SYNOPSYS®

DesignWare DW_apb_uart Databook

DesignWare Synthesizable Components for AMBA 2

DW_apb_uart

Version 3.04a

January 20, 2006

Copyright Notice and Proprietary Information

Destination Control Statement

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

Disclaimer

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

Registered Trademarks (®)

Synopsys, AMPS, Arcadia, C Level Design, C2HDL, C2V, C2VHDL, Cadabra, Calaveras Algorithm, CATS, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSPICE, Hypermodel, I, iN-Phase, in-Sync, Leda, MAST, Meta, Meta-Software, ModelAccess, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PowerMill, PrimeTime, RailMill, Raphael, RapidScript, Saber, SiVL, SNUG, SolvNet, Stream Driven Simulator, Superlog, System Compiler, Testify, TetraMAX, TimeMill, TMA, VCS, Vera, and Virtual Stepper are registered trademarks of Synopsys, Inc.

Trademarks (™)

abraCAD, abraMAP, Active Parasitics, AFGen, Apollo, Apollo II, Apollo-DPII, Apollo-GA, ApolloGAII, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanTestchip, AvanWaves, BCView, Behavioral Compiler, BOA, BRT, Cedar, ChipPlanner, Circuit Analysis, Columbia, Columbia-CE, Comet 3D, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, Cyclelink, Davinci, DC Expert, DC Expert Plus, DC Professional, DC Ultra, DC Ultra Plus, Design Advisor, Design Analyzer, Design Vision, DesignerHDL, DesignTime, DFM-Workbench, Direct RTL, Direct Silicon Access, Discovery, DW8051, DWPCI, Dynamic-Macromodeling, Dynamic Model Switcher, ECL Compiler, ECO Compiler, EDAnavigator, Encore, Encore PQ, Evaccess, ExpressModel, Floorplan Manager, Formal Model Checker, FoundryModel, FPGA Compiler II, FPGA Express, Frame Compiler, Galaxy, Gatran, HDL Advisor, HDL Compiler, Hercules, Hercules-Explorer, Hercules-II, Hierarchical Optimization Technology, High Performance Option, HotPlace, HSPICE-Link, iN-Tandem, Integrator, Interactive Waveform Viewer, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, JVXtreme, Liberty, Libra-Passport, Library Compiler, Libra-Visa, Magellan, Mars, Mars-Rail, Mars-Xtalk, Medici, Metacapture, Metacircuit, Metamanager, Metamixsim, Milkyway, ModelSource, Module Compiler, MS-3200, MS-3400, Nova Product Family, Nova-ExploreRTL, Nova-Trans, Nova-VeriLint, Nova-VHDLlint, Optimum Silicon, Orion_ec, Parasitic View, Passport, Planet, Planet-PL, Planet-RTL, Polaris, Polaris-CBS, Polaris-MT, Power Compiler, PowerCODE, PowerGate, ProFPGA, ProGen, Prospector, Protocol Compiler, PSMGen, Raphael, Raphael-NES, RoadRunner, RTL Analyzer, Saturn, ScanBand, Schematic Compiler, Scirocco, Scirocco-i, Shadow Debugger, Silicon Blueprint, Silicon Early Access, SinglePass-SoC, Smart Extraction, SmartLicense, SmartModel Library, Softwire, Source-Level Design, Star, Star-DC, Star-MS, Star-MTB, Star-Power, Star-Rail, Star-RC, Star-RCXT, Star-Sim, Star-SimXT, Star-Time, Star-XP, SWIFT, Taurus, TimeSlice, TimeTracker, Timing Annotator, TopoPlace, TopoRoute, Trace-On-Demand, True-Hspice, TSUPREM-4, TymeWare, VCS Express, VCSi, Venus, Verification Portal, VFormal, VHDL Compiler, VHDL System Simulator, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (SM)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
All other product or company names may be trademarks of their respective owners.

Contents

Preface

About This Manual

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

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

Related Documents

To see a complete listing of documentation within the DesignWare Synthesizable Components for AMBA 2, refer to the *Guide to DesignWare AMBA IP Component Documentation*.

Note

Information on the DW_apb_uart component in this databook assumes that the reader is fully familiar with the National Semiconductor 16550 (UART) component specification. This specification can be obtained on the web at:

http://www.national.com/ds/PC/PC16550D.pdf

Information provided on IrDA SIR mode assumes that the reader is fully familiar with the IrDa Serial Infrared Physical Layer Specification. This specification can be obtained from the following website:

http://www.irda.org

Manual Overview

This manual contains the following chapters and appendixes:

Chapter 1 “Product Overview”	Provides a DesignWare AMBA System Overview, a component block diagram, basic features, and an overview of the verification environment Overview
Chapter 2 “Building and Verifying a Subsystem”	Provides getting started information that allows you to walk through the process of using the DW_apb_uart with Synopsys’ DesignWare Connect tool.

Chapter 3 “Functional Description”	Describes the functional operation of the DW_apb_uart
Chapter 4 “Parameters”	Identifies the configurable parameters supported by the DW_apb_uart
Chapter 5 “Signals”	Provides a list and description of the DW_apb_uart signals
Chapter 6 “Registers”	Describes the programmable registers of the DW_apb_uart
Chapter 7 “Programming the DW_apb_uart”	Provides information needed to program the configured DW_apb_uart
Chapter 8 “Verification”	Provides an overview of the testbench available for DW_apb_uart verification.
Chapter 9 “Integration Considerations”	Includes information you need to integrate the configured DW_apb_uart into your design
Appendix A “Building and Verifying Your DW_apb_uart”	Provides getting started information that allows you to walk through the process of using the DW_apb_uart with Synopsys’ coreConsultant tool.
Appendix B “Database Description”	Provides deliverables and reference files generated from the coreConsultant flow
Appendix C “DesignWare QuickStart Designs”	Provides getting started information that allows you to walk through the process of using the DW_apb_uart with Synopsys’ coreConsultant tool.
Appendix D “Glossary”	Provides a glossary of general terms

Typographical and Symbol Conventions

The conventions in the following table are used throughout this document:

Table 1: Documentation Conventions

Convention	Description and Example
%	Represents the UNIX prompt.
Bold	User input (text entered by the user). `% cd $LMC_HOME/hdl`
Monospace	System-generated text (prompts, messages, files, reports). `No Mismatches: 66 Vectors processed: 66 Possible"`

Table 1: Documentation Conventions (Continued)

Convention	Description and Example
Italic or `Italic`	Variables for which you supply a specific value. As a command line example: `% setenv LMC_HOME prod_dir` In body text: In the previous example, *prod_dir* is the directory where your product must be installed.
| (Vertical rule)	Choice among alternatives, as in the following syntax example: `-effort_level low | medium | high`
[ ] (Square brackets)	Enclose optional parameters: `pin1 [pin2 ... pinN]` In this example, you must enter at least one pin name (*pin1*), but others are optional ([*pin2* … *pinN*]).
TopMenu > SubMenu	Pulldown menu paths, such as: **File > Save As …**

Getting Help

If you have a question about using Synopsys products, please consult product documentation that is installed on your network or located at the root level of your Synopsys product CD-ROM (if available). You can also access documentation for DesignWare products on the Web:

- Product documentation for many DesignWare products:

  http://www.synopsys.com/designware/docs

- Datasheets for individual DesignWare IP components, located using “Search for IP”:

  http://www.synopsys.com/designware

You can also contact the Synopsys Support Center in the following ways:

- Open a call to your local support center using this page:

  http://www.synopsys.com/support/support.html

- Send an e-mail message to support_center@synopsys.com.
- Telephone your local support center:
  - United States:
    Call 1-800-245-8005 from 7 AM to 5:30 PM Pacific Time, Mon—Fri.
  - Canada:
    Call 1-650-584-4200 from 7 AM to 5:30 PM Pacific Time, Mon—Fri.
  - All other countries:
    Find other local support center telephone numbers at the following URL:

    http://www.synopsys.com/support/support_ctr

Additional Information

For additional Synopsys documentation, refer to the following page:

http://www.synopsys.com/designware/docs

For up-to-date information about the latest Synthesizable IP and verification models, visit the DesignWare home page:

http://www.synopsys.com/designware

Comments?

To report errors or make suggestions, please send e-mail to:

support_center@synopsys.com.

To report an error that occurs on a specific page, select the entire page (including headers and footers), and copy to the buffer. Then paste the buffer to the body of your e-mail message. This will provide us with information to identify the source of the problem.

1 Product Overview

The DW_apb_uart is a programmable Universal Asynchronous Receiver/Transmitter (UART). This component is an AMBA 2.0-compliant Advanced Peripheral Bus (APB) slave device and is part of the family of DesignWare AMBA Synthesizable Components.

This chapter describes the DW_apb_uart in the following sections:

DesignWare AMBA System Overview

The Synopsys DesignWare AMBA 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.

DesignWare AMBA System Block Diagram

The following figure illustrates one example of this environment, including the AHB bus, the APB Bus (includes the APB Bridge), AHB multi-layer interconnect IP, APB peripheral components, verification 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.

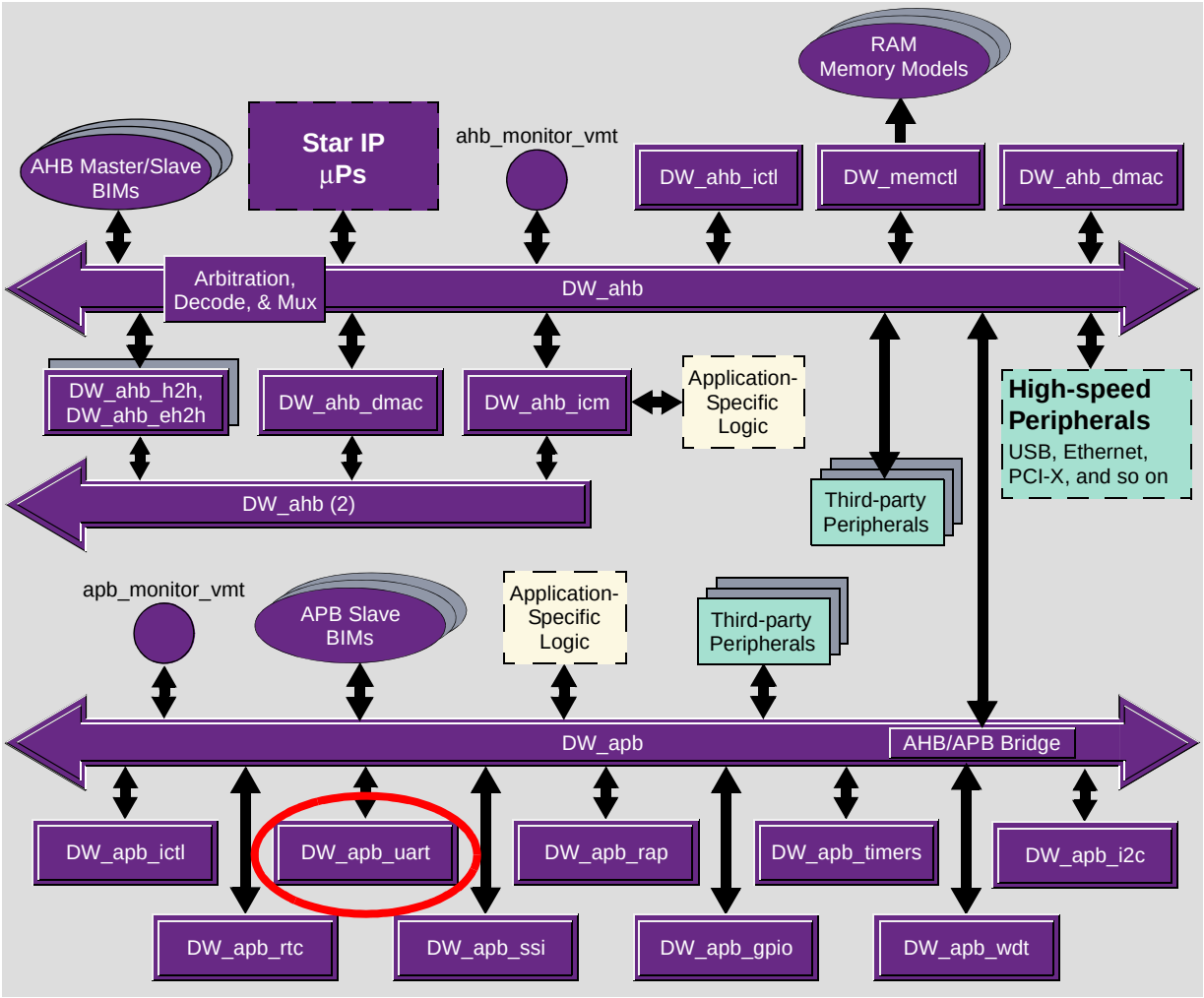


Figure 1: Example of DW_apb_uart in a Complete System

General Product Description

The Synopsys DW_apb_uart has been modeled after the industry-standard 16550. However, the register address space has been relocated to 32-bit data boundaries for APB bus implementation. It can be configured, synthesized, and verified using the Synopsys coreConsultant GUI to produce RTL.

The DW_apb_uart is used for serial communication with a peripheral, modem (data carrier equipment, DCE) or data set. Data is written from a master (CPU) over the APB bus to the UART and it is converted to serial form and transmitted to the destination device. Serial data is also received by the UART and stored for the master (CPU) to read back.

The DW_apb_uart contains registers to control the character length, baud rate, parity generation/checking, and interrupt generation. (Also see “DW_apb_uart Block Diagram” on page 14.) Although there is only one interrupt output signal (intr) from the DW_apb_uart, there are several prioritized interrupt types that can be responsible for its assertion. Each of the interrupt types can be separately enabled/disabled with the control registers.

The following paragraphs describe various functionality that you can configure into the DW_apb_uart:

Transmit and receive data FIFOs:
To reduce the time demand placed on the master by the DW_apb_uart, optional FIFOs are available to buffer transmit and receive data. This means that the master does not have to access the DW_apb_uart each time a single byte of data is received. The optional FIFOs can be selected at configuration time.

The FIFOs can be selected to be either external customer-supplied FIFO RAMs or internal DesignWare D-flip-flop based RAMs (DW_ram_r_w_s_dff). When external RAM support is chosen, both synchronous or asynchronous read-port memories are supported. When FIFO support is selected, an optional test/debug mode is available to allow the receive FIFO to be written by the master and the transmit FIFO to be read by the master.

DMA controller interface:
The DW_apb_uart can interface with a DMA controller by way of external signals (dma_tx_req_n and dma_rx_req_n) to indicate when data is ready to be read or when the transmit FIFO is empty. Additional optional DMA signals are available for DesignWare DMA controller interface compatibility (such as interface with DW_ahb_dmac).

Asynchronous clock support:
To solve problems surrounding CPU data synchronization in relation to the required serial baud clock requirements, an optional separate serial data clock can be selected. When it selected, all data crossing between the two clock domains is guaranteed by full handshaking and level-syncing synchronization.

Auto flow control:
System efficiency can be increased and software load decreased with a 16750-compatible Auto Flow Control Mode. When FIFOs and the Auto Flow Control are selected and enabled, serial data flow is automatically controlled by the request-to-send (rts_n) output and clear-to-send (cts_n) input.

Programmable Transmit Holding Register Empty (THRE) interrupt:
System performance can be increased by a Programmable Transmitter Holding Register Empty (THRE) Interrupt Mode. When FIFOs and the THRE Mode are selected and enabled, THRE Interrupts are active at and below a programmed TX FIFO threshold level. In addition, the Line Status THRE switches from indicating TX FIFO empty, to TX FIFO full. This allows software to set a threshold that keeps the transmitter FIFO from running empty whenever there is data to transmit.

Serial infrared support:
For integration in systems where Infrared SIR serial data format is required, the DW_apb_uart can be configured to have a software-programmable IrDA SIR Mode. If this mode is not selected, only the UART (RS232 standard) serial data format is available.

Increase the built-in diagnostic capabilities:
To increase the built-in diagnostic capabilities of the DW_apb_uart, the Modem Control Loopback Mode has been extended. Modem Status bits actually reflect Modem Control Register deltas, as well as the bits themselves. Also, when FIFOs and Auto Flow Control Mode are selected and enabled, the Modem Control RTS is internally looped back to the CTS to control the transmitter. This allows local testing of the Auto CTS mode. In addition, the controllability of rts_n, via the receiver FIFO threshold, can be observed via the RTS Modem Status bit. This allows local verification of the Auto RTS mode.

Level 1 and 2 debug support:
To help with debug issues, optional debug signals are available on the DW_apb_uart. To comply with level 1 and 2 debug support requirements, many internal points of interest to the debugger are available as outputs.

DW_apb_uart Block Diagram

The following list briefly describes each of the major blocks shown in Figure 2 on page 15:

- Reset block.
- APB slave interface.
- **Register block** - is responsible for the main UART functionality including control, status and interrupt generation.
- **Modem Synchronization block** - synchronizes the modem input signal.
- **FIFO block** (optional) - is responsible for FIFO control and storage (when using internal RAM) or signaling to control external RAM (when used).
- **Synchronization block** (optional) - is implemented when the peripheral is configured to have a separate serial data clock (i.e. two clock implementation).
- **Timeout Detector block** (optional) - indicates the absence of character data movement in the receiver FIFO in a given time period. This is used to generate character timeout interrupts when enabled. This block can also have an optional clock gate enable output(s) (uart_lp_req_pclk for single clock implementations or uart_lp_req_pclk and uart_lp_req_sclk for two clock implementations), to indicate that the TX and RX pipeline is clear (no data), no activity has occurred and the modem control input signals have not changed in a given time period.
- **Baud Clock Generator** - produces the transmitter and receiver baud clock along with the output reference clock signal (baudout_n).
- **Serial Transmitter** - converts the parallel data, written to the UART, into serial form and adds all additional bits, as specified by the control register, for transmission. This makeup of serial data, referred to as a character can exit the block in two forms, either serial UART format or IrDA 1.0 SIR format.
- **Serial Receiver** - converts the serial data character (as specified by the control register) received in either the UART or IrDA 1.0 SIR format to parallel form. Parity error detection, framing error detection and line break detection is carried out in this block.

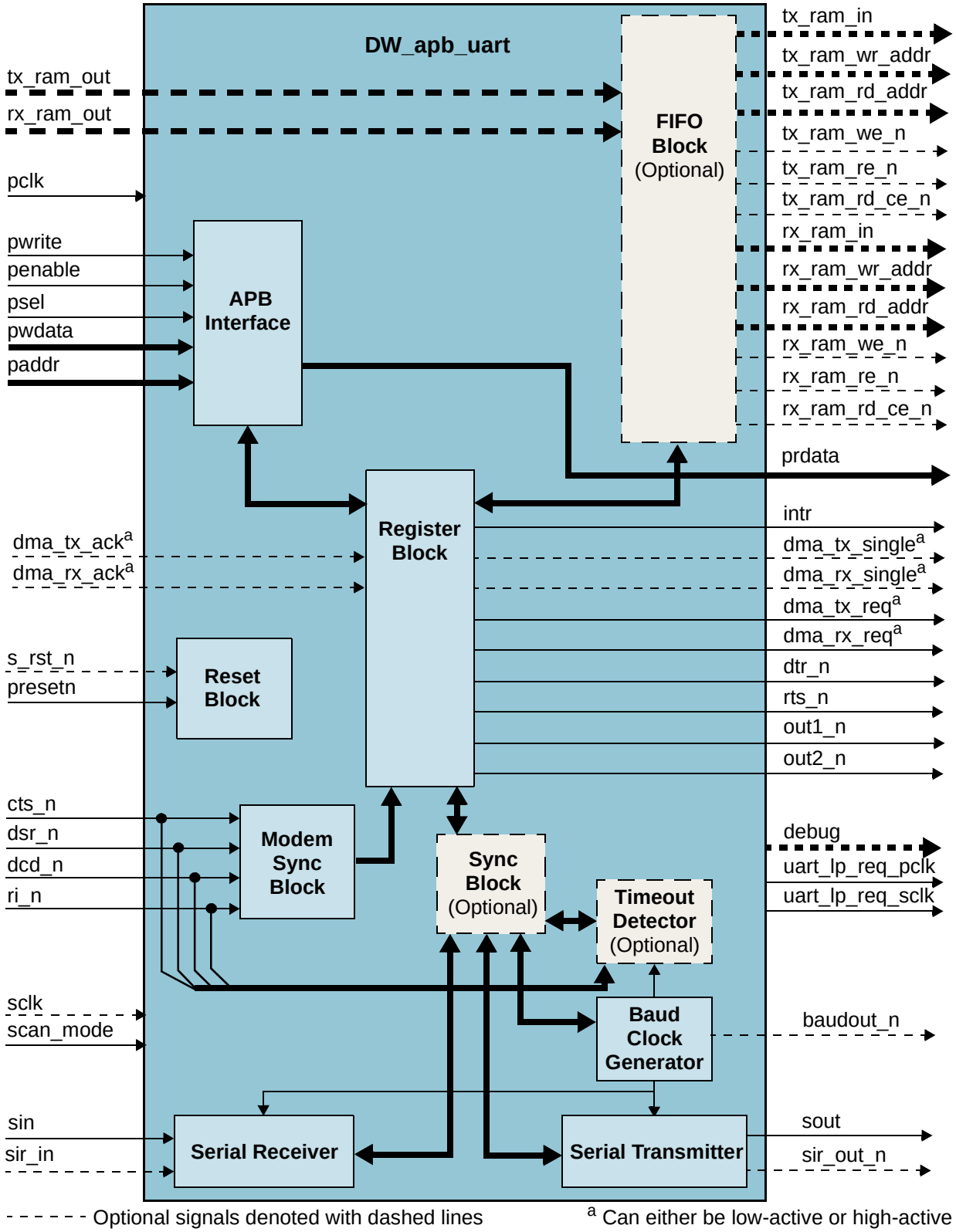


Figure 2: DW_apb_uart Functional Block Diagram

Features

- AMBA APB interface allows easy integration into AMBA SoC implementations
- Configurable parameters for the following:
  - APB data bus widths of 8, 16 and 32
  - Additional DMA interface signals for compatibility with DesignWare DMA interface
  - DMA interface signal polarity
  - Transmit and receive FIFO depths of none, 16, 32, 64,…,2048
  - Internal or external FIFO (RAM) selection
  - Use of two clocks (pclk and sclk) instead of one (pclk)
  - IrDA 1.0 SIR mode support with up to 115.2 Kbaud data rate and a pulse duration (width) as follows: width = 3/16 × *bit period* as specified in the IrDA physical layer specification
  - IrDA 1.0 SIR low-power reception capabilities
  - Baud clock reference output signal
  - Clock gate enable output(s) used to indicate that the TX and RX pipeline is clear (no data) and no activity has occurred for more than one character time, so clocks may be gated
  - FIFO access mode (for FIFO testing) so that the receive FIFO can be written by the master and the transmit FIFO can be read by the master
  - Additional FIFO status registers
  - Shadow registers to reduce software overhead and also include a software programmable reset
  - Auto Flow Control mode as specified in the 16750 standard
  - Loopback mode that enables greater testing of Modem Control and Auto Flow Control features (Loopback support in IrDA SIR mode is available)
  - Transmitter Holding Register Empty (THRE) interrupt mode
- Ability to set some configuration parameters in instantiation
- Configuration identification registers present
- Functionality based on the 16550 industry standard, as follows:
  - Programmable character properties, such as number of data bits per character (5-8), optional parity bit (with odd or even select) and number of stop bits (1, 1.5 or 2)
  - Line break generation and detection
  - DMA signaling with two programmable modes
  - Prioritized interrupt identification
- Programmable FIFO enable/disable
- Programmable serial data baud rate as calculated by the following:
  baud rate = (*serial clock frequency*)/(16×*divisor*)
- External read enable signal for RAM wake-up when using external RAMs
- Modem and status lines are independently controlled
- Complete RTL version
- Separate system resets for each clock domain to prevent metastability

Standards Compliance

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

Note

Information on the DW_apb_uart component in this databook assumes that the reader is fully familiar with the National Semiconductor 16550 (UART) component specification. This specification can be obtained on the web at:
http://www.national.com/pf/PC/PC16550D.html#Datasheet

Information provided on IrDA SIR mode assumes that the reader is fully familiar with the IrDa Serial Infrared Physical Layer Specification. This specification can be obtained from the following website:
http://www.irda.org

Speed and Clock Requirements

This section describes some of the test conditions that have been met to-date by the DW_apb_uart regarding clock speed and baud rates.

The DW_apb_uart has been synthesized and simulated with a pclk of 166 Mhz at 0.18 microns. It met timing requirements at these speeds. The sclk signal was set to 25 MHz with a baud devisor of 1 to give a max baud rate of just over 1.5 M. This is the baud rate referred to in the National 16550 specification.

Verification Environment Overview

The DW_apb_uart is put through a verification process which utilizes constrained randomized testing (or CRT). This process is divided into several “groups” – for testing of the DW_apb_uart’s hardware associated with the transmit, receive, loopback and debug. Under normal verification runs, the test group selected is randomly chosen for a given DW_apb_uart hardware configuration, although some amount of user-controlled selection is possible.

Under each group of tests, two more levels of randomization of the test stimulus are applied – one at the higher “system” level associated with nature of the test chosen, and one at the “parametric” level associated with the DW_apb_uart’s registers. In doing so, control and/or intervention of/in the verification process and scope by the user is reduced to a minimum.

The “system” level of randomization ensures that the DW_apb_uart is, for example, injected with a varying number of characters of arbitrary contents, as well as the type and number of character corruptions applied.

The “parametric” level of randomization applied to the DW_apb_uart ensures that the DW_apb_uart’s hardware is programmed as arbitrarily as possible; for example, the line settings for the characters exchanged during simulations, the varying patterns for the interrupt enables, as well as the various transmit/receive trigger thresholds.

Once the required set of randomized “system” and “parametric” variables are obtained, three separate groups of testcode are kicked off concurrently – one for the generating of the stimulus for the DW_apb_uart and supporting models; one for the overall environment support, such as scoreboarding, messaging, signal transition detections, etc.; and lastly, one for the checkers.

To support the serial exchanges of characters, in both the IrDA and normal transfer modes, VERA models in the SIO VIP are used. Two instances of both the SIOTxrx and the SIOMonitor models assist in verifying that the DW_apb_uart’s hardware functionalities.

To support DMA-controlled transfers to and from the DW_apb_uart, an instance of a AHB DMA BFM is also included. This acts as an independent AHB master issuing AHB transfer commands separately from the AHB master model used to control the DW_apb_uart.

Licenses

Before you begin using the DW_apb_uart, you must have a valid license. For more information, refer to “Licenses” in the *DesignWare AMBA Synthesizable Components Installation Guide.*

Where To Go From Here

At this point, you may want to get started working with the DW_apb_uart 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*.

While coreConsultant is the basic tool used to create a *workspace* for a single component, coreAssembler enables you to work with a component within the context of a subsystem. (A workspace is your working version of a DesignWare AMBA Synthesizable IP component.)

Additionally, the DesignWare Connect tool is designed around coreAssembler, but is customized specifically for DesignWare AMBA-based subsystems. Connect also provides additional subsystem simulation functionality that enhances coreAssembler.

The following table provides common activities and the recommended tool for either single or multiple components.

Table 2: Tool Comparison

Activity	Recommended Tool
Single Component	
Configuration	coreConsultant
Synthesis	coreConsultant
Verification	coreConsultant
Multiple Components	
Configuration	coreAssembler or Connect
Synthesis	coreAssembler or Connect

Table 2: Tool Comparison (Continued)

Activity	Recommended Tool
Formal verification	coreAssembler or Connect
Creation of top-level subsystem RTL	coreAssembler or Connect
Address map creation	Connect
Subsystem simulation	Connect
Creation of subsystem templates	coreAssembler
Importation of non-DesignWare IP	coreAssembler

For more information about implementing your DW_apb_uart component within a DesignWare AMBA subsystem using DesignWare Connect, refer to Chapter 2, “Building and Verifying a Subsystem” on page 21.

For more information about configuring, synthesizing, and verifying just your DW_apb_uart component, refer to Appendix A, “Building and Verifying Your DW_apb_uart” on page 153.

2 Building and Verifying a Subsystem

This chapter documents the step-by-step process you use to connect, configure, synthesize, and verify a DW_apb_uart component within a simple DesignWare AMBA subsystem using the DesignWare Connect tool. You use Connect to create a workspace, which is your working version of a DesignWare AMBA Synthesizable IP (SIP) subsystem. You can create several workspaces to experiment with different design alternatives.

Connect uses coreAssembler as the base tool, but it adds subsystem simulation to the standard coreAssembler functionalities. Complete information about the latest version of Connect is available on the web in the *DesignWare Connect User Guide*. To view documentation specific to your version of Connect, choose the **Help** pull-down menu in the Connect GUI.

For detailed information about coreAssembler, refer to the *coreAssembler User Guide*.

If you want to build and verify only one component, coreConsultant is most likely the best tool for you to use. For specific information about using coreConsultant to configure, synthesize, and verify your DW_apb_uart component, refer to Appendix A on page 153.

The topics in this chapter are as follows:

Setting up Your Environment

DW_apb_uart is included with a DesignWare Synthesizable Components for AMBA 2 release; it is assumed that you have already downloaded and installed the release. However, to download and install the latest versions of required tools, refer to the *DesignWare AMBA Synthesizable Components Installation Guide*.

You also need to set up your environment correctly using specific environment variables, such as DESIGNWARE_HOME, VERA_HOME, PATH, and SYNOPSYS. If you are not familiar with these requirements and the necessary licenses, refer to "Setting up Your Environment" in the *DesignWare AMBA Synthesizable Components Installation Guide.*

Overview of the Configuration and Integration Process

Once you have correctly downloaded and installed a release of DesignWare AMBA synthesizable components and then set up your environment, you can begin building your DesignWare AMBA subsystem with Connect.

Figure 3 illustrates Connect's usage flow from invoking the tool to creating a workspace to stepping through the activities in the GUI. Table 3 on page 23 provides a description of the workspace directory and subdirectories.

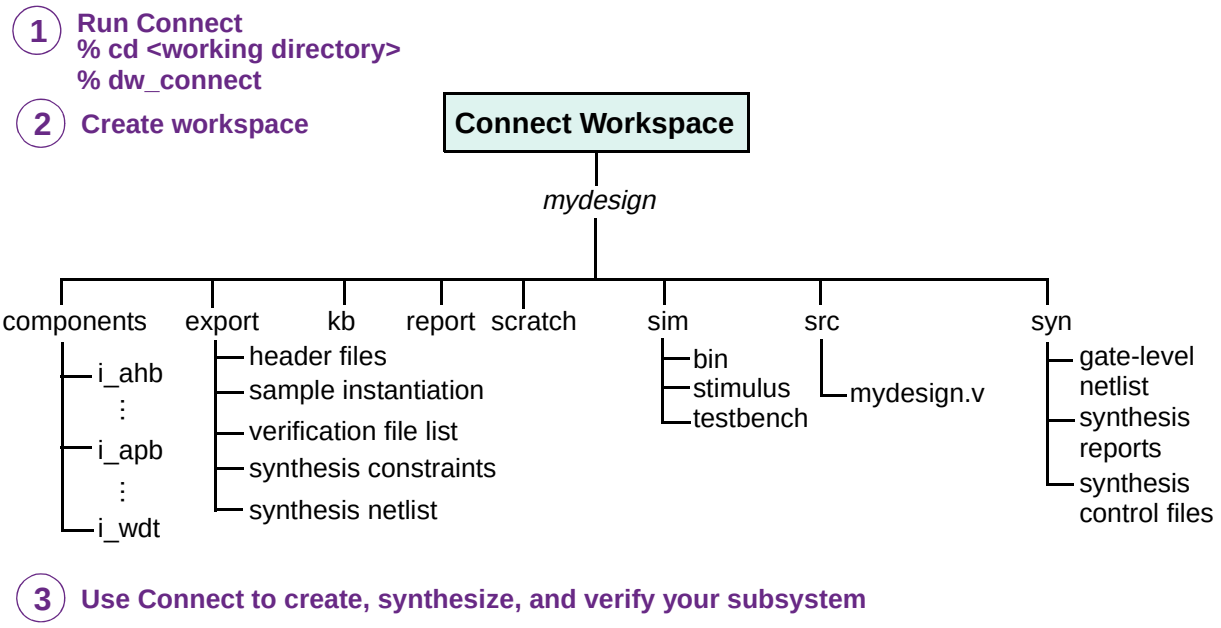


Figure 3: Connect Usage Flow

Table 3: Connect Workspace Directory Contents

Directory/Subdirectory	Description
Directories containing files to be used after exiting Connect.	
export	Contains the files you will need once you exit Connect. These files will be used to integrate the results from the completed source configuration and synthesis activities into your larger system (outside Connect). An index.html file in this directory describes all of the exported files. For more details about the files in this directory, refer to "Export Directory" in the *DesignWare Connect User Guide*.
sim/stimulus/*component_name*	Contains the test stimulus files in Verilog and C.
sim/testbench/all	Includes the subsystem's testbench file, *design_name*_tb.v. subsystem source file list, design_name.lst, and simulation execution script `run.scr`.
src	Includes the subsystem top-level RTL file, *design_name*.v.
Directories containing files not generally used after exiting Connect.	
components	Includes a directory for each DW AMBA Synthesizable IP instance connected in the subsystem.
components/*instance_name*	Contains the data for each IP component instance. This is the instance name of the component used in the design. Each *instance_name* directory is equivalent to a coreConsultant component workspace. See the IP component's databook for details of this directory structure.
kb	Contains knowledge base information used by Connect. These are binary files containing the state of the design.
report	Contains all of the reports created by Connect during build, configuration, test and synthesis phases. An index.html file in this directory links to many of these generated reports.
scratch	Contains temp files used during the Connect processes.
sim	Includes simulation files for the subsystem. This directory is created when you complete the **Simulate Subsystem** activity in uart.
sim/testbench/all/cov_results	Includes the various coverage results of the verified subsystem
syn	Contains synthesis files for the subsystem. This directory is created when you complete all of the activities in the **Create Gate-Level Netlist** (synthesis) activity group in Connect.

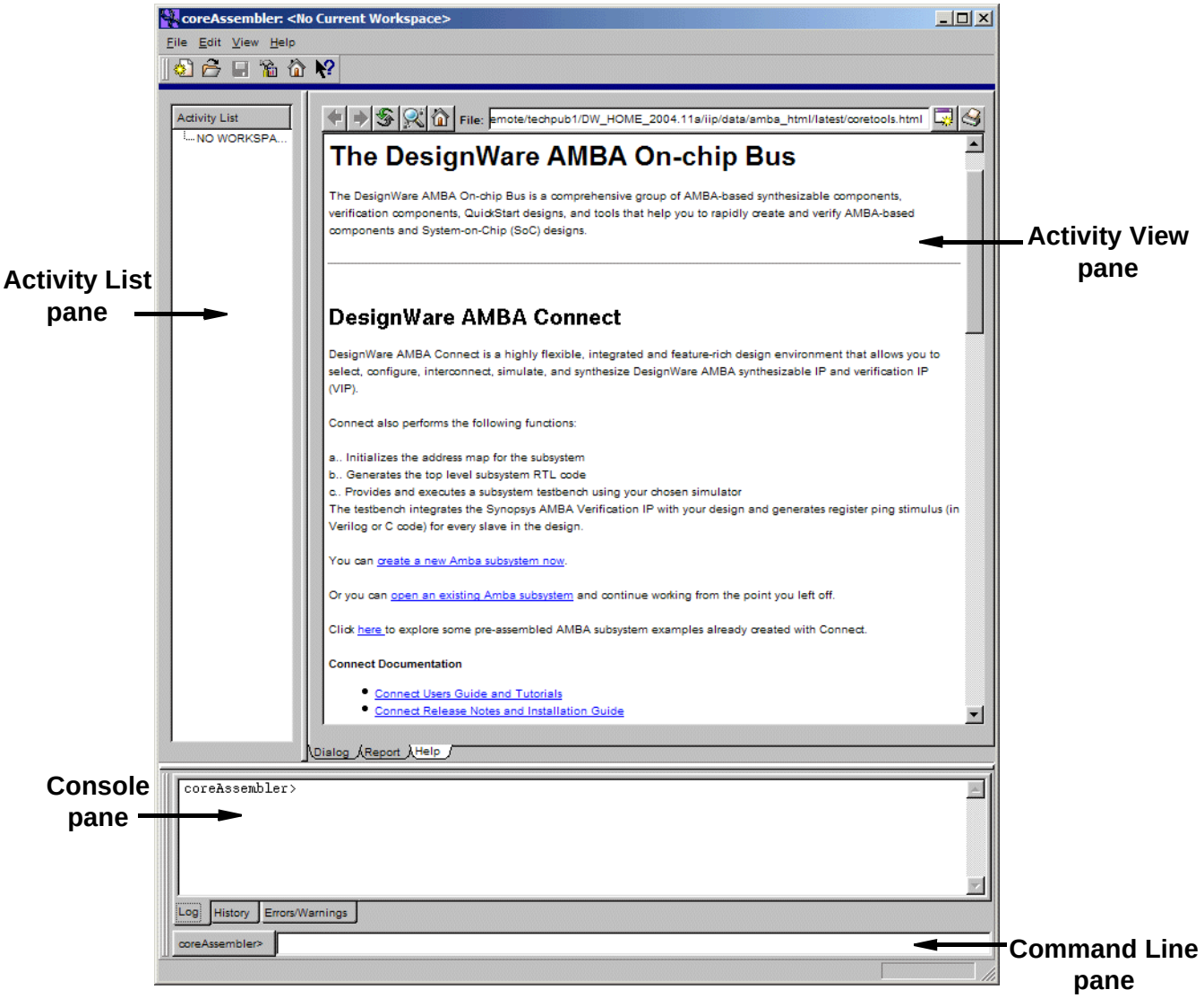
Start Connect

To invoke Connect:

1. In a UNIX shell, navigate to a directory where you plan to locate your component workspace.
2. Invoke the Connect tool:

```
% dw_connect
```

The welcome page is displayed, similar to the one below.



3. Click on “create a new AMBA subsystem now” link to create a new workspace. After you have created a workspace, you can also continue working from the point you left off by using the “open an existing AMBA subsystem” link to open it back up.

   A “Create a New Workspace” message appears, which explains some of the terms used by coreAssembler. Read this information and then click OK.

4. In the resulting dialog box, specify the workspace name, workspace root directory, and design name, or leave the defaults. To find out more about the fields in this dialog box, you can right-click over the specific item to get What's This help.

   The following describes these items in more detail:

   - **Workspace name** - the name of the Unix directory where the database containing all of your design files will be kept.
   - **Workspace root directory** - the name of the Unix directory that is the "parent" to your workspace directory (Workspace name).
   - **Design name** - the top-level design name that is used in the top-level RTL file.

   Connect displays an HTML file that explains general design rules and address map specification rules for Connect subsystems. Familiarize yourself with the information in this file so that you will be familiar with the automatically-generated testbench that will result from this session. If you later want to access this information again, use the **Help > Connect Design Rules** menu. When you have finished, click OK.

   At this point, Connect creates in the workspace an export directory that will eventually contain the files you need once you exit Connect. You can use these files for your own chip-level synthesis and simulation. A README file and an index.html file in this directory both describe all of the exported files in this directory.

   In the Connect GUI, you will see that the DW_ahb component is already displayed in the schematic window and that the Add Subsystem Components activity is highlighted under the Create RTL category in the Activity List on the left.

   For more information about Connect, refer to the *DesignWare Connect User Guide*. For more in-depth Connect tutorials, refer to the "Connect Tutorials" chapter of the *DesignWare Connect User Guide*. For tables that list the contents of the export directory at each step of the Connect process, refer to "Export Directory" in the *DesignWare Connect User Guide*.

Check Your Environment

Before you begin configuring your component, it is recommended that you check your environment to ensure you have the latest tool versions installed and your environment variables set up correctly.

To check your environment, use the **Help > Check Tool Environment** menu path.

An HTML report is displayed in a separate dialog. This report lists the specific tools and versions installed in your environment. It also displays errors when a specific tool is not installed or if you are using an older version than you need.

Note

You do not have to be concerned if this tool reports errors regarding simulators that are not used. Only concern yourself if you receive an error regarding your simulator of choice. For more information about setting the appropriate environment variables for your simulator, refer to "Setting up Your Environment" in the *DesignWare AMBA Synthesizable Components Installation Guide.*

You will also see an error if your $DESIGNWARE_HOME environment variable has not been set up correctly. When you are finished, click OK.

Add DW_apb_uart to the Subsystem

In a minimal subsystem using the DW_apb_uart component, you would also have an AHB bus, an APB bus, and most likely a “dummy” AHB master. Therefore, the subsystem described in this chapter contains the following components: DW_apb_uart, DW_ahb, DW_apb, and AHB Master. The last component is one that you will export up and out of the design to be replaced by your real AHB Master, such as a CPU, which you would probably add in your own environment later in the design process. At least one exported AHB master interface is required in the subsystem if you intend to do a basic “ping test” simulation.

Figure 4 illustrates the DW_apb_uart in a simple subsystem.

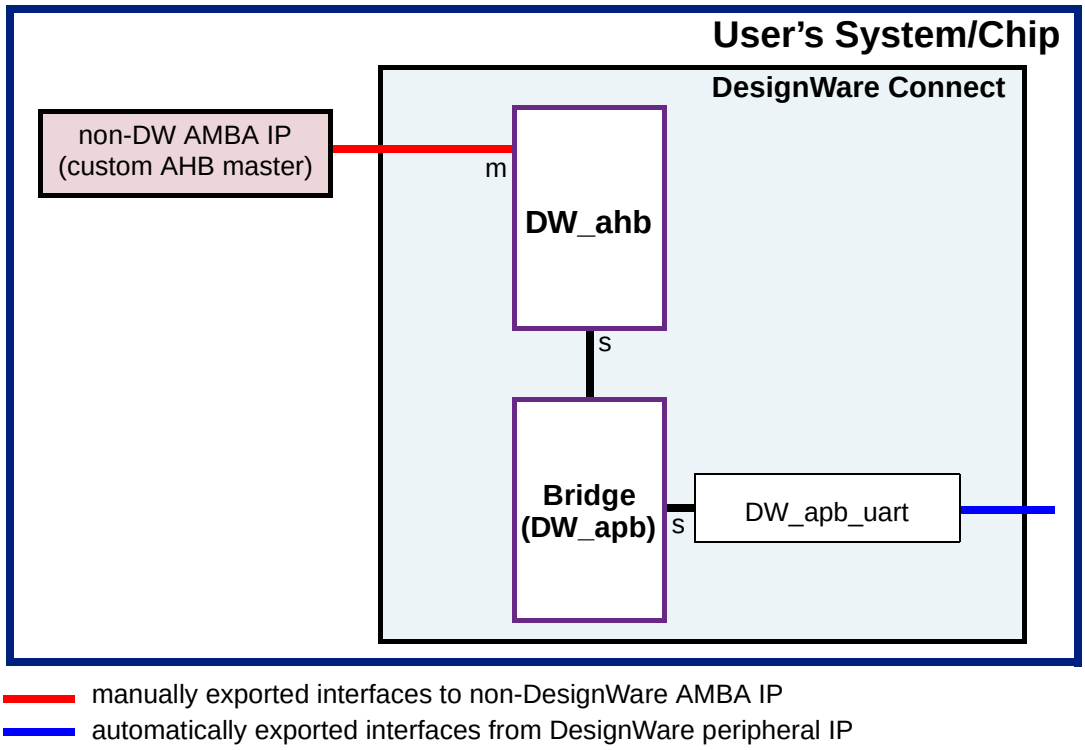


Figure 4: DW_apb_uart in Simple Subsystem

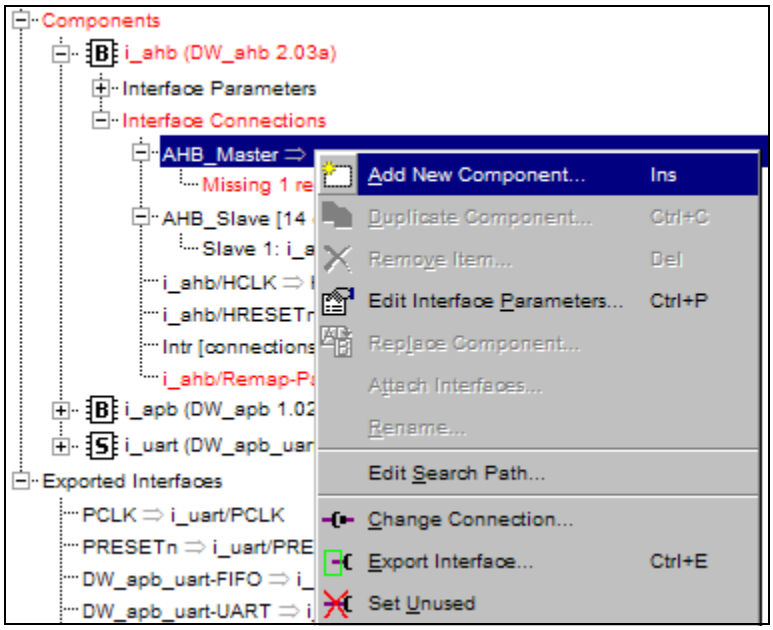
The following procedure steps you through the process of creating a simple subsystem with a DW_apb_uart component.

1. Use the **Schematic > Add New Component** menu item to display the Add Component Instance to Subsystem dialog; alternatively, you can right-click in the schematic window and choose **Add New Component** from the popup menu or use the Insert key.
2. Choose the DW_apb component and click Apply. You will notice that the hresetn and hclk inputs are automatically connected together, and that the AHB_Slave1 output of the DW_ahb is connected to the AHB_Slave input of the DW_apb.
3. Now add the DW_apb_uart component using the Add Component Instance to Subsystem dialog. Click OK. Notice that Connect automatically connects the presetn and pclk signals, and connects the APB_Slave0 output of the DW_apb component to the APB_Slave input of the DW_apb_uart.

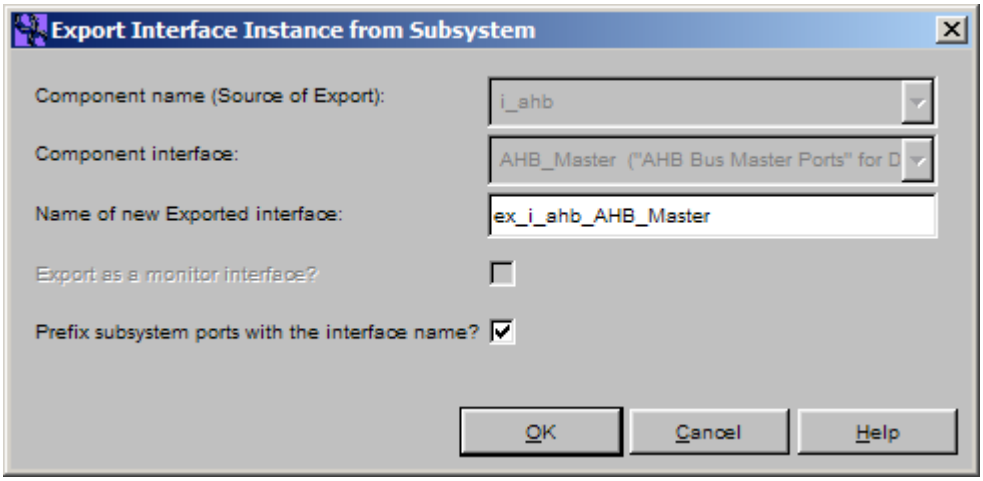
4. Notice that the DW_ahb instance is red in the schematic view. Toggle over to the tree view by clicking the toggle icon on the toolbar and expand the i_ahb component instance. The AHB Master line in the Interface Connections says that it is missing a connection, and the i_ahb/Remap-Pause line shows it as disconnected.

   To resolve this, you are going to export an AHB master interface from the DW_ahb.

5. To export an AHB master interface, select the AHB Master line in the tree view, right-click, and then select **Export Interface** as illustrated in the following figure.

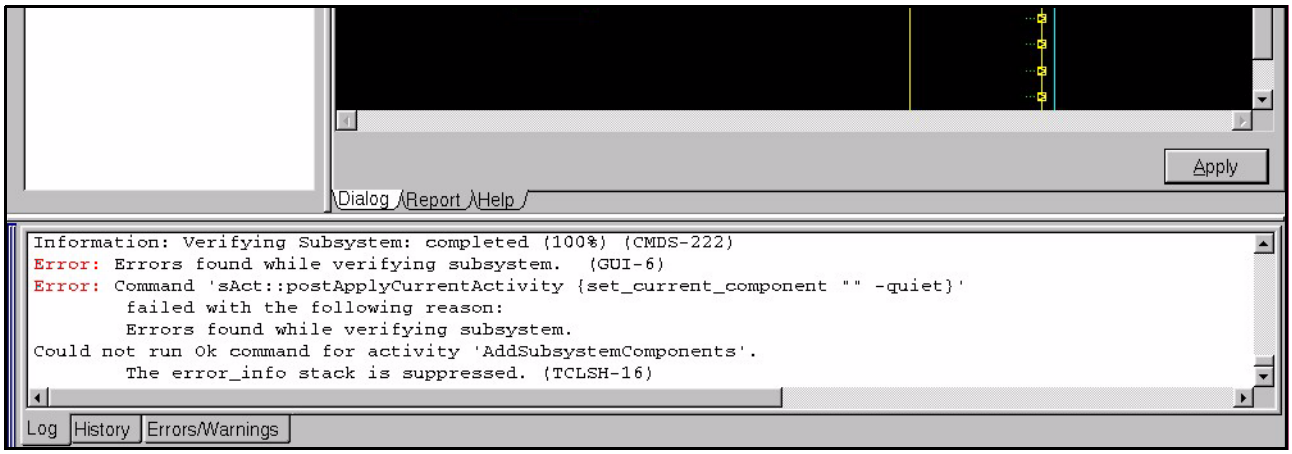


   The “Export Interface Instance from Subsystem” dialog opens. For this exercise, keep the default naming and click OK.



6. You now have a rudimentary subsystem that includes the DW_apb_uart component. Next *try* to complete the Add Subsystem Components activity by clicking the Apply button in the lower right corner below the schematic. Alternatively, you can just click on the next activity (Configure Components), and answer “yes” to the pop-up window.

An error message appears telling you that there is a problem because the remap/pause interface in the DW_ahb is not connected. Notice that the DW_ahb component is still red, indicating that there is some kind of problem. The Console pane at the bottom of the GUI gives you additional information about the error, as illustrated in the following figure.



Note

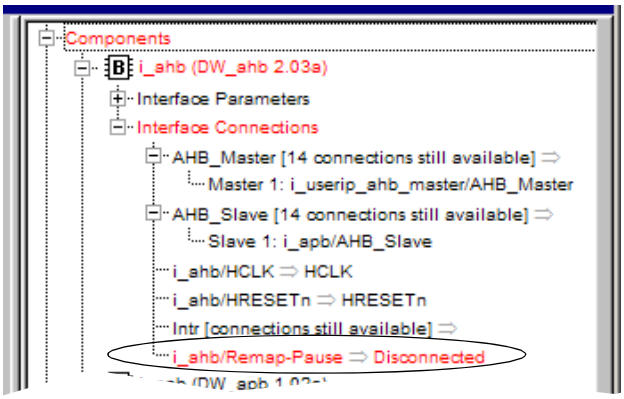
These error messages are generated from TCL code and may seem verbose. In most cases, the first part of the error message contains useful information, whereas the remainder of the message can be ignored. If you want to obtain more information about a particular error, you can issue the following command in the Command Line below the Console pane:

```
% man error number
```

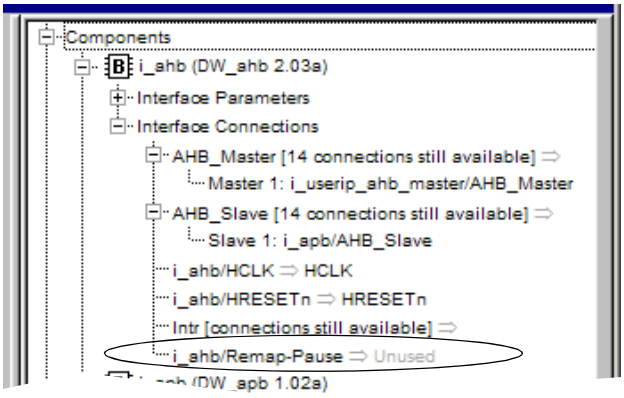
7. Because you do not need the remap/pause feature in this subsystem, you will set that interface as "unused." OK the error message and right-click on the i_ahb/Remap-Pause interface and choose the **Set Unused** menu item. Notice that the DW_ahb is no longer red.

   You can see in the following illustration the difference between how the tree view displays an error and how it looks when the error is resolved.

Before



After



8. Click the Apply button again to complete the Add System Components activity. When a message box asks you if you want to initialize the subsystem address map, click Yes. Automatic address map creation is discussed in more detail in the next section “Configure DW_apb_uart” on page 29.

   Connect creates the files described below in the export directory for this activity.

New Contents of Export Directory after Add Subsystem Components	
Directory or File	**Description**
batch.tcl	Batch script for recreating completed activities associated with subsystem assembly. This file gets updated after the following activities are completed: ● Add Subsystem Components ● Complete Connections ● Simulate Subsystem ● Create Gate-Level Netlist
index.html	HTML file containing descriptions of files created in export directory after the Add Subsystem Components step.
README	Text file containing descriptions of files created in the export directory after the Add Subsystem Components step.

9. Connect displays a report for the subsystem, which includes a number of hyperlinks to sections further down in the page for unconnected interfaces, subsystem components, exported interface connections, component interface connections, and subsystem ports to be created. You should always familiarize yourself with the information in all reports before going to any new activity.

Configure DW_apb_uart

This section steps you through the tasks that configure the component-level parameters (configuration parameters) for DW_apb_uart in Connect. For this exercise, you will not configure any of the other components in this example subsystem, but instead leave them with their default parameter values.

If you need help with any field in the Activity List pane, right-click on the field name and then left-click on the What’s This box to get specific information for that item. Additionally, you can click on the Help tab (lower-left corner of the Activity View pane) for each activity to activate the coreAssembler online help.

1. **Configure Components** – The Configure Components activity is where you specify the basic configuration of the DW_apb_uart; click on that item in the Activity List.
2. Click the DW_apb_uart item (also called i_uart) to display the Top Level Parameters window; look through the defaults.
3. Because you clicked “Yes” when the dialog asked if you wanted to initialize the subsystem address map at the completion of the Add Subsystems Components activity, look at the results.
   a. Click on the DW_apb (i_apb) item, and then click on the “Address Map” item.
   b. Notice that the APB start address is 0x00010000 and that the end address is 0x000103ff, which is the same as the start and end addresses of the DW_apb_uart, listed as Slave 0. If you had connected one or more APB slaves to the DW_apb component, then the start and end

address of the DW_apb would have reflected the start address of Slave 0 and the end address of the last slave. Similarly, you can view the automatically generated address map in the DW_ahb component.

4. Click the Apply button to activate the configuration parameters. Connect creates no files in the export directory for this activity.

   When the configuration setup is complete, the Report tab is displayed, which gives you a list of configuration reports for all the components in the subsystem. At minimum, click on the link to the configuration report for DW_apb_uart. Look at any source files to which you have access (in encrypted format if you have a DesignWare license, and unencrypted if you have a source license) and look at all the parameters that have been set for this particular configuration.

Complete Signal Connections

You can use the Complete Connections activity to connect any pins that were not automatically connected as part of an interface. Unconnected input pins can be connected to unconnected output pins, tied off to a constant value, or exported from the subsystem (that is, connected to an automatically created input port of the subsystem). Unconnected output pins can be connected to an existing input pin, explicitly marked as unconnected (open), or exported from the subsystem.

In this exercise, you will leave everything in its default situation. If you want to learn more about completing signal connections, refer to the "Complete Connections" section in the *coreAssembler User Guide*, which you can access through the **Help > coreAssembler Tool Help > User's Guide** menu item. For now, do the following:

1. **Complete Connections** – Click on Complete Connections in the Activity List.
2. Look at the Manual Connect and Manual Disconnect tabs; leave the defaults and Apply the dialog.
3. Look through the Export Connections Out of Subsystem dialog and then click OK, leaving the defaults.

   Notice that there are hyperlinks to information regarding automatic connections (in a separate HTML file) and sections further down in the file for other connections and unconnected subsystem ports and component pins. Connect adds no new files to the export directory for this activity but only updates the batch.tcl file.

New Contents of Export Directory after Complete Connections	
Directory or File	**Description**
batch.tcl	Updated to include all activities completed to this point. You can use this script to recreate the entire workspace up to this point in the Activity List.

Generate Subsystem RTL

You can create top-level code for the subsystem in either VHDL or Verilog using the Generate Subsystem RTL task in the Activity List. In the dialog that appears in the Activity View pane, you choose the output language.

1. **Generate Subsystem RTL** – Click on Generate Subsystem RTL in the Activity List.

2. If you are using a Verilog simulator (such as VCS), choose the default Verilog language and Apply the activity.
3. If you are using a VHDL simulator, click the button for VHDL as the output language and then choose between std_logic or std_ulogic.

Note

This dialog only selects the HDL language for the top-level RTL for the subsystem. All RTL written for the instantiated components in the subsystem are controlled by the individual IP provider. For all DesignWare Synthesizable Components for AMBA 2, the component RTL is written in Verilog.

4. Click Apply. Regardless of whether you use a Verilog or VHDL simulator, Connect creates both Verilog and VHDL files in *workspace*/src and *workspace*/export directories. If you choose a Verilog simulator, the VHDL files will default to std_logic. Connect creates the following files in the export directory for this activity.

New Contents of Export Directory after Generate Subsystem RTL	
Directory or File	**Description**
batch.tcl	No updates.
workspace.lst	List of source files in proper analysis order for entire subsystem.
*workspace*_comp.vhd	VHDL component declaration for subsystem.
*workspace*_inst.v	Verilog Testbench template; example subsystem instantiation.
*workspace*_inst.vhd	VHDL Testbench template; example subsystem instantiation.
*workspace*_params.h	C subsystem configuration information.
*workspace*_params.v	Verilog subsystem configuration information.
*workspace*_params.vhd	VHDL subsystem configuration information.

5. Familiarize yourself with the generated RTL files.

Create Gate-Level Netlist

To run synthesis on the subsystem and create a gate-level netlist, step through the following tasks in the Connect GUI. You need to click the check box next to each activity in order to access the specific activity dialog. At any time, you can click on the Help tab for each activity to display more information.

1. Look at the tool installation root directories in the Tool Installation Roots dialog, which is accessed from the toolbar menu through **Edit > Tool Installation Roots**, or by using the Tools button on the toolbar. You can type values directly in the data fields, or use the buttons to locate the correct directories. The tool choices are:
   - Design Compiler (dc_shell) – Specifies the location for the root directory of the Design Compiler installation, if different from the default location.
   - Physical Compiler (psyn_shell) – Enables the Physical Compiler if you plan to use an incremental physical synthesis strategy or if you plan to do RTL to place gates.
   - Primetime (pt_shell) – Enables Primetime if you plan to implement budgeting or generate timing models.
   - Formality (fm_shell) – Enables Formality if you plan to formally verify the synthesized gate-level implementation of the core.
   - DC FPGA (fpga_shell) – Enables Design Compiler FPGA if your synthesis targets high-end FPGA devices.

   At a minimum for this exercise, dc_shell must have defined installation directories, and in order to complete the optional formal verification in this chapter, you will also need fm_shell.
2. **Specify Target Technology** – Connect analyzes the target technology library and uses it to generate a synthesis strategy that is optimized for your technology library. In the Design Compiler window, a target and link library must be specified; otherwise, errors occur in Connect.

   Under the Specify Target Technology category in the Activity List, the title in the tabs depends on the compiler you chose in the previous step. Regardless, this screen provides fields for you to enter the search path for the specific compiler, as well as target and link library paths. If necessary, specify the search path for the tool you specified in the previous screen. Also, specify the path to the target and link libraries. Click Apply and familiarize yourself with the resultant report, which gives you the technology information.
3. **Specify Clock(s)** – In the Specify Clock(s) activity, look at the attributes associated with each of the real and virtual clocks in your design. Click Apply and familiarize yourself with the resultant report, which gives you clock information.
4. **Specify Operating Conditions and Wire Loads** – In the Specify Operating Conditions and Wire Loads activity, look at the attributes relating to the chip environment. If you do not see a value beside OperatingConditionsWorst, select an appropriate value from the drop-down list; if there is no value for this attribute, you will get an error message. Click Apply and look at the report, which gives the operating conditions and wireload information.
5. **Specify Port Constraints** – In the Specify Port Constraints activity, look at the attributes associated with input delay, drive strength, DRC constraints, output delay, and load specifications. Click Apply and look at the report, which gives the port constraint checks.
6. **Specify Synthesis Methodology** – In the Specify Synthesis Methodology activity, look at the synthesis strategy attributes. Note that these attributes are typically set by the core developer and are not required to be modified by the core integrator. If you want to add your own commands

during a synthesis, you use the Advanced tab in order to provide path names to your auxiliary scripts. Also, click on the Physical Synthesis tab to familiarize yourself with those options. Click Apply and look at the report, which gives design information. For more information on adding auxiliary scripts, refer to “Advanced Synthesis Methodology Attributes” in the *coreAssembler User Guide*.

7. **Specify Test Methodology** – In the Specify Test Methodology activity, look at the scan test attributes. Also click on the other tabs to familiarize yourself with auto-fix attributes, SoC test wrapper attributes, test wrapper integration attributes, BIST attributes, and BIST testpoint insertion attributes. Click Apply and look at the report, which gives design-for-test information.
8. **Synthesize** – Choose the Synthesize activity. Do the following:
   a. Choose the Strategy tab.
   b. Click the Options button beside DCTCL_opto_strategy and look through the strategy parameters. For example, you can use the Gate Clocks During Elaboration check box in the Clock Gating tab in order to add parameters that enable and control the use of clock gating. Click OK when you are done. For more information on clock gating and other parameters for synthesis strategies, refer to “DC(TCL)_opto_strategy” in the *coreAssembler User Guide*.

      For FPGA synthesis, click the Options button and then select the FPGA Synthesis tab. It is here where you specify the location of your FPGA device and speed grade, synthetic libraries other than DesignWare Foundation libraries, implementation of DC-FPGA operators, and so on. For more information about running synthesis for an FPGA device, refer to the *coreAssembler User Guide.*

      For Design for Test, click the Options button and then select the Design for Test tab. Here you can specify whether to add the -scan option to the initial compile call (Test Read Compile) and/or insert design for test circuitry (Insert Dft). For more information about include DFT in your synthesis run, refer to the *coreAssembler User Guide.*
   c. Choose the Options tab. Look at the values for the parameters listed below.

Field Name	Description
Execution Options	
Generate Scripts only?	**Values**: Enable or Disable **Default Value**: Disable **Description**: Writes the run.scr script, but it is not run when you click Apply. To run the script, go to the component workspace and run the script.
Run Style	**Values**: local, lsf, grd, or remote **Default Value**: local **Description**: Describes how to run the command: locally, through LSF, through GRD, or through the remote shell.
Run Style Options	**Values**: user-defined **Default Value**: none **Description**: Additional options for the run style options except local. For remote, specify the hostname. For LSF and GRD, specify bsub or qsub commands.

Field Name	Description
Parallel job CPU limit	**Values**: user-defined; minimum value is 1 **Default Value**: 1 **Description**: Specifies number of parallel compile jobs that can be run.
Send e-mail	**Values**: current user's name **Description**: E-mail is sent when the command script completes or is terminated.
Skip reading $HOME/.synopsys_dc.setup	**Values**: Enable or Disable **Default Value**: Disable **Description**: Forces tools not to read .synopsys_dc.setup file from $HOME.

d. If it is not already set, choose the "local" Run Style option and keep the other default settings.

e. Look through the Licenses and Reports tabs, and ensure that you have all the licenses that are required to run this synthesis session.

f. Click Apply in the Synthesize Activity pane to start synthesis from Connect. The current status of the synthesis run is displayed in the main window. Click the Reload Page button if you want to update the status in this screen.

9. **Generate Test Vectors** – This option allows you to generate ATPG test vectors with TetraMax. For more information about this, refer to "Generating Test Vectors" in the *coreAssembler User Guide*.

Checking Synthesis Status and Results

To check synthesis status and results, click the Report tab for the synthesis options; Connect displays a dialog that indicates:

- Your selected Run Style (local, lsf, grd, or remote)
- The full path to the HTML file that contains your synthesis results
- The name of the host on which the synthesis is running
- The process ID (Job Id) of the synthesis
- The status of the synthesis job (running or done)

The Results dialog also enables you to kill the synthesis (Kill Job) and to refresh the status display in the Results dialog (Refresh Status). The Results information includes:

- Summary of log files
- Synthesis stages that completed
- Summary of stage results

This information indicates whether the synthesis executed successfully, and lists the transactions that occurred during the scenario(s). Thorough analysis of the scenario execution requires detailed analysis of all synthesis log files and inspection of report summaries.

Synthesis Output Files

All the synthesis results and log files are created under the syn directory in your workspace. Two of the files in the *workspace*/syn directory are:

- run.scr – Top-level synthesis script for the subsystem
- run.log – Synthesis log file

Your final netlist and report directories depend on the QoR effort that you chose for your synthesis (default is medium):

- low – initial
- medium – incr1
- high – incr2

For more information about deliverables that are generated after synthesis is performed, refer to "Database Description" on page 167.

Running Synthesis from Command Line

To run synthesis from the command line prompt for the files generated by Connect, enter the following command:

```
% run.scr
```

This script resides in your *workspace*/syn directory.

Create Component GTECH Simulation Model

DesignWare AMBA Synthesizable IP components are delivered in encrypted format, rather than source code, and every simulator besides VCS cannot read the encrypted source files. In order for these simulators to read the encrypted files, you must either perform a GTECH conversion or purchase a source license from Synopsys.

Note

The Synopsys VCS simulator reads the encrypted files directly and does not require a GTECH conversion. All other supported simulators require a GTECH simulation model. You need DesignWare and Design Compiler licenses to complete the GTECH generation process. If you are a source license customer, then you do not have to generate a GTECH simulation model, even if you are using a non-VCS simulator.

Also, it is not possible to perform a GTECH simulation with DC FPGA.

1. **Create Component GTECH Simulation Model** – To create a GTECH simulation model for the DW_apb_uart component, click on the Generate GTECH Model (for i_uart) activity.

2. Look at the values for the parameters listed below.

Field Name	Description
Execution Options	
Generate Scripts only?	**Values**: Enable or Disable **Default Value**: Disable **Description**: Writes scripts that run the generation of the GTECH simulation model, but they are not run when you click Apply. To run these scripts, go to the gtech directory of the component workspace and run the run.scr script.
Run Style	**Values**: local, lsf, grd, or remote **Default Value**: local **Description**: Describes how to run the command: locally, through lsf, through grd, or through the remote shell.
Run Style Options	**Values**: user-defined **Default Value**: none **Description**: Additional options for run style options other than local. For remote, specify the hostname. For lsf and grd, specify bsub or qsub commands.
Send e-mail	**Values**: current user's name **Description**: E-mail is sent when the command script completes or is terminated.
Synthesis Control	
Ungroup Netlist after Compile	**Values**: Enable or Disable **Default Value**: Disable **Description**: Ungroups the design to provide a non-hierarchical netlist.

Note

For GTECH Simulations Only. Due to the configurable nature of the component, some ports in the testbench may not be needed for your chosen configuration. Warnings about undriven nets may appear. These warnings are to be expected, and you can ignore them. The verification result files show if the verification ran successfully.

3. Click Apply. Connect invokes Design Compiler to perform a low-effort compile (quickmap) of your custom configuration using the Synopsys technology-independent GTECH library. After this activity has completed, an e-mail similar to the following is sent to the specified user name (if you enabled that option):

```
Activity:   GenerateGtechModel
Workspace:  workspace_path
Design:     design_name
Started:    Wed Jul 24 16:19:48 BST 2002
Finished:   Wed Jul 24 16:21:42 BST 2002
Status:     Completed
Results:    workspace_path/components/i_uart/gtech/gtech.log
```

Your simulation model is contained in the DW_apb_uart.v output file that is written to *workspace*/components/i_uart/gtech/qmap/db.

Verify Component

The Verify Component activity in Connect allows you to perform verification for an individual component. For this exercise, you are just going to perform verification for the DW_apb_uart; however, you typically would also perform verification for other components in your subsystem.

To verify DW_apb_uart, use Connect to complete the following steps:

1. **Verify Component** – To run verification for the DW_apb_uart component, click Setup and Run Simulations (for i_uart) in the Verify Component activity.
2. Specify the various options for the Simulator.
   a. In the Select Simulator area, click on the Simulator list item to view available simulators (VCS is the default).
   b. Specify an appropriate Verilog simulator from the drop-down menu.
   c. For installation instructions and information about required tools and versions, refer to "Setting up Your Environment" in the *DesignWare AMBA Synthesizable Components Installation Guide*. For general information about the contents of the release, refer to the *DesignWare DW_apb_uart Release Notes*.
   d. In the Simulator Setup area of the Simulator pane, look at the parameters for the simulator setup, as detailed in the following table.

Field Name	Description
Root Directory of Cadence Installation	The path to the top of the directory tree where the Cadence NC-Verilog executable is found; coreConsultant automatically detects this path. The NC-Verilog executables reside in the ./bin subdirectory.
MTI Include Directory	The path to the include directory contained within your MTI simulator installation area. A valid directory includes the veriuser.h file.
Vera Install Area ($VERA_HOME)	Path to your Vera installation. This parameter defaults to the value of your VERA_HOME environment variable. Changes to this value are propagated as $VERA_HOME in any simulation run.
Vera .vro file cache directory	Cache directory used by Vera to store .vro files, which are generated when building the testbench. Encrypted Vera source is compiled and stored in the cache.
DW Foundation install area ($SYNOPSYS)	Path to your $SYNOPSYS/dw installation. This parameter defaults to the value of your SYNOPSYS environment variable. Any change to this value must be made from the Tool Installation Areas coreConsultant dialog box.
C Compiler for (Vera PLI)	**Values**: gcc or cc **Default Value**: gcc **Description**: Invokes the specific C compiler to create a Vera PLI for your chosen non-VCS simulator. Choose cc if you have the platform-native ANSI C compiler installed. Choose gcc if you have the GNU C compiler installed.

e. In the Waves Setup area of the Simulator pane, look at the parameters for the waves setup, as detailed below.

Note

For the Generate Waves File setting, enable the check box so that the simulation creates a dump file that you can use later for debugging the simulation, if you want to do so.

Field Name	Description
Generate waves file	**Values**: Enable or Disabled **Default Value**: Disabled **Description**: Indicates whether a wave file should be created for debugging with a wave file browser after simulation ends. Uses VPD file format for VCS, and VCD format for the other supported simulators.
Depth of waves to be recorded	**Description**: Enter the depth of the signal hierarchy for which to record waves in the dump file. A depth of 0 indicates all signals in the hierarchy are included in the wave file.

3. Choose the View list choice.

   In the View Selection area of the View pane, look at the choice of views of the design you can simulate from the drop-down list:

   - RTL – requires a source license or Synopsys VCS
   - GTECH – requires that you have completed the Generate GTECH Model activity (refer to page 35) only if you are using a non-VCS simulator and do not have a source license.

4. Choose the Execution Options list choice to set the following options:

Field Name	Description
Do Not Launch Simulation	**Values**: Enable or Disable **Default Value**: Disable **Description**: Determines whether to execute the simulation or just generate the simulation run script. If checked, coreConsultant generates, but does not execute, the simulation run script. You can execute the script at a later time by directly invoking the run script (*workspace*/sim/run.scr) from the UNIX command line or by repeating the Verification activity with the Do Not Launch Simulation option unselected.
Run Style	**Values**: local, lsf, grd, or remote **Default Value**: local **Description**: Describes how to run the command: locally, through lsf, through grd, or through the remote shell.
Run Style Options	**Values**: user-defined **Default Value**: none **Description**: Additional options for run style other than local. For remote, specify the hostname. For lsf and grd, specify bsub or qsub commands.
Send e-mail	**Values**: current user's name **Description**: E-mail is sent to the specified user when the command script completes or is terminated.

5. Select Testbench and look at the options described below:

Field Name	Description
Run test_uart	**Values**: Enable or Disable **Default Value**: Enable **Description**: Tests functionalities of DW_apb_uart.

6. Click Apply to run the simulation.

   When you click Apply, Connect performs the following actions:

   - Sets up the DW_apb_uart verification environment to match your selected DW_apb_uart configuration.
   - Generates the simulation run script (run.scr) and writes it to your *workspace*/components/i_uart/sim directory.
   - Invokes the simulation run script, unless you enabled the Do Not Launch Simulation option.

   The simulation run script, in turn, performs the following actions:

   - Links the generated command files, and recompiles the testbench.
   - Invokes your simulator to simulate the specified scenarios.
   - Writes the simulation output files to your *workspace*/components/i_uart/sim/test_* directory.
   - If an e-mail address is specified, sends the simulation completion information to that e-mail address when the simulation is complete.

   For an overview of the related tests, refer to Chapter 8, “Verification” on page 133.

Checking Simulation Status and Results

To check simulation status and results, click the Report tab for either the GTECH models or for the simulation options; Connect displays a dialog that indicates:

- Your selected Run Style (local, lsf, grd, or remote)
- The full path to the HTML file that contains your simulation results
- The name of the host on which the simulation is running
- The process ID (Job Id) of the simulation
- The status of the simulation job (running or done)

If you selected the “LSF/GRD” option for the Run Style, then the status of the simulation jobs (running or complete) is incorrect. Once all the simulation jobs are submitted to the LSF/GRD queue, the status would indicate “complete.” You should use “bjobs/qstatus” to see whether all the jobs are completed.

The Results dialog also enables you to kill the simulation (Kill Job) and to refresh the status display in the Results dialog (Refresh Status). The Results information includes:

- Vera compile execution messages
- Simulation execution messages
- DW_apb_uart bus transactions

This information indicates whether the simulation executed successfully, and lists the DW_apb_uart transactions that occurred during the scenario(s).

Thorough analysis of the scenario execution requires detailed analysis of all simulation output files and inspection of simulation waveforms with a waveform viewer.

Applying Default Verification Attributes

To reset all DW_apb_uart verification attributes to their default values, use the Default button in the Setup and Run Simulation activity under the Verification tab.

To examine default attribute values without resetting the attribute values in your current workspace, create a new workspace; the new workspace has all the default attribute values. Alternatively, use the Default button to reset the values, and then close your current workspace without saving it.

Verify the Subsystem

To verify the subsystem, use Connect to complete the following activities.

Formal Verification

You can run formal verification scripts using Synopsys Formality (fm_shell) to check two designs for functional equivalence. You can check the gate-level design from a selected phase of a previously executed synthesis strategy against either the RTL version of the design or the gate-level design from another stage of synthesis.

To run formal verification scripts:

- **Formal Verification** – Choose Formal Verification under the Verify Component category and then click Apply.

Simulate Subsystem

Specify the simulation for the subsystem by completing the Simulate Subsystem activity:

1. **Simulate Subsystem** – In the Simulator Setup tab, look at the parameters for the simulator setup, as detailed in the following table.

Field Name	Description
Control Language	**Values**: Verilog or C **Default Value**: Verilog **Description**: The language used to control the testbench.
AHB/APB Monitors Enabled	**Values**: Enable or Disable **Default Value**: Enable **Description**: Determines whether or not to activate the AHB/APB bus monitors.
SIO Monitors Enabled	**Values**: Enable or Disable **Default Value**: Enable **Description**: Determines whether or not to activate the SIO monitors.
Simulator	**Values**: Enable or Disable **Default Value**: Enable **Description**: Choice of simulator to invoke for the testbench.
MTI Include Path	The path to the include directory contained within your ModelSim simulator installation area. A valid directory includes the veriuser.h file.
Root Directory of Cadence Installation	The path to the top of the directory tree where the Cadence NC-Verilog executable is found; Connect automatically detects this path. The NC-Verilog executables reside in the ./bin subdirectory.

Field Name	Description
Generate waves file	**Values**: Enable or Disable **Default Value**: Disable **Description**: Indicates whether a wave file should be created for debugging with a wave file browser after simulation ends. Uses VPD file format for VCS, and VCD format for the other supported simulators.
C Compiler for (Vera PLI)	**Values**: gcc or cc **Default Value**: gcc **Description**: Invokes the specific C compiler to create a Vera PLI for your chosen non-VCS simulator. Choose cc if you have the platform-native ANSI C compiler installed. Choose gcc if you have the GNU C compiler installed.
Path for gcc	**Description**: Path to gcc for compiling System C drivers when using C control for the testbench.

2. In the Execution Options tab, complete the settings below.

Field Name	Description
Execution Options	
Generate Scripts only?	**Values**: Enable or Disable **Default Value**: Disable **Description**: Writes scripts that run the generation of the GTECH simulation model, but they are not run when you click Apply. To run these scripts, go to the gtech directory of the component workspace and run the run.scr script.
Run Style	**Values**: local, lsf, grd, or remote **Default Value**: local **Description**: Describes how to run the command: locally, through lsf, through grd, or through the remote shell.
Run Style Options	**Values**: user-defined **Default Value**: none **Description**: Additional options for run style options other than local. For remote, specify the hostname. For lsf and grd, specify bsub or qsub commands.
Send e-mail	**Values**: current user's name **Description**: E-mail is sent when the command script completes or is terminated.

3. Choose the Testbench Definition tab to determine which slaves you want to be tested by which master – in this case, there is only one slave and one master.

4. Click Apply to run the subsystem simulation. Connect creates no files in the export directory for this activity.

New Contents of Export Directory after Verify Subsystem	
Directory or File	**Description**
No files added.	

Checking Subsystem Verification Status and Results

To check subsystem simulation status and results, click the Report tab. As for the component simulation, Connect displays a dialog that indicates:

- Your selected Run Style (local, lsf, grd, or remote)
- The full path to the HTML file that contains your simulation results
- The name of the host on which the simulation is running
- The process ID (Job Id) of the simulation
- The status of the simulation job (running or done)

The Results information includes:

- How many tests passed out of selected tests
- Link to testbench
- Waveforms
- Coverage results
- Testbench topology
- Connect design rules
- Log data
- Slave test status

Create a Batch Script

It is recommended that you create a batch file that contains information about the workspace, parameters, attributes, and so on.

1. To do this, choose the **File > Write Batch Script** menu item and enter a location (other than your working directory or where your workspace resides) and name for the file. Use the browse button to navigate to the directory where you want this file to reside.
2. Then look at the contents to familiarize yourself with the information that you can get from this file. You can use the batch script to reproduce the workspace.

Note

When you use this file, it deletes your workspace before it recreates it. So all manually edited files will become deleted. Make sure to save any files you want to keep to a different location.

To use this batch script to recreate your subsystem, perform the following:

1. Make sure to run the batch.tcl script from a directory other than where your workspace resides.
2. In the Console at the bottom of the Connect GUI screen, enter the following:

```
% source batch.tcl
```

Export the Subsystem

You can export your subsystem for reuse by third parties by building a subsystem coreKit, or you can create a subsystem template that exports your subsystem as a reconfigurable “box.” You need a separate coreBuilder license for both of these activities. However, the scope of this tutorial does not include exporting a coreKit. If you are interested in learning more about this, refer to the *coreAssembler User Guide*.

3 Functional Description

This chapter describes the functional operation of the DW_apb_uart. The topics are as follows:

UART (RS232) Serial Protocol

Because the serial communication between the DW_apb_uart and the selected device is asynchronous, additional bits (start and stop) are added to the serial data to indicate the beginning and end. Utilizing these bits allows two devices to be synchronized. This structure of serial data accompanied by start and stop bits is referred to as a character, as shown in Figure 5.

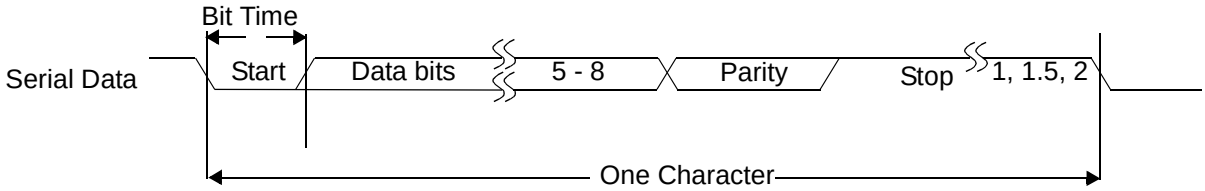


Figure 5: Serial Data Format

An additional parity bit may be added to the serial character. This bit appears after the last data bit and before the stop bit(s) in the character structure to provide the DW_apb_uart with the ability to perform simple error checking on the received data.

The DW_apb_uart Line Control Register ("LCR" on page 100) is used to control the serial character characteristics. The individual bits of the data word are sent after the start bit, starting with the least-significant bit (LSB). These are followed by the optional parity bit, followed by the stop bit(s), which can be 1, 1.5 or 2.

All the bits in the transmission (with exception to the half stop bit when 1.5 stop bits are used) are transmitted for exactly the same time duration. This is referred to as a Bit Period or Bit Time. One Bit Time equals 16 baud clocks. To ensure stability on the line the receiver samples the serial input data at approximately the mid point of the Bit Time once the start bit has been detected. As the exact number of baud clocks that each bit was transmitted for is known, calculating the mid point for sampling is not difficult, that is every 16 baud clocks after the mid point sample of the start bit. Figure 6 shows the sampling points of the first couple of bits in a serial character.

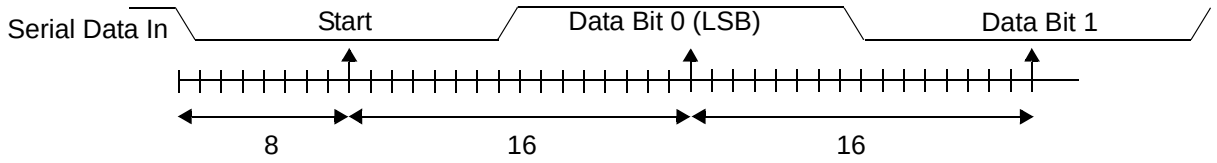


Figure 6: Receiver Serial Data Sample Points

As part of the 16550 standard an optional baud clock reference output signal (baudout_n) is supplied to provide timing information to receiving devices that require it. The baud rate of the DW_apb_uart is controlled by the serial clock (sclk or pclk in a single clock implementation) and the Divisor Latch Register (DLH and DLL). Figure 7 shows the timing diagram for the baudout_n output for different divisor values.

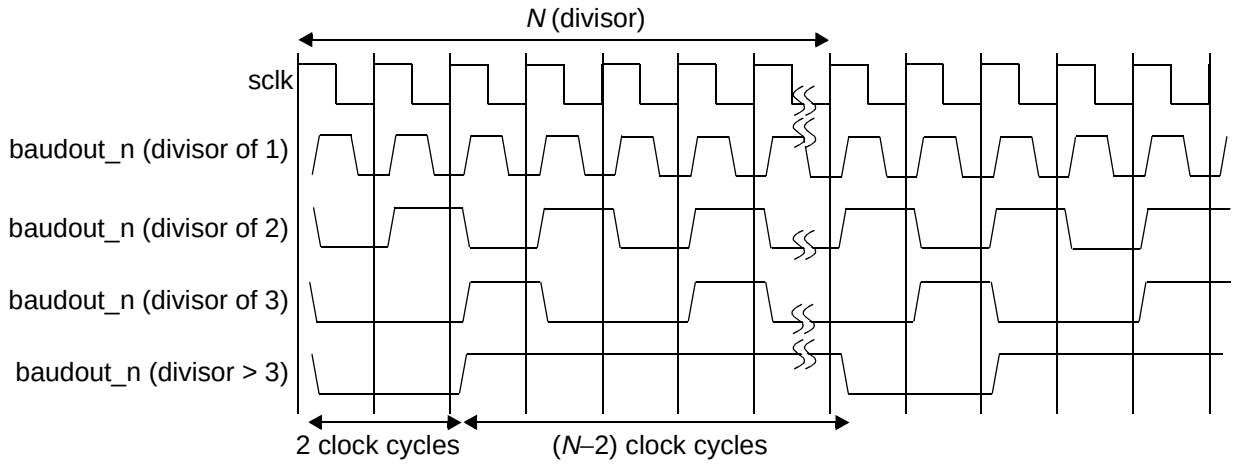


Figure 7: Baud Clock Reference Timing Diagram

IrDA 1.0 SIR Protocol

The Infrared Data Association (IrDA) 1.0 Serial Infrared (SIR) mode supports bi-directional data communications with remote devices using infrared radiation as the transmission medium. IrDA 1.0 SIR mode specifies a maximum baud rate of 115.2 Kbaud.

Attention

Information provided on IrDA SIR mode in this section assumes that the reader is fully familiar with the IrDa Serial Infrared Physical Layer Specifications. This specification can be obtained from the following website:
http://www.irda.org

The data format is similar to the standard serial (sout and sin) data format. Each data character is sent serially, beginning with a start bit, followed by 8 data bits, and ending with at least one stop bit. Thus, the number of data bits that can be sent is fixed. No parity information can be supplied and only one stop bit is used while in this mode.

Trying to adjust the number of data bits sent or enable parity with the Line Control Register (LCR) has no effect. When the DW_apb_uart is configured to support IrDA 1.0 SIR it can be enabled with Mode Control Register (MCR) bit 6. When the DW_apb_uart is not configured to support IrDA SIR mode, none of the logic is implemented and the mode cannot be activated, reducing total gate counts. When SIR mode is enabled and active, serial data is transmitted and received on the sir_out_n and sir_in ports, respectively.

Transmitting a single infrared pulse signals a logic zero, while a logic one is represented by not sending a pulse. The width of each pulse is 3/16ths of a normal serial bit time. Thus, each new character begins with an infrared pulse for the start bit. However, received data is inverted from transmitted data due to the infrared pulses energizing the photo transistor base of the IrDA receiver, pulling its output low. This inverted transistor output is then fed to the DW_apb_uart sir_in port, which then has correct UART polarity. Figure 8 shows the timing diagram for the IrDA SIR data format in comparison to the standard serial format.

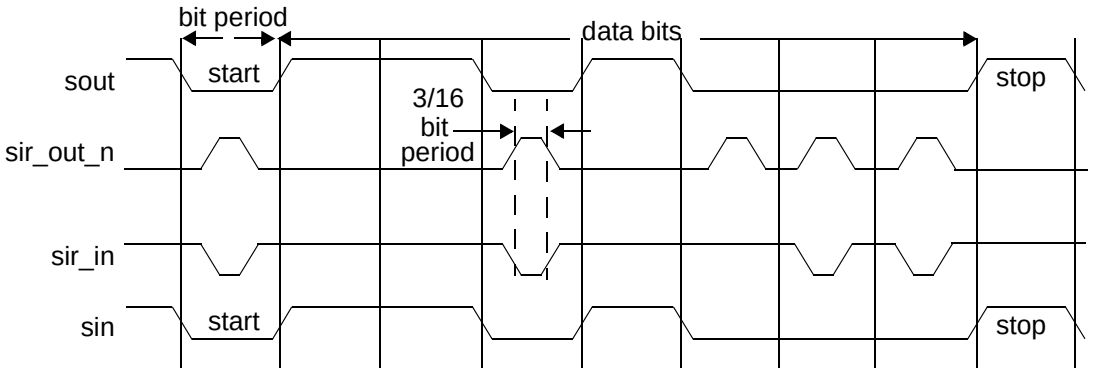


Figure 8: IrDA SIR Data Format

As detailed in the IrDA 1.0 SIR, the DW_apb_uart can be configured to support a low-power reception mode. When the DW_apb_uart is configured in this mode, the reception of SIR pulses of 1.41 microseconds (minimum pulse duration) is supported, as well as nominal 3/16 of a normal serial bit time. Using this low-power reception mode requires the programming of the Low Power Divisor Latch (LPDLL/LPDLH) registers.

When IrDA SIR mode is enabled, the DW_apb_uart operation is similar to when the mode is disabled, with one exception; data transfers can only occur in half-duplex fashion when IrDA SIR mode is enabled. This is because the IrDA SIR physical layer specifies a minimum of 10ms delay between transmission and reception. This 10ms delay must be generated by software.

FIFO Support

The DW_apb_uart can be configured to implement FIFOs (as shown in Figure 2 on page 15) to buffer transmit and receive data. If FIFO support is not selected, then no FIFOs are implemented and only a single receive data byte and transmit data byte can be stored at a time in the RBR and THR. This implies a 16450-compatible mode of operation. In this mode most of the enhanced features are unavailable.

In FIFO mode, the FIFOs can be selected to be either external customer-supplied FIFO RAMs or internal DesignWare D-flip-flop based RAMs (DW_ram_r_w_s_dff). If the configured FIFO depth is greater than 256, the FIFO memory selection is restricted to be external. In addition, selection of internal memory restricts the Memory Read Port Type to D-flip-flop based, Synchronous read port RAMs.

When external RAM support is chosen, either synchronous or asynchronous RAMs can be used. Asynchronous RAM provides read data during the clock cycle that has the memory address and read signals active, for sampling on the next rising clock edge. Synchronous single stage RAM registers the data at the current address out and is not available until the next clock cycle (second rising clock edge). Figure 9 shows the timing diagram for both asynchronous and synchronous RAMs.

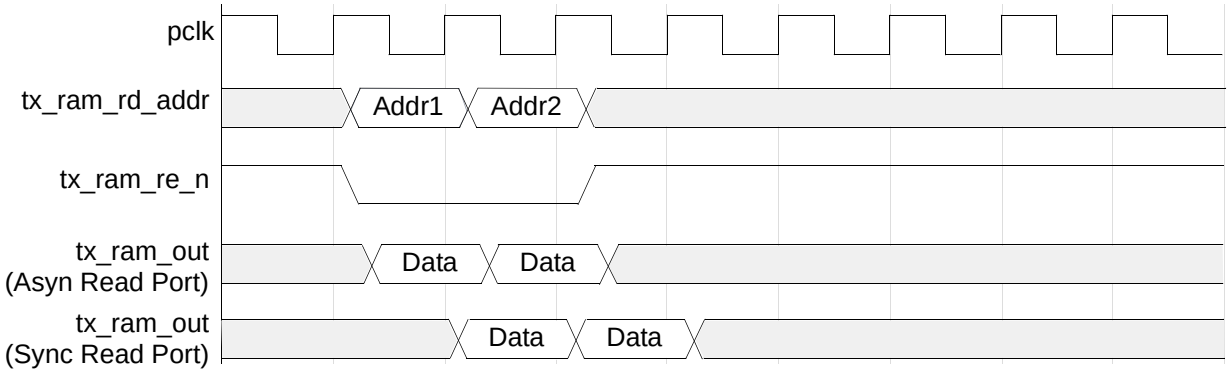


Figure 9: Timing for RAM Reads

Similarly, you can use synchronous RAM for writes, which registers the data at the current address out. Figure 10 shows the timing diagram for RAM writes.

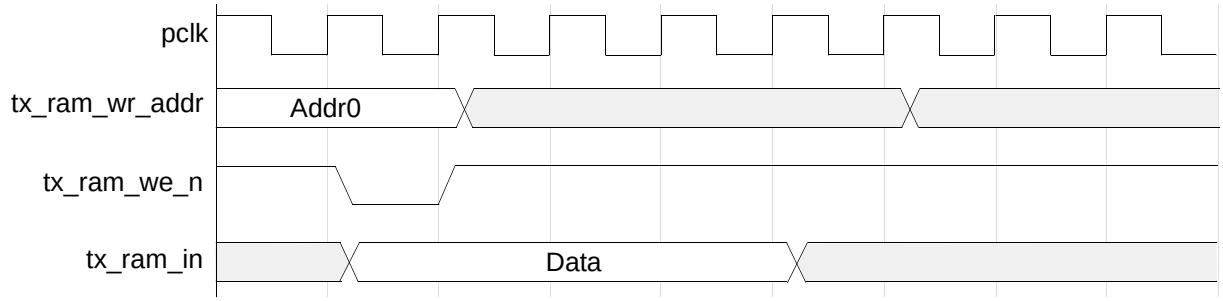


Figure 10: Timing for RAM Writes

When FIFO support is selected, an optional programmable FIFO Access mode is available for test purposes, which allows the receive FIFO to be written by the master and the transmit FIFO to be read by the master. When FIFO Access mode is not selected, none of the corresponding logic is implemented and the mode cannot be enabled, reducing overall gate counts. When FIFO Access mode has been selected it can be enabled with the FIFO Access Register (FAR[0]). Once enabled, the control portions of the transmit and receive FIFOs are reset and the FIFOs are treated as empty.

Data can be written to the transmit FIFO as normal; however no serial transmission occurs in this mode (normal operation halted) and hence no data leave the FIFO. The data that has been written to the transmit FIFO can be read back with the Transmit FIFO Read (TFR) register, which when read gives the current data at the top of the transmit FIFO.

Similarly, data can be read from the receive FIFO as normal. Since the normal operation of the DW_apb_uart is halted in this mode, data must be written to the receive FIFO so it may be read back. Data is written to the receive FIFO with the Receive FIFO Write (RFW) register. The upper two bits of the 10 bit register are used to write framing error and parity error detection information to the receive FIFO. Setting bit RFW[9] to indicate a framing error and RFW[8] to indicate a parity error. Although these bit can not be read back via the Receive Buffer Register they can be checked by reading the Line Status Register and checking the corresponding bits when the data in question is at the top of the receive FIFO.

Clock Support

The DW_apb_uart can be configured to have either one system clock (pclk) or two system clocks (pclk and sclk). Having the second asynchronous serial clock (sclk) implemented accommodates accurate serial baud rate settings, as well as APB bus interface requirements. When using a single system clock, the system clock settings available for accurate baud rates are greatly restricted.

When a two clock design is chosen, a synchronization module is implemented (as shown in Figure 2 on page 15) for synchronization of all control and data across the two system clock boundaries. The RTL diagram for the data synchronization module is shown in Figure 11 on page 50. The data synchronization module can have pending data capability. The timing diagram shown in Figure 12 on page 50 shows this process.

The arrival of new source domain data is indicated by the assertion of start. Since data is now available for synchronization the process is started and busy status is set. If start is asserted while busy and pending data capability has been selected, the new data is stored. When no longer busy the synchronization process starts on the stored pending data. Otherwise the busy status is removed when

the current data has been synchronized to the destination domain and the process continues. If only one clock is implemented, all synchronization logic is absent and signals are simply passed through this module.

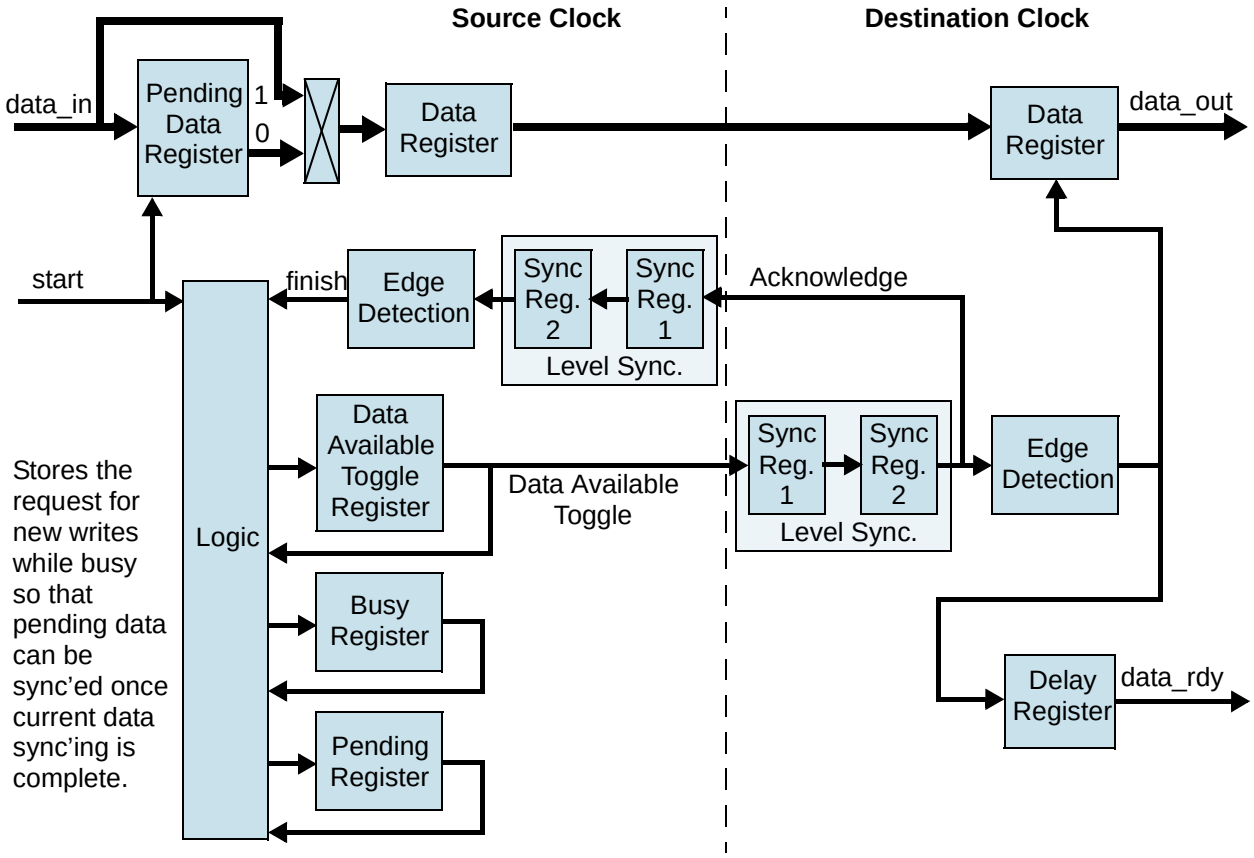


Figure 11: RTL Diagram of Data Synchronization Module

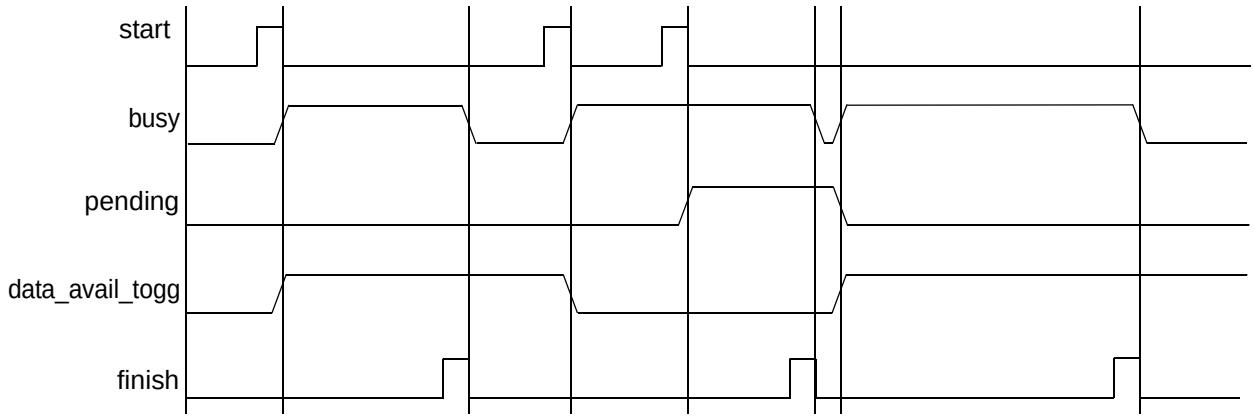


Figure 12: Timing Diagram for Data Synchronization Module

Full synchronization handshake takes place on all signals that are “data synchronized”. All signals that are “level synchronized” are simply passed through two destination clock registers. Both synchronization types incur additional data path latencies. However, this additional latency has no negative affect on received or transmitted data, other than to limit the serial clock (sclk) to being no faster than four-times the pclk clock for back-to-back serial communications with no idle assertion.

A serial clock faster than four-times the pclk signal does not leave enough time for a complete incoming character to be received and pushed into the receiver FIFO. However, in most cases, the pclk signal is faster than the serial clock and this should never be an issue. There is also slightly more time required after initial serial control register programming, before serial data can be transmitted or received.

The serial clock modules must have time to see new register values and reset their respective state machines. This total time is guaranteed to be no more than eight clock cycles of the slower of the two system clocks. Therefore, no data should be transmitted or received before this maximum time expires, after initial configuration.

In systems where only one clock is implemented, there are no additional latencies.

Interrupts

The assertion of the DW_apb_uart interrupt output signal (intr) occurs whenever one of the several prioritized interrupt types are enabled and active. The following interrupt types can be enabled with the IER register:

- Receiver Error
- Receiver Data Available
- Character Timeout (in FIFO mode only)
- Transmitter Holding Register Empty at/below threshold (in Programmable THRE interrupt mode)
- Modem Status

These interrupt types are covered in more detail in Table 8 on page 97.

When an interrupt occurs the master accesses the IIR. See Table 7 on page 96 to determine the source of the interrupt before dealing with it accordingly.

Auto Flow Control

The DW_apb_uart can be configured to have a 16750-compatible Auto RTS and Auto CTS serial data flow control mode available. If FIFOs are not implemented, then this mode cannot be selected. When Auto Flow Control is not selected, none of the corresponding logic is implemented and the mode cannot be enabled, reducing overall gate counts. When Auto Flow Control mode has been selected it can be enabled with the Modem Control Register (MCR[5]). Figure 13 on page 52 shows a block diagram of the Auto Flow Control functionality.

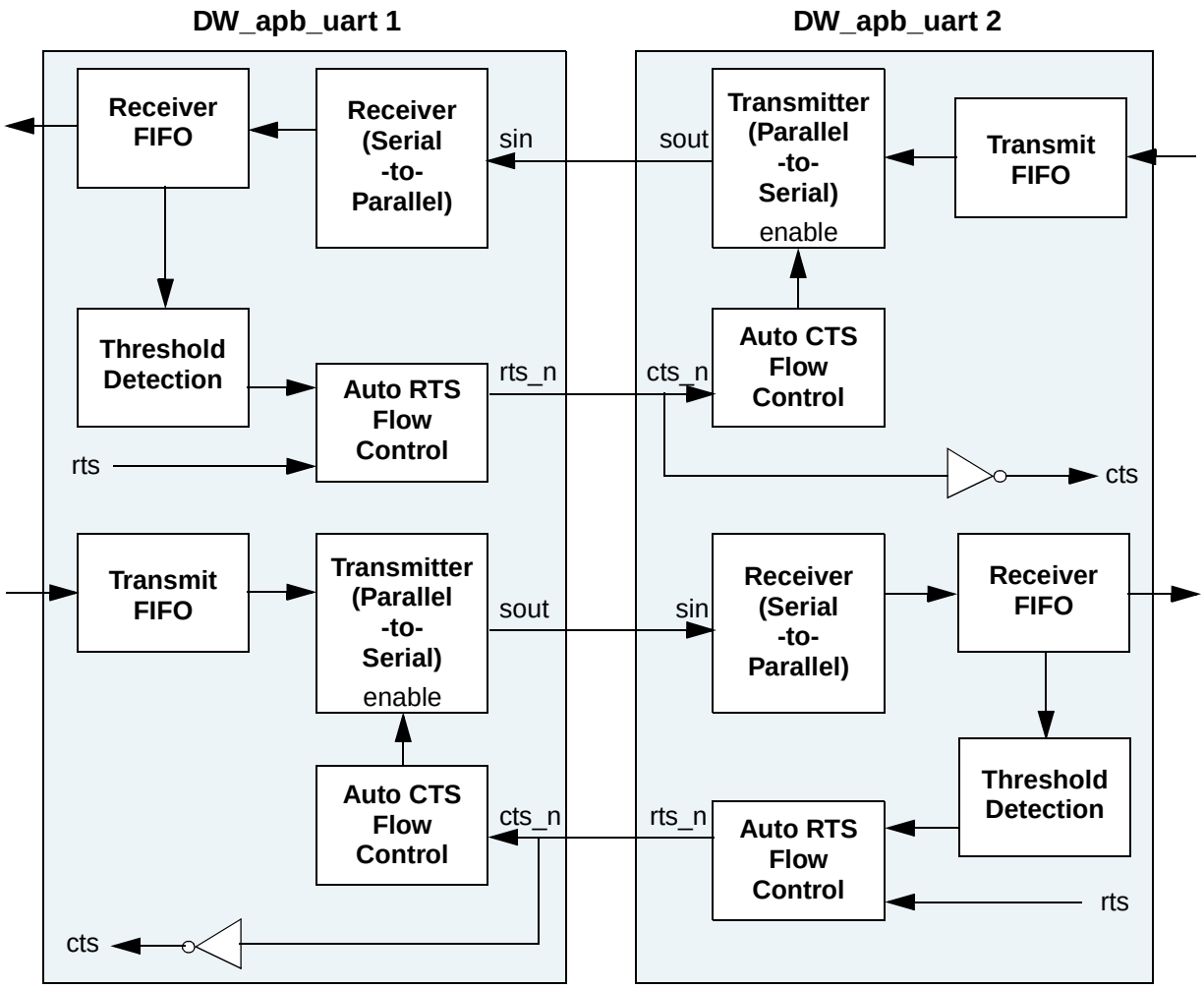


Figure 13: Auto Flow Control Block Diagram

Auto RTS and Auto CTS are described as follows:

Auto RTS – Becomes active when the following occurs:

- Auto Flow Control is selected during configuration
- FIFOs are implemented
- RTS (MCR[1] bit and MCR[5]bit are both set)
- FIFOs are enabled (FCR[0]) bit is set)
- SIR mode is disabled (MCR[6] bit is not set)

When Auto RTS is enabled (active), the rts_n output is forced inactive (high) when the receiver FIFO level reaches the threshold set by FCR[7:6]. When rts_n is connected to the cts_n input of another UART device, the other UART stops sending serial data until the receiver FIFO has available space (until it is completely empty).

The selectable receiver FIFO threshold values are: 1, ¼, ½, and “2 less than full”. Since one additional character may be transmitted to the DW_apb_uart after rts_n has become inactive (due to data already having entered the transmitter block in the other UART), setting the threshold to “2 less than full” allows maximum use of the FIFO with a safety zone of one character.

Once the receiver FIFO becomes completely empty by reading the Receiver Buffer Register (RBR), rts_n again becomes active (low), signalling the other UART to continue sending data.

It is important to note that even if everything else is selected and the correct MCR bits are set, if the FIFOs are disabled through FCR[0] or the UART is in SIR mode (MCR[6] is set to one), Auto Flow Control is also disabled. When Auto RTS is not implemented or disabled, rts_n is controlled solely by MCR[1]. Figure 14 shows a timing diagram of Auto RTS operation.

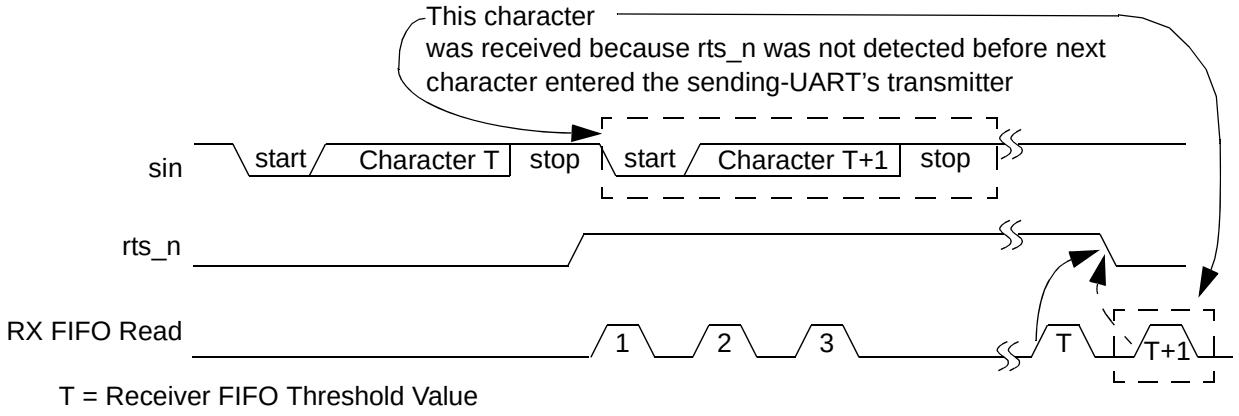


Figure 14: Auto RTS Timing

Auto CTS – becomes active when the following occurs:

- Auto Flow Control is selected during configuration
- FIFOs are implemented
- AFCE (MCR[5] bit is set)
- FIFOs are enabled through FIFO Control Register FCR[0] bit
- SIR mode is disabled (MCR[6] bit is not set)

When Auto CTS is enabled (active), the DW_apb_uart transmitter is disabled whenever the cts_n input becomes inactive (high). This prevents overflowing the FIFO of the receiving UART.

If the cts_n input is not inactivated before the middle of the last stop bit, another character is transmitted before the transmitter is disabled. While the transmitter is disabled, the transmitter FIFO can still be written to, and even overflowed.

Therefore, when using this mode, the following happens:

- The UART status register can be read to check if the transmit FIFO is full (USR[1] set to zero),
- The current FIFO level can be read via the TFL register, or
- The Programmable THRE Interrupt mode must be enabled to access the “FIFO full” status via the Line Status Register (LSR).

When using the “FIFO full” status, software can poll this before each write to the Transmitter FIFO. See “Programmable THRE Interrupt” on page 54 for details. When the cts_n input becomes active (low) again, transmission resumes. It is important to note that even if everything else is selected, if the

FIFOs are disabled via FCR[0], Auto Flow Control is also disabled. When Auto CTS is not implemented or disabled, the transmitter is unaffected by cts_n. A Timing Diagram showing Auto CTS operation can be seen in Figure 15.

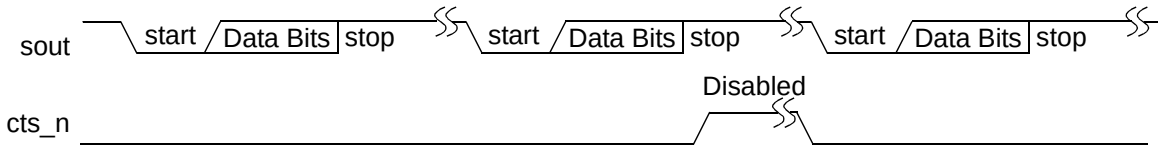


Figure 15: Auto CTS Timing

Programmable THRE Interrupt

The DW_apb_uart can be configured to have a Programmable THRE Interrupt mode available to increase system performance. If FIFOs are not implemented, then this mode cannot be selected. When Programmable THRE Interrupt mode is not selected, none of the logic is implemented and the mode cannot be enabled, reducing the overall gate counts.

When Programmable THRE Interrupt mode is selected it can be enabled via the Interrupt Enable Register (IER[7]). When FIFOs and the THRE Mode are implemented and enabled, THRE Interrupts (when also enabled) and dma_tx_req_n are active at, and below, a programmed transmitter FIFO empty threshold level, as opposed to empty, as shown in the flowchart in Figure 16 on page 55.

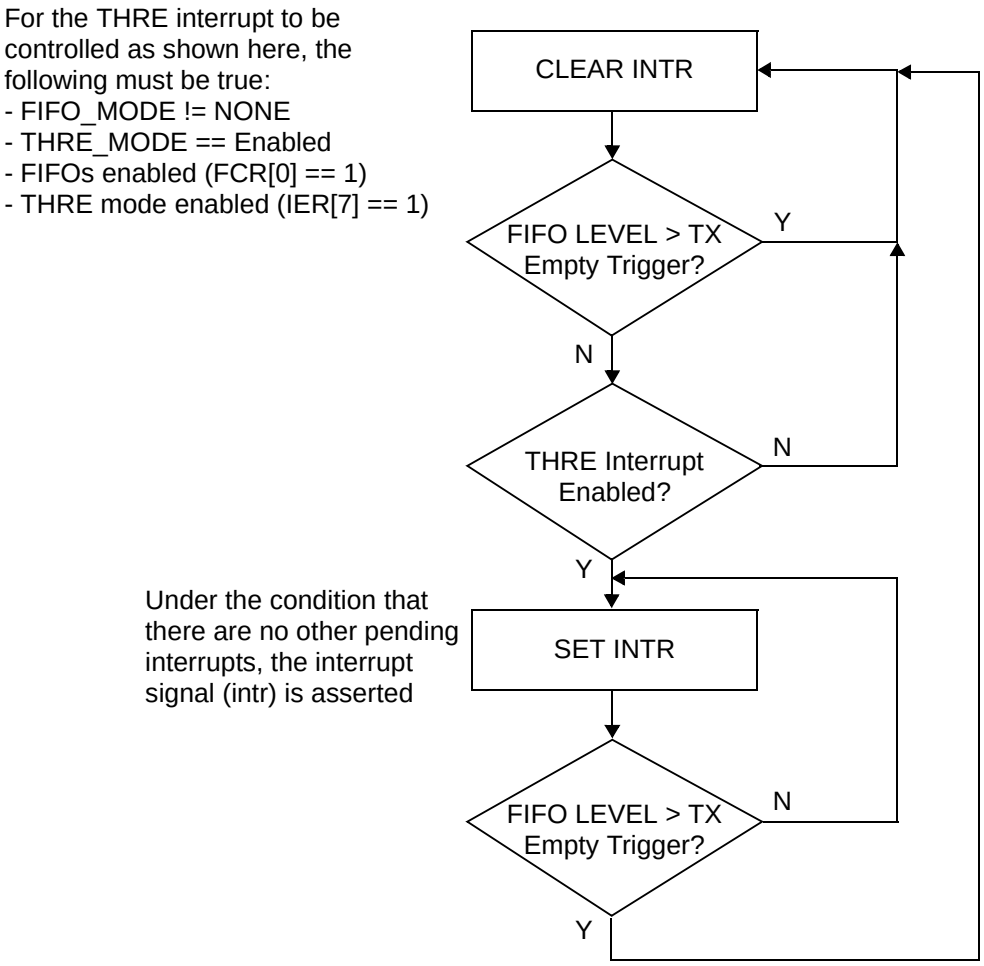


Figure 16: Flowchart of Interrupt Generation for Programmable THRE Interrupt Mode

This threshold level is programmed into FCR[5:4]. The available empty thresholds are: empty, 2, ¼ and ½. See “FCR” on page 98for threshold setting details. Selection of the best threshold value depends on the system's ability to begin a new transmission sequence in a timely manner. However, one of these thresholds should prove optimum in increasing system performance by preventing the transmitter FIFO from running empty.

In addition to the interrupt change, Line Status Register (LSR[5]) also switches function from indicating transmitter FIFO empty, to FIFO full. This allows software to fill the FIFO each transmit sequence by polling LSR[5] before writing another character. The flow then becomes, “fill transmitter FIFO whenever an interrupt occurs and there is data to transmit”, instead of waiting until the FIFO is completely empty. Waiting until the FIFO is empty causes a performance hit whenever the system is too busy to respond immediately. Further system efficiency is achieved when this mode is enabled in combination with Auto Flow Control.

Even if everything else is selected and enabled, if the FIFOs are disabled via FCR[0], the Programmable THRE Interrupt mode is also disabled. When not selected or disabled, THRE interrupts and LSR[5] function normally (both reflecting an empty THR or FIFO). The flowchart of THRE interrupt generation when not in programmable THRE interrupt mode is shown in Figure 17.

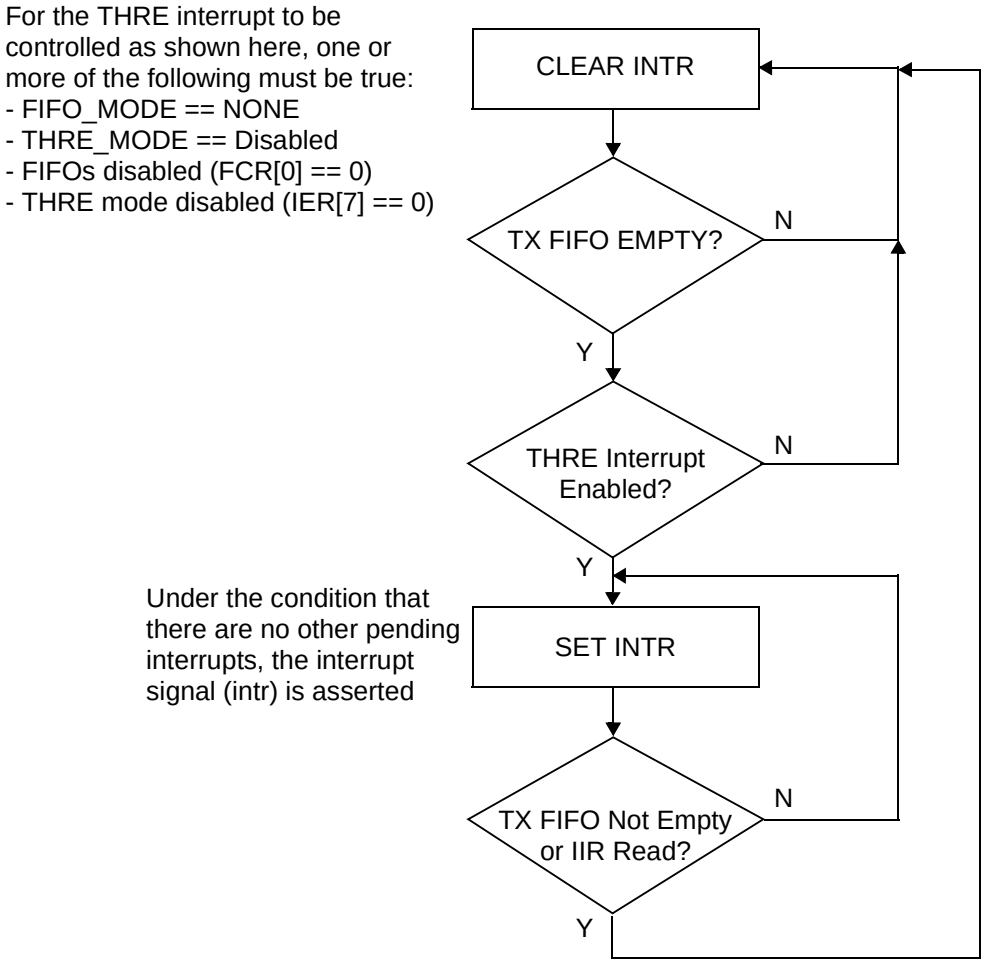


Figure 17: Flowchart of Interrupt generation when not in Programmable THRE Interrupt Mode

Clock Gate Enable

The DW_apb_uart can be configured to have a clock gate enable output. When the clock enable option is not selected, none of the logic is implemented, reducing the overall gate counts.

When the clock gate enable option is selected the clock gate enable signal(s) (uart_lp_req_pclk for single clock implementations or uart_lp_req_pclk and uart_lp_req_sclk for two clock implementations) is used to indicate that the transmit and receive pipeline is clear (no data), no activity has occurred, and the modem control input signals have not changed in more than one character time (the time taken to TX/RX a character) so clocks may be gated. (A character is made up of: *start bit* + *data bits* + *parity* (optional) + *stop bit(s)*). It is an indication that the UART is inactive, so clocks may be gated to put the

device in a low power (lp) mode. Therefore, the following must be true for at least one character time for the assertion of the clock gate enable signal(s) to occur:

- No data in the RBR (in non-FIFO mode) or the RX FIFO is empty (in FIFO mode)
- No data in the THR (in non-FIFO mode) or the TX FIFO is empty (in FIFO mode)
- sin/sir_in and sout/sir_out_n are inactive (sin/sir_in are kept high and sout is high or sir_out_n is low) indicating no activity
- No change on the modem control input signals

Note, the clock gate enable assertion does not occur in the following modes of operation:

- Loopback mode
- FIFO access mode
- When transmitting a break

For example, assume a DW_apb_uart that is configured to have a single clock (pclk) and is programmed to transmit and receive characters of 7 bits (1 start bit, 5 data bits and 1 stop bit) and the baud clock divisor is set to 1. Therefore, the uart_lp_req_pclk signal is asserted if the transmit and receive pipeline is clear, no activity has occurred and the modem control input signals have not changed for 112 (7×16) pclk cycles. Figure 18 illustrates this example.

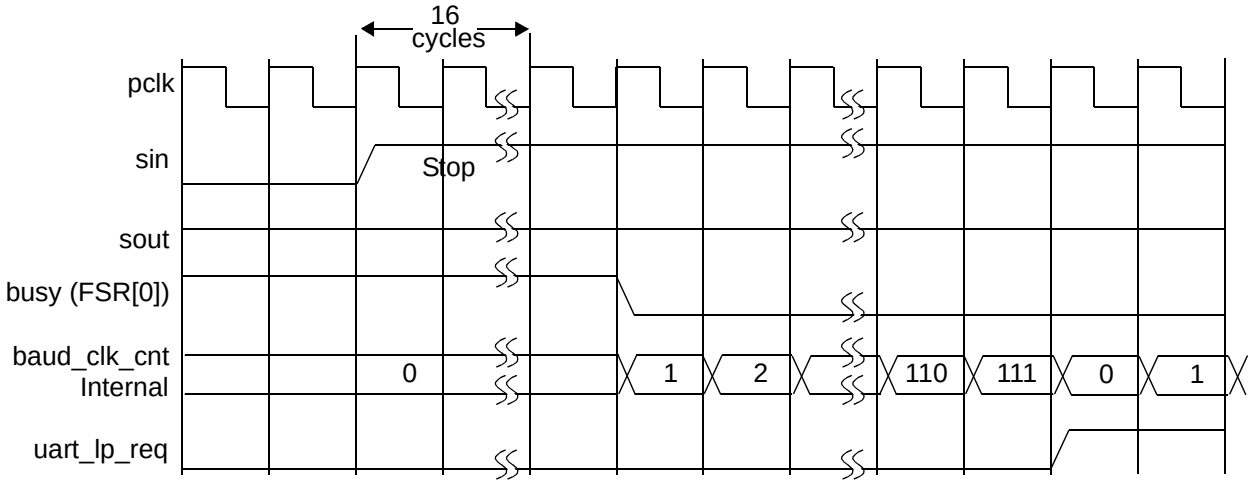


Figure 18: Clock Gate Enable Timing

When either of the signals sin or sir_in goes low, or a write to any of the registers is performed, or the modem control input signals have changed when the DW_apb_uart is in low power (sleep) mode, the clock gate enable signal(s) are de-asserted (as the assertion criteria are no longer met) so that the clock(s) is resumed. The time taken for the clock(s) to resume is important in the prevention of receive data synchronization problems. This is due to the fact that the DW_apb_uart RX block samples at the mid-point of each bit period (after approximately 8 baud clocks) in UART (RS323) mode and then every 16 baud clocks after that for a baud divisor of 1 that is 16 sclks (which for a single clock implementation is 16 pclks). Thus, if 8 or more sclk periods pass before the serial clock starts up again, the UART may get out of sync with the serial data it is receiving. That is, the receiver may sample into the second bit period and if it is still zero, think this is the start bit and so on. Therefore, to avoid this problem the clock should be resumed within 5 clock periods of the baud clock, which is the same as sclk if the baud divisor is set to one. This is worst case. If the divisor is greater, it gives a greater

number of sclk cycles available before the clock must resume. This means a sample point at the 13 baud clock (at the latest) out of the 16 that is transmitted for each bit period of the character in non-SIR mode.

Figure 19 shows the timing diagram that illustrates the previous scenario. This problem is magnified in SIR mode as the pulse width is only 3/16 of a bit period (3 baud clocks, which for a divisor of 1 is 3 sclks). Hence, it could be missed completely. The clocks must resume before 3 baud clock periods elapse. If the first character received while in sleep mode is used purely for wake up reasons and the actual character value is unimportant, this may not be a problem at all.

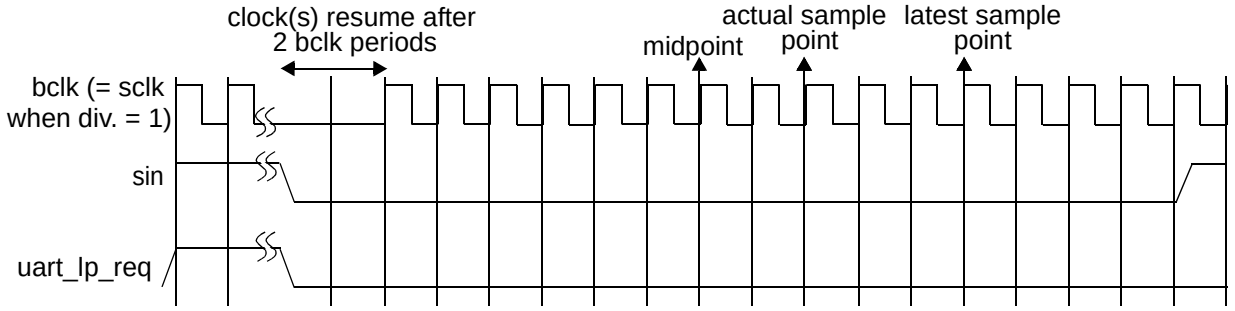


Figure 19: Resuming Clock(s) After Low Power Mode Timing

When the DW_apb_uart is configured to have two clocks, if the timing of the received signal is not affected by the synchronization problem, then the minimum time to receive a character, if the baud divisor is 1, is 112 sclks (1 start bit + 5 data bits + 1 stop bit = 7×16 =112). Therefore the pclk must be available before 112 sclk cycles pass so that the received character can be synchronized over to the pclk domain and stored in the RBR (in non-FIFO mode) or the RX FIFO (in FIFO mode).

DMA Support

The DW_apb_uart supports DMA signalling with the use of two output signals (dma_tx_req_n and dma_rx_req_n) to indicate when data is ready to be read or when the transmit FIFO is empty. The DW_apb_uart uses two DMA channels, one for the transmit data and one for the receive data. There are two DMA modes: mode 0 and mode 1, controllable via bit 3 of the FIFO Control Register (only DMA mode 0 is available when the FIFOs are not implemented or disabled).

DMA mode 0 supports single DMA data transfers at a time. In mode 0, the dma_tx_req_n signal goes active low under the following conditions:

- When the Transmitter Holding Register is empty in non-FIFO mode
- When the transmitter FIFO is empty in FIFO mode with Programmable THRE interrupt mode disabled
- When the transmitter FIFO is at or below the programmed threshold with Programmable THRE interrupt mode enabled

It goes inactive under the following conditions:

- When a single character has been written into the Transmitter Holding Register or transmitter FIFO with Programmable THRE interrupt mode disabled
- When the transmitter FIFO is above the threshold with Programmable THRE interrupt mode enabled.

The dma_rx_req_n signal goes active-low when there is a single character available in the Receiver FIFO or the Receive Buffer Register and it goes inactive when the Receive Buffer Register or Receiver FIFO are empty, depending on FIFO mode.

DMA mode 1 supports multi-DMA data transfers, where multiple transfers are made continuously until the receiver FIFO has been emptied or the transmit FIFO has been filled. In mode 1 the dma_tx_req_n signal is asserted under the following conditions:

- When the transmitter FIFO is empty with Programmable THRE interrupt mode disabled
- When the transmitter FIFO is at or below the programmed threshold with Programmable THRE interrupt mode enabled.

The dma_tx_req_n signal is de-asserted when the transmitter FIFO is completely full. The dma_rx_req_n signal is asserted when the Receiver FIFO is at or above the programmed trigger level, or a character timeout has occurred (that is, the conditions for a character timeout have been met over the required duration and does not refer to a character timeout interrupt, hence ERBFI does not need to be set to one for this to occur), and is de-asserted when the receiver FIFO becomes empty.

The DW_apb_uart can also be configured to have additional DMA interface signals if required for the DMA controller of choice (i.e. DW_ahb_dmac). When selected to have the additional DMA signals the assertion of the fixed DMA signals (dma_tx_req_n and dma_rx_req_n) is similar to what was detailed in the above DMA modes. That is, the dma_tx_req_n signal is asserted under the following conditions:

- When the Transmitter Holding Register is empty in non-FIFO mode
- When the transmitter FIFO is empty in FIFO mode with Programmable THRE interrupt mode disabled
- When the transmitter FIFO is at, or below the programmed threshold with Programmable THRE interrupt mode enabled.

The dma_rx_req_n signal is asserted under the following conditions:

- When there is a single character available in the Receive Buffer Register in non-FIFO mode
- When the Receiver FIFO is at or above the programmed trigger level in FIFO mode

With the presence of the additional handshaking signals the UART does not have to rely on internal status and level values to recognize the completion of a request and hence remove the request. Instead, the de-assertion of the DMA transmit and receive request is controlled by the assertion of the DMA transmit and receive acknowledge respectively.

When the UART is configured to have the additional DMA signals, the data flow (transfer lengths) responsibility falls on the DMA (DW_ahb_dmac) and is controlled by the programmed burst transaction lengths. Thus, there is no need for DMA modes and programming the FCR[3] has no effect.

The extra handshaking signals are explained in the DMA flow below for a DW_apb_uart that is configured with FIFOs and Programmable THRE interrupt mode.

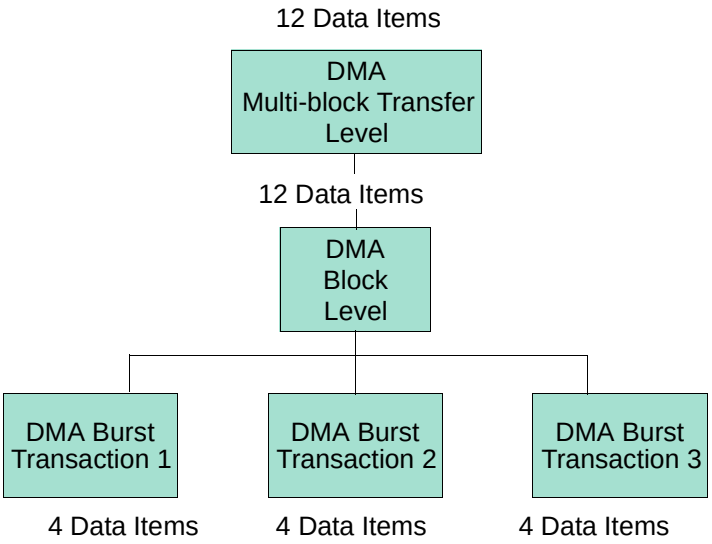
As a block flow control device, the DMA Controller is programmed by the processor with the number of data items (block size) that are to be transmitted or received by the DW_apb_uart; this is programmed into the BLOCK_TS field of the CTL*x* register.

The block is broken into a number of transactions, each initiated by a request from the DW_apb_uart. The DMA Controller must also be programmed with the number of data items (in this case, DW_apb_uart FIFO entries) to be transferred for each DMA request. This is also known as the burst transaction length, and is programmed into the SRC_MSIZE/DEST_MSIZE fields of the DW_ahb_dmac CTL*x* register for source and destination, respectively.

Figure 20 on page 60 shows a single block transfer, where the block size programmed into the DMA Controller is 12 and the burst transaction length is set to 4. In this case, the block size is a multiple of the burst transaction length. Therefore, the DMA block transfer consists of a series of burst transactions. If the DW_apb_uart makes a transmit request to this channel, four data items are written to the DW_apb_uart transmit FIFO. Similarly, if the DW_apb_uart makes a receive request to this channel, four data items are read from the DW_apb_uart receive FIFO. Three separate requests must be made to this DMA channel before all 12 data items are written or read.

Note

The source and destination transfer width settings in the DW_ahb_dmac – DMA.CTLx.SRC_TR_WIDTH and DMA.CTLx.DEST_TR_WIDTH – should be set to 3'b000 because the DW_apb_uart FIFOs are 8 bits wide.



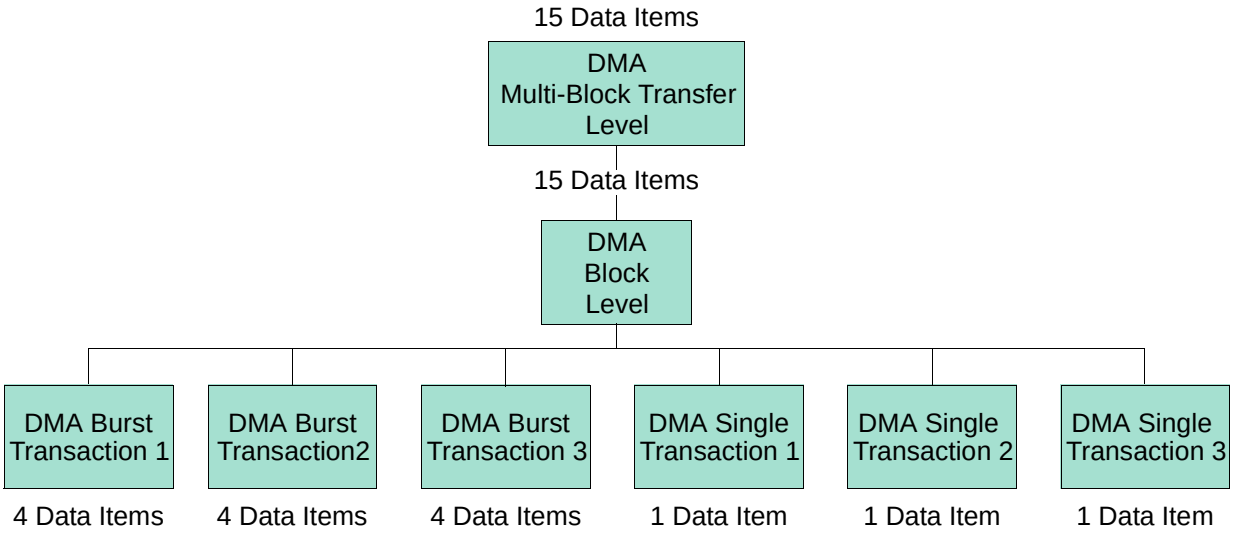
Block Size : DMA.CTL*x*.BLOCK_TS=12

Number of data items per source burst transaction : DMA.CTL*x*.SRC_MSIZE = 4

For a FIFO depth of 16: UART.FCR[7:6] = 01 = FIFO 1/4 full = DMA.CTL*x*.SRC_MSIZE
(for more information, refer to discussion on page 64)

Figure 20: Breakdown of DMA Transfer into Burst Transactions

When the block size programmed into the DMA Controller is not a multiple of the burst transaction length, as shown in Figure 21, a series of burst transactions followed by single transactions are needed to complete the block transfer.



Block Size : DMA.CTLx.BLOCK_TS=15

Number of data items per burst transaction : DMA.CTLx.DEST_MSIZE = 4

For a FIFO depth of 16: UART.FCR[5:4] = 10 =FIFO 1/4 full = 4 = DMA.CTLx.DEST_MSIZE
(for more information, refer to discussion on page 63)

Figure 21: Breakdown of DMA Transfer into Single and Burst Transactions

Transmit Watermark Level and Transmit FIFO Underflow

During DW_apb_uart serial transfers, transmit FIFO requests are made to the DW_ahb_dmac whenever the number of entries in the transmit FIFO is less than or equal to the decoded level of the Transmit Empty Trigger (TET) of the FCR register (bits 5:4); this is known as the watermark level. The DW_ahb_dmac responds by writing a burst of data to the transmit FIFO buffer, of length CTL*x*.DEST_MSIZE.

Data should be fetched from the DMA often enough for the transmit FIFO to perform serial transfers continuously; that is, when the FIFO begins to empty another DMA request should be triggered. Otherwise the FIFO runs out of data (underflow). To prevent this condition, you must set the watermark level correctly.

Choosing the Transmit Watermark Level

Consider the example where the assumption is made:

DMA.CTL*x*.DEST_MSIZE = FIFO_DEPTH - UART.FCR[5:4]

Here the number of data items to be transferred in a DMA burst is equal to the empty space in the Transmit FIFO. Consider two different watermark level settings.

Case 1: FCR[5:4] = 01, which decodes to 2

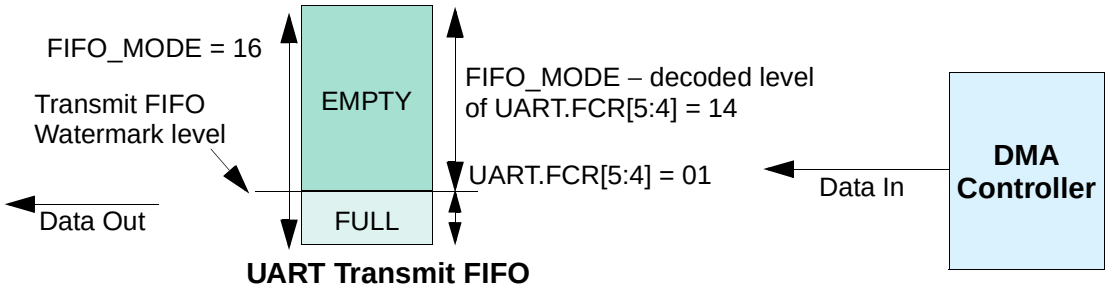


Figure 22: Case 1 Watermark Levels

Transmit FIFO watermark level = decoded level of UART.FCR[5:4] = 2
DMA.CTL*x*.DEST_MSIZE = FIFO_MODE – UART.FCR[5:4] = 14
UART transmit FIFO_MODE = 16
DMA.CTL*x*.BLOCK_TS = 56

Therefore, the number of burst transactions needed equals the block size divided by the number of data items per burst:

DMA.CTL*x*.BLOCK_TS/DMA.CTL*x*.DEST_MSIZE = 56/14 = 4

The number of burst transactions in the DMA block transfer is 4. But the watermark level, decoded level of UART.FCR[5:4], is quite low. Therefore, the probability of an UART underflow is high where the UART serial transmit line needs to transmit data, but where there is no data left in the transmit FIFO. This occurs because the DMA has not had time to service the DMA request before the transmit FIFO becomes empty.

Case 2: FCR[5:4] = 11, which equates to FIFO 1/2 full (decodes to 8)

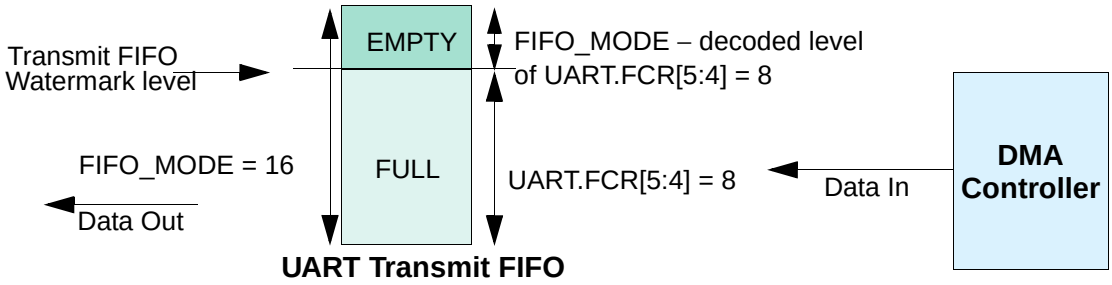


Figure 23: Case 2 Watermark Levels

Transmit FIFO watermark level = decoded level of UART.FCR[5:4] = 8
DMA.CTLx.DEST_MSIZE = FIFO_MODE - UART.FCR[5:4] = 8
UART transmit FIFO_MODE = 16
DMA.CTLx.BLOCK_TS = 56

Number of burst transactions in Block:

DMA.CTLx.BLOCK_TS/DMA.CTLx.DEST_MSIZE = 56/8 = 7

In this block transfer, there are 7 destination burst transactions in a DMA block transfer. But the watermark level, decoded level of UART.FCR[5:4], is high. Therefore, the probability of an UART underflow is low because the DMA controller has plenty of time to service the destination burst transaction request before the UART transmit FIFO becomes empty.

Thus, the second case has a lower probability of underflow at the expense of more burst transactions per block. This provides a potentially greater amount of AMBA bursts per block and worse bus utilization than the former case.

Therefore, the goal in choosing a watermark level is to minimize the number of transactions per block, while at the same time keeping the probability of an underflow condition to an acceptable level. In practice, this is a function of the ratio of the rate at which the UART transmits data to the rate at which the DMA can respond to destination burst requests.

For example, promoting the channel to the highest priority channel in the DMA, and promoting the DMA master interface to the highest priority master in the AMBA layer, increases the rate at which the DMA controller can respond to burst transaction requests. This in turn allows the user to decrease the watermark level, which improves bus utilization without compromising the probability of an underflow occurring.

Selecting DEST_MSIZE and Transmit FIFO Overflow

As can be seen from Figure 23 on page 62, programming DMA.CTL*x*.DEST_MSIZE to a value greater than the watermark level that triggers the DMA request may cause overflow when there is not enough space in the UART transmit FIFO to service the destination burst request. Therefore, the following equation must be adhered to in order to avoid overflow:

DMA.CTL*x*.DEST_MSIZE <= UART.FIFO_DEPTH – decoded level of UART.FCR[5:4] **(1)**

In topic "Case 2: FCR[5:4] = 11, which equates to FIFO 1/2 full (decodes to 8)", the amount of space in the transmit FIFO at the time the burst request is made is equal to the destination burst length, DMA.CTL*x*.DEST_MSIZE. Thus, the transmit FIFO may be full, but not overflowed, at the completion of the burst transaction.

Therefore, for optimal operation, DMA.CTL*x*.DEST_MSIZE should be set at the FIFO level that triggers a transmit DMA request; that is:

DMA.CTLx.DEST_MSIZE = UART.FIFO_DEPTH – decoded level of UART.FCR[5:4] **(2)**

This is the setting used in Figure 21 on page 61.

Adhering to equation (2) reduces the number of DMA bursts needed for a block transfer, and this in turn improves AMBA bus utilization.

Note

The transmit FIFO is not full at the end of a DMA burst transfer if the UART has successfully transmitted one data item or more on the UART serial transmit line during the transfer.

Receive Watermark Level and Receive FIFO Overflow

During DW_apb_uart serial transfers, receive FIFO requests are made to the DW_ahb_dmac whenever the number of entries in the receive FIFO is at or above the decoded level of Receiver Trigger (RT) of the FCR[7:6]. This is known as the watermark level. The DW_ahb_dmac responds by writing a burst of data to the transmit FIFO buffer of length CTL*x*.SRC_MSIZE.

Data should be fetched by the DMA often enough for the receive FIFO to accept serial transfers continuously; that is, when the FIFO begins to fill, another DMA transfer is requested. Otherwise, the FIFO fills with data (overflow). To prevent this condition, you must correctly set the watermark level.

Choosing the Receive Watermark Level

Similar to choosing the transmit watermark level described earlier, the receive watermark level, decoded level of FCR[7:6], should be set to minimize the probability of overflow. It is a trade-off between the number of DMA burst transactions required per block versus the probability of an overflow occurring.

Selecting SRC_MSIZE and Receive FIFO Underflow

As can be seen in Figure 24, programming a source burst transaction length greater than the watermark level may cause underflow when there is not enough data to service the source burst request. Therefore, the following equation must be adhered to avoid underflow:

DMA.CTLx.SRC_MSIZE <= decoded level of FCR[7:6] **(4)**

If the number of data items in the receive FIFO is equal to the source burst length at the time the burst request is made – DMA.CTL*x*.SRC_MSIZE – the receive FIFO may be emptied, but not underflowed, at the completion of the burst transaction. For optimal operation, DMA.CTL*x*.SRC_MSIZE should be set at the watermark level; that is:

DMA.CTL*x*.SRC_MSIZE = decoded level of FCR[7:6] **(5)**

Adhering to equation (5) reduces the number of DMA bursts in a block transfer, and this in turn can improve AMBA bus utilization.

Note

The receive FIFO is not empty at the end of the source burst transaction if the UART has successfully received one data item or more on the UART serial receive line during the burst.

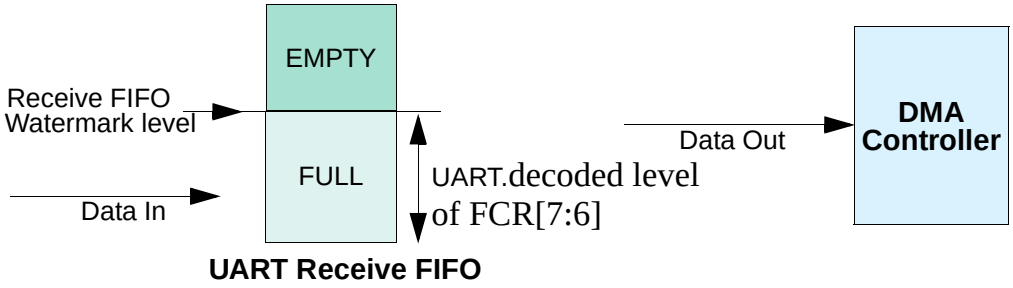


Figure 24: UART Receive FIFO

Handshaking Interface Operation

dma_tx_req_n, dma_rx_req_n – The request signals for source and destination, dma_tx_req_n and dma_rx_req_n, are activated when their corresponding FIFOs reach the watermark levels as discussed earlier.

The DW_ahb_dmac uses edge detection of the dma_tx_req_n signal/dma_rx_req_n to identify a request on the channel. Upon reception of the dma_tx_ack_n/dma_rx_ack_n signal from the DW_ahb_dmac to indicate the burst transaction is complete, the DW_apb_uart de-asserts the burst request signals, dma_tx_req_n/dma_rx_req_n, until dma_tx_ack_n/dma_rx_ack_n is de-asserted by the DW_ahb_dmac.

When the DW_apb_uart samples that dma_tx_ack_n/dma_rx_ack_n is de-asserted, it can re-assert the dma_tx_req_n/dma_rx_req_n of the request line if their corresponding FIFOs exceed their watermark levels (back-to-back burst transaction). If this is not the case, the DMA request lines remain de-asserted. Figure 25 on page 65 shows a timing diagram of a burst transaction where pclk = hclk. Figure 26 shows two back-to-back burst transactions where the hclk frequency is twice the pclk frequency.

The handshaking loop is as follows:

1. dma_tx_req_n/dma_rx_req_n asserted by DW_apb_uart
2. dma_tx_ack_n/dma_rx_ack_n asserted by DW_ahb_dmac
3. dma_tx_req_n/dma_rx_req_n de-asserted by DW_apb_uart
4. dma_tx_ack_n/dma_rx_ack_n de-asserted by DW_ahb_dmac
5. dma_tx_req_n/dma_rx_req_n re-asserted by DW_apb_uart, if back-to-back transaction is required

Note

The burst transaction request signals, dma_tx_req_n and dma_rx_req_n, are generated in the DW_apb_uart off pclk and sampled in the DW_ahb_dmac by hclk. The acknowledge signals, dma_tx_ack_n and dma_rx_ack_n, are generated in the DW_ahb_dmac off hclk and sampled in the DW_apb_uart of pclk. The handshaking mechanism between the DW_ahb_dmac and the DW_apb_uart supports quasi-synchronous clocks; that is, hclk and pclk must be phase-aligned, and the hclk frequency must be a multiple of the pclk frequency.

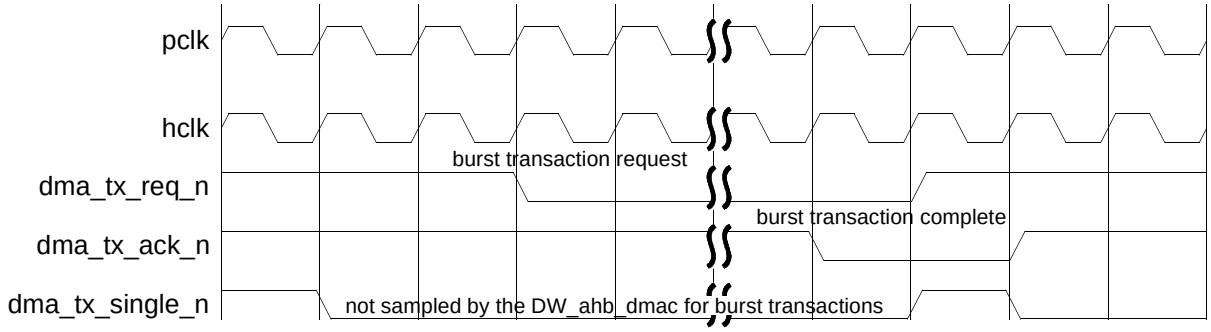


Figure 25: Burst Transaction – pclk = hclk

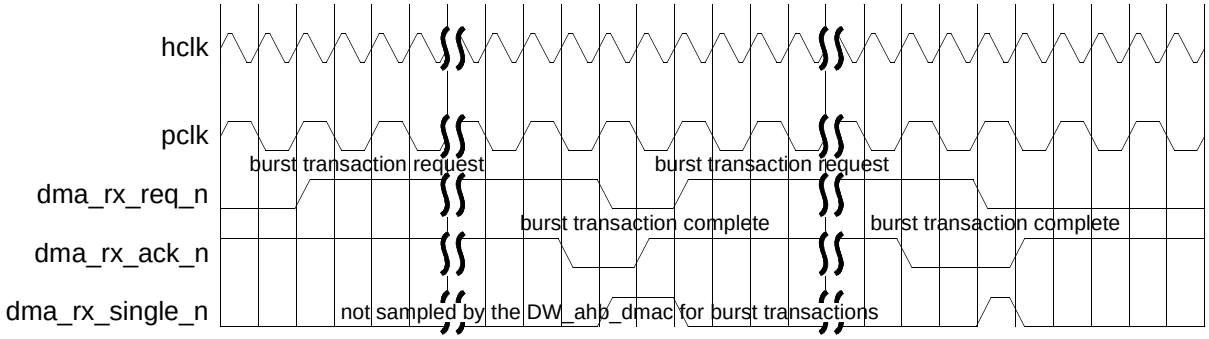


Figure 26: Back-to-Back Burst Transactions – hclk = 2*pclk

Note the following the following:

1. The burst request lines, dma_tx_req_n/dma_rx_req_n, once asserted remain asserted until their corresponding dma_tx_ack_n/dma_rx_ack_n signal is received even if the respective FIFOs drop below their watermark levels during the burst transaction.
2. The dma_tx_req_n/dma_rx_req_n signals are de-asserted when their corresponding dma_tx_ack_n/dma_rx_ack_n signals are asserted, even if the respective FIFOs exceed their watermark levels.

dma_tx_single_n, dma_rx_single_n – The dma_tx_single_n signal is a status signal, and is asserted when there is at least one free entry in the transmit FIFO, and cleared when the transmit FIFO is full. The dma_rx_single_n signal is a status signal, and is asserted when there is at least one valid data entry in the receive FIFO, and is cleared when the receive FIFO is empty.

These signals are needed by only the DW_ahb_dmac for the case where the block size, CTL*x*.BLOCK_TS, that is programmed into the DW_ahb_dmac is not a multiple of the burst transaction length, CTL*x*.SRC_MSIZE, CTL*x*.DEST_MSIZE, as shown in Figure 21. In this case, the DMA single outputs inform the DW_ahb_dmac that it is still possible to perform single data item transfers, so it can access all data items in the transmit/receive FIFO and complete the DMA block transfer. The DMA single outputs from the DW_apb_uart are not sampled by the DW_ahb_dmac otherwise. This is illustrated in the following example.

Consider first an example where the receive FIFO channel of the DW_apb_uart is as follows:

DMA.CTLx.SRC_MSIZE = decoded level of UART.FCR[7:6] = 4
DMA.CTLx.BLOCK_TS = 12

For the example in Figure 20, with the block size set to 12, the dma_rx_req_n signal is asserted when four data items are present in the receive FIFO. The dma_rx_req_n signal is asserted three times during the DW_apb_uart serial transfer, ensuring that all 12 data items are read by the DW_ahb_dmac. All DMA requests read a block of data items and no single DMA transactions are required. This block transfer is made up of three burst transactions.

Now, for the following block transfer:

DMA.CTLx.SRC_MSIZE = decoded level of UART.FCR[7:6] = 4
DMA.CTLx.BLOCK_TS = 15

The first 12 data items are transferred as already described using 3 burst transactions. But when the last three data frames enter the receive FIFO, the dma_rx_req_n signal is not activated because the FIFO level is below the watermark level. The DW_ahb_dmac samples dma_rx_single_n and completes the DMA block transfer using three single transactions. The block transfer is made up of three burst transactions followed by three single transactions.

Figure 27 shows a single transaction. The handshaking loop is as follows:

1. dma_tx_single_n/dma_rx_single_n asserted by DW_apb_uart
2. dma_tx_ack_n/dma_rx_ack_n asserted by DW_ahb_dmac
3. dma_tx_single_n/dma_rx_single_n de-asserted by DW_apb_uart
4. dma_tx_ack_n/dma_rx_ack_n de-asserted by DW_ahb_dmac

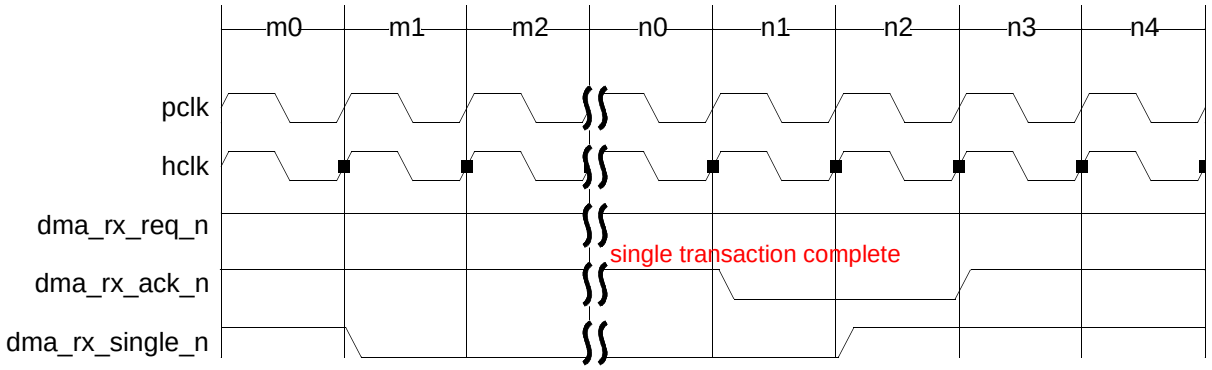


Figure 27: Single Transaction

Figure 28 shows a burst transaction, followed by three back-to-back single transactions, where the hclk frequency is twice the pclk frequency.

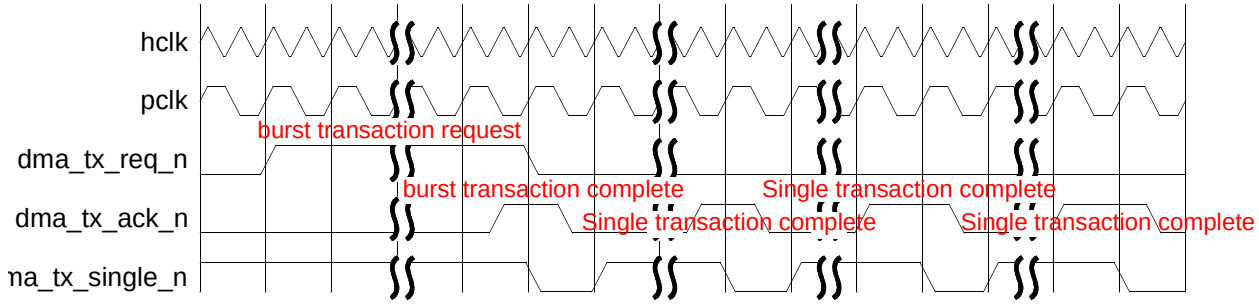


Figure 28: Burst Transaction + 3 Back-to-Back Singles – hclk = 2*pclk

Note

The single transaction request signals, dma_tx_single_n and dma_rx_single_n, are generated in the DW_apb_uart on the pclk edge and sampled in DW_ahb_dmac on hclk. The acknowledge signals, dma_tx_ack_n and dma_rx_ack_n, are generated in the DW_ahb_dmac on the hclk edge hclk and sampled in the DW_apb_uart on pclk. The handshaking mechanism between the DW_ahb_dmac and the DW_apb_uart supports quasi-synchronous clocks; that is, hclk and pclk must be phase aligned and the hclk frequency must be a multiple of pclk frequency.

4 Parameters

This chapter describes the configuration parameters used by the DW_apb_uart.

Parameter Descriptions

The following list identifies the configurable parameters supported by the DW_apb_uart:

Table 4: Top-Level Parameters

Label	Parameter Definition
Use DesignWare Foundation Synthesis Library (active only when source license available)	**Parameter Name:** USE_FOUNDATION **Legal Values:** True (1) or False (0) **Default Value:** True; only if Source license is available. **Dependencies:** Must have Source license. **Description:** Enables source code customers to write out RTL without having a DesignWare license, or to retain DesignWare Foundation Building Block Library parts in their design.
APB Data Bus Width	**Parameter Name:** APB_DATA_WIDTH **Values:** 8, 16, 32 **Default Value:** 32 **Dependencies:** None **Description:** Width of APB data bus to which this component is attached. Note that even though the data width can be set to 8, 16 or 32, only the lowest 8 data bits are ever used, since register access is on 32-bit boundaries. All other bits are held at static 0.
UART FIFO Depth	**Parameter Name:** FIFO_MODE **Values:** NONE, 16, 32, …, 2 KB (2048) **Default Value:** 16 **Dependencies:** None **Description:** Receiver and Transmitter FIFO depth in bytes. A setting of NONE means no FIFOs, which implies the 16450-compatible mode of operation. Most enhanced features are unavailable in the 16450 mode such as the Auto Flow Control and Programmable THRE interrupt modes. Setting a FIFO depth greater than 256 restricts the FIFO Memory to External only. For more details, refer to “FIFO Support” on page 48.

Table 4: Top-Level Parameters (Continued)

Label	Parameter Definition
FIFO Memory Type	**Parameter Name:** MEM_SELECT_USER **Values:** External (0) - User-supplied memory Internal (1) - DesignWare memory instantiation **Default Value:** External (0) **Dependencies:** Only changeable to Internal if (FIFO_MODE != NONE) and (FIFO_MODE <= 256) **Description:** Selects between external, user-supplied memory or internal DesignWare memory (DW_ram_r_w_s_dff) for the receiver and transmitter FIFOs. FIFO depths greater than 256 restrict FIFO Memory selection to external. In addition, selection of internal memory restricts the Memory Read Port Type to D-flip-flop-based, synchronous read port RAMs.
(None)	**Parameter Name:** MEM_MODE_USER **Values:** Not changeable **Default Value:** Async (0) **Description:** This non-changeable parameter has been retained in this release of the DW_apb_uart for backward compatibility with pre-3.00a versions of this component. The later versions (post-3.00a) of the DW_apb_uart no longer require logic changes that used to be controlled by this parameter when you used either synchronous or asynchronous read port RAMs.
Asynchronous Serial Clock Support	**Parameter Name:** CLOCK_MODE **Values:** Disabled (1) - One clock Enabled (2) - Two clocks **Default Value:** Disabled (1) **Dependencies:** Asynchronous Serial Clock Support is automatically enabled when SIR_LP_MODE = Enabled or when SIR_LP_RX = Enabled. **Description:** When set to Disabled, the DW_apb_uart is implemented with one system clock (pclk). When set to Enabled, two system clocks (pclk and sclk) are implemented in order to accommodate accurate serial baud rate settings, as well as APB bus interface requirements. Selecting Disabled, or a one-system clock, greatly restricts system clock settings available for accurate baud rates. For more details, refer to “Clock Support” on page 49.
Auto Flow Control	**Parameter Name:** AFCE_MODE **Values:** Disabled (0) - Auto Flow Control not available Enabled (1) - Auto Flow Control **Default Value:** Disabled (0) **Dependencies:** Changeable to Enabled only when (FIFO_MODE != NONE) **Description:** Configures the peripheral to have the 16750-compatible auto flow control mode. For more details, refer to “Auto Flow Control” on page 51.

Table 4: Top-Level Parameters (Continued)

Label	Parameter Definition
Programmable THRE Interrupt Mode	**Parameter Name:** THRE_MODE_USER **Values:** Disabled (0) - THRE Interrupt mode not available Enabled (1) - THRE Interrupt mode **Default Value:** Disabled (0) **Dependencies:** Changeable to Enabled only when (FIFO_MODE != NONE) **Description:** Configures the peripheral to have a programmable Transmitter Hold Register Empty (THRE) Interrupt mode. For more information, refer to "Programmable THRE Interrupt" on page 54.
IrDA SIR Mode Support	**Parameter Name:** SIR_MODE **Values:** Disabled (0) - IrDA SIR mode not available Enabled (1) - IrDA SIR mode **Default Value:** Disabled (0) **Dependencies:** None **Description:** Configures the peripheral to have IrDA 1.0 SIR infrared mode. For more details, refer to "IrDA 1.0 SIR Protocol" on page 47.
Include Clock Gate Enable Output on I/F?	**Parameter Name:** CLK_GATE_EN **Values:** No (0) - Clock gate enable is not available Yes (1) - Clock gate enable **Default Value:** No (0) **Dependencies:** None **Description:** Configures the peripheral to have a clock gate enable output signal on the interface that indicates that the device is inactive so clocks may be gated.
Include FIFO Access Mode?	**Parameter Name:** FIFO_ACCESS **Values:** No (0) - FIFO access mode is not available Yes (1) - FIFO access mode **Default Value:** No (0) **Dependencies:** None **Description:** Configures the peripheral to have a programmable FIFO access mode. This is used for test purposes to allow the receive FIFO to be written and the transmit FIFO to be read when FIFOs are implemented and enabled. When FIFOs are not implemented or not enabled it allows the RBR to be written and the THR to be read. For more details, refer to "FIFO Support" on page 48.
Include Additional DMA Signals on I/F?	**Parameter Name:** DMA_EXTRA **Values:** No (0) - Additional DMA signals not included Yes (1) - Additional DMA signals included **Default Value:** No (0) **Dependencies:** None **Description:** Configures the peripheral to have four additional DMA signals on the interface so that the device is compatible with the DesignWare DMA controller interface requirements.

Table 4: Top-Level Parameters (Continued)

Label	Parameter Definition
Active low DMA Signals?	**Parameter Name:** DMA_POL **Values:** No (0) - DMA signals set to active-high Yes (1) - DMA signals set to active-low **Default Value:** Yes (1) **Dependencies:** None **Description:** Selects the polarity of the DMA interface signals.
Low Power IrDA SIR Mode Support	**Parameter Name:** SIR_LP_MODE **Values:** Disabled (0) - Low-power IrDA SIR mode not available Enabled (1) - Low-power IrDA SIR mode **Default Value:** Disabled (0) **Dependencies:** This is only changeable when SIR_MODE = Enabled. **Description:** Configures the peripheral to operate in a low-power IrDA SIR mode. As the DW_apb_uart does not support a low-power mode with a counter system to maintain a 1.63us infrared pulse, Asynchronous Serial Clock Support is automatically enabled, and the sclk must be fixed to 1.8432MHz. This provides a 1.63us sir_out_n pulse at 115.2Kbaud.
Support for IrDA SIR Low-Power Reception Capabilities	**Parameter Name:** SIR_LP_RX **Values:** Disabled (0) Enabled (1) **Default Value:** Disabled (0) **Dependencies:** This is only changeable when SIR_MODE = Enabled. **Description:** Configures the peripheral to have SIR low-power reception capabilities. Asynchronous Serial Clock support is automatically enabled in this mode.
Include On-chip Debug Output Signals on I/F?	**Parameter Name:** DEBUG **Values:** No (0) - On-chip debug signals not included Yes (1) - On-chip debug signals included **Default Value:** No (0) **Dependencies:** None **Description:** Configures the peripheral to have on-chip debug signals on the interface.
Include Baud Clock Reference Output Signal (baudout_n) on I/F?	**Parameter Name:** BAUD_CLK **Values:** No (0) - baudout_n signal not included Yes (1) - baudout_n signal included **Default Value:** Yes (1) **Dependencies:** None **Description:** Configures the peripheral to have a baud clock reference output (baudout_n) signal on the interface.

Table 4: Top-Level Parameters (Continued)

Label	Parameter Definition
Make FIFO Status and Shadow Register Options Available?	**Parameter Name:** ADDITIONAL_FEATURES **Values:** No (0) - FIFO Status and Shadow registers not included Yes (1) - Registers included **Default Value:** No (0) **Dependencies:** None **Description:** Configures the peripheral to have both the FIFO Status registers and the Shadow registers available. Also configures the peripheral to have the configuration ID, UART component version and the peripheral ID registers.
Include Software Accessible FIFO Status Registers?	**Parameter Name:** FIFO_STAT **Values:** No (0) - FIFO Status registers not included Yes (1) - FIFO Status registers included **Default Value:** No (0) **Dependencies:** This is only changeable when FIFO_MODE != NONE and ADDITIONAL_FEATURES = YES. **Description:** Configures the peripheral to have three additional FIFO status registers.
Include Additional Shadow Registers for Reducing Software Overhead?	**Parameter Name:** SHADOW **Values:** No (0) - Additional registers not included Yes (1) - Additional Shadow registers included **Default Value:** No (0) **Dependencies:** This is only changeable when ADDITIONAL_FEATURES = YES. **Description:** Configures the peripheral to have seven additional registers that shadow some of the existing register bits that are regularly modified by software. These can be used to reduce the software overhead that is introduced by having to perform read-modify writes.
(None)	**Parameter Name:** LATCH_MODE_USER **Values: (**Not changeable) **Default Value:** No (0) **Dependencies:** None **Description:** This is a non-changeable parameter that is included for software backward compatibility. That is so that no errors arise when the peripheral is used with existing software.
Include Configuration Identification Register	**Parameter Name:** UART_ADD_ENCODED_PARAMS **Values:** No (0) - configuration register not included Yes (1) - configuration register included **Default Value:** No (0) **Dependencies:** This is only changeable when ADDITIONAL_FEATURES = YES. **Description:** Configures the peripheral to have a configuration register.

Table 5: Legacy Parameters

Label	Parameter Definition
(None)	**Parameter Name:** MEM_MODE_USER **Values:** Not changeable **Default Value:** Async (0) **Description:** This non-changeable parameter has been retained in this release of the DW_apb_uart for backward compatibility with pre-3.00a versions of this component. The later versions (post-3.00a) of the DW_apb_uart no longer require logic changes that used to be controlled by this parameter when you used either synchronous or asynchronous read port RAMs.
(None)	**Parameter Name:** LATCH_MODE_USER **Values: (**Not changeable) **Default Value:** No (0) **Dependencies:** None **Description:** This is a non-changeable parameter that is included for software backward compatibility. That is so that no errors arise when the peripheral is used with existing software.

5 Signals

The following subsections describe the DW_apb_uart signals:

DW_apb_uart Interface Diagram

Figure 29 shows an I/O diagram of the DW_apb_uart.

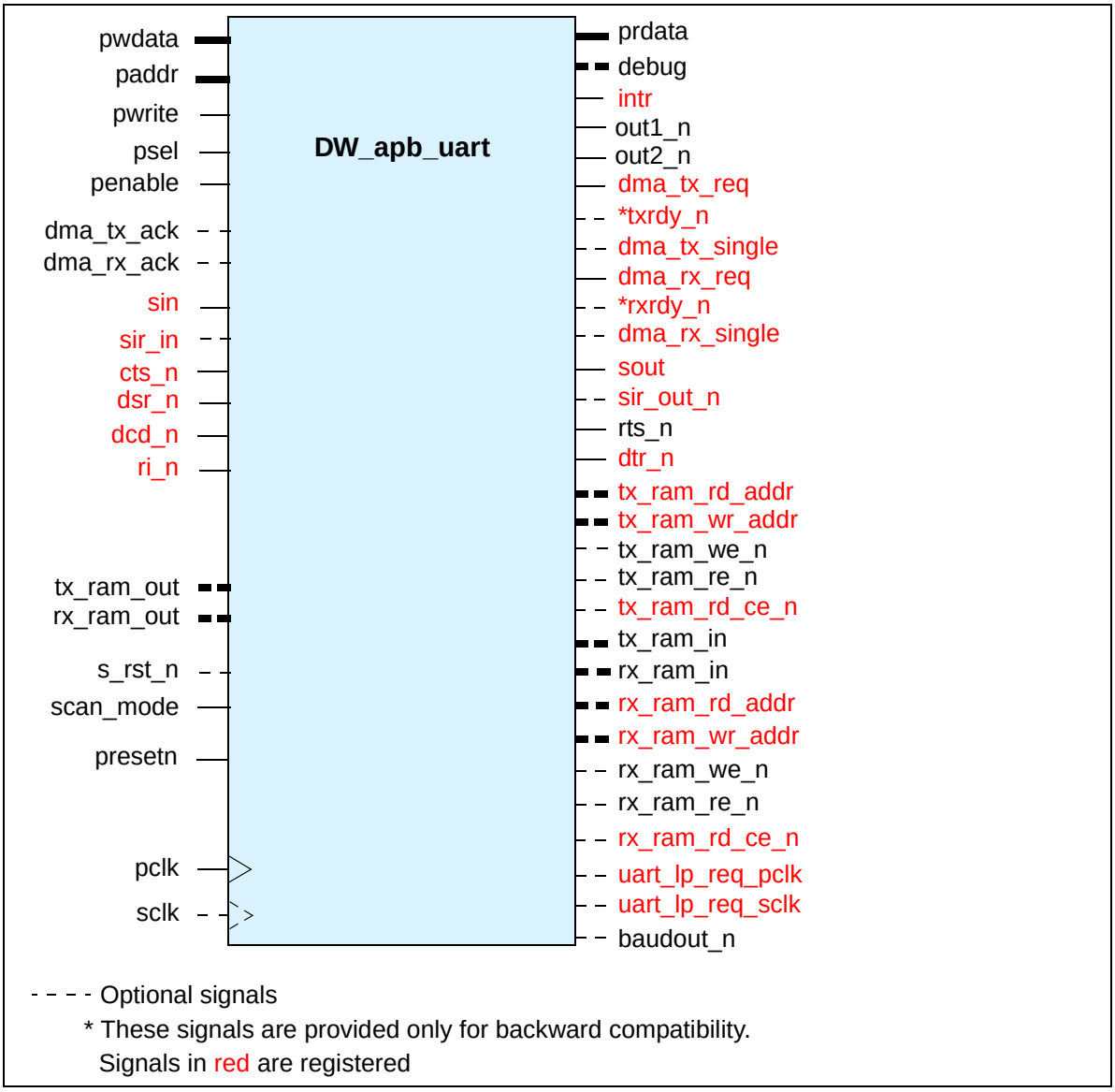


Figure 29: DW_apb_uart I/O Diagram

DW_apb_uart Signal Descriptions

Table 6 provides a list and description of the DW_apb_uart signals.

Note

The Description column in Table 6 provides detailed information about each signal.

In the **Registered** field, a “Yes” indicates whether an I/O signal is directly connected to an internal register and nothing else. An I/O signal is also considered to be registered if the signal is connected to one or more inverters or buffers between the I/O port and internal register, but not connected to any logic that involves another signal.

The **Input/Output Delay** field provides the percentage of the clock cycle assumed to be used by logic outside this design. The given value is used to automatically define the default synthesis constraints for input/output delay. You can override these default values in the Specify Port Constraints activity in coreConsultant. For more information, refer to “Create Gate-Level Netlist” on page 32.

Table 6: DW_apb_uart Signal Description

Name[a]	Width	I/O	Description
APB Slave Interface			
pclk	1 bit	I	APB clock used in the APB interface to program registers **Registered:** No **Synchronous to:** N/A **Default Input Delay:** N/A
presetn	1 bit	I	APB clock-domain reset **Active State:** Low **Registered:** No **Synchronous to:** pclk on de-assertion, asynchronous on assertion **Default Input Delay:** 50% of pclk
psel	1 bit	I	APB peripheral select **Active State:** High **Registered:** No **Synchronous to:** pclk **Default Input Delay:** 50% of pclk
paddr*n*	8 bits	I	APB address bus. Uses the lower bits of the APB address bus for register decode. **Registered:** No **Synchronous to:** pclk **Default Input Delay:** 50% of pclk

Table 6: DW_apb_uart Signal Description

Name[a]	Width	I/O	Description
pwdata*n*	*APB_DATA_WIDTH*	I	APB write data bus **Registered:** No **Synchronous to:** pclk **Default Input Delay:** 50% of pclk
pwrite	1 bit	I	APB write control **Active State:** High **Registered:** No **Synchronous to:** pclk **Input Delay:** 50% of pclk
penable	1 bit	I	APB enable control used for timing read/write operations **Active State:** High **Registered:** No **Synchronous to:** pclk **Default Input Delay:** 50% of pclk
prdata*n*	*APB_DATA_WIDTH*	O	APB read data bus **Registered:** Yes **Synchronous to:** pclk **Default Output Delay:** 90% of pclk
Application Interface			
sclk	1 bit	I	Serial interface clock **Registered:** No **Synchronous to:** Not applicable **Default Input Delay:** Not applicable **Dependencies:** Only present when `CLOCK_MODE==Enabled`.
s_rst_n	1 bit	I	Serial interface reset **Active State:** Low **Registered:** No **Synchronous to:** sclk on de-assertion, asynchronous on assertion **Default Input Delay:** 40% of sclk **Dependencies:** Only present when `CLOCK_MODE==Enabled`.

Table 6: DW_apb_uart Signal Description

Name[a]	Width	I/O	Description
scan_mode	1 bit	I	Scan mode used to ensure that test automation tools can control all asynchronous flop signals. During scan this signal must be set high all the time. In normal operation you must tie this signal low. **Active State:** High **Registered:** No **Synchronous to:** pclk **Default Input Delay:** 20% of pclk
FIFO Interface (**Dependencies:** Present only when `FIFO_MODE!=NONE` and `MEM_SELECT==External`)			
tx_ram_out*n*	8 bits	I	Data to the transmit FIFO RAM **Registered:** No **Synchronous to:** pclk **Default Input Delay:** 60% of pclk
tx_ram_in*n*	8 bits	O	Data from the transmit FIFO RAM **Registered:** No **Synchronous to:** pclk **Default Output Delay:** 60% of pclk
tx_ram_rd_addr*n*	$\log_2$(*FIFO_MODE*)	O	Read address pointer for the transmit FIFO RAM **Registered:** Yes **Synchronous to:** pclk **Default Output Delay:** 60% of pclk
tx_ram_wr_addr*n*	$\log_2$(*FIFO_MODE*)	O	Write address pointer for transmit FIFO RAM **Registered:** Yes **Synchronous to:** pclk **Default Output Delay:** 60% of pclk
tx_ram_we_n	1 bit	O	Write enable for the transmit FIFO RAM **Active State:** Low **Registered:** No **Synchronous to:** pclk **Default Output Delay:** 60% of pclk
tx_ram_re_n	1 bit	O	Read enable for the transmit FIFO RAM wake-up **Active State:** Low **Registered:** No **Synchronous to:** pclk **Default Output Delay:** 60% of pclk

Table 6: DW_apb_uart Signal Description

Name[a]	Width	I/O	Description
tx_ram_rd_ce_n	1 bit	O	Read port chip enable for transmit FIFO RAM **Active State:** Low **Registered:** Yes **Synchronous to:** pclk **Default Output Delay:** 60% of pclk
rx_ram_out*n*	10 bits	I	Data to the receive FIFO RAM **Registered:** No **Synchronous to:** pclk **Default Input Delay:** 60% of pclk
rx_ram_in*n*	10 bits	O	Data from the receive FIFO RAM **Registered:** No **Synchronous to:** pclk **Default Output Delay:** 60% of pclk
rx_ram_rd_addr*n*	$\log_2$(*FIFO_MODE*)	O	Read address pointer for the receive FIFO RAM **Registered:** Yes **Synchronous to:** pclk **Default Output Delay:** 60% of pclk
rx_ram_wr_addr*n*	$\log_2$(*FIFO_MODE*)	O	Write address pointer for the receive FIFO RAM **Registered:** Yes **Synchronous to:** pclk **Default Output Delay:** 60% of pclk
rx_ram_we_n	1 bit	O	Write enable for the receive FIFO RAM **Active State:** Low **Registered:** No **Synchronous to:** pclk **Default Output Delay:** 60% of pclk
rx_ram_re_n	1 bit	O	Read enable for the receive FIFO RAM wake-up **Active State:** Low **Registered:** No **Synchronous to:** pclk **Default Output Delay:** 60% of pclk
rx_ram_rd_ce_n	1 bit	O	Read port chip enable for receive FIFO RAM **Active State:** Low **Registered:** Yes **Synchronous to:** pclk **Default Output Delay:** 60% of pclk

Table 6: DW_apb_uart Signal Description

Name[a]	Width	I/O	Description
Modem Interface			
cts_n	1 bit	I	Clear To Send Modem Status **Active State:** Low **Registered:** Yes **Synchronous to:** pclk **Default Input Delay:** No specific requirement
dsr_n	1 bit	I	Data Set Ready Modem Status input **Active State:** Low **Registered:** Yes **Synchronous to:** pclk **Default Input Delay:** No specific requirement
dcd_n	1 bit	I	Data Carrier Detect Modem Status input **Active State:** Low **Registered:** Yes **Synchronous to:** pclk **Default Input Delay:** No specific requirement
ri_n	1 bit	I	Ring Indicator Modem Status input **Active State:** Low **Registered:** Yes **Synchronous to:** pclk **Default Input Delay:** No specific requirement
dtr_n	1 bit	O	Modem Control Data Terminal Ready output **Active State:** Low **Registered:** No **Synchronous to:** pclk **Default Output Delay:** No specific requirement
rts_n	1 bit	O	Modem Control Request To Send output **Active State:** Low **Registered:** No **Synchronous to:** pclk **Default Output Delay:** No specific requirement
out2_n	1 bit	O	Modem Control Programmable output 2 **Active State:** Low **Registered:** No **Synchronous to:** pclk **Default Output Delay:** No specific requirement

Table 6: DW_apb_uart Signal Description

Name[a]	Width	I/O	Description
out1_n	1 bit	O	Modem Control Programmable output 1 **Active State:** Low **Registered:** No **Synchronous to:** pclk **Default Output Delay:** No specific requirement
DMA Interface (**Active State:** The following signals are shown as active-low signals. An active-high version of each signal is created when DMA_POL==NO)			
dma_tx_req_n	1 bit	O	Transmit Buffer Ready indicates that the Transmit buffer requires service from the DMA controller **Active State:** Low **Registered:** Yes **Synchronous to:** pclk **Default Output Delay:** 90% of pclk
dma_tx_single_n	1 bit	O	DMA Transmit FIFO Single informs the DMA controller that there is at least one free entry in the Transmit buffer/FIFO. This output does not request a DMA transfer. **Active State:** Low **Registered:** Yes **Synchronous to:** pclk **Default Output Delay:** 90% of pclk **Dependencies:** Present only when DMA_EXTRA==Yes
dma_tx_ack_n	1 bit	I	DMA Transmit Acknowledge indicates that the DMA Controller has transmitted the block of data to the DW_apb_uart for transmission **Active State:** Low **Registered:** No **Synchronous to:** pclk **Default Input Delay:** 50% of pclk **Dependencies:** Present only when DMA_EXTRA==Yes
dma_rx_req_n	1 bit	O	Receive Buffer Ready indicates that the Receive buffer requires service from the DMA controller **Active State:** Low **Registered:** Yes **Synchronous to:** pclk **Default Output Delay:** 90% of pclk

Table 6: DW_apb_uart Signal Description

Name[a]	Width	I/O	Description
dma_rx_single_n	1 bit	O	DMA Receive FIFO Single informs the DMA controller that there is at least one free entry in the Receive buffer/FIFO. This output does not request a DMA transfer. **Active State:** Low **Registered:** Yes **Synchronous to:** pclk **Default Output Delay:** 90% of pclk **Dependencies:** Present only when DMA_EXTRA==Yes
dma_rx_ack_n	1 bit	I	DMA Receive Acknowledge indicates that the DMA Controller has received the block of data from the DW_apb_uart. **Active State:** Low **Registered:** No **Synchronous to:** pclk **Default Input Delay:** 50% of pclk **Dependencies:** Present only when DMA_EXTRA==Yes
txrdy_n	1 bit	O	This transmit buffer read signal is used for backward compatibility of older DW_apb_uart components to indicate that the Transmit buffer requires service from the DMA controller **Active State:** Low **Registered:** Yes **Synchronous to:** pclk **Default Output Delay:** 90% of pclk **Dependencies:** Present only when DMA_EXTRA==No
rxrdy_n	1 bit	O	This receive buffer read signal is used for backward compatibility of older DW_apb_uart components to indicate that the Receive buffer requires service from the DMA controller **Active State:** Low **Registered:** Yes **Synchronous to:** pclk **Default Output Delay:** 90% of pclk **Dependencies:** Present only when DMA_EXTRA==No

Table 6: DW_apb_uart Signal Description

Name[a]	Width	I/O	Description
Serial Interface			
sin	1 bit	I	Serial input **Active State:** High **Registered:** Yes **Synchronous to:** pclk (in single clock configuration) sclk (in two clock configuration) **Default Input Delay:** 85% of pclk or sclk, depending on configuration
sout	1 bit	O	Serial output **Active State:** High **Registered:** Yes **Synchronous to:** pclk (in single clock configuration) sclk (in two clock configuration) **Default Output Delay:** 90% of pclk or sclk, depending on configuration
Infrared Interface (**Dependencies:** The following signals are present only when `SIR_MODE==Enabled`)			
sir_in	1 bit	I	IrDA SIR input **Active State:** High **Registered:** Yes **Synchronous to:** pclk (in single clock configuration) sclk (in two clock configuration) **Default Input Delay:** 85% of pclk or sclk, depending on configuration
sir_out_n	1 bit	O	IrDA SIR output **Active State:** Low **Registered:** Yes **Synchronous to:** pclk (in single clock configuration) sclk (in two clock configuration) **Default Output Delay:** 90% of pclk or sclk, depending on configuration
Interrupt Interface			
intr	1 bit	O	Interrupt **Active State:** High **Registered:** Yes **Synchronous to:** pclk **Default Output Delay:** 80% of pclk

Table 6: DW_apb_uart Signal Description

Name[a]	Width	I/O	Description
Debug Interface			
debug*n*	32 bits	O	On-chip debug signals as follows: debug[31:14] = RAZ debug[13] = RX push indication (RBR or RX FIFO) debug[12] = TX pop indication (THR or TX FIFO) debug[11:10] = receiver trigger (FCR[7:6]) debug[9:8] = TX empty trigger (FCR[5:4]) debug[7] = DMA mode (FCR{[3]) debug[6:1] = individual interrupt sources: debug[6] = line status interrupt debug[5] = data available interrupt debug[4] = character timeout interrupt debug[3] = THRE interrupt debug[2] = modem status interrupt debug[1] = busy detect interrupt debug[0] = FIFO enable (FCR{[0]). **Registered:** No **Synchronous to:** pclk **Output Delay:** No specific requirement **Dependencies:** Present only when `DEBUG==Yes`.
Clock Control Interface			
uart_lp_req_pclk	1 bit	O	pclk domain clock gate signal indicates that the UART is inactive, so clocks may be gated to put the device in a low-power (lp) mode. **Active State:** High **Registered:** Yes **Synchronous to:** pclk **Default Output Delay:** 25% of pclk **Dependencies:** Present only when `CLK_GATE_EN==Include`.
uart_lp_req_sclk	1 bit	O	sclk domain clock gate signal indicates that the UART is inactive, so clocks may be gated to put the device in a low-power (lp) mode. **Active State:** High **Registered:** Yes **Synchronous to:** sclk **Default Output Delay:** 25% of sclk **Dependencies:** Present only when `CLK_GATE_EN==Include` and `CLOCK_MODE==Enabled`.

Table 6: DW_apb_uart Signal Description

Name[a]	Width	I/O	Description
baudout_n	1 bit	O	Transmit clock output **Active State:** Low **Registered:** No **Synchronous to:** pclk (in single clock configuration) sclk (in two clock configuration) **Default Output Delay:** 5% of pclk or sclk, depending on configuration **Dependencies:** Present only when `BAUD_CLK==Yes`.

a. An *n* on signal names denotes multiple signals with suffixes of (Width minus 1) down-to 0.

6 Registers

This chapter describes the programmable registers of the DW_apb_uart.

Register Memory Map

The DW_apb_uart has a number of internal registers that are accessed through the 5-bit address bus.

Note

Since DW_apb_uart registers are only located 32-bit boundaries, paddr[1:0] may be tied low permanently, if so desired. This would allow backward compatibility with standard 16550 UART programmability.

The following table summarizes the register memory map for the DW_apb_uart:

Name	Address Offset	Width	R/W	Description
RBR	0x00	32 bits	R	Receive Buffer Register **Reset Value:** 0x0 **Dependencies:** LCR[7] bit = 0
THR		32 bits	W	Transmit Holding Register **Reset Value:** 0x0 **Dependencies:** LCR[7] bit = 0
DLL		32 bits	R/W	Divisor Latch (Low) **Reset Value:** 0x0 **Dependencies:** LCR[7] bit = 1
DLH	0x04	32 bits	R/W	Divisor Latch (High) **Reset Value:** 0x0 **Dependencies:** LCR[7] bit = 1
IER		32 bits	R/W	Interrupt Enable Register **Reset Value:** 0x0 **Dependencies:** LCR[7] bit = 0

Name	Address Offset	Width	R/W	Description
IIR	0x08	32 bits	R	Interrupt Identification Register **Reset Value:** 0x01
FCR		32 bits	W	FIFO Control Register **Reset Value:** 0x0
LCR	0x0C	32 bits	R/W	Line Control Register **Reset Value:** 0x0
MCR	0x10	32 bits	R/W	Modem Control Register **Reset Value:** 0x0
LSR	0x14	32 bits	R	Line Status Register **Reset Value:** 0x60
MSR	0x18	32 bits	R	Modem Status Register **Reset Value:** 0x0
SCR	0x1C	32 bits	R/W	Scratchpad Register **Reset Value:** 0x0
LPDLL	0x20	32 bits	R/W	Low Power Divisor Latch (Low) Register **Reset Value:** 0x0
LPDLH	0x24	32 bits	R/W	Low Power Divisor Latch (High) Register **Reset Value:** 0x0
Reserved	0x28 - 0x2C	–	–	–
SRBR	0x30 - 0x6C	32 bits	R	Shadow Receive Buffer Register **Reset Value:** 0x0 **Dependencies:** LCR[7] bit = 0
STHR		32 bits	W	Shadow Transmit Holding Register **Reset Value:** 0x0 **Dependencies:** LCR[7] bit = 0
FAR	0x70	32 bits	R/W	FIFO Access Register **Reset Value:** 0x0
TFR	0x74	32 bits	R	Transmit FIFO Read **Reset Value:** 0x0
RFW	0x78	32 bits	W	Receive FIFO Write **Reset Value:** 0x0
USR	0x7C	32 bits	R	UART Status Register **Reset Value:** 0x6
TFL	0x80	See Description (page 118)	R	Transmit FIFO Level **Width:** *FIFO_ADDR_WIDTH* + 1 **Reset Value:** 0x0

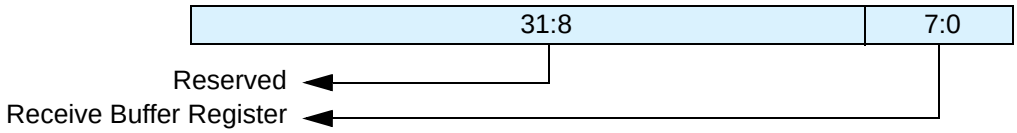
Name	Address Offset	Width	R/W	Description
RFL	0x84	See Description (page 119)	R	Receive FIFO Level **Width:** *FIFO_ADDR_WIDTH* + 1 **Reset Value:** 0x0
SRR	0x88	32 bits	W	Software Reset Register **Reset Value:** 0x0
SRTS	0x8C	32 bits	R/W	Shadow Request to Send **Reset Value:** 0x0
SBCR	0x90	32 bits	R/W	Shadow Break Control Register **Reset Value:** 0x0
SDMAM	0x94	32 bits	R/W	Shadow DMA Mode **Reset Value:** 0x0
SFE	0x98	32 bits	R/W	Shadow FIFO Enable **Reset Value:** 0x0
SRT	0x9C	32 bits	R/W	Shadow RCVR Trigger **Reset Value:** 0x0
STET	0xA0	32 bits	R/W	Shadow TX Empty Trigger **Reset Value:** 0x0
HTX	0xA4	32 bits	R/W	Halt TX **Reset Value:** 0x0
DMASA	0xA8	1 bit	W	DMA Software Acknowledge **Reset Value:** 0x0
–	0xAC - 0xF0	–	–	–
CPR	0xF4	32 bits	R	Component Parameter Register **Reset Value:** Configuration-dependent
UCV	0xF8	32 bits	R	UART Component Version **Reset Value:** See the Releases table in the *DW_apb_uart Release Notes*
CTR	0xFC	32 bits	R	Component Type Register **Reset Value:** 0x44570110

Register and Field Descriptions

The following subsections describe the data fields of the DW_apb_uart registers.

RBR

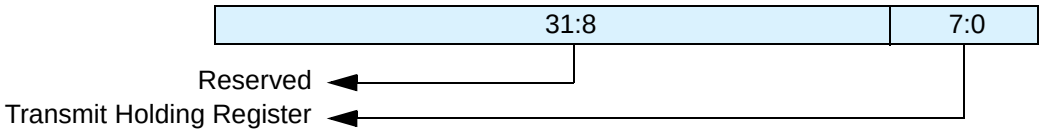
- **Name:** Receive Buffer Register
- **Size:** 32 bits
- **Address Offset:** 0x00
- **Read/write access:** read-only



Bits	Name	R/W	Description
31:8	Reserved and read as zero		
7:0	Receive Buffer Register	R	Data byte received on the serial input port (sin) in UART mode, or the serial infrared input (sir_in) in infrared mode. The data in this register is valid only if the Data Ready (DR) bit in the Line Status Register (LCR) is set. If in non-FIFO mode (FIFO_MODE == NONE) or FIFOs are disabled (FCR[0] set to zero), the data in the RBR must be read before the next data arrives, otherwise it is overwritten, resulting in an over-run error. If in FIFO mode (FIFO_MODE != NONE) and FIFOs are enabled (FCR[0] set to one), this register accesses the head of the receive FIFO. If the receive FIFO is full and this register is not read before the next data character arrives, then the data already in the FIFO is preserved, but any incoming data are lost and an over-run error occurs. **Reset Value:** 0x0

THR

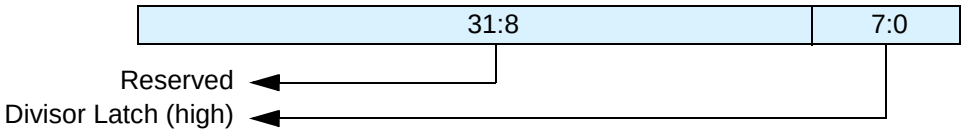
- **Name:** Transmit Holding Register
- **Size:** 32 bits
- **Address Offset:** 0x00
- **Read/write access:** write-only



Bits	Name	R/W	Description
31:8	Reserved and read as zero		
7:0	Transmit Holding Register	W	Data to be transmitted on the serial output port (sout) in UART mode or the serial infrared output (sir_out_n) in infrared mode. Data should only be written to the THR when the THR Empty (THRE) bit (LSR[5]) is set. If in non-FIFO mode or FIFOs are disabled (FCR[0] = 0) and THRE is set, writing a single character to the THR clears the THRE. Any additional writes to the THR before the THRE is set again causes the THR data to be overwritten. If in FIFO mode and FIFOs are enabled (FCR[0] = 1) and THRE is set, x number of characters of data may be written to the THR before the FIFO is full. The number x (default=16) is determined by the value of FIFO Depth that you set during configuration. Any attempt to write data when the FIFO is full results in the write data being lost. **Reset Value:** 0x0

DLH

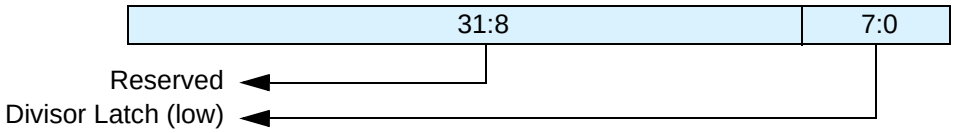
- **Name:** Divisor Latch High
- **Size:** 32 bits
- **Address Offset:** 0x04
- **Read/write access:** read/write



Bits	Name	R/W	Description
31:8	Reserved and read as zero		
7:0	Divisor Latch (High)	R/W	Upper 8-bits of a 16-bit, read/write, Divisor Latch register that contains the baud rate divisor for the UART. This register may only be accessed when the DLAB bit (LCR[7]) is set and the UART is not busy (USR[0] is zero). The output baud rate is equal to the serial clock (pclk if one clock design, sclk if two clock design (CLOCK_MODE == Enabled)) frequency divided by sixteen times the value of the baud rate divisor, as follows: baud rate = (serial clock freq) / (16 * divisor). Note that with the Divisor Latch Registers (DLL and DLH) set to zero, the baud clock is disabled and no serial communications occur. Also, once the DLH is set, at least 8 clock cycles of the slowest DW_apb_uart clock should be allowed to pass before transmitting or receiving data. **Reset Value:** 0x0

DLL

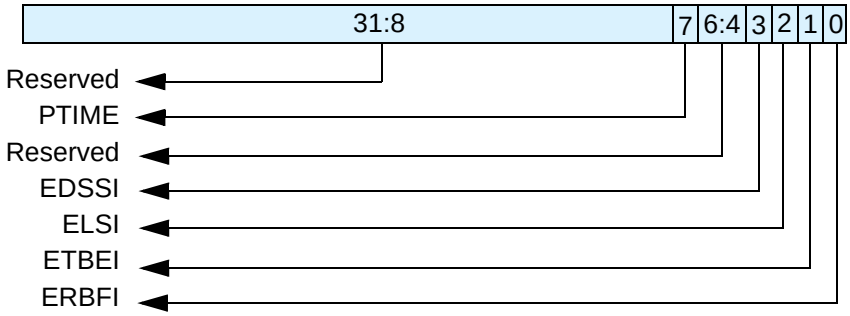
- **Name:** Divisor Latch Low
- **Size:** 32 bits
- **Address Offset:** 0x00
- **Read/write access:** read/write



Bits	Name	R/W	Description
31:8	Reserved and read as zero		
7:0	Divisor Latch (Low)	R/W	Lower 8 bits of a 16-bit, read/write, Divisor Latch register that contains the baud rate divisor for the UART. This register may only be accessed when the DLAB bit (LCR[7]) is set and the UART is not busy (USR[0] is zero). The output baud rate is equal to the serial clock (pclk if one clock design, sclk if two clock design (CLOCK_MODE == Enabled)) frequency divided by sixteen times the value of the baud rate divisor, as follows: baud rate = (serial clock freq) / (16 * divisor). Note that with the Divisor Latch Registers (DLL and DLH) set to zero, the baud clock is disabled and no serial communications occur. Also, once the DLL is set, at least 8 clock cycles of the slowest DW_apb_uart clock should be allowed to pass before transmitting or receiving data. **Reset Value:** 0x0

IER

- **Name:** Interrupt Enable Register
- **Size:** 32 bits
- **Address Offset:** 0x04
- **Read/write access:** read/write



Bits	Name	R/W	Description
31:8	Reserved and read as zero		
7	PTIME	R/W	Programmable THRE Interrupt Mode Enable that can be written to only when THRE_MODE_USER == Enabled, always readable. This is used to enable/disable the generation of THRE Interrupt. 0 = disabled 1 = enabled **Reset Value:** 0x0
6:4	Reserved and read as zero		
3	EDSSI	R/W	Enable Modem Status Interrupt. This is used to enable/disable the generation of Modem Status Interrupt. This is the fourth highest priority interrupt. 0 = disabled 1 = enabled **Reset Value:** 0x0
2	ELSI	R/W	Enable Receiver Line Status Interrupt. This is used to enable/disable the generation of Receiver Line Status Interrupt. This is the highest priority interrupt. 0 = disabled 1 = enabled **Reset Value:** 0x0
1	ETBEI	R/W	Enable Transmit Holding Register Empty Interrupt. This is used to enable/disable the generation of Transmitter Holding Register Empty Interrupt. This is the third highest priority interrupt. 0 = disabled 1 = enabled **Reset Value:** 0x0

Bits	Name	R/W	Description
0	ERBFI	R/W	Enable Received Data Available Interrupt. This is used to enable/disable the generation of Received Data Available Interrupt and the Character Timeout Interrupt (if in FIFO mode and FIFOs enabled). These are the second highest priority interrupts. 0 = disabled 1 = enabled **Reset Value:** 0x0

IIR

- **Name:** Interrupt Identity Register
- **Size:** 32 bits
- **Address Offset:** 0x08
- **Read/write access:** read-only

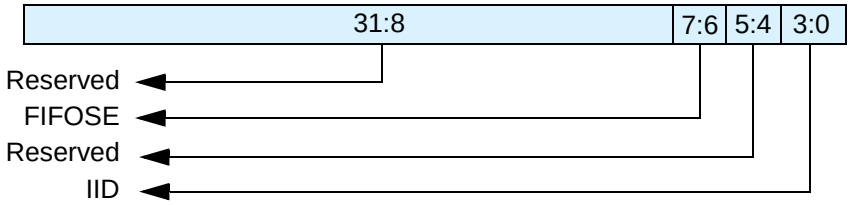


Table 7: Interrupt Identification Register

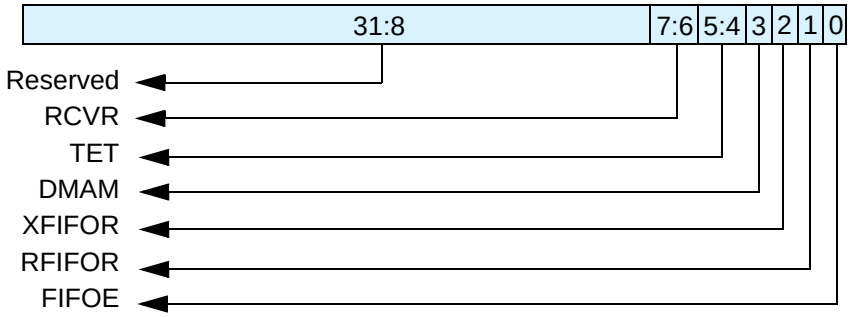
Bits	Name	R/W	Description
31:8	Reserved and read as zero		
7:6	FIFOs Enabled (or FIFOSE)	R	FIFOs Enabled. This is used to indicate whether the FIFOs are enabled or disabled. 00 = disabled 11 = enabled **Reset Value:** 0x00
5:4	Reserved	N/A	Reserved and read as zero
3:0	Interrupt ID (or IID)	R	Interrupt ID. This indicates the highest priority pending interrupt which can be one of the following types: 0000 = modem status 0001 = no interrupt pending 0010 = THR empty 0100 = received data available 0110 = receiver line status 0111 = busy detect 1100 = character timeout The interrupt priorities are split into four levels that are detailed in Table 8 on page 97. Bit 3 indicates an interrupt can only occur when the FIFOs are enabled and used to distinguish a Character Timeout condition interrupt. **Reset Value:** 0x01

Table 8: Interrupt Control Functions

Interrupt ID				Interrupt Set and Reset Functions			
Bit 3	**Bit 2**	**Bit 1**	**Bit 0**	**Priority Level**	**Interrupt Type**	**Interrupt Source**	**Interrupt Reset Control**
0	0	0	1	–	None	None	–
0	1	1	0	Highest	Receiver line status	Overrun/parity/ framing errors or break interrupt	Reading the line status register
0	1	0	0	Second	Received data available	Receiver data available (non-FIFO mode or FIFOs disabled) or RCVR FIFO trigger level reached (FIFO mode and FIFOs enabled)	Reading the receiver buffer register (non-FIFO mode or FIFOs disabled) or the FIFO drops below the trigger level (FIFO mode and FIFOs enabled)
1	1	0	0	Second	Character timeout indication	No characters in or out of the RCVR FIFO during the last 4 character times and there is at least 1 character in it during this time	Reading the receiver buffer register
0	0	1	0	Third	Transmit holding register empty	Transmitter holding register empty (Prog. THRE Mode disabled) or XMIT FIFO at or below threshold (Prog. THRE Mode enabled)	Reading the IIR register (if source of interrupt); or, writing into THR (FIFOs or THRE Mode not selected or disabled) or XMIT FIFO above threshold (FIFOs and THRE Mode selected and enabled).
0	0	0	0	Fourth	Modem status	Clear to send or data set ready or ring indicator or data carrier detect. Note that if auto flow control mode is enabled, a change in CTS (that is, DCTS set) does not cause an interrupt.	Reading the Modem status register
0	1	1	1	Fifth	Busy detect indication	Master has tried to write to the Line Control Register while the DW_apb_uart is busy (USR[0] is set to one).	Reading the UART status register

FCR

- **Name:** FIFO Control Register
- **Size:** 32 bits
- **Address Offset:** 0x08
- **Read/write access:** write-only



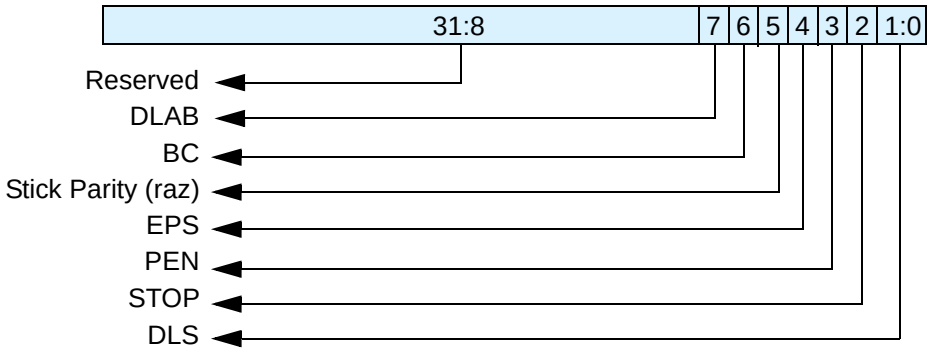
This register is only valid when the DW_apb_uart is configured to have FIFOs implemented (FIFO_MODE != NONE). If FIFOs are not implemented, this register does not exist and writing to this register address has no effect; reading from this register address returns zero.

Bits	Name	R/W	Description
31:8	Reserved and read as zero		
7:6	RCVR Trigger (or RT)	W	RCVR Trigger. This is used to select the trigger level in the receiver FIFO at which the Received Data Available Interrupt is generated. In auto flow control mode it is used to determine when the rts_n signal is de-asserted. It also determines when the dma_rx_req_n signal is asserted in certain modes of operation. For details on DMA support, refer to “DMA Support” on page 58. The following trigger levels are supported: 00 = 1 character in the FIFO 01 = FIFO ¼ full 10 = FIFO ½ full 11 = FIFO 2 less than full **Reset Value:** 0x0
5:4	TX Empty Trigger (or TET)	W	TX Empty Trigger. Writes have no effect when THRE_MODE_USER == Disabled. This is used to select the empty threshold level at which the THRE Interrupts are generated when the mode is active. It also determines when the dma_tx_req_n signal is asserted when in certain modes of operation. For details on DMA support, refer to “DMA Support” on page 58. The following trigger levels are supported: 00 = FIFO empty 01 = 2 characters in the FIFO 10 = FIFO ¼ full 11 = FIFO ½ full **Reset Value:** 0x0

Bits	Name	R/W	Description
3	DMA Mode (or DMAM)	W	DMA Mode. This determines the DMA signalling mode used for the dma_tx_req_n and dma_rx_req_n output signals when additional DMA handshaking signals are not selected (DMA_EXTRA == No). For details on DMA support, refer to “DMA Support” on page 58. 0 = mode 0 1 = mode 1 **Reset Value:** 0x0
2	XMIT FIFO Reset (or XFIFOR)	W	XMIT FIFO Reset. This resets the control portion of the transmit FIFO and treats the FIFO as empty. This also de-asserts the DMA TX request and single signals when additional DMA handshaking signals are selected (DMA_EXTRA == YES). Note that this bit is 'self-clearing'. It is not necessary to clear this bit. **Reset Value:** 0x0
1	RCVR FIFO Reset (or RFIFOR)	W	RCVR FIFO Reset. This resets the control portion of the receive FIFO and treats the FIFO as empty. This also de-asserts the DMA RX request and single signals when additional DMA handshaking signals are selected (DMA_EXTRA == YES). Note that this bit is 'self-clearing'. It is not necessary to clear this bit. **Reset Value:** 0x0
0	FIFO Enable (or FIFOE)	W	FIFO Enable. This enables/disables the transmit (XMIT) and receive (RCVR) FIFOs. Whenever the value of this bit is changed both the XMIT and RCVR controller portion of FIFOs is reset. **Reset Value:** 0x0

LCR

- **Name:** Line Control Register
- **Size:** 32 bits
- **Address Offset:** 0x0C
- **Read/write access:** read/write

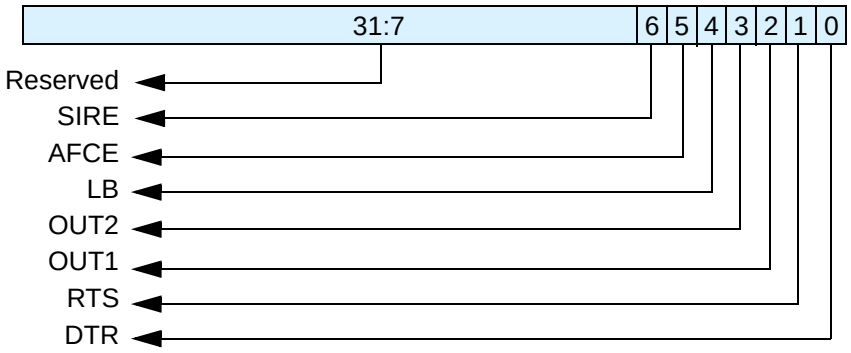


Bits	Name	R/W	Description
31:8	Reserved and read as zero		
7	DLAB	R/W	Divisor Latch Access Bit.Writeable only when UART is not busy (USR[0] is zero), always readable. This bit is used to enable reading and writing of the Divisor Latch register (DLL and DLH) to set the baud rate of the UART. This bit must be cleared after initial baud rate setup in order to access other registers. **Reset Value:** 0x0
6	Break (or BC)	R/W	Break Control Bit.This is used to cause a break condition to be transmitted to the receiving device. If set to one the serial output is forced to the spacing (logic 0) state. When not in Loopback Mode, as determined by MCR[4], the sout line is forced low until the Break bit is cleared. If SIR_MODE == Enabled and active (MCR[6] set to one) the sir_out_n line is continuously pulsed. When in Loopback Mode, the break condition is internally looped back to the receiver and the sir_out_n line is forced low. **Reset Value:** 0x0
5	Stick Parity	Reserved and read as zero	
4	EPS	R/W	Even Parity Select. Writeable only when UART is not busy (USR[0] is zero), always readable. This is used to select between even and odd parity, when parity is enabled (PEN set to one). If set to one, an even number of logic 1s is transmitted or checked. If set to zero, an odd number of logic 1s is transmitted or checked. **Reset Value:** 0x0
3	PEN	R/W	Parity Enable. Writeable only when UART is not busy (USR[0] is zero), always readable. This bit is used to enable and disable parity generation and detection in transmitted and received serial character respectively. 0 = parity disabled 1 = parity enabled **Reset Value:** 0x0

Bits	Name	R/W	Description
2	STOP	R/W	Number of stop bits. Writeable only when UART is not busy (USR[0] is zero), always readable. This is used to select the number of stop bits per character that the peripheral transmits and receives. If set to zero, one stop bit is transmitted in the serial data. If set to one and the data bits are set to 5 (LCR[1:0] set to zero) one and a half stop bits is transmitted. Otherwise, two stop bits are transmitted. Note that regardless of the number of stop bits selected, the receiver checks only the first stop bit. 0 = 1 stop bit 1 = 1.5 stop bits when DLS (LCR[1:0]) is zero, else 2 stop bit **Reset Value:** 0x0
1:0	DLS (or CLS, as used in legacy)	R/W	Data Length Select. Writeable only when UART is not busy (USR[0] is zero), always readable. This is used to select the number of data bits per character that the peripheral transmits and receives. The number of bit that may be selected areas follows: 00 = 5 bits 01 = 6 bits 10 = 7 bits 11 = 8 bits **Reset Value:** 0x0

MCR

- **Name:** Modem Control Register
- **Size:** 32 bits
- **Address Offset:** 0x10
- **Read/write access:** read/write

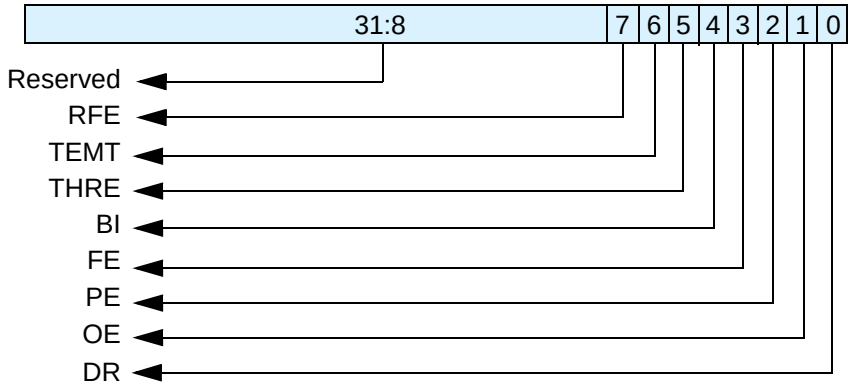


Bits	Name	R/W	Description
31:7	Reserved and read as zero		
6	SIRE	R/W	SIR Mode Enable. Writeable only when SIR_MODE == Enabled, always readable. This is used to enable/disable the IrDA SIR Mode features as described in “IrDA 1.0 SIR Protocol” on page 47. 0 = IrDA SIR Mode disabled 1 = IrDA SIR Mode enabled **Reset Value:** 0x0
5	AFCE	R/W	Auto Flow Control Enable. Writeable only when AFCE_MODE == Enabled, always readable. When FIFOs are enabled and the Auto Flow Control Enable (AFCE) bit is set, Auto Flow Control features are enabled as described in “Auto Flow Control” on page 51. 0 = Auto Flow Control Mode disabled 1 = Auto Flow Control Mode enabled **Reset Value:** 0x0
4	LoopBack (or LB)	R/W	LoopBack Bit. This is used to put the UART into a diagnostic mode for test purposes. If operating in UART mode (SIR_MODE != Enabled or not active, MCR[6] set to zero), data on the sout line is held high, while serial data output is looped back to the sin line, internally. In this mode all the interrupts are fully functional. Also, in loopback mode, the modem control inputs (dsr_n, cts_n, ri_n, dcd_n) are disconnected and the modem control outputs (dtr_n, rts_n, out1_n, out2_n) are looped back to the inputs, internally. If operating in infrared mode (SIR_MODE == Enabled AND active, MCR[6] set to one), data on the sir_out_n line is held low, while serial data output is inverted and looped back to the sir_in line. **Reset Value:** 0x0

Bits	Name	R/W	Description
3	OUT2	R/W	OUT2. This is used to directly control the user-designated Output2 (out2_n) output. The value written to this location is inverted and driven out on out2_n, that is: 0 = out2_n de-asserted (logic 1) 1 = out2_n asserted (logic 0) Note that in Loopback mode (MCR[4] set to one), the out2_n output is held inactive high while the value of this location is internally looped back to an input. **Reset Value:** 0x0
2	OUT1	R/W	OUT1. This is used to directly control the user-designated Output1 (out1_n) output. The value written to this location is inverted and driven out on out1_n, that is: 0 = out1_n de-asserted (logic 1) 1 = out1_n asserted (logic 0) Note that in Loopback mode (MCR[4] set to one), the out1_n output is held inactive high while the value of this location is internally looped back to an input. **Reset Value:** 0x0
1	RTS	R/W	Request to Send. This is used to directly control the Request to Send (rts_n) output. The Request To Send (rts_n) output is used to inform the modem or data set that the UART is ready to exchange data. When Auto RTS Flow Control is not enabled (MCR[5] set to zero), the rts_n signal is set low by programming MCR[1] (RTS) to a high.In Auto Flow Control, AFCE_MODE == Enabled and active (MCR[5] set to one) and FIFOs enable (FCR[0] set to one), the rts_n output is controlled in the same way, but is also gated with the receiver FIFO threshold trigger (rts_n is inactive high when above the threshold). The rts_n signal is de-asserted when MCR[1] is set low. Note that in Loopback mode (MCR[4] set to one), the rts_n output is held inactive high while the value of this location is internally looped back to an input. **Reset Value:** 0x0
0	DTR	R/W	Data Terminal Ready. This is used to directly control the Data Terminal Ready (dtr_n) output. The value written to this location is inverted and driven out on dtr_n, that is: 0 = dtr_n de-asserted (logic 1) 1 = dtr_n asserted (logic 0) The Data Terminal Ready output is used to inform the modem or data set that the UART is ready to establish communications. Note that in Loopback mode (MCR[4] set to one), the dtr_n output is held inactive high while the value of this location is internally looped back to an input. **Reset Value:** 0x0

LSR

- **Name:** Line Status Register
- **Size:** 32 bits
- **Address Offset:** 0x14
- **Read/write access:** read-only



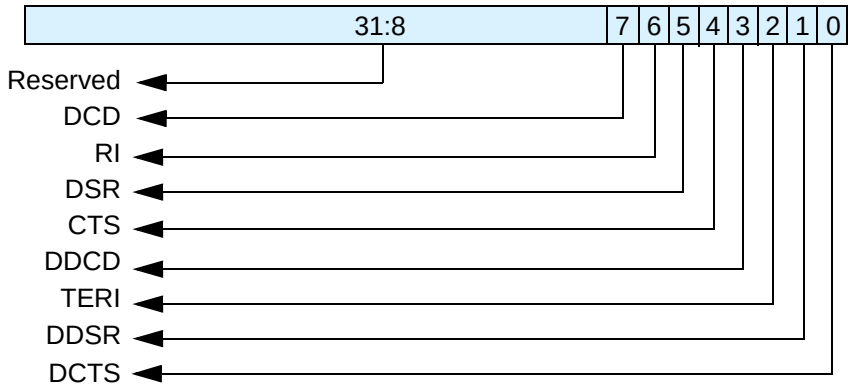
Bits	Name	R/W	Description
31:8	Reserved and read as zero		
7	RFE	R	Receiver FIFO Error bit. This bit is only relevant when FIFO_MODE != NONE AND FIFOs are enabled (FCR[0] set to one). This is used to indicate if there is at least one parity error, framing error, or break indication in the FIFO. 0 = no error in RX FIFO 1 = error in RX FIFO This bit is cleared when the LSR is read and the character with the error is at the top of the receiver FIFO and there are no subsequent errors in the FIFO. **Reset Value:** 0x0
6	TEMT	R	Transmitter Empty bit. If in FIFO mode (FIFO_MODE != NONE) and FIFOs enabled (FCR[0] set to one), this bit is set whenever the Transmitter Shift Register and the FIFO are both empty. If in non-FIFO mode or FIFOs are disabled, this bit is set whenever the Transmitter Holding Register and the Transmitter Shift Register are both empty. **Reset Value:** 0x1
5	THRE	R	Transmit Holding Register Empty bit. If THRE_MODE_USER == Disabled or THRE mode is disabled (IER[7] set to zero) and regardless of FIFO's being implemented/enabled or not, this bit indicates that the THR or TX FIFO is empty. This bit is set whenever data is transferred from the THR or TX FIFO to the transmitter shift register and no new data has been written to the THR or TX FIFO. This also causes a THRE Interrupt to occur, if the THRE Interrupt is enabled. If THRE_MODE_USER == Enabled AND FIFO_MODE != NONE and both modes are active (IER[7] set to one and FCR[0] set to one respectively), the functionality is switched to indicate the transmitter FIFO is full, and no longer controls THRE interrupts, which are then controlled by the FCR[5:4] threshold setting. For more details, see “Programmable THRE Interrupt” on page 54. **Reset Value:** 0x1

Bits	Name	R/W	Description
4	BI	R	Break Interrupt bit. This is used to indicate the detection of a break sequence on the serial input data. If in UART mode (SIR_MODE == Disabled), it is set whenever the serial input, sin, is held in a logic '0' state for longer than the sum of *start time + data bits + parity + stop bits.* If in infrared mode (SIR_MODE == Enabled), it is set whenever the serial input, sir_in, is continuously pulsed to logic '0' for longer than the sum of *start time + data bits + parity + stop bits*. A break condition on serial input causes one and only one character, consisting of all zeros, to be received by the UART. In the FIFO mode, the character associated with the break condition is carried through the FIFO and is revealed when the character is at the top of the FIFO. Reading the LSR clears the BI bit. In the non-FIFO mode, the BI indication occurs immediately and persists until the LSR is read. **Reset Value:** 0x0
3	FE	R	Framing Error bit. This is used to indicate the occurrence of a framing error in the receiver. A framing error occurs when the receiver does not detect a valid STOP bit in the received data. In the FIFO mode, since the framing error is associated with a character received, it is revealed when the character with the framing error is at the top of the FIFO. When a framing error occurs, the UART tries to resynchronize. It does this by assuming that the error was due to the start bit of the next character and then continues receiving the other bit i.e. data, and/or parity and stop. It should be noted that the Framing Error (FE) bit (LSR[3]) is set if a break interrupt has occurred, as indicated by Break Interrupt (BI) bit (LSR[4]). 0 = no framing error 1 = framing error Reading the LSR clears the FE bit. **Reset Value:** 0x0
2	PE	R	Parity Error bit. This is used to indicate the occurrence of a parity error in the receiver if the Parity Enable (PEN) bit (LCR[3]) is set. In the FIFO mode, since the parity error is associated with a character received, it is revealed when the character with the parity error arrives at the top of the FIFO. It should be noted that the Parity Error (PE) bit (LSR[2]) is set if a break interrupt has occurred, as indicated by Break Interrupt (BI) bit (LSR[4]). 0 = no parity error 1 = parity error Reading the LSR clears the PE bit. **Reset Value:** 0x0

Bits	Name	R/W	Description
1	OE	R	Overrun error bit. This is used to indicate the occurrence of an overrun error. This occurs if a new data character was received before the previous data was read. In the non-FIFO mode, the OE bit is set when a new character arrives in the receiver before the previous character was read from the RBR. When this happens, the data in the RBR is overwritten. In the FIFO mode, an overrun error occurs when the FIFO is full and a new character arrives at the receiver. The data in the FIFO is retained and the data in the receive shift register is lost. 0 = no overrun error 1 = overrun error Reading the LSR clears the OE bit. **Reset Value:** 0x0
0	DR	R	Data Ready bit. This is used to indicate that the receiver contains at least one character in the RBR or the receiver FIFO. 0 = no data ready 1 = data ready This bit is cleared when the RBR is read in non-FIFO mode, or when the receiver FIFO is empty, in FIFO mode. **Reset Value:** 0x0

MSR

- **Name:** Modem Status Register
- **Size:** 32 bits
- **Address Offset:** 0x18
- **Read/write access:** read-only



Whenever bits 0, 1, 2 or 3 are set to logic one, to indicate a change on the modem control inputs, a modem status interrupt is generated if enabled through the IER, regardless of when the change occurred. Since the delta bits (bits 0, 1, 3) can get set after a reset if their respective modem signals are active (see individual bits for details), a read of the MSR after reset can be performed to prevent unwanted interrupts.

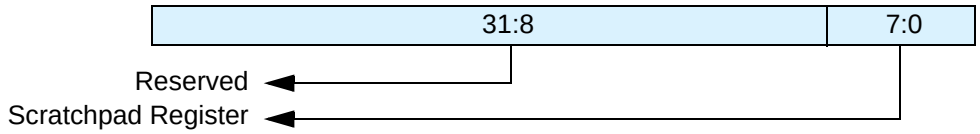
Bits	Name	R/W	Description
31:8	Reserved and read as zero		
7	DCD	R	Data Carrier Detect. This is used to indicate the current state of the modem control line dcd_n. This bit is the complement of dcd_n. When the Data Carrier Detect input (dcd_n) is asserted it is an indication that the carrier has been detected by the modem or data set. 0 = dcd_n input is de-asserted (logic 1) 1 = dcd_n input is asserted (logic 0) In Loopback Mode (MCR[4] set to one), DCD is the same as MCR[3] (Out2). **Reset Value:** 0x0
6	RI	R	Ring Indicator. This is used to indicate the current state of the modem control line ri_n. This bit is the complement of ri_n. When the Ring Indicator input (ri_n) is asserted it is an indication that a telephone ringing signal has been received by the modem or data set. 0 = ri_n input is de-asserted (logic 1) 1 = ri_n input is asserted (logic 0) In Loopback Mode (MCR[4] set to one), RI is the same as MCR[2] (Out1). **Reset Value:** 0x0

Bits	Name	R/W	Description
5	DSR	R	Data Set Ready. This is used to indicate the current state of the modem control line dsr_n. This bit is the complement of dsr_n. When the Data Set Ready input (dsr_n) is asserted it is an indication that the modem or data set is ready to establish communications with the DW_apb_uart. 0 = dsr_n input is de-asserted (logic 1) 1 = dsr_n input is asserted (logic 0) In Loopback Mode (MCR[4] set to one), DSR is the same as MCR[0] (DTR). **Reset Value:** 0x0
4	CTS	R	Clear to Send. This is used to indicate the current state of the modem control line cts_n. This bit is the complement of cts_n. When the Clear to Send input (cts_n) is asserted it is an indication that the modem or data set is ready to exchange data with the DW_apb_uart. 0 = cts_n input is de-asserted (logic 1) 1 = cts_n input is asserted (logic 0) In Loopback Mode (MCR[4] = 1), CTS is the same as MCR[1] (RTS). **Reset Value:** 0x0
3	DDCD	R	Delta Data Carrier Detect. This is used to indicate that the modem control line dcd_n has changed since the last time the MSR was read. 0 = no change on dcd_n since last read of MSR 1 = change on dcd_n since last read of MSR Reading the MSR clears the DDCD bit. In Loopback Mode (MCR[4] = 1), DDCD reflects changes on MCR[3] (Out2). Note, if the DDCD bit is not set and the dcd_n signal is asserted (low) and a reset occurs (software or otherwise), then the DDCD bit is set when the reset is removed if the dcd_n signal remains asserted. **Reset Value:** 0x0
2	TERI	R	Trailing Edge of Ring Indicator. This is used to indicate that a change on the input ri_n (from an active-low to an inactive-high state) has occurred since the last time the MSR was read. 0 = no change on ri_n since last read of MSR 1 = change on ri_n since last read of MSR Reading the MSR clears the TERI bit. In Loopback Mode (MCR[4] = 1), TERI reflects when MCR[2] (Out1) has changed state from a high to a low. **Reset Value:** 0x0
1	DDSR	R	Delta Data Set Ready. This is used to indicate that the modem control line dsr_n has changed since the last time the MSR was read. 0 = no change on dsr_n since last read of MSR 1 = change on dsr_n since last read of MSR Reading the MSR clears the DDSR bit. In Loopback Mode (MCR[4] = 1), DDSR reflects changes on MCR[0] (DTR). Note, if the DDSR bit is not set and the dsr_n signal is asserted (low) and a reset occurs (software or otherwise), then the DDSR bit is set when the reset is removed if the dsr_n signal remains asserted. **Reset Value:** 0x0

Bits	Name	R/W	Description
0	DCTS	R	Delta Clear to Send. This is used to indicate that the modem control line cts_n has changed since the last time the MSR was read. 0 = no change on ctsdsr_n since last read of MSR 1 = change on ctsdsr_n since last read of MSR Reading the MSR clears the DCTS bit. In Loopback Mode (MCR[4] = 1), DCTS reflects changes on MCR[1] (RTS). Note, if the DCTS bit is not set and the cts_n signal is asserted (low) and a reset occurs (software or otherwise), then the DCTS bit is set when the reset is removed if the cts_n signal remains asserted. **Reset Value:** 0x0

SCR

- **Name:** Scratchpad Register
- **Size:** 32 bits
- **Address Offset:** 0x1C
- **Read/write access:** read/write

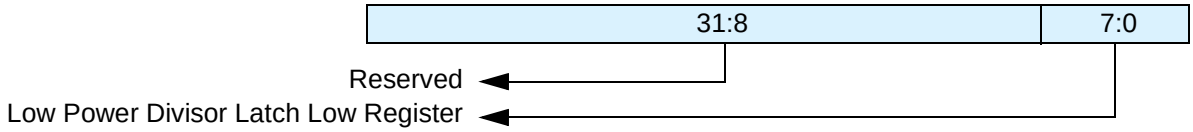


Bits	Name	R/W	Description
31:8	Reserved and read as zero		
7:0	Scratchpad Register	R/W	This register is for programmers to use as a temporary storage space. It has no defined purpose in the DW_apb_uart. **Reset Value:** 0x0

LPDLL

- **Name:** Low Power Divisor Latch Low Register
- **Size:** 32 bits
- **Address Offset:** 0x1C
- **Read/write access:** read/write

This register is only valid when the DW_apb_uart is configured to have SIR low-power reception capabilities implemented (SIR_LP_RX = Yes). If SIR low-power reception capabilities are not implemented, this register does not exist and reading from thsi register address returns zero.

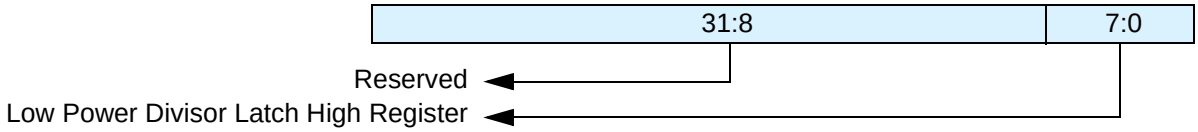


Bits	Name	R/W	Description
31:8	Reserved and read as zero		
7:0	LPDLL	R/W	This register makes up the lower 8-bits of a 16-bit, read/write, Low Power Divisor Latch register that contains the baud rate divisor for the UART, which must give a baud rate of 115.2K. This is required for SIR Low Power (minimum pulse width) detection at the receiver. This register may only be accessed when the DLAB bit (LCR[7]) is set and the UART is not busy (USR[0]) is 0). The output low-power baud rate is equal to the serial clock (sclk) frequency divided by sixteen times the value of the baud rate divisor, as follows: Low power baud rate = (serial clock frequency)/(16* divisor) Therefore, a divisor must be selected to give a baud rate of 115.2K. **NOTE:** When the Low Power Divisor Latch registers (LPDLL and LPDLH) are set to 0, the low-power baud clock is disabled and no low-power pulse detection (or any pulse detection) occurs at the receiver. Also, once the LPDLL is set, at least eight clock cycles of the slowest DW_apb_uart clock should be allowed to pass before transmitting or receiving data. **Reset Value:** 0x0

LPDLH

- **Name:** Low Power Divisor Latch High Register
- **Size:** 32 bits
- **Address Offset:** 0x1C
- **Read/write access:** read/write

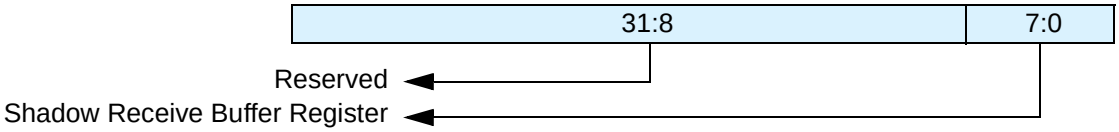
This register is only valid when the DW_apb_uart is configured to have SIR low-power reception capabilities implemented (SIR_LP_RX = Yes). If SIR low-power reception capabilities are not implemented, this register does not exist and reading from thsi register address returns zero.



Bits	Name	R/W	Description
31:8	Reserved and read as zero		
7:0	LPDLH	R/W	This register makes up the upper 8-bits of a 16-bit, read/write, Low Power Divisor Latch register that contains the baud rate divisor for the UART, which must give a baud rate of 115.2K. This is required for SIR Low Power (minimum pulse width) detection at the receiver. This register may only be accessed when the DLAB bit (LCR[7]) is set and the UART is not busy (USR[0]) is 0). The output low-power baud rate is equal to the serial clock (sclk) frequency divided by sixteen times the value of the baud rate divisor, as follows: Low power baud rate = (serial clock frequency)/(16* divisor) Therefore, a divisor must be selected to give a baud rate of 115.2K. **NOTE:** When the Low Power Divisor Latch registers (LPDLL and LPDLH) are set to 0, the low-power baud clock is disabled and no low-power pulse detection (or any pulse detection) occurs at the receiver. Also, once the LPDLH is set, at least eight clock cycles of the slowest DW_apb_uart clock should be allowed to pass before transmitting or receiving data. **Reset Value:** 0x0

SRBR

- **Name:** Shadow Receive Buffer Register
- **Size:** 32 bits
- **Address Offset:** 0x30 - 0x6C
- **Read/write access:** read-only

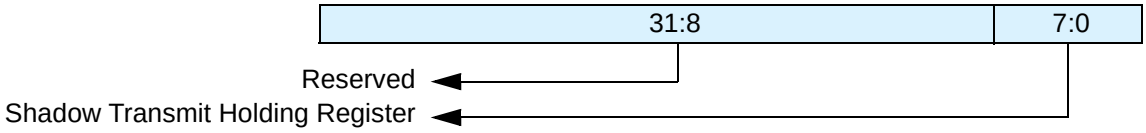


This register is only valid when the DW_apb_uart is configured to have additional shadow registers implemented (SHADOW == YES). If shadow registers are not implemented, this register does not exist and reading from this register address returns zero.

Bits	Name	R/W	Description
31:8	Reserved and read as zero		
7:0	Shadow Receive Buffer Register	R	This is a shadow register for the RBR and has been allocated sixteen 32-bit locations so as to accommodate burst accesses from the master. This register contains the data byte received on the serial input port (sin) in UART mode or the serial infrared input (sir_in) in infrared mode. The data in this register is valid only if the Data Ready (DR) bit in the Line status Register (LSR) is set. If in non-FIFO mode (FIFO_MODE == NONE) or FIFOs are disabled (FCR[0] set to zero), the data in the RBR must be read before the next data arrives, otherwise it is overwritten, resulting in an overrun error. If in FIFO mode (FIFO_MODE != NONE) and FIFOs are enabled (FCR[0] set to one), this register accesses the head of the receive FIFO. If the receive FIFO is full and this register is not read before the next data character arrives, then the data already in the FIFO are preserved, but any incoming data is lost. An overrun error also occurs. **Reset Value:** 0x0

STHR

- **Name:** Shadow Transmit Holding Register
- **Size:** 32 bits
- **Address Offset:** 0x30 - 0x6C
- **Read/write access:** write

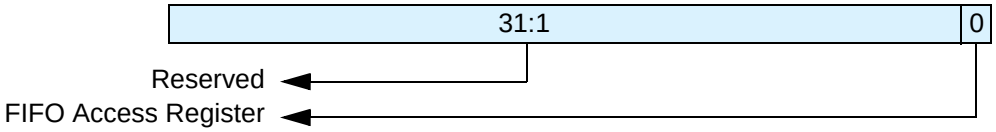


This register is only valid when the DW_apb_uart is configured to have additional shadow registers implemented (SHADOW == YES). If shadow registers are not implemented, this register does not exist, and reading from this register address returns zero.

Bits	Name	R/W	Description
31:8	Reserved and read as zero		
7:0	Shadow Transmit Holding Register	W	This is a shadow register for the THR and has been allocated sixteen 32-bit locations so as to accommodate burst accesses from the master. This register contains data to be transmitted on the serial output port (sout) in UART mode or the serial infrared output (sir_out_n) in infrared mode. Data should only be written to the THR when the THR Empty (THRE) bit (LSR[5]) is set. If in non-FIFO mode or FIFOs are disabled (FCR[0] set to zero) and THRE is set, writing a single character to the THR clears the THRE. Any additional writes to the THR before the THRE is set again causes the THR data to be overwritten. If in FIFO mode and FIFOs are enabled (FCR[0] set to one) and THRE is set, x number of characters of data may be written to the THR before the FIFO is full. The number x (default=16) is determined by the value of FIFO Depth that you set during configuration. Any attempt to write data when the FIFO is full results in the write data being lost. **Reset Value:** 0x0

FAR

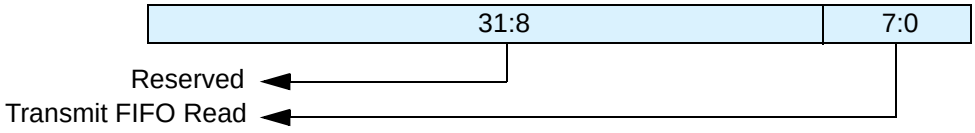
- **Name:** FIFO Access Register
- **Size:** 32 bits
- **Address Offset:** 0x70
- **Read/write access:** read/write



Bits	Name	R/W	Description
31:1	Reserved and read as zero		
0	FIFO Access Register	R/W	Writes have no effect when FIFO_ACCESS == No, always readable. This register is use to enable a FIFO access mode for testing, so that the receive FIFO can be written by the master and the transmit FIFO can be read by the master when FIFOs are implemented and enabled. When FIFOs are not implemented or not enabled it allows the RBR to be written by the master and the THR to be read by the master. 0 = FIFO access mode disabled 1 = FIFO access mode enabled Note, that when the FIFO access mode is enabled/disabled, the control portion of the receive FIFO and transmit FIFO is reset and the FIFOs are treated as empty. **Reset Value:** 0x0

TFR

- **Name:** Transmit FIFO Read
- **Size:** 32 bits
- **Address Offset:** 0x74
- **Read/write access:** read-only

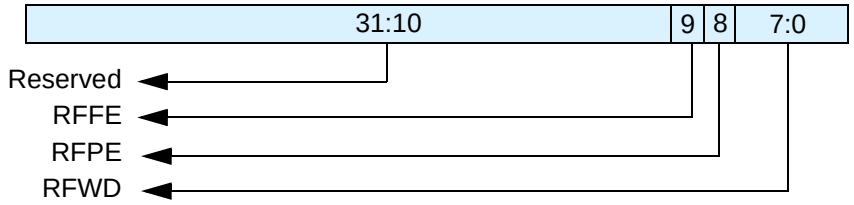


This register is only valid when the DW_apb_uart is configured to have the FIFO access test mode available (FIFO_ACCESS == YES). If not configured, this register does not exist and reading from this register address returns zero.

Bits	Name	R/W	Description
31:8	Reserved and read as zero		
7:0	Transmit FIFO Read	R	Transmit FIFO Read. These bits are only valid when FIFO access mode is enabled (FAR[0] is set to one). When FIFOs are implemented and enabled, reading this register gives the data at the top of the transmit FIFO. Each consecutive read pops the transmit FIFO and gives the next data value that is currently at the top of the FIFO. When FIFOs are not implemented or not enabled, reading this register gives the data in the THR. **Reset Value:** 0x0

RFW

- **Name:** Receive FIFO Write
- **Size:** 32 bits
- **Address Offset:** 0x78
- **Read/write access:** write-only

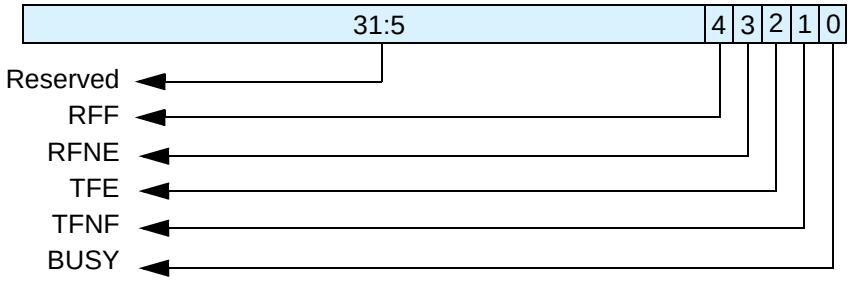


This register is only valid when the DW_apb_uart is configured to have the FIFO access test mode available (FIFO_ACCESS == YES). If not configured, this register does not exist and reading from this register address returns zero.

Bits	Name	R/W	Description
31:10	Reserved and read as zero		
9	RFFE	W	Receive FIFO Framing Error. These bits are only valid when FIFO access mode is enabled (FAR[0] is set to one). When FIFOs are implemented and enabled, this bit is used to write framing error detection information to the receive FIFO. When FIFOs are not implemented or not enabled, this bit is used to write framing error detection information to the RBR. **Reset Value:** 0x0
8	RFPE	W	Receive FIFO Parity Error. These bits are only valid when FIFO access mode is enabled (FAR[0] is set to one). When FIFOs are implemented and enabled, this bit is used to write parity error detection information to the receive FIFO. When FIFOs are not implemented or not enabled, this bit is used to write parity error detection information to the RBR. **Reset Value:** 0x0
7:0	RFWD	W	Receive FIFO Write Data. These bits are only valid when FIFO access mode is enabled (FAR[0] is set to one). When FIFOs are implemented and enabled, the data that is written to the RFWD is pushed into the receive FIFO. Each consecutive write pushes the new data to the next write location in the receive FIFO. When FIFOs are not implemented or not enabled, the data that is written to the RFWD is pushed into the RBR. **Reset Value:** 0x0

USR

- **Name:** UART Status Register
- **Size:** 32 bits
- **Address Offset:** 0x7C
- **Read/write access:** read-only

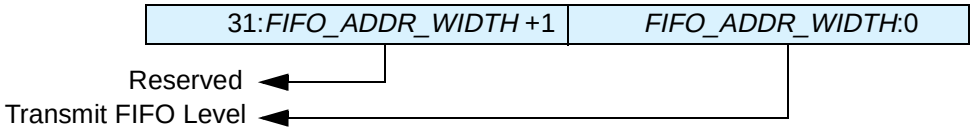


Bits	Name	R/W	Description
31:5	Reserved and read as zero		
4	RFF	R	Receive FIFO Full. This bit is only valid when FIFO_STAT == YES. This is used to indicate that the receive FIFO is completely full. 0 = Receive FIFO not full 1 = Receive FIFO Full This bit is cleared when the RX FIFO is no longer full. **Reset Value:** 0x0
3	RFNE	R	Receive FIFO Not Empty. This bit is only valid when FIFO_STAT == YES. This is used to indicate that the receive FIFO contains one or more entries. 0 = Receive FIFO is empty 1 = Receive FIFO is not empty This bit is cleared when the RX FIFO is empty. **Reset Value:** 0x0
2	TFE	R	Transmit FIFO Empty. This bit is only valid when FIFO_STAT == YES. This is used to indicate that the transmit FIFO is completely empty. 0 = Transmit FIFO is not empty 1 = Transmit FIFO is empty This bit is cleared when the TX FIFO is no longer empty. **Reset Value:** 0x1
1	TFNF	R	Transmit FIFO Not Full. This bit is only valid when FIFO_STAT == YES. This is used to indicate that the transmit FIFO in not full. 0 = Transmit FIFO is full 1 = Transmit FIFO is not full This bit is cleared when the TX FIFO is full. **Reset Value:** 0x1

Bits	Name	R/W	Description
0	BUSY	R	UART Busy. This is indicates that a serial transfer is in progress, when cleared indicates that the DW_apb_uart is idle or inactive. 0 = DW_apb_uart is idle or inactive 1 = DW_apb_uart is busy (actively transferring data) **NOTE:** It is possible for the UART Busy bit to be cleared even though a new character may have been sent from another device. That is, if the DW_apb_uart has no data in THR and RBR and there is no transmission in progress and a start bit of a new character has just reached the DW_apb_uart. This is due to the fact that a valid start is not seen until the middle of the bit period and this duration is dependent on the baud divisor that has been programmed. If a second system clock has been implemented (CLOCK_MODE == Enabled), the assertion of this bit is also delayed by several cycles of the slower clock. **Reset Value:** 0x0

TFL

- **Name:** Transmit FIFO Level
- **Size:** *FIFO_ADDR_WIDTH* + 1
- **Address Offset:** 0x80
- **Read/write access:** read-only

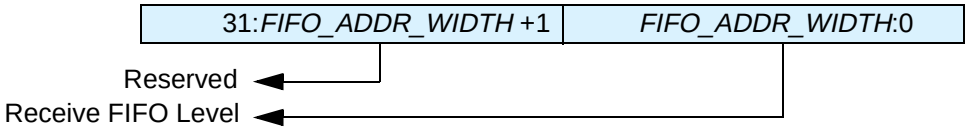


This register is only valid when the DW_apb_uart is configured to have additional FIFO status registers implemented (FIFO_STAT == YES). If status registers are not implemented, this register does not exist and reading from this register address returns zero.

Bits	Name	R/W	Description
31:*FIFO_ADDR_WIDTH* + 1	Reserved and read as zero		
FIFO_ADDR_WIDTH:0	Transmit FIFO Level	R	Transmit FIFO Level. This is indicates the number of data entries in the transmit FIFO. **Reset Value:** 0x0

RFL

- **Name:** Receive FIFO Level
- **Size:** *FIFO_ADDR_WIDTH* + 1
- **Address Offset:** 0x84
- **Read/write access:** read-only

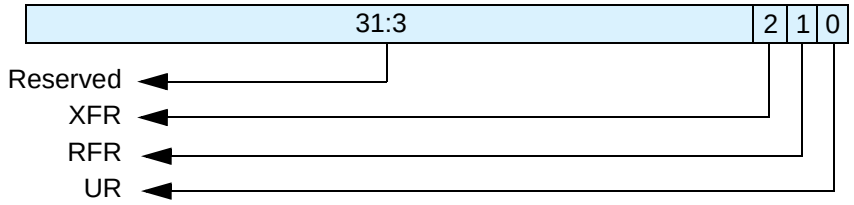


This register is only valid when the DW_apb_uart is configured to have additional FIFO status registers implemented (FIFO_STAT == YES). If status registers are not implemented, this register does not exist and reading from this register address returns zero.

Bits	Name	R/W	Description
31:*FIFO_ADDR_WIDTH* + 1	Reserved and read as zero		
FIFO_ADDR_WIDTH:0	Receive FIFO Level	R	Receive FIFO Level. This is indicates the number of data entries in the receive FIFO. **Reset Value:** 0x0

SRR

- **Name:** Software Reset Register
- **Size:** 32 bits
- **Address Offset:** 0x88
- **Read/write access:** write-only

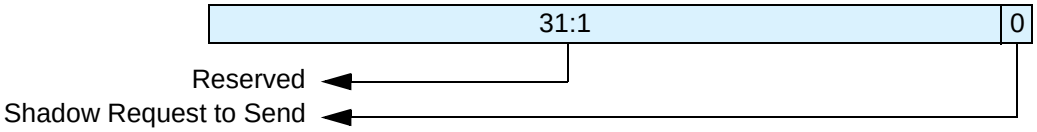


This register is only valid when the DW_apb_uart is configured to have additional shadow registers implemented (SHADOW == YES). If shadow registers are not implemented, this register does not exist and reading from this register address returns zero.

Bits	Name	R/W	Description
31:3	Reserved and read as zero		
2	XFR	W	XMIT FIFO Reset. This is a shadow register for the XMIT FIFO Reset bit (FCR[2]). This can be used to remove the burden on software having to store previously written FCR values (which are pretty static) just to reset the transmit FIFO. This resets the control portion of the transmit FIFO and treats the FIFO as empty. This also de-asserts the DMA TX request and single signals when additional DMA handshaking signals are selected (DMA_EXTRA == YES). Note that this bit is 'self-clearing'. It is not necessary to clear this bit. **Reset Value:** 0x0 **Dependencies:** Writes have no effect when FIFO_MODE == None.
1	RFR	W	RCVR FIFO Reset. This is a shadow register for the RCVR FIFO Reset bit (FCR[1]). This can be used to remove the burden on software having to store previously written FCR values (which are pretty static) just to reset the receive FIFO This resets the control portion of the receive FIFO and treats the FIFO as empty. This also de-asserts the DMA RX request and single signals when additional DMA handshaking signals are selected (DMA_EXTRA == YES). Note that this bit is 'self-clearing'. It is not necessary to clear this bit. **Reset Value:** 0x0 **Dependencies:** Writes have no effect when FIFO_MODE == None.
0	UR	W	UART Reset. This asynchronously resets the DW_apb_uart and synchronously removes the reset assertion. For a two clock implementation both pclk and sclk domains are reset. **Reset Value:** 0x0

SRTS

- **Name:** Shadow Request to Send
- **Size:** 32 bits
- **Address Offset:** 0x8C
- **Read/write access:** read/write

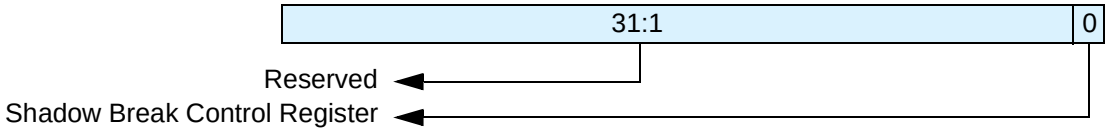


This register is only valid when the DW_apb_uart is configured to have additional shadow registers implemented (SHADOW == YES). If shadow registers are not implemented, this register does not exist and reading from this register address returns zero.

Bits	Name	R/W	Description
31:1	Reserved and read as zero		
0	Shadow Request to Send	R/W	Shadow Request to Send. This is a shadow register for the RTS bit (MCR[1]), this can be used to remove the burden of having to performing a read-modify-write on the MCR. This is used to directly control the Request to Send (rts_n) output. The Request To Send (rts_n) output is used to inform the modem or data set that the DW_apb_uart is ready to exchange data. When Auto RTS Flow Control is not enabled (MCR[5] = 0), the rts_n signal is set low by programming MCR[1] (RTS) to a high. In Auto Flow Control, AFCE_MODE == Enabled and active (MCR[5] = 1) and FIFOs enable (FCR[0] = 1), the rts_n output is controlled in the same way, but is also gated with the receiver FIFO threshold trigger (rts_n is inactive high when above the threshold). Note that in Loopback mode (MCR[4] = 1), the rts_n output is held inactive-high while the value of this location is internally looped back to an input. **Reset Value:** 0x0

SBCR

- **Name:** Shadow Break Control Register
- **Size:** 32 bits
- **Address Offset:** 0x90
- **Read/write access:** read/write

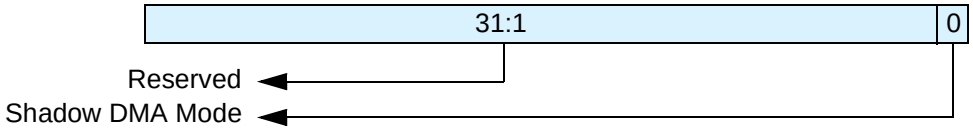


This register is only valid when the DW_apb_uart is configured to have additional shadow registers implemented (SHADOW == YES). If shadow registers are not implemented, this register does not exist and reading from this register address returns zero.

Bits	Name	R/W	Description
31:1	Reserved and read as zero		
0	Shadow Break Control Register	R/W	Shadow Break Control Bit. This is a shadow register for the Break bit (LCR[6]), this can be used to remove the burden of having to performing a read modify write on the LCR. This is used to cause a break condition to be transmitted to the receiving device. If set to one the serial output is forced to the spacing (logic 0) state. When not in Loopback Mode, as determined by MCR[4], the sout line is forced low until the Break bit is cleared. If SIR_MODE == Enabled and active (MCR[6] = 1) the sir_out_n line is continuously pulsed. When in Loopback Mode, the break condition is internally looped back to the receiver. **Reset Value:** 0x0

SDMAM

- **Name:** Shadow DMA Mode
- **Size:** 32 bits
- **Address Offset:** 0x94
- **Read/write access:** read/write

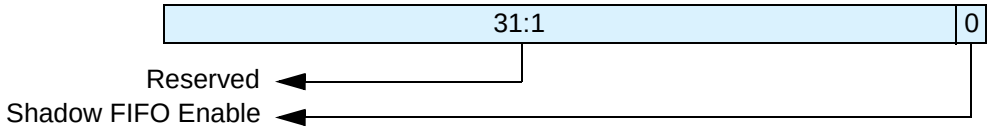


This register is only valid when the DW_apb_uart is configured to have additional FIFO registers implemented (FIFO_MODE != None) and additional shadow registers implemented (SHADOW == YES). If these registers are not implemented, this register does not exist and reading from this register address returns zero.

Bits	Name	R/W	Description
31:1	Reserved and read as zero		
0	Shadow DMA Mode	R/W	Shadow DMA Mode. This is a shadow register for the DMA mode bit (FCR[3]). This can be used to remove the burden of having to store the previously written value to the FCR in memory and having to mask this value so that only the DMA Mode bit gets updated. This determines the DMA signalling mode used for the dma_tx_req_n and dma_rx_req_n output signals when additional DMA handshaking signals are not selected (DMA_EXTRA == NO). 0 = mode 0 1 = mode 1 **Reset Value:** 0x0

SFE

- **Name:** Shadow FIFO Enable
- **Size:** 32 bits
- **Address Offset:** 0x98
- **Read/write access:** read/write

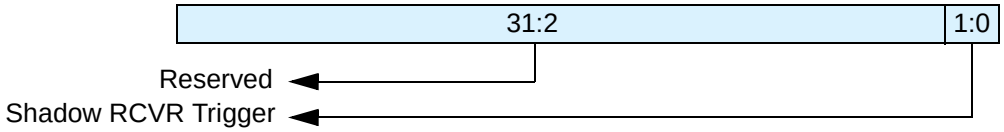


This register is only valid when the DW_apb_uart is configured to have additional FIFO registers implemented (FIFO_MODE != None) and additional shadow registers implemented (SHADOW == YES). If these registers are not implemented, this register does not exist and reading from this register address returns zero.

Bits	Name	R/W	Description
31:1	Reserved and read as zero		
0	Shadow FIFO Enable	R/W	Shadow FIFO Enable. This is a shadow register for the FIFO enable bit (FCR[0]). This can be used to remove the burden of having to store the previously written value to the FCR in memory and having to mask this value so that only the FIFO enable bit gets updated.This enables/disables the transmit (XMIT) and receive (RCVR) FIFOs. If this bit is set to zero (disabled) after being enabled then both the XMIT and RCVR controller portion of FIFOs are reset. **Reset Value:** 0x0

SRT

- **Name:** Shadow RCVR Trigger
- **Size:** 32 bits
- **Address Offset:** 0x9C
- **Read/write access:** read/write

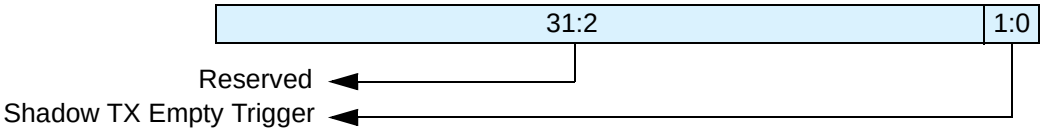


This register is only valid when the DW_apb_uart is configured to have additional FIFO registers implemented (FIFO_MODE != None) and additional shadow registers implemented (SHADOW == YES). If these registers are not implemented, this register does not exist and reading from this register address returns zero.

Bits	Name	R/W	Description
31:2	Reserved and read as zero		
1:0	Shadow RCVR Trigger	R/W	Shadow RCVR Trigger. This is a shadow register for the RCVR trigger bits (FCR[7:6]). This can be used to remove the burden of having to store the previously written value to the FCR in memory and having to mask this value so that only the RCVR trigger bit gets updated. This is used to select the trigger level in the receiver FIFO at which the Received Data Available Interrupt is generated. It also determines when the dma_rx_req_n signal is asserted when DMA Mode (FCR[3]) = 1. The following trigger levels are supported: 00 = 1 character in the FIFO 01 = FIFO ¼ full 10 = FIFO ½ full 11 = FIFO 2 less than full **Reset Value:** 0x0

STET

- **Name:** Shadow TX Empty Trigger
- **Size:** 32 bits
- **Address Offset:** 0xA0
- **Read/write access:** read/write

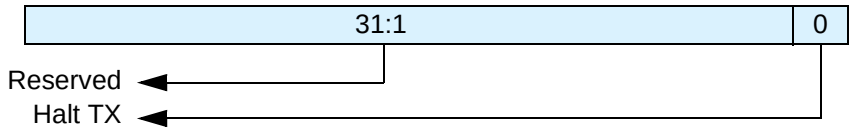


This register is only valid when the DW_apb_uart is configured to have FIFOs implemented (FIFO_MODE != NONE) and THRE interrupt support implemented (THRE_MODE_USER == Enabled) and additional shadow registers implemented (SHADOW == YES). If FIFOs are not implemented or THRE interrupt support is not implemented or shadow registers are not implemented, this register does not exist and reading from this register address returns zero.

Bits	Name	R/W	Description
31:2	Reserved and read as zero		
1:0	Shadow TX Empty Trigger	R/W	Shadow TX Empty Trigger. This is a shadow register for the TX empty trigger bits (FCR[5:4]). This can be used to remove the burden of having to store the previously written value to the FCR in memory and having to mask this value so that only the TX empty trigger bit gets updated. This is used to select the empty threshold level at which the THRE Interrupts are generated when the mode is active. The following trigger levels are supported: 00 = FIFO empty 01 = 2 characters in the FIFO 10 = FIFO ¼ full 11 = FIFO ½ full **Reset Value:** 0x0 **Dependencies:** Writes have no effect when THRE_MODE_USER == Disabled.

HTX

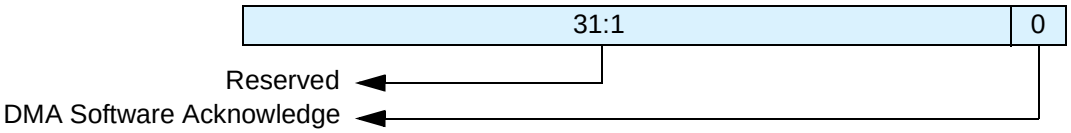
- **Name:** Halt TX
- **Size:** 32 bits
- **Address Offset:** 0xA4
- **Read/write access:** read/write



Bits	Name	R/W	Description
31:1	Reserved and read as zero		
0	Halt TX	R/W	This register is use to halt transmissions for testing, so that the transmit FIFO can be filled by the master when FIFOs are implemented and enabled. 0 = Halt TX disabled 1 = Halt TX enabled Note, if FIFOs are implemented and not enabled, the setting of the halt TX register has no effect on operation. **Reset Value:** 0x0 **Dependencies:** Writes have no effect when FIFO_MODE == None.

DMASA

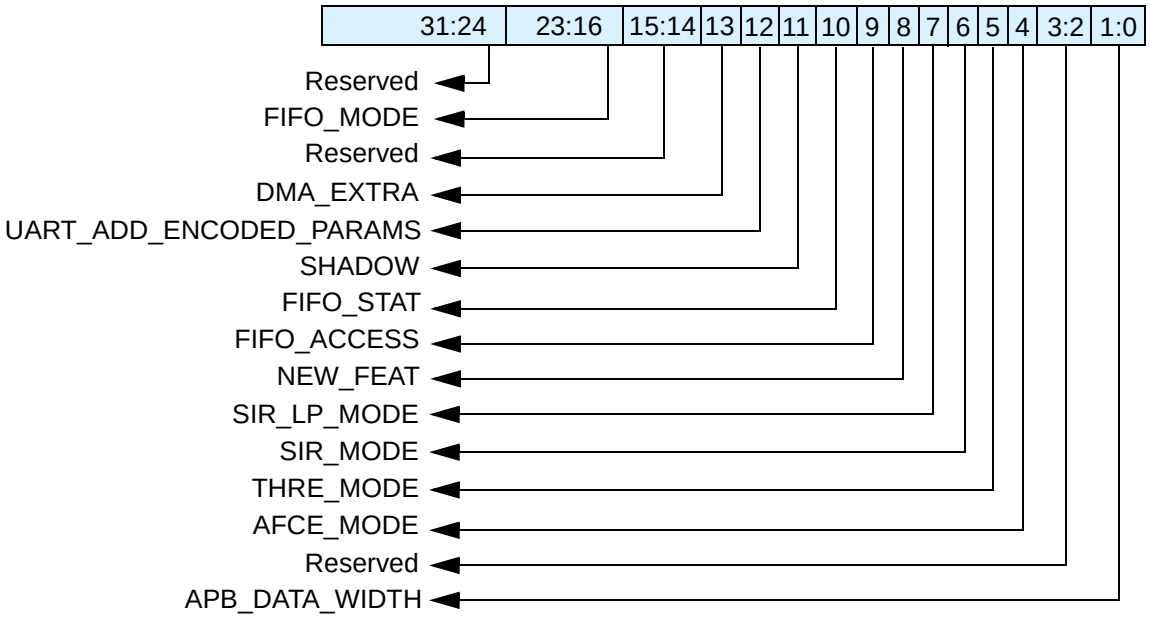
- **Name:** DMA Software Acknowledge
- **Size:** 32 bits
- **Address Offset:** 0xA8
- **Read/write access:** read/write



Bits	Name	R/W	Description
31:1	Reserved and read as zero		
0	DMA Software Acknowledge	W	This register is use to perform a DMA software acknowledge if a transfer needs to be terminated due to an error condition. For example, if the DMA disables the channel, then the DW_apb_uart should clear its request. This causes the TX request, TX single, RX request and RX single signals to de-assert. Note that this bit is 'self-clearing'. It is not necessary to clear this bit. **Reset Value:** 0x0 **Dependencies:** Writes have no effect when DMA_EXTRA == No.

CPR

- **Name:** Component Parameter Register
- **Size:** 32 bits
- **Address Offset:** 0xF4
- **Read/write access:** read-only
- **Dependency:** This register is only valid when the DW_apb_uart is configured to have the Component Parameter register implemented (UART_ADD_ENCODED_PARAMS == YES). If the Component Parameter register is not implemented, this register does not exist and reading from this register address returns zero.

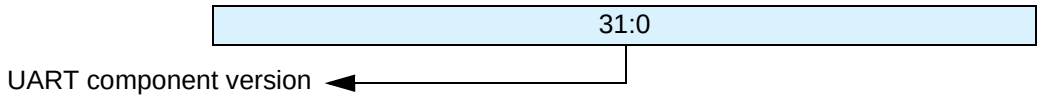


Bits	Name	R/W	Description
31:24	Reserved and read as zero		
23:16	FIFO_MODE	R	0x00 = 0 0x01 = 16 0x02 = 32 to 0x80 = 2048 0x81- 0xff = reserved
15:14	Reserved and read as zero		
13	DMA_EXTRA	R	0 = FALSE 1 = TRUE
12	UART_ADD_ENCODED_PARAMS	R	0 = FALSE 1 = TRUE
11	SHADOW	R	0 = FALSE 1 = TRUE
10	FIFO_STAT	R	0 = FALSE 1 = TRUE

Bits	Name	R/W	Description
9	FIFO_ACCESS	R	0 = FALSE 1 = TRUE
8	ADDITIONAL_FEAT	R	0 = FALSE 1 = TRUE
7	SIR_LP_MODE	R	0 = FALSE 1 = TRUE
6	SIR_MODE	R	0 = FALSE 1 = TRUE
5	THRE_MODE	R	0 = FALSE 1 = TRUE
4	AFCE_MODE	R	0 = FALSE 1 = TRUE
3:2	Reserved and read as zero		
1:0	APB_DATA_WIDTH	R	00 = 8 bits 01 = 16 bits 10 = 32 bits 11 = reserved

UCV

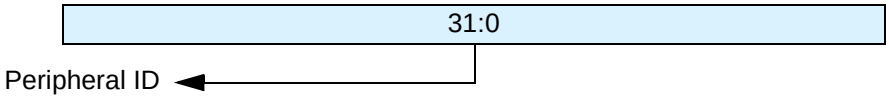
- **Name:** UART Component Version
- **Size:** 32 bits
- **Address Offset:** 0xF8
- **Read/write access:** read-only
- **Dependency:** This register is only valid when the DW_apb_uart is configured to have additional features implemented (ADDITIONAL_FEATURES == YES). If additional features are not implemented, this register does not exist and reading from this register address returns zero.



Bits	Name	R/W	Description
31:0	UART Component Version	R	ASCII value for each number in the version, followed by *. For example 32_30_31_2A represents the version 2.01* **Reset Value:** See the releases table in the *DW_apb_uart Release Notes*.

CTR

- **Name:** Component Type Register
- **Size:** 32 bits
- **Address Offset:** 0xFC
- **Read/write access:** read-only
- **Dependency:** This register is only valid when the DW_apb_uart is configured to have additional features implemented (ADDITIONAL_FEATURES == YES). If additional features are not implemented, this register does not exist and reading from this register address returns zero.



Bits	Name	R/W	Description
31:0	Peripheral ID	R	This register contains the peripherals identification code. **Reset Value:** 0x44570110

7
Programming the DW_apb_uart

The following topics provide information necessary to program the DW_apb_uart.

Software Drivers

The family of DesignWare AMBA Synthesizable Components includes a Driver Kit for the DW_apb_uart component. This low-level driver allows you to program a DW_apb_uart component and integrate your code into a larger software system. The Driver Kit provides the following benefits to IP designers:

- Proven method of access to DW_apb_uart minimizing usage errors
- Rapid software development with minimum overhead
- Detailed knowledge of DW_apb_uart register bit fields not required
- Easy integration of DW_apb_uart into existing software system
- Programming at register level eliminated

You must purchase a source code license (DWC-APB-Periph-Source) to use the DW_apb_uart Driver Kit. However, you can access some Driver Kit files and documentation in $DESIGNWARE_HOME/drivers/DW_apb_uart/latest. For more information about the Driver Kit, refer to the *DW_apb_uart Driver Kit User Guide*. For more information about purchasing the source code license and obtaining a download of the Driver Kit, contact Synopsys at designware@synopsys.com for details.

8 Verification

This chapter provides an overview of the testbench and tests available for DW_apb_uart verification. (Also see “Verification Environment Overview” on page 17). Once the DW_apb_uart has been configured and the verification environment set up, simulations can be automatically ran.

For more information about running simulations for DW_apb_uart in Connect, refer to “Verify Component” on page 37. For more information about verifying DW_apb_uart in coreConsultant, see “Verifying the DW_apb_uart” on page 160.

Note

The DW_apb_uart verification testbench is built with DesignWare AMBA 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 following web page:

www.synopsys.com/products/designware/docs/doc/amba/latest/dw_amba_install.pdf

Overview of DW_apb_uart Testbench

As illustrated in Figure 30, the DW_apb_uart Verilog testbench includes an instantiation of the design under test (DUT), AHB and APB bus models, and a Vera shell.

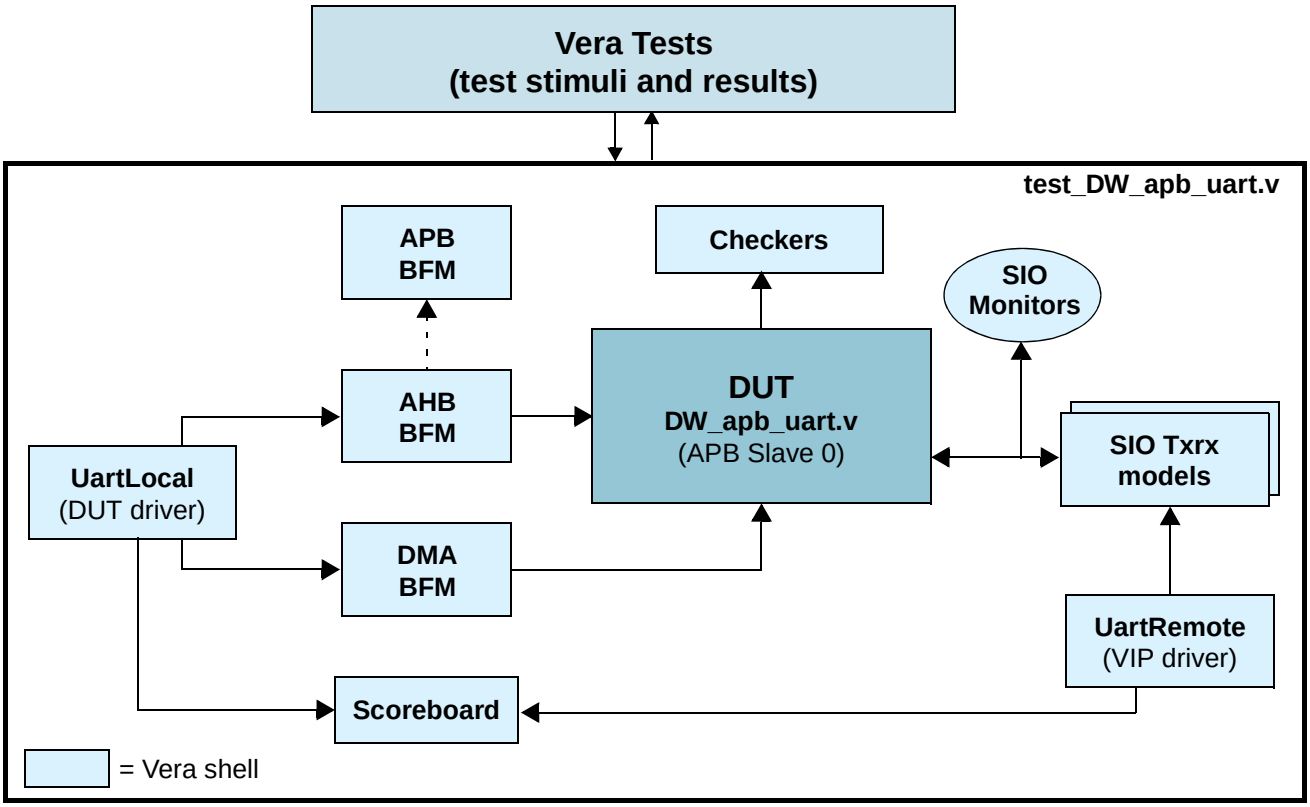


Figure 30: DW_apb_uart Testbench

The DW_apb_uart testbench consists of the following:

- Vera Test – Responsible for enumerating the test conditions under which the DUT (UART) is verified. These conditions steer the simulations in various aspects, such as the register settings of the UART, the transfer direction (UART to SIO_TxRx, SIO_TxRx to UART, loopback) and length (number of characters serially exchanged), number of iterations for a single test scenario, simulation controls, and so on. All this information is randomly created and encapsulated in several classes with associated Vera randomization and constraint constructs. This information is also relayed to the other Vera components.
- Testbench API – Takes in the randomized test conditions and uses the relevant portions for appropriate directing of the simulation controls, such as the number of iterations executed. It is also responsible for ensuring that all test monitors are alerted and set up for the indicated test type, as well as relaying information (in the form of class objects) to the two drivers (UartLocalClass, UartRemoteClass) in order to execute the desired simulation behavior to effect; for example, transfers to and from the DUT.
- DUT Driver, or UartLocalClass – Responsible for translating the information provided by the Testbench API into the desired simulation behaviors. This Vera component ensures that corresponding command and/or sequence of commands are issued to the AHB BFM to effect the desired register settings, transferring of data, toggling of the modem interface signals, loopback

mode, interrupts, and so on in the DUT(UART). Since the information directing the required simulations are shielded by UartLocalClass away from AHB BFM, revised versions of the latter Vera component can be easily accommodated by updating UartLocalClass.

- VIP Driver, or UartRemoteClass – Performs a similar role to that of UartLocalClass, translating the information provided by Testbench API into corresponding SIO_TxRx BFM commands in order to effect the desired simulation behaviors. Note that controls complementary to that of the UartLocalClass are performed in the UartRemoteClass, such that if the DUT performs transmits, then the SIO_TxRx BFM attempts reception(s). UartRemoteClass also serves to shield the rest of the verification environment from revised versions of this VIP component.
- AHB BFM – VIP harness BFM required to imitate as an AHB master. All actual register accesses (reads and writes) required by a current test are performed using AHB BFM commands. Existing class definitions for this BFM are re-used.
- DMA BFM – Exercises the DMA interface of the DUT/UARTv3.0. It behaves as another AHB master, issuing commands to perform reads and writes from/to the UART. These activities are coordinated within the UartLocalClass.
- Checkers – Examine the behavior of the DUT through the DUT signal interfaces, and evaluate the outcome of the prescribed tests targeted at the DUT. The verification tests determine the degree to which the DUT is verified, and is therefore linked to one (or more) test monitors in the test environment. These Checkers operate independently of the main flow in the test code. This form of messaging uses two classes, TestmonAlertClass and TestmonExecuteClass.
- SIOMonitor – Serial monitor VIP from the SIO VIP package. When appropriately parameterized, the SIO_Mon examines the serial bit patterns exchanged between the DUT and the SIO_TxRx.
- SIOTxRx BFM – Vera model of a UART capable of serial data exchanges with any other UART.
- APB Slave BFM – Used to ensure that violations in the APB accesses are appropriately captured and logged.
- Scoreboard – Tracks the data that are exchanged between the UART and the SIOTxrx models. This allows verification of the actual contents transmitted and/or received on either side in either direction.

9 Integration Considerations

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

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.

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.

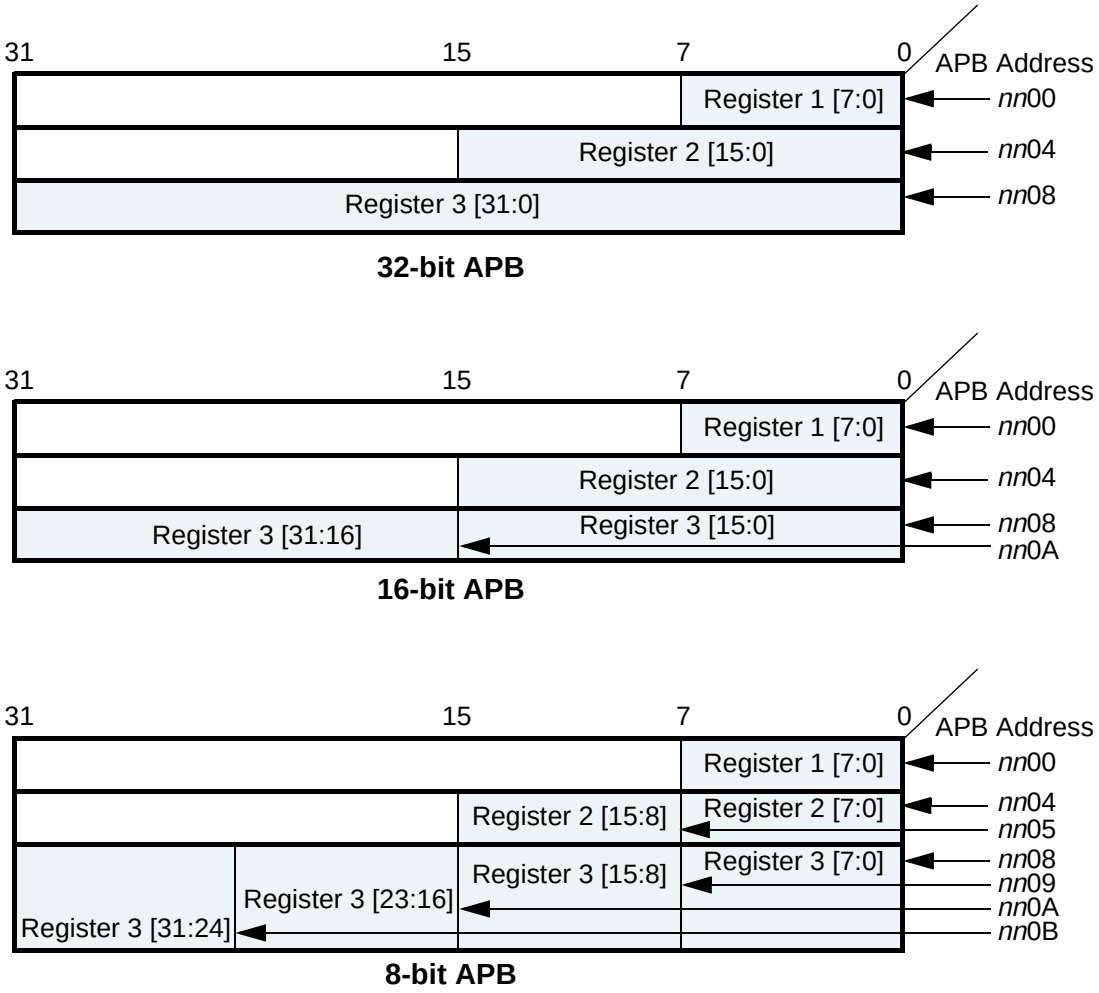


Figure 31: Read/Write Locations for Different APB Bus Data Widths

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.

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.

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.

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.

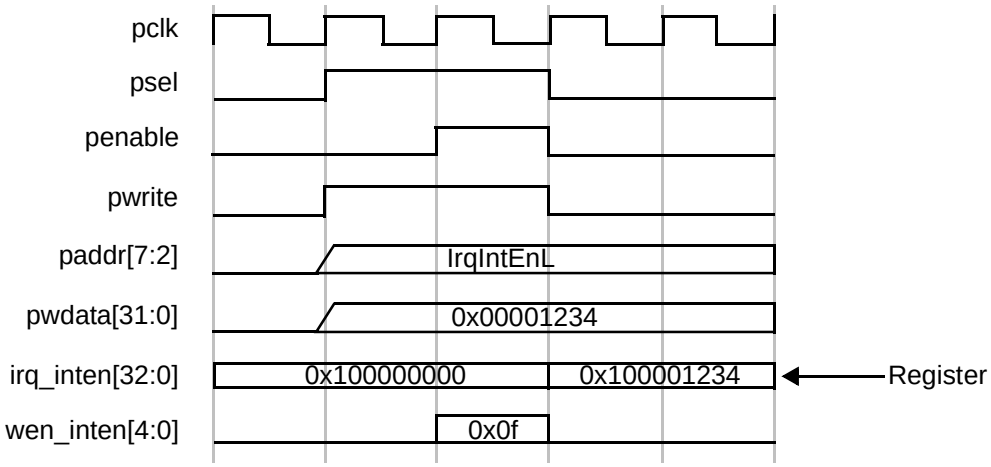


Figure 32: APB Write Transaction

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.

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.

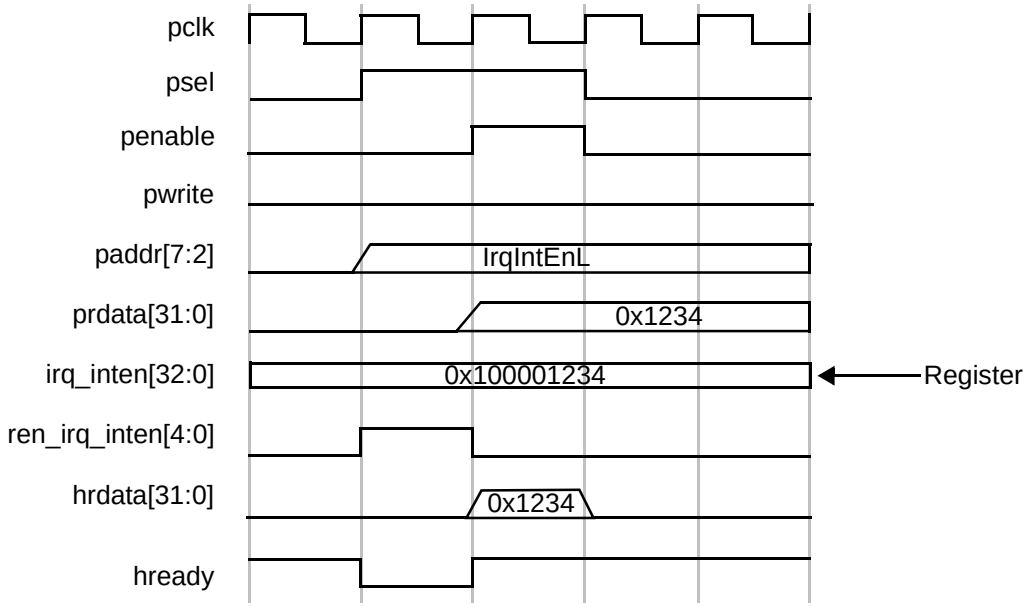


Figure 33: APB Read Transaction

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.

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>
```

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.

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 gives 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 9: Upper Byte Generation

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

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

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

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.

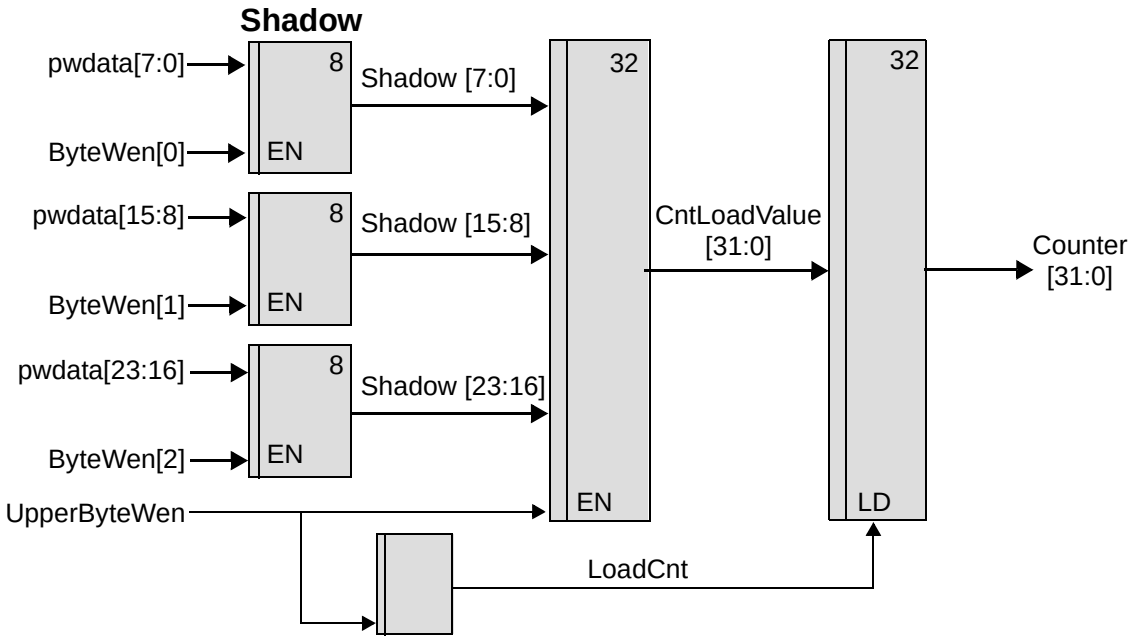


Figure 34: Coherent Loading – Identical Synchronous Clocks

The following timing diagram shows 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.

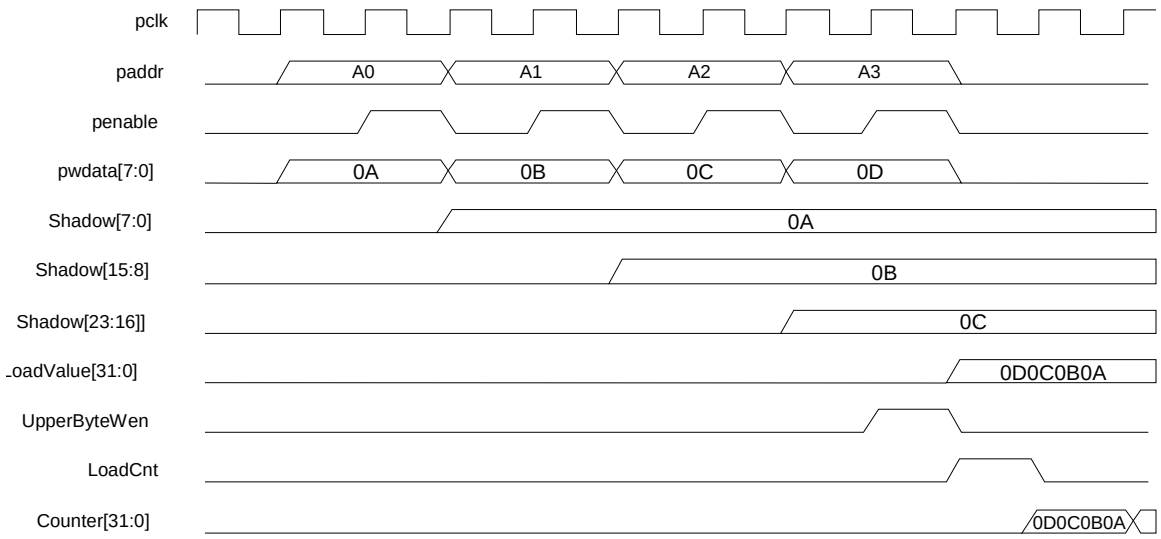


Figure 35: Coherent Loading – Identical Synchronous Clocks

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.

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.

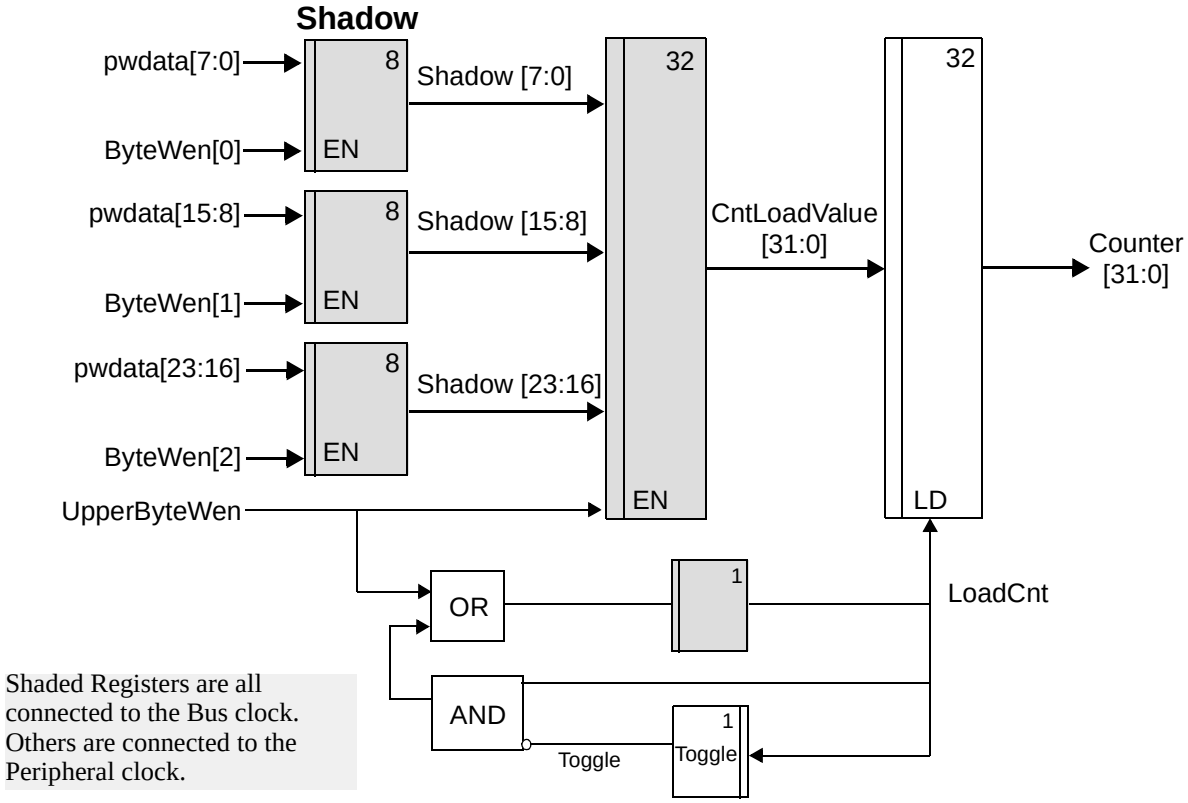


Figure 36: Coherent Loading – Synchronous Clocks

The following timing diagram shows 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.

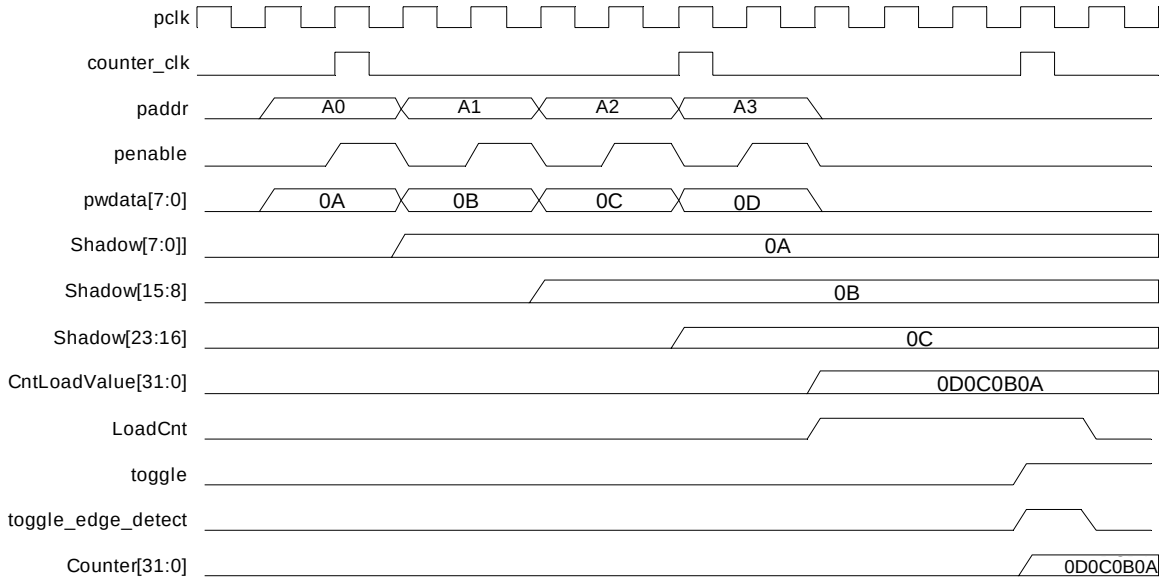
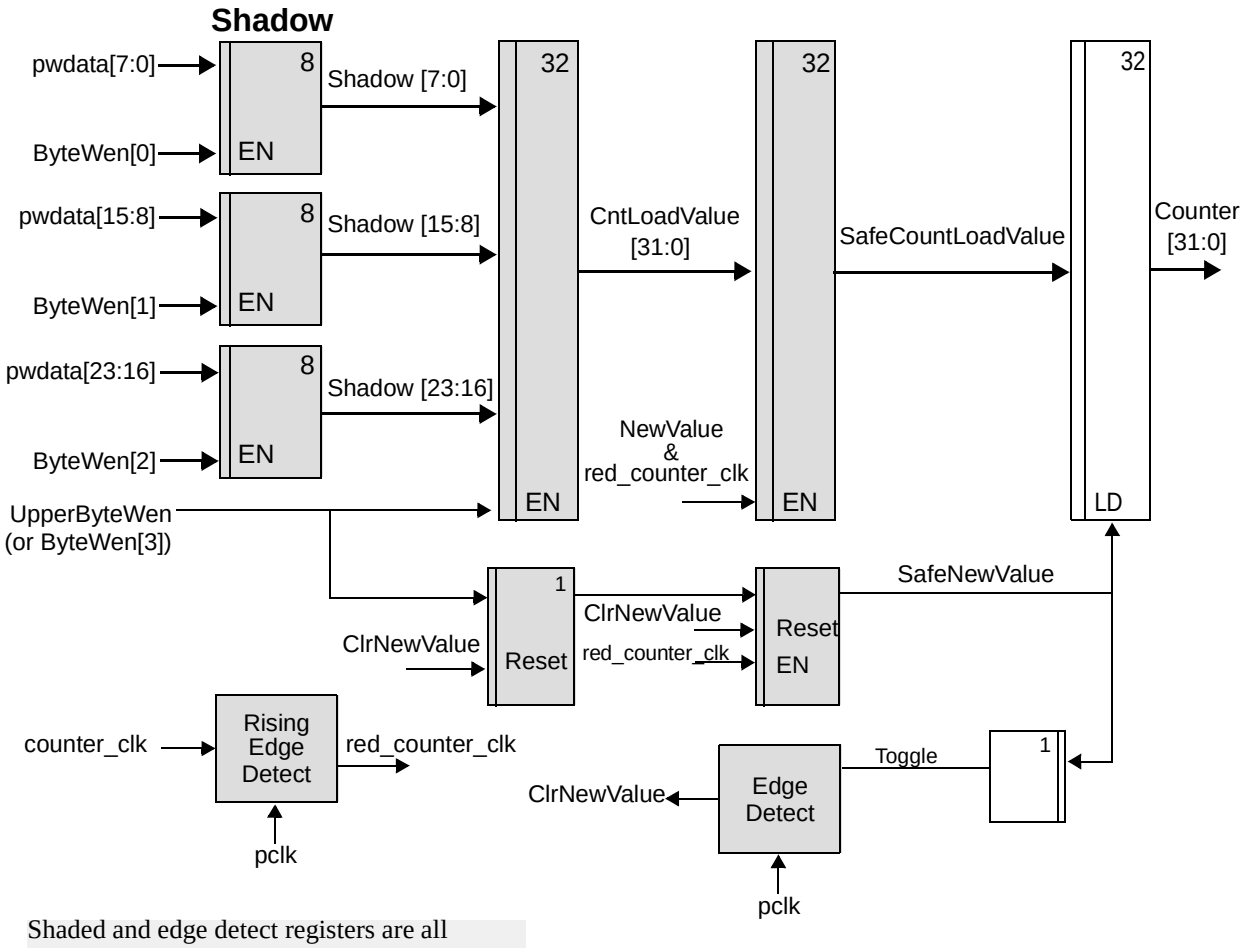


Figure 37: Coherent Loading – Synchronous Clocks

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.



Shaded and edge detect registers are all connected to the Bus clock. Others are connected to the Peripheral clock.

Figure 38: 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 following timing diagram does not show the shadow registers being loaded. This is identical to the loading for the other clock modes. 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.

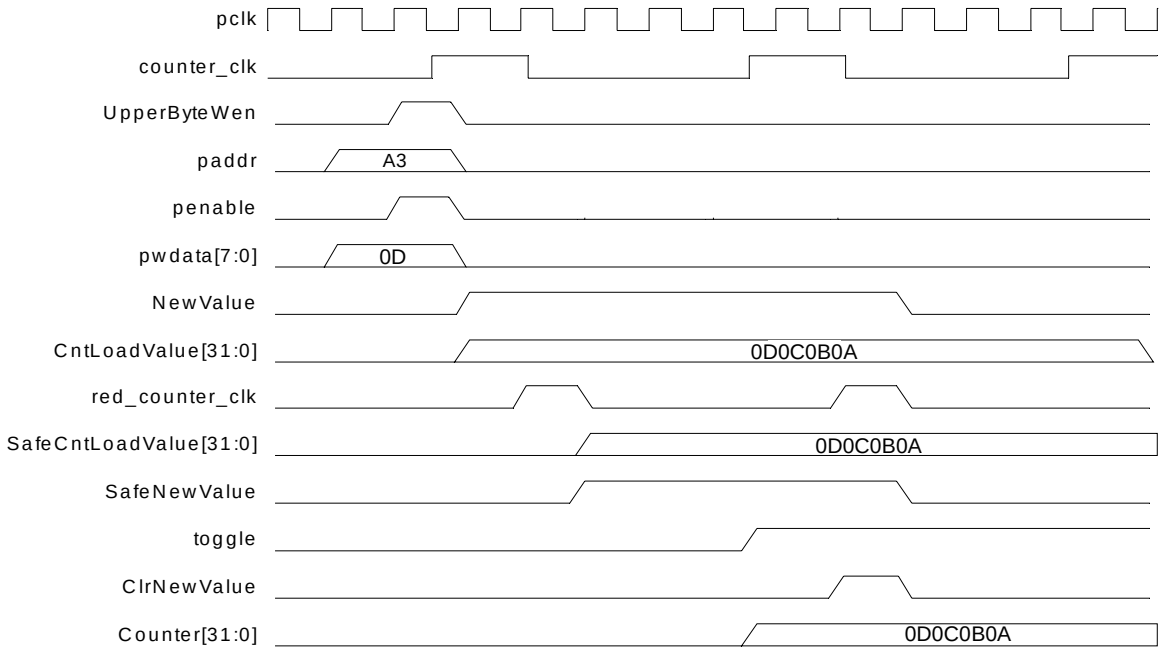


Figure 39: Coherent Loading – Asynchronous Clocks

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. 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.

Table 10: Lower Byte Generation

	Lower Byte Bus Width		
Counter Register Width	8	16	32
1 - 8	NCR	NCR	NCR
9 - 16	0	NCR	NCR
17 - 24	0	0	NCR
25 - 32	0	0	NCR

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

- Identical and/or synchronous
- Asynchronous

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 timing diagram.

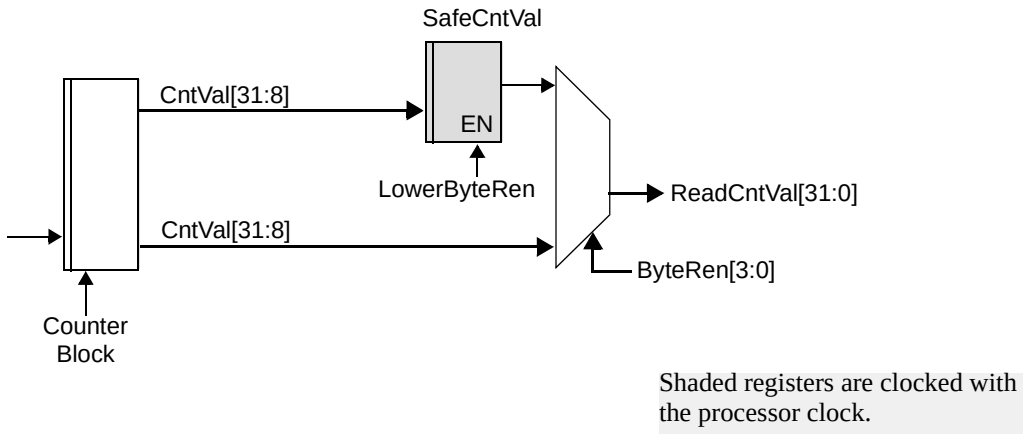


Figure 40: Coherent Registering – Synchronous Clocks

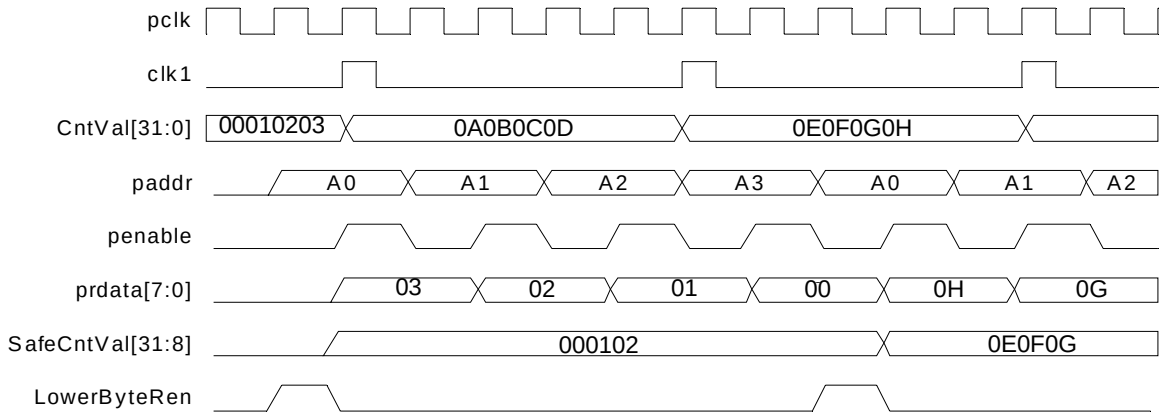


Figure 41: Coherent Registering – Synchronous Clocks

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 and timing diagram illustrate the synchronization of the counter clock and the update of the shadow register.

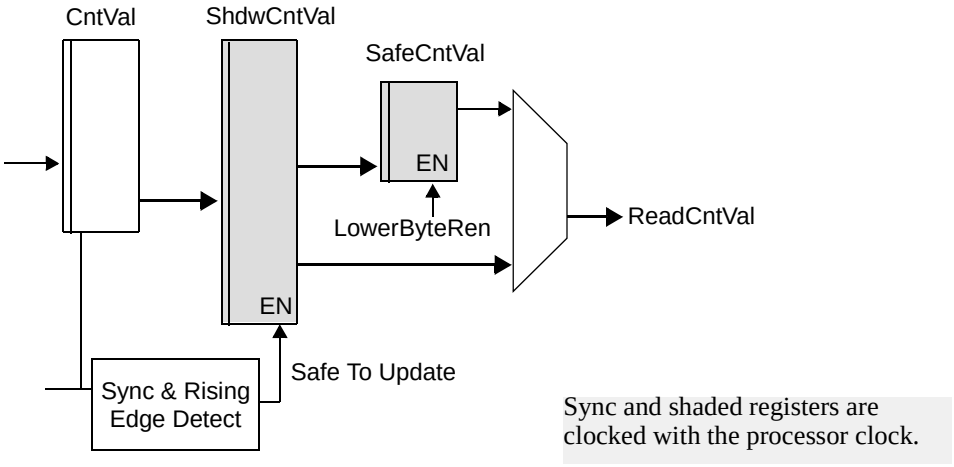


Figure 42: Coherency and Shadow Registering – Asynchronous Clocks

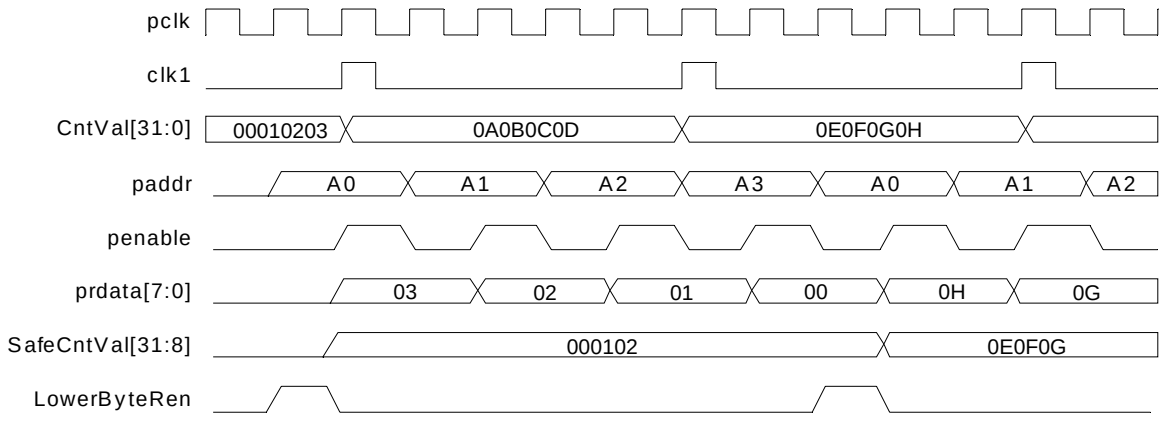


Figure 43: Transfer to Shadowing Registers– Asynchronous Clocks

A

Building and Verifying Your DW_apb_uart

This chapter provides an overview of the step-by-step process you use to configure, synthesize, and verify your DW_apb_uart component using the Synopsys coreConsultant tool. You use coreConsultant to create a workspace that is your working version of a subsystem, where you connect, configure, simulate, and synthesize your implementation of the subsystem. You can create several workspaces to experiment with different design alternatives. The topics are as follows:

If you plan to include the DW_apb_uart as part of a DesignWare AMBA subsystem, then you will want to use the DesignWare Connect tool. This tool is a customized version of coreAssembler. For more information about including DW_apb_uart in a DesignWare AMBA subsystem, refer to Chapter 2, "Building and Verifying a Subsystem" on page 21.

Set up Your Environment

DW_apb_uart is included with a DesignWare Synthesizable Components for AMBA 2 release; it is assumed that you have already downloaded and installed the release. However, to download and install the latest versions of required tools, refer to the *DesignWare AMBA Synthesizable Components Installation Guide*.

You also need to set up your environment correctly using specific environment variables, such as DESIGNWARE_HOME, VERA_HOME, PATH, and SYNOPSYS. If you are not familiar with these requirements and the necessary licenses, refer to "Setting up Your Environment" in the *DesignWare AMBA Synthesizable Components Installation Guide.*

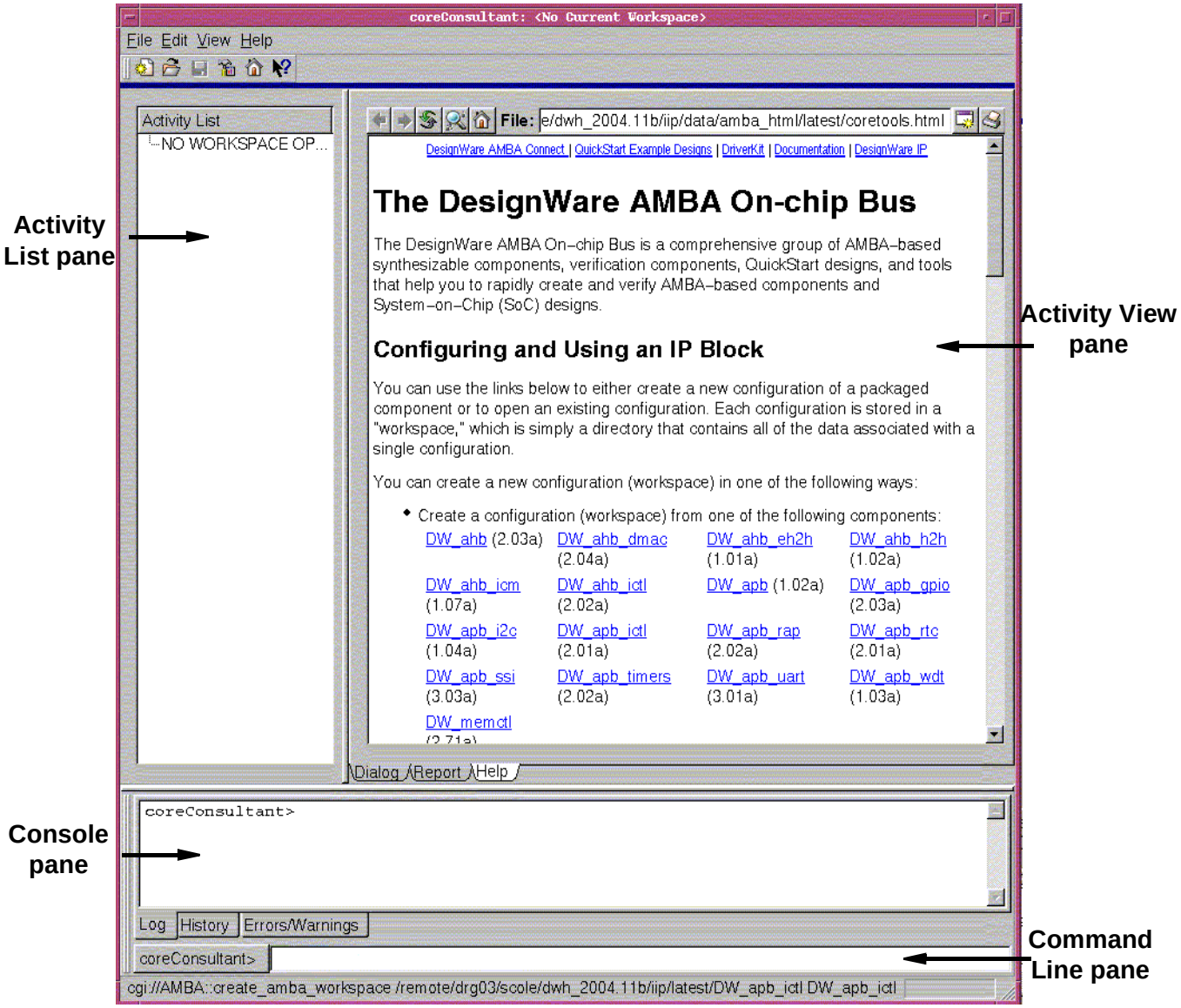
Start coreConsultant

To invoke coreConsultant:

1. In a UNIX shell, navigate to a directory where you plan to locate your component workspace.
2. Invoke the coreConsultant GUI:

```
% coreConsultant
```

The welcome page is displayed, similar to the one below.



3. Click on the DW_apb_uart link in the “Configuring and Using an IP block” section to create a new workspace. After you have created a workspace, you can also continue working from the point you left off by using the “Open” link to open it back up.

   In the resulting dialog box, specify the workspace name and workspace root directory, or use the defaults – a workspace name is the name of a configuration of a core; the workspace root directory is the directory in which the configuration is created. Click OK.

   You may notice that you are already in the Specify Configuration activity under the Create RTL category in the Activity List on the left, and that the Set Design Prefix activity is already checked in the list. It is not necessary for you to set the design prefix at this point of the learning phase. You may use this feature in the future if you ever use multiple versions of a component in a design.

Check Your Environment

Before you begin configuring your component, it is recommended that you check your environment to make sure you have the latest tool versions installed and your environment variables set up correctly.

To check your environment, use the **Help > Check Environment** menu path.

An HTML report is displayed in a separate dialog. This report lists the specific tools and versions installed in your environment. It also displays errors when a specific tool is not installed or if you are using an older version than you need. You will also see an error if your $DESIGNWARE_HOME environment variable has not been set up correctly.

Configure DW_apb_uart

This section steps you through the tasks in the coreConsultant GUI that configure your core. Complete information on the latest version of coreConsultant can be found on the web in the *coreConsultant User Guide*. To view documentation specific to your version of coreConsultant, choose the **Help** pull-down menu from the coreConsultant GUI.

At any time during this process you can click on the Help tab for each activity to activate the coreConsultant online help.

Note

Throughout the remaining steps in this chapter, it is best if you apply the default values so that the directions and descriptions in the chapter will coincide with your display. After you have used the DW_apb_uart in coreConsultant, you can then go back through these steps and change values in order to see how they affect the design.

1. Notice that the Set Design Prefix activity is already checked. This setting is used to make each design in your component have a unique name. This is needed only when you have two or more versions of
2. **Specify Configuration** – The Specify Configuration activity is where you specify the basic configuration of the DW_apb_uart. If you have a Source license, you can choose to use DesignWare Building Block IP (DWBB) components for optimal Synthesis QoR. Alternatively, if you have an RTL source licence, you may use source code for DWBB components without a DesignWare license. If you use RTL source and also have a DesignWare key, you can choose to retain the DWBB parts.

Look through the basic parameters for each item. Click the Next button to view the other configuration defaults. If you need help with any field in the activity pane, right-click on the field name and then left-click on the What's This box. When finished, click Apply.

When the configuration setup is complete, the Report tab is displayed, which gives you all the source files (in encrypted format if you have a DW license, and unencrypted if you have a source license) and all the parameters that have been set for this particular configuration. Reports contain useful information as you complete each step in the coreConsultant process. Familiarize yourself with the report contents before going to the next step.

Create Gate-Level Netlist

To run synthesis on the DW_apb_uart and create a gate-level netlist, step through the following tasks in the coreConsultant GUI. You need to click the check box next to each activity in order to access the specific activity dialog. At any time, you can click on the Help tab for each activity in order to activate the coreConsultant online help.

1. Look at the tool installation root directories in the Tool Installation Roots dialog, which is accessed from the toolbar menu through **Edit > Tool Installation Roots**, or by using the Tools button on the toolbar. You can type values directly in the data fields, or use the buttons to locate the correct directories. The tool choices are:

   - Design Compiler (dc_shell) – Specifies the location for the root directory of the Design Compiler installation, if different from the default location. You are required to select either Design Compiler or FPGA Compiler II.
   - Physical Compiler (psyn_shell) – Enables the Physical Compiler if you plan to use an incremental physical synthesis strategy or if you plan to do RTL to place gates.
   - Primetime (pt_shell) – Enables Primetime if you plan to implement budgeting or generate timing models.
   - Formality (fm_shell) – Enables Formality if you plan to formally verify the synthesized gate-level implementation of the core.
   - DC FPGA (fpga_shell) – Enables Design Compiler FPGA if your synthesis targets high-end FPGA devices.

   At a minimum for this exercise, dc_shell or fc2_shell must have defined installation directories, and in order to complete the optional formal verification in this chapter, you will also need fm_shell.

2. **Specify Target Technology** – coreConsultant analyzes the target technology library and uses it to generate a synthesis strategy that is optimized for your technology library. In the Design Compiler windows, a target and link library must be specified; otherwise, errors occur in coreConsultant.

   Under the Specify Target Technology category in the Activity List, the title in the tabs depends on the compiler you chose in the previous step. Regardless, this screen provides fields for you to enter the search path for the specific compiler, as well as target and link library paths. If necessary, specify the search path for the tool you specified in the previous screen. Also, specify the path to the target and link libraries. Click Apply and familiarize yourself with the resultant report, which gives you the technology information.

3. **Specify Clock(s)** – In the Specify Clock(s) activity, look at the attributes associated with each of the real and virtual clocks in your design. Click Apply and familiarize yourself with the resultant report, which gives you clock information.
4. **Specify Operating Conditions and Wire Loads** – In the Specify Operating Conditions and Wire Loads activity, look at the attributes relating to the chip environment. If you do not see a value beside OperatingConditionsWorst, select an appropriate value from the drop-down list; if there is no value for this attribute, you will get an error message. Click Apply and look at the report, which gives the operating conditions and wireload information.
5. **Specify Port Constraints** – In the Specify Port Constraints activity, look at the attributes associated with input delay, drive strength, DRC constraints, output delay, and load specifications. Click Apply and look at the report, which gives the port constraint checks.
6. **Specify Synthesis Methodology** – In the Specify Synthesis Methodology activity, look at the synthesis strategy attributes. Note that these attributes are typically set by the core developer and are not required to be modified by the core integrator. If you want to add your own commands during a synthesis, you use the Advanced tab in order to provide pathnames to your auxiliary scripts. Also click on the Physical Synthesis, and Fpga Synthesis tabs to familiarize yourself with those items. Click Apply and look at the report, which gives design information. For more information on adding auxiliary scripts, refer to “Advanced Synthesis Attributes” in the *coreConsultant User Guide*.
7. **Specify Test Methodology** – In the Specify Test Methodology activity, look at the scan test attributes. Also click on the other tabs to familiarize yourself with auto-fix attributes, SoC test wrapper attributes, test wrapper integration attributes, BIST attributes, and BIST testpoint insertion attributes. Click Apply and look at the report, which gives design-for-test information.
8. **Synthesize** – Choose the Synthesize activity. Do the following:
   a. Choose the Strategy tab.
   b. Click the Options button beside DCTCL_opto_strategy and look through the strategy parameters. For example, you can use the Gate Clocks During Elaboration check box in the Clock Gating tab in order to add parameters that enable and control the use of clock gating. Click OK when you are done. For more information on clock gating and other parameters for synthesis strategies, refer to “DC(TCL)_opto_strategy” in the *coreConsultant User Guide*.

      For FPGA synthesis, click the Options button and then select the FPGA Synthesis tab. It is here where you specify the location of your FPGA device and speed grade, synthetic libraries other than DesignWare Foundation libraries, implementation of DC-FPGA operators, and so on. For more information about running synthesis for an FPGA device, refer to the *coreConsultant User Guide.*

      For Design for Test, click the Options button and then select the Design for Test tab. Here you can specify whether to add the -scan option to the initial compile call (Test Read Compile) and/or insert design for test circuitry (Insert Dft). For more information about include DFT in your synthesis run, refer to the *coreConsultant User Guide.*

c. Choose the Options tab. Look at the values for the parameters listed below.

Field Name	Description
Execution Options	
Generate Scripts only?	**Values**: Enable or Disable **Default Value**: Disable **Description**: Writes the run.scr script, but it is not run when you click Apply. To run the script, go to the component workspace and run the script.
Run Style	**Values**: local, lsf, grd, or remote **Default Value**: local **Description**: Describes how to run the command: locally, through LSF, through GRD, or through the remote shell.
Run Style Options	**Values**: user-defined **Default Value**: none **Description**: Additional options for the run style options except local. For remote, specify the hostname. For LSF and GRD, specify bsub or qsub commands.
Parallel job CPU limit	**Values**: user-defined; minimum value is 1 **Default Value**: 1 **Description**: Specifies number of parallel compile jobs that can be run.
Send e-mail	**Values**: current user's name **Description**: E-mail is sent when the command script completes or is terminated.
Skip reading $HOME/.synopsys_dc.setup	**Values**: Enable or Disable **Default Value**: Disable **Description**: Forces tools not to read .synopsys_dc.setup file from $HOME.

d. If it is not already set, choose the "local" Run Style option and maintain the other default settings.

e. Look through the Licenses and Reports tabs, and ensure that you have all the licenses that are required to run this synthesis session.

f. Click Apply in the Synthesize activity pane to start synthesis from coreConsultant. The current status of the synthesis run is displayed in the main window. Click the Reload Page button if you want to update the status in this screen.

9. **Generate Test Vectors** – This option allows you to generate ATPG test vectors with TetraMax. For more information about this option, refer to "Generating ATPG Test Vectors" in the *coreConsultant User Guide*.

Checking Synthesis Status and Results

To check synthesis status and results, click the Report tab for the synthesis options; coreConsultant displays a dialog that indicates:

- Your selected Run Style (local, lsf, grd, or remote)
- The full path to the HTML file that contains your synthesis results
- The name of the host on which the synthesis is running
- The process ID (Job Id) of the synthesis
- The status of the synthesis job (running or done)

The Results dialog also enables you to kill the synthesis (Kill Job) and to refresh the status display in the Results dialog (Refresh Status). The Results information includes:

- Summary of log files
- Synthesis stages that completed
- Summary of stage results

This information indicates whether the synthesis executed successfully, and lists the DW_apb_uart transactions that occurred during the scenario(s). Thorough analysis of the scenario execution requires detailed analysis of all synthesis log files and inspection of report summaries.

Synthesis Output Files

All the synthesis results and log files are created under the syn directory in your workspace. Two of the files in the *workspace*/syn directory are:

- run.scr – Top-level synthesis script for DW_apb_uart
- run.log – Synthesis log file

Your final netlist and report directories depend on the QoR effort that you chose for your synthesis (default is medium):

- low – initial
- medium – incr1
- high – incr2

For information about deliverables that are generated after synthesis is performed, refer to “Database Description” on page 167.

Running Synthesis from Command Line

To run synthesis from the command line prompt for the files generated by coreConsultant, enter the following command:

```
% run.scr
```

This script resides in your *workspace*/syn directory.

Verifying the DW_apb_uart

This section provides the steps you use to execute the testbench available for DW_apb_uart verification. Once the DW_apb_uart has been configured and the verification environment has been set up, simulations can be automatically run. In fact, both synthesis and simulation activities can be done in parallel, so you do not have to wait for synthesis to complete in order to start a simulation.

DW_apb_uart verification is detailed in the following sections:

- "Creating GTECH Simulation Models"
- "Verifying the Simulation Model" on page 162

Note

For GTECH Simulations Only. Due to the configurable nature of the component, some ports in the testbench may not be needed for your chosen configuration. Warnings about undriven nets may appear. These warnings are to be expected, and you can ignore them. The verification result files show if the verification ran successfully.

Creating GTECH Simulation Models

DesignWare AMBA Synthesizable Components (coreKit RTL) are delivered in encrypted format, rather than source code, and some simulators cannot read the encrypted source files. In order for these simulators to read the encrypted files, you must either perform a GTECH conversion or purchase a source license from Synopsys.

Note

The Synopsys VCS simulator reads the encrypted files directly and does not require a GTECH conversion. All other supported simulators require a GTECH simulation model. You need a DesignWare license to complete the GTECH generation process. If you are a source license customer, then you do not have to generate a GTECH simulation model, even if you are using a non-VCS simulator.

Also, it is not possible to perform a GTECH simulation with DC FPGA.

1. **Generate GTECH Model** – To create a GTECH simulation model, click on the Generate GTECH Simulation Model activity.
2. Look at the values for the parameters listed below.

Field Name	Description
Execution Options	
Generate Scripts only?	**Values**: Enable or Disable **Default Value**: Disable **Description**: Writes scripts that run the generation of the GTECH simulation model, but they are not run when you click Apply. To run these scripts, go to the gtech directory of the component workspace and run the run.scr script.

Field Name	Description
Run Style	**Values:** local, lsf, grd, or remote **Default Value:** local **Description:** Describes how to run the command: locally, through LSF, through GRR, or through the remote shell.
Run Style Options	**Values:** user-defined **Default Value:** none **Description:** Additional options for the run style options except local. For remote, specify the hostname. For LSF and GRD, specify bsub or qsub commands.
Send e-mail	**Values:** current user's name **Description:** E-mail is sent when the command script completes or is terminated.
Synthesis Control	
Ungroup Netlist after Compile	**Values:** 0 (Enabled) or 1 (Disabled) **Default Value:** 0 (Disabled) **Description:** Ungroups the design to provide a non-hierarchical netlist

3. Click Apply; coreConsultant invokes Design Compiler to perform a low-effort compile (quickmap) of your custom configuration using the Synopsys technology-independent GTECH library. After this activity has completed, an e-mail similar to the following is sent to the specified user name (if you enabled that option):

```
Activity:   GenerateGtechModel
Workspace:  workspace_path
Design:     DW_apb_uart
Started:    Wed Jul 24 16:19:48 BST 2002
Finished:   Wed Jul 24 16:21:42 BST 2002
Status:     Completed
Results:    workspace_path/gtech/gtech.log
```

Your simulation model is contained in the DW_apb_uart.v output file that is written to *workspace*/gtech/qmap/db.

Verifying the Simulation Model

To verify the DW_apb_uart, use coreConsultant to complete the following steps:

1. **(Optional) Formal Verification** – You can run formal verification scripts using Synopsys Formality (fm_shell) to check two designs for functional equivalence. You can check the gate-level design from a selected phase of a previously executed synthesis strategy against either the RTL version of the design or the gate-level design from another stage of synthesis. To run this, choose Formal Verification under the Verify Component category and then click Apply.
2. **Setup and Run Simulations** – Specify the simulation by completing the Setup and Run Simulations activity:
   a. In the Select Simulator area, click on the Simulator view list item to view available simulators (VCS is the default).
   b. Specify an appropriate Verilog simulator from the drop-down menu.

      For installation instructions and information about required tools and versions, refer to "Setting up Your Environment" in the *DesignWare AMBA Synthesizable Components Installation Guide*. For general information about the contents of the release, refer to the *DesignWare DW_apb_uart Release Notes*.
   c. In the Simulator Setup area of the Simulator pane, look at the parameters for the simulator setup, as detailed in the following table.

Field Name	Description
Root Directory of Cadence Installation	The path to the top of the directory tree where the Cadence NC-Verilog executable is found; coreConsultant automatically detects this path. The NC-Verilog executables reside in the ./bin subdirectory.
MTI Include Directory	The path to the include directory contained within your MTI simulator installation area. A valid directory includes the veriuser.h file.
Vera Install Area ($VERA_HOME)	Path to your Vera installation. **Default Value**: value of your $VERA_HOME variable
Vera .vro file cache directory	Location to store .vro files. These files are generated as part of building the testbench. Encrypted Vera is source is compiled and stored in the cache.
DW Foundation Install Area ($SYNOPSYS)	Path to your Synopsys DW Foundation installation, which is set from the Tools Installation Areas dialog box. Any change to this value must be made from the Tool Installation Areas coreConsultant dialog box.
C Compiler for (Vera PLI)	**Values:** gcc or cc **Default Value:** gcc **Description:** Invokes the specific C compiler to create a Vera PLI for your chosen non-VCS simulator. Choose cc if you have the platform native ANSI C compiler installed. Choose gcc if you have GNU C compiler installed.

   d. In the Waves Setup area of the Simulator pane, look at the parameters for the waves setup as detailed below.

Note

For the Generate Waves File setting, enable the check box so that the simulation creates a file that you can use later for debugging the simulation, if you want to do so.

Field Name	Description
Generates waves file	**Values**: Enable or Disabled **Default Value**: Disabled **Description**: Indicates whether a wave file should be created for debugging with a wave file browser after simulation ends. Uses VPD file format for VCS and VCD format for the other supported simulators.
Depth of waves to be recorded	**Description**: Enter the depth of the signal hierarchy for which to record waves in the dump file. A depth of 0 indicates all signals in the hierarchy are included in the wave file.

e. Choose the View list choice.

f. In the View Selection area of the View pane, look at the choice of views of the design you can simulate from the drop-down list:

- RTL – requires a source license or Synopsys VCS
- GTECH – requires that you have completed the Generate GTECH Model activity (refer to page 160) only if you are using a non-VCS simulator and do not have a source license.

g. Choose the Execution Options list choice to set the following options:

Field Name	Description
Do Not Launch Simulation	**Values**: Enable or Disable **Default Value**: Disable **Description**: Determines whether to execute the simulation or just generate the simulation run script. If checked, coreConsultant generates, but does not execute, the simulation run script. You can execute the script at a later time by invoking the run script (*workspace*/sim/run.scr) directly from the UNIX command line or by repeating the Verification activity with Do Not Launch Simulation unselected.
Run Style	**Values**: local, lsf, grd, or remote **Default Value**: local **Description**: Describes how to run the command: locally, through LSF, through GRD, or through the remote shell.
Run Style Options	**Values**: user-defined **Default Value**: none **Description**: Additional options for the run style options except local. For remote, specify the hostname. For LSF and GRD, specify bsub or qsub commands.
Send e-mail	**Values**: current user's name **Description**: E-mail is sent when the command script completes or is terminated.

h. Select Testbench and look at the options described below:

Field Name	Description
Run test_uart	**Values**: Enable or Disable **Default Value**: Enable **Description**: Tests general functions of the DW_apb_uart.

i. Click Apply to run the simulation.

When you click Apply, coreConsultant performs the following actions:

- Sets up the DW_apb_uart verification environment to match your selected DW_apb_uart configuration.
- Generates the simulation run script (run.scr) and writes it to your *workspace*/sim directory.
- Invokes the simulation run script, unless you enabled the Do Not Launch Simulation option.

The simulation run script, in turn, performs the following actions:

- Links the generated command files, and recompiles the testbench.
- Invokes your simulator to simulate the specified scenarios.
- Writes the simulation output files to your *workspace*/sim/test_* directory.
- If an e-mail address is specified, sends the simulation completion information to that e-mail address when the simulation is complete.

For an overview of the related tests, refer to "Verification" on page 133.

Checking Simulation Status and Results

To check simulation status and results, click the Report tab for either the GTECH models or for the simulation options; coreConsultant displays a dialog that indicates:

- Your selected Run Style (local, lsf, grd, or remote)
- The full path to the HTML file that contains your simulation results
- The name of the host on which the simulation is running
- The process ID (Job Id) of the simulation
- The status of the simulation job (running or done)

If you selected the "LSF/GRD" option for the Run Style, then the status of the simulation jobs (running or complete) is incorrect. Once all the simulation jobs are submitted to the LSF/GRD queue, the status would indicate "complete." You should use "bjobs/qstatus" to see whether all the jobs are completed.

The Results dialog also enables you to kill the simulation (Kill Job) and to refresh the status display in the Results dialog (Refresh Status). The Results information includes:

- Vera compile execution messages
- Simulation execution messages
- DW_apb_uart bus transactions

This information indicates whether the simulation executed successfully, and lists the DW_apb_uart transactions that occurred during the scenario(s).

Thorough analysis of the scenario execution requires detailed analysis of all simulation output files and inspection of simulation waveforms with a waveform viewer.

Creating a Batch Script

It sometimes helps to have a batch file that contains information about the workspace, parameters, attributes, and so on. You can then review these by looking at the file in an ASCII editor. To do this, choose the **File > Write Batch Script** menu item and enter a name for the file. Then look at the contents to familiarize yourself with the information that you can get from this file. You can use the batch script to reproduce the workspace.

Applying Default Verification Attributes

To reset all DW_apb_uart verification attributes to their default values, use the Default button in the Setup and Run Simulation activity under the Verification tab.

To examine default attribute values without resetting the attribute values in your current workspace, create a new workspace; the new workspace has all the default attribute values. Alternatively, use the Default button to reset the values, and then close your current workspace without saving it.

B
Database Description

This appendix lists the deliverables and other reference files that are generated from the coreConsultant flow.

This appendix includes the following sections:

Design/HDL Files

The following sections describe the design and HDL files that are produced by coreConsultant when configuring and verifying a DesignWare AMBA component.

RTL-Level Files

The following table describes the RTL files that are generated by the Create RTL activity of the coreConsultant GUI. They are encrypted except where otherwise noted.

Note

Any Synopsys synthesis tool or simulator can read encrypted RTL files.

Table 11: RTL-Level Files

Files	Encrypted?	Purpose
./src/*component*_cc_constants.v	No	Includes definitions and values of all configuration parameters that you have specified for the component.

./src/*component*.v	No	Top-level HDL file.
		When you include the component in your simulation, you must include the DesignWare libraries by using the following options in your simulator invocation:
		`-y ${SYNOPSYS}/packages/gtech/src_ver`
		`-y ${SYNOPSYS}/dw/sim_ver`
		For an example of this process, refer to the *DW_AMBA QuickStart SingleLayer Example Guide*.
		Note: If you could not open the QuickStart documentation, it means that you have not downloaded the QuickStart examples. For download instructions, please refer to the *DesignWare AMBA Synthesizable Components Installation Guide*.

./src/*component_submodule*.v	Yes	Sub-modules of component
./src/*component*_constants.v	No	Includes the constants used internally in the design.
./src/*component*.lst	No	Lists the order in which the RTL files should be read into tools, such as simulators or dc_shell. For example, use the following option to read the design into VCS: `vcs -f component.lst`
./src/*.update	Yes	Ignore these files. Used for VHDL generation
./export/*component*_inst.v	No	Instantiation of configured component for use in design

Simulation Model Files

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

Table 12: Simulation Model Files

Files	Encrypted?	Purpose

./gtech/final/db/*component*.v	No	Simulation model of the component for use with non-Synopsys simulators. A technology-independent, gate-level netlist. VHDL and Verilog versions are generated.
		When you use this simulation model in your simulation, you must include the DesignWare libraries by using the following options in your simulator invocation:
		`-y ${SYNOPSYS}/packages/gtech/src_ver`
		`-y ${SYNOPSYS}/dw/sim_ver`
		For an example of this process, refer to the *DW_AMBA QuickStart SingleLayer Example Guide*.
		Note: If you could not open the QuickStart documentation, it means that you have not downloaded the QuickStart examples. For download instructions, please refer to the *DesignWare AMBA Synthesizable Components Installation Guide*.

Register Map Files

These files only pertain to DW_ahb and DW_apb slaves, basically components that have a programming interface. The DesignWare AMBA components that do not have register map files are the DW_apb, DW_ahb_icm, and DW_ahb_h2h components. These files include address definitions (memory map) for the component. The following table includes a description of the C and Verilog header files generated for components with programming interfaces.

Table 13: Header Files

Files	Encrypted?	Purpose
./c_headers/*component*_defs.h	No	For use when programming the component in a C environment.
./verilog_headers/*component*_defs.v	No	For use when programming the component in a Verilog environment.

Synthesis Files

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

Table 14: Synthesis Files

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

Verification Reference Files

The 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 15: Verification Reference Files

Files	Encrypted?	Purpose
./sim/runtest	No	Perl script that runs the coreConsultant Verify Component activity from the command line.
./sim/runtest.log	No	The overall result of simulation, including pass/fail results.
./sim/test_*testname*/test.result	No	Pass/fail of individual test.
./sim/test_*testname*/test.log	No	Log file for individual test.

For more information about performing verification on your component, see the chapter titled Verification in this databook.

C
DesignWare QuickStart Designs

The DesignWare AMBA Synthesizable Components environment provides many templates and examples to help you be successful with your own design creation process. This section summarizes these system design aids, and points you to more information about them.

QuickStart Example Designs

QuickStart examples are provided with the DesignWare Synthesizable Components and verification models to help you learn about these products. The QuickStart examples show how to connect the DesignWare AMBA Synthesizable Components to the DW_apb and DW_ahb bus IP, and how to set up a verification environment. These are simulation-only subsystems to view waveforms, and not for use in synthesis. Each example design includes the following information:

- Block diagram of subsystem design, showing connections and ports
- Purpose of the example, and features included
- Example directory structure
- Important configuration and parameter information
- Overview of the testbench and tests that are provided
- Instructions on how to quickly perform a simulation run

For more information about QuickStart examples, refer to the *DesignWare AMBA QuickStart_SingleLayer Guide* and the *DesignWare AMBA QuickStart_MultiLayer Guide*.

Note

If you could not open the QuickStart documentation, it means that you have not downloaded the QuickStart examples. For download instructions, please refer to the *DesignWare AMBA Synthesizable Components Installation Guide*.

D
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 AMBA 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 AMBA 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 AMBA 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

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Z