SYNOPSYS®

DesignWare® DW_apb_rtc

Databook

DW_apb_rtc – Product Code

2.08a
December 2020

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at https://www.synopsys.com/company/legal/trademarks-brands.html

All other product or company names may be trademarks of their respective owners.

Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
www.synopsys.com

Contents

Revision History

This section tracks the significant documentation changes that occur from release-to-release and during a release from version 2.01d onward.

Version	Date	Description
2.08a	December 2020	**Added:** ■ “Timing Exceptions” on page 79 ■ “BCM Library Components” on page 83 **Updated:** ■ Version number change to 2020.12a ■ “Performance” on page 80 ■ Parameter Descriptions, Register Descriptions, Signal Descriptions, and Internal Parameter Descriptions are auto-extracted with change bars from the RTL **Renamed:** ■ Clock Relationship to “Clocks and Resets” ■ Synchronizer to “Basic Core Module (BCM) Library” **Removed:** ■ Index chapter
2.07a	July 2018	**Updated:** ■ Version number change to 2018.07a ■ “Performance” on page 80 ■ Parameter Descriptions, Register Descriptions, Signal Descriptions, and Internal Parameter Descriptions are auto-extracted with change bars from the RTL **Removed:** ■ Chapter 2, “Building and Verifying a Component or Subsystem” and added the contents in the newly created user guide.

Version	Date	Description
2.06a	October 2016	■ Version number change to 2016.10a ■ Parameter Descriptions and Register Descriptions, auto-extracted from the RTL ■ Removed the “Running Leda on Generated Code with coreConsultant” section, and reference to Leda directory in Table 2-1 ■ Removed the “Running Leda on Generated Code with coreAssembler” section, and reference to Leda directory in Table 2-4 ■ Moved Internal Parameter Descriptions to Appendix ■ Added “Running VCS XPROP Analyzer” ■ Added an entry for the xprop directory in and Table 2-4. ■ Added “RTC Pre-scaler Counter” on page 18 ■ Added “APB Interface” on page 22
2.05a	June 2015	■ Added “Running SpyGlass® Lint and SpyGlass® CDC” ■ Added “Running SpyGlass on Generated Code with coreAssembler” ■ Signal Descriptions auto-extracted from the RTL ■ Added Internal Parameter Descriptions ■ Added Appendix A, “Basic Core Module (BCM) Library”
2.04a	June 2014	■ Version change for 2014.06a release ■ Added “Performance” section in the “Integration Considerations” chapter ■ Corrected Default Input/Output Delay in Signals chapter
2.03e	May 2013	■ Version change for 2013.05a release ■ Updated the template
2.03d	September 2012	Added the product code on the cover and in Table 1-1
2.03d	March 2012	Version change for 2012.03a release
2.03c	November 2011	Version change for 2011.11a release
2.03b	October 2011	Version change for 2011.10a release
2.03a	June 2011	■ Updated system diagram in Figure 1-1 ■ Enhanced “Related Documents” section in Preface
2.03a	March 2011	Corrected synchronous statement for rtc_rst_n signal
2.03a	September 2010	Corrected names of include files and vcs command used for simulation
2.02a	December 2009	Updated databook to new template for consistency with other IIP/VIP/PHY databooks
2.02a	May 2008	Removed references to QuickStarts, as they are no longer supported
2.02a	October 2008	Version change for 2008.10a release

Version	Date	Description
2.01e	June 2008	Version change for 2008.06a release
2.01d	January 2008	■ Updated for revised installation guide and consolidated release notes titles ■ Changed references of “Designware AMBA” to simply “DesignWare”
2.01d	June 2007	Version change for 2007.06a release

Preface

This databook provides information that you need to interface the DW_apb_rtc component to the Advanced Peripheral Bus (APB). This component conforms to the *AMBA Specification, Revision 2.0* from Arm®.

The information in this databook includes a functional description, signal and parameter descriptions, and a memory map. Also provided are an overview of the component testbench, a description of the tests that are run to verify the coreKit, and synthesis information for the coreKit.

Organization

The chapters of this databook are organized as follows:

- Chapter 1, "Product Overview" provides a system overview, a component block diagram, basic features, and an overview of the verification environment.
- Chapter 2, "Functional Description" describes the functional operation of the DW_apb_rtc.
- Chapter 3, "Parameter Descriptions" identifies the configurable parameters supported by the DW_apb_rtc.
- Chapter 4, "Signal Descriptions" provides a list and description of the DW_apb_rtc signals.
- Chapter 5, "Register Descriptions" describes the programmable registers of the DW_apb_rtc.
- Chapter 6, "Programming the DW_apb_rtc" provides information needed to program the configured DW_apb_rtc.
- Chapter 7, "Verification" provides information on verifying the configured DW_apb_rtc.
- Chapter 8, "Integration Considerations" includes information you need to integrate the configured DW_apb_rtc into your design.
- Appendix A, "Basic Core Module (BCM) Library" documents the synchronizer methods (blocks of synchronizer functionality), and list of BCM library components used in DW_apb_rtc.
- Appendix B, "Internal Parameter Descriptions" provides a list of internal parameter descriptions that might be indirectly referenced in expressions in the Signals chapter.
- Appendix C, "Glossary" provides a glossary of general terms.

Related Documentation

- Using DesignWare Library IP in coreAssembler – Contains information on getting started with using DesignWare SIP components for AMBA 2 and AMBA 3 AXI components within coreTools
- coreAssembler User Guide – Contains information on using coreAssembler

- coreConsultant User Guide – Contains information on using coreConsultant

To see a complete listing of documentation within the DesignWare Synthesizable Components for AMBA 2, refer to the *Guide to Documentation for DesignWare Synthesizable Components for AMBA 2 and AMBA 3 AXI (Documentation Overview)*.

Web Resources

- DesignWare IP product information: https://www.synopsys.com/designware-ip.html
- Your custom DesignWare IP page: https://www.synopsys.com/dw/mydesignware.php
- Documentation through SolvNetPlus: https://solvnetplus.synopsys.com (Synopsys password required)
- Synopsys Common Licensing (SCL): https://www.synopsys.com/keys

Customer Support

Synopsys provides the following various methods for contacting Customer Support:

- Prepare the following debug information, if applicable:
  - For environment set-up problems or failures with configuration, simulation, or synthesis that occur within coreConsultant or coreAssembler, select the following menu:

    File > Build Debug Tar-file

    Check all the boxes in the dialog box that apply to your issue. This option gathers all the Synopsys product data needed to begin debugging an issue and writes it to the <*core tool startup directory*>/debug.tar.gz file.
  - For simulation issues outside of coreConsultant or coreAssembler:
    - Create a waveforms file (such as VPD or VCD).
    - Identify the hierarchy path to the DesignWare instance.
    - Identify the timestamp of any signals or locations in the waveforms that are not understood.
- *For the fastest response*, enter a case through SolvNetPlus:

  a. https://solvnetplus.synopsys.com

  Note SolvNetPlus does not support Internet Explorer. Use a supported browser such as Microsoft Edge, Google Chrome, Mozilla Firefox, or Apple Safari.

  b. Click the **Cases** menu and then click **Create a New Case** (below the list of cases).

  c. Complete the mandatory fields that are marked with an asterisk and click **Save**.

  Ensure to include the following:
  - **Product L1:** DesignWare Library IP
  - **Product L2:** AMBA

  d. After creating the case, attach any debug files you created.

  For more information about general usage information, refer to the following article in SolvNetPlus:

https://solvnetplus.synopsys.com/s/article/SolvNetPlus-Usage-Help-Resources

- Or, send an e-mail message to support_center@synopsys.com (your email will be queued and then, on a first-come, first-served basis, manually routed to the correct support engineer):
  - Include the Product L1 and Product L2 names, and Version number in your e-mail so it can be routed correctly.
  - For simulation issues, include the timestamp of any signals or locations in waveforms that are not understood
  - Attach any debug files you created.
- Or, telephone your local support center:
  - North America:
    Call 1-800-245-8005 from 7 AM to 5:30 PM Pacific time, Monday through Friday.
  - All other countries:
    https://www.synopsys.com/support/global-support-centers.html

Product Code

Table 1-1 lists all the components associated with the product code for DesignWare APB Peripherals.

Table 1-1 DesignWare APB Peripherals – Product Code: 3771-0

Component Name	Description
DW_apb_gpio	General Purpose I/O pad control peripheral for the AMBA 2 APB bus
DW_apb_rap	Programmable controller for the remap and pause features of the DW_ahb interconnect
DW_apb_rtc	A configurable high range counter with an AMBA 2 APB slave interface
DW_apb_timers	Configurable system counters, controlled through an AMBA 2 APB interface
DW_apb_wdt	A programmable watchdog timer peripheral for the AMBA 2 APB bus

1 Product Overview

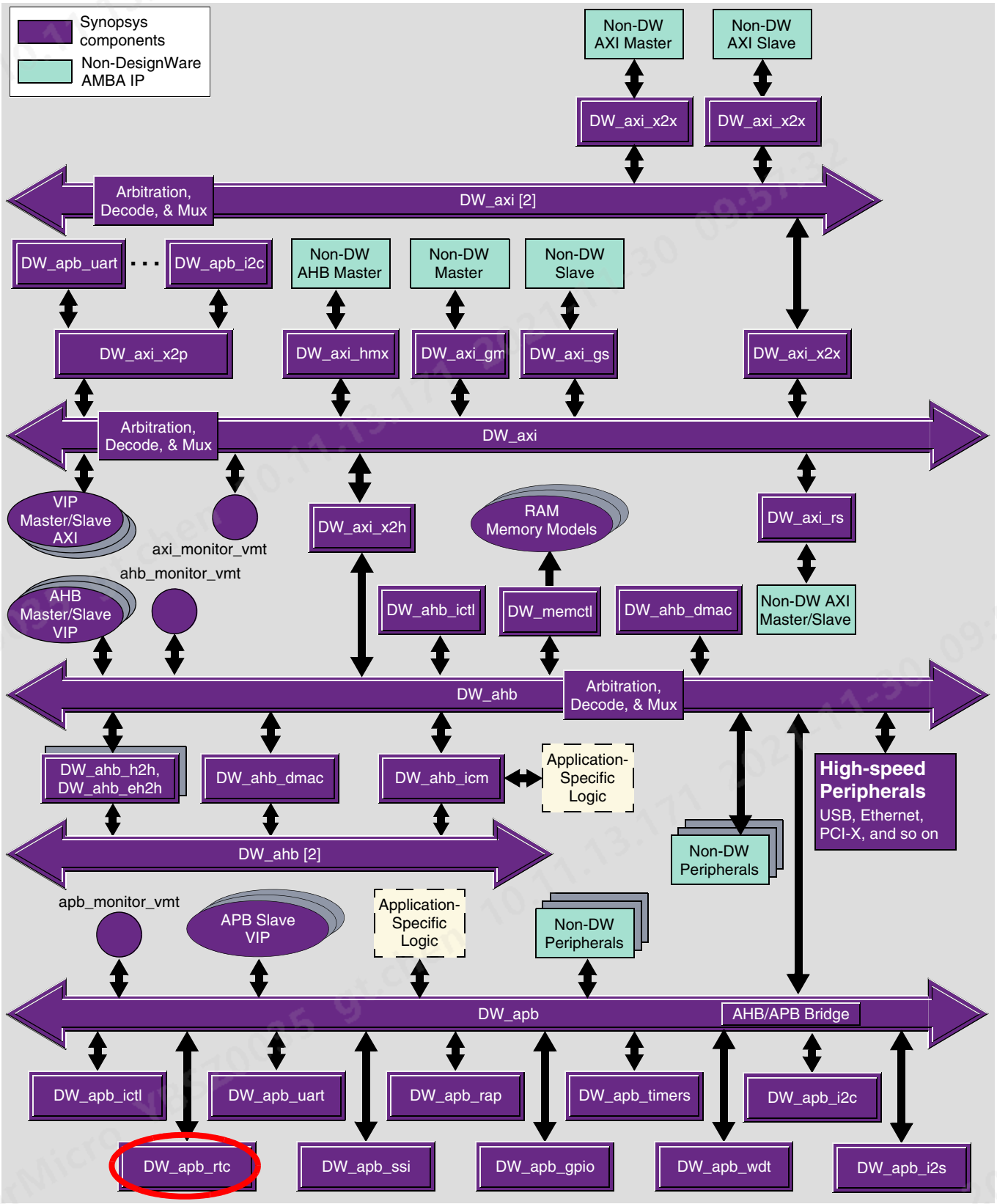
The DW_apb_rtc is a programmable Real Time Clock (RTC) peripheral. This component is an AMBA 2.0-compliant Advanced Peripheral Bus (APB) slave device and is part of the family of DesignWare Synthesizable Components.

1.1 DesignWare System Overview

The Synopsys DesignWare Synthesizable Components environment is a parameterizable bus system containing AMBA version 2.0-compliant AHB (Advanced High-performance Bus) and APB (Advanced Peripheral Bus) components, and AMBA version 3.0-compliant AXI (Advanced eXtensible Interface) components.

Figure 1-1 illustrates one example of this environment, including the AXI bus, the AHB bus, and the APB bus. Included in this subsystem are synthesizable IP for AXI/AHB/APB peripherals, bus bridges, and an AXI interconnect and AHB bus fabric. Also included are verification IP for AXI/AHB/APB master/slave models and bus monitors. To access the product page and documentation for AMBA components, see the DesignWare IP Solutions for AMBA Interconnect page. (SolvNetPlus ID required)

Figure 1-1 Example of DW_apb_rtc in a Complete System



You can connect, configure, synthesize, and verify the DW_apb_rtc within a DesignWare subsystem using coreAssembler, documentation for which is available on the web in the *coreAssembler User Guide*.

If you want to configure, synthesize, and verify a single component such as the DW_apb_rtc component, you might prefer to use coreConsultant, documentation for which is available in the *coreConsultant User Guide*.

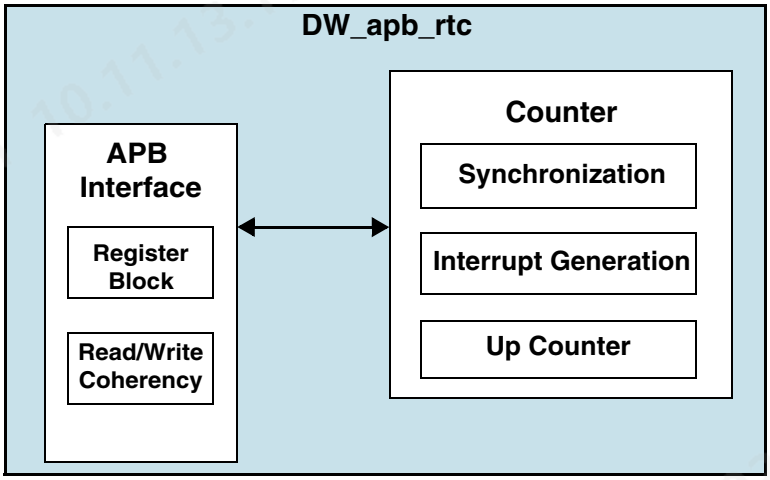
1.2 General Product Description

The Synopsys DW_apb_rtc is a component of the DesignWare Advanced Peripheral Bus (DW_apb) and conforms to the *AMBA Specification, Revision 2.0* from Arm®.

1.2.1 DW_apb_rtc Block Diagram

Figure 1-2 shows a block diagram of the DW_apb_rtc.

Figure 1-2 DW_apb_rtc Block Diagram



1.3 Features

DW_apb_rtc has the following features:

- APB interface supports APB2, APB3, and APB4.
- APB slave interface with read/write coherency for registers
- Incrementing counter and comparator for interrupt generation
- Free-running pclk
- Configurable option to include the prescalar counter.
- User-defined parameters:
  - APB data bus width
  - Counter width
  - Clock relationship between bus clock and counter clock

- Interrupt polarity level
- Interrupt clock domain location
- Counter enable mode
- Counter wrap mode

Some uses of the DW_apb_rtc are:

- Real-time clock – used with software for keeping track of time
- Long-term, exact chronometer – When clocked with a 1 Hz clock, it can keep track of time from now up to 136 years in the future
- Alarm function – generates an interrupt after a programmed number of cycles
- Long-time, base counter – clocked with a very slow clock signal

Source code for this component is available on a per-project basis as a DesignWare Core. Please contact your local sales office for the details.

1.4 Standards Compliance

The DW_apb_rtc component conforms to the *AMBA Specification, Revision 2.0* from Arm®. Readers are assumed to be familiar with this specification.

1.5 Verification Environment Overview

The DW_apb_rtc includes an extensive verification environment, which sets up and invokes your selected simulation tool to execute tests that verify the functionality of the configured component. You can then analyze the results of the simulation.

The "Verification" on page 63 section discusses the specific procedures for verifying the DW_apb_rtc.

1.6 Licenses

Before you begin using the DW_apb_rtc, you must have a valid license. For more information, refer to "Licenses"in the *DesignWare Synthesizable Components for AMBA 2/AMBA 3 AXI Installation Guide*.

1.7 Where To Go From Here

At this point, you may want to get started working with the DW_apb_rtc component within a subsystem or by itself. Synopsys provides several tools within its coreTools suite of products for the purposes of configuration, synthesis, and verification of single or multiple synthesizable IP components—coreConsultant and coreAssembler. For information on the different coreTools, refer to *Guide to coreTools Documentation*.

For more information about configuring, synthesizing, and verifying just your DW_apb_rtc component, refer to "Overview of the coreConsultant Configuration and Integration Process" in *DesignWare Synthesizable Components for AMBA 2 User Guide*.

For more information about implementing your DW_apb_rtc component within a DesignWare subsystem using coreAssembler, refer to "Overview of the coreAssembler Configuration and Integration Process" in *DesignWare Synthesizable Components for AMBA 2 User Guide*.

2 Functional Description

This chapter describes the functional operation of the DW_apb_rtc.

2.1 Up Counter

The DW_apb_rtc has a programmable, binary counter for which the user specifies the width (RTC_CNT_WIDTH). The counter increments on successive positive edges of the input counter clock, rtc_clk. When the Counter Load Register (RTC_CLR) is programmed, the counter is loaded with a start value that allows the counter to increment. When the counter reaches its maximum value (all bits are high), it wraps to 0 and then continues incrementing.

Depending on the user-configured relationship between the counter clock and the APB bus clock, the value loaded into the counter may need to be transferred across clock domains. Once the value is transferred, it is then loaded into the counter. A new value qualifier signal is transferred along with the load value to generate the load enable for the counter.

The sequences of events for the counter are:

1. User programs a new load value by writing to RTC_CLR.
2. RTC_CLR is transferred into the counter clock domain.
3. Transferred value is loaded into counter.
4. Counter increments from the loaded value on the positive edge of the counter clock.

When configuring the DW_apb_rtc, a counter enable mode (RTC_EN_MODE = 1) can be set to require a counter enable bit (rtc_en) in the control register (RTC_CCR). Without this mode, the counter counts freely. The rtc_en bit is generated as an output so that a clock generator can use it for a free-running implementation of pclk that transfers values across clock domains.

When configuring the DW_apb_rtc, a counter wrap mode (RTC_WRAP_MODE = 1) can be set to require a counter wrap enable bit (rtc_wen) in the control register (RTC_CCR). This bit enables the counter to wrap when a match occurs, instead of waiting until the maximum count is reached. The counter wraps to 0 if the RTC_WRAP_2_ZERO = 1; otherwise, it wraps to the last loaded value.

An APB peripheral needs only a pclk when addressed for a read and a write. The interface does not require a pclk to maintain the registers. In the case of the DW_apb_rtc, it requires a clock to set its internal interrupt in the pclk domain; this is referred to as a free-running pclk. It is synchronous to the pclk clock, but is available even when the block is disabled.

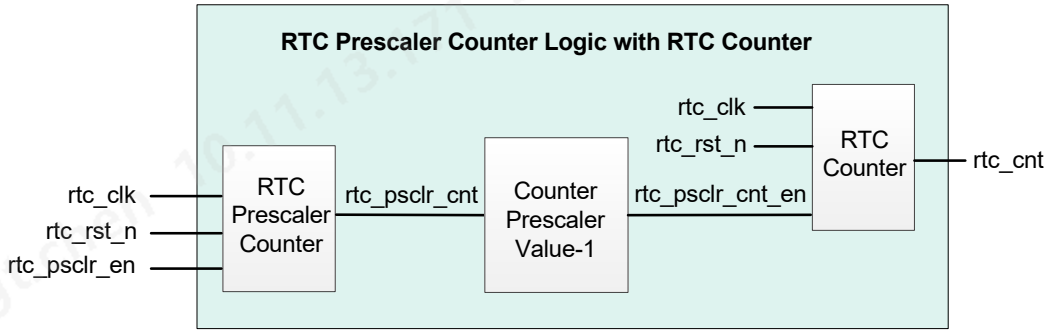
The counter (RTC_CCVR) value can be read at any time and is always the latest safe counter value, regardless of the relationship between rtc_clk and pclk. When there is an asynchronous relationship, then synchronization logic is included in order to prevent propagation of metastable values when reading the counter value.

2.2 RTC Pre-scaler Counter

The DW_apb_rtc supports the RTC Pre-scaler Counter implementation that enables you to update the RTC counter at the rate lower than the RTC clock (rtc_clk/rtc_clk_en/pclk) rate. The RTC pre-scaler counter logic is a configurable option and therefore, maintains backward compatibility such that DW_apb_rtc can run without the RTC pre-scaler counter logic. You can also enable/disable the RTC pre-scaler counter through the software using the rtc_psclr_en bit of the RTC_CCR register.

Figure 2-1 shows the logical implementation of the DW_apb_rtc pre-scaler counter.

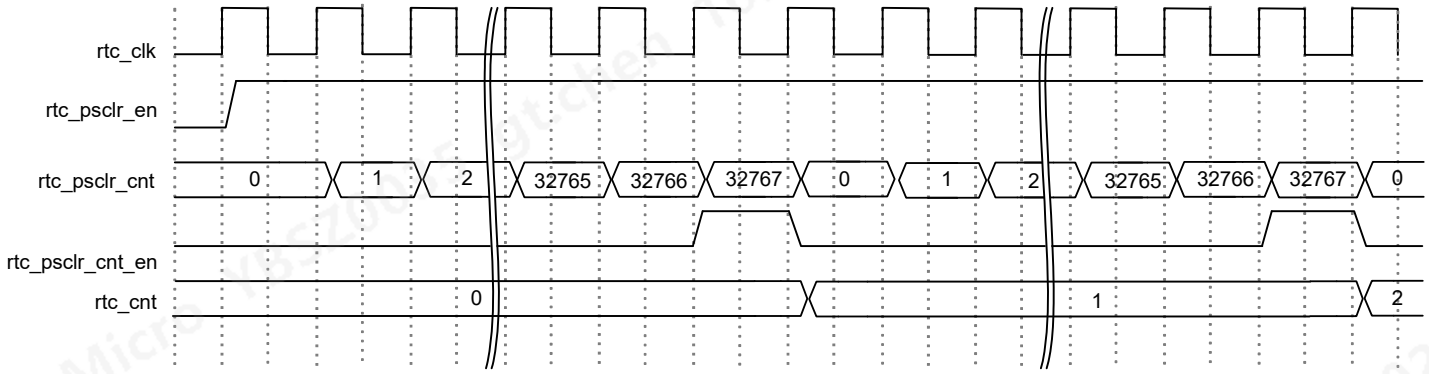
Figure 2-1 RTC Pre-scaler Counter with an RTC Counter



If the RTC_CCR.RTC_PRESCLR_EN bit is set, then RTC pre-scaler counter runs on the RTC clock and provides the RTC pre-scaler count. The RTC pre-scaler counter generates the rtc_psclr_cnt_en pulse when the counter counts ((RTC_CPSR[RTC_PRESCLR_WIDTH-1:0])-1) that is, (Counter Pre-scaler Value-1). The rtc_psclr_cnt_en pulse that is used as the clock enable to the RTC counter, enables the RTC counter to be updated at a rate less than the RTC clock rate.

For instance, consider that DW_apb_rtc is running using a clock of 37.368 KHz. You can generate the 1 Hz clock from the RTC counter by enabling the RTC Pre-scaler Counter and by loading the RTC counter pre-scaler value as 37368, as shown in Figure 2-2.

Figure 2-2 Timing Diagram for RTC Pre-scaler Counter with the RTC Counter



Consider the following points while configuring the RTC pre-scaler counter with the RTC counter:

- When rtc_psclr_en bit of the RTC_CCR register is disabled:
  - The RTC pre-scaler counter is reset to zero. This helps in aligning the RTC pre-scaler counter with the RTC counter, when the rtc_psclr_en bit of the RTC_CCR register is set again.
  - RTC counter runs in normal mode with rtc_clk without the RTC pre-scaler Counter logic.
- The load option is not provided for the RTC pre-scaler counter.
- When appropriate, both rtc_en and rtc_psclr_en bits must be set high in a single write. This ensures that both the RTC counter and the RTC pre-scaler counter are in sync.
- The RTC_CPSR register must be programmed with appropriate value before enabling the rtc_psclr_en bit.
- Re-programming the RTC_CPSR register content in the middle of a normal RTC pre-scaler operation (that is, when the rtc_psclr_en bit is being enabled) is not recommended as this may lead to the following scenarios:
  - If the programmed RTC_CPSR register value is less than or equal to the current RTC pre-scaler counter value, the RTC pre-scaler Counter wraps immediately to zero and the RTC counter is incremented.
  - If the programmed RTC_CPSR register value is greater than the current RTC pre-scaler counter value, the RTC pre-scaler counter increments upto the RTC pre-scaler value and then wraps to zero and in turn incrementing the RTC Counter (Normal RTC Counter and RTC pre-scaler counter behavior as defined previously).

2.3 Clocks and Resets

Clocks and Reset information is described int he following sub-sections:

- "Clock Relationship" on page 19
- "Reset Condition" on page 20

2.3.1 Clock Relationship

The APB bus clock and the counter clock could be related in any of the following ways:

- Identical
- Synchronous
- Asynchronous

Regardless of the clock relationship, circuitry is required so that the internal counter can be loaded with a new value. The loaded value needs to be transferred to the counter clock domain if the clocks are not identical. Likewise, when the counter value is read, it must be transferred to the APB bus clock domain.

The output from the counter is transferred to the APB Bus clock domain each time the counter clock changes. The value read from the counter, RTC_CCVR, is the transferred value that was in the counter. The counter may increment while the count is read, but the value that is read back is coherent. By re-timing the clock signal instead of the counter value and looking for a rising edge detect of the re-timed clock signal, the counter value is safely transferred across domains.

Note When the clocks are asynchronous, there is a constraint on the ratio of the APB Bus clock and the counter clock. The frequency of the APB Bus clock must be greater than three times the frequency of the counter clock.

2.3.2 Reset Condition

After a reset, all counters and configuration registers return to their default states, such as all zeros. Interrupt generation is disabled at reset. If the user has not disabled the counter, the counter increments as soon as the reset is removed.

There may be two clock domains: one for the programming registers, and the other for clocking the internal counter. Although applied asynchronously, it is the user's responsibility to synchronously remove the reset for each clock-domain. The resets must last for a minimum of one cycle per clock domain. After a reset and before the interrupt enable is set prior to using the DW_apb_rtc, the user should program the counter load value, the match register, and the counter enable bit (if configured).

2.4 Match Register and Interrupt Generation

A match register, RTC_CMR, can be programmed and is compared to the internal counter. An interrupt is generated when the match register and the internal counter are equal, but only if interrupt generation is enabled (RTC_CCR, bit 0). This interrupt can be masked if the RTC_CCR register bit 1 is set to 1, which gives the user control of sending interrupts externally. The match register can be read any time.

The polarity of the generated interrupt is a user-specified feature. The interrupt is kept active until it is cleared by an end-of-interrupt clear read access (RTC_EOI). The interrupt status bit is not polarity sensitive. The interrupt is active when the status is read as 1; otherwise, it is inactive. Programming the interrupt mask bit (rtc_mask) within the control register (RTC_CCR) masks the interrupt. The interrupt status can be read at any time. Even when the interrupt is masked, the raw interrupt status can be read.

Note If the RTC_RSTAT register indicates that a pending interrupt exists because a masked interrupt has been generated (rtc_mask bit of RTC_CCR is set to 1), then caution should be taken when programming RTC_CCR. That is, if both the rtc_mask and rtc_en bits of the RTC_CCR register are set to 0, then the interrupt asserts for one clock period. To avoid this scenario, you can use the following two-step process:

1. Program the rtc_en bit of the RTC_CCR register to 0, which clears the interrupt.
2. Then program the RTC_CCR register again to set the rtc_mask bit to 0, which unmasks any future interrupts.

Ideally, interrupts are generated in the pclk domain in order to directly service them by a master, but this is not always possible. There are two scenarios to consider for interrupt generation, described in Figure 2-3.

- **Interrupt generation in the pclk domain.** When an interrupt is generated in the pclk domain, a free-running pclk is required to transfer the counter value each time it changes. When the transferred value matches the match register, the rising edge of the match (red_match in Figure 2-4) causes an interrupt to be generated. The interrupt is held until it is cleared (rtc_eoi_en) or until the counter is disabled (RTC_CCR, bit 2). The clearing of the interrupt is by the end-of-interrupt read access (RTC_EOI, bit 0). Refer to diagram A in Figure 2-3. The timing diagram in Figure 2-4 also shows an example of this type of interrupt generation.

- **Interrupt generation in the rtc_clk domain.** When an interrupt is generated in the rtc_clk domain, only the rtc_clk must be present to detect the interrupt. In this case, there is no pclk available when the rtc_clk is running, so the match register is transferred to the rtc_clk domain. When the counter and the transferred match register are equal, an interrupt is generated. Refer to diagram B in Figure 2-3. By having the interrupt in the rtc_clk domain, clearing the pclk domain is an asynchronous clear and synchronous removal in relation to the rtc_clk. The timing diagram in Figure 2-5 demonstrates this type of interrupt generation.

Figure 2-3 Interrupt Generation

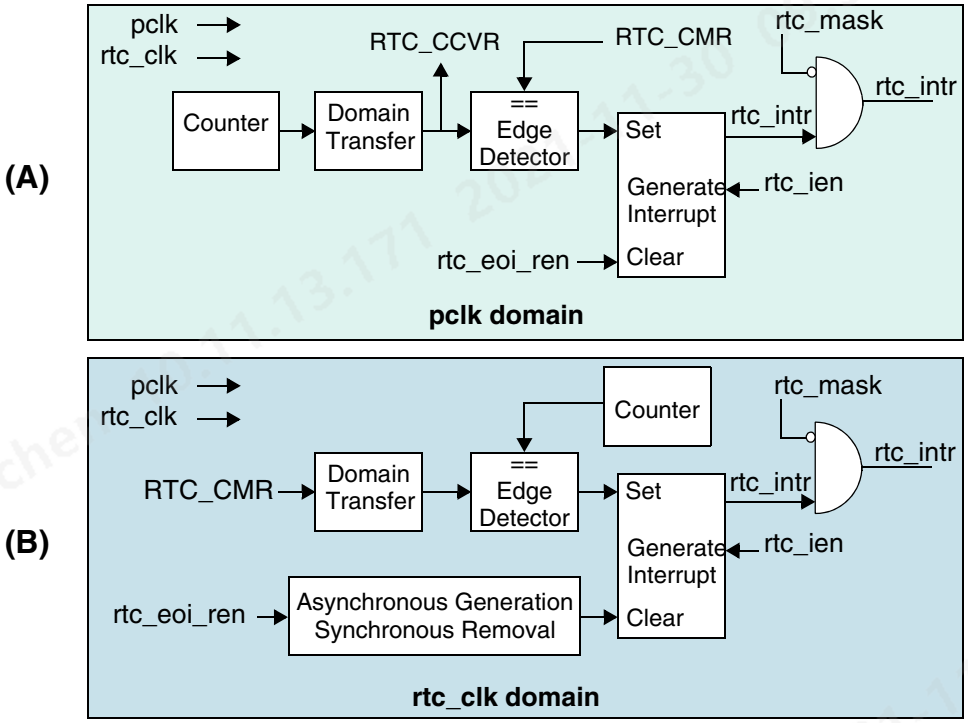


Figure 2-4 Timing for Interrupt Generation in pclk Domain

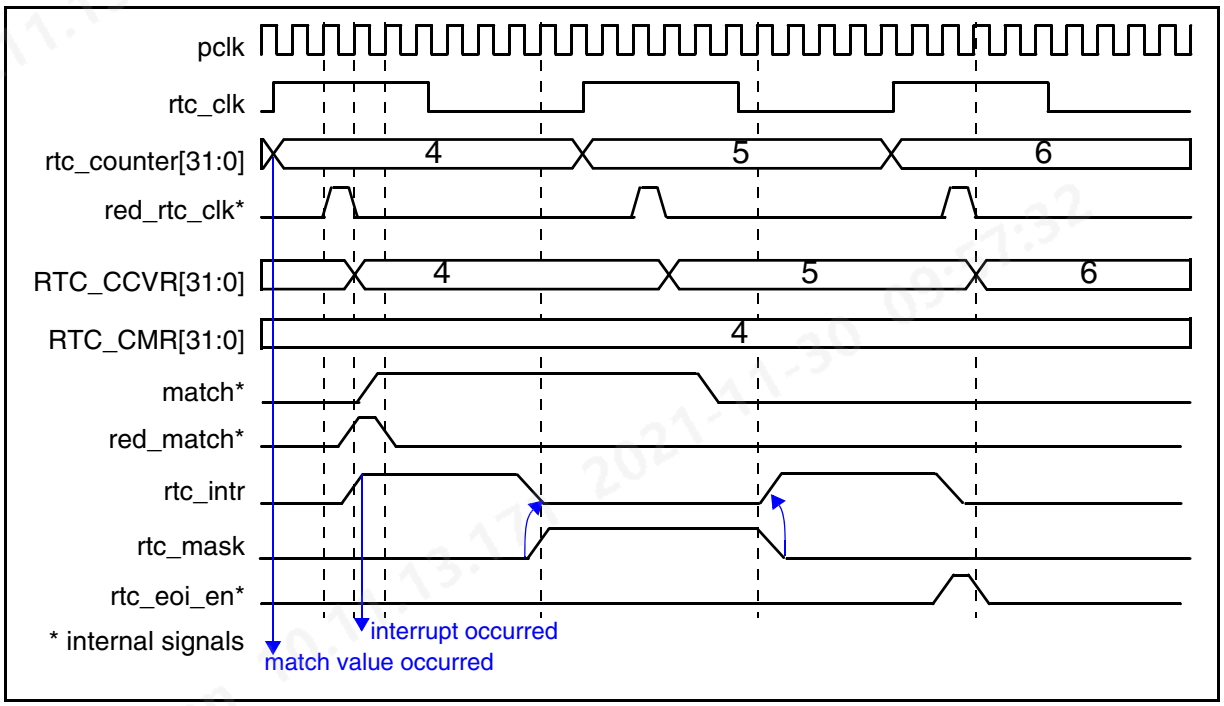
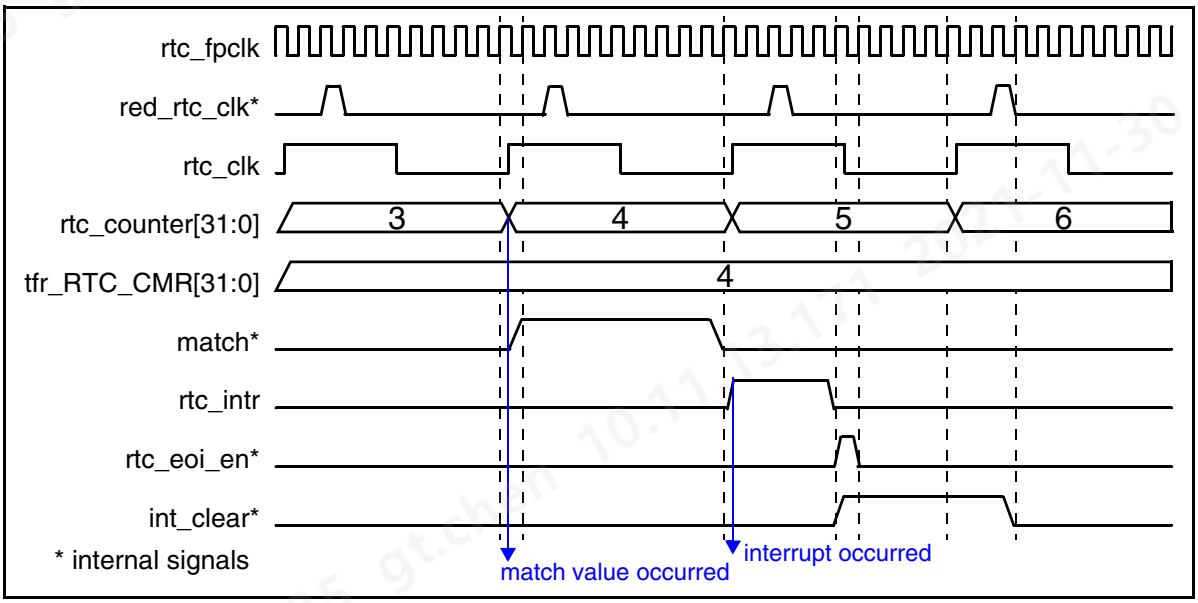


Figure 2-5 Timing for Interrupt Generation in rtc_clk Domain

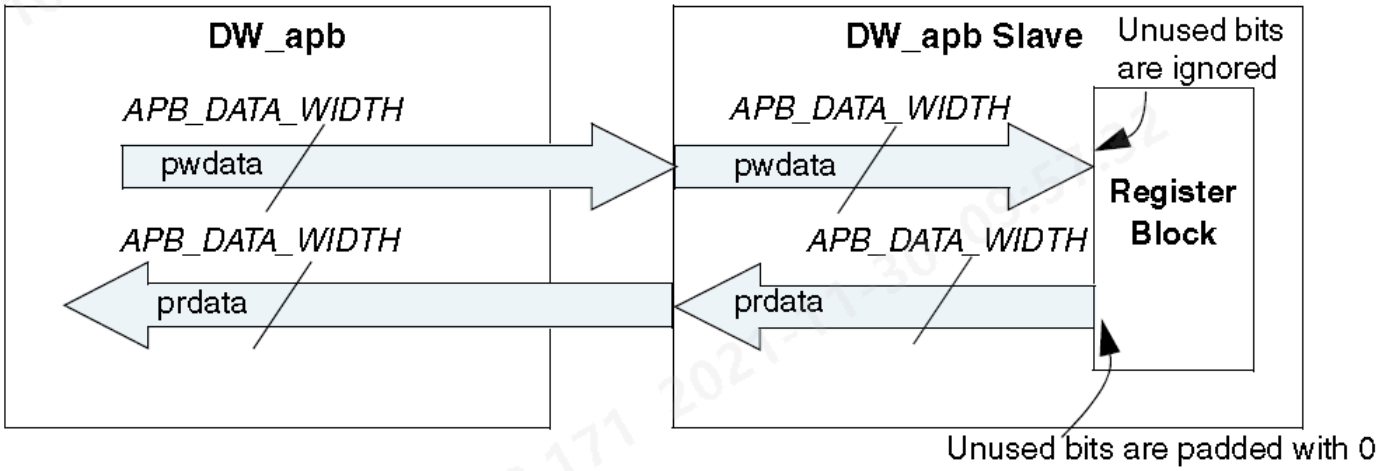


2.5 APB Interface

The host processor accesses internal registers on the DW_apb_rtc peripheral through the AMBA APB 2.0/3.0/4.0 interface. This peripheral supports APB data bus widths of 8, 16, or 32 bits, which is set with the APB_DATA_WIDTH parameter.

Figure 2-6 shows the read/write buses between the DW_apb and the APB slave.

Figure 2-6 Read/Write Buses Between the DW_apb and an APB Slave



The data, control and status registers within the DW_apb_rtc are byte-addressable. The maximum width of the control or status register (except for RTC Component Version register) in the DW_apb_rtc is 8 bits. Therefore, if the APB data bus is 8, 16, or 32 bits wide, all read and write operations to the DW_apb_rtc control and status registers require only one APB access.

The RTC_CCVR, RTC_CMR, and RTC_CLR register widths depend on the RTC_CNT_WIDTH parameter, which can vary from 8 to 32. Depending on the width of timer and width of APB data bus (APB_DATA_WIDTH), the APB interface may need to perform single or multiple accesses to the previously mentioned registers.

"Integration Considerations" on page 69 provides more information about reading to and writing from the APB interface.

The APB 3.0 and APB 4.0 register accesses to the DW_apb_rtc peripheral are discussed in the following sections:

- "APB 3.0 Support" on page 23
- "APB 4.0 Support" on page 24

2.5.1 APB 3.0 Support

The DW_apb_rtc register interface is compliant with the AMBA APB 2.0, APB 3.0 and APB 4.0 specifications. The SLAVE_INTERFACE_TYPE parameter is used to select the APB interface type of the register interface. To comply with the AMBA APB 3.0 specification, DW_apb_rtc supports the following signals:

- PREADY – This signal is always set to its default value that is high for all APB processes.

Note DW_apb_rtc does not use the PREADY signal and it used only for interface consistency.

- PSLVERR – This signal issues an error when protected registers are accessed without relevant authorization levels. The PSLVERR signal is enabled when the SLVERR_RESP_EN parameter is set to 1, so that DW_apb_rtc provides any slave error response from register interface. For more information on this signal, see "APB 4.0 Support" on page 24.

2.5.2 APB 4.0 Support

The DW_apb_rtc register interface is compliant with the AMBA APB 2.0, APB 3.0 and APB 4.0 specifications. To comply with the AMBA APB 4.0 specification, DW_apb_rtc supports the following signals:

- PSTRB – This signal specifies the APB4 write strobe. In a write transaction, the PSTRB signal indicates validity of PWDATA bytes. DW_apb_rtc selectively writes to the bytes of the addressed register whose corresponding bit in the PSTRB signal is high. Bytes strobed low by the corresponding PSTRB bits are not modified.

  When the width of the APB data bus is greater than the register width, the register content is written or read in a single APB access. But, when the width of the APB data bus is less than the register width, the register must be written multiple times to update all the bytes or required bytes, using strobes. The register is written only after one of the following conditions are met as listed in Table 2-1.

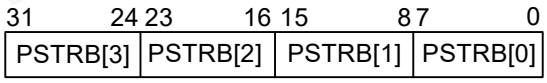
Table 2-1 Register Writer Condition

APB_DATA_WIDTH	REGISTER WIDTH	INTERNAL WRITE CONDITION
8	> 0 AND <= 8	PWRITE AND PSTRB[0]
8	> 8 AND <= 16	PWRITE AND PSTRB[1]
8	> 16 AND <= 24	PWRITE AND PSTRB[2]
8	> 24 AND <= 32	PWRITE AND PSTRB[3]
16	> 0 AND <= 16	PWRITE AND (PSTRB[0] OR PSTRB[1])
16	> 16 AND <= 32	PWRITE AND (PSTRB[0] OR PSTRB[1])
32	> 0 AND <= 32	PWRITE AND (PSTRB[0] OR PSTRB[1] OR PSTRB[2] OR PSTRB[3])

In this table, REGISTER WIDTH specifies the width of the internal register (except the reserved field). The incoming strobe bits for a read transaction is always be zero as per protocol.

Figure 2-7 shows the byte lane mapping of the PSTRB signal.

Figure 2-7 Byte Lane Mapping of the PSTRB Signal



- PPROT – This signal supports the protection feature of the APB4 protocol. The APB4 protection feature is supported only on the RTC_CMR and RTC_CLR registers. The protection level register (RTC_PROT_LEVEL) defines the APB4 protection level, that is the protected registers (RTC_CMR and RTC_CLR) are updated only if the PPROT privilege is more than the protection privilege

programmed in the protection level register (see Table 2-2). Otherwise, PSLVERR is asserted and the protected register is not updated, provided that PSLVERR_RESP_EN is set as high. If the PSLVERR_RESP_EN is low, then protection feature and PSLVERR generation logic is not implemented.

Table 2-2 PPROT Level, Protection Level Programmed in RTC_PROT_LEVEL, and Slave Error Response

PPROT			RTC_PROT_LEVEL			PSLVERR
[2]	[1]	[0]	[2]	[1]	[0]	
X	X	0	X	X	1	HIGH
X	1	X	X	0	X	HIGH
0	X	X	1	X	X	HIGH

2.6 Design for Test

The rtc_clk is re-timed using D-type flip-flops when there is an asynchronous relationship between the pclk and the rtc_clk. The rtc_clk is the D input into the D-type flip-flops, and pclk is the clock input. During scan mode, the pclk and the rtc_clk are generated from the same source. Because of this, there would be a timing violation with these D-type flip-flops, as the D-input would be the same as the clock input and would violate setup-and-hold rules for flip-flops. Therefore, the user must connect scan_mode to the chip-level scan_mode. During scan mode (scan_mode = 1), the D-input is selected as the output of another register, rather than the rtc_clk. It could even have the output of the flip-flop connected to the input, which leaves the register testable and subsequent downstream points controllable.

3 Parameter Descriptions

This chapter details all the configuration parameters. **You can use the coreConsultant GUI configuration reports to determine the actual configured state of the controller.** Some expressions might refer to TCL functions or procedures (sometimes identified as **<functionof>)** that coreConsultant uses to make calculations. The exact formula used by these TCL functions is not provided in this chapter. However, when you configure the controller in coreConsultant, all TCL functions and parameters are evaluated completely; and the resulting values are displayed where appropriate in the coreConsultant GUI reports.

The parameter descriptions in this chapter include the **Enabled:** attribute which indicates the values required to be set on other parameters before you can change the value of this parameter.

These tables define all of the configuration options for this component.

3.1 Top Level Parameters

Table 3-1 Top Level Parameters

Label	Description
APB Slave Configuration	
APB data bus width	Width of APB Data Bus to which this peripheral is attached. **Values:** 8, 16, 32 **Default Value:** 32 **Enabled:** Always **Parameter Name:** APB_DATA_WIDTH
Register Interface Type	Selects the Register Interface type as APB2, APB3 or APB4. By default, DW_apb_rtc supports APB2 interface. **Values:** ■ APB2 (0) ■ APB3 (1) ■ APB4 (2) **Default Value:** APB2 **Enabled:** Always **Parameter Name:** SLAVE_INTERFACE_TYPE
Slave Error Response Enable	Enables Slave Error response signaling. DW_apb_rtc will refrain from signaling an error response if this parameter is disabled. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** SLAVE_INTERFACE_TYPE>1 **Parameter Name:** SLVERR_RESP_EN
RTC Protection Level	Reset Value of RTC_PROT_LEVEL register. A high on any bit of RTC protection level requires a high on the corresponding pprot input bit to gain access to the load-count registers. Else, SLVERR response is triggered. A zero on the protection bit will provide access to the register if other protection levels are satisfied. **Values:** 0x0, ..., 0x7 **Default Value:** 0x0 **Enabled:** SLAVE_INTERFACE_TYPE>1 && SLVERR_RESP_EN==1 **Parameter Name:** PROT_LEVEL_RST

Table 3-1 Top Level Parameters (Continued)

Label	Description
Hard-Code Protection Level?	Checking this parameter makes RTC_PROT_LEVEL a read-only register. The register can be programmed at run-time by a user if this hard-code option is turned off. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** SLAVE_INTERFACE_TYPE>1 && SLVERR_RESP_EN==1 **Parameter Name:** HC_PROT_LEVEL
System Configuration	
Include enable bit (rtc_en) in control register (RTC_CCR) and on the interface?	Includes RTC enable bit in the control register, which can be programmed and the value reflected on the interface signal rtc_en. This signal indicates that the RTC is enabled and requires a free-running pclk, and can therefore be used by a clock generator. **Note**: If RTC_EN_MODE is True, the RTC Counter enable/disable depends on the RTC_CCR register. If presetn is asserted, the RTC_CCR register is reset and the Counter is disabled. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** Always **Parameter Name:** RTC_EN_MODE
Free running pclk?	Describes whether pclk is a free-running clock and always available to the DW_apb_rtc for re-timing the counter clock into the pclk domain. When pclk is not a free-running clock, then a free-running implementation must be connected to rtc_fpclk. This is used to transfer values across clock domains. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** Always **Parameter Name:** RTC_FREE_PCLK
RTC Counter Configuration	
RTC counter width	Size of the internal counter. **Values:** 8, ..., 32 **Default Value:** 32 **Enabled:** Always **Parameter Name:** RTC_CNT_WIDTH

Table 3-1 Top Level Parameters (Continued)

Label	Description
External RTC clock enable?	Includes the rtc_clk_en signal on the interface (rtc_clk is not included). Along with pclk, this signal allows the RTC to be clocked. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** Always **Parameter Name:** RTC_CLK_EN
RTC clock type	Describes the relationship between pclk and rtc_clk. **Values:** ■ Identical (0) ■ Synchronous (3) ■ Asynchronous (1) **Default Value:** [<functionof> RTC_CLK_EN SYNC ASYNC] **Enabled:** RTC_CLK_EN==0 **Parameter Name:** RTC_CLK_TYPE
Include counter wrap enable bit in control register (RTC_CCR)?	Includes the counter wrap enable bit in the control register (RTC_CCR), which can be programmed to enable the counter to wrap after it has reached the match value, instead of only wrapping at the maximum count. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** Always **Parameter Name:** RTC_WRAP_MODE
Wrap counter to zero after a match?	When this option is enabled, the counter wraps to 0 when the counter reaches the match value and the counter wrap enable is set; otherwise, it wraps to the last value loaded to the counter. **Values:** ■ false (0) ■ true (1) **Default Value:** true **Enabled:** RTC_WRAP_MODE==1 **Parameter Name:** RTC_WRAP_2_ZERO

Table 3-1 Top Level Parameters (Continued)

Label	Description
RTC Prescaler Counter Configuration	
RTC Prescaler Counter enable?	This parameter is used to enable the RTC Prescaler Counter logic in the DW_apb_rtc. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** Always **Parameter Name:** RTC_PRESCLR_EN
RTC Prescaler Counter Width	Defines width of RTC Prescaler Counter register. This is used for both RTC Prescaler Counter and RTC_CPSR Registers. **Values:** 2, ..., 32 **Default Value:** 16 **Enabled:** RTC_PRESCLR_EN **Parameter Name:** RTC_PRESCLR_WIDTH
Default Value of RTC Prescaler Counter Register	Defines the default value of RTC Prescaler Counter register. **Values:** 2, ..., 4294967295 **Default Value:** (RTC_PRESCLR_EN ==1) ? 32768:2 **Enabled:** RTC_PRESCLR_EN **Parameter Name:** RTC_PRESCLR_VAL
Hardcode Prescaler value	This parameter is used for hardcoding the RTC_CPSR. If this is true, then RTC_CPSR register will become read-only. Otherwise, the register is read/write capable. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** RTC_PRESCLR_EN **Parameter Name:** RTC_PRESCLR_VAL_HC

Table 3-1 Top Level Parameters (Continued)

Label	Description
Interrupt Generation Configuration	
Interrupt in pclk domain?	Describes the clock domain in which the interrupt is generated. When an interrupt is generated in the pclk domain, a free-running pclk is required to transfer the counter value each time it changes. When generated in the rtc_clk domain, then only the rtc_clk must be present to detect the interrupt. **Values:** ■ false (0) ■ true (1) **Default Value:** true **Enabled:** Always **Parameter Name:** RTC_INT_LOC
Active high interrupt?	Describes the polarity of the generated interrupt. **Values:** ■ false (0) ■ true (1) **Default Value:** true **Enabled:** Always **Parameter Name:** RTC_INT_POL

4 Signal Descriptions

This chapter details all possible I/O signals in the controller. For configurable IP titles, your actual configuration might not contain all of these signals.

Inputs are on the left of the signal diagrams; outputs are on the right.

Attention: For configurable IP titles, do not use this document to determine the exact I/O footprint of the controller. It is for reference purposes only.

When you configure the controller in coreConsultant, you must access the I/O signals for your actual configuration at workspace/report/IO.html or workspace/report/IO.xml after you have completed the report creation activity. That report comes from the exact same source as this chapter but removes all the I/O signals that are not in your actual configuration. This does not apply to non-configurable IP titles. In addition, all parameter expressions are evaluated to actual values. Therefore, the widths might change depending on your actual configuration.

Some expressions might refer to TCL functions or procedures (sometimes identified as **<functionof>)** that coreConsultant uses to make calculations. The exact formula used by these TCL functions is not provided in this chapter. However, when you configure the controller in coreConsultant, all TCL functions and parameters are evaluated completely; and the resulting values are displayed where appropriate in the coreConsultant GUI reports.

In addition to describing the function of each signal, the signal descriptions in this chapter include the following information:

Active State: Indicates whether the signal is active high or active low. When a signal is not intended to be used in a particular application, then this signal needs to be tied or driven to the inactive state (opposite of the active state).

Registered: Indicates whether or not the signal is registered directly inside the IP boundary without intervening logic (excluding simple buffers). A value of No does not imply that the signal is not synchronous, only that there is some combinatorial logic between the signal's origin or destination register and the boundary of the controller. A value of N/A indicates that this information is not provided for this IP title.

Synchronous to: Indicates which clocks in the IP sample this input (drive for an output) when considering all possible configurations. A particular configuration might not have all of the clocks listed. This clock might not be the same as the clock that your application logic should use to clock (sample/drive) this pin. For more details, consult the clock section in the databook.

Exists: Names of configuration parameters that populate this signal in your configuration.

Validated by: Assertion or de-assertion of signals that validates the signal being described.

Attributes used with Synchronous To

- Clock name - The name of the clock that samples an input or drive and output.
- None - This attribute may be used for clock inputs, hard-coded outputs, feed-through (direct or combinatorial), dangling inputs, unused inputs and asynchronous outputs.
- Asynchronous - This attribute is used for asynchronous inputs and asynchronous resets.

The I/O signals are grouped as follows:

- APB Slave Interface on page 35
- Real Time Clock Signals on page 38
- Miscellaneous on page 40
- Interrupt Interface on page 41

4.1 APB Slave Interface Signals

Table 4-1 APB Slave Interface Signals

Port Name	I/O	Description
pclk	I	APB clock. **Exists:** Always **Synchronous To:** None **Registered:** N/A **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
presetn	I	An active-low, asynchronous APB interface domain reset. This signal resets only the bus interface. The signal is asserted asynchronously, but is deasserted synchronously after the rising edge of pclk. The synchronization must be provided external to this component **Exists:** Always **Synchronous To:** Asynchronous **Registered:** N/A **Power Domain:** SINGLE_DOMAIN **Active State:** Low
penable	I	APB enable control that indicates second cycle of APB frame. **Exists:** Always **Synchronous To:** pclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
pwrite	I	APB write control. **Exists:** Always **Synchronous To:** pclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High

Table 4-1 APB Slave Interface Signals (Continued)

Port Name	I/O	Description
pprot[2:0]	I	APB4 Protection type. The input bits should match the corresponding protection activated level bit of the accessed register to gain access to the RTC load-count registers. Else the DW_apb_rtc generates an error. If protection level is turned off, any value on the corresponding bit is acceptable. Signal is ignored if SLVERR_RESP_EN==0. **Exists:** SLAVE_INTERFACE_TYPE>1 **Synchronous To:** pclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
pstrb[((APB_DATA_WIDTH/8)-1):0]	I	APB4 Write strobe bus. A high on individual bits in the pstrb bus indicate that the corresponding incoming write data byte on APB bus is to be updated in the addressed register. **Exists:** SLAVE_INTERFACE_TYPE>1 **Synchronous To:** pclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
pwdata[(APB_DATA_WIDTH-1):0]	I	APB write data bus. **Exists:** Always **Synchronous To:** pclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
paddr[RTC_ADDR_SLICE_LHS:0]	I	APB address bus. **Exists:** Always **Synchronous To:** pclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
psel	I	APB peripheral select. **Exists:** Always **Synchronous To:** pclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High

Table 4-1 APB Slave Interface Signals (Continued)

Port Name	I/O	Description
pready	O	This APB3 protocol signal indicates the end of a transaction when high in the access phase of a transaction. PREADY never goes low in DW_apb_rtc and is tied to one. **Exists:** SLAVE_INTERFACE_TYPE>0 **Synchronous To:** pclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
pslverr	O	APB3 slave error response signal. The signal issues an error when some error condition occurs, as specified in Slave error response section. **Exists:** SLAVE_INTERFACE_TYPE>0 **Synchronous To:** pclk **Registered:** (SLAVE_INTERFACE_TYPE > 1 && SLVERR_RESP_EN==1) ? Yes : No **Power Domain:** SINGLE_DOMAIN **Active State:** High
prdata[(APB_DATA_WIDTH-1):0]	O	APB readback data. **Exists:** Always **Synchronous To:** pclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A

4.2 Real Time Clock Signals

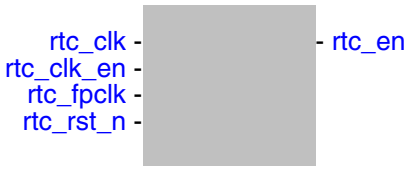


Table 4-2 Real Time Clock Signals

Port Name	I/O	Description
rtc_clk	I	Optional. Real-time counter clock. **Exists:** RTC_CLK_EN==0 **Synchronous To:** None **Registered:** N/A **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
rtc_clk_en	I	Optional. Along with pclk, this signal allows the RTC to be clocked. This signal allows the DW_apb_rtc to be run at a lower rate without the use of a separate RTC clock. **Exists:** RTC_CLK_EN==1 **Synchronous To:** pclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
rtc_fpclk	I	Optional. Free-running pclk that is used to re-time the counter clock so that load values and match values can be transferred to the counter clock domain and interrupts can be generated. This signal is phase and frequency coherent with the bus clock, pclk. A free-running pclk is connected to rtc_fpclk when the relationship between pclk and rtc_clk is asynchronous in order to allow data to be passed over the clock domain. **Exists:** RTC_FREE_PCLK==0 **Synchronous To:** pclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A

Table 4-2 Real Time Clock Signals (Continued)

Port Name	I/O	Description
rtc_rst_n	I	Active-low RTC counter clock domain reset signal. Asynchronous assertion; de-assertion synchronous to counter clock domain **Exists:** Always **Synchronous To:** Asynchronous **Registered:** N/A **Power Domain:** SINGLE_DOMAIN **Active State:** Low
rtc_en	O	Optional. The RTC is enabled and requires a free-running clock. It is not possible to generate an interrupt when rtc_en is low. **Exists:** RTC_EN_MODE==1 **Synchronous To:** pclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** Low

4.3 Miscellaneous Signals

Table 4-3 Miscellaneous Signals

Port Name	I/O	Description
scan_mode	I	Optional. Scan mode. This signal should be asserted that is, driven to logic 1 during scan testing and should be deasserted that is, tied to logic 0 at all other times. **Exists:** (RTC_CLK_TYPE == 1) || ((RTC_INT_LOC == 0) && (RTC_CLK_TYPE != 0)) **Synchronous To:** Asynchronous **Registered:** ((RTC_INT_LOC==0)&&(RTC_CLK_TYPE!=0)) ? Yes : No **Power Domain:** SINGLE_DOMAIN **Active State:** High

4.4 Interrupt Interface Signals

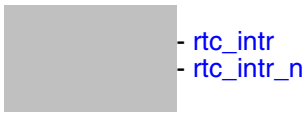


Table 4-4 Interrupt Interface Signals

Port Name	I/O	Description
rtc_intr	O	Optional. Active-high interrupt. **Note:** If RTC_EN_MODE = 1 and rtc_en is low, this signal is set to 0. **Exists:** RTC_INT_POL==1 **Synchronous To:** (RTC_INT_LOC == 1) ? "pclk" : "rtc_clk" **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
rtc_intr_n	O	Optional. Active-Low interrupt. **Note:** If RTC_EN_MODE = 1 and rtc_en is low, this signal is set to 1. **Exists:** RTC_INT_POL==0 **Synchronous To:** (RTC_INT_LOC == 1) ? "pclk" : "rtc_clk" **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** Low

5 Register Descriptions

This chapter details all possible registers in the IP. They are arranged hierarchically into maps and blocks (banks).Your actual configuration might not contain all of these registers.

Attention: For configurable IP titles, do not use this document to determine the exact attributes of your register map. It is for reference purposes only.

When you configure the controller in coreConsultant, you must access the register attributes for your actual configuration at workspace/report/ComponentRegisters.html or workspace/report/ComponentRegisters.xml after you have completed the report creation activity. That report comes from the exact same source as this chapter but removes all the registers that are not in your actual configuration. This does not apply to non-configurable IP titles. In addition, all parameter expressions are evaluated to actual values. Therefore, the Offset and Memory Access values might change depending on your actual configuration.

Some expressions might refer to TCL functions or procedures (sometimes identified as **<functionof>)** that coreConsultant uses to make calculations. The exact formula used by these TCL functions is not provided in this chapter. However, when you configure the controller in coreConsultant, all TCL functions and parameters are evaluated completely; and the resulting values are displayed where appropriate in the coreConsultant GUI reports.

Exists Expressions

These expressions indicate the combination of configuration parameters required for a register, field, or block to exist in the memory map. The expression is only valid in the local context and does not indicate the conditions for existence of the parent. For example, the expression for a bit field in a register assumes that the register exists and does not include the conditions for existence of the register.

Offset

The term *Offset* is synonymous with *Address*.

Memory Access Attributes

The Memory Access attribute is defined as <ReadBehavior>/<WriteBehavior> which are defined in the following table.

Table 5-1 Possible Read and Write Behaviors

Read (or Write) Behavior	Description
RC	A read clears this register field.
RS	A read sets this register field.
RM	A read modifies the contents of this register field.
Wo	You can only write once to this register field.
W1C	A write of 1 clears this register field.
W1S	A write of 1 sets this register field.
W1T	A write of 1 toggles this register field.
W0C	A write of 0 clears this register field.
W0S	A write of 0 sets this register field.
W0T	A write of 0 toggles this register field.
WC	Any write clears this register field.
WS	Any write sets this register field.
WM	Any write toggles this register field.
no Read Behavior attribute	You cannot read this register. It is Write-Only.
no Write Behavior attribute	You cannot write to this register. It is Read-Only.

Table 5-2 Memory Access Examples

Memory Access	Description
R	Read-only register field.
W	Write-only register field.
R/W	Read/write register field.
R/W1C	You can read this register field. Writing 1 clears it.
RC/W1C	Reading this register field clears it. Writing 1 clears it.
R/Wo	You can read this register field. You can only write to it once.

Special Optional Attributes

Some register fields might use the following optional attributes.

Table 5-3 Optional Attributes

Attribute	Description
Volatile	As defined by the IP-XACT specification. If true, indicates in the case of a write followed by read, or in the case of two consecutive reads, there is no guarantee as to what is returned by the read on the second transaction or that this return value is consistent with the write or read of the first transaction. The element implies there is some additional mechanism by which this field can acquire new values other than by reads/writes/resets and other access methods known to IP-XACT. For example, when the core updates the register field contents.
Testable	As defined by the IP-XACT specification. Possible values are unconstrained, untestable, readOnly, writeAsRead, restore. Untestable means that this field is untestable by a simple automated register test. For example, the read-write access of the register is controlled by a pin or another register. readOnly means that you should not write to this register; only read from it. This might apply for a register that modifies the contents of another register.
Reset Mask	As defined by the IP-XACT specification. Indicates that this register field has an unknown reset value. For example, the reset value is set by another register or an input pin; or the register is implemented using RAM.
* Varies	Indicates that the memory access (or reset) attribute (read, write behavior) is not fixed. For example, the read-write access of the register is controlled by a pin or another register. Or when the access depends on some configuration parameter; in this case the post-configuration report in coreConsultant gives the actual access value.

Component Banks/Blocks

The following table shows the address blocks for each memory map. Follow the link for an address block to see a table of its registers.

Table 5-4 Address Banks/Blocks for Memory Map: rtc_memory_map

Address Block	Description
rtc_address_block on page 46	DW_apb_rtc address block **Exists:** Always

5.1 rtc_memory_map/rtc_address_block Registers

DW_apb_rtc address block. Follow the link for the register to see a detailed description of the register.

Table 5-5 Registers for Address Block: rtc_memory_map/rtc_address_block

Register	Offset	Description
RTC_CCVR on page 47	0x0	Current Counter Value Register
RTC_CMR on page 48	0x4	Counter Match Register
RTC_CLR on page 49	0x8	Counter Load Register
RTC_CCR on page 50	0xc	Counter Control Register
RTC_STAT on page 53	0x10	Interrupt Status Register
RTC_RSTAT on page 54	0x14	Interrupt Raw Status Register
RTC_EOI on page 55	0x18	End of Interrupt Register
RTC_COMP_VERSION on page 56	0x1c	Component Version Register
RTC_CPSR on page 57	0x20	Counter PreScaler Register
RTC_CPCVR on page 58	0x24	Current Prescaler Counter Value Register

5.1.1 RTC_CCVR

- **Name:** Current Counter Value Register
- **Description:** Current Counter Value Register
- **Size:** 32 bits
- **Offset:** 0x0
- **Exists:** Always

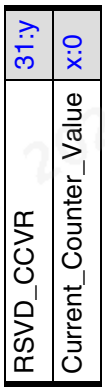


Table 5-6 Fields for Register: RTC_CCVR

Bits	Name	Memory Access	Description
31:y	RSVD_CCVR	R	RTC_CCVR 31 to RTC_CNT_WIDTH Reserved bits - Read Only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** RTC_CNT_WIDTH
x:0	Current_Counter_Value	R	When read, this register is the current value of the internal counter. This value is always read coherently. Bits from RTC_CNT_WIDTH to 31 are read as 0 when RTC_CNT_WIDTH is less than 31. **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** RTC_CNT_WIDTH - 1

5.1.2 RTC_CMR

- **Name:** Counter Match Register
- **Description:** Counter Match Register
- **Size:** 32 bits
- **Offset:** 0x4
- **Exists:** Always

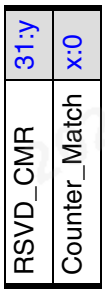


Table 5-7 Fields for Register: RTC_CMR

Bits	Name	Memory Access	Description
31:y	RSVD_CMR	R	RTC_CMR 31toRTC_CNT_WIDTH Reserved bits - Read Only **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** RTC_CNT_WIDTH
x:0	Counter_Match	R/W	Interrupt Match Register. When the internal counter matches this register, an interrupt is generated, provided interrupt generation is enabled. When appropriate, this value is written coherently. Only when all the bytes are written is the register used by the interrupt detection logic. Bits from RTC_CNT_WIDTH and above are read and written as 0 when RTC_CNT_WIDTH is less than 31. **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** RTC_CNT_WIDTH - 1

5.1.3 RTC_CLR

- **Name:** Counter Load Register
- **Description:** Counter Load Register
- **Size:** 32 bits
- **Offset:** 0x8
- **Exists:** Always

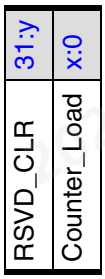


Table 5-8 Fields for Register: RTC_CLR

Bits	Name	Memory Access	Description
31:y	RSVD_CLR	R	RTC_CLR 31toRTC_CNT_WIDTH Reserved bits - Read Only **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** RTC_CNT_WIDTH
x:0	Counter_Load	R/W	Loaded into the counter as the loaded value, which is written coherently. Bits from RTC_CNT_WIDTH and above are read and written as 0 when RTC_CNT_WIDTH is less than 31. **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** RTC_CNT_WIDTH - 1

5.1.4 RTC_CCR

- **Name:** Counter Control Register
- **Description:** Counter Control Register.

  Note: If the RTC_RSTAT register indicates that a pending interrupt exists because a masked interrupt has been generated (rtc_mask bit of RTC_CCR is set to 1), then caution should be taken when programming RTC_CCR. That is, if both the rtc_mask and rtc_en bits of the RTC_CCR register are set to 0, then the interrupt asserts for one clock period. To avoid this scenario, you can use the following two-step process:

  1. Program the rtc_en bit of the RTC_CCR register to 0, which clears the interrupt.

  2. Then program the RTC_CCR register again to set the rtc_mask bit to 0, which unmasks any future interrupts.
- **Size:** 32 bits
- **Offset:** 0xc
- **Exists:** Always

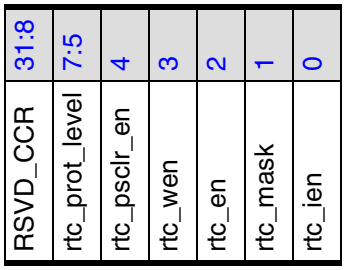


Table 5-9 Fields for Register: RTC_CCR

Bits	Name	Memory Access	Description
31:8	RSVD_CCR	R	RTC_CCR 31to8 Reserved and read as 0. **Value After Reset:** 0x0 **Exists:** Always
7:5	rtc_prot_level	* Varies	This field holds the protection level value of DW_apb_rtc. **Value After Reset:** PROT_LEVEL_RST **Exists:** (SLVERR_RESP_EN==1) **Memory Access:** {(HC_PROT_LEVEL==0) ? "read-write" : "read-only"}

Table 5-9 Fields for Register: RTC_CCR (Continued)

Bits	Name	Memory Access	Description
4	rtc_psclr_en	* Varies	Optional. Allows user to control the usage of RTC Prescaler feature. **Values:** ■ 0x0 (DISABLED): Disables the Prescaler counter ■ 0x1 (ENABLED): Enables the Prescaler counter **Value After Reset:** 0x0 **Exists:** Always **Memory Access:** "((RTC_PRESCLR_EN==1) && (RTC_EN_MODE==1)) ? \"read-write\" : \"read-only\""
3	rtc_wen	* Varies	*Optional*. Allows the user to force the counter to wrap when a match occurs instead of waiting until the maximum count is reached. 0 = Wrap disabled, 1 = Wrap enabled, This bit is writable only when RTC_WRAP_MODE = 1 **Values:** ■ 0x0 (DISABLED): Disables the WRAP ■ 0x1 (ENABLED): Enables the WRAP **Value After Reset:** 0x0 **Exists:** Always **Memory Access:** "(RTC_WRAP_MODE==1) ? \"read-write\" : \"read-only\""
2	rtc_en	* Varies	Optional. Allows the user to control counting in the counter. This bit does not exist if RTC_EN_MODE = 0. Internally, the counter is always enabled. **Values:** ■ 0x0 (DISABLED): Disables the counter ■ 0x1 (ENABLED): Enables the counter **Value After Reset:** 0x0 **Exists:** Always **Memory Access:** "(RTC_EN_MODE==1) ? \"read-write\" : \"read-only\""
1	rtc_mask	R/W	Allows the user to mask interrupt generation. **Values:** ■ 0x0 (UNMASKED): Interrupt unmasked ■ 0x1 (MASKED): Interrupt masked **Value After Reset:** 0x0 **Exists:** Always

Table 5-9 Fields for Register: RTC_CCR (Continued)

Bits	Name	Memory Access	Description
0	rtc_ien	R/W	Allows the user to disable interrupt generation. **Values:** ■ 0x0 (DISABLED): Disables the interrupt generation ■ 0x1 (ENABLED): Enables the interrupt generation **Value After Reset:** 0x0 **Exists:** Always

5.1.5 RTC_STAT

- **Name:** Interrupt Status Register
- **Description:** Interrupt Status Register
- **Size:** 32 bits
- **Offset:** 0x10
- **Exists:** Always

Table 5-10 Fields for Register: RTC_STAT

Bits	Name	Memory Access	Description
31:1	RSVD_RTC_STAT	R	RTC_STAT 31to1 Reserved and read as 0. **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true
0	rtc_stat	R	This register is the masked raw status. **Values:** ■ 0x0 (INACTIVE): Interrupt is inactive ■ 0x1 (ACTIVE): Interrupt is active (regardless of polarity) **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true

5.1.6 RTC_RSTAT

- **Name:** Interrupt Raw Status Register
- **Description:** Interrupt Raw Status Register
- **Size:** 32 bits
- **Offset:** 0x14
- **Exists:** Always

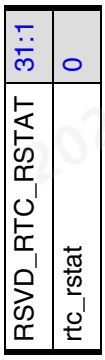


Table 5-11 Fields for Register: RTC_RSTAT

Bits	Name	Memory Access	Description
31:1	RSVD_RTC_RSTAT	R	RTC_RSTAT 31to1 Reserved and read as 0. **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true
0	rtc_rstat	R	Raw Status **Values:** ■ 0x0 (INACTIVE): Interrupt is inactive ■ 0x1 (ACTIVE): Interrupt is active (regardless of polarity) **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true

5.1.7 RTC_EOI

- **Name:** End of Interrupt Register
- **Description:** End of Interrupt Register
- **Size:** 32 bits
- **Offset:** 0x18
- **Exists:** Always

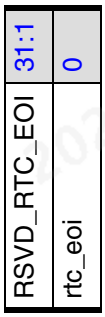


Table 5-12 Fields for Register: RTC_EOI

Bits	Name	Memory Access	Description
31:1	RSVD_RTC_EOI	R	RTC_EOI 31to1 Reserved and read as 0. **Value After Reset:** 0x0 **Exists:** Always
0	rtc_eoi	R	By reading this location, the match interrupt is cleared. Performing read-to-clear on interrupt, the interrupt is cleared at the end of the read. **Value After Reset:** 0x0 **Exists:** Always

5.1.8 RTC_COMP_VERSION

- **Name:** Component Version Register
- **Description:** Component Version Register
- **Size:** 32 bits
- **Offset:** 0x1c
- **Exists:** Always

Table 5-13 Fields for Register: RTC_COMP_VERSION

Bits	Name	Memory Access	Description
31:0	rtc_comp_version	R	ASCII value for each number in the version, followed by *. For example, 32_30_31_2A represents the version 2.01*. **Value After Reset:** RTC_VERSION_ID **Exists:** Always

5.1.9 RTC_CPSR

- **Name:** Counter PreScaler Register
- **Description:** Counter PreScaler Register
- **Size:** 32 bits
- **Offset:** 0x20
- **Exists:** RTC_PRESCLR_EN == 1

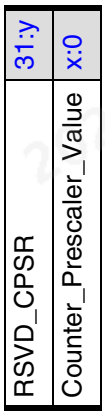


Table 5-14 Fields for Register: RTC_CPSR

Bits	Name	Memory Access	Description
31:y	RSVD_CPSR	R	RTC_CPSR 31toRTC_PRESCLR_WIDTH Reserved bits - Read Only **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** RTC_PRESCLR_WIDTH
x:0	Counter_Prescaler_Value	R/W	Counter Prescaler Register. The RTC counter will be updating at the rate of rtc_clk, rtc_clk_en, or pclk based on the configuration. This register is used to prescale the rate at which the RTC counter updates. When appropriate, this register is written coherently. Only when all the bytes are written, the register used by the prescaler counter logic. **Value After Reset:** RTC_PRESCLR_VAL **Exists:** Always **Range Variable[x]:** RTC_PRESCLR_WIDTH - 1

5.1.10 RTC_CPCVR

- **Name:** Current Prescaler Counter Value Register
- **Description:** Current Prescaler Counter Value Register
- **Size:** 32 bits
- **Offset:** 0x24
- **Exists:** RTC_PRESCLR_EN == 1

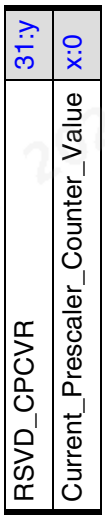


Table 5-15 Fields for Register: RTC_CPCVR

Bits	Name	Memory Access	Description
31:y	RSVD_CPCVR	R	RTC_CPCVR 31toRTC_PRESCLR_WIDTH Reserved bits - Read Only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** RTC_PRESCLR_WIDTH
x:0	Current_Prescaler_Counter_Value	R	When read, this register provides the current value of the internal prescaler counter. This value always is read coherently. Bits from RTC_PRESCLR_WIDTH to 31 are read as 0 when RTC_PRESCLR_WIDTH is less than 31. **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** RTC_PRESCLR_WIDTH - 1

6 Programming the DW_apb_rtc

This chapter describes the programmable features of the DW_apb_rtc.

6.1 Programming Considerations

The DW_apb_rtc module is an APB little-endian, slave peripheral. As the largest register width can be 32-bits, all registers are aligned to 32-bit boundaries. Regardless of the APB bus width, aligning to 32-bit boundaries keeps the same memory map for all bus widths. The APB bus reset signal (presetn) resets all registers within the programming interface section. The base address of DW_apb_rtc is not fixed and is determined by the DW_apb when generating the psel for DW_apb_rtc. Offset addresses from the base address are used for each register.

Some registers require coherency circuitry on write and others on read. When a register is narrower than the smallest data bus width, there is no need for read or write coherency logic, and therefore it is not instantiated.

Note The coherency circuitry incorporates an upper byte method that requires users to program the load register in LSB to MSB order when the peripheral width is smaller than the register width.

When the upper byte is programmed, the value can be transferred and loaded into the load register. When the lower bytes are programmed, they need to be stored in shadow registers so that the previous load register is available to the counter if it needs to reload. When the upper byte is programmed, the contents of the shadow registers and the upper byte are loaded into the load register.

Note Users must read LSB to MSB for the coherency circuitry solution to operate correctly.

7 Verification

This chapter provides an overview of the testbench available for the DW_apb_rtc verification. After the DW_apb_rtc has been configured and the verification setup, simulations can be run automatically. For information on running simulations for DW_apb_rtc in coreAssembler or coreConsultant, see *DesignWare Synthesizable Components for AMBA 2 User Guide*.

Note The DW_apb_rtc verification testbench is built using Synopsys SVT Verification IP (VIP). Ensure that you have the supported version of the VIP components for this release, otherwise, you may experience some tool compatibility problems. For more information about supported tools in this release, see the *DesignWare Synthesizable Components for AMBA 2/AMBA 3 AXI Installation Guide*.

Note The packaged test benches are only for validating the IP configuration in coreConsultant GUI. It is not for system level validation.
IPs that have the Vera test bench packaged, these test benches are encrypted.

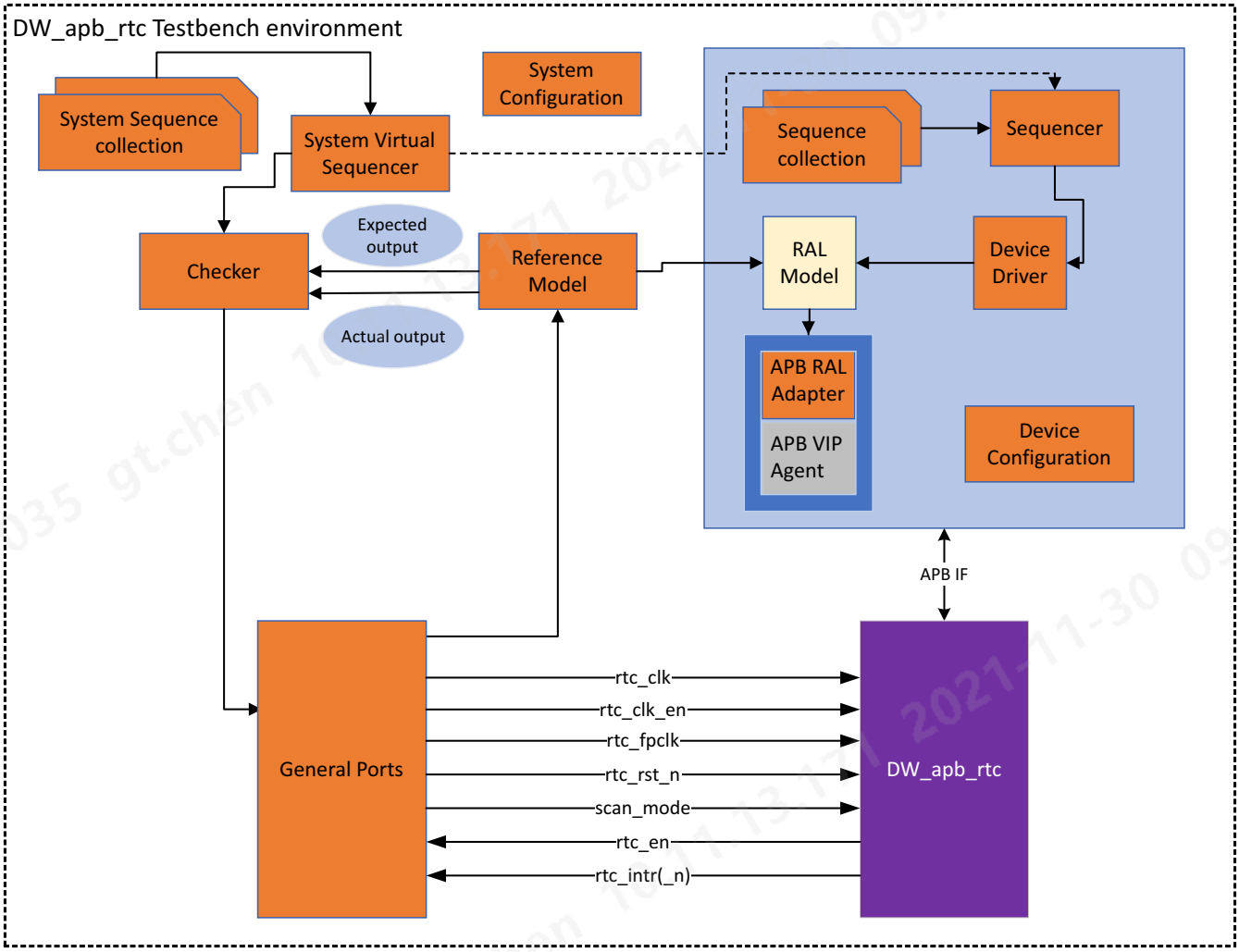
This chapter discusses the following sections:

7.1 Verification Environment

DW_apb_rtc is verified using a UVM-methodology-based constrained random verification environment. The environment is capable of generating random scenarios and the test case has hooks to control the scenarios to be generated.

Figure 7-1 shows the verification environment of the DW_apb_rtc testbench:

Figure 7-1 DW_apb_rtc UVM Verification Environment



The testbench consists of the following elements:

- Testbench uses the standard SVT VIP for the protocol interfaces:
  - APB VIP Interface for connecting master and slave ports of DUT.
  - APB VIP system environment (svt_apb_system_env): APB master agent and APB slave agent for providing the VIP supported randomized transactions with bus instantiation on either side of the DUT.

Testbench uses the custom components:

- Device Agent

  This component is responsible for creating traffic on the application bus interface (APB). This agent follows the standard structure of a UVM agent and has the following sub-blocks:

- Device Sequencer

  This component is a standard UVM sequencer, which fetches the top-level sequence items from the scenario sequences and feeds the device driver. There is no additional logic present in the device sequencer.

- Device Driver

  The core logic of device agent sits in the device driver. This block is responsible for the APB side read and write. The bus protocol driver along with the RAL forms the lower layer, which directly interacts with the design bus interface.

  On the Write and Read paths, the device driver fetches a protocol transaction from the scenario sequences (via device sequencer) and initiates the respective APB register writes or reads.

  The device driver is planned to be independent of the application bus interface. This is accomplished by using RAL. In case of a change of bus protocol, the only change is going to be in the RAL adapter logic.

- RAL Model

  Inside Device Agent we will have RAL model. RAL model is used for two purposes: first is to show (shadow) the values of registers inside DW_apb_rtc and the second is checking whether the values of registers is correct upon reading. In order to facilitate that RAL model will have a two-way connection with the Reference model. Reference model will get the values from RAL to check whether behavior on General Ports is correct and will also be responsible for updates of the RAL, since Device agent is unable to see the activity that is happening on General Ports, where transaction can also change the values of registers in DW_apb_rtc (mostly status registers).

  RAL model is used for its flexibility and reuse because if the adapter and VIP agent (interface we use) is changed, all the other logic can be kept.

- APB RAL Adapter

  APB RAL Adapter converts higher level RAL Model Reads and Writes to APB transactions, these transactions will be driven on the bus by APB VIP Agent. By replacing this part with the different adapter, we make this environment capable of being used by other interfaces.

- APB VIP Agent

  APB VIP Agent drives transactions on APB interface. By replacing APB VIP Agent and APB RAL Adapter with different components (such as AHB components) we can keep the other logic in Device Agent, and use it for that other interface.

- System Virtual Sequencer

  The main responsibility of System Virtual Sequencer is to synchronize General Port's sequencer and RAL sequencer.

- Reference Model / Checker / Scoreboard

The Reference Model is responsible for updating the RAL content for the activity done outside of the main register interface.

Based on the transactions received from the agent monitors (General Port's monitors) and values of registers in RAL, the Reference Model also calculates the expected values of DUT output signals. The actual DUT outputs, as well as the expected values, are then sent to the Checker. In the Checker comparing the expected values of DUT output signals against the values actually driven by the DUT is done.

7.2 Testbench Directories and Files

The DW_apb_rtc verification environment contains the following directories and associated files.

Table 7-1 shows the various directories and associated files:

Table 7-1 DW_apb_rtc Testbench Directory Structure

Directory	Description
<configured workspace>/sim/testbench	Top level testbench module (test_top.sv) and the DUT to the testbench wrapper (dut_sv_wrapper.sv) exist in this folder
<configured workspace>/sim/testbench/env	Contains testbench files. For example, scoreboard, sequences, VIP, environment, sequencers, and agents.
<configured workspace>/sim/	Primarily contains the supporting files to compile and run the simulation. After the completion of the simulation, the log files are present here.
<configured workspace>/sim/test_*	Contains individual test cases. After the completion of the simulation, the test specific log files and if applicable the waveform files are stored here

7.3 Packaged Testcases

The simulation environment that comes as a package files includes some demonstrative tests. Some or all of the packaged demonstrative tests, depending upon their applicability to the chosen configuration, are displayed in Setup and Run Simulations > Testcases in the coreConsultant GUI.

The associated test cases shipped and their description is explained in Table 7-2:

Table 7-2 DW_apb_rtc Test Description

Test Name	Test Description
test_100_reg_reset_bit_bash	This test builds the sequence `dw_apb_rtc_reg_reset_bit_bash_virtual_sequence` which in turn initiates `dw_apb_rtc_reg_reset_bit_bash_sequence`. This test performs ■ Validation of register values after reset. ■ Bit bash for the bits of all the registers.

Table 7-2 DW_apb_rtc Test Description (Continued)

Test Name	Test Description
test_101_rtc_random	This test initiates the random testing of all the other test sequences. It validates the following 1. This test builds the sequence `dw_apb_rtc_enable_virtual_sequence`. If RTC_EN_MODE = 0, It checks a free running counter by reading CCVR register. If RTC_EN_MODE = 1, then it sets rtc_en in CCR register in turn monitors the counter value. 2. This test builds the sequence `dw_apb_rtc_wrap_virtual_sequence`. On RTC_WRAP_MODE = 1, this test checks wrap functionality. - Enabling wrap mode in RTC CCR - Setting the RTC match value - Disabling the wrap in RTC CCR 3. This test builds the sequence `dw_apb_rtc_presclr_virtual_sequence`. - This test checks the prescalar functionality. - It enables the presclr bit and writes a randomized prescalar value. - It monitors the counter before and after enabling of the prescalar function. 4. This test builds the sequence `dw_apb_rtc_match_interrupt_virtual_sequence`. This test enables interrupt generation, setting the counter to the upper limit, interrupt masking and interrupt clearing. 5. This test builds the sequence `dw_apb_rtc_wrap_at_max_val_virtual_sequence.` - It enables the counter and sets a load value that is close to the counter max value. - It creates a scenario that the counter will wrap when reaching its max value.

8 Integration Considerations

After you have configured, tested, and synthesized your component with the coreTools flow, you can integrate the component into your own design environment. The following sections discuss general integration considerations.

8.1 Accessing Top-level Constraints

To get SDC constraints out of coreConsultant, you need to first complete the synthesis activity and then use the "write_sdc" command to write out the results:

1. This cC command sets synthesis to write out scripts only, without running DC:

   ```
   set_activity_parameter Synthesize ScriptsOnly 1
   ```

2. This cC command autocompletes the activity:

   ```
   autocomplete_activity Synthesize
   ```

3. Finally, this cC command writes out SDC constraints:

   ```
   write_sdc <filename>
   ```

8.2 Coherency

Coherency is where bits within a register are logically connected. For instance, part of a register is read at time 1 and another part is read at time 2. Being coherent means that the part read at time 2 is at the same value it was when the register was read at time 1. The unread part is stored into a shadow register and this is read at time 2. When there is no coherency, no shadow registers are involved.

A bus master may need to be able to read the contents of a register, regardless of the data bus width, and be guaranteed of the coherency of the value read. A bus master may need to be able to write a register coherently regardless of the data bus width and use that register only when it has been fully programmed. This may need to be the case regardless of the relationship between the clocks.

Coherency enables a value to be read that is an accurate reflection of the state of the counter, independent of the data bus width, the counter width, and even the relationship between the clocks. Additionally, a value written in one domain is transferred to another domain in a seamless and coherent fashion.

Throughout this appendix the following terms are used:

- **Writing**. A bus master programs a configuration register. An example is programming the load value of a counter into a register.
- **Transferring**. The programmed register is in a different clock domain to where it is used, therefore, it needs to be transferred to the other clock domain.
- **Loading**. Once the programmed register is transferred into the correct clock domain, it needs to be loaded or used to perform its function. For example, once the load value is transferred into the counter domain, it gets loaded into the counter.

8.2.1 Writing Coherently

Writing coherently means that all the bits of a register can be written at the same time. A peripheral may have programmable registers that are wider than the width of the connected APB data bus, which prevents all the bits being programmed at the same time unless additional coherency circuitry is provided.

The programmable register could be the load value for a counter that may exist in a different clock domain. Not only does the value to be programmed need to be coherent, it also needs to be transferred to a different clock domain and then loaded into the counter. Depending on the function of the programmable register, a qualifier may need to be generated with the data so that it knows when the new value is currently transferred and when it should be loaded into the counter.

Depending on the system and on the register being programmed, there may be no need for any special coherency circuitry. One example that requires coherency circuitry is a 32-bit timer within an 8-bit APB system. The value is entirely programmed only after four 8-bit wide write transfers. It is safe to transfer or use the register when the last byte is currently written. An example where no coherency is required is a 16-bit wide timer within a 16-bit APB system. The value is entirely programmed after a single 16-bit wide write transfer.

Coherency circuitry enables the value to be loaded into the counter only when fully programmed and crossed over clock domains if the peripheral clock is not synchronous to the processor clock. While the load register is being programmed, the counter has access to the previous load value in case it needs to reload the counter.

Coherency circuitry is only added in cores where it is needed. The coherency circuitry incorporates an upper byte method that requires users to program the load register in LSB to MSB order when the peripheral width is smaller than the register width. When the upper byte is programmed, the value can be transferred and loaded into the load register. When the lower bytes are being programmed, they need to be stored in shadow registers so that the previous load register is available to the counter if it needs to reload. When the upper byte is programmed, the contents of the shadow registers and the upper byte are loaded into the load register.

The upper byte is the top byte of a register. A register can be transferred and loaded into the counter only when it has been fully programmed. A new value is available to the counter once this upper byte is written into the register. The following table shows the relationship between the register width and the peripheral

bus width for the generation of the correct upper byte. The numbers in the table represent bytes, Byte 0 is the LSB and Byte 3 is the MSB. NCR means that no coherency circuitry is required, as the entire register is written with one access.

Table 8-1 Upper Byte Generation

	Upper Byte Bus Width		
Load Register Width	8	16	32
1 - 8	NCR	NCR	NCR
9 - 16	1	NCR	NCR
17 - 24	2	2	NCR
25 - 32	3	2 (or 3)	NCR

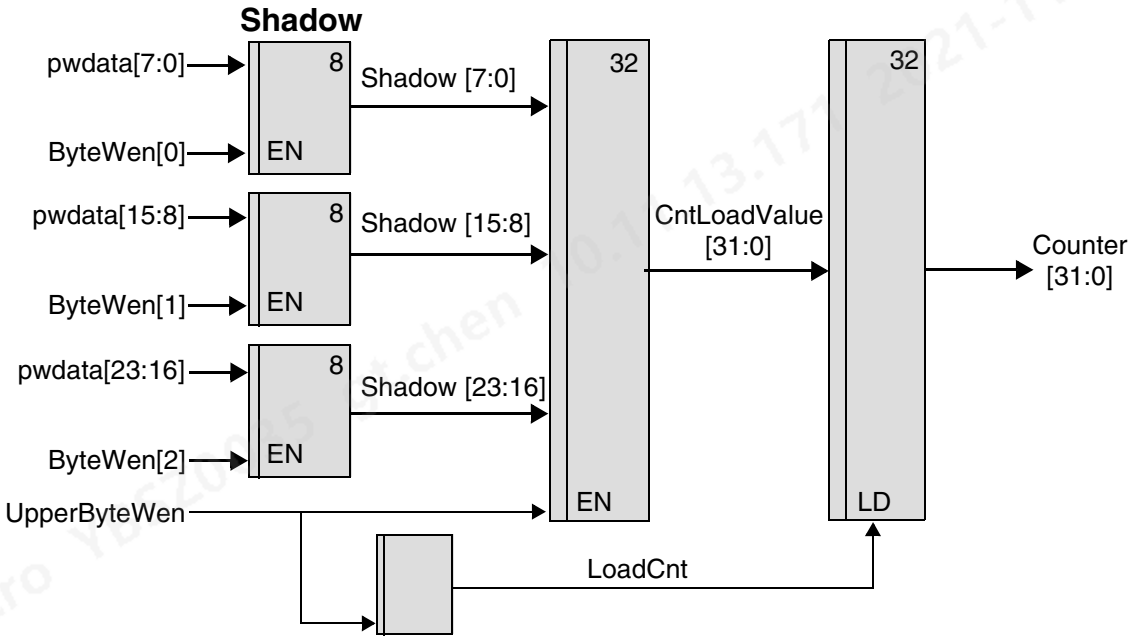
There are three relationship cases to be considered for the processor and peripheral clocks:

- Identical
- Synchronous (phase coherent but of an integer fraction)
- Asynchronous

8.2.1.1 Identical Clocks

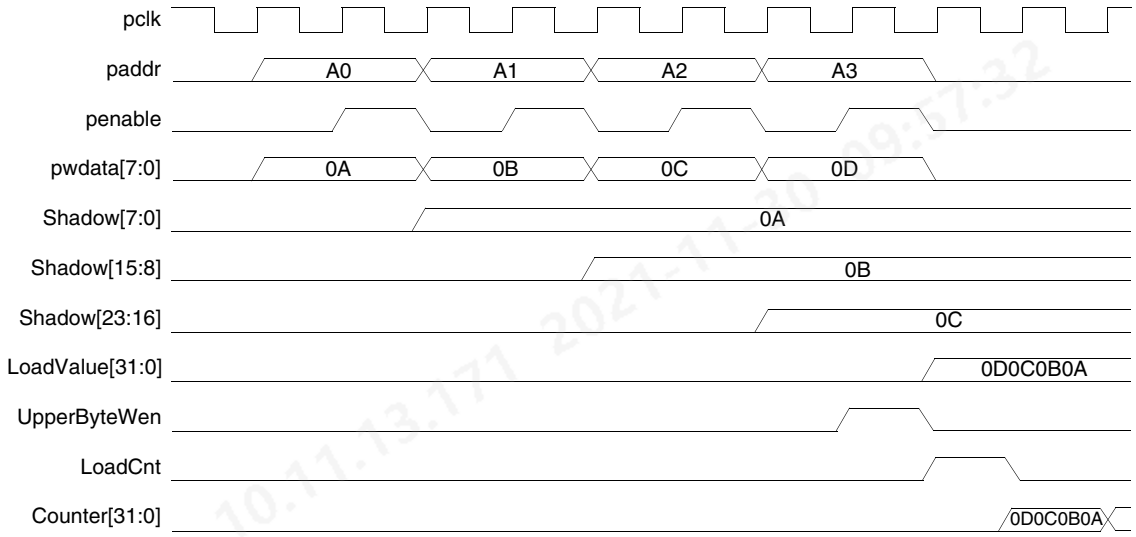
The following figure illustrates an RTL diagram for the circuitry required to implement the coherent write transaction when the APB bus clock and peripheral clocks are identical.

Figure 8-1 Coherent Loading – Identical Synchronous Clocks



The following figure shows a 32-bit register that is written over an 8-bit data bus, as well as the shadow registers being loaded and then loaded into the counter when fully programmed. The LoadCnt signal lasts for one cycle and is used to load the counter with CntLoadValue.

Figure 8-2 Coherent Loading – Identical Synchronous Clocks



Each of the bytes that make up the load register are stored into shadow registers until the final byte is written. The shadow register is up to three bytes wide. The contents of the shadow registers and the final byte are transferred into the CntLoadValue register when the final byte is written. The counter uses this register to load/initialize itself. If the counter is operating in a periodic mode, it reloads from this register each time the count expires.

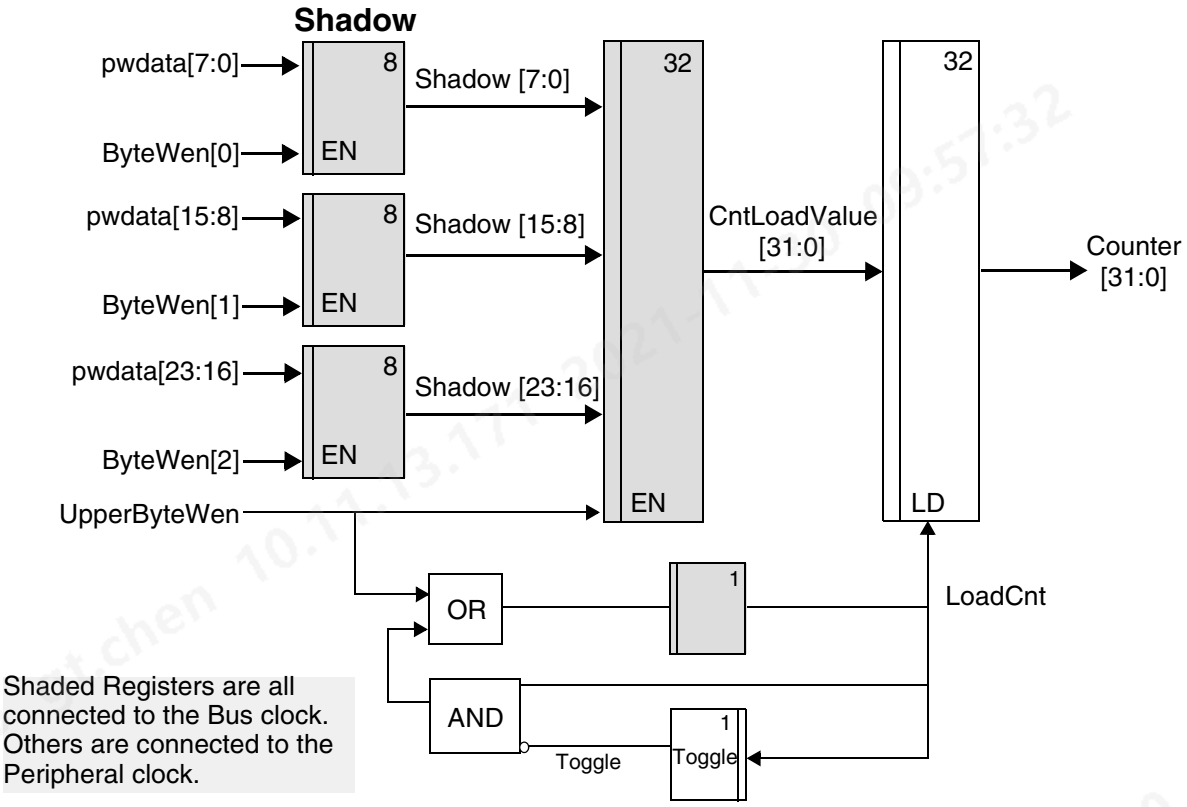
By using the shadow registers, the CntLoadValue is kept stable until it can be changed in one cycle. This allows the counter to be loaded in one access and the state of the counter is not affected by the latency in programming it. When there is a new value to be loaded into the counter initially, this is signaled by LoadCnt = 1. After the upper byte is written, the LoadCnt goes to zero.

8.2.1.2 Synchronous Clocks

When the clocks are synchronous but do not have identical periods, the circuitry needs to be extended so that the LoadCnt signal is kept high until a rising edge of the counter clock occurs. This extension is necessary so that the value can be loaded, using LoadCnt, into the counter on the first counter clock edge. At the rising edge of the counter clock if LoadCnt is high, then a register clocked with the counter clock toggles, otherwise it keeps its current value. A circuit detecting the toggling is used to clear the original LoadCnt by looking for edge changes. The value is loaded into the counter when a toggle has been detected. Once it is loaded, the counter should be free to increment or decrement by normal rules.

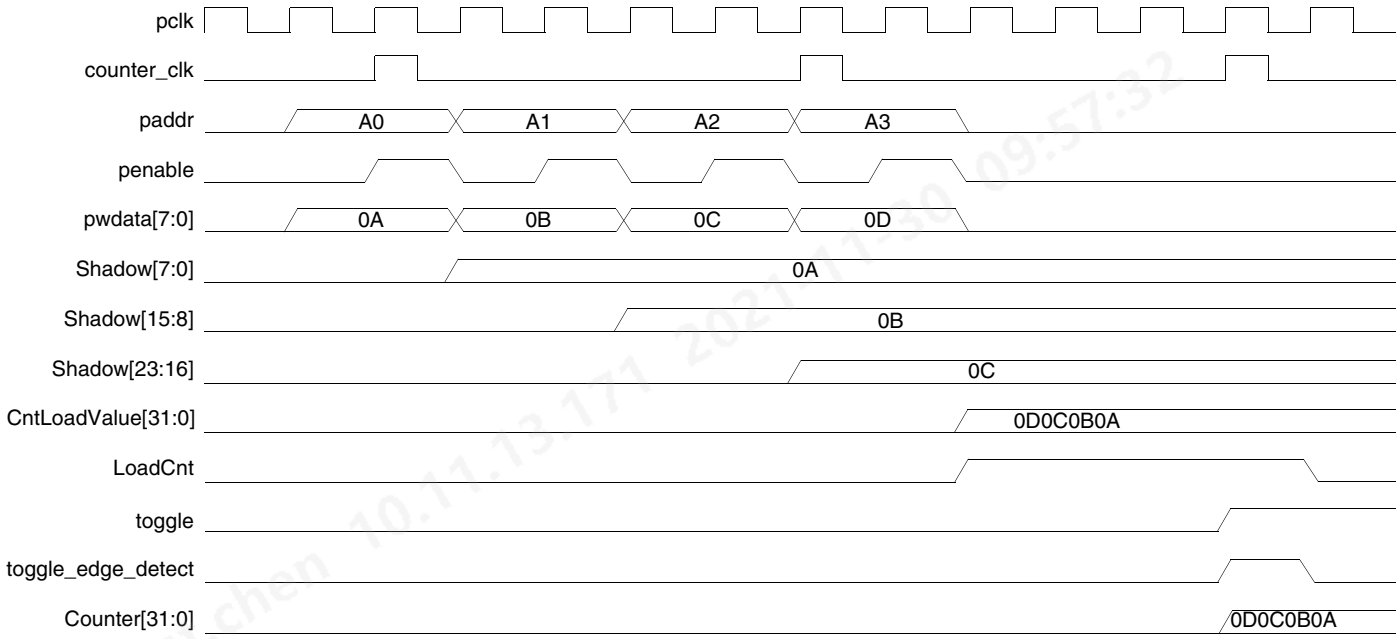
The following figure shows an RTL diagram for the circuitry required to implement the coherent write when the bus and peripheral clocks are synchronous.

Figure 8-3 Coherent Loading – Synchronous Clocks



The following figure shows a 32-bit register being written over an 8-bit data bus, as well as the shadow registers being loaded and then loaded into the counter when fully programmed. The LoadCnt signal is extended until a change in the toggle is detected and is used to load the counter.

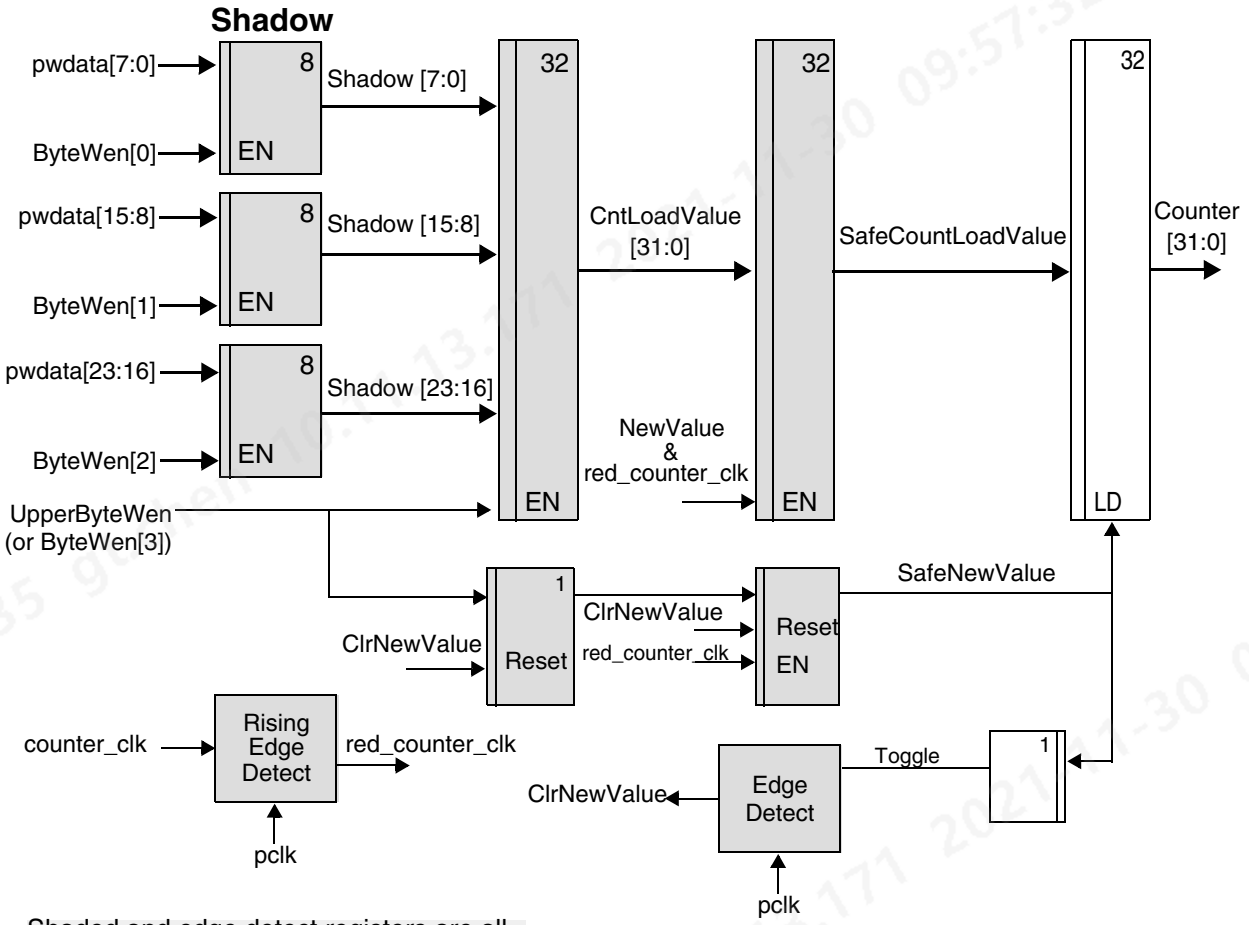
Figure 8-4 Coherent Loading – Synchronous Clocks



8.2.1.3 Asynchronous Clocks

When the clocks are asynchronous, the processor clock needs to be three-times the speed of the peripheral clock for the re-timing to operate correctly. The high pulse time of the peripheral clock needs to be greater than the period of the processor clock. The following figure shows an RTL diagram for the circuitry required to implement the coherent write when the bus and peripheral clocks are asynchronous.

Figure 8-5 Coherent Loading – Asynchronous Clocks



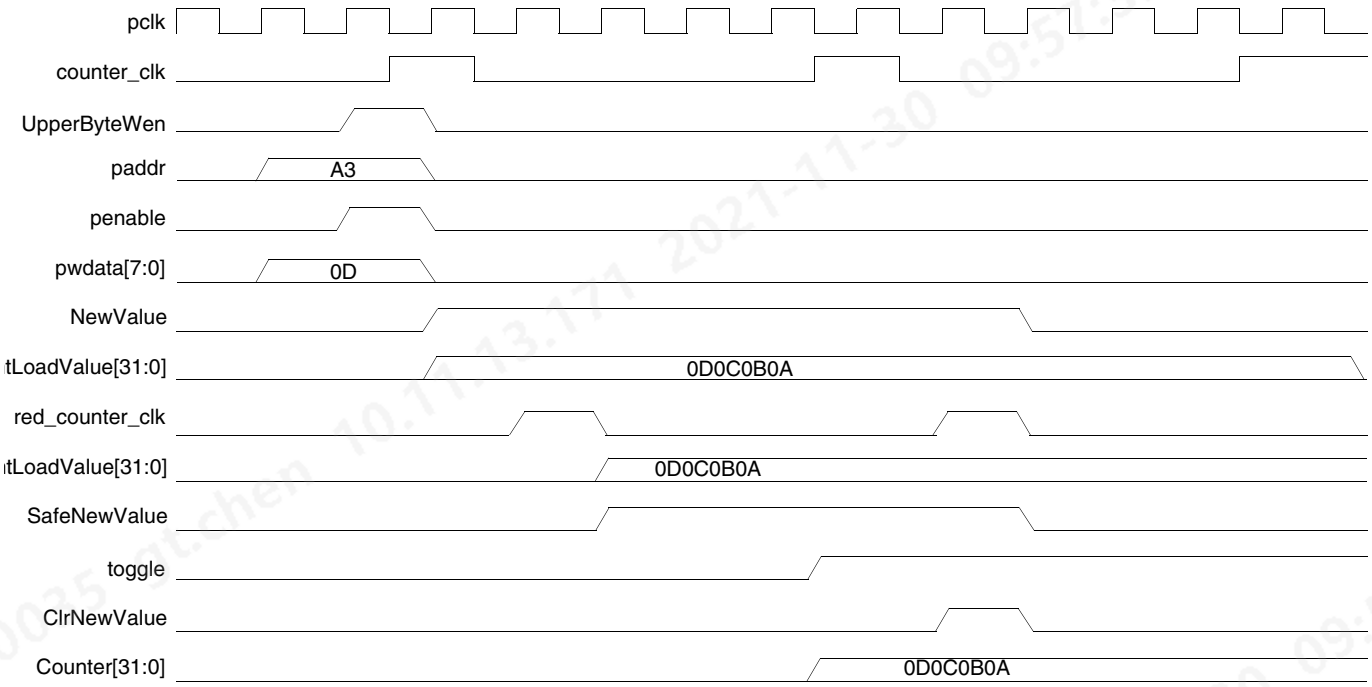
When the clocks are asynchronous, you need to transfer the contents of the register from one clock domain to another. It is not desirable to transfer the entire register through meta-stability registers, as coherency is not guaranteed with this method. The circuitry needed requires the processor clock to be used to re-time the peripheral clock. Upon a rising edge of the re-timed clock, the new value signal, NewValue, is transferred into a safe new value signal, SafeNewValue, which happens after the edge of the peripheral clock has occurred.

Every time there is a rising edge of the peripheral clock detected, the CntLoadValue is transferred into a SafeCntLoadValue. This value is used to transfer the load value across the clock domains. The SafeCntLoadValue only changes a number of bus clock cycles after the peripheral clock edge changes. A

counter running on the peripheral clock is able to use this value safely. It could be up to two peripheral clock periods before the value is loaded into the counter. Along with this loaded value, there also is a single bit transferred that is used to qualify the loading of the value into the counter.

The timing diagram depicted in the following figure does not show the shadow registers being loaded. This is identical to the loading for the other clock modes.

Figure 8-6 Coherent Loading – Asynchronous Clocks



The NewValue signal is extended until a change in the toggle is detected and is used to update the safe value. The SafeNewValue is used to load the counter at the rising edge of the peripheral clock. Each time a new value is written the toggle bit is flipped and the edge detection of the toggle is used to remove both the NewValue and the SafeNewValue.

8.2.2 Reading Coherently

For writing to registers, an upper-byte concept is proposed for solving coherency issues. For read transactions, a lower-byte concept is required. The following table provides the relationship between the register width and the bus width for the generation of the correct lower byte.

Table 8-2 Lower Byte Generation

	Lower Byte Bus Width		
Counter Register Width	8	16	32
1 - 8	NCR	NCR	NCR
9 - 16	0	NCR	NCR

Table 8-2 Lower Byte Generation

	Lower Byte Bus Width		
17 - 24	0	0	NCR
25 - 32	0	0	NCR

Depending on the bus width and the register width, there may be no need to save the upper bits because the entire register is read in one access, in which case there is no problem with coherency. When the lower byte is read, the remaining upper bytes within the counter register are transferred into a holding register. The holding register is the source for the remaining upper bytes. Users must read LSB to MSB for this solution to operate correctly. NCR means that no coherency circuitry is required, as the entire register is read with one access.

There are two cases regarding the relationship between the processor and peripheral clocks to be considered as follows:

- Identical and/or synchronous
- Asynchronous

8.2.2.1 Synchronous Clocks

When the clocks are identical and/or synchronous, the remaining unread bits (if any) need to be saved into a holding register once a read is started. The first read byte must be the lower byte provided in the previous table, which causes the other bits to be moved into the holding register, SafeCntVal, provided that the register cannot be read in one access. The upper bytes of the register are read from the holding register rather than the actual register so that the value read is coherent. This is illustrated in the following figure and in the timing diagram after it.

Figure 8-7 Coherent Registering – Synchronous Clocks

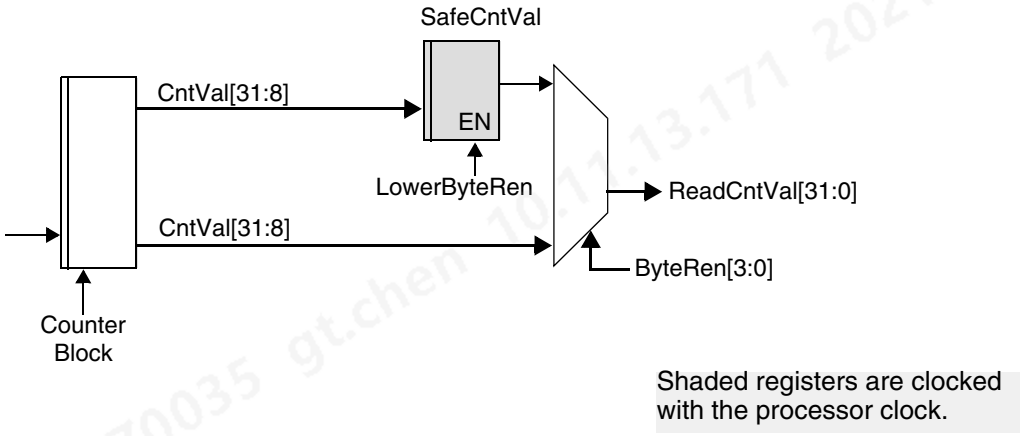
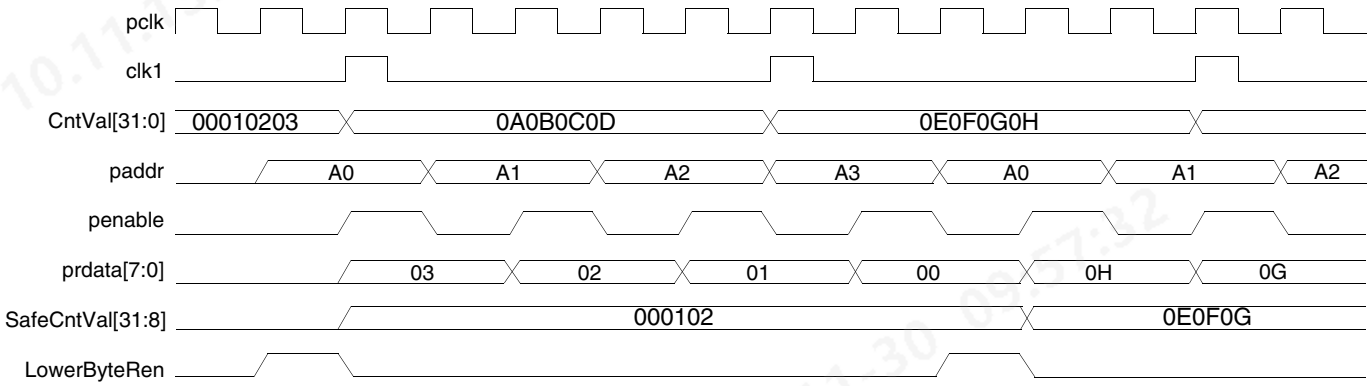


Figure 8-8 Coherent Registering – Synchronous Clocks



8.2.2.2 Asynchronous Clocks

When the clocks are asynchronous, the processor clock needs to be three times the speed of the peripheral clock for the re-timing to operate correctly. The high pulse time of the peripheral clock needs to be greater than the period of the processor clock.

To safely transfer a counter value from the counter clock domain to the bus clock domain, the counter clock signal should be transferred to the bus clock domain. When the rising edge detect of this re-timed counter clock signal is detected, it is safe to use the counter value to update a shadow register that holds the current value of the counter.

While reading the counter contents it may take multiple APB transfers to read the value.

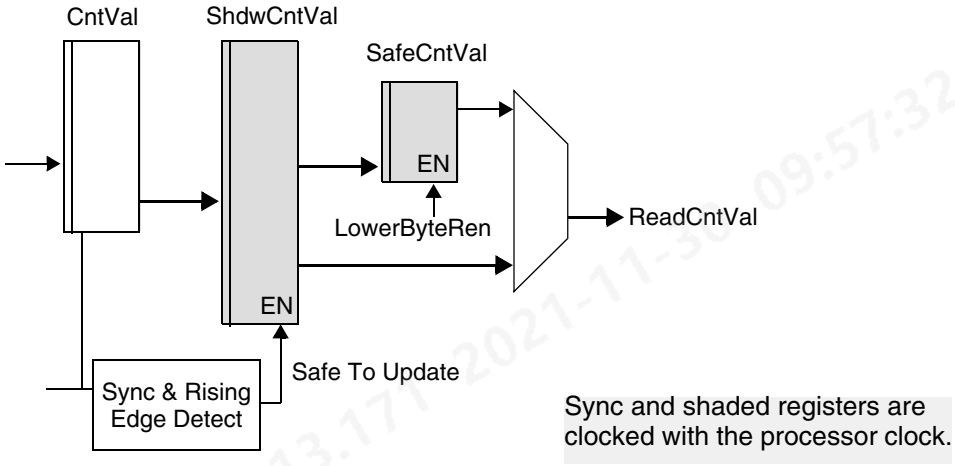
Note You must read LSB to MSB when the bus width is narrower than the counter width.

Once a read transaction has started, the value of the upper register bits need to be stored into a shadow register so that they can be read with subsequent read accesses. Storing these upper bits preserves the coherency of the value that is being read. When the processor reads the current value it actually reads the contents of the shadow register instead of the actual counter value. The holding register is read when the bus width is narrower than the counter width. When the LSB is read, the value comes from the shadow register; when the remaining bytes are read they come from the holding register. If the data bus width is wide enough to read the counter in one access, then the holding registers do not exist.

The counter clock is registered and successively pipelined to sense a rising edge on the counter clock. Having detected the rising edge, the value from the counter is known to be stable and can be transferred into the shadow register. The coherency of the counter value is maintained before it is transferred, because the value is stable.

The following figure illustrates the synchronization of the counter clock and the update of the shadow register.

Figure 8-9 Coherency and Shadow Registering – Asynchronous Clocks



8.3 Timing Exceptions

- For details on multi cycle paths, see the DW_apb_rtc.sdc file generated by the Design Compiler.
- For details on quasi-static signals on the design, refer to manual.sgdc report generated by the SpyGlass tool.

8.4 Performance

This section discusses performance and the hardware configuration parameters that affect the performance of the DW_apb_rtc.

8.4.1 Power Consumption, Frequency, Area and DFT Coverage

Table 8-3 provides information about the synthesis results (power consumption, frequency and area) and DFT coverage of the DW_apb_rtc using the industry standard 7nm technology library.

Table 8-3 Synthesis Results for DW_apb_rtc

Configuration	Operating Frequency	Gate Count	Power Consumption		TetraMax Coverage (%)		SpyGlass StuckAtCov(%)
			Static Power	Dynamic Power	StuckAtTest	Transition	
Default Configuration	pclk=100 MHz rtc_clk=100 MHz	2293	6 nW	0.015 mW	99.71	99.29	99.5
Typical Configuration - 1 APB_DATA_WIDTH = 16 SLAVE_INTERFACE_TYPE = 1 RTC_EN_MODE = 1 RTC_FREE_PCLK = 1 RTC_CNT_WIDTH = 17 RTC_CLK_EN = 1 RTC_WRAP_MODE = 1 RTC_WRAP_2_ZERO = 0 RTC_PRESCLR_EN = 0 RTC_INT_LOC = 1 RTC_INT_POL = 1	pclk=100 MHz rtc_clk=100 MHz	1208	3 nW	0.006 mW	100	100	99

Table 8-3 Synthesis Results for DW_apb_rtc (Continued)

Configuration	Operating Frequency	Gate Count	Power Consumption		TetraMax Coverage (%)		SpyGlass StuckAtCov(%)
			Static Power	Dynamic Power	StuckAtTest	Transition	
Typical Configuration - 2 APB_DATA_WIDTH = 32 SLAVE_INTERFACE_TYPE = 2 SLVERR_RESP_EN = 1 PROT_LEVEL_RST = 0x7 HC_PROT_LEVEL = 0 RTC_EN_MODE = 1 RTC_FREE_PCLK = 0 RTC_CNT_WIDTH = 32 RTC_CLK_EN = 1 RTC_WRAP_MODE = 1 RTC_WRAP_2_ZERO = 0 RTC_PRESCLR_EN = 1 RTC_PRESCLR_WIDTH = 32 RTC_PRESCLR_VAL = 32768 RTC_PRESCLR_VAL_HC = 0 RTC_INT_LOC = 1 RTC_INT_POL = 0	pclk=100 MHz rtc_clk=100 MHz	2641	7 nW	0.012 mW	100	99.98	99.8

A
Basic Core Module (BCM) Library

The Basic Core Module (BCM) Library is a library of commonly used blocks for the Synopsys DesignWare IP development. These BCMs are configurable on an instance-by-instance basis and, for the majority of BCM designs, there is an equivalent (or nearly equivalent) DesignWare Building Block (DWBB) component.

This appendix contains the following sections:

A.1 BCM Library Components

Table A-1 describes the list of BCM library components used in DW_apb_rtc.

Table A-1 BCM Library Components

BCM Module Name	BCM Description	DWBB Equivalent
DW_apb_rtc_bcm21	Single clock data bus synchronizer	DW_sync

A.2 Synchronizer Methods

This section describes the synchronizer methods (blocks of synchronizer functionality) that are used in the DW_apb_rtc to cross clock boundaries.

This section contains the following sections:

Note The DesignWare Building Blocks (DWBB) contains several synchronizer components with functionality similar to methods documented in this appendix. For more information about the DWBB synchronizer components go to:
https://www.synopsys.com/dw/buildingblock.php

A.2.1 Synchronizers Used in DW_apb_rtc

Each of the synchronizers and synchronizer sub-modules are comprised of verified DesignWare Basic Core (BCM) RTL designs. The BCM synchronizer designs are identified by the synchronizer type. The corresponding RTL files comprising the BCM synchronizers used in the DW_apb_rtc are listed and cross referenced to the synchronizer type in Table A-2. Note that certain BCM modules are contained in other BCM modules, as they are used as building blocks.

Table A-2 Synchronizers used in DW_apb_rtc

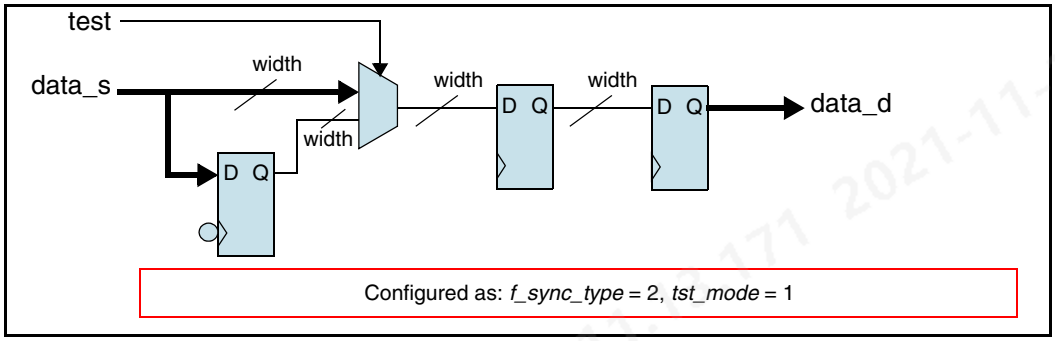
Synchronizer Module File	Synchronizer Type and Number
DW_apb_rtc_bcm21.v	Synchronizer 1: Simple Multiple Register Synchronizer

Note The BCM21 is a basic multiple register based synchronizer module used in the design. It can be replaced with equivalent technology specific synchronizer cell.

A.2.2 Synchronizer 1: Simple Double Register Synchronizer

This is a single clock data bus synchronizer for synchronizing control signals that crosses asynchronous clock boundaries. The synchronization scheme uses two stage synchronization process (Figure A-1) both using positive edge of clock.

Figure A-1 Block Diagram of Synchronizer 1 with Two Stage Synchronization (Both Positive Edge)



B
Internal Parameter Descriptions

Provides a description of the internal parameters that might be indirectly referenced in expressions in the Signals, Parameters, or Registers chapters. These parameters are not visible in the coreConsultant GUI and most of them are derived automatically from visible parameters. **You must not set any of these parameters directly.**

Some expressions might refer to TCL functions or procedures (sometimes identified as **function_of**) that coreConsultant uses to make calculations. The exact formula used by these TCL functions is not provided in this chapter. However, when you configure the core in coreConsultant, all TCL functions and parameters are evaluated completely; and the resulting values are displayed where appropriate in the coreConsultant GUI reports.

Table B-1 Internal Parameters

Parameter Name	Equals To
ASYNC	2'b01
RTC_ADDR_SLICE_LHS	{((RTC_PRESCLR_EN ==1)) ? 5:4}
RTC_VERSION_ID	32'h3230382a
SYNC	2'b11

C
Glossary

active command queue	Command queue from which a model is currently taking commands; see also command queue.
application design	Overall chip-level design into which a subsystem or subsystems are integrated.
BFM	Bus-Functional Model — A simulation model used for early hardware debug. A BFM simulates the bus cycles of a device and models device pins, as well as certain on-chip functions. See also Full-Functional Model.
big-endian	Data format in which most significant byte comes first; normal order of bytes in a word.
blocked command stream	A command stream that is blocked due to a blocking command issued to that stream; see also command stream, blocking command, and non-blocking command.
blocking command	A command that prevents a testbench from advancing to next testbench statement until this command executes in model. Blocking commands typically return data to the testbench from the model.
command channel	Manages command streams. Models with multiple command channels execute command streams independently of each other to provide full-duplex mode function.
command stream	The communication channel between the testbench and the model.
component	A generic term that can refer to any synthesizable IP or verification IP in the DesignWare Library. In the context of synthesizable IP, this is a configurable block that can be instantiated as a single entity (VHDL) or module (Verilog) in a design.
configuration	The act of specifying parameters for a core prior to synthesis; can also be used in the context of VIP.
configuration intent	Range of values allowed for each parameter associated with a reusable core.
cycle command	A command that executes and causes HDL simulation time to advance.

decoder	Software or hardware subsystem that translates from and “encoded” format back to standard format.
design context	Aspects of a component or subsystem target environment that affect the synthesis of the component or subsystem.
design creation	The process of capturing a design as parameterized RTL.
DesignWare Library	A collection of synthesizable IP and verification IP components that is authorized by a single DesignWare license. Products include SmartModels, VMT model suites, DesignWare Memory Models, Building Block IP, and the DesignWare Synthesizable Components.
dual role device	Device having the capabilities of function and host (limited).
endian	Ordering of bytes in a multi-byte word; see also little-endian and big-endian.
Full-Functional Mode	A simulation model that describes the complete range of device behavior, including code execution. See also BFM.
GPIO	General Purpose Input Output.
GTECH	A generic technology view used for RTL simulation of encrypted source code by non-Synopsys simulators.
hard IP	Non-synthesizable implementation IP.
HDL	Hardware Description Language – examples include Verilog and VHDL.
IIP	Implementation Intellectual Property — A generic term for synthesizable HDL and non-synthesizable “hard” IP in all of its forms (coreKit, component, core, MacroCell, and so on).
implementation view	The RTL for a core. You can simulate, synthesize, and implement this view of a core in a real chip.
instantiate	The act of placing a core or model into a design.
interface	Set of ports and parameters that defines a connection point to a component.
IP	Intellectual property — A term that encompasses simulation models and synthesizable blocks of HDL code.
little-endian	Data format in which the least-significant byte comes first.
master	Device or model that initiates and controls another device or peripheral.
model	A Verification IP component or a Design View of a core.
monitor	A device or model that gathers performance statistics of a system.
non-blocking command	A testbench command that advances to the next testbench statement without waiting for the command to complete.

peripheral	Generally refers to a small core that has a bus connection, specifically an APB interface.
RTL	Register Transfer Level. A higher level of abstraction that implies a certain gate-level structure. Synthesis of RTL code yields a gate-level design.
SDRAM	Synchronous Dynamic Random Access Memory; high-speed DRAM adds a separate clock signal to control signals.
SDRAM controller	A memory controller with specific connections for SDRAMs.
slave	Device or model that is controlled by and responds to a master.
SoC	System on a chip.
soft IP	Any implementation IP that is configurable. Generally referred to as synthesizable IP.
static controller	Memory controller with specific connections for Static memories such as asynchronous SRAMs, Flash memory, and ROMs.
synthesis intent	Attributes that a core developer applies to a top-level design, ports, and core.
synthesizable IP	A type of Implementation IP that can be mapped to a target technology through synthesis. Sometimes referred to as Soft IP.
technology-independent	Design that allows the technology (that is, the library that implements the gate and via widths for gates) to be specified later during synthesis.
Testsuite Regression Environment (TRE)	A collection of files for stand-alone verification of the configured component. The files, tests, and functionality vary from component to component.
VIP	Verification Intellectual Property — A generic term for a simulation model in any form, including a Design View.
wrap, wrapper	Code, usually VHDL or Verilog, that surrounds a design or model, allowing easier interfacing. Usually requires an extra, sometimes automated, step to create the wrapper.
zero-cycle command	A command that executes without HDL simulation time advancing.