



DesignWare DW_apb_timers Databook

DW_apb_timers – **Product Code**

Copyright Notice and Proprietary Information Notice

© 2015 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

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 <http://www.synopsys.com/Company/Pages/Trademarks.aspx>.

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

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Contents

Revision History	7
Preface	9
Chapter 1	
Product Overview	13
1.1 DesignWare System Overview	13
1.2 General Product Description	15
1.2.1 DW_apb_timers Block Diagram	15
1.3 Features	15
1.4 Standards Compliance	16
1.5 Verification Environment Overview	16
1.6 Licenses	16
1.7 Where To Go From Here	16
Chapter 2	
Building and Verifying a Component or Subsystem	17
2.1 Setting up Your Environment	17
2.2 Overview of the coreConsultant Configuration and Integration Process	18
2.2.1 coreConsultant Usage	18
2.2.2 Configuring the DW_apb_timers within coreConsultant	20
2.2.3 Creating Gate-Level Netlists within coreConsultant	20
2.2.4 Verifying the DW_apb_timers within coreConsultant	20
2.2.5 Running Leda on Generated Code with coreConsultant	20
2.2.6 Running SpyGlass® Lint and SpyGlass® CDC	20
2.3 Overview of the coreAssembler Configuration and Integration Process	24
2.3.1 coreAssembler Usage	24
2.3.2 Configuring the DW_apb_timers within a Subsystem	27
2.3.3 Creating Gate-Level Netlists within coreAssembler	27
2.3.4 Verifying the DW_apb_timers within coreAssembler	28
2.3.5 Running Leda on Generated Code with coreAssembler	28
2.3.6 Running Spyglass on Generated Code with coreAssembler	28
2.4 Database Files	28
2.4.1 Design/HDL Files	28
2.4.2 Synthesis Files	29
2.4.3 Verification Reference Files	30
Chapter 3	
Functional Description	31
3.1 Timer Operation	31

3.2 DW_apb_timers Usage Flow	31
3.3 DW_apb_timers Configuration	32
3.3.1 Choosing the Number of Timers	32
3.3.2 Enabling and Disabling a Timer	33
3.3.3 Configuring the Width of a Timer	33
3.3.4 Loading a Timer Countdown Value	33
3.3.5 Working with Interrupts	34
3.3.6 Controlling Clock Boundaries and Metastability	36
3.3.7 Generating Toggled Outputs	40
3.3.8 Timer Pause Mode	40
3.4 Design For Test	41
Chapter 4	
Parameters	43
4.1 Parameter Descriptions	43
Chapter 5	
Signal Descriptions	47
Chapter 6	
Registers	53
6.1 Bus Interface	53
6.2 Register Memory Map	54
6.3 Register and Field Descriptions	56
6.3.1 TimerNLoadCount	56
6.3.2 TimerNLoadCount2	57
6.3.3 TimerNCurrentValue	58
6.3.4 TimerNControlReg	59
6.3.5 TimerNEOI	60
6.3.6 TimerNIntStatus	61
6.3.7 TimersIntStatus	62
6.3.8 TimersEOI	63
6.3.9 TimersRawIntStatus	64
6.3.10 TIMERS_COMP_VERSION	65
Chapter 7	
Internal Parameter Descriptions	67
Chapter 8	
Programming Considerations	69
Chapter 9	
Verification	71
9.1 Overview of Vera Tests	71
9.2 Overview of DW_apb_timers Testbench	73
Chapter 10	
Integration Considerations	75
10.1 Reading and Writing from an APB Slave	75
10.1.1 Reading From Unused Locations	75
10.1.2 32-bit Bus System	76

10.1.3 16-bit Bus System	77
10.1.4 8-bit Bus System	77
10.2 Write Timing Operation	78
10.3 Read Timing Operation	79
10.4 Accessing Top-level Constraints	79
10.5 Coherency	80
10.5.1 Writing Coherently	80
10.5.2 Reading Coherently	86
10.6 Performance	89
10.7 Area	89
10.7.1 Power Consumption	93
Appendix A	
Synchronizer Methods	97
A.1 Synchronizers Used in DW_apb_timers	98
A.2 Synchronizer 1: Simple Double Register Synchronizer (DW_apb_timers)	99
Appendix B	
Glossary	101
Index	105

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
2.10a	June 2015	<ul style="list-style-type: none"> Modified default value for TimerNCurrentValue field of the Timer N register Included section “Running SpyGlass® Lint and SpyGlass® CDC” on page 20 Included section “Running Spyglass on Generated Code with coreAssembler” on page 28 Chapter 5, “Signal Descriptions” auto-extracted from the RTL Added Chapter 7, “Internal Parameter Descriptions” Added Appendix A, “Synchronizer Methods”
2.09a	June 2014	<ul style="list-style-type: none"> Version change for 2014.06a release Updated the section 3.3.6 Controlling Clock Boundaries and Metastability Added: <ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> Version change for 2013.05a release Updated the template
2.08a	Sep 2012	Added the product code on the cover and in Table 1-1
2.08a	Jun 2012	Version change for 2012.06a release
2.06c	Mar 2012	Version change for 2012.03a release
2.06b	Nov 2011	Version change for 2011.11a release
2.06a	Oct 2011	Version change for 2011.10a release
2.05a	Jun 2011	<ul style="list-style-type: none"> Updated system diagram in Figure 1-1 Enhanced “Related Documents” section in Preface
2.05a	Sept 2010	Corrected names of include files and vcs command used for simulation

(Continued)

Version	Date	Description
2.03a	Dec 2009	Updated databook to new template for consistency with other IIP/VIP/PHY databooks.
2.03a	July 2009	<ul style="list-style-type: none"> 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	Mar 2009	Corrected TimersIntStatus register illustration
2.03a	Oct 2008	Version change for 2008.10a release
2.02e	Jun 2008	Version change for 2008.06a release
2.02d	Jan 2008	<ul style="list-style-type: none"> Updated to revised installation guide and consolidated release notes Changed references of “Designware AMBA” to simply “DesignWare”
2.02d	Sept 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.

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, “[Building and Verifying a Component or Subsystem](#)” introduces you to using the DW_apb_timers within the coreAssembler and coreConsultant tools.
- Chapter 3, “[Functional Description](#)” describes the functional operation of the DW_apb_timers.
- Chapter 4, “[Parameters](#)” identifies the configurable parameters supported by the DW_apb_timers.
- Chapter 5, “[Signal Descriptions](#)” provides a list and description of the DW_apb_timers signals.
- Chapter 6, “[Registers](#)” describes the programmable registers of the DW_apb_timers.
- Chapter 7, “[Internal Parameter Descriptions](#)” provides a list of internal parameter descriptions that might be indirectly referenced in expressions in the Signals chapter.
- Chapter 8, “[Programming Considerations](#)” provides information needed to program the configured DW_apb_timers.
- Chapter 9, “[Verification](#)” provides information on verifying the configured DW_apb_timers.
- Chapter 10, “[Integration Considerations](#)” includes information you need to integrate the configured DW_apb_timers into your design.
- [Appendix A, “Synchronizer Methods”](#) documents the synchronizer methods (blocks of synchronizer functionality) used in DW_apb_timers to cross clock boundaries.
- [Appendix B, “Glossary”](#) provides a glossary of general terms.

Related Documentation

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

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

Web Resources

- DesignWare IP product information: <http://www.designware.com>
- Your custom DesignWare IP page: <http://www.mydesignware.com>
- Documentation through SolvNet: <http://solvnet.synopsys.com> (Synopsys password required)
- Synopsys Common Licensing (SCL): <http://www.synopsys.com/keys>

Customer Support

To obtain support for your product:

- First, prepare the following debug information, if applicable:
 - For environment setup problems or failures with configuration, simulation, or synthesis that occur within coreConsultant or coreAssembler, use the following menu entry:
File > Build Debug Tar-file

Check all the boxes in the dialog box that apply to your issue. This menu entry gathers all the Synopsys product data needed to begin debugging an issue and writes it to the file `<core tool startup directory>/debug.tar.gz`.
 - 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
- Then, contact Support Center, with a description of your question and supplying the above information, using one of the following methods:
 - *For fastest response*, use the SolvNet website. If you fill in your information as explained below, your issue is automatically routed to a support engineer who is experienced with your product. The **Sub Product** entry is critical for correct routing.
Go to <http://solvnet.synopsys.com/EnterACall> and click on the link to enter a call. Provide the requested information, including:
 - **Product:** DesignWare Library IP
 - **Sub Product:** AMBA
 - **Tool Version:** *product version number*
 - **Problem Type:**

- **Priority:**
- **Title:** DW_apb_timers
- **Description:** For simulation issues, include the timestamp of any signals or locations in waveforms that are not understood

After creating the case, attach any debug files you created in the previous step.

- 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 name, Sub Product name, and Tool Version number in your e-mail (as identified above) 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 in the previous step.
- 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:
<http://www.synopsys.com/Support/GlobalSupportCenters>

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.

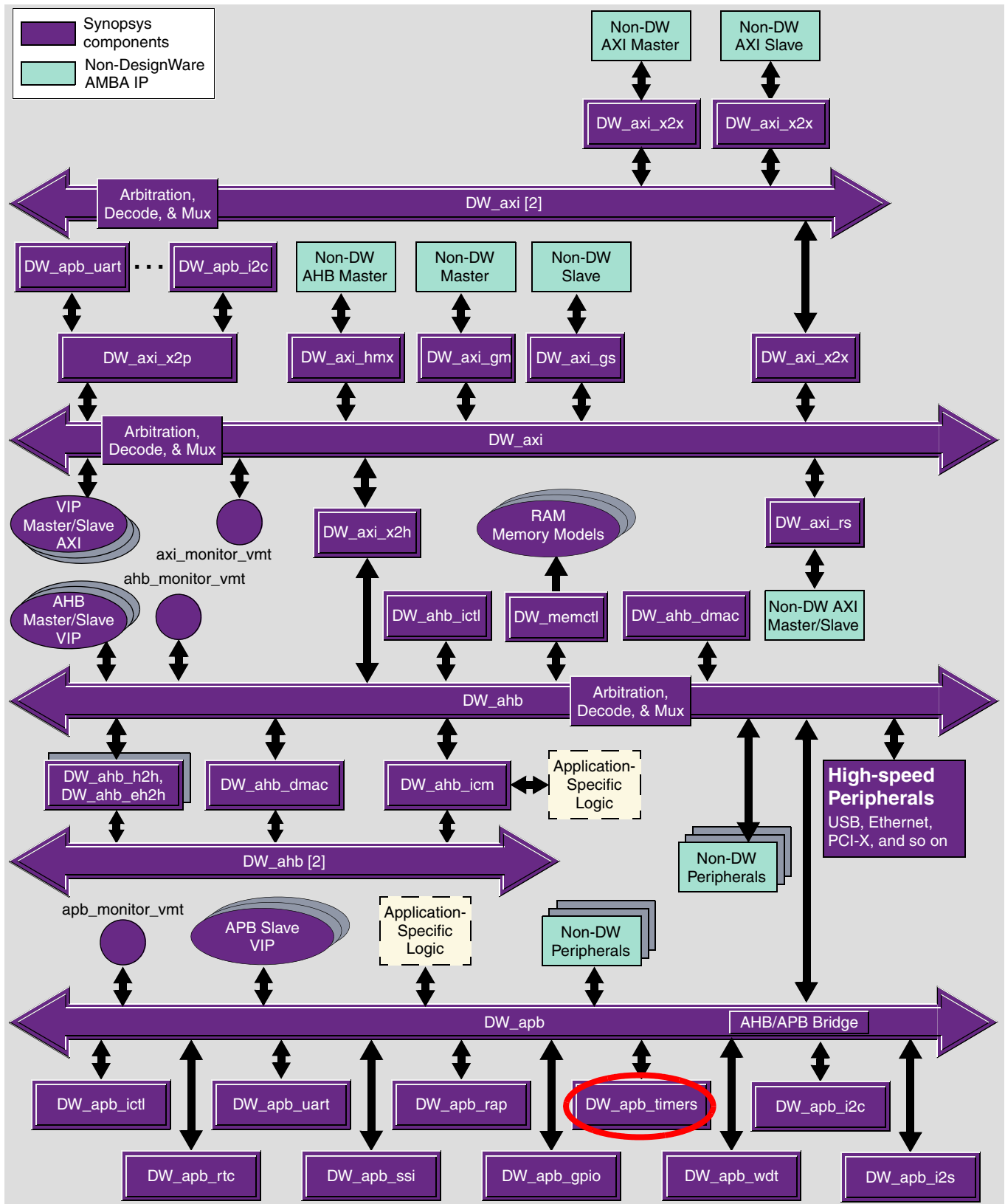
1.1 DesignWare System Overview

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. In order to display the databook for a DW_* component, click on the corresponding component object in the illustration.

**Attention**

Links resolve only if you are viewing this databook from your \$DESIGNWARE_HOME tree, and to only those components that are installed in the tree.

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](#).

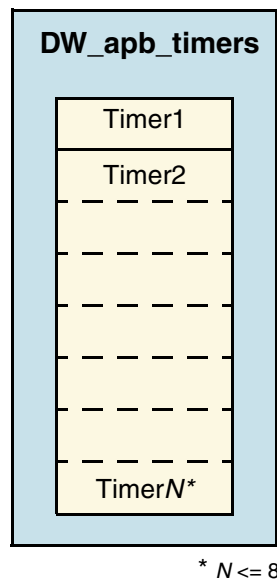
1.2 General Product Description

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:

- 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

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

1.4 Standards Compliance

The DW_apb_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 71.

1.6 Licenses

Before you begin using the DW_apb_timers, you must have a valid license. For more information, refer to [“Licenses”](#) in the *DesignWare Synthesizable Components for AMBA 2/AMBA 3 AXI Installation Guide*.

1.7 Where To Go From Here

At this point, you may want to get started working with the DW_apb_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, refer to [Guide to coreTools Documentation](#).

For more information about configuring, synthesizing, and verifying just your DW_apb_timers component, refer to [“Overview of the coreConsultant Configuration and Integration Process”](#) on page 18.

For more information about implementing your DW_apb_timers component within a DesignWare subsystem using coreAssembler, refer to [“Overview of the coreAssembler Configuration and Integration Process”](#) on page 24.

Building and Verifying a Component or Subsystem

DesignWare Synthesizable IP (SIP) components for AMBA 2 and AMBA 3 AXI are packaged using Synopsys coreTools, which enable the user to configure, synthesize, and run simulations on a single SIP title, or to build a configured AMBA subsystem. You do this by generating a workspace view using one of the following coreTools applications:

- coreConsultant – Used for configuration, RTL generation, synthesis, and execution of packaged verification for a single SIP title. The [coreConsultant User Guide](#) provides complete information on using coreConsultant.
- coreAssembler – Used for building and configuration of a subsystem that connects multiple SIP titles, RTL generation, synthesis, and creation of a template subsystem testbench. The [coreAssembler User Guide](#) provides complete information on using coreAssembler.

A workspace is your working version of a DesignWare SIP component or subsystem. In fact, you can create several workspaces to experiment with different design alternatives.



Hint

If you are unfamiliar with coreTools—which is comprised of the coreAssembler, coreConsultant, and coreBuilder tools—you can go to [Using DesignWare Library IP in coreAssembler](#) to “get started” learning how to work with DesignWare SIP components.

2.1 Setting up Your Environment

The DW_apb_timers is included in a release of DesignWare SIP components. It is assumed that you have already downloaded and installed the release. If you have not, you can download and install the latest versions of required tools using the [DesignWare Synthesizable Components for AMBA 2/AMBA 3 AXI Installation Guide](#).

You also need to set up your environment correctly using specific environment variables, such as DESIGNWARE_HOME, VERA_HOME, PATH, and SYNOPSIS. If you are not familiar with these requirements and the necessary licenses, refer to [“Setting up Your Environment”](#) in the *DesignWare Synthesizable Components for AMBA 2/AMBA 3 AXI Installation Guide*.

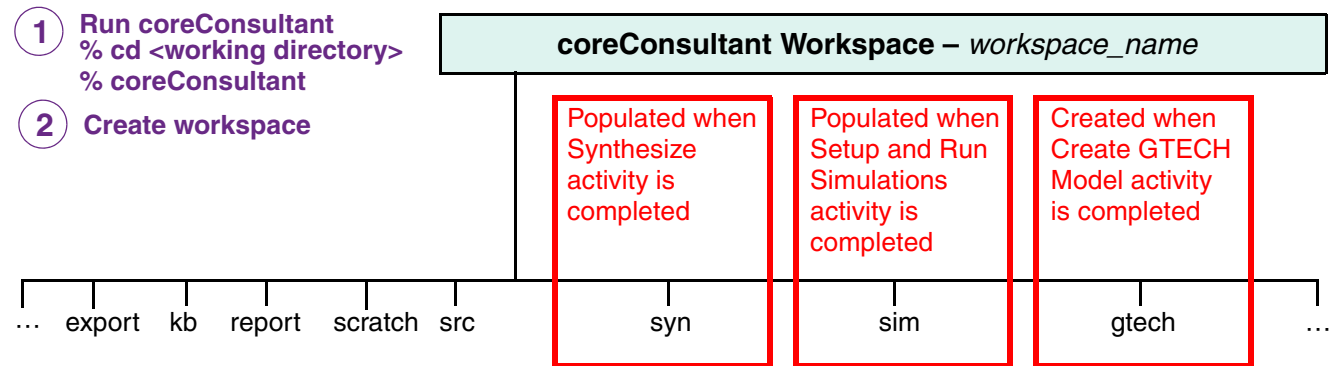
2.2 Overview of the coreConsultant Configuration and Integration Process

Once you have correctly downloaded and installed a release of DesignWare SIP components and then set up your environment, you can begin work on the DW_apb_timers using coreConsultant.

2.2.1 coreConsultant Usage

Figure 2-1 illustrates some general directories and files in a coreConsultant workspace.

Figure 2-1 coreConsultant Usage Flow



3 Use coreConsultant to create, synthesize, and verify your component

Table 2-1 provides a description of the implementation workspace directory and subdirectories.

Table 2-1 coreConsultant Implementation Workspace Directory Contents

Directory/Subdirectory	Description
auxiliary	Scripts and text files used by coreConsultant. Generated upon first creating workspace.
custom	Contains RTL preprocessor scripts. Generated during Specify Configuration activity.
doc	Contains local copies of component-specific databooks. Generated upon first creating workspace.
export	Contains files used to integrate results from the completed source configuration and synthesis activities into your design (outside coreConsultant). Generated upon first creating workspace; populated during Specify Configuration activity.
gtech	Contains synthesis scripts and output netlists from gtech generation; also used for RTL simulation of encrypted source code. Generated during Generate GTECH Model activity.

Table 2-1 coreConsultant Implementation Workspace Directory Contents (Continued)

Directory/Subdirectory	Description
kb	Contains knowledge base information used by coreConsultant. These are binary files containing the state of the design. Generated upon first creating workspace; populated and updated throughout activities.
leda	Contains Leda configuration files for the component. Generated upon first creating workspace; updated during Run Leda Coding Checker activity.
report	Contains all of the reports created by coreConsultant during build, configuration, test and synthesis phases. An index.html file in this directory links to many of these generated reports. Generated upon first creating workspace; populated and updated throughout activities.
scratch	Contains temp files used during the coreConsultant processes. Generated upon first creating workspace; populated and updated throughout activities.
sim	Contains test stimulus and output files. Generated upon first creating workspace; updated during Setup and Run Simulations activity.
spyglass	Contains SpyGlass Lint and CDC configuration files for the component. Generated upon first SpyGlass run; updated during Run Spyglass RTL Checker activity.
src	Includes the top-level RTL file, <i>design_name.v</i> . If you have a source license, this will contain plain-text RTL; if you only have a designware license, this will contain encrypted RTL. Generated upon first creating workspace; populated during Specify Configuration activity.
syn	Contains synthesis files for the component. Generated upon first creating workspace; updated during Synthesis activity and Formal Verification activity.
tcl	Contains synthesis intent scripts. Generated upon first creating workspace.

For details on some key files created during coreConsultant activities, refer to [“Database Files”](#) on page 28.

For information on using coreConsultant, refer to the [coreConsultant User Guide](#).

2.2.2 Configuring the DW_apb_timers within coreConsultant

The “[Parameters](#)” chapter on [page 43](#) describes the DW_apb_timers hardware configuration parameters that you configure using the coreConsultant GUI.

The “Creating the RTL View of a Core” chapter in the [coreConsultant User Guide](#) discusses how to specify a configuration for an individual component like the DW_apb_timers.

2.2.3 Creating Gate-Level Netlists within coreConsultant

The “Creating the Gate-Level Netlist for a Core” chapter in the [coreConsultant User Guide](#) discusses how to create a translation of the RTL view into a technology-specific netlist for an individual component like the DW_apb_timers.

2.2.4 Verifying the DW_apb_timers within coreConsultant

The “[Verification](#)” chapter on [page 71](#) provides an overview of the testbench available for DW_apb_timers verification using the coreConsultant GUI.

The “Verifying Your Implementation” chapter in the [coreConsultant User Guide](#) discusses how to simulate an individual component like the DW_apb_timers.

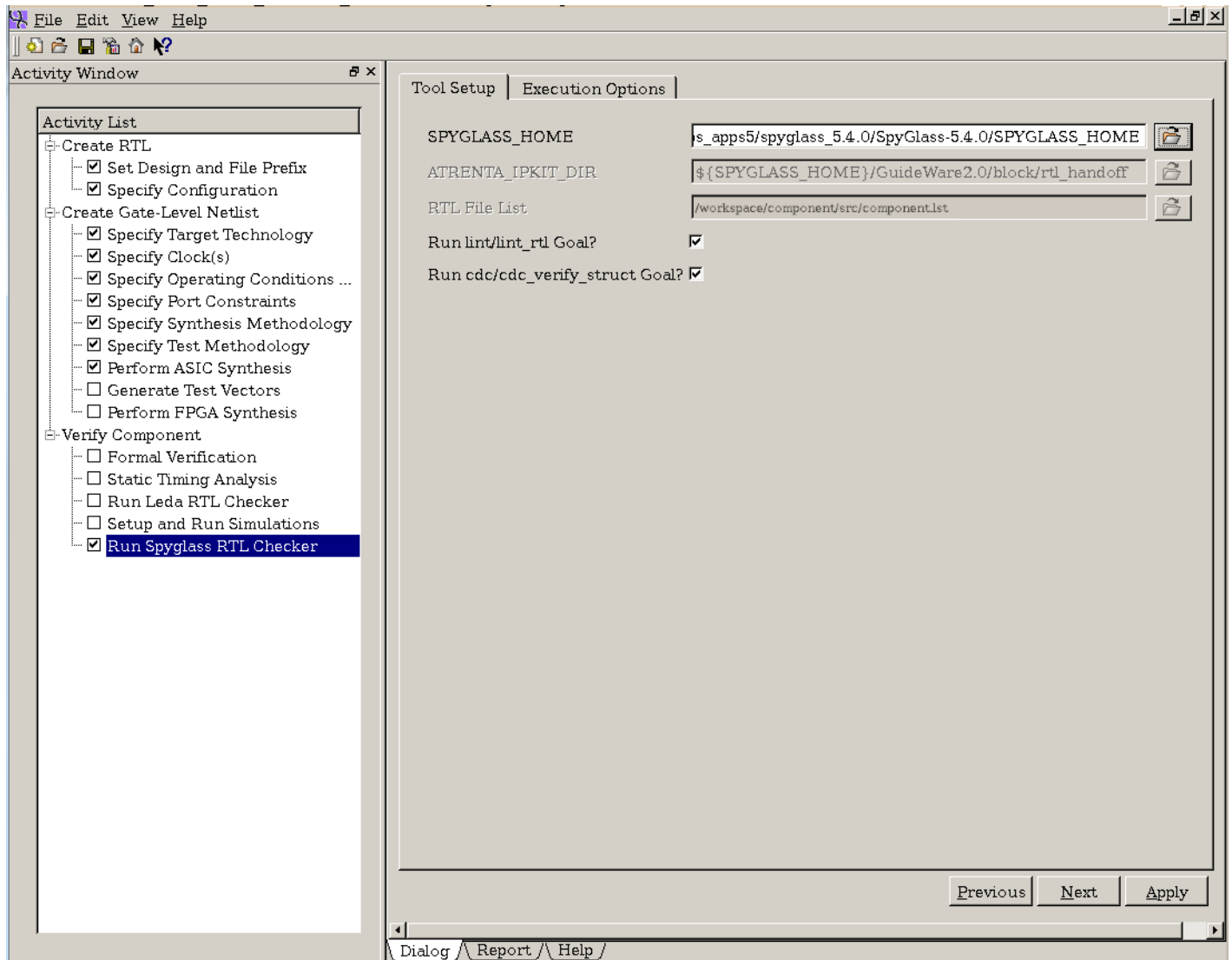
2.2.5 Running Leda on Generated Code with coreConsultant

When you select **Verify Component > Run Leda Coding Checker** from the Activity List, the corresponding Activity View appears. In this Activity View you select rules configuration file and define Leda command line switches.

2.2.6 Running SpyGlass® Lint and SpyGlass® CDC

This section discusses the procedure to run SpyGlass Lint and SpyGlass CDC.

[Figure 2-2](#) shows the coreConsultant GUI in which you run Lint and CDC goals.

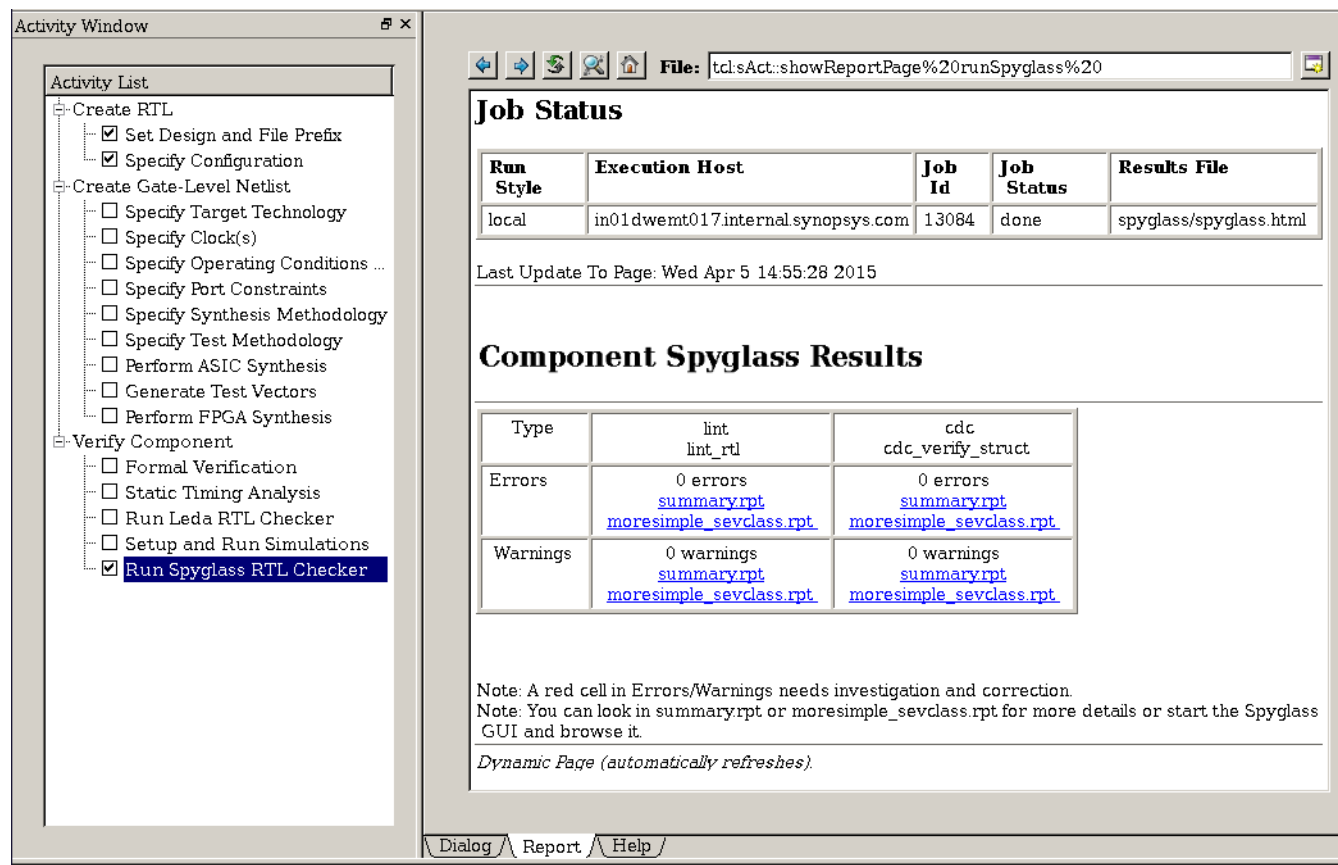
Figure 2-2 SpyGlass Options in coreConsultant

The SpyGlass flow in coreConsultant runs Guideware 2.0 rules for block/rtl_handoff. Within the block/rtl_handoff, only lint/lint_rtl and cdc/cdc_verify_struct goals are run.

In [Figure 2-2](#), select the type of run goals. You can select either Lint run goal or CDC run goal, or both Lint and CDC run goals. By default, both Lint and CDC are selected.

When the Lint and/or CDC is run, the results are available in the Report tab. Errors (if any) are displayed with a red colored cell and warnings (if any) are displayed in yellow colored cell, as shown in [Figure 2-3](#).

Figure 2-3 coreConsultant SpyGlass Report Summary



- 2.2.6.1 Fixed Settings
- The settings are fixed (hardcoded) when you run SpyGlass in coreConsultant.
- 2.2.6.2 SpyGlass Lint

Table 2-2 lists the SpyGlass Link waiver files that are used by the coreConsultant tool.

Table 2-2 Waiver Files for Sypglass Lint

File Name	Description
<configured_workspace>/spyglass/spyglass_design_specific_waivers.swl	These are DW_apb_timers design-specific rule waivers. This file contains Lint waivers for DW_apb_timers (if applicable). The reason for each of the waivers (if any) are included as comments in the file.
<configured_workspace>/spyglass/spyglass_engineering_council_waivers.swl	This file contains rules that Synopsys waives for its IPs.

2.2.6.3 SpyGlass CDC

To define the SpyGlass CDC constraints, it is important to understand the reset and clock logic used in DW_apb_timers. For information on reset and clock logic, refer [“Functional Description”](#) on page 31 and [“Signal Descriptions”](#) on page 47.

2.2.6.3.1 CDC Files

[Table 2-3](#) summarizes files for SpyGlass CDC used by coreConsultant.

Table 2-3 Waiver Files for Sypglass CDC

File Name	Description
<code><configured_workspace>/spyglass/manual.sgcd</code>	These are the constraints pertaining to a given mode.
<code><configured_workspace>/spyglass/ports.sgcd</code>	These are the list of I/O signals and their respective clocks.
<code><configured_workspace>/spyglass/spyglass_design_specific_waivers.swl</code>	These are DW_apb_timers design-specific rule waivers. This file contains CDC waivers for DW_apb_timers (if applicable). The reason for each of the waivers (if any) are included as comments in the file.
<code><configured_workspace>/spyglass/spyglass_engineering_council_waivers.swl</code>	These are rules that Synopsys waives for its IPs.

2.2.6.3.2 CDC Path Debug Using the SpyGlass GUI

For debugging the CDC path, it is necessary to run SpyGlass in interactive mode in the configured workspace. To invoke the SpyGlass GUI and to run CDC, complete the following steps:

1. Go to the `<configured_workspace>/spyglass` directory.
2. Issue `./sh.spyglass` to start the spyGlass GUI or issue `./sh.spyglass -batch` to start the SpyGlass in batch mode.
3. In the SpyGlass GUI, the Goal Setup window opens by default.
4. Uncheck the `lint_rtl` option and click the **Selected Goal (s)** button.
5. After the CDC run is complete, the Analyze Results window displays the results.

Navigate to and select the relevant errors to open a schematic for analysis.

2.3 Overview of the coreAssembler Configuration and Integration Process

Once you have correctly downloaded and installed a release of DesignWare SIP components and then set up your environment, you can begin work on your DesignWare subsystem with coreAssembler.

2.3.1 coreAssembler Usage

Figure 2-4 illustrates some general directories and files in a coreAssembler workspace.

Figure 2-4 coreAssembler Usage Flow

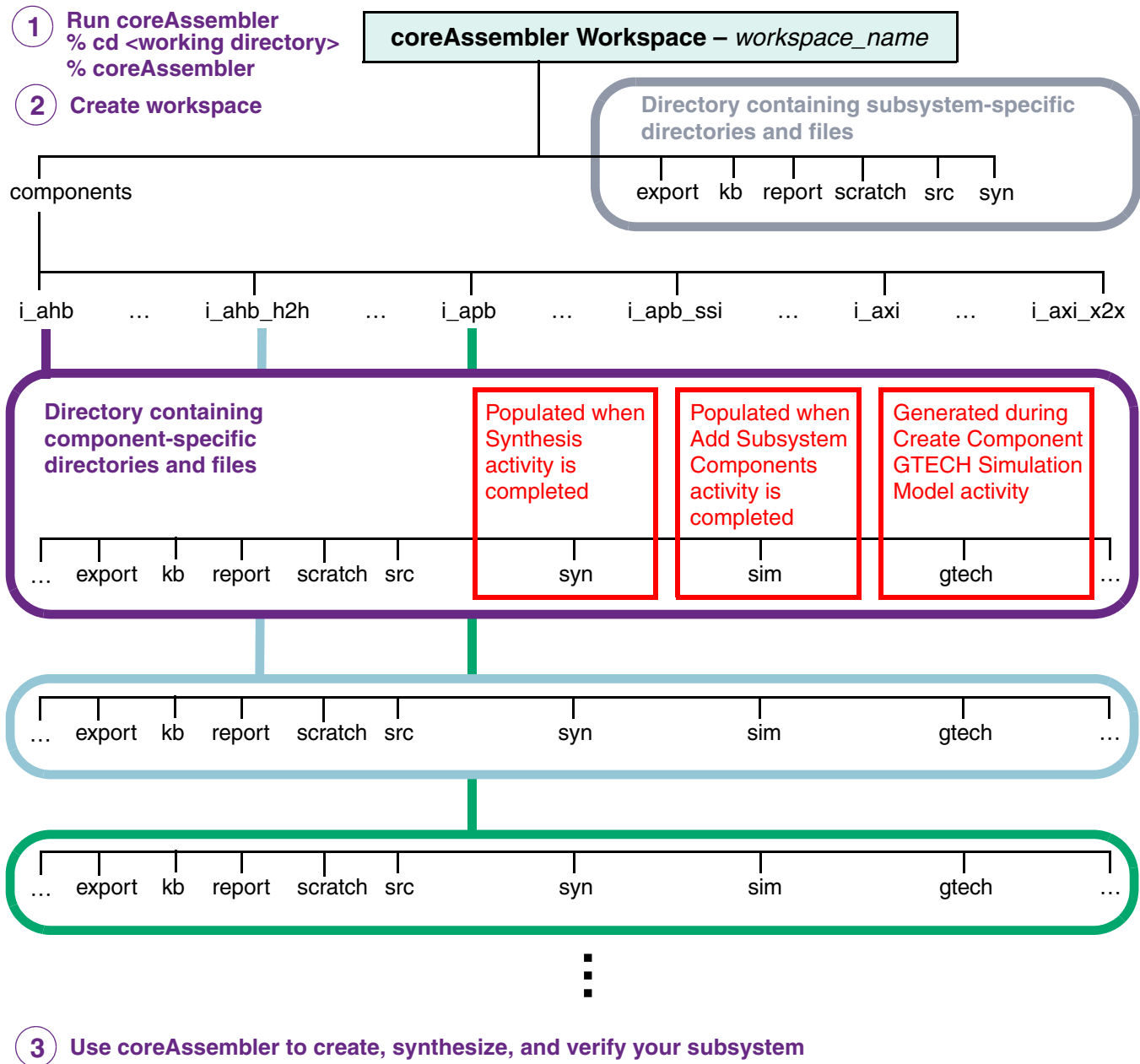


Table 2-4 provides a description of the implementation workspace directory and subdirectories.

Table 2-4 coreAssembler Implementation Workspace Directory Contents

Directory/Subdirectory	Description
components	Contains a directory for each IP component instance connected in the subsystem. Generated and populated with separate component directories upon first adding components; populated and updated throughout activities.
<i>i_component/auxiliary</i>	Scripts and text files used by coreAssembler. Generated during Add Subsystem Components activity.
<i>i_component/custom</i>	Contains RTL preprocessor scripts. Generated during Configure Components activity.
<i>i_component/doc</i>	Contains local copies of component-specific databooks. Generated during Add Subsystem Components activity.
<i>i_component/export</i>	Contains files used to integrate results from the completed source configuration and synthesis activities into your design (outside coreAssembler). Generated during Add Subsystem Components activity; populated during Configure Components activity.
<i>i_component/gtech</i>	Contains synthesis scripts and output netlists from gtech generation; also used for RTL simulation of encrypted source code. Generated during Create Component GTECH Simulation Model activity.
<i>i_component/kb</i>	Contains knowledge base information used by coreAssembler. These are binary files containing the state of the design. Generated during Add Subsystem Components activity; populated and updated throughout activities.
<i>i_component/leda</i>	Contains Leda configuration files for the component. Generated during Add Subsystem Components activity; populated during Run Leda Coding Checker (for <i>/i_component</i>) activity.
<i>i_component/report</i>	Contains all of the reports created by coreAssembler during build, configuration, test and synthesis phases. An index.html file in this directory links to many of these generated reports. Generated during Add Subsystem Components activity; populated and updated throughout activities.
<i>i_component/scratch</i>	Contains temp files used during the coreAssembler processes. Generated during Add Subsystem Components activity; populated and updated throughout activities.
<i>i_component/sim</i>	Contains test stimulus and output files. Generated during Add Subsystem Components activity; updated during Setup and Run Simulations (for <i>/i_component</i>) activity.

Table 2-4 coreAssembler Implementation Workspace Directory Contents (Continued)

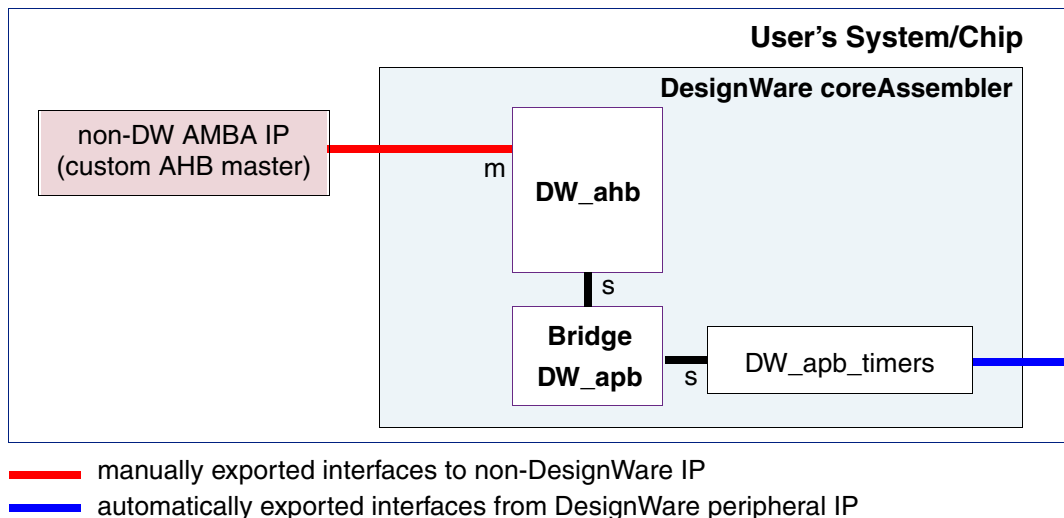
Directory/Subdirectory	Description
<code>i_component/spyglass</code>	Contains SpyGlass Lint and CDC configuration files for the component. Generated upon first SpyGlass run; updated during Run Spyglass RTL Checker activity.
<code>i_component/src</code>	Includes the top-level RTL file, <i>design_name.v</i> . If you have a source license, this will contain plain-text RTL; if you only have a designware license, this will contain encrypted RTL. Generated during Add Subsystem Components activity; populated during Specify Configuration activity.
<code>i_component/syn</code>	Contains synthesis files for the component. Generated during Add Subsystem Components activity; updated during Synthesis activity.
<code>i_component/tcl</code>	Contains synthesis intent scripts. Generated during Add Subsystem Components activity.
<code>export</code>	Contains subsystem files used to integrate the results from the completed source configuration and synthesis activities into your design (outside coreAssembler). Generated upon first creating workspace; populated starting with Memory Map Specification activity.
<code>kb</code>	Contains subsystem knowledge base information used by coreAssembler. These are binary files containing the state of the design. Generated upon first creating workspace; populated and updated throughout activities.
<code>report</code>	Contains subsystem reports created by coreAssembler during build, configuration, test and synthesis phases. An index.html file in this directory links to many of these generated reports. Generated upon first creating workspace; populated and updated throughout activities.
<code>scratch</code>	Contains subsystem temp files used during the coreAssembler processes. Generated upon first creating workspace; populated and updated throughout activities.
<code>src</code>	Includes the RTL related to the subsystem. If you have a source license, this will contain plain-text RTL; if you only have a designware license, this will contain encrypted RTL. Generated upon first creating workspace; populated starting with Generate Subsystem RTL activity.
<code>syn</code>	Contains synthesis files for the subsystem. Generated upon first creating workspace; updated during Synthesize activity and Formal Verification activity.

For details on some key files created during coreAssembler activities, refer to [“Database Files”](#) on page 28.

For information on using coreAssembler, refer to the [coreAssembler User Guide](#). For information on getting started with using DesignWare SIP components for AMBA 2 and AMBA 3 AXI components within coreTools, refer to [Using DesignWare Library IP in coreAssembler](#).

Figure 2-5 illustrates the DW_apb_timers in a simple subsystem.

Figure 2-5 DW_apb_timers in Simple Subsystem



The subsystem in Figure 2-5 contains the following components that you may want to use as you learn to use coreAssembler:

- DW_apb_timers
- DW_ahb
- DW_apb
- AHB Master

The AHB Master is meant to be exported out of the design and then replaced by a real AHB Master – such as a CPU – later in the design process; at least one exported AHB master is required in a subsystem if you intend to do a basic simulation that tests connections.

2.3.2 Configuring the DW_apb_timers within a Subsystem

The “Parameters” chapter on [page 43](#) describes the DW_apb_timers hardware configuration parameters that you configure using the coreAssembler GUI. Corresponding databooks for the other components in a subsystem contain “Parameters” chapters that describe their respective configuration parameters.

The “Creating the RTL View of a Subsystem” chapter in the [coreAssembler User Guide](#) discusses how to configure subsystem components and automatically connect them using the coreAssembler GUI.

2.3.3 Creating Gate-Level Netlists within coreAssembler

The “Creating the Gate-Level Netlist for a Subsystem” chapter in the [coreAssembler User Guide](#) discusses how to create a translation of the RTL view into a technology-specific netlist for a subsystem.

2.3.4 Verifying the DW_apb_timers within coreAssembler

The “[Verification](#)” chapter on [page 71](#) provides an overview of the testbench available for DW_apb_timers verification using the coreAssembler GUI.

The “Verifying Subsystems and Components” chapter in the [coreAssembler User Guide](#) discusses how to simulate a subsystem.

2.3.5 Running Leda on Generated Code with coreAssembler

When you select **Verify Component > Run Leda Coding Checker for /i_component)** from the Activity List, the corresponding Activity View appears. In this Activity View you select rules configuration file and define Leda command line switches.

2.3.6 Running Spyglass on Generated Code with coreAssembler

When you select **Verify Component > Run Spyglass RTL Checker for /i_component** from the Activity List, the corresponding Activity View appears. In this Activity View, you can select to run Spyglass Lint and Spyglass CDC.

2.4 Database Files

The following subsections describe some key files created in coreConsultant and coreAssembler activities.

2.4.1 Design/HDL Files

The following sections describe the design and HDL files that are produced by coreConsultant and coreAssembler when configuring and verifying a DesignWare Synthesizable Component. The following files are created in different directories by coreConsultant and coreAssembler:

- coreConsultant – *workspace/* directory
- coreAssembler – *workspace/components/i_component/* directory

2.4.1.1 RTL-Level Files

The following table describes the RTL files that are generated by the Create RTL activity. They are encrypted except where otherwise noted. Any Synopsys synthesis tool or simulator can read encrypted RTL files.

Table 2-5 RTL-Level Files

Files	Encrypted?	Purpose
<i>./src/component_cc_constants.v</i>	No	Includes definitions and values of all configuration parameters that you have specified for the component.
<i>./src/component.v</i>	No	Top-level HDL file. Include the DesignWare libraries by using the following options in your simulator invocation: +libext+.v+.V -y \${SYNOPSYS}/packages/gtech/src_ver -y \${SYNOPSYS}/dw/sim_ver

Table 2-5 RTL-Level Files (Continued)

Files	Encrypted?	Purpose
<code>./src/component_submodule.v</code>	Yes	Sub-modules of component
<code>./src/component_constants.v</code>	No	Includes the constants used internally in the design.
<code>./src/component_undef.v</code>		Includes an undef for each of the definitions found in the <code>component_cc_constants.v</code> file; compiled in after the last file listed in <code>./src/components.lst</code> when compiling multiple instances of the same IP.
<code>./src/component.lst</code>	No	Lists the order in which the RTL files should be read into tools, such as simulators or <code>dc_shell</code> . For example, use the following option to read the design into VCS: <code>vcs +v2k -f component.lst</code>

2.4.1.2 Simulation Model Files

The following table includes files generated for the component during the Generate GTECH Simulation activity. These files are needed when you are using a non-Synopsys simulator (when you can not use the encrypted RTL).

Table 2-6 Simulation Model Files

Files	Encrypted?	Purpose
<code>./gtech/final/db/component.v</code>	No	Simulation model of the component for use with non-Synopsys simulators. A technology-independent, gate-level netlist; VHDL and Verilog versions are generated. Include the DesignWare libraries by using the following options in your simulator invocation: <code>+libext+.v+.V</code> <code>-y \${SYNOPSYS}/packages/gtech/src_ver</code> <code>-y \${SYNOPSYS}/dw/sim_ver</code>

2.4.2 Synthesis Files

The following table includes files generated after the Create Gate-Level Netlist activity is performed on a component.

Table 2-7 Synthesis Files

Files	Encrypted?	Purpose
<code>./syn/auxScripts</code>	No	Auxiliary files for synthesis.
<code>./syn/final/db/component.db</code>	Binary format	Synopsys <code>.db</code> files (gate level) that can be read into <code>dc_shell</code> for further synthesis, if desired.
<code>./syn/final/db/component.v</code>	No	Gate-level netlist that is mapped to technology libraries that you specify.
<code>./syn/constrain/script/*.*</code>	No	Constraint files for the components.
<code>./syn/final/report/*.*</code>	No	Synthesis result files.

2.4.3 Verification Reference Files

Files described in the following table include information pertaining to the component's operation so that you can verify installation and configuration of the component has been successful. These files are not for re-use during system-level verification.

Table 2-8 Verification Reference Files

Files	Encrypted?	Purpose
./sim/runtest	No	Perl script that runs the Setup and Run Simulations activity from the command line.
./sim/runtest.log	No	The overall result of simulation, including pass/fail results.
./sim/test_ <i>testname</i> /test.result	No	Pass/fail of individual test.
./sim/test_ <i>testname</i> /test.log	No	Log file for individual test.

3

Functional Description

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

3.1 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 TimerNLoadCount 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.

3.2 DW_apb_timers Usage Flow

The procedure illustrated in [Figure 3-1](#) is a basic flow to follow when programming the DW_apb_timers. More advanced functions are discussed later in this chapter.

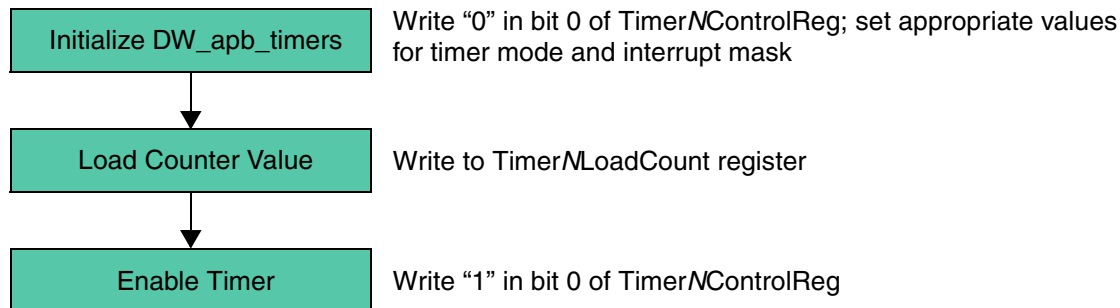
1. Initialize the timer through the [TimerNControlReg](#) register (where N is in the range 1 to 8):
 - a. Disable the timer by writing a “0” to the timer enable bit (bit 0); accordingly, the timer_en output signal is de-asserted.

**Note**

Before writing to a TimerNLoadCount register, you *must* disable the timer by writing a “0” to the timer enable bit of TimerNControlReg in order to avoid potential synchronization problems.

- b. Program the timer mode—user-defined or free-running—by writing a “0” or “1,” 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 3-1 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, refer to “[Controlling Clock Boundaries and Metastability](#)” on page 36.

3.3 DW_apb_timers Configuration

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

3.3.1 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](#) – TimerN load count register
- [TimerNLoadCount2](#) (optional) – TimerN load count register for programming width of HIGH period of timer_N_toggle output
- [TimerNCurrentValue](#) – TimerN current value register
- [TimerNControlReg](#) – TimerN control register
- [TimerNEOI](#) – TimerN end-of-interrupt register
- [TimerNIntStatus](#) – TimerN 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.

3.3.2 Enabling and Disabling a Timer

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

3.3.2.1 Enabling a Timer

If you want to enable a timer, you write a “1” to bit 0 of its [TimerNControlReg](#) register.

3.3.2.2 Disabling a Timer

To disable a timer, write a “0” to bit 0 of its [TimerNControlReg](#) 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. A “0” is always read back when the timer is not enabled; otherwise, the current value of the timer ([TimerNCurrentValue](#) register) is read back.

3.3.3 Configuring the Width of a Timer

You 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 will have to be multiple APB write accesses to load the counter.

3.3.4 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](#).



Note

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 [TimerMLoadCount](#) register and the [TimerMLoadCount2](#) register. For more details, refer to “[Pulse Width Modulation of Toggle Outputs](#)” on page 40.

- Free-running mode – Timer loads the maximum value, which is dependent on the timer width; that is, the `TimerNLoadCount` register is comprised of $2^{\text{TIMER_WIDTH_N}-1}$ 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`.

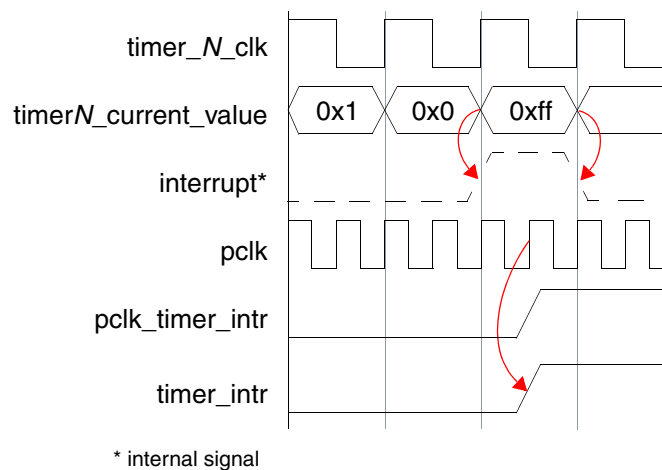
3.3.5 Working with Interrupts

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 3-2](#).

Figure 3-2 Timer Interrupt Set – No Metastability Registers and `TIM_NEWMODE = 0`

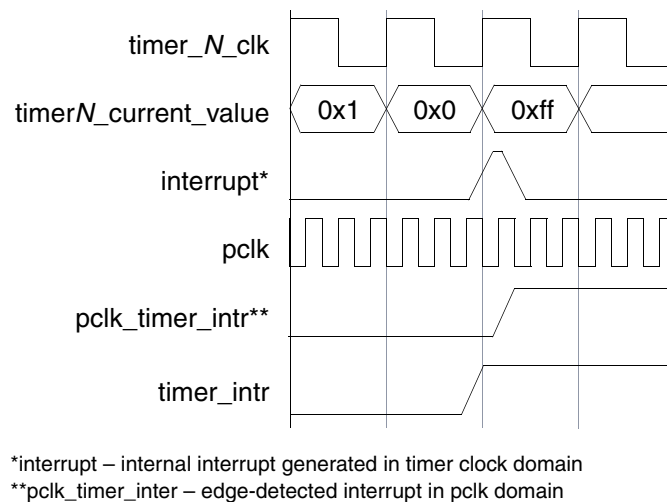


The setting of the internal interrupt signal occurs synchronously to the timer clock domain. This internal interrupt signal is transferred to the `pclk` domain in order to set the timer interrupt. The internal interrupt signal and the timer interrupt are not generated if the timer is disabled; if the 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 3-3, 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 3-3 Timer Interrupt Set - No Metastability Registers and TIM_NEWMODE = 1



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.

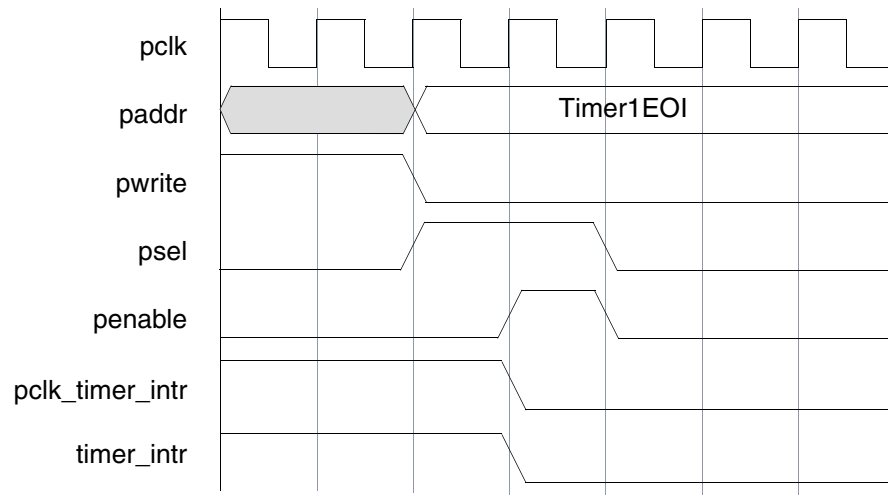
3.3.5.1 Clearing Interrupts

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 3-4 shows the timer interrupt timing when cleared by the TimersEOI register.

Figure 3-4 Clearing an Interrupt From DW_apb_timers



3.3.5.2 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.

3.3.5.3 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 [TimerNControlReg](#).

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

3.3.5.4 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.

3.3.6 Controlling Clock Boundaries and Metastability

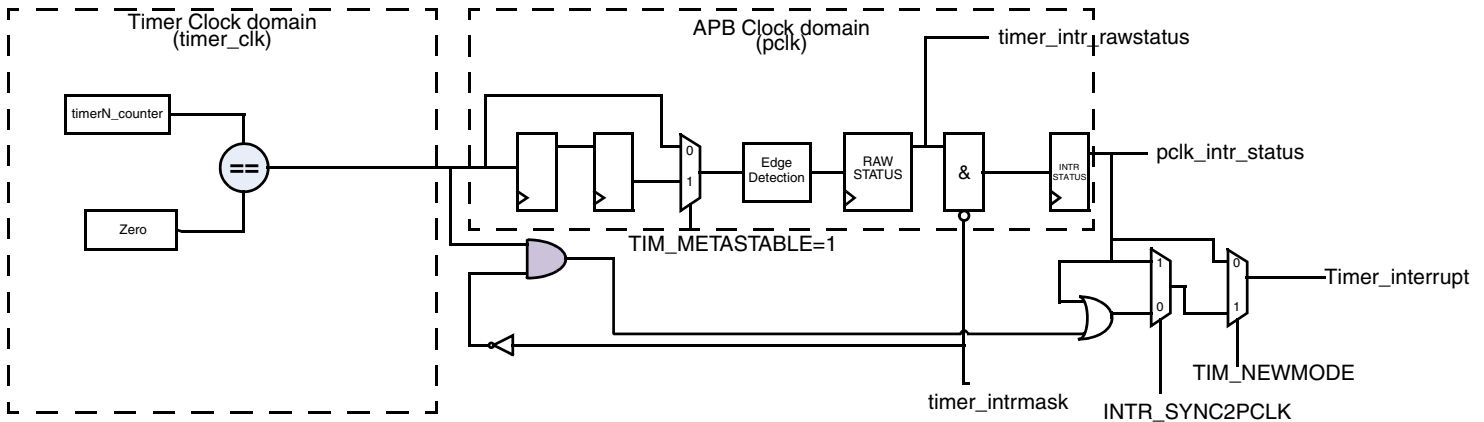
All registers in the APB interface are synchronous to `pclk`. Each of the timers has a separate clock input signal, `timer_N_clk`, that can be asynchronous or synchronous to `pclk`. It is possible to connect `timer_N_clk` to a clock other than `pclk`, but if you do that, you must take into account the possibility of synchronization and metastability issues.

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_resetrn 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 3-5 shows an internal interrupt signal affecting the clock boundaries between the two clock domains.

Figure 3-5 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:

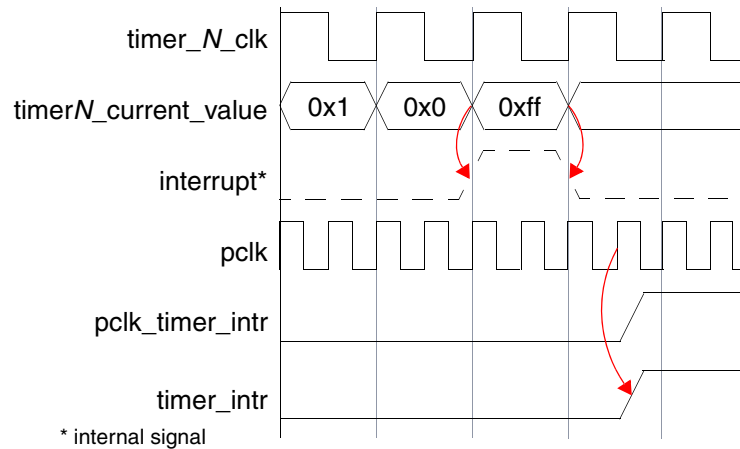
- When TIM_NEWMODE is set to 0 (False), the timer interrupt is asserted when the internal interrupt signal is edge-detected in the pclk domain.
- When TIM_NEWMODE is set to 1 (True) and INTR_SYNC2PCLK is set to 1 (True), the timer interrupt is asserted when the internal interrupt signal is edge-detected 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, illustrated in [Figure 3-6](#).

Figure 3-6 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 TimerNLoadCount 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 TimerNControlReg 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 TimerNControlReg nor the TimerNLoadCount 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 TimerNControlReg to disable the timer, program its timer mode, and then set the interrupt mask.
2. Next, load the timer counter value into the TimerNLoadCount register.
3. Finally, enable the timer through TimerNControlReg.

For more details on this procedure, refer to [“DW_apb_timers Usage Flow”](#) on page 31.

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 3-2](#) on page 34 (no metastability registers) to that in [Figure 3-6](#) on page 38 (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 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 3-7 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 3-7 Timer Interrupt Set – Pulse Extend One Cycle, No Metastability

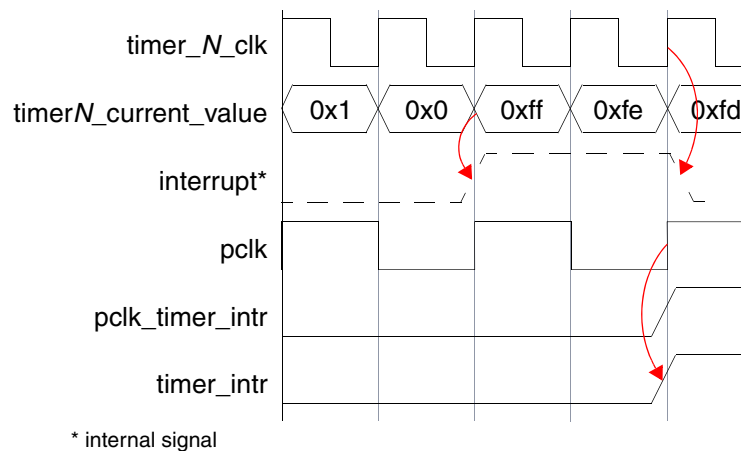
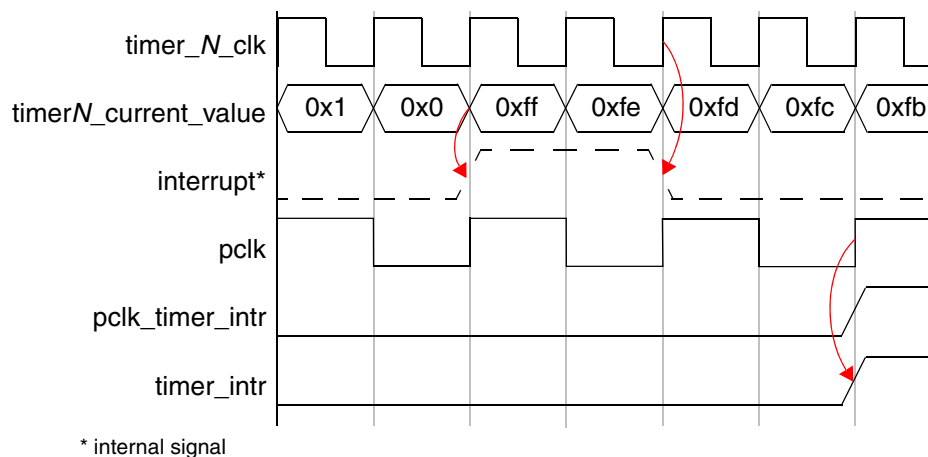


Figure 3-8 illustrates an example where metastability registers are required because pclk is independent of timer_N_clk, and $1 < \text{frequency of timer_N_clk} < 2$ times that of pclk. To accommodate this, the TIM_PULSE_EXTD_N parameter is set to 1 in order to extend the internal interrupt signal by one timer_N_clk clock period.

Figure 3-8 Timer Interrupt Set – Pulse Extend One Cycle, With Metastability

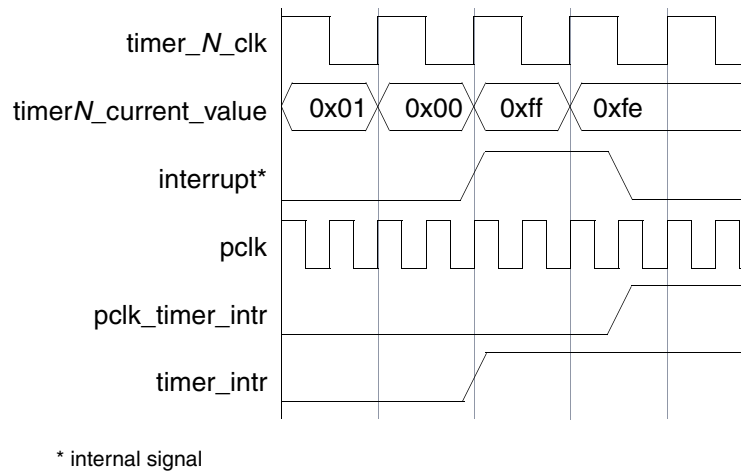


Note

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 3-9 illustrates an example where metastability registers are included when **TIM_NEWMODE = 1**.

Figure 3-9 Timer Interrupt Set – Metastability Registers Included and TIM_NEWMODE=1



3.3.7 Generating Toggled Outputs

You can configure a timer through the **TIMER_HAS_TOGGLE_N** parameter (Include toggle output for timer # on I/F, see [page 45](#)) 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.

3.3.7.1 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 **TimerNLoadCount2** and **TimerNLoadCount** registers.

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

- Width of **timer_N_toggle** HIGH period = (**TimerNLoadCount2** + 1) * **timer_N_clk** clock period
- Width of **timer_N_toggle** LOW period = (**TimerNLoadCount** + 1) * **timer_N_clk** clock period

If **TIM_NEWMODE** is set to 0 or the **TimerNControlReg[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.

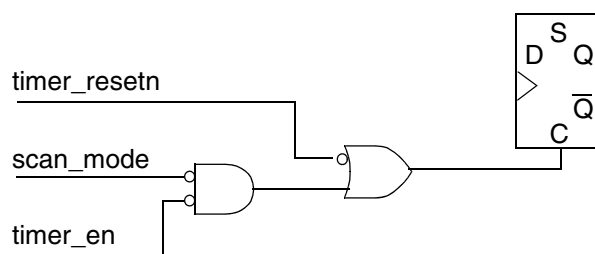
3.3.8 Timer Pause Mode

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.

3.4 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 3-10. 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 3-10 Design For Test – Use of Scan Mode Signal



4

Parameters

This chapter describes the configuration parameters used by the DW_apb_timers. The settings of the configuration parameters determine the I/O signal list of the DW_apb_timers peripheral. You use coreConsultant or coreAssembler to configure the following parameters and generate the configured code.



Attention

When using coreConsultant or coreAssembler, you can right-click on a parameter label to access a “What’s This” popup dialog that will tell you the details for that particular parameter. The information in each What’s This dialog essentially matches the information in the parameter descriptions below.

4.1 Parameter Descriptions

In the following tables, the values 0 and 1 occasionally appear in parentheses in the descriptions for the parameters. These are the logical values for parameter settings that appear in the coreConsultant GUI as check boxes, drop-down lists, a multiple selection, and so on.

Table 4-1 lists the DW_apb_timers top-level parameter descriptions.

Table 4-1 Top-Level Parameters

Field Label	Parameter Definition
APB Data Bus Width (bits)	Parameter Name: APB_DATA_WIDTH Legal Values: 8, 16, or 32 Default Value: 32 Dependencies: None. Description: Width of the APB data bus to which this component is attached.
Number of Timers to instantiate	Parameter Name: NUM_TIMERS Legal Values: 1 to 8 Default Value: 2 Dependencies: None. Description: Number of timers to instantiate in DW_apb_timers. Up to eight timers can be instantiated.


Table 4-1 Top-Level Parameters (Continued)

Field Label	Parameter Definition
Enable Timer New Mode?	<p>Parameter Name: TIM_NEWMODE</p> <p>Legal Values: True (1) or False (0)</p> <p>Default Value: False (0)</p> <p>Dependencies: Enabled only if APB_DATA_WIDTH = 32</p> <p>Description: When set to True (1), this parameter enables the following features in all the timers:</p> <ul style="list-style-type: none"> ■ If Timer/ControlReg[4] is set to 1, the width of LOW and HIGH periods of timer toggle outputs can be separately programmed through Timer/LoadCount and Timer/LoadCount2 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.
Timer Interrupt Clock Domain	<p>Parameter: INTR_SYNC2PCLK</p> <p>Values: 0 (Timer clock Interrupt), 1 (APB clock interrupt)</p> <p>Default Value: 0 (Timer clock Interrupt)</p> <p>Dependencies: TIM_NEWMODE = 1</p> <p>Description: 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.</p>
Interrupt Polarity	<p>Parameter Name: TIM_INTRPT_PLRITY</p> <p>Legal Values: Active-high (1) or active-low (0)</p> <p>Default Value: Active-high (1)</p> <p>Dependencies: Cannot be set to active-low (0) if TIM_NEWMODE is set to 1.</p> <p>Description: Polarity of interrupt signals generated by DW_apb_timers.</p>
Single Combined Interrupt?	<p>Parameter Name: TIM_INTR_IO</p> <p>Legal Values: True (1) or False (0)</p> <p>Default Value: False (0)</p> <p>Dependencies: Cannot be set to True (1) if TIM_NEWMODE is set to 1.</p> <p>Description: 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.</p>

Table 4-2 Timer *i* Configuration Parameters

Field Label	Parameter Definition (<i>N</i> has the range 1 to <i>NUM_TIMERS</i>)
Width of Timer # <i>i</i>	<p>Parameter Name: TIMER_WIDTH_<i>N</i>, where <i>N</i> is in the range 1 to NUM_TIMERS</p> <p>Legal Values: 8 to 32 bits</p> <p>Default Value: 32 bits</p> <p>Dependencies: This option is disabled for the following: for <i>N</i> = 2, when NUM_TIMERS = 1; for <i>N</i> = 3, when NUM_TIMERS <= 2; for <i>N</i> = 4, when NUM_TIMERS <= 3; for <i>N</i> = 5, when NUM_TIMERS <= 4; for <i>N</i> = 6, when NUM_TIMERS <= 5; for <i>N</i> = 7, when NUM_TIMERS <= 6; for <i>N</i> = 8, when NUM_TIMERS <= 7</p> <p>Description: Width of each timer.</p>
Include toggle output for timer # <i>i</i> on I/F?	<p>Parameter Name: TIMER_HAS_TOGGLE_<i>N</i>, where <i>N</i> is in the range 1 to NUM_TIMERS</p> <p>Legal Values: True (1) or False (0)</p> <p>Default Value: False (0)</p> <p>Dependencies: This option is disabled for the following: for <i>N</i> = 2, when NUM_TIMERS = 1; for <i>N</i> = 3, when NUM_TIMERS <= 2; for <i>N</i> = 4, when NUM_TIMERS <= 3; for <i>N</i> = 5, when NUM_TIMERS <= 4; for <i>N</i> = 6, when NUM_TIMERS <= 5; for <i>N</i> = 7, when NUM_TIMERS <= 6; for <i>N</i> = 8, when NUM_TIMERS <= 7</p> <p>Description: When set to True (1), the interface includes an output (timer_<i>N</i>_toggle) that toggles each time the timer counter reloads. The output is disabled to 0 each time the timer is disabled.</p>
Metastability support for interrupt from Timer # <i>i</i>	<p>Parameter Name: TIM_METASTABLE_<i>N</i>, where <i>N</i> is in the range 1 to NUM_TIMERS</p> <p>Legal Values: Absent (0) or Present (1)</p> <p>Default Value: Absent (0)</p> <p>Dependencies: This option is disabled for the following: for <i>N</i> = 2, when NUM_TIMERS = 1; for <i>N</i> = 3, when NUM_TIMERS <= 2; for <i>N</i> = 4, when NUM_TIMERS <= 3; for <i>N</i> = 5, when NUM_TIMERS <= 4; for <i>N</i> = 6, when NUM_TIMERS <= 5; for <i>N</i> = 7, when NUM_TIMERS <= 6; for <i>N</i> = 8, when NUM_TIMERS <= 7</p> <p>Description: This option instantiates metastability registers to synchronize timer interrupt signals to the pclk domain. Set this to Present (1) if timer_<i>N</i>_clk is independent of pclk. If this parameter is set to Absent (0), then timer_<i>N</i>_clk is considered to be connected to or synchronous with pclk</p>

Table 4-2 Timer *i* Configuration Parameters (Continued)

Field Label	Parameter Definition (<i>N</i> has the range 1 to <i>NUM_TIMERS</i>)												
Number of clock cycles by which to extend interrupt	<p>Parameter Name: TIM_PULSE_EXTD_<i>N</i>, where <i>N</i> is in the range 1 to NUM_TIMERS</p> <p>Legal Values: 0 to 3</p> <p>Default Value: 0</p> <p>Dependencies: This option is disabled for the following:</p> <ul style="list-style-type: none"> ■ If TIM_NEWMODE is set to 1 ■ for <i>N</i> = 2, when NUM_TIMERS = 1; for <i>N</i> = 3, when NUM_TIMERS ≤ 2; for <i>N</i> = 4, when NUM_TIMERS ≤ 3; for <i>N</i> = 5, when NUM_TIMERS ≤ 4; for <i>N</i> = 6, when NUM_TIMERS ≤ 5; for <i>N</i> = 7, when NUM_TIMERS ≤ 6; for <i>N</i> = 8, when NUM_TIMERS ≤ 7 <p>Description: 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 “Controlling Clock Boundaries and Metastability” on page 36.</p> <p>Set this parameter to the following values, depending on the timer_<i>N</i>_clk/pclk frequency ratio <i>R</i>:</p> <table> <tr> <th>timer_<i>N</i>_clk/pclk frequency <i>R</i></th><th>PULSE_EXTEND_<i>N</i></th></tr> <tr> <td>$R \leq 1$</td><td>0</td></tr> <tr> <td>$1 < R \leq 2$</td><td>1</td></tr> <tr> <td>$2 < R \leq 3$</td><td>2</td></tr> <tr> <td>$3 < R \leq 4$</td><td>3</td></tr> <tr> <td>$4 < R$</td><td>Not valid</td></tr> </table>	timer_ <i>N</i> _clk/pclk frequency <i>R</i>	PULSE_EXTEND_ <i>N</i>	$R \leq 1$	0	$1 < R \leq 2$	1	$2 < R \leq 3$	2	$3 < R \leq 4$	3	$4 < R$	Not valid
timer_ <i>N</i> _clk/pclk frequency <i>R</i>	PULSE_EXTEND_ <i>N</i>												
$R \leq 1$	0												
$1 < R \leq 2$	1												
$2 < R \leq 3$	2												
$3 < R \leq 4$	3												
$4 < R$	Not valid												
Include Coherency Registers for this timer?	<p>Parameter Name: TIM_COHERENCY_<i>N</i>, where <i>N</i> is in the range 1 to NUM_TIMERS</p> <p>Legal Values: True (1) or False (0)</p> <p>Default Value: False (0)</p> <p>Dependencies: You can set coherency for the timer if its width, TIMER_WIDTH_<i>N</i>, is greater than APB_DATA_WIDTH. If the TIMER_WIDTH is less than or equal to APB_DATA_WIDTH, coherency is not implemented for the specific timer.</p> <p>This option is disabled for the following:</p> <ul style="list-style-type: none"> for <i>N</i> = 2, when NUM_TIMERS = 1; for <i>N</i> = 3, when NUM_TIMERS ≤ 2; for <i>N</i> = 4, when NUM_TIMERS ≤ 3; for <i>N</i> = 5, when NUM_TIMERS ≤ 4; for <i>N</i> = 6, when NUM_TIMERS ≤ 5; for <i>N</i> = 7, when NUM_TIMERS ≤ 6; for <i>N</i> = 8, when NUM_TIMERS ≤ 7 <p>Description: 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 take place while the read operation is in progress.</p> <p> Note Including coherency can dramatically increase the register count of the design.</p>												

5

Signal Descriptions

This chapter details all possible I/O signals in the core. 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 core. It is for reference purposes only.

When you configure the core 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 core in coreConsultant, all TCL functions and parameters are evaluated completely; and the resulting values are displayed where appropriate in the coreConsultant GUI reports.

The I/O signals are grouped as follows:

- APB Interface on [page 48](#)
- Timer Signals on [page 50](#)

5.1 APB Interface Signals

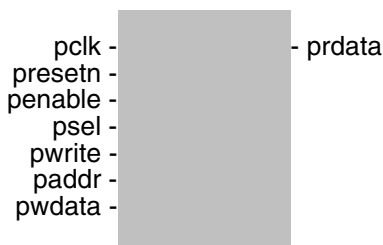


Table 5-1 APB Interface Signals

Port Name	I/O	Description
pclk	I	<p>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.</p> <p>Exists: Always</p> <p>Active State: N/A</p> <p>Synchronous to: N/A</p> <p>Registered: N/A</p>
presetn	I	<p>APB reset. The bus reset signal is used to reset the system and the bus on the DesignWare interface.</p> <p>Exists: Always</p> <p>Active State: Low</p> <p>Synchronous to: 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.</p> <p>Registered: N/A</p>
penable	I	<p>APB enable control that indicates the second cycle of the APB frame.</p> <p>Exists: Always</p> <p>Active State: High</p> <p>Synchronous to: pclk</p> <p>Registered: No</p>
psel	I	<p>APB peripheral select</p> <p>Exists: Always</p> <p>Active State: High</p> <p>Synchronous to: pclk</p> <p>Registered: No</p>

Table 5-1 APB Interface Signals (Continued)

Port Name	I/O	Description
pwrite	I	APB write control. Exists: Always Active State: High Synchronous to: pclk Registered: No
paddr[TIM_ADDR_SLICE_LHS:0]	I	APB address bus. Exists: Always Active State: N/A Synchronous to: pclk Registered: No
pwwdata[(APB_DATA_WIDTH-1):0]	I	APB write data bus. Exists: Always Active State: N/A Synchronous to: pclk Registered: No
prdata[(APB_DATA_WIDTH-1):0]	O	APB readback data. Exists: Always Active State: N/A Synchronous to: pclk Registered: Yes

5.2 Timer Signals

scan_mode	-	timer_en
timer_N_clk (for N = 1; N <= NUM_TIMERS)	-	timer_intr
timer_N_resetrn	-	timer_intr_n
timer_N_pause	-	timer_intr_flag
		timer_intr_flag_n
		timer_N_toggle

Table 5-2 Timer Signals

Port Name	I/O	Description
scan_mode	I	<p>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.</p> <p>Exists: Always</p> <p>Active State: High</p> <p>Synchronous to: N/A</p> <p>Registered: No</p>
timer_N_clk (for N = 1; N <= NUM_TIMERS)	I	<p>Each timer is supplied with its own clock from this bus. The number of these signals is set by NUM_TIMERS parameter.</p> <p>Exists: NUM_TIMERS >= N</p> <p>Active State: High</p> <p>Synchronous to: 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.</p> <p>Registered: N/A</p>
timer_N_resetrn (for N = 1; N <= NUM_TIMERS)	I	<p>Asynchronous reset for each timer. The number of these signals are set by NUM_TIMERS parameter.</p> <p>Exists: NUM_TIMERS >= N</p> <p>Active State: Low</p> <p>Synchronous to: Asynchronous assertion, synchronous de-assertion. Must be synchronously de-asserted after the rising edge of pclk.</p> <p>Registered: N/A</p>
timer_N_pause (for N = 1; N <= NUM_TIMERS)	I	<p>Optional. Input signal; when asserted, causes the timer to pause/freeze.</p> <p>Exists: (NUM_TIMERS >= N) && (TIM_NEWMODE==1)</p> <p>Active State: High</p> <p>Synchronous to: timer_N_clk</p> <p>Registered: Yes</p>

Table 5-2 Timer Signals (Continued)

Port Name	I/O	Description
timer_en[(NUM_TIMERS-1):0]	O	<p>When asserted, activates the necessary timer clocks and ensures the component is supplied with an active pclk while timers are running.</p> <p>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.</p> <p>Exists: Always Active State: High Synchronous to: pclk Registered: Yes</p>
timer_intr[(NUM_TIMERS-1):0]	O	<p>Optional. Timer interrupt active high signals.</p> <p>Exists: (TIM_INTRPT_PLRITY==1) && (TIM_INTR_IO==TIM_INDIVIDUAL) Active State: High Synchronous to: pclk Registered: No</p>
timer_intr_n[(NUM_TIMERS-1):0]	O	<p>Optional. Timer interrupt active low signals.</p> <p>Exists: (TIM_INTRPT_PLRITY==0) && (TIM_INTR_IO==TIM_INDIVIDUAL) Active State: Low Synchronous to: pclk Registered: No</p>
timer_intr_flag	O	<p>Optional. Active High Interrupt flag that is set if any timer interrupt is set.</p> <p>Exists: (TIM_INTRPT_PLRITY==1) && (TIM_INTR_IO==TIM_COMBINED) Active State: High Synchronous to: pclk Registered: No</p>
timer_intr_flag_n	O	<p>Optional. Active Low Interrupt flag that is set if any timer interrupt is set.</p> <p>Exists: (TIM_INTRPT_PLRITY==0) && (TIM_INTR_IO==TIM_COMBINED) Active State: Low Synchronous to: pclk Registered: No</p>

Table 5-2 Timer Signals (Continued)

Port Name	I/O	Description
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) Active State: High Synchronous to: timer_N_clk Registered: Yes

6

Registers

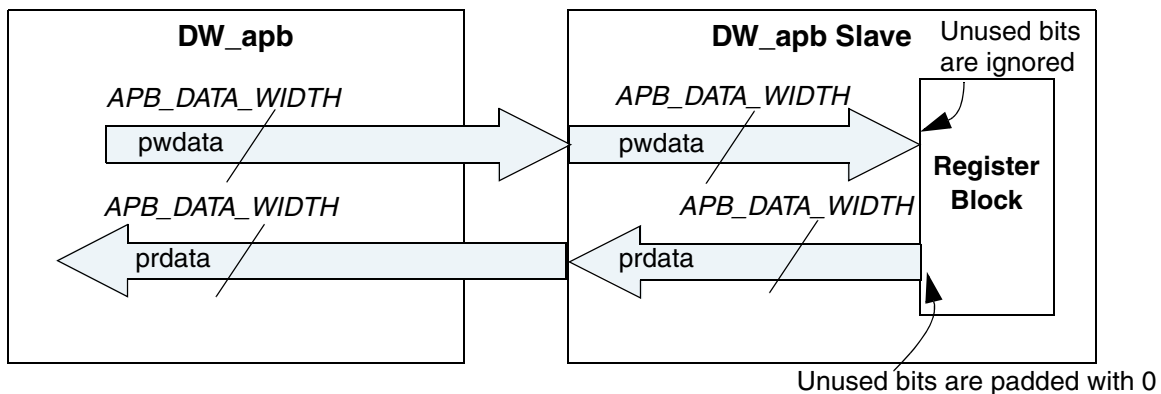
This section describes the programmable registers of the DW_apb_timers.

6.1 Bus Interface

The DW_apb_timers peripheral has a standard AMBA 2.0 APB interface for reading and writing the internal registers. This peripheral supports APB data bus widths of 8, 16, or 32 bits, which is set with the APB_DATA_WIDTH parameter.

Figure 6-1 shows the read/write buses between the DW_apb and the APB slave.

Figure 6-1 Relationship Between DW_apb and Slave Data Widths



“Integration Considerations” on page 75 provides information about reading to and writing from the APB interface.

6.2 Register Memory Map

The following tables contain information about the register memory map.

[Table 6-1](#) lists the address ranges of the registers for each timer; they are aligned to 32-bit boundaries.

Table 6-1 DW_apb_timers Address Range

Address Range (Base +)	Function
0x00 to 0x10	Timer 1 Registers
0x14 to 0x24	Timer 2 Registers
0x28 to 0x38	Timer 3 Registers
0x3c to 0x4c	Timer 4 Registers
0x50 to 0x60	Timer 5 Registers
0x64 to 0x74	Timer 6 Registers
0x78 to 0x88	Timer 7 Registers
0x8c to 0x9c	Timer 8 Registers
0xb0 to 0xcc	TimerMLoadCount2 Registers (present only when TIM_NEWMODE = 1)

[Table 6-2](#) lists registers associated with Timer 1; use this table as an example for timers 2-8.

Table 6-2 Memory Map of Timer 1 Registers

Name	Address Offset	Width	R/W	Description
Timer1LoadCount	0x00	See Description	R/W	Value to be loaded into Timer1 Width: <i>TIMER_WIDTH_1</i> Range: 0 to $[2^{TIMER_WIDTH_1} - 1]$ Default value: 0
Timer1LoadCount2	0xb0	See Description	R/W	Value to be loaded into Timer1 when toggle output changes from 0 to 1 Width: <i>TIMER_WIDTH_1</i> Range: 0 to $[2^{TIMER_WIDTH_1} - 1]$ Default value: <i>TIMER_WIDTH_1</i> 'b0 Dependency: Present only when TIM_NEWMODE = 0
Timer1CurrentValue	0x04	See Description	R	Current Value of Timer1 Width: <i>TIMER_WIDTH_1</i> Range: 0 to $[2^{TIMER_WIDTH_1} - 1]$ Default value: 0 (when TIM_NEWMODE = 0) $2^{((TIMER_WIDTH_1)-1)}$ (when TIM_NEWMODE = 1)

Table 6-2 Memory Map of Timer 1 Registers (Continued)

Name	Address Offset	Width	R/W	Description
Timer1ControlReg	0x08	4 bits	R/W	Control Register for Timer1 Default value: 3'b0
Timer1EOI	0x0C	1 bit	R	Clears the interrupt from Timer1 Default value: 1'b0
Timer1IntStatus	0x10	1 bit	R	Contains the interrupt status for Timer1 Default value: 1'b0

DW_apb_timers also contain the following three system registers, listed in [Table 6-3](#).

Table 6-3 DW_apb_timers System Registers

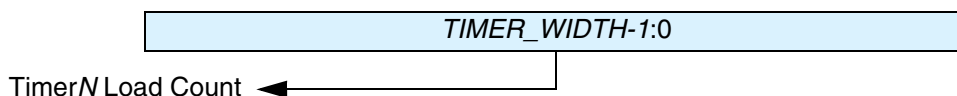
Name	Address Offset	Width	R/W	Description
TimersIntStatus	0xa0	See Description	R	Contains the interrupt status of all timers in the component. Width: NUM_TIMERS-1:0 Default value: NUM_TIMERS'b0
TimersEOI	0xa4	See Description	R	Returns all zeroes (0) and clears all active interrupts. Width: NUM_TIMERS-1:0 Default value: NUM_TIMERS'b0
TimersRawIntStatus	0xa8	See Description	R	Contains the unmasked interrupt status of all timers in the component. Width: NUM_TIMERS-1:0 Default value: NUM_TIMERS'b0
TIMERS_COMP_VERSION	0xac	32 bits	R	Current revision number of the DW_apb_timers component.

6.3 Register and Field Descriptions

The following sections contain the memory diagrams and field descriptions for the individual registers.

6.3.1 TimerNLoadCount

- **Name:** TimerN Load Count Register
- **Size:** *TIMER_WIDTH_N*, where *N* is 1...8
- **Address Offset:**
 - for *N* = 1, 0x00
 - for *N* = 2, 0x14
 - for *N* = 3, 0x28
 - for *N* = 4, 0x3C
 - for *N* = 5, 0x50
 - for *N* = 6, 0x64
 - for *N* = 7, 0x78
 - for *N* = 8, 0x8C
- **Read/write access:** read/write



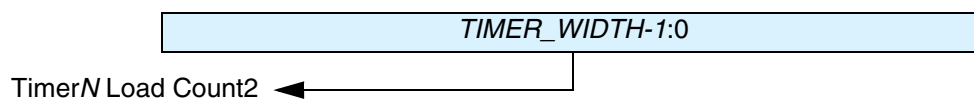
Bits	Name	R/W	Description
<i>TIMER_WIDTH_N-1:0</i> , where <i>N</i> is 1...8	TimerNLoad Count Register	R/W	Value to be loaded into TimerN. This is the value from which counting commences. Any value written to this register is loaded into the associated timer.

6.3.2 TimerNLoadCount2

- **Name:** TimerN Load2 Count Register
- **Size:** *TIMER_WIDTH_N*, where *N* is 1...8
- **Address Offset:**
 - for *N* = 1, 0xb0
 - for *N* = 2, 0xb4
 - for *N* = 3, 0xb8
 - for *N* = 4, 0xbC
 - for *N* = 5, 0xC0
 - for *N* = 6, 0xC4
 - for *N* = 7, 0xC8
 - for *N* = 8, 0xCC

- **Read/write access:** read/write

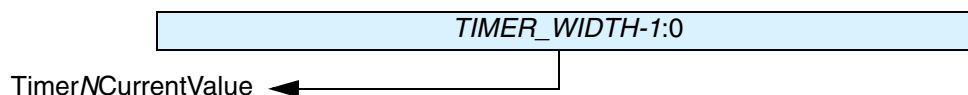
This register exists only if TIM_NEWMODE = 1



Bits	Name	R/W	Description
<i>TIMER_WIDTH_N-1:0</i> , where <i>N</i> is 1...8	TimerNLoad Count2 Register	R/W	Value to be loaded into TimerN when timer_ <i>N</i> _toggle output changes from 0 to 1. This value determines the width of the HIGH period of the timer_ <i>N</i> _toggle output.

6.3.3 TimerNCurrentValue

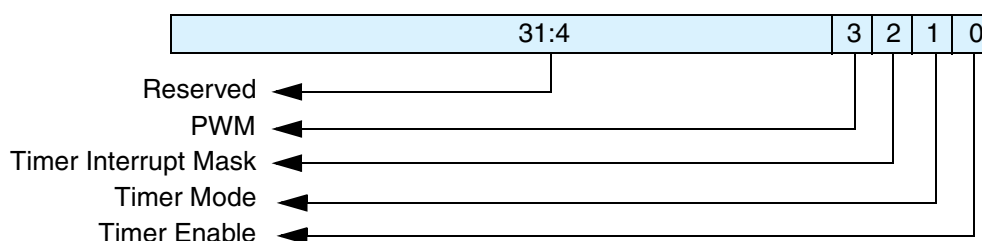
- **Name:** TimerN Current Value Register
- **Size:** *TIMER_WIDTH_N*, where *N* is 1...8
- **Address Offset:**
 - for *N* = 1, 0x04
 - for *N* = 2, 0x18
 - for *N* = 3, 0x2C
 - for *N* = 4, 0x40
 - for *N* = 5, 0x54
 - for *N* = 6, 0x68
 - for *N* = 7, 0x7C
 - for *N* = 8, 0x90
- **Read/write access:** read



Bits	Name	R/W	Description
<i>TIMER_WIDTH_N-1</i> : 0, where <i>N</i> is 1...8	Timer <i>N</i> Current Value	R	Current Value of Timer <i>N</i> . When TIM_NEWMODE=0, this register is supported only when timer_ <i>N</i> _clk is synchronous to pclk. Reading this register when using independent clocks results in an undefined value. When TIM_NEWMODE=1, no restrictions apply.

6.3.4 TimerNControlReg

- **Name:** TimerN Control Register
- **Size:** 4 bits
- **Address Offset:**
 - for $N = 1$, 0x08
 - for $N = 2$, 0x1C
 - for $N = 3$, 0x30
 - for $N = 4$, 0x44
 - for $N = 5$, 0x58
 - for $N = 6$, 0x6C
 - for $N = 7$, 0x80
 - for $N = 8$, 0x94
- **Read/write access:** read/write

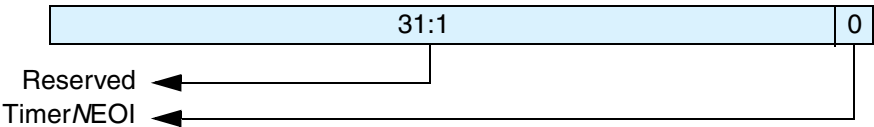


This register controls enabling, operating mode (free-running or defined-count), and interrupt mask of TimerN. You can program each TimerNControlReg to enable or disable a specific timer and to control its mode of operation.

Bits	Name	R/W	Description
31:4	Reserved, read as zero		
3	TIMER_PWM	R/W	Pulse Width Modulation of timer_N_toggle output. 0 – disabled 1 – enabled Dependency: This field present only if TIM_NEWMODE=1
2	Timer Interrupt Mask	R/W	Timer interrupt mask for TimerN 0 – not masked 1 – masked
1	Timer Mode	R/W	Timer mode for TimerN 0 – free-running mode 1 – user-defined count mode For more information about these modes, see “Timer Operation” on page 31.
0	Timer Enable	R/W	Timer enable bit for TimerN 0 – disable 1 – enable

6.3.5 TimerNEOI

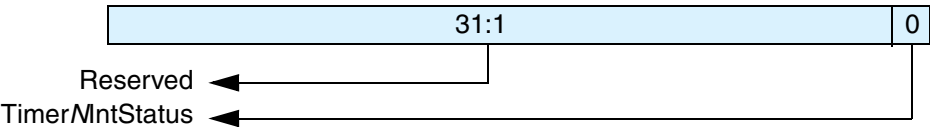
- **Name:** TimerN End-of-Interrupt Register
- **Size:** 1 bit
- **Address Offset:**
for N = 1, 0x0C
for N = 2, 0x20
for N = 3, 0x34
for N = 4, 0x48
for N = 5, 0x5C
for N = 6, 0x70
for N = 7, 0x84
for N = 8, 0x98
- **Read/write access:** read



Bits	Name	R/W	Description
31:1	Reserved, read as zero		
0	TimerN End-of-Interrupt Register	R	Reading from this register returns all zeroes (0) and clears the interrupt from TimerN.

6.3.6 TimerMntStatus

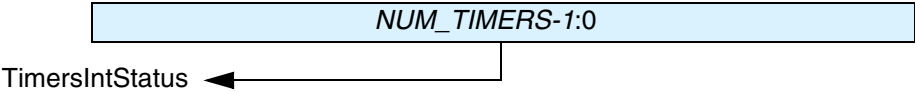
- **Name:** TimerN Interrupt Status Register
- **Size:** 1 bit
- **Address Offset:**
 - for N = 1, 0x10
 - for N = 2, 0x24
 - for N = 3, 0x38
 - for N = 4, 0x4C
 - for N = 5, 0x60
 - for N = 6, 0x74
 - for N = 7, 0x88
 - for N = 8, 0x9C
- **Read/write access:** read



Bits	Name	R/W	Description
31:1	Reserved		Reserved, read as zero
0	TimerN Interrupt Status Register	R	Contains the interrupt status for TimerN.

6.3.7 TimersIntStatus

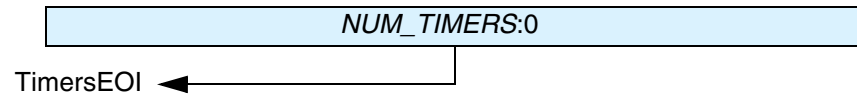
- **Name:** Timers Interrupt Status Register
- **Size:** NUM_TIMERS
- **Address Offset:** 0xa0
- **Read/write access:** read



Bits	Name	R/W	Description
NUM_TIMERS-1:0	Timers Interrupt Status Register	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: 0 – either timer_intr or timer_intr_n is not active after masking 1 – either timer_intr or timer_intr_n is active after masking

6.3.8 TimersEOI

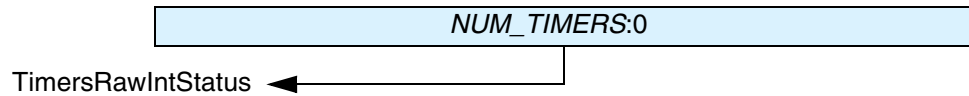
- **Name:** Timers End-of-Interrupt Register
- **Size:** *NUM_TIMERS*
- **Address Offset:** 0xa4
- **Read/write access:** read



Bits	Name	R/W	Description
<i>NUM_TIMERS</i> -1:0	Timers End-of-Interrupt Register	R	Reading this register returns all zeroes (0) and clears all active interrupts.

6.3.9 TimersRawIntStatus

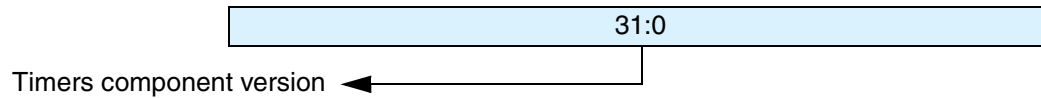
- **Name:** Timers Raw Interrupt Status Register
- **Size:** *NUM_TIMERS*
- **Address Offset:** 0xa8
- **Read/write access:** read



Bits	Name	R/W	Description
<i>NUM_TIMERS</i> -1:0	Timers Raw Interrupt Status Register	R	<p>The register contains the unmasked interrupt status of all timers in the component.</p> <p>0 – either timer_intr or timer_intr_n is not active prior to masking</p> <p>1 – either timer_intr or timer_intr_n is active prior to masking</p>

6.3.10 TIMERS_COMP_VERSION

- **Name:** Timers Component Version
- **Size:** 32 bits
- **Address Offset:** 0xac
- **Read/write access:** read



Bits	Name	R/W	Description
31:0	Timers Component Version	R	Current revision number of the DW_apb_timers component. Reset Value: For the value, see the releases table in the AMBA 2 release notes

7

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 7-1 Internal Parameters

Parameter Name	Equals To
TIM_ADDR_SLICE_LHS	7
TIM_COMBINED	1
TIM_INDIVIDUAL	0
blank	intentionally left blank

8

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 31.

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 TimerNLoadCount register value is loaded into a corresponding TimerN after the timer is enabled – either after a disable or a reset. DW_apb_timers contains both timer-specific and system registers. [Table 6-1](#) on page 54 shows the address range of the registers of each timer, which are aligned to 32-bit boundaries.

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.

**Note**

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.

**Note**

Reading the TimerNCurrentValue 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.

9

Verification

This chapter provides an overview of the testbench available for DW_apb_timers verification. Once you have configured the DW_apb_timers in either coreAssembler or coreConsultant and have set up the verification environment, you can run simulations automatically.

**Note**

The DW_apb_timers verification testbench is built with DesignWare Verification IP (VIP). Please 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, refer to the [DesignWare Synthesizable Components for AMBA 2/AMBA 3 AXI Installation Guide](#).

9.1 Overview of Vera Tests

The DW_apb_timers verification testbench performs tests that have been written to verify three types of functionalities:

- test_readwrite_regs – Tests read/write functionalities of each timer register.
- test_reset – Tests functions related to resetting all timers.
- test_timer – Tests general functions of each timer.

The tests are performed on the following:

- APB Slave Interface – DW_apb_timers consists of an APB slave interface and a separate timer block for each timer instantiated. These tests verify that the APB Slave interface implements the memory map for DW_apb_timers and also contain metastability flip-flops to synchronize interrupt flags coming from the timer clock domains to the bus system clock domain. The tests are run for an 8-bit, 16-bit, and 32-bit APB system.
- Timer blocks – Each timer instantiated in the DW_apb_timers has a block clocked by its own timer_N_clk. These tests verify that the block flags interrupts to the APB slave interface and carries out pulse extension of signals going to the slave interface block to handle scenarios where the timer_N_clk runs at a higher frequency than pclk.

The tests perform the following tasks:

- ❑ Disables a timer, programs the load value, and re-enables it.
- ❑ Verifies that in free-running mode the timer counts from the load value down to zero before wrapping to its maximum value and proceeding with its count.
- ❑ Verifies that in user-defined mode, the timer wraps back to the load value after passing 0.
- ❑ Verifies that pulse extension, when configured, behaves so that the interrupt and current value are correctly extended for one, two, or three timer clock cycles as required.

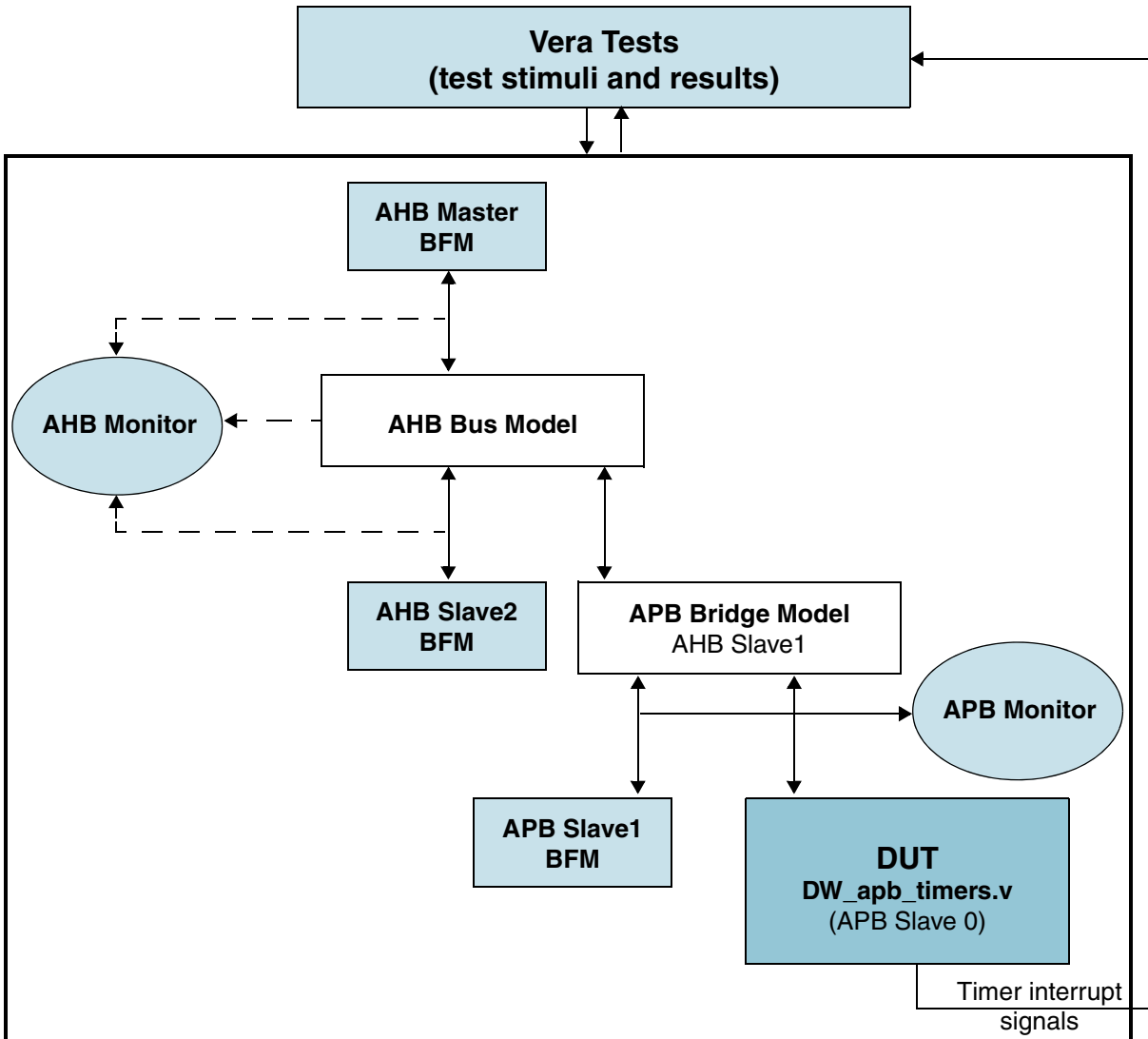
**Note**

All tests have achieved maximum RTL code coverage and use the APB Interface to dynamically program memory-mapped registers during tests.

9.2 Overview of DW_apb_timers Testbench

As illustrated in [Figure 9-1](#), the DW_apb_timers Verilog testbench includes an instantiation of the design under test (DUT), AHB and APB Bridge bus models, and a Vera shell.

Figure 9-1 DW_apb_timers Testbench



The Vera shell consists of an AHB master bus functional model (BFM), two AHB slave BFM, an AHB monitor, APB slave BFM, an APB monitor, test stimuli, BFM configuration, and test results. The AHB monitor tracks activity from the AHB master and slave BFM; the APB monitor oversees activity from the APB slave BFM.

The testbench checks for all possible user configurations selected in the Configure Component activity of coreConsultant. The testbench also determines if the component is AMBA-compliant.

10

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 for the slave interface of APB peripherals:

10.1 Reading and Writing from an APB Slave

When writing to and reading from DesignWare APB slaves, you should consider the following:

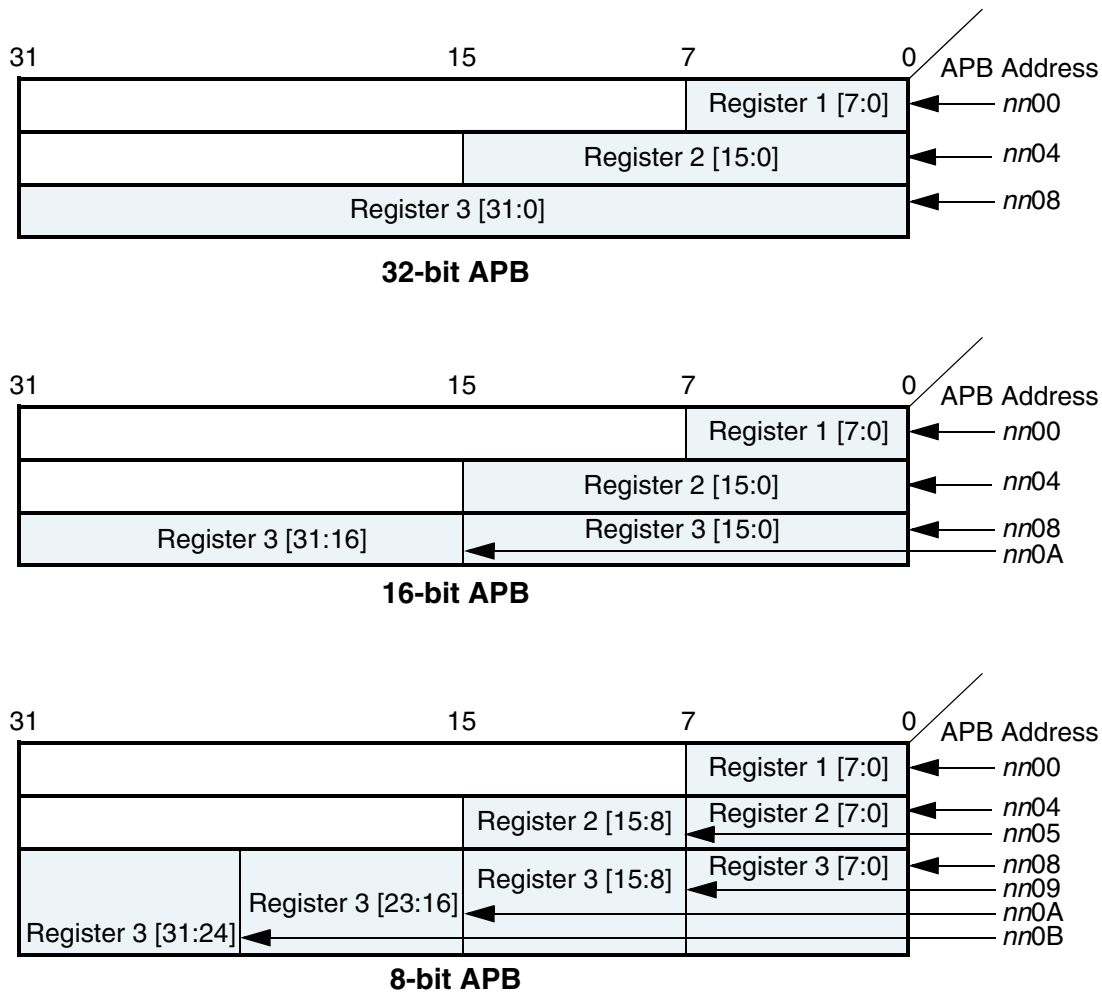
- The size of the APB peripheral should always be set equal to the size of the APB data bus, if possible.
- The APB bus has no concept of a transfer size or a byte lane, unlike the DW_ahb.
- The APB slave subsystem is little endian; the DW_apb performs the conversion from a big-endian AHB to the little-endian APB.
- All APB slave programming registers are aligned on 32-bit boundaries, irrespective of the APB bus size.
- The maximum APB_DATA_WIDTH is 32 bits. Registers larger than this occupies more than one location in the memory map.
- The DW_apb does not return any ERROR, SPLIT, or RETRY responses; it always returns an OKAY response to the AHB.
- For all bus widths:
 - In the case of a read transaction, registers less than the full bus width returns zeros in the unused upper bits.
 - Writing to bit locations larger than the register width does not have any effect. Only the pertinent bits are written to the register.
- The APB slaves do not need the full 32-bit address bus, paddr. The slaves include the lower bits even though they are not actually used in a 32- or 16-bit system.

10.1.1 Reading From Unused Locations

Reading from an unused location or unused bits in a particular register always returns zeros. Unlike an AHB slave interface, which would return an error, there is no error mechanism in an APB slave and, therefore, in the DW_apb.

The following sections show the relationship between the register map and the read/write operations for the three possible APB_DATA_WIDTH values: 8-, 16-, and 32-bit APB buses.

Figure 10-1 Read/Write Locations for Different APB Bus Data Widths



10.1.2 32-bit Bus System

For 32-bit bus systems, all programming registers can be read or written with one operation, as illustrated in the previous figure.

Because all registers are on 32-bit boundaries, *paddr[1:0]* is not actually needed in the 32-bit bus case. But these bits still exist in the configured code for usability purposes.



Note

If you write to an address location not on a 32-bit boundary, the bottom bits are ignored/not used.

10.1.3 16-bit Bus System

For 16-bit bus systems, two scenarios exist, as illustrated in the previous picture:

1. The register to be written to or read from is less than or equal to 16 bits

In this case, the register can be read or written with one transaction. In the case of a read transaction, registers less than 16 bits wide returns zeros in the un-used bits. Writing to bit locations larger than the register width causes nothing to happen, i.e. only the pertinent bits are written to the register.

2. The register to be written to or read from is >16 and ≤ 32 bits

In this case, two AHB transactions are required, which in turn creates two APB transactions, to read or write the register. The first transaction should read/write the lower two bytes (half-word) and the second transaction the upper half-word.

Because the bus is reading a half-word at a time, `paddr[0]` is not actually needed in the 16-bit bus case. But these bits still exist in the configured code for connectivity purposes.

**Note**

If you write to an address location not on a 16-bit boundary, the bottom bits are ignored/not used.

10.1.4 8-bit Bus System

For 8-bit bus systems, three scenarios exist, as illustrated in the previous picture:

1. The register to be written to or read from is less than or equal to 8 bits

In this case, the register can be read or written with one transaction. In the case of a read transaction, registers less than 8 bits wide returns zeros in the unused bits. Writing to bit locations larger than the register width causes nothing to happen, that is, only the pertinent bits are written to the register.

2. The register to be written to or read from is >8 and ≤ 16 bits

In this case, two AHB transactions are required, which in turn creates two APB transactions, to read or write the register. The first transaction should read/write the lower byte and the second transaction the upper byte.

3. The register to be written to or read from is >16 and ≤ 32 bits

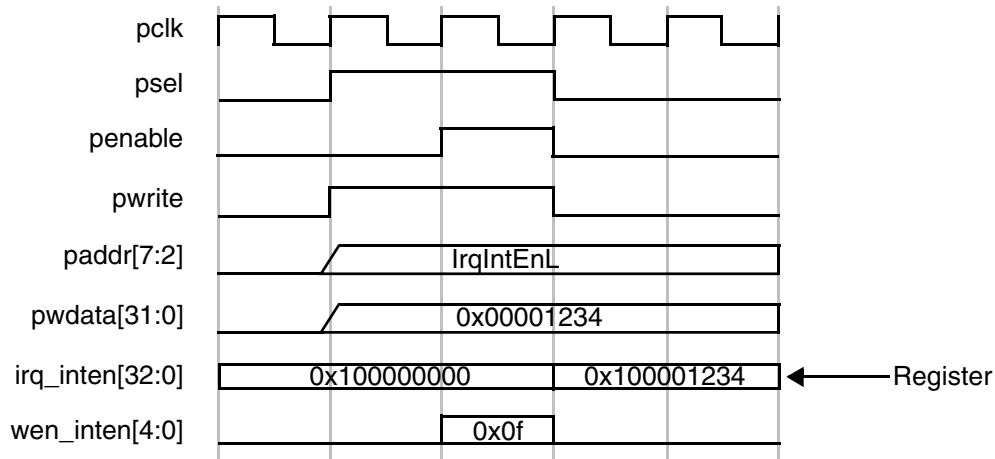
In this case, four AHB transactions are required, which in turn creates four APB transactions, to read or write the register. The first transaction should read/write the lower byte and the second transaction the second byte, and so on.

Because the bus is reading a byte at a time, all lower bits of `paddr` are decoded in the 8-bit bus case.

10.2 Write Timing Operation

A timing diagram of an APB write transaction for an APB peripheral register (an earlier version of the DW_apb_ictl) is shown in the following figure. Data, address, and control signals are aligned. The APB frame lasts for two cycles when `psel` is high.

Figure 10-2 APB Write Transaction



A write can occur after the first phase with `penable` low, or after the second phase when `penable` is high. The second phase is preferred and is used in all APB slave components. The timing diagram is shown with the write occurring after the second phase. Whenever the address on `paddr` matches a corresponding address from the memory map and provided `psel`, `pwrite`, and `penable` are high, then the corresponding register write enable is generated.

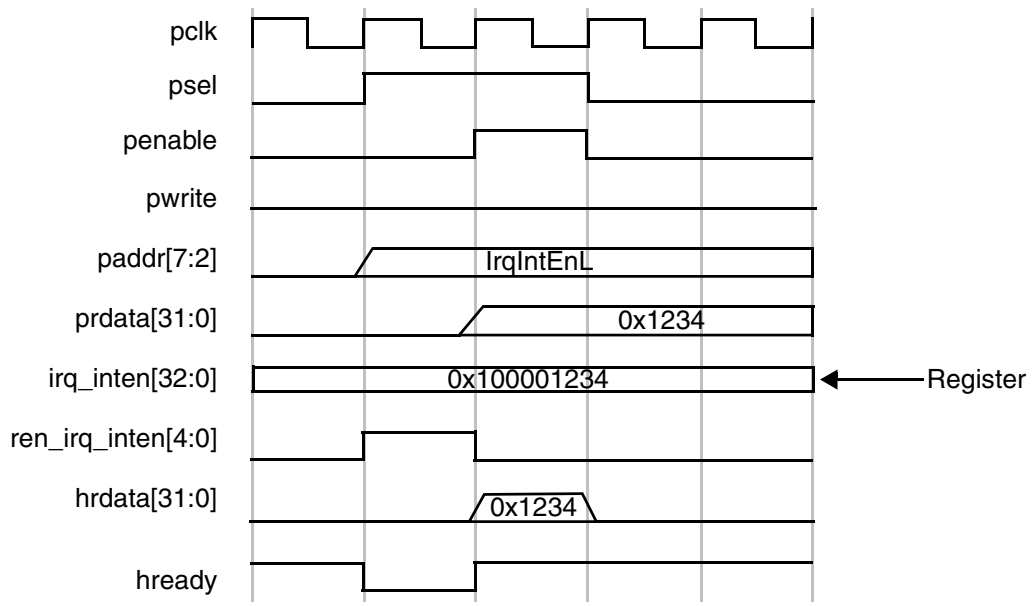
A write from the AHB to the APB does not require the AHB system bus to stall until the transfer on the APB has completed. A write to the APB can be followed by a read transaction from another AHB peripheral (not the DW_apb).

The timing example is a 33-bit register and a 32-bit APB data bus. To write this, 5 byte enables would be generated internally. The example shows writing to the first 32 bits with one write transaction.

10.3 Read Timing Operation

A timing diagram of an APB read transaction for an APB peripheral (an earlier version of the DW_apb_ictl) is shown in the following figure. The APB frame lasts for two cycles, when psel is high.

Figure 10-3 APB Read Transaction



Whenever the address on paddr matches the corresponding address from the memory map – psel is high, pwrite and penable are low – then the corresponding read enable is generated. The read data is registered within the peripheral before passing back to the master through the DW_apb and DW_ahb.

The qualification of the read-back data with hready from the bridge is shown in the timing diagram, but this does not form part of the APB interface. The read happens in the first APB cycle and is passed straight back to the AHB master in the same cycles as it passes through the bridge. By returning the data immediately to the AHB bus, the bridge can release control of the AHB data bus faster. This is important for systems where the APB clock is slower than the AHB clock.

Once a read transaction is started, it is completed and the AHB bus is held until the data is returned from the slave



Note

If a read enable is not active, then the previously read data is maintained on the read-back data bus.

10.4 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>
```

10.5 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.

10.5.1 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 10-1 Upper Byte Generation

Load Register Width	Upper Byte Bus 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

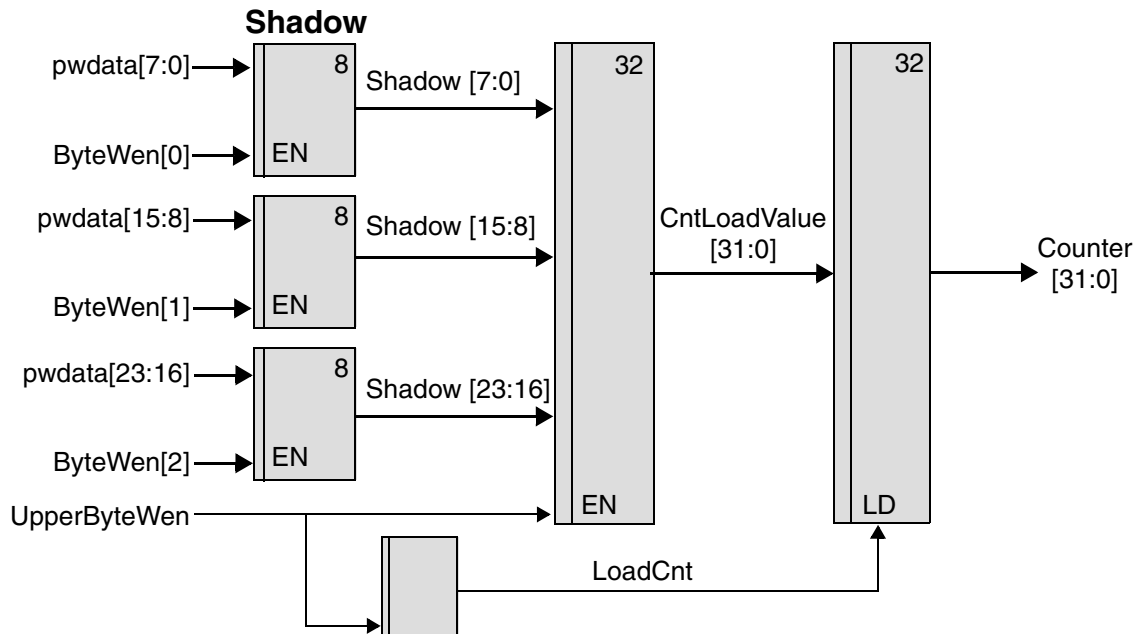
There are three relationship cases to be considered for the processor and peripheral clocks:

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

10.5.1.1 Identical Clocks

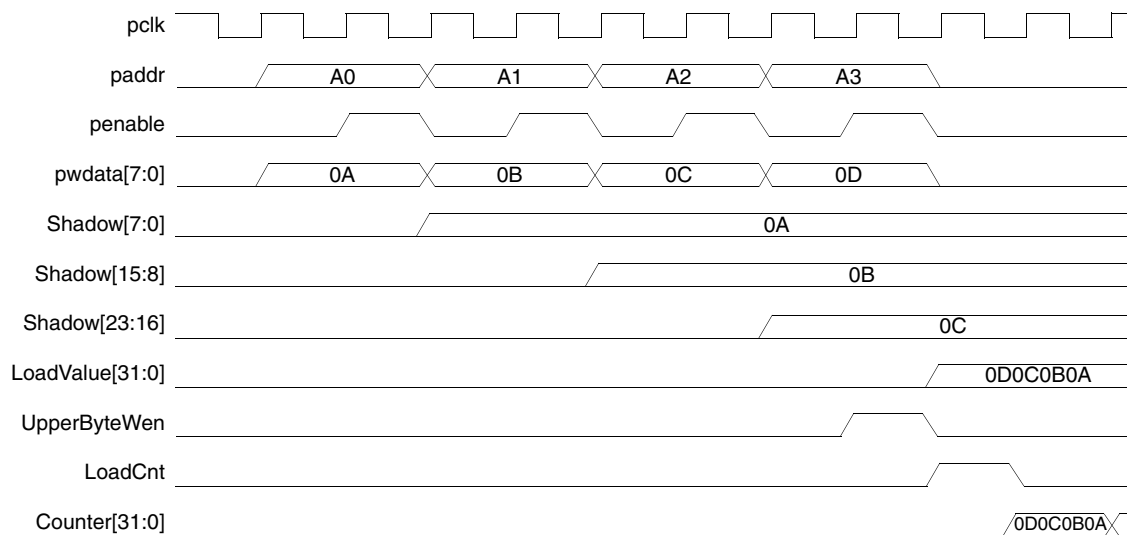
The following figure illustrates an RTL diagram for the circuitry required to implement the coherent write transaction when the APB bus clock and peripheral clocks are identical.

Figure 10-4 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 10-5 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.

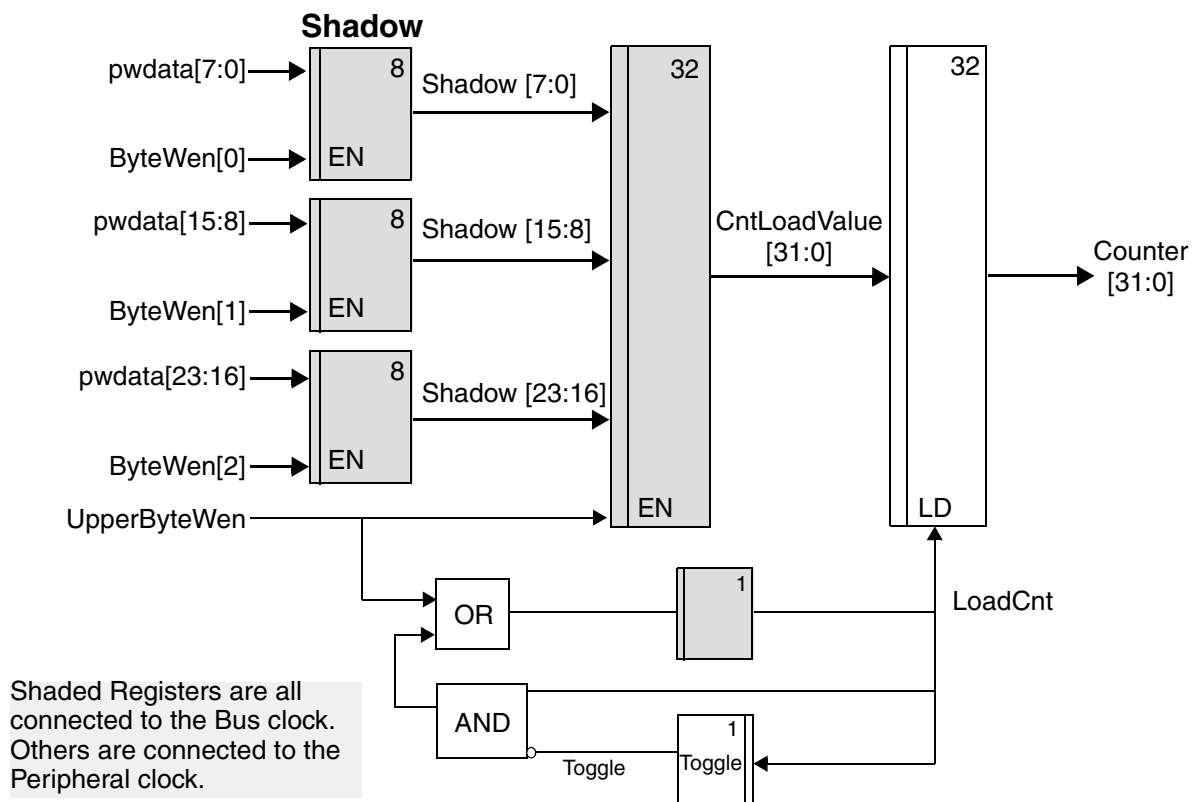
By using the shadow registers, the CntLoadValue is kept stable until it can be changed in one cycle. This allows the counter to be loaded in one access and the state of the counter is not affected by the latency in programming it. When there is a new value to be loaded into the counter initially, this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

10.5.1.2 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.

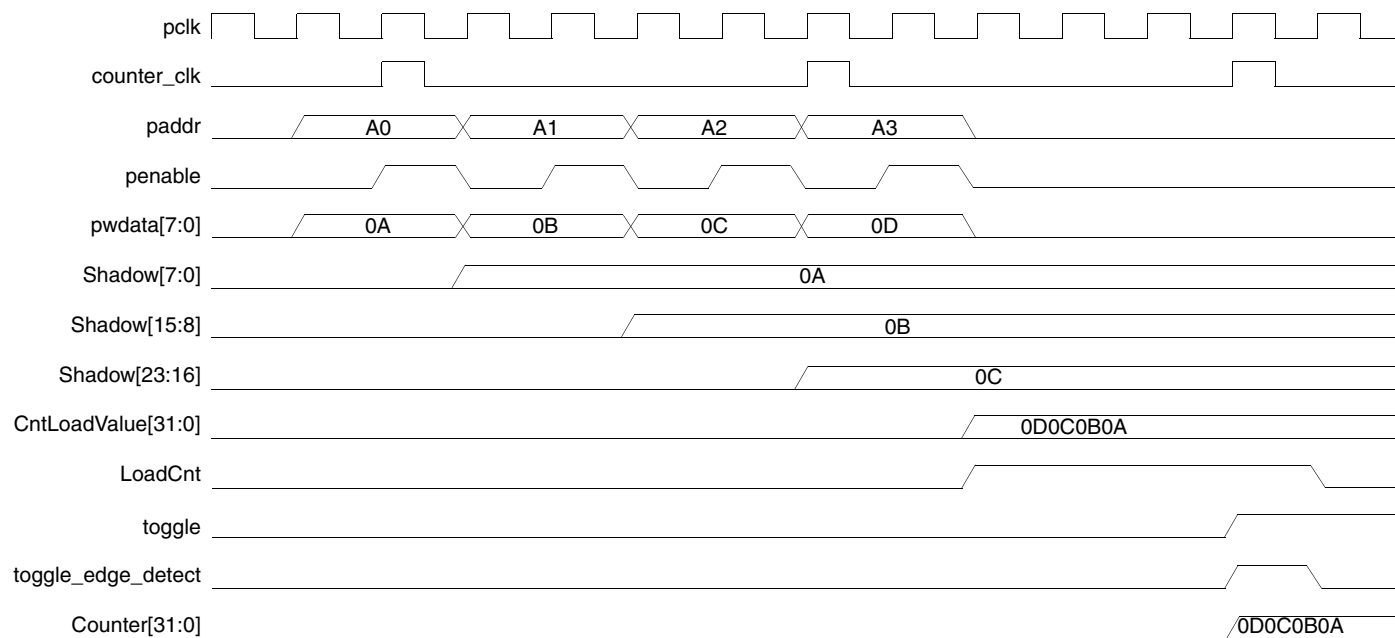
The following figure shows an RTL diagram for the circuitry required to implement the coherent write when the bus and peripheral clocks are synchronous.

Figure 10-6 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.

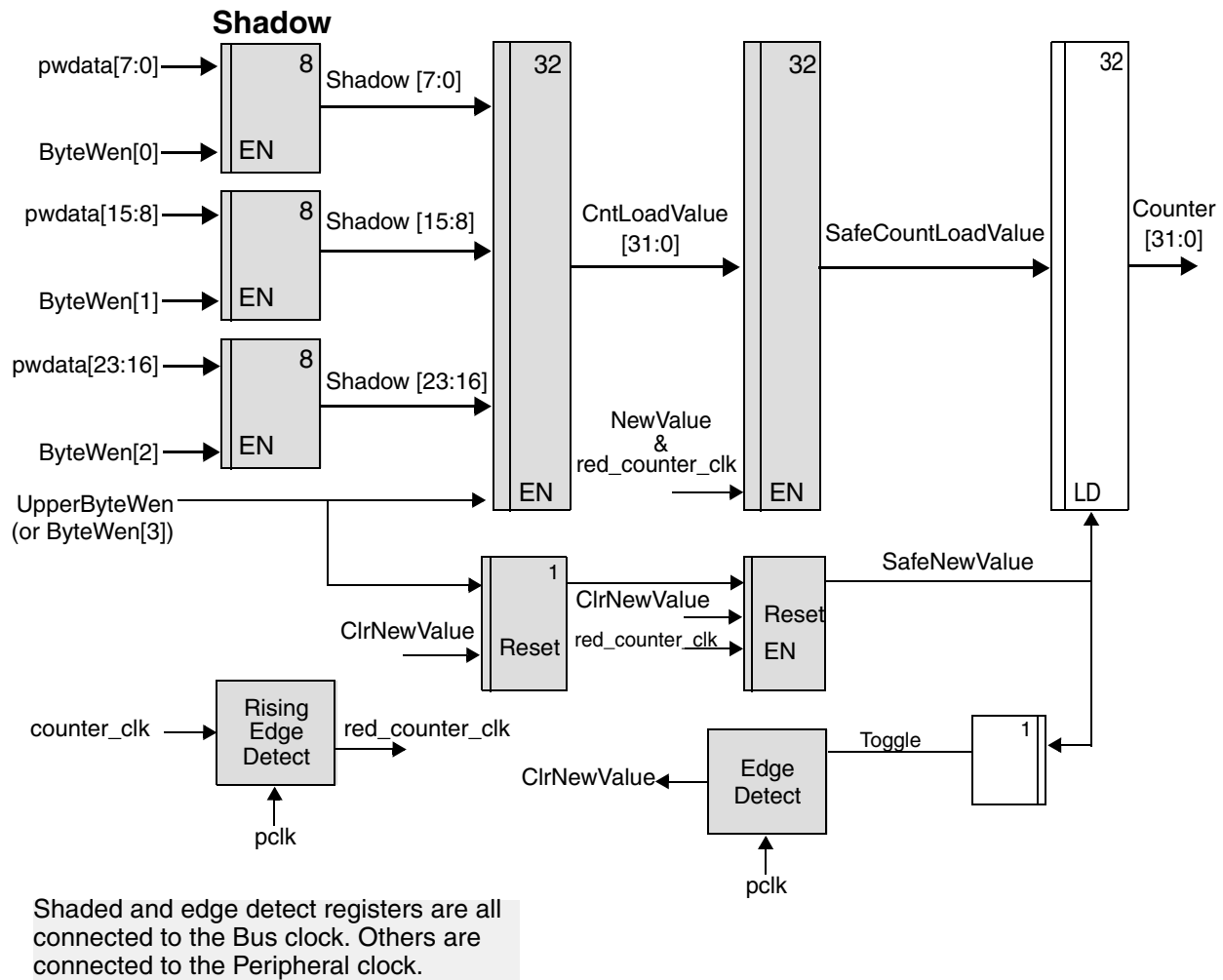
Figure 10-7 Coherent Loading – Synchronous Clocks



10.5.1.3 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 10-8 Coherent Loading – Asynchronous Clocks



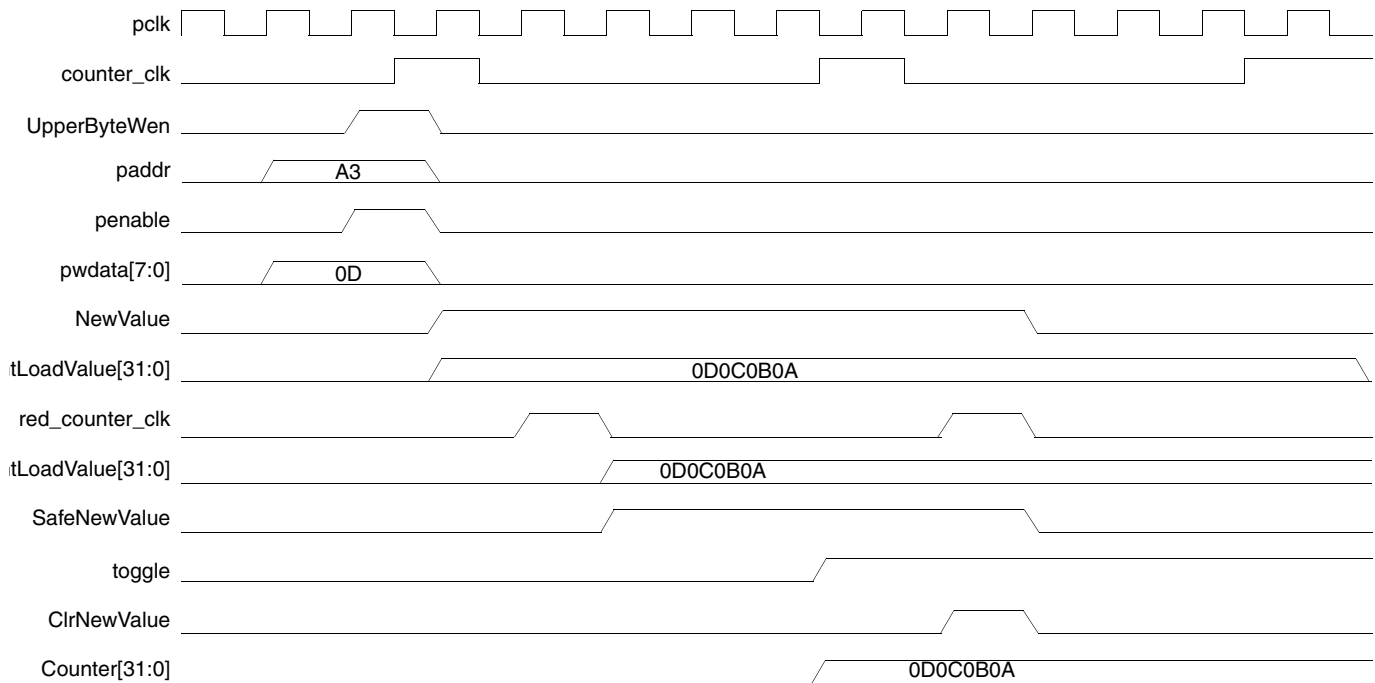
When the clocks are asynchronous, you need to transfer the contents of the register from one clock domain to another. It is not desirable to transfer the entire register through meta-stability registers, as coherency is not guaranteed with this method. The circuitry needed requires the processor clock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

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

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

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

Figure 10-9 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.

10.5.2 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 10-2 Lower Byte Generation

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

Table 10-2 Lower Byte Generation

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

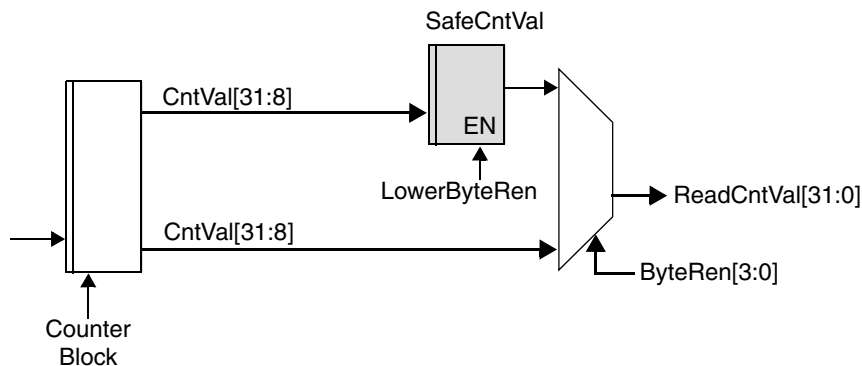
Depending on the bus width and the register width, there may be no need to save the upper bits because the entire register is read in one access, in which case there is no problem with coherency. When the lower byte is read, the remaining upper bytes within the counter register are transferred into a 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:

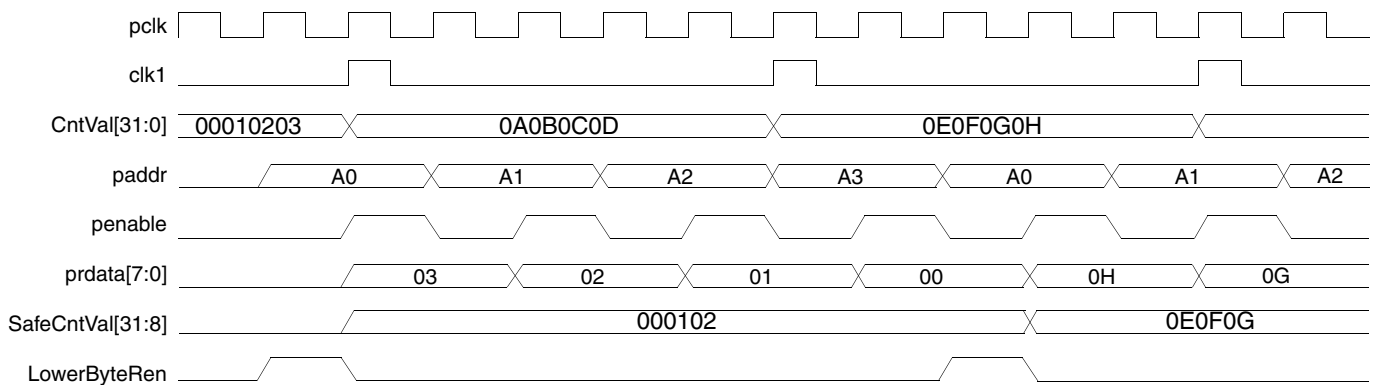
- Identical and/or synchronous
- Asynchronous

10.5.2.1 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 10-10 Coherent Registering – Synchronous Clocks

Shaded registers are clocked with the processor clock.

Figure 10-11 Coherent Registering – Synchronous Clocks**10.5.2.2 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**

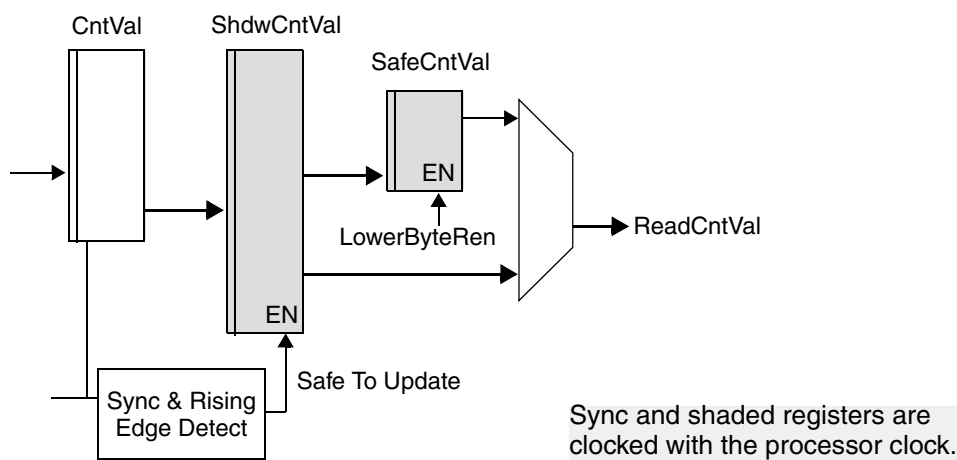
You must read LSB to MSB when the bus width is narrower than the counter width.

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 10-12 Coherency and Shadow Registering – Asynchronous Clocks



10.6 Performance

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

10.7 Area

This section provides information to help you configure area for your configuration.

The following table includes synthesis results that have been generated using the TSMC 65nm technology library.

Table 10-3 Synthesis Results Using TSMC 65 nm Technology Library

Configuration	Operating Frequency	Gate Count
Default Configuration	pclk: 166.67 MHz timer_1_clk: 83.33 MHz timer_2_clk: 83.33 MHz	1845 gates
Minimum Configuration: NUM_TIMERS=1 APB_DATA_WIDTH=8 TIMER_WIDTH_1=8 TIM_METASTABLE_1=0 TIM_PULSE_EXTD_1=0	pclk: 166.67 MHz timer_1_clk: 83.33 MHz	338 gates

Table 10-3 Synthesis Results Using TSMC 65 nm Technology Library (Continued)

Configuration	Operating Frequency	Gate Count
Maximum Configuration: NUM_TIMERS=8 APB_DATA_WIDTH=32 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 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 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	pclk: 166.67 MHz timer_1_clk: 83.33 MHz timer_2_clk: 83.33 MHz timer_3_clk: 83.33 MHz timer_4_clk: 83.33 MHz timer_5_clk: 83.33 MHz timer_6_clk: 83.33 MHz timer_7_clk: 83.33 MHz timer_8_clk: 83.33 MHz	7204 gates

The following table includes synthesis results that have been generated using the TSMC 28nm technology library.

Table 10-4 Synthesis Results Using TSMC 28nm Technology Library

Configuration	Operating Frequency	Gate Count
Default Configuration	pclk: 166.67 MHz timer_1_clk: 83.33 MHz timer_2_clk: 83.33 MHz	1924 gates
Minimum Configuration: NUM_TIMERS=1 APB_DATA_WIDTH=8 TIMER_WIDTH_1=8 TIM_METASTABLE_1=0 TIM_PULSE_EXTD_1=0	pclk: 166.67 MHz timer_1_clk: 83.33 MHz	352 gates

Table 10-4 Synthesis Results Using TSMC 28nm Technology Library (Continued)

Configuration	Operating Frequency	Gate Count
Maximum Configuration: NUM_TIMERS=8 APB_DATA_WIDTH=32 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 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 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	pclk: 166.67 MHz timer_1_clk: 83.33 MHz timer_2_clk: 83.33 MHz timer_3_clk: 83.33 MHz timer_4_clk: 83.33 MHz timer_5_clk: 83.33 MHz timer_6_clk: 83.33 MHz timer_7_clk: 83.33 MHz timer_8_clk: 83.33 MHz	7614 gates

10.7.1 Power Consumption

The following table provides information about the power consumption of the DW_apb_timers using the TSMC 65nm technology library and how it affects performance.

Table 10-5 Power Consumption of DW_apb_timers Using TSMC 65 nm Technology Library

Configuration	Operating Frequency	Static Power Consumption	Dynamic Power Consumption
Default Configuration	pclk: 166.67 MHz timer_1_clk: 83.33 MHz timer_2_clk: 83.33 MHz	456.1735 nW	169.8569 μ W
Minimum Configuration: NUM_TIMERS=1 APB_DATA_WIDTH=8 TIMER_WIDTH_1=8 TIM_METASTABLE_1=0 TIM_PULSE_EXTD_1=0	pclk: 166.67 MHz timer_1_clk: 83.33 MHz	82.9775 nW	31.6783 μ W

Table 10-5 Power Consumption of DW_apb_timers Using TSMC 65 nm Technology Library (Continued)

Configuration	Operating Frequency	Static Power Consumption	Dynamic Power Consumption
Maximum Configuration: NUM_TIMERS=8 APB_DATA_WIDTH=32 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 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 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	pclk: 166.67 MHz timer_1_clk: 83.33 MHz timer_2_clk: 83.33 MHz timer_3_clk: 83.33 MHz timer_4_clk: 83.33 MHz timer_5_clk: 83.33 MHz timer_6_clk: 83.33 MHz timer_7_clk: 83.33 MHz timer_8_clk: 83.33 MHz	1.7756 μ W	607.3238 μ W

The following table provides information about the power consumption of the DW_apb_timers using the TSMC 28nm technology library and how it affects performance.

Table 10-6 Power Consumption of DW_apb_timers Using TSMC 28 nm Technology Library

Configuration	Operating Frequency	Static Power Consumption	Dynamic Power Consumption
Default Configuration	pclk: 166.67 MHz timer_1_clk: 83.33 MHz timer_2_clk: 83.33 MHz	194.2237 μ W	107.8008 μ W
Minimum Configuration: NUM_TIMERS=1 APB_DATA_WIDTH=8 TIMER_WIDTH_1=8 TIM_METASTABLE_1=0 TIM_PULSE_EXTD_1=0	pclk: 166.67 MHz timer_1_clk: 83.33MHz	34.9149 μ W	19.7647 μ W

Table 10-6 Power Consumption of DW_apb_timers Using TSMC 28 nm Technology Library (Continued)

Configuration	Operating Frequency	Static Power Consumption	Dynamic Power Consumption
Maximum Configuration: NUM_TIMERS=8 APB_DATA_WIDTH=32 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 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 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	pclk: 166.67 MHz timer_1_clk: 83.33 MHz timer_2_clk: 83.33 MHz timer_3_clk: 83.33 MHz timer_4_clk: 83.33 MHz timer_5_clk: 83.33 MHz timer_6_clk: 83.33 MHz timer_7_clk: 83.33 MHz timer_8_clk: 83.33 MHz	776.3969 μ W	387.4334 μ W

A

Synchronizer Methods

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

This appendix contains the following sections:

- [“Synchronizers Used in DW_apb_timers”](#) on page 98
- [“Synchronizer 1: Simple Double Register Synchronizer \(DW_apb_timers\)”](#) on page 99

**Note**

The DesignWare Building Blocks (DWBB) contains several synchronizer components with functionality similar to methods documented in this appendix. For more information about the DWBB synchronizer components go to:

www.synopsys.com/products/designware/docs/doc/dwf/datasheets/interface_cdc_overview.pdf

A.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-1](#). Note that certain BCM modules are contained in other BCM modules, as they are used as building blocks

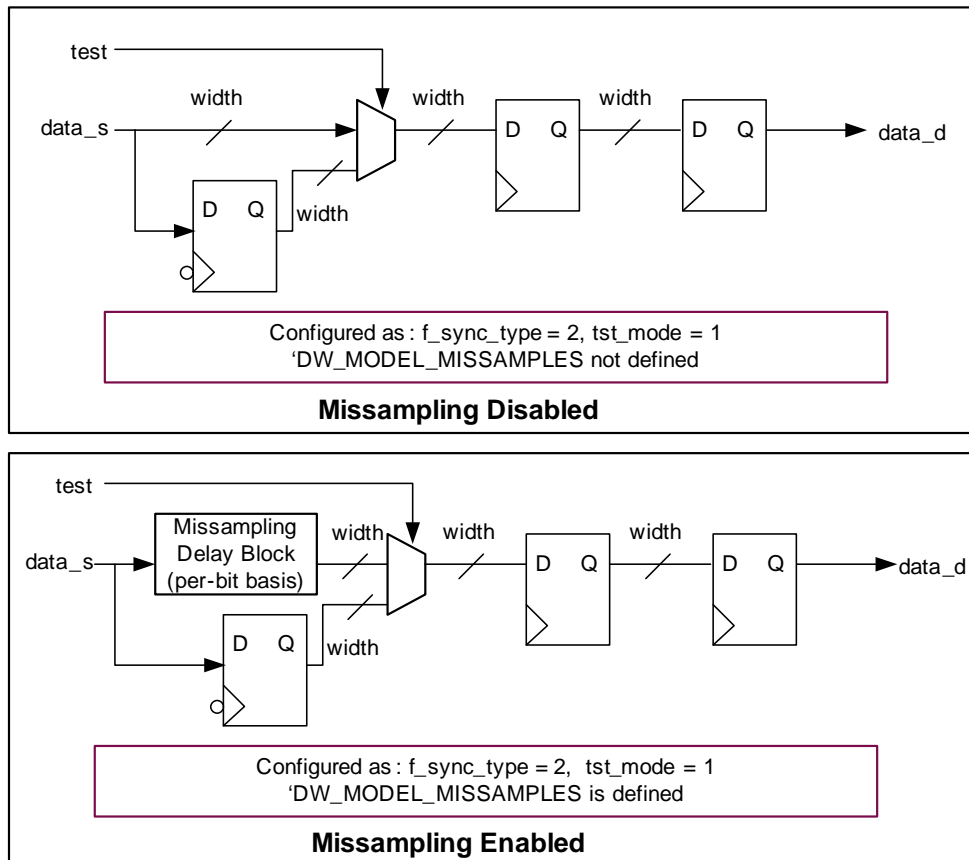
Table A-1 Synchronizers used in DW_apb_timers

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

A.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

Glossary

active command queue	Command queue from which a model is currently taking commands; see also command queue.
activity	A set of functions in coreConsultant that step you through configuration, verification, and synthesis of a selected core.
AHB	Advanced High-performance Bus — high-performance system backbone bus. AHB supports the efficient connection of processors, on-chip memories and off-chip external memory interfaces (ARM Limited specification).
AMBA	Advanced Microcontroller Bus Architecture — a trademarked name by ARM Limited that defines an on-chip communication standard for high speed microcontrollers.
APB	Advanced Peripheral Bus — optimized for minimal power consumption and reduced interface complexity to support peripheral functions (ARM Limited specification).
APB bridge	DW_apb submodule that converts protocol between the AHB bus and APB bus.
application design	Overall chip-level design into which a subsystem or subsystems are integrated.
arbiter	AMBA bus submodule that arbitrates bus activity between masters and slaves.
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.

bus bridge	Logic that handles the interface and transactions between two bus standards, such as AHB and APB. See APB bridge.
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.
core	Any configurable block of synthesizable IP that can be instantiated as a single entity (VHDL) or module (Verilog) in a design. Core is the preferred term for a big piece of IIP. Anything that requires coreConsultant for configuration, as well as anything in the DesignWare Cores library, is a core.
core developer	Person or company who creates or packages a reusable core. All the cores in the DesignWare Library are developed by Synopsys.
core integrator	Person who uses coreConsultant or coreAssembler to incorporate reusable cores into a system-level design.
coreAssembler	Synopsys product that enables automatic connection of a group of cores into a subsystem. Generates RTL and gate-level views of the entire subsystem.
coreConsultant	A Synopsys product that lets you configure a core and generate the design views and synthesis views you need to integrate the core into your design. Can also synthesize the core and run the unit-level testbench supplied with the core.
coreKit	An unconfigured core and associated files, including the core itself, a specified synthesis methodology, interfaces definitions, and optional items such as verification environment files and core-specific documentation.
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.
Design View	A simulation model for a core generated by coreConsultant.
DesignWare Synthesizable Components	The Synopsys name for the collection of AMBA-compliant coreKits and verification models delivered with DesignWare and used with coreConsultant or coreAssembler to quickly build DesignWare Synthesizable Component designs.

DesignWare cores	A specific collection of synthesizable cores that are licensed individually. For more information, refer to www.synopsys.com/designware .
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.
MacroCell	Bigger IP blocks (6811, 8051, memory controller) available in the DesignWare Library and delivered with coreConsultant.
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.
subsystem	In relation to coreAssembler, highest level of RTL that is automatically generated.
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.
workspace	A network location that contains a personal copy of a component or subsystem. After you configure the component or subsystem (using coreConsultant or coreAssembler), the workspace contains the configured component/subsystem and generated views needed for integration of the component/subsystem at the top level.
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.

Index

- A**
 - active command queue
 - definition [101](#)
 - activity
 - definition [101](#)
 - AHB
 - definition [101](#)
 - AMBA
 - definition [101](#)
 - APB
 - definition [101](#)
 - APB bridge
 - definition [101](#)
 - APB interface, read to/writing from [53](#)
 - APB_DATA_WIDTH [43](#)
 - application design
 - definition [101](#)
 - arbiter
 - definition [101](#)
- B**
 - BFM
 - definition [101](#)
 - big-endian
 - definition [101](#)
 - Block diagram, of DW_apb_timers [15](#)
 - blocked command stream
 - definition [101](#)
 - blocking command
 - definition [101](#)
 - bus bridge
 - definition [102](#)
- C**
 - Clock boundaries [36](#)
 - Clock domains [37](#)
 - Coherency
 - about [80](#)
 - read [86](#)
 - write [80](#)
 - command channel
 - definition [102](#)
 - command stream
 - definition [102](#)
 - component
 - definition [102](#)
 - configuration
 - definition [102](#)
 - configuration intent
 - definition [102](#)
 - core
 - definition [102](#)
 - core developer
 - definition [102](#)
 - core integrator
 - definition [102](#)
 - coreAssembler
 - definition [102](#)
 - overview of usage flow [24](#)
 - coreConsultant
 - definition [102](#)
 - overview of usage flow [18](#)
 - coreKit
 - definition [102](#)
 - Customer Support [10](#)
 - cycle command
 - definition [102](#)
- D**
 - decoder
 - definition [102](#)
 - design context
 - definition [102](#)
 - design creation
 - definition [102](#)

- Design for test [41](#)
- Design View
 - definition [102](#)
- DesignWare cores
 - definition [103](#)
- DesignWare Library
 - definition [103](#)
- DesignWare Synthesizable Components
 - definition [102](#)
- dual role device
 - definition [103](#)
- DW_ahb_dmac
 - testbench
 - overview of tests [71](#)
- DW_apb
 - slaves
 - read timing operation [79](#)
 - write timing operation [78](#)
- DW_apb_timers
 - block diagram of [15](#)
 - description [43](#)
 - design for test [41](#)
 - functional description of [15](#)
 - memory map [69](#)
 - parameters [43](#)
 - programming of [53](#), [69](#)
 - setting operating mode [33](#)
 - system registers [55](#)
 - testbench
 - overview of [73](#)
 - timer registers [54](#)
- E**
- endian
 - definition [103](#)
- Environment, licenses [16](#)
- F**
- Full-Functional Mode
 - definition [103](#)
- Functional description [15](#), [43](#)
- G**
- Generating interrupts [34](#)
- GPIO
 - definition [103](#)
- GTECH
 - definition [103](#)
- H**
- hard IP
 - definition [103](#)
- HDL
 - definition [103](#)
- I**
- IIP
 - definition [103](#)
- implementation view
 - definition [103](#)
- instantiate
 - definition [103](#)
- interface
 - definition [103](#)
- Interfacing
 - to APB interface [53](#)
- Interrupt, handling and generation of [34](#)
- IP
 - definition [103](#)
- L**
- Licenses [16](#)
- little-endian
 - definition [103](#)
- M**
- MacroCell
 - definition [103](#)
- master
 - definition [103](#)
- Memory map, of DW_apb_timers [69](#)
- Metastability, about [36](#)
- model
 - definition [103](#)
- monitor
 - definition [103](#)
- N**
- non-blocking command
 - definition [103](#)
- NUM_TIMERS [43](#)
- O**
- Operating mode, setting [33](#)
- Output files
 - GTECH [29](#)
 - RTL-level [28](#)
 - Simulation model [29](#)

synthesis [29](#)
 verification [30](#)

P

Parameters, of DW_apb_timers [43](#)

peripheral
 definition [103](#)

Programming DW_apb_timers
 memory map [53](#), [69](#)
 registers [56](#)

R

Read coherency
 about [86](#)
 and asynchronous clocks [88](#)
 and synchronous clocks [87](#)
 Reading, from unused locations [75](#)

Registers
 system [55](#)
 timers [54](#)
 TimersEOI [55](#)
 TimersIntStatus [55](#)
 TimersRawIntStatus [55](#)

RTL
 definition [104](#)

S

scan_mode [41](#)
 SDRAM
 definition [104](#)
 SDRAM controller
 definition [104](#)
 Setting, operating mode [33](#)
 Simple double register synchronizer [99](#)
 Simulation
 of DW_apb_timers coreKit [73](#)
 slave
 definition [104](#)
 SoC
 definition [104](#)
 SoC Platform
 AHB contained in [13](#)
 APB, contained in [13](#)
 defined [13](#)
 soft IP
 definition [104](#)
 static controller
 definition [104](#)

subsystem
 definition [104](#)
 Synchronizer
 simple double register [99](#)
 synthesis intent
 definition [104](#)
 synthesizable IP
 definition [104](#)
 System registers [55](#)

T

technology-independent
 definition [104](#)
 Testsuite Regression Environment (TRE)
 definition [104](#)
 TIM_INTR_IO [31](#), [44](#)
 TIM_INTRPT_PLRITY [44](#)
 TIM_METASTABLE [45](#)
 TIM_NEWMODE [44](#)
 TIM_PULSE_EXTD [46](#)
 Timer registers [54](#)
 TIMER_HAS_TOGGLE [45](#)
 TIMER_WIDTH [33](#), [45](#)
 TimersEOI [55](#)
 TimersIntStatus [55](#)
 TimersRawIntStatus [55](#)
 Timing
 read operation of DW_apb slave [79](#)
 write operation of DW_apb slave [78](#)
 TRE
 definition [104](#)

V

Vera, overview of tests [71](#)
 Verification
 and Vera tests [71](#)
 of DW_apb_timers coreKit [73](#)
 VIP
 definition [104](#)

W

workspace
 definition [104](#)
 wrap
 definition [104](#)
 wrapper
 definition [104](#)

Write coherency

about [80](#)and asynchronous clocks [85](#)and identical clocks [82](#)and synchronous clocks [83](#)**Z**

zero-cycle command

definition [104](#)