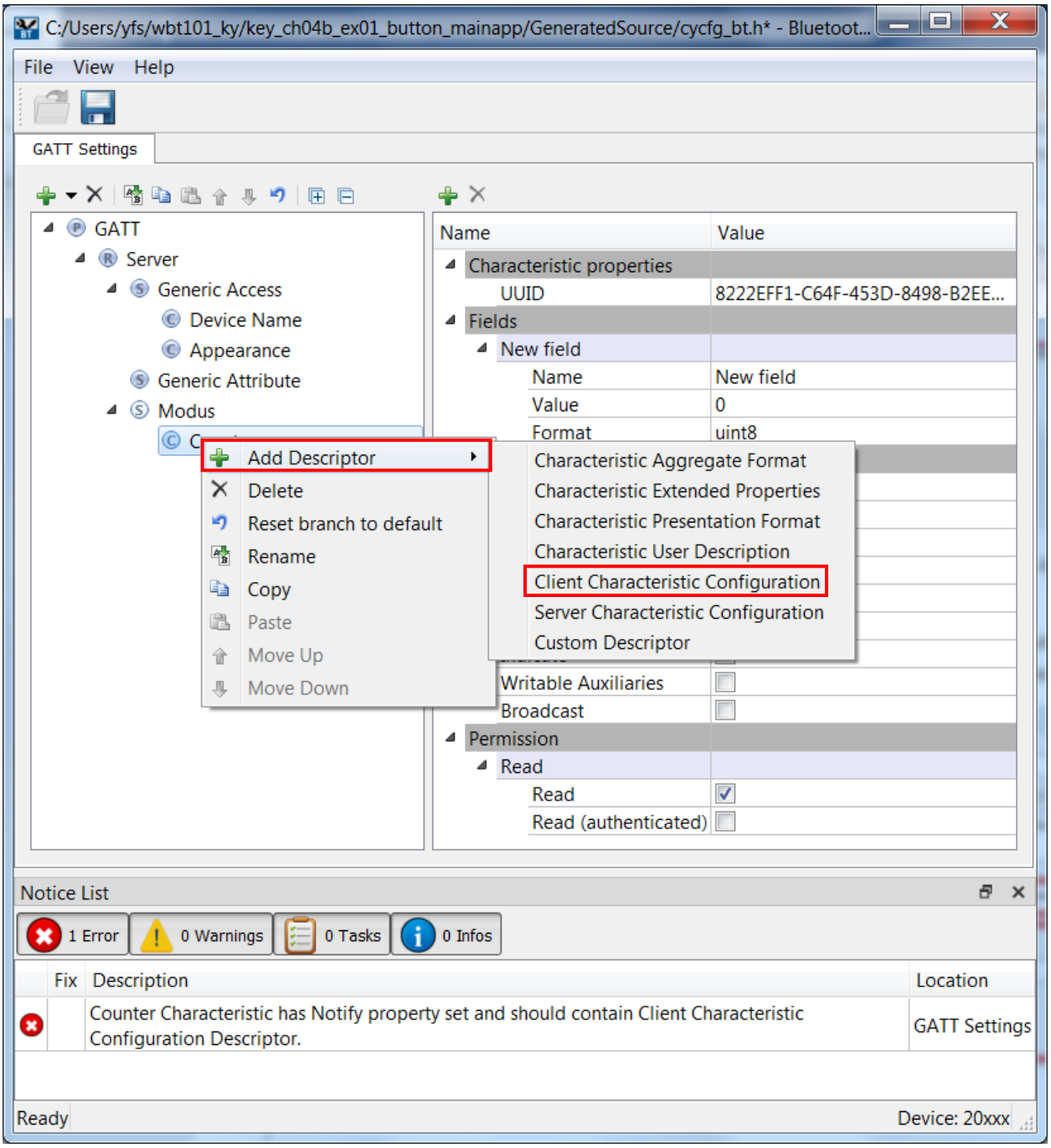
Next, rename the Custom Characteristic to “Counter”, give it the type uint8 and initial value of 0.

Under Properties enable Read. Now check the Permission section. 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. In real-world applications you would also turn on Read (authenticated) so that Read will require an Authenticated (i.e. Paired) link but we shall handle pairing later. Note, you must also leave on Read if you want to require authenticated reads although that does NOT mean that it will be Readable with a non-Authenticated link anymore.

Back under Properties, enable Notify so that the peripheral will tell us when Counter changes. Note that enabling notifications generates an error because you have not yet made that possible. The message tells you to add a CCCD (Client Characteristic Configuration Descriptor), which will shall do next.

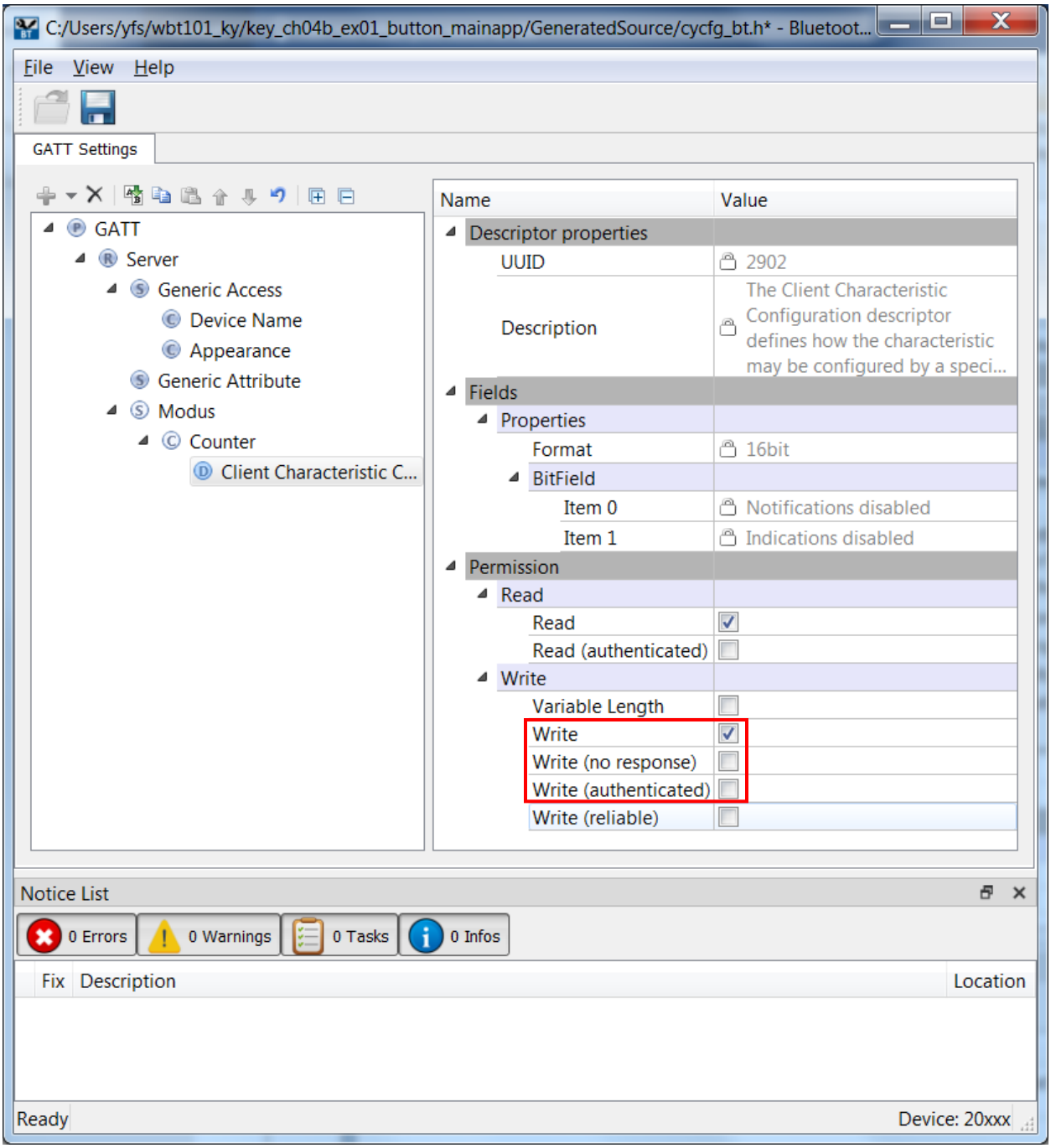


Add a CCCD by right-clicking on Counter, then Add Descriptor, and choose Client Characteristic Configuration.

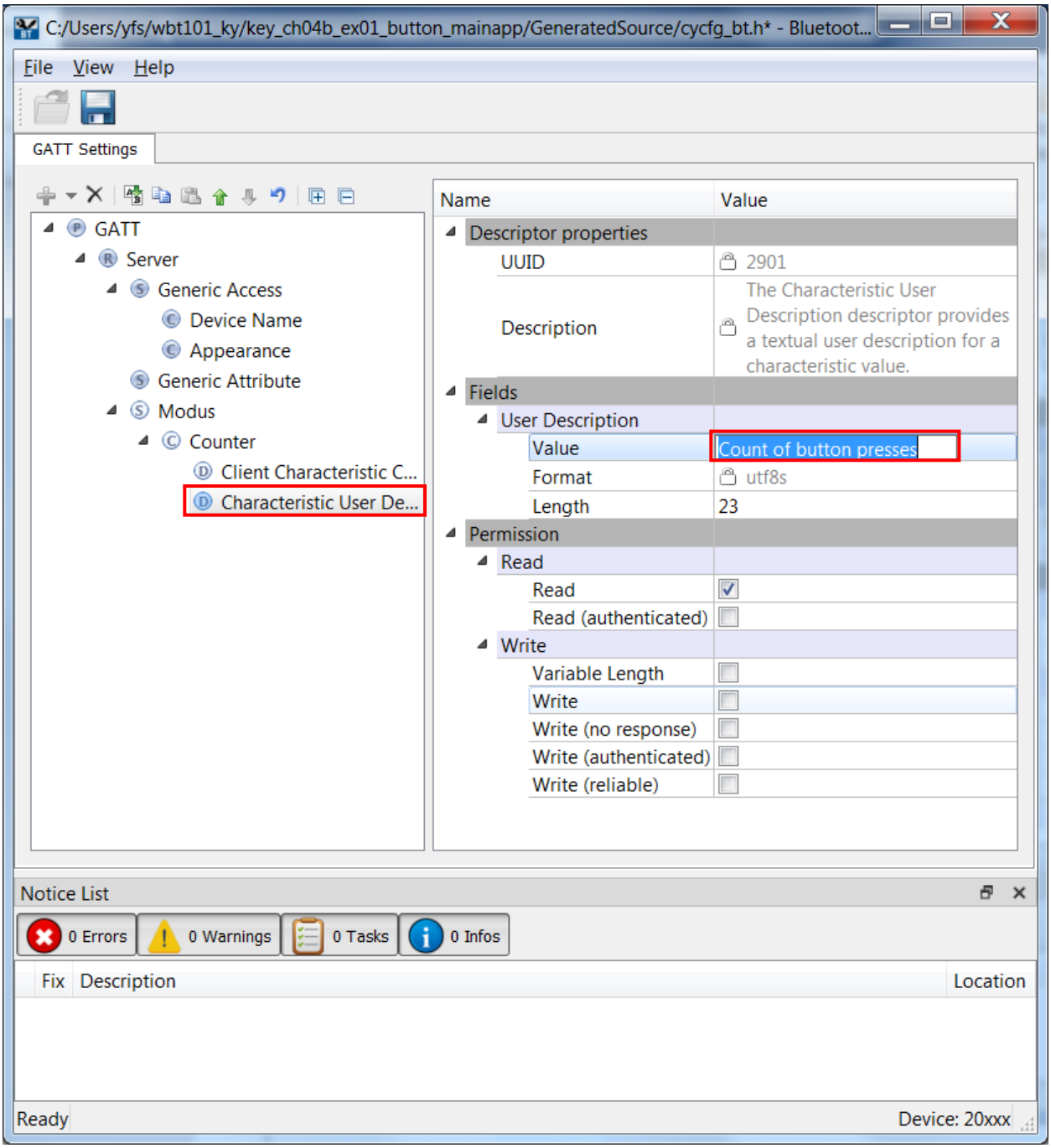


The new characteristic sets Notifications and Indications to disabled by default (and you cannot change that from the tool – if you want to enable them automatically it is better practice to enable them during pairing). Make sure Write permission is set on the Characteristic so that you will be able to set it from CySmart. Note that the error message has gone away.

Make sure the Write (authenticated) permission checkbox is not set because we are not (yet) requiring the devices to pair before enabling notifications.



Repeat the above step to add another Descriptor, this time choose the Characteristic User Description. The Value is a string that describes the Characteristic to the connected central device (Phone) so add a useful message. This Descriptor is read only.



Finally, save your edits and close the Bluetooth Configurator.

4B.3.2 Editing the Firmware

In app.c, we need to do the following (note that the first 4 steps are done for you in the starter template code because you learned how to do them in previous chapters):

1. Add the include for the GATT database header file

```
#include "cycfg_gatt_db.h"
```

2. Switch the debug messages to the PUART in application_start()

```
wiced_set_debug_uart( WICED_ROUTE_DEBUG_TO_PUART );
```

3. Declare a global variable called connection_id. Upon a GATT connection (i.e. in app_gatt_callback), save the connection ID. Upon 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:

```
/* Handle the connection */
connection_id = p_conn_status->conn_id;
```

   GATT Disconnection:

```
/* Handle the disconnection */
connection_id = 0;
```

4. Configure BUTTON_1 as a falling edge interrupt under the BTM_ENABLED_EVT.

```
/* Configure the button to trigger an interrupt when pressed */
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 );
wiced_hal_gpio_register_pin_for_interrupt(
       WICED_GPIO_PIN_BUTTON_1,
       button_cback,
       0 );
```

5. Create the button callback function (the function exists in the template, but it is empty). In the callback we will increment the Button Characteristic value, and then send a notification if we have a connection and the notification is enabled.

   Note that the array app_modus_counter was created by the Bluetooth Configurator. It holds the value for our counter characteristic. The name that the configurator uses is of the form: app_<service_name>_<characteristic_name>. The button callback function will look like this:

```
void button_cback( void *data, uint8_t port_pin )
{
    app_modus_counter[0]++;

    if( connection_id )
    {
       if( app_modus_counter_client_char_config[0] &
GATT_CLIENT_CONFIG_NOTIFICATION )
       {
           WICED_BT_TRACE( "Notifying counter change (%d)\r\n",
app_modus_counter[0] );
           wiced_bt_gatt_send_notification(
              connection_id,
              HDLC_MODUS_COUNTER_VALUE,
              app_modus_counter_len,
              app_modus_counter );
       }
    }

    /* Clear the GPIO interrupt */
    wiced_hal_gpio_clear_pin_interrupt_status( WICED_GPIO_PIN_BUTTON_1 );
}
```

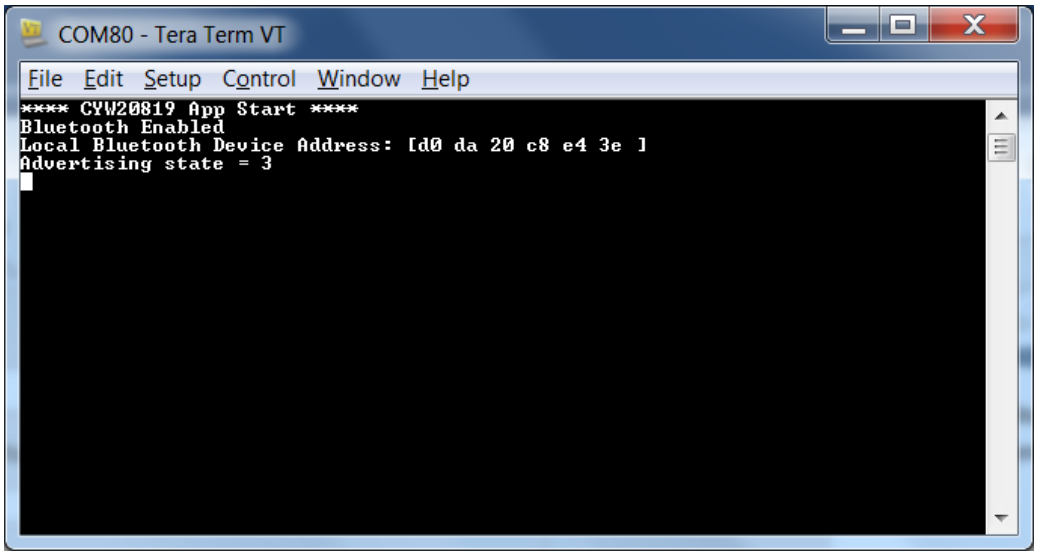
6. Add a debug message to in app_gatt_set_value() so you know when notifications get enabled/disabled. Note that the switch statement is already there but you need to add a new case.

```
switch( attr_handle )
{
    case HDLD_MODUS_COUNTER_CLIENT_CHAR_CONFIG:
       WICED_BT_TRACE( "Setting notify (0x%02x, 0x%02x)\r\n",
                          p_val[0], p_val[1] );
       break;
```

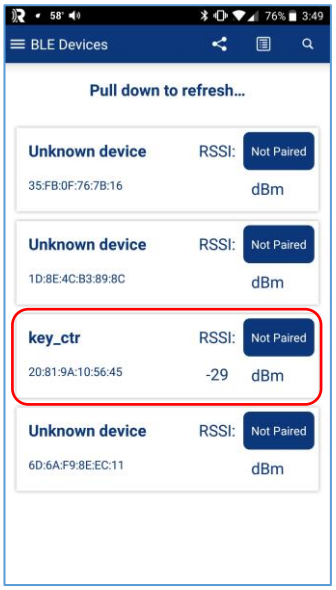
7. Open the makefile and set BT_DEVICE_ADDRESS?=random.
8. In app_bt_cfg.c, change the rpa_refresh_timeout to *WICED_BT_CFG_DEFAULT_RANDOM_ADDRESS_NEVER_CHANGE* so that privacy is disabled.

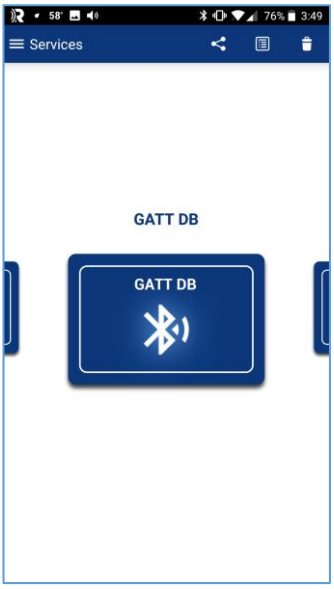
4B.3.3 Testing the Application

Start up a UART terminal to the WICED Peripheral UART port with a baud of 115200 and then program the kit. When the firmware starts up you will see some messages.

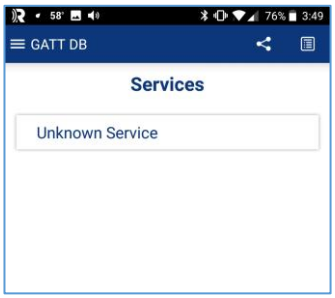


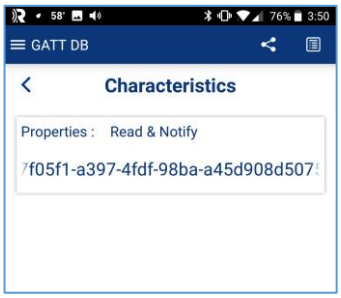
Run CySmart on your phone. When you see the "<inits>_ctr" device, tap on it. CySmart will connect to the device and will show the GATT browser widget.



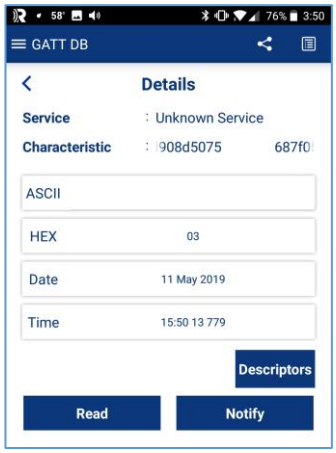


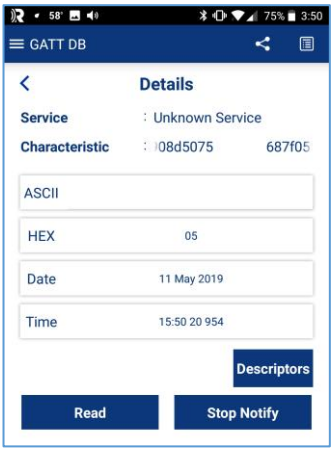
Tap on the GATT DB widget to open the browser. Then tap on the Unknown Service (which we know is Modus) and then on the Characteristic (which we know is Counter).



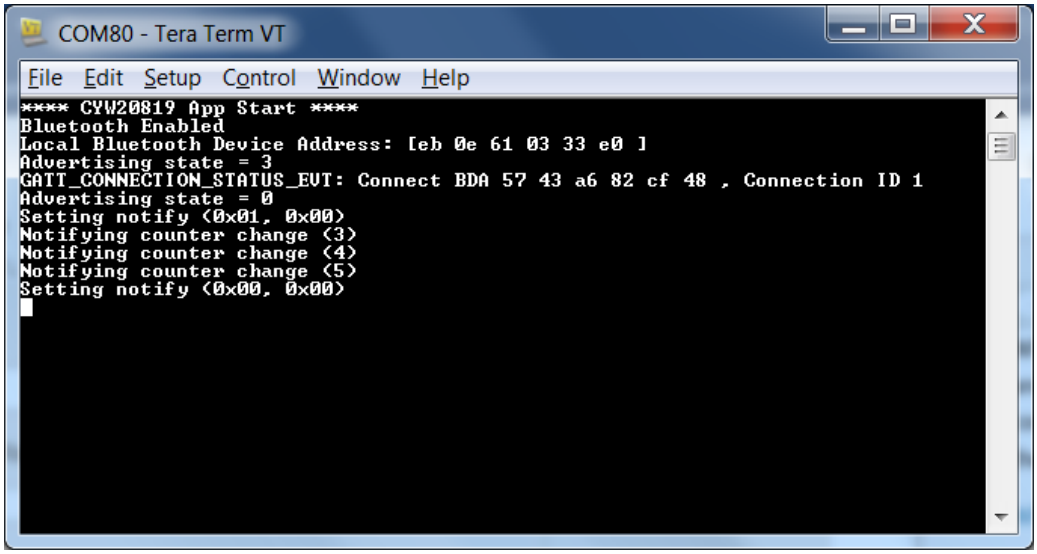


Tap the Read button to read the value. Press the button on the kit a few times and then Read again to see the incremented value. Then tap the Notify button to enable notifications. Now each time you press the button the value is shown automatically.





While playing with CySmart you will see messages like this in the terminal emulator:



4B.4 WICED Configuration: app_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 provided by the BSP or the starter template application and typically resides in the file app_bt_cfg.c

The structure definition is shown below. Note that many of the entries are themselves structures with multiple entries of their own. Note that in BLE peripherals the device name is held in the GATT database instead of the device_name member of this struct.

```
/** 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 also defined in app_bt_cfg.c and there are four different size buffer pools. 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,      6   },      /* Medium Buffer Pool (used for HCI & RFCOMM ctl messages, min rec. size is 360) */
    { 1056,     6   },      /* Large Buffer Pool  (used for HCI ACL messages) */
    { 1056,     0   },      /* Extra Large Buffer Pool - Used for avdt media packets and miscellaneous */
};
```

There is a user guide on the SDK documentation page that contains some additional information on the use of buffer pools. For example, the large buffer pool should be set to at least as large as the MTU value plus 12.

You can read the amount of free memory in the device at initialization and after starting the stack by using the function wiced_memory_get_free_bytes.

4B.6 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).

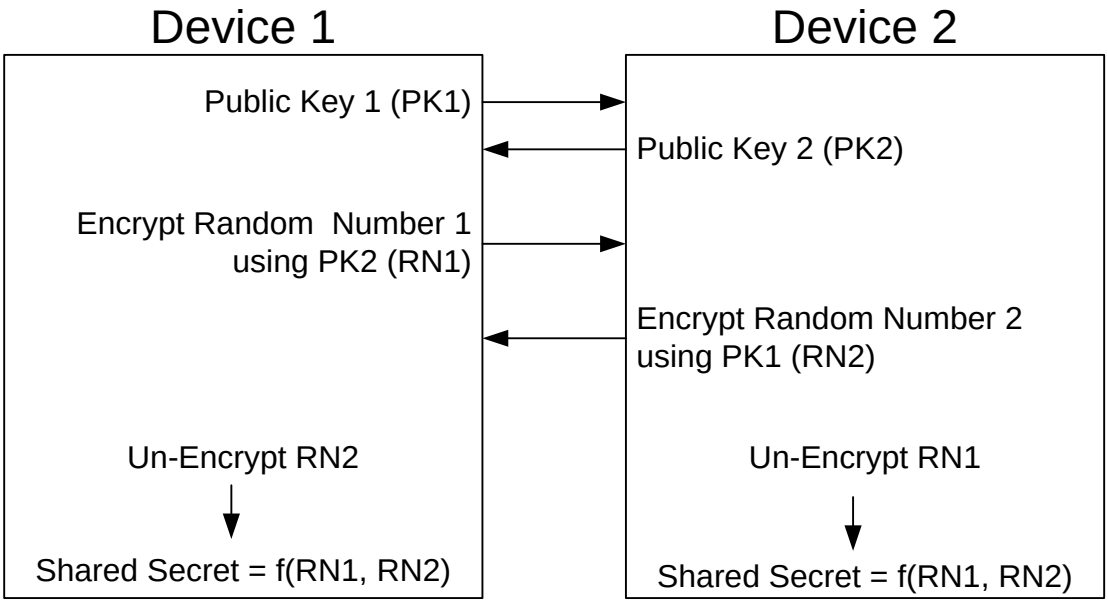
- CypherText = F(SharedSecret,PlainText)

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.6.1 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 (MIM)? There are four possible methods.

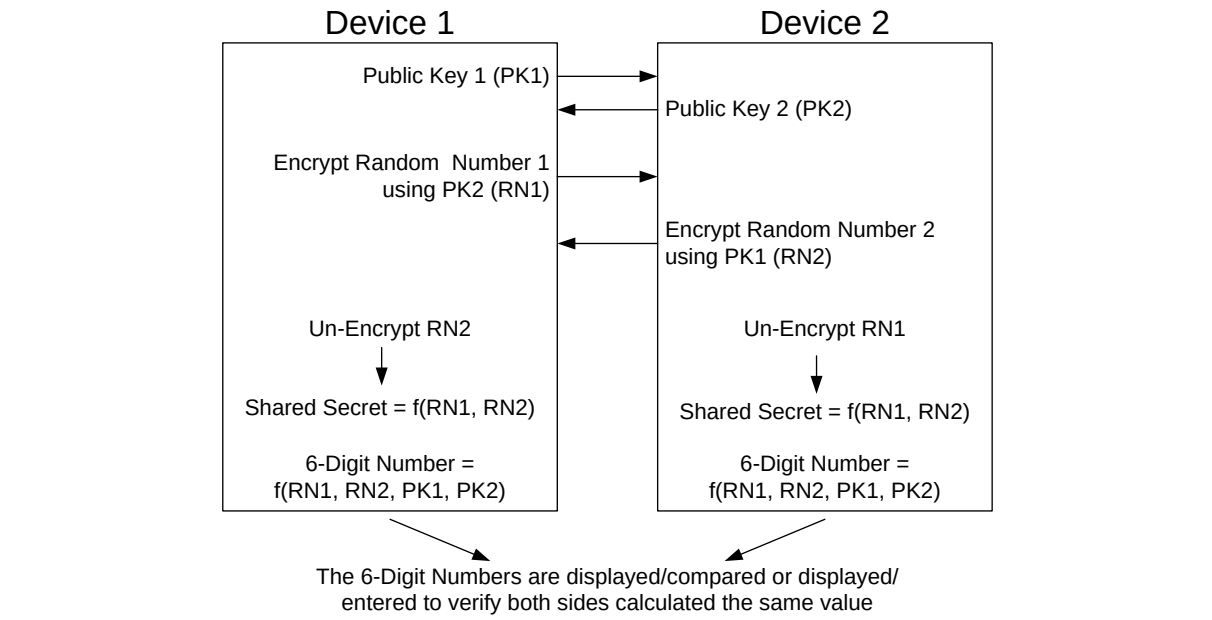
Method 1 is called "Just works". In this mode you have no protection against MIM.

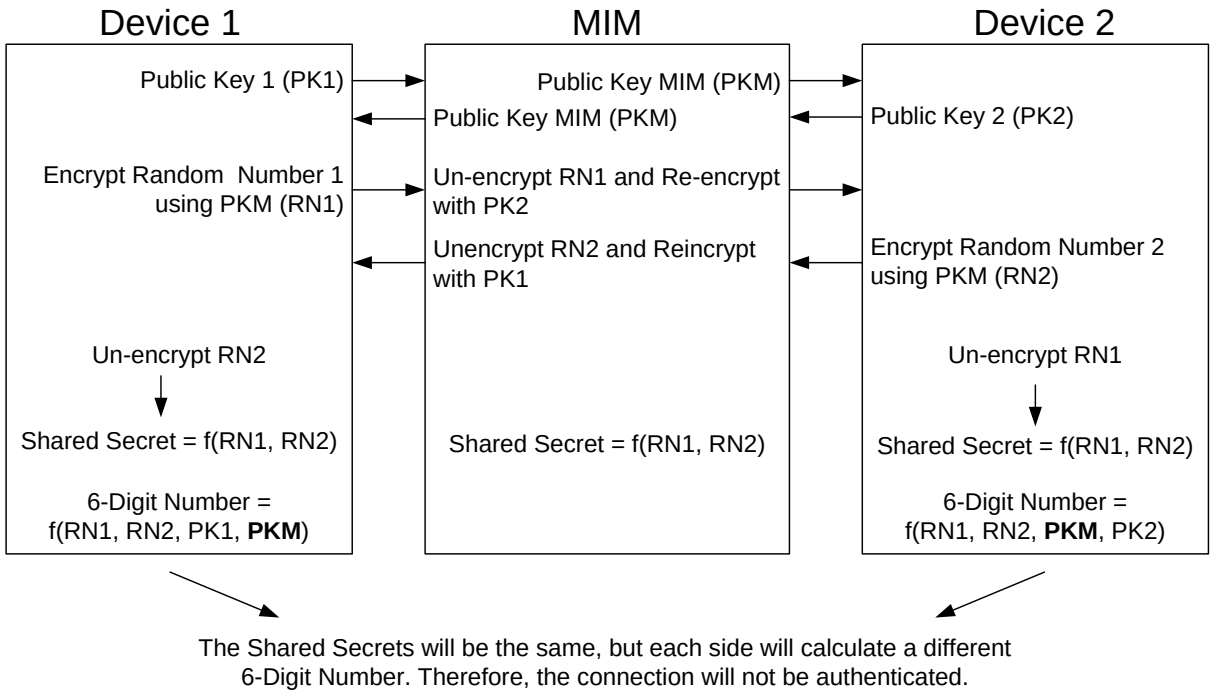
Method 2 is called "Out of Band". Both sides of the connection need to be able to share the PIN via some other connection that is not Bluetooth such as NFC.

Method 3 is called "Numeric Comparison" (V2.PH.7.2.1). In this method, both sides display a 6-digit number that is calculated with a nasty cryptographic function based on the random numbers used to generate the shared key and the public keys of each side. The user observes both devices. If the number is the same on both, then the user confirms on one or both sides. If there is a MITM, then the random numbers on both sides will be different so the 6-digit codes would not match.

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 Passkey. The other side must be able to display the Passkey. One device displays the Passkey and the user is required to enter the Passkey on the other device. Then an exchange and comparison process happens with the Passkeys being divided up, encrypted, exchanged and compared.

Pictorially, the process with no MIM and with MIM is shown below. Note that if there is a man in the middle, the two sides will calculate different numbers because the number is a function of the public keys used to encrypt the random numbers. If both sides used the same two public keys, then there can't be a man in the middle.

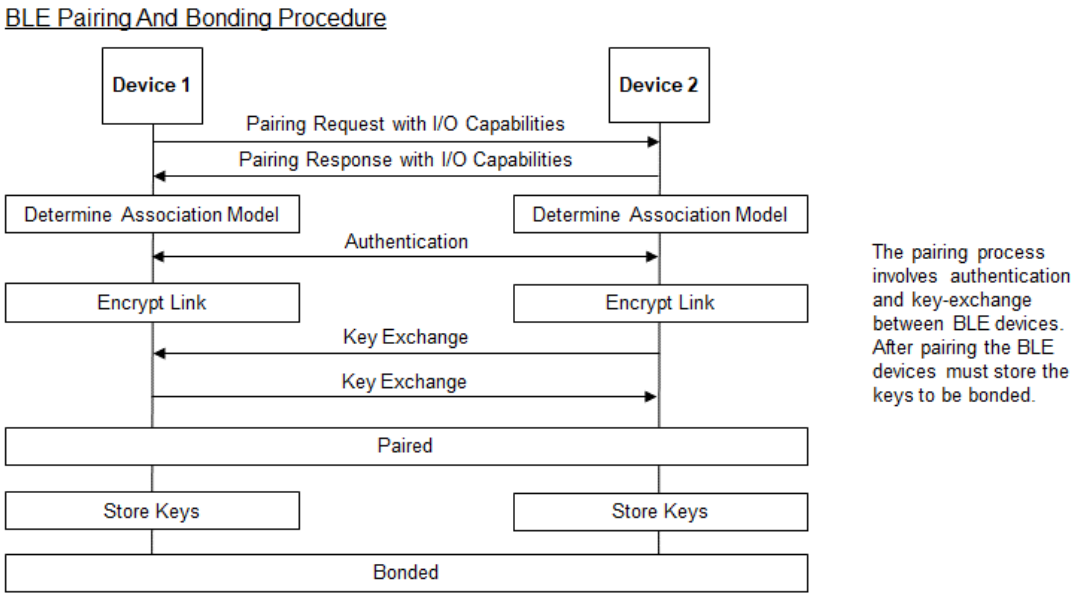




4B.6.2 Bonding

The whole process of Pairing is a bit painful and time consuming. It is also the most vulnerable part of establishing security, so it is beneficial to do it only once. 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. This allows the next connection to launch without repeating the pairing process.

4B.6.3 Pairing & Bonding Process Summary



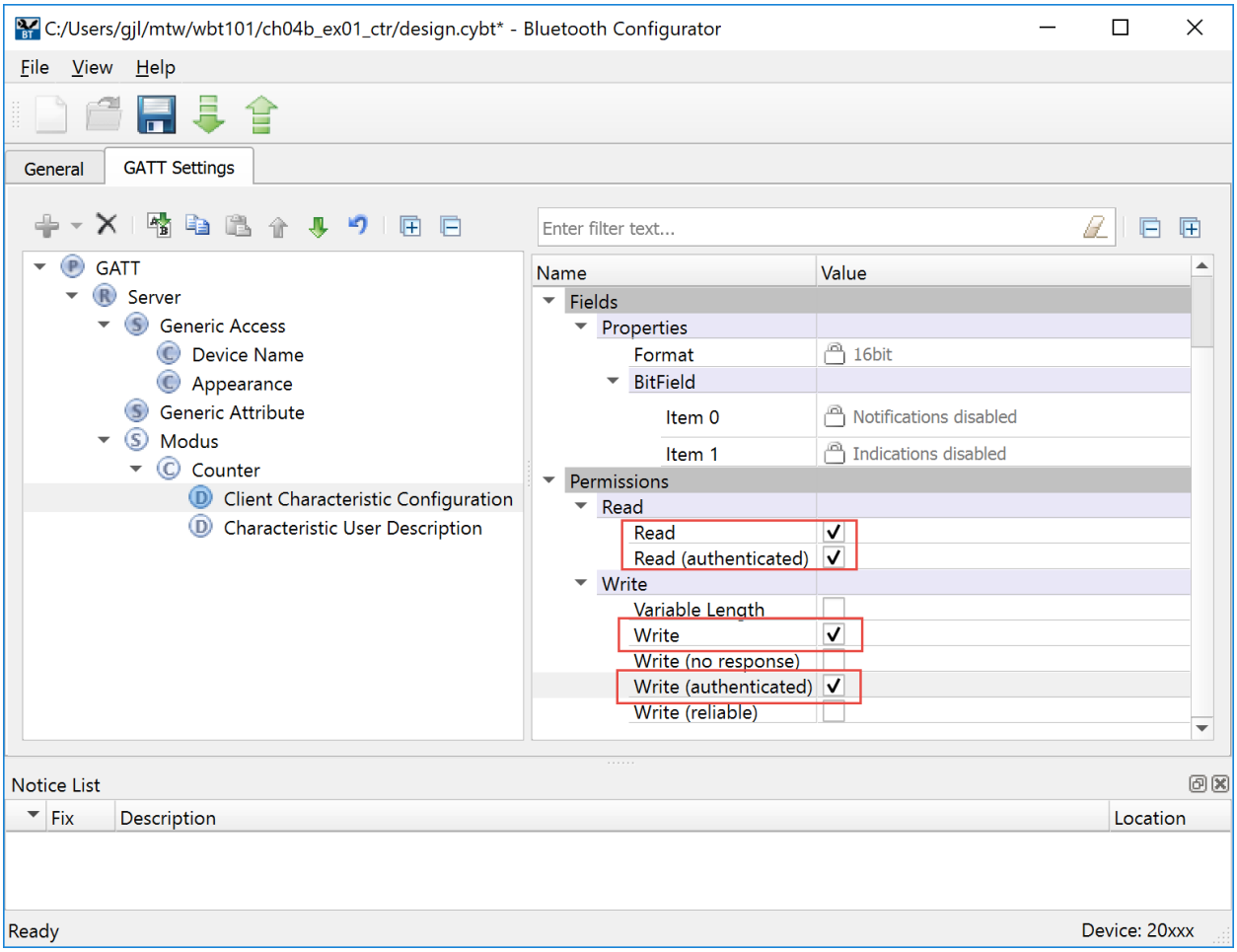
4B.6.4 Authentication, Authorization and the GATT DB

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 not allow someone to turn off/on a switch without entering a password.

4B.6.5 Security in the Bluetooth Configurator

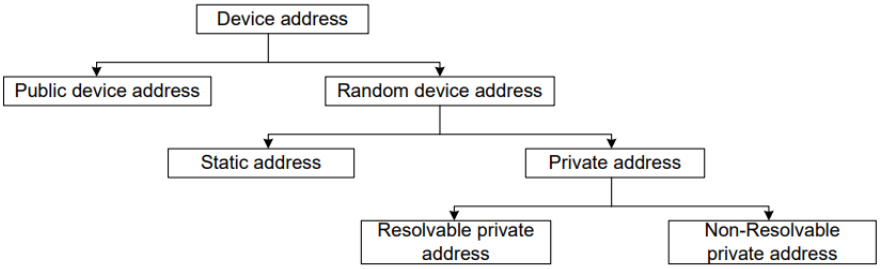
In order to enable security (i.e. to require paring before allowing the client read or write of a characteristic) you just click the "Read (authenticated)" or "Write (authenticated)" button in the permissions for that characteristic. Note that this is a bitmask setting, so you must still keep "Read" and/or "Write" selected when you enable authentication. If not, reads/writes to that characteristic will fail. Enabling security works the same way for Descriptors such as the CCCD. That is, you can require authentication before allowing reads/writes of the CCCD (thereby preventing the client from turning Notifications on/off without pairing first) from the previous example like this:



4B.6.6 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. Link Layer 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. Note that this is different than security (i.e. encrypting of messages).

There are a few different types of address types possible for BLE devices:



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 are generated on first power up or during manufacturing. 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 some interval to ensure that the BLE device cannot be tracked. A Non-Resolvable Private Address cannot be resolved by any device so the peer cannot identify who it is connecting to. Resolvable Private Addresses (RPA) 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 determine the device's address.

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 ModusToolbox, this value is set in app_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.

Apart from this, Bluetooth 5.0 introduced more options in the form of privacy modes. There are two modes: device privacy mode and network privacy mode. A device in device privacy mode is only concerned about the privacy of the device itself and will accept advertising physical channel PDU's (Advertising, Scanning and Initiating packets) from peer devices that contain their identity address as well as ones that contain a private address, even if the peer device has distributed its IRK in the past. In network privacy mode, a device will only accept advertising packets from peer devices that contain a private address. By default, network privacy mode is used when private addresses are resolved and generated by the Controller. The Host can specify the privacy mode to be used with each peer identity on the resolving list. The table below shows the logical representation of the resolving list entries. Depending on the privacy mode entry in the resolving list, the device will behave differently with each peer device.

Device	Local IRK	Peer IRK	Peer Identity Address	Identity Address Type	Privacy Mode
1	Local IRK	Peer 1 IRK	Peer 1 Identity Address	Static/Public	Network/Device
2	Local IRK	Peer 2 IRK	Peer 2 Identity Address	Static/Public	Network/Device
3	Local IRK	Peer 3 IRK	Peer 3 Identity Address	Static/Public	Network/Device

4B.7 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 as long as you return a default value of WICED_BT_SUCCESS.
	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.
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).

Activity	Callback Event Name	Reason
	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_USER_CONFIRMATION_REQUEST_EVT	This event only occurs if the IO capabilities are set such that your device has the capability to display a value and accept Yes/No input, such as BTM_IO_CAPABILITIES_DISPLAY_AND_YES_NO_INPUT. In this event, the firmware should display the passkey so that it can be compared with the value displayed on the Client. This state should also provide confirmation to the Stack (either with or without user input first).
	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 BD_ADDR should be saved in NVRAM at this point. You may also initialize other state information to be saved such as notify settings.
Read Values	GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_READ	The firmware must get the value from the correct location in the GATT database.
Write Values	GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_WRITE	The firmware must store the provided value in the correct location in the GATT database.

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

Activity	Callback Event Name	Reason
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.8 Exercises

Exercise 4B.1 Simple BLE Application with Notifications using Bluetooth Configurator

In this exercise you will create a simple example of notifications without pairing the phone and kit (so the notifications are not authenticated).

1. Create a new application called **ch04b_ex01_ctr** using the template in templates/**ch04b_ex01_ctr**.
2. Follow the instructions in section 4B.3 to use the Bluetooth Configurator to create an application with a Service called Modus and a Characteristic called Counter that will keep track of how many times mechanical button 1 has been pressed and will send a notification if it is enabled by the client.

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

   Hint: Remember to set BT_DEVICE_ADDRESS?=random in the makefile to avoid advertising the same address as other students.

Exercise 4B.2 BLE Pairing and Security

Introduction

In this exercise, you will add Pairing and Security (Encryption) to the previous application.

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.
	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 Button characteristic while pressing button →	GATT_ATTRIBUTE_REQUEST_EVT, GATTS_REQ_TYPE_READ →	Returns button state
Read Button CCCD →	GATT_ATTRIBUTE_REQUEST_EVT, GATTS_REQ_TYPE_READ →	Returns button notification setting
Write 01:00 to Button CCCD →	GATT_ATTRIBUTE_REQUEST_EVT, GATTS_REQ_TYPE_WRITE →	Enables notifications
Press button →		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

Application Creation

1. Create a new application called **ch04b_ex02_pair** using the template in templates/**ch04b_ex02_pair**. This template is the same as application ex01_ctr that you just completed. If you prefer to build on your own code, you can import from your application's folder instead of using the template.
2. The makefile has already been modified to set BT_DEVICE_ADDRESS?=random.
3. Open the Bluetooth Configurator.
    a. Change the Device Name to <inits>_pair.
    b. In the Counter characteristic set the Read (authenticated) permission, which will make the peripheral reject read requests unless the devices are paired.
        i. Hint: You MUST leave "Read" checked also. It will not work with just "Read (authenticated) checked.
    c. Update the Client Characteristic Configuration descriptor to require authenticated read and write. This will cause the application to require pairing to view or change the notification settings.
        i. Hint: You MUST leave "Read" and "Write" checked also.
    d. Save the edits and close the configurator.
4. In app.c, look for the call to `wiced_bt_set_pairable_mode()` mode and set the first argument to WICED_TRUE to allow pairing.
5. In the `BTM_PAIRING_IO_CAPABILITIES_BLE_REQUEST_EVT` management case tell the central that you require MITM protection, but the device has no IO capabilities.

```
p_event_data->pairing_io_capabilities_ble_request.auth_req =
BTM_LE_AUTH_REQ_SC_MITM_BOND;
p_event_data->pairing_io_capabilities_ble_request.init_keys =
BTM_LE_KEY_PENC|BTM_LE_KEY_PID;
p_event_data->pairing_io_capabilities_ble_request.local_io_cap =
BTM_IO_CAPABILITIES_NONE;
```

6. In the `BTM_SECURITY_REQUEST_EVT` management case grant the authorization to the central by using the following code:

```
wiced_bt_ble_security_grant( p_event_data->security_request.bd_addr,
WICED_BT_SUCCESS );
```

Testing

1. Build and program the application to the board.
2. Open the mobile CySmart app.
3. Connect to the device.
4. Open the GATT browser, navigate to the Counter characteristic, press Descriptor and then the Client Characteristic Configuration.
5. If requested, accept the invitation to pair the devices.
6. Return to the Counter, enable notifications and observe the Counter value as you press the button on the kit.
7. Disconnect from the mobile CySmart app.
8. 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 re-connecting. In the next exercise we'll store bonding information on the WICED device so that you will be able to leave the devices paired if you desire.
9. Start the PC CySmart app. Scan for your device. Once it appears in the list, stop scanning and connect to it.
    a. Hint: Hint: Don't forget to update the scan interval and window to 1000 and 100 ms respectively.
10. Click on "Discover all Attributes" and then on "Enable Notifications". Notice that you will get an authentication error. Click "OK" to close the error dialog.
11. Try reading the Counter Characteristic Value manually. Notice that you again get an authentication error. Click "OK" to close the error window.
12. Click on "Pair" and click "No" if you are asked if you want to add the device to the resolving list since we haven't yet enabled privacy.
13. Click on "Enable All Notifications" again. Now when you press the button you will see the characteristic value change.
14. Click on "Disable All Notifications" and then read the Button Characteristic Value manually. It should now work.
15. Click "Disconnect".
16. From the Device List, select a Device Address and select "Clear -> All" since we have not stored bonding information in the device yet.

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 (Advanced) 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 create the application from a template, 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 slowly (1 sec duty cycle) 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. The LED will be ON since the kit is connected. 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 for now). 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.

When you disconnect, LED1 will flash rapidly (200ms duty cycle) to indicate that it is bonded. To remove Bonding information from the kit and return bonding mode, press 'e' in the UART terminal window. This will erase the stored bonding information and put the kit back into Bonding mode. LED1 will now go back to a slow flashing rate. When you reconnect, the key must be entered again to connect. This allows you to Pair/Bond from a Client that has "lost" the bonding information or to Pair/Bond with a new device without having to reprogram the kit.

Application Creation

1. Create a new application called **ch04b_ex03_bond** using the template in templates/**ch04b_ex03_bond**.
2. The makefile has already been modified to set BT_DEVICE_ADDRESS?=random.
3. Open the Bluetooth Configurator.
    a. Change the Device Name to <inits>_bond.
    b. Save the edits and close the configurator.
4. Open app_bt_cfg.c and change the rpa_refresh_timeout from WICED_BT_CFG_DEFAULT_RANDOM_ADDRESS_NEVER_CHANGE to WICED_BT_CFG_DEFAULT_RANDOM_ADDRESS_CHANGE_TIMEOUT. This enables privacy.
5. The remaining code for this exercise has already been implemented for you in app.c
6. Build and program the kit.

Testing

1. Open a UART terminal window to the PUART.
2. Build the application 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 *app_bt_cfg.c*. Also, don't forget to update the scan interval and window to 1000 and 100 ms respectively.
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". Press the button 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.
    a. Hint: You will notice that the LED is flashing rapidly. The firmware was written to do this when it is not connected but has bonding information stored.
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 Device List window at the bottom; both the Device Address and the Public Identity Address are listed. If you click on ‘View …’, some Details concerning the device appear including the Identity Resolving Key. 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. Press the button and observe that notifications are already enabled since they were enabled when you disconnected. This information was retained in NVRAM.
    a. Note: If you use the mobile CySmart app, notifications will be disabled when you reconnect because the app automatically disables notifications before disconnecting.
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 at the bottom of the CySmart window (i.e. click on any device listed and do "Clear -> All".
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 pairing 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.
   b. Hint: It will take a while before pairing times out.
26. Click on Disconnect and close the Authentication failed message window.
27. Press "e" in the UART window and note that LED1 begins flashing slowly. 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 or otherwise clear bonding information since the WICED device 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 Button 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 "bonded" is set as TRUE. Otherwise, it stays FALSE, which means the device can accept new pairing requests.
3. In the BTM_SECURITY_REQUEST_EVENT look to see if bonded is FALSE. Security is only granted if the device is not bonded.
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 bonded to TRUE.

   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. bonded is TRUE), 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 Button CCCD value to the hostinfo structure whenever it is updated and write the value into NVRAM.
12. The UART is configured to accept input. The rx_cback function looks for the key "e". If it has been sent, it sets bonded to FALSE, 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. The PWM that blinks the LED during advertising has two different rates. The PWM is started during initialization, and then stopped/restarted when a disconnect happens or when bonding information is removed. The blink rate is set depending on the value of bonded.
14. 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.4 (Advanced) Add a Pairing Passkey

Introduction

In this exercise, you will modify the previous exercise to require a Passkey to be entered to pair the device the first time. The Passkey will be randomly generated by the stack and you will send it to 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.

Application Creation

1. Create a new application called **ch04b_ex04_pass** using the same template as in the previous exercise - templates/ch04b_ex03_bond.
2. The makefile has already been modified to set BT_DEVICE_ADDRESS?=random.
3. Open the Bluetooth Configurator.
    a. Change the Device Name to <inits>_pass
    b. Save the edits and close the configurator.
4. Open app_bt_cfg.c and change the rpa_refresh_timeout from WICED_BT_CFG_DEFAULT_RANDOM_ADDRESS_NEVER_CHANGE to WICED_BT_CFG_DEFAULT_RANDOM_ADDRESS_CHANGE_TIMEOUT to enable privacy.
5. In app.c, change the pairing_io_capabilities_ble_request.local_iop_cap from `BTM_IO_CAPABILITIES_NONE` to `BTM_IO_CAPABILITIES_DISPLAY_ONLY`
6. Create a new event case statement for `BTM_PASSKEY_NOTIFICATION_EVT` (it is not already in the template code) in `app_management_callback()` and print the value of the Passkey to the UART.
    a. Hint: Make sure you print something (e.g. asterisks) 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 event as:

```
p_event_data->user_passkey_notification.passkey
```

7. Build and program the kit.

Testing

1. Open a UART terminal window.
2. Open the mobile CySmart app.
3. Attempt to connect to the device and then navigate down to the button characteristic in the GATT browser. 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.
4. Once Pairing and Bonding completes, verify that the application still works.
5. Disconnect and reconnect. Observe that the key does not need to be entered to Pair this time.
6. Disconnect, then remove the bonding information from the phone's Bluetooth settings.

7. Press 'e' in the UART terminal to put the kit into Bonding mode (i.e. erase the stored bonding information) and then reconnect. Observe that the key must be entered again to connect.
8. Disconnect again and remove the bonding information from the phone's Bluetooth settings.
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 from the kit. This is necessary since we only allow bonding information from one device to be stored in our firmware. The final 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.5 (Advanced) Add Numeric Comparison

Introduction

In this exercise, you will modify the previous exercise to require the user to compare a 6-digit number on both devices to pair the first time. After comparing that both numbers are the same, the user needs to click "Yes" in CySmart on the PC or in your Phone's Bluetooth connection settings and type "y" in the UART terminal window for the kit before Pairing/Bonding will be allowed.

Note that the code to implement the passkey is kept since it is up to the central to decide which method to use. That is, we have the device capabilities set to BTM_IO_CAPABILITIES_DISPLAY_AND_YES_NO_INPUT. Since the device has both a display and Yes/No input, the central can choose to use either passkey or numeric comparison.

Application Creation

1. Create a new application called **ch04b_ex05_num** using the template in templates/**ch04b_ex05_num**. This is a copy of the solution application for the passkey exercise and so, if you prefer, you can use import from your own application from that exercise.
2. The makefile has already been modified to set BT_DEVICE_ADDRESS?=random.
3. Open the Bluetooth Configurator.
    a. Change the Device Name to <inits>_num
    b. Save the edits and close the configurator.
4. In app.c, change the pairing_io_capabilities_ble_request.local_iop_cap to `BTM_IO_CAPABILITIES_DISPLAY_AND_YES_NO_INPUT`.
5. Create a new event case statement for `BTM_USER_CONFIRMATION_REQUEST_EVT` (it is not already in the template code) in `app_management_callback(`In the event, print the value for the user to confirm with this code.

```
WICED_BT_TRACE("\r\n********************\r\n" );

WICED_BT_TRACE( "\r\nNUMERIC = %06d\r\n\n", p_event_data-
>user_confirmation_request.numeric_value );

WICED_BT_TRACE("\r\nPress 'y' if the codes match and 'n' if they
don't\r\n\n" );

WICED_BT_TRACE("\r\n********************\r\n\n" );
```

6. In the UART receive callback (rx_cback), look for either a 'y' or a 'n' character and send a response back to the stack. If the response is 'y' send WICED_BT_SUCCESS and if the response is 'n' send WICED_BT_ERROR.

```
wiced_bt_dev_confirm_req_reply( WICED_BT_SUCCESS, hostinfo.remote_addr );

wiced_bt_dev_confirm_req_reply( WICED_BT_ERROR, hostinfo.remote_addr );
```

7. Build the application and program the kit.

Testing

1. Open a UART terminal window.
2. Open the PC CySmart app.
3. Attempt to Connect and then Pair to the device. You will see a notification from CySmart asking for you to verify the number printed by both devices is the same. Find the number on the UART terminal window. If the values match, click "Yes" in CySmart, and press 'y' in the UART.
4. Once Pairing and Bonding completes, verify that the application still works.
5. Disconnect and reconnect. Observe that the number does not need to be verified to Pair this time.
6. Disconnect, then clear the Device List in CySmart.
7. Enter "e" in the UART window to put the kit into Bonding mode and then reconnect. Observe that the comparison must be done again to connect.
8. Disconnect again and clear the Device List in CySmart.

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

Introduction

In this exercise, you will create an application from a template, and use it to bond to up to 4 different devices at one time. Note that this application stores bonding information for multiple devices, it does NOT allow multiple simultaneous BLE connections. That is also possible, but it's not demonstrated here.

Note in this example the `wiced_bt_dev_confirm_req_reply` response with WICED_BT_TRUE is automatically sent after printing the numeric code in the `BTM_USER_CONFIRMATION_REQUEST_EVT` instead of looking for user input from the UART. Therefore, you only need to click "Yes" in CySmart to allow pairing. This may be an acceptable implementation depending on the application's security requirements.

Application Creation

1. Create a new application called **ch04b_ex06_mult** using the template in templates/**ch04b_ex06_mult**.
2. The makefile has already been modified to set BT_DEVICE_ADDRESS?=random.
3. Open the Bluetooth Configurator.
    a. Change the Device Name to <inits>_mult.
    b. Save the edits and close the configurator.
4. All the code for this exercise has already been implemented for you.
5. Build and program the kit.

Testing

1. Open a UART terminal window.
2. The device starts out in bonding mode (LED_1 should be flashing slowly – once per second).
3. Open the mobile CySmart app and connect to your device.
4. Open the GATT database. You should get a notification from your phone to verify the numeric code.
5. Once Pairing completes, verify that the application still works. The device will now be in “normal mode”.
6. Disconnect from the device. The LED will be flashing rapidly – once every 200ms. This indicates that the device will not allow any new devices to bond. Keep the bonding information on the phone.
7. To allow bonding another device, press "e" in the UART terminal - the LED will flash slowly.
    a. Note: If you already have the max number of devices bonded and enter "e" in the UART, it will remove the oldest bonded device before going back into bonding mode.
8. Connect to the device using the PC version of CySmart, Discover all Attributes, and Pair with the device.
9. Verify that the application still works.
10. Enter "l" (lower-case letter L) in the UART terminal. You should see a list of the bonded devices on the terminal window.

11. 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.
12. Disconnect from the device, re-connect from the PC, Discover all Attributes, Pair, and verify that the application still works. Again, a passkey should not be required to Pair with the device.
13. Disconnect from the device.
14. Clear the bonding information from the phone and CySmart on the PC.
15. 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 store 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. Update UART receive callback function so that it just toggles whether we are in bonding mode or not when "e" is pressed. 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).
4. Update UART receive callback function that prints bonding information when "l" is pressed.
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.