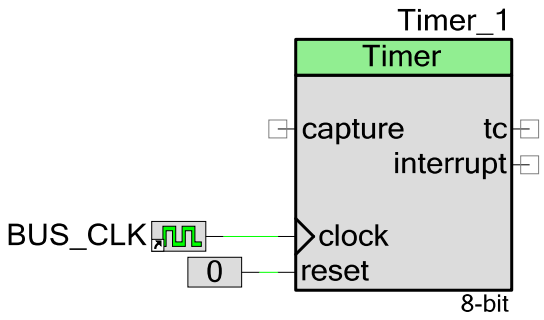
Timer
2.0

Features

- Supports fixed-function and UDB-based implementations
- 8-, 16-, 24-, or 32-Bit Resolution
- Configurable Capture modes
- 4 deep capture FIFO
- Optional capture edge counter
- Configurable Trigger and Interrupts
- Configurable Hardware/Software Enable
- Continuous or 1 shot run modes



General Description

The Timer component provides a capture timer used to time the interval between hardware events. The Timer is designed to provide an easy method of timing complex real time events accurately with minimal CPU intervention. The Timer component features may be combined with other analog and digital components to create complex peripherals.

Timers count only in the down direction starting from the period value and require a single clock input. The input clock period is the minimum time interval able to be measured. The maximum timer measurement interval is the input clock period multiplied by the resolution of the timer. The signal to be captured may be routed from an IO pin or from other internal component outputs. Once started, the Timer component operates continuously and reloads the timer period value on reaching the terminal count.

The Timer component capture input is the most useful feature of the timer. On a capture event the current timer count is copied into a storage location. Firmware may read out the capture value at any time without timing restrictions as long as the capture FIFO has room. You should take care to avoid writing to the FIFO if it is full. If the FIFO is full, the oldest value will be over written and the newly captured value returned in its place the next time the FIFO is read. The Capture FIFO allows storage of up to 4 capture values. The capture event may be generated by software, rising edge, falling edge or all edges allowing great measurement flexibility. To further assist in measurement accuracy of fast signals an optional 7-bit counter may be used to only capture every n[2..127] of the configured edge type.

The trigger and reset inputs allow the Timer component to be synchronized with other internal or external hardware. The optional trigger input is configurable so that a rising edge, falling edge or any edge starts the timer counting. A rising edge on the reset input causes the counter to reset its count as if the terminal count was reached.

An interrupt can be programmed to be generated under any combination of the following conditions; when the Timer component reaches the terminal count, when a capture event occurs or after n[1..4] capture events have occurred. The interrupt signal is a read to clear signal.

When to Use a Timer

A typical use of the Timer is to record the number of clock cycles between events. A common use is to measure the number of clocks between two rising edges as might be generated by a tachometer sensor. A more complex use is to measure the period and duty cycle of a PWM input. For PWM measurement the Timer component is configured to start on a rising edge, capture the next falling edge and then capture and stop on the next rising edge. An interrupt on the final capture signals the CPU that all the captured values are ready in the FIFO. The Timer component can be used as a clock divider by driving a clock into the clock input and using the terminal count output as the divided clock output.

Timers share many features with counters and PWMs. 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 compare output. A PWM component is better used in situations requiring multiple compare outputs with control features like center alignment, output kill and deadband outputs.

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.

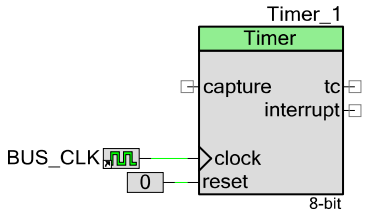
Input	May Be Hidden	Description
clock	N	The clock input defines the operating frequency of the Timer component. That is, the timer period counter value is decremented on the rising edge of this input while the Timer component is enabled.
reset	N	This input is a synchronous reset requiring at least one rising edge of the clock to implement the resets of the counter value and the capture counter. It resets the period counter to the period value. It also resets the capture counter. **Note** For PSoC 3 ES2 silicon, the Terminal Count pin for the fixed function Timer is held HIGH in Reset. A schematic fix for this is provided under "Reset in Fixed Function Block" in the Functional Description section of this data sheet. For PSoC 3 ES3 or later silicon, the Terminal Count pin for the fixed function Timer is held LOW in Reset

Input	May Be Hidden	Description
enable	Y	Hardware enable of the Timer component. This connection enables the period counter to decrement on each rising edge of the clock. If this input is low the outputs are still active but the Timer component does not change states.
capture	Y	Captures the period counter value to a 4-sample FIFO in the UDB, or to a single sample register in the Fixed Function block. The input pin is visible if enabled by the **Capture Mode** parameter as set in the Configure dialog. No values will be captured if the Timer is disabled.
trigger	Y	This input only displays when the **Trigger Mode** parameter is enabled in the Configure dialog. If the **Trigger Mode** parameter is set to "None" it causes the Timer to operate by beginning to down count from the period as soon as the block is started. If the parameter is set to one of the edge types, it causes the Timer to delay the start of the Period down count until the appropriate edge type is detected. The trigger edge is not captured nor does it generate an interrupt. Typically the trigger and capture inputs are tied together but may be used separately for design flexibility.

Output	May Be Hidden	Description
tc	N	Terminal count output goes high if the Timer component has started and the count value is equal to the terminal count (zero). The terminal count output is a zero compare of the period counter value. As long as the period counter is zero and the Timer component is enabled, the output will be high. This output is synchronized to the block clock input of the component.
interrupt	N	The interrupt output is a copy of the interrupt source configured in the hardware. The sources of the interrupt are configured through software as being any of the status bits: • Terminal Count Event • Capture Event • Capture FIFO Full (UDB implementation only) Once an interrupt is triggered, the interrupt output shall remain asserted until the Status Register is read out by the software. In order to receive subsequent interrupts, the interrupt shall be cleared by reading the Status Register using the Timer_ReadStatusRegister() API.
capture_out	Y	The capture_out output is an indicator of when a hardware capture has been triggered. The output pin is available for the UDB implementation only. This output is synchronized to the block clock input of the component.

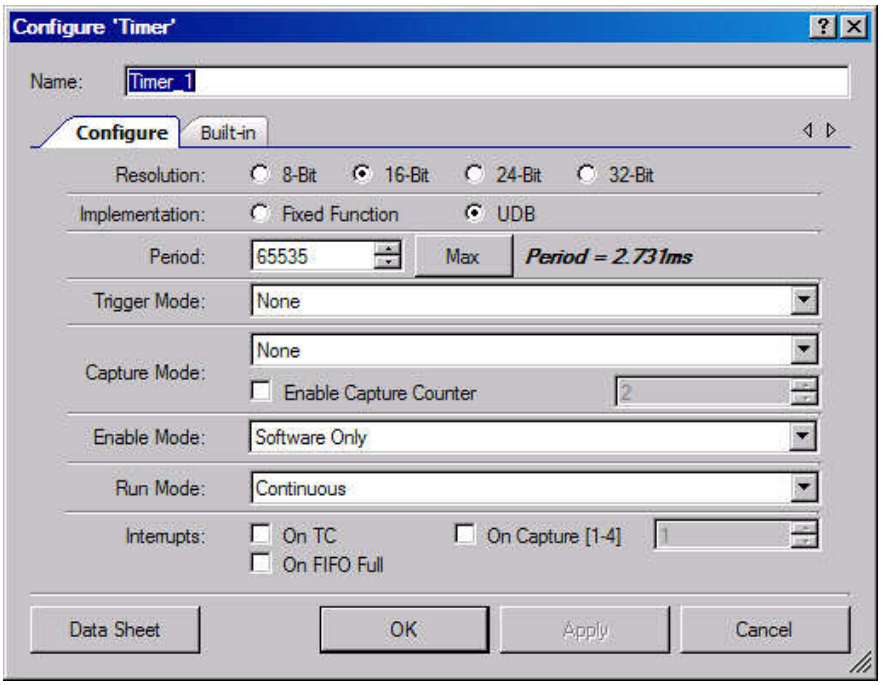
Schematic Macro Information

The default Timer in the Component Catalog is a schematic macro using a Timer component with default settings. It is connected to a 24 MHz bus clock and a Logic Low component.



Parameters and Setup

Drag a Timer component onto your design and double-click it to open the Configure dialog.



Hardware vs. 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 API provided. Most parameters described in the next sections are hardware options. The software options will be noted as such.

Resolution

The Resolution parameter defines the bit-width resolution of the Timer. The default is 8-bit.

Implementation

The Implementation parameter allows you to choose either a fixed function block (default) or a UDB implementation of the Timer.

Period (Software)

The **Period** parameter defines the max counts value (or rollover point) for the period counter. The default is 256. The limits of this value are defined by the **Resolution** parameter. The maximum value of the **Period** value is defined as (2^8), (2^16), (2^24), and (2^32), as shown in the following table. The **Period** value may be modified by the WritePeriod() API.

Resolution	Maximum period count values
8	256
16	65536
24	16777216
32	4294967296

Trigger Mode (Software)

The Trigger Mode parameter configures the implementation of the trigger input. This parameter is only active when Implementation is set to "UDB." This parameter is not available when set to "Fixed Function."

This value is an enumerated type and can be set to any of the following values:

- “None” (default): No trigger implemented and the trigger input pin is hidden
- “Rising Edge”: Trigger (enable) the period counter value on the first rising edge of the trigger input
- “Falling Edge”: Trigger (enable) the period counter value on the first falling edge of the trigger input
- “Either Edge”: Trigger (enable) the period counter value on the first edge of the trigger input
- “Software Controlled”: Control register bits define what edge of the trigger input to use to trigger the timer. May be changed at any time by API by setting calling the Timer_SetTriggerMode() function

Capture Mode (Software)

The Capture Mode section contains three parameters: Capture Mode Value, Enable Capture Counter, and Capture Count.

Capture Mode Value

The Capture Mode Value Parameter configures the implementation of the capture input. This parameter is available in the fixed function timer implementation but is not configurable to which edge. All capture on the fixed function block is implemented on the rising edge of the capture input.

This value is an enumerated type and can be set to any of the following:

- “None”: No capture implemented and the capture input pin is hidden
- “Rising Edge” (default): Capture the period counter value on any rising edge of the capture input
- “Falling Edge”: Capture the period counter value on any falling edge of the capture input
- “Either Edge”: Capture the period counter value on any edge of the capture input
- “Software Controlled”: Control register bits define what edge of the capture input to use to capture data to the FIFO. May be changed at any time by API by setting calling the Timer_SetCaptureMode() function

Enable Capture Counter (Software)

The Capture Counter Enabled parameter allows you to implement a 7-bit counter that is accessible by software to define how many capture events happen before the period counter is captured to the FIFO. It may be necessary to capture every 3rd event in which case the capture counter should be set to a value of 3. If this parameter is set the 7-bit counter may be changed at any time with the API Timer_SetCaptureCount().

Capture Count (Software)

The Capture Count parameter configures the initial value in the capture counter. The capture counter allows for 2-127 capture events to happen before the period counter value is captured to the data FIFO. If the capture count is set to 100 then every 100th capture event will capture the period counter to the FIFO. The capture count value may be modified by the API Timer_SetCaptureCount().

Enable Mode

This parameter specifies the mode of the component:

- “Software Only”: The Timer is enabled only by setting the enable bit in the control register. In this mode the enable input pin will be hidden from the symbol.

- “Hardware Only”: The Timer is enabled only by setting the enable bit in the control register. In this mode the control register may be removed from the implementation but the control register enable bit has no affect on operation. This option is only available when Implementation is set to "UDB."
- “Hardware and Software”: The Timer is enabled only if both the control register bit and the hardware input are active (high).

Run Mode

The **Run Mode** parameter allows you to configure the Timer component to run continuously or in one of two one shot modes:

- “Continuous”: The Timer will run so long as the enable conditions are true
- “One Shot”: The Timer will run through a single period and stops after terminal count. On stop, it reloads period into period counter. It will begin another single cycle if the period counter is reset or Timer component is hardware reset.
- “One Shot halt on Interrupt”: The Timer will run through a single period. Timer stops on occurrence of any interrupt or after terminal count. If the Timer reaches terminal count after stop, it reloads period into period counter. It will begin another single cycle if the period counter is reset or Timer component is hardware reset.

  Note In order to be sure that One Shot mode does not start prematurely, you should use a Trigger Mode to control the start time, or use some form of software enable mode ("Software Only" or "Hardware and Software").

Interrupt (Software)

The Interrupt section contains various "Interrupt On" parameters. These values are ORed with any of the other “Interrupt On” parameters to give a final group of events that can trigger an interrupt. This configures the startup setting, it may be modified at any time with the API Timer_SetInterruptMode().

- **On TC –** Allows you to interrupt on a terminal count.
- **On Capture –** Allows you to configure a valid capture as an interrupt source.
  - **Number Of Captures –** This field is used to specify the number of captures to count before an interrupt on capture is triggered. This allows software to deal with capture data only after the data expected is available and to not be overworked by calling the ISR too often. This value can be set to a value from 1 to 4. It may also be set at any time with the API Timer_SetInterruptCount().
- **On FIFO Full –** The Interrupt on FIFO Full parameter allows you to interrupt when the capture FIFO is full.

Clock Selection

For the Timer component, the clock input can be any signal for which you wish to count the rising edges. It is expected that this input is periodic with its frequency, in combination with the period counts definition of your timer, defining the output period of your timer.

WARNING When configured to utilize the fixed function block in the device, the Timer component will have the following restrictions:

1. 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 **Source** as "BUS_CLK").
2. 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 **Source** as "BUS_CLK"). A local clock with a frequency that matches the bus clock will generate 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* to clock the component, the frequency of the routed clock cannot exceed 1/2 the routed clock's source clock frequency.

- If the routed clock is synchronous to the bus clock, then it is 1/2 the bus clock.
- If the routed clock is synchronous to one of the clock dividers, its maximum is 1/2 of that clock rate.

Placement

The Timer component is placed based on the FixedFunction parameter. Whether you set this option or it is set by the auto placement operation, if the FixedFunction is set then this component will be placed in an available fixed function Counter/Timer block; otherwise it will be placed in the UDB array as determined for best placement for the whole design.

* 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 UDB Timer	1	6*	1	1	0	423	5	-
8 bits FF Timer	0	0	0	0	0	364	2	-
16-Bits UDB Timer	2	6*	1	1	0	465	6	-
16-Bits FF Timer	0	0	0	0	0	393	2	
24-Bits UDB Timer	3	6*	1	1	0	461	8	-
32-Bits UDB Timer	4	6*	1	1	0	461	8	-
8 bit UDB Timer One Shot	1	8*	1	1	0	354	5	-
16 bit UDB Timer One Shot	2	8*	1	1	0	387	6	-

* The enable mode is set to Software Only and Capture mode set to None.

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 “Timer_1” to the first instance of a component in a given design. You can rename it 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 “Timer”.

Function	Description
void Timer_Start(void)	Initialize the Timer with default customizer values. Enable the Timer operation by setting the enable bit of the control register for either of the software controlled enable modes.

Function	Description
void Timer_Stop(void)	Disable the Timer operation; Clears the enable bit of the control register for either of the software controlled enable modes.
void Timer_SetInterruptMode (uint8 interruptMode)	Enables or disables the sources of the interrupt output from the optional interrupt sources defined by the bits in the status register.
void Timer_SetCaptureMode (uint8 captureMode)	Sets the capture mode of the Timer from one of the enumerated type options available
void Timer_SetTriggerMode (uint8 triggerMode)	Sets the trigger mode of the timers trigger input from one of the enumerated type options available
void Timer_EnableTrigger(void)	Enables the trigger mode of the timer by setting the correct bit in the control register
void Timer_DisableTrigger(void)	Disables the trigger mode of the timer by clearing the correct bit in the control register
void Timer_SetInterruptCount (uint8 interruptCount)	Sets the number of captures to count before an interrupt is triggered for the InterruptOnCapture source. Only available in a UDB implementation.
void Timer_SetCaptureCount (uint8 captureCount)	Sets the number of captures events to count before actually capturing the Timer value to the FIFO. Only applicable in the UDB implementation.
uint8 Timer_ReadCaptureCount(void)	Reads the current value of the NumberOfCaptures definition which defines the number of captures counted before an interrupt is triggered for the InterruptOnCapture source
void Timer_SoftwareCapture(void)	Forces a capture of the period counter to the capture FIFO
uint8 Timer_ReadStatusRegister(void)	Reads the status register and returns its state. This function should use defined types for the bit-field information as the bits in this register may be permutable.
uint8 Timer_ReadControlRegister(void)	Reads the control register and returns its state. This function should use defined types for the bit-field information as the bits in this register may be permutable.
void Timer_WriteControlRegister (uint8 control)	Sets the bit-field of the control register. This function should use defined types for the bit-field information as the bits in this register may be permutable.
uint8/16/32 Timer_ReadPeriod(void)	Reads the Period register returning the last period value written to it
void Timer_WritePeriod (uint8/16/32 period)	Writes the Period register with the new desired period or max counts value
uint8/16/32 Timer_ReadCounter(void)	Forces a software capture of the period value into the capture FIFO and oldest data from the capture FIFO
void Timer_WriteCounter (uint8/16/32 counter)	Allows the user to overwrite the counter value as a new value to count down from. Called during initialization to preload the period value.

Function	Description
uint8/16/32 Timer_ReadCapture(void)	Reads the latest captured period counter value. Firmware must check the FIFO status for data before reading
void Timer_ClearFIFO(void)	Clears all previous capture data from the capture FIFO
void Timer_Sleep(void)	Stops the Timer operation and saves the user configuration
void Timer_Wakeup(void)	Restores and enables the user configuration
void Timer_Init(void)	Initializes or Restores default Timer configuration
void Timer_Enable(void)	Enables the Timer.
void Timer_SaveConfig(void)	Saves the configuration of Timer.
void Timer_RestoreConfig(void)	Restores the configuration of Timer.

Global Variables

Variable	Description
Timer_initVar	Indicates whether the Timer has been initialized. The variable is initialized to 0 and set to 1 the first time Timer_Start() is called. This allows the component to restart without reinitialization in after the first call to the Timer_Start() routine. If reinitialization of the component is required, then the Timer_Init() function can be called before the Timer_Start() or Timer_Enable() function.

void Timer_Start(void)

Description: Initializes the Timer with default customizer values. Enables the Timer 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 Timer. If the Enable Mode is set to hardware only this has no affect on the Timer. 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 Timer.

void Timer_Stop(void)

Description: Disable the Timer operation by resetting the 7th bit of the control register for either of the software controlled enable modes. Disables the Fixed function block which had been chosen.

Parameters: None

Return Value: None

Side Effects: Clears the enable bit in the control register of the counter.

void Timer_SetInterruptMode(uint8 interruptMode)

Description: Enables or disables the sources of the interrupt output from the optional interrupt sources defined by the bits in the status register.

Parameters: uint8: interruptMode – Bit-Field containing the status bits you want enabled as interrupt sources. This parameter should be an ORing of the desired status bit masks defined in the *Timer.h* header file.

Return Value: None

Side Effects: All interrupt sources are ORed together to provide a single interrupt output. You must call the Timer_ReadStatusRegister() API to review which enabled status bit caused the interrupt and to clear the interrupt as they are sticky bits in the status register.

void Timer_SetCaptureMode(uint8 captureMode)

Description: Sets the capture mode of the Timer component from one of the enumerated type options available. The 5th and 6th bits of the control register are cleared and rewritten with the new user configuration.

Parameters: enum: captureMode – This parameter should be defined using one of the capture mode constants defined in the Timer.h header file. Available enum values are:

```
Timer__B_TIMER__CM_NONE
Timer__B_TIMER__CM_RISINGEDGE
Timer__B_TIMER__CM_FALLINGEDGE
Timer__B_TIMER__CM_EITHEREDGE
Timer__B_TIMER__CM_SOFTWARE
```

Return Value: None

Side Effects: Only available if the Capture Mode parameter is set to Software Controlled on the Configure dialog. Resource usage may be minimized by not allowing software control of the capture mode and the API resource is minimized by optimizing out.

void Timer_SetTriggerMode(uint8 triggerMode)

Description: Sets the trigger mode of the timer from one of the enumerated type options available.

Parameters: enum: triggerMode – This parameter should be defined using one of the trigger mode constants defined in the *Timer.h* header file.
Available enum values are:

```
Timer__B_TIMER__TM_NONE
Timer__B_TIMER__TM_RISINGEDGE
Timer__B_TIMER__TM_FALLINGEDGE
Timer__B_TIMER__TM_EITHEREDGE
Timer__B_TIMER__TM_SOFTWARE
```

Return Value: None

Side Effects: Only available if the trigger mode is set to Software Controlled. Resource usage may be minimized by not allowing software control of the trigger mode and the API resource is minimized by optimizing out.

void Timer_EnableTrigger(void)

Description: Enables the trigger mode of the timer by setting the 4th bit in the control register.

Parameters: None

Return Value: None

Side Effects: None

void Timer_DisableTrigger(void)

Description: Disables the trigger mode of the timer by resetting the 4th bit in the control register.

Parameters: None

Return Value: None

Side Effects: None

void Timer_SetInterruptCount(uint8 interruptCount)

Description: Sets the number of captures to count before an interrupt is triggered for the InterruptOnCapture source. This function is available only when InterruptOnCaptureCount is enabled and the control register is not removed.

Parameters: uint8: interruptCount – The desired number of capture events to count before the interrupt is on capture event is triggered to the interrupt output. A value between 0-3 is valid.

Return Value: None

Side Effects: None

void Timer_SetCaptureCount(uint8 captureCount)

Description: Sets the number of capture events to count before a capture is actually performed. This function is only available if the **Enable Capture Counter** parameter is selected on the Configure dialog.

Parameters: uint8: captureCount – The desired number of capture events to count before capturing the counter value to the capture FIFO

Return Value: None

Side Effects: None

uint8 Timer_ReadCaptureCount(void)

Description: Reads the current value set for the captureCount parameter as set in the SetCaptureCount function. This function is only available if the **Enable Capture Counter** parameter is selected on the Configure dialog.

Parameters: None

Return Value: uint8: current capture count

Side Effects: None

void Timer_SoftwareCapture(void)

Description: Forces a software capture of the period counter value to the FIFO

Parameters: None

Return Value: none:

Side Effects: Pushes another value onto the capture FIFO

uint8 Timer_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:
[7:4] : Unused (0)
[3] : FIFO not empty status
[2] : FIFO full status
[1] : Registered Capture value
[0] : Terminal count

Side Effects: Interrupt bits in the status register are clear on read.

uint8 Timer_ReadControlRegister(void)

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

Parameters: None

Return Value: uint8: Current control register value.

Side Effects: None

void Timer_WriteControlRegister(uint8 control)

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

Parameters: uint8: Control – Control register Bit-Field. This parameter should be an ORed grouping of the control register constants. The control register bits are:
[7] : Timer Enable
[6:5] : Capture Mode select
[4] : Trigger Enable
[3:2] : Trigger Mode select
[1:0] : Interrupt count

Return Value: None

Side Effects: None

uint8/16/32 Timer_ReadPeriod(void)

Description: Reads the Period register returning the last period value written to it.

Parameters: None

Return Value: uint8/16/32: Period Value

Side Effects: None

void Timer_WritePeriod(uint8/16/32 period)

Description: Writes the Period register with the new desired period or max counts value.

Parameters: uint8/16/32: Period – New Period Value

Return Value: None

Side Effects: This period value will not be implemented until the current Timer period is complete (that is, at TC). When TC is reached the new period value will be loaded into the counter and the new period time will take affect until the period value is written over.
Note If the desired period from the Timer is 'n' times units period, the value passed to this API must be 'n-1'.

uint8/16/32 Timer_ReadCounter(void)

Description: Forces a software capture of the period value into the capture FIFO and oldest data from the capture FIFO

Parameters: None

Return Value: uint8/16/32: Counter Value

Side Effects: If there was already capture data in the FIFO before this function is called then the existing data will be returned and the forced software capture value will be added to the FIFO and may be read later with a Timer_ReadCapture() call. The user should check the status of the FIFO before calling the Timer_ReadCounter() API to avoid reading unexpected data from the FIFO.

void Timer_WriteCounter(uint8/16/32 counter)

Description: Writes a value to the counter enabling the user to restart the counter from any value they deem necessary.

Parameters: uint8/16/32: Counter – New Counter Value

Return Value: None

Side Effects: None

uint8/16/32 Timer_ReadCapture(void)

Description: Reads the latest captured period counter value. If no value has been captured to the FIFO then this value will be the current period counter value.

Parameters: None

Return Value: uint8/16/32: Counter or Capture Value

Side Effects: Once the FIFO is fully read/emptied or it is cleared using Timer_ClearFIFO() and no more captures are taking place, then you can still read the capture FIFO content using the Timer_ReadCapture() API. Timer_ReadCapture() always returns a value from top of the FIFO because it returns TIMER_LSB_PTR. If you use this API, check for capture FIFO empty status before calling this API.

void Timer_ClearFIFO(void)

Description: Clears the capture FIFO of any previously captured data. Here Read capture is called until FIFO becomes empty.

Parameters: None

Return Value: None

Side Effects: None

void Timer_Sleep(void)

Description: Stops the Timer operation and saves the user configuration. If Control Register is being used the enable state of the Timer is saved.

Parameters: None

Return Value: None

Side Effects: None

void Timer_Wakeup(void)

Description: Restores and enables the user configuration.

Parameters: None

Return Value: None

Side Effects: None

void Timer_Init(void)

Description: Initialize/restore default Timer configuration. The Timer starts in "Continuous" run mode by default. The interrupts are chosen as the output from the status register. If UDB mode is set, then the default Compare mode and Capture mode values are set as defined per the user configuration. The FIFO is cleared in case of UDB mode.

Parameters: None

Return Value: None

Side Effects: All registers will be set to their initial values and FIFO is cleared. This will reinitialize the component.

void Timer_Enable(void)

Description: Enables the Timer by setting 7th bit of the control register for either of the software controlled enable modes. In case of Fixed function mode of the Timer, the chosen fixed function block is enabled.

Parameters: None

Return Value: None

Side Effects: If the Enable mode is set to Hardware only then this function has no effect on the operation of the Timer.

void Timer_SaveConfig(void)

Description: Saves the configuration of Timer. Counter and capture values are saved. In UDB mode the control register value is saved if it’s not removed.

Parameters: None

Return Value: None

Side Effects: None

void Timer_RestoreConfig(void)

Description: Restores the configuration of Timer.

Parameters: None

Return Value: None

Side Effects: Calling this function without calling Timer_SaveConfig() may produce unexpected behavior.

Sample Firmware Source Code

PSoC Creator provides numerous example projects that include schematics and example code in the Find Example Project dialog. For component-specific examples, open the dialog from the Component Catalog or an instance of the component in a schematic. For general examples,

open the dialog from the Start Page or **File** menu. As needed, use the **Filter Options** in the dialog to narrow the list of projects available to select.

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

Functional Description

As described previously the Timer component can be configured for multiple uses. This section will describe those configurations in more detail. Before digging into the possible configurations it is important to know the limitations of using the fixed function timer versus the UDB implementation.

Fixed Function Block Limitations

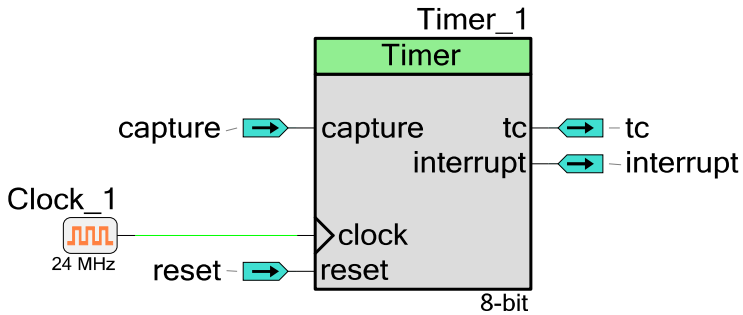
The Counter, Timer, and PWM components have very similar internal requirements that are implemented as fixed function blocks in the chip. There are a few configuration options for one of these blocks, and the limitations of this block as a timer versus the UDB implementation are listed below. The fixed function timer:

- Is 8 or 16-bits only
- Interrupts on Terminal Count and/or Capture only.
- Captures on Rising Edge only.
- Must be run in continuous mode; As a corollary, no trigger mode is available.
- Disables the 7-bit Capture Counter.
- Has a single sample register instead of a 4 sample FIFO for captured values.

Default Configuration

The default configuration of the Timer component provides the most basic timer which simply decrements a period count value on every rising edge of the clock input. With this configuration the component symbol will look like this:

Figure 1: Default Timer Configuration

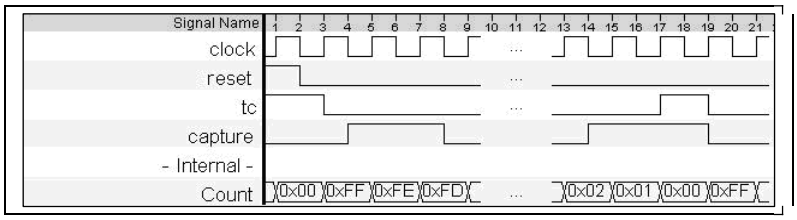


Terminal count indicates in real time whether the counter value is at the terminal count (zero). The period is programmable to be any value from 1 to (2 ^ Resolution) -1.

By default the capture functionality is configured to capture on every rising edge of the capture input. Since the default configuration is using the fixed function block the only option for capture mode is rising edge. Other modes are available when the implementation is changed to the UDB selection.

The following is a waveform showing the expected results.

Figure 2: Default Timer Implementation Example Waveform



Fixed Function Configuration

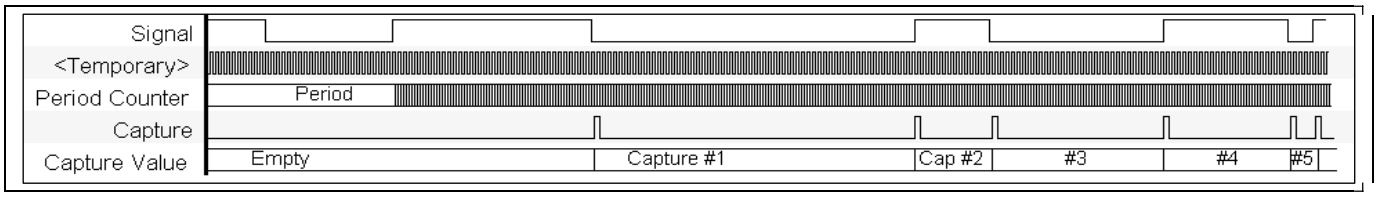
When configured to use the Fixed Function block for the Timer implementation, the Timer component is limited in placement options to one of the Fixed Function blocks on the chip. You should consider your resource needs when choosing to implement the Timer component in the fixed function block as an 8-bit timer placed in the Fixed Function block wastes half of the fixed function block.

High/Low Time Measure Mode

It is often important to measure the high and low times of a signal. The Timer can be configured to make this implementation much simpler. By configuring the Trigger Mode as “Rising Edge" and the Capture Mode as "Either Edge," the Timer will start on the first rising edge at the Period value and count down capturing each edge of the input signal after that.

As long as data is read in a timely manner from the capture FIFO, the calculations for High and Low time will be as follows:

Figure 3 High/Low Time Calculations

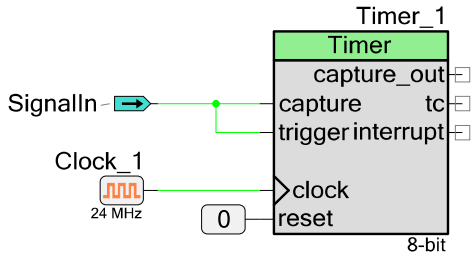


1. High Time #1 = (Period – Capture #1) * Clock Frequency
2. Low Time #1 = (Capture #1 – Capture #2) * Clock Frequency

3. High Time #2 = (Capture #2 – Capture #3) * Clock Frequency
4. Etc.

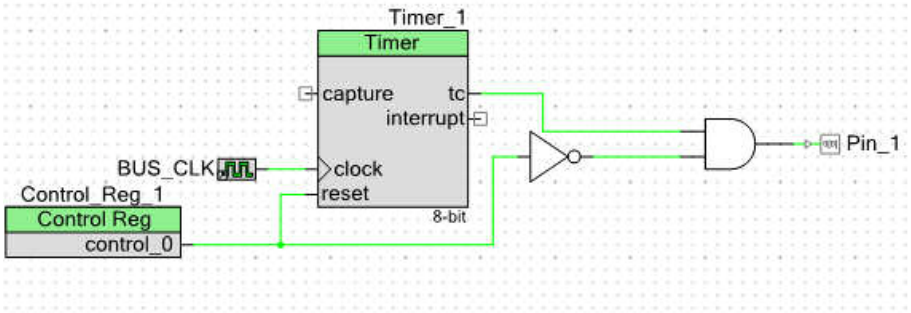
With the following Schematic implementation, setting the Trigger Mode to "Falling Edge" will first measure the low time and continue with alternating edge types until the timer is enabled or reset:

Figure 4 Timer Schematic



Reset in Fixed Function Block

On PSoC3 ES2 silicon, the fixed function implementation of the Timer differs from the UDB implementation in that the TC during reset goes high, whereas in the UDB implementation the TC goes low. The schematic below shows a Fixed Function Timer implementation which drives the TC low while the reset input is active, thus giving the same functionality as the UDB implementation of the same component.



Block Diagram and Configuration

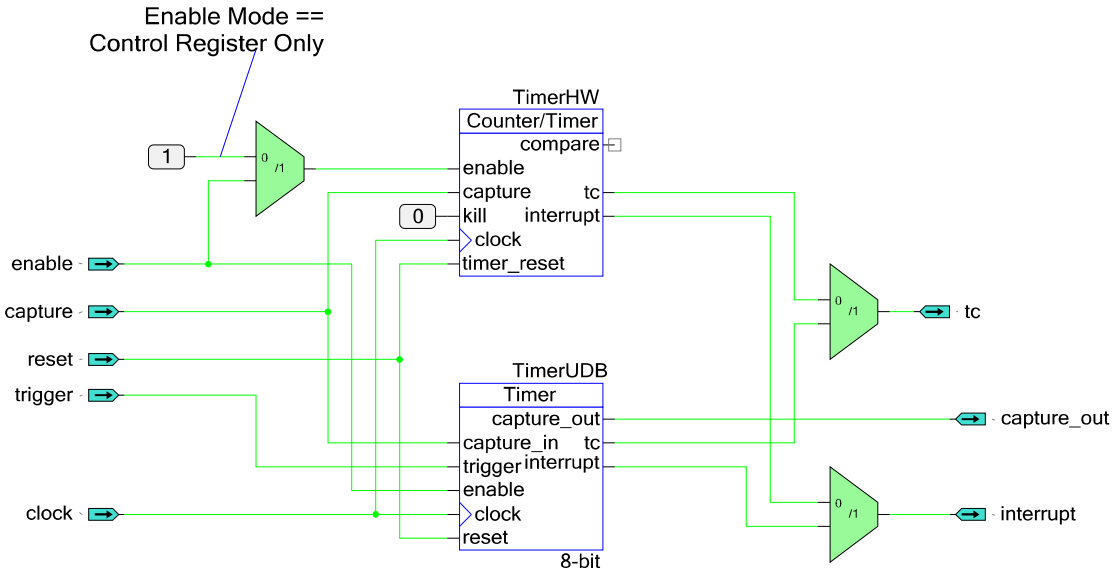
The Timer component may be implemented using a fixed function block or using UDB components. An advance parameter “Implementation” (FixedFunction) allows you to specify the block that you expect this component to be placed in or you may choose the Auto implementation, which will select either of these implementations at build time based on the components and parameters of your entire design.

The Fixed function implementation will consume one of the Timer/Counter/PWM blocks defined in the TRM. In either the fixed function or UDB configuration all of the registers and API are

consolidated to give a single entity look and feel. The API is described in the previous section and the registers are described here to define the overall implementation of the Timer.

The two hardware implementations you chose are selected from a top level schematic as shown in the following diagram:

Figure 5 Timer Implementations Schematic



This configuration allows for either the Fixed Function block or the UDB implementation to be selected and the extra pieces such as the internal interrupt and the routing of the I/O are handled in the background to give this single component look and feel.

Registers

Timer_Status

The status register is a read only register which contains the various status bits defined for the Timer component. The value of this register is available with the Timer_ReadStatusRegister() function call. 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 Timer_SetInterruptMode() function call and upon receiving an interrupt you can retrieve the interrupt source by reading the Status register with the Timer_ReadStatusRegister() function call. The Status register is a clear on read register so the interrupt source is held until the Timer_ReadStatusRegister() function is called. The Timer_ReadStatusRegister() API will handle which interrupts are enabled to provide an accurate report of what the actual source of the interrupt was. All operations on the status register must use the following defines for the bit-fields as these bit-fields may be moved around within the status register during place and route.

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 listed below.

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:

- Timer_STATUS_TC * – Indicates a terminal count has been reached. This bit may be used as an interrupt source.
- Timer_STATUS_CAPTURE * – Indicates a Capture event has been triggered. This bit may be used as an interrupt source.
- Timer_STATUS_FIFO_FULL – Indicates that the capture FIFO is full. This bit may be used as an interrupt source.
- Timer_STATUS_FIFO_NEMPTY – Indicates that the capture FIFO is not empty.

Note The Timer_STATUS_FIFO_FULL and Timer_STATUS_FIFO_NEMPTY bits are not available for the Fixed-Function implementation of the Timer component since there is no FIFO available.

Timer_Control

The Control register allows you to control the general operation of the counter. This register is written with the Timer_WriteControl() function call and read with the Timer_ReadControl(). 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:

- Timer_CTRL_INTCNT – The interrupt count control is a 3-bit field that allows you to configure the number of captures to count before an interrupt is triggered. The value of this bit-field defines a value of 1-4 (as 0-3 + 1) for the number of capture events to count. Set this value by calling the Timer_SetInterruptCount() function with a value of 0-3.
- Timer_CTRL_TRIGPOL – The trigger polarity mode control is a 2-bit field used to define the expected trigger input operation. This bit-field will be 2 consecutive bits in the control register and all operations on this bit-field must use the #defines associated with the capture types available. These are:
  - Timer__B_TIMER__TM_NONE
  - Timer__B_TIMER__TM_RISINGEDGE
  - Timer__B_TIMER__TM_FALLINGEDGE
  - Timer__B_TIMER__TM_EITHEREDGE

  This bit-field is configured at initialization with the trigger type defined in the TriggerMode parameter. Or this functionality can be set with the Timer_SetTriggerMode() function passing one of the enumerated values listed above. There is no trigger function in the Fixed Function block.

- Timer_CTRL_TRIG_EN – The trigger enable control allows you to disable the trigger functionality through software for any period of time when that functionality is not required. While this bit is zero the Timer component will function normally. If this bit set to zero while the timer is waiting for a trigger event the Timer component will resume operation as if a trigger has happened.
- Timer_CTRL_CPOL – The capture polarity mode control is a 2-bit field used to define the expected capture input operation. This bit-field is 2 consecutive bits in the control register and all operations on this bit-field must use the #defines associated with the capture types available. These are:
  - Timer__B_TIMER__CM_NONE
  - Timer__B_TIMER__CM_RISING
  - Timer__B_TIMER__CM_FALLING
  - Timer__B_TIMER__CM_EITHER
  - Timer__B_TIMER__CM_ALT_RISING
  - Timer__B_TIMER__CM_ALT_FALLING

  This bit-field is configured at initialization with the capture type defined in the CaptureMode parameter.
- Timer_CTRL_ENABLE – The enable bit controls software enabling of the Timer component operation. The Timer component has a configurable enable mode defined at build time. If the Enable mode parameter is set to “Input Only” then the functionality of this bit is none. However in either of the other modes the Timer component does not decrement if this bit is not set to one. Normal operation requires that this bit is set and held at one during all operation of the Timer component.

Capture (8, 16, 24 or 32-bit based on Resolution)

The capture register contains the FIFO’s capture counter value. Any hardware capture event will push the current counter value onto this FIFO. The FIFO is read one entry at a time using the Timer_ReadCapture() function call. It may be useful to read the status register for the level indication of the FIFO before trying to read from the FIFO.

Additionally the Timer_1_ReadCounter() function call forces a capture of data to the FIFO and returns the oldest data from the FIFO, thus adding one capture to the FIFO and removing one capture. This information is indicated in the STATUS_FIFOFULL and STATUS_FIFONEMPTY status bits. Hardware captures are blocked while the FIFO is full preventing overwriting of data in the FIFO. The user must handle data in the FIFO in a timely manner to avoid missing capture data.

Period (8, 16, 24 or 32-bit based on Resolution)

The period register contains the period value set by the user through the Timer_WritePeriod() function call and defined by the Period parameter at initialization. The Period register has no

affect on the Timer component until a terminal count is reached at which time the period counter register is reloaded.

Counter (8, 16, 24 or 32-bit based on Resolution)

The counter register contains the period counter value throughout the operation of the Timer component. Any hardware capture event will push the current counter value onto the FIFO. This counter is decremented on each rising edge of the clock input so long as the Timer component is in an enabled state. The counter register should not be written to from the CPU.

Component Debug Window

The Timer component supports the PSoC Creator component debug window. Refer to the appropriate device data sheet for a detailed description of each register. The following registers are displayed in the Timer component debug window. Some registers are available in the UDB implementation (indicated by *) and some registers are only available in the Fixed Function Implementation (indicated by **). All other registers are available for either configuration.

Register: Timer_1_CONTROL

Name: Control Register

Description: Refer to Timer_Control register description above for bit-field definitions.

Register: Timer_1_CONTROL2 **

Name: Fixed Function Control Register #2

Description: The Fixed Function Timer block has a second configuration register. Refer to the Technical Reference Manual for bit-field definitions.

Register: Timer_1_STATUS_MASK *

Name: Status Register Interrupt Mask Configuration

Description: Allows the user to enable any status bit as an interrupt source at the interrupt output pin of the component. Refer to Timer_Status register description above for 1-to-1 correlation of bit-field definitions.

Register: Timer_1_STATUS_AUX_CTRL *

Name: Auxiliary Control Register for the Status Register

Description: Allows the user to enable the interrupt output of the internal status register through the bit-field “INT_EN”. Refer to the Technical Reference Manual for bit-field definitions.

Register: Timer_1_STATUS_MASK *

Name: Status Register Interrupt Mask Configuration

Description: Enables interrupt mask bits with a 1-to-1 correspondence to the status register (Timer_Status) bit-field descriptions above.

Register: Timer_1_PERIOD

Name: Timer Period Register

Description: Defines the period value reloaded into the period counter at the beginning of each cycle of the Timer.

Register: Timer_1_COUNTER

Name: Timer Counter Register

Description: Indicates the current counter value (in clock cycles from Period down to zero) of the current timer period cycle.

Register: Timer_1_GLOBAL_ENABLE **

Name: Fixed Function Timer Global Enable Register

Description: Enables the Fixed Function Timer for operation. Refer to the Technical Reference Manual for bit-field definitions.

Conditional Compilation Information

The counter API requires two conditional compile definitions to handle the multiple configurations it must support. It is required that the API conditionally compile on the Resolution chosen and the Implementation chosen from either the fixed function block or the UDB blocks. The two conditions defined are based on the parameters FixedFunction and Resolution. The API should never use these parameters directly but should use the two defines listed below.

Timer_DataWidth

The DataWidth 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.

Timer_UsingFixedFunction

The Using Fixed Function define is used mostly in the header file to make the correct register assignments as the registers provided in the fixed function block are different than those used when the Timer component is implemented in UDB's. In some cases this define is also used with the DataWidth define because the Fixed Function block is limited to 16 bits maximum data width.

Constants

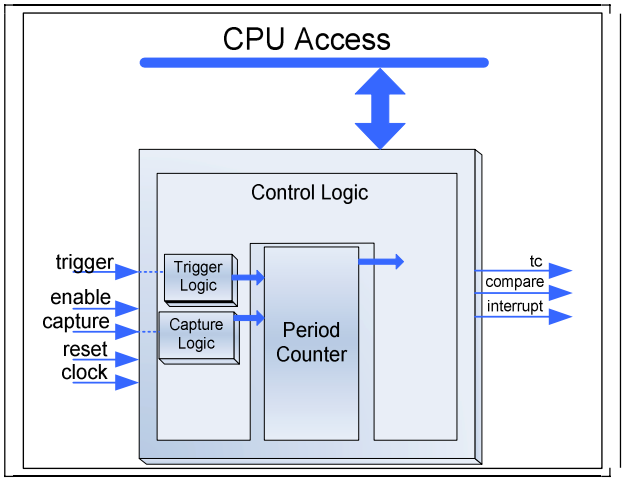
There are several constants defined for the status and control registers as well as some of the enumerated types. Most of these are described above for the Control and Status Register. However there are more constants needed in the header file to make all of this happen. Each of the register definitions requires either a pointer into the register data or a register address. Because of multiple Endianness of the compilers, it is required that the CY_GET_REGX and

CY_SET_REGX macros are used for register accesses greater than 8-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 in that we 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 which defines the bit's offset within the register. These are used in the header file to define the final bit mask as an _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. The UDB implementation is implemented according to the following block diagram.

Figure 6 UDB Implementation



The block diagram above shows the Timer component period counter implemented as a datapath and some control logic. There is a status register and a control register that feed into and come out of the logic cloud as well. All of the logic for the Timer component is the same whether the datapath is 8, 16, 24, or 32 bits wide.

References

Not applicable

DC and AC Electrical Characteristics (FF Implementation)

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

Timer DC Specifications

Parameter	Description	Conditions	Min	Typ	Max	Units
	Block current consumption	16-bit timer, at listed input clock frequency	--	--	--	µA
	3 MHz		--	15	--	µA
	12 MHz		--	60	--	µA
	48 MHz		--	260	--	µA
	67 MHz		--	350	--	µA

Timer 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
	Reset pulse width		15	--	--	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.	Min	Typ	Max	Units
f_{clock}	Component Clock Frequency [1]	8 bit UDB Timer			40	MHz
		16 bit UDB Timer			38	MHz
		24 bit UDB Timer			33	MHz
		32 bit UDB Timer			27	MHz
t_{clockH}	Input Clock High Time [2]	N/A		0.5		$1/f_{clock}$
t_{clockL}	Input Clock Low Time [2]	N/A		0.5		$1/f_{clock}$
Inputs						
t_{PD_ps}	Input path delay, pin to sync [3]	1			STA [4]	ns
t_{PD_ps}	Input path delay, pin to sync [5]	2			8.5	ns
t_{PD_si}	Sync output to Input Path Delay (route)	1,2,3,4			STA [4]	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} [6]			ns
t_{IL}	Input Low Time	1,2,3,4	t_{CY_clock} [6]			ns

[1] If Time Division Multiplex Implementation is selected, then Component Clock Frequency must be 4 times greater than the data rate.

[2] $t_{CY_clock} = 1/f_{clock}$ - Cycle time of one clock period.

[3] 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.

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

[5] 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

[6] $t_{CY_clock} = 4^* [1/f_{clock}]$ if Time Division Multiplex Implementation is selected.

Timing Characteristics "Maximum with All Routing"

Parameter	Description	Config.	Min	Typ	Max [1]	Units
f_{clock}	Component Clock Frequency [2]	8 bit UDB Timer			20	MHz
		16 bit UDB Timer			15	MHz
		24 bit UDB Timer			20	MHz
		32 bit UDB Timer			15	MHz
t_{clockH}	Input Clock High Time [3]	N/A		0.5		$1/f_{clock}$
t_{clockL}	Input Clock Low Time [3]	N/A		0.5		$1/f_{clock}$
Inputs						
t_{PD_ps}	Input path delay, pin to sync [4]	1			STA [5]	ns
t_{PD_ps}	Input path delay, pin to sync [6]	2			8.5	ns
t_{PD_si}	Sync output to Input Path Delay (route)	1,2,3,4			STA [5]	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} [7]			ns
t_{IL}	Input Low Time	1,2,3,4	t_{CY_clock} [6]			ns

[1] Maximum for "All Routing" is calculated by <nominal>/2 rounded to the nearest integer tested against empirical values obtained using the set of characterization unit tests. This value provides a basis for the user to not have to worry about meeting timing if they are running at or below this component frequency.

[2] If Time Division Multiplex Implementation is selected, then Component Clock Frequency must be 4 times greater than the data rate.

[3] $t_{CY_clock} = 1/f_{clock}$ - Cycle time of one clock period.

[4] 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.

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

[6] 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

[7] $t_{CY_clock} = 4^* [1/f_{clock}]$ if Time Division Multiplex Implementation is selected.

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

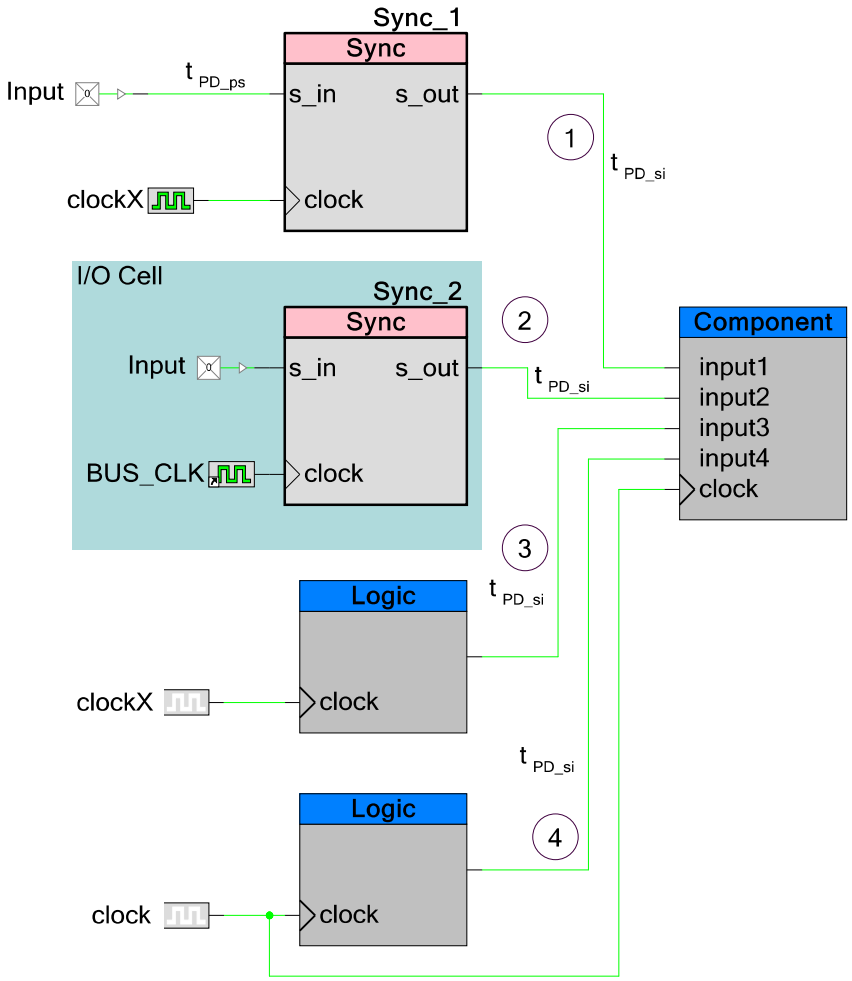
Clock	Actual Freq	Max Freq	Violation
BUS_CLK	24.000 MHz	118.683 MHz	
clock	24.000 MHz	56.967 MHz	

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 7**Error! Reference source not found.**.

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 7. Input Configurations for Component Timing Specifications



Configuration	Component Clock	Synchronizer Clock (Frequency)	Figures
1	master_clock	master_clock	Figure 12
1	clock	master_clock	Figure 10
1	clock	clockX = clock [1]	Figure 8
1	clock	clockX > clock	Figure 9
1	clock	clockX < clock	Figure 11
2	master_clock	master_clock	Figure 12
2	clock	master_clock	Figure 10
3	master_clock	master_clock	Figure 17

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

Configuration	Component Clock	Synchronizer Clock (Frequency)	Figures
3	clock	master_clock	Figure 15
3	clock	clockX = clock [1]	Figure 13
3	clock	clockX > clock	Figure 14
3	clock	clockX < clock	Figure 16
4	master_clock	master_clock	Figure 17
4	clock	clock	Figure 13

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, equal to, or slower than the component clock. It may also be equal to master_clock, which produces the characterization parameters shown in Figure 8, Figure 9, Figure 11, and Figure 12.
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 9 and Figure 12.

Figure 8: Input Configuration 1 and 2; Sync Clock Freq.= Component Clock Freq. (Edge alignment of clock and clockX is not guaranteed)

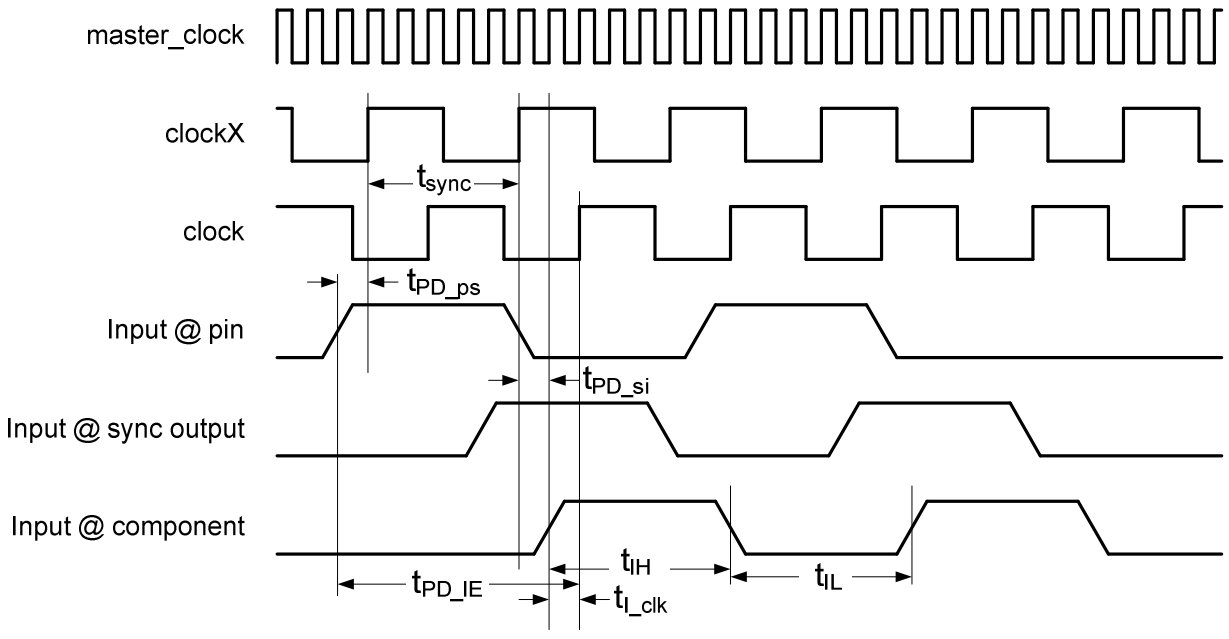


Figure 9: Input Configuration 1 and 2; Sync. Clock Freq. > Component Clock Freq.

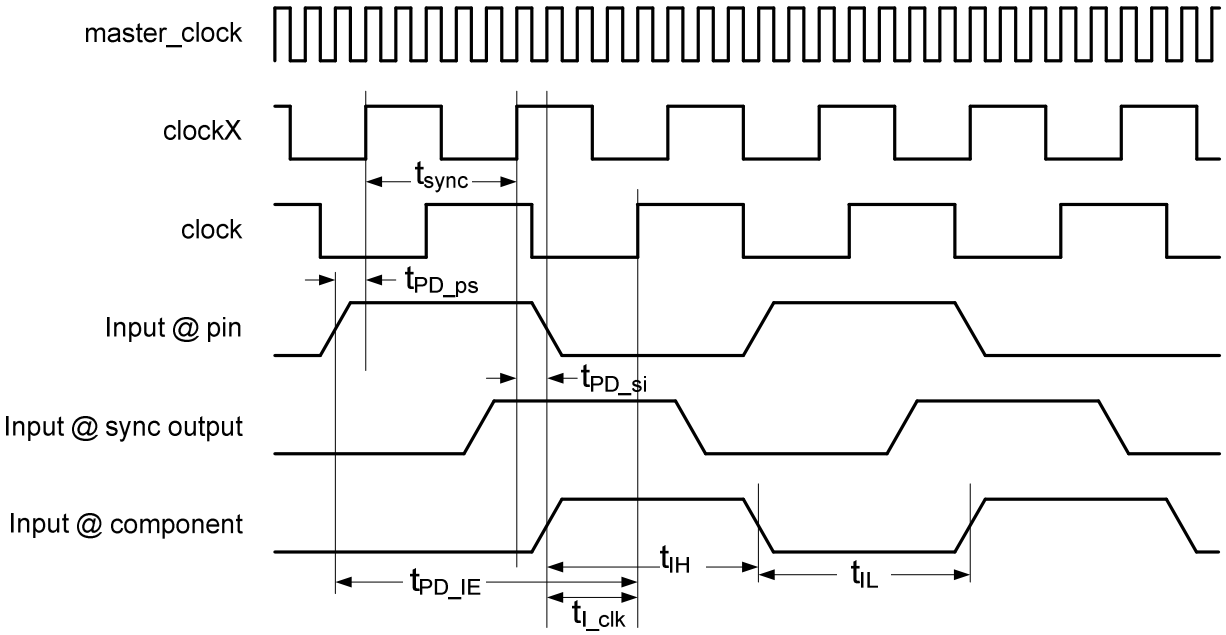


Figure 10: Input Configuration 1 and 2; [Sync. Clock Freq. == master_clock] > Component Clock Freq.

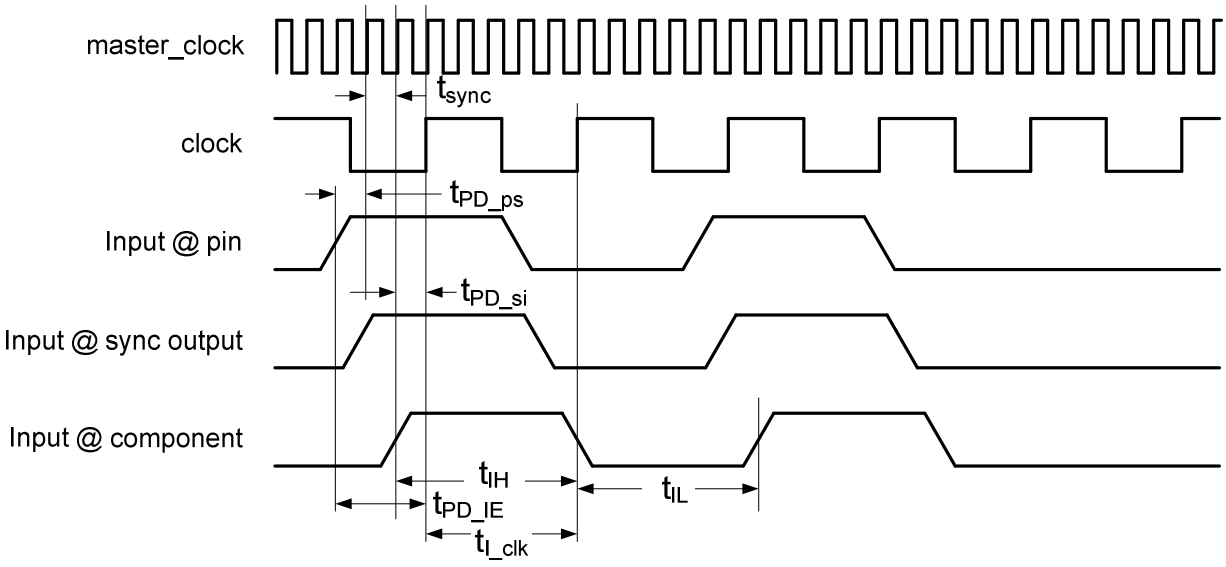


Figure 11: Input Configuration 1; Sync. Clock Freq. < Component Clock Freq.

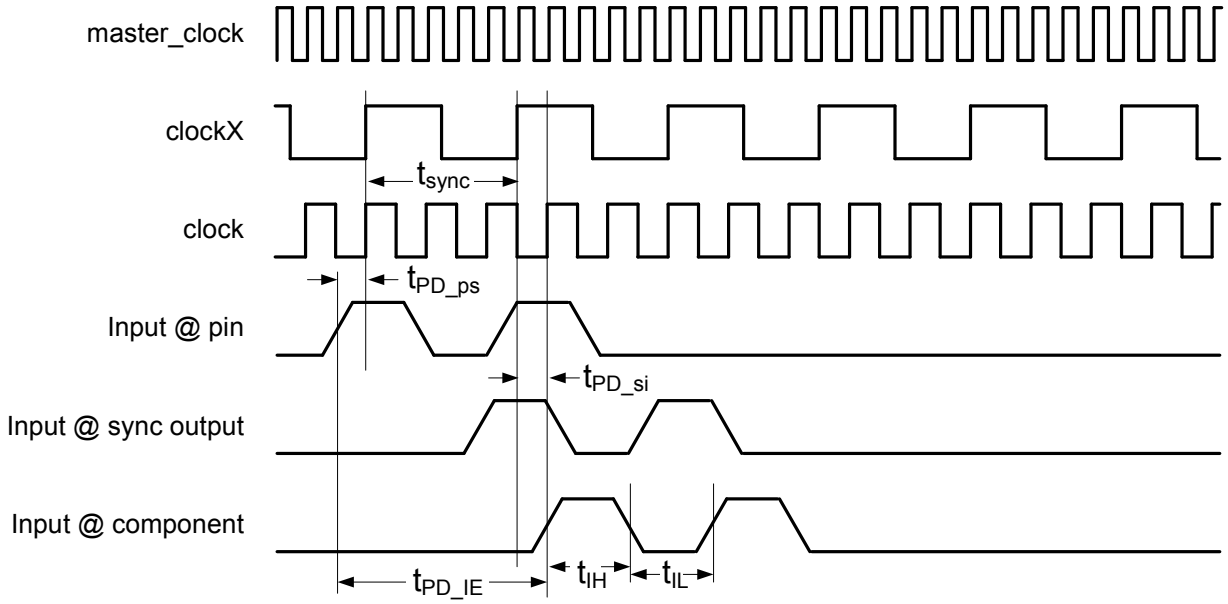
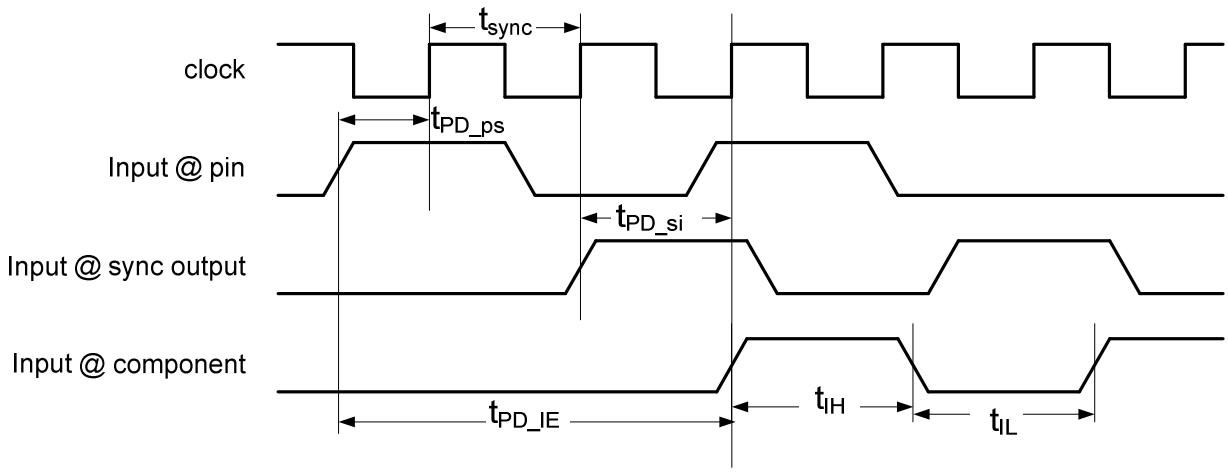


Figure 12: Input Configuration 1 and 2; Sync. Clock = Component Clock = master_clock



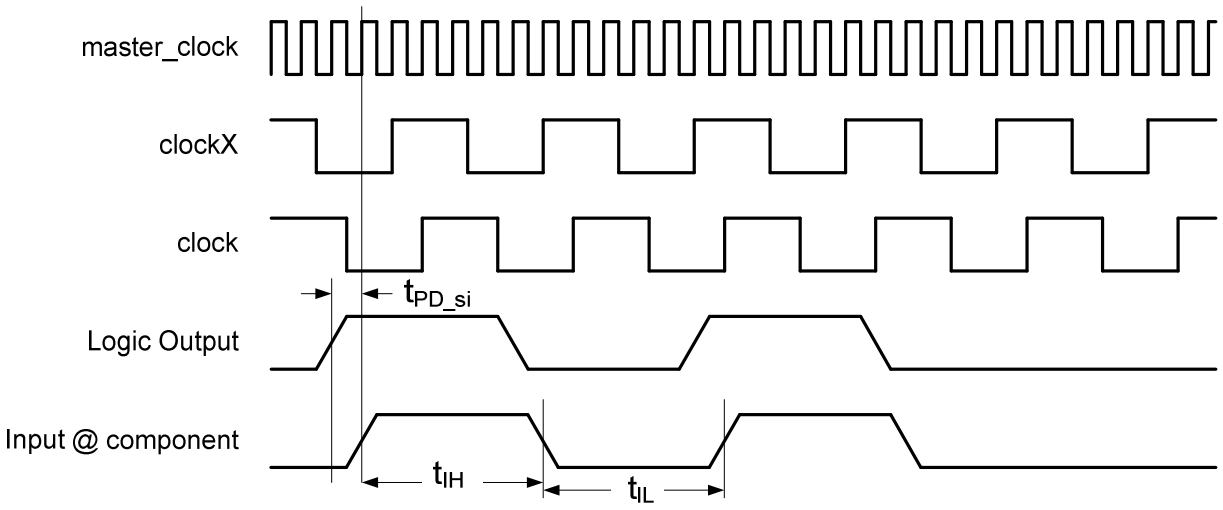
3. The input is driven by logic internal to the PSoC, which is synchronous based on a clock other than the clock the component uses (all internal clocks are derived from master_clock).

   When characterizing inputs configured in this way, the synchronizer clock is faster than, less than, or equal to the component clock, which produces the characterization parameters shown in Figure 13, Figure 14, and Figure 16.

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

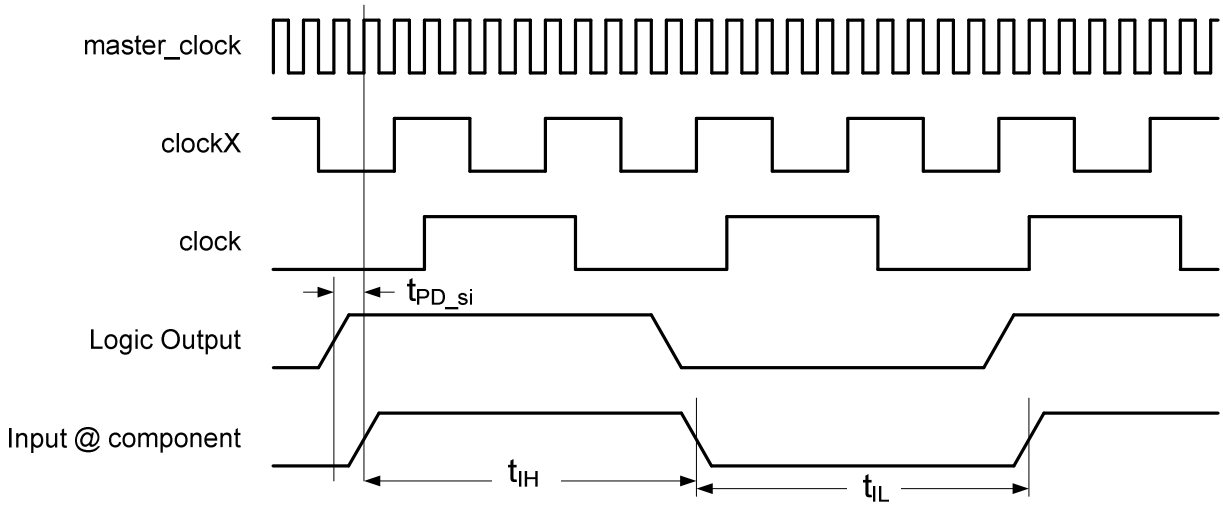
When characterizing inputs configured in this way, the synchronizer clock will be equal to the component clock, which will produce the characterization parameters as shown in Figure 17.

Figure 13: Input Configuration 3 only; Sync. Clock Freq. = Component Clock Freq. (Edge alignment of clock and clockX is not guaranteed)



This figure represents the understanding that Static Timing Analysis holds on 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 1 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 14: Input Configuration 3; Sync. Clock Freq. > Component Clock Freq.



In much the same way as shown in Figure 13, 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 15: Input Configuration 3; Synchronizer Clock Frequency = master_clock > Component Clock Frequency

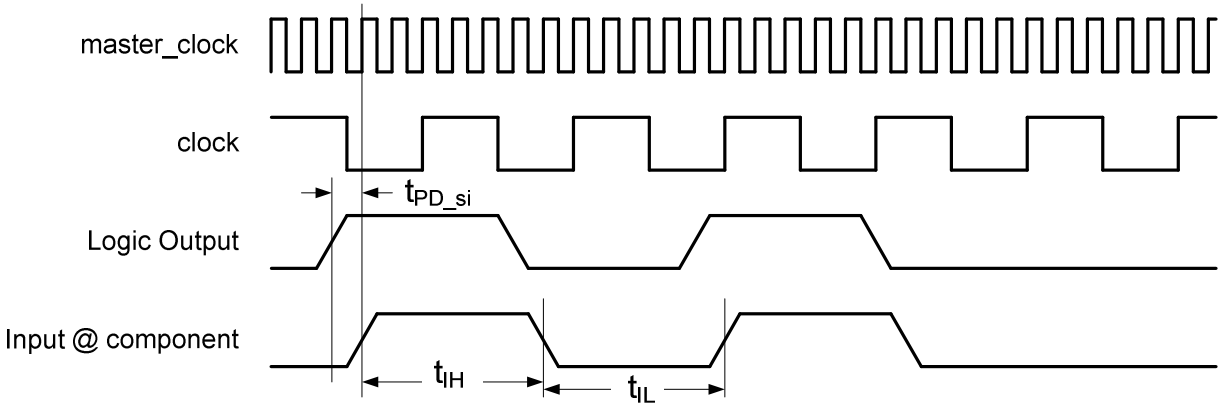
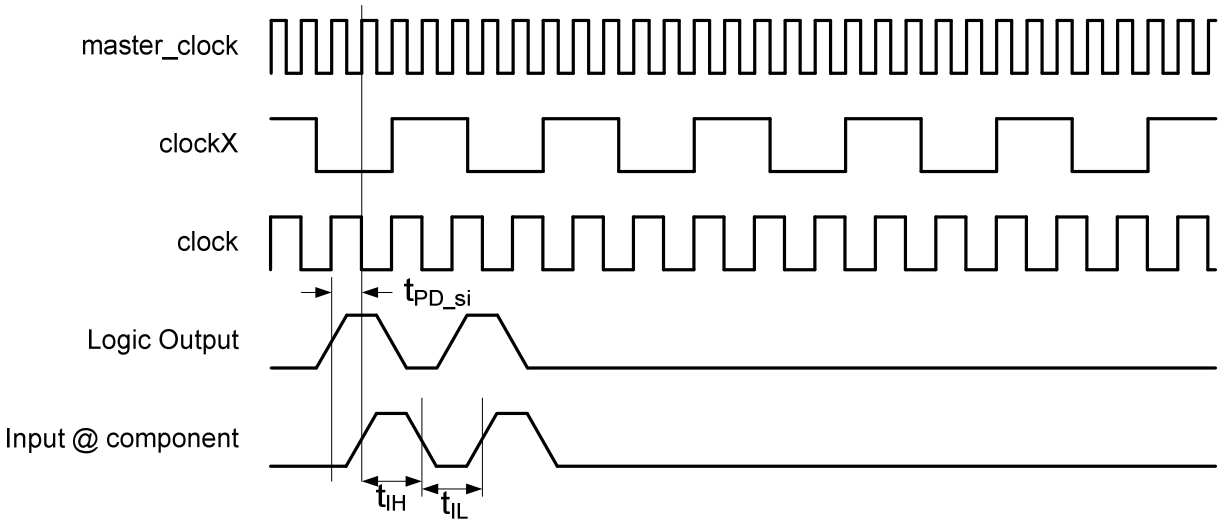
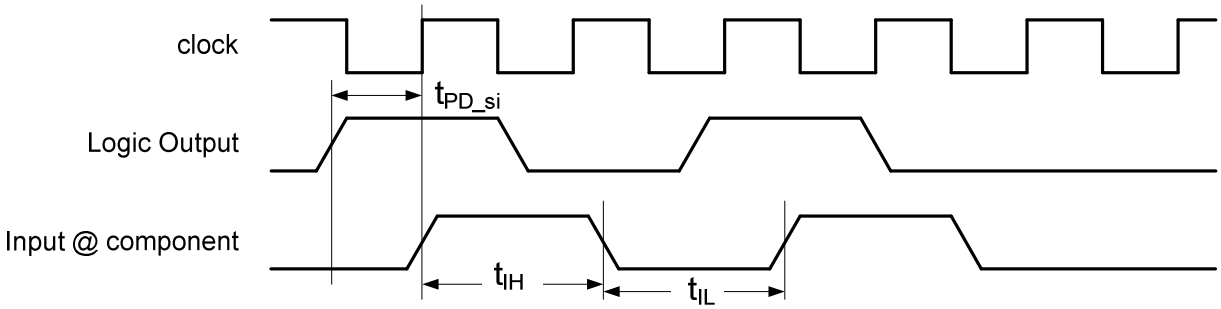


Figure 16: Input Configuration 3; Synchronizer Clock Frequency < Component Clock Frequency



In much the same way as shown in Figure 13, 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 17: 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}. Of critical importance is the fact 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 fan-out of this input may be more than 1 so you will 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} will be defined in the Register-to-register times. You will need to know the name of the net to use the *_timing.html* file. The fan-out of this path may be more than 1 so you will 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
2.0	Synchronized inputs	All inputs are synchronized in the fixed function implementation, at the input of the block.
	Timer_GetInterruptSource() function was converted to a Macro	The Timer_GetInterruptSource() function is exactly the same implementation as the Timer_ReadStatusRegister() function. To save code space this was converted to a macro substitution of the Timer_ReadStatusRegister() function.

Version	Description of Changes	Reason for Changes / Impact
	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.
	Incorrect masking rectified while setting capture mode using SetCaptureMode() API.	Masking used for setting capture mode has erroneous value.
	Added characterization data to datasheet	
	Minor datasheet edits and updates	

© Cypress Semiconductor Corporation, 2008-2010. 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.