

# Chapter 4C: BLE Low Power, Beacons, and OTA

Time: 2 ¾ Hours

This chapter covers more advanced topics such as low power modes, scan response packets, beacons, and OTA firmware upgrade.

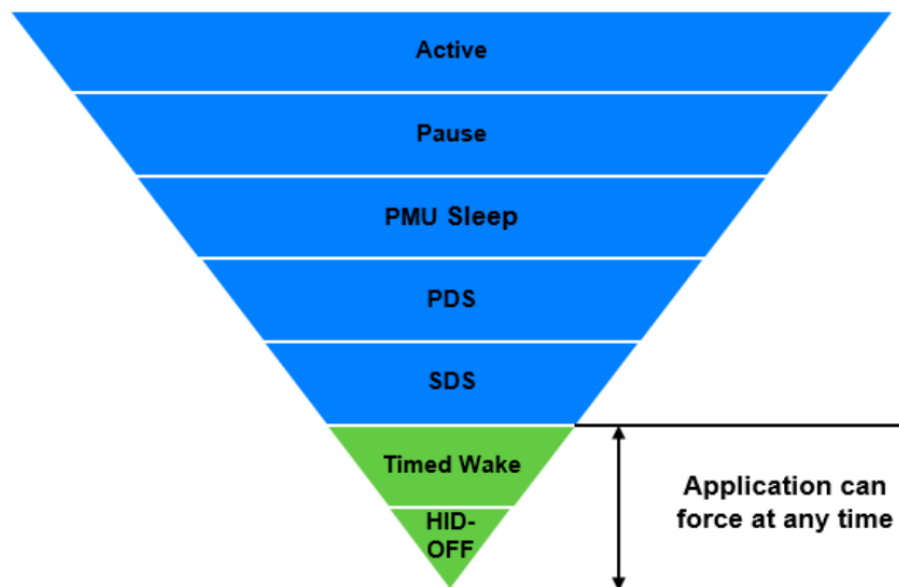
<b>4C.1</b>	<b>LOW POWER.....</b>	<b>2</b>
4C.1.1	POWER MODE OVERVIEW .....	2
4C.1.2	WICED LOW-POWER CODE .....	6
4C.1.3	PROGRAMMING IN LOW POWER MODE .....	10
<b>4C.2</b>	<b>ADVERTISING PACKETS .....</b>	<b>11</b>
4C.2.1	USING THE ADVERTISING PACKET TO GET CONNECTED .....	11
4C.2.2	BEACONS .....	12
<b>4C.3</b>	<b>SCAN RESPONSE PACKETS.....</b>	<b>15</b>
<b>4C.4</b>	<b>OTA (OVER THE AIR) UPGRADE .....</b>	<b>16</b>
4C.4.1	INTRODUCTION .....	16
4C.4.2	DESIGN AND ARCHITECTURE .....	17
4C.4.3	APPLICATIONS FOR LOADING NEW FIRMWARE.....	18
4C.4.4	OTA FIRMWARE .....	21
4C.4.5	SECURE OTA .....	24
<b>4C.5</b>	<b>EXERCISES.....</b>	<b>26</b>
EXERCISE 4C.1	BLE LOW POWER (PDS) .....	26
EXERCISE 4C.2	(ADVANCED) EDDYSTONE URL BEACON.....	28
EXERCISE 4C.3	(ADVANCED) USE MULTI-ADVERTISING ON A BEACON .....	29
EXERCISE 4C.4	(ADVANCED) ADVERTISE MANUFACTURING DATA AND USE SCAN RESPONSE FOR THE UUID.....	31
EXERCISE 4C.5	(ADVANCED) OTA FIRMWARE UPGRADE (NON-SECURE).....	32
EXERCISE 4C.6	(ADVANCED) OTA FIRMWARE UPGRADE (SECURE).....	34
EXERCISE 4C.7	(ADVANCED) BLE LOW POWER (SDS) .....	35

## 4C.1 Low Power

### 4C.1.1 Power Mode Overview

WICED Bluetooth devices support different power modes. However, it is important to note that not all the devices support every mode. The following figure shows the modes from highest to lowest power. The 20719 does not support Timed-Wake or HID-OFF modes.

The Power Management Unit (PMU) is responsible for managing sleep transitions as described below.



The following table provides additional details of each of the 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, an LHL interrupt, or a timer. 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 (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. The LPO source can either be an external 32 kHz crystal, internal 32 kHz crystal, or external 24 MHz crystal. The recommended option is an external 32 kHz crystal which is the default.
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 power domains on the chip (and the peripherals they supply) are managed by the PMU as described in the following table:

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.

## PMU Mode Transitions

The Power Management Unit (PMU) determines all power mode transitions except Timed-Wake and HID-OFF. The firmware can control whether PDS or SDS are allowed but it cannot not prevent Pause or PMU Sleep. It is up to the PMU to determine which sleep mode to enter depending on scheduled events. For example, if the firmware allows SDS, the PMU may decide to go into PDS instead of SDS because of an event scheduled for a short time in the future. In that case, going to SDS would not be beneficial because there is time (and power) required to shut down and re-initialize the system. The PMU can transition to Pause or PMU sleep any time.

In the firmware, you configure sleep by providing wake sources and by providing a sleep permit handler callback function that the PMU will call whenever it wants to go to PDS or SDS. In the callback function, you can: (1) disallow sleep completely; (2) allow PDS; or (3) allow SDS depending on your firmware's requirements.

## Additional SDS Mode Details

A global variable can be set in different sections of the firmware so that whenever the callback is called the appropriate sleep state will be allowed. In general, for a GAP Peripheral:

1. SDS can be used while advertising or with one connection with no pending activities.
  - a. SDS will be entered while advertising only if the timeout for that advertising type is set to 0. If not, the system will only enter PDS. For example, if the high duty cycle timeout is 30 seconds and the low duty cycle timeout is 0 then the device will use PDS during high duty cycle advertising (for 30 seconds) and will use SDS during low duty cycle advertising.
  - b. If SDS is entered while advertising, a connection request will have to be sent twice to initiate a connection. However, this may be worthwhile in applications where advertising continues for a long time.
2. SDS should not be used while pairing (i.e. encryption) is being negotiated.

For a GAP Central, SDS should not be used.

Upon wake from SDS, the device re-initializes from application start. The boot type can be determined as initial power up or reset (cold boot) or wake from SDS (fast boot). Differences required between cold and fast boot are handled by using the boot type.

SDS mode will not be entered if you have any active application threads. Therefore, if you have threads they must include an RTOS delay with `ALLOW_THREAD_TO_SLEEP` as the second parameter so that the thread will sleep.

Variables are not retained in SDS unless they are stored in Always On (AON) RAM. There are 256 bytes of AON RAM available to the application. This will be discussed in more detail in a minute.

For SDS, the connection interval should be set to 100ms or longer. Otherwise, SDS will not save significant power because the system will wake frequently.

Using RPA (i.e. the RPA refresh timeout is set to something other than WICED\_BT\_CFG\_DEFAULT\_RANDOM\_ADDRESS\_NEVER\_CHANGE) causes two issues in SDS:

1. During a connection, the timer for RPA will cause the firmware to wake frequently (about twice per second). This will cause increased power consumption.
2. During advertising, the address will change every time a fast boot occurs. Since this happens about twice per second if RPA is enabled, it is very difficult to connect because the address will change before you can connect to the device.

HCI UART and other BT pins such as host and dev wake are powered down in SDS.

In newer devices (e.g. 20819) SDS is replaced by ePDS. In ePDS, SRAM is retained and CPU registers are saved. Upon wakeup the CPU registers are restored. There are 2 main advantages to ePDS:

1. Upon wake from ePDS, the application resumes execution from where it left off after the system boots.
2. Resuming execution from ePDS can be done in as little as 3ms (currently it is ~7ms) as compared to the 10ms it takes to resume operation from SDS.

It can be difficult to achieve lower power using SDS than with PDS – much depends on how frequent the system must wakeup due to the overhead in re-initializing everything on every wakeup.

### Current Consumption

The exact current consumption depends on the firmware, but the approximate values to be expected are as follows:

Mode	Typical Current (μA)
Active	>50 (often much higher)
PDS	10 – 50
SDS	<10

## 4C.1.2 WICED Low-Power Code

To use low power modes in the firmware, there are 5 (or 6) things required if you are using PDS and 8 if you are using SDS:

1. Add sleep header files
2. Setup sleep configuration
3. Create a sleep permit handler callback function
4. Update connection parameters on a GATT connection
5. Create wakeup events (GPIO interrupts, timers, threads, etc.)
6. Update sleep permission variables at appropriate locations (usually SDS only)
7. Configure AON variables (SDS only)
8. Determine boot type and perform system initialization based on the value (SDS only)

### Sleep Header Files

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.

### Sleep Configuration

The function *wiced\_sleep\_configure* is used to enable 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. The structure is defined 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# to wake the device, 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;
```

In the firmware, you need to: (1) declare a global variable of type *wiced\_sleep\_config\_t*; (2) initialize all the elements of the structure just after stack initialization; and (3) call *wiced\_sleep\_configure*.

The elements in the structure are:

**sleep\_mode:** This can be either *WICED\_SLEEP\_MODE\_NO\_TRANSPORT* or *WICED\_SLEEP\_MODE\_TRANSPORT*. If you select the former, the device will enter sleep only if no host is connected (i.e. HCI UART CTS line not asserted). If you select the latter, the device will enter sleep only when an external HCI host is connected (i.e. HCI UART CTS line is asserted). If the device is being used stand-alone without an external HCI host, you should choose *WICED\_SLEEP\_MODE\_NO\_TRANSPORT*.

**host\_wake\_mode:** This can be either `WICED_SLEEP_WAKE_ACTIVE_LOW` or `WICED_SLEEP_WAKE_ACTIVE_HIGH` depending on the polarity of the interrupt to wake the host (if a host is connected). This only applies if `sleep_mode` is `WICED_SLEEP_MODE_TRANSPORT`. The Host Wake function is on a dedicated device pin, but it can be multiplexed into other IOs (this multiplexing feature is not currently supported in the API).

**device\_wake\_mode:** This can be either `WICED_SLEEP_WAKE_ACTIVE_LOW` or `WICED_SLEEP_WAKE_ACTIVE_HIGH` depending on the polarity of the interrupt for the host to wake the device (if a host is connected). This only applies if `sleep_mode` is `WICED_SLEEP_MODE_TRANSPORT`. The Device Wake function is on a dedicated device pin, but it can be multiplexed into other IOs (this multiplexing feature is not currently supported in the API). This pin is not available on the 20719 40-pin package but the `device_wake_source` pin can be used for this purpose.

**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:** This entry specifies which GPIO is used to wake the device from sleep. This only applies if `device_wake_source` includes GPIO.

**sleep\_permit\_handler:** This element requires you to provide a function pointer for callback function that will be called by the PMU to request sleep permission and when sleep is entered. This function will be described next.

## Sleep Permit Handler Callback Function

The sleep permit handler callback 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 `WICED_SLEEP_POLL_SLEEP_PERMISSION` callback, the return value must be one of the following based on the requirements of the firmware:

- `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. If an application will have activity within a short (e.g. 10ms) period of time, this mode is preferable since SDS requires time to shut down and re-start which may result in increased power.
- `WICED_SLEEP_ALLOWED_WITH_SHUTDOWN` – When this value is returned, the device can enter any of the low power modes including SDS.

If you are using PDS, you can always return that value (unless the firmware requires that the system stay awake sometimes). If you are using SDS, the value you return will depend on the current state of the system. You will typically set Global variables 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 *WICED\_SLEEP\_POLL\_TIME\_TO\_SLEEP* callback, you must return the maximum time that the system should be allowed to sleep. This is typically set to *WICED\_SLEEP\_MAX\_TIME\_TO\_SLEEP* but may also be returned as 0 if you don't want the system to go to sleep at that time. If you want to wake at a specific time, it is better to use a timer. You should not depend on this parameter as a wake source.

Remember that the PMU makes the final decision – it polls the firmware and each peripheral to see which type of sleep is allowed and how long sleep will be possible, and then decides which mode makes sense.

### Connection Parameter Update

The connection interval, latency and timeout are typically chosen by the Central. However, upon a connection, the Peripheral may request to update those values. The Central may or may not agree to the request. To reduce power consumption, it is recommended that the connection interval be set to 100ms or more once a GATT connection is established. In addition, the timeout should be set appropriately based on the connection interval, latency, and other application requirements.

An L2CAP function is used to request new values for the connection interval and timeout. This is done in the GATT connection callback when the connection state is "connected" (i.e. when the connection comes up). The function takes the *BD\_ADDR*, the min and max intervals in units of 1.25ms, latency in number of connection intervals that can be missed, and the timeout in units of 10ms.

To request new connection parameters, use the following function:

```
wiced_bt_l2cap_update_ble_conn_params (  
    wiced_bt_device_address_t rem_bdRa, // BD_ADDR of remote device  
    uint16_t min_int, // Min connection interval in units of 1.25ms  
    uint16_t max_int, // Max connection interval in units of 1.25ms  
    uint16_t latency, // Latency - number of connection intervals  
    uint16_t timeout); // Timeout in units of 10ms
```

### Wakeup Events

The firmware may need events that cause it to wake periodically or on specific events. For example, you may need to read a sensor value every few seconds or respond to user input such as a button press.

For periodic wakeup, you can either use a timer or you can use threads with delays that allow the thread to sleep (i.e. *ALLOW\_THREAD\_TO\_SLEEP*). The device will not enter sleep unless all threads and timers are in a sleep state.

As previously discussed, during sleep configuration the device wake source may be configured. If that source is set to GPIO then the specified GPIO will wakeup the system. However, you will not get a GPIO interrupt handler callback unless you register the callback function using

wiced\_hal\_gpio\_register\_pin\_for\_interrupt. Note that while the GPIO configuration (set using wiced\_hal\_gpio\_configure\_pin) is retained during SDS, the callback registration is NOT retained so you need to register the callback during both cold and fast boot.

### Sleep Permission Variable Update

One or more global variables may be used to keep track of the sleep type allowed. These variables are used in the sleep permit handler callback function to determine what value to return. For example, one possibility would be to:

1. Start out with sleep disabled
2. Allow either PDS or SDS during advertisement
  - a. This depends on how long advertising is expected to last and if it is acceptable to have to send 2 connection requests.
3. Allow only PDS once a GATT connect happens (to allow pairing/encryption)
  - a. Another option is to initiate the pairing request from the device immediately after connection so that the device can go to SDS sooner.
4. Switch between SDS and PDS once encryption of the link has completed (BTM\_ENCRYPTION\_STATUS\_EVT with WICED\_SUCCESS) depending on firmware requirements for activity.

There are other possible flows depending on the firmware's requirements.

If you are not using SDS, then you can always return *WICED\_SLEEP\_ALLOWED\_WITHOUT\_SHUTDOWN* from the sleep permit handler callback function for the poll sleep permission case unless your firmware needs to stay awake at certain times for other reasons. In that case, you won't need any sleep permission variables.

### AON Variable Configuration

During SDS, variables are not retained unless they are stored in Always On (AON) RAM or NVRAM. The AON RAM space is limited to 256 bytes for the user application.

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 when it went to sleep. You may also need to store the advertising type or 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;
```

## Determine Boot Type and Perform System Initialization

The function `wiced_sleep_get_boot_mode` is 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 (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.

For a fast boot, it is important to check to see if the device was already connected when it went into SDS. If it is already connected, then advertisements should not be started, and you may need to read values from NVRAM or restore GATT array values. Typically, NVRAM is only used if bonding information is being saved. If not, you can store GATT array values that need to be saved during SDS in AON RAM.

### 4C.1.3 Programming in Low Power Mode

When the device is asleep it is not listening for HCI commands which are needed to get the device in the correct mode for programming, so it may be difficult to program new firmware. To circumvent this issue, the kit has a "recovery" mode. To enter recovery mode:

- a. Press and hold the recovery button on the base board (left button)
- b. Press and release the reset button (center button)
- c. Release the recovery button
- d. Program from WICED Studio as normal

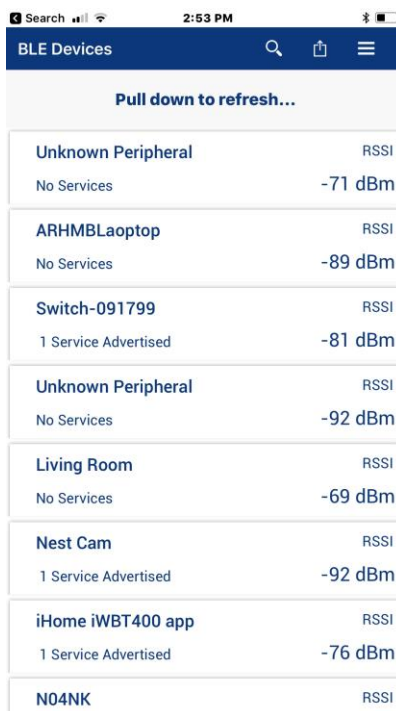
## 4C.2 Advertising Packets

There are two main uses of the advertising packet:

- Identifying a Peripheral with some recognizable data so that a Central knows if it wants to connect and talk to it.
- Sending out data (e.g. beacon data).

### 4C.2.1 Using the Advertising Packet to Get Connected

If you turn on the CySmart scanner, you will find that there are likely a bunch of unknown devices that are advertising around you. For instance, as I sit here right now I can see that there are quite a few Bluetooth LE devices around me that I have no idea what they are.



Pull down to refresh...	
Unknown Peripheral	RSSI
No Services	-71 dBm
ARHMBLaoptop	RSSI
No Services	-89 dBm
Switch-091799	RSSI
1 Service Advertised	-81 dBm
Unknown Peripheral	RSSI
No Services	-92 dBm
Living Room	RSSI
No Services	-69 dBm
Nest Cam	RSSI
1 Service Advertised	-92 dBm
iHome iWBT400 app	RSSI
1 Service Advertised	-76 dBm
N04NK	RSSI

When a Central wants to connect to a Peripheral, how does it know what Peripheral to talk to? There are two answers to that question.

First, it may advertise a service that the Central knows about (because it is defined by the Bluetooth SIG or is custom to your company). As we talked in the previous chapter you can customize the Advertising packet with information. In the picture above, you can see that some of the devices are advertising that they support 1 service. To do that they add a field of one of these types to the advertising packet along with the UUID of the Service:

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

The other scheme that is commonly used is to advertise "Manufacturer's Specific Data". This data has two parts:

- A two-byte manufacturer code as specified by the Bluetooth SIG (e.g. Cypress = 0x0131).
- The actual data which is typically a Product ID that is unique for each product that the company makes.

The way that this works is that you would write a Central application that has a table of known Peripheral Product IDs that it knows how to talk to. Then the Peripherals would advertise their Manufacturer code and Product ID in the Manufacturers Data Field. When a Central sees something that it knows how to talk to, it can make the connection.

#### 4C.2.2 Beacons

BLE Beacons are devices that are intended to send out data using advertising packets. Often, they will allow connections for configuration of the beacon but not for normal use.

Beacons can be used for lots of different purposes such as providing location (especially in large indoor spaces without GPS coverage (like Shinjuku station in Tokyo - <https://allabout-japan.com/en/article/2074>), or links to web sites with geographically relevant information (like a website with sale information for a store that you are currently standing in).

There are (of course) two popular types of beacon: iBeacon, which is defined by Apple, and Eddystone which is defined by Google. Each of these will be discussed in a minute.

#### Multi-Advertisement

Beacons can send out multiple advertisement packets to provide different types of data simultaneously. In many cases, a beacon may send out both iBeacon and Eddystone advertisement packets so that it will appear as both types of beacon. Each advertising instance can have unique parameters if desired.

To do multiple advertisements, you use the following three functions. The functions are all linked to each other using a parameter called "adv\_instance". This can be any integer from 1 to 16 that uniquely identifies each advertising instance.

#### wiced\_set\_multi\_advertisement\_params

This function sets advertisement parameters for each instance such as advertising type, advertising interval, advertising address (if a unique address is desired), etc. Its arguments are:

- |                            |   |
|----------------------------|---|
| • advertising_interval_min | Same as for standard advertising  |
| • advertising_interval_max | Same as for standard advertising  |
| • advertising_type         | Same as for standard advertising  |
| • own_address_type         | Type of address for this advertising instance – see wiced_bt_ble_address_type_t |
| • own_address              | Address if unique for this instance (otherwise use NULL)                        |
| • peer_address_type        | Type of address for the peer (only for directed adv)                            |

- `peer_address` Address if unique for this instance (otherwise use NULL)
- `advertising_channel_map` List of advertising channels to use (can use 37, 38, 39, or a combination)
- `advertising_filter_policy` Filter policy – see `wiced_bt_ble_advert_filter_policy_t`
- `adv_instance` A unique number used by `wiced_start_multi_advertisements`
- `transmit_power` Transmit power in dB - can use `MULTI_ADV_TX_POWER_MIN` or `MULTI_ADV_TX_POWER_MAX`

### **wiced\_set\_multi\_advertisement\_data**

This function sets advertisement data for multi-advertisement packets. It is analogous to `wiced_bt_ble_set_raw_advertisement_data` but the advertising data is sent as a flat array instead of a structure of advertising elements. Therefore, the procedure to setup the advertising packet will be a bit different as you will see in the exercises. Its arguments are:

- `p_data` Pointer to an advertising data array
- `data_len` Length of the advertising data array
- `adv_instance` Number of the instance specified above

### **wiced\_start\_multi\_advertisements**

This function starts advertisements using the parameters specified above. It is analogous to `wiced_bt_start_advertisements`. Its arguments are:

- `advertising_enable` `MULTI_ADVET_START` or `MULTI_ADVERT_STOP`
- `adv_instance:` Number of the instance specified above

## **iBeacon**

iBeacon is an Advertising Packet format defined by Apple. The iBeacon information is embedded in the Manufacturer section of the advertising packet. It simply contains:

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

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

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

## Eddystone

Eddystone is a Google protocol specification that defines a Bluetooth low energy (BLE) Advertising message format for proximity beacon messages. It describes several different frame types that may be used individually or in combinations to create beacons that can be used for a variety of applications.

There are currently four types of Eddystone Frames:

- UID – A unique beacon ID for use in mapping functions
- URL – An HTTP URL in a compressed format
- TLM – Telemetry information about the beacon such as battery voltage, device temperate, counts of packet broadcasts
- EID – Ephemeral ID packets which broadcast a randomly changing number

TLM frames do not show up as a separate beacon but rather are associated with other frames from the same device. For example, you may have a beacon that broadcasts UID frames, URL frames, and TLM frames. In a beacon scanner, that will appear as a UID&TLM beacon and a URL&TLM beacon.

The Advertising Packet has the following fields:

- Flags
  - Type: *BTM\_BLE\_ADVERT\_TYPE\_FLAG (0x01)*
  - Value: *BTM\_BLE\_GENERAL\_DISCOVERABLE\_FLAG | BTM\_BLE\_BREDR\_NOT\_SUPPORTED*
- 16-bit Eddystone Service UUID
  - Type: *BTM\_BLE\_ADVERT\_TYPE\_16SRV\_COMPLETE (0x03)*
  - Value: *0xFEAA*
- Service Data
  - Type: *BTM\_BLE\_ADVERT\_TYPE\_SERVICE\_DATA (0x16)*
  - Value: The Eddystone Service UUID (0xFEAA), the Eddystone frame type, then the actual data. Frame types are:
    - UID – 0x00
    - URL – 0x10
    - TLM – 0x20
    - EID – 0x30

The data required depends on the frame type. For example, a UID frame has: 1 byte of Tx power of the beacon at 0 m and a 16-byte beacon ID consisting of 10-byte namespace, and 6-byte instance.

In the project `snip.ble.eddystone` there is an example of this type of beacon. The example uses multi-advertising so that it will appear as multiple beacons at once on different advertising channels. One beacon sends UID frames, one sends URL frames, and one sends EID frames (there is an option to send TLM frames as well). This is useful if you want to send multiple types of data at once. It is also useful if you want a single device to operate as both an iBeacon and an Eddystone beacon at the same time.

If you are using Eddystone to send a URL, it is limited to 15 characters excluding a prefix (`http://`, `https://`, `http://www.`, or `https://www.`) and a suffix (`.com`, `.com/`, `.org`, `.org/`, `.edu`, `.edu/`, etc.). If you

need to create a shorter URL for a site, use a web browser to go to <https://goo.gl>. That site will allow you to create a short URL.

You can find the detailed spec at <https://github.com/google/eddystone>.

### 4C.3 Scan Response Packets

Once a Central finds a Peripheral and wants to know more about it, the Central can look for scan response data. For a peripheral, the scan response packet looks just like an advertising packet except that the Flags field is not required. Like the advertising packet, the scan response packet is limited to 31 bytes.

In WICED you setup the scan response packet array of advertising elements the same way as you do for the advertising packet. You then call the function `wiced_bt_ble_set_raw_scan_response_data` to pass that information to the Stack. That function takes the same arguments as `wiced_bt_ble_set_raw_advertisement_data` – that is, the number of advertising elements in the array, and a pointer to the array.

When you start advertising with an advertising type other than `_NONCONN_` then the Central will be able to read your scan response data. For example, `_DISCOVERABLE_` will allow the scan response to be read but will not allow connections and `_UNDIRECTED_` will allow the scan response to be read and will allow connections.

## 4C.4 OTA (Over the Air) Upgrade

### 4C.4.1 Introduction

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. This ensures that unknown firmware is not uploaded to your device.

## 4C.4.2 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.

### 4C.4.3 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 but it will only work on Windows 10 or later since BLE is not natively supported in earlier versions.

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\Android\LeOTAApp\app\build\outputs\apk
```

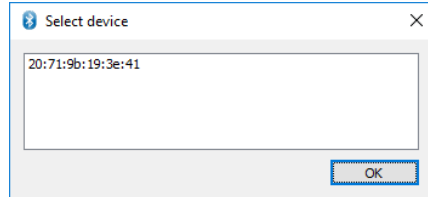
The default <SDK Install Dir> is in your Documents folder.

#### Windows

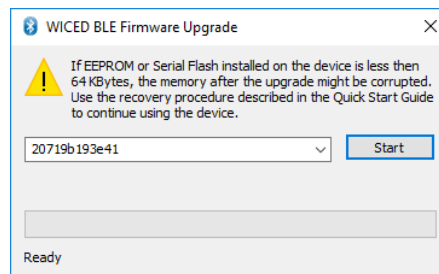
To use the Windows peer application, you must first copy the \*.bin file from the build directory of the WICED application into the same folder as the Windows peer application. Then run the application with the \*.bin file provided as an argument. For example, from a command or PowerShell window:

```
.\WsOtaUpgrade.exe ex05_ota- CYW900719Q40EVB_01-rom-ram-Wiced-release.ota.bin
```

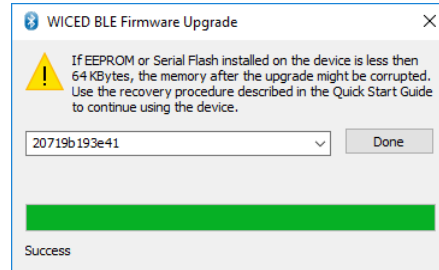
You will get a window that looks like the following. Select the device you want to update and click "OK".



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



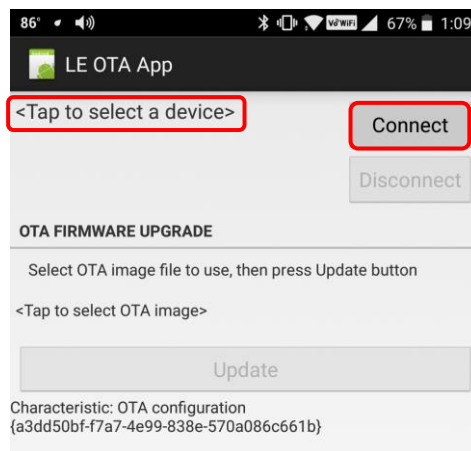
If the update worked, the window will show "Success" at the bottom. Click "Done" to close the window.



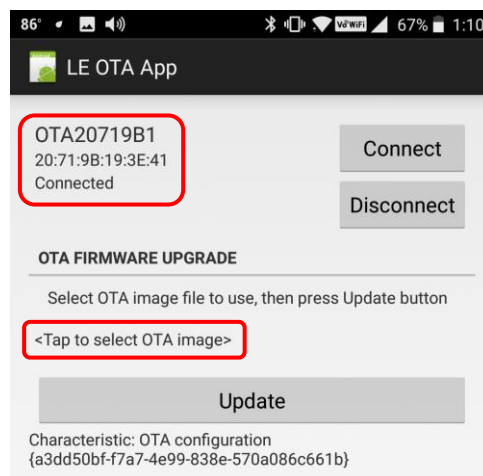
## Android

To use the Android app:

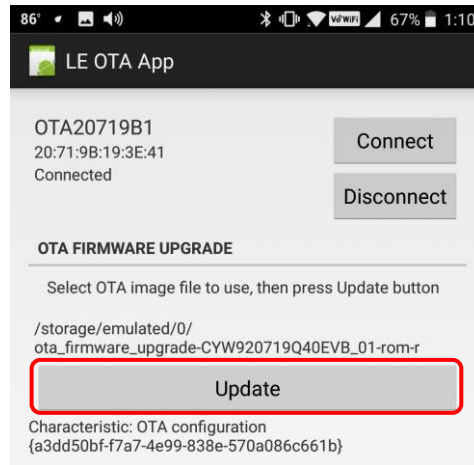
1. Install the app-debug.apk file on your Android device if you have not already done so.
2. Copy the \*.bin file from the build directory onto the device in a location where you can find it.
3. Run the app called *LE OTA App*. The startup screen will look like this:



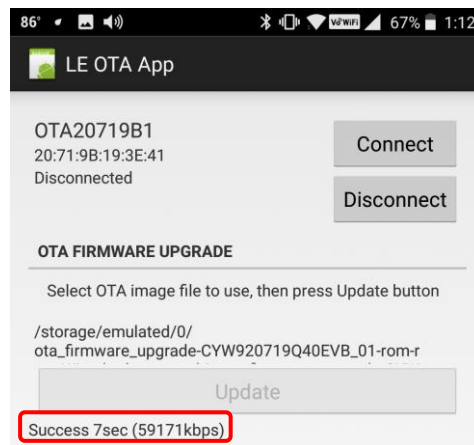
4. Tap where it says <Tap to select a device> and choose your device from the list.
5. Tap on the "Connect" button. Once connected, the screen will look like this:



6. 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:



7. 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.



#### 4C.4.4 OTA Firmware

In the firmware, OTA requires the following:

##### Header Files

Include the following header files at the top of your main .c file and the \_db.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,
    LEGATTDDB_CHAR_PROP_WRITE | LEGATTDDB_CHAR_PROP_NOTIFY | LEGATTDDB_CHAR_PROP_INDICATE,
    LEGATTDDB_PERM_VARIABLE_LENGTH | LEGATTDDB_PERM_WRITE_REQ /*| LEGATTDDB_PERM_AUTH_WRITABLE*/
    ),

    CHAR_DESCRIPTOR_UUID16_WRITABLE(HANDLE_OTA_FW_UPGRADE_CLIENT_CONFIGURATION_DESCRIPTOR,
    UUID_DESCRIPTOR_CLIENT_CHARACTERISTIC_CONFIGURATION, LEGATTDDB_PERM_READABLE |
    LEGATTDDB_PERM_WRITE_REQ /*| LEGATTDDB_PERM_AUTH_WRITABLE */),

    CHARACTERISTIC_UUID128_WRITABLE(HANDLE_OTA_FW_UPGRADE_CHARACTERISTIC_DATA,
    HANDLE_OTA_FW_UPGRADE_DATA, UUID_OTA_FW_UPGRADE_CHARACTERISTIC_DATA,
    LEGATTDDB_CHAR_PROP_WRITE, LEGATTDDB_PERM_VARIABLE_LENGTH | LEGATTDDB_PERM_WRITE_REQ /*|
    LEGATTDDB_PERM_AUTH_WRITABLE */),
```

Note that the LEGATTDDB\_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 Connect Callback

In the GATT connection status callback function, it is necessary to pass the connection status information to the OTA library by calling the following:

```
wiced_ota_fw_upgrade_connection_status_event(p_conn_status);
```

This should be called on both a connection and dis-connection.

## 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 = ex05_ota_indication_cfm_handler(p_data->conn_id, p_data->data.handle);
    break;
```

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

```
wiced_bt_gatt_status_t ex05_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 ex05_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 ex05_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. (Note: it must be at least the MTU size plus 12).

## 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.

### 4C.4.5 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:
  - a. ecdsa256\_key.pri.bin
  - b. ecdsa256\_key.pub.bin
  - c. ecdsa256\_key\_plus.pub.bin
  - d. 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.

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

## 4C.5 Exercises

### Exercise 4C.1 BLE Low Power (PDS)

#### Introduction

In this exercise, you will copy over and analyze a project that implements low power using PDS. The project is based on the exercise that stores BLE bonding information in NVRAM (ch04b/ex04\_ble\_bond).

#### Project Creation

1. Go to the folder in the class files at WBT101\_Files/Patches and follow the instructions in README.txt. This is necessary due to a bug in the WICED SDK in WICED Studio 6.2.1.
2. Copy the folder from the class files at WBT101\_Files/Templates/ch04c/ex01\_low\_power into the ch04c folder for your workspace.
  - a. Hint: Change the name from *key\_lp* to *use your initials instead of "key"* in the wiced\_bt\_cfg.c file and the ex01\_low\_power.c file.
  - b. Hint: If your initials are more than 3 letters, make sure you also update the maxlen and curlen in the GATT database lookup table (gatt\_db\_lookup\_table). Make sure your advertising packet total length will not be more than 31 bytes.
3. Create a Make Target.

#### Testing

1. Download the project onto the kit.
  - a. Hint: If your device is in low power mode you may have to put it into Recovery mode first to program it. To enter Recovery mode:
    - i. Press and hold the recovery button on the base board (left button)
    - ii. Press and release the reset button (center button)
    - iii. Release the recovery button
2. Open a UART terminal window.
3. Open the PC CySmart app. Start scanning and then stop once your device appears.
4. Connect to the device. You will see a notification asking to confirm the connection parameters. Select 'Yes'.
5. Discover all attributes in the GATT database, and Pair with the device.
6. Enable all notifications.
7. Press and release the button repeatedly so that notifications are sent.
8. Disconnect and clear the Device Information from the Device List in CySmart.



## Exercise 4C.2 (Advanced) Eddystone URL Beacon

### Introduction

In this exercise, you will create an Eddystone beacon that will advertise the URL for <https://www.cypress.com>. From your phone you will be able to scan for the beacon (using a beacon scanner app) and then directly connect to the advertised website.

### Project Creation

1. Copy the folder from the class files at WBT101\_Files/Templates/ch04c/ex02\_ble\_eddy into the ch04c folder for your workspace.
  - a. Hint: The template is just the solution from exercise ch04a/ex03\_adv so if you prefer, you can instead copy your answer to that exercise and rename things as necessary.
  - b. Hint: Change the name from *key\_eddy* to use your initials instead of "key" in the *wiced\_bt\_cfg.c* file.
2. Remove the button interrupt configuration and interrupt callback.
3. Change the timeout for low-duty cycle advertising to 0 so that advertising will never timeout.
4. Update the advertising data packet to send out an Eddystone frame with the URL of <https://goo.gl/dfhs4w>. This URL was created on <https://goo.gl> and points to the materials for this class on GitHub. The packet needs:
  - a. Flags Field
  - b. 16-bit Eddystone Service UUID Field
  - c. Service Data Field with:
    - i. Eddystone Service UUID
    - ii. Eddystone Frame Data
5. Hint: The following site details the Eddystone URL Frame Data:  
<https://github.com/google/eddytone/tree/master/eddytone-url>
  - a. Hint: For the TX Power value in the URL Field, use a hardcoded value of 0xF0.
  - b. Hint: The Eddystone Service UUID must be sent little endian (in both places)

### Testing

1. Create a make target and program the project to your kit.
2. On your phone, install a beacon scanner app. For Android, there is an app called "Beacon Scanner" written by Nicolas Bridoux that works well (although it does not recognize EID frames). Similar apps exist for iOS.
3. Look for your Bluetooth Device address in the UART terminal window to find the correct beacon in the list.
  - a. Note: In iOS the Bluetooth Device address can't be identified (Apple doesn't allow it) so you will not be able to identify your specific device.
4. Open the URL for [www.cypress.com](https://www.cypress.com) from the beacon app.
5. If you don't see your device in the beacon app it most likely means your packet isn't correct. Using the CySmart PC application, scan for your device and look at its advertising packet to determine what's wrong.

## Exercise 4C.3 (Advanced) Use Multi-Advertising on a Beacon

### Introduction

In this exercise, you will add UID and TLM frames to the Eddystone beacon from the previous exercise.

### Project Creation

1. Copy exercise ex02\_ble\_eddy to ex03\_ble\_eddy\_multi. Rename the project folder and project files as necessary.
2. Rename the `<appname>_set_advertisement_data` function to `<appname>_set_advertisement_data_url`.
3. Update the new function to use `wiced_set_multi_advertisement_params` and `wiced_set_multi_advertisement_data`.
  - a. Hint: Instead of building up the advertisement array using `wiced_bt_ble_advert_elem_t` elements, you will need to create a single flat array of `uint8_t`.
  - b. Hint: You will need the same three fields: Flags, Complete 16-bit service UUID, and Service Data.
  - c. Hint: Each field needs the length (excluding the length byte), type, and data.
  - d. Hint: Use a different channel for each advertising instance.
4. Add a second function called `<appname>_set_advertisement_data_uid`.
  - a. Change the data to send an Eddystone UID frame.
    - i. Hint: see the Eddystone documentation for details at:  
<https://github.com/google/eddystone/tree/master/eddystone-uid>
    - ii. Hint: Use a hard-coded value of 0xF0 for the Tx power.
    - iii. Hint: Use any random values for the namespace and instance values.
5. Add a third function called `<appname>_set_advertisement_data_tlm`.
  - a. Change the data to send an Eddystone TLM frame.
    - i. Hint: see the Eddystone documentation for details at:  
<https://github.com/google/eddystone/tree/master/eddystone-tlm>
    - ii. Hint: Use Unencrypted TLM frames.
    - iii. Hint: Use hardcoded values for all parameters except time since power-on reboot. Use a global variable for time since power-on reboot.
6. Add a periodic 1 second timer.
7. In the timer callback function, increment the time since power-on reboot variable and call the `<appname>_set_advertisement_data_tlm` function. This will update the TLM data every second.
  - a. Hint: `uint32_t` values are stored little endian on the chip but the values in the TLM packet are big endian.
8. In application initialization, initialize and start the timer, call your functions to setup advertisement data for URL and UID, and then start all three multi-advertisement instances (don't forget to remove the call that starts standard advertising).

## Testing

1. Create a make target and program the project to your kit.
2. On your phone, open the beacon scanner app.
3. Look for your Bluetooth Device address in the UART terminal window to find the correct beacons in the list.
4. You should see two beacons with your address: one that shows URL and TLM information and the other that shows UID and TLM information. Notice how the TLM Uptime value increases every second.
5. It should look something like this:



## Exercise 4C.4 (Advanced) Advertise Manufacturing Data and use Scan Response for the UUID

### Introduction

In this project, you will take a project that advertises the manufacturer ID for Cypress and a value and you will add a scan response packet that sends the service UUID.

### Project Creation

1. Copy the folder from the class files at WBT101\_Files/Templates/ch04c/ex04\_ble\_manu into the ch04c folder for your workspace.
  - a. Hint: The template is just the solution from exercise ch04b/ex02\_ntfy so if you prefer, you can instead copy your answer to that exercise and rename things as necessary.
  - b. Hint: Change the name from *key\_manu* to *use your initials instead of "key"* in the `wiced_bt_cfg.c` file and the `ex04_ble_manu.c` file.
  - c. Hint: If your initials are more than 3 letters, make sure you also update the `maxlen` and `curlen` in the GATT database lookup table (`gatt_db_lookup_table`). Make sure your advertising packet total length will not be more than 31 bytes.
2. Copy the `<appname>_set_advertisement_data` function to a new function called `<appname>_set_scan_response_data`.
  - a. Hint: Don't forget to add function prototype if necessary.
3. Update the scan response packet to send the 128-bit service UUID.
4. At the end of your new function, call the function to set the raw scan response data instead of the raw advertisement data.
5. Call your new function from application init.
6. Update the advertisement data so that instead of the UUID, you send manufacturer info with the Cypress manufacturer ID (0x0131) and a product ID of 0x2A. Remember that the manufacture ID must be send little endian, so the manufacturer data will be 0x31012A.

### Testing

1. Create a make target and program the project to your kit.
2. Open the PC version of CySmart and scan for your device. Stop scanning once you see it.
3. Click on your device and examine the Advertisement data packet to verify it is as expected.

Click the tab above the Advertisement data that says Scan response data and verify it is as expected.

## Exercise 4C.5 (Advanced) OTA Firmware Upgrade (Non-Secure)

### Introduction

In this exercise, you will modify a project that controls one LED over BLE 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 the folder from the class files at WBT101\_Files/Templates/ch04c/ex05\_ota into the ch04c folder for your workspace.
  - a. Hint: The template is based on ch04a/ex01\_key\_LED but it adds the custom pin configuration so that the GPIO that is connected to LED\_1 can drive the LED.
  - b. Hint: Change the name from *key\_ota* to *use your initials instead of "key"* in the *wiced\_bt\_cfg.c* file and the *ex05\_ota.c* file.
  - c. Hint: If your initials are more than 3 letters, make sure you also update the *maxlen* and *curlen* in the GATT database lookup table (*gatt\_db\_lookup\_table*). Make sure your advertising packet total length will not be more than 31 bytes.
2. Create a Make Target for the project.
3. Review the OTA Firmware section in this chapter and update the files as necessary:
  - a. Add additional header files to the C files
  - b. Add the OTA library to *makefile.mk*
  - c. Add the OTA service to *ex05\_ota\_db.c*
  - d. Update the GATT event handler functions
  - e. Increase the large buffer pool size to 1024
  - f. 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. Only LED\_2 should turn on for any value other than 0.
  - a. 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

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.
  - a. Hint: The Windows application only works on Windows 10 or later since earlier versions do not support BLE.
  - b. 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.
  - c. Hint: If you need to find the Bluetooth Address of your device, use CySmart (either a PC or Android Phone) to scan for it.
  - d. 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, reset the kit then connect using CySmart and verify that the new firmware functionality is working.

## Exercise 4C.6 (Advanced) OTA Firmware Upgrade (Secure)

### Introduction

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

### Project Creation

1. Copy the folder from the class files at WBT101\_Files/Templates/ch04c/ex06\_ota\_secure into the ch04c folder for your workspace.
  - a. Hint: The template is just the solution from exercise ch04a/ex05\_ota so if you prefer, you can instead copy your answer to that exercise and rename things as necessary.
  - b. Hint: Change the name from *key\_otas* to *use your initials instead of "key"* in the *wiced\_bt\_cfg.c* file.
2. Create a Make Target for the project.
3. Review the Secure OTA Firmware section in this chapter to generate the keys 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.
  - a. 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.
  - b. Hint: If you need to find the Bluetooth Address of your device, use CySmart (either PC or Phone) to scan for it.
  - c. 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.

## Exercise 4C.7 (Advanced) BLE Low Power (SDS)

### Introduction

In this exercise, you will copy a template project that implements low power using SDS.

### Project Creation

1. Copy the project from the class files at WBT101\_Files/Templates/ch04c/ex07\_low\_power\_sds.
2. Create a Make Target.
3. Change "key" to your initials in ex07\_low\_power\_sds.c and wiced\_app\_cfg.c so that you will be able to find your device.

### Firmware Changes

To enable SDS, the following changes were made:

1. A global variable was added and used to set desired sleep mode in different phases of the application.
2. The sleep callback function was updated to return different values depending on the desired sleep mode.
3. Variables needed during a fast boot were stored in AON RAM.
4. Cold Boot vs. Fast Boot changes were made.
5. Low Duty Cycle Advertising timeout was set to 0 (never timeout).
6. RPA was disabled.

### Testing

1. Open a UART terminal window.
2. Download the project onto the kit.
3. Observe the UART while the device is in high duty cycle advertising and when it is in low duty cycle advertising.
4. Open the PC CySmart app. Reset the kit to re-enter high duty cycle advertising. Start scanning and then stop once your device appears.
  - a. Hint: You won't be able to connect from CySmart when the device is in low duty advertising because it will not send the connection request twice.
5. Connect to the device. You will see a notification asking to confirm the connection parameters. Select 'Yes'.
6. Observe the UART when the connection happens.
7. Discover all attributes in the GATT database, and Pair with the device.
8. Observe the UART when pairing completes.
9. Enable all notifications and press the button.
10. Observe the UART while notifications are being sent.
11. Disconnect and clear the Device Information from the Device List in CySmart.

## Questions

1. What variable is used to control sleep permissions? What are its states?
2. What is printed to the UART when:
  - a. A sleep request is denied: \_\_\_\_\_
  - b. A sleep request is allowed without shutdown: \_\_\_\_\_
  - c. A sleep request is allowed with shutdown: \_\_\_\_\_
  - d. A fast boot occurs: \_\_\_\_\_
3. When is sleep not allowed? When is sleep allowed but without shutdown (PDS)? When is sleep allowed with shutdown (SDS)?
4. When does SDS happen (Hint: you can determine that SDS occurred when you see a fast boot)?
5. What is done differently for a cold boot vs. a fast boot? Why?
6. What is AON RAM? What values is it used for and why?