PSoC® Creator™

System Reference Guide

cy_boot Component v3.10

Document Number: 001-80778, Rev. *A

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, 2012. 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

3 Clocking 11
- PSoC Creator Clocking Implementation 11
- APIs 20

4 Power Management 31
- Clock Configuration 31
- Wakeup Time Configuration 32
- Wakeup Source Configuration 32
- APIs 33
- Instance Low Power APIs 39

5 Interrupts 41
- APIs 41

6 Cache 45
- PSoC 3 Cache Functionality 45
- PSoC 5 Cache Functionality 45

7 Pins 47
- APIs 47

8 Register Access 49
APIs 49

9 DMA 53

10 Flash and EEPROM 55
Memory Architecture 55
ECC 56
Working with Flash and EEPROM 56
APIs 57

11 Bootloader Migration 63
Introduction 63
Migrating Bootloader Designs 64
Migrating Bootloadable Designs 65
Project Application Type Property Changes 65
Migrating Bootloadable Dependency to Bootloader Design 65

12 System Functions 67
General APIs 67
CyDelay APIs 68
Voltage Detect APIs 69

13 Startup and Linking 75
PSoC 3 76
PSoC 5 76
Preservation of Reset Status 77

14 Watchdog Timer 79
APIs 79

15 cy_boot Component Changes 81
Version 3.10 81
Version 3.0 81
Version 2.40 83
Version 2.30 83
Version 2.21 84
Version 2.20 85
Version 2.10 86
Version 2.0 86

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
- Flash
- Clocking
- Power management
- Start up 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: `C:\ …cd\icc\,` user entered text
Italics	Displays file names and reference documentation: *sourcefile.hex*
[**bracketed, bold**]	Displays keyboard commands in procedures: [**Enter**] or [**Ctrl**] [**C**]
File > New Project	Represents menu paths: **File > New Project > Clone**
Bold	Displays commands, menu paths and selections, and icon names in procedures: Click the **Debugger** icon, and then click **Next**.
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 v3.10 **Document Number: 001-80778**		
Revision	**Date**	**Description of Change**
**	6/20/12	New document for version 3.10 of the cy_boot component. Refer to the change section for component changes from previous versions of cy_boot.
*A	6/29/12	Update to CyPmSleep() description for sleep time duration.

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

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_PSOC5A	PSoC 5 Device
CY_PSOC5LP	PSoC 5LP Device

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.

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.

BUS_CLK

This is a special clock. It is closely related to MASTER_CLK. For most designs, MASTER_CLK and BUS_CLK 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 BUS_CLK and all the peripherals will communicate to the CPU and DMA using BUS_CLK. When a clock is synchronized, it is synchronized to MASTER_CLK. When a pin is synchronized it is synchronized to BUS_CLK.

Global Clock

This is a clock that is placed on one of the global low skew digital clock lines. This also includes BUS_CLK. 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 BUS_CLK and MASTER_CLK. 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

Examples of synchronous clocks include:

- Global clock with sync to MASTER_CLK 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

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

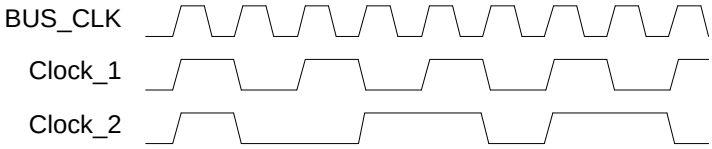
Depending on the source of the clock signal, it can be made synchronous using different methods:

- 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 BUS_CLK. Clock_1 has exactly 2x the period of BUS_CLK. Clock_2 has a period of approximately 3x the period of BUS_CLK. That results in the high and low pulse widths varying between 1 and 2 BUS_CLK periods. In both cases all transitions occur at the rising edge of BUS_CLK.



Routed Clock Implementation

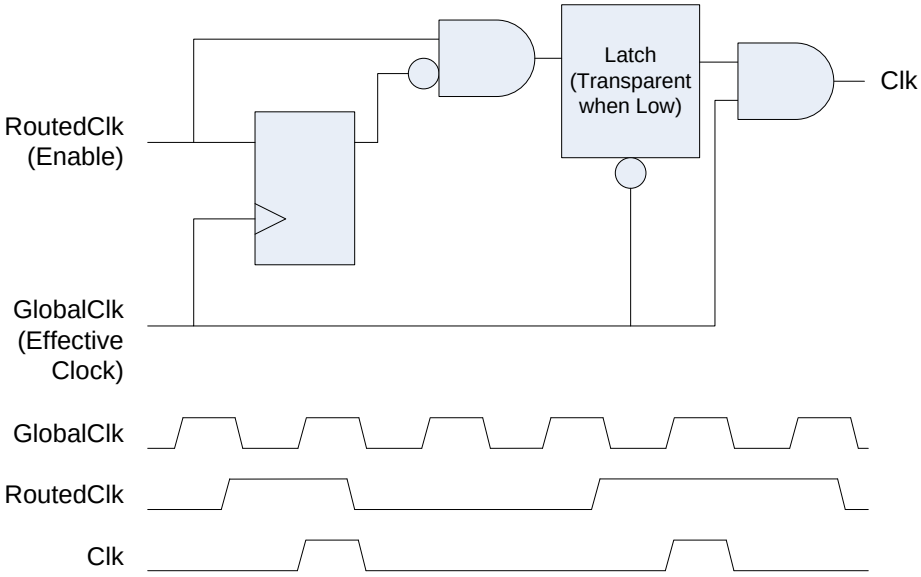
The clocking implementation in PSoC directly connects global clock signals to the clock input of clocked digital logic. This applies to both synchronous and asynchronous clocks. Since global clocks are distributed on low skew clock lines, all clocked elements connected to the same global clock will be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

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.



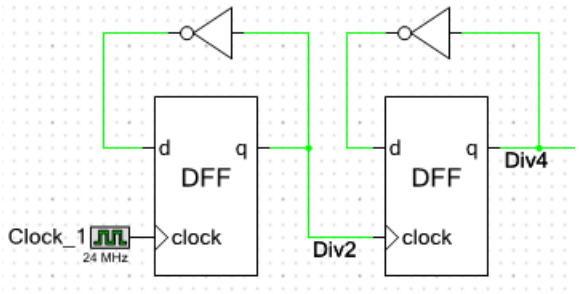
This diagram shows that the resulting clock occurs synchronous to the global clock on the first clock after a rising edge of the routed clock.

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

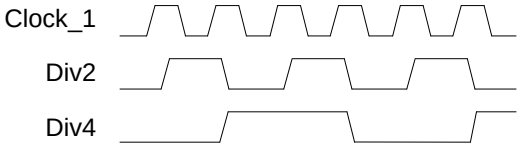
A simple divided clock circuit can be used to observe how this transformation is done. The following circuit clocks the first flip-flop (cydff_1) with a global clock. This generates a clock that is divided by 2 in frequency. That signal is used as a routed clock that clocks the next flip-flop (cydff_2).



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>
```

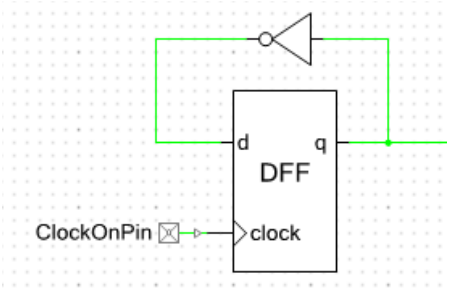
The resulting signals generated by this circuit are as follows.



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 BUS_CLK, the transformed circuit uses BUS_CLK as the Effective Clock and uses the rising edge of the pin as the Enable Signal.



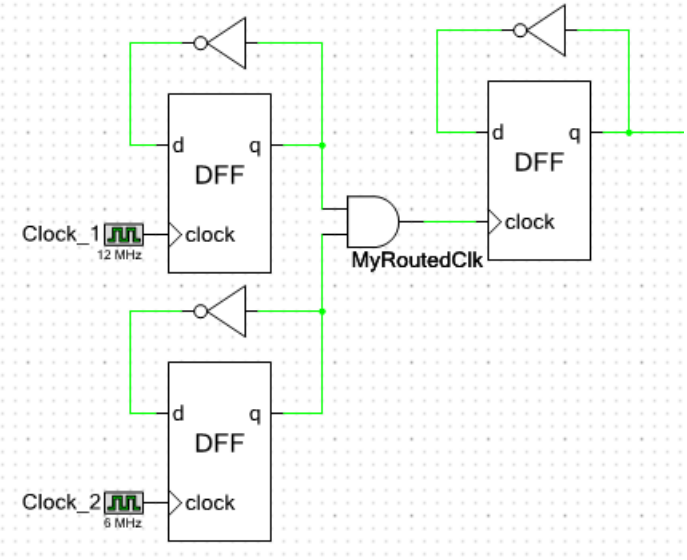

```
<CYPRESSTAG name="Initial Mapping" icon="FILE_RPT_TECHM">
{Global Clock Selection}
<CYPRESSTAG name="UDB Routed Clock Assignment">
    Routed Clock: ClockOnPin(0):iocell.fb
        Effective Clock: BUS_CLK
        Enable Signal: ClockOnPin(0):iocell.fb
</CYPRESSTAG>
```

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 BUS_CLK is the common global clock that becomes the Effective Clock.

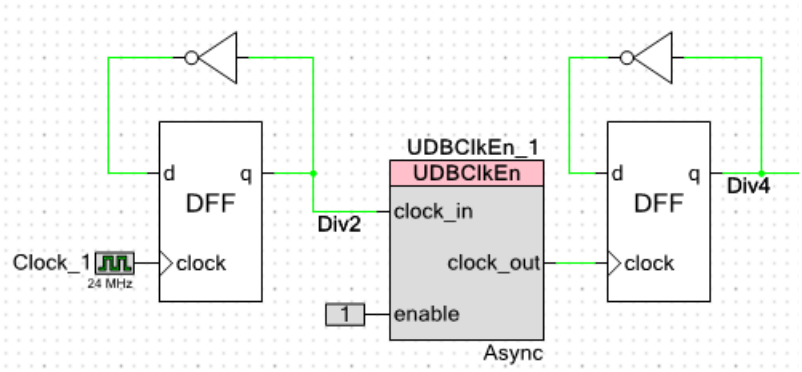



```
<CYPRESSTAG name="Tech mapping">
<CYPRESSTAG name="Initial Mapping" icon="FILE_RPT_TECHM">
<CYPRESSTAG name="Global Clock Selection" icon="FILE_RPT_TECHM">
    Digital Clock 0: Automatic-assigning  clock 'Clock_1'. Fanout=1, Signal=tmp__cydff_1_clk
    Digital Clock 1: Automatic-assigning  clock '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

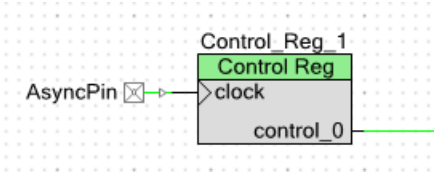
The automatic transformation that PSoC Creator performs on routed clocks is generally the implementation that should be used. There is however a method to force the routed clock to be used directly. The UDBClkEn component configured in Async mode will force the clock used to be the routed clock, as shown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

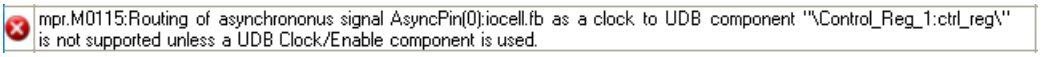


Using Asynchronous Clocks

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.





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 BUS_CLK.

When both clocks are synchronous to BUS_CLK, 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 BUS_CLK not the speed of either of the clock domains.

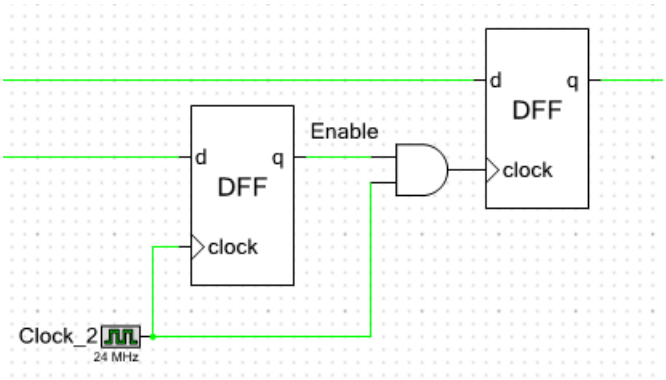
The only guarantee between the clock domains is that their edges will always occur on a rising edge of BUS_CLK. That means that the rising edges of the two clock domains can be as close as a single BUS_CLK cycle apart. This is true even when the clock domains are multiples of each other, since their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

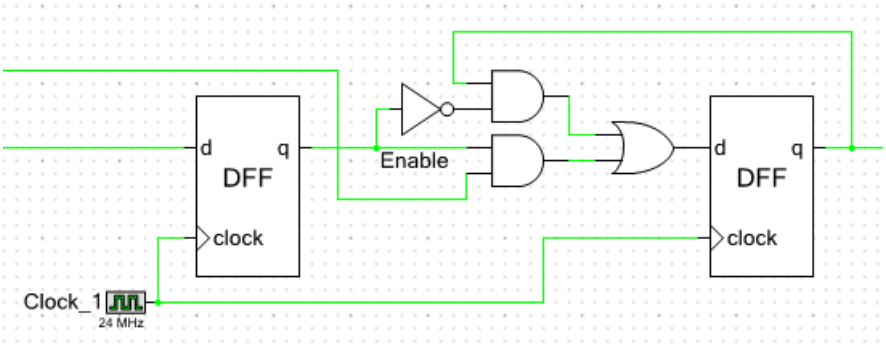
Gated Clocks

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.

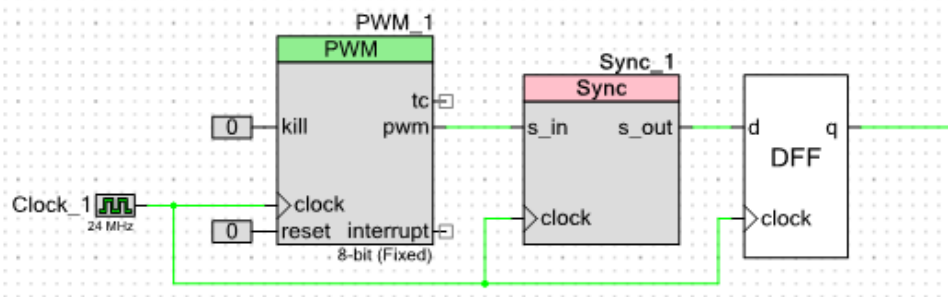
The following circuit implements the equivalent function and is supported by timing analysis, only uses global clocks, and has no reliability issues. This circuit does not gate the clock, but instead logically enables the clocking of new data or maintains the current data.



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.



Cache Configuration

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.

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 to time the wait. Any other use of the Fast Time Wheel 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, Fast Time Wheel, Central Time Wheel or Once Per Second interrupt may be made by interrupt routines for the duration of this function execution. The current operation of the ILO, Central Time Wheel 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 P, uint8 Q, 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

Value	Define	Source
0	CY_PLL_SOURCE_IMO	IMO
1	CY_PLL_SOURCE_XTAL	MHz Crystal
2	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 Fast Time Wheel to time the wait. Any other use of the Fast Time Wheel 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, Fast Time Wheel, Central Time Wheel or Once Per Second interrupt may be made by interrupt routines for the duration of this function execution. The current operation of the ILO, Central Time Wheel 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

Value	Define	Frequency
0	CY_IMO_FREQ_3MHZ	3 MHz
1	CY_IMO_FREQ_6MHZ	6 MHz
2	CY_IMO_FREQ_12MHZ	12 MHz
3	CY_IMO_FREQ_24MHZ	24 MHz
4	CY_IMO_FREQ_48MHZ	48 MHz
5	CY_IMO_FREQ_62MHZ	62.6 MHz (PSoC 3 only)
8	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

Value	Define	Source
0	CY_IMO_SOURCE_IMO	IMO
1	CY_IMO_SOURCE_XTAL	MHz Crystal
2	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 CyMasterClk_SetSource(uint8 source)

Description: Sets the source of the master clock.

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

Value	Define	Source
0	CY_MASTER_SOURCE_IMO	IMO
1	CY_MASTER_SOURCE_PLL	PLL
2	CY_MASTER_SOURCE_XTAL	MHz Crystal
3	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 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 CyUsbClk_SetSource(uint8 source)

Description: Sets the source of the USB clock.

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

Value	Define	Source
0	CY_USB_SOURCE_IMO2X	IMO 2x
1	CY_USB_SOURCE_IMO	IMO
2	CY_USB_SOURCE_PLL	PLL
3	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

Value	Define	Source
0	CY_ILO_SOURCE_100K	ILO 100 KHz
1	CY_ILO_SOURCE_33K	ILO 33 KHz
2	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:

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

Return Value: Previous power mode

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

Value	Define	Source
0x20	CY_XTAL32K_ANA_STAT	Analog measurement 1: Stable 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

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.

PSoC 5 device:
Waits for CY_CLK_XMHZ_MIN_TIMEOUT milliseconds (or number of milliseconds specified by the parameter if it is greater than CY_CLK_XMHZ_MIN_TIMEOUT. The XERR bit status is not checked.

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 Fast Time Wheel to time the wait. Any other use of the Fast Time Wheel 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, Fast Time Wheel, Central Time Wheel or Once Per Second interrupt may be made by interrupt routines for the duration of this function execution. The current operation of the ILO, Central Time Wheel 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. This function is not available for PSoC 5 devices.

Parameters: None

Return Value: None

void CyXTAL_DisableErrStatus()

Description: Disables the generation of the XERR status bit for the megahertz crystal. This function is not available for PSoC 5 devices.

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. This function is not available for PSoC 5 devices.

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. This function is not available for PSoC 5 devices.

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. This function is not available for PSoC 5 devices.

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. This function is not available for PSoC 5 devices.

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. This function is only available for PSoC 3 and PSoC 5LP. This function is not available for PSoC 5 devices.

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.

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

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

Note For PSoC 3 and PSoC 5, 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.

For PSoC 5 architectures, 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 it is masked.

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 for all devices for correct operation in Sleep and Hibernate low power modes. For PSoC 3 devices, the 1 KHz ILO 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.

Caution Do not modify this register.

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: Central Time Wheel (CTW), Fast Timer Wheel (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. For PSoC 5 silicon, the wakeup source is not selectable. In this case, the wakeup source argument is ignored and any of the available wakeup sources will wake the device. For PSoC 5 silicon, the CTW is the only supported wakeup source from sleep.

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.

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. 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.

PSoC 5: The megahertz crystal is given up to 130 ms to stabilize. Its readiness is not verified 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.

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 SleepTimer or RTC components, then specify NONE for the wakeupTime and include the appropriate source for wakeupSource.

PSoC 5: Neither parameter is used for PSoC 5. 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 5 this parameter is ignored.

Value	Define	Time
0	PM_ALT_ACT_TIME_NONE	None
1	PM_ALT_ACT_TIME_ONE_PPS	One PPS: 1 second
2	PM_ALT_ACT_TIME_CTW_2MS	CTW: 2 ms
3	PM_ALT_ACT_TIME_CTW_4MS	CTW: 4 ms
4	PM_ALT_ACT_TIME_CTW_8MS	CTW: 8 ms
5	PM_ALT_ACT_TIME_CTW_16MS	CTW: 16 ms
6	PM_ALT_ACT_TIME_CTW_32MS	CTW: 32 ms
7	PM_ALT_ACT_TIME_CTW_64MS	CTW: 64 ms
8	PM_ALT_ACT_TIME_CTW_128MS	CTW: 128 ms
9	PM_ALT_ACT_TIME_CTW_256MS	CTW: 256 ms
10	PM_ALT_ACT_TIME_CTW_512MS	CTW: 512 ms
11	PM_ALT_ACT_TIME_CTW_1024MS	CTW: 1024 ms
12	PM_ALT_ACT_TIME_CTW_2048MS	CTW: 2048 ms
13	PM_ALT_ACT_TIME_CTW_4096MS	CTW: 4096 ms
14 - 269	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 5 this parameter is ignored.

Value	Define	Source
0	PM_ALT_ACT_SRC_NONE	None
1	PM_ALT_ACT_SRC_COMPARATOR0	Comparator 0
2	PM_ALT_ACT_SRC_COMPARATOR1	Comparator 1
4	PM_ALT_ACT_SRC_COMPARATOR2	Comparator 2
8	PM_ALT_ACT_SRC_COMPARATOR3	Comparator 3
16	PM_ALT_ACT_SRC_INTERRUPT	Interrupt
64	PM_ALT_ACT_SRC_PICU	PICU
128	PM_ALT_ACT_SRC_I2C	I2C
512	PM_ALT_ACT_SRC_BOOSTCONVERTER	Boost Converter
1024*	PM_ALT_ACT_SRC_FTW	Fast Time Wheel
1024*	PM_ALT_ACT_SRC_VD	High and Low Voltage Detection
2048*	PM_ALT_ACT_SRC_CTW	Central Time Wheel
2048*	PM_ALT_ACT_SRC_ONE_PPS	One PPS
4096	PM_ALT_ACT_SRC_LCD	LCD

Note CTW and One PPS wakeup signals are in the same mask bit. FTW and HVI/LVI 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 5 silicon, 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.

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 SleepTimer or RTC components, then specify NONE for the wakeupTime and include the appropriate source for wakeupSource.

PSoC 5: Neither parameter to this function is used for PSoC 5. That means NONE should be passed for the parameters. The device will go into Sleep mode until it is woken by an interrupt from the Central Time Wheel (CTW). The CTW must already be configured to generate an interrupt. It is configured using the SleepTimer component. Only the CTW can be used to wake the device from sleep mode. This function automatically disables other interrupt sources and then restores them after the device is woken by the CTW.

The duration of sleep needs to be controlled so that the device doesn't wake up too soon after going to sleep or remain asleep for too long. Reliable sleep times of between 1 ms and 128 ms can be supported. This requirement is satisfied with CTW settings of 4, 8, 16, 32, 64, 128, or 256 ms. To control the sleep timer, the CTW is reset automatically just before putting the device to sleep. The resulting wakeup time is half the duration programmed into the CTW with an uncertainty of 1 ms due to the arrival time of the first ILO clock edge. For example, the setting of 4 ms will result in a sleep time between 1 ms and 2 ms.

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

Value	Define	Time
0	PM_SLEEP_TIME_NONE	None
1	PM_SLEEP_TIME_ONE_PPS	One PPS: 1 second
2	PM_SLEEP_TIME_CTW_2MS	CTW: 2 ms
3	PM_SLEEP_TIME_CTW_4MS	CTW: 4 ms
4	PM_SLEEP_TIME_CTW_8MS	CTW: 8 ms
5	PM_SLEEP_TIME_CTW_16MS	CTW: 16 ms
6	PM_SLEEP_TIME_CTW_32MS	CTW: 32 ms
7	PM_SLEEP_TIME_CTW_64MS	CTW: 64 ms
8	PM_SLEEP_TIME_CTW_128MS	CTW: 128 ms
9	PM_SLEEP_TIME_CTW_256MS	CTW: 256 ms

CyPmSleep (Continued)

Parameters: wakeupTime (continued)

Value	Define	Time
10	PM_SLEEP_TIME_CTW_512MS	CTW: 512 ms
11	PM_SLEEP_TIME_CTW_1024MS	CTW: 1024 ms
12	PM_SLEEP_TIME_CTW_2048MS	CTW: 2048 ms
13	PM_SLEEP_TIME_CTW_4096MS	CTW: 4096 ms

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 5 this parameter is ignored.

Value	Define	Source
0	PM_SLEEP_SRC_NONE	None
1	PM_SLEEP_SRC_COMPARATOR0	Comparator 0
2	PM_SLEEP_SRC_COMPARATOR1	Comparator 1
4	PM_SLEEP_SRC_COMPARATOR2	Comparator 2
8	PM_SLEEP_SRC_COMPARATOR3	Comparator 3
64	PM_SLEEP_SRC_PICU	PICU
128	PM_SLEEP_SRC_I2C	I2C
512	PM_SLEEP_SRC_BOOSTCONVERTER	Boost Converter
1024	PM_SLEEP_SRC_VD	High and Low Voltage Detection
2048*	PM_SLEEP_SRC_CTW	Central Time Wheel
2048*	PM_SLEEP_SRC_ONE_PPS	One PPS
4096	PM_SLEEP_SRC_LCD	LCD

Note CTW and One PPS wakeup signals are in the same mask bit. For PSoC 5, these are in a different bit (value 1024).

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 for PSoC 3 silicon 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.
PSoC 5: The only method supported for waking up from the Hibernate state is a hardware reset of the device. PICU interrupt sources are automatically disabled by this function before putting the device into the Hibernate 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 for PSoC 3 silicon 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, Central Time Wheel, and Fast Time Wheel 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

Value	Define	Source
1	CY_PM_FTW_INT	Fast Time Wheel
2	CY_PM_CTW_INT	Central Time Wheel
4	CY_PM_ONEPPS_INT	One Pulse Per Second

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

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 (sleep or hibernate). 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.
Call the _Sleep() function before calling the CyPmSleep() or the CyPmHibernate() 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 the interrupt enable for each interrupt.

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 Cache

PSoC 3 Cache Functionality

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 5 Cache Functionality

void CyFlushCache()

Description: Flushes the PSoC 5 cache by invalidating all entries.

Parameters: None

Return Value: None

7 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. These macros all make use of the port pin configuration register that is available for every pin on the 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.

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 5)

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 5)

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 5)

mode: Desired drive mode

Define	Source
PIN_DM_ALG_HIZ	Analog HiZ
PIN_DM_DIG_HIZ	Digital HiZ
PIN_DM_RES_UP	Resistive pull up
PIN_DM_RES_DWN	Resistive pull down
PIN_DM_OD_LO	Open drain - drive low
PIN_DM_OD_HI	Open drain - drive high
PIN_DM_STRONG	Strong CMOS Output
PIN_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 5)

Return Value: Current drive mode for the pin

Define	Source
PIN_DM_ALG_HIZ	Analog HiZ
PIN_DM_DIG_HIZ	Digital HiZ
PIN_DM_RES_UP	Resistive pull up
PIN_DM_RES_DWN	Resistive pull down
PIN_DM_OD_LO	Open drain - drive low
PIN_DM_OD_HI	Open drain - drive high
PIN_DM_STRONG	Strong CMOS Output
PIN_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 5)

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 5)

Return Value: None

8 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.h*, *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 5 processor architecture uses little endian ordering.

SRAM and Flash storage in both the PSoC 3 and PSoC 5 architectures is done using the endianness of the architecture and compilers. However, the registers in both 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 5 will perform accesses using the appropriate 8-, 16- and 32-bit accesses. The PSoC 5 does not require these accesses to be aligned to the width of the transaction.

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 5)

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 5)

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 5)

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 5)

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 5)

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 5)

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 5)

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 5)

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 5.

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 5.

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 5.

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 5.

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 5.

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 5.

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 5.

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 5.

Parameters: reg: Register address

value: Value to write

Return Value: None

9 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.

10 Flash and EEPROM

Memory Architecture

Flash memory in PSoC devices provides nonvolatile storage for user firmware, user configuration data, bulk data storage, and optional ECC data. PSoC EEPROM memory is a byte-addressable nonvolatile memory. See the device datasheet and TRM for more information.

Flash is organized as a set of arrays. Each array consists of 64, 128, or 256 rows. The PSoC 3 architecture has one Flash array, the size of which is 16 KB, 32 KB, or 64 KB plus ECC bytes. Therefore, the only valid array ID for Flash is 0. The PSoC 5 architecture has either one array of 128 or 256 rows, or multiple arrays of 256 rows each. The following table provides Flash/EEPROM definitions that can be used for device-specific numbers. The number of rows in an array can be determined by dividing the size of the array by the size of the row. Each row contains 256 data bytes plus 32 bytes for either error correction code (ECC) (Enable ECC option in the design-wide resources System Editor) or data storage (when ECC is disabled. The device configuration data or user data can be stored there based on Store Configuration Data in ECC Memory option).

An EEPROM is also organized as a set of arrays. Both PSoC 3 and PSoC 5 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.

The Flash and EEPROM API provide following device-specific definitions:

Value	Description
CY_FLASH_BASE	The base pointer of the Flash memory.
CY_FLASH_SIZE	The size of the Flash memory.
CY_FLASH_SIZEOF_ARRAY	The size of Flash array.
CY_FLASH_SIZEOF_ROW	The size of the Flash row.
CY_FLASH_SIZEOF_ECC_ROW	The size of the ECC row.
CY_FLASH_NUMBER_ROWS	The number of Flash row.
CY_FLASH_NUMBER_ARRAYS	The number of Flash arrays.
CY_EEPROM_BASE	The base pointer of the EEPROM memory.
CY_EEPROM_SIZE	The size of the EEPROM memory.
CY_EEPROM_SIZEOF_ARRAY	The size of EEPROM array.
CY_EEPROM_SIZEOF_ROW	The size of the EEPROM row.
CY_EEPROM_NUMBER_ROWS	The number of EEPROM row.
CY_EEPROM_NUMBER_ARRAYs	The number of EEPROM arrays.

ECC

You can enable ECC for high reliability applications. ECC can correct one bit error and detect multiple bit errors per 8 bytes. For more information on using ECC, refer to the *Flash Program Memory* chapter of the TRM.

PSoC 5 devices do not provide error detection/correction functions for the Flash, but ECC flash space can be used for data storage. When ECC is used for data storage, device configuration or user data is stored there. The most common usage in this case is to store configuration data, which is directly supported by PSoC Creator.

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:

- the entire row including ECC
- the entire row without ECC
- just the ECC memory.

Working with Flash and EEPROM

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 APIs simplify interacting with the SPC by abstracting the details away.

Both 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.

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 while performing Flash and EEPROM write operations.

If the Enable ECC option is disabled, you also need to allocate the buffer and pass it into the CySetFlashEEBuffer() function. 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 flash row numbering starts from 0 for each array ID.

APIs

cystatus CySetTemp()

Description: Gets the temperature of the die and leaves the result in a static location used by the Flash and EEPROM writing functions. This function must be called once before executing a series of Flash / EEPROM writing functions.

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.

Flash configuration	Description
Enable ECC – ON 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. The size of the data equals the size of the flash row.
Enable ECC – OFF Store Configuration Data in ECC Memory – ON	Data are written to both flash and ECC rows. 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 Store Configuration Data in ECC Memory – OFF	Data are written to both flash and ECC rows. 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.

Flash configuration	Description
Enable ECC – ON 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. The size of the data that is passed to this function equals the size of the flash row.
Enable ECC – OFF Store Configuration Data in ECC Memory – ON/OFF	Data are written to the flash memory. 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 *rowData)

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

Flash configuration	Description
Enable ECC – ON 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 Store Configuration Data in ECC Memory – ON	This function is not available for this configuration as the ECC is stored in the ECC memory.
Enable ECC – OFF Store Configuration Data in ECC Memory – OFF	Data are written to the ECC row. 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 *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

void CyFlash_Start()

Description: Enables the Flash. By default Flash is enabled.
For PSoC 5, the same bit controls both EEPROM and Flash. Starting or stopping either will cause both to be started or stopped.

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.
For PSoC 5, the same bit controls both EEPROM and Flash. Starting or stopping either will cause either to be started or stopped.

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.
For PSoC 5, the same bit controls both EEPROM and Flash. Starting or stopping either will cause either to be started or stopped. The EEPROM is enabled by default for PSoC 5.

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.
For PSoC 5, the same bit controls both EEPROM and Flash. Starting or stopping either will cause either to be started or stopped.

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

11 Bootloader Migration

This chapter covers information about the new bootloader architecture in 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
- Migrating to PSoC Creator 2.1 without cy_boot Component Update
- Migrating to PSoC Creator 2.1 for Bootloadable Designs

Complete Migration to PSoC Creator 2.1

In this scenario, PSoC Creator 2.1 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 without cy_boot Component Update

In this scenario, PSoC Creator 2.1 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 require cy_boot version 3.0 or above, so you cannot update them without updating cy_boot.

Migrating to PSoC Creator 2.1 for Bootloadable Designs Only

In this scenario, only the bootloadable project is migrated to PSoC Creator 2.1 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 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.

Previous PSoC Creator releases (Bootloader section in the System tab of the DWR)	**PSoC Creator 2.1 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 2.1 bootloader system features, refer to the Bootloader component datasheet.

Migrating Bootloadable Designs

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

Previous PSoC Creator releases (Bootloadable section in the System tab of the DWR)	PSoC Creator 2.1 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).

For more details about the PSoC Creator 2.1 bootloader system features, refer to the Bootloadable component datasheet.

Project Application Type Property Changes

In previous PSoC Creator releases, the **Application Type** option was used to choose the type of the application that will be produced by a build of a project. In 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 previous PSoC Creator releases, 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 the 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.

12 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 whether the device architecture is PSoC 3 or PSoC 5).

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 both PSoC 3 and PSoC 5 architectures. Therefore, to avoid corrupting 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 5 – 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 5, 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 5, 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

Voltage Detect APIs

Voltage monitoring circuits in PSoC 3 and PSoC 5 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 (not available for PSoC 5).

Device	Reset Value	Definition	Register [bits]
PSoC 3	0	Interrupt on digital LVI event	RESET_CR3 [6]
	Non-zero	Reset on digital LVI event	
PSoC 5	Any Value	Interrupt on digital LVI event	
PSoC 5LP	0	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.

Device	Threshold Value	Definition	Register [bits]
PSoC 3	0x00 to 0x0F	Range of 1.70 V (0x00) to 5.45 V (0x0F).	RESET_CR0 [3:0]
PSoC 5	0x03 to 0x0F	Range of 2.45 V (0x03) to 5.45 V (0x0F).	
PSoC 5LP	0x00 to 0x0F	Range of 1.70 V (0x00) to 5.45 V (0x0F).	

Note For PSoC 5, a value less than 0x03 is not valid and must not be passed.

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 (not available for PSoC 5).

Device	Reset Value	Definition	Register [bits]
PSoC 3	0	Interrupt on analog LVI event	RESET_CR3 [7]
	Non-zero	Reset on analog LVI event	
PSoC 5	Any Value	Interrupt on analog LVI event	
PSoC 5LP	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.

Device	Threshold Value	Definition	Register [bits]
PSoC 3	0x00 to 0x0F	Range of 1.70 V (0x00) to 5.45 V (0x0F).	RESET_CR0 [7:4]
PSoC 5	0x03 to 0x0F	Range of 2.45 V (0x03) to 5.45 V (0x0F).	
PSoC 5LP	0x00 to 0x0F	Range of 1.70 V (0x00) to 5.45 V (0x0F).	

Note For PSoC 5, a value less than 0x03 is not valid and must not be passed.

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 low 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.

Device	Value	Definition	Register [bits]
PSoC 3 / PSoC 5 / PSoC 5LP	CY_VD_LVID (0x01)	Persistent status of digital LVI.	RESET_SR0 [0]
	CY_VD_LVIA (0x02)	Persistent status of analog LVI.	RESET_SR0 [1]
	CY_VD_HVIA (0x04)	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.

Device	Value	Definition	Register [bits]
PSoC 3 / PSoC 5 / PSoC 5LP	0x01	Real-time status of digital LVI.	RESET_SR0 [0]
	0x02	Real-time status of analog LVI.	RESET_SR0 [1]
	0x04	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.

13 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 boot loader
- Call main() C entry point

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

PSoC 5

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 5 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.

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 following support is provided:

- Core Peripheral Access Layer
  - core_cm3.c
  - core_cm3.h

These files are used without modification. The same files work for all of our supported platforms.

GCC Implementation

Use all the standard GCC libraries (libcs3, libc, libcs3unhosted, libgcc, libcs3micro). All of these libraries are linked in by default.

Preservation of Reset Status

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
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

14 Watchdog Timer

APIs

void CyWdtStart(uint8 ticks, uint8 lpMode)

Description: Enables the watchdog timer. The timer is configured for the specified count interval, the Central Time Wheel 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 Central Time Wheel (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.

PSoC 5: The watchdog timer should not be used during sleep modes. Since the WDT cannot be disabled after it is enabled, the WDT timeout period can be set to be greater than the sleep wakeup period, then feed the dog on each wakeup from Sleep.

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

Value	Define	Time
0	CYWDT_2_TICKS	4 – 6 ms
1	CYWDT_16_TICKS	32 – 48 ms
2	CYWDT_128_TICKS	256 – 384 ms
3	CYWDT_1024_TICKS	2.048 – 3.072 s

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

Parameters lpMode: Low power mode configuration. This parameter is ignored for PSoC 5. The WDT always acts as if CYWDT_LPMODE_NOCHANGE is passed.

Value	Define	Effect
0	CYWDT_LPMODE_NOCHANGE	No Change
1	CYWDT_LPMODE_MAXINTER	Switch to longest timer mode during sleep / hibernate
3	CYWDT_LPMODE_DISABLED	Disable WDT during sleep / hibernate

Return Value: None

Side effects and restrictions **PSoC 5:** The ILO 1 KHz must be enabled for proper WDT operation. Stopping the ILO 1 kHz could break the active WDT functionality.

void CyWdtClear()

Description: Clears (feeds) the watchdog timer.

Parameters: None

Return Value: None

15 cy_boot Component Changes

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 63 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 63 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.

Description of Version 3.0 Changes	Reason for Changes / Impact
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.
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 are not supported by 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.

Description of Version 3.0 Changes	Reason for Changes / Impact
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.
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. 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

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. The CyILO_SetPowerMode() and CyXTAL_32KHZ_SetPowerMode() can be used to configure clock power modes. The information regarding user responsibility of clock power mode configuration was added at the PM API section.

Description of Version 2.30 Changes	Reason for Changes / Impact
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.
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: Central Time Wheel (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 SleepTimer 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.

Description of Version 2.21 Changes	Reason for Changes / Impact
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.
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

This section lists and describes the major changes in the cy_boot component version 2.20:

Description of Version 2.20 Changes	Reason for Changes / Impact
Updated the CyDelayCycles function to work with instruction cache enabled.	The original CyDelayCycles function was designed to be used with the instruction cache disabled. For PSoC 3 ES3 and Production silicon, when the instruction cache is enabled, the length of the delay was no longer correct.
Updated comments in the code and descriptions in this guide for low power mode functions.	The comments and descriptions for the CyPmSleep(), CyPmHibernate(), and CyPmAltAct() functions we clarified to refer to the silicon errata for a PSoC 3 ES2 and PSoC 5 defect.
Fixed a bootloader issue with the CyBtldr_ValidateApp function.	Fixed an issue in the bootloader for PSoC 5 that would cause it to fail to verify the application code if the application code was larger than 64 K.
Addressed a bootloader wait for traffic issue.	Modified what is required for the bootloader to consider that there is activity and that someone appears to be attempting to communicate. Previously, if the Bootloader received any data over the selected communications component it would then wait forever for data. With this change, the bootloader will now only wait forever if it has received the Enter Bootloader command.
Updated the CySetTemp function to read the die temperature twice to make sure it has a stable value.	Change was made to address an issue. There is no impact.

Version 2.10

This section lists and describes the major changes in the cy_boot component version 2.10:

Description of Version 2.10 Changes	Reason for Changes / Impact
Updated Flash verification code in the bootloader	Fixed an issue in the bootloader's Verify routine that could cause a failure to be reported even for a valid image if flash was disabled.

Version 2.0

This section lists and describes the major changes in the cy_boot component version 2.0:

Description of Version 2.0 Changes	Reason for Changes / Impact
Keil C51 keywords available as macros	These macros allow code to specify the memory space of variables and pointers while maintaining compatibility with other compilers. The most commonly used keywords are CYCODE, CYXDATA, and CYDATA, which represent C51's code, xdata, and data keywords, respectively. On other compilers, these macros are ignored.
CyLib APIs are no longer conditional	It is no longer necessary to define macros such as CYLIB_POWER_MANAGEMENT to include API functions from CyLib.
Add separate section for .dma_init in RealView linker script	This prevents potential errors and warnings when building projects in RealView with DMA-based configuration enabled.
Initialize complete SRAM interrupt vector table	In previous versions, only the first 32 entries of the table were being initialized.
Initialize reserved interrupt vectors to default interrupt handler	In previous versions, the reserved vectors were initialized to 0, which is not a valid interrupt service routine address.
Fix incorrect endian in CyIntSetVector return value on PSoC 3	In previous versions, the high and low bytes were swapped.
Apply interrupt attribute to interrupt service routine declarations	Interrupt service routines declared with CY_ISR/CY_ISR_PROTO/cyisraddress now have the interrupt attribute, which causes the compiler to emit code to adjust the stack alignment. RealView checks for the interrupt attribute as part of the type, so interrupt service routines or vectors that do not have the interrupt attribute will fail to compile on RealView. For this reason, cy_isr_v1_20 and earlier are not compatible with cy_boot_v1_50 when using RealView. This affects PSoC 5 only; the interrupt attribute was already present in the PSoC 3 version.
Make CY_GET_REGxx/CY_SET_REGxx reentrant and improve performance	The functions that implemented this functionality, CyGetRegXX and CySetRegXX, have been replaced with assembly routines. The new routines may be safely used in interrupt service routines.
PSoC 3 ES3 support in CyDelay	CPUCLK_DIV has been moved from a SFR to the CLKDIST_MSTR1 in ES3.

Description of Version 2.0 Changes	Reason for Changes / Impact
Use *limits.h* and *ctype.h* to replace some macros in *CyLib.h*	The LONG_MIN, LONG_MAX, and ULONG_MAX macros are now provided by the toolchain-specific limits.h. This prevents warnings caused by differences between the compiler's limits.h and CyLib.h.
Support for RealView in --gnu mode	Corrected some preprocessor conditionals that caused errors when RealView was used with the --gnu option.
Update PSoC 5 startup code to make better use of standard libraries	The startup code used in PSoC 5 projects has been updated to use standard libraries to provide the initialization. This provides for a standard initialization.
Replaced cydevice.h with cydevice_trm.h	cydevice.h has been marked obsolete, so the APIs and generated code provided with PSoC Creator should not use it.
CYCODE, CYDATA, CYXDATA and CYFAR defines are copied over to cytypes.h	Now any component can make use of these defines, Ex Flash component
Fix for the cache flash cycles incorrect behavior after sleep.	CYREG_CACHE_CR doesn't retain its value in PSOC 3 after the sleep
Miscellaneous Flash/EEPROM API changes.	Define active/standby power configuration registers based on silicon. Define active/standby power configuration register constant based on silicon. Added CyFlash_Start() and CyFlash_Stop() APIs. Removed CyFlashEEActivePower() and CyFlashEEStandbyPower() APIs, Modified CySetFlashEEBuffer()for PSoC 3 so as not to test whether the buffer is not a pointer at 0. Changed an argument in CyWriteRowConfig() API to rowData. Added CyEEPROM_Start(), CyEEPROM_Stop(), CyEEPROM_ReadReserve(), CyEEPROM_ReadRelease(), and CyFlash_SetWaitCycles(uint8 freq) APIs. Power mode registers and power mode register constants are defined based on the silicon. Cache Control register is defined based on the silicon.
Added reentrancy support.	Appropriate APIs in cylib.c/h, cyflash.c/h, cydmac.c/h and cyspc.c/h files were updated to include reentrant support.
Converted cymemset and cymemcpy to macros, removed cymemmove.	The memset/memcpy/memmove provided by the toolchain vendor are typically faster than the generic implementation because they are designed for the specific target platform. Including both the vendor-provided functions and the generic implementation wastes code space. Non-legacy code should use memset and memcpy directly.

Description of Version 2.0 Changes	Reason for Changes / Impact
Added numerous new clock APIs: CyPLL_OUT_Start_confirm CyMasterClock_SetSource CyMasterClock_ActivateIMOAndSet CyMasterClock_ActivatePLLAndSet CyIMO_ActivateAndWait CyIMO_SetFrequency CyIMO_EnableDisableDoubler CyIMO_EnableDoubler CyIMO_DisableDoubler CyIMO2X_SetSource CyPLL_P_SetValue CyPLL_Q_SetValue CyPLL_SetInput CyXTAL_SetGain CyXTAL_SetVref CyILO_1KHZ_Start CyILO_1KHZ_Stop CyILO_100KHZ_Start CyILO_100KHZ_Stop CyILO_33KHZ_Start CyILO_33KHZ_Stop CyILO_SelectClock CyUSB_Start CyUSB_Stop CyUSB_SetClockSource CyUSB_SetClockSource_IMO2X CyUSB_SetClockSource_IMO CyUSB_SetClockSource_PLL CyUSB_SetClockSource_DSI CyUSB_EnableDivider CyUSB_DisableDivider CyCLK_SetDividerValue CyCLK_SetBusDividerValue	To provide additional functionality.
char8 datatype is defined as char.	To support new compilers in future.
Renamed CySleep and CyHibernate APIs to CyPmSleep and CyPmHibernate; added new CyPmAltAct API.	There were deficiencies in the low power APIs before cy_boot 2.0. You must update your design to use cy_boot 2.0 and rewrite the low power portion of the design to use the new APIs.