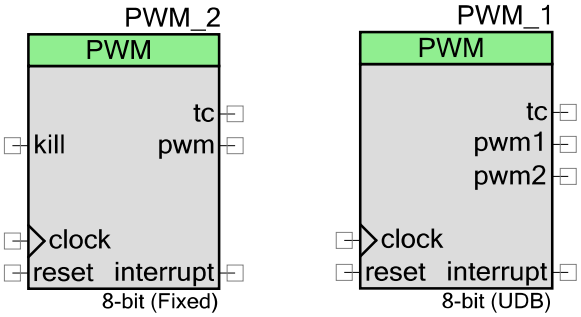
Pulse Width Modulator (PWM)
2.10



Features

- 8- or 16-bit resolution
- Multiple pulse width output modes
- Configurable trigger
- Configurable capture
- Configurable hardware/software enable
- Configurable dead band
- Multiple configurable kill modes
- Customized configuration tool

General Description

The PWM component provides compare outputs to generate single or continuous timing and control signals in hardware. The PWM is designed to provide an easy method of generating complex real-time events accurately with minimal CPU intervention. PWM features may be combined with other analog and digital components to create custom peripherals.

The PWM generates up to two left- or right-aligned PWM outputs or one center-aligned or dual-edged PWM output. The PWM outputs are double buffered to avoid glitches caused by duty cycle changes while running. Left-aligned PWMs are used for most general-purpose PWM uses. Right-aligned PWMs are typically only used in special cases that require alignment opposite of left-aligned PWMs. Center-aligned PWMs are most often used in AC motor control to maintain phase alignment. Dual-edged PWMs are optimized for power conversion where phase alignment must be adjusted.

The optional dead band provides complementary outputs with adjustable dead time where both outputs are low between each transition. The complementary outputs and dead time are most often used to drive power devices in half-bridge configurations to avoid shoot-through currents and resulting damage. A kill input is also available that immediately disables the dead band outputs when enabled. Three kill modes are available to support multiple use scenarios.

Two hardware dither modes are provided to increase PWM flexibility. The first dither mode increases effective resolution by two bits when resources or clock frequency preclude a standard implementation in the PWM counter. The second dither mode uses a digital input to select one of

the two PWM outputs on a cycle-by-cycle basis; this mode is typically used to provide fast transient response in power converts.

The trigger and reset inputs allow the PWM to be synchronized with other internal or external hardware. The optional trigger input is configurable so that a rising edge starts the PWM. A rising edge on the reset input causes the PWM counter to reset its count as if the terminal count was reached. The enable input provides hardware enable to gate PWM operation based on a hardware signal.

An interrupt can be programmed to be generated under any combination of the following conditions: when the PWM reaches the terminal count or when a compare output goes high.

When to Use a PWM

The most common use of the PWM is to generate periodic waveforms with adjustable duty cycles. The PWM also provides optimized features for power control, motor control, switching regulators, and lighting control. You can also use the PWM as a clock divider by driving a clock into the clock input and using the terminal count or a PWM output as the divided clock output.

PWMs, timers, and counters share many capabilities, but each provides specific capabilities. A Counter component is better used in situations that require the counting of a number of events but also provides rising edge capture input as well as a compare output. A Timer component is better used in situations focused on timing the length of events, measuring the interval of multiple rising and/or falling edges, or for multiple capture events.

Input/Output Connections

This section describes the various input and output connections for the Counter. Some I/Os may be hidden on the symbol under the conditions listed in the description of that I/O.

Note All signals are active high unless otherwise specified.

Input	May Be Hidden	Description
clock	N	The clock input defines the signal to count. The counter is incremented or decremented on each rising edge of the clock.
reset	N	Resets the period counter to **Period** and continues normal operation. **Note** While in Reset, the pwm, pwm1, or pwm2 outputs are disabled (driven to ‘0’). For the fixed-function implementation, the pwm output is driven to ‘1’ during reset. A schematic fix for this is provided under Reset in Fixed-Function Block in the Functional Description section. For PSoC 3 ES2 silicon, the Terminal Count pin for the fixed-function PWM is held HIGH in Reset. The schematic fix provided for the pwm output can also be used for the Terminal Count. For Production PSoC 3 or later silicon, the Terminal Count pin for the fixed-function PWM is held low in reset

Input	May Be Hidden	Description
enable	Y	The enable input works in conjunction with software enable and trigger input (if the trigger input is enabled) to enable the period counter. The enable input is not visible if the **Enable Mode** parameter is set to **Software Only**. This input is not available when the fixed-function PWM implementation is chosen.
kill	Y	The kill input disables the PWM outputs. There are several kill modes available, all of which rely on this input to implement the final kill of the output signals. If dead band is implemented, only the dead band outputs (ph1 and ph2) are disabled and the pwm, pwm1, and pwm2 outputs are not disabled. The kill input is not visible if the **Kill Mode** parameter is set to **Disabled**. If you choose the fixed-function PWM implementation and dead band is enabled, kill will kill only the dead band outputs. For PSoC 5, when dead band is disabled, kill will not kill the comparator output. For PSoC 3, when dead band is disabled, kill will kill the comparator output.
cmp_sel	Y	The cmp_sel input selects either the pwm1 or pwm2 output as the final output to the pwm terminal. When the input is '0' (low), the pwm output is pwm1 and when the input is '1' (high), the pwm output is pwm2, as shown in the configuration tool waveform viewer. The cmp_sel input is visible when the **PWM Mode** parameter is set to **Hardware Select**.
capture	Y	The capture input forces the period counter value into the read FIFO. There are several modes defined for this input in the **Capture Mode** parameter. The capture input is not visible if the **Capture Mode** parameter is set to **None**. When the fixed-function PWM implementation is chosen, the capture input is always rising-edge sensitive.
trigger	Y	The trigger input enables the operation of the PWM. The functionality of this input is defined by the **Trigger Mode** and **Run Mode** parameters. After the PWM_Start() API command, the PWM is enabled but the counter does not decrement until the trigger condition has occurred. For the UDB implementation of the PWM, the trigger input is registered with the input clock and so the counter starts one clock after the trigger input is asserted. The trigger condition is set with the **Trigger Mode** parameter. The trigger input is not visible if the parameter is set to **None**.

Output	May Be Hidden	Description
tc	N	The terminal count output is '1' when the period counter is equal to zero. In normal operation this output is '1' for a single cycle where the counter is reloaded with period. If the PWM is stopped with the period counter equal to zero then this signal remains high until the period counter is no longer zero. This output is synchronized to the block clock input of the component.
interrupt	Y	The interrupt output is the logical OR of the group of possible interrupt sources. This signal goes high while any of the enabled interrupt sources are true. The interrupt output remains asserted until the software reads out the status register. In order to receive subsequent interrupts, the interrupt is cleared by reading the status register using the PWM_ReadStatusRegister() API. The interrupt output is only visible if the **Use Interrupt** parameter is set. This allows the status register to be removed for resource optimization as necessary.

Output	May Be Hidden	Description
pwm/pwm1	Y	The pwm or pwm1 output is the first or only pulse-width modulated output. This signal is defined by **PWM Mode**, compare modes, and compare values, as indicated in waveforms in the **Configure** dialog. When the instance is configured in **One Output**, **Dual Edge**, **Hardware Select**, **Center Align**, or **Dither** PWM modes, then the output “pwm” is visible. Otherwise, the output “pwm1” is visible with “pwm2” the other pulse-width signal. This output is synchronized to the block clock input of the component.
pwm2	Y	The pwm2 output is the second pulse width modulated output. The pwm2 output is only visible when **PWM Mode** is set to **Two Outputs**. This output is synchronized to the block clock input of the component.
ph1/ph2	Y	The ph1 and ph2 outputs are the dead band phase outputs of the PWM. In all modes where only the pwm output is visible these are the phased outputs of the pwm signal, which is also visible. In **Two Outputs** mode, these signals are the phased outputs of the pwm1 signal only. Both of these outputs are visible if dead band is enabled in 2 to 4 or 2 to 256 modes and are not visible if dead band is disabled. This output is synchronized to the block clock input of the component.

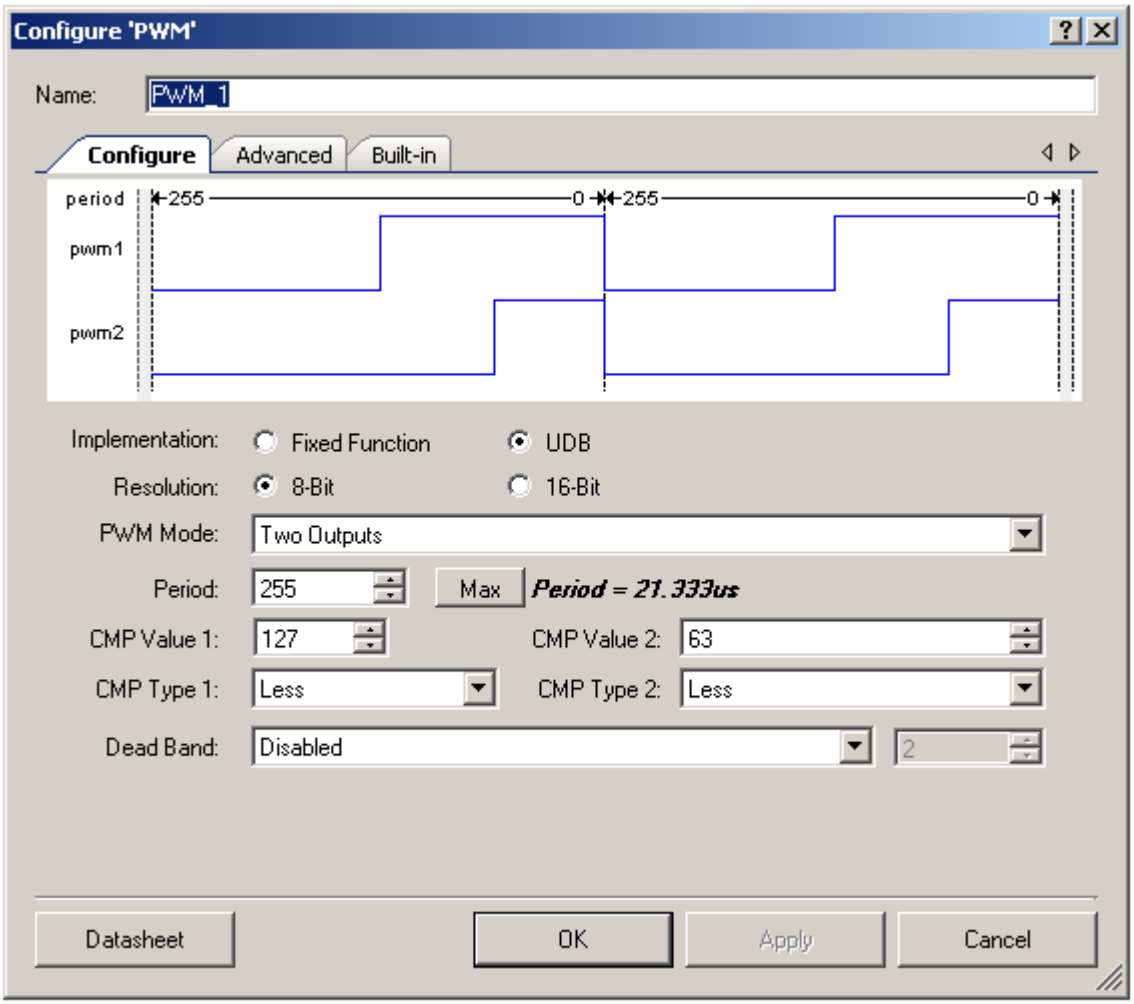
Component Parameters

Drag a PWM component onto your design and double click it to open the **Configure** dialog. The **Configure PWM** dialog contains two tabs: **Configure** and **Advanced**.

Hardware versus Software Configuration Options

Hardware configuration options change the way the project is synthesized and placed in the hardware. You must rebuild the hardware if you make changes to any of these options. Software configuration options do not affect synthesis or placement. When setting these parameters before build time you are setting their initial value, which may be modified at any time with the APIs provided. Most parameters described in the next sections are hardware options. The software options are noted as such.

Configure Tab



Implementation

This parameter allows you to choose between a **Fixed Function** and a **UDB** implementation of the PWM. If this parameter is set to **Fixed Function**, the PWM is implemented in a fixed function block with the associated limitations of that block.

Resolution

The **Resolution** parameter defines the bit-width resolution of the period counter.

Resolution	Maximum Period Count Values
8 (default)	255
16	65,535

Note If **PWM Mode** is set to **Center Align**, the component requires counting up to the period value and then back down to zero, doubling the period of the PWM. In this mode, the limit for an 8-bit PWM is 254 cycles (x2 = 508 cycles) and 65,534 (x2 = 131,068 cycles) for a 16-bit PWM.

PWM Mode

The **PWM Mode** parameter defines the overall functionality of the PWM. It is disabled if **Implementation** is set to **Fixed Function**.

This parameter has a tremendous influence on the visible pins of the symbol as well as the functionality of the pwm, pwm1, and pwm2 outputs as depicted in the waveforms shown in the configuration tool. Options include:

- **One Output** – Only a single PWM output. In this mode the pwm output is visible
- **Two Output** – Two individually configurable PWM outputs. In this mode the pwm1 and pwm2 outputs are visible
- **Dual Edge** – A single dual-edged output created by ANDing together the pwm1 and pwm2 signals. In this mode the pwm output is visible.
- **Center Align** – A single center-aligned output created by having the counter count up to the period value and back down to zero, while creating one center-aligned pulse width based on the compare value. In this mode the pwm output is visible.
- **Dither** – A single output selected from the two internal pwm signals (pwm1 and pwm2) by a hardware state machine included in the pwm hardware implementation. You select between a 0.00, 0.25, 0.50 or 0.75 bit increase in the output pulse width and the hardware controls the selection between the two pwm signals to make this happen. In this case the compare values are set to compare and compare+1. In this mode the pwm output is visible.
- **Hardware Select** – A single output selected from the two internal pwm signals by a hardware input pin cmp_sel. When cmp_sel is low the pwm1 signal is output on the pwm output pin, when cmp_sel is high the pwm2 signal is output on the pwm output pin. In this mode the pwm output is visible.

Period (Software)

The **Period** parameter defines the initial starting value of the counter and the value any time the terminal count is reached and the PWM mode allows reloading of the period counter.

The PWM is implemented as a down counter counting from the **Period** value to zero. The period must be greater than 1 and is limited on the high side by the resolution of the PWM. For an 8-bit PWM the period value has a maximum of 255. Otherwise the period value has a maximum of 65535. When the PWM mode is configured in Center Aligned mode the PWM counts up from zero to the period value and then back down to zero. The period value in Center Aligned mode is twice as long as all other modes because of this special functionality. The period value may be

changed at any time by the PWM_WritePeriod() API Call. The parameter holds only the initial value written during configuration.

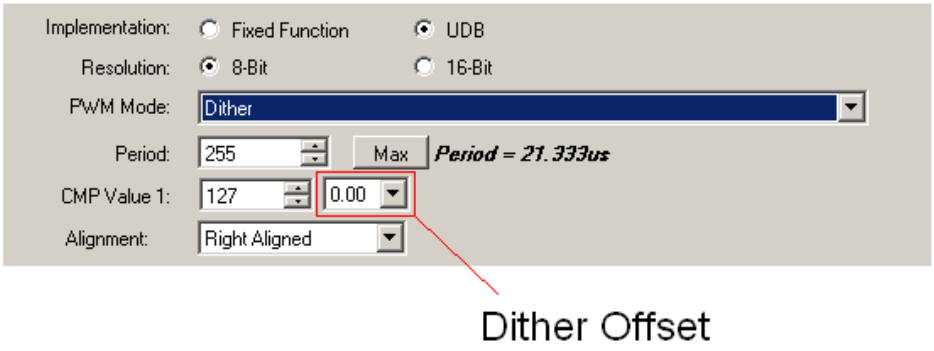
CMP Value 1 / CMP Value2 (Software)

The compare values define the compare output functionality in conjunction with the hardware **Compare Type** options.

The compare values must be greater than 1 and are limited on the high side by the resolution of the PWM. For an 8-bit PWM the compare value has a maximum of 255. Otherwise the compare value has a maximum of 65535. The compare value is also limited by the period. As the period is decreased, the maximum compare values are set to **Period** – 1 to prevent a nonuse compare output. The compare values may be changed at any time by the PWM_WriteCompare1() and PWM_WriteCompare2() API calls. These parameters hold only the initial value written during configuration.

Dither Offset

The **Dither Offset** parameter configures the functionality of the pwm output when the PWM is configured in **Dither** PWM mode.



Dither implements an internal state machine to choose between pwm1 and pwm2 outputs as the final pwm output. The pwm1 and pwm2 outputs are configured to be one-off period values of each other where pwm1 is true for the compare value and pwm2 is true for the compare value + 1. Options include:

- **DO00** – No Dither. The output is always pwm1.
- **DO25** – 0.25 Dither. The output is pwm1 for three of four period counts, and pwm2 for a single period count.
- **DO50** – 0.50 Dither. The output is pwm1 for two of four period counts, and pwm2 for two of the four period counts.
- **DO75** – 0.75 Dither. The output is pwm1 for one of four period counts, and pwm2 for three of the four period counts.

Alignment

The **Alignment** parameter is available when **PWM Mode** is set to **Dither**. Options include:

- **Right Aligned**
- **Left Aligned**

CMP Type 1 / CMP Type 2 (Software)

The compare value parameters define the two period counter comparisons that make up the PWM outputs. These are implemented differently for each of the PWM modes so they are typically controlled with the configuration tool. Each of the two compare mode parameters can be set independently to one of the following enumerated types. Options include:

- **Less** – Compare output is true if period counter is less than the corresponding compare value.
- **Less or Equal** – Compare output is true if period counter is less than or equal to the corresponding compare value.
- **Greater** – Compare output is true if period counter is greater than the corresponding compare value.
- **Greater or Equal** – Compare output is true if period counter is greater than or equal to the corresponding compare value.
- **Equal** – Compare output is true if period counter is equal to the corresponding compare value.
- **Firmware Control** – The **Firmware Control** implementation provides for a more flexible resource usage model in which the compare mode can be set during run time. The compare modes may be changed at any time by the PWM_WriteCompare1() and PWM_WriteCompare2() API calls. These parameters hold only the initial mode written during configuration. If any implementation other than **Firmware Control** is chosen, the hardware is preconfigured and fixed at that configuration at build time. In this case, the WriteCompare APIs are removed from the compilation and therefore are not available.

Dead Band

The **Dead Band** parameter enables or disables the dead band functionality of the PWM. Dead band modes are slightly different in the fixed-function implementation. If dead band mode is one of the two enabled options then the ph1 and ph2 outputs are visible. Options include:

- **Disabled** – No dead band
- **0-3 Counts** – Dead band is implemented on the pwm or the pwm1 output with a maximum of three counts. This is implemented in PLD logic and does not tie up a datapath for the counter.

- **2-4 Clock Cycles** – Dead band is implemented on the pwm or the pwm1 output with a maximum of four clock cycles.
- **2-256 Clock Cycles** – Dead band is implemented on the pwm or the pwm1 output with a maximum of 256 clock cycles. This is implemented in a datapath for the counter.

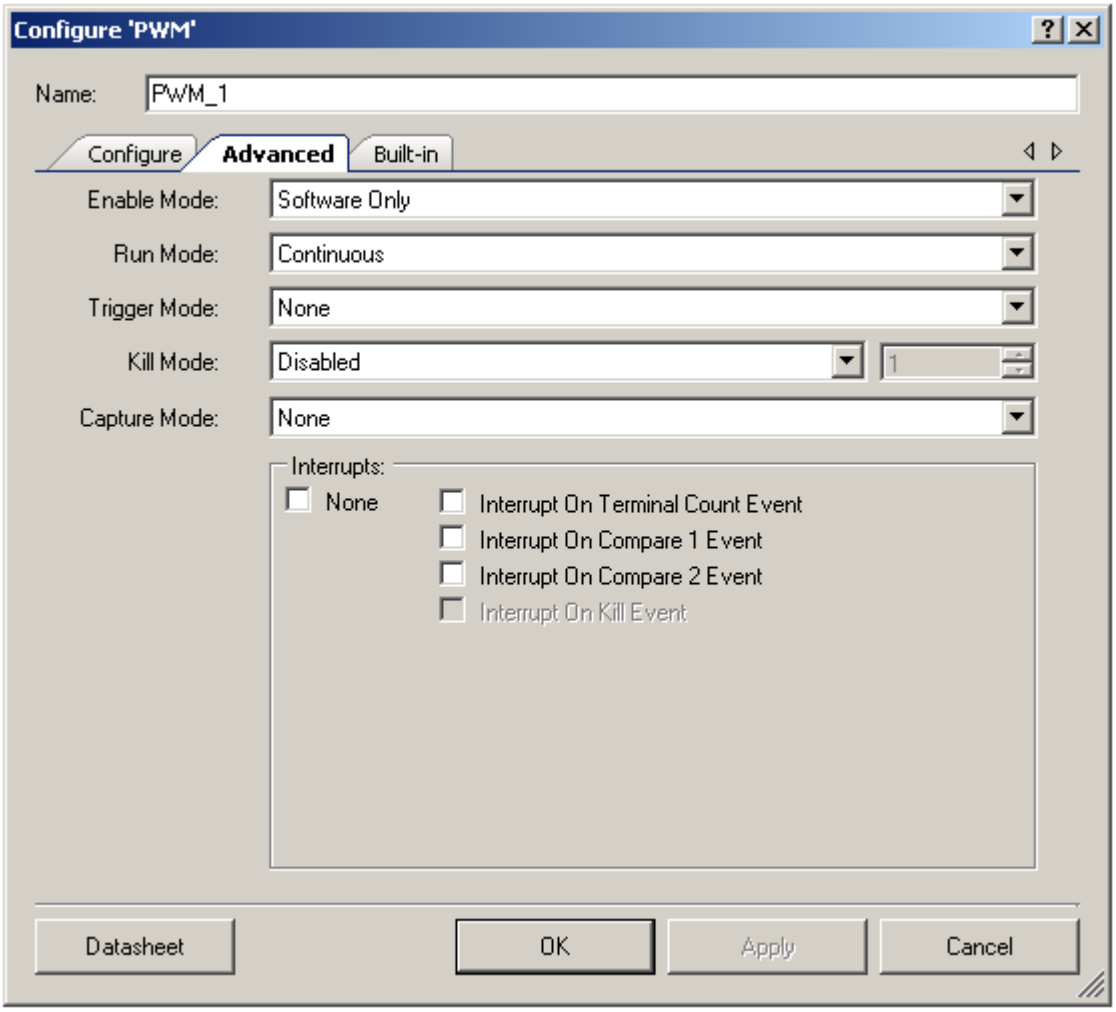
Dead Time (Software)

The dead time value defines the amount of dead time implemented in the dead band output signals ph1 and ph2. This parameter is only valid when **Dead Band** is enabled and is limited based on the hardware configuration option defined in the **Dead Band** parameter.



Dead time is only software configurable when the dead band is enabled with a 2-256 range. This data is controlled with the PWM_WriteDeadTime() and the PWM_ReadDeadTime() API calls. When dead band is enabled with the 2-4 range, the value set in the configuration is built into the hardware and cannot be set using an API.

Advanced Tab



Enable Mode

The **Enable Mode** parameter defines what hardware and software combination is required to enable the overall functionality of the PWM. Options include:

- **Software Only** – The PWM is only enabled when the enable bit in the control register is set by software. The enable input is not visible when the enable mode is set to **Software Only**.
- **Hardware Only** – The PWM is only enabled while the hardware enable input is active (high). In this mode, the PWM_Start() API must be called for proper initialization of the component, to avoid unexpected behavior.
- **Hardware And Software –** The PWM is enabled while both the bit in the control register and the hardware input are active (high).

Run Mode

The **Run Mode** parameter defines how the PWM is triggered to start and continue running. The PWM runs depending on the enable inputs, as described by the following enumerated values.

- **Continuous** – The PWM runs forever on a trigger event.
- **One Shot with Single Trigger** – The PWM runs once on a trigger event
- **One Shot with Multi Trigger** – The PWM runs once on a trigger event. Upon completion of each period the PWM halts until the next trigger event occurs.

Trigger Mode

The trigger mode parameter defines what constitutes a valid trigger event. The trigger input is not visible when Trigger Mode is set to **None**. Options include:

- **None** – No trigger is enabled (trigger is treated as always true)
- **Rising Edge** – A trigger event is signaled on a rising edge of the trigger input.
- **Falling Edge** – A trigger event is signaled on the falling edge of the trigger input.
- **Either Edge** – A trigger event is signal on either a rising edge or a falling edge of the trigger input.

Kill Mode

The **Kill Mode** parameter defines how the hardware handles the pwm outputs when the hardware kill input is active. The kill input is not visible when the kill mode is set to **Disabled**. Options include:

- **Disabled** – No kill is enabled
- **Asynchronous** – The pwm outputs are disabled while kill is active.
- **Single Cycle** – The pwm outputs are disabled while kill is active and are not re-enabled until the end of the period has been reached (that is, tc).
- **Latched** – The pwm outputs are disabled on kill and remain disabled until the PWM is reset.
- **Minimum Time** – The pwm outputs are disabled while kill is active and are not re-enabled until the minimum time has elapsed.

Minimum Kill Time (Software)

The minimum kill time parameter defines the minimum length to be a valid kill signal, of the kill signal necessary when the **Kill Mode** parameter is set to **Minimum Time**.



Minimum Kill Time

The minimum kill time value is controlled with the PWM_WriteKillTime() and PWM_ReadKillTime() API Calls and are limited to values of 1 to 255.

Capture Mode

The **Capture Mode** parameter defines what hardware event will cause a capture of the period counter value to the read FIFO. It is always possible to read the current counter value (that is, a software capture) by calling the PWM_ReadCounter() API. The capture input is not visible when the capture mode is set to **None**. Options include:

- **None** – No capture is enabled
- **Rising Edge** – A capture event is signaled on a rising edge of the capture input.
- **Falling Edge** – A capture event is signaled on the falling edge of the capture input.
- **Either Edge** – A capture event is signaled on either a rising edge or a falling edge of the capture input.

Interrupts

The **Interrupts** parameters allow you to configure the initial interrupt sources. These values are ORed with any of the other interrupt parameters to give a final group of events that can trigger an interrupt. The software can reconfigure this mode at any time, as long as **Interrupts** is not set to **None**. This parameter defines an initial configuration.

- **None** – No interrupts are set.
- **Interrupt On Terminal Count Event** – This option is always available; it is deselected by default.
- **Interrupt On Compare 1 Event** – This option is deselected by default. It is always shown.
- **Interrupt On Compare 2 Event** – This option is deselected by default. It is only available when **UDB** is selected for **Implementation** and **PWM Mode** is set appropriately.
- **Interrupt On Kill Event** – This option is deselected by default. It is only available when **UDB** is selected for **Implementation** and **PWM Mode** is set appropriately.

Local Parameters (For API usage)

These parameters are used in the APIs and not exposed in the GUI.

- **FixedFunctionUsed** – Defined as a '1' (true) if you have chosen to implement the PWM using the fixed-function block.
- **KillModeMinTime** – Defined as a '1' (true) if you have set the **Kill Mode** as **Minimum Time**. This allows PWM_WriteKillTIme() and PWM_ReadKillTime() functions to be included as necessary.
- **PWMModeCenterAligned** – Defined as '1' (true) if you have set the **PWM Mode** as **Center Aligned.** The PWM_ReadCompare() and PWM_WriteCompare() functions are defined differently for this mode than other modes. This parameter is used to add the correct functions and remove the unnecessary functions.
- **DeadBandUsed** – Defined as '1' (true) if you have chosen to implement dead band with the 2-256 enable mode. This is used to conditionally include PWM_WriteDeadTime() and PWM_ReadDeadTime() API functions.
- **DeadBand2_4** – Defined as '1' (true) if you have chosen to implement dead band with the 2-4 counts range. This is used inside of the PWM_WriteDeadTime() and PWM_ReadDeadTime() functions for the different operations that must happen to handle the DeadTime.
- **UseStatus –** Defined as '1' (true) when the configuration warrants the use of a status register. This allows the status register resource to be removed if it is not necessary in the design.
- **UseControl** – Defined as '1' (true) when the configuration warrants the use of a control register. This allows the control register resource to be removed if it is not necessary in the design.
- **UseOneCompareMode** – Defined as '1' (true) when the configuration requires only a single compare mode API to be available. This allows the API to be removed, as defined by the architecture chosen.

Clock Selection

There is no internal clock in this component. You must attach a clock source.

WARNING When configured to use the fixed-function block in the device, the PWM component has the following restrictions:

- The clock input must be from a local clock that is synchronized to the bus clock or directly sourced from the bus clock (configure the clock type as Existing and the source as BUS_CLK).

- If the frequency of the clock matches the bus clock, then the clock must be a direct connection to the bus clock (again configure the clock type as Existing and the source as BUS_CLK). A local clock with a frequency that matches the bus clock generates an error during the build process.

For UDB-Based Components

If the component allows asynchronous clocks, you may use any clock input frequency within the device's frequency range.

If the component requires synchronization to the bus clock, then when using a routed clock[1] to clock the component, the frequency of the routed clock cannot exceed one half the routed clock's source clock frequency.

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

Placement

Placement of the PWM component is based on the **Implementation** parameter selection for the instance of the component. If set to **Fixed Function**, then the PWM is placed in one of the fixed-function blocks. If set to **UDB**, then the PWM is placed throughout the UDB array and all placement information is provided to the API through the *cyfitter.h* file.

[1] A routed clock is anything that is not a clock symbol directly attached to the clock input.

Resources

Resolution	Digital Blocks					API Memory (Bytes)		Pins (per External I/O)
	Datapaths	Macro cells	Status Registers	Control Registers	Counter7	Flash	RAM	
8 Bits One Output Mode[2]	1	6	1	1	0	226	5	-
8 Bits Two Outputs Mode[2]	1	9	1	1	0	237	5	-
8 Bits Dual Edged Mode[2]	1	8	1	1	0	237	5	-
8 Bits Center Align Mode[2]	1	9	1	1	0	226	5	-
8 Bits HW Select Mode[2]	1	9	1	1	0	237	5	-
8 Bits Dither Mode[2]	1	9	1	1	0	231	5	
16 Bits One Output Mode[2]	2	6	1	1	0	275	5	1/2
PWM8 with Dead Band 2-4[3]	1	12	1	2	0	272	6	+2
PWM16 with Dead Band 2-4[3]	2	12	1	2	0	321	7	
PWM8 with Dead Band 2-256[3]	2	11	1	1	0	272	6	+2
PWM16 with Dead Band 2-256[3]	3	11	1	1	0	308	7	

[2] Configuration 1: The PWM in the corresponding PWM mode and resolution is configured with Software Only Enable mode, Continuous Run Mode, Trigger mode set to None, Kill mode and Capture mode disabled with no dead band and Interrupt on TC.

[3] Configuration 2: 2-4 Dead Band range and 2-256 Dead Band range are mutually exclusive. The PWM is configured for the corresponding resolution and One Output PWM mode with Software Only Enable mode, Trigger mode set to None, Kill mode and Capture mode disabled with Interrupt on TC.

Application Programming Interface

Application Programming Interface (API) routines allow you to configure the component using software. The following table lists and describes the interface to each function. The subsequent sections cover each function in more detail.

By default, PSoC Creator assigns the instance name “PWM_1” to the first instance of a component in a given design. You can the rename the instance to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is “PWM.”

Function	Description
PWM_Start()	Initializes the PWM with default customizer values.
PWM_Stop()	Disables the PWM operation. Clears the enable bit of the control register for either of the software controlled enable modes.
PWM_SetInterruptMode()	Configures the interrupts mask control of the interrupt source status register.
PWM_ReadStatusRegister()	Returns the current state of the status register.
PWM_ReadControlRegister()	Returns the current state of the control register.
PWM_WriteControlRegister()	Sets the bit field of the control register.
PWM_SetCompareMode()	Writes the compare mode for compare output when set to Dither mode, Center Align mode or One Output mode.
PWM_SetCompareMode1()	Writes the compare mode for compare1 output into the control register.
PWM_SetCompareMode2()	Writes the compare mode for compare2 output into the control register.
PWM_ReadCounter()	Reads the current counter value (software capture).
PWM_ReadCapture()	Reads the capture value from the capture FIFO.
PWM_WriteCounter()	Writes a new counter value directly to the counter register. This will be implemented only for that currently running period.
PWM_WritePeriod()	Writes the period value used by the PWM hardware.
PWM_ReadPeriod()	Reads the period value used by the PWM hardware.
PWM_WriteCompare()	Writes the compare value when the instance is defined as Dither mode, Center Align mode or One Output mode.
PWM_ReadCompare()	Reads the compare value when the instance is defined as Dither mode, Center Align mode or One Output mode.
PWM_WriteCompare1()	Writes the compare value for the compare1 output.
PWM_ReadCompare1()	Reads the compare value for the compare1 output.
PWM_WriteCompare2()	Writes the compare value for the compare2 output
PWM_ReadCompare2()	Reads the compare value for the compare2 output.

Function	Description
PWM_WriteDeadTime()	Writes the dead time value used by the hardware in dead band implementation.
PWM_ReadDeadTime()	Reads the dead time value used by the hardware in dead band implementation.
PWM_WriteKillTime()	Writes the kill time value used by the hardware when the kill mode is set as Minimum Time.
PWM_ReadKillTime()	Reads the kill time value used by the hardware when the kill mode is set as Minimum Time.
PWM_ClearFIFO()	Clears all capture data from the capture FIFO.
PWM_Sleep()	Stops and saves the user configuration.
PWM_Wakeup()	Restores and enables the user configuration.
PWM_Init()	Initializes component's parameters to those set in the customizer placed on the schematic.
PWM_Enable()	Enables the PWM block operation.
PWM_SaveConfig()	Saves the current user configuration of the component.
PWM_RestoreConfig()	Restores the current user configuration of the component.

Global Variables

Variable	Description
PWM_initVar	Indicates whether the PWM has been initialized. The variable is initialized to 0 and set to 1 the first time PWM_Start() is called. This allows the component to restart without reinitialization after the first call to the PWM_Start() routine. If reinitialization of the component is required, then the PWM_Init() function can be called before the PWM_Start() or PWM_Enable() function.

void PWM_Start(void)

Description: Initializes the PWM with default customizer values. Enables the PWM operation by setting the enable bit of the control register for either of the software controlled enable modes.

Parameters: None

Return Value: None

Side Effects: Sets the enable bit in the control registers of the PWM. If the **Enable Mode** is set to **Hardware Only**, this has no effect on the PWM. If the **Enable Mode** is set to **Hardware and Software**, then this will only enable the software portion of this mode and the hardware input must also be enabled to finally enable the PWM.

void PWM_Stop(void)

Description: Disables the PWM operation by resetting the seventh bit of the control register for either of the software-controlled enable modes. Disables the fixed-function block that has been chosen.

Parameters: None

Return Value: None

Side Effects: Clears the enable bit in the control register of the PWM. If the **Enable Mode** is set to **Hardware Only**, this function has no effect on the PWM. If the **Enable Mode** is set to **Hardware and Software**, this function will disable the software portion of this mode and the hardware input will have no further effect on the enable of the PWM.

void PWM_SetInterruptMode(uint8 interruptMode)

Description: Configures the interrupts mask control of the interrupt source status register.

Parameters: uint8 interruptMode: Bit field containing the interrupt sources enabled

Return Value: None

Side Effects: None

uint8 PWM_ReadStatusRegister(void)

Description: Returns the current state of the status register.

Parameters: None

Return Value: uint8: Current status register value. The status register bits are:

[7:6] : Unused (0)
[5] : Kill event output
[4] : FIFO not empty
[3] : FIFO full
[2] : Terminal count
[1] : Compare output 2
[0] : Compare output 1

Side Effects: Status register bits may be clear on read.

uint8 PWM_ReadControlRegister(void)

Description: Returns the current state of the control register. This API is available only if the control register is not removed.

Parameters: None

Return Value: uint8: Current control register value

Side Effects: None

void PWM_WriteControlRegister(uint8 control)

Description: Sets the bit field of the control register. This API is available only if the control register is not removed.

Parameters: uint8 control: Control register bit field, The status register bits are:

[7] : PWM Enable

[6] : Reset

[5:3] : Compare Mode 2

[2:0] : Compare Mode 2

Return Value: None

Side Effects: None

void PWM_SetCompareMode(enum comparemode)

Description: Writes the compare mode for compare output when set to Dither mode, Center Align mode or One Output mode.

Parameters: enum comparemode: Compare mode enumerated type

Return Value: None

Side Effects: None

void PWM_SetCompareMode1(enum comparemode)

Description: Writes the compare mode for compare1 output into the control register.

Parameters: enum comparemode: Compare mode enumerated type

Return Value: None

Side Effects: None

void PWM_SetCompareMode2(enum comparemode)

Description: Writes the compare mode for compare2 output into the control register.

Parameters: enum comparemode: Compare mode enumerated type

Return Value: None

Side Effects: None

uint8/16 PWM_ReadCounter(void)

Description: Reads the current counter value (software capture).

Parameters: None

Return Value: uint8/uint16: The current period counter value

Side Effects: None

uint8/16 PWM_ReadCapture(void)

Description: Reads the capture value from the capture FIFO.

Parameters: None

Return Value: uint8/uint16: The current capture value

Side Effects: None

Note FIFOs are cleared after going into low-power mode. You must read any data from the capture FIFO before going into low-power mode, if required.

void PWM_WriteCounter(uint8/16 period)

Description: Writes a new counter value directly to the counter register. This will be implemented for that currently running period and only that period.

Parameters: uint8/uint16 period: The period counter value

Return Value: None

Side Effects: None

void PWM_WritePeriod(uint8/16 period)

Description: Writes the period value used by the PWM hardware.

Parameters: period: uint8 or 16 depending on resolution, the new period value

Return Value: None

Side Effects: None

uint8/16 PWM_ReadPeriod(void)

Description: Reads the period value used by the PWM hardware.

Parameters: None

Return Value: uint8/16: Period value

Side Effects: None

void PWM_WriteCompare(uint8/16 compare)

Description: Writes the compare values for the compare output when the **PWM Mode** parameter is set to **Dither** mode, **Center Aligned** mode, or **One Output** mode.

Parameters: uint8/16: Compare value

Return Value: None

Side Effects: Using the PWM_WriteCompare() API when the PWM is running and is functional might change the PWM output immediately, depending on the compare mode and compare type specified. The change in the compare output happens as soon as the new compare value is programmed to the compare register. A change in the PWM output also triggers dead band logic if dead band is enabled.

uint8/16 PWM_ReadCompare(void)

Description: Reads the compare value for the compare output when the **PWM Mode** parameter is set to **Dither** mode, **Center Aligned** mode, or **One Output** mode.

Parameters: None

Return Value: uint8/uint16: Current compare value

Side Effects: This function is only available if the **PWM Mode** parameter is set to one of the modes described above. Otherwise the ReadCompare1/2 functions must be called.

void PWM_WriteCompare1(uint8/16 compare)

Description: Writes the compare value for the compare1 output.

Parameters: uint8/uint16: New compare value for pwm1

Return Value: None

Side Effects: None

uint8/16 PWM_ReadCompare1(void)

Description: Reads the compare value for the compare1 output.

Parameters: None

Return Value: uint8/uint16: Current compare value 1

Side Effects: None

void PWM_WriteCompare2(uint8/16 compare)

Description: Writes the compare value for the compare2 output.

Parameters: uint8/uint16: New compare value for pwm2

Return Value: None

Side Effects: None

uint8/16 PWM_ReadCompare2(void)

Description: Reads the compare value for the compare2 output.

Parameters: None

Return Value: uint8/uint16: The current compare value

Side Effects: None

void PWM_WriteDeadTime(uint8 deadband)

Description: Writes the dead time value used by the hardware in dead band implementation.

Parameters: uint8: Dead Band counts

Return Value: None

Side Effects: None

uint8 PWM_ReadDeadTime(void)

Description: Reads the dead time value used by the hardware in dead band implementation.

Parameters: None

Return Value: uint8: The current setting of Dead Band counts

Side Effects: None

void PWM_WriteKillTime(uint8 killtime)

Description: Writes the kill time value used by the hardware when the **Kill Mode** is set to **Minimum Time**.

Parameters: uint8: Minimum Time kill counts

Return Value: None

Side Effects: None

uint8 PWM_ReadKillTime(void)

Description: Reads the kill time value used by the hardware when the **Kill Mode** is set to **Minimum Time**.

Parameters: None

Return Value: uint8: The current Minimum Time kill counts

Side Effects: None

void PWM_ClearFIFO(void)

Description: Clears the capture FIFO of any previously captured data. Here PWM_ReadCapture() is called until the FIFO is empty.

Parameters: None

Return Value: None

Side Effects: None

void PWM_Sleep(void)

Description: Stops and saves the user configuration.

Parameters: None

Return Value: None

Side Effects: None

void PWM_Wakeup(void)

Description: Restores and enables the user configuration.

Parameters: None

Return Value: None

Side Effects: None

void PWM_Init(void)

Description: Initializes component's parameters to those set in the customizer placed on the schematic. The compare modes are set by setting the respective bits of the control register. The interrupts are chosen as the output from the status register. If you are using fixed-function mode, the chosen fixed-function block is enabled. FIFO is cleared to enable FIFO full bit to be set in the status register. Usually called in PWM_Start().

Parameters: None

Return Value: None

Side Effects: All registers will be reset to their initial values. This reinitializes the component

void PWM_Enable(void)

Description: Enables the PWM block operation by setting the seventh bit of the control register.

Parameters: None

Return Value: None

Side Effects: None

void PWM_SaveConfig(void)

Description: Saves the current user configuration of the component. The period, dead band, counter, and control register values are saved.

Parameters: None

Return Value: None

Side Effects: None

void PWM_RestoreConfig(void)

Description: Restores the current user configuration of the component.

Parameters: None

Return Value: None

Side Effects: None

Sample Firmware Source Code

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

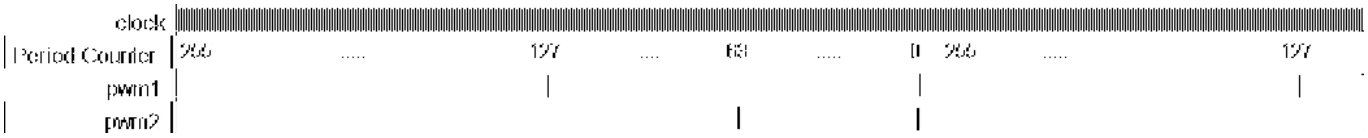
Refer to the “Find Example Project” topic in the PSoC Creator Help for more information.

Functional Description

Default Configuration

The default configuration for the PWM is as a two-output 8-bit PWM that creates one output with a compare of less than 127 (with a period of 255) and a second output of less than 63 using a 12-MHz clock. Figure 1 shows the inputs and outputs of the PWM when it is left in the default configuration.

Figure 1. PWM Inputs and Outputs



Fixed-Function Block Limitations

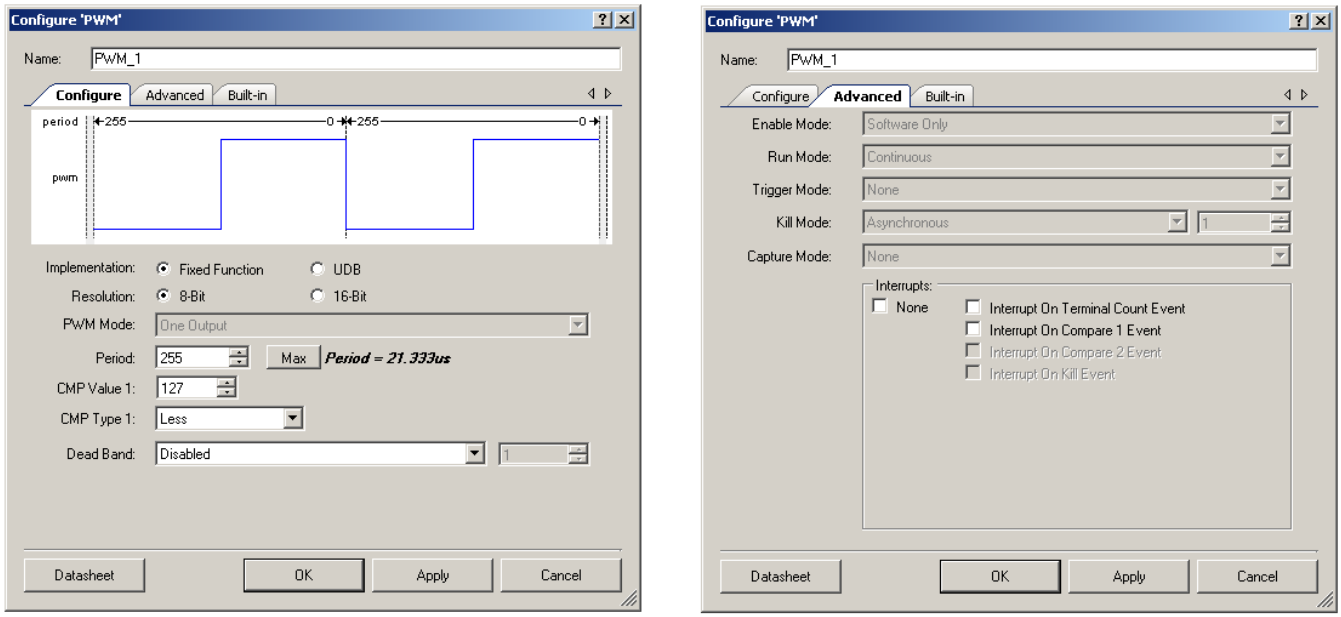
The fixed-function implementation of the PWM provides for less UDB resource use by implementing a PWM with reduced functionality in a configurable hardware block. The functionality of the PWM within one of these blocks has the following limitations:

- No counter value access – PWM_ReadCapture() and PWM_ReadCounter() are not available.
- One output mode only – No Center Align, Dual Edge, Dither, or Two Outputs modes
- Asynchronous kill mode only
- No trigger
- Continuous run mode only
- Software enable only – No Hardware enable mode

- Reduced dead band functionality – Limited to 0 to 3 counts of dead band
- Reduced I/O when dead band is enabled – TC and CMP1 become PH1 and PH2, respectively.

When you choose the **Fixed Function** implementation, the **Configure** tab dialog and the **Advanced** tab dialog indicate these limitations by setting the parameter fields and disabling the options, as shown in Figure 2.

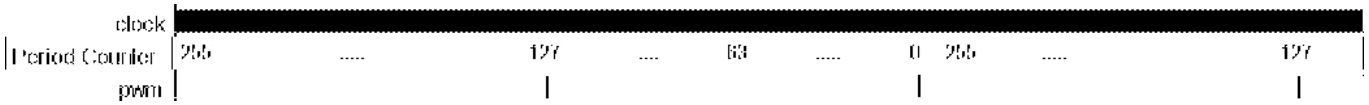
Figure 2. Fixed Function Settings



PWM Mode

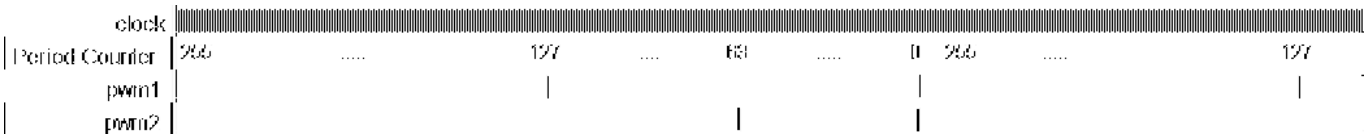
One Output

A one-output PWM has only one output that is controlled by a single compare value and a single compare mode. This waveform can be left-aligned with a compare mode of **Greater** or **Greater or Equal** or it can be right-aligned with a compare mode of **Less** or **Less or Equal**.



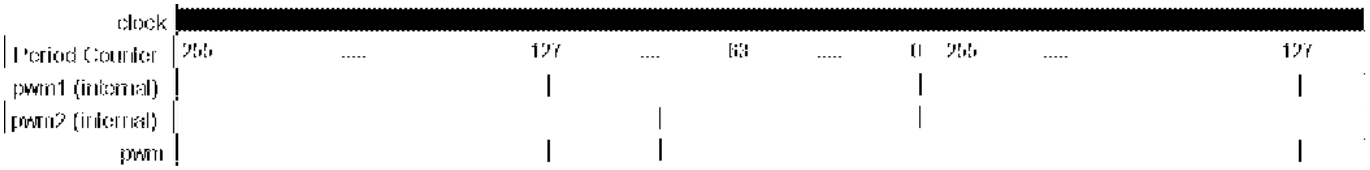
Two Outputs

The two-output PWM is the default configuration. The two PWM outputs are defined independently of each other using two compare values and two compare modes. Each of these two outputs can be left aligned or right aligned, as described previously in One Output mode.



Dual Edge

A dual-edge PWM uses the two compare outputs and two compare modes to generate a single PWM output. The final output is an ANDing of the two different signals defined by the two compare values and compare modes. This mode requires you to have some understanding of what the different modes will generate. The waveform examples in the parameter editing customizer provide help as to what the final waveform will look like. However, the compare values, compare modes, and period values can all be set at run time. Changing these values without understanding the final configuration can easily create a 0 value output.

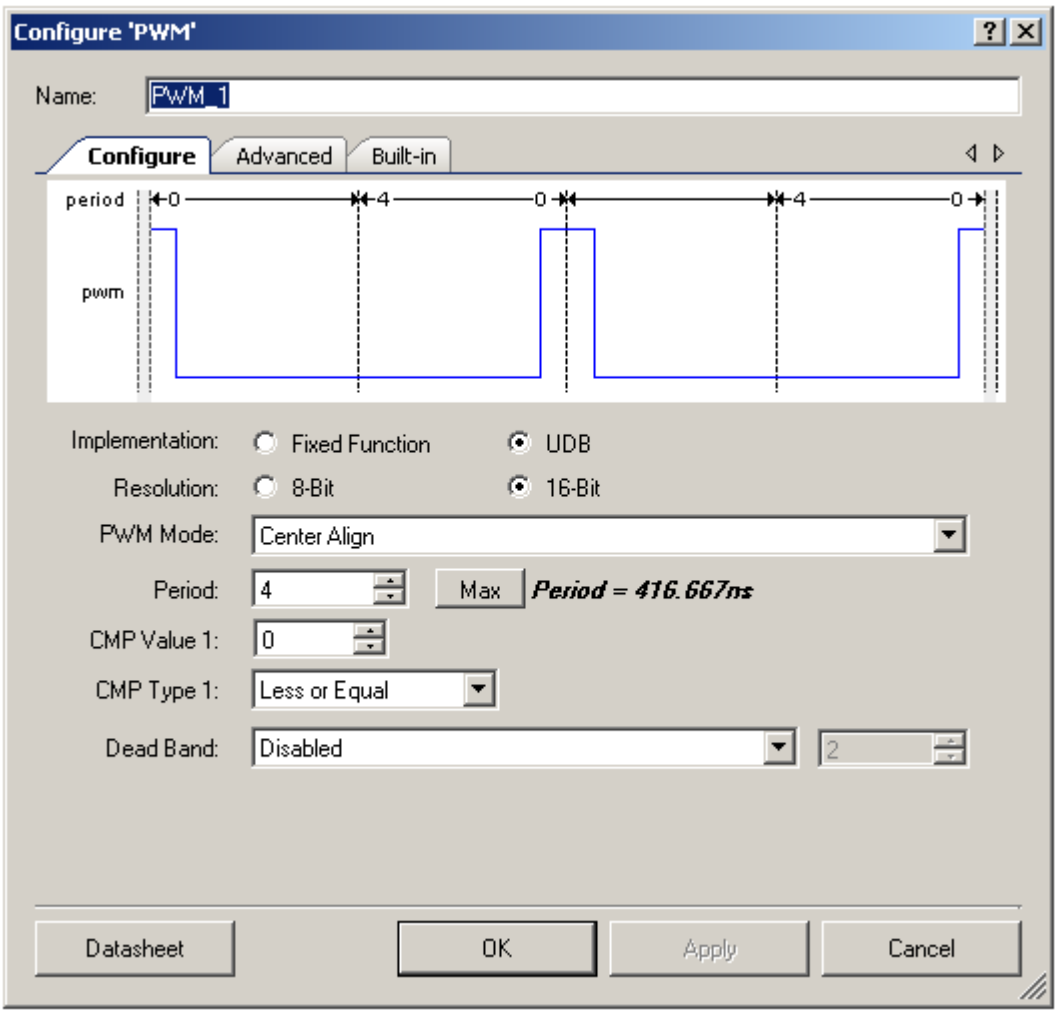


Center Aligned

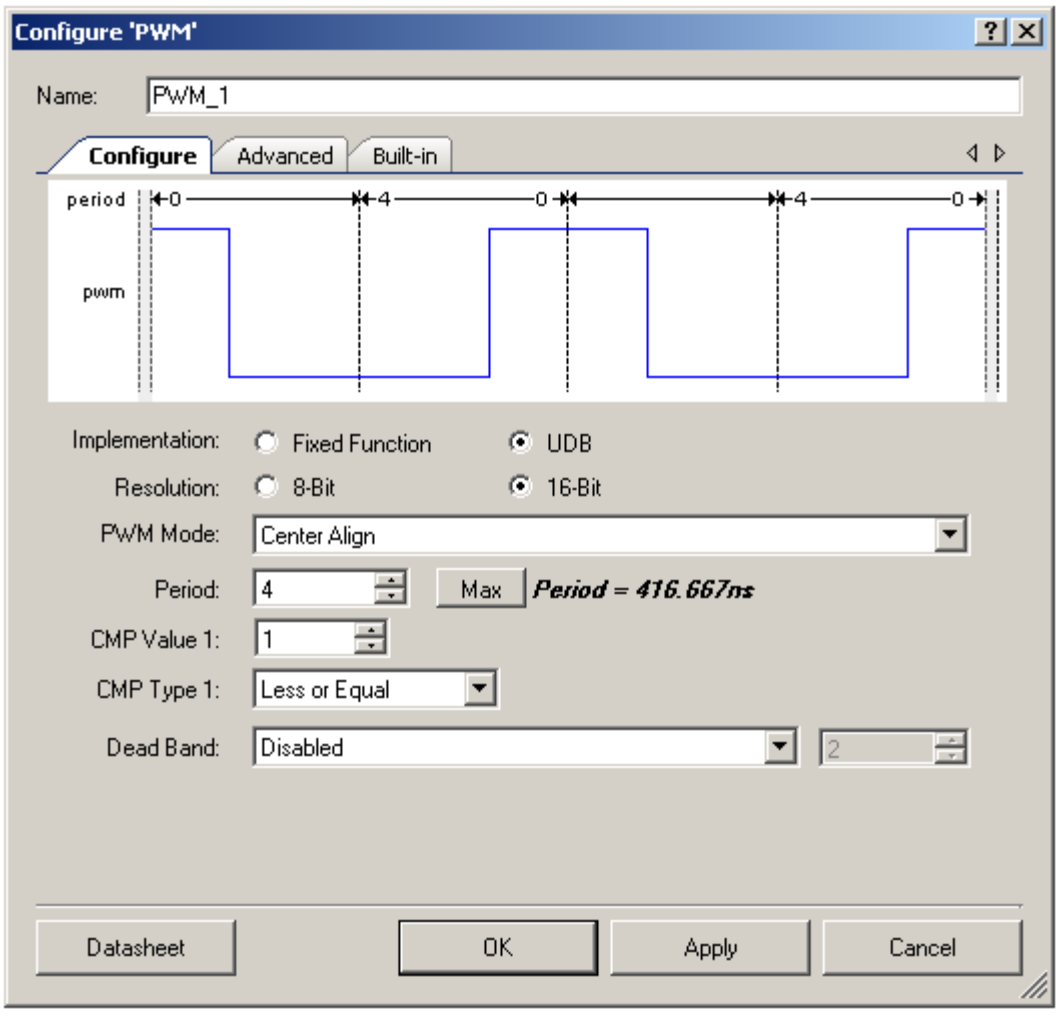
A center-aligned PWM implements the PWM differently from all of the other modes. The desired output requires that the period counter start at zero and count up to the period value, and when the period value is reached the counter starts counting back down to zero. In this mode, the period value is actually half of the period of the final output. A single compare value and compare mode are available for this functionality.

All other modes of the PWM start the period counter at the period setting counting down to 0 and reloading to the period value which makes them period+1 for the actual period time. This is represented in the calculated period displayed in the customizer. For center-aligned mode the calculated period is NOT +1. This is because the period counter counts from 0 to period and immediately starts counting back down. For example, with a period of 4 the counter will count, 0,1,2,3,4,3,2,1,0,1,2… making the period 4 clock cycles.

The customizer indicates this with a half bit time at the beginning and end of the waveforms displayed in the customizer.



For this configuration, the counter is only less than or equal to zero for a single clock cycle, which is indicated as a half bit time at the beginning and end of the first period. The next image shows that the counter is less than or equal to 1 for three clock cycles indicated as 1.5 clock cycles at the beginning and end of each of the two periods indicated in the waveforms.



Hardware Select

A hardware-select PWM is implemented as a two-output PWM, where the implementation has two independent compare values and compare modes. A hardware input cmp_sel selects which of the two inputs is the final PWM output. This allows you to switch between two preconfigured values as necessary, without modifying the parameters.

Dither

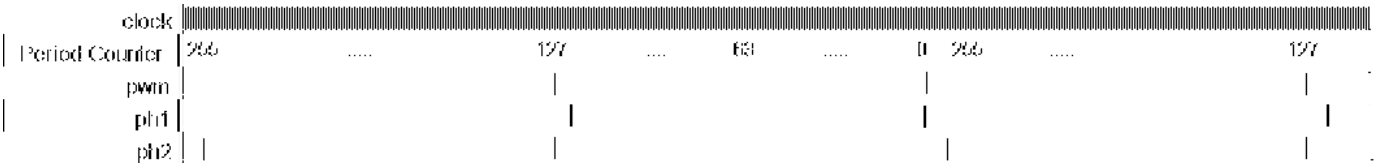
A dither-mode PWM is implemented as a hardware-select-mode PWM with the caveat that the first and compare values have a difference of 1 and both compare modes are identical. There is also a built-in state machine controlling the hardware select. In this mode, the cmp_sel input is not available. You can set the offset as 0.00, 0.25, 0.50, and 0.75, with the parameter field visible in this mode. If the offset is configured as 0.00, then the output is always the compare1 output. When set to 0.25 the output is compare1 for three cycles and compare1 + 1 for a single cycle.

Dither Mode	Cycle 0	Cycle 1	Cycle 2	Cycle 3
0.00	Compare1	Compare1	Compare1	Compare1
0.25	Compare1 + 1	Compare1	Compare1	Compare1
0.50	Compare1	Compare1 + 1	Compare1	Compare1 + 1
0.75	Compare1 + 1	Compare1 + 1	Compare1 + 1	Compare 1

Dead Band

Dead band is an add-on option to any of the PWM modes just described. When dead band is enabled, two new outputs, ph1 and ph2 (phase1 and phase2), become visible on the symbol. The dead band outputs work on a single PWM output. In all modes except two-output mode, the dead band outputs are related to the single PWM output. In two-output mode, the dead band is only implemented on the pwm1 output. In all dead band modes, the original output is available, along with the ph1 and ph2 outputs.

Dead band can be configured as having a range of 2 to 4 or 2 to 256 clock cycles for dead band time. The 2 to 4 cycle range is provided to reduce resource usage by implementing the counter in PLDs instead of using a full datapath. When the 2 to 256 range dead band is selected, a full datapath and the necessary logic are used from the UDB array.



Kill Mode

Like dead band, kill mode is an add-on function that does not interrupt the implementation of the PWM internally. This add-on is placed at the outputs of the PWM and manipulates only the final output signals. When dead band is not implemented, the kill operation disables the PWM outputs by pulling them low. If dead band is implemented, the kill operation disables the ph1 and ph2 outputs by pulling ph1 low and ph2 high.

Asynchronous

In asynchronous kill mode, the outputs are disabled while the kill input is active (high) and the outputs are re-enabled as soon as the kill input goes inactive.

Single Cycle

In single cycle kill mode, the outputs are disabled while the kill input is active (high) and the outputs are re-enabled at the beginning of the next period.

Latched

In latched kill mode, the outputs are disabled when the kill input goes high. After the PWM has been reset, if the kill input is not still active, the PWM outputs are re-enabled; otherwise, they remain in the kill state until the next reset of the PWM with an inactive kill input.

Minimum Time

In minimum-time kill mode, the outputs are disabled while the kill input is active (high). The outputs are re-enabled after the minimum time has elapsed, if the kill input is no longer active. For this mode, you define the minimum kill time in the number of clock counts 1 to 255. The API necessary for controlling the minimum kill time counts is only available if this kill mode is selected.

Run Mode

Continuous

Continuous run mode is the default configuration of the PWM. This mode allows the PWM to run forever while enabled. As long as the PWM is enabled, the output cycles through period after period implementing the specified pulse width output.

One Shot Single

One Shot Single run mode runs the PWM for a single period on a valid trigger event. After the period has completed the PWM halts. The PWM halts after reloading the counter with the value from the period register. A hardware reset pulse will re-arm the PWM and allow the next trigger event to cause the PWM to run another period.

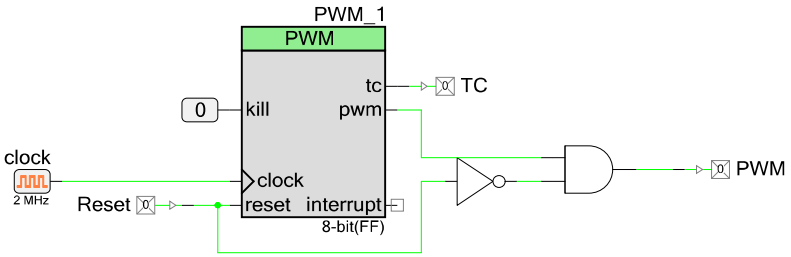
One Shot Multi

One Shot Multi run mode runs the PWM for a single period on a valid trigger event. After the period has completed the PWM halts and re-arms waiting for the next trigger event The PWM halts after reloading the counter with the value from the period register. The difference between One Shot Single and One Shot Multi run modes is that One Shot Multi is re-armed without requiring a reset.

Reset in Fixed-Function Block

The fixed-function implementation of the PWM differs from the UDB implementation in that the pwm output goes high during reset, whereas in the UDB implementation the pwm output goes low. It is very easy to change the functionality of either of the two implementations, as shown in Figure 3. Figure 3 shows a fixed-function PWM implementation that drives the PWM output low while the reset input is active, giving the same functionality as the UDB implementation of the same component.

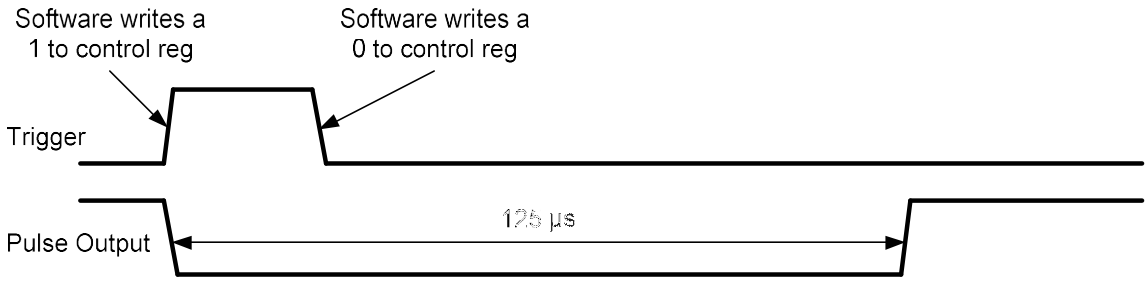
Figure 3. Fixed-Function PWM Implementation Schematic



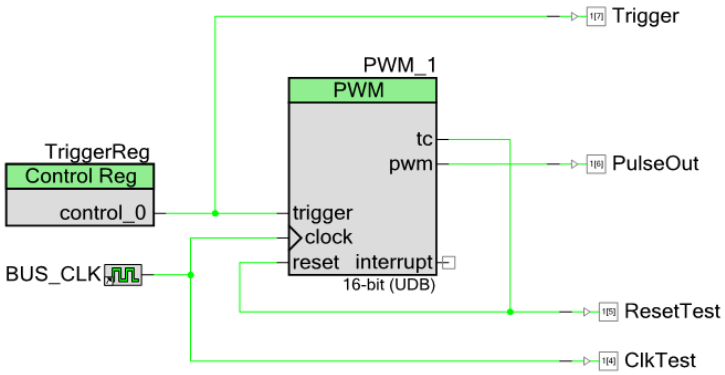
It is better to implement this change on the fixed-function implementation because there is only a single output to deal with, whereas the UDB implementation has a mode with two. It is also better to drive the outputs low during reset in most situations.

PWM Component as a Pulse Generator

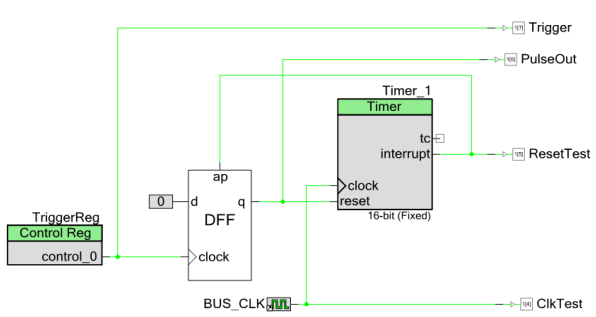
A PWM component can be used to design a software-triggered pulse generator circuit to generate a timing pulse of a known period. The following timing diagram describes an example of a pulse generation application that generates a timing pulse of 125 µs on a software trigger.

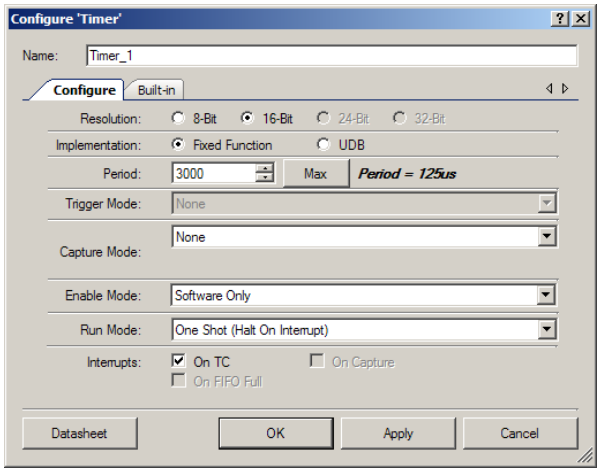


A PWM configured in One Shot Single mode can be used with a control register component to realize this design. In the One Shot Single mode, the PWM should be reset after it reaches the period value to make sure it functions correctly. You can do this by connecting the TC output to its reset input. The following schematic features a UDB implementation of the PWM that creates such a circuit.

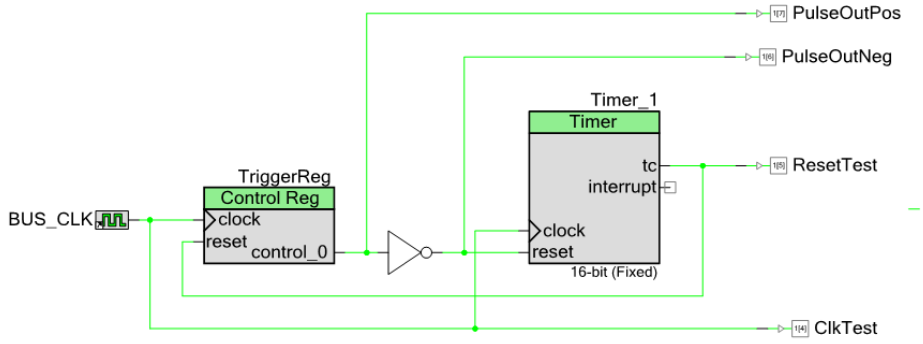


The previous design is UDB resource intensive. You can choose the fixed-function implementation, as shown in the following schematic. This design uses an additional DFF component and, unlike the previous implementation, does not use any of the datapath elements.





In Production PSoC 3, you can configure the control register component differently so that the DFF is no longer needed. In this configuration, the control register write function needs to be called only once to write a 1. This Production PSoC3 implementation of the timing pulse generation circuit is shown in the following schematic.

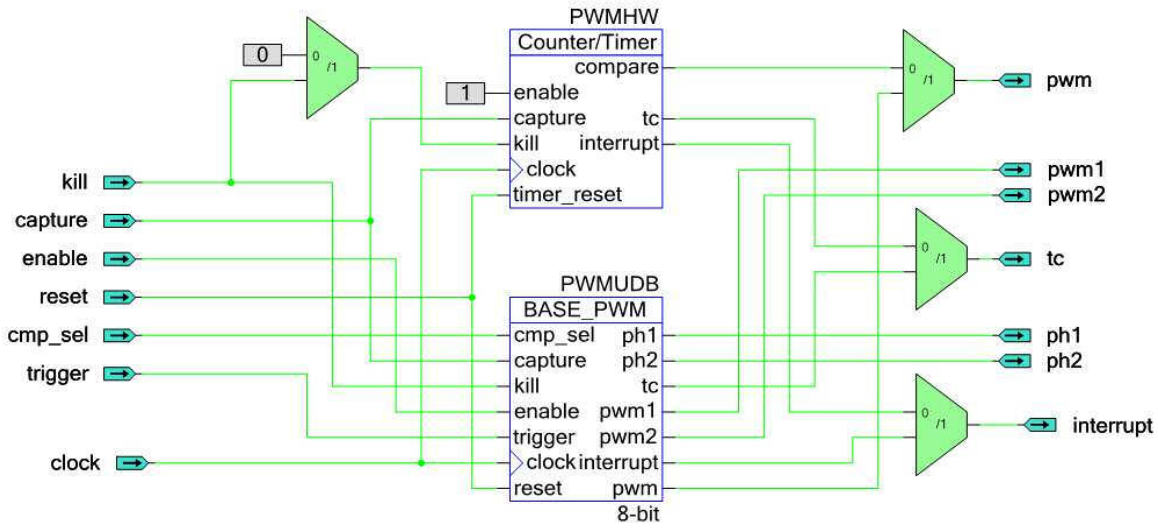


Block Diagram and Configuration

The PWM can be implemented using a fixed-function block or using UDB components. An advanced parameter, **Implementation**, allows you to specify the block in which you expect to place this component. The fixed function implementation consumes one of the Timer/Counter/PWM blocks. In both the fixed-function or UDB configurations all of the registers and APIs are consolidated to give a single entity look and feel. The API is described earlier and the registers are described here to define the overall implementation of the PWM.

The two hardware implementations you chose are selected from a top level schematic as shown in Figure 4.

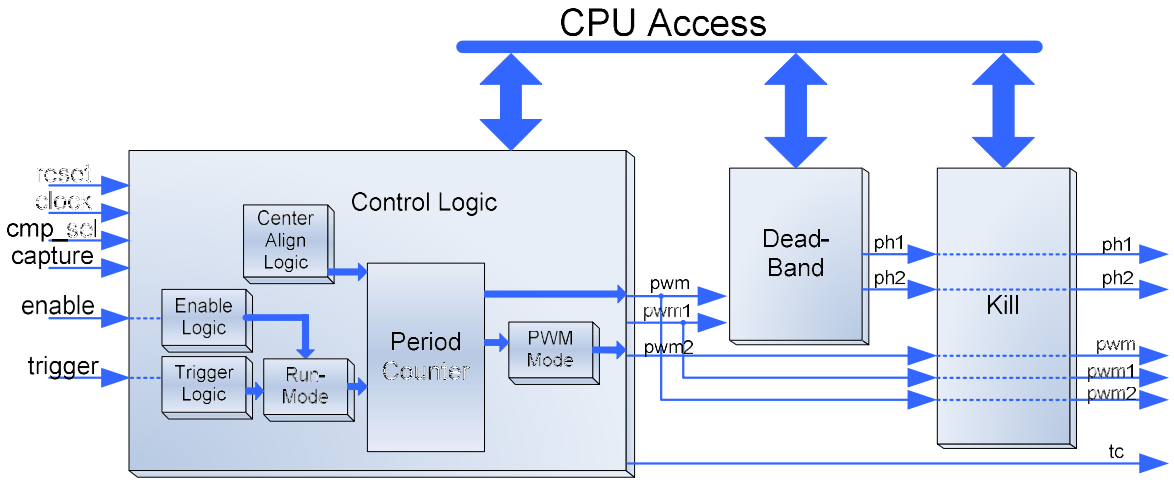
Figure 4. Top Level Schematic



This configuration allows you to select either the fixed-function block or the UDB implementation. The routing of the I/Os is handled in the background to give this single component look and feel.

The UDB implementation is described in Figure 5.

Figure 5. UDB Implementation



Registers

Status

The status register is a read-only register that contains the various status bits defined for the PWM. The PWM_ReadStatusRegister() function call gives you the value of this register. The interrupt output signal (interrupt) is generated from an ORing of the masked bit fields within this register. You can set the mask using the PWM_SetInterruptMode() function call and upon receiving an interrupt you can retrieve the interrupt source by reading the status register with the PWM_ReadStatusRegister() function call. The status register is a clear-on-read register so the interrupt source is held until the PWM_ReadStatusRegister() function is called. The PWM_ReadStatusRegister() API handles which interrupts are enabled to provide an accurate report of the actual source of the interrupt. All operations on the status register must use the following defines for the bit fields because these bit fields may be moved within the status register during place and route.

You may choose to remove the status register completely from the hardware by setting the **None** option in the **Interrupts** section of the configuration editor. If this option is set, the API does not support access to the status register. Building a design with API access to the status register will have errors stating that the PWM_1_PWMUDB_sSTSReg_stsreg__STATUS_REG is an undefined identifier. This can be corrected by removing the API and deselecting the **None** option for interrupts in the configuration editor.

The status data is registered at the input clock edge of the counter, giving all bits configured as Mode = 1 the timing resolution of the counter. These bits are sticky and are cleared on a read of the status register. All other bits configured as Mode = 0 are transparent and read directly from the inputs to the status register; they are not sticky and therefore not clear on read. All bits configured as Mode = 1 are indicated with an asterisk (*) in the defines in the following list.

There are several bit-fields masks defined in the status register. Any of these bit fields may be included as an interrupt source. The #defines are available in the generated header file (.h) as follows:

PWM_STATUS_TC *

Status of the terminal count output. This bit goes high when the terminal count output is high.

PWM_STATUS_CMP1 *

Status of the pwm1 compare value as it relates to the period counter. For a fixed-function implementation, this bit goes high when the comparison output is high. For a UDB implementation, this bit is asserted with the registered version of the comparison output; so, the bit is asserted two clocks after the comparison output is high.

PWM_STATUS_CMP2 *

Status of the pwm2 compare value as it relates to the period counter. For a fixed-function implementation, this bit goes high when the comparison output is high. For a UDB implementation, this bit is asserted with the registered version of the comparison output; so, the bit is asserted two clocks after the comparison output is high.

PWM_STATUS_KILL

Status of the output kill. If it is currently active the output will be high.

PWM_STATUS_FIFOFULL

Status of the Capture FIFO level. This bit is a real-time status of the FIFO level indicating that the FIFO is currently Full. A “0” in this bit of the status register indicates that the FIFO is not full but does not indicate that there is not data in the FIFO.

Control

The control register allows you to control the general operation of the PWM. This register is written with the PWM_WriteControlRegister() function call and read with PWM_ReadControlRegister(). When reading or writing the control register you must use the bit-field definitions as defined in the header (.h) file. The #defines for the control register are as follows:

PWM_CTRL_ENABLE

The enable bit controls software enabling of the PWM operation. The PWM has a configurable enable mode defined at build time. If the **Enable Mode** parameter is set to **Hardware Only**, this bit has no function. However, in either of the other modes the PWM does not decrement unless this bit is set high. Normal operation requires that this bit is set and held high during all operation of the PWM.

PWM_CTRL_CMPMODE1_MASK

The compare mode control is a three-bit field used to define the expected compare output operation for the pwm1 output. This bit field is three consecutive bits in the control register. All operations on this bit-field must use the #defines associated with the compare modes available. These are:

- PWM_1_B_PWM_CM_LESSTHAN
- PWM_1_B_PWM_CM_LESSTHANOREQUAL
- PWM_1_B_PWM_CM_EQUAL
- PWM_1_B_PWM_CM_GREATERTHAN

- PWM_1_B_PWM_CM_GREATERTHANOREQUAL

This bit field is configured at initialization with the compare mode defined in the CompareMode1 parameter and may be modified with the PWM_SetCompareMode() or PWM_SetCompareMode1() API call.

PWM_CTRL_CMPMODE2_MASK

The compare mode control is a three-bit field used to define the expected compare output operation for the pwm2 output. This bit field is three consecutive bits in the control register. All operations on this bit field must use the #defines associated with the compare modes available. These are:

- PWM_1_B_PWM_CM_LESSTHAN
- PWM_1_B_PWM_CM_LESSTHANOREQUAL
- PWM_1_B_PWM_CM_EQUAL
- PWM_1_B_PWM_CM_GREATERTHAN
- PWM_1_B_PWM_CM_GREATERTHANOREQUAL

This bit field is configured at initialization with the compare mode defined in the CompareMode2 parameter and may be modified with the PWM_SetCompareMode2() API call.

Period (8 or 16-bit based on Resolution)

The period register contains the period value set by the user through the PWM_WritePeriod() function call and defined by the **Period** parameter at initialization. The PWM_ReadPeriod() function may be used to find the current value of this register. The period register has no effect on the PWM until a terminal count is reached, at which time the period counter is reloaded with this value.

Compare1/Compare2 (8 or 16-bit based on Resolution)

The compare registers contains the compare values used to determine the state of the pwm or pwm1 and pwm2 outputs (depending on the setting of the **PWM Mode** parameter). The pwm/pwm1 and pwm2 outputs are based on how these registers compare to the period counter value in relation to the compare modes defined in the control register.

Period Counter (8 or 16-bit based on Resolution)

The period counter register contains the counter value throughout the operation of the PWM. During basic operation, this register decrements by 1 while the PWM is enabled and on each rising edge of the clock input. You can read the contents of this register at any time with the PWM_ReadCounter() function call. When the terminal count is reached, this register is reloaded

with the period value you define in the period register through the PWM_WritePeriod() function call or during initialization with the **Period** parameter.

The pwm, pwm1, and pwm2 outputs are based on the relationship between the value held in this register and the value defined in the compare registers through the PWM_WriteCompare() function calls or during initialization with the CompareValue parameters

Conditional Compilation Information

The PWM API requires several conditional compile definitions to handle the multiple configurations it must support. The API must conditionally compile on the resolution chosen, the implementation chosen between the fixed-function block or the UDB blocks, dead band modes, kill modes, and PWM modes. The conditions defined are based on the parameters FixedFunction, Resolution, DeadBand, KillMode and PWMMode. The API should never use these parameters directly but should use the defines listed below.

PWM_Resolution

The resolution define is assigned to the resolution value at build time. It is used throughout the API to compile in the correct data width types for the API functions relying on this information.

PWM_UsingFixedFunction

The using fixed function define is used mostly in the header file to make the correct register assignments. This is necessary because the registers provided in the fixed-function block are different than those used when the PWM is implemented in UDBs.

PWM_DeadBandMode

The dead band mode define is used to conditionally compile in PWM_WriteDeadTime() and PWM_ReadDeadTime() APIs.

PWM_KillModeMinTime

The kill mode minimum time define is used to conditionally compile in PWM_WriteKillTime() and PWM_ReadKillTime() APIs.

PWM_KillMode

The kill mode define is used to define the register access point for the Kill Mode Min Time register if **Kill Mode** is set to **Minimum Time**.

PWM_PWMMode

The PWM mode define is used to include the correct PWM_WriteCompare() and PWM_ReadCompare() API functions as necessary for the mode in use.

PWM_PWMModeIsCenterAligned

The PWM mode is center aligned define is used to redefine the period register address. Center aligned is different from other modes in implementation and requires the use of different registers for operation that must be handled in the header file.

PWM_DeadBandUsed

The deadband used define controls conditionally compiling the PWM_WriteDeadTime() and PWM_ReadDeadTime() APIs.

PWM_DeadBand2_4

The deadband 2-4 define controls conditionally compiling the implementation within the PWM_WriteDeadTime() and PWM_ReadDeadTime() APIs.

PWM_UseStatus

The use status define is used to remove the status register, if the design requires it, in the Verilog code and to conditionally compile out the status register definitions and APIs in the header and C files.

PWM_UseControl

The use control define is used to remove the control register, if the design requires it, in the Verilog code and to conditionally compile out the control register definitions and APIs in the header and C files.

PWM_UseOneCompareMode

The use one compare mode is used to conditionally compile in and out the expected API calls necessary for 1 or 2 compare mode PWM mode functions.

PWM_MinimumKillTime

Provides the initial minimum kill time programmed into the min-time datapath when the **Kill Mode** is set to **Minimum Kill Time**.

PWM_EnableMode

Allows for condition compilation to remove the API provided for specific Enable modes.

Constants

There are several constants defined for the status and control registers, as well as some of the enumerated types. Most of the constants for the control and status registers have been described earlier in this datasheet. However, there are more constants needed in the header file to make all of this happen. Each of the register definitions requires either a pointer into the

register data or a register address. Because of multiple Endianness of the compilers, the CY_GET_REGX and CY_SET_REGX macros must be used for register accesses greater than eight bits. These macros require the use of the _PTR definition for each of the registers.

It is also required that the control and status register bits be allowed to be placed and routed by the fitter engine, because the component must have constants that define the placement of the bits. For each of the status and control register bits there is an associated _SHIFT value that defines the bit's offset within the register. These are used in the header file to define the final bit mask as a _MASK definition. (The _MASK extension is only added to bit fields greater than a single bit, all single bit values drop the _MASK extension.)

The fixed-function block has some limitations compared to the UDB implementations because it is designed with limited configurability.

DC and AC Electrical Characteristics for PSoC 3 (FF Implementation)

The following values indicate expected performance and are based on initial characterization data.

PWM DC Specifications

Parameter	Description	Conditions	Min	Typ	Max	Units
	Block current consumption	16-bit PWM, at listed input clock frequency	–	–	–	µA
	3 MHz		–	15	–	µA
	12 MHz		–	30	–	µA
	48 MHz		–	260	–	µA
	67 MHz		–	350	–	µA

PWM AC Specifications

Parameter	Description	Conditions	Min	Typ	Max	Units
	Operating frequency		DC	–	67	MHz
	Capture pulse width (internal)		15	–	–	ns
	Capture pulse width (external)		30	–	–	ns
	Timer resolution		15	–	–	ns
	Enable pulse width		15	–	–	ns
	Enable pulse width (external)		30	–	–	ns

Parameter	Description	Conditions	Min	Typ	Max	Units
	Reset pulse width		15	–	–	ns
	Reset pulse width (external)		30	–	–	ns

DC and AC Electrical Characteristics for PSoC 5 (FF Implementation)

The following values indicate expected performance and are based on initial characterization data.

PWM DC Specifications

Parameter	Description	Conditions	Min	Typ	Max	Units
	Block current consumption	16-bit PWM, at listed input clock frequency	–	–	–	µA
	3 MHz		–	65	–	µA
	12 MHz		–	170	–	µA
	48 MHz		–	650	–	µA
	67 MHz		–	900	–	µA

PWM AC Specifications

Parameter	Description	Conditions	Min	Typ	Max	Units
	Operating frequency		DC	–	67.01	MHz
	Pulse width		13	–	–	ns
	Pulse width (external)		30	–	–	ns
	Kill pulse width		13	–	–	ns
	Kill pulse width (external)		30	–	–	ns
	Enable pulse width		13	–	–	ns
	Enable pulse width (external)		30	–	–	ns
	Reset pulse width		13	–	–	ns
	Reset pulse width (external)		30	–	–	ns

DC and AC Electrical Characteristics (UDB Implementation)

The following values indicate expected performance and are based on initial characterization data.

Timing Characteristics "Maximum with Nominal Routing"

Parameter	Description	Config. [4]	Min	Typ	Max	Units
f_{CLOCK}	Component clock frequency	Config 1	–	–	50	MHz
		Config 2	–	–	45	MHz
		Config 3	–	–	40	MHz
		Config 4	–	–	50	MHz
		Config 5	–	–	50	MHz
		Config 6	–	–	40	MHz
		Config 7	–	–	35	MHz
		Config 8	–	–	30	MHz
		Config 9	–	–	35	MHz
		Config 10	–	–	35	MHz
t_{CLOCKH}	Input clock high time [5]	N/A	–	0.5	–	$1/f_{CLOCK}$
t_{CLOCKL}	Input clock low time [5]	N/A	–	0.5	–	$1/f_{CLOCK}$

[4] Configurations of the PWM component

Config1:
- Resolution : 8 bits
- PWM Type : One output in Continuous Run Mode

Config2:
- Resolution: 8 bits
- PWM Type: One output in Continuous Run Mode with Dither

Config3:
- Resolution : 8 bits
- PWM Type : Center Aligned Output

Config 4:
- Resolution: 8 bits
- PWM Type: One output with Dead Band

Config 5:
- Resolution : 8 bits
- PWM Type : Continuous Run mode with Kill Mode.

Config6:
- Resolution : 16 bits
- PWM Type : One output in Continuous Run Mode

Config7:
- Resolution: 16 bits
- PWM Type: One output in Continuous Run Mode with Dither

Config8:
- Resolution : 16 bits
- PWM Type : Center Aligned Output

Config 9:
- Resolution: 16 bits
- PWM Type: One output with Dead Band

Config 10:
- Resolution : 16 bits
- PWM Type : Continuous Run mode with Kill Mode.

[5] $t_{CY_clock} = 1/f_{CLOCK}$. This is the cycle time of one clock period.

Parameter	Description	Config. [4]	Min	Typ	Max	Units
Inputs						
t_{PD_ps}	Input path delay, pin to sync [6]	1	–	–	STA [7]	ns
t_{PD_ps}	Input path delay, pin to sync [8]	2	–	–	8.5	ns
t_{PD_si}	Sync output to input path delay (route)	1,2,3,4	–	–	STA [7]	ns
t_{I_clk}	Alignment of clockX and clock	1,2,3,4	0	–	1	t_{CY_clock}
t_{PD_IE}	Input path delay to component Clock (edge-sensitive input)	1,2	t_{PD_ps} + t_{SYNC} + t_{PD_si}	–	t_{PD_ps} + t_{SYNC} + t_{PD_si} + t_{I_clk}	ns
t_{PD_IE}	Input path delay to component clock (edge-sensitive input)	3,4	t_{SYNC} + t_{PD_si}	–	t_{SYNC} + t_{PD_si} + t_{I_clk}	ns
t_{IH}	Input high time	1,2,3,4	t_{CY_clock}	–	–	ns
t_{IL}	Input low time	1,2,3,4	t_{CY_clock}	–	–	Ns

[6] t_{PD_ps} is found in the Static Timing Results as described later. The number listed here is a nominal value based on STA analysis on many inputs.

[7] t_{PD_ps} and t_{PD_si} are route path delays. Because routing is dynamic, these values can change and directly affect the maximum component clock and sync clock frequencies. The values are found in the Static Timing Analysis results.

[8] t_{PD_ps} in configuration 2 is a fixed value defined for each pin of the device. The number listed here is a nominal value of all of the pins available on the device.

Timing Characteristics "Maximum with All Routing"

Parameter	Description	Config. [9]	Min	Typ	Max [10]	Units
f_{CLOCK}	Component clock frequency	Config 1			25	MHz
		Config 2			23	MHz
		Config 3			20	MHz
		Config 4			25	MHz
		Config 5			25	MHz
		Config 6			20	MHz
		Config 7			18	MHz
		Config 8			15	MHz
		Config 9			18	MHz
		Config 10			18	MHz
t_{CLOCKH}	Input clock high time [11]	N/A		0.5		$1/f_{clock}$
t_{CLOCKL}	Input clock low time [11]	N/A		0.5		$1/f_{clock}$

[9] Configurations of the PWM component
- Config1:
  - Resolution : 8 bits
  - PWM Type : One output in Continuous Run Mode
- Config2:
  - Resolution: 8 bits
  - PWM Type: One output in Continuous Run Mode with Dither
- Config3:
  - Resolution : 8 bits
  - PWM Type : Centre Aligned Output
- Config 4:
  - Resolution: 8 bits
  - PWM Type: One output with Deadband
- Config 5:
  - Resolution : 8 bits
  - PWM Type : Continuous Run mode with Kill Mode.
- Config6:
  - Resolution : 16 bits
  - PWM Type : One output in Continuous Run Mode
- Config7:
  - Resolution: 16 bits
  - PWM Type: One output in Continuous Run Mode with Dither
- Config8:
  - Resolution : 16 bits
  - PWM Type : Center Aligned Output
- Config 9:
  - Resolution: 16 bits
  - PWM Type: One output with Dead Band
- Config 10:
  - Resolution : 16 bits
  - PWM Type : Continuous Run mode with Kill Mode.

[10] The Maximum for All Routing timing numbers are calculated by derating the Nominal Routing timing numbers by a factor of 2. If your component instance operates at or below these speeds, then meeting timing should not be a concern for this component.

[11] $t_{CY_clock} = 1/f_{CLOCK}$. This is the cycle time of one clock period.

Parameter	Description	Config. [9]	Min	Typ	Max [10]	Units
Inputs						
t_{PD_ps}	Input path delay, pin to sync [12]	1			STA [13]	ns
t_{PD_ps}	Input path delay, pin to sync [14]	2			8.5	ns
t_{PD_si}	Sync output to input path delay (route)	1,2,3,4			STA [13]	ns
t_{I_clk}	Alignment of clockX and clock	1,2,3,4	0		1	t_{CY_clock}
t_{PD_IE}	Input path delay to component clock (edge-sensitive input)	1,2	t_{PD_ps} + t_{SYNC} + t_{PD_si}		t_{PD_ps} + t_{SYNC} + t_{PD_si} + t_{I_clk}	ns
t_{PD_IE}	Input path delay to component clock (edge-sensitive input)	3,4	t_{SYNC} + t_{PD_si}		t_{SYNC} + t_{PD_si} + t_{I_clk}	ns
t_{IH}	Input high time	1,2,3,4	t_{CY_clock}			ns
t_{IL}	Input low time	1,2,3,4	t_{CY_clock}			ns

How to Use STA Results for Characteristics Data

Nominal route maximums are gathered through multiple test passes with Static Timing Analysis (STA). You can calculate the maximums for your designs using the STA results using the following methods:

f_{CLOCK} Maximum component clock frequency appears in Timing results in the clock summary as the named external clock. The graphic below shows an example of the clock limitations from the *_timing.html*:

-Clock Summary

Clock	Actual Freq	Max Freq	Violation
BUS_CLK	24.000 MHz	118.683 MHz	
clock	24.000 MHz	56.967 MHz	

[12] t_{PD_ps} is found in the Static Timing Results as described later. The number listed here is a nominal value based on STA analysis on many inputs.

[13] t_{PD_ps} and t_{PD_si} are route path delays. Because routing is dynamic, these values can change and directly affect the maximum component clock and sync clock frequencies. The values are found in the Static Timing Analysis results.

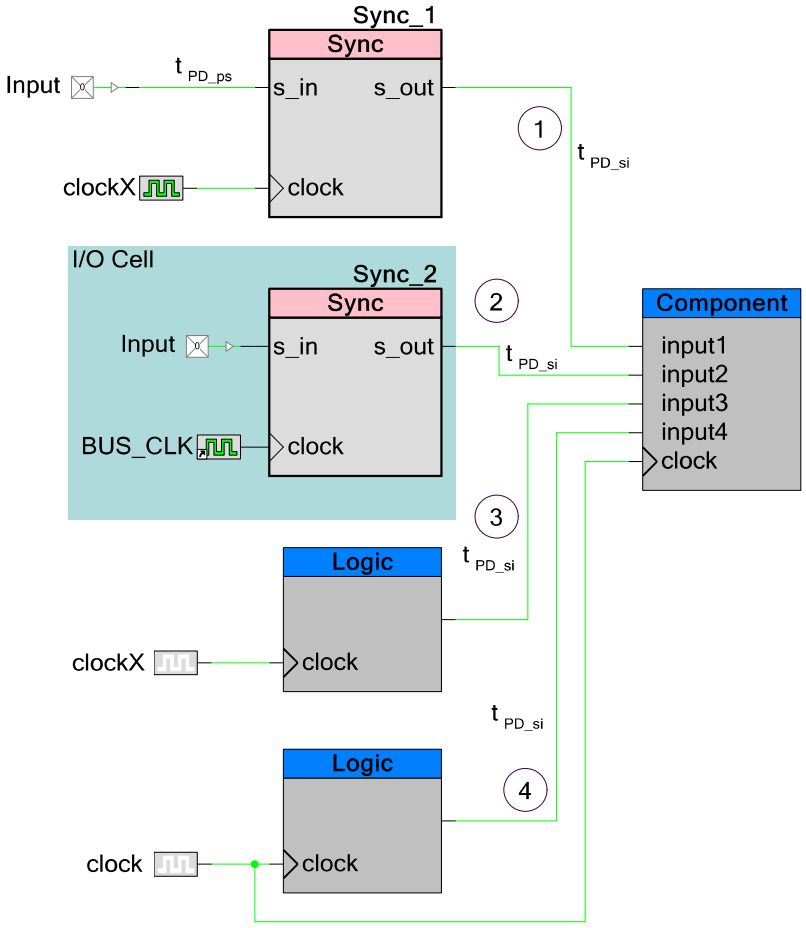
[14] t_{PD_ps} in configuration 2 is a fixed value defined per pin of the device. The number listed here is a nominal value of all of the pins available on the device.

Input Path Delay and Pulse Width

When characterizing the functionality of inputs, all inputs, no matter how you have configured them, look like one of four possible configurations, as shown in Figure 6.

All inputs must be synchronized. The synchronization mechanism depends on the source of the input to the component. To fully interpret how your system will work you must understand which input configuration you have set up for each input and the clock configuration of your system. This section describes how to use the Static Timing Analysis (STA) results to determine the characteristics of your system.

Figure 6. Input Configurations for Component Timing Specifications



Configuration	Component Clock	Synchronizer Clock (Frequency)	Figures
1	master_clock	master_clock	Figure 11
1	clock	master_clock	Figure 9
1	clock	clockX = clock [15]	Figure 7
1	clock	clockX > clock	Figure 8
1	clock	clockX < clock	Figure 10
2	master_clock	master_clock	Figure 11
2	clock	master_clock	Figure 9
3	master_clock	master_clock	Figure 16
3	clock	master_clock	Figure 14
3	clock	clockX = clock [15]	Figure 12
3	clock	clockX > clock	Figure 13
3	clock	clockX < clock	Figure 15
4	master_clock	master_clock	Figure 16
4	clock	clock	Figure 12

1. The input is driven by a device pin and synchronized internally with a “sync” component. This component is clocked using a different internal clock than the clock the component uses (all internal clocks are derived from master_clock).

   When characterizing inputs configured in this way clockX may be faster than, equal to, or slower than the component clock. It may also be equal to master_clock, which produces the characterization parameters shown in Figure 7, Figure 8, Figure 10, and Figure 11.
2. The input is driven by a device pin and synchronized at the pin using master_clock.

   When characterizing inputs configured in this way, master_clock is faster than or equal to the component clock (it is never slower than). This produces the characterization parameters shown in Figure 8 and Figure 11.

[15] Clock frequencies are equal but alignment of rising edges is not guaranteed.

Figure 7. Input Configuration 1 and 2; Synchronizer Clock Frequency = Component Clock Frequency (Edge alignment of clock and clockX is not guaranteed)

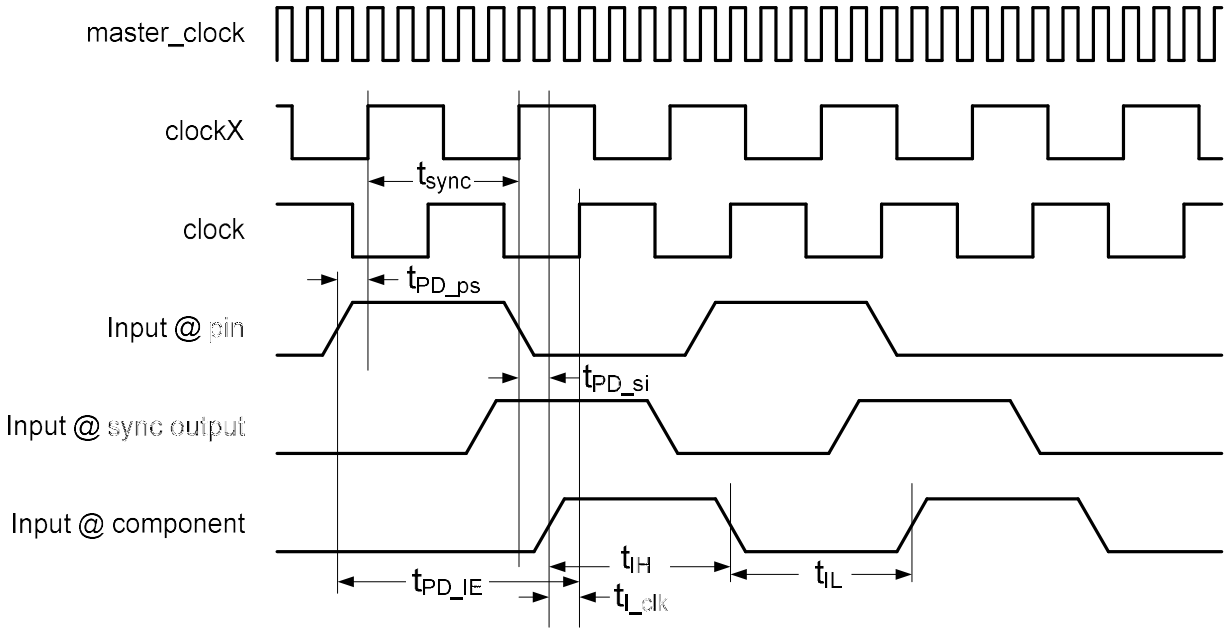


Figure 8. Input Configuration 1 and 2; Synchronizer Clock Frequency > Component Clock Frequency

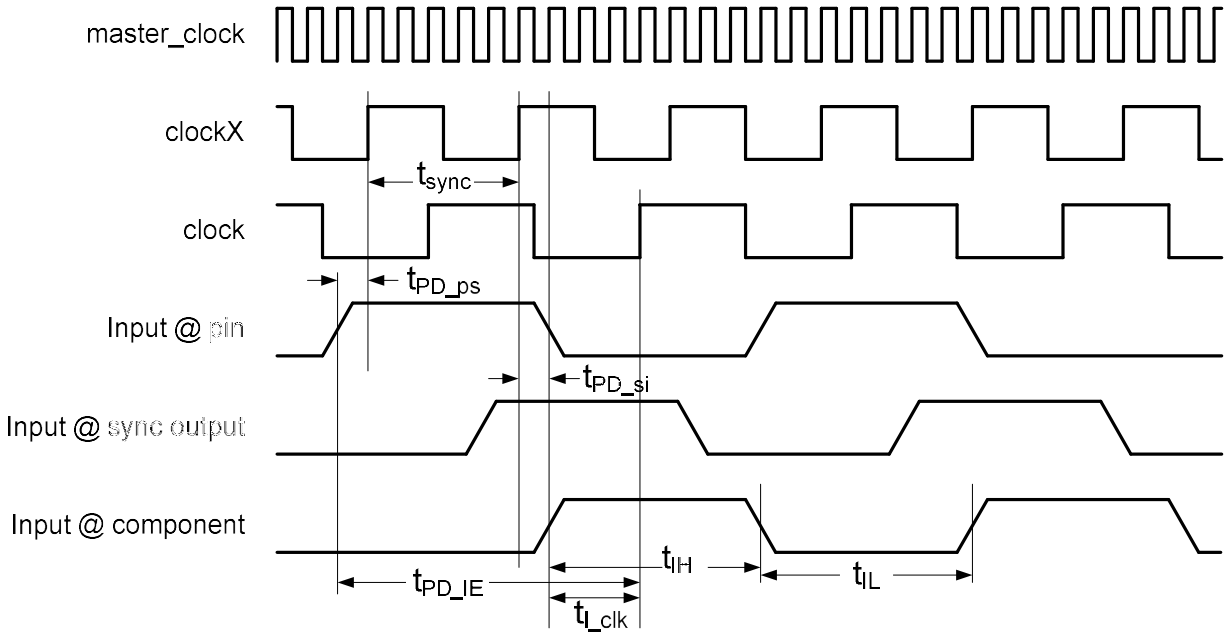


Figure 9. Input Configuration 1 and 2; [Synchronizer Clock Frequency == master_clock] > Component Clock Freq.

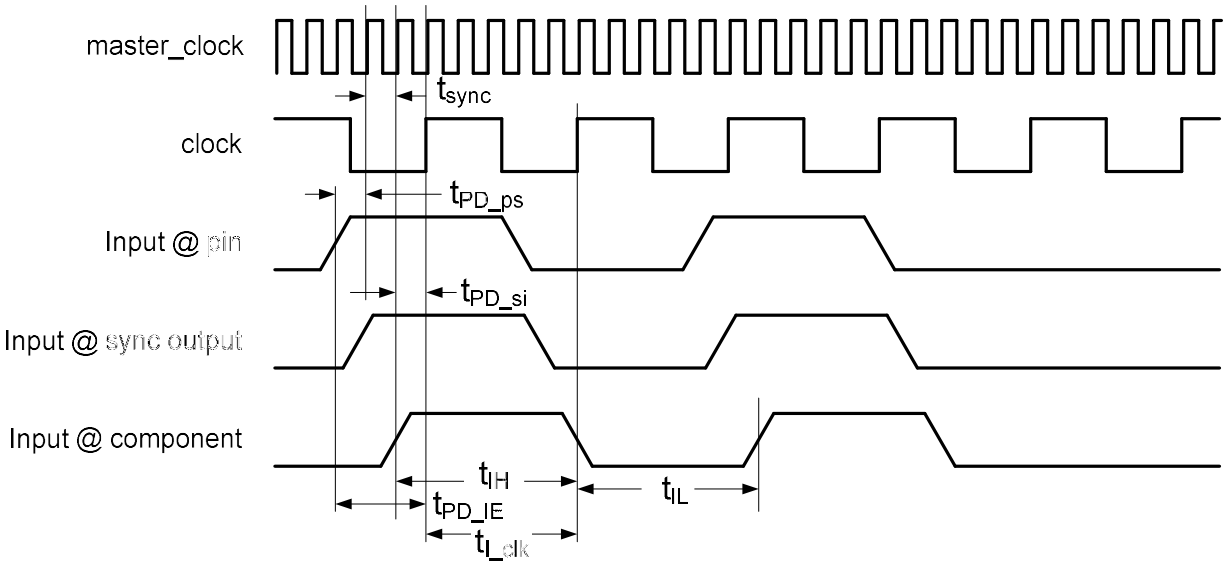


Figure 10. Input Configuration 1; Synchronizer Clock Frequency < Component Clock Frequency

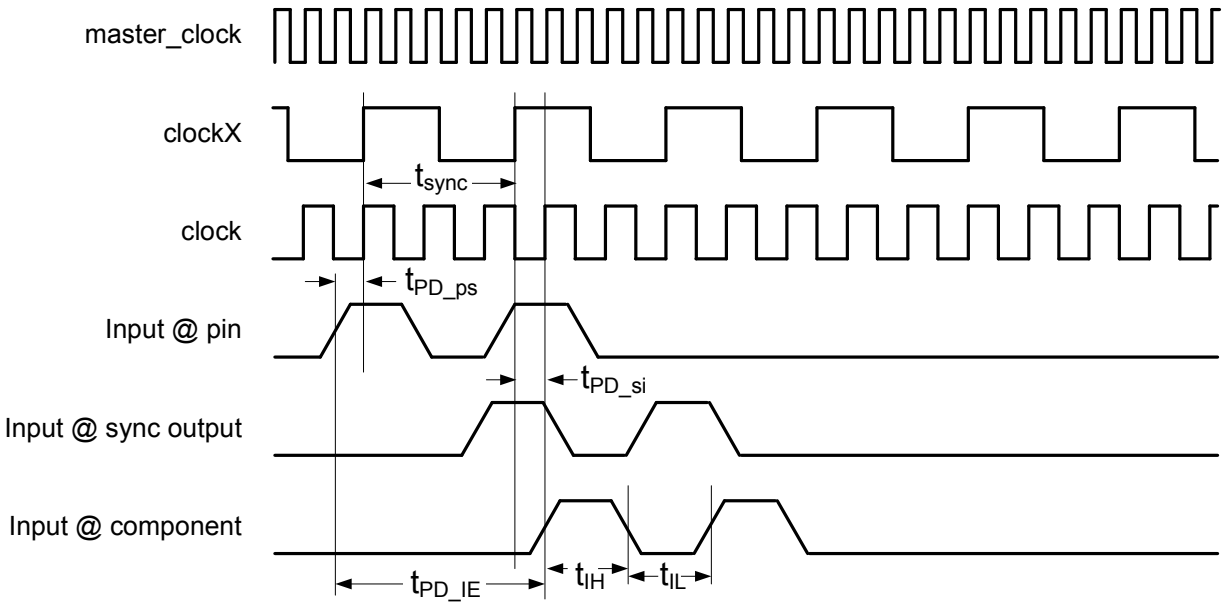
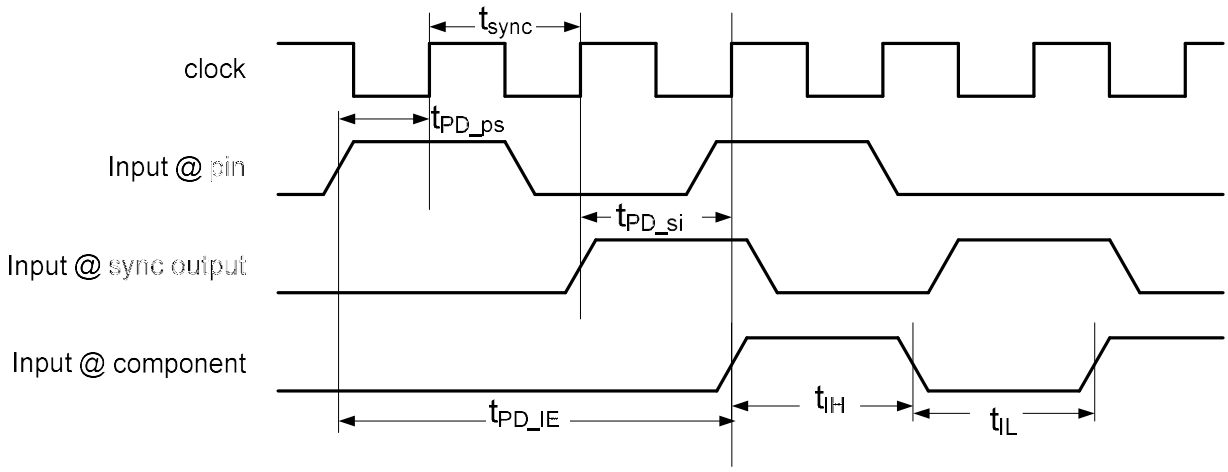


Figure 11. Input Configuration 1 and 2; Synchronizer Clock = Component Clock = master_clock



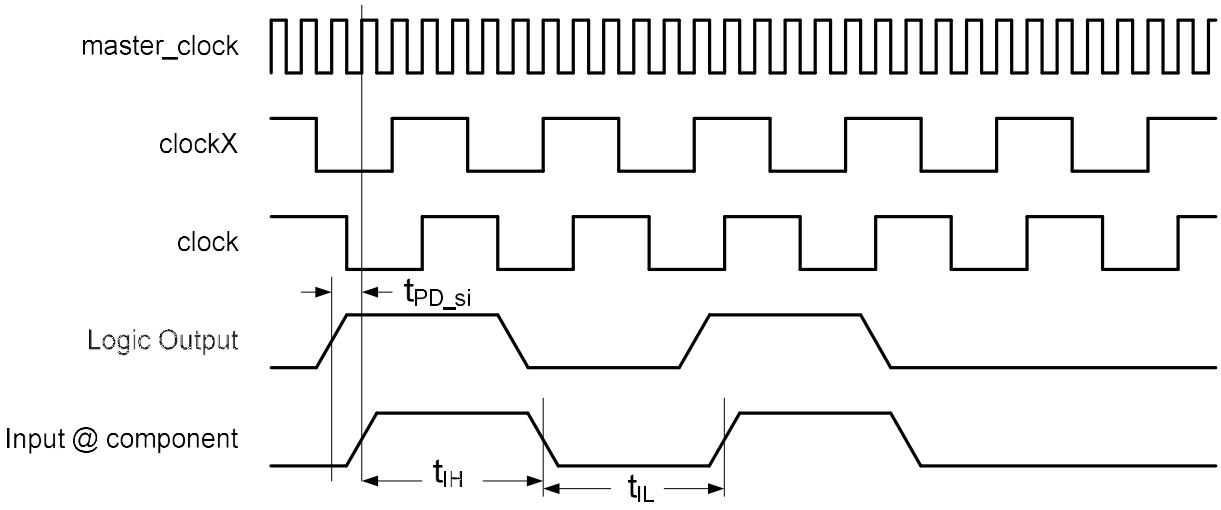
3. The input is driven by logic internal to the PSoC, which is synchronous based on a clock other than the clock the component uses (all internal clocks are derived from master_clock).

   When characterizing inputs configured in this way, the synchronizer clock is faster than, slower than, or equal to the component clock. This produces the characterization parameters shown in Figure 12, Figure 13, and Figure 15.

4. The input is driven by logic internal to the PSoC, which is synchronous based on the same clock the component uses.

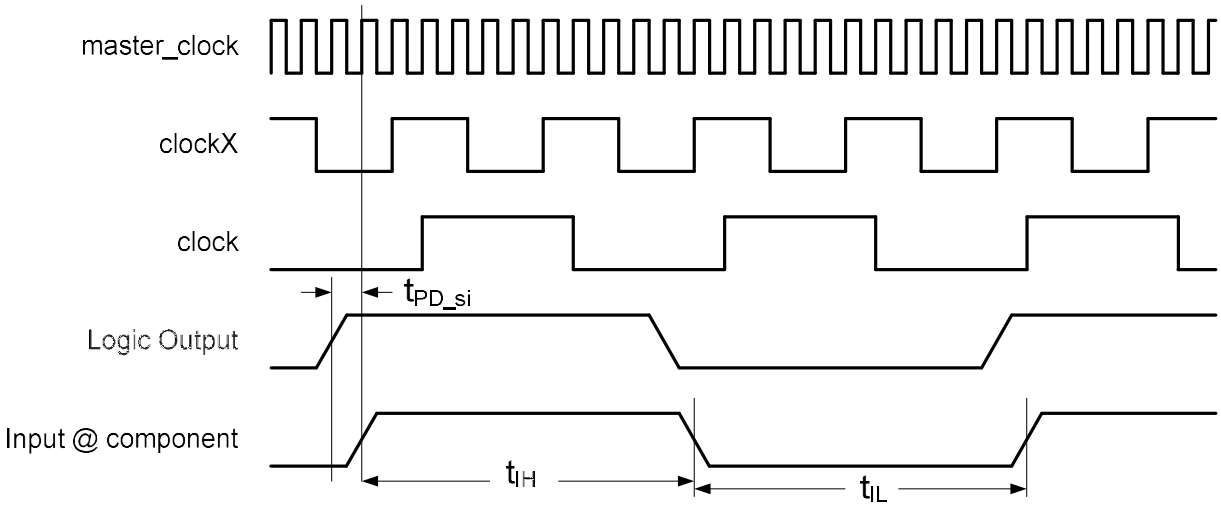
   When characterizing inputs configured in this way, the synchronizer clock is equal to the component clock. This produces the characterization parameters as shown in Figure 16.

Figure 12. Input Configuration 3 only; Synchronizer Clock Frequency = Component Clock Frequency (Edge alignment of clock and clockX is not guaranteed)



This figure represents the understanding that Static Timing Analysis holds about the clocks. All clocks in the digital clock domain are synchronous to master_clock. However, it is possible that two clocks with the same frequency are not rising-edge-aligned. Therefore, the static timing analysis tool does not know which edge the clocks are synchronous to and must assume the minimum of one master_clock cycle. This means that t_{PD_si} now has a limiting effect on master_clock of the system. master_clock setup time violations appear if this path delay is too long. You must change the synchronization clocks of your system or run master_clock at a slower frequency.

Figure 13. Input Configuration 3; Synchronizer Clock Frequency > Component Clock Frequency



In much the same way as shown in Figure 12, all clocks are derived from master_clock. STA indicates the t_{PD_si} limitations on master_clock for one master_clock cycle in this configuration. master_clock setup time violations appear if this path delay is too long. You must change the synchronization clocks of your system or run the master_clock at a slower frequency.

Figure 14. Input Configuration 3; Synchronizer Clock Frequency = master_clock > Component Clock Frequency

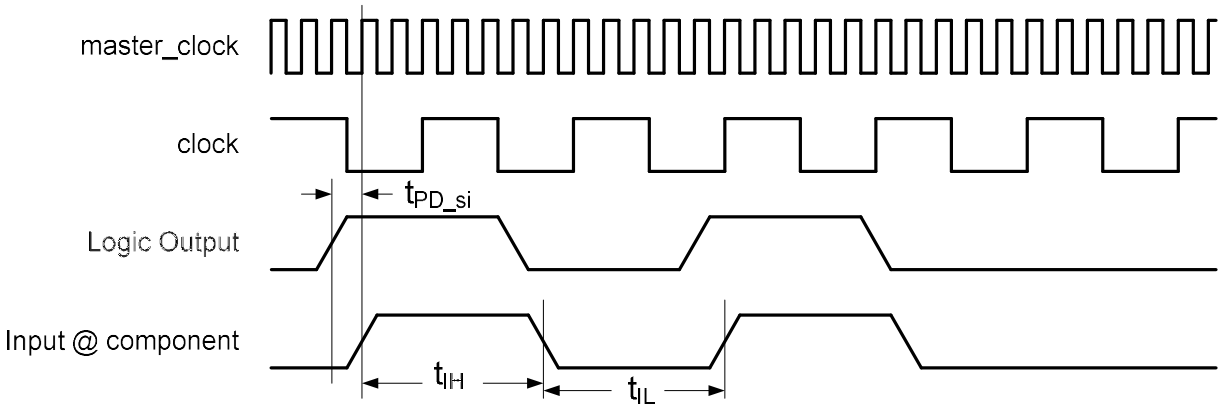
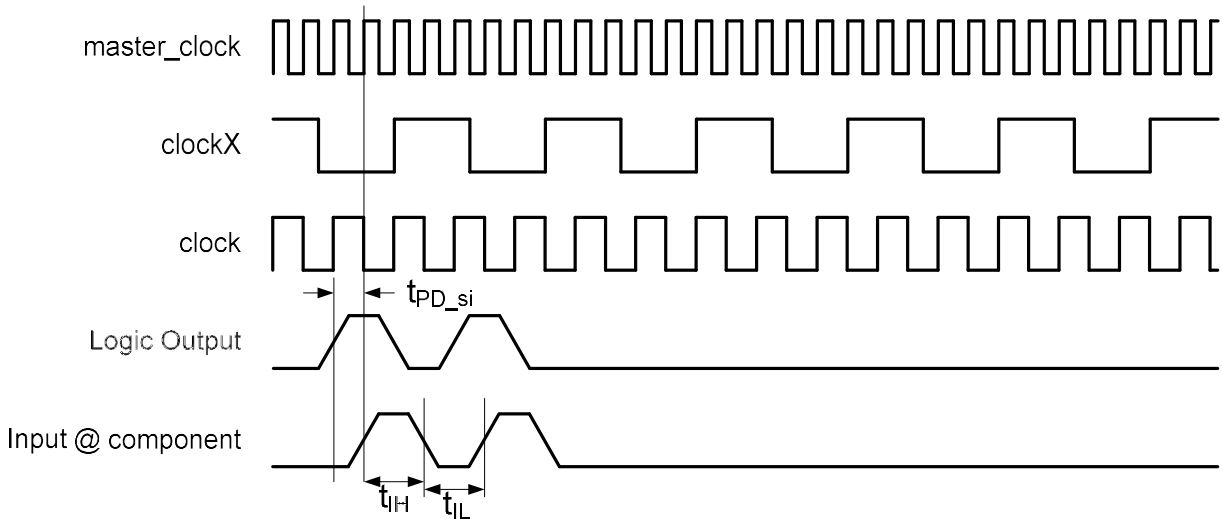
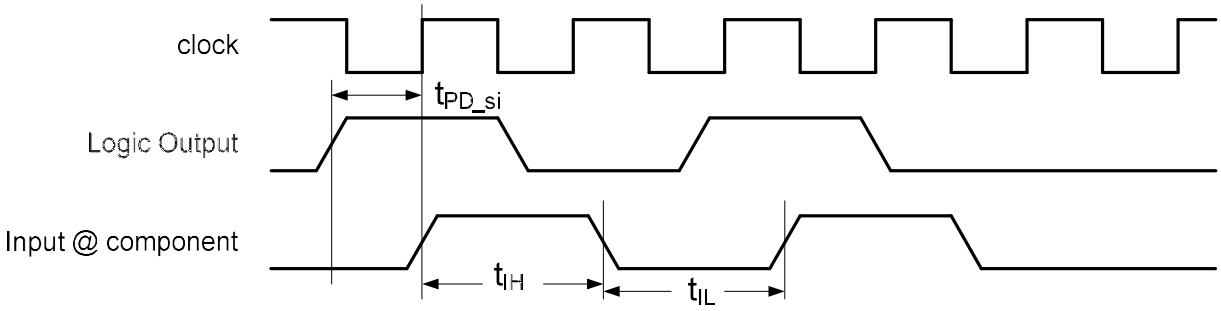


Figure 15. Input Configuration 3; Synchronizer Clock Frequency < Component Clock Frequency



In much the same way as shown in Figure 12, all clocks are derived from master_clock. STA indicates the t_{PD_si} limitations on master_clock for one master_clock cycle in this configuration. master_clock setup time violations appear if this path delay is too long. You must change the synchronization clocks of your system or run master_clock at a slower frequency.

Figure 16. Input Configuration 4 only; Synchronizer Clock = Component Clock



In all previous figures in this section, the most critical parameters to use when understanding your implementation are f_{CLOCK} and t_{PD_IE}. t_{PD_IE} is defined by t_{PD_ps} and t_{SYNC} (for configurations 1 and 2 only), t_{PD_si}, and t_{I_Clk}. It is important to note that t_{PD_si} defines the maximum component clock frequency. t_{I_Clk} does not come from the STA results but is used to represent when t_{PD_IE} is registered. This is the margin left over after the route between the synchronizer and the component clock.

t_{PD_ps} and t_{PD_si} are included in the STA results.

To find t_{PD_ps}, look at the input setup times defined in the *_timing.html* file. The fanout of this input may be more than 1 so you need to evaluate the maximum of these paths.

-Setup times

-Setup times to clock BUS_CLK

Start	Register	Clock	Delay (ns)
input1(0):iocell.pad_in	input1(0):iocell.ind	BUS_CLK	16.500

t_{PD_si} is defined in the Register-to-register times. You need to know the name of the net to use the *_timing.html* file. The fanout of this path may be more than 1 so you need to evaluate the maximum of these paths.

-Register-to-register times

-Destination clock clock

Destination clock clock (Actual freq: 24.000 MHz)

+Source clock clock

-Source clock clock_1

Source clock clock_1 (Actual freq: 24.000 MHz)
Affected clock: BUS_CLK (Actual freq: 24.000 MHz)

Start	End	Period (ns)	Max Freq	Frequency	Violation
\Sync_1:genblk1[0]:INST\:synccell.syncq	\PWM_1:PWMUDB:runmode_enable\:macrocell.mc_d	7.843	127.508 MHz	24.000 MHz	

Output Path Delays

When characterizing the path delays of outputs, you must consider where the output is going in order to know where you can find the data in the STA results. For this component, all outputs are synchronized to the component clock. Outputs fall into one of two categories. The output goes either to another component inside the device, or to a pin to the outside of the device. In the first case, you must look at the Register-to-register times shown for the Logic-to-input descriptions above (the source clock is the component clock). For the second case, you can look at the Clock-to-Output times in the *_timing.html* STA results.

Component Changes

This section lists the major changes in the component from the previous version.

Version	Description of Changes	Reason for Changes / Impact
2.10	Customizer related updates	To fix minor GUI related issues.
	Updated PWM_RestoreConfig() API	To fix an issue with interrupt trigger after wakeup from low power mode.
	Updated PWM_Stop() and PWM_Enable() APIs	To enable the alternate active power mode for FF PWM.
	Updated PWM_SaveConfig() and PWM_RestoreConfig()updates	To restore PWM period register after wakeup.
	Added PSoC 5 FF DC and AC characteristics to datasheet	
2.0.a	Datasheet updates	
2.0	Synchronized inputs	All inputs are synchronized in the fixed function implementation, at the input of the block.
	PWM_GetInterruptSource() function was converted to a Macro	The PWM_GetInterruptSource() function is exactly the same implementation as the PWM_ReadStatusRegister() function. To save code space this was converted to a macro substitution of the PWM_ReadStatusRegister() function.
	Outputs are now Registered to the component clock	To avoid glitches on the outputs of the component it was required that all outputs are synchronized. This is done inside of the Datapath when possible, to avoid excess resource usage.
	Implemented critical regions when writing to Aux Control registers.	CyEnterCriticalSection and CyExitCriticalSections functions are used when writing to Aux Control registers so that it is not modified by any other process thread.
	Added characterization data to datasheet	
	Minor datasheet edits and updates	

© 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™ 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.