PSoC® Creator™

PSoC 4 System Reference Guide

cy_boot Component v5.20

Document Number: 002-00076, Rev. **

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

Contents

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:

- Flash
- Clocking
- Power management
- Startup code
- Various library functions
- Linker scripts

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

Migrating from Previous cy_boot Versions

The cy_boot component version 5.0 and later is fully backward compatible with cy_boot version 4.20 (and previous versions). For PSoC 4 devices, the CyLFClk (low-frequency clock) APIs have been moved into separate files (*CyLFClk.h/CyLFClk.c*).

Firmware projects created using PSoC Creator 3.1 will work with no issues in PSoC Creator 3.2 and later if the *project.h* file is referenced, regardless of the cy_boot update. However, if the *project.h* file is not included in the project being migrated, you must add a reference to the *CyLFClk.h* file in the project for the availability of CyLFClk APIs.

If you choose not to update to cy_boot version 5.0 or later while migrating projects from PSoC Creator 3.1 to PSoC Creator 3.2 and later, *CyLFClk.h/CyLFClk.c* files will not be generated.

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™ PSoC 4 System Reference Guide, cy_boot Component v5.20 **Document Number: 002-00076**		
Revision	**Date**	**Description of Change**
**	8/25/15	New document for version 5.20 of the cy_boot component. Refer to the change section for component changes from previous versions of cy_boot.

2 Standard Types, APIs, and Defines

To support the operation of the same code across multiple CPUs with multiple compilers, the cy_boot component provides types and defines (in the *cytypes.h* file) that create consistent results across platforms.

Base Types

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

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.

Define	Description
__GNUC__	ARM GCC compiler
__ARMCC_VERSION	ARM Realview compiler used by Keil MDK tool sets

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_PSOC4	Any PSoC 4 Device
CY_PSOC4_4000	PSoC 4000 device family.
CY_PSOC4_4100	PSoC 4100 device family.
CY_PSOC4_4200	PSoC 4200 device family.
CY_PSOC4_4100BL	PSoC 4100 device family with BLE support.
CY_PSOC4_4200BL	PSoC 4200 device family with BLE support.
CY_PSOC4_4100M	PSoC 4100M device family.
CY_PSOC4_4200M	PSoC 4200M device family.

Variable Attributes

Define	Description
CY_NOINIT	Specifies that a variable should be placed into uninitialized data section that prevents this variable from being initialized to zero on startup.
CY_ALIGN	Specifies a minimum alignment (in bytes) for variables of the specified type.

CY_PACKED, CY_PACKED_ATTR	Attached to an enum, struct, or union type definition, specified that the minimum required memory be used to represent the type.
	Example:
	`CYPACKED typedef struct {`
	`  uint8 freq;`
	`  uint8 absolute;`
	`} CYPACKED_ATTR imoTrim;`

CY_INLINE	Specifies that compiler can perform inline expansion: insert the function code at the address of each function call.

Instance APIs

General APIs

Most components have an instance-specific set of the APIs that allow you to initialize, enable and disable the component. These functions are listed below generically. Refer to the individual datasheet for specific information.

`=instance_name`_InitVar

Description: This global variable Indicates whether the component has been initialized. The variable is initialized to 0 and set to 1 the first time _Start() is called. This allows the component to restart without reinitialization after the first call to the _Start() routine.

If reinitialization of the component is required, then the _Init() function can be called before the _Start() or _Enable() function.

void `=instance_name`_Start (void)

Description: This function intended to start component operation. The _Start() sets the _initVar variable, calls the _Init function, and then calls the _Enable function.

Parameters: None

Return Value: None

void `=instance_name`_Stop (void)

Description: Disables the component operation.

Parameters: None

Return Value: None

void `=instance_name`_Init (void)

Description: Initializes component's parameters to those set in the customizer placed on the schematic. All registers will be reset to their initial values. This reinitializes the component. Usually called in _Start().

Parameters: None

Return Value: None

void `=instance_name`_Enable (void)

Description: Enables the component block operation.

Parameters: None

Return Value: None

Low Power APIs

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

void `=instance_name`_Sleep (void)

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

- PSoC 4: Call the _Sleep() function before calling the CySysPmDeepSleep() function.

Parameters: None

Return Value: 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

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.

PSoC Creator Generated Defines

PSoC Creator generates the following macros in the *cyfitter.h* file.

Project Type

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

- CYDEV_PROJ_TYPE
- CYDEV_PROJ_TYPE_BOOTLOADER
- CYDEV_PROJ_TYPE_LOADABLE
- CYDEV_PROJ_TYPE_MULTIAPPBOOTLOADER
- CYDEV_PROJ_TYPE_STANDARD
- CYDEV_PROJ_TYPE_LOADABLEANDBOOTLOADER

Chip Configuration Mode

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

All

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

PSoC 4

- CYDEV_CONFIG_READ_ACCELERATOR (Flash read accelerator enabled?)
- CYDEV_USE_BUNDLED_CMSIS (Include the CMSIS standard library.)

Debugging Mode

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

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

Chip Protection Mode

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

- CYDEV_DEBUG_PROTECT
- CYDEV_DEBUG_PROTECT_KILL
- CYDEV_DEBUG_PROTECT_OPEN
- CYDEV_DEBUG_PROTECT_PROTECTED

Stack and Heap

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

- CYDEV_HEAP_SIZE
- CYDEV_STACK_SIZE

Voltage Settings

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

- CYDEV_VARIABLE_VDDA
- CYDEV_VDDA
- CYDEV_VDDA_MV
- CYDEV_VDDD
- CYDEV_VDDD_MV
- CYDEV_VDDIO0
- CYDEV_VDDIO0_MV
- CYDEV_VDDIO1
- CYDEV_VDDIO1_MV
- CYDEV_VDDIO2
- CYDEV_VDDIO2_MV
- CYDEV_VDDIO3
- CYDEV_VDDIO3_MV
- CYDEV_VIO0
- CYDEV_VIO0_MV
- CYDEV_VIO1
- CYDEV_VIO1_MV
- CYDEV_VIO2
- CYDEV_VIO2_MV
- CYDEV_VIO3
- CYDEV_VIO3_MV

System Clock Frequency

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

PSoC 4

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

JTAG/Silicon ID

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

- CYDEV_CHIP_JTAG_ID

IP Block Information

PSoC Creator generates the following macros for the IP blocks that exist on the current device:

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

For example:

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

3 Clocking

PSoC Creator Clocking Implementation

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

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

The System Clock consolidates System Clock (SYSCLK) on PSoC 4 devices. The Master Clock consolidates High-Frequency Clock (HFCLK) on PSoC 4 devices.

Overview

The clock system includes these clock resources:

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

Power Modes

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

The EXTCLK is available in Active and Sleep modes. The system will enter/exit Deep Sleep and Hibernate using external clock. The device will re-enable the IMO if it was enabled before entering Deep Sleep or Hibernate, but it does not wait for the IMO before starting the CPU. After entering Active mode, the IMO may take an additional 2 us to begin toggling. The IMO will startup cleanly without glitches, but any dependency should account for this extra startup time. If desired, firmware may increase wakeup hold-off using CySysPmSetWakeupHoldoff() function to include this 2 us and ensure the IMO is toggling by the time Active mode is reached.

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

Clock Connectivity

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

System Clock

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

Global Clock

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

Routed Clock

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

Clock Synchronization

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

Routed Clock Implementation

The clocking implementation in PSoC directly connects global clock signals to the clock input of clocked digital logic. This applies to both synchronous and asynchronous clocks. Since global clocks are

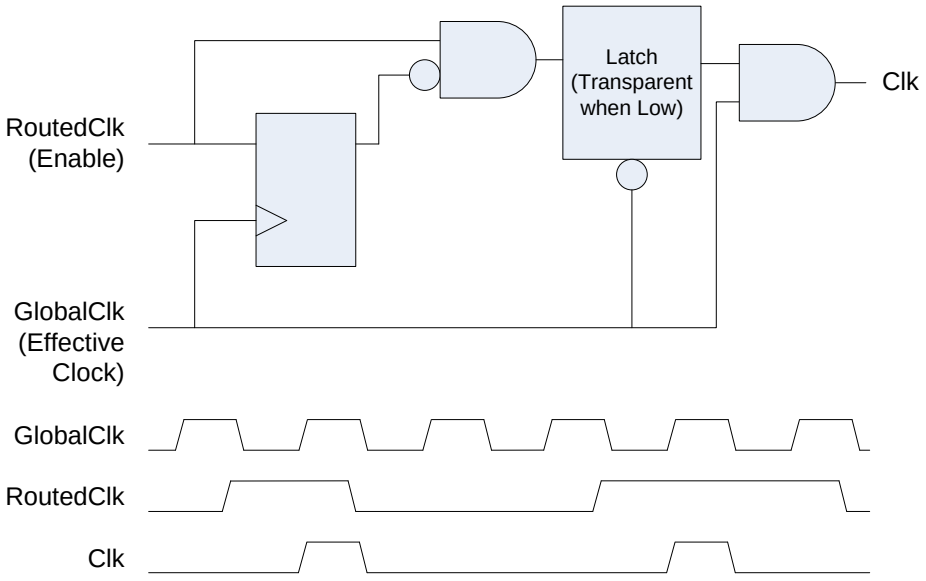
distributed on low skew clock lines, all clocked elements connected to the same global clock will be clocked at the same time.

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

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

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

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



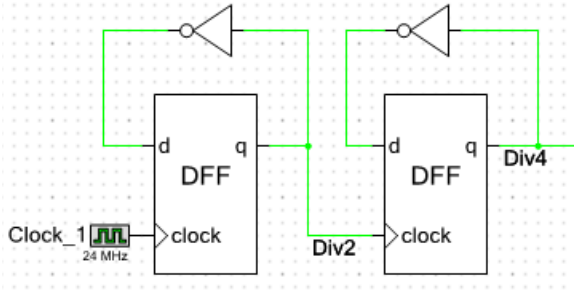
This diagram shows that the resulting clock occurs synchronous to the global clock on the first clock after a rising edge of the routed clock.

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

The clock transformation used for every routed clock is reported in the report file. This file is located in the Workspace Explorer under the **Results** tab after a successful build. The details are shown under the "Initial Mapping" heading. Each routed clock will be shown with the "Effective Clock" and the "Enable Signal". The "Effective Clock" is the global clock that is used and the "Enable Signal" is the routed clock that is edge detected and used as the enable for that clock.

Example with a Divided Clock

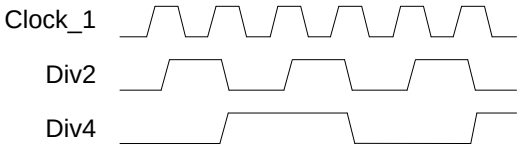
A simple divided clock circuit can be used to observe how this transformation is done. The following circuit clocks the first flip-flop (cydff_1) with a global clock. This generates a clock that is divided by 2 in frequency. That signal is used as a routed clock that clocks the next flip-flop (cydff_2).



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

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

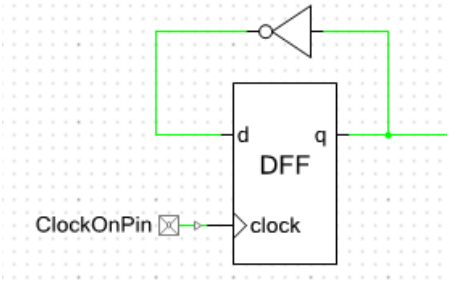
The resulting signals generated by this circuit are as follows.



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

Example with a Clock from a Pin

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



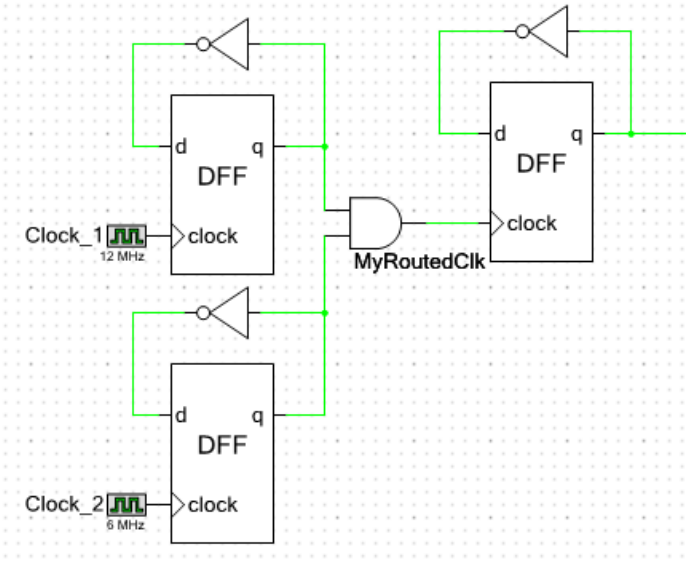
```
<CYPRESSTAG name="Initial Mapping" icon="FILE_RPT_TECHM">
(Global Clock Selection)
<CYPRESSTAG name="UDB Routed Clock Assignment">
    Routed Clock: ClockOnPin(0):iocell.fb
        Effective Clock: BUS_CLK
        Enable Signal: ClockOnPin(0):iocell.fb
</CYPRESSTAG>
```

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

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

Example with Multiple Clock Sources

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

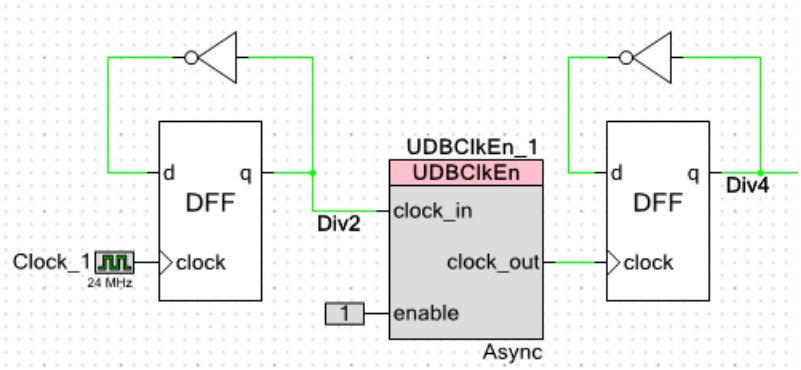



```
<CYPRESSTAG name="Tech mapping">
<CYPRESSTAG name="Initial Mapping" icon="FILE_RPT_TECHM">
<CYPRESSTAG name="Global Clock Selection" icon="FILE_RPT_TECHM">
    Digital Clock 0: Automatic-assigning  clock 'Clock_1'. Fanout=1, Signal=tmp__cydff_1_clk
    Digital Clock 1: Automatic-assigning  clock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

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

Overriding Routed Clock Transformations

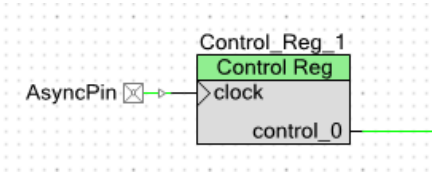
The automatic transformation that PSoC Creator performs on routed clocks is generally the implementation that should be used. There is however a method to force the routed clock to be used directly. The UDBClkEn component configured in Async mode will force the clock used to be the routed clock, as shown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

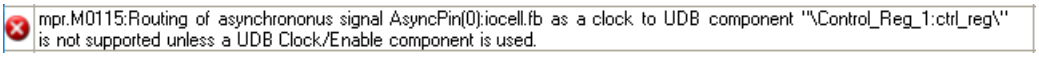


Using Asynchronous Clocks

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

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





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

Clock Crossing

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

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

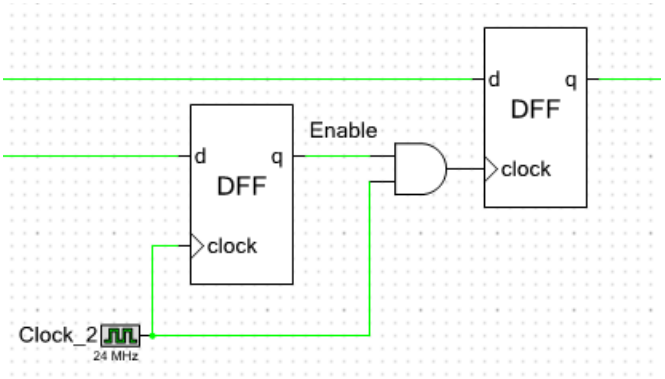
The only guarantee between the clock domains is that their edges will always occur on a rising edge of System Clock. That means that the rising edges of the two clock domains can be as close as a single System Clock cycle apart. This is true even when the clock domains are multiples of each other, since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

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

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

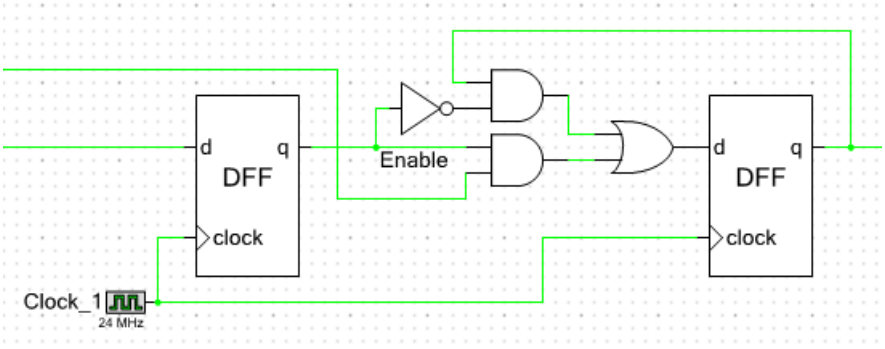
Gated Clocks

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



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

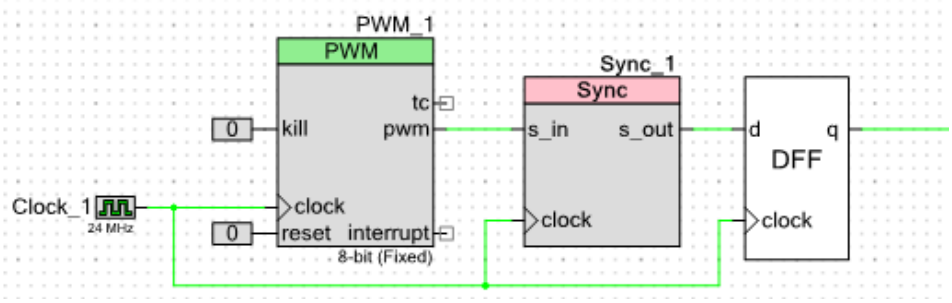
The following circuit implements the equivalent function and is supported by timing analysis, only uses global clocks, and has no reliability issues. This circuit does not gate the clock, but instead logically enables the clocking of new data or maintains the current data.



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

Fixed-Function Clocking

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



UDB-Based Clocking

If the component allows asynchronous clocks, you may use any clock input frequency within the device's frequency range. If the component requires synchronization to the SYSCLK, then when using a routed clock for the component, the frequency of the routed clock cannot exceed one half the routed clock's source clock frequency.

- If the routed clock is synchronous to the SYSCLK, then it is one half the SYSCLK.
- If the routed clock is synchronous to one of the clock dividers, its maximum is one half of that clock rate.

Changing Clocks in Run-time

Impact on Components Operation

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

CyDelay APIs

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

Cache Configuration

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

Low Voltage Analog Boost Clocks

When the operating voltage (Vdda) of a PSoC device drops below 4.0 V, the analog pumps for the analog routing switches must be enabled by calling the SetAnalogRoutingPumps() function with the corresponding parameter. On PSoC 4 devices the pumps may be left on at all voltages, but it is recommended to disable them above 4.0 V so as to reduce current draw. It is the user's responsibility to monitor the Vdda level at run-time and enable/disable the pumps as appropriate.

The analog pumps for the analog routing switches are configured on device startup based on the **Vdda** and **Variable Vdda** design-time options. The **Variable Vdda** option in the **System** tab of the PSoC Creator Design-Wide Resources (DWR) file is added to allow for designs in which the value of **Vdda** is expected to vary at runtime. If **Variable Vdda** is enabled, the SetAnalogRoutingPumps() function described above will be generated. If **Vdda** < 4.0 V, the routing pumps will be automatically enabled on reset.

On PSoC 4 devices, the IMO must be enabled if **Variable Vdda** is enabled or **Vdda** < 4.0 V. This is because the clock for the analog switch pump is driven from the IMO.

APIs

There is one API used for all devices: the SetAnalogRoutingPumps() function. Then, there is a set of APIs used for PSoC 4 devices. Functions starting with CySysClk are applicable to PSoC 4 only.

void SetAnalogRoutingPumps(uint8 enabled)

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

Parameters: enabled:

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

Return Value: None

void CySysClkImoStart(void)

Description: Enables the IMO.

For PSoC 4100M / PSoC 4200M devices, this function will also enable the WCO lock if "Trim with WCO" is selected on the Configure System Clocks dialog.

Parameters: None

Return Value: None

Side Effects and Restrictions: None

void CySysClkImoStop(void)

Description: Disables the IMO.

For PSoC 4100M/PSoC 4200M devices, this function will also disable the WCO lock if "Trim with WCO" is selected on the Configure System Clocks dialog.

Parameters: None

Return Value: None

Side Effects and Restrictions: None

void CySysClkIloStart(void)

Description: Starts the ILO. Refer to the device datasheet for the ILO startup time.

Parameters: None

Return Value: None

Side Effects and Restrictions: None

void CySysClkWriteHfclkDirect (uint32 clkSelect)

Description: Selects the direct source for the HFCLK.

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

Define	Source
CY_SYS_CLK_HFCLK_IMO	IMO
CY_SYS_CLK_HFCLK_EXTCLK	External clock pin
CY_SYS_CLK_HFCLK_ECO	External crystal oscillator (applicable only for PSoC 4100 BLE and PSoC 4200 BLE).

Return Value: None

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

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

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

void CySysClkWriteSysclkDiv (uint32 divider)

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

Parameters: divider: Power of 2 prescaler selection.

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

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

Return Value: None

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

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

void CySysClkWriteImoFreq (uint32 freq)

Description: Sets the frequency of the IMO.
If IMO is currently driving the HFCLK, and if the HFCLK frequency decreases, you can call CySysFlashSetWaitCycles () to improve CPU performance. See CySysFlashSetWaitCycles () for more information.

For PSoC 4000 family of devices, maximum HFCLK frequency is 16 MHz. If IMO is configured to frequencies above 16 MHz, ensure to set the appropriate HFCLK predivider value first.

For PSoC 4100M/PSoC 4200M devices, if "Trim with WCO" is selected on the Configure System Clocks dialog, then this API will disable the WCO lock, write the new IMO frequency, and then re-enable the WCO lock.

Parameters: All PSoC 4 families excluding PSoC 4000: Valid range [3-48] with step size equals 1.
PSoC 4000: Valid range [24-48] with step size equals 4.

Note The CPU is halted if new frequency is invalid and project is compiled in debug mode.

Return Value: None

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

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

void CySysClkImoEnableWcoLock(void)

Description: Enables the IMO to WCO lock feature to increase IMO clock accuracy by locking the IMO to the WCO. This function works only if the WCO is already enabled. If the WCO is not enabled then this function returns without enabling the lock feature.

This function is applicable to PSoC 4100M/PSoC 4200M devices only.

Parameters: None

Return Value: None

void CySysClkImoDisableWcoLock(void)

Description: Disables the IMO to WCO lock feature.
This function is applicable for PSoC 4100M/PSoC 4200M devices only.

Parameters: None

Return Value: None

External Crystal Oscillator (ECO) APIs

cystatus CySysClkEcoStart(uint32 timeoutUs)

Description: Starts the External Crystal Oscillator (ECO). Refer to the device datasheet for the ECO startup time.
The timeout interval is measured based on the system frequency defined by PSoC Creator at build time. If System clock frequency is changed in runtime, the CyDelayFreq() with the appropriate parameter should be called.

Parameters: timeoutUs: Timeout in microseconds. If zero is specified, the function starts the crystal and returns CYRET_SUCCESS. If non-zero value is passed, the CYRET_SUCCESS is returned once crystal is oscillating and amplitude reached 60% and it does not mean 24 MHz crystal is within 50 ppm. If it is not oscillating or amplitude didn't reach 60% after specified amount of time, the CYRET_TIMEOUT is returned.

Return Value: CYRET_SUCCESS - Completed successfully. The ECO is oscillating and amplitude reached 60% and it does not mean 24 MHz crystal is within 50 ppm.
CYRET_TIMEOUT - Timeout occurred

void CySysClkEcoStop(void)

Description: Stops the megahertz crystal.

Parameters: None

Return Value: None

uint32 CySysClkEcoReadStatus(void)

Description: Read status bit for the megahertz crystal.

Parameters: None

Return Value: Non-zero indicates that ECO output reached 50 ppm.

void CySysClkWriteEcoDiv(uint32 divider)

Description: Selects value for the ECO divider.

The ECO must not be the HFCLK clock source when this function is called. The HFCLK source can be changed to the other clock source by call to the CySysClkWriteHfclkDirect() function. If the ECO sources the HFCLK this function will not have any effect if compiler in release mode, and halt the CPU when compiler in debug mode.

Parameters: divider: Power of 2 divider selection.

Define	Divider
CY_SYS_CLK_ECO_DIV1	HFCLK = ECO / 1
CY_SYS_CLK_ECO_DIV2	HFCLK = ECO / 2
CY_SYS_CLK_ECO_DIV4	HFCLK = ECO / 4
CY_SYS_CLK_ECO_DIV8	HFCLK = ECO / 8

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

4 Power Management

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

Mode	PSoC 4000	PSoC 4100 / PSoC 4200 PSoC 4100 BLE / PSoC 4200 BLE
Active	✓	✓
Sleep	✓	✓
Deep Sleep	✓	✓
Hibernate		✓
Stop		✓

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

For the ARM-based devices (PSoC 4), 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.

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

The Power Management API is provided in the *CyPm.c and CyPm.h* files.

Implementation

For **PSoC 4100 and PSoC 4200** devices, the software should set EXT_VCCD bit in the PWR_CONTROL register when Vccd is shorted to Vddd on the board. This impacts the chip internal state transitions where it is necessary to know whether Vccd is connected or floating to achieve minimum current in low power modes. **Note** Setting this bit turns off the active regulator and will lead to a system reset unless both Vddd and Vccd pins are supplied externally. Refer to the device TRM for more information.

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

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

Clock Configuration (PSoC 4100 BLE / PSoC 4200 BLE)

For **PSoC 4100 BLE and PSoC 4200 BLE** devices, the HFCLK source should be set to IMO before switching the device into low power mode. The IMO should be enabled (by calling CySysClkImoStart(), if it is not) and HFCLK source should be changed to IMO by calling CySysClkWriteHfclkDirect(CY_SYS_CLK_HFCLK_IMO).

If the System clock frequency is increased by switching to the IMO, the CySysFlashSetWaitCycles() function with an appropriate parameter should be called beforehand. Also, it can optionally be called after lowering the System clock frequency in order to improve CPU performance. See CySysFlashSetWaitCycles() description for the details.

Power Management APIs

void CySysPmSleep(void)

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

Parameters: None

Return Value: None

Side Effects and Restrictions: None

void CySysPmDeepSleep(void)

Description: Puts the part into the Deep Sleep state.

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

Parameters: None

Return Value: None

Side Effects and Restrictions: None

void CySysPmHibernate(void)

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

Parameters: None

Return Value: None

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

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

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

void CySysPmStop(void)

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

Wakeup from Stop is performed by toggling the wakeup pin (PSoC 4100 / PSoC 4200 / PSoC 4100M / PSoC 4200M – P0.7, PSoC 4100 BLE / PSoC 4200 BLE – P2.2), causing a normal boot procedure to occur.

- To configure the wakeup pin, the Digital Input Pin component should be placed on the schematic, assigned to the appropriate wakeup pin, and resistively pulled up or down to the inverse state of the wakeup polarity.
- To distinguish the wakeup from the Stop mode and the general Reset event, CySysPmGetResetReason() function could be used. The wakeup pin is active low by default. The wakeup pin polarity could be changed with the CySysPmSetWakeupPolarity() function.

Parameters: None

Return Value: None

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

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

void CySysPmSetWakeupPolarity(uint32 polarity)

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

Parameters: polarity: Wakeup pin active level

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

Return Value: None

Side Effects and Restrictions: None

uint32 CySysPmGetResetReason(void)

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

Parameters: None

Return Value: Reset reason

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

Side Effects and Restrictions: None

void CySysPmFreezeIo(void)

Description: Freezes IO-Cells directly to save IO-Cell state on wake up from Hibernate or Stop mode. It is not required to call this function before entering Stop mode, since the CySysPmStop() function freezes IO-Cells implicitly.

This API is not available for PSoC 4000 family of devices.

Parameters: None

Return Value: None

void CySysPmUnfreezeIo(void)

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

If the firmware intent is to retain the data value on the port, then the value must be read and re-written to the data register before calling this API. Furthermore, the drive mode must be re-programmed. If this is not done, the pin state will change to default state the moment the freeze is removed.

This API is not available for PSoC 4000 family of devices.

Parameters: None

Return Value: None

void CySysPmSetWakeupHoldoff(uint32 hfclkFrequencyMhz)

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

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

This function is applicable only for the PSoC 4000 family.

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

Return Value: None

5 Interrupts

The APIs in this chapter apply to all architectures except as noted. The Interrupts API is provided in the *CyLib.c and CyLib.h* files. Refer also to the Interrupt component datasheet for more information about interrupts.

APIs

CyGlobalIntEnable

Description: Macro statement that allows interrupts execution by clearing the PRIMASK register. Refer to the ARM Cortex-M0 documentation for more details.

CyGlobalIntDisable

Description: Macro statement that prevents interrupts execution by setting the PRIMASK register. Refer to the ARM Cortex-M0 documentation for more details.

uint32 CyDisableInts()

Description: Disables all interrupts.

Parameters: None

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

void CyEnableInts(uint32 mask)

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

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

Return Value: None

void CyIntEnable(uint8 number)

Description: Enables the specified interrupt number.

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

Return Value: None

void CyIntDisable(uint8 number)

Description: Disables the specified interrupt number.

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

Return Value: None

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. 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].

Define	Exception Number
CY_INT_NMI_IRQN	Non Maskable Interrupt.
CY_INT_HARD_FAULT_IRQN	Hard Fault Interrupt.
CY_INT_MEM_MANAGE_IRQN	Memory Management Interrupt. Not available for PSoC 4.
CY_INT_BUS_FAULT_IRQN	Bus Fault Interrupt. Not available for PSoC 4.
CY_INT_USAGE_FAULT_IRQN	Usage Fault Interrupt, Not available for PSoC 4.
CY_INT_SVCALL_IRQN	SV Call Interrupt.
CY_INT_DEBUG_MONITOR_IRQN	Debug Monitor Interrupt. Not available for PSoC 4.
CY_INT_PEND_SV_IRQN	Pend SV Interrupt.
CY_INT_SYSTICK_IRQN	System Tick Interrupt.

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. 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: [0-3].

Return Value: None

uint8 CyIntGetPriority(uint8 number)

Description: Gets the priority of the specified interrupt number.

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

Return Value: Interrupt priority

void CyIntSetPending(uint8 number)

Description: Forces the specified interrupt number to be pending.

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

Return Value: None

void CyIntClearPending(uint8 number)

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

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

Return Value: None

6 Pins

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

```
CYREG_PRTx_DR
CYREG_PRTx_PS
CYREG_PRTx_PC
```

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

PSoC 4 APIs

CY_SYS_PINS_READ_PIN(portPS, pin)

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

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

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

CY_SYS_PINS_SET_PIN(portDR, pin)

Description: Set the output value for the pin (data register, DR) to a logic high.
Note that this only has an effect for pins configured as software pins that are not driven by hardware.
The macro operation is not atomic. It is not guaranteed that the shared register will remain uncorrupted during simultaneous read/modify/write operations performed by two threads (main and interrupt threads). To guarantee data integrity in such cases, the macro should be invoked while the specific interrupt is disabled or within the critical section (all interrupts are disabled).

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

Return Value: None

CY_SYS_PINS_CLEAR_PIN(portDR, pin)

Description: This macro sets the state of the specified pin to zero.
The macro operation is not atomic. It is not guaranteed that the shared register will remain uncorrupted during simultaneous read/modify/write operations performed by two threads (main and interrupt threads). To guarantee data integrity in such cases, the macro should be invoked while the specific interrupt is disabled or within the critical section (all interrupts are disabled).

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

Return Value: None

CY_SYS_PINS_SET_DRIVE_MODE(portPC, pin, mode)

Description: Sets the drive mode for the pin (DM).
The macro operation is not atomic. It is not guaranteed that the shared register will remain uncorrupted during simultaneous read/modify/write operations performed by two threads (main and interrupt threads). To guarantee data integrity in such cases, the macro should be invoked while the specific interrupt is disabled or within the critical section (all interrupts are disabled).

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

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

Return Value: None

CY_SYS_PINS_READ_DRIVE_MODE(portPC, pin)

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

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

Return Value: Current drive mode for the pin

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

7 Register Access

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

The PSoC 4 processor architecture use little endian ordering.

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

The PSoC 4 requires these accesses to be aligned to the width of the transaction.

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

APIs

uint8 CY_GET_REG8(uint32 reg)

Description: Reads the 8-bit value from the specified register.

Parameters: reg: Register address (

Return Value: Read value

void CY_SET_REG8(uint32 reg, uint8 value)

Description: Writes the 8-bit value to the specified register.

Parameters: reg: Register address

value: Value to write

Return Value: None

uint16 CY_GET_REG16(uint32 reg)

Description: Reads the 16-bit value from the specified register. This macro implements the byte swapping required for proper operation.

Parameters: reg: Register address

Return Value: Read value

void CY_SET_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.

Parameters: reg: Register address

value: Value to write

Return Value: None

uint32 CY_GET_REG24(uint32 reg)

Description: Reads the 24-bit value from the specified register. This macro implements the byte swapping required for proper operation.

Parameters: reg: Register address

Return Value: Read value

void CY_SET_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.

Parameters: reg: Register address

value: Value to write

Return Value: None

uint32 CY_GET_REG32(uint32 reg)

Description: Reads the 32-bit value from the specified register. This macro implements the byte swapping required for proper operation.

Parameters: reg: Register address

Return Value: Read value

void CY_SET_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.

Parameters: reg: Register address

value: Value to write

Return Value: None

uint8 CY_GET_XTND_REG8(uint32 reg)

Description: Reads the 8-bit value from the specified register. Identical to CY_GET_REG8 for PSoC 4.

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. Identical to CY_SET_REG8 for PSoC 4.

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. Identical to CY_GET_REG16 for PSoC 4.

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. Identical to CY_SET_REG16 for PSoC 4.

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. Identical to CY_GET_REG24 for PSoC 4.

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. Identical to CY_SET_REG24 for PSoC 4.

Parameters: reg: Register address

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. Identical to CY_GET_REG32 for PSoC 4.

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. Identical to CY_SET_REG32 for PSoC 4.

Parameters: reg: Register address

value: Value to write

Return Value: None

Bit Field Manipulation

The following macros shall provide bit field manipulation functionality.

Macro	Description
CY_GET_REG8_FIELD	Reads the specified bit field value from the specified 8-bit register.
CY_SET_REG8_FIELD	Sets the specified bit field value of the specified 8-bit register to the required value.
CY_CLEAR_REG8_FIELD	Clears the specified bit field of the specified 8-bit register.
CY_GET_REG16_FIELD	Reads the specified bit field value from the specified 16-bit register.
CY_SET_REG16_FIELD	Sets the specified bit field value of the specified 16-bit register to the required value.
CY_CLEAR_REG16_FIELD	Clears the specified bit field of the specified 16-bit register.
CY_GET_REG32_FIELD	Reads the specified bit field value from the specified 32-bit register.
CY_SET_REG32_FIELD	Sets the specified bit field value of the specified 32-bit register to the required value.
CY_CLEAR_REG32_FIELD	Clears the specified bit field of the specified 32-bit register.
CY_GET_FIELD	Reads the specified bit field value from the given 32-bit value.
CY_SET_FIELD	Sets the specified bit field value within a given 32-bit value.

CY_GET_REG8_FIELD(registerName, bitFieldName)

Description: Reads the specified bit field value from the specified 8-bit register.
The macro operation is not atomic. It is not guaranteed that shared register will remain uncorrupted during simultaneous read-modify-write operations performed by two threads (main and interrupt threads). To guarantee data integrity in such cases, the macro should be invoked while the specific interrupt is disabled or within critical section (all interrupts are disabled).
Using this macro on registers of 32-bit and 16-bit width will generate a hard fault exception. Examples of 8-bit registers are the UDB registers.

Parameters: registerName: fully qualified name of the PSoC 4 device register

bitFieldName: fully qualified name of the bit field. The biFieldName is automatically appended with __OFFSET and __SIZE by the macro for usage.
For fully qualified names of register and bit field, please refer to the respective PSoC family register TRM.

Return Value: Zero if specified bit field equals zero, and non-zero value, otherwise. The return value is of type uint32.

CY_SET_REG8_FIELD(registerName, bitFieldName, value)

Description: Sets the specified bit field value of the specified 8-bit register to the required value.
The macro operation is not atomic. It is not guaranteed that shared register will remain uncorrupted during simultaneous read-modify-write operations performed by two threads (main and interrupt threads). To guarantee data integrity in such cases, the macro should be invoked while the specific interrupt is disabled or within critical section (all interrupts are disabled).
Using this macro on registers of 32-bit and 16-bit width will generate a hard fault exception. Examples of 8-bit registers are the UDB registers.

Parameters: registerName: fully qualified name of the PSoC 4 device register
bitFieldName: fully qualified name of the bit field. The biFieldName is automatically appended with __OFFSET and __SIZE by the macro for usage.
value: value that the field must be configured for
For fully qualified names of register and bit field and the possible values the field can take, please refer to the respective PSoC family register TRM.

Return Value: None

CY_CLEAR_REG8_FIELD(registerName, bitFieldName)

Description: Clears the specified bit field of the specified 8-bit register.

The macro operation is not atomic. It is not guaranteed that shared register will remain uncorrupted during simultaneous read-modify-write operations performed by two threads (main and interrupt threads). To guarantee data integrity in such cases, the macro should be invoked while the specific interrupt is disabled or within critical section (all interrupts are disabled). Using this macro on registers of 32-bit and 16-bit width will generate a hard fault exception. Examples of 8-bit registers are the UDB registers.

Parameters: registerName: fully qualified name of the PSoC 4 device register
bitFieldName: fully qualified name of the bit field. The biFieldName is automatically appended with __OFFSET and __SIZE by the macro for usage.
For fully qualified names of register and bit field and the possible values the field can take, please refer to the respective PSoC family register TRM.

Return Value: None

CY_GET_REG16_FIELD(registerName, bitFieldName)

Description: Reads the specified bit field value from the specified 16-bit register.
The macro operation is not atomic. It is not guaranteed that shared register will remain uncorrupted during simultaneous read-modify-write operations performed by two threads (main and interrupt threads). To guarantee data integrity in such cases, the macro should be invoked while the specific interrupt is disabled or within critical section (all interrupts are disabled). Using this macro on registers of 32-bit and 8-bit width will generate a hard fault exception. Examples of 16-bit registers are the UDB registers.

Parameters: registerName: fully qualified name of the PSoC 4 device register
bitFieldName: fully qualified name of the bit field. The biFieldName is automatically appended with __OFFSET and __SIZE by the macro for usage.
For fully qualified names of register and bit field, please refer to the respective PSoC family register TRM.

Return Value: Zero if specified bit field equals zero, and non-zero value, otherwise. The return value is of type uint32.

CY_SET_REG16_FIELD(registerName, bitFieldName, value)

Description: Sets the specified bit field value of the specified 16-bit register to the required value.
The macro operation is not atomic. It is not guaranteed that shared register will remain uncorrupted during simultaneous read-modify-write operations performed by two threads (main and interrupt threads). To guarantee data integrity in such cases, the macro should be invoked while the specific interrupt is disabled or within critical section (all interrupts are disabled). Using this macro on registers of 32-bit and 8-bit width will generate a hard fault exception. Examples of 16-bit registers are the UDB registers.

Parameters: registerName: fully qualified name of the PSoC 4 device register
bitFieldName: fully qualified name of the bit field. The biFieldName is automatically appended with __OFFSET and __SIZE by the macro for usage.
value: value that the field must be configured for
For fully qualified names of register and bit field and the possible values the field can take, please refer to the respective PSoC family register TRM.

Return Value: None

CY_CLEAR_REG16_FIELD(registerName, bitFieldName)

Description: Clears the specified bit field of the specified 16-bit register.

The macro operation is not atomic. It is not guaranteed that shared register will remain uncorrupted during simultaneous read-modify-write operations performed by two threads (main and interrupt threads). To guarantee data integrity in such cases, the macro should be invoked while the specific interrupt is disabled or within critical section (all interrupts are disabled). Using this macro on registers of 32-bit and 8-bit width will generate a hard fault exception. Examples of 16-bit registers are the UDB registers.

Parameters: registerName: fully qualified name of the PSoC 4 device register
bitFieldName: fully qualified name of the bit field. The biFieldName is automatically appended with __OFFSET and __SIZE by the macro for usage.
For fully qualified names of register and bit field and the possible values the field can take, please refer to the respective PSoC family register TRM.

Return Value: None

CY_GET_REG32_FIELD(registerName, bitFieldName)

Description: Reads the specified bit field value from the specified 32-bit register.
The macro operation is not atomic. It is not guaranteed that shared register will remain uncorrupted during simultaneous read-modify-write operations performed by two threads (main and interrupt threads). To guarantee data integrity in such cases, the macro should be invoked while the specific interrupt is disabled or within critical section (all interrupts are disabled).
Using this macro on registers of 16-bit and 8-bit width will generate a hard fault exception.

Parameters: registerName: fully qualified name of the PSoC 4 device register
bitFieldName: fully qualified name of the bit field. The biFieldName is automatically appended with __OFFSET and __SIZE by the macro for usage.
For fully qualified names of register and bit field, please refer to the respective PSoC family register TRM.

Return Value: Zero if specified bit field equals zero, and non-zero value, otherwise. The return value is of type uint32.

CY_SET_REG32_FIELD(registerName, bitFieldName, value)

Description: Sets the specified bit field value of the specified 32-bit register to the required value.

The macro operation is not atomic. It is not guaranteed that shared register will remain uncorrupted during simultaneous read-modify-write operations performed by two threads (main and interrupt threads). To guarantee data integrity in such cases, the macro should be invoked while the specific interrupt is disabled or within critical section (all interrupts are disabled).
Using this macro on registers of 16-bit and 8-bit width will generate a hard fault exception.

Parameters: registerName: fully qualified name of the PSoC 4 device register
bitFieldName: fully qualified name of the bit field. The biFieldName is automatically appended with __OFFSET and __SIZE by the macro for usage.
value: value that the field must be configured for
For fully qualified names of register and bit field and the possible values the field can take, please refer to the respective PSoC family register TRM.

Return Value: None

CY_CLEAR_REG32_FIELD(registerName, bitFieldName)

Description: Clears the specified bit field of the specified 32-bit register.

The macro operation is not atomic. It is not guaranteed that shared register will remain uncorrupted during simultaneous read-modify-write operations performed by two threads (main and interrupt threads). To guarantee data integrity in such cases, the macro should be invoked while the specific interrupt is disabled or within critical section (all interrupts are disabled). Using this macro on registers of 16-bit and 8-bit width will generate a hard fault exception.

Parameters: registerName: fully qualified name of the PSoC 4 device register
bitFieldName: fully qualified name of the bit field. The biFieldName is automatically appended with __OFFSET and __SIZE by the macro for usage.
For fully qualified names of register and bit field and the possible values the field can take, please refer to the respective PSoC family register TRM.

Return Value: None

CY_GET_FIELD(regValue, bitFieldName)

Description: Reads the specified bit field value from the given 32-bit value.
The macro operation is not atomic. It is not guaranteed that shared register will remain uncorrupted during simultaneous read-modify-write operations performed by two threads (main and interrupt threads). To guarantee data integrity in such cases, the macro should be invoked while the specific interrupt is disabled or within critical section (all interrupts are disabled). This macro has to be used in conjunction with CY_GET_REG32 for atomic reads.

Parameters: regValue: value as read by CY_GET_REG32
bitFieldName: fully qualified name of the bit field. The biFieldName is automatically appended with __OFFSET and __SIZE by the macro for usage.
For fully qualified names of bit field and the possible values the field can take, please refer to the respective PSoC family register TRM.

Return Value: Zero if specified bit field equals zero, and non-zero value, otherwise. The return value is of type uint32.

CY_SET_FIELD(regValue, bitFieldName, value)

Description: Sets the specified bit field value within a given 32-bit value.
The macro operation is not atomic. It is not guaranteed that shared register will remain uncorrupted during simultaneous read-modify-write operations performed by two threads (main and interrupt threads). To guarantee data integrity in such cases, the macro should be invoked while the specific interrupt is disabled or within critical section (all interrupts are disabled).
This macro has to be used in conjunction with CY_GET_REG32 for atomic reads and CY_SET_REG32 for atomic writes.

Parameters: regValue: value as read by CY_GET_REG32
bitFieldName: fully qualified name of the bit field. The biFieldName is automatically appended with __OFFSET and __SIZE by the macro for usage.
value: value that the field must be configured for
For fully qualified names of bit field and the possible values the field can take, please refer to the respective PSoC family register TRM.

Return Value: None

8 Flash

Memory Architecture

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

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

The Flash and 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_NUMBER_ROWS	The number of Flash row.
CY_FLASH_NUMBER_ARRAYS	The number of Flash arrays.

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

- Unprotected
- Full Protection

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

Working with Flash

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

It can take as many as 20 milliseconds to write to flash. During this time, the device should not be reset, or unexpected changes may be made to portions of the flash. Reset sources include XRES pin, software

reset, and watchdog. Make sure that these are not inadvertently activated. Also, the low voltage detect circuits should be configured to generate an interrupt instead of a reset.

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

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

APIs

uint32 CySysFlashWriteRow(uint32 rowNum, const uint8 rowData[])

Description: Erases a row of Flash and programs it with the new data

Parameters: uint32 rowNum: The flash row number. The number of the flash rows is defined by the CY_FLASH_NUMBER_ROWS macro for the selected device. Refer to the device datasheet for the details.

uint8* rowData: Array of bytes to write. The size of the array must be equal to the flash row size. The flash row size for the selected device is defined by the CY_FLASH_SIZEOF_ROW macro. Refer to the device datasheet for the details.

Return Value: Status:

Value	Description
CY_SYS_FLASH_SUCCESS	Successful
CY_SYS_FLASH_INVALID_ADDR	Specified flash row address is invalid
CY_SYS_FLASH_PROTECTED	Specified flash row is protected
Other non-zero	Failure

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

For PSoC 4000, PSoC 4100 BLE and PSoC 4200 BLE devices (PSoC 4100 BLE and PSoC 4200 BLE devices with 256K of Flash memory are not affected), this API will automatically modify the clock settings for the device. Writing to flash requires that changes be made to the IMO and HFCLK settings. The configuration is restored before returning. This will impact the operation of most of the hardware in the device.

For PSoC 4000 devices, the HFCLK will have several frequency changes during the operation of this API between a minimum frequency of the current IMO frequency divided by 4 and a maximum frequency of 12 MHz.
For PSoC 4100 BLE and PSoC 4200 BLE, the IMO frequency is set to 48 MHz.

void CySysFlashSetWaitCycles(uint32 freq)

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

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

Return Value: None

Side Effects and Restrictions: None

9 System Functions

These functions apply to all architectures.

General APIs

uint8 CyEnterCriticalSection(void)

Description: The function prevents interrupts being executed by setting PRIMASK register and returns previous state to be used for critical section exit using CyExitCriticalSection() function. Please refer to the ARM Cortex-M0 documentation for the more details.

Note To avoid corrupting the processor state, it must be the policy that all interrupt routines restore the interrupt enable bits as they were found on entry.

Parameters: None

Return Value: Returns 0 if interrupts were previously enabled or 1 if interrupts were previously disabled.

void CyExitCriticalSection(uint8 savedIntrStatus)

Description: The function restores the interrupt state as it was before CyEnterCriticalSection() function call. If interrupts were allowed before CyEnterCriticalSection() function call, the CyExitCriticalSection() clears the PRIMASK register. Please refer to the ARM Cortex-M0 documentation for the more details.

If an interrupt was already in the pending state, the processor accepts the interrupt after CyExitCriticalSection() function was executed. However, processor can execute up to one additional instruction before entering the interrupt service routine.

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

void CyGetUniqueID(uint32* uniqueId)

Description: Returns the 64-bit unique id of the device

Parameters: uniqueId: Pointer to a two element 32-bit unsigned integer array.

Return Value: Returns the 64-bit unique id of the device by loading them into the integer array pointed to by uniqueId.

CyDelay APIs

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

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 SYSCLK 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. If the SYSCLK 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 SYSCLK 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: SYSCLK 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.

The execution overhead is in range of 16-23 cycles depending on the number of the delay cycles and device family. The 16-cycle overhead means that CyDelayCycles(100), will be executed for 116 cycles.

Parameters: cycles: Number of cycles to delay. Valid range is from 0 to the maximum uint32 type value.

Return Value: None

Voltage Detect APIs (PSoC 4100 / PSoC 4200 / PSoC 4100 BLE / PSoC 4200 BLE)

void CySysLvdEnable(uint32 threshold)

Description: Sets the voltage trip level, enables the output of the digital low-voltage monitor, and unmasks the associated interrupt in the LVD block.

Note The associated global interrupt enable/disable state is not changed by the function. The Interrupt component's API should be used to register the interrupt service routine and to enable/disable associated interrupt.

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

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

Return Value: None

void CySysLvdDisable(void)

Description: Disables the low voltage detection. Low voltage interrupt is masked in LVD block.

Note The associated global interrupt enable/disable state is not changed by the function. The Interrupt component's API should be used to enable/disable associated interrupt.

Parameters: None

Return Value: None

uint32 CySysLvdGetInterruptSource(void)

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

Parameters: None

Return Value: Interrupt request value:

- CY_SYS_LVD_INT - Indicates an Low Voltage Detect interrupt

void CySysLvdClearInterrupt(void)

Description: Clears the low voltage detection interrupt status.

Parameters: None

Return Value: None

Macro Callbacks

Macro callbacks allow users to execute code from the API files that are automatically generated by PSoC Creator. Refer to the PSoC Creator Help and *Component Author Guide* for the more details.

In order to add code to the macro callback present in the component's generated source files, perform the following:

- Define a macro to signal the presence of a callback (in *cyapicallbacks.h*). This will "uncomment" the function call from the component's source code.
- Write the function declaration (in *cyapicallbacks.h*). This will make this function visible by all the project files.
- Write the function implementation (in any user file).

Macro Callback [1]	Associated Macro	Description
CyBoot_IntDefaultHandler_Exception_EntryCallback	CY_BOOT_INT_DEFAULT_HANDLER_EXCEPTION_ENTRY_CALLBACK	Used at the beginning of the IntDefaultHandler() interrupt handler to perform additional application-specific actions in unhandled exceptions on PSoC 4 devices.

1 The macro callback name is formed by component function name optionally appended by short explanation and "Callback" suffix.

10 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
- Configure the device
- Initialize static and global variables with initialization values
- Clear all remaining static and global variables
- Integrate with the bootloader functionality
- Preserve the reset status
- Call main() C entry point

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.

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.

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

GCC Implementation

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

The newlib-nano is configured by default. To choose newlib library, open the Build Settings dialog > ARM GCC 4.8.4 > Linker > General, and set the "Use newlib-nano" option to False.

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

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

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

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

Realview Implementation (applicable for MDK)

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

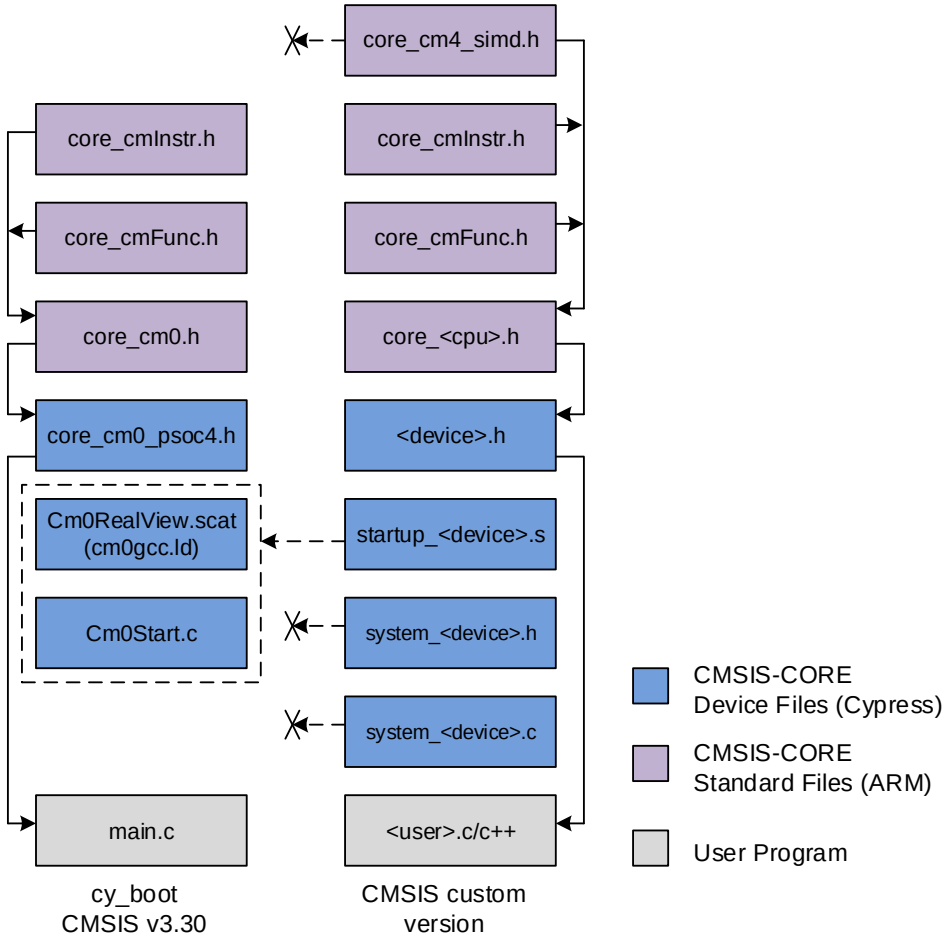
- Support for RTOS and user replacement of routines. This is possible because the library routines are denoted as "weak" allowing their replacement if another implementation is provided.
- A mechanism is provided that allows for the replacement of the provided linker/scatter file with a user version.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

CMSIS Support

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

PSoC Creator 3.2 provides support for CMSIS Core version 4.0. Also, PSoC Creator 3.2 provides the ability to use a custom version of the CMSIS Core.

The following diagram shows how CMSIS Core version 4.0 files are integrated into the cy_boot component and how custom version of CMSIS Core files can be integrated.



The following describe each file from the diagram:

- The *Cm0Start.c* and *cm0gcc.ld* files (part of the cy_boot component) contain Cortex-M0 device startup code and interrupt vector tables and completely substitute CMSIS startup_<device>.s template file.
- Vendor-specific device file <device>.h that includes CMSIS Core standard files is represented in cy_boot component by *core_cm0_psoc4.h*.

- The *core_cmInstr.h* file defines intrinsic functions to access special Cortex-M instructions and *core_cmFunc.h* file provides functions to access the Cortex-M core peripherals. These files were added since CMSIS Core version 2.0.
- The *core_cm4_simd.h* file added to the CMSIS SIMD Instruction Access is relevant for Cortex-M4 only.
- *system_<device>.h, system_<device>.c* – Generic files for system configuration (i.e. processor clock and memory bus system), are partially covered by *Cm0Start.c*.

Manual addition of the CMSIS Core files

Beginning with PSoC Creator 2.2, the “Include CMSIS Core Peripheral Library Files” option is added to the System tab of the DWR file. By default, this option is enabled and CMSIS Core version 4.0 files are added to the project. This option should be disabled if you wish to manually add CMSIS Core files.

Un-check “Include CMSIS Core Peripheral Library Files” option on the System tab of the DWR file to detach CMSIS 4.0 files from the cy_boot component.

Add the following CMSIS Core files to the project:

- core_cmInstr.h
- core_cmFunc.h
- core_cm0.h

Based on the CMSIS vendor-specific template file (<device>.h), create device header file, copy device specific definitions from *core_cm0_psoc4.h* file and add following definitions at the top of the file:

```
#include <cytypes.h>

#define __CHECK_DEVICE_DEFINES

#define __CM0_REV                 0x0000
#define __NVIC_PRIO_BITS          2
#define __Vendor_SysTickConfig    0
```

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

High-Level I/O Functions

To use high-level input/output functions, like printf() or scanf(), the application must implement the base I/O functions. The base I/O API depends on compiler and used C library:

- GCC - Red Hat newlib C Library on sourceware.org/newlib.
- MDK – The ARM C and C++ Libraries on infocenter.arm.com.
- MDK - The ARM C Micro-library on infocenter.arm.com.

The printf() Usage Model

The printf() function formats a series of strings and numeric values and builds a string to write to the output stream. Its implementation relies on the following low-level library functions:

- Keil compiler uses the putchar()
- GCC uses _write()

- MDK uses _sys_write() or fputc(). The micro-library uses fputc().

The application should implement these functions and call the communication component API to send data via selected interface.

Preservation of Reset Status

uint32 CySysGetResetReason(uint32 reason)

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

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

Define	Source
CY_SYS_RESET_WDT	WDT
CY_SYS_RESET_PROTFAULT	Protection Fault
CY_SYS_RESET_SW	Software reset

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

Side Effects and Restrictions: None

API Memory Usage

API memory usage varies significantly depending on the compiler, device, design-wide resource configuration, and component configuration used in the design. The following tables provide the memory usage for the entire empty project with the default design-wide resource configuration options.

The measurements have been done with an associated compiler configured in Release mode with optimization set for Size. For a specific design, the map file generated by the compiler can be analyzed to determine the memory usage.

The following data is provided for a blank design with default settings. Resource usage may increase if any of unused by default cy_boot APIs are used in some particular project.

PSoC 4000 (GCC)

Configuration	PSoC 4000		
	Flash Bytes	SRAM Bytes	Stack
Default	832	144	30

PSoC 4100/PSoC 4200 (GCC)

Configuration	PSoC 4100 / PSoC 4200		
	Flash Bytes	SRAM Bytes	Stack
Default	1024	252	48

PSoC 4100 BLE/PSoC 4200 BLE (GCC)

Configuration	PSoC 4100 BLE / PSoC 4200 BLE		
	Flash Bytes	SRAM Bytes	Stack
Default	1152	252	30

PSoC 4100M/PSoC 4200M (GCC)

Configuration	PSoC 4100 / PSoC 4200		
	Flash Bytes	SRAM Bytes	Stack
Default	1280	252	30

Performance

Functions Execution Time

The API execution time varies depending on the compiler, device, and design-wide resource configuration.

The measurements have been done with the default compiler (GCC) configured in Release mode with optimization set for Size. The project uses default design-wide resource configuration for the measurements.

The following table provides the numbers for the functions whose execution time is considered to have significant impact.

PSoC 4 [2]

Description	Min	Typ	Max	Units
Device initialization time (from reset to the main() entry)	-	4.2	-	ms
The CySysFlashWriteRow() function execution time	-	12.3	-	ms

Critical Sections Duration

The duration of critical sections (code sections with disabled interrupts) varies depending on the compiler, device and, design-wide resource configuration.

The measurements have been done with the default compiler (GCC) configured in Release mode with optimization set for Size. The project used default design-wide resource configuration for the measurements.

[2] The measurements were performed on PSoC 4200 BLE devices.

The following table provides the numbers for the functions whose critical section duration might have meaningful impact.

PSoC 4

Description	Conditions	Min	Typ	Max	Units
The CySysClkWriteImoFreq() function critical section time	Default	-	302	-	cycles
The CySysWdtClearInterrupt() function critical section time	Default	-	78	-	cycles

11 MISRA Compliance

This chapter describes the MISRA-C:2004 compliance and deviations for the PSoC Creator cy_boot component and code generated by PSoC Creator.

MISRA stands for Motor Industry Software Reliability Association. The MISRA specification covers a set of 122 mandatory rules and 20 advisory rules that apply to firmware design and has been put together by the Automotive Industry to enhance the quality and robustness of the firmware code embedded in automotive devices.

There are two types of deviations defined:

- project deviations – deviations that are applicable for all PSoC Creator components
- specific deviations – deviations that are applicable for the specific component

This section provides information on the following items:

- Verification Environment
- Project Deviations
- Documentation Related Rules
- PSoC Creator Generated Sources Deviations
- cy_boot Component-Specific Deviations

Verification Environment

This section provides MISRA compliance analysis environment description.

Component	Name	Version
Test Specification	MISRA-C:2004 Guidelines for the use of the C language in critical systems.	October 2004
Target Device		
	PSoC 4	Production
Target Compiler	PK51	9.51
	GCC	4.8.4
	MDK	4.1
Generation Tool	PSoC Creator	3.1

Component	Name	Version
MISRA Checking Tool	Programming Research QA C source code analyzer for Windows	8.1-R
	Programming Research QA C MISRA-C:2004 Compliance Module (M2CM)	3.2

The MISRA rules 1.5, 2.4, 3.3, and 5.7 are not enforced by Programming Research QA C. The compliance with these rules was verified manually by code review.

Project Deviations

A Project Deviations are defined as a permitted relaxation of the MISRA rules requirements that are applied for source code that is shipped with PSoC Creator. The list of deviated rules is provided in the table below.

MISRA-C: 2004 Rule	Rule Class (R/A) [3]	Rule Description	Description of Deviation(s)
1.1	R	This rule states that code shall conform to C ISO/IEC 9899:1990 standard.	Some C language extensions (like interrupt keyword) relate to device hardware functionality and cannot be practically avoided. In the main.c file that is generate by PSoC Creator the non-standard main() declaration is used: “void main()”. The standard declaration is “int main()” The number of macro definitions exceeds 1024 - program does not conform strictly to ISO:C90.
5.1	R	This rule says that both internal and external identifiers shall not rely on the significance of more than 31 characters.	The length of names based on user-defined names depends on the length of the user-define names.
5.7	A	Verify that no identifier name should is reused.	Local variables with the same name may appear in different functions. Aside from commonly used names such as 'i', generated API functions for multiple instances of the same component will have identical local variable names.
8.7	R	Objects shall be defined at block scope if they are only accessed from within a single function.	The object 'InstanceName_initVar' is only referenced by function 'InstanceName_Start', in the translation unit where it is defined. The intention of this publicly available global variable is to be used by user application.
8.10	R	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.	Components API are designed to be used in user application and might not be used in component API.
11.3	A	This rule states that cast should not be performed between a pointer type and an integral type.	The cast from unsigned int to pointer does not have any unintended effect, as it is a consequence of the definition of a structure based on hardware registers.

[3] Required / Advisory

MISRA-C: 2004 Rule	Rule Class (R/A) [3]	Rule Description	Description of Deviation(s)
14.1	R	There shall be no unreachable code.	Some functions that are part of the component API are not used within component API. Components API are designed to be used in user application and might not be used in component API.
21.1	R	Minimization of run-time failures shall be ensured by the use of at least one of: a) static analysis tools/techniques; b) dynamic analysis tools/techniques; c) explicit coding of checks to handle run-time faults.	Some components in some specific configurations can contain redundant operations introduced because of generalized implementation approach.

Documentation Related Rules

This section provides information on implementation-defined behavior of the toolchains supported by PSoC Creator. The list of deviated rules is provided in the table below.

MISRA-C: 2004 Rule	Rule Class (R/A) [3]	Rule Description	Description
1.3	R	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform.	No multiple compilers and languages can be used at a time for PSoC Creator projects. The PK51 linker produces OMF-51 object module format. The GCC linker produces EABI format files. The MDK linker produces files of ARM ELF format.
1.4	R	The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.	PK51 and GCC treat more than 31 characters of internal and external identifier length, and are case sensitive (e.g., Id and ID are not equal).
1.5	A	Rule states that floating-point implementation should comply with a defined floating-point standard.	Floating-point arithmetic implementation conforms to IEEE-754 standard.
3.1	R	All usage of implementation-defined behavior shall be documented.	For the documentation on PK51 and GCC compilers, refer to the Help menu, Documentation sub-menu, Keil and GCC commands respectively.
3.2	R	The character set and the corresponding encoding shall be documented.	The Windows-1252 (CP-1252) character set encoding is used. Some characters that are used for source code generation in PSoC Creator are not included in character set, defined by ISO-IEC 9899-1900 "Programming languages — C".
3.3	A	This rule states that implementation of integer division should be documented.	When dividing two signed integers, one of which is positive and one negative compiler rounds up with a negative remainder.
3.5	R	This rules requires implementation defined behavior and packing of bit fields be documented.	The use of bit-fields is avoided.
3.6	R	All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.	The C standard libraries provided with C51, GCC, and RVCT have not been reviewed for compliance. Some code uses memset and memcpy. The compiler may also insert calls to its vendor-specific compiler support library.

PSoC Creator Generated Sources Deviations

This section provides the list of deviations that are applicable for the code that is generated by PSoC Creator. The list of deviated rules is provided in the table below.

MISRA-C: 2004 Rule	Rule Class (R/A) [3]	Rule Description	Description of Deviation(s)
3.4	R	All uses of the *#pragma* directive shall be documented.	The #pragma directive is required to ensure that the C51 compiler produces efficient code for generated functions related to the AMuxSeq component.
11.4	A	This rule states that cast should not be performed between a pointer to object type and a different pointer to object type.	CYMEMZERO8 and CYCONFIGCPY8 use void * arguments for compatibility with memset/memcpy but must use a pointer to an actual type internally.
14.1	R	Rule requires that there shall be no unreachable code.	The CYMEMZERO, CYMEMZERO8, CYCONFIGCPY, CYCONFIGCPY8, CYCONFIGCPYCODE, and CYCONFIGCPYCODE8 are often but not always used.
15.2	R	Switch cases must end with *break* statements.	The code structure is required to ensure that the C51 compiler produces efficient code for generated functions related to the AMuxSeq component.
15.3	R	*default* must be the last clause in a *switch* statement.	The code structure is required to ensure that the C51 compiler produces efficient code for generated functions related to the AMuxSeq component.
17.4	R	Array indexing shall be only allowed form of pointer arithmetic.	The CYMEMZERO8 and CYCONFIGCPY8 have void * arguments for compatibility with memset/memcpy.
19.7	A	The rule says that function shall be used instead of function-like macro.	The CYMEMZERO, CYMEMZERO8, CYCONFIGCPY, CYCONFIGCPY8, CYCONFIGCPYCODE, and CYCONFIGCPYCODE8 macros are used to call cymemzero, cyconfigcpy, and cyconfigcpycode in a device-independent way. The macros cannot be converted to functions without significantly increasing the time and memory required for each function call (this is a limitation of C51). The macros have been converted to functions for GCC/RVCT.

cy_boot Component-Specific Deviations [4]

This section provides the list of cy_boot component specific-deviations. The list of deviated rules is provided in the table below.

MISRA-C: 2004 Rule	Rule Class (R/A) [3]	Rule Description	Description of Deviation(s)
6.3	A	typedefs that indicate size and signedness should be used in place of the basic types.	For PSoC 4, the RealView C Library initialization function __main(void) in startup file (Cm0Start.c/Cm3Start.c) file returns value of basic type 'int'.
8.7	R	Objects shall be defined at block scope if they are only accessed from within a single function.	For PSoC 4, the cySysNoInitDataValid variable is intentionally declared as global in Cm0Start.c/Cm3Start.c files to prevent linker from CY_NOINIT section removal.
8.12	R	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.	For PSoC 4 (Cm0Start.c/Cm3Start.c), the __cy_regions array of structures is declared with unknown size.
8.8	R	An external object or function shall be declared in one and only one file.	For the PSoC 4, some objects is being declared with external linkage in Cm3Start.c/Cm3Start.c file and this declaration is not in a header file.
10.1	R	The value of an expression of integer type shall not be implicitly converted to a different underlying type under some circumstances.	PSoC 4: CMSIS Core: An integer constant of 'essentially unsigned' type is being converted to signed type on assignment in CMSIS Core hardware abstraction layer.
10.3	R	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.	The DMA API has a composite expression of 'essentially unsigned' type (unsigned char) is being cast to a wider unsigned type, 'unsigned long'. This deviation is not present for PSoC 4 cy_boot code.
14.3	R	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.	The CYASSERT() macro has null statement is located close to other code.
11.4	A	A cast should not be performed between a pointer to object type and a different pointer to object type.	The DMA and Interrupt API use casts between a pointer to object type and a different pointer to object type.
11.5		A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.	The volatile qualification is lost during pointer cast to pointer to void before passing to the memcpy() function.
17.4	R	Array indexing shall be the only allowed form of pointer arithmetic.	The DMA, Flash and Interrupt APIs use array indexing that are applied to an object of pointer type to access hardware registers, buffer allocated by user and vector tables correspondingly.

[4] The MISRA rules deviations of the CMSIS files are not documented here. Refer to the CMSIS documentation for the list of the deviated rules.

MISRA-C: 2004 Rule	Rule Class (R/A) [3]	Rule Description	Description of Deviation(s)
19.4	R	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.	The CYASSERT(), INTERRUPT_DISABLE_IRQ, INTERRUPT_ENABLE_IRQ, CyGlobalIntEnable, and CyGlobalIntDisable macro defines a braced code statement block.
19.7	A	A function should be used in preference to a function-like macro.	Deviated since function-like macros are used to allow more efficient code.
19.12	A	There shall be at most one occurrence of the # or ## preprocessor operator in a single macro definition.	PSoC 4: Pins and Bit Field Manipulation APIs: Two preprocessor concatenation operations are required as PSoC 4 APIs have two arguments.
19.13	A	The # and ## pre-processor operators should not be used.	PSoC 4: Pins and Bit Field Manipulation APIs: The preprocessor concatenation method is used to allow existing PSoC 3 and PSoC 5LP per-pin APIs to be used in PSoC 4 designs.
20.5	R	The error indicator errno shall not be used.	Caused by use of the error indicator errno used by the sbrk() function. It is used to report errors to the malloc() function if no heap memory is available.

12 System Timer (SysTick)

Functional Description

The SysTick timer is part of the Cortex M0 (PSoC 4) devices. The timer is a down counter with a 24-bit reload/tick value that is clocked by the System clock (or LF clock for the PSoC 4100 BLE and PSoC 4200 BLE devices). The timer has the capability to generate an interrupt when the set number of ticks expires and the counter is reloaded. This interrupt is available as part of the Nested Vectored Interrupt Controller (NVIC) for service by the CPU and can be used for general purpose timing control in user code.

Since the timer is independent of the CPU (except for the clock), this can be handy in applications requiring precise timing that don't have a dedicated timer/counter available for the job.

Refer to the SysTick section (Section 4.4) of the ARM reference guide for complete details on the registers and their usage.

APIs

Functions

Function	Description
CySysTickStart()	Configures and starts the SysTick timer.
CySysTickInit()	Configures the SysTick timer.
CySysTickEnable()	Enables the SysTick timer and its interrupt.
CySysTickStop()	Stops the SysTick timer.
CySysTickEnableInterrupt()	Enables the SysTick interrupt.
CySysTickDisableInterrupt()	Disables the SysTick interrupt.
CySysTickSetReload()	Sets value the counter is set to on startup and after it reaches zero.
CySysTickGetReload()	Returns SysTick reload value.
CySysTickGetValue()	Gets current SysTick counter value.
CySysTickSetClockSource()	Sets the clock source for the SysTick counter.
CySysTickGetCountFlag()	Returns the SysTick count flag value.
CySysTickClear()	Clears the SysTick counter for well-defined startup.
CySysTickSetCallback()	Sets the address(es) to the function(s) that will be called on a SysTick interrupt.
CySysTickGetCallback()	Gets the specified callback pointer.

void CySysTickStart(void)

Description: Configures the SysTick timer to generate an interrupt every 1 ms by calling the CySysTickInit() function and starts the timer by calling the CySysTickEnable() function.
Refer to the corresponding function description for the details.

Parameters: None

Return Value: None

Side Effects and Restrictions: Clears SysTick count flag if it was set.

void CySysTickInit(void)

Description: Initializes the callback addresses with pointers to NULL, associates the SysTick system vector with the function that is responsible for calling registered callback functions, configures SysTick timer to generate interrupt every 1 ms.

Parameters: None

Return Value: None

Side Effects and Restrictions: Clears SysTick count flag if it was set.

The 1 ms interrupt interval is configured based on the frequency determined by PSoC Creator at build time. If System clock frequency is changed in runtime, the CyDelayFreq() with the appropriate parameter should be called to ensure that actual frequency used for SysTick reload value calculation.

void CySysTickEnable(void)

Description: Enables the SysTick timer and its interrupt.

Parameters: None

Return Value: None

Side Effects and Restrictions: Clears SysTick count flag if it was set.

void CySysTickStop(void)

Description: Stops the system timer (SysTick).

Parameters: None

Return Value: None

Side Effects and Restrictions: Clears SysTick count flag if it was set.

void CySysTickEnableInterrupt(void)

Description: Enables the SysTick interrupt.

Parameters: None

Return Value: None

Side Effects and Restrictions: Clears SysTick count flag if it was set.

void CySysTickDisableInterrupt(void)

Description: Disables the SysTick interrupt.

Parameters: None

Return Value: None

Side Effects and Restrictions: Clears SysTick count flag if it was set.

void CySysTickSetReload(uint32 value)

Description: Sets value the counter is set to on startup and after it reaches zero.

Parameters: value: Counter reset value. Valid range [0x0-0x00FFFFFF].

For example, if the SysTick timer is configured to be clocked off the 48 MHz System Clock and interrupt every 100 us is desired, the function parameter should be 4,800 (48,000,000 Hz multiplied by 100/1,000,000 seconds).

Return Value: None

Side Effects and Restrictions: None

uint32 CySysTickGetReload(void)

Description: Returns SysTick reload value.

Parameters: None

Return Value: None

Side Effects and Restrictions: Returns SysTick reload value.

uint32 CySysTickGetValue(void)

Description: Gets current SysTick counter value.

Parameters: None

Return Value: Returns SysTick counter value.

Side Effects and Restrictions: None

void CySysTickSetClockSource(uint32 clockSource)

Description: Sets the clock source for the SysTick counter.

Parameters: uint32 clockSource:

Constant	Description
CY_SYS_SYST_CSR_CLK_SRC_SYSCLK	SysTick is clocked by the System clock.
CY_SYS_SYST_CSR_CLK_SRC_LFCLK	SysTick is clocked by the low frequency clock (LFCLK for PSoC 4).

Return Value: None

Side Effects and Restrictions: Clears SysTick count flag if it was set.

uint32 CySysTickGetCountFlag(void)

Description: The count flag is set once SysTick counter reaches zero. The flag is cleared on read.

Parameters: None

Return Value: Returns non-zero value if the counter is set, otherwise zero is returned.

Side Effects and Restrictions: Clears SysTick count flag if it was set.

void CySysTickClear(void)

Description: Clears the SysTick counter for well-defined startup. This function should be called if SysTick configuration (reload value or timer clock source) is changed. The function is called as part of the CySysTickStart() execution.

Parameters: None

Return Value: None

Side Effects and Restrictions: None

*(void *) CySysTickSetCallback(uint32 number, void(*CallbackFunction)(void))*

Description: This function allows up to five user-defined interrupt service routine functions to be associated with the SysTick interrupt. These are specified through the use of pointers to the function.

To set a custom callback function without the overhead of the system provided one, use CyIntSetSysVector(CY_INT_SYSTICK_IRQN, cyisraddress <address>), where <address> is address of the custom defined interrupt service routine.
Note: a custom callback function overrides the system defined callback functions.

Parameters: uint32 number: The number of the callback function addresses to be set. The valid range is from 0 to 4.

void(*CallbackFunction(void): A pointer to the function that will be associated with the SysTick ISR for the specified number.

Return Value: Returns the address of the previous callback function.
NULL is returned if the specified function address in not initialized.

Side Effects and Restrictions: The registered callback functions will be executed in the interrupt.

*(void *) CySysTickGetCallback(uint32 number)*

Description: The function get the specified callback pointer.

Parameters: uint32 number: The number of callback function address to get. The valid range is from 0 to 4.

Return Value: Returns the address of the specified callback function.
The NULL is returned if the specified address in not initialized.

Side Effects and Restrictions: None

Global Variables

Function	Description
uint32 cySysTickInitVar	Indicates whether or not the SysTick has been initialized. The variable is initialized to 0 and set to 1 the first time CySysTickStart() is called. This allows the component to restart without reinitialization after the first call to the CySysTickStart() routine. If reinitialization of the SysTick is required, call CySysTickInit() before calling CySysTickStart(). Alternatively, the SysTick can be reinitialized by calling the CySysTickInit() and CySysTickEnable() functions.

13 cy_boot Component Changes

Version 5.20

This section lists and describes the major changes in the cy_boot component version 5.20:

Description of Version 5.20 Changes	Reason for Changes / Impact
Updated linker scripts for adding checksum exclude section. See Bootloader/Bootloadable components datasheet for the details.	Provided method to store data in the flash section with the bootloadable application checksum not being computed over it.
Fixed CYSWAP_ENDIAN16() and CYSWAP_ENDIAN32() for signed parameters.	Defect fix.
Added information that Bit Field Manipulation API deviates the MISRA rules 19.12 and 19.13.	Datasheet changes.
Corrected CyIntSetPriority() priority parameter's valid range to be from 0 to 3.	Datasheet changes.
Datasheet update.	Added Macro Callbacks section.

Version 5.10

This section lists and describes the major changes in the cy_boot component version 5.10:

Description of Version 5.10 Changes	Reason for Changes / Impact
Updated Flash API.	Added support for future devices.
Datasheet changes.	Updated descriptions of the CySysClkImoStart(), CySysClkImoStop(), and CySysClkWriteImoFreq() functions with the Trim to WCO feature. Added descriptions of the CySysClkImoEnableWcoLock() and CySysClkImoDisableWcoLock(). Updated the API Memory Usage numbers for PSoC 4100M/PSoC 4200M.

Version 5.0

This section lists and describes the major changes in the cy_boot component version 5.0:

Description of Version 5.0 Changes	Reason for Changes / Impact
Added support for PSoC 4200M / PSoC 4200M family of devices.	New device support.
Added support for PSoC 4100 BLE and PSoC 4200 BLE family of devices with 256 K flash memory.	New device support.

Description of Version 5.0 Changes	Reason for Changes / Impact
For PSoC 4 family of devices, the APIs related to LFCLK including ILO, WCO, WDT are now part of CyLFClk system wide resource.	This change was done to streamline grouping of APIs with respect to functionality. Backward compatibility will not be affected.
New example projects for flash/EEPROM, voltage detection, interrupts, unique id have been added.	
System Reference Guide is now divided into: System Reference Guide - PSoC 3/PSoC 5LP System Reference Guide - PSoC 4 System Reference Guide - DMA (PSoC 4) System Reference Guide - CyLFClk (PSoC 4)	This change was done for ease of use of content.
New CyGetUniqueID() API support for all PSoC families.	The new API assists users in identifying each PSoC device on the field using an unique identification number.
New bit field manipulation APIs for PSoC 4 families.	The new APIs can be used to set, reset and toggle individual bit(s) of registers by their field names.
Voltage Detect API: Updated implementation of the CySysLvdEnable() functions to ensure that no false interrupts are generated.	
Voltage Detect API: Updated description of the CySysLvdEnable() function to clarify that it does not change state of the associated global interrupt.	
Updated CMSIS-Core version from 3.20 to 4.0.	
Removed the Bootloader Migration section.	Section was for older versions of Creator and not applicable to v5.0.
Added support for CMSIS-PACK.	This feature supports exporting PSoC firmware projects to Keil μVision v5.
Added attribute definitions CY_PACKED, CY_PACKED_ATTR and CY_INLINE.	Better programming support.
PSoC 4000 / PSoC 4100 / PSoC 4200: Optimized implementation of the CySysFlashWriteRow() to use less stack space.	
Clock API: optimized implementation of the CySysClkWriteImoFreq() to use less flash memory.	
SysTick API: Fixed incorrect mask being applied in the CySysTickGetValue().	To ensure that correct values are returned.
PSoC 4100 BLE/ PSoC 4200 BLE: updated implementation of the CySysClkWriteEcoDiv() to skip divider update when ECO sources.	The ECO should not source HFCLK when ECO divider value is changed. If ECO divider should be changed: switch to IMO, change ECO divider and switch back to ECO.
PM API: Replaced 'asm' with '__asm'.	To support -std GCC options.
PM API: Updated description of the CySysPmFreezeIo() and CySysPmUnfreezeIo().	
Clock API: PSoC 4200M / PSoC 4200M: Updated CySysClkImoStart(),CySysClkImoStop(), and CySysClkWriteImoFreq() function with the Trim to WCO functionality. Added CySysClkImoEnableWcoLock() and CySysClkImoDisableWcoLock().	

Description of Version 5.0 Changes	Reason for Changes / Impact
Fixed the issue when device may jump to default interrupt handler when the Link-Time Optimization options is enabled.	Ensured compiler will not inline functions executed before main().
Flash API: Updated implementation for the PSoC 4200 BLE family of devices with 256 K flash memory. The CySysFlashWriteRow() does not modify device clock settings: the IMO and HFCLK settings are not changed.	
Flash API: Update CySysFlashWriteRow() function implementation to use less stack space.	
Bootloader: Fixed the issue when bootloadable application was not allowed to be placed in the first available flash row when the “Manual application image placement” option is enabled in the Bootloadable component.	
Updated IAR linker configuration file to ensure that maximum size for the ROM vectors block is not exceeded.	Fix error that causes following message: "Error[Lp004]: actual size exceeds maximum size (0x100) for block "ROMVEC"
Bootloader: Added support for the combination project type. See Bootloader component datasheet for the details.	Added support for a new functionality of the Bootloader component.
Corrected references to #defines in CySysTickSetClockSource() function	

Version 4.20

This section lists and describes the major changes in the cy_boot component version 4.20:

Description of Version 4.20 Changes	Reason for Changes / Impact
Added support for the PSoC 4100 BLE and PSoC 4200 BLE families. Added CySysClkSetLfclkSource() function for the LFCLK clock source selection.	New device support.
PSoC 3/PSoC5LP: Updated CyWriteRowFull() function implementation to return CYRET_BAD_PARAM if invalid parameters values are passed.	
PSoC 3: Fixed a defect that caused the CyResetStatus global variable to lose its value on bootloadable application entry.	
PSoC 4: The implementation of the CY_SYS_PINS_READ_PIN macro was optimized in order to increase performance.	
PSoC 4100/PSoC 4200/PSoC 4100 BLE/ PSoC 4200 BLE: Updated implementation of the CySysClkIloStop() to ensure proper pulse length on LFCLK.	

Description of Version 4.20 Changes	Reason for Changes / Impact
PSoC 4100/PSoC 4200: WDT API: Fixed the defect in CySysWdtWriteClearOnMatch() that caused clear on match feature fails to be disabled.	
PSoC 4100/PSoC 4200/PSoC 4100 BLE/ PSoC 4200 BLE: Updated CySysPmStop() function implementation to match hardware requirements: the software delay was replaced with 2 register read-backs and corrected the procedure of the low power mode entry.	
PSoC 4100/PSoC 4200/PSoC 4100 BLE/ PSoC 4200 BLE: Fixed the order of the Stop mode entry in the CySysPmStop() function to ensure that Stop mode token is set at the beginning of the low power mode entry.	Omit the situation when GPIO pins remain frozen after the reset if reset occurred after IO pin freeze but before Stop mode entry.
Added following attribute macros: CY_PACKED, CY_PACKED_ATTR and CY_INLINE.	
The declaration of the IntDefaultHandler created in CyLib.h.	Previously, the IntDefaultHandler was declared in both interrupt source file and Cm0Start.c files.
PSoC 4000: Corrected the lower bound of the HFCLK frequency change from the current IMO frequency divided by 8 to divided by 4 in the wside effects section of the CySysFlashWriteRow() function.	
PSoC 3/ PSoC 5LP: Updated implementation of the CySetTemp() function in order to improve execution time of the first call after Power-On-Reset (POR).	Significantly improved the first Flash write after POR.
PSoC 4/PSoC 5LP: Added sbrk() function, which is used by malloc() and other heap-utilizing functions to check for available memory.	The fix ensures that malloc(), et al, now correctly handle heap overflow. Note that some projects will now fail to execute due to a lack of available heap. The resolution is to increase the heap size in the Design-Wide Resources System Editor (*<project>.cydwr* file), and re-build the project.
PSoC 4/ PSoC 5LP: Added the following MISRA rule deviations: 20.5.	Caused by use of the error indicator errno used by sbrk() function. It is used to report error to the malloc() function if no heap memory available.
PSoC 4100/PSoC 4200/PSoC 4100 BLE/ PSoC 4200 BLE: • Updated CySysWdtEnable() function implementation to ensure that WDT is enabled upon function exit; • Updated CySysWdtWriteMatch() function implementation to ensure that match value is updated properly: add delay before (ensures that last update applied properly) and after value change (ensures that match update synchronization started). • Updated CySysWdtDisable() function implementation to ensure that WDT is disabled upon function exit.	

Description of Version 4.20 Changes	Reason for Changes / Impact
PSoC 4/PSoC 5LP: Updated IAR linker script file to eliminate warning generated by the IAR EW-ARM v7.10.	
PSoC 4: The CySysFlashWriteRow() function return type changed from cystatus to uint32.	To follow hardware-defined error codes. The basic behavior remains the same: zero for success and non-zero for any type of failure.
PSoC 5LP: The CyFlash_SetWaitCycles() function is updated with 80 MHz parts support.	
PSoC 4/PSoC 5LP: Added System Timer (SysTick) API.	
PSoC 3/PSoC 5LP: Flash/EEPROM API: updated implementation to eliminate requirement to call CySetFlashEEBuffer() function, if the Flash ECC feature is disabled.	No need to allocate buffer and pass it to CySetFlashEEBuffer() for both Flash and EEPROM programming.
PSoC 3/PSoC 5LP: Flash API: added CY_EEPROM_NUMBER_SECTORS and CY_EEPROM_SIZEOF_SECTOR.	Defined macros for the number of EEPROM sectors and size of EEPROM sector.
PSoC 4/PSoC 5LP: Interrupt API: added macros for the CyIntSetSysVector() and CyIntGetSysVector() functions exception type numbers.	
PSoC 3: The CyPmSleep() and CyPmHibernate() functions disable clock to the interrupt controller before Sleep and Hibernate mode entry and re-enable on wakeup.	Satisfy interrupt controller usage model.
PSoC 3/PSoC 5LP: Updated CyFlash_Start() and CyEEPROM_Start() functions implementation.	To ensure that EEPROM and Flash are ready for operation on corresponding function exit.
PSoC 5LP: Changed CyFlushCache() implementation.	To use Instruction Synchronization Barrier (ISB) instruction instead of multiple no operation instructions.
PSoC 4: The CY_SYS_PINS_READ_PIN macro was optimized for the better performance.	
PSoC 4200/PSoC 4100: updated CySysClkWriteImoFreq() function for better performance.	
PSoC 4: Added the following MISRA rule deviations: 19.12 and 19.13.	Added the possibility for existing PSoC 3 and PSoC 5LP per-pin APIs to be used in PSoC 4 designs.
Updated the following MISRA rule deviations: 12.10, 12.13, 13.2, and 13.5.	
PSoC 4000: Update WDT API description to clarify that CySysWdtEnable() and CySysWdtDisable() correspondingly enables and disables the watchdog timer reset generation.	
PSoC 4000: Fixed the implementation of the CySysWdtReadIgnoreBits() to return correct number of the ignored bits in the WDT counter.	
PSoC 3/PSoC 5LP: removed LVI/HVI reset constants for the CyResetStatus global variable in section “Preservation of Reset Status”.	The LVI and HVI resets are not reported by CyResetStatus variable.

Description of Version 4.20 Changes	Reason for Changes / Impact
PSoC 4100/PSoC 4200: Power Management API: Updated CySysPmDeepSleep() function to bypass the flash accelerator before Deep Sleep mode entry and restore it upon wakeup.	Cypress identified a defect with the Flash write functionality upon wakeup from deep-sleep in PSoC 4100 and PSoC 4200 devices. The corrupted data has the potential to be sent to the CPU on device wakeup.

Version 4.11

This section lists and describes the major changes in the cy_boot component version 4.11:

Description of Version 4.11 Changes	Reason for Changes / Impact
The CySysFlashWriteRow() function now checks the data to be written and, if necessary, modifies it to have a non-zero checksum. After writing to Flash, the modified data is replaced (Flash program) with the correct (original) data.	Cypress identified a defect with the Flash write functionality of the PSoC 4000, PSoC 4100, and PSoC 4200 devices. The CySysFlashWriteRow() function in the cy_boot [v4.0 and v4.10] component fails to write a row of flash memory if the data to be written has a zero in the lower 32-bits of the checksum.

Version 4.10

This section lists and describes the major changes in the cy_boot component version 4.10:

Description of Version 4.10 Changes	Reason for Changes / Impact
PSoC 4: Added CySysGetResetReason() function.	Reports the cause for the latest reset(s) that occurred in the system.
Added support for the PSoC 4000 family.	New device support.
PSoC 3: Added reentrancy support for the CySpcLock() and CySpcUnlock() functions.	
PSoC 3/ PSoC 5LP: Fixed the defect in CyPmRestoreClocks() function, that can might to the device halt during the function execution, in some clock system configurations, when PLL is not sourced by IMO and IMO is manually stopped by user code.	
PSoC 4: Added note that enabling or disabling a WDT requires three LFCLK cycles to come into effect, during that period the SYSCLK should be available.	The device should not put into Deep Sleep mode during that period.
PSoC 4: Added note that, after waking from Deep Sleep, the WDT internal timer value is set to zero until the ILO loads the register with the correct value.	This led to an increase in low-power mode current consumption. The work around is to wait for the first positive edge of the ILO clock before allowing the WDT_CTR_* registers to be read by CySysWdtReadCount() function.
Added note to the "Working with Flash and EEPROM" section with the information that CPU code execution can be halted till the flash write is complete.	

Description of Version 4.10 Changes	Reason for Changes / Impact
Added note to the "Working with Flash and EEPROM" section with the information that power manager will not put the device into a low power state if the system performance controller (SPC) is executing a command.	
PSoC 3 / PSoC 5LP: The CyPmRestoreClocks() implementation was enhanced by polling status and proceed as soon as PLL is locked. Added merge section to add ability of handling cases when predefined timeout is not enough.	
PSoC 4: Fixed a defect in CySysWdtClearInterrupt() that caused unintentional clearing of the WDT interrupt status bit.	

Version 4.0

This section lists and describes the major changes in the cy_boot component version 4.0:

Description of Version 4.0 Changes	Reason for Changes / Impact
Added note to the Flash section about unavailability of the Store Configuration Data in ECC Memory DWR option for the bootloader project type.	
Added note to the Working with Flash and EEPROM section that when writing Flash, data in the instruction cache can become stale.	Call CyFlushCache() to invalidate the data in cache and force fresh information to be loaded from Flash.
Fixed issue in the CyDmaChEnable() and CyDmaChDisable() functions.	If DMA request occurred during these functions, the DMA channels configuration could be corrupted. The APIs were changed to address this problem.
Removed references to PSoC 5 device.	PSoC 5 has been replaced by PSoC 5LP.
PSoC Creator Generated Sources Deviations section was updated with the MISRA deviations related to the AMuxSeq component.	
The CY_IMO_FREQ_74MHZ parameter was added to the CyIMO_SetFreq() function.	Support of the 80 MHZ PSoC 5LP devices.
PSoC 4: Added CyExitCriticalSection() function call after WFI instruction in the CySysPmHibernate() function.	If any interrupt occurred between CyEnterCriticalSection() and WFI instruction execution, the device could skip low power mode entry request and continue code execution with global interrupts disabled.

Version 3.40 and Older

Version 3.40

This section lists and describes the major changes in the cy_boot component version 3.40:

Description of Version 3.40 Changes	Reason for Changes / Impact
Added PSoC 4 device support.	New device support.
PSoC 3: Updated CyPmSleep() function description with the information that hardware buzz must be disabled before sleep mode entry. As hardware buzz is required for LVI, HVI, and Brown Out detect operations – they must be disabled before sleep mode entry and restored on wakeup. If LVI or HVI is enabled, CyPmSleep() will halt device if project is compiled in debug mode.	Using hardware buzz in conjunction with other device wakeup sources can cause the device to lockup, halting further code execution. Refer to the device errata for more information.

Version 3.30

This section lists and describes the major changes in the cy_boot component version 3.30:

Description of Version 3.30 Changes	Reason for Changes / Impact
Updates to support PSoC Creator 2.2.	
Added MISRA Compliance section.	
Added Low Voltage Analog Boost Clocks section.	New feature for the SC-based (TIA, Mixer, PGA and PGA_Inv) components.
Added requirement about interrupt configuration, when interrupt is sources from PICU and used as a wakeup event.	For PSoC 5LP, the interrupt component connected to the wakeup source may not use the "RISING_EDGE" detect option. Use the "LEVEL" option instead.
The delay between Bus clock and analog clocks configuration save/restore moved from CyPmSleep() and CyPmHibernate() functions to CyPmSaveClocks() / CyPmRestoreClocks().	This modification decrease CyPmSleep() and CyPmHibernate() functions execution time. The components that use analog clock must not be used after CyPmSaveClocks() execution till the clocks configuration will be restored by CyPmRestoreClocks().
Added float32 and float64 data types. The type float64 is not available for PSoC 3 devices.	

Version 3.20

This section lists and describes the major changes in the cy_boot component version 3.20:

Description of Version 3.20 Changes	Reason for Changes / Impact
Many minor edits throughout the document to distinguish features of PSoC 5 and PSoC 5LP devices.	Improve PSoC 5 and PSoC 5LP documentation.
The PSoC 5LP Alternate Active usage model was changed to be same as for PSoC 5.	No parameters are used for CyPmAltAct(). That means NONE should be passed for the parameters. The device will go into Alternate Active mode until an enabled interrupt occurs.

Description of Version 3.20 Changes	Reason for Changes / Impact
The interface of the CyIMO_SetFreq() function was updated for PSoC 5LP to support 62 and 72 MHz frequencies.	Added interface to configure IMO to 62 and 72 MHz on PSoC 5LP.

Version 3.10

This section lists and describes the major changes in the cy_boot component version 3.10:

Description of Version 3.10 Changes	Reason for Changes / Impact
The Bootloader system was redesigned in cy_boot version 3.0 to separate the Bootloader and Bootloadable components. The change is listed here as well for migrating from older versions.	See Bootolader Migration section in cy_boot version 3.10 System Reference Guide.
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 Bootolader Migration section in cy_boot version 3.0 System Reference Guide.
The CyPmSleep() function implementation was updated to preserve/restore PRES state before/after Sleep mode. The support of the HVI/LVI functionality added.	New functionality support.
Added following Voltage Detect APIs: CyVdLvDigitEnable(),CyVdLvAnalogEnable(),CyVdLvDigitDisable(),CyVdLvAnalogDisable(),CyVdHvAnalogEnable(),CyVdHvAnalogDisable(),CyVdStickyStatus() and CyVdRealTimeStatus().	Added voltage monitoring APIs.
The implementation of the Flash API was slightly modified as the SPC API used in Flash APIs was refactored.	The implementation quality improvements.
The implementation of the CyXTAL_32KHZ_Start(), CyXTAL_32KHZ_Stop(), CyXTAL_32KHZ_ReadStatus() and CyXTAL_32KHZ_SetPowerMode() APIs was updated.	Added additional timeouts to ensure proper block start-up.
The implementation of the CyXTAL_Start() function for PSoC 5 parts was changed. For more information on function see Clocking section.	Changes were made to make sure that MHZ XTAL starts successfully on PSoC 5 parts.

Description of Version 3.0 Changes	Reason for Changes / Impact
The following APIs were removed for PSoC 5 parts: CyXTAL_ReadStatus(), CyXTAL_EnableErrStatus(), CyXTAL_DisableErrStatus(), CyXTAL_EnableFaultRecovery(), CyXTAL_DisableFaultRecovery().	The functionality provided within these APIs is not supported by the PSoC 5 part.
The CyDmacConfigure() function is now called by the startup code only if DMA component is placed onto design schematic.	Increase device startup time in case if DMA is not used within design. The CyDmacConfigure() function should be called manually if DMA functionality is used without DMA component.
The CyXTAL_32KHZ_ReadStatus() function implementation was changed by removing digital measurement status return.	The analog status measurement is the only reliable source.
Updated description of following APIs: CyFlash_SetWaitCycles().	Changes were made to improve power mode configuration.
The address of the top of reentrant stack was decremented from CYDEV_SRAM_SIZE to (CYDEV_SRAM_SIZE - 3) for PSoC 3.	Prevent rewriting CyResetStatus variable with the parameters and/or local variables of the reentrant function during its execution.
The CyIMO_SetFreq() function implementation was updated by removing support of 74 and 62 MHz parameters for PSoC 5 parts.	Removal of the functionality that is not supported by device.
The minimal P divider value for the CyPLL_OUT_SetPQ() was risen from 4 up to 8.	To meet hardware requirements
The CyXTAL_SetFbVoltage()/SetWdVoltage() were added for PSoC 5LP devices.	The functionality provided by these APIs is available in PSoC 5LP.
The description of the CyWdtStart() was updated.	Added notes on WDT operation during low power modes for PSoC 5.
The implementation of the CyPmSleep() for PSoC 5 was changed not to hold CTW in reset on wakeup.	Not putting CTW in reset state on wakeup allows to combine CTW usage in both Active and low power modes for PSoC 5.
The *Preservation of Reset Status* section was updated with more detailed information.	The software reset behavior of other resets is explained. Explained how the reset status variable can be used.
Updated description of following APIs: CyMasterClk_SetDivider(),CyWdtStart(),CyWdtStart().	To reflect implementation better.
The Startup and Linking section was updated. The information on using custom linker script was added.	To provide more information on device operation.
Following macros were removed: CYWDT_TICKS CYWDT_CLEAR, CYWDT_ENABLE CYWDT_DISABLE_AUTO_FEED.	The CyWdtStart() and CyWdtClear() should be used instead.
The CyCpuClk_SetDivider() was removed for PSoC 5 devices.	The hardware does not support this functionality.
The cystrcpy(), cystrlen(), CyGetSwapReg16() and CySetSwapReg16() APIs were removed.	The library functions should be used.
The return value description for CyEnterCriticalSection() function was updated for PSoC 5.	Function returns 0 if interrupts were previously enabled or 1 if interrupts were previously disabled.

Description of Version 3.0 Changes	Reason for Changes / Impact
Added all APIs with the CYREENTRANT keyword when they are included in the .cyre file.	Not all APIs are truly reentrant. Comments in the component API source files indicate which functions are candidates. 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 and Older

Version 2.30 and older are obsolete.