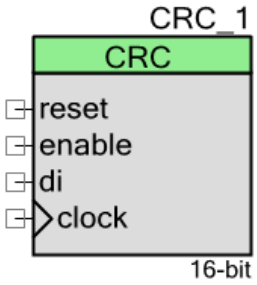
Cyclic Redundancy Check (CRC)

2.50

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 synchronous 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.

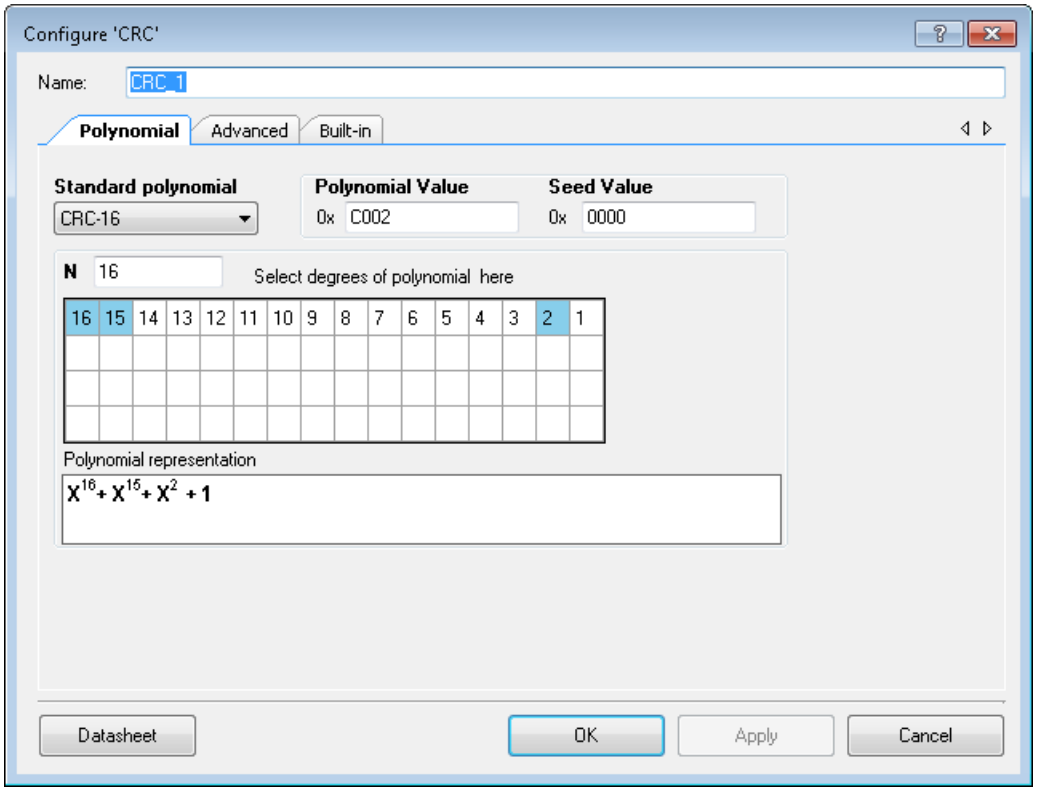
di – Input

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

Polynomial Name	Polynomial	Use
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
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 (the reverse of reciprocal representation is used in the Component). 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.

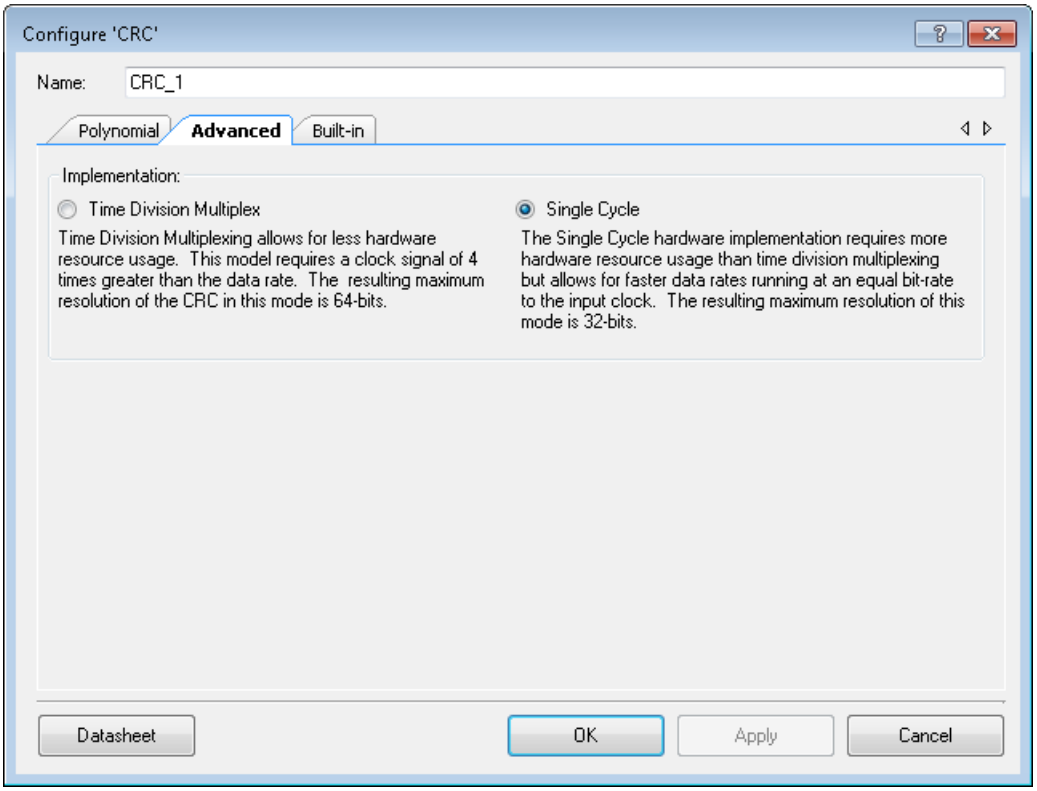
Use Polynomial Degrees

- 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.

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.”

Functions

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_Sleep()	Stops CRC computation and saves the CRC configuration.
CRC_Wakeup()	Restores the CRC configuration and starts CRC computation on rising edge of input clock.
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.

Function	Description
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.

void CRC_Start(void)

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

void CRC_Stop(void)

Description: Stops CRC computation.

void CRC_Sleep(void)

Description: Stops CRC computation and saves the CRC configuration.

void CRC_Wakeup(void)

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

void CRC_Init(void)

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

void CRC_Enable(void)

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

void CRC_SaveConfig(void)

Description: Saves the initial seed and polynomial registers.

void CRC_RestoreConfig(void)

Description: Restores the initial seed and polynomial registers.

void CRC_WriteSeed(uint8/16/32 seed)

Description: Writes the seed value.

Parameters: uint8/16/32 seed: Seed value

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

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

uint8/16/32 CRC_ReadCRC(void)

Description: Reads the CRC value.

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

uint32 CRC_ReadCRCUpper(void)

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

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

uint32 CRC_ReadCRCLower(void)

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

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

void CRC_WritePolynomial(uint8/16/32 polynomial)

Description: Writes the CRC polynomial value.

Parameters: uint8/16/32 polynomial: CRC polynomial

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

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

uint8/16/32 CRC_ReadPolynomial(void)

Description: Reads the CRC polynomial value.

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

uint32 CRC_ReadPolynomialUpper(void)

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

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

uint32 CRC_ReadPolynomialLower(void)

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

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

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. If reinitialization of the Component is required, then the CRC_Init() function can be called before the CRC_Start() or CRC_Enable() function.

Sample Firmware Source Code

PSoC Creator provides many code examples that include schematics and example code in the Find Code Example 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 Code Example” topic in the PSoC Creator Help for more information.

API Memory Usage

The Component memory usage varies significantly, depending on the compiler, device, number of APIs used and Component configuration. The following table provides the memory usage for all APIs available in the given Component configuration.

The measurements have been done with the associated compiler configured in Release mode with optimization set for Size. For a specific design the map file generated by the compiler can be analyzed to determine the memory usage.

Configuration	PSoC 3 (Keil_PK51)		PSoC 4 (GCC)		PSoC 5LP (GCC)		PSoC 6 (GCC)	
	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes
8-Bits Single Cycle	155	2	242	8	246	8	252	2
16-Bits Single Cycle	202	2	260	8	260	8	268	2
24-Bits Single Cycle	294	2	320	8	294	8	396	2
32-Bits Single Cycle	212	2	280	8	282	8	292	2
16-Bits Time Division	234	2	320	8	320	8	352	2
24-Bits Time Division	548	2	404	8	398	8	432	2
32-Bits Time Division	567	2	432	8	434	8	492	2
40-Bits Time Division	687	2	568	8	580	8	628	2
48-Bits Time Division	839	2	610	8	638	8	690	2
56-Bits Time Division	938	2	662	8	702	8	762	2
64-Bits Time Division	1045	2	710	8	718	8	830	2

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.

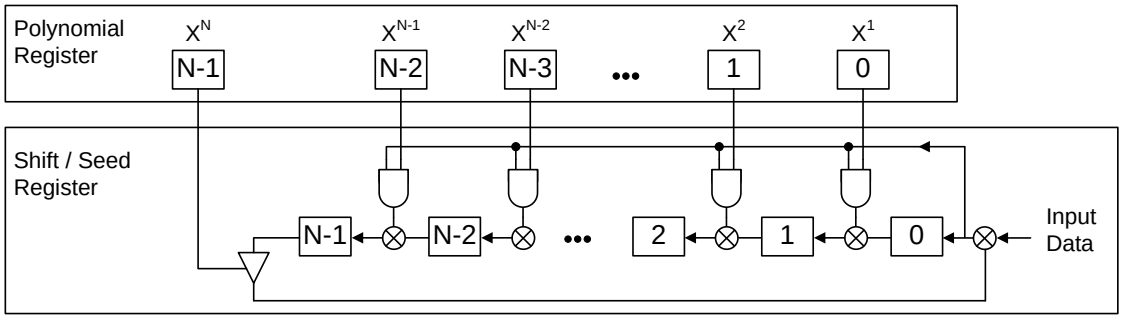
To specify the polynomial specification, write an (N + 1)-bit binary number corresponding to the full polynomial, with 1s for each term present. The CRC-CCITT polynomial would be 10001000000100001b. Then, drop the right-most bit (the X^0 term) to obtain the CRC polynomial

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

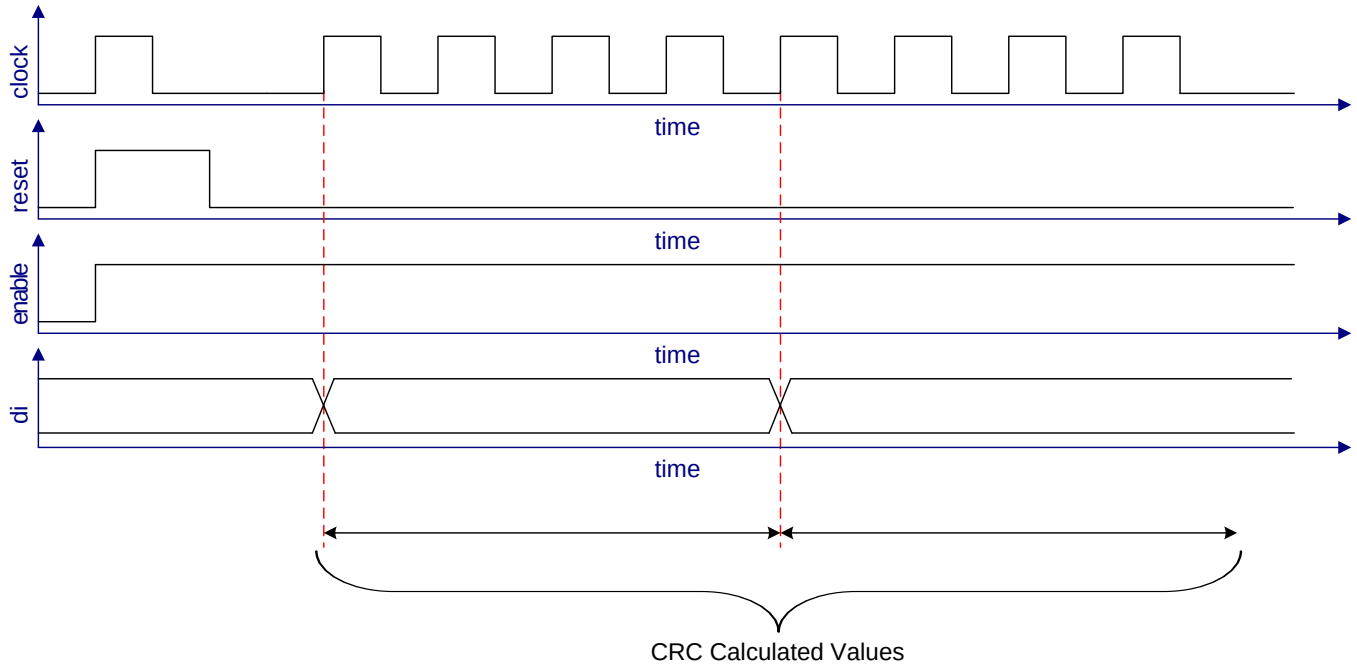
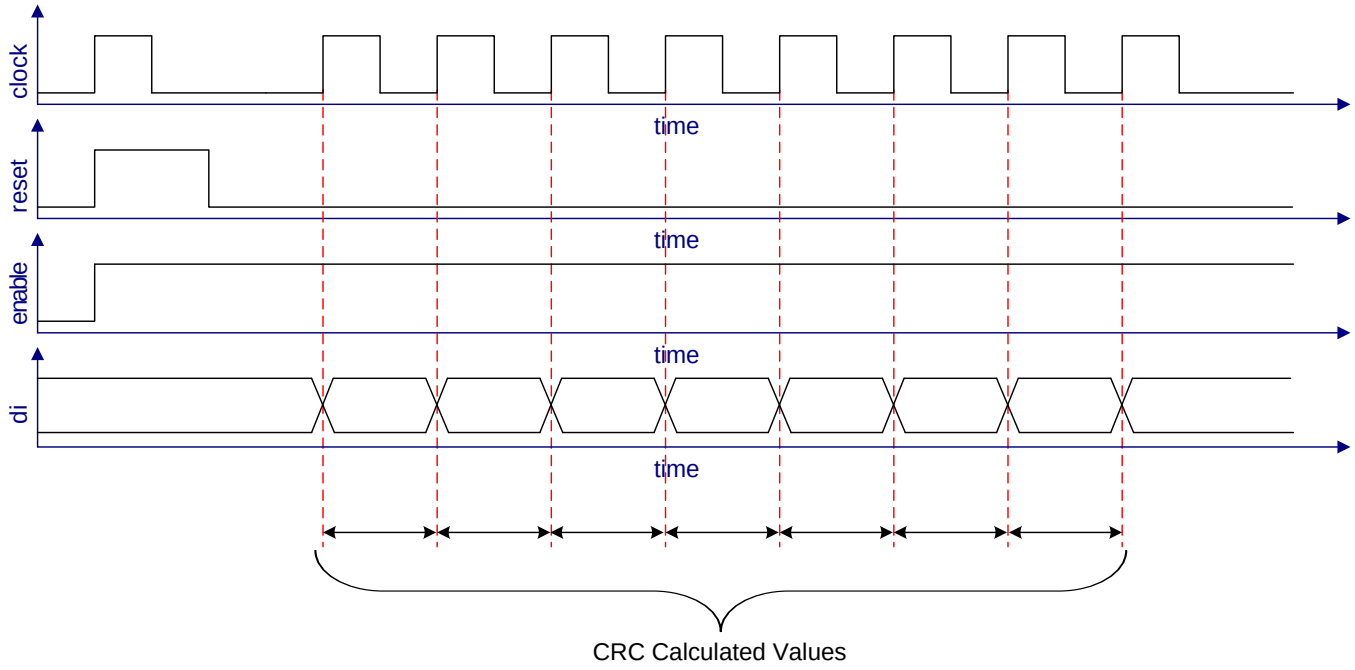


Figure 2. Single Cycle Implementation Mode



Resources

The CRC Component is placed throughout the UDB array. The Component utilizes the following resources.

Configuration	Resource Type					
	Datapath Cells	Macrocells	Status Cells	Control Cells	DMA Channels	Interrupts
8-Bits Single Cycle	1	3	–	1	–	–
16-Bits Single Cycle	2	3	–	1	–	–
24-Bits Single Cycle	3	3	–	1	–	–
32-Bits Single Cycle	4	3	–	1	–	–
16-Bits Time Division	1	9	–	2	–	–
24-Bits Time Division	2	10	–	2	–	–
32-Bits Time Division	2	9	–	2	–	–
40-Bits Time Division	3	10	–	2	–	–
48-Bits Time Division	3	9	–	2	–	–
56-Bits Time Division	4	10	–	2	–	–
64-Bits Time Division	4	9	–	2	–	–

MISRA Compliance

This section describes the MISRA-C:2004 compliance and deviations for the Component. There are two types of deviations defined:

- project deviations – deviations that are applicable for all PSoC Creator Components
- specific deviations – deviations that are applicable only for this Component

This section provides information on Component-specific deviations. Non PSoC 6 project deviations are described in the MISRA Compliance section of the *System Reference Guide* along with information on the MISRA compliance verification environment. For PSoC 6, refer to PSoC Creator Help > Building a PSoC Creator Project > Generated Files (PSoC 6) for information on MISRA compliance and deviations for files generated by PSoC Creator.

The CRC Component has the following specific deviations:

MISRA-C:2004 Rule	Rule Class (Required/ Advisory)	Rule Description	Description of Deviation(s)
14.3	R	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.	Deviation is caused by using a macro that has a group of statements contained within curly braces. The macro is CRC_EXECUTE_DFF_RESET. A MISRA compliant implementation would require the use of this macro to be followed by a semicolon. This could introduce failures in designs that used this macro directly.
19.4	R	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.	Deviation is caused by using a macro that has a group of statements contained within curly braces. The macro is CRC_EXECUTE_DFF_RESET. A MISRA compliant implementation would require the use of this macro to be followed by a semicolon. This could introduce failures in designs that used this macro directly.
19.7	A	A function should be used in preference to a function-like macro.	Deviation is caused by the macro CRC_IS_CRC_ENABLE. If removed this would introduce failures in designs that used this macro directly.

DC and AC Electrical Characteristics

Specifications are valid for –40 °C ≤ T_A ≤ 85 °C and T_J ≤ 100 °C, except where noted.
Specifications are valid for 1.71 V to 5.5 V, except where noted.

DC Characteristics

Parameter	Description	Min	Typ[1]	Max	Units
I_{DD}	Component current consumption				
	8-Bits Single Cycle	–	10	–	µA/MHz
	16-Bits Single Cycle	–	16	–	µA/MHz
	24-Bits Single Cycle	–	26	–	µA/MHz
	32-Bits Single Cycle	–	33	–	µA/MHz
	16-Bits Time Division	–	17	–	µA/MHz
	24-Bits Time Division	–	29	–	µA/MHz
	32-Bits Time Division	–	29	–	µA/MHz
	40-Bits Time Division	–	35	–	µA/MHz
	48-Bits Time Division	–	35	–	µA/MHz
	56-Bits Time Division	–	43	–	µA/MHz
	64-Bits Time Division	–	44	–	µA/MHz

AC Characteristics

Parameter	Description	Min	Typ	Max[2]	Units
f_{CLOCK}	Component clock frequency				
	8-Bits Single Cycle	–	–	41	MHz
	16-Bits Single Cycle	–	–	32	MHz
	24-Bits Single Cycle	–	–	30	MHz
	32-Bits Single Cycle	–	–	28	MHz
	16-Bits Time Division	–	–	34	MHz
	24-Bits Time Division	–	–	24	MHz

1. Device IO and clock distribution current not included. The values are at 25 °C.
2. The values provide a maximum safe operating frequency of the Component. The Component may run at higher clock frequencies, at which point you will need to validate the timing requirements with STA results.

Parameter	Description	Min	Typ	Max[2]	Units
	32-Bits Time Division	–	–	29	MHz
	40-Bits Time Division	–	–	24	MHz
	48-Bits Time Division	–	–	27	MHz
	56-Bits Time Division	–	–	23	MHz
	64-Bits Time Division	–	–	28	MHz

Component Changes

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

Version	Description of Changes	Reason for Changes / Impact
2.50.a	Updated MISRA section. Updated API Memory Usage data for PSoC 6.	
2.50	Added PSoC 6 device support	
2.40.a	Datasheet update	Clarified Polynomial representation in the Polynomial Tab Updated Resources section
2.40	Updated datasheet with memory usage for PSoC 4. Updated Defines for [25 – 32] bit Single Cycle Implementation Mode for PSoC 4.	
2.30	Added MISRA Compliance section.	The Component has specific deviations described.
	Added PSoC 4A support	
	Changed Timing Diagrams	
2.20	Added PSoC 5LP support	
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	

Version	Description of Changes	Reason for Changes / Impact
2.0	Added support for PSoC 3 ES3 silicon. Changes include: ▪ 4x clock for Time Division Multiplex Implementation added ▪ Single Cycle Implementation on 1x clock now available for 1 to 32 bits. ▪ Time Division Multiplex Implementation on 4x clock now available for 9 to 64 bits. ▪ Asynchronous input signal reset is added. ▪ Synchronous input signal enable is added. ▪ 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.
	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.