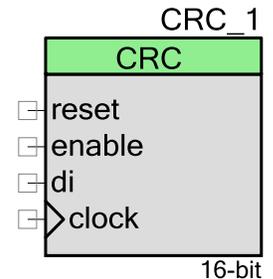


# Cyclic Redundancy Check (CRC)

2.0

## Features

- 1 to 64 bits
- Time Division Multiplexing mode
- Requires clock and data for serial bit stream input
- Serial data in, parallel result
- Standard [CRC-1 (parity bit), CRC-4 (ITU-T G.704), CRC-5-USB, etc.] or custom polynomial
- Standard or custom seed value
- Enable input provides synchronized operation with other components



## General Description

The default use of the Cyclic Redundancy Check (CRC) component is to compute CRC from a serial bit stream of any length. The input data is sampled on the rising edge of the data clock. The CRC value is reset to 0 before starting or can optionally be seeded with an initial value. On completion of the bitstream, the computed CRC value may be read out.

## When to Use a CRC

The default CRC component can be used as a checksum to detect alteration of data during transmission or storage. CRCs are popular because they are simple to implement in binary hardware, are easy to analyze mathematically, and are particularly good at detecting common errors caused by noise in transmission channels.

## Input/Output Connections

This section describes the various input and output connections for the CRC. An asterisk (\*) in the list of I/Os indicates that the I/O may be hidden on the symbol under the conditions listed in the description of that I/O.

### clock – Input

The CRC requires a data input that provides the serial bitstream used to calculate the CRC. A data clock input is also required in order to correctly sample the serial data input. The input data is sampled on the rising edge of the data clock.

### reset – Input

The reset input defines the signal to asynchronous reset CRC.

### enable – Input

The CRC component runs after it is started and as long as the Enable input is held high. This input provides synchronized operation with other components.

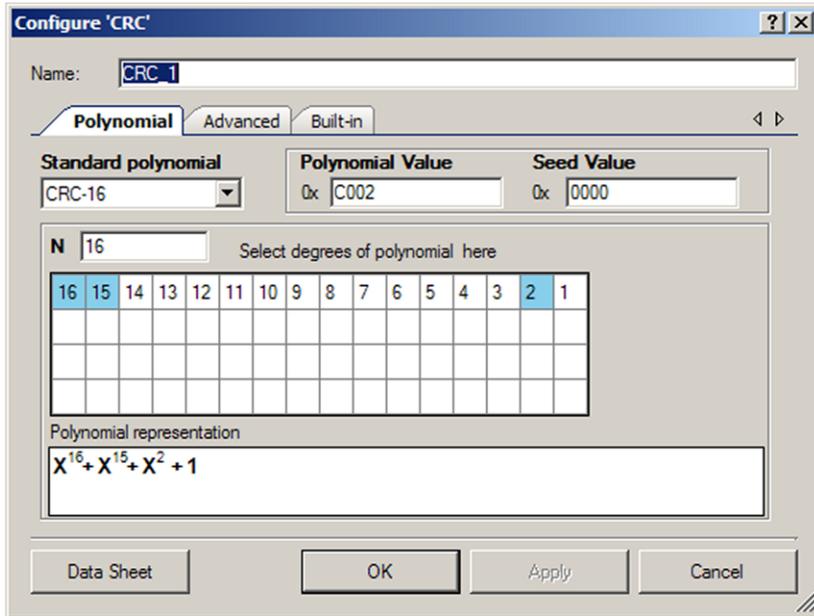
### di – Input

Data input that provides the serial bitstream used to calculate the CRC.

## Component Parameters

Drag a CRC component onto your design and double click it to open the **Configure** dialog. This dialog has several tabs to guide you through the process of setting up the CRC component.

### Polynomial Tab



### Standard Polynomial

This parameter allows you to choose one of the standard CRC polynomials provided in the **Standard polynomial** combo box or generate a custom polynomial. The additional information about each standard polynomial is given in the tool tip. The default is **CRC-16**.

Polynomial Name	Polynomial	Use
Custom	User defined	General
CRC-1	$x + 1$	Parity
CRC-4-ITU	$x^4 + x + 1$	ITU G.704
CRC-5-ITU	$x^5 + x^4 + x^2 + 1$	ITU G.704
CRC-5-USB	$x^5 + x^2 + 1$	USB
CRC-6-ITU	$x^6 + x + 1$	ITU G.704
CRC-7	$x^7 + x^3 + 1$	Telecom systems, MMC
CRC-8-ATM	$x^8 + x^2 + x + 1$	ATM HEC
CRC-8-CCITT	$x^8 + x^7 + x^3 + x^2 + 1$	1-Wire bus
CRC-8-Maxim	$x^8 + x^5 + x^4 + 1$	1-Wire bus



Polynomial Name	Polynomial	Use
CRC-8	$x^8 + x^7 + x^6 + x^4 + x^2 + 1$	General
CRC-8-SAE	$x^8 + x^4 + x^3 + x^2 + 1$	SAE J1850
CRC-10	$x^{10} + x^9 + x^5 + x^4 + x + 1$	General
CRC-12	$x^{12} + x^{11} + x^3 + x^2 + x + 1$	Telecom systems
CRC-15-CAN	$x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$	CAN
CRC-16-CCITT	$x^{16} + x^{12} + x^5 + 1$	XMODEM, X.25, V.41, Bluetooth, PPP, IrDA, CRC-CCITT
CRC-16	$x^{16} + x^{15} + x^2 + 1$	USB
CRC-24-Radix64	$x^{24} + x^{23} + x^{18} + x^{17} + x^{14} + x^{11} + x^{10} + x^7 + x^6 + x^5 + x^4 + x^3 + x + 1$	General
CRC-32-IEEE802.3	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$	Ethernet, MPEG2
CRC-32C	$x^{32} + x^{28} + x^{27} + x^{26} + x^{25} + x^{23} + x^{22} + x^{20} + x^{19} + x^{18} + x^{14} + x^{13} + x^{11} + x^{10} + x^9 + x^8 + x^6 + 1$	General
CRC-32K	$x^{32} + x^{30} + x^{29} + x^{28} + x^{26} + x^{20} + x^{19} + x^{17} + x^{16} + x^{15} + x^{11} + x^{10} + x^7 + x^6 + x^4 + x^2 + x + 1$	General
CRC-64-ISO	$x^{64} + x^4 + x^3 + x + 1$	ISO 3309
CRC-64-ECMA	$x^{64} + x^{62} + x^{57} + x^{55} + x^{54} + x^{53} + x^{52} + x^{47} + x^{46} + x^{45} + x^{40} + x^{39} + x^{38} + x^{37} + x^{35} + x^{33} + x^{32} + x^{31} + x^{29} + x^{27} + x^{24} + x^{23} + x^{22} + x^{21} + x^{19} + x^{17} + x^{13} + x^{12} + x^{10} + x^9 + x^7 + x^4 + x + 1$	ECMA-182

### Polynomial Value

This parameter is represented in hexadecimal format. It is calculated automatically when one of the standard polynomials is selected. You may also enter it manually (see [Custom Polynomials](#)).

### Seed Value

This parameter is represented in hexadecimal format. The maximum possible value is  $2^N - 1$ .

### N

This parameter defines the degree of polynomial. Possible values are 1 to 64 bits. The table with numbers indicates which degrees are included in the polynomial. Cells with selected numbers are blue; others are white. The number of active cells is equal to N. Numbers are arranged in reverse order. You may click on the cell to select or deselect a number.

### Polynomial representation

This parameter displays the resulting polynomial in mathematical notation.



## Custom Polynomials

You may enter a custom polynomial in three different ways:

### Small Changes to Standard Polynomial

- Choose one of the standard polynomials.
- Select the necessary degrees in the table by clicking on the appropriate cells; the text in **Standard polynomial** changes to **Custom**.
- The polynomial value is recalculated automatically based on the polynomial that is represented.

### Use Polynomial Degrees

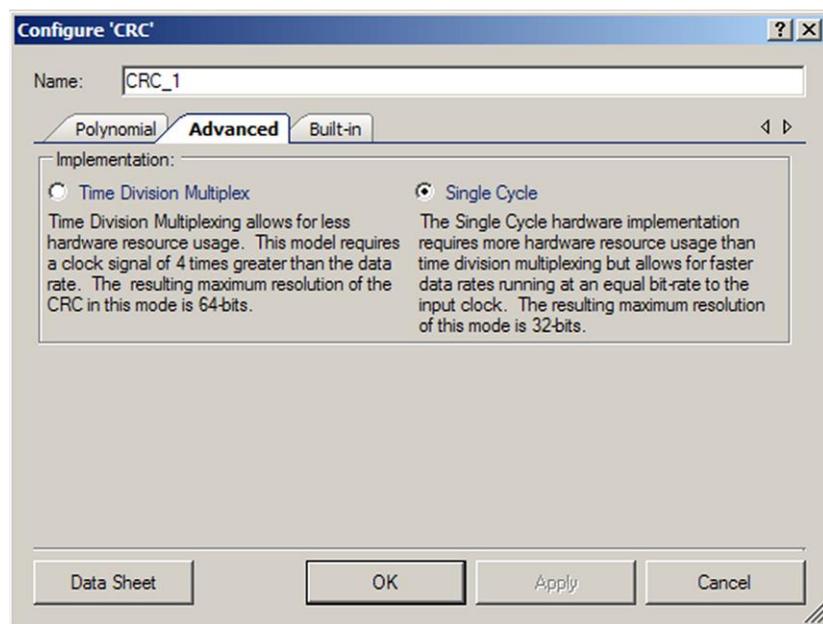
- Enter a custom polynomial in the **N** textbox; the text in **Standard polynomial** changes to **Custom**.
- Select the necessary degrees in the table by clicking on the appropriate cells.
- Check the view of the polynomial in **Polynomial representation**.
- The polynomial value is recalculated automatically based on the polynomial that is represented.

### Use Hexadecimal Format

- Enter a polynomial value in hexadecimal form in the **Polynomial Value** text box.
- Press **[Enter]** or switch to another control; the text in **Standard polynomial** changes to **Custom**.
- The N value and degrees of polynomial will be recalculated based on the entered polynomial value.



## Advanced Tab



### Implementation

This parameter defines the implementation of the CRC component: **Time Division Multiplex** or **Single Cycle**. The default is **Single Cycle**.

### Local Parameters (For API use)

These parameters are used in the API and are not exposed in the GUI:

- **PolyValueLower (uint32)** – Contains the lower half of the polynomial value in hexadecimal format. The default is 0xB8h (LFSR= [8,6,5,4]) because the default resolution is 8.
- **PolyValueUpper (uint32)** – Contains the upper half of the polynomial value in hexadecimal format. The default is 0x00h because the default resolution is 8.
- **SeedValueLower (uint32)** – Contains the lower half of the seed value in hexadecimal format. The default is 0xFFh because the default resolution is 8.
- **SeedValueUpper (uint32)** – Contains the upper half of the seed value in hexadecimal format. The default is 0 because the default resolution is 8.

## Clock Selection

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

**Note** Generation of the proper CRC sequence for a resolution of greater than eight requires a clock signal of four times greater than the data rate, if you select **Time Division Multiplex** for the **Implementation** parameter.

## Placement

The CRC is placed throughout the UDB array and all placement information is provided to the API through the *cyfitter.h* file.

## Resources

### Single Cycle Implementation

Resources	Resource Type			API Memory (Bytes)		Pins (per External I/O)
	Datapath Cells	PLDs	Control/Count7 Cells	Flash	RAM	
1..8-Bits Resolution	1	1	1	166	2	4
9..16-Bits Resolution	2	1	1	210	2	4
17..24-Bits Resolution	3	1	1	287	2	4
25..32-Bits Resolution	4	1	1	288	2	4

### Time Division Multiplex Implementation

Resources	Resource Type			API Memory (Bytes)		Pins (per External I/O)
	Datapath Cells	PLDs	Control/Count7 Cells	Flash	RAM	
9..16-Bits Resolution	1	3	1	242	2	4
17..24-Bits Resolution	2	3	1	538	2	4
25..32-Bits Resolution	2	3	1	615	2	4
33..40-Bits Resolution	3	3	1	763	2	4
41..48-Bits Resolution	3	3	1	894	2	4
49..56-Bits Resolution	4	3	1	999	2	4
57..64-Bits Resolution	4	3	1	1101	2	4



## 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 “CRC\_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 “CRC.”

Function	Description
CRC_Start()	Initializes seed and polynomial registers with initial values. Computation of CRC starts on rising edge of input clock.
CRC_Stop()	Stops CRC computation.
CRC_Wakeup()	Restores the CRC configuration and starts CRC computation on rising edge of input clock.
CRC_Sleep()	Stops CRC computation and saves the CRC configuration.
CRC_Init()	Initializes the seed and polynomial registers with initial values.
CRC_Enable()	Starts CRC computation on rising edge of input clock.
CRC_SaveConfig()	Saves the seed and polynomial registers.
CRC_RestoreConfig()	Restores the seed and polynomial registers.
CRC_WriteSeed()	Writes the seed value.
CRC_WriteSeedUpper()	Writes the upper half of the seed value. Only generated for 33- to 64-bit CRC.
CRC_WriteSeedLower()	Writes the lower half of the seed value. Only generated for 33- to 64-bit CRC.
CRC_ReadCRC()	Reads the CRC value.
CRC_ReadCRCUpper()	Reads the upper half of the CRC value. Only generated for 33- to 64-bit CRC.
CRC_ReadCRCLower()	Reads the lower half of the CRC value. Only generated for 33- to 64-bit CRC.
CRC_WritePolynomial()	Writes the CRC polynomial value.
CRC_WritePolynomialUpper()	Writes the upper half of the CRC polynomial value. Only generated for 33- to 64-bit CRC.
CRC_WritePolynomialLower()	Writes the lower half of the CRC polynomial value. Only generated for 33- to 64-bit CRC.
CRC_ReadPolynomial()	Reads the CRC polynomial value.
CRC_ReadPolynomialUpper()	Reads the upper half of the CRC polynomial value. Only generated for 33- to 64-bit CRC.
CRC_ReadPolynomialLower()	Reads the lower half of the CRC polynomial value. Only generated for 33- to 64-bit CRC.

## Global Variables

Variable	Description
CRC_initVar	<p>Indicates whether the CRC has been initialized. The variable is initialized to 0 and set to 1 the first time CRC_Start() is called. This allows the component to restart without reinitialization after the first call to the CRC_Start() routine.</p> <p>If reinitialization of the component is required, then the CRC_Init() function can be called before the CRC_Start() or CRC_Enable() function.</p>

### void CRC\_Start(void)

<b>Description:</b>	Initializes seed and polynomial registers with initial values. Computation of CRC starts on rising edge of input clock.
<b>Parameters:</b>	None
<b>Return Value:</b>	None
<b>Side Effects:</b>	None

### void CRC\_Stop(void)

<b>Description:</b>	Stops CRC computation.
<b>Parameters:</b>	None
<b>Return Value:</b>	None
<b>Side Effects:</b>	None

### void CRC\_Sleep(void)

<b>Description:</b>	Stops CRC computation and saves the CRC configuration.
<b>Parameters:</b>	None
<b>Return Value:</b>	None
<b>Side Effects:</b>	None



## void CRC\_Wakeup(void)

<b>Description:</b>	Restores the CRC configuration and starts CRC computation on the rising edge of the input clock.
<b>Parameters:</b>	None
<b>Return Value:</b>	None
<b>Side Effects:</b>	None

## void CRC\_Init(void)

<b>Description:</b>	Initializes the seed and polynomial registers with initial values.
<b>Parameters:</b>	None
<b>Return Value:</b>	None
<b>Side Effects:</b>	None

## void CRC\_Enable(void)

<b>Description:</b>	Starts CRC computation on the rising edge of the input clock.
<b>Parameters:</b>	None
<b>Return Value:</b>	None
<b>Side Effects:</b>	None

## void CRC\_SaveConfig(void)

<b>Description:</b>	Saves the initial seed and polynomial registers.
<b>Parameters:</b>	None
<b>Return Value:</b>	None
<b>Side Effects:</b>	None

## void CRC\_RestoreConfig(void)

<b>Description:</b>	Restores the initial seed and polynomial registers.
<b>Parameters:</b>	None
<b>Return Value:</b>	None
<b>Side Effects:</b>	None

### void CRC\_WriteSeed(uint8/16/32 seed)

- Description:** Writes the seed value.
- Parameters:** uint8/16/32 seed: Seed value
- Return Value:** None
- Side Effects:** The seed value is cut according to mask =  $2^{\text{Resolution}} - 1$ .  
For example, if CRC Resolution is 14 bits the mask value is: mask =  $2^{14} - 1 = 0x3FFFu$ .  
The seed value =  $0xFFFFu$  is cut: seed and mask =  $0xFFFFu$  and  $0x3FFFu = 0x3FFFu$ .

### void CRC\_WriteSeedUpper(uint32 seed)

- Description:** Writes the upper half of the seed value. Only generated for 33- to 64-bit CRC.
- Parameters:** uint32 seed: Upper half of the seed value
- Return Value:** None
- Side Effects:** The upper half of the seed value is cut according to mask =  $2^{\text{Resolution} - 32} - 1$ .  
For example, if CRC Resolution is 35 bits the mask value is:  
 $2^{(35 - 32)} - 1 = 2^3 - 1 = 0x0000\ 0007u$ .  
The upper half of the seed value =  $0x0000\ 00FFu$  is cut:  
upper half of seed and mask =  $0x0000\ 00FFu$  and  $0x0000\ 0007u = 0x0000\ 0007u$ .

### void CRC\_WriteSeedLower(uint32 seed)

- Description:** Writes the lower half of the seed value. Only generated for 33- to 64-bit CRC.
- Parameters:** uint32 seed: Lower half of the seed value
- Return Value:** None
- Side Effects:** None

### uint8/16/32 CRC\_ReadCRC(void)

- Description:** Reads the CRC value.
- Parameters:** None
- Return Value:** uint8/16/32: Returns the CRC value
- Side Effects:** None



**uint32 CRC\_ReadCRCUpper(void)**

<b>Description:</b>	Reads the upper half of the CRC value. Only generated for 33- to 64-bit CRC.
<b>Parameters:</b>	None
<b>Return Value:</b>	uint32: Returns the upper half of the CRC value.
<b>Side Effects:</b>	None

**uint32 CRC\_ReadCRCLower(void)**

<b>Description:</b>	Reads the lower half of the CRC value. Only generated for 33- to 64-bit CRC.
<b>Parameters:</b>	None
<b>Return Value:</b>	uint32: Returns the lower half of the CRC value.
<b>Side Effects:</b>	None

**void CRC\_WritePolynomial(uint8/16/32 polynomial)**

<b>Description:</b>	Writes the CRC polynomial value.
<b>Parameters:</b>	uint8/16/32 polynomial: CRC polynomial
<b>Return Value:</b>	None
<b>Side Effects:</b>	The polynomial value is cut according to mask = $2^{\text{Resolution}} - 1$ . For example, if CRC Resolution is 14 bits the mask value is: mask = $2^{14} - 1 = 0x3FFFu$ . The polynomial value = $0xFFFFu$ is cut: polynomial and mask = $0xFFFFu$ and $0x3FFFu = 0x3FFFu$ .

**void CRC\_WritePolynomialUpper(uint32 polynomial)**

<b>Description:</b>	Writes the upper half of the CRC polynomial value. Only generated for 33- to 64-bit CRC.
<b>Parameters:</b>	uint32 polynomial: Upper half of the CRC polynomial value
<b>Return Value:</b>	None
<b>Side Effects:</b>	The upper half of the polynomial value is cut according to mask = $2^{(\text{Resolution} - 32)} - 1$ . For example, if CRC Resolution is 35 bits the mask value is: $2^{(35 - 32)} - 1 = 2^3 - 1 = 0x0000 0007u$ . The upper half of the polynomial value = $0x0000 00FFu$ is cut: upper half of polynomial and mask = $0x0000 00FFu$ and $0x0000 0007u = 0x0000 0007u$ .

## void CRC\_WritePolynomialLower(uint32 polynomial)

<b>Description:</b>	Writes the lower half of the CRC polynomial value. Only generated for 33- to 64-bit CRC.
<b>Parameters:</b>	uint32 polynomial: Lower half of the CRC polynomial value
<b>Return Value:</b>	None
<b>Side Effects:</b>	None

## uint8/16/32 CRC\_ReadPolynomial(void)

<b>Description:</b>	Reads the CRC polynomial value.
<b>Parameters:</b>	None
<b>Return Value:</b>	uint8/16/32: Returns the CRC polynomial value
<b>Side Effects:</b>	None

## uint32 CRC\_ReadPolynomialUpper(void)

<b>Description:</b>	Reads the upper half of the CRC polynomial value. Only generated for 33- to 64-bit CRC.
<b>Parameters:</b>	None
<b>Return Value:</b>	uint32: Returns the upper half of the CRC polynomial value
<b>Side Effects:</b>	None

## uint32 CRC\_ReadPolynomialLower(void)

<b>Description:</b>	Reads the lower half of the CRC polynomial value. Only generated for 33- to 64-bit CRC.
<b>Parameters:</b>	None
<b>Return Value:</b>	uint32: Returns the lower half of the CRC polynomial value.
<b>Side Effects:</b>	None

## 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

The CRC is implemented as a linear feedback shift register (LFSR). The Shift register computes the LFSR function, the Polynomial register holds the polynomial that defines the LFSR polynomial, and the Seed register enables initialization of the starting data.

This component requires that the Seed and Polynomial registers are initialized prior to start.

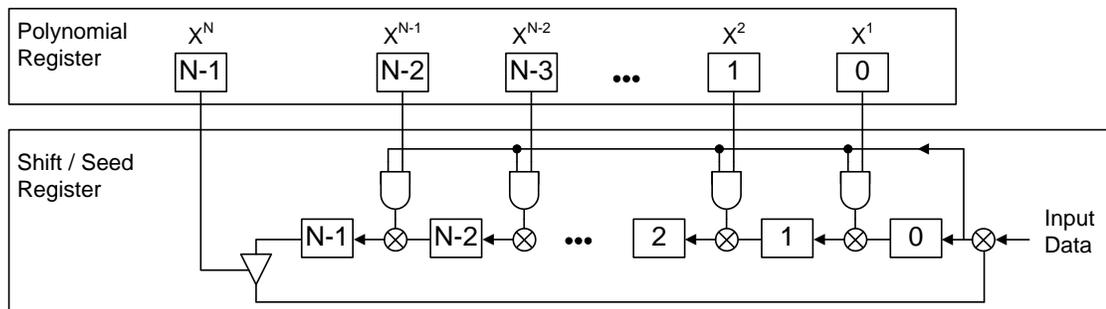
Computation of an N-bit LFSR result is specified by a polynomial with N + 1 terms, the last of which is the  $X^0$  term where  $X^0 = 1$ . For example, the widely used CRC-CCITT 16-bit polynomial is  $X^{16} + X^{12} + X^5 + 1$ . The CRC algorithm assumes the presence of the  $X^0$  term, so that the polynomial for an N-bit result can be expressed by an N bit rather than (N + 1)-bit specification.

To specify the polynomial specification, write an (N + 1)-bit binary number corresponding to the full polynomial, with 1's for each term present. The CRC-CCITT polynomial would be 10001000000100001b. Then, drop the right-most bit (the  $X^0$  term) to obtain the CRC polynomial value. To implement the CRC-CCITT example, the Polynomial register is loaded with a value of 8810h.

A rising edge of the input clock shifts each bit of the input data stream, MSB first, through the Shift register, computing the specified CRC algorithm. Eight clocks are required to compute the CRC for each byte of input data.

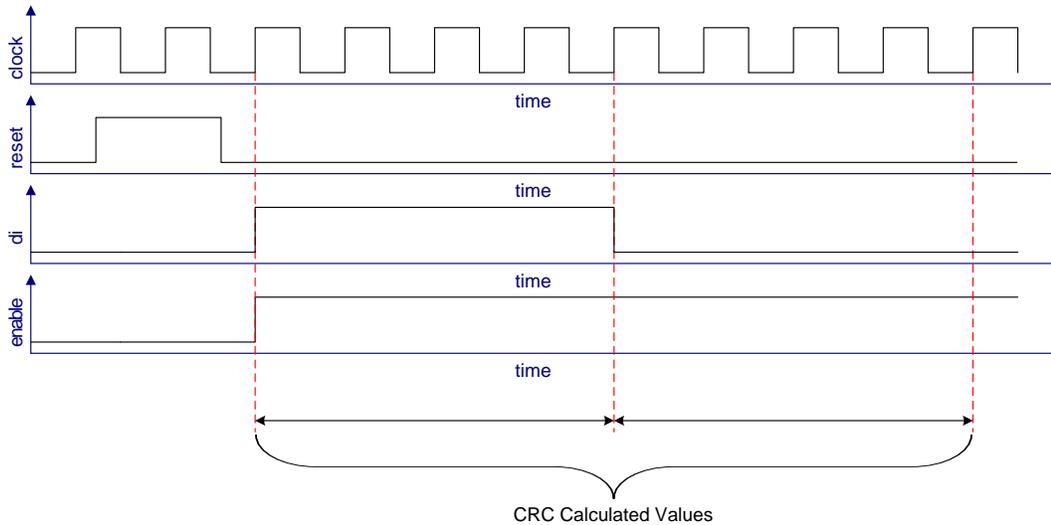
Note that the initial seed value is lost. This is usually of no consequence because the seed value is only used to initialize the Shift register once, for each data set.

## Block Diagram and Configuration

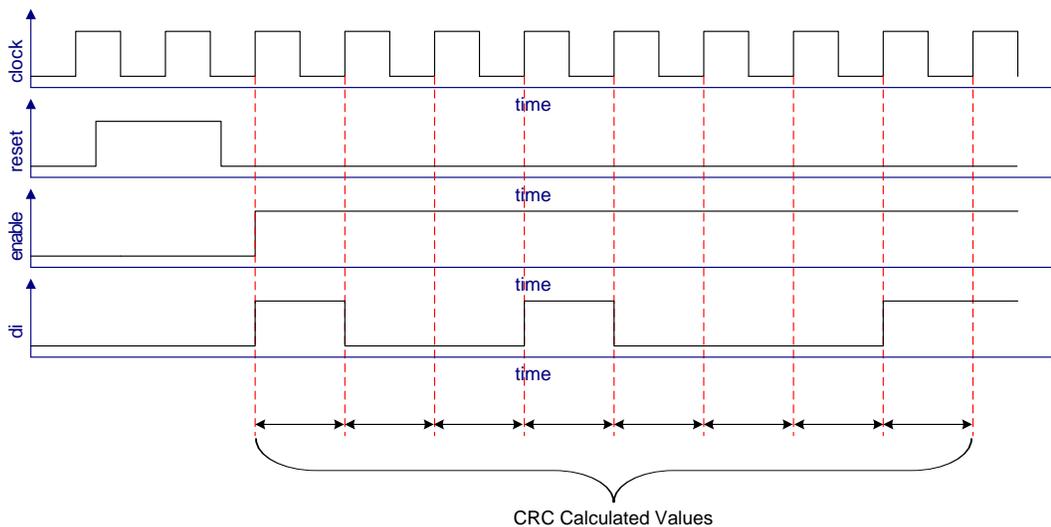


## Timing Diagrams

**Figure 1. Time Division Multiplex Implementation Mode**



**Figure 2. Single Cycle Implementation Mode**



## DC and AC Electrical Characteristics

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



## Timing Characteristics “Maximum with Nominal Routing”

Parameter	Description	Config. <sup>1</sup>	Min	Typ	Max	Units
f <sub>CLOCK</sub>	Component clock Frequency <sup>2</sup>	Config 1			45	MHz
		Config 2			30	MHz
		Config 3			41	MHz
		Config 4			24	MHz
		Config 5			35	MHz
		Config 6			21	MHz
t <sub>CLOCKH</sub>	Input clock high time <sup>3</sup>	N/A		0.5		1/f <sub>CLOCK</sub>
t <sub>CLOCKL</sub>	Input clock low time <sup>3</sup>	N/A		0.5		1/f <sub>CLOCK</sub>
<b>Inputs</b>						
t <sub>PD_ps</sub>	Input path delay, pin to sync <sup>4</sup>	1			STA <sup>5</sup>	ns
t <sub>PD_ps</sub>	Input path delay, pin to sync <sup>6</sup>	2			8.5	ns

### <sup>1</sup> Configurations:

#### Config 1:

Resolution: 8 bits  
Implementation: Single Cycle

#### Config 2:

Resolution: 16 bits  
Implementation: Single Cycle

#### Config 3:

Resolution: 16 bits  
Implementation: Time Division Multiplex

#### Config 4:

Resolution: 32 bits  
Implementation: Single Cycle

#### Config 5:

Resolution: 32 bits  
Implementation: Time Division Multiplex

#### Config 6:

Resolution: 64 bits  
Implementation: Time Division Multiplex

<sup>2</sup> If Time Division Multiplex Implementation is selected, then component clock frequency must be four times greater than the data rate.

<sup>3</sup> t<sub>CY\_clock</sub> = 1/f<sub>CLOCK</sub>. This is the cycle time of one clock period.

<sup>4</sup> t<sub>PD\_ps</sub> 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.

<sup>5</sup> t<sub>PD\_ps</sub> and t<sub>PD\_si</sub> are route path delays. Because routing is dynamic, these values can change will directly affect the maximum component clock and sync clock frequencies. The values must be found in the Static Timing Analysis results.

<sup>6</sup> t<sub>PD\_ps</sub> 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



Parameter	Description	Config. <sup>1</sup>	Min	Typ	Max	Units
t <sub>PD_si</sub>	Sync output to input path delay (route)	1,2,3,4			STA <sup>5</sup>	ns
t <sub>l_clk</sub>	Alignment of clockX and clock	1,2,3,4	0		1	t <sub>CY_clock</sub>
t <sub>PD_IE</sub>	Input path delay to component clock (edge-sensitive input)	1,2	t <sub>PD_ps</sub> + t <sub>SYNC</sub> + t <sub>PD_si</sub>		t <sub>PD_ps</sub> + t <sub>SYNC</sub> + t <sub>PD_si</sub> + t <sub>l_clk</sub>	ns
t <sub>PD_IE</sub>	Input path delay to component clock (edge-sensitive input)	3,4	t <sub>sync</sub> + t <sub>PD_si</sub>		t <sub>sync</sub> + t <sub>PD_si</sub> + t <sub>l_clk</sub>	ns
t <sub>IH</sub>	Input high time	1,2,3,4	t <sub>CY_clock</sub> <sup>7</sup>			ns
t <sub>IL</sub>	Input low time	1,2,3,4	t <sub>CY_clock</sub> <sup>7</sup>			ns

<sup>7</sup> t<sub>CY\_clock</sub> = 4 × [1/f<sub>CLOCK</sub>] if Time Division Multiplex Implementation is selected.

## Timing Characteristics “Maximum with All Routing”

Parameter	Description	Config. <sup>1</sup>	Min	Typ	Max <sup>2</sup>	Units
f <sub>CLOCK</sub>	Component clock frequency <sup>3</sup>	Config 1			23	MHz
		Config 2			15	MHz
		Config 3			21	MHz
		Config 4			12	MHz
		Config 5			18	MHz
		Config 6			11	MHz
T <sub>CLOCKH</sub>	Input clock high time <sup>4</sup>	N/A		0.5		1/f <sub>CLOCK</sub>
T <sub>CLOCKL</sub>	Input clock low time <sup>4</sup>	N/A		0.5		1/f <sub>CLOCK</sub>
<b>Inputs</b>						
t <sub>PD_ps</sub>	Input path delay, pin to sync <sup>5</sup>	1			STA <sup>6</sup>	ns

### <sup>1</sup>Configurations:

#### Config 1:

Resolution: 8 bits  
Implementation: Single Cycle

#### Config 2:

Resolution: 16 bits  
Implementation: Single Cycle

#### Config 3:

Resolution: 16 bits  
Implementation: Time Division Multiplex

#### Config 4:

Resolution: 32 bits  
Implementation: Single Cycle

#### Config 5:

Resolution: 32 bits  
Implementation: Time Division Multiplex

#### Config 6:

Resolution: 64 bits  
Implementation: Time Division Multiplex

<sup>2</sup> Maximum for “All Routing” is calculated by <nominal>/2 rounded to the nearest integer. This value allows you not to worry about meeting timing if the component is running at or below this frequency.

<sup>3</sup> If Time Division Multiplex Implementation is selected, then component clock frequency must be four times greater than the data rate.

<sup>4</sup> t<sub>CY\_clock</sub> = 1/f<sub>CLOCK</sub> – Cycle time of one clock period.

<sup>5</sup> t<sub>PD\_ps</sub> 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.

<sup>6</sup> t<sub>PD\_ps</sub> and t<sub>PD\_si</sub> 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 must be found in the Static Timing Analysis results.



Parameter	Description	Config. <sup>1</sup>	Min	Typ	Max <sup>2</sup>	Units
t <sub>PD_ps</sub>	Input path delay, pin to sync <sup>7</sup>	2			8.5	ns
t <sub>PD_si</sub>	Sync output to input path delay (route)	1,2,3,4			STA <sup>5</sup>	ns
t <sub>l_clk</sub>	Alignment of clockX and clock	1,2,3,4	0		1	t <sub>CY_clock</sub>
t <sub>PD_IE</sub>	Input path delay to component clock (edge-sensitive input)	1,2	t <sub>PD_ps</sub> + t <sub>SYNC</sub> + t <sub>PD_si</sub>		t <sub>PD_ps</sub> + t <sub>SYNC</sub> + t <sub>PD_si</sub> + t <sub>l_clk</sub>	ns
t <sub>PD_IE</sub>	Input path delay to component clock (edge-sensitive input)	3,4	t <sub>SYNC</sub> + t <sub>PD_si</sub>		t <sub>SYNC</sub> + t <sub>PD_si</sub> + t <sub>l_clk</sub>	ns
t <sub>IH</sub>	Input high time	1,2,3,4	t <sub>CY_clock</sub> <sup>8</sup>			ns
t <sub>IL</sub>	Input low time	1,2,3,4	t <sub>CY_clock</sub> <sup>8</sup>			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 with the STA results using the following methods:

f<sub>clock</sub> 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 *\_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 3](#).

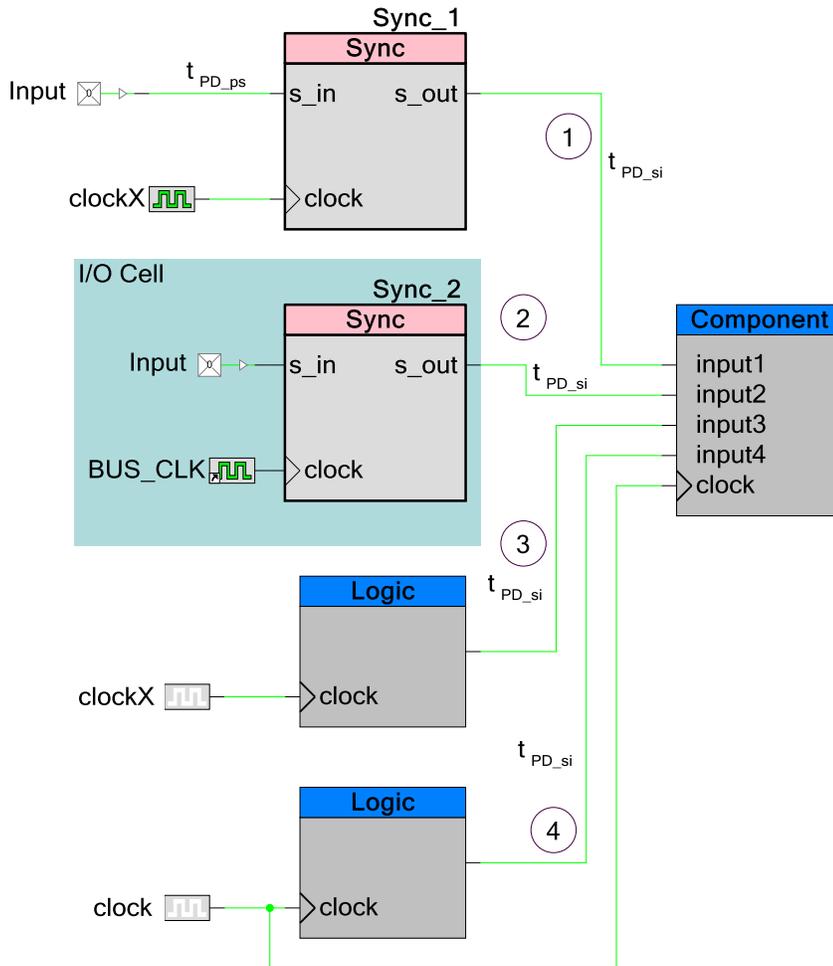
<sup>7</sup> t<sub>PD\_ps</sub> in configuration 2 is a fixed value defined per pin of the device. The number listed here is a nominal value of all of the pins available on the device

<sup>8</sup> t<sub>CY\_clock</sub> = 4 × [1/f<sub>CLOCK</sub>] if Time Division Multiplex Implementation is selected.



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 3. Input Configurations for Component Timing Specifications**



Configuration	Component Clock	Synchronizer Clock (Frequency)	Figures
1	master_clock	master_clock	Figure 8
1	clock	master_clock	Figure 6
1	clock	clockX = clock <sup>1</sup>	Figure 4

<sup>1</sup> Clock frequencies are equal but alignment of rising edges is not guaranteed.

Configuration	Component Clock	Synchronizer Clock (Frequency)	Figures
1	clock	clockX > clock	<a href="#">Figure 5</a>
1	clock	clockX < clock	<a href="#">Figure 7</a>
2	master_clock	master_clock	<a href="#">Figure 8</a>
2	clock	master_clock	<a href="#">Figure 6</a>
3	master_clock	master_clock	<a href="#">Figure 13</a>
3	clock	master_clock	<a href="#">Figure 11</a>
3	clock	clockX = clock <sup>1</sup>	<a href="#">Figure 9</a>
3	clock	clockX > clock	<a href="#">Figure 10</a>
3	clock	clockX < clock	<a href="#">Figure 12</a>
4	master_clock	master_clock	<a href="#">Figure 13</a>
4	clock	clock	<a href="#">Figure 9</a>

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).

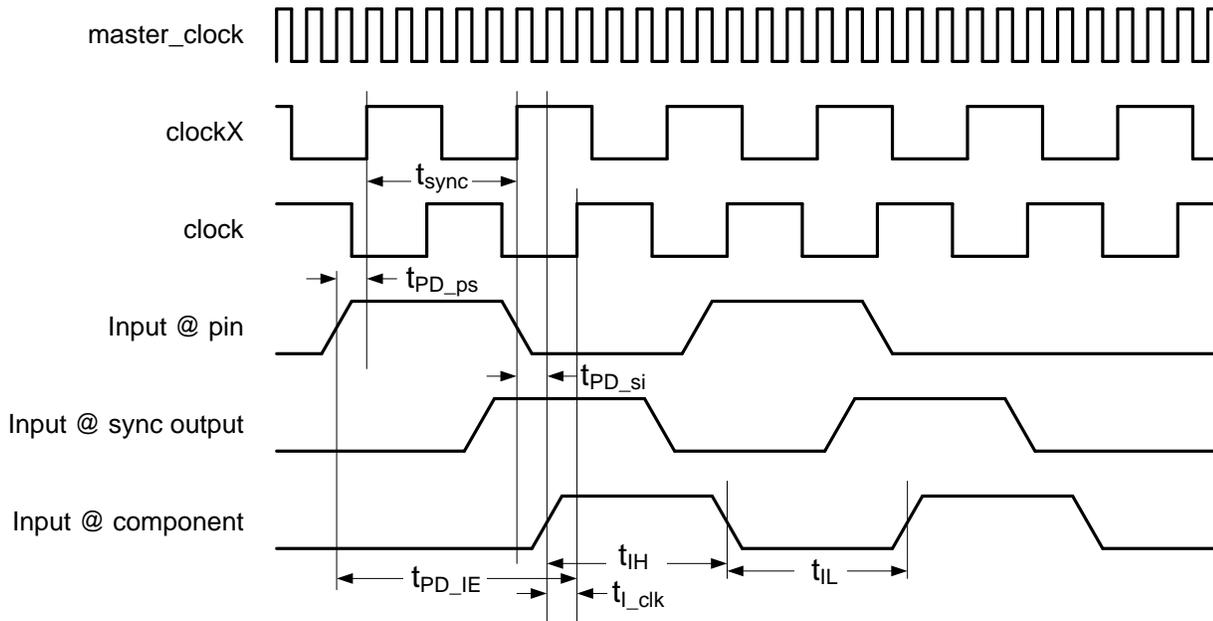
When characterizing inputs configured in this way, clockX may be faster than, equal to, or slower than the component clock. It may also be equal to master\_clock, which produces the characterization parameters shown in [Figure 4](#), [Figure 5](#), [Figure 7](#), and [Figure 8](#).

2. The input is driven by a device pin and synchronized at the pin using master\_clock.

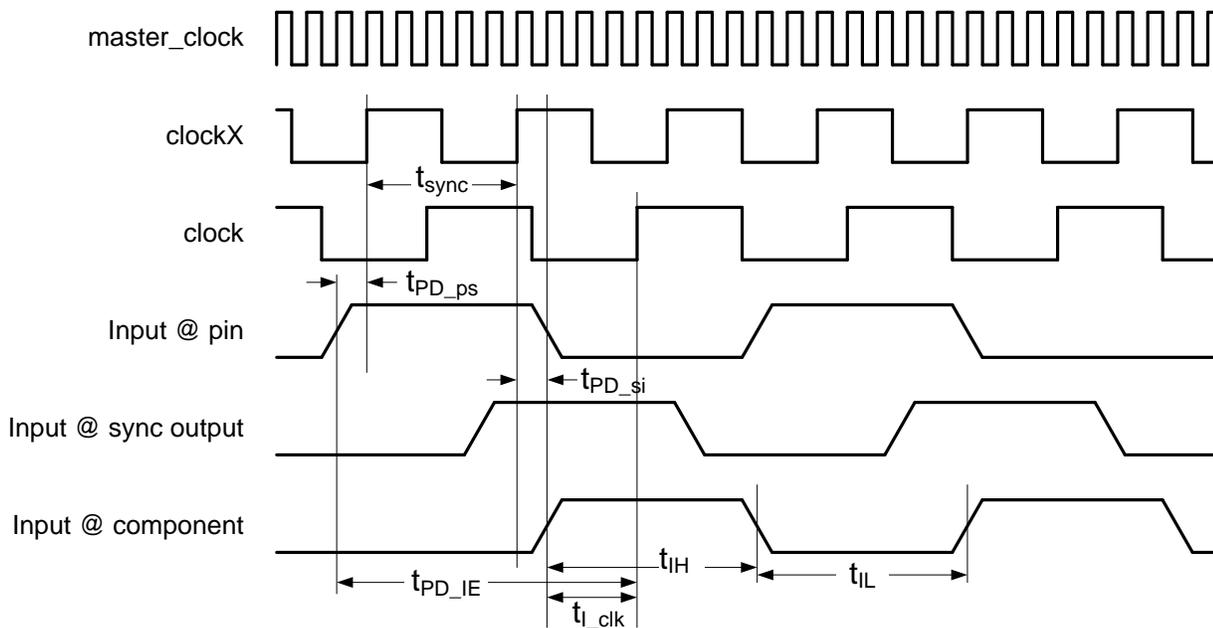
When characterizing inputs configured in this way, master\_clock is faster than or equal to the component clock (it is never slower than). This produces the characterization parameters shown in [Figure 5](#) and [Figure 8](#).



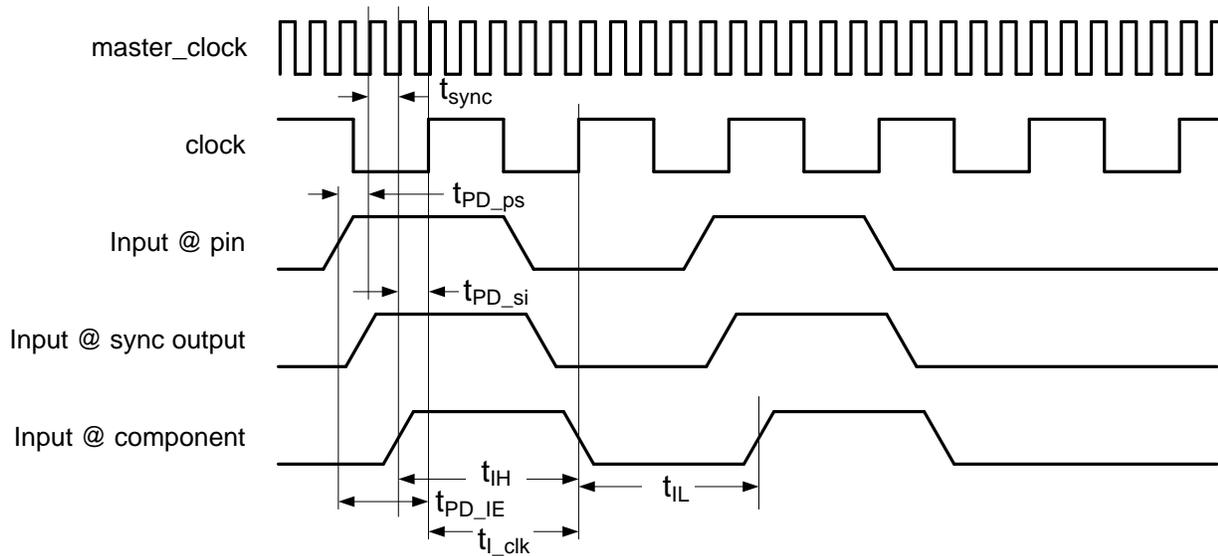
**Figure 4. Input Configuration 1 and 2; Synchronizer Clock Frequency = Component Clock Frequency (Edge alignment of clock and clockX is not guaranteed)**



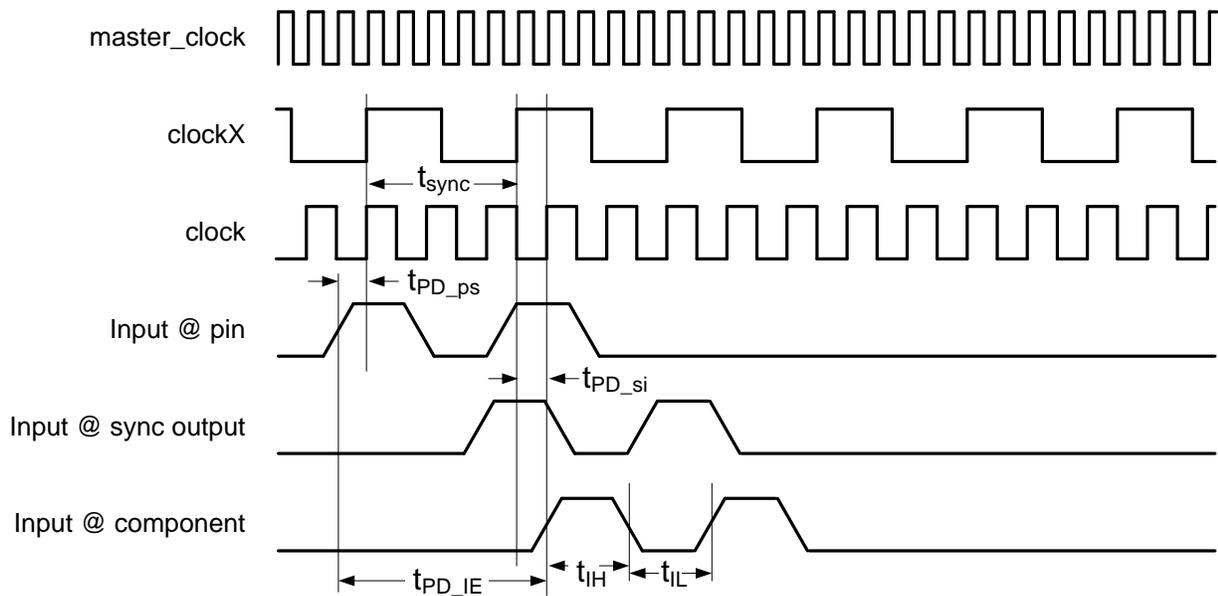
**Figure 5. Input Configuration 1 and 2; Synchronizer Clock Frequency > Component Clock Frequency**



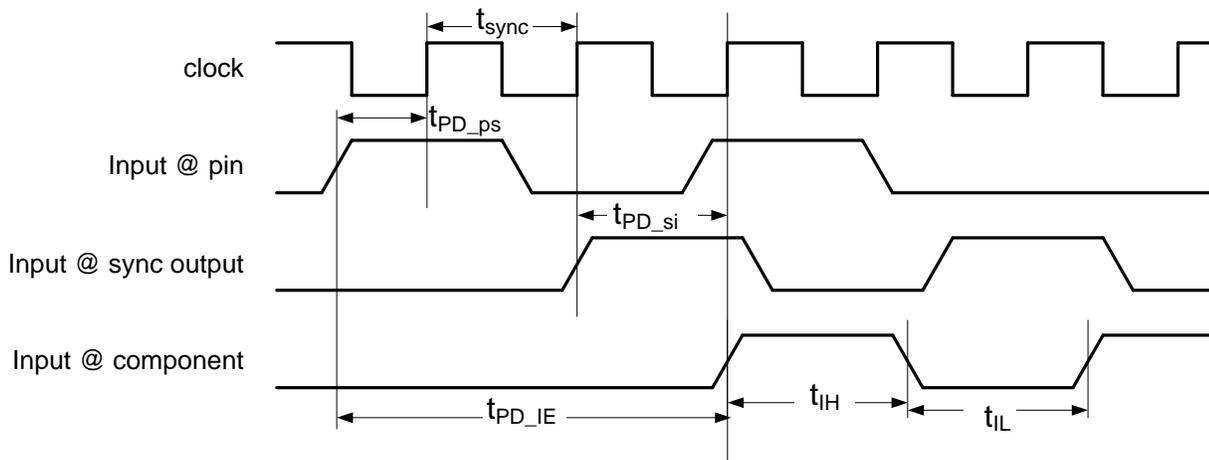
**Figure 6. Input Configuration 1 and 2; [Synchronizer Clock Frequency == master\_clock] > Component Clock Frequency**



**Figure 7. Input Configuration 1; Synchronizer Clock Frequency < Component Clock Frequency**

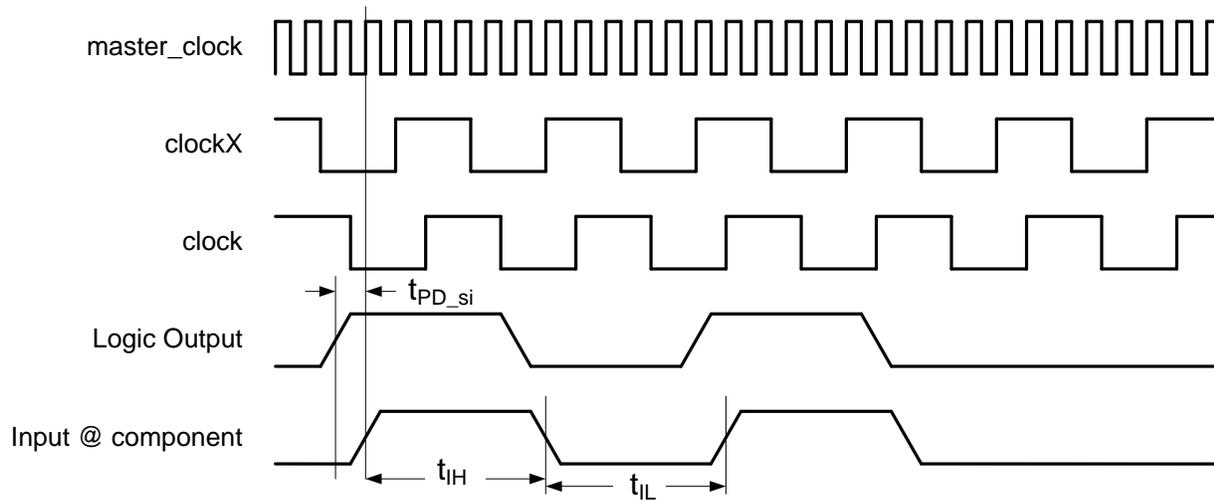


**Figure 8. Input Configuration 1 and 2; Synchronizer Clock = Component Clock = master\_clock**



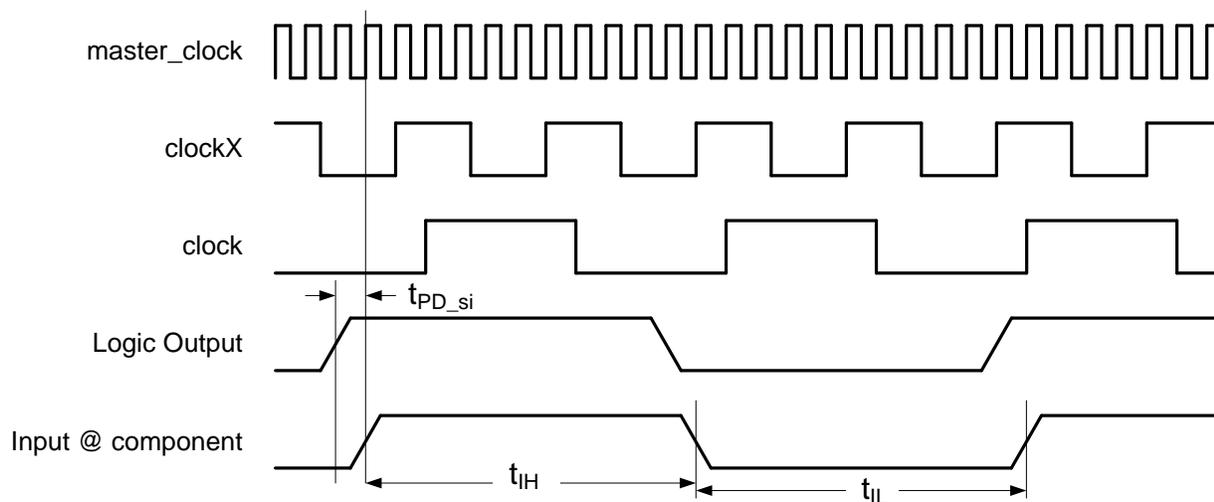
3. The input is driven by logic internal to the PSoC, which is synchronous based on a clock other than the clock the component uses (all internal clocks are derived from master\_clock).  
When characterizing inputs configured in this way, the synchronizer clock is faster than, less than, or equal to the component clock, which produces the characterization parameters shown in [Figure 9](#), [Figure 10](#), and [Figure 12](#).
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, which will produce the characterization parameters as shown in [Figure 13](#).

**Figure 9. Input Configuration 3 only; Synchronizer Clock Frequency = Component Clock Frequency (Edge alignment of clock and clockX is not guaranteed)**



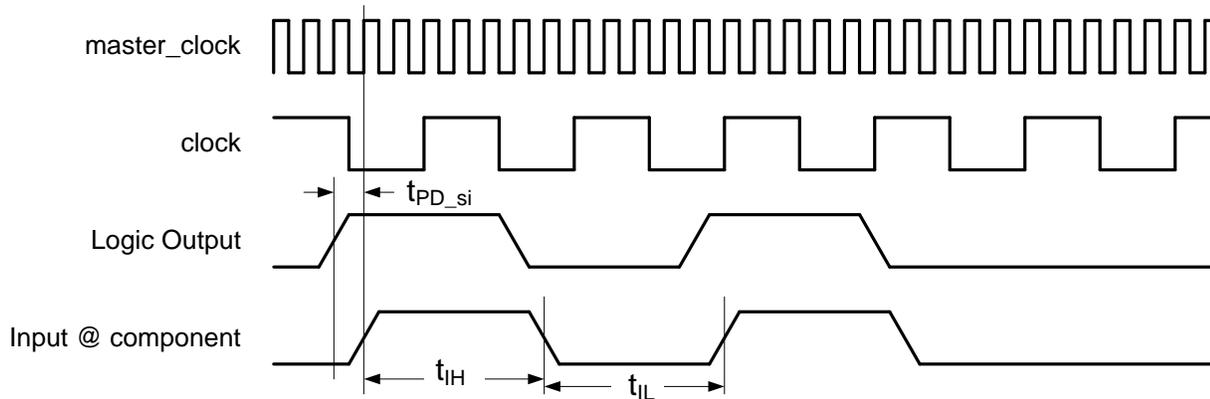
This figure represents the understanding that Static Timing Analysis holds on the clocks. All clocks in the digital clock domain are synchronous to master\_clock. However, it is possible that two clocks with the same frequency are not rising-edge-aligned. Therefore, the static timing analysis tool does not know which edge the clocks are synchronous to and must assume the minimum of 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 10. Input Configuration 3; Synchronizer Clock Frequency > Component Clock Frequency**

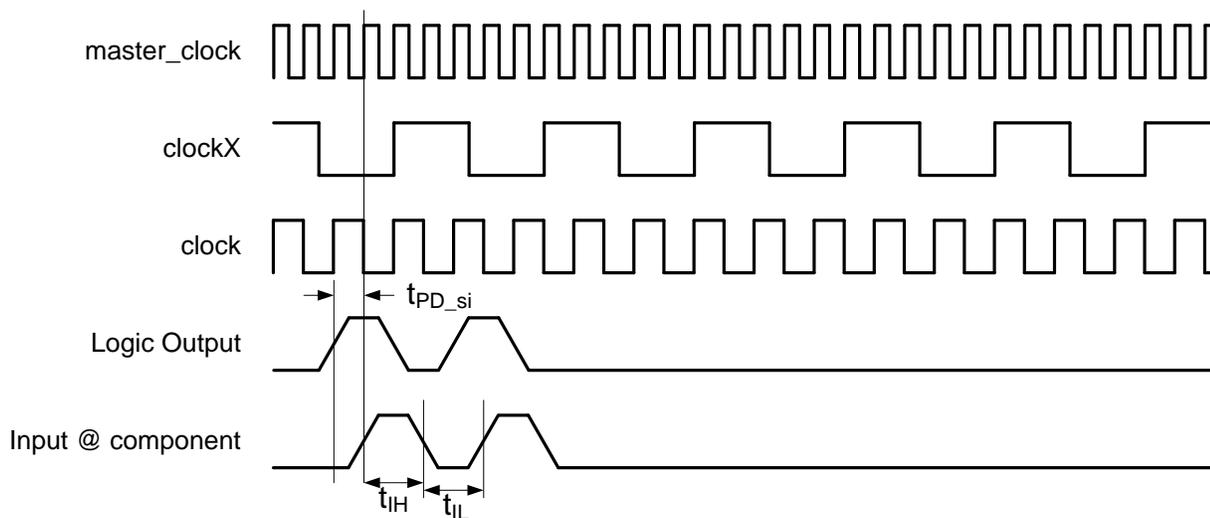


In much the same way as shown in [Figure 9](#), all clocks are derived from `master_clock`. STA indicates the  $t_{PD\_si}$  limitations on `master_clock` for one `master_clock` cycle in this configuration. `master_clock` setup time violations appear if this path delay is too long. You must change the synchronization clocks of your system or run the `master_clock` at a slower frequency.

**Figure 11. Input Configuration 3; Synchronizer Clock Frequency = `master_clock` > Component Clock Frequency**

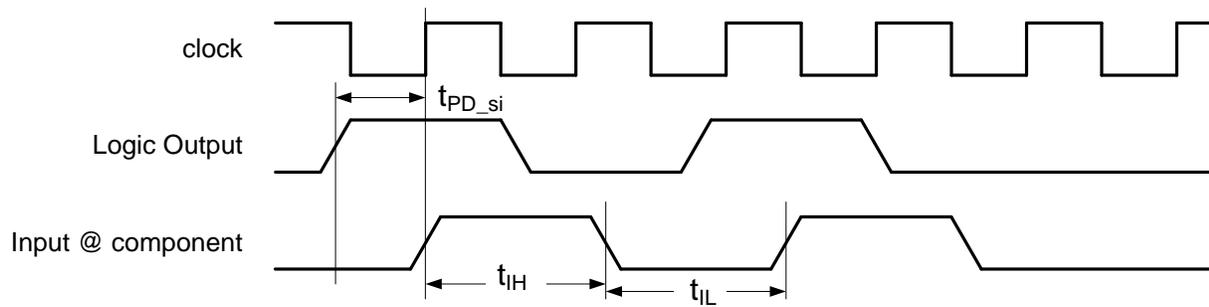


**Figure 12. Input Configuration 3; Synchronizer Clock Frequency < Component Clock Frequency**



In much the same way as shown in [Figure 9](#), 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 13. 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  $f_{\text{CLOCK}}$  and  $t_{\text{PD\_IE}}$ .  $t_{\text{PD\_IE}}$  is defined by  $t_{\text{PD\_ps}}$  and  $t_{\text{SYNC}}$  (for configurations 1 and 2 only),  $t_{\text{PD\_si}}$ , and  $t_{\text{I\_Clk}}$ . Of critical importance is the fact that  $t_{\text{PD\_si}}$  defines the maximum component clock frequency.  $t_{\text{I\_Clk}}$  does not come from the STA results but is used to represent when  $t_{\text{PD\_IE}}$  is registered. This is the margin left over after the route between the synchronizer and the component clock.

$t_{\text{PD\_ps}}$  and  $t_{\text{PD\_si}}$  are included in the STA results.

To find  $t_{\text{PD\_ps}}$ , look at the input setup times defined in the `_timing.html` file. The fanout of this input may be more than 1 so you 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

$t_{\text{PD\_si}}$  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 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.d	Minor datasheet edit.	
2.0.c	Minor datasheet edit.	
2.0.b	Minor datasheet edits and updates	
2.0.a	Added characterization data to datasheet	
	Minor datasheet edits and updates	
2.0	Added support for PSoC 3 ES3 silicon. Changes include: <ul style="list-style-type: none"> <li>4x clock for Time Division Multiplex Implementation added</li> <li>Single Cycle Implementation on 1x clock now available for 1 to 32 bits.</li> <li>Time Division Multiplex Implementation on 4x clock now available for 9 to 64 bits.</li> <li>Asynchronous input signal reset is added.</li> <li>Synchronous input signal enable is added.</li> <li>Added new 'Advanced' page to the Configure dialog for the Implementation (Time Division Multiplex, Single Cycle) parameter</li> </ul>	New requirements to support the PSoC 3 ES3 device, thus a new 2.0 version of the CRC component was created.
	Added CRC_Sleep()/CRC_Wakeup() and CRC_Init()/CRC_Enable() APIs.	To support low-power modes, as well as to provide common interfaces to separate control of initialization and enabling of most components.
	Updated functions CRC_WriteSeed() and CRC_WriteSeedUpper().	The mask parameter was used to cut the seed value to define CRC resolution while writing.
	Add validator to Resolution parameter.	The resolution of CRC is 1 to 64 bits. The validator was added to restrict input values.

Version	Description of Changes	Reason for Changes / Impact
	Add reset DFF triggers to polynomial write functions: CRC_WritePolynomial(), CRC_WritePolynomialUpper() and CRC_WritePolynomialLower().	The DFF triggers need to be set in proper state (most significant bit of polynomial, always 1) before CRC calculation starts. To meet this condition, any write to the Seed or Polynomial registers resets the DFF triggers.
	Updated Configure dialog to allow the Expression View for the following parameters: 'PolyValueLower', 'PolyValueUpper', 'SeedValueLower', 'SeedValueUpper'	Expression View is used to directly access the symbol parameters. This view allows you to connect component parameters with external parameters, if desired.
	Updated Configure dialog to add error icons for various parameters.	If you enter an incorrect value in a text box, the error icon displays with a tool tip of the problem description. This provides easier use than a separate error message.
1.20	Changed method of API generation. In version 1.10, APIs were generated by settings from the customizer. For 1.20, APIs are provided by the .c and .h files like most other components.	This change allows users to view and make changes to the generated API files, and they will not be overwritten on subsequent builds.
	Seed and Polynomial parameters were changed to have hexadecimal representation.	Change was made to comply with corporate standard.

© Cypress Semiconductor Corporation, 2009-2016. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit [cypress.com](http://cypress.com). Other names and brands may be claimed as property of their respective owners.

