Chapter 5d: PSoC 6 Low Power

After completing this chapter, you will understand various low power concepts and how to use the Low Power Assistant (LPA).

Table of Contents

Document conventions

Convention	Usage	Example
`Courier New`	Displays code	`CY_ISR_PROTO(MyISR);`
Italics	Displays file names and paths	*sourcefile.hex*
[**bracketed, bold**]	Displays keyboard commands in procedures	[**Enter**] or [**Ctrl**] [**C**]
Menu > Selection	Represents menu paths	**File > New Project > Clone**
Bold	Displays commands, menu paths and selections, and icon names in procedures	Click the **Debugger** icon, and then click **Next**.

5d.1 Introduction

If we think in the context of Low Power, not only do we want to be able to configure and control how every part of the system works, such as the MCU or Wi-Fi, but we also want configurability and control over how the parts of the system interact with each other. We also want seamless integration with the RTOS, peripherals, and connectivity subsystems.

Plus, it must work just fine in a real-world environment flooded with packets sent to and from hundreds of hotspots and Bluetooth devices around us, as well as other electromagnetic impulses we don't care about. "Works just fine" in the context of Low Power means the battery lasts as long as it is expected by the customer of your "Thing". Save more energy and you win.

ModusToolbox provides tools and middleware to set up and use the low power features of the PSoC 6 MCU and the connectivity devices on your board. We are referring to this set of tools and middleware as the Low Power Assistant. It consists of:

- Device Configurator: Used to configure the peripherals and system resources of the PSoC 6 MCU, configure the low power features of the connectivity device, such as wakeup pins, packet filters, offloads for Wi-Fi, as well as Bluetooth low power. The Device Configurator can also be used to configure the RTOS integration parameters such as the System Idle Power Mode.
- Low Power Assistant (LPA) middleware: This consumes the configuration generated by the Device Configurator, then configures and handles the packet filters, offloads, host wake functionality, etc.

The Low Power Assistant feature is supported for all PSoC 6 MCUs as well as for the CYW43012 and 4343W connectivity modules. The combination of PSoC 6 and CYW43012 offers the lowest power consumption and is available on the CY8CKIT-062S2-43012.

5d.2 Low Power Documentation and Collateral

The best starting point to learn about the Low Power Assistant feature and low power in general is the dedicated Application Note listed below. It includes references to code examples which use the LPA library. The LPA library also offers online documentation with quick start guides and code snippets that show you how to setup and test every feature.

5d.2.1 Application Note

An excellent source of information that you could start from if you didn't go through this chapter is the low power system design application note. Reading it after this chapter will help you to refresh and reinforce the knowledge you have just gained.

- AN227910 - Low-Power System Design with CYW43012 and PSoC 6 MCU

5d.2.2 Low Power Assistant Documentation

The LPA library and its top-level README.md file can be found at:

- https://github.com/infineon/lpa

As with most libraries, there is an API guide included in the library:

- https://infineon.github.io/lpa/lpa_api_reference_manual/html/index.html

Our exercises are mostly based upon the example code snippets provided in the API guide.

5d.3 Overview

5d.3.1 Power Modes

The PSoC 6 has 3 CPU power modes (for both the CM4 and CM0+ and 4 system power modes. A summary of each of the 7 modes is shown in the tables below.

5d.3.1.1 CPU Power Modes

Mode	Features	Resources Available	Wakeup Sources
Active	CPU executing code	All peripherals available CPU clock on	N/A
Sleep	CPU WFI/WFE	All peripherals available CPU clock off	Any peripheral interrupt
Deep Sleep	CPU WIF/WFE Requests System Deep Sleep mode	All peripherals available CPU clock off	Any peripheral interrupt

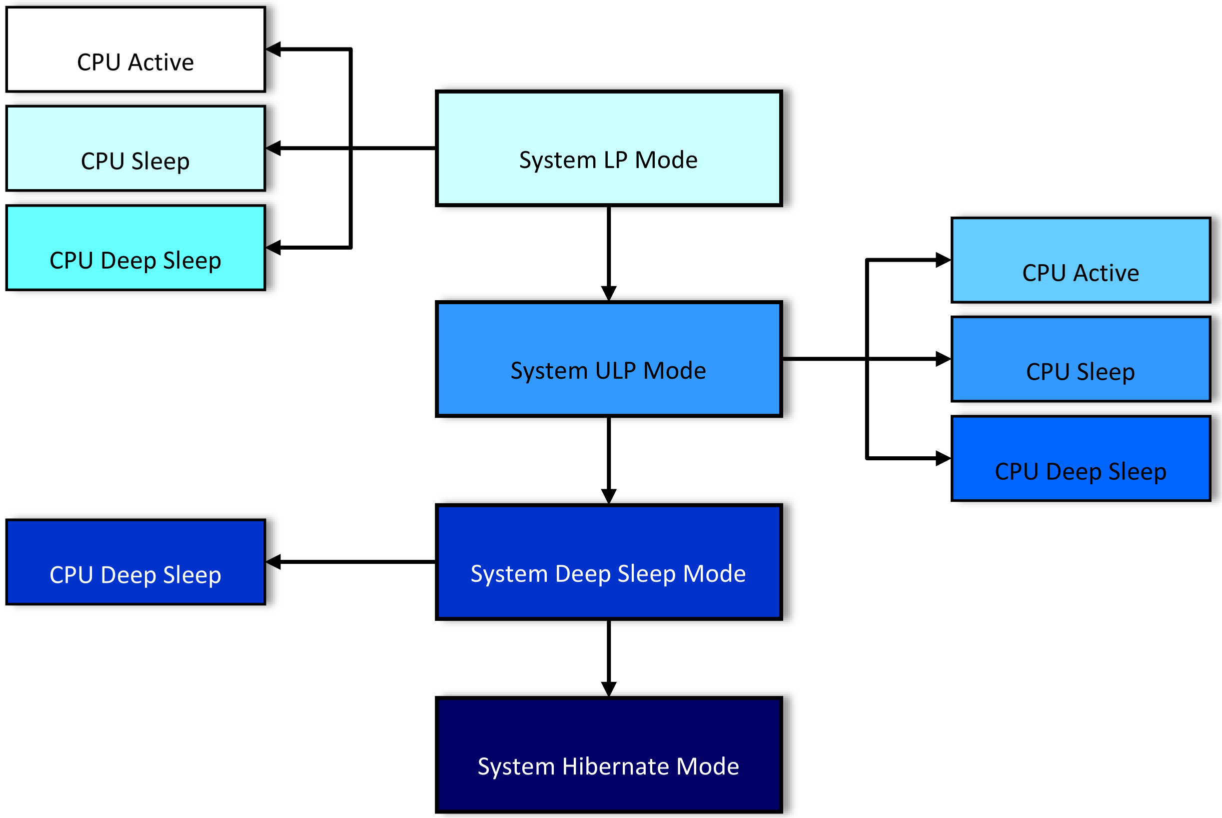
5d.3.1.2 System Power Modes

Mode	Features	Resources Available	Wakeup Sources
System LP	Default mode Max performance Max clock frequencies 1.1 V core voltage	All	Any CPU interrupt
System ULP	Reduced performance Reduced clock frequencies 0.9 V core voltage	HF clock max 50 MHz Peripheral/slow clock max 25 MHz No flash writes allowed	Any CPU interrupt
System Deep Sleep	Requires both CPUs in Deep Sleep LP/ULP regulators off Deep sleep regulators used Buck regulator available Only interrupted CPU wakes up System returns to LP or ULP	HF clocks disabled High speed peripherals disabled Low speed clocks available Low speed peripherals available LPComp, WDT, MCWDT, RTC SRAM can be retained Deep sleep SPI/I2C slave available *	GPIO LPComp SCP CTBm WDT RTC alarm
System Hibernate	LP/ULP regulators off Brown own detection off GPIO states frozen Device resets on wakeup	PWR_HIBERNATE reg retained PWR_HIB_DATA reg retained ILO LPComp, WDT, RTC	Wakeup pins LPComp ** WDT RTC Alarm

* Requires external clocking

** Requires an externally generated compare voltage

The following figure shows which CPU modes are available in each system power mode. Darker colors in the figure indicate relatively lower overall system power.



5d.3.2 Regulators

There are two types of core regulator available - a Linear drop-out (LDO) and a Buck regulator. Each regulator has a normal and a reduced current option available. The buck regulator consumes less power than the LDO but will result in more power supply ripple. See the device datasheet for details on the regulators and their modes.

5d.3.3 Callback Functions

When entering low power modes, it is often necessary to prepare one or more peripherals. Likewise, some action may be required on wake up. These actions can be done by registering callback functions that the PM system calls before any low power event. Some peripherals include a callback function as part of their driver, but you can create custom callback functions as well.

You register a callback by using either the HAL function `cyhal_syspm_register_callback` or the PDL function `Cy_SysPm_RegisterCallback`. Both take pointers to different (but similar) structures.

5d.3.3.1 Using the HAL

First, let's look at how to use the HAL function (`cyhal_syspm_register_callback`). It takes a pointer to a structure of type `cyhal_syspm_callback_data_t` with the following fields:

cyhal_syspm_callback_t	*callback*	Callback to run on power state change.
cyhal_syspm_callback_state_t	*states*	Power states that should trigger calling the callback. Multiple values can be or-ed together.
cyhal_syspm_callback_mode_t	*ignore_modes*	Modes to ignore invoking the callback for. Multiple values can be or-ed together.
*void **	*args*	Argument value to provide to the callback.
*struct cyhal_syspm_callback_data **	*next*	Pointer to the next callback structure. This should be initialized to NULL.

`callback`: The first entry is the name of the callback function.

`states`: The second entry allows you to specify which type of low power transition the function should be called for. The supported values are:

CYHAL_SYSPM_CB_CPU_SLEEP	Flag for MCU sleep callback.
CYHAL_SYSPM_CB_CPU_DEEPSLEEP	Flag for MCU deep sleep callback.
CYHAL_SYSPM_CB_SYSTEM_HIBERNATE	Flag for Hibernate callback.
CYHAL_SYSPM_CB_SYSTEM_NORMAL	Flag for Normal mode callback.
CYHAL_SYSPM_CB_SYSTEM_LOW	Flag for Low power mode callback

`ignore_modes`: By default, the callback function will be called for 4 different. The events are as follows. If you do NOT want the callback to be called for one or more of these events, add them to `ignore_modes`.

CYHAL_SYSPM_CHECK_READY	Callbacks with this mode are executed before entering the low power mode.
CYHAL_SYSPM_CHECK_FAIL	Callbacks with this mode are only executed if the callback returned true for CYHAL_SYSPM_CHECK_READY and a later callback returns false for CYHAL_SYSPM_CHECK_READY. The callback should roll back the actions performed in the previously executed callback with CY_SYSPM_CHECK_READY.

CYHAL_SYSPM_BEFORE_TRANSITION	Callbacks with this mode are executed after the CYHAL_SYSPM_CHECK_READY callbacks' execution returns true. In this mode, the application must perform the actions to be done before entering the low power mode.
CYHAL_SYSPM_AFTER_TRANSITION	In this mode, the application must perform the actions to be done after exiting the low power mode.

`args`: This entry allows you to pass parameters required by the callback function. It may be NULL.

Once the structure and the callback function have been created, you just pass it to `cyhal_syspm_register_callback`. For example:

```
cyhal_syspm_register_callback(&mycallback_structure);
```

For more information, see the System Power Management section of the PDL documentation.

5d.3.3.2 Using the PDL

Now let's look at how to use the PDL function (`Cy_SysPm_RegisterCallback`). Many peripherals make use of the PDL callback registration function (for example CapSense), so it is worth knowing about.

It takes a pointer to a structure of type `cy_stc_syspm_callback_t` with fields that are very similar to the HAL structure:

Cy_SysPmCallback	*callback*	Callback to run on power state change.
cy_en_syspm_callback_type_t	*type*	Power states that should trigger calling the callback. Multiple values can be or-ed together.
unit32_t	*skipMode*	Types to skip invoking the callback for. Multiple values can be or-ed together.
*cy_stc_syspm_callback_params_t **	*callbackParams*	Parameters passed to the callback function.
*struct cy_stc_syspm_callback **	*prevItm*	Previous callback structure in the list.
*struct cy_stc_syspm_callback **	*nextItm*	Next callback structure in the list.
unit8_t	*order*	Order of execution

`callback`: The first entry is the name of the callback function.

`type`: The second entry allows you to specify which `type` of low power mode the function should be called for. The supported values for `type` are:

CY_SYSPM_SLEEP

CY_SYSPM_DEEPSLEEP

CY_SYSPM_HIBERNATE

CY_SYSPM_LP

CY_SYSPM_ULP

`skipMode`: By default, the callback function will be called for 4 different events. The events are as follows. If you do NOT want the callback to be called for one or more of these events, add them to `skipMode`.

CY_SYSPM_SKIP_CHECK_READY	Callbacks with this mode are executed before entering the low power mode. The purpose is to check if the device is ready to enter the low power mode.
CY_SYSPM_SKIP_CHECK_FAIL	Callbacks with this mode are executed after the CY_SYSPM_CHECK_READY callbacks execution returns CY_SYSPM_FAIL. It should roll back the actions performed in the previously executed callback with CY_SYSPM_CHECK_READY.
CY_SYSPM_SKIP_BEFORE_TRANSITION	Callbacks with this mode are executed after the CY_SYSPM_CHECK_READY callbacks execution returns CY_SYSPM_SUCCESS. Performs the actions to be done before entering the low power mode.
CY_SYSPM_SKIP_AFTER_TRANSITION	Performs the actions to be done after exiting the low power mode.

`callbackParams`: This entry allows you to pass parameters required by the callback function. It is a structure containing the base address of a HW instance and context. These may be NULL when a HW instance is not used or where a context is not needed.

*void **	*base*	Base address of a HW instance.
*void **	*context*	Context for the callback function.

`prevItm` and `nextItm`: These are pointers to a previous and next entry, so you can build a linked list of callback structures if an application requires multiple callback functions.

Once we have the structures created, we just call the PM system function to register our callback. For example:

```
/* Register CapSense Deep Sleep event callback */
Cy_SysPm_RegisterCallback(&CapSenseDeepSleep);
```

See the *SysPm Callbacks* section in the SysPm PDL documentation for more information. You will also get a chance to try this in some of the exercises.

5d.4 Hardware Setup and Power Consumption Measurements

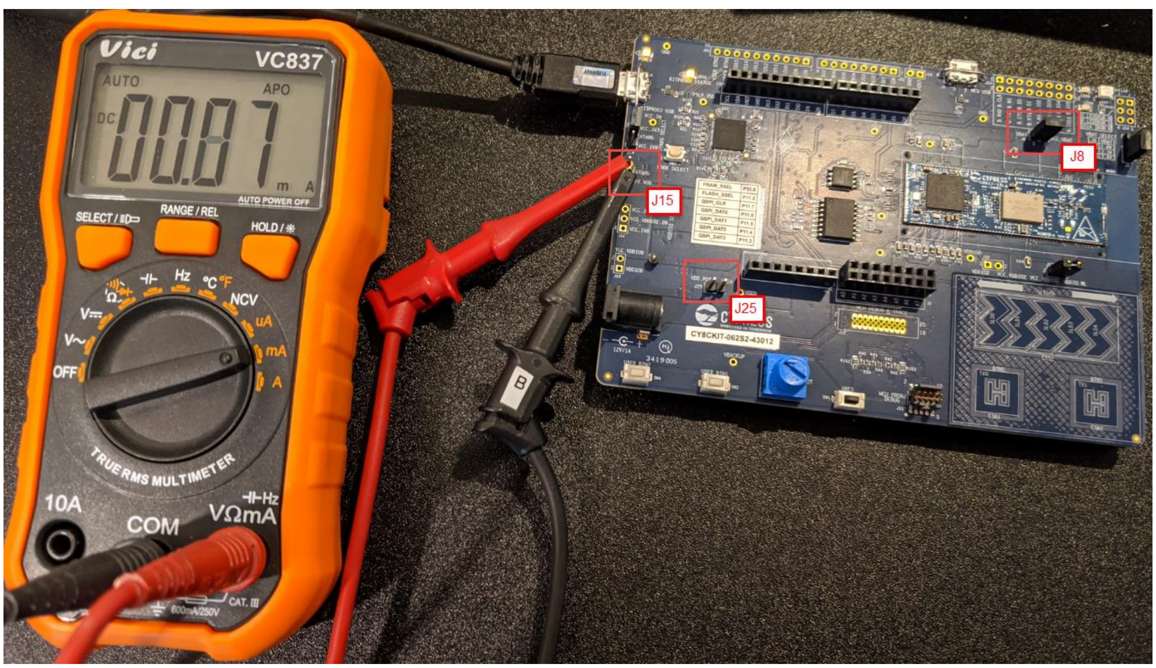
For the exercise in this chapter, we will be using the CY8CKIT-062S2-43012 kit. We will be performing the measurements using a multimeter. If available, it is recommended to use a power analyzer such as a KeySight N6705B. The use of a KeySight N6705B is explained in the low power system design application note referenced earlier.

In order to measure the power consumption on the CY8CKIT-062S2-43012 board configure the multimeter as an ammeter and connect the probes across one of the jumpers depending on which current you want to measure:

Power Rail	Description	Multi-Meter Connection
PSoC 6 Vdd	PSoC 6 Main Power	J15
VBAT	CYW43012 Main Power	J8

You can use USB connection to the KitProg to supply power to the kit, but ensure you disconnect the board from the power supply before connecting and disconnecting the power measurement probes.

Detach the TFT shield from the board as we won't need it for the exercises. Also, remove jumper J25 so that you are not measuring current consumed by the onboard potentiometer.



Note: *Turn on the multimeter before powering the kit via the USB connection to the KitProg - the multimeter path is high impedance when it is turned off.*

Note: *Do not disconnect or turn off the multimeter while the kit is powered.*

5d.5 Low Power for PSoC 6

This section will teach you how to improve the power consumption in projects using the PSoC 6 solution in ModusToolbox. Keep in mind that with ModusToolbox we provide the set of tools that enable you to work in various kinds of environments, IDEs, and RTOS ecosystems. Low power is not an exception but for the exercises we will use the Eclipse IDE and FreeRTOS.

In this part of the training we will select two ModusToolbox code examples and will try to improve the power consumption while preserving the original functionality as much as possible. We will start by making the Hello World example more power efficient, and then we'll do the same on a CapSense code example. We will use non-RTOS versions of the code examples but low power in FreeRTOS will be explained in later exercises. We will use the same CY8CKIT-062S2-43012 board we have used in previous exercises.

5d.5.1 Exercise 1: Making Hello World Energy Efficient

Initial Steps

☐ 1. Disconnect the board from your computer's USB.

☐ 2. Remove the TFT shield and remove jumper J25 to disconnect the potentiometer.

☐ 3. Remove the jumper form J15 and connect an ammeter across the pins to measure the PSoC 6 current consumption.

☐ 4. Turn the ammeter on. Reconnect the board to your computer's USB.

Note: *Use the connection to the ammeter for measuring mA. When making measurements, you may need to switch the setting dial back and forth between µA and mA depending on the current being measured. It is best to start out in the mA setting.*

☐ 1. Start the Eclipse IDE and create new application for the CY8CKIT-062S2-43012 kit.

☐ 2. Select the **Hello World** application and change the name to **ch05d_ex01_hello_lpa**.

☐ 3. Build the application and program the board.

After programming, the application starts automatically. Verify that the application works as expected: LED blinks.

☐ 4. Launch a UART terminal (baud rate = 115200).

a. Connect to the kit and press Enter.
b. Observe the LED stops blinking.
c. Press Enter again and observe the LED resumes blinking.
d. Stop the blinking while the LED is off (so that we can measure the current without the LED's contribution.

☐ 5. Measure current consumption. What is the consumption you see?

________________ (Baseline)

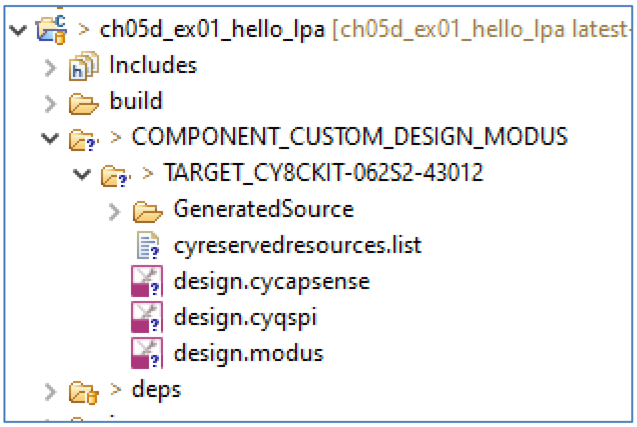
Improve Power Consumption

Now we will change regulator settings and reduce clock speeds to improve power consumption. Note that slowing clocks will not always lead to lower power - sometimes, it is better to speed up clocks so that processing can be done quicker allowing the device more time sleeping.

First, we will create a custom *design.modus* file for our application so that we can make changes without modifying the BSP files directly.

1. Create a new directory in the project's root directory called *COMPONENT_CUSTOM_DESIGN_MODUS* and a sub-directory under that called *TARGET_CY8CKIT-062S2-43012*.

   Copy the entire contents from *libs/TARGET_CY8CKIT-062S2-43012/COMPONENT_BSP_DESIGN_MODUS/* to the new directory that you created.
   The hierarchy should look like this when you finish:



2. Edit the Makefile as follows and save it when you are done:

   Add *CUSTOM_DESIGN_MODUS* to the *COMPONENTS* variable.

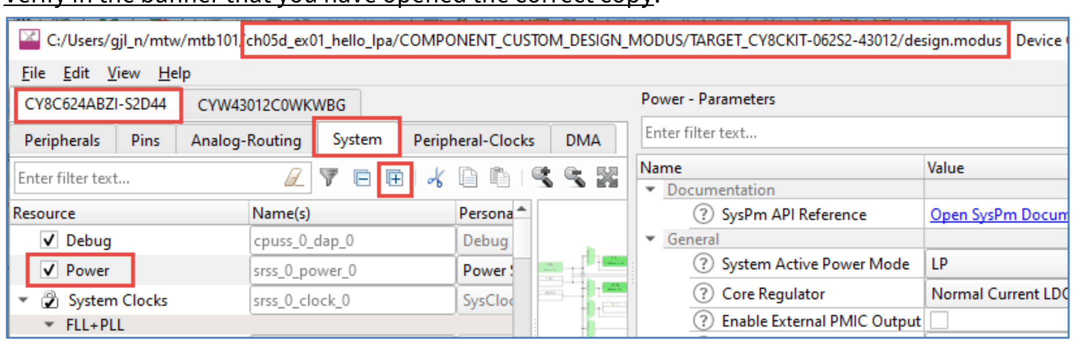
```
COMPONENTS=CUSTOM_DESIGN_MODUS
```

   Add *BSP_DESIGN_MODUS* to the *DISABLE_COMPONENTS* variable.

```
DISABLE_COMPONENTS=BSP_DESIGN_MODUS
```

3. Refresh the quick panel and then click the link to open the device configurator (or double-click on the *design.modus* file in the directory you created).

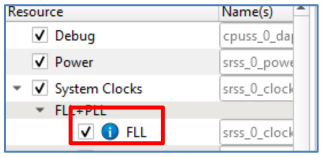
   Verify in the banner that you have opened the correct copy.



4. Select the **CY8C624ABZI-S2D44 > System** tab.

☐ 5. Click the "+" sign to expand all categories and then select the Resource **Power**.

If you see any Resources listed with an exclamation point in a blue circle (such as the FLL shown below in the system clocks section), it might mean the personality being used for that resource is not the latest. You can update to the latest by selecting `File -> Update All Personalities` from the menu.



In the in the **Power – Parameters** pane, change the following parameters:

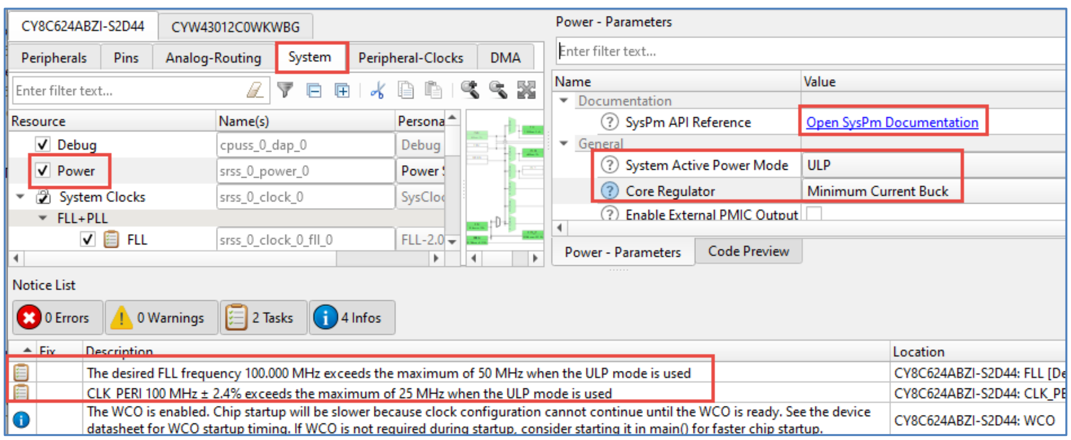
- **System Active Power Mode**: LP -> ULP

  This configures the system for Ultra Low Power. See the [SysPM API documentation] and the device datasheet for the clock frequency and current consumption requirements a system must meet to allow the user of ULP mode.

  You will see messages in the Notice List items that must be changed based on the settings that you selected such as max clock frequencies allowed. We will fix those in a minute.

- **Core Regulator**: Normal Current LDO -> Minimum Current Buck

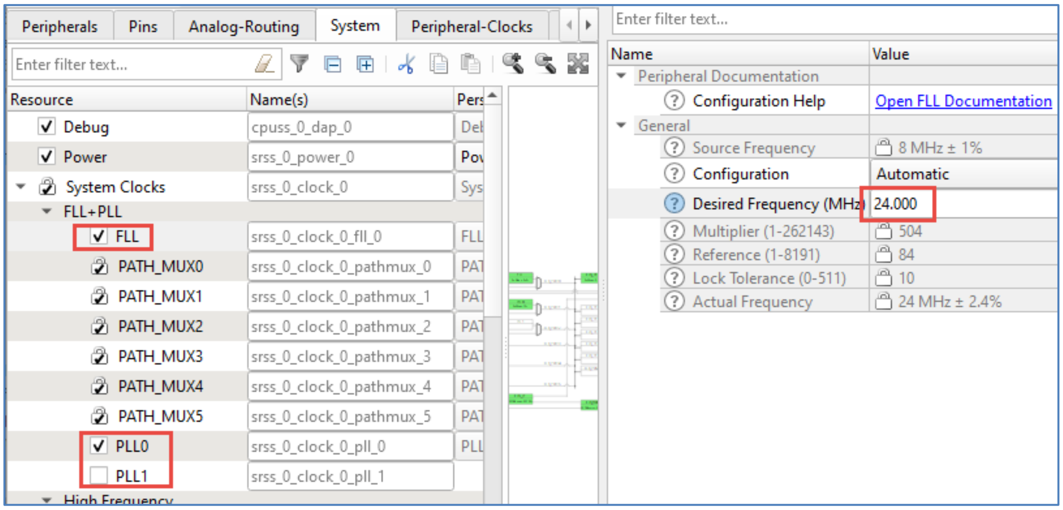
  This configures the core to use the Buck regulator instead of the LDO regulator.



☐ 6. Select the Resource **System Clocks > FLL + PLL**

Note: *You can select a clock to set its parameters either by clicking on it from the list in the Resource pane or by clicking on it in the clock tree diagram. Likewise, you can enable/disable a clock or path by using the checkbox in the resources pane or by double-clicking on it in the clock tree diagram.*

7. For the **FLL**, **PLL0** and **PLL1**, change the Desired Frequency (MHz) to 24 for any of the three that are enabled. The notes at the bottom about invalid frequencies should go away.



8. Click **File > Save** and close the configurator.

9. Build and program the application.

10. Verify that the application works as expected.

11. Measure current consumption again with the LED off. What is the consumption you see?

_______________ (Lower Power Regulators and Slower Clocks)

Disable Unused Resources

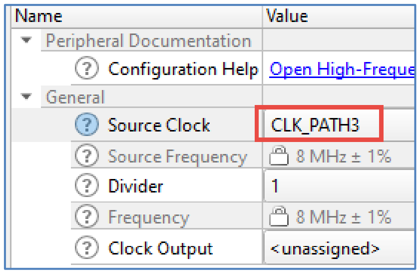
Next, we will disable Unused Resources and further slow the high frequency clock CLK_HF0 by changing its source.

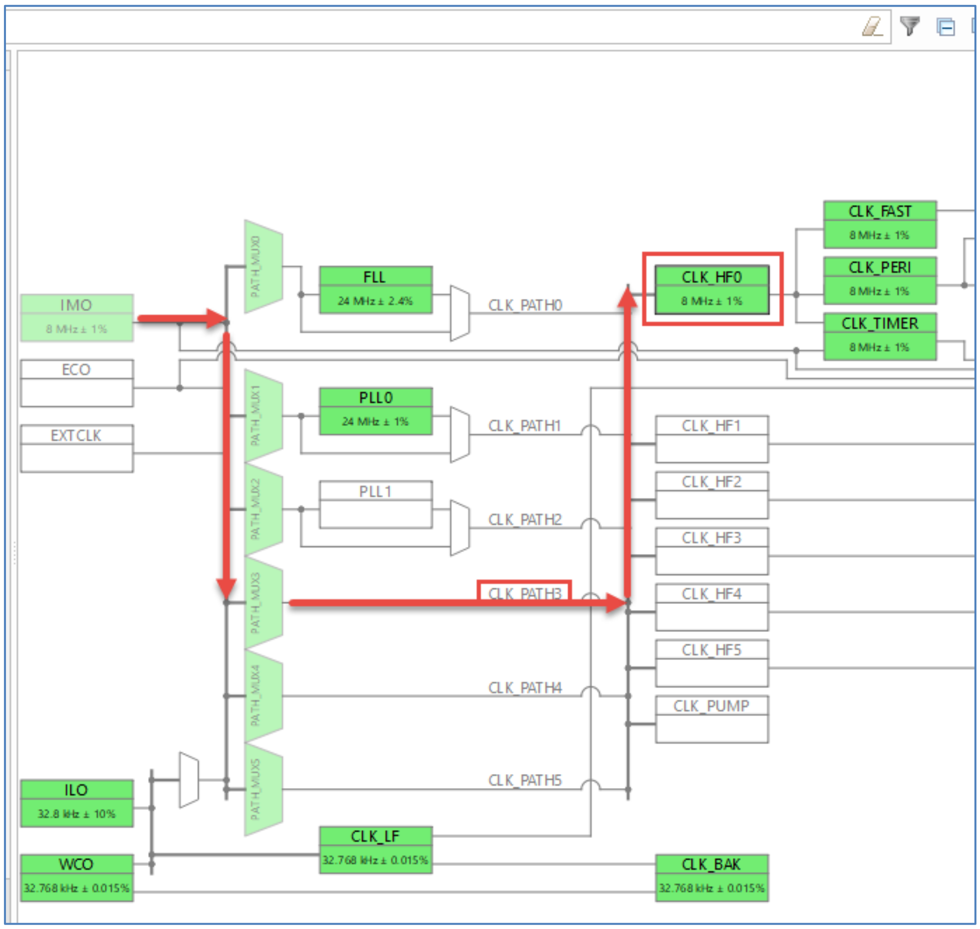
1. Open the device configurator.

2. Navigate to **CY8C624ABZI-S2D44 > System** tab

☐ 3. Expand the Resource **System Clocks** and do the following:

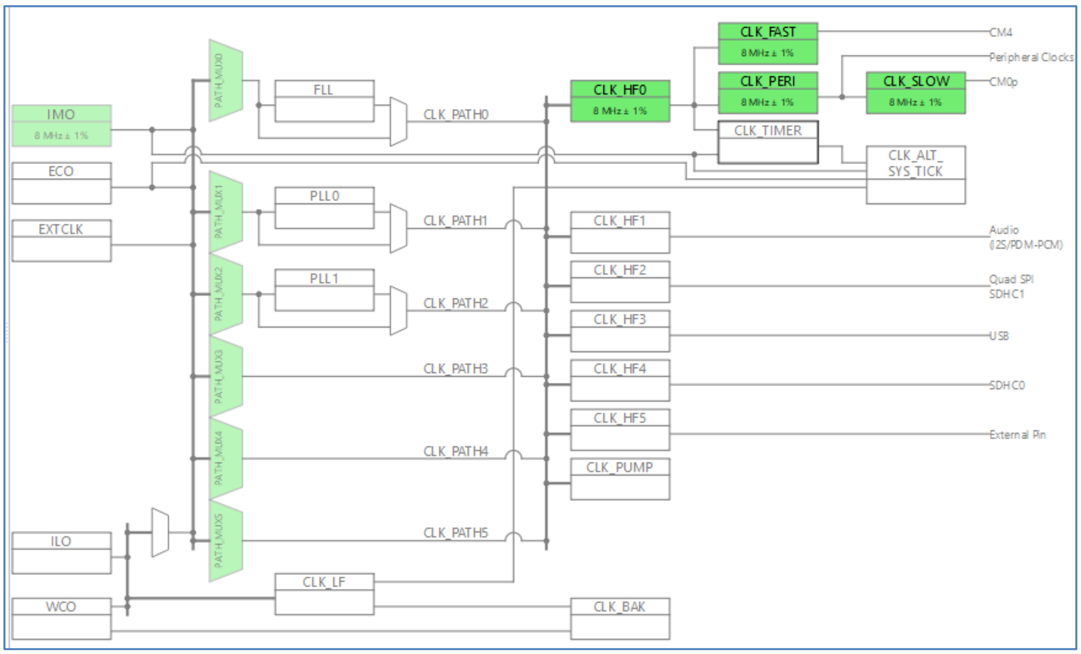
a. Change the source for CLK_HF0 to CLK_PATH3. This selects the IMO (8 MHz) as the clock source instead of the FLL (min 24MHz). You can see the path graphically in the clock tree diagram.



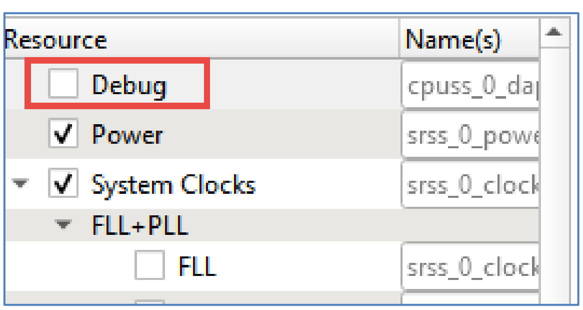


b. Uncheck all listed clocks except these:
- FLL + PLL
  - PATH_MUX0 through PATH_MUX5 (these will be locked)
- High Frequency
  - CLK_FAST (CM4 clock)
  - CLK_HF0 (Source for CLK_FAST)
  - CLK_PERI (Peripheral clock)
  - CLK_SLOW (CM0+ clock)
- Input
  - IMO (Internal Main Oscillator - main clock source - 8 MHz)
- Miscellaneous (all unchecked)

The clock tree diagram will look like this:

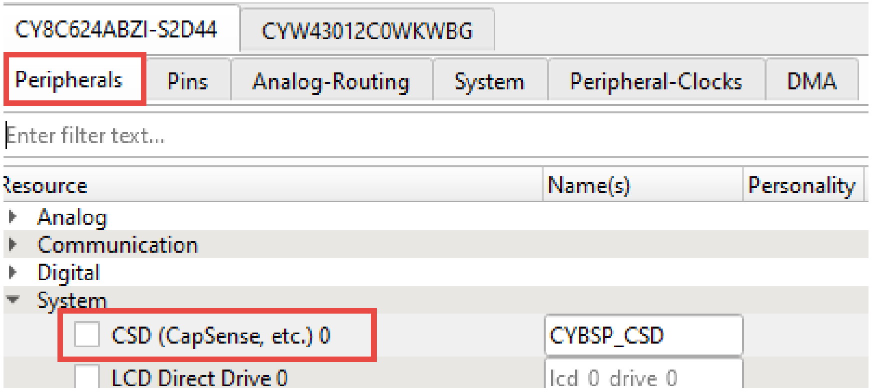


☐ 4. Uncheck (Disable) Resource **Debug.** This disables the ARM debug port.



☐ 5. Navigate to the **Peripherals** tab.

6. Uncheck the Resource **CSD (CapSense, etc.) 0** in the **System** section.



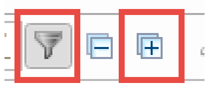
7. Navigate to the **Pins** tab.

8. Uncheck all pins.

   The pins for HAL peripherals such as the LEDs can be accessed via the HAL, so they don't need to be checked in the configurator. This step disables the pins used for Debug, CapSense, and the WCO.

Note: *Click the "+" button to expand all ports.*

Note: *Click the filter button to show only selected pins.*



9. Navigate to the **Peripheral-Clocks** tab.

10. Uncheck Resource **8-bit Divider 0.** This was used by CapSense.

11. Click **File > Save**.

12. In the Makefile, change the value of the `CONFIG` variable from `Debug` to `Release` to change the compiler's optimization settings. Save your changes.

```
CONFIG=Release
```

13. In the Quick Panel under the Launches section, click *Generate Launches for ch05d_ex01_hello_lpa*.

    This step is necessary to update the launch configurations to use the build output from the Release directory.

14. Build and program the application.

15. Verify that the application works as expected.

16. Measure current consumption with the LED off. What is the consumption you see?

    ________________ (Disable Unused Resources)

Enable Sleep

Next, we'll do some code changes to further improve the power. Instead of polling continuously to see if the UART has a new value and to see if the timer has expired, we will put the CPU into sleep mode until the timer expires. When the timer expires, we will:

- Read the UART to see if the user has changed the toggle state
- Toggle the LED (if LED blinking is enabled)
- Go back to sleep

This allows the CPU to be asleep much of the time.

☐ 1. Open the main.c file and add the following global variable:

```
bool uart_command_flag = false;
```

☐ 2. Replace the infinite loop with the following code:

```
    for(;;)
    {
        cyhal_syspm_sleep();
        if(cyhal_uart_getc(&cy_retarget_io_uart_obj, &uart_read_value,
1) \
             == CY_RSLT_SUCCESS)
        {
            if (uart_read_value == '\r')
            {
                led_blink_active_flag ^= 1;
                uart_command_flag = true;
            }
        }

        if(timer_interrupt_flag)
        {
            timer_interrupt_flag = false;
            if (uart_command_flag)
            {
                uart_command_flag = false;
                if (led_blink_active_flag)
                {
                    printf("LED blinking resumed\r\n");
                }
                else
                {
                    printf("LED blinking paused \r\n");
                }
                printf("\x1b[1F");
            }
            if (led_blink_active_flag)
            {
                cyhal_gpio_toggle((cyhal_gpio_t) CYBSP_USER_LED);
            }
        }
    }
```

☐ 3. Build and program the application. Verify that the application works as expected.

4. Measure current consumption with the LED off. What is the consumption you see?

________________ (Utilize Sleep Mode)

Enable Deep Sleep

Next let's add in deep sleep operation. We will need to change the timer that does the blinking to a low power timer (lptimer) and we need to enable the low frequency clock (CLK_LF) that it uses so that the device will have a way to wake back up.

We will also remove the UART because it will not operate in deep sleep mode and since we expect to be in deep sleep most of the time, it won't do us any good.

1. Open the device configurator.
2. Go to the **System** tab and under the Resource **System Clocks** do the following:
   - Enable the ILO
   - Enable CLK_LF
   - Set the CLK_LF source to the ILO
3. Save the configuration and the close the configurator.
4. Open the main.c file and change `cyhal_syspm_sleep` to `cyhal_syspm_deepsleep`.
5. Change the type for the `led_blink_timer` global variable:

```
cyhal_lptimer_t led_blink_timer
```

6. Replace the timer_init function with the following code:

```
#define LPTIMER_MATCH_VALUE (32767)
#define LPTIMER_INTR_PRIORITY (3u)

void timer_init(void)
{
   /* Initialize lptimer. */
   cyhal_lptimer_init(&led_blink_timer );

   /* CLK_LF is 32,768 Hz, so 32,767 counts give us a 1 second interrupt */
   cyhal_lptimer_set_match(&led_blink_timer, LPTIMER_MATCH_VALUE);

   /* Register the interrupt callback handler */
   cyhal_lptimer_register_callback(&led_blink_timer, isr_timer, NULL);

   /* Configure and Enable the LPTIMER events */
   cyhal_lptimer_enable_event(&led_blink_timer,
             CYHAL_LPTIMER_COMPARE_MATCH, LPTIMER_INTR_PRIORITY, true);

   /* Reload/Reset the Low-Power timer to get periodic interrupt. */
   cyhal_lptimer_reload(&led_blink_timer);
}
```

7. Replace the isr_timer function with the following code:

```
static void isr_timer(void *callback_arg, cyhal_lptimer_event_t event)
{
    (void) callback_arg;
    (void) event;

    /* Set the interrupt flag and process it from the main loop */
    timer_interrupt_flag = true;

    /* Reload/Reset the LPTIMER to get periodic interrupt */
    cyhal_lptimer_reload(&led_blink_timer);
}
```

Note: *The second argument's type has changed from `cyhal_timer_event_t` to `cyhal_lptimer_event_t`. Therefore, you will also have to update the function declaration at the top of main.c.*

8. Remove all code associated with the retarget IO library and UART. Specifically:

Include for `cy_retarget_io.h`
Global Variables `uart_read_value` and `uart_command_flag`
Call to `cy_retarget_io_init`
All `printf` statements
If statement and block of code for `if(cyhal_uart_getc)`
If statement and block of code for `if(uart_command_flag)`

You can optionally remove the retarget-io library from the application using the library manager.

9. Build and program the application.

10. Verify that the application works as expected.

11. Measure current consumption when the LED is off. Note that it is not possible to use the UART to stop the LED from blinking, so you will just have to measure the current while it is off. What is the consumption you see?

_______________ (Utilize Deep Sleep Mode)

5d.5.2 Exercise 2: Improving Power Consumption for the CapSense Example

There are certainly more interesting things to do other than a blinking LED, and CapSense is one of them. Let's see how to improve the power consumption for the CapSense code example.

Note that we will reduce the frequency of several clocks in this exercise which will affect the CapSense scan times. In this case, the design has enough margin to still operate properly. In a real application, you may need to re-tune CapSense parameters after changing clocks that affect it.

Create CapSense Design

1. Create the *CapSense Buttons and Slider* Example for the CY8CKIT-062S2-43012 board using the Eclipse IDE. Name the application **ch05d_ex02_capsense_lpa**.
2. Build and Program the board. After programming, the application starts automatically.
3. Verify that the application works as expected by touching the slider, the LED brightness should correspond to the touch position. The buttons can be used to turn the LED on/off.
4. Measure current consumption with the LED off (touch button 1 to turn the LED off) and on (touch button 0 and slide the slider to the right for maximum brightness).

   What is the consumption you see?

   LED off: _________ LED on: _________ (Baseline)

Improve Power Consumption

Now do the following steps to improve power consumption of the application.

1. Start by creating a custom copy of the device configuration from the BSP.
   a. Create a new directory in the project's root directory called *COMPONENT_CUSTOM_DESIGN_MODUS* and a sub-directory under that called *TARGET_CY8CKIT-062S2-43012*.
   b. Copy the entire contents from *libs/TARGET_CY8CKIT-062S2-43012/COMPONENT_BSP_DESIGN_MODUS/* to the new directory that you created.
2. Edit the Makefile as follows and save when you are done:

   Add *CUSTOM_DESIGN_MODUS* to the *COMPONENTS* variable.

   ```
   COMPONENTS=CUSTOM_DESIGN_MODUS
   ```

   Add BSP_DESIGN_MODUS to the *DISABLE_COMPONENTS* variable.

   ```
   DISABLE_COMPONENTS=BSP_DESIGN_MODUS
   ```

3. Refresh the quick panel and then click the link to open the device configurator (or double-click on the *design.modus* file in the directory you created).
4. Select the **CY8C624ABZI-S2D44 > System** tab.
5. Update to the latest Personalities if you see any of the blue exclamation point circles.

6. Select the Resource **Power**, and in the **Power – Parameters** pane change the following parameters:
    - System Active Power Mode: LP -> ULP
    - Core Regulator: Normal Current LDO -> Minimum Current Buck
7. Select the Resource **System Clocks > FLL + PLL**
8. For the **FLL**, **PLL0** and **PLL1**, change the Desired Frequency (MHz) to 24 for any of the three that are enabled.
9. Click **File > Save**. Close the configurator.
10. Build and program the application.
11. Verify that the application works as expected by touching the slider and buttons.
12. Measure current consumption with the LED off and on at maximum brightness. What is the consumption you see?

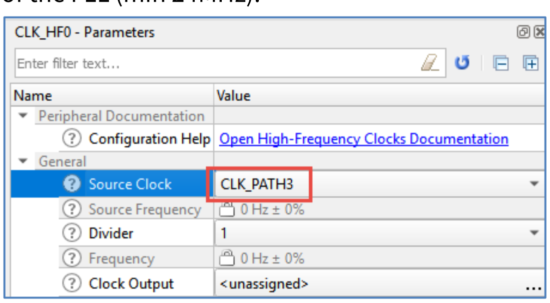
LED off: _________ LED on: _________ (Lower Power Regulators and Slower Clocks)

Disable Unused Chip Resources

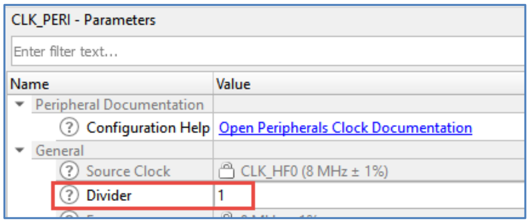
Next let's disable the chip resources we don't use and further slow the high frequency clock CLK_HF0 by changing its source.

1. Open the Device Configurator.
2. Navigate to **CY8C624ABZI-S2D44 > System** tab
3. Expand the Resource **System Clocks**, and do the following:

Change the source for CLK_HF0 to CLK_PATH3. This selects the IMO (8 MHz) as the clock source instead of the FLL (min 24MHz).



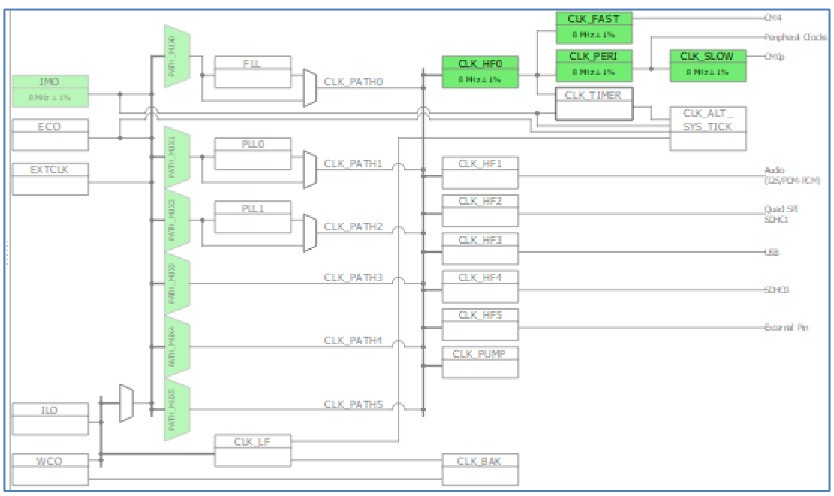
Make sure the divider for CLK_PERI is set to 1. This clock is used for CapSense so we don't want it slowed more than necessary.



Uncheck all listed clocks except these:

- FLL + PLL
  - PATH_MUX0 through PATH_MUX5
- High Frequency
  - CLK_FAST (CM4 clock)
  - CLK_HF0 (Source for CLK_FAST)
  - CLK_PERI (Peripheral clock)
  - CLK_SLOW (CM0+ clock)
- Input
  - IMO (Internal Main Oscillator - main clock source - 8 MHz)
- Miscellaneous (all unchecked)

When you finish, the clock tree should look like this:



4. Uncheck (Disable) Resource **Debug.**

5. Navigate to the **Pins** tab and unselect the Debug pins - P6[4], P6[6] and P6[7] - and the WCO pins - P0[0] and P0[1]. The remaining enabled pins are all used for CapSense.

6. Click **File > Save**. Close the configurator.

7. In the Makefile, change the value of the `CONFIG` variable from `Debug` to `Release`.

8. In the Quick Panel under the Launches section, click *Generate Launches for…*

   This step creates launch configurations that will use the build output from the Release directory.

9. Build and program the application.

10. Verify that the application works as expected by touching the slider and buttons.
11. Measure current consumption with the LED off and on at maximum brightness. What is the consumption you see?

LED off: _________ LED on: _________ (Disable Unused Resources)

Enable Sleep

Finally let's optimize the application code to use sleep mode.

1. Open main.c file and add the highlighted line as follows:

```
/* Initiate next scan */
Cy_CapSense_ScanAllWidgets(&cy_capsense_context);

capsense_scan_complete = false;

cyhal_syspm_sleep();
```

The added line puts the CPU to sleep while a CapSense scan is running. When the scan finishes, it will issue an interrupt to wake the CPU so that it can process the results.

2. Click **File > Save**.
3. Build and program the application.
4. Verify that the application works as expected by touching the slider and buttons.
5. Measure current consumption with the LED off and on at maximum brightness.

What is the consumption you see?

LED off: _________ LED on: _________ (Utilize Sleep Mode)

Note: *The CapSense block does not operate in Deep Sleep. Since the application restarts a CapSense scan as soon as the previous scan completes, using Deep Sleep would not save any power because CapSense is always busy and therefore the application would never enter Deep Sleep.*

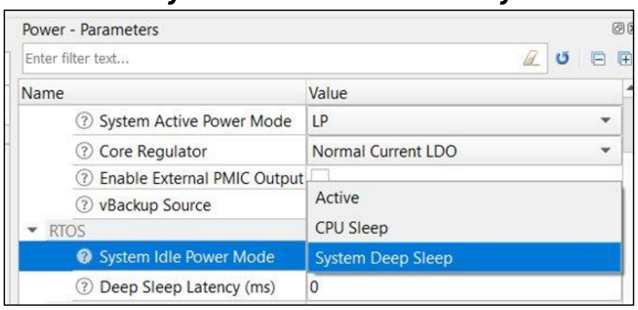
To fix this you would need to add a low power timer to start scans periodically instead of running them continuously. You would also need to use the `Cy_SysPm_RegisterCallback` function to register a callback to the `Cy_CapSense_DeepSleepCallback` function. That function tells the system not to enter sleep unless the CapSense block is not currently performing a scan.

You will see how to do this in the FreeRTOS CapSense example. That example already does a scan every 10ms instead of continuously, so the timer part is already taken care of.

5d.6 Low Power in FreeRTOS

When using Low Power in FreeRTOS, the application must provide a function that knows how to optimize sleep for our hardware and FreeRTOS must be configured for tickles idle mode and to put the CPU into the desired sleep state when it is idle. Both of these are done for you in our FreeRTOS implementation. Specifically:

1. The RTOS abstraction library (*abstraction-rtos*) provides a function called `vApplicationSleep` that understands how to optimize sleep for our hardware. You can override this function with your own if your application needs special sleep handling.
2. The default FreeRTOSConfig.h file looks for a macro called "CY_CFG_PWR_SYS_IDLE_MODE". This is set with the **System > Power > RTOS > System Idle Power Mode** setting in the Device Configurator:



In the template FreeRTOSConfig.h file, the implementation looks like this:

```
/* Check if the ModusToolbox Device Configurator Power personality parameter
 * "System Idle Power Mode" is set to either "CPU Sleep" or "System DeepSleep".
 */
#if defined(CY_CFG_PWR_SYS_IDLE_MODE) && \
    ((CY_CFG_PWR_SYS_IDLE_MODE == CY_CFG_PWR_MODE_SLEEP) || \
     (CY_CFG_PWR_SYS_IDLE_MODE == CY_CFG_PWR_MODE_DEEPSLEEP))

/* Enable low power tickless functionality. The RTOS abstraction library
 * provides the compatible implementation of the vApplicationSleep hook:
 * https://github.com/cypresssemiconductorco/abstraction-rtos#freertos
 * The Low Power Assistant library provides additional portable configuration
 * layer for low-power features supported by the PSoC 6 devices:
 * https://github.com/cypresssemiconductorco/lpa
 */
extern void vApplicationSleep( uint32_t xExpectedIdleTime );
#define portSUPPRESS_TICKS_AND_SLEEP(xIdleTime) vApplicationSleep(xIdleTime)
#define configUSE_TICKLESS_IDLE                 2

#else
#define configUSE_TICKLESS_IDLE                 0
#endif
```

The LP timer provided in the HAL is used to handle timing during sleep instead of any FreeRTOS timers. The timer is configured in the `vApplicationSleep` function, but it is necessary for the application device configuration to have CLK_LF enabled to allow the LP timer to operate.

If your application needs any specific actions associated with low power entry/exit, you can register callback functions that the PM system will call for you. As described earlier, you do this by calling either the HAL function `cyhal_syspm_register_callback` or the PDL function `Cy_SysPm_RegisterCallback`.

5d.6.1 Exercise 3: Improving Power Consumption for a FreeRTOS Application

In addition to the CapSense example that we looked at previously, there is a version of the same example that uses FreeRTOS. Unlike the previous example, instead of running scans continuously, this one uses a FreeRTOS timer to start a new scan every 10 ms. Let's see how power consumption is optimized in this application.

Create CapSense Design

1. Create the *CapSense Buttons and Slider FreeRTOS* Example for the CY8CKIT-062S2-43012 board using the Eclipse IDE. Name the application **ch05d_ex03_capsense_freertos_lpa**.
2. Build and Program the board. After programming, the application starts automatically.
3. Verify that the application works as expected by touching the slider, the LED brightness should correspond to the touch position. The buttons can be used to turn the LED on/off.
4. Measure current consumption with the LED off and on at maximum brightness. What is the consumption you see?

LED off: _________ LED on: _________ (Baseline)

As explained above, by default, FreeRTOS is setup use *System Deep Sleep* when the RTOS is idle.

However, for CapSense to work properly, there must be a callback function that the system can use when it wants to enter Deep Sleep to make sure the CapSense block is not busy and can be shut down properly. You can see the callback function is registered in the *capsense_task.c* file:

```
Cy_SysPm_RegisterCallback(&capsense_deep_sleep_cb);
```

The structure specifies a function that is provided by the CapSense library and configures it to be called on Deep Sleep entry and exit:

```
/* SysPm callback params */
cy_stc_syspm_callback_params_t callback_params =
{
    .base       = CYBSP_CSD_HW,
    .context    = &cy_capsense_context
};
cy_stc_syspm_callback_t capsense_deep_sleep_cb =
{
    Cy_CapSense_DeepSleepCallback,
    CY_SYSPM_DEEPSLEEP,
    (CY_SYSPM_SKIP_CHECK_FAIL | CY_SYSPM_SKIP_BEFORE_TRANSITION |
     CY_SYSPM_SKIP_AFTER_TRANSITION),
    &callback_params,
    NULL,
    NULL
};
```

Improve Power Consumption

Now do the following steps to further improve power consumption of the application.

1. Rather than create a custom configuration from scratch, let's copy over the one from the previous exercise to use as a starting point.

   Copy the entire directory ch05d_ex02_capsense_lpa/COMPONENT_CUSTOM_DESIGN_MODUS to the new application's top directory.

2. Edit the Makefile as follows and save when you are done:

   Add *CUSTOM_DESIGN_MODUS* to the *COMPONENTS* variable (FREERTOS should already be there).

   ```
   COMPONENTS=FREERTOS CUSTOM_DESIGN_MODUS
   ```

   Add BSP_DESIGN_MODUS to the *DISABLE_COMPONENTS* variable.

   ```
   DISABLE_COMPONENTS=BSP_DESIGN_MODUS
   ```

   Change the CONFIG setting to Release.

   ```
   CONFIG=Release
   ```

3. In the Quick Panel under the *Start* section, click *Refresh Quick Panel* so that the Device Configurator will open the correct file.

4. Open the Device Configurator.

5. Go to the **System** tab and under the Resource **System Clocks** do the following:

   - Enable the ILO
   - Enable CLK_LF
   - Set the CLK_LF source to the ILO

   These are necessary for the lp_timer to operate which is what wakes the system from Deep Sleep.

6. Save the configuration and the close the configurator.

7. In the Quick Panel under the *Launches* section, click *Generate Launches for…*

   This step is necessary to create launch configurations that will use the build output from the Release directory.

8. Build and program the application.

9. Measure current consumption with the LED off and on at maximum brightness. What is the consumption you see?

   LED off: _________ LED on: _________ (Lower Power Regulators, Slower Clocks and Disable Unused Resources)

5d.7 Low Power Assistant for Wi-Fi Connected Devices

5d.7.1 Introduction

In this section we will learn how the Low Power Assistant library and functionality helps you to minimize the MCU power consumption in a system connected to Wi-Fi.

As you learned previously, the PSoC 6 communicates with the Wi-Fi device using an SDIO interface. In addition to the SDIO interface, there is a pin dedicated for use in low power. This pin allows the Wi-Fi device to wake the PSoC 6 host. In this way, the PSoC 6 can go to into a low power mode whenever its application allows, and the Wi-Fi device will wake it up only when it is needed to handle specific network activity.

Refer to the LPA Library Guide Part 2 for additional information.

- LPO_IN (32 kHz): 32 kHz sleep clock used for low-power WLAN operation
- SDIO: Clock, Data
- WL_HOST_WAKE: Interrupt line to wake the Host to service Wi-Fi request

Once the wake pin is configured and the LPA library is available, you can offload various Wi-Fi operations such as responding to certain packet types so that they can either be handled by the Wi-Fi device or locally.

5d.7.2 Exercise 4: Basic Low-Power for Wi-Fi Applications

Rather that start from scratch, we'll start with the WLAN Low Power code example to see how basic low power host wake is configured. In later exercises, we'll add various offloads so that the Wi-Fi device can handle more operations without the host MCU being involved.

Since this is an AnyCloud FreeRTOS application, all the setup required for low power in FreeRTOS as described in section 0 is already done.

☐

1. Create the *AnyCloud_WLAN_Low_Power* example for the CY8CKIT-062S2-43012 board using the Eclipse IDE. Name the application **ch05d_ex04_wlan_low_power**.

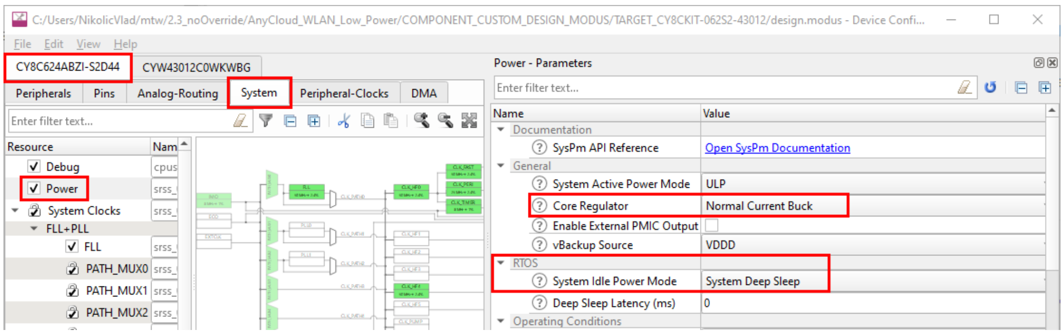
   Once the application has been created, notice that it has a folder called COMPONENT_CUSTOM_DESIGN_MODUS. Inside that folder is a folder for each TARGET supported by the CE. The Makefile has already been set up to use this custom configuration.

☐ 2. Open the custom design.modus file for your kit.

Notice that the Power settings for the PSoC 6 are configured such that the RTOS will use System Deep Sleep when it is idle. The required changes in the FreeRTOSConfig.h file to enable System Deep Sleep have already been done.

Also note that the Normal Current Buck regulator is used for this example.

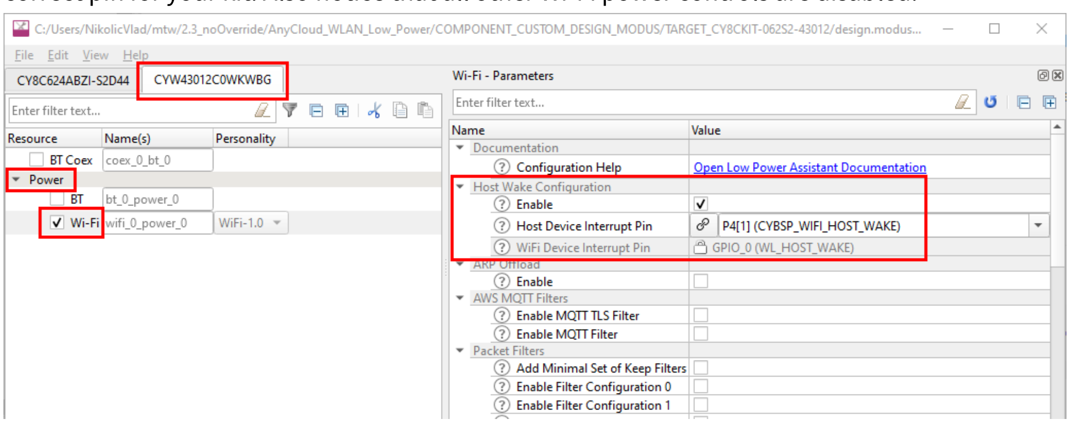
If you don't see "Normal Current Buck" you will need to update the personality. This is also indicated by a blue circle next to the resource name and info messages in the Notice List. To update the personalities, chose **File > Update All Personalities**.



☐ 3. Next, switch to the **CYW43012C0WKWBG** tab. This has settings for the WLAN device.

☐ 4. Click on **Power > Wi-Fi** to see the settings.

Notice that the Host Wake Configuration is enabled, and the Host Device Interrupt Pin is set to the correct pin for your kit. Also notice that all other Wi-Fi power controls are disabled.



With these settings, the WLAN device will not offload any tasks or filter any packets - it will wake the host CPU for any Wi-Fi activity. However, the PSoC 6 does configure the WLAN device for low power operation and suspends the network after periods of inactivity.

☐ 5. Open the file `lowpower_task.c` and look at the `lowpower_task` function.

Note that `lowpower_task` calls `wlan_powersave_handler` to configure the WLAN device for power save without throughput, power save with throughput, or power save disabled. It calls the appropriate `whd_wifi_*_powersave` functions to acomplish this. These functions are provided by the WHD library.

Also note that the `lowpower_task` will suspend the network when it has been inactive for a specified period of time by calling `wait_net_suspend`. This function is part of the LPA library.

6. Open the file `lowpower_task.h`. Update the **WIFI_SSID** and **WIFI_PASSWORD** to match the Wi-Fi network being used for the class. Depending on your Wi-Fi router, you may also need to update **WIFI_SECURITY**.

Note: See the cy_wcm_security_t structure in cy_wcm.h for possible security options - you can find it by right-clicking on CY_WCM_SECURITY_WPA2_MIXED_PSK and choosing "Open Declaration".

```
#define WIFI_SSID                   "SSID"
#define WIFI_PASSWORD               "PASSWORD"

#define WIFI_SECURITY               CY_WCM_SECURITY_WPA2_MIXED_PSK
```

7. Open a UART terminal and then Build and Program the board.
8. Verify in the UART terminal that it successfully connects to Wi-Fi.

   You will see that the Wi-Fi device will wake the host MCU to service many different broadcast and multicast packets issued by the AP such as ARP (Address Resolution Protocol) requests. The frequency and number of events will depend on how heavy network traffic is.

9. Open a command terminal and ping the kit to verify that it responds. Check the UART terminal to see that the host wakes up to service the ping event.

   The IP address for your kit is displayed in the terminal window just after the kit connects to the Wi-Fi AP.

In the next exercises we will configure different offloads to further improve the time for which the host can remain in Deep Sleep.

5d.7.3 Exercise 5: Configure Packet Filter Offload

Packet filters allow the host processor to limit which types of packets get passed up to the host from the WLAN subsystem. This is useful to keep out unwanted / unneeded packets from the network that might otherwise wake the host out of a power-saving System Deep Sleep mode or prevent it from entering System Deep Sleep mode.

Packet filters are useful when:

- Trying to keep the host processor in System Deep Sleep for as long as possible.
- Trying to get the host processor into System Deep Sleep as soon as possible.

Whenever a WLAN packet is destined for the host, the WLAN processor must wake the host (if it is asleep) so it can retrieve the packet for processing. Often the host network stack processes the packet only to discover that the packet should be thrown away, because it is not needed. For example, it is destined for a port or service that is not being used. Packet filters allow these types of packets to be filtered and discarded by the WLAN processor, so the host is not woken.

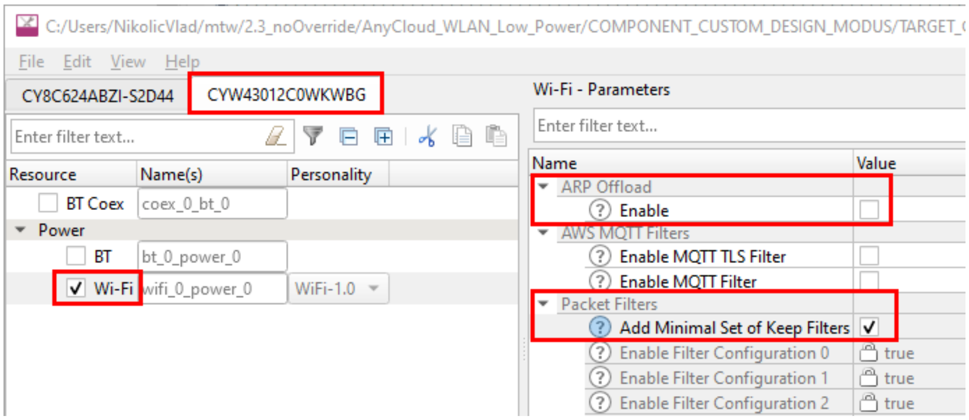
Refer to the Packet Filters section of the LPA documentation for additional information.

You can either edit the previous exercise, or if you want separate copies you can create a new application from the previous exercise (use the **Import** button in the Project Creator to select the previous application). If you choose to do that, call the new application **ch05d_ex05_packet_filter**.

1. Open the Device Configurator. Verify in the banner that you have the custom file for your application opened.
2. Switch to the connectivity device tab.

   Enable **Add minimal set of keep filters**. Keep **ARP Offload** disabled for now.

   These filters allow ARP, DNS, DHCP and 802.11x security packets to wake up the Host. These are needed to connect to a Wi-Fi Access point. Any other Wi-Fi packets will be dropped by the WLAN chip and not forwarded to the Host MCU (PSoC 6).



3. Save the configuration to generate the necessary code.
4. Open a UART terminal to see messages from the board.
5. Build the project and program. Observe the IP address assigned to your kit.
6. Open a command terminal and send a "ping" command to the board.

   Observe the UART terminal or observe the power consumption to see that it does not wake up the PSoC 6 device since there is no "keep" packet filter for ICMP pings. Also observe that there is no response for the pings.

7. Send an "arping" command and observe that:

   - You get the responses.
   - Observe the UART terminal or observe the power consumption to see that the PSoC 6 MCU wakes up to service the ARP ping.

     Linux: arping is built-in

     macOS: you will need to install arping using brew. See:

     https://brewinstall.org/install-arping-on-mac-with-brew/

     Windows: you will need to download a tool to send arping requests. One such tool can be found here:

     https://elifulkerson.com/projects/arp-ping.php

     To use the Windows tool, download it onto your computer. Open a command terminal and go to the directory containing the downloaded file. Run the command:

     ```
     .\arp-ping.exe <ip address>
     ```

     In some cases, the arp-ping may respond with data from a cached ARP table on the PC instead of sending the request to the network. If that is the case, you can delete the table from the PC before each ARP ping by using the -d option, but this requires that you run the command terminal as administrator:

     ```
     .\arp-ping.exe -d <ip address>
     ```

     You can see what is in the cache by running:

     ```
     arp -a
     ```

5d.7.4 ARP (Address Resolution Protocol) Offload

Recall that Wi-Fi devices will send broadcast ARP requests to all devices. When a device hears an ARP request for its IP address, it must respond with its MAC address - this is how devices figure out how to send packets through the network to the correct place. Devices often monitor and cache ARP responses for other devices that they hear on the network so that they can build up an ARP table of IP to MAC address that exist on the network. That way, they don't need to send ARP requests as often.

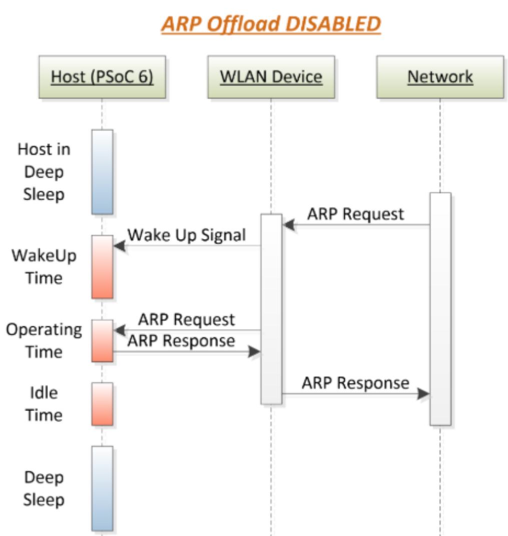
ARP traffic is normally forwarded from the WLAN device to the host MCU. If the Host is sleeping, the Device wakes it up. Having the Device handle some of the ARP traffic will reduce the frequency that the Host sleeps/wakes up, reducing Host power consumption by staying in CPU Sleep and System Deep Sleep states longer. By enabling ARP offload, the WLAN device will not wake the host for all ARP requests. Rather, it will send the response itself whenever an ARP request is addressed to it.

Likewise, when the host wants to send an ARP request, the WLAN device can respond directly without sending the request to the network if the WLAN device has the requested address cached.

The ARP offload features can be found in the LPA library documentation here: ARP Offload features. The main features are summarized below.

5d.7.4.1 Disabled

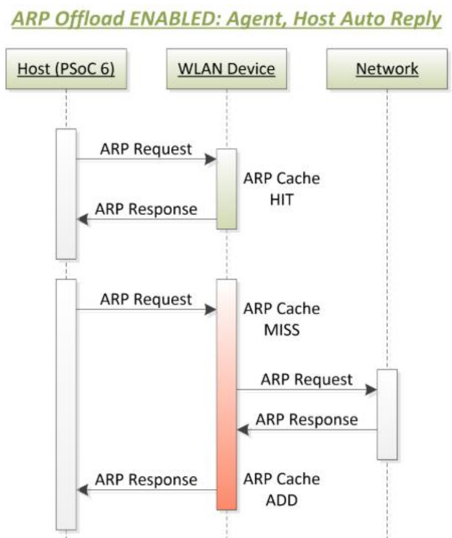
With ARP offload disabled, all ARP requests from the network will wake the host:



In addition, all ARP requests from the host will be sent out to the network by the WLAN device (this case is not shown).

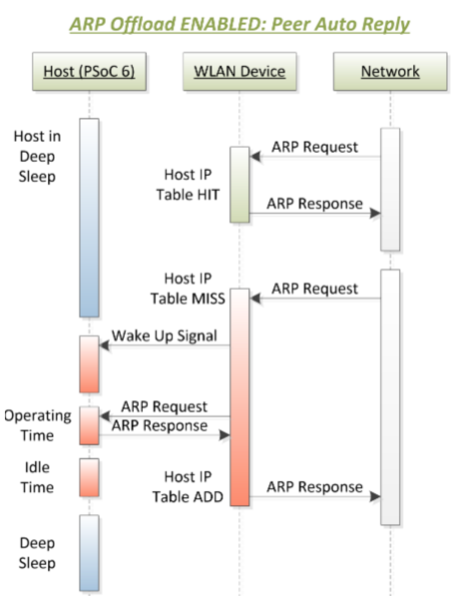
5d.7.4.2 Host Auto Reply

Host Auto Reply is a power-saving and network traffic reduction feature. Using the ARP Offload Host Auto Reply feature, the WLAN device will answer ARP requests from the host MCU without broadcasting the request to the Network if it has the requested information in its cache:



5d.7.4.3 Peer Auto Reply

Peer Auto Reply targets requests in the other direction. That is, the WLAN device will respond to ARP requests from Peers (i.e. from the network) without waking up the Host Processor:



5d.7.4.4 Host IP Snoop

When enabled, the Snoop facility watches for ARP responses from the host to the network, and caches them in the WLAN Device Host IP Table. The size of this table is 8 entries, which allows for the Device to support multiple IP addresses.

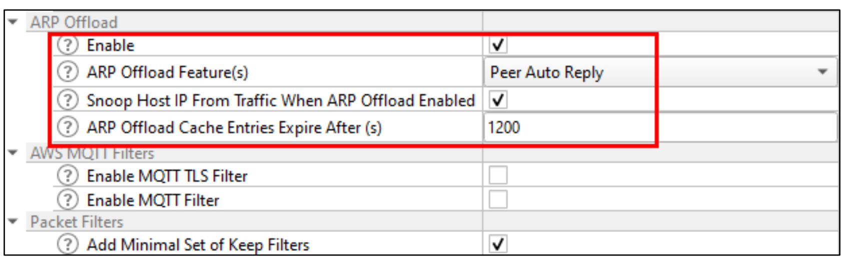
5d.7.4.5 ARP Offload Cache Entry Expiry

This is an "age out" value that you can set in the ARP Offload configuration to determine the length of time the ARP cache entry is valid. This ensures that the WLAN ARP cache is updated appropriately. Now let's add ARP Offload to our previous exercise.

5d.7.5 Exercise 6: ARP Offload

You can either edit the previous exercise, or if you want separate copies you can create a new application from the previous exercise (run the Project Creator, select your kit, and then and use the **Import** button on the application selection screen to select the previous application). If you choose to do that, call the new application **ch05d_ex06_ARP_offload.**

1. Open the Device Configurator. Verify in the banner that the correct file is opened.
2. Switch to the connectivity device tab.
   - Enable **ARP offload**.
   - Set **ARP offload Feature(s)** to "Peer Auto Reply".
   - Enable **Snoop Host IP From Traffic When ARP Offload Enabled**.
   - Set **ARP Offload Cache Entries Expire after (s)** to "1200".
   - Keep Packet Filters as configured.
   - Save your changes.



3. Build the project and program.
4. Send a "ping" command to the board and observe no change in behavior.

   In the UART terminal, you will see that it does not wake up the PSoC 6 device since there is no "keep" packet filter for ICMP pings. Observe the power consumption to see that the PSoC 6 MCU remains in Deep Sleep mode.
5. Send an "arping" command.
   - Observe that you still get the responses.
   - Observe the UART terminal and power consumption to see that the PSoC 6 MCU no longer wakes up.

   The PSoC 6 (Host) is offloaded from ARP packet handling and can remain in sleep.

5d.7.6 Exercise 7: Allow ICMP (Ping) Packets Through the Filter

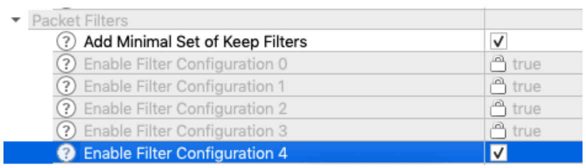
This explains how to modify Packet filters so that ICMP Ping packets are not filtered by the WLAN and are sent to the host.

You can either edit the previous exercise, or if you want separate copies you can create a new application from the previous exercise (use the Import button in the Project Creator to select the previous application). If you choose to do that, call the new application **ch05d_ex07_allow_ping.**

1. Open the Device Configurator. Verify in the banner that you have the correct file opened.
2. Switch to the connectivity device tab and select Resource **Power > Wi-Fi**

   Check the box for **Enable Filter Configuration 4**.

Note: *Filter configurations 0-3 are locked because they are used for the minimal set of keep filters that you previously added.*



3. Scroll down and set the configuration for filter 4 as follows.

   Refer to the LPA documentation for details:
   Packet Filters Quick Start Guide
   Filter Types
   Filter Type set to "IP Type" (IP Protocol Filter)
   IP Protocol set to "1' (1 refers to ICMP packet : List_of_IP_protocol_numbers)

4. Save your changes, build the project and program.
5. Send a "ping" command to the board and observe the UART.

   It now wakes the Host up because you have just added a "keep" packet filter for ICMP pings. You can see the response in the terminal and can observe increased power consumption upon receiving the ping packet.

6. Send an "arping" command.

   This behavior did not change; you get the responses to ARP in the PC terminal window while the PSoC 6 MCU remains in sleep.

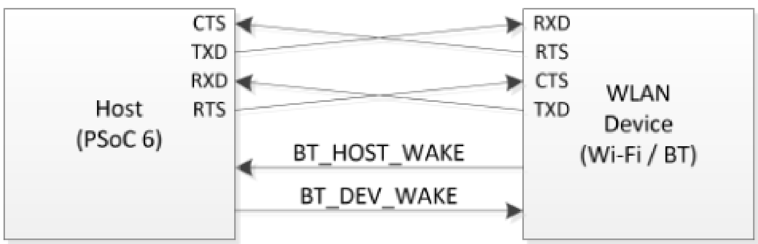
5d.8 Low Power Assistant for Bluetooth Connected Devices

5d.8.1 Introduction

In this section we will learn how the Low Power Assistant helps you to minimize the MCU power consumption in a system using Bluetooth.

As you learned previously, the PSoC 6 communicates with the Bluetooth portion of the connectivity device using a UART interface. In addition to the UART interface, there are two pins dedicated for use in low power. These pins allow the Bluetooth LE 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.

Refer to the LPA Library Guide Part 3 for additional information.



- UART: (CTS / TXD / RXD / RTS)
- BT_HOST_WAKE (host wake): MCU input pin which can wake the MCU with interrupt.
- BT_DEV_WAKE (device wake): an output MCU host pin which is connected as input Bluetooth device pin which interrupts the Bluetooth device when set in active state.

5d.8.2 Bluetooth Low Power Configuration Structure

The Bluetooth configuration has a structure of type cybt_platform_config_t. It is typically defined in the BSP. That structure looks like this:

```
typedef struct
{
    cybt_hci_transport_config_t    hci_config;
    cybt_controller_config_t       controller_config;
    uint32_t                       task_mem_pool_size;
} cybt_platform_config_t;
```

The controller_config structure inside that looks like this:

```
typedef struct
{
    cyhal_gpio_t                    bt_power_pin;
    cybt_controller_sleep_config_t  sleep_mode;
} cybt_controller_config_t;
```

Finally, the sleep_mode structure inside there has the sleep mode configuration settings:

```
typedef struct
{
    uint8_t                 sleep_mode_enabled;
    cyhal_gpio_t            device_wakeup_pin;
    cyhal_gpio_t            host_wakeup_pin;
    uint8_t                 device_wake_polarity;
    uint8_t                 host_wake_polarity;
} cybt_controller_sleep_config_t;
```

These settings include one value to enable or disable the sleep mode, settings for the Host Wake and Dev Wake pins, and settings to configure the polarity for the Host Wake and Dev wake pins.

The BSP typically uses the settings from the **Power > BT** section in the LPA:

```
.sleep_mode                          =
{
    .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
}
```

Usually the BSP enables low power mode by default and there is no reason to disable – after all, who wants to waste power?

On the PSoC 6 side of things, the pins are initialized by the bluetooth-freertos library, so you don't need to worry about them.