


# PSoC® Creator™

# System Reference Guide

**cy_boot Component v4.11**

**Document Number: 001-94480, Rev. **

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone (USA): 800.858.1810
Phone (Intl): 408.943.2600
http://www.cypress.com

© Cypress Semiconductor Corporation, 2014. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life-saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC® is a registered trademark, and PSoC Creator™ and Programmable System-on-Chip™ are trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.

# Contents




**1 Introduction ..... 5**
- Conventions ..... 6
- References ..... 6
- Sample Firmware Source Code ..... 6
- Revision History ..... 6

**2 Standard Types and Defines ..... 7**
- Base Types ..... 7
- Hardware Register Types ..... 7
- Compiler Defines ..... 7
- Keil 8051 Compatibility Defines ..... 8
- Return Codes ..... 8
- Interrupt Types and Macros ..... 9
- Intrinsic Defines ..... 9
- Device Version Defines ..... 9
- Variable Attributes ..... 9
- PSoC Creator Generated Defines ..... 10

**3 Clocking ..... 13**
- PSoC Creator Clocking Implementation ..... 13
- APIs ..... 24

**4 Power Management ..... 37**
- PSoC 3/PSoC 5LP Implementation ..... 37
- PSoC 4 Implementation ..... 46
- Instance Low Power APIs ..... 49

**5 Interrupts ..... 51**
- APIs ..... 51

**6 Pins ..... 55**
- PSoC 3/PSoC 5LP APIs ..... 55
- PSoC 4 APIs ..... 57

**7 Register Access ..... 59**
- APIs ..... 59

**8 DMA ..... 63**

**9 Flash and EEPROM ..... 65**
PSoC 3/PSoC 5LP Implementation ..... 65
PSoC 4 Implementation ..... 73

**10 Bootloader Migration ..... 77**
Introduction ..... 77
Migrating Bootloader Designs ..... 78
Migrating Bootloadable Designs ..... 79
Project Application Type Property Changes ..... 79
Migrating Bootloadable Dependency to Bootloader Design ..... 79

**11 System Functions ..... 81**
General APIs ..... 81
CyDelay APIs ..... 82
PSoC 3/PSoC 5LP Voltage Detect APIs ..... 83
PSoC 4100 and 4200 Voltage Detect APIs ..... 87
Cache Functionality ..... 88

**12 Startup and Linking ..... 89**
PSoC 3 ..... 90
PSoC 4/PSoC 5LP ..... 90
Preservation of Reset Status ..... 93

**13 Watchdog Timer ..... 95**
PSoC 3/PSoC 5LP ..... 95
PSoC 4100 and PSoC 4200 ..... 96
PSoC 4000 ..... 103

**14 MISRA Compliance ..... 109**
Verification Environment ..... 109
Project Deviations ..... 110
Documentation Related Rules ..... 111
PSoC Creator Generated Sources Deviations ..... 112
cy_boot Component-Specific Deviations ..... 113

**15 cy_boot Component Changes ..... 115**
Version 4.11 ..... 115
Version 4.10 ..... 115
Version 4.0 ..... 116
Version 3.40 and Older ..... 117
Version 2.40 and Older ..... 120

# 1 Introduction




This System Reference Guide describes functions supplied by the PSoC Creator cy_boot component. The cy_boot component provides the system functionality for a project to give better access to chip resources. The functions are not part of the component libraries but may be used by them. You can use the function calls to reliably perform needed chip functions.

The cy_boot component is unique:

- Included automatically into every project
- Only a single instance can be present
- No symbol representation
- Not present in the Component Catalog (by default)

As the system component, cy_boot includes various pieces of library functionality. This guide is organized by these functions:

- DMA (unavailable for PSoC 4)
- Flash
- Clocking
- Power management
- Startup code
- Various library functions
- Linker scripts

The cy_boot component presents an API that enables user firmware to accomplish the tasks described in this guide. There are multiple major functional areas that are described separately.

## Conventions

The following table lists the conventions used throughout this guide:

| Convention | Usage |
|---|---|
| `Courier New` | Displays file locations and source code:<br>`C:\ …cd\icc\`, user entered text |
| *Italics* | Displays file names and reference documentation:<br>*sourcefile.hex* |
| [**bracketed, bold**] | Displays keyboard commands in procedures:<br>[**Enter**] or [**Ctrl**] [**C**] |
| **File > New Project** | Represents menu paths:<br>**File > New Project > Clone** |
| **Bold** | Displays commands, menu paths and selections, and icon names in procedures:<br>Click the **Debugger** icon, and then click **Next**. |
| Text in gray boxes | Displays cautions or functionality unique to PSoC Creator or the PSoC device. |

## References

This guide is one of a set of documents pertaining to PSoC Creator and PSoC devices. Refer to the following other documents as needed:

- PSoC Creator Help
- PSoC Creator Component Datasheets
- PSoC Creator Component Author Guide
- PSoC Technical Reference Manual (TRM)

## Sample Firmware Source Code

PSoC Creator provides numerous example projects that include schematics and example code in the Find Example Project dialog. For component-specific examples, open the dialog from the Component Catalog or an instance of the component in a schematic. For general examples, open the dialog from the Start Page or **File** menu. As needed, use the **Filter Options** in the dialog to narrow the list of projects available to select.

Refer to the “Find Example Project” topic in the PSoC Creator Help for more information.

## Revision History

| **Document Title: PSoC® Creator™ System Reference Guide, cy_boot Component v4.11**<br>**Document Number: 001-94480** | | |
|---|---|---|
| **Revision** | **Date** | **Description of Change** |
| ** | 9/30/14 | New document for version 4.11 of the cy_boot component. Refer to the change section for component changes from previous versions of cy_boot. |

# 2 Standard Types and Defines

To support the operation of the same code across multiple CPUs with multiple compilers, the cy_boot component provides types and defines that create consistent results across platforms.

## Base Types

| Type | Description |
|---|---|
| char8 | 8-bit (signed or unsigned, depending on the compiler selection for char) |
| uint8 | 8-bit unsigned |
| uint16 | 16-bit unsigned |
| uint32 | 32-bit unsigned |
| int8 | 8-bit signed |
| int16 | 16-bit signed |
| int32 | 32-bit signed |
| float32 | 32-bit float |
| float64 | 64-bit float (unavailable for PSoC 3) |
| int64 | 64-bit signed (unavailable for PSoC 3) |
| uint64 | 64-bit unsigned (unavailable for PSoC 3) |

## Hardware Register Types

Hardware registers typically have side effects and therefore are referenced with a volatile type.

| Define | Description |
|---|---|
| reg8 | Volatile 8-bit unsigned |
| reg16 | Volatile 16-bit unsigned |
| reg32 | Volatile 32-bit unsigned |

## Compiler Defines

The compiler being used can be determined by testing for the definition of the specific compiler. For example, to test for the PSoC 3 Keil compiler:

```
#if defined(__C51__)
```

| Define | Description |
|---|---|
| __C51__ | Keil 8051 compiler |
| __GNUC__ | ARM GCC compiler |
| __ARMCC_VERSION | ARM Realview compiler used by Keil MDK and RVDS tool sets |

## Keil 8051 Compatibility Defines

The Keil 8051 compiler supports type modifiers that are specific to this platform. For other platforms these modifiers must not be present. For compatibility these types are supported by defines that map to the appropriate string when compiled for Keil and an empty string for other platforms. These defines are used to create optimized Keil 8051 code while still supporting compilation on other platforms.

| Define | Keil Type | Other Platforms |
|---|---|---|
| CYBDTA | bdata | |
| CYBIT | bit | uint8 |
| CYCODE | code | |
| CYCOMPACT | compact | |
| CYDATA | data | |
| CYFAR | far | |
| CYIDATA | idata | |
| CYLARGE | large | |
| CYPDATA | pdata | |
| CYREENTRANT | reentrant | |
| CYSMALL | small | |
| CYXDATA | xdata | |

## Return Codes

Return codes from Cypress routines are returned as an 8-bit unsigned value type: cystatus. The standard return values are:

| Define | Description |
|---|---|
| CYRET_SUCCESS | Successful |
| CYRET_UNKNOWN | Unknown failure |
| CYRET_BAD_PARAM | One or more invalid parameters |
| CYRET_INVALID_OBJECT | Invalid object specified |
| CYRET_MEMORY | Memory related failure |
| CYRET_LOCKED | Resource lock failure |
| CYRET_EMPTY | No more objects available |
| CYRET_BAD_DATA | Bad data received (CRC or other error check) |
| CYRET_STARTED | Operation started, but not necessarily completed yet |
| CYRET_FINISHED | Operation completed |
| CYRET_CANCELED | Operation canceled |
| CYRET_TIMEOUT | Operation timed out |
| CYRET_INVALID_STATE | Operation not setup or is in an improper state |

## Interrupt Types and Macros

Types and macros provide consistent definition of interrupt service routines across compilers and platforms. Note that the macro to use is different between the function definition and the function prototype.

Function definition example:

```
CY_ISR(MyISR)
{
      /* ISR Code here */
}
```

Function prototype example:

```
CY_ISR_PROTO(MyISR);
```

### Interrupt vector address type

| Type | Description |
|---|---|
| cyisraddress | Interrupt vector (address of the ISR function) |

## Intrinsic Defines

| Define | Description |
|---|---|
| CY_NOP | Processor NOP instruction |

## Device Version Defines

| Define | Description |
|---|---|
| CY_PSOC3 | Any PSoC 3 Device |
| CY_PSOC5 | Any PSoC 5 Device |
| CY_PSOC4 | PSoC 4 Device |

## Variable Attributes

| Define | Description |
|---|---|
| CY_NOINIT | Specifies that a static variable should be placed into uninitialized data section that prevents this variable from being initialized to zero on startup.<br>For PSoC 3 no code is generated for this macro. |

# PSoC Creator Generated Defines

## Project Type

The following are defines for project type (from **Project > Build Settings**):

- CYDEV_PROJ_TYPE
- CYDEV_PROJ_TYPE_BOOTLOADER
- CYDEV_PROJ_TYPE_LOADABLE
- CYDEV_PROJ_TYPE_MULTIAPPBOOTLOADER
- CYDEV_PROJ_TYPE_STANDARD

## Chip Configuration Mode

The following are defines for chip configuration mode (from System DWR). Options vary by device:

### *All*

- CYDEV_CONFIGURATION_MODE
- CYDEV_CONFIGURATION_MODE_COMPRESSED
- CYDEV_CONFIGURATION_MODE_DMA
- CYDEV_CONFIGURATION_MODE_UNCOMPRESSED
- CYDEV_DEBUGGING_ENABLE or
  CYDEV_PROTECTION_ENABLE (Debugging or protection enabled. Mutually exclusive.)

### *PSoC 3*

- CYDEV_CONFIGURATION_CLEAR_SRAM (Startup code clear SRAM?)

### *PSoC 3 and PSoC 5LP*

- CYDEV_CONFIGURATION_COMPRESSED (Configuration data compressed?)
- CYDEV_CONFIGURATION_DMA (Configuration data loaded via DMA?)
- CYDEV_CONFIGURATION_ECC (Configuration data stored in ECC?)
- CYDEV_CONFIG_FASTBOOT_ENABLED (Device startup at 48 MHz at boot? If not, 12 MHz.)
- CYDEV_INSTRUCT_CACHE_ENABLED (Instruction cache enabled?)
- CYDEV_DMA_CHANNELS_AVAILABLE (Number of DMA channels available for configuration.)
- CYDEV_ECC_ENABLE (ECC enabled?)
- CYDEV_DEBUGGING_XRES (Optional XRES pin enabled as XRES?)

### *PSoC 4*

- CYDEV_CONFIG_READ_ACCELERATOR (Flash read accelerator enabled?)

### *PSoC 4 and PSoC 5LP*

- CYDEV_USE_BUNDLED_CMSIS (Include the CMSIS standard library.)

## Debugging Mode

The following are defines for debugging mode (from System DWR):

- CYDEV_DEBUGGING_DPS
- CYDEV_DEBUGGING_DPS_Disable
- CYDEV_DEBUGGING_DPS_JTAG_4
- CYDEV_DEBUGGING_DPS_JTAG_5
- CYDEV_DEBUGGING_DPS_SWD
- CYDEV_DEBUGGING_DPS_SWD_SWV

## Chip Protection Mode

The following are defines for chip protection mode (from System DWR):

- CYDEV_DEBUG_PROTECT
- CYDEV_DEBUG_PROTECT_KILL
- CYDEV_DEBUG_PROTECT_OPEN
- CYDEV_DEBUG_PROTECT_PROTECTED

## Stack and Heap

The following are defines for the number of bytes allocated to the stack and heap (from System DWR). These are only for PSoC 4 and PSoC 5LP.

- CYDEV_HEAP_SIZE
- CYDEV_STACK_SIZE

## Voltage Settings

The following are defines for voltage settings (from System DWR). Options vary by device:

- CYDEV_VARIABLE_VDDA
- CYDEV_VDDA
- CYDEV_VDDA_MV
- CYDEV_VDDD
- CYDEV_VDDD_MV
- CYDEV_VDDIO0
- CYDEV_VDDIO0_MV
- CYDEV_VDDIO1
- CYDEV_VDDIO1_MV
- CYDEV_VDDIO2
- CYDEV_VDDIO2_MV
- CYDEV_VDDIO3
- CYDEV_VDDIO3_MV

- CYDEV_VIO0
- CYDEV_VIO0_MV
- CYDEV_VIO1
- CYDEV_VIO1_MV
- CYDEV_VIO2
- CYDEV_VIO2_MV
- CYDEV_VIO3
- CYDEV_VIO3_MV

## System Clock Frequency

The following are defines for system clock frequency (from Clock DWR):

### *PSoC 3 and PSoC 5*

- BCLK__BUS_CLK__HZ
- BCLK__BUS_CLK__KHZ
- BCLK__BUS_CLK__MHZ

### *PSoC 4*

- CYDEV_BCLK__HFCLK__HZ
- CYDEV_BCLK__HFCLK__KHZ
- CYDEV_BCLK__HFCLK__MHZ
- CYDEV_BCLK__SYSCLK__HZ
- CYDEV_BCLK__SYSCLK__KHZ
- CYDEV_BCLK__SYSCLK__MHZ

## JTAG/Silicon ID

The following is the define for JTAG/Silicon ID for the current device:

- CYDEV_CHIP_JTAG_ID

## IP Block Information

PSoC Creator generates the following macros in the *cyfitter.h* file for the IP blocks that exist on the current device:

```
#define CYIPBLOCK_<BLOCK NAME>_VERSION <version>
```

For example:

```
#define CYIPBLOCK_P3_TIMER_VERSION 0
#define CYIPBLOCK_P3_USB_VERSION 0
#define CYIPBLOCK_P3_VIDAC_VERSION 0
```

# 3 Clocking

## PSoC Creator Clocking Implementation

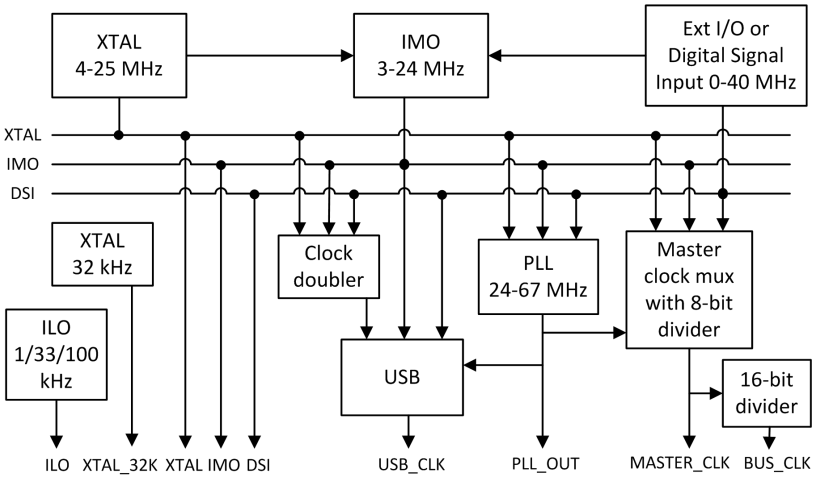
PSoC devices supported by PSoC Creator have flexible clocking capabilities. These clocking capabilities are controlled in PSoC Creator by selections within the Design-Wide Resources settings, connectivity of clocking signals on the design schematic, and API calls that can modify the clocking at runtime.

This section describes how PSoC Creator maps clocks onto the device and provides guidance on clocking methodologies that are optimized for the PSoC architecture.

PSoC 3/PSoC 5LP devices have different clocking architectures than PSoC 4 devices. The System Clock consolidates Bus Clock (BUS_CLK) on PSoC 3/PSoC 5LP devices and System Clock (SYSCLK) on PSoC 4 devices. The Master Clock consolidates Master Clock (MASTER_CLK) on PSoC 3/PSoC 5LP devices and High-Frequency Clock (HFCLK) on PSoC 4 devices.

### PSoC 3/PSoC 5LP

#### *Overview*




The clock system includes these clock resources:

- Four internal clock sources increase system integration:
  - 3 to 48 MHz Internal Main Oscillator (IMO) ±1% at 3 MHz
  - 1 kHz, 33 kHz, 100 kHz Internal Low Speed Oscillator (ILO) outputs
  - USB Clock Domain, sourced from IMO, MHz External Crystal Oscillator (MHzECO), and Digital System Interconnect (DSI)
  - 24 to 67 MHz fractional Phase-Locked Loop (PLL) sourced from IMO, MHzECO, and DSI

- Clock generated using a DSI signal from an external I/O pin or other logic
- Two external clock sources provide high precision clocks:
  - 4 to 25 MHzECO
  - 32.768 kHz External Crystal Oscillator (kHzECO) for Real Time Clock (RTC)
- Dedicated 16-bit divider for bus clock
- Eight individually sourced 16-bit clock dividers for the digital system peripherals
- Four individually sourced 16-bit clock dividers with skew for the analog system peripherals
- IMO has a USB mode that synchronizes to USB host traffic, requiring no external crystal for USB. (USB equipped parts only)

## PSoC 4

### *Overview*

The clock system includes these clock resources:

- Two internal clock sources increase system integration:
  - PSoC 4000: 24, 32 and 48 MHz IMO ±2% across all frequencies when Vddd is above or equal to 2.0 V and +/-4% below 2.0 V.
  - PSoC 4100 and PSoC 4200: 3 to 48 MHz IMO ±2% across all frequencies
  - 32 kHz ILO outputs
- External Clock (EXTCLK) generated using a signal from a single designated I/O pin:
  - The allowable external clock frequency has the same limits as the system clock frequency.
  - The device always starts up using the IMO and the external clock must be enabled, so the device cannot be started from a reset clocked by the external clock.
- HFCLK selected from IMO or external clock:
- PSoC 4000: The HFCLK frequency cannot exceed 16 MHz.
- PSoC 4100 and PSoC 4200: The HFCLK frequency cannot exceed 48 MHz.
- Low-Frequency Clock (LFCLK) sourced by ILO
- Dedicated prescaler for SYSCLK sourced by HFCLK. The SYSCLK must be equal to or faster than all other clocks in the device.
  - PSoC 4000: The SYSCLK frequency cannot exceed 16 MHz.
  - PSoC 4100 and PSoC 4200: The SYSCLK frequency cannot exceed 48 MHz.
- Four peripheral clock dividers, each containing three chainable 16-bit dividers
- 16 digital and analog peripheral clocks

### *Power Modes*

The IMO is available in Active and Sleep modes. It is automatically disabled/enabled for the proper Deep Sleep and Hibernate mode entry/exit. The IMO is disabled during Deep Sleep and Hibernate modes.

The EXTCLK is available in Active and Sleep modes. The system will enter/exit Deep Sleep and Hibernate using external clock. The device will re-enable the IMO if it was enabled before entering Deep Sleep or Hibernate, but it does not wait for the IMO before starting the CPU. After entering Active mode, the IMO may take an additional 2 us to begin toggling. The IMO will startup cleanly without glitches, but

any dependency should account for this extra startup time. If desired, firmware may increase wakeup hold-off using CySysPmSetWakeupHoldoff() function to include this 2 us and ensure the IMO is toggling by the time Active mode is reached.

The ILO is available in all modes except Hibernate and Stop.

## Clock Connectivity

The PSoC architecture includes flexible clock generation logic. Refer to the *Technical Reference Manual* for a detailed description of all the clocking sources available in a particular device. The usage of these various clocking sources can be categorized by how those clocks are connected to elements of a design.

### *System Clock*

This is a special clock. It is closely related to Master Clock. For most designs, Master Clock and System Clock will be the same frequency and considered to be the same clock. These must be the highest speed clocks in the system. The CPU will be running off of System Clock and all the peripherals will communicate to the CPU and DMA using System Clock. When a clock is synchronized, it is synchronized to Master Clock. When a pin is synchronized it is synchronized to System Clock.

### *Global Clock*

This is a clock that is placed on one of the global low skew digital clock lines. This also includes System Clock. When a clock is created using a Clock component, it will be created as a global clock. This clock must be directly connected to a clock input or may be inverted before connection to a clock input. Global clock lines connect only to the clock input of the digital elements in PSoC. If a global clock line is connected to something other than a clock input (that is, combinatorial logic or a pin), then the signal is not sent using low skew clock lines.

### *Routed Clock*

Any clock that is not a global clock is a routed clock. This includes clocks generated by logic (with the exception of a single inverter) and clocks that come in from a pin.

## Clock Synchronization

Each clock in a PSoC device is either synchronous or asynchronous. This is in reference to System Clock and Master Clock. PSoC is designed to operate as a synchronous system. This was done to enable communication between the programmable logic and either the CPU or DMA. If these are not synchronous to a common clock, then any communication requires clocking crossing circuitry. Generally, asynchronous clocking is not supported except for PLD logic that does not interact with the CPU system.

### *Synchronous Clock (PSoC 3/ PSoC 5LP)*

Examples of synchronous clocks include:

- Global clock with sync to Master Clock option set. This option is set by default on the **Advanced** tab of the Clock component Configure dialog.
- Clock from an input pin with the "Input Synchronized" option selected. This option is set by default on the **Input** tab of the Pins component Configure dialog.
- Clock derived combinatorially from signals that were all generated from registers that are clocked by synchronous clocks.

### *Asynchronous Clock (PSoC 3/ PSoC 5LP)*

An asynchronous clock is any clock that is not synchronous. Some examples are:

- Any signal coming in from the Digital System Interconnect (DSI) other than a synchronized pin. These signals must be considered asynchronous because their timing is not guaranteed. This includes:
  - What would normally be a global clock (if connected directly to a clock input) that is fed through logic before being used as a clock
  - Fixed function block outputs (that is, Counter, Timer, PWM)
  - Digital signals from the analog blocks
- Global clock without the sync option set
- Clock from an input pin with Input Synchronized not selected
- Clock that is combinatorially created using any asynchronous signal

### *Making Signals Synchronous (PSoC 3/ PSoC 5LP)*

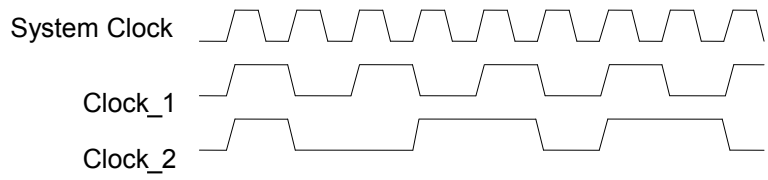
Depending on the source of the clock signal, it can be made synchronous using different methods:

- An asynchronous global clock can be made synchronous by checking the **Sync with MASTER_CLK** option in the Clock component Configure dialog (this is the default selection).
- A routed clock coming from a pin can be made synchronous by checking the **Input Synchronized** option in the Pins component Configure dialog (this is the default selection, under the **Pins** tab).
- Any signal can be made synchronous by using the Sync component and a synchronous clock as the clock signal.

When synchronizing a signal:

- The synchronizing clock must be at least 2x the frequency of the signal being synchronized. If this rule is violated, then incoming clock edges can be missed and therefore not reflected in the resulting synchronized clock.
- The clock signal output will have all its transitions on the rising edge of the synchronizing clock.
- The clock signal output will have its edges moved from their original timing.
- The clock signal output will have variation in the high and low pulse widths unless the incoming clock and the synchronizing clock are directly related to each other.

The following example shows two clocks that have been synchronized to System Clock. Clock_1 has exactly 2x the period of System Clock. Clock_2 has a period of approximately 3x the period of System Clock. That results in the high and low pulse widths varying between 1 and 2 System Clock periods. In both cases all transitions occur at the rising edge of System Clock.



## Routed Clock Implementation

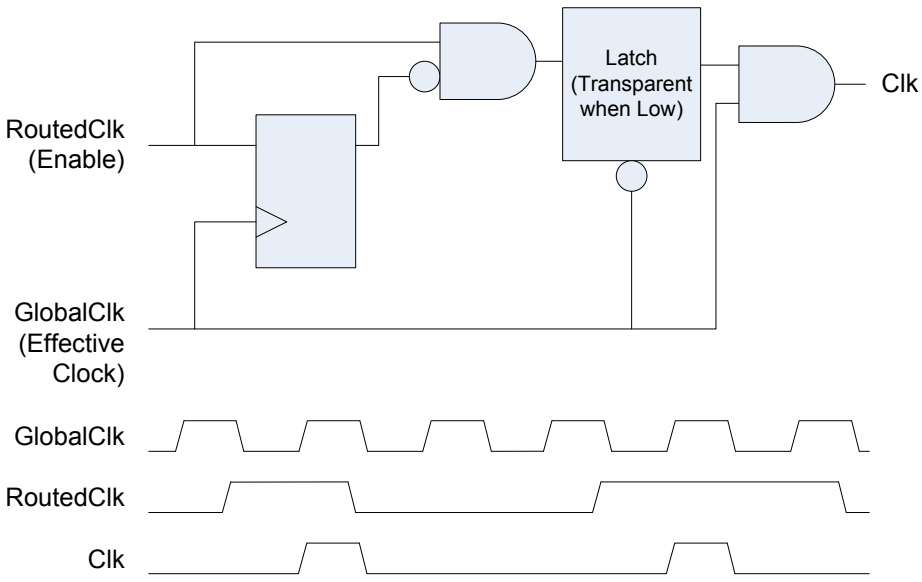
The clocking implementation in PSoC directly connects global clock signals to the clock input of clocked digital logic. This applies to both synchronous and asynchronous clocks. Since global clocks are distributed on low skew clock lines, all clocked elements connected to the same global clock will be clocked at the same time.

Routed clocks are distributed using the general digital routing fabric. This results in the clock arriving at each destination at different times. If that clock signal was used directly as the clock, then it would force the clock to be considered an asynchronous clock. This is because it cannot be guaranteed to transition at the rising edge of System Clock. This can also result in circuit failures if the output of a register clocked by an early arriving clock is used by a register clocked by a late arriving version of the same clock.

Under some circumstances, PSoC Creator can transform a routed clock circuit into a circuit that uses a global clock. If all the sources of a routed clock can be traced back to the output of registers that are clocked by common global clocks, then the circuit is transformed automatically by PSoC Creator. The cases where this is possible are:

- All signals are derived from the same global clock. This global clock can be asynchronous or synchronous.
- All signals are derived from more than one synchronous global clock. In this case, the common global clock is System Clock.

The clocking implementation in PSoC includes a built-in edge detection circuit that is used in this transformation. This does not use PLD resources to implement. The following shows the logical implementation and the resulting clock timing diagram.




This diagram shows that the resulting clock occurs synchronous to the global clock on the first clock after a rising edge of the routed clock.

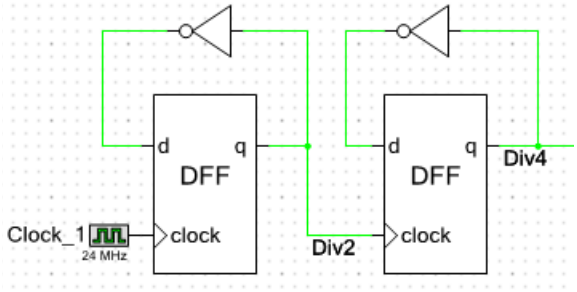
When analyzing the design to determine the source of a routed clock, another routed clock that was transformed may be encountered. In that case, the global clock used in that transformation is considered the source clock for that signal.

The clock transformation used for every routed clock is reported in the report file. This file is located in the Workspace Explorer under the **Results** tab after a successful build. The details are shown under the

"Initial Mapping" heading. Each routed clock will be shown with the "Effective Clock" and the "Enable Signal". The "Effective Clock" is the global clock that is used and the "Enable Signal" is the routed clock that is edge detected and used as the enable for that clock.

### *Example with a Divided Clock*

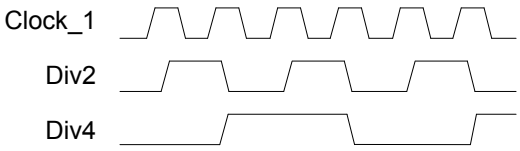
A simple divided clock circuit can be used to observe how this transformation is done. The following circuit clocks the first flip-flop (cydff_1) with a global clock. This generates a clock that is divided by 2 in frequency. That signal is used as a routed clock that clocks the next flip-flop (cydff_2).




The report file indicates that one global clock has been used and that the single routed clock has been transformed using the global clock as the effective clock.

```
<CYPRESSTAG name="Tech mapping">
<CYPRESSTAG name="Initial Mapping" icon="FILE_RPT_TECHM">
<CYPRESSTAG name="Global Clock Selection" icon="FILE_RPT_TECHM">
    Digital Clock 0: Automatic-assigning  clock 'Clock_1'. Fanout=1, Signal=tmp__cydff_1_clk
</CYPRESSTAG>
<CYPRESSTAG name="UDB Routed Clock Assignment">
    Routed Clock: tmp__cydff_1_reg:macrocell.q
        Effective Clock: Clock_1
        Enable Signal: tmp__cydff_1_reg:macrocell.q
</CYPRESSTAG>
```

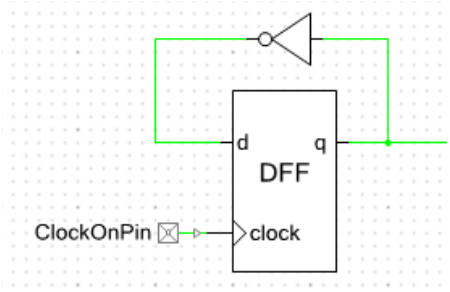
The resulting signals generated by this circuit are as follows.




It may appear that the Div4 signal is generated by the falling edge of the Div2 signal. This is not the case. The Div4 signal is generated on the first Clock_1 rising edge following a rising edge on Div2.

### *Example with a Clock from a Pin*

In the following circuit, a clock is brought in on a pin with synchronization turned on. Since synchronization of pins is done with System Clock, the transformed circuit uses System Clock as the Effective Clock and uses the rising edge of the pin as the Enable Signal.



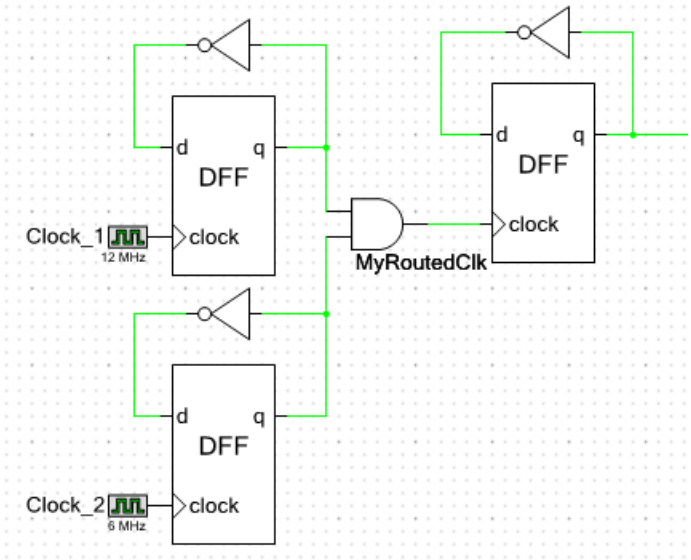

```
<CYPRESSTAG name="Initial Mapping" icon="FILE_RPT_TECHM">
{Global Clock Selection}
<CYPRESSTAG name="UDB Routed Clock Assignment">
    Routed Clock: ClockOnPin(0):iocell.fb
        Effective Clock: BUS_CLK
        Enable Signal: ClockOnPin(0):iocell.fb
</CYPRESSTAG>
```

If input synchronization was not enabled at the pin, there would not be a global clock to use to transform the routed clock, and the routed clock would be used directly.

```
<CYPRESSTAG name="Initial Mapping" icon="FILE_RPT_TECHM">
<CYPRESSTAG name="Global Clock Selection" icon="FILE_RPT_TECHM">
</CYPRESSTAG>
<CYPRESSTAG name="UDB Routed Clock Assignment">
    Routed Clock: ClockOnPin(0):iocell.fb
        Effective Clock: ClockOnPin(0):iocell.fb
        Enable Signal: True
</CYPRESSTAG>
```

### *Example with Multiple Clock Sources*

In this example, the routed clock is derived from flip-flops that are clocked by two different clocks. Both of these clocks are synchronous, so System Clock is the common global clock that becomes the Effective Clock.

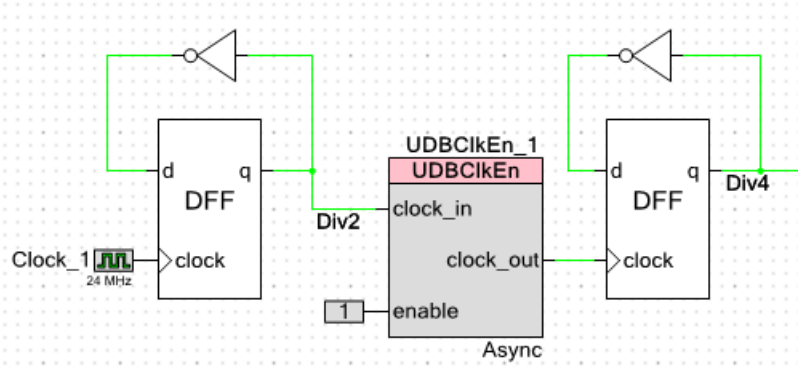


```
<CYPRESSTAG name="Tech mapping">
<CYPRESSTAG name="Initial Mapping" icon="FILE_RPT_TECHM">
<CYPRESSTAG name="Global Clock Selection" icon="FILE_RPT_TECHM">
    Digital Clock 0: Automatic-assigning  clock 'Clock_1'. Fanout=1, Signal=tmp__cydff_1_clk
    Digital Clock 1: Automatic-assigning  clock 'Clock_2'. Fanout=1, Signal=tmp__cydff_2_clk
</CYPRESSTAG>
<CYPRESSTAG name="UDB Routed Clock Assignment">
    Routed Clock: MyRoutedClk:macrocell.q
        Effective Clock: BUS_CLK
        Enable Signal: MyRoutedClk:macrocell.q
</CYPRESSTAG>
<CYPRESSTAG name="UDB Clock/Enable Remapping Results">
</CYPRESSTAG>
```

If either of these clocks had been asynchronous, then the routed clock would have been used directly.

### *Overriding Routed Clock Transformations*

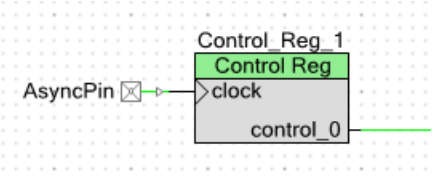
The automatic transformation that PSoC Creator performs on routed clocks is generally the implementation that should be used. There is however a method to force the routed clock to be used directly. The UDBClkEn component configured in Async mode will force the clock used to be the routed clock, as shown in the following circuit.




## Using Asynchronous Clocks

Asynchronous clocks can be used with PLD logic. However, they are not automatically supported by control registers, status registers and datapath elements because of the interaction with the CPU those elements have. Most Cypress library components will only work with synchronous clocks. They specifically force the insertion of a synchronizer automatically if the clock provided is asynchronous. Components that are designed to work with asynchronous clocks such as the SPI Slave will specifically describe how they handle clocking in their datasheet.

If an asynchronous clock is connected directly to something other than PLD logic, then a Design Rule Check (DRC) error is generated. For example, if an asynchronous pin is connected to a control register clock, a DRC error is generated.




mpr.M0115:Routing of asynchrononus signal AsyncPin(0):iocell.fb as a clock to UDB component '\Control_Reg_1:ctrl_reg\' is not supported unless a UDB Clock/Enable component is used.

As stated in the error message, the error can be removed by using a UDBClkEn component in async mode. That won't remove the underlying synchronization issue, but it will allow the design to override the error if the design has handled synchronization in some other way.

## Clock Crossing

Multiple clock domains are commonly needed in a design. Often these multiple domains do not interact and therefore clocking crossings do not occur. In the case where signals generated in one clock domain need to be used in another clock domain, special care must be taken. There is the case where the two clock domains are asynchronous from each other and the case where both clock domains are synchronous to System Clock.

When both clocks are synchronous to System Clock, signals from the slower clock domain can be freely used in the other clock domain. In the other direction, care must be taken that the signals from the faster clock domain are active for a long enough period that they will be sampled by the slower clock domain. In both directions the timing constraints that must be met are based on the speed of System Clock not the speed of either of the clock domains.

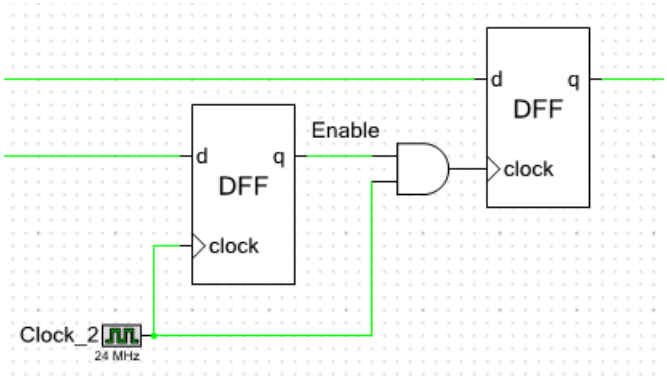
The only guarantee between the clock domains is that their edges will always occur on a rising edge of System Clock. That means that the rising edges of the two clock domains can be as close as a single System Clock cycle apart. This is true even when the clock domains are multiples of each other, since their clock dividers are not necessarily aligned. If combinatorial logic exists between the two clock domains, a flip-flop may need to be inserted to keep from limiting the frequency of System Clock operation. By inserting the flip-flop, the crossing from one clock domain to the other is a direct flip-flop to flip-flop path.

When the clock domains are unrelated to each other, a synchronizer must be used between the clock domains. The Sync component can be used to implement the synchronization function. It should be clocked by the destination clock domain.

The Sync component is implemented using a special mode of the status register that implements a double synchronizer. The input signal must have a pulse width of at least the period of the sampling clock. The exact delay to go through the synchronizer will vary depending on the alignment of the incoming signal to the synchronizing clock. This can vary from just over one clock period to just over two clock periods. If multiple signals are being synchronized, the time difference between two signals entering the synchronizer and those same two signals at the output can change by as much as one clock period, depending on when each is successfully sampled by the synchronizer.

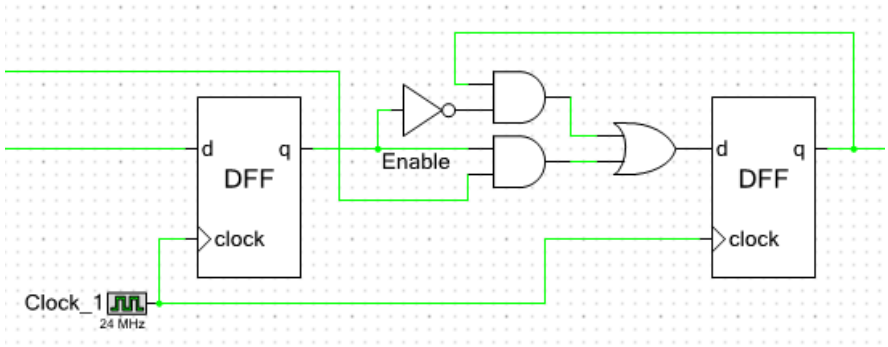
## Gated Clocks

Global clocks should not be used for anything other than directly clocking a circuit. If a global clock is used for logic functionality, the signal is routed using an entirely different path without guaranteed timing. A circuit such as the following should be avoided since timing analysis cannot be performed.




This circuit is implemented with a routed clock, has no timing analysis support, and is prone to the generation of glitches on the clock signal when the clock is enabled and disabled.

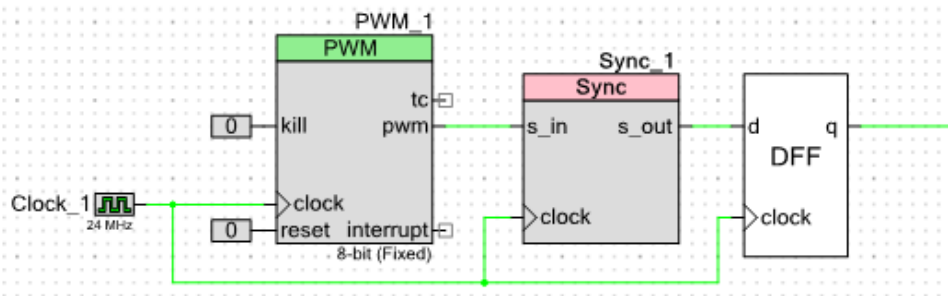
The following circuit implements the equivalent function and is supported by timing analysis, only uses global clocks, and has no reliability issues. This circuit does not gate the clock, but instead logically enables the clocking of new data or maintains the current data.




If access to a clock is needed, for example to generate a clock to send to a pin, then a 2x clock should be used to clock a toggle flip-flop. The output of that flip-flop can then be used with the associated timing analysis available.

## Fixed-Function Clocking

On the schematic, the clock signals sent to fixed-function peripherals and to UDB-based peripherals appear to be the same clock. However, the timing relationship between the clock signals as they arrive at these different peripheral types is not guaranteed. Additionally the routing delay for the data signals is not guaranteed. Therefore when fixed-function peripherals are connected to signals in the UDB array, the signals must be synchronized as shown in the following example. No timing assumptions should be made about signals coming from fixed-function peripherals.



## Changing Clocks in Run-time

### *Impact on Components Operation*

The components with internal clocks are directly impacted by the change of the system clock frequencies or sources. The components clock frequencies obtained using design-time dividers. The run-time change of components clock source will correspondingly change the internal component clock. Refer to the component datasheet for the details.

### *CyDelay APIs*

The CyDelay APIs implement simple software-based delay loops. The loops compensate for system clock frequency. The CyDelayFreq() function must be called in order to adjust CyDelay(), CyDelayUs() and CyDelayCycles() functions to the new system clock value.

### *Cache Configuration*

If the CPU clock frequency increases during device operation, the number of clock cycles cache will wait before sampling data coming back from Flash should be adjusted. If the CPU clock frequency decreases, the number of clock cycles can be also adjusted to improve CPU performance. See "CyFlash_SetWaitCycles()" for PSoC 3/PSoC 5LP and “CySysFlashSetWaitCycles()” for PSoC 4 for more information.

## Low Voltage Analog Boost Clocks (PSoC 3/PSoC 5LP)

When the operating voltage (Vdda) of a PSoC 3/PSoC 5LP device is below 2.7 V, analog positive pumps of the SC-blocks require a boost clock input in order to keep performance within specification. Between 2.7 V and 4.0 V, the boost is optional and may improve performance. Above 4.0 V, the boost must not be used.

In previous releases of PSoC Creator, an analog clock resource was silently reserved for every component instance of the requiring the boost (TIA, Mixer, PGA, and PGA_Inv). This means that analog designs would prematurely exhaust available resources when Vdda was low. This feature is implemented to automatically share the design-wide analog clock resource and provide the ability to control analog pumps and boost clock for all utilized SC-blocks during run-time using the CySetScPumps() function.

Components that are implemented in SC-blocks (TIA, Mixer, PGA, and PGA_Inv) will be initialized based on Vdda and Variable Vdda design-time options. The CySetScPumps() function can be used to enable/disable positive pumps and boost clock at run-time if operating voltage (Vdda) drops below the 2.7 V level. It is the user's responsibility to monitor the Vdda level.

When the operating voltage (Vdda) of a PSoC 3/PSoC 5LP device drops below 4.0 V, the analog pumps for the analog routing switches must be enabled by calling the SetAnalogRoutingPumps() function with the corresponding parameter. When Vdda rises above 4.0 V, the analog pumps for the analog routing switches must be disabled. It is the user's responsibility to monitor the Vdda level at run-time. The analog pumps for the analog routing switches are configured on device startup based on the **Vdda** and **Variable Vdda** design-time options.

A “Variable Vdda” option in the **System** tab of the PSoC Creator Design-Wide Resources (DWR) file is added to allow the creation of a design-wide analog clock resource (ScBoostClk) that will be used as a boost clock source for analog blocks. This clock is created with a desired frequency of 10 MHz. The

constraint is that one of the system clock sources (MASTER_CLK, PLL_OUT, XTAL, etc) should have a value which can produce a 10-12 MHz frequency via an integer divide. In order to ensure that sufficient current is provided to the SC-block by the pump, this value cannot be changed.

The dependency between **Vdda** and **Variable Vdda** values configured in the System tab of the PSoC Creator Design-Wide Resources (DWR) file is explained in the following table.

| Vdda | < 2.7 V | ≥ 2.7 V | ≥ 2.7 V |
|---|---|---|---|
| Variable Vdda | Always Enabled | Enabled | Disabled |
| ScBoostClk clock created | Yes | Yes | No |
| ScBoostClk started on reset | Yes | No | No |

**Note** The previous versions SC-block components (TIA, Mixer, PGA and, and PGA_Inv) will continue to use a dedicated local clock and the new Low Voltage Analog Boost Clocks APIs will not affect those clocks.

# APIs

There is one set of APIs used for PSoC 3 and PSoC 5LP devices, and a different set of APIs used for PSoC 4 devices. Functions starting with CySysClk are applicable to PSoC 4 only. All other functions are applicable to PSoC 3and PSoC 5LP only.

## PSoC 3/PSoC 5LP APIs

### *uint8 CyPLL_OUT_Start(uint8 wait)*

**Description:** Enables the PLL. Optionally waits for it to become stable. Waits at least 250 us or until it is detected that the PLL is stable.

**Parameters:** wait:

- 0: Return immediately after configuration
- 1: Wait for PLL lock or timeout

**Return Value:** Status

- CYRET_SUCCESS - Completed successfully
- CYRET_TIMEOUT - Timeout occurred without detecting a stable clock. If the input source of the clock is jittery, then the lock indication may not occur. However, after the timeout has expired the generated PLL clock can still be used.

**Side Effects and Restrictions:** If wait is enabled, this function uses the Fast Time Wheel (FTW) to time the wait. Any other use of the FTW will be stopped during the period of this function and then restored.
This function uses the 100 KHz ILO. If the 100 KHz ILO is not enabled, this function will enable it for the duration of this function execution.

No changes to the setup of the ILO, FTW, Central Time Wheel (CTW) or Once Per Second interrupt may be made by interrupt routines for the duration of this function execution. The current operation of the ILO, CTW and Once Per Second interrupt are maintained during the operation of this function, provided the reading of the Power Manager Interrupt Status Register is only done using the CyPmReadStatus() function.

### *void CyPLL_OUT_Stop()*

**Description:** Disables the PLL.

**Parameters:** None

**Return Value:** None

### *void CyPLL_OUT_SetPQ(uint8 pDiv, uint8 qDiv, uint8 current)*

**Description:** Sets the P and Q dividers and the charge pump current. The Frequency Out will be P/Q * Frequency In. The PLL must be disabled before calling this function.

**Parameters:** P: Valid range [8 - 255]

Q: Valid range [1 - 16]. Input Frequency / Q must be in the range of 1 MHz to 3 MHz.

current: Valid range [1 - 7]. Charge pump current in uA. Refer to the device TRM and datasheet for more information.

**Return Value:** None

**Side Effects and Restrictions:** If the CPU clock frequency increases during device operation, call CyFlash_SetWaitCycles() with the appropriate parameter to adjust the number of clock cycles cache will wait before sampling data coming back from Flash. If the CPU clock frequency decreases, you can call CyFlash_SetWaitCycles() to improve CPU performance. See "CyFlash_SetWaitCycles()" for more information.

### *void CyPLL_OUT_SetSource(uint8 source)*

**Description:** Sets the input clock source to the PLL. The PLL must be disabled before calling this function.

**Parameters:** source: One of the three available PLL clock sources

| Define | Source |
|---|---|
| CY_PLL_SOURCE_IMO | IMO |
| CY_PLL_SOURCE_XTAL | MHz Crystal |
| CY_PLL_SOURCE_DSI | DSI |

**Return Value:** None

**Side Effects and Restrictions:** If the CPU clock frequency increases during device operation, call CyFlash_SetWaitCycles() with the appropriate parameter to adjust the number of clock cycles cache will wait before sampling data coming back from Flash. If the CPU clock frequency decreases, you can call CyFlash_SetWaitCycles() to improve CPU performance. See "CyFlash_SetWaitCycles()" for more information.

### *void CyIMO_Start(uint8 wait)*

**Description:** Enables the IMO. Optionally waits at least 6us for it to settle.

**Parameters:** wait:

- 0: Return immediately after configuration
- 1: Wait for at least 6us for the IMO to settle

**Return Value:** None

**Side Effects and Restrictions:** If wait is enabled, this function uses the FTW to time the wait. Any other use of the FTW will be stopped during the period of this function and then restored. This function uses the 100 KHz ILO. If the 100 KHz ILO is not enabled, this function will enable it for the duration of this function execution.

No changes to the setup of the ILO, FTW, CTW, or Once Per Second interrupt may be made by interrupt routines for the duration of this function execution. The current operation of the ILO, CTW, and Once Per Second interrupt are maintained during the operation of this function, provided the reading of the Power Manager Interrupt Status Register is only done using the CyPmReadStatus() function.

### *void CyIMO_Stop()*

**Description:** Disables the IMO.

**Parameters:** None

**Return Value:** None

### *void CyIMO_SetFreq(uint8 freq)*

**Description:** Sets the frequency of the IMO. Changes may be made while the IMO is running.

**Parameters:** freq: Frequency of IMO operation

| Define | Frequency |
|---|---|
| CY_IMO_FREQ_3MHZ | 3 MHz |
| CY_IMO_FREQ_6MHZ | 6 MHz |
| CY_IMO_FREQ_12MHZ | 12 MHz |
| CY_IMO_FREQ_24MHZ | 24 MHz |
| CY_IMO_FREQ_48MHZ | 48 MHz |
| CY_IMO_FREQ_62MHZ | 62.6 MHz |
| CY_IMO_FREQ_74MHZ | 74.7 MHz |
| CY_IMO_FREQ_USB | 24 MHz (Trimmed for USB operation) |

**Return Value:** None

**Side Effects and Restrictions:** If the CPU clock frequency increases during device operation, call CyFlash_SetWaitCycles() with the appropriate parameter to adjust the number of clock cycles cache will wait before sampling data coming back from Flash. If the CPU clock frequency decreases, you can call CyFlash_SetWaitCycles() to improve CPU performance. See "CyFlash_SetWaitCycles()" for more information.

When the USB setting is chosen, the USB clock locking circuit is enabled. Otherwise this circuit is disabled. The USB block must be powered before selecting the USB setting.

### *void CyIMO_SetSource(uint8 source)*

**Description:** Sets the source of the clock output from the IMO block. The output from the IMO is by default the IMO itself. Optionally the MHz Crystal or a DSI input can be the source of the IMO output instead.

**Parameters:** source: One of the three available IMO output sources

| Define | Source |
|---|---|
| CY_IMO_SOURCE_IMO | IMO |
| CY_IMO_SOURCE_XTAL | MHz Crystal |
| CY_IMO_SOURCE_DSI | DSI |

**Return Value:** None

**Side Effects and Restrictions:** If the CPU clock frequency increases during device operation, call CyFlash_SetWaitCycles() with the appropriate parameter to adjust the number of clock cycles cache will wait before sampling data coming back from Flash. If the CPU clock frequency decreases, you can call CyFlash_SetWaitCycles() to improve CPU performance. See "CyFlash_SetWaitCycles()" for more information.

### *void CyIMO_EnableDoubler()*

**Description:** Enables the IMO doubler. The 2x frequency clock is used to convert a 24 MHz input to a 48 MHz output for use by the USB block.

**Parameters:** None

**Return Value:** None

### *void CyIMO_DisableDoubler()*

**Description:** Disables the IMO doubler.

**Parameters:** None

**Return Value:** None

### *void CyBusClk_SetDivider(uint16 divider)*

**Description:** Sets the divider value used to generate Bus Clock.

**Parameters:** divider: Valid range [0-65535]. The clock will be divided by this value + 1. For example to divide by 2 this parameter should be set to 1.

**Return Value:** None

**Side Effects and Restrictions:** If the CPU clock frequency increases during device operation, call CyFlash_SetWaitCycles() with the appropriate parameter to adjust the number of clock cycles cache will wait before sampling data coming back from Flash. If the CPU clock frequency decreases, you can call CyFlash_SetWaitCycles() to improve CPU performance. See "CyFlash_SetWaitCycles()" for more information.

### *void CyCpuClk_SetDivider(uint8 divider)*

**Description:** Sets the divider value used to generate the CPU Clock. Applies to PSoC 3 only.

**Parameters:** divider: Valid range [0-15]. The clock will be divided by this value + 1. For example to divide by 2 this parameter should be set to 1.

**Return Value:** None

**Side Effects and Restrictions:** If the CPU clock frequency increases during device operation, call CyFlash_SetWaitCycles() with the appropriate parameter to adjust the number of clock cycles cache will wait before sampling data coming back from Flash. If the CPU clock frequency decreases, you can call CyFlash_SetWaitCycles() to improve CPU performance. See "CyFlash_SetWaitCycles()" for more information.

### *void CyMasterClk_SetSource(uint8 source)*

**Description:** Sets the source of the master clock.

**Parameters:** source: One of the four available Master clock sources

| Define | Source |
|---|---|
| CY_MASTER_SOURCE_IMO | IMO |
| CY_MASTER_SOURCE_PLL | PLL |
| CY_MASTER_SOURCE_XTAL | MHz Crystal |
| CY_MASTER_SOURCE_DSI | DSI |

**Return Value:** None

**Side Effects and Restrictions:** The current source and the new source must both be running and stable before calling this function.

If the CPU clock frequency increases during device operation, call CyFlash_SetWaitCycles() with the appropriate parameter to adjust the number of clock cycles cache will wait before sampling data coming back from Flash. If the CPU clock frequency decreases, you can call CyFlash_SetWaitCycles() to improve CPU performance. See "CyFlash_SetWaitCycles()" for more information.

### *void CyMasterClk_SetDivider(uint8 divider)*

**Description:** Sets the divider value used to generate Master Clock.

**Parameters:** divider: Valid range [0-255]. The clock will be divided by this value + 1. For example to divide by 2 this parameter should be set to 1.

**Return Value:** None

**Side Effects and Restrictions:** When changing the Master or Bus Clock divider value from div-by-n to div-by-1, the first clock cycle output after the div-by-1 can be up to 4 ns shorter than the final/expected div-by-1 period.

If the CPU clock frequency increases during device operation, call CyFlash_SetWaitCycles() with the appropriate parameter to adjust the number of clock cycles cache will wait before sampling data coming back from Flash. If the CPU clock frequency decreases, you can call CyFlash_SetWaitCycles() to improve CPU performance. See "CyFlash_SetWaitCycles()" for more information.

### *void CyUsbClk_SetSource(uint8 source)*

**Description:** Sets the source of the USB clock.

**Parameters:** source: One of the four available USB clock sources

| Define | Source |
|---|---|
| CY_USB_SOURCE_IMO2X | IMO 2x |
| CY_USB_SOURCE_IMO | IMO |
| CY_USB_SOURCE_PLL | PLL |
| CY_USB_SOURCE_DSI | DSI |

**Return Value:** None

### *void CyILO_Start1K()*

**Description:** Enables the ILO 1 KHz oscillator.

**Note** The ILO 1 KHz oscillator is always enabled by default, regardless of the selection in the Clock Editor. Therefore, this API is only needed if the oscillator was turned off manually.

**Parameters:** None

**Return Value:** None

### *void CyILO_Stop1K()*

**Description:** Disables the ILO 1 KHz oscillator.

**Note** The ILO 1 KHz oscillator must be enabled if Sleep or Hibernate low power mode APIs are expected to be used. For more information, refer to the Power Management section of this document.

**Parameters:** None

**Return Value:** None

### *void CyILO_Start100K()*

**Description:** Enables the ILO 100 KHz oscillator.

**Parameters:** None

**Return Value:** None

### *void CyILO_Stop100K()*

**Description:** Disables the ILO 100 KHz oscillator.

**Parameters:** None

**Return Value:** None

### *void CyILO_Enable33K()*

**Description:** Enables the ILO 33 KHz divider.

**Note** The 33 KHz clock is generated from the 100 KHz oscillator, so it must also be running in order to generate the 33 KHz output.

**Parameters:** None

**Return Value:** None

### *void CyILO_Disable33K()*

**Description:** Disables the ILO 33 KHz divider.

**Note** that the 33 KHz clock is generated from the 100 KHz oscillator, but this API does not disable the 100 KHz clock.

**Parameters:** None

**Return Value:** None

### *void CyILO_SetSource(uint8 source)*

**Description:** Sets the source of the clock output from the ILO block.

**Parameters:** source: One of the three available ILO output sources

| Define | Source |
|---|---|
| CY_ILO_SOURCE_100K | ILO 100 KHz |
| CY_ILO_SOURCE_33K | ILO 33 KHz |
| CY_ILO_SOURCE_1K | ILO 1 KHz |

**Return Value:** None

### *uint8 CyILO_SetPowerMode(uint8 mode)*

**Description:** Sets the power mode used by the ILO during power down. Allows for lower power down power usage resulting in a slower startup time.

**Parameters:** mode:

| Define | Description |
|---|---|
| CY_ILO_FAST_START | Faster start-up, internal bias left on when powered down. |
| CY_ILO_SLOW_START | Slower start-up, internal bias off when powered down. |

**Return Value:** Previous power mode

### *uint8 CyXTAL_Start(uint8 wait)*

**Description:** Enables the MHz crystal.
Waits until the XERR bit is low (no error) for a millisecond or until the number of milliseconds specified by the wait parameter has expired.

**Parameters:** wait: Valid range [0-255]. This is the timeout value in milliseconds. The appropriate value is crystal specific.

**Return Value:** Status
CYRET_SUCCESS - Completed successfully
CYRET_TIMEOUT - Timeout occurred without detecting a low value on XERR.

**Side Effects and Restrictions:** If wait is enabled (non-zero wait), this function uses the FTW to time the wait. Any other use of the FTW will be stopped during the period of this function and then restored.
This function also uses the 100 KHz ILO. If the 100 KHz is not enabled, this function will enable it for the duration of this function execution.

No changes to the setup of the ILO, FTW, CTW, or Once Per Second interrupt may be made by interrupt routines for the duration of this function execution. The current operation of the ILO, CTW, and Once Per Second interrupt are maintained during the operation of this function provided the reading of the Power Manager Interrupt Status Register is only done using the CyPmReadStatus() function.

### *void CyXTAL_Stop()*

**Description:** Disables the megahertz crystal oscillator.

**Parameters:** None

**Return Value:** None

### *void CyXTAL_EnableErrStatus()*

**Description:** Enables the generation of the XERR status bit for the megahertz crystal.

**Parameters:** None

**Return Value:** None

### *void CyXTAL_DisableErrStatus()*

**Description:** Disables the generation of the XERR status bit for the megahertz crystal.

**Parameters:** None

**Return Value:** None

### *uint8 CyXTAL_ReadStatus()*

**Description:** Reads the XERR status bit for the megahertz crystal. This status bit is a sticky clear on read value.

**Parameters:** None

**Return Value:** Status: 0: No error, 1: Error

### *void CyXTAL_EnableFaultRecovery()*

**Description:** Enables the fault recovery circuit which will switch to the IMO in the case of a fault in the megahertz crystal circuit. The crystal must be up and running with the XERR bit at 0, before calling this function to prevent immediate fault switchover.

**Parameters:** None

**Return Value:** None

### *void CyXTAL_DisableFaultRecovery()*

**Description:** Disables the fault recovery circuit which will switch to the IMO in the case of a fault in the megahertz crystal circuit.

**Parameters:** None

**Return Value:** None

### *void CyXTAL_SetStartup(uint8 setting)*

**Description:** Sets the startup settings for the crystal.

**Parameters:** setting: Valid range [0-31]. Value is dependent on the frequency and quality of the crystal being used. Refer to the device TRM and datasheet for more information.

**Return Value:** None

### *void CyXTAL_SetFbVoltage(uint8 setting)*

**Description:** Sets the feedback reference voltage to use for the crystal circuit.

**Parameters:** setting: Valid range [0-15]. Refer to the device TRM and datasheet for more information.

**Return Value:** None

**Side Effects and Restrictions:** The feedback reference voltage must be greater than the watchdog reference voltage.

### *void CyXTAL_SetWdVoltage(uint8 setting)*

**Description:** Sets the reference voltage used by the watchdog to detect a failure in the crystal circuit.

**Parameters:** setting: Valid range [0-7]. Refer to the device TRM and datasheet for more information.

**Return Value:** None

**Side Effects and Restrictions:** The feedback reference voltage must be greater than the watchdog reference voltage.

### *void CyXTAL_32KHZ_Start()*

**Description:** Enables the 32 KHz Crystal Oscillator.

**Parameters:** None

**Return Value:** None

### *void CyXTAL_32KHZ_Stop()*

**Description:** Disables the 32 KHz Crystal Oscillator.

**Parameters:** None

**Return Value:** None

### *uint8 CyXTAL_32KHZ_ReadStatus()*

**Description:** Reads the two status bits for the 32 KHz oscillator.

**Parameters:** None

**Return Value:** Status

| Define | Source |
|---|---|
| CY_XTAL32K_ANA_STAT | Analog measurement<br>1: Stable<br>0: Not stable |

### *uint8 CyXTAL_32KHZ_SetPowerMode(uint8 mode)*

**Description:** Sets the power mode for the 32 KHz oscillator used during sleep mode. Allows for lower power during sleep when there are fewer sources of noise. During active mode the oscillator is always run in high power mode.

**Parameters:** mode:

- 0: High power mode
- 1: Low power mode during sleep

**Return Value:** Previous power mode

### *void CySetScPumps(uint8 enable)*

**Description:** Starts/stops analog boost clock and configures SC-blocks positive pumps.

**Parameters:** enable:

- 1: Starts analog boost clock and enables positive pumps.
- 0: Disables positive pumps for enabled SC-blocks and stops analog boost clock if all SC-blocks are disabled.

**Return Value:** None

### *void SetAnalogRoutingPumps(uint8 enabled)*

**Description:** Enables or disables the analog pumps feeding analog routing switches. Intended to be called at startup, based on the Vdda system configuration; may be called during operation when the user informs us that the Vdda voltage crossed the pump threshold.

**Parameters:** enabled:

- 1: Enable the pumps.
- 0: Disable the pumps.

**Return Value:** None

## PSoC 4 APIs

### *void CySysClkImoStart(void)*

**Description:** Enables the IMO.

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions:** None

### *void CySysClkImoStop(void)*

**Description:** Disables the IMO.

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions:** This function will have no effect if WDT is locked (CySysWdtLock() is called). Call CySysWdtUnlock() to unlock WDT and be able to stop ILO. Note that ILO is required for WDT's operation.

### *void CySysClkIloStart(void)*

**Description:** Starts the ILO.

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions:** None

### *void CySysClkIloStop(void)*

**Description:** Disables the ILO.

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions:** None

### *void CySysClkWriteHfclkDirect (uint32 clkSelect)*

**Description:** Selects the direct source for the HFCLK.

**Parameters:** clkSelect: One of the available HFCLK direct sources.

| Define | Source |
|---|---|
| CY_SYS_CLK_HFCLK_IMO | IMO |
| CY_SYS_CLK_HFCLK_EXTCLK | External clock pin |

**Return Value:** None

**Side Effects and Restrictions:** If the SYSCLK frequency increases during device operation, call CySysFlashSetWaitCycles() with the appropriate parameter to adjust the number of clock cycles the cache will wait before sampling data comes back from Flash. If the SYSCLK frequency decreases, call CySysFlashSetWaitCycles() to improve CPU performance. See CySysFlashSetWaitCycles() description for more information.

PSoC 4000: The SYSCLK has a maximum speed of 16 MHz, so HFCLK and SYSCLK dividers should be selected in a way to not to exceed 16 MHz for the System clock.

### *void CySysClkWriteSysclkDiv (uint32 divider)*

**Description:** Selects the prescaler divide amount for SYSCLK from HFCLK.

**Parameters:** divider: Power of 2 prescaler selection.

| Define | Divider |
|---|---|
| CY_SYS_CLK_SYSCLK_DIV1 | 1 |
| CY_SYS_CLK_SYSCLK_DIV2 | 2 |
| CY_SYS_CLK_SYSCLK_DIV4 | 4 |
| CY_SYS_CLK_SYSCLK_DIV8 | 8 |
| CY_SYS_CLK_SYSCLK_DIV16 | 16 |
| CY_SYS_CLK_SYSCLK_DIV32 | 32 |
| CY_SYS_CLK_SYSCLK_DIV64 | 64 |
| CY_SYS_CLK_SYSCLK_DIV128 | 128 |

**Note** The dividers above CY_SYS_CLK_SYSCLK_DIV8 are not available for the PSoC 4000 family.

**Return Value:** None

**Side Effects and Restrictions:** If the SYSCLK frequency increases during device operation, call CySysFlashSetWaitCycles() with the appropriate parameter to adjust the number of clock cycles the cache will wait before sampling data comes back from Flash. If the SYSCLK clock frequency decreases, call CySysFlashSetWaitCycles() to improve CPU performance. See CySysFlashSetWaitCycles() description for more information.

PSoC 4000: The SYSCLK has a maximum speed of 16 MHz, so HFCLK and SYSCLK dividers should be selected in a way to not to exceed 16 MHz for the System clock.

### *void CySysClkWriteImoFreq (uint32 freq)*

**Description:** Sets the frequency of the IMO.

**Parameters:** freq: Frequency for operation of the IMO.
PSoC 4000: Valid values are 24, 32 and 48 MHz.
PSoC 4100 and PSoC 4200: Valid range [3-48] with step size equal to 1 MHz.

**Return Value:** None

**Side Effects and Restrictions:** If the SYSCLK frequency increases during device operation, call CySysFlashSetWaitCycles() with the appropriate parameter to adjust the number of clock cycles the cache will wait before sampling data comes back from Flash. If the SYSCLK clock frequency decreases, call CySysFlashSetWaitCycles() to improve CPU performance. See CySysFlashSetWaitCycles() description for more information.

PSoC 4000: The SYSCLK has maximum speed of 16 MHz, so HFCLK and SYSCLK dividers should be selected in a way, to not to exceed 16 MHz for the System clock.

# 4 Power Management

There is a full range of power modes supported by PSoC devices to control power consumption and the amount of available resources. See the following table for the supported power modes.

Table 1. Power modes

| Architecture | PSoC 3 / PSoC 5LP | PSoC 4 | |
|---|---|---|---|
| **Family** | **All** | **4000** | **4100 and 4200** |
| Active | ✓ | ✓ | ✓ |
| Alternate Active | ✓ | | |
| Sleep | ✓ | ✓ | ✓ |
| Deep Sleep | | ✓ | ✓ |
| Hibernate | ✓ | | ✓ |
| Stop | | | ✓ |

PSoC 3/PSoC 5LP devices support the following power modes (in order of high to low power consumption): Active, Alternate Active, Sleep, and Hibernate.

PSoC 4 devices support the following power modes (in order of high to low power consumption): Active, Sleep, Deep Sleep, Hibernate, and Stop. Active, Sleep and Deep-Sleep are standard ARM defined power modes, supported by the ARM CPUs. Hibernate/Stop are even lower power modes that are entered from firmware just like Deep-Sleep, but on wakeup the CPU (and all peripherals) goes through a full reset.

For the ARM-based devices (PSoC 4/PSoC 5LP), an interrupt is required for the CPU to wake up. The Power Management implementation assumes that wakeup time is configured with a separate component (component-based wakeup time configuration) for an interrupt to be issued on terminal count. For more information, refer to the "Wakeup Time Configuration" section.

All pending interrupts should be cleared before the device is put into low power mode, even if they are masked.

## PSoC 3/PSoC 5LP Implementation

### Low Power Usage

PSoC 5 devices will not go into low power modes while the debugger is running.

For PSoC 3/PSoC 5LP devices, the power manager will not put the device into a low power state if the system performance controller (SPC) is executing a command. The device will go into low power mode after the SPC completes command execution. The SPC is used by Flash API, EEPROM and DieTemp components. Please refer to the corresponding component datasheet for the more information.

## Clock Configuration

There are a few device configuration requirements for proper low power mode entry and wakeup.

- The clock system should be prepared before entering Sleep and Hibernate mode to ensure that it will switch between Active modes and low power modes as expected.
- The CyPmSaveClocks() and CyPmRestoreClocks() functions are responsible for preparing clock configuration before entering low power mode and after waking up to Active mode, respectively. In general, CyPmSaveClocks() saves the configuration and sets the requirements for low power mode entry. CyPmRestoreClocks() restores the clock configuration to its original state.
- The IMO is required to be the source for the Master clock. So, the IMO clock value is set corresponding to the "Enable Fast IMO During Startup" option on the Design-Wide Resources System Editor. If this option is enabled, the IMO clock frequency is set to 48 MHz; otherwise, is set to 12 MHz.

  **Note** The IMO clock frequency is always set to 12 MHz on PSoC 5 devices. The PLL and MHz ECO are turned off when the Master clock is sourced by IMO.

  The IMO value must be 12 MHz just before entering Sleep and Hibernate modes. The IMO frequency is set to 12 MHz by CyPmSleep()/CyPmHibernate() just before entering the specified low power mode (without correcting the number of wait cycles for the flash). The IMO frequency is restored immediately on wakeup.
- The Bus and Master clock dividers are set to a divide-by-one value and the new value of flash wait cycles is set to match the new value of the CPU frequency. Refer to the description of the CyFlash_SetWaitCycles() function for more information.

The 1 KHz ILO must be enabled (it is always enabled by default, regardless of the selection in the Clock Editor) for all devices for correct operation in Sleep and Hibernate low power modes. It is used to measure the Hibernate/Sleep regulator settling time after a reset. During this time, the system ignores requests to enter these modes. The hold-off delay is measured using rising edges of the 1 kHz ILO. The terminal count is set by the Sleep Regulator Trim Register (PWRSYS_SLP_TR). **Caution** Do not modify this register. Refer to the corresponding device Registers TRM for more information.

The 32.768-kHz external crystal oscillator (32kHzECO) provides precision timing with minimal power consumption using an external 32.768-kHz watch crystal. The oscillator’s power mode during device’s Sleep mode configured by the CyXTAL_32KHZ_SetPowerMode() function. By default, oscillator runs in the high power mode.

Calling the CyPmSaveClocks() function will modify device clocking configuration. As a result, any component that relies on clocking should not be used until calling the CyPmRestoreClocks() function, which will restore the original clocking configuration. For information on component clocking requirements, refer to the corresponding component datasheet.

## Wakeup Time Configuration

There are three timers that can wake up a device from low power mode: CTW, FTW, and one pulse per second (One PPS). Refer to the device TRM and datasheet for more information on these timers.

There are two ways of configuring wakeup time:

- Parameter-based wakeup time configuration is done by calling the CyPmSleep() and CyPmAltAct() functions with desired parameters. This configuration method is available only for the PSoC 3 devices.
- Component-based wakeup time configuration. The CTW wakeup interval is configured with the Sleep Timer component. The one second interval is configured with the RTC component.

There is no wakeup time configuration available for the Hibernate mode.

It is important to keep in mind that it is only guaranteed that the first CTW and FTW intervals will be less than specified. To make subsequent intervals to have nominal values, the corresponding timer is enabled by the CyPmSleep() and CyPmAltAct() functions, and the timer left enabled. Note that some APIs can also use this timer. This can cause the timer to always be enabled (the timer interval can be changed only if the corresponding timer is disabled) before low power mode entry and hence the wakeup interval will always be less than expected.

The CyPmReadStatus() function must be called just after wakeup with a corresponding parameter (for example, with CY_PM_CTW_INT if the device is configured to wake up on CTW) to clear interrupt status bits.

When CTW is used as a wakeup timer, the CyPmReadStatus() function must always be called (when wakeup is configured in a parameter or component based method) after wakeup to clear the CTW interrupt status bit. It is required for this function to be called within 1 ms (1 clock cycle of the ILO) after the CTW event occurred.

## Wakeup Source Configuration

You can configure which wakeup source may wake up the device from Alternate Active and Sleep low power modes. The source is not configured to wake up the device; it just allows doing that. The component associated with the wakeup source has to be properly configured to act as a wakeup source.

For PSoC 5LP devices, the interrupts associated with wakeup sources must also be enabled to also wake up the CPU.

### *PSoC 3 Alternate Active Mode Specific Issues*

- Any interrupt, whether it is enabled at the interrupt controller or not, will wake the device from Alternate Active power mode.
- The edge detector is also bypassed, so the wakeup source is always level triggered.
- Directly connected DMA interrupts will not wake from this mode. They must be routed through the DSI in order to generate a wakeup condition.

### *PSoC 5LP Specific Issues*

For PSoC 5LP, the wakeup source is available for Sleep mode and is not available for Alternate Active mode. In the case of Alternate Active mode, the wakeup source argument is ignored and any of the available sources will wake the device.

For PSoC 5LP, the interrupt component connected to the wakeup source may not use the "RISING_EDGE" detect option. Use the "LEVEL" option instead.

## Power Management APIs

### *void CyPmSaveClocks()*

**Description:** This function is called in preparation for entering sleep or hibernate low power modes. Saves all state of the clocking system that doesn't persist during sleep/hibernate or that needs to be altered in preparation for sleep/hibernate. Shuts down all the digital and analog clock dividers for the active power mode configuration.
Switches the master clock over to the IMO and shuts down the PLL and MHz Crystal. The IMO frequency is set to either 12 MHz or 48 MHz to match the Design-Wide Resources System Editor "Enable Fast IMO During Startup" setting. The ILO and 32 KHz oscillators are not impacted. The current Flash wait state setting is saved and the Flash wait state setting is set for the current IMO speed.
**Note** If the Master Clock source is routed through the DSI inputs, then it must be set manually to another source before using the CyPmSaveClocks() / CyPmRestoreClocks() functions.

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions:** All peripheral clocks will be off after this API method call.

### *void CyPmRestoreClocks()*

**Description:** Restores any state that was preserved by the last call to CyPmSaveClocks. The Flash wait state setting is also restored.
**Note** If the Master Clock source is routed through the DSI inputs, then it must be set manually to another source before using the CyPmSaveClocks() / CyPmRestoreClocks() functions.
The merge region could be used to process state when the megahertz crystal is not ready after the hold-off timeout.

**Parameters:** None

**Return Value:** None

### *void CyPmAltAct(uint16 wakeupTime, uint16 wakeupSource)*

**Description:** Puts the part into the Alternate Active (Standby) state. The Alternate Active state can allow for any of the capabilities of the device to be active, but the operation of this function is dependent on the CPU being disabled during the Alternate Active state. The configuration code and the component APIs will configure the template for the Alternate Active state to be the same as the Active state with the exception that the CPU will be disabled during Alternate Active.

**Note** Before calling this function, you must manually configure the power mode of the source clocks for the timer that is used as the wakeup timer.

**PSoC 3:** Before switching to Alternate Active, if a wakeupTime other than NONE is specified, then the appropriate timer state is configured as specified with the interrupt for that timer disabled. The wakeup source will be the combination of the values specified in the wakeupSource and any timer specified in the wakeupTime argument. Once the wakeup condition is satisfied, then all saved state is restored and the function returns in the Active state.

**Note** If the wakeupTime is made with a different value, the period before the wakeup occurs can be significantly shorter than the specified time. If the next call is made with the same wakeupTime value, then the wakeup will occur the specified period after the previous wakeup occurred.
If a wakeupTime other than NONE is specified, then upon exit the state of the specified timer will be left as specified by wakeupTime with the timer enabled and the interrupt disabled. If the CTW, FTW or One PPS is already configured for wakeup, for example with the Sleep Timer or RTC components, then specify NONE for the wakeupTime and include the appropriate source for wakeupSource.

**PSoC 5LP:** Neither parameter is used. That means NONE should be passed for the parameters. The device will go into Alternate Active mode until an enabled interrupt occurs.

**Parameters:** wakeupTime: Specifies a timer wakeup source and the frequency of that source. For PSoC 5LP this parameter is ignored.

| Define | Time |
|---|---|
| PM_ALT_ACT_TIME_NONE | None |
| PM_ALT_ACT_TIME_ONE_PPS | One PPS: 1 second |
| PM_ALT_ACT_TIME_CTW_2MS | CTW: 2 ms |
| PM_ALT_ACT_TIME_CTW_4MS | CTW: 4 ms |
| PM_ALT_ACT_TIME_CTW_8MS | CTW: 8 ms |
| PM_ALT_ACT_TIME_CTW_16MS | CTW: 16 ms |
| PM_ALT_ACT_TIME_CTW_32MS | CTW: 32 ms |
| PM_ALT_ACT_TIME_CTW_64MS | CTW: 64 ms |
| PM_ALT_ACT_TIME_CTW_128MS | CTW: 128 ms |
| PM_ALT_ACT_TIME_CTW_256MS | CTW: 256 ms |
| PM_ALT_ACT_TIME_CTW_512MS | CTW: 512 ms |
| PM_ALT_ACT_TIME_CTW_1024MS | CTW: 1024 ms |
| PM_ALT_ACT_TIME_CTW_2048MS | CTW: 2048 ms |
| PM_ALT_ACT_TIME_CTW_4096MS | CTW: 4096 ms |
| PM_ALT_ACT_TIME_FTW(1-256) | FTW: 10 µs to 2.56 ms |

The PM_ALT_ACT_TIME_FTW() macro takes an argument that specifies how many increments of 10 µs to delay. For PSoC 3 silicon the valid range of values is 1 to 256.

### *CyPmAltAct (Continued)*

**Parameters:** wakeupSource: Specifies a bitwise mask of wakeup sources. In addition, if a wakeupTime has been specified, the associated timer will be included as a wakeup source. The wakeup source configuration is restored before function exit. For PSoC 5LP this parameter is ignored.

| Define | Source |
|---|---|
| PM_ALT_ACT_SRC_NONE | None |
| PM_ALT_ACT_SRC_COMPARATOR0 | Comparator 0 |
| PM_ALT_ACT_SRC_COMPARATOR1 | Comparator 1 |
| PM_ALT_ACT_SRC_COMPARATOR2 | Comparator 2 |
| PM_ALT_ACT_SRC_COMPARATOR3 | Comparator 3 |
| PM_ALT_ACT_SRC_INTERRUPT | Interrupt |
| PM_ALT_ACT_SRC_PICU | PICU |
| PM_ALT_ACT_SRC_I2C | I2C |
| PM_ALT_ACT_SRC_BOOSTCONVERTER | Boost Converter |
| PM_ALT_ACT_SRC_FTW | Fast Time Wheel |
| PM_ALT_ACT_SRC_VD | High and Low Voltage Detection |
| PM_ALT_ACT_SRC_CTW | Central Time Wheel |
| PM_ALT_ACT_SRC_ONE_PPS | One PPS |
| PM_ALT_ACT_SRC_LCD | LCD |

**Note** CTW and One PPS wakeup signals are in the same mask bit. FTW and Low Voltage Interrupt (LVI)/High Voltage Interrupt (HVI) wakeup signals are in the same mask bit.

When specifying a Comparator as the wakeupSource, use an instance specific define that will track with the specific comparator for that instance. As an example, for a Comparator instance named "MyComp" the value to OR into the mask is: MyComp_ctComp__CMP_MASK.
When CTW, FTW, or One PPS is used as a wakeup source, the CyPmReadStatus function must be called upon wakeup, with the corresponding parameter. Refer to the CyPmReadStatus API for more information.

**Return Value:** None

**Side Effects and Restrictions:** For PSoC 5LP, the wakeup source is not selectable. In this case the wakeupSource argument is ignored and any of the available wakeup sources will wake the device.
If a wakeupTime other than NONE is specified, then upon exit the state of the specified timer will be left as specified by wakeupTime with the timer enabled and the interrupt disabled. Also, the ILO 1 KHz (if CTW timer is used as wakeup time) or ILO 100 KHz (if FTW timer is used as wakeup time) will be left started.

### *void CyPmSleep(uint8 wakeupTime, uint16 wakeupSource)*

**Description:** Puts the part into the Sleep state.

**Note** Before calling this function, you must manually configure the power mode of the source clocks for the timer that is used as wakeup timer.

**Note** Before calling this function, you must prepare clock tree configuration for the low power mode by calling CyPmSaveClocks(). And restore clock configuration after CyPmSleep() execution by calling CyPmRestoreClocks(). See Power Management section, Clock Configuration subsection of the System Reference Guide for more information.

**PSoC 3:** Before switching to Sleep, if a wakeupTime other than NONE is specified, then the appropriate timer state is configured as specified with the interrupt for that timer disabled. The wakeup source will be the combination of the values specified in the wakeupSource and any timer specified in the wakeupTime argument. Once the wakeup condition is satisfied, then all saved state is restored and the function returns in the Active state.

**Note** If the wakeupTime value is different from the previous value, the period before the wakeup occurs can be significantly shorter than the specified time. If the next call is made with the same wakeupTime value, then the wakeup will occur with the specified period after the previous wakeup occurred.

If a wakeupTime other than NONE is specified, then upon exit the state of the specified timer will be left as specified by wakeupTime with the timer enabled and the interrupt disabled. If the CTW or One PPS is already configured for wakeup, for example with the Sleep Timer or RTC components, then specify NONE for the wakeupTime and include the appropriate source for wakeupSource.

**PSoC 5LP:** The wakeupTime parameter is not used and the only NONE can be specified. The wakeup time must be configured with the component, Sleep Timer for CTW intervals and RTC for 1PPS interval. The component must be configured to generate an interrupt.

**Parameters:** wakeupTime: Specifies a timer wakeup source and the frequency of that source. For PSoC 5LP, this parameter is ignored.

| Define | Time |
|---|---|
| PM_SLEEP_TIME_NONE | None |
| PM_SLEEP_TIME_ONE_PPS | One PPS: 1 second |
| PM_SLEEP_TIME_CTW_2MS | CTW: 2 ms |
| PM_SLEEP_TIME_CTW_4MS | CTW: 4 ms |
| PM_SLEEP_TIME_CTW_8MS | CTW: 8 ms |
| PM_SLEEP_TIME_CTW_16MS | CTW: 16 ms |
| PM_SLEEP_TIME_CTW_32MS | CTW: 32 ms |
| PM_SLEEP_TIME_CTW_64MS | CTW: 64 ms |
| PM_SLEEP_TIME_CTW_128MS | CTW: 128 ms |
| PM_SLEEP_TIME_CTW_256MS | CTW: 256 ms |
| PM_SLEEP_TIME_CTW_512MS | CTW: 512 ms |
| PM_SLEEP_TIME_CTW_1024MS | CTW: 1024 ms |
| PM_SLEEP_TIME_CTW_2048MS | CTW: 2048 ms |
| PM_SLEEP_TIME_CTW_4096MS | CTW: 4096 ms |

### *CyPmSleep (Continued)*

**Parameters:** wakeupSource: Specifies a bitwise mask of wakeup sources. In addition, if a wakeupTime has been specified, the associated timer will be included as a wakeup source. The wakeup source configuration is restored before function exit.

| Define | Source |
|---|---|
| PM_SLEEP_SRC_NONE | None |
| PM_SLEEP_SRC_COMPARATOR0 | Comparator 0 |
| PM_SLEEP_SRC_COMPARATOR1 | Comparator 1 |
| PM_SLEEP_SRC_COMPARATOR2 | Comparator 2 |
| PM_SLEEP_SRC_COMPARATOR3 | Comparator 3 |
| PM_SLEEP_SRC_PICU | PICU |
| PM_SLEEP_SRC_I2C | I2C |
| PM_SLEEP_SRC_BOOSTCONVERTER | Boost Converter |
| PM_SLEEP_SRC_VD | High and Low Voltage Detection |
| PM_SLEEP_SRC_CTW | Central Time Wheel |
| PM_SLEEP_SRC_ONE_PPS | One PPS |
| PM_SLEEP_SRC_LCD | LCD |

**Note** CTW and One PPS wakeup signals are in the same mask bit.

When specifying a Comparator as the wakeupSource, use an instance specific define that will track with the specific comparator for that instance. As an example for a Comparator instance named "MyComp" the value to OR into the mask is: MyComp_ctComp__CMP_MASK.
When CTW or One PPS is used as a wakeup source, the CyPmReadStatus function must be called upon wakeup, with the corresponding parameter. Refer to the CyPmReadStatus API for more information.

**Return Value:** None

**Side Effects and Restrictions:** If a wakeupTime other than NONE is specified, then upon exit the state of the specified timer will be left as specified by wakeupTime with the timer enabled and the interrupt disabled. Also, the ILO 1 KHz (if CTW timer is used as wake up time) will be left started.

The 1 kHz ILO clock is expected to be enabled to measure Hibernate/Sleep regulator settling time after a reset. The hold-off delay is measured using rising edges of the 1 kHz ILO.

### *void CyPmHibernate()*

**Description:** Puts the part into the Hibernate state.

Before switching to Hibernate, the current status of the PICU wakeup source bit is saved and then set. This configures the device to wake up from the PICU.

Make sure you have at least one pin configured to generate a PICU interrupt. For pin Px.y, the register "PICU_INTTYPE_PICUx_INTTYPEy" controls the PICU behavior. In the TRM, this register is "PICU[0..15]_INTTYPE[0..7]." In the Pins component datasheet, this register is referred to as the IRQ option. Once the wakeup occurs, the PICU wakeup source bit is restored and the PSoC returns to the Active state.

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions:** Applications must wait 20 µs before re-entering hibernate or sleep after waking up from hibernate. The 20 µs allows the sleep regulator time to stabilize before the next hibernate / sleep event occurs. The 20 µs requirement begins when the device wakes up. There is no hardware check that this requirement is met. The specified delay should be done on ISR entry.

After wakeup PICU interrupt occurs, the Pin_ClearInterrupt() function (where "Pin" is the instance name of the Pins component) must be called to clear the latched pin events. This allows proper Hibernate mode entry and enables detection of future events.

The 1 kHz ILO clock is expected to be enabled to measure Hibernate/Sleep regulator settling time after a reset. The hold-off delay is measured using rising edges of the 1 kHz ILO.

### *uint8 CyPmReadStatus(uint8 mask)*

**Description:** Manages the Power Manager Interrupt Status Register. This register has the interrupt status for the one pulse per second, CTW, and FTW timers. This hardware register clears on read. To allow for only clearing the bits of interest and preserving the other bits, this function uses a shadow register that retains the state. This function reads the status register and ORs that value with the shadow register. That is the value that is returned. Then the bits in the mask that are set are cleared from this value and written back to the shadow register.

**Note** You must call this function within 1 ms (1 clock cycle of the ILO) after a CTW event has occurred.

**Parameters:** mask: Bits in the shadow register to clear

| Define | Source |
|---|---|
| CY_PM_FTW_INT | Fast Time Wheel |
| CY_PM_CTW_INT | Central Time Wheel |
| CY_PM_ONEPPS_INT | One Pulse Per Second |

**Return Value:** Status. Same enumerated bit values as used for the mask parameter.

## PSoC 4 Implementation

The software should set EXT_VCCD bit in the PWR_CONTROL register when Vccd is shorted to Vddd on the board. This impacts the chip internal state transitions where it is necessary to know whether Vccd is connected or floating to achieve minimum current in low power modes.

**Note** Setting this bit turns off the active regulator and will lead to a system reset unless both Vddd and Vccd pins are supplied externally. Refer to the device TRM for more information.

It is safe to call PM APIs from the ISR. The wakeup conditions for Sleep and DeepSleep low power modes are illustrated in the table below:

| Interrupts State | Condition | Wakeup | ISR Execution |
|---|---|---|---|
| Unmasked | IRQ priority > current level | Yes | Yes |
| | IRQ priority ≤ current level | No | No |
| Masked | IRQ priority > current level | Yes | No |
| | IRQ priority ≤ current level | No | No |

### Power Management APIs

#### *void CySysPmSleep(void)*

**Description:** Puts the part into the Sleep state. This is a CPU-centric power mode. It means that the CPU has indicated that it is in “sleep” mode and its main clock can be removed. It is identical to Active from a peripheral point of view. Any enabled interrupts can cause wakeup from a Sleep mode.

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions:** None

#### *void CySysPmDeepSleep(void)*

**Description:** Puts the part into the Deep Sleep state.

If firmware attempts to enter this mode before the system is ready (that is, when PWR_CONTROL.LPM_READY = 0), then the device will go into Sleep mode instead and automatically enter the originally intended mode when the hold-off expires. The wakeup occurs when an interrupt is received from a DeepSleep or Hibernate peripheral. For more details, see corresponding peripheral’s datasheet.

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions:** None

### *void CySysPmHibernate(void)*

**Description:** It puts the part into the Hibernate state. Only SRAM and UDBs are retained; most internal supplies are off. Wakeup is possible from a pin or a hibernate comparator only.

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions:** This function does not apply to the PSoC 4000 family.

It is expected that the firmware has already frozen the IO-Cells using CySysPmFreezeIo() function before the call to this function. If this is omitted the IO-cells will be frozen in the same way as they are in the Active to Deep Sleep transition, but will lose their state on wake up (because of the reset occurring at that time).

Because all CPU state is lost, the CPU will start up at the reset vector. To save firmware state through Hibernate low power mode, corresponding variable should be defined with CY_NOINIT attribute. It prevents data from being initialized to zero on startup. The interrupt cause of the hibernate peripheral is retained, such that it can be either read by the firmware or cause an interrupt after the firmware has booted and enabled the corresponding interrupt. To distinguish the wakeup from the Hibernate mode and the general Reset event, the CySysPmGetResetReason() function could be used.

### *void CySysPmStop(void)*

**Description:** Puts the part into the Stop state. All internal supplies are off; no state is retained.

Wakeup from Stop is performed by toggling the wakeup pin (P0.7), causing a normal Boot procedure to occur. To configure the wakeup pin, the Digital Input Pin component should be placed on the schematic, assigned to the P0.7, and resistively pulled up or down to the inverse state of the wakeup polarity. To distinguish the wakeup from the Stop mode and the general Reset event, CySysPmGetResetReason() function could be used. The wakeup pin is active low by default. The wakeup pin polarity could be changed with the CySysPmSetWakeupPolarity() function.

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions:** This function does not apply to the PSoC 4000 family.

This function freezes IO cells implicitly. It is not possible to enter STOP mode before freezing the IO cells. The IO cells remain frozen after awake from the Stop mode until the firmware unfreezes them after booting explicitly with CySysPmUnfreezeIo() function call.

### *void CySysPmSetWakeupPolarity(uint32 polarity)*

**Description:** Wake up from stop mode is performed by toggling the wakeup pin (P0.7), causing a normal boot procedure to occur. This function assigns the wakeup pin active level. Setting the wakeup pin to this level will cause the wakeup from stop mode. The wakeup pin is active low by default.

**Parameters:** polarity: Wakeup pin active level

| Define | Description |
|---|---|
| CY_PM_STOP_WAKEUP_ACTIVE_LOW | Logical zero will wake up the chip |
| CY_PM_STOP_WAKEUP_ACTIVE_HIGH | Logical one will wake up the chip |

**Return Value:** None

**Side Effects and Restrictions:** None

### *uint32 CySysPmGetResetReason(void)*

**Description:** Retrieves last reset reason - transition from OFF/XRES/STOP/HIBERNATE to RESET state. Note that waking up from STOP using XRES will be perceived as general RESET.

**Parameters:** None

**Return Value:** Reset reason

| Define | Reset reason |
|---|---|
| CY_PM_RESET_REASON_UNKN | Unknown |
| CY_PM_RESET_REASON_XRES | Transition from OFF/XRES to RESET |
| CY_PM_RESET_REASON_WAKEUP_HIB | Transition/wakeup from HIBERNATE to RESET |
| CY_PM_RESET_REASON_WAKEUP_STOP | Transition/wakeup from STOP to RESET |

**Side Effects and Restrictions:** None

### *void CySysPmFreezeIo(void)*

**Description:** Freezes IO-Cells directly to save IO-Cell state on wake up from Hibernate or Stop mode.

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions:** It is not required to call this function before entering Stop mode, since CySysPmStop() function freezes IO-Cells implicitly.

### *void CySysPmUnfreezeIo(void)*

**Description:** The IO-Cells remain frozen after awake from Hibernate or Stop mode until the firmware unfreezes them after booting. The call of this function unfreezes IO-Cells explicitly.

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions:** None

### *void CySysPmSetWakeupHoldoff(uint32 hfclkFrequencyMhz)*

**Description:** Sets the Deep Sleep wakeup time by scaling the hold-off to the HFCLK frequency. This function must be called before increasing HFCLK clock frequency. It can optionally be called after lowering HFCLK clock frequency in order to improve Deep Sleep wakeup time.

It is functionally acceptable to leave the default hold-off setting, but Deep Sleep wakeup time may exceed the specification.

This function is applicable only for the PSoC 4000 family.

**Parameters:** uint32 hfclkFrequencyMhz: The HFCLK frequency in MHz. For example, if IMO frequency is 24 MHz, and HFCLK divider is 2, the function should be called with parameter 12 (the SYSCLK divider value should not be taken into account).

**Return Value:** None

**Side Effects and Restrictions:** None

## Instance Low Power APIs

Most components have an instance-specific set of low power APIs that allow you to put the component into its low power state. These functions are listed below generically. Refer to the individual datasheet for specific information regarding register retention information if applicable.

### void `=instance_name`_Sleep (void)

**Description:** The _Sleep() function checks to see if the component is enabled and saves that state. Then it calls the _Stop() function and calls _SaveConfig() function to save the user configuration.

- PSoC 3/PSoC 5LP: Call the _Sleep() function before calling the CyPmSleep() or the CyPmHibernate() function.
- PSoC 4: Call the _Sleep() function before calling the CySysPmDeepSleep() function.

**Parameters:** None

**Return Value:** None

**Side Effects:** None

### void `=instance_name`_Wakeup (void)

**Description:** The _Wakeup() function calls the _RestoreConfig() function to restore the user configuration. If the component was enabled before the _Sleep() function was called, the _Wakeup() function will re-enable the component.

**Parameters:** None

**Return Value:** None

**Side Effects:** Calling the _Wakeup() function without first calling the _Sleep() or _SaveConfig() function may produce unexpected behavior.

### void `=instance_name`_SaveConfig(void)

**Description:** This function saves the component configuration. This will save non-retention registers. This function will also save the current component parameter values, as defined in the Configure dialog or as modified by appropriate APIs. This function is called by the _Sleep() function.

**Parameters:** None

**Return Value:** None

**Side Effects:** None

### void `=instance_name`_RestoreConfig(void)

**Description:** This function restores the component configuration. This will restore non-retention registers. This function will also restore the component parameter values to what they were prior to calling the _Sleep() function.

**Parameters:** None

**Return Value:** None

**Side Effects:** Calling this function without first calling the _Sleep() or _SaveConfig() function may produce unexpected behavior.

# 5 Interrupts

The APIs in this chapter apply to all architectures except as noted. Refer also to the Interrupt component datasheet for more information about interrupts.

**Note** For PSoC 3, Keil C compiler run-time libraries do not disable interrupts. The only exception is found in the C51 run-time library when using large reentrant functions. Interrupts are disabled for 4 CPU instructions (8 CPU cycles) to adjust the large reentrant stack.

## APIs

### CyGlobalIntEnable

**Description:** Macro statement that enables interrupts using the global interrupt mask.

### CyGlobalIntDisable

**Description:** Macro statement that disables interrupts using the global interrupt mask.

### uint32 CyDisableInts()

**Description:** Disables all interrupts.

**Parameters:** None

**Return Value:** 32-bit mask of interrupts previously enabled

### void CyEnableInts(uint32 mask)

**Description:** Enables all interrupts specified in the 32-bit mask.

**Parameters:** mask: 32-bit mask of interrupts to enable

**Return Value:** None

**Note** Interrupt service routines must follow the policy that they restore the CYDEV_INTC_CSR_EN register bits and interrupt enable state (EA) to the way they were found on entry. The ISR does not need to do anything special as long as it uses properly nested CyEnterCriticalSection() and CyExitCriticalSection() function calls.

### void CyIntEnable(uint8 number)

**Description:** Enables the specified interrupt number.

**Parameters:** number: Interrupt number. Valid range: [0-31]

**Return Value:** None

**Note** Interrupt service routines must follow the policy that they restore the CYDEV_INTC_CSR_EN register bits and interrupt enable state (EA) to the way they were found on entry. The ISR does not need to do anything special as long as it uses properly nested CyEnterCriticalSection() and CyExitCriticalSection() function calls.

### void CyIntDisable(uint8 number)

**Description:** Disables the specified interrupt number.

**Parameters:** number: Interrupt number. Valid range: [0-31]

**Return Value:** None

**Note** Interrupt service routines must follow the policy that they restore the CYDEV_INTC_CSR_EN register bits and interrupt enable state (EA) to the way they were found on entry. The ISR does not need to do anything special as long as it uses properly nested CyEnterCriticalSection() and CyExitCriticalSection() function calls.

### uint8 CyIntGetState(uint8 number)

**Description:** Gets the enable state of the specified interrupt number.

**Parameters:** number: Interrupt number. Valid range: [0-31]

**Return Value:** Enable status: 1 if enabled, 0 if disabled

### cyisraddress CyIntSetVector(uint8 number, cyisraddress address)

**Description:** Sets the interrupt vector of the specified interrupt number.

**Parameters:** number: Interrupt number. Valid range: [0-31]

address: Pointer to an interrupt service routine

**Return Value:** Previous interrupt vector value

### cyisraddress CyIntGetVector(uint8 number)

**Description:** Gets the interrupt vector of the specified interrupt number.

**Parameters:** number: Interrupt number. Valid range: [0-31]

**Return Value:** Interrupt vector value

### cyisraddress CyIntSetSysVector(uint8 number, cyisraddress address)

**Description:** This function applies to ARM based processors only and therefore does not apply to the PSoC 3 device. It sets the interrupt vector of the specified exception. These exceptions in the ARM architecture operate similar to user interrupts, but are specified by the system architecture of the processor. The number of each exception is fixed. Note that the numbering of these exceptions is separate from the numbering used for user interrupts.

**Parameters:** number: Exception number. Valid range: [0-15].

address: Pointer to an interrupt service routine

**Return Value:** Previous interrupt vector value

### cyisraddress CyIntGetSysVector(uint8 number)

**Description:** This function applies to ARM based processors only and therefore does not apply to the PSoC 3 device. It gets the interrupt vector of the specified exception. These exceptions in the ARM architecture operate similar to user interrupts, but are specified by the system architecture of the processor. The number of each exception is fixed. Note that the numbering of these exceptions is separate from the numbering used for user interrupts.

**Parameters:** number: Exception number. Valid range: [0-15].

**Return Value:** Interrupt vector value

### void CyIntSetPriority(uint8 number, uint8 priority)

**Description:** Sets the priority of the specified interrupt number.

**Parameters:** number: Interrupt number. Valid range: [0-31]

priority: Interrupt priority. 0 is the highest priority. Valid range: [0-7]

**Return Value:** None

### uint8 CyIntGetPriority(uint8 number)

**Description:** Gets the priority of the specified interrupt number.

**Parameters:** number: Interrupt number. Valid range: [0-31]

**Return Value:** Interrupt priority

### void CyIntSetPending(uint8 number)

**Description:** Forces the specified interrupt number to be pending.

**Parameters:** number: Interrupt number. Valid range: [0-31]

**Return Value:** None

### void CyIntClearPending(uint8 number)

**Description:** Clears any pending interrupt for the specified interrupt number.

**Parameters:** number: Interrupt number. Valid range: [0-31]

**Return Value:** None

# 6 Pins

In addition to the functionality provided for pins as part of the Pins component, a library of pin macros is provided in the *cypins.h* file for the PSoC 3/PSoC 5LP devices. There is no library of pin macros available for the PSoC 4 devices. These macros all make use of the port pin configuration register that is available for every pin on the PSoC 3/PSoC 5LP device. The address of that register is provided in the *cydevice_trm.h* file. Each of these pin configuration registers is named:

```
CYREG_PRTx_PCy
```

where x is the port number and y is the pin number within the port.

For PSoC 4, there are status registers, data output registers, and port configuration registers only, so the macro takes two arguments: port register and pin number. Each port has these registers addresses defined:

```
CYREG_PRTx_DR
CYREG_PRTx_PS
CYREG_PRTx_PC
```

The x is the port number, and the second argument is the pin number.

## PSoC 3/PSoC 5LP APIs

### uint8 CyPins_ReadPin(uint16/uint32 pinPC)

**Description:** Reads the current value on the pin (pin state, PS).

**Parameters:** pinPC: Port pin configuration register (uint16 PSoC 3, uint32 PSoC 5LP)

**Return Value:** Pin state

0: Logic low value

Non-0: Logic high value

### void CyPins_SetPin(uint16/uint32 pinPC)

**Description:** Set the output value for the pin (data register, DR) to a logic high. Note that this only has an effect for pins configured as software pins that are not driven by hardware.

**Parameters:** pinPC: Port pin configuration register (uint16 PSoC 3, uint32 PSoC 5LP)

**Return Value:** None

## void CyPins_ClearPin(uint16/uint32 pinPC)

**Description:** Clear the output value for the pin (data register, DR) to a logic low. Note that this only has an effect for pins configured as software pins that are not driven by hardware.

**Parameters:** pinPC: Port pin configuration register (uint16 PSoC 3, uint32 PSoC 5)

**Return Value:** None

## void CyPins_SetPinDriveMode(uint16/uint32 pinPC, uint8 mode)

**Description:** Sets the drive mode for the pin (DM).

**Parameters:** pinPC: Port pin configuration register (uint16 PSoC 3, uint32 PSoC 5LP)

mode: Desired drive mode

| Define | Source |
|---|---|
| CY_PINS_DM_ALG_HIZ | Analog HiZ |
| CY_PINS_DM_DIG_HIZ | Digital HiZ |
| CY_PINS_DM_RES_UP | Resistive pull up |
| CY_PINS_DM_RES_DWN | Resistive pull down |
| CY_PINS_DM_OD_LO | Open drain - drive low |
| CY_PINS_DM_OD_HI | Open drain - drive high |
| CY_PINS_DM_STRONG | Strong CMOS Output |
| CY_PINS_DM_RES_UPDWN | Resistive pull up/down |

**Return Value:** None

## uint8 CyPins_ReadPinDriveMode(uint16/uint32 pinPC)

**Description:** Reads the drive mode for the pin (DM).

**Parameters:** pinPC: Port pin configuration register (uint16 PSoC 3, uint32 PSoC 5LP)

**Return Value:** Current drive mode for the pin

| Define | Source |
|---|---|
| CY_PINS_DM_ALG_HIZ | Analog HiZ |
| CY_PINS_DM_DIG_HIZ | Digital HiZ |
| CY_PINS_DM_RES_UP | Resistive pull up |
| CY_PINS_DM_RES_DWN | Resistive pull down |
| CY_PINS_DM_OD_LO | Open drain - drive low |
| CY_PINS_DM_OD_HI | Open drain - drive high |
| CY_PINS_DM_STRONG | Strong CMOS Output |
| CY_PINS_DM_RES_UPDWN | Resistive pull up/down |

## void CyPins_FastSlew(uint16/uint32 pinPC)

**Description:** Set the slew rate for the pin to fast edge rate. Note that this only applies for pins in strong output drive modes, not to resistive drive modes.

**Parameters:** pinPC: Port pin configuration register (uint16 PSoC 3, uint32 PSoC 5LP)

**Return Value:** None

### void CyPins_SlowSlew(uint16/uint32 pinPC)

**Description:** Set the slew rate for the pin to slow edge rate. Note that this only applies for pins in strong output drive modes, not to resistive drive modes.

**Parameters:** pinPC: Port pin configuration register (uint16 PSoC 3, uint32 PSoC 5LP)

**Return Value:** None

## PSoC 4 APIs

### CY_SYS_PINS_READ_PIN(portPS, pin)

**Description:** Reads the current value on the pin (pin state, PS).

**Parameters:** portPS: Address of port pin status register (uint32). Definitions for each port are provided in the *cydevice_trm.h* file in the form: CYREG_PRTx_PS, where x is a port number 0 - 4.
pin: pin number 0 – 7.

**Return Value:** Pin state:
0: Logic low value
Non-0: Logic high value

### CY_SYS_PINS_SET_PIN(portDR, pin)

**Description:** Set the output value for the pin (data register, DR) to a logic high.
Note that this only has an effect for pins configured as software pins that are not driven by hardware.

**Parameters:** portDR: Address of port output pin data register (uint32). Definitions for each port are provided in the *cydevice_trm.h* file in the form: CYREG_PRTx_DR, where x is a port number 0 - 4.
pin: pin number 0 - 7.

**Return Value:** None

### CY_SYS_PINS_CLEAR_PIN(portDR, pin)

**Description:** This macro sets the state of the specified pin to zero.

**Parameters:** portDR: Address of port output pin data register (uint32). Definitions for each port are provided in the *cydevice_trm.h* file in the form: CYREG_PRTx_DR, where x is a port number 0 - 4.
pin: pin number 0 – 7.

**Return Value:** None

## CY_SYS_PINS_SET_DRIVE_MODE(portPC, pin, mode)

**Description:** Sets the drive mode for the pin (DM).

**Parameters:** portPC: Address of port configuration register (uint32). Definitions for each port are provided in the *cydevice_trm.h* file in the form: CYREG_PRTx_PC, where x is a port number 0 - 4.
pin: pin number 0 – 7.
mode: Desired drive mode

| Define | Source |
|---|---|
| CY_SYS_PINS_DM_ALG_HIZ | Analog HiZ |
| CY_SYS_PINS_DM_DIG_HIZ | Digital HiZ |
| CY_SYS_PINS_DM_RES_UP | Resistive pull up |
| CY_SYS_PINS_DM_RES_DWN | Resistive pull down |
| CY_SYS_PINS_DM_OD_LO | Open drain - drive low |
| CY_SYS_PINS_DM_OD_HI | Open drain - drive high |
| CY_SYS_PINS_DM_STRONG | Strong CMOS Output |
| CY_SYS_PINS_DM_RES_UPDWN | Resistive pull up/down |

**Return Value:** None

## CY_SYS_PINS_READ_DRIVE_MODE(portPC, pin)

**Description:** Reads the drive mode for the pin (DM).

**Parameters:** portPC: Address of port configuration register (uint32). Definitions for each port are provided in the *cydevice_trm.h* file in the form: CYREG_PRTx_PC, where x is a port number 0 - 4.
pin: pin number 0 – 7.

**Return Value:** Current drive mode for the pin

| Define | Source |
|---|---|
| CY_SYS_PINS_DM_ALG_HIZ | Analog HiZ |
| CY_SYS_PINS_DM_DIG_HIZ | Digital HiZ |
| CY_SYS_PINS_DM_RES_UP | Resistive pull up |
| CY_SYS_PINS_DM_RES_DWN | Resistive pull down |
| CY_SYS_PINS_DM_OD_LO | Open drain - drive low |
| CY_SYS_PINS_DM_OD_HI | Open drain - drive high |
| CY_SYS_PINS_DM_STRONG | Strong CMOS Output |
| CY_SYS_PINS_DM_RES_UPDWN | Resistive pull up/down |

# 7 Register Access

A library of macros provides read and write access to the registers of the device. These macros are used with the defined values made available in the generated *cydevice_trm.h* and *cyfitter.h* files. Access to registers should be made using these macros and not the functions that are used to implement the macros. This allows for device independent code generation.

PSoC 3 is an 8-bit architecture, so the processor does not have endianness. However, the compiler for an 8-bit architecture will implement endianness. For PSoC 3, the Keil compiler implements a big endian (MSB in lowest address) ordering. The PSoC 4/PSoC 5LP processor architectures use little endian ordering.

SRAM and Flash storage in all architectures is done using the endianness of the architecture and compilers. However, the registers in all these chips are laid out in little endian order. These macros allow register accesses to match this little endian ordering. If you perform operations on multi-byte registers without using these macros, you must consider the byte ordering of the specific architecture. Examples include usage of DMA to transfer between memory and registers, as well as function calls that are passed an array of bytes in memory.

The PSoC 3 is an 8-bit processor, so all accesses will be done a byte at a time. The PSoC 5LP will perform accesses using the appropriate 8-, 16- and 32-bit accesses. The PSoC 4 requires these accesses to be aligned to the width of the transaction.

The PSoC 4 requires peripheral register accesses to match the hardware register size. Otherwise, the peripheral might ignore the transfer and Hard Fault exception will be generated.

## APIs

### uint8 CY_GET_REG8(uint16/uint32 reg)

**Description:** Reads the 8-bit value from the specified register. For PSoC 3, the address must be in the lower 64 K address range.

**Parameters:** reg: Register address (uint16 PSoC 3, uint32 PSoC 4/PSoC 5LP)

**Return Value:** Read value

### void CY_SET_REG8(uint16/uint32 reg, uint8 value)

**Description:** Writes the 8-bit value to the specified register. For PSoC 3 the address must be in the lower 64 K address range.

**Parameters:** reg: Register address (uint16 PSoC 3, uint32 PSoC 4/PSoC 5LP)

value: Value to write

**Return Value:** None

### uint16 CY_GET_REG16(uint16/uint32 reg)

**Description:** Reads the 16-bit value from the specified register. This macro implements the byte swapping required for proper operation. For PSoC 3 the address must be in the lower 64 K address range.

**Parameters:** reg: Register address (uint16 PSoC 3, uint32 PSoC 4/PSoC 5LP)

**Return Value:** Read value

### void CY_SET_REG16(uint16/uint32 reg, uint16 value)

**Description:** Writes the 16-bit value to the specified register. This macro implements the byte swapping required for proper operation. For PSoC 3 the address must be in the lower 64 K address range.

**Parameters:** reg: Register address (uint16 PSoC 3, uint32 PSoC 4/PSoC 5LP)

value: Value to write

**Return Value:** None

### uint32 CY_GET_REG24(uint16/uint32 reg)

**Description:** Reads the 24-bit value from the specified register. This macro implements the byte swapping required for proper operation. For PSoC 3 the address must be in the lower 64 K address range.

**Parameters:** reg: Register address (uint16 PSoC 3, uint32 PSoC 4/PSoC 5LP)

**Return Value:** Read value

### void CY_SET_REG24(uint16/uint32 reg, uint32 value)

**Description:** Writes the 24-bit value to the specified register. This macro implements the byte swapping required for proper operation. For PSoC 3 the address must be in the lower 64 K address range.

**Parameters:** reg: Register address (uint16 PSoC 3, uint32 PSoC 4/PSoC 5LP)

value: Value to write

**Return Value:** None

### uint32 CY_GET_REG32(uint16/uint32 reg)

**Description:** Reads the 32-bit value from the specified register. This macro implements the byte swapping required for proper operation. For PSoC 3 the address must be in the lower 64 K address range.

**Parameters:** reg: Register address (uint16 PSoC 3, uint32 PSoC 4/PSoC 5LP)

**Return Value:** Read value

### void CY_SET_REG32(uint16/uint32 reg, uint32 value)

**Description:** Writes the 32-bit value to the specified register. This macro implements the byte swapping required for proper operation. For PSoC 3 the address must be in the lower 64 K address range.

**Parameters:** reg: Register address (uint16 PSoC 3, uint32 PSoC 4/PSoC 5LP)

value: Value to write

**Return Value:** None

### uint8 CY_GET_XTND_REG8(uint32 reg)

**Description:** Reads the 8-bit value from the specified register. Supports the full address space for PSoC 3, but requires more execution cycles than the standard register get function. Identical to CY_GET_REG8 for PSoC 4/PSoC 5LP.

**Parameters:** reg: Register address

**Return Value:** Read value

### void CY_SET_XTND_REG8(uint32 reg, uint8 value)

**Description:** Writes the 8-bit value to the specified register. Supports the full address space for PSoC 3, but requires more execution cycles than the standard register set function. Identical to CY_SET_REG8 for PSoC 4/PSoC 5LP.

**Parameters:** reg: Register address

value: Value to write

**Return Value:** None

### uint16 CY_GET_XTND_REG16(uint32 reg)

**Description:** Reads the 16-bit value from the specified register. This macro implements the byte swapping required for proper operation. Supports the full address space for PSoC 3, but requires more execution cycles than the standard register get function. Identical to CY_GET_REG16 for PSoC 4/PSoC 5LP.

**Parameters:** reg: Register address

**Return Value:** Read value

### void CY_SET_XTND_REG16(uint32 reg, uint16 value)

**Description:** Writes the 16-bit value to the specified register. This macro implements the byte swapping required for proper operation. Supports the full address space for PSoC 3, but requires more execution cycles than the standard register set function. Identical to CY_SET_REG16 for PSoC 4/PSoC 5LP.

**Parameters:** reg: Register address

value: Value to write

**Return Value:** None

## uint32 CY_GET_XTND_REG24(uint32 reg)

**Description:** Reads the 24-bit value from the specified register. This macro implements the byte swapping required for proper operation. Supports the full address space for PSoC 3, but requires more execution cycles than the standard register get function. Identical to CY_GET_REG24 for PSoC 4/PSoC 5LP.

**Parameters:** reg: Register address

**Return Value:** Read value

## void CY_SET_XTND_REG24(uint32 reg, uint32 value)

**Description:** Writes the 24-bit value to the specified register. This macro implements the byte swapping required for proper operation. Supports the full address space for PSoC 3, but requires more execution cycles than the standard register set function. Identical to CY_SET_REG24 for PSoC 4/PSoC 5LP.

**Parameters:** reg: Register address

**value:** Value to write

**Return Value:** None

## uint32 CY_GET_XTND_REG32(uint32 reg)

**Description:** Reads the 32-bit value from the specified register. This macro implements the byte swapping required for proper operation. Supports the full address space for PSoC 3, but requires more execution cycles than the standard register get function. Identical to CY_GET_REG32 for PSoC 4/PSoC 5LP.

**Parameters:** reg: Register address

**Return Value:** Read value

## void CY_SET_XTND_REG32(uint32 reg, uint32 value)

**Description:** Writes the 32-bit value to the specified register. This macro implements the byte swapping required for proper operation. Supports the full address space for PSoC 3, but requires more execution cycles than the standard register set function. Identical to CY_SET_REG32 for PSoC 4/PSoC 5LP.

**Parameters:** reg: Register address

value: Value to write

**Return Value:** None

# 8 DMA

The DMA files provide the API functions for the DMA controller, DMA channels and Transfer Descriptors. This API is the library version, not the code that is generated when the user places a DMA component on the schematic. The automatically generated code would use the APIs in this module.

Refer to the DMA component datasheet for more information.

**Note** The linked list of all the Transfer Descriptors to be allocated is created (by CyDmacConfigure() function call from the startup code) only if a DMA component is placed onto the schematic.

PSoC 4 devices have no DMA capability.

This page intentionally left blank.

# 9 Flash and EEPROM

## PSoC 3/PSoC 5LP Implementation

### Flash Architecture

Flash memory in PSoC devices provides nonvolatile storage for user firmware, user configuration data, bulk data storage, and optional error correcting code (ECC) data. The main flash memory area contains up to 256 KB of user program space, depending on the device type.

Flash is organized as a set of arrays. Each array consists of 64, 128, or 256 rows. Each row contains 256 data bytes plus 32 bytes of ECC area. If ECC is not used, this space can store device configuration data and bulk user data. User code may not be run out of the ECC flash memory section.

**Flash Memory Array Structure**

| | | |
|---|---|---|
| **Row 0** | Data (256 bytes) | ECC (32 bytes) |
| **Row 1** | Data | ECC |
| | • • • • • | |
| **Row N** | Data | ECC |

PSoC 3 flash memory has the following features:

- organized as one array of 64, 128, or 256 rows;
- each row contains 256 data bytes plus 32 bytes for either ECC or data storage.

PSoC 5LP flash memory has the following features:

- organized as either one array of 128 or 256 rows, or as multiple arrays of 256 rows each;
- each row contains 256 data bytes plus 32 bytes for either ECC or data storage.

See the device datasheet and TRM for more information on Flash architecture.

The **System** tab of the PSoC Creator Design-Wide Resources (DWR) file contains configuration options that define ECC area utilization:

| DWR Option | ECC Function |
|---|---|
| Enable Error Correcting Code (ECC) | ECC corrects one bit error and detect multiple bit errors per 8 bytes. ECC area stores error correcting code data. |

| DWR Option | ECC Function |
|---|---|
| Store Configuration Data in ECC Memory | The device configuration data will be stored in ECC area to reduce main FLASH memory usage. Error correction may not be used when this option is enabled.<br>**Note** This option is always disabled for bootloader projects, as ECC area is dedicated for bootloadable projects. |
| None | ECC function is disabled. The user data can be stored in the ECC area. |

For more information on using ECC, refer to the *Flash Program Memory* chapter of the TRM.

PSoC devices include a flexible flash-protection model that prevents access and visibility to on-chip flash memory. The device offers the ability to assign one of four protection levels to each row of flash:

- Unprotected
- Factory Upgrade
- Field Upgrade
- Full Protection

The required protection level can be selected using the **Flash Security** tab of the PSoC Creator DWR file. Flash protection levels can only be changed by performing a complete flash erase. The Flash programming APIs will fail to write a row with Full Protection level. For more information on protection model, refer to the *Flash Security Editor* section in the PSoC Creator Help.

## EEPROM Architecture

PSoC EEPROM memory is byte-addressable nonvolatile memory. The EEPROM is also organized as a set of arrays. Both PSoC 3 and PSoC 5LP architectures have one EEPROM array, the size of which is 512 bytes, 1 KB, or 2 KB. The array consists of 32, 64, or 128 rows, depending on the device. Each row contains 16 bytes of data.

## Working with Flash and EEPROM

Flash and EEPROM are mapped into memory space and can be read directly. To get the address of the first Flash / EEPROM row in a specified array ID, the array ID should be multiplied by array size, and added to Flash / EEPROM base address. To access any row in the same array ID, the size of the row should be multiplied by the desired row number and added to the first row address of the specified array.

**Note** When writing Flash, data in the instruction cache can become stale. Therefore, the cache data does not correlate to the data just written to Flash. A call to CyFlushCache() is required to invalidate the data in cache and force fresh information to be loaded from Flash.

The following table provides definitions of the device-specific Flash parameters:

| Value | Description |
|---|---|
| CY_FLASH_BASE | The base address of the Flash memory. |
| CY_FLASH_SIZE | The size of the Flash memory in bytes. |
| CY_FLASH_SIZEOF_ARRAY | The size of a Flash array in bytes. |
| CY_FLASH_SIZEOF_ROW | The size of a Flash row in bytes. |
| CY_FLASH_SIZEOF_ECC_ROW | The size of the ECC row in bytes. |
| CY_FLASH_NUMBER_ROWS | The number of Flash rows. |
| CY_FLASH_NUMBER_ARRAYS | The number of Flash arrays. |

The EEPROM API provides the following device-specific definitions:

| Value | Description |
|---|---|
| CY_EEPROM_BASE | The base address of the EEPROM memory. |
| CY_EEPROM_SIZE | The size of the EEPROM memory in bytes. |
| CY_EEPROM_SIZEOF_ARRAY | The size of an EEPROM array in bytes. |
| CY_EEPROM_SIZEOF_ROW | The size of an EEPROM row in bytes. |
| CY_EEPROM_NUMBER_ROWS | The number of EEPROM rows. |
| CY_EEPROM_NUMBER_ARRAYs | The number of EEPROM arrays. |

Both Flash and EEPROM are programmed through the system performance controller (SPC). To interface with the SPC, information is pushed into, and pulled from, a single register. The Flash/EEPROM specific API provides unified approach to work with Flash as well as EEPROM and simplifies interacting with the SPC by abstracting the details away.

In PSoC 3/PSoC 5LP devices, flash can be read either by the cache controller or the SPC. Flash write can be performed only by SPC. Both SPC and cache cannot simultaneously access the flash memory. If the cache controller tries to access flash at the same time as the SPC, then it must wait until the SPC completes its flash access operation. The CPU, which accesses the flash memory through the cache controller, is therefore also stalled in this circumstance. If a CPU code fetch has to be done from the flash memory due to a cache miss condition, then the cache would have to wait till the SPC completes the flash write operation. Thus the CPU code execution will also be halted till the flash write is complete.

It can take as many as 20 milliseconds to write to EEPROM or Flash. During this time the device should not be reset, or unexpected changes may be made to portions of EEPROM or Flash. Reset sources include XRES pin, software reset, and watchdog; care should be taken to make sure that these are not inadvertently activated. Also, the low voltage detect circuits should be configured to generate an interrupt instead of a reset.

PSoC devices have an on-chip temperature sensor that is used to measure the internal die temperature. You must acquire the temperature at least once to use Flash and EEPROM write functions. If the application will be used in an environment where the die temperature changes 10 °C or more, the temperature should be refreshed to adjust the write times to the Flash for optimal performance. The die temperature is obtained by calling the CySetTemp() function. This function queries SPC for the die temperature and stores it in a global variable, which is used implicitly while performing Flash and EEPROM write operations.

When programming Flash with error detection/correction function disabled (ECC flash space is used for data storage), there are multiple methods for writing a row of data:

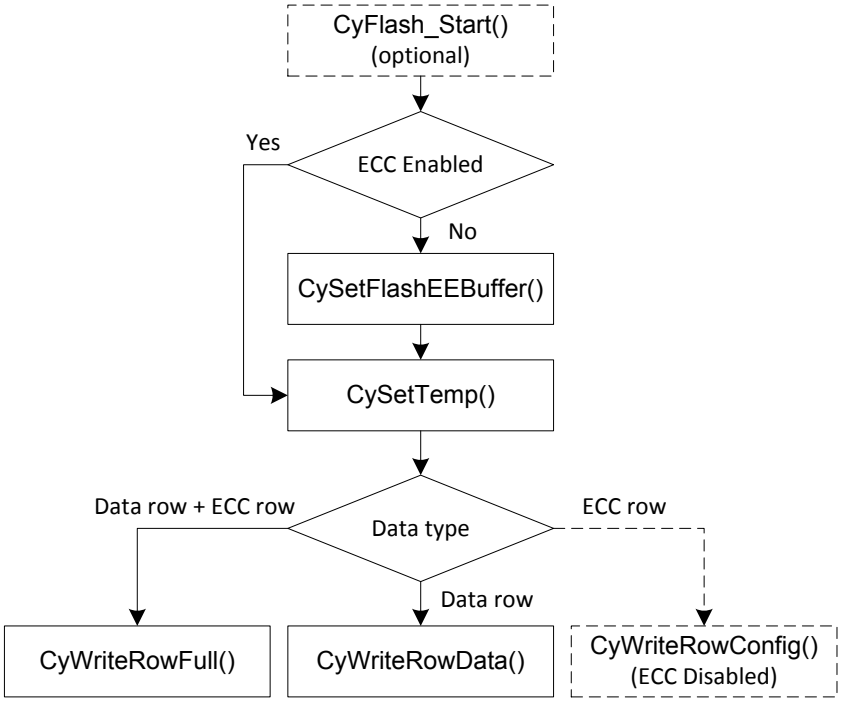
- Use CyWriteRowFull() to write the entire row including ECC;
- Use CyWriteRowData() to write the entire row without ECC;
- Use CyWriteRowConfig() to write just the ECC memory.

If the Flash ECC feature is disabled, you also need to allocate the buffer and pass it into the CySetFlashEEBuffer() function for both Flash and EEPROM programming. This buffer is used to store intermediate data while communicating with the SPC. For more information on the buffer allocation, refer to the CySetFlashEEBuffer() function description in the APIs section of this chapter.

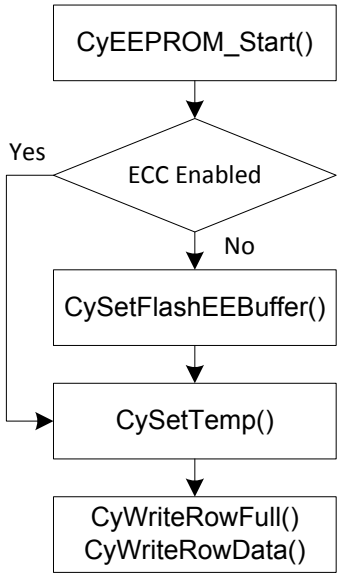
Flash or EEPROM can be written by one row at a time by calling the CyWriteRowData() function. The first parameter determines the Flash or EEPROM array. The number of arrays that are Flash and the number

of arrays that are EEPROM are specific to the exact device selected. Refer to device TRM to determine which array IDs are valid. The row numbering starts from 0 for each array ID.

**Flash Programming Diagram**




**EEPROM Programming Diagram**



### *Power Modes*

For PSoC 3/PSoC 5LP devices, the power manager will not put the device into a low power state if the system performance controller (SPC) is executing a command. The device will go into low power mode after the SPC completes command execution.

## Flash and EEPROM APIs

### *cystatus CySetTemp()*

**Description:** Updates the static snapshot of current chip temperature value obtained from on-chip temperature sensor. This function must be called once before executing a series of Flash / EEPROM writing functions. In case the application will be used in an environment where the die temperature changes significantly (10 °C or more), care should be taken to keep the temperature snapshot value up-to-date to adjust the write times to the Flash for optimal performance.

**Parameters:** None

**Return Value:** Status

| Value | Description |
|---|---|
| CYRET_SUCCESS | Successful |
| CYRET_LOCKED | Flash / EEPROM writing already in use |
| CYRET_UNKNOWN | Failure |

**Side Effects and Restrictions:** The function does not return until the SPC has returned to an idle state.

### *cystatus CySetFlashEEBuffer(uint8 *buffer)*

**Description:** Sets the buffer used for temporary storage of a complete row of flash plus associated ECC used during writes to Flash and EEPROM. This buffer is only necessary when Flash ECC is disabled.

**Parameters:** uint8 *buffer: Address of the allocated buffer with the size that equals sum of flash and ECC row sizes.

**Return Value:** Status

| Value | Description |
|---|---|
| CYRET_SUCCESS | Successful |
| CYRET_LOCKED | Flash / EEPROM writing already in use |

### *cystatus CyWriteRowFull(uint8 arrayId, uint16 rowAddress, uint8 *rowData, uint16 rowSize)*

**Description:** Allows a row to be erased and programmed.
If the array is a Flash array:

| DWR Flash configuration | Description |
|---|---|
| Enable ECC – ON<br>Store Configuration Data in ECC Memory – N/A | Data are written to the flash row. The ECC for these data are calculated and written automatically.<br>The size of the data equals the size of the flash row. |
| Enable ECC – OFF<br>Store Configuration Data in ECC Memory – ON | Data are written to both flash and ECC rows.<br>To prevent overwriting of the configuration data stored in the ECC flash space, the size of the data must be equal to the size of the flash row. |
| Enable ECC – OFF<br>Store Configuration Data in ECC Memory – OFF | Data are written to both flash and ECC rows.<br>The size of the data equals the sum of flash row and ECC row sizes. |

If the array is an EEPROM array, the size of data equals EEPROM row size.

**Parameters:** uint8 arrayId: ID of the array to write. The type of write, Flash or EEPROM, is determined from the array ID. The arrays in the part are sequential starting at the first ID for the specific memory type. The array ID for the Flash memory lasts from 0x00 to 0x3F and for the EEPROM memory it lasts from 0x40 to 0x7F.

uint16 rowAddress: Row address within the specified arrayId.

uint8 *rowData: Address of the data to be programmed. **Note** This cannot be the same buffer as allocated by CySetFlashEEBuffer() function.

uint16 rowSize: Number of bytes of row data

**Return Value:** Status

| Value | Description |
|---|---|
| CYRET_SUCCESS | Successful |
| CYRET_LOCKED | Flash / EEPROM writing already in use |
| CYRET_CANCELED | Command not accepted |
| Other non-zero | Failure |

### *cystatus CyWriteRowData(uint8 arrayId, uint16 rowAddress, uint8 *rowData)*

**Description:** Writes a row of Flash or EEPROM.
If the array is a Flash array:

| DWR Flash configuration | Description |
|---|---|
| Enable ECC – ON<br>Store Configuration Data in ECC Memory – N/A | Data are written to the flash row. The ECC for these data are calculated and written automatically.<br>The size of the data that is passed to this function equals the size of the flash row. |
| Enable ECC – OFF<br>Store Configuration Data in ECC Memory – ON/OFF | Data are written to the flash memory.<br>The data stored in ECC memory are preserved using the buffer provided by CySetFlashEEBuffer() function. |

If the array is an EEPROM array, the size of data equals EEPROM row size.

**Parameters:** uint8 arrayId: ID of the array to write. The type of write, Flash or EEPROM, is determined from the array ID. The arrays in the part are sequential starting at the first ID for the specific memory type. The array ID for the Flash memory lasts from 0x00 to 0x3F and for the EEPROM memory it lasts from 0x40 to 0x7F.

uint16 rowAddress: Row address within the specified arrayId.

uint8 *rowData: Address of the data to be programmed.

**Return Value:** Status

| Value | Description |
|---|---|
| CYRET_SUCCESS | Successful |
| CYRET_LOCKED | Flash / EEPROM writing already in use |
| CYRET_CANCELED | Command not accepted |
| Other non-zero | Failure |

### *cystatus CyWriteRowConfig(uint8 arrayId, uint16 rowAddress, uint8 *rowECC)*

**Description:** Writes the ECC portion of a Flash. This function is only valid for Flash array IDs (not for EEPROM).

| DWR Flash configuration | Description |
|---|---|
| Enable ECC – ON<br>Store Configuration Data in ECC Memory – N/A | This function is not available for this configuration as the ECC is stored in the ECC memory. |
| Enable ECC – OFF<br>Store Configuration Data in ECC Memory – ON | This function is not available for this configuration as the Configuration Data are stored in the ECC memory. |
| Enable ECC – OFF<br>Store Configuration Data in ECC Memory – OFF | Data are written to the ECC row.<br>The data stored in Flash row are preserved using the buffer provided by CySetFlashEEBuffer() function. |

**Parameters:** uint8 arrayId: ID of the array to write. The arrays in the part are sequential starting at the first ID for the specific memory type. The array ID for the Flash memory lasts from 0x00 to 0x3F.

uint16 rowAddress: Row address within the specified arrayId.

uint8 *rowECC: Address of the data to be programmed.

**Return Value:** Status

| Value | Description |
|---|---|
| CYRET_SUCCESS | Successful |
| CYRET_LOCKED | Flash / EEPROM writing already in use |
| CYRET_CANCELED | Command not accepted |
| Other non-zero | Failure |

### *void CyFlash_Start()*

**Description:** Enables the Flash. By default Flash is enabled.

**Parameters:** None

**Return Value:** None

### *void CyFlash_Stop()*

**Description:** Disables the Flash. This setting is ignored as long as the CPU is currently running. This will only take effect when the CPU is later disabled.

**Parameters:** None

**Return Value:** None

### *void CyFlash_SetWaitCycles(uint8 freq)*

**Description:** Sets the number of clock cycles the cache will wait before it samples data coming back from Flash. This function must be called before increasing CPU clock frequency. It can optionally be called after lowering CPU clock frequency in order to improve CPU performance.

**Parameters:** freq: CPU operation frequency in Megahertz.

**Return Value:** None

### *void CyEEPROM_Start()*

**Description:** Enables the EEPROM.
The EEPROM is controlled by a separate bit and must be started before it can be used.

**Parameters:** None

**Return Value:** None

### *void CyEEPROM_Stop()*

**Description:** Disables the EEPROM.
The EEPROM is controlled by a separate bit and can be stopped independently.

**Parameters:** None

**Return Value:** None

### *void CyEEPROM_ReadReserve()*

**Description:** Request access to the EEPROM for reading and waits until that access is available. The access to EEPROM is arbitrated between the controller that writes to the EEPROM and the normal access to read from EEPROM. It is not required to reserve access to the EEPROM for reading, but if a write is still active and a read is attempted a fault is generated and the wrong data is returned.

**Parameters:** None

**Return Value:** None

### *void CyEEPROM_ReadRelease()*

**Description:** Releases the read reservation of the EEPROM. If the EEPROM has been reserved for reading, then it must be released before further writes to the EEPROM can be performed.

**Parameters:** None

**Return Value:** None

# PSoC 4 Implementation

## Memory Architecture

PSoC 4 does not support EEPROM or the Flash ECC feature. The configuration of the Flash memory varies by device, so the device-specific constants provided for the device must be used.

The following table provides definitions of the device-specific Flash parameters:

| Value | Description |
|---|---|
| CY_FLASH_BASE | The base pointer of the Flash memory. |
| CY_FLASH_SIZE | The size of the Flash memory in bytes. |
| CY_FLASH_SIZEOF_ARRAY | The size of a Flash array in bytes. |

| Value | Description |
|---|---|
| CY_FLASH_SIZEOF_ROW | The size of a Flash row in bytes. |
| CY_FLASH_NUMBER_ROWS | The number of Flash rows. |
| CY_FLASH_NUMBER_ARRAYS | The number of Flash arrays. |

## Working with Flash

Flash programming operations are implemented as system calls. System calls are executed out of SROM in the privileged mode of operation. Users have no access to read or modify the SROM code. The CPU requests the system call by writing the function opcode and parameters to the System Performance Controller (SPC) input registers, and then requesting the SROM to execute the function. Based on the function opcode, the SPC executes the corresponding system call from SROM and updates the SPC status register. The CPU should read this status register for the pass/fail result of the function execution. As part of function execution, the code in SROM interacts with the SPC interface to do the actual flash programming operations.

It can take as many as 20 milliseconds to write to flash. During this time, the device should not be reset, or unexpected changes may be made to portions of the flash. Reset sources include the XRES pin, a software reset, and the watchdog. Make sure that these are not inadvertently activated. Also, the low voltage detect circuits should be configured to generate an interrupt instead of a reset.

The flash can be read either by the cache controller or the SPC. Flash writes can be performed only by the SPC. Both the SPC and the cache cannot simultaneously access flash memory. If the cache controller tries to access flash at the same time as the SPC, then it must wait until the SPC completes its flash access operation. The CPU, which accesses the flash memory through the cache controller, is therefore also stalled in this circumstance. If a CPU code fetch has to be done from flash memory due to a cache miss condition, then the cache would have to wait until the SPC completes the flash write operation. Thus the CPU code execution will also be halted until the flash write is complete.

Flash is directly mapped into memory space and can be read directly.

Note: Flash write operations on the PSoC 4000 devices modify the clock settings of the device during the period of the write operation. Refer to the CySysFlashWriteRow() API documentation for details.

## Flash APIs

### *cystatus CySysFlashWriteRow(uint32 rowNum, const uint8 rowData[])*

**Description:** Writes a row of Flash.

**Parameters:** uint32 rowNum: Row number. The valid range is dependent on the device.

uint8 rowData: Array of bytes for a complete row to write.

**Return Value:** Status:

| Value | Description |
|---|---|
| CYRET_SUCCESS | Successful |
| CYRET_LOCKED | Flash already in use |
| CYRET_CANCELED | Command not accepted |
| CYRET_BAD_PARAM | One or more invalid parameters |
| Other non-zero | Failure |

**Side Effects and Restrictions:** The IMO must be enabled before calling this function. The operation of the flash writing hardware is dependent on the IMO.

For PSoC 4000 devices this API will automatically modify the clock settings for the device. Writing to flash requires that changes be made to the IMO and HFCLK settings. The configuration is restored before returning. HFCLK will have several frequency changes during the operation of this API between a minimum frequency of the current IMO frequency divided by 8 and a maximum frequency of 12 MHz. This will impact the operation of most of the hardware in the device.

### *void CySysFlashSetWaitCycles(uint32 freq)*

**Description:** Sets the number of clock cycles the cache will wait before it samples data coming back from Flash. This function must be called before increasing the SYSCLK clock frequency. It can optionally be called after lowering the SYSCLK clock frequency in order to improve CPU performance.

**Parameters:** freq: Valid range [3-48]. Frequency for operation of the IMO.
Note: Invalid frequencies will be ignored.

**Return Value:** None

**Side Effects and Restrictions:** None

This page intentionally left blank.

# 10 Bootloader Migration

This chapter covers information about the bootloader architecture introduced with PSoC Creator 2.1, as well as various known issues that you may encounter when migrating designs with the bootloader system from PSoC Creator 2.0 or earlier.

## Introduction

Beginning with PSoC Creator 2.1, the bootloader system has been reorganized to provide more configuration options. In previous releases, the bootloader system was part of the cy_boot component (a required component that is automatically and invisibly instantiated in all designs). The bootloader system now includes two independent Bootloader and Bootloadable components. You can find the components in the PSoC Creator Component Catalog, and they include component datasheets like all other components.

Also, the bootloader system configuration options have been moved from the PSoC Creator Design-Wide Resources (DWR) file into the corresponding Bootloader and Bootloadable component configuration dialogs. See Migrating Bootloader Designs and Migrating Bootloadable Designs sections in this chapter for more details.

The switch to the new bootloader system is performed automatically by updating the cy_boot component to version 3.0 or above. However, some changes may impact your existing bootloader and bootloadable designs. This will require maintenance when moving to the new bootloader system architecture. The bootloader system configuration settings must be moved manually.

The three key migration alternatives include:

- Complete Migration to PSoC Creator 2.1 or later
- Migrating to PSoC Creator 2.1 or later without cy_boot Component Update
- Migrating to PSoC Creator 2.1 or later for Bootloadable Designs

### Complete Migration to PSoC Creator 2.1 or later

In this scenario, PSoC Creator 2.1 or later is used to work with the latest version of the cy_boot component and new bootloader architecture.

When you open a project that was last saved in a previous PSoC Creator release, you will be prompted to update components to the latest version.

- Use the Component Update Tool and choose the latest version of the cy_boot component. It is recommended that all components be updated together. Use the “Update All to Latest” button to ensure the newest versions are selected for update.

- Open the bootloader system design and place a Bootloader component onto the schematic for a **Bootloader** or **Multi-application bootloader** application type. Place a Bootloadable component onto the schematic for a **Bootloadable** application type. See Project Application Type Property Changes for more information.
- Transfer the bootloader system settings from the DWR file to the component configuration dialogs. See Migrating Bootloader Designs and Migrating Bootloadable Designs for details.

### Migrating to PSoC Creator 2.1 or later without cy_boot Component Update

In this scenario, PSoC Creator 2.1 or later is used to work with the bootloader system design that was created in previous PSoC Creator releases. No any additional care is required in order to continue working with the bootloader system.

All the components present on the design schematic can be updated to the latest available versions, except the cy_boot component. The cy_boot component cannot be updated to the version 3.0 or above without switching to the new bootloader architecture at the same time.

**Note** Some of the latest component versions shipped with PSoC Creator 2.1 or later require cy_boot version 3.0 or above, so you cannot update them without updating cy_boot.

### Migrating to PSoC Creator 2.1 or later for Bootloadable Designs Only

In this scenario, only the bootloadable project is migrated to PSoC Creator 2.1 or later and cy_boot 3.0 or above. Existing bootloader designs are completely compatible with the newly created or migrated bootloadable designs. PSoC Creator 2.1 or later without an updated cy_boot component or previous PSoC Creator releases can be used for making changes to the bootloader design.

## Migrating Bootloader Designs

The following table shows the correlation between the bootloader system in previous PSoC Creator releases and the bootloader system introduced in PSoC Creator 2.1.

| **PSoC Creator releases before 2.1 (Bootloader section in the System tab of the DWR)** | **PSoC Creator 2.1 or later with cy_boot 3.0 or above (Bootloader component)** |
|---|---|
| IO Component | Communication component |
| Set by **Application Type** property of the design. Refer to the Project Application Type Property Changes section. | Multi-application bootloader |
| Wait for Command | Wait for command |
| Wait for Command Time (ms) | Wait for command time (ms) |
| Version | Bootloader application version |
| Packet Checksum Type | Packet checksum type |
| Fast Application Verification | Fast bootloadable application validation |
| N/A | Bootloader application validation |
| N/A | Optional commands |

For more details about the PSoC Creator bootloader system features, refer to the Bootloader component datasheet.

## Migrating Bootloadable Designs

The following table shows the correlation between the bootloadable system introduced with PSoC Creator 2.1 versus releases before PSoC Creator 2.1.

| PSoC Creator releases before 2.1 (Bootloadable section in the System tab of the DWR) | PSoC Creator 2.1 or later with cy_boot 3.0 or above (Bootloadable component) |
|---|---|
| Version | Application version |
| Application ID | Application ID |
| Custom ID | Application custom ID |
| N/A | Manual application image placement |
| N/A | Placement address |
| Set in the **Bootloader** tab of the project's Dependencies window. Refer to the Project Application Type Property Changes section. | Bootloader hex file (**Dependencies** tab of the bootloader component's Configure dialog). |

The CyBtldr_Load() function previously declared in the Bootloadable project was removed. Use the Bootloadable_1_Load() function instead, where Bootloadable_1 is the instance name of the Bootloadable component.

For more details about the PSoC Creator bootloader system features, refer to the Bootloadable component datasheet.

## Project Application Type Property Changes

In PSoC Creator releases prior to version 2.1, the **Application Type** option was used to choose the type of the application that will be produced by a build of a project. Beginning with PSoC Creator 2.1, the **Application Type** project property provides a way to verify that the design is created as intended.

The following table correlates the **Application Type** option value with the placed component configuration:

| Application Type | Expected Component |
|---|---|
| Bootloader | Bootloader component with Multi-application bootloader option disabled. |
| Multi App Bootloader | Bootloader component with Multi-application bootloader option enabled. |
| Bootloadable | Bootloadable component. |

The **Application Type** option is set in the advanced section of the **New Project** dialog and can be modified later in the project's **Build Settings** window, under the **Code Generation** section.

## Migrating Bootloadable Dependency to Bootloader Design

In PSoC Creator releases prior to version 2.1, the **Bootloader** tab of the project's Dependencies window was used to link the bootloadable project to the bootloader project. With the introduction of the Bootloadable component in PSoC Creator 2.1 with cy_boot version 3.0 or above, this option was moved to the **Dependencies** tab of the Bootloadable component Configure dialog. Refer to the component datasheet for more information.

This page intentionally left blank.

# 11 System Functions



These functions apply to all architectures.

## General APIs

### uint8 CyEnterCriticalSection(void)

**Description:** CyEnterCriticalSection disables interrupts and returns a value indicating whether interrupts were previously enabled (the actual value depends on the device architecture).

**Note** Implementation of CyEnterCriticalSection manipulates the IRQ enable bit with interrupts still enabled. The test and set of the interrupt bits is not atomic; this is true for all architectures. Therefore, to avoid corrupting the processor state, it must be the policy that all interrupt routines restore the interrupt enable bits as they were found on entry.

**Parameters:** None

**Return Value:** uint8

PSoC 3 – Returns a value containing two bits:
- bit 0: 1 if interrupts were enabled before CyEnterCriticalSection was called.
- bit 1: 1 if IRQ generation was disabled before CyEnterCriticalSection was called.

PSoC 4/PSoC 5LP – Returns 0 if interrupts were previously enabled or 1 if interrupts were previously disabled.

### void CyExitCriticalSection(uint8 savedIntrStatus)

**Description:** CyExitCriticalSection re-enables interrupts if they were enabled before CyEnterCriticalSection was called. The argument should be the value returned from CyEnterCriticalSection.

**Parameters:** uint8 savedIntrStatus: Saved interrupt status returned by the CyEnterCriticalSection function.

**Return Value:** None

### void CYASSERT(uint32 expr)

**Description:** Macro that evaluates the expression and if it is false (evaluates to 0) then the processor is halted. This macro is evaluated unless NDEBUG is defined. If NDEBUG is defined, then no code is generated for this macro. NDEBUG is defined by default for a Release build setting and not defined for a Debug build setting.

**Parameters:** expr: Logical expression. Asserts if false.

**Return Value:** None

### void CyHalt(uint8 reason)

**Description:** Halts the CPU.

**Parameters:** reason: Value to be passed for debugging. This value may be useful to know the reason why CyHalt() was invoked.

**Return Value:** None

### void CySoftwareReset(void)

**Description:** Forces a software reset of the device.

**Parameters:** None

**Return Value:** None

## CyDelay APIs

There are four CyDelay APIs that implement simple software-based delay loops. The loops compensate for bus clock frequency.

The CyDelay functions provide a minimum delay. If the processor is interrupted, the length of the loop will be extended by as long as it takes to implement the interrupt. Other overhead factors, including function entry and exit, may also affect the total length of time spent executing the function. This will be especially apparent when the nominal delay time is small.

### void CyDelay(uint32 milliseconds)

**Description:** Delay by the specified number of milliseconds. By default the number of cycles to delay is calculated based on the clock configuration entered in PSoC Creator. If the clock configuration is changed at run-time, then the function CyDelayFreq is used to indicate the new Bus Clock frequency. CyDelay is used by several components, so changing the clock frequency without updating the frequency setting for the delay can cause those components to fail.

**Parameters:** milliseconds: Number of milliseconds to delay.

**Return Value:** None

**Side Effects and Restrictions:** CyDelay has been implemented with the instruction cache assumed enabled. When instruction cache is disabled on PSoC 5LP, CyDelay will be two times larger. For example, with instruction cache disabled CyDelay(100) would result in about 200 ms delay instead of 100 ms.

### void CyDelayUs(uint16 microseconds)

**Description:** Delay by the specified number of microseconds. By default the number of cycles to delay is calculated based on the clock configuration entered in PSoC Creator. If the clock configuration is changed at run-time, then the function CyDelayFreq is used to indicate the new Bus Clock frequency. CyDelayUs is used by several components, so changing the clock frequency without updating the frequency setting for the delay can cause those components to fail.

**Parameters:** microseconds: Number of microseconds to delay.

**Return Value:** Void

**Side Effects and Restrictions:** CyDelayUS has been implemented with the instruction cache assumed enabled. When instruction cache is disabled on PSoC 5LP, CyDelayUs will be two times larger. For example, with instruction cache disabled CyDelayUs(100) would result in about 200 us delay instead of 100 us.
If the bus clock frequency is a small non-integer number, the actual delay can be up to twice as long as the nominal value. The actual delay cannot be shorter than the nominal one.

### void CyDelayFreq(uint32 freq)

**Description:** Sets the Bus Clock frequency used to calculate the number of cycles needed to implement a delay with CyDelay. By default the frequency used is based on the value determined by PSoC Creator at build time.

**Parameters:** freq: Bus clock frequency in Hz.

0: Use the default value

non-0: Set frequency value

**Return Value:** None

### void CyDelayCycles(uint32 cycles)

**Description:** Delay by the specified number of cycles using a software delay loop.

**Parameters:** cycles: Number of cycles to delay.

**Return Value:** None

## PSoC 3/PSoC 5LP Voltage Detect APIs

Voltage monitoring circuits in these devices can be configured to generate an interrupt when Vdda or Vddd is outside a defined range. Low-voltage interrupts are available for both the analog and digital supplies, and a high-voltage interrupt is available for the analog supply. The trip levels for the low-voltage detectors are independently configurable. The trip level for the high-voltage detector is fixed at 5.75 V. The analog and digital low-voltage monitoring circuits can also be configured to reset the device instead of generating an interrupt.

Bits [2:0] in the RESET_CR1 register control whether or not the voltage monitoring circuits generate an interrupt if the supply is outside the trip level. If the LVI interrupts are enabled, then bits [7:6] in RESET_CR3 control whether or not the device is reset when a low-voltage event occurs. The LVI resets are part of the precision reset circuit and generate a momentary hardware POR reset.

The status of the voltage monitor circuits is stored in two different registers. Bits [2:0] of the RESET_SR0 register are set to 1 if a low-voltage or high-voltage event has occurred. This register is cleared when read or upon POR reset. Bits [2:0] of the RESET_SR2 register hold the real-time status of the voltage monitor circuits' outputs, which means they will only be set to '1' for the duration that the event is occurring.

The GlobalSignalRef component can be used to connect the LVI and HVI interrupt signals to other components in the project schematic, or an interrupt component if you wish to execute an interrupt on an LVI/HVI event. If "Low/High Voltage Detect (LVI/HVI)" is selected in the component, the output of GlobalSignalRef is set to 1 whenever any of the enabled LVI or HVI circuits detect an event. The output will remain at 1 as long as an LVI or HVI event is occurring. Refer to the GlobalSignalRef component datasheet for more information.

The following API functions configure and manage the voltage monitoring circuits and associated interrupt status registers. For more information on the voltage monitoring circuits, refer to the *Voltage Monitoring* section of the device TRM and the *Power Voltage Level Monitors* section of the device datasheet.

## void CyVdLvDigitEnable(uint8 reset, uint8 threshold)

**Description:** Enables the output of the digital low-voltage monitor when Vddd is at or below the trip point, selects whether or not the device generates an interrupt or resets the part upon the low voltage event, and sets the voltage trip level.

**Parameters:** **reset**: Enables device reset on digital LVI interrupt:

- Interrupt on digital LVI event
- Non-zero - Reset on digital LVI event

**threshold**: Sets the trip point of the digital low-voltage monitoring circuit in steps of approximately 250 mV in range from 1.70 V (0x00) to 5.45 V (0x0F).

**Return Value:** None

**Side Effects and Restrictions** The LVI resets are momentary. When an LVI reset occurs, the RESET_CR1 and RESET_CR3 registers are restored to their default values. This means that the LVI circuit is no longer enabled and the device exits reset. If the supply is below the trip level and firmware enables the LVI reset functionality, the device will reset again. This will continue as long as the supply is below the trip level or as long as the user enables the reset functionality of the LVI.
When any LVI reset occurs, the RESET_SR0 and RESET_SR2 status registers are cleared. This means that the Analog LVI, Digital LVI, and Analog HVI status bits are not persistent across any LVI reset.

## void CyVdLvAnalogEnable(uint8 reset, uint8 threshold)

**Description:** Enables the output of the analog low-voltage monitor when Vdda is at or below the trip point, selects whether or not the device generates an interrupt or resets the part upon the low voltage event, and sets the voltage trip level.

**Parameters:** **reset**: Enables device reset on analog LVI interrupt:

- 0 - Interrupt on analog LVI event
- Non-zero - Reset on analog LVI event.

**threshold**: Sets the trip point of the analog low-voltage monitoring circuit in steps of approximately 250 mV in range from 1.70 V (0x00) to 5.45 V (0x0F).

**Return Value:** None

**Side Effects and Restrictions** The LVI resets are momentary. When an LVI reset occurs, the RESET_CR1 and RESET_CR3 registers are restored to their default values. This means that the LVI circuit is no longer enabled and the device exits reset. If the supply is below the trip level and firmware enables the LVI reset functionality, the device will reset again. This will continue as long as the supply is below the trip level or as long as the user enables the reset functionality of the LVI.
When any LVI reset occurs, the RESET_SR0 and RESET_SR2 status registers are cleared. This means that the Analog LVI, Digital LVI, and Analog HVI status bits are not persistent across any LVI reset.

## void CyVdLvDigitDisable(void)

**Description:** Disables the digital low voltage monitor (interrupt and device resets are disabled).

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions** Interrupt and LVI resets are disabled, but pending interrupts and status bits are not cleared.

## void CyVdLvAnalogDisable(void)

**Description:** Disables the analog low voltage monitor (interrupt and device resets are disabled).

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions** Interrupt and LVI resets are disabled, but pending interrupts and status bits are not cleared.

## void CyVdHvAnalogEnable(void)

**Description:** Enables the analog high voltage monitor to generate an interrupt on 5.75 V threshold detection for Vdda.

**Parameters:** None

**Return Value:** None

## void CyVdHvAnalogDisable(void)

**Description:** Disables the analog high voltage monitor (interrupt is disabled).

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions** Interrupt and HVI resets are disabled, but pending interrupts and status bits are not cleared.

## uint8 CyVdStickyStatus(uint8 mask)

**Description:** Reads and clears the voltage detection status bits in the RESET_SR0 register. The bits are set to 1 by the voltage monitor circuit when the supply is outside the detector's trip point. They stay set to 1 until they are read or a POR / LVI / PRES reset occurs. This function uses a shadow register, so only the bits passed in the parameter will be cleared in the shadow register.

**Parameters:** mask: Bits in the RESET_SR0 shadow register to clear and return.

| Value | Definition | Register [bits] |
|---|---|---|
| CY_VD_LVID | Persistent status of digital LVI. | RESET_SR0 [0] |
| CY_VD_LVIA | Persistent status of analog LVI. | RESET_SR0 [1] |
| CY_VD_HVIA | Persistent status of analog HVI. | RESET_SR0 [2] |

**Return Value:** Status. Same enumerated bit values as used for the mask parameter. A zero is returned for bits not used in the mask parameter.

**Side Effects and Restrictions** When an LVI reset occurs, the RESET_SR0 status registers are cleared. This means that the voltage detection status bits are not persistent across an LVI reset and cannot be used to determine a reset source.

## uint8 CyVdRealTimeStatus(void)

**Description:** Reads the real-time voltage detection status bits in the RESET_SR2 register. The bits are set to 1 by the voltage monitor circuit when the supply is outside the detector's trip point, and set to 0 when the supply is inside the trip point.

**Parameters:** None

**Return Value:** Status of the LVID, LVIA, and HVIA bits in the RESET_SR2 register.

| Value | Definition | Register [bits] |
|---|---|---|
| CY_VD_LVID | Real-time status of digital LVI. | RESET_SR0 [0] |
| CY_VD_LVIA | Real-time status of analog LVI. | RESET_SR0 [1] |
| CY_VD_HVIA | Real-time status of analog HVI. | RESET_SR0 [2] |

**Side Effects and Restrictions** When an LVI reset occurs, the RESET_SR2 status registers are cleared. This means that the voltage detection status bits are not persistent across an LVI reset and cannot be used to determine a reset source.

# PSoC 4100 and 4200 Voltage Detect APIs

## void CySysLvdEnable(uint32 threshold)

**Description:** Enables the output of the low-voltage monitor when Vddd is at or below the trip point, configures device to generate an interrupt, and sets the voltage trip level.

**Parameters:** threshold: Threshold selection for Low Voltage Detect circuit. Threshold variation is +/- 2.5% from these typical voltage choices.

| Define | Voltage threshold |
|---|---|
| CY_LVD_THRESHOLD_1_75_V | 1.75 V |
| CY_LVD_THRESHOLD_1_80_V | 1.80 V |
| CY_LVD_THRESHOLD_1_90_V | 1.90 V |
| CY_LVD_THRESHOLD_2_00_V | 2.00 V |
| CY_LVD_THRESHOLD_2_10_V | 2.10 V |
| CY_LVD_THRESHOLD_2_20_V | 2.20 V |
| CY_LVD_THRESHOLD_2_30_V | 2.30 V |
| CY_LVD_THRESHOLD_2_40_V | 2.40 V |
| CY_LVD_THRESHOLD_2_50_V | 2.50 V |
| CY_LVD_THRESHOLD_2_60_V | 2.60 V |
| CY_LVD_THRESHOLD_2_70_V | 2.70 V |
| CY_LVD_THRESHOLD_2_80_V | 2.80 V |
| CY_LVD_THRESHOLD_2_90_V | 2.90 V |
| CY_LVD_THRESHOLD_3_00_V | 3.00 V |
| CY_LVD_THRESHOLD_3_20_V | 3.20 V |
| CY_LVD_THRESHOLD_4_50_V | 4.50 V |

**Return Value:** None

## void CySysLvdDisable(void)

**Description:** Disables the low voltage detection. Low voltage interrupt is disabled.

**Parameters:** None

**Return Value:** None

## uint32 CySysLvdGetInterruptSource(void)

**Description:** Gets the low voltage detection interrupt status (without clearing).

**Parameters:** None

**Return Value:** Interrupt request value:
CY_SYS_LVD_INT - Indicates an Low Voltage Detect interrupt

## void CySysLvdClearInterrupt(void)

**Description:** Clears the low voltage detection interrupt status.

**Parameters:** None

**Return Value:** None

# Cache Functionality

## PSoC 3

The PSoC 3 cache is enabled by default. It can be disabled using the PSoC Creator Design-Wide Resources System Editor. There are no defines, functions or macros for cache handling for PSoC 3.

## PSoC 5LP

### *void CyFlushCache()*

| | |
|---|---|
| **Description:** | Flushes the PSoC 5/PSoC 5LP cache by invalidating all entries. |
| **Parameters:** | None |
| **Return Value:** | None |
| **Side Effects and Restrictions:** | When writing Flash, data in the instruction cache can become stale. Therefore, the cache data does not correlate to the data just written to Flash. A call to CyFlushCache() is required to invalidate the data in cache and force fresh information to be loaded from Flash. |

## PSoC 4

PSoC 4 devices have no cache capability.

# 12 Startup and Linking

The cy_boot component is responsible for the startup of the system. The following functionality has been implemented:

- Provide the reset vector
- Setup processor for execution
- Setup interrupts
- Setup the stack including the reentrant stack for the 8051
- Configure the device
- Initialize static and global variables with initialization values
- Clear all remaining static and global variables
- Integrate with the bootloader functionality
- Preserve the reset status
- Call main() C entry point

See application note AN60616 for more details on PSoC 3 and PSoC 5LP startup.

The device startup procedure configures the device to meet datasheet and PSoC Creator project specifications. Startup begins after the release of a reset source, or after the end of a power supply ramp. There are two main portions of startup: hardware startup and firmware startup. During hardware startup, the CPU is halted, and other resources configure the device. During firmware startup, the CPU runs code generated by PSoC Creator to configure the device. When startup ends, the device is fully configured, and its CPU begins execution of user-authored main() code.

The hardware startup configures the device to meet the general performance specifications given in the datasheet. The hardware startup phase begins after a power supply ramp or reset event. There are two phases of hardware startup: reset and boot. After hardware startup ends, code execution from Flash begins.

Firmware startup configures the PSoC device to behave as described in the PSoC Creator project. It begins at the end of hardware startup. The PSoC device's CPU begins executing user-authored main() code after the completion of firmware startup. The main task of firmware startup is to populate configuration registers such that the PSoC device behaves as designed in the PSoC Creator project. This includes configuring analog and digital peripherals, as well as system resources such as clocks and routing.

The startup procedure may be altered to better fit a specific application's needs. There are two ways to modify device startup: using the PSoC Creator design-wide resources (DWR) interface, and modifying the device startup code.

## PSoC 3

Startup is all handled by a single assembly file (*KeilStart.a51*), which is based on a template provided by Keil. There isn't a file specifically associated with linking. The linker directives are used instead. For more information on the 8051 architecture, refer to the Language Extensions section on www.keil.com.

For the more information on the PSoC 3's architecture refer to the AN54181.

## PSoC 4/PSoC 5LP

The startup and linker scripts have been custom developed by Cypress, but both of the toolchain vendors that we currently support provide example linker implementations and complete libraries that solve many of the issues that have been created by our custom implementations.

The memory layout of the final binary and hex images, as well as the placement in PSoC 5LP memory is described in the linker descriptor (.scat) file generated as part of the PSoC Creator build. The custom linker descriptor file can be used when building the project instead of the default one by going to **Build Settings** window and specifying path to the file in the **Custom Linker Script** field of the **Linker** category.

For the more information on the PSoC 5LP's CPU architecture, refer to the Cortex™-M3 Technical Reference Manual on infocenter.arm.com. There is also Application Note 179 - Cortex™ -M3 Embedded Software Development on infocenter.arm.com.

For the more information on the PSoC 4's CPU architecture, refer to the Cortex™-M0 Technical Reference Manual on infocenter.arm.com.

### Unaligned Transfers (PSoC 5LP)

The PSoC 5LP supports unaligned transfers on single accesses with a number of limitations common for Cortex-M3 platform. When unaligned transfers are used, they are actually converted into multiple aligned transfers. This conversion is transparent from the software point of view. When an unaligned transfer takes place, it is broken into separate transfers. So, it takes more clock cycles for a single data access. To get the best performance and to ensure code compatibility through PSoC processors family, the unaligned transfers should be avoided.

In PSoC 5LP, SRAM is located at Cortex-M3 Code/SRAM regions boundary. The unaligned accesses that cross memory map boundaries are architecturally unpredictable. The linker configuration files used for PSoC 5LP do not protect against unaligned accesses that cross this boundary.

When unaligned accesses must be used, use a function that checks for the possible boundary problem and do byte accesses at the boundary or modify the linker script to force the memory that needs to be accessed in an unaligned fashion to not span this border.

### GCC Implementation

PSoC Creator integrates the GCC ARM Embedded compiler including making the Newlib-nano and newlib libraries. Refer to the Red Hat newlib C Library for the C library reference manual.

The newlib-nano is configured by default. To choose newlib library, open the Build Settings dialog > ARM GCC 4.7.3 > Linker > General, and set the "Use newlib-nano" option to False.

By default, with the GNU ARM compiler, the string formatting functions in the C run-time library return empty strings for floating-point conversions. The newlib-nano library is a stripped-down version of the full C newlib. It does not include support for floating point formatting and other memory-intensive features.

There are two solutions to this problem: enable floating-point formatting support in newlib-nano, or change the library to the full newlib.

To enable floating-point formatting, open the Build Settings dialog, go to the Linker page, and add the string `-u _printf_float` to the command line options. This change will result in an increase in Flash and RAM usage in your application.

**Note** If you also wish to use the scanf functions with floating-point numbers you should add the string `-u _scanf_float` as well, with another increase in Flash and RAM usage.

## Realview Implementation (applicable for MDK and RVDS)

Use all the standard libraries (C standardlib, C microlib, fplib, mathlib). All of these libraries are linked in by default.

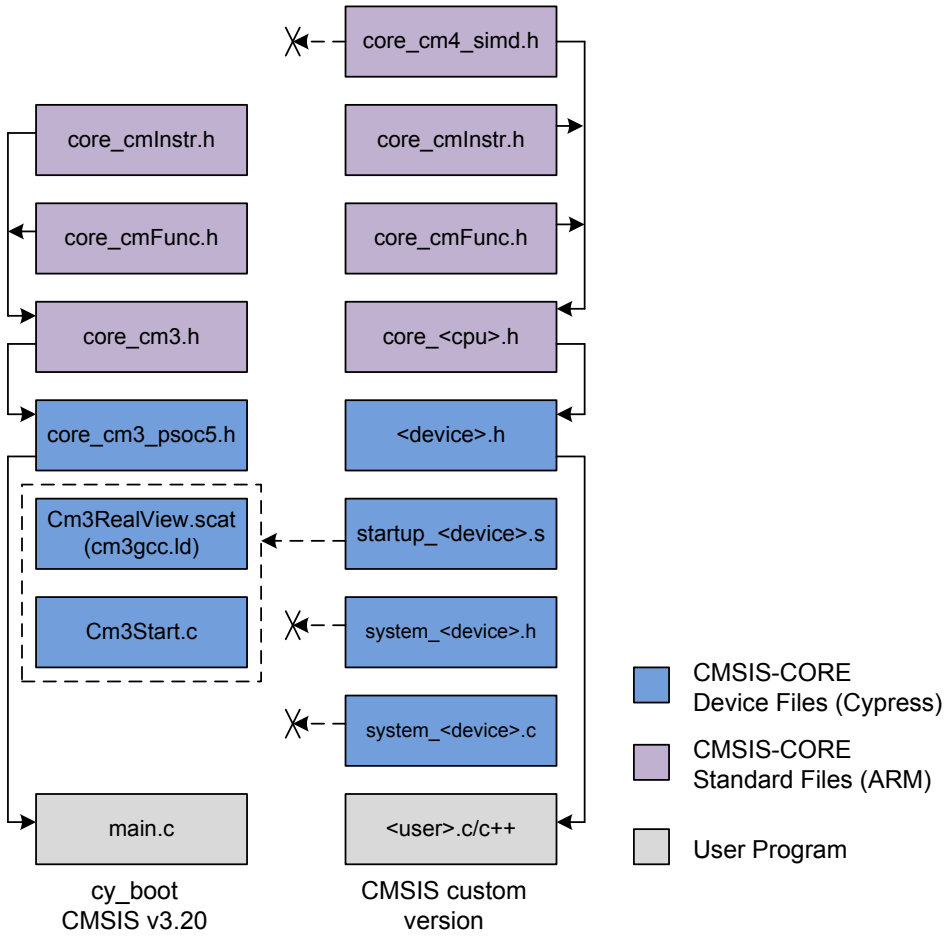
- Support for RTOS and user replacement of routines. This is possible because the library routines are denoted as "weak" allowing their replacement if another implementation is provided.
- A mechanism is provided that allows for the replacement of the provided linker/scatter file with a user version. This is implemented by allowing the user to create the file local to their project and having a build setting that allows the specification of this file as the linker/scatter file instead of the file provided automatically.
- Currently the heap and stack size are specified as a fixed quantity (4 K Stack, 1 K Heap). If possible the requirement to specify Heap and Stack sizes should be removed entirely. If that is not possible, then these values should be the defaults with the option to choose other values in the Design-Wide Resources GUI.
- All the code in the Generated Source tree is compiled into a single library as part of the build process. Then that compiled library is linked in with the user code in the final link.

## CMSIS Support

Cortex Microcontroller Software Interface Standard (CMSIS) is a standard from ARM for interacting with Cortex M-series processors. There are multiple levels of support. The Core Peripheral Access Layer (CMSIS Core) support is provided. For the more information refer to CMSIS - Cortex Microcontroller Software Interface Standard on www.arm.com.

PSoC Creator 3.0 provides support for CMSIS Core version 3.20. Also, PSoC Creator 3.0 provides the ability to use a custom version of the CMSIS Core.

The following diagram shows how CMSIS Core version 3.20 files are integrated into the cy_boot component and how custom version of CMSIS Core files can be integrated.




The following describe each file from the diagram:

- The *Cm3Start.c* and *cm3gcc.ld* files (part of the cy_boot component) contain Cortex-M3 device startup code and interrupt vector tables and completely substitute CMSIS startup_<device>.s template file.
- Vendor-specific device file <device>.h that includes CMSIS Core standard files is represented in cy_boot component by *core_cm3_psoc5.h*.
- The *core_cmInstr.h* file defines intrinsic functions to access special Cortex-M instructions and *core_cmFunc.h* file provides functions to access the Cortex-M core peripherals. These files were added since CMSIS Core version 2.0.
- The *core_cm4_simd.h* file added to the CMSIS SIMD Instruction Access is relevant for Cortex-M4 only.
- *system_<device>.h, system_<device>.c* – Generic files for system configuration (i.e. processor clock and memory bus system), are partially covered by *Cm3Start.c*.

### *Manual addition of the CMSIS Core files*

Beginning with PSoC Creator 2.2, the “Include CMSIS Core Peripheral Library Files” option is added to the System tab of the DWR file. By default, this option is enabled and CMSIS Core version 3.20 files are added to the project. This option should be disabled if you wish to manually add CMSIS Core files.

Un-check “Include CMSIS Core Peripheral Library Files” option on the System tab of the DWR file to detach CMSIS 3.20 files from the cy_boot component.

Add the following CMSIS Core files to the project:

- core_cmInstr.h
- core_cmFunc.h
- core_cm3.h

Based on the CMSIS vendor-specific template file (<device>.h), create device header file, copy device specific definitions from core_cm3_psoc5.h file and add following definitions at the top of the file:

```
#include <cytypes.h>

#define __CHECK_DEVICE_DEFINES

#define __CM3_REV              0x0201

#define __MPU_PRESENT              0
#define __NVIC_PRIO_BITS           3
#define __Vendor_SysTickConfig     0
```

Include the previously created vendor-specific device header file to the application.

## Preservation of Reset Status

### PSoC 3/PSoC 5LP

The value of the reset status register (RESET_SR0) is read and cleared any time the device is booted. That value is saved to a global SRAM variable. Note that the IPOR, PRES, and LVI reset sources clear the RESET_SR0 register, and the WDT, Software reset and XRES reset sources preserve the RESET_SR0 register. For more information, refer to the device datasheet and TRM.

Some PSoC 3 devices perform an additional software reset. If any other than a software device reset previously occurred, it will reload the NVLs and apply the correct settings. This operation is transparent to the normal boot process and will not interfere with bootloading, debugging, or normal device functionality. See the device errata for more information.

To retain user-defined status that persists through many resets, use the CY_RESET_GP0 and CY_RESET_GP1 bits in the RESET_SR0 register. The CyResetStatus variable is used to obtain value of these bits after a device reset. These bits are used by Bootloader and Bootloadable projects and cannot be used by user.

#### *uint8 CyResetStatus*

| Name | Description |
|---|---|
| CY_RESET_LVID | Low voltage detect digital |
| CY_RESET_LVIA | Low voltage detect analog |

| Name | Description |
|---|---|
| CY_RESET_HVIA | High voltage detect analog |
| CY_RESET_WD | Watchdog reset |
| CY_RESET_SW | Software reset |
| CY_RESET_GP0 | General purpose bit 0 |
| CY_RESET_GP1 | General purpose bit 1 |

## PSoC 4

### *uint32 CySysGetResetReason(uint32 reason)*

**Description:** The function returns the cause for the latest reset(s) that occurred in the system and clears those that are defined with the parameter.
All bits in the RES_CAUSE register assert when the corresponding reset cause occurs and must be cleared by firmware. These bits are cleared by hardware only during XRES, POR, or a detected brown-out.

**Parameters:** reason: bits in the RES_CAUSE register to clear.

| Define | Source |
|---|---|
| CY_SYS_RESET_WDT | WDT |
| CY_SYS_RESET_PROTFAULT | Protection Fault |
| CY_SYS_RESET_SW | Software reset |

**Return Value:** Status. Same enumerated bit values as used for the reason parameter.

**Side Effects and Restrictions:** None

# 13 Watchdog Timer

## PSoC 3/PSoC 5LP

### void CyWdtStart(uint8 ticks, uint8 lpMode)

**Description:** Enables the watchdog timer. The timer is configured for the specified count interval, the CTW is cleared, the setting for low power mode is configured, and the watchdog timer is enabled.

The hardware implementation of the watchdog timer prevents any modification of the timer once it has been enabled. It also prevents the timer from being disabled once it has been enabled. This protects the watchdog timer from changes caused by errant code. As a result, only the first call to CyWdtStart() after reset will have any effect.

The watchdog counts each time the CTW reaches the period specified. The watchdog must be cleared using the CyWdtClear() function before the watchdog counts to three. The CTW is free running, so this will occur after between 2 and 3 timer periods elapse.

**Parameters:** ticks: One of the four available timer periods. Once WDT enabled, the interval cannot be changed.

| Define | Time |
|---|---|
| CYWDT_2_TICKS | 4 – 6 ms |
| CYWDT_16_TICKS | 32 – 48 ms |
| CYWDT_128_TICKS | 256 – 384 ms |
| CYWDT_1024_TICKS | 2.048 – 3.072 s |

### void CyWdtStart(uint8 ticks, uint8 lpMode) (continued)

**Parameters** lpMode: Low power mode configuration.

| Define | Effect |
|---|---|
| CYWDT_LPMODE_NOCHANGE | No Change |
| CYWDT_LPMODE_MAXINTER | Switch to longest timer mode during sleep / hibernate |
| CYWDT_LPMODE_DISABLED | Disable WDT during sleep / hibernate |

**Return Value:** None

### void CyWdtClear()

**Description:** Clears (feeds) the watchdog timer.

**Parameters:** None

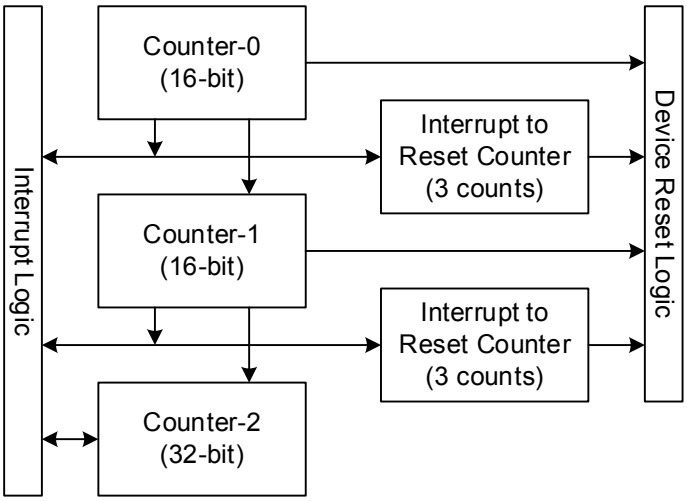
**Return Value:** None

# PSoC 4100 and PSoC 4200

## PSoC 4100 and PSoC 4200 Functional Description

The WDT asserts an interrupt or a hardware reset to the device after a preprogrammed interval, unless it is periodically serviced in firmware. The WDT has two 16- bit counters (Counter-0 and Counter-1) and one 32-bit counter (Counter-2).

These counters can be configured to work independently or in cascade. The cascade configuration provides an option to increase the reset or interrupt interval.




The Counter-0 and Counter-1 generate an interrupt or a reset on reaching the specified terminal count for the first time, or generate a reset after three continuous unhandled interrupt, whereas Counter-2 only generates an interrupt. The interrupt must be cleared after entering the Interrupt Service Routine (ISR) in firmware by calling CySysWdtClearInterrupt() with the corresponding parameter.

The Counter-0 and Counter-1 perform action when the corresponding counter value equals corresponding match value configured by call to CySysWdtWriteMode(). The Counter-2 perform action when the bit defined by call to CySysWdtWriteToggleBit() is toggled in Counter-2. For example, if toggle bit is bit number 7 (configured by call to the CySysWdtWriteToggleBit(7) function), the Counter-2 will generate one interrupt per 2^7=128 WDT clocks.

### *Power Modes*

In the Active mode, the interrupt request from the WDT is sent to the CPU via IRQ 9. In the Sleep or Deep Sleep power mode the CPU subsystem is powered-down, so the interrupt request from the WDT is directly sent to the WakeUp Interrupt Controller (WIC), which will then wake up the CPU. Then, the CPU acknowledges the interrupt request and executes the Interrupt Service Routine (ISR).

Enabling or disabling a WDT requires three LFCLK cycles to come into effect. During that period the SYSCLK clock should be available. That means the device should be not put into Deep Sleep mode during that period.

After waking from Deep Sleep, an internal timer value is set to zero until the ILO loads the register with the correct value. This led to an increase in low-power mode current consumption. The work around is to wait for the first positive edge of the ILO clock before allowing the WDT_CTR_* registers to be read by CySysWdtReadCount() function.

### Clock Source

The WDT is clocked by LFCLK sourced by the 32 kHz ILO. By default, this oscillator is enabled by PSoC Creator on device startup. Therefore, the ILO is enabled by WDT API that accesses WDT registers.

According to the device datasheet, the ILO accuracy is +/-60% over voltage and temperature. This means that the timeout period may vary by 60% from the configured. Appropriate margins should be added while configuring WDT intervals to make sure that unwanted device resets does not occur on some devices.

Please refer to the device datasheet for the more information on oscillator accuracy.

### Register Locking

Accidental corruption of the WDT configuration can be prevented by setting the bit-field WDT_LOCK of the CLK_SELECT register by calling the CySysWdtLock() function. When WDT is locked, any writing to the WDT_* and CLK_ILO* registers is ignored.

The CySysWdtUnlock() function should be called in order to allow WDT registers modification.

### Clearing WDT

The LFCLK clock is asynchronous to the SYSCLK. So, generally, it takes 3 LFCLK cycles for the WDT registers changes to come into effect. This is important to remember that WDT should be cleared at least 4 cycles (3 + 1 for sure) before timeout occurs, especially when small match values / low toggle bit numbers are used.

The WDT counters should be cleared by calling the CySysWdtResetCounters() function with the parameter corresponding to the counters whose values are going to be cleared.

It is recommended to clear WDT counters from the portion of the code that is not directly associated with the WDT interrupt. It is possible that the main function of the firmware has crashed or is in an endless loop, but that the WDT interrupt vector is still intact and the WDT is getting serviced properly.

### Reset Detection

The CySysResetReason() function can be used to detect if the watchdog has triggered device reset.

### Interrupt Configuration

The Global Signal Reference and Interrupt components can be used for the ISR configuration. If the WDT is configured to generate interrupt, the pending interrupts must be cleared within ISR (otherwise, the interrupt will be generated continuously):

- Pending interrupt to the interrupt controller must be cleared by the call to the WDTISR_ClearPending() function, where WDTISR is the instance name of the interrupt component.
- Pending interrupt to the WDT block must be cleared by the call to the CySysWdtClearInterrupt() function. The call to the function will clear the unhandled WDT interrupt counter, if WDT is configured to be in “generate interrupts and reset on 3rd unhandled interrupt” mode.

It is recommended to use WDT ISR as timer to trigger certain actions and to change the next WDT match value.

## PSoC 4100 and PSoC 4200 APIs

### *void CySysWdtLock(void)*

**Description:** Locks out configuration changes to the Watchdog timer registers and ILO configuration register.

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions:** This API enables the ILO, if it was disabled. After this API is called, the ILO can't be disabled until calling of CySysWdtUnlock().

### *void CySysWdtUnlock(void)*

**Description:** Unlocks the Watchdog Timer configuration register.

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions:** This API enables the ILO, if it was disabled.

### *void CySysWdtWriteMode(uint32 counterNum, uint32 mode)*

**Description:** Writes the mode of one of the three WDT counters. If WDT mode is not configured, WDT timers are in a free running mode.

**Parameters:** counterNum: Valid range [0-2]. Number of the WDT counter.
mode: Mode of operation for the counter:

| Define | Mode |
|---|---|
| CY_SYS_WDT_MODE_NONE | Free running |
| CY_SYS_WDT_MODE_INT | Interrupt generated on match for counter 0 and 1, and on bit toggle for counter 2 |
| CY_SYS_WDT_MODE_RESET | Reset on match (valid for counter 0 and 1 only) |
| CY_SYS_WDT_MODE_INT_RESET | Generate an interrupt and generate a reset on the 3rd unhandled interrupt. (valid for counter 0 and 1 only) |

**Return Value:** None

**Side Effects and Restrictions:** WDT counter counterNum should be disabled to set mode. Otherwise this function call will have no effect. If the specified counter is enabled, call CySysWdtDsiable() function with the corresponding parameter to disable specified counter and wait for it to stop. This may take up to three LFCLK cycles.
This API enables the ILO, if it was disabled.

### *uint32 CySysWdtReadMode(uint32 counterNum)*

**Description:** Reads the mode of one of the three WDT counters.

**Parameters:** counterNum: Valid range [0-2]. Number of the WDT counter.

**Return Value:** Mode of the counter. Same enumerated values as mode parameter used in CySysWdtWriteMode().

### *void CySysWdtWriteClearOnMatch(uint32 counterNum, uint32 enable)*

**Description:** Configures the WDT counter clear on a match setting. If configured to clear on match the counter will count from 0 to the MatchValue giving it a period of (MatchValue + 1).

**Parameters:** counterNum: Valid range [0-1]. Number of the WDT counter. Match values are not supported by counter 2.

enable: 0 to disable, 1 to enable

**Return Value:** None

**Side Effects and Restrictions:** WDT counter counterNum should be disabled. Otherwise this function call will have no effect. If the specified counter is enabled, call CySysWdtDsiable() function with the corresponding parameter to disable specified counter and wait for it to stop. This may take up to three LFCLK cycles.
This API enables the ILO, if it was disabled.

### *uint32 CySysWdtReadClearOnMatch(uint32 counterNum)*

**Description:** Reads the clear on match setting for the specified counter.

**Parameters:** counterNum: Valid range [0-1]. Number of the WDT counter. Match values are not supported by counter 2.

**Return Value:** Clear on Match status: 1 if enabled, 0 if disabled.

### *void CySysWdtEnable(uint32 counterMask)*

**Description:** Enables the specified WDT counters. All counters specified in the mask are enabled.

**Parameters:** counterMask: Mask of all counters to enable:

| Define | Counter |
|---|---|
| CY_SYS_WDT_COUNTER0_MASK | 0 |
| CY_SYS_WDT_COUNTER1_MASK | 1 |
| CY_SYS_WDT_COUNTER2_MASK | 2 |

**Return Value:** None

**Side Effects and Restrictions:** Enabling or disabling a WDT requires three LF clock cycles to come into effect. Therefore, the WDT enable state must not be changed more than once in that period.

After WDT is enabled, it is illegal to write WDT configuration (WDT_CONFIG) and control (WDT_CONTROL) registers. That means that all WDT functions that contain ‘write’ in the name (with the exception of CySysWdtWriteMatch() function) are illegal to call once WDT enabled.

This API enables the ILO, if it was disabled.

### *void CySysWdtDisable(uint32 counterMask)*

**Description:** Disables the specified WDT counters. All counters specified in the mask are disabled.

**Parameters:** counterMask: Mask of all counters to disable:

| Define | Counter |
|---|---|
| CY_SYS_WDT_COUNTER0_MASK | 0 |
| CY_SYS_WDT_COUNTER1_MASK | 1 |
| CY_SYS_WDT_COUNTER2_MASK | 2 |

**Return Value:** None

**Side Effects and Restrictions:** Enabling or disabling a WDT requires three LF clock cycles to come into effect. Therefore, the WDT enable state must not be changed more than once in that period.

This API enables the ILO, if it was disabled.

### *uint32 CySysWdtReadEnabledStatus (uint32 counterNum)*

**Description:** Reads the enabled status of one of the three WDT counters.

**Parameters:** counterNum: Valid range [0-2]. Number of the WDT counter.

**Return Value:** Status of WDT counter:
0 - counter is disabled, 1 - counter is enabled

**Side Effects and Restrictions:** This API returns actual WDT counter status from status register. It may take up to Three LFCLK cycles since WDT counter was enabled for WDT status register to contain actual data.

### *void CySysWdtWriteCascade(uint32 cascadeMask)*

**Description:** Writes the two WDT cascade values based on the combination of mask values specified.

**Parameters:** cascadeMask: Mask value used to set or clear both of the cascade values:

| Define | Counter |
|---|---|
| CY_SYS_WDT_CASCADE_NONE | Neither |
| CY_SYS_WDT_CASCADE_01 | Cascade 01 |
| CY_SYS_WDT_CASCADE_12 | Cascade 12 |

To set both cascade modes, two defines should be ORed:
(CY_SYS_WDT_CASCADE_01 | CY_SYS_WDT_CASCADE_12)

**Return Value:** None

**Side Effects and Restrictions:** If only one cascade mask is specified, the second cascade is disabled. To set both cascade modes, two defines should be ORed:
(CY_SYS_WDT_CASCADE_01 | CY_SYS_WDT_CASCADE_12)

WDT counters that are part of the specified cascade, should be disabled. Otherwise this function call will have no effect. If the specified counter is enabled, call CySysWdtDsiable() function with the corresponding parameter to disable specified counter and wait for it to stop. This may take up to three LFCLK cycles. This API enables the ILO, if it was disabled.

### *uint32 CySysWdtReadCascade(void)*

**Description:** Reads the two WDT cascade values returning a mask of the bits set.

**Parameters:** None

**Return Value:** Mask of cascade values set:

| Define | Cascade |
|---|---|
| CY_SYS_WDT_CASCADE_NONE | Neither |
| CY_SYS_WDT_CASCADE_01 | Cascade 01 |
| CY_SYS_WDT_CASCADE_12 | Cascade 12 |

### *void CySysWdtWriteMatch(uint32 counterNum, uint32 match)*

**Description:** Configures the WDT counter match comparison value.

**Parameters:** counterNum: Valid range [0-1]. Number of the WDT counter. Match values are not supported by counter 2.

match: Valid range [0-65535]. Value to be used to match against the counter.

**Return Value:** None

**Side Effects and Restrictions:** This API enables the ILO, if it was disabled.

### *void CySysWdtWriteToggleBit(uint32 bits)*

**Description:** Configures which bit in the WDT Counter 2 to monitor for a toggle. When that bit toggles, an interrupt is generated if the mode for Counter 2 has interrupts enabled.

**Parameters:** bit: Valid range [0-31]. Counter 2 bit to monitor for a toggle.

**Return Value:** None

**Side Effects and Restrictions:** WDT Counter 2 should be disabled. Otherwise this function call will have no effect. If the specified counter is enabled, call CySysWdtDsiable() function with the corresponding parameter to disable specified counter and wait for it to stop. This may take up to three LFCLK cycles.

This API enables the ILO, if it was disabled.

### *uint32 CySysWdtReadToggleBit(void)*

**Description:** Reads which bit in the WDT counter 2 is monitored for a toggle.

**Parameters:** None

**Return Value:** Bit that is monitored (range of 0 to 31).

### *uint32 CySysWdtReadMatch(uint32 counterNum)*

**Description:** Reads the WDT counter match comparison value.

**Parameters:** counterNum: Valid range [0-1]. Number of the WDT counter. Match values are not supported by counter 2. Changes may take up to three LFCLK to be applied.

**Return Value:** 16-bit match value.

### *uint32 CySysWdtReadCount(uint32 counterNum)*

**Description:** Reads the current WDT counter value.

**Parameters:** counterNum: Valid range [0-2]. Number of the WDT counter.

**Return Value:** Live counter value. Counter 0 and 1 are 16 bit counters and counter 2 is a 32 bit counter.

### *uint32 CySysWdtGetInterruptSource(void)*

**Description:** Reads a mask containing all the WDT interrupts that are currently set.

**Parameters:** None

**Return Value:** Mask of interrupts set:

| Define | Counter |
|---|---|
| CY_SYS_WDT_COUNTER0_INT | 0 |
| CY_SYS_WDT_COUNTER1_INT | 1 |
| CY_SYS_WDT_COUNTER2_INT | 2 |

### *void CySysWdtClearInterrupt(uint32 counterMask)*

**Description:** Clears all WDT counter interrupts set in the mask. Calling this API also prevents Reset from happening when counter mode is set to generate 3 interrupts and then reset device. All WDT interrupts are to be cleared by firmware, otherwise interrupts will be generated continuously.

**Parameters:** counterMask: Mask of all counters to enable:

| Define | Counter |
|---|---|
| CY_SYS_WDT_COUNTER0_INT | 0 |
| CY_SYS_WDT_COUNTER1_INT | 1 |
| CY_SYS_WDT_COUNTER2_INT | 2 |

**Return Value:** None

**Side Effects and Restrictions:** This API enables the ILO, if it was disabled; temporarily removes the Watchdog lock, if it was set; and restores the lock state, after cleaning WDT interrupts that were set in mask.

### *void CySysWdtResetCounters(uint32 counterMask)*

**Description:** Resets all WDT counters set in the mask.

**Parameters:** counterMask: Mask of all counters to reset:

| Define | Counter |
|---|---|
| CY_SYS_WDT_COUNTER0_RESET | 0 |
| CY_SYS_WDT_COUNTER1_RESET | 1 |
| CY_SYS_WDT_COUNTER2_RESET | 2 |

**Return Value:** None

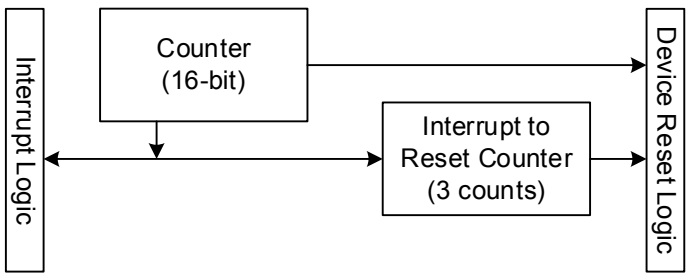
**Side Effects and Restrictions:** This API enables the ILO, if it was disabled. This API call will not reset counter values if the Watchdog is locked.

## PSoC 4000

**Note** It is highly recommended to enable WDT if the power supply can produce sudden brownout events that may compromise the CPU functionality. This ensures that, after a brown-out compromises the CPU functionality, the system will always recover.

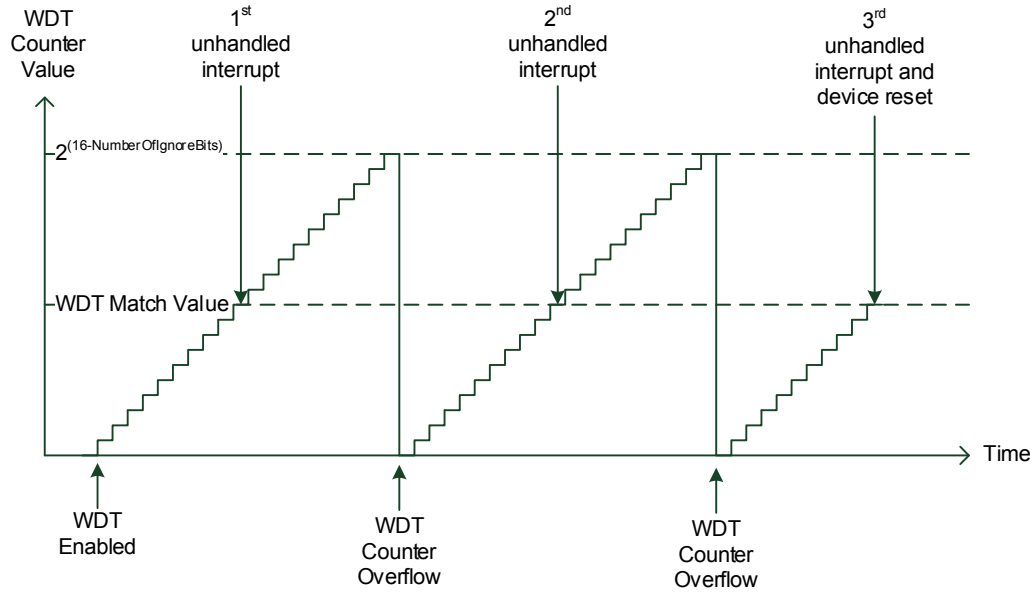
### PSoC 4000 Functional Description

The WDT asserts an interrupt or a hardware reset to the device after a preprogrammed interval, unless it is periodically serviced in firmware. The WDT is a 16- bit free-running up-counter.



The WDT generates an interrupt when the count value in the counter equals the configured match value.

It is important to take into considerations that the counter is not reset on match. When the counter reaches match value, it generates an interrupt and then keeps counting up till it overflows and rolls back to zero and reaches the match value again at which point another interrupt is generated.




In case of the intention to use WDT for a periodic interrupt generation, the match value should be incremented in the ISR. As a result, the next WDT interrupt will be generated when the counter reaches the new match value.

Also some functionality is added to reduce entire WDT counter period, by specifying the number of most significant bits that are cut-off in the WDT counter. For example, if the CySysWdtWriteIgnoreBits() function called with parameter 3, the WDT counter becomes a 13-bit free-running up-counter.

The WDT reset period can be calculated using following equation:

$$WDT_{\mathrm{Re}setTime} = 2 * (\mathrm{LFCLK}_{Period} * (2^{(16 - NumberOfIgnoreBits)})) + (LFCLK_{Period} * WDT_{MatchValue})$$

### *Power Modes*

In the Active mode, the interrupt request from the WDT is sent to the CPU via IRQ 4. In the Sleep or Deep Sleep power mode the CPU subsystem is powered-down, so the interrupt request from the WDT is directly sent to the WakeUp Interrupt Controller (WIC), which will then wake up the CPU. Then, the CPU acknowledges the interrupt request and executes the Interrupt Service Routine (ISR).

Enabling or disabling a WDT requires three LFCLK cycles to come into effect. During that period the SYSCLK clock should be available. That means that the device should not be put into Deep Sleep mode during that period.

After waking from Deep Sleep, an internal timer value is set to zero until the ILO loads the register with the correct value. This led to an increase in low-power mode current consumption. The work around is to wait for the first positive edge of the ILO clock before allowing the WDT_CTR_* registers to be read by CySysWdtReadCount() function.

### Clock Source

The WDT is clocked by the LFCLK sourced by the 32 kHz ILO. The WDT reset must be disabled before disabling the ILO. Otherwise, any register write to disable the ILO will be ignored. Enabling the WDT reset will automatically enable the ILO.

According to the device datasheet, the ILO accuracy is +/-60% over voltage and temperature. This means that the timeout period may vary by 60% from the configured. An appropriate margin should be added while configuring WDT intervals to make sure that unwanted device resets do not occur on some devices.

Please refer to the device datasheet for the more information on oscillator accuracy.

### Register Locking

This feature is not available for the device.

### Clearing WDT

The LFCLK clock is asynchronous to the SYSCLK. So, generally, it takes three LFCLK cycles for the WDT registers changes to come into effect. This is important to remember that WDT should be cleared at least four cycles (3 LFCLK cycles + 1 to be sure) before timeout occurs, especially when small match values / low toggle bit number are used.

It is recommended to clear WDT counters from the portion of the code that is not directly associated with the WDT interrupt. It is possible that the main function of the firmware has crashed or is in an endless loop, but that the WDT interrupt vector is still intact and the WDT is getting serviced properly.

### Reset Detection

The CySysResetReason() function can be used to detect if the watchdog has triggered device reset.

### Interrupt Configuration

The Global Signal Reference and Interrupt components can be used for the ISR configuration. If the WDT is configured to generate interrupt, the pending interrupts must be cleared within ISR (otherwise, the interrupt will be generated continuously):

- Pending interrupt to the interrupt controller must be cleared by the call to the WDTISR_ClearPending() function, where WDTISR is the instance name of the interrupt component.
- Pending interrupt to the WDT block must be cleared by the call to the CySysWdtClearInterrupt() function. The call to the function will clear the unhandled WDT interrupt counter, if WDT is configured to be in “generate interrupts and reset on 3$^{rd}$ unhandled interrupt” mode.

It is recommended to use WDT ISR as timer to trigger certain actions and to change the next WDT match value.

## PSoC 4000 APIs

### *uint32 CySysWdtReadEnabledStatus(void)*

**Description:** Reads the enabled status WDT counter.

**Parameters:** None

**Return Value:** Status of WDT counter:
0 - counter is disabled
1 - counter is enabled

**Side Effects and Restrictions:** None

### *void CySysWdtEnable(void)*

**Description:** Enables the WDT counter.

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions:** This API enables ILO, if it was disabled.

### *void CySysWdtWriteMatch(uint32 match)*

**Description:** Configures the WDT counter match comparison value.

**Parameters:** match:
Valid range [0-65535]. Value to be used to match against the counter.

**Return Value:** None

**Side Effects and Restrictions:** This API enables ILO, if it was disabled.

### *uint32 CySysWdtReadMatch(void)*

**Description:** Reads the WDT counter match comparison value.

**Parameters:** None

**Return Value:** Counter match value.

### *uint32 CySysWdtReadCount(void)*

**Description:** Reads the current WDT counter value.

**Parameters:** None

**Return Value:** Live counter value.

**Side Effects and Restrictions:** None

### *void CySysWdtWriteIgnoreBits(uint32 bitsNum)*

**Description:** Configures the number of MSB bits of the watchdog timer that are not checked against match.

**Parameters:** bitsNum: Valid range [0-15]. Number of MSB bits.

**Return Value:** None

**Side Effects and Restrictions:** The value of bitsNum provides control over the time-to-reset of the watchdog (which happens after 3 successive matches).

### *uint32 CySysWdtReadIgnoreBits(void)*

**Description:** Reads the number of MSB bits of the watchdog timer that are not checked against match.

**Parameters:** None

**Return Value:** Number of MSB bits.

**Side Effects and Restrictions:** None

### *void CySysWdtClearInterrupt(void)*

**Description:** Cleans WDT match flag, which is set every time WDT counter reaches WDT match value.

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions:** Clearing this bit also feeds the watch dog. Missing 2 interrupts in a row will generate brown-out reset.

### *void CySysWdtMaskInterrupt(void)*

**Description:** After masking interrupts from WDT, they are not passed to CPU.

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions:** This API does not disable the WDT reset generation on 2 missed interrupts.

### *void CySysWdtUnmaskInterrupt(void)*

**Description:** Unmasks WDT interrupts.

**Parameters:** None

**Return Value:** None

**Side Effects and Restrictions:** None

This page intentionally left blank.

# 14 MISRA Compliance

This chapter describes the MISRA-C:2004 compliance and deviations for the PSoC Creator cy_boot component and code generated by PSoC Creator.

MISRA stands for Motor Industry Software Reliability Association. The MISRA specification covers a set of 122 mandatory rules and 20 advisory rules that apply to firmware design and has been put together by the Automotive Industry to enhance the quality and robustness of the firmware code embedded in automotive devices.

There are two types of deviations defined:

- project deviations – deviations that are applicable for all PSoC Creator components
- specific deviations – deviations that are applicable for the specific component

This section provides information on the following items:


- Verification Environment
- Project Deviations
- Documentation Related Rules
- PSoC Creator Generated Sources Deviations
- cy_boot Component-Specific Deviations


## Verification Environment

This section provides MISRA compliance analysis environment description.

| Component | Name | Version |
|---|---|---|
| Test Specification | MISRA-C:2004 Guidelines for the use of the C language in critical systems. | October 2004 |
| Target Device | PSoC 3 | Production |
| | PSoC 4 | Production |
| | PSoC 5LP | Production |
| Target Compiler | PK51 | 9.51 |
| | GCC | 4.7.3 |
| | RVDS | 4.1 |
| | MDK | 4.1 |
| Generation Tool | PSoC Creator | 3.0 SP1 |

| Component | Name | Version |
|---|---|---|
| MISRA Checking Tool | Programming Research QA C source code analyzer for Windows | 8.1-R |
| | Programming Research QA C MISRA-C:2004 Compliance Module (M2CM) | 3.2 |

The MISRA rules 1.5, 2.4, 3.3, and 5.7 are not enforced by Programming Research QA C. The compliance with these rules was verified manually by code review.

## Project Deviations

A Project Deviations are defined as a permitted relaxation of the MISRA rules requirements that are applied for source code that is shipped with PSoC Creator. The list of deviated rules is provided in the table below.

| MISRA-C: 2004 Rule | Rule Class (R/A)[1] | Rule Description | Description of Deviation(s) |
|---|---|---|---|
| 1.1 | R | This rule states that code shall conform to C ISO/IEC 9899:1990 standard. | Some C language extensions (like interrupt keyword) relate to device hardware functionality and cannot be practically avoided.<br>In the main.c file that is generate by PSoC Creator the non-standard main() declaration is used: “void main()”. The standard declaration is “int main()”<br>The number of macro definitions exceeds 1024 - program does not conform strictly to ISO:C90. |
| 5.1 | R | This rule says that both internal and external identifiers shall not rely on the significance of more than 31 characters. | The length of names based on user-defined names depends on the length of the user-define names. |
| 5.7 | A | Verify that no identifier name should is reused. | Local variables with the same name may appear in different functions. Aside from commonly used names such as 'i', generated API functions for multiple instances of the same component will have identical local variable names. |
| 8.7 | R | Objects shall be defined at block scope if they are only accessed from within a single function. | The object 'InstanceName_initVar' is only referenced by function 'InstanceName_Start', in the translation unit where it is defined. The intention of this publicly available global variable is to be used by user application. |
| 8.10 | R | All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required. | Components API are designed to be used in user application and might not be used in component API. |
| 11.3 | A | This rule states that cast should not be performed between a pointer type and an integral type. | The cast from unsigned int to pointer does not have any unintended effect, as it is a consequence of the definition of a structure based on hardware registers. |

[1] Required / Advisory

| MISRA-C: 2004 Rule | Rule Class (R/A)[1] | Rule Description | Description of Deviation(s) |
|---|---|---|---|
| 14.1 | R | There shall be no unreachable code. | Some functions that are part of the component API are not used within component API. Components API are designed to be used in user application and might not be used in component API. |
| 21.1 | R | Minimization of run-time failures shall be ensured by the use of at least one of:<br>a) static analysis tools/techniques;<br>b) dynamic analysis tools/techniques;<br>c) explicit coding of checks to handle run-time faults. | Some components in some specific configurations can contain redundant operations introduced because of generalized implementation approach. |

## Documentation Related Rules

This section provides information on implementation-defined behavior of the toolchains supported by PSoC Creator. The list of deviated rules is provided in the table below.

| MISRA-C: 2004 Rule | Rule Class (R/A)[1] | Rule Description | Description |
|---|---|---|---|
| 1.3 | R | Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform. | No multiple compilers and languages can be used at a time for PSoC Creator projects.<br>The PK51 linker produces OMF-51 object module format. The GCC linker produces EABI format files. The RVDS and MDK linkers produce files of ARM ELF format. |
| 1.4 | R | The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers. | PK51 and GCC treat more than 31 characters of internal and external identifier length, and are case sensitive (e.g., Id and ID are not equal). |
| 1.5 | A | Rule states that floating-point implementation should comply with a defined floating-point standard. | Floating-point arithmetic implementation conforms to IEEE-754 standard. |
| 3.1 | R | All usage of implementation-defined behavior shall be documented. | For the documentation on PK51 and GCC compilers, refer to the Help menu, Documentation sub-menu, Keil and GCC commands respectively. |
| 3.2 | R | The character set and the corresponding encoding shall be documented. | The Windows-1252 (CP-1252) character set encoding is used.<br>Some characters that are used for source code generation in PSoC Creator are not included in character set, defined by ISO-IEC 9899-1900 "Programming languages — C". |
| 3.3 | A | This rule states that implementation of integer division should be documented. | When dividing two signed integers, one of which is positive and one negative compiler rounds up with a negative remainder. |
| 3.5 | R | This rules requires implementation defined behavior and packing of bit fields be documented. | The use of bit-fields is avoided. |
| 3.6 | R | All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation. | The C standard libraries provided with C51, GCC, and RVCT have not been reviewed for compliance. Some code uses memset and memcpy. The compiler may also insert calls to its vendor-specific compiler support library. |

## PSoC Creator Generated Sources Deviations

This section provides the list of deviations that are applicable for the code that is generated by PSoC Creator. The list of deviated rules is provided in the table below.

| MISRA-C: 2004 Rule | Rule Class (R/A)[1] | Rule Description | Description of Deviation(s) |
|---|---|---|---|
| 3.4 | R | All uses of the *#pragma* directive shall be documented. | The #pragma directive is required to ensure that the C51 compiler produces efficient code for generated functions related to the AMuxSeq component. |
| 11.4 | A | This rule states that cast should not be performed between a pointer to object type and a different pointer to object type. | CYMEMZERO8 and CYCONFIGCPY8 use void * arguments for compatibility with memset/memcpy but must use a pointer to an actual type internally. |
| 14.1 | R | Rule requires that there shall be no unreachable code. | The CYMEMZERO, CYMEMZERO8, CYCONFIGCPY, CYCONFIGCPY8, CYCONFIGCPYCODE, and CYCONFIGCPYCODE8 are often but not always used. |
| 15.2 | R | Switch cases must end with *break* statements. | The code structure is required to ensure that the C51 compiler produces efficient code for generated functions related to the AMuxSeq component. |
| 15.3 | R | *default* must be the last clause in a *switch* statement. | The code structure is required to ensure that the C51 compiler produces efficient code for generated functions related to the AMuxSeq component. |
| 17.4 | R | Array indexing shall be only allowed form of pointer arithmetic. | The CYMEMZERO8 and CYCONFIGCPY8 have void * arguments for compatibility with memset/memcpy. |
| 19.7 | A | The rule says that function shall be used instead of function-like macro. | The CYMEMZERO, CYMEMZERO8, CYCONFIGCPY, CYCONFIGCPY8, CYCONFIGCPYCODE, and CYCONFIGCPYCODE8 macros are used to call cymemzero, cyconfigcpy, and cyconfigcpycode in a device-independent way. The macros cannot be converted to functions without significantly increasing the time and memory required for each function call (this is a limitation of C51). The macros have been converted to functions for GCC/RVCT. |

## cy_boot Component-Specific Deviations

This section provides the list of cy_boot component specific-deviations. The list of deviated rules is provided in the table below.

| MISRA-C: 2004 Rule | Rule Class (R/A)[1] | Rule Description | Description of Deviation(s) |
|---|---|---|---|
| 6.3 | A | typedefs that indicate size and signedness should be used in place of the basic types. | For PSoC 4/PSoC 5LP, the RealView C Library initialization function __main(void) in startup file (Cm0Start.c/Cm3Start.c) file returns value of basic type 'int'. |
| 8.7 | R | Objects shall be defined at block scope if they are only accessed from within a single function. | For PSoC 4/PSoC 5LP, the cySysNoInitDataValid variable is intentionally declared as global in Cm0Start.c/Cm3Start.c files to prevent linker from CY_NOINIT section removal. |
| 8.12 | R | When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization. | For PSoC 4/PSoC 5LP (Cm0Start.c/Cm3Start.c), the __cy_regions array of structures is declared with unknown size. |
| 12.10 | R | The comma operator shall not be used. | The Cm0Start.c/Cm3Start.c files, contain the complex for() statement that performs sections initialization and has the mentioned deviations. |
| 12.13 | A | The increment (++) and decrement (--) operators should not be mixed with other operators in an expression. | |
| 13.2 | A | Tests of a value against zero should be made explicit, unless the operand is effectively Boolean. | |
| 13.5 | R | The three expressions of a for statement shall be concerned only with loop control. | |
| 8.8 | R | An external object or function shall be declared in one and only one file. | For the PSoC 4/PSoC 5LP, some objects is being declared with external linkage in Cm3Start.c/Cm3Start.c file and this declaration is not in a header file. |
| 10.1 | R | The value of an expression of integer type shall not be implicitly converted to a different underlying type under some circumstances. | PSoC 4/ PSoC 5LP: CMSIS Core: An integer constant of 'essentially unsigned' type is being converted to signed type on assignment in CMSIS Core hardware abstraction layer. |
| 10.3 | R | The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression. | The DMA API has a composite expression of 'essentially unsigned' type (unsigned char) is being cast to a wider unsigned type, 'unsigned long'. This deviation is not present for PSoC 4 cy_boot code. |
| 14.3 | R | Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character. | The CYASSERT() macro has null statement is located close to other code. |
| 11.4 | A | A cast should not be performed between a pointer to object type and a different pointer to object type. | The DMA and Interrupt API use casts between a pointer to object type and a different pointer to object type. |

| MISRA-C: 2004 Rule | Rule Class (R/A)[1] | Rule Description | Description of Deviation(s) |
|---|---|---|---|
| 11.5 | | A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer. | The volatile qualification is lost during pointer cast to pointer to void before passing to the memcpy() function. |
| 14.7 | R | A function shall have a single point of exit at the end of the function. | The CyPmSleep() and CyPmHibernate() functions has complex conditional structure and one more `return` path is added for PSoC 3/PSoC 5LP to return immediately if device is not ready for low power mode entry. This deviation is not present for PSoC 4 cy_boot code. |
| 17.4 | R | Array indexing shall be the only allowed form of pointer arithmetic. | The DMA, Flash and Interrupt APIs use array indexing that are applied to an object of pointer type to access hardware registers, buffer allocated by user and vector tables correspondingly. |
| 19.4 | R | C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct. | The CYASSERT(), INTERRUPT_DISABLE_IRQ, INTERRUPT_ENABLE_IRQ, CyGlobalIntEnable, and CyGlobalIntDisable macro defines a braced code statement block. |
| 19.7 | A | A function should be used in preference to a function-like macro. | Deviated since function-like macros are used to allow more efficient code. |

# 15 cy_boot Component Changes

## Version 4.11

This section lists and describes the major changes in the cy_boot component version 4.11:

| Description of Version 4.11 Changes | Reason for Changes / Impact |
|---|---|
| The CySysFlashWriteRow() function now checks the data to be written and, if necessary, modifies it to have a non-zero checksum. After writing to Flash, the modified data is replaced (Flash program) with the correct (original) data. | Cypress identified a defect with the Flash write functionality of the PSoC 4000, PSoC 4100, and PSoC 4200 devices. The CySysFlashWriteRow() function in the cy_boot [v4.0 and v4.10] component fails to write a row of flash memory if the data to be written has a zero in the lower 32-bits of the checksum. |

## Version 4.10

This section lists and describes the major changes in the cy_boot component version 4.10:

| Description of Version 4.10 Changes | Reason for Changes / Impact |
|---|---|
| PSoC 4: Added CySysGetResetReason() function. | Reports the cause for the latest reset(s) that occurred in the system. |
| Added support for the PSoC 4000 family. | New device support. |
| PSoC 3: Added reentrancy support for the CySpcLock() and CySpcUnlock() functions. | |
| PSoC 3/ PSoC 5LP: Fixed the defect in CyPmRestoreClocks() function, that can might to the device halt during the function execution, in some clock system configurations, when PLL is not sourced by IMO and IMO is manually stopped by user code. | |
| PSoC 4: Added note that enabling or disabling a WDT requires three LFCLK cycles to come into effect, during that period the SYSCLK should be available. | The device should not put into Deep Sleep mode during that period. |
| PSoC 4: Added note that, after waking from Deep Sleep, the WDT internal timer value is set to zero until the ILO loads the register with the correct value. | This led to an increase in low-power mode current consumption.<br><br>The work around is to wait for the first positive edge of the ILO clock before allowing the WDT_CTR_* registers to be read by CySysWdtReadCount() function. |

| Description of Version 4.10 Changes | Reason for Changes / Impact |
|---|---|
| Added note to the Working with Flash and EEPROM section with the information that CPU code execution can be halted till the flash write is complete. | |
| Added note to the Working with Flash and EEPROM section with the information that power manager will not put the device into a low power state if the system performance controller (SPC) is executing a command. | |
| PSoC 3 / PSoC 5LP: The CyPmRestoreClocks() implementation was enhanced by polling status and proceed as soon as PLL is locked. Added merge section to add ability of handling cases when predefined timeout is not enough. | |
| PSoC 4: Fixed a defect in CySysWdtClearInterrupt() that caused unintentional clearing of the WDT interrupt status bit. | |

## Version 4.0

This section lists and describes the major changes in the cy_boot component version 4.0:

| Description of Version 4.0 Changes | Reason for Changes / Impact |
|---|---|
| Added note to the Flash and EEPROM section about unavailability of the Store Configuration Data in ECC Memory DWR option for the bootloader project type. | |
| Added note to the Working with Flash and EEPROM section that when writing Flash, data in the instruction cache can become stale. | Call CyFlushCache() to invalidate the data in cache and force fresh information to be loaded from Flash. |
| Fixed issue in the CyDmaChEnable() and CyDmaChDisable() functions. | If DMA request occurred during these functions, the DMA channels configuration could be corrupted. The APIs were changed to address this problem. |
| Removed references to PSoC 5 device. | PSoC 5 has been replaced by PSoC 5LP. |
| PSoC Creator Generated Sources Deviations section was updated with the MISRA deviations related to the AMuxSeq component. | |
| The CY_IMO_FREQ_74MHZ parameter was added to the CyIMO_SetFreq() function. | Support of the 80 MHZ PSoC 5LP devices. |
| PSoC 4: Added CyExitCriticalSection() function call after WFI instruction in the CySysPmHibernate() function. | If any interrupt occurred between CyEnterCriticalSection() and WFI instruction execution, the device could skip low power mode entry request and continue code execution with global interrupts disabled. |

# Version 3.40 and Older

## Version 3.40

This section lists and describes the major changes in the cy_boot component version 3.40:

| Description of Version 3.40 Changes | Reason for Changes / Impact |
|---|---|
| Added PSoC 4 device support. | New device support. |
| PSoC 3: Updated CyPmSleep() function description with the information that hardware buzz must be disabled before sleep mode entry.<br><br>As hardware buzz is required for LVI, HVI, and Brown Out detect operations – they must be disabled before sleep mode entry and restored on wakeup. If LVI or HVI is enabled, CyPmSleep() will halt device if project is compiled in debug mode. | Using hardware buzz in conjunction with other device wakeup sources can cause the device to lockup, halting further code execution. Refer to the device errata for more information. |

## Version 3.30

This section lists and describes the major changes in the cy_boot component version 3.30:

| Description of Version 3.30 Changes | Reason for Changes / Impact |
|---|---|
| Updates to support PSoC Creator 2.2. | |
| Added MISRA Compliance section. | |
| Added Low Voltage Analog Boost Clocks section. | New feature for the SC-based (TIA, Mixer, PGA and PGA_Inv) components. |
| Added requirement about interrupt configuration, when interrupt is sources from PICU and used as a wakeup event. | For PSoC 5LP, the interrupt component connected to the wakeup source may not use the "RISING_EDGE" detect option. Use the "LEVEL" option instead. |
| The delay between Bus clock and analog clocks configuration save/restore moved from CyPmSleep() and CyPmHibernate() functions to CyPmSaveClocks() / CyPmRestoreClocks(). | This modification decrease CyPmSleep() and CyPmHibernate() functions execution time. The components that use analog clock must not be used after CyPmSaveClocks() execution till the clocks configuration will be restored by CyPmRestoreClocks(). |
| Added float32 and float64 data types. The type float64 is not available for PSoC 3 devices. | |

## Version 3.20

This section lists and describes the major changes in the cy_boot component version 3.20:

| Description of Version 3.20 Changes | Reason for Changes / Impact |
|---|---|
| Many minor edits throughout the document to distinguish features of PSoC 5 and PSoC 5LP devices. | Improve PSoC 5 and PSoC 5LP documentation. |
| The PSoC 5LP Alternate Active usage model was changed to be same as for PSoC 5. | No parameters are used for CyPmAltAct(). That means NONE should be passed for the parameters. The device will go into Alternate Active mode until an enabled interrupt occurs. |

| Description of Version 3.20 Changes | Reason for Changes / Impact |
|---|---|
| The interface of the CyIMO_SetFreq() function was updated for PSoC 5LP to support 62 and 72 MHz frequencies. | Added interface to configure IMO to 62 and 72 MHz on PSoC 5LP. |

## Version 3.10

This section lists and describes the major changes in the cy_boot component version 3.10:

| Description of Version 3.10 Changes | Reason for Changes / Impact |
|---|---|
| The Bootloader system was redesigned in cy_boot version 3.0 to separate the Bootloader and Bootloadable components. The change is listed here as well for migrating from older versions. | See Bootloader Migration on page 77 for more information. |
| A few edits were applied to the Voltage Detect APIs: fixed a typo in the register definition, added CyVdLvDigitEnable() function threshold parameter mask to protect from invalid parameter values, updated CyVdLvDigitEnable() and CyVdLvAnalogEnable() functions to use delay instead of while loop during hardware initialization. | To improve the overall implementation of these APIs. |
| Minor updates to the CyPmSleep() function. | Better support of latest PSoC 3 devices. |

## Version 3.0

This section lists and describes the major changes in the cy_boot component version 3.0:

| Description of Version 3.0 Changes | Reason for Changes / Impact |
|---|---|
| The Bootloader system was redesigned to separate the Bootloader and Bootloadable components. | See Bootloader Migration on page 77 for more information. |
| The CyPmSleep() function implementation was updated to preserve/restore PRES state before/after Sleep mode. The support of the HVI/LVI functionality added. | New functionality support. |
| Added following Voltage Detect APIs: CyVdLvDigitEnable(),CyVdLvAnalogEnable(),CyVdLvDigitDisable(),CyVdLvAnalogDisable(),CyVdHvAnalogEnable(),CyVdHvAnalogDisable(),CyVdStickyStatus() and CyVdRealTimeStatus(). | Added voltage monitoring APIs. |
| The implementation of the Flash API was slightly modified as the SPC API used in Flash APIs was refactored. | The implementation quality improvements. |
| The implementation of the CyXTAL_32KHZ_Start(), CyXTAL_32KHZ_Stop(), CyXTAL_32KHZ_ReadStatus() and CyXTAL_32KHZ_SetPowerMode() APIs was updated. | Added additional timeouts to ensure proper block start-up. |
| The implementation of the CyXTAL_Start() function for PSoC 5 parts was changed. For more information on function see Clocking section. | Changes were made to make sure that MHZ XTAL starts successfully on PSoC 5 parts. |

| Description of Version 3.0 Changes | Reason for Changes / Impact |
|---|---|
| The following APIs were removed for PSoC 5 parts: CyXTAL_ReadStatus(), CyXTAL_EnableErrStatus(), CyXTAL_DisableErrStatus(), CyXTAL_EnableFaultRecovery(), CyXTAL_DisableFaultRecovery(). | The functionality provided within these APIs is not supported by the PSoC 5 part. |
| The CyDmacConfigure() function is now called by the startup code only if DMA component is placed onto design schematic. | Increase device startup time in case if DMA is not used within design. The CyDmacConfigure() function should be called manually if DMA functionality is used without DMA component. |
| The CyXTAL_32KHZ_ReadStatus() function implementation was changed by removing digital measurement status return. | The analog status measurement is the only reliable source. |
| Updated description of following APIs: CyFlash_SetWaitCycles(). | Changes were made to improve power mode configuration. |
| The address of the top of reentrant stack was decremented from CYDEV_SRAM_SIZE to (CYDEV_SRAM_SIZE - 3) for PSoC 3. | Prevent rewriting CyResetStatus variable with the parameters and/or local variables of the reentrant function during its execution. |
| The CyIMO_SetFreq() function implementation was updated by removing support of 74 and 62 MHz parameters for PSoC 5 parts. | Removal of the functionality that is not supported by device. |
| The minimal P divider value for the CyPLL_OUT_SetPQ() was risen from 4 up to 8. | To meet hardware requirements |
| The CyXTAL_SetFbVoltage()/SetWdVoltage() were added for PSoC 5LP devices. | The functionality provided by these APIs is available in PSoC 5LP. |
| The description of the CyWdtStart() was updated. | Added notes on WDT operation during low power modes for PSoC 5. |
| The implementation of the CyPmSleep() for PSoC 5 was changed not to hold CTW in reset on wakeup. | Not putting CTW in reset state on wakeup allows to combine CTW usage in both Active and low power modes for PSoC 5. |
| The *Preservation of Reset Status* section was updated with more detailed information. | The software reset behavior of other resets is explained. Explained how the reset status variable can be used. |
| Updated description of following APIs: CyMasterClk_SetDivider(),CyWdtStart(),CyWdtStart(). | To reflect implementation better. |
| The Startup and Linking section was updated. The information on using custom linker script was added. | To provide more information on device operation. |
| Following macros were removed: CYWDT_TICKS CYWDT_CLEAR, CYWDT_ENABLE CYWDT_DISABLE_AUTO_FEED. | The CyWdtStart() and CyWdtClear() should be used instead. |
| The CyCpuClk_SetDivider() was removed for PSoC 5 devices. | The hardware does not support this functionality. |
| The cystrcpy(), cystrlen(), CyGetSwapReg16() and CySetSwapReg16() APIs were removed. | The library functions should be used. |
| The return value description for CyEnterCriticalSection() function was updated for PSoC 5. | Function returns 0 if interrupts were previously enabled or 1 if interrupts were previously disabled. |

| Description of Version 3.0 Changes | Reason for Changes / Impact |
|---|---|
| Added all APIs with the CYREENTRANT keyword when they are included in the .cyre file. | Not all APIs are truly reentrant. Comments in the component API source files indicate which functions are candidates.<br>This change is required to eliminate compiler warnings for functions that are not reentrant used in a safe way: protected from concurrent calls by flags or Critical Sections. |
| Added PSoC 5LP support | |

# Version 2.40 and Older

## Version 2.40

This section lists and describes the major changes in the cy_boot component version 2.40:

| Description of Version 2.40 Changes | Reason for Changes / Impact |
|---|---|
| Updated the CyPmSleep() and CyPmHibernate() APIs. | Changes were made to improve power mode configuration. |

## Version 2.30

This section lists and describes the major changes in the cy_boot component version 2.30:

| Description of Version 2.30 Changes | Reason for Changes / Impact |
|---|---|
| CyIntEnable and CyIntDisable functions have been changed to be CYREENTRANT by default. | Many components require CyIntEnable and CyIntDisable to be reentrant and these components have no way to cause that to happen. This means you no longer need to populate a cyre file for these functions that you do not call. |
| The implementation of CyPmSleep() and CyPmAltActive() functions were modified by removing 32 KHz ECO,100 KHz and 1 KHz ILO power mode configuration before device low power mode entry. | User was made responsible for clock power modes configuration during Sleep and Alternate Active mode.<br><br>The CyILO_SetPowerMode() and CyXTAL_32KHZ_SetPowerMode() can be used to configure clock power modes.<br><br>The information regarding user responsibility of clock power mode configuration was added at the PM API section. |
| The implementation of the CyPmSaveClocks() was updated to set IMO clock frequency to 48 MHz when "Enable Fast IMO during startup" is enabled, and to 12 MHz otherwise. The IMO frequency is always set to 12 MHz just before the low power mode entry and restored immediately after wakeup. The CyPmRestoreClocks() restores original value of the IMO clock. | The IMO value should match FIMO and FIMO is always 12 MHz. |

| Description of Version 2.30 Changes | Reason for Changes / Impact |
|---|---|
| The implementation of the CyPmRestoreClocks() function was updated by removing restoring MHz ECO and PLL disabled state. | The CyPmRestoreClocks() function is expected to be called only after CyPmSaveClocks(), while the last one always disables MHz ECO and PLL. |
| Global interrupts are disabled on CyPmSleep()/CyPmHibernate() entry and restored before return from the function. | Interrupts are disabled to prevent their occurrence before the state of the device has been restored. |
| The CyPmSleep() and CyPmAltAct() function implementations for PSoC 5 devices were updated to ignore all parameters. The PSoC 5 device will go into Sleep mode until it is woken by an interrupt from one of three wakeup sources: CTW, Once per second, or Port Interrupt Controller (PICU). These wakeup sources must already be configured to generate an interrupt. The CTW is configured using the Sleep Timer component and the Once per second interrupt using the Real Time Clock component. | The CyPmSleep() and CyPmAltAct() functions have to be used only with the following parameters: CyPmSleep(PM_SLEEP_TIME_NONE, PM_SLEEP_SRC_NONE) and CyPmAltAct(PM_ALT_ACT_TIME_NONE, PM_ALT_ACT_SRC_NONE). |
| Updated architecture- and silicon-specific #defines to be used across the content. | |
| Improved performance of non-DMA configuration on 8051 devices. | These modifications decrease the startup time and slightly reduce consumption of code memory and internal data memory. |
| The implementation of the CyPmRestoreClocks() was updated for PSoC 5 silicon. The megahertz crystal is given 130 ms to stabilize. Its readiness is not verified after the hold-off timeout. | These modifications increase crystal startup time, but ensure that crystal is ready to be used. |
| The power mode of the source clocks for the timer used as the wakeup timer was removed from PM API functions. | Before calling PM API function, you must manually configure the power mode of the source clocks for the timer that will be used as the wakeup timer. |
| PSoC Creator Power Management section was updated. | More detailed information on Power Management API usage was added. |

## Version 2.21

This section lists and describes the major changes in the cy_boot component version 2.21:

| Description of Version 2.21 Changes | Reason for Changes / Impact |
|---|---|
| Provide a new option for selecting how to compute checksums on data transferred from the bootloader host to the bootloader. | Provide a more robust way to check for errors during IO transfers. |
| Provide a generic option to allow users to define their own custom bootloader communication functions. | Make adding support for additional communication protocols (SPI, UART, ...) easier. Also provides a means of supporting multiple communications components concurrently in the same design. |
| Updated a few Power Management functions to prevent some possible issues. | Some parentheses were missing which could cause items to be evaluated in the wrong order. |

| Description of Version 2.21 Changes | Reason for Changes / Impact |
|---|---|
| Added a variable CyResetStatus that can be used to get the information from the RESET_SR0 register. | This is provided because many of the fields contained within the RESET_SR0 register are clear-on-read. Since the bootloader needs to access this register as part of its operation, it prevents the actual application code from accessing the values. The variable is provided so that the application can still get access to all the information. |
| Added a workaround for some PSoC 3 devices to ensure that the NVL values have been properly initialized. | On some PSoC 3 devices the NVL information may not be initialized properly. This workaround is provided to ensure that the NVLs are properly loaded before performing any of the startup code. |

## Version 2.20

Version 2.20 and older are obsolete.