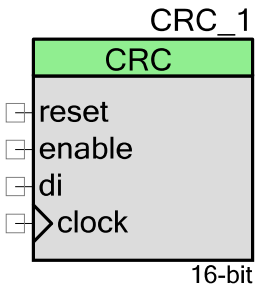
# Cyclic Redundancy Check (CRC)

2.10

## 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 the 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

You can use the default CRC component 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 the 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.

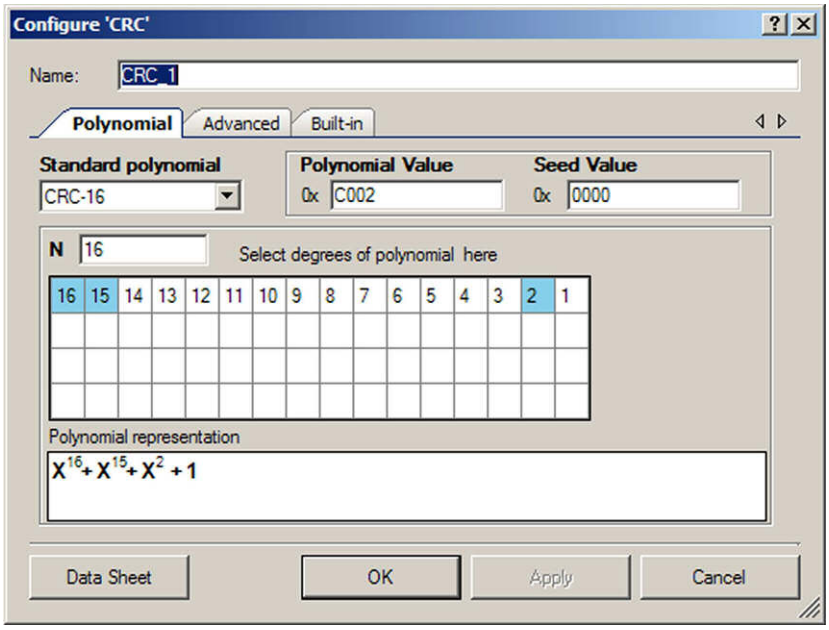
## di – Input

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.

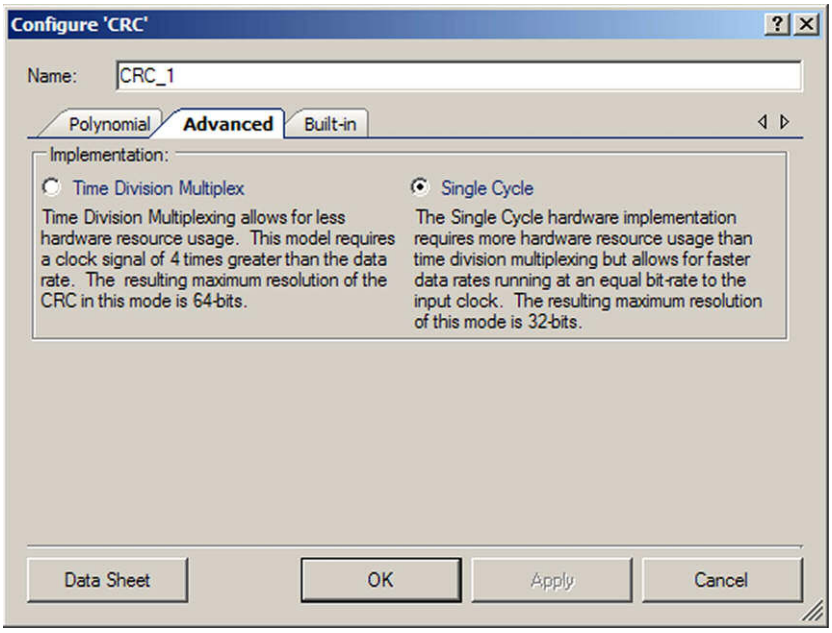
### Use Polynomial Degrees

- 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 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 | 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.<br>If reinitialization of the component is required, then the CRC_Init() function can be called before the CRC_Start() or CRC_Enable() function. |

## void CRC_Start(void)

**Description:** Initializes seed and polynomial registers with initial values. Computation of CRC starts on rising edge of input clock.

**Parameters:** None

**Return Value:** None

**Side Effects:** None

## void CRC_Stop(void)

**Description:** Stops CRC computation.

**Parameters:** None

**Return Value:** None

**Side Effects:** None

## void CRC_Sleep(void)

**Description:** Stops CRC computation and saves the CRC configuration.

**Parameters:** None

**Return Value:** None

**Side Effects:** None

## void CRC_Wakeup(void)

**Description:** Restores the CRC configuration and starts CRC computation on the rising edge of the input clock.

**Parameters:** None

**Return Value:** None

**Side Effects:** None

## void CRC_Init(void)

**Description:** Initializes the seed and polynomial registers with initial values.

**Parameters:** None

**Return Value:** None

**Side Effects:** None

## void CRC_Enable(void)

**Description:** Starts CRC computation on the rising edge of the input clock.

**Parameters:** None

**Return Value:** None

**Side Effects:** None

## void CRC_SaveConfig(void)

**Description:** Saves the initial seed and polynomial registers.

**Parameters:** None

**Return Value:** None

**Side Effects:** None

## void CRC_RestoreConfig(void)

**Description:** Restores the initial seed and polynomial registers.

**Parameters:** None

**Return Value:** None

**Side Effects:** 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 = $2^{Resolution} - 1$.
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 = $2^{Resolution - 32} - 1$.
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)

**Description:** Reads the upper half of the CRC value. Only generated for 33- to 64-bit CRC.

**Parameters:** None

**Return Value:** uint32: Returns the upper half of the CRC value

**Side Effects:** None

## uint32 CRC_ReadCRCLower(void)

**Description:** Reads the lower half of the CRC value. Only generated for 33- to 64-bit CRC.

**Parameters:** None

**Return Value:** uint32: Returns the lower half of the CRC value

**Side Effects:** None

## void CRC_WritePolynomial(uint8/16/32 polynomial)

**Description:** Writes the CRC polynomial value.

**Parameters:** uint8/16/32 polynomial: CRC polynomial

**Return Value:** None

**Side Effects:** The polynomial value is cut according to mask = $2^{Resolution} - 1$. 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)

**Description:** Writes the upper half of the CRC polynomial value. Only generated for 33- to 64-bit CRC.

**Parameters:** uint32 polynomial: Upper half of the CRC polynomial value

**Return Value:** None

**Side Effects:** The upper half of the polynomial value is cut according to mask = $2^{(Resolution - 32)} - 1$. 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)

**Description:** Writes the lower half of the CRC polynomial value. Only generated for 33- to 64-bit CRC.

**Parameters:** uint32 polynomial: Lower half of the CRC polynomial value

**Return Value:** None

**Side Effects:** None

## uint8/16/32 CRC_ReadPolynomial(void)

**Description:** Reads the CRC polynomial value.

**Parameters:** None

**Return Value:** uint8/16/32: Returns the CRC polynomial value

**Side Effects:** None

## uint32 CRC_ReadPolynomialUpper(void)

**Description:** Reads the upper half of the CRC polynomial value. Only generated for 33- to 64-bit CRC.

**Parameters:** None

**Return Value:** uint32: Returns the upper half of the CRC polynomial value

**Side Effects:** None

## uint32 CRC_ReadPolynomialLower(void)

**Description:** Reads the lower half of the CRC polynomial value. Only generated for 33- to 64-bit CRC.

**Parameters:** None

**Return Value:** uint32: Returns the lower half of the CRC polynomial value.

**Side Effects:** None

# Sample Firmware Source Code

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

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

# Functional Description

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.

The seed and polynomial registers must be initialized before starting the component.

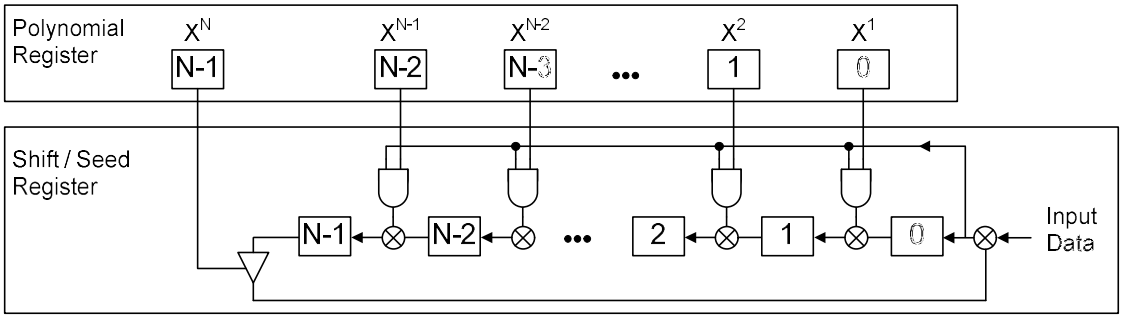
Computation of an N-bit LFSR result is specified by a polynomial with N + 1 terms, the last of which is the $X^0$ term where $X^0 = 1$. For example, the widely used CRC-CCITT 16-bit polynomial is $X^{16} + X^{12} + X^5 + 1$. The CRC algorithm assumes the presence of the $X^0$ term, so that the polynomial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

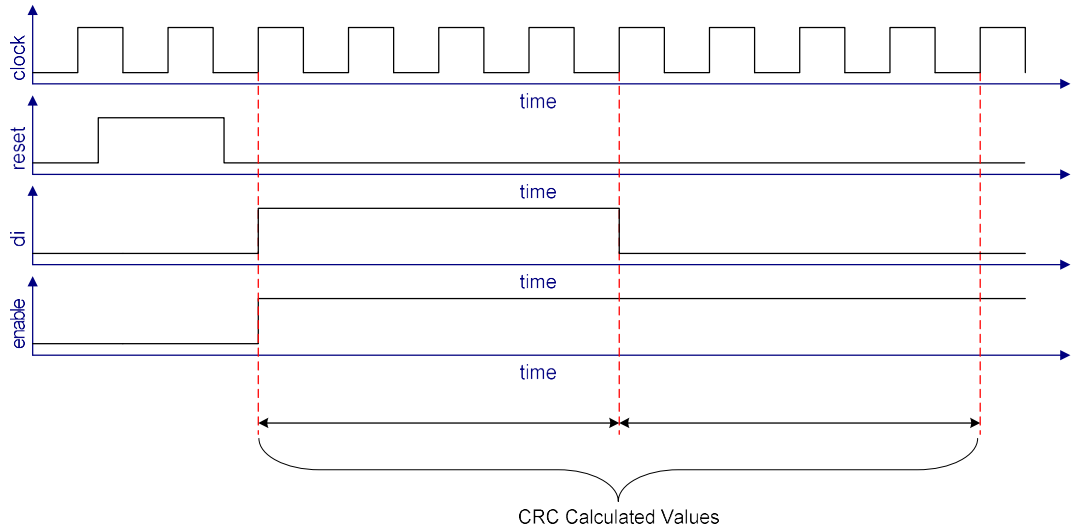
Note that the initial seed value is lost. This is usually of no consequence because the seed value is only used to initialize the Shift register once, for each data set.

# Block Diagram and Configuration

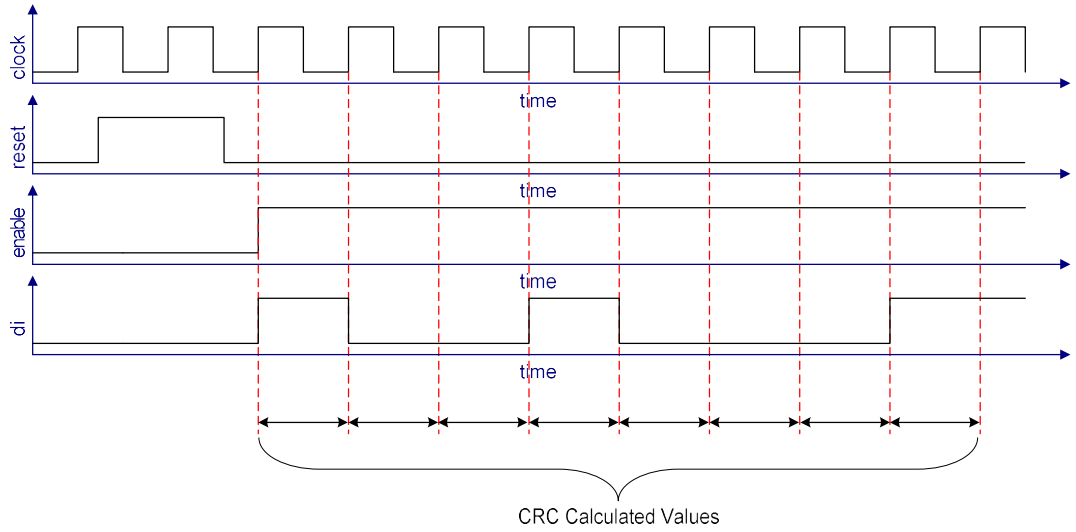


## Timing Diagrams

### 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.[1] | Min | Typ | Max | Units |
|---|---|---|---|---|---|---|
| $f_{CLOCK}$ | Component clock Frequency [2] | Config 1 | | | 45 | MHz |
| | | Config 2 | | | 30 | MHz |
| | | Config 3 | | | 41 | MHz |
| | | Config 4 | | | 24 | MHz |
| | | Config 5 | | | 35 | MHz |
| | | Config 6 | | | 21 | MHz |
| $t_{CLOCKH}$ | 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 |

[1] 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

[2] If Time Division Multiplex implementation is selected, component clock frequency must be four times greater than the data rate.

[3] $t_{CY_clock} = 1/f_{CLOCK}$. This is the 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 directly affect the maximum component clock and sync clock frequencies. The values are 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

| Parameter | Description | Config.[1] | Min | Typ | Max | Units |
|---|---|---|---|---|---|---|
| $t_{PD_si}$ | 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 | $t_{CY_clock}$ [7] | | | ns |

[7] $t_{CY_clock} = 4 \times [1/f_{CLOCK}]$ if Time Division Multiplex Implementation is selected.

## Timing Characteristics "Maximum with All Routing"

| Parameter | Description | Config. [8] | Min | Typ | Max [9] | Units |
|---|---|---|---|---|---|---|
| $f_{CLOCK}$ | Component clock frequency [10] | Config 1 | | | 23 | MHz |
| | | Config 2 | | | 15 | MHz |
| | | Config 3 | | | 21 | MHz |
| | | Config 4 | | | 12 | MHz |
| | | Config 5 | | | 18 | MHz |
| | | Config 6 | | | 11 | MHz |
| $T_{CLOCKH}$ | Input clock high time [11] | N/A | | 0.5 | | $1/f_{CLOCK}$ |
| $T_{CLOCKL}$ | Input clock low time [11] | N/A | | 0.5 | | $1/f_{CLOCK}$ |
| **Inputs** | | | | | | |
| $t_{PD_ps}$ | Input path delay, pin to sync [12] | 1 | | | STA [13] | ns |

[8] 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

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

[10] If Time Division Multiplex implementation is selected, component clock frequency must be four times greater than the data rate.

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

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

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

| Parameter | Description | Config. [8] | Min | Typ | Max [9] | Units |
|---|---|---|---|---|---|---|
| $t_{PD_ps}$ | Input path delay, pin to sync [14] | 2 | | | 8.5 | ns |
| $t_{PD_si}$ | Sync output to input path delay (route) | 1,2,3,4 | | | STA [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 | $t_{CY_clock}$ [15] | | | ns |
| $t_{IL}$ | Input low time | 1,2,3,4 | $t_{CY_clock}$ [15] | | | 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:

**$f_{CLOCK}$** 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 *_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

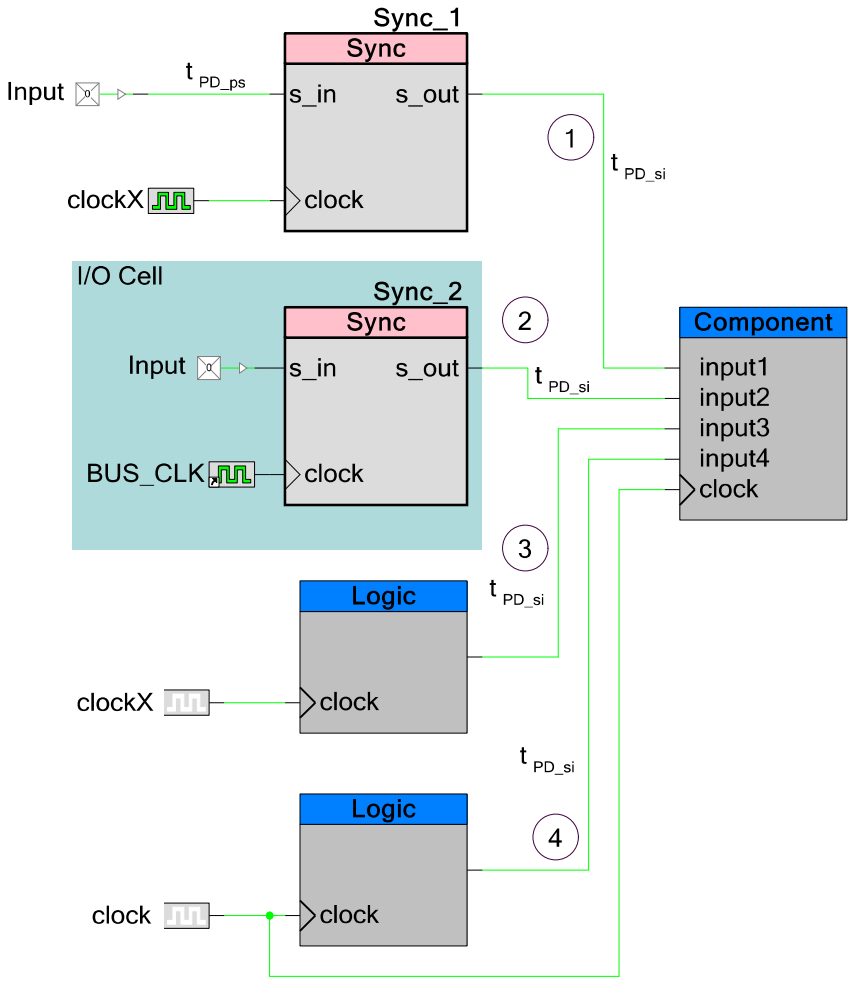
When characterizing the functionality of inputs, all inputs, no matter how you have configured them, look like one of four possible configurations, as shown in Figure 3.

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

[15] $t_{CY_clock} = 4 \times [1/f_{CLOCK}]$ 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 [16] | Figure 4 |

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

| Configuration | Component Clock | Synchronizer Clock (Frequency) | Figures |
|---|---|---|---|
| 1 | clock | clockX > clock | Figure 5 |
| 1 | clock | clockX < clock | Figure 7 |
| 2 | master_clock | master_clock | Figure 8 |
| 2 | clock | master_clock | Figure 6 |
| 3 | master_clock | master_clock | Figure 13 |
| 3 | clock | master_clock | Figure 11 |
| 3 | clock | clockX = clock [16] | Figure 9 |
| 3 | clock | clockX > clock | Figure 10 |
| 3 | clock | clockX < clock | Figure 12 |
| 4 | master_clock | master_clock | Figure 13 |
| 4 | clock | clock | Figure 9 |

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)**

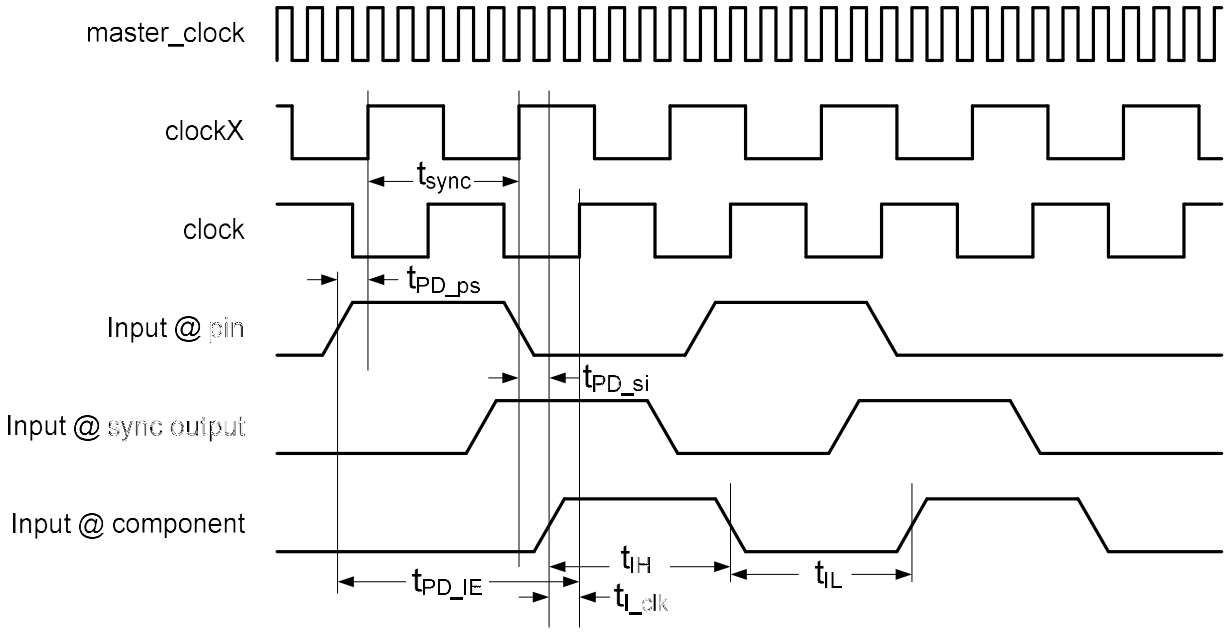



**Figure 5. Input Configuration 1 and 2; Synchronizer Clock Frequency > Component Clock Frequency**

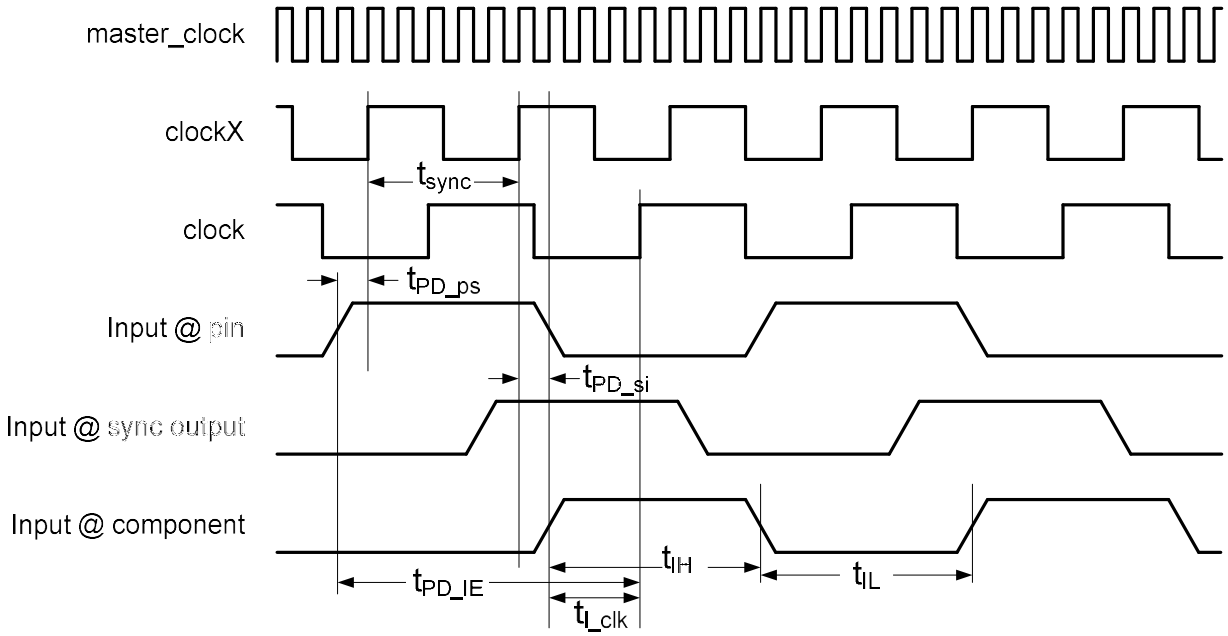


**Figure 6. Input Configuration 1 and 2; [Synchronizer Clock Frequency == master_clock] > Component Clock Frequency**

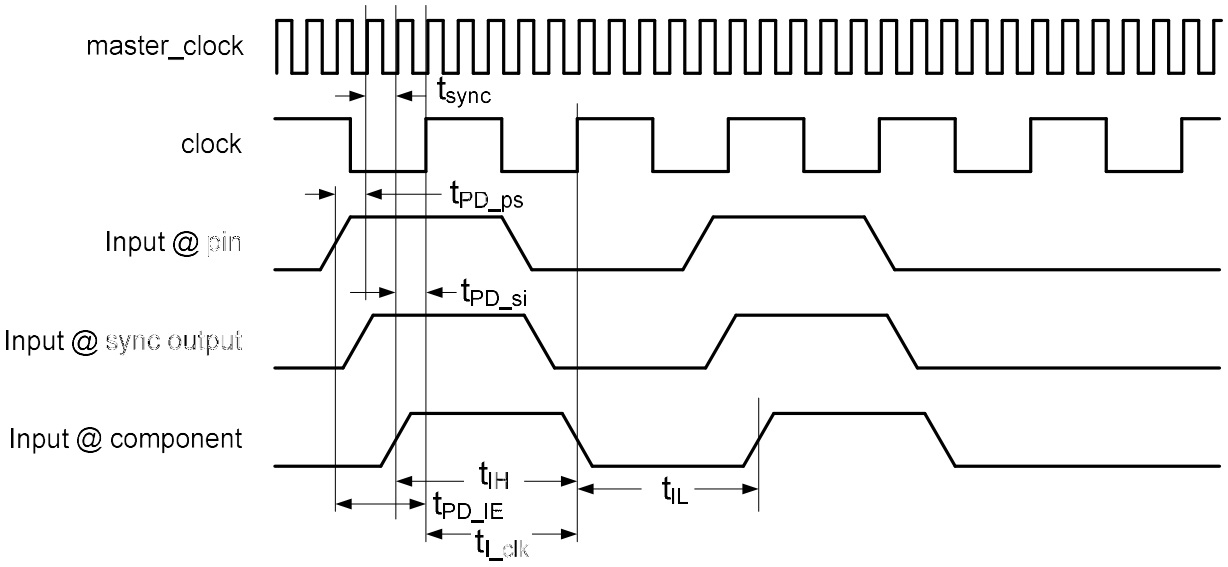



**Figure 7. Input Configuration 1; Synchronizer Clock Frequency < Component Clock Frequency**

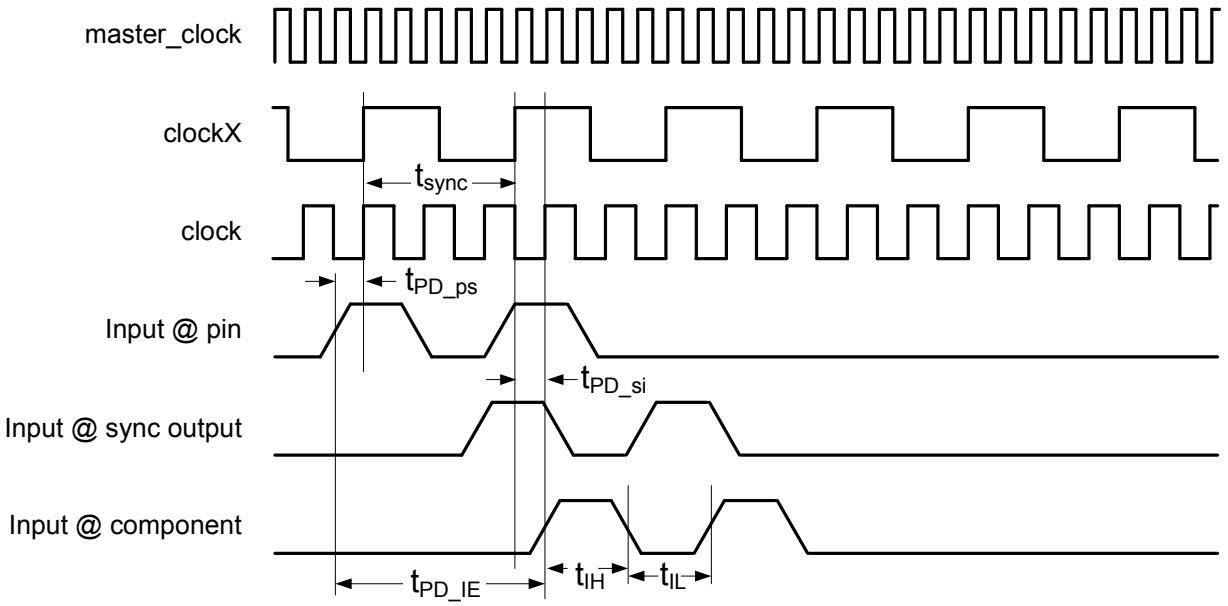
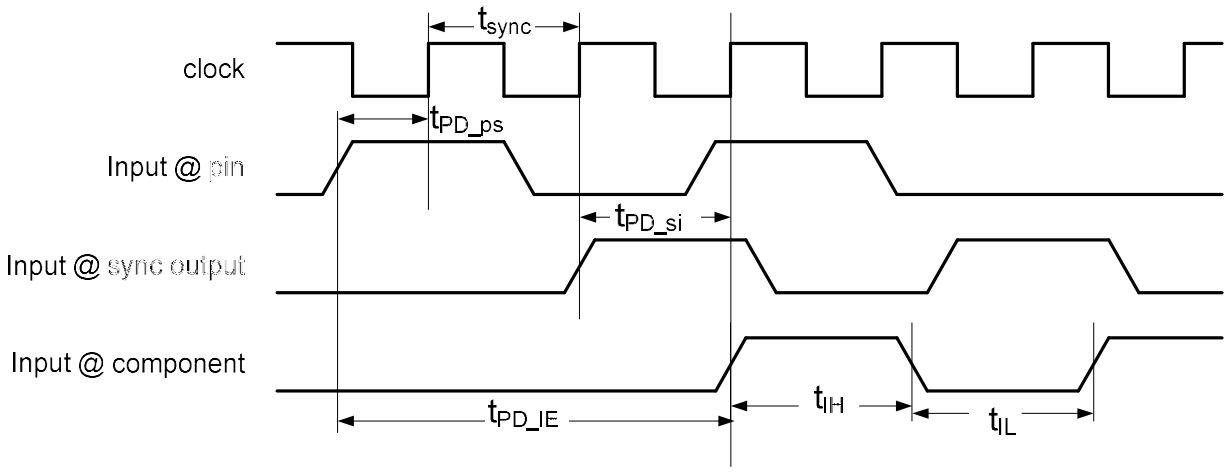


**Figure 8. Input Configuration 1 and 2; Synchronizer Clock = Component Clock = master_clock**



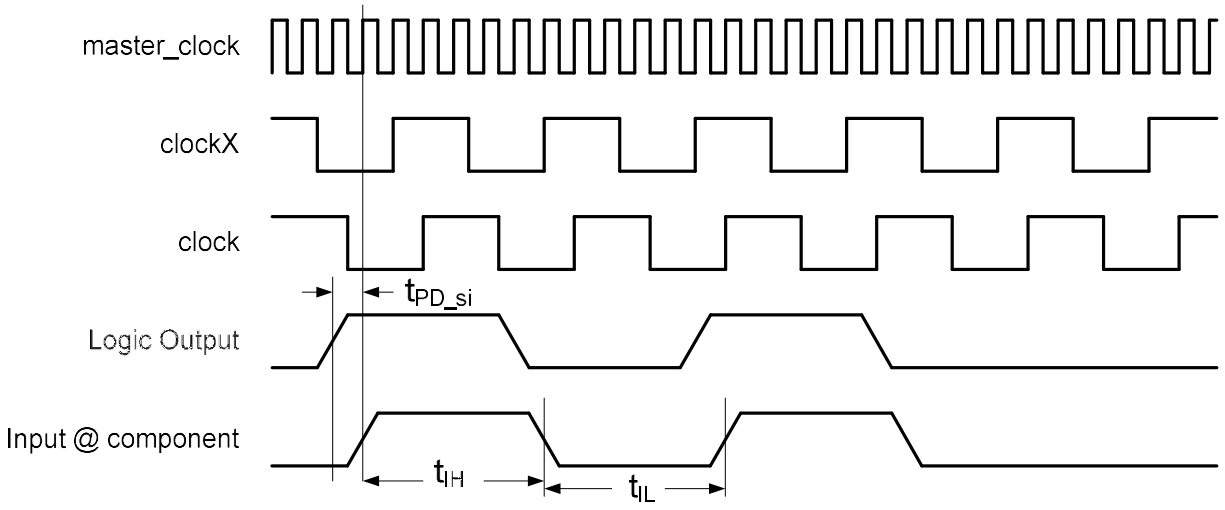

3. The input is driven by logic internal to the PSoC, which is synchronous based on a clock other than the clock the component uses (all internal clocks are derived from master_clock).

   When characterizing inputs configured in this way, the synchronizer clock is faster than, slower 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.

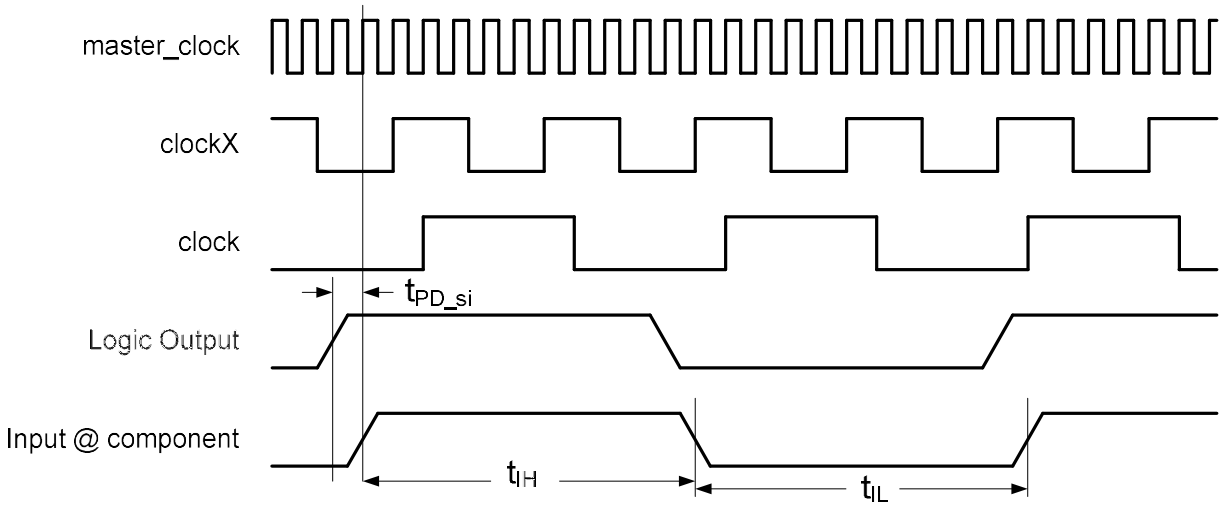
   When characterizing inputs configured in this way, the synchronizer clock is equal to the component clock, which produces the characterization parameters as shown in Figure 13.

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

**Figure 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**

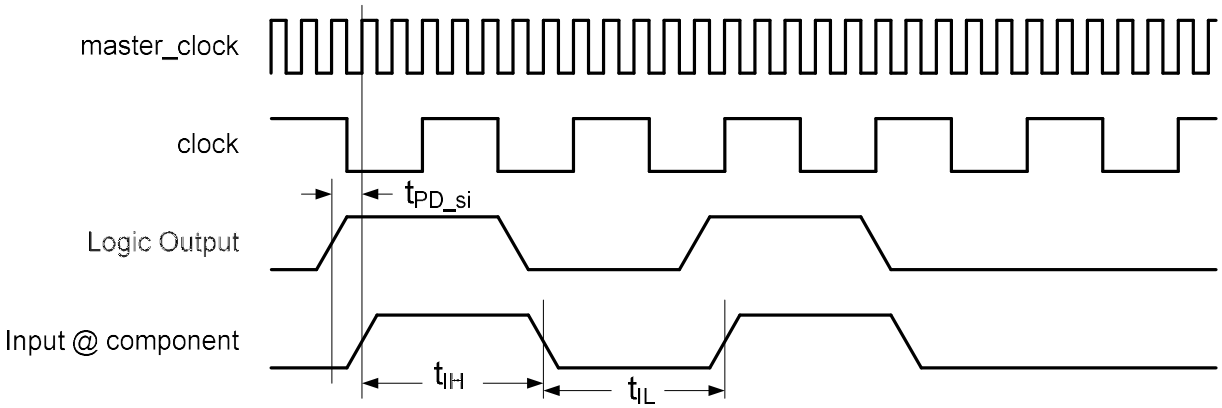
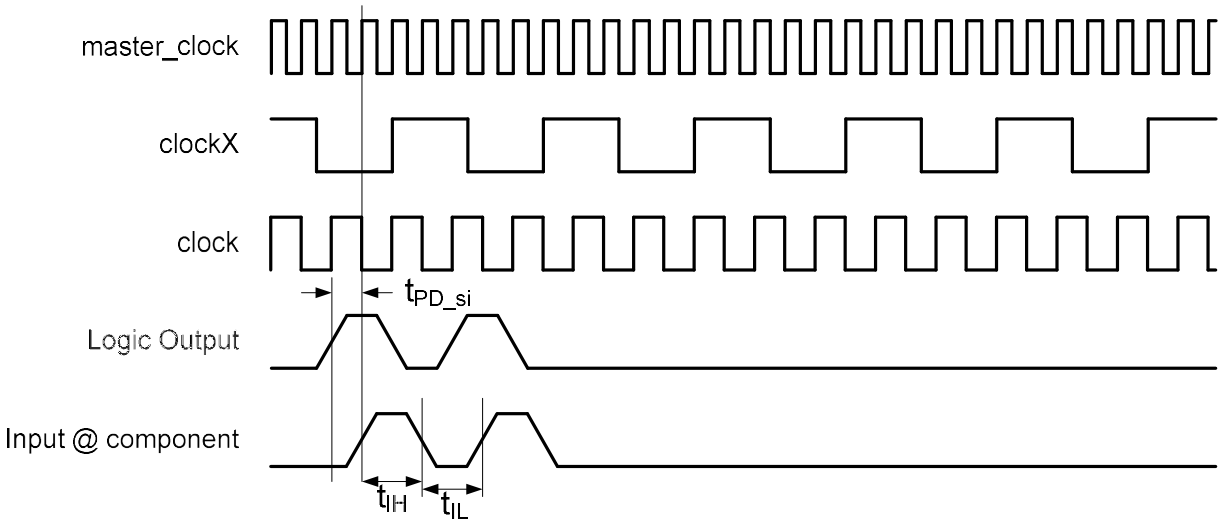
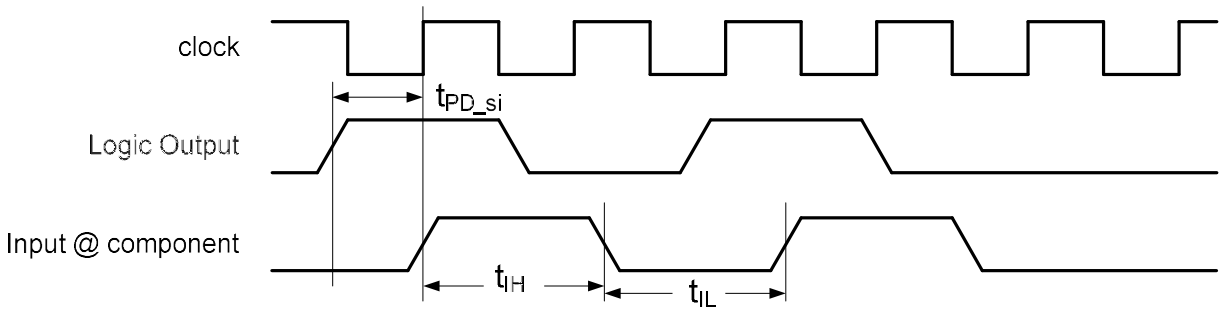



**Figure 12. Input Configuration 3; Synchronizer Clock Frequency < Component Clock Frequency**




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 $f_{CLOCK}$ 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 fanout of this input may be more than 1 so you need to evaluate the maximum of these paths.

**-Setup times**

**-Setup times to clock BUS_CLK**

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

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

**-Register-to-register times**

**-Destination clock clock**

Destination clock clock (Actual freq: 24.000 MHz)

**+Source clock clock**

**-Source clock clock_1**

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

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

### Output Path Delays

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

## Component Changes

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

| Version | Description of Changes | Reason for Changes / Impact |
|---|---|---|
| 2.10 | Changed error messages and their appearance for implementation parameter. | |
| | Fixed setting polynomial degree 'N' to 64-bit resolution. | |
| | Fixed polynomial value validation. | |
| 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:<br>▪ 4x clock for Time Division Multiplex Implementation added<br>▪ Single Cycle Implementation on 1x clock now available for 1 to 32 bits.<br>▪ Time Division Multiplex Implementation on 4x clock now available for 9 to 64 bits.<br>▪ Asynchronous input signal reset is added.<br>▪ Synchronous input signal enable is added.<br>▪ Added new 'Advanced' page to the Configure dialog for the Implementation (Time Division Multiplex, Single Cycle) parameter | 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. |

| Version | Description of Changes | Reason for Changes / Impact |
|---|---|---|
| | Add validator to Resolution parameter. | The resolution of CRC is 1 to 64 bits. The validator was added to restrict input values. |
| | 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', 'SeedValueLower' | 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, 2009-2011. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

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

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

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A HALFICULAR 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.