Chapter 8: PSoC 6 + 43xxx Bluetooth

After completing this chapter, you will understand how to create a Bluetooth application using a CYW43012 in hosted mode with a PSoC 6 acting as the host.

8.1 Embedded vs. Hosted Bluetooth

Up until now, we have been using a single Bluetooth device (the CYW20819) in embedded mode. That means that the user application and the entire Bluetooth stack runs on a single device. In this chapter, we will talk about using a CYW43012 in hosted mode with a PSoC 6 as the host.

Before we get into the details, I'd like to tell you about 2 other classes that are available:

- **ModusToolbox 101**: In this class will mainly use the PSoC 6 as a host to the Bluetooth device. We will cover some basic MCU functions and peripherals, but if you want to learn more, ModusToolbox 101 covers a lot of ground including using different IDEs with ModusToolbox, using the command line, managing applications and libraries, using FreeRTOS, Low Power operation, AnyCloud, etc.
- **Wi-Fi 101**: The CYW43012 device is capable of both Bluetooth and Wi-Fi operation. We will only cover Bluetooth in this chapter, but if you are interested in much more information about Wi-Fi, the Wi-Fi 101 class is the place to go.

The good news with regards to this class is that most of the Bluetooth API functions are identical to the ones you have used already with the CYW20819. Likewise, the firmware architecture is the same – you start the stack, provide a stack callback function, register the GATT database, provide a GATT callback function and then implement just about everything else inside the callbacks.

There are a few notable differences in the details which we will cover as we go.

8.2 CY8CKIT-062S2-43012

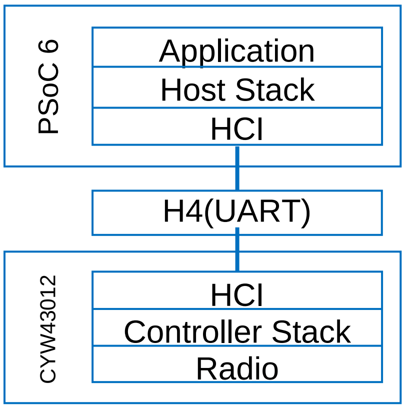
The kit that we will use for the exercises in this chapter is the CY8CKIT-062S2-43012. It's the same kit that is used for the ModusToolbox 101 and Wi-Fi 101 classes so if you did either of those classes you probably already have one lying around. If not, they are widely available on the web.

It has lots of great features like CapSense buttons and sliders, multiple user buttons and LEDs (including an RGB LED), a potentiometer for analog voltage input, Arudino headers and an external serial flash. We won't use a lot of those things in this class, but the ModusToolbox 101 class covers them.

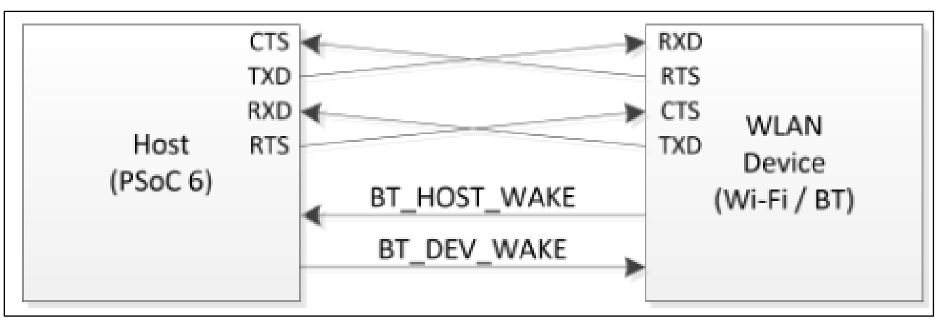
8.3 Host Interface

The interface between the PSoC 6 and the CYW43012 uses the Host Controller Interface (HCI). As we touched on in an earlier chapter, that means the lower level of the Bluetooth stack (the Controller Stack) will run on the CYW43012 while the higher level of the Bluetooth stack (the Host Stack) will run on the PSoC 6 along with the user application.

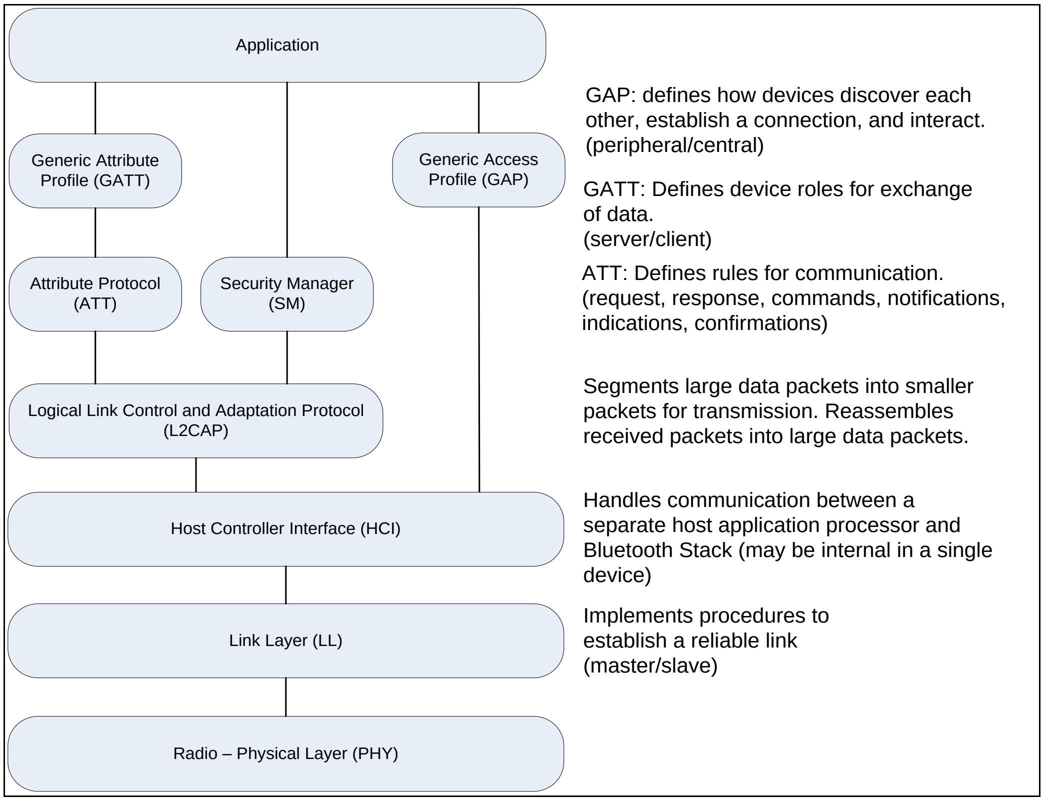
Here is a repeat of a picture that you saw in an earlier chapter that illustrates the connection.



The HCI interface physically runs using a 4 pin UART interface. The PSoC 6 that we are using has multiple UARTs on it so don't worry – you will still have a UART interface to print debug messages. There are also two wake pins that are used for low power which we will cover later.



The division of layers in the stack can be seen in another picture taken from an earlier chapter. Again, everything below HCI runs on the 43012 and everything above HCI runs on the PSoC 6.



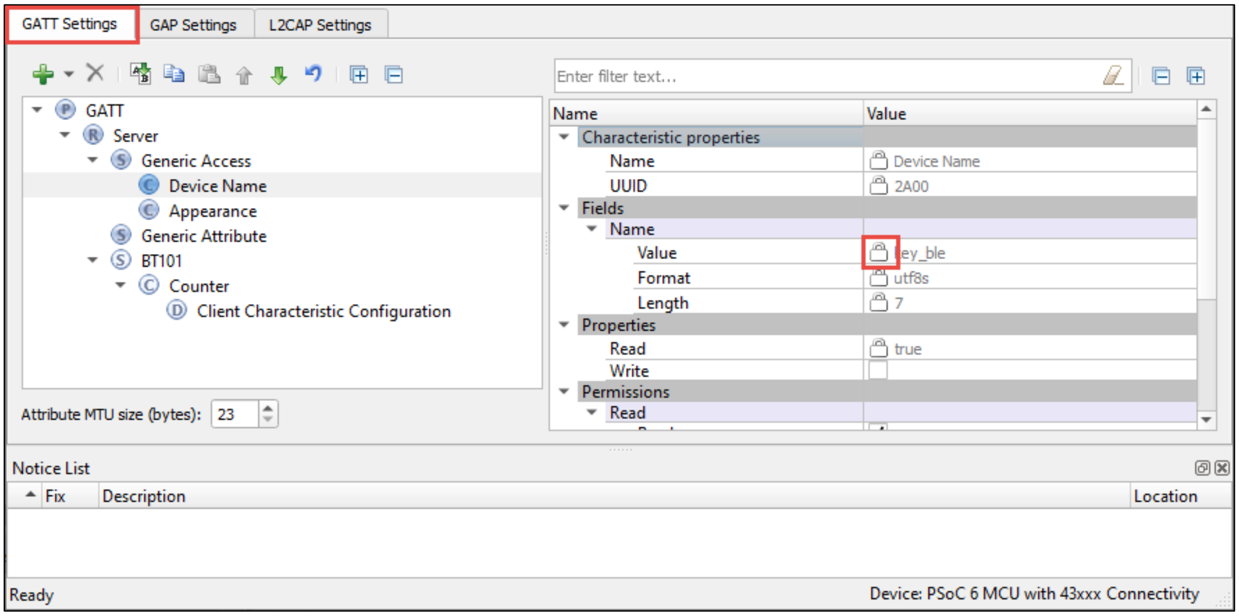
8.4 Bluetooth Configurator

The Bluetooth Configurator is used to simplify creation of the Bluetooth firmware. For the 43xxx devices, the configurator includes some additional configuration settings to generate even more of the code automatically. Namely, in addition to the GATT database information, the configurator generates the advertisement packets, scan response packets, and stack configuration information (i.e. the information that is in the *app_bt_cfg.c* file).

Because of the added functionality, the configurator has tabs that are not present for the Embedded applications that we looked at in previous chapters. Each tab is shown below with an explanation of what settings it controls.

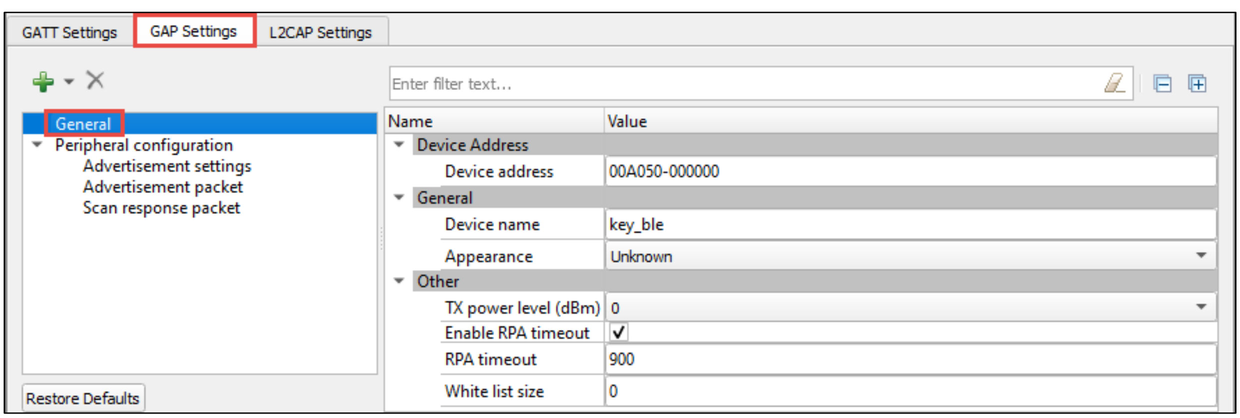
8.4.1 GATT Settings tab

The GATT Settings tab works just like it did for the Embedded applications with one notable difference. In Hosted applications, the name is not configurable on this tab (notice the lock symbol next to the name) – it will instead be set on the GAP Settings tab.

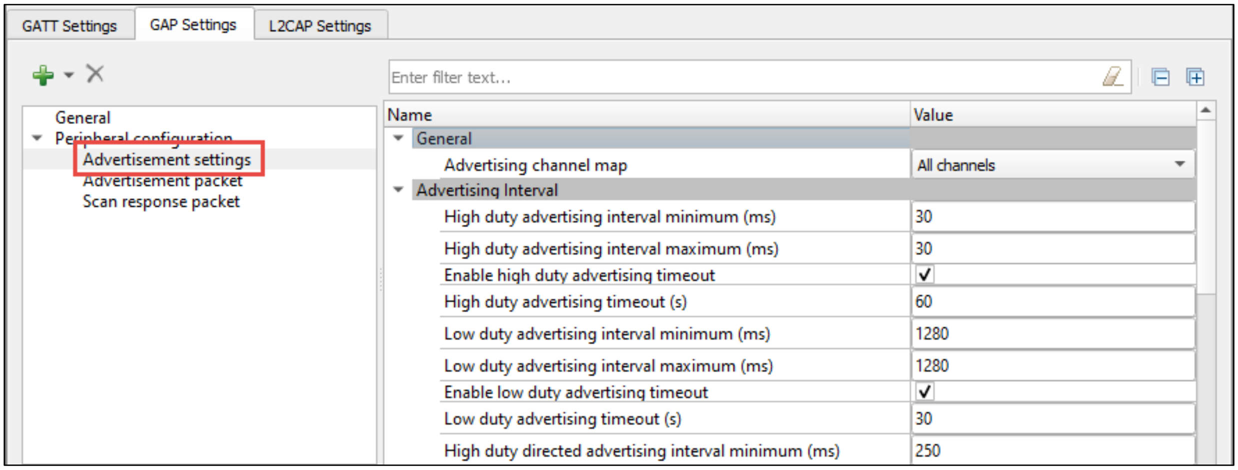


8.4.2 GAP Settings tab

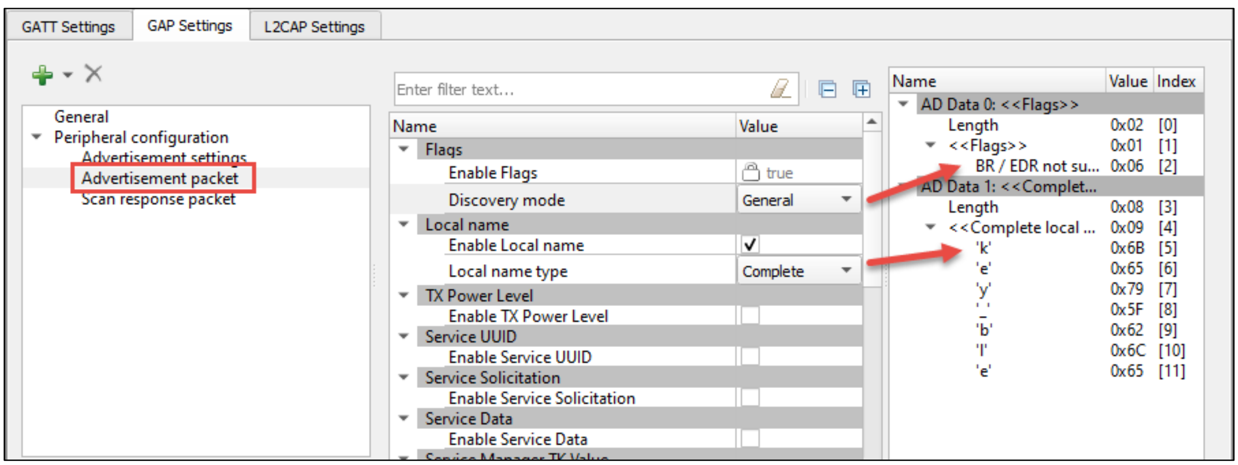
In addition to the device name, the GAP Settings tab allows you to set other "General" device level attributes such as BT address and RPA timeout.



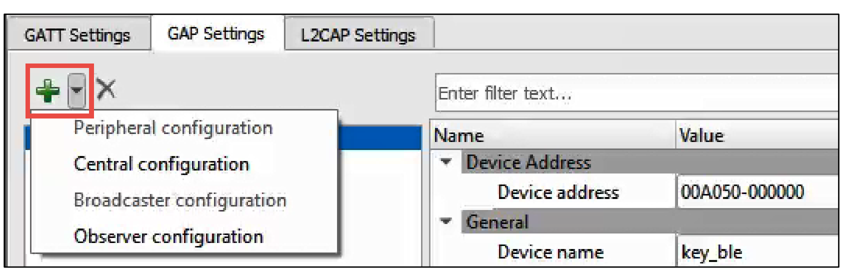
The "Advertisement settings" allows you to specify intervals and timeouts for each type of advertising.



The "Advertisement packet" and "Scan response" packet allow you to setup advertisement and scan response data. The middle panel is where you make your selections and the right panel shows what the resulting packet contents will be.

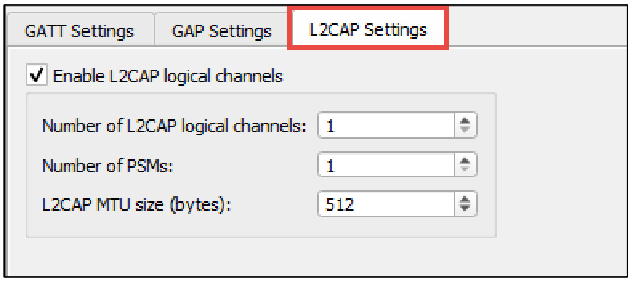


The "+" sign at the top allows you to add and configure settings for other roles besides peripherals. That is, you can enable/configure Central, Broadcaster, and Observer GAP settings.



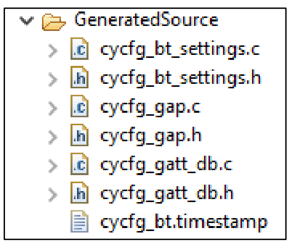
8.4.3 L2CAP Settings tab

The L2CAP Settings tab allows you to enable/disable L2CAP channels and to configure their settings. We will not use the L2CAP Settings tab in our exercises.



8.4.4 GeneratedSource

Once the configurator settings are saved, the GeneratedSource directory will look like this:



- *cycfg_gatt_db.c/.h* contain the GATT database information and they looks exactly like the GATT database files for the CYW20819 embedded applications.
- *cycfg_gap.c/.h* contain the device name, advertising settings, advertising packet and scan response packet.
- *cycfg_bt_settings.c/.h* contain the `wiced_bt_cfg_settings` structure. This is analogous to the *app_bt_cfg.c/.h* files from the CYW20819 applications but many of the values (such as advertisement settings) use macros that are defined in *cycfg_gap.h*.

8.5 43xxx Bluetooth Libraries and Settings

8.5.1 Libraries

The host stack running on the PSoC 6 requires an RTOS. By default, FreeRTOS is used. There is one top level library that must be included by the application:

- *bluetooth-freertos* – Bluetooth controller firmware written using FreeRTOS.

Several additional libraries are included indirectly by *bluetooth-freertos*:

- *freertos* – standard FreeRTOS library.
- *btstack* – Bluetooth host stack implementation.

- *abstraction-rtos* - used to abstract RTOS functions from the specific RTOS chosen. Allows other libraries to use generic RTOS functions that get mapped to the appropriate underlying RTOS functions.

- *clib-support* – support library that provides hooks to make C library functions such as malloc and free thread-safe.

8.5.2 Makefile Settings

Several `COMPONENT` settings are required in the application *Makefile* to include the appropriate library code. Specifically:

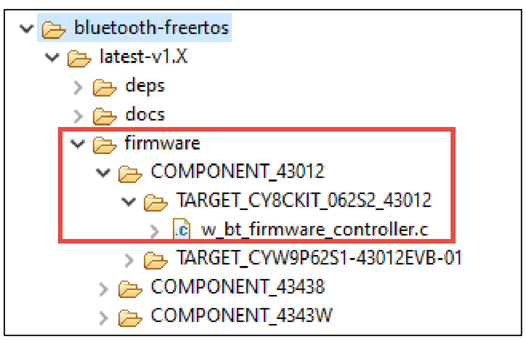
```
COMPONENTS=FREERTOS WICED_BLE
```

- `FREERTOS` is required to pull in the correct code from the *abstraction-rtos* library.

- `WICED_BLE` is required to pull in the correct code from the *btstack* library.

8.5.3 BSP Settings

The Bluetooth controller firmware for the CYW43012 is contained in the *bluetooth-freertos* library. It requires `TARGET` and `COMPONENT` settings in the application so that it can pull in the correct code. Those settings are typically done in the BSP's makefile. The directory structure of the library can be seen here:



In our case, the BSP we are using has a `TARGET` of `CY8CKIT_062S2_43012`. The BSP will include a `COMPONENT` for `43012`. With the `TARGET` and `COMPONENT` specified, the correct *w_bt_firmware_controller.c* file from the *bluetooth-freertos* library is included in the build.

For end-user applications and hardware, the *w_bt_firmware_controller.c* file may require modifications. In that case, that file must be replaced with a custom file which is typically placed in a custom BSP. Modifications to that file are beyond the scope of this class.

8.5.4 FreeRTOS Settings

The *clib-support* library requires a few changes in the *FreeRTOSConfig.h* file. This file should be copied from the *freertos* library (*freertos/<version>/Source/portable/FreeRTOSConfig.h*) to the root of the application. The changes required are:

```
#define configUSE_MUTEXES                     1
#define configUSE_RECURSIVE_MUTEXES           1
#define configSUPPORT_STATIC_ALLOCATION       1
#define configUSE_COUNTING_SEMAPHORES         1
#define configHEAP_ALLOCATION_SCHEME           (HEAP_ALLOCATION_TYPE3)
```

8.6 Firmware Architecture

The firmware to control the PSoC 6 is different than for the CYW20819 for things like GPIOs and other peripherals, but the Bluetooth part of the firmware is nearly identical.

You will be provided with a template to use when doing the exercises, but the notable differences are:

- Non-Bluetooth API functions are different than those used in the Embedded applications. For example, the functions to configure and set GPIO pins use the PSoC 6 HAL instead of the WICED HAL.
- Several header files must be included for the libraries and the BT configurator files:

```
/* FreeRTOS */
#include <FreeRTOS.h>
#include <task.h>
#include <queue.h>
/* bluetooth-freertos */
#include "cybt_platform_config.h"
/* btstack */
#include "wiced_bt_stack.h"

/* BT configurator */
#include "cycfg_bt_settings.h"
#include "cycfg_gap.h"
#include "cycfg_gatt_db.h"
```

- The user entry function is called `main` instead of `application_start`.
- The interface to the CYW43012 device must be initialized by calling `cybt_platform_config_init` with a pointer to a structure of type `cybt_platform_config_t`. The structure contains settings such as communication interface and low power interface. The structure is currently defined in the application but in the future, it will be defined in the BSP. The `cybt_platform_config_init` function must be called before calling `wiced_bt_stack_init`.
- The `wiced_bt_stack_init` function takes only two arguments instead of 3 – the callback function and the configuration settings. There is no argument to specify buffer pools. Rather, memory allocation is handled using RTOS functionality.

- The `wiced_bt_gatt_db_init` function takes a third argument to support dynamic databases. In our case, we will set that parameter to `NULL`.
- The FreeRTOS scheduler must be started by calling `vTaskStartScheduler` after initializing the stack.
- The standard C-function `printf` is used instead of `WICED_BT_TRACE`. To use `printf`, you must include the *retarget-io* library and initialize it by calling `cy_retarget_io_init`.
- The *Makefile* setting to create a random Bluetooth device address does not exist in the hosted solution (yet). In order to generate a random Bluetooth device address we will use the PSoC 6 HAL TRNG (true random number generator) API.
- Non-volatile memory storage is done using the Emulated EEPROM library. The functionality is similar, but the API is considerably different from the NVRAM API used in embedded CYW20819 applications. You will see this in the bonding exercise.
- Bluetooth stack functions should not be called from inside an interrupt service routine (ISR). Rather, the ISR should use an RTOS construct such as a notification, semaphore or queue to signal a task to proceed as required.
- Timers should use the `timer_ext` API instead of the `timer` API. For example, `wiced_init_timer` is replaced by `wiced_init_timer_ext` and `wiced_start_timer` is replaced by `wiced_start_timer_ext`. This is necessary to have timers that are thread-safe for the FreeRTOS implementation of the stack.

The firmware in the template is contained in the file *main.c* and the initial function is called `main`. It does the following:

1. Initialize the BSP resources.
2. Enable interrupts.
3. Initialize the retarget IO library and a GPIO to drive a user LED on the board.
4. Initialize the Bluetooth device and provide the platform configuration settings structure.
5. Initialize the Bluetooth stack and provide the BT management callback function and Bluetooth configuration settings structure.
6. Optionally set up other non-Bluetooth RTOS tasks that your application will require.
7. Start the FreeRTOS scheduler.

Once the FreeRTOS scheduler is started, the rest of it works the same as the Embedded Bluetooth applications. For example, you will get a `BTM_ENABLED_EVT` callback event when the stack has completed its initialization. You can use that callback to (for example) initialize other hardware, register a GATT callback functions, initialize the GATT database, enable pairing, start Bluetooth advertisements, etc.

The functions themselves (BT management callback, GATT callback, GATT database read/write functions) are work exactly the same as they do for the Embedded Bluetooth applications that you are already familiar with.

8.7 Low Power

The PSoC 6 and the CYW43012 both have extensive low power modes. Both devices work together to optimize overall system power. To enable that, there are three pins connected between the PSoC 6 and the CYW43012. The first is the BT power pin which the PSoC 6 uses to indicate wither the CYW43012 low power modes should be enabled. The other two are wake-up pins. These pins allow the BLE device to wake the PSoC 6 host and vice versa. In this way, either device can go into a low power mode whenever its application allows, and the other device can wake it up when it is needed to handle specific activity.

The `cybt_platform_config_t` structure has an entry for `controller_config` that defines which pin on the PSoC 6 connects to the Bluetooth device's low power control pin. It also has a `sleep_config` entry that defines which pins on the PSoC 6 connect to the two wake-up pins on the Bluetooth device and their polarities.

The wake-up pins can be set using the Low Power Assistant (LPA) in the configurator but since they are already defined as part of the BSP it is simpler to just specify them directly in the `cybt_platform_config_t` structure in the application. For example, the following will use the LPA settings for the wake-up pins if the LPA is enabled or it will use the BSP's settings if not.

```
.controller_config =
{
  .bt_power_pin      = CYBSP_BT_POWER,
  .sleep_mode =
  {
    #if (bt_0_power_0_ENABLED == 1) /* BT Power control is enabled in the LPA */
    .sleep_mode_enabled   = CYCFG_BT_LP_ENABLED,
    .device_wakeup_pin    = CYCFG_BT_DEV_WAKE_GPIO,
    .host_wakeup_pin      = CYCFG_BT_HOST_WAKE_GPIO,
    .device_wake_polarity = CYCFG_BT_DEV_WAKE_POLARITY,
    .host_wake_polarity   = CYCFG_BT_HOST_WAKE_IRQ_EVENT
    #else  /* BT Power control is disabled in the LPA, use BSP's LP config */
    .sleep_mode_enabled   = WICED_TRUE,
    .device_wakeup_pin    = CYBSP_BT_DEVICE_WAKE,
    .host_wakeup_pin      = CYBSP_BT_HOST_WAKE,
    .device_wake_polarity = CYBT_WAKE_ACTIVE_LOW,
    .host_wake_polarity   = CYBT_WAKE_ACTIVE_LOW
    #endif
}
```

For low power examples and more details on the Low Power Assistant, see the ModusToolbox 101 class.

8.8 Exercises

These exercises are a repeat of exercises that were done in earlier chapters to introduce basic BLE functionality, notifications, pairing, and bonding. This will demonstrate how similar the firmware is between Embedded and Hosted Bluetooth applications. More background on the functionality can be found in the prior chapters where the concepts were first covered.

Exercise - 8.1 Basic BLE Peripheral

Introduction

For the first exercise, you will recreate the basic BLE peripheral exercise from the BLE Basics chapter. That is, it will have a Service called BT101 with a Characteristic called LED that you can read/write from the client. Writing the value will cause the LED to turn on or off.

Application Creation

☐ 1. Create a new application with the CY8CKIT-062S2-43012 BSP.

Use the template application from the class files under *Templates/ch08_ex01_ble*.

☐ 2. Open the Bluetooth Configurator.

☐ 3. On the GAP Settings tab:

☐ a. Enter a Device name of <inits>_ble

Hint Leave the address as-is. We will override this with a random value in the firmware using the random number generator from the PSoC 6 HAL

☐ b. Disable the RPA timeout.

☐ c. Configure the advertisement packet to include the complete local name.

☐ 4. On the GATT Settings tab:

☐ a. Add a new Custom Service and rename it BT101.

☐ b. Rename the Custom Characteristic to LED.

☐ c. Configure the LED Characteristic as a uint8 with an initial value of 0. Set Properties for Read and Write.

☐ 5. Save your edits and close the configurator.

☐ 6. In *main.c*, search for "TODO Ex 01" and fill in the required code.

Hint The functions are identical to those required for the first exercise from the basic BLE chapter using the Embedded solution with a few minor exceptions:

a. There are three include files needed from the configurator instead of one.
b. Use `cyhal_gpio_write` instead of `wiced_hal_gpio_set_pin_output`.
c. Use `CYBSP_USER_LED` instead of `LED2` for the name of the LED.
d. The `wiced_bt_gatt_db_init` function requires a third argument of `NULL`.
e. Use `printf` instead of `WICED_BT_TRACE`.

Testing

1. Open a UART terminal to the KitProg3 UART with a baud rate of 115200.
2. Program the kit.
3. Open the mobile CySmart app and connect to your device.
4. Go to the Service and then the Characteristic. Write a value of 1 to turn on the LED and a value of 0 to turn off the LED.
5. Disconnect from the kit.

Exercise - 8.2 Notifications

Introduction

In this exercise, we will add a new Characteristic that counts button presses to the previous exercise. The Characteristic will allow notifications.

Application Creation

1. Create a new application with the CY8CKIT-062S2-43012 BSP.

   Use the template application from the class files under *Templates/ch08_ex02_notify*.
   Hint The template is just the solution to the previous exercise.
2. Open the Bluetooth Configurator.
3. On the GAP Settings tab, Change the device name to <inits>_ntfy.

   Hint the RPA timeout is already disabled and the advertisement packet is set to advertise the device name.
4. On the GATT Settings tab:
   a. Add a new Custom Characteristic to the BT101 Service and rename it "Counter".
   b. Set the format to `unit8` and the initial value to 0.
   c. Set the Properties for "Read" and "Notify".
   d. Add a Descriptor to the Button Characteristic for Client Characteristic Configuration.
5. Save your edits and close the configurator.

☐ 6. In *main.c*, make the following changes:

Hint Search for "TODO Ex 02" to find the locations for the required changes.

☐ a. Declare a global variable called `connection_id`.

Upon a GATT connection (i.e. in `app_gatt_callback`), save the connection ID. Upon a GATT disconnection, reset the connection ID. The ID is needed to send a notification. You need to tell it which connected device to send the notification to. In our case we only allow one connection at a time, but there are devices that allow multiple connections.

```
Global Variable:
uint16_t connection_id = 0;

GATT Connection:
/* Handle the connection */
connection_id = p_conn->conn_id;

GATT Disconnection:
/* Handle the disconnection */
connection_id = 0;
```

☐ b. Declare a global variable called `CounterTaskHandle`.

This will be the handle for a task we will create that will send notifications when the button is pressed. The task will be unlocked by the button ISR.

```
TaskHandle_t CounterTaskHandle = NULL;
```

☐ c. Configure `CYBSP_USER_BTN` as a falling edge interrupt during initialization.

```
/* Configure CYBSP_USER_BTN for a falling edge interrupt */
cyhal_gpio_init(CYBSP_USER_BTN,CYHAL_GPIO_DIR_INPUT,
CYHAL_GPIO_DRIVE_PULLUP, CYBSP_BTN_OFF);
cyhal_gpio_register_callback(CYBSP_USER_BTN, button_cback, NULL);
cyhal_gpio_enable_event(CYBSP_USER_BTN, CYHAL_GPIO_IRQ_FALL, 3, true);
```

☐ d. Create a function (and a declaration) for the button callback.

In the callback we will just increment the Button Characteristic value and unlock the counter task.
Hint The array `app_bt101_counter` was created by the Bluetooth Configurator. It holds the value for our counter characteristic. The name that the configurator uses is of the form: `app_<service_name>_<characteristic_name>`.
The button callback function will look like the following:

```
void button_cback(void *handler_arg, cyhal_gpio_irq_event_t event)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Increment button counter */
    app_bt101_counter[0]++;

    /* Notify the counter task that the button was pressed */
    vTaskNotifyGiveFromISR( CounterTaskHandle,
                            &xHigherPriorityTaskWoken );

    /* If xHigherPriorityTaskWoken is now set to pdTRUE then a context
       Switch should be performed to ensure the interrupt returns
       directly to the highest priority task.  The macro used for this
       purpose is dependent on the port in use and may be called
       portEND_SWITCHING_ISR(). */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

☐ e. Create a task (and a function declaration) to send a notification.

The function will wait to be unlocked by the button ISR and will send a notification if we have a connection and the notification is enabled.
The function will look like the following:

```
/* Counter task to send a notification */
static void counter_task(void * arg)
{
    /* Notification values received from ISR */
    uint32_t ulNotificationValue;
    while(true)
    {
       /* Wait for the button ISR */
       ulNotificationValue = ulTaskNotifyTake(pdFALSE, portMAX_DELAY);

       /* If button was pressed increment value and check to see if a
        * BLE notification should be sent. If this value is not 1, then
        * it was not a button press (most likely a timeout) that caused
        * the event so we don't want to send a BLE notification. */
        if (ulNotificationValue == 1)
        {
            if( connection_id ) /* Check if we have an active
                                   connection */
            {
                /* Check to see if the client has asked for
                   notifications */
                 if( app_bt101_counter_client_char_config[0] &
                 GATT_CLIENT_CONFIG_NOTIFICATION )
                 {
                     printf( "Notifying counter change (%d)\n",
                         app_bt101_counter[0] );
                     wiced_bt_gatt_send_notification(
                         connection_id,
                         HDLC_BT101_COUNTER_VALUE,
                         app_bt101_counter_len,
                         app_bt101_counter);
                 }
             }
         }

         else
         {
                /* The call to ulTaskNotifyTake() timed out. */
        }
    }
}
```

f. Start the counter task in main before starting the scheduler.

```
/* Start task to handle Counter notifications */
xTaskCreate (counter_task,
"CounterTask",
TASK_STACK_SIZE,
NULL,
TASK_PRIORITY,
&CounterTaskHandle);
```

g. Add a debug message in `app_gatt_set_value` so you know when notifications get enabled/disabled.

Hint The switch statement is already there but you need to add a new case.

```
switch( attr_handle )
{
    case HDLD_BT101_COUNTER_CLIENT_CHAR_CONFIG:
        printf("Setting notify (0x%02x, 0x%02x)\n", p_val[0],
                p_val[1]);
        break;
```

Testing

1. Open a UART terminal to the KitProg3 UART with a baud rate of 115200.
2. Program the kit.
3. Open the mobile CySmart app and connect to your device.
4. Go to the Service and then the Button Characteristic. Enable Notifications.
5. Press User Button 1 and observe the value incrementing on CySmart.
6. Disconnect from the kit.

Exercise - 8.3 Pairing

Introduction

In this exercise, we will add a pairing to the previous exercise.

Application Creation

1. Create a new application with the CY8CKIT-062S2-43012 BSP. Use the template application from the class files under *Templates/ch08_ex03_pair.*

   Hint The template is just the solution to the previous exercise.
2. Open the Bluetooth Configurator.
3. On the GAP Settings tab, change the device name to <inits>_pair.

4. On the GATT Settings tab:
    a. In the Counter characteristic, set the "Read Authentication Required" permission, which will make the peripheral reject read requests unless the devices are paired.

       Hint You MUST leave "Read" checked also. It will not work with just "Read Authentication Required" checked.
    b. Update the Client Characteristic Configuration descriptor to require authenticated read and write.

       This will cause the application to require pairing to view or change the notification settings.
       Hint You must leave "Read" and "Write" checked too.
5. Save your edits and close the configurator.
6. In *main.c*, make the following changes.
    a. Look for the call to `wiced_bt_set_pairable_mode` mode and set the first argument to `WICED_TRUE` to allow pairing.

       Hint Leave the second argument as `WICED_FALSE` - when it is set to true, this argument indicates that ONLY previously paired devices are allowed to connect.
    b. In the `BTM_PAIRING_IO_CAPABILITIES_BLE_REQUEST_EVT` management case tell the central that you require MITM protection, but the device has no IO capabilities. The required code is shown below.

```
p_event_data->pairing_io_capabilities_ble_request.auth_req =
BTM_LE_AUTH_REQ_SC_MITM_BOND;
p_event_data->pairing_io_capabilities_ble_request.init_keys =
BTM_LE_KEY_PENC|BTM_LE_KEY_PID;
p_event_data->pairing_io_capabilities_ble_request.local_io_cap =
BTM_IO_CAPABILITIES_NONE;
p_event_data->pairing_io_capabilities_ble_request.max_key_size = 0x10;
p_event_data->pairing_io_capabilities_ble_request.resp_keys =
BTM_LE_KEY_PENC|BTM_LE_KEY_PID;
p_event_data->pairing_io_capabilities_ble_request.oob_data =
BTM_OOB_NONE;
```

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

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

Testing

1. Open a UART terminal to the KitProg3 UART with a baud rate of 115200.
2. Program the kit.
3. Open the mobile CySmart app and connect to your device.
4. Watch the UART messages during the next steps to see which BT events occur.
5. Connect to the device.

6. Open the GATT browser, navigate to the Counter characteristic (the one with Read and Notify Properties), press Descriptors and then the Client Characteristic Configuration.
7. If requested, accept the invitation to pair the devices.
8. Return to the Counter Characteristic, enable Notifications and observe the Counter value as you press the button on the kit.
9. Disconnect from the mobile CySmart app.
10. Go to the phone's Bluetooth settings and remove the <inits>_pair device from the paired devices list.

    This is necessary so that when you re-program the kit the phone won't have stale bonding information stored which could prevent you from re-connecting. In the next exercise we'll store bonding information on the BLE device so that you will be able to leave the devices paired if you desire.

Exercise - 8.4 Bonding

Introduction

In this exercise, we will save bonding information to the EEPROM. It also uses a Resolvable Private Address. This exercise has been fully implemented in the template. User LED2 is used to indicate whether bonding information has been saved or not. The states are:

OFF	Not advertising
Slow Blink	Advertising, not bonded
Fast Blink	Advertising, bonded
ON	Connected

Once bonding information has been stored, it can be erased from the device by entering the letter 'e' in the UART terminal window.

Application Creation

1. Create a new application with the CY8CKIT-062S2-43012 BSP. Use the template application from the class files under *Templates/ch08_ex02_bond*.
2. Open the Bluetooth Configurator.
3. On the GAP Settings tab Change the device name to <inits>_bond.
4. Verify the box for **Enable RPA timeout** is checked and the RPA timeout is 900.
5. Save your edits and close the configurator.
6. Review the code to see how bonding information is stored and retrieved using the emulated EEPROM library.

Testing

1. Open a UART terminal window to the PUART.
2. Build the application and program it to the board.
3. Open the CySmart mobile application.
4. Start scanning and locate your device.

   Your device shows up with a Random Bluetooth address now since privacy is enabled. (Note that on iOS you can't see the Bluetooth device address).
5. Connect to your device, open the GATT browser, click on the Service, and then on the Counter Characteristic. Click "Read" to get pairing to occur.
6. If requested, accept the invitation to pair the devices.
7. Note down the Stack events that occur during pairing. This information is displayed in the UART.
8. Disconnect from the device. Do NOT remove the device from the phone's list of paired devices this time.

   Hint You will notice that the LED is blinking at 5 Hz. The firmware was written to do this when it is not connected but has bonding information stored.
9. Re-scan and find your device in the list.
10. Re-connect to your device and read the Counter Characteristic.
11. Once again note down the Stack events that occur during pairing. You will notice that fewer steps are required this time.
12. Disconnect again.
13. Reset or power cycle the board.

    Hint If you power cycle the board, you will need to either reset or re-open the UART terminal window.
14. Start a scan, find your device in the list, connect to your device for a third time and then read the Counter Characteristic.
15. Note down the Stack events that occur this time during pairing. Compare to the previous two connections.
16. Disconnect again.
17. Remove the device from the list of bonded devices in the Phone's Bluetooth settings.
18. Start a scan and find your device.

19. Connect to your device and try to read the Counter Characteristic.

    Note that pairing will not complete because CySmart no longer has the required keys to use. You will not be able to read the Counter value because it requires an authenticated connection.
    Hint: If you look in the UART window you will see a message about the security request being denied.

20. Disconnect from the device.

21. Press "e" in the UART window to erase bonding information and reset the kit.

    This forces it to restart advertising (it would restart advertising automatically if you waited long enough for the disallowed pairing operation to timeout).
    Note that `CYBSP_USER_LED` begins blinking at 1 Hz. This indicates that the bonding information has been cleared from the device and it will now allow a new connection.

22. Scan, Connect, and attempt to read the Counter Characteristic again. Allow pairing if requested. This time it should work.

23. Note the steps that the firmware goes through this time.

24. Disconnect a final time and remove the device from the phone's paired Bluetooth devices so that the saved boding information won't interfere with any future tests.

    Hint You should clear the bonding information anytime you are going to reprogram the kit or otherwise clear bonding information since the BLE device will no longer have the bonding information on its side.

Overview of Changes

- There are a lot of messages printed in this example for learning purposes. In a real application, most if not all these messages would be removed.

- The LED characteristic and functionality were removed so that `CYBSP_USER_LED2` can indicate connection status. A PWM is added that will operate the LED as follows:

Advertising?	Connected?	Bonded?	LED
No	No	N/A	OFF
No	Yes	N/A	ON
Yes	No	No	Blinking at 1 Hz
Yes	No	Yes	Blinking at 5 Hz
Yes	Yes	N/A	N/A - this case doesn't occur

- A structure called `bondinfo` is created which holds the `BD_ADDR` of the bonded device and the value of the Button CCCD. The `BD_ADDR` is used to determine when we have reconnected to the same device while the CCCD value is saved so that the state of notifications can be retained across connections for bonded devices.

- Before initializing the GATT database, existing keys (if any) are loaded from EEPROM. If no keys are available this step will fail so it is necessary to look at the result of the EEPROM read. If the read was successful, then the keys are copied to the address resolution database and the variable called `bonded` is set as `TRUE`. Otherwise, it stays `FALSE`, which means the device can accept new pairing requests.

- In the `BTM_SECURITY_REQUEST_EVENT` look to see if bonded is `FALSE`. Security is only granted if the device is not bonded.

- In the Stack event `BTM_PAIRING_COMPLETE_EVT` if bonding was successful write the information from the `bondinfo` structure into the EEPROM and set bonded to `TRUE`.

    - This saves `bondinfo` upon initial pairing. This event is not called when bonded devices reconnect.

- In the Stack event `BTM_ENCRYPTION_STATUS_EVT`, if the device is bonded (i.e. bonded is `TRUE`), read bonding information from the EEPROM into the `bondinfo` structure.

    - This reads `bondinfo` upon a subsequent connection when devices were previously bonded.

- In the Stack event `BTM_PAIRED_DEVICE_LINK_KEYS_UPDATE_EVT`, save the keys for the peer device to EEPROM.

- In the Stack event `BTM_PAIRED_DEVICE_LINK_KEYS_REQUEST_EVT`, read the keys for the peer device from EEPROM.

- In the Stack event `BTM_LOCAL_IDENTITY_KEYS_UPDATE_EVT`, save the keys for the local device to EEPROM.

- In the Stack event `BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT`, read the keys for the local device from EEPROM.

- In the GATT connect callback:

    - For a connection, save the `BD_ADDR` of the remote device into the `bondinfo` structure. This will be written to EEPROM in the `BTM_PAIRING_COMPLETE_EVT`.
    - For a disconnection, clear out the `BD_ADDR` from the `bondinfo` structure and reset the CCCD to 0.
    - In the GATT set value function, save the Button CCCD value to the `bondinfo` structure whenever it is updated and write the value into EEPROM.
- The UART is configured to accept input with a receive callback. Instead of using *retarget-io*, the UART is used directly from the HAL. The `rx_cback` function sends the received character to a UART task. This is done because you cannot call any BT Stack functions from inside the ISR.

- The UART task looks for the key "e". If it has been sent, it sets bonded to `FALSE`, removes the bonded device from the list of bonded devices, removes the device from the address resolution database, and clears out the bonding information stored in EEPROM.

- Finally, privacy is enabled in *wiced_bt_cfg.c* by updating the `rpa_refresh_timeout` to `WICED_BT_CFG_DEFAULT_RANDOM_ADDRESS_CHANGE_TIMEOUT`.

Questions

☐ 1. What items are stored in EEPROM?

☐ 2. Which event stores each piece of information?

☐ 3. Which event retrieves each piece of information?

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

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