­Chapter 4C: Even More Advanced BLE

Time 2 Hours

This chapter covers more advanced topics such as low power, OTA firmware upgrade, and mesh networks.

4C.1 Low Power 2

Exercise 4C.1 Power Modes 2

Exercise 4C.2 WICED Low-Power code 4

4C.2 OTA (Over the Air) Upgrade 6

Exercise 4C.1 Design and Architecture 6

4C.3 Mesh Networks 16

4C.4 Exercises 17

Exercise 4C.1 BLE Low Power 17

Exercise 4C.2 OTA Firmware Upgrade (Non-Secure) 19

Exercise 4C.3 (Advanced) OTA Firmware Upgrade (Secure) 21

Exercise 4C.4 (Advanced) Mesh Networks 22

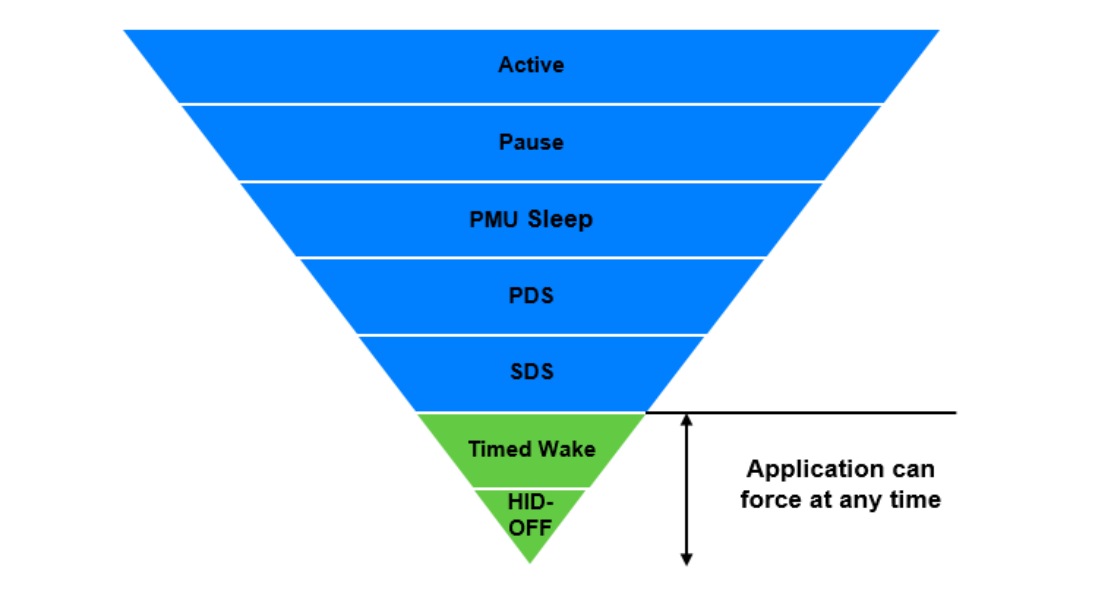
# Low Power

## 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, a.k.a. fastboot) 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 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. |

## 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. You should also include *wiced\_bt\_l2c.h* so that we will be able to update the connection parameters in the firmware.

**wiced\_sleep\_configure()**: Use this function to enable the low power operation of the device. The parameter passed to this function is a pointer to a structure of type wiced\_sleep\_config\_t that contains the sleep configuration information. It looks like this:

/\*\* Sleep configuration parameters \*/

**typedef** **struct**

{

wiced\_sleep\_mode\_type\_t sleep\_mode; /\*\*< Requested sleep mode \*/

wiced\_sleep\_wake\_type\_t host\_wake\_mode; /\*\*< Active level for host wake \*/

wiced\_sleep\_wake\_type\_t device\_wake\_mode; /\*\*< Active level for device wake \*/

uint8\_t device\_wake\_source; /\*\*< Device wake source(s). GPIO mandatory for

WICED\_SLEEP\_MODE\_TRANSPORT \*/

uint32\_t device\_wake\_gpio\_num; /\*\*< GPIO# for host wake, mandatory for

WICED\_SLEEP\_MODE\_TRANSPORT \*/

wiced\_sleep\_allow\_check\_callback sleep\_permit\_handler; /\*\*< Call back to be called by sleep framework

to poll for sleep permission \*/

}wiced\_sleep\_config\_t;

The elements in the structure are:

**sleep\_mode**: This can be either *WICED\_SLEEP\_NO\_TRANSPORT* or *WICED\_SLEEP\_MODE\_TRANSPORT* depending on whether you want to allow sleep when transport is connected or not.

**host\_wake\_mode**: This can be either *WICED\_SLEEP\_WAKE\_ACTIVE\_LOW* or *WICED\_SLEEP\_WAKE\_ACTIVE\_HIGH* depending on the polarity of the wake interrupt desired for the host.

**device\_wake\_mode**: This can be either *WICED\_SLEEP\_WAKE\_ACTIVE\_LOW* or *WICED\_SLEEP\_WAKE\_ACTIVE\_HIGH* depending on the polarity of the wake interrupt desired for the device.

**device\_wake\_source**: The wake source can be keyscan, quadrature sensor, GPIO, or a combination of those. For example, you may want to use an interrupt from a sensor as a GPIO wake source so that the device wakes whenever new sensor data is available.

/\*\* Wake sources.\*/

**#define** WICED\_SLEEP\_WAKE\_SOURCE\_KEYSCAN (1<<0) /\*\*< Enable wake from keyscan \*/

**#define** WICED\_SLEEP\_WAKE\_SOURCE\_QUAD (1<<1) /\*\*< Enable wake from quadrature sensor \*/

**#define** WICED\_SLEEP\_WAKE\_SOURCE\_GPIO (1<<2) /\*\*< Enable wake from GPIO \*/

**device\_wake\_gpio\_num**: If the device\_wake\_source includes GPIO, this entry specifies which GPIO is to be used for waking from sleep.

**sleep\_permit\_handler**: This element requires you to provide a function pointer for callback function that will be called by the PMU to poll for sleep permission and when sleep is entered. The function takes one argument of type wiced\_sleep\_poll\_type\_t which specifies the reason for the callback (*WICED\_SLEEP\_POLL\_SLEEP\_PERMISSION* or *WICED\_SLEEP\_POLL\_TIME\_TO\_SLEEP*) and it returns a uint32\_t.

For a sleep permission callback, the function 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.

The value returned will depend on the current state of the system. Global variables will typically be set in various BLE stack callback events so that sleep permit handler can determine what type of sleep (if any) should be allowed at any given time.

For a sleep entry callback, the function must return the maximum time that the system should be allowed to sleep. This is typically set to *WICED\_SLEEP\_MAX\_TIME\_TO\_SLEEP*.

**wiced\_sleep\_get\_boot\_mode**: This function can be called in the application\_start function to determine whether the chip is starting up for the first time (cold boot) or coming out of SDS (warm boot or fast boot). It returns a variable of type *wiced\_sleep\_boot\_type\_t* which can be *WICED\_SLEEP\_COLD\_BOOT* or *WICED\_SLEEP\_FAST\_BOOT*. Settings such as GPIO configuration are retained during SDS so they only need to be configured for a cold boot.

In the application initialization, it is important to check to see if the device is already connected in the case of a fast boot since the device can stay connected during SDS. If it is already connected, then advertisements should not be started.

Values that need to be retained during SDS should be stored in Always-On-RAM. For example, you should store whether the device is connected or not so that when fast boot occurs you will be able to determine if the device was connected at when it went to sleep. You may also need to store sensor values so that you can determine if they have changed from when the device went to sleep (e.g. to determine if a notification should be sent). To store values in AON RAM, use *PLACE\_IN\_ALWAYS\_ON\_RAM* when declaring the variable. For example:

PLACE\_IN\_ALWAYS\_ON\_RAM uint16\_t connection\_id;

# OTA (Over the Air) Upgrade

The firmware upgrade feature provided in WICED Studio allows an external device to use the Bluetooth link to transfer and install a newer firmware version on devices equipped with CYW20719 (as well as CYW20706 and CYW20735) chips. This section describes the functionality of the WICED Firmware Upgrade library used in various WICED Studio sample applications.

The library is split into two parts. The over the air (OTA) firmware upgrade module of the library provides a simple implementation of the GATT procedures to interact with the device performing the upgrade. The firmware upgrade HAL module of the library provides support for storing data in the non-volatile memory and switching the device to use the new firmware when the upgrade is completed. The embedded application may use the OTA module functions (which in turn use the HAL module functions), or the application may choose to use the HAL module functions directly.

The library contains functionality to support secure and non-secure versions of the upgrade. In the non-secure version, a simple CRC32 verification is performed to validate that all bytes that have been sent from the device performing the upgrade are correctly saved in the serial flash of the device. The secure version of the upgrade validates that the image is correctly signed and has correct production information in the header.

## Design and Architecture

External or on-chip flash memory of the Cypress WICED chips is organized into two partitions for the failsafe upgrade capability. During the startup operation the boot code of the chip checks the first partition and if a valid image is found, assumes that the first partition is active and then starts executing the code in the first partition. If the first partition does not contain a valid image, the boot code checks the second partition and then starts execution of the code in the second partition if a valid image is found there. If neither partition is valid, the boot code enters download mode and waits for the code to be downloaded over HCI UART. The addresses of the partitions are programmed in a file with extension “btp” located in the platform directory of the SDK.

The firmware upgrade process stores received data in the inactive partition. When the download procedure is completed and the received image is verified and activated, the currently active partition is invalidated, and then the chip is restarted. After the chip reboots, the previously inactive partition becomes active. If for some reason the download or the verification step is interrupted, the valid partition remains valid and the chip is not restarted. This guarantees the failsafe procedure.

The following table shows the recommended memory configuration for an application upgrading the firmware on a device with external 4Mbit serial flash:

|  |  |  |  |
| --- | --- | --- | --- |
| **Section Name** | **Offset** | **Length** | **Description** |
| Static Section (SS) | 0x0000 | 0x2000 | Static section used internally by the chip firmware |
| Volatile Section (VS1) | 0x2000 | 0x1000 | First volatile section used for the application and the stack to store data in the external or on-chip flash memory. One serial flash sector. |
| Volatile Section (VS2) | 0x3000 | 0x1000 | Used internally by the firmware when VS1 needs to be defragmented. |
| Data Section (DS1) | 0x4000 | 0x3E000 | First partition. |
| Data Section (DS2) | 0x42000 | 0x3E000 | Second partition. |

During an OTA upgrade the device performing the procedure (Downloader) pushes chunks of the new image to the device being upgraded. The embedded application receives the image and stores it in the external or on-chip flash. When all the data has been transferred, the Downloader sends a command to verify the image and passes a 32-bit CRC checksum. The embedded app reads the image from the flash and verifies the image as follows. For the non-secure download, the library calculates the CRC and verifies that it matches received CRC. For the secure download case, the library performs ECDSA verification and verifies that the Product Information stored in the new image is consistent with the Product Information of the firmware currently being executed on the device. If verification succeeds, the embedded application invalidates the active partition and restarts the chip. The simple CRC check can be easily replaced with crypto signature verification if desired, without changing the download algorithm described in this document.

### Applications for Loading New Firmware

WICED Studio contains two peer applications that can be used to transmit new firmware – one for Android and one for Windows over BLE. Both applications contain the source file as well as pre-compiled executables (.apk for Android and .exe for Windows). The Windows executable is provided for 32-bit (x86) and 64-bit (x64) architectures.

These peer applications can be found in the peer\_apps folder inside the ota\_firmware\_upgrade application folder, or in the file system at:

*<SDK Install Dir>\WICED-Studio-n.n\common\peer\_apps\ota\_firmware\_upgrade\Windows\WsOtaUpgrade\Release*

*<SDK Install Dir>\WICED-Studio-n.n\common\peer\_apps\ota\_firmware\_upgrade\Andoird\LeOTAApp\app\build\outputs\apk*

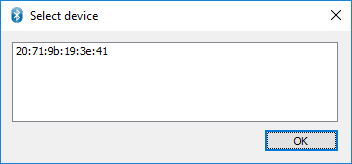
The default <SDC Install Dir> is in your Documents folder.

#### Windows

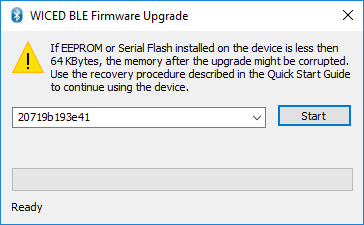
To use the Windows peer application, you must first copy the \*.bin file from the build directory of the WICED application into the same folder as the Windows peer application. Then run the application with the \*.bin file provided as an argument. For example, from a command or PowerShell window:

*.\WsOtaUpgrade.exe ex01\_ota-WW101\_2\_CYW900719Q40EVB\_01-rom-ram-Wiced-release.ota.bin*

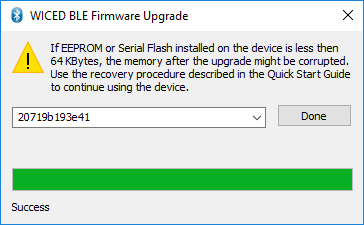
You will get a window that looks like the following. Select the device you want to update and click "OK".



On the next window, verify that the device type is correct and click "Start".



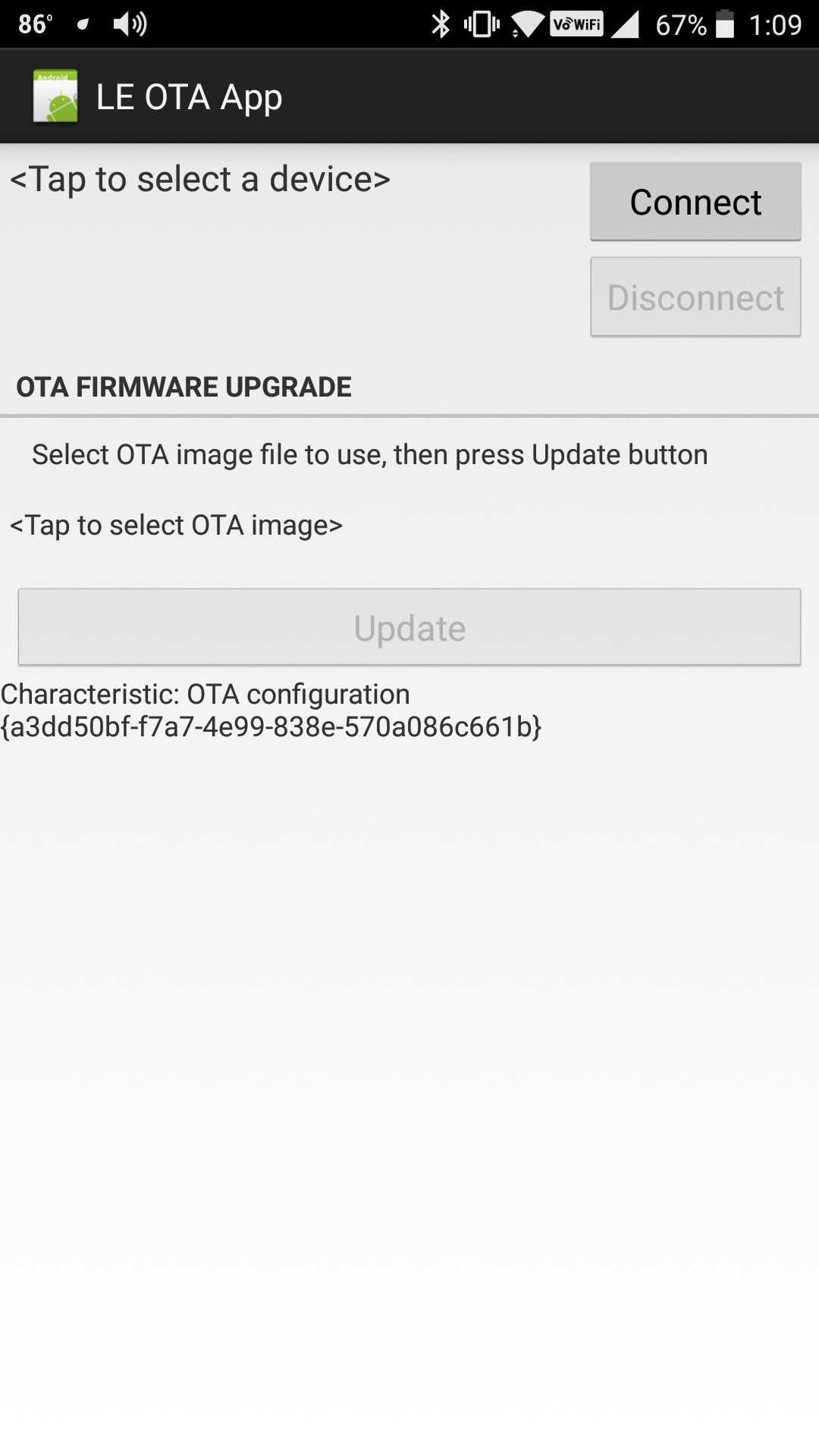
If the update worked, the window will show "Success" at the bottom. Click "Done" to close the window.



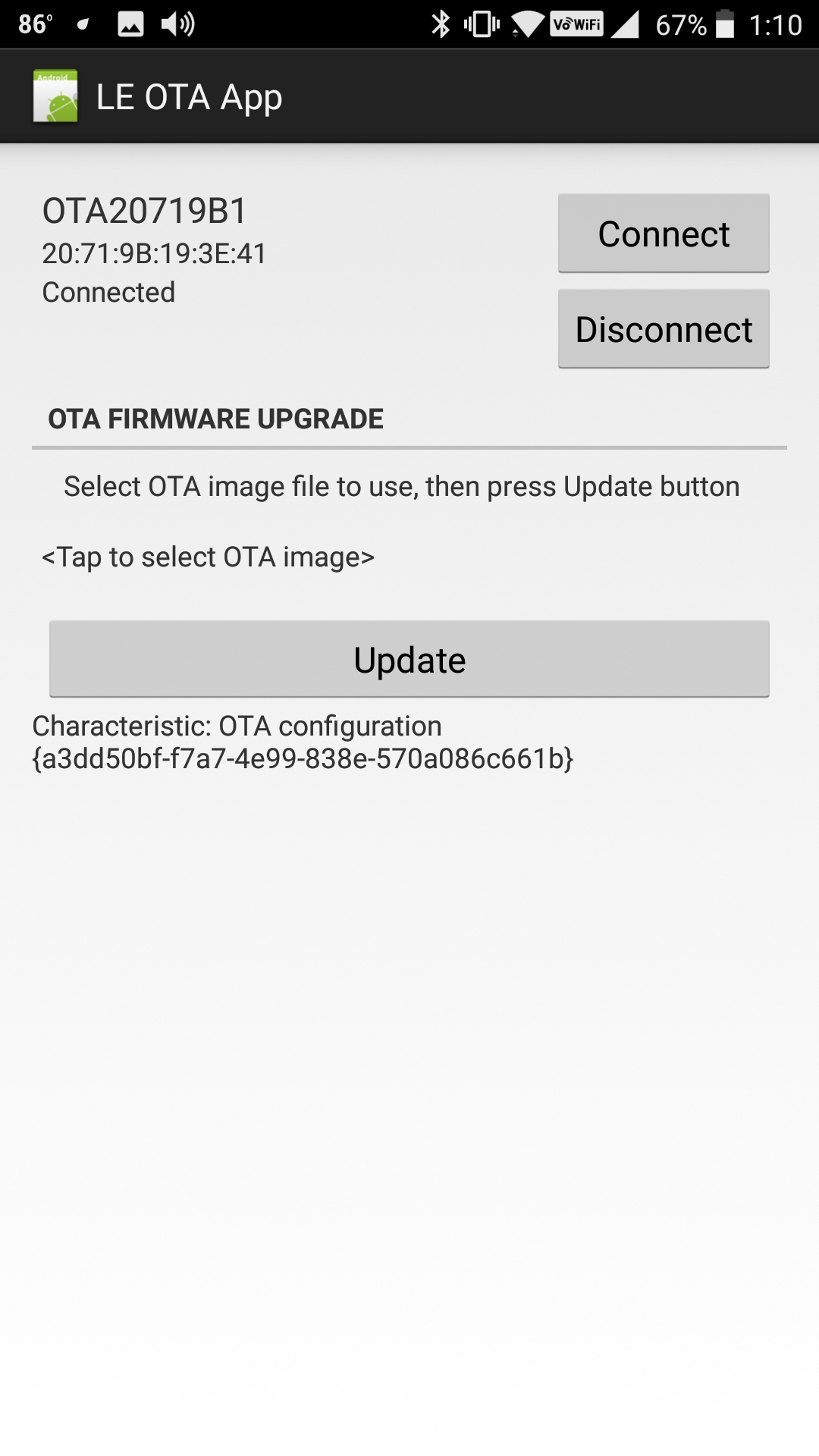
#### Android

To use the Android app:

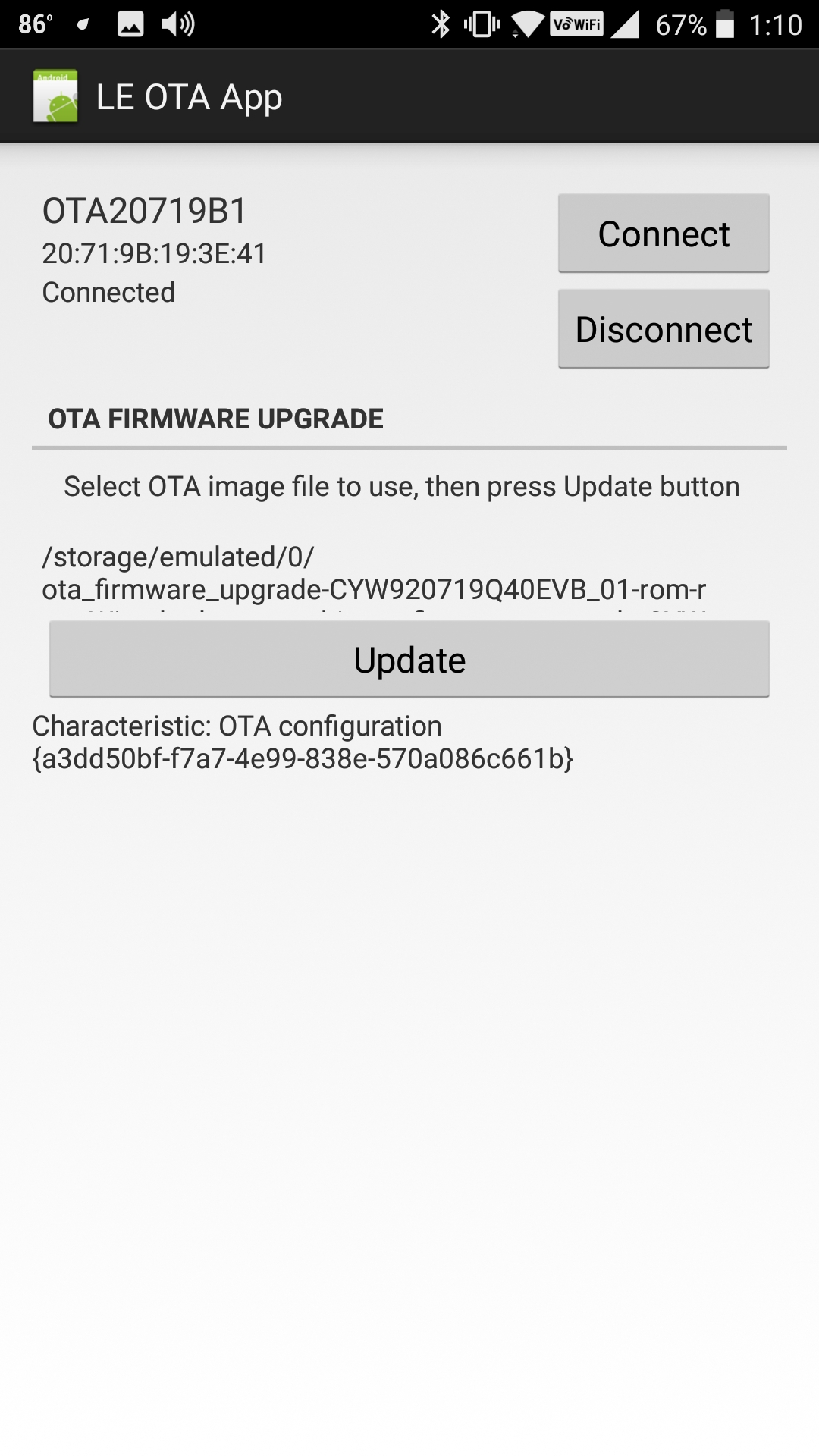
1. Install the app-debug.apk file on your Android device if you have not already done so.
2. Copy the \*.bin file from the build directory onto the device in a location where you can find it.
3. Run the app called *LE OTA App*. The startup screen will look like this:



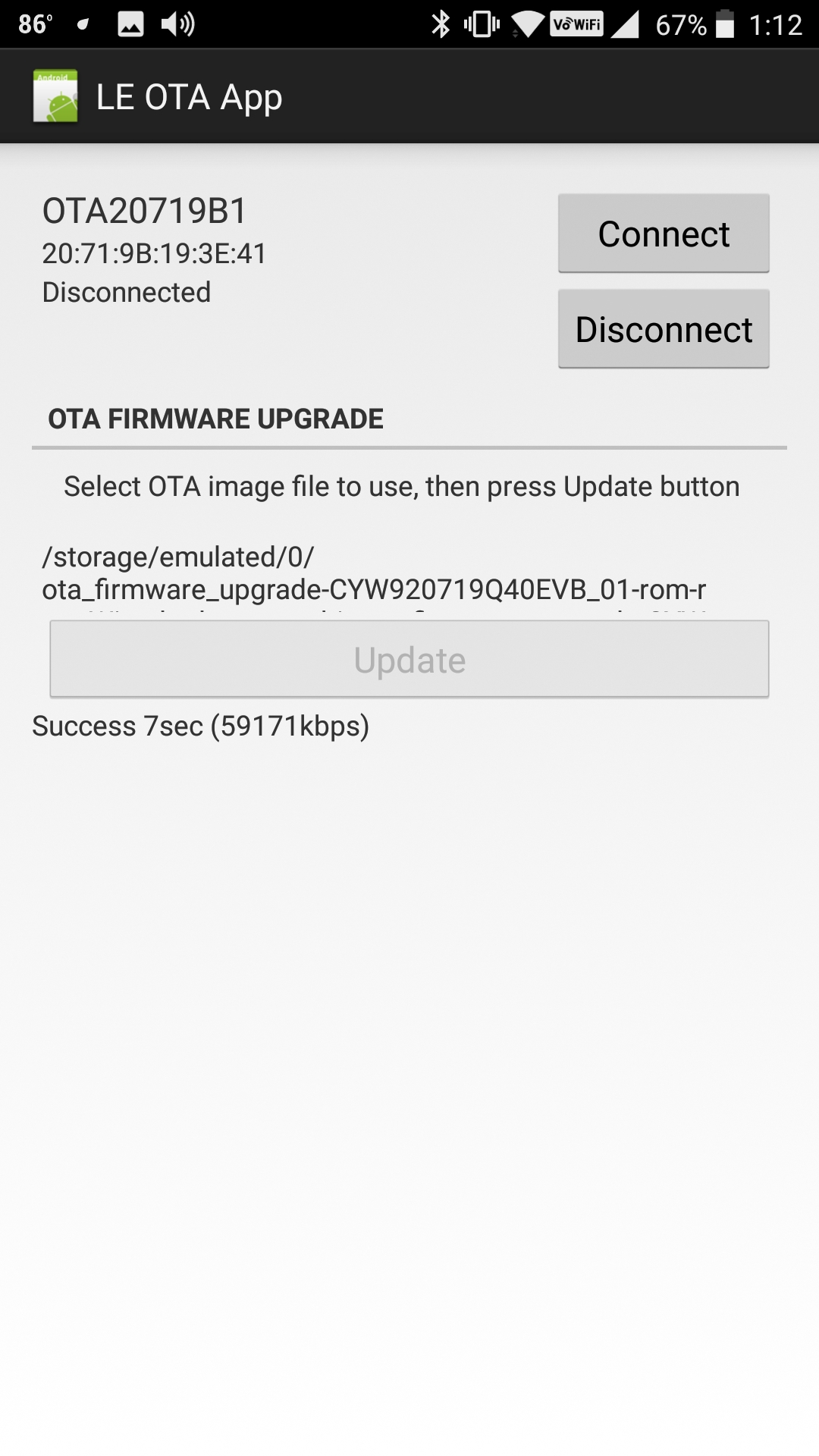
1. Tap where it says <Tap to select a device> and choose your device from the list.
2. Tap on the "Connect" button. Once connected, the screen will look like this:



1. Tap where it says <Tap to select OTA Image>, navigate to where you saved the \*.bin file on your device and select it. Once the file is selected, the screen will look like this:



1. Tap the Update button. Once the update is done, you should see "Success" at the bottom of the screen. Disconnect from the device and close the app.



### OTA Firmware

In the firmware, OTA requires the following:

#### Header Files

Include the following header files at the top of your main C file:

**#include** "wiced\_bt\_firmware\_upgrade.h"

**#include** "wiced\_bt\_fw\_upgrade.h"

#### Library

Include the OTA library in the makefile.mk:

**$(NAME)\_COMPONENTS :**= fw\_upgrade\_lib.a

#### BLE OTA Service (Non-Secure)

The GATT database must have a Primary Service defined for the OTA service. The Service and its two Characteristics are defined as follows:

// OTA Firmware Upgrade Service

PRIMARY\_SERVICE\_UUID128(HANDLE\_OTA\_FW\_UPGRADE\_SERVICE, UUID\_OTA\_FW\_UPGRADE\_SERVICE),

CHARACTERISTIC\_UUID128\_WRITABLE(HANDLE\_OTA\_FW\_UPGRADE\_CHARACTERISTIC\_CONTROL\_POINT,

HANDLE\_OTA\_FW\_UPGRADE\_CONTROL\_POINT, UUID\_OTA\_FW\_UPGRADE\_CHARACTERISTIC\_CONTROL\_POINT,

LEGATTDB\_CHAR\_PROP\_WRITE | LEGATTDB\_CHAR\_PROP\_NOTIFY | LEGATTDB\_CHAR\_PROP\_INDICATE,

LEGATTDB\_PERM\_VARIABLE\_LENGTH | LEGATTDB\_PERM\_WRITE\_REQ /\*| LEGATTDB\_PERM\_AUTH\_WRITABLE\*/

),

CHAR\_DESCRIPTOR\_UUID16\_WRITABLE(HANDLE\_OTA\_FW\_UPGRADE\_CLIENT\_CONFIGURATION\_DESCRIPTOR,

UUID\_DESCRIPTOR\_CLIENT\_CHARACTERISTIC\_CONFIGURATION, LEGATTDB\_PERM\_READABLE |

LEGATTDB\_PERM\_WRITE\_REQ /\*| LEGATTDB\_PERM\_AUTH\_WRITABLE \*/),

CHARACTERISTIC\_UUID128\_WRITABLE(HANDLE\_OTA\_FW\_UPGRADE\_CHARACTERISTIC\_DATA,

HANDLE\_OTA\_FW\_UPGRADE\_DATA, UUID\_OTA\_FW\_UPGRADE\_CHARACTERISTIC\_DATA,

LEGATTDB\_CHAR\_PROP\_WRITE, LEGATTDB\_PERM\_VARIABLE\_LENGTH | LEGATTDB\_PERM\_WRITE\_REQ /\*|

LEGATTDB\_PERM\_AUTH\_WRITABLE \*/),

Note that the LEGATTBD\_PERM\_AUTH\_WRITABLE are commented out in all three of the above so writes can be done without an authenticated link. Uncomment them if you want an authenticated link before allowing OTA updates.

#### Initialization

During the application initialization (typically just after initializing the GATT database with wiced\_bt\_gatt\_db\_init), the following function call must be made:

/\* Initialize OTA (non-secure) \*/

wiced\_ota\_fw\_upgrade\_init(NULL, NULL);

#### GATT Event Handler Functions

The handler functions for the GATT events *GATTS\_REQ\_TYPE\_READ*, *GATTS\_REQ\_TYPE\_WRITE*, and *GATTS\_REQ\_TYPE\_CONF* must call the appropriate OTA functions. Note that these are in addition to any other functionality required for the normal application functionality.

If your starting project does not have an indication confirmation handler function, it must be created, and that case must be added to the GATT event callback function:

**case** *GATTS\_REQ\_TYPE\_CONF*:

status = ex02\_ota\_indication\_cfm\_handler(p\_data->conn\_id, p\_data->data.handle);

**break**;

Handler for *GATTS\_REQ\_TYPE\_READ:*

wiced\_bt\_gatt\_status\_t **ex02\_ota\_read\_handler**( wiced\_bt\_gatt\_read\_t \*p\_read\_req, uint16\_t conn\_id )

{

wiced\_bt\_gatt\_status\_t status = *WICED\_BT\_GATT\_INVALID\_HANDLE*;

**switch**(p\_read\_req->handle)

{

// If the indication is for an OTA service handle, pass it to the library to process

**case** HANDLE\_OTA\_FW\_UPGRADE\_SERVICE:

**case** HANDLE\_OTA\_FW\_UPGRADE\_CHARACTERISTIC\_CONTROL\_POINT:

**case** HANDLE\_OTA\_FW\_UPGRADE\_CONTROL\_POINT:

**case** HANDLE\_OTA\_FW\_UPGRADE\_CLIENT\_CONFIGURATION\_DESCRIPTOR:

**case** HANDLE\_OTA\_FW\_UPGRADE\_CHARACTERISTIC\_DATA:

**case** HANDLE\_OTA\_FW\_UPGRADE\_DATA:

**case** HANDLE\_OTA\_FW\_UPGRADE\_CHARACTERISTIC\_APP\_INFO:

**case** HANDLE\_OTA\_FW\_UPGRADE\_APP\_INFO:

status = wiced\_ota\_fw\_upgrade\_read\_handler(conn\_id, p\_read\_req);

**break**;

**default**:

// Handle normal (non-OTA) read requests here

}

**return** status;

}

Handler for *GATTS\_REQ\_TYPE\_WRITE*:

wiced\_bt\_gatt\_status\_t **ex01\_ota\_write\_handler**( wiced\_bt\_gatt\_write\_t \*p\_write\_req, uint16\_t conn\_id )

{

wiced\_bt\_gatt\_status\_t status = *WICED\_BT\_GATT\_INVALID\_HANDLE*;

**switch**(p\_write\_req->handle)

{

// If the indication is for an OTA service handle, pass it to the library to process

**case** HANDLE\_OTA\_FW\_UPGRADE\_SERVICE:

**case** HANDLE\_OTA\_FW\_UPGRADE\_CHARACTERISTIC\_CONTROL\_POINT:

**case** HANDLE\_OTA\_FW\_UPGRADE\_CONTROL\_POINT:

**case** HANDLE\_OTA\_FW\_UPGRADE\_CLIENT\_CONFIGURATION\_DESCRIPTOR:

**case** HANDLE\_OTA\_FW\_UPGRADE\_CHARACTERISTIC\_DATA:

**case** HANDLE\_OTA\_FW\_UPGRADE\_DATA:

**case** HANDLE\_OTA\_FW\_UPGRADE\_CHARACTERISTIC\_APP\_INFO:

**case** HANDLE\_OTA\_FW\_UPGRADE\_APP\_INFO:

wiced\_set\_debug\_uart( *WICED\_ROUTE\_DEBUG\_NONE*); // This is needed due to a timing issue with the Windows peer client

status = wiced\_ota\_fw\_upgrade\_write\_handler(conn\_id, p\_write\_req);

**break**;

**default**:

// Handle normal (non-OTA) read requests here

}

**return** status;

}

*Handler for GATTS\_REQ\_TYPE\_CONF:*

wiced\_bt\_gatt\_status\_t **ex01\_ota\_indication\_cfm\_handler**(uint16\_t handle, uint16\_t conn\_id)

{

wiced\_bt\_gatt\_status\_t status = *WICED\_BT\_GATT\_INVALID\_HANDLE*;

**switch**(handle)

{

// If the indication is for an OTA service handle, pass it to the library to process

**case** HANDLE\_OTA\_FW\_UPGRADE\_SERVICE:

**case** HANDLE\_OTA\_FW\_UPGRADE\_CHARACTERISTIC\_CONTROL\_POINT:

**case** HANDLE\_OTA\_FW\_UPGRADE\_CONTROL\_POINT:

**case** HANDLE\_OTA\_FW\_UPGRADE\_CLIENT\_CONFIGURATION\_DESCRIPTOR:

**case** HANDLE\_OTA\_FW\_UPGRADE\_CHARACTERISTIC\_DATA:

**case** HANDLE\_OTA\_FW\_UPGRADE\_DATA:

**case** HANDLE\_OTA\_FW\_UPGRADE\_CHARACTERISTIC\_APP\_INFO:

**case** HANDLE\_OTA\_FW\_UPGRADE\_APP\_INFO:

status = wiced\_ota\_fw\_upgrade\_indication\_cfm\_handler(conn\_id, handle);

**break**;

**default**:

// Handle normal (non-OTA) indication confirmation requests here

}

**return** status;

}

#### Buffer Pool Sizes

The large buffer pool (defined at the bottom of wiced\_bt\_cfg.c) should be increased from the default size of 512 to a size of 1024 for OTA.

#### Disabling of PUART

Note in the WRITE handler above, the debug UART is disabled before calling the OTA library write handler function. This is only required if the PUART is being used. For some reason, the PUART interferes with the OTA process when using the Windows peer app and causes the update to fail frequently.

### Secure OTA

To use secure OTA firmware upgrade, we must create a key pair (public/private) and make a few changes in the firmware. The changes are shown in detail below.

#### Key Generation

Tools are provided in the WICED SDK to create, sign, and verify random keys. Executables for Windows can be found in:

<WICED\_SDK\_INSTALL\_DIR>/wiced\_tools/ecdsa256/Win32

The steps are:

1. Double-click on ecdsa\_genkey.exe from Windows explorer to run the program. This will generate random keys. Note that if you re-run the program, it will overwrite any existing key files. The files created are:
   1. ecdsa256\_key.pri.bin
   2. ecdsa256\_key.pub.bin
   3. ecdsa256\_key\_plus.pub.bin
   4. ecdsa256\_pub.c
2. Copy the file ecdsa256\_pub.c to the application folder.
3. The OTA file will be signed once it is generated below.

#### Include Keys in Firmware

Add a line to the project's makefile.mk to include the ecdsa256\_pub.c file. For example:

**APP\_SRC +**= ecdsa256\_pub.c

#### Header files and Global Variables

Add the following header files to the main application C file:

**#include** "bt\_types.h"

**#include** "p\_256\_multprecision.h"

**#include** "p\_256\_ecc\_pp.h"

Add an external global variable declaration of type "Point" for the public key that is defined in ecdsa256\_pub.c. For example:

**extern** Point ecdsa256\_public\_key;

#### BLE OTA Service (Secure)

In the OTA BLE Service description in the GATT database, change the UUID for the service to the UUID for secure OTA. The change is shown here in bold:

// OTA Firmware Upgrade Service

PRIMARY\_SERVICE\_UUID128(HANDLE\_OTA\_FW\_UPGRADE\_SERVICE, **UUID\_OTA\_SEC\_FW\_UPGRADE\_SERVICE**)

#### Initialization

In the firmware initialization section, change the first argument to the OTA init function from NULL to a pointer to a public key that was generated earlier. For example:

/\* Initialize OTA (secure) \*/

wiced\_ota\_fw\_upgrade\_init**(&ecdsa256\_public\_key**, NULL);

#### Build Firmware and Sign OTA Image

To build the firmware and generate then convert the output file to a signed output file follow these steps:

1. Build the firmware as usual.
2. Once the firmware is built, copy the bin file from the build folder for the application to the <WICED\_SDK\_INSTALL\_DIR>/wiced\_tools/ecdsa256/Win32 folder.
3. Open a command terminal or power shell window (shift-right-click in the folder from Windows explorer) and enter the command:

.\ecdsa\_sign.exe .\<fileName>.bin

This will produce a file called <fileName>.bin.signed.

1. Load the signed file into the device using the preferred OTA tool (i.e. Windows or Android).

# Mesh Networks

TBD

# Clients

TBD

# Exercises

## BLE Low Power

### Introduction

In this exercise, you will copy a template project that implements low power. You will measure the power consumption in different power modes. The template is based on the previous exercise that stores BLE bonding information in NVRAM.

### Project Creation

1. Copy the project from the class files under templates/ch04c/ex01\_low\_power.
2. Create a Make Target.
3. Change "key" to your initials in ex01\_low\_power.c and wiced\_app\_cfg.c so that you will be able to find your device.

### Testing

Use the steps listed below to fill in the table with measured current consumption values:

|  |  |  |
| --- | --- | --- |
| **Condition** | **Low Power** | |
| **Enabled** | **Disabled** |
| Advertising |  |  |
| Connected |  |  |
| Paired |  |  |
| Notifications Enabled – Notification Not Being Sent |  |  |
| Notifications Enabled – Notification Being Sent |  |  |

Note: the PUART debug printing will increase power from the device. If you want to see power numbers without debug printing change *wiced\_set\_debug\_uart(WICED\_ROUTE\_DEBUG\_TO\_PUART)* to *wiced\_set\_debug\_uart(WICED\_ROUTE\_DEBUG\_NONE)* in the firmware before building/programming the kit.

1. Connect an ammeter across jumper J15 to allow current measurement.
2. Download the project onto the kit.
3. Open a UART terminal window (note: this won't display anything if you disabled the PUART in the firmware).
4. Open the PC CySmart app.
5. Record the current (Advertising)
6. Click ‘Configure Master Settings’, then ‘Connection Parameters’, enter the following values, and hit OK.
   1. Connection Interval Minimum: 100ms
   2. Connection Interval Maximum: 100ms
   3. Supervision Timeout: 5120ms
7. Attempt to Connect to the device. You will see a notification asking to confirm the connection parameters. Select ‘Yes’.
8. Record the current (Connected)
9. 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.
10. Record the current (Paired)
11. Once Pairing completes, verify that the application still works. The device will now be in SDS mode.
12. 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.
13. Record the current (Notifications Enabled – Notification Not Being Sent)
14. Tap one of the CapSense buttons repeatedly so that notifications are sent.
15. Record the current (Notifications Enabled – Notification Being Sent)
16. Once the device is done sending the series of notifications it will go back to sleep.
17. Compare the power consumption values between when the device is sending notifications and when it is sleeping.
18. Comment out the line containing the call to wiced\_sleep\_configre to disable low power and repeat the current measurements.

### Questions

1. Which variable is used to control sleep permissions? What are its states?
2. What is printed to the UART when:
3. A sleep request is denied: \_\_\_\_\_
4. A sleep request is allowed without shutdown: \_\_\_\_\_
5. A sleep request is allowed with shutdown: \_\_\_\_\_
6. When is sleep not allowed? When is sleep allowed but without shutdown? When is sleep allowed with shutdown?
7. What is done differently for a cold boot vs. a fast boot? Why?
8. What is AON RAM? What values is it used for and why?
9. Where are the connection parameters updated? Why?

OTA Firmware Upgrade (Non-Secure)

### Introduction

In this exercise, you will modify chapter 4A exercise 1 to add OTA firmware upgrade capability. Once OTA support is added, you will modify the project to control 2 LEDs instead of just one and you will upload the new firmware using OTA.

### Project Creation

1. Copy Chapter 4A, Exercise 1 to a new folder (i.e. ch04c). Rename the project folder and project files to *ex02\_ota*.
   1. Hint: Don't forget to update header file names in the two C files and don't forget to update the source file names in the makefile.
   2. Hint: Change the device name from *<inits>\_LED* to *<inits>\_ota* in the wiced\_bt\_cfg.c file and the <inits>\_ota.c file.
   3. Hint: Many function names and variable names start with *key\_led*. You can do a global search/replace to change these to *ex01\_ota* if you want them to be consistent with the project name. Make sure you do this in the ex01\_ota\_db.h file too.
   4. Hint: Delete the .wic file since it is no longer a valid starting point for this project.
2. Create a Make Target for the project and verify that it still builds before proceeding.
3. Review the OTA Firmware section in this chapter and update the files as necessary:
   1. Add additional header files to the C files
   2. Add the OTA library to makefile.mk
   3. Add the OTA service to ex01\_ota\_db.c
   4. Update the GATT event handler functions
   5. Increase the large buffer pool size to 1024
   6. Disable the PUART before OTA write

### Testing

1. Build the project and program it to your kit.
2. Use CySmart to make sure the project functions as expected. Write values of 00, 01, 02, and 03 to the LED characteristic. The LED should only turn on for a value of 01.
   1. Hint: The Service with a single Characteristic is the LED Service and the Service with two Characteristics is the OTA Service.
3. Disconnect from the kit in CySmart.
4. Unplug the kit from your computer. This will ensure that OTA is used instead of regular programming to update the firmware.
5. Update the project so that the values written control the LEDs like this:

|  |  |  |
| --- | --- | --- |
| **Characteristic Value** | **LED2** | **LED1** |
| 00 | OFF | OFF |
| 01 | OFF | ON |
| 02 | ON | OFF |
| 03 | ON | ON |

1. Copy the Make Target for the project and change "download" to "build". This will allow you to build the project without it trying to download to the kit.
2. Build the project.
3. Connect your kit directly to a power outlet using a USB charger.
4. Use OTA to update your kit. You can use either the Windows or the Android app.
   1. Hint: Don't forget to copy over the \*.bin file from the build folder every time you re-build the project so that you are updating the latest firmware.
   2. Hint: If you need to find the Bluetooth Address of your device, use CySmart (either PC or Phone) to scan for it.
   3. Hint: If the OTA process fails on Windows, try resetting the kit and trying again. If that still fails, try using the Android version since it is more robust.
5. Once OTA upgrade is done, connect to the kit using CySmart and verify that the new firmware functionality is working.

(Advanced) OTA Firmware Upgrade (Secure)

### Introduction

In this exercise, you will update the previous exercise to use Secure OTA firmware upgrade.

### Project Creation

1. Copy and rename the previous exercise to *ex03\_ota\_secure*.
   1. Hint: Don't forget to update header file names in the two C files and don't forget to update the source file names in the makefile.
   2. Hint: Change the device name from *<inits>\_ota* to *<inits>\_otas* in the wiced\_bt\_cfg.c file and the <inits>\_ota\_secure.c file.
   3. Hint: Many function names and variable names start with <inits>*\_ota*. You can do a global search/replace to change these to *ex03\_ota\_secure* if you want them to be consistent with the project name. Make sure you do this in the ex03\_ota\_secure\_db.h file too.
2. Create a Make Target for the project and verify that it still builds before proceeding.
3. Review the Secure OTA Firmware section in this chapter to generate the keys and and update the files as necessary.

### Testing

1. Build the project and program it to your kit.
2. Use CySmart to make sure the project functions as expected.
3. Disconnect from the kit in CySmart.
4. Unplug the kit from your computer. This will ensure that OTA is used instead of regular programming to update the firmware.
5. Update the project to change the LED behavior in some way.
6. Copy the Make Target for the project and change "download" to "build". This will allow you to build the project without it trying to download to the kit.
7. Build the project.
8. Connect your kit directly to a power outlet using a USB charger.
9. Use OTA to update your kit. You can use either the Windows or the Android app.
   1. Hint: Don't forget that every time you re-build the project you must re-sign the bin file and copy the resulting \*.bin.signed to the Windows OTA folder or Android device.
   2. Hint: If you need to find the Bluetooth Address of your device, use CySmart (either PC or Phone) to scan for it.
   3. Hint: If the OTA process fails on Windows, try resetting the kit and trying again. If that still fails, try using the Android version since it is more robust.
10. Once OTA upgrade is done, connect to the kit using CySmart and verify that the new firmware functionality is working.

(Advanced) Mesh Networks

### Introduction

### Project Creation

### Testing