SYNOPSYS®

DesignWare® DW_apb_timers

Databook

DW_apb_timers – Product Code

2.13a
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 table shows the revision history for the databook from release to release. This is being tracked from version 2.02d onward.

Version	Date	Description
2.13a	December 2020	**Added:** ■ "Clocks and Resets" on page 31 ■ Chapter 7, "Verification" ■ "BCM Library Components" on page 99 ■ "Timing Exceptions" on page 93 **Updated:** ■ Version number changed for 2020.12a release ■ "Controlling Clock Boundaries and Metastability" on page 24 ■ "Performance" on page 94 ■ "Parameter Descriptions" on page 35, "Register Descriptions" on page 51, "Signal Descriptions", and "Internal Parameter Descriptions" are auto extracted with change bars from the RTL **Renamed** ■ Synchronizer Methods to Appendix A, "Basic Core Module (BCM) Library" **Removed:** ■ Index chapter

(Continued)

Version	Date	Description
2.12a	July 2018	**Added:** ■ Support for configurable Synchronization Depth through the following parameters: TIM_SYNC_DEPTH_1, TIM_SYNC_DEPTH_2, TIM_SYNC_DEPTH_3, TIM_SYNC_DEPTH_4, TIM_SYNC_DEPTH_5, TIM_SYNC_DEPTH_6, TIM_SYNC_DEPTH_7 and TIM_SYNC_DEPTH_8 **Updated:** ■ Version number changed for 2018.07a release ■ "Performance" on page 94 ■ "Parameter Descriptions" on page 35, "Register Descriptions" on page 51, "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.
2.11a	October 2016	■ Version number change to 2016.10a ■ "Parameter Descriptions" on page 35 and "Register Descriptions" on page 51 autoextracted 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 ■ Replaced Figure 2-2 and Figure 2-3 to remove references to Leda ■ Added "Running VCS XPROP Analyzer" ■ Moved Appendix B, "Internal Parameter Descriptions" to Appendix ■ Added an entry for the xprop directory in Table 2-1 and Table 2-4. ■ Added "Pulse Width Modulation with 0% and 100% Duty Cycle" ■ Added "APB Interface"
2.10a	June 2015	■ Modified default value for TimerNCurrentValue field of the Timer N register ■ Included section "Running SpyGlass® Lint and SpyGlass® CDC" ■ Included section "Running SpyGlass on Generated Code with coreAssembler" ■ Chapter 4, "Signal Descriptions" auto-extracted from the RTL ■ Added Chapter B, "Internal Parameter Descriptions" ■ Added Appendix A, "Basic Core Module (BCM) Library"

(Continued)

Version	Date	Description
2.09a	June 2014	■ Version change for 2014.06a release ■ Updated the section 3.3.6 Controlling Clock Boundaries and Metastability ■ Added: - A new parameter INTR_SYNC2PCLK for interrupt synchronization - “Performance” section in the “Integration Considerations” chapter ■ Corrected External Input/Output Delay in Signals chapter
2.08b	May 2013	■ Version change for 2013.05a release ■ Updated the template
2.08a	September 2012	Added the product code on the cover and in Table 1-1
2.08a	June 2012	Version change for 2012.06a release
2.06c	March 2012	Version change for 2012.03a release
2.06b	November 2011	Version change for 2011.11a release
2.06a	October 2011	Version change for 2011.10a release
2.05a	June 2011	■ Updated system diagram in Figure 1-1 ■ Enhanced “Related Documents” section in Preface
2.05a	September 2010	Corrected names of include files and vcs command used for simulation
2.03a	December 2009	Updated databook to new template for consistency with other IIP/VIP/PHY databooks.
2.03a	July 2009	■ Corrected and enhanced free-running and user-defined modes ■ Corrected values for setting interrupt mask as either masked or not masked in “DW_apb_timers Usage Flow” section
2.03a	May 2009	Removed references to QuickStarts, as they are no longer supported
2.03a	March 2009	Corrected TimersIntStatus register illustration
2.03a	October 2008	Version change for 2008.10a release
2.02e	June 2008	Version change for 2008.06a release
2.02d	January 2008	■ Updated to revised installation guide and consolidated release notes ■ Changed references of “Designware AMBA” to simply “DesignWare”
2.02d	September 2007	Corrected red circles in Figure 1
2.02d	June 2007	Version change for 2007.06a release

Preface

This databook provides information that you need to interface the DesignWare® APB Timers peripheral, referred to as the DW_apb_timers throughout the remainder of this databook. It is a component of the DesignWare Advanced Peripheral Bus (DW_apb) and conforms to the *AMBA Specification, Revision 2.0* from Arm®.

The information in this databook includes a functional description, signal and parameter descriptions, programmable register descriptions and a memory map. The databook also provides step-by-step information about using the DW_apb_timers in the coreConsultant flow. It also includes an overview of the component testbench and a description of the tests that are run to verify the coreKit. The databook also contains several appendices that provide additional information to help you integrate the component into your higher-level design.

Databook 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_timers.
- Chapter 3, "Parameter Descriptions" identifies the configurable parameters supported by the DW_apb_timers.
- Chapter 4, "Signal Descriptions" provides a list and description of the DW_apb_timers signals.
- Chapter 5, "Register Descriptions" describes the programmable registers of the DW_apb_timers.
- Chapter 6, "Programming Considerations" provides information needed to program the configured DW_apb_timers.
- Chapter 7, "Verification" provides information on verifying the configured DW_apb_timers.
- Chapter 8, "Integration Considerations" includes information you need to integrate the configured DW_apb_timers 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_timers.
- Chapter 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, see 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

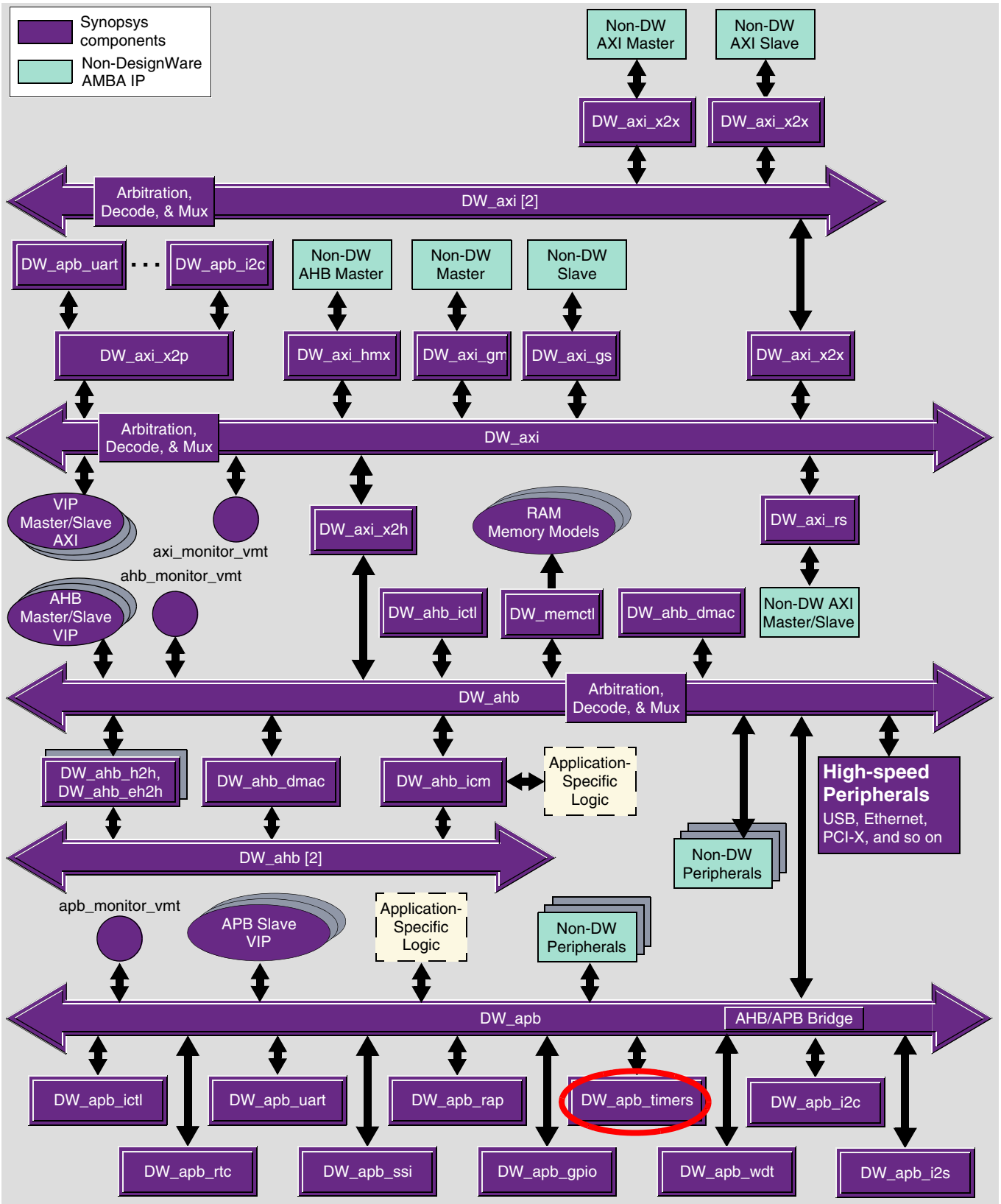
The DW_apb_timers is a programmable timers peripheral. This component is an AMBA 2.0-compliant Advanced Peripheral Bus (APB) slave device and is part of the family of DesignWare Synthesizable Components.

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_timers in a Complete System



You can connect, configure, synthesize, and verify the DW_apb_timers 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_timers component, you might prefer to use coreConsultant, documentation for which is available in the *coreConsultant User Guide*.

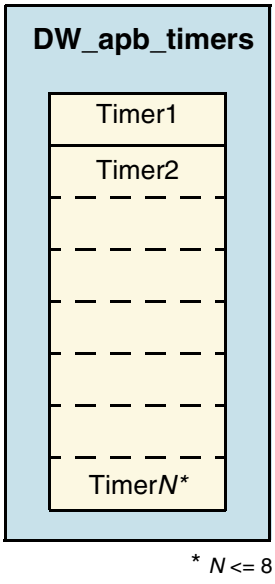
1.2 General Product Description

The Synopsys DW_apb_timers is a component of the DesignWare Advanced Peripheral Bus (DW_apb).

1.2.1 DW_apb_timers Block Diagram

Figure 1-2 shows the block diagram of the DW_apb_timers.

Figure 1-2 DW_apb_timers Block Diagram



1.3 Features

DW_apb_timers has the following features:

- APB interface supports APB2, APB3, and APB4
- Up to eight programmable timers
- Configurable timer width: 8 to 32 bits
- Support for two operation modes: free-running and user-defined count
- Support for independent clocking of timers
- Configurable polarity for each individual interrupt
- Configurable option for a single or combined interrupt output flag

- Configurable option to have read/write coherency registers for each timer
- Configurable option to include timer toggle output, which toggles whenever timer counter reloads
- Configurable option to enable programmable pulse-width modulation of timer toggle outputs
- Configurable option to include pulse width modulation of timer toggle output with 0% and 100% duty cycle.

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

1.4 Standards Compliance

The DW_apb_timers 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_timers includes an extensive verification environment, detailed in "Verification" on page 77.

1.6 Licenses

Before you begin using the DW_apb_timers, you must have a valid license. For more information, see "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_timers 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, see *Guide to coreTools Documentation*.

For more information about configuring, synthesizing, and verifying just your DW_apb_timers component, see "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_timers component within a DesignWare subsystem using coreAssembler, see "Overview of the coreAssembler Configuration and Integration Process" *DesignWare Synthesizable Components for AMBA 2 User Guide*.

2 Functional Description

This chapter describes the following sections how you can use the DW_apb_timers.

2.1 Timer Operation

The DW_apb_timers component implements up to eight identical but separately-programmable timers, which are accessed through a single AMBA APB interface.

Timers count down from a programmed value and generate an interrupt when the count reaches zero. You can use the TIM_INTR_IO parameter (Single Combined Interrupt) to create a single combined interrupt, which is active whenever any of the individual timer interrupts is active.

Each timer has an independent clock input, timer_*N*_clk (where *N* is in the range 1 to 8), that you can connect to pclk (also known as the system clock or the APB clock) or to an external clock source. You can configure the width of a timer from 8 to 32 bits using the TIMER_WIDTH_*N* parameter (Width of Timer *N*), where *N* is in the range 1 to NUM_TIMERS, the number of instantiated timers.

The initial value for each timer – that is, the value from which it counts down – is loaded into the timer using the appropriate load count register (TimerNLoadCount). Two events can cause a timer to load the initial count from its Timer*N*LoadCount register:

- Timer is enabled after being reset or disabled
- Timer counts down to 0

All interrupt status registers and end-of-interrupt registers can be accessed at any time.

2.2 DW_apb_timers Usage Flow

The procedure illustrated in Figure 2-1 is a basic flow to follow when programming the DW_apb_timers. More advanced functions are discussed later in this chapter.

1. Initialize the timer through the TimerNControlReg register (where *N* is in the range 1 to 8):
   a. Disable the timer by writing a “0” to the timer enable bit (bit 0); accordingly, the timer_en output signal is de-asserted.

Note Before writing to a Timer*N*LoadCount register, you *must* disable the timer by writing a “0” to the timer enable bit of Timer*N*ControlReg in order to avoid potential synchronization problems.

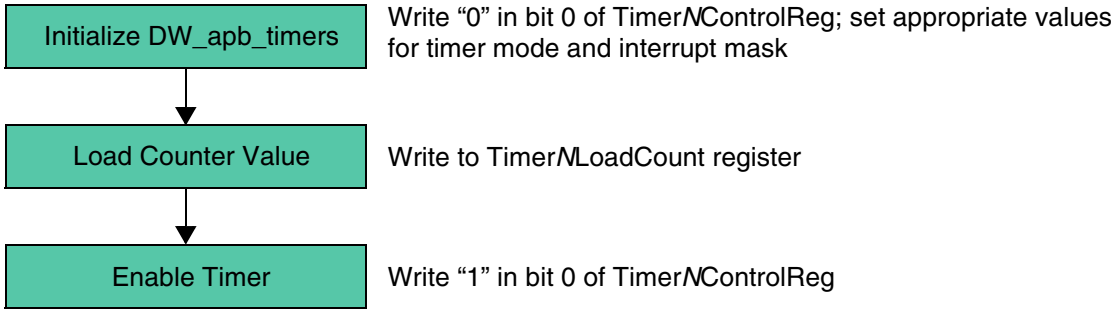
b. Program the timer mode—user-defined or free-running—by writing a "1" or "0," respectively, to the timer mode bit (bit 1).

c. Set the interrupt mask as either masked or not masked by writing a "1" or "0," respectively, to the timer interrupt mask bit (bit 2).

2. Load the timer counter value into the TimerNLoadCount register (where *N* is in the range 1 to 8).

3. Enable the timer by writing a "1" to bit 0 of TimerNControlReg.

Figure 2-1 DW_apb_timers Usage Flow



As an example, suppose you have only timer1, and the timer_1_clk signal is asynchronous to pclk. When you disable the timer enable bit (bit 0 of Timer1ControlReg), the timer_en output signal is de-asserted and, accordingly, timer_1_clk should stop. Then when you enable the timer, the timer_en signal is asserted and timer_1_clk should start running. This is not necessary, however, as long as the timer_1_clk is synchronous to pclk; in this case, you can choose to directly tie timer_1_clk to pclk.

It is also not necessary to stop the timer_1_clk if the TIM_NEWMODE parameter is set to 1 (True). For more information on this parameter and on synchronization and metastability issues, see "Controlling Clock Boundaries and Metastability" on page 24.

2.3 DW_apb_timers Configuration

The following sections tell you how to set up the DW_apb_timers.

2.3.1 Choosing the Number of Timers

You can have up to eight timers in your design. There are several registers with names specific to the number of timers that you choose (where *N* is from 1 to 8):

- TimerNLoadCount – Timer*N* load count register
- TimerNLoadCount2 (optional) – Timer*N* load count register for programming width of HIGH period of timer_*N*_toggle output
- TimerNCurrentValue – Timer*N* current value register
- TimerNControlReg – Timer*N* control register
- TimerNEOI – Timer*N* end-of-interrupt register
- TimerNIntStatus – Timer*N* interrupt status register

Thus you have five individual registers for each of the timers in your design. All other registers control their respective functions for all active timers, rather than for individual timers.

2.3.2 Enabling and Disabling a Timer

You can use bit 0 of the TimerNControlReg, where *N* is in the range 1 to 8, to either enable or disable a timer.

2.3.2.1 Enabling a Timer

To enable a timer, write "1" to bit 0 of its Timer*N*ControlReg register.

2.3.2.2 Disabling a Timer

To disable a timer, write "0" to bit 0 of its Timer*N*ControlReg register.

When a timer is enabled and running, its counter decrements on each rising edge of its clock signal, timer_N_clk. When a timer transitions from disabled to enabled, the current value of its TimerNLoadCount register is loaded into the timer counter on the next rising edge of timer_*N*_clk.

When the timer enable bit is de-asserted and the timer stops running, the timer counter and any associated registers in the timer clock domain, such as the toggle register, are asynchronously reset.

When the timer enable bit is asserted, then a rising edge on the timer_en signal is used to load the initial value into the timer counter. "0" is always read back when the timer is not enabled; otherwise, the current value of the timer (TimerNCurrentValue register) is read back.

2.3.3 Configuring the Width of a Timer

You can configure the width of a timer through the TIMER_WIDTH_*N* parameter; each timer can be from 8 bits to 32 bits. You do this for each timer through the Timer *N* Configuration section of the Specify Configuration activity in coreConsultant. You should bear in mind that, if the width of the APB bus is smaller than the width of a timer—the APB data bus can be 8, 16, or 32 bits wide—there has to be multiple APB write accesses to load the counter.

2.3.4 Loading a Timer Countdown Value

When a timer counter is enabled after being reset or disabled, the count value is loaded from the TimerNLoadCount register; this occurs in both free-running and user-defined count modes.

When a timer counts down to 0, it loads one of two values, depending on the timer operating mode:

- User-defined count mode – Timer loads the current value of the TimerNLoadCount register. Use this mode if you want a fixed, timed interrupt. Designate this mode by writing a "1" to bit 1 of TimerNControlReg.

If you set the TIM_NEWMODE parameter to 1, the value that is loaded to the timer—when it counts down to 0—alternates between the value of the Timer*N*LoadCount register and the Timer*N*LoadCount2 register. For more details, see "Pulse Width Modulation of Toggle Outputs" on page 28.

- Free-running mode – Timer loads the maximum value, which is dependent on the timer width; that is, the Timer*N*LoadCount register is comprised of $2^{TIMER\_WIDTH\_N - 1}$ bits, all of which are loaded with 1s. The timer counter wrapping to its maximum value allows time to reprogram or disable the timer before another interrupt occurs. Use this mode if you want a single timed interrupt. Designate this mode by writing a "0" to bit 1 of TimerNControlReg.

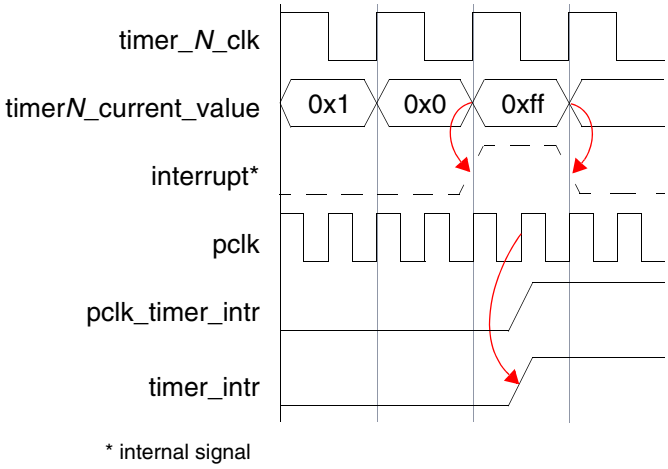
2.3.5 Working with Interrupts

The TimerNIntStatus and TimerNEOI registers handle interrupts in order to ensure safe operation of the interrupt clearing. Because of the hclk/pclk ratio, if pclk can perform a write to clear an interrupt, it could continue with another transfer on the bus without knowing whether the write has occurred. Therefore, it is much safer to clear the interrupt by a read operation.

To detect and service an interrupt, the system clock must be active if the TIM_NEWMODE parameter is set to 0 (False). The timer_en output bus from this block is used to activate the necessary timer clocks and to ensure that the component is supplied with an active system clock while timers are running.

In both the free-running and user-defined count modes of operation, a timer generates an internal interrupt signal when its count changes from 0 to its maximum count value, as shown in Figure 2-2.

Figure 2-2 Timer Interrupt Set – No Metastability Registers and TIM_NEWMODE = 0

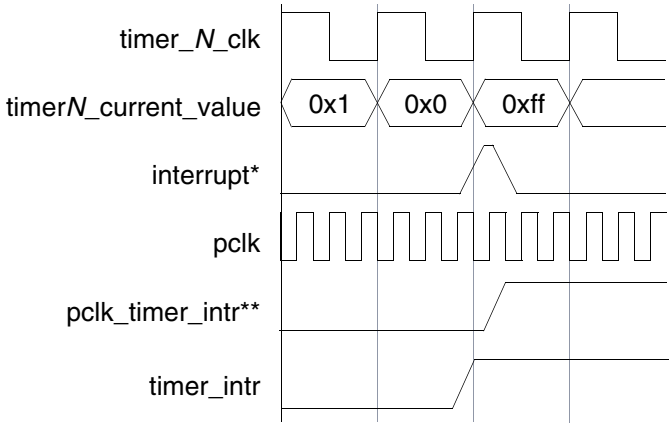


The setting of the internal interrupt signal occurs synchronously to the timer clock domain. This internal interrupt signal is transferred to the pclk domain in order to set the timer interrupt. The internal interrupt signal and the timer interrupt are not generated if the timer is disabled; if the timer interrupt is set, then it is cleared when the timer is disabled.

When the TIM_NEWMODE = 1 and INTR_SYNC2PCLK = 0, interrupt detection can occur even when the system clock is disabled. The timer_intr interrupt output signal is asserted when the interrupt is detected in the timer clock domain.

As shown in Figure 2-3, the timer_intr signal remains asserted until pclk is re-started and the TimerNEOI or TimerEOI registers are read to clear the interrupt, or the timer is disabled.

Figure 2-3 Timer Interrupt Set - No Metastability Registers and TIM_NEWMODE = 1



*interrupt – internal interrupt generated in timer clock domain
**pclk_timer_inter – edge-detected interrupt in pclk domain

If the system bus (AHB) can perform a write to clear a timer interrupt, it could continue with another transfer on the bus without knowing whether the write has occurred because of the hclk/pclk ratio. Therefore, it is much safer to clear the timer interrupt by a read operation.

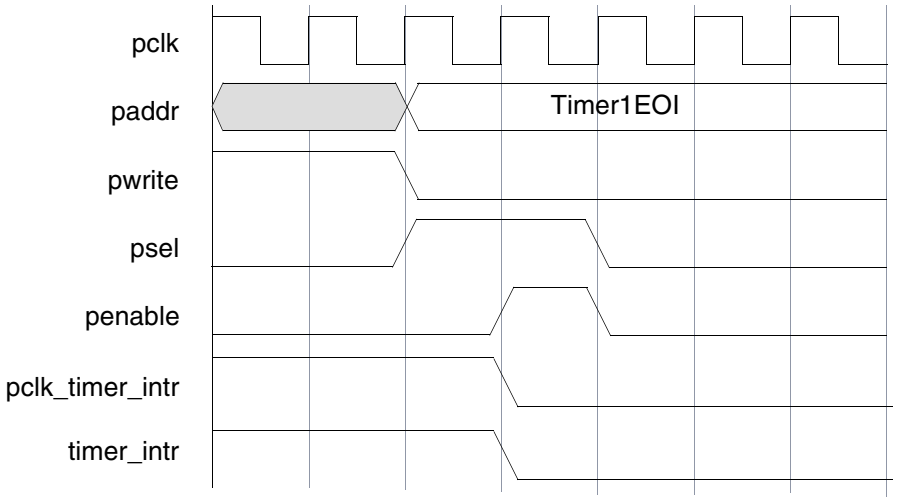
2.3.5.1 Clearing Interrupts

Provided the timer is enabled, its interrupt remains asserted until it is cleared by reading one of two end-of-interrupt registers (TimerNEOI or TimersEOI, the individual and global end-of-interrupt registers, respectively). When the timer is disabled, the timer interrupt is cleared. You can clear an individual timer interrupt by reading its TimerNEOI register. You can clear all active timer interrupts at once by reading the global TimersEOI register or by disabling the timer.

When reading the TimersEOI register, timer interrupts are cleared at the rising edge of pclk and when penable is low. If an end-of-interrupt register is read during the time when the internal interrupt signal is high, the timer interrupt is set. This occurs because setting timer interrupts takes precedence over clearing them.

Figure 2-4 shows the timer interrupt timing when cleared by the TimersEOI register.

Figure 2-4 Clearing an Interrupt From DW_apb_timers



2.3.5.2 Checking Interrupt Status

You can query the interrupt status of an individual timer without clearing its interrupt by reading the TimerNIntStatus register. You can query the interrupt status of all timers without clearing the interrupts by reading the global TimersIntStatus register.

2.3.5.3 Masking Interrupts

Each individual timer interrupt can be masked using its TimerNControlReg register. To mask an interrupt, you write a "1" to bit 2 of Timer*N*ControlReg.

If all individual timer interrupts are masked, then the combined interrupt is also masked.

2.3.5.4 Setting Interrupt Polarity

The polarity of the generated timer interrupts can be configured to be either active-high or active-low using the TIM_INTRPT_PLRITY parameter (Interrupt Polarity). In addition to an interrupt output signal for each timer, there is also a single, global interrupt flag, timer_intr_flag, that is asserted if any timer asserts its interrupt. This global interrupt flag shares the same polarity characteristic with the other generated interrupts; thus, multiple interrupt service schemes can be supported.

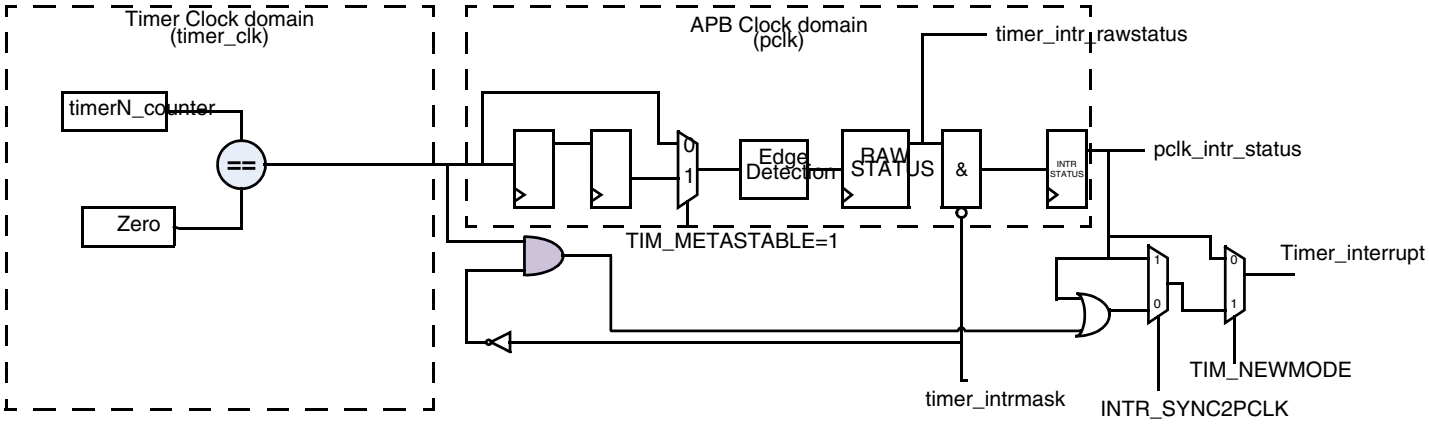
2.3.6 Controlling Clock Boundaries and Metastability

If a timer clock is asynchronous to pclk, you must ensure that the clocks are stopped whenever the timer is disabled. This restriction does not apply when the TIM_NEWMODE parameter is set to 1. If TIM_NEWMODE is enabled, the timer_en signal is synchronized from the pclk domain to the timer clock domain, which eliminates any risk of metastability if the timer_*N*_clk is kept running while the timer is disabled. Therefore, with TIM_NEWMODE set, the timer_*N*_clk can be free-running and does not have to be stopped whenever the timer is to be disabled.

The timer_*N*_resetn signal resets all of the registers in the timer_*N*_clk domain, including the timer counter. For each timer, there are several factors that internally affect the boundaries between the pclk and timer clock domains.

Each timer generates an internal interrupt signal that is synchronized to the pclk domain. Figure 2-5 shows an internal interrupt signal affecting the clock boundaries between the two clock domains.

Figure 2-5 Boundary Between Clock Domains



The internal interrupt signal is generated in the timer clock domain when the timer counter rolls over to its maximum value.

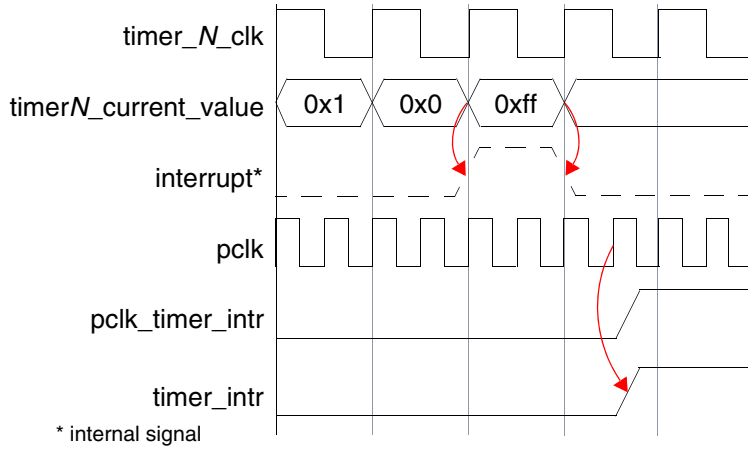
The timer interrupt (timer_intr) is asserted based on the value of the TIM_NEWMODE and INTR_SYNC2PCLK parameters as follows:

- When TIM_NEWMODE is set to 0 (False), the timer interrupt is asserted when the internal interrupt signal is edge-detected in the pclk domain.
- When TIM_NEWMODE is set to 1 (True) and INTR_SYNC2PCLK is set to 1 (True), the timer interrupt is asserted when the internal interrupt signal is edge-detected in the pclk domain.
- When TIM_NEWMODE is set to 1 (True) and INTR_SYNC2PCLK is set to 0 (False), the timer interrupt is asserted along with the internal interrupt signal generated in the timer clock domain when the timer counter rolls over to its maximum value.
- When TIM_NEWMODE is set to 1 (True) and INTR_SYNC2PCLK is set to 0 (False), the internal interrupt remains set until it is transferred to the pclk domain and edge detected there. Then it is cleared automatically, leaving the pclk interrupt set. The pclk domain interrupt is cleared when software reads the TimerNEOI registers. This mode allows the timer interrupt to be detected, even when pclk is disabled.

  In the case when pclk is stopped and INTR_SYNC2PCLK is set to 0 (False), the timer interrupt remains asserted until pclk is restarted and the interrupt is serviced, or the timer is disabled or reset.

The internal interrupt signal is edge-detected in the pclk domain in order to set the timer interrupt, as illustrated in Figure 2-6.

Figure 2-6 Timer Interrupt Set – Metastability Registers Included and TIM_NEWMODE = 0



A timer_en signal is edge-detected in the timer clock domain. When it transitions from 0 to 1, the timer counter is loaded with the value of the Timer*N*LoadCount register. This guarantees that the timer is in a known state when enabled. If you disable a timer counter by writing a "0" to bit 0 of its Timer*N*ControlReg register, it also synchronously disables interrupts for that timer counter in the pclk domain. This prevents spurious interrupts because of mis-sampling in the timer clock domain.

Neither the timer mode bit of Timer*N*ControlReg nor the Timer*N*LoadCount register are synchronized between the pclk domain and the timer clock domain. Because of this, it is important that you disable a timer before programming its mode or load count value so that any information on these signals is always communicated to the timer while it is inactive. Thus you must ensure that these signals are stable whenever a timer is enabled. In practice, this means that you must follow at least this basic procedure:

1. First use the Timer*N*ControlReg to disable the timer, program its timer mode, and then set the interrupt mask.
2. Next, load the timer counter value into the Timer*N*LoadCount register.
3. Finally, enable the timer through Timer*N*ControlReg.

For more details on this procedure, see "DW_apb_timers Usage Flow" on page 19.

When you connect a timer_*N*_clk input to a clock source that is independent of pclk, metastability registers must be instantiated by setting the TIM_METASTABLE_*N* parameter (Metastability support for interrupt from Timer *N*) to "Present" (where *N* is in the range 1 to 8). By instantiating the metastability registers, an extra two pclk periods of latency occurs between when a timer maximum count is reached and when its interrupt goes active. To see the difference, compare the timing in Figure 2-2 (no metastability registers) to that in Figure 2-6 (metastability registers included).

The DW_apb_timers component supports timer clocks that are up to four times the frequency of pclk. If you connect a timer_*N*_clk to a clock source that is faster than pclk, you must extend the width of the internal interrupt signal to allow adequate time for it to be sampled in the pclk domain.

You can extend the width of the interrupt signal up to three timer_*N*_clk clock periods by setting the TIM_PULSE_EXTD_*N* parameter (Number of clock cycles by which to extend interrupt, where *N* is in the range 1 to 8) to a non-zero value.

Figure 2-7 illustrates an example of related pclk and timer_*N*_clk, where the frequency of timer_*N*_clk is two times that of pclk. To accommodate this, the TIM_PULSE_EXTD_*N* parameter is set to 1 in order to extend the internal interrupt signal by one timer_*N*_clk clock period.

Figure 2-7 Timer Interrupt Set – Pulse Extend One Cycle, No Metastability

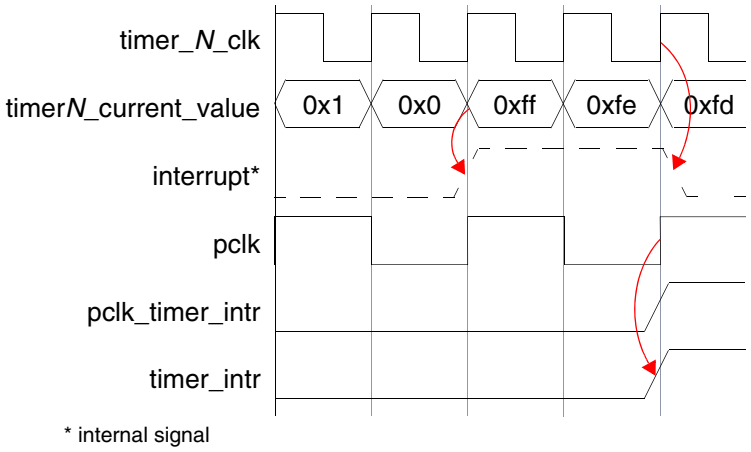
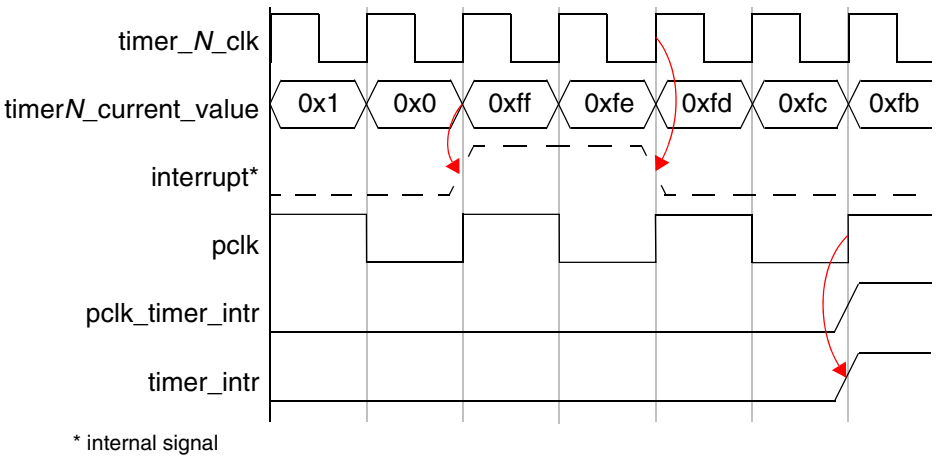


Figure 2-8 illustrates an example where metastability registers are required because pclk is independent of timer_*N*_clk, and 1 < frequency of timer_*N*_clk < 2 times that of pclk. To accommodate this, the TIM_PULSE_EXTD_*N* parameter is set to 1 in order to extend the internal interrupt signal by one timer_*N*_clk clock period.

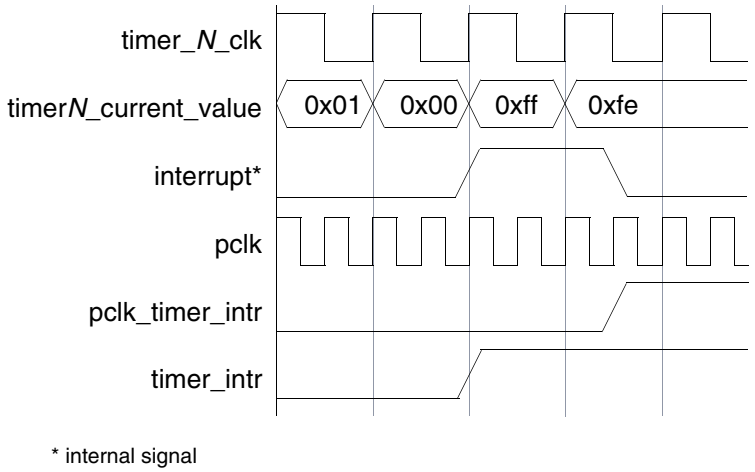
Figure 2-8 Timer Interrupt Set – Pulse Extend One Cycle, With Metastability



Note When the TIM_NEWMODE parameter is set to 1, it is not required to extend the width of the internal interrupt signal, since it remains asserted until the interrupt is detected in the pclk domain. Therefore, when TIM_NEWMODE is set to 1, the TIM_PULSE_EXTD_N parameter is disabled.

Figure 2-9 illustrates an example where metastability registers are included when **TIM_NEWMODE = 1**.

Figure 2-9 Timer Interrupt Set – Metastability Registers Included and TIM_NEWMODE=1



2.3.7 Generating Toggled Outputs

You can configure a timer through the TIMER_HAS_TOGGLE_*N* parameter (Include toggle output for timer # on I/F, see "Parameter Descriptions" on page 35) in order to generate an output that toggles whenever the timer counter reaches 0. You do this for each timer through the Timer *N* Configuration section of the Specify Configuration activity in coreConsultant.

2.3.7.1 Pulse Width Modulation of Toggle Outputs

The TIM_NEWMODE parameter allows the toggle output from each of the timers—that is, timer_N_toggle—to be pulse-width modulated. If TIM_NEWMODE is set to 1 and register bit TimerNControlReg[4] (TIMER_PWM bit) is set to 1, the HIGH and LOW periods of the toggle outputs can be controlled separately by programming the Timer*N*LoadCount2 and Timer*N*LoadCount registers.

The pulse widths of the toggle outputs are controlled as follows:

- Width of timer_*N*_toggle HIGH period = (Timer*N*LoadCount2 + 1) * timer_*N*_clk clock period
- Width of timer_*N*_toggle LOW period = (Timer*N*LoadCount + 1) * timer_*N*_clk clock period

If TIM_NEWMODE is set to 0 or the Timer*N*ControlReg[4] (TIMER_PWM bit) is set to 0, the HIGH and LOW periods of the timer_*N*_toggle outputs are the same and equal to (TimerNLoadCount + 1) * timer_*N*_clk clock period.

Note TIM_NEWMODE is enabled only when APB Data Bus Width = 32.

2.3.7.2 Pulse Width Modulation with 0% and 100% Duty Cycle

DW_apb_timers supports the programming for 0% and 100% duty cycle pulse width modulation of toggle outputs (timer_N_toggle) through the TimerNLoadCount and TimerNLoadCount2 registers, when 0% and 100% duty cycle mode is enabled. You can enable the duty cycle mode either by setting the TimerNControlReg [4] register or by configuring the TIMER_0N100_PWM_HC_EN parameter.

Note The TimerNLoadCount register defines the LOW period and the TimerNLoadCount2 register defines the HIGH period values.

The definition of the duty cycles (with TimerNLoadCount and TimerNLoadCount2) is as follows (note that the high period signifies the duty cycle number):

- 0% duty cycle – Continuous Low and no high
  - TimerNLoadCount = Do not care
  - TimerNLoadCount2 = 0
- 100% duty cycle – No low period and continuous high
  - TimerNLoadCount = 0
  - TimerNLoadCount2 = Do not care
- Other duty cycle – When 0% and 100% duty cycle mode is enabled (with timer PWM mode and the user-defined count mode is enabled), the definition of the toggle high and low period changes as follows for a duty cycle other than 0% or 100%:
  - Width of timer_N_toggle HIGH period = TimerNLoadCount2 * timer_N_clk clock period
  - Width of timer_N_toggle LOW period = TimerNLoadCount * timer_N_clk clock period

Note The above definition is applicable only if the 0% and 100% duty cycle mode is enabled along with the timer Pulse Width Modulation (PWM) mode and the user-defined count mode. If any of these modes are not enabled, that is, if the PWM mode or user-defined mode is not enabled, then the new definition of the High and Low period is not applicable. The previous definition of the High and Low period is applicable.

Table 2-1 provides information on the relation between Duty cycle, TimerNLoadCount, and TimerNLoadCount2 values, considering that the maximum value is 100.

Table 2-1 Duty Cycle, TimerNLoadCount, TimerNLoadCount2 Relationship Table

Duty Cycle (%)	TimerNLoadCount	TimerNLoadCount2
0	X	0
1	99	1
2	98	2
3	97	3
4	96	4
....	....	....
96	4	96
97	3	97

Table 2-1 Duty Cycle, TimerNLoadCount, TimerNLoadCount2 Relationship Table

Duty Cycle (%)	TimerNLoadCount	TimerNLoadCount2
98	2	98
99	1	99
100	0	100

Table 2-2 provides an example of the programming TimerNLoadCount and TimerNLoadCount2 registers for a timer, with timer width set to 8 bits.

Table 2-2 Duty Cycle, TimerNLoadCount, TimerNLoadCount2 Relationship Table (8-Bit Timer)

Duty Cycle (%)	TimerNLoadCount	TimerNLoadCount2
0	X	0
1	FC	02
2	F9	05
3	F7	07
4	F4	0A
....	….	….
96	0A	F4
97	07	F7
98	05	F9
99	02	FC
100	0	X

When TimerNLoadCount=0 and TimerNLoadCount2=0, DW_apb_timer considers this as 100% by providing higher priority to the TimerNLoadCount register.

Following are some points that you must consider while configuring the pulse width modulation with 0% and 100% duty cycle feature:

- The 0% and 100% duty cycle mode is enabled when DW_apb_timers is configured or programmed with the following:
  - Configured with TIM_NEWMODE = 1 and TIMER_0N100_PWM_HC_EN = 0
    - Enable TimerNControlReg [4] – 0% and 100% duty cycle mode enable bit.
    - Enable TimerNControlReg [3] – PWM enable bit.

        - Timer mode is configured as user-defined count mode, that is by setting the TimerNControlReg [1] bit.
    - Configured with TIM_NEWMODE = 1 and TIMER_0N100_PWM_HC_EN = 1
        - Enable TimerNControlReg [3] – PWM enable bit.
        - Timer mode is configured as user-defined count mode that is by setting the TimerNControlReg [1] bit.
- The 0% and 100% duty cycle mode is enabled (TIMER_0N100_PWM_MODE=1) only when any of the timer has TIMER_HAS_TOGGLE_N=1 and TIM_NEWMODE=1. Otherwise, this mode is not enabled.
- When TIMER_0N100_PWM_MODE=1 and toggle output is in 0% and 100% duty cycle mode, the timer interrupts are generated.
- The 0% and 100% duty cycle mode can be enabled only if the following bits are set:
    - TimerNControlReg [3] (PWM enable bit)
    - TimerNControlReg [1] (Timer mode bit)
    - TimerNControlReg [4] (0% and 100% duty cycle mode bit)

  If any of these modes are not enabled, that is if PWM mode or user-defined mode is not enabled, then the timer operates in the normal PWM mode or free running mode.
- The 0% and 100% duty cycle mode can be enabled by programming 0x0 into the TimerNLoadCount or TimerNLoadCount2 register as described in “Pulse Width Modulation with 0% and 100% Duty Cycle” on page 28. If any other combination of values is programmed, then it operates in the normal PWM mode.
- The TimerNControlReg [4] bit enables the “0% and 100% duty cycle mode". You must not set any random value to this bit.

2.3.8 Timer Pause Mode

The operation of a timer can be paused by asserting the respective timer_*N*_pause input signal, which is synchronized to the timer_*N*_clk domain.

2.4 Clocks and Resets

2.4.1 Clocks

DW_apb_timers has the following clocks:

- pclk – APB clock or system clock, times all the bus transfers
- timer_N_clk – Timer clock

All registers in the APB interface are synchronous to pclk. Each timer_N_clk can be synchronous or asynchronous to pclk. It is possible to connect timer_N_clk to a clock other than pclk, in that case the possibility of synchronization and metastability must be taken into account.

2.4.2 Resets

The DW_apb_timers includes separate reset signals dedicated to each clock domain

- The presetn signal resets logic in pclk domain
- The timer_N_resetn signal resets logic in the timer_N_clk domain

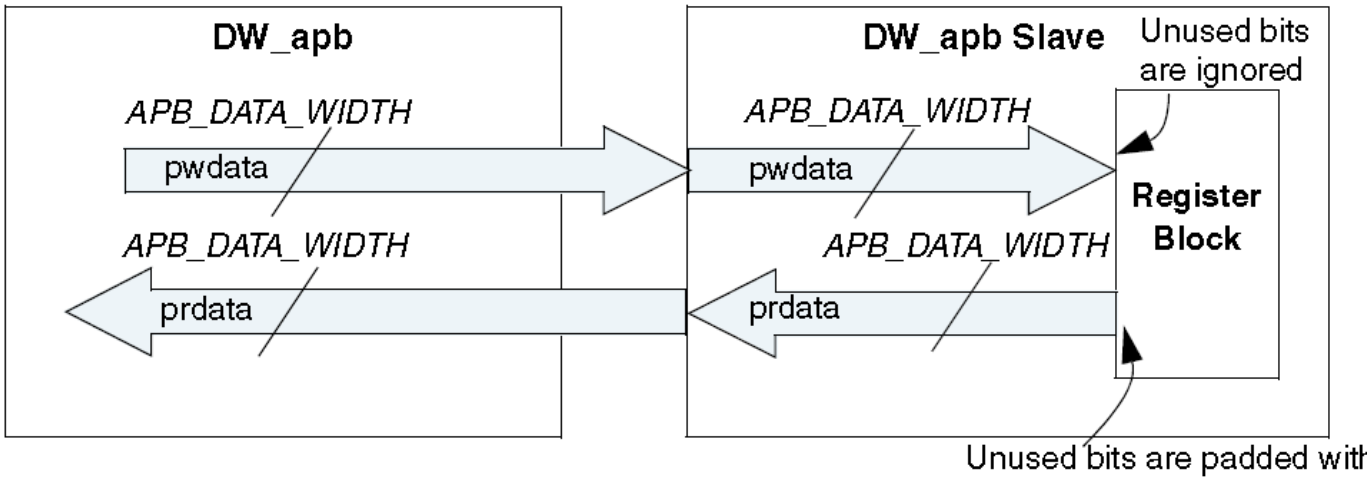
The resets are active-low and can be asynchronously asserted but the de-assertion must be synchronous to the respective clock. DW_apb_timers does not contain logic to perform this synchronization, it must be taken care externally.

2.5 APB Interface

The host processor accesses internal registers on the DW_apb_timers 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. The SLAVE_INTERFACE_TYPE parameter is used to select the register interface type as APB2, APB3 or APB4. By default, DW_apb_timers supports the APB2 interface.

Figure 2-10 shows the read/write buses between the DW_apb and the APB slave.

Figure 2-10 Read/Write Buses Between the DW_apb and an APB Slave



The data, control and status registers within the DW_apb_timers are byte-addressable. The maximum width of the control or status register (except for the TIMERS_COMP_VERSION register) in the DW_apb_timers 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_timers control and status registers require only one APB access.

The timer load count and current value register width depends on the TIMER_WIDTH_N parameter, which can vary from 8 to 32. Depending on the previously mentioned width of the timer and the APB data bus width (that is, the APB_DATA_WIDTH parameter), the APB interface may need to perform single or multiple accesses to the timer load count and current value register.

"Integration Considerations" on page 83 provides information about reading to and writing from the APB interface.

The APB 3 and APB4 register accesses to the DW_apb_timers peripheral are discussed in the following sections:

- "APB 3.0 Support" on page 33
- "APB 4.0 Support" on page 33

2.5.1 APB 3.0 Support

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

- PREADY – This signal specifies the end of a transaction when there is a high in the access phase of a transaction. This signal is always set to its default value that is high for all APB processes.
- 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_timers provides any slave error response from register interface. For more information on this signal, see "APB 4.0 Support" on page 33.

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

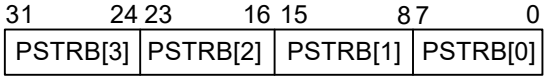
2.5.2 APB 4.0 Support

The DW_apb_timers 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_timers 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_timers 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. The incoming strobe bits for a read transaction is always zero as per the AMBA APB 4.0 protocol.

  Figure 2-11 shows the byte lane mapping of the PSTRB signal.

Figure 2-11 PSTRB Signal Byte-Lane Mapping



- PPROT – This signal supports the protection feature of the APB4 protocol. The APB4 protection feature is supported only on the TimerNLoadCount and TimerNLoadCount2 registers. The protection level register (TIMER_N_PROT_LEVEL) defines the APB4 protection level, that is the protected registers (TimerNLoadCount and TimerNLoadCount2) are updated only if the PPROT privilege is more than the protection privilege programmed in the protection level register (see Table 2-3). 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

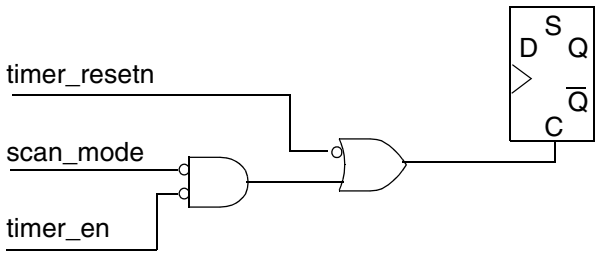
Table 2-3 PPROT Level, Protection Level Programmed in TIMER_N_PROT_LEVEL, and Slave Error Response

PPROT			TIMER_N_PROT_LEVEL			
[2]	[1]	[0]	[2]	[1]	[0]	PSLVERR
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

A scan_mode signal controls the asynchronous clear signal of some of the flip-flops during scan testing; the operation of this is shown in Figure 2-12. In normal operation, in order to load a new value into a timer, the timer must be disabled. The new value is loaded into the timer on the first rising edge of the clock when the timer is re-enabled. To implement this, an asynchronous end-of-interrupt signal is supplied to some internal flip-flops. If scan_mode is asserted, this asynchronous signal is controlled by the timer reset signal. The scan_mode signal must be asserted during scan testing in order to ensure that all flip-flops in the design can be controlled and observed during scan testing; at all other times, this signal must be de-asserted.

Figure 2-12 Design For Test – Use of Scan Mode Signal



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
Top Level Parameters	
Register Interface Type	Selects Register Interface type as APB2, APB3 or APB4. By default, DW_apb_timers 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. The component 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
TIMERS Protection Level	Reset Value of TIMER_N_PROT_LEVEL register. A high on any bit of timer 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:** 0x2 **Enabled:** SLAVE_INTERFACE_TYPE>1 && SLVERR_RESP_EN==1 **Parameter Name:** PROT_LEVEL_RST
Hard-Code Protection Level?	Checking this parameter makes TIMERS_N_PROT_LEVEL a read-only register, reflecting default PROT_LEVEL_RST when read. 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

Table 3-1 Top Level Parameters (Continued)

Label	Description
APB Data Bus Width	Width of the APB data bus to which this component is attached. **Values:** 8, 16, 32 **Default Value:** 32 **Enabled:** Always **Parameter Name:** APB_DATA_WIDTH
Enable Timer New Mode ?	When set to True (1), this parameter enables the following features in all the timers: ■ If TimerNControlReg[4] is set to 1, the width of LOW and HIGH periods of timer toggle outputs can be separately programmed through TimerNLoadCount and TimerNLoadCount2 registers, respectively. ■ Timer_N_clk can be free-running; that is, timer_n_clk does not have to be stopped when timer is disabled. ■ Timer interrupt can be detected, even when pclk is stopped. ■ Timer can be paused using timer_N_pause inputs. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** APB_DATA_WIDTH==32 **Parameter Name:** TIM_NEWMODE
Interrupt Synchronized to System clock(pclk)/Timer clock(timer_clk) ?	When TIM_NEWMODE is enabled, the timer interrupt can be generated either in the system clock (pclk) or in the Timer clock (timer_clk) domain. When set to 0, the timer interrupt is generated in the Timer clock domain; when set to 1, the timer interrupt is generated in the system clock domain. **Values:** ■ Timer clock (timer_clk) (0) ■ system clock (pclk) (1) **Default Value:** Timer clock (timer_clk) **Enabled:** TIM_NEWMODE==1 **Parameter Name:** INTR_SYNC2PCLK

Table 3-1 Top Level Parameters (Continued)

Label	Description
Enable Timer 0% and 100% PWM Mode ?	When set to True (1), this parameter enables the 0% and 100% PWM mode on the toggle output. This feature adds 1-bit to the TimerNControlReg as follows: TimerNControlReg[4] - Timer 0% and 100% duty cycle Mode Enable **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** ((TIMER_HAS_TOGGLE_1 == 1) || (TIMER_HAS_TOGGLE_2 == 1) || (TIMER_HAS_TOGGLE_3 == 1) || (TIMER_HAS_TOGGLE_4 == 1) || (TIMER_HAS_TOGGLE_5 == 1) || (TIMER_HAS_TOGGLE_6 == 1) || (TIMER_HAS_TOGGLE_7 == 1) || (TIMER_HAS_TOGGLE_8 == 1)) && (TIM_NEWMODE_VAL == 1) **Parameter Name:** TIM_0N100_PWM_MODE
Hardcode Timer 0% and 100% PWM Mode enable bit?	When set to True (1), this parameter hardcodes the 0% and 100% PWM mode enable bit in the TimerNControlReg in the register. This is provided to reduce the software overhead. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** TIM_0N100_PWM_MODE == 1 **Parameter Name:** TIMER_0N100_PWM_HC_EN
Number of Timers to instantiate	Number of timers to instantiate in DW_apb_timers. Up to eight timers can be instantiated. **Values:** 1, 2, 3, 4, 5, 6, 7, 8 **Default Value:** 2 **Enabled:** Always **Parameter Name:** NUM_TIMERS
Interrupt Polarity	Polarity of interrupt signals generated by DW_apb_timers. **Values:** ■ Active Low (0) ■ Active High (1) **Default Value:** Active High **Enabled:** TIM_NEWMODE==0 **Parameter Name:** TIM_INTRPT_PLRITY

Table 3-1 Top Level Parameters (Continued)

Label	Description
Single Combined Interrupt?	When set to True (1), the component generates a single interrupt combining all timer interrupts. If set to False (0), the component generates an interrupt output for each timer. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** TIM_NEWMODE==0 **Parameter Name:** TIM_INTR_IO

3.2 Timer N Configuration Parameters

Table 3-2 Timer N Configuration Parameters

Label	Description
Timer N Configuration	
Width of Timer #N (for N = 1; N <= NUM_TIMERS)	Width of each Timer. **Values:** 8, ..., 32 **Default Value:** 32 **Enabled:** NUM_TIMERS >= N **Parameter Name:** TIMER_WIDTH_(N)
Include toggle output for timer #N on I/F? (for N = 1; N <= NUM_TIMERS)	When set to True (1), the interface includes an output (timer_N_toggle) that toggles each time the timer counter reloads. The output is disabled to 0 each time the timer is disabled. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** NUM_TIMERS >= N **Parameter Name:** TIMER_HAS_TOGGLE_(N)
Metastability support for interrupt from Timer #N (for N = 1; N <= NUM_TIMERS)	This option instantiates metastability registers to synchronize timer interrupt signals to the pclk domain. Set this to Present (1) if timer_N_clk is independent of pclk. If this parameter is set to Absent (0), then timer_N_clk is considered to be connected to or synchronous with pclk. **Values:** ■ Absent (0) ■ Present (1) **Default Value:** TIM_NEWMODE **Enabled:** (TIM_NEWMODE == 0) AND (NUM_TIMERS >= N) **Parameter Name:** TIM_METASTABLE_(N)
Timer N Clock Domain Crossing Synchronization Depth (for N = 1; N <= NUM_TIMERS)	Sets the number of synchronization stages to be placed on clock domain crossing signals for timer N. ■ 2: 2-stage synchronization with positive-edge capturing at both the stages ■ 3: 3-stage synchronization with positive-edge capturing at all stages ■ 4: 4-stage synchronization with positive-edge capturing at all stages **Values:** 2, 3, 4 **Default Value:** 2 **Enabled:** TIM_METASTABLE_(N)==1 **Parameter Name:** TIM_SYNC_DEPTH_(N)

Table 3-2 Timer N Configuration Parameters (Continued)

Label	Description
Number of clock cycles by which to extend interrupt for Timer #N (for N = 1; N <= NUM_TIMERS)	If this timer clock is faster than the system bus clock, you can extend the internal interrupt by up to three timer clock cycles to guarantee that it is seen in the bus clock domain. A 0 value in this field means that no pulse extension is performed. Also refer to the "Controlling Clock Boundaries and Metastability" section in the DW_apb_timers databook. Set this parameter to the following values, depending on the timer_N_clk/pclk frequency ratio R: **timer_N_clk/pclk frequency RPULSE_EXTEND_N** R<=1 ---------- 0 1<R<=2 ---------- 1 2<R<=3 ---------- 2 3<R<=4 ---------- 3 R>4 ---------- Not Valid **Values:** 0, 1, 2, 3 **Default Value:** 0 **Enabled:** (TIM_NEWMODE == 0) AND (NUM_TIMERS >= N) **Parameter Name:** TIM_PULSE_EXTD_(N)
Include Coherency Registers for this Timer? (for N = 1; N <= NUM_TIMERS)	When set to True (1), a bank of registers is added between this timer and the APB interface of DW_apb_timers to guarantee that the timer value read back from this block is coherent. It does not reflect ongoing changes in the timer value that takes place while the read operation is in progress. **Note**: Including coherency can dramatically increase the register count of the design. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** (TIMER_WIDTH_N > APB_DATA_WIDTH) AND (NUM_TIMERS >= N) **Parameter Name:** TIM_COHERENCY_(N)

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 Interface on page 45
- Timer Signals on page 48

4.1 APB Interface Signals

Table 4-1 APB Interface Signals

Port Name	I/O	Description
pclk	I	APB clock; also known as the system clock. This clock times all bus transfers. All signal timings are related to the rising edge of pclk. **Exists:** Always **Synchronous To:** None **Registered:** N/A **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
presetn	I	APB reset. The bus reset signal is used to reset the system and the bus on the DesignWare interface. 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. DW_apb_timers does not contain logic to perform this synchronization, so it must be provided externally. **Exists:** Always **Synchronous To:** Asynchronous **Registered:** N/A **Power Domain:** SINGLE_DOMAIN **Active State:** Low
penable	I	APB enable control that indicates the second cycle of the APB frame. **Exists:** Always **Synchronous To:** pclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High

Table 4-1 APB Interface Signals (Continued)

Port Name	I/O	Description
psel	I	APB peripheral select. **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
paddr[TIM_ADDR_SLICE_LHS:0]	I	APB address bus. **Exists:** Always **Synchronous To:** pclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
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
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 timer load-count registers. Else the DW_apb_timers 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

Table 4-1 APB Interface Signals (Continued)

Port Name	I/O	Description
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
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_timers 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 an incoming transaction does not have necessary authorisation. This signal is tied to low in case SLVERR_RESP_EN parameter is switched off. **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 Timer Signals

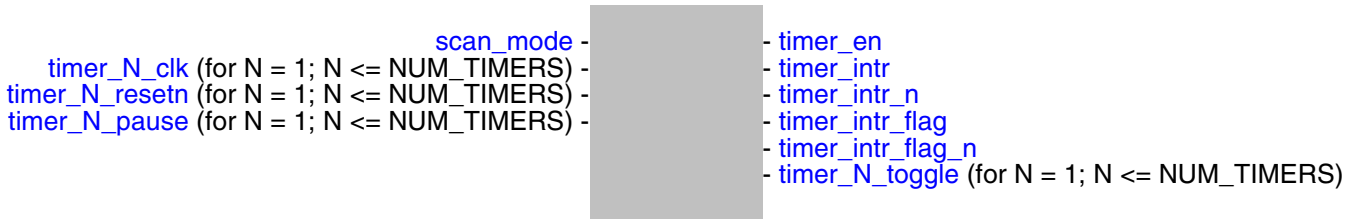


Table 4-2 Timer Signals

Port Name	I/O	Description
scan_mode	I	Active-high scan mode used to ensure that test automation tools can control all asynchronous flip-flop signals. This signal should be asserted that is, driven to logic 1 during scan testing, and should be deasserted (tied to logic 0) at all other times. **Exists:** Always **Synchronous To:** Asynchronous **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
timer_N_clk (for N = 1; N <= NUM_TIMERS)	I	Each timer is supplied with its own clock from this bus. The number of these signals is set by NUM_TIMERS parameter. This signal can be asynchronous or synchronous to pclk. If a timer clock is asynchronous to pclk, you must ensure that the clocks are stopped whenever the timer is disabled. **Exists:** NUM_TIMERS >= N **Synchronous To:** None **Registered:** N/A **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
timer_N_resetn (for N = 1; N <= NUM_TIMERS)	I	Asynchronous reset for each timer. The number of these signals are set by NUM_TIMERS parameter. Asynchronous assertion, synchronous de-assertion. Must be synchronously de-asserted after the rising edge of timer_1_clk. **Exists:** NUM_TIMERS >= N **Synchronous To:** Asynchronous **Registered:** N/A **Power Domain:** SINGLE_DOMAIN **Active State:** Low

Table 4-2 Timer Signals (Continued)

Port Name	I/O	Description
timer_N_pause (for N = 1; N <= NUM_TIMERS)	I	Optional. Input signal; when asserted, causes the timer to pause/freeze. **Exists:** (NUM_TIMERS >= N) && (TIM_NEWMODE==1) **Synchronous To:** timer_N_clk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** High
timer_en[(NUM_TIMERS-1):0]	O	When asserted, activates the necessary timer clocks and ensures the component is supplied with an active pclk while timers are running. You can tie a timer clock to pclk, but if pclk is asynchronous to a timer clock, then you must stop the timer clock before programming it. Timer clock should start and stop depending on assertion and de-assertion of the timer_en output signal when the timer clock is asynchronous to pclk. **Exists:** Always **Synchronous To:** pclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** High
timer_intr[(NUM_TIMERS-1):0]	O	Optional. Timer interrupt active high signals. It's assertion is synchronous to timer_N_clk and de-assertion is synchronous to pclk. **Exists:** (TIM_INTRPT_PLRITY==1) && (TIM_INTR_IO==TIM_INDIVIDUAL) **Synchronous To:** None **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
timer_intr_n[(NUM_TIMERS-1):0]	O	Optional. Timer interrupt active low signals. It's assertion is synchronous to timer_N_clk and de-assertion is synchronous to pclk. **Exists:** (TIM_INTRPT_PLRITY==0) && (TIM_INTR_IO==TIM_INDIVIDUAL) **Synchronous To:** None **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** Low

Table 4-2 Timer Signals (Continued)

Port Name	I/O	Description
timer_intr_flag	O	Optional. Active High Interrupt flag that is set if any timer interrupt is set. **Exists:** (TIM_INTRPT_PLRITY==1) && (TIM_INTR_IO==TIM_COMBINED) **Synchronous To:** pclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
timer_intr_flag_n	O	Optional. Active Low Interrupt flag that is set if any timer interrupt is set. **Exists:** (TIM_INTRPT_PLRITY==0) && (TIM_INTR_IO==TIM_COMBINED) **Synchronous To:** pclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** Low
timer_N_toggle (for N = 1; N <= NUM_TIMERS)	O	Optional. Signal that toggles each time the timer counter reloads. The output is disabled to 0 each time the timer is disabled. **Exists:** (NUM_TIMERS >= N) && (TIMER_HAS_TOGGLE_N==1) **Synchronous To:** timer_N_clk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** High

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: DW_apb_timers_mem_map

Address Block	Description
DW_apb_timers_addr_block on page 54	DW_apb_timers address block **Exists:** Always

5.1 DW_apb_timers_mem_map/DW_apb_timers_addr_block Registers

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

Table 5-5 Registers for Address Block: DW_apb_timers_mem_map/DW_apb_timers_addr_block

Register	Offset	Description
TimerNLoadCount (for N = 1; N <= NUM_TIMERS) on page 55	0x00 + (N-1)*0x14	Timer N Load Count Register
TimerNCurrentValue (for N = 1; N <= NUM_TIMERS) on page 57	0x04 + (N-1)*0x14	Current value of Timer N
TimerNControlReg (for N = 1; N <= NUM_TIMERS) on page 60	0x08 + (N-1)*0x14	Timer N Control Register
TimerNEOI (for N = 1; N <= NUM_TIMERS) on page 63	0x0C + (N-1)*0x14	Timer N End-of-Interrupt Register
TimerNIntStatus (for N = 1; N <= NUM_TIMERS) on page 65	0x10 + (N-1)*0x14	Timer N Interrupt Status Register
TimersIntStatus on page 67	0xa0	Timers Interrupt Status Register
TimersEOI on page 69	0xa4	Timers End-of-Interrupt Register
TimersRawIntStatus on page 70	0xa8	Timers Raw Interrupt Status Register
TIMERS_COMP_VERSION on page 71	0xac	Timers Component Version
TimerNLoadCount2 (for N = 1; N <= NUM_TIMERS) on page 72	0xb0 + (N-1)*0x04	Timer N Load Count2 Register
TIMER_N_PROT_LEVEL (for N = 1; N <= NUM_TIMERS) on page 73	0xd0 + (N-1)*0x04	Timer_N Protection level

5.1.1 TimerNLoadCount (for N = 1; N <= NUM_TIMERS)

- **Name:** Timer N Load Count Register
- **Description:** Value to be loaded into Timer N
- **Size:** 32 bits
- **Offset:** 0x00 + (N-1)*0x14
- **Exists:** NUM_TIMERS >= N

Table 5-6 Fields for Register: TimerNLoadCount (for N = 1; N <= NUM_TIMERS)

Bits	Name	Memory Access	Description
31:y	RSVD_TIMER1LOADCOUNT	R	TIMER1LOADCOUNT 31toTIMER_WIDTH_N Reserved field. **Note:** ■ The reserved field name for Timer 2 will be RSVD_TIMER2LOADCOUNT. ■ The reserved field name for Timer 3 will be RSVD_TIMER3LOADCOUNT. ■ The reserved field name for Timer 4 will be RSVD_TIMER4LOADCOUNT. ■ The reserved field name for Timer 5 will be RSVD_TIMER5LOADCOUNT. ■ The reserved field name for Timer 6 will be RSVD_TIMER6LOADCOUNT. ■ The reserved field name for Timer 7 will be RSVD_TIMER7LOADCOUNT. ■ The reserved field name for Timer 8 will be RSVD_TIMER8LOADCOUNT. **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** TIMER_WIDTH_N
x:0	TIMER1LOADCOUNT	R/W	Value to be loaded into Timer 1. This is the value from which counting commences. Any value written to this register is loaded into the associated timer. **Note:** ■ The field name for Timer 2 will be TIMER2LOADCOUNT. ■ The field name for Timer 3 will be TIMER3LOADCOUNT. ■ The field name for Timer 4 will be TIMER4LOADCOUNT. ■ The field name for Timer 5 will be TIMER5LOADCOUNT. ■ The field name for Timer 6 will be TIMER6LOADCOUNT. ■ The field name for Timer 7 will be TIMER7LOADCOUNT. ■ The field name for Timer 8 will be TIMER8LOADCOUNT. **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** TIMER_WIDTH_N - 1

5.1.2 TimerNCurrentValue (for N = 1; N <= NUM_TIMERS)

- **Description:** Current value of Timer N
- **Size:** 32 bits
- **Offset:** 0x04 + (N-1)*0x14
- **Exists:** NUM_TIMERS >= N

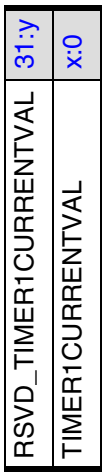


Table 5-7 Fields for Register: TimerNCurrentValue (for N = 1; N <= NUM_TIMERS)

Bits	Name	Memory Access	Description
31:y	RSVD_TIMER1CURRENTVAL	R	TIMER1CURRENTVAL 31toTIMER_WIDTH_N Reserved field. **Note:** ■ The reserved field name for Timer 2 will be RSVD_TIMER2CURRENTVAL. ■ The reserved field name for Timer 3 will be RSVD_TIMER3CURRENTVAL. ■ The reserved field name for Timer 4 will be RSVD_TIMER4CURRENTVAL. ■ The reserved field name for Timer 5 will be RSVD_TIMER5CURRENTVAL. ■ The reserved field name for Timer 6 will be RSVD_TIMER6CURRENTVAL. ■ The reserved field name for Timer 7 will be RSVD_TIMER7CURRENTVAL. ■ The reserved field name for Timer 8 will be RSVD_TIMER8CURRENTVAL. **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** TIMER_WIDTH_N

Table 5-7 Fields for Register: TimerNCurrentValue (for N = 1; N <= NUM_TIMERS) (Continued)

Bits	Name	Memory Access	Description
x:0	TIMER1CURRENTVAL	R	Current Value of Timer 1. When TIM_NEWMODE=0, This register is supported only when timer_1_clk is synchronous to pclk. Reading this register when using independent clocks results in an undefined value. When TIM_NEWMODE=1, no restrictions apply. **Note:** ■ The field name for Timer 2 will be TIMER2CURRENTVAL. ■ The field name for Timer 3 will be TIMER3CURRENTVAL. ■ The field name for Timer 4 will be TIMER4CURRENTVAL. ■ The field name for Timer 5 will be TIMER5CURRENTVAL. ■ The field name for Timer 6 will be TIMER6CURRENTVAL. ■ The field name for Timer 7 will be TIMER7CURRENTVAL. ■ The field name for Timer 8 will be TIMER8CURRENTVAL. **Note:** TIMERNCURRENTVALUE is synchronized from timer clock domain to the APB clock domain. When TIM_NEWMODE=1, reading register TIMERNCURRENTVALUE immediately after reset returns all zeros. Reading this register after synchronization depth number of clock cycles returns the synchronized value which is TIM_RST_CURRENTVAL_N. **Value After Reset:** TIM_RST_CURRENTVAL_[N] **Exists:** Always **Volatile:** true **Range Variable[x]:** TIMER_WIDTH_N - 1

5.1.3 TimerNControlReg (for N = 1; N <= NUM_TIMERS)

- **Name:** Timer N Control Register
- **Description:** Control Register for Timer N. This register controls enabling, operating mode (free-running or defined-count), and interrupt mask of Timer N. You can program each Timer1ControlReg to enable or disable a specific timer and to control its mode of operation.
- **Size:** 32 bits
- **Offset:** 0x08 + (N-1)*0x14
- **Exists:** NUM_TIMERS >= N

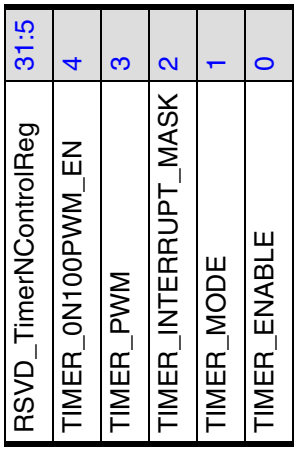


Table 5-8 Fields for Register: TimerNControlReg (for N = 1; N <= NUM_TIMERS)

Bits	Name	Memory Access	Description
31:5	RSVD_TimerNControlReg	R	TimerNControlReg 31to5 Reserved field **Value After Reset:** 0x0 **Exists:** Always

Table 5-8 Fields for Register: TimerNControlReg (for N = 1; N <= NUM_TIMERS) (Continued)

Bits	Name	Memory Access	Description
4	TIMER_0N100PWM_EN	((TIM_0N100_PWM_MODE==1) AND (TIMER_HAS_TOGGLE_N==1) AND (TIMER_0N100_PWM_HC_EN==0)) ? read-write : read-only	Optional. Allows user to enable or disable the usage of Timer 0% and 100% mode feature. This bit is present only when (TIM_0N100_PWM_MODE=1 and TIMER_HAS_TOGGLE_N=1). Otherwise reserved. **Values:** ■ 0x1 (ENABLED): Timer 0% and 100% PWM duty cycle mode is enabled ■ 0x0 (DISABLE): Timer 0% and 100% PWM duty cycle mode is disabled **Value After Reset:** {((TIM_0N100_PWM_MODE==1) && (TIMER_HAS_TOGGLE_N==1) && (TIMER_0N100_PWM_HC_EN==1)) ? 0x1 : 0x0} **Exists:** TIM_0N100_PWM_MODE && TIMER_HAS_TOGGLE_N
3	TIMER_PWM	R/W	Pulse Width Modulation of timer_N_toggle output. This field is only present when TIM_NEWMODE is enabled **Values:** ■ 0x1 (ENABLED): PWM for timer_N_toggle o/p is enabled ■ 0x0 (DISABLE): PWM for timer_N_toggle o/p is disabled **Value After Reset:** 0x0 **Exists:** TIM_NEWMODE==1
2	TIMER_INTERRUPT_MASK	R/W	Timer interrupt mask for Timer N. **Values:** ■ 0x1 (MASKED): Timer N interrupt is masked ■ 0x0 (UNMASKED): Timer N interrupt is unmasked **Value After Reset:** 0x0 **Exists:** Always
1	TIMER_MODE	R/W	Timer mode for Timer N. **Note**: You must set the Timer1LoadCount register to all 1s before enabling the timer in free-running mode. **Values:** ■ 0x1 (USER_DEFINED): User-Defined mode of operation ■ 0x0 (FREE_RUNNING): Free Running mode of operation **Value After Reset:** 0x0 **Exists:** Always

Table 5-8 Fields for Register: TimerNControlReg (for N = 1; N <= NUM_TIMERS) (Continued)

Bits	Name	Memory Access	Description
0	TIMER_ENABLE	R/W	Timer enable bit for Timer N. **Values:** ■ 0x1 (ENABLED): Timer N is enabled ■ 0x0 (DISABLE): Timer N is disabled **Value After Reset:** 0x0 **Exists:** Always

5.1.4 TimerNEOI (for N = 1; N <= NUM_TIMERS)

- **Name:** Timer N End-of-Interrupt Register
- **Description:** Clears the interrupt from Timer N
- **Size:** 32 bits
- **Offset:** 0x0C + (N-1)*0x14
- **Exists:** NUM_TIMERS >= N

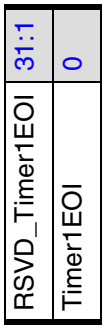


Table 5-9 Fields for Register: TimerNEOI (for N = 1; N <= NUM_TIMERS)

Bits	Name	Memory Access	Description
31:1	RSVD_Timer1EOI	R	Timer1EOI 31to1 Reserved field. **Note:** ■ The reserved field name for Timer 2 will be RSVD_Timer2EOI. ■ The reserved field name for Timer 3 will be RSVD_Timer3EOI. ■ The reserved field name for Timer 4 will be RSVD_Timer4EOI. ■ The reserved field name for Timer 5 will be RSVD_Timer5EOI. ■ The reserved field name for Timer 6 will be RSVD_Timer6EOI. ■ The reserved field name for Timer 7 will be RSVD_Timer7EOI. ■ The reserved field name for Timer 8 will be RSVD_Timer8EOI. **Value After Reset:** 0x0 **Exists:** Always

Table 5-9 Fields for Register: TimerNEOI (for N = 1; N <= NUM_TIMERS) (Continued)

Bits	Name	Memory Access	Description
0	Timer1EOI	R	Reading from this register returns all zeroes (0) and clears the interrupt from Timer 1. **Note:** ■ The field name for Timer 2 will be Timer2EOI. ■ The field name for Timer 3 will be Timer3EOI. ■ The field name for Timer 4 will be Timer4EOI. ■ The field name for Timer 5 will be Timer5EOI. ■ The field name for Timer 6 will be Timer6EOI. ■ The field name for Timer 7 will be Timer7EOI. ■ The field name for Timer 8 will be Timer8EOI. **Value After Reset:** 0x0 **Exists:** Always

5.1.5 TimerNIntStatus (for N = 1; N <= NUM_TIMERS)

- **Name:** Timer N Interrupt Status Register
- **Description:** Contains the interrupt status for Timer N
- **Size:** 32 bits
- **Offset:** 0x10 + (N-1)*0x14
- **Exists:** NUM_TIMERS >= N

Table 5-10 Fields for Register: TimerNIntStatus (for N = 1; N <= NUM_TIMERS)

Bits	Name	Memory Access	Description
31:1	RSVD_TIMER1INTSTAT	R	Timer1IntStatus 31to1 Reserved field **Note:** ■ The reserved field name for Timer 2 will be RSVD_TIMER2INTSTAT. ■ The reserved field name for Timer 3 will be RSVD_TIMER3INTSTAT. ■ The reserved field name for Timer 4 will be RSVD_TIMER4INTSTAT. ■ The reserved field name for Timer 5 will be RSVD_TIMER5INTSTAT. ■ The reserved field name for Timer 6 will be RSVD_TIMER6INTSTAT. ■ The reserved field name for Timer 7 will be RSVD_TIMER7INTSTAT. ■ The reserved field name for Timer 8 will be RSVD_TIMER8INTSTAT. **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true
0	TIMER1INTSTAT	R	Contains the interrupt status for Timer 1. **Note:** ■ The field name for Timer 2 will be TIMER2INTSTAT. ■ The field name for Timer 3 will be TIMER3INTSTAT. ■ The field name for Timer 4 will be TIMER4INTSTAT. ■ The field name for Timer 5 will be TIMER5INTSTAT. ■ The field name for Timer 6 will be TIMER6INTSTAT. ■ The field name for Timer 7 will be TIMER7INTSTAT. ■ The field name for Timer 8 will be TIMER8INTSTAT. **Values:** ■ 0x1 (ACTIVE): Timer N Interrupt is active ■ 0x0 (INACTIVE): Timer N Interrupt is inactive **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true

5.1.6 TimersIntStatus

- **Name:** Timers Interrupt Status Register
- **Description:** Contains the interrupt status of all timers in the component.
- **Size:** 32 bits
- **Offset:** 0xa0
- **Exists:** Always

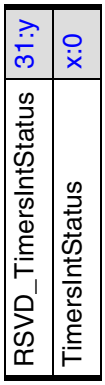


Table 5-11 Fields for Register: TimersIntStatus

Bits	Name	Memory Access	Description
31:y	RSVD_TimersIntStatus	R	TimersIntStatus 31toNUM_TIMERS Reserved field **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** NUM_TIMERS

Table 5-11 Fields for Register: TimersIntStatus (Continued)

Bits	Name	Memory Access	Description
x:0	TimersIntStatus	R	Contains the interrupt status of all timers in the component. If a bit of this register is 0, then the corresponding timer interrupt is not active and the corresponding interrupt could be on either the timer_intr bus or the timer_intr_n bus, depending on the interrupt polarity you have chosen. Similarly, if a bit of this register is 1, then the corresponding interrupt bit has been set in the relevant interrupt bus. In both cases, the status reported is the status after the interrupt mask has been applied. Reading from this register does not clear any active interrupts. **Values:** ■ 0x1 (ACTIVE): Timer_intr(_n) is active ■ 0x0 (INACTIVE): Timer_intr(_n) is inactive **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** NUM_TIMERS - 1

5.1.7 TimersEOI

- **Name:** Timers End-of-Interrupt Register
- **Description:** Returns all zeroes (0) and clears all active interrupts.
- **Size:** 32 bits
- **Offset:** 0xa4
- **Exists:** Always

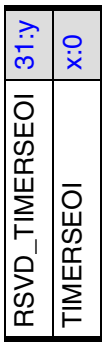


Table 5-12 Fields for Register: TimersEOI

Bits	Name	Memory Access	Description
31:y	RSVD_TIMERSEOI	R	TimersEOI 31toNUM_TIMERS Reserved field **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** NUM_TIMERS
x:0	TIMERSEOI	R	Reading this register returns all zeroes (0) and clears all active interrupts. **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** NUM_TIMERS - 1

5.1.8 TimersRawIntStatus

- **Name:** Timers Raw Interrupt Status Register
- **Description:** Contains the unmasked interrupt status of all timers in the component.
- **Size:** 32 bits
- **Offset:** 0xa8
- **Exists:** Always

Table 5-13 Fields for Register: TimersRawIntStatus

Bits	Name	Memory Access	Description
31:y	RSVD_TIMERSRAWINTSTAT	R	TimersRawIntStatus 31toNUM_TIMERS Reserved field **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** NUM_TIMERS
x:0	TIMERSRAWINTSTAT	R	The register contains the unmasked interrupt status of all timers in the component. **Values:** ■ 0x1 (ACTIVE): Raw Timer_intr(_n) is active ■ 0x0 (INACTIVE): Raw Timer_intr(_n) is inactive **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** NUM_TIMERS - 1

5.1.9 TIMERS_COMP_VERSION

- **Name:** Timers Component Version
- **Description:** Current revision number of the DW_apb_timers component.
- **Size:** 32 bits
- **Offset:** 0xac
- **Exists:** Always

Table 5-14 Fields for Register: TIMERS_COMP_VERSION

Bits	Name	Memory Access	Description
31:0	TIMERSCOMPVERSION	R	Current revision number of the DW_apb_timers component.For the value, see the releases table in the AMBA 2 release notes **Value After Reset:** TIM_VERSION_ID **Exists:** Always

5.1.10 TimerNLoadCount2 (for N = 1; N <= NUM_TIMERS)

- **Name:** Timer N Load Count2 Register
- **Description:** Value to be loaded into Timer N when toggle output changes from 0 to 1
- **Size:** 32 bits
- **Offset:** 0xb0 + (N-1)*0x04
- **Exists:** TIM_NEWMODE==1 AND NUM_TIMERS >= N

Table 5-15 Fields for Register: TimerNLoadCount2 (for N = 1; N <= NUM_TIMERS)

Bits	Name	Memory Access	Description
31:y	RSVD_TIMERNLOADCOUNT2	R	TimerNLoadCount2 31toTIMER_WIDTH_N Reserved field **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** TIMER_WIDTH_N
x:0	TIMERNLOADCOUNT2	R/W	Value to be loaded into Timer N when timer_N_toggle output changes from 0 to 1. This value determines the width of the HIGH period of the timer_N_toggle output. **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** TIMER_WIDTH_N - 1

5.1.11 TIMER_N_PROT_LEVEL (for N = 1; N <= NUM_TIMERS)

- **Name:** Timer_N Protection level
- **Description:** Timer_N Protection level register

  Read/Write Access:

  - R/W if HC_PROT_LEVEL=0, else R

  Enabling protection on any of its three bits would require a greater or equal privilege on the input PPROT signal to gain access to protected registers of the timer.
- **Size:** 32 bits
- **Offset:** 0xd0 + (N-1)*0x04
- **Exists:** (SLAVE_INTERFACE_TYPE > 1 AND SLVERR_RESP_EN==1 AND HC_PROT_LEVEL==0 AND NUM_TIMERS >= N) ? 1 : 0

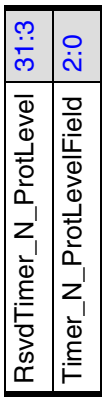


Table 5-16 Fields for Register: TIMER_N_PROT_LEVEL (for N = 1; N <= NUM_TIMERS)

Bits	Name	Memory Access	Description
31:3	RsvdTimer_N_ProtLevel	R	TIMER_N_PROT_LEVEL 31to3 Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always
2:0	Timer_N_ProtLevelField	R/W	This field holds protection value of TIMER_N_PROT_LEVEL register. **Value After Reset:** PROT_LEVEL_RST **Exists:** Always

6 Programming Considerations

This chapter describes the programmable features of the DW_apb_timers.

In order to avoid potential synchronization problems when initializing, loading, and enabling a timer, you should follow the basic procedure outline in "DW_apb_timers Usage Flow" on page 19.

The DW_apb_timers module is little-endian. All timers are disabled on reset and are enabled by writing "1" to the timer enable bit of the TimerNControlReg. The Timer*N*LoadCount register value is loaded into a corresponding Timer*N* after the timer is enabled—either after a disable or a reset. DW_apb_timers contains both timer-specific and system registers.

If a timer is wider than the read data bus to which the slave is attached, more than one access must be performed to read the TimerNCurrentValue register. If more than one access is performed to read a timer value, the coherency of the value read cannot be guaranteed unless you configure read/write coherency for the specific timer. Read/write coherency is meaningful only if the TIMER_WIDTH is greater than the APB_DATA_WIDTH, under which circumstances the coherency registers are never instantiated in the design.

If there is no coherency set for a specific timer, software should read the registers more than once. For example, the software should read least-significant bits (LSBs), then most-significant bits (MSBs), and then LSBs again.

The coherency circuitry incorporates an upper byte method that requires you to program the load register in LSB-to-MSB order when the peripheral width is smaller than the register width. Additionally, you must read LSB-to-MSB for the coherency circuitry solution to operate correctly.

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 timer 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.

Reading the Timer*N*CurrentValue register is not supported if timer_*N*_clk is asynchronous to pclk. Any attempt to read this register when the clocks are independent may result in an undefined value.

6.1 Programming the 0% and 100% Duty Cycle Mode

You can use the following programming flow to enable the 0% and 100% duty cycle mode:

1. Disable the timer enable bit in the TimerNControlReg register.
2. Program the TimerNLoadCount and TimerNLoadCount2 registers with appropriate values as described in "Pulse Width Modulation with 0% and 100% Duty Cycle" on page 28.
3. Enable the 0% and 100% duty cycle mode bit, the Pulse width modulation bit and set the Timer mode to user-defined count mode in the TimerNControlReg register.
4. Set the timer enable bit in the TimerNControlReg register such that the toggle output is 100% (high) or 0% (low).

   When the 0% and 100% duty cycle mode is enabled, internal timer is disabled. The internal timers can be enabled again by switching to Normal toggle output mode or to Pulse width modulation toggle output mode.

7 Verification

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

The DW_apb_timers verification testbench is built with DesignWare Verification IP (VIP). Make sure 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.*

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.

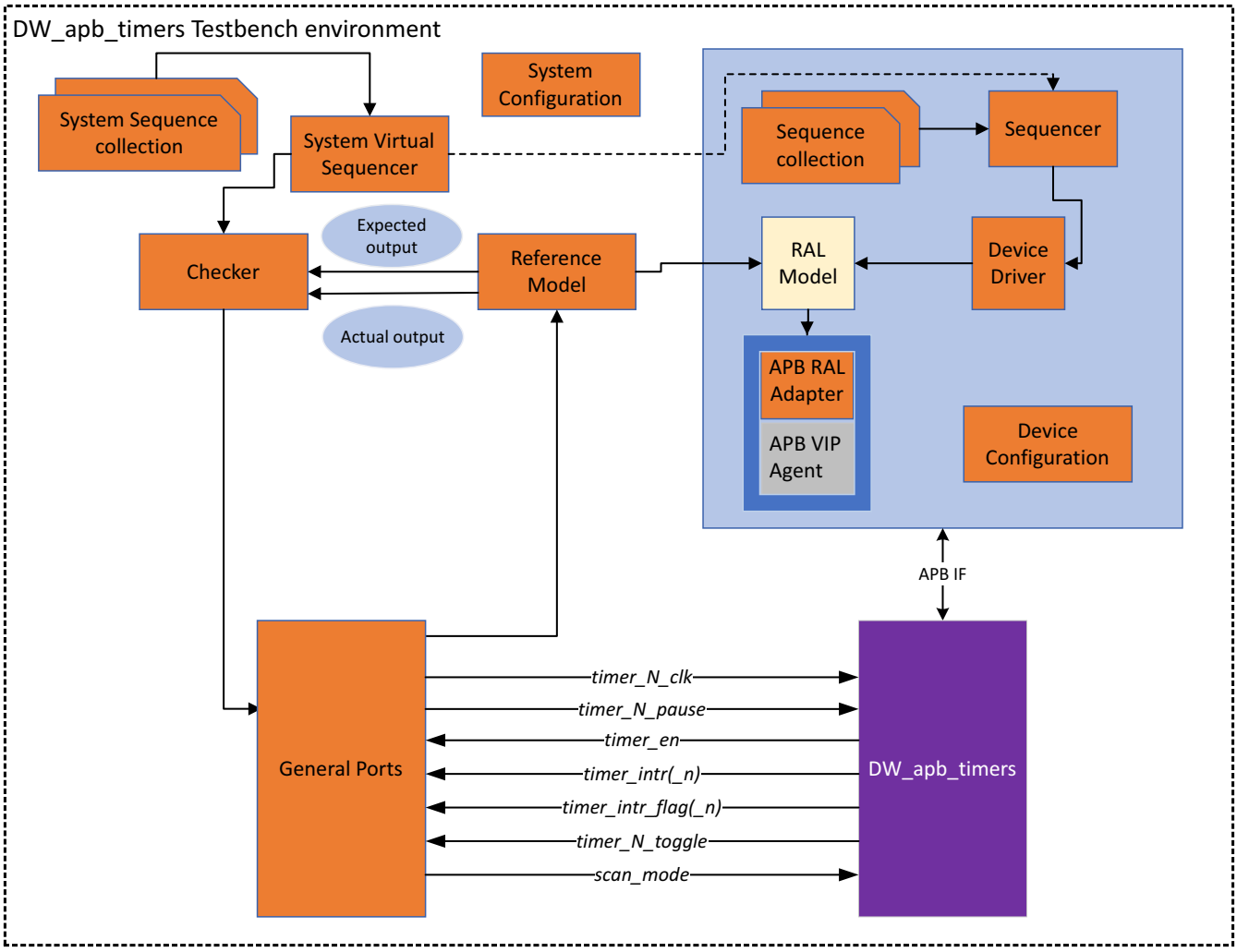
This chapter discusses the following sections:

7.1 Verification Environment

DW_apb_timers 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_timers testbench:

Figure 7-1 DW_apb_timers 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 DUT.

- Testbench uses the following 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 the 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

    RAL model is Inside Device Agent. RAL model is used for two purposes: to show (shadow) the values of registers inside DW_apb_timers, and to check whether the values of registers is correct upon reading. To facilitate RAL model to have a two-way connection with the Reference model, Reference model gets the values from RAL model to check whether behavior on General Ports is correct and is responsible for updates of the RAL model. This is because the 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_timers (mostly status registers).

    The RAL model is used for flexibility and reuse because if you change the adapter and VIP agent (interface you use), you can keep all the other logic.
  - APB RAL Adapter

    APB RAL Adapter converts higher level RAL Model Reads and Writes to APB transactions, these transactions are driven on the bus by APB VIP Agent. By replacing this part with the different adapter, you can 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) you 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. The Checker compares the expected values of DUT output signals against the values actually driven by the DUT.

7.2 Testbench Directories and Files

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

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

Table 7-1 DW_apb_timers Testbench Directory Structure

Directory	Description
<configured workspace>/sim/testbench	Contains the 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/	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_timers Test Description

Test Name	Test Description
test_100_reg_reset_bit_bash	This test builds the sequence `dw_apb_timers_reg_reset_bit_bash_virtual_sequence`, which in turn initiates `dw_apb_timers_reg_reset_bit_bash_sequence`. ■ Validates the register values after reset. ■ Performs bit bash for the bits of all the registers.

Table 7-2 DW_apb_timers Test Description (Continued)

Test Name	Test Description
test_101_random	This test initiates the random testing of all the other test sequences, It performs the following validations. 1. This test builds the sequence `dw_apb_timers_enable_virtual_sequence`. - All the existing timers as per the configuration are made user defined. - Enables by setting respective TimerNControlReg bit 0. - Checks the timers counter change on reading TimerNCurrentValue. 2. This test builds the sequence `dw_apb_timers_intr_virtual_sequence`. - Handles the timer counter activity by enabling it and setting its load value. - Checks the generation of interrupt when it meets the max value loaded. - Clears the interrupt. - Performs these checks for all the existing N timers for the given configuration. 3. This test builds the sequence `dw_apb_timers_pause_virtual_sequence`. - Enables the timer. - Enables the pause mode on asserting timer_N_pause input signal. - Checks the TimerNCurrentValue to see that no change in the counter value on pausing the timer. - Disbales the pause mode and check change in counter value. - Follow these steps for the existing timers for the given configuration. 4. This test builds the sequence `dw_apb_timers_prot_virtual_sequence`, which in turn calls `dw_apb_timers_prot_feature_sequence`. - Checks the prot feature for registers TimerNLoadCount and TimerNLoadCount2 when PSLVERR_RESP_EN is set as high.
test_102_pwm	This test builds the sequence `dw_apb_timers_pwm_virtual_sequence`. ■ Enables the timer. ■ Enables the PWM mode by setting the TimerNControlReg[4] to 1. ■ Sets condition TIM_HAS_TOGGLE_N = 1. ■ Writes the TimerNLoadCount register for LOW period and the TimerNLoadCount2 register for High period values to vary the DUTY cycle. ■ Follow these steps for all the existing N timers for the given configuration.

8 Integration Considerations

After you have configured, tested, and synthesized your component in either coreAssembler or coreConsultant, you can integrate the subsystem or 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

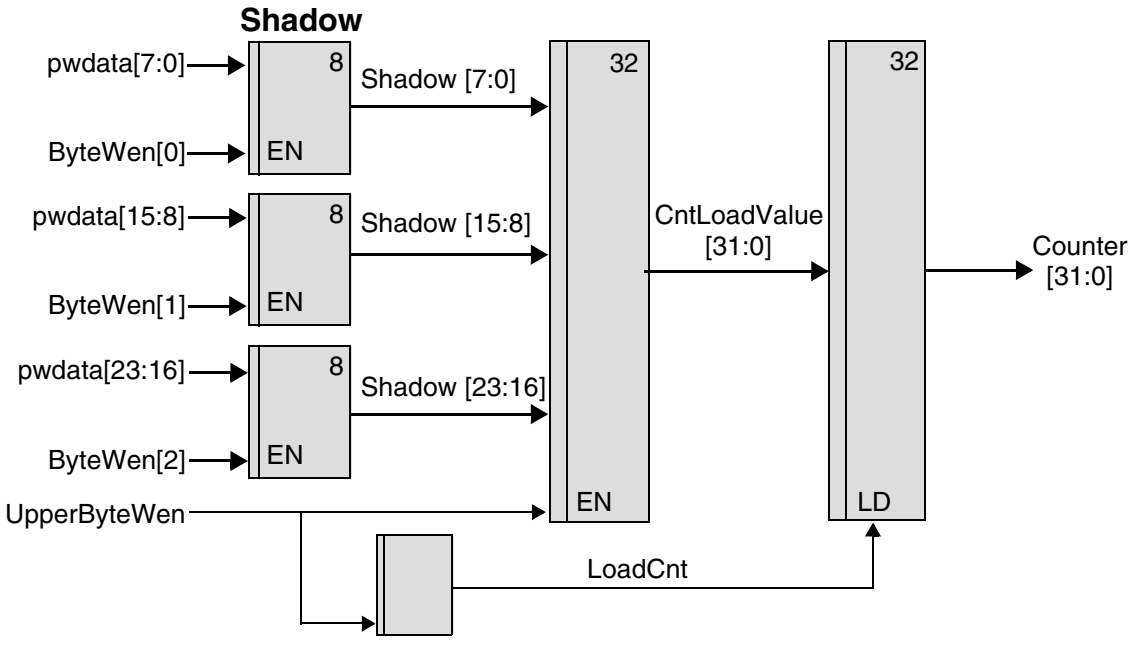
There are three relationship cases to be considered for the processor and peripheral clocks:

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

8.2.1.1 Identical Clocks

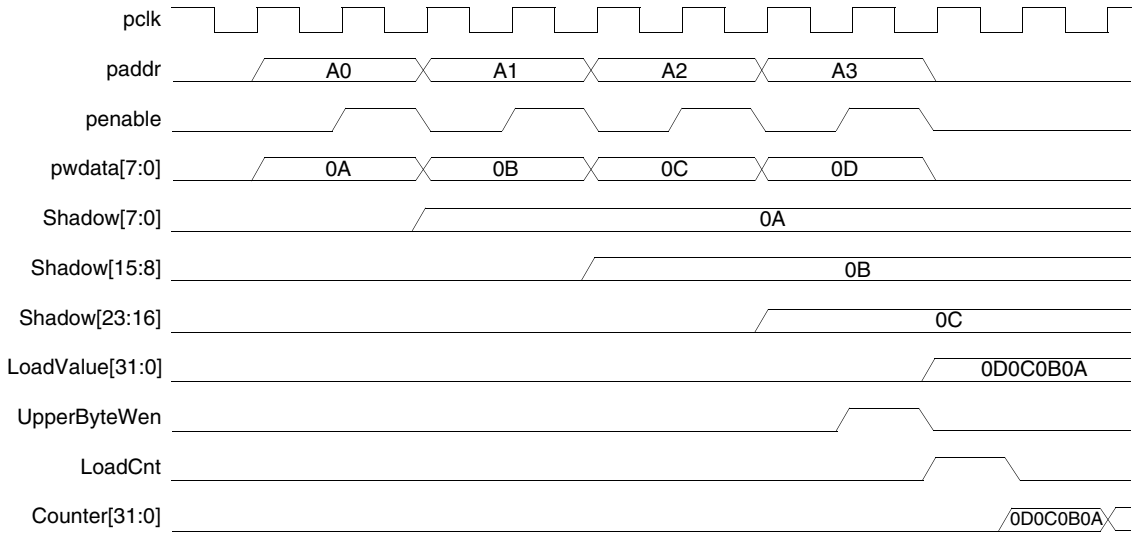
The following figure illustrates an RTL diagram for the circuitry required to implement the coherent write transaction when the APB bus clock and peripheral clocks are identical.

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.

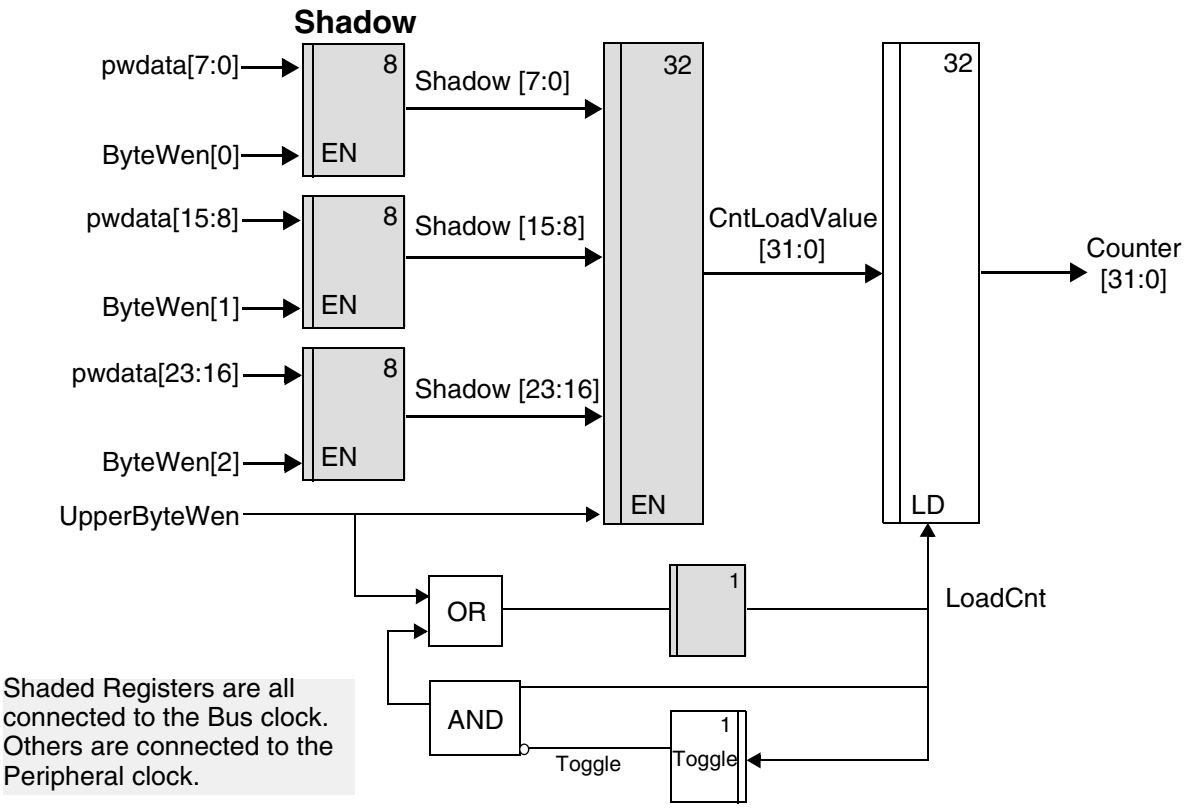
By using the shadow registers, the CntLoadValue is kept stable until it can be changed in one cycle. This allows the counter to be loaded in one access and the state of the counter is not affected by the latency in programming it. When there is a new value to be loaded into the counter initially, this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

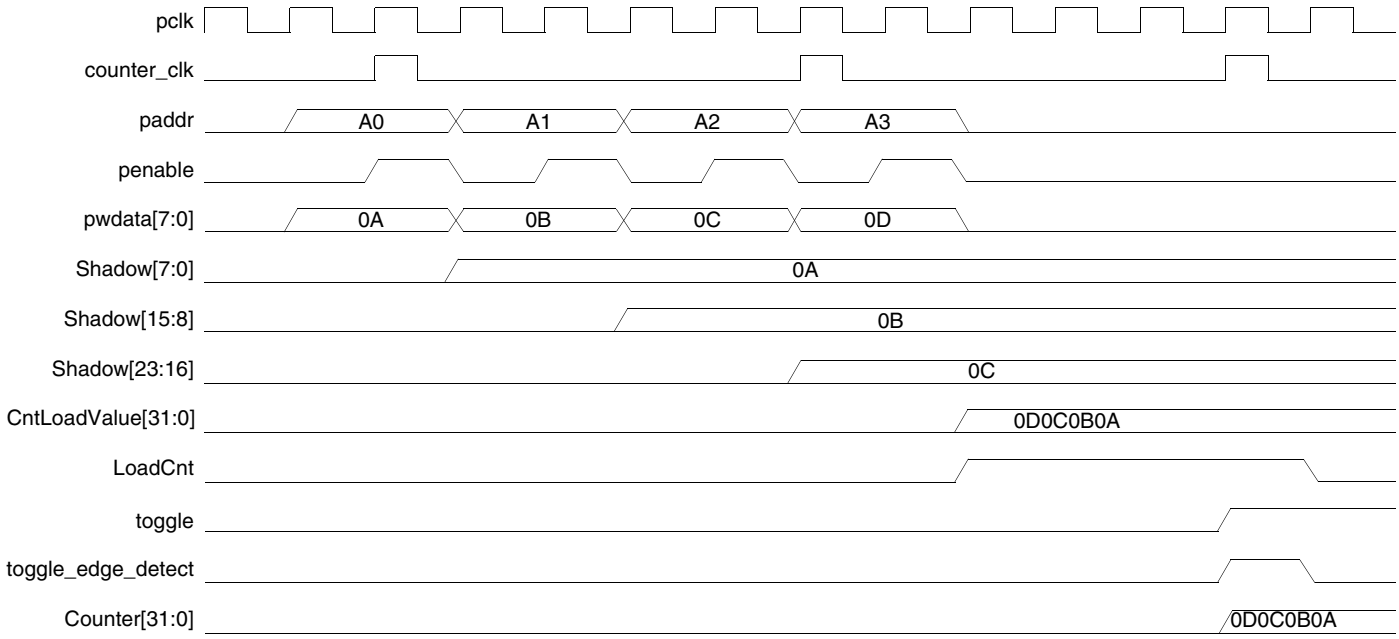
The following figure shows an RTL diagram for the circuitry required to implement the coherent write when the bus and peripheral clocks are synchronous.

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.

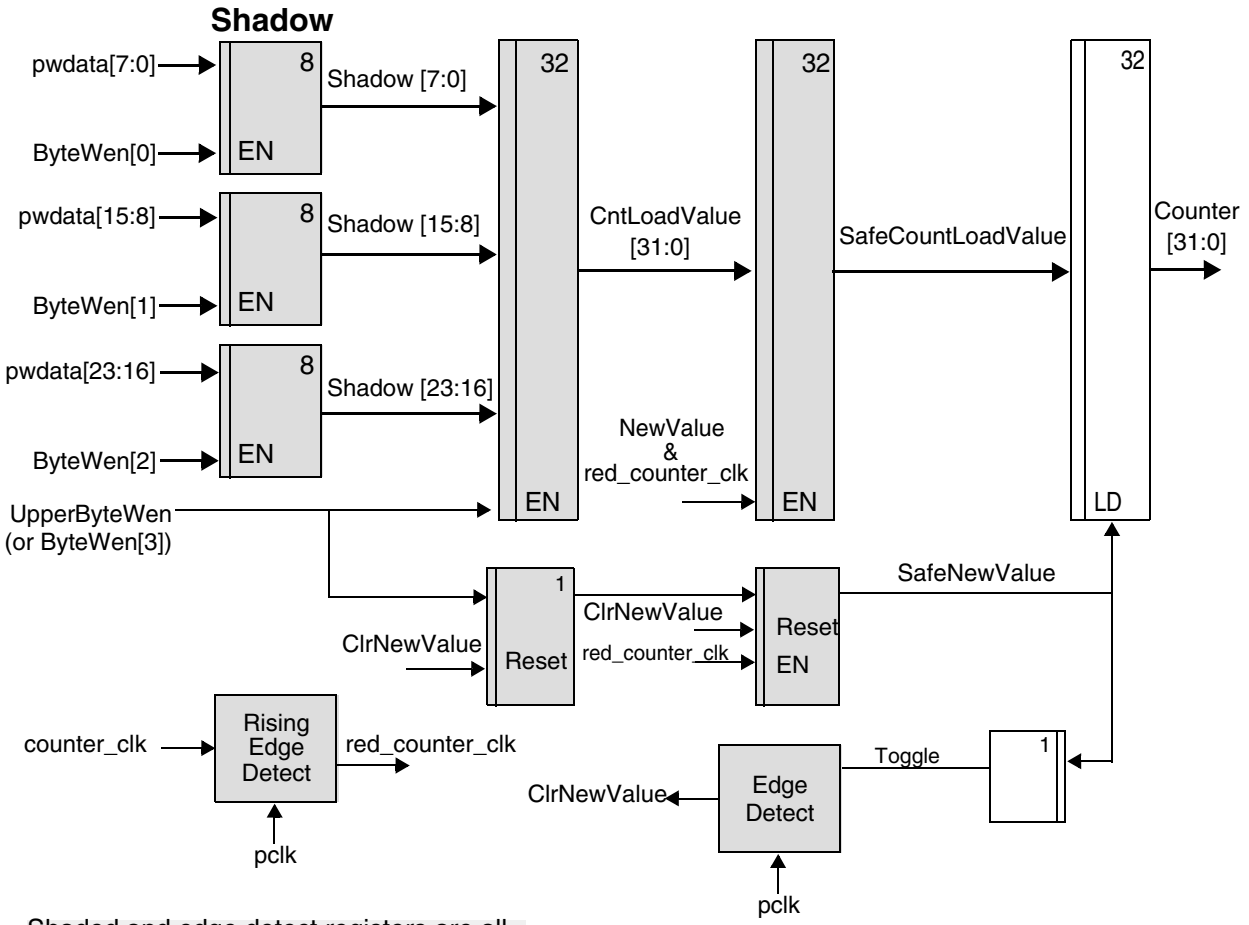
Figure 8-4 Coherent Loading – Synchronous Clocks



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



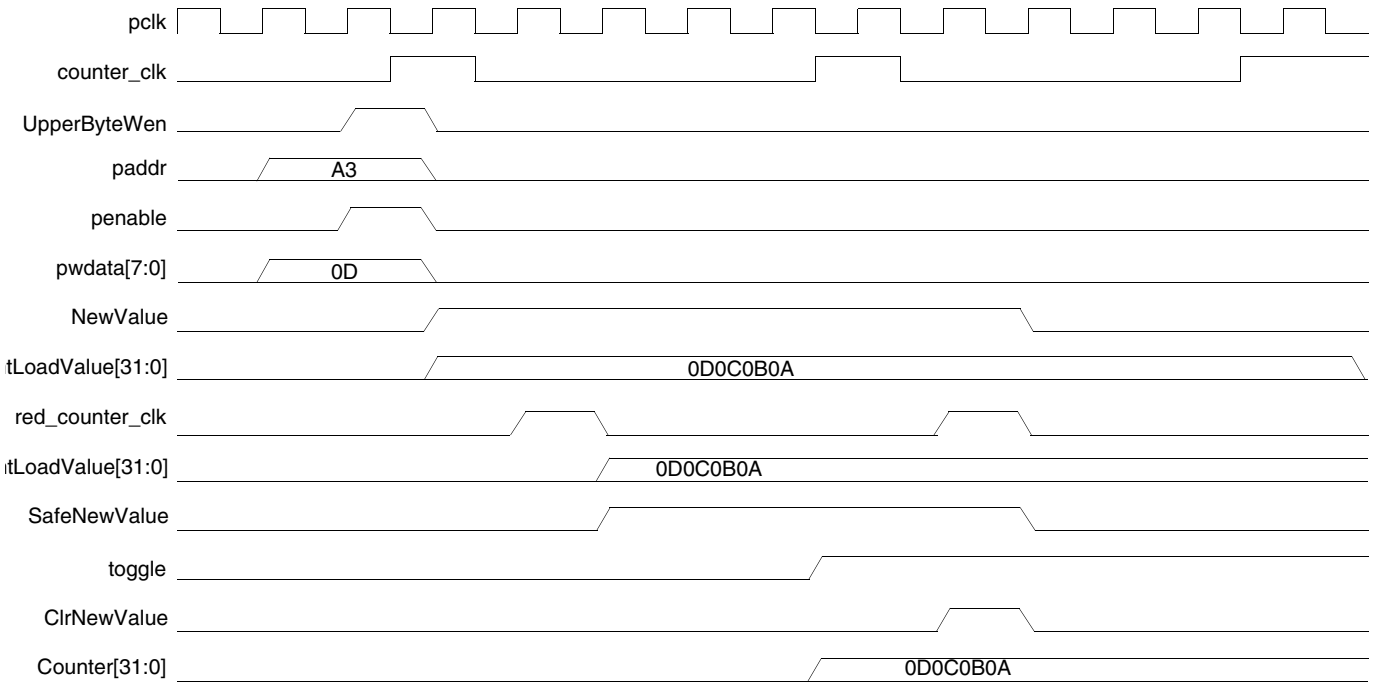
When the clocks are asynchronous, you need to transfer the contents of the register from one clock domain to another. It is not desirable to transfer the entire register through meta-stability registers, as coherency is not guaranteed with this method. The circuitry needed requires the processor clock 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 is also 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

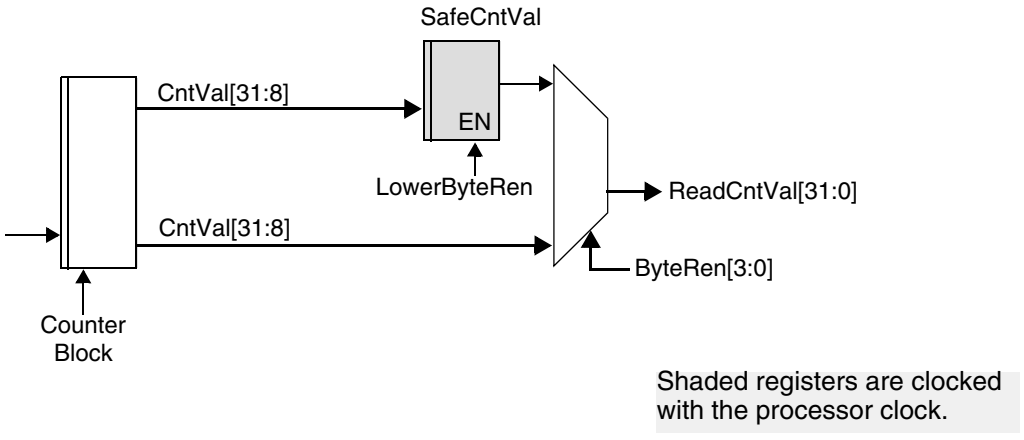
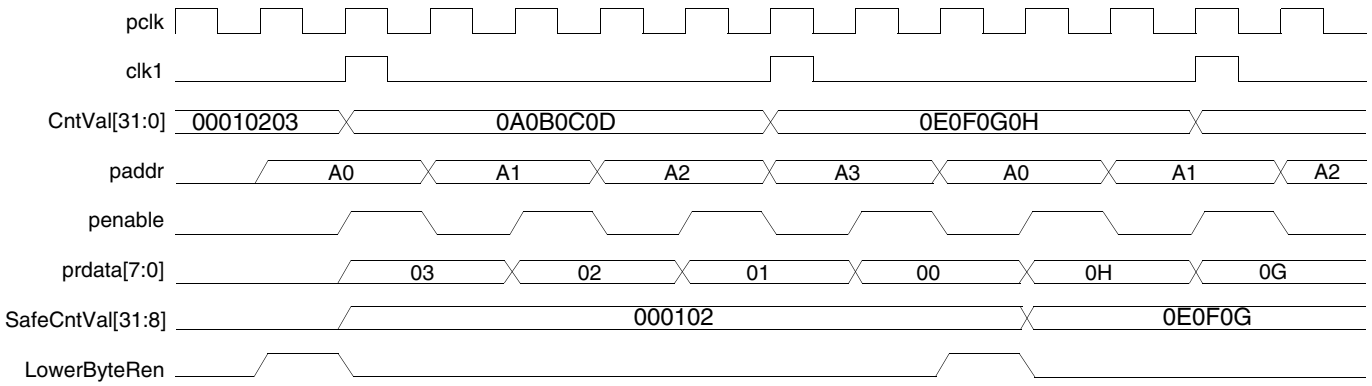


Figure 8-8 Coherent Registering – Synchronous Clocks



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.

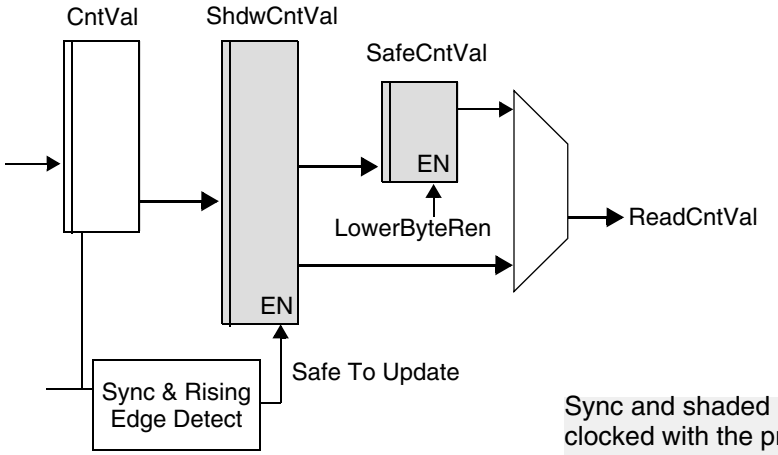
Note You must read LSB to MSB when the bus width is narrower than the counter width.

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 quasi-static signals in 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_timers.

8.4.1 Power Consumption, Frequency, and Area Results

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

Table 8-3 Synthesis Results for DW_apb_timers

Configuration	Operating Frequency	Gate Count	Power Consumption		TetraMax Coverage (%)		SpyGlass StuckAtCov (%)
			Static Power	Dynamic Power	StuckAtTest	Transition	
Default Configuration	pclk = 100 MHz timer_N_clk = 100MHz (N = 1, 2)	2084	5 nW	0.011 mW	99.82	99.89	99.9
Typical Configuration 1 APB_DATA_WIDTH = 32 NUM_TIMERS = 8 TIMER_HAS_TOGGLE_1 = 1 TIMER_HAS_TOGGLE_2 = 1 TIMER_HAS_TOGGLE_3 = 1 TIMER_HAS_TOGGLE_4 = 1 TIMER_HAS_TOGGLE_5 = 1 TIMER_HAS_TOGGLE_6 = 1 TIMER_HAS_TOGGLE_7 = 1 TIMER_HAS_TOGGLE_8 = 1 TIMER_WIDTH_1 = 32 TIMER_WIDTH_2 = 32 TIMER_WIDTH_3 = 32 TIMER_WIDTH_4 = 32	pclk = 100 MHz timer_N_clk = 100 MHz (N = 1 to 8)	30532	80 nW	0.255 mW	99.91	99.91	100

Table 8-3 Synthesis Results for DW_apb_timers (Continued)

Configuration	Operating Frequency	Gate Count	Power Consumption		TetraMax Coverage (%)		SpyGlass StuckAtCov (%)
			Static Power	Dynamic Power	StuckAtTest	Transition	
TIMER_WIDTH_5 = 32 TIMER_WIDTH_6 = 32 TIMER_WIDTH_7 = 32 TIMER_WIDTH_8 = 32 TIM_METASTABLE_1 = 1 TIM_METASTABLE_2 = 1 TIM_METASTABLE_3 = 1 TIM_METASTABLE_4 = 1 TIM_METASTABLE_5 = 1 TIM_METASTABLE_6 = 1 TIM_METASTABLE_7 = 1 TIM_METASTABLE_8 = 1 SLAVE_INTERFACE_TYPE = 2 SLVERR_RESP_EN = 1 TIM_0N100_PWM_MODE = 1 TIMER_0N100_PWM_HC_EN = 0 TIM_NEWMODE = 1 HC_PROT_LEVEL = 0							

Table 8-3 Synthesis Results for DW_apb_timers (Continued)

Configuration	Operating Frequency	Gate Count	Power Consumption		TetraMax Coverage (%)		SpyGlass StuckAtCov (%)
			Static Power	Dynamic Power	StuckAtTest	Transition	
Typical Configuration 2 APB_DATA_WIDTH = 8 NUM_TIMERS = 8 TIMER_HAS_TOGGLE_1 = 1 TIMER_HAS_TOGGLE_2 = 1 TIMER_HAS_TOGGLE_3 = 1 TIMER_HAS_TOGGLE_4 = 1 TIMER_HAS_TOGGLE_5 = 1 TIMER_HAS_TOGGLE_6 = 1 TIMER_HAS_TOGGLE_7 = 1 TIMER_HAS_TOGGLE_8 = 1 TIMER_WIDTH_1 = 32 TIMER_WIDTH_2 = 32 TIMER_WIDTH_3 = 32 TIMER_WIDTH_4 = 32 TIMER_WIDTH_5 = 32 TIMER_WIDTH_6 = 32 TIMER_WIDTH_7 = 32 TIMER_WIDTH_8 = 32 TIM_METASTABLE_1 = 1 TIM_METASTABLE_2 = 1 TIM_METASTABLE_3 = 1 TIM_METASTABLE_4 = 1 TIM_METASTABLE_5 = 1 TIM_METASTABLE_6 = 1 TIM_METASTABLE_7 = 1 TIM_METASTABLE_8 = 1	pclk = 100 MHz timer_N_clk = 100 MHz (N = 1 to 8)	7914	20 nW	0.052 mW	99.82	99.91	99.9

Table 8-3 Synthesis Results for DW_apb_timers (Continued)

Configuration	Operating Frequency	Gate Count	Power Consumption		TetraMax Coverage (%)		SpyGlass StuckAtCov (%)
			Static Power	Dynamic Power	StuckAtTest	Transition	
TIM_PULSE_EXTD_1 = 3 TIM_PULSE_EXTD_2 = 3 TIM_PULSE_EXTD_3 = 3 TIM_PULSE_EXTD_4 = 3 TIM_PULSE_EXTD_5 = 3 TIM_PULSE_EXTD_6 = 3 TIM_PULSE_EXTD_7 = 3 TIM_PULSE_EXTD_8 = 3 SLAVE_INTERFACE_TYPE = 1 SLVERR_RESP_EN = 0 TIM_0N100_PWM_MODE = 0 TIMER_0N100_PWM_HC_EN = 0 TIM_NEWMODE = 0 HC_PROT_LEVEL = 0							

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_timers.

Table A-1 BCM Library Components

BCM Module Name	BCM Description	DWBB Equivalent
DW_apb_timers_bcm21	Single Clock Data Bus Synchronizer	DW_sync
DW_apb_timers_bcm36_nhs	Bus Delay Component	--

A.2 Synchronizer Methods

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

This section contains the following sections:

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_timers

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_timers 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_timers

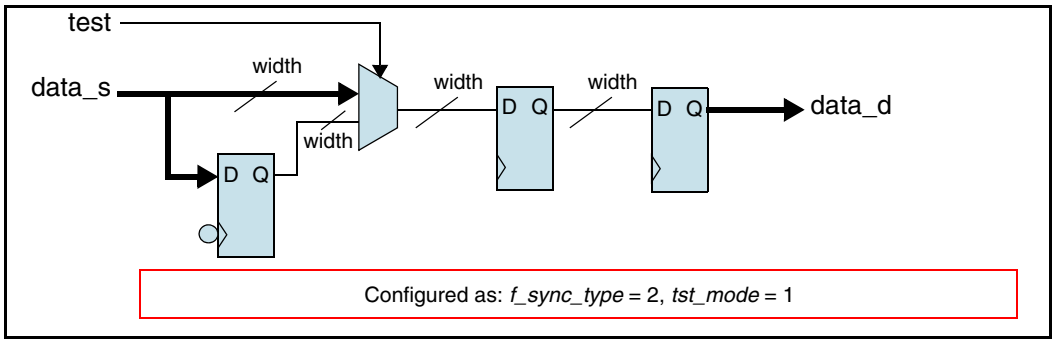
Synchronizer module file	Synchronizer Type and Number
DW_apb_timers_bcm21.v	Synchronizer 1: Simple Multiple Register Synchronizer

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 (DW_apb_timers)

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 Edges)



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
TIM_ADDR_SLICE_LHS	7
TIM_COMBINED	1
TIM_INDIVIDUAL	0
TIM_NEWMODE_VAL	{(TIM_NEWMODE == 1)}
TIM_VERSION_ID	32'h3231332a

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.