



DesignWare DW_apb_i2c Databook

DesignWare Synthesizable Components for AMBA 2
DW_apb_i2c

Version 1.08a

April 16, 2007

Copyright Notice and Proprietary Information

Copyright © 2007 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Destination Control Statement

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

Disclaimer

SYNOPTYS, 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, ApolloGAI, 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, DFT Compiler, 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, Gatan, 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-VHDLint, 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-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, Taurus-Device, Taurus-Layout, Taurus-Lithography, Taurus-Process, Taurus-Topography, Taurus-Visual, Taurus-Workbench, 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	7
About This Manual	7
Related Documents	7
Manual Overview	7
Typographical and Symbol Conventions	8
Revision History	9
Getting Help	9
Additional Information	10
Comments?	10
Chapter 1	
Product Overview	11
DesignWare AMBA System Overview	11
DesignWare AMBA System Block Diagram	11
General product Description	13
DW_apb_i2c Block Diagram	13
Features	13
Standards Compliance	14
Verification Environment Overview	14
Licenses	14
Where To Go From Here	15
Chapter 2	
Building and Verifying a Subsystem	17
Setting up Your Environment	17
Overview of the Configuration and Integration Process	18
Start coreAssembler	21
Add DW_apb_i2c to the Subsystem	22
Check Your Environment	27
Configure DW_apb_i2c	28
Complete Signal Connections	29
Generate Subsystem RTL	29
Create Gate-Level Netlist	30
Checking Synthesis Status and Results	33
Synthesis Output Files	34
Running Synthesis from Command Line	34
Create Component GTECH Simulation Model	34
Verify Component	36
Checking Simulation Status and Results	39
Applying Default Verification Attributes	39
Verify the Subsystem	40
Formal Verification	40
Create Testbench	40

Checking Subsystem Verification Status and Results	43
Create a Batch Script	44
Export the Subsystem	44
Chapter 3	
Functional Description	45
Overview	45
I ² C Terminology	47
I ² C Bus Terms	47
Bus Transfer Terms	48
I ² C Behavior	49
I ² C Protocols	50
START and STOP Conditions	50
Addressing Slave Protocol	50
Transmitting and Receiving Protocol	52
START BYTE Transfer Protocol	53
Multiple Master Arbitration	54
Clock Synchronization	55
Operation Modes	56
Slave Mode Operation	56
Master Mode Operation	60
Disabling DW_apb_i2c	62
IC_CLK Frequency Configuration	63
DMA Controller Interface	65
Enabling the DMA Controller Interface	66
Overview of Operation	66
Transmit Watermark Level and Transmit FIFO Underflow	68
Choosing the Transmit Watermark Level	68
Selecting DEST_MSIZ and Transmit FIFO Overflow	69
Receive Watermark Level and Receive FIFO Overflow	70
Choosing the Receive Watermark level	70
Selecting SRC_MSIZ and Receive FIFO Underflow	70
Handshaking Interface Operation	71
APB Interface	74
Chapter 4	
Parameters	75
Parameter Descriptions	75
Configuration Parameters	76
Chapter 5	
Signals	85
DW_apb_i2c Interface Diagram	86
I/O Connections	87
DW_apb_i2c Signal Descriptions	88

Chapter 6

Registers	99
Register Memory Map	100
Registers and Field Descriptions	104
Operation of the Interrupt Registers	135

Chapter 7

Programming the DW_apb_i2c	155
Software Registers	155
Software Drivers	155

Chapter 8

Verification	157
Overview of Vera Tests	157
APB Slave Interface	158
DW_apb_i2c Master Operation	158
DW_apb_i2c Slave Operation	159
DW_apb_i2c Interrupts	159
DMA Handshaking Interface	159
DW_apb_i2c Dynamic IC_TAR and IC_10BITADDR_MASTER Update	159
Generate NACK as a Slave-Receiver	159
SCL Held Low for Duration Specified in IC_SDA_SETUP	159
Generate ACK/NACK for General Call	160
Overview of DW_apb_i2c Testbench	160

Chapter 9

Integration Considerations	163
Digital/Analog Domain Functional Partitioning	163
Reading and Writing from an APB Slave	164
Reading From Unused Locations	164
32-bit Bus System	165
16-bit Bus System	166
8-bit Bus System	166
Write Timing Operation	167
Read Timing Operation	168
Accessing Top-level Constraints	168
.....	169

Appendix A

Building and Verifying Your DW_apb_i2c	171
Setting Up Your Environment	171
Starting coreConsultant	172
Checking Your Environment	173
Configuring the DW_apb_i2c	173
Synthesizing the DW_apb_i2c	174
Checking Synthesis Status and Results	174
Synthesis Output Files	174
Running Synthesis from Command Line	175

Other Synthesis Information	175
Verifying the DW_apb_i2c	175
Creating GTECH Simulation Models	175
Verify the Simulation Model	177
Checking Simulation Status and Results	180
Creating a Batch Script	180
Applying Default Verification Attributes	181
Appendix B	
Database Description	183
Design/HDL Files	184
RTL-Level Files	184
Simulation Model Files	185
Register Map Files	185
Synthesis Files	186
Verification Reference Files	186
Appendix C	
DesignWare QuickStart Designs	187
QuickStart Example Designs	187
Appendix D	
DW_apb_i2c Application Notes	189
Appendix E	
Glossary	191
Index	195

Preface

About This Manual

This databook provides information that you need to interface the DW_apb_i2c to the Advanced Peripheral Bus (APB). The DW_apb_i2c conforms to the *AMBA Specification, Revision 2.0* from ARM.

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

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*.

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.
Chapter 2 “Building and Verifying a Subsystem”	Provides getting started information that allows you to walk through the process of using the DW_apb_i2c with Synopsys’ coreAssembler tool.
Chapter 3 “Functional Description”	Describes the functional operation of the DW_apb_i2c.
Chapter 4 “Parameters”	Identifies the configurable parameters supported by the DW_apb_i2c.
Chapter 5 “Signals”	Provides a list and description of the DW_apb_i2c signals.
Chapter 6 “Registers”	Describes the programmable registers of the DW_apb_i2c.
Chapter 7 “Programming the DW_apb_i2c”	Provides information needed to program the configured DW_apb_i2c.
Chapter 8 “Verification”	Provides information on verifying the configured DW_apb_i2c.
Chapter 9 “Integration Considerations”	Includes information you need to integrate the configured DW_apb_i2c into your design.

Appendix A “Building and Verifying Your DW_apb_i2c”	Provides getting started information that allows you to walk through the process of using the DW_apb_i2c with Synopsys coreConsultant tool.
Appendix B “Database Description”	Provides deliverables and reference files generated from the coreConsultant flow.
Appendix C “DesignWare QuickStart Designs”	Provides the locations of QuickStart examples that integrate most DesignWare AMBA Synthesizable Components into an SoC design that you can simulate.
Appendix D “DW_apb_i2c Application Notes”	Contains information about application notes for the DW_apb_i2c component.
Appendix E “Glossary”	Provides a glossary of general terms.

Typographical and Symbol Conventions

The following conventions 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"
<i>Italic</i> or <i>Italic</i>	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, <i>prod_dir</i> 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 (<i>pin1</i>), but others are optional (<i>[pin2 ... pinN]</i>).
TopMenu > SubMenu	Pulldown menu paths, such as: File > Save As ...

Revision History

This table shows the revision history for the databook from release to release. This is being tracked from version 1.08a onward.

Table 2: Databook Revision History

Version	Databook Date	Description
1.06a	September 19, 2006	
1.08a	March 6, 2007	<ul style="list-style-type: none"> ● Fixed following doc and RTL STARs: 9000087109: Updated the IC_TAR register update rules to remove RdReqMode. 9000099055: Corrected an incorrect description of the RD_REQ bit in the IC_RAW_INTR_STAT register. 9000099184: Remove an incorrect statement in the IC_ENABLE[0] register bit field concerning the disabling of the I²C. The incorrect statement regarding what occurs when I²C is disabled was: “The interrupt bits in the IC_RAW_INTR_STAT register are cleared.” 9000160810: Updated the IC_DMA_TDLR register with the correct width of TX_ABW-1:0. 9000160811: Corrected the reserved bit field of the IC_DMA_TDLR register to be 31:TX_ABW instead of 31: TX_ABW+1. 9000160811: Added an explanation of how the Derived Constants in Table 9 on page 84 are created. ● Removed IC_RX_FULL_GEN_NACK configuration parameter. ● RTL bug fixes. See the DesignWare DW_apb_i2c Release Notes. ● Fixed a “glitch” that was found when DW_apb_i2c generated a RESTART. For more information, see the DesignWare DW_apb_i2c Release Notes.

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

This chapter describes the DesignWare APB I²C Interface Peripheral, referred to as DW_apb_i2c. The DW_apb_i2c component is an AMBA 2.0-compliant Advanced Peripheral Bus (APB) slave device and is part of the family of DesignWare AMBA Synthesizable Components.

The topics included in this chapter are:

- [“DesignWare AMBA System Overview”](#)
- [“General product Description” on page 13](#)
- [“Features” on page 13](#)
- [“Standards Compliance” on page 14](#)
- [“Verification Environment Overview” on page 14](#)
- [“Licenses” on page 14](#)
- [“Where To Go From Here” on page 15](#)

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.

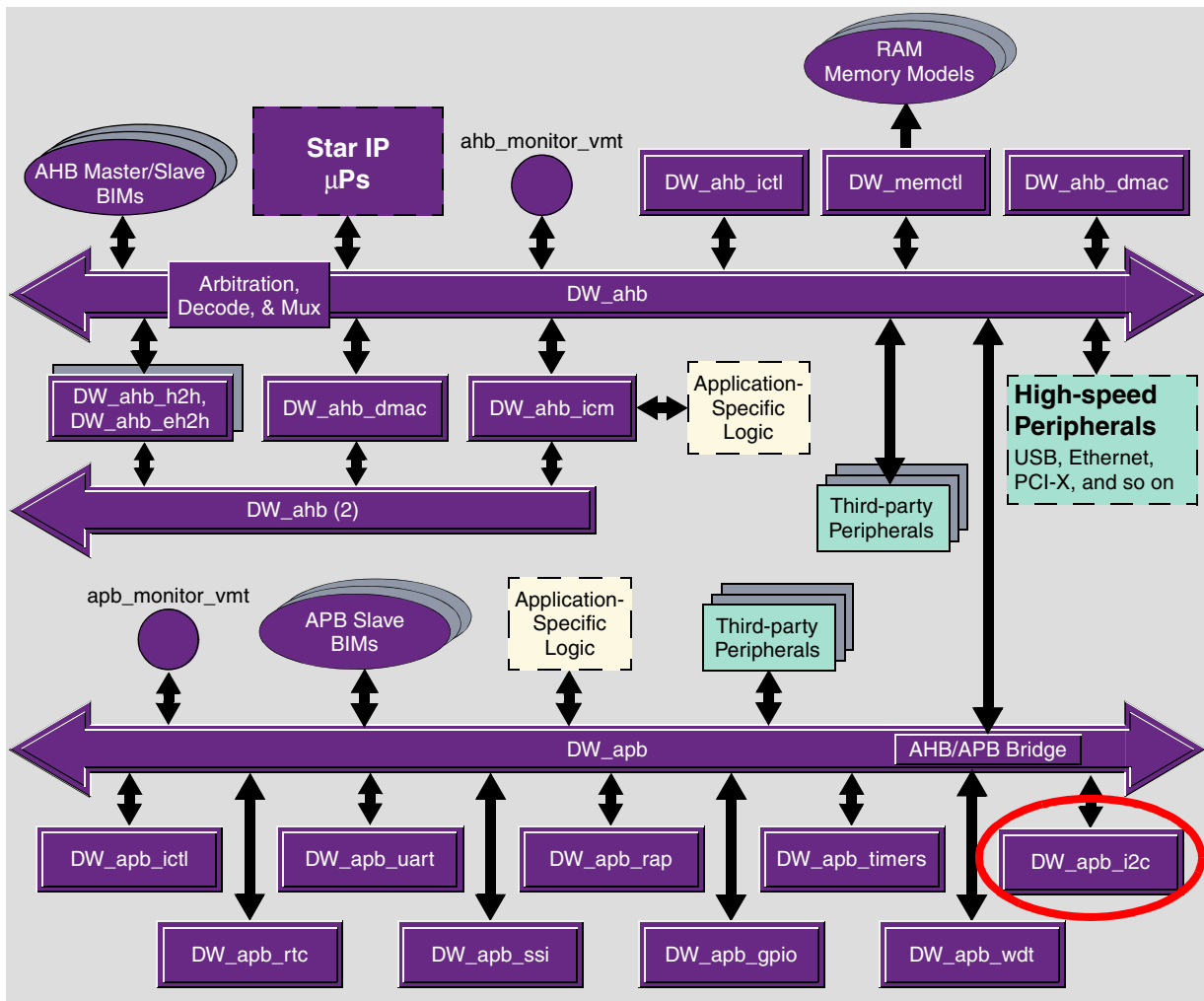


Figure 1: Example of DW_apb_i2c in a Complete System

General product Description

The DW_apb_i2c is a configurable, synthesizable, and programmable control bus that provides support for the communications link between integrated circuits in a system. It is a simple two-wire bus with a software-defined protocol for system control, which is used in temperature sensors and voltage level translators to EEPROMs, general-purpose I/O, A/D and D/A converters, CODECs, and many types of microprocessors.

DW_apb_i2c Block Diagram

Figure 2 illustrates a simple block diagram of DW_apb_i2c. For a more detailed block diagram and description of the component, refer to [Chapter 3, “Functional Description”](#) on page 45.

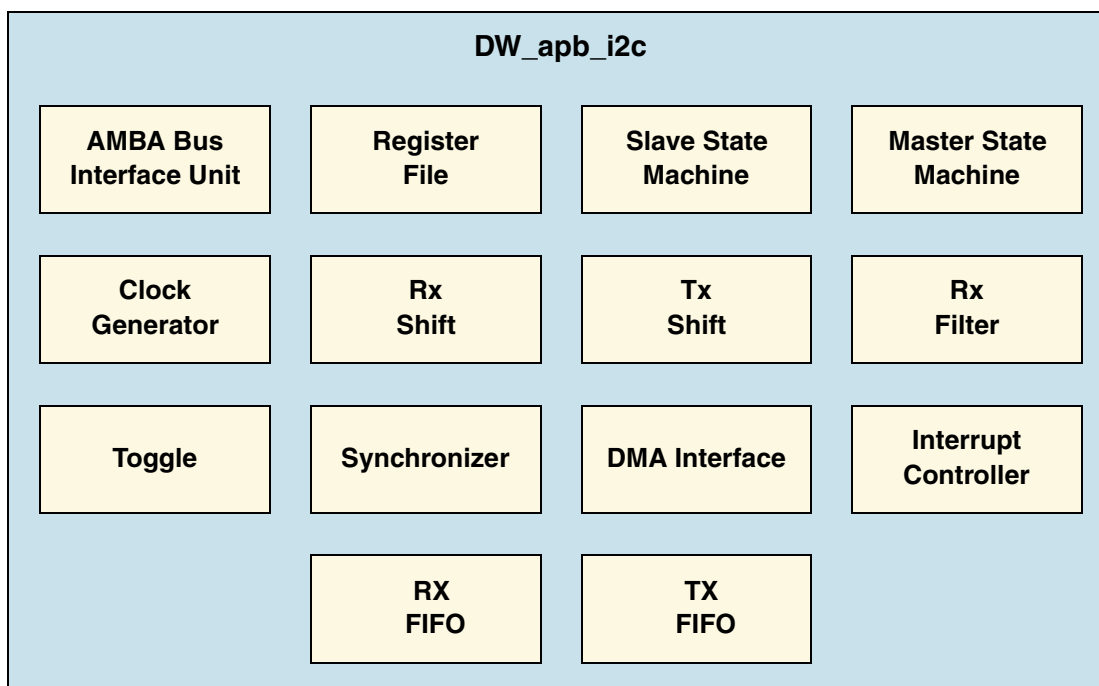


Figure 2: Block Diagram of DW_apb_i2c

Features

DW_apb_i2c has the following features:

I²C Features

- Two-wire I²C serial interface – consists of a serial data line (SDA) and a serial clock (SCL)
- Three speeds:
 - Standard mode (100 Kb/s)
 - Fast mode (400 Kb/s)
 - High-speed mode (3.4 Mb/s)
- Clock synchronization
- Master OR slave I²C operation

- 7- or 10-bit addressing
- 7- or 10-bit combined format transfers
- Bulk transmit mode
- Ignores CBUS addresses (an older ancestor of I²C that used to share the I²C bus)
- Transmit and receive buffers
- Interrupt or polled-mode operation
- Handles Bit and Byte waiting at all bus speeds
- Simple software interface consistent with DesignWare APB peripherals
- Component parameters for configurable software driver support
- DMA handshaking interface compatible with the DW_ahb_dmac handshaking interface

The DW_apb_i2c requires external hardware components as support in order to be compliant in an I²C system. The descriptions are detailed later in this document.

It must also be noted that the DW_apb_i2c should only be operated either as (but not both):

- A sole master in an I²C system and programmed only as a Master; OR
- A slave in an I²C system and programmed only as a Slave.

DesignWare AMBA APB Slave Interface

- Support for APB data bus widths of 8, 16, and 32 bits

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

Standards Compliance

The DW_apb_i2c component conforms to the [AMBA Specification, Revision 2.0](#) from ARM. Readers are assumed to be familiar with this specification.

Verification Environment Overview

The DW_apb_i2c includes an extensive verification environment, which sets up and invokes your selected simulation tool to execute tests that verify the functionality of the configured component. You can then analyze the results of the simulation.

The “[Verification](#)” on [page 157](#) chapter discusses the specific procedures for verifying the DW_apb_i2c.

Licenses

Before you begin using the DW_apb_i2c, 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_i2c 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, coreAssembler 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 3: Tool Comparison

Activity	Recommended Tool
Single Component	
Configuration	coreConsultant
Synthesis	coreConsultant
Verification	coreConsultant
Multiple Components	
Configuration	coreAssembler
Synthesis	coreAssembler
Formal verification	coreAssembler
Creation of top-level subsystem RTL	coreAssembler
Address map creation	coreAssembler
Subsystem simulation	coreAssembler
Creation of subsystem templates	coreAssembler
Importation of non-DesignWare IP	coreAssembler

For more information about implementing your DW_apb_i2c component within a DesignWare AMBA subsystem using coreAssembler, refer to [Chapter 2, “Building and Verifying a Subsystem”](#) on page 17.

For more information about configuring, synthesizing, and verifying just your DW_apb_i2c component, refer to [Appendix A, “Building and Verifying Your DW_apb_i2c”](#) on page 171.

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_i2c component within a simple DesignWare AMBA subsystem using the coreAssembler tool. You use coreAssembler 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.

When used with the DesignWare Library, coreAssembler adds subsystem simulation to the standard coreAssembler functionalities. 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_i2c component, refer to [Appendix A on page 171](#).

The topics in this chapter are as follows:

1. “Setting up Your Environment” on page 17
2. “Overview of the Configuration and Integration Process” on page 18
3. “Start coreAssembler” on page 21
4. “Check Your Environment” on page 27
5. “Add DW_apb_i2c to the Subsystem” on page 22
6. “Configure DW_apb_i2c” on page 28
7. “Complete Signal Connections” on page 29
8. “Generate Subsystem RTL” on page 29
9. “Create Gate-Level Netlist” on page 30
10. “Create Component GTECH Simulation Model” on page 34
11. “Verify Component” on page 36
12. “Verify the Subsystem” on page 40
13. “Create a Batch Script” on page 44
14. “Export the Subsystem” on page 44

Setting up Your Environment

DW_apb_i2c 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 SYNOPSISYS. 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 coreAssembler.

Figure 3 illustrates coreAssembler’s usage flow from invoking the tool to creating a workspace to stepping through the activities in the GUI. Table 4 on page 18 provides a description of the workspace directory and subdirectories.

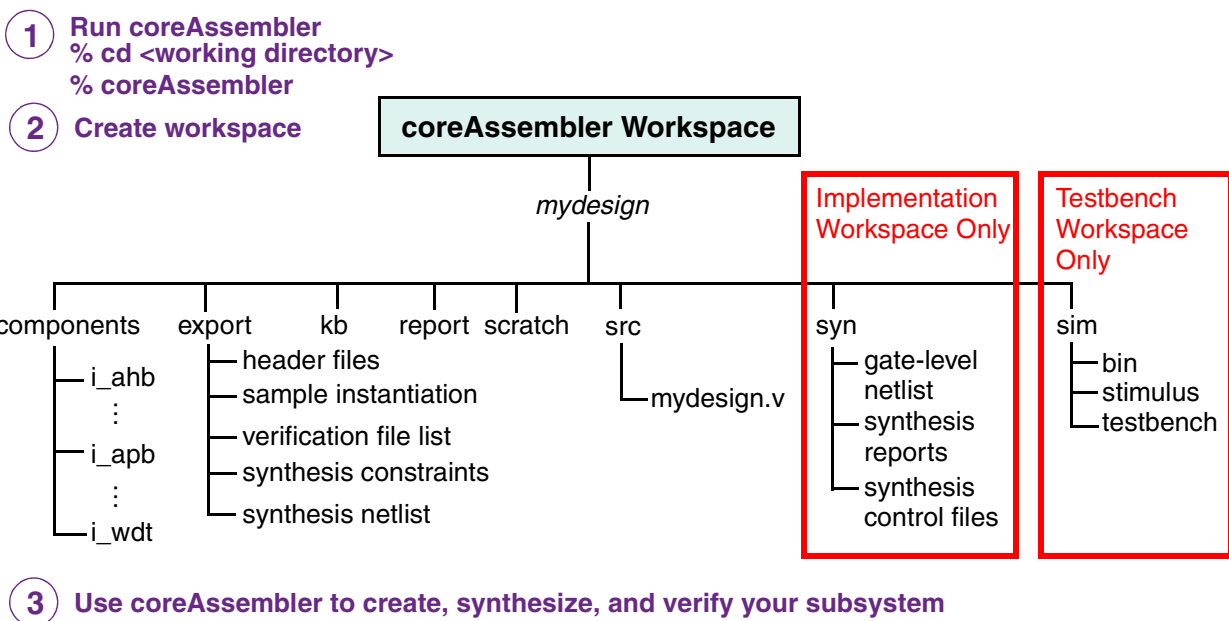


Figure 3: coreAssembler Usage Flow

Table 4: coreAssembler Workspace Directory Contents

Directory/Subdirectory	Description
Directories containing files to be used after exiting coreAssembler.	
export	Contains the files you will need once you exit coreAssembler. These files will be used to integrate the results from the completed source configuration and synthesis activities into your larger system (outside coreAssembler). 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 <i>Using DesignWare Library IP in coreAssembler</i> .
src	Includes the subsystem top-level RTL file, <i>design_name.v</i> .

Table 4: coreAssembler Workspace Directory Contents (Continued)

Directory/Subdirectory	Description
Directories containing files not generally used after exiting coreAssembler.	
components	Includes a directory for each DW AMBA Synthesizable IP instance connected in the subsystem.
components/ <i>instance_name</i>	Contains the data for each IP component instance. This is the instance name of the component used in the design. Each <i>instance_name</i> 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 coreAssembler. These are binary files containing the state of the design.
report	Contains all of the reports created by coreAssembler during build, configuration, test and synthesis phases. An index.html file in this directory links to many of these generated reports.
scratch	Contains temp files used during the coreAssembler processes.
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 coreAssembler.

Table 5: coreAssembler Testbench Workspace Directory Contents

Directory/Subdirectory	Description
Directories containing files to be used after exiting coreAssembler.	
export	Contains the files you will need once you exit coreAssembler. These files will be used to integrate the results from the completed verification activities into your larger system (outside i2c). 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 <i>Using DesignWare Library IP in coreAssembler</i> .
sim/stimulus/ <i>component_name</i>	Contains the test stimulus files in Verilog and C.
sim/testbench/all	Includes the testbench file for the DUT, <i>design_name_tb.v</i> , subsystem source file list, <i>design_name.lst</i> , and simulation execution script <i>run.scr</i> .
src	Includes the testbench top-level RTL file, <i>design_name.v</i> .
Directories containing files not generally used after exiting coreAssembler.	
components	Includes a directory for the DUT, and a directory for each DW AMBA Verification IP connected in the subsystem.
kb	Contains knowledge base information used by coreAssembler. These are binary files containing the state of the design.

Table 5: coreAssembler Testbench Workspace Directory Contents (Continued)

Directory/Subdirectory	Description
report	Contains all of the reports created by coreAssembler during testbench build and configuration phases. An index.html file in this directory links to many of these generated reports.
scratch	Contains temp files used during the coreAssembler processes.
sim	Includes simulation files for the testbench. This directory is created when you complete the Simulate Subsystem activity.
sim/testbench/all/cov_results	Includes the various coverage results of the verified subsystem
syn	This directory is created but not populated for the Create Testbench activity group in coreAssembler.

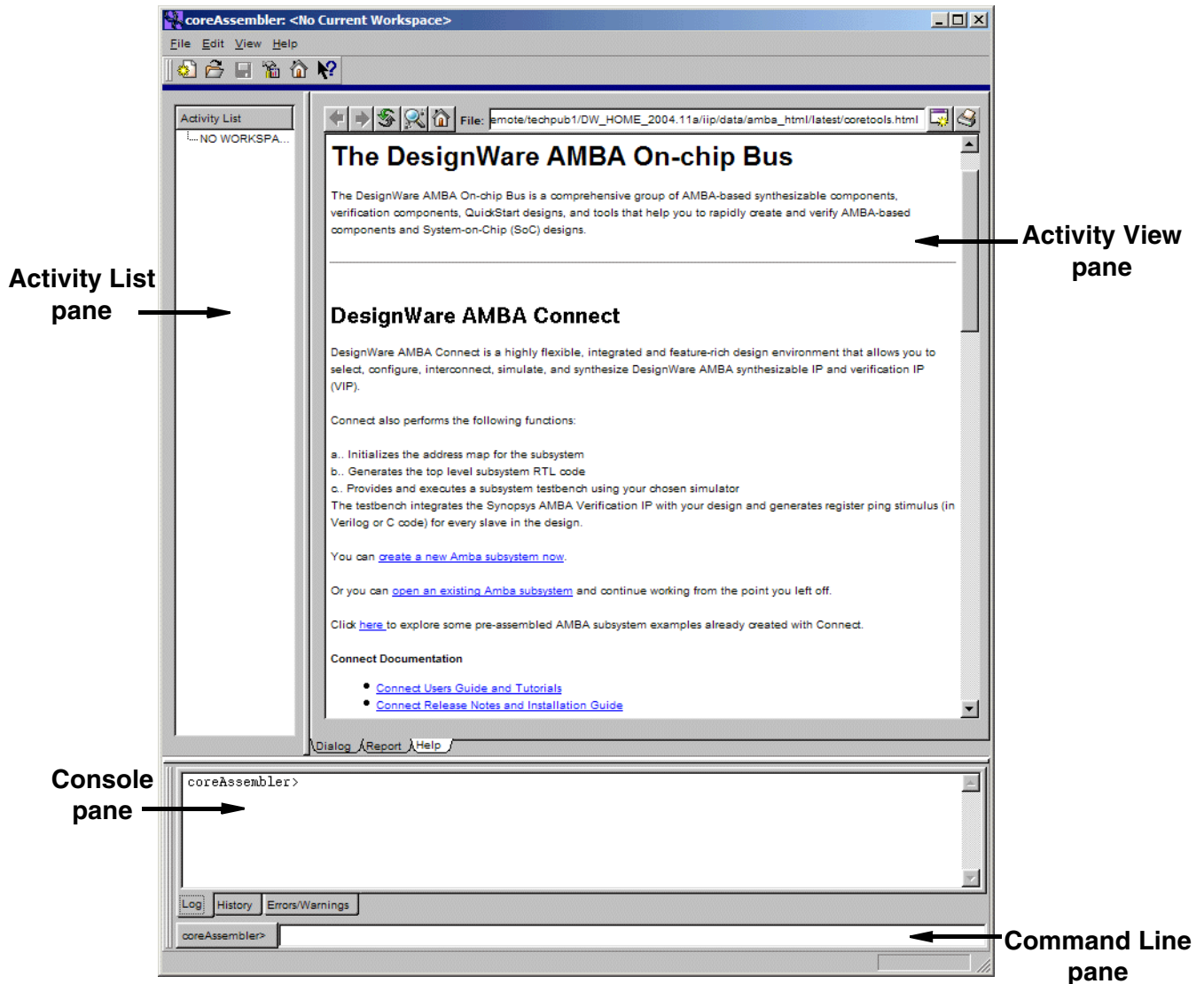
Start coreAssembler

To invoke coreAssembler:

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

At this point, coreAssembler creates in the workspace an export directory that will eventually contain the files you need once you exit coreAssembler. For an explanation of this directory, see [Table 4 on page 18](#). You can use these files for your own chip-level synthesis and simulation. A README file and an index.html file in this directory (created after you add components) both describe all of the exported files in this directory.

In the coreAssembler GUI, an empty schematic window is displayed and the Add Subsystem Components activity is highlighted under the Create RTL category in the Activity List on the left.

For more information about coreAssembler, refer to the [coreAssembler User Guide](#). For more in-depth tutorials, refer to the “[DesignWare Library IP in coreAssembler Tutorials](#)” chapter of *Using DesignWare Library IP in coreAssembler*. For tables that list the contents of the export directory at each step of the Subsystem assembly process, refer to “[Export Directory](#)” in *Using DesignWare Library IP in coreAssembler*.

Add DW_apb_i2c to the Subsystem

In a minimal subsystem using the DW_apb_i2c 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_i2c, 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_i2c in a simple subsystem.

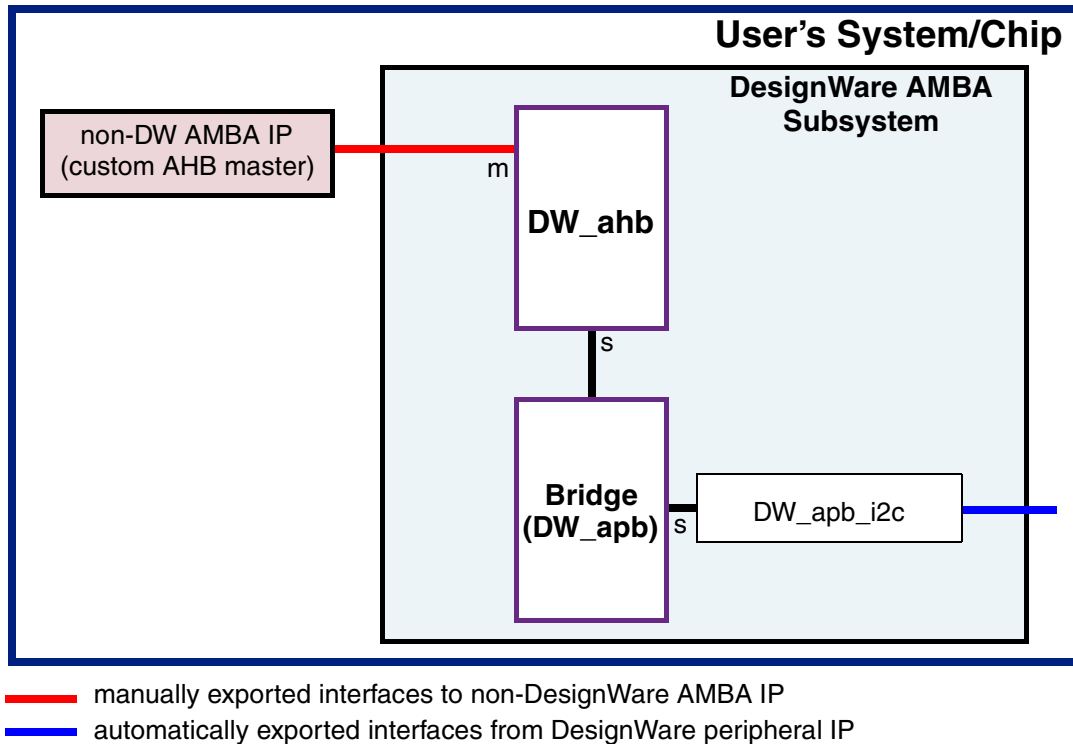



Figure 4: DW_apb_i2c in Simple Subsystem

The following procedure steps you through the process of creating a simple subsystem with a DW_apb_i2c component.

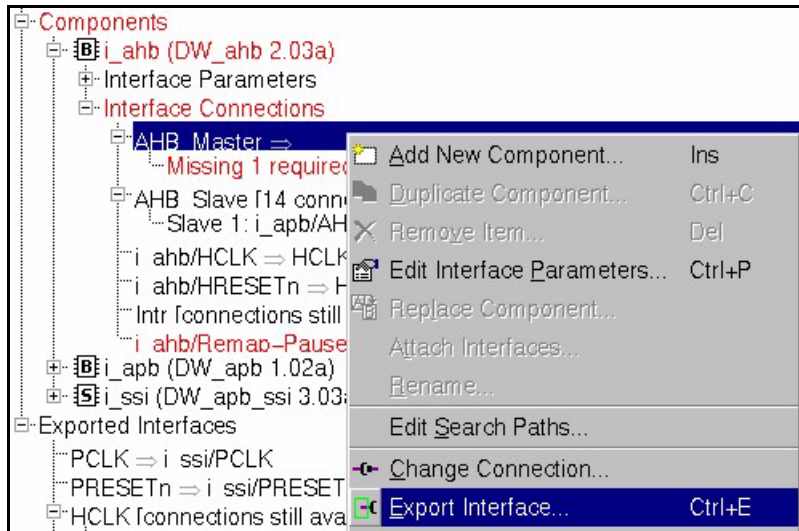
1. Use the **Schematic > Add New Component** menu item to display the Add Component Instance to Subsystem dialog; alternatively, you can right-click in the schematic window and choose **Add New Component** from the popup menu or use the Insert key.
2. In the Add Component Instance to Subsystem dialog, click on the specific component to add it: DW_apb, DW_ahb, DW_apb_i2c. 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.

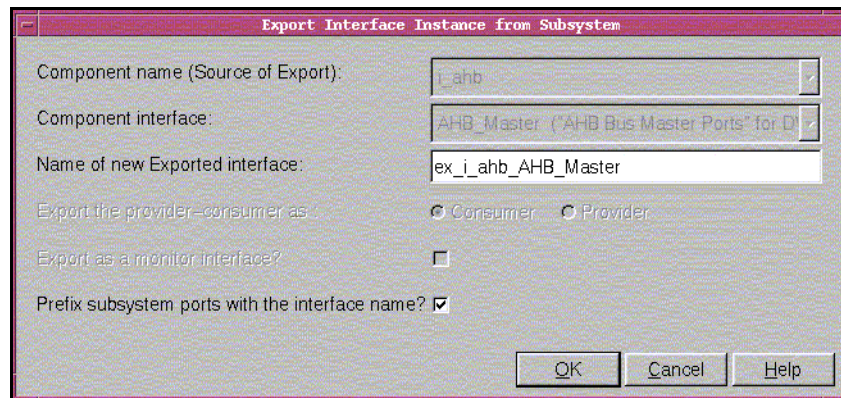
3. Notice that the DW_ahb instance is red in the schematic view. Toggle over to the tree view by clicking the toggle icon  on the toolbar and expand the i_ahb component instance. The AHB Master line in the Interface Connections says that it is missing a connection, and the i_ahb/Remap-Pause line shows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

First, you are going to export an AHB master interface from the DW_ahb.

- 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.



Note

There are two types of configuration: that which affects external interfaces, and that which doesn't. Changing address and/or data bus widths, and endianness, affects external interfaces. These configuration changes must be completed during the Add Subsystem Components activity, since they affect other subsystem components as well. Later you will use the Configure Components activity to change a component's internal configuration.

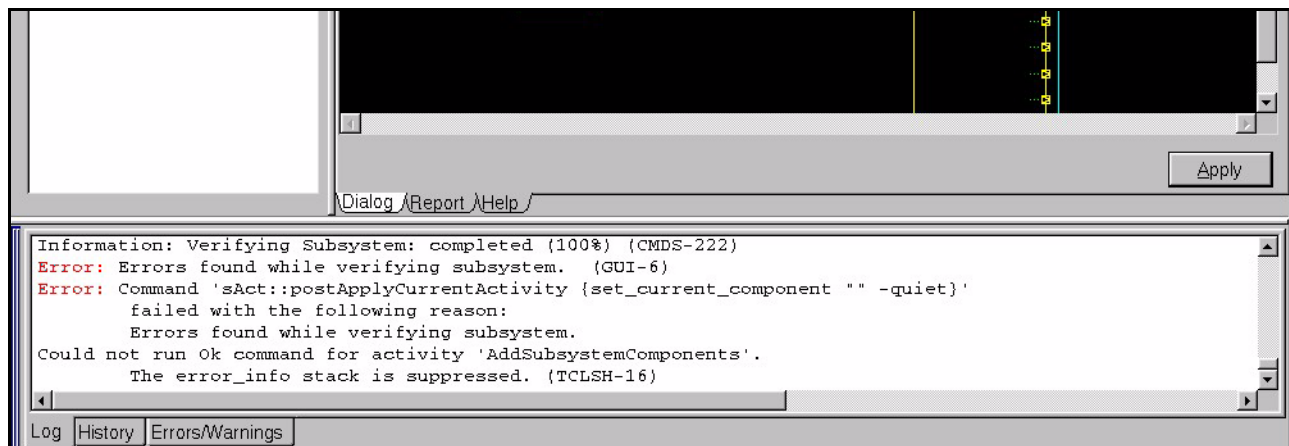
At this stage, you can configure DW_apb_i2c to include an interrupt output for every interrupt. You can also specify the polarity of the interrupts.

These parameters are Interface Parameters because changing them may affect connectivity with other subsystem components. Therefore, you must determine these settings at the subsystem configuration level. Later in “Configure DW_apb_i2c” on page 28, Configuration Parameters are defined. These are component-level parameters and do not affect interfaces to other components.

For demonstration purposes, change the configuration so that the component has a individual signals for all of the interrupts. To change this setting, do the following:

- a. Right-click on the DW_apb_i2c component (i_i2c) in the tree view and choose the **Edit Interface Parameters** menu item to display the i_i2c Interface Configuration dialog.
 - b. Click the check box in the “Single Interrupt output port present” field.
 - c. Click OK.
5. You now have a rudimentary subsystem that includes the DW_apb_i2c 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 signal 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.



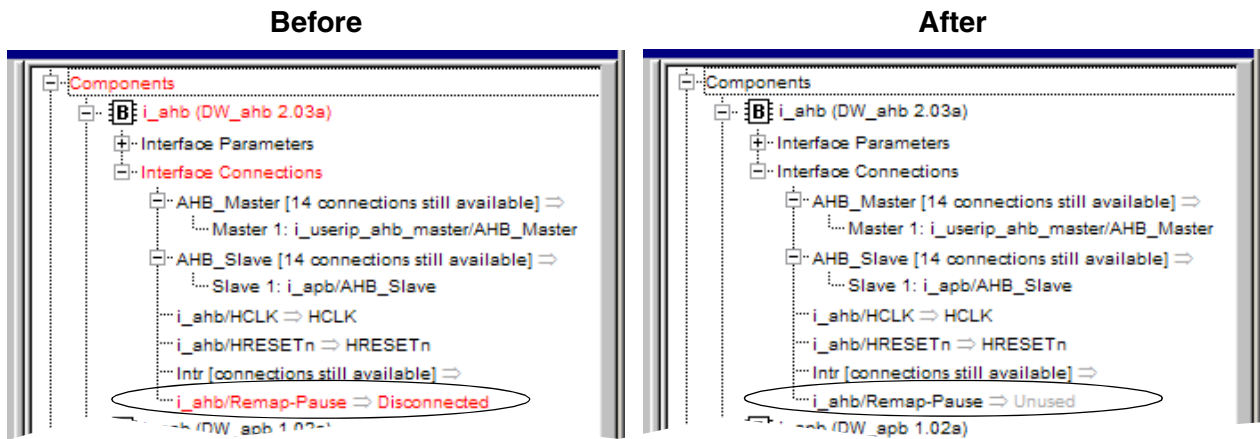
Note

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
```

6. Because you do not need the remap/pause feature in this subsystem, 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.



7. Click the Apply button again to complete the Add System Components activity.
8. 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_i2c” on page 28](#). The coreAssembler tool 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: <ul style="list-style-type: none"> ● 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.



Attention

If you are using a newer version of coreAssembler, you are presented an IP Update Check report window, comparing your components to those in your DESIGNWARE_HOME tree, and also the latest currently available from Synopsys (via the web). STAR updates are also listed in this report, to help you determine if you need to make an update. Viewing STARs and downloading components from Synopsys requires SolvNet authentication.

The first time you use this feature, you are prompted to enable automatic update checking, and to specify the interval between checks. You can change these preferences at any time using the **Edit > Preferences** menu item.

For more information on the IP Update Checking feature, refer to “Component Update Checking” in the *coreAssembler User Guide*.

9. coreAssembler 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 familiarize yourself with the information in all reports before going to any new activity.

Check Your Environment

Before you begin configuring your components, it is recommended that you check your environment to ensure you have the latest tool versions installed and your environment variables set up correctly. You must have at least one DesignWare Library component instantiated in your workspace for this environment check feature to appear.

To check your environment, use the **Help > Help for component /comp > 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 required.



Note

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.

Configure DW_apb_i2c

This section steps you through the tasks that configure the component-level parameters (configuration parameters) for DW_apb_i2c in coreAssembler. For this exercise, you will not configure any of the 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_i2c; click on that item in the Activity List.
2. Click the DW_apb_i2c item (also called i_i2c) to display the Top Level Parameters window. Notice that several parameters are greyed out, such as APB data bus width.

Some of these parameters are interface parameters—parameters that may affect component interfaces. Interface parameters are always defined during the Add Subsystem Components activity. In the previous exercise, you changed the interrupt pinout configuration to a single combined interrupt.

If you were to decide at this point to change any of the interface parameters, you would simply click on the Add Subsystem Components activity, again click on the i_i2c component in either the schematic or tree views, and then change and apply the new interface parameter values.

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 0x00000000 and that the end address is 0x000043ff, which is the same as the start and end addresses of the DW_apb_i2c, 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.

All DW_apb_i2c parameters are explained in detail in [“Parameters” on page 75](#).

4. Click the Apply button to the default configuration parameters. coreAssembler 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_i2c. 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. Examine the Manual Connect and Manual Disconnect tabs; leave the defaults and Apply the dialog.

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. coreAssembler 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`. You can also choose whether to include testbench probe signals.
4. You can optionally insert comment text into the comment field that will be inserted into the header of each subsystem-level RTL file.



Note

This dialog only selects the HDL language for the top-level RTL for the subsystem. For all DesignWare Synthesizable Components for AMBA 2, the component RTL is written in Verilog.

- Click Apply. Regardless of whether you use a Verilog or VHDL simulator, coreAssembler 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*. coreAssembler 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.
<i>workspace.lst</i>	List of source files in proper analysis order for entire subsystem.
<i>workspace_comp.vhd</i>	VHDL component declaration for subsystem.
<i>workspace_inst.v</i>	Verilog Testbench template; example subsystem instantiation.
<i>workspace_inst.vhd</i>	VHDL Testbench template; example subsystem instantiation.
<i>workspace_params.h</i>	C subsystem configuration information.
<i>workspace_params.v</i>	Verilog subsystem configuration information.
<i>workspace_params.vhd</i>	VHDL subsystem configuration information.

- 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 coreAssembler 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.

- 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.
 - Primitime (*pt_shell*) – Enables Primitime 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.
 - Tetramax (*tmax*) – Specifies the path to the Tetramax utility that is used with ATPG.

- VCS (vcs) – Specifies the path to the VCS simulator.
- VCSI (vcsi) – Specifies the path to the VCSI simulator.
- Vera (vera) – Specifies the path to the Vera used for VIP simulation.
- MTI ModelSim (vsim) – Specifies the location of the ModelSim simulator
- NC Verilog/VHDL (ncsim) – Specifies the location of the NC Verilog/VHDL simulator.

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`. You can also specify simulator paths during setup for the verification activities.

Select the “64 Bit?” checkbox if the 64-bit version of the tool is needed (and available).

Cancel the “Set tool installation roots” to examine the following activities.

2. **Specify Target Technology** – `coreAssembler` analyzes the target technology library and uses it to generate a synthesis strategy that is optimized for your technology library. A separate specification exists for Logical (`dc_shell`) and Physical (`psyn_shell`) libraries by choosing the appropriate tab. For the Logical Library paths, a target and a link library path must be specified for `dc_shell`; otherwise, errors occur in `coreAssembler`.

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. **Initialize Subsystem Constraints** – In this activity, you can review and modify any existing subsystem-level clocks and then initialize subsystem constraints from component constraints. Click Apply and familiarize yourself with the resultant report.
4. **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.
5. **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.
6. **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.
7. **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*.

8. **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. This activity only defines the test methodology. Design for Test insertion is enabled or disabled in the Synthesis activity, explained next. Click Apply and look at the report, which gives design-for-test information.
9. **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 Power Compiler 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 at the top of the configuration screen. Look at the values for the parameters listed below.

Field Name	Description
Execution Options	
Generate Scripts only?	<p>Values: Enable or Disable</p> <p>Default Value: Disable</p> <p>Description: Writes the run.scr script, but it is not run when you click Apply. To run the script, go to the <i>workspace/syn</i> directory and run the script (run.scr) from the Unix command line.</p>
Run Style	<p>Values: local, lsf, grd, or remote</p> <p>Default Value: local</p> <p>Description: Describes how to run the command: locally on the current machine, through LSF, through GRD, or through the remote shell command. Jobs can be executed on different machines, but must be run on the same operating system as the current operating system.</p>
Run Style Options	<p>Values: user-defined</p> <p>Default Value: none</p> <p>Description: Additional options for the run style options except local. For remote, specify the hostname. For LSF and GRD, specify bsub or qsub commands.</p>

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 coreAssembler. 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.
10. **Generate Test Vectors** – This option allows you to generate ATPG test vectors with TetraMax after you have used insert DFT during the Synthesis activity. 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; coreAssembler 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. For more information about coreAssembler synthesis and synthesis stages, see the [coreAssembler User Guide](#).

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 183](#).

Running Synthesis from Command Line

To run synthesis from the command line prompt for the files generated by `coreAssembler`, 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 either:

- encrypted format (when using a DesignWare license which is provided with the DesignWare Library product) or
- RTL source format (when using a DesignWare AMBA Synthesizable IP source license).



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_i2c` component, click on the Generate GTECH Model (for `i_i2c`) activity.

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

Field Name	Description
Execution Options	
Generate Scripts only?	<p>Values: Enable or Disable</p> <p>Default Value: Disable</p> <p>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 <i>workspace/components/DW_apb_i2c instance/gtech</i> directory and run the run.scr script from the Unix command line.</p>
Run Style	<p>Values: local, lsf, grd, or remote</p> <p>Default Value: local</p> <p>Description: Describes how to run the command: locally on the current machine, through lsf, through grd, or through the remote shell command. Jobs can be executed on different machines, but must be run on the same operating system as the current operating system.</p>
Run Style Options	<p>Values: user-defined</p> <p>Default Value: none</p> <p>Description: Additional options for run style options other than local. For remote, specify the hostname. For lsf and grd, specify bsub or qsub command options.</p>
Send e-mail	<p>Values: current user's name</p> <p>Description: E-mail is sent when the command script completes or is terminated.</p>
Synthesis Control	
Ungroup Netlist after Compile	<p>Values: Enable or Disable</p> <p>Default Value: Disable</p> <p>Description: Ungroups the design to provide a non-hierarchical netlist.</p>



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.

- Click Apply. coreAssembler 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_i2c/gtech/gtech.log

```

Your simulation model is contained in the DW_apb_i2c.v output file that is written to *workspace/components/i_i2c/gtech/qmap/db*.

Verify Component

The Verify Component activity in coreAssembler allows you to perform verification for an individual component. For this exercise, you are just going to perform verification for the DW_apb_i2c; however, you typically would also perform verification for other components in your subsystem.

To verify DW_apb_i2c, use coreAssembler to complete the following steps:

- To run verification for the DW_apb_i2c component, click Specify and Run Simulations (for i_i2c) in the Verify Component activity.
- 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 34](#)) only if you are using a non-VCS simulator and do not have a source license.

Field Name	Description
View Selection	Values: user-defined Default Value: RTL Description: Determines which design view to simulate: RTL or GTECH.

- In the VIP pane, click on the VMT and AMBA versions to see the available versions; leave these in the default “latest” mode.
- Specify the various options for the Simulator.
 - In the Select Simulator area, click on the Simulator list item to view available simulators (VCS is the default).
 - 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_i2c 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 Path	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.

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.

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

Field Name	Description
Do Not Launch Simulation	<p>Values: Enable or Disable</p> <p>Default Value: Disable</p> <p>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 (<i>workspace/sim/run.scr</i>) from the UNIX command line or by repeating the Verification activity with the Do Not Launch Simulation option unselected.</p>
Run Style	<p>Values: local, lsf, grd, or remote</p> <p>Default Value: local</p> <p>Description: Describes how to run the command: locally, through lsf, through grd, or through the remote shell.</p>
Run Style Options	<p>Values: user-defined</p> <p>Default Value: none</p> <p>Description: Additional options for run style other than local. For remote, specify the hostname. For lsf and grd, specify bsub or qsub commands.</p>
Send e-mail	<p>Values: current user's name</p> <p>Description: E-mail is sent to the specified user when the command script completes or is terminated.</p>

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

Field Name	Description
Let each Test decide default Timeout Period	<p>Values: Enable or Disable</p> <p>Default Value: Enable</p> <p>Description: Allows the test to default the timeout period value.</p> <p>Note: It is highly recommended that you leave this option enabled if you want the simulation to complete normally.</p>
Number of clocks before simulation timeout	<p>Minimum Value: 1</p> <p>Default Value: 999999</p> <p>Dependencies: This setting is activated when the "Let each Test decide default Timeout Period" is unchecked.</p> <p>Description: Enabled if default timeout period is disabled. Enter the number of clock periods of simulation that, if passed, causes the simulation to fail. This is used to avoid runaway simulations or to debug truncated simulation runs. Note: If you experience a timeout during the simulation for your specific configuration, you may need to increase this value.</p>
APB Clock Ratio	<p>Values: 1-8 (currently only 1 is allowed)</p> <p>Default Value: 1</p> <p>Description: Specifies the ratio of the APB clock (also known as pclk or the system clock).</p>
Run test_i2c	<p>Values: Enable or Disable</p> <p>Default Value: Enable</p> <p>Description: Tests functionalities of DW_apb_i2c.</p>

7. Click Apply to run the simulation.

When you click Apply, coreAssembler performs the following actions:

- Sets up the DW_apb_i2c verification environment to match your selected DW_apb_i2c configuration.
- Generates the simulation run script (run.scr) and writes it to your *workspace/components/i_i2c/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_i2c/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 157](#).

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; coreAssembler displays a dialog that indicates:

- Your selected Run Style (local, Isf, 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_i2c bus transactions

This information indicates whether the simulation executed successfully, and lists the DW_apb_i2c 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_i2c 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 coreAssembler 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:

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

Create Testbench

The Create Testbench activity allows you to create a separate verification workspace where you simulate your subsystem. When you use DesignWare Library components, many of the steps can be completed automatically. These steps include:

- Generate Subsystem RTL for the design subsystem
- Save and close the design subsystem workspace
- Open a new Testbench workspace
- Instantiate the DUT (from the subsystem design workspace)
- Instantiate Verification IP models for exported master and slave interfaces
- Attach Verification IP to device communication ports (GPIO, SSI, UART)
- Instantiate Verification monitors (optional) for each type of bus (AXI, AHB, APB) used

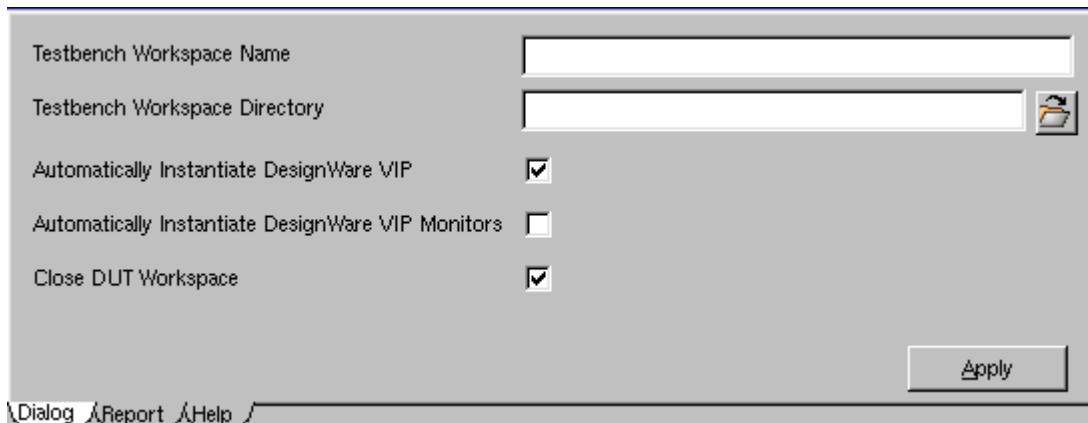
Prior to performing a simulation run, you can:

- Add or remove components in the testbench
- Add a clock tree and specify testbench clocking
- Change configuration information for the VIP components (not for the DUT)
- Make monitor connections (probes) to lower levels of hierarchy inside the DUT

To create the testbench:

1. Click on the “Create Testbench” activity in the Subsystem Activity list.

A dialog gives you configuration options for how to set up the testbench workspace:



These options include:

Table 6: Create Testbench Options

Option text string	Type	Comment
Testbench Workspace Name	string	Default: tb_<DUT_wksp_name>
Testbench Workspace Directory	string	Default: current working dir
Automatically Instantiate DesignWare VIP	boolean	Default: TRUE (checked)
Automatically Instantiate DesignWare VIP Monitors	boolean	Default: FALSE (unchecked)
Close DUT workspace	boolean	Default: TRUE (checked)

2. Apply this activity using the defaults shown. The following automatically generated steps occur.
 - Generates Subsystem RTL for the design subsystem DUT
 - Saves and closes the design subsystem workspace
 - Open a new Testbench workspace
 - Instantiate the DUT (from the subsystem design workspace)
 - Instantiates Verification IP models and connects to exported master and slave interfaces

When this activity completes, you should see the schematic view of your testbench in the new “testbench” workspace, and the Add Testbench Components activity is highlighted.

3. A new menu item “Testbench” is now available after the “Help” menu. Choose the **Add Monitors** item from this menu.

coreAssembler creates a hierarchical block with the appropriate Monitor VIP components connected to the top-level interfaces and clocks.

4. Double click on the Hierarchical cell containing the Monitor(s). Note the connections that are made from the monitor(s) to the top level.

Note: Monitors add overhead to any simulation. If they are not necessary, you will achieve higher simulation performance without them.

5. Choose **Schematic Menu > Exit Cell** to return to the top-level schematic.

6. Click on the Simulate Subsystem activity in order to auto-complete configuring and generating testbench HDL. Click “Yes” to auto-completing the remaining activities.

If you wanted to make configuration changes to the Verification components, you could step through each of these activities individually. By using the auto-complete feature, coreAssembler places default values for these activities, and proceeds to the selected activity.

When Simulate Subsystem is the current activity, the Simulate Subsystem dialog is presented with three tab options:

7. Choose the Testbench Definition tab to determine which slaves you want to be tested by which master.

If there were multiple masters, you choose the master that will test each slave or component. You can also choose “Do Not Test” component entry to bypass testing of a component.

8. In the Execution Options tab, you can specify the following settings.

Field Name	Description
Execution Options	
Generate Scripts only?	<p>Values: Enable or Disable Default Value: Disable Description: Writes scripts that run the simulation, but they are not run when you click Apply. To run these scripts, go to the <i>testbench workspace/sim/testbench/all</i> directory of the testbench workspace and run the run.scr script from the Unix command line.</p>
Run Style	<p>Values: local, lsf, grd, or remote Default Value: local Description: Describes how to run the command: locally on the current machine, through lsf, through grd, or through the remote shell command. Jobs can be executed on different machines, but must be run on the same operating system as the current operating system.</p>
Run Style Options	<p>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 command options.</p>
Send e-mail To:	<p>Values: enable or disable Values: current user’s e-mail address Description: E-mail is sent when the command script completes or is terminated.</p>

9. Click the Simulator Setup tab and look at the parameters for the simulator setup, as detailed in the following table.

Field Name	Description
Simulator	Values: VCS, MTI_Verilog, NC_Verilog Default Value: VCS 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; coreAssembler automatically detects this path. The NC-Verilog executables reside in the ./bin subdirectory.
Generate 'waves' file	Values: Enable or Disable Default Value: Enabled 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.

10. Click Apply to run the subsystem simulation.

Checking Subsystem Verification Status and Results

To check subsystem simulation status and results, click the Report tab. As for the component simulation, coreAssembler 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
- coreAssembler 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 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 coreAssembler GUI screen, enter the following:

```
% source batch.tcl
```

Or restart coreAssembler, specifying the batch script on the Unix command line, like this:

```
% coreAssembler -f 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 behavior of DW_apb_i2c in more detail. The topics included in this chapter are:

- “Overview”
- “I2C Terminology” on page 47
- “I2C Protocols” on page 50
- “Multiple Master Arbitration” on page 54
- “Clock Synchronization” on page 55
- “Operation Modes” on page 56
- “IC_CLK Frequency Configuration” on page 63
- “DMA Controller Interface” on page 65
- “APB Interface” on page 74

Overview

The I²C bus is a two-wire serial interface, consisting of a serial data line (SDA) and a serial clock (SCL). These wires carry information between the devices connected to the bus. Each device is recognized by a unique address and can operate as either a “transmitter” or “receiver,” depending on the function of the device. Devices can also be considered as masters or slaves when performing data transfers. A master is a device that initiates a data transfer on the bus and generates the clock signals to permit that transfer. At that time, any device addressed is considered a slave.



Note

The DW_apb_i2c must only be programmed to operate in either master OR slave mode only. Operating as a master and slave simultaneously is not supported.

The DW_apb_i2c module can operate in standard mode (with data rates up to 100 Kb/s), fast mode (with data rates up to 400 Kb/s), and high-speed mode (with data rates up to 3.4 Mb/s). The DW_apb_i2c can communicate with devices only of these modes as long as they are attached to the bus. Additionally, high-speed mode and fast mode devices are downward compatible. For instance, high-speed mode devices can communicate with fast mode and standard mode devices in a mixed-speed bus system; fast mode devices can communicate with standard mode devices in 0 to 100 Kb/s I²C bus system. However, standard mode devices are not upward compatible and should not be incorporated in a fast-mode I²C bus system as they cannot follow the higher transfer rate and unpredictable states would occur.

An example of high-speed mode devices are LCD displays, high-bit count ADCs, and high capacity EEPROMs. These devices typically need to transfer large amounts of data. Most maintenance and control applications, the common use for the I²C bus, typically operate at 100 kHz (in standard and fast modes).

Any DW_apb_i2c device can be attached to an I²C-bus and every device can talk with any master, passing information back and forth. There needs to be at least one master (such as a microcontroller or DSP) on the bus but there can be multiple masters, which require them to arbitrate for ownership. Multiple masters and arbitration are explained later in this chapter.



Note

In an I²C environment with multiple masters, make sure the DW_apb_i2c is programmed to operate only as a Slave.

The DW_apb_i2c is made up of an AMBA APB slave interface, an I²C interface, and FIFO logic to maintain coherency between the two interfaces. A simplified block diagram of the component is illustrated in [Figure 5](#).

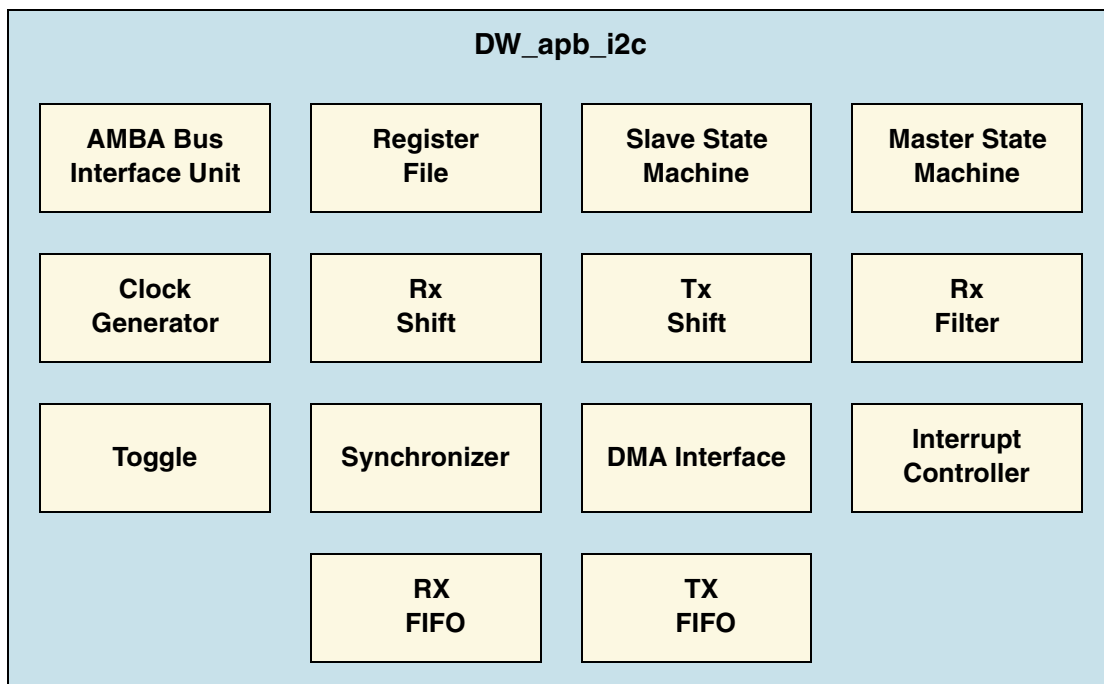


Figure 5: DW_apb_i2c Block Diagram

The following define the file names and functions of the blocks in [Figure 5](#):

- AMBA Bus Interface Unit—DW_apb_i2c_biu.v—Takes the APB interface signals and translates them into a common generic interface that allows the register file to be bus protocol-agnostic.
- Register File—DW_apb_i2c_regfile—Contains configuration registers and is the interface with software.
- Slave State Machine—DW_apb_i2c_slv fsm—Follows the protocol for a slave and monitors bus for address match.

- Master State Machine—DW_apb_i2c_mstfsm—Generates the I²C protocol for the master transfers.
- Clock Generator—DW_apb_i2c_clk_gen.v—Calculates the required timing to do the following:
 - Generate the SCL clock when configured as a master
 - Check for bus idle
 - Generate a START and a STOP
 - Setup the data and hold the data
- Rx Shift—DW_apb_i2c_rx_shift—Takes data into the design and extracts it in byte format.
- Tx Shift—DW_apb_i2c_tx_shift—Presents data supplied by CPU for transfer on the I²C bus.
- Rx Filter—DW_apb_i2c_rx_filter—Detects the events in the bus; for example, start, stop and arbitration lost.
- Toggle—DW_apb_i2c_toggle—Generates pulses on both sides and toggles to transfer signals across clock domains.
- Synchronizer—DW_apb_i2c_sync—Transfers signals from one clock domain to another.
- DMA Interface—DW_apb_i2c_dma—Generates the handshaking signals to the central DMA controller in order to automate the data transfer without CPU intervention.
- Interrupt Controller—DW_apb_i2c_intctl—Generates the raw interrupt and interrupt flags, allowing them to be set and cleared.
- RX FIFO/TX FIFO—DW_apb_i2c_fifo—Holds the RX FIFO and TX FIFO register banks and controllers, along with their status levels.



Note

The `ic_clk` frequency must be greater than or equal to the `pclk` frequency. This restriction occurs because the configuration registers are programmed on `pclk`, and the peripheral enable is the last bit to be programmed; it is then transferred to the other domain, which validates the other bits.

I²C Terminology

The following terms are used throughout this manual and are defined as follows:

I²C Bus Terms

The following terms relate to how the role of the I²C device and how it interacts with other I²C devices on the bus.

Transmitter – the device that sends data to the bus. A transmitter can either be a device that initiates the data transmission to the bus (a *master-transmitter*) or responds to a request from the master to send data to the bus (a *slave-transmitter*).

Receiver – the device that receives data from the bus. A receiver can either be a device that receives data on its own request (a *master-receiver*) or in response to a request from the master (a *slave-receiver*).

Master — the component that initializes a transfer (START command), generates the clock (SCL) signal and terminates the transfer (STOP command). A master can be either a transmitter or a receiver.

Slave – the device addressed by the master. A slave can be either receiver or transmitter.

These concepts are illustrated in [Figure 6 on page 48](#)

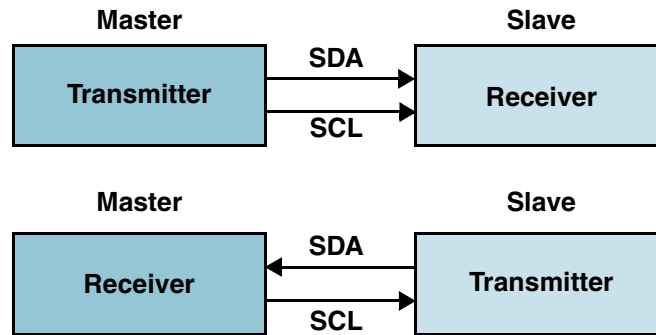


Figure 6: Master/Slave and Transmitter/Receiver Relationships

Multi-master – the ability for more than one master to co-exist on the bus at the same time without collision or data loss.

Arbitration – the predefined procedure that authorizes only one master at a time to take control of the bus. For more information about this behavior, refer to [“Multiple Master Arbitration” on page 54](#).

Synchronization – the predefined procedure that synchronizes the clock signals provided by two or more masters. For more information about this feature, refer to [“Clock Synchronization” on page 55](#).

SDA – data signal line (Serial Data)

SCL – clock signal line (Serial Clock)

Bus Transfer Terms

The following terms are specific to data transfers that occur to/from the I²C bus.

START (RESTART) – data transfer begins with a START or RESTART condition. The level of the SDA data line changes from high to low, while the SCL clock line remains high. When this occurs, the bus becomes busy.



Note

START and RESTART conditions are functionally identical.

STOP – data transfer is terminated by a STOP condition. This occurs when the level on the SDA data line passes from the low state to the high state, while the SCL clock line remains high. When the data transfer has been terminated, the bus is free or idle once again. The bus stays busy if a RESTART is generated instead of a STOP condition.

I²C Behavior

The DW_apb_i2c can be controlled via software to be either:

- The sole I²C master only, communicating with other I²C slaves; OR
- An I²C slave only, communicating with one more I²C masters.

The master is responsible for generating the clock and controlling the transfer of data. The slave is responsible for either transmitting or receiving data to/from the master. The acknowledgement of data is sent by the device that is receiving data, which can be either a master or a slave. As mentioned previously, the I²C protocol also allows multiple masters to reside on the I²C bus and uses an arbitration procedure to determine bus ownership.



Note

In a multi-master environment, the DW_apb_i2c should only be allowed to operate as a Slave only.

Each slave has a unique address that is determined by the system designer. When a master wants to communicate with a slave, the master transmits a START/RESTART condition that is then followed by the slave's address and a control bit (R/W) to determine if the master wants to transmit data or receive data from the slave. The slave then sends an acknowledge (ACK) pulse after the address.

If the master (master-transmitter) is writing to the slave (slave-receiver), the receiver gets one byte of data. This transaction continues until the master terminates the transmission with a STOP condition. If the master is reading from a slave (master-receiver), the slave transmits (slave-transmitter) a byte of data to the master, and the master then acknowledges the transaction with the ACK pulse. This transaction continues until the master terminates the transmission by not acknowledging (NACK) the transaction after the last byte is received, and then the master issues a STOP condition or addresses another slave after issuing a RESTART condition. This behavior is illustrated in [Figure 7](#).

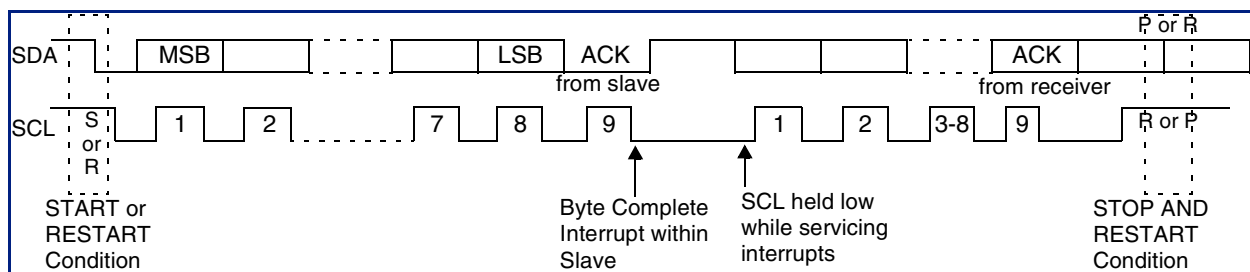


Figure 7: Data transfer on the I²C Bus

The DW_apb_i2c is a synchronous serial interface. The SDA line is a bidirectional signal and changes only while the SCL line is low, except for STOP, START, and RESTART conditions. The output drivers are open-drain or open-collector to perform wire-AND functions on the bus. The maximum number of devices on the bus is limited by only the maximum capacitance specification of 400 pF. Data is transmitted in byte packages.

The I²C protocols implemented in DW_apb_i2c are described in more details in the following section, “I²C Protocols” on page 50.

I²C Protocols

The DW_apb_i2c has the following protocols:

- “START and STOP Conditions”
- “Addressing Slave Protocol” on page 50
- “Transmitting and Receiving Protocol” on page 52
- “START BYTE Transfer Protocol” on page 53

START and STOP Conditions

When the bus is idle, both the SCL and SDA signals are pulled high through external pull-up resistors on the bus. When the master wants to start a transmission on the bus, the master issues a START condition. This is defined to be a high-to-low transition of the SDA signal while SCL is 1. When the master wants to terminate the transmission, the master issues a STOP condition. This is defined to be a low-to-high transition of the SDA line while SCL is 1. [Figure 8](#) shows the timing of the START and STOP conditions. When data is being transmitted on the bus, the SDA line must be stable when SCL is 1.

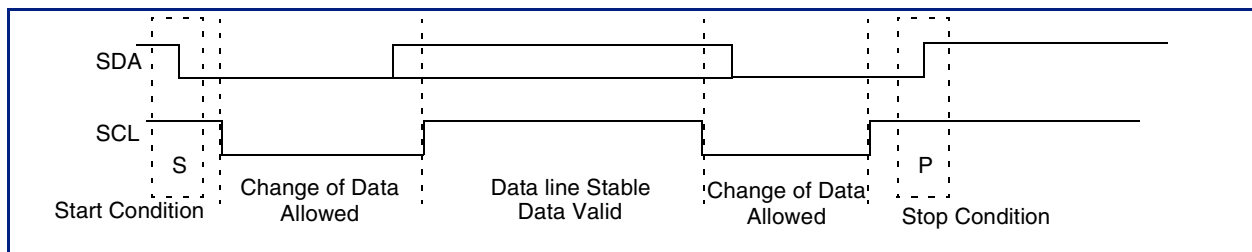


Figure 8: START and STOP Condition



Note

The signal transitions for the START/STOP conditions, as depicted in [Figure 8](#), reflect those observed at the output signals of the Master driving the I²C bus. Care should be taken when observing the SDA/SCL signals at the input signals of the Slave(s), because unequal line delays may result in an incorrect SDA/SCL timing relationship.

Addressing Slave Protocol

There are two address formats: the 7-bit address format and the 10-bit address format.

7-bit Address Format

During the 7-bit address format, the first seven bits (bits 7:1) of the first byte set the slave address and the LSB bit (bit 0) is the R/W bit as shown in [Figure 9](#). When bit 0 (R/W) is set to 0, the master writes to the slave. When bit 0 (R/W) is set to 1, the master reads from the slave.

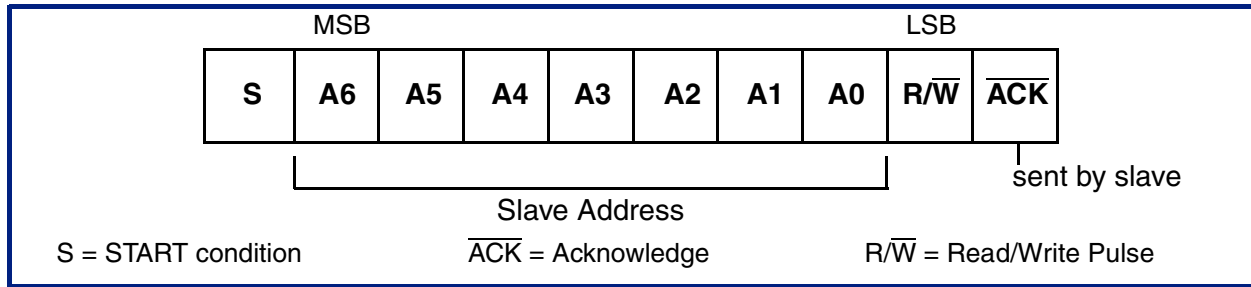


Figure 9: 7-bit Address Format

10-bit Address Format

During 10-bit addressing, two bytes are transferred to set the 10-bit address. The transfer of the first byte contains the following bit definition. The first five bits (bits 7:3) notify the slaves that this is a 10-bit transfer followed by the next two bits (bits 2:1), which set the slaves address bits 9:8, and the LSB bit (bit 0) is the R/W bit. The second byte transferred sets bits 7:0 of the slave address. [Figure 10](#) shows the 10-bit address format, and [Table 7 on page 51](#) defines the special purpose and reserved first byte addresses.

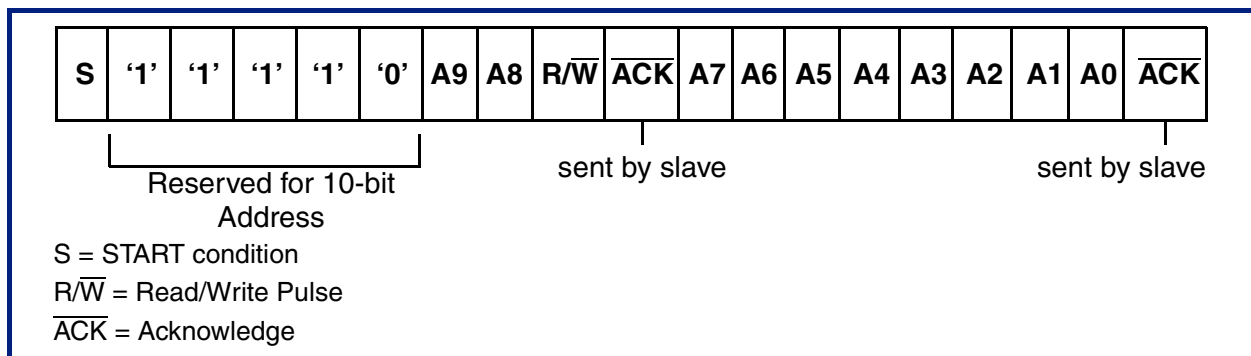


Figure 10: 10-bit Address Format

Table 7: I²C Definition of Bits in First Byte

Slave Address	R/W Bit	Description
0000 000	0	General Call Address. DW_apb_i2c places the data in the receive buffer and issues a General Call interrupt.
0000 000	1	START byte. For more details, refer to “START BYTE Transfer Protocol” on page 53 .
0000 001	X	CBUS address. DW_apb_i2c ignores these accesses.
0000 010	X	Reserved.
0000 011	X	Reserved.
0000 1XX	X	High-speed master code (for more information, refer to “Multiple Master Arbitration” on page 54).
1111 1XX	X	Reserved.
1111 0XX	X	10-bit slave addressing.



Attention

DW_apb_i2c does not restrict you from using these reserved addresses. However, if you use these reserved addresses, you may run into incompatibilities with other I²C components.

Transmitting and Receiving Protocol

The master can initiate data transmission and reception to/from the bus, acting as either a master-transmitter or master-receiver. A slave responds to requests from the master to either transmit data or receive data to/from the bus, acting as either a slave-transmitter or slave-receiver, respectively.

Master-Transmitter and Slave-Receiver

All data is transmitted in byte format, with no limit on the number of bytes transferred per data transfer. After the master sends the address and R/W bit or the master transmits a byte of data to the slave, the slave-receiver must respond with the acknowledge signal (ACK). When a slave-receiver does not respond with an ACK pulse, the master aborts the transfer by issuing a STOP condition. The slave must leave the SDA line high so that the master can abort the transfer.

If the master-transmitter is transmitting data as shown in Figure 11, then the slave-receiver responds to the master-transmitter with an acknowledge pulse after every byte of data is received.

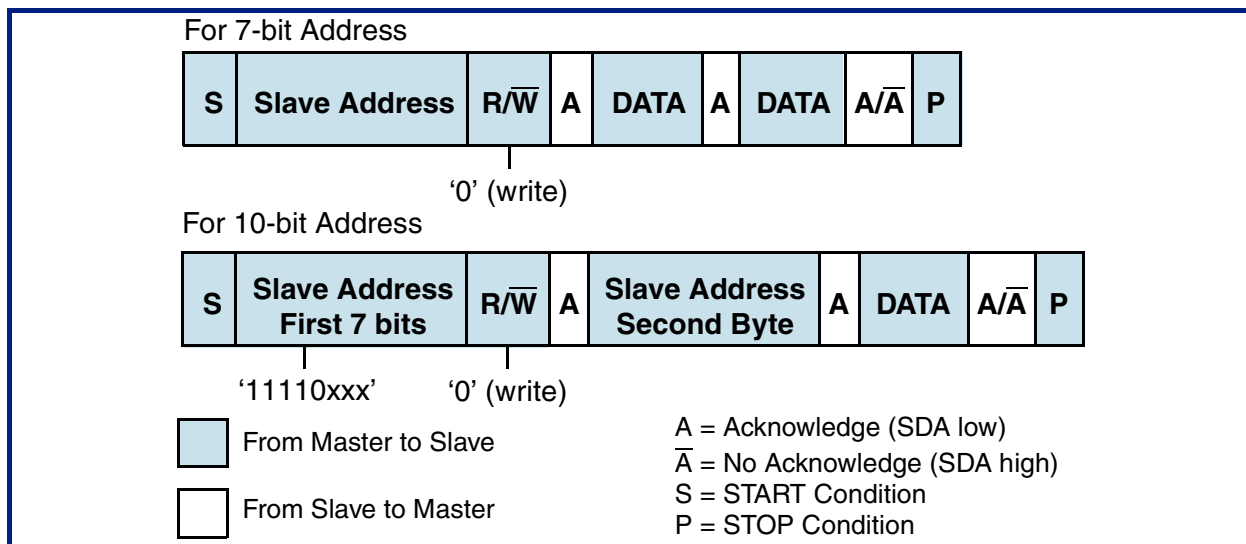


Figure 11: Master-Transmitter Protocol

Master-Receiver and Slave-Transmitter

If the master is receiving data as shown in Figure 12 on page 53, then the master responds to the slave-transmitter with an acknowledge pulse after a byte of data has been received, except for the last byte. This is the way the master-receiver notifies the slave-transmitter that this is the last byte. The slave-transmitter relinquishes the SDA line after detecting the No Acknowledge (NACK) so that the master can issue a STOP condition.

When a master does not want to relinquish the bus with a STOP condition, the master can issue a RESTART condition. This is identical to a START condition except it occurs after the ACK pulse. The master can then communicate with the same slave or a different slave.

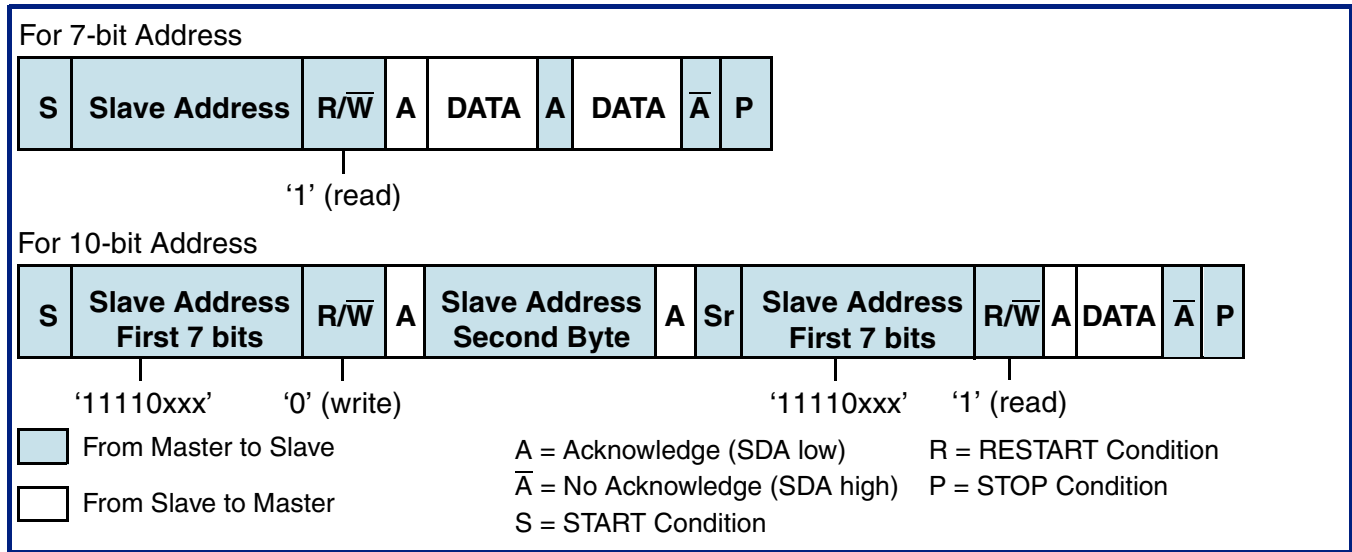


Figure 12: Master-Receiver Protocol

START BYTE Transfer Protocol

The START BYTE transfer protocol is set up for systems that do not have an on-board dedicated I²C hardware module. When the DW_apb_i2c is addressed as a slave, it always samples the I²C bus at the highest speed supported so that it never requires a START BYTE transfer. However, when DW_apb_i2c is a master, it supports the generation of START BYTE transfers at the beginning of every transfer in case a slave device requires it. This protocol consists of seven zeros being transmitted followed by a 1, as illustrated in Figure 13. This allows the processor that is polling the bus to under-sample the address phase until 0 is detected. Once the microcontroller detects a 0, it switches from the under sampling rate to the correct rate of the master.

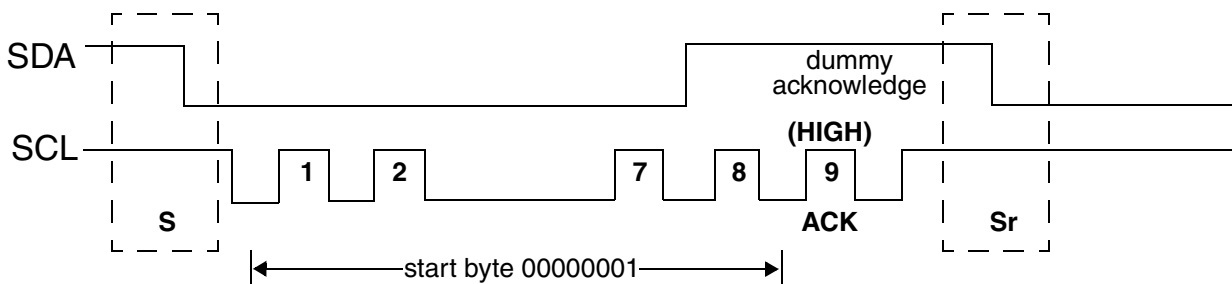


Figure 13: START BYTE Transfer

The START BYTE procedure is as follows:

1. Master generates a START condition.
2. Master transmits the START byte (0000 0001).
3. Master transmits the ACK clock pulse. (Present only to conform with the byte handling format used on the bus)
4. No slave sets the ACK signal to 0.
5. Master generates a RESTART (R) condition.

A hardware receiver does not respond to the START BYTE because it is a reserved address and resets after the RESTART condition is generated.

Multiple Master Arbitration



Note In a multiple master I²C bus system, the DW_apb_i2c should not be programmed as a master device. For multiple master systems, the DW_apb_i2c can only be operated as a slave (set `IC_CON.MASTER_MODE` to 0 (master disabled)).

The DW_apb_i2c bus protocol allows multiple masters to reside on the same bus. If there are two masters on the same I²C-bus, there is an arbitration procedure if both try to take control of the bus at the same time by generating a START condition at the same time. Once a master (for example, a microcontroller) has control of the bus, no other master can take control until the first master sends a STOP condition and places the bus in an idle state.

Arbitration takes place on the SDA line, while the SCL line is 1. The master, which transmits a 1 while the other master transmits 0, loses arbitration and turns off its data output stage. The master that lost arbitration can continue to generate clocks until the end of the byte transfer. If both masters are addressing the same slave device, the arbitration could go into the data phase. Figure 14 on page 54 illustrates the timing of when two masters are arbitrating on the bus.

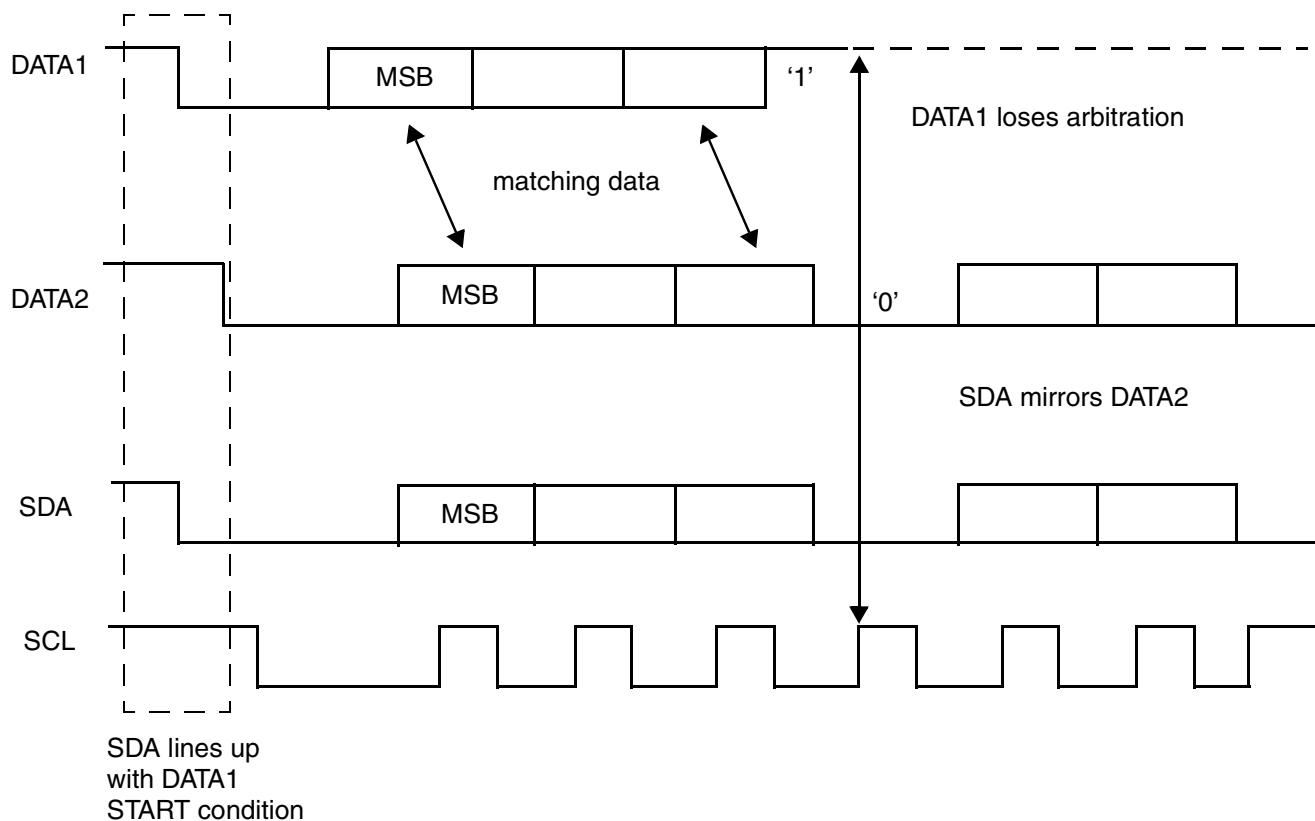


Figure 14: Multiple Master Arbitration

For high-speed mode, the arbitration cannot go into the data phase because each master is programmed with a unique high-speed master code. This 8-bit code is defined by the system designer and is set by writing to the High Speed Master Mode Code Address Register, `IC_HS_MADDR`. Because the codes are unique, only one master can win arbitration, which occurs by the end of the transmission of the high-speed master code.

Control of the bus is determined by address or master code and data sent by competing masters, so there is no central master nor any order of priority on the bus.

Arbitration is not allowed between the following conditions:

- A RESTART condition and a data bit
- A STOP condition and a data bit
- A RESTART condition and a STOP condition

Slaves are not involved in the arbitration process.

Clock Synchronization

When two or more masters try to transfer information on the bus at the same time, they must arbitrate and synchronize the SCL clock. All masters generate their own clock to transfer messages. Data is valid only during the high period of SCL clock. Clock synchronization is performed using the wired-AND connection to the SCL signal. When the master transitions the SCL clock to 0, the master starts counting the low time of the SCL clock and transitions the SCL clock signal to 1 at the beginning of the next clock period. However, if another master is holding the SCL line to 0, then the master goes into a HIGH wait state until the SCL clock line transitions to 1.

All masters then count off their high time, and the master with the shortest high time transitions the SCL line to 0. The masters then count out their low time and the one with the longest low time forces the other master into a HIGH wait state. Therefore, a synchronized SCL clock is generated, which is illustrated in [Figure 15](#). Optionally, slaves may hold the SCL line low to slow down the timing on the I²C bus.

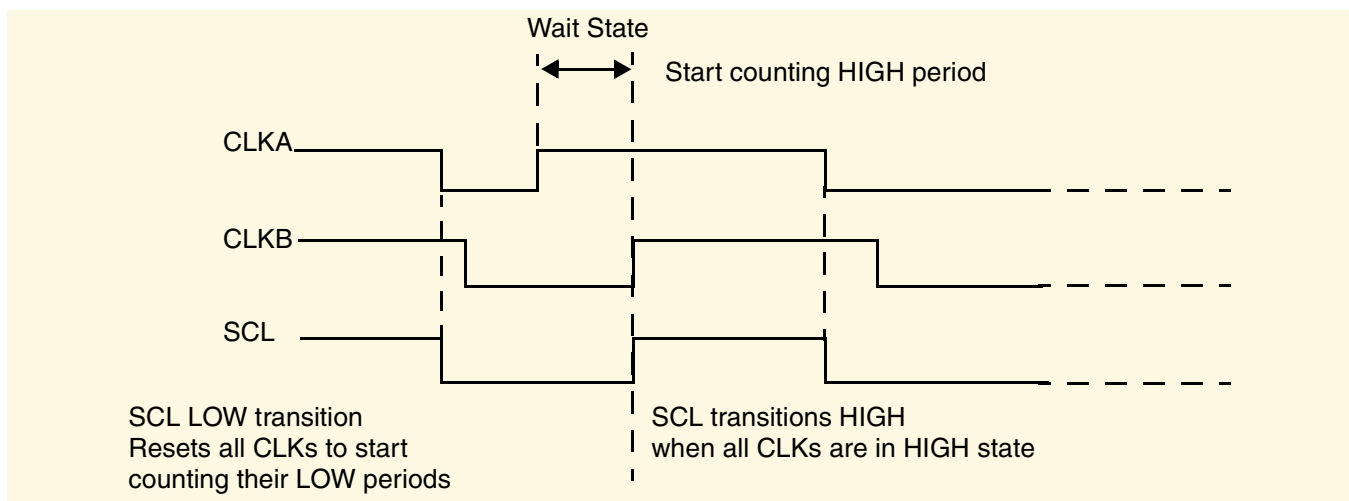


Figure 15: Multi-Master Clock Synchronization

Operation Modes

This section provides information on the following topics:

- “Slave Mode Operation”
- “Master Mode Operation” on page 60
- “Disabling DW_apb_i2c” on page 62

Note

It is important to note that the DW_apb_i2c should only be set to operate as an I²C Master, or I²C Slave, but not both simultaneously. This is achieved by ensuring that bit 6 (IC_SLAVE_DISABLE) and 0 (IC_MASTER_MODE) of the IC_CON register are never set to 0 and 1, respectively.

Slave Mode Operation

This section includes the following procedures:

- “Initial Configuration”
- “Slave-Transmitter Operation for a Single Byte” on page 57
- “Slave-Receiver Operation for a Single Byte” on page 58
- “Slave-Transfer Operation For Bulk Transfers” on page 58

Initial Configuration

To use the DW_apb_i2c as a slave, perform the following steps:

1. Disable the DW_apb_i2c by writing a ‘0’ to bit 0 of the IC_ENABLE register.
2. Write to the IC_SAR register (bits 9:0) to set the slave address. This is the address to which the DW_apb_i2c responds.
3. Write to the IC_CON register to specify which type of addressing is supported (7- or 10-bit by setting bit 3). Enable the DW_apb_i2c in slave-only mode by writing a ‘0’ into bit 6 (IC_SLAVE_DISABLE) and a ‘0’ to bit 0 (MASTER_MODE).

Note

Slaves and masters do not have to be programmed with the same type of addressing 7- or 10-bit address. For instance, a slave can be programmed with 7-bit addressing and a master with 10-bit addressing, and vice versa.

4. Enable the DW_apb_i2c by writing a ‘1’ in bit 0 of the IC_ENABLE register.

Note

Depending on the reset values chosen, steps 2 and 3 may not be necessary because the reset values can be configured. For instance, if the device is only going to be a master, there would be no need to set the slave address because you can configure DW_apb_i2c to have the slave disabled after reset and to enable the master after reset. The values stored are static and do not need to be reprogrammed if the DW_apb_i2c is disabled.

Slave-Transmitter Operation for a Single Byte

When another I²C master device on the bus addresses the DW_apb_i2c and requests data, the DW_apb_i2c acts as a slave-transmitter and the following steps occur:

1. The other I²C master device initiates an I²C transfer with an address that matches the slave address in the [IC_SAR](#) register of the DW_apb_i2c.
2. The DW_apb_i2c acknowledges the sent address and recognizes the direction of the transfer to indicate that it is acting as a slave-transmitter.
3. The DW_apb_i2c asserts the RD_REQ interrupt (bit 5 of the [IC_RAW_INTR_STAT](#) register) and holds the SCL line low. It is in a wait state until software responds.

If the RD_REQ interrupt has been masked, due to [IC_INTR_MASK](#)[5] register (M_RD_REQ bit field) being set to 0, then it is recommended that a hardware and/or software timing routine be used to instruct the CPU to perform periodic reads of the [IC_RAW_INTR_STAT](#) register.

- a. Reads that indicate [IC_RAW_INTR_STAT](#)[5] (R_RD_REQ bit field) being set to 1 must be treated as the equivalent of the RD_REQ interrupt being asserted.
- b. Software must then act to satisfy the I²C transfer.
- c. The timing interval used should be in the order of 10 times the fastest SCL clock period the DW_apb_i2c can handle. For example, for 400 kb/s, the timing interval is 25us.



Note

The value of 10 is recommended here because this is approximately the amount of time required for a single byte of data transferred on the I²C bus.

4. If there is any data remaining in the TX FIFO before receiving the read request, then the DW_apb_i2c asserts a TX_ABRT interrupt (bit 6 of the [IC_RAW_INTR_STAT](#) register) to flush the old data from the TX FIFO.



Note

Because the DW_apb_i2c's TX FIFO is forced into a flushed/reset state whenever a TX_ABRT event occurs, it is necessary for software to release the DW_apb_i2c from this state by reading the [IC_CLR_TX_ABRT](#) register before attempting to write into the TX FIFO. See register [IC_RAW_INTR_STAT](#) for more details.

If the TX_ABRT interrupt has been masked, due to of [IC_INTR_MASK](#)[6] register (M_TX_ABRT bit field) being set to 0, then it is recommended that re-using the timing routine (described in the previous step), or a similar one, be used to read the [IC_RAW_INTR_STAT](#) register.

- a. Reads that indicate bit 6 (R_TX_ABRT) being set to 1 must be treated as the equivalent of the TX_ABRT interrupt being asserted.
 - b. There is no further action required from software.
 - c. The timing interval used should be similar to that described in the previous step for the [IC_RAW_INTR_STAT](#)[5] register.
5. Software writes to the [IC_DATA_CMD](#) register with the data to be written (by writing a '0' in bit 8).

6. Software must clear the RD_REQ and TX_ABRT interrupts (bits 5 and 6, respectively) of the [IC_RAW_INTR_STAT](#) register before proceeding.
If the RD_REQ and/or TX_ABRT interrupts have been masked, then clearing of the [IC_RAW_INTR_STAT](#) register will have already been performed when either the R_RD_REQ or R_TX_ABRT bit has been read as 1.
7. The DW_apb_i2c releases the SCL and transmits the byte.
8. The master may hold the I²C bus by issuing a RESTART condition or release the bus by issuing a STOP condition.

Slave-Receiver Operation for a Single Byte

When another I²C master device on the bus addresses the DW_apb_i2c and is sending data, the DW_apb_i2c acts as a slave-receiver and the following steps occur:

1. The other I²C master device initiates an I²C transfer with an address that matches the DW_apb_i2c's slave address in the [IC_SAR](#) register.
2. The DW_apb_i2c acknowledges the sent address and recognizes the direction of the transfer to indicate that the DW_apb_i2c is acting as a slave-receiver.
3. DW_apb_i2c receives the transmitted byte and places it in the receive buffer.



Note

If the RX FIFO is completely filled with data when a byte is pushed, then an overflow occurs and the DW_apb_i2c continues with subsequent I²C transfers. Because a NACK is not generated, software must recognize the overflow when indicated by the DW_apb_i2c (by the R_RX_OVER bit in the [IC_INTR_STAT](#) register) and take appropriate actions to recover from lost data. Hence, there is a real time constraint on software to service the RX FIFO before the latter overflow as there is no way to re-apply pressure to the remote transmitting master. You must select a deep enough RX FIFO depth to satisfy the interrupt service interval of their system.

4. DW_apb_i2c asserts the RX_FULL interrupt ([IC_RAW_INTR_STAT\[2\]](#) register).
If the RX_FULL interrupt has been masked, due to setting [IC_INTR_MASK\[2\]](#) register to 0 or setting [IC_TX_TL](#) to a value larger than 0, then it is recommended that a timing routine (described in “[Slave-Transmitter Operation for a Single Byte](#)” on page 57) be implemented for periodic reads of the “[IC_STATUS](#)” on page 137 register. Reads of the IC_STATUS register, with bit 3 (RFNE) set at 1, must then be treated by software as the equivalent of the RX_FULL interrupt being asserted.
5. Software may read the byte from the [IC_DATA_CMD](#) register (bits 7:0).
6. The other master device may hold the I²C bus by issuing a RESTART condition or release the bus by issuing a STOP condition.

Slave-Transfer Operation For Bulk Transfers

In the standard I²C protocol, all transactions are single byte transactions and the programmer responds to a remote master read request by writing one byte into the slave's TX FIFO. When a slave (slave-transmitter) is issued with a read request (RD_REQ) from the remote master (master-receiver), at a minimum there should be at least one entry placed into the slave-transmitter's TX FIFO.

DW_apb_i2c is designed to handle more data in the TX FIFO so that subsequent read requests can take that data without raising an interrupt to get more data. Ultimately, this eliminates the possibility of significant latencies being incurred between raising the interrupt for data each time had there been a restriction of having only one entry placed in the TX FIFO.

This mode only occurs when DW_apb_i2c is acting as a slave-transmitter. If the remote master acknowledges the data sent by the slave-transmitter and there is no data in the slave's TX FIFO, the DW_apb_i2c holds the I²C SCL line low while it raises the read request interrupt (RD_REQ) and waits for data to be written into the TX FIFO before it can be sent to the remote master.

If the RD_REQ interrupt is masked, due to bit 5 (M_RD_REQ) of the [IC_INTR_STAT](#) register being set to 0, then it is recommended that a timing routine be used to activate periodic reads of the [IC_RAW_INTR_STAT](#) register. Reads of IC_RAW_INTR_STAT that return bit 5 (R_RD_REQ) set to 1 must be treated as the equivalent of the RD_REQ interrupt referred to in this section. This timing routine is similar to that described in “[Slave-Transmitter Operation for a Single Byte](#)” on page 57.

The RD_REQ interrupt is raised upon a read request, and like interrupts, must be cleared when exiting the interrupt service handling routine (ISR). The ISR allows you to either write 1 byte or more than 1 byte into the TX FIFO. During the transmission of these bytes to the master, if the master acknowledges the last byte, then the slave must raise the RD_REQ again because the master is requesting for more data.

If the programmer knows in advance that the remote master is requesting a packet of n bytes, then when another master addresses DW_apb_i2c and requests data, the TX FIFO could be written with n number bytes and the remote master receives it as a continuous stream of data. For example, the DW_apb_i2c slave continues to send data to the remote master as long as the remote master is acknowledging the data sent and there is data available in the TX FIFO. There is no need to hold the SCL line low or to issue RD_REQ again.

If the remote master is to receive n bytes from the DW_apb_i2c but the programmer wrote a number of bytes larger than n to the TX FIFO, then when the slave finishes sending the requested n bytes, it clears the TX FIFO and ignores any excess bytes.

The DW_apb_i2c generates a transmit abort (TX_ABRT) event to indicate the clearing of the TX FIFO in this example. At the time an ACK/NACK is expected, if a NACK is received, then the remote master has all the data it wants. At this time, a flag is raised within the slave's state machine to clear the leftover data in the TX FIFO. This flag is transferred to the processor bus clock domain where the FIFO exists and the contents of the TX FIFO is cleared at that time.

Master Mode Operation

This section includes the following topics:

- “Initial Configuration”
- “Dynamic IC_TAR or IC_10BITADDR_MASTER Update” on page 61
- “Master Transmit and Master Receive” on page 62

Initial Configuration

The initial configuration procedure for Master Mode Operation depends on the configuration parameter I2C_DYNAMIC_TAR_UPDATE. When set to “Yes” (1), the target address and address format can be changed dynamically without having to disable DW_apb_i2c. This parameter only applies to when DW_apb_i2c is acting as a master because the slave requires the component to be disabled before any changes can be made to the address. For more information about this parameter, see [page 83](#). For more information about how this parameter affects the IC_TAR register, see [page 100](#).

The procedures are very similar and are only different with regard to where the IC_10BITADDR_MASTER bit is set (either bit 4 of IC_CON register or bit 12 of IC_TAR register).

I2C_DYNAMIC_TAR_UPDATE = 0

To use the DW_apb_i2c as a master when the I2C_DYNAMIC_TAR_UPDATE configuration parameter is set to “No” (0), perform the following steps:

1. Disable the DW_apb_i2c by writing 0 to the [IC_ENABLE](#) register.
2. Write to the [IC_CON](#) register to set the maximum speed mode supported (bits 2:1) and the desired speed of the DW_apb_i2c master-initiated transfers, either 7-bit or 10-bit addressing (bit 4). Ensure that bit 6 (IC_SLAVE_DISABLE) is written with a ‘1’ and bit 0 (MASTER_MODE) is written with a ‘1’.



Note

Slaves and masters do not have to be programmed with the same type of addressing 7- or 10-bit address. For instance, a slave can be programmed with 7-bit addressing and a master with 10-bit addressing, and vice versa.

3. Write to the [IC_TAR](#) register the address of the I²C device to be addressed (bits 9:0). This register also indicates whether a General Call or a START BYTE command is going to be performed by I²C.
4. *Only applicable for high-speed mode transfers.* Write to the [IC_HS_MADDR](#) register the desired master code for the DW_apb_i2c. The master code is programmer-defined.
5. Enable the DW_apb_i2c by writing a ‘1’ in bit 0 of the [IC_ENABLE](#) register.
6. Now write transfer direction and data to be sent to the [IC_DATA_CMD](#) register. If the [IC_DATA_CMD](#) register is written before the DW_apb_i2c is enabled, the data and commands are lost as the buffers are kept cleared when DW_apb_i2c is disabled.

This step generates the START condition and the address byte on the DW_apb_i2c. Once DW_apb_i2c is enabled and there is data in the TX FIFO, DW_apb_i2c starts reading the data.

**Note**

Depending on the reset values chosen, steps 2, 3, 4, and 5 may not be necessary because the reset values can be configured. The values stored are static and do not need to be reprogrammed if the DW_apb_i2c is disabled, with the exception of the transfer direction and data.

I2C_DYNAMIC_TAR_UPDATE = 1

To use the DW_apb_i2c as a master when the I2C_DYNAMIC_TAR_UPDATE configuration parameter is set to “Yes” (1), perform the following steps:

1. Disable the DW_apb_i2c by writing 0 to the **IC_ENABLE** register.
2. Write to the **IC_CON** register to set the maximum speed mode supported for slave operation (bits 2:1) and to specify whether the DW_apb_i2c starts its transfers in 7/10 bit addressing mode when the device is a slave (bit 3).
3. Write to the **IC_TAR** register the address of the I²C device to be addressed. It also indicates whether a General Call or a START BYTE command is going to be performed by I²C. The desired speed of the DW_apb_i2c master-initiated transfers, either 7-bit or 10-bit addressing, is controlled by the IC_10BITADDR_MASTER bit field (bit 12).
4. *Only applicable for high-speed mode transfers.* Write to the **IC_HS_MADDR** register the desired master code for the DW_apb_i2c. The master code is programmer-defined.
5. Enable the DW_apb_i2c by writing a 1 in the **IC_ENABLE** register.
6. Now write the transfer direction and data to be sent to the **IC_DATA_CMD** register. If the **IC_DATA_CMD** register is written before the DW_apb_i2c is enabled, the data and commands are lost as the buffers are kept cleared when DW_apb_i2c is not enabled.

**Note**

For multiple I²C transfers, perform additional writes to the TX FIFO such that the TX FIFO does not become empty during the I²C transaction. If the TX FIFO is completely emptied at any stage, then further writes to the TX FIFO results in an independent I²C transaction.

Dynamic IC_TAR or IC_10BITADDR_MASTER Update

The DW_apb_i2c supports dynamic updating of the IC_TAR (bits 9:0) and IC_10BITADDR_MASTER (bit 12) bit fields of the IC_TAR register. In order to perform a dynamic update of the IC_TAR register, the I2C_DYNAMIC_TAR_UPDATE configuration parameter must be set to “Yes” (1). You can dynamically write to the IC_TAR register provided the following conditions are met:

1. DW_apb_i2c is not enabled (IC_ENABLE=0);
OR
2. DW_apb_i2c is enabled (IC_ENABLE=1); AND
DW_apb_i2c is NOT engaged in any Master (tx, rx) operation (IC_STATUS[5]=0); AND
DW_apb_i2c is enabled to operate in Master mode (IC_CON[0]=1); AND
there are NO entries in the TX FIFO (IC_STATUS[2]=1)

Master Transmit and Master Receive

The DW_apb_i2c supports switching back and forth between reading and writing dynamically. To transmit data, write the data to be written to the lower byte of the I²C Rx/Tx Data Buffer and Command Register (**IC_DATA_CMD**). The *CMD* bit [8] should be written to 0 for I²C write operations. Subsequently, a read command may be issued by writing “don’t cares” to the lower byte of the **IC_DATA_CMD** register, and a 1 should be written to the *CMD* bit.

Disabling DW_apb_i2c

The register **IC_ENABLE_STATUS** is added to allow software to unambiguously determine when the hardware has completely shutdown in response to the **IC_ENABLE** register being set from 1 to 0. Only one register is required to be monitored, as opposed to monitoring two registers (**IC_STATUS** and **IC_RAW_INTR_STAT**) which is a requirement for DW_apb_i2c versions 1.05a or earlier.

Procedure

1. Define a timer interval (t_{i2c_poll}) equal to the 10 times the signaling period for the highest I²C transfer speed used in the system and supported by DW_apb_i2c. For example, if the highest I²C transfer mode is 400 kb/s, then this t_{i2c_poll} is 25 μ s.
2. Define a maximum time-out parameter, **MAX_T_POLL_COUNT**, such that if any repeated polling operation exceeds this maximum value, an error is reported.
3. Execute a blocking thread/process/function that prevents any further I²C master transactions to be started by software, but allows any pending transfers to be completed.



Note

This step can be ignored if DW_apb_i2c is programmed to operate as an I²C slave only.

4. The variable **POLL_COUNT** is initialized to zero.
5. Set **IC_ENABLE** to 0.
6. Read the **IC_ENABLE_STATUS** register and test the **IC_EN** bit (bit 0). Increment **POLL_COUNT** by one. If **POLL_COUNT** \geq **MAX_T_POLL_COUNT**, exit with the relevant error code.
7. If **IC_ENABLE_STATUS**[0] is 1, then sleep for t_{i2c_poll} and proceed to the previous step. Otherwise, exit with a relevant success code.

IC_CLK Frequency Configuration

When the DW_apb_i2c is configured as a master, the *CNT registers must be set before any I²C bus transaction can take place to ensure proper I/O timing. The *CNT registers are:

- IC_SS_SCL_HCNT
- IC_SS_SCL_LCNT
- IC_FS_SCL_HCNT
- IC_FS_SCL_LCNT
- IC_HS_SCL_HCNT
- IC_HS_SCL_LCNT



Note

It is not necessary to program any of the *CNT registers if DW_apb_i2c is enabled to operate only as an I²C slave, because these registers are only used to determine the SCL timing requirements for operation as an I²C master.

Setting the *_LCNT registers configures the number of ic_clk signals that are required for setting the low time of the SCL clock in each speed mode. Setting the *_HCNT* registers configures the number of ic_clk signals that are required for setting the high time of the SCL clock in each speed mode. Setting the registers to the correct value is described as follows.

The equation to calculate the proper number of ic_clk signals required for setting the proper SCL clocks high and low times is as follows:

$$IC_xCNT = (\text{ROUNDUP}(\text{MIN_SCL_}xxx\text{time} * \text{OSCFREQ}, 0))$$

ROUNDUP is an explicit Excel function call that is used to convert a real number to its equivalent integer number.

MIN_SCL_HIGHTime = Minimum High Period

MIN_SCL_HIGHTime = 4000 ns for 100 kbps
 600 ns for 400 kbps
 60 ns for 3.4 Mbs, bus loading = 100pF
 160 ns for 3.4 Mbs, bus loading = 400pF

MIN_SCL_LOWtime = Minimum Low Period

MIN_SCL_LOWtime = 4700 ns for 100 kbps
 1300 ns for 400 kbps
 120 ns for 3.4Mbs, bus loading = 100pF
 320 ns for 3.4Mbs, bus loading = 400pF

OSCFREQ = ic_clk Clock Frequency (Hz).

For example:

OSCFREQ = 100 MHz
 I2Cmode = fast, 400 kbit/s
 MIN_SCL_HIGHTime = 600 ns.
 MIN_SCL_LOWtime = 1300 ns.

$$IC_xCNT = (\text{ROUNDUP}(\text{MIN_SCL_HIGH_LOWtime} * \text{OSCFREQ}, 0))$$

$$IC_HCNT = (\text{ROUNDUP}(600 \text{ ns} * 100 \text{ MHz}, 0))$$

$$IC_HCNTSCL_PERIOD = 60$$

$$IC_LCNT = (\text{ROUNDUP}(1300 \text{ ns} * 100 \text{ MHz}, 0))$$

$$IC_LCNTSCL_PERIOD = 130$$

Actual MIN_SCL_HIGHTime = 60*(1/100 MHz) = 600 ns

Actual MIN_SCL_LOWtime = 130*(1/100 MHz) = 1300 ns

When the DW_apb_i2c operates as an I²C Master, in both transmit and receive transfers, the minimum value that can be programmed in the *_LCNT registers is 8. Likewise, the minimum value allowed for the *_HCNT registers is 6.

Note

The actual SCL low and high times are larger than the values written into the *_LCNT and *_HCNT registers, respectively. One additional ic_clk period for the SCL low time is added by the DW_apb_i2c, while eight additional ic_clk periods for the SCL high time is added. Alternatively, you can subtract 1 from the calculated low count and 8 from the calculated high count and use the resulting values for programming into the *_LCNT and *_HCNT registers in order to account for this behavior.

The following points explain why this behavior occurs:

- The counting logic for the SCL low and high times actually uses (*_LCNT+1) and (*_HCNT+1) values.
- Both the SDA and SCL signals are monitored for contentions, which may result in loss of arbitration during Master transfer operations, as well as ensuring that the SCL high count is started correctly. Since these signals can be asynchronous to ic_clk, digital filtering is applied to both SDA and SCL lines.
- The digital filtering applied to the SCL line incurs a delay of four ic_clk cycles. This filtering includes metastability removal and a 2-out-of-3 majority vote processing on SDA and SCL transitions (edges).
- Whenever SCL is driven “1” to “0” by the DW_apb_i2c—that is, completing the SCL high time—an internal logic latency of three ic_clk cycles is incurred.

Consequently, the minimum SCL low time, which the DW_apb_i2c is capable of, is 9 ic_clk periods—(8+1) ic_clk periods—while the minimum SCL high time is 14 ic_clk periods—(6+1+4+3) ic_clk periods.

At standard mode (100 kb/s), the required SCL frequency is 100kHz. The corresponding period is 10 μ s, to be counted with 23 ic_clks—(9+14) ic_clks. This gives an initial estimate for the ic_clk frequency of 2.3 MHz [$1.0/(10e^{-6}/23)$].

This selection results in SCL low and high times of 3.91 μ s—(9/2.3e⁶) μ s—and 6.09 ns—(14/23e⁶) ns. It should be noted that the SCL low time does not meet I²C specification requirements. The following table shows various frequency selections for the ic_clk and the corresponding SCL low/high settings and times.

ic_clk _{freq} (MHz)	SCL Low Count	SCL Low Time (μ s)	SCL High Count	SCL High Time (μ s)	Remarks
2.3	9	3.91	14	6.1	SCL low does not meet
2.4	11	4.6	13	—	SCL high count invalid
2.4	12	5.0	12	—	SCL high count invalid
2.5	11	4.4	14	5.6	SCL low time does not meet
2.6	11	4.2	15	5.7	SCL low time does not meet
2.6	12	4.6	14	5.8	SCL low time <i>almost</i> meets

ic_clk _{freq} (MHz)	SCL Low Count	SCL Low Time (μs)	SCL High Count	SCL High Time (μs)	Remarks
2.7	12	4.4	15	5.6	SCL low time does not meet
2.7	13	4.8	14	5.2	OK

As you can see in the table, the DW_apb_i2c can meet I²C standard mode transfers using an ic_clk frequency of 2.7 MHz. All the above calculations can be automated by using a spreadsheet or a C program.



Note

Using the above selected SCL low and high counts, the registers IC_SS_SCL_LCNT and IC_SS_SCL_HCNT, should be programmed with the values 12 and 6—that is, (13-1) and (14-8)—respectively.

At fast mode (400 kb/s), the lowest ic_clk frequency is 12 MHz, with SCL low and high counts of 16 and 14 respectively.

For high speed modes, transfer rates of 3.4 MB/s and 1.7 MB/s require the lowest ic_clk frequencies of 105.4 MHz and 56 MHz, respectively. The required corresponding SCL low/high counts are 15/14 and 19/14.

DMA Controller Interface

The DW_apb_i2c has an optional built-in DMA capability that can be selected at configuration time; it has a handshaking interface to a DMA Controller to request and control transfers. The APB bus is used to perform the data transfer to or from the DMA. While the DW_apb_i2c DMA operation is designed in a generic way to fit any DMA controller as easily as possible, it is designed to work seamlessly, and best used, with the DesignWare DMA Controller, the DW_ahb_dmac. The settings of the DW_ahb_dmac that are relevant to the operation of the DW_apb_i2c are discussed here, mainly bit fields in the DW_ahb_dmac channel control register, CTL_x, where *x* is the channel number.



Note

When the DW_apb_i2c interfaces to the DW_ahb_dmac, the DW_ahb_dmac is always a flow controller; that is, it controls the block size. This must be programmed by software in the DW_ahb_dmac. The DW_ahb_dmac always transfers data using DMA burst transactions if possible, for efficiency. For more information, refer to the [DesignWare DW_ahb_dmac Databook](#). Other DMA controllers act in a similar manner.

The relevant DMA settings are discussed in the following sections:

- [“Enabling the DMA Controller Interface”](#)
- [“Overview of Operation”](#) on page 66
- [“Transmit Watermark Level and Transmit FIFO Underflow”](#) on page 68
- [“Choosing the Transmit Watermark Level”](#) on page 68
- [“Selecting DEST_MSIZ and Transmit FIFO Overflow”](#) on page 69
- [“Receive Watermark Level and Receive FIFO Overflow”](#) on page 70
- [“Choosing the Receive Watermark level”](#) on page 70

- [“Selecting SRC_MSIZE and Receive FIFO Underflow” on page 70](#)
- [“Handshaking Interface Operation” on page 71](#)

**Note**

The DMA output `dma_finish` is a status signal to indicate that the DMA block transfer is complete. `DW_apb_i2c` does not use this status signal, and therefore does not appear in the I/O port list.

Enabling the DMA Controller Interface

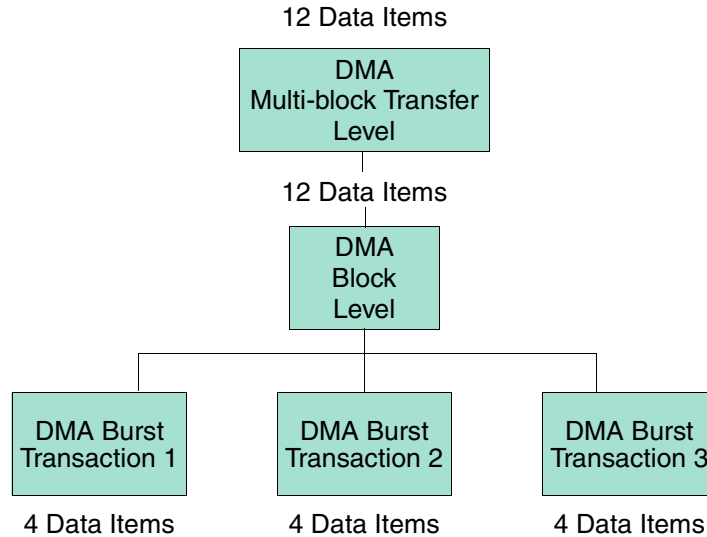
To enable the DMA Controller interface on the `DW_apb_i2c`, you must write the DMA Control Register (`IC_DMA_CR`). Writing a 1 into the `TDMAE` bit field of `IC_DMA_CR` register enables the `DW_apb_i2c` transmit handshaking interface. Writing a 1 into the `RDMAE` bit field of the `IC_DMA_CR` register enables the `DW_apb_i2c` receive handshaking interface.

Overview of Operation

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 `DW_apb_i2c`; this is programmed into the `BLOCK_TS` field of the `DW_ahb_dmac CTLx` register.

The block is broken into a number of transactions, each initiated by a request from the `DW_apb_i2c`. The DMA Controller must also be programmed with the number of data items (in this case, `DW_apb_i2c` 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 CTLx` register for source and destination, respectively.

[Figure 16 on page 67](#) 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_i2c` makes a transmit request to this channel, four data items are written to the `DW_apb_i2c` TX FIFO. Similarly, if the `DW_apb_i2c` makes a receive request to this channel, four data items are read from the `DW_apb_i2c` RX FIFO. Three separate requests must be made to this DMA channel before all 12 data items are written or read.



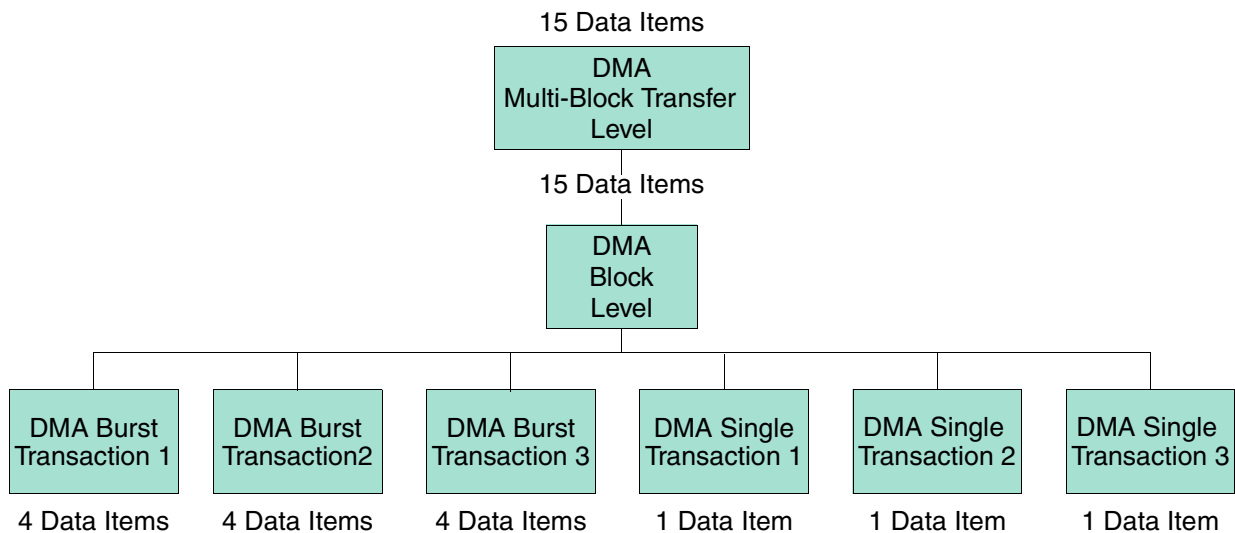
Block Size : DMA.CTLx.BLOCK_TS=12

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

I²C receive FIFO watermark level: I2C.DMARDLR + 1 = DMA.CTLx.SRC_MSIZ = 4
 (for more information, refer to discussion on [page 70](#))

Figure 16: 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 17](#), 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_MSIZ = 4

I²C transmit FIFO watermark level: I2C.IC_DMA_TDLR = DMA.CTLx.DEST_MSIZ = 4
 (for more information, refer to discussion on [page 69](#))

Figure 17: Breakdown of DMA Transfer into Single and Burst Transactions

Transmit Watermark Level and Transmit FIFO Underflow

During DW_apb_i2c 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 DMA Transmit Data Level Register (IC_DMA_TDLR) value; 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 CTLx.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 will run out of data causing a STOP to be inserted on the I²C bus. To prevent this condition, the user must set the watermark level correctly.

Choosing the Transmit Watermark Level

Consider the example where the assumption is made:

$$\text{DMA.CTLx.DEST_MSIZE} = \text{FIFO_DEPTH} - \text{I2C.IC_DMA_TDLR}$$

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: IC_DMA_TDLR = 2

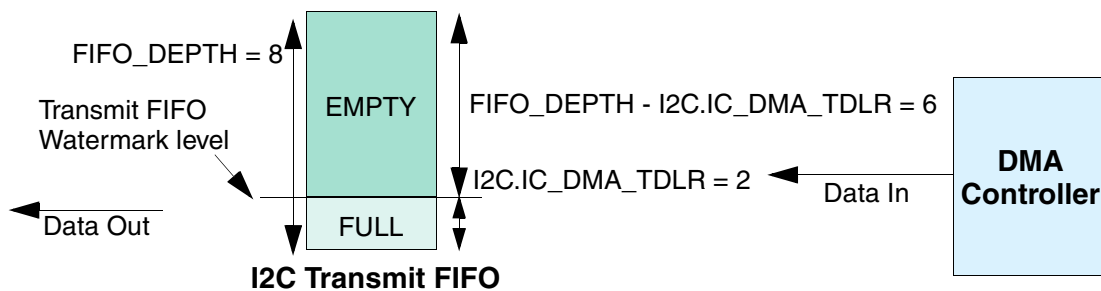


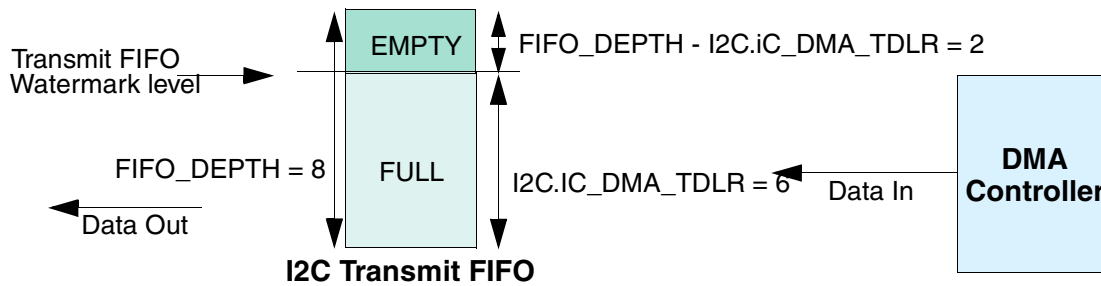
Figure 18: Case 1 Watermark Levels

$$\begin{aligned} \text{Transmit FIFO watermark level} &= \text{I2C.IC_DMA_TDLR} = 2 \\ \text{DMA.CTLx.DEST_MSIZE} &= \text{FIFO_DEPTH} - \text{I2C.IC_DMA_TDLR} = 6 \\ \text{I2C transmit FIFO_DEPTH} &= 8 \\ \text{DMA.CTLx.BLOCK_TS} &= 30 \end{aligned}$$

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

$$\text{DMA.CTLx.BLOCK_TS}/\text{DMA.CTLx.DEST_MSIZE} = 30/6 = 5$$

The number of burst transactions in the DMA block transfer is 5. But the watermark level, I2C.IC_DMA_TDLR, is quite low. Therefore, the probability of an I²C underflow is high where the I²C 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: IC_DMA_TDLR = 6**Figure 19: Case 2 Watermark Levels**

Transmit FIFO watermark level = $I2C.IC_DMA_TDLR = 6$
 $DMA.CTLx.DEST_MSIZE = FIFO_DEPTH - I2C.IC_DMA_TDLR = 2$
 $I2C\ transmit\ FIFO_DEPTH = 8$
 $DMA.CTLx.BLOCK_TS = 30$

Number of burst transactions in Block:

$$DMA.CTLx.BLOCK_TS/DMA.CTLx.DEST_MSIZE = 30/2 = 15$$

In this block transfer, there are 15 destination burst transactions in a DMA block transfer. But the watermark level, $I2C.IC_DMA_TDLR$, is high. Therefore, the probability of an I²C underflow is low because the DMA controller has plenty of time to service the destination burst transaction request before the I²C 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 I²C 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_MSIZ and Transmit FIFO Overflow

As can be seen from [Figure 19 on page 69](#), programming $DMA.CTLx.DEST_MSIZE$ to a value greater than the watermark level that triggers the DMA request may cause overflow when there is not enough space in the I²C transmit FIFO to service the destination burst request. Therefore, the following equation must be adhered to in order to avoid overflow:

$$DMA.CTLx.DEST_MSIZE \leq I2C.FIFO_DEPTH - I2C.IC_DMA_TDLR \quad (1)$$

In “[Case 2: IC_DMA_TDLR = 6](#)”, the amount of space in the transmit FIFO at the time the burst request is made is equal to the destination burst length, $DMA.CTLx.DEST_MSIZE$. Thus, the transmit FIFO may be full, but not overflowed, at the completion of the burst transaction.

Therefore, for optimal operation, DMA.CTLx.DEST_MSIZEx should be set at the FIFO level that triggers a transmit DMA request; that is:

$$\text{DMA.CTLx.DEST_MSIZE} = \text{I2C.FIFO_DEPTH} - \text{I2C.IC_DMA_TDLR} \quad (2)$$

This is the setting used in [Figure 17 on page 67](#).

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 will not be full at the end of a DMA burst transfer if the I²C has successfully transmitted one data item or more on the I²C serial transmit line during the transfer.

Receive Watermark Level and Receive FIFO Overflow

During DW_apb_i2c 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 DMA Receive Data Level Register; that is, IC_DMA_RDLR+1. 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 CTLx.SRC_MSIZEx.

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 will fill with data (overflow). To prevent this condition, the user must correctly set the watermark level.

Choosing the Receive Watermark level

Similar to choosing the transmit watermark level described earlier, the receive watermark level, IC_DMA_RDLR+1, should be set to minimize the probability of overflow, as shown in [Figure 20 on page 71](#). It is a trade-off between the number of DMA burst transactions required per block versus the probability of an overflow occurring.

Selecting SRC_MSIZEx and Receive FIFO Underflow

As can be seen in [Figure 20 on page 71](#), 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:

$$\text{DMA.CTLx.SRC_MSIZE} \leq \text{I2C.IC_DMA_RDLR} + 1 \quad (4)$$

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.CTLx.SRC_MSIZEx – the receive FIFO may be emptied, but not underflowed, at the completion of the burst transaction. For optimal operation, DMA.CTLx.SRC_MSIZEx should be set at the watermark level; that is:

$$\text{DMA.CTLx.SRC_MSIZE} = \text{I2C.IC_DMA_RDLR} + 1 \quad (5)$$

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 will not be empty at the end of the source burst transaction if the I²C has successfully received one data item or more on the I²C serial receive line during the burst.

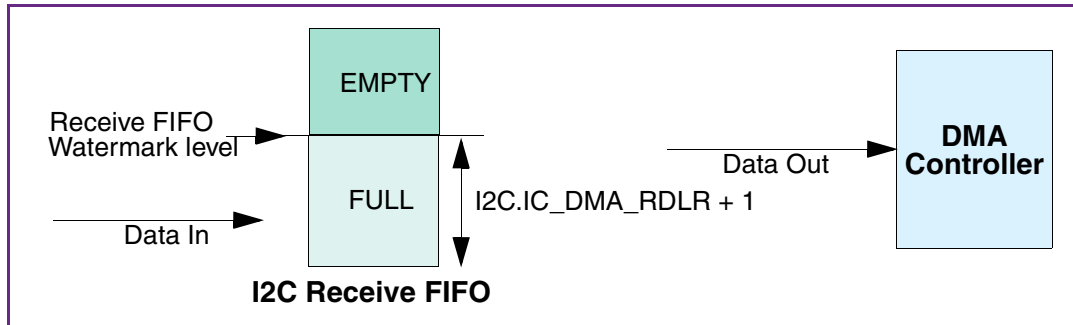


Figure 20: I²C Receive FIFO

Handshaking Interface Operation

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

The DW_ahb_dmac uses rising-edge detection of the `dma_tx_req` signal/`dma_rx_req` to identify a request on the channel. Upon reception of the `dma_tx_ack`/`dma_rx_ack` signal from the DW_ahb_dmac to indicate the burst transaction is complete, the DW_apb_i2c de-asserts the burst request signals, `dma_tx_req`/`dma_rx_req`, until `dma_tx_ack`/`dma_rx_ack` is de-asserted by the DW_ahb_dmac.

When the DW_apb_i2c samples that `dma_tx_ack`/`dma_rx_ack` is de-asserted, it can re-assert the `dma_tx_req`/`dma_rx_req` 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 21 on page 72](#) shows a timing diagram of a burst transaction where `pclk = hclk`. [Figure 22 on page 72](#) shows two back-to-back burst transactions where the `hclk` frequency is twice the `pclk` frequency.

The handshaking loop is as follows:

- dma_tx_req/dma_rx_req asserted by DW_apb_i2c
- > dma_tx_ack/dma_rx_ack asserted by DW_ahb_dmac
- > dma_tx_req/dma_rx_req de-asserted by DW_apb_i2c
- > dma_tx_ack/dma_rx_ack de-asserted by DW_ahb_dmac.
- > dma_tx_req/dma_rx_req reasserted by DW_apb_i2c, if back-to-back transaction is required.

Note

The burst transaction request signals, `dma_tx_req` and `dma_rx_req`, are generated in the DW_apb_i2c off `pclk` and sampled in the DW_ahb_dmac by `hclk`. The acknowledge signals, `dma_tx_ack` and `dma_rx_ack`, are generated in the DW_ahb_dmac off `hclk` and sampled in the DW_apb_i2c of `pclk`. The handshaking mechanism between the DW_ahb_dmac and the DW_apb_i2c 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.

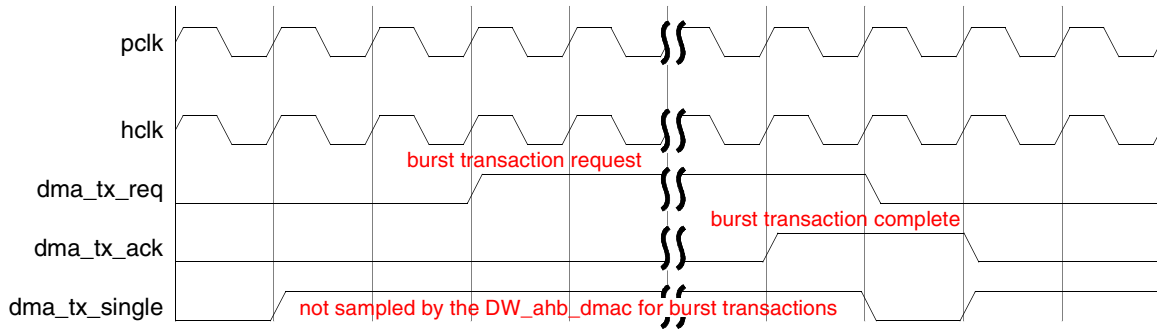


Figure 21: Burst Transaction – pclk = hclk

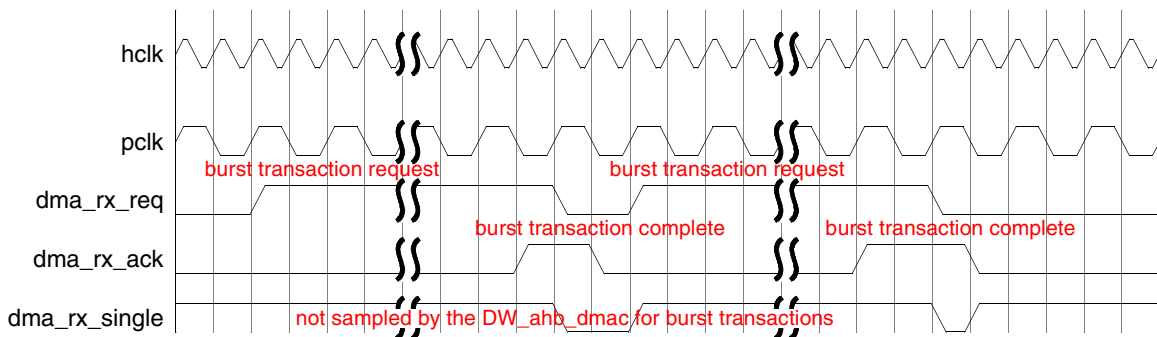


Figure 22: Back-to-Back Burst Transactions – hclk = 2*pclk

Two things to note here:

1. The burst request lines, dma_tx_req signal/dma_rx_req, once asserted remain asserted until their corresponding dma_tx_ack/dma_rx_ack signal is received even if the respective FIFO's drop below their watermark levels during the burst transaction.
2. The dma_tx_req/dma_rx_req signals are de-asserted when their corresponding dma_tx_ack/dma_rx_ack signals are asserted, even if the respective FIFOs exceed their watermark levels.

dma_tx_single, dma_rx_single – The dma_tx_single signal is a status signal. It 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 signal is a status signal. It is asserted when there is at least one valid data entry in the receive FIFO and cleared when the receive FIFO is empty.

These signals are needed by only the DW_ahb_dmac for the case where the block size, CTLx.BLOCK_TS, that is programmed into the DW_ahb_dmac is not a multiple of the burst transaction length, CTLx.SRC_MSIZE, CTLx.DEST_MSIZE, as shown in [Figure 17 on page 67](#). 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_i2c 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_i2c is as follows:

$$\begin{aligned} \text{DMA.CTLx.SRC_MSIZE} &= \text{I2C.iC_DMA_RDLR} + 1 = 4 \\ \text{DMA.CTLx.BLOCK_TS} &= 12 \end{aligned}$$

For the example in [Figure 16 on page 67](#), with the block size set to 12, the `dma_rx_req` signal is asserted when four data items are present in the receive FIFO. The `dma_rx_req` signal is asserted three times during the DW_apb_i2c 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 = I2C.IC_DMA_RDLR + 1 = 4
DMA.CTLx.BLOCK_TS = 15
```

The first 12 data items are transferred as already described using three burst transactions. But when the last three data frames enter the receive FIFO, the `dma_rx_req` signal is not activated because the FIFO level is below the watermark level. The DW_ahb_dmac samples `dma_rx_single` 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 23](#) shows a single transaction. The handshaking loop is as follows:

```
dma_tx_single/dma_rx_single asserted by DW_apb_i2c
-> dma_tx_ack/dma_rx_ack asserted by DW_ahb_dmac
-> dma_tx_single/dma_rx_single de-asserted by DW_apb_i2c
-> dma_tx_ack/dma_rx_ack de-asserted by DW_ahb_dmac.
```

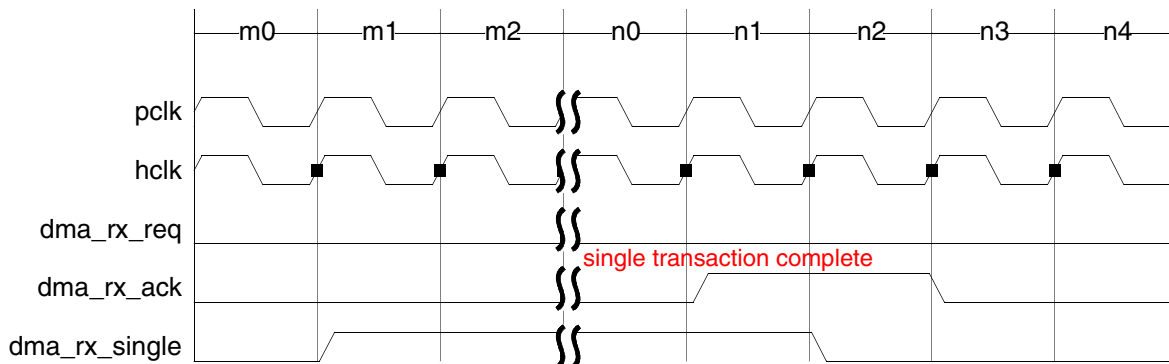


Figure 23: Single Transaction

[Figure 24](#) shows a burst transaction, followed by three back-to-back single transactions, where the `hclk` frequency is twice the `pclk` frequency.

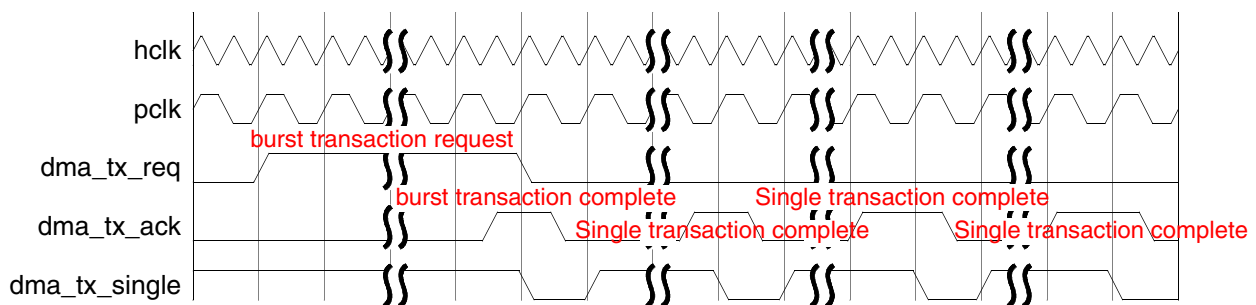


Figure 24: Burst Transaction + 3 Back-to-Back Singles – $hclk = 2 * pclk$

**Note**

The single transaction request signals, `dma_tx_single` and `dma_rx_single`, are generated in the `DW_apb_i2c` on the `pclk` edge and sampled in `DW_ahb_dmac` on `hclk`. The acknowledge signals, `dma_tx_ack` and `dma_rx_ack`, are generated in the `DW_ahb_dmac` on the `hclk` edge and sampled in the `DW_apb_i2c` on `pclk`. The handshaking mechanism between the `DW_ahb_dmac` and the `DW_apb_i2c` supports quasi-synchronous clocks; that is, `hclk` and `pclk` must be phase aligned and the `hclk` frequency must be a multiple of `pclk` frequency.

APB Interface

The host processor accesses data, control, and status information on the `DW_apb_i2c` through the APB interface. The `DW_apb_i2c` supports APB data bus widths of 8, 16, and 32 bits.

For more information about the APB Interface and data widths, refer to [“Integration Considerations” on page 163](#).

4

Parameters

This chapter describes the configuration parameters used by the DW_apb_i2c. The settings of the configuration parameters determine the I/O signal list of the DW_apb_i2c peripheral.

Parameter Descriptions

You use the Synopsys coreConsultant tool to configure the parameters shown in the following tables:

- [“Top-Level Parameters” on page 76](#)
- [“Derived Constants” on page 84](#)

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

Note

There are references to both hardware parameters and software registers throughout this chapter. Parameters and many of the register bits are prefixed with an *IC_**. However, the software register bits are distinguished in this chapter by italics. For instance, *IC_MAX_SPEED_MODE* is a hardware parameter and configured once using Synopsys coreConsultant, whereas the *IC_SLAVE_DISABLE* bit in the *IC_CON* register controls whether I2C has its slave disabled.

Configuration Parameters

You use the Synopsys coreConsultant GUI to configure the following parameters and generate the configured code.

Table 8: Top-Level Parameters

coreConsultant Field Label	Parameter Definition
I2C Source Code Configuration	
Use DesignWare Foundation Synthesis Library (active only when source license available)	<p>Parameter Name: USE_FOUNDATION</p> <p>Legal Values: True (1) or False (0)</p> <p>Default Value: True; only if Source license is available.</p> <p>Dependencies: Must have Source license.</p> <p>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.</p>
System Configuration	
APB data bus width	<p>Parameter Name: APB_DATA_WIDTH</p> <p>Values: 8, 16, or 32</p> <p>Default Value: 8</p> <p>Dependencies: None</p> <p>Description: Width of the APB data bus.</p>
Device Configuration	
Highest speed I2C mode supported	<p>Parameter Name: IC_MAX_SPEED_MODE</p> <p>Values: Standard (1), Fast (2), High (3)</p> <p>Default Value: High (3)</p> <p>Dependencies: None</p> <p>Description: Maximum I²C mode supported. Controls the reset value of the <i>SPEED</i> bit field [2:1] of the I²C Control Register (<i>IC_CON</i>). Count registers are used to generate the outgoing clock SCL on the I²C interface. For the speed modes that are not selected, the corresponding registers are not present in the top-level RTL as described as follows:</p> <ul style="list-style-type: none"> ● If this parameter is set to “Standard,” then the <i>IC_FS_SCL_*</i>, <i>IC_HS_MADDR</i>, and <i>IC_HS_SCL_*</i> registers are not present. ● If this parameter is set to “Fast,” then the <i>IC_HS_MADDR</i>, and <i>IC_HS_SCL_*</i> registers are not present.
Has I2C default slave address of?	<p>Parameter Name: IC_DEFAULT_SLAVE_ADDR</p> <p>Values: 0x000 to 0x3ff</p> <p>Default Value: 0x055</p> <p>Description: Reset value of DW_apb_i2c slave address. Controls the reset value of the I²C Slave Address Register (<i>IC_SAR</i>). The default values cannot be any of the reserved address locations: 0x00 to 0x07 or 0x78 to 0x7f.</p>

Table 8: Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
Has I ² C default target slave address of?	<p>Parameter Name: IC_DEFAULT_TAR_SLAVE_ADDR</p> <p>Value: 0x000 to 0x3ff</p> <p>Default Value: 0x055</p> <p>Description: Reset value of DW_apb_i2c target slave address. Controls the reset value of the IC_TAR bit field (9:0) of the I²C Target Address Register (<i>IC_TAR</i>). The default values cannot be any of the reserved address locations: 0x00 to 0x07 or 0x78 to 0x7f.</p>
Has High Speed mode master code of?	<p>Parameter Name: IC_HS_MASTER_CODE</p> <p>Values: 0x0 to 0x7</p> <p>Default Value: 0x1</p> <p>Dependencies: None</p> <p>Description: High-speed mode master code of DW_apb_i2c. Controls the reset value of the I²C HS Master Mode Code Address Register (<i>IC_HS_MADDR</i>). This is a unique code that alerts other masters on the I²C bus that a high-speed mode transfer is going to begin. For more information about this code, refer to “Multiple Master Arbitration” on page 54.</p>
Is an I ² C master?	<p>Parameter Name: IC_MASTER_MODE</p> <p>Values: Unchecked (0) or Checked (1)</p> <p>Default Value: Checked (1)</p> <p>Dependencies: None</p> <p>Description: Controls whether DW_apb_i2c is enabled to be a master after reset. This parameter controls the reset value of bit 0 of the I²C Control Register (<i>IC_CON</i>). To enable the component to be a master, you must write a 1 in bit 0 of the <i>IC_CON</i> register.</p> <p>NOTE: If this parameter is checked (1), then you must ensure that the parameter IC_SLAVE_DISABLE is checked (1) as well.</p>
Disable Slave after reset?	<p>Parameter Name: IC_SLAVE_DISABLE</p> <p>Values: Unchecked (0), Checked (1)</p> <p>Default Value: Unchecked (0)</p> <p>Dependencies: None.</p> <p>Description: Controls whether DW_apb_i2c has its slave enabled or disabled after reset. If checked, the DW_apb_i2c slave interface is disabled after reset. The slave also can be disabled by programming a 1 into bit 6 of the I²C Control Register (<i>IC_CON</i>). By default, the slave is enabled.</p> <p>NOTE: If this parameter is unchecked (0), then you must ensure that the parameter IC_MASTER_MODE is unchecked (0) as well.</p>
Supports 10-bit addressing in master mode?	<p>Parameter Name: IC_10BITADDR_MASTER</p> <p>Values: Unchecked (0) or Checked (1)</p> <p>Default Value: Checked (1)</p> <p>Dependencies: None</p> <p>Description: Controls whether DW_apb_i2c supports 7- or 10-bit addressing on the I²C interface after reset when acting as a master. Controls the reset value of bit 4 of the I²C Control Register (<i>IC_CON</i>). Master-generated transfers use this number of address bits. Additionally, it can be reprogrammed by software by writing to the <i>IC_CON</i> register.</p>

Table 8: Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
Supports 10-bit addressing in slave mode?	<p>Parameter Name: IC_10BITADDR_MASTER</p> <p>Values: Unchecked (0) or Checked (1)</p> <p>Default Value: Checked (1)</p> <p>Dependencies: None</p> <p>Description: Controls whether DW_apb_i2c slave supports 7- or 10-bit addressing on the I2C interface after reset when acting as a slave. Controls reset value of part of the <i>IC_CON</i> register. DW_apb_i2c responds to this number of address bits when acting as a slave; it can be programmed by software.</p>
Depth of transmit buffer is?	<p>Parameter Name: IC_TX_BUFFER_DEPTH</p> <p>Values: 2 to 256</p> <p>Default Value: 8</p> <p>Dependencies: None</p> <p>Description: Depth of the transmit buffer. The buffer is 9-bits wide; 8 bits for the data, and 1 bit for the read or write command.</p>
Depth of receive buffer is?	<p>Parameter Name: IC_RX_BUFFER_DEPTH</p> <p>Values: 2 to 256</p> <p>Default Value: 8</p> <p>Dependencies: None</p> <p>Description: Depth of receive buffer; the buffer is 8 bits wide.</p>
Transmit buffer threshold level is?	<p>Parameter Name: IC_TX_TL</p> <p>Values: 0 to (IC_TX_BUFFER_DEPTH – 1)</p> <p>Default Value: 0</p> <p>Dependencies: None</p> <p>Description: Reset value for the threshold level of the transmit buffer. This parameter controls the reset value of the I²C Transmit FIFO Threshold Level Register (<i>IC_TX_TL</i>).</p>
Receive buffer threshold value is?	<p>Parameter Name: IC_RX_TL</p> <p>Values: 0 to (IC_RX_BUFFER_DEPTH – 1)</p> <p>Default Value: 0</p> <p>Dependencies: None</p> <p>Description: Reset value for the threshold level of the receive buffer. This parameter controls the reset value of the I²C Receive FIFO Threshold Level Register (<i>IC_RX_TL</i>).</p>

Table 8: Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
Allow restart conditions to be sent when acting as a master?	<p>Parameter Name: IC_RESTART_EN</p> <p>Values: Checked (1) or Unchecked (0)</p> <p>Default Value: Checked (1)</p> <p>Dependencies: None</p> <p>Description: Controls the reset value of bit 5 (<i>IC_RESTART_EN</i>) in the <i>IC_CON</i> register. By default, this parameter is checked, which allows RESTART conditions to be sent when DW_apb_i2c is acting as a master. Some older slaves do not support handling RESTART conditions; however, RESTART conditions are used in several I²C operations. When the RESTART is disabled, the master is prohibited from performing the following functions:</p> <ul style="list-style-type: none"> ● Change direction within a transfer (split) ● Send a START BYTE ● Perform any high-speed mode operation ● Perform combined format transfers in 7-bit addressing modes ● Perform a read operation with a 10-bit address ● Send multiple bytes per transfer
Hardware reset value for IC_SDA_SETUP register	<p>Parameter Name: IC_DEFAULT_SDA_SETUP</p> <p>Legal Values: 0x00 to 0xff</p> <p>Default Value: 0x64</p> <p>Dependencies: None</p> <p>Description: Assigns the default reset value for the IC_SDA_SETUP register.</p>
IC_ACK_GENERAL_CALL set to acknowledge I ² C general calls on reset	<p>Parameter Name: IC_DEFAULT_ACK_GENERAL_CALL</p> <p>Unchecked (0) or Checked (1)</p> <p>Default Value: Checked (1)</p> <p>Dependencies: None</p> <p>Description: Assigns the default reset value for the IC_ACK_GENERAL_CALL register.</p>
External Configuration	
Include DMA handshaking interface signals?	<p>Parameter Name: IC_HAS_DMA</p> <p>Values: Checked (1) or Unchecked (0)</p> <p>Default Value: Unchecked (0)</p> <p>Dependencies: None</p> <p>Description: When checked, includes the DMA handshaking interface signals at the top-level I/O. For more information about these signals, see “DW_apb_i2c Signal Descriptions” on page 88.</p>
Single Interrupt output port present?	<p>Parameter Name: IC_INTR_IO</p> <p>Values: Unchecked (0) or Checked (1)</p> <p>Default Value: Unchecked (0)</p> <p>Dependencies: None</p> <p>Description: If unchecked, each interrupt source has its own output. If checked, all interrupt sources are combined into a single output.</p>

Table 8: Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
Polarity of interrupts is active high?	<p>Parameter Name: IC_INTR_POL</p> <p>Values: Unchecked (0) or Checked (1)</p> <p>Default Value: Checked (1)</p> <p>Dependencies: None</p> <p>Description: By default, the polarity of the output interrupt lines is active high (checked).</p>
Internal Configuration	
Add Encoded Parameters	<p>Parameter Name: IC_ADD_ENCODED_PARAMS</p> <p>Values: Unchecked (0) or Checked (1)</p> <p>Default Value: Checked (1)</p> <p>Dependencies: None</p> <p>Description: By adding the encoded parameters gives firmware an easy and quick way of identifying the DesignWare component within an I/O memory map. Some critical design-time options determine how a driver should interact with the peripheral. There is a minimal area overhead by including these parameters. This option allows a single driver to be developed for each component, which will be self-configurable.</p> <p>When bit 7 of the <i>IC_COMP_PARAM_1</i> is read and contains a '1,' the encoded parameters can be read via software. If this bit is a '0,' then the entire register is '0' regardless of the setting of any of the other parameters that are encoded in the register's bits. For details about this register, see the <i>IC_COMP_PARAM_1</i> register on page 151.</p>
Specify clock counts directly instead of supplying clock frequency?	<p>Parameter Name: IC_USE_COUNTS</p> <p>Values: Checked (1) or Unchecked (0)</p> <p>Default Value: Unchecked (0)</p> <p>Dependencies: None</p> <p>Description: Determines whether *CNT values are provided directly or by specifying the ic_clk clock frequency and letting coreConsultant (or coreAssembler) calculate the count values.</p> <p>When this parameter is checked, the reset values of the *CNT registers are specified by the corresponding *COUNT configuration parameters, which may be user-defined or derived (see standard, fast, or high speed mode parameters later in this table).</p> <p>When unchecked (default setting), the reset values of the *CNT registers are calculated from the configuration parameter IC_CLOCK_PERIOD.</p>
Hard code the count values for each mode?	<p>Parameter Name: IC_HC_COUNT_VALUES</p> <p>Values: Checked (1) or Unchecked (0)</p> <p>Default Value: Unchecked (0)</p> <p>Dependencies: None.</p> <p>Description: By checking this parameter, the *CNT registers are set to read only. Unchecking this parameter (default setting) allows the *CNT registers to be writable.</p> <p>Regardless of the setting, the *CNT registers are always readable and have reset values from the corresponding *COUNT configuration parameters, which may be user-defined or derived (see standard, fast, or high speed mode parameters later in this table). The count registers begin on page 113.</p>

Table 8: Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
ic_clk has a period of? (ns integers only)	<p>Parameter Name: IC_CLOCK_PERIOD</p> <p>Values: 2 to 2147483647 (ns)</p> <p>Default Value: 10 (ns) – high-speed mode</p> <p>Dependencies: This parameter is disabled if the IC_USE_COUNTS parameter is checked (1).</p> <p>Description: Specifies the period of incoming ic_clk, which is used to generate outgoing I²C interface SCL clock (ns integers only). When the count values are used to generate the IC_CLOCK_PERIOD, then the IC_MAX_SPEED_MODE setting determines the actual period:</p> <p>IC_MAX_SPEED_MODE = Standard => 500 ns IC_MAX_SPEED_MODE = Fast => 100 ns IC_MAX_SPEED_MODE = High => 10 ns</p>
Relationship between pclk and ic_clk is?	<p>Parameter Name: IC_CLK_TYPE</p> <p>Values: Identical (0), Synchronous (3), Asynchronous (1)</p> <p>Default Value: Asynchronous (1)</p> <p>Dependencies: None.</p> <p>Description: Specifies the relationship between pclk and ic_clk.</p> <p>NOTE: ic_clk frequency must be greater than or equal to pclk frequency.</p> <p>Identical (0): clocks are identical; no metastability flops are required for data passing between clock domains.</p> <p>Synchronous (3): clocks are not the same frequency but have coincident rising edges. Synchronization flops are required for data passing between clock domains because the clocks can be different frequencies on either side. These flops are holding the registers longer to ensure that the synchronization is registered.</p> <p>Asynchronous (1): clocks may be completely asynchronous to each other, metastability flops are required for data passing between clock domains.</p>
Standard Speed Mode Configuration	
Std speed SCL high count is?	<p>Parameter Name: IC_SS_SCL_HIGH_COUNT</p> <p>Values: Hex value in range 0x0006 to 0xffff</p> <p>Default Value: 0x0190 (400 based on 100 MHz ic_clk)</p> <p>Dependencies: This parameter is active when the IC_USE_COUNTS parameter is checked (1); otherwise, this value is automatically calculated using the IC_CLK_PERIOD parameter. If the IC_MAX_SPEED_MODE parameter is set to “fast” or “high”, this parameter is irrelevant.</p> <p>Description: Reset value of Standard Speed I²C Clock SCL High Count register (<i>IC_SS_SCL_HCNT</i>). The value must be calculated based on the I²C data rate desired and I²C clock frequency. For more information, see the <i>IC_SS_SCL_HCNT</i> register on page 113.</p>

Table 8: Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
Std speed SCL low count is?	<p>Parameter Name: IC_SS_SCL_LOW_COUNT</p> <p>Values: Hex value in range 0x0008 to 0xffff</p> <p>Default Value: 0x01d6 (470 based on 100 MHz ic_clk)</p> <p>Dependencies: This parameter is active when the IC_USE_COUNTS parameter is checked (1); otherwise, this value is automatically calculated using the IC_CLK_PERIOD parameter. If the IC_MAX_SPEED_MODE parameter is set to “fast” or “high”, this parameter is irrelevant.</p> <p>Description: Reset value of Standard Speed I2C Clock SCL Low Count register (<i>IC_SS_SCL_HCNT</i>). Value must be calculated based on I²C data rate desired and I²C clock frequency. For more information, see IC_SS_SCL_LCNT register on page 115. When parameter IC_USE_COUNTS = 0, this parameter is automatically calculated using the IC_CLK_PERIOD parameter.</p>
Fast Speed Mode	
Fast speed SCL high count is?	<p>Parameter Name: IC_FS_SCL_HIGH_COUNT</p> <p>Values: Hex value in range 0x0006 to 0xffff</p> <p>Default Value: 0x003c (60 based on 100 MHz ic_clk)</p> <p>Dependencies: This parameter is active when the IC_USE_COUNTS parameter is checked (1); otherwise, this value is automatically calculated using the IC_CLK_PERIOD parameter. If the IC_MAX_SPEED_MODE parameter is set to “standard” or “high”, this parameter is irrelevant.</p> <p>Description: Reset value of Fast Speed I2C Clock SCL High Count register (<i>IC_FS_SCL_HCNT</i>). Value must be calculated based on I²C data rate desired and I²C clock frequency. For more information, see IC_FS_SCL_HCNT register on page 116.</p>
Fast speed SCL low count is?	<p>Parameter Name: IC_FS_SCL_LOW_COUNT</p> <p>Values: Hex value in range 0x0008 to 0xffff</p> <p>Default Value: 0x0082 (130 based on 100 MHz ic_clk)</p> <p>Dependencies: This parameter is active when the IC_USE_COUNTS parameter is checked (1); otherwise, this value is automatically calculated using the IC_CLK_PERIOD parameter. If the IC_MAX_SPEED_MODE parameter is set to “standard” or “high” this parameter is irrelevant.</p> <p>Description: Reset value of Fast Speed I2C Clock SCL Low Count register (<i>IC_FS_SCL_LCNT</i>). Value must be calculated based on I²C data rate and I2C clock frequency. For more information, see the IC_FS_SCL_LCNT register on page 118.</p>
High Speed Mode	
For high speed mode systems the I ² C bus loading is? (pF)	<p>Parameter Name: IC_CAP_LOADING</p> <p>Values: 100 or 400</p> <p>Default Value: 100</p> <p>Dependencies: This parameter is not present in non-high speed mode systems (IC_MAX_SPEED_MODE != high).</p> <p>Description: For high-speed mode, the bus loading affects the high and low pulse width of SCL.</p>

Table 8: Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
High speed SCL high count is?	<p>Parameter Name: IC_HS_SCL_HIGH_COUNT</p> <p>Values: Hex value in range 0x0006 to 0xffff</p> <p>Default Value: 0x006 (6 based on 100 MHz ic_clk, 400pF bus loading)</p> <p>Dependencies: This parameter is active when the IC_USE_COUNTS parameter is checked (1); otherwise, this value is automatically calculated using the IC_CLK_PERIOD parameter. If the IC_MAX_SPEED_MODE parameter is set to “standard” or “fast”, this parameter is irrelevant.</p> <p>Description: Reset value of High Speed I2C Clock SCL High Count register (<i>IC_HS_SCL_HCNT</i>). Value must be calculated based on I²C data rate desired and high speed I²C clock frequency. For more information, see IC_HS_SCL_HCNT register on page 120.</p>
High speed SCL low count is?	<p>Parameter Name: IC_HS_SCL_LOW_COUNT</p> <p>Values: Hex value in range 0x0008 to 0xffff</p> <p>Default Value: 0x0010 (16 based on 100 MHz ic_clk, 400pF bus loading)</p> <p>Dependencies: This parameter is active when the IC_USE_COUNTS parameter is checked (1); otherwise, this value is automatically calculated using the IC_CLK_PERIOD parameter. If the IC_MAX_SPEED_MODE parameter is set to “standard” or “fast”, this parameter is irrelevant.</p> <p>Description: Reset value of High Speed I2C Clock SCL Low Count register (<i>IC_HS_SCL_LCNT</i>). The value must be calculated based on I2C data rate and I2C clock frequency. For more information, see IC_HS_SCL_LCNT register on page 122.</p>
Additional Features	
Allow dynamic updating of the TAR address?	<p>Parameter Name: I2C_DYNAMIC_TAR_UPDATE</p> <p>Values: Unchecked (0) or Checked (1)</p> <p>Default Value: Unchecked (0)</p> <p>Dependencies: None</p> <p>Description: When checked, allows the IC_TAR register to be updated dynamically even while the slave interface of DW_apb_i2c is involved in an I²C transfer. The setting of this parameter affects the operation of DW_apb_i2c when it is in master mode. For more details, see “Master Mode Operation” on page 60.</p>

Table 8: Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
Enable register to generate NACKs for data received by Slave?	<p>Parameter Name: IC_SLV_DATA_NACK_ONLY</p> <p>Values: Unchecked (0) or Checked (1)</p> <p>Default Value: Unchecked (0)</p> <p>Dependencies: None</p> <p>Description: Enables an additional register to control whether DW_apb_i2c generates a NACK after a data byte has been transferred to it. This NACK generation only occurs when DW_apb_i2c is a slave-receiver. If this register is set to a value of 1, it can only generate a NACK after a data byte is received; hence, the data transfer is aborted and the data received is not pushed to the receive buffer.</p> <p>When the register is set to a value of 0, it generates NACK/ACK, depending on normal criteria. If this option is selected, the default value of the IC_SLV_DATA_NACK_ONLY register is always 0. The register must be explicitly programmed to a value of 1 if NACKs are to be generated. The register can only be written to successfully if DW_apb_i2c is disabled (IC_ENABLE[0] = 0) or the slave part is inactive (IC_STATUS[6] = 0).</p>

The following table includes parameters that are derived from the user selected parameters in coreConsultant.

Table 9: Derived Constants

Parameter	Legal Range	Description
TX_ABW	1 to 8 Default: 3	Transmit data width of FIFO (for writes).
RX_ABW	1 to 8 Default: 3	Receive data width of FIFO (for reads)

These constants in [Table 9](#) are derived using the following equation:

$$X = \text{IC_TX_BUFFER_DEPTH}$$

$$\text{Log}_2(\text{IC_TX_BUFFER_DEPTH}) \text{ rounded up to the nearest integer}$$

5

Signals

The following subsections describe the DW_apb_i2c I/O signals:

- [“DW_apb_i2c Interface Diagram” on page 86](#)
- [“I/O Connections” on page 87](#)
- [“DW_apb_i2c Signal Descriptions” on page 88](#)

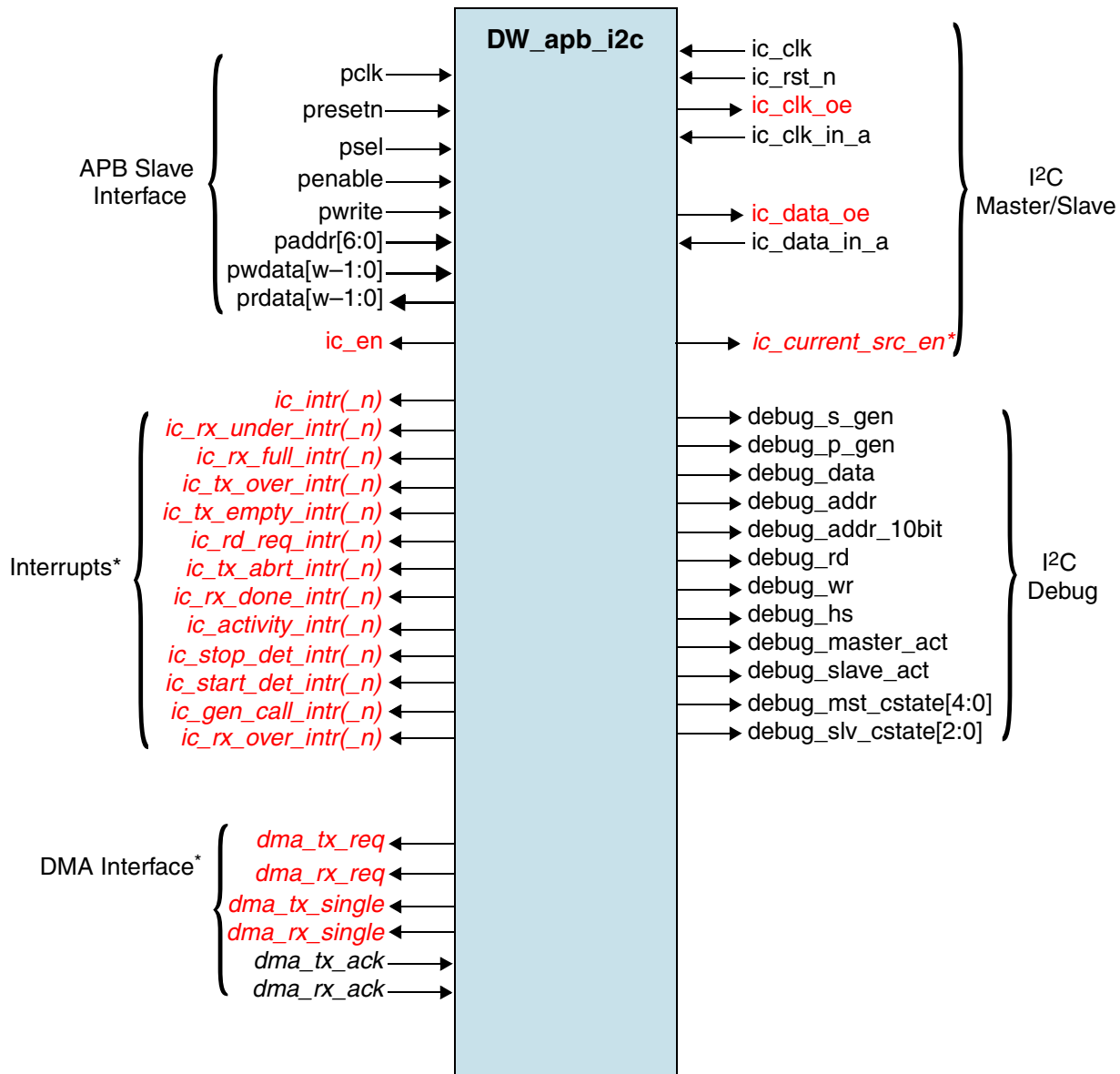


Note

There are references to both hardware parameters and software registers throughout this chapter. Both have prefixed with an *IC_**. However, the software registers are distinguished by italics. For instance, *IC_MAX_SPEED_MODE* is a hardware parameter and configured once using Synopsys coreConsultant, whereas *IC_ENABLE* is a software register that enables the DW_apb_i2c.

DW_apb_i2c Interface Diagram

Figure 25 shows the interface diagram for DW_apb_i2c.



* These signals in italics are optional depending on configuration parameters set in coreAssembler or coreConsultant.

w = 8, 16, or 32 corresponding to APB_DATA_WIDTH

Signals in red are registered. For more information about these signals, refer to Table 10 on page 88.

Figure 25: DW_apb_i2c Interface Diagram

I/O Connections

As illustrated in [Figure 26](#), the I²C interface consists of two wires, a clock (SCL) and data (SDA). For high-speed systems, the names are SCLH and SDAH. For high-speed mode, a current source pull-up may be used on the SCLH line. It is enabled during some active master transactions. The SDA and SDAH connections are the same at any speed. There are no special connections required for the DesignWare AMBA APB slave interface side of the DW_apb_i2c.

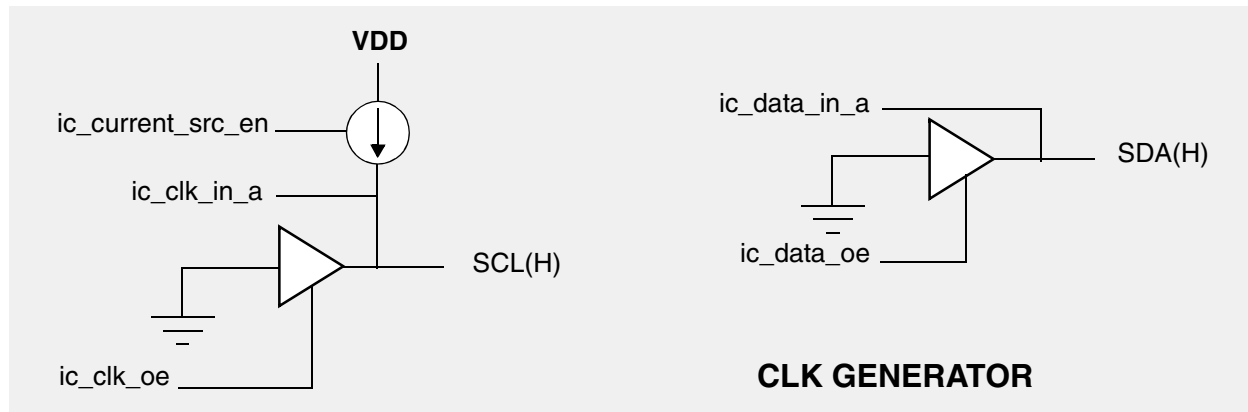


Figure 26: I/O Connection to I²C Interface

DW_apb_i2c Signal Descriptions

Table 10 identifies the signals that are associated with the DW_apb_i2c. The signals in italics are optional depending on configuration parameter settings. The debug signals give visibility to the internals of the DW_apb_i2c design. They are used only for observation and serve no other purpose.



Note

The **Input/Output Delay** fields in the Description column list the default external input or output delays. You can change these values by completing the Specify Clocks activity in coreAssembler or coreConsultant. For more information, refer to “[Create Gate-Level Netlist](#)” on page 30.

Table 10: DW_apb_i2c Signal Description

Name	Width	I/O	Description
APB Slave Interface			
pclk	1 bit	In	APB clock for the bus interface unit. NOTE: ic_clk frequency must be greater than or equal to pclk frequency. Active State: N/A Synchronous to: The configuration parameter IC_CLK_TYPE indicates the relationship between pclk and ic_clk. It can be asynchronous (1), synchronous (3), or identical (0). For more information about this parameter, refer to page 81 . Registered: No Default Input Delay: N/A
presetn	1 bit	In	An APB interface domain reset. Active State: Low Synchronous to: The signal is asserted asynchronously, but is deasserted synchronously after the rising edge of pclk. The synchronization must be provided external to this component. Registered: No Default Input Delay: 30%
pssel	1 bit	In	APB peripheral select that lasts for two pclk cycles. When asserted, indicates that the peripheral has been selected for a read/write operation. Active State: High Synchronous to: pclk Registered: No Default Input Delay: 30%
penable	1 bit	In	APB enable control. Asserted for a single pclk cycle and used for timing read/write operations. Active State: High Synchronous to: pclk Registered: No Default Input Delay: 30%

Table 10: DW_apb_i2c Signal Description (Continued)

Name	Width	I/O	Description
pwrite	1 bit	In	APB write control. When high, indicates a write access to the peripheral; when low, indicates a read access. Active State: N/A Synchronous to: pclk Registered: No Default Input Delay: 30%
paddr	7 bits	In	APB address bus. Uses lower 7 bits of the address bus for register decode. Active State: N/A Synchronous to: N/A Registered: No Default Input Delay: 30%
pwdata	w-1:0	In	APB write data bus. Driven by the bus master (DW_ahb to DW_apb bridge) during write cycles. Can be 8, 16, or 32 bits wide depending on APB_DATA_WIDTH parameter. Active State: N/A Synchronous to: N/A Registered: No Default Input Delay: 30%
prdata	w-1:0	Out	APB readback data. Driven by the selected peripheral during read cycles. Can be 8, 16, or 32 bits wide depending on APB_DATA_WIDTH parameter. Active State: N/A Synchronous to: N/A Registered: Yes Default Output Delay: 10%
I²C Interface (Master/Slave)			
ic_clk	1 bit	In	Peripheral clock. DW_apb_i2c runs on this clock and is used to clock transfers in standard, fast, and high-speed mode. NOTE: ic_clk frequency must be greater than or equal to pclk frequency. Active State: N/A Synchronous to: The configuration parameter IC_CLK_TYPE indicates the relationship between pclk and ic_clk. It can be asynchronous (1), synchronous (3), or identical (0). For more information about this parameter, see page 81 . Registered: No Default Input Delay: N/A
ic_rst_n	1 bit	In	I ² C reset. Used to reset flip-flops that are clocked by the ic_clk clock. Active State: Low Synchronous to: ic_clk Registered: No Default Input Delay: 30%

Table 10: DW_apb_i2c Signal Description (Continued)

Name	Width	I/O	Description
ic_clk_oe	1 bit	Out	Outgoing I ² C clock. Open drain synchronous with ic_clk. Active State: High Synchronous to: ic_clk Registered: Yes Default Output Delay: 30%
ic_clk_in_a	1 bit	In	Incoming I ² C clock. This is the input SCL signal. Double-registered for metastability synchronisation and glitch-suppressed using 2-out-of-3 majority vote circuit. NOTE: DW_apb_i2c provides filtering on the SDA (ic_data_in_a) and SCL (ic_clk_in_a) inputs, suppressing noise and signal spikes with durations less than one ic_clk period. Active State: High Synchronous to: This signal is asynchronous to ic_clk. Registered: No Default Input Delay: N/A
ic_data_oe	1 bit	Out	Outgoing I ² C Data. Open Drain Synchronous to ic_clk. Active State: High Synchronous to: ic_clk Registered: Yes Default Output Delay: 30%
ic_data_in_a	1 bit	In	Incoming I ² C Data. It is the input SDA signal. Double-registered for metastability synchronisation and glitch-suppressed using 2-out-of-3 majority vote circuit. NOTE: DW_apb_i2c provides filtering on the SDA (ic_data_in_a) and SCL (ic_clk_in_a) inputs, suppressing noise and signal spikes with durations less than one ic_clk period. Active State: High Synchronous to: This signal is asynchronous to ic_clk. Registered: No Default Input Delay: N/A
ic_en	1 bit	Out	I ² C interface enable. Indicates whether DW_apb_i2c is enabled; this signal is set to 0 when the IC_ENABLE register is set to 0 (disabled). Because DW_apb_i2c always finishes its current transfer before turning off ic_en, this signal may be used by a clock generator to control whether the DW_apb_i2c ic_clk is active or inactive. Active State: Low Synchronous to: pclk Registered: Yes Default Output Delay: 30%

Table 10: DW_apb_i2c Signal Description (Continued)

Name	Width	I/O	Description
<i>ic_current_src_en</i>	1 bit	Out	<p><i>Optional.</i> Current source pull-up. Controls the polarity of the current source pull-up on the SCLH. This pull-up is used to shorten the rise time on SCLH by activating an user-supplied external current source pull-up circuit. It is disabled after a RESTART condition and after each A/A bit when acting as the active master.</p> <p>This signal enables other devices to delay the serial transfer by stretching the LOW period of the SCLH signal. The active master re-enables its current source pull-up circuit again when all devices have released and the SCLH signal reaches high level, therefore, shortening the last part of the SCLH signal's rise time.</p> <p>Active State: Low Synchronous to: <i>ic_clk</i> Registered: Yes Default Output Delay: 30%</p> <p>Dependencies: This current source is necessary for only high-speed mode operation. This signal is present only if the configuration parameter <i>IC_MAX_SPEED_MODE</i> = high.</p>
Interrupts			
<i>ic_intr(_n)</i>	1 bit	Out	<p><i>Optional.</i> Combined interrupt. This signal is included on the interface when the configuration parameter <i>IC_INTR_IO</i> is checked (1) to indicate that only one interrupt line appears on the I/O (as opposed to individual interrupt signals).</p> <p>Active State: High. Polarity is set by the configuration parameter <i>IC_INTR_POL</i> (checked = active high). When <i>IC_INTR_POL</i> is unchecked (0), the <i>ic_intr_n</i> signal is included on the interface to indicate active low polarity.</p> <p>Synchronous to: <i>pclk</i> Registered: Yes Default Output Delay: 30%</p>
<i>ic_rx_over_intr(_n)</i>	1 bit	Out	<p><i>Optional.</i> Receive buffer overflow interrupt. This signal is included on the interface when the configuration <i>IC_INTR_IO</i> parameter is unchecked (0), which indicates that individual interrupt lines appear on the I/O.</p> <p>When the module is disabled, this interrupt keeps its level until the master or slave state machines go into idle and bit 0 of the <i>IC_ENABLE</i> register is 0. When <i>ic_en</i> goes to 0, this interrupt is cleared.</p> <p>Active State: High. Polarity is set by the configuration parameter <i>IC_INTR_POL</i> (checked = active high). When <i>IC_INTR_POL</i> is unchecked (0), the <i>ic_rx_over_intr_n</i> signal is included on the interface to indicate active low polarity.</p> <p>Synchronous to: <i>pclk</i> Registered: Yes Default Output Delay: 30%</p>

Table 10: DW_apb_i2c Signal Description (Continued)

Name	Width	I/O	Description
<i>ic_rx_under_intr(_n)</i>	1 bit	Out	<p><i>Optional.</i> Receive buffer underflow interrupt. This signal is included on the interface when the configuration IC_INTR_IO parameter is unchecked (0), which indicates that individual interrupt lines appear on the I/O.</p> <p>When the module is disabled, this interrupt keeps its level until the master or slave state machines go into idle and bit 0 of the IC_ENABLE register is 0. When ic_en goes to 0, this interrupt is cleared.</p> <p>Active State: High. Polarity is set by the configuration parameter IC_INTR_POL (checked = active high). When IC_INTR_POL is unchecked (0), the ic_rx_under_intr_n signal is included on the interface to indicate active low polarity</p> <p>Synchronous to: pclk</p> <p>Registered: Yes</p> <p>Default Output Delay: 30%</p>
<i>ic_tx_over_intr(_n)</i>	1 bit	Out	<p><i>Optional.</i> Transmit buffer overflow interrupt. This signal is included on the interface when the configuration IC_INTR_IO parameter is unchecked (0), which indicates that individual interrupt lines appear on the I/O.</p> <p>When the module is disabled, this interrupt keeps its level until the master or slave state machines go into idle and bit 0 of the IC_ENABLE register is 0. When ic_en goes to 0, this interrupt is cleared.</p> <p>Active State: High. Polarity is set by the configuration parameter IC_INTR_POL (checked = active high). When IC_INTR_POL is unchecked (0), the ic_tx_over_intr_n signal is included on the interface instead to indicate active low polarity.</p> <p>Synchronous to: pclk</p> <p>Registered: Yes</p> <p>Default Output Delay: 30%</p>
<i>ic_tx_abrt_intr(_n)</i>	1 bit	Out	<p><i>Optional.</i> Transmit abort interrupt.</p> <p>Active State: High. Polarity is set by the configuration parameter IC_INTR_POL (checked = active high). When IC_INTR_POL is unchecked (0), the ic_tx_abrt_intr_n signal is included on the interface instead to indicate active low polarity.</p> <p>Synchronous to: pclk</p> <p>Registered: Yes</p> <p>Default Output Delay: 30%</p>

Table 10: DW_apb_i2c Signal Description (Continued)

Name	Width	I/O	Description
<i>ic_rx_done_intr(_n)</i>	1 bit	Out	<p><i>Optional.</i> Receive done interrupt. This signal is included on the interface when the configuration IC_INTR_IO parameter is unchecked (0), which indicates that individual interrupt lines appear on the I/O.</p> <p>Active State: High. Polarity is set by the configuration parameter IC_INTR_POL (checked = active high). When IC_INTR_POL is unchecked (0), the <i>ic_rx_done_intr_n</i> signal is included on the interface instead to indicate active low polarity.</p> <p>Synchronous to: pclk</p> <p>Registered: Yes</p> <p>Default Output Delay: 30%</p>
<i>ic_rx_full_intr(_n)</i>	1 bit	Out	<p><i>Optional.</i> Receive buffer full interrupt. This signal is included on the interface when the configuration IC_INTR_IO parameter is unchecked (0), which indicates that individual interrupt lines appear on the I/O.</p> <p>When bit 0 of the IC_ENABLE register is 0, the RX FIFO is flushed and held in reset—the RX FIFO is not full—so this <i>ic_rx_full_intr</i> bit is cleared once the <i>ic_enable</i> bit is programmed with a 0, regardless of the activity that continues.</p> <p>Active State: High. Polarity is set by the configuration parameter IC_INTR_POL (checked = active high). When IC_INTR_POL is unchecked (0), the <i>ic_rx_full_intr_n</i> signal is included on the interface instead to indicate active low polarity.</p> <p>Synchronous to: pclk</p> <p>Registered: Yes</p> <p>Default Output Delay: 30%</p>
<i>ic_rd_req_intr(_n)</i>	1 bit	Out	<p><i>Optional.</i> Slave read request interrupt. This signal is included on the interface when the configuration IC_INTR_IO parameter is unchecked (0), which indicates that individual interrupt lines appear on the I/O.</p> <p>Active State: High. Polarity is set by the configuration parameter IC_INTR_POL (checked = active high). When IC_INTR_POL is unchecked (0), the <i>ic_rd_req_intr_n</i> signal is included on the interface instead to indicate active low polarity.</p> <p>Synchronous to: pclk</p> <p>Registered: Yes</p> <p>Default Output Delay: 30%</p>

Table 10: DW_apb_i2c Signal Description (Continued)

Name	Width	I/O	Description
<i>ic_tx_empty_intr(_n)</i>	1 bit	Out	<p><i>Optional.</i> Transmit buffer empty interrupt. This signal is included on the interface when the configuration IC_INTR_IO parameter is unchecked (0), which indicates that individual interrupt lines appear on the I/O.</p> <p>When bit 0 of the IC_ENABLE register is 0, the TX FIFO is flushed and held in reset, where it looks like it has no data within it. The ic_tx_empty_intr bit is raised when bit 0 of the IC_ENABLE register is 0, provided there is activity in the master or slave state machines. When there is no longer activity, then this interrupt bit is masked with ic_en.</p> <p>Active State: High. Polarity is set by the configuration parameter IC_INTR_POL (checked = active high). When IC_INTR_POL is unchecked (0), the ic_tx_empty_intr_n bit is included on the interface instead to indicate active low polarity.</p> <p>Synchronous to: pclk</p> <p>Registered: Yes</p> <p>Default Output Delay: 30%</p>
<i>ic_activity_intr(_n)</i>	1 bit	Out	<p><i>Optional.</i> I2C activity interrupt. This signal is included on the interface when the configuration IC_INTR_IO parameter is unchecked (0), which indicates that individual interrupt lines appear on the I/O.</p> <p>Active State: High. Polarity is set by the configuration parameter IC_INTR_POL (checked = active high). When IC_INTR_POL is unchecked (0), the ic_activity_intr_n signal is included on the interface instead to indicate active low polarity.</p> <p>Synchronous to: pclk</p> <p>Registered: Yes</p> <p>Default Output Delay: 30%</p>
<i>ic_stop_det_intr(_n)</i>	1 bit	Out	<p><i>Optional.</i> Stop condition detect on I2C interrupt. This signal is included on the interface when the configuration IC_INTR_IO parameter is unchecked (0), which indicates that individual interrupt lines appear on the I/O.</p> <p>Active State: High. Polarity is set by the configuration parameter IC_INTR_POL (checked = active high). When IC_INTR_POL is unchecked (0), the ic_stop_det_intr_n signal is included on the interface instead to indicate active low polarity.</p> <p>Synchronous to: pclk</p> <p>Registered: Yes</p> <p>Default Output Delay: 30%</p>

Table 10: DW_apb_i2c Signal Description (Continued)

Name	Width	I/O	Description
<i>ic_start_det_intr(_n)</i>	1 bit	Out	<p><i>Optional.</i> Start condition detect on I2C interrupt. This signal is included on the interface when the configuration IC_INTR_IO parameter is unchecked (0), which indicates that individual interrupt lines appear on the I/O.</p> <p>Active State: High. Polarity is set by the configuration parameter IC_INTR_POL (checked = active high). When IC_INTR_POL is unchecked (0), the <i>ic_start_det_intr_n</i> signal is included on the interface instead to indicate active low polarity.</p> <p>Synchronous to: pclk</p> <p>Registered: Yes</p> <p>Default Output Delay: 30%</p>
<i>ic_gen_call_intr(_n)</i>	1 bit	Out	<p><i>Optional.</i> General Call received interrupt. This signal is included on the interface when the configuration IC_INTR_IO parameter is unchecked (0), which indicates that individual interrupt lines appear on the I/O.</p> <p>Active State: High. Polarity is set by the configuration parameter IC_INTR_POL (checked = active high). When IC_INTR_POL is unchecked (0), the <i>ic_gen_call_intr_n</i> signal is included on the interface instead to indicate active low polarity.</p> <p>Synchronous to: pclk</p> <p>Registered: Yes</p> <p>Default Output Delay: 30%</p>
DMA Interface (only present when configured with DMA interface) refer to “DMA Controller Interface” on page 65			
<i>dma_tx_req</i>	1 bit	Out	<p><i>Optional.</i> Transmit FIFO DMA Request. Asserted when the transmit FIFO requires service from the DMA Controller; that is, the transmit FIFO is at or below the watermark level.</p> <p>0 – not requesting 1 – requesting</p> <p>Software must set up the DMA controller with the number of words to be transferred when a request is made. When using the DW_ahb_dmac, this value is programmed in the SRC_MSIZ field of the CTLx register.</p> <p>Active State: High</p> <p>Synchronous to: pclk</p> <p>Registered: Yes</p> <p>Default Output Delay: 10%</p>

Table 10: DW_apb_i2c Signal Description (Continued)

Name	Width	I/O	Description
<i>dma_rx_req</i>	1 bit	Out	<p><i>Optional.</i> Receive FIFO DMA Request. Asserted when the receive FIFO requires service from the DMA Controller; that is, the receive FIFO is at or above the watermark level.</p> <p>0 – not requesting 1 – requesting</p> <p>Software must set up the DMA controller with the number of words to be transferred when a request is made. When using the DW_ahb_dmac, this value is programmed in the DEST_MSIZE field of the CTLx register.</p> <p>Active State: High Synchronous to: pclk Registered: Yes Default Output Delay: 10%</p>
<i>dma_tx_single</i>	1 bit	Out	<p><i>Optional.</i> DMA Transmit FIFO Single Signal. This DMA status output informs the DMA Controller that there is at least one free entry in the transmit FIFO. This output does not request a DMA transfer.</p> <p>0: Transmit FIFO is full 1: Transmit FIFO is not full</p> <p>Active State: High Synchronous to: pclk Registered: Yes Default Output Delay: 10%</p>
<i>dma_rx_single</i>	1 bit	Out	<p><i>Optional.</i> DMA Receive FIFO Single Signal. This DMA status output informs the DMA Controller that there is at least one valid data entry in the receive FIFO. This output does not request a DMA transfer.</p> <p>0: Receive FIFO is empty 1: Receive FIFO is not empty</p> <p>Active State: High Synchronous to: pclk Registered: Yes Default Output Delay: 10%</p>
<i>dma_tx_ack</i>	1 bit	In	<p><i>Optional.</i> DMA Transmit Acknowledgement. Sent by the DMA Controller to acknowledge the end of each APB transfer burst to the transmit FIFO.</p> <p>Active State: High Synchronous to: pclk Registered: Yes Default Input Delay: 50%</p>
<i>dma_rx_ack</i>	1 bit	In	<p><i>Optional.</i> DMA Receive Acknowledgement. Sent by the DMA controller to acknowledge the end of each APB transfer burst from the receive FIFO.</p> <p>Active State: High Synchronous to: pclk Registered: Yes Default Input Delay: 50%</p>

Table 10: DW_apb_i2c Signal Description (Continued)

Name	Width	I/O	Description
I²C Debug			
debug_s_gen	1 bit	Out	In the master mode of operation, this signal is set to 1 when DW_apb_i2c is driving a START condition on the bus. Active State: Low Synchronous to: N/A Registered: N/A Default Output Delay: N/A
debug_p_gen	1 bit	Out	In the master mode of operation, this signal is set to 1 when DW_apb_i2c is driving a STOP condition on the bus. Active State: Low Synchronous to: N/A Registered: N/A Default Output Delay: N/A
debug_data	1 bit	Out	In the master or slave mode of operation, this signal is set to 1 when a byte of data is actively being read or written by DW_apb_i2c. This bit remains 1 until the transaction has completed. Active State: High Synchronous to: N/A Registered: N/A Default Output Delay: N/A
debug_addr	1 bit	Out	In the master or slave mode of operation, this signal is set to 1 when the addressing phase is active on the I ² C bus. Active State: High Synchronous to: N/A Registered: N/A Default Output Delay: N/A
debug_addr_10bit	1 bit	Out	In the master or slave mode of operation, this signal is set to 1 after a 10-bit address code has been detected. Active State: High Synchronous to: N/A Registered: N/A Default Output Delay: N/A
debug_rd	1 bit	Out	In the master mode of operation, this signal is set to 1 whenever the master is receiving data. This bit remains 1 until the transfer is complete or until the direction changes. Active State: High Synchronous to: N/A Registered: N/A Default Output Delay: N/A

Table 10: DW_apb_i2c Signal Description (Continued)

Name	Width	I/O	Description
debug_wr	1 bit	Out	<p>In the master mode of operation, this signal is set to 1 whenever the master is transmitting data. This bit remains 1 until the transfer is complete or the direction changes.</p> <p>Active State: High Synchronous to: N/A Registered: N/A Default Output Delay: N/A</p>
debug_hs	1 bit	Out	<p>In the master mode of operation, this signal is set to 1 when DW_apb_i2c is performing high-speed mode transfers. This bit is set after the high-speed master code is transmitted and remains 1 until the master leaves high-speed mode.</p> <p>Active State: High Synchronous to: N/A Registered: N/A Default Output Delay: N/A</p>
debug_master_act	1 bit	Out	<p>This bit is set to 1 when the master module is active.</p> <p>Active State: High Synchronous to: N/A Registered: N/A Default Output Delay: N/A</p>
debug_slave_act	1 bit	Out	<p>This bit is set to 1 when the slave module is active.</p> <p>Active State: High Synchronous to: N/A Registered: N/A Default Output Delay: N/A</p>
debug_mst_cstate	5 bits	Out	<p>Master FSM state vector.</p> <p>Active State: N/A Synchronous to: N/A Registered: N/A Default Output Delay: N/A</p>
debug_slv_cstate	3 bit	Out	<p>Slave FSM state vector.</p> <p>Active State: N/A Synchronous to: N/A Registered: N/A Default Output Delay: N/A</p>

6

Registers

This section describes the programmable registers of the DW_apb_i2c and contains the following sections:

- [“Register Memory Map” on page 100](#)
- [“Registers and Field Descriptions” on page 104](#)



Note

There are references to both hardware parameters and software registers throughout this chapter. Parameters and many of the register bits are prefixed with an *IC_**. However, the software register bits are distinguished in this chapter by italics. For instance, *IC_MAX_SPEED_MODE* is a hardware parameter and configured once using Synopsys coreConsultant, whereas the *IC_SLAVE_DISABLE* bit in the *IC_CON* register controls whether I2C has its slave disabled.

Register Memory Map



Note

A read operation to an address location that contains unused bits results in a 0 value being returned on each of the unused bits.

Shipped with the DW_apb_i2c component is an address definition (memory map) C header file. This can be used when the DW_apb_i2c is programmed in a C environment. [Table 11](#) provides the details of the DW_apb_i2c memory map. Reset values are affected by the configuration parameters specified in [Table 8 on page 76](#).

Table 11: Memory Map of DW_apb_i2c

Name	Address Offset	Width	R/W	Description
IC_CON	0x00	7 bits	R/W or R-only on bit 4	I ² C Control R/W: If configuration parameter I2C_DYNAMIC_TAR_UPDATE is 0, all bits are Read/Write. If I2C_DYNAMIC_TAR_UPDATE is 1, bit 4 is Read-only. Reset Value: Reset values for the 6 bit fields correspond to the following configuration parameters: 6: IC_SLAVE_DISABLE 5: IC_RESTART_EN 4: IC_10BITADDR_MASTER 3: IC_10BITADDR_SLAVE 2:1:IC_MAX_SPEED_MODE 0: IC_MASTER_MODE
IC_TAR	0x04	12 or 13 bits	R/W	I ² C Target Address Width: 13, if I2C_DYNAMIC_TAR_UPDATE = 1 12, if I2C_DYNAMIC_TAR_UPDATE = 0 Reset Value: Reset values for the four bit fields correspond to the following: 12: IC_10BITADDR_MASTER configuration parameter 11: 0x0 10: 0x0 9:0: IC_DEFAULT_TAR_SLAVE_ADDR
IC_SAR	0x08	10 bits	R/W	I ² C Slave Address Reset Value: IC_DEFAULT_SLAVE_ADDR
IC_HS_MADDR	0x0C	3 bits	R/W	I ² C HS Master Mode Code Address Reset Value: IC_HS_MASTER_CODE

Table 11: Memory Map of DW_apb_i2c (Continued)

Name	Address Offset	Width	R/W	Description
IC_DATA_CMD	0x10	9 (writes) 8 (reads)	R/W	I ² C Rx/Tx Data Buffer and Command Reset Value: 0x0 NOTE: With nine bits required for writes, the DW_apb_i2c requires 16-bit data on the APB bus transfers when writing into the transmit FIFO. Eight-bit transfers remain for reads from the receive FIFO.
IC_SS_SCL_HCNT	0x14	16 bits	R/W	Standard speed I ² C Clock SCL High Count Reset Value: IC_SS_SCL_HIGH_COUNT
IC_SS_SCL_LCNT	0x18	16 bits	R/W	Standard speed I ² C Clock SCL Low Count Reset Value: IC_SS_SCL_LOW_COUNT
IC_FS_SCL_HCNT	0x1C	16 bits	R/W	Fast speed I ² C Clock SCL High Count Reset Value: IC_FS_SCL_HIGH_COUNT
IC_FS_SCL_LCNT	0x20	16 bits	R/W	Fast speed I ² C Clock SCL Low Count Reset Value: IC_FS_SCL_LOW_COUNT
IC_HS_SCL_HCNT	0x24	16 bits	R/W	High speed I ² C Clock SCL High Count Reset Value: IC_HS_SCL_HIGH_COUNT
IC_HS_SCL_LCNT	0x28	16 bits	R/W	High speed I ² C Clock SCL Low Count Reset Value: IC_HS_SCL_LOW_COUNT
IC_INTR_STAT	0x2C	12 bits	R	I ² C Interrupt Status Reset Value: 0x0
IC_INTR_MASK	0x30	12 bits	R/W	I ² C Interrupt Mask Reset Value: 12'h8ff
IC_RAW_INTR_STAT	0x34	12 bits	R	I ² C Raw Interrupt Status Reset Value: 0x0
IC_RX_TL	0x38	8 bits	R/W	I ² C Receive FIFO Threshold Reset Value: IC_RX_TL configuration parameter
IC_TX_TL	0x3C	8 bits	R/W	I ² C Transmit FIFO Threshold Reset Value: IC_TX_TL configuration parameter
IC_CLR_INTR	0x40	1 bit	R	Clear Combined and Individual Interrupts Reset Value: 0x0
IC_CLR_RX_UNDER	0x44	1 bit	R	Clear RX_UNDER Interrupt Reset Value: 0x0
IC_CLR_RX_OVER	0x48	1 bit	R	Clear RX_OVER Interrupt Reset Value: 0x0

Table 11: Memory Map of DW_apb_i2c (Continued)

Name	Address Offset	Width	R/W	Description
IC_CLR_TX_OVER	0x4C	1 bit	R	Clear TX_OVER Interrupt Reset Value: 0x0
IC_CLR_RD_REQ	0x50	1 bit	R	Clear RD_REQ Interrupt Reset Value: 0x0
IC_CLR_TX_ABRT	0x54	1 bit	R	Clear TX_ABRT Interrupt Reset Value: 0x0
IC_CLR_RX_DONE	0x58	1 bit	R	Clear RX_DONE Interrupt Reset Value: 0x0
IC_CLR_ACTIVITY	0x5c	1 bit	R	Clear ACTIVITY Interrupt Reset Value: 0x0
IC_CLR_STOP_DET	0x60	1 bit	R	Clear STOP_DET Interrupt Reset Value: 0x0
IC_CLR_START_DET	0x64	1 bit	R	Clear START_DET Interrupt Reset Value: 0x0
IC_CLR_GEN_CALL	0x68	1 bit	R	Clear GEN_CALL Interrupt Reset Value: 0x0
IC_ENABLE	0x6C	1 bit	R/W	I ² C Enable Reset Value: 0x0
IC_STATUS	0x70	7 bits	R	I ² C Status register Reset Value: 0x6
IC_TXFLR	0x74	TX_ABW +1	R	Transmit FIFO Level Register Reset Value: 0x0
IC_RXFLR	0x78	RX_ABW +1	R	Receive FIFO Level Register Reset Value: 0x0
Reserved	0x7C			
IC_TX_ABRT_SOURCE	0x80	16 bits	R/W	I ² C Transmit Abort Status Register Reset Value: 0x0
IC_SLV_DATA_NACK_ONLY	0x84	1 bit	R/W	Generate SLV_DATA_NACK Register Reset Value: 0x0
IC_DMA_CR	0x88	2 bits	R/W	DMA Control Register for transmit and receive handshaking interface Reset Value: 0x0
IC_DMA_TDLR	0x8c	TX_ABW	R/W	DMA Transmit Data Level Reset Value: 0x0
IC_DMA_RDLR	0x90	RX_ABW	R/W	DMA Receive Data Level Reset Value: 0x0

Table 11: Memory Map of DW_apb_i2c (Continued)

Name	Address Offset	Width	R/W	Description
IC_SDA_SETUP	0x94	8 bits	R/W	I ² C SDA Setup Register Reset Value: IC_DEFAULT_SDA_SETUP configuration parameter
IC_ACK_GENERAL_CALL	0x98	1 bit	R/W	I ² C ACK General Call Register Reset Value: IC_DEFAULT_ACK_GENERAL_CALL configuration parameter
IC_ENABLE_STATUS	0x9C	3 bits	R	I ² C Enable Status Register Reset Value: 0x0
IC_COMP_PARAM_1	0xf4	32 bits	R	Component Parameter Register Reset Value: Reset value depends on configuration parameters. For more information on component parameters and the values therefore set by them, refer to Table 8 on page 76 .
IC_COMP_VERSION	0xf8	32 bits	R	Component Version ID Reset Value: See the releases table in the DW_apb_i2c Release Notes
IC_COMP_TYPE	0xfc	32 bits	R	DesignWare Component Type Register Reset Value: 0x44570140

Registers and Field Descriptions

This section describes the registers listed in [Table 11 on page 100](#). Registers are on the pclk domain, but status bits reflect actions that occur in the ic_clk domain. Therefore, there is delay when the pclk register reflects the activity that occurred on the ic_clk side.

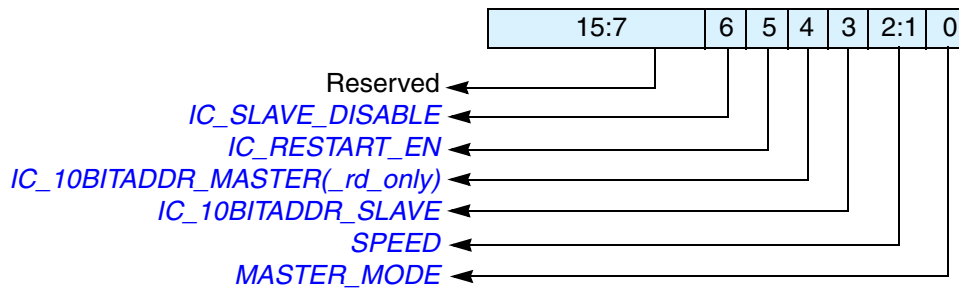
Some registers may be written only when the DW_apb_i2c is disabled, programmed by the [IC_ENABLE](#) register. Software should not disable the DW_apb_i2c while it is active. If the DW_apb_i2c is in the process of transmitting when it is disabled, it stops as well as deletes the contents of the transmit buffer after the current transfer is complete. The slave continues receiving until the remote master aborts the transfer, in which case the DW_apb_i2c could be disabled. Registers that cannot be written to when the DW_apb_i2c is enabled are indicated in their descriptions.

Unless the clocks pclk and ic_clk are identical (IC_CLK_TYPE = 0), there is a two-register delay for synchronous and asynchronous modes.


IC_CON

- **Name:** I²C Control Register
- **Size:** 7 bits
- **Address Offset:** 0x00
- **Read/Write Access:**
 - If configuration parameter I2C_DYNAMIC_TAR_UPDATE = 0, all bits are Read/Write.
 - If I2C_DYNAMIC_TAR_UPDATE = 1, bit 4 is Read-only.

This register can be written only when the DW_apb_i2c is disabled, which corresponds to the IC_ENABLE register being set to 0. Writes at other times have no effect.



Bits	Name	R/W	Description
15:7	Reserved	N/A	Reserved.
6	<i>IC_SLAVE_DISABLE</i>	R/W	<p>This bit controls whether I²C has its slave disabled, which means once the preseln signal is applied, then this bit takes on the value of the configuration parameter <i>IC_SLAVE_DISABLE</i>. You have the choice of having the slave enabled or disabled after reset is applied, which means software does not have to configure the slave. By default, the slave is always enabled (in reset state as well). If you need to disable it after reset, set this bit to 1.</p> <p>If this bit is set (slave is disabled), <i>DW_apb_i2c</i> functions only as a master and does not perform any action that requires a slave.</p> <p>0: slave is enabled 1: slave is disabled</p> <p>Reset value: <i>IC_SLAVE_DISABLE</i> configuration parameter</p> <p>NOTE: Software should ensure that if this bit is written with '0,' then bit 0 should also be written with a '0'.</p>
5	<i>IC_RESTART_EN</i>	R/W	<p>Determines whether RESTART conditions may be sent when acting as a master. Some older slaves do not support handling RESTART conditions; however, RESTART conditions are used in several <i>DW_apb_i2c</i> operations.</p> <p>0: disable 1: enable</p> <p>When RESTART is disabled, the master is prohibited from performing the following functions:</p> <ul style="list-style-type: none"> ● Change direction within a transfer (split) ● Send a START BYTE ● High-speed mode operation ● Combined format transfers in 7-bit addressing modes ● Read operation with a 10-bit address ● Send multiple bytes per transfer <p>By replacing RESTART condition followed by a STOP and a subsequent START condition, split operations are broken down into multiple <i>DW_apb_i2c</i> transfers. If the above operations are performed, it will result in setting bit 6 (<i>TX_ABORT</i>) of the IC_RAW_INTR_STAT register.</p> <p>Reset value: <i>IC_RESTART_EN</i> configuration parameter</p>

Bits	Name	R/W	Description
4	<i>IC_10BITADDR_MASTER</i> or <i>IC_10BITADDR_MASTER_rd_only</i>	R/W or R	<p>If the <i>I2C_DYNAMIC_TAR_UPDATE</i> configuration parameter is set to “No” (0), this bit is named <i>IC_10BITADDR_MASTER</i> and controls whether the DW_apb_i2c starts its transfers in 7- or 10-bit addressing mode when acting as a master.</p> <p>If <i>I2C_DYNAMIC_TAR_UPDATE</i> is set to “Yes” (1), the function of this bit is handled by bit 12 of <i>IC_TAR</i> register, and becomes a read-only copy called <i>IC_10BITADDR_MASTER_rd_only</i>.</p> <p>0: 7-bit addressing 1: 10-bit addressing</p> <p>Dependencies: If <i>I2C_DYNAMIC_TAR_UPDATE</i> = 1, then this bit is read-only. If <i>I2C_DYNAMIC_TAR_UPDATE</i> = 0, then this bit can be read or write.</p> <p>Reset value: <i>IC_10BITADDR_MASTER</i> configuration parameter</p>
3	<i>IC_10BITADDR_SLAVE</i>	R/W	<p>When acting as a slave, this bit controls whether the DW_apb_i2c responds to 7- or 10-bit addresses.</p> <p>0: 7-bit addressing. The DW_apb_i2c ignores transactions that involve 10-bit addressing; for 7-bit addressing, only the lower 7 bits of the <i>IC_SAR</i> register are compared.</p> <p>1: 10-bit addressing. The DW_apb_i2c responds to only 10-bit addressing transfers that match the full 10 bits of the <i>IC_SAR</i> register.</p> <p>Reset value: <i>IC_10BITADDR_SLAVE</i> configuration parameter</p>
<p> Note —————</p> <p>Bits 3 and 4 of this register can be programmed differently and in any combination depending on which format is required for the transfers. For example, master mode can be configured with 10-bit addressing and slave mode can be configured with 7-bit addressing.</p>			
2:1	<i>SPEED</i>	R/W	<p>These bits control at which speed the DW_apb_i2c operates; its setting is relevant only if one is operating the DW_apb_i2c in master mode. Hardware protects against illegal values being programmed by software. This register should be programmed only with a value in the range of 1 to <i>IC_MAX_SPEED_MODE</i>; otherwise, hardware updates this register with the value of <i>IC_MAX_SPEED_MODE</i>.</p> <p>1: standard mode (100 kbit/s) 2: fast mode (400 kbit/s) 3: high speed mode (3.4 Mbit/s)</p> <p>Reset value: <i>IC_MAX_SPEED_MODE</i> configuration</p>
0	<i>MASTER_MODE</i>	R/W	<p>This bit controls whether the DW_apb_i2c master is enabled.</p> <p>0: master disabled 1: master enabled</p> <p>Reset value: <i>IC_MASTER_MODE</i> configuration parameter</p> <p>NOTE: Software should ensure that if this bit is written with ‘1,’ then bit 6 should also be written with a ‘1’.</p>

**Note**

Because the DW_apb_i2c should only be used either as an I²C master or I²C slave (but not both) at any one time, care should be taken in software that certain combinations of the two bits IC_SLAVE_DISABLE and IC_MASTER_MODE are not programmed into the “IC_CON” on [page 104](#) register. In particular, IC_SLAVE_DISABLE and IC_MASTER_MODE must not be set to ‘0’ and ‘1,’ respectively at any given time.

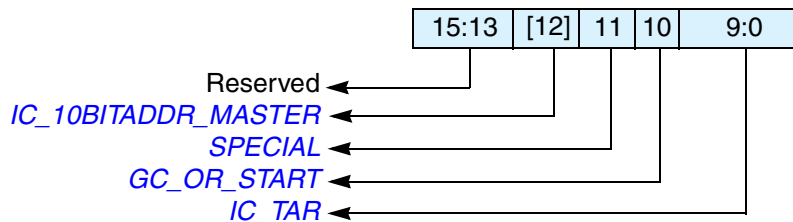
IC_TAR

- **Name:** I²C Target Address Register
- **Size:** 12 bits or 13 bits; 13 bits only when I2C_DYNAMIC_TAR_UPDATE = 1
- **Address Offset:** 0x04
- **Read/Write Access:** Read/Write

If the configuration parameter I2C_DYNAMIC_TAR_UPDATE is set to “No” (0), this register is 12 bits wide, and bits 15:12 are reserved. This register can be written to only when IC_ENABLE is set to 0.

However, if I2C_DYNAMIC_TAR_UPDATE = 1, then the register becomes 13 bits wide. All bits can be dynamically updated as long as any set of the following conditions are true:

- DW_apb_i2c is NOT enabled (IC_ENABLE is set to 0); or
- DW_apb_i2c is enabled (IC_ENABLE=1); AND
 DW_apb_i2c is NOT engaged in any Master (tx, rx) operation (IC_STATUS[5]=0); AND
 DW_apb_i2c is enabled to operate in Master mode (IC_CON[0]=1); AND
 there are NO entries in the TX FIFO (IC_STATUS[2]=1)



Bits	Name	R/W	Description
15:13	Reserved	N/A	Reserved.
12	IC_10BITADDR_MASTER	R/W	This bit controls whether the DW_apb_i2c starts its transfers in 7- or 10-bit addressing mode when acting as a master. 0: 7-bit addressing 1: 10-bit addressing Dependencies: This bit exists in this register only if the I2C_DYNAMIC_TAR_UPDATE configuration parameter is set to “Yes” (1). Reset value: IC_10BITADDR_MASTER configuration parameter
11	SPECIAL	R/W	This bit indicates whether software performs a General Call or START BYTE command. 0: ignore bit 10 GC_OR_START and use IC_TAR normally 1: perform special I ² C command as specified in GC_OR_START bit Reset value: 0x0

Bits	Name	R/W	Description
10	<i>GC_OR_START</i>	R/W	<p>If bit 11 (<i>SPECIAL</i>) is set to 1, then this bit indicates whether a General Call or START byte command is to be performed by the DW_apb_i2c.</p> <p>0: General Call Address – after issuing a General Call, only writes may be performed. Attempting to issue a read command results in setting bit 6 (TX_ABRT) of the <i>IC_RAW_INTR_STAT</i> register. The DW_apb_i2c remains in General Call mode until the <i>SPECIAL</i> bit value (bit 11) is cleared.</p> <p>1: START BYTE</p> <p>Reset value: 0x0</p>
9:0	<i>IC_TAR</i>	R/W	<p>This is the target address for any master transaction. When transmitting a General Call, these bits are ignored. To generate a START BYTE, the CPU needs to write only once into these bits.</p> <p>Reset value: IC_DEFAULT_TAR_SLAVE_ADDR configuration parameter</p> <p>If the <i>IC_TAR</i> and <i>IC_SAR</i> are the same, loopback exists but the FIFOs are shared between master and slave, so full loopback is not feasible. Only one direction loopback mode is supported (simplex), not duplex. A master cannot transmit to itself; it can transmit to only a slave.</p>

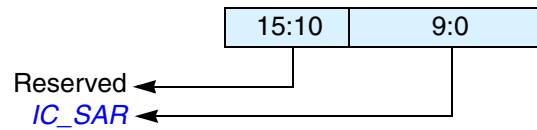



Note

It is not necessary to perform any write to this register if DW_apb_i2c is enabled as an I²C slave only.

IC_SAR

- **Name:** I²C Slave Address Register
- **Size:** 10 bits
- **Address Offset:** 0x08
- **Read/Write Access:** Read/Write



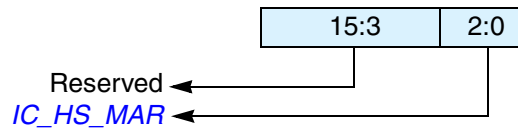
Bits	Name	R/W	Description
15:10	Reserved	N/A	Reserved.
9:0	<i>IC_SAR</i>	R/W	<p>The <i>IC_SAR</i> holds the slave address when the I²C is operating as a slave. For 7-bit addressing, only <i>IC_SAR</i>[6:0] is used.</p> <p>This register can be written only when the I²C interface is disabled, which corresponds to the <i>IC_ENABLE</i> register being set to 0. Writes at other times have no effect.</p> <p> Note — The default values cannot be any of the reserved address locations: that is, 0x00 to 0x07, or 0x78 to 0x7f. The correct operation of the device is not guaranteed if you program the <i>IC_SAR</i> or <i>IC_TAR</i> to a reserved value. Refer to Table 7 on page 51 for a complete list of these reserved values.</p> <p>Reset value: <i>IC_DEFAULT_SLAVE_ADDR</i> configuration parameter</p>

**Note**

It is not necessary to perform any write to this register if DW_apb_i2c is enabled as an I²C master only.

IC_HS_MADDR

- **Name:** I²C High Speed Master Mode Code Address Register
- **Size:** 3 bits
- **Address Offset:** 0x0c
- **Read/Write Access:** Read/Write



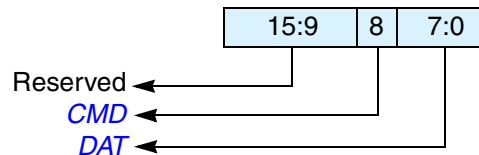
Bits	Name	R/W	Description
15:3	Reserved	N/A	Reserved.
2:0	<i>IC_HS_MAR</i>	R/W	<p>This bit field holds the value of the I²C HS mode master code. HS-mode master codes are reserved 8-bit codes (00001xxx) that are not used for slave addressing or other purposes. Each master has its unique master code; up to eight high-speed mode masters can be present on the same I²C bus system. Valid values are from 0 to 7. This register goes away and becomes read-only returning 0's if the <i>IC_MAX_SPEED_MODE</i> configuration parameter is set to either Standard (1) or Fast (2).</p> <p>This register can be written only when the I²C interface is disabled, which corresponds to the <i>IC_ENABLE</i> register being set to 0. Writes at other times have no effect.</p> <p>Reset value: <i>IC_HS_MASTER_CODE</i> configuration parameter</p>

**Note**

It is not necessary to perform any write to this register if DW_apb_i2c is enabled as an I²C slave only.

IC_DATA_CMD

- **Name:** I²C Rx/Tx Data Buffer and Command Register; this is the register the CPU writes to when filling the TX FIFO and the CPU reads from when retrieving bytes from RX FIFO
- **Size:** 9 bits (writes)
8 bits (reads)
- **Address Offset:** 0x10
- **Read/Write Access:** Read/Write



Bits	Name	R/W	Description
15:9	Reserved	N/A	Reserved
8	<i>CMD</i>	R/W	<p>This bit controls whether a read or a write is performed. This bit does not control the direction when the DW_apb_i2c acts as a slave. It controls only the direction when it acts as a master.</p> <p>1 = Read 0 = Write</p> <p>When a command is entered in the TX FIFO, this bit distinguishes the write and read commands. In slave-receiver mode, this bit is a “don’t care” because writes to this register are not required. In slave-transmitter mode, a “0” indicates that CPU data is to be transmitted and as DAT or IC_DATA_CMD[7:0].</p> <p>When programming this bit, you should remember the following: attempting to perform a read operation after a General Call command has been sent results in a TX_ABORT interrupt (bit 6 of the IC_RAW_INTR_STAT register), unless bit 11 (SPECIAL) in the IC_TAR register has been cleared.</p> <p>If a “1” is written to this bit after receiving a RD_REQ interrupt, then a TX_ABORT interrupt occurs.</p> <p>NOTE: It is possible that while attempting a master I²C read transfer on DW_apb_i2c, a RD_REQ interrupt may have occurred simultaneously due to a remote I²C master addressing DW_apb_i2c. In this type of scenario, DW_apb_i2c ignores the IC_DATA_CMD write, generates a TX_ABORT interrupt, and waits to service the RD_REQ interrupt. For more details, see “Operation Modes” on page 56.</p> <p>Reset value: 0x0</p>
7:0	<i>DAT</i>	R/W	<p>This register contains the data to be transmitted or received on the I²C bus. If you are writing to this register and want to perform a read, bits 7:0 (DAT) are ignored by the DW_apb_i2c. However, when you read this register, these bits return the value of data received on the DW_apb_i2c interface.</p> <p>Reset value: 0x0</p>

IC_SS_SCL_HCNT

- **Name:** Standard Speed I²C Clock SCL High Count Register
- **Size:** 16 bits
- **Address Offset:** 0x14
- **Read/Write Access:** Read/Write



Bits	Name	R/W	Description
15:0	<i>IC_SS_SCL_HCNT</i>	R/W ¹	<p>This register must be set before any I²C bus transaction can take place to ensure proper I/O timing. This register sets the SCL clock high-period count for standard speed. The table below shows some sample <i>IC_SS_SCL_HCNT</i> calculations. These values apply only if the <i>ic_clk</i> is set to the given frequency in the table.</p> <p>This register can be written only when the I²C interface is disabled which corresponds to the <i>IC_ENABLE</i> register being set to 0. Writes at other times have no effect.</p> <p>The minimum valid value is 6; hardware prevents values less than this being written, and if attempted results in 6 being set. For designs with <i>APB_DATA_WIDTH</i> = 8, the order of programming is important to ensure the correct operation of the DW_apb_i2c. The lower byte must be programmed first. Then the upper byte is programmed.</p> <p>When the configuration parameter <i>IC_HC_COUNT_VALUES</i> is set to 1, this register is read only.</p> <p>NOTE: This register must not be programmed to a value higher than 65525, because DW_apb_i2c uses a 16-bit counter to flag an I²C bus idle condition when this counter reaches a value of <i>IC_SS_SCL_HCNT</i> + 10.</p> <p>Reset value: <i>IC_SS_SCL_HIGH_COUNT</i> configuration parameter</p>
¹ Read-only if <i>IC_HC_COUNT_VALUES</i> = 1.			

 **Note**

The following table contains minimum values; combining both Low and High settings does not result in the correct baud rate, but rather a higher baud rate. The user should increase the value programmed into the *IC_SS_SCL_HCNT* register so that the correct, compliant I²C speed is achieved.

I ² C Data Rate (kbps)	<i>ic_clk</i> _{freq} (MHz)	SCL High required min (μs)	Minimum H_CNT	Actual SCL High Time (μs)
100	3	4	6	4.67
100	5	4	12	4.00
100	10	4	32	4.00
100	15	4	52	4.00
100	20	4	72	4.00

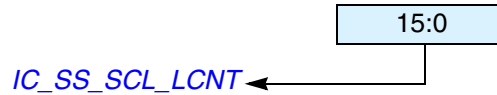
I ² C Data Rate (kbps)	ic_clk _{freq} (MHz)	SCL High required min (μs)	Minimum H_CNT	Actual SCL High Time (μs)
100	50	4	192	4.00
100	100	4	392	4.00

**Note**

It is not necessary to perform any write to this register if DW_apb_i2c is enabled as an I²C slave only.

IC_SS_SCL_LCNT

- **Name:** Standard Speed I²C Clock SCL Low Count Register
- **Size:** 16 bits
- **Address Offset:** 0x18
- **Read/Write Access:** Read/Write



Bits	Name	R/W	Description
15:0	<i>IC_SS_SCL_LCNT</i>	R/W ¹	<p>This register must be set before any I²C bus transaction can take place to ensure proper I/O timing. This register sets the SCL clock low period count for standard speed. The table below shows some sample <i>IC_SS_SCL_LCNT</i> calculations. These values apply only if the <i>ic_clk</i> is set to the given frequency in the table.</p> <p>This register can be written only when the I²C interface is disabled which corresponds to the <i>IC_ENABLE</i> register being set to 0. Writes at other times have no effect.</p> <p>The minimum valid value is 8; hardware prevents values less than this being written, and if attempted, results in 8 being set. For designs with <i>APB_DATA_WIDTH</i> = 8, the order of programming is important to ensure the correct operation of DW_apb_i2c. The lower byte must be programmed first, and then the upper byte is programmed.</p> <p>When the configuration parameter <i>IC_HC_COUNT_VALUES</i> is set to 1, this register is read only.</p> <p>Reset value: <i>IC_SS_SCL_LOW_COUNT</i> configuration parameter</p>
¹ Read-only if <i>IC_HC_COUNT_VALUES</i> = 1.			

**Note**

The following table contains minimum values; combining both Low and High settings does not result in the correct baud rate, but rather a higher baud rate. You should increase the value programmed into the *IC_SS_SCL_LCNT* register so that the correct required I²C speed is achieved.

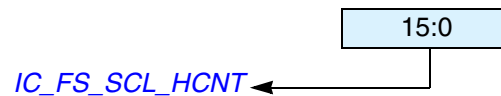
I ² C Data Rate (kbps)	<i>ic_clk_{freq}</i> (MHz)	SCL Low required min (μs)	Minimum L_CNT	Actual SCL Low Time (μs)
100	3	4.7	14	5.00
100	5	4.7	23	4.80
100	10	4.7	46	4.70
100	15	4.7	70	4.73
100	20	4.7	93	4.70
100	50	4.7	234	4.70
100	100	4.7	469	4.70

 **Note**

It is not necessary to perform any write to this register if DW_apb_i2c is enabled as an I²C slave only.

IC_FS_SCL_HCNT

- **Name:** Fast Speed I²C Clock SCL High Count Register
- **Size:** 16 bits
- **Address Offset:** 0x1c
- **Read/Write Access:** Read/Write



Bits	Name	R/W	Description
15:0	<i>IC_FS_SCL_HCNT</i>	R/W ¹	<p>This register must be set before any I²C bus transaction can take place to ensure proper I/O timing. This register sets the SCL clock high-period count for fast speed. It is used in high-speed mode to send the Master Code and START BYTE or General CALL. The table below shows some sample <i>IC_FS_SCL_HCNT</i> calculations. These values apply only if the <i>ic_clk</i> is set to the given frequency in the table.</p> <p>This register goes away and becomes read-only returning 0s if <i>IC_MAX_SPEED_MODE</i> = standard. This register can be written only when the I²C interface is disabled, which corresponds to the <i>IC_ENABLE</i> register being set to 0. Writes at other times have no effect.</p> <p>The minimum valid value is 6; hardware prevents values less than this being written, and if attempted results in 6 being set. For designs with <i>APB_DATA_WIDTH</i> == 8 the order of programming is important to ensure the correct operation of the DW_apb_i2c. The lower byte must be programmed first. Then the upper byte is programmed.</p> <p>When the configuration parameter <i>IC_HC_COUNT_VALUES</i> is set to 1, this register is read only.</p> <p>Reset value: <i>IC_FS_SCL_HIGH_COUNT</i> configuration parameter</p>

¹ Read-only if *IC_HC_COUNT_VALUES* = 1.

 **Note**

The following table contains minimum values; combining both Low and High settings does not result in the correct baud rate, but rather a higher baud rate. The user should increase the value programmed into the *IC_FS_SCL_HCNT* register so that the correct required I²C speed is achieved.

I ² C Data Rate (kbps)	<i>ic_clk</i> _{freq} (MHz)	SCL High required min (μs)	Minimum H_CNT	Actual SCL High Time (μs)
400	15	0.6	6	0.93
400	20	0.6	6	0.70

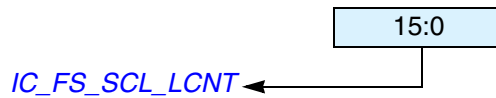
I ² C Data Rate (kbps)	ic_clk _{freq} (MHz)	SCL High required min (μs)	Minimum H_CNT	Actual SCL High Time (μs)
400	50	0.6	22	0.60
400	100	0.6	52	0.60
400	150	0.6	82	0.60

**Note**

It is not necessary to perform any write to this register if DW_apb_i2c is enabled as an I²C slave only.

IC_FS_SCL_LCNT

- **Name:** Fast Speed I²C Clock SCL Low Count Register
- **Size:** 16 bits
- **Address Offset:** 0x20
- **Read/Write Access:** Read/Write



Bits	Name	R/W	Description
15:0	<i>IC_FS_SCL_LCNT</i>	R/W ¹	<p>This register must be set before any I²C bus transaction can take place to ensure proper I/O timing. This register sets the SCL clock low period count for fast speed. It is used in high-speed mode to send the Master Code and START BYTE or General CALL. The table below shows some sample <i>IC_FS_SCL_LCNT</i> calculations. These values apply only if the <i>ic_clk</i> is set to the given frequency in the table.</p> <p>This register goes away and becomes read-only returning 0s if <i>IC_MAX_SPEED_MODE</i> = standard.</p> <p>This register can be written only when the I²C interface is disabled, which corresponds to the <i>IC_ENABLE</i> register being set to 0. Writes at other times have no effect.</p> <p>The minimum valid value is 8; hardware prevents values less than this being written, and if attempted results in 8 being set. For designs with <i>APB_DATA_WIDTH</i> = 8 the order of programming is important to ensure the correct operation of the DW_apb_i2c. The lower byte must be programmed first. Then the upper byte is programmed. If the value is less than 8 then the count value gets changed to 8.</p> <p>When the configuration parameter <i>IC_HC_COUNT_VALUES</i> is set to 1, this register is read only.</p> <p>Reset value: <i>IC_FS_SCL_LOW_COUNT</i> configuration parameter</p>
<p>¹ Read-only if <i>IC_HC_COUNT_VALUES</i> = 1.</p>			

Note

The following table contains minimum values; combining both Low and High settings does not result in the correct baud rate, but rather a higher baud rate. The user should increase the value programmed into the *IC_FS_SCL_LCNT* register so that the correct required I²C speed is achieved.

I ² C Data Rate (kbps)	<i>ic_clk</i> _{freq} (MHz)	SCL Low required min (μs)	Minimum L_CNT	Actual SCL Low Time (μs)
400	15	1.3	19	1.33
400	20	1.3	25	1.30
400	50	1.3	64	1.30

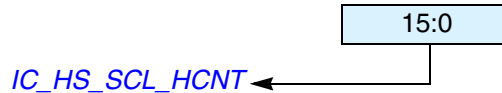
I ² C Data Rate (kbps)	ic_clk _{freq} (MHz)	SCL Low required min (μs)	Minimum L_CNT	Actual SCL Low Time (μs)
400	100	1.3	129	1.30
400	150	1.3	194	1.30

**Note**

It is not necessary to perform any write to this register if DW_apb_i2c is enabled as an I²C slave only.

IC_HS_SCL_HCNT

- **Name:** High Speed I²C Clock SCL High Count Register
- **Size:** 16 bits
- **Address Offset:** 0x24
- **Read/Write Access:** Read/Write



Bits	Name	R/W	Description
15:0	<i>IC_HS_SCL_HCNT</i>	R/W ¹	<p>This register must be set before any I²C bus transaction can take place to ensure proper I/O timing. This register sets the SCL clock high period count for high speed. The table below shows some sample <i>IC_HS_SCL_HCNT</i> calculations. These values apply only if the <i>ic_clk</i> is set to the given frequency in the table.</p> <p>The SCL High time depends on the loading of the bus. For 100pF loading, the SCL High time is 60ns; for 400pF loading, the SCL High time is 120ns.</p> <p>This register goes away and becomes read-only returning 0s if <i>IC_MAX_SPEED_MODE</i> != high.</p> <p>This register can be written only when the I²C interface is disabled, which corresponds to the <i>IC_ENABLE</i> register being set to 0. Writes at other times have no effect.</p> <p>The minimum valid value is 6; hardware prevents values less than this being written, and if attempted results in 6 being set. For designs with <i>APB_DATA_WIDTH</i> = 8 the order of programming is important to ensure the correct operation of the DW_apb_i2c. The lower byte must be programmed first. Then the upper byte is programmed.</p> <p>When the configuration parameter <i>IC_HC_COUNT_VALUES</i> is set to 1, this register is read only.</p> <p>Reset value: <i>IC_HS_SCL_HIGH_COUNT</i> configuration parameter</p>
<p>¹ Read-only if <i>IC_HC_COUNT_VALUES</i> = 1.</p>			



Note

The following table contains minimum values; combining both Low and High settings does not result in the correct baud rate, but rather a higher baud rate. The user should increase the value programmed into the *IC_HS_SCL_HCNT* register so that the correct required I²C speed is achieved.

I ² C Data Rate (kbps)	<i>ic_clk_{freq}</i> (MHz)	SCL High required min (μs)	Minimum H_CNT	Actual SCL High Time (μs)
3400	100	0.06	6	0.14
3400	120	0.06	6	0.12
3400	150	0.06	6	0.09
3400	200	0.06	6	0.07

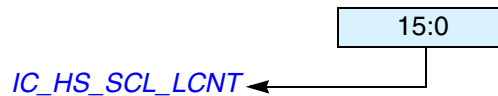
I ² C Data Rate (kbps)	ic_clk _{freq} (MHz)	SCL High required min (μs)	Minimum H_CNT	Actual SCL High Time (μs)
1700	100	0.12	6	0.14
1700	120	0.12	6	0.12
1700	150	0.12	10	0.12
1700	200	0.12	16	0.12

**Note**

It is not necessary to perform any write to this register if DW_apb_i2c is enabled as an I²C slave only.

IC_HS_SCL_LCNT

- **Name:** High Speed I²C Clock SCL Low Count Register
- **Size:** 16 bits
- **Address Offset:** 0x28
- **Read/Write Access:** Read/Write



Bits	Name	R/W	Description
15:0	<i>IC_HS_SCL_LCNT</i>	R/W ¹	<p>This register must be set before any I²C bus transaction can take place to ensure proper I/O timing. This register sets the SCL clock low period count for high speed. The table below shows some sample <i>IC_HS_SCL_LCNT</i> calculations. These values apply only if the <i>ic_clk</i> is set to the given frequency in the table.</p> <p>The SCL low time depends on the loading of the bus. For 100pF loading, the SCL low time is 160ns; for 400pF loading, the SCL low time is 320ns.</p> <p>This register goes away and becomes read-only returning 0s if <i>IC_MAX_SPEED_MODE</i> != high.</p> <p>This register can be written only when the I²C interface is disabled, which corresponds to the <i>IC_ENABLE</i> register being set to 0. Writes at other times have no effect.</p> <p>The minimum valid value is 8; hardware prevents values less than this being written, and if attempted results in 8 being set. For designs with <i>APB_DATA_WIDTH</i> == 8 the order of programming is important to ensure the correct operation of the DW_apb_i2c. The lower byte must be programmed first. Then the upper byte is programmed. If the value is less than 8 then the count value gets changed to 8.</p> <p>When the configuration parameter <i>IC_HC_COUNT_VALUES</i> is set to 1, this register is read only.</p> <p>Reset value: <i>IC_HS_SCL_LOW_COUNT</i> configuration parameter</p>
<p>¹ Read-only if <i>IC_HC_COUNT_VALUES</i> = 1.</p>			



Note

The following table contains minimum values; combining both Low and High settings does not result in the correct baud rate, but rather a higher baud rate. The user should increase the value programmed into the *IC_HS_SCL_LCNT* register so that the correct required I²C speed is achieved.

I ² C Data Rate (kbps)	<i>ic_clk</i> _{freq} (MHz)	SCL Low required min (μs)	Minimum L_CNT	Actual SCL Low Time (μs)
3400	100	0.16	15	0.16
3400	120	0.16	18	0.16
3400	150	0.16	23	0.16
3400	200	0.16	30	0.16

I ² C Data Rate (kbps)	ic_clk _{freq} (MHz)	SCL Low required min (μs)	Minimum L_CNT	Actual SCL Low Time (μs)
1700	100	0.32	31	0.32
1700	120	0.32	37	0.32
1700	150	0.32	47	0.32
1700	200	0.32	63	0.32

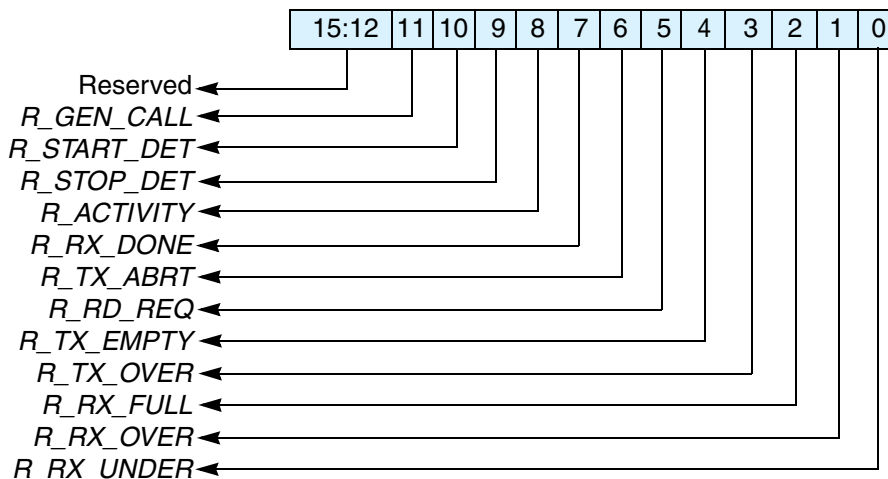
**Note**

It is not necessary to perform any write to this register if DW_apb_i2c is enabled as an I²C slave only.

IC_INTR_STAT

- **Name:** I²C Interrupt Status Register
- **Size:** 12 bits
- **Address Offset:** 0x2C
- **Read/Write Access:** Read

Each bit in this register has a corresponding mask bit in the [IC_INTR_MASK](#) register. These bits are cleared by reading the matching interrupt clear register. The unmasked raw versions of these bits are available in the [IC_RAW_INTR_STAT](#) register.

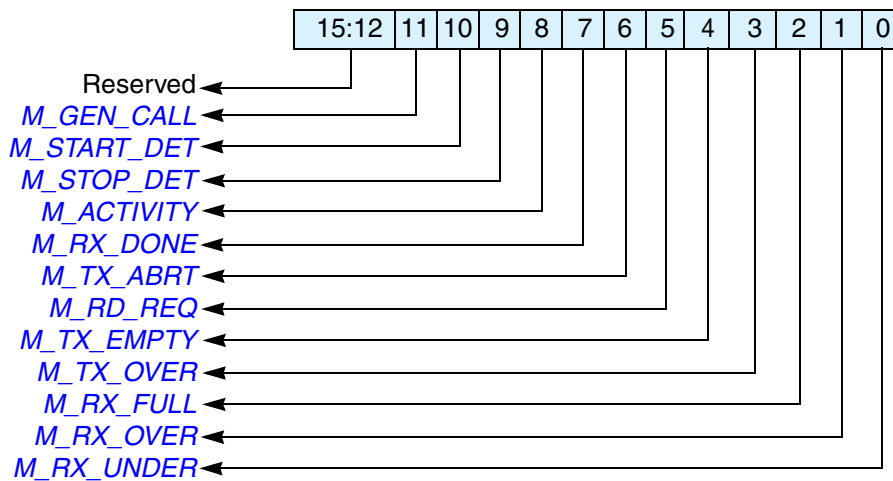


Bits	Name	R/W	Description
15:12	Reserved	N/A	Reserved.
11	<i>R_GEN_CALL</i>	R	See “ IC_RAW_INTR_STAT ” on page 126 for a detailed description of these bits. Reset value: 0x0
10	<i>R_START_DET</i>	R	
9	<i>R_STOP_DET</i>	R	
8	<i>R_ACTIVITY</i>	R	
7	<i>R_RX_DONE</i>	R	
6	<i>R_TX_ABRT</i>	R	
5	<i>R_RD_REQ</i>	R	
4	<i>R_TX_EMPTY</i>	R	
3	<i>R_TX_OVER</i>	R	
2	<i>R_RX_FULL</i>	R	
1	<i>R_RX_OVER</i>	R	
0	<i>R_RX_UNDER</i>	R	

IC_INTR_MASK

- **Name:** I²C Interrupt Mask Register
- **Size:** 12 bits
- **Address Offset:** 0x30
- **Read/Write Access:** Read/Write

These bits mask their corresponding interrupt status bits. They are active high; a value of 0 prevents a bit from generating an interrupt.

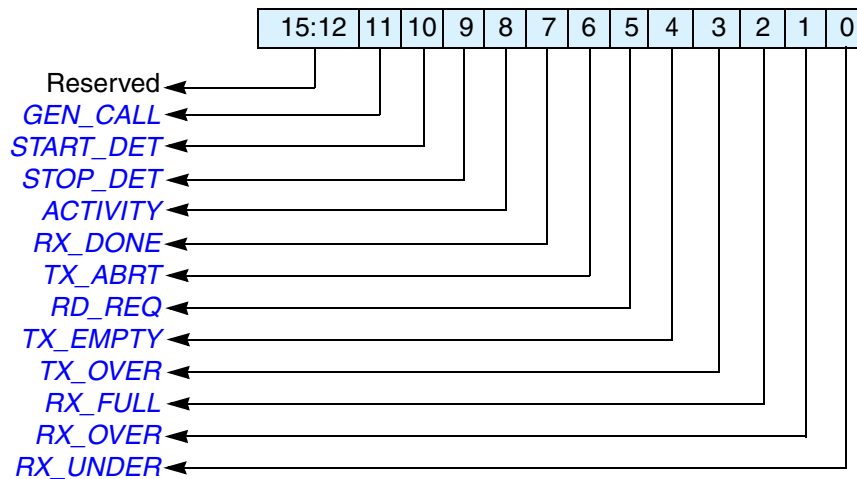


Bits	Name	R/W	Description
15:12	Reserved	N/A	Reserved.
11	<i>M_GEN_CALL</i>	R/W	These bits mask their corresponding interrupt status bits in the IC_INTR_STAT register. This bit should be set to “1” when the IC_ACK_GENERAL_CALL register is set to “0”. Reset value: 12’h8ff
10	<i>M_START_DET</i>	R/W	These bits mask their corresponding interrupt status bits in the IC_INTR_STAT register. Reset value: 12’h8ff
9	<i>M_STOP_DET</i>	R/W	
8	<i>M_ACTIVITY</i>	R/W	
7	<i>M_RX_DONE</i>	R/W	
6	<i>M_TX_ABRT</i>	R/W	
5	<i>M_RD_REQ</i>	R/W	
4	<i>M_TX_EMPTY</i>	R/W	
3	<i>M_TX_OVER</i>	R/W	
2	<i>M_RX_FULL</i>	R/W	
1	<i>M_RX_OVER</i>	R/W	
0	<i>M_RX_UNDER</i>	R/W	

IC_RAW_INTR_STAT

- **Name:** I²C Raw Interrupt Status Register
- **Size:** 12 bits
- **Address Offset:** 0x34
- **Read/Write Access:** Read

Unlike the *IC_INTR_STAT* register, these bits are not masked so they always show the true status of the DW_apb_i2c.



Bits	Name	R/W	Description
15:12	Reserved	N/A	Reserved.
11	<i>GEN_CALL</i>	R	Set only when a General Call address is received and it is acknowledged. It stays set until it is cleared either by disabling DW_apb_i2c or when the CPU reads bit 0 of the <i>IC_CLR_GEN_CALL</i> register. DW_apb_i2c stores the received data in the Rx buffer. Reset value: 0x0
10	<i>START_DET</i>	R	Indicates whether a START or RESTART condition has occurred on the I ² C interface regardless of whether DW_apb_i2c is operating in slave or master mode. Reset value: 0x0
9	<i>STOP_DET</i>	R	Indicates whether a STOP condition has occurred on the I ² C interface regardless of whether DW_apb_i2c is operating in slave or master mode. Reset value: 0x0



Note

Bits 9 and 10 are used in debug mode.

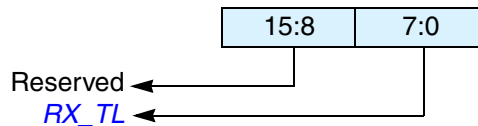
There is no status bit for a RESTART condition because it is detected as a normal start condition. The I²C protocol does not care whether it is a START or RESTART because both conditions start from the IDLE state and send the message to all the slaves on the bus.

Bits	Name	R/W	Description
8	<i>ACTIVITY</i>	R	<p>This bit captures DW_apb_i2c activity and stays set until it is cleared. There are four ways to clear it:</p> <ul style="list-style-type: none"> ● Disabling the DW_apb_i2c ● Reading the <i>IC_CLR_ACTIVITY</i> register ● Reading the <i>IC_CLR_INTR</i> register ● System reset <p>Once this bit is set, it stays set unless one of the four methods is used to clear it. Even if the DW_apb_i2c module is idle, this bit remains set until cleared, indicating that there was activity on the bus.</p> <p>Reset value: 0x0</p>
7	<i>RX_DONE</i>	R	<p>When the DW_apb_i2c is acting as a slave-transmitter, this bit is set to 1 if the master does not acknowledge a transmitted byte. This occurs on the last byte of the transmission, indicating that the transmission is done.</p> <p>Reset value: 0x0</p>
6	<i>TX_ABRT</i>	R	<p>This bit indicates if DW_apb_i2c, as an I²C transmitter, is unable to complete the intended actions on the contents of the transmit FIFO. This situation can occur both as an I²C master or an I²C slave, and is referred to as a “transmit abort”.</p> <p>When this bit is set to 1, the <i>IC_TX_ABRT_SOURCE</i> register indicates the reason why the transmit abort takes places.</p> <p>NOTE: The DW_apb_i2c flushes/resets/empties the TX FIFO whenever this bit is set. The TX FIFO remains in this flushed state until the register <i>IC_CLR_TX_ABRT</i> is read. Once this read is performed, the TX FIFO is then ready to accept more data bytes from the APB interface.</p> <p>Reset value: 0x0</p>
5	<i>RD_REQ</i>	R	<p>This bit is set to 1 when DW_apb_i2c is acting as a slave and another I²C master is attempting to read data from DW_apb_i2c. The DW_apb_i2c holds the I²C bus in a wait state (SCL=0) until this interrupt is serviced, which means that the slave has been addressed by a remote master that is asking for data to be transferred. The processor must respond to this interrupt and then write the requested data to the <i>IC_DATA_CMD</i> register. This bit is set to 0 just after the processor reads the <i>IC_CLR_RD_REQ</i> register.</p> <p>Reset value: 0x0</p>
4	<i>TX_EMPTY</i>	R	<p>This bit is set to 1 when the transmit buffer is at or below the threshold value set in the <i>IC_TX_TL</i> register. It is automatically cleared by hardware when the buffer level goes above the threshold. When the <i>IC_ENABLE</i> bit 0 is 0, the TX FIFO is flushed and held in reset. When the TX FIFO looks like it has no data within it, so this bit is set to 1, provided there is activity in the master or slave state machines. When there is no longer activity, then with <i>ic_en</i>=0, this bit is set to 0.</p> <p>Reset value: 0x0</p>
3	<i>TX_OVER</i>	R	<p>Set during transmit if the transmit buffer is filled to <i>IC_TX_BUFFER_DEPTH</i> and the processor attempts to issue another I²C command by writing to the <i>IC_DATA_CMD</i> register. When the module is disabled, this bit keeps its level until the master or slave state machines go into idle, and when <i>ic_en</i> goes to 0, this interrupt is cleared.</p> <p>Reset value: 0x0</p>

Bits	Name	R/W	Description
2	<i>RX_FULL</i>	R	Set when the receive buffer reaches or goes above the <i>RX_TL</i> threshold in the <i>IC_RX_TL</i> register. It is automatically cleared by hardware when buffer level goes below the threshold. If the module is disabled (<i>IC_ENABLE</i> [0]=0), the RX FIFO is flushed and held in reset; therefore the RX FIFO is not full. So this bit is cleared once the <i>IC_ENABLE</i> bit 0 is programmed with a 0, regardless of the activity that continues. Reset value: 0x0
1	<i>RX_OVER</i>	R	Set if the receive buffer is completely filled to <i>IC_RX_BUFFER_DEPTH</i> and an additional byte is received from an external I2C device. The DW_apb_i2c acknowledges this, but any data bytes received after the FIFO is full are lost. If the module is disabled (<i>IC_ENABLE</i> [0]=0), this bit keeps its level until the master or slave state machines go into idle, and when <i>ic_en</i> goes to 0, this interrupt is cleared. Reset value: 0x0
0	<i>RX_UNDER</i>	R	Set if the processor attempts to read the receive buffer when it is empty by reading from the <i>IC_DATA_CMD</i> register. If the module is disabled (<i>IC_ENABLE</i> [0]=0), this bit keeps its level until the master or slave state machines go into idle, and when <i>ic_en</i> goes to 0, this interrupt is cleared. Reset value: 0x0

IC_RX_TL

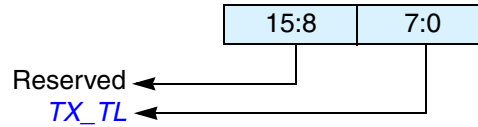
- **Name:** I²C Receive FIFO Threshold Register
- **Size:** 8bits
- **Address Offset:** 0x38
- **Read/Write Access:** Read/Write



Bits	Name	R/W	Description
15:8	Reserved	N/A	Reserved.
7:0	<i>RX_TL</i>	R/W	Receive FIFO Threshold Level Controls the level of entries (or above) that triggers the <i>RX_FULL</i> interrupt (bit 2 in <i>IC_RAW_INTR_STAT</i> register). The valid range is 0-255, with the additional restriction that hardware does not allow this value to be set to a value larger than the depth of the buffer. If an attempt is made to do that, the actual value set will be the maximum depth of the buffer. A value of 0 sets the threshold for 1 entry, and a value of 255 sets the threshold for 256 entries. Reset value: <i>IC_RX_TL</i> configuration parameter

IC_TX_TL

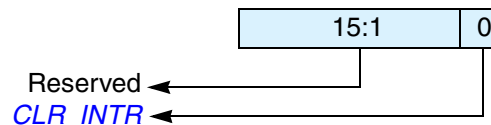
- **Name:** I²C Transmit FIFO Threshold Register
- **Size:** 8 bits
- **Address Offset:** 0x3c
- **Read/Write Access:** Read/Write



Bits	Name	R/W	Description
15:8	Reserved	N/A	Reserved.
7:0	<i>TX_TL</i>	R/W	<p>Transmit FIFO Threshold Level</p> <p>Controls the level of entries (or below) that trigger the <i>TX_EMPTY</i> interrupt (bit 4 in IC_RAW_INTR_STAT register). The valid range is 0-255, with the additional restriction that it may not be set to value larger than the depth of the buffer. If an attempt is made to do that, the actual value set will be the maximum depth of the buffer.</p> <p>A value of 0 sets the threshold for 0 entries, and a value of 255 sets the threshold for 255 entries.</p> <p>Reset value: IC_TX_TL configuration parameter</p>

IC_CLR_INTR

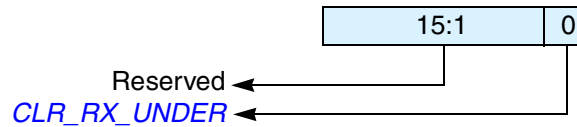
- **Name:** Clear Combined and Individual Interrupt Register
- **Size:** 1 bit
- **Address Offset:** 0x40
- **Read/Write Access:** Read



Bits	Name	R/W	Description
15:1	Reserved	N/A	Reserved.
0	<i>CLR_INTR</i>	R	<p>Read this register to clear the combined interrupt, all individual interrupts, and the <i>IC_TX_ABRT_SOURCE</i> register. This bit does not clear hardware clearable interrupts but software clearable interrupts. Refer to Bit 9 of the <i>IC_TX_ABRT_SOURCE</i> register for an exception to clearing <i>IC_TX_ABRT_SOURCE</i>.</p> <p>Reset value: 0x0</p>

IC_CLR_RX_UNDER

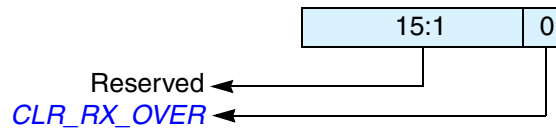
- **Name:** Clear RX_UNDER Interrupt Register
- **Size:** 1 bit
- **Address Offset:** 0x44
- **Read/Write Access:** Read



Bits	Name	R/W	Description
15:1	Reserved	N/A	Reserved.
0	<i>CLR_RX_UNDER</i>	R	Read this register to clear the <i>RX_UNDER</i> interrupt (bit 0) of the <i>IC_RAW_INTR_STAT</i> register. Reset value: 0x0

IC_CLR_RX_OVER

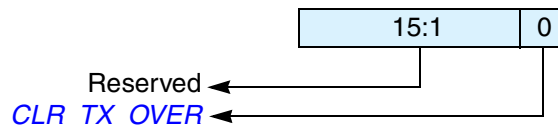
- **Name:** Clear RX_OVER Interrupt Register
- **Size:** 1 bit
- **Address Offset:** 0x48
- **Read/Write Access:** Read



Bits	Name	R/W	Description
15:1	Reserved	N/A	Reserved.
0	<i>CLR_RX_OVER</i>	R	Read this register to clear the <i>RX_OVER</i> interrupt (bit 1) of the <i>IC_RAW_INTR_STAT</i> register. Reset value: 0x0

IC_CLR_TX_OVER

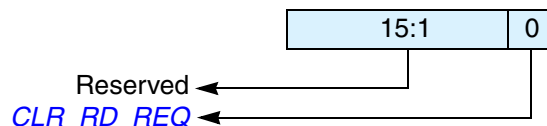
- **Name:** Clear TX_OVER Interrupt Register
- **Size:** 1 bit
- **Address Offset:** 0x4c
- **Read/Write Access:** Read



Bits	Name	R/W	Description
15:1	Reserved	N/A	Reserved.
0	<i>CLR_TX_OVER</i>	R	Read this register to clear the <i>TX_OVER</i> interrupt (bit 3) of the <i>IC_RAW_INTR_STAT</i> register. Reset value: 0x0

IC_CLR_RD_REQ

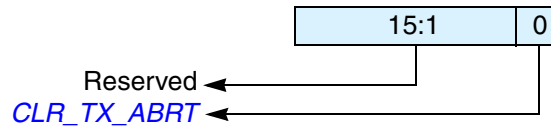
- **Name:** Clear RD_REQ Interrupt Register
- **Size:** 1 bit
- **Address Offset:** 0x50
- **Read/Write Access:** Read



Bits	Name	R/W	Description
15:1	Reserved	N/A	Reserved.
0	<i>CLR_RD_REQ</i>	R	Read this register to clear the <i>RD_REQ</i> interrupt (bit 5) of the <i>IC_RAW_INTR_STAT</i> register. Reset value: 0x0

IC_CLR_TX_ABRT

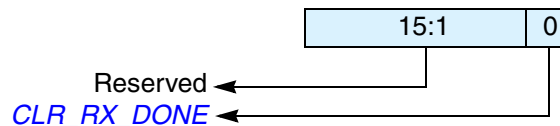
- **Name:** Clear TX_ABRT Interrupt Register
- **Size:** 1 bit
- **Address Offset:** 0x54
- **Read/Write Access:** Read



Bits	Name	R/W	Description
15:1	Reserved	N/A	Reserved.
0	CLR_TX_ABRT	R	Read this register to clear the TX_ABRT interrupt (bit 6) of the IC_RAW_INTR_STAT register, and the IC_TX_ABRT_SOURCE register. This also releases the TX FIFO from the flushed/reset state, allowing more writes to the TX FIFO. Refer to Bit 9 of the IC_TX_ABRT_SOURCE register for an exception to clearing IC_TX_ABRT_SOURCE. Reset value: 0x0

IC_CLR_RX_DONE

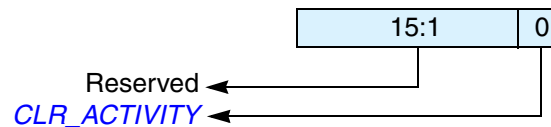
- **Name:** Clear RX_DONE Interrupt Register
- **Size:** 1 bit
- **Address Offset:** 0x58
- **Read/Write Access:** Read



Bits	Name	R/W	Description
15:1	Reserved	N/A	Reserved.
0	CLR_RX_DONE	R	Read this register to clear the RX_DONE interrupt (bit 7) of the IC_RAW_INTR_STAT register. Reset value: 0x0

IC_CLR_ACTIVITY

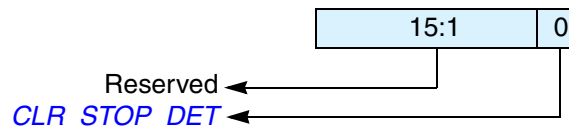
- **Name:** Clear ACTIVITY Interrupt Register
- **Size:** 1 bit
- **Address Offset:** 0x5c
- **Read/Write Access:** Read



Bits	Name	R.W	Description
15:1	Reserved	N/A	Reserved.
0	<i>CLR_ACTIVITY</i>	R	Reading this register clears the <i>ACTIVITY</i> interrupt if the I ² C is not active anymore. If the I ² C module is still active on the bus, the <i>ACTIVITY</i> interrupt bit continues to be set. It is automatically cleared by hardware if the module is disabled and if there is no further activity on the bus. The value read from this register to get status of the <i>ACTIVITY</i> interrupt (bit 8) of the <i>IC_RAW_INTR_STAT</i> register. Reset value: 0x0

IC_CLR_STOP_DET

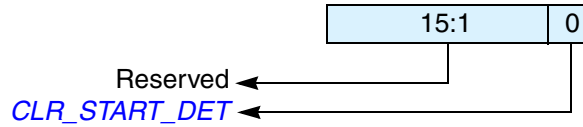
- **Name:** Clear STOP_DET Interrupt Register
- **Size:** 1 bit
- **Address Offset:** 0x60
- **Read/Write Access:** Read



Bits	Name	R/W	Description
15:1	Reserved	N/A	Reserved.
0	<i>CLR_STOP_DET</i>	R	Read this register to clear the <i>STOP_DET</i> interrupt (bit 9) of the <i>IC_RAW_INTR_STAT</i> register. Reset value: 0x0

IC_CLR_START_DET

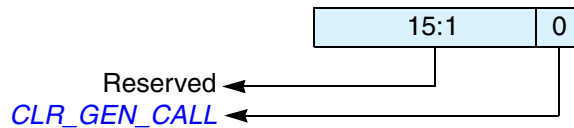
- **Name:** Clear START_DET Interrupt Register
- **Size:** 1 bit
- **Address Offset:** 0x64
- **Read/Write Access:** Read



Bits	Name	R/W	Description
15:1	Reserved	N/A	Reserved.
0	<i>CLR_START_DET</i>	R	Read this register to clear the <i>START_DET</i> interrupt (bit 10) of the <i>IC_RAW_INTR_STAT</i> register. Reset value: 0x0

IC_CLR_GEN_CALL

- **Name:** Clear GEN_CALL Interrupt Register
- **Size:** 1 bit
- **Address Offset:** 0x68
- **Read/Write Access:** Read



Bits	Name	R/W	Description
15:1	Reserved	N/A	Reserved.
0	<i>CLR_GEN_CALL</i>	R	Read this register to clear the <i>GEN_CALL</i> interrupt (bit 11) of the <i>IC_RAW_INTR_STAT</i> register. Reset value: 0x0

Operation of the Interrupt Registers

The following figures illustrate the operation of the DW_apb_i2c interrupt registers and how they are set and cleared. Some bits are set by hardware and cleared by software, whereas other bits are set and cleared by hardware, as indicated in [Table 12](#). [Figure 27](#) shows the operation of the interrupt registers where the bits are set by hardware and cleared by software.

Table 12: Setting and Clearing of Interrupt Bits

Interrupt Bit Fields	Set by Hardware/ Cleared by Software	Set and Cleared by Hardware
GEN_CALL	✓	✗
START_DET	✓	✗
STOP_DET	✓	✗
ACTIVITY	✗	✓
RX_DONE	✓	✗
TX_ABRT	✓	✗
RD_REQ	✓	✗
TX_EMPTY	✗	✓
TX_OVER	✓	✗
RX_FULL	✗	✓
RX_OVER	✓	✗
RX_UNDER	✓	✗

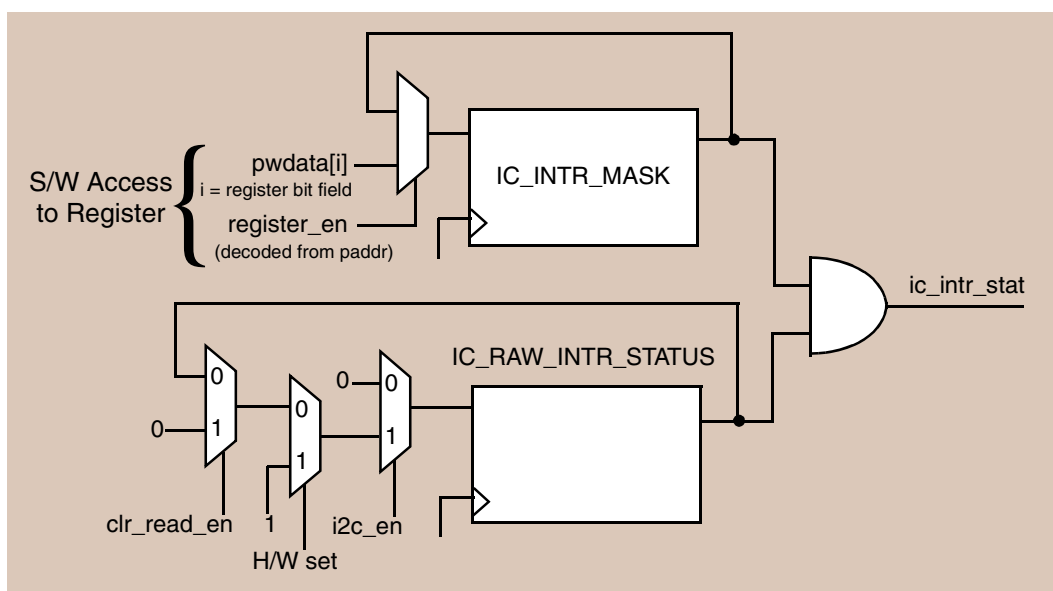
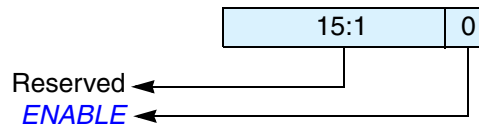


Figure 27: Interrupt Scheme

IC_ENABLE

- Name: I²C Enable Register
- Size: 1 bit
- Address Offset: 0x6c
- Read/Write Access: Read/Write



Bits	Name	R/W	Description
15:1	Reserved	N/A	Reserved.
0	<i>ENABLE</i>	R/W	<p>Controls whether the DW_apb_i2c is enabled.</p> <p>0: Disables DW_apb_i2c (TX and RX FIFOs are held in an erased state)</p> <p>1: Enables DW_apb_i2c</p> <p>Software can disable DW_apb_i2c while it is active. However, it is important that care be taken to ensure that DW_apb_i2c is disabled properly. A recommended procedure is described in “Disabling DW_apb_i2c” on page 62.</p> <p>When DW_apb_i2c is disabled, the following occurs:</p> <ul style="list-style-type: none"> • The TX FIFO and RX FIFO get flushed. • Status bits in the <i>IC_INTR_STAT</i> register are still active until DW_apb_i2c goes into IDLE state. <p>If the module is transmitting, it stops as well as deletes the contents of the transmit buffer after the current transfer is complete. If the module is receiving, the DW_apb_i2c stops the current transfer at the end of the current byte and does not acknowledge the transfer.</p> <p>In systems with asynchronous pclk and ic_clk when IC_CLK_TYPE parameter set to asynchronous (1), there is a two ic_clk delay when enabling or disabling the DW_apb_i2c.</p> <p>For a detailed description on how to disable DW_apb_i2c, refer to “Disabling DW_apb_i2c” on page 62.</p> <p>Reset value: 0x0</p>

IC_STATUS

- **Name:** I²C Status Register
- **Size:** 7 bits
- **Address Offset:** 0x70
- **Read/Write Access:** Read

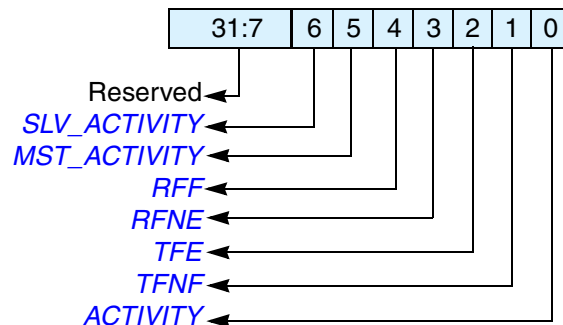
This is a read-only register used to indicate the current transfer status and FIFO status. The status register may be read at any time. None of the bits in this register request an interrupt.


When the I²C is disabled by writing 0 in bit 0 of the *IC_ENABLE* register:

- Bits 1 and 2 are set to 1
- Bits 3 and 4 are set to 0

When the master or slave state machines goes to idle and *ic_en*=0:

- Bits 5 and 6 are set to 0



Bits	Name	R/W	Description
31:7	<i>Reserved</i>	N/A	Reserved.
6	<i>SLV_ACTIVITY</i>	R	Slave FSM Activity Status. When the Slave Finite State Machine (FSM) is not in the IDLE state, this bit is set. 0: Slave FSM is in IDLE state so the Slave part of DW_apb_i2c is not Active 1: Slave FSM is not in IDLE state so the Slave part of DW_apb_i2c is Active Reset value: 0x0
5	<i>MST_ACTIVITY</i>	R	Master FSM Activity Status. When the Master Finite State Machine (FSM) is not in the IDLE state, this bit is set. 0: Master FSM is in IDLE state so the Master part of DW_apb_i2c is not Active 1: Master FSM is not in IDLE state so the Master part of DW_apb_i2c is Active  Note ————— IC_STATUS[0]—that is, <i>ACTIVITY</i> bit—is the OR of <i>SLV_ACTIVITY</i> and <i>MST_ACTIVITY</i> bits. ————— Reset value: 0x0

Bits	Name	R/W	Description
4	<i>RFF</i>	R	Receive FIFO Completely Full. When the receive FIFO is completely full, this bit is set. When the receive FIFO contains one or more empty location, this bit is cleared. 0: Receive FIFO is not full 1: Receive FIFO is full Reset value: 0x0
3	<i>RFNE</i>	R	Receive FIFO Not Empty. This bit is set when the receive FIFO contains one or more entries; it is cleared when the receive FIFO is empty. 0: Receive FIFO is empty 1: Receive FIFO is not empty Reset value: 0x0
2	<i>TFE</i>	R	Transmit FIFO Completely Empty. When the transmit FIFO is completely empty, this bit is set. When it contains one or more valid entries, this bit is cleared. This bit field does not request an interrupt. 0: Transmit FIFO is not empty 1: Transmit FIFO is empty Reset value: 0x1
1	<i>TFNF</i>	R	Transmit FIFO Not Full. Set when the transmit FIFO contains one or more empty locations, and is cleared when the FIFO is full. 0: Transmit FIFO is full 1: Transmit FIFO is not full Reset value: 0x1
0	<i>ACTIVITY</i>	R	I ² C Activity Status. Reset value: 0x0

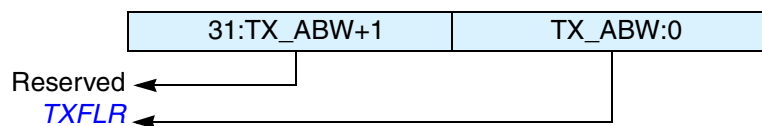
IC_TXFLR

- **Name:** I²C Transmit FIFO Level Register
- **Size:** TX_ABW + 1
- **Address Offset:** 0x74
- **Read/Write Access:** Read

This register contains the number of valid data entries in the transmit FIFO buffer. It is cleared whenever:

- The I²C is disabled
- There is a transmit abort—that is, *TX_ABRT* bit is set in the *IC_RAW_INTR_STAT* register
- The slave bulk transmit mode is aborted

The register increments whenever data is placed into the transmit FIFO and decrements when data is taken from the transmit FIFO.



Bits	Name	R/W	Description
31:TX_ABW+1	Reserved	N/A	Reserved
TX_ABW:0	<i>TXFLR</i>	R	Transmit FIFO Level. Contains the number of valid data entries in the transmit FIFO. Reset value: 0x0

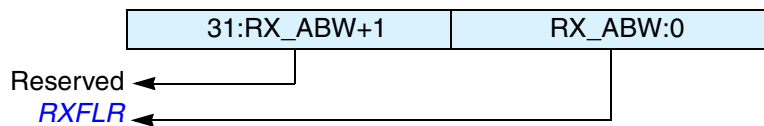
IC_RXFLR

- **Name:** I²C Receive FIFO Level Register
- **Size:** RX_ABW + 1
- **Address Offset:** 0x78
- **Read/Write Access:** Read

This register contains the number of valid data entries in the receive FIFO buffer. It is cleared whenever:

- The I²C is disabled
- Whenever there is a transmit abort caused by any of the events tracked in *IC_TX_ABRT_SOURCE*

The register increments whenever data is placed into the receive FIFO and decrements when data is taken from the receive FIFO.

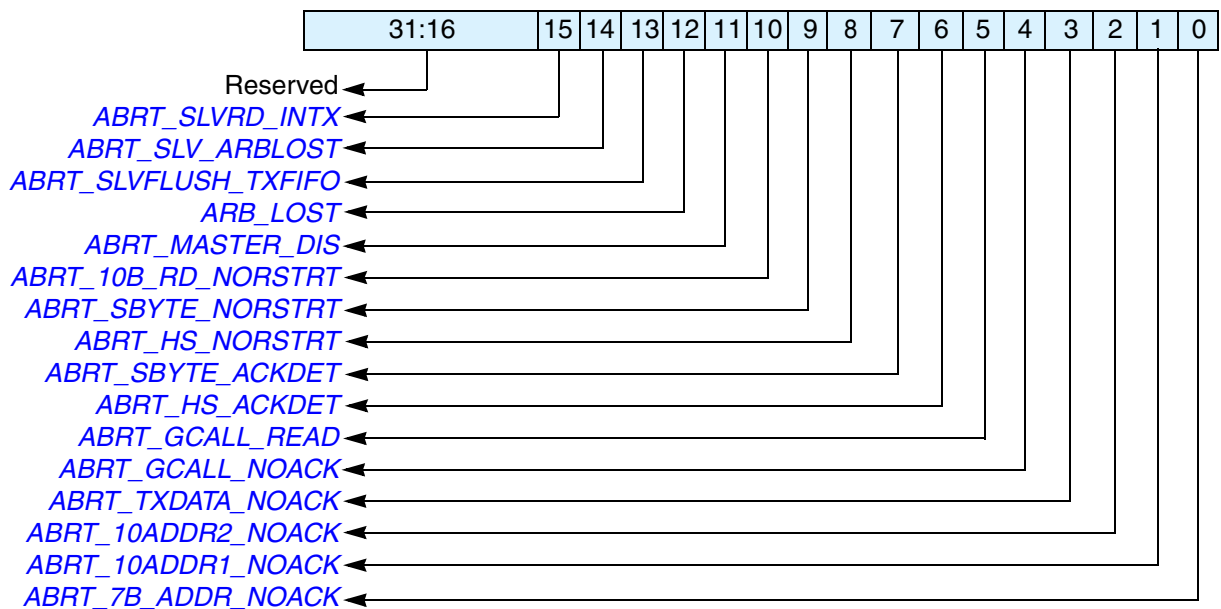


Bits	Name	R/W	Description
31:RX_ABW+1	Reserved	N/A	Reserved
RX_ABW:0	<i>RXFLR</i>	R	Receive FIFO Level. Contains the number of valid data entries in the receive FIFO. Reset value: 0x0

IC_TX_ABRT_SOURCE

- **Name:** I²C Transmit Abort Source Register
- **Size:** 16 bits
- **Address Offset:** 0x80
- **Read/Write Access:** Read/Write

This register has 16 bits that indicate the source of the *TX_ABRT* bit. Except for Bit 9, this register is cleared whenever the *IC_CLR_TX_ABRT* register or the *IC_CLR_INTR* register is read. To clear Bit 9, the source of the *ABRT_SBYTE_NORSTRT* must be fixed first; RESTART must be enabled (*IC_CON*[5]=1), the SPECIAL bit must be cleared (*IC_TAR*[11]), or the GC_OR_START bit must be cleared (*IC_TAR*[10]). Once the source of the *ABRT_SBYTE_NORSTRT* is fixed, then this bit can be cleared in the same manner as other bits in this register. If the source of the *ABRT_SBYTE_NORSTRT* is not fixed before attempting to clear this bit, Bit 9 clears for one cycle and is then re-asserted.



Bits	Name	R/W	Description	Role of DW_apb_i2c
31:16	Reserved	N/A	Reserved	
15	<i>ABRT_SLVRD_INTX</i>	R/W	1: When the processor side responds to a slave mode request for data to be transmitted to a remote master and user writes a 1 in <i>CMD</i> (bit 8) of <i>IC_DATA_CMD</i> register. Reset value: 0x0	Slave-Transmitter

Bits	Name	R/W	Description	Role of DW_apb_i2c
14	<i>ABRT_SLV_ARBLOST</i>	R/W	1: Slave lost the bus while transmitting data to a remote master. <i>IC_TX_ABRT_SOURCE</i> [12] is set at the same time. Note: Even though the slave never “owns” the bus, something could go wrong on the bus. This is a fail safe check. For instance, during a data transmission at the low-to-high transition of SCL, if what is on the data bus is not what is supposed to be transmitted, then DW_apb_i2c no longer own the bus. Reset value: 0x0	Slave-Transmitter
13	<i>ABRT_SLVFLUSH_TXFIFO</i>	R/W	1: Slave has received a read command and some data exists in the TX FIFO so the slave issues a <i>TX_ABRT</i> interrupt to flush old data in TX FIFO. Reset value: 0x0	Slave-Transmitter
12	<i>ARB_LOST</i>	R/W	1: Master has lost arbitration, or if <i>IC_TX_ABRT_SOURCE</i> [14] is also set, then the slave transmitter has lost arbitration. Note: I ² C can be both master and slave at the same time. Reset value: 0x0	Master-Transmitter or Slave-Transmitter
11	<i>ABRT_MASTER_DIS</i>	R/W	1: User tries to initiate a Master operation with the Master mode disabled. Reset value: 0x0	Master-Transmitter or Master-Receiver
10	<i>ABRT_10B_RD_NORSTRT</i>	R/W	1: The restart is disabled (<i>IC_RESTART_EN</i> bit (<i>IC_CON</i> [5]) = 0) and the master sends a read command in 10-bit addressing mode. Reset value: 0x0	Master-Receiver

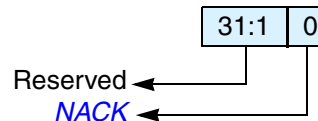
Bits	Name	R/W	Description	Role of DW_apb_i2c
9	<i>ABRT_SBYTE_NORSTRT</i>	R/W	To clear Bit 9, the source of the <i>ABRT_SBYTE_NORSTRT</i> must be fixed first; restart must be enabled (<i>IC_CON</i> [5]=1), the <i>SPECIAL</i> bit must be cleared (<i>IC_TAR</i> [11]), or the <i>GC_OR_START</i> bit must be cleared (<i>IC_TAR</i> [10]). Once the source of the <i>ABRT_SBYTE_NORSTRT</i> is fixed, then this bit can be cleared in the same manner as other bits in this register. If the source of the <i>ABRT_SBYTE_NORSTRT</i> is not fixed before attempting to clear this bit, bit 9 clears for one cycle and then gets re-asserted. 1: The restart is disabled (<i>IC_RESTART_EN</i> bit (<i>IC_CON</i> [5]) = 0) and the user is trying to send a START Byte. Reset value: 0x0	Master
8	<i>ABRT_HS_NORSTRT</i>	R/W	1: The restart is disabled (<i>IC_RESTART_EN</i> bit (<i>IC_CON</i> [5]) = 0) and the user is trying to use the master to transfer data in High Speed mode. Reset value: 0x0	Master-Transmitter or Master-Receiver
7	<i>ABRT_SBYTE_ACKDET</i>	R/W	1: Master has sent a START Byte and the START Byte was acknowledged (wrong behavior). Reset value: 0x0	Master
6	<i>ABRT_HS_ACKDET</i>	R/W	1: Master is in High Speed mode and the High Speed Master code was acknowledged (wrong behavior). Reset value: 0x0	Master
5	<i>ABRT_GCALL_READ</i>	R/W	1: DW_apb_i2c in master mode sent a General Call but the user programmed the byte following the General Call to be a read from the bus (<i>IC_DATA_CMD</i> [9] is set to 1). Reset value: 0x0	Master-Transmitter
4	<i>ABRT_GCALL_NOACK</i>	R/W	1: DW_apb_i2c in master mode sent a General Call and no slave on the bus acknowledged the General Call. Reset value: 0x0	Master-Transmitter

Bits	Name	R/W	Description	Role of DW_apb_i2c
3	<i>ABRT_TXDATA_NOACK</i>	R/W	1: This is a master-mode only bit. Master has received an acknowledgement for the address, but when it sent data byte(s) following the address, it did not receive an acknowledge from the remote slave(s). Reset value: 0x0	Master-Transmitter
2	<i>ABRT_10ADDR2_NOACK</i>	R/W	1: Master is in 10-bit address mode and the second address byte of the 10-bit address was not acknowledged by any slave. Reset value: 0x0	Master-Transmitter or Master-Receiver
1	<i>ABRT_10ADDR1_NOACK</i>	R/W	1: Master is in 10-bit address mode and the first 10-bit address byte was not acknowledged by any slave. Reset value: 0x0	Master-Transmitter or Master-Receiver
0	<i>ABRT_7B_ADDR_NOACK</i>	R/W	1: Master is in 7-bit addressing mode and the address sent was not acknowledged by any slave. Reset value: 0x0	Master-Transmitter or Master-Receiver

IC_SLV_DATA_NACK_ONLY

- **Name:** Generate Slave Data NACK Register
- **Size:** 1 bit
- **Address Offset:** 0x84
- **Read/Write Access:** Read/Write

The register is used to generate a NACK for the data part of a transfer when DW_apb_i2c is acting as a slave-receiver. This register only exists when the IC_SLV_DATA_NACK_ONLY parameter is set to 1. When this parameter disabled, this register does not exist and writing to the register's address has no effect.

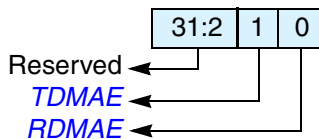


Bits	Name	R/W	Description
31:1	Reserved	N/A	Reserved.
0	<i>NACK</i>	R/W	<p>Generate NACK. This NACK generation only occurs when DW_apb_i2c is a slave-receiver. If this register is set to a value of 1, it can only generate a NACK after a data byte is received; hence, the data transfer is aborted and the data received is not pushed to the receive buffer.</p> <p>When the register is set to a value of 0, it generates NACK/ACK, depending on normal criteria.</p> <p>1 = generate NACK after data byte received 0 = generate NACK/ACK normally</p> <p>Reset value: 0x0</p>

IC_DMA_CR

- **Name:** DMA Control Register
- **Size:** 2 bits
- **Address Offset:** 0x88
- **Read/Write Access:** Read/Write

This register is only valid when DW_apb_i2c is configured with a set of DMA Controller interface signals (IC_HAS_DMA = 1). When DW_apb_i2c is not configured for DMA operation, this register does not exist and writing to the register’s address has no effect and reading from this register address will return zero. The register is used to enable the DMA Controller interface operation. There is a separate bit for transmit and receive. This can be programmed regardless of the state of IC_ENABLE.

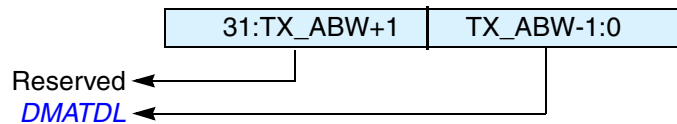


Bits	Name	R/W	Description
31:2	Reserved	N/A	Reserved.
1	<i>TDMAE</i>	R/W	Transmit DMA Enable. This bit enables/disables the transmit FIFO DMA channel. 0 = Transmit DMA disabled 1 = Transmit DMA enabled Reset value: 0x0
0	<i>RDMAE</i>	R/W	Receive DMA Enable. This bit enables/disables the receive FIFO DMA channel. 0 = Receive DMA disabled 1 = Receive DMA enabled Reset value: 0x0

IC_DMA_TDLR

- **Name:** DMA Transmit Data Level Register
- **Size:** TX_ABW-1:0
- **Address Offset:** 0x8c
- **Read/Write Access:** Read/Write

This register is only valid when the DW_apb_i2c is configured with a set of DMA interface signals (IC_HAS_DMA = 1). When DW_apb_i2c is not configured for DMA operation, this register does not exist; writing to its address has no effect; reading from its address returns zero.

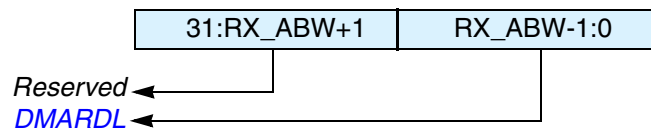


Bits	Name	R/W	Description
31:TX_ABW	Reserved	N/A	Reserved
TX_ABW-1:0	<i>DMATDL</i>	R/W	Transmit Data Level. This bit field controls the level at which a DMA request is made by the transmit logic. It is equal to the watermark level; that is, the dma_tx_req signal is generated when the number of valid data entries in the transmit FIFO is equal to or below this field value, and <i>TDMAE</i> = 1. Reset value: 0x0

IC_DMA_RDLR

- **Name:** I²C Receive Data Level Register
- **Size:** RX_ABW-1:0
- **Address Offset:** 0x90
- **Read/Write Access:** Read/Write

This register is only valid when DW_apb_i2c is configured with a set of DMA interface signals (IC_HAS_DMA = 1). When DW_apb_i2c is not configured for DMA operation, this register does not exist; writing to its address has no effect; reading from its address returns zero.

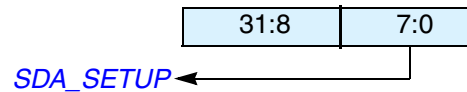


Bits	Name	R/W	Description
31:RX_ABW	Reserved	N/A	Reserved
RX_ABW-1:0	<i>DMARDL</i>	R/W	Receive Data Level. This bit field controls the level at which a DMA request is made by the receive logic. The watermark level = <i>DMARDL</i> +1; that is, dma_rx_req is generated when the number of valid data entries in the receive FIFO is equal to or more than this field value + 1, and <i>RDMAE</i> = 1. For instance, when <i>DMARDL</i> is 0, then dma_rx_req is asserted when 1 or more data entries are present in the receive FIFO. Reset value: 0x0

IC_SDA_SETUP

- **Name:** I²C SDA Setup Register
- **Size:** 8 bits
- **Address Offset:** 0x94
- **Read/Write Access:** Read/Write

This register controls the amount of time delay (in terms of number of ic_clk clock periods) introduced in the rising edge of SCL, relative to SDA changing, when DW_apb_i2c services a read request in a slave-transmitter operation. The relevant I²C requirement is $t_{SU:DAT}$ (note 4) as detailed in the *I2C Bus Specification*.

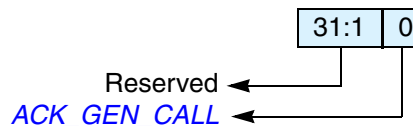


Bits	Name	R/W	Description
31:8	Reserved	N/A	Reserved
7:0	<i>SDA_SETUP</i>	R/W	SDA Setup. It is recommended that if the required delay is 1000ns, then for an ic_clk frequency of 10 MHz, IC_SDA_SETUP should be programmed to a value of 11. Default Reset value: 0x64, but can be hardcoded by setting the IC_DEFAULT_SDA_SETUP configuration parameter.

IC_ACK_GENERAL_CALL

- **Name:** I²C ACK General Call Register
- **Size:** 1 bit
- **Address Offset:** 0x98
- **Read/Write Access:** Read/Write

The register controls whether DW_apb_i2c responds with a ACK or NACK when it receives an I²C General Call address.



Bits	Name	R/W	Description
31:1	Reserved	N/A	Reserved
0	<i>ACK_GEN_CALL</i>	R/W	ACK General Call. When set to 1, DW_apb_i2c responds with a ACK (by asserting ic_data_oe) when it receives a General Call. Otherwise, DW_apb_i2c responds with a NACK (by negating ic_data_oe). Default Reset value: 0x1, but can be hardcoded by setting the IC_DEFAULT_ACK_GENERAL_CALL configuration parameter.

IC_ENABLE_STATUS

- **Name:** I²C Enable Status Register
- **Size:** 3 bits
- **Address Offset:** 0x9C
- **Read/Write Access:** Read

The register is used to report the DW_apb_i2c hardware status when the *IC_ENABLE* register is set from 1 to 0; that is, when DW_apb_i2c is disabled.

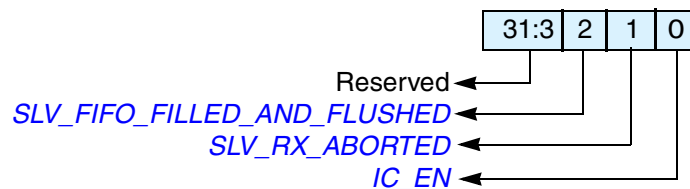
If *IC_ENABLE* has been set to 1, bits 2:1 are forced to 0, and bit 0 is forced to 1.

If *IC_ENABLE* has been set to 0, bits 2:1 is only be valid as soon as bit 0 is read as '0'.



Note

When *IC_ENABLE* has been written with '0,' a delay occurs for bit 0 to be read as '0' because disabling the DW_apb_i2c depends on I²C bus activities.

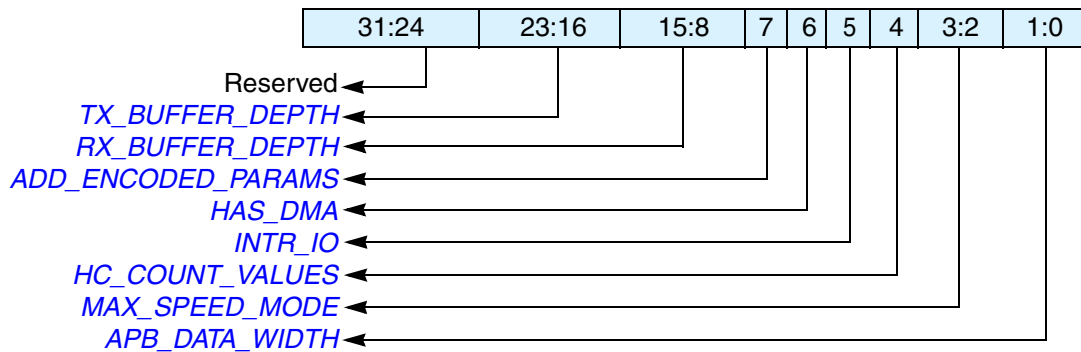



Bits	Name	R/W	Description
31:3	Reserved	N/A	Reserved
2	<i>SLV_RX_DATA_LOST</i>	R	<p>Slave Received Data Lost. This bit indicates if a Slave-Receiver operation has been aborted with at least one data byte received from an I²C transfer due to the setting of <i>IC_ENABLE</i> from 1 to 0.</p> <p>When read as 1, DW_apb_i2c is deemed to have been actively engaged in an aborted I²C transfer (with matching address) and the data phase of the I²C transfer has been entered, even though a data byte has been responded with a NACK. NOTE: If the remote I²C master terminates the transfer with a STOP condition before the DW_apb_i2c has a chance to NACK a transfer, and <i>IC_ENABLE</i> has been set to 0, then this bit is also set to 1.</p> <p>When read as 0, DW_apb_i2c is deemed to have been disabled without being actively involved in the data phase of a Slave-Receiver transfer. NOTE: The CPU can safely read this bit when <i>IC_EN</i> (bit 0) is read as 0.</p> <p>Reset value: 0x0</p>

Bits	Name	R/W	Description
1	<i>SLV_DISABLED_WHILE_BUSY</i>	R	<p>Slave Disabled While Busy (Transmit, Receive). This bit indicates if a potential or active Slave operation has been aborted due to the setting of the <i>IC_ENABLE</i> register from 1 to 0. This bit is set when the CPU writes a 0 to the <i>IC_ENABLE</i> register while: (a) <i>DW_apb_i2c</i> is receiving the address byte of the Slave-Transmitter operation from a remote master; OR, (b) address and data bytes of the Slave-Receiver operation from a remote master.</p> <p>When read as 1, <i>DW_apb_i2c</i> is deemed to have forced a NACK during any part of an I²C transfer, irrespective of whether the I²C address matches the slave address set in <i>DW_apb_i2c</i> (<i>IC_SAR</i> register) OR if the transfer is completed before <i>IC_ENABLE</i> is set to 0 but has not taken effect.</p> <p>NOTE: If the remote I²C master terminates the transfer with a STOP condition before the <i>DW_apb_i2c</i> has a chance to NACK a transfer, and <i>IC_ENABLE</i> has been set to 0, then this bit will also be set to 1.</p> <p>When read as 0, <i>DW_apb_i2c</i> is deemed to have been disabled when there is master activity, or when the I²C bus is idle.</p> <p>NOTE: The CPU can safely read this bit when <i>IC_EN</i> (bit 0) is read as 0.</p> <p>Reset value: 0x0</p>
0	<i>IC_EN</i>	R	<p>ic_en Status. This bit always reflects the value driven on the output port <i>ic_en</i>.</p> <p>When read as 1, <i>DW_apb_i2c</i> is deemed to be in an enabled state.</p> <p>When read as 0, <i>DW_apb_i2c</i> is deemed completely inactive.</p> <p>NOTE: The CPU can safely read this bit anytime. When this bit is read as 0, the CPU can safely read <i>SLV_RX_DATA_LOST</i> (bit 2) and <i>SLV_DISABLED_WHILE_BUSY</i> (bit 1).</p> <p>Reset value: 0x0</p>

IC_COMP_PARAM_1

- **Name:** Component Parameter Register 1
- **Size:** 32 bits
- **Address Offset:** 0xf4
- **Read/Write Access:** Read

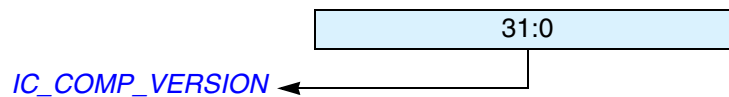


Bits	Name	R/W	Description
 Note ————— This is a constant read-only register that contains encoded information about the component's parameter settings. The reset value depends on coreConsultant parameter(s).			
31:24	Reserved	N/A	Reserved
23:16	<i>TX_BUFFER_DEPTH</i>	R	The value of this register is derived from the IC_TX_BUFFER_DEPTH coreConsultant parameter. 0x00 = Reserved 0x01 = 2 0x02 = 3 to 0xFF = 256
15:8	<i>RX_BUFFER_DEPTH</i>	R	The value of this register is derived from the IC_RX_BUFFER_DEPTH coreConsultant parameter. For a description of this parameter, see Table 8 on page 76 . 0x00 = Reserved 0x01 = 2 0x02 = 3 to 0xFF = 256
7	<i>ADD_ENCODED_PARAMS</i>	R	The value of this register is derived from the IC_ADD_ENCODED_PARAMS coreConsultant parameter. For a description of this parameter, see Table 8 on page 76 . Reading 1 in this bit means that the capability of reading these encoded parameters via software has been included. Otherwise, the entire register is 0 regardless of the setting of any other parameters that are encoded in the bits. 0: False 1: True

Bits	Name	R/W	Description
6	<i>HAS_DMA</i>	R	The value of this register is derived from the IC_HAS_DMA coreConsultant parameter. For a description of this parameter, see Table 8 on page 76 . 0: False 1: True
5	<i>INTR_IO</i>	R	The value of this register is derived from the IC_INTR_IO coreConsultant parameter. For a description of this parameter, see Table 8 on page 76 . 0: Individual 1: Combined
4	<i>HC_COUNT_VALUES</i>	R	The value of this register is derived from the IC_HC_COUNT_VALUES coreConsultant parameter. For a description of this parameter, see Table 8 on page 76 . 0: False 1: True
3:2	<i>MAX_SPEED_MODE</i>	R	The value of this register is derived from the IC_MAX_SPEED_MODE coreConsultant parameter. For a description of this parameter, see Table 8 on page 76 . 0x0 = Reserved 0x1 = Standard 0x2 = Fast 0x3 = High
1:0	<i>APB_DATA_WIDTH</i>	R	The value of this register is derived from the APB_DATA_WIDTH coreConsultant parameter. For a description of this parameter, see Table 8 on page 76 . 0x0 = 8 bits 0x1 = 16 bits 0x2 = 32 bits 0x3 = Reserved

IC_COMP_VERSION

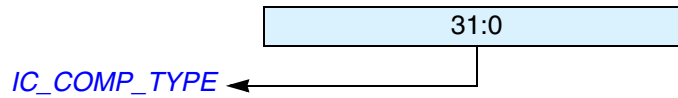
- **Name:** I²C Component Version Register
- **Size:** 32 bits
- **Address Offset:** 0xf8
- **Read/Write Access:** Read



Bits	Name	R/W	Description
31:0	IC_COMP_VERSION	R	Specific values for this register are described in the Releases Table in the DW_apb_i2c Release Notes .

IC_COMP_TYPE

- **Name:** I²C Component Type Register
- **Size:** 32 bits
- **Address Offset:** 0xfc
- **Read/Write Access:** Read



Bits	Name	R/W	Description
31:0	IC_COMP_TYPE	R	Designware Component Type number = 0x44_57_01_40. This assigned unique hex value is constant and is derived from the two ASCII letters “DW” followed by a 16-bit unsigned number.

7

Programming the DW_apb_i2c

The DW_apb_i2c can be programmed via software registers or the DW_apb_i2c low-level software driver. This chapter contains the following topics:

- “Software Registers”
- “Software Drivers”

Software Registers

For information about programming the software registers in terms of DW_apb_i2c operation, refer to “Slave Mode Operation” on page 56 and “Master Mode Operation” on page 60. The software registers are described in more detail in Chapter 6 on page 99, “Registers”.

Software Drivers

The family of DesignWare AMBA Synthesizable Components includes a Driver Kit for the DW_apb_i2c component. This low-level Driver Kit allows you to easily program a DW_apb_i2c 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_i2c minimizing usage errors
- Rapid software development with minimum overhead
- Detailed knowledge of DW_apb_i2c register bit fields not required
- Easy integration of DW_apb_i2c into existing software system
- Programming at register level eliminated

You must purchase a source code license (DWC-APB-Advanced-Source) to use the DW_apb_i2c Driver Kit. However, you can access some Driver Kit files and documentation in \$DESIGNWARE_HOME/drivers/DW_apb_i2c/latest. For more information about the Driver Kit, refer to the *DW_apb_i2c 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 available for DW_apb_i2c verification. Once you have configured the DW_apb_i2c in coreConsultant and have set up the verification environment, you can run simulations automatically. The following sections describe the testbench:

- “Overview of Vera Tests”
- “Overview of DW_apb_i2c Testbench” on page 160

For more information about running simulations for DW_apb_i2c in coreAssembler, refer to “Verify Component” on page 36. For more information about verifying DW_apb_i2c in coreConsultant, see “Verifying the DW_apb_i2c” on page 175.



Note

The DW_apb_i2c 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 Vera Tests

The DW_apb_i2c verification testbench performs the following set of tests that have been written to exhaustively verify the functionality and have also achieved maximum RTL code coverage.



Note

All tests use the APB Interface to program memory mapped registers dynamically during tests.

- “APB Slave Interface” on page 158
- “DW_apb_i2c Master Operation” on page 158
- “DW_apb_i2c Slave Operation” on page 159
- “DW_apb_i2c Interrupts” on page 159
- “DMA Handshaking Interface” on page 159

APB Slave Interface

This suite of tests is run to verify that the APB interface functions correctly by checking the following:

- All non-configuration parameter register reset values are verified.
- All read-only registers are written to with opposite values to verify that they are read only.
- All writable registers are written to with opposite values to verify that they can be written.
- Some registers can be written only when the DW_apb_i2c is disabled. Confirm that those registers are non-writable in that mode. Attempt to write the opposite values to those registers while the DW_apb_i2c is disabled and verify that the writes are ignored.
- The *CNT registers can be written to only if the configuration parameter IC_HC_COUNT_VALUES = 0. Verify that the registers are read-only when IC_HC_COUNT_VALUES = 0 and writable when IC_HC_COUNT_VALUES = 1.
- Confirm that it is not possible to write the transmit buffer threshold level (IC_TX_TL) higher than the size of the transmit buffer. Verify that if a larger value is written that the value becomes set to the size of the transmit buffer (max).
- Confirm that it is not possible to write the receive buffer threshold level (IC_RX_TL) higher than the size of the transmit buffer. Verify that if a larger value is written that the value becomes set to the size of the transmit buffer (max).
- Write illegal value 0 to SPEED bits in IC_CON and verify that the new value is parameter IC_MAX_SPEED_MODE.
- Verify that the SPEED bits in IC_CON cannot be written to higher speeds than configuration parameter IC_MAX_SPEED_MODE.

DW_apb_i2c Master Operation

This suite of tests is run only when the DW_apb_i2c is configured as a master. For instance, these tests go through all combinations of speed, addressing, read/write, and multi-byte transfers. Commands are issued to the DW_apb_i2c, and the I²C Slave is the target and used to verify the transfers. The tests also verify the following:

- SCL low and SCL high times are with I²C specification
- Operation of all registers
- Master arbitration
- Debug outputs
- Disabling of DW_apb_i2c shown correctly on ic_en output
- Programmed count values for all the *CNT registers
- The current source enable output operates correctly
- Combined format operation (7- and 10-bit addressing modes)
- Restart enable and disable
- Clock synchronization by stretching SCL
- Loop-back operation by performing simultaneous master-transmitter, slave-receiver sending multiple bytes. A single-byte transfer with master-receiver, slave-transmitter is also performed

DW_apb_i2c Slave Operation

This suite of tests is run only when the DW_apb_i2c is configured as a slave. Similar to the tests developed for the master, the driving force is the Serial Master BFM. For instance, these tests go through all combinations of speed, addressing, read/write, and multi-byte transfers. The I²C master is used to generate transfers and the DW_apb_i2c is the target; the AHB Master is used to verify the transfers. The tests also verify the following:

- Operation of all registers
- Debug outputs
- Disabling of DW_apb_i2c shown correctly on ic_en output
- Combined format operation (7- and 10-bit addressing modes)

DW_apb_i2c Interrupts

These tests verify that the DW_apb_i2c generates and handles the servicing of interrupts correctly. They also verify operation of the debug ports.

DMA Handshaking Interface

These tests verify that DW_apb_i2c generates and responds through the handshaking interface. Transfers are generated within the DMA BFM and transmitted through the I²C protocol from the DUT to the ALT_DUT and vice versa. Different watermark levels are selected to control the clearing on the dma_tx_req/dma_rx_req lines once an acknowledgement is received. A random number of bytes are transferred using only the handshaking interface.

DW_apb_i2c Dynamic IC_TAR and IC_10BITADDR_MASTER Update

This test is run only if the DW_apb_i2c is configured as a master and the parameter I2C_DYNAMIC_TAR_UPDATE = 1. This test verifies that DW_apb_i2c Master Target address (IC_TAR) and the parameter IC_10BITADDR_MASTER can be updated dynamically while the DW_apb_i2c Slave is involved in an I²C transfer on the I²C bus.

Generate NACK as a Slave-Receiver

This test is always run and tests the functionality of DW_apb_i2c, depending on whether the parameter IC_SLV_DATA_NACK_ONLY is set to 0 or 1. This test verifies that ACK/NACKs are generated correctly when DW_apb_i2c is acting as a slave-receiver, depending on whether IC_SLV_DATA_NACK_ONLY register exists (set by having parameter IC_SLV_DATA_NACK_ONLY=1). If the register exists, its value is set to 1 for the duration of the test. If the register exists (and therefore its value is 1), a NACK is generated by the slave when data is sent to it, the transfer is aborted, and data is not written to the receive buffer. Otherwise, ACKs are generated for the duration of the transfer, the transfer completes successfully, and the data is written to the receive buffer successfully.

SCL Held Low for Duration Specified in IC_SDA_SETUP

This test verifies that during a Slave-Receive I²C transfer, DW_apb_i2c asserts the output port ic_data_oe, holding SCL low for the minimum period specified in the IC_SDA_SETUP register. This only happens every time the I²C master ACKs a data byte, and the transmit FIFO in DW_apb_i2c is not filled to satisfy this read request.

Generate ACK/NACK for General Call

This test verifies that the IC_ACK_GENERAL_CALL bit controls whether DW_apb_i2c ACK or NACKs an I²C general call address.

Overview of DW_apb_i2c Testbench

As illustrated in [Figure 28 on page 161](#), the Verilog DW_apb_i2c testbench includes two instantiations of the design under test (DUT), AHB and APB Bridge bus models, and a Vera shell. The Vera shell consists of a number of serial slave BFM, a master slave BFM, and a DMA BFM to simulate and stimulate traffic to and from the DW_apb_i2c.

The test_DW_apb_i2c.v file shows the instantiation of the top-level MacroCell in a testbench and resides in the *workspace/sim/testbench* directory. The testbench tests the user configuration specified in the Specify Configuration task of coreConsultant. The testbench also tests that the component is AMBA-compliant and includes a self-checking mechanism. When a coreKit has been unpacked and configured, the verification environment is stored in *workspace/sim*. Files in *workspace/sim/test_i2c* form the actual testbench for DW_apb_i2c.

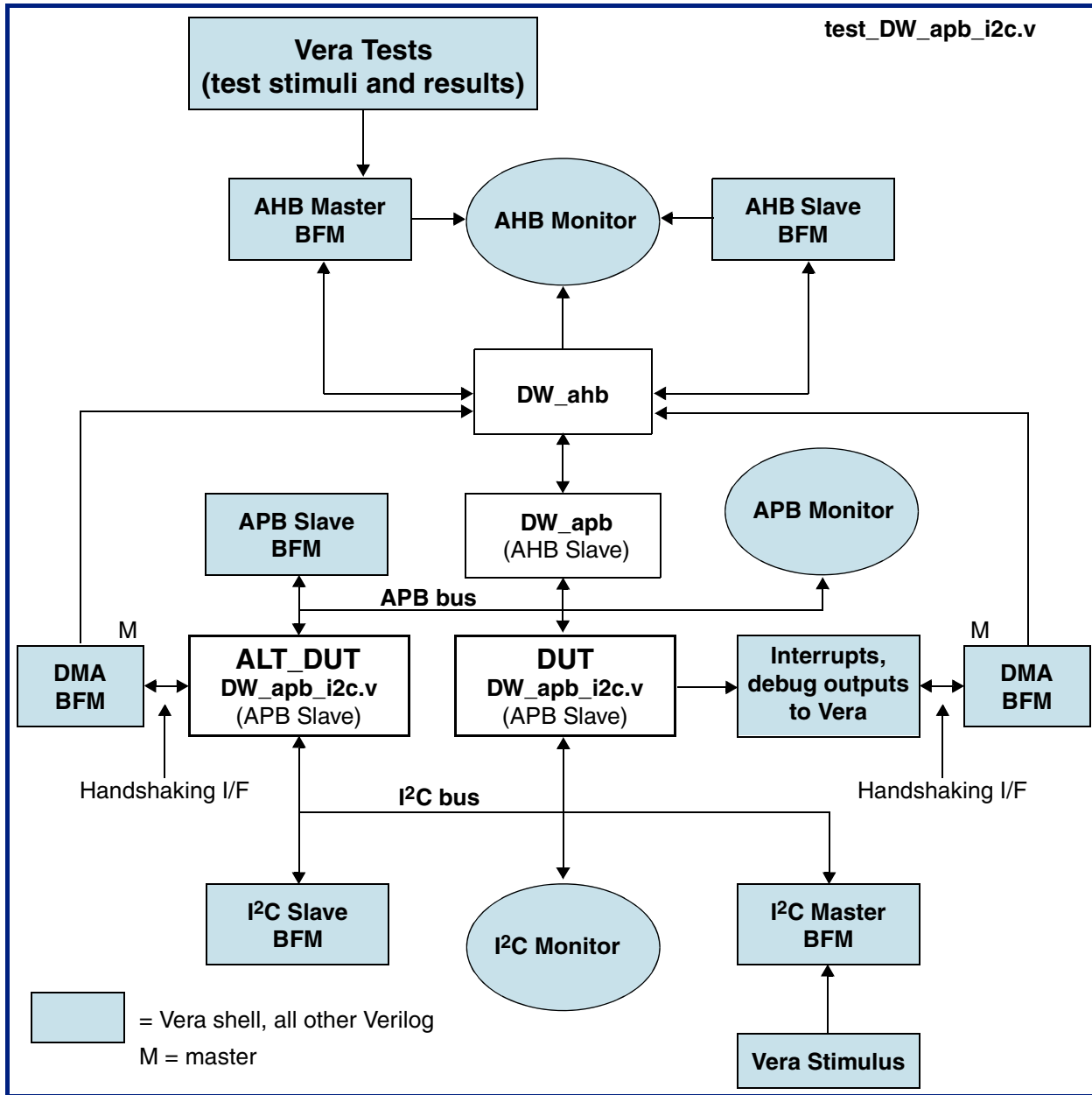


Figure 28: DW_apb_i2c Testbench

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:

- [“Digital/Analog Domain Functional Partitioning” on page 163](#)
- [“Reading and Writing from an APB Slave” on page 164](#)
- [“Write Timing Operation” on page 167](#)
- [“Read Timing Operation” on page 168](#)
- [“Accessing Top-level Constraints” on page 168](#)

Digital/Analog Domain Functional Partitioning

The I²C protocol requires that an I²C device (Digital controller and I/O pad) implement 50nS of spike rejection and include a 300ns SDA hold time. It does not specify where these functional elements should be implemented—in the pad or in the digital controller. In order to meet these two timing requirements, you must ensure that the I/O pads, used in conjunction with DW_apb_i2c, implement the 50ns spike rejection, as well as the 300ns SDA hold time. These are described as follows

1. The 300ns hold requirement in the Philips *I2C-Bus Specification* is related to the SDA relative to the **falling** edge of SCL for I²C receivers.

The receiver inside DW_apb_i2c does not provide the 300ns of hold time outlined in the I²C bus specification, and this hold time functionality should be implemented externally to the DW_apb_i2c, such as the I/O pad.

When DW_apb_i2c is receiving data as:

- a. a slave, the received data is sampled on the **rising** edge of SCL.
- b. a master, the received data is sampled **just prior** to the **falling** edge.

This means that:

- a. DW_apb_i2c, as a slave-receiver, either (1) requires the I²C master to implement the 300ns hold time; OR (2) requires the I/O pad to implement the hold time.
- b. DW_apb_i2c, as a master-receiver, always samples the data **ahead** of the falling edge of SCL, because DW_apb_i2c controls the SCL.

[Appendix D on page 189](#) in this databook contains information to advise customers on how to fix non-Synopsys I²C receivers, which are communicating with DW_apb_i2c acting as an I²C master-transmitter. This means that **holding** the transmitter output from DW_apb_i2c for that 300ns, to emulate the 300ns hold time, can help such non-compliant, non-Synopsys I²C receivers.

2. There is a requirement in the *I²C-Bus Specification* that in FS mode, a I²C device needs to suppress input spikes that are up to 50ns wide in the FS mode (such as, the tSP value from Table 4 of the Philips specification) and up to 10ns in the HS mode.

The I²C standard requires the receiver to take responsibility of this requirement, but does not specify how or where to implement it (for instance, either in the receiver I/O pad or digital control logic).

If ic_clk is 20MHz or less, then DW_apb_i2c filters away the 50ns spike in FS mode, as required by the specification. In HS mode, the width of the spike required to be filtered away is 10ns and DW_apb_i2c achieves 9.49ns with ic_clk at 105.4MHz.

If the previous requirements cannot be met satisfactorily, then you must ensure that such spike rejection be handled externally to DW_apb_i2c, such as in the I/O pad.

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.

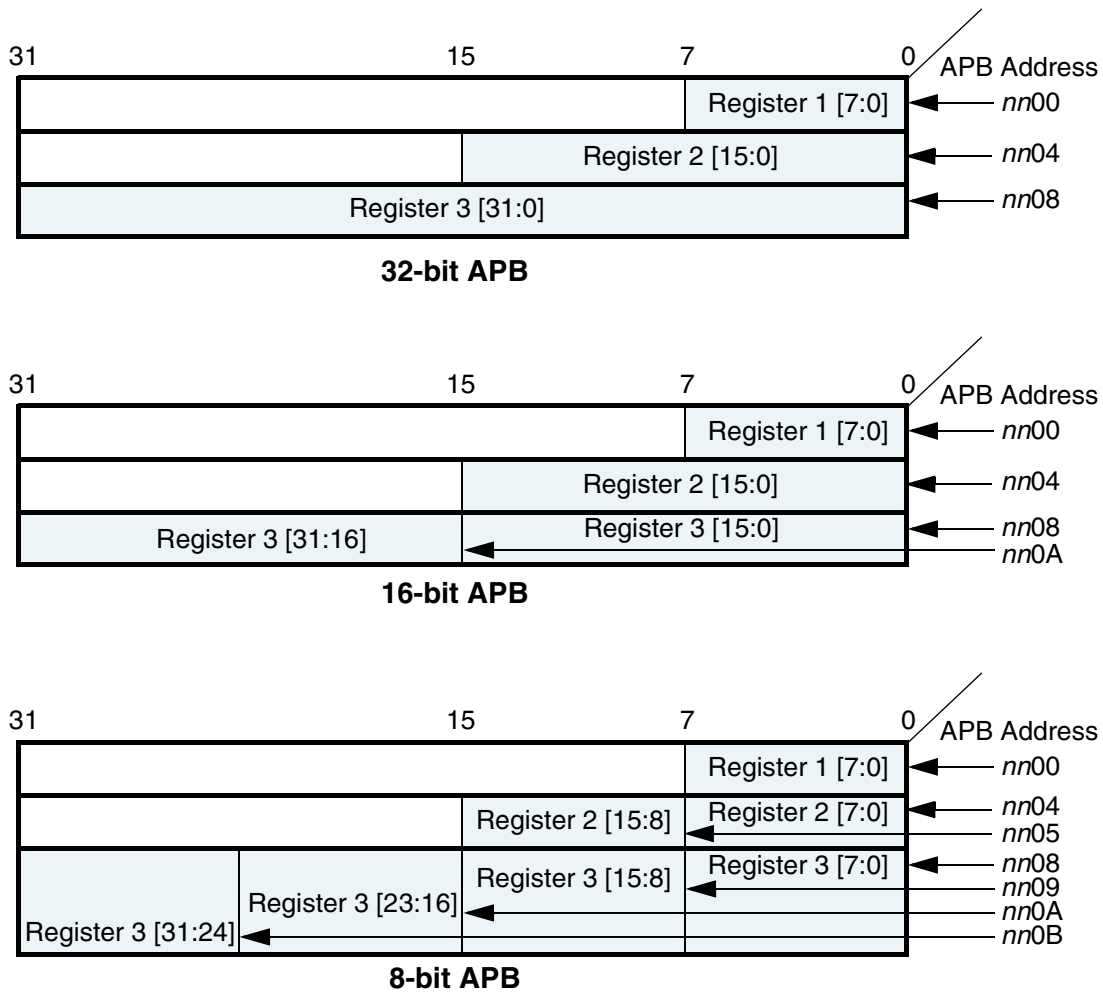


Figure 29: Read/Write Locations for Different APB Bus Data Widths

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.

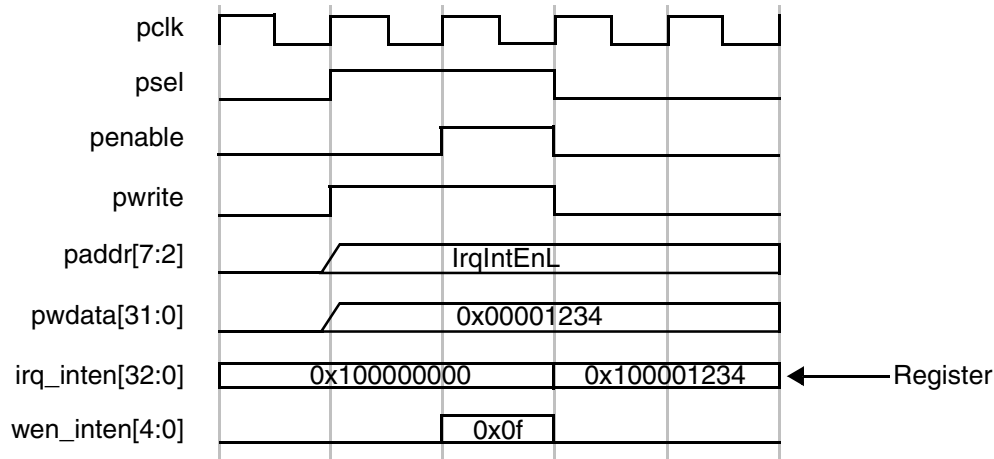


Figure 30: APB Write Transaction

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.

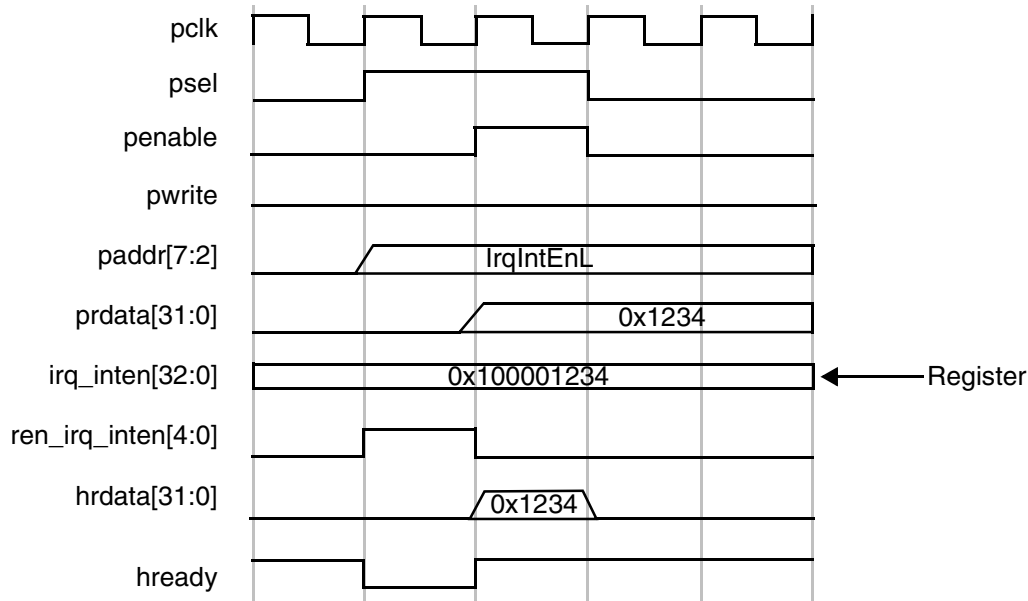


Figure 31: APB Read Transaction

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>
```

A

Building and Verifying Your DW_apb_i2c

This chapter provides an overview of the step-by-step process you use to configure, synthesize, and verify your DW_apb_i2c 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:

- [“Setting Up Your Environment”](#)
- [“Starting coreConsultant” on page 172](#)
- [“Checking Your Environment” on page 173](#)
- [“Configuring the DW_apb_i2c”](#)
- [“Synthesizing the DW_apb_i2c” on page 174](#)
- [“Verifying the DW_apb_i2c” on page 175](#)

If you plan to include the DW_apb_i2c as part of a DesignWare AMBA subsystem, then you will want to use the coreAssembler tool. This tool is a customized version of coreAssembler. For more information about including DW_apb_i2c in a DesignWare AMBA subsystem, refer to [Chapter 2, “Building and Verifying a Subsystem” on page 17](#).

Setting Up Your Environment

DW_apb_i2c 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 SYNOPSIS. 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*.

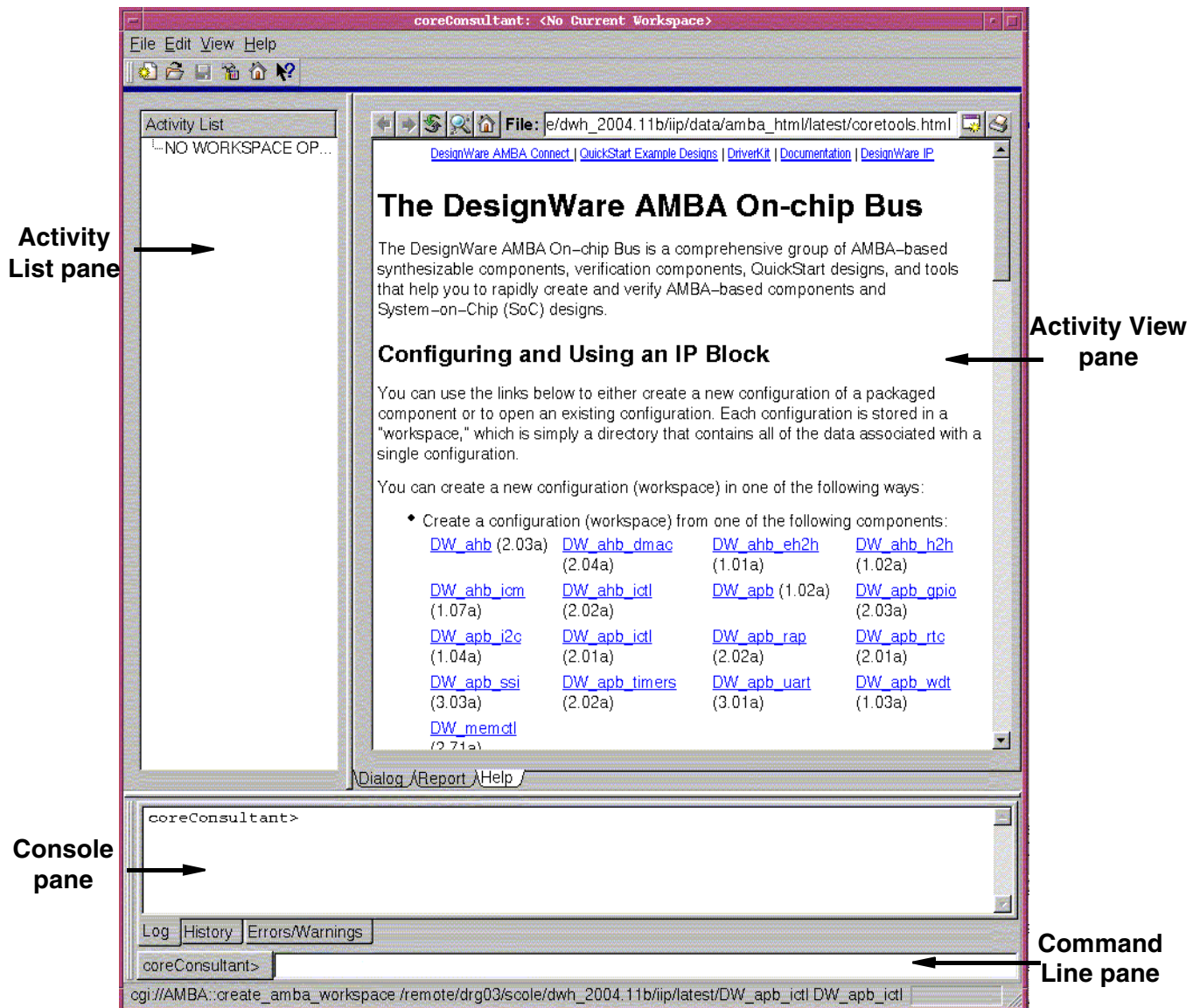
Starting coreConsultant

To invoke coreConsultant:

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

```
% coreConsultant
```

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



3. Click on the DW_apb_i2c 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 enabled 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.

Checking 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.

Configuring the DW_apb_i2c

This section steps you through the tasks in the coreConsultant GUI that configure your core. Complete information about the latest version of coreConsultant is available 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.



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_i2c in coreConsultant, you can then go back through these steps and change values in order to see how they affect the design.

1. **Specify Configuration** – The Specify Configuration activity is where you specify the basic configuration of the DW_apb_i2c. 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 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.

Synthesizing the DW_apb_i2c

The steps to generate a gate-level netlist for a component in coreConsultant are the similar when running synthesis on a subsystem using coreAssembler. To see the procedures for performing synthesis, refer to [“Create Gate-Level Netlist” on page 30](#). For more information about running synthesis in coreConsultant, refer to 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_i2c 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_i2c
- 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 183](#).

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.

Other Synthesis Information

The following are the false paths and timing exceptions for the DW_apb_i2c.

- If clocks are asynchronous, then false paths exist from registers in pclk to ic_clk and vice versa.
- There are false paths defined from ic_rst_n, and from presetn.

Verifying the DW_apb_i2c

This section provides the steps you use to execute the testbench available for DW_apb_i2c verification. Once the DW_apb_i2c 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_i2c verification is detailed in the following sections:

- [“Creating GTECH Simulation Models”](#)
- [“Verify the Simulation Model” on page 177](#)



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 Model activity.

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

Table 13: Parameters for Generate GTECH Model

Field Name	Description
Execution Options	
Generate Scripts only?	<p>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.</p>
Run Style	<p>Values: local, lsf, grd, or remote Default Value: local Description: Describes how to run the command: locally, via lsf, via grd, or through the remote shell.</p>
Run Style Options	<p>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.</p>
Send e-mail	<p>Values: current user's name Description: E-mail is sent when the command script completes or is terminated.</p>
Synthesis Control	
Ungroup Netlist after Compile	<p>Values: Enable or Disable Default Value: Disable Description: Ungroups the design to provide a non-hierarchical netlist</p>

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_i2c
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_i2c.v output file that is written to *workspace/gtech/qmap/db*.

Verify the Simulation Model

To verify DW_apb_i2c, 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 implementation of the design or the gate-level design from another stage of synthesis. To run this, click Formal Verification under the Verify Component activity.
2. **Setup and Run Simulations** – Specify the simulation by completing the Setup and Run Simulations activity:
 - a. In the VIP pane, click on the VMT and AMBA versions to see the available versions; leave these in the default "latest" mode.
 - b. In the Select Simulator area, click on the Simulator view list item to view available simulators (VCS is the default).
 - c. 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_i2c 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 Path	The path to the include directory contained within your MTI simulator installation area. A valid directory includes the file veriuser.h.
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. These files are generated as part of building the testbench. Encrypted Vera source is compiled and stored in the cache.
DW Foundation install area	Path to your Synopsys/DW Foundation installation. This parameter defaults to the value of your SYNOPSIS environment variable. Any change to this value must be made from the Tool Installation Areas coreConsultant dialog box.

Field Name	Description
C Compiler for (Vera PLI)	<p>Values: gcc or cc</p> <p>Default Value: gcc</p> <p>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.</p>

- 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 will create a file that you can use later for debugging the simulation, if you want to do so.

Field Name	Description
Generates waves file	<p>Values: Enable or Disabled</p> <p>Default Value: Disable</p> <p>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.</p>
Depth of waves to be recorded	<p>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.</p>

- f. Choose the View list choice.
- g. 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 (see [page 175](#)) only if you are using a non-VCS simulator and do not have a source license.
- h. Choose the Execution Options list choice to set the following options:

Field Name	Description
Do Not Launch Simulation	<p>Values: Enable or Disable</p> <p>Default Value: Disable</p> <p>Description: Determines whether to execute the simulation or just generate the simulation run script. If enabled, 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.</p>

Field Name	Description
Run Style	<p>Values: local, lsf, grd, or remote</p> <p>Default Value: local</p> <p>Description: Describes how to run the command: locally, via lsf, via grd, or through the remote shell.</p>
Run Style Options	<p>Values: user-defined</p> <p>Default Value: none</p> <p>Description: Additional options for the run style options except local. For remote, specify the hostname. For lsf and grd, specify bsub or qsub commands.</p>
Send e-mail	<p>Values: current user's name</p> <p>Description: E-mail is sent when the command script completes or is terminated.</p>

- i. Select Testbench and look at the options described below:

Field Name	Description
Let each Test decide default Timeout Period	<p>Values: Enable or Disable</p> <p>Default Value: Enable</p> <p>Description: Allows the test to default the timeout period value.</p> <p>NOTE: It is highly recommended that you leave this option enabled if you want the simulation to complete normally.</p>
Number of clocks before simulation timeout	<p>Minimum Value: 1</p> <p>Default Value: 999999</p> <p>Dependencies: This setting is activated when the "Let each Test decide default Timeout Period" is disabled.</p> <p>Description: Enabled if default timeout period not enabled. Enter the number of clock periods of simulation that, if passed, cause the simulation to fail. This is used to avoid runaway simulations or to debug truncated simulation runs.</p> <p>NOTE: If you are experiencing a timeout during the simulation for your specific configuration, you may need to increase this value.</p>
APB Clock Ratio	<p>Values: 1-8 (currently only 1 is allowed)</p> <p>Default Value: 1</p> <p>Description: Specifies the ratio of the APB clock (also known as pclk or the system clock).</p>
Run test_i2c	This is automatically set and runs the specific tests to verify the DW_apb_i2c.

- j. Click Apply to run the simulation.

When you click Apply, coreConsultant performs the following actions:

- Sets up the DW_apb_i2c verification environment to match your selected DW_apb_i2c 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 157](#).

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_i2c bus transactions

This information indicates whether the simulation executed successfully, and lists the DW_apb_i2c 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_i2c 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.

If you are interested, you might want to go back through the process in this chapter and change parameters in order to see how the results vary to the defaults.

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” on page 184](#)
- [“Register Map Files” on page 185](#)
- [“Synthesis Files” on page 186](#)
- [“Verification Reference Files” on page 186](#)

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 14: RTL-Level Files

Files	Encrypted?	Purpose
<code>./src/component_cc_constants.v</code>	No	Includes definitions and values of all configuration parameters that you have specified for the component.
<code>./src/component.v</code>	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: <pre>-y \${SYNOPTSYS}/packages/gtech/src_ver -y \${SYNOPTSYS}/dw/sim_ver</pre> 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 .
<code>./src/component_submodule.v</code>	Yes	Sub-modules of component
<code>./src/component_constants.v</code>	No	Includes the constants used internally in the design.
<code>./src/component.lst</code>	No	Lists the order in which the RTL files should be read into tools, such as simulators or <code>dc_shell</code> . For example, use the following option to read the design into VCS: <pre>vcs -f component.lst</pre>
<code>./src/*.update</code>	Yes	Ignore these files. Used for VHDL generation
<code>./export/component_inst.v</code>	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 15: Simulation Model Files

Files	Encrypted?	Purpose
<code>./gtech/final/db/component.v</code>	No	Simulation model of the component for use with non-Synopsys simulators. A technology-independent, gate-level netlist. VHDL and Verilog versions are generated. When you use this simulation model in your simulation, you must include the DesignWare libraries by using the following options in your simulator invocation: <pre>-y \${SYNOPTSYS}/packages/gtech/src_ver -y \${SYNOPTSYS}/dw/sim_ver</pre> 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 16: Header Files

Files	Encrypted?	Purpose
<code>./c_headers/component_defs.h</code>	No	For use when programming the component in a C environment.
<code>./verilog_headers/component_defs.v</code>	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 17: 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 18: 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](#).

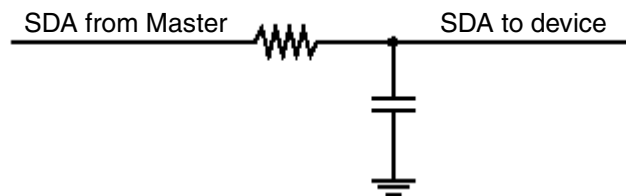
D

DW_apb_i2c Application Notes

The following are application notes for the DW_apb_i2c component.

- The tSDAH (SDA data hold time) detailed in the *I2C-Bus Specification* should be 300-900ns for Fast Mode Devices. However, the SDA data hold time in the DW_apb_i2c component is one-clock cycle based. This tSDAH may be insufficient for some slave devices. A few slave devices may not receive the valid address due to the lack of SDA hold time and will not acknowledge even if the address is valid. If the SDA data hold time is insufficient, an error may occur.

Workaround: If a device needs more SDA data hold time than one clock cycle, an RC delay circuit is needed on the SDA line as illustrated in the following figure:



For example, R=K and C=200pF.

E

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 an “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

A

active command queue
 definition 191

activity
 definition 191

Adding component, to subsystem 22

AHB
 definition 191

AMBA
 definition 191

APB
 definition 191

APB bridge
 definition 191

APB Interface, and DW_apb_i2c 74

APB_DATA_WIDTH 76

application design
 definition 191

arbiter
 definition 191

Arbitration, of master 54

ATPG, with TetraMax 33

B

BFM
 definition 191

big-endian
 definition 191

Block diagram, of DW_apb_i2c 13

blocked command stream
 definition 191

blocking command
 definition 192

Building a subsystem, with coreAssembler 17

bus bridge
 definition 192

C

C header files 185

Check tool environment, in coreAssembler 27

Clock synchronization 55

command channel
 definition 192

command stream
 definition 192

component
 definition 192

Configuration
 of IC_CLK frequency 63

configuration
 definition 192

configuration intent
 definition 192

Configuration parameters 76

Configuring components
 in coreAssembler 28

core
 definition 192

core developer
 definition 192

core integrator
 definition 192

coreAssembler
 building a subsystem 17
 configuring components 28
 creating a batch script 44
 creating gate-level netlist 30
 creating subsystem RTL 29
 definition 192
 formal verification 40
 overview of usage flow 18
 starting 21
 verifying a component 36

coreConsultant
 definition 192
 formal verification 177

coreKit
 definition 192

Creating
 batch script of workspace 44
 gate-level netlist in coreAssembler 30

cycle command
 definition 192

D

dc_shell 30

debug_addr 97

debug_addr_10bit 97

debug_data 97

debug_hs 98

debug_master_act 98

debug_mst_cstate 98

[debug_p_gen](#) 97
[debug_rd](#) 97
[debug_s_gen](#) 97
[debug_slave_act](#) 98
[debug_slv_cstate](#) 98
[debug_wr](#) 98
decoder
 definition 192
design context
 definition 192
design creation
 definition 192
Design for Test, synthesis options 32
Design View
 definition 192
DesignWare AMBA Synthesizable Components
 definition 193
DesignWare cores
 definition 193
DesignWare Library
 definition 193
Disabling DW_apb_i2c
 version 1.06a 62
DMA Controller
 and DW_apb_i2c 65
DMA interface, signals 95
[dma_rx_ack](#) 96
[dma_rx_req](#) 96
[dma_rx_single](#) 96
[dma_tx_ack](#) 96
[dma_tx_req](#) 95
[dma_tx_single](#) 96
dual role device
 definition 193
DW_apb
 slaves
 read timing operation 168
 write timing operation 167
DW_apb_i2c
 block diagram of 13
 functional behavior 45
 functional overview 13
 I/O description 86
 memory map of 100
 operation modes 56
 overview of 45
 parameters 75, 76
 programming of 99
 protocols 50
 registers 104

synthesis
 output files 34, 174
 synthesis of 174
testbench
 overview of 160
 overview of tests 157
Dynamic update of IC_TAR
 initial configuration of master mode 60
 or 10-bit addressing for master mode 61

E

endian
 definition 193
Environment, licenses 14
Exporting, a subsystem 44

F

[fm_shell](#) 30
Formal verification
 in coreAssembler 40
 in coreConsultant 177
FPGA, running synthesis for 32
[fpga_shell](#) 30
Full-Functional Mode
 definition 193
Functional behavior, of DW_apb_i2c 45
Functional overview, of DW_apb_i2c 13

G

Generating
 subsystem RTL 29
GPIO
 definition 193
GTECH
 definition 193
GTECH, generation of 34, 175

H

hard IP
 definition 193
HDL
 definition 193

I

I/O connections 87
I/O signals, description of 86
IC_10BITADDR_MASTER 77, 78

- IC_ACK_GENERAL_CALL 148
 - ic_activity_intr(_n) 94
 - IC_ADD_ENCODED_PARAMS 80
 - IC_CAP_LOADING 82
 - ic_clk 89
 - IC_CLK frequency, configuration of 63
 - ic_clk_in_a 90
 - ic_clk_oe 90
 - IC_CLOCK_FREQ 81
 - IC_CLR_ACTIVITY 133
 - IC_CLR_GEN_CALL 134
 - IC_CLR_INTR 129
 - IC_CLR_RD_REQ 131
 - IC_CLR_RX_DONE 132
 - IC_CLR_RX_OVER 130
 - IC_CLR_RX_UNDER 130
 - IC_CLR_START_DET 134
 - IC_CLR_STOP_DET 133
 - IC_CLR_TX_ABRT 132
 - IC_CLR_TX_OVER 131
 - IC_CON 104
 - ic_current_src_en 91
 - IC_DATA_CMD 112
 - ic_data_in_a 90
 - ic_data_oe 90
 - IC_DEFAULT_SLAVE_ADDR 76, 77
 - IC_DMA_CR 146
 - IC_DMA_RDLR 147
 - IC_DMA_TDLR 147
 - ic_en 90
 - IC_ENABLE 136
 - IC_ENABLE_STATUS 149
 - IC_FS_SCL_HCNT 116
 - IC_FS_SCL_HIGH_COUNT 82
 - IC_FS_SCL_LCNT 118
 - IC_FS_SCL_LOW_COUNT 82
 - ic_gen_call_intr(_n) 95
 - IC_HC_COUNT_VALUES 80
 - IC_HS_MADDR 111
 - IC_HS_MASTER_CODE 77
 - IC_HS_SCL_HCNT 120
 - IC_HS_SCL_HIGH_COUNT 83
 - IC_HS_SCL_LCNT 122
 - IC_HS_SCL_LOW_COUNT 77, 83
 - ic_intr(_n) 91
 - IC_INTR_IO 79
 - IC_INTR_MASK 125
 - IC_INTR_POL 80
 - IC_INTR_STAT 124
 - IC_MASTER_MODE 77
 - IC_MAX_SPEED_MODE 76
 - IC_RAW_INTR_STAT 126
 - ic_rd_req_intr(_n) 93
 - IC_RESTART_EN 79
 - ic_rst_n 89
 - IC_RX_BUFFER_DEPTH 78
 - ic_rx_done_intr(_n) 93
 - ic_rx_over_intr(_n) 91
 - IC_RX_TL 78, 128
 - ic_rx_under_intr(_n) 92
 - IC_RXFLR 140
 - IC_SAR 110
 - IC_SDA_SETUP 148
 - IC_SLV_DATA_NACK_ONLY 145
 - IC_SS_CDNT 115
 - IC_SS_HCNT 113
 - IC_SS_SCL_HIGH_COUNT 81
 - IC_SS_SCL_LOW_COUNT 82
 - ic_start_det_intr(_n) 95
 - IC_STATUS 137
 - ic_stop_det_intr(_n) 94
 - IC_TAR 108
 - ic_tx_abrt_intr(_n) 92
 - IC_TX_ABRT_SOURCE 141
 - IC_TX_BUFFER_DEPTH 78
 - ic_tx_ecmpty_intr(_n) 94
 - ic_tx_over_intr(_n) 92
 - IC_TX_TL 78, 129
 - IC_TXFLR 139
 - IC_USE_COUNTS 80
 - IIP
 - definition 193
 - implementation view
 - definition 193
 - instantiate
 - definition 193
 - interface
 - definition 193
 - Interfaces
 - APB 74
 - DMA Controller 65
 - IP
 - definition 193
- L**
- Licenses 14
 - little-endian
 - definition 193

M

MacroCell
 definition 193
 master
 definition 193
 Master arbitration 54
 Master mode 60
 Memory map, of DW_apb_i2c 100
 model
 definition 193
 monitor
 definition 193

N

non-blocking command
 definition 193

O

Operation modes 56
 Output files
 GTECH 185
 header files 185
 register map 185
 RTL-level 184
 Simulation model 185
 synthesis 186
 verification 186

P

paddr 89
 Parameters
 description of 76
 pclk 88
 penable 88
 peripheral
 definition 194
 prdata 89
 presetn 88
 Programming DW_apb_i2c
 memory map 99
 registers 104
 Protocols, of I²C 50
 psel 88
 psyn_shell 30
 pt_shell 30
 pwwdata 89
 pwrite 89

R

Reading, from unused locations 164
 Register
 IC_HS_MADDR 111
 Registers
 Clear ACTIVITY Interrupt 133
 Clear Combined and Individual Interrupts 129
 Clear GEN_CALL Interrupt 134
 Clear RD_REQ Interrupt 131
 Clear RX_DONE Interrupt 132
 Clear RX_OVER Interrupt 130
 Clear RX_UNDER Interrupt 130
 Clear START_DET Interrupt 134
 Clear STOP_DET Interrupt 133
 Clear TX_ABRT Interrupt 132
 Clear TX_OVER Interrupt 131
 Control 104
 DMA Control 146
 DMA Transmit Data Level 147
 Enable Status 149
 Fast Speed I2C Clock SCL High Count 116
 Fast Speed I2C Clock SCL Low Count 118
 Generate Slave Data NACK 145
 High Speed I2C Clock SCL High Count 120
 High Speed I2C Clock SCL Low Count 122
 HS Master Mode Code Address 111
 I2C Enable 136
 I2C Receive Data Level 147
 IC_ACK_GENERAL_CALL 148
 IC_CLR_ACTIVITY 133
 IC_CLR_GEN_CALL 134
 IC_CLR_INTR 129
 IC_CLR_RD_REQ 131
 IC_CLR_RX_DONE 132
 IC_CLR_RX_OVER 130
 IC_CLR_RX_UNDER 130
 IC_CLR_START_DET 134
 IC_CLR_STOP_DET 133
 IC_CLR_TX_ABRT 132
 IC_CLR_TX_OVER 131
 IC_CON 104
 IC_DATA_CMD 112
 IC_DMA_CR 146
 IC_DMA_RDLR 147
 IC_DMA_TDLR 147
 IC_ENABLE 136
 IC_ENABLE_STATUS 149
 IC_FS_SCL_HCNT 116
 IC_FS_SCL_LCNT 118
 IC_HS_SCL_HCNT 120
 IC_HS_SCL_LCNT 122
 IC_INTR_MASK 125

IC_INTR_STAT [124](#)
 IC_RAW_INTR_STAT [126](#)
 IC_RX_TL [128](#)
 IC_RXFLR [140](#)
 IC_SAR [110](#)
 IC_SDA_SETUP [148](#)
 IC_SLV_DATA_NACK_ONLY [145](#)
 IC_SS_HCNT [113](#)
 IC_SS_LCNT [115](#)
 IC_STATUS [137](#)
 IC_TAR [108](#)
 IC_TX_ABRT_SOURCE [141](#)
 IC_TX_TL [129](#)
 IC_TXFLR [139](#)
 Interrupt Mask [125](#)
 Interrupt Status [126](#)
 of DW_apb_i2c [104](#)
 Raw Interrupt Status [124](#)
 Receive Buffer Threshold [128](#)
 Rx/Tx Data buffer and Command [112](#)
 SDA Setup [148](#)
 Slave Address [110](#)
 Standard Speed I2C Clock SCL High Count [113](#)
 Standard Speed I2C Clock SCL Low Count [115](#)
 Target Address [108](#)
 Transmit Buffer Threshold [129](#)

RTL
 definition [194](#)
 run.scr [34](#), [174](#)

S

SDRAM
 definition [194](#)
 SDRAM controller
 definition [194](#)
 Signals, description of [86](#)
 Simulation
 generating GTECH models [34](#), [175](#)
 of a component [36](#)
 of a subsystem [40](#)
 of DW_apb_i2c [160](#)
 results [39](#), [43](#), [180](#)
 status [39](#), [43](#), [180](#)
 slave
 definition [194](#)
 Slave mode [56](#)
 SoC
 definition [194](#)
 SoC Platform
 AHB contained in [11](#)
 APB, contained in [11](#)
 defined [11](#)

soft IP
 definition [194](#)
 SSI_HAS_DMA [79](#)
 Starting
 coreAssembler [21](#)
 static controller
 definition [194](#)
 subsystem
 definition [194](#)
 Synthesis
 of DW_apb_i2c [174](#)
 output files [34](#), [174](#)
 results [34](#), [174](#)
 running from command line [175](#)
 target technology, specifying [31](#), [173](#)
 synthesis intent
 definition [194](#)
 synthesizable IP
 definition [194](#)
 Synthesizing subsystem [30](#)

T

Target technology, specifying [31](#), [173](#)
 technology-independent
 definition [194](#)
 Test vectors, generating [33](#)
 test_DW_apb_i2c.v [160](#)
 Testsuite Regression Environment (TRE)
 definition [194](#)
 Timing
 read operation of DW_apb slave [168](#)
 write operation of DW_apb slave [167](#)
 TRE
 definition [194](#)

U

USE_FOUNDATION [76](#), [79](#)

V

Vera, overview of tests [157](#)
 Verification
 and Vera tests [157](#)
 generating GTECH models [34](#), [175](#)
 of a component [36](#)
 of a subsystem [40](#)
 of DW_apb_i2c [160](#)
 Verilog header files [185](#)
 VIP
 definition [194](#)

W

workspace
 definition [194](#)
wrap
 definition [194](#)
wrapper
 definition [194](#)

Z

zero-cycle command
 definition [194](#)