Chapter 5: 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 and text commands	`CY_ISR_PROTO(MyISR);` `make build`
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 GUI commands, menu paths and selections, and icon names in procedures	Click the **Debugger** icon, and then click **Next**.

5.1 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 connectivity device, 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. In the case of a connected solution, the devices may be flooded with packets sent to and from hundreds of Wi-Fi hotspots and Bluetooth® devices around them, 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. Save more energy and you win.

ModusToolbox™ software provides tools and middleware to set up and use the low power features of the PSoC™ 4/PSoC™ 6 MCUs and the connectivity devices on your board. We are referring to this set of tools and middleware as the Low Power Assistant. In this class we will focus just on low power on the PSoC device, but the same concepts will be used in later classes that cover Bluetooth® and Wi-Fi. The Low Power Assistant consists of:

- Device Configurator: Used to configure the peripherals and system resources of the PSoC™ 4/PSoC™ 6 MCUs, 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: (PSoC™ 6 devices only) This consumes the configuration generated by the Device Configurator, then configures and handles Wi-Fi packet filters, offloads, host wake functionality, etc.

Note: *The Low Power Assistant middleware is only used by the Wi-Fi firmware. It does not need to be included for applications that are MCU only or for Bluetooth®.*

The Low Power Assistant feature is supported for all PSoC™ 6 MCUs as well as for the CYW43012 and CYW4343W connectivity modules. The combination of PSoC™ 6 and CYW43012 offers the lowest power consumption and is available on the CY8CKIT-062S2-43012.

5.2 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.

5.2.1 Application note

A few excellent sources of information are the low power system design application notes. Reading them after this chapter will help you to refresh and reinforce the knowledge you have just gained.

- AN86233 - PSoC™ 4 Low-Power Modes and Power Reduction Techniques
- AN219528 - PSoC™ 6 MCU Low-Power Modes and Power Reduction Techniques
- AN227910 - Low-Power System Design with CYW43012 and PSoC™ 6 MCU

5.2.2 Low Power Assistant documentation

The LPA middleware 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.

Note: *The LPA library documentation contains information and examples for MCU, Bluetooth® and Wi-Fi even though the library itself is only needed for Wi-Fi.*

5.3 Overview

5.3.1 PSoC™ 4 Power modes

In PSoC™ 4 devices, power modes are enabled by an IP block called the System Resources Sub-system (SRSS). There are two versions of this IP block that are used in PSoC™ 4 devices, the SRSS and the SRSS-Lite. Each of these blocks enables a different set of power modes:

SRSS Power Modes	SRSS-Lite Power Modes
• Active • Sleep • Deep Sleep • Hibernate • Stop	• Active • Sleep • Deep Sleep

The Active, Sleep, and Deep Sleep power modes are the same between the two blocks.

The PSoC™ 4 MCU on the CY8CKIT-149 uses the SRSS-Lite IP block. To determine what SRSS block other PSoC™ 4 devices use, refer to the appropriate device documentation.

5.3.1.1 PSoC™ 4 Power modes

Mode	CPU State	Resources Available	Wakeup Sources
Active	CPU executing code	All peripherals available All clocks on	N/A
Sleep	CPU Wait for Interrupt/ Wait for Event (WFI/WFE)	All peripherals available All clocks on	Any peripheral interrupt
Deep Sleep	CPU WFI/WFE	Low-frequency peripherals available High frequency clocks off (IMO, ECO, PLLs)	Available peripheral interrupts
Hibernate	CPU off	Asynchronous peripherals available All clocks off	Available peripheral interrupts

Mode	CPU State	Resources Available	Wakeup Sources
Stop	CPU off	GPIO states frozen, all other peripherals disabled All clocks off	Dedicated wakeup pin

5.3.2 PSoC™ 6 Power modes

PSoC™ 6 devices have 3 CPU power modes which are applicable to both cores. They also have 4 system power modes. A summary of each of the 7 modes is shown in the tables below.

5.3.2.1 PSoC™ 6 CPU Power modes

Mode	CPU State	Resources Available	Wakeup Sources
Active	CPU executing code	All peripherals available CPU clock on	N/A
Sleep	CPU WFI/WFE (Wait for Interrupt/ Wait for Event)	All peripherals available CPU clock off	Any peripheral interrupt
Deep Sleep	CPU WFI/WFE Request System Deep Sleep mode	All peripherals available CPU clock off	Any peripheral interrupt

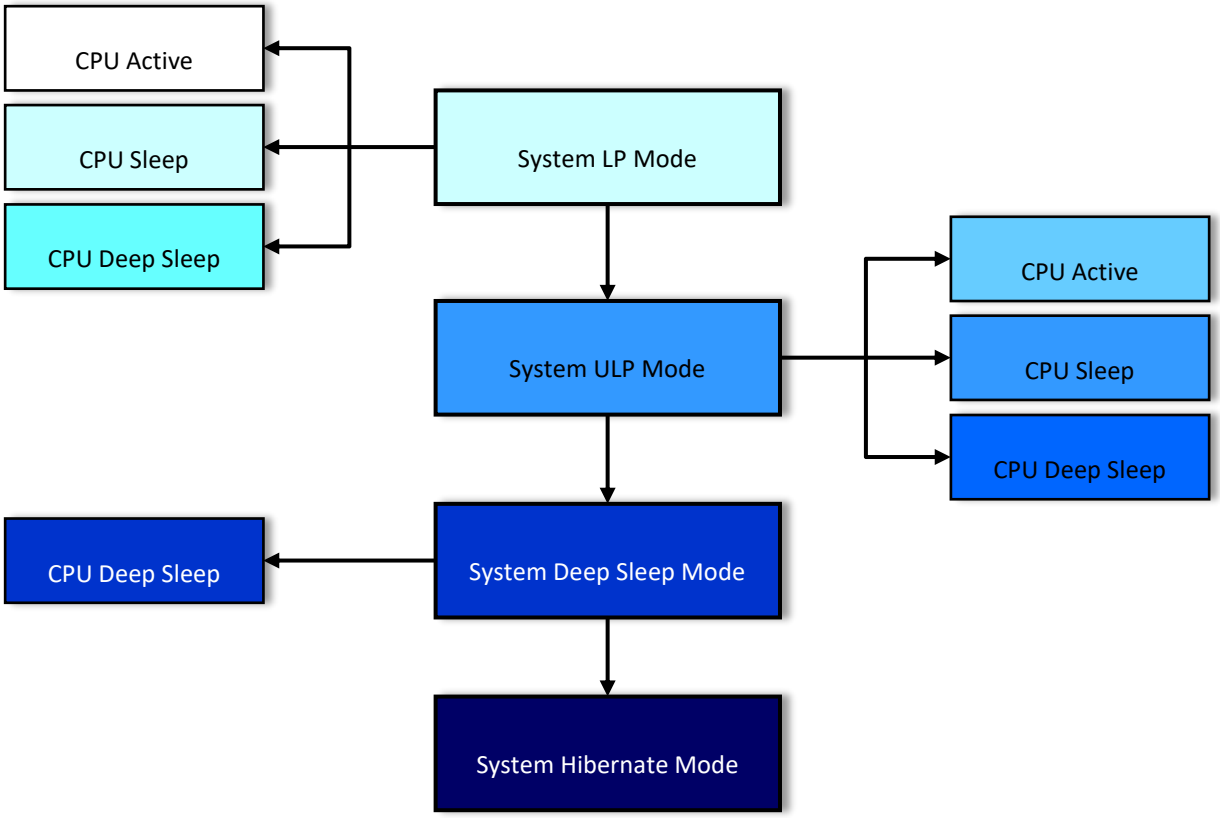
5.3.2.2 PSoC™ 6 System power modes

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 out 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 PSoC™ 6 CPU power modes are available in each system power mode. Darker colors in the figure indicate relatively lower overall system power consumption.



5.3.3 PSoC™ 6 Regulators

On PSoC™ 6 devices 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.

5.3.4 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.

5.3.4.1 Using the HAL

The HAL currently does not support the full set of PSoC™ 4 power modes, only the modes enabled by the SRSS-Lite IP block.

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.	PSoC 4™/PSoC™ 6 devices
CYHAL_SYSPM_CB_CPU_DEEPSLEEP	Flag for MCU deep sleep callback.	PSoC 4™/PSoC™ 6 devices
CYHAL_SYSPM_CB_SYSTEM_HIBERNATE	Flag for Hibernate callback.	PSoC™ 6 devices
CYHAL_SYSPM_CB_SYSTEM_NORMAL	Flag for Normal mode callback.	PSoC™ 6 devices
CYHAL_SYSPM_CB_SYSTEM_LOW	Flag for Low power mode callback	PSoC™ 6 devices

`ignore_modes`: 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 `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.

5.3.4.2 Using the PDL

The CAT2 PDL currently does not support the full set of PSoC™ 4 power modes, only the modes enabled by the SRSS-Lite IP block.

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™ touch sensing), so it is worth knowing about.

Note: The PSoC™ 4 (CAT2) and PSoC™ 6 (CAT1) PDL both define this function in the same way.

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	CAT1/CAT2
CY_SYSPM_DEEPSLEEP	CAT1/CAT2
CY_SYSPM_HIBERNATE	CAT1 Only
CY_SYSPM_LP	CAT1 Only
CY_SYSPM_ULP	CAT1 Only

`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 System PM 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.

5.4 Hardware setup and power consumption measurements

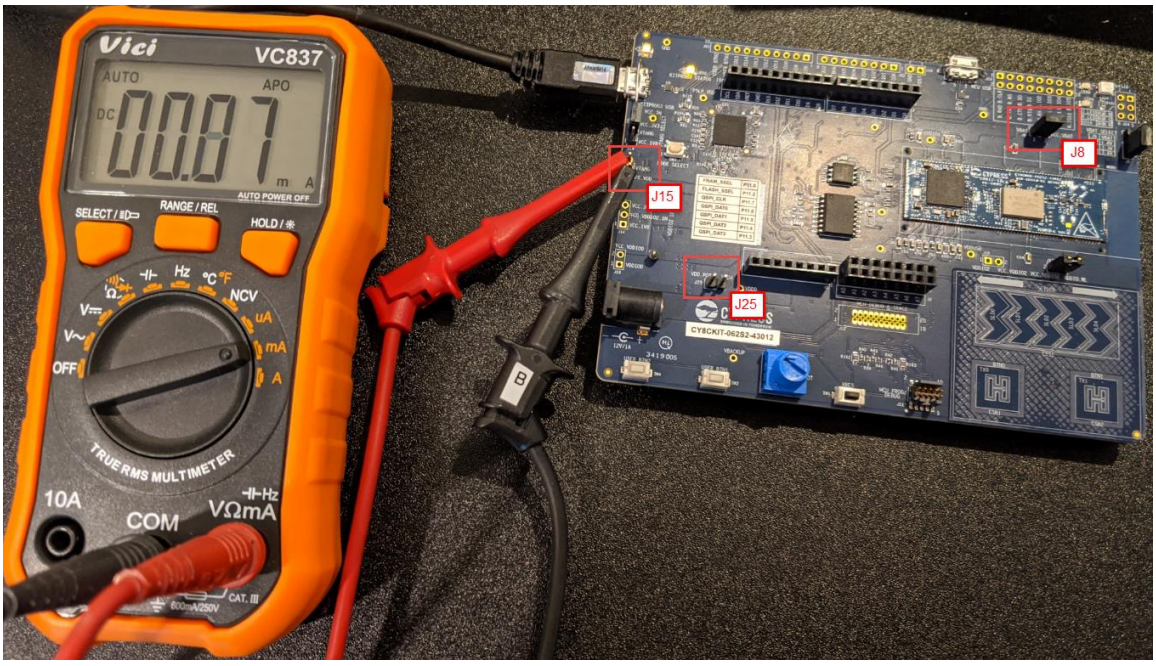
For the exercises in this chapter, we will be using the CY8CKIT-062S2-43012 kit. We will be performing the measurements using a multimeter. If available, it is recommended to use a power analyzer such as a KeySight N6705B. The use of a KeySight N6705B is explained in the low power system design application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

5.5 Low power for PSoC™ 6 devices

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 the ModusToolbox™ solution, we provide a 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 for ModusToolbox™ and FreeRTOS.

In this part of the training we will select two ModusToolbox™ code examples and will 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 to a CAPSENSE™ code example. We will use non-RTOS versions of the code examples but low power in FreeRTOS will be explained later.

Note: *When using the Device Configurator, remember that the files it uses are typically part of the BSP so be aware that any edits you make may result in a dirty git repo for the BSP if you are using a standard BSP. Refer to the ModusToolbox™ Software Training Level 1 - Getting Started class for details on how to create a custom BSP or override the configuration from the BSP within a single application.*

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 jumper 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 since that typically places a smaller shunt resistor in series with the power supply.*

5. Start the Eclipse IDE and create new application for the CY8CKIT-062S2-43012 kit.
6. Select the **Hello World** application and change the name to **ch05_ex01_hello_lpa**.
7. Build the application and program the board.

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

8. 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.

9. Measure current consumption when the LED is off. What is the consumption you see?

_____ (Baseline)

Improve Power Consumption

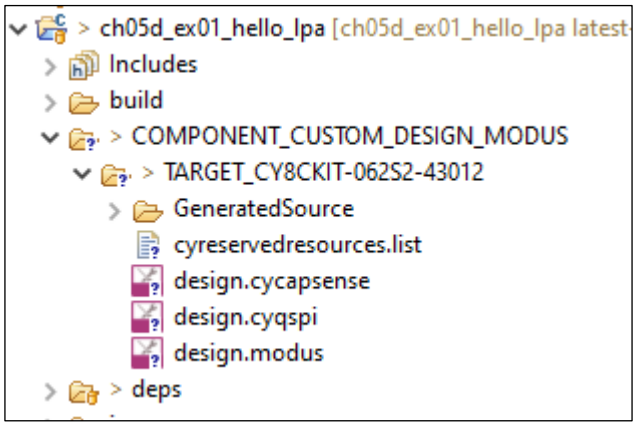
Now we will change regulator settings and reduce clock speeds to improve power consumption. Note that slowing clocks will not always lead to lower power - sometimes, it is better to speed up clocks so that processing can be done quicker allowing the device to spend more time sleeping.

First, we will create a custom *design.modus* file for our application so that we can make changes without modifying the BSP files directly. This is done to prevent dirtying the BSP repo, which would prevent you from getting updated versions of the BSP in the future.

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*.

   a. Copy the entire contents from *libs/TARGET_CY8CKIT-062S2-43012/COMPONENT_BSP_DESIGN_MODUS/* to the new directory that you created.

   b. The hierarchy should look like this when you finish:



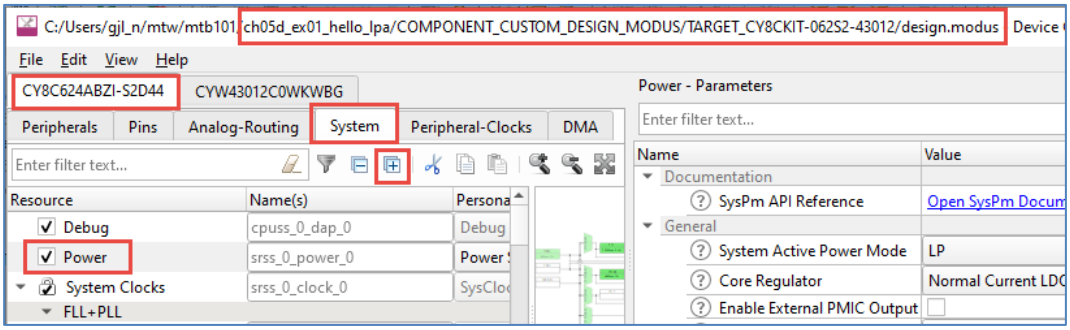
2. Edit the Makefile as follows and save it when you are done:

   a. Add `CUSTOM_DESIGN_MODUS` to the `COMPONENTS` variable:
   `COMPONENTS=CUSTOM_DESIGN_MODUS`

   b. 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:

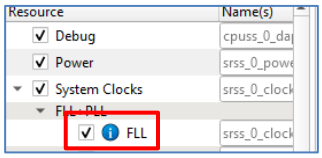


4. Select the **CY8C624ABZI-S2D44 > System** tab.

5. Click the **+** button 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.

a. You can update to the latest by selecting **File > Update All Personalities** from the menu.



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

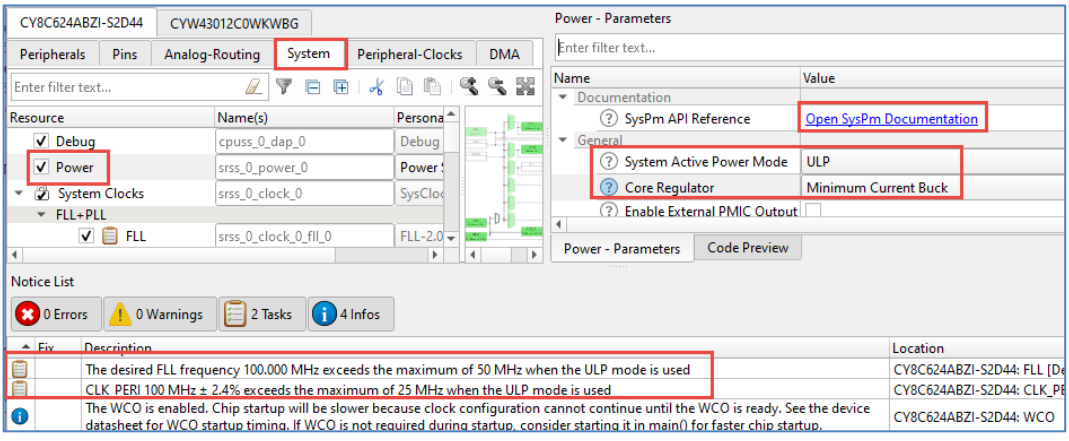
- **System Active Power Mode**: LP -> ULP

  This configures the system for Ultra Low Power. See the SysPM API reference 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)

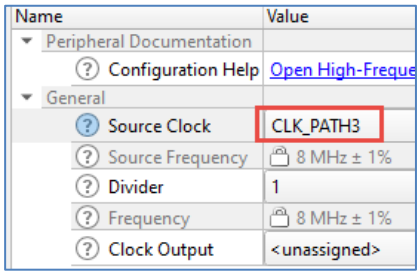
Disable Unused Resources

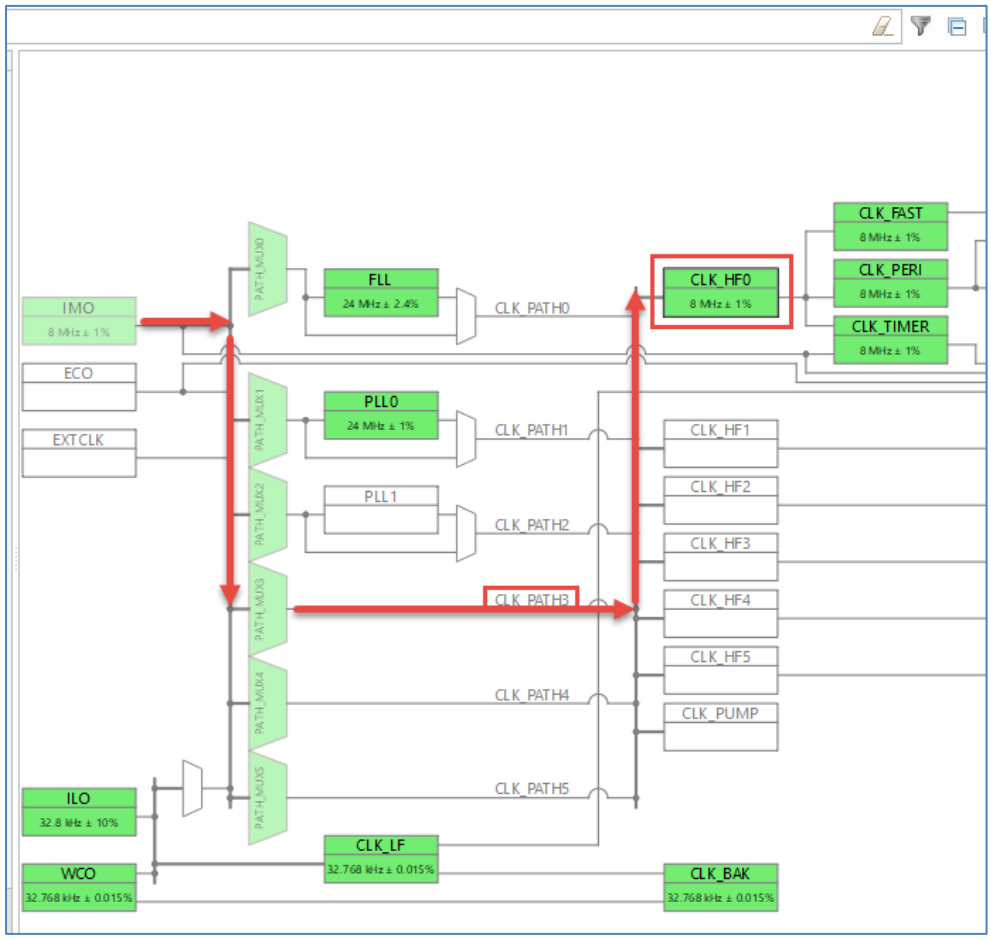
Next, we will disable unused resources and further slow the high frequency clock CLK_HF0 by changing its source.

1. Open the device configurator.
2. Navigate to **CY8C624ABZI-S2D44 > System** tab

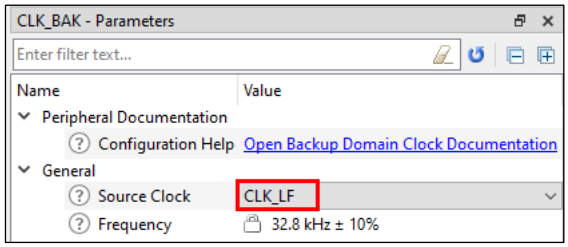
3. Expand the Resource **System Clocks** and do the following:

   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.

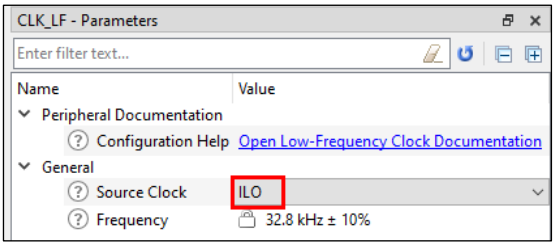




   b. Change the source for CLK_BAK to CLK_LF.



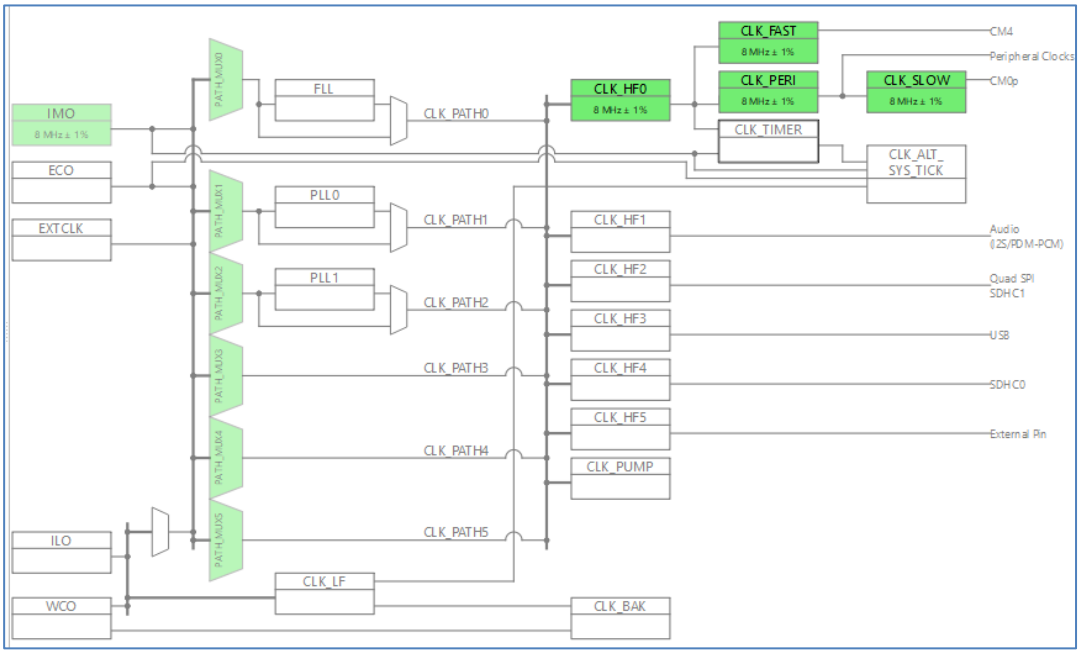
c. Change the source for CLK_LF to ILO.



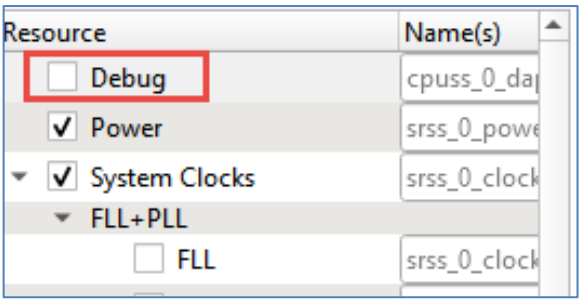
d. Uncheck all listed clocks except these:
   - FLL + PLL
     - PATH_MUX0 through PATH_MUX5 (these will be locked)
   - High Frequency
     - CLK_FAST (CM4 clock)
     - CLK_HF0 (Source for CLK_FAST)
     - CLK_PERI (Peripheral clock)
     - CLK_SLOW (CM0+ clock)
   - Input
     - IMO (Internal Main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

Miscellaneous (all unchecked)

The clock tree diagram will look like this:

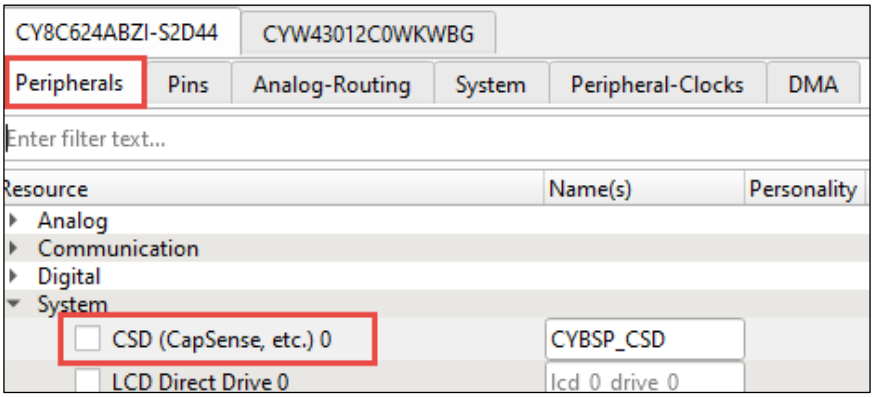


4. Uncheck (Disable) Resource **Debug.** This disables the ARM debug port.



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 to make it easier to de-select them.*

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 ch05_ex01_hello_lpa*.

Note: *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: *Note that 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)

Exercise 2: Improving power consumption for CAPSENSE™

There are certainly more interesting things to do other than blinking an LED, and touch sensing 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 **ch05_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:
   a. Add `CUSTOM_DESIGN_MODUS` to the `COMPONENTS` variable.
      `COMPONENTS=CUSTOM_DESIGN_MODUS`
   b. *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?

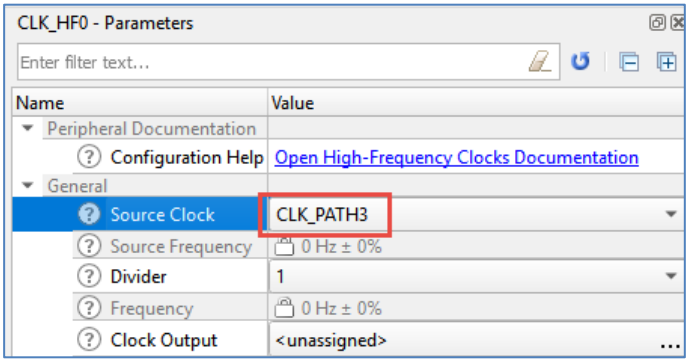
    LED off: _____ LED on: _____ (Lower Power Regulators and Slower Clocks)

Disable Unused Chip Resources

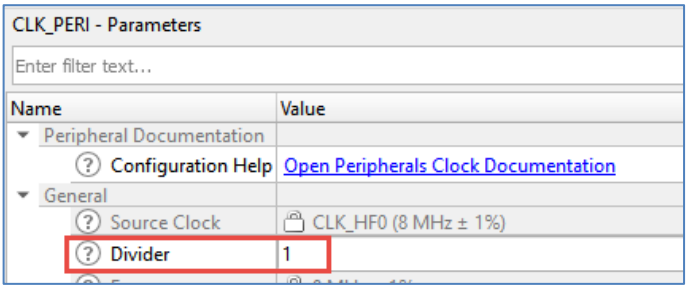
Next let's disable the chip resources we don't use and further slow the high frequency clock CLK_HF0 by changing its source.

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

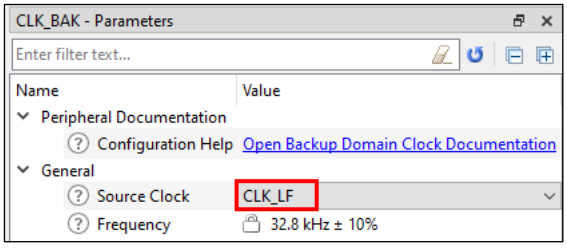
   Change the source for CLK_HF0 to CLK_PATH3. This selects the IMO (8 MHz) as the clock source instead of the FLL (min 24MHz).



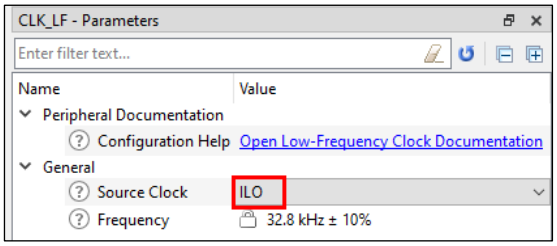
   Make sure the divider for CLK_PERI is set to 1. This clock is used for the CAPSENSE™ HW so we don't want it slowed more than necessary.



   Change the source for CLK_BAK to CLK_LF.



Change the source for CLK_LF to ILO.

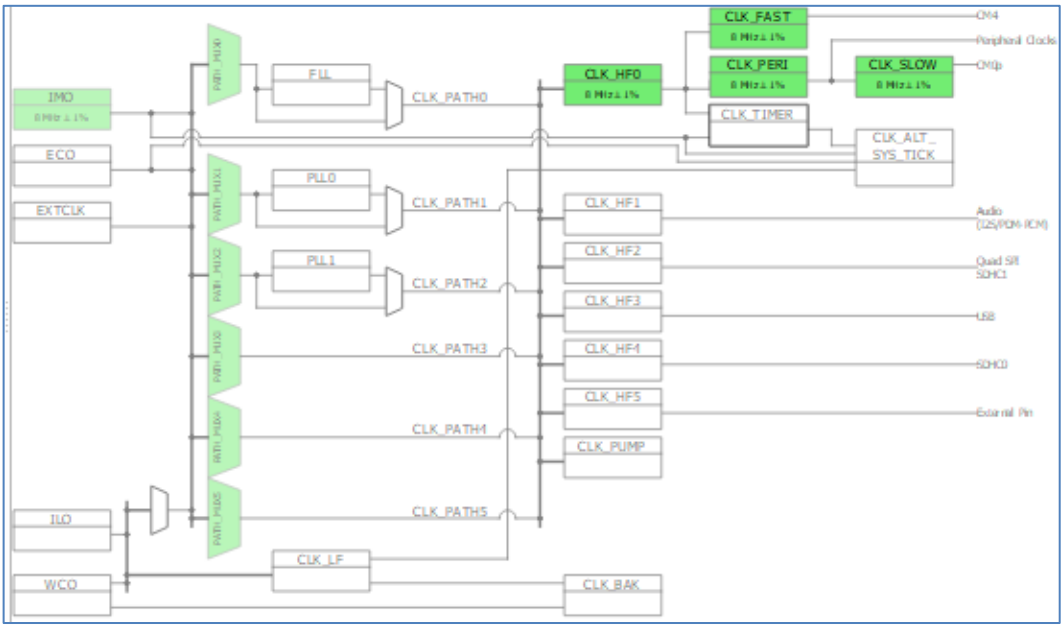


Uncheck all listed clocks except these:

- 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…*

Note: *This step is necessary to create 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 your *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™ HW 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 use Deep Sleep effectively, 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.*

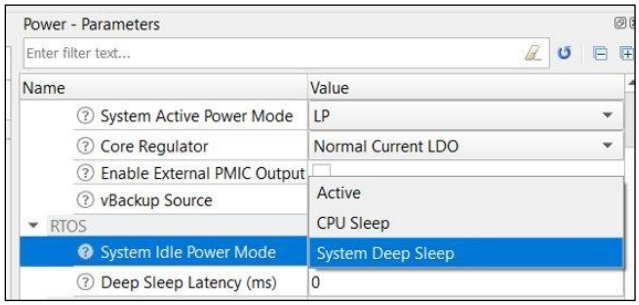
You will see how to do this in the FreeRTOS CAPSENSE™ example. That example already does a scan every 10ms instead of continuously, so the timer part is already taken care of.

5.6 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/infineon/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/infineon/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`.

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.

5.6.1.1 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 **ch05_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 touch sensing 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™ HW 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
};
```

5.6.1.2 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 *ch05_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:

   a. Add `CUSTOM_DESIGN_MODUS` to the `COMPONENTS` variable (`FREERTOS` and `RTOS_AWARE` should already be there – if not, add them too).

   ```
   COMPONENTS=FREERTOS RTOS_AWARE CUSTOM_DESIGN_MODUS
   ```

   b. Add `BSP_DESIGN_MODUS` to the `DISABLE_COMPONENTS` variable.

   ```
   DISABLE_COMPONENTS=BSP_DESIGN_MODUS
   ```

   c. 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…*

Note: *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)

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Published by

Infineon Technologies AG

81726 Munich, Germany

IMPORTANT NOTICE

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics (“Beschaffenheitsgarantie”) .

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer’s compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer’s products and any use of the product of Infineon Technologies in customer’s applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer’s technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (**www.infineon.com**).

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies’ products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.