PSoC® Creator™

System Reference Guide

cy_boot Component v2.21

Document Number: 001-67776, Rev. *A

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

© Cypress Semiconductor Corporation, 2011. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

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

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

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

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

Contents



1 Introduction 5
- Conventions 6
- References 6
- Revision History 6

2 Standard Types and Defines 7
- Base Types 7
- Hardware Register Types 7
- Compiler Defines 7
- Keil 8051 Compatibility Defines 8
- Return Codes 8
- Interrupt Types and Macros 9
- Intrinsic Defines 9

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

4 Power Management 31
- APIs 31
- Instance Low Power APIs 37

5 Interrupts 39
- APIs 39

6 Cache 43
- PSoC 3 Cache Functionality 43
- PSoC 5 Cache Functionality 43

7 Pins 45
- APIs 45

8 Register Access 49
- APIs 49

9 DMAC 55

10 Flash and EEPROM 57
Hardware Configuration Options 57
APIs 58

11 Bootloader System 63
Bootloader Component 63
Communications Component 64
Custom Bootloader Component 65
Bootloader Project Types 67
Memory Usage 68
Bootloader Parameters 70
Bootloader API 72
Bootloader Commands 72
Bootloader Packets 76
Bootloader Application Status/Error Codes 77
Bootloader Host Status/Error Codes 77
Bootloader Application Code & Data File Format 77
Bootloader Host Tool 78

12 System Functions 80
APIs 80

13 Startup and Linking 83
PSoC 3 83
PSoC 5 83
CMSIS Support (PSoC 5 only) 84
Preservation of Reset Status 84

14 Watchdog Timer 85
APIs 85

15 cy_boot Component Changes 87
Version 2.21 87
Version 2.20 87
Version 2.10 88
Version 2.0 88

1 Introduction



This System Reference document 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. The developer 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

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

- DMA
- Flash
- SPC
- Clocking
- Power management
- Start up code
- Various library functions
- Linker scripts

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

Conventions

The following table lists the conventions used throughout this guide:

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

References

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

- PSoC Creator Help
- PSoC Creator Component Datasheets
- PSoC Creator Component Author Guide
- PSoC 3 and PSoC 5 Technical Reference Manual (TRM)

Revision History

Document Title: PSoC® Creator™ System Reference Guide, cy_boot Component v2.21 **Document Number: 001-67776, Rev. *A**		
Revision	**Date**	**Description of Change**
**	5/18/11	New document for version 2.21 of cy_boot component. Refer to change section for component changes from previous version.
*A	7/13/11	Updated Bootloader section for custom communication option.

2 Standard Types and Defines

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

Base Types

Type	Description
char8	8-bit unsigned
uint8	8-bit unsigned
uint16	16-bit unsigned
uint32	32-bit unsigned
int8	8-bit signed
int16	16-bit signed
int32	32-bit signed

Hardware Register Types

Hardware registers in PSoC 3 and PSoC 5 typically have side effects and therefore are referenced with a volatile type.

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

Compiler Defines

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

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

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

Keil 8051 Compatibility Defines

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

Define	Keil Type
CYBDATA	bdata
CYBIT	bit
CYCODE	code
CYCOMPACT	compact
CYDATA	data
CYFAR	far
CYIDATA	idata
CYLARGE	large
CYPDATA	pdata
CYREENTRANT	reentrant
CYSMALL	small
CYXDATA	xdata

Return Codes

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

Define	Description
CYRET_SUCCESS	Successful
CYRET_UNKNOWN	Unknown failure
CYRET_BAD_PARAM	One or more invalid parameters
CYRET_INVALID_OBJECT	Invalid object specified
CYRET_MEMORY	Memory related failure

Define	Description
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

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 (DWR) settings, connectivity of clocking signals on the design schematic, and API calls that can modify the clocking at runtime.

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

Clock Connectivity

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

BUS_CLK

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

Global Clock

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

Routed Clock

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

Clock Synchronization

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

Synchronous Clock

Examples of synchronous clocks include:

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

Asynchronous Clock

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

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

Making Signals Synchronous

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

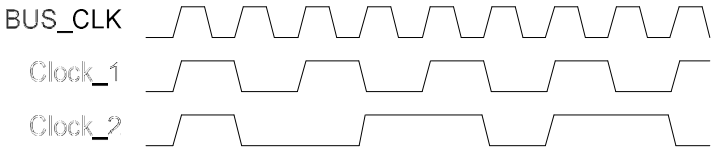
- An asynchronous global clock can be made synchronous by checking the "Sync with MASTER_CLK" option in the Clock component Configure dialog (this is the default selection).
- A routed clock coming from a pin can be made synchronous by checking the "Input Synchronized" option in the Pins component Configure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

When synchronizing a signal:

- The synchronizing clock must be at least 2x the frequency of the signal being synchronized. If this rule is violated, then incoming clock edges can be missed and therefore not reflected in the resulting synchronized clock.
- The clock signal output will have all its transitions on the rising edge of the synchronizing clock.

- The clock signal output will have its edges moved from their original timing.
- The clock signal output will have variation in the high and low pulse widths unless the incoming clock and the synchronizing clock are directly related to each other.

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



Routed Clock Implementation

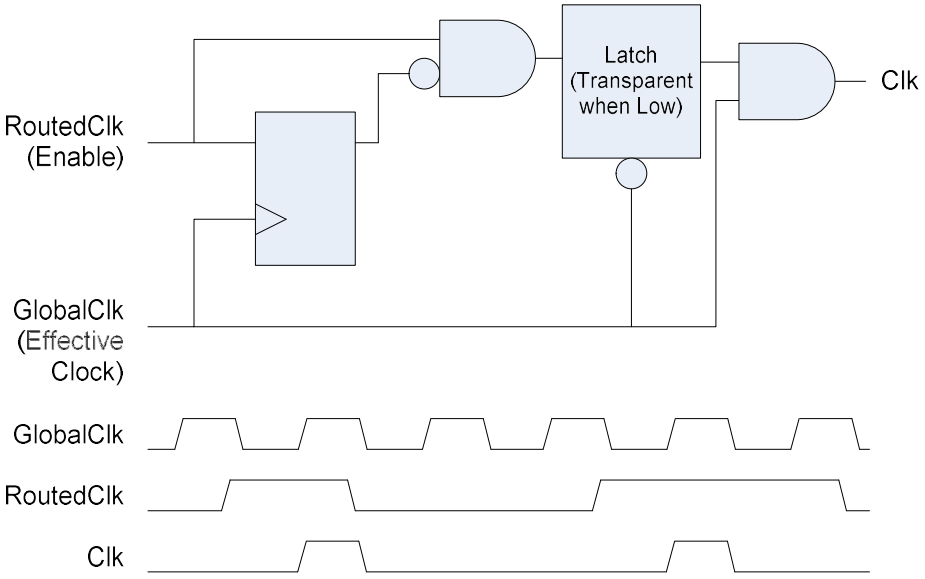
The clocking implementation in PSoC directly connects global clock signals to the clock input of clocked digital logic. This applies to both synchronous and asynchronous clocks. Since global clocks are distributed on low skew clock lines, all clocked elements connected to the same global clock will be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

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

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

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

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



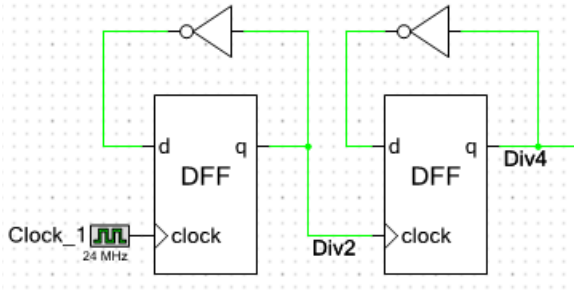
This diagram shows that the resulting clock occurs synchronous to the global clock on the first clock after a rising edge of the routed clock.

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

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

Example with a Divided Clock

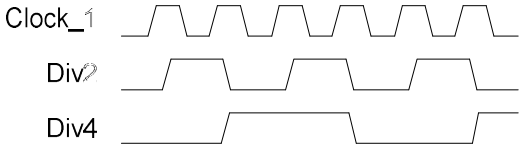
A simple divided clock circuit can be used to observe how this transformation is done. The following circuit clocks the first flip-flop (cydff_1) with a global clock. This generates a clock that is divided by 2 in frequency. That signal is used as a routed clock that clocks the next flip-flop (cydff_2).



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

```
<CYPRESSTAG name="Tech mapping">
<CYPRESSTAG name="Initial Mapping" icon="FILE_RPT_TECHN">
<CYPRESSTAG name="Global Clock Selection" icon="FILE_RPT_TECHN">
    Digital Clock 0: Automatic-assigning  clock 'Clock_1'. Fanout=1, Signal=tmp_cydff_1_clk
</CYPRESSTAG>
<CYPRESSTAG name="UDB Routed Clock Assignment">
    Routed Clock: tmp_cydff_1_reg:macrocell.q
        Effective Clock: Clock_1
        Enable Signal: tmp_cydff_1_reg:macrocell.q
</CYPRESSTAG>
```

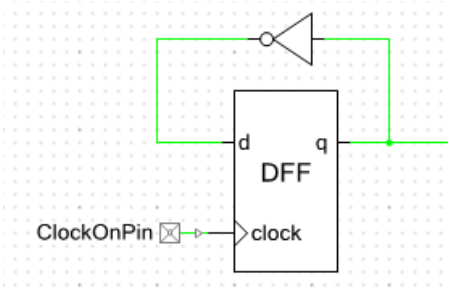
The resulting signals generated by this circuit are as follows.



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

Example with a Clock from a Pin

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



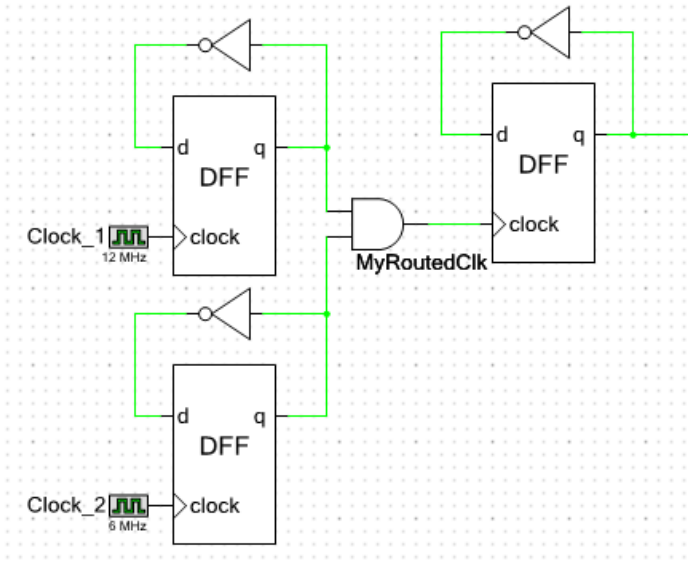

```
<CYPRESSTAG name="Initial Mapping" icon="FILE_RPT_TECHN">
{Global Clock Selection}
<CYPRESSTAG name="UDB Routed Clock Assignment">
    Routed Clock: ClockOnPin(0):iocell.fb
        Effective Clock: BUS_CLK
        Enable Signal: ClockOnPin(0):iocell.fb
</CYPRESSTAG>
```

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

Example with Multiple Clock Sources

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

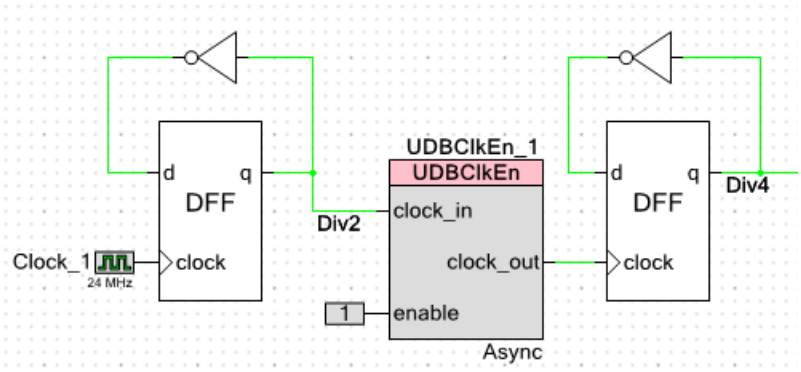



```
<CYPRESSTAG name="Tech Mapping">
<CYPRESSTAG name="Clock Mapping" icon="CLK_RPT_TREE">
<CYPRESSTAG name="Global Clock Selection" icon="CLK_RPT_TREE">
    Digital Clock 0: Automatic-assigning  clock 'Clock_1', Fanout: 1, Signal: tmp__cydff_1_clk
    Digital Clock 1: Automatic-assigning  clock 'Clock_2',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

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

Overriding Routed Clock Transformations

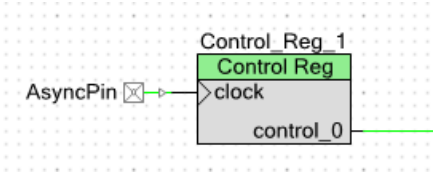
The automatic transformation that PSoC Creator performs on routed clocks is generally the implementation that should be used. There is however a method to force the routed clock to be used directly. The UDBClkEn component configured in Async mode will force the clock used to be the routed clock, as shown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

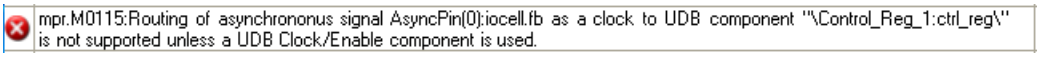


Using Asynchronous Clocks

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

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





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

Clock Crossing

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

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

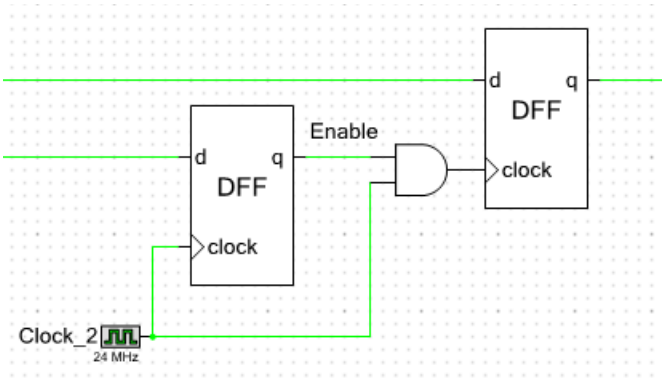
The only guarantee between the clock domains is that their edges will always occur on a rising edge of BUS_CLK. That means that the rising edges of the two clock domains can be as close as a single BUS_CLK cycle apart. This is true even when the clock domains are multiples of each other, since their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

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

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

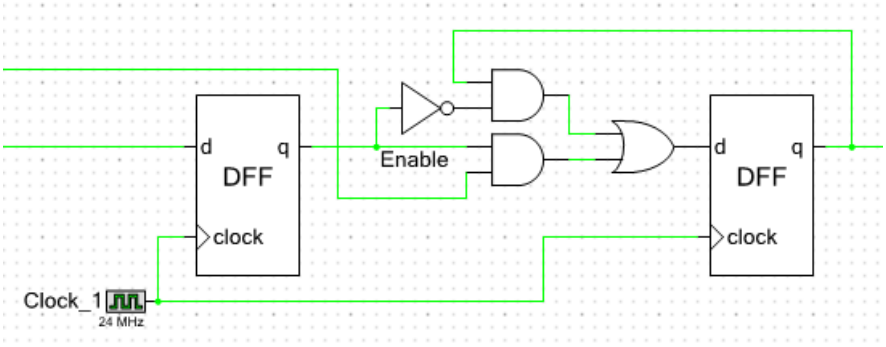
Gated Clocks

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



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

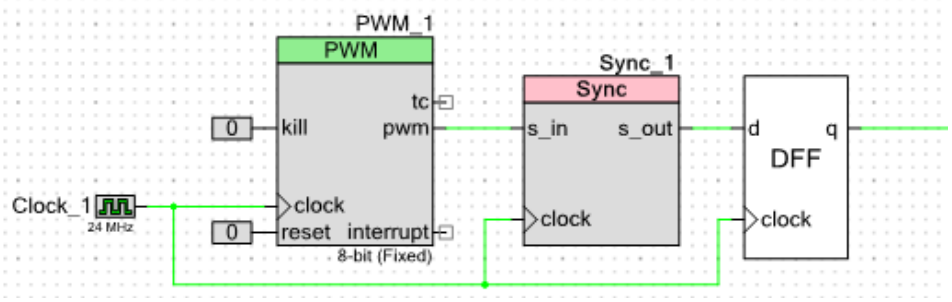
The following circuit implements the equivalent function and is supported by timing analysis, only uses global clocks, and has no reliability issues. This circuit does not gate the clock, but instead logically enables the clocking of new data or maintains the current data.



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

Fixed-Function Clocking

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



Clock APIs

uint8 CyPLL_OUT_Start(uint8 wait)

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

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

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

Side Effects and Restrictions: If wait is enabled:
Uses the Fast Timewheel to time the wait. Any other use of the Fast Timewheel will be stopped during the period of this function and then restored.
Uses the 100KHz ILO. If not enabled, this function will enable the 100KHz ILO for the period of this function.
No changes to the setup of the ILO, Fast Timewheel, Central Timewheel or Once Per Second interrupt may be made by interrupt routines during the period of this function. The current operation of the ILO, Central Timewheel and Once Per Second interrupt are maintained during the operation of this function provided the reading of the Power Manager Interrupt Status Register is only done using the CyPMReadStatus() function.

void CyPLL_OUT_Stop()

Description: Disables the PLL.

Parameters: None

Return Value: None

void CyPLL_OUT_SetPQ(uint8 P, uint8 Q, uint8 current)

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

Parameters: P: Valid range [4 - 255]

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

current: Valid range [1 - 7]. Charge pump current in uA. Recommendation of 2uA for output frequencies of 67MHz or less and 3uA for higher output frequencies.

Return Value: None

void CyPLL_OUT_SetSource(uint8 source)

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

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

Value	Define	Source
0	CY_PLL_SOURCE_IMO	IMO
1	CY_PLL_SOURCE_XTAL	MHz Crystal
2	CY_PLL_SOURCE_DSI	DSI

Return Value: None

void CyIMO_Start(uint8 wait)

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

Parameters: wait:

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

Return Value: None

Side Effects and Restrictions: If wait is enabled:
Uses the Fast Timewheel to time the wait. Any other use of the Fast Timewheel will be stopped during the period of this function and then restored.
Uses the 100KHz ILO. If not enabled, this function will enable the 100KHz ILO for the period of this function.
No changes to the setup of the ILO, Fast Timewheel, Central Timewheel or Once Per Second interrupt may be made by interrupt routines during the period of this function. The current operation of the ILO, Central Timewheel and Once Per Second interrupt are maintained during the operation of this function provided the reading of the Power Manager Interrupt Status Register is only done using the CyPMReadStatus() function.

void CyIMO_Stop()

Description: Disables the IMO.

Parameters: None

Return Value: None

void CyIMO_SetFreq(uint8 freq)

Description: Sets the frequency of the IMO. Changes may be made while the IMO is running. When the USB setting is chosen the USB clock locking circuit is enabled. Otherwise this circuit is disabled. If the IMO is currently driving the Master clock, then the Flash wait states must be set appropriately before making this change by using CyFlash_SetWaitCycles().

Parameters: freq: Frequency of IMO operation

Value	Define	Frequency
0	CY_IMO_FREQ_3MHZ	3 MHz
1	CY_IMO_FREQ_6MHZ	6 MHz
2	CY_IMO_FREQ_12MHZ	12 MHz
3	CY_IMO_FREQ_24MHZ	24 MHz
4	CY_IMO_FREQ_48MHZ	48 MHz
5	CY_IMO_FREQ_62MHZ	62.6 MHz
6	CY_IMO_FREQ_74MHZ	74.7 MHz (PSoC 5)
8	CY_IMO_FREQ_USB	24 MHz (Trimmed for USB operation)

Return Value: None

void CyIMO_SetSource(uint8 source)

Description: Sets the source of the clock output from the IMO block. The output from the IMO is by default the IMO itself. Optionally the MHz Crystal or a DSI input can be the source of the IMO output instead. If the IMO is currently driving the Master clock, then the Flash wait states must be set appropriately before making this change by using CyFlash_SetWaitCycles().

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

Value	Define	Source
0	CY_IMO_SOURCE_IMO	IMO
1	CY_IMO_SOURCE_XTAL	MHz Crystal
2	CY_IMO_SOURCE_DSI	DSI

Return Value: None

void CyIMO_EnableDoubler()

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

Parameters: None

Return Value: None

void CyIMO_DisableDoubler()

Description: Disables the IMO doubler.

Parameters: None

Return Value: None

void CyMasterClk_SetSource(uint8 source)

Description: Sets the source of the master clock. The current source and the new source must both be running and stable before calling this function. The Flash wait states must be set appropriately before making this change by using CyFlash_SetWaitCycles().

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

Value	Define	Source
0	CY_MASTER_SOURCE_IMO	IMO
1	CY_MASTER_SOURCE_PLL	PLL
2	CY_MASTER_SOURCE_XTAL	MHz Crystal
3	CY_MASTER_SOURCE_DSI	DSI

Return Value: None

void CyMasterClk_SetDivider(uint8 divider)

Description: Sets the divider value used to generate Master Clock. The Flash wait states must be set appropriately before making this change by using CyFlash_SetWaitCycles().

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

Return Value: None

void CyBusClk_SetDivider(uint16 divider)

Description: Sets the divider value used to generate Bus Clock. The Flash wait states must be set appropriately before making this change by using CyFlash_SetWaitCycles().

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

Return Value: None

void CyCpuClk_SetDivider(uint8 divider)

Description: Sets the divider value used to generate the CPU Clock. Only applicable for PSoC 3.

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

Return Value: None

void CyUsbClk_SetSource(uint8 source)

Description: Sets the source of the USB clock.

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

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

Return Value: None

void CyILO_Start1K()

Description: Enables the ILO 1 KHz oscillator.

Parameters: None

Return Value: None

void CyILO_Stop1K()

Description: Disables the ILO 1 KHz oscillator.

Parameters: None

Return Value: None

void CyILO_Start100K()

Description: Enables the ILO 100 KHz oscillator.

Parameters: None

Return Value: None

void CyILO_Stop100K()

Description: Disables the ILO 100 KHz oscillator.

Parameters: None

Return Value: None

void CyILO_Enable33K()

Description: Enables the ILO 33 KHz divider. Note that the 33 KHz clock is generated from the 100 KHz oscillator, so it must also be running in order to generate the 33 KHz output.

Parameters: None

Return Value: None

void CyILO_Disable33K()

Description: Disables the ILO 33 KHz divider. Note that the 33 KHz clock is generated from the 100 KHz oscillator, but this API does not disable the 100 KHz clock.

Parameters: None

Return Value: None

void CyILO_SetSource(uint8 source)

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

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

Value	Define	Source
0	CY_ILO_SOURCE_100K	ILO 100 KHz
1	CY_ILO_SOURCE_33K	ILO 33 KHz
2	CY_ILO_SOURCE_1K	ILO 1 KHz

Return Value: None

uint8 CyILO_SetPowerMode(uint8 mode)

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

Parameters: mode:

- 0: Faster start-up, internal bias left on when powered down
- 1: Slower start-up, internal bias off when powered down

Return Value: Previous power mode

void CyXTAL_32KHZ_Start()

Description: Enables the 32 KHz Crystal Oscillator.

Parameters: None

Return Value: None

void CyXTAL_32KHZ_Stop()

Description: Disables the 32 KHz Crystal Oscillator.

Parameters: None

Return Value: None

uint8 CyXTAL_32KHZ_ReadStatus()

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

Parameters: None

Return Value: Status

Value	Define	Source
0x20	CY_XTAL32K_ANA_STAT	Analog measurement 1: Stable 0: Not stable
0x10	CY_XTAL32K_DIG_STAT	Digital measurement (Requires the 33 KHz ILO to make this measurement) 1: Stable 0: Not stable

uint8 CyXTAL_32KHZ_SetPowerMode(uint8 mode)

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

Parameters: mode:

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

Return Value: Previous power mode

uint8 CyXTAL_Start(uint8 wait)

Description:	Enables the megahertz crystal. Waits until the XERR bit is low (no error) for a millisecond or until the number of milliseconds specified by the wait parameter has expired.
Parameters:	wait: Valid range [0-255]. This is the timeout value in milliseconds. The appropriate value is crystal specific.
Return Value:	Status CYRET_SUCCESS - Completed successfully CYRET_TIMEOUT - Timeout occurred without detecting a low value on XERR.
Side Effects and Restrictions:	If wait is enabled (non-zero wait): Uses the Fast Timewheel to time the wait. Any other use of the Fast Timewheel will be stopped during the period of this function and then restored. Uses the 100 KHz ILO. If not enabled, this function will enable the 100 KHz ILO for the period of this function. No changes to the setup of the ILO, Fast Timewheel, Central Timewheel or Once Per Second interrupt may be made by interrupt routines during the period of this function. The current operation of the ILO, Central Timewheel and Once Per Second interrupt are maintained during the operation of this function provided the reading of the Power Manager Interrupt Status Register is only done using the CyPMReadStatus() function.

void CyXTAL_Stop()

Description:	Disables the megahertz crystal oscillator.
Parameters:	None
Return Value:	None

void CyXTAL_EnableErrStatus()

Description:	Enables the generation of the XERR status bit for the megahertz crystal.
Parameters:	None
Return Value:	None

void CyXTAL_DisableErrStatus()

Description:	Disables the generation of the XERR status bit for the megahertz crystal.
Parameters:	None
Return Value:	None

uint8 CyXTAL_ReadStatus()

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

Parameters: None

Return Value: Status
- 0: No error
- 1: Error

void CyXTAL_EnableFaultRecovery()

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

Parameters: None

Return Value: None

void CyXTAL_DisableFaultRecovery()

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

Parameters: None

Return Value: None

void CyXTAL_SetStartup(uint8 setting)

Description: Sets the startup settings for the crystal.

Parameters: setting: Valid range [0-31]. Value is dependent on the frequency and quality of the crystal being used. Refer to the TRM for appropriate values for a specific crystal.

Return Value: None

void CyXTAL_SetFbVoltage(uint8 setting)

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

Parameters: setting: Valid range [0-15]. Refer to the TRM for details on the mapping of the setting value to specific voltages.

Return Value: None

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

void CyXTAL_SetWdVoltage(uint8 setting)

Description:	Sets the reference voltage used by the watchdog to detect a failure in the crystal circuit.
Parameters:	setting: Valid range [0-7]. Refer to the TRM for details on the mapping of the setting value to specific voltages.
Return Value:	None
Side Effects and Restrictions:	The feedback reference voltage must be greater than the watchdog reference voltage.

4 Power Management

APIs

void CyPmSaveClocks()

Description: This function is called in preparation for entering sleep or hibernate low power modes. Saves all state of the clocking system that doesn't persist during sleep/hibernate or that needs to be altered in preparation for sleep/hibernate. Shuts down all the digital and analog clock dividers. Switches the master clock over to the IMO and shuts down the PLL and MHz Crystal. If the IMO is not currently running it is enabled at 3 MHz before the switch over. The ILO and 32 KHz oscillators are not impacted. The current Flash wait state setting is saved and the Flash wait state setting is set for the current IMO speed.

Parameters: None

Return Value: None

void CyPmRestoreClocks()

Description: Restores any state that was preserved by the last call to CyPmSaveClocks. The Flash wait state setting is also restored.

Parameters: None

Return Value: None

void CyPmAltAct(uint16 wakeupTime, uint16 wakeupSource)

Description: Puts the part into the Alternate Active (Standby) state. Before switching to Alternate Active, if a wakeupTime other than NONE is specified, then the appropriate timer state is configured as specified with the interrupt for that timer disabled. The wakeup source will be the combination of the values specified in the wakeupSource and any timer specified in the wakeupTime argument. Once the wakeup condition is satisfied, then all saved state is restored and the function returns in the Active state.
Note If the previous wakeupTime value is different than the desired wakeupTime value, then the period before the wakeup occurs can be significantly shorter than the specified time. If the next call is made with the same wakeupTime value, then the wakeup will occur the specified period after the previous wakeup occurred.

If a wakeupTime other than NONE is specified, then upon exit the state of the specified timer will be left as specified by wakeupTime with the timer enabled and the interrupt disabled. If the CTW, FTW or One PPS is already configured for wakeup, for example with the SleepTimer or RTC components, then specify NONE for the wakeupTime and include the appropriate source for wakeupSource.
Note PSoC 3 ES2 and PSoC 5 ES1 silicon have a defect that causes connections to several analog resources to be unreliable when the device is placed in a low power mode. See the silicon errata for details. This defect is fixed on production silicon (that is, PSoC 3 ES3).

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

Parameters: wakeupTime: Specifies a timer wakeup source and the frequency of that source.

Value	Define	Time
0	PM_ALT_ACT_TIME_NONE	None
1	PM_ALT_ACT_TIME_ONE_PPS	One PPS: 1 second
2	PM_ALT_ACT_TIME_CTW_2MS	CTW: 2 ms
3	PM_ALT_ACT_TIME_CTW_4MS	CTW: 4 ms
4	PM_ALT_ACT_TIME_CTW_8MS	CTW: 8 ms
5	PM_ALT_ACT_TIME_CTW_16MS	CTW: 16 ms
6	PM_ALT_ACT_TIME_CTW_32MS	CTW: 32 ms
7	PM_ALT_ACT_TIME_CTW_64MS	CTW: 64 ms
8	PM_ALT_ACT_TIME_CTW_128MS	CTW: 128 ms
9	PM_ALT_ACT_TIME_CTW_256MS	CTW: 256 ms
10	PM_ALT_ACT_TIME_CTW_512MS	CTW: 512 ms
11	PM_ALT_ACT_TIME_CTW_1024MS	CTW: 1024 ms
12	PM_ALT_ACT_TIME_CTW_2048MS	CTW: 2048 ms
13	PM_ALT_ACT_TIME_CTW_4096MS	CTW: 4096 ms
14 - 269	PM_ALT_ACT_TIME_FTW(1-256)	FTW: 10 µs to 2.56ms

CyPmAltAct (Continued)

Parameters: PM_ALT_ACT_TIME_FTW() is a macro that takes an argument that specifies how many increments of 10 µs to delay. For PSoC 3 ES2 silicon and earlier and PSoC 5 ES1 silicon and earlier the valid range of values is 1 to 32. For later silicon the valid range of values is 1 to 256.

wakeupSource: Specifies a bitwise mask of wakeup sources. In addition, if a wakeupTime has been specified the associated timer will be included as a wakeup source.

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

Note CTW and One PPS wakeup signals are in the same mask bit. For PSoC 5 ES1 these are also the same address as the FTW (value 1024).

When specifying a Comparator as the wakeupSource, use an instance specific define that will track with the specific comparator for that instance. As an example, for a Comparator instance named "MyComp" the value to OR into the mask is: MyComp_ctComp__CMP_MASK.

Return Value: None

Side Effects and Restrictions: For PSoC 3 ES2 silicon or earlier, the wakeup source is not selectable. In this case the wakeupSource argument is ignored and any of the available wakeup sources will wake the device.
For PSoC 5 ES1 silicon or earlier, the LCD is not a selectable wakeup source.

void CyPmSleep(uint8 wakeupTime, uint16 wakeupSource)

Description: Puts the part into the Sleep state. Before switching to Sleep, if a wakeupTime other than NONE is specified, then the appropriate timer state is configured as specified with the interrupt for that timer disabled. The wakeup source will be the combination of the values specified in the wakeupSource and any timer specified in the wakeupTime argument. Once the wakeup condition is satisfied, then all saved state is restored and the function returns in the Active state.
Note If the previous wakeupTime value is different than the desired wakeupTime value, then the period before the wakeup occurs can be significantly shorter than the specified time. If the next call is made with the same wakeupTime value, then the wakeup will occur the specified period after the previous wakeup occurred.

If a wakeupTime other than NONE is specified, then upon exit the state of the specified timer will be left as specified by wakeupTime with the timer enabled and the interrupt disabled. If the CTW or One PPS is already configured for wakeup, for example with the SleepTimer or RTC components, then specify NONE for the wakeupTime and include the appropriate source for wakeupSource.
Note PSoC 3 ES2 and PSoC 5 ES1 silicon have a defect that causes connections to several analog resources to be unreliable when the device is placed in a low power mode. See the silicon errata for details. This defect is fixed on production silicon (that is, PSoC 3 ES3).

Parameters: wakeupTime: Specifies a timer wakeup source and the frequency of that source.

Value	Define	Time
0	PM_SLEEP_TIME_NONE	None
1	PM_SLEEP_TIME_ONE_PPS	One PPS: 1 second
2	PM_SLEEP_TIME_CTW_2MS	CTW: 2 ms
3	PM_SLEEP_TIME_CTW_4MS	CTW: 4 ms
4	PM_SLEEP_TIME_CTW_8MS	CTW: 8 ms
5	PM_SLEEP_TIME_CTW_16MS	CTW: 16 ms
6	PM_SLEEP_TIME_CTW_32MS	CTW: 32 ms
7	PM_SLEEP_TIME_CTW_64MS	CTW: 64 ms
8	PM_SLEEP_TIME_CTW_128MS	CTW: 128 ms
9	PM_SLEEP_TIME_CTW_256MS	CTW: 256 ms
10	PM_SLEEP_TIME_CTW_512MS	CTW: 512 ms
11	PM_SLEEP_TIME_CTW_1024MS	CTW: 1024 ms
12	PM_SLEEP_TIME_CTW_2048MS	CTW: 2048 ms
13	PM_SLEEP_TIME_CTW_4096MS	CTW: 4096 ms

CyPmSleep (Continued)

Parameters: wakeupSource: Specifies a bitwise mask of wakeup sources. In addition, if a wakeupTime has been specified the associated timer will be included as a wakeup source.

Value	Define	Source
0	PM_SLEEP_SRC_NONE	None
1	PM_SLEEP_SRC_COMPARATOR0	Comparator 0
2	PM_SLEEP_SRC_COMPARATOR1	Comparator 1
4	PM_SLEEP_SRC_COMPARATOR2	Comparator 2
8	PM_SLEEP_SRC_COMPARATOR3	Comparator 3
64	PM_SLEEP_SRC_PICU	PICU
128	PM_SLEEP_SRC_I2C	I2C
512	PM_SLEEP_SRC_BOOSTCONVERTER	Boost Converter
2048*	PM_SLEEP_SRC_CTW	Central Timewheel
2048*	PM_SLEEP_SRC_ONE_PPS	One PPS
4096	PM_SLEEP_SRC_LCD	LCD

Note CTW and One PPS wakeup signals are in the same mask bit. For PSoC 5 ES1 these are in a different bit (value 1024).

When specifying a Comparator as the wakeupSource, use an instance specific define that will track with the specific comparator for that instance. As an example for a Comparator instance named "MyComp" the value to OR into the mask is: MyComp_ctComp__CMP_MASK.

Return Value: None

Side Effects and Restrictions: For PSoC 3 ES2 silicon or earlier, the wakeup source is not selectable. In this case the wakeupSource argument is ignored and any of the available wakeup sources will wake the device.
For PSoC 5 ES1 silicon or earlier, the LCD is not a selectable wakeup source.

void CyPmHibernate()

Description: Puts the device into the Hibernate state. Before switching to Hibernate the port interrupt control unit (PICU) wakeup source bit is saved and then set. The PICU is the only available source to wakeup from the Hibernate state. Once the wakeup condition is satisfied, then the PICU wakeup source bit is restored and the function returns in the Active state.
There is a firmware requirement that specifies applications cannot wakeup from hibernate for 20 µs. The time starts from the wakeup event and gives the sleep regulator time to stabilize. There is no hardware check that this requirement is met. The specified delay should be done on ISR entry.
Note PSoC 3 ES2 and PSoC 5 ES1 silicon have a defect that causes connections to several analog resources to be unreliable when the device is placed in a low power mode. See the silicon errata for details. This defect is fixed on production silicon (that is, PSoC 3 ES3).

Parameters: None

Return Value: None

uint8 CyPmReadStatus(uint8 mask)

Description: Manages the Power Manager Interrupt Status Register. This register has the interrupt status for the one pulse per second, central timewheel and fast timewheel timers. This hardware register clears on read. To allow for only clearing the bits of interest and preserving the other bits, this function uses a shadow register that retains the state. This function reads the status register and ORs that value with the shadow register. That is the value that is returned. Then the bits in the mask that are set are cleared from this value and written back to the shadow register. It is required to call CyPmReadStatus() within 1 ms (1 clock cycle of the ILO) after the CTW event occurred.

Parameters: mask: Bits in the shadow register to clear

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

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

Instance Low Power APIs

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

void `=instance_name`_Sleep (void)

Description: The _Sleep() function checks to see if the component is enabled and saves that state. Then it calls the _Stop() function and calls _SaveConfig() function to save the user configuration.
Call the _Sleep() function before calling the CyPmSleep() or the CyPmHibernate() function. Refer to the PSoC Creator System Reference Guide for more information about power management functions.

Parameters: None

Return Value: None

Side Effects: None

void `=instance_name`_Wakeup (void)

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

Parameters: None

Return Value: None

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

void `=instance_name`_SaveConfig(void)

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

Parameters: None

Return Value: None

Side Effects: None

void `=instance_name`_RestoreConfig(void)

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

Parameters: None

Return Value: None

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

5 Interrupts



See also Interrupt data sheet.

APIs

CyGlobalIntEnable

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

CyGlobalIntDisable

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

uint32 CyDisableInts()

Description: Disables the interrupt enable for each interrupt.

Parameters: None

Return Value: 32 bit mask of interrupts previously enabled

void CyEnableInts(uint32 mask)

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

Parameters: mask: 32 bit mask of interrupts to enable

Return Value: None

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

void CyIntEnable(uint8 number)

Description: Enables the specified interrupt number.

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

Return Value: None

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

void CyIntDisable(uint8 number)

Description: Disables the specified interrupt number.

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

Return Value: None

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

uint8 CyIntGetState(uint8 number)

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

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

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: Valid range [0-31]. Interrupt number

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: Valid range [0-31]. Interrupt number

Return Value: Interrupt vector value

cyisraddress CyIntSetSysVector(uint8 number, cyisraddress address)

Description: This API only applies to PSoC 5 devices. It sets the interrupt vector of the specified system interrupt number. System interrupts are present only for the ARM platform. These interrupts are for SysTick, PendSV and others.

Parameters: number: Valid range [0-15]. Interrupt number

address: Pointer to an interrupt service routine

Return Value: Previous interrupt vector value

cyisraddress CyIntGetSysVector(uint8 number)

Description: This API only applies to PSoC 5 devices. It gets the interrupt vector of the specified system interrupt number. System interrupts are present only for the ARM platform. These interrupts are for SysTick, PendSV and others.

Parameters: number: Valid range [0-15]. Interrupt number

Return Value: Interrupt vector value

void CyIntSetPriority(uint8 number, uint8 priority)

Description: Sets the priority of the specified interrupt number.

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

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

Return Value: None

uint8 CyIntGetPriority(uint8 number)

Description: Gets the priority of the specified interrupt number.

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

Return Value: Interrupt priority

void CyIntSetPending(uint8 number)

Description: Forces the specified interrupt number to be pending.

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

Return Value: None

void CyIntClearPending(uint8 number)

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

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

Return Value: None

6 Cache

PSoC 3 Cache Functionality

The PSoC 3 cache is non-functional and disabled. There are no defines, functions or macros for cache handling for PSoC 3.

PSoC 5 Cache Functionality

void CyFlushCache()

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

Parameters: None

Return Value: None

7 Pins

In addition to the functionality provided for pins as part of the Pins component, a library of pin macros is provided in the *cypins.h* file. These macros all make use of the port pin configuration register that is available for every pin on the device. The address of that register is provided in the *cydevice_trm.h* file. Each these pin configuration registers is named:

```
CYREG_PRTx_PCy
```

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

APIs

uint8 CyPins_ReadPin(uint16/uint32 pinPC)

Description:	Reads the current value on the pin (pin state, PS).
Parameters:	pinPC: Port pin configuration register (uint16 PSoC 3 / uint32 PSoC 5)
Return Value:	Pin state
	0: Logic low value
	Non-0: Logic high value

void CyPins_SetPin(uint16/uint32 pinPC)

Description:	Set the output value for the pin (data register, DR) to a logic high. Note that this only has an effect for pins configured as software pins that are not driven by hardware.
Parameters:	pinPC: Port pin configuration register (uint16 PSoC 3 / uint32 PSoC 5)
Return Value:	None

void CyPins_ClearPin(uint16/uint32 pinPC)

Description:	Clear the output value for the pin (data register, DR) to a logic low. Note that this only has an effect for pins configured as software pins that are not driven by hardware.
Parameters:	pinPC: Port pin configuration register (uint16 PSoC 3 / uint32 PSoC 5)
Return Value:	None

void CyPins_SetPinDriveMode(uint16/uint32 pinPC, uint8 mode)

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

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

mode: Desired drive mode

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

Return Value: None

uint8 CyPins_ReadPinDriveMode(uint16/uint32 pinPC)

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

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

Return Value: Current drive mode for the pin

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

void CyPins_FastSlew(uint16/uint32 pinPC)

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

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

Return Value: None

void CyPins_SlowSlew(uint16/uint32 pinPC)

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

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

Return Value: None

8 Register Access

A library of macros provides read and write access to the registers of the device. These macros are used with the defined values made available in the generated cydevice.h, cydevice_trm.h and cyfitter.h files.

Access to registers should be made using these macros and not the functions that are used to implement these macros. This allows for device independent code generation.

APIs

uint8 CY_GET_REG8(uint16/uint32 reg)

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

Parameters: reg: Register address (uint16 PSoC 3 / uint32 PSoC 5)

Return Value: Read value

void CY_SET_REG8(uint16/uint32 reg, uint8 value)

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

Parameters: reg: Register address (uint16 PSoC 3 / uint32 PSoC 5)

value: Value to write

Return Value: None

uint16 CY_GET_REG16(uint16/uint32 reg)

Description: Reads the 16-bit value from the specified register. The registers are defined to be in little endian order (LSB in lowest address), but the compiler for PSoC 3 operates in big endian order. This macro implements the byte swapping required for proper operation. For PSoC 3 the address must be in the lower 64 K address range.

Parameters: reg: Register address (uint16 PSoC 3 / uint32 PSoC 5)

Return Value: Read value

void CY_SET_REG16(uint16/uint32 reg, uint16 value)

Description: Writes the 16-bit value to the specified register. The registers are defined to be in little endian order (LSB in lowest address), but the compiler for PSoC 3 operates in big endian order. This macro implements the byte swapping required for proper operation. For PSoC 3 the address must be in the lower 64 K address range.

Parameters: reg: Register address (uint16 PSoC 3 / uint32 PSoC 5)

value: Value to write

Return Value: None

uint32 CY_GET_REG24(uint16/uint32 reg)

Description: Reads the 24-bit value from the specified register. The registers are defined to be in little endian order (LSB in lowest address), but the compiler for PSoC 3 operates in big endian order. This macro implements the byte swapping required for proper operation. For PSoC 3 the address must be in the lower 64 K address range.

Parameters: reg: Register address (uint16 PSoC 3 / uint32 PSoC 5)

Return Value: Read value

void CY_SET_REG24(uint16/uint32 reg, uint32 value)

Description: Writes the 24-bit value to the specified register. The registers are defined to be in little endian order (LSB in lowest address), but the compiler for PSoC 3 operates in big endian order. This macro implements the byte swapping required for proper operation. For PSoC 3 the address must be in the lower 64 K address range.

Parameters: reg: Register address (uint16 PSoC 3 / uint32 PSoC 5)

value: Value to write

Return Value: None

uint32 CY_GET_REG32(uint16/uint32 reg)

Description: Reads the 32-bit value from the specified register. The registers are defined to be in little endian order (LSB in lowest address), but the compiler for PSoC 3 operates in big endian order. This macro implements the byte swapping required for proper operation. For PSoC 3 the address must be in the lower 64 K address range.

Parameters: reg: Register address (uint16 PSoC 3 / uint32 PSoC 5)

Return Value: Read value

void CY_SET_REG32(uint16/uint32 reg, uint32 value)

Description: Writes the 32-bit value to the specified register. The registers are defined to be in little endian order (LSB in lowest address), but the compiler for PSoC 3 operates in big endian order. This macro implements the byte swapping required for proper operation. For PSoC 3 the address must be in the lower 64 K address range.

Parameters: reg: Register address (uint16 PSoC 3 / uint32 PSoC 5)

value: Value to write

Return Value: None

uint8 CY_GET_XTND_REG8(uint32 reg)

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

Parameters: reg: Register address

Return Value: Read value

void CY_SET_XTND_REG8(uint32 reg, uint8 value)

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

Parameters: reg: Register address

value: Value to write

Return Value: None

uint16 CY_GET_XTND_REG16(uint32 reg)

Description: Reads the 16-bit value from the specified register. The registers are defined to be in little endian order (LSB in lowest address), but the compiler for PSoC 3 operates in big endian order. This macro implements the byte swapping required for proper operation. Supports the full address space for PSoC 3, but requires more execution cycles than the standard register get function. Identical to CY_GET_REG16 for PSoC 5.

Parameters: reg: Register address

Return Value: Read value

void CY_SET_XTND_REG16(uint32 reg, uint16 value)

Description: Writes the 16-bit value to the specified register. The registers are defined to be in little endian order (LSB in lowest address), but the compiler for PSoC 3 operates in big endian order. This macro implements the byte swapping required for proper operation. Supports the full address space for PSoC 3, but requires more execution cycles than the standard register set function. Identical to CY_SET_REG16 for PSoC 5.

Parameters: reg: Register address

value: Value to write

Return Value: None

uint32 CY_GET_XTND_REG24(uint32 reg)

Description: Reads the 24-bit value from the specified register. The registers are defined to be in little endian order (LSB in lowest address), but the compiler for PSoC 3 operates in big endian order. This macro implements the byte swapping required for proper operation. Supports the full address space for PSoC 3, but requires more execution cycles than the standard register get function. Identical to CY_GET_REG24 for PSoC 5.

Parameters: reg: Register address

Return Value: Read value

void CY_SET_XTND_REG24(uint32 reg, uint32 value)

Description: Writes the 24-bit value to the specified register. The registers are defined to be in little endian order (LSB in lowest address), but the compiler for PSoC 3 operates in big endian order. This macro implements the byte swapping required for proper operation. Supports the full address space for PSoC 3, but requires more execution cycles than the standard register set function. Identical to CY_SET_REG24 for PSoC 5.

Parameters: reg: Register address

value: Value to write

Return Value: None

uint32 CY_GET_XTND_REG32(uint32 reg)

Description: Reads the 32-bit value from the specified register. The registers are defined to be in little endian order (LSB in lowest address), but the compiler for PSoC 3 operates in big endian order. This macro implements the byte swapping required for proper operation. Supports the full address space for PSoC 3, but requires more execution cycles than the standard register get function. Identical to CY_GET_REG32 for PSoC 5.

Parameters: reg: Register address

Return Value: Read value

void CY_SET_XTND_REG32(uint32 reg, uint32 value)

Description: Writes the 32-bit value to the specified register. The registers are defined to be in little endian order (LSB in lowest address), but the compiler for PSoC 3 operates in big endian order. This macro implements the byte swapping required for proper operation. Supports the full address space for PSoC 3, but requires more execution cycles than the standard register set function. Identical to CY_SET_REG32 for PSoC 5.

Parameters: reg: Register address

value: Value to write

Return Value: None

9 DMAC

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

Refer to the DMA component data sheet for more information.

10 Flash and EEPROM

Flash and EEPROM are programmed through the SPC (System Performance Controller) calls. The Flash/EEPROM specific API abstracts this for simplicity.

Some parts have ECC (Error correction codes) storage. This can be configured to be true ECC or used for configuration storage. If it is used for configuration storage the user must be aware when erasing sectors. No API is supplied for erasing a sector but can be created with SPC calls.If the configuration data is erased the hardware will not be configured on the next reset.

An alternative to erasing a sector is programming a row. A ROW (part specific size) of flash or EEPROM can be erased and programmed in one step. The functions provided will deal with configuration data if it exists and allow smaller granularity of erasing.

The caller must first call the CySetTemp and CySetFlashEEBuffer functions. The temperature is needed to adjust the write times to the flash for optimal performance. The Buffer is used to store intermediate data while communicating with the SPC. The SPC is push/pull with a register to send commands to and read data back from.

Flash or EEPROM can be written to one row at a time by calling the same function "CyWriteRowData". The first parameter will determine the flash or EEPROM array. The number of Arrays that are flash and the number of Arrays that are EEPROM are specific to the exact part selected. The following lists the array IDs assigned to Flash and EEPROM.

- 0x00-0x3E : (63) Flash Arrays
- 0x3F : (1) Selects all Flash arrays(for parallel programming)
- 0x40-0x7F : (64) Embedded EEPROM Arrays

Check your part to know which array IDs are valid.

Hardware Configuration Options

A PSoC device can be selected that supports ECC or not. If a device supports ECC, it can be used for true Error Correction Codes or it can be used to store configuration data. If the ECC is being used for configuration data some are may be unused by configuration data and available to the user. Care must be taken not to lose the configuration data. The device will not function as configured by the user with out this data.

APIs

The Flash and EEPROM are programmed using a common set of functions.

cystatus CySetTemp()

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

Parameters: None

Return Value: Status

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

Side Effects and Restrictions: Execution of this function takes an extended period of time.

cystatus CySetFlashEEBuffer(uint8 *buffer)

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

Parameters: buffer: Allocated buffer that is (SIZEOF_FLASH_ROW + SIZEOF_ECC_ROW) bytes. For PSoC 3 and PSoC 5 the buffer is 256 bytes + 32 bytes = 288 bytes.

Return Value: Status

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

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

Description: Allows a row to be erased and programmed. If the array is a flash array and ECC is being used for configuration storage, the function expects that both the row data and the ECC data have been provided as part of rowData.

Parameters: uint8 arrayId – ID of the array that contains the sector to be erased.

uint16 rowAddress – Row Number to erase and then program. 0, 1, 2, …

uint8 * rowData – Address of the data to be programmed. The size of this row will be SIZEOF_FLASH_ROW or SIZEOF_EEPROM_ROW depending on the 'arrayId'.
Note This cannot be the same buffer passed as SPC buffer

uint16 rowSize – Number of bytes of row data

Return Value: CYRET_SUCCESS if successful.
CYRET_LOCKED if the SPC is already in use.
CYRET_UNKNOWN if there was an SPC error.

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

Description: Writes a row of Flash or EEPROM. For Flash the ECC will be written automatically if ECC is enabled. If ECC is disabled the current contents of the ECC memory is maintained.

Parameters: arrayId: ID of the array to write. The type of write, Flash or EEPROM, is determined from the array ID. The arrays in the part are sequential starting at the first ID for the specific memory type.

Type	First ID	Array Size
Flash	FIRST_FLASH_ARRAYID	64K Bytes
EEPROM	FIRST_EE_ARRAYID	2K Bytes

rowAddress: Row address within the specified arrayId.

Type	Rows per Array	Row size in Bytes
Flash	256	SIZEOF_FLASH_ROW (256)
EEPROM	128	SIZEOF_EEPROM_ROW (16)

rowData: Array of bytes to write.

Return Value: Status

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

cystatus CyWriteRowConfig(uint8 arrayId, uint16 rowAddress, uint8 rowData)

Description: Writes the ECC portion of a Flash row. This function is only present when ECC is disabled and is not being used to store configuration data. This function is only valid for Flash array IDs (not for EEPROM).

Parameters: arrayId: ID of the array to write. The arrays in the part are sequential starting at the first ID for the specific memory type.

Type	First ID	ECC Array Size
Flash	FIRST_FLASH_ARRAYID	8 K Bytes

rowAddress: Row address within the specified arrayId.

Type	Rows per Array	ECC Row size in Bytes
Flash	256	SIZEOF_ECC_ROW (32)

rowData: Array of bytes to write.

Return Value: Status

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

void CyFlash_Start()

Description: Enables the Flash. By default Flash is enabled. For ES2 or earlier PSoC 3 and ES1 or earlier PSoC 5, the same bit controls both EEPROM and Flash. Starting or stopping either will cause both to be started or stopped.

Parameters: None

Return Value: None

void CyFlash_Stop()

Description: Disables the Flash. This setting is ignored as long as the CPU is currently running. This will only take effect when the CPU is later disabled. For ES2 or earlier PSoC 3 and ES1 or earlier PSoC 5 the same bit controls both EEPROM and Flash. Starting or stopping either will cause both to be started or stopped.

Parameters: None

Return Value: None

void CyFlash_SetWaitCycles(uint8 freq)

Description: Sets the correct number of wait cycles for the flash based on the frequency of operation of the devices. This function should be called before increasing the clock frequency. It can optionally be called after lowering the clock frequency in order to improve CPU performance.

Parameters: freq: Frequency of operation in Megahertz

Return Value: None

void CyEEPROM_Start()

Description: Enables the EEPROM. For ES2 or earlier PSoC 3 and ES1 or earlier PSoC 5 the same bit controls both EEPROM and Flash. Starting or stopping either will cause both to be started or stopped. Also for those silicon versions the EEPROM is enabled by default. For later silicon the EEPROM is controlled by a separate bit and must be started before it can be used.

Parameters: None

Return Value: None

void CyEEPROM_Stop()

Description: Disables the EEPROM. For ES2 or earlier PSoC 3 and ES1 or earlier PSoC 5 the same bit controls both EEPROM and Flash. Starting or stopping either will cause both to be started or stopped.

Parameters: None

Return Value: None

void CyEEPROM_ReadReserve()

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

Parameters: None

Return Value: None

void CyEEPROM_ReadRelease()

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

Parameters: None

Return Value: None

11 Bootloader System

In PSoC Creator, the bootloader system manages the process of updating the device flash memory with new application code and/or data ("code"). This is accomplished by the following parts:

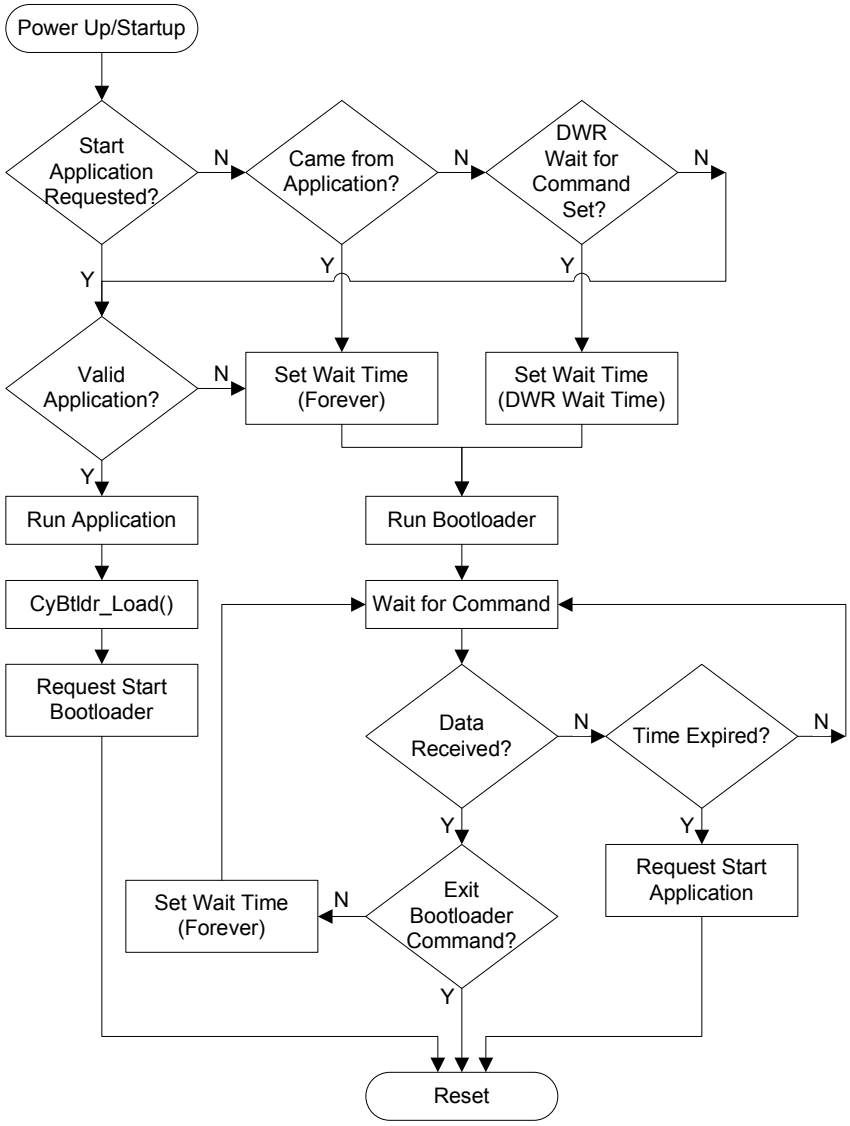
- Bootloader component
- Communications component
- Bootloader project, used to create the bootloader component
- Bootloadable project, used to create the code

The following sections describe the various aspects of the bootloader process in greater detail.

Bootloader Component

The bootloader component allows you to update the device flash memory with new code. The bootloader accepts and executes commands, and passes responses to those commands back to the communications component. The bootloader collects and arranges the received data and manages the actual writing of flash through a simple command/status register interface. The bootloader component is not a typical component. It is not available in the Component Catalog. Instead it is always present, behind the scenes, if you have a bootloader type project.

The following diagram shows how the bootloader works.



Communications Component

The communications component manages the communications protocol to receive commands from an external system, and passes those commands to the bootloader. It also passes command responses from the bootloader back to the off-chip system.

Note USB and I^2C are the only officially supported communication methods for the bootloader. Refer to the USBFS or I^2C component datasheet as needed for more details about the appropriate communication method. There is also a Custom option to add bootloader support to any existing communications component. See "Custom Bootloader Component."

You can also create your own bootloader component for any number of communication methods. For information and instructions on how to do this, refer to the *Component Author Guide*.

Custom Bootloader Component

You can create custom bootloader components using any desired communication component. Under the DWR System Editor "IO Component" select the "Custom_Interface" option. See also the "IO Component" parameter. This allows for implementing the necessary functions in any way necessary in your application code. The following shows an example of code inserted into the *main.c* file for an SPI communication component. For more information about the CyBtldr_ APIs, refer to the *Component Author Guide*.

```
void CyBtldrCommStart(void)
{
    SPIS_1_Start();
}

void CyBtldrCommStop (void)
{
    SPIS_1_Stop();
}

void CyBtldrCommReset(void)
{
}

cystatus CyBtldrCommWrite(uint8* buffer, uint16 size, uint16* count, uint8
timeOut)
{
    uint16 i;
    cystatus status = CYRET_EMPTY;

    uint8 intStatus = CyEnterCriticalSection();

    SPIS_1_ClearRxBuffer();
    SPIS_1_ClearTxBuffer();

    for (i = 0; i < size; i++)
    {
        SPIS_1_WriteTxData(buffer[i]);
    }

    CyExitCriticalSection(intStatus);

    while (timeOut-- > 0)
    {
        if ((SPIS_1_ReadTxStatus() & SPIS_1_STS_SPI_DONE) !=
           SPIS_1_STS_SPI_DONE)
        {
            *count = size;
            status = CYRET_SUCCESS;
            break;
        }

        CyDelay(10);
    }
```

```
    return status;
}

cystatus CyBtldrCommRead (uint8* buffer, uint16 size, uint16* count, uint8
timeOut)
{
    uint16 i = 0;
    cystatus status = CYRET_EMPTY;

    uint8 validData = 0;
    uint8 dataByte;

    while (timeOut-- > 0)
    {
        if (SPIS_1_GetRxBufferSize() > 0 && SPIS_1_GetTxBufferSize() == 0)
        {
            while (!validData && SPIS_1_GetRxBufferSize() > 0)
            {
                dataByte = SPIS_1_ReadRxData();

                validData = (1 == dataByte);
            }

            if (validData)
            {
                buffer[0] = dataByte;
                i = 1;
            }

            CyDelay(10);
            while (SPIS_1_GetRxBufferSize() > 0)
            {
                buffer[i++] = SPIS_1_ReadRxData();
            }

            if (i > 0)
            {
                while(buffer[--i] != 0x17);
                *count = i+1;
                status = CYRET_SUCCESS;
                break;
            }
        }
        CyDelay(10);
    }
    return status;
}
```

Bootloader Project Types

In order to implement both a bootloader component and the code, you must create special PSoC Creator project types: bootloader Project and bootloadable Project.

Bootloader Project

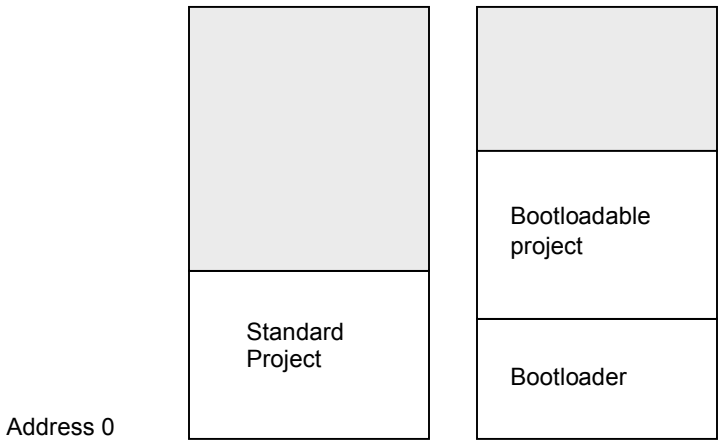
The bootloader component only exists within a PSoC Creator design project of type "bootloader." When you create a bootloader project, a bootloader component automatically exists in the project.

You typically complete a bootloader design project by dragging a communications component onto the schematic, routing the I/O to pins, setting up clocks, etc. A bootloader project with a communications component implements the basic boot loader function of receiving new code and writing it to flash. You can add custom functions to a basic bootloader project by dragging other components onto the schematic or by adding source code.

Bootloadable Project

The bootloadable project is actually the code. It is very similar to a "standard" design project; the project type can easily be changed between the two during the design phase. The main differences are that a bootloadable project is always associated with a bootloader project, and a standard project is never associated with a bootloader project.

A standard project resides in flash starting at address zero. A bootloadable project occupies flash at an address above zero; the associated bootloader project occupies flash starting at address zero, as shown in the following:



Bootloader and Bootloadable Project Functions

The bootloader project performs overall transfer of a bootloadable project, or new code, to the flash via the bootloader project's communications component. After the transfer, the processor is always reset. The bootloader project is also responsible at reset time for testing certain conditions and possibly auto-initiating a transfer if the bootloadable project is non-existent or is corrupt.

At startup, the bootloader code loads configuration bytes for its own configuration. It must also initialize the stack and other resources and peripherals to do the transfer. When the transfer is complete, control is passed to the bootloadable project via a software reset.

The bootloadable project then loads configuration bytes for its own configuration; and re-initializes the stack and other resources and peripherals for its functions. The bootloadable project may call the CyBtldr_Load() function in the bootloader project to initiate a transfer (this results in another software reset).

PSoC Creator Project Output Files

When either project type – bootloader or bootloadable - is built, an output file is created for that project.

In addition, an output file for both projects – a "combination" file – is created when the bootloadable project is build. This file includes both the bootloader and bootloadable projects. This file is typically used to facilitate downloading both projects (via JTAG / SWD) to device flash in a production environment.

For bootloader projects the configuration bytes are always stored in the main flash that is occupied by the bootloader, and never in ECC flash.

Configuration bytes for bootloadable projects may be stored in either main flash or in ECC flash. The format of the bootloadable project output file is such that when the device has ECC bytes which are disabled, transfer operations are executed in less time. This is done by interleaving records in the bootloadable main flash address space with records in the ECC flash address space. The bootloader takes advantage of this interleaved structure by programming the associated flash row once – the row contains bytes for both main flash and ECC flash.

Each project has its own checksum. The checksums is included in the output files at project build time.

Memory Usage

The following diagram shows the device's flash memory layout.



The bootloader project always occupies the bottom N 256-byte blocks of flash. N is set so that there is enough flash for:

- the vector table for this project, starting at address 0 (PSoC 5 only), and
- the bootloader project configuration bytes, and
- the bootloader project code and data, and
- the checksum for the bootloader portion of flash.

Note that the bootloader project configuration bytes are always stored in main flash, never in ECC flash. The relevant option is removed from the project's .cydwr file.

The bootloader portion of flash is protected; it can only be overwritten by downloading via JTAG / SWD.

The bootloadable project occupies flash starting at the first 256-byte boundary after the bootloader, and includes:

- the vector table for the project (PSoC 5 only), and
- the bootloadable project code and data.

The bootloadable project's configuration bytes may be stored in the same manner as in a standard project, i.e. in either main flash or in ECC flash, per settings in the project's .cydwr file.

The highest 64-byte block of flash is used as a common area for both projects. Various parameters are saved in this block, which may include:

- the entry in flash of the bootloadable project (4 byte address)
- the amount of flash occupied by the bootloadable project (Number of flash rows)
- the checksum for the bootloadable portion of flash (one byte)
- the size of the bootloadable portion of flash (4 bytes, in bytes)

PSoC 3 8051 Details

In the PSoC 3, the only "exception vector" is the 3-byte instruction at address 0, which is executed at processor reset. (The interrupt vectors are not in flash – they are supplied by the Interrupt Controller (IC)). So at reset the 8051 bootloader code simply starts executing from flash address 0.

PSoC 5 ARM Cortex-M3 Details

In the PSoC 5, a table of exception vectors must exist at address 0. (The table is pointed to by the Vector Table Offset Register, at address 0xE000ED08, whose value is set to 0 at reset.) The bootloader code starts immediately after this table.

The table contains the initial stack pointer (SP) value for the bootloader project, and the address of the start of the bootloader project code. It also contains vectors for the exceptions and interrupts to be used by the bootloader.

The bootloadable project also has its own vector table, which contains that project's starting SP value and first instruction address. When the transfer is complete, as part of passing control to the bootloadable project the value in the Vector Table Offset Register is changed to the address of the bootloadable project's table.

Bootloader Parameters

To access the bootloader parameters, open the DWR System Editor and expand the bootloader section of the editor.

Wait for Command

Description:	At reset, if the bootloader detects that the checksum in bootloadable project flash is valid then it may optionally wait for a command to start a transfer operation before jumping to the bootloadable project code.
Settings:	Yes or no that the wait will take place.
Default:	Yes
Modifiable by API or Driver Firmware:	No
Relationship to other parameters	If the selection is "yes" then the Wait for Command Time parameter is editable. If the selection is "no" then that parameter is grayed out. In that case an external system typically is not able to initiate a transfer, however the bootloadable project code can still launch a transfer operation by calling Bootloader_Start().

Wait for Command Time

Description:	At reset, if the bootloader detects that the checksum in bootloadable project flash is valid then it may optionally wait for a command to start a transfer operation before jumping to the bootloadable project code. This parameter is the wait timeout period.
Settings:	1 – 255, in units of 10 msec.
Default:	10, or 100 msec.
Modifiable by API or Driver Firmware:	No
Relationship to other parameters	This parameter is editable only if the Wait for Command parameter is set to "yes", otherwise it is grayed out.

IO Component

Description:	This is the communications component that the bootloader uses to receive commands and send responses. One and only one communications component must be selected. Only two-way communications components are used. For example, a UART must have both RX and TX enabled, and an infrared (IrDA) component could not be used. A design rule check (DRC) exists for the case where no two-way communications component has been placed onto the bootloader project schematic.
Settings:	This property is a list of the available IO communications protocols on the schematic that have bootloader support. In all cases, independent of what is on the schematic, there is also a Custom option available that allows for implementing the bootloader functions directly.
Default:	If no communications component is on the schematic, then the Custom option will be selected. This allows for implementing the communication in any way.
Modifiable by API or Driver Firmware:	No
Relationship to other parameters	None.

Packet Checksum Type

Description:	Provides a couple of options for the type of checksum to use when transferring packets of data between the host and the bootloader.
Settings:	Basic Sum adds up all the bytes and takes a 2's compliment. 16bit CRC uses the CCITT algorithm.
Default:	Basic Sum.
Modifiable by API or Driver Firmware:	No
Relationship to other parameters:	None.

Bootloader API

The bootloader provides a public API call solely for the purpose of launching a transfer operation from bootloadable project code. Once called, a software reset is executed, and then the bootloader takes over the CPU. Bootloadable project code, including interrupt handlers, is not executed.

When a transfer operation starts, resources and peripherals are reconfigured as needed. All other resources and peripherals are disabled. When the transfer operation is complete, the CPU is reset.

void CyBtldr_Load(void)

Description:	Starts a transfer operation. Reconfigures the device per bootloader project configuration.
Parameters:	void
Return Value:	None. The processor is reset when the transfer is complete.
Side Effects:	None

Bootloader Commands

The bootloader supports the following commands. All received bytes that do not start with one of the set of command bytes listed below is discarded with no response generated. All multi byte fields are output LSB-first.

Enter Bootloader

All other commands are ignored until this command is received.

Input

- Command Byte: 0x38
- Data Bytes: NA

Output

- Status/Error Codes:
  - Success
  - Error Command
  - Error Data
  - Error Length
  - Error Checksum
- Data Bytes:
  - 4 bytes - Silicon ID
  - 1 byte - Silicon Rev
  - 3 bytes - Bootloader Version

Get Flash Size

Responds with the first and last available rows in the selected flash array.

Input

- Command Byte: 0x32
- Data Bytes:
  - 1 byte - Flash Array ID

Output

- Status/Error Codes:
  - Success
  - Error Command
  - Error Data
  - Error Length
  - Error Checksum
- Data Bytes:
  - 2 bytes - First available row
  - 2 bytes - Last available row

Program Row

Writes one row of flash data to the device.

Input

- Command Byte: 0x39
- Data Bytes:
  - 1 byte - Flash Array ID
  - 2 bytes - Flash Row Number
  - n bytes - Data to write into the flash row

Output

- Status/Error Codes:
  - Success
  - Error Command
  - Error Data
  - Error Length
  - Error Checksum
  - Error Flash Row
- Data Bytes: NA

Erase Row

Erases the contents of the provided flash row.

Input

- Command Byte: 0x34
- Data Bytes:
  - 1 byte - Flash Array ID
  - 2 bytes - Flash Row Number

Output

- Status/Error Codes:
  - Success
  - Error Command
  - Error Data
  - Error Length
  - Error Checksum
  - Error Flash Row
- Data Bytes: NA

Verify Row

Gets a 1 byte checksum for the contents of the provided row of flash

Input

- Command Byte: 0x3A
- Data Bytes:
  - 1 byte - Flash Array ID
  - 2 bytes - Flash Row Number

Output

- Status/Error Codes:
  - Success
  - Error Command
  - Error Data
  - Error Length
  - Error Checksum
- Data Bytes:
  - 1 byte - Row checksum

Verify Checksum

Gets a 1-byte value indicating whether the checksum for the flash matches the expected checksum value. A return value of 1 indicates that the checksums match and that the application is considered good. A return value of 0 indicates that the checksums do not match, and that the application is invalid. This is the same check that the bootloader will run before attempting to run the application code.

Input

- Command Byte: 0x31
- Data Bytes: N/A

Output

- Status/Error Codes:
  - Success
  - Error Command
  - Error Data
  - Error Length
  - Error Checksum
- Data Bytes:
  - 1 byte - Application checksum valid

Send Data

Sends a block of data to the device. This data is buffered up in anticipation of another command that will inform the bootloader what to do with the data. If multiple send data commands are issued back-to-back, the data is appended to the previous block. This command is used to breakup large transfers into smaller pieces to prevent bus starvation in some protocols.

Input

- Command Byte: 0x37
- Data Bytes: n bytes - Data to save in the device

Output

- Status/Error Codes:
  - Success
  - Error Command
  - Error Data
  - Error Length
  - Error Checksum
- Data Bytes: NA

Sync bootloader

Resets the bootloader to a clean state, ready to accept a new command. Any data that was buffered will be thrown out. This is only necessary if the host and client get out of sync with each other.

Input

- Command Byte: 0x35
- Data Bytes: NA

Output

- NA - This packet is not acknowledged

Exit Bootloader

This stops the booting process and triggers the device to reset itself.

Input

- Command Byte: 0x3B
- Data Bytes: NA

Output

- NA - This packet is not acknowledged

Bootloader Packets

Packets sent to the bootloader have the following structure:

- 1-byte packet start (0x01)
- 1-byte command
- 2-bytes data length
- n-bytes data
- 2-bytes checksum
- 1-byte packet end (0x17)

Packets output from the bootloader have the following structure:

- 1-byte packet start (0x01)
- 1-byte status/error code
- 2-bytes data length
- n-bytes data
- 2-bytes checksum
- 1-byte packet end (0x17)

Bootloader Application Status/Error Codes

The possible status/error codes output from the bootloader are:

Success

The command was successfully received and executed. Value = 0x00

Error Length (CYRET_ERR_LENGTH)

The amount of data available is outside the expected range. Value = 0x03

Error Data (CYRET_ERR_DATA)

The data is not of the proper form. Value = 0x04

Error Command (CYRET_ERR_CMD)

The command is not recognized. Value = 0x05

Error Checksum (ERR_CHECKSUM)

The checksum does not match the expected value. Value = 0x08

Error Flash Row (ERR_ROW)

The flash row is not valid. Value = 0x0A

Error Unknown (ERR_UNK)

An unknown error occurred. Value = 0x0F

Bootloader Host Status/Error Codes

Error Device (CYRET_ERR_DEVICE)

The expected device does not match the detected device. Value = 0x06

Error Version (CYRET_ERR_VERSION)

The bootloader version detected is not supported. Value = 0x07

Bootloader Application Code & Data File Format

The bootloader application code & data (.cyacd) file format is used to store the bootloadable portion of a design. The file consists of a header followed by lines of flash data. Excluding the header, each line in the .cyacd file represents an entire row of flash data. The data is stored as ASCII data in Big Endian format.

The header record has the format:

```
[4-byte SiliconID][1-byte SiliconRev]
```

The data records have the format:

```
:[1-byte ArrayID][2-byte RowNumber][2-byte DataLength][N-byte Data][1-byte Checksum]
```

The Basic Summation Checksum is computed by summing all bytes (excluding the checksum itself) and then taking the 2's complement.

Bootloader Host Tool

PSoC Creator ships with a bootloader host tool (*bootloader_host.exe*) that can be used to test out the bootloader running on a PSoC chip. The bootloader host tool is the application that communicates with the bootloader itself to send new bootloadable images. The bootloader host tool provided is intended to be used as a development and testing tool only.

Source Code

In addition to the host executable itself, much of the source code used is also provided. This source code can be reused to create your own bootloader host applications. The source code is located in the following directory:

<install directory>\cybootloaderutils

By default, this directory is:

C:\Program Files\Cypress\PSoC Creator\1.0\PSoC Creator\cybootloaderutils

This source code is broken up into four different modules. These modules provide implementations for the various pieces of functionality required for a bootloader host. Depending on the desired level of control, some or all of these modules can be used in developing a custom bootloader host application.

cybtldr_command.c/h

This module handles construction of packets to send to the bootloader, and the parsing of packets received from the bootloader. It has a single function for constructing each type of packet that the bootloader understands, and a single function for parsing the results for each packet the bootloader can send back.

cybtldr_parse.c/h

This module handles the parsing of the *.cyacd file that contains the bootloadable image to send to the device. It has functions for Setting up access to the file, Reading the header, Reading the row data, and closing the file.

cybtldr_api.c/h

Is a row level API that allows for sending a single row of data at a time to the bootloader using a supplied communication mechanism. It has functions for setting up the bootload operation, programming a row, erasing a row, verifying a row, and ending the bootload operation.

cybtldr_api2.c/h

Is a higher level API that handles the entire bootload process. It has functions for programming the device, erasing the device, verifying the device, and aborting the current operation.

Versions

The following are the features provided with different versions of the bootloader host tool:

PSoC Creator 1.0

Initial version; provides APIs for:

- Parsing *.cyacd file
- Constructing packets of data to send to the bootloader
- Functions for Programming, Erasing, Verifying rows of data
- Functions for performing the entire bootload operation

PSoC Creator 1.0, Service Pack 1

Provides all the functionality contained in the beta release. The production version adds support for using either a basic sum or a new 16-bit CCITT checksum for ensuring packet integrity when communicating with the bootloader.

12 System Functions

APIs

uint8 CyEnterCriticalSection(void)

Description: CyEnterCriticalSection disables interrupts and returns a value indicating whether interrupts were previously enabled (the actual value depends on whether the device is PSoC 3 or PSoC 5).

Parameters: None

Return Value: uint8
PSoC 3 – Returns a value containing two bits:
bit 0: 1 if interrupts were enabled before CyEnterCriticalSection was called.
bit 1: 1 if IRQ generation was disabled before CyEnterCriticalSection was called.
PSoC 5 – Returns 1 if interrupts were previously enabled or 0 if interrupts were previously disabled.

Implementation of CyEnterCriticalSection manipulates the IRQ enable bit with interrupts still enabled. The test and set of the interrupt bits is not atomic; this is true for both PSoC3 and PSoC5. Therefore, to avoid corrupting processor state, it must be the policy that all interrupt routines restore the interrupt enable bits as they were found on entry.

void CyExitCriticalSection(uint8 savedIntrStatus)

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

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

Return Value: None

void CYASSERT(uint32 expr)

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

Parameters: expr: Logical expression. Asserts if false.

Return Value: None

void CyHalt(uint8 reason)

Description: Halts the CPU.

Parameters: reason: Value to be used during debugging.

Return Value: None

void CySoftwareReset(void)

Description: Forces a software reset of the device. During the reset the startup code will detect that the reset was the result of a software reset. In this case the SRAM memory area indicated by the arguments will not be cleared. If any of this area has initialization assignments that initialization will still occur.

Parameters: None

Return Value: None

void CyDelay(uint32 milliseconds)

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

Parameters: milliseconds: Number of milliseconds to delay.

Return Value: None

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

Note The CyDelay functions implement simple software-based delay loops. The loops are designed to compensate for bus clock frequency and some other factors, but there are additional factors that may also influence the actual time spent in the loop.

Function entry and exit, and other overhead factors, may also affect the total amount of time spent executing the function. This may be especially apparent when the nominal delay time is small.

void CyDelayUs(uint16 microseconds)

Description:	Delay by the specified number of microseconds. By default the number of cycles to delay is calculated based on the clock configuration entered in PSoC Creator. If the clock configuration is changed at run-time, then the function CyDelayFreq is used to indicate the new Bus Clock frequency. CyDelayUs is used by several components, so changing the clock frequency without updating the frequency setting for the delay can cause those components to fail.
Parameters:	microseconds: Number of microseconds to delay.
Return Value:	Void
Side Effects and Restrictions:	CyDelayUS has been implemented with the instruction cache assumed enabled. When instruction cache is disabled on PSoC 5, CyDelayUs will be two times larger. For example, with instruction cache disabled CyDelayUs(100) would result in about 200 us delay instead of 100 us.

void CyDelayFreq(uint32 freq)

Description:	Sets the Bus Clock frequency used to calculate the number of cycles needed to implement a delay with CyDelay. By default the frequency used is based on the value determined by PSoC Creator at build time.
Parameters:	freq: Bus clock frequency in Hz. 0: Use the default value non-0: Set frequency value
Return Value:	None

void CyDelayCycles(uint32 cycles)

Description:	Delay by the specified number of cycles using a software delay loop.
Parameters:	cycles: Number of cycles to delay.
Return Value:	None

13 Startup and Linking

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

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

PSoC 3

Startup is all handled by a single assembly file (KeilStart.a51) which is based on a template provided by Keil. There isn't a file specifically associated with linking.

PSoC 5

The startup and linker scripts for PSoC 5 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.

GCC Implementation

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

Realview Implementation (applicable for MDK and RVDS)

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

- Support for RTOS and user replacement of routines. This is possible because the library routines are denoted as "weak" allowing their replacement if another implementation is provided.
- A mechanism is provided that allows for the replacement of the provided linker/scatter file with a user version. This is implemented by allowing the user to create the file local to their project and

having a build setting that allows the specification of this file as the linker/scatter file instead of the file provided automatically.

- The Heap and Stack sizes are set by default (4 K Stack, 1 K Heap), but can be modified in the DWR System Editor.
- 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 (PSoC 5 only)

Cortex Microcontroller Software Interface Standard (CMSIS) is a standard from ARM for interacting with Cortex M-series processors. There are multiple levels of support. The following support is provided:

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

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

Preservation of Reset Status

The value of the reset status register shall be read and cleared any time the device is booted and that value shall be saved to a global SRAM variable. This register is RESET_SR0 for PSoC 3 and PSoC 5. That variable along with defines for the fields in the register shall be provided.

uint8 CyResetStatus

Name	Description
CY_RESET_LVID	Low voltage detect digital
CY_RESET_LVIA	Low voltage detect analog
CY_RESET_HVIA	High voltage detect analog
CY_RESET_WD	Watchdog reset
CY_RESET_SW	Software reset
CY_RESET_GP0	General purpose bit 0
CY_RESET_GP1	General purpose bit 1

14 Watchdog Timer

APIs

void CyWdtStart(uint8 ticks, uint8 lpMode)

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

Once enabled the watchdog cannot be disabled. The watchdog must be cleared using the CyWdtClear() function before the watch dog timer rolls over three times. Failing to clear the watchdog timer results in a software reset.

Parameters: ticks: One of the four available timer periods

Value	Define	Time
0	CYWDT_2_TICKS	2 CTW Ticks
1	CYWDT_16_TICKS	16 CTW Ticks
2	CYWDT_128_TICKS	128 CTW Ticks
3	CYWDT_1024_TICKS	1024 CTW Ticks

lpMode: Low power mode configuration

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

Return Value: None

Side effects and restrictions The 'lpMode' parameter will not work on PSoC 3 ES2, PSoC 5 ES1, or earlier versions of these silicon. It will work on PSoC 3 ES3 and newer silicon.

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

void CyWdtClear()

Description: Clears (feeds) the watchdog timer.

Parameters: None

Return Value: None

15 cy_boot Component Changes

Version 2.21

This section lists and describes the major changes in the cy_boot component version 2.21:

Description of Version 2.21 Changes	Reason for Changes / Impact
Added support for PSoC 3 parts with 12 K SRAM.	
Provide a new option for selecting how to compute checksums on data transferred from the bootloader host to the bootloader.	Provide a more robust way to check for errors during IO transfers.
Provide a generic option to allow users to define their own custom bootloader communication functions.	Make adding support for additional communication protocols (SPI, UART, ...) easier. Also provides a means of supporting multiple communications components concurrently in the same design.
Updated a few Power Management functions to prevent some possible issues.	Some parentheses were missing which could cause items to be evaluated in the wrong order.
Added a variable CyResetStatus that can be used to get the information from the RESET_SR0 register.	This is provided because many of the fields contained within the RESET_SR0 register are clear-on-read. Since the bootloader needs to access this register as part of its operation, it prevents the actual application code from accessing the values. The variable is provided so that the application can still get access to all the information.
Added a workaround for some PSoC3 devices to ensure that the NVL values have been properly initialized.	On some PSoC 3 devices the NVL information may not be initialized properly. This workaround is provided to ensure that the NVLs are properly loaded before performing any of the startup code.

Version 2.20

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

Description of Version 2.20 Changes	Reason for Changes / Impact
Updated the CyDelayCycles function to work with instruction cache enabled.	The original CyDelayCycles function was designed to be used with the instruction cache disabled. For PSoC 3 ES3 and Production silicon, when the instruction cache is enabled, the length of the delay was no longer correct.

Description of Version 2.20 Changes	Reason for Changes / Impact
Updated comments in the code and descriptions in this guide for low power mode functions.	The comments and descriptions for the CyPmSleep(), CyPmHibernate(), and CyPmAltAct() functions we clarified to refer to the silicon errata for a PSoC 3 ES2 and PSoC 5 ES1 defect.
Fixed a bootloader issue with the CyBtldr_ValidateApp function.	Fixed an issue in the bootloader for PSoC 5 that would cause it to fail to verify the application code if the application code was larger than 64 K.
Addressed a bootloader wait for traffic issue.	Modified what is required for the bootloader to consider that there is activity and that someone appears to be attempting to communicate. Previously, if the Bootloader received any data over the selected communications component it would then wait forever for data. With this change, the bootloader will now only wait forever if it has received the Enter Bootloader command.
Updated the CySetTemp function to read the die temperature twice to make sure it has a stable value.	Change was made to address an issue. There is no impact.

Version 2.10

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

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

Version 2.0

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

Description of Version 2.0 Changes	Reason for Changes / Impact
Keil C51 keywords available as macros	These macros allow code to specify the memory space of variables and pointers while maintaining compatibility with other compilers. The most commonly used keywords are CYCODE, CYXDATA, and CYDATA, which represent C51's code, xdata, and data keywords, respectively. On other compilers, these macros are ignored.
CyLib APIs are no longer conditional	It is no longer necessary to define macros such as CYLIB_POWER_MANAGEMENT to include API functions from CyLib.
Add separate section for .dma_init in RealView linker script	This prevents potential errors and warnings when building projects in RealView with DMA-based configuration enabled.

Description of Version 2.0 Changes	Reason for Changes / Impact
Initialize complete SRAM interrupt vector table	In previous versions, only the first 32 entries of the table were being initialized.
Initialize reserved interrupt vectors to default interrupt handler	In previous versions, the reserved vectors were initialized to 0, which is not a valid interrupt service routine address.
Fix incorrect endian in CyIntSetVector return value on PSoC 3	In previous versions, the high and low bytes were swapped.
Apply interrupt attribute to interrupt service routine declarations	Interrupt service routines declared with CY_ISR/CY_ISR_PROTO/cyisraddress now have the interrupt attribute, which causes the compiler to emit code to adjust the stack alignment. RealView checks for the interrupt attribute as part of the type, so interrupt service routines or vectors that do not have the interrupt attribute will fail to compile on RealView. For this reason, cy_isr_v1_20 and earlier are not compatible with cy_boot_v1_50 when using RealView. This affects PSoC 5 only; the interrupt attribute was already present in the PSoC 3 version.
Make CY_GET_REGxx/CY_SET_REGxx reentrant and improve performance	The functions that implemented this functionality, CyGetRegXX and CySetRegXX, have been replaced with assembly routines. The new routines may be safely used in interrupt service routines.
PSoC 3 ES3 support in CyDelay	CPUCLK_DIV has been moved from a SFR to the CLKDIST_MSTR1 in ES3.
Use *limits.h* and *ctype.h* to replace some macros in *CyLib.h*	The LONG_MIN, LONG_MAX, and ULONG_MAX macros are now provided by the toolchain-specific limits.h. This prevents warnings caused by differences between the compiler's limits.h and CyLib.h.
Support for RealView in --gnu mode	Corrected some preprocessor conditionals that caused errors when RealView was used with the --gnu option.
Update PSoC 5 startup code to make better use of standard libraries	The startup code used in PSoC 5 projects has been updated to use standard libraries to provide the initialization. This provides for a standard initialization.
Replaced cydevice.h with cydevice_trm.h	cydevice.h has been marked obsolete, so the APIs and generated code provided with PSoC Creator should not use it.
CYCODE, CYDATA, CYXDATA and CYFAR defines are copied over to cytypes.h	Now any component can make use of these defines, Ex Flash component
Fix for the cache flash cycles incorrect behavior after sleep.	CYREG_CACHE_CR doesnt retain its value in PSOC 3 after the sleep

Description of Version 2.0 Changes	Reason for Changes / Impact
Miscellaneous Flash/EEPROM API changes.	Define active/standby power configuration registers based on silicon. Define active/standby power configuration register constant based on silicon. Added CyFlash_Start() and CyFlash_Stop() APIs. Removed CyFlashEEActivePower() and CyFlashEEStandbyPower() APIs, Modified CySetFlashEEBuffer()for PSoC 3 so as not to test whether the buffer is not a pointer at 0. Changed an argument in CyWriteRowConfig() API to rowData. Added CyEEPROM_Start(), CyEEPROM_Stop(), CyEEPROM_ReadReserve(), CyEEPROM_ReadRelease(), and CyFlash_SetWaitCycles(uint8 freq) APIs. Power mode registers and power mode register constants are defined based on the silicon. Cache Control register is defined based on the silicon.
Added reentrancy support.	Appropriate APIs in cylib.c/h, cyflash.c/h, cydmac.c/h and cyspc.c/h files were updated to include reentrant support.
Converted cymemset and cymemcpy to macros, removed cymemmove.	The memset/memcpy/memmove provided by the toolchain vendor are typically faster than the generic implementation because they are designed for the specific target platform. Including both the vendor-provided functions and the generic implementation wastes code space. Non-legacy code should use memset and memcpy directly.

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