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

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

4C.1 Low Power

4C.1.1 Power Mode Overview

You will get a chance to experiment with Low Power in Exercise 4C.1 .

WICED Bluetooth devices support different power modes. However, it is important to note that not all the devices support every mode. The following figures show the modes from highest to lowest power for the 20819 and 20719 families.

Further details can be found in application notes CYW20819 Low-Power Guidelines (AN225270) and CYW20706 and CYW20719 Low-Power Modes.

20819	20719
Active	Active
Idle	Pause
Sleep	PMU Sleep
PDS	PDS
ePDS	SDS
Timed Wake	
HID-OFF	
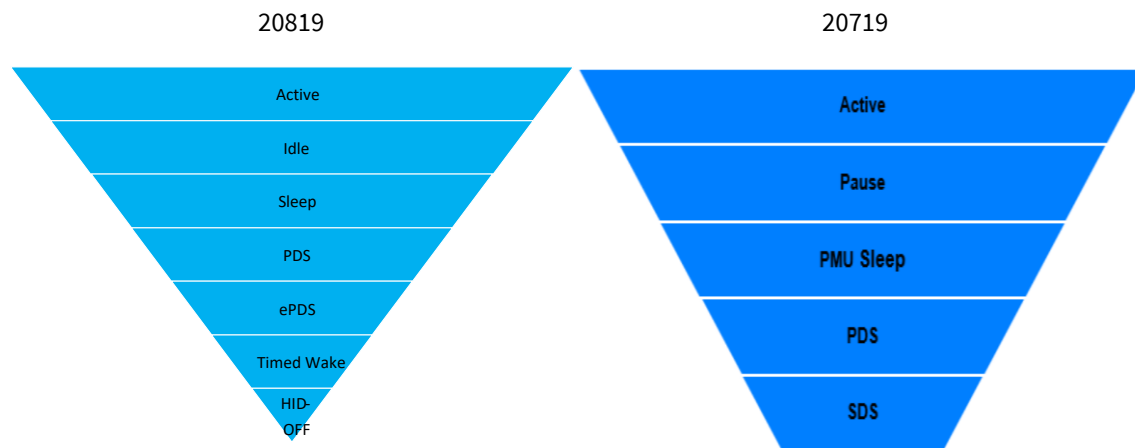
The following table provides additional details of each of the WICED Power Modes:

Mode	Family	Description
Active	Both	Active mode is the normal operating mode in which all peripherals are available, and the CPU is active.
Idle Pause	20819 20719	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. Idle/Pause mode is chosen when the other lower power modes are not possible. Any configured interrupt can wake the device.
Sleep	Both	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. Wake can be from various sources such as GPIO, timer, or BT activity.
Power Down Sleep (PDS)	20719	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)	20719	Everything is turned OFF except LHL GPIOs, RTC, and LPO. The device can come out of this mode either due to Bluetooth activity, a GPIO 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

Mode	Family	Description
		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 a 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.
Enhanced Power Down Sleep (ePDS)	20819	The entire device is OFF except for RTC, LPO, and a comparator to wake the device. The entire SRAM memory is retained. The device enters this state through WFI but exits through a reset vector. Before entering this mode, the PMU stores the CPU state and some register values (including the program counter) in SRAM, which are restored after wakeup so that the CPU starts executing from where it stopped. It can wake up using a GPIO, a timer, or BT activity.
Timed-Wake	20819	LHL GPIOs, RTC, and LPO are the only active blocks. The only wakeup sources are a timer or a GPIO interrupt. SRAM is not retained. The device starts executing from reset; therefore, the wakeup time is the same as that of power on reset.
HID-OFF	20819	This mode is the same as Timed-Wake, but in HID-OFF mode even the LPO and RTC are turned OFF. The only wakeup source is a GPIO interrupt. SRAM is not retained. The device starts executing from reset; therefore, the wakeup time is the same as that of power on reset.

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	SRAM, Patch RAM PWM (20719)	PWM: If ACLK is used, then the hardware block can operate until the PMU enters Sleep. If the LHL clock is used, then the hardware block can operate until the PMU enters SDS.
2	VDDCP (20819)	PWM (20819)	PWM is switched OFF in ePDS mode but can work in higher power modes.
3	VDDCG	I2C, SPI, PUART, WDT, ARM GPIO, Dual Input 32-bit Timer, flash, ROM	The hardware blocks can operate until the PMU enters Sleep (20719) or ePDS (20819)
4	VBAT/LHL	LHL GPIO, Analog PMU, RTC	The hardware blocks can operate until the PMU enters SDS (20179) or HID-Off (20819)
5	VBAT/LHL	Aux ADC	The Aux ADC can operate until the PMU enters Sleep (20719) or ePDS (29819).

PMU Mode Transitions

The Bluetooth device contains a Power Management Unit (PMU) which is responsible for all power mode transitions. The firmware can control whether PDS/SDS (201719) or ePDS/Timed Wake/HID Off (20819) are allowed but it cannot not prevent Pause/Idle or 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 Idle/Pause or 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/SDS (20719) or ePDS/Timed Wake/HID Off (20819). In the callback function, you can disallow or allow the requested low power mode transition. For the 20719, you can choose to only allow PDS or allow both PDS and SDS depending on your firmware's requirements.

In ePDS/PDS mode, the device will wake every time a BLE event must be serviced. For example, during advertising, the device will wake every time a new advertising packet needs to be sent. Once a connection has been established, the device will wake for each connection interval.

In SDS mode, the micro-Bluetooth Core Scheduler (μBCS) is used to service BLE events such as sending advertising packets and keeping the connection alive, so it does not need to wake for these events. During a connection, the device will typically only wake when data needs to be sent over the link, such as due to a timer that sends data periodically or due to a GPIO interrupt indicating a change.

Using RPA (i.e. the RPA refresh timeout is set to something other than WICED_BT_CFG_DEFAULT_RANDOM_ADDRESS_NEVER_CHANGE) causes the firmware to wake frequently (about twice per second). This will cause increased power consumption. Therefore, RPA should not be used in power critical applications.

Additional SDS Mode Details (Advanced)(20719 family only)

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 an additional issue in SDS. 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.

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.

Wake Times

Approximate wake times for various modes are as follows:

Mode	Typical Wake Time (ms)
Active	N/A
Idle/Pause	ISR service time – almost instantaneous
Sleep	0.5 – 2
PDS	
SDS	10
ePDS	2
Timed Wake	Same as POR – a few hundred ms
HID Off	Same as POR – a few hundred ms

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 (very FW dependent - often much higher)
Idle/Pause	
PDS	<20
SDS	<10
ePDS	~10 with no BLE
Timed Wake	~3
HID Off	~3

4C.1.2 WICED Low-Power Code

To use low power modes in the firmware, the following steps are required:

1. Add sleep header files
2. Setup sleep configuration
3. Create a sleep permit handler callback function
4. Create a post-sleep callback function

5. Update connection parameters on a GATT connection
6. Create wakeup events (GPIO interrupts, timers, threads, etc.)
7. Update Advertising Settings
8. Update BT Address Setting

The following additional steps are required only when SDS is used on the 20719 family:

9. Update sleep permission variables at appropriate locations
10. Configure AON variables
11. Determine boot type and perform system initialization based on the boot type

Sleep Header Files

The header file *wiced_sleep.h* contains the API related to low power operation. 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:

```
/** This structure defines the sleep configuration parameters to be passed to
 *   wiced_sleep_configure API
 */
typedef struct
{
    wiced_sleep_mode_type_t          sleep_mode;    /**< Defines whether to sleep with or without transport */
    wiced_sleep_wake_type_t          host_wake_mode; /**< Defines the active level for host wake signal */
    wiced_sleep_wake_type_t          device_wake_mode;/**< Defines the active level for device wake signal
                                         If device wake signal is not present on the device then
                                         GPIO defined in device_wake_gpio_num is used */
    uint8_t                          device_wake_source; /**< Device wake source(s). See 'wake sources' defines
                                         for more details. GPIO is mandatory if WICED_SLEEP_MODE_TRANSPORT
                                         is used as sleep_mode*/
    uint32_t                         device_wake_gpio_num;   /**< GPIO# for device 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_post_sleep_callback post_sleep_cback_handler;/**< Callback to application on wake from sleep */
} 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*. Note that in ModusToolbox 2.x hosted operation is not supported and so this element is always *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. An alternate way to configure the GPIO for wake up is to register the GPIO for interrupt in the user application.

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.

post_sleep_cback_handler: This element requires you to provide a function that is called when the device wakes from sleep in case the application needs to take any action at that time. It is passed one Boolean argument called "restore_configration" that indicates whether the application needs to restore configurations or not based on the wakeup event.

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 device and the requirements of the firmware:

- *WICED_SLEEP_NOT_ALLOWED* – The application can return this value if it does not want the device to enter PDS or lower sleep modes.
- *WICED_SLEEP_ALLOWED_WITHOUT_SHUTDOWN* -The application can return this value if low power is allowed. For the 20819, this value should be returned if the firmware wishes to allow PDS/ePDS but not Timed-Wake or HID-Off when sleep is allowed. For the 20719, this value should be returned if the firmware wishes to allow PDS but not SDS.
- *WICED_SLEEP_ALLOWED_WITH_SHUTDOWN* –When this value is returned, the device can enter any of the low power modes including Timed-Wake or HID-Off (for 20819) or SDS (for 20719).

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 typically should not depend on this parameter as a wake source.

If you want to enter Timed-Wake mode (i.e., HID-Off with wakeup at a predetermined time), this parameter can be used to specify the time the device should remain HID-Off mode. For example, if you pass 10000000, the device will enter HID-Off mode for 10 seconds. If you specify WICED_SLEEP_MAX_TIME_TO_SLEEP, the device will enter HID-Off mode and will not wake until a GPIO interrupt occurs.

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.

Post-Sleep Callback Function

The post-sleep callback function takes one argument of type wiced_bool_t, which specifies whether the application needs to re-initialize any peripherals. If the parameter value is TRUE, the peripherals need to be re-initialized and configured; otherwise not. Only the PUART is initialized by the stack by default, so you don't need to re-initialize it. The return type is void.

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
```

Advertising Settings

In ePDS/PDS mode, the device will wake every time a new advertising packet needs to be sent. Therefore, slower advertising will result in lower power. There is also a timer that runs to schedule advertising timeouts, so power will be lower during advertising if the timeout is set to 0 which doesn't require a timer.

BT Address Settings

There is a timer that is used to track resolvable private addresses (RPA). This will cause the firmware to wake frequently during advertising. Therefore, RPA should not be used in power critical applications.

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 interrupt will wake up 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.

Wakeup Reason

After waking up from HID-Off or Timed-Wake mode, the device starts from the reset vector. The wakeup reason can be determined by using the following API function call:

```
wiced_sleep_wake_reason_t wiced_sleep_hid_off_wake_reason(void);
```

This function will return the wakeup reason to the application as Power-On Reset (POR), wakeup from Timed-Wake mode, or GPIO wakeup (i.e. HID-Off).

Sleep Permission Variable Update (Advanced)(20719 with SDS only)

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 (Advanced)(20719 with SDS only)

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;
```

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 (red button on the left)
      b. Press and release the reset button (blue button in the center)
      c. Release the recovery button
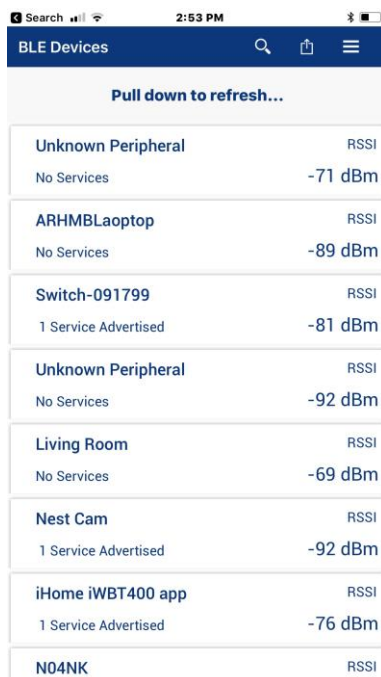   d. Program as normal

4C.2   Advertising Packets

There are three 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).
- Implementing a Bluetooth Mesh network (which uses advertising to send and receive 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.

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

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

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

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

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

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

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_advertisment_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_advertisment_data <u>but the advertising data is sent as a flat array instead of a structure of advertising elements</u>. 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

The Bluetooth SDK includes a library called beacon_lib that includes a number of functions to help format packets correctly before calling `wiced_set_multi_advertisement_data()`. To create an Eddystone UID (described later) packet, for example, you would use the `wiced_bt_eddystone_set_data_for_url()` function from this library.

To get access to the library API from your application, add this to your source code:

```
#include "wiced_bt_beacon.h"
```

You must also include the library by adding the following to the makefile in the "Components (middleware libraries)" section (replace the existing "COMPONENTS+=" line):

```
#
# Components (middleware libraries)
#
COMPONENTS += beacon_lib
```

Finally, add the following line in the "paths to shared_libs targets section" (add after the existing line that starts SEARCH_LIBS_AND_INCLUDES):

```
SEARCH_LIBS_AND_INCLUDES += $(CY_SHARED_PATH)/dev-kit/libraries/btsdk-ble
```

wiced_start_multi_advertisements

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

- advertising_enable   MULTI_ADVERT_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

You will use this in Exercise 4C.2 and Exercise 4C.3 .

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 temperature, 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 Eddystone 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 application BLE-20819EVB02.beacon there is an example of a beacon that acts as both an Eddystone and iBeacon at the same time. The example uses 5 advertising instances - one for each of the four Eddystone frame types and one for iBeacon.

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, there are sites that will allow you to create a short URL alias such as www.tinyurl.com.

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

Note, there is a middleware library of Eddystone helper functions that can be accessed by including "beacon_lib" in your application. Once the middleware library is added, you also need to include "wiced_bt_beacon.h" in your application.

4C.3 Scan Response Packets

You will use this in Exercise 4C.4

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 set up 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 allows an external device to use the Bluetooth link to transfer and install a newer firmware version to devices that support OTA. This section describes the functionality of the Firmware Upgrade library used in various sample applications.

For 20819 devices, there may not enough on-chip flash to fit two copies of the application firmware that are required during OTA, so it is sometimes required to have external flash. In that case, the OTA process uses the external flash during OTA.

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

The OTA update process depends on how much flash the system has and how much is on-board. There are three possibilities:

1. If there is enough on-board flash to hold 2 complete application images, the flash memory is organized into two partitions (DS1 and DS2) for failsafe upgrade capability. During startup the boot code of the chip checks DS1 and if a valid image is found, it starts executing application code from there. If DS1 does not contain a valid image, the boot code checks DS2 and starts application code execution from there if a valid image is found. If neither partition is valid, the boot code enters download mode and waits for the code to be downloaded over the HCI UART.

   The addresses of the DS1 and DS2 partitions are programmed in a file with extension "btp" in the platforms directory of the SDK. For example:

   wiced_sdk/dev-kit/baselib/20819A1/platforms/208XX_OCF.btp:

ConfigDSLocation =	0x501400
ConfigDS2Location =	0x520A00

   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.

   During an OTA upgrade the device performing the procedure (Downloader) pushes chunks of the new image to the device being upgraded. 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.

2. If there is not enough on-board flash to hold 2 complete application images, then a single on-board flash region (DS1) always contains the executable firmware. In this case, the update image is downloaded to a storage area on external flash, is validated, and is copied to the DS1 location. A small program in a small corner of the external flash (DS2) is used to copy this data while the active DS1 area is not in use.

3. If the device has no on-chip flash at all, then external flash is used with two regions (DS1 and DS2) just like the first case.

Mapping the three cases to our devices:

Case	Devices	On-Chip	Flash	Default OTA Storage

		Flash Size	Organization	
1	20719 and 20720	1 MB	DS1 and DS2	on_chip_flash
2	20819 and 20820	256 kB	DS1	external_sflash on_chip_flash (for CYBT-213043-EVAL kit)
3	20706 and 20735	0	DS1 and DS2	external_sflash

The default OTA storage method is set in the BSP makefile. For example:

wiced_btsdk/dev-kit/bsp/TARGET_CYW920819EVB-02/CYW920819EVB-02.mk:

> *CY_CORE_OTA_FW_UPGRADE_STORE?=external_sflash*

4C.4.3 Applications for Loading New Firmware

The ModusToolbox installation contains peer applications that can be used to transmit new firmware over BLE – one for Windows, one for iOS and one for Android. The source code is available for all three and pre-compiled executables are provided for Android (.apk) and Windows (.exe) 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.

The pre-compiled peer applications can be found in the wiced_btsdk:

wiced_btsdk/tools/btsdk-peer-apps-ota/Windows/WsOtaUpgrade/Release/x64/WsOtaUpgrade.exe

wiced_btsdk/tools/btsdk-peer-apps-ota/Windows/WsOtaUpgrade/Release/x86/WsOtaUpgrade.exe

wiced_btsdk/tools/btsdk-peer-apps-ota/Android/LeOTAApp/app/build/outputs/apk/app-debug.apk

Windows

To use the Windows peer application, you must first copy the *.bin file from the build Debug directory of the 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:
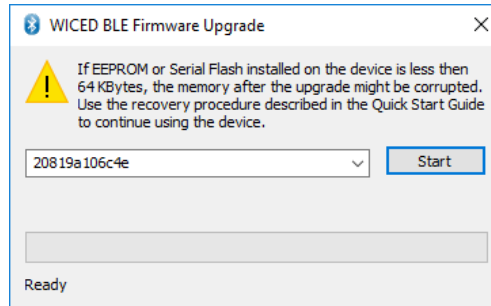
> *.\WsOtaUpgrade.exe app_CYW920819EVB-02.ota.bin*

Note: You can also just drag the .bin file onto WSOtaUpgrade.exe from a Windows file explorer window to run the application with the bin file as its input.
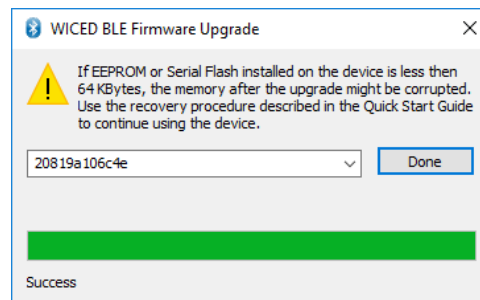
You will get a window that looks like the following. Select the device you want to update and click "OK".
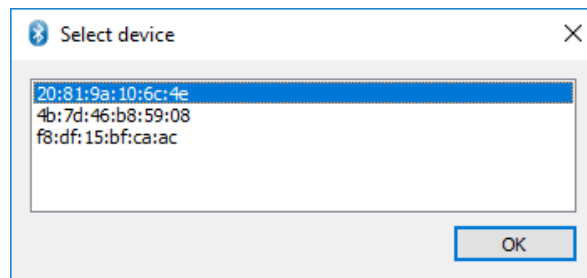
On the next window, verify that the device type is correct and click "Start".
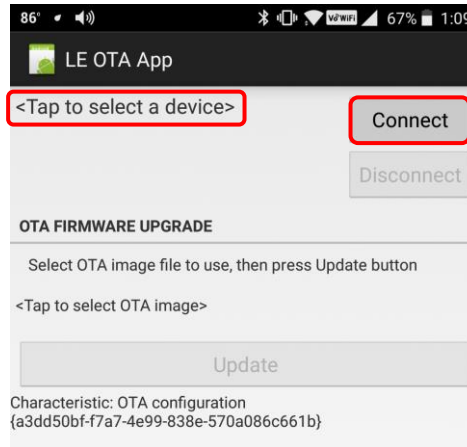
If the update worked, the window will show "Success" at the bottom. Click "Done" to close the window.

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 Debug 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:
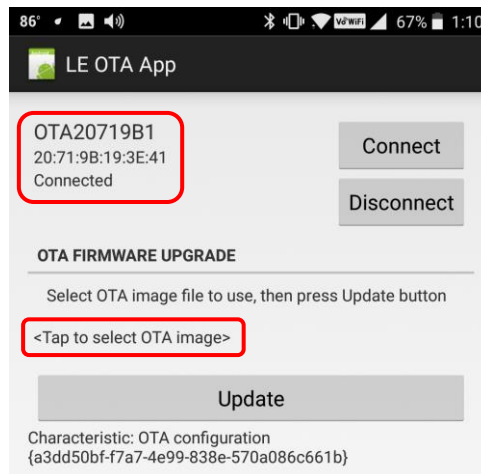
6. Tap where it says <Tap to select OTA Image>, navigate to where you saved the *.bin file on your device and select it. Once the file is selected, the screen will look like this:
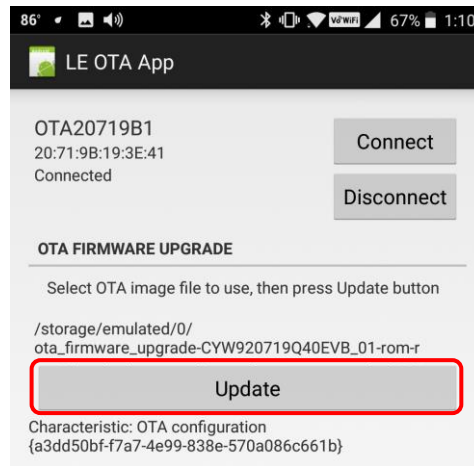
7. Tap the Update button. Once the update is done, you should see "Success" at the bottom of the screen. Disconnect from the device and close the app.
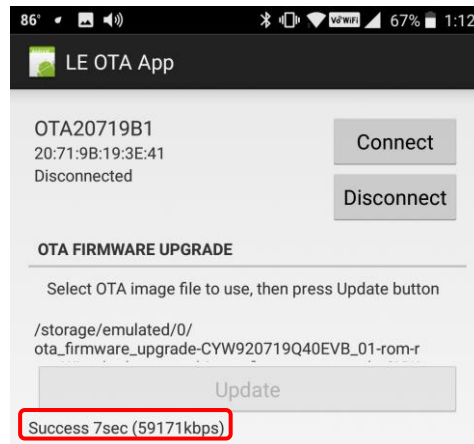
4C.4.4 OTA Firmware

You will use this in Exercise 4C.5

In the firmware, OTA requires the following:

makefile

The application must have the following setting in the makefile to setup the proper build settings:

```
OTA_FW_UPGRADE ?=1
```

This setting is what causes the OTA .bin file to be generated by the build. That's the file that you will send over the air to update the firmware.

You must also include the OTA library by adding the following to the makefile in the "Components (middleware libraries)" section (replace the existing "COMPONENTS+=" line):

```
#
# Components (middleware libraries)
#
ifeq ($(OTA_FW_UPGRADE),1)
COMPONENTS+=fw_upgrade_lib
endif
```

Finally, include the following in the "paths to shared_libs targets section" (add a new line below the line that says "SEARCH_LIBS_AND_INCLUDES…")

```
ifeq ($(OTA_FW_UPGRADE),1)
SEARCH_LIBS_AND_INCLUDES+=$(CY_SHARED_PATH)/dev-kit/libraries/btsdk-ota
endif
```

BLE OTA Service (Non-Secure)

The GATT database must have a Primary Service for OTA. This is a custom Service that is defined by Cypress with two Characteristics, one of which has a CCCD. Luckily, Cypress has included this service in the list of choices, so everything will automatically be setup correctly.

From the Server, right click, select Add Service, and choose "Custom OTA Firmware Upgrade". All of the Characteristics, Properties, and UUIDs should be left with the default values.
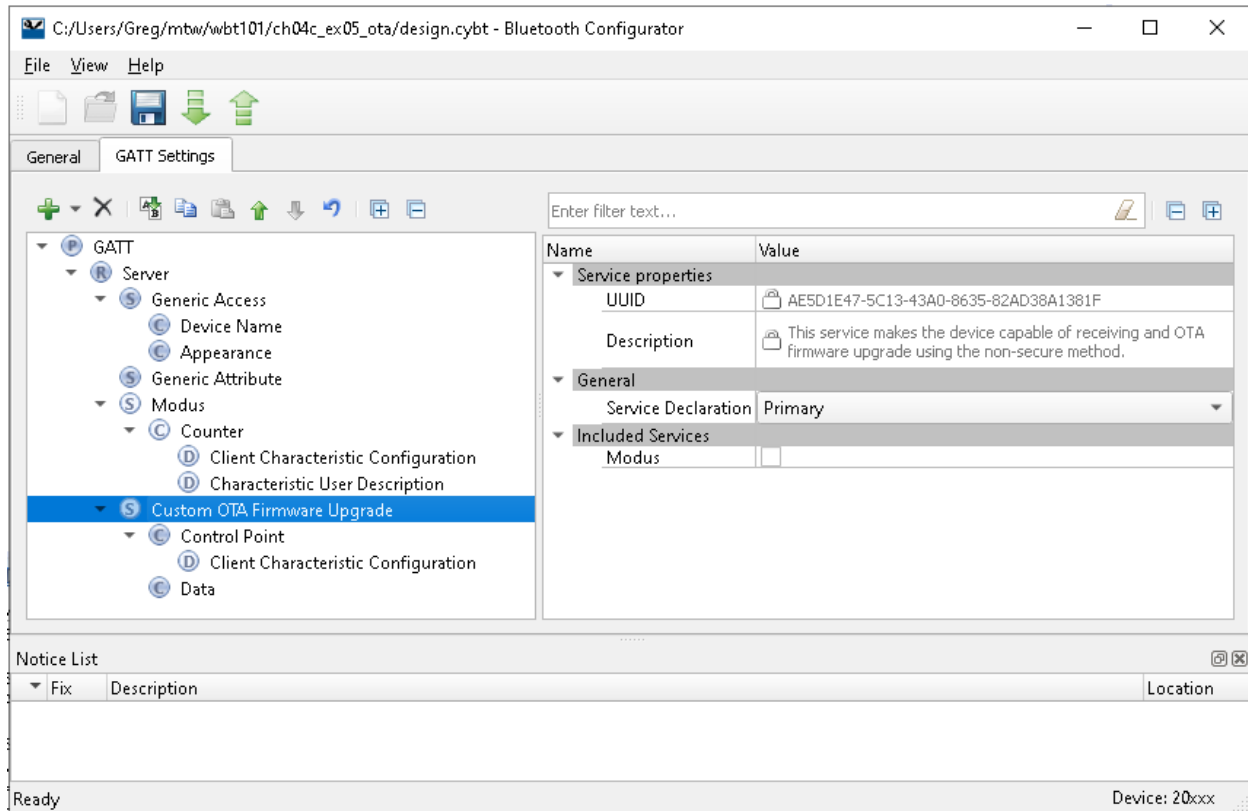
Initialization

Include the following in your source file to get access to the OTA API:

```
#include "wiced_bt_ota_firmware_upgrade.h"
```

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, NULL);
```

GATT Connect Event

In the GATT connection status event, it is necessary to pass the connection status information to the OTA library by calling the following (conn is a pointer of type `wiced_bt_gatt_connection_status_t` to the event connection status).

```
wiced_ota_fw_upgrade_connection_status_event(p_conn);
```

This should be called on <u>both</u> a connection and disconnection.

GATT Attribute Request Event

Several of the *GATT_ATTRIBUTE_REQUEST_EVT* events - namely *GATTS_REQ_TYPE_READ*, *GATTS_REQ_TYPE_WRITE*, *GATTS_REQ_TYPE_PREP_WRITE* and *GATTS_REQ_TYPE_CONF* - must call the

appropriate OTA function when the GATT request is made to one of the OTA characteristics or descriptors. <u>Note that OTA has its own library functions to handle these events, so they must be called for OTA events instead of the normal application code that is called for normal application functionality.</u>

Namely the following functions must be called:

For *GATTS_REQ_TYPE_READ*:

```
result = wiced_ota_fw_upgrade_read_handler(p_attr->conn_id, &(p_attr->data.read_req));
```

For *GATTS_REQ_TYPE_WRITE* or *GATTS_REQ_TYPE_PREP_WRITE*:

```
result = wiced_ota_fw_upgrade_write_handler(p_attr->conn_id, &(p_attr->data.write_req));
```

For *GATTS_REQ_TYPE_CONF*:

```
result = wiced_ota_fw_upgrade_indication_cfm_handler(p_attr->conn_id, p_attr->data.handle);
```

The OTA characteristics and descriptors in the switch statements that must trigger these calls are:

```
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:
```

This functionality has already been added for you in the template. Review the GATT_ATTRIBUTE_REQUEST_EVT case to understand what was added.

In addition, code has been added to the read condition to handle reads of the device name and appearance characteristics.

Buffer Pool Sizes

The large buffer pool must be at least the max MTU size plus 12. Both are defined in app_bt_cfg.c. Verify that the buffer pools are large enough in your application.

Disabling of PUART

For some reason, the PUART sometimes interferes with the OTA process when using the Windows peer app and may causes the update to fail. If this is the case, you can disable the debug UART in the GATT write handler before calling the OTA library write handler function. The code in the template contains that line of code but you can comment it out if you want to see additional debug messages during OTA.

4C.4.5 Secure OTA

You will use this in Exercise 4C.6

If you have a paired connection, all packets are encrypted, so why do we need Secure OTA? The answer is that you may want to verify that the firmware being sent to your device came from a reliable source and not from

some hacker trying to change your device's behavior unintentionally. Secure OTA allows you to verify that the new firmware is from that reliable source (i.e. the same source that initially programmed your device).

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.

BLE OTA Service (Secure)

In the Bluetooth Configurator, remove the "Custom OTA Firmware Upgrade" Service and replace it with the "Custom Secure OTA Firmware Upgrade" Service.

Key Generation

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

wiced_btsdk/tools/btsdk-utils/ecdsa256/bin/<Windows|Linux64|OSX>

The steps are (shown for Windows but others are similar):

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.

Header files and Global Variables

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

```c
#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:

```c
extern Point    ecdsa256_public_key;
```

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:

```c
/* Initialize OTA (secure) */
wiced_ota_fw_upgrade_init(&ecdsa256_public_key, NULL, NULL);
```

Build Firmware and Sign OTA Image

After building the firmware as usual, follow these steps to convert the output file to a signed output file:

1. Once the firmware is built, copy the bin file from the build Debug folder for the application to the *wiced_btsdk/tools/btsdk-utils/ecdsa256/bin/Windows* folder that contains the keys you generated earlier.

2. Open a command terminal or power shell window (shift-right-click in the folder from Windows explorer) and enter the command:

```
.\ecdsa_sign.exe .\app_CYW920819EVB-02.ota.bin
```

   This will produce a file called *app_CYW920819EVB-02.ota.bin.signed*. This operation uses the private key. Only someone with the private key can generate this signed image. Therefore, if you don't give out your private key, you are the only one who can supply new firmware to be loaded onto the device.

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

4C.5   Exercises

Exercise 4C.1 BLE Low Power

Introduction

In this exercise, you will create the applications from the HAL code example set. You will then program and analyze the low power application from that set of examples. The low power modes are covered in 4C.1

Application Creation

1. Create an application using the starter application HAL_20819EVB02. Before clicking next, change the application name to **ch04c_ex01_hal**. This will create a set of applications in your workspace.
2. Find the ch04c_ex01_hal.low_power application.
3. Open the makefile and set BT_DEVICE_ADDRESS?=random.
4. Open the Bluetooth configurator, change the name of the device to *<inits>*_lp, then save and close the configurator.
   a. Hint: this isn't currently used by the application, so you must also do the following:
      i. Open the file app_bt_cfg.c and find the line:

```
uint8_t BT_LOCAL_NAME[] = { 'l','o','w','_','p','o','w','e','r','_','2','0','8','x','x','\0' };
```

      ii. Change the name to <inits>_lp. For example:

```
uint8_t BT_LOCAL_NAME[] = { 'k', 'e', 'y', '_', 'l', 'p', '\0'};
```

5. Look at the READ_ME.md file and review the files low_power_208xx.c and 208xx_blt.c to familiarize yourself with how the application works.

Testing

1. Program the application onto the kit.
   a. Hint: If your device is in a 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. This will allow you to see sleep events and to determine how often the device wakes up and goes back to sleep during different operations.
3. The device beings in HID-off mode. Press the user button on the kit to wake it up and to start advertisements.
4. Open the PC CySmart app. Start scanning and then stop once your device appears.
5. Connect to the device in CySmart. You will see a notification asking to confirm the connection parameters. Select 'Yes'.
6. Discover all attributes in the GATT database, and Pair with the device.
7. Enable all notifications.
8. The application will send a battery level notification every 5 seconds.

9. Press the user button on the kit to disconnect and put the device into HID-off mode with a timed wake at 10 seconds. After that time, the device will reset. You can press the user button again to re-start advertisements.

Questions

1. Which lines in the code are used to configure and initialize sleep?

2. What is used as a wakeup source?

3. What is the name of the sleep permit handler function?

4. When are the connection interval min, max, latency, and timeout values updated in the code and what values are used?

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. This material is covered in the section titled Eddystone.

Application Creation

1. Create a new application called **ch04c_ex02_eddy** using the template in templates/ch04c_ex02_eddy. This is a very simple application with no GATT support. All it does is advertise.
2. The makefile has already been modified to set BT_DEVICE_ADDRESS?=random.
3. Open the makefile and:
    a. Add the following in the components section after the existing COMPOENTS line:

        `COMPONENTS +`= beacon_lib

    b. Add the following line in the "paths to shared_libs targets section" (add after the existing line that starts SEARCH_LIBS_AND_INCLUDES):

    `SEARCH_LIBS_AND_INCLUDES += $(CY_SHARED_PATH)/dev-kit/libraries/btsdk-ble`

4. In app.c, add the include for the beacon library: wiced_bt_beacon.h.
5. In app_set_advertisement_data() create an advertising packet with the following three elements:
    a. Hint: don't forget to update the number of elements in the array of advertising elements.
    b. BTM_BLE_ADVERT_TYPE_FLAG
        i. This is the same element you have used in chapter 4A. It is already included in the template.
    c. BTM_BLE_ADVERT_TYPE_16SRV_COMPLETE
        i. Two-byte Eddystone Service UUID (0xFEAA). Note that this is little-endian so the LSB must be the first element in the array.
    d. BTM_BLE_ADVERT_TYPE_SERVICE_DATA
        i. Eddystone Service UUID (again little-endian).
        ii. Eddystone frame type for URL (see EDDYSTONE_FRAME_TYPE_URL in wiced_bt_beacon.h).
        iii. Transmit power (use 0xF0)
        iv. URL Scheme Prefix (see EDDYSTONE_URL_SCHEME_1 in wiced_bt_beacon.h)
        v. Data (the URL itself as a list of characters)
            1. Hint: 'c', 'y', 'p', 'r', 'e', 's', 's'
        vi. Eddystone suffix for .com (see the Eddystone website).
6. In app_bt_cfg.c, change the value of wiced_bt_cfg_settings.ble_advert_cfg.high_duty_duration to 0, which indicates to the stack that advertising should never time out.
7. Build the application and program the kit.

Testing

1. 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.

2. 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.
3. Open the URL for www.cypress.com from the beacon app.
4. If you don't see your device in the beacon app it most likely means your packet isn't correct, so it isn't identifying your device as an Eddystone beacon. 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 use multi-advertising to send UID, URL and TLM frames to the listening devices. This material is covered in Eddystone.

Application Creation

1. Create a new application called **ch04b_ex03_multi** using the template in templates/ch04b_ex03_multi. This template already changes the advertising timeout, so you will not need to edit app_bt_cfg.c again.
2. The makefile has already been modified to set BT_DEVICE_ADDRESS?=random.
8. Open the makefile and:
    a. Add the following in the components section (add to the existing *COMPOENTS*+= line):

        ```
        COMPONENTS += beacon_lib
        ```

    b. Add the following line in the "paths to shared_libs targets section" (add after the existing line that starts SEARCH_LIBS_AND_INCLUDES):

        ```
        SEARCH_LIBS_AND_INCLUDES += $(CY_SHARED_PATH)/dev-kit/libraries/btsdk-ble
        ```

3. In app.c, add the include for the beacon library: wiced_bt_beacon.h.
4. The code in app_set_advertisement_data() shows how to set up the URL advertising using multi-advertising. It uses the library function wiced_bt_eddystone_set_data_for_url() which is included in the beacon_lib middleware library.
5. Find the global that defines the advertising parameters and speed up the advertising rate by changing the value of adv_int_max from BTM_BLE_ADVERT_INTERVAL_MAX (0x4000) to 100.
6. Create a global array for the new packet (e.g. tlm_packet[]) which will look similar to the url_packet array.
7. Make a copy of the four lines of code that create the URL packet and edit them to create a TLM packet.
    a. Instead of wiced_bt_eddystone_set_data_for_url(), call the function for TLM (unencrypted).
    b. Hint: Use 0 for the vbatt, temp, adv_cnt and sec_cnt arguments.
    c. Hint: You can re-use the packet_len argument.
    d. Hint: Use the provided #define BEACON_EDDYSTONE_TLM to set up the second packet.
8. Repeat the above two steps for a UID packet. That is:
    a. Start by creating another packet array (e.g. uid_packet).
    b. Make another copy of the four lines of code and edit them to create a UID packet.
        i. Instead of wiced_bt_eddystone_set_data_for_url(), call the function for UID.

        Hint: Use 0 for the ranging data argument.

        ii. Create arrays for uid_namespace and uid_instance similar to the url array so that you will be able to pick out your beacon in the app.

        Hint: The namespace is 10 bytes and the instance is 6 bytes.

        Hint: Use the provided #define BEACON_EDDYSTONE_UID to set up the third packet.

9. The template includes a timer that fires every 100ms. Use the provided callback function to increment the "seconds" parameter in the TLM advertising packet.

Note: the parameter is actually tenths of a second according to the Eddystone spec even though some scanner apps say seconds.

Hint: In the callback just reuse two lines of code from app_set_advertisement_data() to re-generate the packet and re-set the advertising data.

Hint: Provide the tenths variable value in when you generate the packet instead of a value of 0. Remember that the value needs to be sent little endian – you can use the macro SWAP_ENDIAN_32(tenths) to do the swapping for you.

Testing

1. Build and program the application 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 application, you will take an application that advertises the manufacturer ID for Cypress and a product ID of 0x2A and you will add a scan response packet that sends the Service UUID for the Modus Service. This material is covered in 4C.3

Application Creation

1. Create a new application called **ch04c_ex04_scan** using the template in templates/ch04c_ex04_scan. This template is a solution to exercise ch04b/ex02_ntfy.
2. The makefile has already been modified to set BT_DEVICE_ADDRESS?=random.
3. Open the Bluetooth configurator, change the name of the device to *<inits>*_scan, then save and close the configurator.
4. Copy the app_set_advertisement_data function to a new function called app_set_scan_response_data.
   a. Hint: Don't forget to add function prototype at the top of the file.
5. Update the scan response packet to send the 128-bit service UUID for the MODUS Service.
   a. Hint: the advertising types can be found in the file wiced_bt_ble.h.
   b. Hint: there is a macro for the service UUID in cycfg_gatt_db.h that you can use in the array that you set up.
   c. Hint: Remember, the Scan response packet doesn't have flags.
6. At the end of your new function, call the function to set the raw scan response data instead of the raw advertisement data.
7. Call your new function before starting advertising.

Testing

1. Build and program the application to your kit.
2. Using the CySmart mobile app, scan for devices and note that it reports "1 Service Advertised".
   a. Note: This feature is only on the iOS version of CySmart. It shows services advertised instead of the Bluetooth Device Address.
3. Open the PC version of CySmart and scan for your device. Stop scanning once you see it.
4. 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 an application that counts button presses to add OTA firmware upgrade capability. Once OTA support is added, you will modify the application to decrement the count instead of incrementing and you will upload the new firmware using OTA. This material is covered in 4C.4.4

Application Creation

1. Create a new application from templates/ch04c_ex05_ota.
    a. Hint: The template is just the solution application for the pairing exercise that counts button presses from the previous chapter.
2. The makefile has already been modified to set BT_DEVICE_ADDRESS?=random.
3. Open the makefile and:
    a. Set OTA_FW_UPGRADE?=1 in the app features.
    b. Add the following in the components section (below the existing *COMPOENTS*+= line):

    ```
    ifeq ($(OTA_FW_UPGRADE),1)
    COMPONENTS+=fw_upgrade_lib
    endif
    ```

    c. Add the following to the paths to shared libs targets section (add below the existing *SEARCH_LIBS_AND_INCLUDES*+=… line):

    ```
    ifeq ($(OTA_FW_UPGRADE),1)
    SEARCH_LIBS_AND_INCLUDES+=$(CY_SHARED_PATH)/dev-kit/libraries/btsdk-
    ota
    endif
    ```

4. Use the Bluetooth Configurator to:
    a. Change the device name *to <inits>_ota*.
    b. Add the Custom OTA Firmware Upgrade Service to the Server.
5. Save and close the Configurator.
6. Edit app.c:
    a. Add **#include** "wiced_bt_ota_firmware_upgrade.h"
    b. Add **wiced_ota_fw_upgrade_init**(NULL, NULL, NULL); after the GATT database is initialized.
    c. Add **wiced_ota_fw_upgrade_connection_status_event**(p_conn); in the GATT connection status event (for both connect and disconnect events).
    d. The GATT Attribute Request Event handler code has been provided in the template. Review it to understand how OTA GATT attribute request events are passed to the library.
        i. Hint: Search for "HANDLE_OTA" in app.c.

Testing

1. Build the application and program it to your kit.
2. Look at the PUART window during initialization and write down your kit's Bluetooth Device Address. You will need this to identify the correct device when you perform OTA upgrade.
3. Use CySmart to make sure the application functions as expected. Press the button and notice that the counter characteristic increases each time the button is pressed
   a. Hint: The Service with a single Characteristic is the Counter Service and the Service with two Characteristics is the OTA Service.
4. Disconnect from the kit in CySmart.
5. Clear the Device List in CySmart or remove the device from the paired devices on your phone. This is necessary because the kit will lose bonding information once the firmware is reloaded.
6. Unplug the kit from your computer. This will ensure that OTA is used instead of regular programming to update the firmware.
7. Update the application so that each button press decrements the value instead of incrementing it. Build the application without programming.
   a. Hint: In the Quick Panel, use the link "Build ch04c_ex05_ota Application".
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 of Windows do not support BLE.
   b. Hint: Don't forget to copy over the *.bin file from the build Debug folder every time you re-build the application so that you are updating the latest firmware.
   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, 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. This material is covered in 4C.4.5

Application Creation

1. Create a new application from templates/ch04c_ex06_ota_sec.
   a. Hint: The template is just the solution application for ch04c_ex05_ota modified to count up again.
2. The makefile has already been modified with all the changes from the previous exercise so you don't need to make any changes.
3. Use the Bluetooth Configurator to:
   a. Change the name to <inits>_*otas.*
   b. Remove the Custom OTA Firmware Service and add the Custom OTA Secure Firmware Upgrade Service.
4. Save and close the configurator.
5. Generate keys as described in the BLE OTA Service (Secure) section of this manual. Once generated, copy ecdsa256_pub.c to the project folder (same folder as app.c).
6. Add the additional required includes to app.c:

   ```
   #include "bt_types.h"
   #include "p_256_multprecision.h"
   #include "p_256_ecc_pp.h"
   ```

7. Add an external global variable declaration to app.c of type "Point" for the public key that is defined in ecdsa256_pub.c:

   ```
   extern Point    ecdsa256_public_key;
   ```

8. 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, NULL);
   ```

Testing

1. Build the application and program it to your kit.
2. Use CySmart to make sure the application functions as expected.
3. Disconnect from the kit in CySmart.
4. Clear the Device List in CySmart or remove the device from the paired devices on your phone. This is necessary because the kit will lose bonding information once the firmware is reloaded.
5. Unplug the kit from your computer. This will ensure that OTA is used instead of regular programming to update the firmware.
6. Make the same change as the previous exercise to count down instead of up on each button press.
7. Build the application.
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 application you must sign the bin file and copy the resulting *.bin.signed to the Windows OTA folder or Android device. Remember, instructions on signing the file are in Build Firmware and Sign OTA Image
   b. 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. Note that a failure may also mean that you didn't properly sign the image.
10. Once OTA upgrade is done, connect to the kit using CySmart and verify that the new firmware functionality is working.
11. Try doing OTA with the unsigned image. Notice that it will fail after loading the image. The original firmware will be retained in this case.