



DesignWare DW_apb_i2c Databook

*DW_apb_i2c – **Product Code***

Copyright Notice and Proprietary Information Notice

© 2014 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Destination Control Statement

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

Disclaimer

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

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at

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

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

Synopsys, Inc.
700 E. Middlefield Road
Mountain View, CA 94043

www.synopsys.com

Contents

Preface	7
Revision History	11
Chapter 1	
Product Overview	15
1.1 DesignWare System Overview	15
1.2 General Product Description	17
1.2.1 DW_apb_i2c Block Diagram	17
1.3 Features	18
1.3.1 I ² C Features	18
1.3.2 DesignWare APB Slave Interface	18
1.4 Standards Compliance	19
1.5 Verification Environment Overview	19
1.6 Licenses	19
Chapter 2	
Building and Verifying a Component or Subsystem	21
2.1 Setting up Your Environment	21
2.2 Overview of the coreConsultant Configuration and Integration Process	22
2.2.1 coreConsultant Usage	22
2.2.2 Configuring the DW_apb_i2c within coreConsultant	23
2.2.3 Creating Gate-Level Netlists within coreConsultant	24
2.2.4 Verifying the DW_apb_i2c within coreConsultant	24
2.2.5 Running Leda on Generated Code with coreConsultant	24
2.3 Overview of the coreAssembler Configuration and Integration Process	25
2.3.1 coreAssembler Usage	25
2.3.2 Configuring the DW_apb_i2c within a Subsystem	28
2.3.3 Creating Gate-Level Netlists within coreAssembler	29
2.3.4 Verifying the DW_apb_i2c within coreAssembler	29
2.3.5 Running Leda on Generated Code with coreAssembler	29
2.4 Database Files	29
2.4.1 Design/HDL Files	29
2.4.2 Synthesis Files	30
2.4.3 Verification Reference Files	31
Chapter 3	
Functional Description	33
3.1 Overview	33
3.2 I ² C Terminology	35

3.2.1 I ² C Bus Terms	35
3.2.2 Bus Transfer Terms	36
3.3 I ² C Behavior	36
3.3.1 START and STOP Generation	37
3.3.2 Combined Formats	38
3.4 I ² C Protocols	38
3.4.1 START and STOP Conditions	38
3.4.2 Addressing Slave Protocol	39
3.4.3 Transmitting and Receiving Protocol	40
3.4.4 START BYTE Transfer Protocol	41
3.5 Tx FIFO Management and START, STOP and RESTART Generation	42
3.5.1 Tx FIFO Management When IC_EMPTYFIFO_HOLD_MASTER_EN = 0	42
3.5.2 Tx FIFO Management When IC_EMPTYFIFO_HOLD_MASTER_EN = 1	43
3.6 Multiple Master Arbitration	47
3.7 Clock Synchronization	49
3.8 Operation Modes	49
3.8.1 Slave Mode Operation	50
3.8.2 Master Mode Operation	53
3.8.3 Disabling DW_apb_i2c	56
3.8.4 Aborting I2C Transfers	57
3.9 Spike Suppression	57
3.10 Fast Mode Plus Operation	59
3.11 IC_CLK Frequency Configuration	59
3.11.1 Minimum High and Low Counts	59
3.11.2 Minimum IC_CLK Frequency	61
3.12 SDA Hold Time	64
3.12.1 SDA Hold Timings in Receiver	65
3.12.2 SDA Hold Timings in Transmitter	66
3.13 DMA Controller Interface	67
3.13.1 Enabling the DMA Controller Interface	68
3.13.2 Overview of Operation	68
3.13.3 Transmit Watermark Level and Transmit FIFO Underflow	69
3.13.4 Choosing the Transmit Watermark Level	69
3.13.5 Selecting DEST_MSIZ and Transmit FIFO Overflow	71
3.13.6 Receive Watermark Level and Receive FIFO Overflow	71
3.13.7 Choosing the Receive Watermark level	72
3.13.8 Selecting SRC_MSIZ and Receive FIFO Underflow	72
3.13.9 Handshaking Interface Operation	72
3.14 APB Interface	76
Chapter 4	
Parameters	77
4.1 Parameter Descriptions	77
4.2 Configuration Parameters	77
Chapter 5	
Signals	91
5.1 DW_apb_i2c Interface Diagram	91
5.2 I/O Connections	92

5.3 DW_apb_i2c Signal Descriptions	93
Chapter 6	
Registers	105
6.1 Register Memory Map	105
6.2 Operation of Interrupt Registers	111
6.3 Registers and Field Descriptions	112
6.3.1 IC_CON	112
6.3.2 IC_TAR	117
6.3.3 IC_SAR	119
6.3.4 IC_HS_MADDR	120
6.3.5 IC_DATA_CMD	121
6.3.6 IC_SS_SCL_HCNT	124
6.3.7 IC_SS_SCL_LCNT	125
6.3.8 IC_FS_SCL_HCNT	126
6.3.9 IC_FS_SCL_LCNT	127
6.3.10 IC_HS_SCL_HCNT	128
6.3.11 IC_HS_SCL_LCNT	129
6.3.12 IC_INTR_STAT	130
6.3.13 IC_INTR_MASK	131
6.3.14 IC_RAW_INTR_STAT	133
6.3.15 IC_RX_TL	137
6.3.16 IC_TX_TL	138
6.3.17 IC_CLR_INTR	139
6.3.18 IC_CLR_RX_UNDER	139
6.3.19 IC_CLR_RX_OVER	140
6.3.20 IC_CLR_TX_OVER	140
6.3.21 IC_CLR_RD_REQ	141
6.3.22 IC_CLR_TX_ABRT	141
6.3.23 IC_CLR_RX_DONE	142
6.3.24 IC_CLR_ACTIVITY	142
6.3.25 IC_CLR_STOP_DET	143
6.3.26 IC_CLR_START_DET	143
6.3.27 IC_CLR_GEN_CALL	144
6.3.28 IC_ENABLE	145
6.3.29 IC_STATUS	147
6.3.30 IC_TXFLR	150
6.3.31 IC_RXFLR	151
6.3.32 IC_SDA_HOLD	152
6.3.33 IC_TX_ABRT_SOURCE	153
6.3.34 IC_SLV_DATA_NACK_ONLY	157
6.3.35 IC_DMA_CR	158
6.3.36 IC_DMA_TDLR	159
6.3.37 IC_DMA_RDLR	160
6.3.38 IC_SDA_SETUP	161
6.3.39 IC_ACK_GENERAL_CALL	162
6.3.40 IC_ENABLE_STATUS	163
6.3.41 IC_FS_SPKLEN	165
6.3.42 IC_HS_SPKLEN	166

6.3.43 IC_CLR_RESTART_DET	167
6.3.44 IC_COMP_PARAM_1	168
6.3.45 IC_COMP_VERSION	170
6.3.46 IC_COMP_TYPE	170
Chapter 7	
Programming the DW_apb_i2c	171
7.1 Software Registers	171
7.2 Software Drivers	171
7.3 Programming Example	172
Chapter 8	
Verification	179
8.1 Overview of Vera Tests	179
8.1.1 APB Slave Interface	179
8.1.2 DW_apb_i2c Master Operation	180
8.1.3 DW_apb_i2c Slave Operation	180
8.1.4 DW_apb_i2c Interrupts	181
8.1.5 DMA Handshaking Interface	181
8.1.6 DW_apb_i2c Dynamic IC_TAR and IC_10BITADDR_MASTER Update	181
8.1.7 Generate NACK as a Slave-Receiver	181
8.1.8 SCL Held Low for Duration Specified in IC_SDA_SETUP	181
8.1.9 Generate ACK/NACK for General Call	181
8.2 Overview of DW_apb_i2c Testbench	182
Chapter 9	
Integration Considerations	185
9.1 Reading and Writing from an APB Slave	185
9.1.1 Reading From Unused Locations	185
9.1.2 32-bit Bus System	186
9.1.3 16-bit Bus System	187
9.1.4 8-bit Bus System	187
9.2 Write Timing Operation	188
9.3 Read Timing Operation	189
9.4 Accessing Top-level Constraints	189
9.5 Performing	190
9.5.1 Area	190
9.5.2 Power Consumption	191
Appendix A	
Glossary	193
Index	197

Preface

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.

Organization

The chapters of this databook are organized as follows:

- Chapter 1, [“Product Overview”](#) provides a system overview, a component block diagram, basic features, and an overview of the verification environment.
- Chapter 2, [“Building and Verifying a Component or Subsystem”](#) introduces you to using the DW_apb_i2c within the coreAssembler and coreConsultant tools.
- 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, [“Glossary”](#) provides a glossary of general terms.

Related Documentation

- [DW_apb_i2c Driver Kit User Guide](#) – Contains information on the Driver Kit for the DW_apb_i2c; requires source code license (DWC-APB-Periph-Source)
- [Using DesignWare Library IP in coreAssembler](#) – Contains information on getting started with using DesignWare SIP components for AMBA 2 and AMBA 3 AXI components within coreTools
- [coreAssembler User Guide](#) – Contains information on using coreAssembler
- [coreConsultant User Guide](#) – Contains information on using coreConsultant

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

Web Resources

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

Customer Support

To obtain support for your product:

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

Check all the boxes in the dialog box that apply to your issue. This menu entry gathers all the Synopsys product data needed to begin debugging an issue and writes it to the file `<core tool startup directory>/debug.tar.gz`.
 - For simulation issues outside of coreConsultant or coreAssembler:
 - Create a waveforms file (such as VPD or VCD)
 - Identify the hierarchy path to the DesignWare instance
 - Identify the timestamp of any signals or locations in the waveforms that are not understood
- Then, contact Support Center, with a description of your question and supplying the above information, using one of the following methods:
 - *For fastest response*, use the SolvNet website. If you fill in your information as explained below, your issue is automatically routed to a support engineer who is experienced with your product. The **Sub Product** entry is critical for correct routing.

Go to <http://solvnet.synopsys.com/EnterACall> and click on the link to enter a call. Provide the requested information, including:
 - **Product:** DesignWare Library IP
 - **Sub Product:** AMBA
 - **Tool Version:** `<product version number>`
 - **Problem Type:**
 - **Priority:**
 - **Title:** DW_apb_i2c
 - **Description:** For simulation issues, include the timestamp of any signals or locations in waveforms that are not understood

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

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

Product Code

[Table 1-1](#) lists all the components associated with the product code for DesignWare APB Advanced Peripherals.

Table 1-1 DesignWare APB Advanced Peripherals – Product Code: 3772-0

Component Name	Description
DW_apb_i2c	A highly configurable, programmable master or slave i2c device with an APB slave interface
DW_apb_i2s	A configurable master or slave device for the three-wire interface (I2S) for streaming stereo audio between devices
DW_apb_ssi	A configurable, programmable, full-duplex, master or slave synchronous serial interface
DW_apb_uart	A programmable and configurable Universal Asynchronous Receiver/Transmitter (UART) for the AMBA 2 APB bus

Revision History

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

Version	Date	Description
1.22a	June 2014	Added: <ul style="list-style-type: none"> ■ New features: <ul style="list-style-type: none"> - Blocking the Tx FIFO commands using IC_TX_CMD_BLOCK field in IC_ENABLE register - Indication for first data byte received after the address in IC_DATA_CMD register - Detection of STOP interrupt only if master is active ■ coreConsultant parameter (IC_AVOID_RX_FIFO_FLUSH_ON_TX_ABORT) introduced to avoid flushing of RX FIFO during TX Abort ■ New bits in IC_STATUS register for Indicating a reason for bus holding ■ Performance section in Integration considerations Updated: <ul style="list-style-type: none"> ■ Width of TX_FLUSH_CNT field in the IC_TX_ABORT_SOURCE register ■ Default Input/Output Delays in Signals chapter
1.21a	May 2013	Added: <ul style="list-style-type: none"> ■ Section on Fast Mode Plus ■ Configuration Parameters: <ul style="list-style-type: none"> - IC_RX_FULL_HLD_BUS_EN - IC_SLV_RESTART_DET_EN ■ Signals: <ul style="list-style-type: none"> - ic_restart_det_intr(_n) signal to enable restart detect in slave mode ■ Registers <ul style="list-style-type: none"> - RESTART_DET bit of IC_INTR_STAT, IC_INTR_MASK and IC_RAW_INTR_STAT registers Bit detects a repeated start when the DW_apb_i2c is the addressed slave - IC_CLR_RESTART_DET to clear the RESTART_DET interrupt - MST_ON_HOLD bit to the IC_INTR_STAT, IC_INTR_MASK and IC_RAW_INTR_STAT registers. This bit indicates whether a master is holding the bus and the Tx FIFO is empty. Added the signal ic_mst_on_hold_intr(_n) ■ Programming flow for DW_apb_i2c master with TAR update

(Continued)

Version	Date	Description
1.21a <i>Cont'd</i>	May 2013 <i>Cont'd</i>	<p><i>Continued</i></p> <p>Updated:</p> <ul style="list-style-type: none"> ■ References to Fast Mode Plus ■ Registers: <ul style="list-style-type: none"> - TX_FLUSH_CNT field of the IC_TX_ABRT_SOURCE register - TX_ABRT field of the IC_RAW_INTR_STAT register - IC_CON - IC_RAW_INTR_STAT - IC_SDA_HOLD <p>Signals:</p> <ul style="list-style-type: none"> ■ Active state of the ic_current_src_en signal ■ Programming flow for DW_apb_i2c as master in standard or fast mode ■ Method for deriving ic_clk values in high-speed modes ■ Documentation template <p>Removed:</p> <ul style="list-style-type: none"> ■ Text stating that Fast Mode Plus is not supported ■ Note in the IC_TX_ABRT_SOURCE register description stating DW_apb_i2c can be a master and slave at the same time
1.20a	Oct 2012	Added the product code on the cover and in Table 1-1.
1.20a	June 2012	Edited calculations for driving SDA in “High-Speed Modes” section; updated IC_ENABLE and IC_TX_ABRT_SOURCE registers.
1.17a	Mar 2012	Enhanced DW_ahb_dmac and DW_apb_i2c programming example; updated definition of IC_FS_SPKLEN and IC_HS_SPKLEN register descriptions; corrected programming values for dma_tx_req and dma_rx_req signals.
1.16b	Dec 2011	Enhanced description of IC_ADD_ENCODED_PARAMS parameter.
1.16b	Nov 2011	Version change for 2011.11a release.
1.16a	Oct 2011	Version change for 2011.10a release.
1.15a	14 June 2011	Removed “Digital/Analog Domain Functional Partitioning” section (9.1) – irrelevant now with Spike Suppression functionality.
1.15a	June 2011	Updated system diagram in Figure 1-1; enhanced description of ic_rst_n signal; enhanced “Related Documents” section in Preface.
1.15a	21 Apr 2011	Clarified description of C_DEFAULT_SDA_HOLD parameter.
1.15a	12 Apr 2011	Corrected IC_DEFAULT_FS_SPKLEN and IC_DEFAULT_HS_SPKLEN default values.
1.15a	Apr 2011	Added spike suppression material; corrected R/W locations in timing diagrams in “Tx FIFO Management and START, STOP and RESTART Generation” section
1.14a	Dec 2010	Corrected subsection numbering in Registers chapter.

(Continued)

Version	Date	Description
1.13a	Oct 2010	Added information on calculating maximum value for IC_DEFAULT_SDA_HOLD parameter and IC_SDA_HOLD register; “SDA Hold Time” section, description of IC_DEFAULT_SDA_HOLD parameter, and IC_SDA_HOLD register updated
1.12a	7 Sep 2010	Corrected DW_ahb_dmac response in “Receive Watermark Level and Receive FIFO Overflow” section
1.12a	Sep 2010	Corrected names of include files and vcs command used for simulation
1.11a	Mar 2010	Corrected information regarding how DW_apb_i2c communicates with slaves when operating in master mode; corrected default value for IC_DEFAULT_SDA_SETUP parameter; added SDA hold time information; added IC_SDA_HOLD register description; removed references to 300ns hold time in integration considerations; removed DW_apb_i2c Application Notes appendix.
1.10a	Jan 2010	Removed reference to I2C protocol created by Philips (NXP).
1.10a	Dec 2009	Corrected dependencies for IC_SS_SCL_HIGH_COUNT, IC_SS_SCL_LOW_COUNT, IC_FS_SCL_HIGH_COUNT, and IC_FS_SCL_LOW_COUNT parameters; corrected IC_RESTART_EN parameter description; modified description of IC_SDA_SETUP register; updated databook to new template for consistency with other IIP/VIP/PHY databooks.
1.10a	Jul 2009	Corrected equations for avoiding underflow when programming a source burst transaction.
1.10a	Jun 2009	Corrected name of IC_10BITADDR_SLAVE parameter in “Parameters” chapter.
1.10a	May 2009	Removed references to QuickStarts, as they are no longer supported.
1.10a	24 Apr 2009	Enhanced IC_CON description with table for IC_SLAVE_DISABLE and MASTER_MODE combinations that result in configuration errors.
1.10a	23 Apr 2009	Enhanced “Master Transmit and Master Receive” subsection to clarify reads for multiple bytes.
1.10a	Oct 2008	IC_RX_FULL_GEN_NACK parameter removed; IC_INTR_MASK is active low; dependency changed for IC_HS_MASTER_CODE parameter; IC_SLAVE_DISABLE default changed to 1; values for HS mode corrected in Table 8; debug_* signal default values corrected; version change for 2008.10a release.
1.09a	Jul 2008	Removed IC_RX_FULL_GEN_NACK configuration parameter and its conditional text. Changed reference to non-existent table for IC_*S_SCL_*CNT registers to link to “IC_CLK Frequency Configuration” section. Removed USE_FOUNDATION parameter.
1.09a	Jun 2008	Removed Synchronous value from IC_CLK_TYPE parameter; clarified that putting data into the FIFO generates a START and emptying the FIFO generates a STOP; clarified description of I2C_DYNAMIC_TAR_UPDATE parameter; clarification of IC_TAR description.
1.08b	11 Feb 2008	Modified note on restriction; page 47.

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 Synthesizable Components.

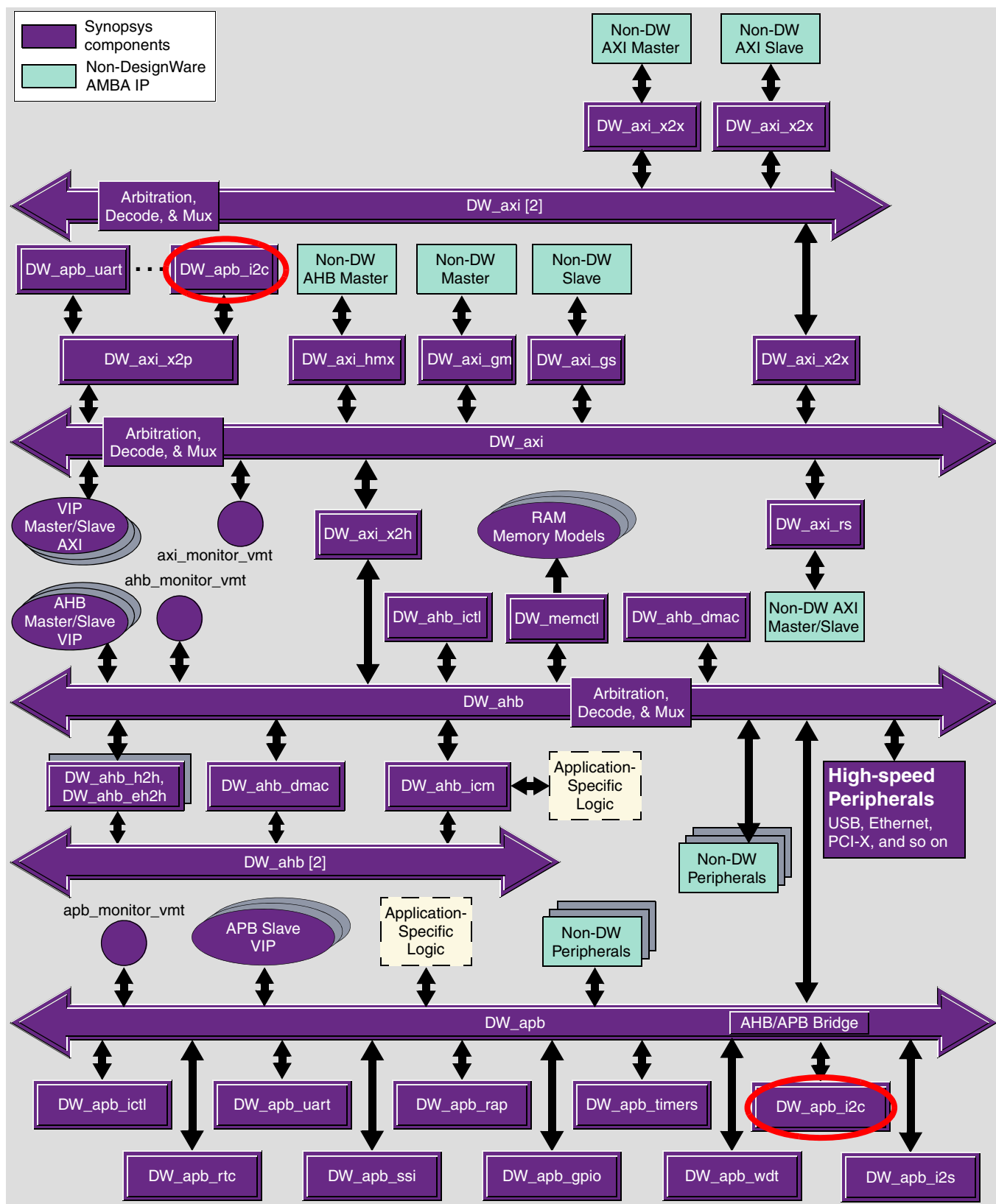
1.1 DesignWare System Overview

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

[Figure 1-1](#) illustrates one example of this environment, including the AXI bus, the AHB bus, and the APB bus. Included in this subsystem are synthesizable IP for AXI/AHB/APB peripherals, bus bridges, and an AXI interconnect and AHB bus fabric. Also included are verification IP for AXI/AHB/APB master/slave models and bus monitors. In order to display the databook for a DW_* component, click on the corresponding component object in the illustration.

**Attention**

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

Figure 1-1 Example of DW_apb_i2c in a Complete System

You can connect, configure, synthesize, and verify the DW_apb_i2c within a DesignWare subsystem using coreAssembler, documentation for which is available on the web in the [coreAssembler User Guide](#).

If you want to configure, synthesize, and verify a single component such as the DW_apb_i2c component, you might prefer to use coreConsultant, documentation for which is available in the [coreConsultant User Guide](#).

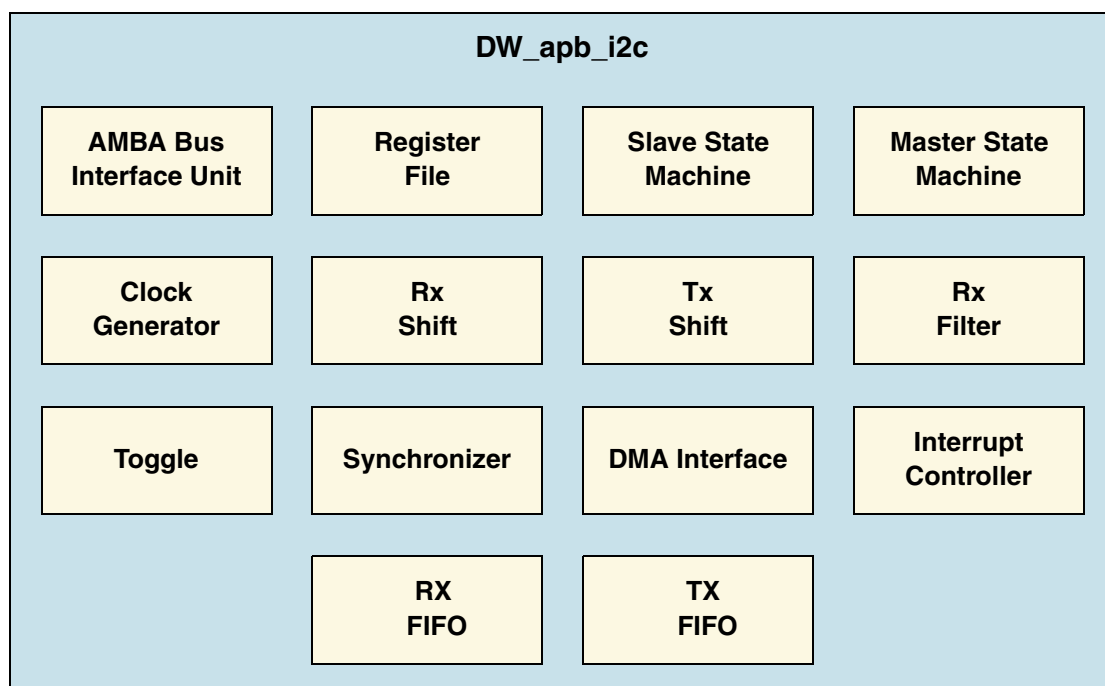
1.2 General Product Description

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.

1.2.1 DW_apb_i2c Block Diagram

[Figure 1-2](#) illustrates a simple block diagram of DW_apb_i2c. For a more detailed block diagram and description of the component, refer to “[Functional Description](#)” on page 33.

Figure 1-2 Block Diagram of DW_apb_i2c



1.3 Features

DW_apb_i2c has the following features:

1.3.1 I²C Features

- Two-wire I²C serial interface – consists of a serial data line (SDA) and a serial clock (SCL)
- Three speeds:
 - Standard mode (0 to 100 Kb/s)
 - Fast mode (≤ 400 Kb/s) or fast mode plus (≤ 1000 Kb/s)¹
 - 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
- Programmable SDA hold time (tHD;DAT)

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

1.3.2 DesignWare 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.

1. In this document, references to fast mode also apply to fast mode plus, unless specifically stated otherwise.

1.4 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. The DW_apb_i2c was designed for [The I 2C-Bus Specification, Version 2.1](#), dated January 2000.

1.5 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 179 chapter discusses the specific procedures for verifying the DW_apb_i2c.

1.6 Licenses

Before you begin using the DW_apb_i2c, you must have a valid license. For more information, refer to the [DesignWare Synthesizable Components for AMBA 2/AMBA 3 AXI Installation Guide](#).

2

Building and Verifying a Component or Subsystem

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

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

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

**Hint**

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

2.1 Setting up Your Environment

The DW_apb_i2c is included in a release of DesignWare SIP components. It is assumed that you have already downloaded and installed the release. If you have not, you can download and install the latest versions of required tools using the [DesignWare Synthesizable Components for AMBA 2/AMBA 3 AXI Installation Guide](#).

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

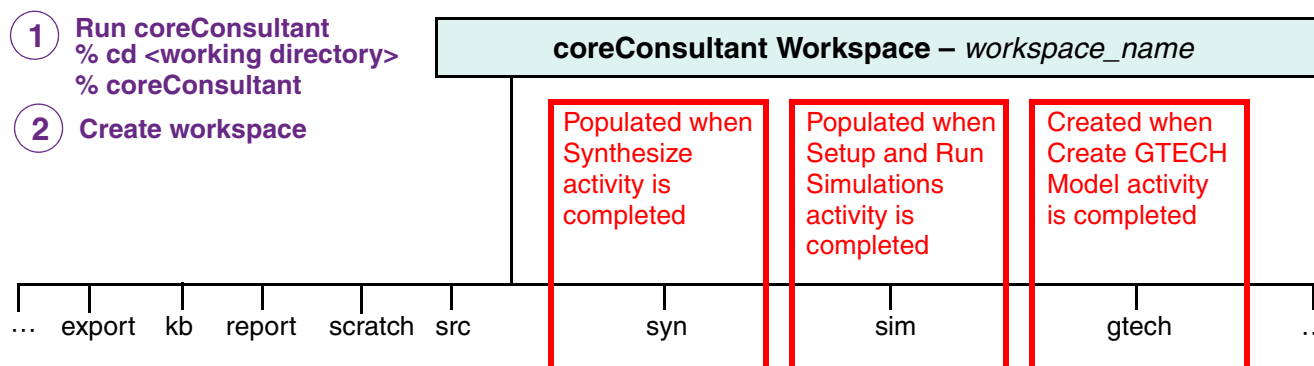
2.2 Overview of the coreConsultant Configuration and Integration Process

Once you have correctly downloaded and installed a release of DesignWare SIP components and then set up your environment, you can begin work on the DW_apb_i2c using coreConsultant.

2.2.1 coreConsultant Usage

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

Figure 2-1 coreConsultant Usage Flow



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

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

Table 2-1 coreConsultant Implementation Workspace Directory Contents

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

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

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

For details on some key files created during coreConsultant activities, refer to [“Database Files”](#) on page 29.

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

2.2.2 Configuring the DW_apb_i2c within coreConsultant

The [“Parameters”](#) chapter on [page 77](#) describes the DW_apb_i2c hardware configuration parameters that you configure using the coreConsultant GUI.

The [“Creating the RTL View of a Core”](#) chapter in the [coreConsultant User Guide](#) discusses how to specify a configuration for an individual component like the DW_apb_i2c.

2.2.3 Creating Gate-Level Netlists within coreConsultant

The “Creating the Gate-Level Netlist for a Core” chapter in the *coreConsultant User Guide* discusses how to create a translation of the RTL view into a technology-specific netlist for an individual component like the DW_apb_i2c.

2.2.4 Verifying the DW_apb_i2c within coreConsultant

The “[Verification](#)” chapter on [page 179](#) provides an overview of the testbench available for DW_apb_i2c verification using the coreConsultant GUI.

The “Verifying Your Implementation” chapter in the *coreConsultant User Guide* discusses how to simulate an individual component like the DW_apb_i2c.

2.2.5 Running Leda on Generated Code with coreConsultant

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

2.3 Overview of the coreAssembler Configuration and Integration Process

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

2.3.1 coreAssembler Usage

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

Figure 2-2 coreAssembler Usage Flow

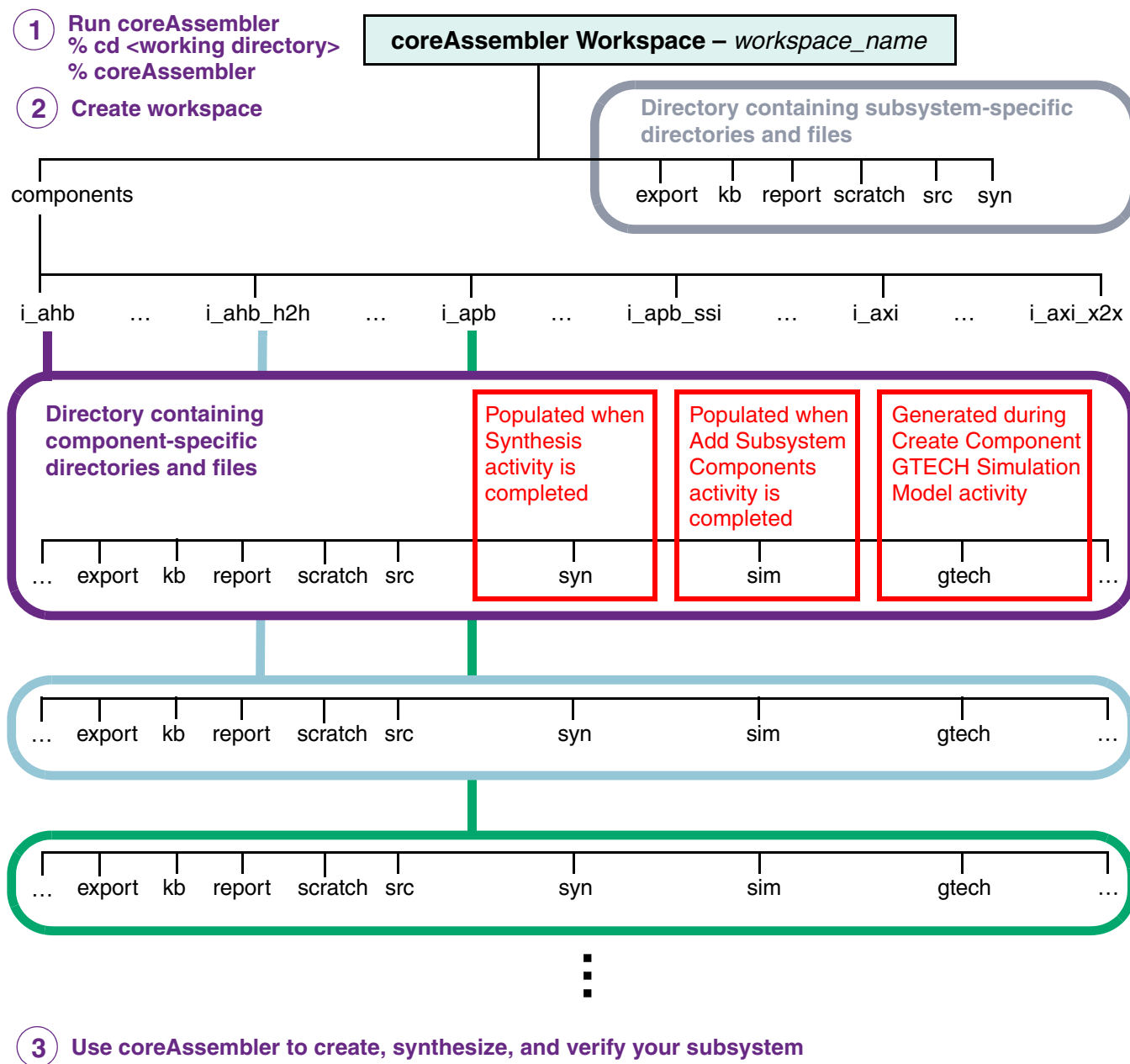


Table 2-2 provides a description of the implementation workspace directory and subdirectories.

Table 2-2 coreAssembler Implementation Workspace Directory Contents

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

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

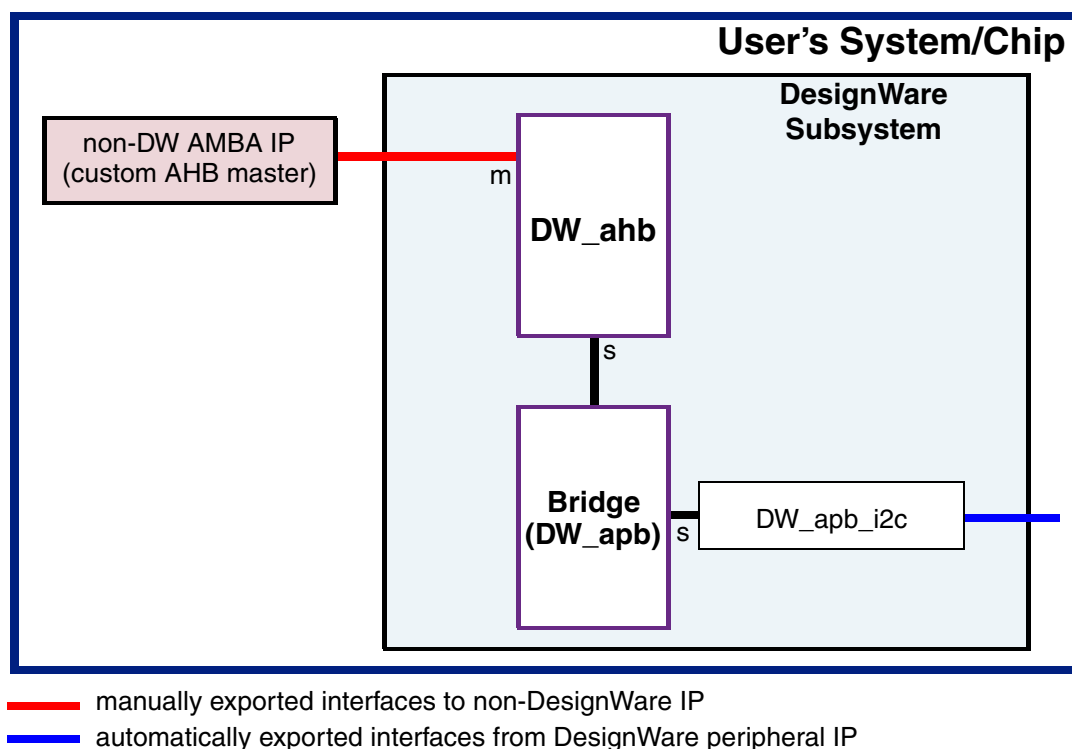
Directory/Subdirectory	Description
<i>i_component/src</i>	Includes the top-level RTL file, <i>design_name.v</i> . If you have a source license, this will contain plain-text RTL; if you only have a designware license, this will contain encrypted RTL. Generated during Add Subsystem Components activity;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

For details on some key files created during coreAssembler activities, refer to [“Database Files”](#) on page 29.

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

Figure 2-3 illustrates the DW_apb_i2c in a simple subsystem.

Figure 2-3 DW_apb_i2c in Simple Subsystem



The subsystem in Figure 2-3 contains the following components that you may want to use as you learn to use coreAssembler:

- DW_apb_i2c
- DW_ahb
- DW_apb
- AHB Master

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

2.3.2 Configuring the DW_apb_i2c within a Subsystem

The “Parameters” chapter on page 77 describes the DW_apb_i2c hardware configuration parameters that you configure using the coreAssembler GUI. Corresponding databooks for the other components in a subsystem contain “Parameters” chapters that describe their respective configuration parameters.

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

2.3.3 Creating Gate-Level Netlists within coreAssembler

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

2.3.4 Verifying the DW_apb_i2c within coreAssembler

The “Verification” chapter on [page 179](#) provides an overview of the testbench available for DW_apb_i2c verification using the coreAssembler GUI.

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

2.3.5 Running Leda on Generated Code with coreAssembler

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

2.4 Database Files

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

2.4.1 Design/HDL Files

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

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

2.4.1.1 RTL-Level Files

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

Table 2-3 RTL-Level Files

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

Table 2-3 RTL-Level Files (Continued)

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

2.4.1.2 Simulation Model Files

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

Table 2-4 Simulation Model Files

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

2.4.2 Synthesis Files

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

Table 2-5 Synthesis Files

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

2.4.3 Verification Reference Files

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

Table 2-6 Verification Reference Files

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

3

Functional Description

This chapter describes the functional behavior of DW_apb_i2c in more detail.

3.1 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 0 to 100 Kb/s), fast mode (with data rates less than or equal to 400 Kb/s), fast mode plus (with data rates less than or equal to 1000 Kb/s), and high-speed mode (with data rates less than or equal to 3.4 Mb/s).

**Note**

In this document, references to fast mode also apply to fast mode plus, unless specifically stated otherwise.

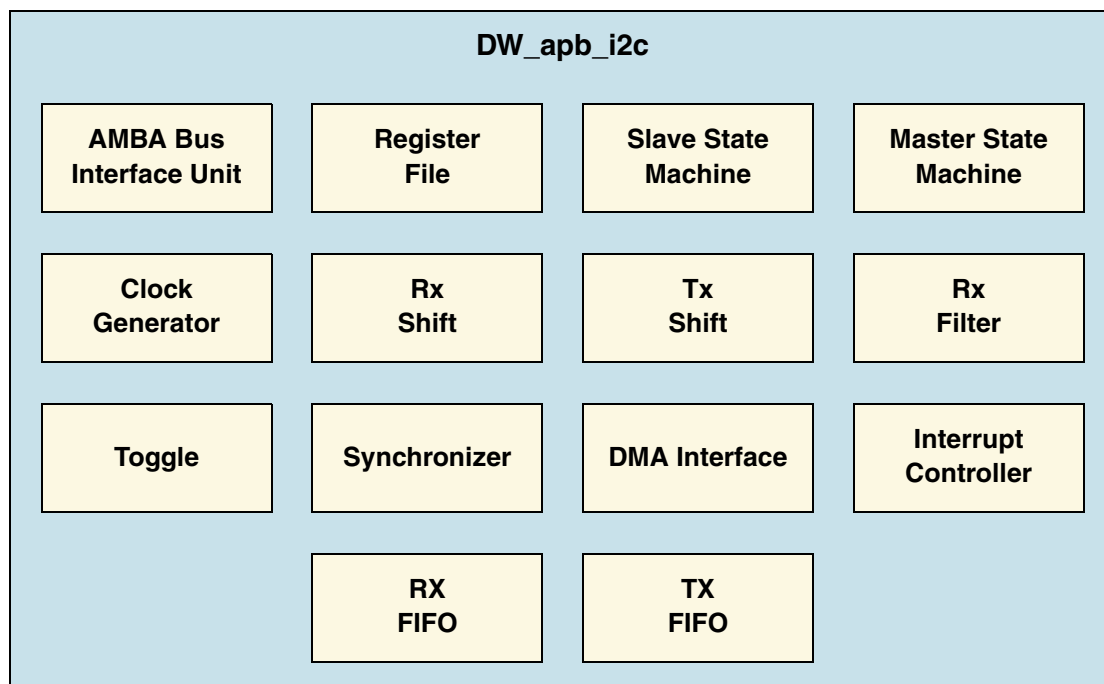
The DW_apb_i2c can communicate with devices only of these modes as long as they are attached to the bus. Additionally, high-speed mode and fast mode devices are downward compatible. For instance, high-speed mode devices can communicate with fast mode and standard mode devices in a mixed-speed bus system; 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.

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 3-1](#).

Figure 3-1 DW_apb_i2c Block Diagram



The following define the file names and functions of the blocks in [Figure 3-1](#):

- 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_slvfsm – 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 clock domain-crossing scheme within the DW_apb_i2c does not support pclk faster than ic_clk.

3.2 I²C Terminology

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

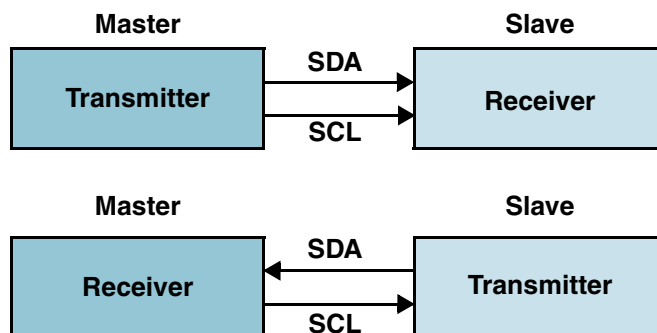
3.2.1 I²C Bus Terms

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 3-2](#).

Figure 3-2 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 47.
- **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 49.
- **SDA** – data signal line (Serial DAta)
- **SCL** – clock signal line (Serial CLock)

3.2.2 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.

3.3 I²C Behavior

The DW_apb_i2c can be controlled via software to be either:

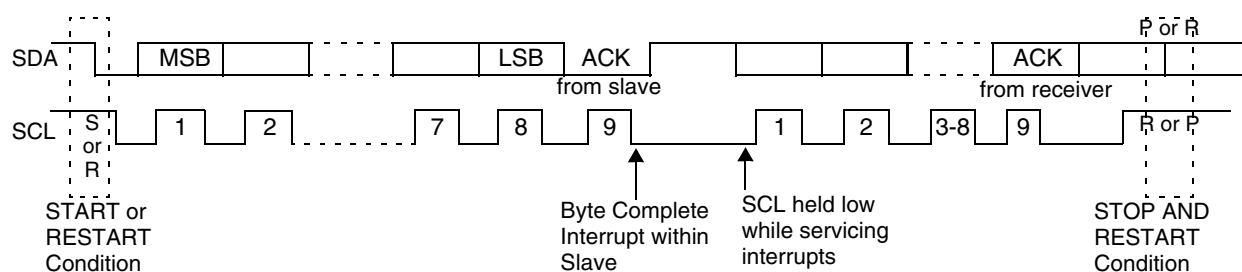
- An I²C master only, communicating with other I²C slaves; OR
- An I²C slave only, communicating with one more I²C masters.

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.

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 3-3.

Figure 3-3 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 “I2C Protocols” on page 38.

3.3.1 START and STOP Generation

When operating as an I²C master, putting data into the transmit FIFO causes the DW_apb_i2c to generate a START condition on the I²C bus. If the IC_EMPTYFIFO_HOLD_MASTER_EN parameter is set to 0, allowing the transmit FIFO to empty causes the DW_apb_i2c to generate a STOP condition on the I²C bus. If IC_EMPTYFIFO_HOLD_MASTER_EN is set to 1, then writing a 1 to IC_DATA_CMD[9] causes the DW_apb_i2c to generate a STOP condition on the I²C bus; a STOP condition is not issued if this bit is not set, even if the transmit FIFO is empty.

When operating as a slave, the DW_apb_i2c does not generate START and STOP conditions, as per the protocol. However, if a read request is made to the DW_apb_i2c, it holds the SCL line low until read data has been supplied to it. This stalls the I²C bus until read data is provided to the slave DW_apb_i2c, or the DW_apb_i2c slave is disabled by writing a 0 to bit 0 of the IC_ENABLE register.

3.3.2 Combined Formats

The DW_apb_i2c supports mixed read and write combined format transactions in both 7-bit and 10-bit addressing modes.

The DW_apb_i2c does not support mixed address and mixed address format — that is, a 7-bit address transaction followed by a 10-bit address transaction or vice versa — combined format transactions.

To initiate combined format transfers, IC_CON.IC_RESTART_EN should be set to 1. With this value set and operating as a master, when the DW_apb_i2c completes an I2C transfer, it checks the transmit FIFO and executes the next transfer. If the direction of this transfer differs from the previous transfer, the combined format is used to issue the transfer. If the transmit FIFO is empty when the current I2C transfer completes — depending on the value of IC_EMPTYFIFO_HOLD_MASTER_EN:

- Either a STOP is issued or,
- IC_DATA_CMD[9] is checked *and*:
 - If set to 1, a STOP bit is issued.
 - If set to 0, the SCL is held low until the next command is written to the transmit FIFO.

For more details, refer to “Tx FIFO Management and START, STOP and RESTART Generation” on page 42.

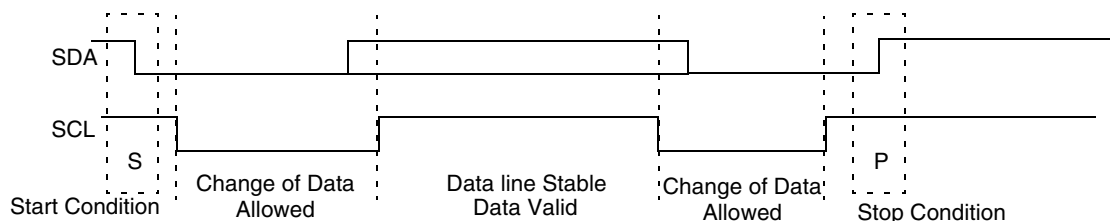
3.4 I²C Protocols

The DW_apb_i2c has the protocols discussed in this section.

3.4.1 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 3-4](#) 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 3-4 START and STOP Condition



Note

The signal transitions for the START/STOP conditions, as depicted in [Figure 3-4](#), 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.

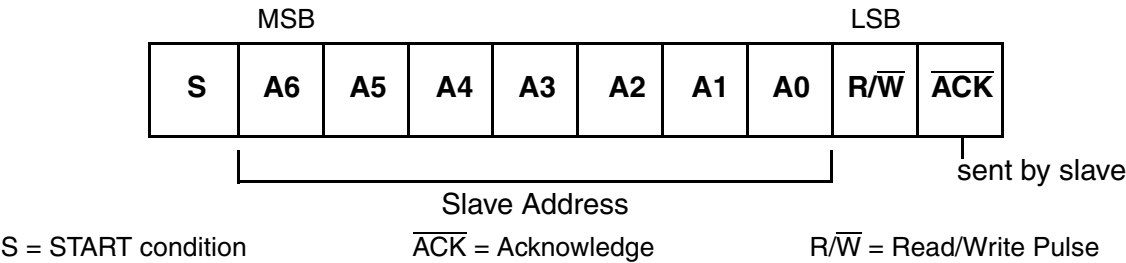
3.4.2 Addressing Slave Protocol

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

3.4.2.1 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 3-5](#). 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.

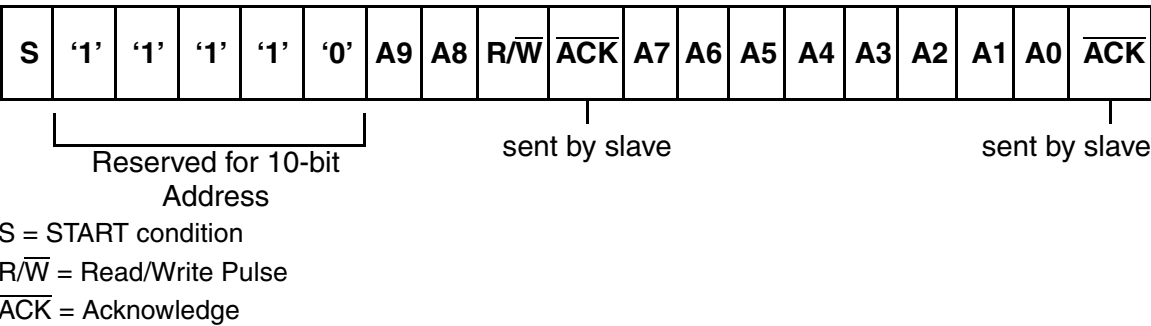
Figure 3-5 7-bit Address Format



3.4.2.2 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 3-6](#) shows the 10-bit address format.

Figure 3-6 10-bit Address Format



[Table 3-1](#) on page [39](#) defines the special purpose and reserved first byte addresses.

Table 3-1 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 41 .
0000 001	X	CBUS address. DW_apb_i2c ignores these accesses.

Table 3-1 I²C Definition of Bits in First Byte (Continued)

Slave Address	R/W Bit	Description
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 47).
1111 1XX	X	Reserved.
1111 0XX	X	10-bit slave addressing.

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.

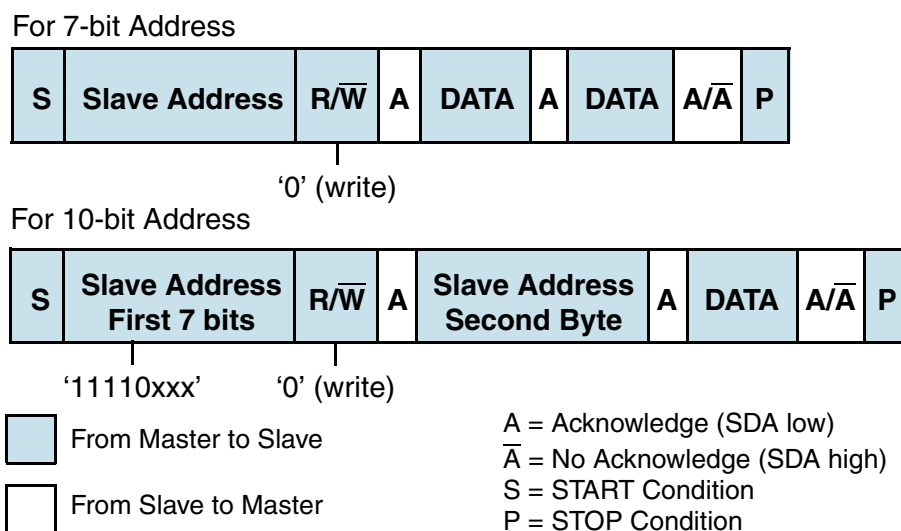
3.4.3 Transmitting and Receiving Protocol

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.

3.4.3.1 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 3-7, then the slave-receiver responds to the master-transmitter with an acknowledge pulse after every byte of data is received.

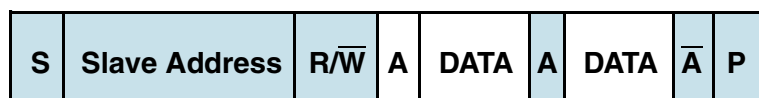
Figure 3-7 Master-Transmitter Protocol

3.4.3.2 Master-Receiver and Slave-Transmitter

If the master is receiving data as shown in [Figure 3-8](#), 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.

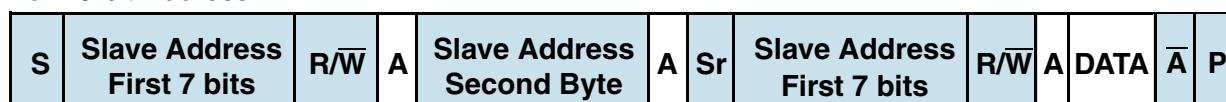
Figure 3-8 Master-Receiver Protocol

For 7-bit Address



'1' (read)

For 10-bit Address



'11110xxx'

'0' (write)

'11110xxx'

'1' (read)

From Master to Slave

A = Acknowledge (SDA low)

R = RESTART Condition

From Slave to Master

A-bar = No Acknowledge (SDA high)

P = STOP Condition

S = START 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. Operating in master mode, the DW_apb_i2c can then communicate with the same slave using a transfer of a different direction. For a description of the combined format transactions that the DW_apb_i2c supports, refer to [“Combined Formats”](#) on page 38.



Note

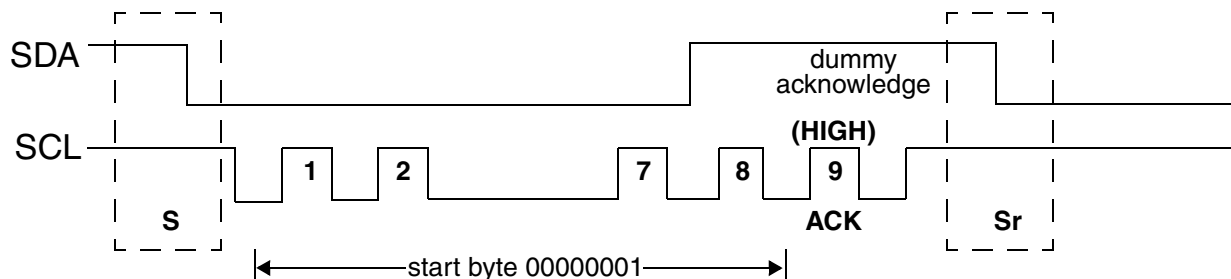
The DW_apb_i2c must be completely disabled—if I2C_DYNAMIC_TAR_UPDATE = 0—or inactive on the serial port—if I2C_DYNAMIC_TAR_UPDATE = 1—before the target slave address register (IC_TAR) can be reprogrammed.

3.4.4 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 3-9](#). 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 3-9 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.

3.5 Tx FIFO Management and START, STOP and RESTART Generation

When operating as a master, the DW_apb_i2c component supports two modes of Tx FIFO management. You use the IC_EMPTYFIFO_HOLD_MASTER_EN parameter to select between these two modes:

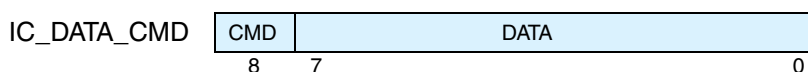
- IC_EMPTYFIFO_HOLD_MASTER_EN equals 0, illustrated in [Figure 3-10](#)
- IC_EMPTYFIFO_HOLD_MASTER_EN equals 1, illustrated in [Figure 3-13](#) on page 44

3.5.1 Tx FIFO Management When IC_EMPTYFIFO_HOLD_MASTER_EN = 0

When the value of IC_EMPTYFIFO_HOLD_MASTER_EN is 0, the component generates a STOP on the bus whenever the Tx FIFO becomes empty. If RESTART generation capability is enabled, the component generates a RESTART when the direction of the transfer in the Tx FIFO commands changes from Read to Write or vice-versa; if RESTART is not enabled, a STOP followed by a START is generated in this situation.

Figure 3-10 shows the bits in the **IC_DATA_CMD** register if **IC_EMPTYFIFO_HOLD_MASTER_EN** = 0.

Figure 3-10 IC_DATA_CMD Register if IC_EMPTYFIFO_HOLD_MASTER_EN = 0



DATA –Read/Write field; data retrieved from slave is read from this field; data to be sent to slave is written to this field.
 CMD –Write-only field; this bit determines whether transfer to be carried out is Read (CMD=1) or Write (CMD=0)

Figure 3-11 shows a timing diagram that illustrates the behavior of the DW_apb_i2c when Tx FIFO becomes empty while operating as a master transmitter when **IC_EMPTYFIFO_HOLD_MASTER_EN**=0.

Figure 3-11 Master Transmitter — Tx FIFO Becomes Empty If IC_EMPTYFIFO_HOLD_MASTER_EN = 0

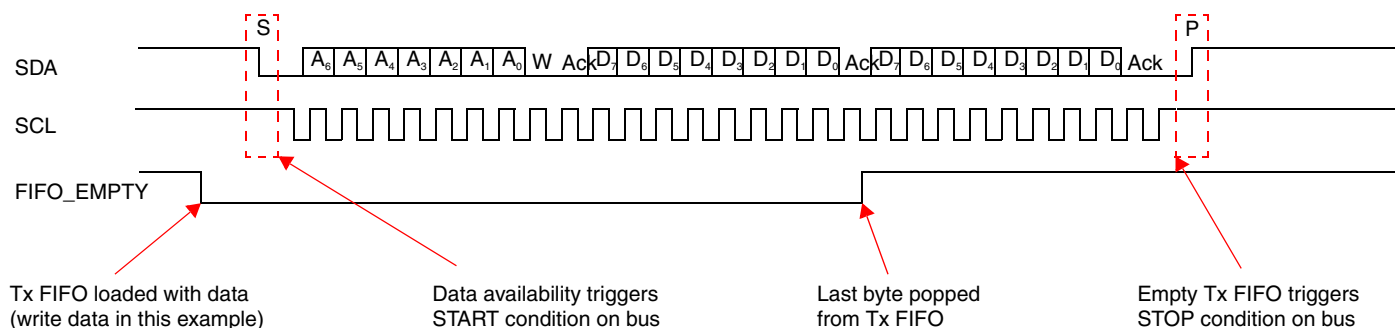
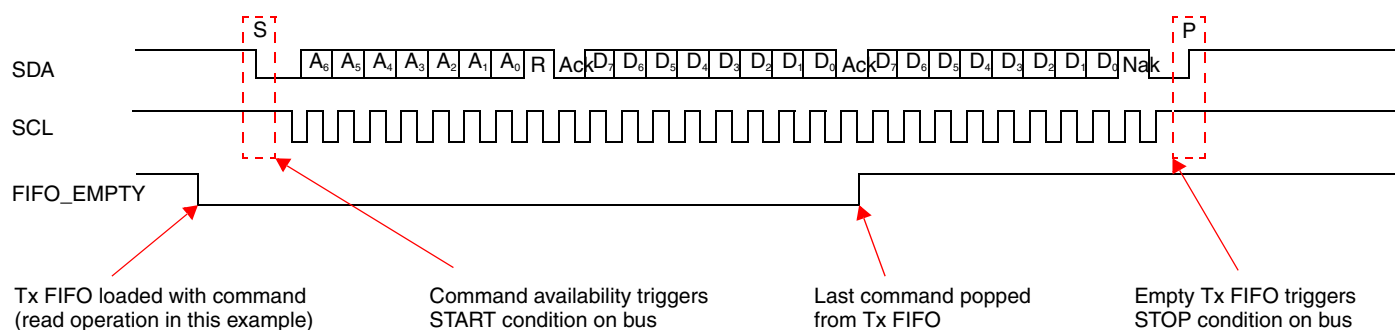


Figure 3-12 shows a timing diagram that illustrates the behavior of the DW_apb_i2c when Tx FIFO becomes empty while operating as a master receiver when **IC_EMPTYFIFO_HOLD_MASTER_EN**=0.

Figure 3-12 Master Receiver — Tx FIFO Becomes Empty If IC_EMPTYFIFO_HOLD_MASTER_EN = 0

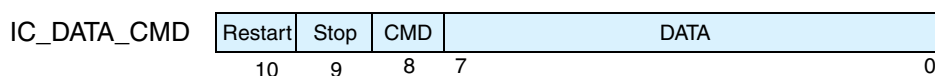


3.5.2 Tx FIFO Management When **IC_EMPTYFIFO_HOLD_MASTER_EN** = 1

When the value of **IC_EMPTYFIFO_HOLD_MASTER_EN** is 1, the component does not generate a STOP if the Tx FIFO becomes empty; in this situation the component holds the SCL line low, stalling the bus until a new entry is available in the Tx FIFO. A STOP condition is generated only when the user specifically requests it by setting bit 9 (Stop bit) of the command written to **IC_DATA_CMD** register.

Figure 3-13 shows the bits in the **IC_DATA_CMD** register if **IC_EMPTYFIFO_HOLD_MASTER_EN** = 1.

Figure 3-13 IC_DATA_CMD Register if IC_EMPTYFIFO_HOLD_MASTER_EN = 1



DATA –Read/Write field; data retrieved from slave is read from this field; data to be sent to slave is written to this field
 CMD –Write-only field; this bit determines whether transfer to be carried out is Read (CMD=1) or Write (CMD=0)
 Stop –Write-only field; this bit determines whether STOP is generated after data byte is sent or received
 Restart – Write-only field; this bit determines whether RESTART (or STOP followed by START in case of restart capability is not enabled) is generated before data byte is sent or received

Figure 3-14 illustrates the behavior of the DW_apb_i2c when the Tx FIFO becomes empty while operating as a master transmitter, as well as showing the generation of a STOP condition when **IC_EMPTYFIFO_HOLD_MASTER_EN**=1.

Figure 3-14 Master Transmitter — Tx FIFO Empties/STOP Generation If IC_EMPTYFIFO_HOLD_MASTER_EN = 1

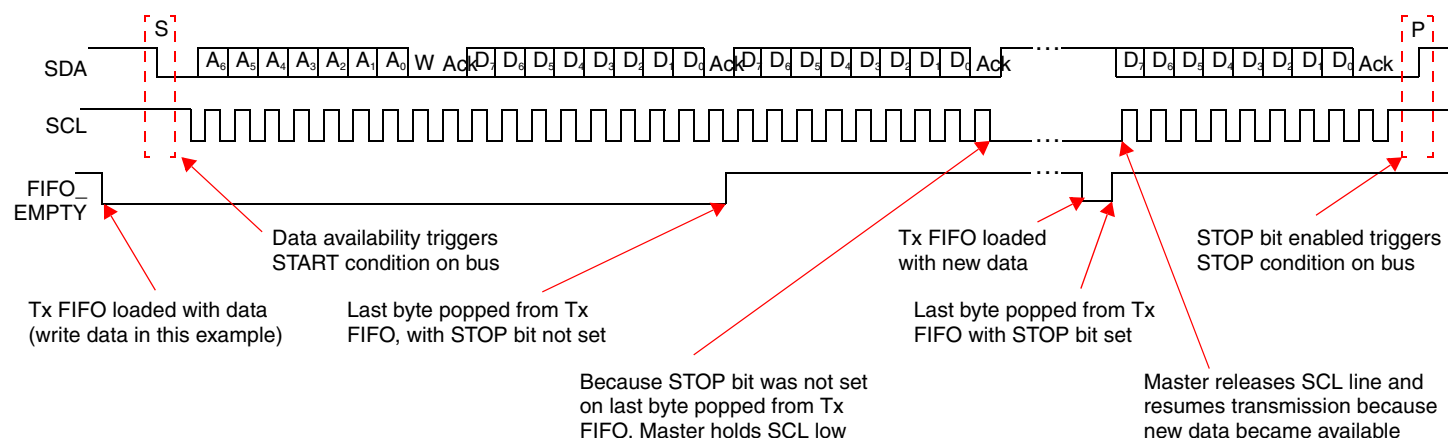


Figure 3-15 Master Receiver — Tx FIFO Empties/STOP Generation If IC_EMPTYFIFO_HOLD_MASTER_EN = 1

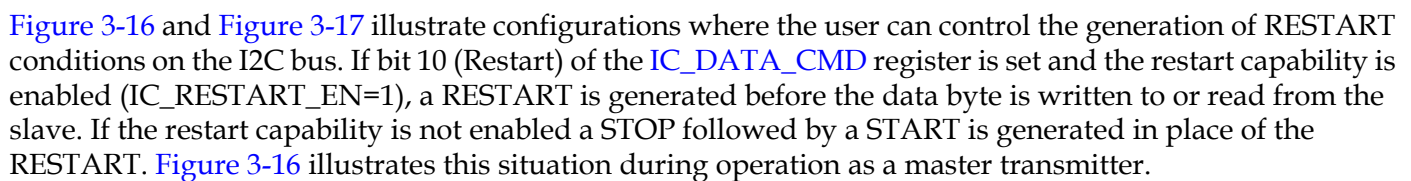


Figure 3-16 Master Transmitter — Restart Bit of IC_DATA_CMD Is Set (IC_EMPTYFIFO_HOLD_MASTER_EN = 1)

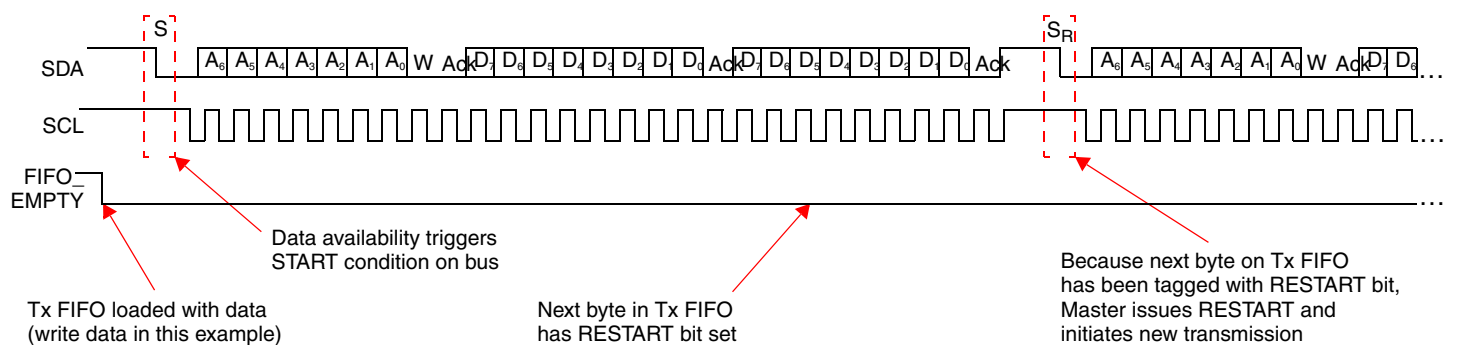


Figure 3-17 illustrates the same situation, but during operation as a master receiver.

Figure 3-17 Master Receiver — Restart Bit of IC_DATA_CMD Is Set (IC_EMPTYFIFO_HOLD_MASTER_EN = 1)

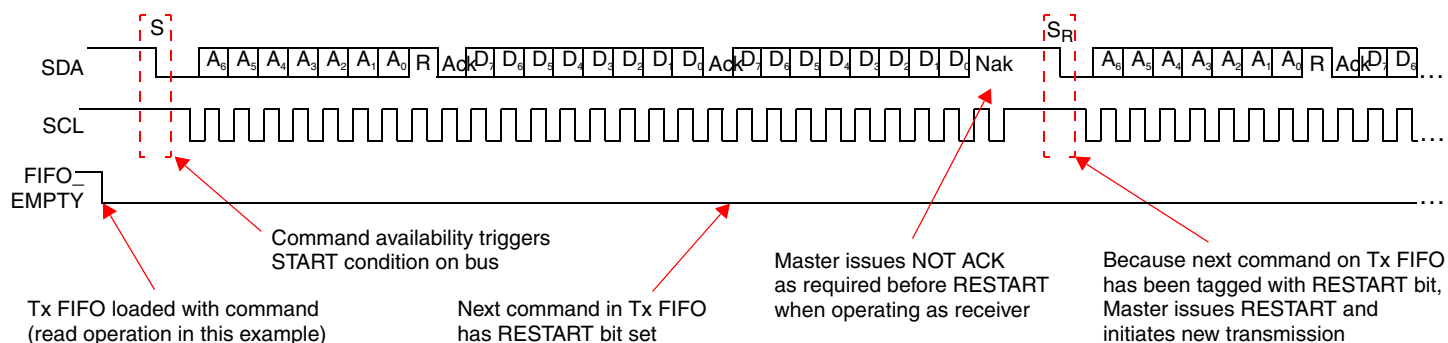


Figure 3-18 illustrates operation as a master transmitter where the Stop bit of the IC_DATA_CMD register is set and the Tx FIFO is not empty (IC_EMPTYFIFO_HOLD_MASTER_EN=1).

Figure 3-18 Master Transmitter — Stop Bit of IC_DATA_CMD Set/Tx FIFO Not Empty (IC_EMPTYFIFO_HOLD_MASTER_EN=1)

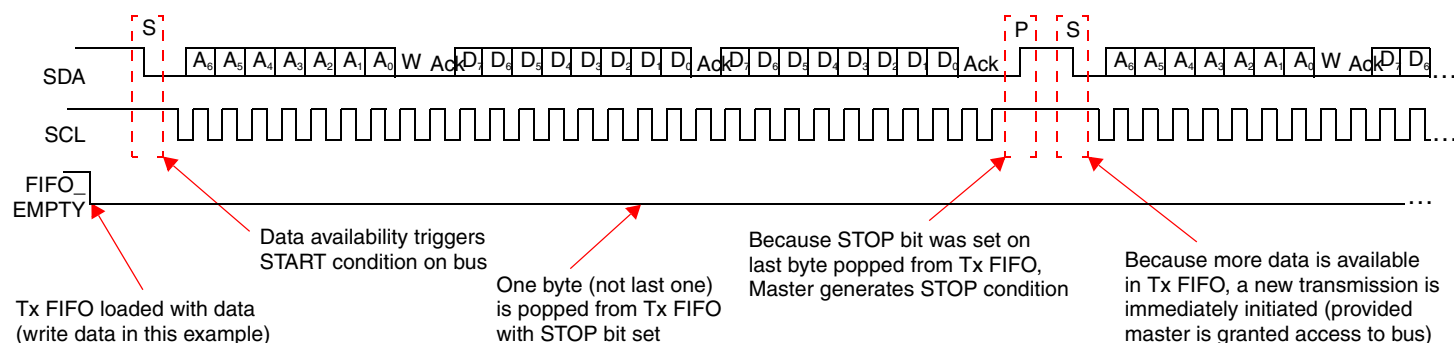


Figure 3-19 illustrates operation as a master transmitter where the first byte loaded into the Tx FIFO is allowed to go empty with the Restart bit set (IC_EMPTYFIFO_HOLD_MASTER_EN=1).

Figure 3-19 Master Transmitter — First Byte Loaded Into Tx FIFO Allowed to Empty, Restart Bit Set (IC_EMPTYFIFO_HOLD_MASTER_EN=1)

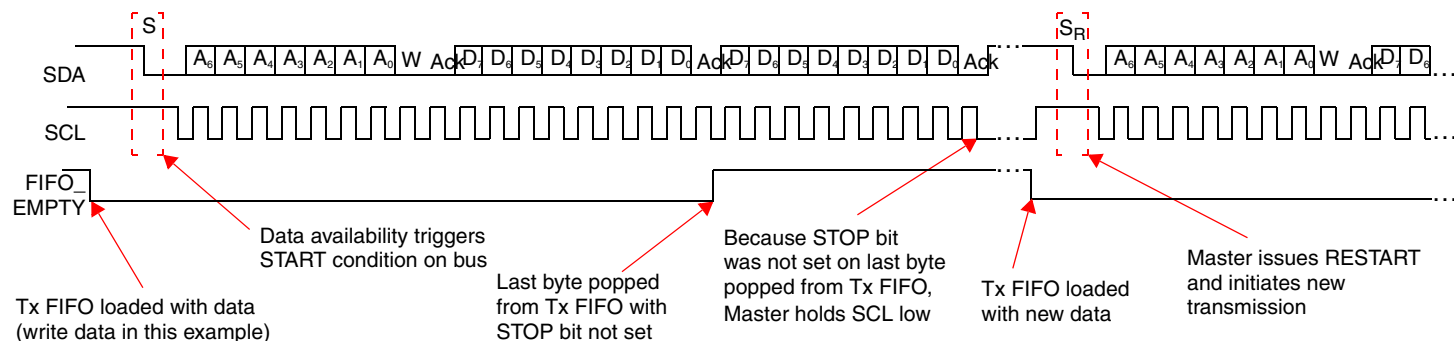


Figure 3-20 illustrates operation as a master receiver where the Stop bit of the `IC_DATA_CMD` register is set and the Tx FIFO is not empty (`IC_EMPTYFIFO_HOLD_MASTER_EN=1`).

Figure 3-20 Master Receiver — Stop Bit of `IC_DATA_CMD` Set/Tx FIFO Not Empty (`IC_EMPTYFIFO_HOLD_MASTER_EN=1`)

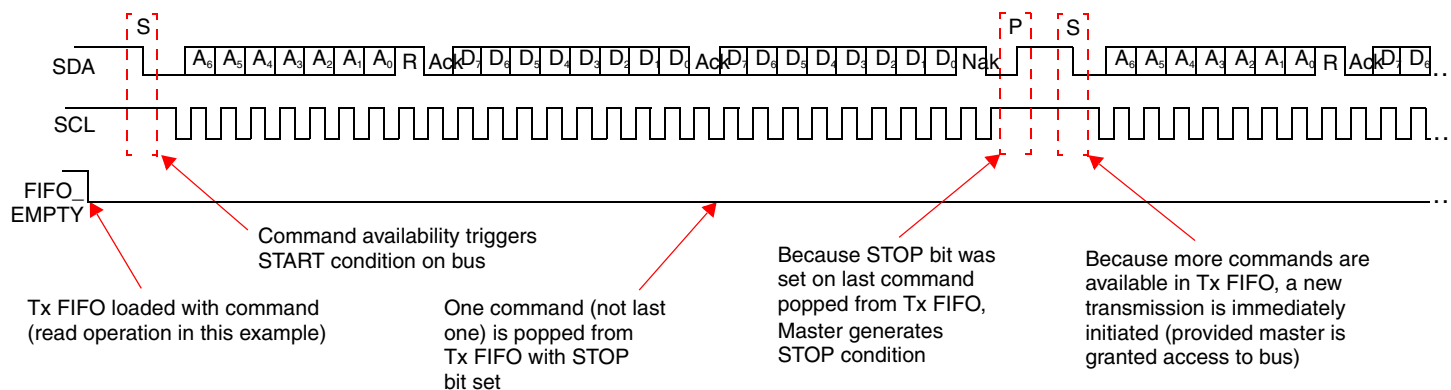
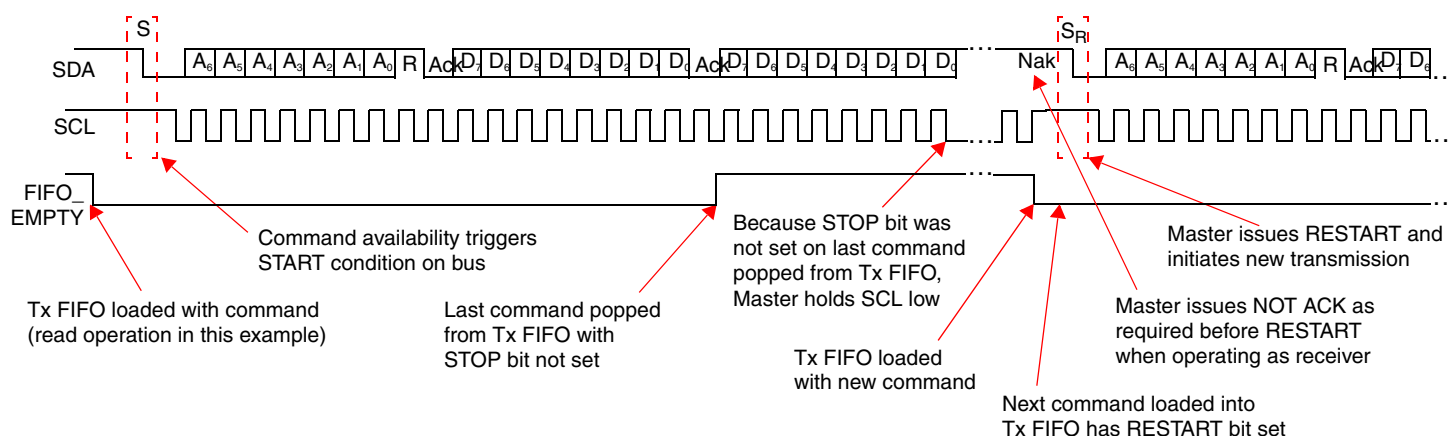


Figure 3-21 illustrates operation as a master receiver where the first command loaded after the Tx FIFO is allowed to empty and the Restart bit is set (`IC_EMPTYFIFO_HOLD_MASTER_EN=1`).

Figure 3-21 Master Receiver — First Command Loaded After Tx FIFO Allowed to Empty/Restart Bit Set (`IC_EMPTYFIFO_HOLD_MASTER_EN=1`)



3.6 Multiple Master Arbitration

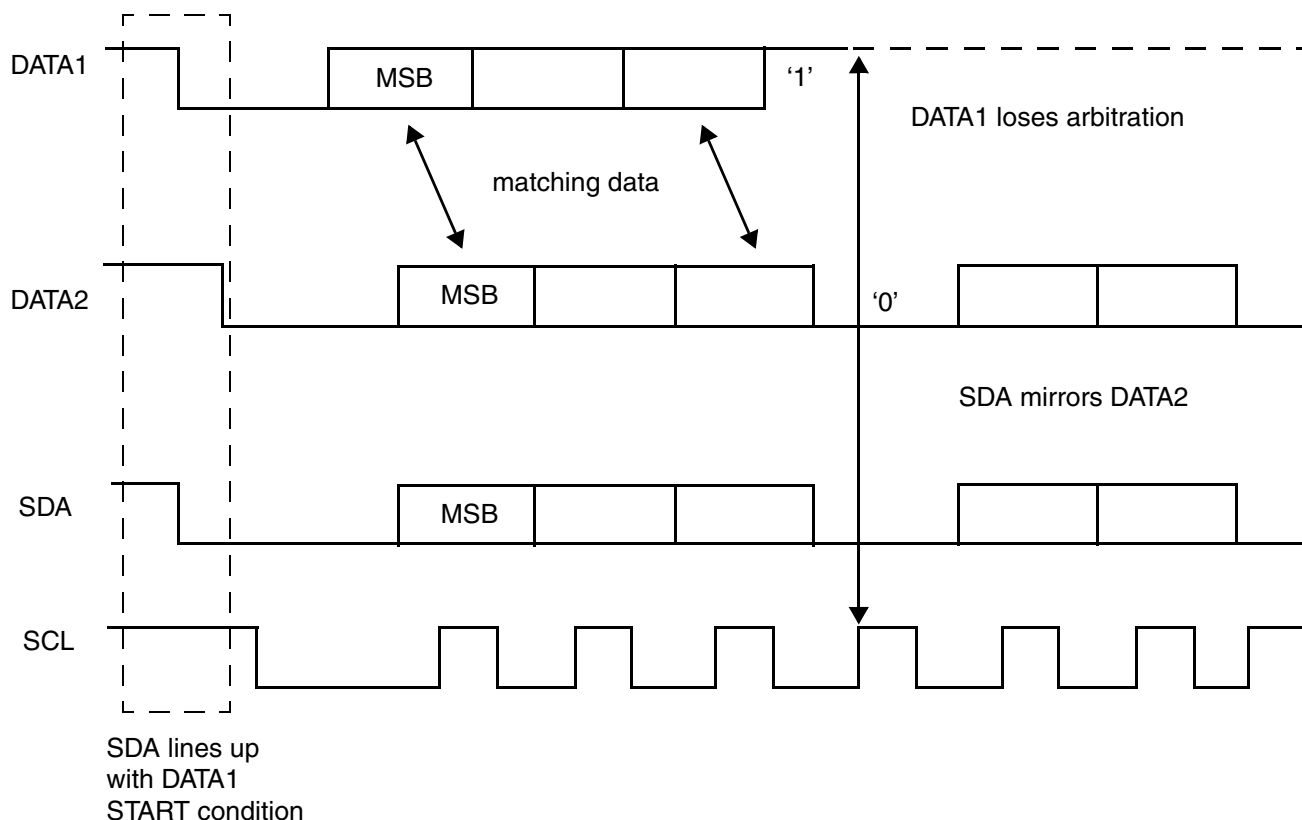
The DW_apb_i2c bus protocol allows multiple masters to reside on the same bus. If there are two masters on the same I²C-bus, there is an arbitration procedure if both try to take control of the bus at the same time by generating a START condition at the same time. Once a master (for example, a microcontroller) 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.

Upon detecting that it has lost arbitration to another master, the DW_apb_i2c will stop generating SCL (ic_clk_oe).

Figure 3-22 illustrates the timing of when two masters are arbitrating on the bus.

Figure 3-22 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

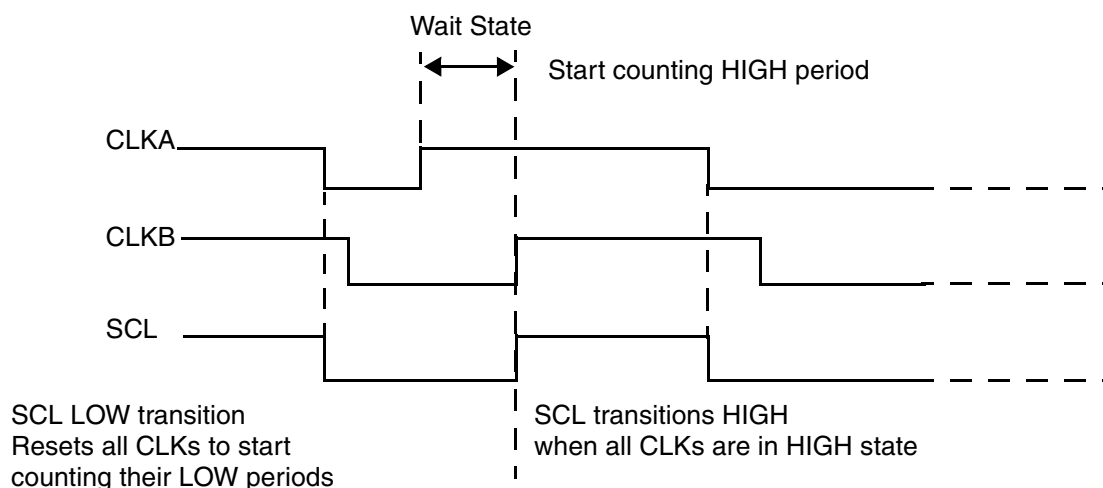
Slaves are not involved in the arbitration process.

3.7 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 counts 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 3-23](#). Optionally, slaves may hold the SCL line low to slow down the timing on the I²C bus.

Figure 3-23 Multi-Master Clock Synchronization



3.8 Operation Modes

This section provides information on operation modes.



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.

3.8.1 Slave Mode Operation

This section discusses slave mode procedures.

3.8.1.1 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.



Attention

It is recommended that the DW_apb_i2c Slave be brought out of reset only when the I2C bus is IDLE. De-asserting the reset when a transfer is ongoing on the bus causes internal synchronization flip-flops used to synchronize SDA and SCL to toggle from a reset value of 1 to the actual value on the bus. This can result in SDA toggling from 1 to 0 while SCL is 1, thereby causing a false START condition to be detected by the DW_apb_i2c Slave. This scenario can also be avoided by configuring the DW_apb_i2c with IC_SLAVE_DISABLE = 1 and IC_MASTER_MODE = 1 so that the Slave interface is disabled after reset. It can then be enabled by programming IC_CON[0] = 0 and IC_CON[6] = 0 after the internal SDA and SCL have synchronized to the value on the bus; this takes approximately 6 ic_clk cycles after reset de-assertion.

3.8.1.2 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.

3.8.1.3 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, and `IC_RX_FULL_HLD_BUS_EN = 0`, 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 overflows, 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 the system.

If the Rx FIFO is completely filled with data when a byte is pushed, and `IC_RX_FULL_HLD_BUS_EN = 1`, then the DW_apb_i2c slave holds the I2C SCL line low until the Rx FIFO has some space, and then continues with the next read request.

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 50) be implemented for periodic reads of the [IC_STATUS](#) 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.

3.8.1.4 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 50.

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.

3.8.2 Master Mode Operation

This section discusses master mode procedures.

3.8.2.1 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 86](#). For more information about how this parameter affects the IC_TAR register, see [page 106](#).

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).

3.8.2.1.1 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 bit 0 of 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 to 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.

3.8.2.1.2 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 bit 0 of 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 to bit 0 of 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.



When a DW_apb_i2c Master is configured with `IC_EMPTYFIFO_HOLD_MASTER_EN = 0`, then 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.

3.8.2.2 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 software ensures that there are no other commands in the Tx FIFO that use the existing TAR address. If the software does not ensure this, then IC_TAR should be re-programmed only if the following conditions are met:

- DW_apb_i2c is not enabled (`IC_ENABLE[0]=0`);

OR

DW_apb_i2c is enabled (`IC_ENABLE[0]=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`);¹

You can change the TAR address dynamically without losing the bus, only if the following conditions are met.

- DW_apb_i2c is enabled (`IC_ENABLE[0]=1`); AND
`IC_EMPTYFIFO_HOLD_MASTER_EN` configuration parameter is set to 1; AND
 DW_apb_i2c is enabled to operate in Master mode (`IC_CON[0]=1`); AND
 there are NO entries in the Tx FIFO and the master is in HOLD state (`IC_INTR_STAT[13]=1`);¹



DW_apb_i2c uses the TAR address if either of the following conditions is true:

- The command has either RESTART or STOP bit set.
- The direction is changed in commands with a read command following a write command or vice versa

The updated TAR address comes into effect only when the next START or RESTART occurs on the bus.

3.8.2.3 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. The DW_apb_i2c master continues to initiate transfers as long as there are commands present in the transmit FIFO. If the transmit FIFO becomes empty — depending on the value of

1. If the software or application is aware the the DW_apb_i2c is not using the TAR address for the pending commands in the Tx FIFO, then it is possible to update the TAR address even while the Tx FIFO has entries (`IC_STATUS[2] = 0`).

IC_EMPTYFIFO_HOLD_MASTER_EN, the master either inserts a STOP condition after completing the current transfers, or it checks to see if IC_DATA_CMD[9] is set to 1.

- If set to 1, it issues a STOP condition after completing the current transfer.
- If set to 0, it holds SCL low until next command is written to the transmit FIFO.

For more details, refer to “Tx FIFO Management and START, STOP and RESTART Generation” on page 42.

3.8.3 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 bit 0 of 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.



Note

When IC_EMPTYFIFO_HOLD_MASTER_EN = 1, the DW_apb_i2c Master can be disabled only if the current command being processed—when the ic_enable de-assertion occurs—has the STOP bit set to 1.

When an attempt is made to disable the DW_apb_i2c Master while processing a command without the STOP bit set, the DW_apb_i2c Master continues to remain active, holding the SCL line low until a new command is received in the Tx FIFO.

3.8.3.1 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 25us.
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 bit 0 of the IC_ENABLE register 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 >= 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.

3.8.4 Aborting I2C Transfers

The ABORT control bit of the IC_ENABLE register allows the software to relinquish the I2C bus before completing the issued transfer commands from the Tx FIFO. In response to an ABORT request, the controller issues the STOP condition over the I2C bus, followed by Tx FIFO flush. Aborting the transfer is allowed only in master mode of operation.

3.8.4.1 Procedure

1. Stop filling the Tx FIFO (IC_DATA_CMD) with new commands.
2. When operating in DMA mode, disable the transmit DMA by setting TDMAE to 0.
3. Set bit 1 of the IC_ENABLE register (ABORT) to 1.
4. Wait for the M_TX_ABRT interrupt.
5. Read the IC_TX_ABRT_SOURCE register to identify the source as ABRT_USER_ABRT.

3.9 Spike Suppression

The DW_apb_i2c contains programmable spike suppression logic that match requirements imposed by the *I²C Bus Specification* for SS/FS (tSP, Table 4) and HS (tSP, Table 6) modes.

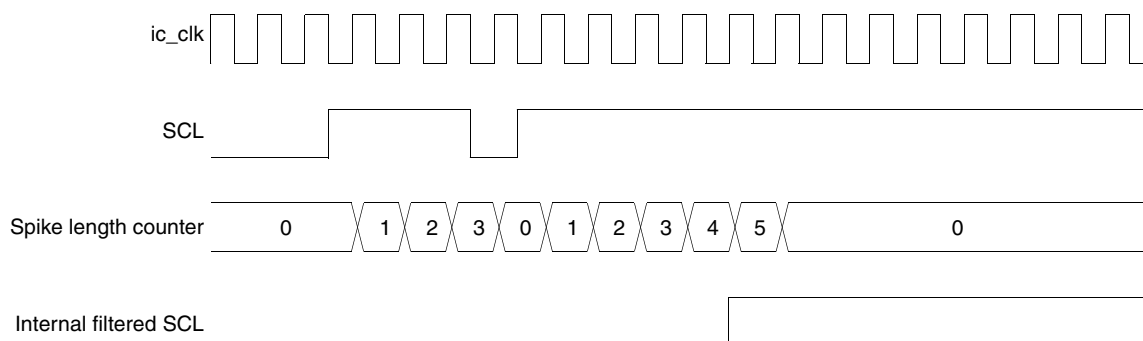
This logic is based on counters that monitor the input signals (SCL and SDA), checking if they remain stable for a predetermined amount of ic_clk cycles before they are sampled internally. There is one separate counter for each signal (SCL and SDA). The number of ic_clk cycles can be programmed by the user and should be calculated taking into account the frequency of ic_clk and the relevant spike length specification.

Each counter is started whenever its input signal changes its value. Depending on the behavior of the input signal, one of the following scenarios occurs:

- The input signal remains unchanged until the counter reaches its count limit value. When this happens, the internal version of the signal is updated with the input value, and the counter is reset and stopped. The counter is not restarted until a new change on the input signal is detected.
- The input signal changes again before the counter reaches its count limit value. When this happens, the counter is reset and stopped, but the internal version of the signal is not updated. The counter remains stopped until a new change on the input signal is detected.

The timing diagram in [Figure 3-24](#) illustrates the behavior described above.

Figure 3-24 Spike Suppression Example



The count limit value used in this example is 5 and was calculated for a 10 ns `ic_clk` period and for SS/FS operation (50 ns spike suppression).

**Note**

There is a 2-stage synchronizer on the SCL input, but for the sake of simplicity this synchronization delay was not included in the timing diagram in [Figure 3-24](#).

The *I²C Bus Specification* calls for different maximum spike lengths according to the operating mode – 50 ns for SS and FS; 10 ns for HS – so two registers are required to store the values needed for each case:

- Register `IC_FS_SPKLEN` holds the maximum spike length for SS and FS modes
- Register `IC_HS_SPKLEN` holds the maximum spike value for HS mode.

**Note**

`IC_HS_SPKLEN` is implemented only if the component is configured for HS operation; that is, (`IC_MAX_SPEED` = High).

These registers are 8 bits wide and accessible through the APB interface for read and write purposes; however, they can be written to only when the `DW_apb_i2c` is disabled. The minimum value that can be programmed into these registers is 1; attempting to program a value smaller than 1 results in the value 1 being written.

The default value for these registers is automatically calculated in coreConsultant based on the value of `ic_clk` period, but this value can be overridden by the user when configuring the component.

**Note**

- Because the minimum value that can be programmed into the `IC_FS_SPKLEN` and `IC_HS_SPKLEN` registers is 1, the spike length specification can be exceeded for low frequencies of `ic_clk`. Consider the simple example of a 10 MHz (100 ns period) `ic_clk`; in this case, the minimum spike length that can be programmed is 100 ns, which means that spikes up to this length are suppressed.
- Standard synchronization logic (two flip-flops in series) is implemented upstream of the spike suppression logic and is not affected in any way by the contents of the spike length registers or the operation of the spike suppression logic; the two operations (synchronization and spike suppression) are completely independent.
Because the SCL and SDA inputs are asynchronous to `ic_clk`, there is one `ic_clk` cycle uncertainty in the sampling of these signals; that is, depending on when they occur relative to the rising edge of `ic_clk`, spikes of the same original length might show a difference of one `ic_clk` cycle after being sampled.
- Spike suppression is symmetrical; that is, the behavior is exactly the same for transitions from 0 to 1 and from 1 to 0.

3.10 Fast Mode Plus Operation

In fast mode plus, the DW_apb_i2c allows the fast mode operation to be extended to support speeds up to 1000 Kb/s. To enable the DW_apb_i2c for fast mode plus operation, perform the following steps before initiating any data transfer:

1. Configure the Maximum Speed mode of DW_apb_i2c Master or Slave to Fast Mode or High Speed mode (`IC_MAX_SPEED_MODE` = 2).
2. Set `ic_clk` frequency greater than or equal to 32 MHz (refer to “[Standard Mode, Fast Mode, and Fast Mode Plus](#)” on page 61).
3. Program the `IC_CON` register [2:1] = 2'b10 for fast mode or fast mode plus.
4. Program `IC_FS_SCL_LCNT` and `IC_FS_SCL_HCNT` registers to meet the fast mode plus SCL (refer to “[IC_CLK Frequency Configuration](#)” on page 59).
5. Program the `IC_FS_SPKLEN` register to suppress the maximum spike of 50ns.
6. Program the `IC_SDA_SETUP` register to meet the minimum data setup time (tSU; DAT).

3.11 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 in order 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 the DW_apb_i2c is enabled to operate only as an I²C slave, since these registers are used only to determine the SCL timing requirements for operation as an I²C master.

3.11.1 Minimum High and Low Counts

When the DW_apb_i2c operates as an I²C master, in both transmit and receive transfers:

- `IC_SS_SCL_LCNT` and `IC_FS_SCL_LCNT` register values must be larger than `IC_FS_SPKLEN` + 7.
- `IC_SS_SCL_HCNT` and `IC_FS_SCL_HCNT` register values must be larger than `IC_FS_SPKLEN` + 5.
- If the component is programmed to support HS, `IC_HS_SCL_LCNT` register value must be larger than `IC_HS_SPKLEN` + 7.
- If the component is programmed to support HS, `IC_HS_SCL_HCNT` register value must be larger than `IC_HS_SPKLEN` + 5.

Details regarding the DW_apb_i2c high and low counts are as follows:

- The minimum value of $IC_*_SPKLEN + 7$ for the $*_LCNT$ registers is due to the time required for the DW_apb_i2c to drive SDA after a negative edge of SCL.
- The minimum value of $IC_*_SPKLEN + 5$ for the $*_HCNT$ registers is due to the time required for the DW_apb_i2c to sample SDA during the high period of SCL.
- The DW_apb_i2c adds one cycle to the programmed $*_LCNT$ value in order to generate the low period of the SCL clock; this is due to the counting logic for SCL low counting to $(*_LCNT + 1)$.
- The DW_apb_i2c adds $IC_*_SPKLEN + 7$ cycles to the programmed $*_HCNT$ value in order to generate the high period of the SCL clock; this is due to the following factors:
 - The counting logic for SCL high counts to $(*_HCNT+1)$.
 - The digital filtering applied to the SCL line incurs a delay of $SPKLEN + 2$ ic_clk cycles, where SPKLEN is:
 - IC_FS_SPKLEN if the component is operating in SS or FS
 - IC_HS_SPKLEN if the component is operating in HS.
 This filtering includes metastability removal and the programmable spike suppression on SDA and SCL 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 of which the DW_apb_i2c is capable is nine (9) ic_clk periods ($7 + 1 + 1$), while the minimum SCL high time is thirteen (13) ic_clk periods ($6 + 1 + 3 + 3$).

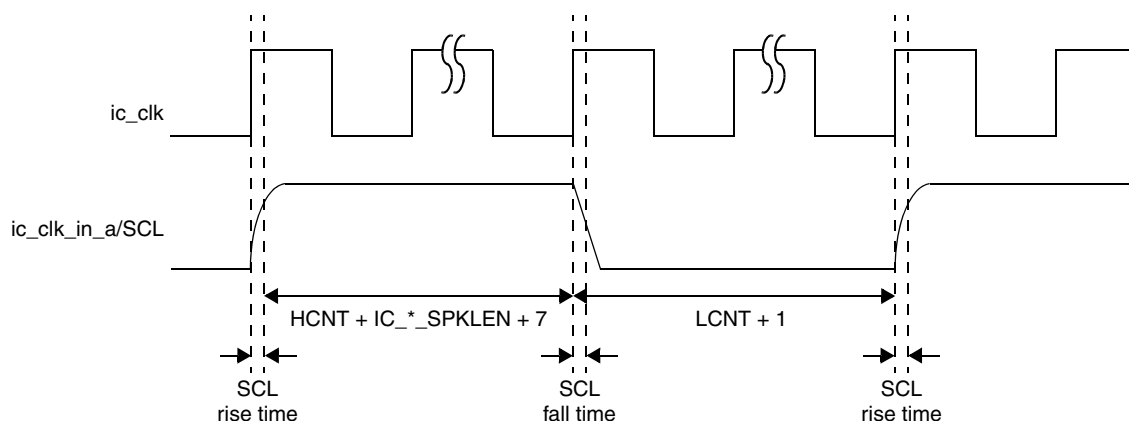


Note

The total high time and low time of SCL generated by the DW_apb_i2c master is also influenced by the rise time and fall time of the SCL line, as shown in the illustration and equations in [Figure 3-25](#) on page 61. It should be noted that the SCL rise and fall time parameters vary, depending on external factors such as:

- Characteristics of IO driver
- Pull-up resistor value
- Total capacitance on SCL line, and so on

These characteristics are beyond the control of the DW_apb_i2c.

Figure 3-25 Impact of SCL Rise Time and Fall Time on Generated SCL

$$\text{SCL_High_time} = [(\text{HCNT} + \text{IC_}\ast\text{_SPKLEN} + 7) * \text{ic_clk}] + \text{SCL_Fall_time}$$

$$\text{SCL_Low_time} = [(\text{LCNT} + 1) * \text{ic_clk}] - \text{SCL_Fall_time} + \text{SCL_Rise_time}$$

3.11.2 Minimum IC_CLK Frequency

This section describes the minimum ic_clk frequencies that the DW_apb_i2c supports for each speed mode, and the associated high and low count values. In Slave mode, IC_SDA_HOLD (Thd;dat) and IC_SDA_SETUP (Tsu;dat) need to be programmed to satisfy the I2C protocol timing requirements.

The following examples are for the case where IC_FS_SPKLEN and IC_HS_SPKLEN are programmed to 2.

3.11.2.1 Standard Mode, Fast Mode, and Fast Mode Plus

This section details how to derive a minimum ic_clk value for standard and fast modes of the DW_apb_i2c. Although the following method shows how to do fast mode calculations, you can also use the same method in order to do calculations for standard mode and fast mode plus.



Note

The following computations do not consider the SCL_Rise_time and SCL_Fall_time.

Given conditions and calculations for the minimum DW_apb_i2c ic_clk value in fast mode:

- Fast mode has data rate of 400kb/s; implies SCL period of $1/400\text{kHz} = 2.5\mu\text{s}$
- Minimum hcnt value of 14 as a seed value; IC_HCNT_FS = 14
- Protocol minimum SCL high and low times:
 - MIN_SCL_LOWtime_FS = 1300ns
 - MIN_SCL_HIGHtime_FS = 600ns

Derived equations:

$$\frac{\text{SCL_PERIOD_FS}}{\text{IC_HCNT_FS} + \text{IC_LCNT_FS}} = \text{IC_CLK_PERIOD}$$

$$\text{IC_LCNT_FS} \times \text{IC_CLK_PERIOD} = \text{MIN_SCL_LOWtime_FS}$$

Combined, the previous equations produce the following:

$$IC_LCNT_FS \times \frac{SCL_PERIOD_FS}{IC_LCNT_FS + IC_HCNT_FS} = MIN_SCL_LOWtime_FS$$

Solving for IC_LCNT_FS:

$$IC_LCNT_FS \times \frac{2.5\mu s}{IC_LCNT_FS + 14} = 1.3\mu s$$

The previous equation gives:

$$IC_LCNT_FS = \text{roundup}(15.166) = 16$$

These calculations produce IC_LCNT_FS = 16 and IC_HCNT_FS = 14, giving an ic_clk value of:

$$\frac{2.5\mu s}{16 + 14} = 83.3ns = 12Mhz$$

Testing these results shows that protocol requirements are satisfied.

3.11.2.2 High-Speed Modes

The method used for standard and fast modes can also be used to derive ic_clk values for high-speed modes. For example, given a high-speed mode with a 100pf bus loading, using the standard and fast modes method produces the following:

- IC_LCNT_HS = 17
- IC_HCNT_HS = 14
- ic_clk = 105.4 Mhz

Table 3-2 lists the minimum ic_clk values for all modes with high and low count values.

Table 3-2 ic_clk in Relation to High and Low Counts

Speed Mode	ic_clk _{freq} (MHz)	Minimum Value of IC_*_SPK LEN	SCL Low Time in ic_clks	SCL Low Program Value	SCL Low Time	SCL High Time in ic_clks	SCL High Program Value	SCL High Time
SS	2.7	1	13	12	4.7 μs	14	6	5.2 μs
FS	12.0	1	16	15	1.33 μs	14	6	1.16 μs
FM+	32	2	16	15	500 ns	16	7	500 ns
HS (400pf)	51	1	17	16	333 ns	14	6	274 ns
HS (100pf)	105.4	1	17	16	161 ns	14	6	132 ns



- The IC_*_SCL_LCNT and IC_*_SCL_HCNT registers are programmed using the SCL low and high program values in [Table 3-2](#), which are calculated using SCL low count minus 1, and SCL high counts minus 8, respectively.

The values in [Table 3-2](#) are based on IC_SDA_RX_HOLD = 0. The maximum IC_SDA_RX_HOLD value depends on the IC_*CNT registers in Master mode, as described in “[SDA Hold Timings in Receiver](#)” on page 65.

- In order to compute the HCNT and LCNT considering RC timings, use the following equations:

$$\text{IC_HCNT_}^* = [(\text{HCNT} + \text{IC_}\text{*}\text{_SPKLEN} + 7) * \text{ic_clk}] + \text{SCL_Fall_time}$$

$$\text{IC_LCNT_}^* = [(\text{LCNT} + 1) * \text{ic_clk}] - \text{SCL_Fall_time} + \text{SCL_Rise_time}$$

3.11.2.3 Calculating High and Low Counts

The calculations below show how to calculate SCL high and low counts for each speed mode in the DW_apb_i2c. For the calculations to work, the ic_clk frequencies used must not be less than the minimum ic_clk frequencies specified in [Table 3-2](#).

The DW_apb_i2c coreConsultant GUI can automatically calculate SCL high and low count values. By specifying an integer ic_clk period value in nanoseconds for the IC_CLK_PERIOD parameter, SCL high and low count values are automatically calculated for each speed mode. The ic_clk period must not specify a clock of a lower frequency than required for all supported speed modes. It is possible that the automatically calculated values may result in a baud rate higher than the maximum rate specified by the protocol. If this happens, either the low or high count values can be scaled up to reduce the baud rate.

The minimum IC_CLK calculations for high-speed mode show how to do this; for details, refer to “[High-Speed Modes](#)” on page 62.

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:

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

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

260 ns for 1000 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

500 ns for 1000 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 = (ROUNDUP(MIN_SCL_HIGH_LOWtime*OSCFREQ,0))

IC_HCNT = (ROUNDUP(600 ns * 100 MHz,0))
IC_HCNTSCL PERIOD = 60
IC_LCNT = (ROUNDUP(1300 ns * 100 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

```



Note

Once the default values for SCL HighCount and LowCount are computed by the coreConsultant GUI, check that the values are consistent with the required baud rate. In case the computed values do not match with the required values, you can manually scale the values, as described in the section “[High-Speed Modes](#)” on page 62.

3.12 SDA Hold Time

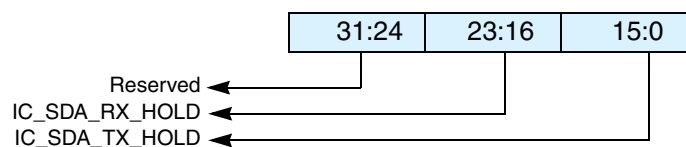
The I²C protocol specification requires 300ns of hold time on the SDA signal (t_{HD;DAT}) in standard mode and fast mode, and a hold time long enough to bridge the undefined part between logic 1 and logic 0 of the falling edge of SCL in high speed mode and fast mode plus.

Board delays on the SCL and SDA signals can mean that the hold-time requirement is met at the I²C master, but not at the I²C slave (or vice-versa). As each application encounters differing board delays, the DW_apb_i2c contains a software programmable register (IC_SDA_HOLD) to enable dynamic adjustment of the SDA hold-time.

The bits [15:0] are used to control the hold time of SDA during transmit in both slave and master mode (after SCL goes from HIGH to LOW).

The bits [23:16] are used to extend the SDA transition (if any) whenever SCL is HIGH in the receiver (in either master or slave mode).

Figure 3-26 IC_SDA_HOLD Register



If different SDA hold times are required for different speed modes, the IC_SDA_HOLD register must be reprogrammed when the speed mode is being changed. The IC_SDA_HOLD register can be programmed only when the DW_apb_i2c is disabled (IC_ENABLE[0] = 0).

The reset value of the IC_SDA_HOLD register can be set via the coreConsultant parameter IC_DEFAULT_SDA_HOLD

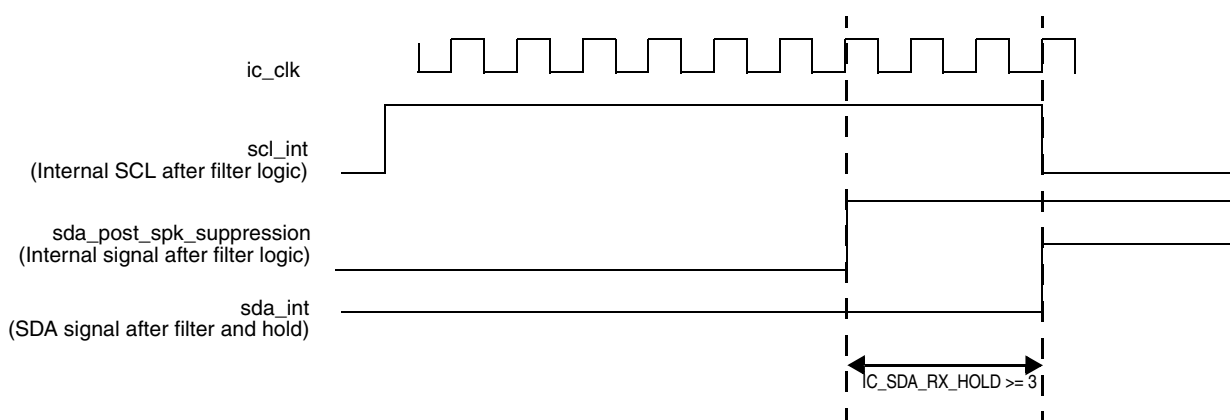
3.12.1 SDA Hold Timings in Receiver

When DW_apb_i2c acts as a receiver, according to the I²C protocol, the device should internally hold the SDA line to bridge undefined gap between logic 1 and logic 0 of SCL.

IC_SDA_RX_HOLD can be used to alter the internal hold time which DW_apb_i2c applies to the incoming SDA line. Each value in the IC_SDA_RX_HOLD register represents a unit of one ic_clk period. The minimum value of IC_SDA_RX_HOLD is 0. This hold time is applicable only when SCL is HIGH. The receiver does not extend the SDA after SCL goes LOW internally.

Figure 3-27 shows the DW_apb_i2c as receiver with IC_SDA_RX_HOLD programmed to greater than or equal to 3.

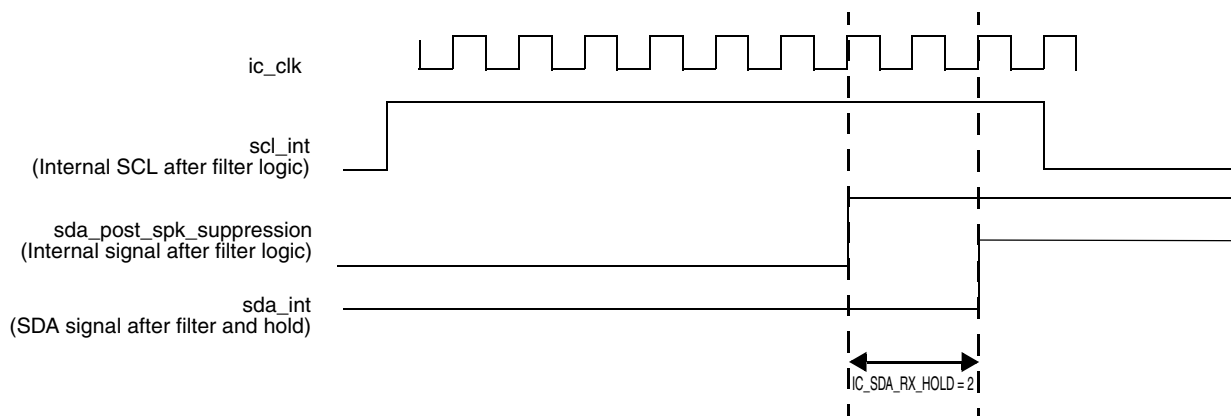
Figure 3-27



If IC_SDA_RX_HOLD is greater than 3, DW_apb_i2c does not hold SDA beyond 3 ic_clk cycles, because SCL goes LOW internally.

Figure 3-28 shows the DW_apb_i2c as receiver with IC_SDA_RX_HOLD programmed to 2.

Figure 3-28



The maximum values of IC_SDA_RX_HOLD that can be programmed in the register for the respective speed modes are derived from the equations show in [Table 3-3](#).

Table 3-3 Maximum Values for IC_SDA_RX_HOLD

Speed Mode	Maximum IC_SDA_RX_HOLD Value
Standard Mode	$IC_SS_SCL_HCNT - IC_FS_SPKLEN - 3$
Fast Mode or Fast Mode Plus	$IC_FS_SCL_HCNT - IC_FS_SPKLEN - 3$
High Speed (IC_CAP_LOADING =100)	$\text{Min} \{IC_FS_SCL_HCNT - IC_FS_SPKLEN - 3, IC_HS_SCL_LCNT - IC_HS_SPKLEN - 3\}$
High Speed (IC_CAP_LOADING =400)	$\text{Min} \{IC_FS_SCL_HCNT - IC_FS_SPKLEN - 3, (IC_HS_SCL_LCNT/2) - IC_HS_SPKLEN - 3\}$



Note

The maximum values in [Table 3-3](#) is applicable in Master mode. In Slave mode, make sure the IC_SDA_RX_HOLD does not exceed the maximum SCL fall time (tf in SS and FS mode or tfcl in HS Mode).

3.12.2 SDA Hold Timings in Transmitter

The IC_SDA_TX_HOLD register can be used to alter the timing of the generated SDA (ic_data_oe) signal by the DW_apb_i2c. Each value in the IC_SDA_TX_HOLD register represents a unit of one ic_clk period.

When the DW_apb_i2c is operating in Master Mode, the minimum tHD:DAT timing is one ic_clk period. Therefore even when IC_SDA_TX_HOLD has a value of zero, the DW_apb_i2c will drive SDA (ic_data_oe) one ic_clk cycle after driving SCL (ic_clk_oe) to logic 0. For all other values of IC_SDA_TX_HOLD, the following is true:

- Drive on SDA (ic_data_oe) occurs *IC_SDA_TX_HOLD* ic_clk cycles after driving SCL (ic_clk_oe) to logic 0

When the DW_apb_i2c is operating in Slave Mode, the minimum tHD:DAT timing is *SPKLEN* + 7 ic_clk periods, where *SPKLEN* is:

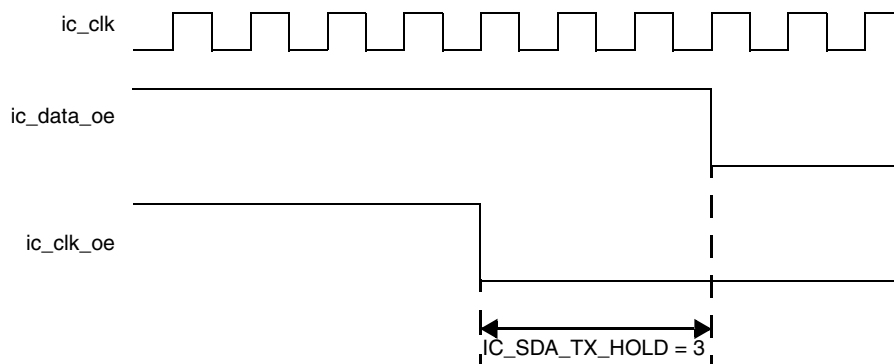
- IC_FS_SPKLEN if the component is operating in standard mode, fast mode, or fast mode plus
- IC_HS_SPKLEN if the component is operating in high speed mode

This delay allows for synchronization and spike suppression on the SCL (ic_clk_in_a) sample. Therefore, even when IC_SDA_TX_HOLD has a value less than *SPKLEN* + 7, the DW_apb_i2c drives SDA (ic_data_oe) *SPKLEN* + 7 ic_clk cycles after SCL (ic_clk_in) has transitioned to logic 0. For all other values of IC_SDA_TX_HOLD, the following is true:

- Drive on SDA (ic_data_oe) occurs *IC_SDA_TX_HOLD* ic_clk cycles after SCL (ic_clk_in_a) has transitioned to logic 0.

Figure 3-29 shows the tHD:DAT timing generated by the DW_apb_i2c operating in Master Mode when IC_SDA_TX_HOLD = 3.

Figure 3-29 DW_apb_i2c Master Implementing tHD:DAT with IC_SDA_HOLD = 3



Note

The programmed SDA hold time cannot exceed at any time the duration of the low part of scl. Therefore the programmed value cannot be larger than `N_SCL_LOW-2`, where `N_SCL_LOW` is the duration of the scl period measured in `ic_clk` cycles.

3.13 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, `CTLx`, 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.



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.

3.13.1 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.

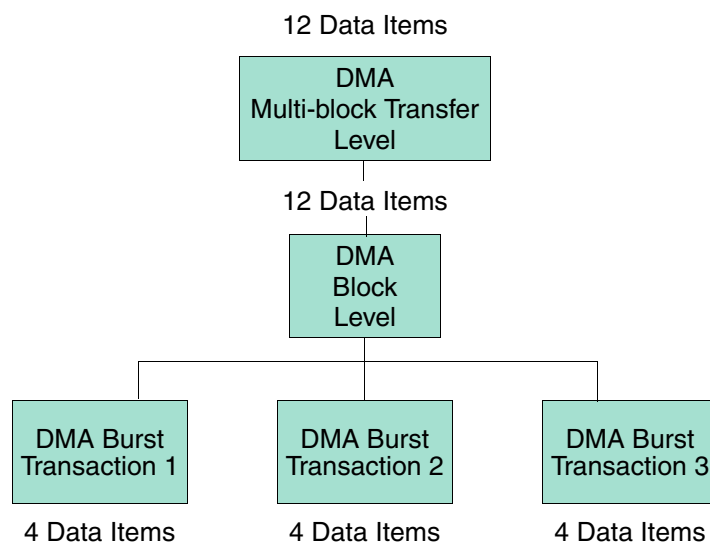
3.13.2 Overview of Operation

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_MSIZ/DEST_MSIZ fields of the DW_ahb_dmac CTLx register for source and destination, respectively.

Figure 3-30 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.

Figure 3-30 Breakdown of DMA Transfer into Burst Transactions



Block Size: DMA.CTLx.BLOCK_TS=12

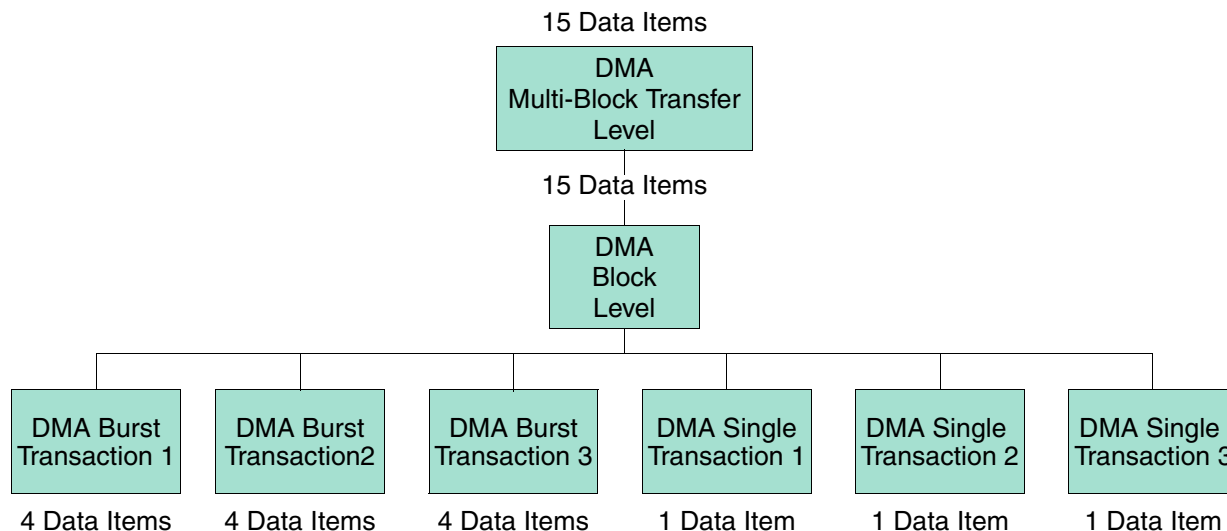
Number of data items per source burst transaction: DMA.CTLx.SRC_MSIZ = 4

I²C receive FIFO watermark level: I2C.DMARDLR + 1 = DMA.CTLx.SRC_MSIZ = 4

(for more information, refer to discussion on [page 72](#))

When the block size programmed into the DMA Controller is not a multiple of the burst transaction length, as shown in [Figure 3-31](#), a series of burst transactions followed by single transactions are needed to complete the block transfer.

Figure 3-31 Breakdown of DMA Transfer into Single and Burst Transactions



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 71](#))

3.13.3 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_MSIZ.

If IC_EMPTYFIFO_HOLD_MASTER_EN parameter is set to 0, 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.

3.13.4 Choosing the Transmit Watermark Level

Consider the example where the assumption is made:

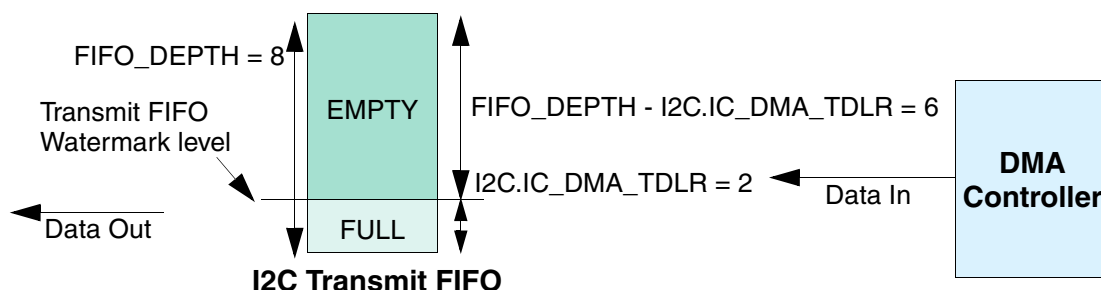
$$\text{DMA.CTLx.DEST_MSIZ} = \text{FIFO_DEPTH} - \text{I2C.IC_DMA_TDLR}$$

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.

3.13.4.1 Case 1: IC_DMA_TDLR = 2

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

Figure 3-32 Case 1 Watermark Levels



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

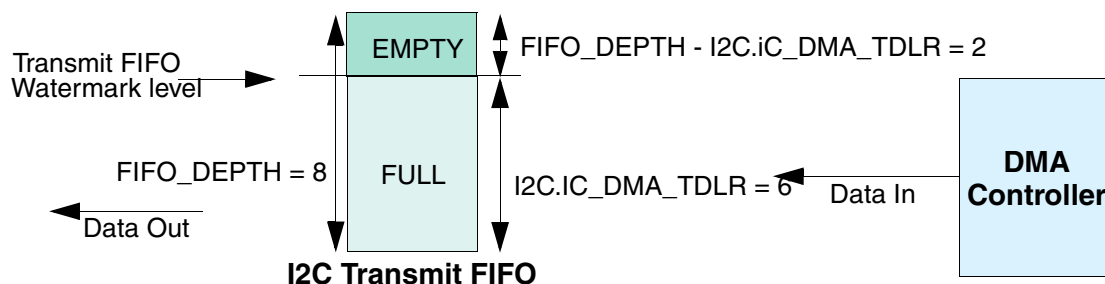
$$\text{DMA.CTLx.BLOCK_TS} / \text{DMA.CTLx.DEST_MSIZ} = 30 / 6 = 5$$

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.

3.13.4.2 Case 2: IC_DMA_TDLR = 6

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

Figure 3-33 Case 2 Watermark Levels



Number of burst transactions in Block:

$$\text{DMA.CTLx.BLOCK_TS} / \text{DMA.CTLx.DEST_MSIZ} = 30 / 2 = 15$$

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.

3.13.5 Selecting DEST_MSIZ and Transmit FIFO Overflow

As can be seen from [Figure 3-33](#) on page 70, programming DMA.CTLx.DEST_MSIZ 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:

$$\text{DMA.CTLx.DEST_MSIZ} \leq \text{I2C.FIFO_DEPTH} - \text{I2C.IC_DMA_TDLR} \quad (1)$$

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_MSIZ. Thus, the transmit FIFO may be full, but not overflowed, at the completion of the burst transaction.

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

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

This is the setting used in [Figure 3-31](#) on page 69.

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.

3.13.6 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 fetching a burst of data from the receive FIFO buffer of length CTLx.SRC_MSIZ.

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.

3.13.7 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 3-34. It is a trade-off between the number of DMA burst transactions required per block versus the probability of an overflow occurring.

3.13.8 Selecting SRC_MSIZ and Receive FIFO Underflow

As can be seen in Figure 3-34, 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, equation 3 below must be adhered to avoid underflow.

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

$$\text{DMA.CTLx.SRC_MSIZ} = \text{I2C.IC_DMA_RDLR} + 1 \quad (3)$$

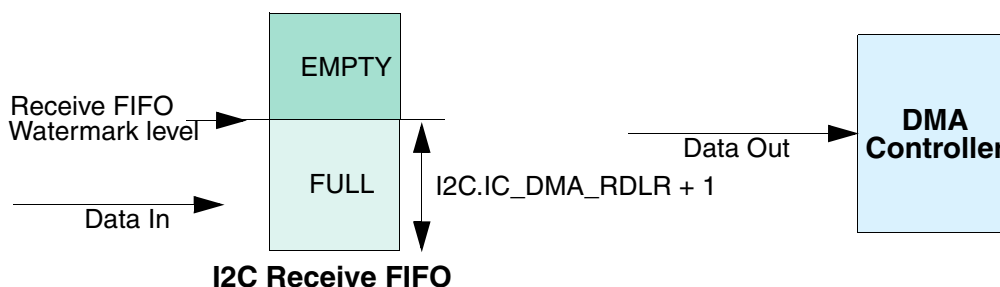
Adhering to equation (3) reduces the number of DMA bursts in a block transfer, which in turn can avoid underflow and improve AMBA bus utilization.



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 3-34 I²C Receive FIFO



3.13.9 Handshaking Interface Operation

The following sections discuss the handshaking interface.

3.13.9.1 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 3-35 shows a timing diagram of a burst transaction where pclk = hclk.

Figure 3-35 Burst Transaction – pclk = hclk

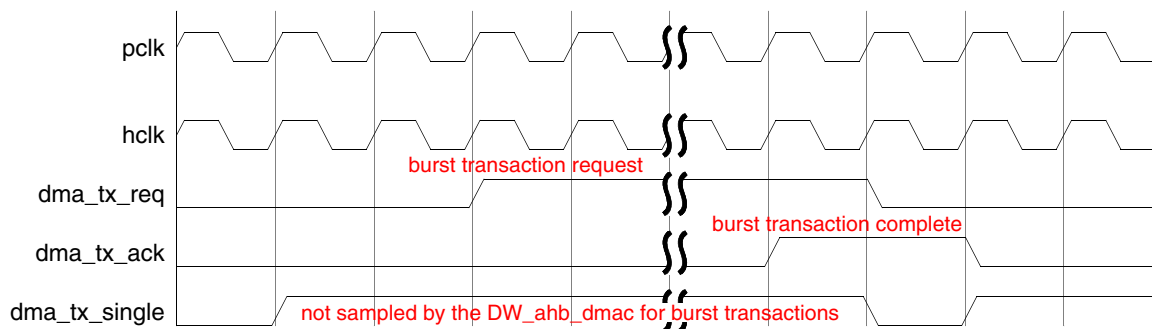
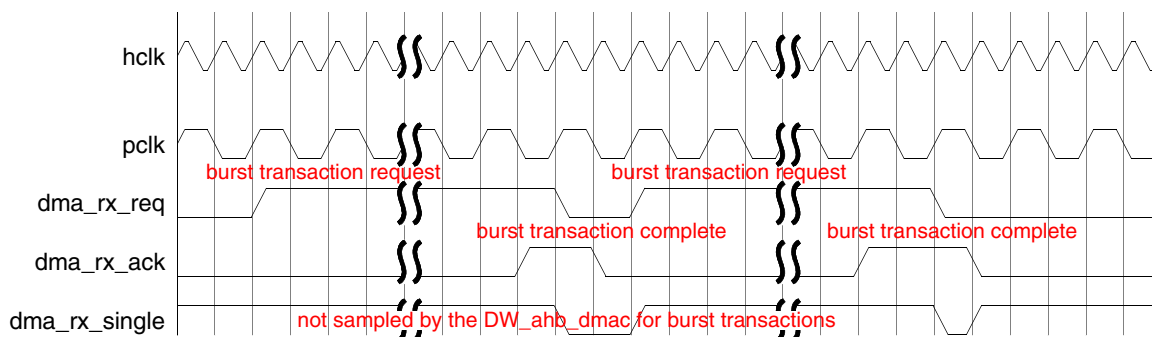


Figure 3-36 shows two back-to-back burst transactions where the hclk frequency is twice the pclk frequency.

Figure 3-36 Back-to-Back Burst Transactions – hclk = 2*pclk



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



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.

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.

3.13.9.2 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 3-31](#) on page 69. 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:

$$\text{DMA.CTLx.SRC_MSIZE} = \text{I2C.iC_DMA_RDLR} + 1 = 4$$

$$\text{DMA.CTLx.BLOCK_TS} = 12$$

For the example in [Figure 3-30](#) on page 68, 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:

$$\text{DMA.CTLx.SRC_MSIZE} = \text{I2C.IC_DMA_RDLR} + 1 = 4$$

$$\text{DMA.CTLx.BLOCK_TS} = 15$$

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 3-37 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 3-37 Single Transaction

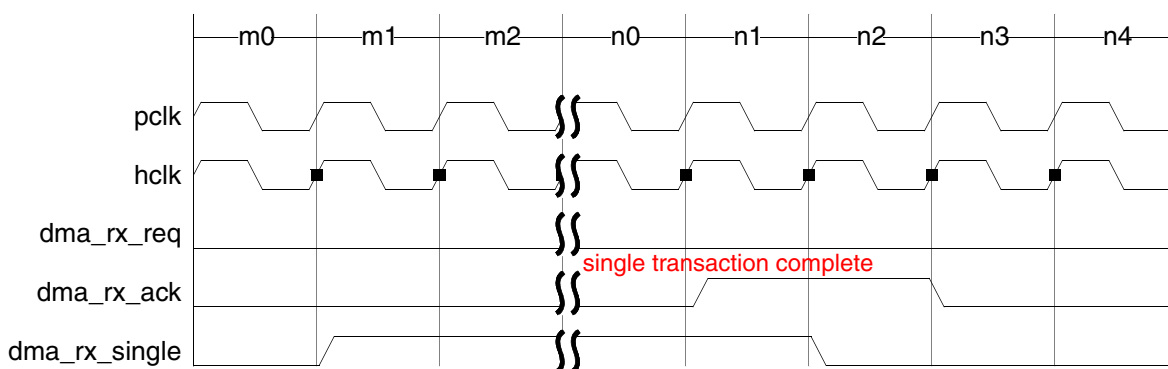
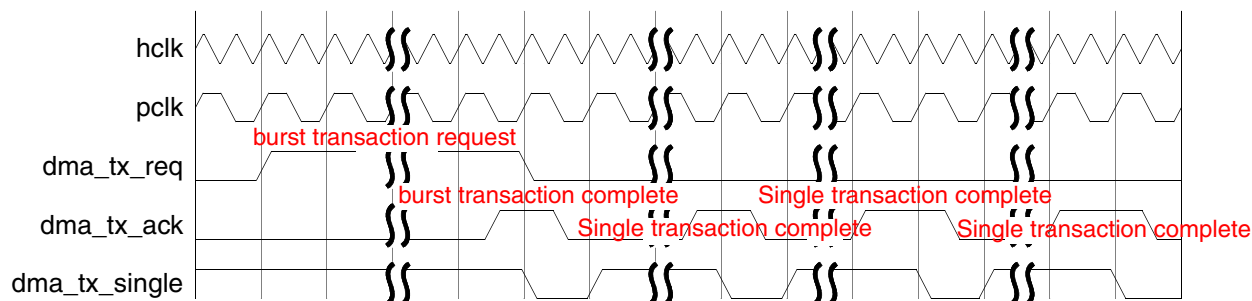


Figure 3-38 shows a burst transaction, followed by three back-to-back single transactions, where the hclk frequency is twice the pclk frequency.

Figure 3-38 Burst Transaction + 3 Back-to-Back Singles – $hclk = 2 \cdot pclk$



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.

3.14 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 185.

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.

4.1 Parameter Descriptions

You use coreConsultant or coreAssembler to configure the following parameters and generate the configured code.



Attention

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

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

4.2 Configuration Parameters

Table 4-1 lists the DW_apb_i2c parameter descriptions.

Table 4-1 Top-Level Parameters

coreConsultant Field Label	Parameter Definition
System Configuration	
APB data bus width	Parameter Name: APB_DATA_WIDTH Legal Values: 8, 16, or 32 Default Value: 8 Dependencies: None Description: Width of the APB data bus.

Table 4-1 Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
Device Configuration	
Highest speed I2C mode supported	<p>Parameter Name: IC_MAX_SPEED_MODE</p> <p>Legal Values: Standard Mode (1), Fast Mode or Fast Mode Plus (2), High Speed Mode (3)</p> <p>Default Value: High Speed Mode (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 speed modes faster than the configured maximum speed mode, 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>Legal 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>
Has I ² C default target slave address of?	<p>Parameter Name: IC_DEFAULT_TAR_SLAVE_ADDR</p> <p>Legal Values: 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 <i>IC_TAR</i> 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>Legal Values: 0x0 to 0x7</p> <p>Default Value: 0x1</p> <p>Dependencies: This parameter is enabled if IC_MAX_SPEED_MODE is set to High (3).</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 47.</p>

Table 4-1 Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
Is an I ² C master?	<p>Parameter Name: IC_MASTER_MODE</p> <p>Legal 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 (IC_CON). To enable the component to be a master, you must write a 1 in bit 0 of the IC_CON 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>Legal Values: Unchecked (0), Checked (1)</p> <p>Default Value: Checked (1)</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 (IC_CON). 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>Legal 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 (IC_CON). Master-generated transfers use this number of address bits. Additionally, it can be reprogrammed by software by writing to the IC_CON register.</p>
Supports 10-bit addressing in slave mode?	<p>Parameter Name: IC_10BITADDR_SLAVE</p> <p>Legal 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 IC_CON register. DW_apb_i2c responds to this number of address bits when acting as a slave; it can be programmed by software.</p>

Table 4-1 Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
Depth of transmit buffer is?	<p>Parameter Name: IC_TX_BUFFER_DEPTH</p> <p>Legal 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>Legal 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>Legal 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 (IC_TX_TL).</p>
Receive buffer threshold value is?	<p>Parameter Name: IC_RX_TL</p> <p>Legal 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 (IC_RX_TL).</p>
Allow restart conditions to be sent when acting as a master?	<p>Parameter Name: IC_RESTART_EN</p> <p>Legal 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 (IC_RESTART_EN) in the IC_CON 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 DW_apb_i2c master is incapable of performing the following functions:</p> <ul style="list-style-type: none"> ■ Sending a START BYTE ■ Performing any high-speed mode operation ■ Performing direction changes in combined format mode ■ Performing a read operation with a 10-bit address

Table 4-1 Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
Hardware reset value for IC_SDA_SETUP register	<p>Parameter Name: IC_DEFAULT_SDA_SETUP</p> <p>Legal Values: 0x02 to 0xff</p> <p>Default Value: 0x64</p> <p>Dependencies: None</p> <p>Description: Determines the reset value for the IC_SDA_SETUP register, which in turn controls the time delay—in terms of the number of ic_clk periods—introduced in the rising edge of SCL relative to SDA changing when a read-request is serviced.</p>
Hardware reset value for IC_SDA_HOLD register	<p>Parameter Name: IC_DEFAULT_SDA_HOLD</p> <p>Legal Values: 0x01 to 0xfffff</p> <p>Default Value: 0x1</p> <p>Dependencies: None</p> <p>Description: Determines the reset value for the IC_SDA_HOLD register, which in turn controls the SDA hold time implemented by DW_apb_i2c (when transmitting or receiving, as either master or slave). The relevant I2C requirement is tHD;DAT as detailed in the I2C Bus Specifications.</p> <p>The programmed SDA hold time as transmitter cannot exceed at any time the duration of the low part of scl. Therefore it is recommended that the configured default value should not be larger than N_SCL_LOW-2, where N_SCL_LOW is the duration of the low part of the scl period measured in ic_clk cycles, for the maximum speed mode the component is configured for.</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>Legal Values: 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>Legal 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 93.</p>
Single Interrupt output port present?	<p>Parameter Name: IC_INTR_IO</p> <p>Legal 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 4-1 Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
Polarity of interrupts is active high?	<p>Parameter Name: IC_INTR_POL</p> <p>Legal 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>Legal Values: Unchecked (0) or Checked (1)</p> <p>Default Value: Checked (1)</p> <p>Dependencies: None</p> <p>Description: 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.</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 IC_COMP_PARAM_1 register on page 168.</p> <p>Note: Unique drivers must be developed for each configuration of the DW_apb_i2c. Based on the configuration, the registers in the IP can differ; thus the same driver cannot be used with different configurations of the IP.</p>
Specify clock counts directly instead of supplying clock frequency?	<p>Parameter Name: IC_USE_COUNTS</p> <p>Legal 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 the standard mode, fast mode or fast mode plus, and 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> <p>Note: For fast mode plus, reprogram the IC_FS_SCL_*CNT register to achieve the required data rate when unchecked.</p>

Table 4-1 Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
Hard code the count values for each mode?	<p>Parameter Name: IC_HC_COUNT_VALUES</p> <p>Legal 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, fast mode plus, or high speed mode parameters later in this table). The count registers begin on page 124.</p> <p>Note: Since the DW_apb_i2c uses the same high and low count registers for fast mode and fast mode plus, if this parameter is checked (1) the IC_FS_SCL_*CNT registers are hard coded to either fast mode or fast mode plus. Consequently, DW_apb_i2c can operate in either fast mode or fast mode plus, but not in both modes simultaneously.</p> <p>For fast mode plus, it is recommended that this parameter be Unchecked (0).</p>
ic_clk has a period of? (ns integers only)	<p>Parameter Name: IC_CLOCK_PERIOD</p> <p>Legal 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> <p>Note: For fast mode plus, reprogram the IC_FS_SCL_*CNT register to achieve the required data rate.</p>
Relationship between pclk and ic_clk is?	<p>Parameter Name: IC_CLK_TYPE</p> <p>Legal Values: Identical (0), 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 used for data passing between clock domains.</p> <p>Asynchronous (1): clocks may be completely asynchronous to each other, metastability flops are used for data passing between clock domains.</p>

Table 4-1 Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
Standard Speed Mode Configuration	
Std speed SCL high count is?	<p>Parameter Name: IC_SS_SCL_HIGH_COUNT</p> <p>Legal 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.</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 IC_SS_SCL_HCNT register on page 124.</p>
Std speed SCL low count is?	<p>Parameter Name: IC_SS_SCL_LOW_COUNT</p> <p>Legal 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.</p> <p>Description: Reset value of Standard Speed I2C Clock SCL Low Count register (<i>IC_SS_SCL_LCNT</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 125. When parameter IC_USE_COUNTS = 0, this parameter is automatically calculated using the IC_CLK_PERIOD parameter.</p>
Fast Mode or Fast Mode Plus	
Fast Mode or Fast Mode Plus SCL high count is?	<p>Parameter Name: IC_FS_SCL_HIGH_COUNT</p> <p>Legal Values: Hex value in range 0x0006 to 0xffff</p> <p>Default Value: 0x003c (60 based on 100 MHz ic_clk in Fast Mode)</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,” this parameter is disabled.</p> <p>Description: Reset value of Fast Mode or Fast Mode Plus 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 126.</p>

Table 4-1 Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
Fast Mode or Fast Mode Plus SCL low count is?	<p>Parameter Name: IC_FS_SCL_LOW_COUNT</p> <p>Legal Values: Hex value in range 0x0008 to 0xffff</p> <p>Default Value: 0x0082 (130 based on 100 MHz ic_clk in Fast Mode)</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,” this parameter is disabled.</p> <p>Description: Reset value of Fast Mode or Fast Mode Plus 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 127.</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>Legal 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>
High speed SCL high count is?	<p>Parameter Name: IC_HS_SCL_HIGH_COUNT</p> <p>Legal Values: Hex value in range 0x0006 to 0xffff</p> <p>Default Value: 0x006 (6 based on 100 MHz ic_clk, 400 pF 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 128.</p>
High speed SCL low count is?	<p>Parameter Name: IC_HS_SCL_LOW_COUNT</p> <p>Legal Values: Hex value in range 0x0008 to 0xffff</p> <p>Default Value: 0x0010 (16 based on 100 MHz ic_clk, 400 pF 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 129.</p>

Table 4-1 Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
Spike Suppression Configuration	
Maximum length (in ic_clk cycles) of suppressed spikes in Standard Mode, Fast Mode, and Fast Mode Plus	<p>Parameter Name: IC_DEFAULT_FS_SPKLEN</p> <p>Legal Values: Hex value in range 0x01 to 0xFF</p> <p>Default Value: Max(1,ceiling(50/IC_CLOCK_PERIOD))</p> <p>Dependencies: Initial value automatically calculated using <i>IC_CLK_PERIOD</i>.</p> <p>Description: Reset value of maximum suppressed spike length register in Standard Mode, Fast Mode, and Fast Mode Plus (IC_FS_SPKLEN register). Spike length is expressed in <i>ic_clk</i> cycles, and this value is calculated based on the value of <i>IC_CLOCK_PERIOD</i></p>
Maximum length (in ic_clk cycles) of suppressed spikes in HS mode	<p>Parameter Name: IC_DEFAULT_HS_SPKLEN</p> <p>Legal Values: Hex value in range 0x01 to 0xFF</p> <p>Default Value: Max(1,ceiling(10/IC_CLOCK_PERIOD))</p> <p>Dependencies: Initial value automatically calculated using <i>IC_CLK_PERIOD</i>. If MAX_SPEED_MODE is set to standard or fast, this parameter is irrelevant.</p> <p>Description: Reset value of maximum suppressed spike length register in HS modes (IC_HS_SPKLEN register). Spike length is expressed in <i>ic_clk</i> cycles, and this value is calculated based on the value of <i>IC_CLOCK_PERIOD</i>.</p>
Additional Features	
Allow dynamic updating of the TAR address?	<p>Parameter Name: I2C_DYNAMIC_TAR_UPDATE</p> <p>Legal Values: Unchecked (0) or Checked (1)</p> <p>Default Value: Unchecked (0)</p> <p>Dependencies: None</p> <p>Description: When checked, allows the <i>IC_TAR</i> register to be updated dynamically. Setting this parameter affects the operation of DW_apb_i2c when it is in master mode. For more details, see “Master Mode Operation” on page 53.</p>
Enable register to generate NACKs for data received by Slave?	<p>Parameter Name: IC_SLV_DATA_NACK_ONLY</p> <p>Legal 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 <i>IC_SLV_DATA_NACK_ONLY</i> 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>

Table 4-1 Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
Hold transfer when Tx FIFO is empty?	<p>Parameter Name: IC_EMPTYFIFO_HOLD_MASTER_EN</p> <p>Legal Values: Unchecked (0) or Checked (1)</p> <p>Default Value: Unchecked (0)</p> <p>Dependencies: None</p> <p>Description: If this parameter is set, the master only completes a transfer—that is, issues a STOP—when it finds a Tx FIFO entry tagged with a Stop bit. If the Tx FIFO empties and the last byte does not have the Stop bit set, the master stalls the transfer by holding the SCL line low.</p> <p>If this parameter is not set, the master completes a transfer when the Tx FIFO is empty.</p>
When Rx FIFO is physically full, hold the bus till Rx FIFO has space available?	<p>Parameter Name: IC_RX_FULL_HLD_BUS_EN</p> <p>Legal Values: Unchecked (0), Checked (1)</p> <p>Default Value: Unchecked (0)</p> <p>Dependencies: None</p> <p>Description: When the Rx FIFO is physically full to its RX_BUFFER_DEPTH, this parameter provides a hardware method to hold the bus till the Rx FIFO data is read out and there is space available in the FIFO.</p> <p>This parameter can be used when DW_apb_i2c is either a slave-receiver (that is, data is written to the device) or a master-receiver (that is, the device reads data from a slave).</p> <p>NOTE: If this parameter is checked, then the RX_OVER interrupt is never set to 1 as the criteria to set this interrupt are never met. The RX_OVER interrupt can be found in the IC_INTR_STAT and IC_RAW_INTR_STAT registers. It is also an optional output signal, ic_rx_over_intr(_n).</p>
Enable restart detect interrupt in slave mode?	<p>Parameter Name: IC_SLV_RESTART_DET_EN</p> <p>Legal Values: Unchecked (0), Checked (1)</p> <p>Default Value: Unchecked (0)</p> <p>Dependencies: None</p> <p>Description: When checked, allows the slave to detect and issue the restart interrupt when the slave is addressed. Setting this parameter affects the operation of DW_apb_i2c only when it is in slave mode. This controls the RESTART_DET bit in the IC_RAW_INTR_STAT, IC_INTR_MASK, IC_INTR_STAT, and IC_CLR_RESTART_DET registers. This also controls the ic_restart_det_intr(_n) and ic_intr(_n) signals.</p>

Table 4-1 Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
Generate STOP_DET interrupt only if Master is active?	<p>Parameter Name: IC_STOP_DET_IF_MASTER_ACTIVE</p> <p>Legal Values: Unchecked (0), Checked (1)</p> <p>Default Value: Unchecked (0)</p> <p>Dependencies: None</p> <p>Description: Controls whether DW_apb_i2c generates a STOP_DET interrupt when the master is active.</p> <ul style="list-style-type: none"> ■ Checked (1): Allows the master to detect and issue the stop interrupt when the master is active. ■ Unchecked (0): The master always detects and issues the stop interrupt irrespective of whether it is active. <p>This parameter affects the operation of DW_apb_i2c when it is in master mode. This controls the STOP_DET bit of the C_RAW_INTR_STAT, IC_INTR_MASK, IC_INTR_STAT, and IC_CLR_STOP_DET registers. This parameter also controls the ic_stop_det_intr(_n) and ic_intr(_n) signals.</p>
Include Status bits to indicate the reason for clock stretching?	<p>Parameter Name: IC_STAT_FOR_CLK_STRETCH</p> <p>Legal Values: Unchecked (0), Checked (1)</p> <p>Default Value: Unchecked (0)</p> <p>Dependencies: None</p> <p>Description: If this parameter is set, the DW_apb_i2c consists of Status bits that indicate the reason for clock stretching in the IC_STATUS Register.</p>
Include programmable bit for blocking Master commands?	<p>Parameter Name: IC_TX_CMD_BLOCK</p> <p>Legal Values: Unchecked (0), Checked (1)</p> <p>Default Value: Unchecked (0)</p> <p>Dependencies: None</p> <p>Description: Controls whether DW_apb_i2c transmits data on I²C bus as soon as data is available in the Tx FIFO. When checked, allows the master to hold the transmission of data on the I²C bus when the Tx FIFO has data to transmit.</p>
Enable blocking Master commands after reset?	<p>Parameter Name: IC_TX_CMD_BLOCK_DEFAULT</p> <p>Legal Values: Unchecked (0), Checked (1)</p> <p>Default Value: Unchecked (0)</p> <p>Dependencies: IC_TX_CMD_BLOCK=1</p> <p>Description: Controls whether DW_apb_i2c has its transmit command block enabled or disabled after reset. If checked, the DW_apb_i2c blocks the transmission of data on the I²C bus.</p>

Table 4-1 Top-Level Parameters (Continued)

coreConsultant Field Label	Parameter Definition
Include First data byte indication in IC_DATA_CMD register?	<p>Parameter Name: IC_FIRST_DATA_BYTE_STATUS</p> <p>Legal Values: Unchecked (0), Checked (1)</p> <p>Default Value: Unchecked (0)</p> <p>Dependencies: None</p> <p>Description: Controls whether DW_apb_i2c generates the FIRST_DATA_BYTE status bit in the IC_DATA_CMD register. When checked, the master/slave receiver sets the FIRST_DATA_BYTE status bit in the IC_DATA_CMD register to indicate whether the data present in the IC_DATA_CMD register is the first data byte after the address phase of a receive transfer.</p> <p>Note: In the case when APB_DATA_WIDTH is set to 8, you must perform two APB reads to the IC_DATA_CMD register to get status on bit 11.</p>
Avoid Rx FIFO Flush on Transmit Abort?	<p>Parameter Name: IC_AVOID_RX_FIFO_FLUSH_ON_TX_ABRT</p> <p>Legal Values: Unchecked (0), Checked (1)</p> <p>Default Value: Unchecked (0)</p> <p>Dependencies: None</p> <p>Description: This Parameter controls the Rx FIFO Flush during the Transmit Abort. If this parameter is checked(1), only the Tx FIFO is flushed (not the Rx FIFO) on the Transmit Abort. If this parameter is unchecked(0), both the Tx FIFO and Rx FIFO are flushed on Transmit Abort.</p>

Table 4-2 includes parameters that are derived from the user selected parameters in coreConsultant.

Table 4-2 Derived Parameters

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 the table 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

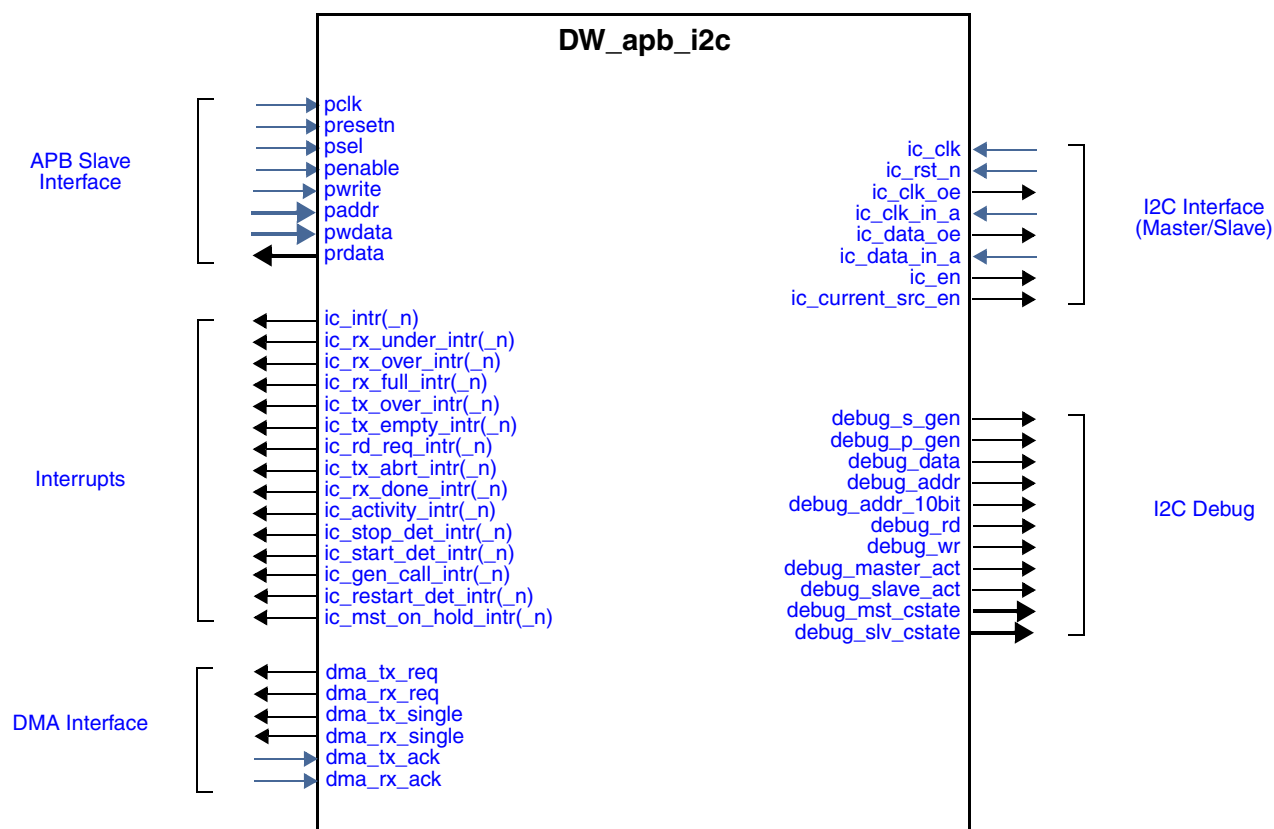
The following subsections describe the DW_apb_i2c I/O signals.

**Note**

There are references to both hardware parameters and software registers throughout this chapter. Both are 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.

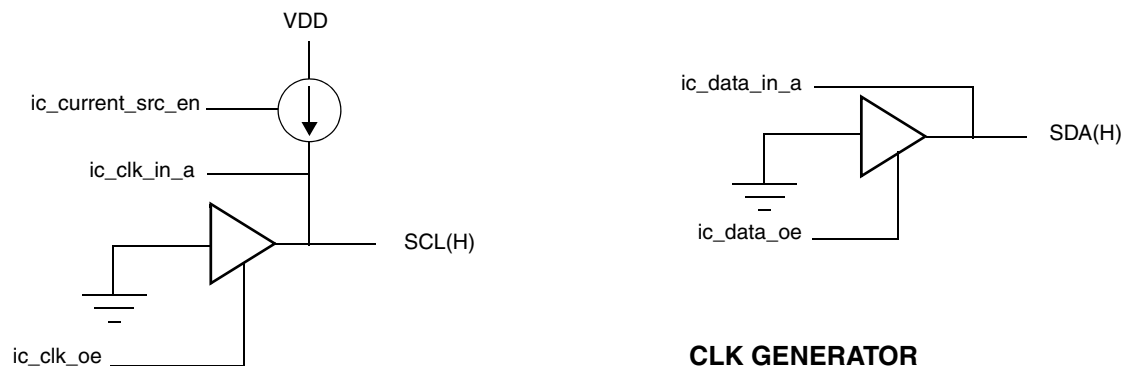
5.1 DW_apb_i2c Interface Diagram

[Figure 5-1](#) shows the interface diagram for DW_apb_i2c.

Figure 5-1 DW_apb_i2c Interface Diagram

5.2 I/O Connections

As illustrated in Figure 5-2, 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 APB slave interface side of the DW_apb_i2c.

Figure 5-2 I/O Connection to I²C Interface

5.3 DW_apb_i2c Signal Descriptions

Table 5-1 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.

Table 5-1 DW_apb_i2c Signal Description

Name	Width	I/O	Description
APB Slave Interface			
pclk	1 bit	In	<p>APB clock for the bus interface unit.</p> <p>NOTE: ic_clk frequency must be greater than or equal to pclk frequency.</p> <p>Active State: N/A</p> <p>Registered: N/A</p> <p>Synchronous to: The configuration parameter IC_CLK_TYPE indicates the relationship between pclk and ic_clk. It can be asynchronous (1) or identical (0). For more information about this parameter, refer to page 83.</p> <p>Default Input Delay: N/A</p>
presetn	1 bit	In	<p>An APB interface domain reset.</p> <p>Active State: Low</p> <p>Registered: N/A</p> <p>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.</p> <p>Default Input Delay: 40%</p>
psel	1 bit	In	<p>APB peripheral select that lasts for two pclk cycles. When asserted, indicates that the peripheral has been selected for a read/write operation.</p> <p>Active State: High</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>Default Input Delay: 40%</p>
penable	1 bit	In	<p>APB enable control. Asserted for a single pclk cycle and used for timing read/write operations.</p> <p>Active State: High</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>Default Input Delay: 40%</p>

Table 5-1 DW_apb_i2c Signal Description (Continued)

Name	Width	I/O	Description
pwrite	1 bit	In	<p>APB write control. When high, indicates a write access to the peripheral; when low, indicates a read access.</p> <p>Active State: N/A</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>Default Input Delay: 40%</p>
paddr	7 bits	In	<p>APB address bus. Uses lower 7 bits of the address bus for register decode.</p> <p>Active State: N/A</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>Default Input Delay: 40%</p>
pwdata	w-1:0	In	<p>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.</p> <p>Active State: N/A</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>Default Input Delay: 40%</p>
prdata	w-1:0	Out	<p>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.</p> <p>Active State: N/A</p> <p>Registered: Yes</p> <p>Synchronous to: pclk</p> <p>Default Output Delay: 60%</p>
I²C Interface (Master/Slave)			
ic_clk	1 bit	In	<p>Peripheral clock. DW_apb_i2c runs on this clock and is used to clock transfers in standard, fast, and high-speed mode.</p> <p>NOTE: ic_clk frequency must be greater than or equal to pclk frequency.</p> <p>Active State: N/A</p> <p>Registered: N/A</p> <p>Synchronous to: The configuration parameter IC_CLK_TYPE indicates the relationship between pclk and ic_clk. It can be asynchronous (1) or identical (0). For more information about this parameter, see page 83.</p> <p>Default Input Delay: N/A</p>

Table 5-1 DW_apb_i2c Signal Description (Continued)

Name	Width	I/O	Description
ic_rst_n	1 bit	In	<p>I²C reset. Used to reset flip-flops that are clocked by the ic_clk clock.</p> <p>NOTE: This signal does not reset DW_apb_i2c control, configuration, and status registers.</p> <p>Active State: Low</p> <p>Registered: N/A</p> <p>Synchronous to: The signal is asserted asynchronously, but is deasserted synchronously after the rising edge of ic_clk. The synchronization must be provided external to this component.</p> <p>Default Input Delay: 40%</p>
ic_clk_oe	1 bit	Out	<p>Outgoing I²C clock. Open drain synchronous with ic_clk.</p> <p>Active State: High</p> <p>Registered: Yes</p> <p>Synchronous to: ic_clk</p> <p>Default Output Delay: 30%</p>
ic_clk_in_a	1 bit	In	<p>Incoming I²C clock. This is the input SCL signal. Double-registered for metastability synchronization and glitch-suppressed using 2-out-of-3 majority vote circuit.</p> <p>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.</p> <p>Active State: High</p> <p>Registered: Yes</p> <p>Synchronous to: This signal is asynchronous to ic_clk.</p> <p>Default Input Delay: N/A</p>
ic_data_oe	1 bit	Out	<p>Outgoing I²C Data. Open Drain Synchronous to ic_clk.</p> <p>Active State: High</p> <p>Registered: Yes</p> <p>Synchronous to: ic_clk</p> <p>Default Output Delay: 30%</p>
ic_data_in_a	1 bit	In	<p>Incoming I²C Data. It is the input SDA signal. Double-registered for metastability synchronization and glitch-suppressed using 2-out-of-3 majority vote circuit.</p> <p>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.</p> <p>Active State: High</p> <p>Registered: Yes</p> <p>Synchronous to: This signal is asynchronous to ic_clk.</p> <p>Default Input Delay: N/A</p>

Table 5-1 DW_apb_i2c Signal Description (Continued)

Name	Width	I/O	Description
ic_en	1 bit	Out	<p>I²C interface enable. Indicates whether DW_apb_i2c is enabled; this signal is set to 0 when IC_ENABLE[0] 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.</p> <p>Active State: Low</p> <p>Registered: Yes</p> <p>Synchronous to: pclk</p> <p>Default Output Delay: 60%</p>
ic_current_src_en	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: High</p> <p>Registered: Yes</p> <p>Synchronous to: ic_clk</p> <p>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 IC_MAX_SPEED_MODE = high.</p>
Interrupts			
ic_intr(_n)	1 bit	Out	<p><i>Optional.</i> Combined interrupt. This signal is included on the interface when the configuration parameter IC_INTR_IO 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 IC_INTR_POL (checked = active high). When IC_INTR_POL is unchecked (0), the ic_intr_n signal is included on the interface to indicate active low polarity.</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>Default Output Delay: 30%</p>

Table 5-1 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>Registered: Yes</p> <p>Synchronous to: pclk</p> <p>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 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_over_intr_n signal is included on the interface to indicate active low polarity.</p> <p>Registered: Yes</p> <p>Synchronous to: pclk</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 ic_rx_full_intr bit is cleared once the ic_enable 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 ic_rx_full_intr_n signal is included on the interface instead to indicate active low polarity.</p> <p>Registered: Yes</p> <p>Synchronous to: pclk</p> <p>Default Output Delay: 30%</p>

Table 5-1 DW_apb_i2c Signal Description (Continued)

Name	Width	I/O	Description
<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>Registered: Yes</p> <p>Synchronous to: pclk</p> <p>Default Output Delay: 30%</p>
<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>Registered: Yes</p> <p>Synchronous to: pclk</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 ic_rd_req_intr_n signal is included on the interface instead to indicate active low polarity.</p> <p>Registered: Yes</p> <p>Synchronous to: pclk</p> <p>Default Output Delay: 30%</p>

Table 5-1 DW_apb_i2c Signal Description (Continued)

Name	Width	I/O	Description
<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 <i>ic_tx_abrt_intr_n</i> signal is included on the interface instead to indicate active low polarity.</p> <p>Registered: Yes</p> <p>Synchronous to: pclk</p> <p>Default Output Delay: 30%</p>
<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>Registered: Yes</p> <p>Synchronous to: pclk</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 <i>ic_activity_intr_n</i> signal is included on the interface instead to indicate active low polarity.</p> <p>Registered: Yes</p> <p>Synchronous to: pclk</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 <i>ic_stop_det_intr_n</i> signal is included on the interface instead to indicate active low polarity.</p> <p>Registered: Yes</p> <p>Synchronous to: pclk</p> <p>Default Output Delay: 30%</p>

Table 5-1 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>Registered: Yes</p> <p>Synchronous to: pclk</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>Registered: Yes</p> <p>Synchronous to: pclk</p> <p>Default Output Delay: 30%</p>
<i>ic_restart_det_intr(_n)</i>	1 bit	Out	<p><i>Optional.</i> Restart 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_restart_det_intr_n</i> signal is included on the interface instead to indicate active low polarity.</p> <p>Registered: Yes</p> <p>Synchronous to: pclk</p> <p>Default Output Delay: 30%</p> <p>Dependencies: This interrupt is present only when IC_SLV_RESTART_DET_EN is checked (1).</p>

Table 5-1 DW_apb_i2c Signal Description (Continued)

Name	Width	I/O	Description
ic_mst_on_hold_intr(_n)	1 bit	Out	<p><i>Optional.</i> Master on hold I2C interrupt. This signal is included on the interface when the configuration parameters I2C_DYNAMIC_TAR_UPDATE and IC_EMPTYFIFO_HOLD_MASTER_EN are checked (1) and the configuration parameter IC_INTR_IO is unchecked (0), indicating 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_mst_on_hold_intr_n signal is included on the interface instead to indicate active low polarity.</p> <p>Registered: Yes</p> <p>Synchronous to: pclk</p> <p>Default Output Delay: 30%</p>
DMA Interface (only present when configured with DMA interface) refer to “DMA Controller Interface” on page 67			
dma_tx_req	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 DEST_MSIZE field of the CTLx register.</p> <p>Active State: High</p> <p>Registered: Yes</p> <p>Synchronous to: pclk</p> <p>Default Output Delay: 30%</p>
dma_rx_req	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 SRC_MSIZE field of the CTLx register.</p> <p>Active State: High</p> <p>Registered: Yes</p> <p>Synchronous to: pclk</p> <p>Default Output Delay: 30%</p>

Table 5-1 DW_apb_i2c Signal Description (Continued)

Name	Width	I/O	Description
dma_tx_single	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 Registered: Yes Synchronous to: pclk Default Output Delay: 30%</p>
dma_rx_single	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 Registered: Yes Synchronous to: pclk Default Output Delay: 30%</p>
dma_tx_ack	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 Registered: No Synchronous to: pclk Default Input Delay: 70%</p>
dma_rx_ack	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 Registered: No Synchronous to: pclk Default Input Delay: 70%</p>
I²C Debug			
debug_s_gen	1 bit	Out	<p>In the master mode of operation, this signal is set to 1 when DW_apb_i2c is driving a START condition on the bus.</p> <p>Active State: Low Registered: Yes Synchronous to: ic_clk Default Output Delay: N/A</p>

Table 5-1 DW_apb_i2c Signal Description (Continued)

Name	Width	I/O	Description
debug_p_gen	1 bit	Out	<p>In the master mode of operation, this signal is set to 1 when DW_apb_i2c is driving a STOP condition on the bus.</p> <p>Active State: Low</p> <p>Registered: Yes</p> <p>Synchronous to: ic_clk</p> <p>Default Output Delay: N/A</p>
debug_data	1 bit	Out	<p>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.</p> <p>Active State: High</p> <p>Registered: Yes</p> <p>Default Output Delay: N/A</p>
debug_addr	1 bit	Out	<p>In the master or slave mode of operation, this signal is set to 1 when the addressing phase is active on the I²C bus.</p> <p>Active State: High</p> <p>Synchronous to: ic_clk</p> <p>Registered: Yes</p> <p>Synchronous to: ic_clk</p> <p>Default Output Delay: N/A</p>
debug_addr_10bit	1 bit	Out	<p>In the master or slave mode of operation, this signal is set to 1 after a 10-bit address code has been detected.</p> <p>Active State: High</p> <p>Registered: Yes</p> <p>Synchronous to: ic_clk</p> <p>Default Output Delay: N/A</p>
debug_rd	1 bit	Out	<p>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.</p> <p>Active State: High</p> <p>Registered: Yes</p> <p>Synchronous to: ic_clk</p> <p>Default Output Delay: N/A</p>
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</p> <p>Registered: Yes</p> <p>Synchronous to: ic_clk</p> <p>Default Output Delay: N/A</p>

Table 5-1 DW_apb_i2c Signal Description (Continued)

Name	Width	I/O	Description
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 Registered: Yes Synchronous to: ic_clk 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 Registered: Yes Synchronous to: ic_clk 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 Registered: Yes Synchronous to: ic_clk Default Output Delay: N/A</p>
debug_mst_cstate	5 bits	Out	<p>Master FSM state vector.</p> <p>Active State: N/A Registered: Yes Synchronous to: ic_clk Default Output Delay: N/A</p>
debug_slv_cstate	3 bit	Out	<p>Slave FSM state vector.</p> <p>Active State: N/A Registered: Yes Synchronous to: ic_clk Default Output Delay: N/A</p>

6

Registers

This section describes the programmable registers of the DW_apb_i2c.



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.

6.1 Register Memory Map



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 6-1 provides the details of the DW_apb_i2c memory map. Reset values are affected by the configuration parameters specified in Table 4-1 on page 77.

Table 6-1 Memory Map of DW_apb_i2c

Name	Address Offset	Width	R/W	Description
IC_CON	0x00	11 bits	R/W or R-only on bit 4, bit 9, and bit 10	<p>I²C Control</p> <p>R/W:</p> <ul style="list-style-type: none"> I2C_DYNAMIC_TAR_UPDATE=1, bit 4 is read only. IC_RX_FULL_HLD_BUS_EN =0, bit 9 is read only. IC_STOP_DET_IF_MASTER_ACTIVE =0, bit 10 is read only. <p>Reset Value:</p> <p>10: 0x0 9: 0x0 8: 0x0 7: 0x0 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</p>
IC_TAR	0x04	12 or 13 bits	R/W	<p>I²C Target Address</p> <p>Width: 13, if I2C_DYNAMIC_TAR_UPDATE = 1 12, if I2C_DYNAMIC_TAR_UPDATE = 0</p> <p>Reset Value: Reset values for the four bit fields correspond to the following:</p> <p>12: IC_10BITADDR_MASTER configuration parameter 11: 0x0 10: 0x0 9: IC_DEFAULT_TAR_SLAVE_ADDR</p>
IC_SAR	0x08	10 bits	R/W	<p>I²C Slave Address</p> <p>Reset Value: IC_DEFAULT_SLAVE_ADDR</p>
IC_HS_MADDR	0x0C	3 bits	R/W	<p>I²C HS Master Mode Code Address</p> <p>Reset Value: IC_HS_MASTER_CODE</p>

Table 6-1 Memory Map of DW_apb_i2c

Name	Address Offset	Width	R/W	Description
IC_DATA_CMD	0x10	Refer to Description	R/W	<p>I²C Rx/Tx Data Buffer and Command</p> <p>Reset Value: 0x0</p> <p>Width:</p> <p>Write:</p> <ul style="list-style-type: none"> 11 bits when IC_EMPTYFIFO_HOLD_MASTER_EN=1 9 bits when IC_EMPTYFIFO_HOLD_MASTER_EN=0 <p>Read:</p> <ul style="list-style-type: none"> 12 bits when IC_FIRST_DATA_BYTE_STATUS =1 8 bits when IC_FIRST_DATA_BYTE_STATUS = 0 <p>Notes:</p> <ul style="list-style-type: none"> With nine or eleven 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. In order for the DW_apb_i2c to continue acknowledging reads, a read command should be written for every byte that is to be received; otherwise the DW_apb_i2c will stop acknowledging.
IC_SS_SCL_HCNT	0x14	16 bits	R/W	<p>Standard speed I²C Clock SCL High Count</p> <p>Reset Value: IC_SS_SCL_HIGH_COUNT</p>
IC_SS_SCL_LCNT	0x18	16 bits	R/W	<p>Standard speed I²C Clock SCL Low Count</p> <p>Reset Value: IC_SS_SCL_LOW_COUNT</p>
IC_FS_SCL_HCNT	0x1C	16 bits	R/W	<p>Fast Mode and Fast Mode Plus I²C Clock SCL High Count</p> <p>Reset Value: IC_FS_SCL_HIGH_COUNT</p>
IC_FS_SCL_LCNT	0x20	16 bits	R/W	<p>Fast Mode and Fast Mode Plus I²C Clock SCL Low Count</p> <p>Reset Value: IC_FS_SCL_LOW_COUNT</p>
IC_HS_SCL_HCNT	0x24	16 bits	R/W	<p>High speed I²C Clock SCL High Count</p> <p>Reset Value: IC_HS_SCL_HIGH_COUNT</p>
IC_HS_SCL_LCNT	0x28	16 bits	R/W	<p>High speed I²C Clock SCL Low Count</p> <p>Reset Value: IC_HS_SCL_LOW_COUNT</p>

Table 6-1 Memory Map of DW_apb_i2c

Name	Address Offset	Width	R/W	Description
IC_INTR_STAT	0x2C	14 bits	R	I ² C Interrupt Status Reset Value: 0x0
IC_INTR_MASK	0x30	14 bits	R/W or Read-only on bits 12 and 13	I ² C Interrupt Mask Reset Value: 14'h8ff
IC_RAW_INTR_STAT	0x34	14 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
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

Table 6-1 Memory Map of DW_apb_i2c

Name	Address Offset	Width	R/W	Description
IC_CLR_GEN_CALL	0x68	1 bit	R	Clear GEN_CALL Interrupt Reset Value: 0x0
IC_ENABLE	0x6C	Refer to Description	R/W	I ² C Enable Width: <ul style="list-style-type: none"> 2 bits if IC_TX_CMD_BLOCK = 0 3 bits if IC_TX_CMD_BLOCK = 1 Reset Value: 0x0
IC_STATUS	0x70	Refer to Description	R	I ² C Status register Width: <ul style="list-style-type: none"> 7 bits if IC_STAT_FOR_CLK_STRETCH = 0 11 bits if IC_STAT_FOR_CLK_STRETCH = 1 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
IC_SDA_HOLD	0x7C	24 bits	R/W	SDA hold time length register Reset Value: IC_DEFAULT_SDA_HOLD
IC_TX_ABRT_SOURCE	0x80	32 bits	R	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
IC_SDA_SETUP	0x94	8 bits	R/W	I ² C SDA Setup Register Reset Value: IC_DEFAULT_SDA_SETUP configuration parameter

Table 6-1 Memory Map of DW_apb_i2c

Name	Address Offset	Width	R/W	Description
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_FS_SPKLEN	0xA0	8 bits	R/W	ISS and FS spike suppression limit Reset Value: IC_DEFAULT_FS_SPKLEN configuration parameter
IC_HS_SPKLEN	0xA4	8 bits	R/W	HS spike suppression limit Reset Value: IC_DEFAULT_HS_SPKLEN configuration parameter
IC_CLR_RESTART_DET	0xA8	1 bit	R	Clear RESTART_DET Interrupt 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 4-1 on page 77.
IC_COMP_VERSION	0xf8	32 bits	R	Component Version ID Reset Value: See the releases table in the AMBA 2 release notes
IC_COMP_TYPE	0xfc	32 bits	R	DesignWare Component Type Register Reset Value: 0x44570140

6.2 Operation of Interrupt Registers

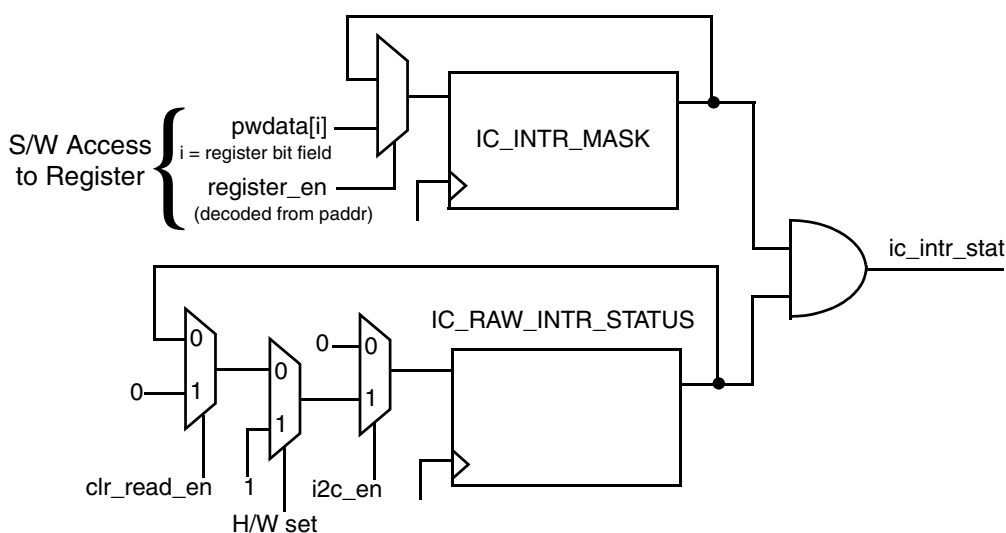
Table 6-2 lists the operation of the DW_apb_i2c interrupt registers and how they are set and cleared. Some bits are set by hardware and cleared by software, whereas other bits are set and cleared by hardware.

Table 6-2 Clearing and Setting of Interrupt Registers

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

Figure 6-1 shows the operation of the interrupt registers where the bits are set by hardware and cleared by software.

Figure 6-1 Interrupt Scheme



6.3 Registers and Field Descriptions

This section describes the registers listed in [Table 6-1](#) on page 106. 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.

6.3.1 IC_CON

- **Name:** I²C Control Register
- **Size:** 10 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 IC_ENABLE[0] being set to 0. Writes at other times have no effect.

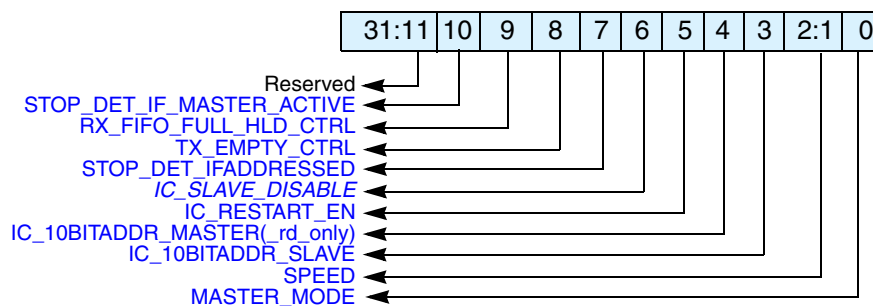


Table 6-3 IC_CON Register Fields

Bits	Name	R/W	Description
31:11	Reserved	N/A	Reserved
10	STOP_DET_IF_MASTER_ACTIVE	R/W	<p>In Master mode</p> <ul style="list-style-type: none"> 1'b1: Issues the STOP_DET interrupt only when the master is active 1'b0: Issues the STOP_DET irrespective of whether the master is active <p>Reset value: 1'b0</p> <p>Dependencies: This Register bit value is applicable only when IC_STOP_DET_IF_MASTER_ACTIVE=1.</p>
9	RX_FIFO_FULL_HLD_CTRL	R/W or R	<p>This bit controls whether DW_apb_i2c should hold the bus when the Rx FIFO is physically full to its RX_BUFFER_DEPTH, as described in the IC_RX_FULL_HLD_BUS_EN parameter.</p> <p>Dependencies: This register bit value is applicable only when the IC_RX_FULL_HLD_BUS_EN configuration parameter is set to 1. If IC_RX_FULL_HLD_BUS_EN = 0, then this bit is read-only. If IC_RX_FULL_HLD_BUS_EN = 1, then this bit can be read or write.</p> <p>Reset value: 0x0</p>
8	TX_EMPTY_CTRL	R/W	<p>This bit controls the generation of the TX_EMPTY interrupt, as described in the IC_RAW_INTR_STAT register.</p> <p>Reset value: 0x0</p>
7	STOP_DET_IFADDRESSED	R/W	<p>In slave mode:</p> <p>1'b1 – issues the STOP_DET interrupt only when it is addressed.</p> <p>1'b0 – issues the STOP_DET irrespective of whether it's addressed or not.</p> <p>Dependencies: This register bit value is applicable in the slave mode only (MASTER_MODE = 1'b0)</p> <p>Reset value: 1'b0</p> <p>NOTE: During a general call address, this slave does not issue the STOP_DET interrupt if STOP_DET_IF_ADDRESSED = 1'b1, even if the slave responds to the general call address by generating ACK. The STOP_DET interrupt is generated only when the transmitted address matches the slave address (SAR).</p>

Table 6-3 IC_CON Register Fields (Continued)

Bits	Name	R/W	Description
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 IC_SLAVE_DISABLE. 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), DW_apb_i2c 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: IC_SLAVE_DISABLE 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 DW_apb_i2c operations.</p> <p>0: disable 1: enable</p> <p>When the RESTART is disabled, the DW_apb_i2c master is incapable of performing the following functions:</p> <ul style="list-style-type: none"> ■ Sending a START BYTE ■ Performing any high-speed mode operation ■ Performing direction changes in combined format mode ■ Performing a read operation with a 10-bit address <p>By replacing RESTART condition followed by a STOP and a subsequent START condition, split operations are broken down into multiple DW_apb_i2c transfers. If the above operations are performed, it will result in setting bit 6 (<i>TX_ABORT</i>) of the <i>IC_RAW_INTR_STAT</i> register.</p> <p>Reset value: IC_RESTART_EN configuration parameter</p>

Table 6-3 IC_CON Register Fields (Continued)


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 I2C_DYNAMIC_TAR_UPDATE 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 I2C_DYNAMIC_TAR_UPDATE 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 I2C_DYNAMIC_TAR_UPDATE = 1, then this bit is read-only. If I2C_DYNAMIC_TAR_UPDATE = 0, then this bit can be read or write.</p> <p>Reset value: IC_10BITADDR_MASTER 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 IC_SAR 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 IC_SAR register.</p> <p>Reset value: IC_10BITADDR_SLAVE configuration parameter</p>
 Note 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.			
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 IC_MAX_SPEED_MODE; otherwise, hardware updates this register with the value of IC_MAX_SPEED_MODE.</p> <ul style="list-style-type: none"> ■ 1: standard mode (0 to 100 Kb/s) ■ 2: fast mode (≤ 400 Kb/s) or fast mode plus (≤ 1000 Kb/s) ■ 3: high speed mode (≤ 3.4 Mb/s) <p>Reset value: IC_MAX_SPEED_MODE configuration</p>

Table 6-3 IC_CON Register Fields (Continued)

Bits	Name	R/W	Description
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: IC_MASTER_MODE 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>

Certain combinations of the IC_SLAVE_DISABLE (bit 6) and MASTER_MODE (bit 0) result in a configuration error. [Table 6-4](#) lists the states that result from the combinations of these two bits.

Table 6-4 States for IC_SLAVE_DISABLE (bit 6) and MASTER_MODE (bit 0)

IC_SLAVE_DISABLE (IC_CON[6])	MASTER_MODE IC_CON[0]	State
0	0	Slave Device
0	1	Config Error
1	0	Config Error
1	1	Master Device

**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 112 register. In particular, IC_SLAVE_DISABLE and IC_MASTER_MODE must not be set to '0' and '1,' respectively at any given time.

6.3.2 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 31:12 are reserved. Writes to this register succeed only when IC_ENABLE[0] is set to 0.

However, if I2C_DYNAMIC_TAR_UPDATE = 1, then the register becomes 13 bits wide. In this case, writes to IC_TAR succeed when one of the following conditions are true:

- DW_apb_i2c is NOT enabled (IC_ENABLE[0] is set to 0); or
- DW_apb_i2c is enabled (IC_ENABLE[0]=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)¹

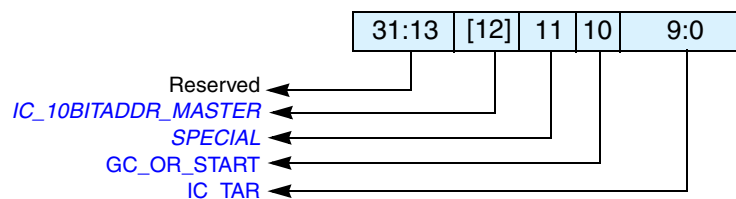


Table 6-5 IC_TAR Register Fields

Bits	Name	R/W	Description
31:13	Reserved	N/A	Reserved
12	<i>IC_10BITADDR_MASTER</i>	R/W	<p>This bit controls whether the DW_apb_i2c starts its transfers in 7-or 10-bit addressing mode when acting as a master.</p> <ul style="list-style-type: none"> ■ 0: 7-bit addressing ■ 1: 10-bit addressing <p>Dependencies: This bit exists in this register only if the I2C_DYNAMIC_TAR_UPDATE configuration parameter is set to Yes (1).</p> <p>Reset value: IC_10BITADDR_MASTER configuration parameter</p>
11	<i>SPECIAL</i>	R/W	<p>This bit indicates whether software performs a General Call or START BYTE command.</p> <ul style="list-style-type: none"> ■ 0: ignore bit 10 <i>GC_OR_START</i> and use IC_TAR normally ■ 1: perform special I²C command as specified in <i>GC_OR_START</i> bit <p>Reset value: 0x0</p>

1. If the software or application is aware the the DW_apb_i2c is not using the TAR address for the pending commands in the Tx FIFO, then it is possible to update the TAR address even while the Tx FIFO has entries (IC_STATUS[2]=0).

Table 6-5 IC_TAR Register Fields (Continued)

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> <ul style="list-style-type: none"> 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_ABORT) 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. 1: START BYTE <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.

6.3.3 IC_SAR

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

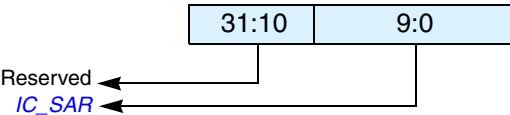


Table 6-6 IC_SAR Register Fields

Bits	Name	R/W	Description
31:10	Reserved	N/A	Reserved
9:0	IC_SAR	R/W	<p>The IC_SAR holds the slave address when the I²C is operating as a slave. For 7-bit addressing, only IC_SAR[6:0] is used.</p> <p>This register can be written only when the I²C interface is disabled, which corresponds to IC_ENABLE[0] 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 IC_SAR or IC_TAR to a reserved value. Refer to Table 3-1 on page 39 for a complete list of these reserved values.</p> <p>Reset value: IC_DEFAULT_SLAVE_ADDR 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.

6.3.4 IC_HS_MADDR

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

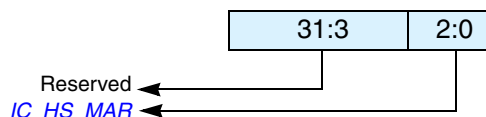


Table 6-7 IC_HS_MADDR Register Fields

Bits	Name	R/W	Description
31:3	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 IC_MAX_SPEED_MODE 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 IC_ENABLE[0] being set to 0. Writes at other times have no effect.</p> <p>Reset value: IC_HS_MASTER_CODE 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.

6.3.5 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



Note

In order for the DW_apb_i2c to continue acknowledging reads, a read command should be written for every byte that is to be received; otherwise the DW_apb_i2c will stop acknowledging.

- **Size:**
 - Write
 - 11 bits when IC_EMPTYFIFO_HOLD_MASTER_EN=1
 - 9 bits when IC_EMPTYFIFO_HOLD_MASTER_EN=0
 - Read
 - 12 bits when IC_FIRST_DATA_BYTE_STATUS = 1
 - 8 bits when IC_FIRST_DATA_BYTE_STATUS = 0
- **Address Offset:** 0x10
- **Read/Write Access:** Read/Write

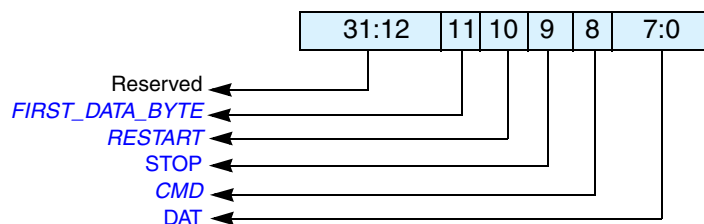


Table 6-8 IC_DATA_CMD Register Fields

Bits	Name	R/W	Description
31:12	Reserved	N/A	Reserved
11	FIRST_DATA_BYTE	R	<p>Indicates the first data byte received after the address phase for receive transfer in Master receiver or Slave receiver mode.</p> <p>Reset value: 0x0</p> <p>Dependencies: This Register bit value is applicable only when FIRST_DATA_BYTE_STATUS=1.</p> <p>Note: In case of APB_DATA_WIDTH=8:</p> <ol style="list-style-type: none"> 1. You must perform two APB Reads to IC_DATA_CMD to get status on 11 bit. 2. To read the 11 bit, you must perform the first data byte read [7:0] (offset 0x10) and then perform the second read[15:8](offset 0x11) to know the status of 11 bit (whether the data received in previous read is a first data byte). 3. The 11th bit is an optional read field. You can ignore 2nd byte read [15:8] (offset 0x11) if not interested in the FIRST_DATA_BYTE status.
10	RESTART	W	<p>This bit controls whether a RESTART is issued before the byte is sent or received. This bit is available only if IC_EMPTYFIFO_HOLD_MASTER_EN is configured to 1.</p> <ul style="list-style-type: none"> ■ 1 – If IC_RESTART_EN is 1, a RESTART is issued before the data is sent/received (according to the value of CMD), regardless of whether or not the transfer direction is changing from the previous command; if IC_RESTART_EN is 0, a STOP followed by a START is issued instead. ■ 0 – If IC_RESTART_EN is 1, a RESTART is issued only if the transfer direction is changing from the previous command; if IC_RESTART_EN is 0, a STOP followed by a START is issued instead.
9	STOP	W	<p>This bit controls whether a STOP is issued after the byte is sent or received. This bit is available only if IC_EMPTYFIFO_HOLD_MASTER_EN is configured to 1.</p> <ul style="list-style-type: none"> ■ 1 – STOP is issued after this byte, regardless of whether or not the Tx FIFO is empty. If the Tx FIFO is not empty, the master immediately tries to start a new transfer by issuing a START and arbitrating for the bus. ■ 0 – STOP is not issued after this byte, regardless of whether or not the Tx FIFO is empty. If the Tx FIFO is not empty, the master continues the current transfer by sending/receiving data bytes according to the value of the CMD bit. If the Tx FIFO is empty, the master holds the SCL line low and stalls the bus until a new command is available in the Tx FIFO.

Table 6-8 IC_DATA_CMD Register Fields (Continued)

Bits	Name	R/W	Description
8	<i>CMD</i>	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> <ul style="list-style-type: none"> ■ 1 = Read ■ 0 = Write <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 the data in IC_DATA_CMD is to be transmitted.</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 <i>TX_ABRT</i> interrupt (bit 6 of the <i>IC_RAW_INTR_STAT</i> register), unless bit 11 (<i>SPECIAL</i>) in the <i>IC_TAR</i> register has been cleared.</p> <p>If a “1” is written to this bit after receiving a <i>RD_REQ</i> interrupt, then a <i>TX_ABRT</i> interrupt occurs.</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 (<i>DAT</i>) 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>

6.3.6 IC_SS_SCL_HCNT

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

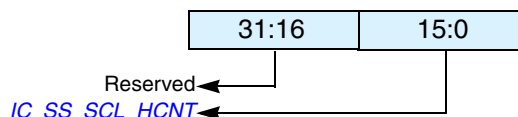


Table 6-9 IC_SS_SCL_HCNT Register Fields

Bits	Name	R/W	Description
31:16	Reserved	N/A	Reserved
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. For more information, refer to “IC_CLK Frequency Configuration” on page 59.</p> <p>This register can be written only when the I²C interface is disabled which corresponds to IC_ENABLE[0] 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 APB_DATA_WIDTH = 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 IC_HC_COUNT_VALUES 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 IC_SS_SCL_HCNT + 10.</p> <p>Reset value: IC_SS_SCL_HIGH_COUNT configuration parameter</p>

¹Read-only if IC_HC_COUNT_VALUES = 1.

6.3.7 IC_SS_SCL_LCNT

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

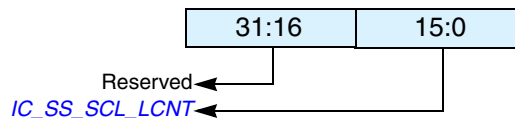


Table 6-10 IC_SS_SCL_LCNT Register Fields

Bits	Name	R/W	Description
31:16	Reserved	N/A	Reserved
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. For more information, refer to “IC_CLK Frequency Configuration” on page 59.</p> <p>This register can be written only when the I²C interface is disabled which corresponds to IC_ENABLE[0] 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 APB_DATA_WIDTH = 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 IC_HC_COUNT_VALUES is set to 1, this register is read only.</p> <p>Reset value: IC_SS_SCL_LOW_COUNT configuration parameter</p>

¹Read-only if IC_HC_COUNT_VALUES = 1.

6.3.8 IC_FS_SCL_HCNT

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

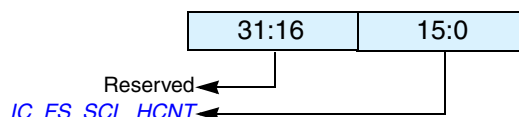


Table 6-11 IC_FS_SCL_HCNT Register Fields

Bits	Name	R/W	Description
31:16	Reserved	N/A	Reserved
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 mode or fast mode plus. It is used in high-speed mode to send the Master Code and START BYTE or General CALL. For more information, refer to “IC_CLK Frequency Configuration” on page 59.</p> <p>This register goes away and becomes read-only returning 0s if IC_MAX_SPEED_MODE = standard. This register can be written only when the I²C interface is disabled, which corresponds to IC_ENABLE[0] 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 APB_DATA_WIDTH == 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 IC_HC_COUNT_VALUES is set to 1, this register is read only.</p> <p>Reset value: IC_FS_SCL_HIGH_COUNT configuration parameter</p>

¹Read-only if IC_HC_COUNT_VALUES = 1.

6.3.9 IC_FS_SCL_LCNT

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

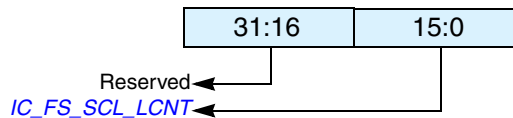


Table 6-12 IC_FS_SCL_LCNT Register Fields

Bits	Name	R/W	Description
31:16	Reserved	N/A	Reserved
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 mode or fast mode plus. It is used in high-speed mode to send the Master Code and START BYTE or General CALL. For more information, refer to “IC_CLK Frequency Configuration” on page 59.</p> <p>This register goes away and becomes read-only returning 0s if IC_MAX_SPEED_MODE = standard.</p> <p>This register can be written only when the I²C interface is disabled, which corresponds to IC_ENABLE[0] 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 APB_DATA_WIDTH = 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 IC_HC_COUNT_VALUES is set to 1, this register is read only.</p> <p>Reset value: IC_FS_SCL_LOW_COUNT configuration parameter</p>

¹Read-only if IC_HC_COUNT_VALUES = 1.

6.3.10 IC_HS_SCL_HCNT

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

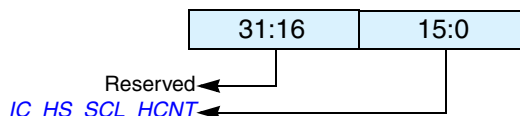


Table 6-13 IC_HS_SCL_HCNT Register Fields

Bits	Name	R/W	Description
31:16	Reserved	N/A	Reserved
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. For more information, refer to “IC_CLK Frequency Configuration” on page 59.</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. This register goes away and becomes read-only returning 0s if IC_MAX_SPEED_MODE != high.</p> <p>This register can be written only when the I²C interface is disabled, which corresponds to IC_ENABLE[0] 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 APB_DATA_WIDTH = 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 IC_HC_COUNT_VALUES is set to 1, this register is read only.</p> <p>Reset value: IC_HS_SCL_HIGH_COUNT configuration parameter</p>
¹ Read-only if IC_HC_COUNT_VALUES = 1.			

6.3.11 IC_HS_SCL_LCNT

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

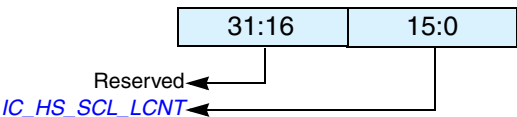


Table 6-14 IC_HS_SCL_LCNT Register Fields

Bits	Name	R/W	Description
31:16	Reserved	N/A	Reserved
15:0	IC_HS_SCL_LCNT	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. For more information, refer to “IC_CLK Frequency Configuration” on page 59.</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 IC_MAX_SPEED_MODE != high.</p> <p>This register can be written only when the I²C interface is disabled, which corresponds to IC_ENABLE[0] 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 APB_DATA_WIDTH == 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 IC_HC_COUNT_VALUES is set to 1, this register is read only.</p> <p>Reset value: IC_HS_SCL_LOW_COUNT configuration parameter</p>

¹Read-only if IC_HC_COUNT_VALUES = 1.

6.3.12 IC_INTR_STAT

- **Name:** I²C Interrupt Status Register
- **Size:** 14 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.

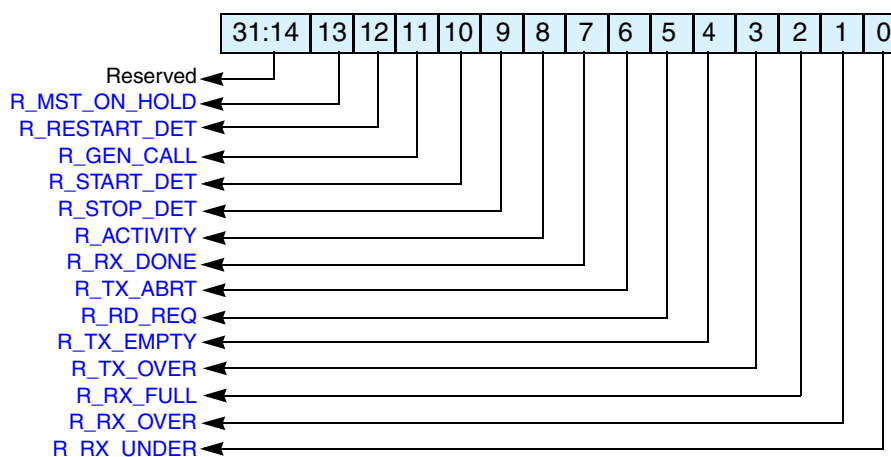


Table 6-15 IC_INTR_STAT Register Fields

Bits	Name	R/W	Description
31:14	Reserved	N/A	Reserved
13	<i>R_MST_ON_HOLD</i>	R	See “ IC_RAW_INTR_STAT ” on page 133 for a detailed description of these bits. Reset value: 0x0
12	<i>R_RESTART_DET</i>		
11	<i>R_GEN_CALL</i>		
10	<i>R_START_DET</i>		
9	<i>R_STOP_DET</i>		
8	<i>R_ACTIVITY</i>		
7	<i>R_RX_DONE</i>		
6	<i>R_TX_ABRT</i>		
5	<i>R_RD_REQ</i>		
4	<i>R_TX_EMPTY</i>		
3	<i>R_TX_OVER</i>		
2	<i>R_RX_FULL</i>		
1	<i>R_RX_OVER</i>		
0	<i>R_RX_UNDER</i>		

6.3.13 IC_INTR_MASK

- **Name:** I²C Interrupt Mask Register

- **Size:** 14 bits

- **Address Offset:** 0x30

- **Read/Write Access:** Read/Write

However, if configuration parameter IC_SLV_RESTART_DET = 0, bit 13 is read only; if configuration parameter I2C_DYNAMIC_TAR_UPDATE = 0 or IC_EMPTYFIFO_HOLD_MASTER_EN = 0, bit 14 is read only

These bits mask their corresponding interrupt status bits. This register is active low; a value of 0 masks the interrupt, whereas a value of 1 unmask the interrupt.

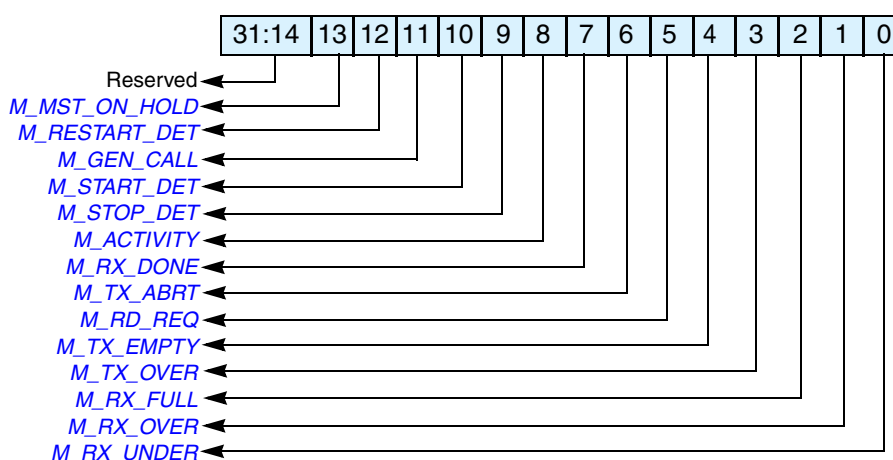


Table 6-16 IC_INTR_MASK Register Fields

Bits	Name	R/W	Description
31:14	Reserved	N/A	Reserved
13	<i>M_MST_ON_HOLD</i>	R or R/W	This bit masks the R_MST_ON_HOLD interrupt bit in the IC_INTR_STAT register. Dependencies: If I2C_DYNAMIC_TAR_UPDATE = 1 and IC_EMPTYFIFO_HOLD_MASTER_EN = 1, then M_MST_ON_HOLD is read/write. Otherwise M_MST_ON_HOLD is read-only. Reset value: 14'h8ff
12	<i>M_RESTART_DET</i>	R or R/W	This bit masks the R_RESTART_DET interrupt status bit in the IC_INTR_STAT register. Dependencies: If IC_SLV_RESTART_DET_EN = 1, then M_RESTART_DET is read/write. Otherwise M_RESTART_DET is read-only. Reset value: 14'h8ff

Table 6-16 IC_INTR_MASK Register Fields (Continued)

Bits	Name	R/W	Description
11	<i>M_GEN_CALL</i>	R/W	These bits mask their corresponding interrupt status bits in the IC_INTR_STAT register. Reset value: 14'h8ff
10	<i>M_START_DET</i>		
9	<i>M_STOP_DET</i>		
8	<i>M_ACTIVITY</i>		
7	<i>M_RX_DONE</i>		
6	<i>M_TX_ABRT</i>		
5	<i>M_RD_REQ</i>		
4	<i>M_TX_EMPTY</i>		
3	<i>M_TX_OVER</i>		
2	<i>M_RX_FULL</i>		
1	<i>M_RX_OVER</i>		
0	<i>M_RX_UNDER</i>		

6.3.14 IC_RAW_INTR_STAT

- **Name:** I²C Raw Interrupt Status Register
- **Size:** 14 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.

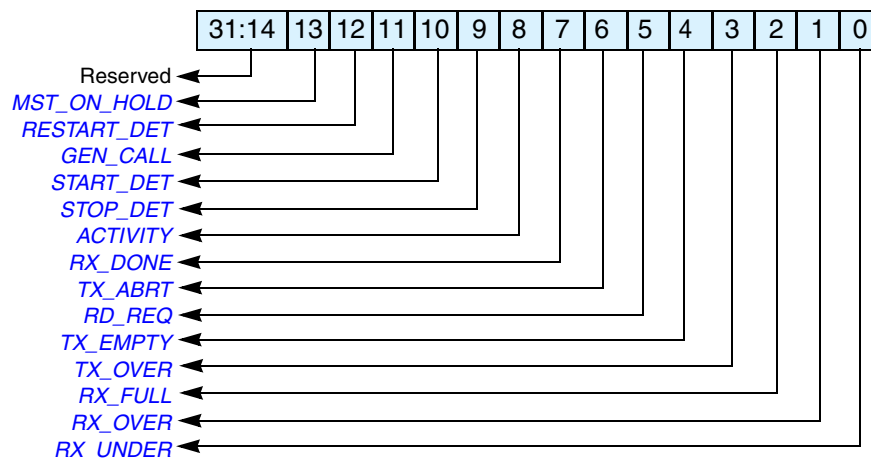


Table 6-17 IC_RAW_INTR_STAT Register Fields

Bits	Name	R/W	Description
31:14	Reserved	N/A	Reserved
13	<i>MST_ON_HOLD</i>	R	Indicates whether a master is holding the bus and the Tx FIFO is empty. Enabled only when <code>I2C_DYNAMIC_TAR_UPDATE = 1</code> and <code>IC_EMPTYFIFO_HOLD_MASTER_EN = 1</code> Reset value: 0x0
12	<i>RESTART_DET</i>	R	Indicates whether a RESTART condition has occurred on the I2C interface when DW_apb_i2c is operating in slave mode and the slave is the addressed slave. Enabled only when <code>IC_SLV_RESTART_DET_EN = 1</code> NOTE: However, in high-speed mode or during a START BYTE transfer, the RESTART comes before the address field as per the I2C protocol. In this case, the slave is not the addressed slave when the RESTART is issued, therefore DW_apb_i2c does not generate the RESTART_DET interrupt. Reset value: 0x0

Table 6-17 IC_RAW_INTR_STAT Register Fields (Continued)

Bits	Name	R/W	Description
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. In Slave Mode: <ul style="list-style-type: none"> If IC_CON[7]=1'b1 (STOP_DET_IFADDRESSED), the STOP_DET interrupt is generated only if the slave is addressed. Note: During a general call address, this slave does not issue a STOP_DET interrupt if STOP_DET_IF_ADDRESSED=1'b1, even if the slave responds to the general call address by generating ACK. The STOP_DET interrupt is generated only when the transmitted address matches the slave address (SAR). <ul style="list-style-type: none"> If IC_CON[7]=1'b0 (STOP_DET_IFADDRESSED), the STOP_DET interrupt is issued irrespective of whether it is being addressed. In Master Mode: <ul style="list-style-type: none"> If IC_CON[10]=1'b1 (STOP_DET_IF_MASTER_ACTIVE), the STOP_DET interrupt is issued only if the master is active. If IC_CON[10]=1'b0 (STOP_DET_IFADDRESSED), the STOP_DET interrupt is issued irrespective of whether the master is active. Reset value: 0x0
8	<i>ACTIVITY</i>	R	This bit captures DW_apb_i2c activity and stays set until it is cleared. There are four ways to clear it: <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 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. Reset value: 0x0
7	<i>RX_DONE</i>	R	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. Reset value: 0x0

Table 6-17 IC_RAW_INTR_STAT Register Fields (Continued)

Bits	Name	R/W	Description
6	<i>TX_ABORT</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 IC_TX_ABORT_SOURCE register indicates the reason why the transmit abort takes places.</p> <p>NOTE: The DW_apb_i2c flushes/resets/empties only the TX_FIFO whenever there is a transmit abort caused by any of the events tracked by the IC_TX_ABORT_SOURCE register. The Tx FIFO remains in this flushed state until the register IC_CLR_TX_ABORT is read. Once this read is performed, the Tx FIFO is then ready to accept more data bytes from the APB interface. RX FIFO is flushed because of TX_ABORT is controlled by the coreConsultant parameter IC_AVOID_RX_FIFO_FLUSH_ON_TX_ABORT.</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 IC_DATA_CMD register. This bit is set to 0 just after the processor reads the IC_CLR_RD_REQ register.</p> <p>Reset value: 0x0</p>
4	<i>TX_EMPTY</i>	R	<p>The behavior of the TX_EMPTY interrupt status differs based on the TX_EMPTY_CTRL selection in the IC_CON register.</p> <ul style="list-style-type: none"> When TX_EMPTY_CTRL = 0: <p>This bit is set to 1 when the transmit buffer is at or below the threshold value set in the IC_TX_TL register.</p> When TX_EMPTY_CTRL = 1: <p>This bit is set to 1 when the transmit buffer is at or below the threshold value set in the IC_TX_TL register and the transmission of the address/data from the internal shift register for the most recently popped command is completed.</p> <p>It is automatically cleared by hardware when the buffer level goes above the threshold. When IC_ENABLE[0] is set to 0, the TX FIFO is flushed and held in reset. There 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 any activity, then with ic_en=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 IC_TX_BUFFER_DEPTH and the processor attempts to issue another I²C command by writing to the IC_DATA_CMD register. When the module is disabled, this bit keeps its level until the master or slave state machines go into idle, and when ic_en goes to 0, this interrupt is cleared.</p> <p>Reset value: 0x0</p>

Table 6-17 IC_RAW_INTR_STAT Register Fields (Continued)

Bits	Name	R/W	Description
2	<i>RX_FULL</i>	R	<p>Set when the receive buffer reaches or goes above the RX_TL 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 (IC_ENABLE[0]=0), the RX FIFO is flushed and held in reset; therefore the RX FIFO is not full. So this bit is cleared once IC_ENABLE[0] is set to 0, regardless of the activity that continues.</p> <p>Reset value: 0x0</p>
1	<i>RX_OVER</i>	R	<p>Set if the receive buffer is completely filled to IC_RX_BUFFER_DEPTH 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 (IC_ENABLE[0]=0), this bit keeps its level until the master or slave state machines go into idle, and when ic_en goes to 0, this interrupt is cleared.</p> <p>NOTE: If the configuration parameter IC_RX_FULL_HLD_BUS_EN is enabled and bit 9 of the IC_CON register (RX_FIFO_FULL_HLD_CTRL) is programmed to HIGH, then the RX_OVER interrupt never occurs, because the Rx FIFO never overflows.</p> <p>Reset value: 0x0</p>
0	<i>RX_UNDER</i>	R	<p>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 (IC_ENABLE[0]=0), this bit keeps its level until the master or slave state machines go into idle, and when ic_en goes to 0, this interrupt is cleared.</p> <p>Reset value: 0x0</p>

6.3.15 IC_RX_TL

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

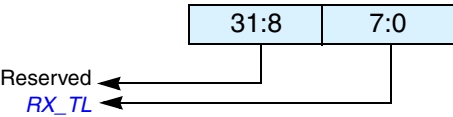


Table 6-18 IC_RX_TL Register Fields

Bits	Name	R/W	Description
31:8	Reserved	N/A	Reserved
7:0	<i>RX_TL</i>	R/W	<p>Receive FIFO Threshold Level</p> <p>Controls the level of entries (or above) that triggers the <i>RX_FULL</i> interrupt (bit 2 in IC_RAW_INTR_STAT 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.</p> <p>A value of 0 sets the threshold for 1 entry, and a value of 255 sets the threshold for 256 entries.</p> <p>Reset value: IC_RX_TL configuration parameter</p>

6.3.16 IC_TX_TL

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

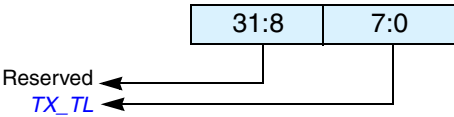


Table 6-19 IC_TX_TL Register Fields

Bits	Name	R/W	Description
31:8	Reserved	N/A	Reserved
7:0	TX_TL	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>

6.3.17 IC_CLR_INTR

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

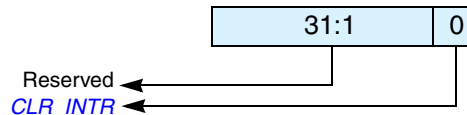


Table 6-20 IC_CLR_INTR Register Fields

Bits	Name	R/W	Description
31:1	Reserved	N/A	Reserved
0	<i>CLR_INTR</i>	R	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> . Reset value: 0x0

6.3.18 IC_CLR_RX_UNDER

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

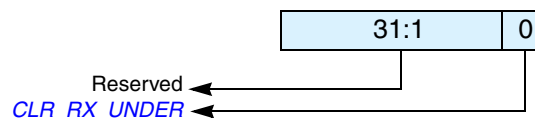


Table 6-21 IC_CLR_RX_UNDER Register Fields

Bits	Name	R/W	Description
31:1	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

6.3.19 IC_CLR_RX_OVER

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

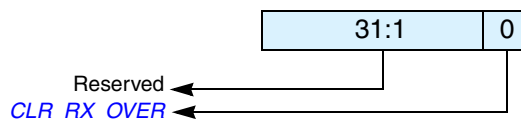


Table 6-22 IC_CLR_RX_OVER Register Fields

Bits	Name	R/W	Description
31:1	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

6.3.20 IC_CLR_TX_OVER

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

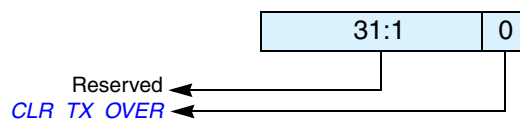


Table 6-23 IC_CLR_TX_OVER Register Fields

Bits	Name	R/W	Description
31:1	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

6.3.21 IC_CLR_RD_REQ

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

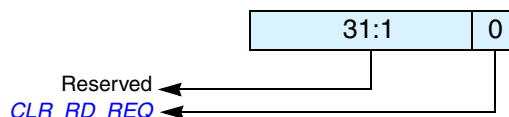


Table 6-24 IC_CLR_RD_REQ Register Fields

Bits	Name	R/W	Description
31:1	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

6.3.22 IC_CLR_TX_ABORT

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

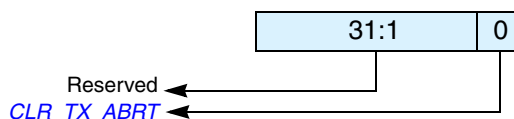


Table 6-25 IC_CLR_TX_ABORT Register Fields

Bits	Name	R/W	Description
31:1	Reserved	N/A	Reserved
0	<i>CLR_TX_ABORT</i>	R	Read this register to clear the <i>TX_ABORT</i> interrupt (bit 6) of the <i>IC_RAW_INTR_STAT</i> register, and the <i>IC_TX_ABORT_SOURCE</i> 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 <i>IC_TX_ABORT_SOURCE</i> register for an exception to clearing <i>IC_TX_ABORT_SOURCE</i> . Reset value: 0x0

6.3.23 IC_CLR_RX_DONE

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

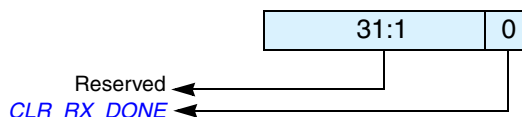


Table 6-26 IC_CLR_RX_DONE Register Fields

Bits	Name	R/W	Description
31:1	Reserved	N/A	Reserved
0	CLR_RX_DONE	R	Read this register to clear the <i>RX_DONE</i> interrupt (bit 7) of the <i>IC_RAW_INTR_STAT</i> register. Reset value: 0x0

6.3.24 IC_CLR_ACTIVITY

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

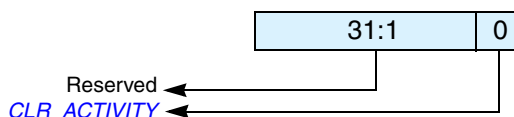


Table 6-27 IC_CLR_ACTIVITY Register Fields

Bits	Name	R.W	Description
31:1	Reserved	N/A	Reserved
0	CLR_ACTIVITY	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

6.3.25 IC_CLR_STOP_DET

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

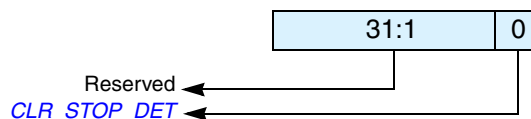


Table 6-28 IC_CLR_STOP_DET Register Fields

Bits	Name	R/W	Description
31:1	Reserved	N/A	Reserved
0	CLR_STOP_DET	R	Read this register to clear the STOP_DET interrupt (bit 9) of the IC_RAW_INTR_STAT register. Reset value: 0x0

6.3.26 IC_CLR_START_DET

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

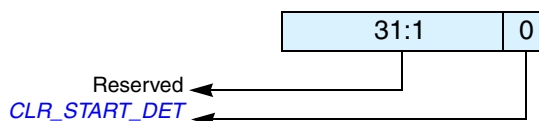


Table 6-29 IC_CLR_START_DET Register Fields

Bits	Name	R/W	Description
31:1	Reserved	N/A	Reserved
0	CLR_START_DET	R	Read this register to clear the START_DET interrupt (bit 10) of the IC_RAW_INTR_STAT register. Reset value: 0x0

6.3.27 IC_CLR_GEN_CALL

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

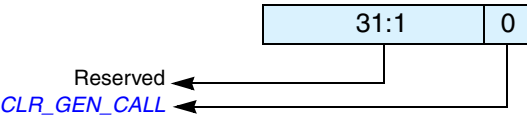


Table 6-30 IC_CLR_GEN_CALL Register Fields

Bits	Name	R/W	Description
31:1	Reserved	N/A	Reserved
0	CLR_GEN_CALL	R	Read this register to clear the GEN_CALL interrupt (bit 11) of IC_RAW_INTR_STAT register. Reset value: 0x0

6.3.28 IC_ENABLE

- Name: I²C Enable Register
- Size: 2 bits
- Address Offset: 0x6c
- Read/Write Access: Read/Write

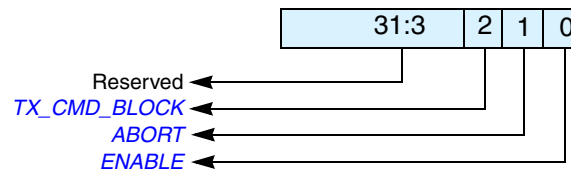


Table 6-31 IC_ENABLE Register Fields

Bits	Name	R/W	Description
31:3	Reserved	N/A	Reserved
2	TX_CMD_BLOCK	R/W	<p>In Master mode</p> <ul style="list-style-type: none"> ■ 1'b1: Blocks the transmission of data on I²C bus even if Tx FIFO has data to transmit. ■ 1'b0: The transmission of data starts on I²C bus automatically, as soon as the first data is available in the Tx FIFO. <p>Reset value: IC_TX_CMD_BLOCK_DEFAULT</p> <p>Dependencies: This Register bit value is applicable only when IC_TX_CMD_BLOCK = 1.</p> <p>Note: To block the execution of Master commands, set the TX_CMD_BLOCK bit only when Tx FIFO is empty (IC_STATUS[2]=1) and the master is in the Idle state (IC_STATUS[5] == 0). Any further commands put in the Tx FIFO are not executed until TX_CMD_BLOCK bit is unset.</p>
1	ABORT	R/W	<p>When set, the controller initiates the transfer abort.</p> <ul style="list-style-type: none"> ■ 0: ABORT not initiated or ABORT done ■ 1: ABORT operation in progress <p>The software can abort the I2C transfer in master mode by setting this bit. The software can set this bit only when ENABLE is already set; otherwise, the controller ignores any write to ABORT bit. The software cannot clear the ABORT bit once set. In response to an ABORT, the controller issues a STOP and flushes the Tx FIFO after completing the current transfer, then sets the TX_ABORT interrupt after the abort operation. The ABORT bit is cleared automatically after the abort operation.</p> <p>For a detailed description on how to abort I2C transfers, refer to “Aborting I2C Transfers” on page 57.</p> <p>Reset value: 0x0</p>

Table 6-31 IC_ENABLE Register Fields (Continued)

Bits	Name	R/W	Description
0	<i>ENABLE</i>	R/W	<p>Controls whether the DW_apb_i2c is enabled.</p> <ul style="list-style-type: none"> 0: Disables DW_apb_i2c (TX and RX FIFOs are held in an erased state) 1: Enables DW_apb_i2c <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 56. 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 56.</p> <p>Reset value: 0x0</p>

6.3.29 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 to 10 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

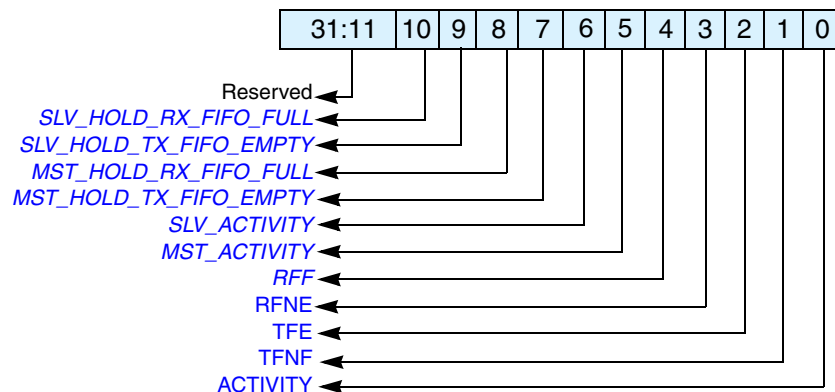


Table 6-32 IC_STATUS Register Fields

Bits	Name	R/W	Description
31:11	Reserved	N/A	Reserved
10	SLV_HOLD_RX_FIFO_FULL	R	<p>This bit indicates the BUS Hold in Slave mode due to the Rx FIFO being Full and an additional byte being received (this kind of Bus hold is applicable if IC_RX_FULL_HLD_BUS_EN is set to 1).</p> <p>Reset value: 0x0</p> <p>Dependencies: This Register bit value is applicable only when IC_STAT_FOR_CLK_STRETCH=1.</p>

Table 6-32 IC_STATUS Register Fields (Continued)

Bits	Name	R/W	Description
9	SLV_HOLD_TX_FIFO_EMPTY	R	<p>This bit indicates the BUS Hold in Slave mode for the Read request when the Tx FIFO is empty. The Bus is in hold until the Tx FIFO has data to Transmit for the read request.</p> <p>Reset value: 0x0</p> <p>Dependencies: This Register bit value is applicable only when IC_STAT_FOR_CLK_STRETCH=1.</p>
8	MST_HOLD_RX_FIFO_FULL	R	<p>This bit indicates the BUS Hold in Master mode due to Rx FIFO is Full and additional byte has been received (This kind of Bus hold is applicable if IC_RX_FULL_HLD_BUS_EN is set to 1).</p> <p>Reset value: 0x0</p> <p>Dependencies: This Register bit value is applicable only when IC_STAT_FOR_CLK_STRETCH=1</p>
7	MST_HOLD_TX_FIFO_EMPTY		<p>If the IC_EMPTYFIFO_HOLD_MASTER_EN parameter is set to 1, the DW_apb_i2c master stalls the write transfer when Tx FIFO is empty, and the the last byte does not have the Stop bit set.</p> <p>This bit indicates the BUS hold when the master holds the bus because of the Tx FIFO being empty, and the the previous transferred command does not have the Stop bit set. (This kind of Bus hold is applicable if IC_EMPTYFIFO_HOLD_MASTER_EN is set to 1).</p> <p>Reset value: 0x0</p> <p>Dependencies: This Register bit value is applicable only when IC_STAT_FOR_CLK_STRETCH=1</p>
6	SLV_ACTIVITY	R	<p>Slave FSM Activity Status. When the Slave Finite State Machine (FSM) is not in the IDLE state, this bit is set.</p> <ul style="list-style-type: none"> 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 <p>Reset value: 0x0</p>
5	MST_ACTIVITY	R	<p>Master FSM Activity Status. When the Master Finite State Machine (FSM) is not in the IDLE state, this bit is set.</p> <ul style="list-style-type: none"> 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 <p>NOTE: IC_STATUS[0]—that is, <i>ACTIVITY</i> bit—is the OR of <i>SLV_ACTIVITY</i> and <i>MST_ACTIVITY</i> bits.</p> <p>Reset value: 0x0</p>

Table 6-32 IC_STATUS Register Fields (Continued)

Bits	Name	R/W	Description
4	<i>RFF</i>	R	<p>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.</p> <ul style="list-style-type: none"> 0: Receive FIFO is not full 1: Receive FIFO is full <p>Reset value: 0x0</p>
3	<i>RFNE</i>	R	<p>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.</p> <ul style="list-style-type: none"> 0: Receive FIFO is empty 1: Receive FIFO is not empty <p>Reset value: 0x0</p>
2	<i>TFE</i>	R	<p>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.</p> <ul style="list-style-type: none"> 0: Transmit FIFO is not empty 1: Transmit FIFO is empty <p>Reset value: 0x1</p>
1	<i>TFNF</i>	R	<p>Transmit FIFO Not Full. Set when the transmit FIFO contains one or more empty locations, and is cleared when the FIFO is full.</p> <ul style="list-style-type: none"> 0: Transmit FIFO is full 1: Transmit FIFO is not full <p>Reset value: 0x1</p>
0	<i>ACTIVITY</i>	R	<p>I²C Activity Status.</p> <p>Reset value: 0x0</p>

6.3.30 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.

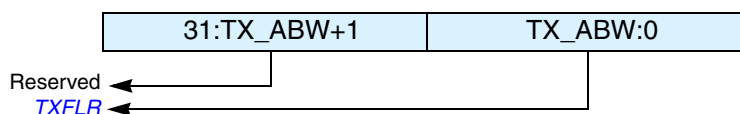


Table 6-33 IC_TXFLR Register Fields

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

6.3.31 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.

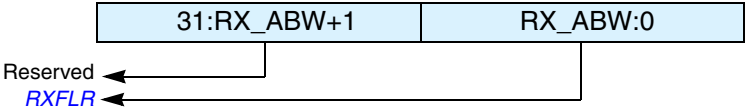


Table 6-34 IC_RXFLR Register Fields

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

6.3.32 IC_SDA_HOLD

- **Name:** I²C SDA Hold Time Length Register
- **Size:** 24 bits
- **Address Offset:** 0x7C
- **Read/Write Access:** Read/Write

The bits [15:0] of this register are used to control the hold time of SDA during transmit in both slave and master mode (after SCL goes from HIGH to LOW).

The bits [23:16] of this register are used to extend the SDA transition (if any) whenever SCL is HIGH in the receiver in either master or slave mode.

Writes to this register succeed only when IC_ENABLE[0]=0.

The values in this register are in units of ic_clk period. The value programmed in IC_SDA_TX_HOLD must be greater than the minimum hold time in each mode – one cycle in master mode, seven cycles in slave mode – for the value to be implemented.

The programmed SDA hold time during transmit (IC_SDA_TX_HOLD) cannot exceed at any time the duration of the low part of scl. Therefore the programmed value cannot be larger than N_SCL_LOW-2, where N_SCL_LOW is the duration of the low part of the scl period measured in ic_clk cycles.

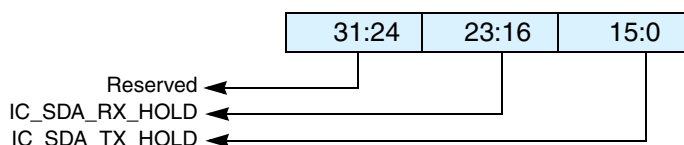


Table 6-35 IC_SDA_HOLD Register Fields

Bits	Name	R/W	Description
31:24	Reserved	N/A	Reserved
23:16	<i>IC_SDA_RX_HOLD</i>	R/W	Sets the required SDA hold time in units of ic_clk period, when DW_apb_i2c acts as a receiver. Reset value: IC_DEFAULT_SDA_HOLD
15:0	<i>IC_SDA_TX_HOLD</i>	R/W	Sets the required SDA hold time in units of ic_clk period, when DW_apb_i2c acts as a transmitter. Reset value: IC_DEFAULT_SDA_HOLD

6.3.33 IC_TX_ABRT_SOURCE

- **Name:** I²C Transmit Abort Source Register
- **Size:** 32 bits
- **Address Offset:** 0x80
- **Read/Write Access:** Read

This register has 32 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.

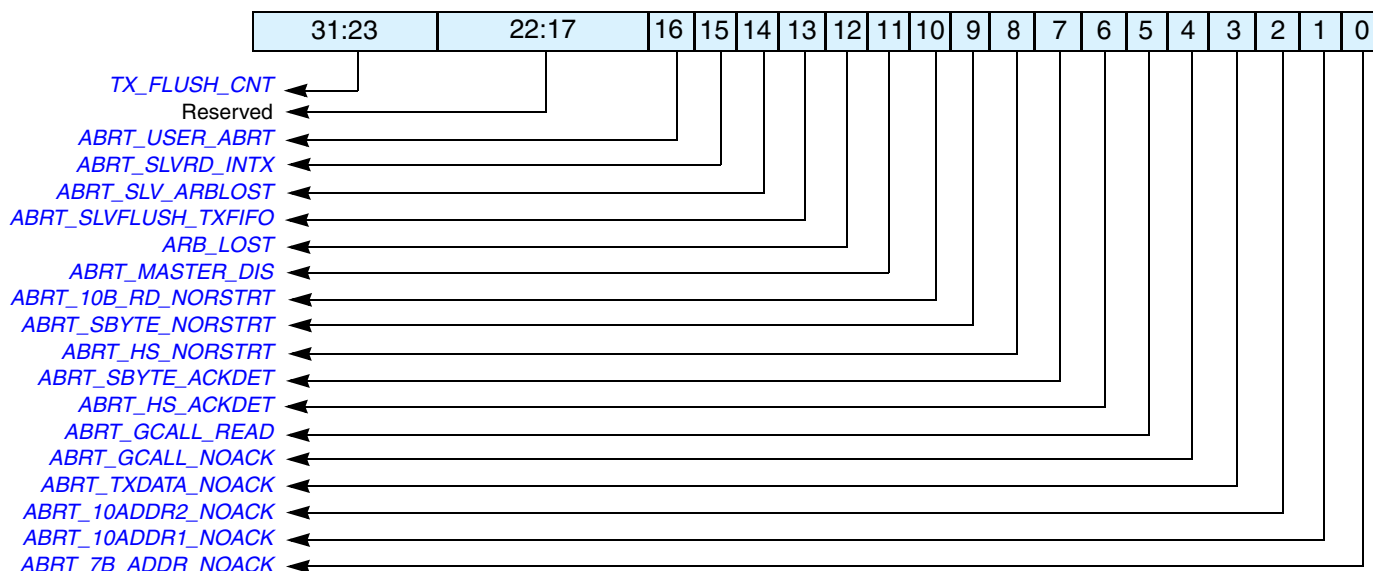


Table 6-36 IC_TX_ABRT_SOURCE Register Fields

Bits	Name	R/W	Description	Role of DW_apb_i2c
31:23	<i>TX_FLUSH_CNT</i>	R	This field indicates the number of Tx FIFO data commands that are flushed due to <i>TX_ABRT</i> interrupt. It is cleared whenever I ² C is disabled. Reset value: 0x0	Master-Transmitter or Slave-Transmitter
22:17	Reserved			
16	<i>ABRT_USER_ABORT</i>	R	This is a master-mode-only bit. Master has detected the transfer abort (<i>IC_ENABLE</i> [1]). Reset value: 0x0	Master-Transmitter

Table 6-36 IC_TX_ABRT_SOURCE Register Fields (Continued)

Bits	Name	R/W	Description	Role of DW_apb_i2c
15	<i>ABRT_SLVRD_INTX</i>	R	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
14	<i>ABRT_SLV_ARBLOST</i>	R	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 1: Master has lost arbitration, or if <i>IC_TX_ABRT_SOURCE</i>[14] is also set, then the slave transmitter has lost arbitration. Reset value: 0x0	Master-Transmitter or Slave-Transmitter
11	<i>ABRT_MASTER_DIS</i>	R	<ul style="list-style-type: none"> 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_NORSTR</i>	R	<ul style="list-style-type: none"> 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

Table 6-36 IC_TX_ABRT_SOURCE Register Fields (Continued)

Bits	Name	R/W	Description	Role of DW_apb_i2c
9	<i>ABRT_SBYTE_NORSTRT</i>	R	<p>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.</p> <p>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 <i>START</i> Byte.</p> <p>Reset value: 0x0</p>	Master
8	<i>ABRT_HS_NORSTRT</i>	R	<p>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.</p> <p>Reset value: 0x0</p>	Master-Transmitter or Master-Receiver
7	<i>ABRT_SBYTE_ACKDET</i>	R	<p>1: Master has sent a <i>START</i> Byte and the <i>START</i> Byte was acknowledged (wrong behavior).</p> <p>Reset value: 0x0</p>	Master
6	<i>ABRT_HS_ACKDET</i>	R	<p>1: Master is in High Speed mode and the High Speed Master code was acknowledged (wrong behavior).</p> <p>Reset value: 0x0</p>	Master
5	<i>ABRT_GCALL_READ</i>	R	<p>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).</p> <p>Reset value: 0x0</p>	Master-Transmitter
4	<i>ABRT_GCALL_NOACK</i>	R	<p>1: DW_apb_i2c in master mode sent a General Call and no slave on the bus acknowledged the General Call.</p> <p>Reset value: 0x0</p>	Master-Transmitter

Table 6-36 IC_TX_ABRT_SOURCE Register Fields (Continued)

Bits	Name	R/W	Description	Role of DW_apb_i2c
3	<i>ABRT_TXDATA_NOACK</i>	R	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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

6.3.34 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.

A write can occur on this register if both of the following conditions are met:

- DW_apb_i2c is disabled ([IC_ENABLE](#)[0] = 0)
- Slave part is inactive ([IC_STATUS](#)[6] = 0)



Note

The IC_STATUS[6] is a register read-back location for the internal slv_activity signal; the user should poll this before writing the ic_slv_data_nack_only bit.

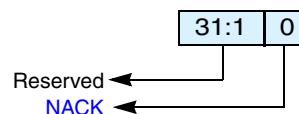


Table 6-37 IC_SLV_DATA_NACK_ONLY Register Fields

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> <ul style="list-style-type: none"> ■ 1 = generate NACK after data byte received ■ 0 = generate NACK/ACK normally <p>Reset value: 0x0</p>

6.3.35 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.

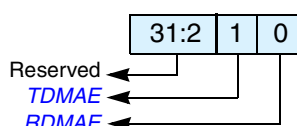


Table 6-38 IC_DMA_CR Register Fields

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. <ul style="list-style-type: none"> ■ 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. <ul style="list-style-type: none"> ■ 0 = Receive DMA disabled ■ 1 = Receive DMA enabled Reset value: 0x0

6.3.36 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.

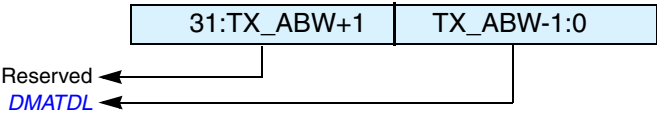


Table 6-39 IC_DMA_TDLR Register Fields

Bits	Name	R/W	Description
31:TX_ABW	Reserved	N/A	Reserved
TX_ABW-1:0	DMATDL	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 TDMAE = 1. Reset value: 0x0

6.3.37 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.

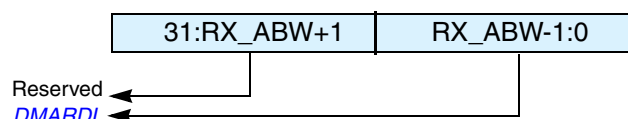


Table 6-40 IC_DMA_RDLR Register Fields

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

6.3.38 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 – by holding SCL low when DW_apb_i2c services a read request while operating as a slave-transmitter. The relevant I²C requirement is t_{SU:DAT} (note 4) as detailed in the *I2C Bus Specification*. This register must be programmed with a value equal to or greater than 2.

Writes to this register succeed only when IC_ENABLE[0] = 0.



The length of setup time is calculated using [(IC_SDA_SETUP - 1) * (ic_clk_period)], so if the user requires 10 ic_clk periods of setup time, they should program a value of 11. The IC_SDA_SETUP register is only used by the DW_apb_i2c when operating as a slave transmitter.

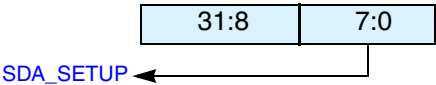


Table 6-41 IC_SDA_SETUP Register Fields

Bits	Name	R/W	Description
31:8	Reserved	N/A	Reserved
7:0	SDA_SETUP	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. IC_SDA_SETUP must be programmed with a minimum value of 2. Default Reset value: 0x64, but can be hardcoded by setting the IC_DEFAULT_SDA_SETUP configuration parameter.

6.3.39 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 an ACK or NACK when it receives an I²C General Call address. This register is applicable only when the DW_apb_i2c is in the slave mode.

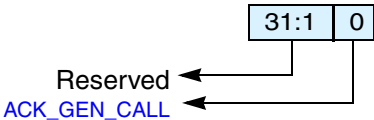


Table 6-42 IC_ACK_GENERAL_CALL Register Fields

Bits	Name	R/W	Description
31:1	Reserved	N/A	Reserved
0	ACK_GEN_CALL	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. When set to 0, the DW_apb_i2c does not generate General Call interrupts. Default Reset value: 0x1, but can be hardcoded by setting the IC_DEFAULT_ACK_GENERAL_CALL configuration parameter.

6.3.40 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 **IC_ENABLE**[0] is set from 1 to 0; that is, when DW_apb_i2c is disabled.

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

If **IC_ENABLE**[0] 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**[0] has been set to 0, a delay occurs for bit 0 to be read as 0 because disabling the DW_apb_i2c depends on I²C bus activities.

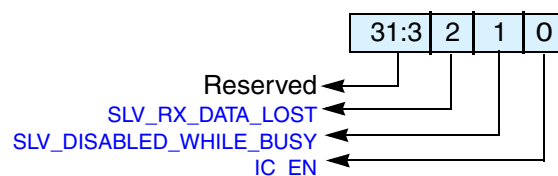


Table 6-43 IC_ENABLE_STATUS Register Fields

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 setting IC_ENABLE[0] from 1 to 0. 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.</p> <p>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 IC_ENABLE[0] 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.</p> <p>NOTE: The CPU can safely read this bit when IC_EN (bit 0) is read as 0.</p> <p>Reset value: 0x0</p>

Table 6-43 IC_ENABLE_STATUS Register Fields (Continued)

Bits	Name	R/W	Description
1	SLV_DISABLED_WHILE_BUSY	R	<p>Slave Disabled While Busy (Transmit, Receive). This bit indicates if a potential or active Slave operation has been aborted due to setting bit 0 of the IC_ENABLE register from 1 to 0. This bit is set when the CPU writes a 0 to bit 0 of IC_ENABLE while: (a) DW_apb_i2c 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, DW_apb_i2c 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 DW_apb_i2c (IC_SAR register) OR if the transfer is completed before bit 0 of IC_ENABLE 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 DW_apb_i2c has a chance to NACK a transfer, and bit 0 of IC_ENABLE has been set to 0, then this bit will also be set to 1.</p> <p>When read as 0, DW_apb_i2c 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	IC_EN	R	<p>ic_en Status. This bit always reflects the value driven on the output port ic_en.</p> <ul style="list-style-type: none"> When read as 1, DW_apb_i2c is deemed to be in an enabled state. When read as 0, DW_apb_i2c is deemed completely inactive. <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>

6.3.41 IC_FS_SPKLEN

- **Name:** I²C SS and FS Spike Suppression Limit Register
- **Size:** 8 bits
- **Address Offset:** 0xA0
- **Read/Write Access:** Read/Write

This register is used to store the duration, measured in *ic_clk* cycles, of the longest spike that is filtered out by the spike suppression logic when the component is operating in standard mode, fast mode, or fast mode plus. The relevant I²C requirement is tSP (Table 4) as detailed in the *I²C Bus Specification*. This register must be programmed with a minimum value of 1.

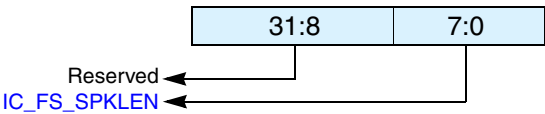


Table 6-44 IC_FS_SPKLEN Register Fields

Bits	Name	R/W	Description
31:8	Reserved		
7:0	<i>IC_FS_SPKLEN</i>	R/W	<p>This register must be set before any I²C bus transaction can take place to ensure stable operation. This register sets the duration, measured in <i>ic_clk</i> cycles, of the longest spike in the SCL or SDA lines that are filtered out by the spike suppression logic; for more information, refer to “Spike Suppression” on page 57.</p> <p>This register can be written only when the I²C interface is disabled, which corresponds to IC_ENABLE[0] being set to 0. Writes at other times have no effect.</p> <p>The minimum valid value is 1; hardware prevents values less than this being written, and if attempted, results in 1 being set.</p> <p>Reset value: <i>IC_DEFAULT_FS_SPKLEN</i> configuration parameter</p>

6.3.42 IC_HS_SPKLEN

- **Name:** I²C HS Spike Suppression Limit Register
- **Size:** 8 bits
- **Address Offset:** 0xA4
- **Read/Write Access:** Read/Write

This register is used to store the duration, measured in *ic_clk* cycles, of the longest spike that is filtered out by the spike suppression logic when the component is operating in HS mode. The relevant I²C requirement is tSP (Table 6) as detailed in the *I²C Bus Specification*. This register must be programmed with a minimum value of 1 and is implemented only if the component is configured to support HS mode; that is, if the IC_MAX_SPEED_MODE parameter is set to 3.

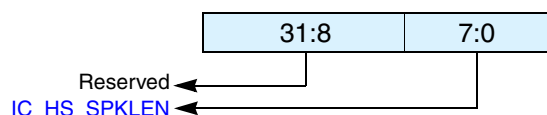


Table 6-45 IC_HS_SPKLEN Register Fields

Bits	Name	R/W	Description
31:8	Reserved		
7:0	<i>IC_HS_SPKLEN</i>	R/W	<p>This register must be set before any I²C bus transaction can take place to ensure stable operation. This register sets the duration, measured in <i>ic_clk</i> cycles, of the longest spike in the SCL or SDA lines that are filtered out by the spike suppression logic; for more information, refer to “Spike Suppression” on page 57.</p> <p>This register can be written only when the I²C interface is disabled, which corresponds to IC_ENABLE[0] being set to 0. Writes at other times have no effect.</p> <p>The minimum valid value is 1; hardware prevents values less than this being written, and if attempted, results in 1 being set.</p> <p>This register is implemented only if the component is configured to support HS mode; that is, if the IC_MAX_SPEED_MODE parameter is set to 3.</p> <p>Reset value: <i>IC_DEFAULT_HS_SPKLEN</i> configuration parameter</p>

6.3.43 IC_CLR_RESTART_DET

- **Name:** Clear RESTART_DET Interrupt Register
- **Size:** 1 bit
- **Address Offset:** 0xA8
- **Read/Write Access:** Read

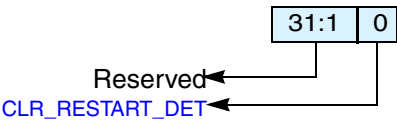


Table 6-46 IC_CLR_RESTART_DET Register Fields

Bits	Name	R/W	Description
31:1	Reserved	N/A	Reserved
0	CLR_RESTART_DET	R	Read this register to clear the RESTART_DET interrupt (bit 12) of the IC_RAW_INTR_STAT register. Dependencies: This register is present only when IC_SLV_RESTART_DET_EN = 1. Reset value: 0x0

6.3.44 IC_COMP_PARAM_1

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

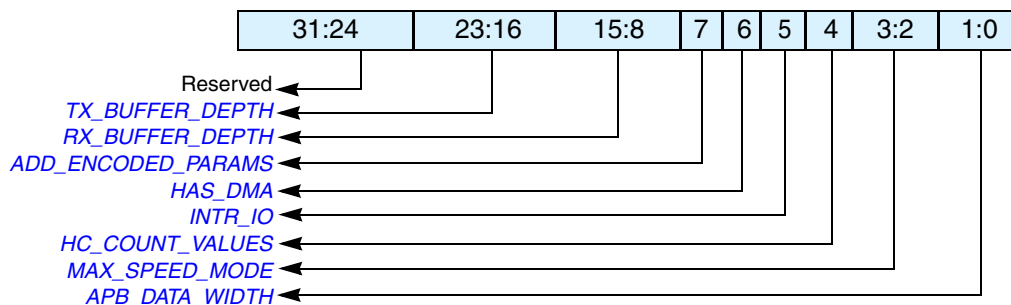


Table 6-47 IC_COMP_PARAM_1 Register Fields


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	TX_BUFFER_DEPTH	R	The value of this register is derived from the IC_TX_BUFFER_DEPTH coreConsultant parameter. <ul style="list-style-type: none"> ■ 0x00 = Reserved ■ 0x01 = 2 ■ 0x02 = 3 ... ■ 0xFF = 256
15:8	RX_BUFFER_DEPTH	R	The value of this register is derived from the IC_RX_BUFFER_DEPTH coreConsultant parameter. For a description of this parameter, see Table 4-1 on page 77. <ul style="list-style-type: none"> ■ 0x00 = Reserved ■ 0x01 = 2 ■ 0x02 = 3 ... ■ 0xFF = 256

Table 6-47 IC_COMP_PARAM_1 Register Fields (Continued)

Bits	Name	R/W	Description
7	<i>ADD_ENCODED_PARAMS</i>	R	<p>The value of this register is derived from the IC_ADD_ENCODED_PARAMS coreConsultant parameter. For a description of this parameter, see Table 4-1 on page 77. 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.</p> <ul style="list-style-type: none"> 0: False 1: True
6	<i>HAS_DMA</i>	R	<p>The value of this register is derived from the IC_HAS_DMA coreConsultant parameter. For a description of this parameter, see Table 4-1 on page 77.</p> <ul style="list-style-type: none"> 0: False 1: True
5	<i>INTR_IO</i>	R	<p>The value of this register is derived from the IC_INTR_IO coreConsultant parameter. For a description of this parameter, see Table 4-1 on page 77.</p> <ul style="list-style-type: none"> 0: Individual 1: Combined
4	<i>HC_COUNT_VALUES</i>	R	<p>The value of this register is derived from the IC_HC_COUNT_VALUES coreConsultant parameter. For a description of this parameter, see Table 4-1 on page 77.</p> <ul style="list-style-type: none"> 0: False 1: True
3:2	<i>MAX_SPEED_MODE</i>	R	<p>The value of this register is derived from the IC_MAX_SPEED_MODE coreConsultant parameter. For a description of this parameter, see Table 4-1 on page 77.</p> <ul style="list-style-type: none"> 0x0 = Reserved 0x1 = Standard 0x2 = Fast 0x3 = High
1:0	<i>APB_DATA_WIDTH</i>	R	<p>The value of this register is derived from the APB_DATA_WIDTH coreConsultant parameter. For a description of this parameter, see Table 4-1 on page 77.</p> <ul style="list-style-type: none"> 0x0 = 8 bits 0x1 = 16 bits 0x2 = 32 bits 0x3 = Reserved

6.3.45 IC_COMP_VERSION

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

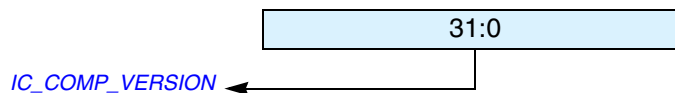


Table 6-48 IC_COMP_VERSION Register Fields

Bits	Name	R/W	Description
31:0	IC_COMP_VERSION	R	Specific values for this register are described in the Releases Table in the AMBA 2 release notes

6.3.46 IC_COMP_TYPE

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

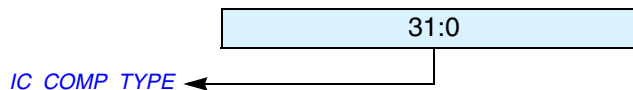


Table 6-49 IC_COMP_TYPE Register Fields

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.

7.1 Software Registers

For information about programming the software registers in terms of DW_apb_i2c operation, refer to [“Slave Mode Operation”](#) on page 50 and [“Master Mode Operation”](#) on page 53. The software registers are described in more detail in [“Registers”](#) on page 105.

7.2 Software Drivers

The family of DesignWare 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:

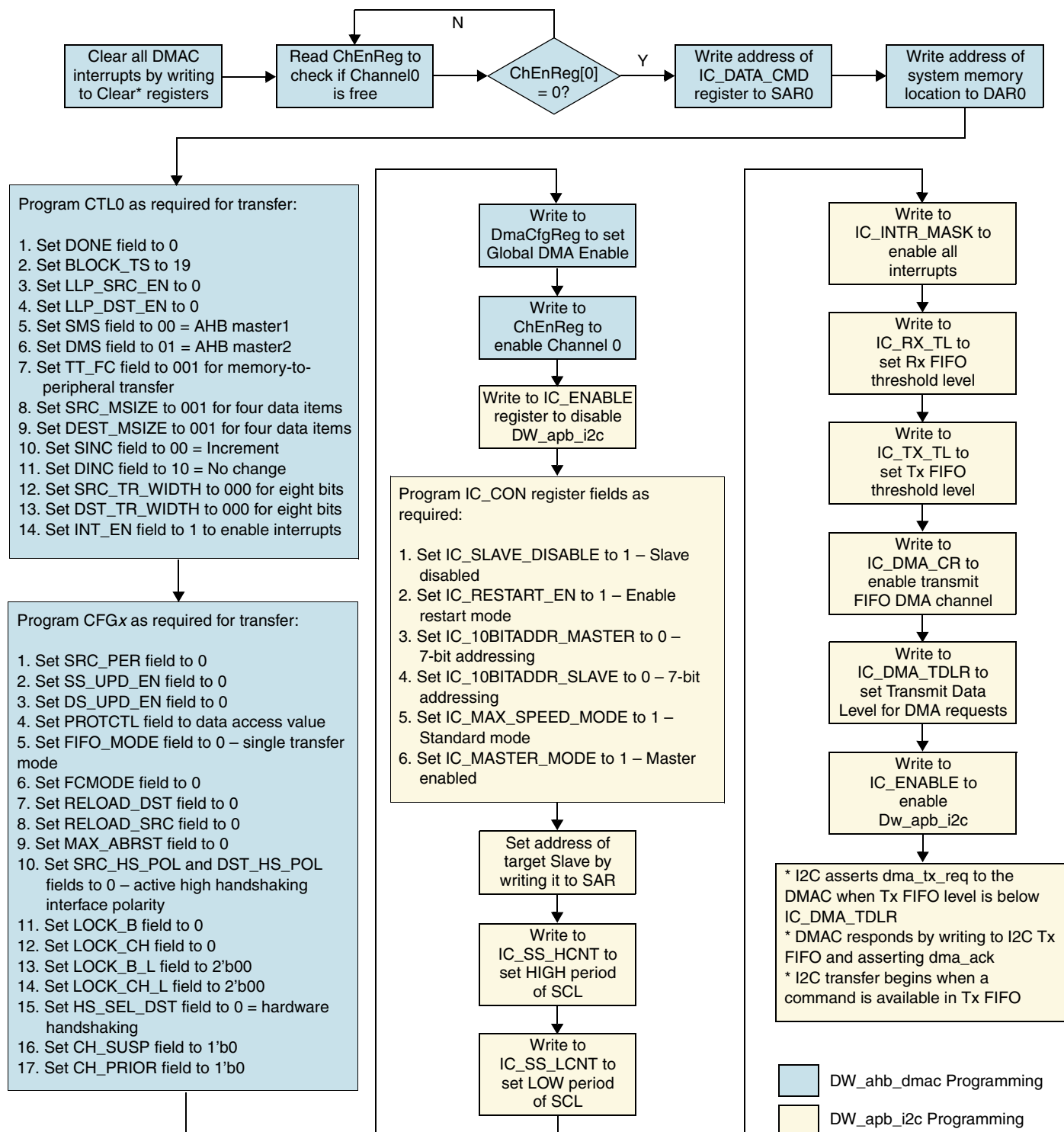
- Proven method of access to DW_apb_i2c minimizing usage errors
- Rapid software development with minimum overhead
- Detailed knowledge of DW_apb_i2c register bit fields not required
- Easy integration of DW_apb_i2c into existing software system
- Programming at register level eliminated

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.

7.3 Programming Example

The flow diagram in Figure 7-1 shows an overview of programming the DW_apb_i2c.

Figure 7-1 Flowchart for DW_ahb_dmac and DW_apb_i2c Programming Example





When there is at least one entry in the DW_apb_i2c Rx FIFO, the DW_apb_i2c asserts `dma_single` to the DMAC. When the number of entries in the DW_apb_i2c Rx FIFO reaches `IC_DMA_RDLR`, the DW_apb_i2c asserts `dma_rx_req` to the DMAC. In this example, in order to read nineteen data items from the DW_apb_i2c Rx FIFO, the DMAC samples `dma_req` for three BURST transfers of four beats of size 1 byte each, and it samples `dma_single` for three SINGLE transfers of size 1 byte each.

The following outlines details regarding reads and writes to/from DW_apb_i2c masters/slaves and VIP master/slaves:

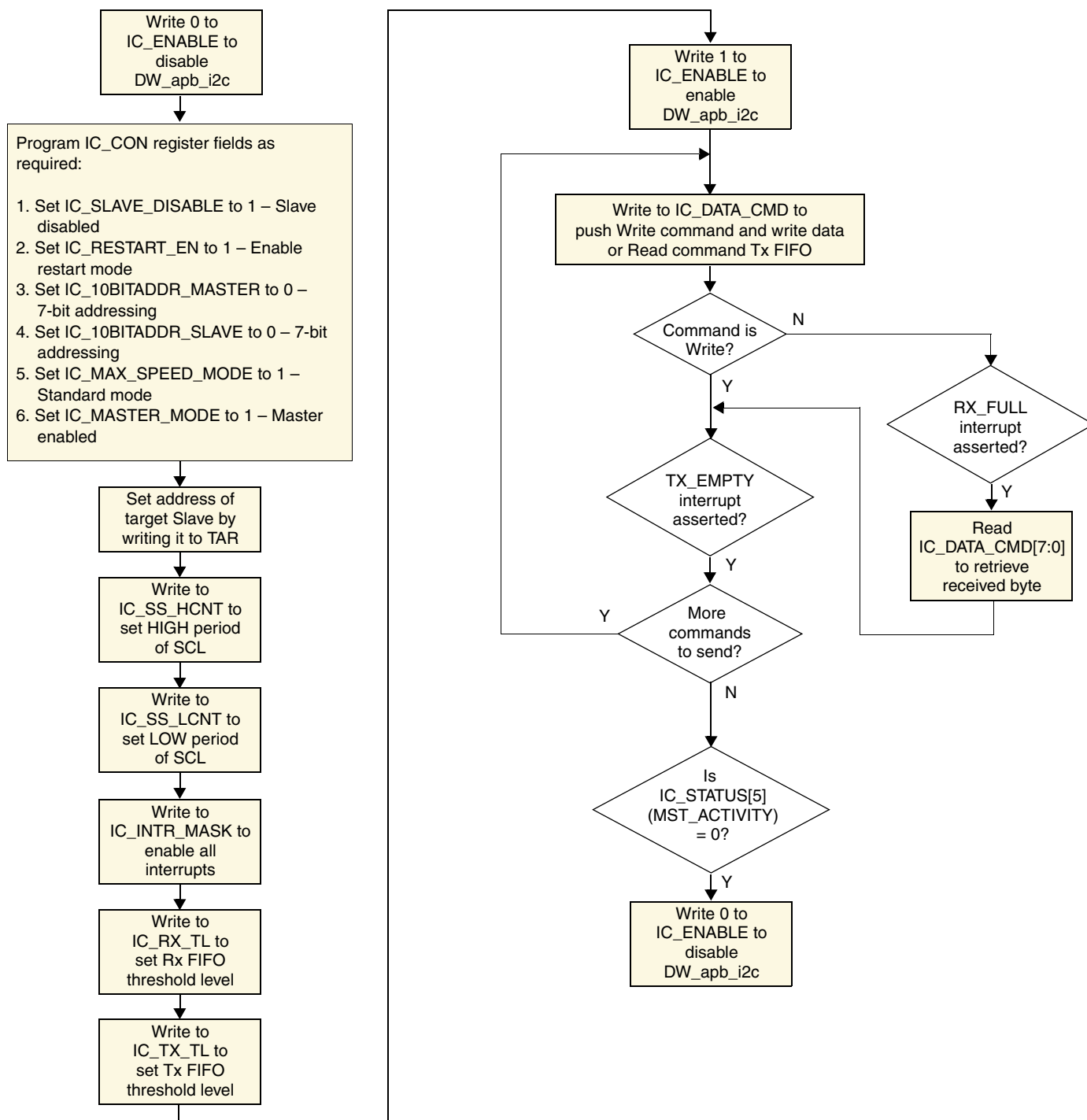
- For DW_apb_i2c master writes to a slave VIP model, bear in mind when using the DMA that you are writing characters from the byte stream. However, for a write, the DW_apb_i2c needs a halfword. To use the DMA, you should write software similar to the following:


```
short int temp_array[];
char * ptr=(char *) temp_array;
foreach byte in bytes {
    store byte ptr++;
    store '0x01' write command ptr++
}
```

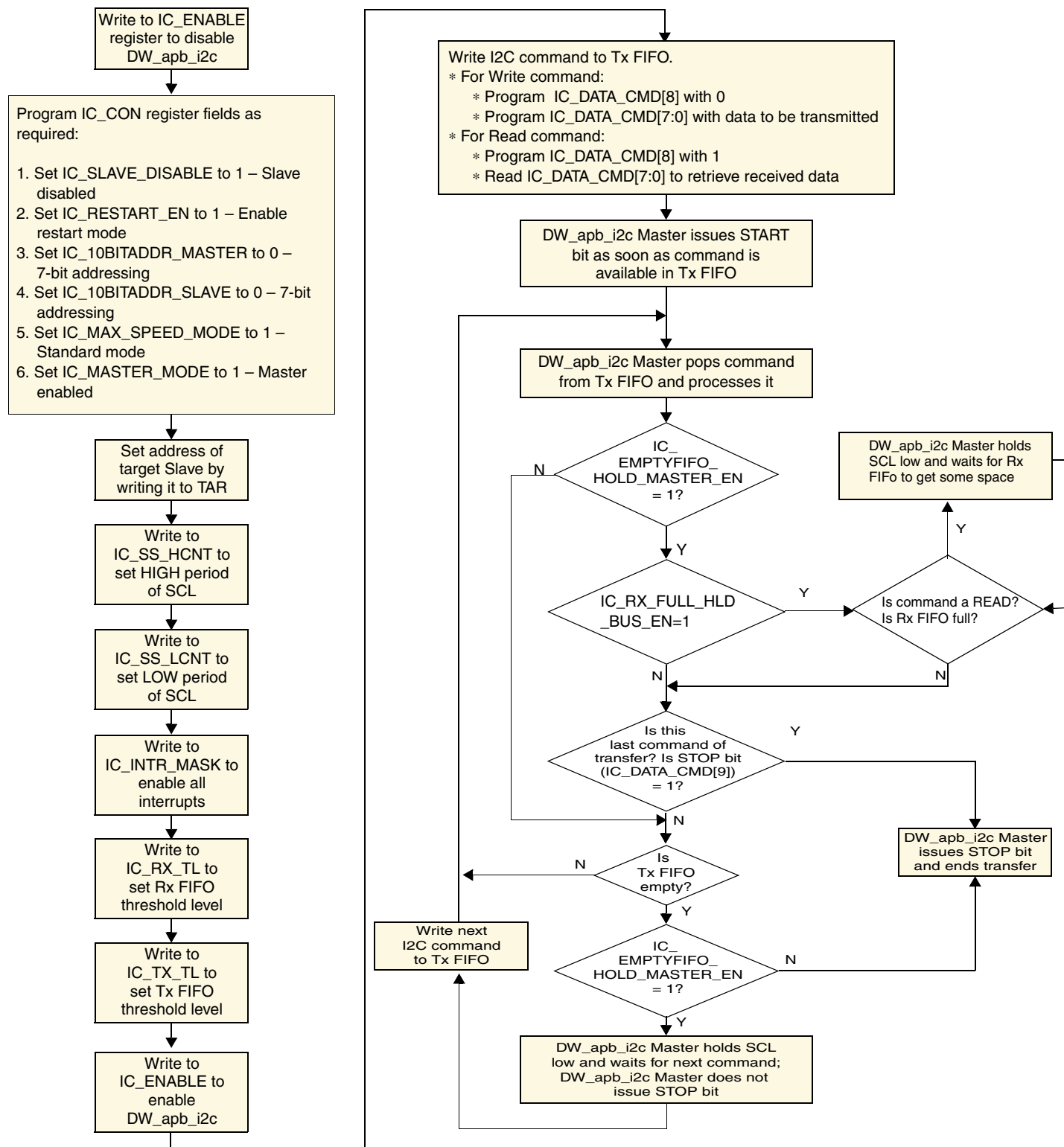
 - a. Program the DMA to read a stream of halfwords from memory and write them to the DW_apb_i2c using the hardware interface.
 - b. Program the DW_apb_i2c to do a write using the transmit DMA.
- For DW_apb_i2c master reads from a slave VIP model:
 - a. Create a read command halfword.
 - b. Program DMA channel 0 to do a fixed read of the read command halfword—that is, no address increment—to the DW_apb_i2c transmit buffer.
 - c. Program DMA channel 1 to read the data from the read buffer and store it in memory.
 - d. Program the DW_apb_i2c to do a master read by setting *both* DMA channels.
- For DW_apb_i2c slave writes from a master VIP model:
 - a. Program the DW_apb_i2c to be a slave with the RX buffer DMA enabled.
 - b. Program the DMA to read the buffer and store the bytes in memory.
- For DW_apb_i2c slave reads from a master VIP model:
 - a. Enable `IC_INTR_MASK.RD_REQ`; otherwise the DW_apb_i2c will not acknowledge the read.
 - b. When you get the `RD_REQ` interrupt, program the DMA to write the TX buffer with the read data.
 - c. Program the DW_apb_i2c to enable the TX DMA.

The flow diagram in [Figure 7-2](#) shows a programming example for the DW_apb_i2c Master.

Figure 7-2 Flowchart for DW_apb_i2c Master



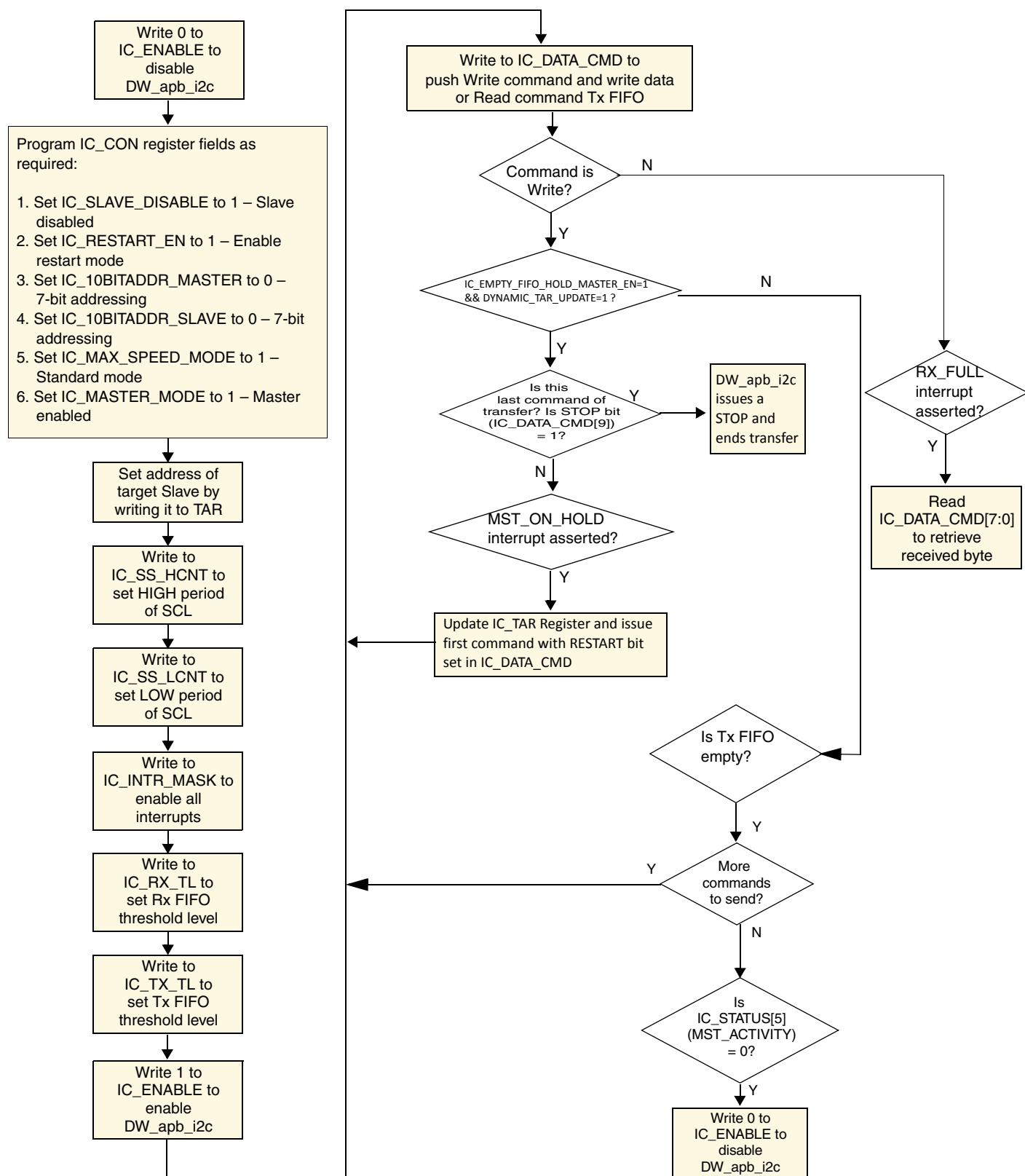
The flow diagram in [Figure 7-3](#) shows a programming example for the DW_apb_i2c master in standard mode, fast mode, or fast mode plus with 7-bit addressing.

Figure 7-3 Flowchart for DW_apb_i2c Master in Standard Mode, Fast Mode, or Fast Mode Plus

The flow diagram in [Figure 7-4](#) shows a programming example for DW_apb_i2c as master with TAR address update. This flow shows how the MST_ON_HOLD interrupt is used when the software needs information from the hardware to safely update the TAR address.

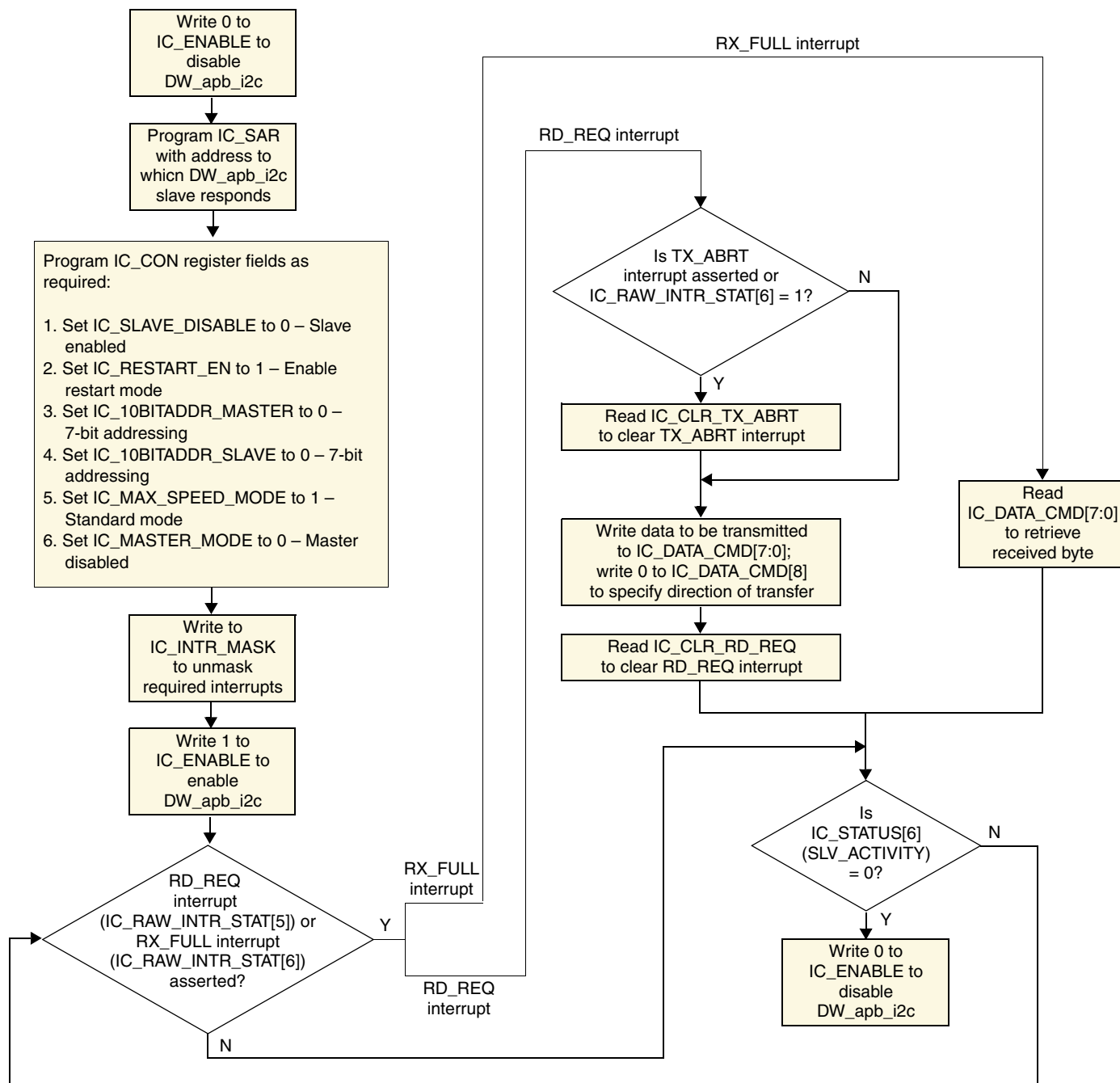


When the software has full knowledge of when it is safe to update the TAR address without requiring information from hardware, the MST_ON_HOLD interrupt is not required to update the TAR address.

Figure 7-4 Flowchart for DW_apb_i2c Master with TAR Address Update

The flow diagram in Figure 7-5 shows a programming example for the DW_apb_i2c Slave in standard mode, fast mode, or fast mode plus with 7-bit addressing.

Figure 7-5 Flowchart for DW_apb_i2c Slave in Standard Mode, Fast Mode, or Fast Mode Plus with 7-bit Addressing



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.

**Note**

The DW_apb_i2c verification testbench is built with DesignWare Verification IP (VIP). Please make sure you have the supported version of the VIP components for this release, otherwise, you may experience some tool compatibility problems. For more information about supported tools in this release, refer to the following web page:

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

8.1 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.

8.1.1 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.

8.1.2 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

8.1.3 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)

8.1.4 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.

8.1.5 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.

8.1.6 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.

8.1.7 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.

8.1.8 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.

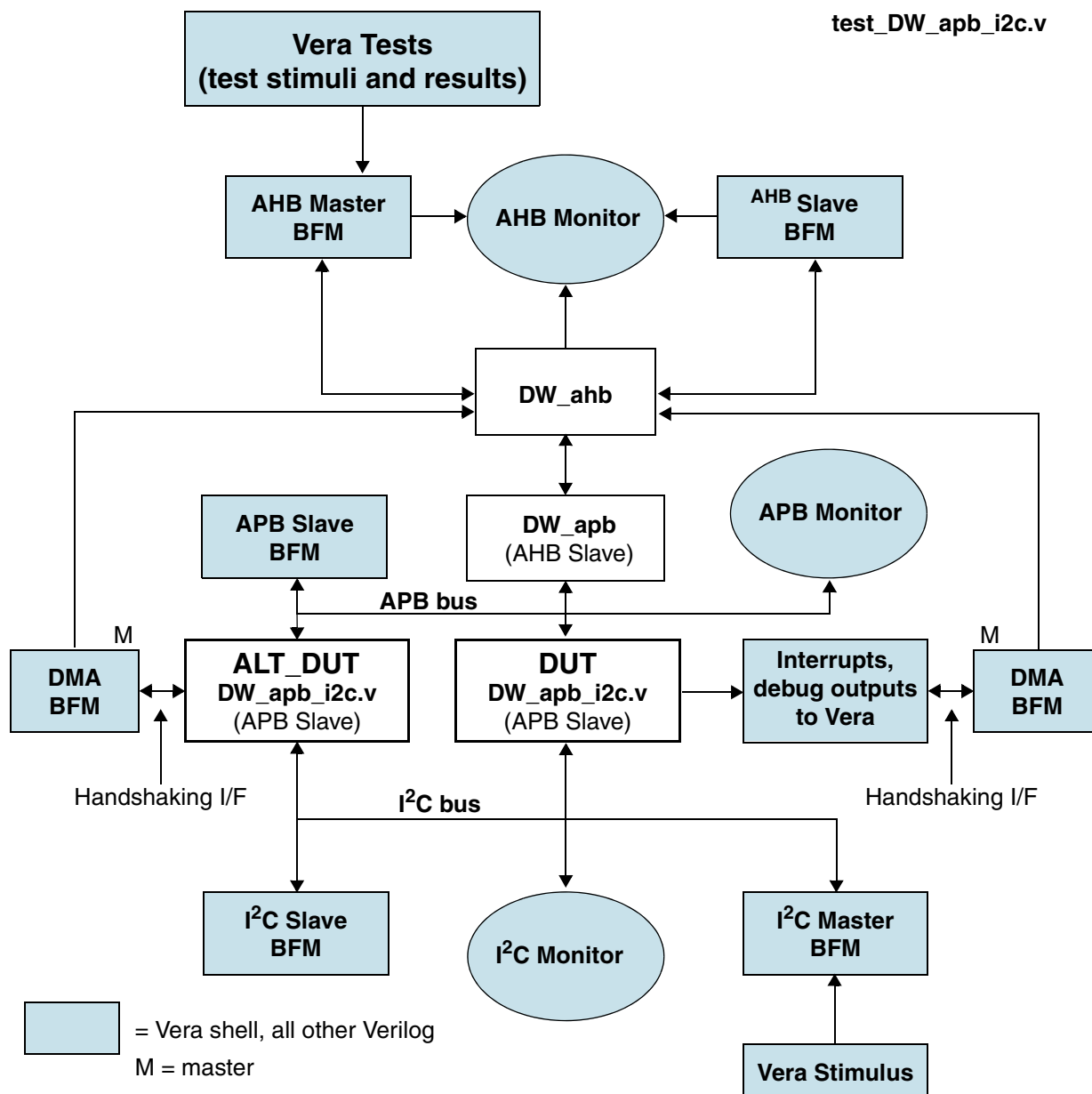
8.1.9 Generate ACK/NACK for General Call

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

8.2 Overview of DW_apb_i2c Testbench

As illustrated in [Figure 8-1](#), 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.

Figure 8-1 DW apb i2c Testbench



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.

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.

9.1 Reading and Writing from an APB Slave

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

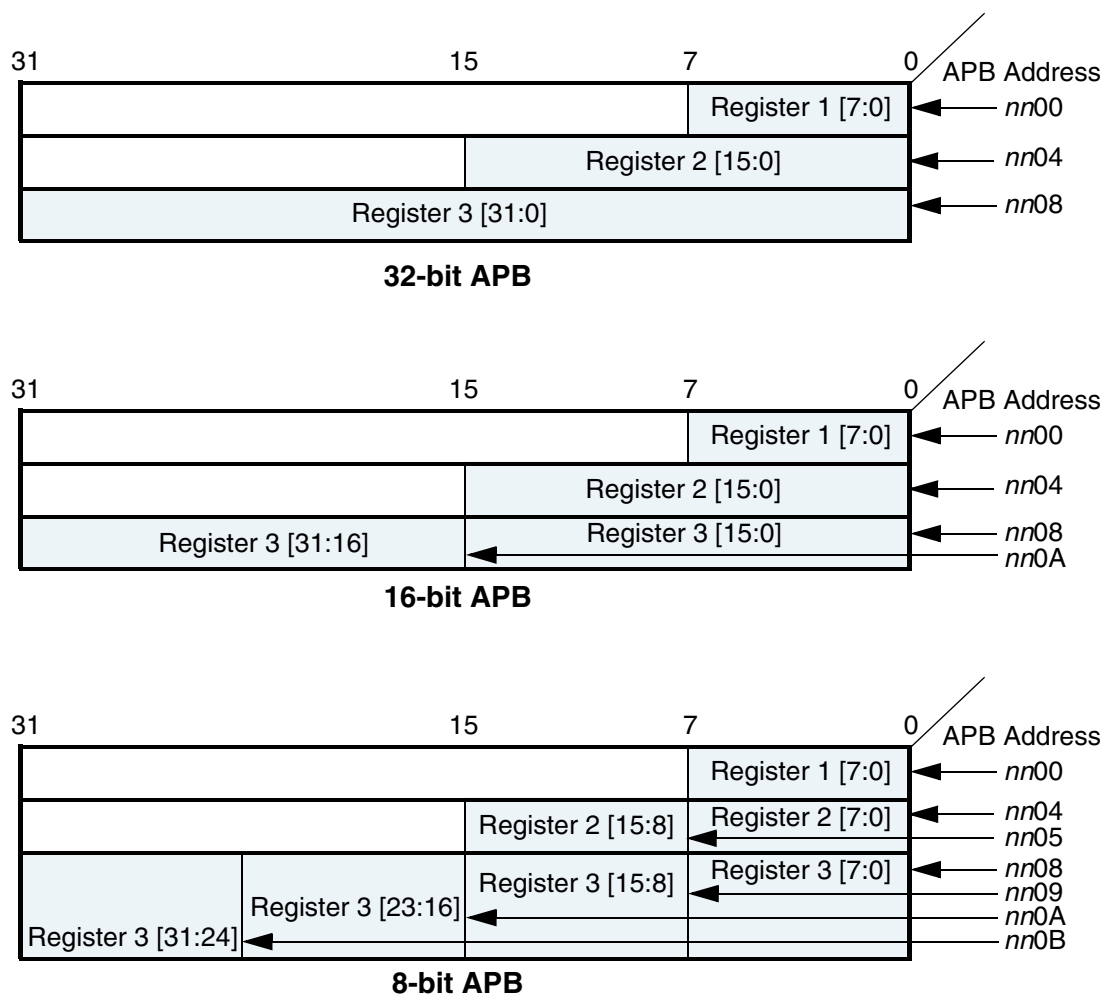
- The size of the APB peripheral should always be set equal to the size of the APB data bus, if possible.
- The APB bus has no concept of a transfer size or a byte lane, unlike the DW_ahb.
- The APB slave subsystem is little endian; the DW_apb performs the conversion from a big-endian AHB to the little-endian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

9.1.1 Reading From Unused Locations

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

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

Figure 9-1 Read/Write Locations for Different APB Bus Data Widths



9.1.2 32-bit Bus System

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

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



Note

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

9.1.3 16-bit Bus System

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

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

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

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

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

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

**Note**

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

9.1.4 8-bit Bus System

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

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

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

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

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

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

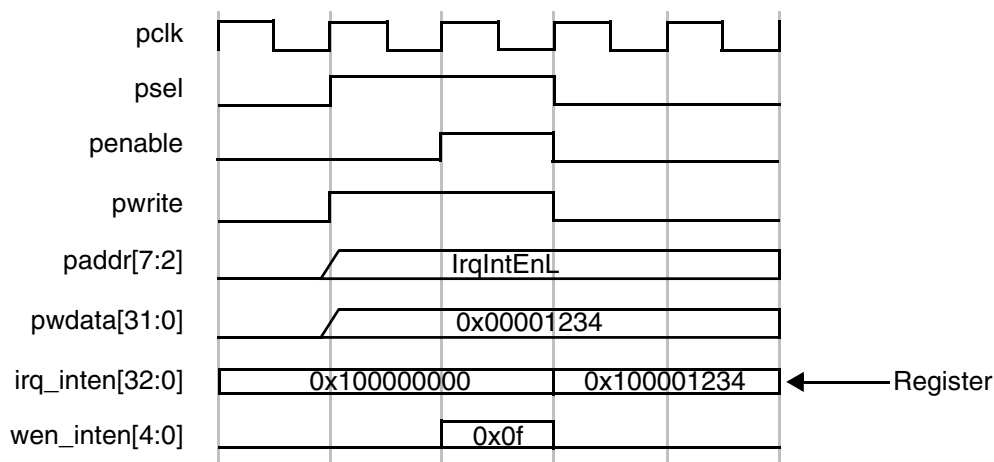
In this case, four AHB transactions are required, which in turn creates four APB transactions, to read or write the register. The first transaction should read/write the lower byte and the second transaction the second byte, and so on.

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

9.2 Write Timing Operation

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

Figure 9-2 APB Write Transaction



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

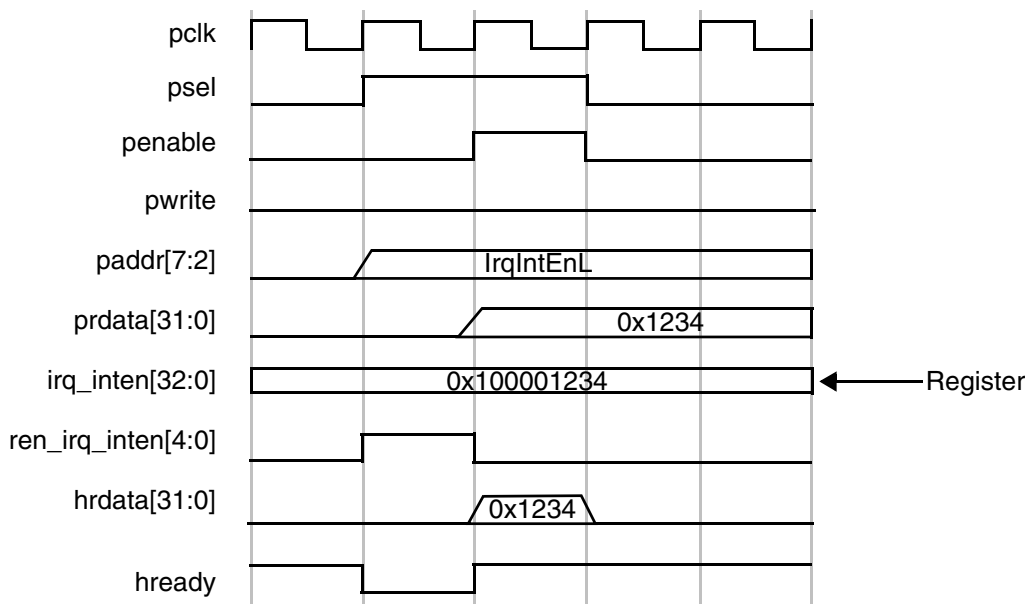
A write from the AHB to the APB does not require the AHB system bus to stall until the transfer on the APB has completed. A write to the APB can be followed by a read transaction from another AHB peripheral (not the DW_apb).

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

9.3 Read Timing Operation

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

Figure 9-3 APB Read Transaction



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

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

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



Note

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

9.4 Accessing Top-level Constraints

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

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

```
set_activity_parameter Synthesize ScriptsOnly 1
```

2. This cC command autocompletes the activity:

```
autocomplete_activity Synthesize
```

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

```
write_sdc <filename>
```

9.5 Performing

This section discusses performance and the hardware configuration parameters that affect the performance of the DW_apb_i2c.

9.5.1 Area

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

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

Table 9-1 Synthesis Results Using TSMC 65nmTechnology Library

Configuration	Operating Frequency	Gate Count
Default Configuration	166 MHz	11419 gates
Minimum Configuration: IC_CLK_TYPE=0 IC_MAX_SPEED_MODE=1 IC_10BITADDR_MASTER=0 IC_10BITADDR_SLAVE=0 IC_MASTER_MODE=0 IC_TX_BUFFER_DEPTH=2 IC_RX_BUFFER_DEPTH=2 IC_HC_COUNT_VALUES=1	166 MHz	6253 gates
Maximum Configuration: IC_CLK_TYPE=1 APB_DATA_WIDTH=32 IC_TX_BUFFER_DEPTH=16 IC_RX_BUFFER_DEPTH=16	166 MHz	12768 gates

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

Table 9-2 Synthesis Results Using TSMC 28nm Technology Library

Configuration	Operating Frequency	Gate Count
Default Configuration	166 MHz	11164 gates
Minimum Configuration: IC_CLK_TYPE=0 IC_MAX_SPEED_MODE=1 IC_10BITADDR_MASTER=0 IC_10BITADDR_SLAVE=0 IC_MASTER_MODE=0 IC_TX_BUFFER_DEPTH=2 IC_RX_BUFFER_DEPTH=2 IC_HC_COUNT_VALUES=1	166 MHz	6151 gates
Maximum Configuration: IC_CLK_TYPE=1 APB_DATA_WIDTH=32 IC_TX_BUFFER_DEPTH=16 IC_RX_BUFFER_DEPTH=16	166 MHz	12659 gates

9.5.2 Power Consumption

The following table provides information about the power consumption of the DW_apb_i2c using the TSMC 65nm technology library and how it affects performance.

Table 9-3 Power Consumption of DW_apb_i2c Using TSMC 65nm Technology Library

Configuration	Operating Frequency	Static Power Consumption	Dynamic Power Consumption
Default Configuration	166 MHz	3.0260 μ W	1.2895 mW
Minimum Configuration: IC_CLK_TYPE=0 IC_MAX_SPEED_MODE=1 IC_10BITADDR_MASTER=0 IC_10BITADDR_SLAVE=0 IC_MASTER_MODE=0 IC_TX_BUFFER_DEPTH=2 IC_RX_BUFFER_DEPTH=2 IC_HC_COUNT_VALUES=1	166 MHz	1.6878 μ W	671.0817 μ W
Maximum Configuration: IC_CLK_TYPE=1 APB_DATA_WIDTH=32 IC_TX_BUFFER_DEPTH=16 IC_RX_BUFFER_DEPTH=16	166 MHz	3.3545 μ W	1.4755 mW

The following table provides information about the power consumption of the DW_apb_i2c using the TSMC 28nm technology library and how it affects performance.

Table 9-4 Power Consumption of DW_apb_i2c Using TSMC 28nm Technology Library

Configuration	Operating Frequency	Static Power Consumption	Dynamic Power Consumption
Default Configuration	166 MHz	1.1252 mW	877.4711 μ W
Minimum Configuration: IC_CLK_TYPE=0 IC_MAX_SPEED_MODE=1 IC_10BITADDR_MASTER=0 IC_10BITADDR_SLAVE=0 IC_MASTER_MODE=0 IC_TX_BUFFER_DEPTH=2 IC_RX_BUFFER_DEPTH=2 IC_HC_COUNT_VALUES=1	166 MHz	625.7431 μ W	452.3128 μ W
Maximum Configuration: IC_CLK_TYPE=1 APB_DATA_WIDTH=32 IC_TX_BUFFER_DEPTH=16 IC_RX_BUFFER_DEPTH=16	166 MHz	1.2889 mW	999.9554 μ W

A

Glossary

active command queue	Command queue from which a model is currently taking commands; see also command queue.
activity	A set of functions in coreConsultant that step you through configuration, verification, and synthesis of a selected core.
AHB	Advanced High-performance Bus — high-performance system backbone bus. AHB supports the efficient connection of processors, on-chip memories and off-chip external memory interfaces (ARM Limited specification).
AMBA	Advanced Microcontroller Bus Architecture — a trademarked name by ARM Limited that defines an on-chip communication standard for high speed microcontrollers.
APB	Advanced Peripheral Bus — optimized for minimal power consumption and reduced interface complexity to support peripheral functions (ARM Limited specification).
APB bridge	DW_apb submodule that converts protocol between the AHB bus and APB bus.
application design	Overall chip-level design into which a subsystem or subsystems are integrated.
arbiter	AMBA bus submodule that arbitrates bus activity between masters and slaves.
BFM	Bus-Functional Model — A simulation model used for early hardware debug. A BFM simulates the bus cycles of a device and models device pins, as well as certain on-chip functions. See also Full-Functional Model.
big-endian	Data format in which most significant byte comes first; normal order of bytes in a word.
blocked command stream	A command stream that is blocked due to a blocking command issued to that stream; see also command stream, blocking command, and non-blocking command.

blocking command	A command that prevents a testbench from advancing to next testbench statement until this command executes in model. Blocking commands typically return data to the testbench from the model.
bus bridge	Logic that handles the interface and transactions between two bus standards, such as AHB and APB. See APB bridge.
command channel	Manages command streams. Models with multiple command channels execute command streams independently of each other to provide full-duplex mode function.
command stream	The communication channel between the testbench and the model.
component	A generic term that can refer to any synthesizable IP or verification IP in the DesignWare Library. In the context of synthesizable IP, this is a configurable block that can be instantiated as a single entity (VHDL) or module (Verilog) in a design.
configuration	The act of specifying parameters for a core prior to synthesis; can also be used in the context of VIP.
configuration intent	Range of values allowed for each parameter associated with a reusable core.
core	Any configurable block of synthesizable IP that can be instantiated as a single entity (VHDL) or module (Verilog) in a design. Core is the preferred term for a big piece of IIP. Anything that requires coreConsultant for configuration, as well as anything in the DesignWare Cores library, is a core.
core developer	Person or company who creates or packages a reusable core. All the cores in the DesignWare Library are developed by Synopsys.
core integrator	Person who uses coreConsultant or coreAssembler to incorporate reusable cores into a system-level design.
coreAssembler	Synopsys product that enables automatic connection of a group of cores into a subsystem. Generates RTL and gate-level views of the entire subsystem.
coreConsultant	A Synopsys product that lets you configure a core and generate the design views and synthesis views you need to integrate the core into your design. Can also synthesize the core and run the unit-level testbench supplied with the core.
coreKit	An unconfigured core and associated files, including the core itself, a specified synthesis methodology, interfaces definitions, and optional items such as verification environment files and core-specific documentation.
cycle command	A command that executes and causes HDL simulation time to advance.
decoder	Software or hardware subsystem that translates from and “encoded” format back to standard format.
design context	Aspects of a component or subsystem target environment that affect the synthesis of the component or subsystem.
design creation	The process of capturing a design as parameterized RTL.

Design View	A simulation model for a core generated by coreConsultant.
DesignWare Synthesizable Components	The Synopsys name for the collection of AMBA-compliant coreKits and verification models delivered with DesignWare and used with coreConsultant or coreAssembler to quickly build DesignWare Synthesizable Component designs.
DesignWare cores	A specific collection of synthesizable cores that are licensed individually. For more information, refer to www.synopsys.com/designware .
DesignWare Library	A collection of synthesizable IP and verification IP components that is authorized by a single DesignWare license. Products include SmartModels, VMT model suites, DesignWare Memory Models, Building Block IP, and the DesignWare Synthesizable Components.
dual role device	Device having the capabilities of function and host (limited).
endian	Ordering of bytes in a multi-byte word; see also little-endian and big-endian.
Full-Functional Mode	A simulation model that describes the complete range of device behavior, including code execution. See also BFM.
GPIO	General Purpose Input Output.
GTECH	A generic technology view used for RTL simulation of encrypted source code by non-Synopsys simulators.
hard IP	Non-synthesizable implementation IP.
HDL	Hardware Description Language – examples include Verilog and VHDL.
IIP	Implementation Intellectual Property — A generic term for synthesizable HDL and non-synthesizable “hard” IP in all of its forms (coreKit, component, core, MacroCell, and so on).
implementation view	The RTL for a core. You can simulate, synthesize, and implement this view of a core in a real chip.
instantiate	The act of placing a core or model into a design.
interface	Set of ports and parameters that defines a connection point to a component.
IP	Intellectual property — A term that encompasses simulation models and synthesizable blocks of HDL code.
little-endian	Data format in which the least-significant byte comes first.
MacroCell	Bigger IP blocks (6811, 8051, memory controller) available in the DesignWare Library and delivered with coreConsultant.
master	Device or model that initiates and controls another device or peripheral.
model	A Verification IP component or a Design View of a core.
monitor	A device or model that gathers performance statistics of a system.

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

Index

A

active command queue
definition 193

activity
definition 193

AHB
definition 193

AMBA
definition 193

APB
definition 193

APB bridge
definition 193

APB Interface, and DW_apb_i2c 76

APB_DATA_WIDTH 77

application design
definition 193

arbiter
definition 193

Arbitration, of master 47

B

BFM
definition 193

big-endian
definition 193

Block diagram, of DW_apb_i2c 17

blocked command stream
definition 193

blocking command
definition 194

bus bridge
definition 194

C

Clock synchronization 49

command channel
definition 194

command stream
definition 194

component
definition 194

Configuration
of IC_CLK frequency 59

configuration
definition 194

configuration intent
definition 194

Configuration parameters 77

core
definition 194

core developer
definition 194

core integrator
definition 194

coreAssembler
definition 194
overview of usage flow 25

coreConsultant
definition 194
overview of usage flow 22

coreKit
definition 194

Customer Support 8

cycle command
definition 194

D

debug_addr 103

debug_addr_10bit 103

debug_data 103

debug_hs 104

debug_master_act 104

debug_mst_cstate 104

debug_p_gen 103

- debug_rd [103](#)
- debug_s_gen [102](#)
- debug_slave_act [104](#)
- debug_slv_cstate [104](#)
- debug_wr [103](#)
- decoder
 - definition [194](#)
- design context
 - definition [194](#)
- design creation
 - definition [194](#)
- Design View
 - definition [195](#)
- DesignWare cores
 - definition [195](#)
- DesignWare Library
 - definition [195](#)
- DesignWare Synthesizable Components
 - definition [195](#)
- Disabling DW_apb_i2c
 - version 1.06a [56](#)
- DMA Controller
 - and DW_apb_i2c [67](#)
- dma_rx_ack [102](#)
- dma_rx_req [101](#)
- dma_rx_single [102](#)
- dma_tx_ack [102](#)
- dma_tx_req [101](#)
- dma_tx_single [102](#)
- dual role device
 - definition [195](#)
- DW_apb
 - slaves
 - read timing operation [189](#)
 - write timing operation [188](#)
- DW_apb_i2c
 - block diagram of [17](#)
 - functional behavior [33](#)
 - functional overview [17](#)
 - I/O description [91](#)
 - memory map of [106](#)
 - operation modes [49](#)
 - overview of [33](#)
 - parameters [77](#)
 - programming of [105](#)
 - protocols [38](#)

- registers [112](#)
 - testbench
 - overview of [182](#)
 - overview of tests [179](#)
- Dynamic update of IC_TAR
 - initial configuration of master mode [53](#)
 - or 10-bit addressing for master mode [55](#)

E

- endian
 - definition [195](#)
- Environment, licenses [19](#)

F

- Full-Functional Mode
 - definition [195](#)
- Functional behavior, of DW_apb_i2c [33](#)
- Functional overview, of DW_apb_i2c [17](#)

G

- GPIO
 - definition [195](#)
- GTECH
 - definition [195](#)

H

- hard IP
 - definition [195](#)
- HDL
 - definition [195](#)

I

- I/O connections [92](#)
- I/O signals, description of [91](#)
- IC_10BITADDR_MASTER [79](#)
- IC_ACK_GENERAL_CALL [162](#)
- ic_activity_intr(_n) [99](#)
- IC_ADD_ENCODED_PARAMS [82](#)
- IC_CAP_LOADING [85](#)
- ic_clk [94](#)
- IC_CLK frequency, configuration of [59](#)
- ic_clk_in_a [95](#)
- ic_clk_oe [95](#)
- IC_CLOCK_FREQ [83](#)
- IC_CLR_ACTIVITY [142](#)
- IC_CLR_GEN_CALL [144](#)
- IC_CLR_INTR [139](#)
- IC_CLR_RD_REQ [141](#)

[IC_CLR_RX_DONE](#) [142](#)
[IC_CLR_RX_OVER](#) [140](#)
[IC_CLR_RX_UNDER](#) [139](#)
[IC_CLR_START_DET](#) [143](#)
[IC_CLR_STOP_DET](#) [143](#)
[IC_CLR_TX_ABRT](#) [141](#)
[IC_CLR_TX_OVER](#) [140](#)
[IC_CON](#) [112](#)
[ic_current_src_en](#) [96](#)
[IC_DATA_CMD](#) [121](#)
[ic_data_in_a](#) [95](#)
[ic_data_oe](#) [95](#)
[IC_DEFAULT_ACK_GENERAL_CALL](#) [81](#)
[IC_DEFAULT_FS_SPKLEN](#) [86](#)
[IC_DEFAULT_HS_SPKLEN](#) [86](#)
[IC_DEFAULT_SDA_HOLD](#) [81](#)
[IC_DEFAULT_SDA_SETUP](#) [81](#)
[IC_DEFAULT_SLAVE_ADDR](#) [78](#)
[IC_DMA_CR](#) [158](#)
[IC_DMA_RDLR](#) [160](#)
[IC_DMA_TDRL](#) [159](#)
[ic_en](#) [96](#)
[IC_ENABLE](#) [145](#)
[IC_ENABLE_STATUS](#) [163](#)
[IC_FS_SCL_HCNT](#) [126](#)
[IC_FS_SCL_HIGH_COUNT](#) [84](#)
[IC_FS_SCL_LCNT](#) [127](#)
[IC_FS_SCL_LOW_COUNT](#) [85](#)
[ic_gen_call_intr\(_n\)](#) [100](#)
[IC_HC_COUNT_VALUES](#) [83](#)
[IC_HS_MADDR](#) [120](#)
[IC_HS_MASTER_CODE](#) [78](#)
[IC_HS_SCL_HCNT](#) [128](#)
[IC_HS_SCL_HIGH_COUNT](#) [85](#)
[IC_HS_SCL_LCNT](#) [129](#)
[IC_HS_SCL_LOW_COUNT](#) [79, 85](#)
[IC_INTR_IO](#) [81](#)
[IC_INTR_MASK](#) [131](#)
[IC_INTR_POL](#) [82](#)
[IC_INTR_STAT](#) [130](#)
[ic_intr\(_n\)](#) [96](#)
[IC_MASTER_MODE](#) [79](#)
[IC_MAX_SPEED_MODE](#) [78](#)
[IC_RAW_INTR_STAT](#) [133](#)
[ic_rd_req_intr\(_n\)](#) [98](#)
[IC_RESTART_EN](#) [80](#)
[ic_rst_n](#) [95](#)
[IC_RX_BUFFER_DEPTH](#) [80](#)
[ic_rx_done_intr\(_n\)](#) [97, 99](#)
[ic_rx_over_intr\(_n\)](#) [97](#)
[IC_RX_TL](#) [80, 137](#)
[ic_rx_under_intr\(_n\)](#) [97](#)
[IC_RXFLR](#) [151](#)
[IC_SAR](#) [119](#)
[IC_SDA_HOLD](#) [152](#)
[IC_SDA_SETUP](#) [161](#)
[IC_SLV_DATA_NACK_ONLY](#) [157](#)
[IC_SS_CDNT](#) [125](#)
[IC_SS_HCNT](#) [124](#)
[IC_SS_SCL_HIGH_COUNT](#) [84](#)
[IC_SS_SCL_LOW_COUNT](#) [84](#)
[ic_start_det_intr\(_n\)](#) [100](#)
[IC_STATUS](#) [147](#)
[ic_stop_det_intr\(_n\)](#) [99](#)
[IC_TAR](#) [117](#)
[ic_tx_abrt_intr\(_n\)](#) [99](#)
[IC_TX_ABRT_SOURCE](#) [153](#)
[IC_TX_BUFFER_DEPTH](#) [80](#)
[ic_tx_ecmplt_intr\(_n\)](#) [98](#)
[ic_tx_over_intr\(_n\)](#) [98](#)
[IC_TX_TL](#) [80, 138](#)
[IC_TXFLR](#) [150](#)
[IC_USE_COUNTS](#) [82](#)
[IIP](#)
 [definition](#) [195](#)
 [implementation view](#)
 [definition](#) [195](#)
 [instantiate](#)
 [definition](#) [195](#)
 [interface](#)
 [definition](#) [195](#)
 [Interfaces](#)
 [APB](#) [76](#)
 [DMA Controller](#) [67](#)
 [IP](#)
 [definition](#) [195](#)

L

Licenses [19](#)

little-endian
definition [195](#)

M

MacroCell
definition [195](#)

master
definition [195](#)

Master arbitration [47](#)

Master mode [53](#)

Memory map, of DW_apb_i2c [106](#)

model
definition [195](#)

monitor
definition [195](#)

N

non-blocking command
definition [196](#)

O

Operation modes [49](#)

Output files
GTECH [30](#)
RTL-level [29](#)
Simulation model [30](#)
synthesis [30](#)
verification [31](#)

P

paddr [94](#)

Parameters
description of [77](#)

pclk [93](#)

penable [93](#)

peripheral
definition [196](#)

prdata [94](#)

presetn [93](#)

Programming DW_apb_i2c
memory map [105](#)
registers [112](#)

Protocols, of I²C [38](#)

psel [93](#)

pwdata [94](#)

pwrite [94](#)

R

Reading, from unused locations [185](#)

Register
IC_HS_MADDR [120](#)

Registers

Clear ACTIVITY Interrupt [142](#)
Clear Combined and Individual Interrupts [139](#)
Clear GEN_CALL Interrupt [144](#)
Clear RD_REQ Interrupt [141](#)
Clear RX_DONE Interrupt [142](#)
Clear RX_OVER Interrupt [140](#)
Clear RX_UNDER Interrupt [139](#)
Clear START_DET Interrupt [143](#)
Clear STOP_DET Interrupt [143](#)
Clear TX_ABRT Interrupt [141](#)
Clear TX_OVER Interrupt [140](#)
Control [112](#)
DMA Control [158](#)
DMA Transmit Data Level [159](#)
Enable Status [163](#)
Fast Speed I2C Clock SCL High Count [126](#)
Fast Speed I2C Clock SCL Low Count [127](#)
Generate Slave Data NACK [157](#)
High Speed I2C Clock SCL High Count [128](#)
High Speed I2C Clock SCL Low Count [129](#)
HS Master Mode Code Address [120](#)
HS Spike Suppression Limit [166](#)
I2C Enable [145](#)
I2C Receive Data Level [160](#)
IC_ACK_GENERAL_CALL [162](#)
IC_CLR_ACTIVITY [142](#)
IC_CLR_GEN_CALL [144](#)
IC_CLR_INTR [139](#)
IC_CLR_RD_REQ [141](#)
IC_CLR_RX_DONE [142](#)
IC_CLR_RX_OVER [140](#)
IC_CLR_RX_UNDER [139](#)
IC_CLR_START_DET [143](#)
IC_CLR_STOP_DET [143](#)
IC_CLR_TX_ABRT [141](#)
IC_CLR_TX_OVER [140](#)
IC_CON [112](#)
IC_DATA_CMD [121](#)
IC_DMA_CR [158](#)
IC_DMA_RDLR [160](#)
IC_DMA_TDLR [159](#)
IC_ENABLE [145](#)
IC_ENABLE_STATUS [163](#)
IC_FS_SCL_HCNT [126](#)

- IC_FS_SCL_LCNT [127](#)
- IC_FS_SPKLENS [165](#)
- IC_HS_SCL_HCNT [128](#)
- IC_HS_SCL_LCNT [129](#)
- IC_HS_SPKLENS [166](#)
- IC_INTR_MASK [131](#)
- IC_INTR_STAT [130](#)
- IC_RAW_INTR_STAT [133](#)
- IC_RX_TL [137](#)
- IC_RXFLR [151](#)
- IC_SAR [119](#)
- IC_SDA_HOLD [152](#)
- IC_SDA_SETUP [161](#)
- IC_SLV_DATA_NACK_ONLY [157](#)
- IC_SS_HCNT [124](#)
- IC_SS_LCNT [125](#)
- IC_STATUS [147](#)
- IC_TAR [117](#)
- IC_TX_ABRT_SOURCE [153](#)
- IC_TX_TL [138](#)
- IC_TXFLR [150](#)
- Interrupt Mask [131](#)
- Interrupt Status [133](#)
- of DW_apb_i2c [112](#)
- Raw Interrupt Status [130](#)
- Receive Buffer Threshold [137](#)
- Rx/Tx Data buffer and Command [121](#)
- SDA Setup [161](#)
- Slave Address [119](#)
- SS/FS Spike Suppression Limit [165](#)
- Standard Speed I2C Clock SCL High Count [124](#)
- Standard Speed I2C Clock SCL Low Count [125](#)
- Target Address [117](#)
- Transmit Buffer Threshold [138](#)
- RTL
 - definition [196](#)
- S**
- SDRAM
 - definition [196](#)
- SDRAM controller
 - definition [196](#)
- Signals, description of [91](#)
- Simulation
 - of DW_apb_i2c [182](#)
- slave
 - definition [196](#)
- Slave mode [50](#)
- SoC
 - definition [196](#)
- SoC Platform
 - AHB contained in [15](#)
 - APB, contained in [15](#)
 - defined [15](#)
- soft IP
 - definition [196](#)
- SSI_HAS_DMA [81](#)
- static controller
 - definition [196](#)
- subsystem
 - definition [196](#)
- synthesis intent
 - definition [196](#)
- synthesizable IP
 - definition [196](#)
- T**
- technology-independent
 - definition [196](#)
- test_DW_apb_i2c.v [182](#)
- Testsuite Regression Environment (TRE)
 - definition [196](#)
- Timing
 - read operation of DW_apb slave [189](#)
 - write operation of DW_apb slave [188](#)
- TRE
 - definition [196](#)
- V**
- Vera, overview of tests [179](#)
- Verification
 - and Vera tests [179](#)
 - of DW_apb_i2c [182](#)
- VIP
 - definition [196](#)
- W**
- workspace
 - definition [196](#)
- wrap
 - definition [196](#)
- wrapper
 - definition [196](#)
- Z**
- zero-cycle command
 - definition [196](#)

